THE VLSI DESIGN OF A GENERAL PURPOSE FFT PROCESSING NODE

by

BRIAN CLIFFORD MCKINNEY
B.Sc., University of Victoria, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE March 27, 86

in the Department of
Electrical Engineering

We accept this thesis as conforming
to the required standard

[redacted]

Supervisor Dr. F. El Guibaly

[redacted]

Dr. J.C. Muzio

[redacted]

Dr. M. Clements

[redacted]

Dr. P. Agathoklis

[redacted]

Dr. V. K. Bhargava

UNIVERSITY OF VICTORIA

January 1986

Supervisor: Assistant Professor F. El Guibaly

ABSTRACT

This thesis presents the design of a general-purpose floating-point FFT processing node. The node is currently configured to operate in the Pipelined Cascade implementation of a radix-2 decimation in frequency algorithm. The implementation is capable of performing a 1024 point FFT in approximately 0.9 ms, or in 1.2 ms if on-chip twiddle factor updating is employed. The output signal-to-noise ratio for a 1024-point FFT is calculated to be better than 60 dB.

The general-purpose processing node is designed in a three-micron single metal ISO-CMOS technology. The node design occupies some 6400 X 6400 micron^2 and contains approximately 18,000 transistors. Power dissipation estimates are placed at 1.5 watts per processing node. The node is currently designed to handle a modified IEEE standard 32-bit floating-point format in which the 23-bit mantissa is truncated down to 14 bits in length. Analysis performed on the effects of finite register length on FFT calculation has indicated that in order to achieve a signal-to-noise ratio greater than 60 dB for a 1024-point FFT, a mantissa length of only 14 bits is required.

Central to the processing node operation is a fast floating-point arithmetic processing unit, or APU. The APU design is based on two multiple access pipelines or MAP structures operating concurrently. This architecture combines the flexibility of bus structures with a highly-concurrent pipeline processor to realize a fast highly-flexible processing unit. The APU is microprogram controlled, and is capable of performing the decimation-in-frequency butterfly operation in under 700 ns.

Processor node control is achieved by means of a master-slave control configuration in which the microprogrammed APU controller acts as slave to the node-level master controller. The node-level master controller is a sequential logic unit which is responsible for on-chip data routing, and for overseeing the proper sequencing of the APU operation.

The Pipelined Cascade implementation is a highly flexible architecture. This thesis shows that with slight modification of the present design, fast calculation of inverse and multidimensional FFT's are possible. The speed and flexibility of the processing node design and proposed network implementation are ideal for many task-specific problems requiring fast spectrum analysis.

Examiners: [redacted]

Dr. F. El Guibaly

[redacted]

Dr. J.C. Muzio

[redacted]

Dr. M. Clements

[redacted]

Dr. P. Agathoklis

[redacted]

Dr. V. K. Bhargava

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. F. El Guibaly for his guidance, during the course of this project and for his helpful advice during the preparation of this manuscript. I would also like to thank Mr. G. Csanyi-Fritz for his assistance and suggestions, in dealing with the Metheus workstation. Financial assistance received from Dr. F. El Guibaly (NSERC Operating Grant #3-45388) and the University of Victoria was greatly appreciated.

To my wife,

Debbie

Chapter 1

Introduction

1.1 Introduction

The high component densities available by the advent of VLSI technology make feasible the hardware implementation of various digital filtering and Fourier transform algorithms. The principal objective of the set of algorithms known as the fast Fourier transforms, or FFT's, is to reduce the computation time required to calculate the discrete Fourier transform or DFT. In some cases improvement factors exceeding 100 may be realized [1]. Since the DFT is central to most spectral analysis problems, it is important that the FFT operation be thoroughly understood. Sections 1.2 and 1.3 of this chapter review the development of the radix-2 decimation-in-time and frequency algorithms [1,2].

In Chapter 2, the concepts of pipeline and parallel processing are introduced. A general survey of proposed DFT and FFT implementations is also made, in which the feasibility of implementation for each network is analyzed. An area-time^2 performance factor is also determined for each implementation.

Chapter 3 presents a detailed analysis on the effects of finite register lengths on FFT performance. Roundoff and truncation noise models are presented for both fixed and floating point data formats. The output signal-to-noise ratios for signal power and output signal variance are determined as functions of register length and transform size. From these ratios, it is possible to determine the minimum register length required to conform to a specified output signal-to-noise ratio.

Chapter 4 outlines the VLSI design methodology used in designing the processing node. The project design specifications are determined on both the network and processing node levels. Also in this chapter, the actual design of the processing node is presented in a bottom-up hierarchal fashion. The function of each node component is given along with the appropriate circuit simulations to verify device performance. Also in Chapter 4, the multiple access pipeline structure is introduced. This structure is used in the design of the floating-point arithmetic processing unit. It combines the processing efficiency of pipelined computations with the flexibility of a bus architecture to create a flexible highly-concurrent processing structure.

Chapter 5 addresses the problem of control logic design. In this chapter a master-slave control scheme is presented in which the APU control unit acts as slave to the node-level master controller. Detailed control specifications for both units are also developed at this point. This chapter also reviews the principles of microprogram control, and develops a coded microinstruction for the APU control unit. Two microprograms are then presented which are responsible for performing complex-number multiplication and sum/difference calculations.

In Chapter 6 estimates are made concerning both processor node and system level performance times. The design tools and simulators used are described and criticized. Other points of discussion in this chapter include the processor node power requirements, affects of scaling, and design for testability.

Chapter 7 summarizes the processing node and system level performance results, and operating parameters. Pertinent conclusions reached in previous chapters are reviewed, and future directions for research are proposed. These proposals include discussions of applications to real-time systems, windowing, and multi-dimensional FFT implementations.

1.2 Decimation in Time FFT

The discrete Fourier transform for a finite duration series of length N, may be defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi/N)nk} \qquad k=0,1,2,...N-1 \tag{1.1}$$

For convenience we may write equation (1.1) as:

$$X(k) = \sum_{n=0}^{N-1} x(n)\omega_N{}^{nk} \tag{1.2}$$

where ω_N is known as the twiddle factor, and $\omega_N=e^{-j2\pi/N}$. From equation (1.1) we see that when x(n) is a sequence of complex-numbers, direct evaluation of the DFT requires some $(N-1)^2$ complex-number multiplications and N(N-1) complex-number additions. For large values of N therefore, an inordinate amount of calculation is required.

The principal concept behind the FFT is to break the DFT into two shorter sequences. The DFT's of the shortened sequences are then recombined to give the DFT of the original sequence [3]. For example if N were even, each of the N/2-point DFT's would require on the order of

$(N/2)^2$ complex number multiplications. Consequently, a total of $2(N/2)^2$ complex multiplications is required. This represents a factor of two improvement over the number required to calculate the DFT directly. By assuming N to be a power of two, this splitting process may be repeated until a series of 2-point DFT's has been obtained introducing an approximate factor of two improvement per splitting. Notice that this factor of two savings per splitting is only approximate because we have not yet accounted for a way to recombine the smaller DFT's to form larger ones.

Consider the case for which N is a power of two. We now define two (N/2)-point input sequences such that $x_1(n)$ and $x_2(n)$ are the even and odd members of the sequence x(n) respectively. These sequences are represented as:

$$x_1(n) = x(2n) \qquad n=0,1,2,...N/2-1 \tag{1.3}$$

$$x_2(n) = x(2n+1) \qquad n=0,1,2,...N/2-1 \tag{1.4}$$

The N-point DFT is now described by:

$$X(k) = [\sum_{n=0}^{N-1} x(n)(\omega_N)^{nk}]_{n_{even}} + [\sum_{n=0}^{N-1} x(n)(\omega_N)^{nk}]_{n_{odd}} \tag{1.5}$$

$$X(k) = \sum_{n=0}^{N/2-1} x(2n)(\omega_N)^{2nk} + \sum_{n=0}^{N/2-1} x(2n+1)(\omega_N)^{(2n+1)k} \tag{1.6}$$

By recognizing that $(\omega_N)^2$ may be written as:

$$(\omega_N)^2=[e^{j2\pi/N}]^2=e^{j2\pi/N/2}=(\omega_{N/2}) \qquad (1.7)$$

We can write equation (1.6) as:

$$X(k) = \sum_{n=0}^{N/2-1} x_1(n)(\omega_{N/2})^{nk} + (\omega_N)^k \sum_{n=0}^{N/2-1} x_2(n)(\omega_{N/2})^{nk} \qquad (1.8)$$

or,

$$X(k) = X_1(k) + (\omega_N)^k X_2(k) \qquad 0\leq k\leq N/2 \qquad (1.9)$$

where $X_1(k)$ and $X_2(k)$ are the (N/2)-point DFT's of $x_1(n)$ and $x_2(n)$. Notice however that $X(k)$ is defined for $0\leq k\leq N-1$ while $X_1(k)$ and $X_2(k)$ are defined only for $0\leq k\leq N/2$. A way must be determined then to evaluate equation (1.9) for values of $k>N/2$. Since $X_1(k)$ and $X_2(k)$ are both periodic, each with period N/2, we can express equation (1.9) as:

$$X(k+N/2) = X_1(k+N/2) + (\omega_N)^{(k+N/2)} X_2(k+N/2) \qquad (1.10)$$

By observing that $(\omega_N)^{(k+N/2)}=-(\omega_N)^k$ due to the periodicity of the DFT, we rewrite equation (1.10) as:

$$X(k+N/2) = X_1(k)-(\omega_N)^k X_2(k) \qquad (1.11)$$

Equations (1.9) and (1.11) are jointly referred to as the butterfly calculations. This operation is represented graphically in figure 1.1(a). In this figure, the dot represents an adder/subtractor unit, with the sum term always appearing on the top output branch and the difference term on the bottom. The arrow represents a multiplier unit with the value of the multiplicand given beside it. In general, the inputs and outputs to the butterfly structure are taken to be complex-numbers.

Figure 1.2(a) illustrates the flow graph process required to evaluate an eight-point DFT, using two four-point transforms. In this flow graph the input sequence is shuffled into the even and odd $x_1(n)$ and $x_2(n)$ sequences which are transformed to yield the $X_1(k)$ and $X_2(k)$ transforms. These values are then recombined via the butterfly operation to yield the final transform X(k).

The above process can be repeated, expressing each of the N/2-point DFT's as a combination of two (N/4)-point DFT's and so on. In this case we can write $X_1(k)$, $0 \leq k \leq N/2 - 1$ as:

$$X_1(k) = A(k) + (\omega_{N/2})^k \, B(k) \tag{1.12}$$

or,

$$X_1(k) = A(k) + (\omega_N)^{2k} \, B(k) \tag{1.13}$$

where A(k) and B(k) are the (N/4)-point DFT's of the even and odd members of $x_1(n)$, respectively. From this point the development proceeds exactly as described above. Figure 1.2(b) illustrates the eight-point DFT calculated from two four-point DFT's, which are in turn calculated from four two-point DFT's. In this figure the C(k) and D(k) are the corresponding A(k) and B(k) terms elements for the $x_2(n)$ input sequence, respectively. In general this splitting process is repeated until just the two-point DFT's are left to calculate. The two-point DFT's are evaluated for k=0,1 and lead to a simple sum/difference calculation requiring no multiplications. This comes from the fact that $\omega_2{}^0$ and $\omega_2{}^1$ are +1 and -1, respectively. Figure 1.3 illustrates the final radix two flow graph in which the two-point DFT's are evaluated using a butterfly operator with a multiplier of ω^0 to realize the sum/difference calculation.

By analyzing figure 1.3 we find that each stage or halving of the FFT requires N/2 complex multiplications. Since there are $\log_2 N$ stages, the total number of complex multiplications required to evaluate the FFT may be approximated by $N/2 \log_2 N$. This value is approximate because certain multiplications, i.e. ω^0,$\omega^{N/2}$, $\omega^{N/4}$ are simply complex additions and subtractions.

The algorithm described above is commonly referred to as the decimation-in-time (DIT) algorithm. At each stage of the process, the input sequence, i.e. the time sequence, is divided or decimated into smaller sequences for processing. A second commonly applied FFT algorithm is the decimation-in-frequency (DIF) algorithm. This algorithm is the subject of section 1.3. A point worth noting is that for the output sequence to be calculated in a natural order, that is X(k), k=0,1,2,...N-1, the input sequence must be input in a bit-reversed manner. In the example in figure 1.3 the required input sequence is: x(0), x(4), x(2), x(6), x(1), x(5), x(3), and x(7). Table 1.1 illustrates what is meant by the bit reversal operation, using an 8 element input sequence as an example.

1.3 The Decimation in Frequency Algorithm

In the DIF algorithm, the input sequence is again partitioned into two smaller input sequences. Unlike the DIT algorithm, the first sequence $x_1(n)$ consists of the first N/2 points while $x_2(n)$ consists of the last N/2 points of x(n). We have then:

$$x_1(n) = x(n) \qquad n=0,1,2,\ldots N/2-1 \tag{1.14}$$

$$x_2(n) = x(n+N/2) \qquad n=0,1,2,\ldots N/2-1 \tag{1.15}$$

The N-point DFT of x(n) may then be written as:

$$X(k) = \sum_{n=0}^{N/2-1} x(n)(\omega_N)^{nk} + \sum_{n=N/2}^{N-1} x(n)(\omega_N)^{nk} \tag{1.16}$$

$$X(k) = \sum_{n=0}^{N/2-1} x_1(n)(\omega_N)^{nk} + \sum_{n=0}^{N/2-1} x_2(n)(\omega_N)^{(n+N/2)k} \tag{1.17}$$

$$X(k) = \sum_{n=0}^{N/2-1} [x_1(n) + e^{-j\pi k}\, x_2(n)](\omega_N)^{nk} \tag{1.18}$$

where we substituted $e^{-j\pi k}$ for $(\omega_N)^{Nk/2}$. We now consider the even and odd output terms of the DFT separately. The even output terms are given by

$$X(2k) = \sum_{n=0}^{N/2-1} [x_1(n) + x_2(n)](\omega_N)^{2nk} \tag{1.19}$$

$$X(2k) = \sum_{n=0}^{N/2-1} [x_1(n) + x_2(n)](\omega_{N/2})^{nk} \tag{1.20}$$

and the odd output terms are given by

$$X(2k+1) = \sum_{n=0}^{N/2-1} [x_1(n) - x_2(n)](\omega_N)^{n(2k+1)} \tag{1.21}$$

$$X(2k+1) = \sum_{n=0}^{N/2-1} ([x_1(n) - x_2(n)](\omega_N)^{n})(\omega_{N/2})^{nk} \tag{1.22}$$

The substitutions $e^{j\pi k}=1$ for k even and $e^{j\pi k}=-1$ for k odd were made. The even and odd terms of the DFT then can be obtained from the (N/2)-point DFT's of the input sequences f(n) and g(n), where f(n) and g(n) are defined as:

$$f(n) = x_1(n) + x_2(n) \qquad n=0,1,2,\ldots N/2-1 \tag{1.23}$$

$$g(n) = [x_1(n) - x_2(n)](\omega_N)^{n} \qquad n=0,1,2,\ldots N/2-1 \tag{1.24}$$

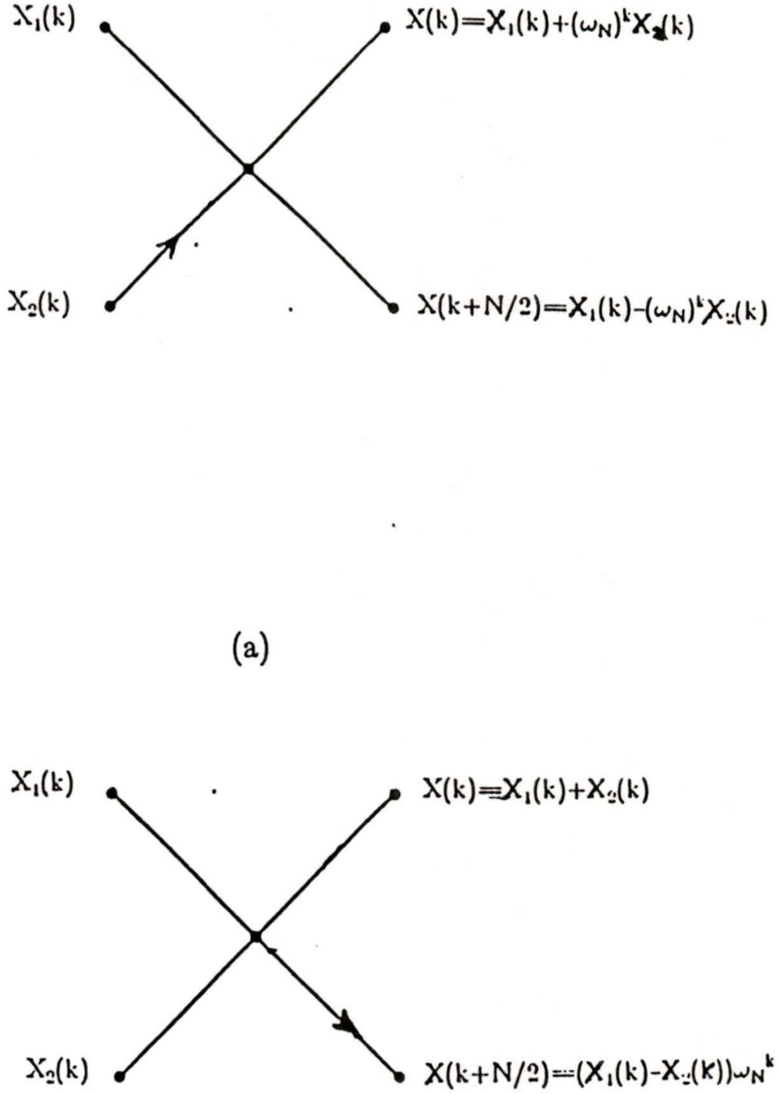
These operations are known as the butterfly calculations for the decimation in frequency algorithm. The flow graph for this butterfly operation is shown in figure 1.1(b). As in the case of the DIT algorithm, the above procedure is repeated so that each of the N/2-point DFT's is expressed as two N/4-point DFT's and so on. Figures 1.4(a-b) show the flow graph procedures for N=8 using reduction from an eight-point DFT to two four-point transforms, and further reduction down to four two-point DFT's. Figure 1.5 shows the complete in-plane eight-point decimation-in-frequency algorithm.

As in the case of the DIT algorithm, approximately $N/2 \log_2 N$ complex-number multiplications are required to fully evaluate X(k). The DIF algorithm must also perform a bit reversal operation, however as seen from the flow graph in figure 1.5, this reversal occurs at the output not the input. The second major difference between the DIF and DIT algorithms is that the complex-number multiplication in the DIF occurs after the sum/difference calculation not before as in the case of DIT. As will be shown in Chapter 2, this is an important consideration deciding on possible network implementations.

1.4 Summary

In summary this thesis investigates the CMOS VLSI implementation of a radix-2 decimation-in-frequency fast Fourier transform algorithm. The design of a three-micron CMOS general-purpose FFT processing node for implementation in parallel and pipelined networks is presented. This node utilizes a fast, highly flexible custom-built arithmetic processing unit to perform all the required floating-point calculations [4].

The proposed network implementation can calculate real, complex, or inverse FFT's. Estimates indicate that a 1024-point FFT could be calculated in approximately 0.9 ms. For certain applications therefore, real-time FFT calculations are possible. For example, the screen refresh rate for most video terminals is usually no more than 60 Hz. In this case, a new 1024-point FFT could be calculated and output within the screen update period, thus resulting in a real-time FFT video display system. Such a chip would find extensive application in almost any field requiring fast spectrum analysis. Examples include speech and image processing, seismic exploration, as well as sonar and radar applications. The speed and flexibility of the processing node design and the proposed network layout result in a very powerful digital signal processing tool.

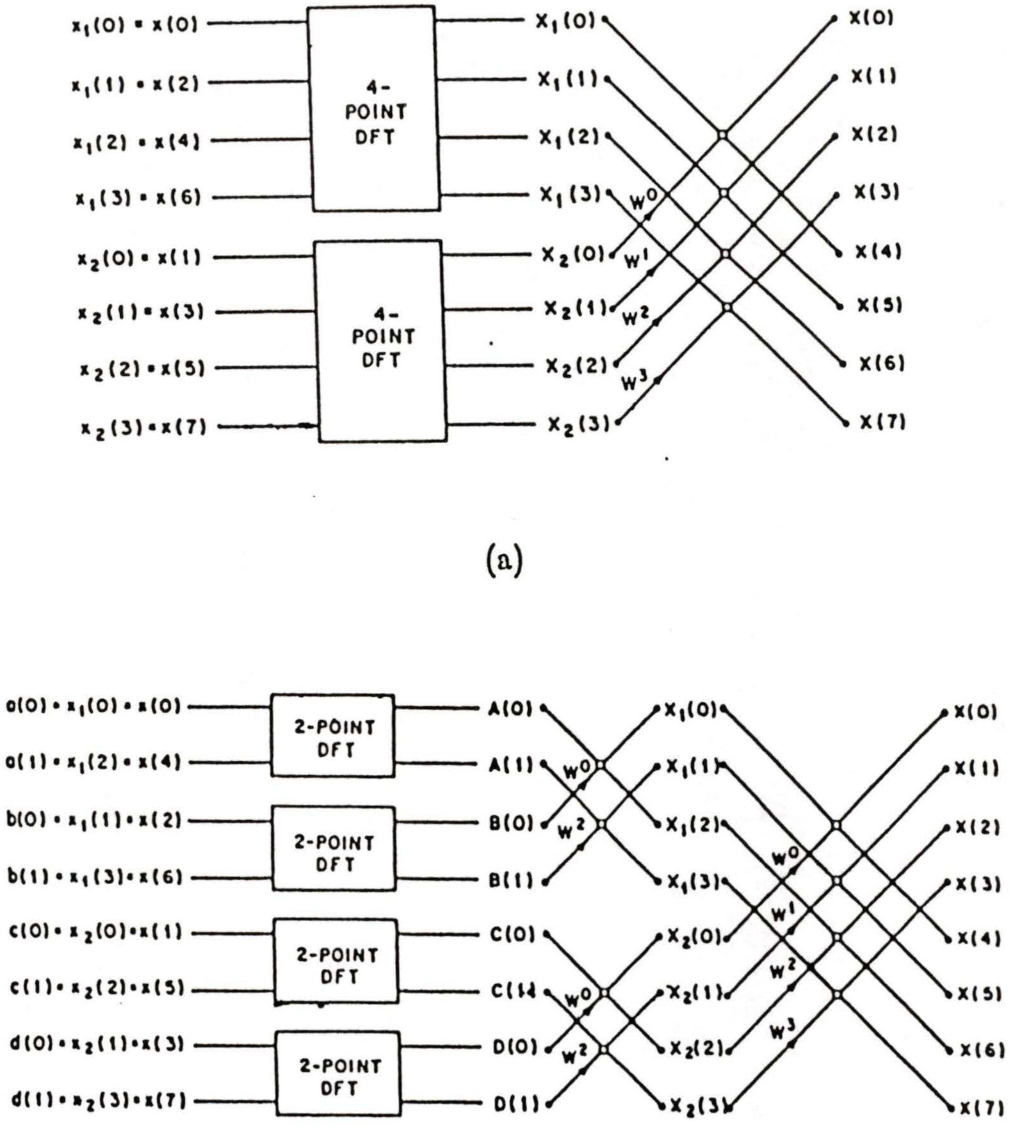


(a)

(b)

Figure 1.1

A butterfly flow graph for a) a decimation-in-time algorithm and b) a decimation-in-frequency algorithm.



(a)

(b)

Figure 1.2

A decimation-in-time flow graph for a) an eight-point DFT from two four-point DFT's and b) an eight-point DFT from two four-point DFT's that are in turn constructed from four two-point DFT's.

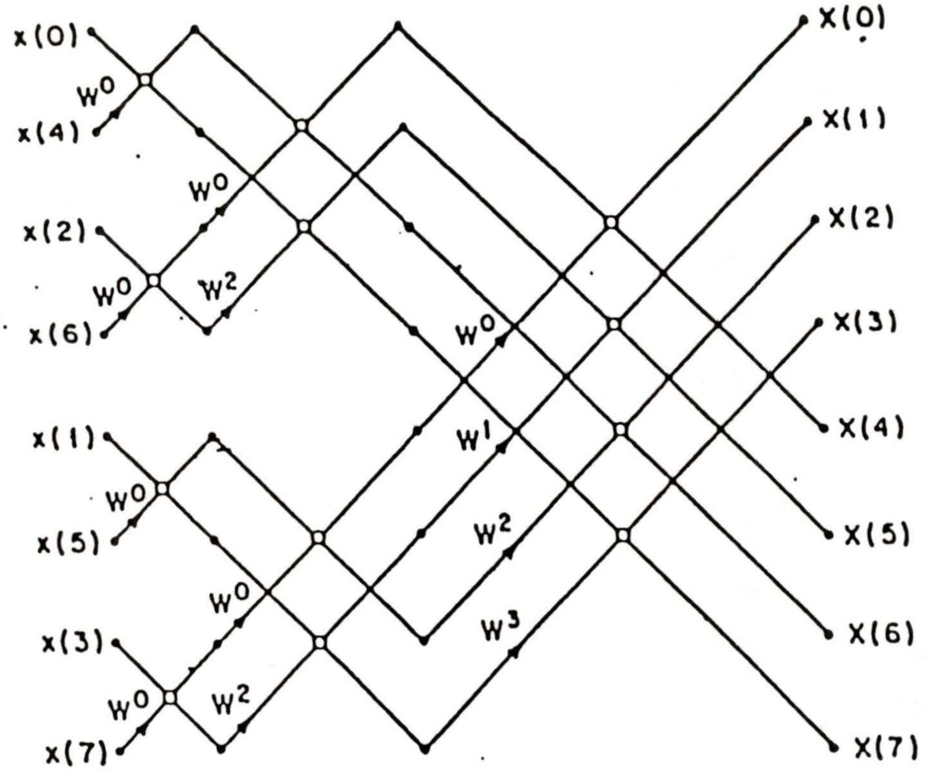


Figure 1.3

A complete in-plane decimation-in-time flow graph for an eight-point FFT.

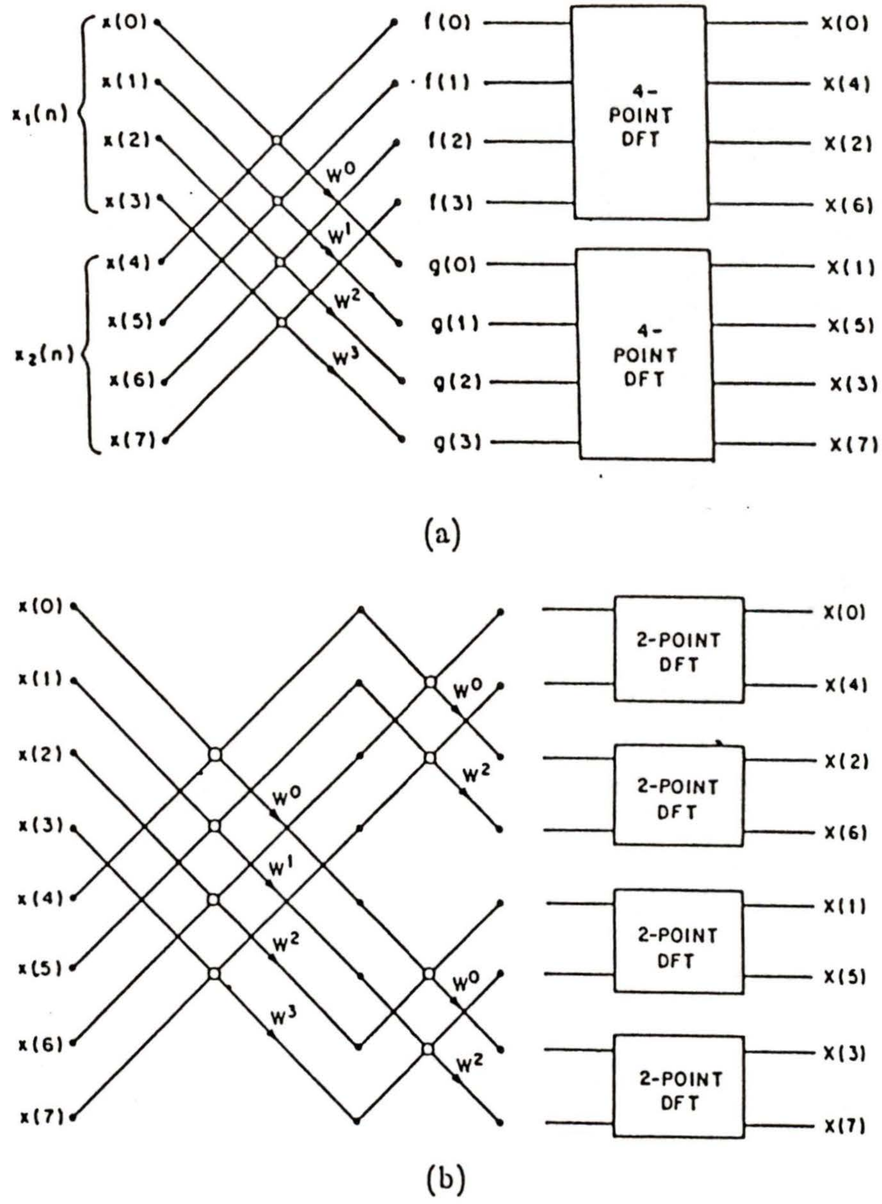


Figure 1.4

A decimation-in-frequency flow graph for a) an eight-point DFT from two four-point DFT's and b) an eight-point DFT from two four-point DFT's that are in turn constructed from four two-point DFT's.

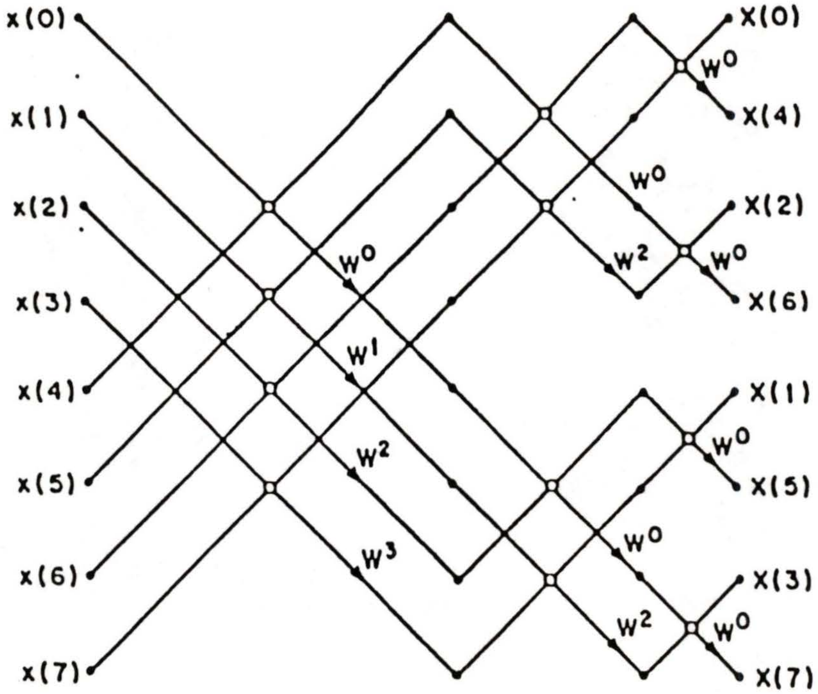


Figure 1.5

A complete in-plane decimation in frequency flow graph for an eight-point FFT.

Bit Reversal Example			
Input Sequence	Initial Binary Indice	Reversed Binary Indice	Output Sequence
X(0)	000	000	X(0)
X(1)	001	100	X(4)
X(2)	010	010	X(2)
X(3)	011	110	X(6)
X(4)	100	001	X(1)
X(5)	101	101	X(5)
X(6)	110	011	X(3)
X(7)	111	111	X(7)

Table 1.1
A bit reversal example for an eight-point input sequence.

Chapter 2

Parallel and Pipelined Processing Systems for FFT Implementation

2.1 Introduction

In recent years, more and more emphasis has been placed on the development of highly concurrent processing systems, capable of performing numerous tasks at the same time. With the onset of VLSI technology the means by which highly concurrent systems can be implemented is realized. As opposed to the conventional "von Neumann machine" which uses a single processor element to retrieve and sequentially execute operating instructions, the increased element density available through VLSI allows several processing units to operate concurrently in close proximity to the memory elements [5]. Thus the process is sped up both through the implementation of concurrent processors, and by the reduction of the length of the data path between the memory cell and the processor.

Improvements to the speed of concurrent systems are obtained by the efficient utilization of the available memory. This is realized by the development of a memory hierarchy, which essentially breaks up the bulk memory into a set of sub-memory blocks. Typically these blocks increase in both size and access time the further removed they are from the processing unit.

A major field of active interest with respect to concurrent processing systems is in the design and configuration of the processors themselves. In this case the object is to increase the degree of concurrent operations within a system by means of the appropriate design of processor configurations [5]. Two of the most successful approaches are the parallel and pipelined processing structures which prove to be the subjects of sections 2.2 and 2.3 respectively. Having established the basic concepts of parallel and pipelined processing, a review will then be made of various proposed implementations for FFT and DFT computational networks. Area and computational time constraints for each network will be analyzed. Following this a more complete discussion of the network chosen for implementation, the pipelined cascade network, will be made.

2.2 Parallel Processing

Put simply, parallel processing makes use of several processing elements each of which simultaneously performs usually identical tasks on independent sets of input data. The requirement that the input data is independent of one another is necessary to ensure that no recursive interaction between processors takes place. Important points for consideration, when designing a parallel processing structure include:

1. Simplicity of processor communication
2. Regular interconnections between processors.
3. Data input/output schemes.
4. Interaction and data exchange between processors.

Each of these points is addressed in the analysis of the proposed FFT implementations found in section 2.4.

As a simple example of parallel processing consider the multiplication of a real n x n matrix by some real constant factor C. In the typical single-processor system this task would be performed by sequentially looping through the row/column indices of the matrix, multiplying each element by the constant C. Thus completion of this task would require a total of n^2 multiplications or n^2T_m time units, where T_m is the time taken to perform a single multiplication. By implementing a parallel network of n multipliers, this same operation could be performed by assigning each row of the matrix to a given multiplier and simply incrementing through the n elements of each row. In this way the total multiplication time of the n x n matrix is reduced from n^2T_m to nT_m. Further parallelism could be realized by implementing a network of n^2 multipliers, i.e. one for each of the matrix elements, and thus the matrix multiplication time would be reduced to that of a single element multiplication. Figures 2.1(a-c) illustrate the basic principle of parallel processing using the matrix multiplication example.

Notice that in each multiplication scheme of the preceding example the area time product remains a constant.

$$\text{Area} \times \text{Time} = \text{Constant} \tag{2.1}$$

Any incrementation of the area by a factor of x necessarily requires a reduction of a factor of x in the time term.

2.3 Pipelined Processing

Unlike parallel structures, pipelined systems make use of several usually different processing elements, generally connected in a sequential order, each of which performs a unique task on the data. In designing a pipelined processing system, consideration should be given to the following important points:

1. Simplicity of nearest-neighbor connections.
2. Data I/O at the pipeline periphery.
3. Control methods.
4. Speed of operation.
5. No interaction is made with previous or on-coming samples.

These points in conjunction with those outlined in the previous section form the basic criteria upon which the choice of implementation was made.

A useful analogy for describing pipeline processing is to compare its operation to that of a manufacturing production line in which various parts (data) proceed one after the other down the production line. At various points along the line, workers (processors) remove the parts (data) and perform some given operation on them (multiplication, addition etc.). Once completed, the parts (data) are placed back on the line, where they proceed down the line to the next worker who performs a different operation on them. Figure 2.2 illustrates a typical pipelined network comprised of a set of adders and multipliers.

In comparison to parallel structures pipelined systems are perhaps better suited towards special-purpose applications requiring that several operations be performed sequentially on input data. For this reason and the fact that most signal processing data is sampled digitally and produced in a data stream fashion, pipelined configurations are particularly well suited for handling signal processing operations.

An important modification of both the parallel and pipelined processing networks that should also be mentioned is that of the parallel pipeline. The parallel pipeline can be thought of as being the combination of a pipelined network on the word level or organization and a parallel network at the bit level. This configuration is achieved by modifying the processors of the pipelined network to perform the assigned tasks, and communicate with one another in a bit-parallel rather that a bit sequential manner. The parallel pipeline network has proven to be a highly efficient processor configuration and is currently the object of expanded implementation especially in the field of digital signal processing [5,6].

2.4 Fourier Transform Implementations

In this section a general overview is made of several proposed VLSI implementations of Fourier Transforms [5,7,8,9]. These implementations are presented for both the Discrete Fourier Transform and the Fast Fourier Transform methods. In general, they make extensive use of the parallel and pipelined networks described previously. In a paper by Thompson, an analysis is made by comparing network performance to that of a proven optimal area-time^2 performance of $O(N^2 \log^2 N)$, where N is the transform sample size [7]. Thompson's analysis is based on the restriction that only single level metalization is used and that all communication and circuit processing units are operating in a bit serial fashion. In the following analysis a slightly different analytical approach is taken.

In an attempt to improve the performance of the processing speed, the assumption is made that each of the processors will operate in a bit parallel fashion. As discussed in section 2.2 this will reduce the processing time for an n-bit datum by a factor of n. This gain in time performance however, usually comes at the cost of a factor of n increase in processor size. As it is the processing speed however and not area that is the principle consideration in this implementation this area time trade off is considered acceptable.

In order to further facilitate the analysis, we consider each network to be implemented using essentially identical processing units. Each processor contains, in addition to its internal shift registers and control logic, a fast arithmetic processing unit. In theory each FFT processor would receive a pair of data elements and perform one of the basic butterfly operations seen in figure 1.1. In the case of DFT network implementations slight modifications to the processor's control unit will be necessary. These modifications are concerned primarily with the problem of external data routing, and the sequencing of on-chip operations. In general the DFT network processor will perform a single complex-number multiplication followed by a single complex-number addition. In contrast the FFT butterfly operation requires a single complex-number multiplication and two complex-number additions. Notice that in terms of actual multiplications required, a single complex-complex multiplication of two unequal terms requires a total of 4 real multiplications. Similarly a complex additions may be thought of as being comprised of two real additions. Thus for analytical timing considerations we use the real multiplication time, T_m, and the real addition time, T_a, as the base time units. The approximated processing time for a single FFT processor element $T_{p,FFT}$ is given as:

$$T_{p,FFT} = 4 \times T_m + 4 \times T_{a,} \tag{2.2}$$

For the DFT processing element the $T_{p,DFT}$ is given by:

$$T_{p,DFT} = 4 \times T_m + 2 \times T_a \tag{2.3}$$

Updating of the twiddle factor can be achieved through the use of look-up tables. To explicitly store these constants on-chip however would require a prohibitively large amount of area. Off-chip memory would have to be used, requiring extensive off-chip communication. For this reason on-chip updating of the twiddle factor is employed. In terms of the DFT defined in equation (1.1), this is achieved simply by multiplying the twiddle factor of the preceding term ${\omega_N}^{(n-1)k}$ by ${\omega_N}^k$.

$${\omega_N}^{nk} = {\omega_N}^k \times {\omega_N}^{(n-1)k} \tag{2.4}$$

In the case of FFT implementation twiddle factor update is slightly different and more dependent on the configuration used. The end result however for both cases is that on-chip update requires one additional complex-number multiplication ($4T_m$) to be added to the total processing time. As will be shown in Chapter 4, by using fast parallel multipliers, we obtain, $T_m \approx 2T_a$. The processing time for the DFT and FFT processing elements with on-chip twiddle factor update is then:

$$T_{p,FFT} = 8 \times T_m + 4 \times T_a \approx 10 \times T_m \tag{2.5}$$

for the FFT processing element, and

$$T_{p,DFT} = 8 \times T_m + 2 \times T_a \approx 9 \times T_m \tag{2.6}$$

for the DFT processing node.

As a means of quantitative analysis for comparing the various networks, we will consider the product of the total network area and the total network processing time, $T_{net.}$, squared, for an N-point transform. This area–time^2 performance factor is determined for each implementation, and it serves as a measure of the networks processing efficiency. Taking the area of the processing units as being approximately constant and assigning to them a value of unity, the total network area is expressed as the total number of processors in the network. The network processing time will be based in terms of the time factor T_m. The squaring of this term takes into account the relative weighting placed on the area-time considerations. Ideally the optimal network will be the one with minimum AT^2 product [7]. The final implementation decision however will not be based on this criterion alone. Analysis based on practical considerations is also needed to determine the implementation feasibility with regards to such matters as parallel communication between processors, and the length of off-chip communication lines. In some implementations it is not possible to fit the total number of components on a single chip. This is particularly true for those implementations requiring large amounts of RAM or a large number of shift registers. In these cases one of the structures, typically the memory component, must be located off-chip. This requirement leads to an increase in the processors off-chip communication thus slowing the speed of processing and necessitating the need for additional I/O ports and output drivers. In general, the requirement of off-chip components to support processing operations increases both the complexity and processing time of the processor.

2.4.1 Single Cell DFT Implementation

Direct calculation of the DFT given in equation (1.1) may be realized by following the algorithm presented in figure 2.3. In this algorithm the discrete Fourier transform is calculated requiring a total of N^2 complex-number multiplications. It has been shown by Kung, that this algorithm can be treated as a simple matrix-vector multiplication [5]. An N-element input vector **X** is multiplied by a constant matrix **C**, to produce the transformed output vector **Y**. Viewed in this light three degrees of concurrent processing operations present themselves. As suggested in the example in section 2.2 we can perform this operation by implementing one, N, or N^2 processing elements. In the case of large input data vectors (ie. 1024 points) the N and N^2 element processing networks would require a prohibitively large amount of area. For this reason the N andN^2 cases are not considered for implementation.

Figure 2.4 shows a block diagram network layout for the single-cell DFT. Input data are received from the digitizing unit, and immediately input into off-chip RAM, where it is held throughout the computation. The base twiddle factor ω_N is then loaded into the processor to be used as the seed for subsequent on-chip update. Two on-chip counters are employed to keep track of the looping indices n and k and to ensure proper updating of the twiddle factor ω_N^{nk}. Calculation of the output element, X(k) proceeds by directly following the DFT algorithm in figure 2.3. A complex valued input element x(n) is read into the processor sequentially from RAM and multiplied by ω_N^{nk}. The product is passed into an on-chip accumulator and the twiddle factor is updated. The next input element in the series is read, and the process repeats itself. The output element X(k) is determined after all input elements x(n) have been multiplied and passed into the accumulator. The "k" counter increments by one, and the next value of X(k) is calculated. The single cell-DFT utilizes the least amount of parallelism of any of the proposed networks reviewed here, and as such requires the longest time to process an N-point DFT. The implementation also requires off-chip RAM to store the N-element input vector which must be preloaded before calculation commences. This is needed since each of the N-input elements x(n) are called once for every iteration of the "k^{th}" loop of the discrete Fourier transform algorithm. In this implementation the transform vector is output in order, with a delay time between each element of $NT_{p,DFT}$. This corresponds to a single iteration of the loop index n. Since N output elements need to be calculated, the total computation time for an N-point DFT is proportional to N^2:

$$T_{net.} \approx N^2\ T_{p,DFT} \approx 9\ N^2\ T_m \tag{2.7}$$

Thus on grounds of time delay considerations the N^2 dependence makes this configuration too inefficient for implementation purposes. The area-time^2 product is given by:

$$AT^2_{net.} \approx 81 \times N^4\ T_m^2 \tag{2.8}$$

Table 2.1 reviews the $AT^2_{net.}$ factors for all the implementations reviewed. From this table it is apparent that the single-cell DFT offers the lowest performance factor of all the implementations.

2.4.2 The FFT Network

The FFT network is the first of the proposed networks to take advantage of a developed FFT algorithm, in this case a radix-2 decimation-in-time. Figure 2.5 presents the decimation-in-time algorithm employed by Thompson in his review of FFT implementations [7]. By employing this algorithm, the number of required complex-number multiplications is reduced from being proportional to N^2 down to $\frac{N}{2}\log_2 N$.

In the FFT network, full parallelism is reached in that each multiplication employs its own processor. That is a total of $\frac{N}{2}\log_2 N$ processors are used. Figure 2.6 shows one possible configuration for the FFT network. In this case the processors are configured in an array of $\log_2 N$ rows by $\frac{N}{2}$ columns. Each row corresponds to a unique iteration of the loop index M of the algorithm in figure 2.5. The conditional found in line 8 of the algorithm directs the butterfly interconnect pattern between rows. This point was discussed in detail in Chapter 1.

Notice from figure 2.6 that the FFT network data elements are input in a bit-shuffled order, and the subsequent outputs appear in a bit-reversed order. As the data pair enters the processor from the top, it immediately undergoes the typical butterfly calculation common to most FFT algorithms. The output data are then appropriately routed into the next processing stage by means of the butterfly interconnection pattern. A simple method of configuring the interconnection layout is to number the cells in order from left to right from 0 to $\frac{N}{2}$ - 1. The output of cell i in the first row is then connected to cell i and cell $(i + \frac{N}{4})\mathrm{mod}(\frac{N}{2})$ of the second row. In general cell i of row j is connected to cell i and cell $(\frac{i}{N/2^j})+(i+\frac{N}{2^{j+1}})\mathrm{mod}(\frac{N}{2})$ in the $j+1^{st}$ row.

As before, the area occupied by the network is given by the total number of processors employed, in this case $\frac{N}{2}\log_2 N$. Because of the parallelism employed in each row, the total computational time is given by the product of the processor time $T_{p,FFT} \approx 10T_m$, and the number of rows, $\log_2 N$. Thus T_{net} is given by:

$$T_{net} \approx \log_2(N)10T_m \tag{2.9}$$

The area-time^2 factor is:

$$AT_{net.}^{\ 2} \approx \frac{N}{2}\log N \left(\log_2{}^2N\ 100(T_m{}^2)\right) \tag{2.10}$$

$$AT_{net}^{\ 2} \approx \frac{100N(\log_2 N)^3(T_m)^2}{2,} \tag{2.11}$$

This represents a performance improvement over the network previously analyzed.

The FFT network shows excellent prospects when subjected to a quantitative analysis, particularly with respect to its area-time^2 product. Practical considerations however brings to surface some of the problems associated with the FFT network. To accomplish the bit shuffling at the input, data must be preloaded into RAM, then reordered for output to appropriate column locations. At the output, a similar process is required, as the "bit-reversed" products must be reordered and stored in RAM for proper output sequencing. The major problem associated with the FFT network arises as a result of the long butterfly interconnections between processors. These interconnections must all be made by means of off-chip signal lines. The long lines required are highly capacitive compared to on-chip communications lines and thus signal propagation between processors is slowed and drivers are required to drive the lines. The necessary cross overs for the butterfly configuration also poses layout problems for the metal wiring. Implementation would require a PC board with multilevel interconnects. Both these problems are further magnified if we consider as proposed, bit parallel communication between processors. In short, even though analysis of the FFT network yielded excellent quantitative results, problems intrinsic to the network layout pose barriers to its implementation.

2.4.3 Mesh Configuration

The basic outline for the square mesh implementation is seen in figure 2.7 which depicts a mesh of N processors in an array of $\sqrt{N}$ by $\sqrt{N}$ dimensions [7]. This implementation requires that $\sqrt{N}$ be an integer. Each processor in the mesh configuration is assigned the responsibility of calculating an individual output element. Communication in this case is performed in a word-parallel manner between adjacent processors. Using this implementation an N-point FFT is performed in a total of $\log_2 N$ steps, with each step representing a single iteration of the loop index M, and each processor performing the computation for a single value of k. This implementation is essentially a time-multiplexed version of the FFT Network in that for each step, $\frac{N}{2}$ of the mesh's processors perform the role of a single row in the FFT network.

In order to better visualize the mesh implementation, the mesh processors are indexed in natural row-major ordering. Processor n is thus designated the home of element x(n), and after processing, it contains transformed element X(k). During each step, the conditional statement of line 8 in figure 2.5 is performed by having each processor examine the M^{th} bit of reverse(k). In the case where the M^{th} bit is found to be 0 the even numbered cells are designated to proceed and perform the subsequent calculation as required by the algorithm. At the same time each odd processor sends its present intermediate transformed element x(k) to the processor $k + 2^M$. Statements 9 and 10 of the algorithm are then performed, and the updated x(k) elements are returned to their original processors. This form of data routing is referred to as a "distance-2^r route" and is performed immediately before and after the execution of statements 9 and 10 [6].

As this network is comprised of N processors, each taken to have a unit area, the net area of the configuration is approximately N. The total computation time is made up of $\log_2 N$ steps each of duration $T_{p,FFT}$. Therefore we have:

$$T_{net} = \log_2(N)T_{p,FFT} = 10T_m\log_2(N) \tag{2.12}$$

The net area-time^2 term is then:

$$AT_{net.}{}^2 = 100(T_m)^2N^2\log^2 N \tag{2.13}$$

Based on this criterion the mesh implementation appears to be more efficient than the previous two networks. Further analysis however reveals that as in the case of the FFT Network complications arise with data transfer. In this implementation data are transferred with every iteration of the loop index M. The number of shifts, and thus the transmission time is dependent on the value of M, and ranges from a shift between adjacent processors when M=0 to one of $\frac{N}{2}$ processors when $M = \log_2 N - 1$.

In order to avoid the extensive problems that would be encountered if this routing were attempted directly, data are simply passed between adjacent processors. Thus the data elements "ripple" their way through the processor mesh to their destination. In order to achieve this data transfer in an ordered fashion, each output driver must be clocked, and controlled by a central common control unit. All data transfers are then synchronized and performed simultaneously, until the 2^M transfer is achieved. The appropriate calculations are performed on the data, and they are then returned to their home processors simply by reversing the original shift pattern.

The routing time can be determined if we express the base routing time from the output driver of one processor to the shift registers of its nearest neighbor in terms of a constant T_{shift}. The total time spent in data routing T_{route} then is the sum of the products of the total number of data shifts required by the algorithm and T_{shift}. T_{route} may be expressed as:

$$T_{route} = \sum_{M=0}^{\log_2 N - 1} 2^M T_{shift} \approx N T_{shift} \tag{2.14}$$

since,

$$\sum_{M=0}^{\log_2 N - 1} 2^M \approx N \tag{2.15}$$

Thus with this in mind the total computational time must now be modified to include the routing time. So we now have:

$$T_{net} \approx 10 T_m \log_2 N + N T_{shift} \tag{2.16}$$

and,

$$AT_{net}^2 \approx N(10T_m \log_2 N + NT_{shift})^2 \tag{2.17}$$

We see that the mesh implementation is not as efficient as first appeared. In addition to a computational time now proportional to $(N+\log_2 N)$, we require the development of an external controller to direct routing of data within the mesh itself. Also additional shift registers and additional on-chip logic would be required to hold the data elements during the intermediate stages of routing.

2.4.4 The Pipeline Cascade Implementation

After having reviewed the various suggested FFT and DFT implementations, we find the options appear rather polarized. The DFT implementation for example offered a fundamental advantage in the simplicity of the calculations and in network configuration. Unfortunately this simplicity comes at the expense of calculation time performance. The FFT configurations on the other hand, offer excellent performance characteristics in terms of maximized parallelism and minimized calculation time. This high performance however is offset by problems in the implementations of the configurations and in the complexity of data transfer between processors. Clearly some middle ground must be found.

The pipelined cascade FFT implementation provides the compromise needed. While performing a reduced number of calculations by utilizing a radix-2 decimation-in-frequency algorithm, this network maintains the simple configuration required for efficient bit-parallel communication [7,8,9]. Figure 2.8 illustrates the algorithm used by this implementation for computing the DIF FFT. The pipelined cascade implementation is essentially a linear configuration of $\log_2 N$ processors. It is based on a configuration suggested by Despain [8,9], for realizing FFT implementations using CORDIC iterations. In the case of this implementation, Despain's "CORDIC rotator module" is replaced by a complex multiplier, which essentially performs the same operation. As in the previous implementations the basic processor is comprised of a fast APU for performing the DIF butterfly operation, and a set of shift registers. Figure 2.9 shows the pipelined cascade implementation layout for an N-point FFT implementation.

The network is organized into $\log_2 N$ stages with each stage representing a single value of the loop index M in figure 2.8. The leftmost cell in this case performs iterations of the j loop based on a value of $M=\log_2 N-1$, while the rightmost cell is based on $M=0$. The conditional statement of the decimation-in-frequency algorithm is implemented by having each processor analyze the M^{th} bit of the total count of input elements. When this bit is low the conditional statement is found to be false, and the incoming data element is passed unchanged into a shift register of length $2^{\log_2 N-1}$. When the M^{th} bit is high, the conditional is found valid and the operations of lines 8 and 9 are performed. This occurs once a total of 2^M input elements have entered the pipeline. The incoming data element $x(n+2^M)$ is combined with the $x(n)$ element found at the head of the shift registers, and the pair are passed into the APU. In this way a word shift of 2^M has been attained. Once in the APU, the initial sum/difference calculation of the DIF butterfly operation is performed, producing the terms:

$$S_{ij} = x(i)+x(j) \qquad j=i+2^M \tag{2.18}$$

and,

$$D_{ij} = x(i)-x(j) \qquad j=i+2^M \tag{2.19}$$

In this case S_{ij} represents the sum term of the $x(i)$ and the $x(j)$ elements, and D_{ij}, the difference term. The fact that $j=i+2^M$ ensures a word shift of 2^M has been performed. The S_{ij} elements are immediately passed on to the next processor which treats them as elements of a new time sequence $x(n)$. As such they are immediately passed into the shift register to undergo a shifting of 2^{M-1}. Simultaneously the D_{ij} elements (the difference terms) are passed back into the tail of the shift register. After all the input elements have entered, and the x(n) terms have combined with the $x(n+2^M)$ terms, the term D_{ij} will be at the head of the shift registers. This term is then read back into the APU where a complex multiplication by the twiddle factor, $\omega_N{}^{jq}$ where $q=N/2^{M+1}$ as defined in the algorithm in figure 2.8. The product term is then passed on to the next processor. Figure 2.10(a-e) shows the data flow for the calculation of an eight-point FFT.

Figure 2.10(a) shows the input elements x(n) after having undergone the required first stage shift of four elements. In figure 2.10(b) the results of the subsequent operation are seen as the sum/difference terms are calculated. The sum term S_{04} is passed to the second stage while the difference term D_{04} is passed back into the shift registers. This process continues until the last input element has entered the system. At this point the first difference term D_{04} is sitting at the head of the shift register and a 2^{M-1} shift has been realized in each stage. In subsequent stages, the registers are all holding the difference of sum terms. This state is seen in figure 2.10(c). Twiddle factor multiplication now begins. This multiplication continues until all the difference terms sitting in the registers have been multiplied. During this time fresh data elements have been loaded in and the sum/difference calculation is repeated. The data position subsequent to this initial multiplication is seen in figure 2.10(d). During the next operations the twiddle factors are updated and the difference term multiplication continues. Each processing stage contains a constant base twiddle factor, $\omega_N{}^q$ where $q = N/2^{M+1}$. Thus as the index j is incremented, the twiddle factor is increased by a factor of $\omega_N{}^q$. This process continues until the last of the difference terms has been output. Should the stage have been reloaded with new elements, the entire process is repeated.

Each of the subsequent processing stages is identical except for the shift register length and the value of the base twiddle factor. The delaying of the $x(n+2^M)$ elements is necessary to ensure the proper sequence of data flow is maintained. In this way the first stage realizes a data shift of N/2 elements, the second stage of N/4 elements and so on. This is the sequencing required for the butterfly interconnections described in Chapter 1 when discussing the implementation of the radix-2 DIF algorithm. Although the elements enter in a bit-parallel, word sequential manner the output elements appear in a bit-reversed order, and thus must be reordered.

The pipelined cascade implementation is ideal for performing FFT operations for digital signal processing because it is capable of receiving the input data in a sequential form directly from the digitizing unit. Input is read into the system at a rate dependent on the butterfly network. Under general operation, this network performs a single complex addition and a single complex subtraction per data pair, requiring a total operation time equivalent to 4 real additions. The data input rate is constrained by the time taken to perform the sum/difference calculations, $T_{\text{sum/diff.}}$. Thus the input sampling rate is approximately $1/T_{\text{sum/diff.}}$. Each element pair still undergoes only a single DIF butterfly operation, requiring a processing time of $T_{p,FFT}$. Recall that $T_{p,FFT}$ is given by equation (2.5) for the case of on-chip twiddle factor update and by equation (2.2) without the update. The worst case delay time through each processing stage T_{Delay} is given by:

$$T_{\text{Delay}} \approx 2^M (T_{p,FFT}) \tag{2.20}$$

This figure is arrived at by considering that each processing stage acts on a total of $\frac{2^M}{2}$ pairs of data. Thus a total of 2^{M-1} DIF butterfly operations need be performed to process the entire input sequence at stage M. The network processing time is given by the sum of the worst case processing stage delays.

$$T_{net.} = \sum_{M=0}^{\log_2 N-1} 2^M (T_{p,FFT}) \approx N(T_{p,FFT}) \tag{2.21}$$

Thus from equation (2.5)

$$T_{net.} \approx N 10 T_m$$

In view of future calculations it will prove advantageous to consider $T_{p,FFT}$ in terms of the complex-number multiplication time $T_{comp.mult.}$ and the sum/difference calculation time $T_{sum/diff.}$. In this light, $T_{p,FFT}$ is given by:

$$T_{p,FFT} \approx T_{comp.mult.} + T_{sum/diff.} \tag{2.22}$$

without on-chip twiddle factor updating, or by:

$$T_{p,FFT} \approx 2T_{comp.mult} + T_{sum/diff.} \tag{2.23}$$

with the twiddle factor update. Notice that by substituting $T_{comp.mult} = 4T_m$ and $T_{sum/diff.} = 4T_a$ the results of equation (2.5) are obtained. The net area of the pipelined cascade implementation is given by:

$$A = \log_2 N, \tag{2.24}$$

and the area-time^2 term is

$$AT^2 \approx 100 N^2 T_m{}^2 \log_2 N \tag{2.25}$$

While this performance factor is worse than either the FFT network or the mesh configuration, the pipelined cascade implementation was found to offer significant advantages. The pipelined structure makes this network readily adaptable to bit-parallel communication between processors. The configuration is readily implemented and is easily expanded. Doubling the size of the FFT requires only that an additional processing stage be added onto the front end of the network. In terms of size considerations its $\log_2 N$ processors are much more easily dealt with than the $\frac{N}{2}\log_2 N$ or the N processors required by the FFT network or the mesh configuration. Expansion of the latter two networks to larger FFT's is a much more complicated matter, requiring the use of many more processors and much longer off-chip communication paths. Correspondingly the power requirements for the pipelined cascade implementation are much less than those of the larger networks. For estimates of the system level power requirements the reader is referred to Chapter 6.

Problems inherent to this structure lie in its extensive use of shift registers. Each processor utilizes a total of 2^M shift registers, where the value of M is dependent on the processor location in the network. In a worst case scenario of $M = \log_2 N - 1$, as is the case for the first processor, a shift register of length $2^{\log_2 N - 1}$ must be used. Taking each shift register to be 24 bits wide, a total of $24(2^{\log_2 N - 1})$ shift registers are required, which for typical values of N, say N=1024, result in a memory element too large to be implemented on the same chip as the other processor elements. For practical considerations then the word shifts must be achieved by means of off-chip shift registers.

One proposed method is to employ a set of variable length FIFO's (first-in-first-out memory blocks). The store and retrieve time for such devices is typically less than T_m, and thus their use should in no way impede the calculation time or speed of data transfer. It will however increase the network area by some constant factor depending on the number of FIFO's required per stage. All things considered however, the pipelined cascade network is an area/time-efficient method of FFT implementation employing a simple layout configuration.

2.5 Summary

This chapter has reviewed several proposed FFT and DFT implementations. Other proposals were not addressed in this chapter and may be found in any of the previously-mentioned references.

In analyzing the proposals for feasibility of implementation, bit-parallel communication was assumed, and the problems associated with off-chip data transfer were taken into consideration. As a result, the analysis indicated that the near optimal networks, as determined by the area–time^2 performance factors, were not necessarily the best suited for implementation. In the case of the FFT network, and in the mesh configuration, extensive and complex off-chip data transfer was required. These problems were magnified when bit-parallel communications were considered. The pipelined cascade implementation on the other hand offers a very simple linear layout configuration and is ideally suited for bit-parallel communication. The implementation is easily extended to accommodate larger FFT's and as will be shown in Chapter 7, it may also be used in determining inverse and multidimensional FFT's. It requires a total of $\log_2 N$ processing nodes, and the net delay time through the network is proportional to N. For these reasons the pipelined cascade implementation has been chosen as the most suitable for implementation.

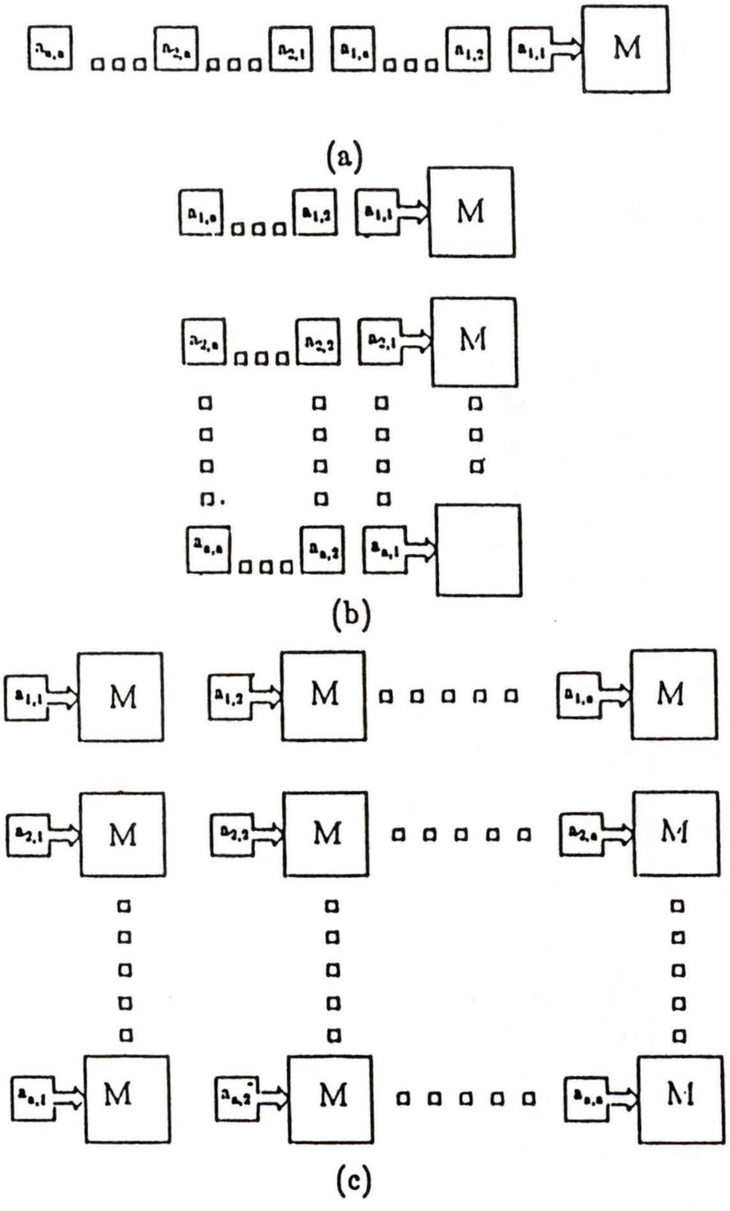


Figure 2.1

Levels of parallel processing as applied to array multiplication. This figure illustrates array multiplication by a constant using (a) a single processor,(b) n processors and (c) n^2 processors.

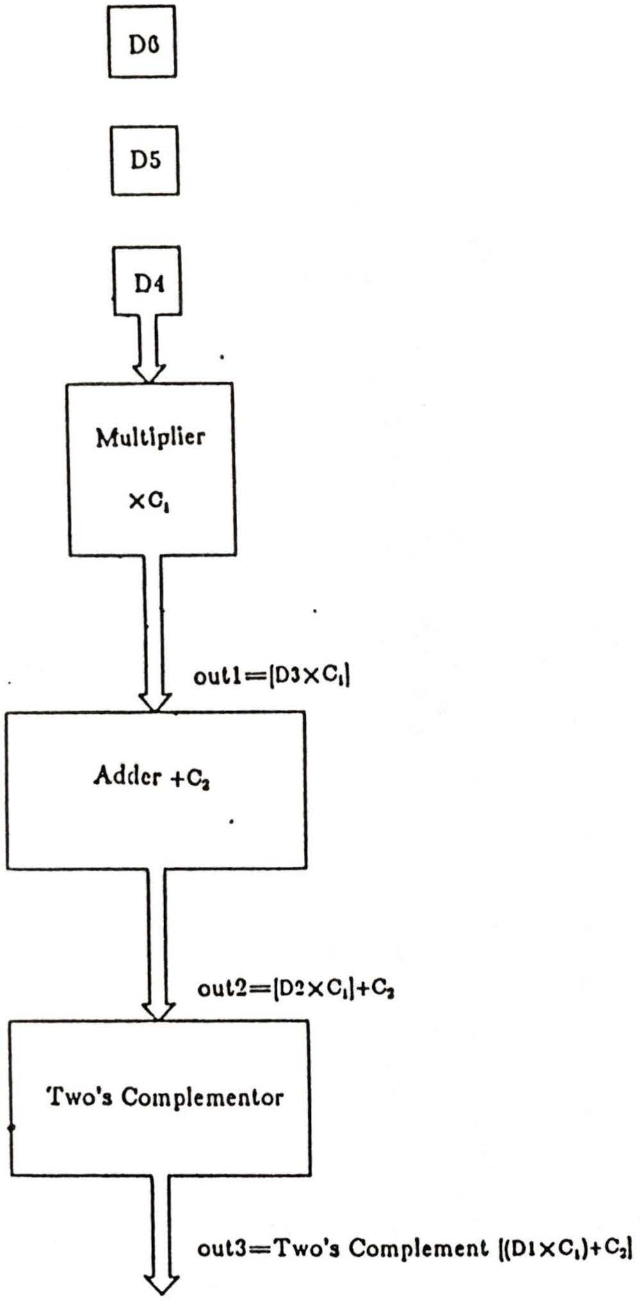


Figure 2.2

A block layout for a simple pipelined processing network. This layout depicts multiplication by a constant followed by an addition and subsequent two's complementation of the product. The values out1, out2, and out3 indicate the calculated intermediate results for data elements D1 - D3.

```
For k=0 to N-1 Do;
  X(k)=0;
  For n=0 to N-1 Do;
    X(k)=X(k)+$\omega^{nk}$x(n);
  End Do;
End Do;
```

Figure 2.3

A simple algorithm for computing DFT's directly.

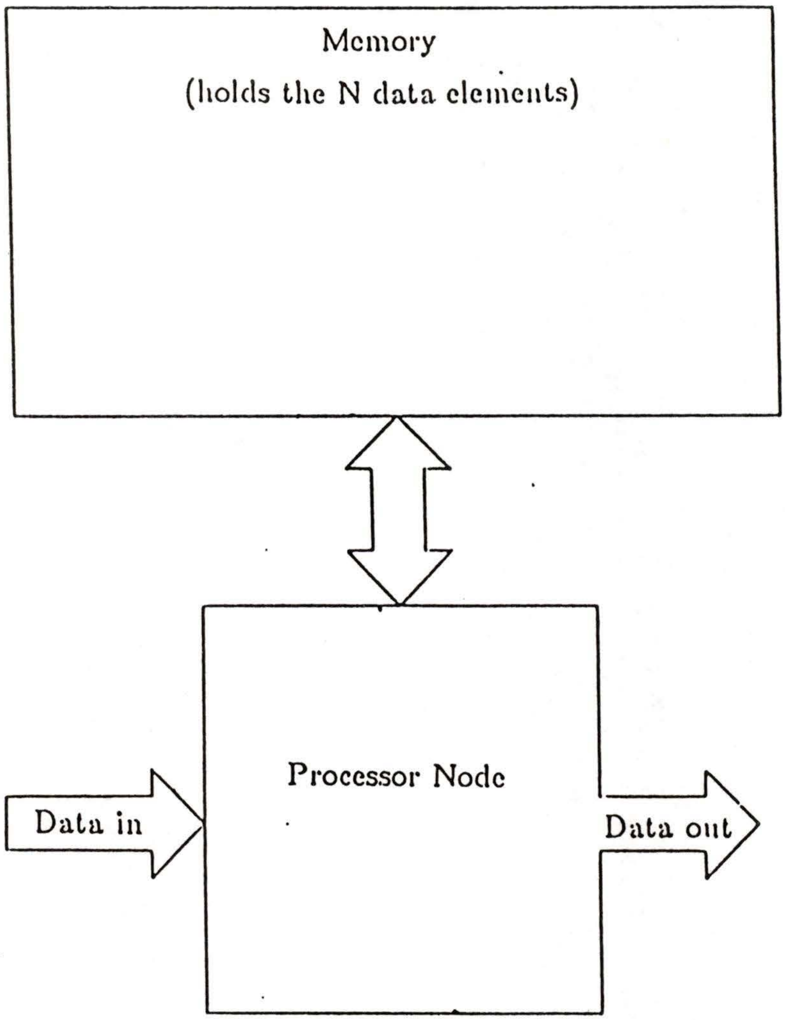


Figure 2.4

Block layout for the single-cell DFT implementation. Notice that the large memory unit located off-chip is required to store the input data array for subsequent processing. The processing node is responsible for performing the required complex-number multiplication and complex-number addition.

```
1.  For M=log(N) - 1 to 0 step -1 Do
2.    p=$2^M$;
3.    q=N/p; /* note that N=pq */;
4.    z=$\omega^p$;
5.      For n=0 to N-1 Do
6.        j= n(mod q)
7.        k= reverse (n)
8.          If(k(mod p))=(k(mod2p) then do
9.            X(k)=x(k)+$z^j$x(k+p);
10.           X(k+p)=x(k)-$z^j$x(k+p);
11.         End If;
12.     End Do;
13. End Do;
```

Figure 2.5

A decimation-in-time FFT algorithm. This algorithm is presented in a paper by Clark [5] and is used in the implementation of the FFT network, and the Mesh configuration described in this chapter. The conditional statement found in line 8 is used to ensure proper mixing of the data elements.

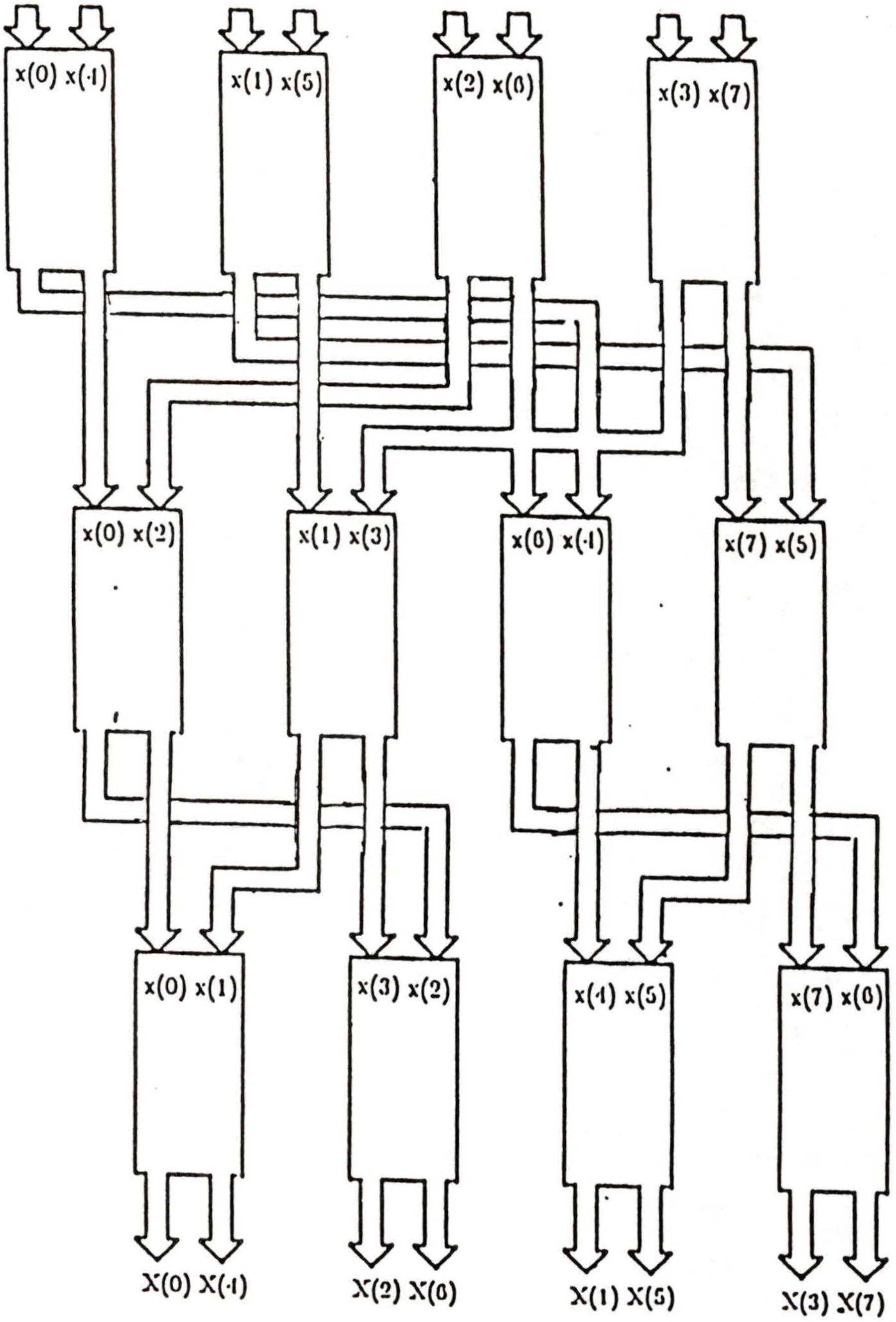


Figure 2.6

An eight element FFT network. This network implements an array of N/2 by $\log_2 N$ processing nodes. The interconnect pattern of the nodes is described in section 2.4.2. Notice the bit-shuffled order in the input, and the corresponding bit-reversed output.

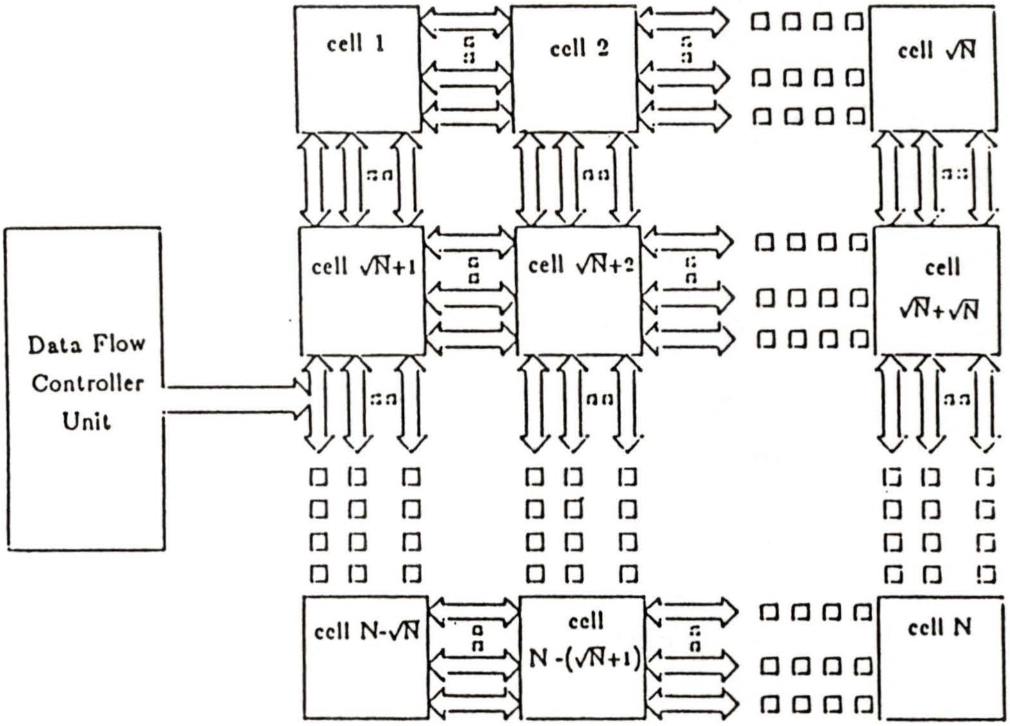


Figure 2.7

The Mesh configuration. This layout utilizes a mesh of N processors in an array of $\sqrt{N} \times \sqrt{N}$ dimensions. In this implementation each processor is responsible for computing an individual output element. Mesh data flow is extensive and is described in detail in section 2.4.3.

```
For M=log(N) - 1 to 0 step -1 Do
  p=$2^M$;
  q=N/p; /* note that N=pq */;
  z=$\omega^{q/2}$;
     For n=0 to N-1 Do
        j= n(mod q)
           If(n(mod2p))=j then do
              X(n)=x(n)+x(n+p);
              X(n+p)=$z^j$x(n)-$z^j$x(n+p);
           End If;
      End Do;
End Do;
```

Figure 2.8

The decimation-in-frequency FFT algorithm [5]. This algorithm is used to implement the Pipelined Cascade FFT implementation.

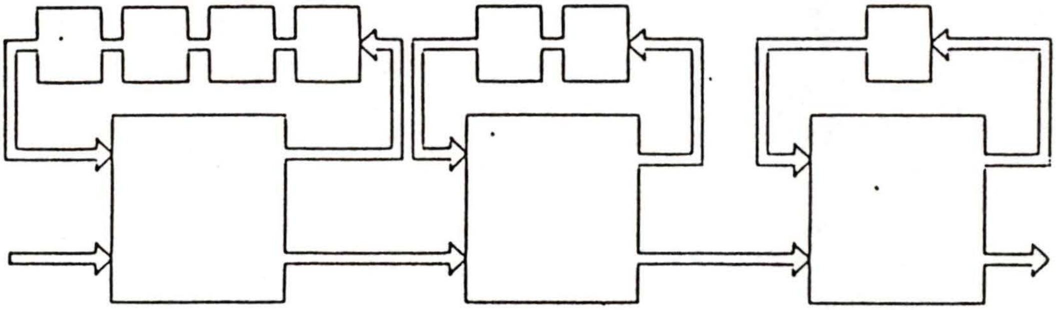


Figure 2.9

The Pipelined Cascade implementation for an eight- point FFT. The square elements represent the processing nodes, while each rectangular element represents a single word of memory.

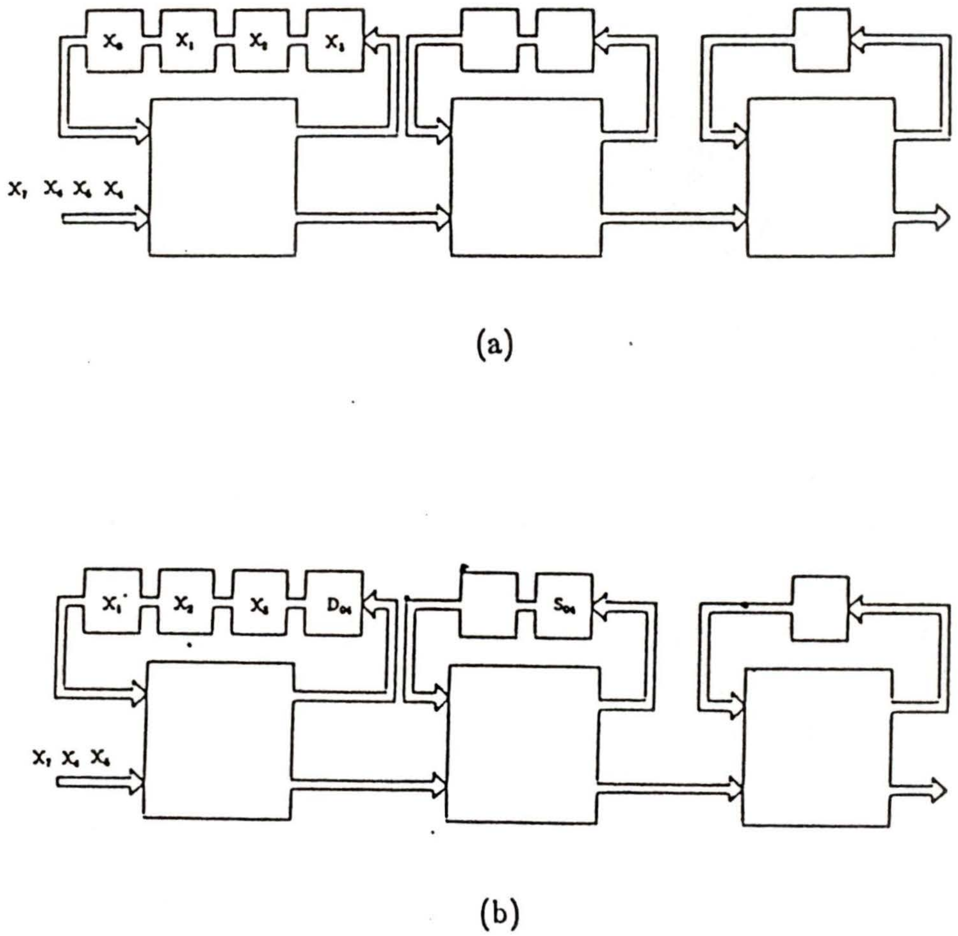
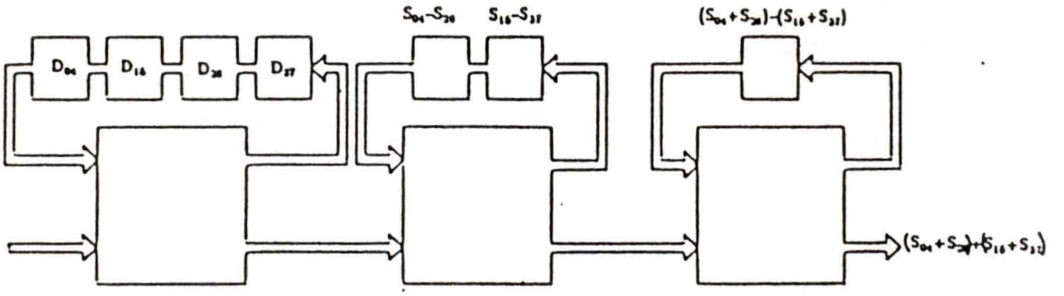
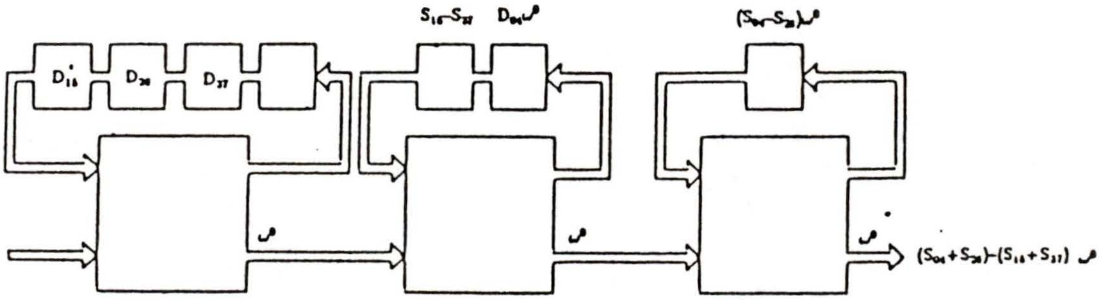


Figure 2.10

Data flow sequence for the Pipelined Cascade implementation. Figure 2.10(a) illustrates the point at which N/2 data elements have entered the network and been passed into shift registers. Figure 2.10(b) shows the data positions subsequent to the calculation of the first set of sum/difference terms. The term S_{ij} represents the sum of elements x(i) and x(j), while D_{ij} represents their difference.



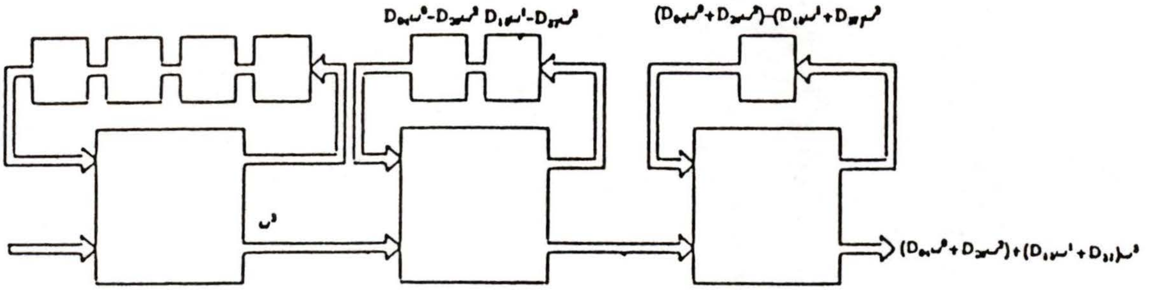
(c)



(d)

Figure 2.10 ctn'd

Figure 2.10(c) shows the data positions immediately after the last input element has been read in and the sum/difference calculation performed. Notice that at this point difference terms are sitting at the head of all the shift registers. Figure 2.10(d) shows the data after the first multiplication by the twiddle factor. This follows immediately after the step shown in figure 2.10(c).



(e)

Figure 2.10 cont'd

Figure 2.10(e) shows the location of the data immediately after the last element has passed out of the first processor. Notice the twiddle factor updating differs from stage to stage. The nature of this difference is described in section 2.4.4. Data flow continues in the manner described until the last element passes out of the last stage.

Implementation Performance			
Implementation	Area	Net Processing Time T_{Net}	Area-Time^2
Single Cell DFT	1	$9N^2T_m$	$81N^4T^2{}_m$
FFT Network	$(N/2)\log_2$	$10T_m\log_2N$	$50N(\log_2N)^3T^2{}_m$
Mesh Configuration	N	$(10T_m\log_2N+NT_{shift})$	$N(10T_m\log_2N+NT_{shift})^2$
Pipelined Cascade	$\log_2N$	$10NT_m$	$100N^2T^2{}_m\log_2N$

Table 2.1

AT^2 Performance factors for FFT and DFT implementations.

Chapter 3

Analysis of Rounding Methods and the Effects of Finite Register Lengths in the Fast Fourier Transform

3.1 Introduction

Computer processing of digitized signals requires that the signals be represented by a finite set of values usually in a binary format of finite register length. Digital computers employ a radix-2 or binary number representation, in which numbers are represented by a finite length sequence of 0's and 1's. While decimal representation employs a decimal point to separate the integer from the fractional part of a number, the binary representation utilizes a radix or binary point to achieve the same function. The radix point is located immediately to the right of the 2^0 bit.

Effects of a finite register length manifest themselves in several ways. One immediate consequence is that digital formats are restricted to representing only a finite number of values. Thus in the case of analog input signals, the digitization to a finite set of values results in a quantization of the original waveform. The level of this quantization is strictly dependent on the number of possible binary values representable, and thus on the length of the registers.

Problems associated with finite register lengths persist beyond the original digitization and continue to appear throughout subsequent numerical processing [10,11,12]. Often the processing operations produce digitized values requiring extended register lengths for numerical representation. As an example of this consider the multiplication of an n-bit number by an n-bit number. In binary representation this multiplication results in a product of length 2n bits. Further multiplications by n-bit numbers extend the product length by n bits for each multiplication performed. Thus the product length continues to grow. Truncation or rounding of excess bits is required in order to conform to register length restrictions. This introduces further uncertainty into the processing operations.

The magnitude of the uncertainty or error introduced by the affects of the finite register lengths is greatly dependent on the type of binary representation used, as well as the method of rounding employed [10]. For example, it will be shown that error introduced in a round toward zero or truncation operation is twice that associated with simple rounding. This chapter will investigate the affect of simple rounding and truncation on fixed and floating-point data formats. A statistical model for round-off error is reported, and analysis of this error is made as applied to the calculation of the discrete Fourier transform, using fast Fourier transform algorithms. The signal-to-roundoff noise ratio for this algorithm is then developed and compared for both fixed and floating-point arithmetic [10,12].

3.2 Fixed and Floating-Point Data Formats

3.2.1 Fixed-Point Formats

As mentioned previously the amount of error introduced by finite register lengths is dependent on the type of numerical representation employed. Two common methods of numerical representation are the fixed and floating-point formats. Fixed-point formats are based on the radix point remaining in a stationary position within the register. The means by which fixed-point binary arithmetic is implemented is directly dependent on the position of the fixed radix point within the register. The most common fixed-point data formats are the integer and fractional formats. In integer format the radix point is fixed to the extreme right of the least significant bit, while in the fractional format it is located to the left of the most significant bit.

The two main problems encountered in any fixed-point data format are that of range, and data overflow [13]. The range of numbers available from a given format is given by the extreme values that the numbers may take on. In the case of an n-bit register length the maximum absolute value the number may take, may not exceed 2^n -1. Thus for a 16-bit register length, a fixed-point integer format is restricted to representing integer values between 0 and $2^{16}-1$. Negative values of representation are generally indicated by means of a designated sign bit incorporated into the format or by a technique known as two's complementation.

Fixed-point arithmetic operations are highly susceptible to problems involving data overflow. In this case the result of some arithmetic operation surpasses the upper range bound of the format and information is then lost to the left of the most significant bit. This is a potential problem for both integer and fractional formats during binary addition and to the integer format during multiplication. One method of overcoming the problem of data overflow is suggested in a paper by Oppenheim [10]. This technique requires that the input data be small enough, so that the possibility of overflow is avoided. In his paper, Oppenheim makes the restriction that no numerical result may exceed 2^n-1 in magnitude, where n is the length of the register. An application of this method to the case of fixed-point FFT calculation is presented later in this Chapter. A more sophisticated technique for avoiding overflow problems is that of scaling [13]. Scaling essentially analyses the mathematical variables and transforms them so that the mathematical result is assured of falling within the range of a given format. This technique requires that a running account of the variables magnitudes be maintained throughout the processing procedure, so that proper retransformation is ensured. Overall scaling is a complex technique, and is much more amenable to software applications than to direct hardware implementation.

In fixed-point arithmetic the operation of binary addition is independent of the location of the radix point, and requires only that its relative position be the same for all data values. In this way fractional binary addition is indistinguishable from integer binary addition. In the case of multiplication however the location of the binary point affects the outcome of the result, where in general the product of two numbers of length b bits is 2b-bits long. Thus if the radix point is located n bits to the right of the most significant bit in the multiplicands it will be shifted an additional n bits to the right in the product. In the case of integer arithmetic the product remains in integer format and in fractional arithmetic the product will remain fractional. In order to accommodate the restriction of fixed register lengths to the multiplication operation, it is usually necessary to approximate the 2b-bit product by a b-bit result. In the case of integer arithmetic this is quite difficult and requires that either scaling techniques be employed on the product or that the original multiplicands must be strictly bound. The problem of data overflow in multiplication does not arise when using a fractional number format. Overflow can never occur in this case since the product of two fractions is always a fraction. Approximation of the 2b-bit product is easily realized by rounding or truncating the last b bits. For this reason fractional fixed-point format is often preferred over integer fixed-point format for digital processing applications [13].

3.2.3 Floating-Point Representations

The alternative to fixed-point formats is that of floating-point formats. Floating-point number formats are generally comprised of a sign bit, s, a set of exponential bits, E, followed by a set of mantissa bits, M. The mantissa is generally in a fractional form. Some standards however, for example the IEEE standard, employ a hidden bit which is implicitly set to a 1 immediately to the left of the radix point. In general, a number N may be represented as:

$$N = (-1)^s \times 2^E \times (1.M)_2 \tag{3.1}$$

It is common practice to positively bias the exponential term by some amount, generally 2^{k-1}, where k is the number of bits in the binary exponent. In this way a k bit binary exponent ranging from $-(2^{k-1}-1)$ to $+(2^{k-1}+1)$ is expressed as some positive biased value e. The true exponential value is obtained by subtracting E-e.

In floating-point representations the mantissa is always maintained in a normalized form,. That is its leading bit is always a 1. At any time during processing should a result be formed in which this is not the case, the binary point is shifted to the immediate left of the leading 1, and the exponent is incremented or decremented accordingly. This process is known as normalization, and the mantissa element is referred to as being "left justified".

Floating-point representations are particularly well suited to carry out binary multiplication. In this operation the two mantissas are multiplied as in fixed-point arithmetic, and the two exponential terms are added together. Since the product of two mantissas of the form seen in equation (3.1) will always be between 1 and 4, normalization is usually required. The addition operation for floating-point arithmetic is somewhat more involved.

Before binary addition of floating-point numbers can be carried out, proper alignment of the mantissa's radix points must be ensured. This alignment requires that their respective exponential terms be equal. To carry out floating-point addition the difference between the exponents is first calculated, and the mantissa of the smaller term is shifted to the right accordingly until the difference is realized. Once this operation has been performed and the exponents are equal, addition of the mantissas is carried out just as in the case of fixed-point arithmetic. The sum is then normalized, and the larger exponent term is readjusted. At present several floating-point standards are available. Figure 3.1 (a-c) illustrates some of the most common floating and fixed-point formats available. Finer points regarding negative number representation and exponential biasing, are explained in the figures.

Overall, the exponential terms of floating-point representations provide for a far greater range than that attainable by fixed-point representations of equal register lengths. This fact greatly reduces any problems incurred by data overflow, and in most cases eliminates the problem altogether. A fractional mantissa is easily rounded or truncated in order to comply with any register length restrictions. As in fixed-point formats however, this introduces a finite roundoff error into the processing calculations.

3.3 Roundoff Error Modeling

In this section, mathematical formulae will be developed to describe the roundoff error associated with basic rounding and truncation techniques [10,12]. Both fixed and floating-point numbers will be considered. The fixed-point numbers and floating-point mantissas will be represented as binary fractions of length n+1 bits. The extra bit, is located to the immediate left of the radix point and is used in negative number representation both in the case of sign-magnitude and in two's complement representation. In fractional form, the decimal value of the least significant bit is given by 2^{-n}. This is the minimum value by which the fractional term can be incremented or decremented, and is referred to as the quantization step. We consider first the effects of truncation and rounding as applied to positive and negative fixed-point numbers. Analysis of floating-point roundoff and truncation errors is dealt with in section 3.3.2

3.3.1 Fixed-Point Error Analysis

Truncation Error Analysis

Of all the available rounding methods truncation is the easiest to implement. It is achieved by simply truncating or chopping the number to the desired register length and ignoring or masking any of the lesser significant bits. The error due to truncation is taken as the difference between the number's value after truncation and its value before truncation. Thus for positive numbers, if n is the total number of bits before truncation, and m is the number of bits after, the associated truncation error is given by:

$$E_T = \sum_{i=1}^{i=m} a_i 2^{-i} - \sum_{j=1}^{j=n} a_j 2^{-j} \tag{3.2}$$

$$= - \sum_{i=n+1}^{i=m} a_i 2^{-i}$$

where, $a_i = 0$ or 1. Notice that this value is always negative when dealing with positive numbers. Truncation therefore serves to reduce the value of the numbers and for this reason is often referred to as a "round toward 0" technique. Bounds on the limit of the truncation error may be determined if we consider a worst case situation, (i.e. $a_i=1$). Thus we have:

$$E_T = - \sum_{i=m+1}^{i=n} 2^{-i} \tag{3.3}$$

Recognizing that we may express a number 2^{-m} as:

$$2^{-m} = 2^{-(m+1)} + 2^{-(m+2)} + ... + 2^{-(n-1)} + 2^{-(n)} + 2^{-(n)} \tag{3.4}$$

$$= \sum_{i=m+1}^{i=n} 2^{-i} + 2^{-n}$$

and substituting into equation (3.3) for $\sum_{i=m+1}^{i=n} 2^{-i}$, we get

$$| E_T | \leq | - (2^{-m} - 2^{-n}) | \leq | 2^{-m} | \tag{3.5}$$

From our original claim that E_T is negative under these conditions the limits are found to be:

$$0 \geq E_T \geq - (2^{-m} - 2^{-n}) \tag{3.6}$$

For example if we consider the truncation of a 23-bit mantissa to one of 14 bits, the associated truncation error is found to be less than or equal to 2^{-14} or 0.000061.

In terms of negative number representation by means of a sign bit, truncation increases the number. Thus for negative numbers in a sign magnitude format, E_T is always positive, and its limits are given by:

$$0 \leq E_T \leq 2^{-m} - 2^{-n} \tag{3.7}$$

In two's complement representation of negative numbers, a somewhat different approach is taken. This is due to the fact that the leading bit is no longer simply a sign bit, but the most significant bit of a numerical representation. In two's complement a negative fractional number is represented as $1.a_1\, a_2\, a_3 \cdots a_m$ and its magnitude by definition is taken to be:

$$\mathrm{MAG}_1 = 2.0 - \mathrm{COMP} \tag{3.8}$$

The two's complement of a number, referred to here as COMP may be expressed as:

$$\mathrm{COMP} = 1 + \sum_{i=1}^{i=m} a_i 2^{-i} \tag{3.9}$$

where $a_i = 0$ or 1. Truncation to m bits in this case produces a new magnitude value of:

$$MAG_2 = 2.0 - COMP_T \tag{3.10}$$

Thus the change in magnitude between the original and truncated value is given by

$$\Delta MAG = MAG_2 - MAG_1 = \sum_{i=m+1}^{i=n} a_i b^{-i} \tag{3.11}$$

where m is the bit length of the truncated element. Notice that this value is always positive. Thus truncation of two's complement negative numbers tends to increase their magnitude thus making E_T negative. In the case of negative two's complement representations the error limits may be given by:

$$0 \geq E_T \geq -(2^{-m} - 2^{-n}) \tag{3.12}$$

Roundoff Error Analysis

Roundoff errors due to basic rounding techniques are independent of negative value representations. This is due to the fact that basic rounding proves to be symmetric about the bit in question for both positive and negative values. The direction of rounding, that is up or down, is determined through the analysis of bits discarded after truncation. Should the value of these bits be greater than one half the new width of quantization, the number is rounded up, otherwise the number is left in truncated form. This determination is easily realized by investigating the nature of the bit immediately to the right of the bit to be rounded. This bit is known as the first guard bit. A high value indicates the value of the discarded bits are greater than or equal to half the quantization value, and thus the original number is closer to the next incremented value. The number is then rounded upwards by adding a 1 to the rounding bit position. Conversely a low guard bit indicates the original value is closest to its truncated value and the number is left in truncated form. In this context the error due to rounding proves to be one half the difference between the value before and after truncation. Thus:

$$\mathrm{E_R} = 2\mathrm{E_T} \tag{3.13}$$

By extending the relationship described in equation (3.5) we can express $\mathrm{E_R}$ as:

$$-\frac{1}{2}2^{-m} \leq \mathrm{E_R} \leq \frac{1}{2}2^{-m} \tag{3.14}$$

The graph in figure 3.2 gives the estimated roundoff and truncation error as a function of register length based on a 23-bit fractional number.

3.3.2 Floating-Point Roundoff Analysis

In the case of floating-point arithmetic, roundoff or truncation effects are only manifest in the mantissa. In as much as the mantissa may be treated as a fixed-point fractional number the magnitude and limits of these errors is identical to those previously determined. In view of future analysis however it will prove convenient to describe this error in a multiplicative sense as opposed to an additive one as in the case of fixed-point numbers. In order to achieve this we let x represent the value of the number before rounding or truncating, and designate some value Q(x) to represent the value after rounding. Q(x) is then given by:

$$Q(x) = x + x\epsilon = x(1+\epsilon) \tag{3.15}$$

where $x\epsilon$ is the associated roundoff or truncation error.

As was pointed out in the previous paragraph, the limits on the roundoff and truncation errors in the mantissa elements will be identical to the case developed for the fixed-point analysis. Referring back to equation (3.1) however, we see that the final magnitude of the floating-point representation is obtained by multiplying the mantissa term by some exponential value, 2^E. Thus any error associated with the mantissa must also be scaled by a factor of 2^E in the final representation. This is easily seen by substituting for the mantissa term 1.M in equation (3.1) with the terms $(1.M_R \pm E_R)$ or $(1.M_T \pm E_T)$ where M_R and M_T represent the rounded and truncated mantissa forms. By extending our previous results for fixed-point rounding to the floating-point case, we have:

$$-\frac{1}{2}2^{-m}\times 2^{E} \leq E_R \leq \frac{1}{2}2^{-m}\times 2^{E} \tag{3.16}$$

where E_R in this case represents the net roundoff error for the floating point representation. Now substituting equation (3.15) for E_R in (3.16) we get:

$$-\frac{1}{2}2^{-m}\times 2^{E} \leq Q(x) - x \leq \frac{1}{2}2^{-m}\times 2^{E} \tag{3.17}$$

or,

$$-\frac{1}{2}2^{-m}\times 2^{E} \leq x\epsilon \leq \frac{1}{2}2^{-m}\times 2^{E} \tag{3.18}$$

Now since

$$2^{E-1} \leq x \leq 2^{E} \tag{3.18}$$

for the case of a fractional mantissa, it follows from equation (3.17) that:

$$-2^{-n} \leq \epsilon \leq 2^{-n} \tag{3.19}$$

A similar procedure is used to convert the truncation errors in floating point format to a multiplicative expression.

3.4 Roundoff Effects in FFT Calculations

FFT algorithms in general play a central role in many signal processing applications [1]. At present there exist many forms of the algorithm from which to choose, and the effects of quantization or roundoff differ for each of them. The most common of these are the radix-2 decimation-in-time and frequency algorithms. Recall from Chapter 1 that the FFT calculates an N-point DFT in $\log_2 N$ stages. Each stage in turn processes the entire N-point array of complex values by passing pairs of values through the basic computational unit known as the "butterfly node". We designate each of the $\log_2 N$ stages by a subscript m where m runs from 0 to $(\log_2 N)-1$. The butterfly calculation of the m+1 stage is given as:

$$X_{m+1}(i) = X_m(i) + \omega_N{}^i X_m(j) \tag{3.20}$$

$$X_{m+1}(j) = X_m(i) - \omega_N{}^i X_m(j) \tag{3.21}$$

for the case of decimation-in-time, and

$$X_{m+1}(i) = X_m(i) + X_m(j) \tag{3.22}$$

$$X_{m+1}(j) = [X_m(i) - X_m(j)]\omega_N{}^i \tag{3.23}$$

for decimation-in-frequency. Thus in each of the $\log_2 N$ stages a total of $\frac{N}{2}$ butterfly calculations are performed, requiring a total of $\frac{N}{2}$

complex-number multiplications per stage. The subsequent analysis will consider the decimation-in-frequency algorithm. Modification of the results to the case of decimation-in-time are easily made as the differences between the two basic butterfly structures are slight. Even then the basic dependence of the signal-to-noise ratio on N remains essentially the same for both cases [12]. The following analysis is based on previously reported analysis by Oppenheim, Knuck, and Weinstein [10,11,12].

The analysis of roundoff noise considers only fractional mantissas and fixed-point formats. The analysis is also made in terms of rounding as opposed to truncation. This is due to several reasons. An assumption is made that the variable E_R is random within its bounding limits, and is thus independent of the value of X(k). This assumption fails to hold in the case of truncation, where the value of E_T is directly correlated to the sign of X(k). The second reason for considering only rounding noise is that E_R is symmetrically bound between $\pm\frac{1}{2}2^{-n}$ and thus the mean roundoff noise is zero. In the case of truncation the limits are not symmetric about zero, and thus the mean round off noise is not zero.

In the case of floating-point representations the variable ϵ is considered random, and independent of X. Recall that for basic rounding it is bound between the limits of $\pm 2^{-n}$. The assumption is also made that ϵ is uniformly distributed between these bounds with a roundoff variance of σ_{ϵ}^{2}. Recent empirical studies on the effects of roundoff noise on the FFT algorithm [10,12], have estimated this value to be approximately:

$$\sigma_{\epsilon}^{2} \approx \frac{1}{12} 2^{-2n} \tag{3.24}$$

Similarly in the case of the input signal, a uniformly distributed white noise model is assumed. Its variance is assumed to be:

$$\sigma_{x}^{2} \approx \frac{1}{3} 2^{-2L} \tag{3.25}$$

where L in this case represents the register length of the original input element.

3.4.1 Fixed-Point Analysis

In executing the radix-2 algorithm the transmission of any intermediate results between adjacent butterfly nodes involves multiplication by a complex constant, the twiddle factor. This is seen in the FFT flow graph in figures 1.2-1.5. Each complex-number multiplication is made up of four real multiplications each of which are rounded independently. Based on our previous assumption that E_R is uncorrelated with variance σ_ϵ^2, we treat each real multiplier operation as an independent noise source. Thus the total variance associated with each complex-number multiplication, σ_C^2, may be given by:

$$\sigma_C^2 = 4\sigma_\epsilon^2 = 4\frac{2^{-2n}}{12} \tag{3.26}$$

where σ_ϵ^2 is substituted for from equation 3.24.

In general it can be shown by inspection of the FFT subroutines or flow graph found in figure 1.5 that a total of N-1 complex multiplications are required in the determination of each output value. Since we have assumed all signal and noise sources to be uncorrelated, the noise variance at each output node is given as the sum of the variances of all the noise sources propagating to the output node. Thus the expected total output noise variance may then be given by:

$$\sigma_E^2 = (N-1)\sigma_C^2 \tag{3.27}$$

For large N we can approximate (3.25) by:

$$\sigma_E^2 = N\sigma_C^2 \tag{3.28}$$

We see that the output noise then is directly proportional to the number of points transformed. If N is doubled, and another stage is added to the FFT, the output noise variance is doubled accordingly. This development is readily extended to the magnitude of roundoff errors given by equations (3.5) and (3.13) [12]. The net magnitude of output noise due to rounding may be given as:

$$E_R(\mathrm{out}) \approx N E_R \tag{3.29}$$

Figure 3.3 shows the ratio of input quantization to output noise magnitude or net relative uncertainty for several roundoff values n, as a function of the sample number N.

Closer inspection of our assumptions reveal weak points in the previous analysis. The primary weakness lies in the assumption that each of the butterfly nodes produces noise of equal variance. In some cases however the multiplications may be performed noiselessly for example when $\omega=1$ or i. Taking this fact into account, the actual output noise variance will be slightly less than the value stated in equation (3.27). A second problem arises in the assumption that the outputs of the noise sources are uncorrelated. In fact, the outputs of the butterfly nodes are negatives of one another and as such are totally correlated. This does not affect the output variance however since each of the two butterfly outputs propagates to a different output node. It does imply however some degree of correlation between the output noise variance at the various output nodes.

Recall from previous discussions that the major problem with fixed point formats was that of data overflow. One possible means of controlling this problem is to ensure input values small enough so that overflow would not occur. We now extend this method to the relationships given in equations (3.19) - (3.22), and we begin by establishing the following constraints.

$$\max [\,|X_m(i)|\,] \leq \max [\,|X_{m+1}(i)|\,] \tag{3.30}$$

$$\max [\,|X_{m+1}(i)|\,] \leq 2\max [\,|X_m(i)|\,] \tag{3.31}$$

$$\max [\,|X_m(j)|\,] \leq \max [\,|X_{m+1}(j)|\,] \tag{3.32}$$

$$\max [\,|X_{m+1}(j)|\,] \leq 2\max [\,|X_m(j)|\,] \tag{3.33}$$

These constraints ensure that the maximum modulus of each intermediate element in the FFT is nondecreasing from stage to stage. Thus if the maximum magnitude of the output of the FFT is less that one, as must be the case for a fractional format, then the magnitude of any intermediate stage element is necessarily less than one. In this event overflow is prevented throughout the FFT processing stages.

In order to establish an appropriate bound on the input sequence, we recognize that we may express the maximum possible output,

$X(k) \mid_{\max}$ from the FFT as:

$$X(k) \mid \max \leq \sum_{n=0}^{n=N-1} \mid x(n) \mid \; \mid \omega^{nk} \mid \tag{3.34}$$

or,

$$X(k) \mid_{\max} \leq \; \mid x(n) \mid_{\max} \sum_{n=0}^{n=N-1} \mid \omega^{nk} \mid \tag{3.35}$$

where $\mid x(n) \mid_{\max}$ is the magnitude of the largest input element. Now since $\mid \omega^{nk} \mid = 1$ we have:

$$X(k) \mid_{\max} \leq N(x(n)) \mid_{\max} \tag{3.36}$$

Thus a maximum output X(k) bound by unity requires that the input sequence x(n) be bound by $\frac{1}{N}$.

$$\mid x(n) \mid \; \leq \frac{1}{N} \tag{3.37}$$

An expression for the output signal variance, $\sigma_X{}^2$ is obtained by employing our initial input signal assumptions and following a development analogous to the case of the output noise variance. The output signal variance is thus given as:

$$\sigma_X{}^2 \approx N\sigma_x{}^2 = N\overline{\mid x(n) \mid ^2} \tag{3.38}$$

Now applying equation (3.25), and assuming x(n) white with real and imaginary parts each uniformily distributed in $(1/\sqrt{2}N, 1/\sqrt{2}N)$ we have:

$$\sigma_X^2 \approx \frac{1}{3N} \tag{3.39}$$

Thus the output noise to signal variance $\frac{\sigma_E^2}{\sigma_X^2}$ may be given as:

$$\frac{\sigma_E^2}{\sigma_X^2} \approx 3N^2\sigma_C^2 \tag{3.40}$$

It is apparent then that the ratio of the noise-to-signal variance is proportional to N^2.

As mentioned an alternative method for the prevention of overflow was that of scaling. One scaling method described by Oppenheim and Weinstein [10], has proven to be quite suitable for overflow prevention in FFT calculations. Essentially the technique is based on the fact that the maximum modulus increases by no more than a factor of two per stage. In this case overflow is prevented by requiring that $|x(n)| < 1$ and by attenuating the output of each stage by a factor of two. By employing a stage by stage scaling on a white input signal, Oppenheim and Weinstein obtained a noise-to-signal variance ratio of:

$$\frac{\sigma_E^2}{\sigma_X^2} = 5N(2^{-2n}) \tag{3.41}$$

Derivation of this result is quite complicated, and the reader is referred to the above mentioned reference for a detailed derivation of this result.

It is important to note that the assumption that the input signal be white is not essential to these derivations. In general the results remain proportional to N and N^2, without this constraint.

3.4.2 Floating-Point Analysis

As in the case of the fixed-point analysis, roundoff error in floating-point systems is introduced due to each butterfly computation. The nature of this noise however, is somewhat more complex than in the fixed point case. In fixed-point analysis the noise source associated with each butterfly is assumed to be introduced solely from the multiplier operations. In contrast, the floating-point analysis considers the addition and subtraction operations as independent noise sources as well.

This difference is due to the fact that in fixed-point FFT's, input elements are bound in magnitude so as to preclude any possibility of data overflow. This restriction eliminates the necessity for roundoff in the addition or subtraction operations, thus removing them as potential noise sources within the butterfly operations. In the case of floating-point numbers however, the large dynamic range made available by the format greatly relaxes any bound on the magnitude of the input variables. Addition and subtraction of two variables whose magnitudes differ by more than an order of magnitude are now possible. In this event rounding or truncation of the shifted mantissa element is necessary and thus new sources of noise variance are introduced into the butterfly operation For this reason the floating-point roundoff noise analysis is somewhat more complex than that for fixed-point formats. Analysis is even further complicated unless the input signal source is assumed to be white. For a more detailed theoretical and experimental analysis of the effects of a nonwhite input signal the reader is referred to Kaneko and Liu [14].

By assuming a white input signal, Oppenheim and Weinstein [10] developed a floating-point noise model describing the butterfly operation. They were able to show that the variance resulting from a butterfly operation performed in the m^{th} stage could be given as:

$$\overline{|U_m|^2} = 4\sigma_\epsilon^2 \overline{|X_m|^2} \tag{3.42}$$

where $\overline{|U_m|^2}$ is the variance introduced by a butterfly node at stage m, and $\overline{|X_m|^2}$ is the variance in the input signal to that node. The term σ_ϵ is as defined in equation (3.24). Since the input signal variance to array m+1 is the output noise variance from stage m, we see that the noise generated in computing the $(m+1)^{th}$ stage is $4\sigma_\epsilon^2$ times the signal variance of the m^{th} stage. By assuming a white noise input signal of variance σ_x^2 the value of $\overline{|X_m|^2}$ can be derived from equations (3.32) and (3.33) and found to be:

$$\overline{|X_m|^2} = 2^m\sigma_x^2 \tag{3.43}$$

Thus the noise generated in the $(m+1)^{th}$ array is:

$$\overline{|X_m|^2} = 2^m\sigma_x^2(4\sigma_\epsilon^2) \tag{3.44}$$

If we now define σ_{om}^2 as that component of the net output noise due to the noise generated in the $(m+1)^{th}$ array, we have:

$$\sigma_{om}^2 = 2^{\nu-(m+1)}2^m\sigma_x^2 4\sigma_\epsilon^2 = 2N\sigma_\epsilon^2\sigma_x^2 \tag{3.45}$$

where $\nu = \log_2 N$. This relationship is perhaps more clearly visualized if we consider that for the later stages, say $m=\nu-1$ the term $2^{\nu-(m+1)}$ is fairly small. This indicates that the noise generated in the later stage arrays contributes a smaller fraction of roundoff noise to the net output noise variance than say the noise generated in the first array which contributes and propagates its noise component through each of the subsequent arrays.

By assuming as was done in the case of the fixed-point numbers that the output noise from each array is independent, the total output noise variance may be given by:

$$\sigma_E^2 = 2\nu N \sigma_\epsilon^2 \sigma_x^2 \tag{3.46}$$

Now recalling from equation (3.35) that:

$$\sigma_X^2 \approx N \sigma_x^2 \tag{3.47}$$

the output signal-to-noise variance is given by:

$$\frac{\sigma_X^2}{\sigma_E^2} = \frac{1}{2\sigma_\epsilon^2 \nu} \tag{3.48}$$

The result of equation (3.48) can be taken yet a step further in order to determine the output signal-to-noise ratio. This value is obtained simply by performing an FFT, and an inverse FFT in series on the same white input signal x(n). Since the inverse FFT introduces as much roundoff noise variance as the FFT the resulting output noise is found to be $2\sigma_E$. Therefore the resulting output signal-to-noise ratio is given by:

$$\frac{\sigma_x^2}{\sigma_E^2} = \frac{1}{4\sigma_E^2 \nu} \tag{3.45}$$

The graph in figure 3.4 shows the output signal-to-noise ratio in dB as a function of ν for several register lengths n. From this graph it is easily seen that for reduced register lengths, between 12 and 23 bits long, excellent signal-to-noise ratios (>60 dB) for relatively large values of N are obtained. These results indicate that for given operating constraints it may be feasible to work with shortened mantissa lengths. Shortened mantissa lengths would serve not only to reduce the complexity of the arithmetic operators, as well as reduce the silicon real estate requirements, but would greatly speed up the processing time.

3.5 Summary

In this chapter a review was made of fixed and floating-point number formats, and the affects of roundoff and truncation were presented for both cases. For truncation of fixed-point values the truncation error E_T was found to be, $E_T \leq 2^{-m}$, where m is the register length of the truncated representation. In the case of roundoff noise, this error is reduced by a factor of two. For floating-point formats, it was shown that the magnitude of the roundoff or truncation error was given by the product of the mantissa error and the exponential term. Thus the roundoff error associated with floating-point formats will always be equal to or less than the error associated with a fixed-point format of the same register length as the floating-point mantissa. For this reason as well as for the greater dynamic range offered by floating-point representations the decision was made to implement the pipelined cascade network using floating-point data format. The format chosen was the IEEE standard 32-bit floating-point format found in figure 1.1. As indicated, this format is made up of a 23-bit mantissa which employs an implicit leading 1, and a designated sign bit. The eight-bit exponent is positively biased by 127.

In section 3.4, a detailed analysis was presented on the effects of finite register lengths in FFT calculations for fixed and floating-point representations. Equations describing the output signal-to-noise variance ratio's were developed for both cases. Also determined was an expression for the output signal-to-noise ratio as a function of register length n, and FFT size N. This relationship is significant in that it allows one to determine the minimum register length needed to conform with prescribed design specifications. In the case of this implementation, a signal-to-noise ratio of $>$60dB for a 1024 point FFT was desired. While the FFT size was arrived at somewhat arbitrarily, the signal-to-noise output of 60dB appears to be a standard value for digital signal instrumentation [15]. In order to comply with this specification, a mantissa register length of 14 bits is needed. This may be achieved simply by truncating the IEEE standard mantissa length from 23 to 14 bits. In so doing processor size requirements are reduced and processing speed is increased, greatly enhancing the overall network performance.

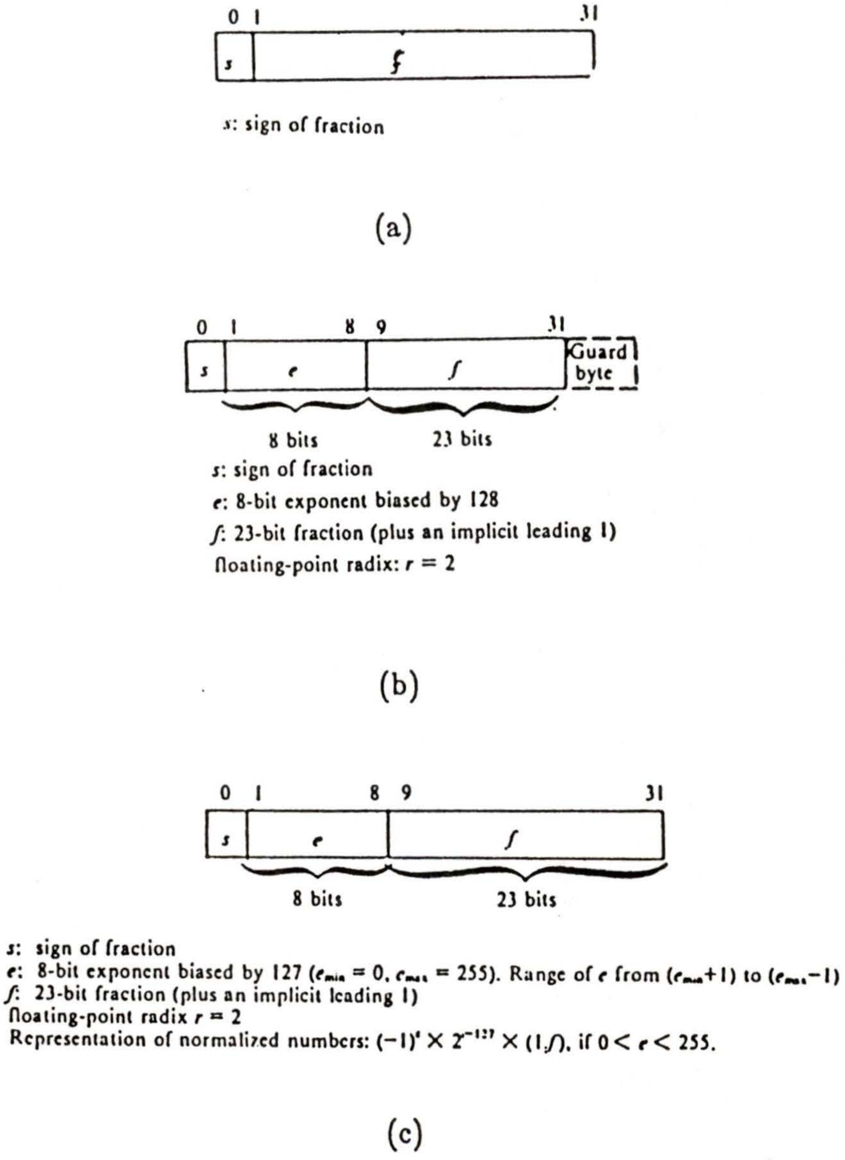


Figure 3.1

Fixed and floating-point formats.
Figure 3.1(a) shows a simple fixed-point 32-bit fractional format. Figures 3.1(b,c) illustrate the DEC and IEEE 32-bit floating-point formats respectively.

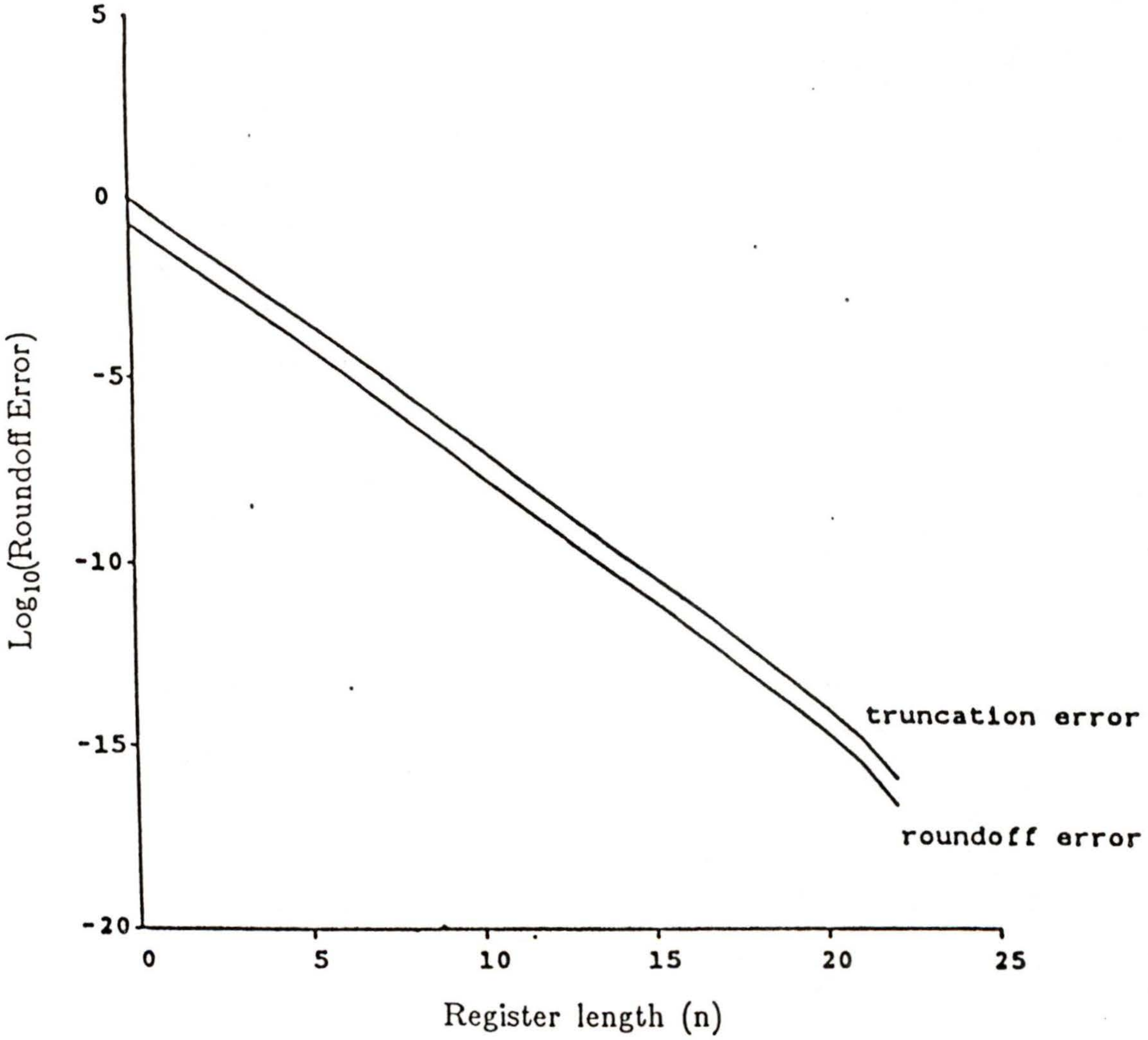


Figure 3.2

Graph of Log$_{10}$(roundoff error) vs. register length (n) for the cases of simple rounding and truncation.

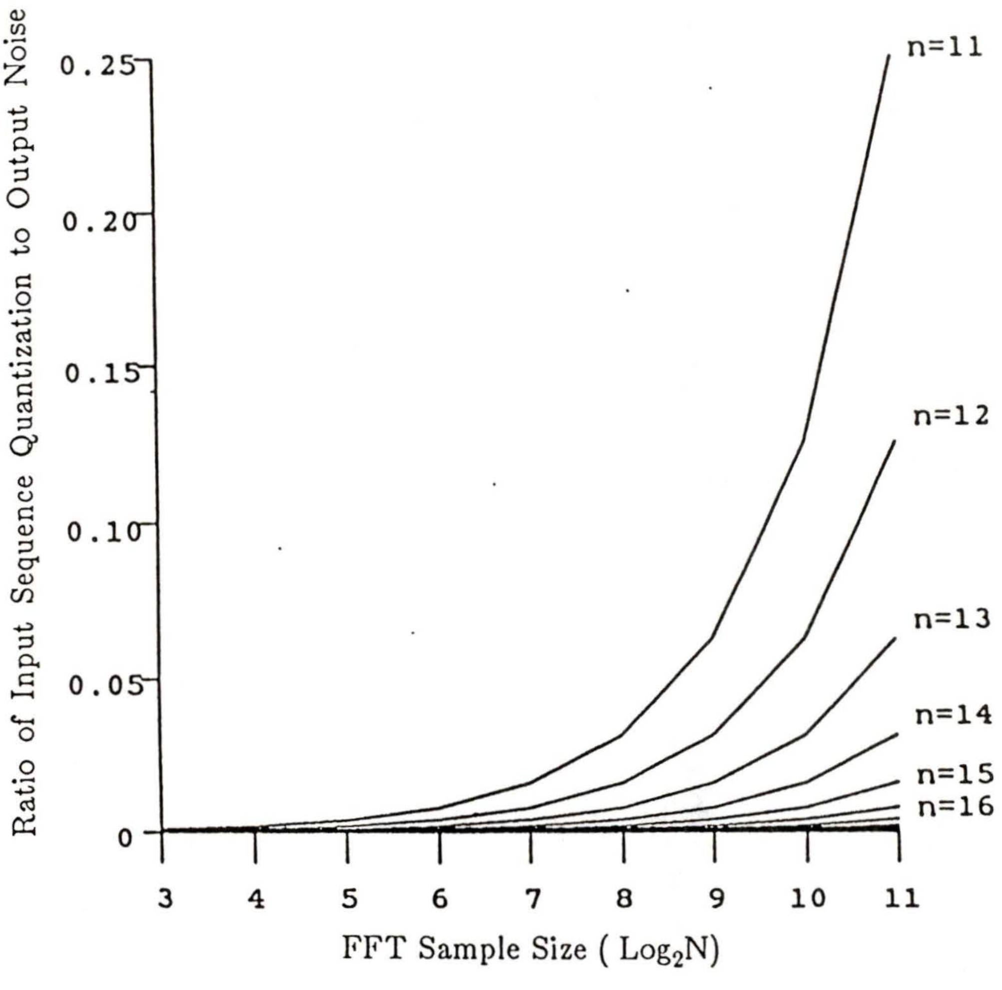


Figure 3.3

Graph of the ratio of input sequence quantization to net FFT output noise as a function of FFT sample size (N), and register length (n).

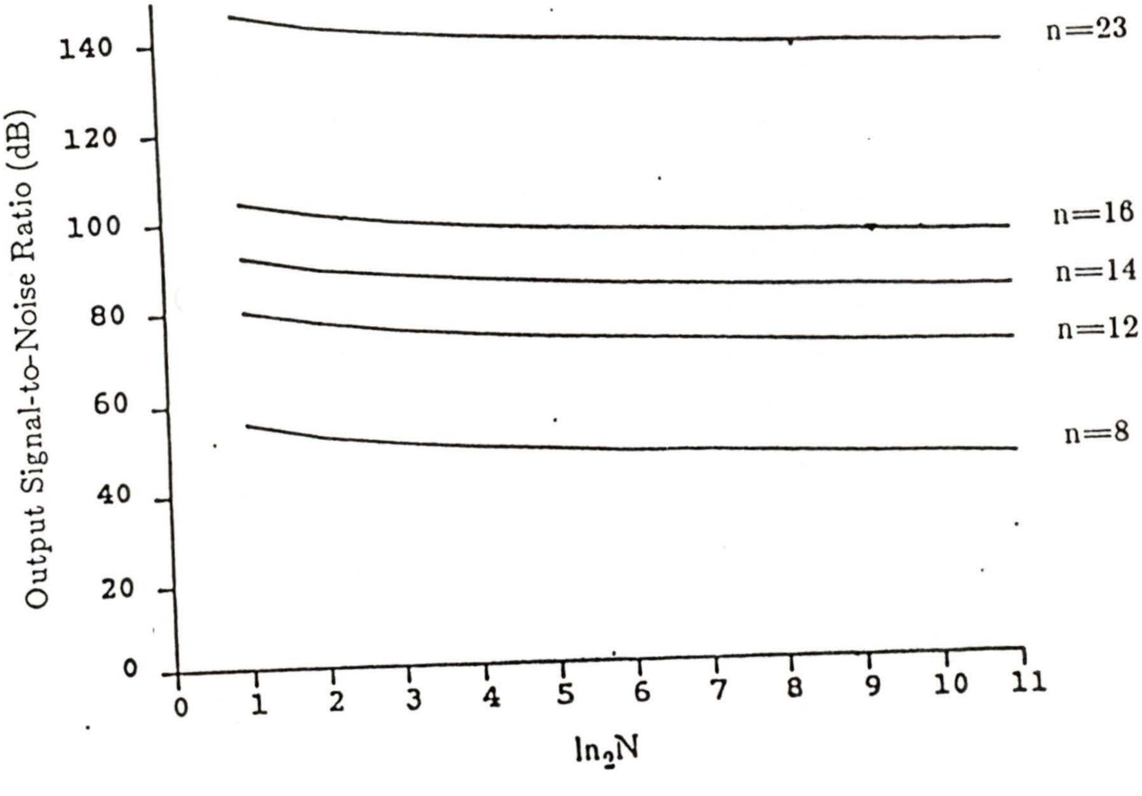


Figure 3.4

Graph of the output signal-to-noise ratio (dB) as a function of FFT sample size (N) and for several register lengths (n), for the case of floating-point formats.

Chapter 4

Component Design and Performance

4.1 Introduction

Of prime consideration in the implementation of any parallel or pipelined processor network is the design of the processing nodes. In Chapters 1 and 2, a suitable algorithm and processing network for implementation was determined. Chapter 3 established the necessary data format requirements needed in order to meet given design specifications. In this chapter the design of the FFT processor node is described.

The processing node described in this chapter is general-purpose in nature in that the processor is capable of operating within any of the previously-described DFT or FFT processing networks. For operation in the parallel pipelined cascade implementation, the nodes require a network clock signal and an indication as to the location of the processing stage in the network. This information is given to each node by a set of coded external control lines. The information is then read by the controller unit of each node, which is responsible for sequencing the processor operations. Design of the node control unit is discussed in more detail in Chapter 5.

Section 4.2 of this chapter investigates a design methodology used to deal with the complexity of VLSI structures. A discussion of node specifications and the overall logic design process follows in section 4.3. The bulk of this chapter deals with the physical design and performance of the system on a component level. This description is found in section 4.4. Also presented in this section, is the multiple access pipeline structure and a fast floating-point APU.

4.2 Design Methodology

As mentioned previously the criteria distinguishing VLSI circuits from lower integration levels does not rest solely with chip size and component density. A major difference also lies in the design methodology, CAD tools used, organization, and programming of the highly concurrent structures associated with VLSI [15]. The advent of VLSI has brought with it great increases in the degree of circuit complexity made available to hardware designers. In order to cope with this complexity, highly structured design methodologies have been developed. This section discusses in detail the procedures involved in a hierarchal VLSI design methodology.

The outset of any hardware design project begins with a determination of an initial set of specifications governing the project. Often these specifications are incomplete or in some cases unachievable. During the course of the project the designer is required to make various design decisions with respect to both the logic design and the physical design of the hardware. These decisions often result in modifications to the original specifications. Thus a cyclic pattern of decisions and modifications begins to emerge until a final design is realized. Figure 4.1 depicts the nested interactions of these steps.

The ideal design process is comprised of essentially three concurrent design operations, behavioral design, structural design and physical design. Behavioral design refers to the decomposition of the initial design specifications into simpler subdesigns and subspecifications. This decomposition process is often referred to as a top-down design technique in that the design progresses from the overall design specifications down to those of the constituent components. Using the top-down and bottom-up approaches, objectives at higher levels become the design specifications for the lower-level designs. This type of design allows for an iterative redefinition of the original design specifications.

The second design operation, structural design, is defined as the realization of a logical block or structure through the interconnection of more primitive structures. At some point in the project design a mapping from the behavioral hierarchy into the structural domain is required. This mapping is referred to as the logic design process. It is important to note that different behavioral descriptions may be associated with each module on each level of structural design. At each level of the structural design one needs to verify that the intended behavior of a given structure actually matches the performance of the proposed structure. Such a verification is often accomplished using a high-level logic simulator such as "HILO" or "Hg". A review of these simulators is made in Chapter 6.

The physical design operation refers to the actual realization of a structure in a given technology. The physical design often utilizes a bottom-up design approach by combining lower-level physical designs to realize higher-level ones. As in the previous case a mapping of the structural hierarchy into the physical domain is required at some time during the design process. This mapping is known as the physical layout process. During this stage of design, extensive use is made of design automation tools and CAD programs. In this project, a Metheus VLSI workstation equipped with the Berkeley design tools was used. Also on this level detailed simulations using SPICE were performed to allow for a detailed behavioral description of the circuit.

In summary the major advantages of a hierarchal design scheme is reduction of design time and handling of complexity. Often a top-down planning approach is taken in determining the overall behavioral and structural hierarchys of the circuit's global layout. Bottom-up design is usually restricted to the physical design process. Clearly, the overall design process is a dynamic and highly interactive process which requires the concurrent operation of several design techniques in order to realize a final product.

4.3 Logic Design Process

As mentioned in section 4.2, the logic design process consists of the mapping of a behavioral hierarchy based on the initial specifications to a structural level. Before proceeding with the logic design process then, we require an initial list of specifications governing the behavior of the overall FFT network and node. From Chapters 1 and 2, it was determined that the system level FFT implementation will be based on a radix-2 decimation in frequency algorithm. The FFT network chosen to realize this algorithm was the pipeline cascade implementation described in Chapter 2. In order to increase the dynamic range of the system it was also decided that calculations be performed in a floating-point format. It was shown in Chapter 3 that acceptable output signal-to-noise ratios are obtainable for a 1024-point FFT utilizing a 14-bit mantissa length. As a target performance objective, calculation of a 1024-point FFT in under 1 millisecond was sought.

On the node level, the behavioral specifications are essentially detailed expansions of the system level specifications. The node specifications are listed as follows:

1. The FFT node must be capable of performing operations as required by the pipeline cascade implementation, as outlined in Chapter 2.
2. It must be capable of bit-parallel communication with its nearest neighbors

3. It must be capable of working in a floating-point format.
4. The node requires the use of highly-concurrent architectures to reduce system delay time.
5. In order to perform a 1024-point FFT in under 1 ms, each node must be capable of performing a butterfly operation in within 980 ns.
6. In order to reduce processor communication lines, and controller complexity each structure should be designed to operate as independently as possible.

The proposed structural hierarchy and block diagram for the FFT node as derived from these specifications are seen in figures 4.2 and 4.3. The processor node specifications described above apply to the top level of this hierarchy. Likewise each sublevel structure also has its own associated behavioral specifications. These sublevel specifications are such that through them, realization of the higher level specifications is assured. Consider the example of the arithmetic processing unit (APU) at the second level of the hierarchy. At the lower levels of the hierarchy, the specifications governing operations become more and more performance and task-specific. Some of the specifications governing the design of the APU then are as follows:

1. The APU must handle a floating-point format of a 14-bit mantissa length (or more).
2. It must be able to perform a complex-number multiplication in less than 550 ns. and a complex-number addition in less than 200 ns.
3. It must be capable of performing desired bit shifts, mantissa justification, and two's-complementation as well as the operations of multiplication and addition .
4. It must operate with a high level of concurrency.
5. It must be capable of recursive data flow.
6. It should be microprogrammable.

Based on these specifications an APU was configured, utilizing a multiple access pipeline structure. A detailed discussion of this structure is found in section 4.3.4. The block layout for the APU derived from these specifications is seen in figure 4.4. This procedure of determining hierarchal structural designs and behavioral specifications continues to proceed in a downward direction until the simplest of the structural levels is reached. Realization of the final structural and behavioral hierarchies referred to above was accomplished only through a process of structural modification, and behavioral respecification. This process is ongoing in the attempt to further improve upon the design of the system discussed in this thesis.

4.4 The Physical Design Process

As was previously mentioned, the physical design process refers to the mapping of the structural hierarchy into the physical domain, where the actual realization of a structure in a given technology takes place. The physical design of this processor uses a bottom-up design methodology based on a four-level hierarchal design scheme. The designs of each level are comprised of the simpler designs making up the level below it. The four design levels are referred to as the microcellular level, the cellular level, the macrocellular level, and finally the processor levels of design. This section describe the constituent elements and configurations making up each of these levels, as well as discuss how these levels are integrated together to realize the final processing node. All designs have been made in a three micron complementary MOS or CMOS single-level metallization technology The design rules followed those outlined by the Seattle-Silicon Foundry. At present eight components have been submitted to the Canadian Microelectronics Corporation, CMC, for fabrication in a five micron CMOS technology.

4.5 The Microcellular Level

The microcellular level of design represents the simplest and the most primitive of the four design levels. It supplies the basic building blocks upon which all the other levels are based. The designs making up this level include a set of full adders, transfer gates, and data latches, as well as clocked and simple inverters. Also included is a logic cell library consisting of NAND, NOR, and XOR logic cell designs.

4.5.1 The Inverter

The simple inverter represents the most basic logic level design realized in this implementation, and as such it sees the greatest amount of utilization in the higher level cell designs. It is essentially a single input, single output gate comprised of two PMOS and NMOS transistors. Appendix A analyzes general MOS transistor operation. The inverter is responsible for inverting the logic level at its input. A high input signal, a "1", results in a low output signal or a "0". It is designed such that the output signal is determined after a single gate delay. In this thesis, the gate delay is defined as the time taken for the output waveform to rise/fall to 0.9/0.1 VDD after the input wave form has fallen/risen below/above 0.9/0.1 VDD. The gate delay is the basic time unit upon which component performance is based. A more detailed description of the gate delay may be found in Appendix B, which investigates the operation of the CMOS inverter in detail. SPICE simulations performed on the inverter show that the basic gate delay for this technology is approximately 1.2 ns.

Modification of the simple inverter is that of the clocked inverter. In this design two additional input lines representing clock and $\overline{\text{clock}}$ are required. In the clocked inverter the input signal is only evaluated when the clock line is high. Figures 4.5 (a-b) illustrate the simple and clocked inverters, while figure 4.5(c) illustrates the SPICE simulation of inverter operation under a single gate load. As a matter of convenience, the physical layouts are color coded to distinguish between the various layer representations. In this coloring scheme, red represents polysilicon, blue represents metal, green represents diffusion, and black represents a contact cut.

4.5.2 Logic Cells: NAND, NOR, XOR

The basic logic cells utilized in the higher level designs are the NAND, NOR, and the XOR logic functions. Their designs and associated logic tables are seen in figures 4.6 (a-c). Each cell is based on a two-input single output configuration. Propagation delays for the NAND and NOR cells is approximately 1.2 ns., while for the somewhat more complicated XOR function propagation delay is about 2.4 ns. Notice that expansion of these cells to more than two inputs is a simple matter, only requiring slight layout alteration. Also note that realization of the AND or OR logic functions simply requires the addition of a simple inverter to the output of the gate. This action however results in an additional gate delay. For this reason the negative logic functions are the preferred design elements.

4.5.3 The Transfer Gate

The transfer gate or pass gate provides the basic logic signal steering functions as designated by the controller unit. The transfer gate, can in simple terms, be thought of as an electrical switch. With the appropriate control logic applied to the gates of the NMOS and PMOS transistors, the source to drain path may be opened or closed. In this way data signals are routed. Figure 4.7 illustrates a simple transfer gate with its associated control and control_bar lines. SPICE circuit simulations have been performed on the CMOS transfer gate. In contrast to its parallel structure in NMOS which suffers greatly from time-delay problems and threshold voltage loss, the simulation results indicate that the CMOS implementation adds a delay time of less than 1.5 ns. Based on these results CMOS transfer gates are used extensively to implement the processor control logic.

4.5.4 The Data Flip-Flop

The CMOS data flip-flop represents a single bit memory location and is the simplest form of memory cell utilized in this implementation. The cell is realized by three interconnected inverters, two of which are clocked. The design and schematic of the data latch is seen in figure 4.8 (a). Data flow into the latch is controlled by a clock and $\overline{\text{clock}}$ set of control lines located on the input and feedback inverters. A high clock signal applied to the input inverter initializes the flip-flop for writing, and the output of the latch simply follows that of the input signal line. On the low-going clock pulse the first inverter is disabled, and thus the latch is isolated from the input signal. At the same time however the feedback inverter is engaged, and the last logic level is cycled through the two looped inverters. At this point the output remains constant and independent of the level of the input. Notice also that the stored bit is output on a continuous basis. In practice this is not a desirable feature, and thus it is often necessary to disable the output line by means of a transfer gate until the particular flip-flop in question is addressed. A timing diagram for flip-flop operation is found in figure 4.8(b).

The latch configuration described here is extremely simple and its design may be realized in a small area. As seen from figure 4.8 (a), the design of the latch is a highly regular configuration which may be readily implemented in an array type structure. Dimensions of the latch are approximately 100 microns ×160 microns. Unfortunately to achieve this simplicity sacrifices had to be made in terms of latch sophistication. This configuration does not offer immunity against the possibility of clock skew that the more sophisticated edge triggered latches do. For this reason the timing sequence for the data latch is critical. Sufficient time must be made available for the signal D to settle and achieve a steady state before lowering the clock lines. This is particularly true for those cases where D is the output from an adder or multiplier and is subject to several changes of state before its final logic level is determined. In this implementation however the clock pulses are governed by a worst-case operation time plus a small signal settling period of 5 ns., which according to SPICE simulations is sufficient to ensure line settling. This eliminates the problem of undetermined states at the lowering of the clock lines.

4.5.5 The Full Adder

The full adder cell and its various modifications represent the highest level of design sophistication on the microcellular level. The full adder represents the bit-level building block upon which the parallel multiplier, parallel adders, two's-complementers, and biasers are built. It is responsible for performing the arithmetic addition operation on two binary bits, yielding a sum and a carry-out bits. Thus each full adder is comprised of three input lines, the carry-in, input A, and input B, as well as two output lines, sum and carry-out. The truth table and logic circuitry describing the full adder is seen in figure 4.9 (a-b).

The full adder design is realized here by means of a two-level gate implementation. In this implementation the use of transfer gates, and the time-costly XOR gates is avoided. The sum and carry-out terms are generated simultaneously after only a two-gate delay. The two gate time-delay on the sum out term proves to be one or two gate delays faster than that of the more conventional transfer gate implementations. This fact is extremely important if we consider that the performance time of the parallel multiplier, to be described later, is directly proportional to the sum and C_{out} propagation time.

The increase in the sum and C_{out} propagation speed comes at the cost of increased design area. Even though the full adder design seen in figure 4.10(a) is optimally compacted by means of extensive hand layout, it still represents approximately a one-third increase in area over the more conventional transfer gate implementations. This cost is considered acceptable in view of the 25-50% decrease in processing time obtained.

The dimensions of the full adder cell are 183 microns by 100 microns, and its regular structure makes it ideal for implementation in the common array type architecture associated with parallel multipliers. The full adder cell also represents the largest cellular design on which SPICE analysis was run. This cell contains some 30 transistors which appears to be about the maximum number which this version of SPICE can handle efficiently. From the graphical output of the SPICE analysis in figure 4.10(b), we can see that both the carry-out and sum out signals are both ready after approximately a 2.5 ns delay.

4.6 Cellular Level Design

Cellular level design offers an increased level of design sophistication over that of the previous level. Typical design sizes on this level run from approximately 200 to 8000 transistors. These designs include such components as parallel adders, latch-banks, justifiers, shifter units, and the parallel multiplier. Each of these designs is built up in some sense from the more basic designs described in the microcellular level.

4.6.1 Parallel Adder

Parallel adders and the modified parallel adders are used extensively in the mantissa and exponential pipeline of the arithmetic processors. As is described later, these elements serve primarily as adders, biasers, counters, and two's-complementers. In the pipelines of the arithmetic processor the size of these elements varies from 8 bits when dealing with the exponential components of the floating point format to 15 bits when working with a two's-complemented mantissa.

In general, an N-bit-parallel adder operates by implementing a network of N full adder cells in a bit-parallel fashion, with the carry-out of each full adder cell serving as the carry-in of the more significant digit.

The carry-in of the least significant term is conventionally pinned to ground. This implementation is realized simply by creating a one dimensional array of full adder cells. These cells have been designed such that voltage and ground lines will become common to all cell during the arraying process and proper alignment of the carry-in and carry-out lines is ensured. The net area of the parallel adder unit is simply given by:

$$\text{Area} = \text{N}\times(\text{full adder area}) \tag{4.1}$$

where N refers to the bit size of the adder.

The worst case performance time is governed by the time taken to propagate a high carry-out bit from the least significant adder cell through to the most significant one. Thus the theoretical worst case performance time is given by:

$$\text{T}_{\text{Adder}} = \text{NT}_{\text{C}_{\text{out}}} \tag{4.2}$$

An Hg logic simulation for the 14-bit adder is seen in figure 4.11, indicating a worst case performance time of approximately 30 ns.

Realization of modified adder circuits is usually accomplished through direct hardwiring of the circuits. For example in the case of the two's-complementer, inverters are placed on one of the adder cells inputs, while the second inputs are held to ground except the least significant bit which is held high. Recall:

$$2'{}^{s}{}_{comp.}(A_i) = \overline{A_i} + 1 \tag{4.3}$$

In the case of counters which happen to be used primarily in the control logic implementation the sum terms are passed directly into a bank of flip-flops. These flip-flops in turn feed back into one set of the adders input elements. The second set of input elements are again held to ground, with the exception of the least significant bit which in this case is tied to the counter's clock. Figure 4.12 (a) illustrates the parallel adder block layout.

4.6.2 Latch Banks

As in the case of parallel adders, the latch banks are realized simply by arraying together the single bit latches of the microcellular level. They are generally configured such that each latch is capable of holding a 23-bit word, which represents the truncated 32-bit data element. The latch-banks are responsible for storing the input and output data elements for each processing node. They are also used within the arithmetic processor unit where they hold the initial, final and intermediate results. In this case their size is reduced, and varies from 8-16-bit words as required. Read/Write enable commands to each of the latch banks are controlled by the control units via a simple decoder element. This will be elaborated upon in detail in Chapter 5.

4.6.3 The Justifier Unit

The justifier unit design occurs exclusively within the arithmetic processor, and is required only in the use of floating-point operations. It is responsible for determining the size and direction of bit shift required to left justify the number's most significant bit. This information must then be used to increment or decrement the exponent term of the number accordingly.

Recall from Chapter 3 that the IEEE standard to be utilized in this implementation operates with an exponent term biased by 127. Upon entering the arithmetic processor, this bias is removed and is subsequently replaced after processing of the exponent and prior to output. The justifier unit incorporates this rebiasing term into its logical output. Thus the rebiasing phase and the appropriate exponential incrementation due to mantissa shift may be realized in a single parallel addition cycle.

The justifier design is seen in figure 4.13 (a). It operates using a nested NAND configuration in conjunction with a transfer gate switching network. The most significant bit location is determined from one of 15 possible locations via the NAND plane, and a single high output is produced on one of 15 output lines. This single high output triggers the transfer gate switching network to output an associated hardwired response. In a worst case scenario, that of a 14-bit shift, the proper output is determined after 15 gate delays, or approximately 22 ns. A complete justifier operation therefore takes place within the time period associated with a 14-bit-parallel addition.

4.6.4 The Shifter Unit

The shifter network is a routing system achieved by means of transfer gates. The implementation described here is capable of achieving a 0-14-bit shift with only a 4 ns. delay time. Again this unit is employed strictly within the arithmetic processing unit and is primarily concerned with performing shifts on the mantissa element to obtain proper decimal point alignment or justification. The design of the 0-14-bit shifter unit and basic shift element is seen in figures 4.14 (a,b).

The amount of shift performed is controlled from a 4-bit binary coded number. The shifter itself is subdivided into 4 routing stages, one associated with each of the control bits. The first routing stage is controlled by the most significant control bit and represents either a 0 or a 2^3-bit shift. The output of this stage is in turn passed in parallel to the second stage where either a 0 or 2^2-bit shift is performed. This progresses to the fourth and final shifter stage where a 0 or 1-bit shift is made. Thus by applying the appropriate binary values to the control line any shift from 0 to 14 bits is possible.

4.6.5 The Parallel Multiplier

The parallel multiplier is by far the largest and most sophisticated of the designs on the cellular level. It utilizes some 7500 transistors, and its dimensions are approximately 3200 microns by 1800 microns. It is capable of multiplying two 14-bit numbers to produce a 28-bit product in binary multiplication in approximately 70 ns. While this is quite respectable for a multiplication of this size, it is still about twice the worst case add time for the 14-bit-parallel adder. As in most arithmetic unit implementations, the multiplier unit proves to be the bottleneck to system performance. In order to reduce the net processing time of the FFT, the time spent for each multiplication must be minimized. This is achieved by utilizing the fullest degree of parallelism possible in the multiplier.

The parallel multiplier implementation described in this design is a modification of the conventional "carry save" adder array [17]. This implementation utilizes a negative logic with a repetitive modular layout of N by N-1 adder cells, where N is the bit length of the numbers to be multiplied. Figure 4.15 (a) illustrates the basic cellular level layout for the parallel multiplier. In this figure the squares represent the full adder cells, while crosses represent single NAND operators which are responsible for performing the bit-level multiplication.

In the multiplication of two 14-bit operands, X and Y, bits $X_0 - X_{13}$ are applied to the multipliers input rows 0-13, while the bits Y_0–Y_{13} are applied to input columns 0-13. The adders used in this modified adder array implementation are the same as that described previously. Recall that this adder requires three input bits, A, B, and C_{in} and produces two output signals, C_{out} and Sum. Figures 4.15(b-c) illustrates the signal propagation path for the C_{out} and Sum signals for each of the adder cells in the array. From this figure it is seen that the C_{out} propagates in a downward direction to the bottom of the multiplier where it propagates from the least significant product bit to the most significant one. Each Sum signal propagates in a direction 45 degrees to that of the carry-out signal. In this way each sum signal runs along a line of fixed binary weights.

The input A to each adder cell is taken as the product of the two operand bits such that $A = X_i Y_j$. Input B represents the Sum signal from some previous adder cell somewhere along the line of fixed binary weight, while the C_{out} signal is propagated down from the C_{out} line of the adder cell directly above it. The main difference between this and the more conventional implementations lies in the nature of the transfer of the Sum signals. In the conventional adder array, each intermediate Sum signal, is propagated to the next cell immediately along the line of fixed binary weights. In this implementation, the net sum propagation is sped up by transferring two intermediate sums to two new cells along the diagonal simultaneously. That is the Sum from each cell is propagated such that it skips the subsequent cell along the diagonal. This is equivalent to having two rows of adder cells summing the intermediate product terms in parallel, thus speeding up the calculation.

Notice from figure 4.15 (b), that in this implementation the array of N by N-1 adder cells is shifted two bits to the right of the least significant X bit and and two bits below the most significant Y bit. Thus we have one row and the two bottom lines of adders lying off the

N by N operand bit grid. This frees up the input signal line A of each of these adders. As the two intermediate sum values propagate simultaneously towards the array perimeters they serve as the new inputs A and B to the outermost row and the next to the bottom line of the adder cells. The bottom line of cells is responsible for summing the Sum and C_{out} signals of the previous line to the propagating C_{out} of the outermost row. This summing procedure is made clear by referring back to figures 4.15(b-d).

Because of the speeding up of the intermediate sums, the worst case multiply time is now dictated by the time taken for the carry-out signal to propagate down and across the perimeter of the array. This is given by:

$$T_{\text{worst case}}=(2N-1)T_c \tag{4.4}$$

where T_c is the carry-out time-delay from the adder. As seen in the SPICE simulation for the full adder cell in figure 4.10(b), T_c was approximately 2.6 ns. Therefore the predicted worst case multiply time for a 14-bit by 14-bit multiplication is approximately 70 ns. While a 70 ns multiply time is quite respectable when compared to the performance of other commercially available products, there is still room for improvement. These improvements are investigated in Chapter 7.

4.7 The Macrocellular Design Level

Macrocellular designs integrate designs of the cellular and microcellular levels together to realize a specific function or process. In general macrocellular designs offer high levels of processing sophistication and often require the use of specialized control units to co-ordinate data flow and process operations. In the case of the general-purpose node only a single macrocellular design was utilized, that being the general-purpose arithmetic unit.

4.7.1 The Arithmetic Processor Unit

The APU design is based on two concurrently implemented multiple access pipeline (MAP) structures, one each for the mantissa and exponential components of the floating-point formats. The multiple access pipeline structure is a combination of bus and pipeline architectures. It combines the processing efficiency of pipelined computations with the flexibility of a bus architecture to create a very flexible, highly concurrent processing system. In order to reduce communication requirements and processor crosstalk each structure was designed to act in an independent fashion. Communication between the pipelines is restricted to only two sets of data lines referred to here as the exponential difference term and the exponential rebiasing term. A block diagram of the APU and MAP architecture is seen in figure 4.4.

Data flow within the MAP structure occurs as either pipelined or bus data flow. In pipelined data flow, the data elements propagate from one processing stage of the pipeline to the next. The data flow is unidirectional and is restricted to a flow rate of one processing stage per clock cycle. Due to the unidirectional nature of the pipelined data flow, recursive iteration of previous processing stages is not possible. While this structure does restrict data flow in terms of direction and transmission rate, it offers the advantage of high levels of concurrent operation. Because the data elements are local to every stage, and the transmission rate is constant along the pipe, multiple data elements may exist concurrently at different positions along the pipe. In this way each processing stage may act on a different data element before passing it along on the next clock cycle. Thus we have the capability of concurrent multiple processing operations. The longer the pipe, the higher the potential for concurrent operation.

In a bus type architecture, data are very quickly broadcast to all processing elements. Through proper routing, data may be transferred from any one processing stage to another within a single clock cycle thus making this architecture extremely fast and flexible. The disadvantage to bus data flow is that only a single data element may access the bus at any one time, thus greatly reducing the levels of potential processing concurrency.

The MAP structure implements both the pipeline and bus architectures simultaneously with a parallel data bus being implemented alongside the pipeline. The data bus is connected to the pipeline by means of multiple "ports" at the input and output of each processing stage. Each port consists of a parallel set of tristated data lines connecting the pipe and bus. Upon reception of an appropriate control signal the gates are opened, and a parallel bidirectional communication link between the pipe and data bus is established. It is important to notice that in order to avoid the possible conflict of having multiple data elements on the bus at the same time, a restriction needs to be made that no more than one output port may be open to the bus during any one clock cycle. This restriction is built into the control structure governing the data bus access.

With the introduction of the data bus we have greatly increased the flexibility of the system. We now have the option of accessing any one or any sequence of pipeline processing stages we desire. Another important fact is that by utilizing the access ports we are only interfering with the pipeline operation at the stage immediately behind the port. As the rest of the processing stages remain isolated from these ports they are free to maintain normal operation. In this way high levels of processing concurrency are maintained.

Another advantage offered by this structure is that of recursive processing operations. Utilizing the bidirectional feature of the data bus operations such as convolution or simple recursive filtering operations are easily implemented.

The MAP implementation requires complex control sequencing. This will be thoroughly discussed in Chapter 5. Essentially however great care must be taken in the design of the control so as to avoid the back-logging or in some cases the loss of data.

4.7.3 Components and Configuration

The APU structure was conceived for floating-point complex-number multiplication and addition. Recall that one of the original behavioral specifications called for the design of a floating-point processor to perform the butterfly operation as required by the FFT node. The arithmetic processor was then configured so that the functions of complex-number multiplication and addition might be realized as efficiently as possible. In order to understand the reasons for the structural layout one needs to fully understand the steps needed to realize a complex floating-point number multiplication.

In general the multiplication of two complex-numbers, $(a + ib)$ and $(c + id)$ may be given as:

$$(a + ib)\times(c + id) = ac - bd + i(zc + ad) \tag{4.5}$$

Determination of the real and imaginary components of the product, therefore requires a total of 4 real multiplications and two real additions. In dealing with a floating-point format, these operations are complicated by the presence of the exponential elements. In the case of multiplication, the sum of the exponential elements must be determined, while for floating-point addition it is necessary to calculate their difference. Once calculated, this difference term is used to shift one of the mantissa elements an appropriate amount to ensure proper alignment of the binary point during the floating-point addition and subtraction. As discussed in Chapter 3, the data format chosen for this implementation is a modified IEEE format. The mantissa pipeline is designed to handle a mantissa element which has been truncated from the standard 23-bit length down to 14 bits. The exponential pipeline operates on an 8-bit exponential element which has been positively biased by a factor of 127.

The APU was designed such that the clock rate for the exponential pipeline is approximately twice that for the mantissa pipeline. The clock pulse width of the exponential pipeline is based on the worst case addition time for an 8-bit-parallel addition, which is taken as $T_{8bit} \approx 20$ ns. The mantissa pipeline operates on a clock pulse width of 40 ns. This is slightly greater than the worst case addition time for a 16-bit parallel addition, and approximately half the worst case multiply time for a 14-bit by 14-bit multiplication. Such a clock width ensures that all mantissa processing operations with the exception of the parallel multiplier, are completed within a single clock pulse. Parallel multiplication in this implementation is a two mantissa clock cycle operation.

The advantages of utilizing two independent clocks for the pipeline operations are easily seen if we consider the example of floating-point addition. As previously mentioned this operation requires that an exponential difference term need be calculated before the shifting and addition operations may be carried out on the mantissa. To streamline system performance then this difference term should be calculated as fast as possible. By implementing different clocks for the mantissa and exponent pipelines, this is made available nearly twice as fast as would otherwise be possible. In terms of time savings, this implementation reduces the floating-point addition operation by more than 100 ns.

Since in this implementation, the APU is concerned only with performing the butterfly operation the component integration of the systems pipelines will be discussed as viewed from these operations. For convenience in the following discussion, mantissa elements are identified by the use of a subscript M, while exponent elements will carry a subscript E. Unsubscripted elements refer to the combined mantissa and exponential elements.

The APU operation of complex-number multiplication begins with the data elements a, b, c, and d being read into the processor in a bit-parallel, word-serial fashion. Immediately upon entering, each data element is split up into its mantissa and exponent components which are passed into data registers located at the head of their respective pipes. For the exponential elements, this operation is accompanied by the addition of a debiasing factor of -127. Processing of the complex product begins immediately following the reception of element c into its appropriate register. Beginning with the mantissa pipeline, the first pair of elements, a_M, and c_M are placed into the multiplier at the same time as element d is being read into the processor. After a period of two clock cycles, the intermediate product ac_M has been determined and stored in a latch. At this point two new elements b_M and d_M are loaded into the multiplier, and the product term ac_M is passed down the pipe where it undergoes a possible shift or two's-complementation, as directed by the control unit. During this time the exponential pipeline has passed the appropriate elements into the first adder unit, and calculated the intermediate exponential sums $(a_E + c_E)$ and $(b_E + d_E)$ for the terms ac_M and bd_M respectively. These sums are stored in an intermediate latch-bank seen in figure 4.4. One of the sum terms is then negated through a two's-complementor unit, and the exponential difference term between $(a_E + c_E)$ - $(b_E + d_E)$ is determined by the subsequent addition of the complemented sum and the other non-complemented sum. The difference term is then passed to the mantissa pipeline where it directs the shifter to shift the smaller product either ac_M or bd_M a predetermined amount. All this is done during the two mantissa clock cycles taken to determine the first product ac_M.

At this point the appropriately-shifted and/or two's-complemented ac_M term is stored in another latch, and waits for the bd_M term to "catch up". Both terms ac_M and bd_M are loaded into a parallel adder unit which forms the real part of the complex product. The next mantissa processing stage to be encountered is the justifier unit, which determines the appropriate shift required to rejustify the mantissa element to that the most significant bit is a leading "1". The justifier also incorporated into this operation is a hardwired binary coded shift which has been biased by a factor of +127. This term is sent to the rebiaser unit of the exponential pipeline which readjusts the greater term $(a_E + c_E)$ or $(z_E + d_E)$ for the mantissa shift term and the rebiasing factor in the same operation. The mantissa element is simultaneously passed back into the shifter unit by means of the data bus, and is ready for output. It has been predetermined that no conflict in shifter usage will occur using this timing sequence. The imaginary component of the complex product undergoes essentially identical operations, and follows the real term by a single clock cycle on output. The net processing time from data input to data output is 520 ns. or 13 mantissa clock cycles.

The operations of floating-point complex addition and subtraction are identical to that of complex multiplication, except that the multiplication step is bypassed by means of the data bus. The appropriate mantissa elements are passed directly into the shifter and two's complementer once the exponential difference term is complete. The net processing time for determining both the real and imaginary sums of a complex addition is 240 ns. Notice that the final shifting operation occurs simultaneously with the data output. As this in no way impedes the net processing time, it is not included as a processing cycle in the above quoted performance times.

A useful tool to analyze pipeline performance is the operational concurrency graph. Essentially this is a bar graph capable of displaying levels of pipeline concurrency. Examples of operational timing graphs for complex-number multiplication and addition are seen in figures 4.16 and 4.17. Concurrency is readily indicated by the number of operators functioning at any one period of time. In the case of the complex-number multiplication we can see that any two or three of the five operators are usually functioning at any one time. Without this level of processing concurrency offered by the pipeline, each of these operations would have to occur in a sequential order. In this event the complex multiplication would take some 1040 ns. to calculate, over a factor of 2 greater than that achieved by the pipeline system.

To make full use of the concurrency potential offered by this pipeline, we now look at the pipelining of two complex-number multiplications. The operational timing graph for the mantissa in this case is seen in figure 4.18. As seen from this graph we may begin the second complex multiplication some 160 ns. or four clock cycles before the completion of the first multiplication with no conflict over pipeline component usage. The pipeline sequencing for the second multiplication then remains unchanged from that of the first multiplication. Through this approach we may perform two pipelined multiplications in 880 ns., yielding an average complex multiplication time of 440 ns. per multiply, a 15% improvement over the single multiplication case. This pipelining would be directly applicable to the case of on-chip twiddle factor updating, where the factor is updated in one multiplication and multiplied by the butterfly midterm in the second. This same principle when applied to complex addition reduces the average processing time from 240 ns. to 190 ns.

Through the use of the pipelined processor, extremely fast floating-point complex-number calculations are possible. The performance results indicate that the original behavioral specifications have been met, and as will be shown in Chapter 6, these performance times are sufficient to achieve a sub-millisecond calculation of a 1024-point FFT. These results offer significant improvements over currently available processing techniques.

4.8 Node Level Design

The node level design is the highest level of design realized in the hierarchal physical design process. Its purpose is to integrate together components of the macrocellular and cellular levels to realize the final FFT node configuration. In this sense design on the node level is primarily oriented towards control and routing of the data elements as opposed to their actual calculation. Because the substructures have been designed for independent or isolated operation, the node level designer need not concern himself with the lower-level sub-operations. Such independent operation on the macrocellular level greatly facilitates the problem of design on the node level. The block diagram for the FFT processing node was derived earlier in accordance with the structural and behavioral design requirements. It is found in figure 4.3.

The key component aside from the APU on the node design level is the controller unit. This unit is responsible for knowing where in the network the node is located, and thus what operations are required from it. It is comprised of a simple clock-driven counter, a two-level logic block and a a few associated ROM. Its operation is discussed in detail in Chapter 5. In brief the node located at some position M (recall from Chapter 2 that M runs from 0 to $\log_2 N$) is responsible for overseeing the following functions:

1. The node reads data from some previous source, and pipes the data directly through to associated off-chip ROM. This continues until a total of 2^{M-1} elements have been read.
2. Next, the node combines the input element with the first-read data element and performs the initial part of the butterfly operation. The sum term is passed to the next node, while the difference term is passed back to the off-chip RAM or FIFO. This again continues for a total of 2^{M-1} operations.
3. The first difference term is then read from the RAM and multiplied by the appropriate twiddle factor, ω, and passed out of the cell. The twiddle factor is updated and the next number is multiplied. This continues until all the values in the ROM are read. Concurrent to these multiplications, step 1 may be repeated thus res-

tarting the sequence.

Referring to figure 4.3 we see that the data element enters the node in a bit-parallel fashion passing directly into a set of input RAM. At this point depending on the control signal received, the element is passed either into the APU for processing or is piped directly through to the 3-way multiplexed I/O port. These ports are responsible for the transmission of data on to the next node, as well as for effecting the read/write operations to the off-chip memory. Thus two data elements may be read into the node simultaneously through the input ports at the front end of the node as well as from the off-chip memory through the I/O ports. Input elements from the I/O ports are also temporarily stored in the input RAM block prior to processing in the APU. After computation the results are output to a set of output RAM where they are held until they are to be transmitted off-chip. Also included on this design level is the clock unit. This unit operates in conjunction with the controller unit and is tied to a common clock serving all the processor nodes.

The node requires some 46 pins to accommodate the 23-bit floating point format, and 8 control pins as well as two pins for VDD and GND, making for a total of 56 pin outs. The node occupies some 6400 microns by 6400 microns in area. Discussion of the nodes performance and controller operations is deferred to Chapters 5 and 6.

4.9 Summary

This chapter has reviewed the basic VLSI design methodology used to design the general-purpose processor node. Two design processes, the logic design, and the physical design were carried out concurrently. In the logic level design a top-down approach was employed, in which the original and revised design specifications were determined. In the physical cell design a bottom-up approach was described in which each of the higher-level cells were made up from the simpler lower-level designs. The function and operation of each component was discussed. In most cases the actual physical layouts and circuit simulation were also given.

In section 4.7 of this chapter, the MAP structure was presented. This structure combines the concepts of pipelined and bus architectures to create a highly flexible processing element. The processing node's APU utilizes two such structures. The arithmetic processor unit, or APU is the most important element described in this implementation. In reference to FFT networks the APU is solely responsible for performing the butterfly operations. Functionally, however it may be thought of as a fast general purpose arithmetic processor for floating-point numbers. As a coprocessor it is capable of operation not only in any of the FFT networks described in Chapter 2 but also in any other endeavor, signal processing or otherwise, requiring fast floating-point mathematical computations.

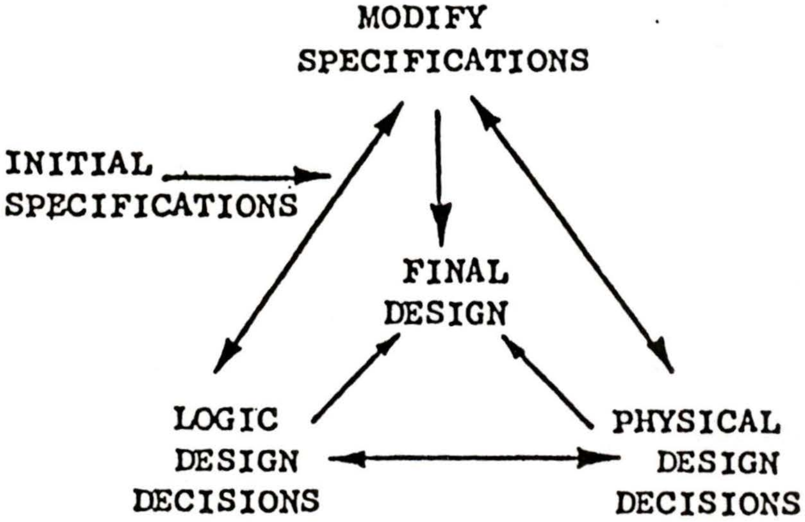


Figure 4.1

The cyclic pattern of design decisions and modifications associated with logic level design.

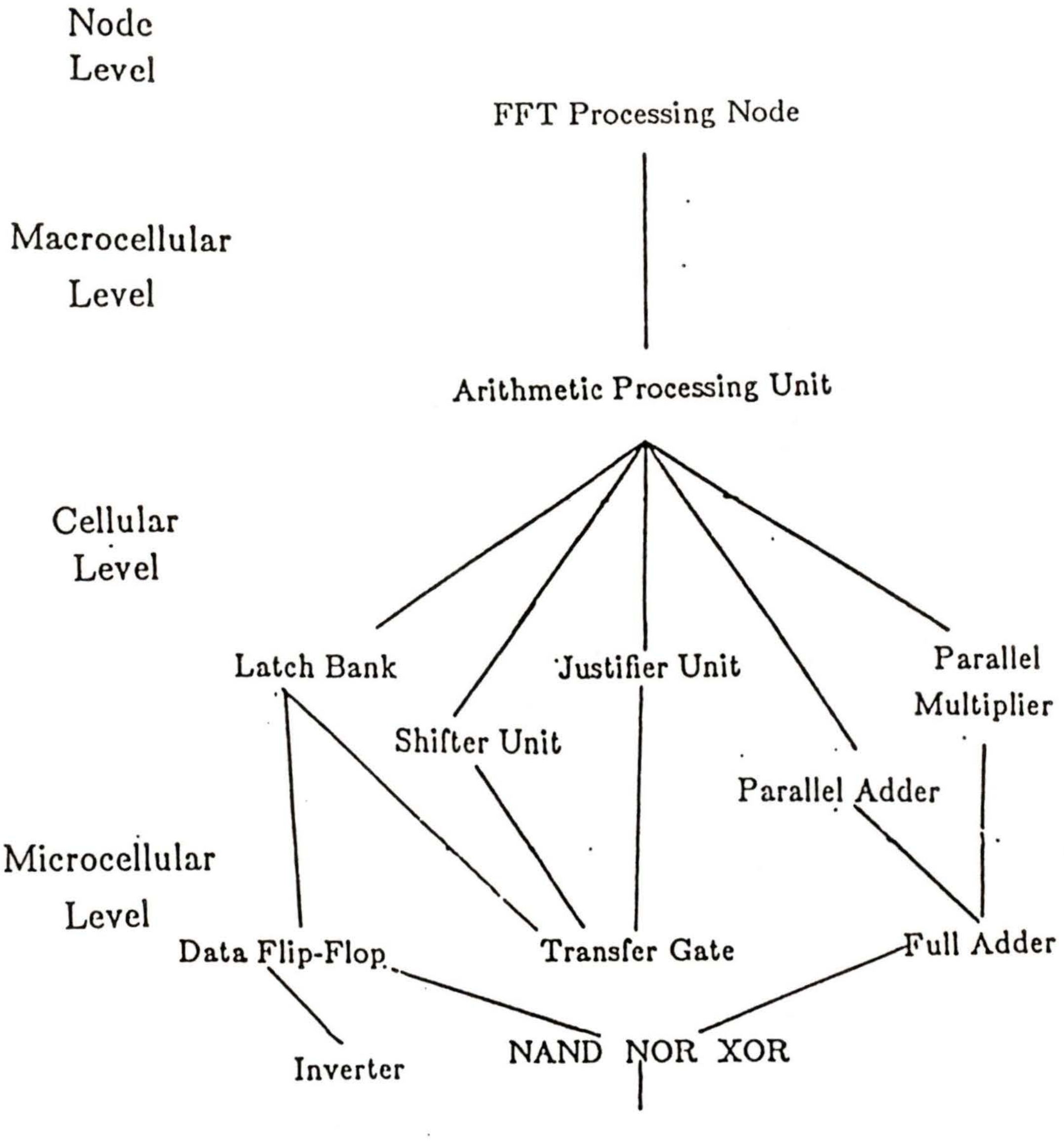


Figure 4.2

A proposed hierarchal structural layout for the FFT processing node. Notice that the structural layout is broken down into four distinct levels of design, with each of the higher level components being built up from the lower level elements.

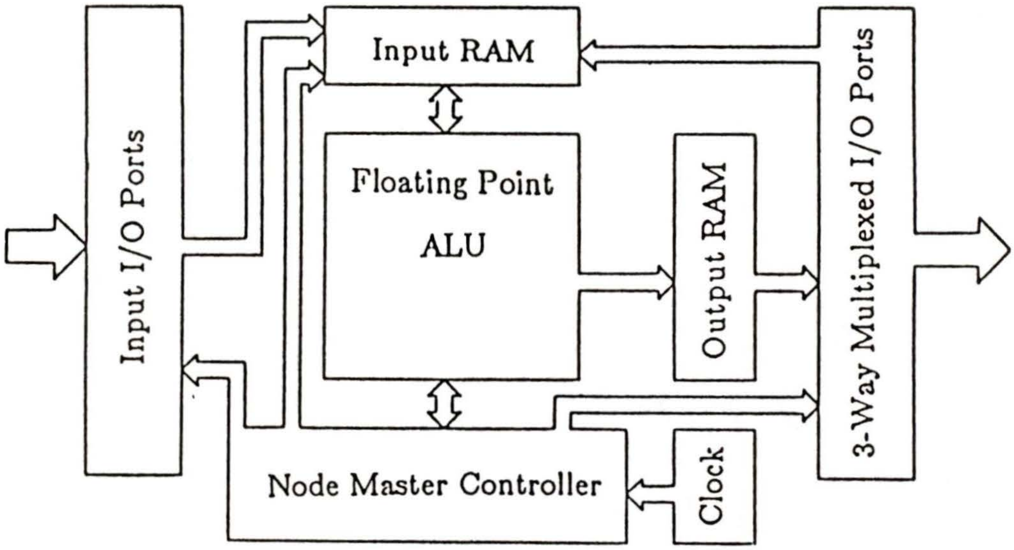


Figure 4.3

A proposed block diagram for the FFT processing node. Viewed with respect to figure 4.3 we see that the node is comprised of a single macro-cellular design, that being the APU. The other constituent components include a node level control unit, temporary data registers, and multiplexed I/O ports. A detailed discussion of each of these elements is found in Chapters 4 and 5.

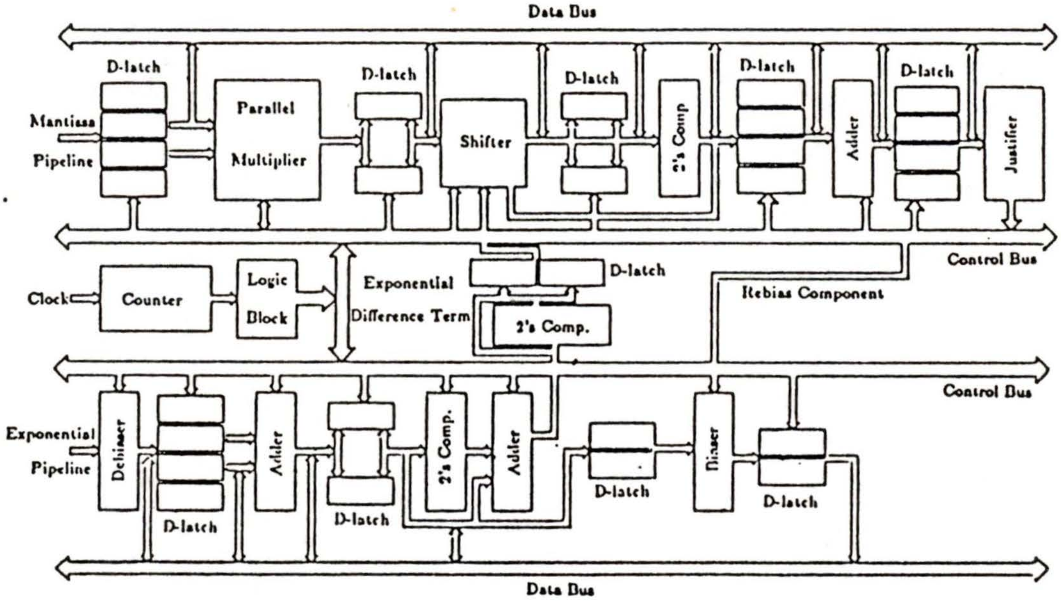
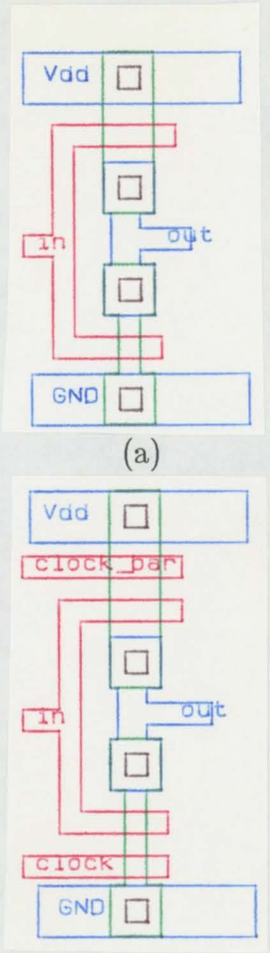


Figure 4.4

A block layout for the APU unit. The APU unit is comprised of two concurrently operating multiple access pipeline or MAP structures, one each for the mantissa and exponential elements. Data proceeds through the pipelines in a bit parallel word serial manner. Each of the pipelines processing units is separated by a set of intermediate data latches. Pipeline processing is controlled through the appropriate application of read/write commands to these latches. A parallel data bus runs along side each of the pipelines. Access to the bus is made through a set of "ports" located between each processing unit and data latch.

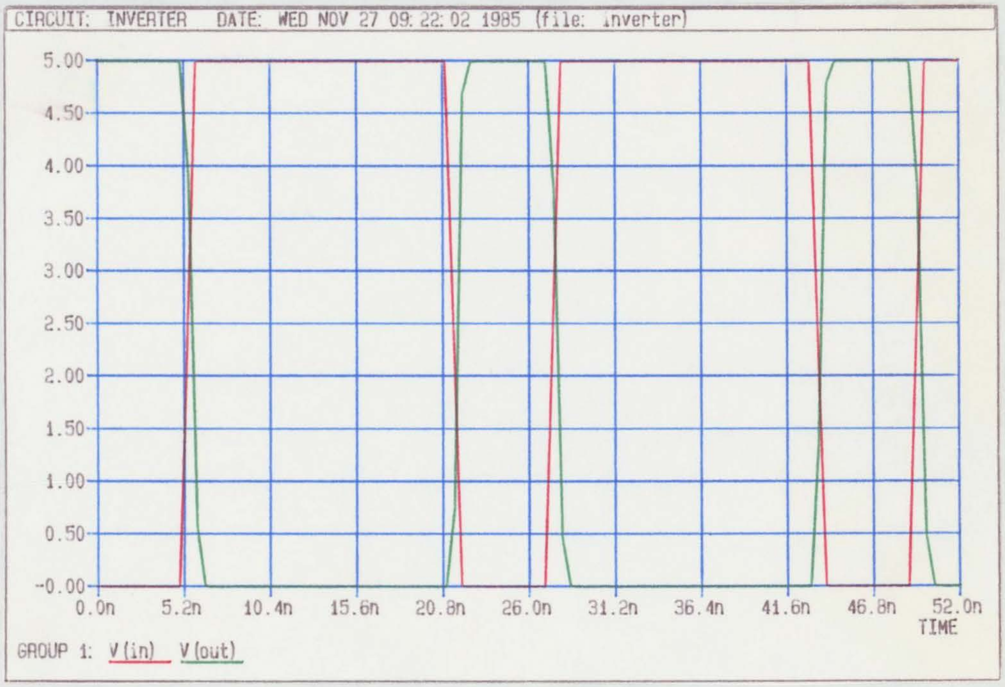


(a)

(b)

Figure 4.5

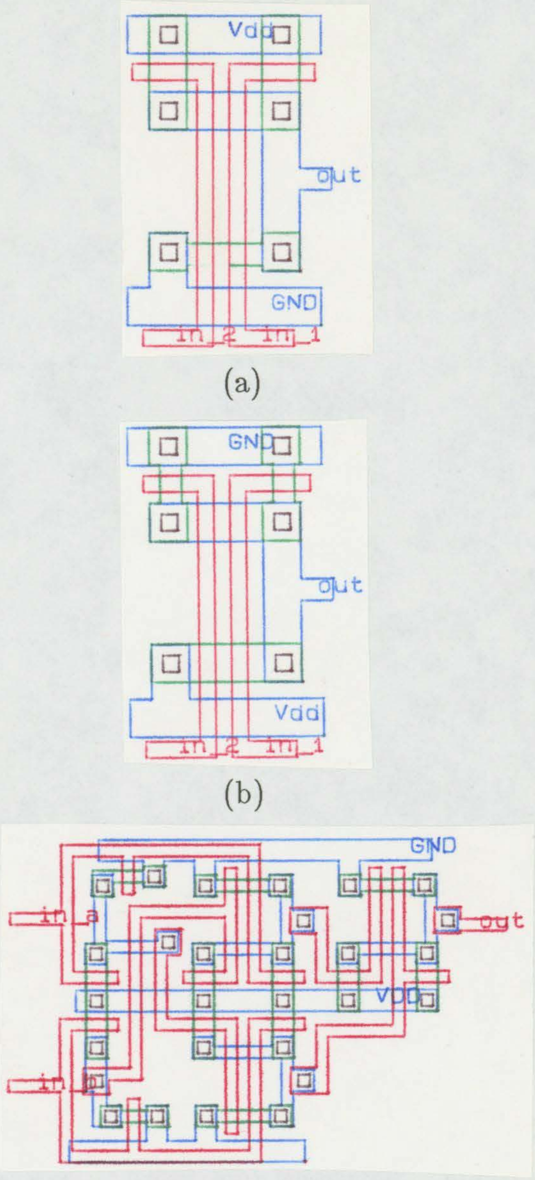
Figures 4.5 (a,b) illustrate the simple and clocked inverters respectively. In this and all other color figures, a common coloring scheme is used to represent the various layers of materials. Red represents polysilicon. This layer serves as the transistor gate material, and as a conductive medium for signal propagation. Blue represent metal, which is used to carry signals, and to supply the voltage and ground lines. The green lines define the regions of implant or diffusion. These regions represent the transistors source and drain. A transistor is formed at each crossing of the red polysilicon and the green diffusion. The PMOS and NMOS transistors may be distinguished through their size. The PMOS transistors are made up of the wider diffusion layers.



(c)

Figure 4.5 cont'd

A SPICE simulation for the simple inverter. In this simulation a single gate load was placed on the simple inverter. The gate delay calculated from this simulation was found to be 1.2 ns. It is defined as the time taken for the inverter output to rise/fall to 0.9/0.1 VDD after the input gate voltage has fallen/risen from 0.9/0.1 VDD.



(a)

(b)

(c)

Figure 4.6

Figures (a-c) illustrates the physical layouts of the two-input CMOS NAND, NOR, and XOR gates respectively.

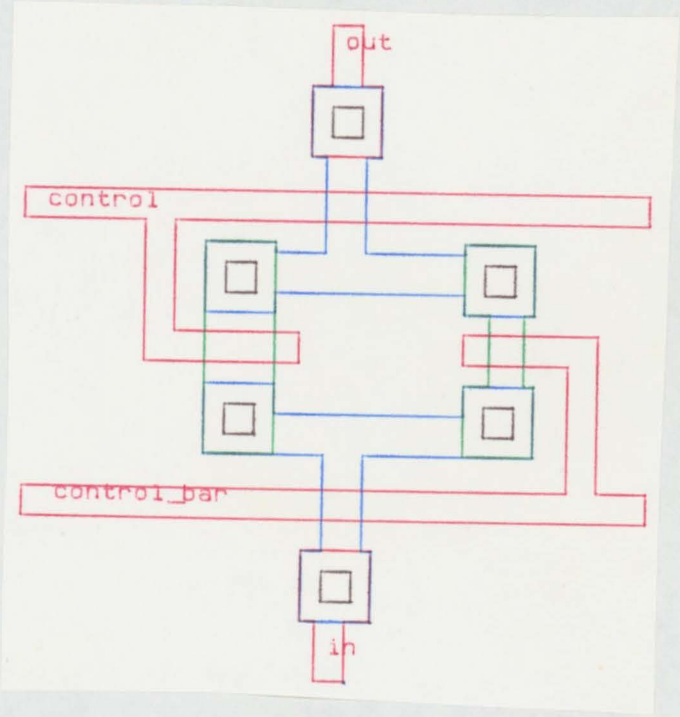
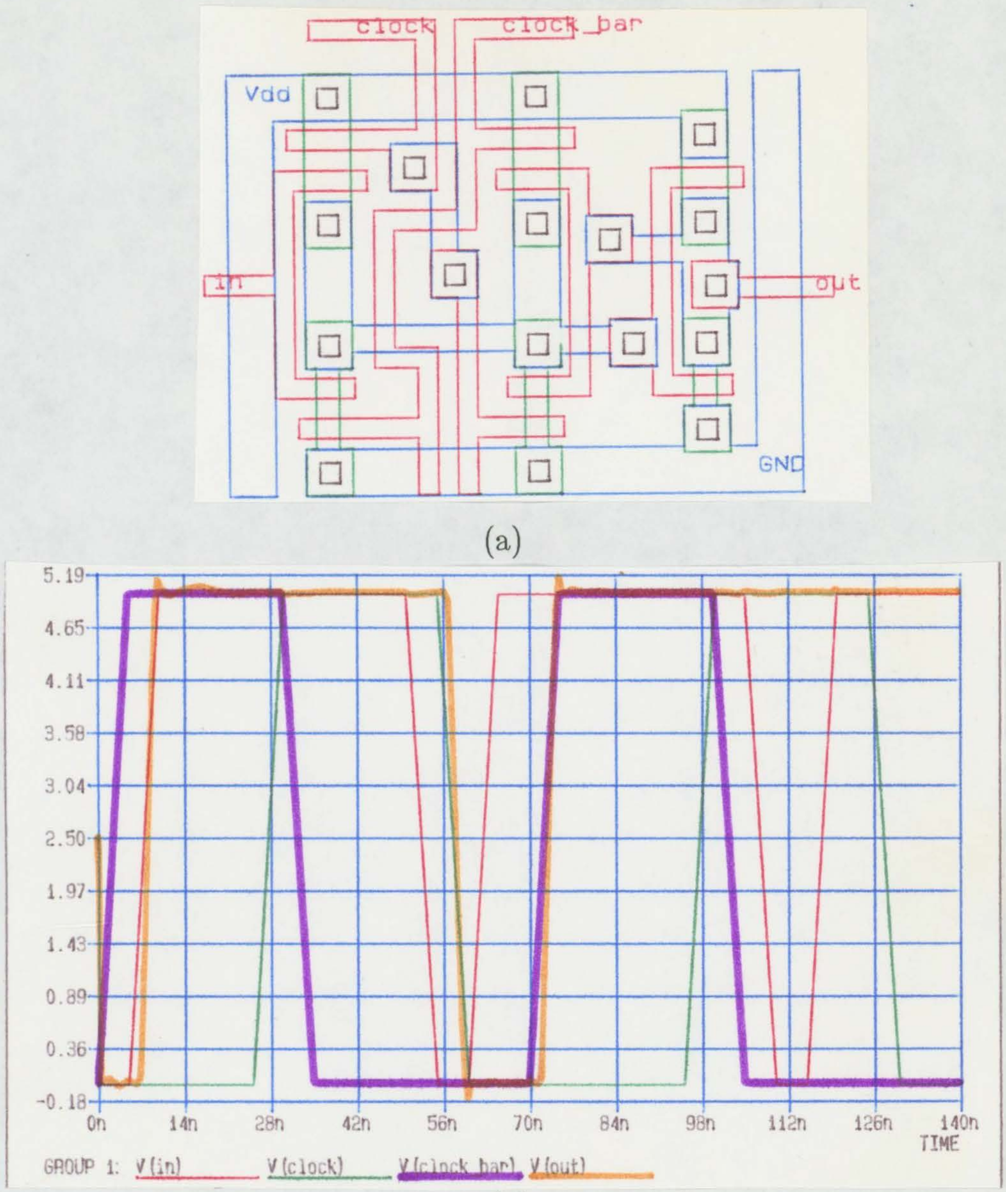


Figure 4.7

The CMOS transfer gate. Notice the two control lines C and C_bar which are used to turn on and off the NMOS and PMOS transistors.



(a)

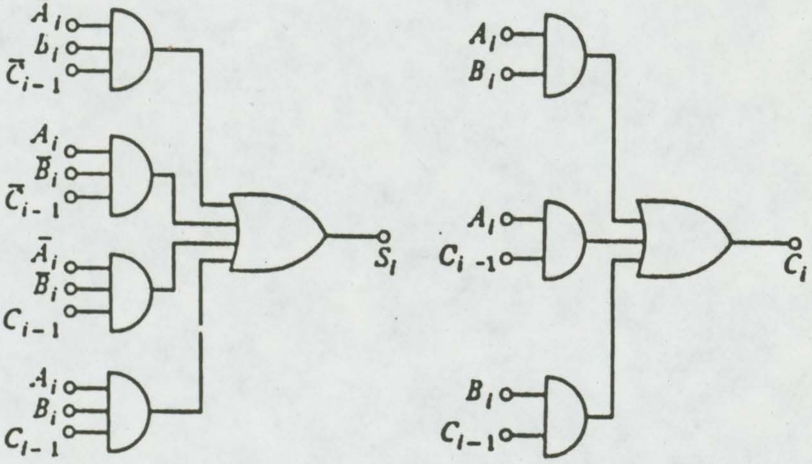
(b)

Figure 4.8

The data latch, and its Hg timing simulation. In figure 4.8(a), the physical layout for a single bit CMOS data latch is shown. It uses a single data input line labelled "in" and the stored bit of information is continually output through a data output line labelled "out". Write control to the latch is obtained via the two control lines C and C_bar. The Hg simulation for the data latch is found in figure 4.8(b). Notice that data is written into the latch only during the period of time that control line C is held high.

Inputs			Outputs	
Carry in	Addend	Augend	Sum	Carry out
C_{i-1}	A_i	B_i	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

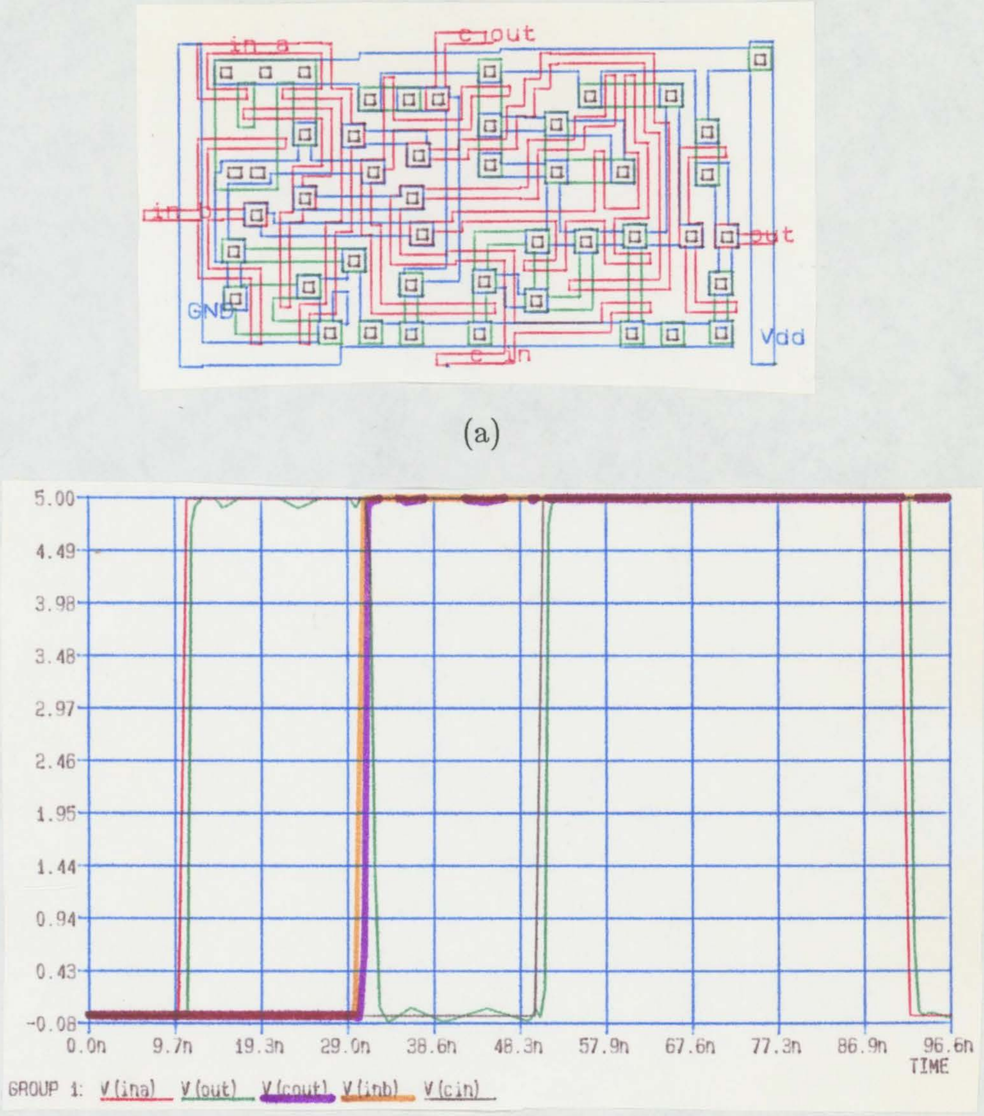
(a)



(b)

Figure 4.9

The truth table and logic level diagram describing the requirements and the implementation of a full adder unit.



(a)

(b)

Figure 4.10

The physical layout and SPICE simulation for the full adder cell. In the physical layout found if figure 4.10(a), the three input signal lines are labelled in_a, in_b, and c_in respectively. The two output lines are labelled sum and c_out. Both output lines are available after a two gate delay from the time of data input. Figure 4.10(b) shows the SPICE simulation for the adder operation as a function of various input signals. From a closeup of this figure it is apparent that the sum and c_out are available after approximately 2.4 ns.

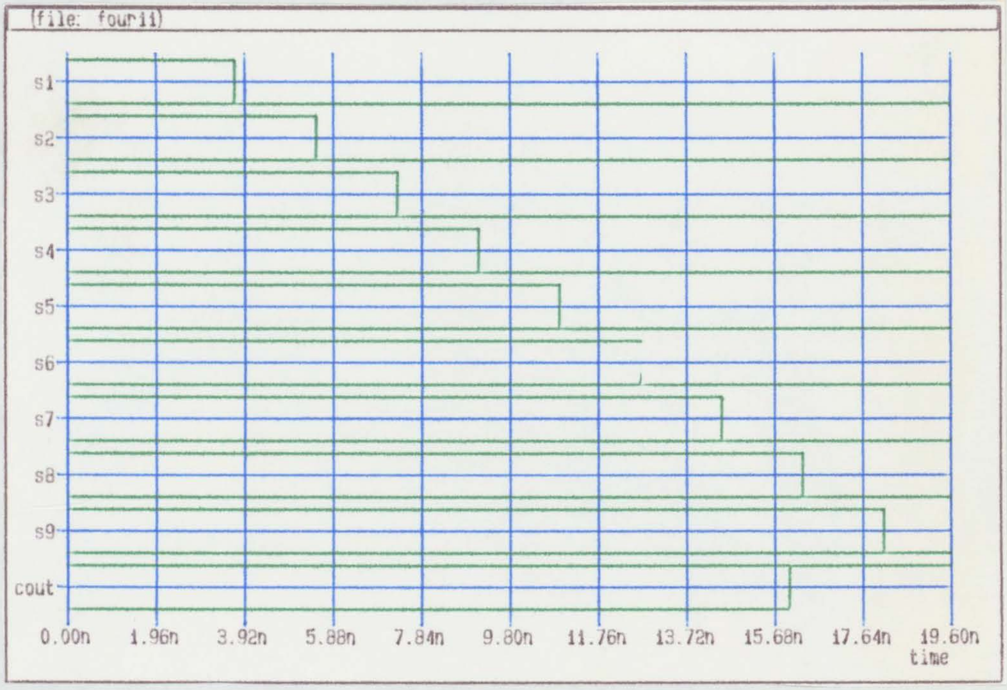


Figure 4.11

An Hg timing simulation for a 9-bit-parallel adder. The bit-parallel output lines are labelled labelled s1 - s9. The simulation indicates a worst case 14-bit addition is performed in approximately 20 ns.

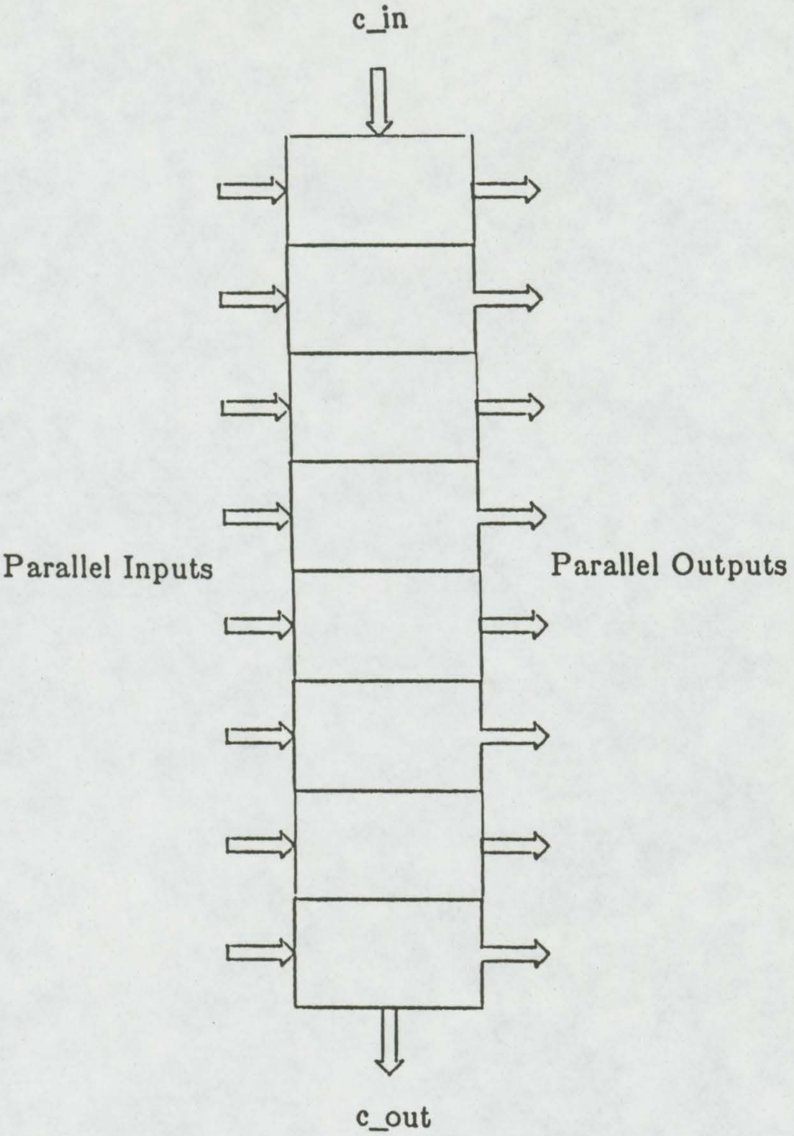


Figure 4.12

A block layout diagram for the 9-bit parallel adder. Each of the blocks is made up of the full adder cell found in figure 4.10(a). The inputs and outputs are in a bit parallel fashion, and the c_out line of each cell serves as the c_in line for the next cell in line. For normal operation the c_in line of the least significant adder cell is tied to ground.

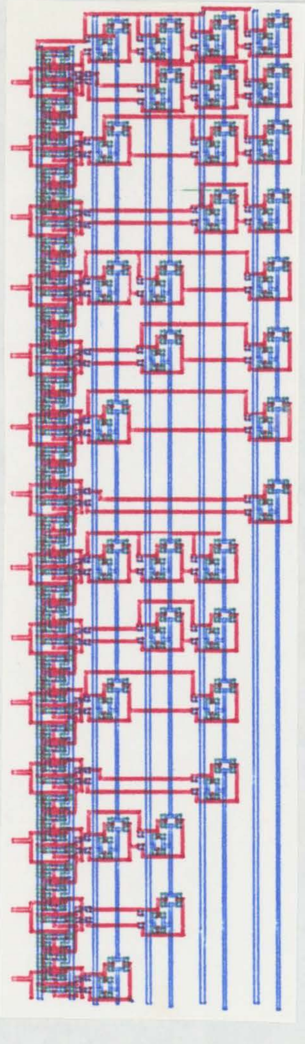
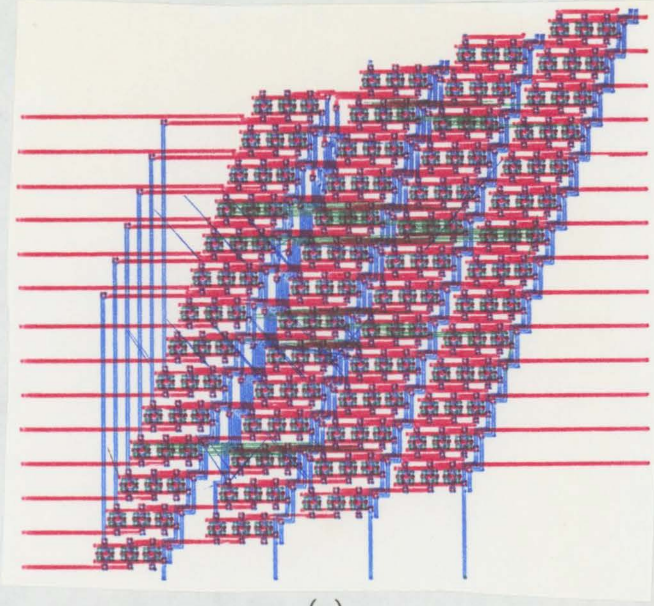
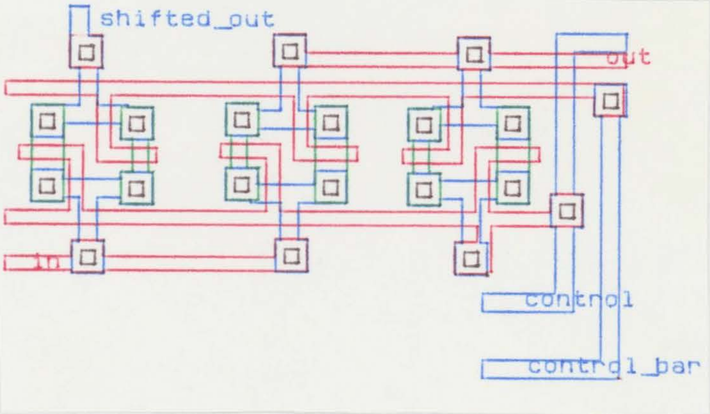


Figure 4.13

The physical layout design of the justifier unit. This unit is responsible for determining the exact location of the mantissa elements most significant bit and calculating the shift required to left justify the term. The mantissa enters the unit in a bit-parallel fashion, and the location of the most significant bit is determined via an interconnected set of NAND gates. The information as to the bit location is output on one of 15 output lines to a transfer gate switching network which outputs the desired hardwired response. This signal is used to update the exponential term, and control the final justification shift of the mantissa.



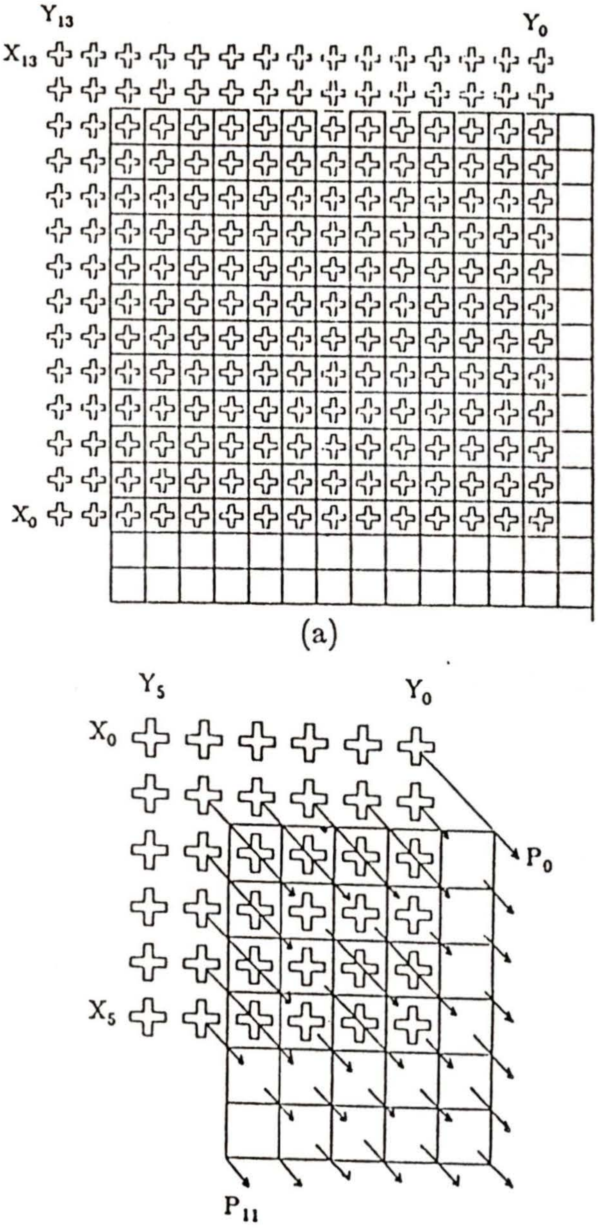
(a)



(b)

Figure 4.14

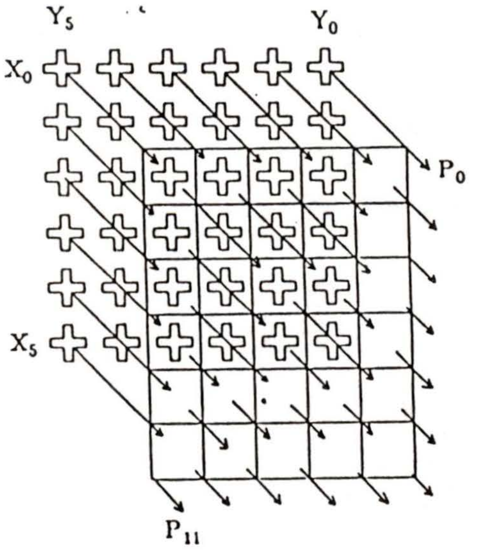
The shifter unit and the basic shift element. The shifter is capable of performing a 0-14-bit shift operation on the mantissa element. Data enters in a bit-parallel fashion from the left side of the unit, with the bottom line representing the most significant bit. Through the application of appropriate control signals which enter the unit from the bottom, a 0-14-bit shift is obtained. A detailed description of shifter operation is found in section 4.6.4. Figure 4.14(b) illustrates the basic shift element used. This element is comprised of three transfer gates and performs a single bit shift upon receiving the proper control signal.



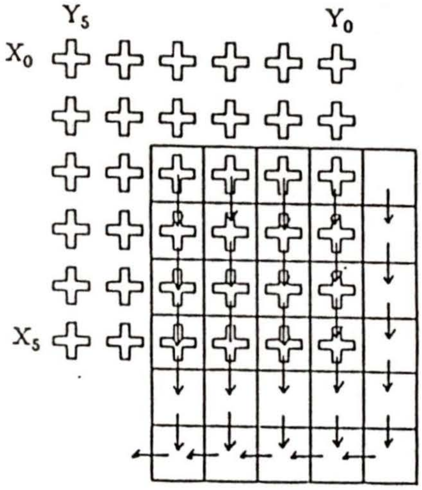
(b)

Figure 4.15

The block layout and signal propagation path for the parallel multiplier. The parallel multiplier is made up of an array of full adder cells indicated by small squares while the crosses indicate simple NAND operators. In figure 4.15(b) one of the signal propagation paths for the intermediate sum term for a 6-bit parallel multiplier are shown. The sum and carry paths for the 14-bit multiplier follow the same format. P_{11} represents the most significant product term, and P_0 the leastmost significant.



(c)



(d)

Figure 4.15 cont'd

Signal propagation paths for the 6-bit parallel multiplier. Figure 4.15(c) shows the diagonal propagation path for the second set of intermediate product terms. Figure 4.15(d) shows the propagation for the c_out signals which propagate vertically downward, and horizontally to the left.

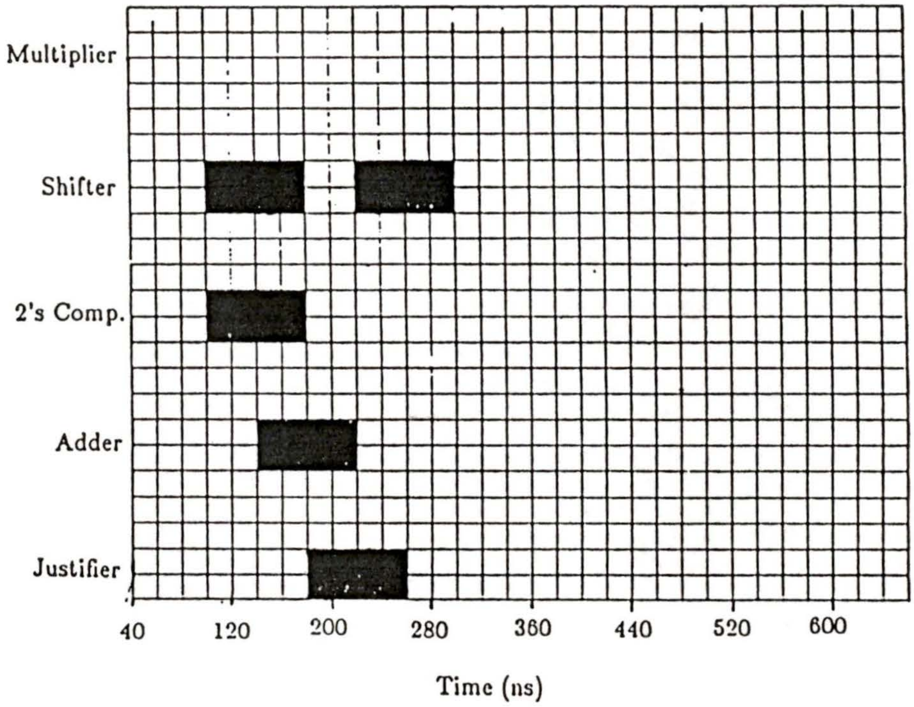


Figure 4.16

In this figure the operation of complex floating-point addition is described by illustrating which of the mantissa pipeline's processing elements is required to operate when, in order to perform the operation. Processing concurrency is seen when multiple processors are operating during the same time period. In a nonconcurrent processing system, each of the blacked out area's would occur in a sequential manner, greatly increasing the time required to complete the addition. The pipeline processors are shown to begin operation after the data has been read into the APU.

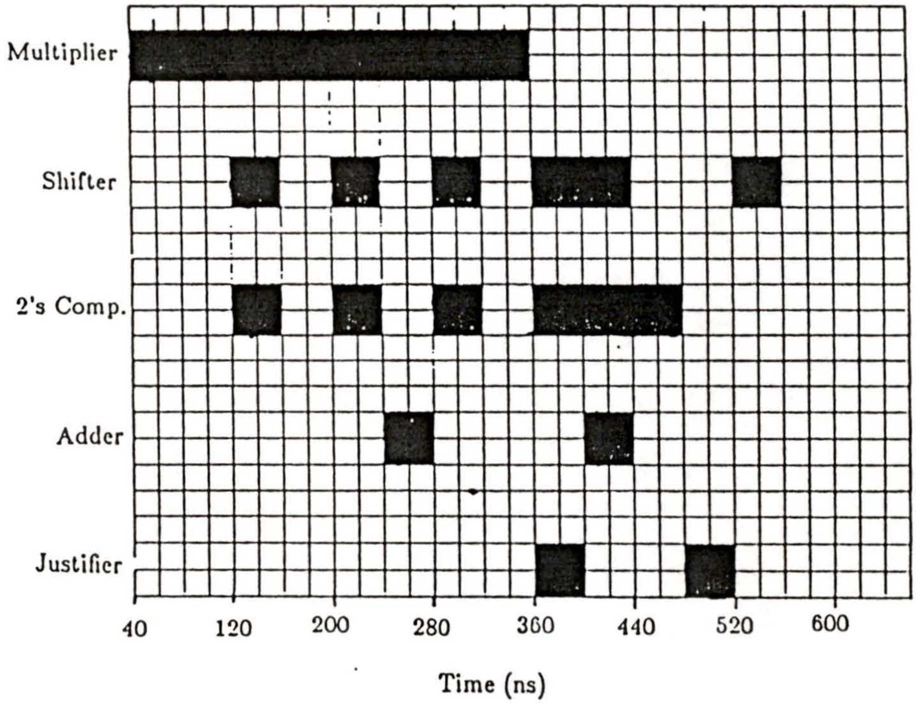


Figure 4.17

The mantissa operational timing graph for the case of complex number multiplication. In this figure the high levels of processing concurrency available through the MAP structured APU are seen. The multiplier in this figure performs four multiplications and is an obvious bottleneck to system performance. That is no other mantissa pipeline processors may operate until the first multiplication is complete, which requires a total of two mantissa clock cycles. The two product terms are output via the shift unit in a single operation.

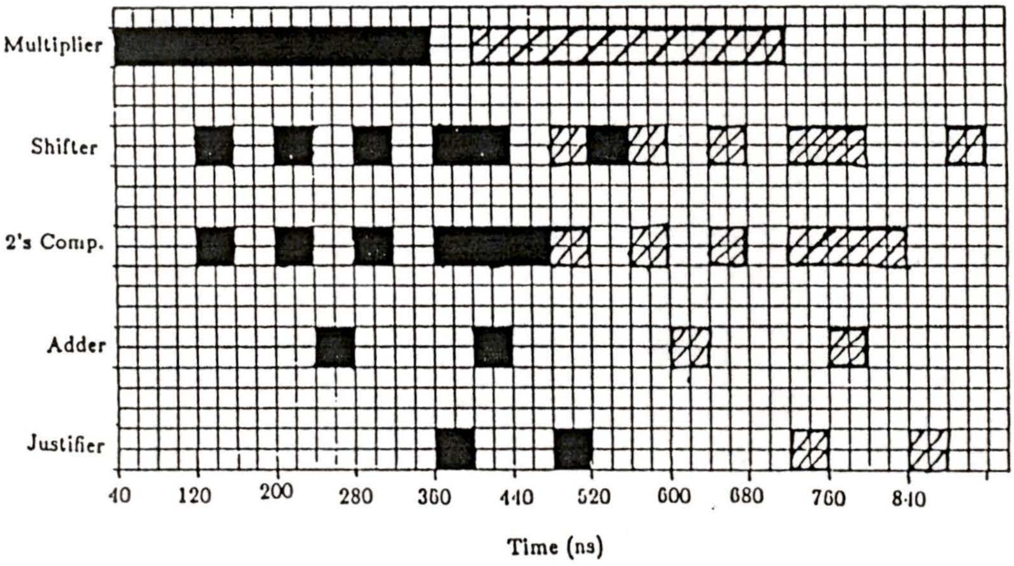


Figure 4.18

An operational timing graph for two pipelined complex number multiplications. In this figure only the mantissa operations are shown. The periods of element operation for the first complex number multiplication are indicated by blacked out squares, while those for the second are shown by hatched squares. As seen from this diagram, it is possible to begin the multiplication of the second set of numbers some four clock cycles before the first operation is complete without any conflict in element performance. By pipelining in this way, a 20% improvement in performance speed is realized.

Chapter 5

Control Logic Design

5.1 Introduction

Binary information is stored in digital systems either in scratch-pad or memory registers. The function of this information is either data or control in nature. Data elements are discrete sources of information which are manipulated through a series of microperations. The control information specifies the function and sequence of the micro-operations to be performed. The objective of control logic design is to develop the circuitry through which the desired sequence of operations may be implemented.

Figure 5.1 illustrates the general level of interaction between the system controller and the data processing elements. The control unit is a sequential logic unit responsible for generating the sequence of micro-operations to be sent to the processing unit. Its operation may be influenced through external control lines or by the present status conditions of the processor. From these input signals, the control unit determines the next micro-operation to be implemented. Implementation of the control unit is generally realized through one of four methods [6,13]:

1. The one flip-flop per state method.
2. The sequence register and decoder method.
3. PLA control.
4. Microprogram control.

This implementation required the design of a microprogram controller. Reasons for this choice and a description of microprogram controller operations, are given in section 5.2. Section 5.3 introduces the basic processor control requirements. In this section the application of the microprogrammed control unit to the case of the FFT node is discussed. This implementation utilizes two control units in a master-slave configuration, with an APU controller acting as "slave" to the node-level "master" controller. These units are responsible for the control of the arithmetic processor unit and the node-level data routing, respectively. For the sake of convenience we consider the two control units separately in sections 5.4 and 5.5. Section 5.5 also discusses how the two units are integrated together to realize the final master-slave configuration.

5.2 Microprogram Control

The function of the control unit is to generate a sequential set of microinstructions to be used in controlling processor data manipulation. Prime considerations in deciding on the method of control are size, flexibility and speed. Both the one flip-flop per state and the sequenced register and decoder techniques are examples of hard-wired controllers [6]. As such, they are best suited for simple task-specific control applications. Any changes in the control logic sequence requires extensive rewiring to fulfill the new output requirements. In the case of the processor node controller, the control sequence requirements are dependent on the processor's location within the network. High control flexibility becomes a requirement, thus eliminating the one flip-flop per state and the sequenced register and decoder methods as potential candidates for control implementation.

Although quite fast and highly flexible, a programmable logic array or PLA, also proves to be impractical for implementation. As will be shown later, some 29 output control lines need to be produced from approximately 12 input lines. A PLA capable of accommodating these requirements would result in two logic planes of approximately 900 microns by 1100 microns in a static CMOS implementation. Such a size proves to be impractical for controller implementation. A microprogram controller however while not as fast as the PLA, offers a highly flexible and size-efficient alternative for achieving control logic implementation.

Microprogram control is based on a finite set of fixed-length binary variables known as control words. Each control word is responsible for performing a specific micro-operation within the processor. A microprogrammed controller stores these control words in memory, each control word of memory being referred to as a microinstruction. A microprogram is made up of a sequence of microinstructions, and is responsible for carrying out the more complex processor functions. Figure 5.2 shows the general block configuration of the microprogram control logic.

The microprogrammed controller is divided into three components, the control memory, the control address register, and the next address generator. Once established, alteration of the microprogram is seldom required. For this reason it may be stored in ROM. Each microinstruction is assigned a unique address in the ROM, and the microprogram is then realized through subsequent read operations of the appropriate ROM addresses. Once the micro-operation has been performed, the control unit must determine the location or address of the next microinstruction to be called.

Generation of this address takes place in the next address generator. The next address generator is comprised of a counter, a logic block, and a set of RAM. In most cases the next address is generated sequentially from the counter unit. This need not always be the case however. The address generated from this unit may also be a function of the external input conditions, processor status conditions or control bits from the current microinstruction. Once generated, the new address is stored in the control address register, where it is used to call the next microinstruction.

In more sophisticated control schemes the microinstruction may contain secondary address locations as well as a set of operational control bits. These control bits are commonly referred to as the OP-code. Such a microcode format allows microinstructions such as return to zero, jump to, and conditional operations to be performed. As an example consider the case of the conditional operation. The OP-code of such a microinstruction may signal the next address generator to compare a value in memory to that of the counter or some other status variable of the processor. The next address to be input into the next address register is dependent on the nature and the outcome of this comparison. It may be either the next address in the counter sequence, or it may be the secondary address location specified in the microinstruction. In this way highly sophisticated microprograms may be implemented. The microprogrammed controller designed for this implementation requires a simpler microinstruction format than that described above. The format used contains no secondary address locations, and uses only a single OP-code bit. A detailed discussion of the microinstruction format is found in section 5.4.

5.3 Control Requirements

The sequence of operations performed by each node is a function of the node's location within the network. Recall that in general, stage M is responsible for performing an input shift of 2^M. Thus for a 1024-point FFT at stage $M=\log_2 N$, a 512-point shift is realized. Notice that even if inputs to the first stage are all real numbers, after multiplication by a twiddle factor input to the subsequent processing stages becomes complex. This incongruity in data types is handled by treating all data elements subsequent to their initial input into the first processor as being complex. Data transfer on- and off-chip is treated as a two-clock cycle operation where the real input elements are considered as complex-numbers with an imaginary component equal to zero. This is necessary to insure proper synchronization of data between processors. If this were not the case, it is conceivable that a real data element might mistakingly be read as the imaginary component of the preceding datum. The procedure described above, although slow, ensures a consistent data exchange.

In general the M^{th} processing node is responsible for performing and repeating the following sequence of operations (see Chapter 2):

1. Read a single input data element and pass it on directly to off-chip RAM or FIFO. This operation continues for a total of 2^M read operations.
2. Complex values are simultaneously read from the input port, and from the off-chip memory. The sum and difference terms are calculated with the sum term being passed on to the next processor node, and the difference term being passed back to the off-chip memory. This step begins with the completion of step 1 and continues for a total of 2^M steps.

3. In this implementation two complex multiplications occur, first the twiddle factor update followed by the multiplication of the twiddle factor with the difference term. These two multiplications are pipelined together as described in Chapter 4, and the product term is passed on to the next processing stage. Again this continues for a total of 2^M steps.

Notice that except for the initial loading of data into the stage, step 3 and step 1 may occur simultaneously. In performing the FFT, the processing steps sequence as follows: step 1, step 2, step 1&3, step 2, step 1&3, step 2, ... Each stage is required to repeat the steps in sequence a total of $2^{(\log_2 N-1)-M}$ times, to complete the FFT calculation. In general, steps 2 and 3 are microprogram controlled via the APU while step 3 is realized by means of sequential logic located in the master controller unit.

5.4 APU Controller

To achieve control over the APU data flow, two sets of control lines are required, one for the mantissa pipeline and the other for the exponential pipeline. Recall from Chapter 4 that for optimum time performance it was necessary for the exponential pipeline to operate at twice the clock rate of the mantissa pipeline. To avoid the added complexity of synchronizing and implementing two controllers of different clock rates, it was decided to use a single controller for both pipelines. To achieve the proper timing ratios, the controller is timed to the requirements of the exponential pipeline, and those control bits responsible for mantissa operation are repeated over two microinstructions.

Control design for the multiple access pipeline structure is a complex undertaking. In order to realize the full potential of this structure, bus access control must be made concurrent with pipeline control operations. In order to achieve this level of control concurrency, the microinstruction must be capable of addressing each pipeline element simultaneously. In this implementation each APU processing element is bound by intermediate data latches. Control over the processing elements is realized through the appropriate application of read/write commands to these latches. Each latch bank requires two control lines, one each for the read and write commands. In addition to the latch bank control lines, a single control line is required for each bus access port. Thus for the case of the mantissa pipeline a total of 28 control lines are required for the 14 latch banks, plus 7 control lines required for bus access. For the exponential pipeline a total of 20 control lines are needed. For parallel control operation then a microinstruction word of 55 bits would be needed. Obviously this is far too many control lines to handle efficiently.

To reduce the number of control bits required, the microinstruction was coded into three groups of 4-bits each and one 5-bit unit for the mantissa pipeline, and three four-bit units for the exponential pipeline. In this way the total number of control bits required was reduced to 29. A particular 4-bit group of the microinstruction is responsible for the control of the read/write commands to the data latches surrounding a particular pipeline stage. Each partitioned area is assigned an individual decoder which decodes the four appropriate bits of the microinstruction into the required control lines. The 5-bit code of the mantissa and one 4-bit code for the exponent are responsible for controlling pipeline access to the data bus. Access is restricted so that in each pipeline, no more than two pipeline ports are opened to the data bus at one time. Tables 5.1 (a-b) illustrate the partitioned microcode instructions and their associated operations for both the exponential and mantissa pipelines. The locations of the referenced elements in this table are found in figure 5.3. A single OP-code bit is also included in the APU microinstruction. Set to zero, this bit signals the next address generator to sequence through to the next microinstruction as designated by the generator's counter.

When set to one, the bit signals the next address register to read a value determined by external input bits. This point is elaborated on in section 5.4.1.

Having established a set of microinstructions it remains to construct the desired microprograms from them. Two microprograms have been constructed, one for a pipelined pair of complex-number multiplications, and one for a complex-number sum/difference determination. Construction of these programs utilized no special programming techniques, and the final versions were realized through inspection. The complete microprograms for the above operations are found in tables 5.2 and 5.3.

5.4.1 The Next Address Generator

The next address generator is responsible for determining the location of the next microinstruction to be called. It consists of a counter, a set of three 7-bit ROMs, and a two-dimensional logic block. Output to the next address register is a function of the generators counter value, the microinstruction's single OP-code bit, and four control lines. The control lines originate from the processors master controller. These control lines are comprised of a single Data Valid signal and three functional request lines. Upon reception of a high data valid signal from the master controller, the generator counter is enabled and begins to count with each clock cycle. Its sequenced output serves as the address location for the next microinstruction. Of the three 7-bit registers, two contain the address locations of the first line of microinstruction for the sum/difference and complex-number multiplication routines. The third register contains the address to which the complex-number routine must return to in the event of performing multiple pipelined multiplications. The functional request lines are used to enable the appropriate ROM for reading.

The microinstructions OP-code bit plays an important role in realizing the proper control sequence. A low OP-code bit signals the generator to increment the counter and to use the counter value as input to the next address register. A high OP-code bit indicates that the next address register is to receive input from one of the 7-bit generator registers. The master controller ensures that the appropriate register is enabled for reading at the required time. The counter is also set to the new address value, and the sequential count continues from this point. Notice from the listings of the two microprograms that the OP-code is only set high at the completion of the microprogram, after the results have been determined. Thus a high OP-code bit also serves as a valid data output signal, which tells the master controller that data is ready for output to off-chip memory or to a subsequent processor. Figure 5.4 (a-b) shows the block layout of the APU controller unit and its associated Hg simulation for the sum/difference microprogram.

5.5 The Node Level Master Controller

The node-level master controller unit is responsible for on- and off-chip data routing, as well as for coordinating the APU operations with the rest of the processor. Control requirements at this level are few, and the implementation is realized through an 11-bit counter in conjunction with a two-dimensional logic block. Figure 5.5 illustrates a block diagram of the master controller unit, indicating its various input and output control lines. These lines will be defined as they appear in the following discussion. Two clock lines are input to the control unit, one 50 MHz clock for the APU and a second 5.2 MHz clock which controls data input and output. Both clocks are common to all the network processors. The 5.2 MHz clock is based on twice the cycle rate of the sum/difference operations and it is the maximum rate at which data may enter or leave the processor. Notice that in general these clock rates deal with complex-number data elements, therefore real data entering the first processor stage must be clocked at an input rate of 2.6 MHz.

The master controller counter is enabled on each high signal of the 5.2 MHz clock. It is triggered either by a high Data1 Valid input signal during this period or in the event that the flow of input data has stopped it is triggered by the Data Valid out signal from the APU. The Data1 Valid, or DV1 signal accompanies the data and it originates either from the previous processing stage, or from the original source of data. It indicates to the processing unit that valid data is available for reading at the input ports. The counter bit of principle concern is the $(M+2)^{th}$ bit, where M is the network stage at which the processor is located. We look at the M+2 bit of the counter register because the value of M runs from M=0 upward. M=0 therefore corresponds to the first bit in the counter register This counter bit is used to determine when the appropriate number of operations have been performed, i.e. data shifts, sum/difference calculations, or multiplications). Recall from section 5.3 that the number of operations performed in each processing step was 2^M Thus the counter bit of concern should have been the $M+1^{th}$ bit. However since the complex data elements are read in over two clock periods the counter counts twice for each element. The bit of concern must then be shifted one location to the left in the counter register, making the M+2 register bit the bit of significance. For convenience we shall henceforth refer to the $(M+2)^{th}$ bit of the counter as bit A. This bit is identified by decoding the set of four external input lines which describe the location of the processor within the network.

Upon reception of a high DV1 signal the data at the parallel input ports A_{0-23} is read and input into a temporary input register. In a similar manner a Data2 Valid, or DV2 signal indicates that data from off-chip memory is available for reading. These values enter the processor via the three-way multiplexed I/O ports, B_{0-23} and may be read simultaneously with ports A_{0-23}. Once stored on-chip, operations to be performed on the data are dependent on the present operating status of the processor. This status is determined by analyzing bit A, and the DV1 and DV2 lines in conjunction with the processing requirements outlined in section 5.3

Step 1

A low bit A in conjunction with a high DV1 line indicates that valid input data is ready to be read and immediately output to the off-chip memory. This represents step 1 of the processing requirement outlined in section 5.3.

Step 2

A high bit A and high DV1 and DV2 signals indicate that the appropriate data shift has been realized, and the sum/difference calculations outlined as step 2 may commence. In this event the master controller outputs a high data valid line to the APU controller, and enables the appropriate register in the APU address generator for reading. As the APU controller begins sequencing, the first line of microinstruction directs the next address register to read its value from one of the three ROMs located in the next address generator. The register enabled by the master controller in this case contains the memory location of the first line of the sum/difference microprogram. This register remains enabled until otherwise instructed by the master controller. The master controller counter continues to count, performing a single sum/difference calculation per counter increment. After a total of 2^{M+1} incrementations, bit A returns to zero and the data valid line to the APU controller is lowered. A total of 2^{M} sum/difference terms have been calculated and step 2 of the processing requirements is complete.

Step 3

In processing step 3, complex-number multiplication and twiddle factor updating begins immediately upon the completion of step 2. This state is indicated by a low bit A and a high DV2 line. The high DV2 line indicates that the off-chip data, the difference terms from step 2, are now ready to be multiplied by the twiddle factor. The master controller sends a data valid signal to the APU controller and the APU generator memory containing the address for the multiplication routine is enabled. On the next 5.2 MHz. clock cycle, the memory containing the address for step 9 of table 5.3 is enabled. The microprogram is written in such a way that by returning to this point multiple complex-number multiplications may be pipelined together. The products of this multiplication are then output to the next processing stage. In the event that the DV1 line is high during this period, processing step 1 may be performed concurrently with step three. Once bit A returns high the appropriate number of multiplications have been carried out and the operation is halted. Step 2 may or may not be repeated depending on the status of the DV2 line at this time.

Thus far, discussion of the master controller operation dealt with co-ordinating the APU controller and overall algorithm realization. The master controller is also responsible for all node-level data routing operations. As mentioned, data may be input to the processor either through the input ports A_{0-23} or by way of the multiplexed I/O ports B_{0-23}. Data is output through ports B_{0-23} as well. Upon reception of high DV1 or DV2 signals the master controller responds by returning a high read A and or a high read B signal to the input ports. The data is read directly into a set of temporary data registers. The processor node contains four pairs of temporary input data registers, each of width 23 bits. In this way the processor can hold up to four complex data elements for processing. Write access control over the input registers is achieved by decoding the four least significant bits of the master controller counter in conjunction with the DV1 and DV2 lines. The write commands to the input latches of the mantissa and exponent pipelines in conjunction with bit A serve as the read commands to the input registers. For the case of on-chip twiddle factor updating, the initial twiddle factor seed values for a twelve stage network are stored on-chip in ROM. Recall from Chapter 2 that these seed values represent the factor by which the twiddle factor is updated at each stage. For example in the $M=\log_2 N$ stage the twiddle factor seed is ω^1 and the twiddle factor is updated by a factor of ω for each multiplication. Access to the required twiddle factor seed register is obtained by decoding the four input bits describing the processor's network location. This is the same decoder used to identify bit A.

It is important to note that I/O ports B_{0-23} are responsible for three way communications. Data is both output and input to and from off-chip memory and is output to the next processing stage. The data flow through the I/O port is realized by the following sequence:

1) Data is read from off-chip memory.
2) Data is written to off-chip memory.
3) Data is sent to next processing stage.

The first two operations each occur within a 80 ns period. This corresponds to 4 clock cycles of the 50 MHz clock. By appropriately decoding the output of the APU controller's 50 MHz counter in conjunction with the 50MHz clock pulse, the timing sequence required for proper I/O port operation is achieved. The output data valid line is lowered during a write to memory. In this way the subsequent processing unit cannot mistakingly read data being written to memory.

On first inspection, the previously-described control system might appear unfeasible due to the fact that data is input to the processor at a rate based on the sum/difference calculation while node calculations such as complex-number multiplication take longer to perform. This apparently results in the problem that data is being input faster than it can be processed, thus resulting in processing bottlenecks. Such is not the case. The algorithm implemented is such that after the first processing stage completes the required sum/difference calculations, all input data elements are internal to the network (see Chapter 2). A processing stage M then performs 2^M complex-number multiplications while stage M-1 performs $2^{(M-1)}$ multiplications followed by $2^{(M-1)}$ sum/difference calculations. Since the multiplication process takes longer to complete than the sum/difference determination, we have a situation in which data from processor M-1 is processing faster than or at the same rate as processor M. Thus the case in which a processor outputs data faster than the subsequent processor can receive never occurs. In fact the reverse happens and the rate of data flow is governed by the rate it is processed in the previous processing stage.

While the nodes themselves represent synchronous systems data transference on the network level more closely resembles that of self-timed systems. In this case data is not read by the processor unit until its preceding counterpart has valid data ready for output. This way network processors begin processing the data as soon as it becomes available. If the network data transfer was purely synchronous, the data transfer rates would have to be timed around the slowest computational time unit, in this case the complex-number multiplication time. Data transference would be at a rate of approximately 1 MHz., resulting in a significant drop in system performance. Calculations indicate that by utilizing the self-timed network data transfer scheme a 43% improvement in performance times is realized over the totally synchronized system.

5.6 Summary

In summary this chapter has presented the master-slave controller configuration used by the processing node. The concepts of microprogrammed control were outlined, and an encoded microinstruction for use by the APU microprogrammed control unit was developed. The encoded microinstruction was necessary in order to efficiently address each of the APU data latches simultaneously. In this way bus access and pipeline operations are able to be performed concurrently. Tables 5.1(a-d) outline the microinstruction operation and its associated binary representation. Two microprograms were also written to perform the pipelined complex-number multiplication, and the sum/difference calculations.

Operation of the node-level master controller was also outlined in this chapter. The master controller unit is responsible for node-level data routing, and ensuring that the proper sequencing of APU microprograms is obtained. Overall, the master-slave configuration provided a fast simple means of control over a sequenced set of concurrent operations.

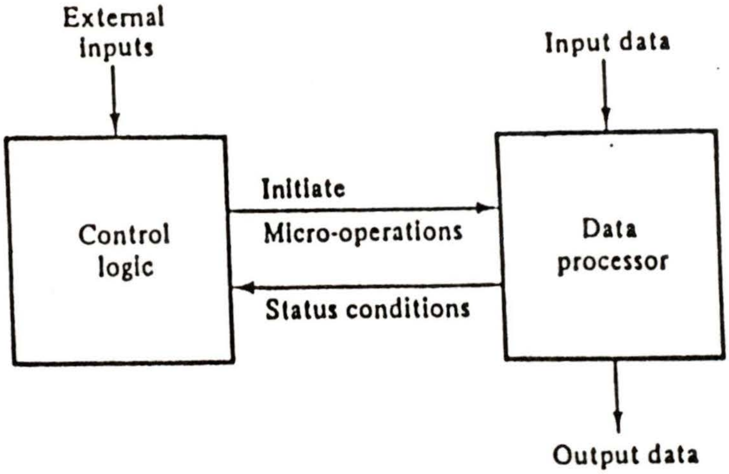


Figure 5.1

Control and data processor interaction. Note that the control unit output is a function of both external inputs and present status conditions of the data processing unit.

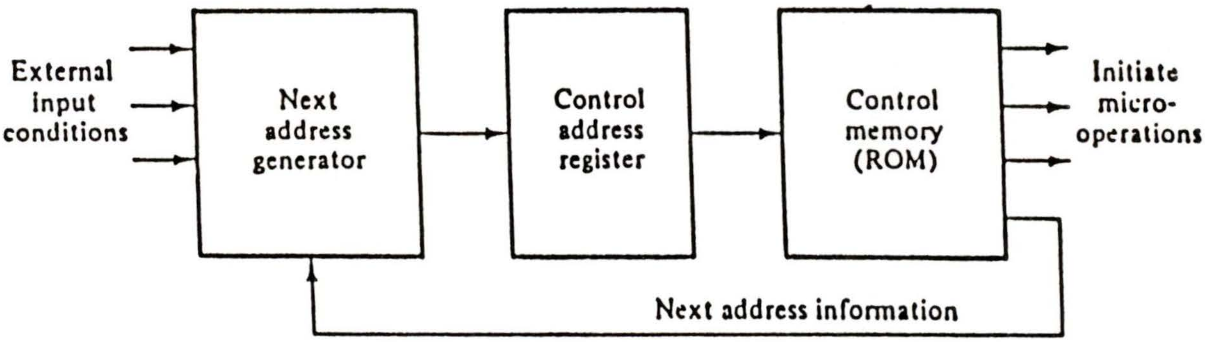


Figure 5.2

The general configuration of the microprogram control unit. In the design for this implementation, the next address in- formation line represents the single op-code bit of the micro- instruction.

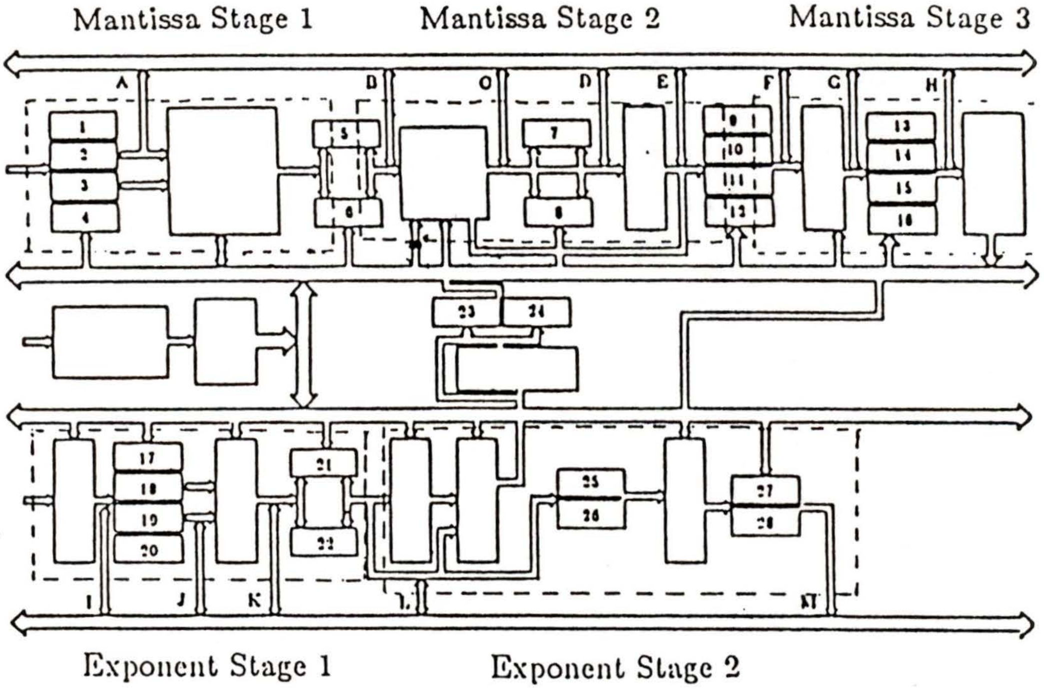


Figure 5.3

The general sectioning of the APU into segments corresponding to the coded partitions of the microinstruction format. In this figure the numbered components refer to intermediate word memory locations and the letters to the various bus access ports. The processing elements are left unlabelled. This figure should be used to clarify the microinstruction operations specified in tables 5.1 to 5.3.

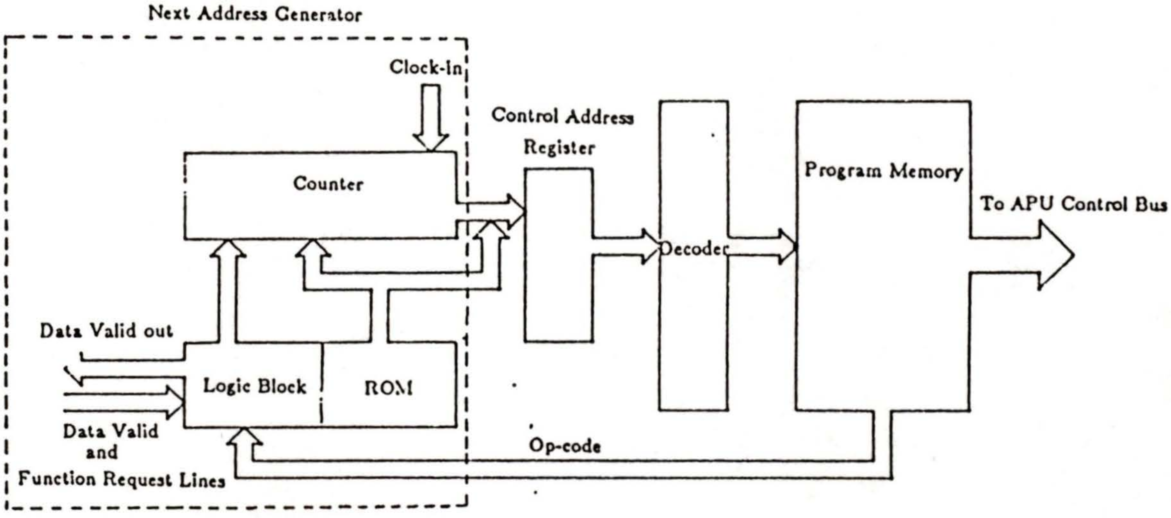


Figure 5.4

The block layout for the APU controller unit.

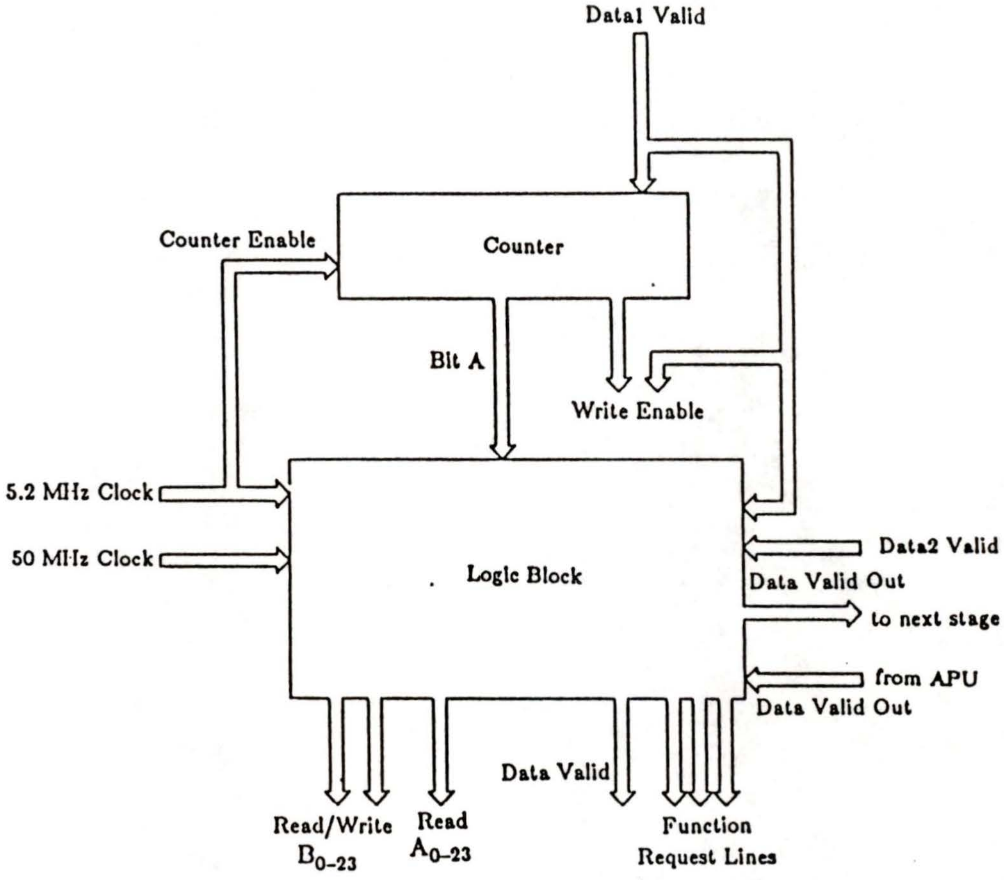


Figure 5.5

Block layout of the node level master controller unit.

Mantissa Pipeline Control Code				
Operation	Bit Code			
	stage 1	stage2	stage3	Bus Access
No Op	0000	0000	0000	00000
Write 1	0001	xxxx	xxxx	xxxxx
Write 2	0010	xxxx	xxxx	xxxxx
Write 3	0011	xxxx	xxxx	xxxxx
Write 4	0100	xxxx	xxxx	xxxxx
Read 1	0101	xxxx	xxxx	xxxxx
Read 2	0110	xxxx	xxxx	xxxxx
Read 3	0111	xxxx	xxxx	xxxxx
Read 4	1000	xxxx	xxxx	xxxxx
Read 1,3 Write 5	1001	xxxx	xxxx	xxxxx
Read 2,4 Write 6	1010	xxxx	xxxx	xxxxx
Read 1,4 Write 5	1011	xxxx	xxxx	xxxxx
Read 2,3 Write 6	1100	xxxx	xxxx	xxxxx
Read 1,3 Write 4,5	1101	xxxx	xxxx	xxxxx
Read 2,3 Write 1,6	1110	xxxx	xxxx	xxxxx
Read 7 Write 9	xxxx	0001	xxxx	xxxxx
Read 8 Write 10	xxxx	0010	xxxx	xxxxx
Write 7	xxxx	0011	xxxx	xxxxx
Write 8	xxxx	0100	xxxx	xxxxx
Read 5	xxxx	0101	xxxx	xxxxx
Read 6	xxxx	0110	xxxx	xxxxx
Read 5 Write 7	xxxx	0111	xxxx	xxxxx
Read 6 Write 8	xxxx	1000	xxxx	xxxxx
Write 9	xxxx	1001	xxxx	xxxxx
Write 10	xxxx	1010	xxxx	xxxxx
Write 11	xxxx	1011	xxxx	xxxxx
Write 12	xxxx	1100	xxxx	xxxxx
Read 9	xxxx	1101	xxxx	xxxxx
Read 10	xxxx	1110	xxxx	xxxxx
Read 11	xxxx	1111	xxxx	xxxxx

Table 5.1(a)

This table illustrates the microinstruction control code, and the corresponding micro-operations for the mantissa pipeline stages 1 and 2. A coded four bit segment of the microinstuction is used to achieve independent control over each of the mantissa pipeline segments outlined in figure 5.3. Since each processing element is separated by a set of data latches, propagation down the pipe is controlled by application of appropriate read/write commands to the data latches. The operation write 1 for example means write to data latch 1, where data latch 1 is defined in figure 5.3. The x's of the instruction refer to "don't care" states.

Mantissa Pipeline Control Code (cont.)				
Operation	Bit Code			
	stage 1	stage2	stage3	Bus Access
Write 13	xxxx	xxxx	0001	xxxxx
Write 14	xxxx	xxxx	0010	xxxxx
Write 15	xxxx	xxxx	0011	xxxxx
Write 16	xxxx	xxxx	0100	xxxxx
Read 13	xxxx	xxxx	0101	xxxxx
Read 14	xxxx	xxxx	0110	xxxxx
Read 15	xxxx	xxxx	0111	xxxxx
Read 16	xxxx	xxxx	1000	xxxxx
Read 9,10 Write 13	xxxx	xxxx	1001	xxxxx
Read 9,11 Write 14	xxxx	xxxx	1010	xxxxx
Read 9,11 Write 15	xxxx	xxxx	1011	xxxxx
Read 9,12 Write 16	xxxx	xxxx	1100	xxxxx
Read 9,11,13 Write 14	xxxx	xxxx	1101	xxxxx
Read 9,11,15 Write 14	xxxx	xxxx	1110	xxxxx

Table 5.1(b)

This table is essentially a continuation of table 5.1(a). It illustrates the micro-operation and corresponding microcode required for control over stage 3 of the mantissa pipeline.

Data Bus Access Control Code

Operation	Bit Code			
	stage 1	stage2	stage3	Bus Access
Open Port(s) A to Bus	xxxx	xxxx	xxxx	00001
Open Port(s) B to Bus	xxxx	xxxx	xxxx	00010
Open Port(s) C to Bus	xxxx	xxxx	xxxx	00011
Open Port(s) D to Bus	xxxx	xxxx	xxxx	00100
Open Port(s) E to Bus	xxxx	xxxx	xxxx	00101
Open Port(s) F to Bus	xxxx	xxxx	xxxx	00110
Open Port(s) G to Bus	xxxx	xxxx	xxxx	00111
Open Port(s) H to Bus	xxxx	xxxx	xxxx	01000
Open Port(s) A,B to Bus	xxxx	xxxx	xxxx	01001
Open Port(s) A,C to Bus	xxxx	xxxx	xxxx	01010
Open Port(s) A,D to Bus	xxxx	xxxx	xxxx	01011
Open Port(s) A,E to Bus	xxxx	xxxx	xxxx	01100
Open Port(s) A,F to Bus	xxxx	xxxx	xxxx	01101
Open Port(s) A,G to Bus	xxxx	xxxx	xxxx	01110
Open Port(s) A,H to Bus	xxxx	xxxx	xxxx	01111
Open Port(s) B,D to Bus	xxxx	xxxx	xxxx	10000
Open Port(s) B,E to Bus	xxxx	xxxx	xxxx	10001
Open Port(s) B,F to Bus	xxxx	xxxx	xxxx	10010
Open Port(s) B,G to Bus	xxxx	xxxx	xxxx	10011
Open Port(s) B,H to Bus	xxxx	xxxx	xxxx	10100
Open Port(s) C,D to Bus	xxxx	xxxx	xxxx	10101
Open Port(s) C,E to Bus	xxxx	xxxx	xxxx	10110
Open Port(s) C,F to Bus	xxxx	xxxx	xxxx	10111
Open Port(s) C,G to Bus	xxxx	xxxx	xxxx	11000
Open Port(s) C,H to Bus	xxxx	xxxx	xxxx	11001
Open Port(s) D,E to Bus	xxxx	xxxx	xxxx	11010
Open Port(s) D,F to Bus	xxxx	xxxx	xxxx	11011
Open Port(s) D,G to Bus	xxxx	xxxx	xxxx	11100
Open Port(s) D,H to Bus	xxxx	xxxx	xxxx	11101
Open Port(s) F,H to Bus	xxxx	xxxx	xxxx	11110
Open Port(s) G,H to Bus	xxxx	xxxx	xxxx	11111

Table 5.1(c)

Illustrated in this table is the coded 5 bit microinstruction segment required for control over the mantissa pipeline data bus access. The associated micro-operations are such that no more than two ports are open to the bus simultaneously.

Exponential Pipeline Control Code			
Operation	Bit Code		
	stage 1	stage2	Bus Access
No Op	0000	0000	0000
Write 17	0001	xxxx	xxxx
Write 18	0010	xxxx	xxxx
Write 19	0011	xxxx	xxxx
Write 20	0100	xxxx	xxxx
Read 17	0101	xxxx	xxxx
Read 18	0110	xxxx	xxxx
Read 19	0111	xxxx	xxxx
Read 20	1000	xxxx	xxxx
Read 17,19 Write 21	1001	xxxx	xxxx
Read 18,20 Write 22	1010	xxxx	xxxx
Read 17,19 Write 22,20	1011	xxxx	xxxx
Read 17,20 Write 22	1100	xxxx	xxxx
Read 18,19 Write 21	1101	xxxx	xxxx
Read 17 Write 18	1110	xxxx	xxxx
Read 19 Write 20	1111	xxxx	xxxx

Table 5.1(d)

This table gives the microcode for the three four bit segments of the microinstruction responsible for control over the exponential pipeline.

Exponential Pipeline Control Code			
Operation	Bit Code		
	stage 1	stage2	Bus Access
Read 22	xxxx	0001	xxxx
Read 21, 2's Comp.	xxxx	0010	xxxx
Read 23	xxxx	0011	xxxx
Write 23,25	xxxx	0100	xxxx
Write 25	xxxx	0101	xxxx
Write 26,27 Read 25	xxxx	0110	xxxx
Read 24	xxxx	0111	xxxx
Read 23 Write 24,26	xxxx	1000	xxxx
Read 25 Write 27	xxxx	1001	xxxx
Read 26,23 Write 28,25	xxxx	1010	xxxx
Read 23,2's Comp.	xxxx	1011	xxxx
Read 22,23	xxxx	1100	xxxx
Write 22	xxxx	1101	xxxx
Write 21 Read 22	xxxx	1110	xxxx
Write 22,23	xxxx	1111	xxxx
Open I	xxxx	xxxx	0001
Open J	xxxx	xxxx	0010
Open K	xxxx	xxxx	0011
Open L	xxxx	xxxx	0100
Open IJ	xxxx	xxxx	0101
Open IK	xxxx	xxxx	0110
Open IL	xxxx	xxxx	0111
Open JK	xxxx	xxxx	1000
Open JL	xxxx	xxxx	1001
Open KL	xxxx	xxxx	1010

Table 5.1(d) cont'd

This table is an extension of table 5.1(d) on the previous page. It continues to describe the microinstruction for the three four bit segments responsible for control over the exponential pipeline.

Pipelined Complex Number Multiplication Microprogram

Step	Exponential Bit Code			Mantissa Bit Code				
	stage 1	stage2	Bus Access	stage1	stage2	stage3	Bus Access	OP-code
step 0	0000	0000	0000	0000	0000	0000	00000	0
step 1	0001	0000	0000	0001	0000	0000	00000	0
step 2	0010	0000	0000	0010	0000	0000	00000	0
step 3	0011	0000	0000	0011	0000	0000	00000	0
step 4	1011	0000	0000	1101	0000	0000	00000	0
step 5	1010	0001	0000	1001	0000	0000	00000	0
step 6	0000	0010	0000	1001	0000	0000	00000	0
step 7	0000	0100	0000	1001	0000	0000	00000	0
step 8	0000	0011	0000	1010	0111	0000	00000	0
step 9	1100	0000	0000	1010	0001	0000	00000	0
step 10	1101	0001	0000 .	1010	0001	0000	00000	0
step 11	0000	0010	0000	1010	0000	0000	00000	0
step 12	0000	0011	0000	1011	1000	0000	00000	0
step 13	0000	1000	0000	1011	0010	0000	00000	0
step 14	0000	0000	0000	1011	0010	0000	00000	0
step 15	0000	0000	0000	1011	0000	1010	00000	0
step 16	0000	0111	0000	1100	0111	1010	00000	0
step 17	0000	0000	0000	1100	0001	0110	01000	0
step 18	0000	1001	1011	1100	0001	0000	00000	0
step 19	0001	0000	0000	1110	0000	0000	00000	0
step 20	0010	0111	0000	0010	1000	0000	00000	0
step 21	0011	0000	0000	0011	0010	0000	00000	0
step 22	1011	0000	0000	1101	0010	0000	00000	0
step 23	1010	0001	0000	1001	0000	1011	00000	0
step 24	0000	0010	0000	1001	0000	1011	00000	0
step 25	0000	0100	0000	1001	0000	0111	00000	0
step 26	0000	1010	1011	1010	0111	0000	00000	1

Table 5.2

This table illustrates the sequence of microinstructions required to perform a pipelined complex number multiplication. This microprogram is completed in a total of 26 steps, where each step corresponds to a single clock cycle of the 50 MHz clock. The program is configured so that a second complex number multiplication might be pipelined by returning to step 9 after completing step 26. Notice the high op-code bit serves to indicate a result is ready for output.

Sum-Difference Microprogram								
Step	Exponential Bit Code			Mantissa Bit Code				
	stage 1	stage2	Bus Access	stage1	stage2	stage3	Bus Access	OP-code
step 0	0000	0000	0000	0000	0000	0000	00000	0
step 1	0001	1101	0110	0001	0000	0000	00000	0
step 2	0010	1110	0110	0010	0000	0000	00000	0
step 3	0011	0010	0111	0011	0000	0000	00000	0
step 4	0100	1111	0110	0100	0000	0000	00000	0
step 5	0000	1100	0000	0101	1001	0000	01001	0
step 6	0110	1011	1001	0101	1001	0000	01001	0
step 7	0000	1000	0000	0111	1010	0000	01001	0
step 8	0000	0011	0000	0111	1010	0000	01001	0
step 9	0000	0011	0000	0111	1011	1001	01001	0
step 10	0000	0011	0000	0111	1011	1001	01001	0
step 11	0000	0111	0000	0110	1100	1011	01001	0
step 12	0000	0111	0000	0110	1100	1011	01001	0
step 13	xxxx	0111	1001	1000	1001	0000	01001	0
step 14	xxxx	0111	1001	1000	1001	0000	01001	0
step 15	0000	0111	0000	0110	1010	1100	01001	0
step 16	0000	0111	0000	0110	1010	1100	01001	0
step 17	xxxx	0101	1001	0000	0000	1101	01000	0
step 18	xxxx	0110	1001	0000	0000	0111	00000	0
step 19	0000	1010	0000	0000	0000	0110	00000	1

Table 5.3

This table illustrates the 19 step sum/difference microprogram. The x's in exponential bit code stage 1 found in steps 17 and 18 indicate that the operation is dependent of the outcome of cal- culations performed elsewhere in the pipeline.

Chapter 6

Results and Discussion

6.1 Introduction

The previous chapters have dealt with the design and control of an FFT processing node for implementation in the pipelined cascade FFT network. In this chapter the results of simulations performed on the node's processing and control elements will be discussed, and extrapolated in order to predict the performance of the network. These discussions are found in sections 6.3 and 6.4. A review of the various CAD tools and simulators used for system analysis precede these discussions and is found in section 6.2.

Design of the processing node was carried out in a three-micron ISO-CMOS technology offered by the Seattle Silicon Foundry. The final design contains approximately 17,000 transistors and occupies some 6.4 mm^2 of silicon area. The decision to design in a three-micron process was made when it was discovered that the processor node design in five microns would be too large for single-chip implementation. We also wanted our design implemented on the three-micron technology of CMC available this January. In the event that the design implementation was to be taken beyond the prototype stage, manufacturing would almost certainly be made in a 3- or 1.5-micron technology. Extrapolation of the three-micron simulation results down to 1.5 microns is the subject of section 6.5.

Actual implementation of the entire three-micron design has not yet been undertaken. The costs involved to finance a single three-micron fabrication prove to be prohibitive at this time. Instead, some 8 devices have been submitted in a five-micron technology to the Canadian Microelectronics Corporation for fabrication in their multi-project chip program. In this program, the CMC offers to canadian universities an opportunity for free five-micron design fabrication three times a year. However since this project entails implementing multiple projects on a single chip, the availability of silicon area and pin outs are extremely restricted. The components of this processor which were submitted for fabrication had to be broken up into smaller elements in order to comply with these restrictions. These submissions were designed such that through the appropriate configuration of the fabricated elements the entire APU might be reconstructed. Due to problems encountered with the Metheus VLSI Design work station, the May and September 1985 submission deadlines could not be met. These submissions have been made for the January fabrication run.

Since the actual device testing was not possible, the reported processor and network performance times are based on extensive circuit simulations carried out on the Metheus work station. Section 6.2 discusses the work station's available CAD tools and circuit simulators. The processor power requirements were estimated using the SPICE simulator, and are presented in section 6.6.

6.2 CAD Tools and Circuit Simulators

Design and analysis of the FFT processing node was carried out on a Metheus VLSI work station. During the course of the design, several operating problems were encountered with the Metheus work station. During a twelve month period between September 1984 and August 1985, the work station saw two hard disc failures, and severe power supply problems. These failures resulted in an intermittent system down time of approximately four months over the one-year period, and are the reasons behind the failure to meet the CMC submission deadlines. Since the last repair operation, performed in August 1985, the work station has been fully functional.

Actual circuit design of the processor was made using the Metheus physical layout editor, PHLED. Using this CAD tool the circuits of the type seen in Chapter 4 were constructed. Once completed the circuits were checked for any design rule violations using the built-in design rule checker. Following this the circuits were extracted from the PHLED format, and input into the standard CIF or simulator format. CIF, or Caltech Intermediate Format, is the standardized format used to describe circuit layout. It is used by the foundry to generate the masks required in the fabrication sequence. In the simulator format, the physical layout of transistors has been mapped into an interconnecting set of devices and node specifications required to perform simulation of the circuits.

The circuit simulators used in this implementation were Hg and SPICE. SPICE is a sophisticated circuit simulator which models circuit performance at the device level of operation. All results pertaining to device performance such as voltage rise and fall times, gate delays, current leakage, and power requirements were obtained using SPICE. Because the SPICE simulator analyzes circuits in such depth, simulation is restricted to small circuits. A forty transistor circuit appears to be about the largest size that SPICE can handle efficiently. SPICE simulations were used to analyze the circuit performance of all the microcellular and cellular designs described in Chapter 4.

The second simulator which saw extensive use was Hg. Hg, pronounced mercury, is a logic level timing simulator for MOS integrated circuits. It was used to verify the operation of the complex combinational and sequential logic circuits making up the processing node. Output from Hg is in the form of sequenced 0,1 logic levels, and its maximum simulation capability is about 7000 transistor circuits. This presents a problem in that no simulator currently available on the Metheus work station is capable of simulating circuits the size of either the processing node or the APU. In order to circumvent this problem several simulations were made on overlapping processing segments of suitable size. The output of the first simulated segment served as the initial conditions for the second simulated segment, and so on. In this way, verification of the circuit operation was made to as high a level of confidence as is presently possible.

6.3 APU Performance Results

The APU is a 14000 transistor design which occupies a silicon area of 6200 microns × 5600 microns. It utilizes a microprogrammed controller unit, and operates on a clock cycle of 50 MHz. While direct simulation of the APU was not possible, its operational verification was carried out using the technique described in section 6.2. Verification of the APU's component cells such as multipliers and shift units is found in sections 4.1 to 4.5. The description of the controller sequencing for the complex-number multiplication and the sum/difference microprograms is found in Chapter 5.

For simulation purposes, the APU was broken down into five smaller units, corresponding to the control sections described in Chapter 5. The clock period taken for these simulations was 50 MHz, the same as that for the control unit. From these simulations it was found that the product of two complex-numbers was determined after 520 ns. while the sum/difference term was calculated in 380 ns.

Figure 6.1 illustrates the timing diagrams associated with the complex-number multiplication microprogram shown in Table 5.2. Not shown in this diagram are the intermediate operations of shifting, two's-complementation, or justification. These operations are accounted for in the periods of undetermined states in the intermediate elements as indicated in the diagrams. In these diagrams, the undetermined states are represented by hatched lines. Verification of these two microprograms also verifys the simpler arithmetic operations from which they are made up. Table 6.1 lists some of the more common floating-point arithmetic operations which the APU is capable of performing, and their associated processing times.

6.4 Network Performance

In this section the results of the APU performance are extrapolated to predict the network performance results. The net processing time for the FFT is given by the sum of the node delays encountered by the slowest element to propagate through the network. Recall from Chapter 2 that the net processing time may be approximated by:

$$T_{\mathrm{Network}} \approx \sum_{M=0}^{\log_2 N/2\ -1} (T_{\mathrm{sum/diff}} + T_{\mathrm{comp.Mult.}})2^M \tag{6.1}$$

A more accurate approximation takes into account the fact that each element is also subjected to additional delays from data transference to and from the on-chip temporary registers. In a worst case situation this results in an additional delay of three clock1 cycles per stage. Thus the net processing time for the network is estimated by:

$$T_{\text{Network}} \approx \sum_{M=0}^{\log_2 N/2 - 1} [(T_{\text{sum/diff.}} + T_{\text{Comp.Mult.}})2^M + 60\text{ns}] \qquad (6.2)$$

for off-chip twiddle factor storage or,

$$T_{\text{Network}} \approx \sum_{M=0}^{\log_2 N/2 - 1} [(T_{\text{sum/diff.}} + 2T_{\text{Comp.Mult.}})2^M + 60\text{ns}] \qquad (6.3)$$

for on-chip twiddle factor updating. From these equations, the net processing time for a 1024-point FFT is approximately 1.2 ms for on-chip twiddle factor updating or 0.9 ms. for off-chip twiddle factor storage. The associated signal-to-noise ratio for off-chip twiddle factor storage in the 1024-point FFT was approximately 60 dB. Table 6.2 gives the predicted values for FFT performance as a function of N. Should these predicted results hold in the actual implementation, a very powerful digital signal processing structure would be realized. This structure would be capable of reading and processing a 1024-point FFT at a rate greater than 1kHz.

6.5 Power Requirements

The major advantage in designing in a CMOS as opposed to an NMOS technology is that of reduced power consumption. A circuit designed in CMOS requires much less power to operate than its NMOS counterpart. This is due to the fact that in NMOS, a current flows through the enhancement node transistor half the time, while in CMOS current flow is present only during logic level transition. Between these transitional periods, the circuits current flow is restricted to the small subthreshold leakage currents through the NMOS or PMOS transistors (see Appendix B). In the following analysis this current is considered negligible, and the assumption is made that the steady state current draw is zero. Since in most clocked systems, the logic level determinations occur during the positive going clock pulses, the circuits power requirements are a function of clock frequency.

Figure 6.2 shows the voltage and current curves for the typical operation of a full adder cell. These curves were obtained using the SPICE simulator. Evident in these figures is the fact that current flow coincides with the change in logic levels, and that during periods of steady state logic levels the current is approximately zero. The amount of energy transferred to the circuit per clock cycle, E, may be given by

$$E = \int_0^T P dt \tag{6.4}$$

where T is the period of the clock cycle, and P is the power. Assuming a constant voltage source of V volts we may rewrite equation (6.1) as

$$E = V \int_0^T I dt \tag{6.5}$$

The integral $\int_0^T I dt$ represents the area under the current-time graph over a single clock period. Since the steady-state current level is approximately zero, this area may be approximated by the area of the triangular current pulse associated with the high going clock cycle. This triangular pulse is clearly seen in figure 6.2. The simulation seen in figures 6.2(a,b) indicate a maximum peak current of 498 micro-amps is reached. Taking this as a worst case approximation, the net energy required per clock cycle, E_{cycle}, by the full adder unit, may be given by

$$E_{cycle} = 5v \times (\text{area of current/time graph}) \tag{6.6}$$

or,

$$E_{cycle} = 5v \times 498\text{micro–A} \times 0.5(2.9\text{ns}) \tag{6.7}$$

Thus,

$$E_{cycle} = 3.61 \times 10^{-12}\text{joules/clock cycle} \tag{6.8}$$

Multiplying this amount by the frequency of the fastest clock, 50 MHz, the energy consumed by a single adder unit per second is 1.80×10^{-4}joules 1.80×10^{-4} watts. Similar calculations were performed on the data latch which was found to require 1.22×10^{-4} watts to operate. In order to estimate the net processor power requirements, we consider that adder cells and data latches comprise the majority of the processing node components. The processing node is comprised of approximately 281 full adder cells and 560 data latches. The net power requirements for the processing node P_{node} is then approximated by

$$P_{node} \approx (281)P_{full\_adder} + (560)P_{data\_latch} \tag{6.9}$$

where P_{full_adder} and P_{data_latch} are the power requirements for the full adder cell and the data latch respectively. Substituting these values into equation (6.9) P_{node} is estimated to be 110 mW. A second method for approximating the processor power requirements involves determining the power requirements of a simple inverter. This was done as in the case of the full adder and the data latch, and the power requirements were found to be 2.88×10^{-5} watts per inverter. Assuming the 18000 transistors making up the node are in the form of 9000 inverters. The total power requirements of the processor in this approximation is estimated to be 259 mW. Based on these two estimates, the worst case power requirements is taken to be 259 mW. This is a very conservative estimate in that it assumes each element undergoes a logic level transition at a frequency of 50 MHz. In this implementation however only the exponential pipeline operates at 50 MHz. and all other elements operate at lower frequencies. Also a state in which all the logic levels undergo a change simultaneously for a period of 1 second is highly unlikely. Thus we may consider the power requirements estimate of 259 mW to be an overestimate of the actual case.

6.6 Scaling Considerations

At present the processing node is designed in a three-micron single-metal CMOS technology. In the event of commercial fabrication, a more current state-of-the-art process technology would most likely be used. An example of a likely fabrication technology would be a 1.5-micron double-metal CMOS process. The advantages of implementation in such a technology are several. Introduction of a second metal layer removes the necessity of using long polysilicon lines for data transferral. The lower resistance, and reduced capacitive affects of the metal would speed up signal propagation, and reduce the designs power requirements. The second level of metal would also increase the level of circuit compaction attainable in the cellular designs, by allowing three different signal lines to be stacked on top of one another.

By reducing the dimensional requirements from three microns down to 1.5 microns a scaling factor of two is introduced. The affects of such scaling are manifest in many aspects of system performance. A brief discussion on the affects of scaling with respect to the systems operating parameters is found in Appendix C. One consequence of scaling by some factor α ($\alpha > 1$), is that the gate capacitance and gate delay times are scaled by α accordingly. In this example, the gate delays and capacitance at 1.5 microns are half their corresponding values at three microns. Extending this fact, a factor of two improvement in device performance is realized. Thus for the case of a 1.5-micron implementation, the sum/difference and complex multiplications could be calculated in approximately 260 and 190 ns respectively. On a system level these results predict that a 1024-point FFT could be calculated in 0.45 ms without on-chip twiddle factor updating, or in 0.55 ms with the update. Notice that these are rough estimates, which do not take into account any potential improvements in performance due to the introduction of a second metal layer.

As discussed in Appendix C, the affects of scaling were determined using the assumption of constant field scaling [5,19,20]. In this event the electrostatic fields remained constant before and after scaling, and the total power dissipated per unit area remains constant.

6.7 Design for Testability

With the increasing levels of circuit complexity available through VLSI design of readily testable circuits is becoming an important concern. At high levels of component density design for testability is required in order to efficiently determine the existence and location of any circuit design or process induced faults. Currently the principal techniques for realizing a testable design are broken down into two categories, ad-hoc and structured designs. Ad-hoc techniques solve the testability problem for a given design, and are generally not applicable to all designs. Ad-hoc design techniques include partitioning, board-level design, and bus architectural design [21]. In contrast to the ad-hoc techniques, are the structured design approaches. These techniques are generally applicable, and involve design by a structured set of design rules. The objective of a structured testability approach is to reduce the sequential complexity and aid in test generation and verification. Examples of structured design approaches are a) the multiplexor technique, b) level sensitive scan design (LSSD), the scan/set logic approach, and the built-in logic block observation (BILBO) [21,22].

The FFT processing node described here utilizes the ad-hoc method of bus architectural design for testability within the APU. This particular design structure is intrinsic to the MAP architecture. Through the appropriate application of control signals, it is possible to isolate and test any component or set of components in the mantissa or exponential pipelines. No provisions have been made for testability design in elements outside the APU. These components are primarily intermediate memory storage devices and make up only a small portion of the processing node.

6.8 Summary

In this chapter, the overall processing node and performance time estimates were presented. It was shown that the DIF butterfly operation is performed in approximately 900 ns. The processing node performance times were then extrapolated to yield the system level performance times found in table 6.2. This table indicates that a 1024-point FFT would be calculated in approximately 0.9 ns without on-chip twiddle factor updating, or in 1.1 ns with the update. The upper level power estimates were placed at 259 mw per processor node. Also presented in this chapter were scaling considerations should commercial manufacture of this device be considered. In section 6.7 the problem of designing for testability was discussed. It was shown that the MAP structure is ideally suited for testability design in that it allows the isolation of any component or sequence of components for testing within the pipeline. In summary, the performance results reported in this chapter do indeed meet the original design specifications established in Chapter 4.

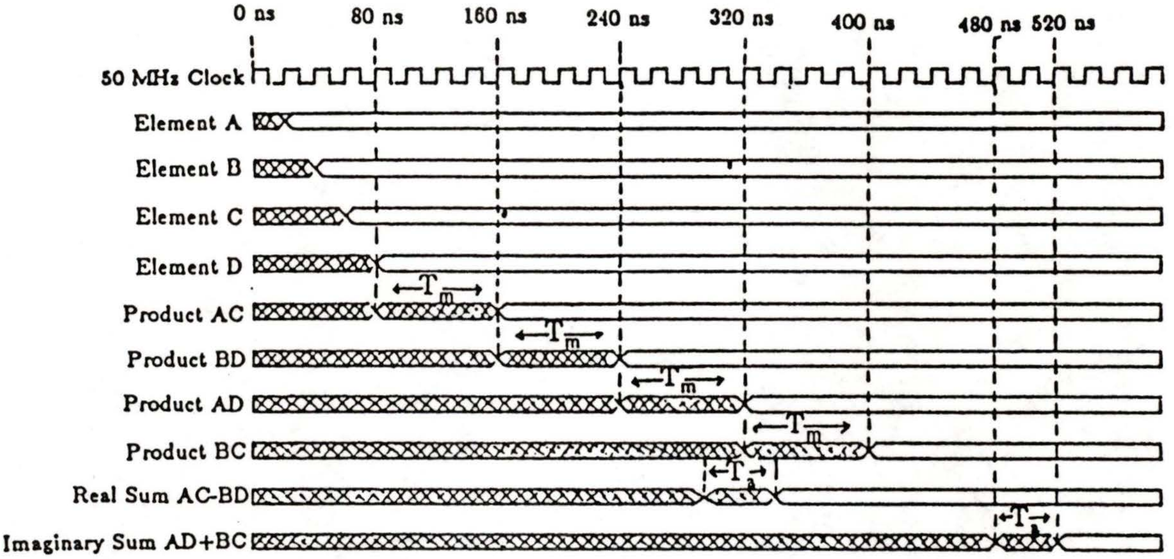
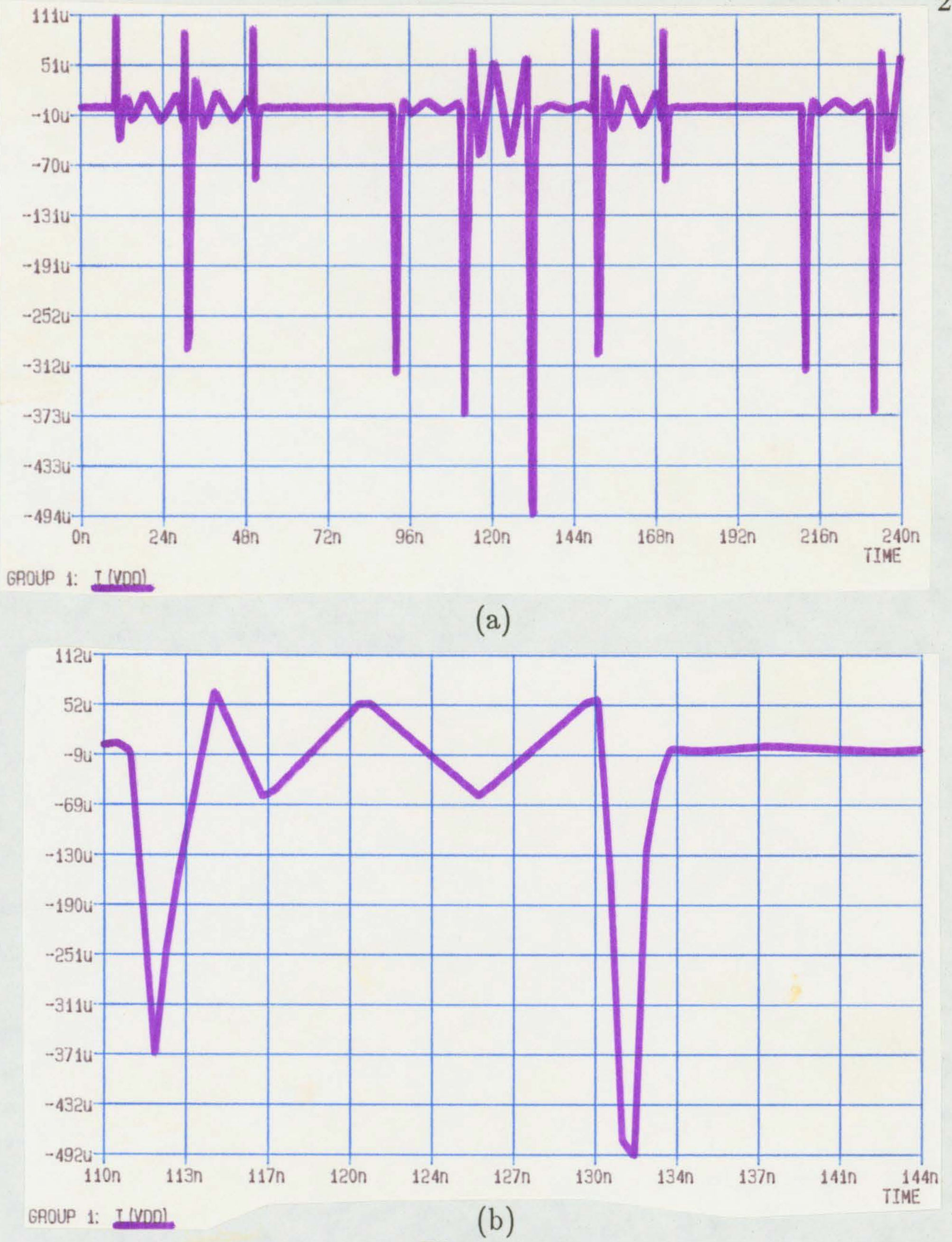


Figure 6.1

The timing diagram for the pipelined complex-number multiplication. In this diagram, the undetermined states are indicated by "hatched" lines. Only the intermediate terms of primary performance are indicated in this diagram. Operations such as two's-complementation, shifting and justification are accounted for during the periods of undetermined states.



(a)

(b)

Figure 6.2

The SPICE simulation for current flow through the full adder cell. Figure 6.3(a) indicates the current flow corresponding to the SPICE simulation seen in figure 4.10(b). The peaked periods of high current correspond to the changes in input logic levels. Figure 6.3(b) is a close-up of the largest peak. This peak was the one used in determining the full adder power requirements.

Floating Point ALU Performance Times	
Operation	Time
Real Addition	160 ns
Real Multiplication	120 ns
Real Multiply-Accumulate	240 ns
Complex Addition	240 ns
Complex Multiplication	520 ns
Complex Sum-Difference	380 ns
Butterfly Operation	900 ns

Table 6.1

Simulated APU performance time results for real and complex number floating point operations.

FFT Size (# points)	Net Processing Time Without Twiddle Factor Update (# stages)	·Net Processing Time With Twiddle Factor Update (micro-seconds)
32	29.0	40.5
64	57.9	81.0
128	115.6	161.7
256	230.9	323.0
512	461.3	645.7
1024	922.2	1280.1

Table 6.2

Estimated FFT performance times for the Pipelined Cascade implementation. Two sets of performance times are indicated, one for on-chip twiddle factor updating, and the other when the twiddle factors are read from off chip.

Chapter 7

Conclusions and Future Research Considerations

7.1 Introduction

In this thesis the design of a general-purpose FFT processing node has been presented. This node is presently configured for operation in the pipelined cascade implementation, which realizes a radix-2 decimation in frequency FFT algorithm. The processing node is designed in a three micron ISO-CMOS technology, and operates on a modified IEEE floating-point data format. It is capable of performing a complete butterfly operation in 900 ns. The design contains some 18,000 transistors and occupies an area of 6.4 mm^2. Its power requirements are estimated to be approximately 259 mW. Further estimates indicate that by implementing this processing node in the pipelined cascade implementation, a 1024-point real or complex number FFT could be computed in approximately 0.9 ms.

In Chapter 1 the thesis objectives were outlined and a brief review of the discrete Fourier transform was made. A development of the fast

Fourier transform was made for the cases of the radix-2 decimation in time and frequency algorithms. The concept of the butterfly operator was introduced, and a general flow graph for the operation of these two algorithms was presented.

In Chapter 2, a general survey was made of several proposed VLSI implementations for computing the discrete and fast Fourier transforms. The implementations reviewed consisted of the single cell DFT, the pipelined DFT network, the FFT network, the Mesh configuration, and the pipeline cascade implementation. For each implementation, an area-time^2 performance factor was determined, and the advantages and disadvantages of each were analyzed. As a result of this analysis it was decided that the pipeline cascade Implementation was most suitable for implementation, even though its area–time^2 performance factor was found to be nonoptimal. This decision was based on such factors as network expandability, ease of implementation, simplicity, and amenability to bit-parallel communication. The approximated performance time for the pipelined cascade implementation to calculate an N-point FFT was given by:

$$\mathrm{T_{total}} = \sum_{\mathrm{M=0}}^{\log_2\mathrm{N}-1} 2^{\mathrm{M}} \times (\mathrm{T_{sum/difference}} + 2\mathrm{T_{comp.\ Mult.}}) \tag{7.1}$$

In Chapter 3 several fixed and floating-point data formats were presented and the advantages and disadvantages of each were discussed. A study was then made on the affects of truncation and roundoff for both the fixed and floating-point formats. A detailed analysis on the affects of round-off error and finite register lengths in the FFT calculations was also made. This analysis considered both the fixed and floating-point data formats, and determined the ratio of output signal-to-noise variance for both cases. In the case of the floating format the input signal to output noise ratio was also determined, and graphed as a function of FFT size and register length. Based on the results of these analysis the choice was made to use a floating-point format. In order to conform with some prescribed standard the IEEE 32-bit floating-point format was chosen. From the results of the signal-to-noise ratio graph it was found that output levels in excess of 60 dB were obtainable for a 1024-point FFT using a 14-bit mantissa length. Since 60 dB is considered an acceptable level for this implementation the decision was made to truncate the IEEE standard mantissa length from 23 to 14 bits. Notice that for different design specifications the required output signal-to-noise ratios may be lower than those specified for this implementation. As a result the mantissa length may be further truncated until this specification is reached. This reduction in mantissa length serves to both speed up the processing times and reduces the design area requirements.

Chapter 4 dealt with the actual design of the FFT processing node and describes in detail the arithmetic processing unit, or APU. In this chapter a VLSI design methodology is presented and the design specifications for the processing node were determined. The physical design of the processor was described in a hierarchal fashion starting at the lowest level of design. For each component design pertinent information regarding design size, operation, and performance times were given. SPICE and Hg simulations on the designs were also performed and discussed in this chapter. In section 4.7.2 the multiple access pipeline or MAP structure was presented. The MAP structure was used to implement the processor node APU and has proven to be a very powerful processing architecture. This structure combines bus and pipeline architectures to realize a highly flexible concurrent processing system. Two such structures were utilized in the APU to operate on the mantissa and the exponential elements of the data. The APU itself is a fast general-purpose floating-point arithmetic processor. In the context of this design its operation is limited to performing complex-number multiplications and sum/difference calculations. The APU is microprogram controlled and due to the flexibility of the MAP structure it is possible to isolate any individual components or series of components of the pipeline for operation. The MAP structure also allows recursive data flow to occur making this APU ideal for many other digital signal processing applications. Table 6.1 outlines the performance times for some of the arithmetic operations performed by the APU.

In Chapter 5 the design of the control logic was described. This implementation utilized a master slave control configuration with the APU controller acting as slave to the node level master controller. Control requirements for both node level and system level operations were specified in detail. The APU operates on a microprogrammed controller and a specially-encoded microinstruction was developed to ensure that individual bus access control was made concurrent with the pipeline control operations. Two microprograms for performing the complex-number multiplications and the sum/difference calculations were then presented followed by a discussion of the master controller operation. These programs are found in figures 5.4 and 5.5.

Chapter 6 discussed the design and simulation aids used in the course of this work. The simulated APU performance times were presented and estimates were made regarding the pipeline cascade implementation performance. These estimates were performed for the cases of on-chip twiddle factor updating and off-chip twiddle factor storage. Based on SPICE simulations, calculations were performed estimating the power requirements of the chip to be 259mW. Scaling effects and design for testability were also discussed in this chapter.

Assuming the processing estimates are correct, a 1024-point FFT would be processed in 1.2 ms with on-chip twiddle factor updating and in 0.9 ms without the update. Such a system would see a wide range of application in the field of digital signal processing. Its high performance would allow a 1024-point FFT to be taken and processed well within the 60 Hz. refresh cycle of most video terminals. In this context real time FFT calculation may be achieved. Direct application would be found in such area's as speech analysis, seismic exploration, radar, sonar and image processing.

The network described is capable of handling real or complex-number FFT's. In the event that pure real numbers are to be transformed, processing of two sequences might be carried out simultaneously. Assuming the two sequences to be of equal length, N this is accomplished by encoding one sequence as the complex components of the other. The FFT is then performed on a single sequence of "complex" numbers. Using the property of linearity in the FFT, it is possible to extract the two real transformed sequences from the complex transformed output [1]. If we designate $X_c(k)$ to be the combined transform sequence, then the two real transformed sequences $X_1(k)$ and $X_2(k)$ may be extracted as follows:

$$X_1(k) = \frac{X_c(k) + X_c^*(N-k)}{2} \tag{7.2}$$

$$X_2(k) = \frac{X_c(k) - X_c^*(N-k)}{2i} \tag{7.3}$$

where $X_c^*(N-k)$ is the complex conjugate of the $(N-1)^{th}$ term of the transformed output. For a proof of this result the reader is referred to the above mentioned references. Through utilization of this coding scheme it is possible to perform two real FFT's in a slightly longer time than it takes to complete a single complex FFT. The extra processing time results from calculating equations (7.2) and (7.3). In practice these calculations would likely be performed using software programs, however a specially configured processing node could also perform thes task.

7.2 Future Design Considerations

The processor node design described in this thesis has by no means reached an optimal state of performance. Through the course of the design several areas in which improvement could be made became apparent. In the processing node these improvements are primarily concerned with increasing performance speeds and expanding flexibility. At the system level, future research directions include expanding the system to perform inverse and multidimensional FFTs. The following sections detail those areas which in the authors opinion merit future research considerations should this design be taken to its fullest potential.

7.2.1 The Parallel Multiplier

As mentioned in section 4.4.7 the parallel multiplier serves as the bottleneck for mantissa pipeline performance. Any improvements made on the performance time of the multiplier would be manifest in the system and processor performance. One possibility open for investigation is to attempt to pipeline the parallel multiplier.

In the parallel multiplier implementation each adder cell calculates an intermediate product term, and passes it diagonally down the line of fixed binary weights. Once this has been done, the row of adder cells remains idle until the beginning of the next multiplication. In a pipelined multiplier, each row could become an intermediate processing stage of a pipeline. Thus having determined the intermediate product, each row would immediately begin calculating the intermediate product for the next multiplication. This procedure converts the idle time of the parallel multiplier into active processing time. At present, no calculations have been made to estimate either the performance time or size requirements of such an implementation.

7.2.2 Dynamic Logic Implementation

Performance times of some of the APU processing elements for example multipliers or adders, might be improved upon by implementing them in a dynamic logic. Two such possibilities are the NORA and DOMINO logics [18,19]. In these logics two non-overlapping clocks are used. The lines are precharged during the first high clock cycle and the logic evaluated during the second. These structures have the advantage that the bulk of the logic elements are implemented with the faster NMOS transistors. The typically large power requirements associated with NMOS circuits are avoided by the use of the precharge phase [22,23]. Whether such an implementation will prove faster than the CMOS circuits currently used and if so at what cost in terms of power has yet to be determined.

7.2.3 Improved Microprogram Control

The APU described in this thesis has the potential to become a very powerful general-purpose processing tool. Future research should center on developing this potential. The microprogrammed control structure of the APU is one are where improvements could be made to increase the processors flexibility. The difficulties involved in designing the microprogrammed controller were discussed in detail in Chapter 5. Future research in this area should consider the development of a more efficient coding scheme for the microinstruction format. At present the code format is designed so that the sum/difference and complex multiplication microprograms might be realized in as few steps as possible. This format is not necessarily the optimal one for a general-purpose microprogram control. The microprogram might be made dynamic so that programs could be entered from off-chip sources and additional microprograms written to perform convolution or simple recursive digital filters using the APU. In addition to this work, provisions could be made so that the processor might perform either floating- or fixed-point operations on demand. This would require being able to isolate the exponential pipeline, and bypassing the mantissa shifter unit. Provisions for overflow and underflow would also have to be made in this event.

7.2.4 Inverse FFTs

The pipelined cascade implementation also has the potential to perform inverse FFTs without any changes in its basic algorithm. The inverse DFT of an N-point sequence is given by

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\omega^{-nk} \tag{7.4}$$

By taking the complex conjugate of equation (7.4) and multiplying both sides by N, we get

$$Nx^{*}(n) = \sum_{k=0}^{N-1} X^{*}(k)\omega^{nk} \tag{7.5}$$

We recognize that the right side of equation 7.5 is simply the DFT of the input sequence $X^{*}(k)$. The desired output x(n) can be obtained by computing the complex conjugate of the DFT in equation 7.5 and dividing by N. That is

$$x(n) = \frac{1}{N}\left[\sum_{k=0}^{N-1} X^{*}(k)\omega^{nk}\right]^{*} \tag{7.6}$$

In order to utilize the pipeline cascade implementation to perform this calculation, provisions must be made to calculate the complex conjugate of both the input and output sequences. Each output element must also be multiplied by 1/N. These operations might be performed using software applications, or by implementing specially-programmed processing nodes at the pipelines input and output. Once again this stems back to increasing the microprogammed flexibility of the APU controller unit.

7.2.5 Windowing Functions

The pipeline cascade implementation is presently configured using a rectangular windowing function. By employing an extra processing node at the beginning of the pipeline, different windows such as the Hamming or the Kaiser windows might be employed. This window processing node would contain a slightly modified node control unit. It would be responsible for premultiplying the input data by the appropriate value of the windowing function. Depending on the extent of the control modification, the window factors may be stored in off-chip memory or they may be continually updated on-chip using an appropriate microprogram. The off-chip memory storage is the simplest means of achieving different windowing functions in that the only control change is that required to read the data from off-chip memory. Also the window function values may be predetermined thus avoiding any update delays encountered with the second method.

7.2.6 Multidimensional FFTs

Future research might consider the potential for calculation of multidimensional FFTs using the implementation described in this thesis. In general this poses a much more sophisticated problem than for the one dimensional case. As an example let us consider the case of a two-dimensional DFT. The two-dimensional DFT may be defined as:

$$X(k_1,k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2}^{N_2-1} x(n_1,n_2) e^{-j(2\pi/N_1)n_1k_1} e^{-j(2\pi/N_2)n_2k_2} \tag{7.7}$$

where N_1 and N_2 are the number of samples along the first and second dimensions respectively. We may rewrite equation (7.7) as follows:

$$X(k_1,k_2) = \sum_{n_1=0}^{N_1-1} \omega_1^{n_1k_1} [\sum_{n_2=0}^{N_2-1} x(n_1,n_2) \omega_2^{n_2k_2}] \tag{7.8}$$

where $\omega_1 = e^{-j2\pi/N_1}$ and $\omega_2 = e^{-j2\pi/N_2}$. We recognize that the bracketed term of equation 7.7 is simply the one-dimensional DFT of $x(n_1,n_2)$. Thus in order to calculate $X(k_1,k_2)$ a total of N_1 N_2-point FFTs and N_2 N_1-point FFTs need to be calculated. For large N_1 and N_2 this is a large amount of calculation. For example if $N_1 = N_2 = 1024$ then the number of complex-number multiplications needed to be performed is approximately 2×10^9. If both N_1 and N_2 are powers of two, FFT algorithms may be employed. In this event approximately 10^7 complex multiplications are needed.

In order to calculate the two-dimensional FFT it is convenient to consider an $N_1 \times N_2$ memory array. We begin by computing the bracketed term in equation 7.8. This term represents a series of N_1 N_2-point one-dimensional FFTs obtained by varying n_1 0 N_1-1. Each transform for constant a n_1 consists of N_2 points, and represents the row vectors of the memory matrix. Having completed all N_1 of the N_2-point FFTs, the remainder of equation 7.7 is calculated. In this calculation N_2 N_1-point one-dimensional FFTs are determined. In this case the N_1 column vectors of the memory array serve as the input vectors to the FFT. Notice that each column vector is made up of single transform elements from each of the individual row transforms. After calculation the N_1-point transforms are restored in the columns from which the input vector was sourced. Upon completion of the N_2 N_1-point transforms, the matrix contains the final transformed values.

Obviously this is a much more involved task than for the one-dimensional case. Research considerations in this area should center on the possibility of networking entire pipeline cascade implementations together to realize a larger processing system. Notice however that the basic processing components of such a system could be the general-purpose FFT processing node described in this thesis. With an appropriately designed off-chip controller and sufficient memory allocation, it should be possible to realize a two-dimensional FFT using a single pipelined cascade Implementation. In order to do this we must assume that $N_1 = N_2$. As an example consider the previous case in which $N_1 = N_2 = 1024$. If we assume that each 1024-point FFT takes approximately 1 ms to calculate, then a very gross system approximation suggests that the two-dimensional FFT might be calculated in the order of 2 s. This value comes from the fact that we need to perform 1024 1024-point FFTs followed by another 1024 1024-point FFTs. In general the same procedure outlined above may be used to calculate FFTs of higher dimensions as well.

7.3 Summary

This thesis has presented the design of a general-purpose FFT processing node for implementation in the pipelined cascade network. Estimates indicate that this network is capable of sub millisecond calculation of a 1024-point FFT. Utilizing the capability of on-chip twiddle factor updating, the FFT is calculated in 1.2 ms. Thus for some applications, real time video display of the FFT results is possible. The speed and compact size of the network make it ideal for use in many task-specific problems requiring fast spectrum analysis. Application of such a system would be found in such areas as radar and sonar, speech recognition, and image processing both in a medical and nonmedical environment. In conclusion, the general-purpose FFT processor's speed and flexibility combine with the pipelined cascade implementations compact size, and flexible nature, to create a powerful digital signal processing tool.

REFERENCES

[1] L. R. Rabiner, and B. Gold, "Theory and Application of Digital Signal Processing", Englewood Cliffs, N.J., Prentice-Hall, Inc., 1975.

[2] A. Antoniou, "Digital Filters: Analysis and Design", New York, N.Y., McGraw-Hill, Inc., 1979.

[3] J. W. Cooley, and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", *Math. Comp.,* vol. 19, pp.297-301, April 1965.

[4] B. McKinney, "A Pipelined Complex-Number Multiplier for VLSI Implementation", Technical Digest, 1985 Conference on Very Large Scale Integration, pp. 34-37, November 1985.

[5] C. Mead, and L. Conway, "Introduction to VLSI Systems", Reading, Mass., Addison-Wesley, Inc., 1980.

[6] M. Morris Mano, "Digital Logic and Computer Design", Englewood Cliffs, N.J., Prentice-Hall, Inc., 1979.

[7] C. D. Thompson, "Fourier Transforms in VLSI", *IEEE Transactions on Computers*, vol. C-32, pp. 1047-1057, November 1983.

[8] A. M. Despain, "Very Fast Fourier Transform Algorithms Hardware for Implementation", *IEEE Transactions on Computers*, vol. C-28, pp. 333-341, May 1979.

[9] E. H. Wold, and A. M. Despain, "Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations", *IEEE Transactions on Computers*, vol. C-33, pp. 414-425, May 1984.

[10] A. V. Oppenheim, and C. J. Weinstein, "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform", *Proceedings IEEE*, vol. 60, pp. 957-976, August 1972.

[11] D. J. Kuck, D. S. Parker, and A. H. Sameh, "Analysis of Rounding Methods in Floating-Point Arithmetic", *IEEE Transactions on Computers*, vol. C-26, pp.643-650, July 1977.

[12] C. J. Weinstein, "Roundoff Noise in Floating Point Fast Fourier Transform Computation", *IEEE Transactions on Audio and Electroacoustics*, vol. AU-17, pp. 209-215, September 1969.

[13] N. A. Alexandridis, "Microprocessor System Design Concepts", Rockville, Maryland, Computer Science Press, Inc., 1984.

[14] B. Liu, and T. Kaneko, "Error Analysis of Digital Filters Realized with Floating-Point Arithmetic", *Proceedings of IEEE*, vol. 57, pp. 1735-1747, October 1969.

[15] G. Csanyi-Fritz, (verbal communications), September 1985.

[16] W. M. vanCleemput, "Hierarchical Design for VLSI: Problems and Advantages", Caltech Conference on VLSI, pp. 259-272, January 1979.

[17] C. P. Lerouge, P. Girard, and J. S. Colardelle, "A Fast 16-Bit NMOS Parallel Multiplier", *IEEE Journal of Solid-State Circuits*, vol. SC-19, pp.338-342, June 1984.

[18] B. McKinney and F. El Guibaly, "A Fourier Transform Algorithm and VLSI Implementation", Proceedings, IEEE Electronicom '85, October 1985.

[19] J. Mavor, M. A. Jack, and P. B. Denyer, "Introduction to MOS LSI Design", Reading, Mass., Addison-Westly, Inc., 1983.

[20] A. Reisman, "Device, Circuit, and Technology Scaling to Micron and Submicron Dimensions", *Proceedings IEEE*, vol. 71, pp. 550-565, May 1983.

[21] T. W. Williams and K. Parker, "Design for Testability-A Survey", *IEEE Transactions on Computers*, vol. C-31, pp. 2-15, January 1982.

[22] R. H. Krambeck, C. M. Lee, and H. S. Law, "High-Speed Compact Circuits with CMOS", *IEEE Journal of Solid State Circuits*, vol. SC-17, pp.614-619, June 1982.

[23] N. F. Goncalves, and H. J. DeMan, "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures", *IEEE Journal of Solid State Circuits*, vol. SC-18, pp. 261-266, June 1983.

[24] M. Elmasry, "Digital MOS Integrated Circuits: A Tutorial", IEEE Press, Section 1.3, 1981.

Appendix A

The MOS Transistor

The MOS transistor is a member of the field-effect transistor (FET) family. The transistor is formed from a layered conductor-insulator-semiconductor structure. A cross section of a typical MOSFET transistor is shown in figure A.1. The various terms and dimensions in this diagram are defined as they appear in the text.

The top plate of the MOS transistor is referred to as the gate. Typically the gate is made up of polycrystalline silicon. The insulating material is sandwiched between the substrate and the gate, and has a thickness t. This material is an intrinsic dielectric made up SiO_2, which is grown directly on the silicon surface. The substrate semiconductor for this analysis is silicon. The two diffused implant regions seen in figure A.1 serve as the device output terminals, and are referred to as the drain and source. Because of device symmetry, the drain and source are interchangeable. The surface region between the source and drain is known as the channel. Conduction through this channel is controlled by an applied voltage to the gate.

The semiconductor body may be either p-type or n-type corresponding to the p-channel or n-channel transistors respectively. The difference lies in the fact that the current carriers for the NMOS transistors are electrons, while those for the PMOS transistors are holes.

Considering an NMOS transistor, in the absence of any applied voltage to the gate, the transistor channel is like an open switch. Application of a positive gate potential induces an electrostatically induced negative charge on the semiconductor, repelling holes from the channel region. This tends to reduce the surface charge density below that of the bulk semiconductor, causing a depletion region to be formed at the surface. As the channel holes are being repelled from the gate, electrons from the n^+ source and drain regions move into the channel under the attraction of the positive gate. This process continues until at some gate potential, the electron density within the channel exceeds that of the hole density, and a conductive path from the source to drain is formed. At this point, the channel current, I_{DS}, rises above the off state leakage current. The voltage at which this occurs is known as the threshold voltage V_T. In general, all voltage sources are measured relative to the source voltage. Therefore when $V_{gate}-V_{source}=V_{GS} > V_T$, a conductive path is formed between the source and drain. In the case of a PMOS transistor, the exact opposite of the above argument applies. For the PMOS transistor when $V_{GS} > V_T$ the conductive source to drain path is open [19,24]. That is the PMOS transistor is normally on with zero potential on the gate.

In modelling the MOSFET transistor, three regions of operation are identified [24].

1. The off, or subthreshold region,

$$V_{GS}<V_T \tag{A.1}$$

$$I_{DS}\approx 0 \tag{A.2}$$

2. The nonsaturation or linear region, where

$$V_{DS}<V_{GS}-V_T \tag{A.3}$$

$$I_{DS}=\beta[(V_{GS}-V_T)V_{DS}-\frac{1}{2}V^2{}_{DS}] \tag{A.4}$$

3. The saturation region where,

$$V_{DS}\geq V_{GS}-V_T \tag{A.5}$$

$$I_{DS}=\frac{\beta}{2}[V_{GS}-V_T]^2 \tag{A.6}$$

The variable β of the previous equations is defined as

$$\beta = \frac{W}{L} \frac{\epsilon_{ox}\mu}{t_{ox.}} \tag{A.7}$$

The variables W, L, and $t_{ox.}$ refer to the transistor width, length, and oxide thickness respectively. The term $\epsilon_{ox.}$ is the permittivity of the gate oxide, and μ is the average surface mobility of the charge carriers. For the case of electrons in NMOS, $\mu=\mu_n$, while for holes in PMOS, $\mu=\mu_p$. The drain currents for the PMOS or NMOS transistors are obtained by substituting the appropriate mobility factors into the β' s of equations (A.4) and (A.6). Notice that the drain current I_{DS} is directly proportional to the width:length ratio of the transistor. Therefore for a given set of operating voltages, at constant L, increasing W increases the drain current. Increasing W however also increases the area of the source and drain diffusion regions. This results in an increase in the source-substrate and drain-substrate junction capacitances. Thus a capacitance-current trade-off exists, which must be optimized in accordance with the design requirements.

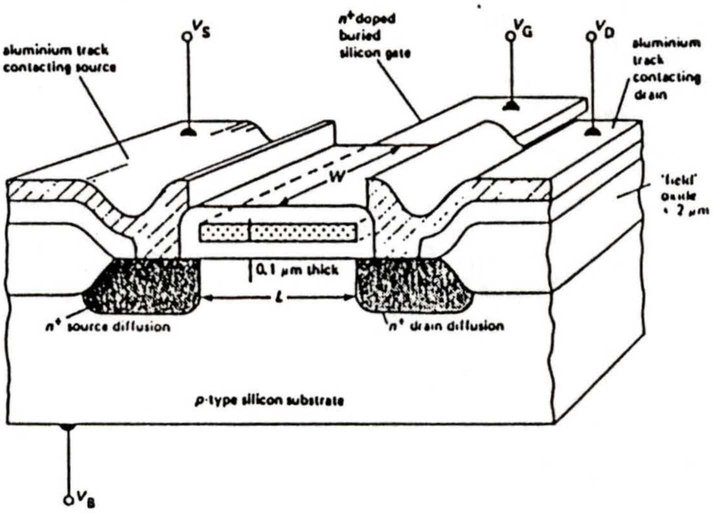


Figure A.1

The cross section of the MOS transistor.

Appendix B

The CMOS Inverter

The basic CMOS inverter consists of a PMOS transistor and an NMOS transistor connected in series across a common voltage source VDD. The same input signal is applied to the gates of both transistors, and the output signal is obtained from their common drains. Figure B.2 illustrates the CMOS inverter cross section. If we assume the subthreshold leakage currents through the PMOS and NMOS transistors to be negligible, the logic levels of the CMOS inverter are close to VDD and ground. In the event that the leakage currents are not considered negligible, the β values of the PMOS and NMOS transistors is increased until the assumption is valid [19,24].

The switching threshold voltage, $V_{inv.}$ of the inverter is defined as the point at which the input voltage V_{in} equals the output voltage V_{out}. In perfectly symmetrical inverter, $V_{inv.}=0.5$ VDD. The switching threshold voltage is found by equating the two drain currents of the saturated devices. Equation (A.6) yields

$$\frac{1}{2}\beta_{NMOS}(V_{inv.}-V_{T_{NMOS}})^2=\frac{-1}{2}\beta_{PMOS}(V_{inv.}-VDD-V_{T_{PMOS}})^2 \qquad (B.1)$$

Solving for $V_{inv.}$ we find

$$V_{inv.}=\frac{\beta^{1/2}{}_{PMOS}(VDD+V_{T_{PMOS}})+\beta^{1/2}V_{T_{NMOS}}}{\beta^{1/2}{}_{PMOS}+\beta^{1/2}{}_{NMOS}} \tag{B.2}$$

If we take the switching threshold voltage $V_{inv.}$ to be $V_{inv.}=0.5VDD$ then as the input voltage switches between logic levels, one MOS transistor will switch on while the other switches off. Thus in a steady-state there is no direct current path to ground since one transistor is always off. Current only flows to ground when both transistors are in a transitional state, during the switching of the gate potential. Thus the net power required is a function of the switching frequency of the device. A more detailed power analysis is found in Chapter 6.

The condition of $V_{inv.}=0.5VDD$ for both the PMOS and NMOS enhancement transistors occurs if we assume that $V_{T_{NMOS}}=|V_{T_{PMOS}}|$, and if symmetrical devices are used. Symmetrical devices are such that their respective β' s are equal. Thus for $\beta_{NMOS}=\beta_{PMOS}$, it is required that:

$$\frac{(\frac{W}{L})_{PMOS}}{(\frac{W}{L})_{NMOS}}=\frac{\mu_{NMOS}}{\mu_{PMOS}}\approx 2.5 \tag{B.1}$$

In order to equalize the switching speeds of the PMOS and NMOS transistors and to minimize the power requirements, the W/L ratio of the PMOS transistor is usually increased to 2.5:1.

We now consider a CMOS inverter with an ideal input step voltage from 0 to VDD. The discharge time, or fall time t_f, of the inverter is defined as the time required for the output capacitance C_{out}, to discharge through the NMOS transistor from an output level of $V_1 \approx VDD$ to one of $V_0 \approx 0$. The fall time of the inverter can be shown to be [24]:

$$t_f = \tau_{NMOS}\left[\frac{2}{\frac{VDD}{V_{T_{NMOS}}}-1} + \ln\left(\frac{2(VDD - V_{T_{NMOS}})}{V_0} - 1\right)\right] \tag{B.2}$$

where,

$$\tau_{NMOS} = \frac{C_{out}}{\beta_{NMOS}(VDD - V_{T_{NMOS}})} \tag{B.3}$$

Similarly, the charge time or rise time of the inverter t_r is the time taken to charge C_{out} through the PMOS transistor from $V_0 \approx 0$ to $V_1 \approx VDD$. Because of the symmetry with the case of the fall time through the NMOS enhancement transistor, the rise time is given by:

$$t_r = \tau_{PMOS}\left[\frac{2}{\frac{VDD}{|V_{T_{PMOS}}|}-1} + \ln\left(\frac{2(VDD - |V_{T_{PMOS}}|)}{VDD - V_1} - 1\right)\right] \tag{B.4}$$

where,

$$\tau_{PMOS}=\frac{C_{out}}{\beta_{PMOS}(VDD-V_{T_{PMOS}})} \tag{B.5}$$

If as was previously suggested, the threshold voltages are set equal to 0.5 VDD, and the devices are designed symmetrically, $\beta_{NMOS}=\beta_{PMOS}$, then the rise time and the fall time of the inverter are the same.

Another important parameter is the inverter delay $T_{inv.}$. This is the delay time between the input and the output waveforms measured at the VDD/2 points. This term is approximated by [24]:

$$T_{inv.}\approx\frac{0.9C_{out}}{VDD\beta_{NMOS}}[1+\frac{\beta_{NMOS}}{\beta_{PMOS}}] \tag{B.6}$$

Analogous to the inverter delay is the gate delay. Several definitions of gate delay currently exist. In this thesis the gate delay is defined as the time required for the output waveform to rise/fall to 0.9/0.1 VDD after the input waveform has fallen/risen below/above 0.9/0.1 VDD. SPICE simulations presented in Chapter 4 indicate a gate delay of 1.2 ns is achieved.

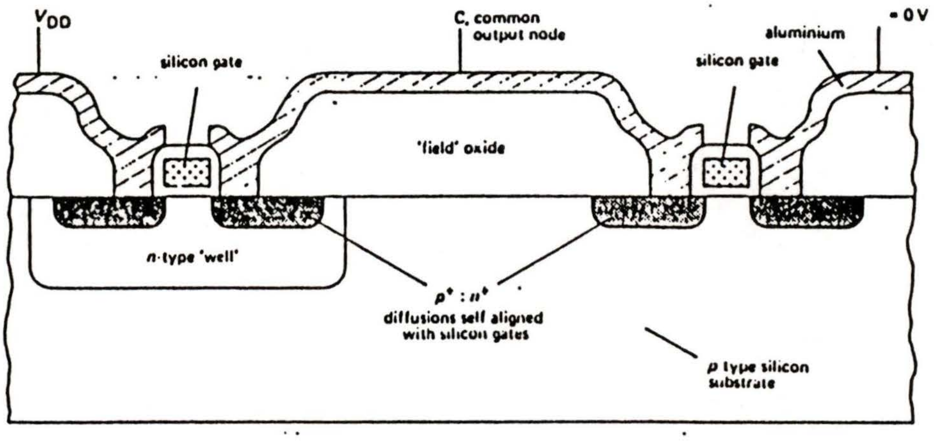


Figure B.1

A cross sectional view of a typical CMOS inverter.

Appendix C

Scaling Considerations

The most significant contribution to improvements in device complexity has been the scaling of minimum process feature sizes. The scaling considerations outlined in this appendix are based on the relationships developed by Dennard [20]. For simplicity we assume all horizontal and vertical dimensions to be reduced by a factor of s, $(s>1)$. In order to avoid excessively high electric fields, we assume all circuit voltages to be scaled by s as well. This type of scaling is referred to as constant field scaling. A second form of scaling is that of constant potential scaling [21].

Table C.1 illustrates the effects of constant field and constant potential scaling on device performance. From this table we see that the net power-delay product of the constant field scaling is superior to that of the constant potential. It is important to note that current density increases with scaling for both cases. In the constant potential case however this increase is a factor of s greater than for the case of constant field. In terms of power dissipation, the power dissipated per gate is reduced by a factor of s^2 for constant field and increased by a factor of s for constant potential. In terms of total chip power density, the s^2 reduction for constant field is canceled by the s^2 increase in packing density. The net chip power density thus remains constant. For the case of constant field scaling however, a chip power density increase of s^3 is realized. This is a highly undesirable response. For most cases therefore, constant field scaling appears to offer the greatest overall advantage.

Unfortunately, scaling by s also introduces some detrimental effects. Notable among these is the increase in current density. At high current densities, a phenomena known as electro-migration occurs. Electromigration is a physical process wherebye a current density exceeding a certain limit in a metal conductor causes metal atoms to move in the direction of the current. This further increases the current density even more, and thus an increasing number of atoms are removed. This process can eventually result in open circuits. Other effects of scaling include an increase sensitivity to "soft errors"[5,19]. Soft errors occur as a result of high energy alpha particles entering the substrate. Energy given up by the particle results in the generation of electron-hole pairs. Any minority carriers generated within the depletion region of the transistor are attracted to the transistor terminals. In the case of highly scaled capacitances, these minority carriers can cause the a premature discharging of the terminals, resulting in lost or corrupted information.

Dynamic logic and MOS memory cells are particularly sensitive to soft errors. Other second order effects which prove detrimental to circuit performance as a result of scaling include source-drain punch through, hot electron trapping, and gate oxide and junction breakdown. In summary, while advantages are realized in device performance as a result of scaling, attention must be given to its limitations and ramifications.

Effects of Scaling		
Scaled Parameters	Constant Field	Constant Potential
Length	s^{-1}	s^{-1}
Width	s^{-1}	s^{-1}
Depth	s^{-1}	s^{-1}
Substrate Doping	s	s
Supply Voltage (VDD)	s^{-1}	1
Affected Parameters		
Device Current	s^{-1}	s
Gate & Millar Capacitances	s^{-1}	s^{-1}
Gate Delay	s^{-1}	s^{-2}
Line Sheet Resistance	s	s
Current Density	s	s^{3}
RC Delay	1	1
Breakdown Voltage	$s^{0.5}$	$s^{0.5}$
Power	s^{-2}	s
Power-Delay	s^{-3}	s^{-1}

Table C.1

The first order scaling properties of the MOS transistor.

VITA

Surname: MCKINNEY Given Names: BRIAN CLIFFORD

Place of Birth: REDWOOD CITY, CALIF., USA.

Date of Birth: March 27, 1961

Educational Institutions Attended, with Dates of Entering and Leaving:

UNIVERSITY OF VICTORIA 1979 to 1986

to

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc.	1984	University of Victoria, Victoria

Honors and Awards:

B.C. Provincial Scholarship, 1979

Presidents Special Entrance Scholarship, 1979

Publications:

B. McKinney and G. McBean, "A Geostrophic Wind Analysis of Mesoscale Systems". To be published in Boundary Layer Meteorology.

B. McKinney and F. El Guilbaly, "A Fourier Transform Algorithm and VLSI Implementation", Proceedings, IEEE Electronicom '85, October 1985.

B. McKinney, "A Pipelined Complex Number Multiplier for VLSI Implementation" Technical Digest, 1985 Conference on Very Large Scale Integration, November 1985.

PARTIAL COPYRIGHT LICENSE

Title of Thesis

THE VLSI DESIGN OF A GENERAL

PURPOSE FFT PROCESSING NODE

Author [redacted]

Signature

Brian Clifford McKinney

Name

January 17, 1986

Date