THE UNIVERSITY OF CALGARY

THE DESIGN OF A HARDWARE PROCESSOR CAPABLE OF PERFORMING AUTO-REGRESSIVE MODELING IN REAL TIME

by

STACY WILLIAM NICHOLS

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

CALGARY, ALBERTA

October, 1986

ISBN 0-315-36011-9

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled *The Design of a Hardware Processor Capable of Performing Auto-Regressive Modeling in Real Time,* submitted by Stacy William Nichols in partial fulfillment of the requirements for the degree of Master of Science

Dr. M. R. Smith, Chairman
Department of Electrical Engineering

Dr. R. A. Stein
Department of Electrical Engineering

Dr. L. E. Turner
Department of Electrical Engineering

Dr. R. B. Hicks
Department of Physics

Date: 3 November 86

ABSTRACT

The design and implementation of a high speed processor dedicated to auto-regressive (AR) modeling is presented in this thesis. Several AR algorithms capable of operating in a real time environment are examined; with the Burg algorithm being chosen as it offers a good trade-off between speed and resolution. Errors arising in the block floating point implementation of the algorithm are discussed and methods of reducing these errors are presented and implemented. An algorithm that quickly and accurately performs a division operation is introduced and included in the implementation of the Burg algorithm.

The hardware architecture is heavily pipelined and consists of bit-slice microprogrammable chips that can be programmed independently. This permits full utilization of the resources by using parallel programming techniques. A high speed complex number processor composed of two ALUs, a multiplier, two memory units and a number of components associated with the above units is found to be the best trade-off between hardware complexity and speed. Using wirewrap techniques, a prototype AR processor was developed and tested.

Results indicate the accuracy of the overall implementation is comparable to that of floating point implementations. The hardware implementation is capable of performing a 16^{th} order AR model of 128 complex data points in 5.4 ms. The effective sampling rate is 23 kHz; real time operation for most applications.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. M. R. Smith, for his valuable input and guidance which helped make this thesis possible, and Dr. R. A. Stein, whose timely suggestions are gratefully acknowledged.

The financial assistance given by the University of Calgary (Graduate Assistantships), the Natural Science and Engineering Research Council of Canada (operating funds), the Province of Alberta (scholarship), the Robert B. Paugh Memorial Bursary Fund (bursary), and Advanced Micro Devices Ltd. (donation of significant hardware components) is deeply appreciated.

To my Mother and Father
For their patience and support

Table of Contents

List of Tables

List of Figures

List of Symbols

a_i	AR model coefficients / Forward PEF coefficients
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
APU	Arithmetic Processing unit
AR	Auto-regressive
ARMA	Auto-regressive Moving Average
B	Denominator
b	Backward Prediction Error
b_i	MA Model Coefficients / Backward PEF Coefficients
CC	Condition Code
CS	Control Store
D	Denominator
Δ	The Weight of the Least Significant Bit
DFT	Discrete Fourier Transform
$dshift$	Shift Count in Denominator
e	Forward Prediction Error
ε	Round-off Error
FFT	Fast Fourier Transform
FLP	Floating Point Format
FXP	Fixed Point Format
Hz	Frequency
IM	Imaginary Component of a Complex Number
I/O	Input/Output
LSB	Least Significant Bit
MA	Moving Average
MMSE	Minimum Mean Square Error
MSE	Mean Square Error
μ	Mean Rounding Error
N	The Number of Sample Data Points
$nshift$	Shift Count in Numerator
p	Final Order of the AR Model / Prescaling Error
P	Sum of Squares of Forward and Backward Errors
PEF	Prediction Error Filter
r	Number of Bits
RAM	Random Access Memory
RE	Real Component of a Complex Number
R(k)	Autocorrelation Sequence of a Signal
s	Binary Bit / Seconds
$s(n)$	An Input Signal
T	A Unit Time Delay
t,u,v,w,c,d	Intermediate Values

u(k)	An Gaussian White Process
v(n)	A Signal Composed of Gaussian White Noise
σ^2	Variance in the Round-off Error
x(k)	Data Signal / Intermediate Signal
Y	Accumulation Variable
y(k)	Output Signal

CHAPTER 1
INTRODUCTION

1.0 INTRODUCTION

Signal analysis is important to many different fields of science and can be performed in a number of different ways. Autoregressive (AR) modeling is an approach to signal analysis that has received a lot of attention. It is a parametric technique that attempts to predict the present value of a signal from weighted past values of that signal. AR modeling is used in a number of research areas [1,2,3] to obtain accurate models of the processes under examination.

This modeling approach has been applied to the analysis of human speech [1] where it has been used for word recognition and speech synthesis. These applications require high speed microprocessors to sample and store the voice data. It would be desirable if a high speed AR processor was developed to perform the analysis at the same rate the data is received.

Suppression and classification of radar clutter caused by the echos generated from the earth, weather phenomena and birds is another application of AR modeling. Research indicates that these echos can be accurately modeled as low order complex AR processes [2]. AR modeling, specifically the Burg algorithm, can be applied to produce a filter that suppresses the clutter. Again a high speed processor capable of performing complex arithmetic in real time would be useful.

The results in the literature [3] have shown that for small data lengths, AR modeling tends to produce higher resolution spectral estimates than the classical

Fourier transform based methods of spectral analysis. The ability to provide high resolution spectral estimates is one of the reasons for the popularity of AR modeling.

The present trend in the above areas is to develop processors and systems that operate in real time. This means operating at speeds fast enough to process the data as soon as it is received rather than storing the data for analysis at some future date. To perform real time speech analysis requires a processor capable of sampling at 12.5 kHz [4]. Real time requirements tend to force designers to turn away from accurate but computationally time consuming algorithms and implement fast Fourier transform (FFT) processors to perform spectral analysis. However, there are a number of AR algorithms that are efficient and can be applied in a real time situation. One such algorithm is the Burg algorithm [3]. Its computational simplicity stems from the fact that it operates directly on the data whereas some other AR algorithms form covariance matrices. Further it uses information already calculated from lower order models to determine higher order model parameters. A real time AR processor would provide a high resolution model of the signal and would be useful in the applications discussed.

Resolution and speed are two reasons why AR modeling should be used in a high resolution real time digital spectrum analyzer (DSA). The requirement of real time means that a hardware implementation of an AR algorithm is necessary. Campbell [5] showed that implementation of a high resolution DSA proposed by Ng [6] on a single 6809 microprocessor system was not acceptable for real time operation. Its failure was due to time consuming address calculations. To overcome this problem a multiboard DSA has been proposed [7,8].

The DSA, fig. 1.1, is composed of several blocks that perform specific functions [8]. A processing stage samples, demodulates, filters and decimates the input data and is followed by an AR stage that models the received data. To determine the frequency spectrum of the AR model, a DFT stage is included with the frequency output being displayed by a video stage. A system consisting of a specialized processor for each stage could perform the DSA function in real time. Orbay [7] has designed and demonstrated real time operation of a processing unit that is dedicated to performing the FFT. The subject of this thesis is the design of an AR processor which is the central element in this DSA and the next element in the DSA that requires development. There are many factors that must be considered if the implementation of the Burg algorithm is to be successful. These factors are presented in this thesis.

Chapter 2 reviews AR modeling techniques and the Burg algorithm. It starts with the basic premise of AR modeling and proceeds to develop the Levinson algorithm. A review of the Levinson algorithm shows that it is fast but it does not have the desired accuracy. The Burg algorithm is reviewed and examined. It is shown that the Burg algorithm is a good candidate for implementation because it is relatively fast and accurate.

Block floating point arithmetic must be used to efficiently perform arithmetic operations in hardware. Chapter 3 examines the errors associated with block floating point arithmetic operations and methods of reducing the errors are introduced. The methods include the use of a tree addition algorithm to reduce roundoff errors in summations and rounding schemes to ensure the stability of the Burg algorithm. An investigation into division algorithms is performed in order to determine a technique

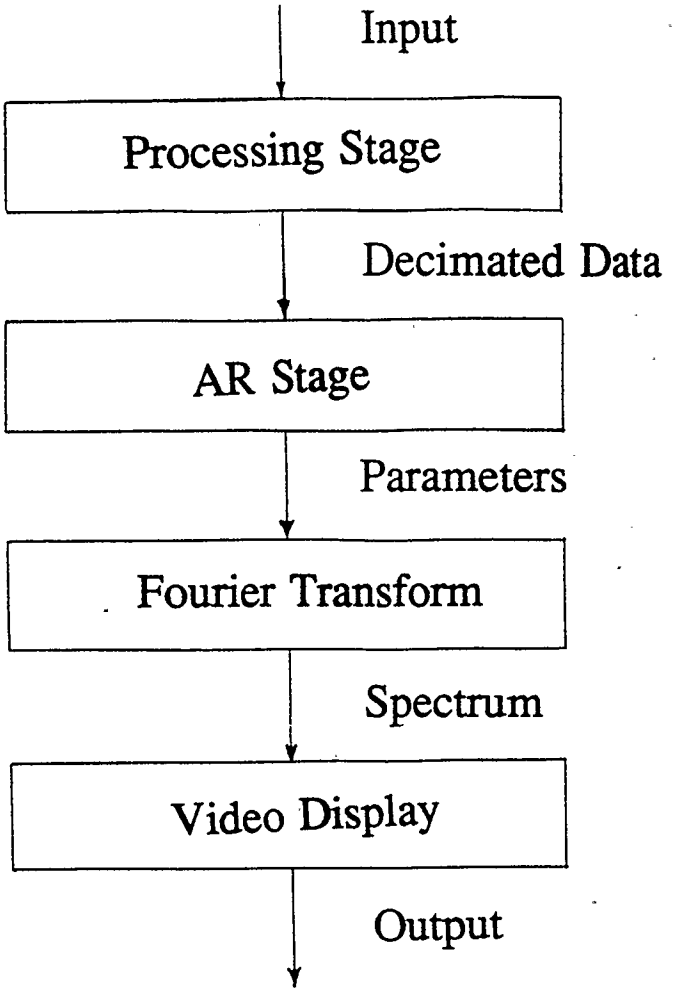


Figure 1.1 A Block Diagram of the DSA

that will satisfy the trade-off between speed and accuracy. It is shown that overflows in the lattice structure can be avoided by scaling the input data by one half.

The hardware development can be based on multiple microprocessors or a microprogrammable system. In chapter 4 the relative merits of these two alternatives are determined. A possible hardware configuration that is suitable for real time operation is proposed. This configuration is determined by examining the computational requirements of the Burg algorithm and finding the simpliest hardware configuration that will perform these requirements in real time. Wire-wrap methods, which provide a high degree of flexibility in the physical layout and fast development times, are used to inter-connect the actual hardware components of a prototype DSA. This is an experimental processor and the ability to change the layout is an asset that outweighs the noise problems that exist in wire-wrap prototypes.

The microprogramming of the Burg algorithm is detailed in chapter 5. The implementation of the different parts of the algorithm are discussed and attention is given to efficient programming methods and rounding schemes that ensure stability. The maximum theoretical clock speed is determined and the run time equation of the Burg algorithm is found to show that the hardware designed in chapter 4 is capable of performing the algorithm in real time.

Results from the hardware implementation are examined in chapter 6. The actual spectral estimates and run-times are compared against their theoretical counterparts and some conclusions are made. The overall performance of the hardware and the block floating point implementation of the Burg algorithm are analyzed. A brief summary is provided and areas where improvements on speed, design and accuracy can be made.

CHAPTER 2
SPECTRAL ESTIMATION BY AUTOREGRESSIVE MODELING

2.0 INTRODUCTION

In many scientific applications [1,2,10] the determination of the spectral content of a signal is very useful. Spectral analysis has been applied to a number of fields mentioned in the first chapter. The chief concern in these areas is the frequency distribution of power in the signal, better known as the Power Spectral Density (PSD). A common approach to determining the PSD is to apply the fast Fourier transform (FFT) algorithm [10] either directly to the data or to the autocorrelation sequence of the data. This technique produces adequate results for a number of situations. However, there are areas where this method provides a very poor estimate of the PSD due to the fact that the resolution of the discrete Fourier transform (DFT) varies inversely with the number of data points. Therefore other methods should be used when examining short data records.

2.1 PARAMETRIC MODELING

The inability of the DFT to produce a high resolution spectral estimate from short data records has led to the development of several other approaches. Among these methods are a number of parametric techniques which attempt to model a signal as a process in which another signal, usually Gaussian white noise, is passed through a filter. The PSD of the filter's output is obtained and used as an estimate of the signal's PSD.

Three types of models can be used to represent a signal [10]. The ARMA model (fig. 2.1), which consists of feedback terms a_i and feedforward terms b_i, generally provides the best estimate of an unknown signal. However the ARMA algorithms are computationally complex and they are not suitable for real time implementation. MA models (fig. 2.2) provide a good estimate for systems that have a finite impulse response but do not perform well for most other signals. Further, determining the MA model parameters involves solving a number of non-linear equations [9]. A number of iterative algorithms have been developed to solve for these parameters but these algorithms are complex and convergence to a stable solution is not guaranteed [9].

The most widely used technique is the AR model (fig. 2.3) [1]. This method yields good spectral results for a large class of signals that are primarily of an all-pole nature. Some of the algorithms used in determining the parameters are computationally efficient [3]. In light of these facts, AR modeling was chosen to be used in this real time DSA application. The theory behind AR modeling and which algorithms are computationally efficient are outlined in this chapter.

2.2 THEORETICAL ASPECTS OF AR MODELING

The AR model assumes that the present value of the signal $x(n)$ can be estimated as the sum of weighted past values of the signal. This is expressed as :

$$\hat{x}(n) = -a_1 x(n-1) - a_2 x(n-2) \cdots - a_p x(n-p) \qquad (2.1)$$

where $\hat{x}(n)$ is the estimate of $x(n)$, a_i are the AR weighting coefficients and p is the model order.

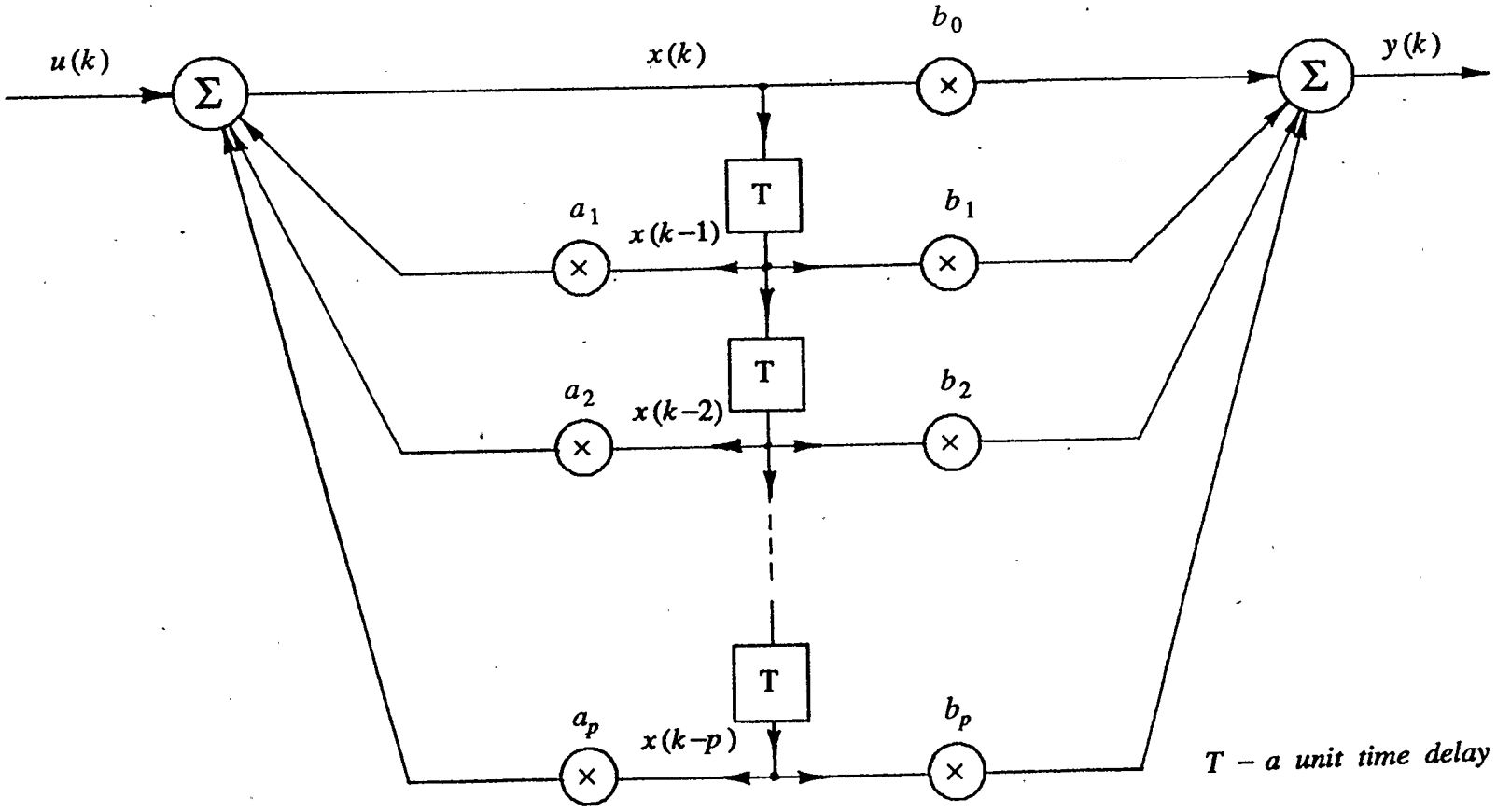


Figure 2.1 A Block Diagram of an ARMA Model

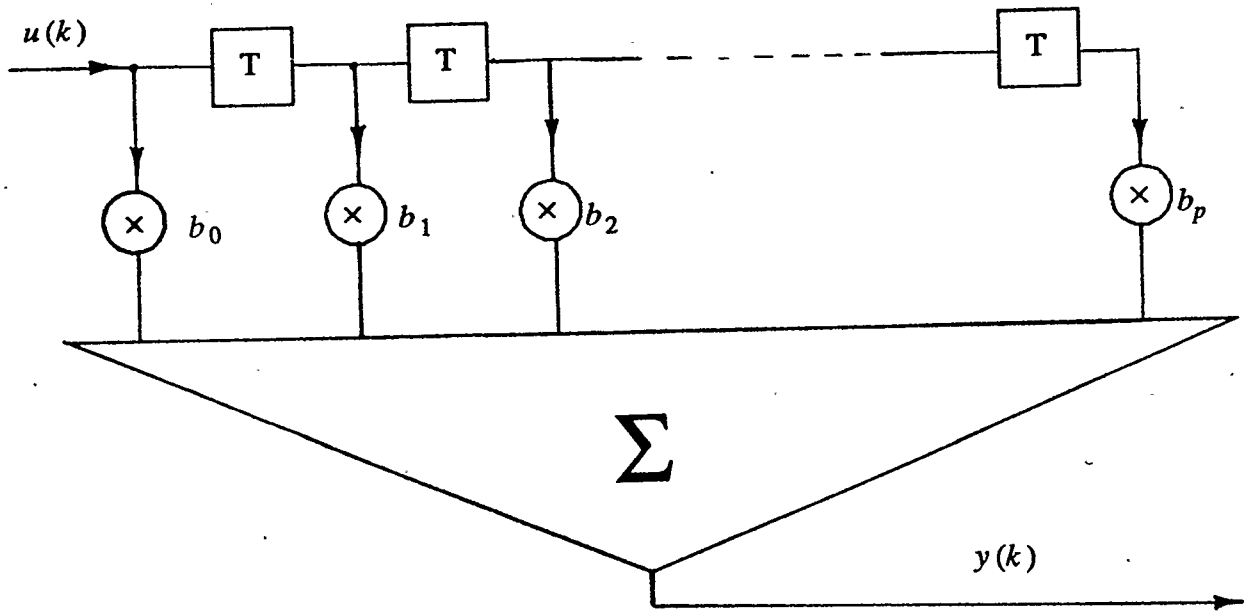


Figure 2.2 A Block Diagram of a MA Model

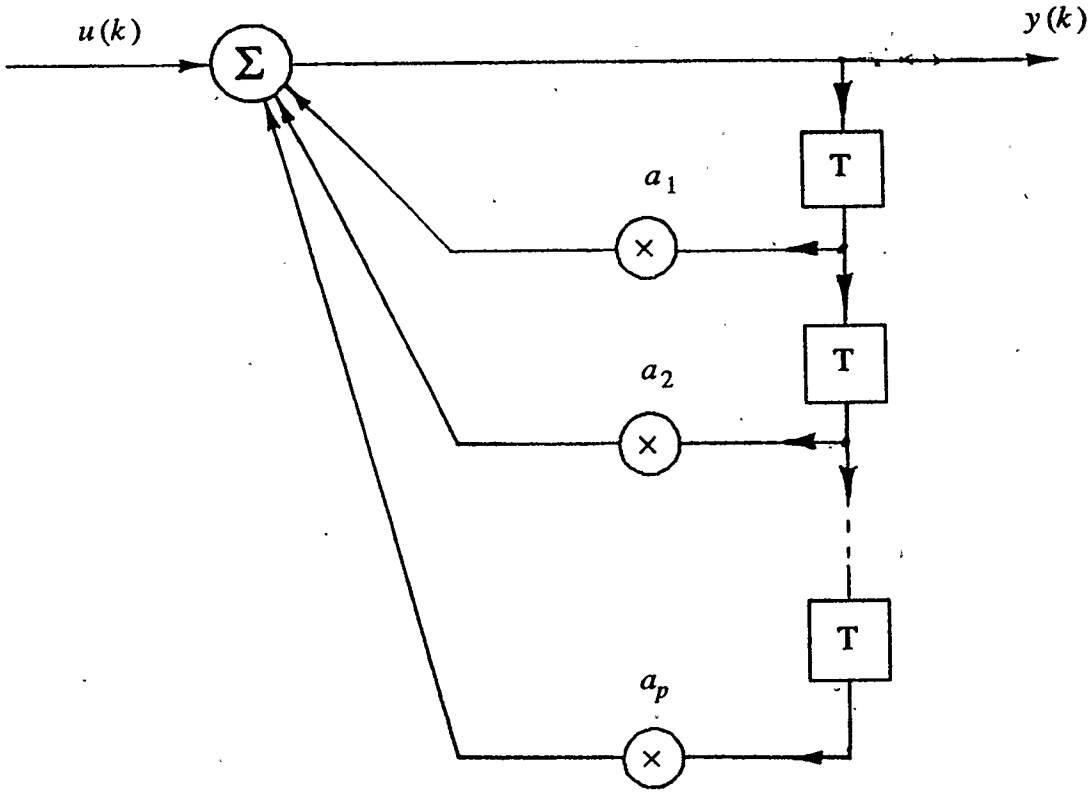


Figure 2.3 A Block Diagram of an AR Model

The goal of AR modeling is to determine the coefficients a_i such that the mean squared error (MSE) between the estimate and the signal value is minimized. This can be expressed as:

$$MSE = \sum_{n=-\infty}^{\infty} (x(n) - \hat{x}(n))^2. \tag{2.2}$$

Substituting for $\hat{x}(n)$ from eqn(2.1) yields:

$$MSE = \sum_{n=-\infty}^{\infty} (x(n) - \sum_{k=1}^{p} a_k\, x(n-k))^2. \tag{2.3}$$

Minimizing with respect to the coefficients a_i yields the Yule - Walker equations [10]

$$\begin{bmatrix} R[0] & R[-1] & \cdots & R[-(p-1)] \\ R[1] & R[0] & \cdots & R[-(p-2)] \\ \vdots & \vdots & & \vdots \\ R[p-1] & R[p-2] & \cdots & R[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_p \end{bmatrix} = \begin{bmatrix} MMSE \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.4}$$

where $MMSE$ is the minimum MSE and

$$R(i) = \lim_{N\to\infty} \frac{1}{2N+1} \sum_{i=-N}^{N} x(n)\, x(n-i). \tag{2.5}$$

$R(i)$ is known as the autocorrelation lag of $x(n)$ with $x(n-i)$ and is an even function:

$$R(i) = R(-i). \tag{2.6}$$

The matrix in eqn(2.4) is a positive definite, symmetric Toeplitz matrix where all the elements along any diagonal are identical. The Toeplitz nature of the matrix was exploited by Levinson [1] in his technique of solving for a_i and the $MMSE$. Further refining of the Levinson algorithm by Durbin [10] yielded the well known Levinson-

Durbin algorithm.

The Levinson-Durbin algorithm is simply a highly efficient method of solving eqn(2.4). The algorithm is initialized by

$$MMSE_0 = R(0) \tag{2.7}$$

$$a_{1,1} = \frac{-R(1)}{R(0)} \tag{2.8}$$

$$MMSE_1 = (1 - |a_{1,1}|^2)\, MMSE_0. \tag{2.9}$$

The recursive algorithm for $k = 2,3, \cdots p$ is then

$$a_{k,k} = \frac{-\left[R(k) + \sum_{l=1}^{k-1} a_{k-1,l}\, R(k-1)\right]}{MMSE_{k-1}} \tag{2.10}$$

$$a_{i,k} = a_{i,k-1} + a_{k,k}\, a^*_{k-i,k-1} \tag{2.11}$$

$$MMSE_k = (1 - |a_{k,k}|^2) \cdot MMSE_{k-1} \tag{2.12}$$

where $a_{i,j}$ is the a_i coefficient determined during the j^{th} iteration.

This algorithm is fast when compared to the more traditional matrix inversion techniques. Methods such as Gaussian elimination and Cholesky decomposition require on the order of $O(p^3)$ operations to generate a solution whereas the Levinson-Durbin algorithm takes $O(p^2)$ operations [10]. The reduction in computational time makes this algorithm a candidate for use in a real time environment. It has been shown by Yung [11] that a parallel VLSI implementation of this algorithm can operate in real time.

The Levinson-Durbin algorithm produces better spectral estimates than standard FFT methods but there are some inherent limitations in the algorithm which reduce its overall resolution [1]. Most of these limitations stem from the assumption of an infinite

data set. In any real time application the data is finite in length, meaning that the autocorrelation lags used in eqn(2.4) are only estimates of the true autocorrelation function and should be expressed as:

$$\hat{R}(i) = \frac{1}{N} \sum_{i=0}^{N-1} x(n)\, x(n-i). \tag{2.13}$$

Using this estimate instead of the true autocorrelation lag implies that the data outside of the finite sequence is assumed to be zero which can be viewed as an implied windowing of the signal. The windowing decreases the resolution of the Levinson algorithm in the same way windowing decreases the resolution of the DFT. The finite data length indicates that the diagonal terms of the matrix in eqn(2.4) are not exactly equal to each other. The Levinson-Durbin algorithm provides an approximate solution to eqn(2.4) since the assumption of infinite data is no longer valid.

If an exact solution to eqn(2.4) is required then the elegant solution proposed by Levinson can not be used and a more generalized inversion algorithm taking $O(p^3)$ operations must be applied. Such algorithms may not be implementable in real time and therefore other AR modeling approaches that retain the real time speed but eliminate these problems must be considered.

2.3 PREDICTION ERROR FILTERS

As the next algorithm under review incorporates the use of prediction error filters (PEF) it is useful to introduce the concept of a PEF at this point in the discussion. Trying to predict a present value from weighted past values using eqn. (2.1) is

equivalent to determining the output of a finite impulse response (FIR) filter run over the data. The coefficients for this filter, shown in fig. 2.4, are the AR model parameters $1, a_1, \ldots a_p$. This filter is known as a PEF because it generates the error $e(k)$ associated with the prediction of the signal.

Until now, the theory has been developed on the basis that a data point can be represented as a linear combination of past values. This is known as forward prediction due to the idea that the PEF is moving forward in time. There is also the possibility of a backward predictor. If all the data is present then a PEF can be run backward in time giving

$$\hat{x}(n-p) = -\sum_{i=1}^{p} b_i x(n-p+i). \tag{2.14}$$

Which means that the present value is a linear combination of future values. The coefficients, b_i, can be found in a similar manner to that used in determining the forward coefficients a_i. When the signal under analysis is shift invariant or independent of time, the backward PEF simply becomes the complex conjugate of the forward PEF, $b_i = a_i^*$. This concept was exploited by Burg [3] in his algorithm to determine the AR coefficients.

2.4 THE BURG ALGORITHM

Due to the implied windowing in the Levinson-Durbin algorithm, the overall resolution is decreased. To alleviate this problem Burg suggested a method that made no assumptions about the data outside of the signal already obtained. In Burg's method, the known part of the autocorrelation sequence $(R_x(0), R_x(1), \ldots R_x(p))$ is

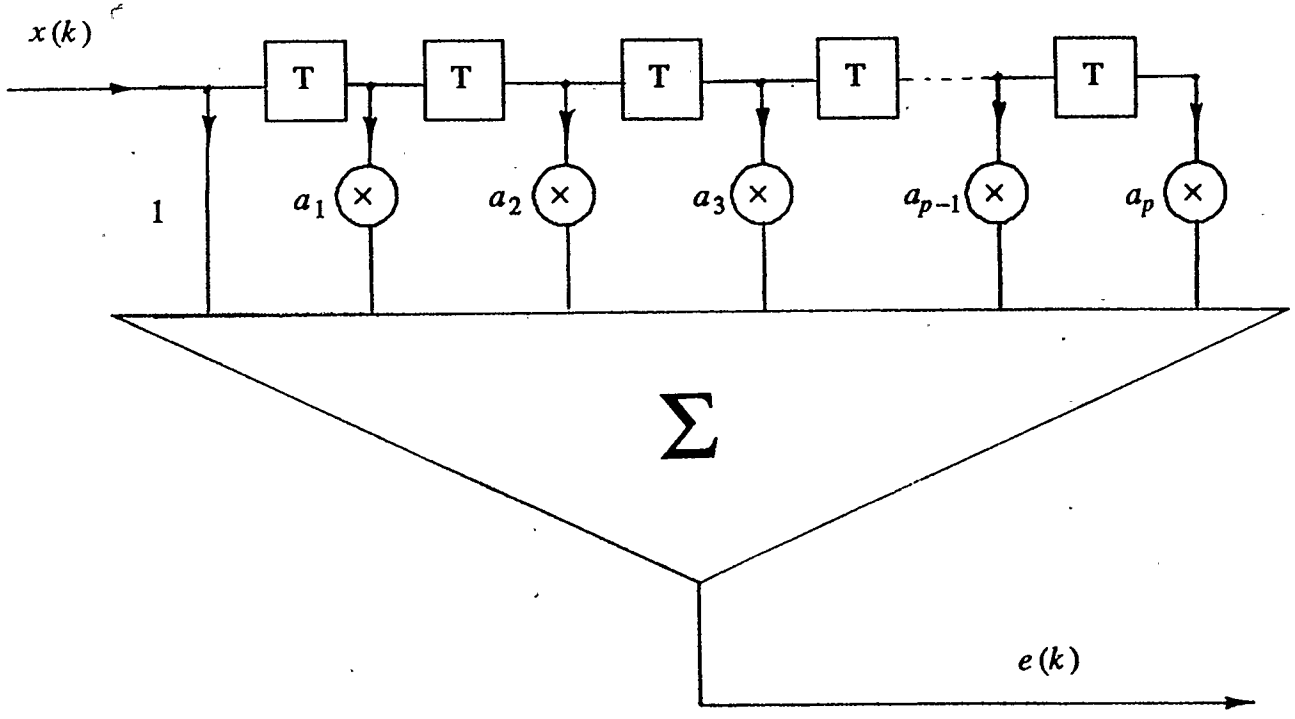


Figure 2.4 A Block Diagram of a Prediction Error Filter

extrapolated to produce an estimate of the unknown autocorrelation sequence ($R_x(p+1)$, $R_x(p+2)\cdots$), effectively removing the windowing of the data. So as not to impose any further constraints on the sequence, Burg proposed that the resulting time series have maximum entropy. Thus the method is known as the maximum entropy method (MEM).

Burg's method operates directly on the data and does not invert the autocorrelation matrix. Burg used a forward and backward PEF to obtain information from the signal. The use of a backward predictor permits information to be obtained about the points that can not be predicted by the forward PEF when the forward PEF is not allowed to be "run off" the data. Fig. 2.5 demonstrates what is meant by not running off of the data.

Burg then minimized the sum of the forward $e_{j,n}$ given by

$$e_{j,n} = x(n) + \sum_{k=1}^{j} a_{k,j}\, x(n-k), \tag{2.15}$$

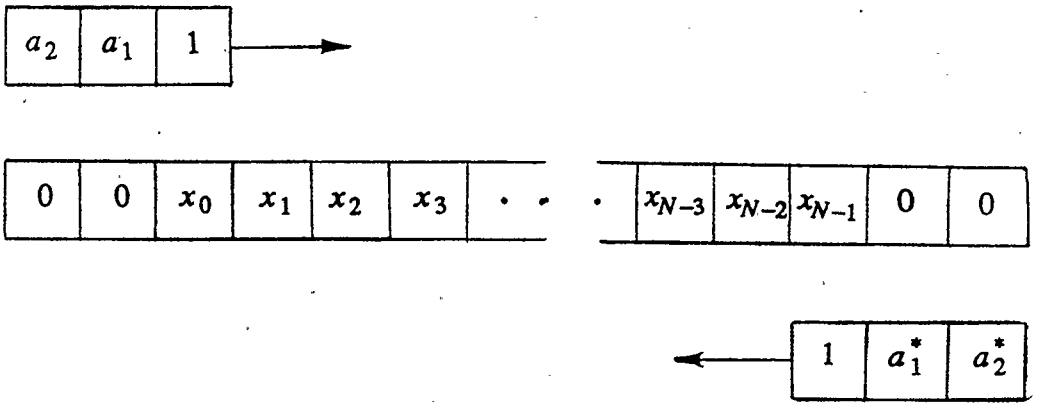
and the backward error

$$b_{j,n} = x(n-j) + \sum_{k=1}^{j} a^{*}_{k,p}\, x(n-j+k), \tag{2.16}$$

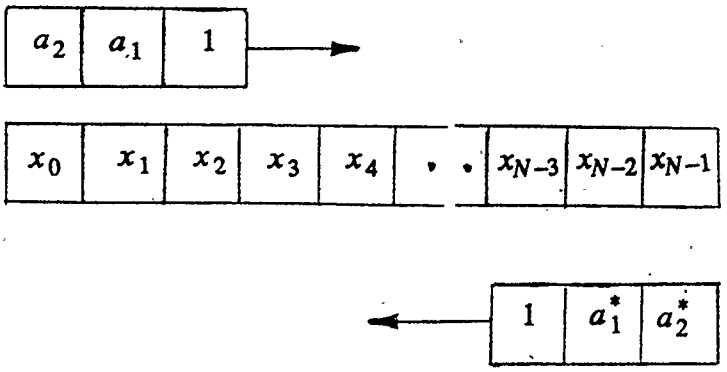
where j is the model order and $j \le n < N$. The minimization of the error

$$P_j = \sum_{n=j}^{N-1} |e_{j-1,n}|^2 + |b_{j,n}|^2 \tag{2.17}$$

yields the reflection coefficient



A) Initial Locations for PEFs that are Run Off the Data



B) Initial Locations for PEFs that are not Run Off the Data

Figure 2.5 What is Meant by the Term "Running Off of the Data

$$a_{j,j} = -2 \sum_{n=j}^{N-1} \frac{e_{j-1,n}\ b^*_{j-1,n-1}}{D_j} \tag{2.18}$$

where

$$D_j = \sum_{n=j}^{N-1} |e_{j-1,n}|^2 + |b_{j-1,n-1}|^2. \tag{2.19}$$

The remaining PEF coefficients, $a_{i,j}$, $0 \le i \le j-1$, are then determined using the Levinson recursion given in eqn(2.11). To update the errors, two prediction error filters, based on eqns (2.15) and (2.16), can be applied directly to the data which is a time consuming process. To reduce the computational time, Burg proposed the use of a lattice structure that makes use of the reflection coefficient and the prediction errors of the previous stage to update the errors.

The PEF's can be folded into the lattice structure. Substituting for $a_{k,j}$ from eqn(2.11) into eqn(2.15) yields

$$e_{j,n} = x(n) + \sum_{k=1}^{j-1} (a_{k,j-1} + a_{j,j}\ a^*_{j-k,j-1})\ x(n-k) + a_{j,j}\ x(n-p). \tag{2.20}$$

Incorporating eqn(2.16) into eqn(2.20) yields

$$e_{j,n} = e_{j-1,n} + a_{j,j}\ b_{j-1,n-1}. \tag{2.21}$$

Similarly, the backward error is given by

$$b_{j,n} = b_{j-1,n-1} + a^*_{j,j} e_{j-1,n}. \tag{2.22}$$

The lattice structure, shown schematically in fig. 2.6, is obtained using these two equations. Using the lattice structure to update the prediction errors and recursively computing the PEF saves a great deal of time in the Burg algorithm. In fact, this

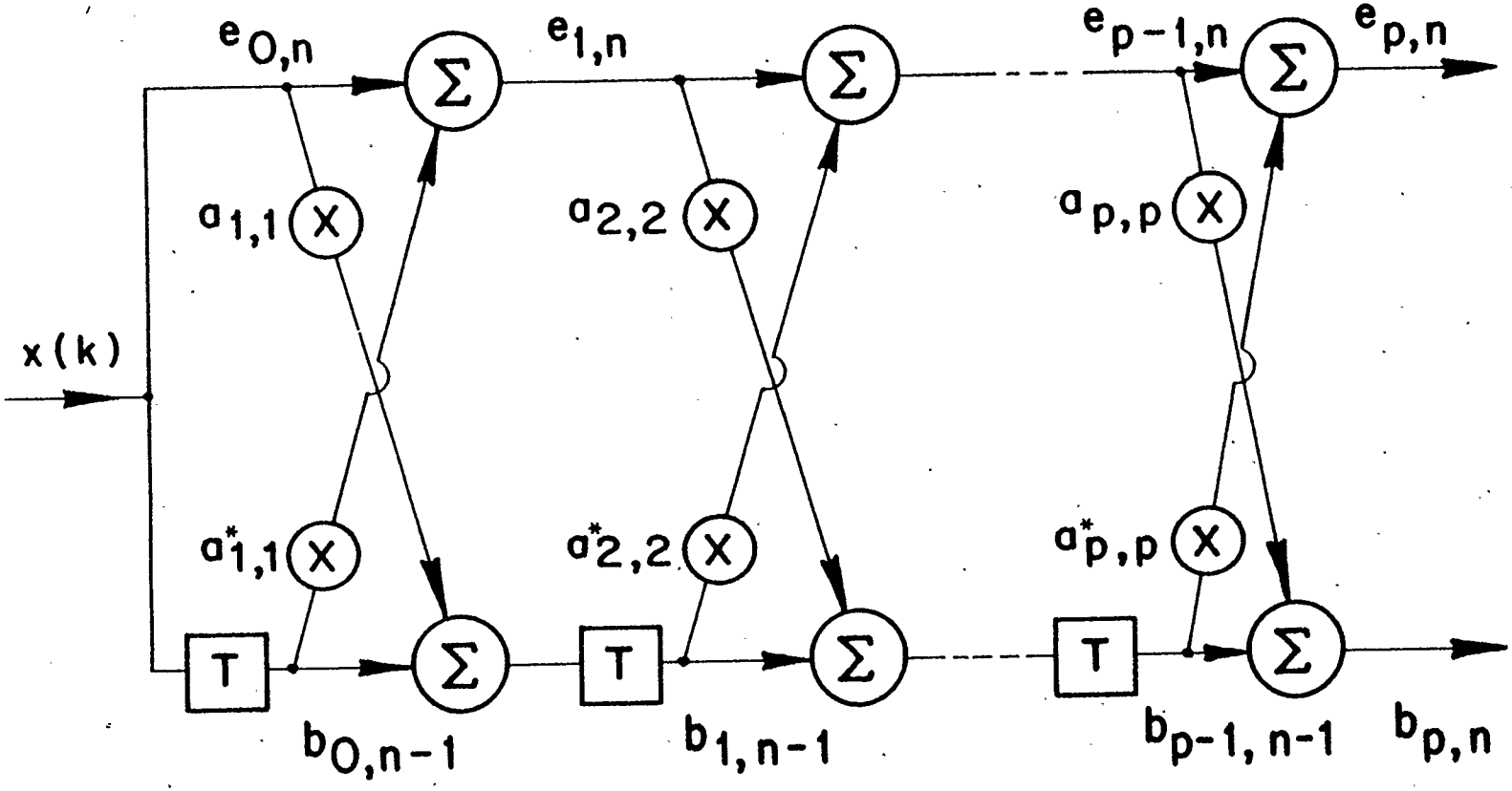


Figure 2.6 A Schematic Representation of the Lattice Structure

method takes $O(p^2)$ operations to determine the PEF coefficients which implies that the algorithm is a prime candidate for implementation in real time [10].

The literature indicates that there are some inherent problems with the Burg algorithm [10,12]. The occurrence of closely spaced multiple peaks, line splitting [12], in a spectral estimate where only one peak should exist, is the most serious problem in the Burg algorithm. Line splitting arises when the algorithm is applied to signals with high signal to noise ratios (e.g. SNR = 40dB) and only a few data points are present (e.g. N = 15). Frequency biasing [13] results when the initial phase of a sinusoid to be modeled is non-zero and there are only a few data points present. Though these limitations might appear serious, in most applications there are a sufficient number of data points to mitigate these effects and a number of authors [14,15] have also proposed methods of reducing the effects if they appear significant. As the Burg algorithm has better resolution than the Levinson algorithm, by removing the implied windowing, and operates much faster than most matrix inversion routines, it was chosen as the algorithm that would be implemented in this thesis.

2.5 SUMMARY

The concepts of AR modeling and the relative strengths and weaknesses of the Burg and Levinson algorithms have been examined in this chapter. The Burg algorithm was chosen for implementation as it offers better resolution than the Levinson algorithm and operates much faster than other algorithms that incorporate matrix inversion routines.

CHAPTER 3
BLOCK FLOATING POINT ARITHMETIC

3.0 INTRODUCTION

As the goal of this thesis is to develop a high speed implementation of the Burg algorithm, high speed hardware components must be used. Currently there are a number of commercial chips capable of providing high speed additions, subtractions and multiplications. The vast majority of these chips operate on, and output, data that is represented by a string of binary (1 or 0) bits in a predetermined format known as Fixed Point (FXP). Though a floating point (FLP) format could be used, it would be very time consuming and inefficient. A compromise between FXP and FLP is block floating point (BFP) format. With this format the data is stored in FXP format and a scale factor associated with a block of data is also stored. Due to this scale factor, variable scaling can be used in BFP as opposed to the predetermined scaling that occurs in FXP [16]. When variable scaling is used the data is only scaled when necessary, unlike prescaling which tends over scale the data as the prescaling value is usually determined by some form of worst case analysis. This implies that BFP is more accurate than FXP. As BFP still retains the speed of FXP and provides greater accuracy, it is used in this implementation. As BFP has a finite precision it is possible for errors to occur when an operation produces a result that exceeds the bounds of the fixed format. How these errors occur and what is done to reduce their effect is discussed in this chapter.

The operation of division is the only basic arithmetic operation that is not performed by a dedicated chip. To perform a division, a software algorithm must be

used. A number of algorithms available to perform this operation are examined and compared in terms of accuracy, speed and ease of implementation.

3.1 FIXED POINT REPRESENTATION

In FXP a number is represented by a string series of binary bits. A common FXP format is fractional two's complement (FTC). The number x can be represented in FTC notation as:

$$x = -s_0 + \sum_{i=1}^{r-1} s_i \, 2^{-i} \tag{3.1}$$

where $-1 \leq x < 1$ and s_i are binary numbers. The number of bits, r, is limited by the hardware that is used. In this thesis r is taken to be 16 which results in an acceptable trade-off between numerical accuracy and hardware complexity.

3.2 BFP ADDITION

When adding two BFP numbers there is a possibility that the result will require added precision. Consider the addition of a number with itself. If the original value is represented by 16 bits then it possible that twice the original value may need 17 bits to be accurately represented. In BFP a 17 bit number must be reduced to a 16 bit value and the scaling factor adjusted. With the architecture selected, this was accomplished by scaling the original 16 bit numbers down to 15 bit values. The above addition then produces a 16 bit result which can be represented in FXP format.

Scaling, a method of reducing the number of bits present, can be performed using a number of techniques. In the simpliest method, down rounding, the 16 bit value is divided by two and the remainder is dropped This is accomplished by shifting the

original value to the right by one and dropping the least significant bit (LSB). A negative bias is introduced as a result of dropping the LSB. Up rounding can be used if a negative bias is undesirable. In this scheme, a one is added the LSB and then the result is shifted to the right by one. This technique introduces a positive bias into the answer because of the addition of the one to the LSB.

Other schemes must be employed where no bias is tolerable. The magnitude truncation method adds the sign bit to the LSB before shifting. Assuming equal probability of positive and negative numbers, this scheme does not introduce an overall bias as the negative numbers are positively biased and the positive numbers are negatively biased. In random bit addition, a random bit is added to the LSB so that no bias is introduced when rounding. Implementation of this scheme in hardware requires a random bit generator in addition to the hardware used in the previous scaling techniques.

All of the scaling schemes discussed introduce errors into the resulting 16 bit value. These errors occur randomly and the mean, mean square, and variance are used to describe their effects. Fig. 3.1 shows the error distributions for all four schemes and table 3.1 gives the mean, mean square, and variance of the three in terms of the weight of the LSB (Δ). Though these results are given without proof, they can be easily verified by example.

3.3 BFP SUMMATION ALGORITHMS

The last section focused on the errors that occurred due to a single rounding operation. This section is concerned with how these errors grow when a large set of

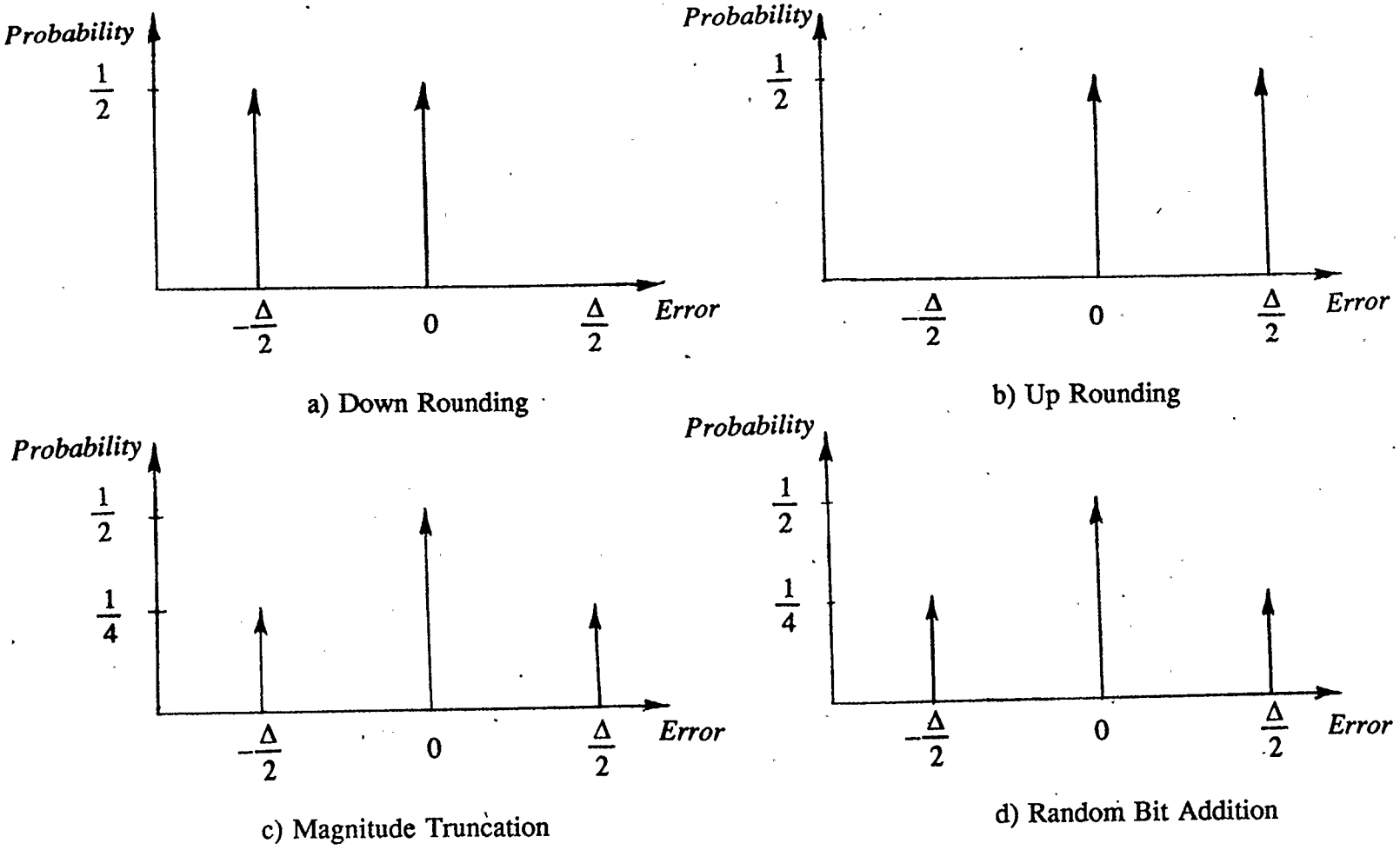


Figure 3.1 Round-off Error Distributions for the Various Rounding Schemes

Table 3.1 Round-off Errors Associated with Addition/Subtraction

Scaling Method	Mean μ	Variance σ^2	Mean Square
Down Rounding	$-\frac{\Delta}{4}$	$\frac{\Delta^2}{16}$	$\frac{\Delta^2}{8}$
Up Rounding	$\frac{\Delta}{4}$	$\frac{\Delta^2}{16}$	$\frac{\Delta^2}{8}$
Magnitude Truncation	0	$\frac{\Delta^2}{8}$	$\frac{\Delta^2}{8}$
Random Bit Addition	0	$\frac{\Delta^2}{8}$	$\frac{\Delta^2}{8}$

BFP numbers are added together to form the summation

$$Y = \sum_{i=0}^{N-1} x_i . \tag{3.2}$$

Clearly, the growth of these errors will depend on the method of summation. Two summation techniques, the accumulation algorithm and the tree addition algorithm, are analyzed in this section to determine which one performs the best in terms of avoiding roundoff error.

3.3.1 THE ACCUMULATION ALGORITHM

When forming a summation, the most straightforward method is the accumulation technique. In this technique a number x_i is added to the sum of all the previous numbers Y_{i-1} to form a new sum Y_i

$$Y_i = Y_{i-1} + x_i \tag{3.3}$$

for $i = 1 \cdots N-1$ and $Y_0 = x_0$. In floating point arithmetic, the sum is permitted to grow as successive terms are added. This luxury is not present in BFP arithmetic because when the sum overflows it must be shifted to stay within the FXP format. In turn all future values added to the sum must also be scaled. The errors generated by these scalings are shown in fig. 3.2. The errors generated by prescaling of the numbers before they are included in the sum are denoted by p_i and the errors due to scaling of the sum are denoted by ε_i . The roundoff error in the summation, ε_{acc}, is given by :

$$\varepsilon_{acc} = \sum_{i=0}^{m-1} \varepsilon_{m-i-1} 2^{-i} + \sum_{k=1}^{m} \sum_{j=2^{k-1}}^{2^k-1} p_{j-1} 2^{k-m} \tag{3.3}$$

where $m = int(\log_2(N))$. The worst case mean error μ_{accWC} can be expressed in terms

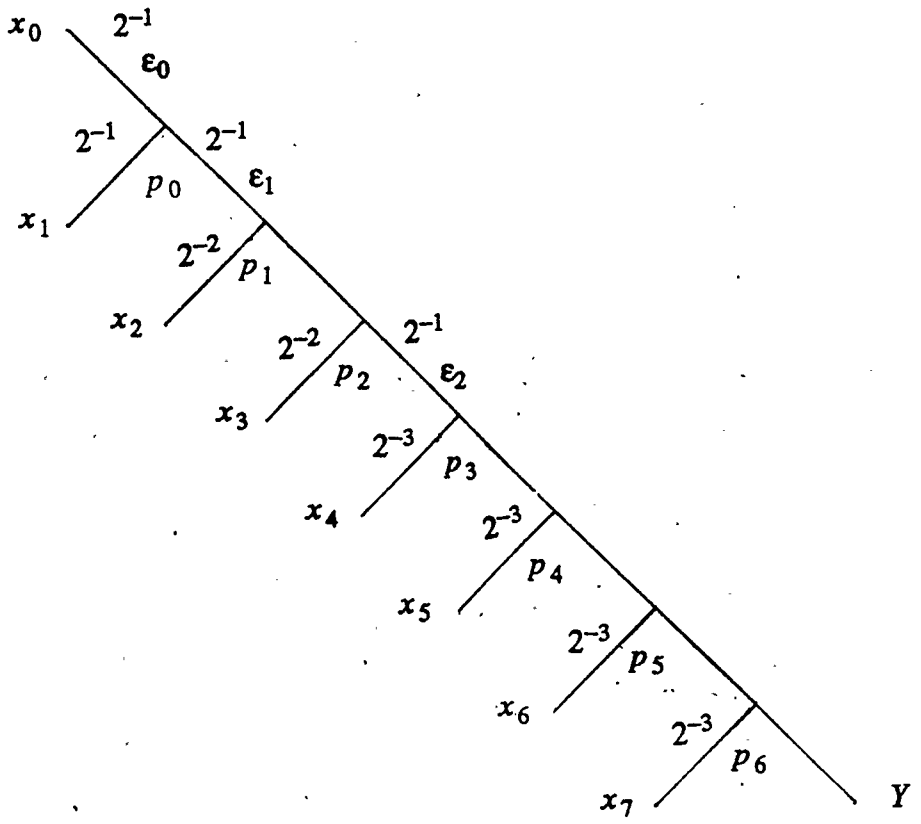


Figure 3.2 Roundoff Error Generated in the Accumulation Algorithm for Summation

of the mean error involved in a single scaling μ_s as:

$$\mu_{accWC} = \sum_{i=0}^{m-1} \mu_s 2^{-i} + \sum_{k=1}^{m} \sum_{j=2^{k-1}}^{2^k-1} \mu_{acc}(k) 2^{k-m} \tag{3.4}$$

where $\mu_{acc}(k)$ denotes the variable mean error arising from the prescalings. The last term contains a variable mean due to the fact that the prescaling increases with each overflow. A relationship between this variable mean and the mean for one scaling can be found by using the concept of multiple scaling [16]. If a variable is scaled k times then the resulting mean error ($\mu_{acc}(k)$) is:

$$\mu_{acc}(k) = \sum_{i=0}^{k-1} \mu_s \; 2^{-i} \tag{3.5}$$

which reduces to

$$\mu_{acc}(k) = \mu_s \; 2(1 - 2^{-k}). \tag{3.6}$$

Substituting this into eqn(3.4) and determining the closed form expression yields:

$$\mu_{accWC} = \frac{2}{3}\mu_s[2^{m+1} - 2^{-m+1}]. \tag{3.7}$$

Using a similar analysis the worst case variance, σ^2_{accWC}, due to a single scaling, σ^2_s, is found to be

$$\sigma^2_{accWC} = \frac{4}{21}\sigma^2_s(4 \cdot 2^m + 7 - 7 \cdot 2^{-m} - 4 \cdot 2^{-2m}) \tag{3.8}$$

The mean square error is:

$$MSE = \mu^2_{accWC} + \sigma^2_{accWC} \tag{3.9}$$

which becomes

$$MSE = \frac{4}{9}\mu_s^2(\,2^{m+1} - 2^{-m+1}\,)^2 + \frac{4}{21}\sigma_s^2\,(\,4{\cdot}2^m + 7 - 7{\cdot}2^{-m} - 4{\cdot}2^{-2m}\,). \quad (3.10)$$

3.3.2 THE TREE ADDITION ALGORITHM

The tree algorithm attempts to reduce the errors arising from the addition of a large sum and a small data value by only adding numbers of the same magnitude. Fig. 3.3 shows the operation of the algorithm and the errors that can arise in this kind of addition. From this figure, the roundoff error ε_{tree} can be determined by:

$$\varepsilon_{tree} = \sum_{k=1}^{m}\sum_{i=0}^{2^{k-1}} \varepsilon_{k,i}\; 2^{-k+1}. \quad (3.11)$$

Therefore the mean error μ_{tree} is:

$$\mu_{tree} = \sum_{k=1}^{m}\sum_{i=0}^{2^k-1} \mu_s\; 2^{-k+1} \quad (3.12)$$

which simplifies to:

$$\mu_{tree} = 2m\,\mu_s. \quad (3.13)$$

By a similar analysis the variance is found to be:

$$\sigma_{tree}^2 = 4\;\sigma_s^2\;[1 - 2^{-m}]. \quad (3.14)$$

The mean square error is:

$$MSE_{tree} = 4m^2\;\mu^2 + 4\sigma_s^2\;[1 - 2^{-m}]. \quad (3.15)$$

A graph comparing the normalized MSE ($\Delta = 1$) of the two algorithms is shown in fig. 3.4. The biased errors arise when up rounding or down rounding schemes are used. These errors are said to be biased because of their non-zero mean error.

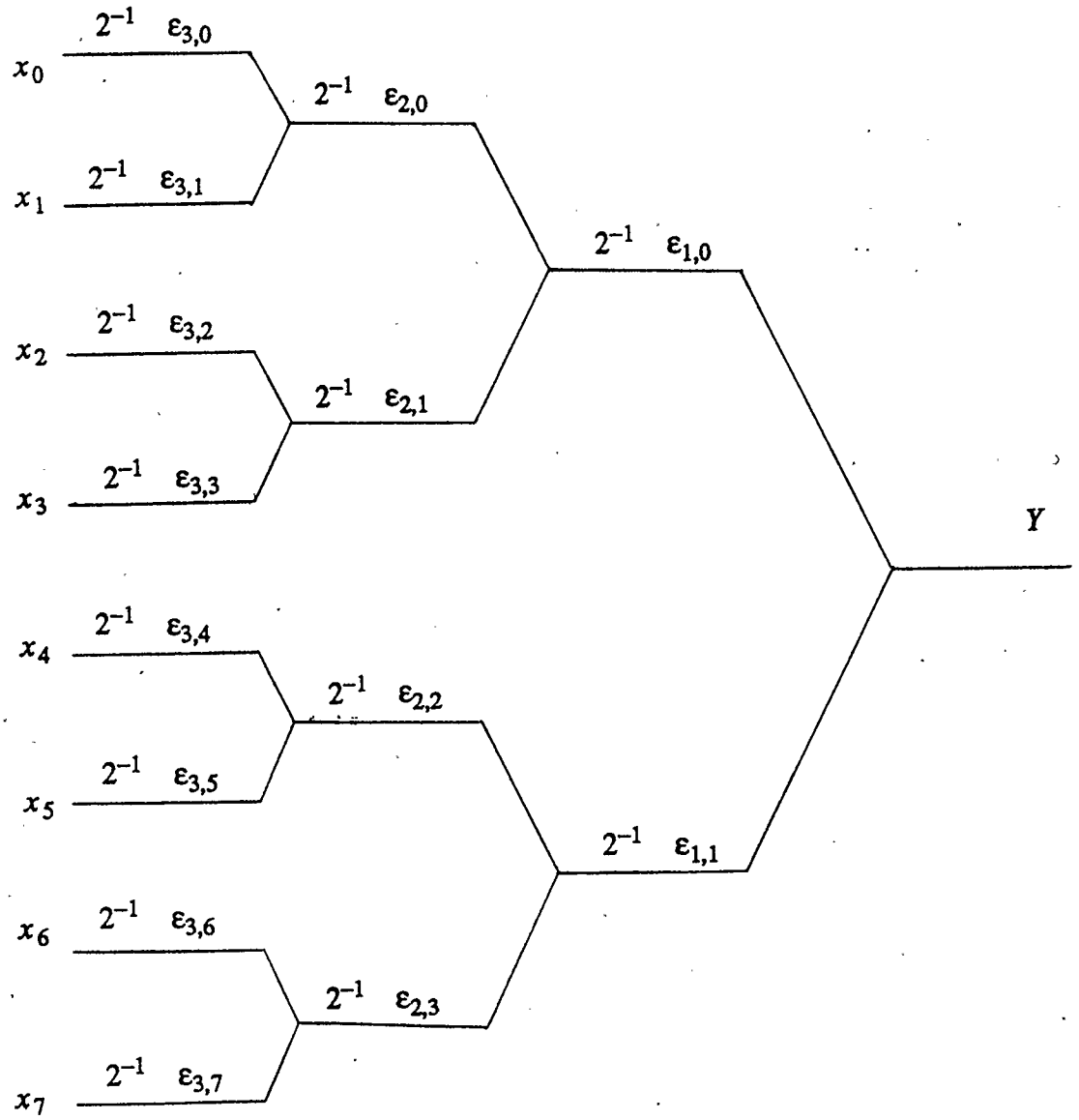


Figure 3.3 Roundoff Error Generated in the Tree Algorithm for Summation

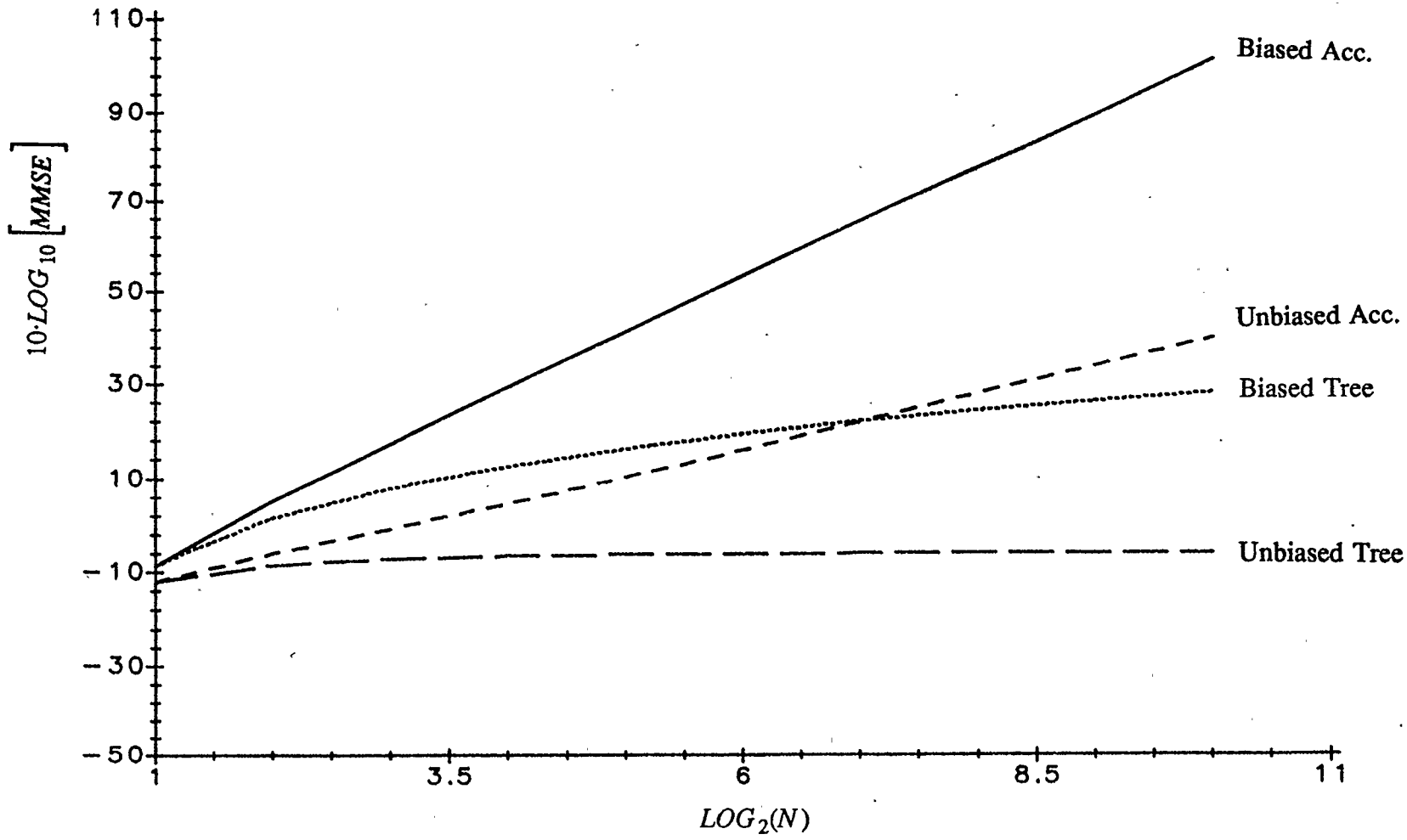


Figure 3.4 Errors Associated with Block Floating Point Summations

Unbiased, or zero-mean, errors occur when magnitude truncation or random bit addition are applied. Since the MSE in the tree algorithm does not grow as quickly as in the accumulation algorithm, the tree algorithm was chosen to perform the required summations.

3.4 MULTIPLICATION

In general, the multiplication of two 16 bit numbers results in a 32 bit number. In FXP, this value must be reduced to 16 bits before proceeding to the next arithmetic operation. This reduction can be accomplished by applying one of the rounding procedures previously discussed. The error distributions that occur when the various rounding schemes are applied to the multiplication [5] are shown in fig. 3.5. Again it is useful to evaluate these errors in terms of mean, mean square, and variance values and these quantities are shown in table 3.2.

3.5 DIVISION

Division is performed by a software algorithm as there are no commercial chips that are dedicated to performing division. The speed, accuracy, and ease of implementation must be considered in selecting a suitable algorithm. As most algorithms can be classified as *subtraction and shift* or as *convergence* type algorithms, it is useful to examine an algorithm from each type to determine their various strengths and weakness. A convergence type division algorithm based on the Taylor series expansion of $(1-x)^{-1}$ was chosen.

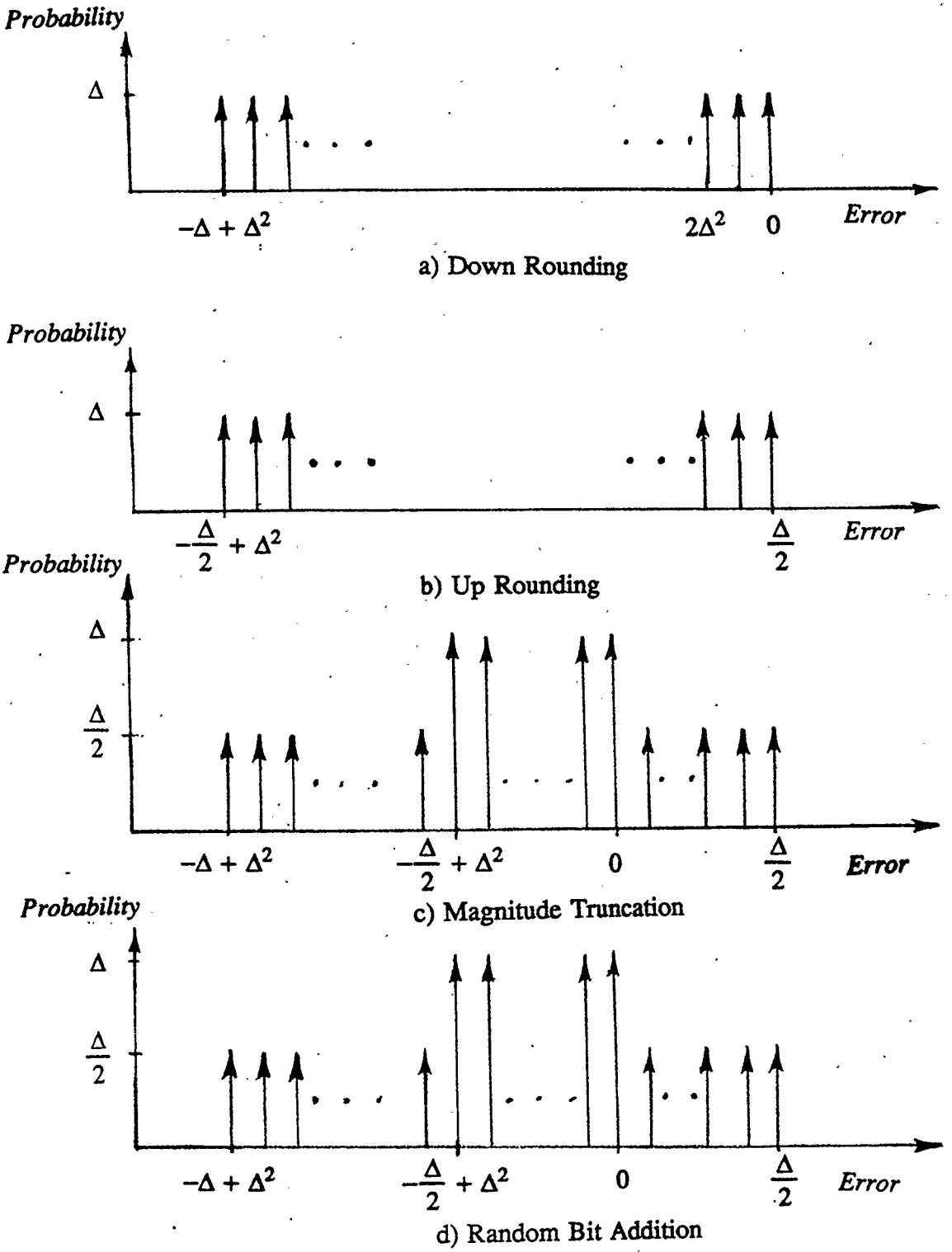


Figure 3.5 Round-off Errors in Multiplication

Table 3.2 Errors Associated with Multiplication and Sequential Division

Scaling Method	Mean μ	Variance σ^2	Mean Square
Down Rounding	$-\frac{\Delta}{2}$	$\frac{\Delta^2}{12}$	$\frac{\Delta^2}{3}$
Up Rounding	0	$\frac{\Delta^2}{12}$	$\frac{\Delta^2}{12}$
Magnitude Truncation	$-\frac{\Delta}{4}$	$\frac{7\Delta^2}{48}$	$\frac{5\Delta^2}{24}$
Random Bit Addition	$-\frac{\Delta}{4}$	$\frac{7\Delta^2}{48}$	$\frac{5\Delta^2}{24}$

3.5.1 SEQUENTIAL SUBTRACT AND SHIFT DIVISION METHODS

This class of algorithms can be described as the pencil and paper method [17,18]. The operations involved in this technique are best shown by way of an example. The division of 13 (the dividend) by 4 (the divisor) produces a quotient of 3 and remainder of 1 as shown in fig. 3.6. This class of algorithms produce the most accurate answer possible with FXP arithmetic and the roundoff error distributions are identical to those of multiplication which were given in fig 3.5 and table 3.2. These division algorithms are accurate but slow because they are sequential in nature meaning that the current addition or subtraction cannot be performed until the results of the previous operation are known. An examination of a very efficient *subtract and shift* method known as the non-restoring algorithm gives an indication of the relative speed of this type of division. From the flowchart, given in fig 3.7 [18], it can be seen that it will require at least 5 cycles per quotient bit. Two 16 bit divisions (the numerator in the Burg algorithm is complex) require close to 180 cycles to produce the result. This method could easily require 200 cycles when initialization and sign correction steps are included to handle signed numbers.

3.5.2 CONVERGENCE DIVISION

It is possible to perform division by iterative multiplications when a hardware multiplier is present [17]. One such method, the Newton-Raphson method, finds the inverse of the denominator and then multiplies the inverse with the numerator to perform the division. The iteration equation is

```
    1  1  0  1                          (13) Dividend
    0  1  0  0                          (4) Divisor

                          0  0  1  1     (3) Quotient
0  1  0  0   | 0  0  0  0  1  1  0  1
              |    0  1  0  0
              |    ------------------
              |       0  1  0  1
              |       0  1  0  0
              |       ------------
              |       0  0  0  1        (1) Remainder
```

Figure 3.6 An Example of the "Paper and Pencil Method"

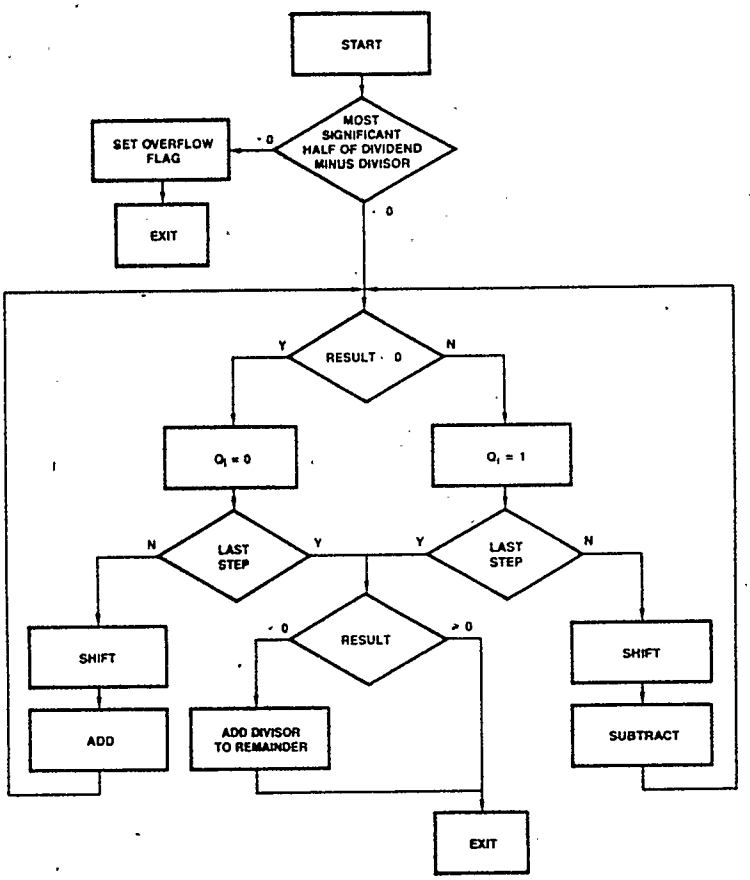


Figure 3.7 The Flowchart for Nonrestoring Division

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{3.16}$$

where

$$f(x) = \frac{1}{x} - B. \tag{3.17}$$

The root of eqn. (3.17) is $x = 1/B$, the reciprocal of the denominator B. Taking the derivative of (3.17) and substituting into (3.16) yields:

$$x_{i+1} = x_i(2 - B\ x_i). \tag{3.18}$$

This algorithm converges quadratically [17] meaning that it will only take a few iterations to produce the reciprocal provided there is a good initial guess. Though this approach possesses high speed there is a problem associated with the initial guess. The authors of [17] indicate that the initial guess, x_0, must fall in the range $0 < x_0 < 2/B$ to guarantee convergence of the algorithm. To obtain an accurate initial guess a ROM look up table is needed, thereby requiring further hardware.

3.5.3 TAYLOR SERIES EXPANSION

The division algorithms examined have either been too time consuming or require additional hardware. Thus an algorithm that is both fast and implementable will be independently developed. The method generates the reciprocal 1/B by forming the Taylor series expansion of $(1 - x)^{-1}$. Expanding this function in a Taylor series yields

$$(1 - x)^{-1} = 1 + \sum_{n=1}^{\infty} x^n \tag{3.19}$$

where

$$B = 1 - x. \tag{3.20}$$

This algorithm converges for:

$$-1 < x < 1. \tag{3.21}$$

Noting that the denominator in the Burg algorithm is always positive it may be normalized so that:

$$\frac{1}{2} \leq B < 1 \tag{3.22}$$

implying that x falls within the range:

$$0 < x \leq \frac{1}{2}. \tag{3.23}$$

With a number system where the LSB is 2^{-15} it takes 15 terms in the series to accurately form the inverse for the largest value of x, (x=1/2). The sum of all higher order terms produces a value less than the LSB. Eqn(3.19) can be rewritten as:

$$(1 - x)^{-1} \approx 1 + \sum_{n=1}^{15} x^n. \tag{3.24}$$

Since $x \leq 1/2$, the summation term of eqn(3.24) is always less than one and there is no need to check for overflows, indicating that this algorithm could perform division at a high speed and be easily implemented.

The algorithm was written and found to require only 21 cycles to perform the division, a vast improvement over the 200 cycles of the non-restoring algorithm. The roundoff error in the algorithm has yet to be examined. The generated errors are shown in fig. 3.8 The worst case error (ε_{WC}) can be determined to be:

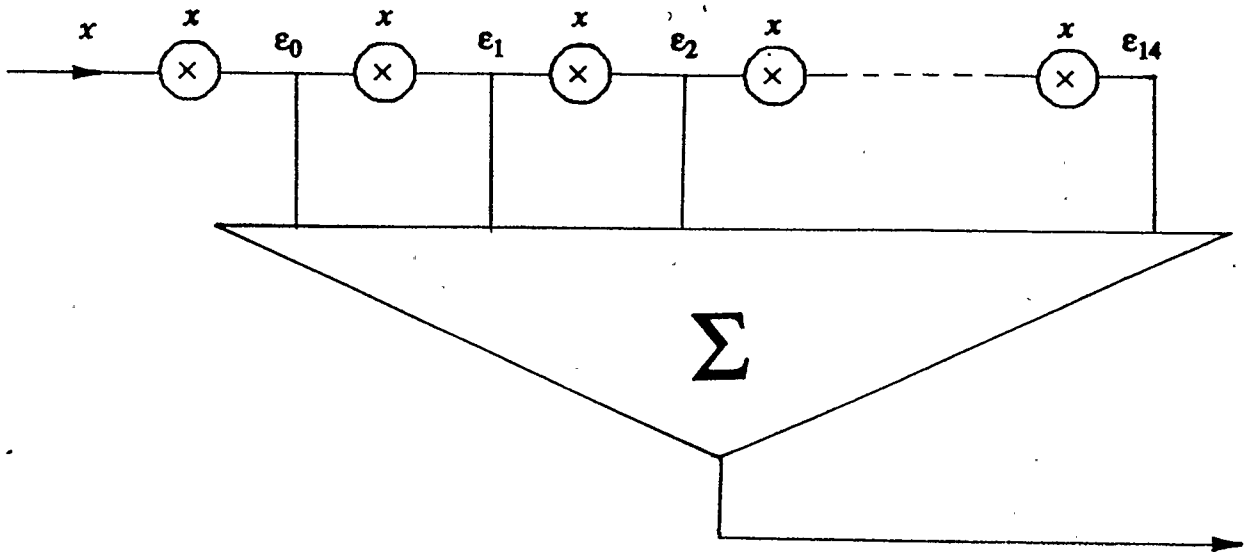


Figure 3.8 Roundoff Error in the Unmodified Taylor Series used in Determining $(1 - x)^{-1}$

$$\varepsilon_{WC} = \sum_{i=0}^{14} \sum_{k=0}^{i} \varepsilon_{14-i}\, x^k. \tag{3.26}$$

Expressing the mean error, μ_{WC}, in terms of the mean error of a single multiplication μ_m, we have

$$\mu_{WC} = \sum_{i=0}^{14} \sum_{k=0}^{i} \mu_m x^k \tag{3.27}$$

which reduces to

$$\mu_{WC} = 15\, \frac{\mu_m}{1-x} - \frac{x}{1-x} \cdot \frac{1-x^{15}}{1-x}\, \mu_m. \tag{3.28}$$

For the region $0 < x \leq 1/2$, this is a maximum when x = 1/2 giving:

$$\mu_{WC} = 30\mu_m - 2\mu_m(1 - 2^{-15}) \tag{3.29}$$

or

$$\mu_{WC} \approx 28\mu_m. \tag{3.30}$$

A similar analysis yields a variance of

$$\sigma_{WC}^2 \approx \frac{59}{3}\sigma_m^2 \tag{3.31}$$

where σ_m^2 is the variance associated with a single multiplication.

The mean square error is

$$MSE \approx 784\mu_m^2 + \frac{59}{3}\sigma_m^2 \tag{3.32}$$

Values are shown in table 3.3 for the various rounding schemes. The error in the unmodified taylor series algorithm for division appear large when compared to the

error in *sequential subtract and shift* methods of division.

One way to reduce this error is to perform the summation in the following manner:

$$(1-x)^{-1} = 2[1/2 + x(1/2 + x(1/2 + \cdots x(1/2 + x/2) \cdots)]. \tag{3.33}$$

This can be recursively expressed as:

$$d_i = 1/2 + x\ d_{i-1} \tag{3.34}$$

for i = 1 ... 14 and:

$$d_0 = 1/2 + x/2. \tag{3.35}$$

The initial division by two ensures that there will be no overflow during the summation and a corresponding up scaling by two is required at the end of this procedure. Though this form may appear cumbersome, it possesses an improved roundoff behavior when compared to the previous form. The errors due to the multiplications and the initial scaling are shown in fig. 3.9. From this figure, the worst case error can be determined as:

$$\varepsilon_{WC} = \sum_{i=0}^{13} \varepsilon_i\ x^i + \varepsilon_{14}\, x^{14}. \tag{3.36}$$

Again the maximum worst case mean and variance can be found by setting x = 1/2. The mean value then becomes:

$$\mu_{WC} = 2\mu_m\ (1 - 2^{-14}) + \mu_s\ 2^{-14} \tag{3.37}$$

or

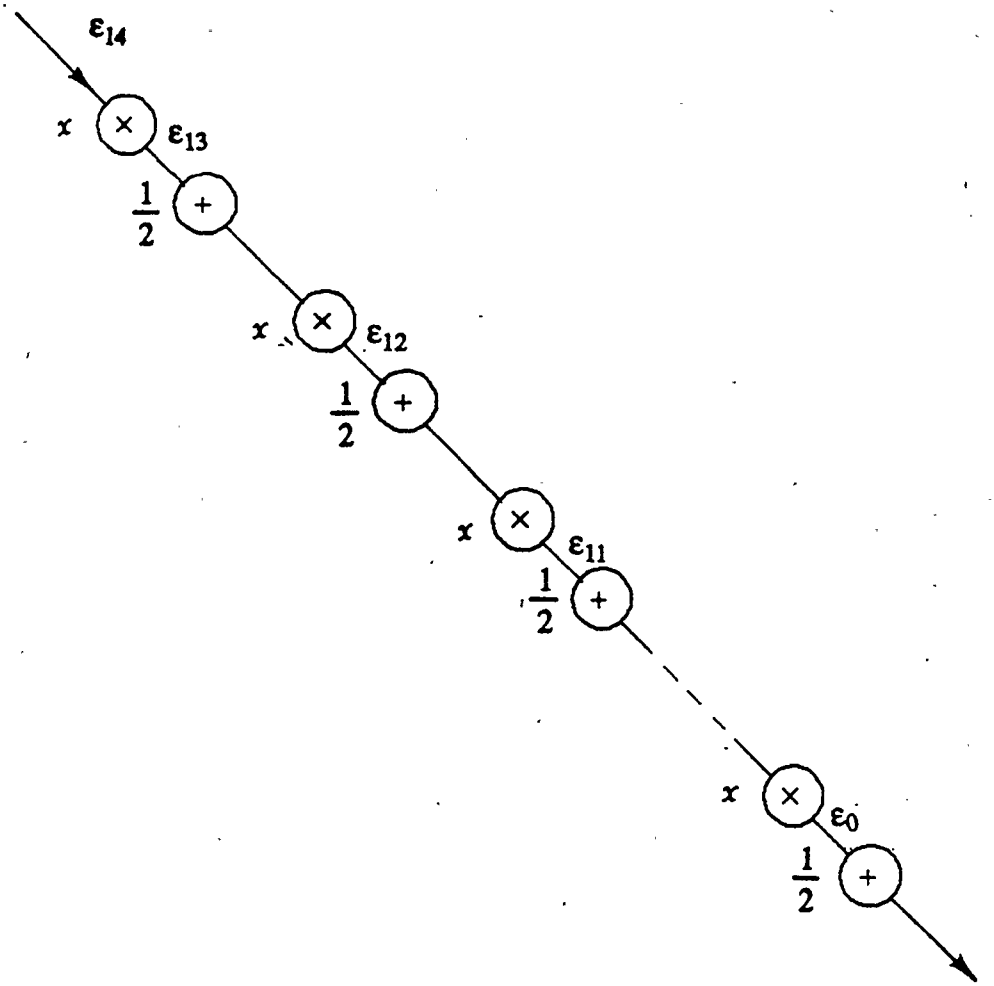


Figure 3.9 Roundoff Error in the Modified Taylor Series used in Determining $(1-x)^{-1}$

$$\mu_{wc} \approx 2\mu_m. \tag{3.38}$$

The variance was found to be

$$\sigma_{WC}^2 \approx \frac{4}{3}\sigma_m^2 \tag{3.39}$$

resulting in a *MSE* of:

$$MSE \approx 4\mu_m^2 + \frac{16}{9}\sigma_m^2 \tag{3.40}$$

The mean, mean squared error, and the variance of the modified Taylor series algorithm are shown in table 3.4. Though the last form of this algorithm requires a slightly longer time (60 cycles) it still provides a reasonable compromise between accuracy and speed.

3.6 OVERFLOW AND SCALING IN UPDATING THE PREDICTION ERRORS

Although overflow cannot be avoided in a number of stages in the Burg algorithm, proper scaling of the input data could remove the possibility of overflow in the lattice filter. This would speed up the software as the handling of overflows can be ignored. The goal of scaling is to remove the possibility of overflow, while maintaining the largest possible dynamic range for the input data. To do this the structure of the lattice must be examined to determine conditions under which an overflow might occur. The lattice update equations are:

$$e_{j,n} = e_{j-1,n} + a_{j,j}\ b_{j-1,n-1} \tag{3.41}$$

and

Table 3.3 Errors in the Unmodified Taylor Series Algorithm for Division

Scaling Method	Mean μ	Variance σ^2	Mean Square
Down Rounding	-13Δ	$\frac{11\Delta^2}{16}$	$\frac{1521\Delta^2}{9}$
Up Rounding	$\frac{\Delta}{4}$	$\frac{11\Delta^2}{9}$	$\frac{11\Delta^2}{9}$
Magnitude Truncation	$-\frac{13\Delta}{2}$	$\frac{77\Delta^2}{36}$	$\frac{60685\Delta^2}{1296}$
Random Bit Addition	$-\frac{13\Delta}{8}$	$\frac{77\Delta^2}{36}$	$\frac{60685\Delta^2}{1296}$

Table 3.4 Error Statistics for Modified Taylor Series Division

Scaling Method	Mean μ	Variance σ^2	Mean Square
Down Rounding	$-\frac{\Delta}{2}$	$\frac{\Delta^2}{9}$	$\frac{11\Delta^2}{9}$
Up Rounding	0	$\frac{\Delta^2}{9}$	$\frac{\Delta^2}{9}$
Magnitude Truncation	$-\frac{\Delta}{2}$	$\frac{7\Delta^2}{36}$	$\frac{4\Delta^2}{9}$
Random Bit Addition	$-\frac{\Delta}{2}$	$\frac{7\Delta^2}{36}$	$\frac{4\Delta^2}{9}$

$$b_{j,n} = b_{j-1,n-1} + a^*_{j,j}\, e_{j-1,n} \tag{3.42}$$

From these equations, the worst case for overflows occurs when the magnitudes of $e_{j-1,n}$, $b_{j-1,n-1}$, $a_{j,j}$ are close to unity. A resulting output value of two is then possible. It should be noted that the Burg algorithm attempts to minimize the error functions $e_{j-1,n}$, and $b_{j-1,n}$, and therefore the only gain that can occur will happen when $e_{j-1,n}$, $b_{j-1,n}$, are the actual data values. After the first update most of the prediction error values are theoretically reduced and there should be no fear of overflow. This indicates scaling the input data by 1/2 will remove the possibility of overflow. In a strict sense, this conclusion may not be valid if the input is not primarily AR in nature. In that case it might be necessary to scale the input data by more than two and incorporate overflow handling into the algorithm.

3.7 SUMMARY

The implications of BFP arithmetic have been examined in this chapter. A summation scheme that reduce the effect of roundoff errors. A high speed division scheme was developed that was fast and accurate. Scaling the input data by two ensured that no overflows occur in the lattice structure when the data was AR in nature.

CHAPTER 4
THE HARDWARE DESIGN OF THE PROCESSOR

4.0 INTRODUCTION

The Burg algorithm has been examined and found suitable for real time implementation. In designing an architecture that would permit real time operation a number of factors must be addressed. These factors include hardware type, configuration and the number of components. The goal of this chapter is to present a hardware design that will permit real time operation of the Burg algorithm.

4.1 HARDWARE TYPE

Two approaches can be taken when designing high speed digital signal processing (DSP) processors. A microprocessor based implementation or a microprogrammable bit slice system can be developed. A custom designed VLSI chip and the use of systolic array processors are considered to be beyond the scope of this thesis [11].

4.1.1 MICROPROCESSORS

A number of specialized microprocessors that perform digital signal processing operations are available. An example is the TMS32010 microprocessor [19] which can be described as state of the art in DSP microprocessors. This chip has an ALU, multiplier, shifter, and internal memory, and operates at a clock period of 200ns.

While providing many desirable features, the TMS32010 has a number of drawbacks symptomatic of all microprocessors. The preset instructions permit fast software development but limit the overall performance. Though some pipelining has

been incorporated to speed up certain operations, most instructions only use one part of the processor such as the ALU, while the other resources sit idle. This inefficient use of resources decreases the overall speed of the system.

Reading and writing from external memory also cause problems. There are no single cycle instructions that permit external memory to be loaded into the ALU. This will decrease the operational speed as all of the computational resources are idle. Though this is a specific problem related to the TMS32010 chip, it brings to light the I/O bottleneck associated with most microprocessors. The I/O bottleneck means that access to external data is slow, limiting the overall performance of the system. Campbell [5] showed that one processor is not sufficient for real time operation of the DSA and a multiprocessor approach would be needed. External memory that can be accessed by all of the processors is essential to this configuration. The I/O problems and the inefficient use of resources are two major drawbacks of microprocessors while fast development time is the major advantage of such systems.

4.1.2 MICROPROGRAMMABLE SYSTEMS

Unlike a microprocessor, a bit-slice microprogrammable system does not have a preset architecture or instruction set. This permits the designer to customize the hardware and software to the task at hand. With no preset structure, the I/O can be designed so that no bottlenecks exist. The great flexibility in designing the architecture means that each component has its own set of control signals. The control signals permit the use of parallel programming methods which efficiently use the resources.

All this power and flexibility makes the microprogram development difficult. Each of the microprogrammable components is controlled through its own instructions. The control unit must be capable of generating extremely long control words which can easily exceed 100 bits whereas 16 bit instructions are used in the TMS32010. Microprograms require a specialized development tool such as a meta-assembler [20] that can systematically generate the very long control words. A downloading unit and a control system [7,21] must be developed to handle the long control words present. The requirements for specialized program development tools and long control words are some of the drawbacks of these systems.

The number of chips in a microprogrammable implementation can be large. The flexibility of being able to determine the configuration means that many of the interconnections that are made in silicon in a microprocessor have to be manually connected by the designer, leading to longer development times.

In summary, most of the constraints of microprocessors are exhibited during run time while their main advantages are very fast development times and relatively few chips. Microprogrammable systems have faster run times while development time and chip count are usually higher than microprocessors. In selecting a hardware type, the main consideration in this thesis is operational speed. It was decided to use a microprogrammable system as it offered the best run time performance which is the main consideration in this design. The excessively long development times are shortened somewhat by the use of a meta-assembler, a downloading unit and a generalized micro-sequencer.

4.2 MICROPROGRAMMING CONCEPTS

A microprogrammed bit-slice architecture was selected as the hardware in this implementation. Microprogramming concepts are not commonplace and a brief examination of these concepts is given here.

4.2.1 MICROPROGRAM CONTROL

In a microprogrammable system, control is generally achieved by using a microprogram sequencer in conjunction with a microprogram memory and a pipeline register. The task of the microsequencer is to output an address to the microprogram memory (known as a control store (CS)) which in turn sends out control words to the rest of the system including the sequencer. The typical sequencer, shown in fig. 4.1, has several sources including a stack, a direct input, a program counter and a counter from which it can generate the CS address. The proper address source is selected depending on the instructions from the CS and a condition code (CC), that contains the status of the controlled system.

The ordered structure of a CS separates the microprogrammed system from most other control systems that use sequential logic techniques to implement control. The instructions are simply and easily changed by changing the contents of the CS. This high degree of flexibility is one advantage of microprogrammed systems. In the development system built by Orbay [7], static high speed RAM and start-up EPROMS were used to implement the CS.

The last element of the control unit, the pipeline register, provides a number of services. The overall control system, shown in fig. 4.2, the registers are clocked and

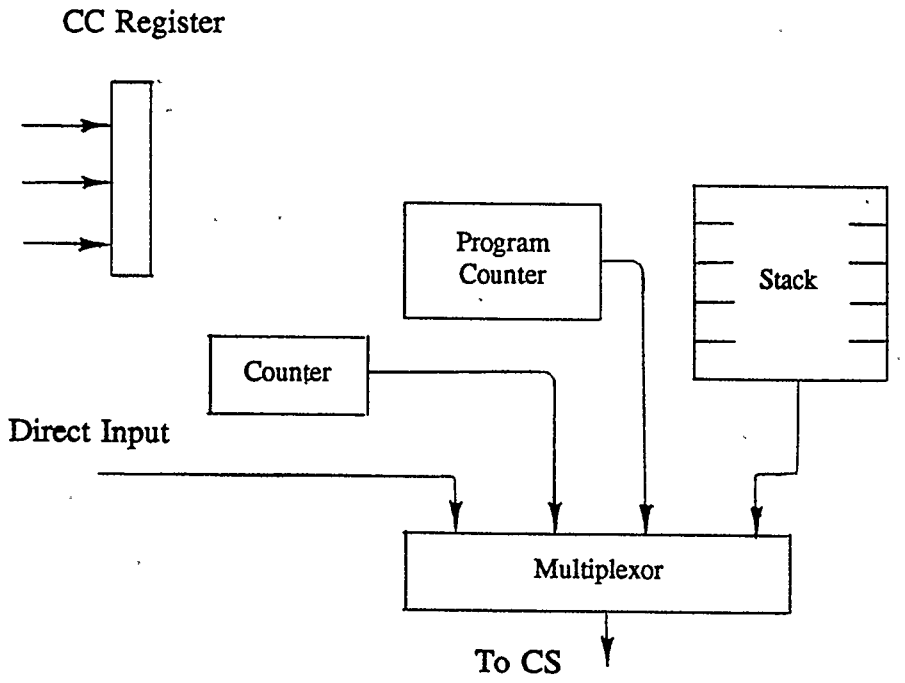


Figure 4.1 The Typical Components of a Microsequencer

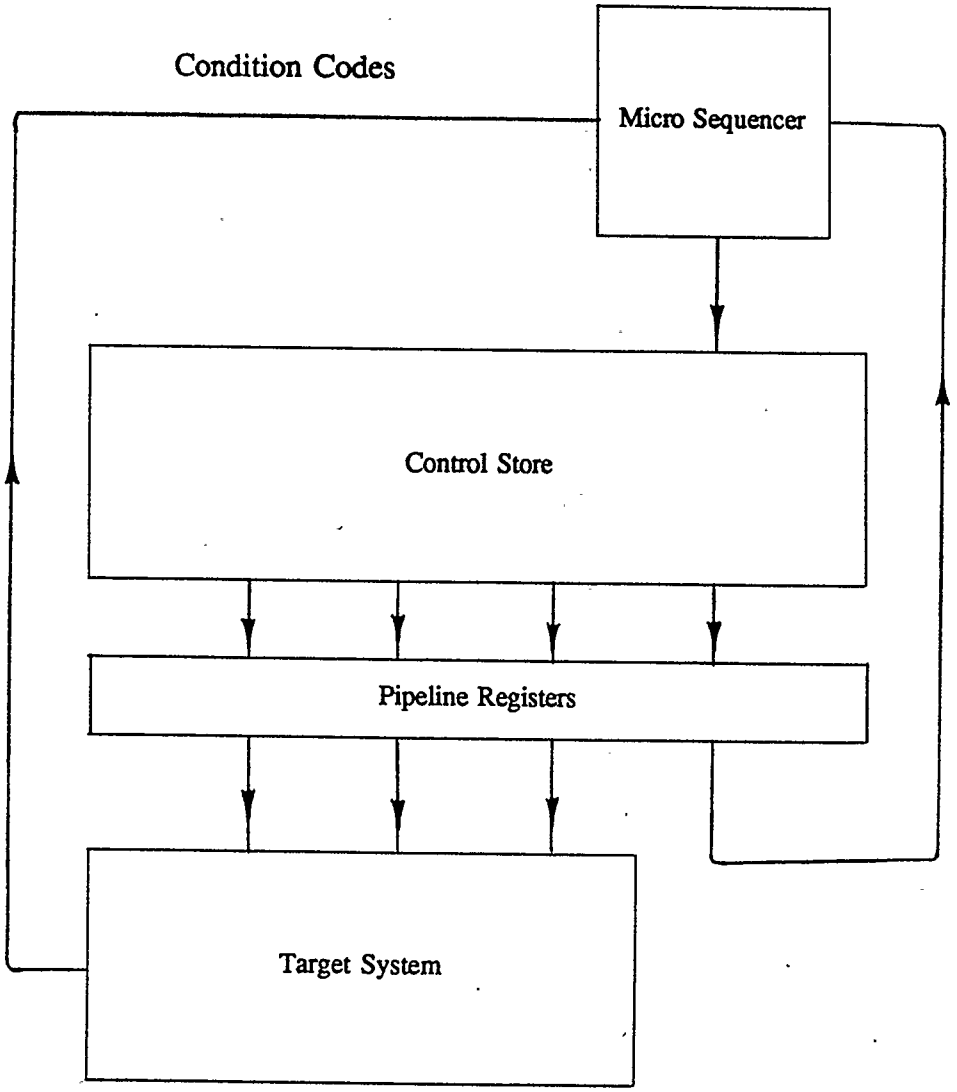


Figure 4.2 A Microprogram Control Unit
(For Simplicity the Clock Signals are not shown.)

provide a delay in the feedback loop, thereby removing any race conditions that might exist between the sequencer and the CS. The pipeline register also isolates the controller from the target system, thereby permitting the application of modular design concepts.

4.2.2 THE BIT-SLICE CONCEPT

Bit-slice design can be described as a "building block" approach to designing systems. The basic block is a slice of a computing element that is 4-bits wide. The units can be cascaded together to form a larger element that meets the design requirements. This approach gives the designer flexibility in selecting the appropriate word length required. Consider the task of addressing 4K of memory which requires a 12 bit address. Cascading 3 bit-slice ALUs, each 4 bits wide, would meet the addressing requirements. An extension of the bit-slice concept, the byte slice, has an 8-bit wide slice as the fundamental building block. Byte slice components were used extensively throughout the design of this processor.

4.2.3 PIPELINING AND PARALLEL PROGRAMMING

In the discussion of the control unit two advantages of pipelining were discussed. One advantage is the isolation of the control unit from the hardware processor thereby permitting the control unit to operate with a certain degree of independence. The same concept can be incorporated within the processor to increase throughput as shown in the following example. Fig. 4.3a shows an processor with no intermediate pipeline registers to hold the data that is passed between elements. As a result, only one operation can be performed at a given time and only one element can be functional at

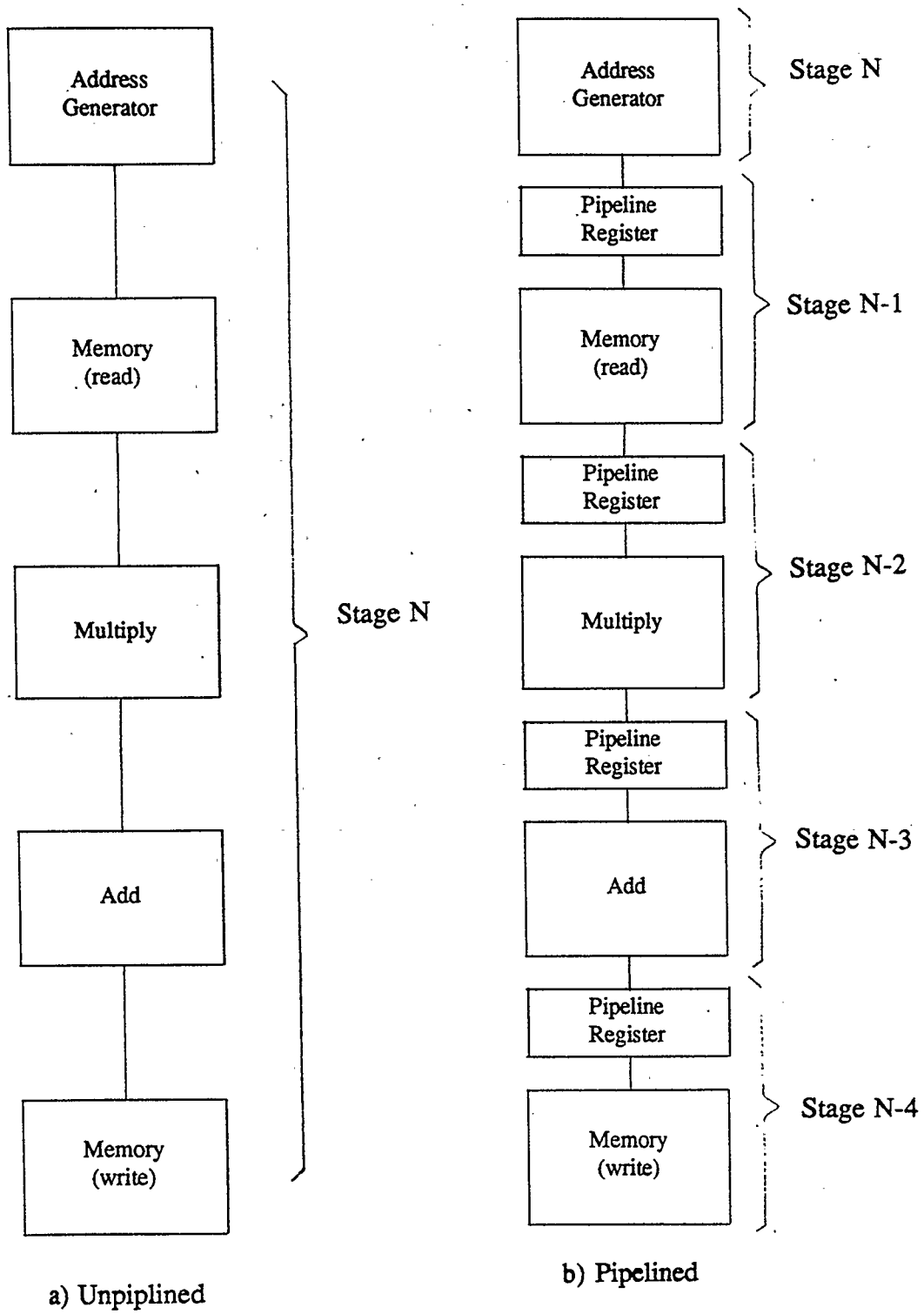


Figure 4.3 An Example of Pipelined and Unpipelined Architectures

a given stage of the operation. The introduction of pipeline registers, shown in fig. 4.3b, permits the elements to operated independently of each other. Pipelining implies that all of the processor elements are operating simultaneously.

To fully realize the advantage of pipelining, parallel programming techniques are applied. One aspect, particular to parallel programming, is the concept of overlapping instructions which is illustrated in fig. 4.4. Once the pipeline has been filled all resources are operating independently on part of the algorithm. The throughput of the system is increased as results are generated every cycle rather than every i^{th} cycle where i is the number of operations performed on the data. Digital signal processing algorithms are particularly amenable to parallel programming techniques because of the repetitive nature involved.

4.3 COMPUTATIONAL REQUIREMENTS OF THE BURG ALGORITHM

By examining eqns (2.16 - 2.24), the arithmetic operations in the Burg algorithm can be broken into the basic computational functions given below.

1) The complex multiply and add operation,

$$A = B + C \cdot D, \tag{4.1}$$

where A, B, C, D are complex variables, is used extensively in the lattice filter and the PEF coefficient computation. The only significant difference between the PEF and lattice stages lies in their addressing requirements. The design of the address generator is considered at a later stage.

Cycle	1	2	3	4	5	6	7
Address Generator	I_N	I_{N+1}	I_{N+2}	I_{N+3}	I_{N+4}	I_{N+5}	I_{N+6}
RAM	-	I_N	I_{N+1}	I_{N+2}	I_{N+3}	I_{N+4}	I_{N+5}
Multiply	-	-	I_N	I_{N+1}	I_{N+2}	I_{N+3}	I_{N+4}
Add	-	-	-	I_N	I_{N+1}	I_{N+2}	I_{N+3}
RAM	-	-	-	-	I_N	I_{N+1}	I_{N+2}

Loading Pipeline Fully Pipelined

Figure 4.4 The Overlapping of Instructions

2) A magnitude squared operation,

$$E = |A|^2 + |B|^2, \tag{4.2}$$

where E is a real variable occurs primarily in the formation of the denominator and is used in the computation of the MMSE.

3) A general summation

$$A = \sum_i B_i, \tag{4.3}$$

represents a fundamental block because of the specialized hardware needed to perform the address comparisons and the data shifts that occur in any large summation.

Division was not considered a basic function because it was performed using a Taylor series expansion of $(1 - x)^{-1}$ which is simply a combination of the three basic functions described above.

4.4 OPERATIONAL ELEMENTS

A number of computing elements must be combined to perform the basic functions of the Burg algorithm. These elements are listed below.

1) The *Memory Unit* stores the data. Random access memory (RAM) was used for this application.

2) An *Address Generator* is required to access the data in memory. The design of this unit is independent of the main processing unit and is handled later in the chapter.

2) The importance of *pipeline registers* in the operation of the APU has already been stated. As mentioned, they are extremely useful in permitting elements to operate independently and improve throughput by permitting the overlapping of instructions.

3) An *arithmetic logic unit ALU,* capable of performing addition, subtraction and a number of logic operations, is an important element in a processing unit. In addition to the basic arithmetic and logic functions, most ALU's also contain a number of internal registers. These registers are essentially pipeline registers that can be used as scratch pad memory to store values arising from intermediate calculations. The result is a reduced usage of the I/O ports avoiding the bottlenecks that would otherwise arise.

5) *Shifters* must be present to scale the data because the threat of overflow exists in a number of operations performed by the Burg algorithm. They are also essential when floating point notation is used to represent a number. This situation arises in the division stage of the algorithm and is discussed in chapter 5. One of the drawbacks of a bit slice implementation is that a barrel shifter capable of performing multiple shifts in a single operation, is not cascadable and cannot be included in a bit slice ALU.

6) A *Multiplier* is required in all the basic operations in the Burg algorithm except the summation. A hardware multiplier chip must be used to meet the requirement of real time operation.

7) In a multi-bus configuration, *multiplexers* should be used on the inputs of the multiplier to permit quick access to multiple sources of data. This is useful when forming the square of a number as both inputs of the multiplier come from the same source as opposed to a standard multiplication where the numbers come from different sources.

4.5 COMPONENT TECHNOLOGY

A number of microprogrammable components are commercially available. One company, Advanced Micro Devices (AMD) [18], has a full set of bit and byte-sliced microprogrammable chips. They have introduced one family of chips, the 29500 series, that is ideally suited to signal processing applications. These chips are fabricated using ECL technology for speed and TTL technology for external interfacing. Combining this fabrication process with a highly pipelined internal architecture has produced a set of chips that operate at a fast clock rate and have a high data throughput.

The AM29501 is a byte-slice ALU. In addition to the ALU, this chip contains 6 scratch pad registers, 2 unidirectional data ports (one input and one output) and a bidirectional data port. The multiplier chip, the AM29517, is a high speed 16 bit multiplier. The internal pipelining of this chip permits it to output a product every clock cycle. AMD also provides a number of support chips such as shifters, bus

drivers and high speed memory.

4.5.1 WORDLENGTH

A number of factors must be considered when determining a wordlength for the processor, The wordlength should be large enough to accurately represent the final answer without significant roundoff error. As the the wordlength grows, the number of byte slice components must also grow. A good compromise between hardware complexity and numerical accuracy is a 16 bit wordlength. This wordlength should be able to represent the input data which is acquired via an 8 to 12 bit A/D and the number of hardware components is not excessive.

4.6 COMPONENT QUANTITY

In any design there exists a trade-off between high speed operation and system complexity. In order to optimize this trade-off, the tasks that the processor has to perform must be known. Eqns (4.1-4.3) showed the basic functional blocks required to perform the Burg algorithm. The most significant block is:

$$A = B + C \cdot D \tag{4.1}$$

The hardware design should proceed with this function in mind. Expanding eqn(4.1) into real and imaginary parts yields the following expressions:

$$A_{RE} = B_{RE} + C_{RE} \cdot D_{RE} - C_{IM} \cdot D_{IM} \tag{4.4}$$

$$A_{IM} = B_{IM} + C_{IM} \cdot D_{RE} + C_{RE} \cdot D_{IM} \tag{4.5}$$

where RE represents a real component and IM represents an imaginary component.

Assuming that the scratch pad registers hold all the intermediate values, there are 8 I/O operations. Four addition/subtraction operations along with 4 multiplications must be performed on the data by the arithmetic processing unit (APU). With only one data bus, one ALU, and a multiplier operating in parallel the computation takes 8 cycles with the limiting resource with the I/O data busses. The number of resources can be increased to improve speed. The decrease in the number of cycles required to perform eqn(4.1) as the number of hardware components is increased is detailed in table 4.1.

In performing this analysis it must be kept in mind that some combinations of resources do not result in any real savings in time. An example of this is the 2 bus, 1 ALU, 1 multiplier (2-1-1) configuration. Here the two data busses are loading one ALU. Even though it will only take the two data busses 4 cycles to load the necessary information, it will still take the ALU 8 cycles to read the data because it only has one I/O port which communicates with the busses. The additional bus only becomes effective when a second ALU is added. The possibility exists that one I/O bus can be attached to the multiplier. However if a data value is to be used more than once, it must be held in a scratch pad register which is located in the ALU chip. This implies that values read into the multiplier would still have to be read into the ALU defeating any gain in speed that was achieved by connecting the bus to the multiplier.

4.6.1 THE 2-2-1 CONFIGURATION

A configuration containing 2 data busses, 2 ALUs and one multiplier appears to offer the best trade-off between speed and complexity in the design of this proto-type

Table 4.1 The Effect of Increasing the Components

Components					
I/O Data Busses		ALU		Multiplier	
#	cycles	#	cycles	#	cycles
1	8	1	4	1	4
2	4	1	4	1	4
2	4	2	2	1	4
2	4	2	2	2	2
4	2	2	2	2	2
4	2	4	1	2	2
4	2	4	1	4	1
8	1	4	1	4	1

processor [18]. The twofold increase in speed that would be gained by using a 4-4-2 does not justify the fourfold increase in hardware. A 2-2-2 configuration appeared to offer an increase in speed with only moderate increase in hardware. Upon closer examination, it was found that the additional multiplier could not be utilized efficiently because of insufficient ALU resources and the apparent gains in speed were not realized. Two data busses and two ALUs seem to be the natural form for an APU that deals with complex numbers because these numbers are comprised of two components, a real and imaginary part. Separating the busses and ALUs into two units, one for the real and imaginary components allows the interconnections between busses and ALUs to be minimized. The real and imaginary ALUs and busses can operate independently with no need for direct interconnections.

Placing the shifters between the data busses and the ALUs permits the data to be shifted without disturbing the normal flow of the data. The multiplier can be connected through multiplexers to the ALUs and does not need to be connected to the data busses as the ALUs can supply the multiplier with input data and receive its output. This setup reduces the complexity by removing connections between the multiplier and data bus that might be otherwise be needed. The overall organization of the APU is shown in fig. 4.5.

4.6.2 HARDWARE ROUNDING

Fig. 4.5 includes a rounding control block that has not been discussed. Roundoff errors exist after every multiplication. If unattended, these errors can grow and cause the numerator to become greater than the denominator which results in a reflection

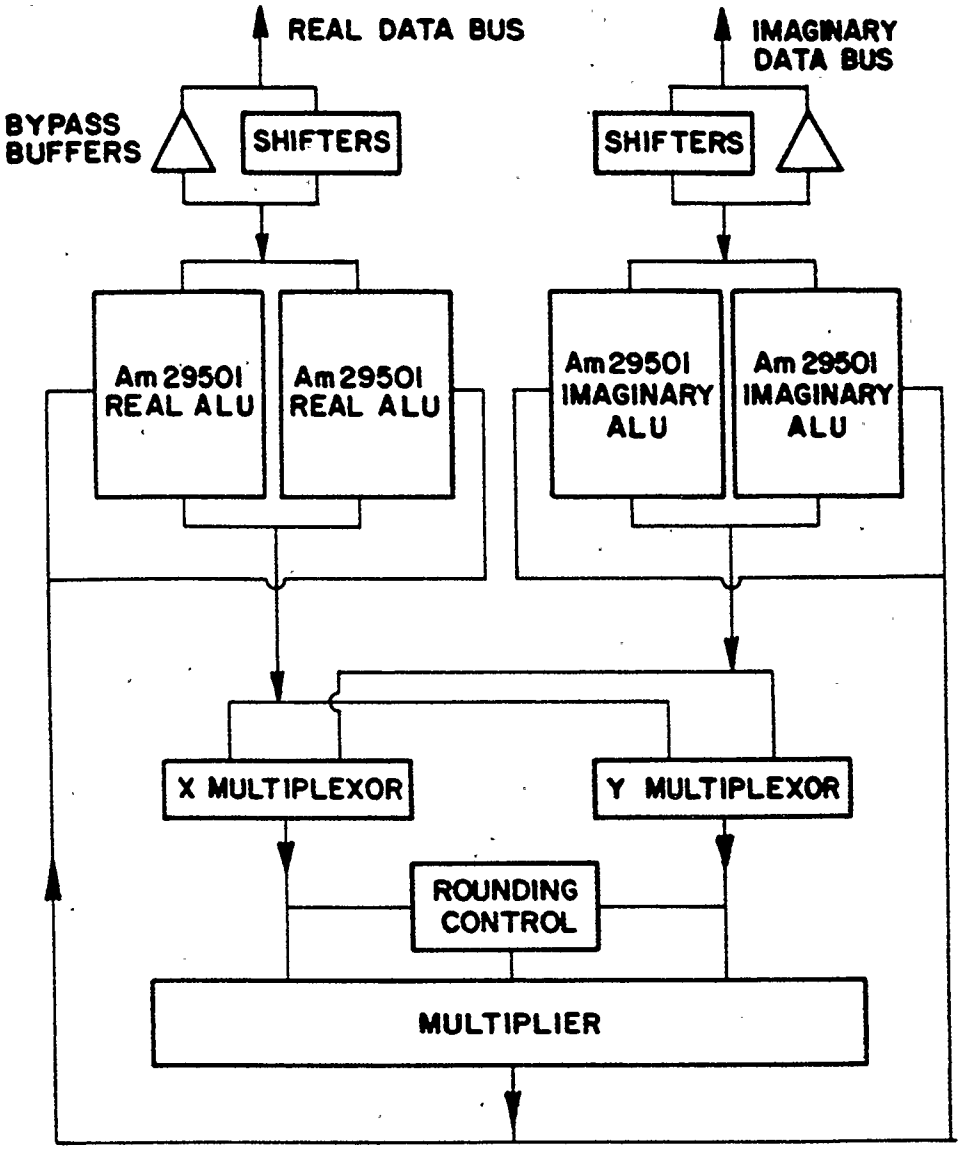


Figure 4.5 A Block Diagram of the APU Hardware

coefficient, $a_{j,j}$, greater than the theoretical maximum. Thus the Burg algorithm has become unstable due to roundoff error. This situation arises when most of the numerator products are negative. Ignoring the lower 16 bits of the multiplier output can viewed as subtracting a positive quantity from the multiplier product. When the product is negative this subtraction increases the magnitude of the number and it is then possible for a sum of negative numbers to increase while a sum of positive numbers is decreased relative to their respective theoretical values. To alleviate this problem magnitude truncation for the numerator and up rounding for the denominator are applied. A hardware rounding unit is used to speed up the rounding involved in each multiplication. This unit precalculates the value of the rounding bit by examining the sign bits of the multiplicands. The relationship between the input sign bits and the rounding bit needed for magnitude truncation is an Exclusive OR operation with the result being added to the LSB. In view of this, the hardware consists of an Exclusive OR chip with associated peripheral and a control line that, when asserted, would override the magnitude truncation scheme and insert an uprounding bit.

4.7 ADDRESSING REQUIREMENTS

The addressing in the Burg algorithm is generally quite simple. The updating of the prediction errors simply requires a pair of counters capable of down-counting. However, the tree algorithm produces a complicated set of addressing requirements that a simple counter cannot fulfill due to the shifting involved. The need to be able to re-address memory in the event of an overflow means that the selected addressing device should have internal memory to store previous values. In light of these

requirements what is needed is a device that can add, subtract, shift, and store values.

The solution can be provided by an ALU, such as the AM2901 which is a microprogrammable bit-slice ALU. It performs most standard ALU functions, and contains 16 internal registers, 3 external ports and shifting capability. The size of the data memory to be accessed must be known before the number of AM2901s can be determined. AR modeling is normally applied to short data records and it is felt that 2K of RAM for both the real and imaginary data blocks would be sufficient. This requires 11 bits of addressing, meaning that 3 AM2901s must be used for the addressing. One problem with these chips is that the output port comes directly from the ALU part of the chip. The timing for the complex addressing sequences becomes rather difficult if the output of the ALU is tied directly to the data RAM. Feeding the ALU output into a two sets of specialized pipeline registers (AM29520s), which contain their own set of internal registers from which the RAM address can be selected, provides an excellent solution to the problem and also permits independent addressing of the two data blocks. Fig. 4.6 shows the block diagram of the address generator and the memory unit. A comparator is incorporated to provide the high speed address comparisons that are required in some parts of the Burg algorithm.

4.8 EXTERNAL INTERFACING

The final task in the processor design is to provide an external interface unit to permit the APU to communicate to devices such as A/Ds and D/As, other processors, and other systems. Each of these applications has its own set of interface requirements meaning that the interface unit must be flexible enough to handle the various

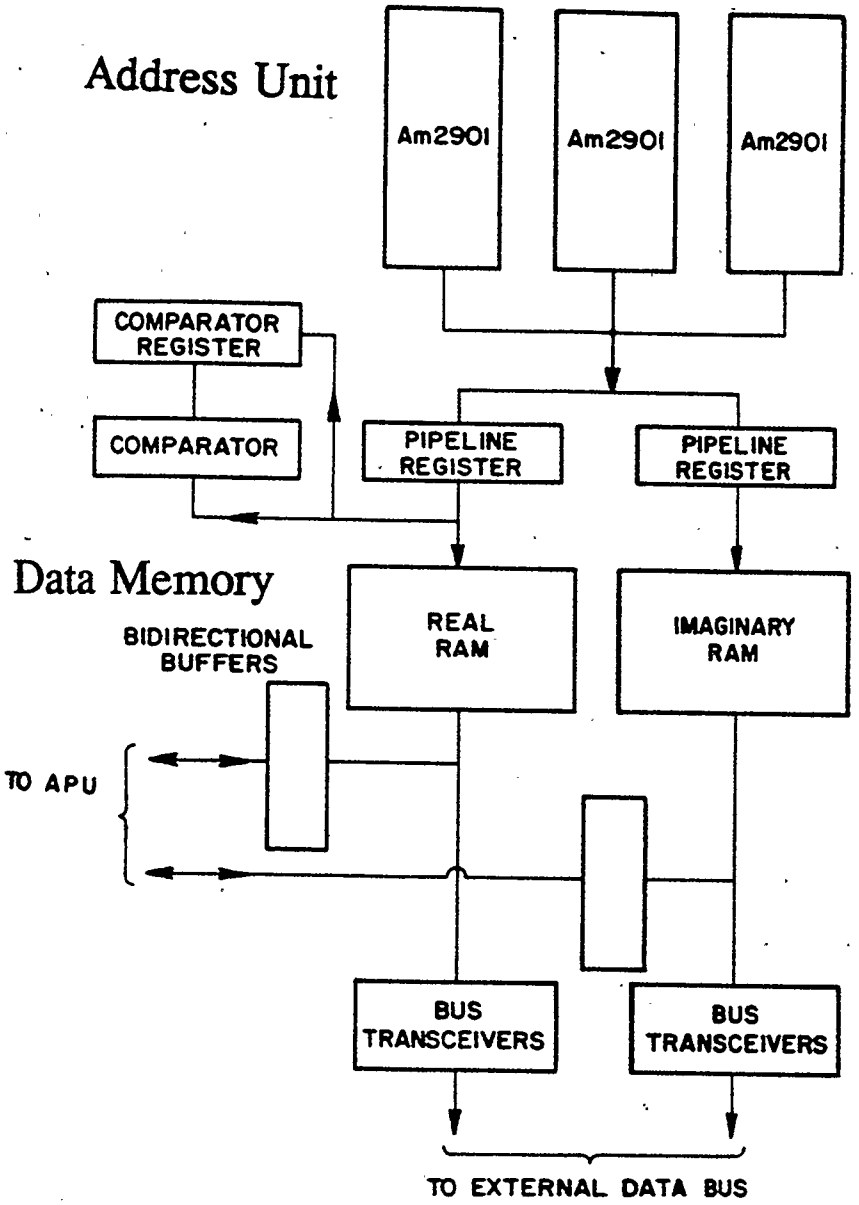


Figure 4.6 The Block Diagram of the Addressing Unit

requirements.

The interface is composed of two bus transceivers that consist of an input and output data register and a flip-flop associated with each register. The flip-flops are used to provide the handshaking between the APU and the external device. When a device loads a register with data, it sets that register's flip-flop indicating the data is ready. When the receiving device reads the data, it clears the flip-flop, thereby telling the sending device that the buffer is empty. This system is implemented with AM2950s which are transceivers with 8 bit registers and an associated flip-flop.

4.9 SUMMARY

A design of a microprogrammable architecture capable of running in real time has been presented. A number of microprogramming concepts have been reviewed and a family of microprogrammable chips described. The overall architecture, given in fig. 4.7, contains two data busses, two ALUs, a multiplier, address generator, I/O ports, RAM and a number of support chips.

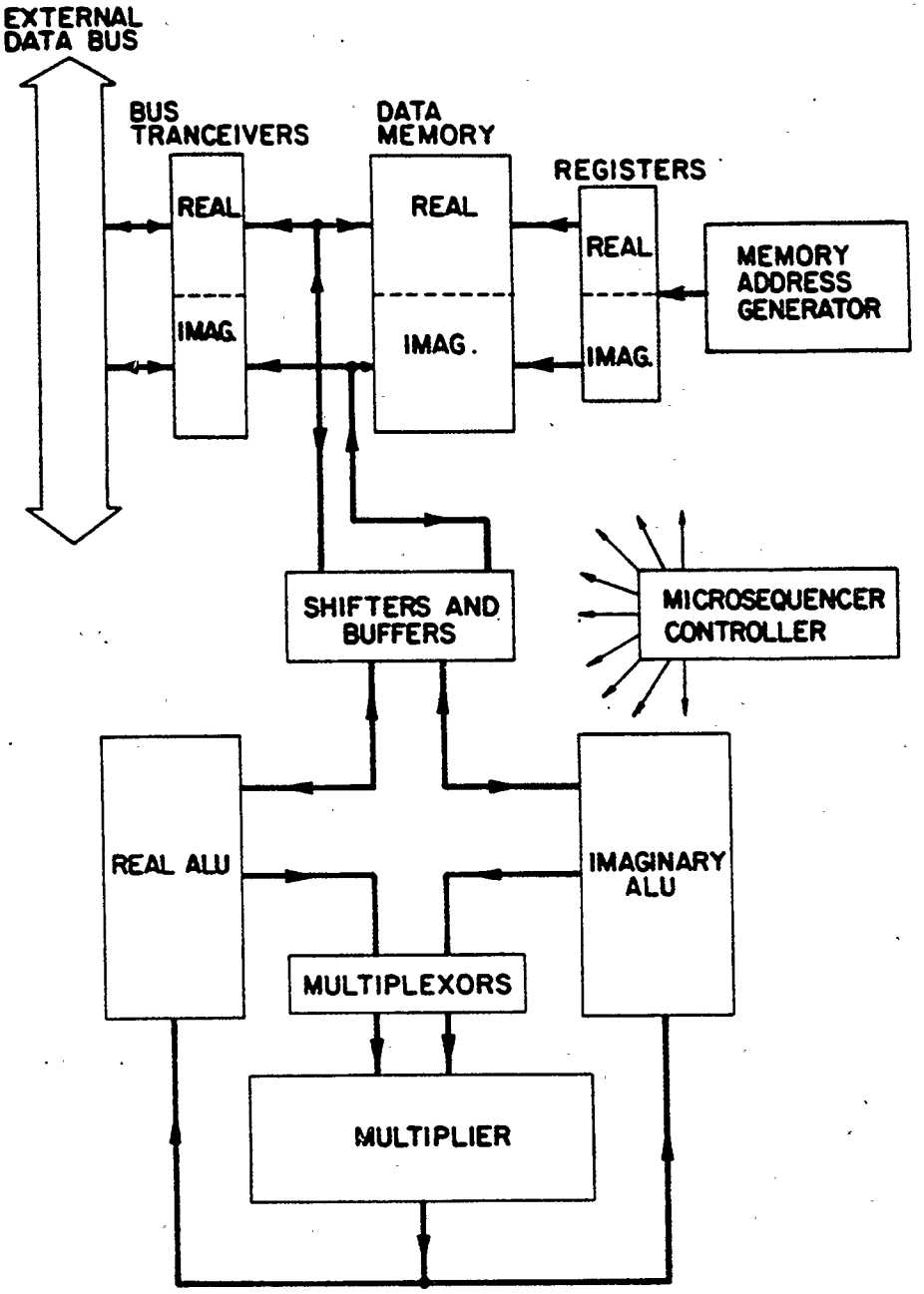


Figure 4.7 A Schematic of the Overall Configuration

CHAPTER 5
MICROPROGRAMMABLE IMPLEMENTATION OF THE BURG ALGORITHM

5.0 INTRODUCTION

Having proposed a hardware configuration capable of meeting the requirements of the Burg algorithm, the next task is to develop the microprogram code for the Burg algorithm. A modular approach is taken in developing the microcode for this algorithm. To demonstrate possible real time operation, the theoretical run time of the Burg algorithm must be determined. This involves determining the maximum clock rate of the hardware and the time required to perform the arithmetic operations in the Burg algorithm.

5.1 PARTITIONING OF THE BURG ALGORITHM

To reduce the programming complexity, the Burg algorithm can be separated into several subsections which can be independently developed, tested and implemented. This approach might lead to a slightly slower implementation of the Burg algorithm because the initialization stages within each module might not make the best use of the resources available. However, this small decrease in speed is compensated by the large decrease in development time.

Examining eqns (2.16-2.24) shows that the algorithm can be separated into the following stages:

- formation of the denominator,
- formation of the numerator,

- division and computation of the reflection coefficient and computation of the *MMSE*,
- updating the prediction errors,
- computing the prediction error filter coefficients.

In this chapter the microcode required for the implementation of each stage will be discussed.

5.2 FORMATION OF THE DENOMINATOR

Expanding the denominator (eqn(2.19)) into real and imaginary components yields:

$$D_p = \sum_{i=p}^{N-1} [\,(e_{re}(i))^2 + (b_{re}(i-1))^2 + (e_{im}(i))^2 + (b_{im}(i-1))^2]. \tag{5.1}$$

Initially, the errors are the actual data points, $e(n), b(n) = x(n)$. In the implementation eqn(5.1) is broken into two smaller pieces, a squaring section followed a tree summation stage. The resulting summation was then normalized.

The squaring section formed partial sums a_{re}, a_{im} in the real and imaginary ALUs respectively. These sums consisted of 4 values that had been squared

$$a_{re}(j) = (e_{re}(i))^2 + (e_{re}(i-1))^2 + (b_{re}(i-1))^2 + (b_{re}(i-2))^2 \tag{5.2}$$

and

$$a_{im}(j) = (e_{im}(i))^2 + (e_{im}(i-1))^2 + (b_{im}(i-1))^2 + (b_{im}(i-2))^2 \tag{5.3}$$

where $i = N-1 \cdots p$ and $j = 0 \cdots int((N-p)/2)$.

As the input data is scaled by 1/2, no overflow will occur in this stage as the squared values are less than 1/4. The fact that this stage forms partial sums means that it reduces the overall time taken to perform the tree addition by a factor of two. Up rounding was used when performing the multiplications in this stage. The pseudo code for this operation is shown in fig. 5.1. The fully pipelined microcode, given in fig. 5.2, shows the operation of each element of the APU during the different cycles. It should be noted that the intermediate values, t,u,v,w,c,d, are stored in scratch pad memory. The addressing and control aspects of the microprogram are not shown in this figure but can be found in an internal departmental report [22].

5.3 THE TREE ALGORITHM

It was shown in chapter 3 that the tree addition algorithm produces an accurate fixed point summation. The pseudo code for this algorithm is shown in fig. 5.3. The tree algorithm is basically sequential and the microcode is equivalent to the pseudo-code.

This algorithm incorporates two interesting techniques to handle the addressing and overflows. By using an additional index counter and a comparator, it avoids the need to balance the tree and thereby reduces the run time of the summation. The use of the comparator provides single cycle address comparisons and reduce the time taken to perform the tree addition by about 25%. Overflows must also be considered in any addition scheme. Campbell [5] used a tree algorithm that simply scaled every result and thereby avoided the overflow problem. This approach removes the overflow problem at the cost of reduced accuracy. The loss of precision results when there are

DENOMINATOR

```
begin

  count = N - 1 - order;   (* SET UP COUNTER, ADDRESS POINTER *)
  tree_count = -1;                  (* AND TREE POINTER *)
  addr = N - 1;

  while( count != 0 )
  begin
    re_sum = sqr(re_ep[addr]) + sqr(re_del[addr-1]);  (* HALF OF EQN 5.2 *)
    im_sum = sqr(im_ep[addr]) + sqr(im_del[addr-1]);  (* HALF OF EQN 5.3 *)

    count = count - 1;              (* DECREMENT POINTERS *)
    addr = addr - 1;

    if( count != 0 ) then
    begin

      re_sum = sqr(re_ep[addr]) + sqr(re_del[addr]) + re_sum; (* REST OF EQN 5.2 *)
      im_sum = sqr(im_ep[addr]) + sqr(im_del[addr]) + im_sum; (* REST OF EQN 5.3 *)

      tree_count = tree_count + 1;  (* PREPARE FOR TREE ADDITION *)
      re_tree_data[ tree_count ] = re_sum;
      im_tree_data[ tree_count ] = im_sum;

      count = count - 1;    (* DECREMENT POINTERS *)
      addr = addr - 1;

    end
    else
    begin

      tree_count = tree_count + 1;  (* PREPARE FOR TREE ADDITION *)
      re_tree_data[ tree_count ] = re_sum;
      im_tree_data[ tree_count ] = im_sum;

    end
  end

  return()
end
```

Figure 5.1 Pseudo Code for the Denominator

	Busses		Real	Imaginary	Multiplier
	Real	Imag	ALU	ALU	$X \cdot Y$
read	e_{re}^{i}	e_{im}^{i}	$a_{re}(j-1) = d_{re} + w_{re}$		$w_{im} = b_{im}^{i} \cdot b_{im}^{i}$
read	b_{re}^{i-1}	b_{im}^{i-1}		$a_{im}(j-1) = d_{im} + w_{re}$	$t_{re} = e_{re}^{i} \cdot e_{re}^{i}$
write	$a_{re}(j-1)$	$a_{im}(j-1)$			$t_{im} = e_{im}^{i} \cdot e_{im}^{i}$
read	e_{re}^{i-1}	e_{im}^{i-1}			$u_{re} = b_{re}^{i-1} \cdot b_{re}^{i-1}$
read	b_{re}^{i-2}	b_{im}^{i-2}	$c_{re} = t_{re} + u_{re}$		$u_{im} = b_{im}^{i-1} \cdot b_{im}^{i-1}$
				$c_{im} = t_{im} + u_{im}$	$v_{re} = e_{re}^{i-1} \cdot e_{re}^{i-1}$
			$d_{re} = c_{re} + v_{re}$		$v_{im} = e_{im}^{i-1} \cdot e_{im}^{i-1}$
				$d_{im} = c_{im} + v_{im}$	$w_{re} = b_{re}^{i-2} \cdot b_{re}^{i-2}$

Figure 5.2 The Resource Usage in the Fully Pipelined Denominator Stage

TREE ADDITION

```
begin

 tree_shift = 0;            (* INITIALIZE COUNTER *)

 while (tree_count > 0)
 begin
  i = 0;
  j = 0;

  while( i <= tree_count )
  begin
   re_sum[j] = re_tree_data[i] + re_tree_data[i+1];  (* PERFORM ADDITIONS *)
   im_sum[j] = im_tree_data[i] + im_tree_data[i+1];

   if( overflow == TRUE)        (* IF OVERFLOW OCCURS *)
   begin
    tree_shift = tree_shift + 1;  (* INCREMENT OVERFLOW COUNTER *)
    stage2 = j;                   (* SAVE LOCATION OF OVERFLOW *)

    while ( i < tree_count )
    begin
     re_sum[j] = re_tree_data[i]/2 + re_tree_data[i+1]/2; (* ADD WITH A SCALING BY 1/2 *)
     im_sum[j] = im_tree_data[i]/2 + im_time_data[i+1]/2;

     i = i + 2;
     j = j + 1;

     if( i == tree_count )
     begin
      re_sum[j] = re_tree_data[i]/2;     (* HANDLE THE POSSIBILITY OF AN ODD *)
      im_sum[j] = im_time_data[i]/2;     (* NUMBER OF DATA POINTS *)
     end
    end

    tree_count = j - 1
    i = 0;
    j = 0;

    begin                          (* SECOND PASS AFTER OVERFLOW *)

     re_sum[j] = re_tree_data[i]/2 + re_tree_data[i+1]/2;
     im_sum[j] = im_tree_data[i]/2 + im_tree_data[i+1]/2;

     i = i + 2;
     j = j + 1;

     if( i > stage2 )
     begin
      re_sum[j-1] = re_tree_data[i-2]/2;
      im_sum[j-1] = im_tree_data[i-2]/2;
      i = stage2;
```

```
        end
      end
      j = j - 1;  (* PREPARE TO RE-ENTER MAIN TREE ADDITION ALGORITHM *)
    end

    if( i == tree_count )

    begin
      re_sum[j] = re_tree_data[i];
      im_sum[j] = im_tree_data[i];
    end
  end
  tree_count = j - 1;
 end
 return();
end
```

Figure 5.3 Pseudo Code for Tree Summation

stages in the tree algorithm where no overflows occur but scaling is performed.

To retain the maximum accuracy, an algorithm was developed that only shifted the data after an overflow occurred. The algorithm checked the overflow flag after every addition. When an overflow occurred, the addition causing the overflow was repeated after its input values were scaled by 1/2. For all the remaining additions in current stage of the tree algorithm the input data was scaled by 1/2 before being added. This ensures that no further overflows will occur in that stage. Instead of repeating the additions that were performed prior to the overflow, the algorithm proceeded to the next stage of the tree and performs the necessary scaling during that stage. The operation of the algorithm is shown in fig. 5.4. A shift counter is incremented to keep track of the number of overflows. When rounding after an overflow has occurred, up rounding was used in the denominator and magnitude truncation was used in the numerator to ensure stability.

After the summation is completed the values are stored in a floating point format. That is:

$$\sum x_i = r\ 2^n \tag{5.4}$$

where $1 > r \geq 1/2$ and n is the shift count related to the number of overflows that occurred during the summations.

5.4 COMPUTATION OF THE NUMERATOR

The numerator is described by:

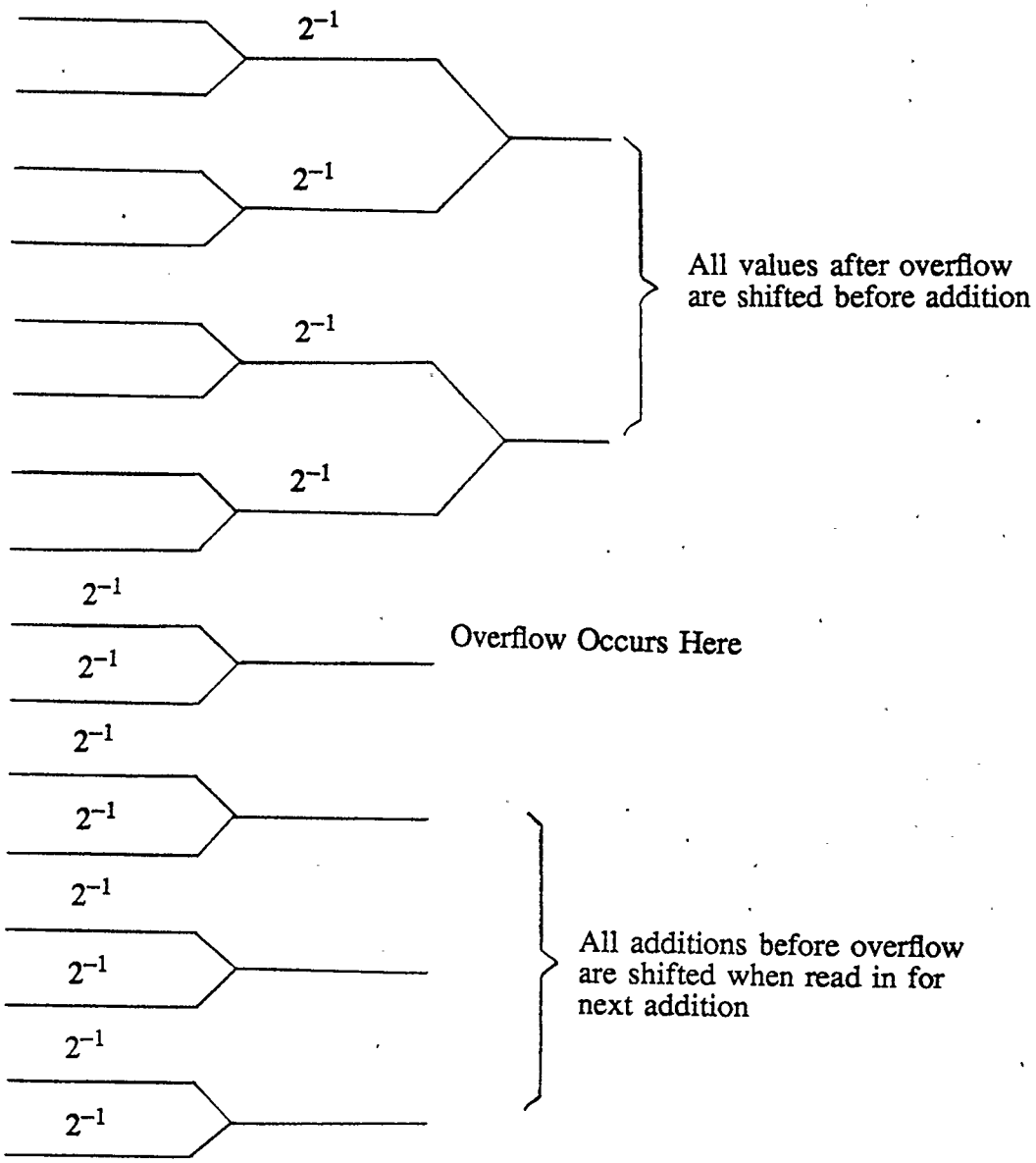


Figure 5.4 Operation of the Tree Algorithm when an Overflow Occurs.

$$N_p = \sum_{i=p}^{N-1} [\, e_{re}(i) \cdot b_{re}(i-1) + e_{im}(i) \cdot b_{im}(i-1) \qquad (5.5)$$

$$+ e_{re}(i) \cdot b_{im}(i-1) - e_{im}(i) \cdot b_{re}(i-1) \,]$$

As with the denominator implementation, this stage was broken into two stages, conjugate multiplication and tree addition. The equations for the partial sums involved in the conjugate multiplication stage are:

$$a_{re}(j) = e_{re}(i) \cdot b_{re}(i-1) + e_{re}(i-1) \cdot b_{re}(i-2) \qquad (5.6)$$

$$+ e_{im}(i) \cdot b_{im}(i-1) + e_{im}(i-1) \cdot b_{im}(i-2)$$

$$a_{im}(j) = e_{re}(i) \cdot b_{im}(i-1) + e_{re}(i-1) \cdot b_{im}(i-2) \qquad (5.7)$$

$$- e_{im}(i) \cdot b_{re}(i-1) - e_{im}(i-1)) \cdot b_{re}(i-2)$$

where $i = N-1 \;\cdots\; p$ and $j = 0 \;\cdots\; int((N-p)/2)$.

The basic programming differences in the numerator and denominator calculations are that the numerator required conjugate multiplication and employed magnitude truncation whereas the denominator performed squaring and up rounding was used. The pseudo-code for this stage is shown in fig. 5.5 with the microcode in fig. 5.6.

5.5 DETERMINATION OF THE REFLECTION COEFFICIENT AND THE MMSE

Having formed the denominator and numerator, the reflection coefficient can be found which involves a division of the form

$$a_{ii} = \frac{-2\, N_i\; 2^{nshift}}{D_i\; 2^{dshift}}. \qquad (5.8)$$

Simplifying yields:

NUMERATOR

```
begin

  count = N - 1 - order;   (* SET UP NECESSARY POINTERS *)
  tree_count = -1;
  addr = N - 1;

  while( count != 0 )
  begin
    re_sum = re_ep[addr]*re_del[addr-1] + im_ep[addr]*im_del[addr-1];
                                        (* HALF OF EQN 5.6 *)
    im_sum = re_ep[addr]*im_del[addr-1] - re_ep[addr]*im_del[addr-1];
                                        (* HALF OF EQN 5.7 *)

    count = count - 1;      (* DECREMENT POINTERS *)
    addr = addr - 1;

    if ( count != 0 )
    begin
      re_sum = re_ep[addr]*re_del[addr-1] + im_ep[addr]*im_del[addr-1]
             + re_sum;      (* REST OF EQN 5.6 *)
      im_sum = re_ep[addr]*im_del[addr-1] - re_ep[addr]*im_del[addr-1]
             + im_sum;      (* REST OF EQN 5.7 *)

      count = count - 1;    (* DECREMENT POINTERS *)
      addr = addr - 1;

      tree_count = tree_count + 1;  (* PREPARE FOR TREE SUMMATION *)
      re_tree_data[ tree_count ] = re_sum;
      im_tree_data[ tree_count ] = im_sum;

    end
    else
    begin

      tree_count = tree_count + 1;  (* PREPARE FOR TREE SUMMATION *)
      re_tree_data[ tree_count ] = re_sum;
      im_tree_data[ tree_count ] = im_sum;

    end
  end

  return()
end
```

Figure 5.5 Pseudo Code for the Numerator Stage

	Busses		Real	Imaginary	Multiplier
	Real	Imag	ALU	ALU	$X \cdot Y$
read	e_{re}^{i}	b_{im}^{i-1}		$d_{im} = w_{im} + v_{im}$	$w_{re} = e_{im}^{i+1} \cdot b_{im}^{i}$
read	b_{re}^{i-1}	e_{im}^{i}	$d_{re} = v_{re} + w_{re}$	$a_{im}(j-1) = d_{im} + c_{im}$	$t_{im} = e_{re}^{i} \cdot b_{im_{i-1}}$
			$a_{re}(j-1) = d_{re} + c_{re}$		$t_{re} = e_{re}^{i} \cdot b_{re}^{i-1}$
write	$a_{re}(j-1)$	$a_{im}(j-1)$			$u_{im} = b_{re}^{i-1} \cdot e_{im}^{i}$
read	e_{re}^{i-1}	b_{im}^{i-2}		$c_{im} = u_{im} - t_{im}$	$u_{re} = e_{im}^{i} \cdot b_{im}^{i-1}$
read	b_{re}^{i-2}	e_{im}^{i-1}	$c_{re} = u_{re} + t_{re}$		$v_{im} = e_{re}^{i-1} \cdot b_{im}^{i-2}$
					$v_{re} = e_{re}^{i-1} \cdot b_{re}^{i-2}$
					$w_{im} = b_{re}^{i-2} \cdot e_{im}^{i-1}$

Figure 5.6 The Resource Usage in the Fully Pipelined Numerator Stage

$$a_{ii} = -\frac{N_i}{D_i} 2^{-(dshift - nshift - 1)} \tag{5.9}$$

where $1 > N_i$, and $D_i \geq 1/2$. The scaling factors for the numerator and denominator are $nshift$ and $dshift$ respectively.

The Taylor series expansion method discussed in chapter 3 was used to perform the division. A simple shifting program was written to evaluate the difference in the scaling factors. Magnitude truncation was used in performing this part of algorithm. This algorithm was essentially sequential in nature and the microprogram effectively follows the pseudo-code, shown in fig. 5.7.

The $MMSE$ of order j was calculated using

$$MMSE_j = (1 - (a_{re}^{jj})^2 - (a_{im}^{jj})^2) \cdot MMSE_{j-1} \tag{5.8}$$

It was a relatively straightforward task to perform this operation and the corresponding pseudo code is given in fig. 5.8.

5.6 UPDATING THE PREDICTION ERRORS

The lattice structure shown in fig. 2.6 is used by the Burg algorithm to update the prediction errors. This structure is described by the following equations:

$$e_{re}(i) = e_{re}(i) + a_{re}^{jj} \cdot b_{re}(i-1) - a_{im}^{jj} \cdot b_{im}(i-1) \tag{5.10}$$

$$e_{im}(i) = e_{im}(i) + a_{re}^{jj} \cdot b_{im}(i-1) + a_{im}^{jj} \cdot b_{re}(i-1) \tag{5.11}$$

$$b_{re}(i) = b_{re}(i-1) + a_{re}^{jj} \cdot e_{re}(i) + a_{im}^{jj} \cdot e_{im}(i) \tag{5.12}$$

$$b_{im}(i) = b_{im}(i-1) + a_{re}^{jj} \cdot e_{im}(i) - a_{im}^{jj} \cdot e_{re}(i) \tag{5.13}$$

Implementation of these equations is straightforward as no overflow problem will be

DIVISION

```
begin
   x = 1 - denom;                    (* x = 1 - B *)
   z = 1/2 + x/2;          (* EQN 3.35 *)

   for ( k = 0 ; k < 14; ++k)
     z = 1/2 + x*z;                  (* EQN 3.34 *)

   shift_adj = dshift - nshift - 1;

   for ( i = shift_adj; i >= 0; --i)
    z = z/2;              (* ADJUST FOR SHIFT DIFFERENCES *)

   return();
end
```

Figure 5.7 Pseudo Code for Taylor Series Division

MMSE CALCULATION

```
begin

  mmse = mmse*(1 - re_aii*re_aii - im_aii*im_aii);   (* EQN 5.8 *)

  return();
end
```

Figure 5.8 Pseudo Code for MMSE Computation

encountered due to the prescaling of the data. Starting at the end of the data (i.e. $i = N - 1$) and working towards the start of the data ($i = p$) permits the errors to be updated in place. The pseudo code for the lattice filter is shown in fig. 5.9 and the microcode is shown in fig. 5.10.

5.7 DETERMINATION OF THE PREDICTION ERROR FILTER COEFFICIENTS

Computation of the PEF coefficients is done by using the following equations:

$$a_{re}^{i,j} = a_{re}^{i,j-1} + a_{re}^{j,j} \cdot a_{re}^{j-i,j-1} + a_{im}^{j,j} \cdot a_{re}^{j-i,j-1} \tag{5.14}$$

$$a_{im}^{i,j} = a_{im}^{i,j-1} + a_{re}^{j,j} \cdot a_{im}^{j-i,j-1} - a_{im}^{j,j} \cdot a_{re}^{j-i,j-1} \tag{5.15}$$

In order to reduce the number of I/O operations and to perform the computations in place, the computation of the real and imaginary components of $a_{i,j}$ and $a_{j-i,j}$ were performed simultaneously via the Levinson butterfly. The relationship between these two terms is shown in fig. 5.11. Examining this figure indicates that the computation of the PEF will be similar to the calculation involved in the lattice structure. The differences lie in the addressing and the fact that an overflow can occur. A block floating format of number representation is used in this part of the Burg algorithm to accommodate overflows. The pseudo code for the PEF is given in fig. 5.12 and the microcode is given in fig. 5.13.

5.8 RUN TIME EQUATIONS

Having proposed a possible implementation, the next task is to theoretically determine the maximum speed attainable with this architecture. The task can be

LATTICE

```
begin

  count = N - 1 - order;
  addr = N - 1;

  while( count >= 0)
  begin
    re_eptemp = re_ep[addr] + re_aii*re_del[addr-1] - im_aii*im_del[addr-1];
                                        (* EQN 5.10 *)
    im_eptemp = im_ep[addr] + re_aii*im_del[addr-1] + im_aii*re_del[addr-1];
                                        (* EQN 5.11 *)
    re_del[addr] = re_del[addr-1] + re_aii*re_ep[addr] + im_aii*im_ep[addr];
                                        (* EQN 5.12 *)
    im_del[addr] = im_del[addr-1] + re_aii*im_ep[addr] - im_aii*re_ep[addr];
                                        (* EQN 5.13 *)

    re_ep[addr] = re_eptemp;
    im_ep[addr] = im_eptemp;

    count = count - 1;
    addr = addr - 1;
  end
  return()
end
```

Figure 5.9 Pseudo Code for Updating the Predictions Errors

	Busses		Real	Imaginary	Multiplier
	Real	Imag	ALU	ALU	$X \cdot Y$
read	a_{re}^{ii}	a_{im}^{ii}		$d_{im} = b_{im}^{i-2} - v_{im}$	$w_{re} = a_{im}^{ii} \cdot e_{im}^{i-1}$
read	b_{re}^{i-1}	b_{im}^{i-1}	$b_{re}^{i-2} = d_{re} + v_{re}$		$w_{im} = a_{re}^{ii} \cdot e_{im}^{i-1}$
read	e_{re}^{i}	e_{im}^{i}		$b_{im}^{i-2} = d_{im} + w_{im}$	$t_{im} = a_{re}^{ii} \cdot b_{im}^{i-1}$
write	b_{re}^{i-2}	b_{im}^{i-2}			$t_{re} = a_{im}^{ii} \cdot b_{im}^{i-1}$
				$c_{im} = e_{im}^{i} + t_{im}$	$u_{im} = a_{im}^{ii} \cdot b_{re}^{i-1}$
			$c_{re} = e_{re}^{i} - t_{re}$	$e_{im}^{i} = c_{im} + u_{im}$	$u_{re} = a_{re}^{ii} \cdot b_{re}^{i-1}$
			$e_{re}^{i} = c_{re} + u_{re}$		$v_{re} = a_{re}^{ii} \cdot e_{re}^{i}$
write	e_{re}^{i}	e_{im}^{i}	$d_{im} = b_{im}^{i} + v_{re}$		$v_{im} = a_{im}^{ii} \cdot e_{re}^{i}$

Figure 5.10 The Resource Usage in the Fully Pipelined Lattice Stage

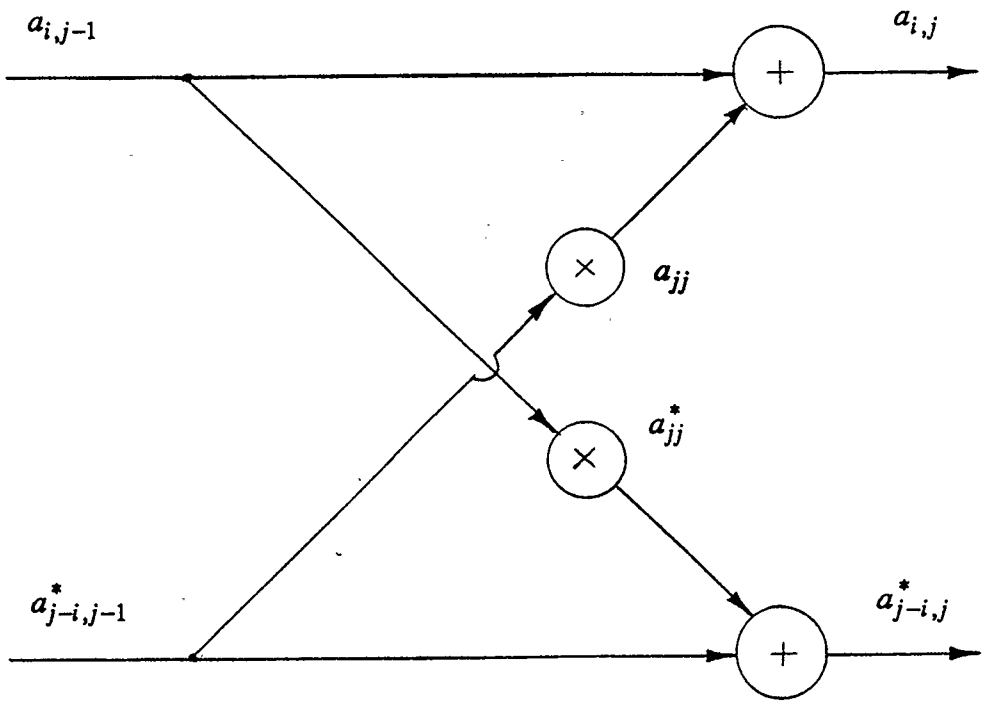


Figure 5.11 The Schematic Representation of the Levinson Butterfly

PEF

```
begin

 i = order/2;
 if( i == 0) return();
 j = order - i;

 while ( i > 0 )
 begin

  re_tempi = re_aij[i] + re_aii*re_aij[j] + im_aii*im_aij[j]; (* EQN 5.14 *)
  im_tempi = im_aij[i] + re_aii*im_aij[j] + im_aii*re_aij[j]; (* EQN 5.15 *)
  re_tempj = re_aij[j] + re_aii*re_aij[i] + im_aii*im_aij[i];
                                  (* EQN 5.14 for j-i coefficient *)
  im_tempj = im_aij[j] + re_aii*im_aij[i] - im_aii*re_aij[i];
                                  (* EQN 5.15 for j-i coefficient *)

  if( overflow == TRUE)
  begin
   pef_shift = pef_shift + 1;   (* INCREMENT OVERFLOW COUNTER *)

   for( k = 1; k < i ; ++k)     (* SCALE DOWN ALL PREVIOUSLY COMPUTED COEFFICIENTS *)
   begin
    re_aij[k] = re_aij[k]/2;
    im_aij[k] = im_aij[k]/2;
    re_aij[order - k] = re_aij[order - k]/2;
    im_aij[order - k] = im_aij[order - k]/2;
   end

   while ( i > 0 )

(* FOR ALL REMAINING COEFFICIENTS COMPUTE WITH SHIFT OF 1/2 *)

   begin
    re_tempi = re_aij[i]/2 + re_aii*re_aij[j]/2 + im_aii*im_aij[j]/2; (* EQN 5.14 *)
    im_tempi = im_aij[i]/2 + re_aii*im_aij[j]/2 + im_aii*re_aij[j]/2; (* EQN 5.15 *)
    re_tempj = re_aij[j]/2 + re_aii*re_aij[i]/2 + im_aii*im_aij[i]/2; (* EQN 5.14 *)
    im_tempj = im_aij[j]/2 + re_aii*im_aij[i]/2 - im_aii*re_aij[i]/2; (* EQN 5.15 *)

    re_aij[i] = re_tempi;              (* TRANSFER VALUES INTO COEFFICIENT ARRAY *)
    im_aij[i] = im_tempi;
    re_aij[j] = re_tempj;
    im_aij[j] = im_tempj;
    i = i - 1;
    j = order - i;

   end
  end

  re_aij[i] = re_tempi;   (* TRANSFER VALUE INTO COEFFICIENT ARRAY *)
  im_aij[i] = im_tempi;
  re_aij[j] = re_tempj;
```

```
    im_aij[j] = im_tempj;
  end

  for( i = 1; i = pef_shift; ++i)
(* SCALE REFLECTION COEFFICIENT BY CORRECT AMOUNT *)

  begin
    re_aii = re_aii/2
    im_aii = im_aii/2
  end

  re_aij[order] = re_aii;
  im_aij[order] = im_aii;

  return()
end
```

Figure 5.12 Pseudo Code for PEF Coefficients

	Busses		Real	Imaginary	Multiplier
	Real	Imag	ALU	ALU	$X \cdot Y$
read	a_{re}^{j-i}	a_{im}^{j-i}	$d_{re} = v_{re} + w_{re}$		$w_{im} = a_{im}^{i-1} \cdot s_{re}^{ii}$
read	a_{re}^{i}	a_{im}^{i}		$d_{re} = v_{re} - w_{re}$	$t_{im} = a_{re}^{j-i} \cdot a_{im}^{ii}$
			$a_{re}^{j-i+1} = a_{re}^{j-i+1} + d_{im}$	$a_{im}^{j-i+1} = a_{im}^{j-i+1} + d_{im}$	$t_{re} = a_{im}^{j-i} \cdot a_{im}^{ii}$
					$u_{im} = a_{im}^{j-i} \cdot a_{re}^{ii}$
write	a_{re}^{i-1}	a_{im}^{i-1}		$c_{im} = u_{im} - t_{im}$	$u_{re} = a_{re}^{j-i} \cdot a_{re}^{ii}$
write	a_{re}^{j-i+1}	a_{im}^{j-i+1}	$c_{re} = u_{re} + t_{re}$		$v_{re} = a_{re}^{i} \cdot a_{re}^{ii}$
					$v_{im} = a_{re}^{i} \cdot a_{im}^{ii}$
			$a_{im}^{i} = a_{re}^{i} + c_{re}$	$a_{im}^{i} = a_{im}^{i} + c_{im}$	$w_{re} = a_{im}^{i} \cdot a_{im}^{ii}$

Figure 5.13 The Resource Usage in the Fully Pipelined PEF Stage

broken into two areas, determining the clock speed and finding out how many clock cycles are required for each stage of the algorithm.

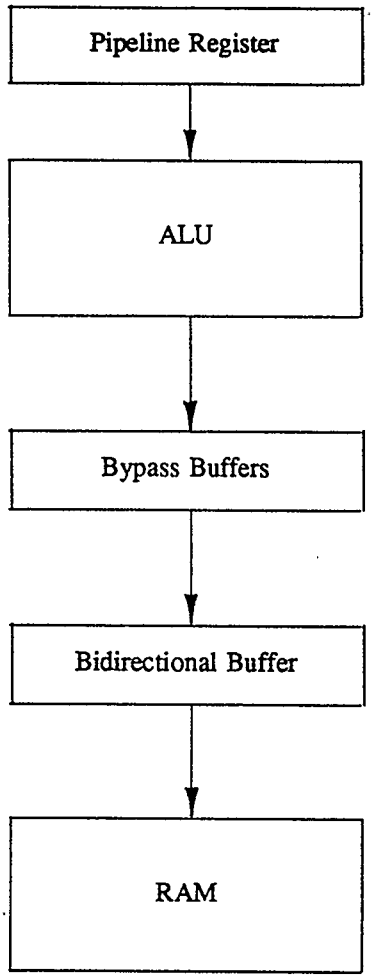
5.8.1 DETERMINATION OF THE CRITICAL PATH

The maximum operating clock speed is governed by the propagation delay in any given data or control path. The path that has the most propagation delay is known as the critical path. Fig. 5.14. shows the two candidates for the critical path in this hardware. The timing analysis, shown in table 5.1, indicates that the data path is the critical path and that 138 ns are required for the data to be written to the RAM. This means the clock speed is limited to 7.25 MHz.

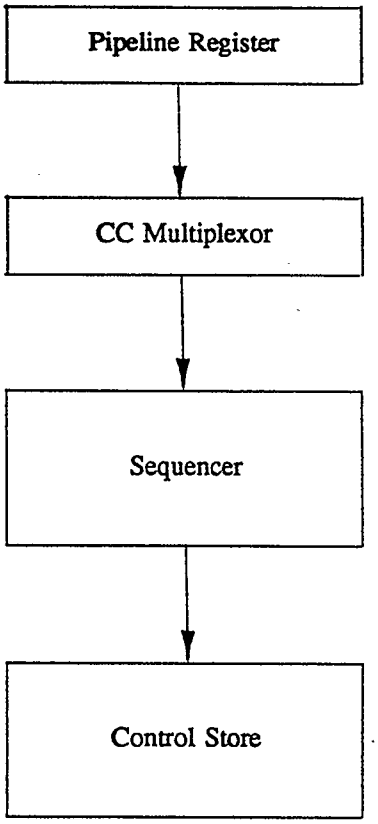
5.8.2 RUN TIME OF THE BURG ALGORITHM

The run time equation of the Burg algorithm can be found so that the operational speed can be determined. This analysis is performed by determining the critical resource for each stage and the number of cycles needed for that resource to complete the operation. Each stage of the Burg algorithm consists of three basic parts when it comes to determining the run time. In each subroutine there is an initialization phase where parameters used in that stage are set. As the pipeline takes one sequence of instructions to load and another to unload, the first and last instructions take additional cycles as they are not fully pipelined. Finally there is the fully pipelined stage of the instruction.

Table 5.2 shows the cycles taken to perform the above stages for each subroutine. It should be noted that most subroutines do not have a fixed cycle time. For example there are two main paths in the tree algorithm and which path is chosen depends on



a) Data Path



b) Control Path

Figure 5.14 The Control and Data Paths.

Table 5.1 Timing Analysis of the Control and Data Paths

Path	Device	Action	Component	Time
Control Path	Pipeline	clk to output	74LS374	20 ns
	CC multiplexor	select to output	74157	25 ns
	Sequencer	CC to output	Am2910	30 ns
	CS	Addr. to output	Am9150	25 ns
	Total			100 ns
Data Path	Pipeline	clk to output	74LS374	20 ns
	ALU	Register to output	Am29501	21 ns
	Bypass Buffer	OE to output	74LS241	25 ns
	Bidirectional Buffer	input to output	74LS241	12 ns
	Ram	Write Pulse	Am9128	60 ns
	Total			138 ns

Table 5.2: The Run Times of the Various Stages of the Burg Algorithm

Function	T_{init}	T_{part}	T_{pipe}
DENOMINATOR	6p	11p	$4Np - 2p^2 - 10p$
NUMERATOR	8p	12p	$4Np - 2p^2 - 10p$
LATTICE	5p	24p	$8Np - 4p^2 - 20p$
TREE ADDITION (Denominator)			
min	4p		$7mp/2 + 3Np/2 - 3p^2/2$
max	4p		$21mp/2 + 7Np/2 - 7p^2/2$
TREE ADDITION (Numerator)			
min	4p		$7mp/2 + 3Np/2 - 3p^2/2$
max	4p		$21mp/2 + 9Np/2 - 9p^2/2$
PEF			
min	6p	14p	$2p^2 - 8p$
max	6p	14p	$3p^2$
MISC. (Division, normalizations, MMSE, initializations, etc.)			
min			23 + 128p
max			23 + 233p

whether or not an overflow occurs. In this case a minimum time (no overflows) and a maximum time (overflows in every stage) were derived.

The total run time of the Burg algorithm can now be determined. This is done by adding all of the subroutines in table 5.2. It should be noted that the minimum run time for the miscellaneous section was used in both the maximum and minimum run time equations. The maximum run time of the miscellaneous section physically corresponds to the case were the data values are very small and all other run time sections are a minimum.

$$T_{Burg\,\max} = (182p + 24Np - 13p^2 + 21mp + 23)T_{clock} \quad (5.16)$$

$$T_{Burg\,\min} = (174p + 19Np - 9p^2 + 7mp + 23)T_{clock} \quad (5.17)$$

where p is the model order, N is the number of data points and $m = \log_2(N)$. Using eqn(5.16) and the theoretical clock speed of 7.25 MHz, a 16^{th} order model using 64 complex data points can be computed in $3615\ \mu s$. This translates to a theoretical worst case sampling rate of 17.7 kHz which is acceptable for real time operation.

5.9 SUMMARY

The microprogramming requirements of the Burg algorithm were examined in this chapter. A modular approach towards implementing the algorithm produced an implementation that was relatively simple. The maximum clock speed was determined using critical path analysis and the run time equations the Burg algorithm were developed. The real time operation of the processor was shown to be feasible.

CHAPTER 6
RESULTS AND CONCLUSIONS

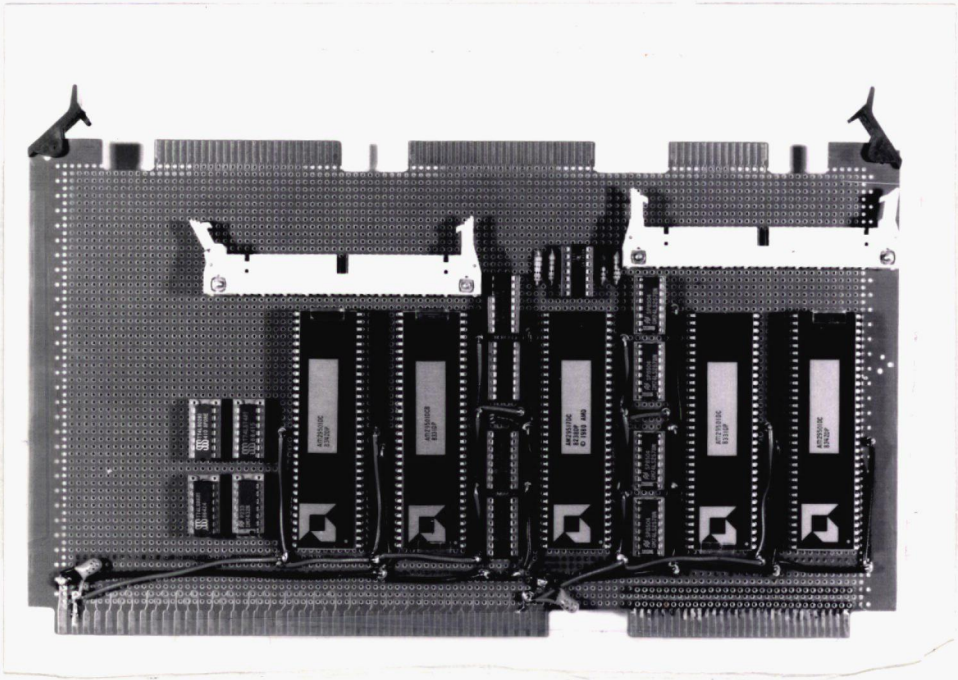
6.0 INTRODUCTION

The hardware discussed in chapter 4 was built using wire-wrap technology and is shown in fig. 6.1. The microprogramming discussed in chapter 5 was written with the aid of a meta-assembler and tested through a downloading unit. The performance of the hardware and the microprogramming are analyzed in this chapter. The overall accuracy, maximum experimental clock speed and the run time performance are discussed. Finally, recommendations for areas of improvement and future development are suggested.

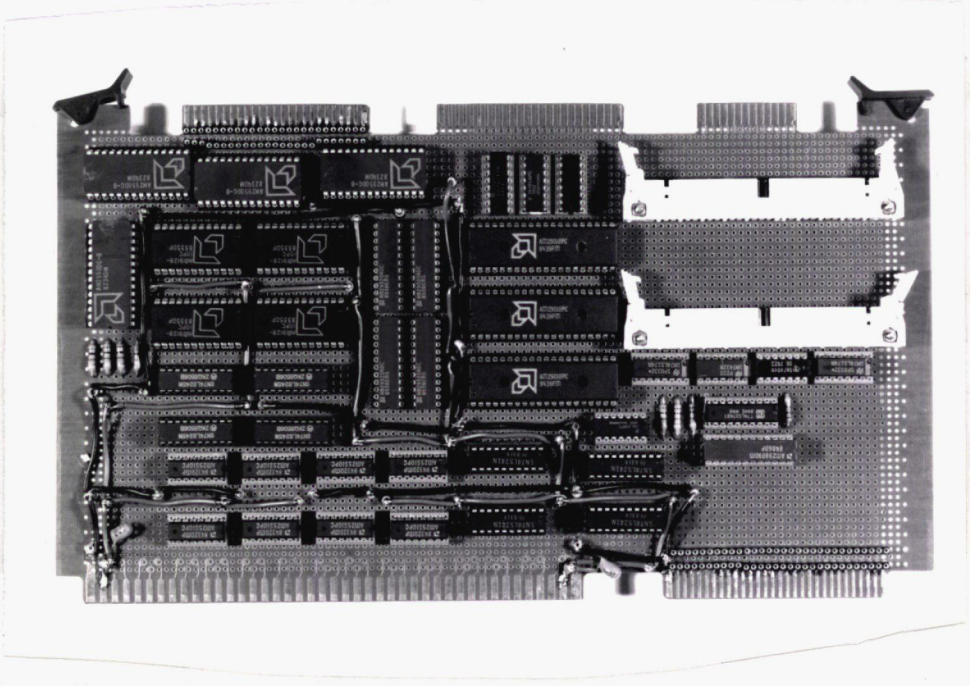
6.1 OVERALL ACCURACY

When dealing with fixed point numbers a certain degree of roundoff error is encountered. The effects of this error as it applied to certain areas of the implementation were described in chapter 3. The overall effect of roundoff error was not examined in detail as the error is dependent on the data. In lieu of a theoretical roundoff error analysis for the complete algorithm, a comparison between a floating point and the fixed point Burg algorithm was conducted. A floating point algorithm was written in Fortran 77 and run on the research VAX750 in the department. The fixed point algorithm has been discussed in chapter 5.

Two test signals were used, one real and one complex. The real test signal is described by the following equation:



A) A Photograph of the APU Board (Excluding Shifters)



B) A Photograph of the Memory Board and Shifters

Figure 6.1 Photographs of the Actual Hardware

$$x(n) = \cos(2\pi(0.25627)n) + \cos(2\pi(0.26877)n) + v(n) \tag{6.1}$$

where $v(n)$ is Gaussian white noise with an RMS amplitude such that the signal to noise ratio was 20 dB. The complex test signal consisted of the sum of 8 complex exponentials:

$$s(n) = \sum_{i=1}^{8} e^{j\omega_i n} \tag{6.2}$$

Figure 6.2 shows the pole locations of the complex exponentials used in this test signal. Noise was not included in this test so that the effects of roundoff errors could be determined.

Fig. 6.3 shows the spectral estimates obtained by applying applying a 16^{th} order model to 128 data points of the real test signal. A 16^{th} order model was used due to the close spacing of the peaks and the presence of noise. Clearly the BFP algorithm performs well when compared to the FLP algorithm. Fig. 6.4 shows the results when 64 data points of the complex signal were modeled with an 8^{th} order Burg model. Again the BFP algorithm compared favorably to the FLP algorithm demonstrating that the hardware implementation is accurate. The small differences that are present can be attributed to the roundoff error in the BFP implementation.

6.2 ROUND-OFF ERROR IN THE PREDICTION ERRORS

Having examined the spectral estimation, attention was directed towards the roundoff error present in the algorithm itself. A good measure of these errors can be determined by finding the mean and variance of the error in the prediction errors. The fact that the prediction errors are used in every stage of the Burg algorithm and are

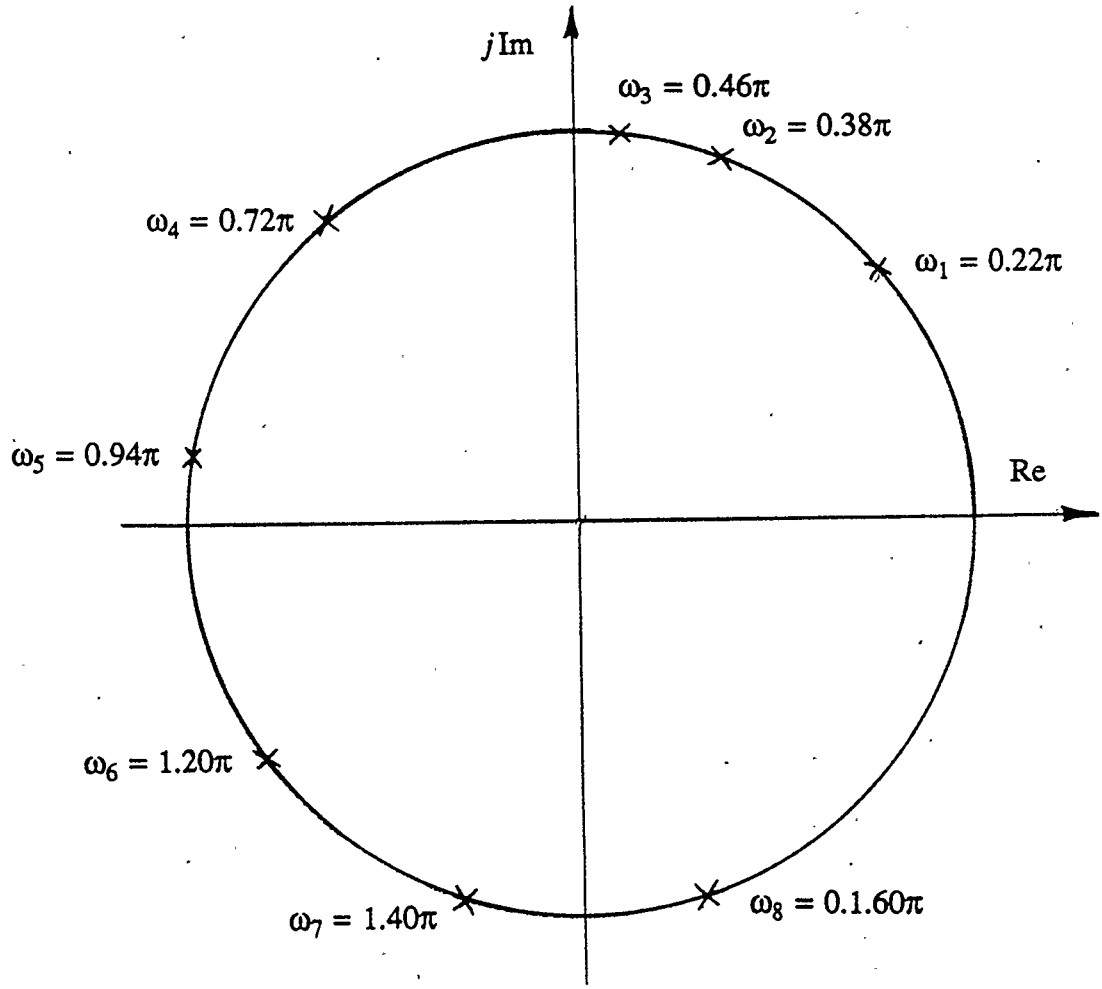


Figure 6.2 Pole Locations of the Complex Exponentials

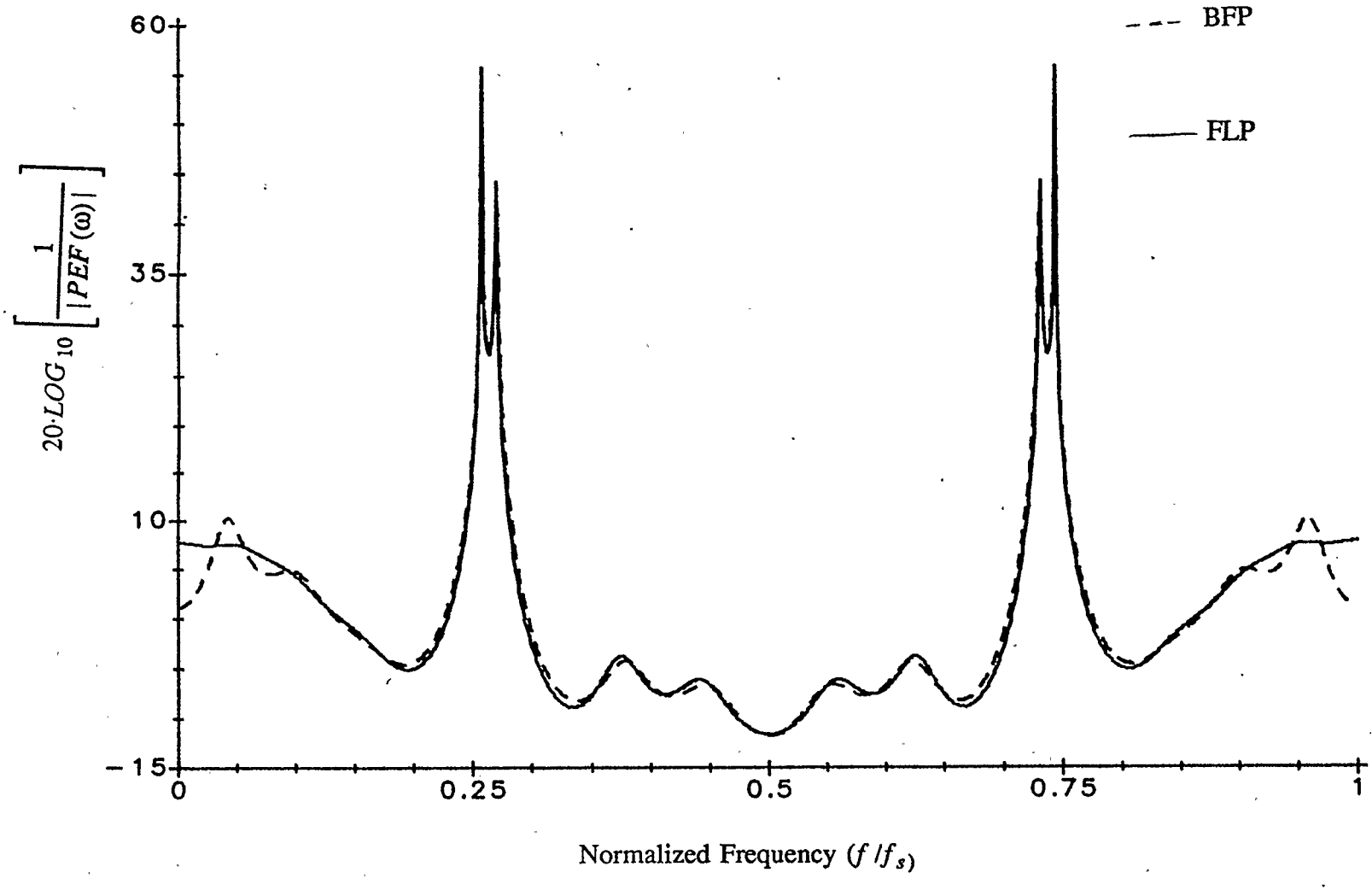


Figure 6.3 Comparision of the Spectral Estimates obtained from BFP and FLP Burg Algorithms when applied to Real Data.

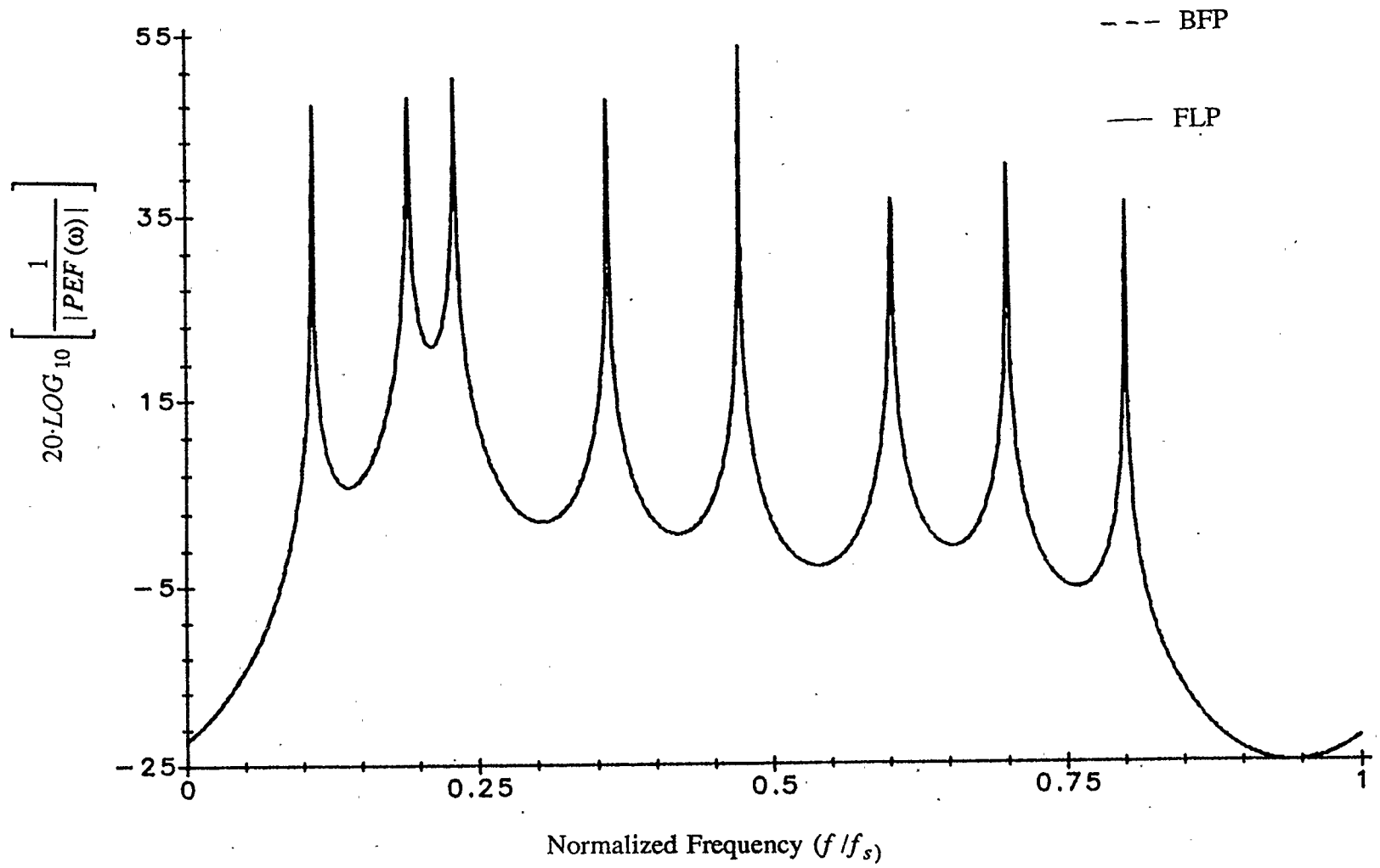


Figure 6.4 Comparision of the Spectral Estimates obtained from BFP and FLP Burg Algorithms when applied to Complex Data

regenerated for every order based on previously computed values implies that a large amount of roundoff error could accumulate. The mean and variance of the error between the BFP and FLP Burg algorithms for various model orders are given in table 6.1 for both test cases. For the given data sets it can be seen that the worst roundoff error had a value of 8.9 which translates into 3 to 4 bits of lost resolution. and the average mean error lies between 2 and 3 which translates to a 2 bit error. In most real time situations the data would be gathered by an 8 to 12 bit A/D and the sampling roundoff error is much more significant than the modeling error.

While the mean error does not increase dramatically with model order, the variance increases tremendously. As the model order increases, the prediction errors theoretically decrease. Therefore the round off error that is present becomes significant and is also modeled thus causing a dramatic increase in the variance.

6.3 ACTUAL RUN TIMES

The actual operating clock speed of the implementation lies somewhere between 7.2 MHz where the algorithm would sporadically fail and 7.5 MHz where it would frequently fail. This ambiguity is due to the wire-wrap implementation which introduces a great deal of noise. The lower clock speed was taken as the maximum operating limit. An operating speed of 7.2 MHz translates to a clock period of 139 ns which is close to the predicted value of 138 ns.

Table 6.2 shows the experimental run times obtained when the clock speed was 7.2 MHz. For comparison, the minimum and maximum theoretical run times determined in chapter 5 are given. Examining table 6.2 shows that experimental run

Table 6.1 : Block Floating Point Round off Error Statistics

order p	normalized variance σ^2/Δ^2		normalized mean error μ/Δ	
	real	complex	real	complex
0	0.97	2.2	0.02	0.005
1	2.6	24	0.32	0.87
2	11	41	1.1	1.9
4	1000	2400	0.83	8.9
8	5500	16700	2.67	1.1
16	16000		3.30	

Table 6.2 Run Times for Different Values of N, and the Order

order	N	min	actual	max
		μs	μs	μs
1	16	72	76	92
	32	114	136	143
	64	199	228	247
	128	368	400	453
2	16	140	125	173
	32	225	240	281
	64	394	420	489
	128	732	840	902
4	16	272	200	338
	32	440	440	552
	64	779	820	970
	128	1455	1700	1794
8	32	847	1000	1079
	64	1523	1600	1915
	128	2875	3100	3564
16	64	2898	2900	3739
	128	5603	5400	7035

times lie in between the projected minimum and maximum run times for most cases. The times that lie below the projected minimum can be attributed to the experimental procedure used in determining the actual run times. The corresponding sampling rates for the run times are shown in table 6.3. It is clear that real time operation can be achieved for high model orders since a 16^{th} order, 128 point model was shown to be performed in 5.4 ms with an effective sampling rate of 23 kHz.

6.4 FURTHER CONSIDERATIONS

Though specifically designed to perform the Burg algorithm this APU can be used for a number of applications. The remaining stages of the DSA mentioned in the introduction can be developed using the processor as the major hardware component. Should this processor be incorporated in the DSA, the external I/O interface should be changed from the bus transceivers to first-in first-out stacks to increase the speed of the data transfers.

Reliability can be increased by producing a printed circuit board version of this processor and using updating algorithms. Using an updating Burg algorithm [23] would improve the speed but at the cost of resolution. Campbell [5] has suggested that band selectable digital filtering be employed to reduce the model order of the signal under analysis. By employing this preprocessing technique and using the high speed processor discussed in this thesis, it might be feasible to implement a single board DSA with the high speed processor developed in this thesis being used as the

Table 6.3 Corresponding Sampling Frequencies (In kHz)

data points N	model order p				
	1	2	4	8	16
16	211	128	80		
32	235	133	72.7	32.0	
64	280	152	78	40	22
128	320	152	75	41	23

hardware for the complete DSA.

6.5 CONCLUSIONS

The goal of this thesis was to design a processor capable of performing AR modeling in real time. The Burg algorithm was selected as the modeling algorithm because it offered the best compromise between speed and accuracy. To ensure the stability and accuracy of the algorithm in a block floating point environment the errors arising from block floating point operations were examined. A number of methods including a tree addition algorithm, a modified division algorithm and a hardware rounding unit were used to mitigate the effects of these errors.

Microprogrammable components were incorporated in a highly pipelined architecture that supported real time operation. To perform complex arithmetic in real time a processing unit consisting of two data busses, two ALUs and a multiplier was proposed and implemented. Although additional hardware could have increased the overall speed the above configuration offered a good trade-off between hardware complexity and computational speed.

The Burg algorithm was broken into stages and an efficient microprogrammed implementation of each stage was developed. The use of a modular approach to the programming provided a good compromise between program development time and operational speed. The architecture was able to operate at a clock speed of 7.2 MHz and permitted real time operation of the Burg algorithm. The BFP algorithm was in good agreement with the FLP algorithm for the test cases considered. A worst case roundoff error of 4 bits was observed when an error analysis between the BFP and

FLP algorithms was conducted. This work has been summarized in a paper whose abstract has been accepted by the proceedings of the IEEE. The full paper is presently under review [25].

REFERENCES

(1) J. Makhoul, *Linear Prediction: A Tutorial Review,* Proc. IEEE, Vol. 63, pp. 561 - 580, April 1975.

(2) S. Haykin, *Radar Signal Processing,* IEEE ASSP Magazine, Vol. 2, No.2, pp. 2 - 18, April 1985.

(3) J. P. Burg, *Maximum Entropy Spectral Analysis,* Proceedings of the 37th Meeting of Exploration Geophysicists, 1967.

(4) R. E. Morley, Jr., et. al., *A Multiprocessor Digital Signal Processing System for Real-Time Audio Applications,* IEEE Trans. Acoust., Speech, Signal Processing, Vol. 34, No. 2 pp. 225-231

(5) K. Campbell, *A Microprocessor Based Spectrum Analyzer Using Autoregressive Modeling Techniques,* U. of Calgary, M. Sc. Thesis, September 1984.

(6) M. Ng, *Short Time Fourier Analysis and Synthesis Incorporated with Autoregressive Modeling Techniques,* U. of Calgary, M. Sc. Thesis, April 1983.

(7) H. Orbay, *Architectural Design of a Computing Element for Signal Processing,* U. of Calgary, M. Sc. Thesis, April 1986.

(8) M.R. Smith and S.T. Nichols, *Improved Detection of Signals using Modelling Techniques Combined with Deconvolution,* Steel Industry/Researcher Sensor Research Workshop, May 8-9, 1984, Burlington, Ontario.

(9) M. J. E. Salami, *ARMA Models in Multicomponent Signal Analysis,* U. of Calgary, Ph. D. Thesis, pp. 26, April 1985.

(10) S. M. Kay and S. L. Marple, Jr., *Spectral Analysis - A Modern Perspective,* Proc. IEEE, Vol. 69, No. 11, pp. 1380 - 1419, November 1981.

(11) S. Y. Kung and H. Y. Hu, *A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems,* IEEE Trans. Acoust., Speech, Signal Processing, Vol. 31, No. 1, pp. 66-76, Febuary 1983.

(12) P. F. Fougere, E. J. Zawalick and H. R. Radosk, *Spontaneous Line Splitting in Maximum Entropy Power Spectrum Analysis,* Physics of the Earth and Planetary Interiors, Vol. 12, 1976, pp. 201-207.

(13) D. N. Swingler *Frequency Errors in MEM Processing,* IEEE Trans., Acoust., Speech, Signal Processing, Vol. 28, No. 2 pp. 257-259, nov. 1978.

(14) B. I. Helme and C. L. Nikias, *Improved Spectrum Performance Via a Data-Adaptive Wieghted Burg Technique,* IEEE Trans. Acoust., Speech, Signal Processing, Vol. 33, No. 4, pp. 903-910, Aug. 1985.

(15) P. F. Fougere, *A Solution to the Problem of Spontaneous Line Splitting in Maximum Entropy Power Spectrum Methods,* Journal of Geophysical Research, vol. 82, No. 7, pp. 1051-1054, Dec. 1980.

(16) D. B. Coldham, *Block Floating-Point Arithmetic with Applications,* U. of Calgary, Ph. D. Thesis, April 1977.

(17) J. F. Cavanagh, *Digital Computer Arithmetic Design and Implementation,* MacGraw-Hill, New York, pp. 236 - 303, 1984.

(18) Data Book, *Bipolar Microprocessor Logic and Interface,* Advanced Micro Devices (1983), pp. 524-527.

(19) Data Book, *TMS32010 Users Guide,* Texas Instruments (1985), pp. 2-2.

(20) M. R. Smith, *A METAASSEMBLER for developing microwords for a microprogrammed architecture,* Report, #19 PS 85, Dept. of Elec. Eng., U of Calgary, (Sept. 1985).

(21) H. Orbay and M. R. Smith *A Development Tool for Microprogrammable Systems: General Purpose Controller,* Report, #28 CO 85, Dept. of Elec. Eng., U. of Calgary, (Sept. 1985).

(22) J. W. Locke, *Designing Digital Signal/Array Processors with the AM29500 Family,* Advanced Micro Devices (1984).

(23) S. W. Nichols and M. R. Smith, *The Hardware and Microcode for the Burg Algorithm,* Internal Report, Dept. of Elec. Eng., U of Calgary, (expected completion Nov. 86).

(24) C. J. Gibson and S Haykin *Learning Characteristics of Adaptive Lattice Filtering Algorithms* IEEE Trans., Acous., Speech, Signal Processing, Vol. 28, No. 6, pp. 681-691, December 1980.

(25) S. W. Nichols, M. R. Smith and H. Orbay *Hardware Implementation of the Burg Algorithm,* Proc. IEEE Special Issue on Hardware and Software for Digital Signal Processing, (under review).