

1985

# A REAL TIME GENERAL PURPOSE SIGNAL PROCESSOR ARCHITECTURE.

MOHSIN MOHAMMAD. JAMALI

*University of Windsor*

Follow this and additional works at: <http://scholar.uwindsor.ca/etd>

---

## Recommended Citation

JAMALI, MOHSIN MOHAMMAD., "A REAL TIME GENERAL PURPOSE SIGNAL PROCESSOR ARCHITECTURE." (1985).  
*Electronic Theses and Dissertations*. Paper 727.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email ([scholarship@uwindsor.ca](mailto:scholarship@uwindsor.ca)) or by telephone at 519-253-3000ext. 3208.

# CANADIAN THESES ON MICROFICHE

I.S.B.N.

## THESES CANADIENNES SUR MICROFICHE



National Library of Canada  
Collections Development Branch

Canadian Theses on  
Microfiche Service

Ottawa, Canada  
K1A 0N4

Bibliothèque nationale du Canada  
Direction du développement des collections

Service des thèses canadiennes  
sur microfiche

### NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION  
HAS BEEN MICROFILMED  
EXACTLY AS RECEIVED

### AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ  
MICROFILMÉE TELLE QUE  
NOUS L'AVONS REÇUE

A REAL TIME GENERAL PURPOSE SIGNAL PROCESSOR ARCHITECTURE

by

Mohsin Mohammad Jamali

A dissertation  
presented to the University of Windsor  
in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy  
in  
The Department of Electrical Engineering



Windsor, Ontario, 1984

(c) Mohsin Mohammad Jamali, 1984

## ACKNOWLEDGEMENT

The author would like to express his sincere thanks and appreciations to his supervisors Dr. G. A. Jullien and DR. S. I. Ahmad for their valuable advice, help and constant encouragement throughout the progress of this research. The author is also thankful to Dr. W. C. Miller and Dr. J. J. Soltis for their valuable advice and suggestions. Thanks are also extended to the graduate students for their help during the course of this work.


The author also wishes to thank his parents, brothers, sisters and friends for their help and unlimited support.

## LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
CCP	Cyclic Convolution Property
CPU	Central Processing Unit
DFG	Data Flow Graph
DFSP	Data Flow Signal Processor
DFT	Discrete Fourier Transform
DIF	Decimation in Frequency
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FNTT	Fast Number Theoretic Transform
IIR	Infinite Impulse Response
MQRNS	Modified Quadratic Residue Number System
NTT	Number Theoretic Transform
PE	Processing Element
PSP	Programmable Signal Processor
QRNS	Quadratic Residue Number System
RAM	Random Access Memory
RNS	Residue Number System
ROM	Read Only Memory
VLSI	Very Large Scale Integration

## LIST OF FIGURES

Number	Figures	Page
2.1	A bit serial array multiplication	17
2.2	Programmable Signal Processor Architecture	20
2.3	X-Pipe evaluation of arithmetic expression	25
2.4	An overview of the expression processor	26
2.5	sytolic arrays	29
2.6	Von-Neumann Model	31
2.7	Mago's tree structure	34
2.8	N*N communication network	37
2.9	Cell structure of a data flow architecture	38
3.1	Hardware implementation of the expression	47
3.2	Bandpass filter structure	54
3.3	Computational element	55
3.4	Normalized frequency response	56
4.1	Binary tree structure	63
4.2	Tree cell structure	67
4.3	Base cell C architecture	68
4.4	Base cell interconnection	69
4.5	Control bits for addressing	70
4.6	Host computer Interface	74
4.7	Ram loading loop table interconnections	75
4.8	Binary tree structure for filters	78
4.9	Filter Outputs for #7 and #8	79



5.1	Radix 2 DIF butterfly	89
5.2	Butterfly for $4n+1$ prime	90
5.3	Twiddle factor multiplication using large moduli	91
5.4	Implementation of NTT on the DFSP	92
5.5	2 D direct convolution algorithm	100
5.6	Computation time for 2 D convolution	101
5.7a	Original piston Head Image	108
5.7b	Filtered image	109
6.1	Data flow graph	124
6.2	Data flow graph for command program	125
A.1	Block diagram of the filter structure	139
A.2	Block diagram of the complete system	140
A.3	binary to residue conversion unit	141
A.4	Modified Scaler unit	144
B.1	QRNS implementation of FIR filters	149
B.2	1 D convolution using QRNS	150

## ABSTRACT

Digital signal processing has many applications in the areas of signal, radar, speech and image processing and real time implementation requires a very high throughput rate. These applications need computation of repeated simple arithmetic operations and few I/O operations. Various processor architectures have been investigated [1-6] which reveals that a special purpose processor appropriate to the algorithms should be designed in order to achieve high throughput rates. Recently research efforts [7-14] are directed towards the exploitation of parallelism in the algorithms and parallel computation of these algorithms. It has been established that the Residue Number System (RNS) can be exploited for high speed signal processing applications [15-25]. The objective of this work is to propose new concepts for high speed computation of real time general purpose signal processing algorithms. As a starting point a very basic example of a set of band pass filter is used to investigate the feasibility of achieving the goal of high speed signal processing using RNS. The digital filter is first constructed and its structure is extended to a more general purpose signal processor architecture.



A novel architecture of a real time general purpose data flow signal processor (DFSP), based on the binary tree structure, is proposed for real time signal processing applications. The data flow signal processor exploits distributed, parallel, and pipeline processing approaches to achieve high throughput rates. The processor utilizes the residue number system (RNS) for high speed signal processing applications. The arithmetic operations in RNS can be performed via Random Access Memory (RAM) look up tables, and the execution time of any particular arithmetic operation is reduced to the access time of a RAM.

The data flow signal processor is demonstrated to be suitable for performing recursive, non-recursive digital filtering and convolution operations. Various signal processing algorithms are studied in order to investigate the adoptability of the DFSP to their special needs. Finally the thesis describes various alternatives for programming the DFSP, and data flow graphs are used to demonstrate the programming of the data flow signal processor. The data flow graphs are incorporated into an interactive program environment which is used to write application programs without knowing the internal architecture of the DFSP.

## CONTENTS

ACKNOWLEDGEMENT . . . . .	ii
LIST OF ABBREVIATIONS . . . . .	iii
LIST OF FIGURES . . . . .	iv
ABSTRACT . . . . .	vi
<u>Chapter</u>	<u>page</u>
I. INTRODUCTION . . . . .	1
The Objective and Outline of the Research . . . . .	8
Thesis Outline . . . . .	10
II. GENERAL SIGNAL PROCESSING ARCHITECTURES . . . . .	11
Introduction . . . . .	11
Bit Slice Approach for Array Multiplications . . . . .	13
Microprogrammable Signal Processor . . . . .	18
Fast Digital signal Processor Using RNS . . . . .	21
Expression Tree Processor . . . . .	22
Systolic Architectures . . . . .	27
Data Flow Structures . . . . .	30
Mago's Tree Structure . . . . .	32
Data Flow Structures Using Packet communication Network . . . . .	35
Summary & Comments . . . . .	39
III. INITIAL STEPS FOR RNS IMPLEMENTATION OF DIGITAL FILTERS . . . . .	42
Introduction . . . . .	42
RNS Review . . . . .	44
Implementation of the Digital Filter . . . . .	49
Extension to General Purpose Signal Processor Architecture . . . . .	57
Summary . . . . .	58
IV. DATA FLOW SIGNAL PROCESSOR . . . . .	59
Introduction . . . . .	59
Architecture Of The Signal Processor . . . . .	64
Tree Structure . . . . .	64
Base Cell Architecture . . . . .	65

Host Computer Interface . . . . .	71
Operation of the Signal Processor . . . . .	76
Comments & Summary . . . . .	80
<b>V. FIR FILTER IMPLEMENTATION . . . . .</b>	<b>82</b>
Introduction . . . . .	82
Number Theoretic Transform . . . . .	83
Direct Convolution . . . . .	93
Two Dimension Image Convolver . . . . .	94
RNS Implementation of direct Convolution . . . . .	96
Throughput Rate Considerations . . . . .	103
Computer Simulation . . . . .	105
Summary . . . . .	110
<b>VI. PROGRAMMING CONCEPTS OF THE NEW DATA FLOW       SIGNAL PROCESSOR . . . . .</b>	<b>112</b>
introduction . . . . .	112
Data Flow Graphs . . . . .	114
Algorithm for Interactive programming on DFSP . . . . .	116
Input Data Algorithm . . . . .	118
Allocation Algorithm . . . . .	119
Example 1. Two Dimensional Convolution . . . . .	120
Summary . . . . .	126
<b>VII. CONCLUSION . . . . .</b>	<b>128</b>
<b>REFERENCES . . . . .</b>	<b>133</b>
<u>Appendix</u> . . . . .	<u>page</u>
<b>A. STRUCTURAL DESIGN OF A FILTER . . . . .</b>	<b>137</b>
Hardware Implementation . . . . .	142
<b>B. COMPLEX DIGITAL SIGNAL PROCESSING . . . . .</b>	<b>145</b>
Hardware Implementation of QRNS . . . . .	147
<b>C. COMPUTER PROGRAMS . . . . .</b>	<b>151</b>
<b>VITA AUCTORIS . . . . .</b>	<b>167</b>

## Chapter I

### INTRODUCTION

Digital signal processing has many applications in the area of speech, geophysics, radar, sonar, and image processing. These applications may be speech synthesis, speech recognition, signal generation, matched filtering, image enhancement, image restoration. They all have two dependent factors. The first concerns various parameters, such as frequency contents, dynamic range and signal to noise ratio. The second factor is of signal modelling which actually determines the processing requirements. The signals are mathematically modelled and manipulated to recover desired signal information. The signal manipulations consist of very few signal processing operations such as difference equation calculations for filtering operations, evaluation of coefficients for the discrete fourier transform (DFT), convolution and vector or matrix arithmetic operations. These common processing functions are the basic tools of digital signal processing and are widely described in the literature [3,25,39]. In the following we give a brief description of some basic signal processing operations and their requirements for the type of architecture employed.

The basic difference equation of an Nth order digital filter is of the form

$$Y(n) = \sum_{k=0}^{N-1} a_k \cdot X(n-k) - \sum_{k=1}^{M-1} b_k \cdot Y(n-k) \quad (1.1)$$

Recursive, or infinite impulse response (IIR), filters may be represented using eqn. (1.1).

$X(n)$ ,  $Y(n)$  are the input and filtered output sequences, respectively. The parameters  $N$ ,  $M$ ,  $a_k$ ,  $b_k$  characterize the mathematical properties of the difference eqn. (1.1). The output  $Y(n)$  is the weighted sum of the  $N$  past and present inputs and  $M$  past outputs. If we remove the feedback elements by setting  $b_k = 0$  and  $k > 1$  then:

$$Y(n) = \sum_{k=0}^{N-1} a_k \cdot X(n-k) \quad (1.2)$$

This equation represents a non-recursive or finite impulse response (FIR) filter and is commonly called the convolution sum. For FIR filters, the convolution sum can be directly evaluated to determine the output of the digital system; for IIR filters the equivalent convolution sum has, in general, an infinite number of terms.

Signal processing operations of frequency translation are commonly used by multiplying the input sequences of samples with the complex exponential which is shown as follows.

$$Y(n) = X(n) \cdot \text{Exp}(j2\pi nk) \quad (1.3)$$

Equation (1.3) is a part of the discrete Fourier transform which is used to translate data from time domain into frequency domain. The convolution property of the discrete Fourier transform can be utilized for implementing FIR filters. The discrete Fourier transform can be defined as:

$$\text{DFT: } X(k) = \sum_{n=0}^{N-1} x(n) \cdot \text{Exp}(-j2\pi nk/N) \quad (1.4)$$

$$\text{DFT}^{-1}: x(n) = \sum_{k=0}^{N-1} X(k) \cdot \text{Exp}(j2\pi nk/N) \quad (1.5)$$

where  $n=0, 1, \dots, N-1$

Various parameters and suitable signal processing operation for a particular digital signal processing (DSP) application specify a signal processing system. It has been shown above that signal processing operations need extensive addition, subtraction and multiplication operations. This arithmetic intensity characteristic of these operations requires a considerable amount of computation time if a processor is not designed to exploit parallelism. The implementation of these signal processing techniques for various applications varies from the use of general purpose computers, to the construction of special purpose hardware. The speed, cost and processing flexibility are factors in the choice of the implementation structure.

An efficient implementation of signal processing algorithms is very important for many applications. There are two

ways to achieve this goal: the first is to have an efficient algorithm, and the second is to implement it using fast processors. It is desirable to have an efficient approach for both algorithm and implementation for real time applications. The algorithms may be made efficient by using new techniques to reduce the number of arithmetic operations. An example may be of reducing the number of multiplications by first adding or subtracting the data which are multiplied by the same coefficient. Jeng [28] obtained an efficient convolution algorithm by dividing the input into smaller sequences and computing the convolution in parallel by using a multiprocessor system. Huang [38] described the convolution sum requiring 25 multiplication and 25 additions which was reduced to 6 multiplications and 15 additions by first adding data which are multiplied by the same coefficients.

The second approach of efficient implementation is to use fast computers. The processing requirements for some of the applications are very high and can only be fulfilled by using a high speed and high performance computer. General purpose computers normally have many extra options such as complex arithmetic logic units, large memory, direct memory access and interrupt capabilities. The cost of these computers as compared to special purpose computers may be very high and some of their options may not be useful for signal processing applications. The implementation of these algor-

ithms on a general purpose computer does not match with architecture and requires arithmetic and memory intensive architectures. An architecture more appropriate for the implementation of these algorithms would require several high speed arithmetic logic unit (ALU), and separate data and coefficient memories. The execution speed of the ALU and the transfer rate of the data between the ALU and the memory should be balanced. The operands should be available for every execution cycle and there should be no excessive wait states either for the operands or for the execution of the arithmetic logic units.

In recent years there has been a growing interest in the implementation of DSP algorithms on special purpose hardware. This hardware varies from dedicated hardwired processors, for a particular algorithm, to specialized processors attached to a host computer. The computational speed is the major factor leading to the choice of a particular implementation dictated by the signal processing application. The basic building blocks of a processor are the components which determine the speed of operation. The components depend upon the current state - of- the art in the appropriate technology. The growth in VLSI technology has led to the implementation of special purpose signal processors. Further improvement in computational speed are obtained by exploiting parallelism in the algorithms by performing concurrent execution of the arithmetic operations.



Several processing elements are connected in parallel in order to perform computations concurrently. The approach of exploiting parallelism can provide high speed computations.

Peled & Liu [3] proposed a bit slice approach for the mechanization of the array multiplication which is a multiplication operation frequently used in signal processing algorithms. This operation can also be performed using a multiplication unit but their computation time will be high and will depend upon the size of the array. Peled & Liu's approach utilizes precomputed multiplication operations which are stored in a Read Only Memory (ROM). The computation time for any array multiplication will be  $B$  times the access time of a ROM ( $B$  is the word length). The Peled & Liu's structure is useful for dedicated signal processing algorithms. Hartung [2] proposed a programmable signal processor using two multiplication accumulator units and separate data memories. Multiplication accumulator units require two cycles to compute one operation. Using two units, an output would be produced in each cycle. The use of multiple memories eliminated delays associated with the waiting of the operands. Huang et al [38] proposed a fast digital processor for performing a two dimensional pulse matching convolution. Their approach adds the data which are multiplied with the same coefficients in order to reduce the number of multiplications. This approach is suitable for one particular application. Jenq [28] has manipulated the convolution al-

gorithm in order to save arithmetic operations and proposed a sequence of operations which leads to saving in computation time. Jenq showed that the technique of breaking the convolution sum in smaller sequences provided faster computation time than the FFT approach. Using several multipliers and adders connected in the form of a tree structure increased the throughput rate.

The exploitation of parallelism in the algorithm is receiving wide attention. Massive parallelism can be achieved if the algorithms are designed to introduce a high degree of pipelining and multiprocessing. This has simulated work in the areas of data flow machines [7-12] and systolic arrays [13]. Data flow machines utilize multiple processing elements for concurrent computation of one algorithm. In a systolic array [13] data is fetched one element at a time from the memory, which can be used by all the processing elements in the array. Data flow machines and systolic arrays are very attractive architectures for high speed implementation of signal processing functions.

Recently the Residue Number System (RNS) [15-25] has been utilized for high speed digital signal processing. Using RNS arithmetic operations are precomputed and are stored in ROM's in the form of look up tables. The computation time of the arithmetic operation is the access time of the ROM. RNS offers carry free operations and is more appropriate for parallel and pipelined computation. This work explores the

use of RNS for the design of a general purpose signal processor.

The architecture of the processor should be able to perform signal processing algorithms efficiently. The processor should be software driven and be able to adopt various applications by modifying the software. This goal can be achieved by exploiting parallel, pipelining and distributed processing approaches to increase the throughput rates. The goals for developing a novel architecture have been described in the following section.

#### 1.1 THE OBJECTIVE AND OUTLINE OF THE RESEARCH

The principal objective of this research is to explore various signal processor architectures which are described in the literature and to propose a novel architecture for a real time general purpose signal processor. The proposed processor should be programmable and suitable for wide variety of signal processing applications. The developments in the area of VLSI has led to the design of more and more signal processors which may be single chip processors [39,40] or special purpose processors optimized for a particular application.

It has been established that the RNS can be used for arithmetic operations associated with high speed signal processing applications. The RNS allows computation of arithmetic operation via look up tables where the operations of

addition, subtraction and multiplication are precalculated and are stored in the look up tables. The computation time required for these operations is the access time of the memory. The RNS offers parallel computation of arithmetic operations and is suitable for pipelining.

This work investigates various architectures which are available in the literature and explores the implementation of digital filters. As a starting point we explore the implementation of a set of band pass filters utilizing RNS approach. The structure of the filter will be extended to adopt wide variety of signal processing algorithms. We propose a binary tree data flow structure which will be capable of performing real time signal processing applications. The tree structure is capable of performing various operations and is very appropriate for distributed, parallel and pipeline processing. The binary tree structure is attached to a host computer which facilitates the programming of the processor.

We show that this proposed processor can accommodate recursive and non recursive filter operations. It is also shown that the direct convolution approach is better than the transform approach for the computation of convolution on this signal processor. Finally the software techniques for programming the processor are considered and various alternatives for programming the data flow signal processor are evaluated. Data flow graphs are shown as a software tool for programming for this processor.

## 1.2 THESIS OUTLINE

In chapter 2, a review of the implementation of the various signal processors is discussed. The architectures for general computations are also explored. Specifically data flow computers and systolic arrays are discussed and we evaluate if these architectures can be exploited for signal processing applications.

Chapter 3 covers the review of RNS and applications of RNS approach for signal processing operations. A description of the construction of a set of band pass filters is provided. The extension of this structure for more general purpose signal processing applications is discussed.

In chapter 4, the proposed novel architecture of a data flow signal processor is discussed. The binary tree structure and its host interface are shown and a scheme of loading the look up tables in the tree structure is described.

Chapter 5 describes the implementation of FIR filters using transform approach and direct convolution. A comparison is shown for a 2 D convolution example of an edge enhancement filtering operation of an image. A complex filtering approach using Quadratic Residue Number System (QRNS) is also presented.

Chapter 6 evaluates various software techniques available for programming parallel computers. A data flow graph approach is shown to program the data flow signal processor.

Chapter 7, concludes with a summary of this research.

## Chapter II

### GENERAL SIGNAL PROCESSING ARCHITECTURES

#### 2.1 INTRODUCTION

The growing demand for high speed computation has led to the design of faster computers utilizing the state of the art components. Efforts have been directed towards exploiting the parallelism in the algorithms and performing parallel operations. The approaches of pipelining in the instruction and utilizing more than one CPU's are also applied in order to reduce computation time. Array processors are used to provide faster computational ability. The concept of dual or multi processors is also a popular technique in the design of general purpose computers. The area of digital signal processing demands higher computation speed for real time applications. Many dedicated systems have been implemented for various signal processing algorithms. The techniques of developing efficient algorithms and using high speed hardware have been utilized and many researchers exploited the characteristics of a particular algorithm by tuning the architecture for maximum speed. This chapter reviews some of the dedicated signal processing approaches that have been adopted for dedicated signal processing hardware. Three different structures are discussed representing

techniques which are used to improve throughput rate of a system. First approach discussed is of Peled & Liu [3] which mechanizes the multiplication accumulation operation frequently used in signal processing applications and represents a novel technique for high speed computation. The second structure described is the programmable signal processor [2] and illustrates an example of using multiple functional units and memories. This technique represents a parallel processing of algorithms. The last approach presented is the implementation of fast digital signal processor [38] and is an example of algorithm modelling for efficient computation of algorithms.

A review of other non-signal processing hardware is also presented which includes the expression processor, data flow computers and systolic arrays. The expression processor is based upon a binary tree structure and its processing elements are connected in the form of a binary tree which processes data concurrently. Data flow computers reject the well known Von-Neumann model and, instead, utilize multiple processors executing operations in parallel. The systolic structure provides an opportunity to utilize data for many operations and avoids repeated reading and writing of data from the memory. This property of systolic arrays reduces the computation time of a particular algorithm. The structure of systolic arrays and data flow computers are studied for their feasibility for implementing general purpose sig-

nal processing architecture. In the following sections we review various signal processing architectures that have been adopted to handle the high speed arithmetic functions required for many real-time applications.

## 2.2 BIT SLICE APPROACH FOR ARRAY MULTIPLICATIONS

The multiplication of coefficients and data is a frequent arithmetic operation in the implementation of digital filters. This operation may be implemented using a multiplier and accumulator unit. Its sequential operation may require a longer computation time than that obtained using several multiplication units computing in parallel. Peled & Liu [3] proposed a new hardware realization for multiplication of a priori known coefficients and data using precomputed function which is stored in a ROM. The computation time will be the access time of the ROM times its word length. This approach is described using mathematical equations for calculating the function which can be precomputed and stored in the form of a lookup table. A frequent arithmetic operation which is repeatedly performed is of an array multiplications as shown by the following equation.

$$y = \sum_{j=1}^L a_j \cdot x_j \quad (2.1)$$

$a_j$  is the set of predetermined coefficients and  $x_j$  are the input data values and  $y$  is the output. The scaled values of



input data  $|x_j| < 1$  can be represented in the 2's complement code with B bits accuracy in fixed point notation and can be rewritten as

$$y = \sum_{j=1}^L a_j \cdot \sum_{k=1}^{B-1} x_j^k \cdot 2^{-1-k} \quad (2.2)$$

If we interchange the order of summation over the indices j and k yields

$$y = \sum_{k=1}^{B-1} 2^{-k} \cdot \sum_{j=1}^L x_j^k \cdot a_j = \sum_{j=1}^L a_j \cdot x_j^0 \quad (2.3)$$

a function F with L binary valued arguments can be defined as

$$F(x^1, x^2, \dots, x^L) = \sum_{j=1}^L x_j^j \cdot a_j \quad (2.4)$$

Now equation (2.2) can be rewritten as.

$$y = \sum_{k=1}^{B-1} 2^{-k} F(X_1^k, X_2^k, \dots, X_n^L) - F(X_1^0, \dots, X_n^L)$$

The values of the function are prestored in a ROM. There will be  $2^L$  possible values of  $F$  since  $F$  can take on the values 0 or 1 only and all possible combinations can be stored in a ROM of size  $2^L$ . The output  $y$  is computed using the given value of the function and adding (subtracting for  $k=0$ ) the partial outputs during the bit serial operations as shown in Figure 2.1. The data sequence  $X_j$  is shifted serially into shift register SR1 to SRL by first shifting the least significant bit followed by all  $B-1$  remaining bits. The address of read only memory (ROM) is constructed by tapping bits  $X_j$  at the output of each shift register. The output of all shift registers becomes the address of the ROM. The value of the function  $F$  is accessed from the ROM and will be loaded into R1 and added to value in R2 and the result appears in R3. The operation is repeated  $B$  times except for the last operation where a subtraction is per-

formed. The computed value of Y is available in R3 at the end of B cycles.

The above approach of Peled & Liu mechanizes array multiplication and is useful for small values of L. An increase in the value of L will exponentially increase the memory requirements. For example an increase in B from 15 to 16 would require an increase in memory from  $2^{15} = 32K$  to  $2^{16} = 64K$ . The design is appropriate for a particular dedicated application. Any change in the algorithm would require hardware modifications to the design. Peled & Liu have shown in [3] various trade offs of the hardware and speeds. Their design suffers with the dependence of the algorithm on the hardware.

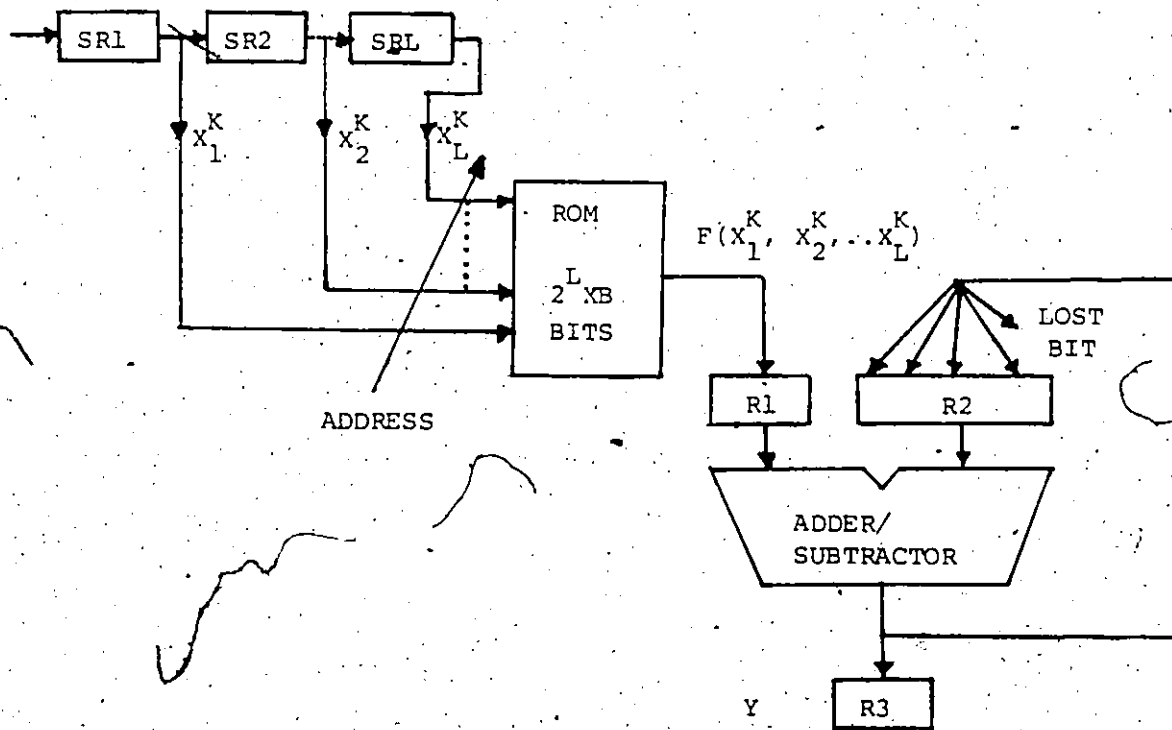


Figure 2.1 A bit serial mechanization of an  $L$  term array multiplication

### 2.3 MICROPROGRAMMABLE SIGNAL PROCESSOR

An architecture review of a microprogrammable signal processor is presented in this section. This design is proposed by Hartung [2] and the processor is attached to a DEC LSI 11 system which is capable of controlling the Programmable Signal Processor (PSP). The architecture utilizes two overlapped functional units, one arithmetic logic unit, three separate memories and separate data, memory and control buses as shown in Figure 2.2. Various functional units, memory and buses operate in parallel to improve the computational speed. The function (multiplication & accumulation) units require two cycles to perform one operation. Using two overlapped functional units can produce results in every cycle provided if the input operands are supplied in every cycle. One functional unit is loaded with two operands while the other outputs to either the scaler unit or to the memory, this maintains one operation per cycle. The philosophy of overlapping the two functional units can be extended to more complex operations by providing the same number of functional units as the number of cycles required to compute one operation. If a functional unit requires  $N$  cycles to perform one operation, then utilizing  $N$  functional units will produce one result in every cycle after  $N$  cycles.

The microprogrammable signal processor proposed by Hartung [2] can implement sequential algorithms such as wave digital filters more efficiently. The proposed processor

does not optimize the functional units, control unit structure and program sequences. It is obvious from this design that by employing multiple functional units the processor speed can be improved.

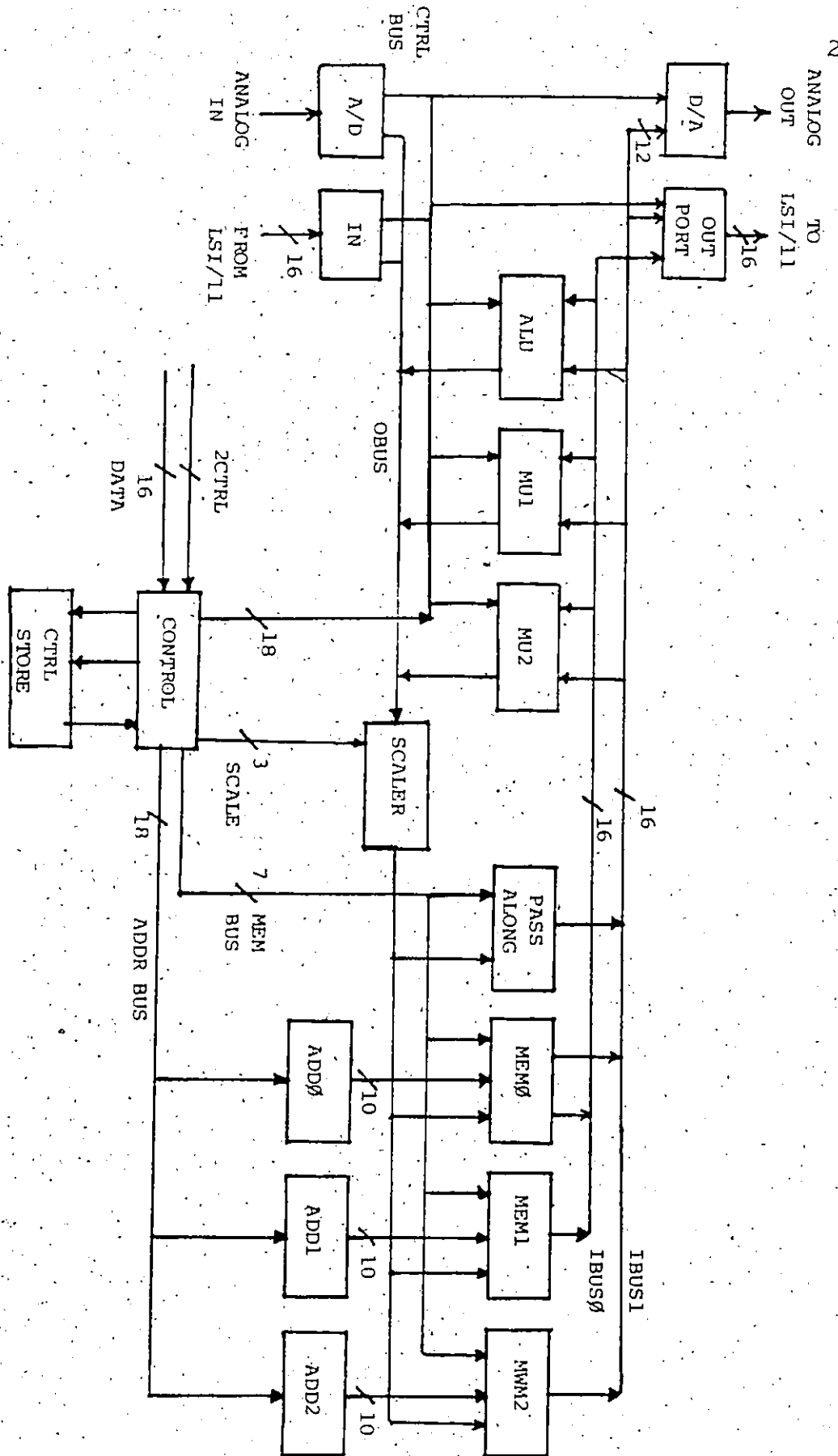


Figure 2.2 Programmable Signal Processor Architecture

## 2.4 FAST DIGITAL SIGNAL PROCESSOR USING RNS

Huang et al [38] proposed a fast digital signal processor for performing a two dimensional pulse matching convolution. A two dimensional convolution is performed using a 5\*5 data with a 5\*5 predetermined filter kernel. The filter is symmetrical and the coefficients of row 1 & row 5 and row 2 & row 4 are identical. In order to save computation time the data of row 1 & row 5, row 2 & row 4 can be added since they are multiplied by the same coefficients. The addition of the data will reduce the required 25 multiplications and 25 additions to 6 multiplications and fifteen additions which will save considerable computation time. The algorithm proposed by Huang et al [38] is an example of algorithm modelling which reduces 25 multiplications and 25 additions for every outputs to 6 multiplications and fifteen additions. This approach offers an efficient algorithm which can be executed with a higher throughput rate than the original algorithm. Huang implemented this algorithm using dedicated hardware controlled by an Intel 8086 microprocessor. Huang utilized a residue number system (RNS) for arithmetic operations. These operations are prestored in the ROM look up tables and the computation time of the arithmetic operation is the access time of the ROM. The detailed design can be found in reference [38].

Processor designs discussed above were directed towards dedicated signal processing applications and now we present



a few general non-signal processing architectures in the following sections.

## 2.5 EXPRESSION TREE PROCESSOR

An expression tree processor [14] based on a pipelined multiple processor architecture which is developed by Vanak-en and Zick and is discussed in this section. The processor is a low cost machine constructed as a binary tree network. It is capable of exploiting the potential concurrency in numerical task and is intended to speed up the execution of a single large task rather than be multiplexed among a number of smaller tasks. The expression processor consists of a number of processing elements (PE's) and a large register space for high speed access of a data. It interfaces to a main memory containing data and its program code.

The operation of an expression tree pipeline (X-pipe) of seven PE's evaluating the expression tree for

$$(7*8+6/2)-(3*4+5+1)$$

is shown in Figure 2.3.

The processing elements are data driven and a series of expression trees can be executed in a pipeline for enhanced concurrency. The design emphasis of the processor is on limiting the complexity of the interconnection topology and distributed control functions in a modular fashion.

An expression processor containing 32 PE's and eight register modules are considered in this design. The 32 PE's are

connected in a binary tree network. Data is supplied from 8 fast RAM modules operating in parallel at a rate of eight words per cycle. The result from X-pipe emerges in serial fashion and is collected in an eight-word register buffer, which is transferred to the register modules in parallel. A block diagram of expression processor is shown in Figure 2.4.

The processing elements are not allowed to exchange data directly with the main memory. Operands are routed to inputs of eight PE's through a set of eight operand corridors. A general purpose alignment network is used to route operands from the register module to the operand corridor. A block transfer of data takes place between the memory and the register space, which is managed by D0-D1 pointers. The main memory stores both data and program codes. A set of I/O channels are used to interface the processor with the outside world. The central control unit manages the program counter, interprets instructions and updates register pointers.

The processor has modular structure, using identical PE's. It simplifies the design of the system, and control functions do not have to be implemented in random logic. It uses a central control unit to perform global coordination, initiating request for register data and issues opcodes to PE's. Most of the interconnection complexity is located in the interface between the PE's and the register modules.

The expression processor concept can be utilized for signal processing applications. A drawback of this design is that operands can not be supplied to all processing elements in the base of the tree structure in one clock cycle. The processor will require redesigning of the processing elements, register modules and memory. An additional module is required for scaling operations and control among all the moduli units.

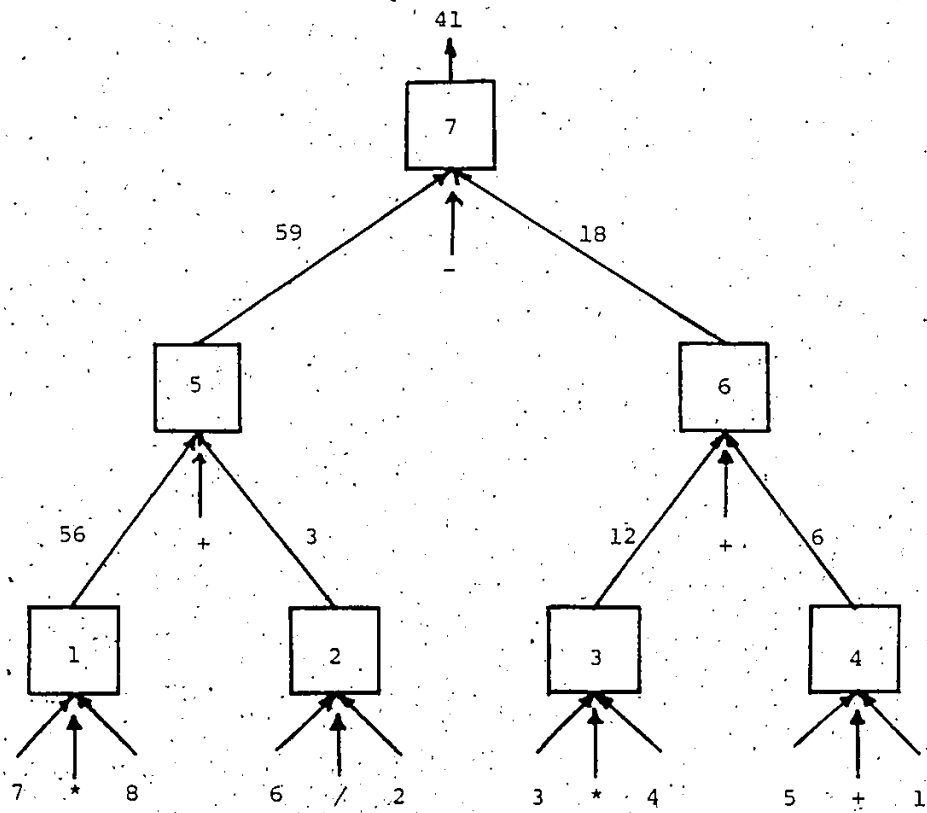


Figure 2.3 X-pipe evaluation of arithmetic expression

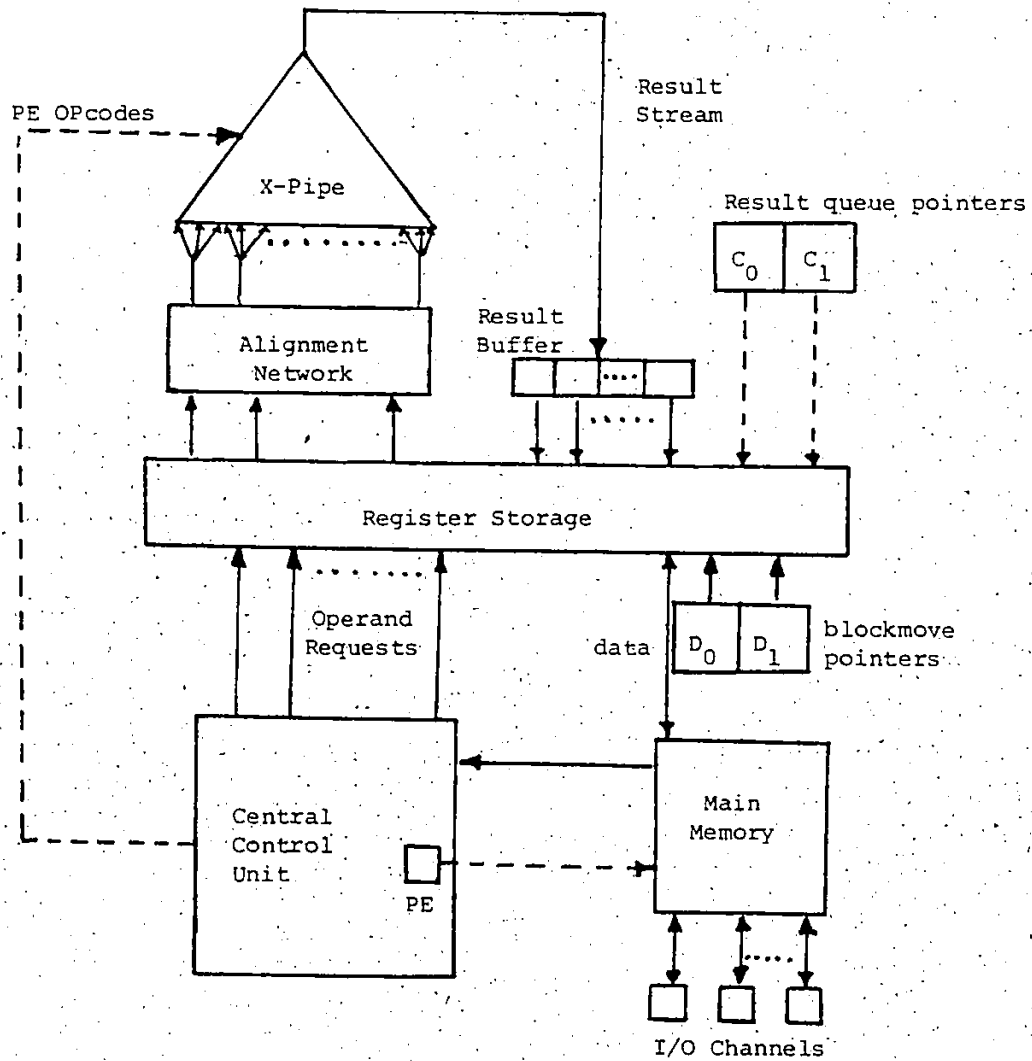


Figure 2.4 An overview of the expression processor

## 2.6 SYSTOLIC ARCHITECTURES

An appropriate method to increase the computation speed is to utilize several processing elements which can operate in parallel. The processing elements should be connected in such a way as to avoid any communication problem between the memory and its processing element (PE). The balance between the execution time of an operation and obtaining the operands from the memory should be maintained. If an operand fetched from the memory can be utilized by several PE's then an increase in the computation speed can be achieved. Kung [13] proposed the concept of systolic architectures for mapping high level computations into hardware structures. The data in a system flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory much as blood circulates to and from the heart.

A systolic system consists of a set of interconnected cells each capable of performing some arithmetic operations as shown in Figure 2.5. Data in a systolic system flows in a pipeline fashion and communication with the outside world occurs only at the boundary cells. In a systolic array the boundary cells may be the I/O ports. The basic principle of systolic structure is to replace single processing element with an array of PE's. This type of architectures can achieve higher computation throughput and use simpler communication and control structures.

An example of systolic design for the convolution problem is presented here.

Given: the sequence of weights  $(w_1, w_2, \dots, w_k)$  and input sequence  $(x_1, x_2, \dots, x_n)$

compute the result sequence  $(y_1, y_2, \dots, y_{n+1-k})$  defined by

$$y_i = w_1 \cdot x(i) + w_2 \cdot x(i+1) + \dots + w_k \cdot x(i+k-1)$$

The convolution problem can be viewed as a problem of combining two data streams  $w_i$ 's and  $x_i$ 's to obtain a result stream of  $y_i$ 's. There are several ways of implementation, one of the three data values are stored in the cells and the other two can be moved either in the same or opposite directions.

Systolic arrays have been utilized for 1 dimensional, 2 dimensional convolutions and matrix multiplications. The convolution problems perform more computation operations than I/O operations. This problem is well suited to systolic arrays since an input data  $x_i$  can be multiplied by all the weighting coefficients. One of the advantages of the systolic arrays is that they utilize identical PE's which are more adoptable for VLSI implementation. Systolic arrays are utilized by various researchers for signal processing applications and offers a promising technique.

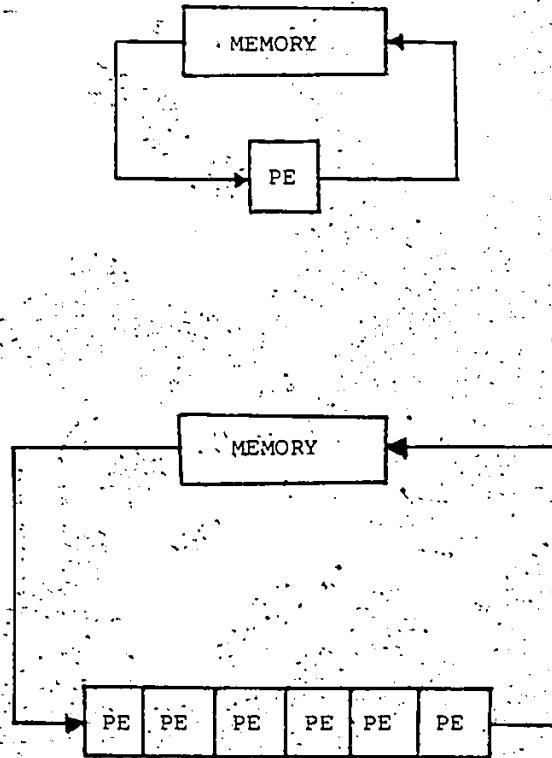


Figure 2.5 Systolic Array



## 2.7 DATA FLOW STRUCTURES

A general purpose computer is based upon the John Von-Neumann model [7-12] which has a single Central Processing Unit (CPU), memory and a link connecting the CPU and the memory as shown in Figure 2.6. The huge contents of the memory must pass through the link which connect CPU and the memory. This link blocks parallel execution of the algorithm and is known as the Von-Neumann bottleneck. The philosophy of the execution of a general purpose computer is based on storage. Instructions are executed by first accessing operands from the storage, perform arithmetic operation and store the output result back in the memory. Algorithms are executed sequentially even if the algorithm may have operations which can be performed in parallel. The sequential execution of algorithms can be changed using direct memory access and interrupt techniques. Direct memory access and interrupt approaches are the earlier efforts to block the sequential operation of a particular algorithm.

Recently attention has been focussed on the data flow computers which rejects the Von-Neumann model. The data flow computers can employ multiple processing elements and provide parallel execution of the algorithms. Two data flow structures namely Mago's [7,9,10] tree structure and U-Interpreter [10] proposed by Dennis and Arvind are presented in the following sections.

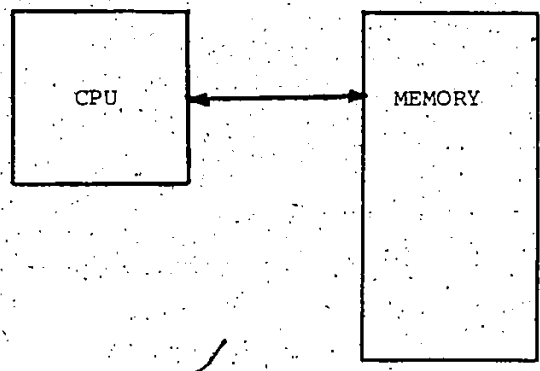


Figure 2.6 Von-Neumann Model

### 2.7.1 Mago's Tree Structure

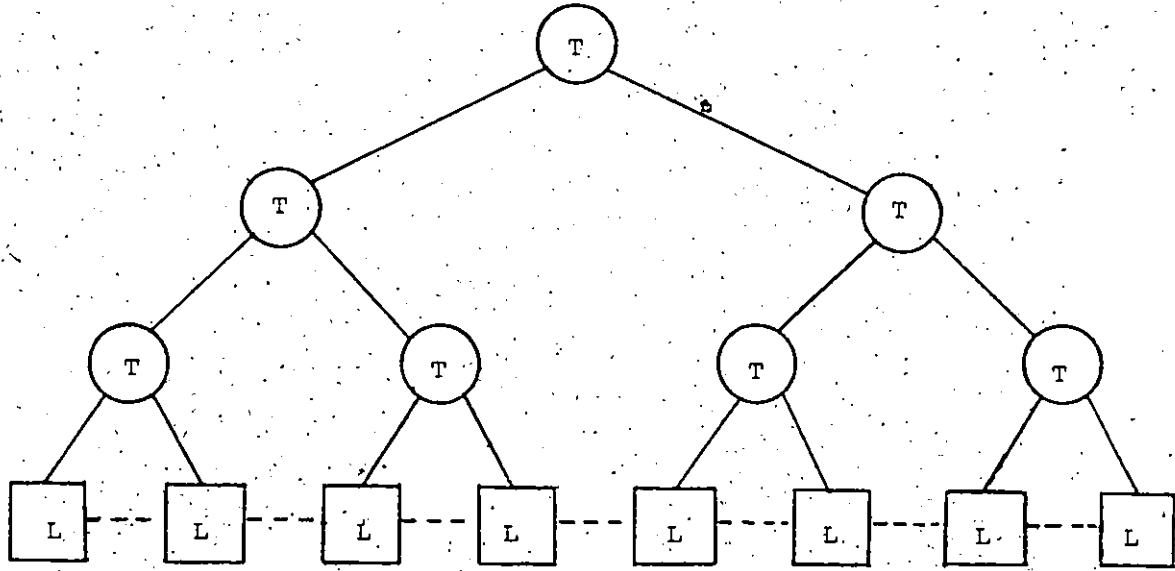
Mago's data flow computer consists of an arbitrarily large number of cells connected in a binary tree [7,9] as shown in Figure 2.7. There are two types of cells namely leaf or L cells and tree or T cells. Each cell communicates with one parent cell and two child cells. The leaf cells form the base of the tree. They are connected to their two neighbouring L cells and are the child of a T cell. All the L cells and T cells are identical, so that the overall computer design is simple and well adapted to VLSI technology.

The functional programs and data are fed into L cells. The T cell partitions the expressions, breaking it down into smaller sub-expressions. They are composed of a function and the data; each function can be evaluated at the same time. The L cell has its local storage and processor, while T cells manage communication between the L with a very simple processing unit. The operation of the computer consists of a series of upward and downward sweeps of information. The process begins with the innermost parts and ends when the entire expression has been evaluated.

T cells distribute the microprogram required by each L cell, and program execution is performed by all the T cells and L cells. The information will sweep up to the top of the tree working on a given expression in each cycle. The information is collected at the top T cell and is sent down for further processing by L cells. These upward and down-

ward sweeps of processing are continued until an entire expression is computed.

This architecture has a fairly simple cell containing a microprocessor and its storage. The cost of the system will not be very expensive since all the cells are identical. The approach is deterministic and can be applied to signal processing applications with several modifications in the structure which will be described later in chapter 4.



L = Leaf Cell (CPU, Condition registers and u storage)  
T = Tree Cell (Data register and simple processing unit)

Figure 2.7 Magq's tree structure computer

### 2.7.2 Data Flow Structures Using Packet communication Network

An important alternate approach of data flow architectures [7,10-12] is led by Dennis and Arvind followed by a number of researcher including Burkowski [12] and Watson [11]. Their machine can employ multiple processing elements connected by a packet communication network as shown in Figure 2.8. The information is conveyed from one unit to another using the tokens or packets. A token contains the data, source and destination tag and sufficient control information to travel across the network. The tokens carry the information telling to what processor the token is to be sent and with which other tokens its data is to be combined. When a processor receives a token it matches with its mate, performs the appropriate operation, and forms a resultant token.

The architecture of a processor is as shown in Figure 2.9. A token normally arrives at the waiting matching section, which keeps tokens waiting for their mates. When a pair of matching tokens is formed, it is sent to instruction fetch retrieving the needed instruction from its program memory. The tokens are then forwarded to ALU for the computation of arithmetic operation. The result is sent to the output section where it is incorporated into a new token whose control information is obtained from the tags of the incoming tokens. The output token is sent back to the network.

There are a few drawbacks of this type of data flow architectures.

1. There is a 200% overhead in the bits of the tokens.
2. The tokens are transmitted byte serially using an asynchronous packet communication protocol.
3. The approach is non-deterministic and it will be difficult to calculate the throughput rate of the system.

These problems lead to the conclusion that this type of architecture is not suitable for real time signal processing applications, since there is an unlimited delay at the waiting matching unit. The token may have to wait for a longer period of time for its mate. Moreover, if a token is lost then the particular algorithm cannot be performed which will affect the performance of the system.

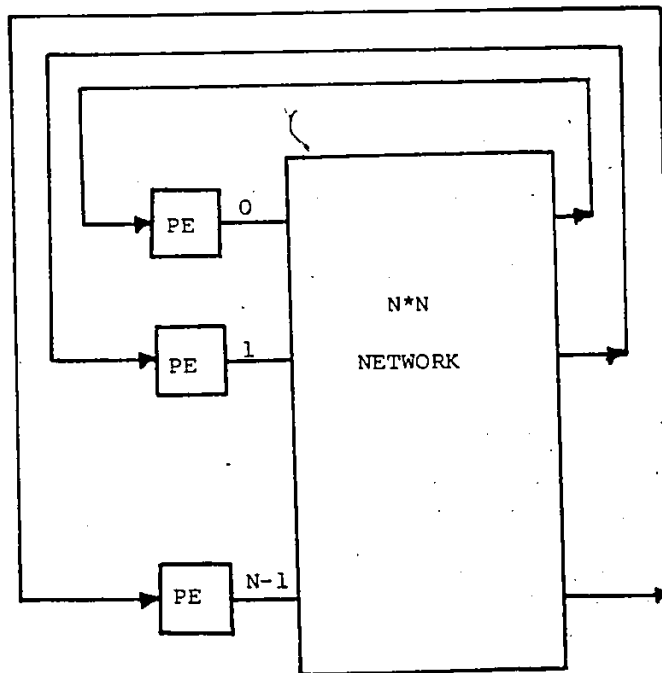


Figure 2.8: N\*N Communication Network



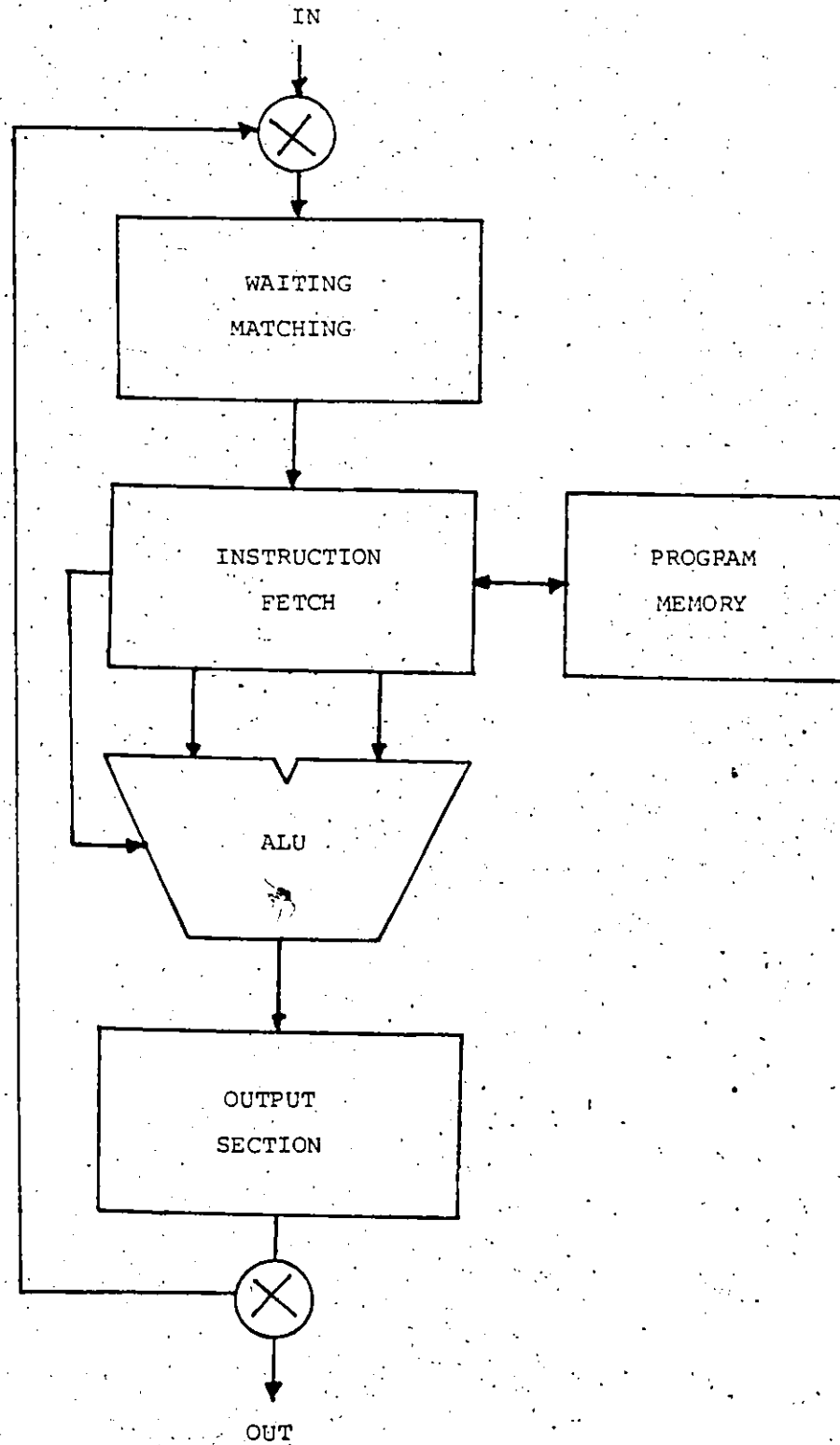


Figure 2.9 Cell structure of a data flow architecture

## 2.8 SUMMARY & COMMENTS

7 A review of various digital signal processing hardware available in the literature has been presented. This review is by no means complete and a full review will be beyond the scope of this thesis. The implementation of a bit slice approach for array multiplication and its limitation were discussed. The main philosophy behind the programmable signal processor proposed by Hartung is the utilization of overlapped functional units which requires two cycles to complete one operation. Using two functional units, the processor can produce an output result in every cycle. Hartung's idea advocates the use of many processing elements to improve the computation speed. The programmable signal processor can be used for the implementation of wave digital filters.

Various other signal processing architectures have been reported in the literature [39-40]. Jenq [28] proposed an algorithm for computing direct convolution. He showed that performing direct convolution using multiprocessor system provides better speed than using transform approach. Baraniecka [21], Jenkins [25], Akhter [23], Modak [24], and Nagpal [22] showed the implementation of number theoretic transform (NTT) using residue number system. Their approach of implementing NTT can be utilized for fast digital signal processing applications. Nagpal et al [20] proposed an architecture for two dimensional convolver which performs 2D convolution using NTT approach within the video rate limits.

The expression tree proposed by Vanaken [14] utilizes multiprocessors connected in a tree structure. This design has a limitation of providing the operands to all the processors in one cycle and better memory management approach is required to further improve its speed of operation. Systolic arrays provide a solution to the balancing of the execution time of the processor and I/O transfer between the memory and the processor. All processors in the systolic arrays can utilize data fetched from the memory repeatedly. This saves the repeated access of the data from the memory and improves the computation speed.

Finally the approach of non-Von Neumann computer such as data flow computers was also evaluated. The data flow computer proposed by Arvind is not appropriate for real time signal processing applications. The main drawbacks of the U-Interpreter are the indefinite time delay in the waiting matching units and the 200% overhead in the communication of the tokens. The tree structure proposed by Mago [9] can be utilized for signal processing applications.

It can be concluded at this point that there is a need for a general purpose signal processor which should be appropriate for a wide variety of signal processing algorithms. The processor should be software configurable and hardware independent. The architecture should exploit all the parallel processing approaches to provide a high throughput rate. A new concept of a general purpose pro-

programmable signal processor architecture is discussed in the next two chapters.

## Chapter III

### INITIAL STEPS FOR RNS IMPLEMENTATION OF DIGITAL FILTERS

#### 3.1 INTRODUCTION

Most of the research work in the area of hardware implementation of digital signal processing algorithms utilizes binary number system (fixed or floating point). It has been established that DSP algorithms require extensive simple addition, subtraction, and multiplication operations. The multiplication is considered as the most time consuming operation and many forms of binary multiplier structure have been proposed to reduce the multiplication time. The commercially available high speed multipliers are restricted to a limited dynamic range. Any increase in the dynamic range may require either an extra multiplier or reduction in the computational speed. The residue number system (RNS) offers carry borrow free arithmetic operations which does not introduce internal delays due to carry-borrow digit propagation, flexibility in the dynamic range, equal computation time for all the arithmetic operations and is described in the following section.

Recent work in the area of residue number systems (RNS) [15-25] provides the opportunity of parallel computation or multiprocessing of high speed digital signal processing al-

gorithms, even though the algorithms do not possess inherent parallelism. Using the RNS arithmetic operations can be pre-stored in ROM arrays. The time required for a particular operation is the access time of the ROM and the time to capture the data in a latch for the pipelining purposes. The memory structures are appropriately utilized by the RNS because of the parallel nature of the arithmetic. The arithmetic operations within a given dynamic range are performed using  $N$  independent parallel paths. The dynamic range can be increased by adding additional parallel paths. The major problems with RNS arithmetic are the computation of division, sign detection and magnitude comparison. Fortunately, the operations can be effectively dispersed with in most of the useful signal processing functions. The other problem, that of efficient scaling has been addressed in great detail in the published literature and several schemes are available to provide overflow protection.

Jenkin and Leon [25] proposed the implementation of FIR filters using RNS. A ROM oriented scaling operations and implementation of recursive filters have been described by Jullien [16]. The implementation of the number theoretic transform has been investigated by Baraniecka [21], Akhter [23], Modak [24], and Nagpal [22]. It has been established by the above mentioned works and the works of other researchers that RNS can be effectively used to implement high speed signal processing algorithms.

## 3.2

## RNS REVIEW

The residue number system (RNS) has been discussed profusely in the literature. Here we present a brief review for completeness.

A number in the RNS can be represented in the form of an n tuple

$$X = (x_1, x_2, x_3, \dots, x_n) \quad (3.1)$$

where  $x_1, x_2, \dots, x_n$  are the residues w.r.t. the set of moduli  $m_1, m_2, m_3, \dots, m_n$  where  $\text{GCD}(m_1, m_2, m_3, \dots, m_n) = 1$ .

The residues are normally represented as  $|x_i|_{m_i}$ . The range of numbers that can be uniquely represented is the product of the moduli  $m_i$ . A binary operation (\*) between two numbers can be represented as:

$$Z = (X * Y) \quad (3.2)$$

It can be shown that the operations can be performed independently between matching residues such that

$$z = |(x_i * y_i)|_{m_i}$$

Thus the result can be computed with n-parallel arithmetic functions. By restricting the size of each module, all possible arithmetic results can be precomputed and stored in the look-up tables. Using RNS arithmetic, addition, subtraction have no carry or borrow operations and multiplication is free from the generation of partial products. A

signed integer can be represented using RNS by assuming positive numbers in the range of  $(0, M/2-1)$ ,  $(0, M-1/2)$  for even and odd  $M$  respectively. The negative numbers can be represented in the interval  $(M/2, M-1)$ ,  $((M+1)/2, M-1)$  for even and odd  $M$  respectively. Division is a complicated process in RNS, normally a scaling operation is used in which the divisor is limited to a predetermined factor and should be a product of some of the moduli.

Recent advances in the memory technology provide an opportunity to perform arithmetic operations using look up tables stored in the memory. The arithmetic operations are precalculated and are stored in a location addressed by the two input operands. A memory size of  $(2^{2B} * B)$  bits is required to store the look up tables for a  $B$  bit modulus. For example  $m_i \leq 32$  can be stored in a commercially available ROM package of  $1k * 8$ . The main advantage of look up table implementation is that it requires equal time to compute addition, subtraction and multiplication. Figure 3.1 shows the implementation of

$$\left\lfloor \frac{5*(A*B) + 3*(C*D)}{31} \right\rfloor \quad (3.3)$$

One of the advantages of RNS is that the multiplication by constants 5 and 3 used in Eqn. (3.3) will not require an extra look up table. Using RNS it is easy to pipeline arithmetic operations since the output of one stage becomes the input address of the next stage. The use of RNS arithmetic requires the conversion of binary numbers into and out of the



RNS representation. A binary to residue operation is simpler than residue to binary conversion and can be implemented using ROM arrays. For example, in an L-moduli RNS the  $i$ th residue of number  $x$  is given by

$$x_i = \left| \sum_{j=0}^{B-1} b_j \cdot 2^j \right|_{m_i} \quad (3.4)$$

where  $b_j$  is the  $j$ th bit of binary representation of  $X$ . For  $B < 10$  bits and  $m < 32$  equation 3.4 can be implemented with a ROM package of  $1k \times 8$  bit.

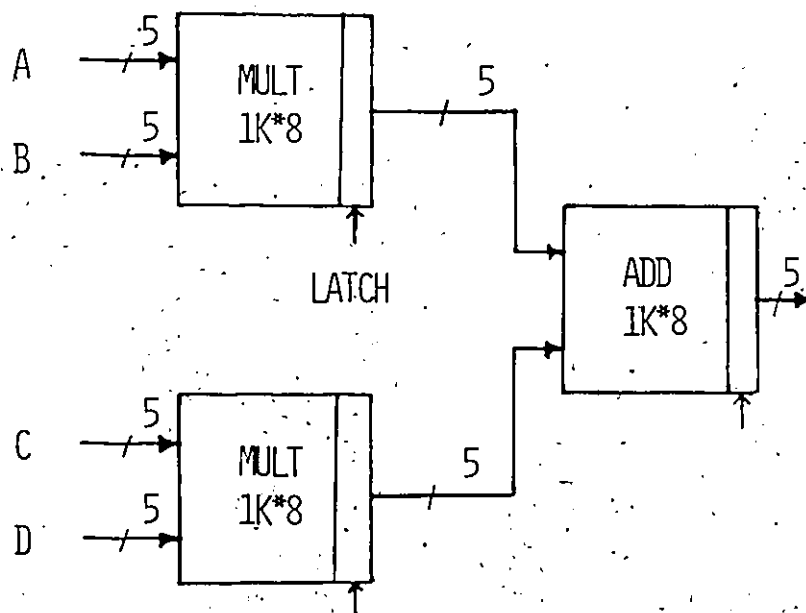


Figure 3.1 Hardware Implementation of  $|5*(A*B) + 3*(C*D)|_{31}$

There are two general techniques for residue to binary conversion based on Chinese Remainder Theorem (CRT) and mixed radix conversion (MRC) schemes.

Using CRT a number  $X$  in the range 0 to  $M-1$  is given by

$$|X|_M = \left| \sum_{j=1}^n m_j \cdot \left| \frac{r_j}{\hat{m}_j} \right|_{m_j} \right|$$

where  $M = \prod_{i=1}^n m_i$ ,  $\hat{m}_i = \frac{M}{m_i}$

and  $\left| \frac{1}{\hat{m}_j} \right|_{m_j}$  is the multiplicative inverse of

The hardware implementation of (3.5) requires a large modulo  $M$  adder which is difficult to implement for large values of  $M$ . The chinese remainder theorem conversion scheme may not be preferable for many applications.

The mixed radix conversion scheme is normally used for high speed implementation of residue to binary conversion. Using MRC a number in the range 0 to  $M-1$  has the representation

$$X = \sum_{i=0}^{L-1} a_i \cdot P_i$$

where  $P_0 = 1$  and  $P_i = \prod_{k=0}^{i-1} m_k$  and the  $a$  are the mixed radix digits in  $0 < a < m$

The  $[a_i]$  can be generated using look up table implementation by forming the partial outputs. The MRC approach is widely used and has been discussed in the literature [21].

RNS arithmetic is applied to a digital filtering problem in order to explore its usefulness and determine the degree of difficulty in its implementations. An example of a set of band pass filters using RNS is analysed and simulated on the digital computer to check the proper flow of data and verify the output results. The filter is then constructed as a prototype to uncover any hidden problems in the architecture.

### 3.3 IMPLEMENTATION OF THE DIGITAL FILTER

Many hardware implementations of digital signal processing algorithms are available in the literature. There are hardware which are dedicated to one particular algorithm. Others are more flexible which may incorporate any changes in the algorithm. The process of designing hardware for a particular algorithm is performing a marriage between the algorithm and the hardware. Before this marriage can be performed there are number of questions which should be answered ie word length, quantization, speed of operations the algorithm characteristics and a simulation study should be used to investigate any flaws in the design. In terms of

hardware considerations, it has been shown earlier that signal processing algorithms require arithmetic and memory intensive architectures. Multiple computational units and memory units can be utilized to perform concurrent arithmetic operations. Multiple memories can provide operands in one clock cycle which will avoid any delay associated with the waiting of operands.

Using an appropriate model of input signal, the filter structure is simulated on the digital computer. The output of the simulated filter is compared with that obtained, using infinite precision to find the error between the outputs. If the error is not within the acceptable limits, the word length can be varied in order to achieve an acceptable error. The wordlength may be decided based on the simulation results. With the specified dynamic range and quantization level, a suitable hardware can be realized. The computation is again performed within the specified wordlength and checked for any overflow during the execution of the algorithm. If there is any overflow dynamic range may be increased. After verification from the simulation of the hardware, efforts can be directed towards a more detailed design of the hardware. In order to verify these steps for building general hardware for digital signal processing algorithm, we select a very basic design of a set of band pass filters which is first designed and will be constructed. We will use this example for a set of band pass filters as a

test vehicle for more general purpose structure. The process of designing hardware for a digital filter is divided into the step of structural design of the filter and its hardware implementation. The structural design of the filter has been described in detail in Appendix A.

A set of band pass filters can be implemented by multiplexing a single filter section as shown in Figure 3.2. The first part of the filter is a comb filter which can be used unchanged for all the filters. The latter part of the filter is a second order section. The result is a filter with the maximum gain at the particular cancelling frequency and reducing gain side lobes. This filter is expressed mathematically as follows.

$$Y(n) = X(n) - a \cdot \cos(\pi k/32) \cdot X(n-1) - a^{64} \cdot X(n-64) + a^2 \cdot Y(n-2) \\ + a^{65} \cdot \cos(\pi k/32) \cdot X(n-65) + 2a \cdot \cos(\pi k/32) \cdot Y(n-1)$$

where  $k=1, \dots, 32$ , and  $a=1-2^{-5}$

It can be seen from the above equation that the filter implementation requires multiplication, addition and subtraction operations. These operations can be precalculated and stored in ROM's in the form of look up tables. One of the advantage of RNS is that multiplying with a constant

does not require an extra look up table. This property has been exploited and multiplication with fixed coefficients are incorporated in the look up tables. The block diagram of the computational element is shown in Figure 3.3. Four five bit moduli 27, 29, 31 and 32 are selected giving a dynamic range of 20 bits which is sufficient for this application. The detailed structure of the filter is given in Appendix A. Binary data are converted into four residues which are inputs to the computational element. A programmable delay controller is implemented and a FIFO is used to provide an arbitrary delay. Thus, when initializing the 64 stage delay FIFO, we start the shift in pulse train, wait for 64 pulses to occur and then start the shift out pulse train. Using this scheme, there is one pitfall to watch out for. We have to make sure that the data has sufficient time to 'fall through' from the input to the output. Fall through time has to be less than or equal to the delay of the FIFO. There are four computational elements representing each moduli and data is processed in four redundant paths. The computational element produces four output residues and they are scaled down using an efficient scaler which gives a direct binary output. The filter is constructed using EPROM's, shift registers and FIFO memories. The performance of the filters, in terms of throughput rate depends on the components chosen. The basic pipeline element for the arithmetic circuit is an EPROM followed by a latch. We used the 2708 EPROM and

8212 latch. The worst propagation delay is  $(470+30)=500$  ns. The delay time can be reduced if we select faster EPROM's. The basic pipeline element for the delay stages is a FIFO. We used the MMI 67402 5 bit FIFO which gives the worst case propagation delay of  $3\mu s/D$  (D is the FIFO delay) or 100 ns whichever is the larger. For delays, D, greater than 30, the worst case delay is 100 ns. This covers all of the FIFO's except those in the scaler array. In the later case the worst case propagation delay is 115 ns. Using an average figure  $(500+115)/2=305$  ns, the output data will be generated at a rate of 3.2 M-Hz which translates into a rate of 100 K-Hz for input data. The filter bank can thus handle signals with bandwidths of up to 50 K-Hz. ➤

A varying frequency sinusoid of maximum allowed magnitude was applied to the input of the filters. The output of each filter was monitored on an oscilloscope. A maximum positive magnitude of the sinusoid output was measured of each increment in the input frequency. Normalized filter gain versus frequency was plotted for three consecutive filter outputs 13,14,15 as shown in Figure 3.4. We used for convenience an 8 bit A/D and an 8 bit D/A chip. Some errors are to be expected between simulated and experimental results because of this excessive quantization. Even so, the results as shown in the comparison plot are in remarkably close agreement.



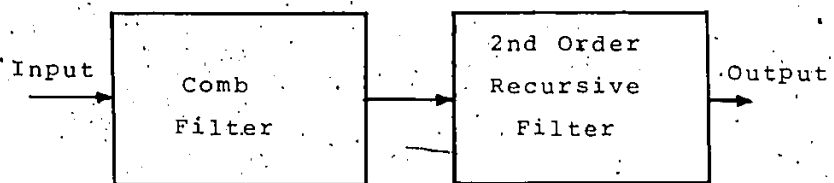


Figure 3.2: Bandpass Filter Structure

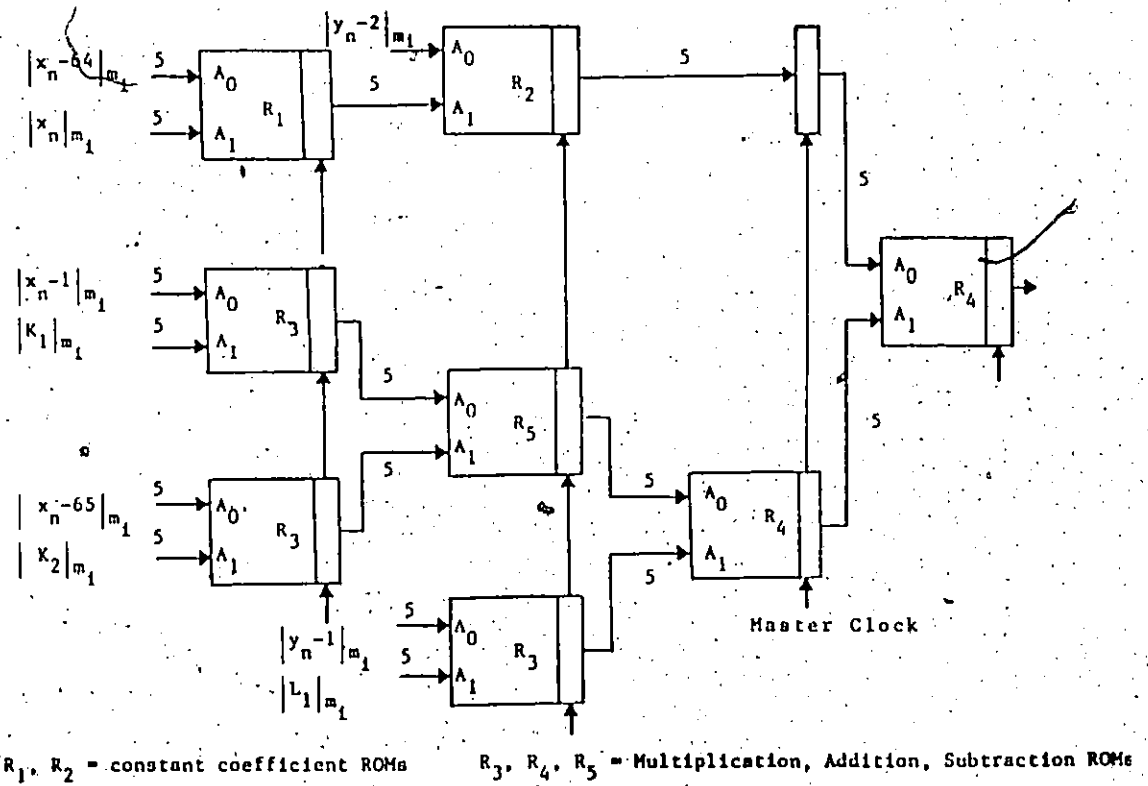


Figure 3.3: Computational Element

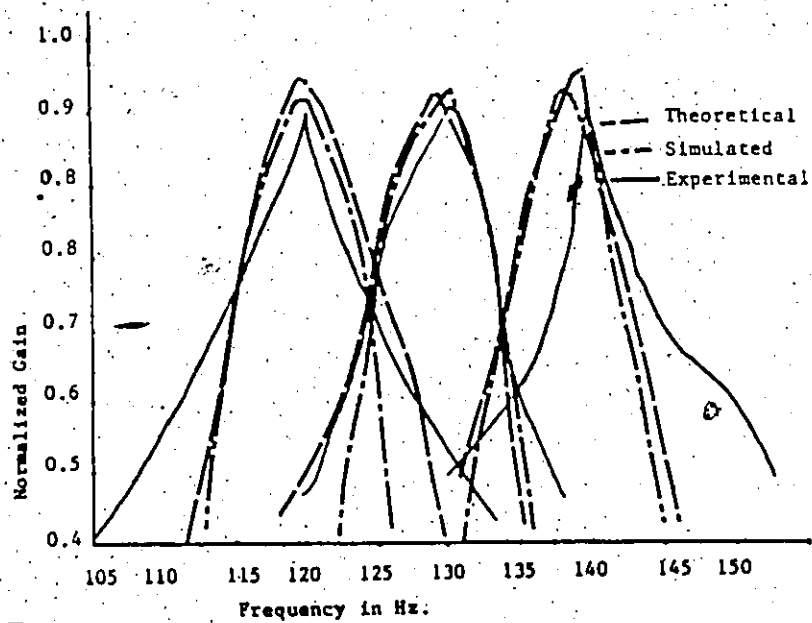


Figure 3.4: Normalized Frequency Response

### 3.4 EXTENSION TO GENERAL PURPOSE SIGNAL PROCESSOR ARCHITECTURE

The implementation of the digital filter described above utilizes EPROM's for storing precalculated arithmetic operations and shift registers & FIFO's to provide input and output data delays. This structure of digital filter can be extended to a more general purpose signal processor by changing all the EPROM's used for storing arithmetic operation and coefficients into RAM's. The data can also be stored in RAM's and any arbitrary delay can be obtained by an appropriate address generation scheme.

The structure of the above filter has many arithmetic operations which are independent of each other and can be performed in parallel. This structure will then resemble a binary tree having a number of cells at every level. The base of the tree structure can be used as multipliers. The outputs of the base cells are forwarded to the next level for additions and subtractions. The final output can be obtained at the last stage. If this structure is replaced with a binary tree structure then it can be used by a large number signal processing algorithms. This idea is implemented into a general purpose signal processor and the example of the set of bandpass filters will be used to verify the adaptability to the new architecture.

### 3.5 SUMMARY

A review of the RNS was presented and it was shown that arithmetic operations can be precalculated and stored in ROM's in the form of look up tables. The RNS offers carry borrow free multiplication, addition and subtraction operations and operations with fixed numbers do not require an extra look up table. The time required for a particular arithmetic operation is the access time of the ROM and time to capture data in a latch for pipelining purposes.

This work demonstrates that practical high speed digital filtering designs can be realized using the residue number system. The resulting memory intensive structure which implement the filter example shows the viability of inherent parallel processing, and ease of pipelining. This is evident even in the implementation of recursive filter structure with inherent feedback loops. Although the structure is programmable in nature changing coefficients and moduli will require re-programming of EPROM's and some changes to the control circuitry. It was discussed that the filter structure can be extended to more general signal processing architecture.

## Chapter IV

### DATA FLOW SIGNAL PROCESSOR

#### 4.1 INTRODUCTION

Advances in the computer technology has led to a growing demand for real time applications of digital signal processing. Algorithms for these applications require repeated simple arithmetic operations and few I/O operations. The characteristics of these algorithms can be exploited by designing arithmetic and memory intensive architectures. Many arithmetic units can be utilized to perform arithmetic operations concurrently and separate memories can provide data and coefficients as required operands in one cycle. Accessing of two operands in one cycle will improve the throughput rate. Many researchers have exploited these characteristics and designed various processors for dedicated signal processing applications [1-6,15-26]. These hardware seem to have one common thing that they are dedicated to, one or another class of algorithms such as filtering or transform algorithms.

Hardware discussed in the literature [2-5,28,38] have exploited parallelism to a certain extent and some efforts are made to exploit parallelism in the algorithm. In order to execute a particular algorithm efficiently both algorithms

and their implementation should be exploited for maximum throughput rates. Many of the hardware in the literature tend to be dedicated to a particular algorithm. Our intention is to introduce a new concept which can be used to develop a general purpose signal processor architecture. The processor can be utilized for a wide variety of signal processing algorithms.

We utilize the results of previous example of the band pass filter for implementation of the digital filters and develop a novel general purpose signal processor. It can be shown that a binary tree may be able to adopt a wide variety of algorithms and perform computations concurrently. The concurrent computation of a particular algorithm will give a very high throughput rate which will be suitable for real time signal processing applications. Processing elements can be connected in a binary tree and may have their CPU and memory. The parallel operation of these processing elements will be independent of each other. The operands will be available simultaneously to perform a particular arithmetic operation. Outputs will be forwarded to the next level of the processing elements. This architecture rejects the Von-Neumann model and will not require any central control unit. This structure can be named as data flow tree structure since the data flows without any control unit. The structure may require a host computer to provide a user interface. We propose a general purpose data flow signal processor archi-

itecture which can be used for many signal processing applications and should be software driven.

This chapter on the Data Flow Signal Processor (DFSP) describes a novel structure of a programmable general purpose signal processor for real time signal processing operations. The DFSP architecture is based on a binary tree structure [7,9,14,18] as shown in Figure 4.1.

There may be an arbitrary number of processing cells in the binary tree, depending upon the design, processing data simultaneously. A particular algorithm can be divided into smaller tasks and distributed to several processing cells. A host computer distributes the control program to the base cells of the tree structure. The base cells perform individual tasks, and facilitate the multiprocessing and pipelining of the tree structure. The data flows without requiring any central control units, unlike Von-Neumann architectures.

The processor utilizes RNS for the arithmetic operations which can be performed in parallel for each modulus. By restricting the size of each modulus, all possible arithmetic operations can be precomputed and stored in the look up tables. The structure of the DFSP discussed in this thesis is restricted to four five bit moduli but the number of moduli can be increased. Identical tree structure are required for each modulus as discussed in the previous chapter. The tree structure is appropriately utilized by the RNS because of the parallel nature of the arithmetic. This chapter demon-



strates the efficiencies available in the RNS by describing a real time general purpose data flow signal processor. First we describe the architecture of the DFSP and use the example of the set of band pass filter to show the operation of the DFSP.

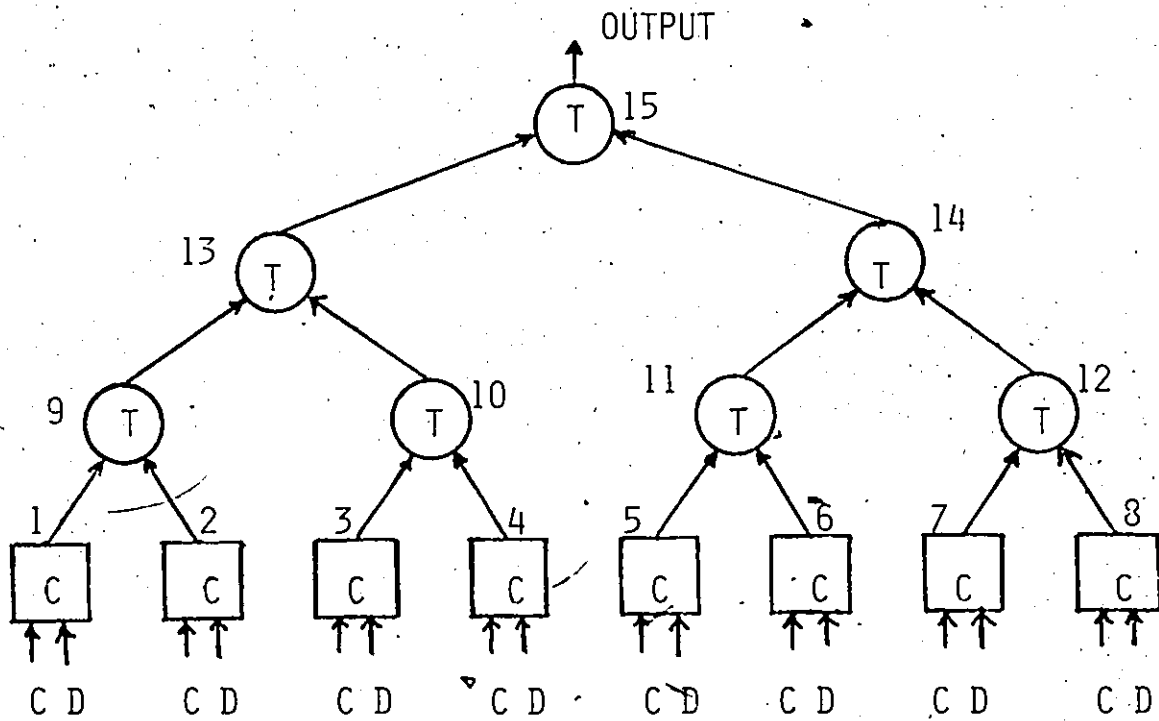


Figure 4.1: Binary Tree Structure

## 4.2 ARCHITECTURE OF THE SIGNAL PROCESSOR

The nucleus of a real time general purpose signal processor is its architecture. It has been discussed earlier that the architecture should be able to exploit parallelism in the algorithm. This goal is achieved by using a binary tree structure which facilitates distributed, parallel and pipeline processing of algorithms. The processor architecture consists of three sections: namely the tree structure, the base cell and the host interface. The tree structure describes its various cells and their interconnections.

## 4.3 TREE STRUCTURE

The computational element is based upon a binary tree structure using 15 cells as shown in Figure 4.1. Every cell in the system is connected to a parent cell and to two child cells except the base cells which do not have any child cells. Each cell receives two operands from its child cell and performs the specified arithmetic operation. The output is forwarded to the parent cell. All the cells operate in parallel and the data moves in a pipeline fashion. It is assumed that four identical tree structures, one for each modulus, will be adequate for a large number of applications. The output of the tree structure is scaled down using a scaler unit which also provides a direct binary output. The scaler unit forwards either the partial outputs to

the base cells or final outputs to the output units for digital to analog conversion. There are two types of cells in the tree structure, the base cell C and the parent cell T. The parent cells consist of RAM's storing one or more look up tables for the arithmetic operations. These tables can be utilized between different passes of the program execution by switching the memory. The number of look up tables is restricted to four tables as shown in Figure 4.2. This number can be increased by allowing more memory in the T-cells. The architecture of the base cell is described in the following section.

#### 4.4 BASE CELL ARCHITECTURE

The base cells in the tree structure are the most important and the most complex. They provide an interface between the tree structures and the host computer. The host computer distributes the individual task to each base cell in the form of microcode. The base cells execute the microcode to perform the five functions.

1. To receive data and partial outputs from the data buses.
2. To obtain coefficients and data from respective memories.
3. To perform a specified arithmetic operation and forward an output operand.

4. To facilitate loading of the look up tables in the T cells in order to configure the hardware to implement the required signal processing algorithm.
5. To provide control bits to enable one of the four look up tables in each T cell.

One of the efficiencies of this architecture is to utilize one base cell for all the four computational elements. The four tree structures are identical and they execute the same operation at every stage. This approach divides base cells into two cells; a main cell, P, and a modulus cell, M, as shown in Figure 4.3 and Figure 4.4. The division of the cell contributes to hardware savings and simplifies control circuitry. The main cells are common to all the computational elements. They consist of a microprocessor, memory, and address generation logic for coefficient and data memories. The modulus cell is comprised of a separate coefficient, data storage and look up table memory for the desired arithmetic operations for each modulus. The coefficient memory stores the coefficients in five least significant bits. The unused bits 6,7 and 8 from each of the operands are utilized to enable one of 4 look up tables stored in the T cells as shown in Figure 4.5.

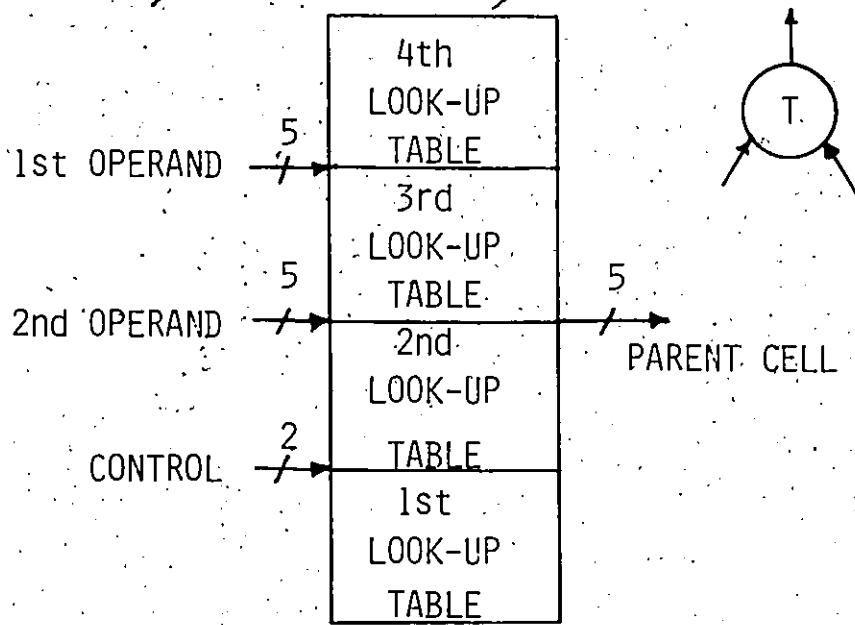


Figure 4.2: Tree Cell Structure

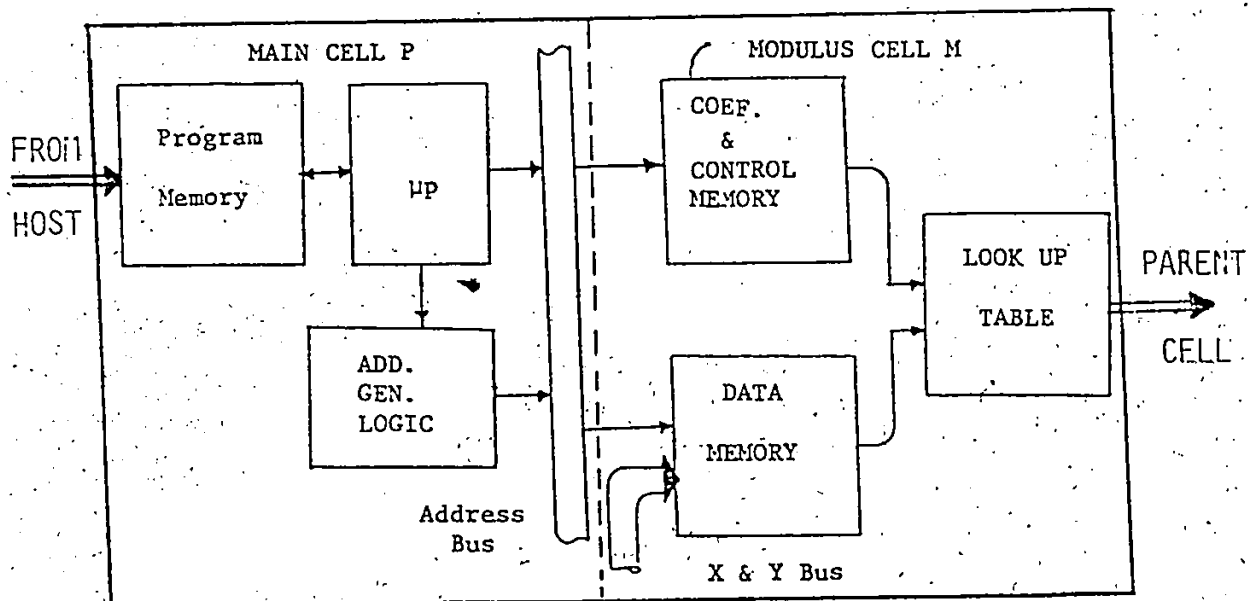


Figure 4.3: Base Cell C Architecture

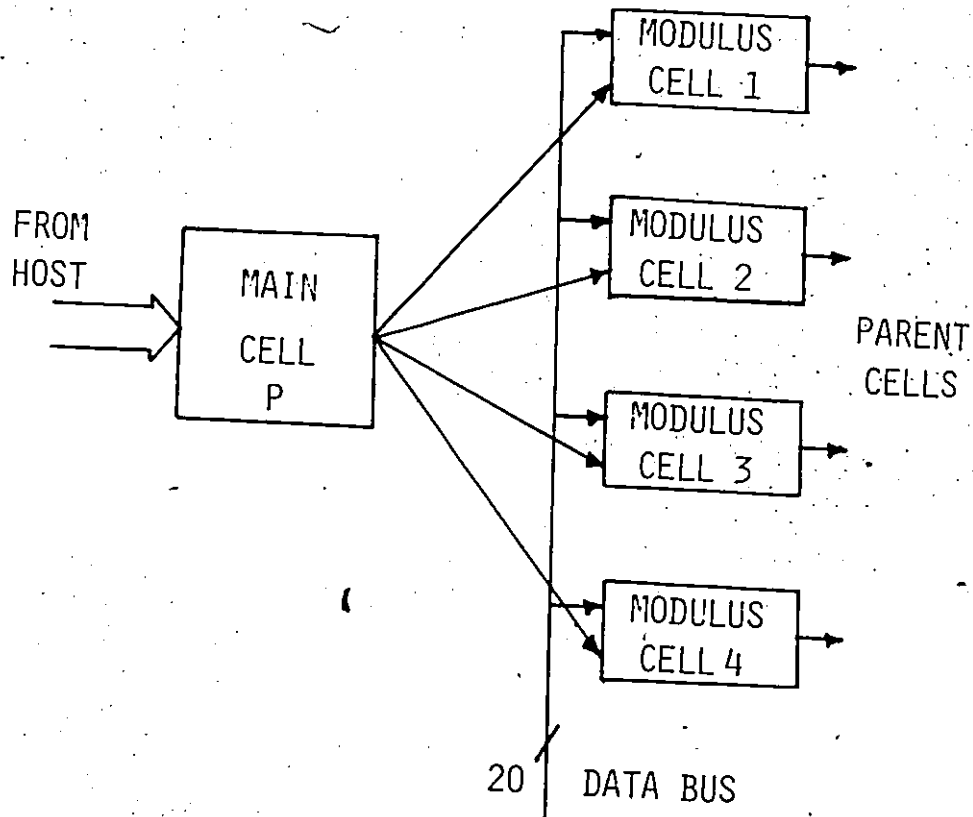


Figure 4.4: Base Cell Interconnection



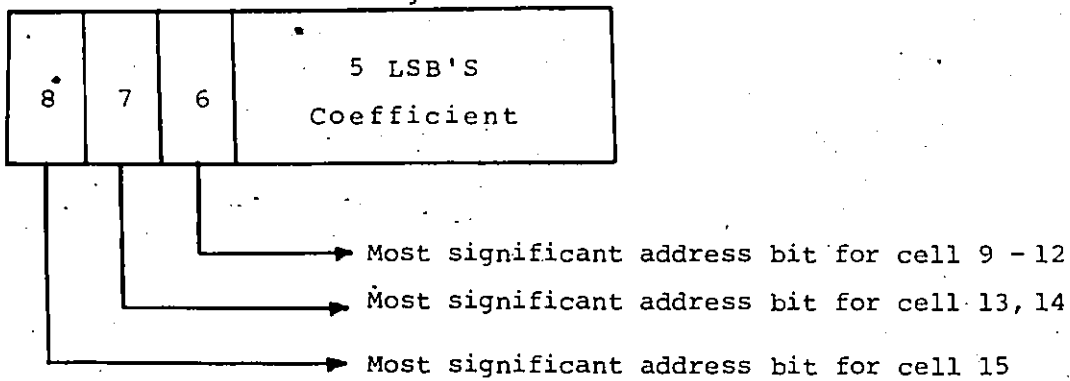


Figure 4.5: Coefficients and Control Bits

#### 4.5 HOST COMPUTER INTERFACE

The host computer is a general purpose computer providing an interface between a programmer and the signal processor. The host computer facilitates writing of specific programs for any particular signal processing algorithm. The processor has been simulated for recursive, and non-recursive filters and convolution for one and two dimensions and has demonstrated flexibility. There are two main objectives of the host computer. The first is to analyze algorithms and determine the operation performed by the various cells. The second objective is to program the main cells with an appropriate microcode, program the lookup tables and monitor real time execution of any signal processing application. The host computer is a 16 bit machine and shares its memory with the main cells of the tree structure, as shown in Figure 4.6. The computer generates the micro code for each individual task and writes it in the respective memory locations of the cells P1 to P8. The host computer programs the lookup tables for scaler units, T cells, coefficient and binary to residue conversions. The microprocessor of the main cell executes its microcode to generate control bits and also programs the address generation logic for coefficients and data memories.

The processor requires loading of all the RAM's in the tree structure prior to any processing of the data. The binary tree is based on a parallel and pipeline structure

which uses the output data of one stage as the address for the next stage, thereby eliminating separate data and address buses. Loading of the RAM's in the tree cell would normally require separate address and data buses which contribute to an increase in the number of connections in the system. A simpler scheme is proposed to reduce the wire complexity by first storing the addresses in the RAM's as shown in Figure 4.7. There will be an increase in the loading time but this scheme will eliminate the need for the separate address bus. The system will only require a data bus connecting the memories with the host computer. The host computer can load look up tables in any part of the memory of the tree cell. A controller from the host computer can enable the selected memory to load a particular look up table using bits 6-8 as shown in Figure 4.7. The loading of the look up tables in the tree structure is divided into two operations. The first operation will be to store a sequence of 0,1,2,.....31 in all the T cells which can be used as a 5 bit address. The second operation will utilize the addresses stored in the RAM's and control bits to load the actual lookup tables in the T cells. First of all address 0 to 31 is generated in processor P1 to P8 which are rippled through to tree cell 13 and 14. Cells 13,14 and bit 8 provide a 12 bit address which is used to load the look up table in cell 15. The second operation of the host computer is to load the actual look up tables in all the memories.

All the stages are loaded by rippling back to the first stage.

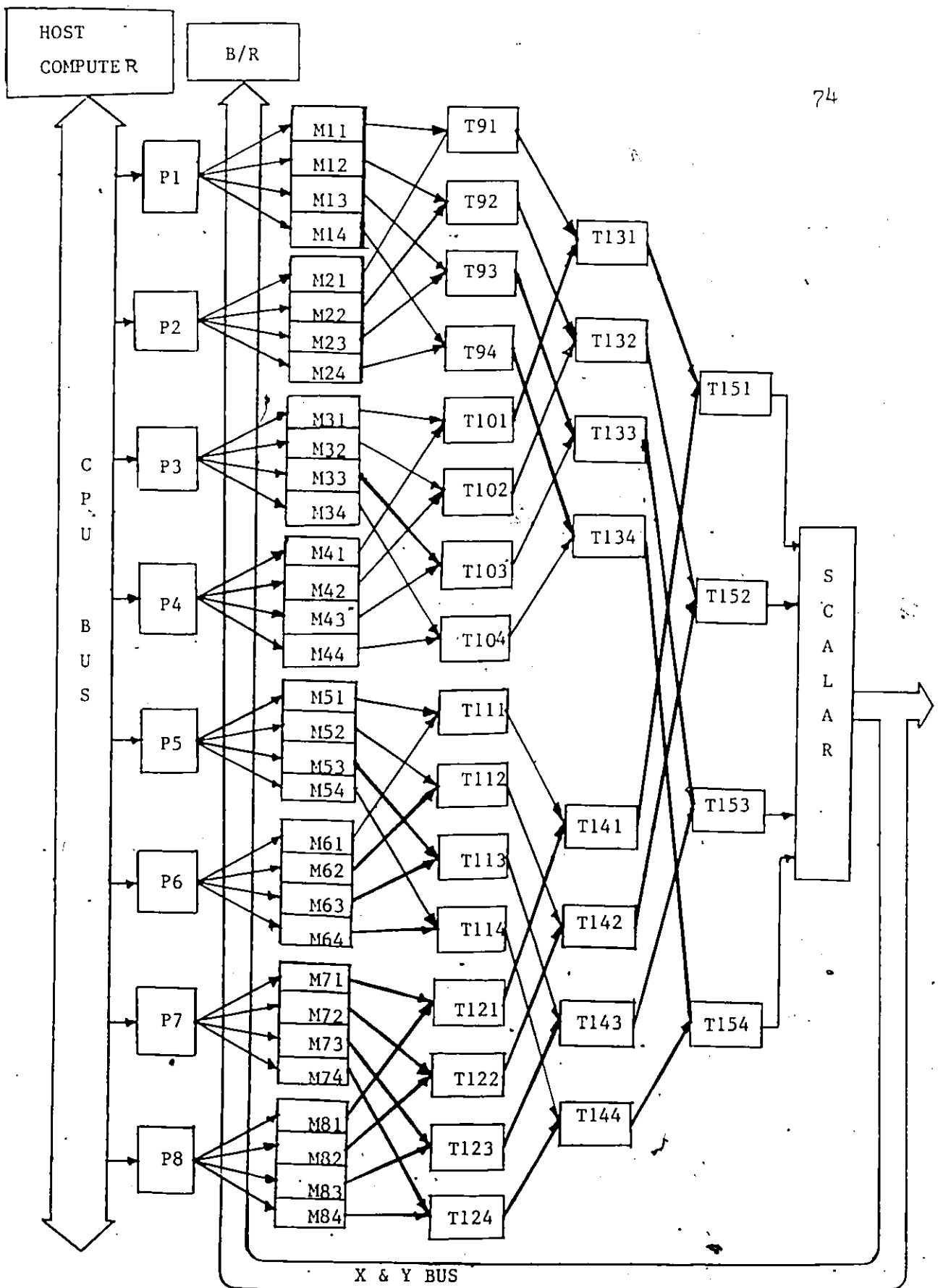


Figure 4.6: Host Computer Interface Interconnecting.  
Four Identical Binary Tree Structures

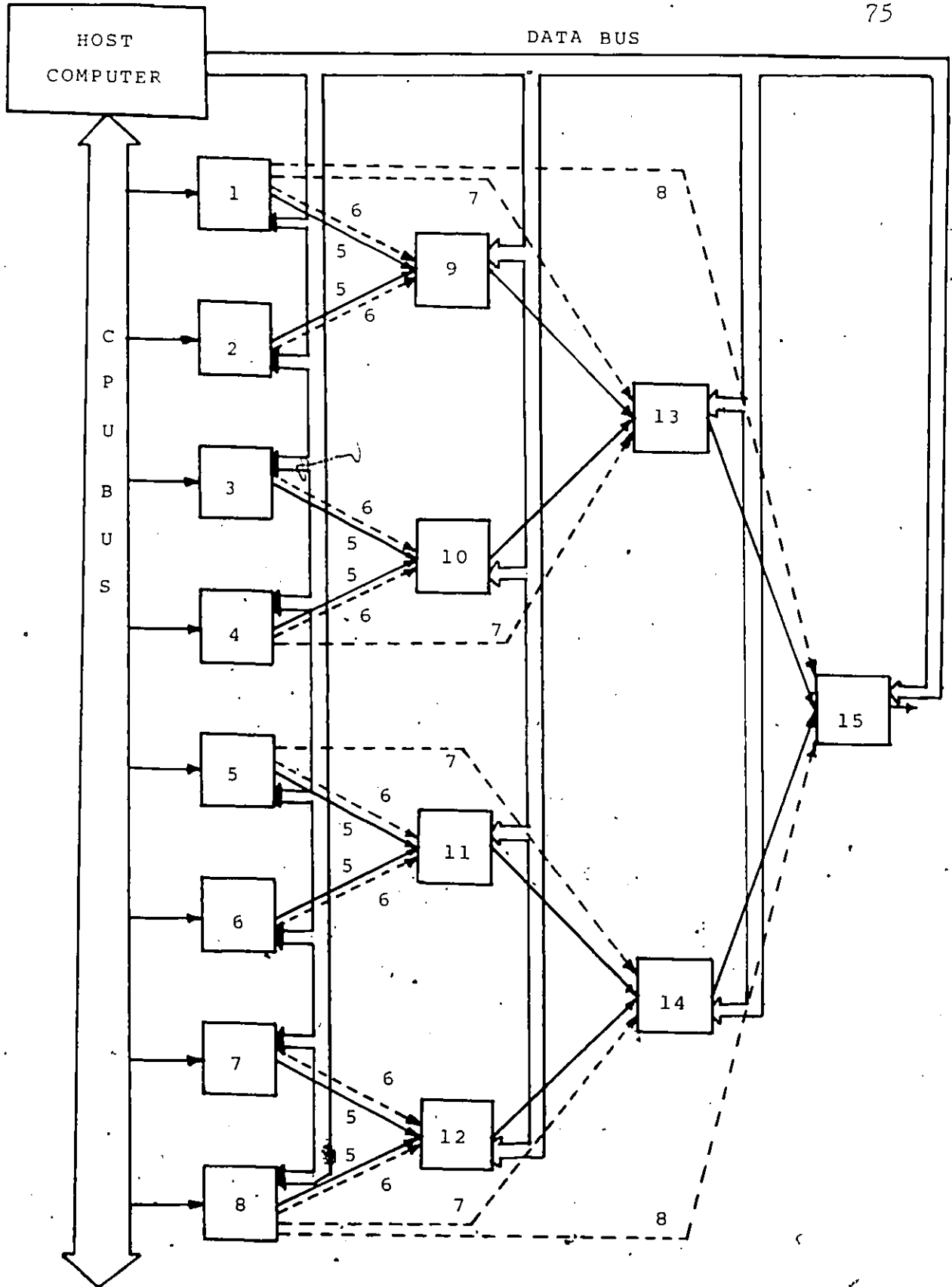


Figure 4.7: RAM Loading and Look-up Table Enable Interconnection

#### 4.6 OPERATION OF THE SIGNAL PROCESSOR

The previous example of a set of band pass filters is used to illustrate the operation of the signal processor. The above algorithm has six multiplications, two additions and three subtractions. Six base processors are assigned to perform multiplication between coefficient and the data as shown in Figure 4.8. The algorithm is shorter than the number of cells in the tree structure. First of all the arithmetic operations are mapped on the tree structure assigning base cell to multiply data and coefficients and tree cells are used to perform additions and subtractions. The unused cells are carefully disabled and they only pass the unchanged operands to their parent cells. The input data is converted into residues using a binary to residue conversion unit. The output of the four tree structures is scaled down using a scaler which also produces a binary output. The processors P1-P8 are responsible for cyclic address generation logic schemes for reading/writing data in the memories from the X & Y buses.

The host computer generates the lookup tables for all the cells in the tree structure and stores all the look up tables in the specific RAM's. The control is transferred to the operator to start processing the signals. Binary data is provided as an address to a binary to residue conversion unit and the four input residues are written in all the cell memories. The appropriate data, or delayed data, is combined

with the coefficients using the lookup tables in the base cell. The output operands are fanned out to the tree structure for further processing. The operands travel in an upward direction to the output units for data output. The process continues in real time performing the specified signal processing application without any communication bottleneck. There are 32 filters in this example and any one of the filter outputs can be seen at the output units. This example of the set of band pass filter is simulated using the architecture of the DFSP.

The processor has been simulated on a digital computer and the program listing is provided in Appendix C. Two programs have been written for the processor simulation. One develops all the look up tables while the other computes all the arithmetic and scaling operations. An arithmetic operation is performed by first forming an address with the two input numbers and then reading the result from memory. The computation is performed in four parallel paths and produces direct binary output. The example of the set of band pass filters is illustrated to check the validity of the structure by plotting the frequency response of the filter. The output of filters 7 and 28 are plotted, it can be seen that the filter passed the output of filter 7 and blocked for filter 28 and is shown in Figure 4.9.



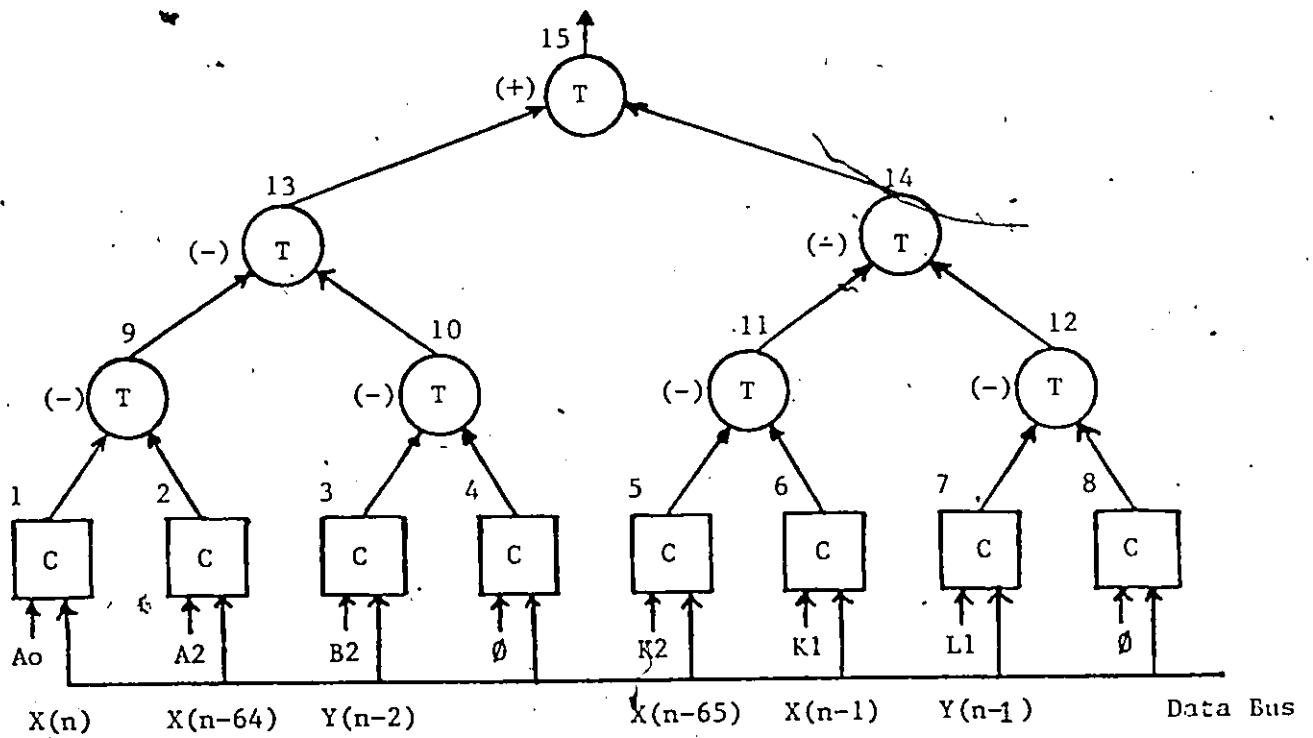


Figure 4.8: Binary Tree Structure

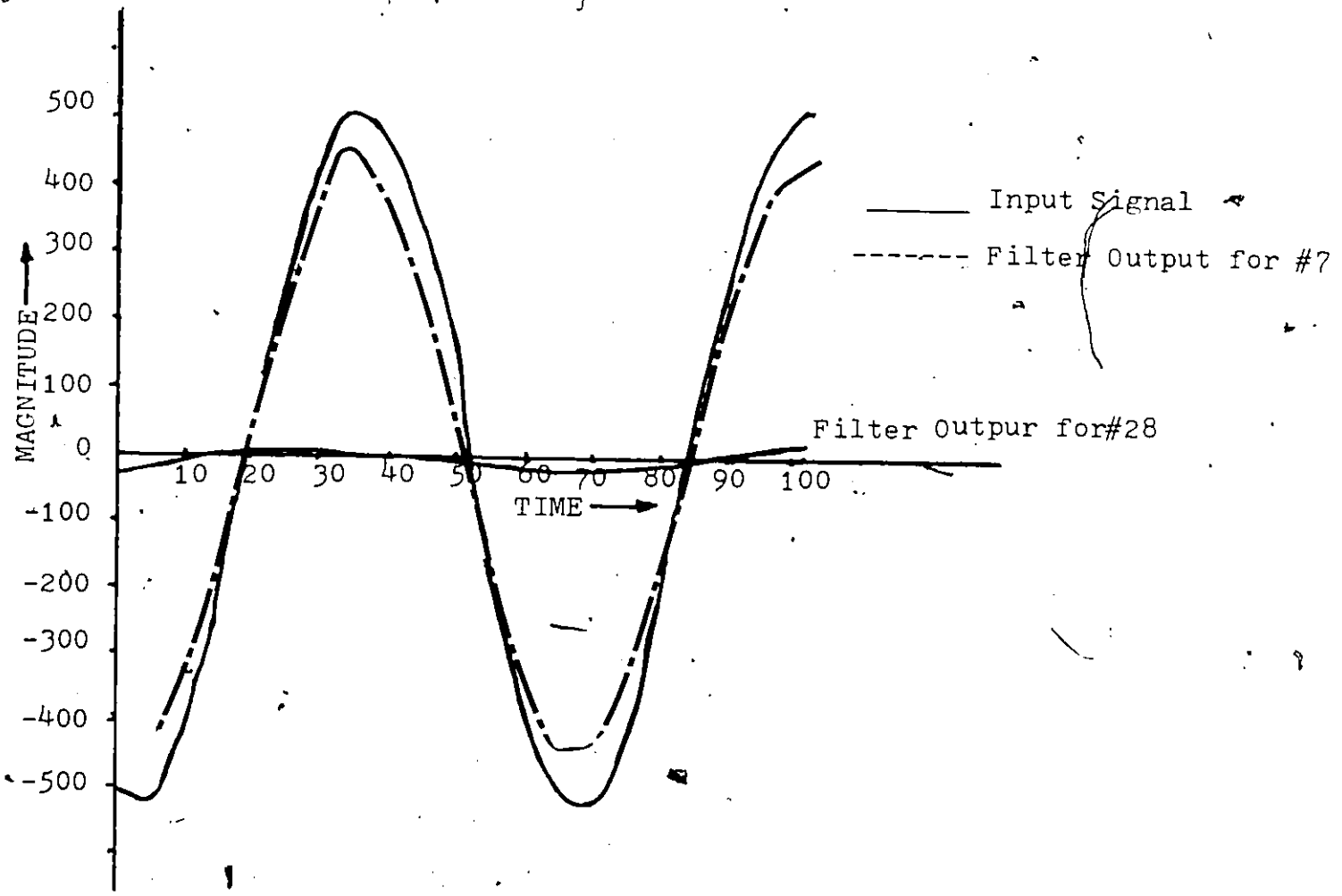


Figure 4.9: Filter Outputs for filter#7 and filter#28

#### 4.7 COMMENTS & SUMMARY

A new concept of a general purpose signal processor architecture has been defined and demonstrated to perform real time execution of wide variety of signal processing algorithms. The architecture is based upon the binary tree structure using a data flow approach to eliminate the central processing units required in general purpose machines. The processor exploits all the fast computational approaches of distributed, parallel and pipeline techniques and uses the residue number system for high speed signal processing applications. The execution of recursive, non-recursive filters and convolution for one and two dimensions can be performed easily. The processor proposed in this chapter is programmable and allows the user to modify the algorithms in software. Algorithms larger than the tree structure can be executed by repeated passes through the tree structure. The number of times a partial output is fed back to the tree structure depends upon the complexity of the algorithm. The hardware can be expanded with only software modifications being required.

The concept of general purpose data flow signal processor in this chapter provides the solution to many problems associated with conventional data flow architectures discussed in the literature. The following points are pertinent.

1. The overhead bits are eliminated by incorporating them with the coefficient memory. This can be com-

pared with the 200% overhead bits in other data flow machines.

2. The operands are transmitted in parallel, avoiding all the overheads associated with the packet serial communication protocol.
3. The approach is deterministic and throughput rate can be calculated.
4. The computation time of any arithmetic operation is reduced to the access time of the RAM type used in the system.

The above comments are pertinent to conventional data flow machines such as U Interpretor(8) etc. An added advantage of the RNS is to keep the operand size within 5 data bits and only a few control bits. The size of the operand allows parallel transmission and the system is also free from any communication protocol for transmitting the operands serially.

## Chapter V

### FIR FILTER IMPLEMENTATION

#### 5.1 INTRODUCTION

Finite impulse response digital filters offers some desirable properties of stable structures, less sensitive to quantization errors etc which are very useful in image and speech processing applications. The realization of FIR filters needs a high computation time because long sequences of the impulse response are required to control the frequency response of the filter. Their implementation requires large number of multiplications, and additions. Jenkin [24] utilized Residue Number System to increase the computational speed for implementing the FIR filters. With the advent of fast Fourier transform (FFT) algorithms, the implementation of FIR became very attractive due to its computational efficiencies. It has been established that transform techniques [20-23,25,26] having cyclic convolution property can be used to indirectly perform convolution. The characteristics of these transforms are such that the transformation of convolution in the time domain is equal to the term by term product in the transform domain.

The computation of the convolution sum using transform approach is normally faster than that performing using di-

rect approach. This chapter investigates both the approaches of computing the convolution sum using transform approach and the direct approach. It is shown that the direct computation of the convolution sum using the DFSP exhibits more advantages than the transform approach. First of all a discussion on the transform approach is presented in this section.

The approach of computing the fast fourier transform requires multiplications with the transcendental coefficients (sines and cosines) making exact computation impossible on a digital machine. At each stage the output has to be scaled down which introduces extra computational errors. An alternate approach for computing convolution is to utilize number theoretic transforms (NTT) [19-24] which exhibits a similar cyclic convolution property to the FFT but which are defined over finite rings. A brief review of NTT is presented in the next section.

## 5.2 NUMBER THEORETIC TRANSFORM

The Number Theoretic Transform (NTT) [19-24] has been discussed in detail by various researchers. Here the implementation of NTT and its selection of different parameters are discussed briefly. The general form of the transform pair is given by

$$\text{NTT: } X(k) = \sum_{n=0}^{N-1} x(n) \cdot a^{nk} \quad (5.1)$$

$$\text{NTT}^{-1} : x(k) = N^{-1} \sum_{k=0}^{N-1} X(k) \cdot a^{-nk} \quad (5.2)$$

$$k=0, 1, \dots, N-1$$

where  $a$  is the  $n$ th root of unity and  $n$  belongs to the ring/field. Unlike the DFT, NTT's do not allow arbitrary transform length and the maximum attainable length  $n$  depends upon the choice of the ring or field or . The implementation of NTT requires selection of number of parameters eg transform length, dynamic range, primes, generator  $a$  etc. The criteria for selecting different NTT parameters have been described in the literature [20-23]. It is assumed that a transform length of 128 and a dynamic range of 20 bits will be sufficient for most of the signal processing applications.

All the prime numbers can be divided into two groups.

$4n+1$  e.g. 1, 5, 13, 17, .....

$4n+3$  e.g. 3, 7, 11, 19, .....

The primes can be determined for a particular transform length using the following relation.

$$m = q \cdot 2^P + 1 \quad \text{for } 4n+1 \text{ primes}$$

$$m = q \cdot 2^P - 1 \quad \text{for } 4n+3 \text{ primes}$$

where  $p$  is 6 for a 128 point transform length and  $q$  is odd. Primes can be selected as 193, 449 and 191, and their repre-

resentation of numbers in bits are 7.592, 8.811 and 7.577 bits respectively. The primes 193,449 and 191 supports the transform length of 128 however their dynamic range is restricted to 8 bits. In order to increase the dynamic range NTT can be performed using RNS concepts of selecting more than one prime moduli. The operation for each moduli can be performed in parallel and the final result can be reconstructed either using the chinese remainder theorem or mixed radix conversion scheme.

The computational efficiencies in FFT can also be exploited for NTT and fast algorithm to compute NTT can be called Fast Number Theoretic Transform (FNTT). The main computational element in FNTT is known as butterfly. The structure of DIF type radix 2 butterfly is shown in Figure 5:1. This butterfly is accessed  $N/2 \cdot \log N$  times where  $N$  is the transform length. The input to butterfly is two input complex points and they are first added and subtracted and the result is then multiplied by the twiddle factors.

It has been shown previously [20,22-23] that the structure of butterfly requires more hardware for  $4n+3$ , as compared to  $4n+1$  type of primes. The objective of this work is to perform signal processing operations at a very high speed using the binary tree structure. Utilizing  $4n+3$  type of primes would also require more hardware and more computation time. The discussion here will be restricted to  $4n+1$  type of primes which can support a transform length of 128.



Two different structures of the butterfly are shown in Figure 5.2 for even and odd powers of  $a$ . The look up table implementation of the butterfly structure would require two stages and six binary operations.

We are interested in implementing arithmetic operations using  $1k \times 8$  RAM's of the binary tree structure. The realization of the butterfly structure on the DFSP has been studied. Since the DFSP utilizes look up tables to perform arithmetic operations, this approach will be used to implement butterfly via RAM look up tables. The implementation of an NTT butterfly [19,20,22-23] requires large prime moduli of 193,449 etc as discussed previously. Use of large moduli  $64 \leq m_i < 128$  would require a  $16K \times 8$  bit RAM or sixteen  $1k \times 8$  bit RAM's for any arithmetic operation. The utilization of a large number of memories, to implement a particular arithmetic operation does not seem to be a very efficient scheme. The technique of breaking up of large moduli into smaller sub-moduli can be utilized to implement on the tree structure. The only constraints on the choice of sub-moduli is that they should be large enough to contain the result of the operation modulo main modulus and should be relatively prime.

Jullien in his paper [19] showed that the multiplication of large moduli can be implemented using an index method if the modulus is a prime number. Jullien proposed that the the multiplication can be replaced by addition and can be

implemented using the sub-moduli approach for large moduli. The method has been described in the literature [19,20,22,23,27].

Consider the operation  $|X.Y|_{449} = Z$  and choose sub-moduli 30 and 31 which gives a composite modulus

$30.31=931 > 2.449$ . The block diagram for multiplication modulus 449 using sub moduli 30 and 31 is shown in Figure 5.3. The block diagram forms a tree structure using 7 cells. It is clear from Figure 5.3 that the proposed tree structure of the signal processor can very well adopt the algorithm of large moduli multiplications. The multiplication of  $(a-a')^n$  or  $(b-b')^n$  using sub-moduli approach will require all the 15 cells of the binary tree. The butterfly operations as shown in Figure 5.4 would require two out of four tree networks proposed in the DFSP architecture. The addition, subtraction operations of  $(a+a')$ ,  $(a-a')$ ,  $(b+b')$  and  $(b-b')$  would have to be provided by the base cells requiring extra hardware. These hardware modifications are necessary in order to accommodate one butterfly structure into two tree networks. Thus four tree networks can only accommodate two  $4n+1$  type of primes. Using the transform length of 128, two  $4n+1$  can be chosen as 193 and 449 giving a combined dynamic range of 16 bits. Large prime moduli of 641 and 769 can also be chosen giving a dynamic range of 19 bits. The selection of these moduli would need 6 bit sub-moduli requiring 4k of memory for one look up table and an operand length of 6

bits. This will restrict to one look up table storage in the T-cells.

Apart from the extra hardware requirements for the butterfly structure a complex control circuitry would be required. The control circuitry will have to keep track of the number of butterfly operation and its stages. The memory requirement of the base cell will have to be modified in order to accommodate the intermediate output data of the butterflies. It is obvious from this example that if we allow modifications in the base cells, the computation of NTT can be performed using the two moduli of 193, 449 with a dynamic range of 16 bits.

An alternate approach of performing direct convolution for the realization of FIR filters is studied, which is less cumbersome, does not require any hardware modifications, gives more dynamic range and is presented in the next section.



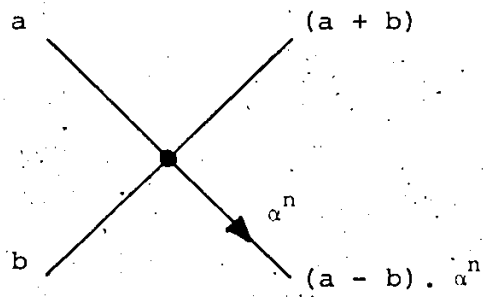


Figure 5.1 Radix 2 DIF butterfly

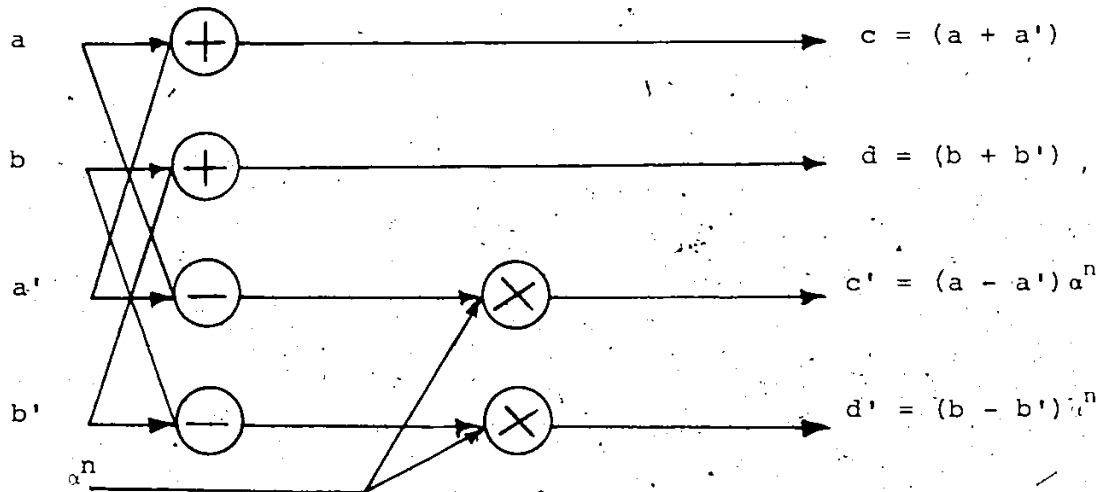


Figure 5.2(a) Butterfly for  $4n + 1$  prime ( $n = \text{even}$ )

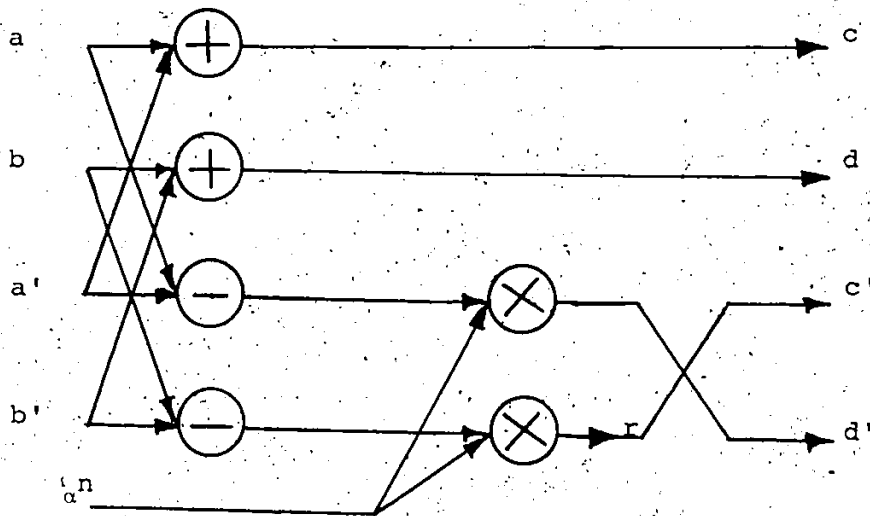


Figure 5.2(b) Butterfly for  $4n + 1$  prime ( $n = \text{odd}$ )

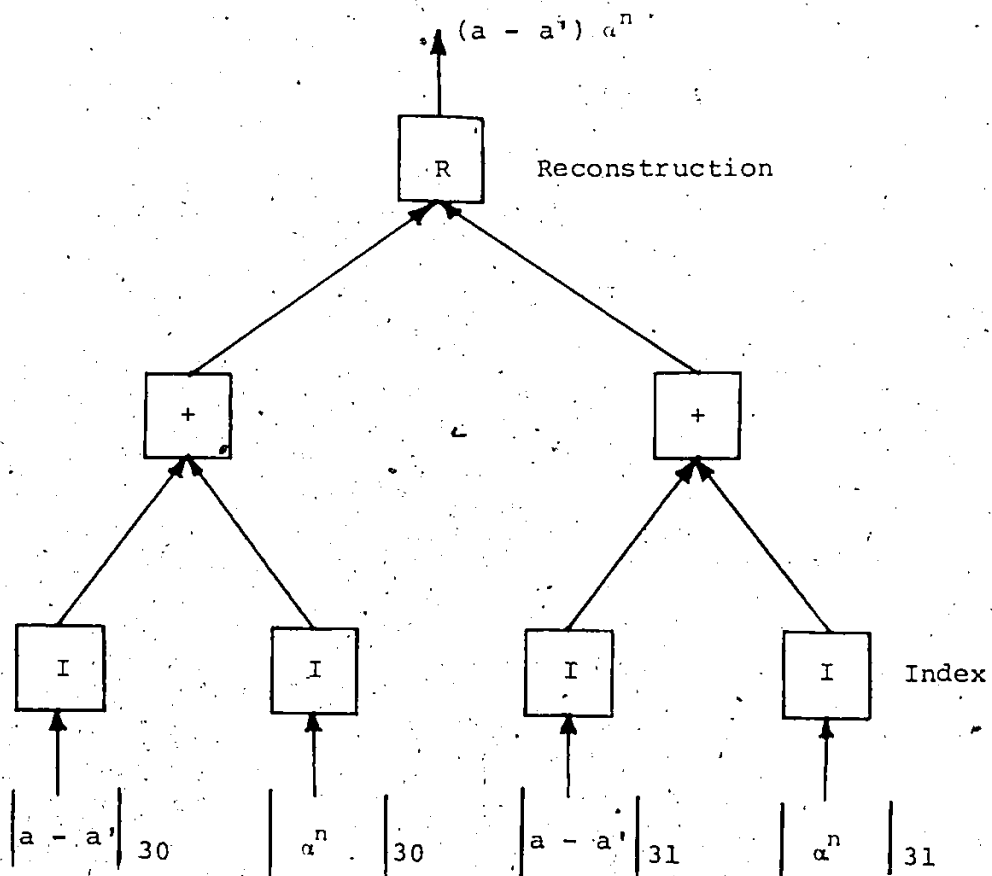
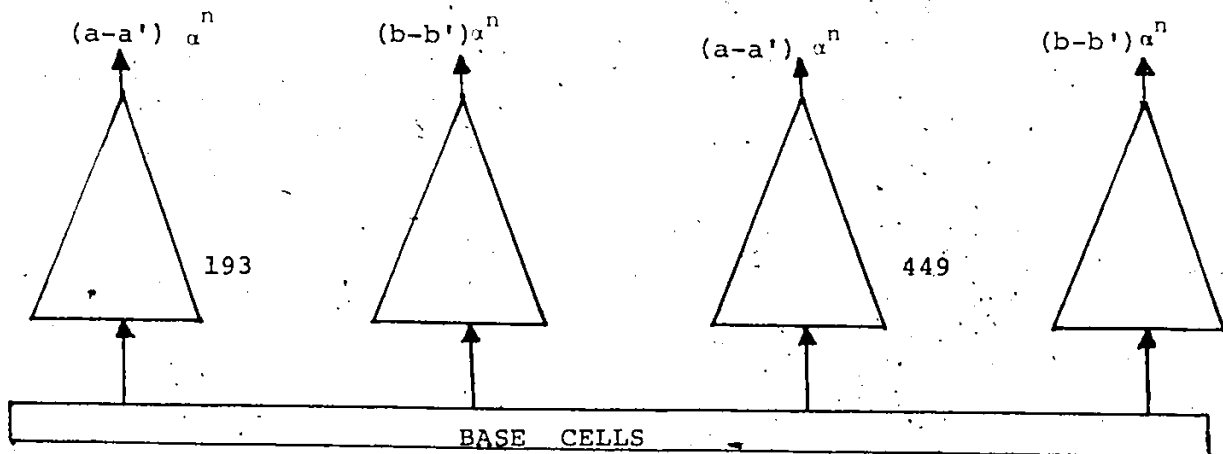


Figure 5.3 Twiddle Factor Implementation Using Large Moduli



$(a + a'), (b + b'), (a-a'), (b-b')$  are performed by Base Cells

Figure 5.4 Implementation of NTT on the DFSP

### 5.3 DIRECT CONVOLUTION

The convolution operation is the basic tool in many digital signal processing applications. The direct convolution of two sequences of length  $N$  requires  $N*N$  multiplications and  $(N^2 - 1)$  additions. It seems that this approach is computationally unfeasible. Normally an indirect approach of transform for computing the convolution is more popular. The transform approach beside its fast computational capabilities has several drawbacks.

1. FFT involves complex multiplications even for real multiplications.
2. It requires multiplications of transcendental coefficients sine & cosine making exact computation impossible on a digital machine. An alternate approach of NTT has its own limitations of large moduli etc. as discussed in the previous section.
3. Transform techniques perform cyclic convolution, in order to perform linear convolution zeros must be appended which will make the transform approach less efficient.

Many researchers have concentrated on developing some efficient algorithms for digital convolution [28,29]. The research in the area of parallel processing has opened many channels for performing direct convolution. The feasibility of performing direct convolution has been investigated using two examples. First of all an example of two dimension im-



age convolver is used and is presented in the next section. The second example is of complex digital signal processing which is separately discussed in Appendix B.

### 5.3.1 Two Dimension Image Convolver

The areas of robotic vision and image processing has many applications of real time filtering of image data. The digital convolution of two dimension finite duration sequences,  $x, h$  and of size  $(N*N)$  and  $(L*L)$  respectively is defined as

$$Y(m, n) = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} F(k, l) \cdot X(m-k, n-l) \quad (5.3)$$

where  $M \geq N+L-1$

An image enhancement example is considered here to illustrate the feasibility of the direct convolution for an image size of  $(128*128)$  and a filter kernel [30] of size  $(17*17)$ . The direct convolution would require  $144*144$  computations for each point in the output image. It can be visualized from the above equation that there are only  $17*17$  non-zero terms in each computation. If we can isolate all the non-zero terms, the computation will become more efficient and a maximum of  $17*17$  computations will have to be performed for each point.

The direct convolution is performed by first reversing the filter kernel and then sliding the reversed kernel on the image data. The inverted filter kernel slides on the image data from point (1,1) to point (128,128). The overlapped points of the filter and image are non-zero points and will require computation. There will be one multiplication for point (1,1)  $h_0.x_0$  and operation of  $h_1.x_0+h_0.x_1$  is performed for point (1,2). For the first computational row, only the first row of the image and filter will be multiplied and for the second row two rows of each filter and image data will be multiplied and their product will have to be added.

The computational process is graphically shown in Figure 5.5. Let (x,y) be the two coordinates of a point in the image. Let the filter kernel is in position (4,7), giving  $x=4$ ,  $y=7$ . It is clear from Figure 5.5 that there are 4 rows and seven columns overlap of the filter kernel and the image data requiring 28 multiplications and 27 additions. The 7 multiplications in each row can be performed simultaneously using 1-7 base cells and cell 8 can be reserved for partial outputs. There will be 4 passes, one for each row of the tree structure to compute the output point of (4,7). In order to compute point (4,8), (4,9) the filter kernel will be moved one point towards right and for points (5,7), (6,7) and (7,7) the filter kernel will have to be moved downwards.

In order to perform computation efficiently a scheme has been proposed. The execution is performed for row 1, computing from its point (1,1) to point (1,128) each by sliding the filter kernel to the right. For the computation of the second row the filter kernel is moved back to position (2,1) and all the points in the second row are computed. Similarly all the 128 rows can be computed. It is obvious from this discussion that for every increment in the second (y) coordinate of the point filter kernel is moved towards right and for an increment in the first (x) coordinate the filter kernel is moved downwards. The implementation of this algorithm is explained in the next section.

### 5.3.2 RNS Implementation of direct Convolution

The direct convolution algorithm requires two simple arithmetic operations of multiplications and additions. The filter coefficients and the image data is multiplied for all the overlapped points as shown in Figure 5.5. An addition is performed of all the products of the filter coefficients and the image data. The DFSP structure is used to implement this convolution algorithm. The DFSP has four binary tree networks having 15 cells in each network. Four five moduli of 27,29,31 and 32 giving a dynamic range of 20 bits are chosen. This dynamic range of 20 bits is sufficient for most of the signal processing applications.

The number of arithmetic operations are more than 15 in most of the computations of this filtering process. The algorithm will be divided into different sections which can be executed in one pass of the binary tree network. The partial output of the previous section is fed back with the computation of the next section of the algorithm. The number of passes depends upon the required number of arithmetic operations for a particular image point.

There are only eight base cells, 7 tree cells in the binary tree network. The base cell can be used for multiplications of image data and the filter coefficients. The tree cell will be used to sum their products. Seven base cells are allocated for performing multiplications and cell 8 is reserved for passing the partial output.

With this configuration the points 1-7 in first row will require only  $I$  cycle and point 8 to 14 will require  $2I$  cycles where  $I$  varies from 1 to 17. Similarly the rows 17 to 128 will require 17, 34 and 51 cycles in their sections. This division of image facilitates the allocation and scheduling of the data to the tree structure and helps in the control of the image and filter data management.

The direct convolution is performed row by row. The computation in each row will first be performed from point 1-128, 8-128 and 15-128 to allow all the  $17 \times 17$  computations for each output point.

The number of multiplications required for each point in the output image depends upon its location in the image. The computational breakdown of the image is shown in Figure 5.6. The image has been divided into six sections and the number of passes required for each point in different sections are also shown in Figure 5.6. This breakdown of the computation is based on the number of cells in the tree structure and it will be different if the number of cells in the tree network is increased or decreased.

It is interesting to note that 4k data memory in the base cell will be sufficient to store first 32 rows of the image. After computing  $(32-17)=15$  rows the memory can be overwritten in a cyclic fashion to bring new row of image data. The coefficient memory is sufficiently large to store  $17*17$  filter coefficients and they can be preloaded at the initialization time. The data and filter coefficients to the base cells can be fed by just incrementing the address of their respective memories. The address will be reinitialized to a certain value dictated by the data flow graph as discussed in the chapter 6.

In terms of hardware it can be seen that the design of the DFSP is highly modular and identical. Each tree cell requires a 4K random access data memory and the base cells will each require three 4K RAM's, microprocessor and hardware for address generation logic. The sections of the binary to residue conversion and scaler or mixed radix conver-

sion unit can be designed using RAM's. In addition to this some control circuitry will also be required. The structure can be interfaced to a host computer. If we utilize Toshiba TMM 32 RAM with the access time of 35 n sec the instruction rate can be achieved approximately 428 MOPS. The hardware requirements for the DFSP has been shown in Table 1.

$$Y(m,n) = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} F(k,l) \cdot X(m-k,n-l)$$

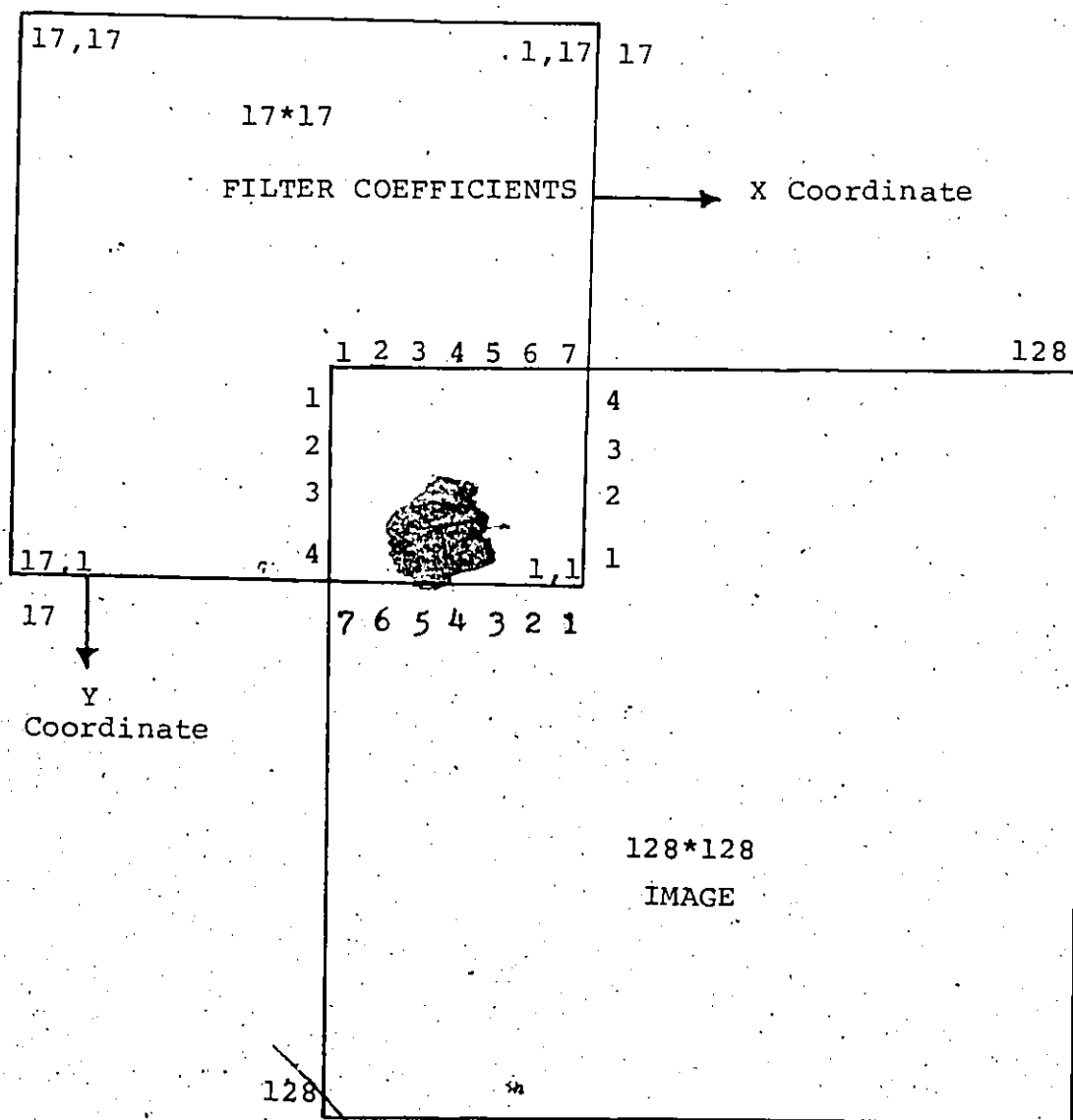
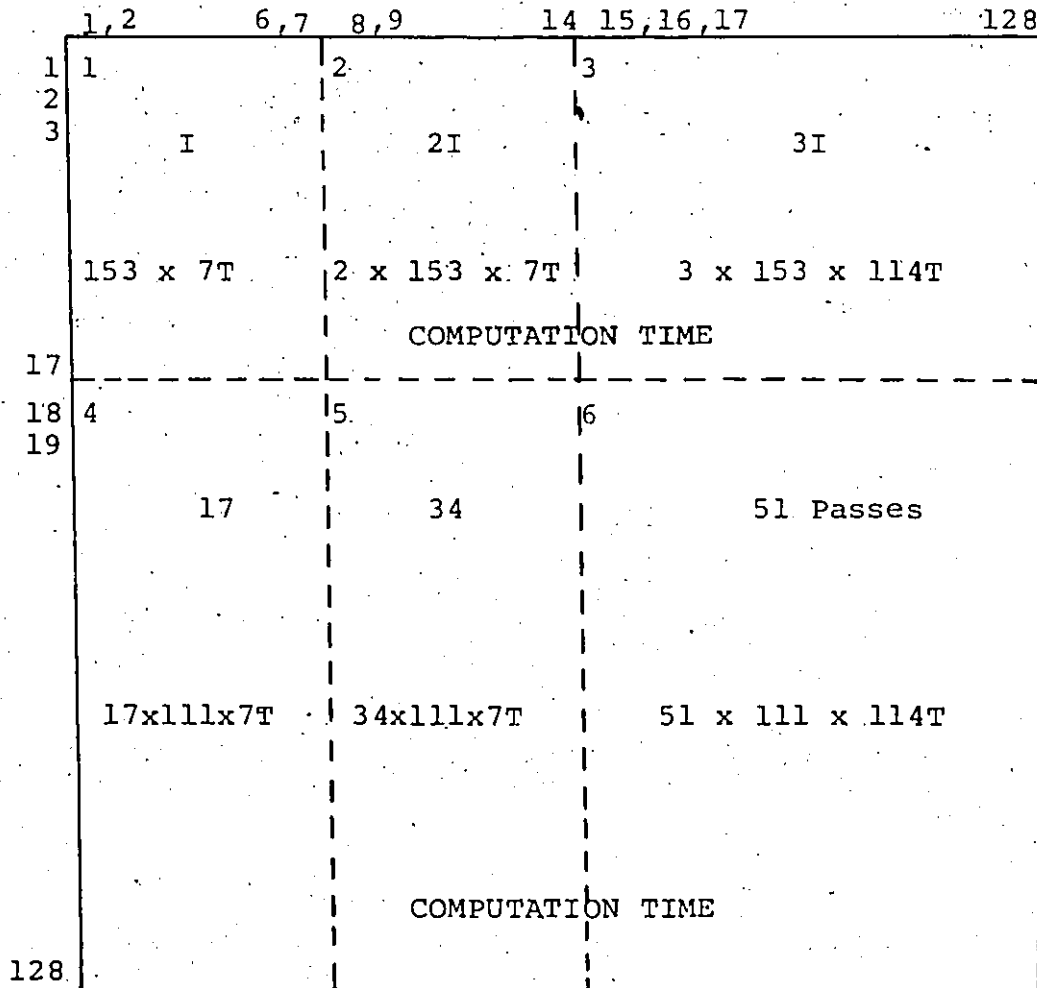


Figure 5.5 2D Direct Convolution Algorithm



where  $1/T$  is the pipeline throughput rate

Figure 5.6 Computation Time for 2D Convolution



HARDWARE UNITS	PACKAGE	# OF PACKAGES	SIZE
TREE CELLS	RAM	$4 \times 15 = 60$	4K*8
DATA MEMORY	RAM	$4 \times 8 = 32$	4K*8
COEFFICIENT MEMORY	RAM	$4 \times 8 = 32$	4K*8
MICROPROCESSORS		$1 \times 8 = 8$	16 bit
SCALER UNIT	RAM	$1 \times 8 = 8$	4K*8
R/B UNIT	RAM	$1 \times 4 = 4$	1K*8
BUFFERS		72	8 bit
TOTAL PACKAGES			= 216

Table 5.1: Hardware Requirements of  
the Data Flow Signal Processor

#### 5.4 THROUGHPUT RATE CONSIDERATIONS

The computation time for direct convolution of an image of size (128\*128) with a filter kernel of (17\*17) is calculated as follows. The image has been partitioned into six computational sections and the cycle time required to perform computation for each section is shown in Figure 5.6.

The Computation time for section 1	=1071 T
" " " " " 2	=2142 T
" " " " " 3	=52326T
" " " " " 4	=13209T
" " " " " 5	=26418T
" " " " " 6	=645354 T

---

The total computational time =740520 T

Where T is the pipeline throughput rate.

If we choose Toshiba TMM 32 RAM with an access time of 35 n sec ( $T=35$  n sec). The computation time of the image will be 25.91 m sec. Assuming a video rate of 30 picture/sec or 30 m sec for one picture frame, then the above computation time of 25.9 m sec is well below the video rate.

The throughput rate is also calculated for the transform technique as discussed in section 5.2 for comparison with that obtained for the direct convolution approach using two options. First option is that the DFSP hardware can accommodate 1 D butterfly structure with required extra hardware in the base cell. It is assumed that the base cell will be

able to perform  $(a+a')$ ,  $(a-a')$ ,  $(b+b')$ , and  $(b-b')$  operations. The second option is that the existing hardware is not modified. The operations of  $(a+a')$ ,  $(a-a')$  and  $(a-a')a^n$  will be performed by the tree structure. The example of 2 D convolution for image filtering process has been considered. The image of size  $(128 \times 128)$  is convolved with an edge enhancement filter of size  $(17 \times 17)$ . In order to perform 2 D NTT transform using 1 D NTT butterfly, first of all the transform of all the rows is computed and then the transform for all the column is performed. It is assumed that the filter coefficient will be stored in the transform domain.

The time required to perform a transform of an  $M \times M$  image with hardware modifications is given by

$$\text{computation time} = T \cdot (M^2 / 2) \cdot \log_2 M$$

where  $M=N+L-1$ , image size= $N \times N$ , filter size= $L \times L$

where  $1/T$  is the pipeline throughput rate and

Four convolution operations will be required to filter an image of size  $(N \times N)$ .

$$\text{Computation time for transform} = 8 \cdot (128^2 / 2) \log_2 128 = 458752T$$

$$\text{Multiplication time for two transform} (128 \times 128) = 278520 T$$

$$\text{Total time for filtering process} = 458752T + 278520T$$

$$= 25.8 \text{ msec @ } T=35 \text{ n. sec}$$

The time required to perform a transform of an  $M \times M$  image with existing hardware is given by

$$\text{computation time} = T \cdot (M^2 / 2) \cdot \log_2 M$$

$$\text{Computation time} = 3.8 \cdot (128^2 / 2) \log_2 128 = 1376256T$$

Multiplication time for two transform(128\*128)=278520 T

Total time for filtering process=1376256T+278520T

=57.9 msec T=35 n. sec

The two execution times one from the direct convolution approach and the second one for the transform approach with hardware modifications are approximately the same. The third execution time of 57.9 m sec for the convolution process with the existing hardware is higher than the direct convolution execution time. The direct convolution is simpler than the transform approach in terms of arithmetic operations and the program control. The transform technique would either require extra hardware or more execution time, and a complex control program for controlling the butterfly stages. The direct convolution approach utilizes four moduli of 27, 29, 31 and 32 giving a dynamic range of 20 bits. The transform approach will utilize two moduli of 193, and 449 giving a dynamic range of 16 bits. This leads to a conclusion that the filtering process can be performed easily with the direct approach without sacrificing the execution time.

### 5.5 COMPUTER SIMULATION

The algorithm of the two dimension direct convolution as discussed in the previous section for an image filtering process has been simulated on SEL 27/32 mini computer. The simulation software consist of a host computer loading pro-

gram and a control program. The host computer loading program generates all the multiplications, additions look up tables for all the four moduli. These look up tables stored in memory will be used by the control program at the filter processing time. The control program is responsible for reading the input image, filter coefficients, and performing computation of the algorithm. This program also stores the output image on disk for display at a later stage.

The control program performs filtering process from row 1, and computes all its 128 points. The image data and filter coefficients are accessed from their respective memories form an address of the multiplication look up tables stored in the base cells. The output of two multiplication tables is combined to form an address of the addition look up table. The process of forming addresses and reading results from cells moves data in an upward direction until all points in a row are computed. The control program writes the output data on disk and process continues for all the 128 rows.

An image of a machine part is used as an input data to the control program. The original image and the filtered images are shown in Figure 5.7. It can be seen that the edges of the filtered image are enhanced since an edge enhancement filter [30] was used for convolution purposes. This filtered image is compared with another filtered images which was filtered using a transform technique as shown in Figure 5.7.

It is obvious from the two images that they both enhance the edges of the image.

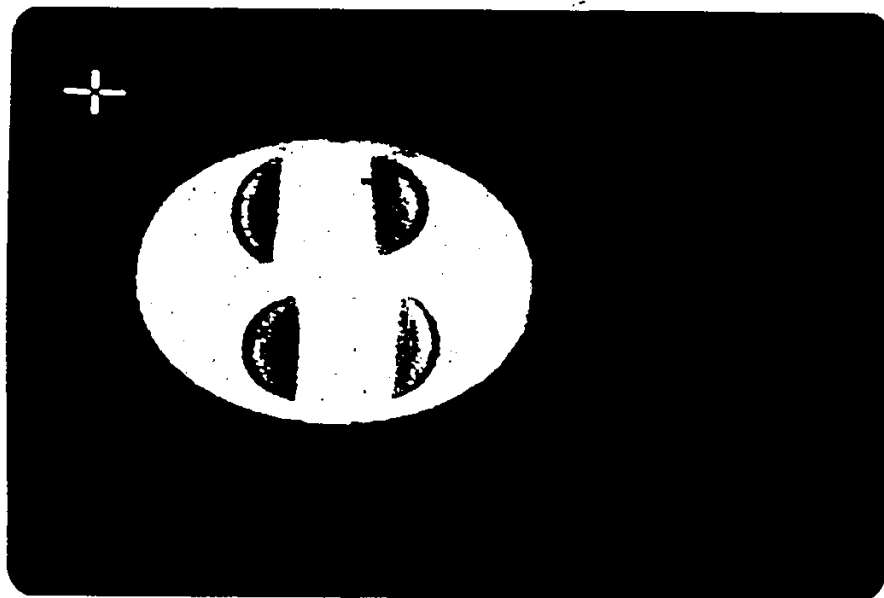


Figure 5.7a: Original Piston Head Image (128\*128)

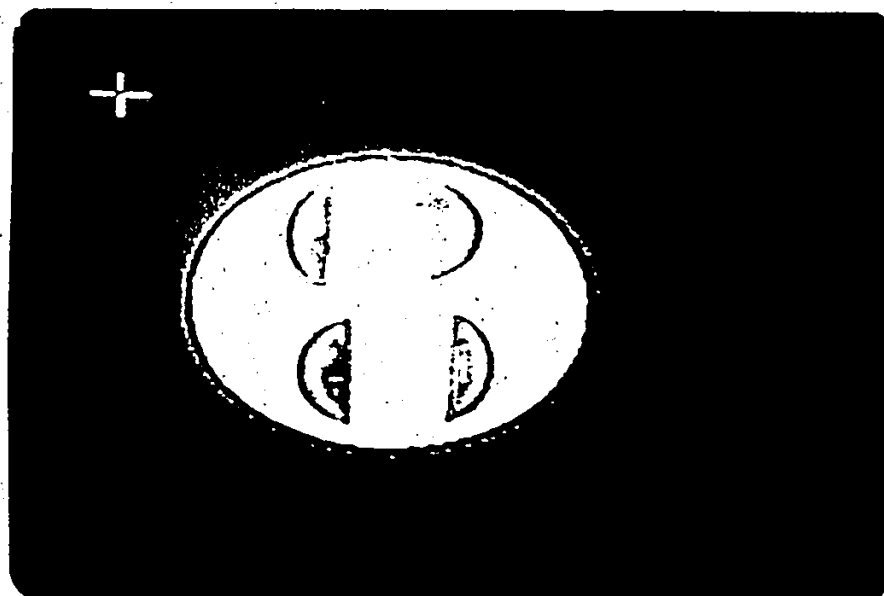


Figure 5.7b: Filtered Image using Direct Convolution  
(RNS Arithmetic)

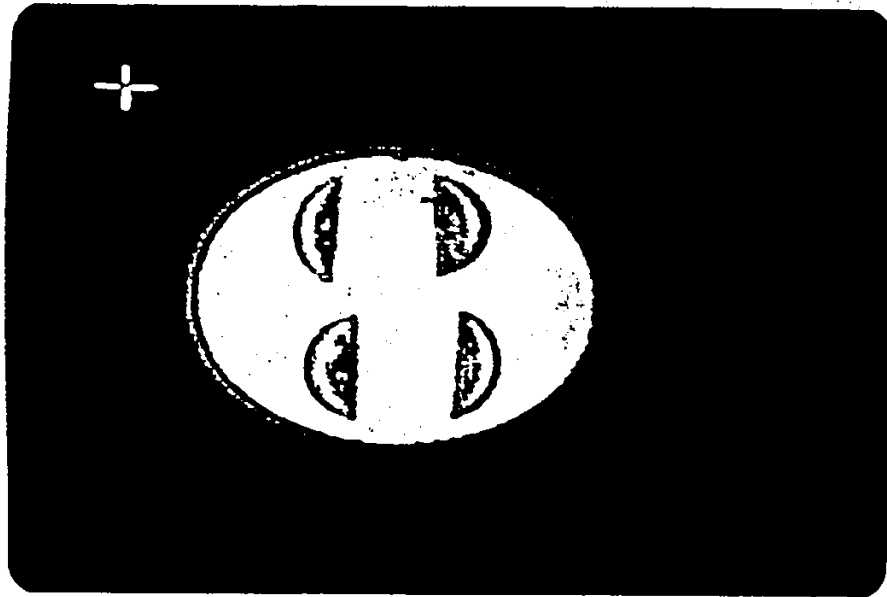


Figure 5.7c: Filtered Image using Direct Convolution

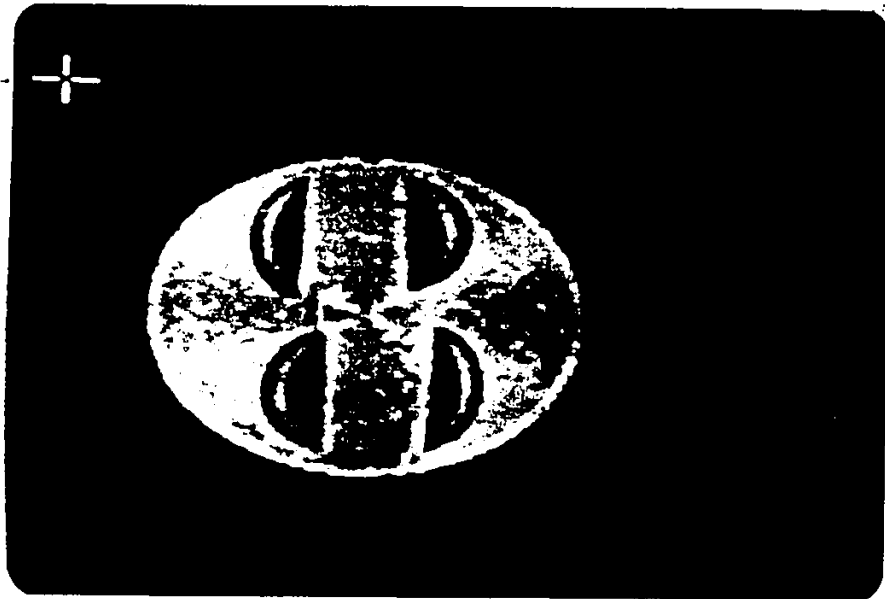


Figure 5.7d: Filtered Image Using Transform Approach



## 5.6 SUMMARY

Finite impulse response filters have many advantages which are very useful in speech and image processing applications. Two approaches of direct and indirect convolution were discussed. The use of cyclic convolution property of the fast Fourier Transform is very attractive to perform convolution using FFT. The transform approach offers faster computation of the convolution but also suffers with many drawbacks. An alternate approach of using the Number Theoretic Transform was discussed for performing convolution operations which performs exact computation but imposes many restrictions on the choice of its parameters. The NTT butterfly for a transform length of 128 utilizes large moduli multiplications utilizing two out of four tree networks for its implementation. Moreover some extra hardware would also be required in the base cells. The NTT approach uses two moduli of 193 and 449 giving a dynamic range of 16 bits with an execution time of 25.8 m sec. It was also shown that the transform approach with the existing hardware would require more execution time.

A direct approach of using the convolution sum was proposed for two dimension direct convolution, which utilizes only the non-zero terms for efficient computation. The approach of considering non-zero terms in the convolution sum is not evident from normal considerations and results in a lower computation time and therefore it is an important ap-

proach. An example of filtering an image of size  $128 \times 128$  with an edge enhancement filter of size  $17 \times 17$  was used to demonstrate the validity of the algorithm. The algorithm was simulated and an image with enhanced edges was obtained. The transform approach was also used to perform convolution and similar image with enhanced edges was obtained. The execution time for the filtering process was 25.9 m sec for the two approaches, which is below the video rates. The direct convolution approach gives more dynamic range, simple control circuit, requires only multiplications and addition operations with lower execution time than obtained from the transform approach.

## Chapter VI

### PROGRAMMING CONCEPTS OF THE NEW DATA FLOW SIGNAL PROCESSOR

#### 6.1 INTRODUCTION

The conversion of an abstract algorithm into a program which can be run on a particular computer is called programming. Over the years the programming has been done in conventional languages such as Fortran, Basic, Assembler, Pascal and Cobol etc. These languages may be different but their style remains the same as that of Von-Neumann model for executing programs sequentially. With the advent of data flow computers for fast parallel computations, there is a need to change the style of programming in order to accommodate parallel computing. Conventional programming languages are not suitable for programming these fast parallel machines. Most of the programs do have some sort of parallelism which can be exploited for faster computation. Efforts are being made to detect parallelism hidden in the program. There are numerous machines for performing parallel computation and there is no standard language which can be described as a parallel language.

Recent work in the area of data flow computers has generated considerable interest in parallel programming

[7-12,35-38]. There are a number of ways to express and detect parallelism and different researchers are exploiting the programming of data flow computers. Four alternatives are described here for parallel programming. First of all data flow graphs [35-38] can be used to express parallelism, describe the processor interconnections, node functions and data queues etc. Once a data flow graph is drawn it can easily be encoded into the machine language of the processor.

The second option is to write a program in Fortran, or some other proper language and use a compiler to uncover the parallelism. The main advantage of this approach is that the vast number of scientific programs written in Fortran can be recompiled and will be able to execute on data flow machines. The difficulties with this approach are that the compiler will be very complex, moreover the compiler may not be able to uncover all the concurrencies hidden in the program. There may be some side effects provided by the language which will make it difficult as the program can indeed be executed concurrently. The third option is to use a functional languages such as Lisp and Backus's Functional programming language for programming data flow computers.

The last option is to define a new data flow language which can be designed specifically for data flow machines. The main advantage with developing a new language is that it will allow a general form of parallelism. The programmers

will be motivated to learn a new language with a new philosophy which will assist them in their reasoning about programs. The examples of these data flow languages are Id, Lau and Val [35-38]. Data flow graphs are utilized to demonstrate new concepts for designing software tools for the DFSP and they are discussed in the following section.

## 6.2 DATA FLOW GRAPHS

Graphic models have been used in different areas of electrical engineering and control theory and digital filters are often represented graphically. The signal processing algorithms are data driven and data independent and it is easy to show graphically the flow of data and various functions which will be performed. The DFG's represent three types of information as follows:

1. They describe various processors and their interconnection topologies.
2. They specify functions of the nodes and their input output data queues.
3. They provide a command program which will maintain the execution of a particular algorithm.

The DFG is used to demonstrate the programming of the DFSP. The architecture of the DFSP is based on the binary tree and has been described in chapter 4. The DFG is very close to the hardware so its structure will also have a binary tree shape. The structure of DFSP, its various cells

and its interconnections are fixed. In order to execute a particular algorithm concurrently, the algorithm should be scanned to recognize parallelism. The algorithm can then be partitioned and allocate cells to perform various arithmetic operations.

A flow graph consists of nodes and arcs; nodes of the graphs are the cells which can perform arithmetic operations and arcs represents the data paths. There are two input and one output arcs of each node which bring the two operands and carry the output operand to other nodes. The input to the tree structure is through the base cells and the output can only be obtained from cell 15 as shown in Figure 4.1. Similarly the DFG will have the input data queues at node 1-8 and output queue at node 15. The interconnections of the intermediate nodes are fixed by the tree structure and they receive input data from their child nodes and forward output to their parent nodes. The functions of the nodes are prestored in the look up tables as described in section 4.3. The data which appears on the input arcs becomes the address of the data stored in the cell and output is produced. One of the several properties of the data flow graphs is that it can clearly illustrate concurrency in a program. Input data queues will normally consist of filter coefficients and data values. These input queues will be managed by the base cell nodes to maintain proper execution of the DFG. The output data can be forwarded either to the

output units or partial output are fed back to the base cell for further processing.

The data flow graphs discussed above can be incorporated into an interactive program which will be running on the host computer. The interactive program will require the input data for a particular algorithm either in the form of an input module or typed in operation by operation. The interactive program on the host computer will generate the information for assigning the function for the cells, and input output data queues. The algorithm for the interactive program has been described in the next section.

### 6.3 ALGORITHM FOR INTERACTIVE PROGRAMMING ON DESP

An application program can be written on the host computer in an interactive environment. The host computer accepts the input program, analyzes the program, allocates operations to various cells and generates the look up tables etc. These objectives are achieved with the help of flow graphs which are incorporated in the allocation program. The algorithm may not be of the same size as the tree structure. The program will determine if the application program algorithm can be executed in one pass of the tree structure. If the algorithm is smaller than the tree structure some of the cells will be disabled and will simply forward the operands. The allocation procedure divides the algorithm into sections if the application program is larger than the tree structure

then the program will be executed in more than one pass of the tree structure. The sections are individually mapped on the tree structure for the execution of the operands. The mapping of these sections is organized in such a way as to avoid any delay in the pipeline and incorporate proper inputting of the partial outputs. The interactive program which resides in the host computer is responsible for receiving the required information and performing the following functions.

1. Receive the input operands and their operations to be performed either individually or in the form of an input module.
2. Divide the algorithm into operations which require two external operands and operations which are performed on the partial outputs.
3. Divide the algorithm into sections in order to map one section at a time onto the tree structure.
4. The operations which require external data are mapped on the base cells, and partial product operations are allocated on the tree cells for each section.
5. The sections are executed on the tree structure avoiding any delay due to pipelining and incorporating partial outputs of other sections.
6. The various arithmetic operations are specified for all the cells. The individual task allocated to base cells provides proper input data.



7. The host computer generates the look up tables and stores in the cells. The program for the base cells is also stored in its memory.
8. The control is transferred to the operator after initializing the processor.

The above mentioned operations are performed using two algorithms which will be executed on the host computer. One of them is the input data algorithm which is responsible for receiving all the information regarding the particular signal processing algorithm. The second algorithm will be for an allocation program which will use the information provided by the previous input data algorithm and perform various other functions. The two algorithm for the two programs are described as follows:

### 6.3.1 Input Data Algorithm

```

/* input data algorithm */
  initialize the number of arithmetic operations
  to zero.
  while
  accept two input operands.
  accept arithmetic operations to be performed
  if more arithmetic operations
  then specify the arithmetic operation to
  be performed with the next operands.
  else stop
  end if

```

```

increment the number of arithmetic operations.
end while
end /* input data algorithm */

```

The second algorithm requires the first one as an input data, the allocation algorithm is described in the next section.

### 6.3.2 Allocation Algorithm

This algorithm performs a number of operations and is described as follows.

```

/* allocation algorithm */
begin
  separate arithmetic operations
  number of operations (N) requiring two
  external operands.
  operations which are performed on the
  partial output operands.
  if  $N < 8$ 
  then algorithm will require one pass
  of the tree structure.
  assign operations to the base & tree cells.
  else  $(N/7+1)$  passes will be required.
  divide the algorithm into  $(N/7+1)$  sections.
  assign cell 1-7 for external operand operations
  to tree cells.
  map each section on the tree structure.
end if

```

```
assign arithmetic functions for all the cells.  
generate & store look up tables.  
generate & store control program in base cells.  
transfer control to operator.  
end /* allocation algorithm */
```

The above two algorithms can be translated into a program which can be executed on the host computer in an interactive environment. This approach is transparent and the programmer does not have to know the internal architecture of the signal processor. An example of an image processing filter is used to explain the various flow graphs which will be generated internally for real time programming of any algorithm.

#### 6.4 EXAMPLE 1. TWO DIMENSIONAL CONVOLUTION

An example of a two dimensional convolution for an image of size  $128 \times 128$  with a filter kernel of size  $17 \times 17$  is considered and the algorithm has already been defined in section 5.3.2. This example is used to illustrate various steps of the interactive program which will be performed for its execution. A manual allocation of the functions for various cells and generation of I/O queues are shown. One of the objectives of this example is to show how communication bottleneck can be avoided.

The algorithm of this example is larger than the tree structure and the number of arithmetic operations to be performed varies as shown in Figure 5.6. This algorithm will

be executed in sections and will require partial outputs which will be fed back to the tree structure. In order to achieve maximum throughput rate or minimum execution time for any algorithm an attempt has to be made to keep the pipeline full. There are four stages in the tree structure and it will take 3 cycles for a partial output to come out of the pipeline, This will introduce 3 extra delays for the rest of the algorithm to obtain proper feedback of the partial output.

A scheme has been suggested to overcome this communication problem. First of all the computation is performed row by row. The computation of all the 128 points will be performed in each row and their partial outputs will be stored in FIFO fashion in Cell 8. The computation will start from point 1 to 128 with their partial outputs stored in cell 8, this will be repeated until all the required feedback passes are complete.

The tree cell has only 8 base cells, we utilize 7 base cells to perform multiplications and we keep cell 8 to pass the partial output as the feedback which is added in the parent cells. So for the first row we perform computation once from point 1 to 128, 8 to 128 and 15 to 128. After computation of every row the cell 8 memory is over written to store the partial outputs for the next row and computation proceeds for all the 128 rows. This approach keeps the pipeline full and avoids any idle cycles. The output of the

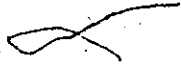
tree cell in the last computation process is directed to the output unit for storing on disk for later use.

Two data flow graphs are used to demonstrate programming of the DFSP as shown in Figure 6.1 and Figure 6.2. One of the DFG shows various cells, their interconnections and functions. Cells 1 to 8 can be assigned as multipliers which will require image data and filter coefficients. Cells 9 to 15 can be used as to add the outputs of cell 1 to 8 in a tree structure. The function of the cell 8 is redefined and is used to pass the partial output to cell 12 and the multiplication with unity does not alter the data which will make it easier to load the multiplication tables in cells 1 to 8 simultaneously at the initialization time. Cell 8 will have one operand as partial output feedback from cell 15 and the second operand of unity, since the data remains unchanged when it is multiplied by unity. The distribution box at the output of cell 15 distributes the data either to the output unit or as a partial output to cell 8 depending upon the control signal provided from the base cells.

The second DFG shows the command program for the input and output data queues as shown in Figure 6.2. The algorithm for the command program is identical for cells 1 to 7. The DFG starts computation with the first row and checks the number of superimposed rows between the image data and the filter kernel. This is achieved by the decision box. There will be  $I$  or 17 computations if  $I$  is less than 17 or great-

er than 17 respectively. The DFG has three sub sections which perform computations from point 1 to 128, 8 to 128 and 15 to 128 respectively for every row. The computation continues by checking for I less than 128. The processor stops after computing all the 128 rows.

The execution of this algorithm was shown in Figure 5.5. The data points and the coefficients are stored in the data and coefficient memories of the base cell respectively. The control flow shown in Figure 6.2 can be incorporated in the base cells to feed the proper values of the image data and the filter impulse response. This flow graph algorithm has been simulated using Sel 32/27 and output images were shown in Figure 5.7. The simulation programs run smoothly without any communication problems. This example and its two flow graphs can be used as a model to write an interactive program which can be generalized for a wide variety of signal processing applications.



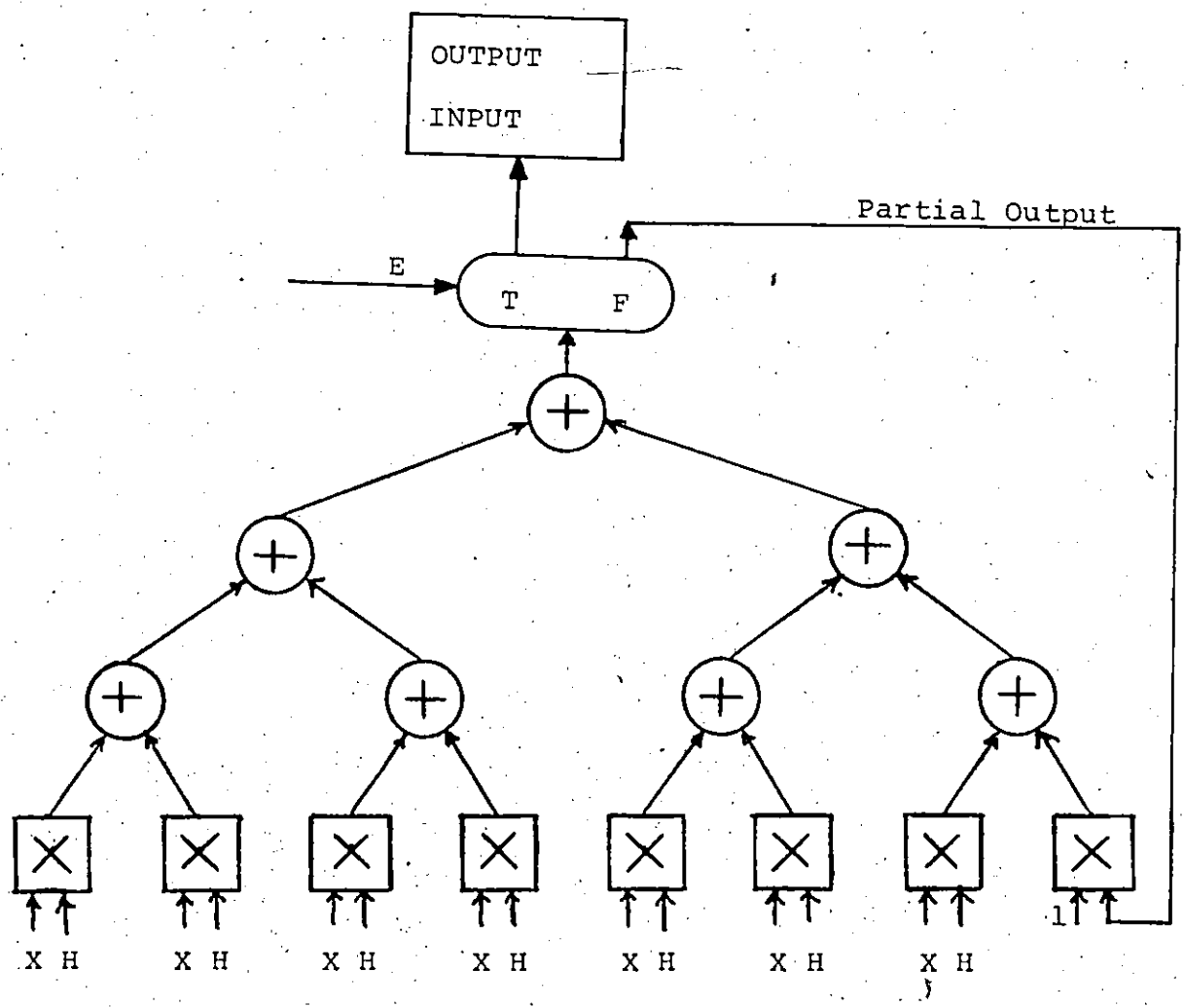


Figure 6.1 Data Flow Graph

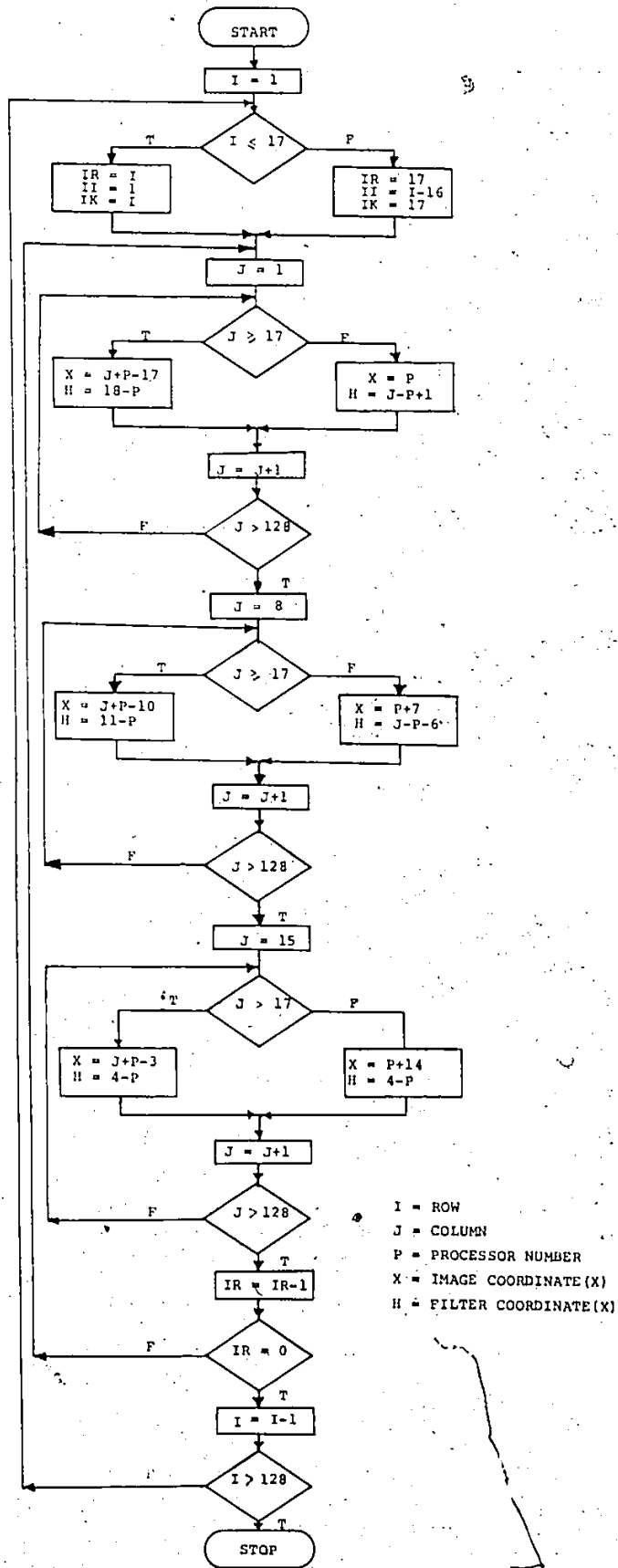


Figure 6.2 Data Flow Graph (Command Program)



## 6.5 SUMMARY

Various alternatives for programming the data flow signal processor have been discussed. It was found that data flow graphs can be used to program the DFSP and can then be incorporated in an interactive environment. It was shown that DFG shows the processor interconnection topologies, and describes node functions and I/O queues to maintain proper execution of the algorithms. The information provided by the DFG is very vital to write an interactive program. The example of two dimension direct convolution was used to illustrate steps which will be involved in programming of the DFSP. Two DFG were shown, one of them describes the processor interconnections and its different functions performed by all the cells of the tree structure. The second DFG shows the command program to be executed in the base cells for feeding proper image data and filter coefficients. The criterion for assigning functions for various cells was described. A manual scheme for generating I/O queues was shown for the two dimension convolution example. Efforts should be directed for automatic generation of I/O queues which can avoid communication bottlenecks. The two flow graphs can be used as an example to write an interactive program which will be responsible to receive the input information and will be capable to perform all the specified operations as discussed in section 6.4. The interactive program environment is transparent to the user and does not require any

knowledge of the internal architecture of the data flow signal processor.

## Chapter VII

### CONCLUSION

The main objective of this work was to present a novel structure for a general purpose programmable signal processor which is appropriate to a number of real time digital signal processing applications utilizing residue number system. In order to achieve this goal we explored the signal processing hardware structures described in the literature; various other architectures were also studied. Real time applications of a great many signal processing algorithms require a very high throughput rate which, although it may be available by using high speed, high performance general purpose computers, introduces a very high cost. Much of the hardware proposed in the literature is dedicated to simple applications, changing the algorithm may require changes in the hardware, and the cost of redesigning custom hardware is very high.

A fast signal processor can more appropriately be designed by exploiting parallelism in the algorithm and utilizing several processing elements to improve the throughput rate. Another approach to provide additional parallelism is to use appropriate arithmetic modules which inherently provide relative independent operation as only part of the re-

sult. The residue number system is ideally suited to this purpose, since operations of addition, subtraction and multiplication are completely independent from digit to digit. The use of RNS has received wide attention for its exploitation in high speed digital signal processing hardware.

Initial investigation were made into the feasibility of using RNS for high speed signal processing applications. This was achieved by constructing a set of band pass filters using EPROM's, FIFO's and shift registers. This hardware revealed that this idea can be extended to a more general purpose programmable signal processor. The objective of generalizing the constructed hardware was achieved by utilizing the binary tree shaped structure, replacing EPROM's, FIFO's, and shift registers with RAM's. This transformation gave birth to a more general purpose signal processor.

The processor is based on the data flow tree structure which has two types of cells, tree cells and the base cells. One of the efficiencies of this structure is to utilize one base cell for all the four computational elements. Hardware savings can be achieved by dividing base cells into a main cell and a modulus cell. Main cells are common to all the computational elements. The base cells provide an interface between the tree structure and the host computer. The host computer is responsible for receiving all the input information for the signal processing algorithm which is analyzed and is divided into smaller tasks. The host

computer distributes these tasks to the base cells and they execute them in parallel, the output data is pipelined. Thus the processor utilizes distributed parallel and pipeline processing techniques to achieve a high throughput rate.

The DFSP structure provides solution to many problems associated with the conventional data flow computers. The following points are pertinent.

1. The overhead bits are eliminated by incorporating them with the coefficient memory. This can be compared with the 200% overhead bits in other data flow machines (U-Interpreter [10]).
2. The operands are transmitted in parallel, avoiding all the overheads associated with the packet serial communication protocol.
3. The approach is deterministic and throughput rate can be calculated.
4. The computation time of any arithmetic operation is reduced to the access time of the RAM type used in the system.

The processor has been simulated for many applications including recursive, non-recursive and complex filtering operations. An example of image enhancement was used to demonstrate the operations of the DFSP. The two approaches of implementing the two dimensional convolution using NTT and direct convolution were investigated. An interesting conclusion is that the direct convolution approach provides a

lower computation time than the transform approach. This confirms that the proper manipulation of algorithms can provide their efficient implementation.

The theory of quadratic residue number system (QRNS) was also investigated for complex filtering applications. The DFSP can be utilized for complex filtering using QRNS and provide lower computation time as compared to the implementation using RNS.

Finally this work investigated the programming aspect of the DFSP using data flow graphs. Initially data flow graphs are drawn which can be easily converted into an interactive program for programming the data flow signal processor. An algorithm for an interactive program has been described which can be used to write a program on the host computer. The algorithm provides an opportunity to the user to write a real time signal processing program without knowing the architecture of the DFSP. The programming of the DFSP had been explained via an example and flow graphs have been shown. The assigning of the functions for the various cells and generation of I/O queues have been performed manually. An automatic generation of I/O queues will require further investigation in order to avoid any communication bottlenecks.

The proposed data flow signal processor is suitable for wide variety of real time signal processing applications. Its memory and arithmetic intensive architecture easily adopts to the requirements of the signal processing algo-

ithms. The processor is highly modular and utilizes identical components, this property can be exploited for its VLSI implementation. The processor is programmable and operate at a instruction rate of approximately 428 MOPS.

## REFERENCES

1. Jonathan Allen, " Computer Architecture for Signal Processing," Proceeding of IEEE, Vol. 63, No.4, April 1975.
2. John Hartung, " A Microprogrammable Digital Signal Processor: Concept, Design, Applications." Ph.d Thesis, State University of New Jersey, Jan. 1980.
3. A. Peled, B. Liu, 'Digital Signal Processing', Addison Wesley
4. R. R. Shively, " Architecture of a Programmable Digital Signal Processor." IEEE Trans. on Computers, Vol. C-31, No. 1, Jan. 1982, PP. 16-21.
5. J. S. Thompson, S. K. Tewksburg, " Lsi Signal Processor Architecture for Telecommunication Applications." IEEE Trans. ASSP Vol. ASSSP-30, No. 4, Aug. 1982, PP. 613-631.
6. Stanley L. Freeny, " Special purpose Hardware for Digital Filtering." Proceedings of the IEEE, Vol. 63, No. 4, April 1975, PP. 633-648.
7. John Backus, " Functional Level Computing." Spectrum Aug. 1982. PP.22-27.
8. John Backus, " Can Programming be Liberated from the Von-Neumann Style?", A Functional Style and Its Algebra of Programs." Communications of the ACM, Aug. 1978, PP. 613-641.
9. G. A. Mago, " A Network of Microprocessors to Execute Reduction Languages." Part I and II. International Journal of Computer and Information Science, Oct. & Dec. 1979. PP.349-385, and PP. 435-471.
10. Arvind, K. P. Gosetelow, " The U- Interpreter." IEEE Computer Feb. 1982. PP 51-57.
11. Ian Watson, J. Gurd, " A Practical Data Flow Computer", IEEE Computer, Feb. 1982, pp 51-57.
12. F. J. Burkowski, " A Multi-User Data Flow Architecture," 8th annual symposium on Computer Architecture, May 1981, pp 327- 334.



13. H. T. Kung, "Why Systolic Architectures." Computer, Vol. 15 No. 1, January 1982.
14. J. Vanaken and G. Zick, "The Expression Processor: A Pipelined Multiple- Processor Architecture." IEEE Trans. Computer, Vol c-30, No. 8, Aug. 1981, PP 525-536.
15. N. S. Szabo, R.I. Tanaka, "Residue Arithmetic and its Applications to Computer Technology." McGraw Hill 1967.
16. G. A. Jullien, "Residue Number Scaling and other Operations Using ROM Arrays" IEEE Trans. Computer, vol. c-7, NO. 4, PP.325 -337, April 1978.
17. G. A. Jullien, W. C. Miller, M. M. Jamali, "Implemetation of a Spectrum Analyzer Using a Memory Intensive Architecture." Canadian Communication & Energy Conference, Montreal, Oct. 1982.
18. M.M. Jamali, G.A. Jullien, W.C. Miller, S.I. Ahmad, "A real Time General Purpose Signal Processor." accepted for presentation at the ICASP-84 in San-Diego.
19. G.A. Jullien, "Implementation of Multiplication Modulo a Prime Number, with Applications to Number Theoretic Transform," IEEE Trans. Comput., Vol. C-29, pp. 899-905, Oct. 1980.
20. H.K. Nagpal, G.A. Jullien, W.C. Miller, "Processor Architectures for Two-Dimensional Convolvers Using a Single Multiplexed Computational Element with Finite Field Arithmetic," IEEE Trans. Comput. Vol. C-32, No.11, pp. 989-1001, Nov. 1983.
21. A. Z. Baraniecka, "Digital Filtering Using Number Theoretic Transform to Implement Fast Digital Convolution", Ph. D Dissertation, Electrical Engineering, University of Windsor, 1980.
22. H. K. Nagpal, "Processor Architectures for Fast Computation of Multip-dimensional Unitary Transforms", Ph. D. Dissertation Electrical Engineering, University of Windsor, 1981.
23. M. Akhtar, "A Read-Only-Memory Orientd Implementation of the Number Theoretic Transform Butterfly Unit", MASc Thesis, Electrical Engineering, University of Windsor, 1981.
24. P. B. Modak, "Implementation of an RNS Based Sequential NTT Convolver", MASc Thesis, Electrical Engineering, University of Windsor, 1982.

25. W. K. Jenkins, B. J. Leon, "The Use of Residue Number Systems in the Design of Finite Impulse Response Digital Filters", IEEE Trans. on Circuits and Systems, Vol. CAS-24, NO. 4, PP. 191-201, April 1977.
26. L. R. Rabiner, B. Gold, "Theory and Applications of Digital Signal Processing", Prentice Hall Inc., Englewood Cliffs, N.J.
27. F. Taylor, 'Digital Filter Design Handbook'
28. Y. C. Jenq, "Digital Convolution Algorithm for Pipelining Multiprocessor Systems", IEEE Trans. Computers, Vol. C-30, No. 12, PP. 966-972. Dec. 1981.
29. T. Mimaroglu, "A High Speed 2 D Hardware Convolver for Image Processing", Proceedings of Pattern Recognition and Image processing, PP. 386-389, 1982.
30. J. W. Modestino, R. W. Fries, 'Edge Detection in Noisy Images Using Recursive Filtering', Computer Graphics and Image Processing, 1977, PP. 409-433.
31. S. H. Leung, "Application of Residue Number Systems to Complex Digital Filters", Proc. of the Fifteenth Asilomar Conference on Circuit ans Systems, and Computers, Pacific Grove, PP. 70-74, Nov. 1981.
32. M. A. Sonderstrand, G. D. Poe, "Applications of Quadratic-Like Complex Residue Number System Arithmetic to Ultrasonics", IEEE Int. Conf. on ASSP, Vol. 2, PP. 28A.5.1-28A.5.4, Mar. 1984.
33. R. Krishnan, G. A. Jullien, W. C. Miller, "Complex Digital Signal Processing Using Quadratic Residue Number Systems", submitted for publication.
34. W. K. Jenkins, "Quadratic Modular Number Codes for Complex Digital Signal Processing", IEEE Int. Symposium on Circuit & Sytems, May 1984.
35. S. Jajodia, J. Liu, P. A. Ng, "A Scheme of Parallel Processing for MIMD Systems", IEEE Trans. on Software Engineering, Vol. SE-9, No.4, PP. 436-445, July 1983.
36. J. R. McGraw, "Data Flow Computing- Software Developments", IEEE Trans. on Computers, Vol. C-29, NO. 12, PP. 1095-1103, Dec. 1980.
37. B. A. Bowen, W. R. Brown, "VLSI System Seign for Digital Signal Processing", Vol. 1., Prentice Hall, 1982.

38. Huang et al, 'Implementation of a Fast Digital Signal Processor Using RNS', IEEE Trans. on Circuit & Systems, Vol- CAS-28, No.1 Jan. 1981, PP. 32-37.
39. J. H. Hesson et al, 'A 32 Bit Programmable Signal Processor for a Multiprocessor System Environment', IEEE Trans. on ASSP, Vol. ASSP-31, No. 4, Aug. 1983, PP.912-921.
40. F. M. Mintzer et al, 'The Real Time Signal Processor', IEEE Trans. on ASSP, Vol. ASSP-31, No. 1, Feb 1983, PP. 83-95.

## Appendix A

### STRUCTURAL DESIGN OF A FILTER

A structural design of the filter is presented in this section. A set of bandpass filter can be implemented by cascading a comb filter and a second order section. The result is a filter with the maximum gain at the cancelling frequency and reducing side lobes. The block diagram of the filter is shown in Figure A.1 which shows the positions of the scaling arrays. There is only one scaling array for six multiplications and six additions and is situated at the output of the filter. A hardware saving can be made in half of the multipliers because the non-trigonometric coefficients are fixed. We can, in fact, combine the fixed coefficient multipliers with part of the six input summers. The other multipliers are trigonometric coefficients which, though fixed for a given filter, vary if we are to use the same filter section for all the filters. The filter coefficients are converted to integers by multiplying by appropriate weighting factors. If we let the weighting factor for the coefficients of the numerator be  $W_z$  and for the coefficients of denominator as  $W_p$ , then the ratio is chosen so as to reduce the effective maximum gain of the filter to less than unity. This helps to prevent the overflow of the dynamic range.

The computational element is shown in Figure 3.3 and four identical elements are used one for each element.

We chose four five bit moduli 27,29,31,32 and a scaling factor of 783. We also let this equal the weighting factor  $W_p$  and found the best value for  $W_z$  as 50.

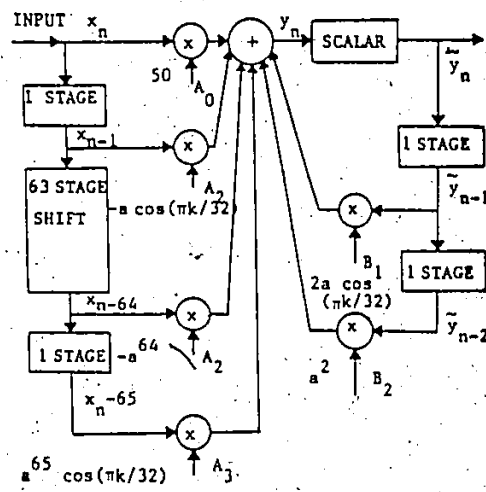


Figure A.1: Block Diagram of the Filter Structure

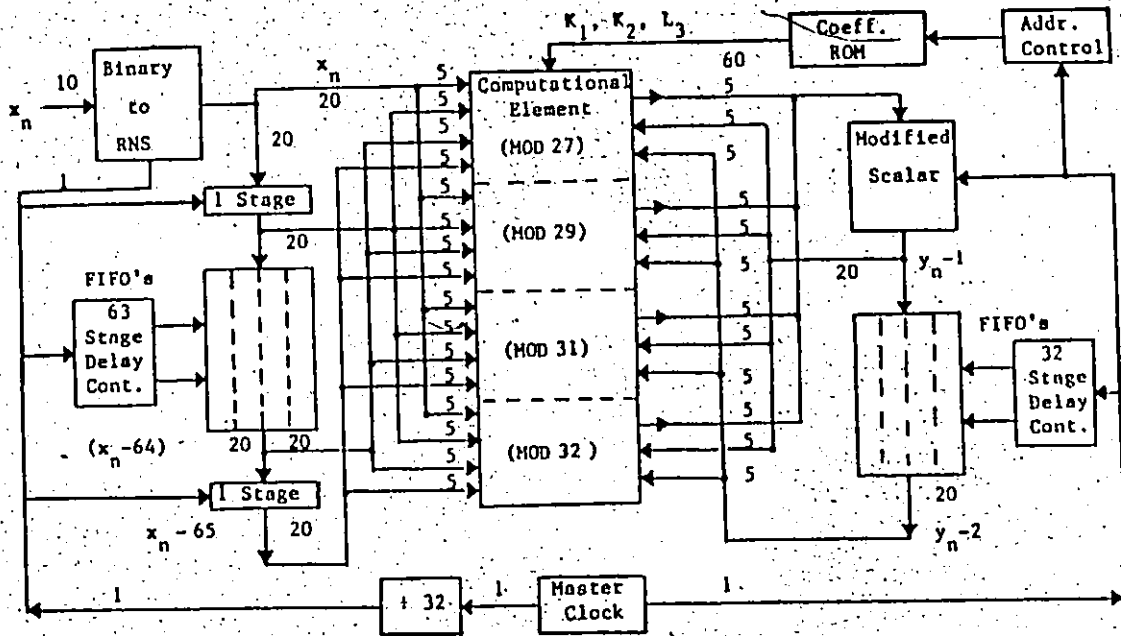


Figure A.2. Block Diagram of the Complete System

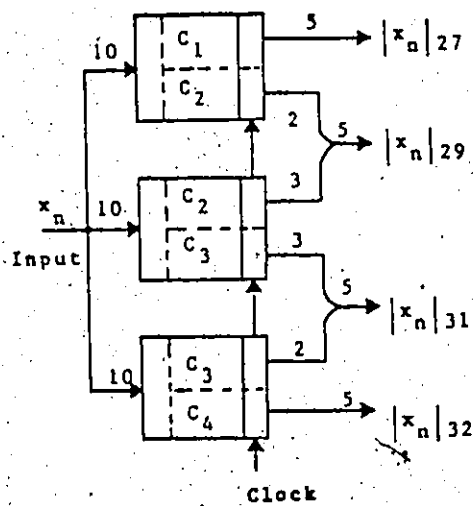


Figure A.3: Binary to Residue Conversion Unit



## A.1 HARDWARE IMPLEMENTATION

The filter is implemented as shown in Figure A.2. The input analog signal is digitized utilizing a sample/hold circuit and an A/D converter. The binary input is converted to four residues using the look-up tables shown in Figure A.3. The sample data can be delayed using a simple shift register to obtain a sample for  $x(n-1)$ . A programmable delay controller is implemented and a FIFO is used to provide an arbitrary delay. Thus, when initializing the 63 stage delay FIFO, we start the shift in pulse train, wait for 63 pulses to occur and then start the shift out pulse train. Using this scheme, there is one pitfall to watch out for. We have to make sure that the data has sufficient time to 'fall through' from the input to the output. Fall through time has to be less than or equal to the delay of the FIFO. A further delay is provided for  $x(n-65)$ . There are four computational elements representing each moduli and data is processed in four redundant paths. The computational element produces four output residues and they are scaled down using an efficient scaler as shown in Figure A.4.

A saving can be made in the number of FIFO packages by moving the FIFO from the output of the scaler to the middle of the scaler. If the FIFO is connected between the scaling and residue recovery parts of the scaler, then only 2 residues have to be delayed rather than 4 at the output. This yields a saving of 10 bits or 2 FIFO packages. We can elimi-

nate the registers (or latches) required at the output of S4 and S5 of the scaling array by changing the FIFO delays from 25 to 26 stages. The modified scaler is shown in Figure A.4. The binary output will experience a further delay over the original scaler; this is not important since the throughput rate is not affected. The binary output can be used for demultiplexing and digital to analog conversion for individual filter output. The four output residues are further delayed in output FIFOs to obtain  $y(n-2)$ . The input computational element and scaler are implemented in a pipeline. The clock pulses for the pipeline are derived from a decoder using the master clock. The whole hardware is controlled by the master clock. The input stages run 32 times slower than the computational element.

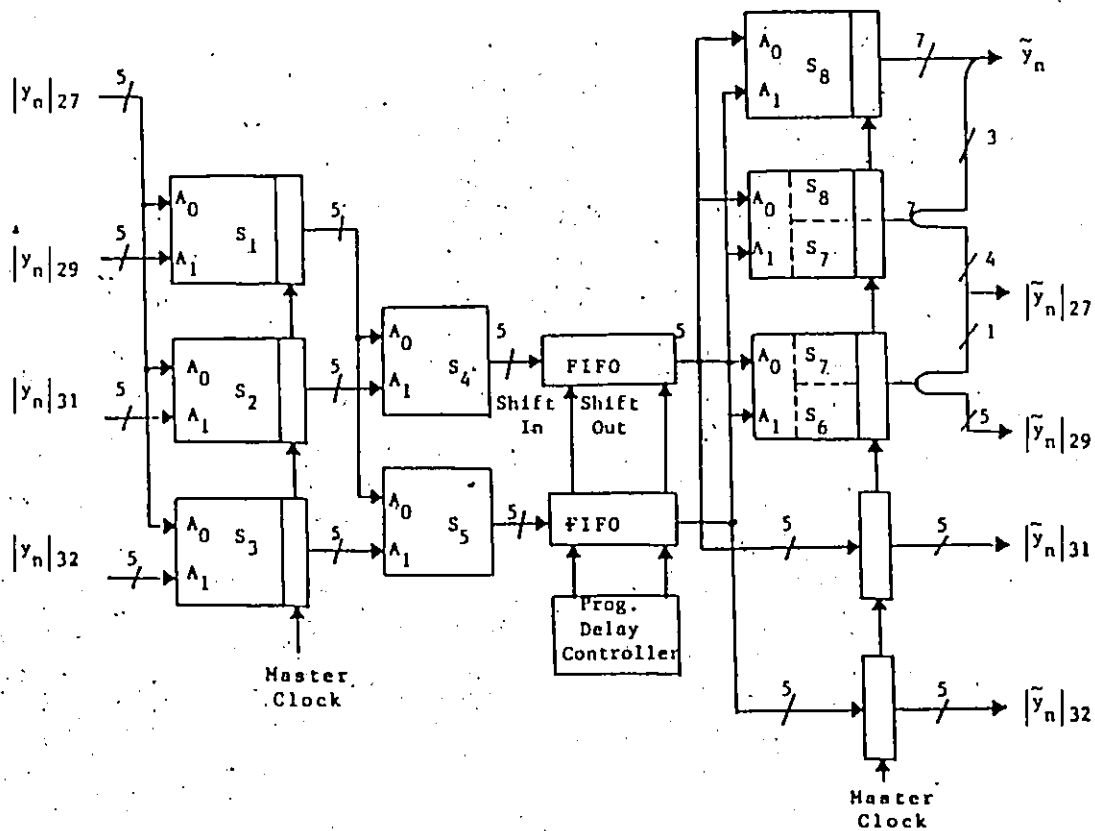


Figure A.4: Modified Scaler Unit

5

## Appendix B ..

### COMPLEX DIGITAL SIGNAL PROCESSING

Many of the signal processing applications have representation of data as complex numbers. The processing of the complex data requires complex multiplication which are normally performed by four integer multiplications and two additions. Recently Quadratic Residue Number System (QRNS) was proposed by Leung [31] to reduce the number of integer multiplications to two utilizing  $4n+1$  type of primes. More recently Sondersand and Poe [32] extended QRNS approach to accomodate  $4n+3$  type of primes, they call it Quadratic Like Residue Number System (QLRNS). This approach of QLRNS accomodates  $4n+3$  type of prime but the dynamic range is reduced. Krishnan and Jullien [33] modified QRNS to include  $4n+3$  type of prime and they named the new system as MQRNS by allowing an extra multiplication.

The purpose of this research is to show that complex signal processing algorithms can be executed on DFSP. We restrict the study to QRNS and assume that four  $4n+1$  type of prime would be sufficient for a large number of real time complex signal processing application.

Let us consider the set of  $4n+1$  type of moduli which contains  $-1$  as their quadratic residue [31-34]

$$x_i^2 = |-1|_{m_i} \quad (B.1)$$

has a solution for all  $i$ .

If  $J$  denotes the solution to (B.1) we have

$$J_i^2 = |-1|_{m_i} \quad (\text{B.2})$$

There exists a multiplicative inverse of  $j$  modulo  $M$  and is equal to  $-j$ .

Example:  $M=29$

$$j=12 \quad j^{-1} = -12=17$$

multiplicative inverse of 12 modulo 29 is 17.

Let us consider two complex numbers  $X$  and  $Y$

Conventional complex multiplications of  $X$  and  $Y$  as described in the literature [31-34] can be defined as

$$Z=X.Y \quad (\text{B.3})$$

$$\text{where } X=X_R + jX_I, \quad Y=Y_R + jY_I$$

$$\text{real part } Z = X_R.Y_R - X_I.Y_I \quad (\text{B.4})$$

$$\text{imaginary part } Z = X_R.Y_I + X_I.Y_R \quad (\text{B.5})$$

Now consider complex multiplications in QRNS with modulus  $M$

$$X=X_R + jX_I, \quad X^* = X_R - jX_I \quad (\text{B.6})$$

$$Y=Y_R + jY_I, \quad Y^* = Y_R - jY_I \quad (\text{B.7})$$

$$\text{Let } Q=X.Y \quad \text{and } Q^*=X^*.Y^* \quad (\text{B.8})$$

then the real and imaginary parts of the products can be performed as

$$\begin{aligned} Q_R &= \left| 2^{-1} \cdot (Q+Q^*) \right|_M \\ Q_I &= \left| 2^{-1} \cdot j^{-1} \cdot (Q-Q^*) \right|_M \end{aligned} \quad (\text{B.9})$$

It shows that computing a complex multiplication requires two integer multiplications.

#### B.0.1 Hardware Implementation of QRNS

The execution of any complex signal processing algorithm can be performed on the DFSP. The arithmetic operations w.r.t  $Q$  and  $Q^*$  are multiplexed on the tree structure. The coding of residue into QRNS is performed in the beginning of the execution of the algorithm and all arithmetic operations are executed using  $Q$  and  $Q^*$ . The QRNS are converted into real and imaginary part of the RNS and use mixed radix conversion to map into real and imaginary part. The computational efficiencies of the complex multiplications are more than the overhead associated with the QRNS. We take an example of FIR filter implementation using QRNS.

$$\begin{aligned} \text{Let } x(n) &= 1+j0 && \text{for } 0 \leq n \leq 6 \\ &= 0 && \text{for } n > 6 \\ h(n) &= 1+j0 && \text{for } 0 \leq n \leq 6 \\ &= 0 && \text{for } n > 6 \end{aligned}$$

The 1 D convolution of the two complex requirements can be defined as

$$y(n) = \sum_{l=0}^{N-1} x(l) \cdot h(n-l) \quad (\text{B.11})$$

This example is used to demonstrate complex filtering operations on the DFSP. The algorithm is allocated on the tree structure as shown in Figure B.1. The operations related to  $Q$  and  $Q^*$  are multiplexed and pipelined. The output of  $Q$  and  $Q^*$  can be demultiplexed from cell 15 of the tree structure. The convolution sum requires simple multiplication and additions and the look up tables are the same for both  $Q$  and  $Q^*$  operations. This example has been simulated on the SE1 27/32 and the results were obtained as expected and is shown in Figure B.2.

The conversion of the data into QRNS is performed by B/R unit. The filter coefficients can be stored in the QRNS form and the output is recovered using R/B conversion in the output unit. This example requires two cycles for each operations using QRNS which can be compared with that of RNS operation requiring 4 cycles. This gives a 50% increase in the throughput rate using QRNS. Similarly MQRNS [33] can be performed on DFSP which will require an extra cycle and lower the throughput rate. It can be seen that the DFSP can be used for complex digital signal processing applications.

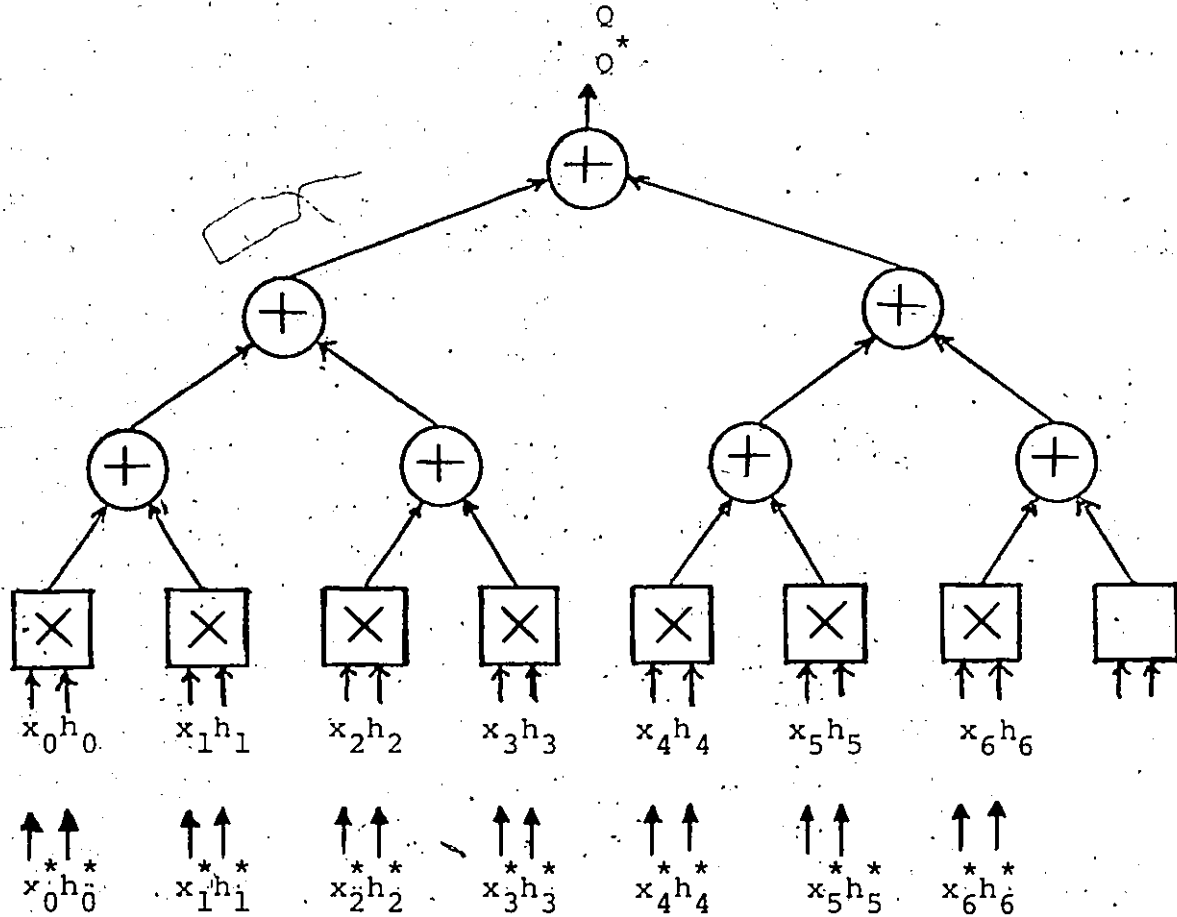
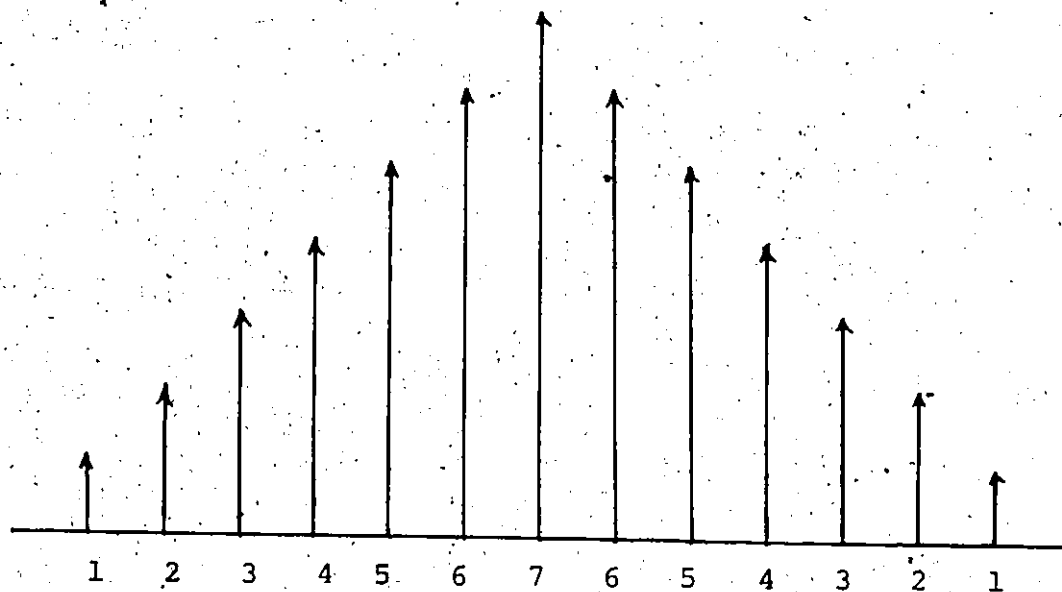
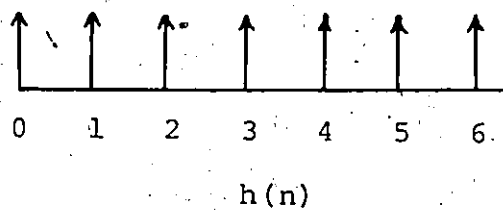
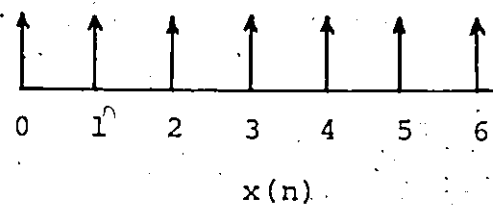


Figure B.1 QRNS Implementation of FIR Filters





$$y(n) = \sum_{k=0}^{N-1} x(k) \cdot h(n-k)$$

Figure B.2 1-D Convolution Using QRNS

Appendix C  
COMPUTER PROGRAMS

\*\*\*\*\* SIM \*\*\*\*\*  
Simulation program for Data Flow Signal Processor

This program simulates the example of the set of tables  
The program reads all the look up tables from the  
Program Table should be executed first before this

MODIFIED ON NOV.23 1983  
INTEGER S(3), DEL(4), IA(8), IX1(8), LIE(307,32)  
INTEGER I4(4), T3(4,2), T2(4,4), T1(4,8), DX(4,8)  
DIMENSION CND(100), ABS(100), Lr(100), LA1(4)  
INTEGER A0(+,8,0+), A(+), I(+), A0, A2, B2  
INITIALIZE ALL THE ARRAYS  
Z1=Z1AR(1.0)/3.0

```
DO 5 I=1,8  
5 S(I)=0  
DO 6 I=1,100  
CND(I)=0  
ABS(I)=0  
T1(I)=0  
CONTINUE  
DO 10 M=1,4  
I4(M)=0  
X(M)=0  
Y(M)=0  
LA1(M)=0  
DO 20 I=1,2  
T3(M,I)=0  
CONTINUE  
DO 30 I=1,4  
T2(M,I)=0  
CONTINUE  
DO 40 I=1,8  
I1(M,I)=0  
IX(M,I)=0  
DO 50 J=1,64  
C0(M,I,J)=0  
CONTINUE  
CONTINUE  
DEL(1)=27  
DEL(2)=29  
DEL(3)=31  
DEL(4)=32  
  
READ(11) LPS  
  
IF(LPS=0) IF(ND) LOGTABLE LIST IS REQUIRED  
  
LPS=0  
IF (LPS=0) DO 100 300  
IA=1  
IB=32  
DO 101 I=1,3  
DO 111 J=1,4  
A: IIA(0,121) DL(0)  
121 FORMAT ('0',10X,'TABLE FOR LOGCLO=',I3, '//')  
DO 131 K=1A, 13
```

POOR PRINT  
Epreuve illisible

```
131 CONTINUE
141 FORMAT (1X, I3, 4X, 4(4X, 3(1X, 12)))
IA=IA+32
IB=IB+32
111 CONTINUE
101 CONTINUE

WRITE(6, 131)
151 FORMAT ('0', 10X, 'COEFFICIENT TABLES'///)
DO 101 I=385, 396
WRITE(6, 141) I, (LIE(I, L), L=1, 32)
1101 CONTINUE

C
C INPUT DATA HAS BEEN INSERTED IN LISTING
C
400 A0=50
A2=7

B2=7.55
NPI=400
NPL=100
IFIL=28
JPRN=0

J=1
K=3
L=9
IIC=387
IIS=385
DO 310 I=1, 4
LIE(IBC, J)=A0B(A0, IDL(I))
LIE(IBC, K)=A2B(A2, IDL(I))
LIE(IBC, L)=B2B(B2, IDL(I))
J=J+1
K=K+1
L=L+1
310 CONTINUE
C PRINT WHATEVER WAS SENT
WRITE(6, 341) NPI, NPL, IFIL
341 FORMAT (' ', 10X, 'NO. OF ITERATIONS=', I5, ///,
1 10X, 'PLOTTED POINTS=', I5, ///, 10X,
2 'ITERP=', I5)
WRITE(6, 350) (IDL(I), I=1, 4)
350 FORMAT (' ', 10X, 'HOSTILE ARG', 4(3X, 12))
WRITE(6, 360) A0, A2, B2, (LIE(IBC, I), I=1, 12)
360 FORMAT (' ', 9X, 'CONST COEFF', 1X, 316, 12(2X, 12))
C INPUT DATA CASE HERE
C
I=0
DO 210 I=1, 1000
A1=111-1
SINPR2=3.0

C
C FIELD ERR. CHANGED TO CHECK ATTENUATION
C
FCU1=7.00
FCU2 IS FIELD COEFF
IIP=512.0*SIN((21*(111)*PI)/7.00)
```

159

✓

```

      EG 380  IR=1,4
      K(IR) = MOD(INP, MDL(IR))
      IF (K(IR) .LT. 0) K(IR) = K(IR) + MDL(IR)
380  CONTINUE
420  DO 340  M=1,4
      DA(M,1) = K(M)
      DX(M,2) = X0(M,2,04)
      DA(M,3) = X0(M,0,1)
      LA(M,3) = X0(M,3,1)
      X0(M,3,1) = X0(M,2,04)
      X0(M,0,1) = X0(M,1)
      NN=04
      N1=NN-1
      IR=NN
      DO 300  I=1,N1
      X0(M,2,IR) = X0(M,2,IR-1)
      IR=IR-1
300  CONTINUE
      X0(M,2,1) = X0(M,1)
340  CONTINUE

      IF (IPRGT.EQ.0) GO TO 430

      WRITE(6,370) III, (DA(M,1), M=1,4), (DX(M,0), M=1,4),
1  (DA(M,2), M=1,4), (LA(M,3), M=1,4), INP
370  FORMAT(14,2X,4(3A,4(1X,14)),3A,10)
      SCALING ALGORITHM
      FIND STAGE BASE EXTENSION

430  DO 290  IFL=1,32

      S(3) = 31*(S(5) - S(4))
      S(5) = MOD(S(3), 32)
      IF (S(3) .LT. 0) S(3) = S(3) + 32
      IF (S(3) .GE. 16) S(3) = S(3) - 32
      S(6) = S(4) + 31*S(3)
      DO 100  I=1,2
      S(I+5) = MOD(S(3), MDL(I))
      IF (S(I+5) .LT. 0) S(I+5) = S(I+5) + MDL(I)
      IS=I+2
      Y(I) = S(I+5)
      Z(I) = LA1(I)
100  CONTINUE
      IF (IPRGT .GT. 0) GO TO 601
      WRITE(6,450) (S(I), I=4,7), (Y(I), I=1,4), S(6)
450  FORMAT(2(2X,4(1X,12)),110)

      SECOND STAGE
      S(4) = 15*(S(2) - S(1))
      S(5) = 21*(S(3) - S(1))
      DO 110  I=3,4
      S(I+1) = MOD(S(I+1), MDL(I))
      IF (S(I+1) .LT. 0) S(I+1) = S(I+1) + MDL(I)
      LA1(I) = S(I+1)
110  CONTINUE
      IF (IPRGT .GT. 0) GO TO 602
      WRITE(6,451) (S(I), I=1,5)
451  FORMAT(2X,5(1X,12))

```

```

FIRST STAGE
102 S(1)=14*(T4(2)-T4(1))+14
S(2)=23*(T4(3)-T4(1))+14
S(3)=19*(T4(4)-T4(1))+14
DO 120 I=2,4
S(I-1)=MOD(S(I-1),MDL(I))
IF(S(I-1).EQ.0) S(I-1)=S(I-1)+MDL(I)
120 CONTINUE

BINARY TREE NETWORK
4TH STAGE
T=0
DO 130 I=1,4
IP1=13(A,1)+32*(N-1)+1
IP2=13(A,2)+1
T4(I)=LFB(IP1,IP2)
130 CONTINUE

3RD STAGE
DO 150 M=1,4
IP1=12(S,1)+129+32*(M-1)
IP2=12(S,2)+1
I3(M,1)=LFB(IP1,IP2)
IP1=12(S,3)+52*(M-1)+1
IP2=12(S,4)+1
I3(M,2)=LFB(IP1,IP2)
150 CONTINUE

5TH STAGE
J=1
T=1
DO 160 I=1,6,2
DO 170 M=1,4
II1=11(A,I)+129+J*(M-1)
IP2=11(A,I+1)+1
CALL ACCESS
T2(M,I)=LFB(IP1,IP2)
170 CONTINUE

J=J+1
180 CONTINUE
IF(II1.EQ.0) GO TO 603
WRITE(6,510) ((T2(M,I),M=1,4),I=1,6),((I3(M,I),M=1
,I=1,2),I=1,4),IFL,II1
510 FORMAT(+ (2X,1),4(2X,4(1X,12)),5X,4(1X,12)
FIRST STAGE
FIRST CELL
T=1

```

```
005 DO 250 N=1,4  
C=0  
IL=257+32*(N-1)  
IP1=IL+DX(N,1)  
A=4*C+A  
IP2=LIS(IEC,A)+1  
T1(N,1)=LTB(IP1,IP2)
```

C  
C  
C  
C

SECOND CELL

```
C=1  
IP1=IL+DX(N,2)  
IAD=4*C+A  
IP2=LIS(IAC,IAD)+1  
T1(N,2)=LTB(IP1,IP2)  
THIRD CELL
```

C  
C  
C

UPDATES

```
DX(N,3)=XO(N,3,32)  
M=32  
I1=IN-1  
IK=IN  
DO 270 IL=1,IN  
XO(N,3,IK)=X(N,3,IK-1)  
IK=IK-1
```

```
270 CONTINUE
```

```
M=26  
XO(N,3,1)=XO(N,7,M)  
M=M-1  
DX(N,7)=XO(N,7,M)  
IK=M  
DO 280 IL=1,IN  
XO(N,7,IK)=X(N,7,IK-1)  
IK=IK-1
```

```
280 CONTINUE
```

```
XO(N,7,1)=Y(1)  
C=2  
IAD=4*C+A  
IP1=IL+DX(N,3)  
IP2=LIS(IAC,IAD)+1  
T1(N,3)=LTB(IP1,IP2)  
FOURTH CELL  
T1(N,4)=0  
FIFTH CELL
```

C  
C  
C  
C

MULTIPLY X2 AND X3-13

```
C=1  
IP1=IL+DX(1,3)  
IAD=IIS+4*C+I-1  
IP2=LIS(IAD,IP1)+1  
T1(N,3)=LTB(IP1,IP2)  
SIXTH CELL
```

C  
C

MULTIPLY A1 AND A3-1

T=2  
C=0  
IP1=ID+IX(M,5)  
IAD=IES+4\*C+1-1  
IP2=LTB(IAD,IFL)+1  
T1(N,6)=LTB(IP1,IP2)

SEVENTH CELL  
C=2  
IP1=ID+IX(M,7)  
IAD=IES+4\*C+1-1  
IP2=LTB(IAD,IFL)+1  
T1(N,7)=LTB(IP1,IP2)

EIGHTH CELL  
T1(N,8)=C

250 CONTINUE

IF (IT1.GE.318) GO TO 305  
WRITE(6,453) (DX(I,5),I=1,4), (DX(N,7),I=1,4)  
493 FORMAT(2(2X,4(1X,12)))

ICM=NPI-IT1  
IF (ICM.GE.10) GO TO 290  
IF (IFL.LE.IF12) GO TO 290  
IM=IM+1

ORD(1,1)=3(2)  
ABCS(1,1)=IT1

IF (IC)=IM?

290 CONTINUE

210 CONTINUE

DO 330 I=1,50  
J=50+I

WRITE(6,320) I,ORD(1),ABCS(1),IP(I),J,ORD(J),ABCS(J)

320 FORMAT(2(5X,13,3(2X,F8.2)))

330 CONTINUE

410 CONTINUE

WRITE(6,362) POST,SAMPLE

362 FORMAT(DX,'FILTER OUTPUT=',F8.1,9X,'FAC.=',F8.1)

CALL PLOT3(ABCS,ORD,IM)

CALL PLOT3(ABCS,IP,IC)

STOP

END



PROGRAM TO GENERATE LOOK TABLE  
AND STORE ON THE DISK  
INTEGER ATAB(1024), STAB(1024), MTAB(1024), AD(397, 32)  
DIMENSION M(4)

158

DOUBLE PRECISION CMULT, SCALE, PI, RAD, A1, A3, E1  
SELECTED MODALI ARE M(I)

M(1)=27

M(2)=29

M(3)=31

M(4)=32

KD=1

KS=129

KM=257

SCALE FACTORS

CMULT=50.0

SCALE=783.0

DO 10 I=1, 397

DO 10 J=1, 32

AD(I, J)=-1

LOOK UP TABLES FOR ADD, SUB & MULT

DO 30 IK=1, 4

DO 20 I=1, 1024

ATAB(I)=-1

STAB(I)=-1

MTAB(I)=-1

CONTINUE

MD=M(IK)

DO 35 I=1, MD

I1=I-1

I2=I1\*32

DO 40 J=1, MD

J1=J-1

J2=I2+J

ATAB(J2)=I1+J1

ATAB(J2)=MOD(ATAB(J2), M(IK))

STAB(J2)=I1-J1

STAB(J2)=MOD(STAB(J2), M(IK))

IF(STAB(J2).LT.0) STAB(J2)=STAB(J2)+M(IK)

MTAB(J2)=I1\*J1

MTAB(J2)=MOD(MTAB(J2), M(IK))

CONTINUE

CONTINUE

STORE THESE TABLES IN MATRIX AD

IC=1

DO 60 JJ=1, 1024

AD(KD, IC)=ATAB(JJ)

AD(KS, IC)=STAB(JJ)

AD(KM, IC)=MTAB(JJ)

IC=IC+1

IF(IC.LE.32) GO TO 60

```

        IC=1
        KD=KD+1
        KS=KS+1
        KM=KM+1
60      CONTINUE
30      CONTINUE
C
C      CALCULATE COEFFICIENTS
C
        KBIT=5
        PI=DATAN(1.00)/8.00
        LBIT=-1*KBIT
        RAD=1-2.**LBIT
        DO 70 JJ=1,32
            A1=CMULT*RAD*DCOS(PI*JJ)
            A3=CMULT*DCOS(PI*JJ)*RAD**65
            B1=2.0*SCALE/CMULT*A1
            MTAB(1)=A1
            MTAB(2)=A3
            MTAB(3)=B1
            DO 80 I=1,3
                DO 90 J=1,4
                    K=385+4*(I-1)+J-1
                    AD(K, JJ)=MOD(MTAB(I),M(J))
                    IF(AD(K, JJ).LT.0) AD(K, JJ)=AD(K, JJ)+M(J)
90          CONTINUE
80          CONTINUE
            WRITE(6,170) A1,A3,B1
170         FORMAT(3(2X,F16.8))
70         CONTINUE
        IA=1
        IB=32
        DO 100 I=1,3
            DO 110 J=1,4
                WRITE(6,120) M(J)
120         FORMAT('0',10X,'TABLE FOR MODULO=',I3, '//')
                DO 130 K=IA,IB
                    WRITE(6,140) K, (AD(K,L),L=1,32)
130         CONTINUE
140         FORMAT(1X,I3,4X,4(4X,8(1X,I2)))
                IA=IA+32
                IB=IB+32
110        CONTINUE
100       CONTINUE
C
        WRITE(6,150)
150       FORMAT('0',10X,'COEFFICIENT TABLES'//')
        DO 160 I=385,396
            WRITE(6,140) I, (AD(I,L),L=1,32)
160       CONTINUE
C        WRITE(11) AD
        STOP
        END

```



```

NK=JK+1
CALL COMPINF2CNK(SINF)
DO 15 J=1,4
  STOR1(J)=KIN(J)
  STOR2(J)=REG1(J)
  NN=63
  IK=NN
  NN=NN-1
  STOR3(J)=TENREG(J,NN)
  DO 430 I=1,NN
    TENREG(J,IK)=TENREG(J,IK-1)
    IK=IK-1
100  CONTINUE
    TENREG(J,1)=STOR2(J)
    STOR4(J)=REG2(J)
15  CONTINUE
210  NI=JF11
    NN=31
    DO 30 MODE=1,4
      NUM=4
      NN=NN-1
      STOR5(MODE)=OFIF(MODE,31)
      IK=NN
      DO 35 J=1,NN
        OFIF(MODE,IK)=OFIF(MODE,IK-1)
        IK=IK-1
35  CONTINUE
        OFIF(MODE,1)=SCALEOUT(MODE)
30  CONTINUE
    WRITE(12,300) STOR5
    C300  FORMAT(100) (OUTPUT OF FIFOS(32)= (%.10%)(5),15)
    YA=31+(SCALEIN(4)-SCALEIN(3))
    YA=MOD(YA,32)
    IF(YA.LT.0) YA=YA+32
    IF(YA.GE.16) YA=YA-32
    YA=SCALEIN(3)+31+YA
    SCALEIN(4)=MOD(YA,37)
    IF(SCALEIN(1).LT.0) SCALEIN(1)=SCALEIN(1)+27
    SCALEIN(2)=MOD(YA,29)
    IF(SCALEIN(2).LT.0) SCALEIN(2)=SCALEIN(2)+29
    INTOUT=YA
    DO 32 MODE=1,4
      SCALEOUT(MODE)=SCALEIN(MODE)
32  DO 40 MODE=3,4
      NN=27
      NN=NN-1
      IK=NN
      NUM=MODE-2
      SCALEIN(MODE)=SFIF(NUM,27)
      DO 45 J=1,NN
        SFIF(NUM,IK)=SFIF(NUM,IK-1)
        IK=IK-1
45  CONTINUE
40  CONTINUE
    YOUT(3)=15+(TEMP(2)-TEMP(1))
    YOUT(3)=MOD(YOUT(3),31)
    IF(YOUT(3).LT.0) YOUT(3)=YOUT(3)+31
    YOUT(4)=21+(TEMP(3)-TEMP(1))
    YOUT(4)=MOD(YOUT(4),33)
    IF(YOUT(4).LT.0) YOUT(4)=YOUT(4)+32
    SFIF(1,1)=YOUT(3)
    SFIF(2,1)=YOUT(4)

```

```

548 PI=DATA(1,1,0,0,0,0)
ACCEPT"INPUT FREQ = ",FREQ
MULT=32
DO 1 M=1,4
R1(M)=0
R2(M)=0
R3(M)=0
R4(M)=0
R5(M)=0
R6(M)=0
R8(M)=0
R9(M)=0
STOR1(M)=0
STOR2(M)=0
STOR3(M)=0
STOR4(M)=0
STOR5(M)=0
STOR6(M)=0
SCALEIN(M)=0
TIP(M)=0
TOP(M)=0
REG1(M)=0
REG2(M)=0
YOUT(M)=0
TEMP(M)=0
SCALEOUT(M)=0
DO 2 J=1,64
TEMPREG(M,J)=0
2 CONTINUE
POINT(M)=0
DO 4 I=1,32
Y1(M,I)=0
Y3(M,I)=0
4 CONTINUE
1 CONTINUE
DO 6 I=1,65
XCOM(I)=0
6 CONTINUE
DO 7 I=1,400
ABS(I)=0
7 ORD(I)=0
ACCEPT"# OF FILTER=",IFIL
ACCEPT"# OF POINTS (NOT MORE THAN 999)=",NPOINT
ACCEPT"# OF POINTS FOR PLOTTING PPO=",PPO
ACCEPT"# OF INPUT FREQ POINTS,ONG=",ONG
ACCEPT" WISH TO PRINT PLOTTING DATA, 0=NO=1, 1=PRINT
NGP=NPOINT-PPO
JEIL=0
DO 470 I=1,4
DO 470 J=1,32
470 OFIF(I,J)=0
DO 475 I=1,3
DO 475 J=1,27
475 SFIF(I,J)=0
IN=0
0201 FORMAT('0',30,' COMPUTATIONAL POINT NO. =',2X,I3)
DO 600 IPQ=ONG,FREQ,10
IN=IN+1
DO 200 IX=1,NPOINT

```

```
SCALING FROMS FOR S1, S2 AND S3
TEMP(1)=14+(R9(2)-R9(1))+14
TEMP(1)=MOD(TEMP(1), 29)
IF(TEMP(1).LT.0) TEMP(1)=TEMP(1)+29
TEMP(2)=23+(R9(3)-R9(1))+14
TEMP(2)=MOD(TEMP(2), 31)
IF(TEMP(2).LT.0) TEMP(2)=TEMP(2)+31
TEMP(3)=19+(R9(4)-R9(1))+14
TEMP(3)=MOD(TEMP(3), 32)
IF(TEMP(3).LT.0) TEMP(3)=TEMP(3)+32
WRITE(12, 326) TEMP

C
C
C
COMPUTATION ELEMENT 4TH STAGE
DO 238 J=1, 4
CALL ACCESS1(ADDIND(J), R8(J), TOP(J), MULT, R9(J))
CONTINUE
238
C
C
3RD STAGE ADDITION FROM T7
DO 235 J=1, 4
CALL ACCESS1(ADDIND(J), R6(J), R4(J), MULT, R8(J))
TOP(J)=R5(J)
235
CONTINUE
WRITE(12, 375) R8
C
3375
3340
FORMAT(10X, 'ADDITION FROM T7=', 4(5X, I5))
FORMAT(10X, 'COMPUTATION OUTPUT R9=', 5(2X, I2), 2X, I2)
C
C
2ND STAGE COMPUTATION
DO 240 J=1, 4
X=SCALEOUT(J)
NMI=NI-1
IF(NI.EQ.0) NMI=31
Y=ROCEXN(COFIND(J, 30)+NMI)
TOP(J)=Y
CALL ACCESS1(MULIND(J), X, Y, MULT, R4(J))
CALL ACCESS1(SUBIND(J), R3(J), R2(J), MULT, R6(J))
CALL ACCESS1(COIND(J, 2), R1(J), STOR6(J), MULT, R5(J))
CONTINUE
240
```

```
C      1ST STAGE COMPUTATION
C
DO 250 J=1,4
CALL ACCESS1(COIND(J,1),STOR1(J),STOR3(J),MULT,R1(J))
X=STOR2(J)
Y=ACCESS1(COIND(J,1)+NI)
BY=Y
CALL ACCESS1(MULIND(J),X,Y,MULT,R2(J))
X=STOR4(J)
Y=ACCESS1(COIND(J,2)+NI)
BY=Y
CALL ACCESS1(MULIND(J),X,Y,MULT,R3(J))
250 CONTINUE
IF(KK.LT.NGP) GO TO 490
IF(IFIL.NE.JFIL) GO TO 490
IF(ORD(IND).LT.INTOUT) ORD(IND)=INTOUT
ABS(IND)=IFQ
CALL COMPUT1(JFIL,CUR)
IF(IPRINT.NE.0) GO TO 490
WRITE(12,480) IND,ORD(IND),CUR,SIN(ANK),ABS(IND)
480 FORMAT(10X,13,5X,4110)
490 JFIL=JFIL+1
IF(JFIL.LT.32) GO TO 210
JFIL=0
DO 80 J=1,4
REG1(J)=STOR1(J)
REG2(J)=STOR3(J)
80 CONTINUE
NN=65
DO 550 J=1,64
XCON(NN)=XCON(NN-1)
NN=NN-1
550 CONTINUE
JUE
XCON(1)=INF
600 CONTINUE
CONTINUE
E
ACCEPT "READY FOR PLOTTING TYPE #0", (TYPE
CALL PLIT(ABS,ORD,IND)
ACCEPT "WISH TO CONTINUE, 0=YES, OTHERWISE NO, INISH-
1000 IF(INISH.EQ.0) GO TO 510
CLOSE 0
STOP
END
```

```
SUBROUTINE GASNTAB(N, IPRT, TABIND)
C SUBROUTINE TO GENERATE ADDITION, SUBTRACTION & MULTIPLICATION TABLE
C FOR THE MODULUS 'N'.
  IMPLICIT INTEGER (A-Z)
  EXTERNAL POWER2
  INTEGER STAB(1024), ATAB(1024), MTAB(1024)
  COMMON STAB, ATAB, MTAB
  MULTP=POWER2(N)
  CALL INITM1(STAB, -1, 1024)
  CALL INITM1(ATAB, -1, 1024)
  CALL INITM1(MTAB, -1, 1024)
  NPTS=MULTP*MULTP
  DO 100 I=1, M
    I1=I-1
    I2=I1*MULTP
    DO 50 J=1, M
      J1=J-1
      J2=I2+J1
      ATAB(J2)=I1+J1
      ATAB(J2)=MOD(ATAB(J2), N)
      STAB(J2)=I1-J1
      IF(STAB(J2).GE.0) GO TO 20
      STAB(J2)=STAB(J2)+M
20    STAB(J2)=MOD(STAB(J2), N)
      NTAB(J2)=I1*J1
      MTAB(J2)=MOD(MTAB(J2), N)
50    CONTINUE
100  CONTINUE
  IF(IPRT.EQ.0) GO TO 110
  WRITE(12, 150)
150  FORMAT(///, 20X) " ADDITION, SUBTRACTION & MULTIPLICATION TABLES"
110  CALL OUTPUT(ATAB, NPTS, 0, M, IPRT, TABIND)
  CALL OUTPUT(STAB, NPTS, 0, M, IPRT, TABIND)
  CALL OUTPUT(MTAB, NPTS, 0, M, IPRT, TABIND)
  RETURN
  END
```



```
***** GCOMETAB*****  
PROGRAM TO GENERATE TABLES FOR THE COMB FILTER  
IMPLICIT INTEGER (A-Z)  
INTEGER SNAME(5)  
INTEGER STAB(1024), ATAB(1024), NTAB(1024), M(10), FNAME(5)  
COMMON STAB, ATAB, NTAB  
DOUBLE PRECISION CMULT, SCALE  
COMMON/BLK1/ CMULT, SCALE  
ACCEPT "TOTAL # OF MODULI IN THE SYSTEM. N= ", N  
ACCEPT "# OF SCALING MODULI = ", NS  
IF(N.GT.NS) GO TO 4  
TYPE "ERROR: N MUST BE GREATER THAN S ??? N=", N, " S=", NS, " ???"  
STOP  
NS MUST BE LESS THAN N. THE SCALING MODULI MUST BE SPECIFIED FROM  
M0, M1, M2, ---, MS-1 AND THE REST OF FROM NS, MS+1, ---, MN-1, WITH  
THE LAST MODULUS MN-1 AN EVEN MODULUS.  
ACCEPT "VALUES OF M0, M1, M2, ---, MN-1= ", (M(I), I=1, N)  
ACCEPT "# OF BITS IN RADIOUS = ", KBIT  
ACCEPT "COFF. MULT & SACLE FACT. = ", CMULT, SCALE  
ACCEPT "CONST. COFFS. - A0, A2, B2 = ", CON1, CON2, CON3  
WRITE(12, 7) N, NS, (M(I), I=1, N), (M(I), I=1, NS), KBIT, CMULT, SCALE,  
CON1, CON2, CON3  
1  
7  
1  
2  
3  
4  
5  
20  
50  
100  
FORMAT(" TOTAL # OF MODULI = ", I3, ", " # OF SCALING MODULI = ", I3,  
", " THE MODULI M0, M1---MN ARE = ", 4(I3, 2X), ", " THE SCALING MODULI  
ARE = ", 2(I3, 5X), ", " KBITS WHERE RADIOUS = (1-2**(-KBIT  
), " COFF. MULT. = ", F14, 7, ", " SCALE FACT. = ", F14, 7, ", " CONST.  
COFFS. A0, A2, B2 ARE = ", 3(I6, 5X))  
TYPE "FILENAME TO STORE TABLES"  
READ(11, 5) (FNAME(I), I=1, 5)  
FORMAT(5S2)  
OPEN 0, FNAME  
ACCEPT "WISH TO PRINT TABLES. 0-NO. 1-YES ", IWISH  
TABIND=0  
DO 50 I=1, N  
IF(I.EQ.N) GO TO 20  
CALL GSCTAB(M, N, NS, I, IWISH, TABIND)  
IF(NS+1.LT.N.AND.I.LE.NS+1) GO TO 50  
CALL GXTAB(M, N, NS, I, IWISH, TABIND)  
CONTINUE  
DO 100 I=1, N  
CALL GRSNTAB(M(I), IWISH, TABIND)  
CALL GCONTAB(M(I), CON1, CON2, CON3, IWISH, TABIND)  
CONTINUE  
CALL GCOFFTAB(M, N, KBIT, IWISH, TABIND)  
TYPE "CMULT= ", CMULT  
CLOSE 0  
STOP  
END
```

## VITA AUCTORIS

- 1954 Born On September 28th, Moradabad, U.P., India.
- 1975 Graduated from the Aligarh Muslim University,  
Aligarh India, With the degree of Bachelor  
of Science (Electrical Engineering)
- 1979 Graduated from the University of Saskatchewan,  
Saskatoon, Saskatchewan, with the degree of M.Sc.  
in Electrical Engineering.
- 1981 Worked as an Electronic Engineer for Advanced  
business Computer Systems Inc., Windsor, Ontario.
- 1983 Worked as a Sessional Instructor for the School of  
Computer Science, University of Windsor, Windsor,  
Ontario, Canada.
- 1984 Candidate for the degree of Doctor of Philosophy  
in Electrical Engineering, University of Windsor,  
Windsor, Ontario, Canada.