INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeib Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR
AHUJA, Sudhir Raman, 1950-
A MODULAR MULTIPROCESSOR ARCHITECTURE
FOR ARRAY PROCESSING.

Rice University, Ph.D., 1977
Computer Science

Xerox University Microfilms, Ann Arbor, Michigan 48106
RICE UNIVERSITY

A MODULAR MULTIPROCESSOR ARCHITECTURE
FOR ARRAY PROCESSING

by

SUDHIR R. AHUJA

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Thesis Director's Signature:

Houston, Texas

January, 1977
ACKNOWLEDGEMENTS

I wish to express my sincere appreciation and gratitude to Dr. J.R. Jump, my thesis advisor, for his able guidance. Dr. E.A. Feustel and Dr. K. Kennedy are to be thanked for serving on my thesis committee and for useful suggestions.

I am grateful to Jan Bodey for sharing the triumphs and the trials of my graduate education. I thank my friends, William Bain, Rick White, Pankaj Agarwal, Satish and Rohini Joshi, Rajan Kapur, Alok Sagar and Anand Jagannathan for making my stay at Rice and UCLA enjoyable.

Last, but not the least, I am thankful to Mrs. Pat Angel for going the extra mile in typing of this thesis.

This research was supported by National Science Foundation Grant DCR 74-14283.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>INTRODUCTION TO STREAM PROCESSING</td>
<td></td>
</tr>
<tr>
<td>1-1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1-2</td>
<td>Overview of Array Processing</td>
<td>6</td>
</tr>
<tr>
<td>1-3</td>
<td>Multiprocessor Architectures</td>
<td>11</td>
</tr>
<tr>
<td>1-4</td>
<td>Pipelined Systems and Parallel Pipelines</td>
<td>18</td>
</tr>
<tr>
<td>1-5</td>
<td>Stream Processing</td>
<td>27</td>
</tr>
<tr>
<td>1-6</td>
<td>Comparison</td>
<td>40</td>
</tr>
<tr>
<td>II</td>
<td>ANALYSIS OF STREAM PROCESSING SYSTEM</td>
<td></td>
</tr>
<tr>
<td>2-1</td>
<td>Introduction</td>
<td>44</td>
</tr>
<tr>
<td>2-2</td>
<td>Analysis of the Two Loop System</td>
<td>46</td>
</tr>
<tr>
<td>2-3</td>
<td>Parallelism in Stream Processors</td>
<td>73</td>
</tr>
<tr>
<td>III</td>
<td>MEMORY SYSTEMS</td>
<td></td>
</tr>
<tr>
<td>3-1</td>
<td>Introduction</td>
<td>82</td>
</tr>
<tr>
<td>3-2</td>
<td>Parallel Memories</td>
<td>84</td>
</tr>
<tr>
<td>3-3</td>
<td>Modified Simultaneous Access Scheme</td>
<td>91</td>
</tr>
<tr>
<td>3-4</td>
<td>Independent Access Scheme</td>
<td>98</td>
</tr>
<tr>
<td>3-5</td>
<td>Buffer Requirements and Analysis</td>
<td>115</td>
</tr>
<tr>
<td>IV</td>
<td>INTERCONNECTION SCHEMES</td>
<td></td>
</tr>
<tr>
<td>4-1</td>
<td>Introduction</td>
<td>133</td>
</tr>
<tr>
<td>4-2</td>
<td>Data Alignment in Array Processing</td>
<td>134</td>
</tr>
<tr>
<td>4-3</td>
<td>The Interconnection Network</td>
<td>148</td>
</tr>
<tr>
<td>4-4</td>
<td>Unscrambling a d-ordered vector</td>
<td>165</td>
</tr>
<tr>
<td>4-5</td>
<td>Data Transfer</td>
<td>171</td>
</tr>
</tbody>
</table>
CHAPTER V  APPLICATIONS

5-1  Introduction ........................................ 184
5-2  The Perfect Shuffle Interconnection ............ 187
5-3  Polynomial Evaluation ................................. 192
5-4  Fast Fourier Transform ............................... 203
5-5  Matrix Transposition ................................. 218
5-6  Linear Recurrence and Recursive Doubling ....... 221

CHAPTER VI  CONCLUSIONS

6-1  Summary of Results ................................. 233
6-2  Extension and Future Work ......................... 236

REFERENCES ............................................. 242
CHAPTER I

INTRODUCTION TO STREAM PROCESSING

1.1 Introduction

This work presents a multiprocessor architecture that is particularly suited for vector processing. The main features of the architecture are:

1. **Highly parallel**; the system works in a highly overlapped fashion leading to pipeline-like performance of the processing unit and the memory system.

2. **Highly modular**; processor modules and memory modules can be added or removed from the system without requiring any changes in the control structure or the systems programs. The processing unit appears like a single fast processor to the memory. A change in the number of processors in the processing unit is perceived only as a single fast memory module to the processing unit. Any change in the number of modules in the memory is perceived only as a change in the net memory bandwidth.

3. **Very flexible**; the modules in the processing unit are general purpose processors which execute a set of instructions defined by microprograms stored in their memories. A user can easily execute any non-standard instruction or a set of instructions by first storing the appropriate microprograms in the processors. Different sets of such microprograms can be used to tailor the system to a particular requirement of the user.

4. **Low cost**; the system is designed to have a very simple and low cost interconnection structure between the memory modules and the processor modules. Most of the necessary control is distributed so that it can be modularized. This leads to a simpler control structure.
The processor modules are envisioned to be LSI microprocessors. The cost of the modules therefore is very small compared to the processing units of current large computer systems, though the performance is comparable.

5. *Vector processor;* the system works most efficiently when the tasks performed by the processors are all of the same duration and independent. This is exactly the situation with most vector operations (they consist of several identical and independent sub-operations).

6. *A high-level language machine;* complete vectors are handled by the memory system and the processing unit. The memory system indexes through the vectors without any external commands. Vector and array operations are therefore treated as data primitives. The system implements a high level language such as APL very efficiently.

The basic architecture is developed in three steps:

   i) design of a modular processing unit comprising a large number of identical processors,

   ii) design of a modular memory system which can access and store data at a speed comparable to that of the processing unit. Further, the memory system is designed to index through vector elements without requiring an instruction every time,

   iii) design of a simple interconnection network that collects the operands from the memory and sends them to the processing unit in packets.
The system is called a Stream Processing System and is intended to be used as a special functional unit in a general computer system, as shown in Fig. 1.1. The user and systems programs as well as input and output are handled by a central control unit called the main processor. For execution of vector operations, a vector instruction is issued to the Stream Processing System, which executes the vector operation at a very high speed while the main processor proceeds to execute subsequent instructions, if any.

Stream processing essentially implies that the processors receive tasks (instruction and data) in a single stream. The initiation of processing of various processors is staggered in time thus producing a pipeline-like behaviour. The concept of Stream Processing is developed based on existing architectures in this chapter. It is shown that a Stream Processor has some of the versatility of multiprocessor architectures like VAMP [SENZIG 65], ILLIAC IV [BARNES 68], and the simple memory and interconnection requirements of pipelined processors like TI-ASC [ASC 73], CDC-STAR/100 [CDC 73].

The second chapter discusses the behaviour of a Stream Processor. A detailed analysis of its performance is presented and it is shown that a Stream Processing System with N processors has a better or the same performance as an N-stage pipeline, for vector operations. The third chapter discusses the design of the memory system. It is shown that the address indexing can be accomplished in the memory modules. The memory system allows any number of variables in a vector operation. It is shown that the addressing schemes ensure that the memory operates at maximum bandwidth, except for a small start-up time. It is also shown
FIG. 1.1 A GENERAL PURPOSE COMPUTER
that the memory system is modular.

Chapter IV discusses the interconnection structure. It is shown that a very simple interconnection structure is sufficient to ensure proper operand selection from the memory. It is also shown that the same network can handle the problem of data alignment and unscrambling of skewed vectors. Finally, the chapter illustrates that the interconnection scheme is modular and can be implemented as a part of the memory.

Chapter V illustrates how the stream system efficiently implements some special interconnection schemes and some special algorithms. In particular, implementation of the fast Fourier transform, evaluation of polynomials, solution of linear recurrence problems and matrix transposition are presented. It is shown that the system efficiently implements these algorithms for a different number of input data points without requiring any change in the number of processors, memory modules or the interconnection structure.

Chapter VI summarizes the results and presents some extensions of the basic architecture together with an indication of the directions which future research might take.
1-2 An Overview of Array Processing

The economics of the computing environment demands faster and cheaper machines. It is an accepted fact that little increase in computing speed can be expected from advances in basic hardware components technology, since the switching speeds of the logic gates already compares with the travel-delay of signals in the wires. Therefore, an increasing interest is developing in improving the throughput of computers by making use of the parallelism inherent in the problem. In order to make use of this parallelism it is necessary to

i) study and define the existence of parallelism; it is necessary to know the nature of parallelism (at program level, at data level, etc.) in order to fully take advantage of it.

ii) expose the entities that can be processed in parallel, and

iii) exploit the parallelism found in the given problem.

The various kinds of parallelism encountered in computing are loosely categorized as follows [REIGEL 70]:

i) program parallelism; two or more programs that are mutually independent can be executed in parallel,

ii) task parallelism; this refers to parallelism at subroutine procedure level,

iii) instruction parallelism; two or more instructions which do not influence each other can be executed in parallel,

iv) expression parallelism; \( A + (B+C) + (D+E) \) Here \( B+C \) and \( D+E \) can be evaluated in parallel,

v) data parallelism; depending on the definition of the data structure, parallelism can be implicit. Consider a vector operation
in APL, \( A \cdot B \cdot C \) where \( A, B, C \) are vectors. This instruction implies a set of sub-operations,

\[
\begin{align*}
& \quad \vdots \\
A[N] & \leftarrow B[N] + C[N]
\end{align*}
\]

All these sub-operations can be executed in parallel.

vi) parallelism at the microprogram level; computer instructions can be divided into several independent sub-operations at the microprogramming level; these sub-operations can be executed in parallel.

Array processing is concerned mainly with the exploitation of parallelism inherent in data structures such as vectors and arrays. The parallelism in such structures is implicit in the definition and needs no special algorithms to expose the entities that can be processed in parallel. There is a large class of problems in Signal Processing, Seismic Data Processing, Weather Forecasting and other fields that require computations to be performed on very large vectors at a very high speed. This has spurred a great interest in the design of large and high-speed vector processing machines.

Vector processing systems come under the class of single instruction scheme--multiple data stream (SIMD) processors [FLYNN 72]. SIMD processors are characterized by the fact that a single instruction is executed on different sets of data elements in different processors (or in the same functional unit at different times) thus overlapping
the processing of a large number of identical operations. The approaches taken in design of SIMD processors can be loosely categorized as follows:

1) Array Processor: Such processors have one control unit and N directly connected processing elements. Each processing element is independent, that is, has its own registers and storage, but only operates on command from the control unit. Examples of this approach are computer architectures such as SOLOMON [SLOTNICK 62], ILLIAC IV [BARNES 68].

2) Pipelined Processor: This is a time-multiplexed version of the array processor. A pipeline consists of a number of functional units, each tailored to a particular function or sub-function, which are arranged in sequence in a production-line fashion. The control unit issues a vector operation to memory. Memory is arranged so that it is suitable to a high-speed data transfer and produces the source operands which are entered, one pair every \( \Delta t \) time units, into the first functional unit. The results from the first functional unit are entered, at the same time, into the second functional unit and so on. The result stream returns to memory. Pipelined processors have much simpler interconnection between the memory and the functional units. Overall pipelined processors are more economical than array processors. Examples of pipelined processors are CDC-STAR/100 [CDC 73], TI-ASC [ASC 73].

3) Associative Processors: This is a variation of array processor. Processing elements are not directly addressed. The processing elements of the associative processor are activated when a generalised match relation is satisfied between an input register and characteristic data contained in each of the processing elements. For those designated
elements the control unit instruction is carried out. The other units remain idle.

This approach has been taken in the design of STARAN computers designed by Goodyear Aerospace Systems [STARAN 74], and the OMEN-60 designed by Sanders Associates [OMEN 74]. The performance of associative processors is similar to array processors. Associative processors generally require a special memory design to allow them to search associatively over a large section of memory. Because of the large cost of associative memories, such processors are only used in very special applications such as processing multiple radar tracks in air-traffic-control situations. A description of most of the current multiprocessor systems can be obtained in [ENSLOW 74].

The array processors and the pipelined processors are studied in more detail in the following sections on the basis of cost, performance, complexity of processor-memory interconnections and versatility (that is, ability to handle a large class of problems).

SIMD processors allow vectors and arrays (or other such data structures) to be treated as data primitives. Thus a vector operation may be treated in the same manner as a scalar operation. This reduces the software overhead and simplifies a system programmer’s task [HASSIT 73]. Further, it facilitates the implementation of high level languages like APL. Giloi and Berg [GILOI 75] have designed a system called STARLET which uses an SIMD processor to effectively implement APL at a very high level. They use two processors to keep track of vector operations, one being a data processor that executes the operations and the second being a structure processor that generates the address for
vector elements. The structure processor has three index generators which are set initially to starting addresses of three vectors, two input vectors and one result vector. The data processor employs a pipelined functional unit to achieve a high throughput. This scheme requires that vector operations can have, at maximum, two input variables and one output variable. The scheme is also non-modular.

Since it is possible to design fast vector processors and such systems are currently available, it is of interest to determine if some conventional non-vector algorithms can be reconfigured or broken up into vector operations. Parallel algorithms have been developed for the fast Fourier transform [PEASE 68], for the solution of linear recurrence problems [KOGGE 73], [TROUT 72], polynomial evaluation, sorting [STONE 71] and other such problems. However, most of these parallel algorithms require some modifications or special interconnection networks (between the processors and the memories) in order to be effectively implemented on vector processors. Efficient implementations of some of these algorithms are discussed in detail in Chapter V.
1-3 Multiprocessor Architectures

Multiple processor architectures are particularly well-suited to vector processing. A vector operation essentially consists of a set of independent scalar operations which can be executed in parallel by different processors. Consider a vector operation \( A + B \times C \) where \( A, B \) and \( C \) are vectors of length \( Z \). Then, the vector operation consists of the following scalar operations,

\[
\begin{align*}
&\vdots \\
A[z] &+ B[z] \times C[z]
\end{align*}
\]

where \( A_i, B_i \) and \( C_i \) are the \( i \)th elements of vectors \( A, B \) and \( C \), respectively. If \( N \) processors are available, then ideally \( N \) operations can be performed in parallel at one time. Thus, by using \( N \) processors it is possible to achieve an \( N \)-fold increase in throughput over that of a single processor. Similarly, an increase in memory bandwidth can be obtained by using memory modules in parallel. If the vectors are stored across \( M \) memory modules then \( M \) data elements can be fetched in one memory cycle. A memory system with \( M \) modules can produce data at a rate of \( M \) times the data rate of a single module.

Thus, a general approach to multiprocessor design is to have \( N \) processors, \( M \) memory modules and an interconnection network between the processors and the memory modules, as shown in Fig. 1.2. The interconnection network is used to route data from the memory system to the processors and results from the processors to the memory. In general, it is necessary to provide any one-to-one interconnection path between
FIG. 1.2  A GENERAL MULTIPROCESSOR CONFIGURATION
the memories and the processors such that at any given time N processors can be connected to an arbitrary set of N memory modules (assuming $M \geq N$). The necessity of such an interconnection network arises from the problem of data alignment [KUCK 75]. Consider a vector operation


$$\begin{align*}
& \vdots \\
\end{align*}$$

The processors fetch, first, $A[1]$ through $A[N]$ and then $A[4]$ through $A[N+3]$. It is clear that during the first memory fetch $P_1$ requires access to memory module $M_1$, $P_2$ requires access to $M_2$, and so on. For the second memory fetch $P_1$ requires access to $M_4$, $P_2$ requires access to $M_5$, and so on. If the required $N$ data paths are available during each memory fetch, then it can be seen that the $N$ operations can be executed in parallel, as desired. Data alignment is discussed in more detail in section 4.2.

Senzig and Smith [SENZIG 65] suggested such an architecture for array processing. They proposed $N$ memory modules and $N$ processors interconnected through a crossbar switch. A crossbar switch can provide any permutation of $N$ interconnection paths between the $N$ memory
modules and the N processors. Since all the processors perform the same operation, they are operated in lockstep. A program instruction is decoded in a central control unit which then feeds a common command to all the processors. The processors are, thus, used as powerful functional units. Since all the processors operate in lockstep, some may be idle during an operation if the number of scalar operations to be performed is less than N.

Consider the vector operation A + B x C where A, B and C are vectors of length Z. Assume it takes a time period \( \rho \) for the N processors to receive the necessary data and send to memory the results of a previous operation, if any. If \( R \) is the time per processor of execution of a scalar operation, then N scalar operations can be completed in the time \( T = R + \rho \). Note that this is also the time taken if the number of operations is less than N. In general, if N operations are executed at a time by the N processors, then the total time of operation for Z operations is

\[
T_1 = \left[ \frac{Z}{N} \right] \times T
= \left[ \frac{Z}{N} \right] \times (R+\rho)
-- E 1.1 .
\]

For large Z, \( \left[ \frac{Z}{N} \right] \approx \frac{Z}{N} \), therefore

\[
T_1 \approx \frac{Z}{N} \cdot (R+\rho) = Z \cdot \frac{(R+\rho)}{N} .
\]

The average time per scalar operation is given by

\[
T_{av} = \frac{T_1}{Z} \approx \frac{R+\rho}{N} .
\]

This implies that on the average a scalar operation is done in \( \frac{1}{N} \)th of the processing time of a single process, hence the system works at N times the rate of a single processor.
The major advantage of such a multiprocessor system is clearly the improvement obtained in the throughput of the system. Since general purpose processors are used as the functional units it is possible to execute a wide variety of operations and even sets of operations with a high degree of parallelism. The major disadvantage of this multiprocessor architecture is

1) the cost of processors

2) the cost of interconnection network.

The cost of processors can be significantly reduced by using microprocessors, however, there is no easy way to reduce the cost of the interconnection network. A crossbar switch can be set, and data can be transmitted in \( O(\log N) \) gate delays. However, the cost of such a switch is quite high, namely, \( O(n^2) \) gates [Kuck 75]. Many other networks have been studied. Lang [Lang 73] proposed a Shuffle exchange network which can transmit data in \( O(\log N) \) gate delays and has \( O(N \log N) \) gates. Lawrie [Lawrie 73] has proposed the \( \Omega \) network which requires \( O(N \log N) \) gates and can be set and transmit data in only \( O(\log N) \) gate delays. The latter two networks, however, cannot perform arbitrary permutations of the \( N \) interconnection paths. However, they do perform most of the permutations necessary for vector processing. The networks described above still suffer from the disadvantage of high cost. Another disadvantage of these networks is that they are designed specifically for \( N \) memories and \( N \) processors. They are not easily expandable. In order to include more processors, or memory modules, the interconnection network may have to be completely changed. Thus, the system is not very flexible or modular.
In order to simplify the interconnection paths required, a different approach was used in the design of ILLIAC IV [Barnes 68]. A memory module is assigned to a particular processor. A Processing Element (PE), thus, consists of a processor and a small memory unit, as shown in Fig. 1.3. Each processor is provided with interconnection paths to four of the nearest neighbours. Thus a processor can communicate with any memory module over a series of these interconnection paths. However, when data is being transferred between two PE's, all other PE's are held idle. Normally, all PEs execute the same operation in lockstep, the operation command being obtained from a central control unit. An architecture such as ILLIAC IV operated very efficiently when the processors execute operations on data elements resident in their local memories. This is because there is no set up delay or transmission delay associated with interconnection network. Consider, however, an operation of the form \( B[I] + A[I] + A[I+3] \) for \( I=1 \) to \( Z-3 \). Let the elements \( A \) be spread across the local memories with \( A[1] \) in \( PE_1 \), \( A[2] \) in \( PE_2 \) and so on. Then \( P_1 \) has to access \( A[1] \) from local memory and \( A[4] \) from \( PE_4 \). It is easily seen that the movement of \( A[4] \) from \( PE_4 \) to \( PE_1 \) through \( PE_3 \) and \( PE_2 \) requires 3 shift operations during which other processors may be idle. This has to be done to obtain \( A[I+3] \) for every PE. It is clear that more time may be spent in transfer of data than in processing. Thus this architecture is not very suitable for operations which require interconnections other than the four nearest neighbours transfers.
64 Processing Elements,
One control unit

FIG. 1.3 ILLIAC IV
1-4 Pipelined Systems and Parallel Pipelines

One of the more commonly used techniques for high speed array processing is pipelining. Vector Computers such as the CDC/STAR-100 [CDC 73], TI-ASC[ASC 73] and IBM/360-91 [AND 67] use pipelined arithmetic units to increase their vector processing speeds. Pipelining is a hardware design technique for utilizing the parallelism inherent in a vector operation. Pipelining can be used to realize an algorithm whenever that algorithm can be divided into a fixed number of steps that are to be executed in sequence. A pipelined realization of such an algorithm consists of several hardware stages separated by registers. There is one stage for each step of the algorithm and they are interconnected in the same order that the steps are executed, as illustrated in Fig. 1.4. This organisation can only be used effectively when the algorithm is to be applied repeatedly to a stream of input data. For example, in order to perform a vector multiplication it is necessary to multiply distinctly the data elements of the given vectors. Thus a multiplier could be used repetitively for Z multiplications where Z is the vector length. Consider a pipelined multiplier with n steps. In that case, the first operands are sent to stage 1 and step 1 is executed. When step 1 is completed, the intermediate results, produced as the output of stage 1, are sent to stage 2 as input for step 2. At the same time, the second set of operands are brought into stage 1 and step 1 is repeated for this new data. Thus, both stages 1 and 2 are operating concurrently during the second phase of execution. If there is a total of n stages, then after n executions of the algorithm have been initiated, all n stages will be operating concurrently, each realizing a different
FIG 1.4 LINEAR PIPELINE
execution of the algorithm. Since all of the stages of an n-stage pipeline can operate concurrently, a pipelined system has the potential to operate at \( n \) times the computation rate of an equivalent non-pipelined system. Two factors will prevent the full realization of this gain. First, the rate will be determined by the slowest stage. If this stage requires \( t \) time units, then at most one execution of the algorithm can be completed by the system every \( t \) units of time. If the entire operation takes \( R \) time units then \( t \) may be greater than \( R/N \). In effect, the pipeline will operate at a rate less than \( N \) times that of a non-pipelined system. Second, there is a delay between the initiation of the first execution and its completion during which no results are produced by the system. If the total number of executions to be performed is small, then this delay can significantly increase the average time per execution [JUMP 76].

The major advantage of pipelining over the parallel design techniques of section 1-2 is that frequently the same improvement in performance can be obtained for less cost using pipelining. This happens because i) an n-stage pipeline is obtained by partitioning a non-pipelined system into \( n \) smaller subsystems and then adding registers between the stages to hold the intermediate results. Thus, the dominant additional cost of a pipelined system is due to the added registers and this is frequently small compared to the cost of the stages.

ii) a simple interconnection network is required to transfer the data between the memory and the pipelined unit. Since data is supplied to the pipeline in a single stream, the memory has to generate only one stream of data. It can do this very simply by accessing the modules in
parallel and then transferring the data to the buffers in a serial stream. The data is then formed into relevant packets and sent in a stream to the pipelined functional unit as shown in Fig. 1.5 [ASC 73].

The major disadvantages of pipelining compared to the design techniques of section 1-3 are

a) limited versatility; only some arithmetic operations or a set of operations can be conveniently pipelined. Common pipelined operations are multiplication, division, addition, etc. However, it may be desirable to execute other operations, such as square root or reduction of a vector, which cannot be easily or efficiently pipelined. It is generally convenient to custom design a pipeline for a particular operation. Thus in order to have a functional unit with the capability of executing a large set of operations it is necessary to have a pipelined unit for every operation or for subsets of the operations. This also results in inefficient usage of the pipelined units because the only one of the pipelined units is used for one given operation. Even in a particular pipelined functional unit all the stages of the pipeline may not be used in every operation.

b) limited maximum parallelism; most arithmetic operations can be partitioned into only a small (4 to 8) number of distinct steps. For example TI-ASC has only an 8-stage pipeline [ASC 73]. Thus, in the case of ASC the maximum parallelism obtainable is limited to order of eight.

c) limited expandability; since the hardware for most pipelined operations has to be custom designed depending on the operations and the number of stages, it is very diffi-
cult to expand a pipeline or remove stages in order to enhance or lower the computational rate of a pipelined system.

It should be noted here that these disadvantages are overcome in the designs of section 1-3 by using a general purpose processor as the basic building block. It will be shown in section 1-5 that the same approach is taken in the design of a stream processing system.

The performance of pipelined systems has been analysed by Hallin and Flynn [HALLIN 72], Chen [CHEN 71], Ramamoorthy [RAMAMOORTHY 74] and the author [AHUJA 74]. The most common parameter used to evaluate the performance of a pipelined system is its throughput. It is easier, however, to obtain the average time per operation which is $= 1$/throughput. The author has shown [AHUJA74] that for a linear pipelined system of $N$ stages the average time per operation is given by

$$T_{av} = \frac{R(N-1)}{N} + Z \cdot \frac{R}{N}$$  \hspace{1cm} -- E 1.2

where, $Z = \text{vector length}$

$R = \text{total process time}$

$= \text{actual process time} + \text{total register transfer delay.}$

For very large $Z$,

$$T_{av} \approx Z \cdot \frac{R}{N},$$ implying an average execution time of $R/N$ per operation. This indicates a parallelism of $N$ since an operation is done in $\frac{1}{N}$th of the time of a non-pipelined system.

One of the ways of overcoming the disadvantage of limited maximum
parallelism is to use multiple pipelines (Fig. 1.6). This is the approach taken in the design of TI-ASC vector computer [ASC 73]. The ASC can employ up to a maximum of four pipelines. In general, we can achieve as much parallelism as desired, for vector operations, by using more pipelines in parallel. All the pipelines operate at the same rate, that is, at the rate of the slowest stage among all the pipelines. The pipelines are supplied with data in parallel and results are sent to the memory in parallel. Thus the system behaviour can be studied from the point of view of a) parallel functional units, and b) as pipelined functional units.

Consider 'r' parallel pipelined units, each consisting of p stages. Since the data is supplied in parallel to r pipelines, a pipelined unit receives only a maximum of \[ \left\lfloor \frac{Z}{r} \right\rfloor \] vector elements. The processing time for \[ \left\lfloor \frac{Z}{r} \right\rfloor \] operations, from E 1.2 is

\[ T = R \frac{(p-1)}{p} + \left\lfloor \frac{Z}{r} \right\rfloor \cdot \frac{R}{p} \]

-- E 1.3

where \( Z \) = vector length
\( R \) = complete processing time/operation of a pipelined unit.

If \( N = r \times p \), then

\[ T = R \frac{(p-1)}{p} + \frac{R}{p} \left\lfloor \frac{Z}{N} \cdot p \right\rfloor \]

-- E 1.4

Note again that if \( Z \) is very large then

\[ T \approx \frac{R}{p} \cdot \frac{Z \cdot p}{N} = Z \cdot \frac{R}{N} \]. Thus for very large vectors, parallelism of the order of \( N \) is achieved. The inherent parallelism ob-
FIG. 1.6 PARALLEL PIPELINES
tainable in pipelined systems, parallel pipelines, and the design techniques of section 1-3 is the same for an equivalent number stages and processors. The differences lie in

i) cost of hardware,

ii) versatility (types of operations possible; expandability, etc.),

iii) maximum parallelism (i.e., how large can N be?).

We have seen that by using parallel pipelines we could theoretically obtain N as large as desired, thus obtain as much parallelism as desired. However, parallel pipelines incur the disadvantages of

i) limited versatility (from use of pipelined units), and

ii) cost of hardware; this stems not only from the hardware required to implement the multiple pipelines but also from the interconnection network which increases in complexity at least as \( O(r \log r) \).
1.5 *Stream Processing*

It was observed in the preceding sections that a considerable increase in the throughput of a processing system is obtained by using a large number of processors in parallel. It was also pointed out that the major drawbacks of this scheme were

a) cost of the processors

b) cost of the interconnection network

With the availability of low cost microprocessors it is feasible to consider architectures involving a large number of processors. However, the cost of the complex interconnection networks required for schemes described earlier (section 1.3) is still a major drawback. It was shown in section 1.4 that pipelined systems have a very simple interconnection scheme but suffer from limited parallelism. It is then natural to ask if the two approaches (section 1.3 and section 1.4) can be combined to suggest a low cost but versatile parallel processing scheme.

In this section we suggest one such scheme which incorporates a large number of processor modules (using microprocessors) and the simple interconnection scheme used in pipelined systems. Fig. 1.7 illustrates the scheme. As shown, the memory generates only a single stream of data packets which is sent to the processing system on the single data path. The processing system comprises a large number of processors, all of them use the same data path for communication with the memory. Only one processor may access the data bus at a given time. In order to ensure this, there is a central bus control or some kind of distributed control which resolves the contention for the bus amongst several processors and
FIG. 1.7 A STREAM PROCESSOR
assigns the bus to one particular processor. Thus the processors time share the bus for input and output but execute operations in parallel. All free processors, then, put up their request for the bus. The bus control assigns the bus to one processor. This processor will output the results of a previous operation, if any, and receive as input a data packet and a command packet specifying the operations to be performed on the data. Once the processor has received all its data, it releases the bus and starts execution on its data. The bus is now assigned to some other free processor which receives the next data packet and the command packet. It then releases the bus and starts execution. The bus is then assigned to the next free processor, and so on. It is clear that the input/output phase of a processor is overlapped (executed in parallel) with the execution phases of other processors and that the execution phases of all the processors overlap to a large extent.

Fig. 1.7 shows that the processing system has an open-ended configuration. This implies that processors may be added or removed without requiring any changes in the memory scheme or the control structure. This is also facilitated by the fact that the data packets contain all the information for execution of an operation, hence the packets are not addressed to any specific processor. This allows the processing system to be completely modular, the processor modules being added or removed from the system without requiring any changes in the rest of the system. Indeed, to the memory the processing system appears as a single fast processor.

The operations performed by the processors are scalar operations such as addition, multiplication, or groups of these operations. All
operations or groups of operations are represented by single command words which are sent to all the processors at the beginning of the operation. Thus the processors know what instruction to execute on the data that is received subsequently. Once the command word has been sent, the data packets are sent in a stream to the processors. For example, in case of the vector operation $A + B \times C$, the command multiply is sent initially. Then, the input data packets, $\langle a_1, b_1 \rangle$, $\langle a_2, b_2 \rangle$, ..., $\langle a_z, b_z \rangle$ are sent to the processors. Each processor receives only one data packet at a time and performs the required computation. The results from the processors are sent back to the memory in the sequence $\langle c_1 \rangle$, $\langle c_2 \rangle$, ..., $\langle c_z \rangle$.

In order to accomplish the above efficiently a two bus system is used, as illustrated in Fig. 1.8. The processors are assigned to the busses by two polling schemes; the access to the busses being controlled by two control loops. The control loops are implemented as cyclic bit shift registers. Each control loop has only one 'logical bit' circulating in it. The control loop, which controls access of the input bus, is called the input control loop and the circulating bit is called the input control bit or I-bit. The control loop which controls the access of the output bus is called the output control loop and the circulating bit in it is called the output control bit or the 0-bit. Whichever processor has the I-bit, at a given time, has access to the input bus. When the processor has received a data packet, the I-bit is shifted, cyclically, to the next processor. Similarly, whichever processor has the output bit (0-bit) gets access to the output bus. Once the processor
has finished the transfer of the result packet, the 0-bit is shifted, cyclically, to the next processor. The bus control for each bus is thus distributed and no central bus control is required for this scheme. The control loop (shift register) is easily implemented by using a single latch in each interface unit and connecting the latches as a shift register. This requires only one control line from a processor to its neighbour. The state of the flip-flops tell the interface if it is allowed access to any of the busses. If an access is allowed then the interface allows the processor to signal on the data control lines, discussed in Chapter IV, and request for data packets or indicate that the results are ready to be sent. The I-bit (or the 0-bit) stay at a processor until it has finished an input data transfer (or output data transfer). If the processor is busy, the control bit just waits at that processor until the processor finishes processing and then completes a data transfer. Clearly, the scheme allows the processors to be scheduled in the cyclic sequence \( P_1, P_2, \ldots, P_{N-1}, P_N, P_1, P_2, \ldots, P_{N-1}, P_N, \ldots \). The output control loop ensures that the results are sent in that sequence also. Hence, the behaviour of the polling scheme is similar to that of the Left Priority Scheme; the performance and the behaviour of the polling scheme is discussed in more detail in section 2.2. The advantages of the two-loop scheme with respect to scheduling of the processors are:

i) It uses a very simple distributed control for the busses. Only one control line is required. No central bus control is needed.
ii) It requires very small time for shift of control. Clearly, in this scheme there is no arbitration required as well as no signalling for initiating request and asserting control of the bus. The time required for the relocation of control is merely that of a shift of a bit.

iii) It takes a uniform time for relocation of control of a bus. Since the time taken for relocation of control is always that of a shift by one place in the control loop, the time taken is the same irrespective of where the control bit initially resides. In other bus schemes with a central bus control the time for relocation of the control of the bus was dependent on the distance of the processor (which receives the control) from the bus control.

The behaviour of the scheme is best illustrated by an example. Consider a two-loop scheme having N processors, P_1 to P_N as shown in Fig. 1.8. Let the time taken to transfer a data packet from the input bus to a processor be r clock cycles. Let the time taken to transfer the result packet from a processor the output data be, also, r cycles. The two times are taken to be the same for simplicity of description. An analysis of all the other cases is presented in section 2-2. The processing time of a data packet in a processor is assumed to be R. Let the length of the vector operation by Z. Then there are Z data packets to be processed. Initially both the control bits are at processor P_1.

The Figure 1.9 illustrates the behaviour of the system for N = 3. For each processor, three phases are plotted as a function of time. The three phases are i) input; this indicates the time during which input data is transferred to that processor, ii) processing; this indicates
FIG. 1.9  Behaviour of the Two-Loop System for $R < (N-1).r$
the time during which \( P_1 \) is processing some data packet, and \( \text{iii)} \) output; this plot indicates the time during which the processor transfers the result packet on the output bus.

The superscript on \( R \) in the processing phases of the processor indicates which data packet is being processed at that time. The superscript on \( \gamma \) indicates which data packet is being input (or output) during those data transfer phases.

Since the input control bit is at \( P_1 \), it gets the first data packet. If this transfer is initiated at time \( 'T_0' \) then \( P_1 \) receives the first packet by the time \( (T_0+\gamma) \). The I-bit is then shifted to \( P_2 \). Processor \( P_2 \) receives the second data packet in the next \( \gamma \) cycles, that is, by the time \( (T_0+2\gamma) \). The I-bit is then shifted to processor \( P_3 \). \( P_3 \) receives the third data packet in the next \( \gamma \) cycles, by the time \( (T_0+3\gamma) \). The I-bit is shifted to \( P_4 \), and so on. Processor \( P_N \) then receives the \( N \)th packet by the time \( (T_0+N\gamma) \), after which the I-bit goes back to \( P_1 \). Since \( P_1 \) started processing at time \( (T_0+\gamma) \) it will finish processing \( R \) cycles later by the time \( (T_0+\gamma+R) \). If this time is greater than \( (T_0+N\gamma) \), that is \( R>(N-1)\gamma \), then \( P_1 \) is busy processing the first packet when I-bit comes back to \( P_1 \). When processor \( P_1 \) finishes processing the first packet, it can input the \((N+1)\)st packet as well as output the results of the first packet in parallel since it has both the control bits, as shown in Fig. 1.9, \( \gamma \) cycles later both the control bits will be shifted to processor \( P_2 \). So \( P_2 \) receives the control bit at time \( (T+R+\gamma) + \gamma \). By this time \( P_2 \) has finished processing the second packet, so it will output the results of the second packet and input the \((N+2)\)nd data packet in
the next r cycles, that is, by the time \((T+R+2r) + r\). After this time both the control bits are shifted to the third processor, and so on.

It should be noted that the 0-bit always travels along with the 1-bit so there is no necessity of incorporating separate output control loop. The I-bit could represent both the control functions. However, the separate 0-bit is necessary where \(R < (N-1)r\). This case is illustrated in Fig. 1.10. In this case processor \(P_1\) finishes processing the first packet before the I-bit comes back around to it. If only one control loop was used then \(P_1\) can output the results only after the I-bit gets back to \(P_1\), that is, after the time \((T_0+Nr)\). This effectively increases the processing time of a packet. The packet stays in the processing system longer than the time ideally required.

In order to avoid this additional delay, the second control loop is provided. Since \(P_1\) has the 0-bit when it finishes processing the first packet, it transfers results in \(p\) cycles immediately after it finishes processing, that is, by the time \((T_0+r+R+r)\). The 0-bit is then shifted to \(P_2\). Since \(P_2\) finishes processing by the time \((T_0+R+2r)\), it has the 0-bit as soon as it finishes processing. Hence it transfers results in the next \(r\) cycles, by the time \((T_0+R+2r+r)\). The 0-bit is then shifted to processor \(P_3\). It can be seen that each processor can output the results of a packet immediately after it finishes processing the packet. The output loop thus allows this scheme to achieve the same or better performance as the left priority scheme for the case \(R < (N-1)r\).

Each packet, now, stays in the processing system for a total time of \((r+R+r)\) cycles which is less than \((R+Nr)\) cycles with a single loop scheme.
FIG. 1.10 Behaviour of the Two-Loop System for $R \geq (N-1)r$
From Fig. 1.10 it can be seen that since $P_1$ finishes the data transfer by the time $(T_0 + R + r)$, it is idle till it receives the I-bit. In general, each processor will be idle for some time or not transfer the input and output packets completely in parallel. However, it should be noted that both the data busses are busy all the time, since a transfer takes place every $r$ cycles on each bus. This implies that the system has more processing power than required to maintain a maximum data transfer rate on the data busses. It is shown in section 2-2 that for this case ($R < (N-1)r$) the effective number of processors used is given by

$$N_{max} = \frac{R + r}{r}.$$  

It is shown in section 2-2 that for Z operation, the total time of processing is

a) \[ T = Zr + R + r + \left( \frac{Z}{N} - 1 \right) (R - (N-1)r) \] for $N < N_{max}$

\[ \text{That is } N < \frac{R + r}{r} \]

and b) \[ T = Zr + R + r \] for $N > N_{max}$.

--- E 1.5

For a given operation, that is, a given $R$ and for a given $r$, one can always add enough processors to obtain

$$T = Zr + k + r$$ for a Z length vector operation. It should be noted that this time of operation is independent of the number of processors, so increasing $N$ beyond $N_{max}$ doesn't give any lower time of operation. The minimum time of operation for Z operations is then

$$T_{min} = Zr + R + r.$$
Then, \( T_{av} = \text{time of execution per operation} \)
\[ = \frac{T_{\text{min}}}{Z} = r + \frac{R+r}{Z} \]  \( -- E \ 1.6 \)

If \( Z \) is very large,
\[ T_{av} \approx r . \]

This implies that the processing system effectively processes one data packet in the time it takes to ship the data packet out to the processors and get the results back. This effectively implies 'zero' processing time. This is the upperbound on thruput due to limitation on transferring the data on the bus in a single stream.

We have seen, thus, that with \( N > N_{\text{max}} \) and large \( Z \) the processing system will almost reach the ideal maximum thruput possible from the memory. This also suggests that the final bottleneck is the data transfer rate. If that can be made larger (i.e., \( r \) smaller compared to \( R \)) one could always add more processors to achieve the new maximum possible thruput.
6 Comparison

It was seen that stream processing can indeed lead to a highly parallel but low-cost system. The stream processor has characteristics similar to pipelined systems in the sense that it executes a stream of operations and has an initial build up time and has to be flushed out at the end. It also behaves like the general parallel-processor configuration in the sense that it incorporates separate processors for execution of operations thus achieving a more versatile system. We would thus expect its performance to be quite close to both pipelined systems and a parallel-processor system.

If the memory system is assured to supply the data to the processing system whenever so required, the performances of the three systems is essentially given by expressions E 1.1, E 1.2 and E 1.5. These expressions are evaluated for a specific example of N=8, p = 8, R = 12, ρ = 1 for varying Z. The resulting values are given in Table 1.1 and are illustrated in Fig. 1.1. The time of operation is plotted against the number of operations being performed.

The main feature to be noted in Fig. 1.1 is that all the architectures have the same asymptotic slope. This implies that all the schemes exhibit the same degree of parallelism and if Z is large enough the performance of all those schemes would be almost identical. Thus, it is not any more advantageous to use 8 processors in parallel than to use an 8-stage pipeline or an 8-module stream processor, from the point of view of performance. The question of selection of a system should, then, be based on cost and versatility. As discussed earlier, stream processors have some of the versatility of general parallel processor schemes and
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>13.5</td>
<td>12</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>15</td>
<td>12</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>16.5</td>
<td>12</td>
<td>15</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>18</td>
<td>18</td>
<td>16</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>19.5</td>
<td>18</td>
<td>17</td>
<td>72</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>22.5</td>
<td>18</td>
<td>19</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>108</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
<td>25.5</td>
<td>24</td>
<td>25</td>
<td>120</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
<td>27</td>
<td>24</td>
<td>26</td>
<td>132</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>28.5</td>
<td>24</td>
<td>27</td>
<td>144</td>
</tr>
<tr>
<td>13</td>
<td>24</td>
<td>30</td>
<td>30</td>
<td>28</td>
<td>156</td>
</tr>
<tr>
<td>14</td>
<td>24</td>
<td>315</td>
<td>30</td>
<td>29</td>
<td>168</td>
</tr>
<tr>
<td>15</td>
<td>24</td>
<td>33</td>
<td>30</td>
<td>30</td>
<td>180</td>
</tr>
<tr>
<td>16</td>
<td>24</td>
<td>34.5</td>
<td>30</td>
<td>31</td>
<td>192</td>
</tr>
<tr>
<td>17</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>204</td>
</tr>
</tbody>
</table>

N = 8 \quad p = 1

p = 8 \quad R = 11
FIG. 1.11
the simplicity and low cost of a pipelined system. It then appears to provide the best combination for Vector Processing.
CHAPTER II
ANALYSIS OF STREAM PROCESSING SYSTEM

2-1 Introduction

This chapter examines the behaviour of the Two-Loop Scheme in detail. It was shown in section 1.5 that the behaviour of Two-Loop Scheme is very close to that of a pipelined processor. However, only one case, the input packet transfer time being equal to the output packet transfer time, was considered. In section 2-2 all the cases, for various values of packet transfer times and processing time, are examined. It is shown that the system behaviour essentially still remains the same as described in section 1-5. It is observed that the data transfer time is governed by the larger of the input packet transfer time \( r_1 \) and the output packet transfer time \( r_2 \).

Section 2-2 also analyses the performance of the system in terms of total time of operation for a given vector operation. It is shown that the total time for processing a vector of length \( Z \) is given by

\[
T_Z = Z\max(r_1, r_2) + R + \min(r_1, r_2) + \left( \left[ \frac{Z}{N} \right] - 1 \right) (R + (N-1) \max(r_1, r_2)),
\]

where,
- \( R \) = processing time per processor,
- \( N \) = number of processors,
- \( r_1 \) = time to transfer an input packet,
- \( r_2 \) = time to transfer an output packet.

Section 2-3 discusses the parallelism achieved by the Two-Loop Scheme. It is shown that the effective parallelism increases with the vector lengths and that for the case \( R < (N-1) \max(r_1, r_2) \) the maximum parallelism is less than \( N \). It is shown that the maximum number of useful
processors in a Stream System is given by

\[ N_{\text{max}} = \left\lceil \frac{R + r_{\text{max}}}{r_{\text{max}}} \right\rceil, \text{ where } r_{\text{max}} = \max(r_1, r_2). \]

When the number of processors are greater than or equal to \( N_{\text{max}} \), then the Stream System is operating at its maximum capacity, dictated by data transfer times; the throughput being one computation every \( r_{\text{max}} \) time units. This is the maximum throughput because data cannot be transferred between the processors and the memory system any faster.
2.2 Analysis of the two-loop system.

This section analyses the performance of the two-loop system. The performance is measured in terms of the total time of operation required to execute a vector operation of a given length.

**Notations and Assumptions**

i) The term 'processing' refers to the execution of a scalar operation on a data packet in a processor.

ii) The term 'execution of a packet' refers to the following three operations: a) transfer of input data from memory to the processor, b) 'processing' of the data packet, c) transfer of the results of this operation from the processors to the memory.

iii) The term 'an event finishes by the time T' is taken to mean that the event finishes in the Tth clock cycle. All time is measured in terms of machine clock cycles.

iv) The term 'an event starts at time T' implies that the event starts in the (T+1)st clock cycle, immediately after the end of Tth clock cycle.

v) The ith processor in the loop is denoted by $P_i$.

vi) The ith packet for a given vector operation is denoted by $p_i$.

vii) The 'execution time' of a packet, $P_i$, refers to the time
when execution of $p_i$ terminates. It is denoted by $T_i$.

viii) $T_i$ denotes the time when the processing of the $i$th packet terminates.

ix) The processing time of a packet is assumed to be $R$ clock cycles.

x) The time taken to transfer an input data packet to a processor is assumed to be $r_1$ cycles.

xi) The time taken to transfer an output data packet from the processor to the memory is assumed to be $r_2$ cycles.

xii) The vector operation begins at time $T_0$. That is, the transfer of the first packet is initiated at time $T_0$.

The analysis of the system is presented in stages. First, the performance of the system is analysed for vectors of lengths less than or equal to $N$. Next, the performance is analysed for vectors of lengths less than or equal to $2N$. From these analyses the expressions for the total time of operation for arbitrary-length vectors is obtained. It is shown that if $Z$ is the length of the vectors, that is, the total number of data packets, then the total time for operation is given by

$$T_Z = T_0 + R + Z \max(r_1, r_2) + \min(r_1, r_2) + (\lceil \frac{Z}{N} \rceil - 1)(R - (N - 1)) \max(r_1, r_2)$$

where $(x-y) = x-y$ for $x \geq y$, else is equal to 0.

We first present some observations which follow from the description of the scheme. These observations are used later in the proofs of various lemmas and theorems.

Observation 1: If at any given time processor $P_i$ has a control bit, then the next processor to get that control is $P_{i+1}$, for $1 \leq i < N$. If
\( P_N \) currently has a control bit then the next processor to get that control bit is \( P_1 \).

**Observation 2:** If at the start of a vector operation both the input control bit and the output control bit are at processor \( P_1 \) then a processor \( P_i \) will process the data packets \( P_i, P_{i+N}, P_{i+2N}, \ldots \).

**Observation 3:** If the transfer of the first data packet, to processor \( P_1 \), is initiated at time \( T_0 \) then for all \( i \) in the range \( 1 \leq i \leq N \), the \( i \)th packet is completely transferred to processor \( P_i \) by the time \( (T_0 + \text{ir}_1) \).

The following lemma gives the performance of the two-loop system for the first \( N \) packets:

**Lemma 2.1** For a given vector operation the processing of data and the transfer of results of the \( i \)th packet is completely finished by the time \( T_i \), where

\[
T_i = T_0 + R + i \cdot \max(r_1, r_2) + \min(r_1, r_2)
\]  \( \ldots \) **E 2.1**

for \( 1 \leq i \leq N \).

The system behaviour is determined by the slower of the two control loops, that is, the larger of \( r_1 \) and \( r_2 \). Fig. 2.1 shows the case where \( r_1 \geq r_2 \). That is, \( \max(r_1, r_2) = r_1 \) and \( \min(r_1, r_2) = r_2 \). It can be seen from the figure that

\[
T_i = T_0 + R + \text{ir}_1 + r_2 , \quad \text{that is,}
\]

\[
T_i = T_0 + R + i \max(r_1, r_2) + \min(r_1, r_2) .
\]

Fig. 2.2 shows the case where \( r_2 \geq r_1 \). Again it can be seen from the
FIG. 2.2
figure that
\[ T_1 = T_0 + R + i r_2 + r_1, \] that is
\[ T_1 = T_0 + R + i \max(r_1, r_2) + \min(r_1, r_2), \] since
\[ \max(r_1, r_2) = r_2 \text{ and } \min(r_1, r_2) = r_1. \]
Thus in both cases
\[ T_1 = T_0 + R + i \max(r_1, r_2) + \min(r_1, r_2). \]
This expression also gives the execution time \( T_z \), for a vector of length \( Z \), where \( 1 \leq Z \leq N \). That is,
\[ T_z = T_0 + R + Z \max(r_1, r_2) + \min(r_1, r_2). \] \(- \text E 2.2\)

For \( Z > N \), the two cases \( r_1 \geq r_2 \) and \( r_1 < r_2 \) are analysed separately.
Lemma 2.2 gives the performance of the system for the case of \( r_1 \geq r_2 \) and
Lemma 2.3 gives the performance of the system for the case of \( r_1 < r_2 \).
Theorem 2.1 combines the results of all the lemmas to give the performance of the system for all values of \( Z, r_1 \) and \( r_2 \).

**Lemma 2.2** For \( r_1 \geq r_2 \),
a) the \((N+i)\)th packet is completely executed, by processor \( P_i \), by the time
\[ T_{N+i} = T_i + N r_1 + [R - (N-1)r_1], \]
\[ = T_0 + R + (N+i)r_1 + r_2 + [R - (N-1)r_1], \] \(- \text E 2.3\)
for all \( i \) such that \( 1 \leq i \leq N \), where
\[ [R - (N-1)] = R - (N-1)r_1 \text{ for } R \geq (N-1)r_1 \]
and \[ = 0 \text{ for } R < (N-1)r_1. \]
b) the \((kN+i)\)th packet is completely executed by the time
\[ T_{kN+i} = T_1 + k[Nr_1 + [R^2(N-1)r_1]] , \quad E \text{ 2.4} \]
where \(k\) is an integer \(\geq 0\).

**Proof**

a) The proof is by induction on \(i\).

**Basis step** \(i = 1\). It is necessary to show that \((N+1)\)st packet is completely executed by the time
\[ T_{N+1} = T_1 + Nr_1 + [R^2(N-1)r_1] . \]

In order to do that it is necessary to find out

i) the time at which \(P_1\) gets the I-bit and receives the \((N+1)\)st packet,

ii) the time when \(P_1\) finishes the execution of the previous packet, \(p_1\), so it can start transfer and processing of the \((N+1)\)st packet,

iii) the time at which \(P_1\) gets the 0-bit so it can output the results of the \((N+1)\)st packet.

\(P_1\) will receive the \((N+1)\)st packet only if it has the I-bit, that is access to the input bus, and it has finished processing the previous packet, \(p_1\). \(P_1\) receives the I-bit after \(P_N\) finishes the transfer of the \(N\)th packet. From Observation 3, \(P_N\) finishes the transfer of the \(N\)th packet by the time \((T_0 + Nr_1)\). So the I-bit is shifted to \(P_1\) after time \((T_0 + Nr_1)\). From Observation 3, \(P_1\) received the first packet by the time \((T_0 + r_1)\).

\(P_1\) finishes processing \(R\) cycles after it receives the packet, that is, by the time \((T_0 + R + r_1)\). Then, the data transfer of the \((N+1)\)st packet will be initiated after time \(T_0 + R + r_1\) if \(T_0 + R + r_1 \geq T_0 + Nr_1\).
That is, \( R \geq (N-1)r_1 \), as shown in Fig. 2.3. If \( R < (N-1)r_1 \), then 
\[ T_0 + Nr_1 > T_0 + R + r_1 \], and the transfer will begin after \( (T_0 + Nr_1) \), as shown in Fig. 2.4. The data transfer will be initiated after the time

i) \( (T_0 + R + r_1) = T_0 + Nr_1 + [R-(N-1)r_1] \), for \( R \geq (N-1)r_1 \) and

ii) \( T_0 + Nr_1 \), \( R < (N-1)r_1 \)

The two expressions can be combined to give the time of start of the transfer as \( T_0 + Nr_1 + [R-(N-1)r_1] \) for all \( R \).

The processor \( P_1 \) finishes data transfer \( r_1 \) cycles later and is ready to process the \((N+1)\)st packet by the time, \( T_a \), where

\[ T_a = T_0 + Nr_1 + r_1 + [R-(N-1)r_1] \] . The processing may begin if both input of the \((N+1)\)st packet and output of the first packet are finished.

Since initially \( P_1 \) already has the 0-bit it transfers the results of the first packet immediately after \( P_1 \) finishes processing the first packet, that is after time \( (T_0 + R + r_1) \). So the output is finished \( r_2 \) cycles later by the time \( (T_0 + R + r_1 + r_2) \). If \( R < (N-1)r_1 \) then the output time \( T_0 + R + r_1 + r_2 < T_a = (T_0 + Nr_1 + r_1) \) because \( r_2 \leq r_1 \). If \( R \geq (N-1)r_1 \) then again \( (T_0 + R + r_1 + r_2) < T_a = (T_0 + R + r_1 + r_1) \) because \( r_2 \leq r_1 \). In either case, \( P_1 \) finishes the transfer of the results of the first packet before it finishes the transfer of input data for the \((N+1)\)st packet. So the processing of the \((N+1)\)st packet begins after \( T_a = (T_0 + Nr_1 + r_1 + [R-(N-1)r_1]) \) and finishes \( R \) cycles later by the time \( T_{N+1}' = T_0 + (N+1)r_1 + R + [R-(N-1)r_1] \). From Lemma 2.1 \( P_N \) finishes result transfer of the \( N \)th packet by the time \( (T_0 + Nr_1 + R + r_2) \) which is \leq
\[ r_i \geq r_j \]
\[ R < (N-1)r_1 \]

FIG. 2.4
\[ T_0 + (N+1)r_1 + R + [R^2(N-1)r_1] = T'_{N+1} \]

So the 0-bit will be, again, available to \( P_1 \) before it finishes processing the \((N+1)st\) packet. \( P_1 \) then outputs the results immediately following the end of processing, in the next \( r_2 \) cycles. Thus, the result transfer of \((N+1)st\) packet is completed by the time, \( T'_{N+1} \), where

\[ T'_{N+1} = T_0 + R + (N+1)r_1 + [R^2(N-1)r_1] + r_2 . \quad \text{E 2.5} \]

**Induction hypothesis:** The \((N+i)th\) packet is received by processor \( P_i \) by the time \( T_0 + (N+i)r_1 + [R^2(N-1)r_1] \) and completely executed by the time

\[ T_{N+i} = T_0 + (N+i)r_1 + R + [R^2(N-1)r_1] + r_2, \quad i < N. \]

It was shown earlier that the hypothesis holds for \( i = 1 \). It remains to be shown that the hypothesis holds for the \((N+i+1)st\) packet.

It follows from Observation 2 that \( P_{i+1} \) receives the \((N+i+1)st\) packet. Processor \( P_{i+1} \) may initiate the transfer of \((N+i+1)st\) packet only if it has the I-bit and it has finished processing the previous packet, \( P_i+1 \). Since \( P_i \) releases the bus at time \( T_0 + (N+i)r_1 + [R^2(N-1)r_1] \), processor \( P_{i+1} \) gets the I-bit at this time. Denote this time by \( T_i' \).

From Lemma 2.1, processor \( P_{i+1} \) finishes the processing of \((i+1)st\) packet by the time \( T_i' + 1 \), where \( T_i' + 1 = T_0 + R + (i-1)r_1 \). Thus, processor \( P_{i+1} \) will initiate the transfer of the \((N+i+1)st\) packet after the larger of \( T_i \) and \( T_i' + 1 \).

Consider \( R < (N-1)r_1 \). Then,
\[ T_i = T_0 + (N+i)r_1 + [R^2(N-1)r_1] \]
\[ = T_0 + (N+i)r_1 \quad \text{for} \quad R < (N-1)r_1 \]
\[ = T_0 + (i+1)r_1 + (N-1)r_1 \]
\[ > T_0 + (i+1)r_1 + R = T'_{i+1} \]

Consider \( R \geq (N-1)r_1 \)

\[ T_i = T_0 + (N+i)r_1 + R - (N-1)r_1 \quad \text{for} \quad R \geq (N-1)r_1 \]
\[ = T_0 + R + (i+1)r_1 = T'_{i+1} \quad . \text{Thus for all } R, \]
\[ T_i > T'_{i+1} \quad . \]

Therefore, \( P_{i+1} \) receives the \((N+i+1)st\) packet in \( r_1 \) cycles after \( T_i \), that is, by the time

\[ T_0 + (N+i)r_1 + [R^2(N-1)r_1] + r_1 \quad . \]

Processor \( P_{i+1} \) finishes the processing \( R \) cycles later by the time

\[ T'_{N+i+1} = T_0 + (N+i+1)r_1 + R + [R^2(N-1)r_1] \quad . \text{Since } P_{i+1} \]
gets the 0-bit after \( T'_{N+i} \), and \( T'_{N+i+1} \geq T_{i+N} \), processor \( P_{i+1} \) has the 0-bit before it finishes processing. So it can output the results in \( r_2 \) cycles following \( T'_{N+i+1} \), that is by the time, \( T_{N+i+1} \), which is given by

\[ T_{N+i+1} = T'_{N+i+1} + r_2 = T_0 + (N+i+1)r_1 + R + [R^2(N-1)r_1]+r_2 \quad . \]

Note that \( T_{N+i+1} \) in part a) is
\[
T_0 + R + i_r_1 + N_r_1 + r_2 + [R_z(N-1)r_1]
= T_0 + R + i_r_1 + r + N_r + [R_z(N-1)r_1]
= T_i + N_r_1 + [R_z(N-1)r_1].
\]

The term \(N_r_1 + [R_z(N-1)r_1]\) is a constant which is independent of the value of \(i\). Therefore the second set of \(N\) packets finish in the same staggered sequence, with \(r_1\) cycles between their finishing times, as the first \(N\) packets. Hence, the set of initial conditions faced by the third set of \(N\) packets will be the same as those for the 2nd set of \(N\) packets. It follows then that the finishing time of a packet belonging to the third set of \(N\) packets, say \(p_{2N+i}\), will be more than \(T_{N+i}\) by the same constant. That is
\[
T_{2N+i} = T_{N+i} + [N_r_1 + R_z(N-1)r_1],
= T_i + 2[N_r_1 + R_z(N-1)r_1], \quad \text{for } 1 \leq i \leq N.
\]

Each processor then processes a packet every \(N_r_1 + [R_z(N-1)r_1]\) cycles. The same reasoning can be extended to the \((KN+i)\)th packet, which is processed by \(P_i\). Then,
\[
T_{KN+i} = T_{(K-1)N+i} + N_r_1 + [R_z(N-1)r_1]
= T_{(K-2)N+i} + 2[N_r_1 + [R_z(N-1)r_1]]
= \ldots
= T_i + K[N_r_1 + (R_z(N-1)r_1)]
\]

--- E 2.6
\[ = T_0 + R + i r_1 + r_2 + KNr_1 + K[R^2(N-1)r_1] \]
\[ = T_0 + R + (KN+i)r_1 + r_2 + K[R^2(N-1)r_1] \] .

--- E 2.7

proved

Lemma 2.6  Given that \( r_1 \leq r_2 \),

a)  the \((N+i)\)th data packet is completely processed and its results
    transferred by the time, \( T_{N+i} \), where

\[ T_{N+i} = T_i + N r_2 + [R^2(N-1)r_2] \]
\[ = T_0 + R + r_1 + (N+i)r_2 + [R^2(N-1)r_2] \] ,  --- E 2.8

for all \( i \) such that \( 1 \leq i \leq N \).

b)  The \((KN+i)\)th packet is completely processed and its results
    are transferred by the time

\[ T_{KN+i} = T_i + K [ N r_2 + R^2(N-1)r_2 ] \]  --- E 2.9

for \( K \geq 0 \).

Proof  a)  The proof is by induction on \( i \).

Basis step: \( i = 1 \). From Observation 3, \( P_N \) finishes input data
transfer by the time \( T_0 + Nr_1 \). After this time the I-bit goes
back to process \( P_1 \). Thus \( P_1 \) receives the \((N+1)\)st data packet after

time \( T_0 + Nr_1 \) if \( P_1 \) is free at that time. If \( P_1 \) is busy then it
receives the data packet after it finishes processing, that is, by the time \(T_0 + R + r_1\). Using the same argument as used earlier in the proof of Lemma 2.2, it can be shown that the data transfer of the \((N+1)\)st packet is initiated by the time \(T_0 + N + r_1 + \lfloor R + (N-1)r_1 \rfloor\), and is completed \(r_1\) cycles later by the time \(T_0 + (N+1)r_1 + \lfloor R + (N-1)r_1 \rfloor\).

--- E 2.10

In order to find out when the results of the \((N+1)\)st packet are transferred it is necessary to know

i) the time, \(T_{N+1}'\), at which processing of the \((N+1)\)st packet terminates,

and ii) the time, \(T_N\), at which the 0-bit is available to \(P_1\) for the transfer of results.

Clearly, the larger of these two times will be the time after which the results can be transferred. The latter of these times is available from Lemma 2.1. Processor \(P_N\) finishes processing by the time

\[ T_N = (T_0 + R + r_1 + N) , \text{ for } r_2 > r_1 \]

--- E 2.11

The expression E 2.10 gives the time, \(T_2\), at which the processor \(P_1\) has completed the input data transfer. The processing can start after \(T\) if the output of the first packet is finished by this time (\(T_1 < T_2\)). Otherwise the processing will begin after the output transfer of the first packet is completed, that is by the time \(T_1\).

Case 1) If \(R \geq (N-1)r_1\) then \(T_0 = T_0 + R + 2r_1\). Therefore,

\[ T_1 = T_0 + R + r_1 + r_2 > T_0 + T_0 + R + r_1 + r_1 . \]
Processing will then begin after $T_1$. The processing of the 
$(N+1)$st packet is completed $R$ cycles later by the time $(T_1 + R)$.

**Case 2)** If $R < (N-1)r_1$, then $T_a = T_0 + (N+1)r_1$

Then, $T_1 = T_0 + R + r_1 + r_2 \geq T_a = T_0 + (N+1)r_1$ only if $R + r_2 \geq Nr_1$

If $R + r_2 < Nr_1$, then $T_1 < T_a$.

Thus, the processing will begin after $T_1$ if $R \geq Nr_2 - r_2$ and after $T_a$ if $R < Nr_1 - r_2$.

The processing thus terminates by the time, $T'_{N+1}$, where

$$T'_{N+1} = (T_1 + R) \text{ for } R \geq Nr_1 - r_2 \text{ (this includes the range } R \geq (N-1)r_1).$$

$$T'_{N+1} = (T_a + R) \text{ for } R < Nr_1 - r_2 .$$

It is easily shown that $T_N = (T_0 + R + r_1 + Nr_2) > T'_{N+1} = T_a + R$.

This follows from $Nr_2 > Nr_1$

Therefore $T_0 + R + r_1 + Nr_2 > T_0 + R + r_1 + Nr_1$

$$> T_0 + R + (N+1)r_1$$

$$> T_a + R \text{ for } R < Nr_1 - r_2$$

For $R \geq Nr_1 - r_2$,

$$T_N = T_0 + R + r_1 + Nr_2 > T_0 + R + r_1 + r_2 + R \text{ if and only if }$$

$$Nr_2 > R + r_2 , \text{ That is } R < (N-1)r_2 .$$

so for $(Nr_1 - r_2) < R < (N-1)r_2$, $T_N > T'_{N+1}$.

and for $R \geq (N-1)r_2$, $T'_{N+1} = T_0 + r_1 + R + r_2 + R \geq T_N$.

$$T'_{N+1} = T_0 + r_1 + R + Nr_2 + [R-(N-1)r_2] .$$
Thus, for all \( R < (N-1)r_2 \), \( T_N > T_{N+1} \) and

for all \( R \geq (N-1)r_2 \), \( T_{N+1} \geq T_N \). These two cases are illustrated in Figs. 2.5 and 2.6.

The transfer of output results will be initiated when both the events, that is, processing of \((N+1)st\) packet and arrival of the 0-bit, are completed. That is, if \( R < (N-1)r_2 \), then the output transfer is initiated after the time

\[
T_N = T_0 + R + r_1 + Nr_2, \text{ since } T_N > T_{N+1}.
\]

If \( R \geq (N-1)r_2 \), then the output transfer is initiated after the time

\[
T_{N+1} = T_0 + R + r_1 + Nr_2 + [R-(N-1)r_2].
\]

The two expressions can be combined together as

\[
t = T_0 + R + r_1 + Nr_2 + [R-(N-1)r_2], \text{ where}
\]

\[
R-(N-1)r_2 = 0 \text{ for } R < (N-1)r_2
\]

\[
= R-(N-1)r_2 \text{ for } R \geq (N-1)r_2.
\]

-- E 2.12

The output transfer of the results of \((N+1)st\) packet is completed \( r_2 \) cycles after its initiation. So the transfer is completed by the time, \( T_{N+1} \), where

\[
T_{N+1} = t + r_2 = T_0 + R + r_1 + (N+1)r_2 + [R-(N-1)r_2].
\]

That is, the \((N+1)st\) is completely processed and its results are transferred by the time

\[
T_{N+1} = T_0 + R + r_1 + (N+1)r_2 + [R-(N-1)r_2],
\]

*Induction hypothesis:* The \((N+i)th\) packet is transferred to the processor \( P_i \) by the time \( T_0 + (N+i)r_1 + [R-(N-1)r_1] \) and its processing and the result transfer is finished by the time
$v_2 = 3$
$v_1 = 2$
$N = 5$
$Y_2 < Y_1$
$R < (N-1)Y_2$

FIG. 2.5
\[ T_{N+i} = T_0 + (N+i)r_2 + R + [R^2(N-1)r_2] + r_1, \]
\[ i < N, \]

It is, then, necessary to show that the hypothesis holds for \((N+i+1)st\) packet.

It follows from Observation 2 that the next packet (after the \((N+i)th\) packet) goes to processor \(P_{i+1}\). That is the \((N+i+1)st\) packet is processed by \(P_{i+1}\). In order to find out when \(P_{i+1}\) finishes executing the \((N+i+1)st\) packet it is necessary to know

i) when \(P_{i+1}\) finishes processing the \((N+i+1)st\) packet \((T'_{N+i+1})\)

ii) when \(P_{i+1}\) gets the 0-bit and access to the output bus.

The latter is easily found from the hypothesis of the lemma. Processor \(P_i\) finishes the transfer of results by the time

\[ T_{N+i} = T_0 + R + r_1 + (N+i)r_2 + [R^2(N-1)r_2] \]

The processor \(P_{i+1}\) can transfer the results only where the processing of \((N+i+1)st\) packet is finished and when it has access to the output bus. So the processor will transfer the results after the larger of \(T'_{N+i+1}\) and \(T_{N+i+1}\), since that ensures that both the conditions are met. We need to evaluate \(T'_{N+i+1}\).

Since \(P_i\) finished execution by time \(T_{N+i}\), it must have received the \(i\)th packet at worst by the time \([T_{N+i} - R - r_2]\).

So the I-bit was shifted to \(P_{i+1}\) latest by time \((T_{N+i} - R - r_2)\), as shown in Fig. 2.7. Hence, \(P_{i+1}\) received the \((N+i+1)st\) packet latest by the time \(T_a = (T_{N+i} - R - r_2) + r_1\). \(P_{i+1}\) can start processing \((N+i+1)st\) packet after it has received the \((N+i+1)st\) packet and after it has finished the transfer of results of the previous packet, the
(i+1)st packet. From Lemma 2.1 the execution of (i+1)st packet is finished by the time
\[ T_{i+1} = T_0 + R + r_1 + (i+1)r_2. \]

**Case 1)** If \( R \geq (N-1)r_2 \), as shown in Fig. 2.7. Only processors \( P_{i+1} \)
are shown.

then \[ T_a = T_{N+i} - R - r_2 = T_0 + R + r_1 + (i+1)r_2 + R - r_2 + r_1 \]
\[ = T_0 + R + r_1 + i r_2 + r_1 \]
\[ < T_{i+1} = T_0 + R + r_1 + (i+1)r_2. \]

So \( P_{i+1} \) starts processing of \((N+i+1)st packet after \( T_{i+1} \). It will finish processing \( R \) cycles later by the time \( (T_{i+1} + R) \). So, \( T'_{N+i+1} = T_{i+1} + R \),
\[ = T_0 + R + r_1 + (i+1)r_2 + R. \]
\[ = T_0 + R + r_1 + (N+i)r_2 + [R - (N-1)r_2]. \]
\[ = T_{N+i}. \]

**Case 2)** \( R < (N-1)r_2 \). This case is illustrated in Fig. 2.8.
\[ T_a = T_{N+i} - R - r_2 + r_1 = T_0 + R + r_1 + (N+i)r_2 + r_1 - R - r_2 \]
\[ = T_0 + r_1 + r_1 + (N-1)r_2 + i r_2 \]

\[ T_a = T_0 + 2r_1 + i r_2 + (N-1)r_2. \] \( T_{i+1} = T_0 + R + r_1 + r_2. \)
if \( (N-1)r_2 + r_1 \geq R + r_2 \)
\[ \text{or } (N-2)r_2 + r_1 \geq R. \]
if \( (N-2)r_2 + r_1 < R < (N-1)r_2 \) then \[ T_a < T_{i+1}. \]

We have already seen that if \( T_{i+1} > T_a \), then \[ T'_{i+N+1} = T_{i+1} + R = T_{N+i}. \]
So consider \( R \leq (N-2)r_2 + r_1 \). Since \( T_a \geq T_{i+1} \), the processing of the \((N+i+1)st\) packet will start at the latest (it may start earlier) by the time \( T_a \). Then, \( P_{i+1} \) finishes processing by the time \((T_a + R)\). Thus,

\[
T'_{N+i+1} = T_a + R
= T_0 + (N+i-1)r_2 + 2r_1 + R,
= T_0 + r_1 + R + (N+i)r_2 + r_1 - r_2.
\]

Then, for all values of \( R \), \( T_{N+i} \geq T'_{N+i+1} \).

So, the processing of the \((N+i+1)st\) packet is completed before or at the same time as the time at which \( P_{i+1} \) gets access to the output bus.

Thus, \( P_{i+1} \) transfers results in the next \( r_2 \) cycles. Therefore the execution of the \((N+i+1)st\) packet is finished by the time \( T_{N+i+1} \), which is

\[
T_{N+i+1} = T_{N+i} + r_2
= T_0 + R + r_1 + (N+i)r_2 + [R:(N-1)r_2] + r_2
= T_0 + R + r_1 + (N+i+1)r_2 + [R:(N-1)r_2]
= T_{i+1} + Nr_2 + [R:(N-1)r_2].
\]

b) Part a) shows that for the second set of \( N \) packets the execution time for any processor, \( P_{N+i} \), is given by

\[
T_{N+i} = T_i + Nr_2 + [R:(N-1)r_2] \quad \text{for} \quad r_1 < r_2
\]

\[1 \leq i \leq N \]

The time taken by any processor for executing a packet belonging to the second set is \([Nr_2 + R : (N-1)r_2]\), which is a constant independent of \( i \). Therefore, the packets belonging to the second set finish in the
same staggered sequence with \( r_2 \) cycles between their finishing times, as the first \( N \) packets. Since this was the only initial condition used in proving part a), the initial conditions for the third set of \( N \) packets is again the same as that for the second set. It follows then that the same constant time, \( N r_2 + [R \cdot (N-1) r_2] \) will be added for execution of a packet belonging to the third set. Thus if an arbitrary packet of the third set, say \( P_{2N+i} \), is considered then,

\[
T_{2N+i} = T_{N+i} + Nr_2 + [R \cdot (N-1) r_2] .
\]

\[
= T_i + 2[Nr_2 + R \cdot (N-1) r_2] , \quad \text{for} \quad 1 \leq i \leq N .
\]

The reasoning can be extended for the \( Kth \) set of \( N \) packets. Then the execution time for a packet, say \([(K-1)N + i]th \) packet, of the \( Kth \) set will be given by

\[
T_{(K-1)N+i} = T_{(K-2)N+i} + Nr_2 + [R \cdot (N-1) r_2]
\]

\[
= T_{(K-3)N+i} + 2[Nr_2 + (R \cdot (N-1) r_2)]
\]

\[
= \cdots
\]

\[
= T_i + (K-1)[N r_2 + (R \cdot (N-1) r_2)] .
\]

-- PROVED

**Theorem 2.1** For a given vector operation of length \( Z \), the total execution time is given by \( T_Z \), where

\[
T_Z = T_0 + R + Z \max(r_1, r_2) + \min(r_1, r_2)
\]

\[
+ \left( \left\lceil \frac{Z}{N} \right\rceil - 1 \right) [R \cdot (N-1) \max(r_1, r_2)] .
\]

-- E 2.13

**Proof:** Since the \( Zth \) packet is sent to the processors last it is processed last and its results are transferred last. So the
vector operation is completely executed when the execution of
the Zth packet is finished.

Case 1) \( r_1 \geq r_2 \). Let \( Z = kN+i \) where \( k, i \) are integers \( \geq 0 \).

From Lemma 2.2, the Zth packet is completely executed by the
time \( T_Z = T_{kN+i} \), where

\[
T_{kN+i} = T_i + k[Nr_1 + [R^{\infty}(N-1)r_1]]
\]

\[
= T_0 + R + ir_1 + r_2 + k[Nr_1 + k[R^{\infty}(N-1)r_1]]
\]

\[
= T_0 + R + (kN+i)r_1 + r_2 + k[R^{\infty}(N-1)r_1]
\]

Since \( Z = kN+i \), \( k = \left(\left\lceil \frac{Z}{N}\right\rceil - 1\right) \) where \( \left\lceil \frac{Z}{N}\right\rceil \) represents the ceiling of \( \frac{Z}{N} \) =

nearest larger integer.

Then, \( T_{kN+i} = T_Z = T_0 + R + Zr_1 + r_2 + \left(\left\lceil \frac{Z}{N}\right\rceil - 1\right)[R^{\infty}(N-1)\max(r_1, r_2)] \)

Case 2) \( r_2 > r_1 \). From Lemma 2.3,

\[
T_Z = T_{kN+i} = T_i + k[Nr_2 + (R^{\infty}(N-1)r_2)]
\]

\[
= T_0 + R + r_1 + ir_2 + k[Nr_2 + k[R^{\infty}(N-1)r_2]]
\]

\[
= T_0 + R + (kN+i)r_2 + r_1 + k[R^{\infty}(N-1)r_2]
\]

Since \( Z = kN+i \) and \( k = \left(\left\lceil \frac{Z}{N}\right\rceil - 1\right) \),

\[
T_Z = T_0 + R + Zr_2 + r_1 + \left(\left\lceil \frac{Z}{N}\right\rceil - 1\right)[R^{\infty}(N-1)r_2] \cdot
\]

For \( r_2 > r_1 \), \( \max(r_1, r_2) = r_2 \) and

\( \min(r_1, r_2) = r_1 \). Using
these expressions,

\[ T_z = T_0 + R + Z \max(r_1, r_2) + \min(r_1, r_2) \]

\[ + \left( \left\lfloor \frac{Z}{N} \right\rfloor - 1 \right) \left[ R \cdot (N-1) \max(r_1, r_2) \right] \]

Thus in both cases

\[ T_z = T_0 + R + Z \max(r_1, r_2) + \min(r_1, r_2) + \left( \left\lfloor \frac{Z}{N} \right\rfloor - 1 \right) \left[ R \cdot (N-1) \max(r_1, r_2) \right] \]

\textit{Proved.}
2-3 Parallelism in Stream Processors

If a multiprocessor system consists of N processors, it is natural to expect that the system performance should be N times that of a single processor. However, such an improvement in performance is generally limited by other factors such as vector fitting, start-up time in case of a pipeline or a stream processor [FLYNN 72], delays in transfer of data, and other such factors. It is important to know how these limitations affect the performance of a multiprocessor system with respect to a single processor system. A measure of this parallelism is the effective number of processors.

The effective number of processors in a multiprocessor system is defined as $N_{eff}$, where $N_{eff}$ is the

$$\text{ratio} = \frac{\text{Thruput of the multiprocessor system for a given set of operations}}{\text{Thruput of a single processor system for the same set of operations}}.$$

The maximum value of $N_{eff}$ is N. We will also refer to this ratio as the effective parallelism of the multiprocessor system.

In terms of the total time for Z operations, the effective number of processors is given by

$$N_{eff} = \frac{\text{Total time for Z operations for a single processor}}{\text{Total time for Z operations for the multiprocessor system}}$$

$$= \frac{T_{Z,1}}{T_{Z,N}}$$, where $T_{Z,N}$ denotes the time for Z operations for an N-processor system.
We define the ratio $N_{\text{eff}}/N$ as the 'Linearity' of the system. Thus, a Linearity of one implies $N_{\text{eff}} = N$. That is, the performance of the system increases linearly as the number of processors. In this section we determine $N_{\text{eff}}$, for a given set of operations, for a stream processor consisting of $N$ processors. We also determine the conditions for ensuring the effective number of processors $\geq kN$, or Linearity $\geq k$ where $0 < k \leq 1$. Clearly, it is desirable that $k$ be as close to one as possible since that implies that system achieves close to the maximum possible performance.

Consider a system with only one processor. There is no overlap of processing because only one task can be executed at one time. The processor receives the first input in $r_1$ cycles. It processes it for $R$ cycles. At this time, $(r_1 + R)$, it can output the result in $r_2$ cycles and receive in parallel the second input in cycles. It can start processing the second packet after both input and output are finished, that is, at time $(r_1 + R) + \max(r_1, r_2)$ as shown in Fig. 2.9. $R$ cycles later it can again receive a new packet and output the results in $\max(r_1, r_2)$ cycles. Clearly, this repeats for every subsequent packet. Thus every packet after the first is processed in $[\max(r_1, r_2) + R]$ cycles. After the last packet is processed its results are output in $r_2$ cycles. So the total time for $Z$ operations is

\[
T_{Z,1} = r_2 + R + (Z-1) \left[ \max(r_1, r_2) + R \right] + r_1
= r_2 + r_1 + ZR + (Z-1) \max(r_1, r_2)
= ZR + Z \max(r_1, r_2) + \min(r_1, r_2)
\]

--- E 2.14

since \((r_1 + r_2) = \max (r_1, r_2) + \min (r_1, r_2)\).
Now, consider a Stream Processor with \( N \) processors. As shown earlier in section 1-5, the system performs differently in the ranges \( R < (N-1) \max(r_1, r_2) \) and \( R \geq (N-1) \max(r_1, r_2) \). Therefore, these two cases are considered separately.

Case 1) \( R \leq (N-1) \max(r_1, r_2) \), that is \( \frac{R + \max(r_1, r_2)}{\max(r_1, r_2)} \leq N \).

From Eq. 2.13, the processing time for \( Z \) operations is

\[
T_{Z,N} = Z \max(r_1, r_2) + R + \min(r_1, r_2)
\]

Then,

\[
N_{\text{eff}} = \frac{T_{Z,1}}{T_{Z,N}} = \frac{Z \max(r_1, r_2) + ZR + \min(r_1, r_2)}{Z \max(r_1, r_2) + R + \min(r_1, r_2)}
\]

\[
= \frac{\max(r_1, r_2) + R + \frac{\min(r_1, r_2)}{Z}}{\max(r_1, r_2) + \frac{R + \min(r_1, r_2)}{Z}}
\]

Therefore, \( N_{\text{eff}} \leq \frac{R + \max(r_1, r_2)}{\max(r_1, r_2)} \) for all \( Z \).

The maximum value of \( N_{\text{eff}} \) is \( \frac{R + \max(r_1, r_2)}{\max(r_1, r_2)} \) which is less than or equal to \( N \). Therefore in this case however large the number of processors, the maximum value of the effective number of processors remains

\[
N_{\text{ub}} = \frac{R + \max(r_1, r_2)}{\max(r_1, r_2)}, \text{ implying that adding any more processors}
\]

\[
\frac{R + \max(r_1, r_2)}{\max(r_1, r_2)} \text{ will not improve the performance of the system.}
\]

It was explained earlier that this upper bound exists because the busses cannot transfer data any faster. Since the upper bound on \( N_{\text{eff}} \) depends on the ratio \( R/\max(r_1, r_2) \), the system can achieve better performance when \( R \) is much greater than \( \max(r_1, r_2) \). This allows the
$Z \propto \frac{k}{1-k}$ vs $k$

**FIG. 2.10**
system to effectively use more processors, hence achieve better performance.

In order to achieve an effective parallelism of kN, it is necessary that \( N_{eff} \geq kN \).

That is,

\[
\frac{R + \max(r_1, r_2) + \frac{\min(r_1, r_2)}{Z}}{\max(r_1, r_2) + \frac{R + \min(r_1, r_2)}{Z}} \geq kN
\]

That is,

\[
N_{ub} + \frac{\min(r_1, r_2)}{Z \max(r_1, r_2)} \geq kN + kN \cdot \frac{R + \min(r_1, r_2)}{Z \max(r_1, r_2)}
\]

This can be rewritten to give

\[
Z \geq \frac{kN}{(N_{ub} - kN)} \cdot \frac{R + \min(r_1, r_2) - \frac{\min(r_1, r_2)}{kN}}{\max(r_1, r_2)}
\]

Now, \( R + \min(r_1, r_2) > > \frac{\min(r_1, r_2)}{kN} \) if \( k > \frac{1}{N} \).

Then,

\[
Z \geq \frac{k/k_{ub}}{(1-k/k_{ub})} \cdot \frac{R + \min(r_1, r_2)}{\max(r_1, r_2)} \quad -- \text{E 2.15}
\]

where, \( k_{ub} = \frac{N_{ub}}{N} \) and \( k > \frac{1}{N} \).

The variation of \( \frac{k/k_{ub}}{(1-k/k_{ub})} \) is shown in Figure 2-10 for various values of \( k/k_{ub} \). It can be seen that in order to achieve \( k/k_{ub} \) close to one a very large value of \( \frac{k/k_{ub}}{(1-k/k_{ub})} \) hence of \( Z \), is required. This illustrates that the upper bound of \( N_{ub} \) for effective number of processors
is achieved for very large vectors only. However, E 2.15 gives us a way of determining the required vector sizes and other parameters in order to achieve a certain improvement.

Case 2)

\[ R \geq (N-1) \max(r_1, r_2) \]

From E 2.13, the time for processing Z operations is given by

\[ T_{Z,N} = Z \max(r_1, r_2) + \left( \left\lfloor \frac{Z}{N} \right\rfloor - 1 \right) (R - (N-1) \max(r_1, r_2)) + R + \min(r_1, r_2) \]

Since \( \left\lfloor \frac{Z}{N} \right\rfloor \geq \frac{Z}{N} \),

\[ T_{Z,N} \geq Z \max(r_1, r_2) + \frac{Z}{N} \left[ R - (N-1) \max(r_1, r_2) \right] + R + \min(r_1, r_2) - \left[ R - (N-1) \max(r_1, r_2) \right] \]

Then,

\[ N_{\text{eff}} = \frac{T_{Z,1}}{T_{Z,N}} \leq \frac{Z \max(r_1, r_2) + ZR + \min(r_1, r_2)}{\frac{Z}{N} \left[ R + \max(r_1, r_2) \right] + (N-1) \max(r_1, r_2) + \min(r_1, r_2)} \]

-- E 2.16

\[ \leq \frac{\min(r_1, r_2)}{N + \frac{Z(R + \max(r_1, r_2))}{1 + \frac{(N-1) \max(r_1, r_2) + \min(r_1, r_2)}{Z(R + \max(r_1, r_2))}}) \]

Since \( (N-1) \max(r_1, r_2) + \min(r_1, r_2) \geq \min(r_1, r_2) \), it follows that \( N_{\text{eff}} \leq N \). Thus the upper bound on the parallelism for this case is \( N \).

This is achieved where \( Z = \infty \), that is \( Z \) is very large and/or \( R \) is
very large compared to \((N-1) \max(r_1, r_2)\). Thus the improvement in performance achieved over a single processor is \(O(N)\) for very large vectors. This is the performance achieved by other more general multiprocessor configurations as was shown in section 1-6. Also, this confirms the comparison presented in section 1-6 to illustrate the fact that the stream processor has inherently the same order of parallelism as other multiprocessor or equivalent pipeline configurations.

In actual operations a parallelism of \(N\) requires extremely large vectors. In order to achieve a parallelism of order \(kN\), it is necessary that \(N_{\text{eff}} \geq kN\), where \(0 < k \leq 1\). From E 2.16, it follows that

\[
\frac{Z \max(r_1, r_2) + Z R + \min(r_1, r_2)}{\frac{Z}{N} (R + \max(r_1, r_2)) + (N-1) \max(r_1, r_2) + \min(r_1, r_2)} \geq kN
\]

then \(N_{\text{eff}} \geq kN\).

That is,

\[
Z(\max(r_1, r_2) + R) + \min(r_1, r_2) \geq kN \frac{Z}{N} (R + \max(r_1, r_2)) + kN (N-1) \max(r_1, r_2) + \min(r_1, r_2)
\]

That is,

\[
Z \geq \frac{k}{1-k} \cdot \frac{N}{R + \max(r_1, r_2)} [ (N-1) \max(r_1, r_2) + \min(r_1, r_2) ]
\]

That is,

\[
Z \geq \frac{k}{1-k} \cdot \frac{N}{R + \max(r_1, r_2)} [ (N-1) \max(r_1, r_2) + \min(r_1, r_2) ]
\]

\[\text{-- E 2.17}\]
This gives us an expression relating vector length, processing time, data transfer times and the number of processors to the desired value of \( k \). The variation of required vector length with respect to \( k \) is the same as for Case 1). The only difference is that \( k \) can achieve a maximum value of 1 in this case. Again, Fig. 2.10 illustrates that it is necessary to have very large vectors and/or \( R \) in order to achieve parallelism close to \( N \).
CHAPTER III
MEMORY SYSTEMS

3-1 Introduction

It was assumed in Chapters 1 and 2 that the memory system supplies operands without delay whenever a processor requests them. Also, the memory system accepts data whenever it is presented with results from a processor. It is necessary to examine whether these assumptions are reasonable and the desired behaviour of the memory can indeed be achieved. It is also of importance to examine various techniques for implementing the required memory scheme in order to achieve a low-cost and modular memory system. Modularity is emphasised here in order to i) achieve expandability in a given system, ii) make it easier, conceptually, for a user to design a system that meets his requirements, and iii) simplify the physical assembly of a memory system.

Data is transferred from the memory to the processors and from the processors to the memory on two single word data busses for the two directions of data transfer. The maximum transfer rate is given by the bandwidth of the data path, which is one word per cycle for input and one word per cycle for output. The memory system should then have a bandwidth greater than or equal to the bus bandwidth in order to maintain the maximum rate of data transfer to the processors.

The memory system comprises two main parts, i) a set of memory modules for storage and access of data, and ii) an interconnection network for collection of the data into packets and transfer of the packets to the processors.
In this chapter only the first part, that is the access and storage of data, is discussed. The interconnection network is presented in Chapter IV. For the purpose of this chapter it is assumed that there exists an interconnection scheme which can transfer data from the memory modules (actually from the buffers, as will be seen later) to the processors at the maximum rate and transfer results from the processors to the memory modules at the maximum rate.

In section 3-2 general techniques for improving the bandwidth of memory systems are discussed. In particular a memory scheme suited to vector operations is described. The scheme is called the 'Simultaneous Access Scheme' (SAS) and it represents the approach taken by existing vector computers in order to achieve high bandwidth. In order to overcome some of the disadvantages of the Simultaneous Access Scheme, a modified version is presented in section 3-3. It is shown that this scheme is modular in nature and is more efficient than SAS. This scheme is called the 'Modified Simultaneous Access Scheme' (MSAS). This scheme requires a distinct buffer for each variable in all of the memory modules. In section 3.4 another scheme called the Independent Access Scheme (IAS) is presented. It is shown that IAS has the same output behaviour as MSAS but requires only one distinct buffer for all the input variables and one distinct buffer for all the output variables, or every module. However, IAS requires a more complex algorithm to generate and load the address sequences in the modules. The buffer requirements for MSAS and IAS are discussed in section 3-5. It is shown that the optimal buffer space for both the schemes is the same and is equal to one word per variable per module.
The memory schemes are studied and compared from the point of view of modularity, complexity of address generation, and buffer requirements.

3.2 Parallel Memories

It was shown in the earlier chapters that the processing speed of a stream processor can be increased by adding more processors. However, the final limitation to an increase in processing speed is the bandwidth of the data bus and the bandwidth of the memory system. It is then necessary to consider techniques for achieving high memory speeds. In the past this has partly been achieved by new technologies such as thin film memories and semiconductor memories. Additional speed up has been obtained through the use of parallelism at the word level. This is accomplished by

i) having large word size; computers like the CDC 6600 use 64 bit words thus accessing 64 bits at a time from the memory.

ii) fetching two or more consecutive words simultaneously; This is called 'Superword Access'. Most of the current large computer systems such as the TI-ASC [ASC 73] and CDC/STAR-100 [CDC 73] use superword access. Specifically, in the case of the TI-ASC, eight consecutive words of length 32 bits each are fetched simultaneously from a memory module. However, this scheme is very inefficient when the words to be accessed are not in consecutive locations, since only one word out of
every eight may be utilized, leading to no effective gain in bandwidth.

A more general approach to increasing the bandwidth of a memory system is to use a number of memory units in parallel. This technique is called memory interleaving. In the late 1950's, ILLIAC II and IBM STRETCH introduced the first two-way interleaved memories [KUCK 75]. It is clear that if a word can be fetched from each of the M memory units at once, then the effective memory bandwidth is increased by a factor of M. Most of current large computing systems like the CDC STAR-100, TI-ASC, etc., use a combination of interleaved memories with superword access to increase the effective bandwidth of their memory systems. Interleaved memories are particularly suited for storage and access of vectors.

Consider a vector \( A = a_1, a_2, \ldots, a_z \). The vector elements are stored across the M memory modules as shown in Fig. 3.1. Let the storage start from the ith module, then \( a_1 \) is stored at the location \( L \) in module \( j \). \( a_2 \) is stored at location \( L \) in module \( j+1 \), \( a_{M-j+1} \) is stored at \( L \) in module \( M \). The next M elements are stored at address \( L+1 \) in the M modules, and so on. Note that the address is incremented when the next element is to be stored in module 1 and is held constant till module M. This simplifies greatly the address generation in order to store the elements of a vector since only one address need be sent to all the modules for storing M consecutive elements in parallel. The same holds true for the access of M elements such that the first element of those M elements is in module 1.
INTERLEAVED MEMORY STORAGE SCHEME

FIG. 3.1

- Addresses: M_1, M_j, M_{j+1}, M_m
- Addresses: L, L+1, L+2
- Elements: a_1, a_2, a_{m+1}, a_{m+2}, a_{m-j+1}, a_{m-j+2}, a_{2m-j+1}, a_{2m-j+2}, a_{2m-j+3}
For convenience of notation, a sequence of $M$ consecutive data elements starting with element $a_1$ will be referred to as a 'row' starting at $a_1$. In the following section, the 1st row refers to the first $M$ elements of a vector, for example, $a_1, a_2, \ldots, a_M$. In general the '$j$th row' refers to the set of elements $a_{(j-1)M+1}, a_{(j-1)M+2}, \ldots, a_{(j-1)M+M}$. Consider a vector operation $C \cdot A \times B$, where $A$, $B$, $C$ are vectors of length $Z$. The output of the memory system should be the stream of packets

$$<a_1, b_1>, <a_2, b_2>, \ldots, <a_Z, b_Z>.$$ 

The result packets sent back from the processors to the memory are $C_1$, $C_2$, $\ldots$, $C_Z$. The requirements of an efficient memory system are then,

1) to be able to produce the stream packets, as shown above, at the maximum rate. In order to do this it has to be able to, first, access the vectors at a very high rate and, second, put together the packets in correct order at a very high rate.

2) to be able to receive the stream of result packets at the maximum rate. This requires splitting of result packets into proper vector segments and storing the vector segments efficiently.

A common approach to handling these two steps is described below. It was shown in the earlier discussions that it is efficient to access a complete row at a time. In order to generate the packets one row of each vector can be accessed in a sequence, the sequence being called the
address sequence. Thus, a row of A is accessed first; next, a row of B is accessed; next, a row of C is stored and this sequence is repeated. This allows data to be fetched and stored in the memory at the maximum bandwidth. It also allows the data elements of all the vectors to be fetched in an interlaced fashion thus facilitating the formation of packets without too long a delay. For instance, all the elements of a given packet are obtained in a maximum of three (in general, V cycles for V variables) consecutive memory cycles.

Since M data elements of each variable are fetched simultaneously, it is necessary to hold them in buffers while they are grouped into packets and sent to the processors. A separate buffer is used for each variable in order to incorporate a simple scheme for aligning the data. The data elements are thus fetched from the memory and held in buffers where they are collected into packets and sent to the processors. Similarly, result packets are sent to the buffers from the processors. The result packets are split into rows of corresponding vectors and finally stored in the memory.

In order to hold a row of each variable it is necessary that the buffer space be at least M words for each variable. Once M words of each variable are available in the buffers which are FIFO queues, the first element from each queue is sent to the processors, then the next element from each queue, and so on. In order to ensure a continuous transfer of packets of data elements, the next row of each variable is fetched while the elements of previous row are sent to the processors. This requires that the buffer space, at worst, be 2M words to store two
rows of a variable. The maximum total buffer space is $2V_m M$ words where $V_m$ is the maximum number of variables in the possible set of array operations.

An implementation of such a scheme is illustrated in Fig. 3.2. The scheme will be called the Simultaneous Access Scheme because all the memory modules operate in lockstep and always access simultaneously. The same address is sent by the memory control unit to all the modules. This simplifies address generation and sequencing since it is all handled in the memory control unit. It should be noted that the data elements are sent sequentially from memory modules to the buffer. It takes $M$ data cycles to transfer the $M$ words to the buffer for each row of elements. Also, another row cannot be fetched from the memory until all the words of the current row have been removed from the memory modules. This increases the total time to obtain $M$ words in the buffers, the time being a sum of the memory cycle time and the time to transfer $M$ words. This is indeed the main disadvantage of this scheme.

The other disadvantages are that the scheme is not modular and that the buffer space has to be changed whenever the number of modules is changed.
3.3 Modified Simultaneous Access Scheme

In order to overcome the disadvantages of the simultaneous access scheme, a modified scheme is presented here. The modifications are:

i) A buffer space for the elements of all the variables is provided locally in every memory unit (instead of a common external buffer). Data is transferred directly from a memory unit to the processors and from the processors to a memory unit.

ii) An interconnection network is provided such that an element can be transferred from any memory module's buffer onto the data bus. This allows the elements of a packet to be transferred in consecutive cycles even though the elements may reside in different modules (this essentially solves the problem of data alignment).

iii) Data access is initiated independently in the modules. It is not necessary that data be fetched simultaneously from the modules. A fetch for an element of a given variable is initiated in a module if there are empty locations in the buffer for that variable. Similarly, a store is initiated if a data element is available in the buffer, else the store cycle for that variable is skipped. Fig. 3.4 illustrates the overall scheme. Data elements are read from the memory modules into their local buffers. This requires only one clock cycle as opposed to \( M \) cycles in the case of the previous scheme. The interconnection network selects data elements from the buffers and transfers them to the processors (and vice versa). It takes only one cycle to transfer any data element onto the one-word bus. This delay is the same as that of
the previous scheme. The net result is that the total delay in transfer of a data element is M clock cycles less than the delay encountered in the previous scheme.

The second main advantage of the modified scheme is that the buffer space is modularised. It was shown earlier that the total buffer space required for each variable is proportional to M, say = kM. When this space is divided M ways, only a fixed size of k words is required in every module. If each module has the required fixed amount of buffer space for each variable then whatever the number of modules in the system, the total buffer space will always be proportional to M, as desired. Memory modules can be added or removed from the memory system without requiring any changes in the buffers. The assumption made here is that splitting the buffer space has no effect on the output behaviour of the memory system. The memory still supplies the processors with data packets at the maximum rate. It is shown in section 3.5 that this is indeed true. It is also shown that the third modification allows a smaller buffer space per variable. Specifically, a space of only one word per variable is required in every module. This amounts to a total of M words of buffer for each variable.

Description:

Address Sequence: The variables are addressed in the same sequence in all the modules. This sequence is the sequence defined by the precedence relation of the given vector operation. That is, the same address sequence as that of the previous scheme is used. If the input variables, in order of precedence, are $X_1, X_2, \ldots, X_p$ and the output variables are
$X_{p+1}, X_{p+2}, \ldots, X_{p+q}$ then the basic address sequence is $S_{\text{basic}} = X_1, X_2, X_3, \ldots, X_p, X_{p+1}, \ldots, X_{p+q}$. The variables are addressed in this sequence. The complete sequence is obtained by repeating the basic sequence. Since the modules operate independently, it is necessary to store the basic sequence in each module. The memory modules, therefore, have at least $V = p+q$ address registers.

**Accessing Algorithm**

1. Given a vector operation, the basic address sequence is computed, as mentioned earlier, in a control unit.

2. From the starting address of a vector, the offset address of the first element of the vector in every module is computed. This is done for all the vectors.

3. For each module, the offset addresses of the first elements of all the variables in the module are put together in the order specified by the address sequence. The list is then loaded into the address registers of the module. A bit ($S$-bit) in each address register indicates whether the address register represents an input or output variable.

4a. Each module initiates an 'access' (fetch or store) to the location specified by the contents of the first address register. If the access is successfully completed then the contents of the address register are incremented by one. After the first access the location specified by the second address register is accessed, and so on. After access of the location specified by $(p+q)$th register, the access sequence
starts back at the first register and repeats until the end of the vector operation.

4b. Whether the 'access' is fetch or store is determined by the S-bit in the address register under consideration. If a fetch is indicated then it is successfully completed only if there is an empty location in the buffer space for the variable (which is being accessed). If the buffer is full then the fetch is not initiated. The contents of the address register are left unchanged and the control proceeds to the next address register in sequence. Similarly, if a store is indicated then it is successfully completed only when there is at least one data element available in the buffer. If the buffer is empty then the store cycle is aborted. The control proceeds to the next address register.

Buffering

The memory system thus interfaces with the buffers only. It was shown in Fig. 3.4 that a separate buffer is provided for each variable in every module. This is necessary in order to avoid any conflicts in the buffers. Consider the example of section 3.2. In the first memory cycle the first row of A is fetched. The buffers then contain the data elements as shown below,

\[ a_1 \quad a_2 \quad a_3 \quad \ldots \quad a_M \]

Next the first row of B is fetched. If B starts in module 2 and if the data elements are stored in the same buffer (the one used for elements
of A), then the contents of the buffers are

\[
\begin{array}{cccc}
\text{Buffer 1} & B_2 & B_3 & B_M \\
\hline
b_M & b_1 & b_2 & b_{M-1} \\
a_1 & a_2 & a_3 & a_M
\end{array}
\]

queue

Consider the transfer of the first packet \( a_1, b_1 \). The interconnection network can easily pull \( a_1 \) from the top of the buffer and send it to the processors. However, \( b_1 \) cannot be transferred because it is behind \( a_2 \) in the queue. Similarly, \( b_2 \) is behind \( a_3 \), \( b_3 \) is behind \( a_4 \), and so on. In order to avoid such conflicts, a separate buffer is provided for each variable in every module. For the example considered above, there are 3 buffers per module, one for each variable. The buffers can have, at minimum, one word space. The contents of the modified buffer after two memory cycles will be as shown below.

\[
\begin{array}{cccc}
\hline
a_1 & b_M & a_2 & b_1 \\
\hline
a_3 & b_2 & & \\
\hline
a_M & b_{M-1} & \\
\end{array}
\]

Clearly, \( a_1 \) and \( b_1 \) can be accessed without any conflicts and sent to the processors. Similarly, the remaining packets can be obtained without any conflicts. In order to provide a buffer space for each variable, at least one word must be made available per variable in every module.

This implies a minimum total of \( M \) words per variable for the memory system.
The main disadvantages of this scheme are

a) A distinct buffer is required for each variable in every module. This requires the implementation of \( V \) FIFO queues in every module if the maximum number of variables used is \( v \);

b) A switch (and the control to operate it) is required to route the data words from the memory to the correct buffer and from a particular buffer to the memory.

c) More hardware is necessary to provide \( v \) address registers in every module; the circuitry required to sequel through them and to increment them whenever an access is successfully completed.
3.4 Independent Access Scheme (IAS)

In section 3.3 it was pointed out that MSAS requires a switched buffer in every module. The switched buffer was required in order to resolve the conflicts in the buffers. Another way to resolve those conflicts is to rearrange the address sequences of the modules in a manner such that the elements are fetched in the correct order in the buffers. In the case of the example considered in section 3.3 elements $a_1, b_1, b_2, \ldots, b_{M-1}$ will be fetched in the first cycle and then $b_M, a_2, a_3, \ldots, a_M$ will be fetched in the second cycle. The contents are then

$$
\begin{array}{cccccccc}
  & b_M & a_2 & a_3 & \cdots & \cdots & a_M \\
\hline
  a_1 & b_1 & b_2 & \cdots & b_{M-1}
\end{array}
$$

Clearly, the first packet $a_1, b_1$ as well as other packets can now be transferred on the bus without any conflicts. This is achieved by reordering the address sequence for every module in such a way that at the output, the variables are read from the memory in order of increasing indices. Thus, if there is a simple FIFO queue as a buffer, the variables are available at the front of the queue in the correct order.

The scheme is best described with the help of another example. Consider an interleaved memory consisting of six modules. Let the variables to be fetched be A, B, C and D. Let vector A start in module $M_2$, vector B start in module $M_4$, vector C start in module 1 and vector D start in module 4. The address sequences of the modules are assigned as follows:
module $M_1$ fetches the variables in the sequence repeat
   -- C, B, D, A, C, B, D, A, . . . .
module $M_2$ fetches in the sequence
   -- A, C, B, D, A, C, B, D, . . . .
module $M_3$ fetches in the sequence
   -- A, C, B, D, A, C, B, D, . . . .
module $M_4$ fetches in the sequence
   -- B, D, A, C, B, D, A, C, . . . .
module $M_5$ fetches in the sequence
   -- B, D, A, C, B, D, A, C, . . . .
and module $M_6$ fetches in the sequence
   -- B, D, A, C, B, D, A, C, . . . .

If these sequences are followed by the memory modules then the contents of their buffers will be as illustrated in Fig. 3.5. The figure shows the contents of the buffers after the first six memory accesses of each module, under the assumption that there is sufficient buffer space available and none of the elements have been sent to the processor. It can be easily observed that the indices of variable elements in the buffers are in an increasing order. Thus all the elements ahead of any given data element $X_i[j]$ in a buffer have smaller indices, hence belong to earlier data packets. When the $j$th packet is being sent, $X_i[j]$ will be available at the front of the buffer, since all the elements ahead of $X_i[j]$ would already have been sent to the processors. We have essentially found a software alternative to using the switch and buffers for resolving buffer conflicts. It is, then, necessary to i) show that
<table>
<thead>
<tr>
<th>M_1</th>
<th>M_2</th>
<th>M_3</th>
<th>M_4</th>
<th>M_5</th>
<th>M_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

ADDRESS SEQUENCES

MEMORY MODULES

BUFFER QUEUES

A starts in module 2
B starts in module 1
B and D start in module 4

FIG. 3.5
these address sequences can be constructed for an arbitrary number of variables and memory units, and ii) prove that these address sequences do indeed ensure that the data elements are accessed in an increasing order in every module. In order to accomplish the above aims we will first present some definitions then present an algorithm for constructing the necessary address sequence for each module. Finally, a proof of the fact that data elements are accessed in an increasing order in every module is presented.

**DEFINITION:** A breakpoint is defined as a module in which at least one variable starts.

A breakpoint will be indicated by the module number $b_j$ where $M_{b_j}$ is a breakpoint. There may be $k$ breakpoints for a vector operation where $k \leq v$,

$v = \text{number of variables involved in the vector operation.}$

$= \text{number of input variables (}=p) + \text{number of output variables (}=q)$. We name the breakpoints $b_1, b_2, \ldots, b_k$ in an order such that $b_1 < b_2 < b_3 \ldots < b_k$.

In case of the example considered earlier,

$\begin{align*}
  b_1 &= 1 \quad (C \text{ starts in } M_1), \\
  b_2 &= 2 \quad (A \text{ starts in } M_2), \\
  \text{and} \quad b_3 &= 4 \quad (B, D \text{ start in } M_4).
\end{align*}$

Note that in the case of the breakpoint $b_3$ there are two variables starting in the module.

Each breakpoint defines an *Interval* of modules following the break-
point, up to the next breakpoint.

**Definition:** An interval \( I_j \) is defined as the set
\[
I_j = \left\{ M_i \mid b_j \leq i < b_{j+1} \right\}
\]
The last interval is defined as
\[
I_k = \left\{ M_i \mid (b_k \leq i \leq M) \lor (i < b_1) \right\}.
\]
In case of the earlier example,
\[
I_1 = \{ M_1 \}, \quad I_2 = \{ M_2, M_3 \}, \quad I_3 = \{ M_4, M_5, M_6 \}.
\]

**Definition:** Let a relation \( R \) on the set of variables be defined such that for any two members, \( X_i \) and \( X_j \), of the set, \( X_i R X_j \) implies \( X_i \) is sent to the processors before \( X_j \). That is, \( X_i \) comes earlier in the sequence of variables in a data packet.

Such a relationship is imposed by the vector operation being performed and hence specified in the instructions for execution of the vector operation.

We name the variables \( X_1, X_2, \ldots, X_r \) in an order such that
\[
X_1R X_2, X_2R X_3, \ldots X_{r-1}R X_r.
\]
This order will be represented by the sequence \( S_a = X_1, X_2, \ldots X_v \).

**Definition:** Let \( X_{c_1}, X_{c_2}, \ldots, X_{c_j} \) be the vectors which start in breakpoint \( b_j \). A subsequence \( S_j \) is defined for every breakpoint \( b_j \) as the sequence formed by \( X_{c_1}, X_{c_2}, \ldots, X_{c_j} \) under the relation \( R \).

If \( X_{c_1} R X_{c_2}, X_{c_2} R X_{c_3}, \ldots, X_{c_{j-1}} R X_{c_j} \).
then \( S_i = X_{c_1}X_{c_2} \ldots X_{c_j} \).

Such a sequence exists because there is a unique ordering under \( R \) of all the variables and \( S_j \) is formed by taking simply the variables \( X_{c_1}, X_{c_2}, \ldots, X_{c_j} \) in the order they occur in sequence \( S_a \).

**DEFINITION:** The universal address sequence is defined as the sequence \( S_u = S_k, S_{k-1}, \ldots, S_2, S_1 \). That is, it is the concatenation of the subsequences \( S_i \) in the reverse order.

**Algorithm for obtaining the address sequence for an arbitrary module:**

(A 3.3)

Step a) Generate all the subsequences \( S_i \) as given in the definition.

Step b) The basic address sequence \( S_i \) for a breakpoint \( b_j \) is obtained by cyclically shifting the universal address sequence \( S_u \) until \( S_j \) is at the beginning of the new sequence. Then,

\[
S_i = S_j, S_{j-1}, \ldots, S_2, S_1, S_k, S_{k-1}, \ldots S_{j+1}.
\]

In order to obtain the complete sequence, the basic sequence is repeated as often as required (until the length of the sequence = \( Z \)). This is done for all modules. Thus the complete sequence is

\[
S_i' = S_i, S_i, S_i, \ldots.
\]

Step c) For all other modules in interval \( I_j \), the basic address sequence is the same as \( S_i \).

This completes the algorithm. If the address sequence is generated
for input variables according to the algorithm, then data elements of the variables will be fetched in an increasing order of indices in every module. A proof of this proposition is presented below.

**Lemma 3.1:** All modules fetch all variables in the same cyclic sequence except that the modules may start their fetches at different points of the basic sequence.

**Proof:** This lemma follows from steps b) and c) of the algorithm.

**Lemma 3.2:** Consider two variables $X_a$ and $X_b$ which are fetched in some consecutive memory cycles $C_1$ and $(C_1+1)$ from a module $M_r$.

Then,

a) Elements of $X_a$ and $X_b$ are always fetched in consecutive memory accesses in the module $M_r$.

b) Elements of $X_a$ and $X_b$ are fetched in consecutive memory cycles in all other modules, also.

c) Let $X_a[i]$ and $X_b[j]$ be the data elements fetched in memory cycles $C_1$ and $(C_1+1)$ from module $M_r$. Let $X_a[k]$ and $X_b[l]$ be the data elements fetched in some later two consecutive cycles, say $C_2$ and $(C_2+1)$, from module $M_s$, $S = (r+k-i)(mod M)$. Then, it is true that

$$(j-i) = (l-k) .$$

**Proof:** a) It was stated in step b) of algorithm A that a complete address sequence is obtained by repeating the basic sequence over and over again. Therefore, if $X_b$ follows $X_a$ in the basic sequence, it will do so in all repetitions.
of the basic sequence, hence in the complete sequence. Therefore, data elements of $X_a$ and $X_b$ will always be fetched consecutively in a module.

b) From Lemma 3.1 it is clear that all the modules fetch the variables in the same cyclic sequence. Thus, if $X_b$ follows $X_a$ in module $M_r$, then it will do so in all other modules, also.

c) It was stated that $X_a[i]$ and $X_b[j]$ were accessed in cycles $C_1$ and $(C_1+1)$ from module $M_r$. Since the vectors are stored across the modules, $X_a[i+1]$ and $X_b[j+1]$ will be available in module $M_{r+1}$, $X_a[i+2]$ and $X_b[j+1]$ will be available in module $M_{r+2}(\mod M)$. In general, $X_a[i+E]$ and $X_b[j+E]$ will be available in module $M_{(r+E)(\mod M)}$.

Consider $E = (k-i)$. Then $X_a[k]$ will be available in module $M_{r+k-i}(\mod M)$. Also, $X_b[j+k-i]$ will be available in module $M_{r+k-i}(\mod M)$.

That is $S = r + k - i \ (\mod M)$.

Now, from parts a) and b) it is clear that every time an element of $X_a$ is accessed, an element of $X_b$ is accessed from the same module in its next memory fetch. Thus, if $X_a[i+1]$ is fetched from module $M_{r+1}$, then $X_b[j+1]$ would be fetched in the next cycle (all other earlier elements of $B$ would have accessed in earlier cycles). Similarly, after $X_a[j+2]$ is accessed from module $M_{r+2}$, $X_b[j+2]$ would be accessed in the next cycle, and so on. Thus, the data elements accessed from $M_{r+k-i}(\mod M)$ would be $X_a[k]$ and $X_b[j+k-i]$ in consecutive cycles. Hence

$$1 = j + k - i, \ \text{i.e.,} \ j - i = 1 - k$$

-- PROVED
We can now state the theorem and prove it using the results stated in the lemmas.

**Theorem 3.1:** If \( X_a[i] \) is a data element fetched from a module \( M_r \) at some time \( T_1 \) and \( X_b[j] \) is a data element fetched from the module \( M_r \) at some arbitrary later time, \( T_2 \), then it is true that

\[
i \leq j.
\]

This implies that for any two elements fetched in a module, the data element fetched later always has a higher (or equal) index value than the former data element.

**Proof:** We will first prove the theorem for the case when \( X_b[j] \) is accessed in the next immediate fetch after the access of \( X_a[i] \). We will then use this result to show that the theorem holds for any arbitrary two accesses from the same module.

\( a. \) Let the variable \( X_a \) begin in breakpoint \( b_a \). Then module \( M_{b_a} \) contains the elements \( X_a[1], X_a[1+M], X_a[1+2M], \ldots \).

Let the address sequence for module \( M_{b_a} \) be

\[
X_a, X_b, X_1, X_2, \ldots, X_k, X_a, X_b, X_1, \ldots.
\]

Note that we are assuming here that \( X_b \) follows immediately after \( X_a \) in the sequence. We will consider two cases.

**Case 1)** \( X_b \) begins in module \( M_{b_a} \). Then the elements of \( X_b \) available in \( M_{b_a} \) are

\[
X_b[1], X_b[1+M], X_b[1+2M], \ldots.
\]
The first access, according to the address sequence, will fetch $X_a[1]$, and the next access will fetch $X_b[1]$. Let, in some later two consecutive cycles, $X_a[i]$ and $X_b[j]$ be fetched from module $M_a$. Then by Lemma 3.2,

$$j-i = (1-1) = 0$$

that is, $j = i$, or $j \geq i$ is true.

It should be noted here that some other variables besides $X_a$ and $X_b$ may also start in $M_{b_a}$. However, this does not affect the proof as long as $X_a$ and $X_b$ are in successive positions in the address sequence. The only change introduced will be that some $k$th access will fetch $X_a[1]$ and $(k+1)$st access will fetch $X_b[1]$. Also, by Lemma 3.2 b) the proof holds for all modules.

**Case 2)** $X_b$ does not begin in $M_{b_a}$. From the definition of universal address sequence it is clear that $X_b$ must start in breakpoint $b_{a-1}$. This is because $X_b$ follows $X_a$ in the address sequence but does not belong to the same subsequence. If the interval $I_{a-1}$ has $c_{a-1}$ modules, then the data elements of $X_b$ are organized as follows:

<table>
<thead>
<tr>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{b_{a-1}}$</td>
</tr>
<tr>
<td>$M_{b_{a-1}}+1$</td>
</tr>
<tr>
<td>$M_{b_{a-1}}+2$</td>
</tr>
<tr>
<td>$M_{b_{a-1}}+c_{a-1}-1$</td>
</tr>
<tr>
<td>$M_{b_a}$</td>
</tr>
</tbody>
</table>

- $X_b[1]$  
- $X_b[2]$  
- $X_b[3]$  
- $X_b[c_{a-1}]$  
- $X_b[c_{a-1}+1]$  
- $X_b[1+M]$  
- $X_b[2+M]$  
- $X_b[3+M]$  
- $X_b[c_{a-1}+1+M]$  
- $\vdots$
Thus the smallest element of $X_b$ in module $M_{ba}$ is $X_b[Ca-1 + 1]$. The data elements are fetched according to the address sequence. If in the kth access $X_a[1]$ is fetched then in the (k+1)st access $X_b[Ca-1 + 1]$ will be fetched.

It follows from Lemma 3.2 b) that the elements of $X_a$ and $X_b$ will always be fetched in consecutive accesses in every module. Let $X_a[i]$ and $X_b[j]$ be two elements fetched in consecutive accesses in some module $M_r$ then by Lemma 3.2 c) it is true that

\[(j-i) = (C_{a-1}+1-1) = C_{a-1}\]

Since $C_{a-1} \geq 1$ ,

\[(j-i) \geq 1 > 0\]

i.e., $j \geq i$

Hence, it is true that $j \geq i$.

b. Now we will consider the more general case where $X_b$ follows $X_a$ in the address sequence but not necessarily in the successive position. Consider an address sequence, for module $M_{ba}$,

$X_a, X_1, X_2, \ldots, X_k, X_b, X_{k+1}, \ldots, X_d, X_a, X_1, X_2, \ldots, X_k, X_b, X_{k+1}, \ldots$

..... where the basic sequence is $X_a, X_1, \ldots, X_d$.

There are k variables $X_1, X_2, X_3, \ldots, X_k$ between $X_a$ and $X_b$. For an arbitrary set of successive accesses, let the elements of the variables
in module \( M_b \)
\[
X_a[i], X_1[i_1], X_2[i_2], \ldots, X_k[i_k], X_b[j], \ldots
\]

We will show by induction on length 'k' that theorem holds for arbitrary value of \( k \geq 0 \). We will refer to the sequence \( X_1, X_2, \ldots, X_k \) as a partial sequence.

**Basis step:** The proposition that "\( i \preceq j \) for any two elements \( X_a[i] \) and \( X_b[j] \) if \( X_a[i] \) is accessed earlier", is true for \( k = 0 \) (shown in part a)).

**Induction hypothesis:** Let the proposition hold for a partial sequence of length \( k \). That is
\[
i \preceq i_1 \preceq i_2 \preceq i_3 \ldots \preceq i_k \preceq j . \quad -- 1
\]

To show that the proposition holds when length of the sequence = \( k+1 \).

**Proof:** Let the \((k+1)\)th variable in the partial sequence be \( X_{k+1} \). Consider the subsequence \( X_k, X_{k+1} \). Since these two variables are in consecutive positions in the address sequence, then the theorem holds for \( X_k, X_{k+1} \), by the basis step.

That is
\[
i_{k+1} \geq i_k . \quad -- 2
\]

Now consider the two variables \( X_{k+1} \) and \( X_b \). They are in successive positions in the address sequence, hence by the basis step,
\[
j \geq i_{k+1} . \quad -- 3
\]
For the inequalities 1, 2, and 3 it is clear that
\[ i_1 \leq i_2 \leq \ldots \leq i_k \leq i_{k+1} \leq j. \]
Hence, \( i \leq j \) for all cases. \( \text{PROVED--} \)

The previous discussion dealt only with input variables. Clearly, the input buffer cannot be used for output since the output data moves in the opposite direction to the movement of input data. Hence, an output buffer, a FIFO queue from the result bus to the module's MDR is provided for each module. The result packets arrive from the processors in an increasing order hence the elements are backed into the output buffer in order of increasing indices. By Theorem 3.1, the variables will be available in the buffer of a module in the sequence \( S_0 \) where \( S_0 \) is the sequence computed for that module using algorithm A 3.3. The combined address sequence for all the variables is given by \( S = S_1 \cup S_0 \). Thus \( S_1 \) and \( S_0 \) have to be computed independently and then concatenated to obtain the basic sequence which is loaded in the address registers.

Having seen that it is indeed possible to generate the address sequences as desired, we now give a description of the scheme. The scheme is illustrated in Fig. 3.6. It is essentially the same as MSAS except for the change in the buffer structure and the change required in the control of address registers (described later).

Accessing Algorithms:

1) Given a vector operation, the basic address sequence is computed using the algorithm described earlier.
ii) From the starting address of a vector, the offset address of the first element of the vector in every module is computed. This is done for all the vectors.

iii) For each module the offset addresses of the first elements of all the variables in the module are put together in a list in the order specified by the address sequence of that module.

iv) The list for a module is loaded into the address register of that module. A bit in the address register indicates whether the address register represents an input or output variable.

v) Each module initiates an access (fetch or store) to the location specified by the first address register. If the access is successfully completed then the contents of the address register are incremented by one. Next, step vi) is executed. If the access is unsuccessful, then step vii) is executed.

vi) The location specified by the next register is accessed. If the access is successful, then the contents of the address register are incremented by one. Then this step is repeated. Note that the register following the \( (p+q) \) th register is the first register.

If the access is unsuccessful, then the following step is executed,

vii) a) If the previous unsuccessful access was for an input variable, then all the remaining input variables are skipped and the next access is initiated (execution of step vi) ) for the output variable where an access was
aborted. If there is no such variable then the access is initiated at the first output variable, that is, at (p+1)st register.

b) If the previous unsuccessful access was for an output variable, then all the remaining output variables are skipped and the next access (step vi) is initiated for the input variable where an access was aborted. If there is no such variable, then the access is initiated at the first input variable, that is, at the first register.

viii) The steps vi) and vii) are executed until the end of vector operation is indicated by the memory control unit.

NOTE: A fetch is successfully completed only if the input buffer has an empty location, else the fetch is aborted. Similarly, a store is successfully completed only if there is at least one data element in the output buffer.

The step vii) is necessary to ensure that the elements in the buffers satisfy Theorem 3.1. Consider three input variables A, B, C. If after access of an element of A the input buffer is full, then the access of B is aborted. Therefore, access of C is skipped and next time the access of input variables is initiated at variable B. This ensures the elements received in the input buffer are in the order A, B, C, as required by the IAS scheme. The same reasoning applies to the access of the output variables.

It should be noted that the implementation of this scheme requires more addressing hardware than the MSAS scheme. In particular this scheme
requires two additional counters to keep track of where the last input variable and the last output variable access were aborted.

**Buffer requirements:** As shown earlier, only one input buffer and one output buffer is required in every module. The scheme can function with only one word in each buffer. Hence it has a total buffer space of only 2M words as against VM words as in MSAS. However, with this buffer size the packet generation and transfer rate is much below the desired maximum rate.

It is shown in section 3.5 that in order to be able to generate and transfer data packets at the maximum rate, at least p words are required for the input buffer and q words for the output buffer.
3-5 Buffer Requirements and Analysis

It was shown in section 3-3 that it is necessary to have buffers in the memory modules in order to allow data selection and data transfer at a very high rate. It was pointed out that total required buffer space may be as large as $2MV_m$ words where $M$ is the number of memory modules and $V_m$ is the maximum number of variables in the set of possible vector operations. From the point of view of cost, it is necessary to consider as small a buffer space as possible. In this section it is shown that the minimum buffer space required to maintain data transfer at the maximum rate is indeed less than $2MV_m$ words.

It is shown that for the Modified Simple Access Scheme the optimal size for total input buffer space is $p$ words per module and the optimal size for total output buffer space is $q$ words per module. It is further shown that the optimal output buffer size for the Independent Access Scheme is $q$ words per buffer and that $p$ words are sufficient per input buffer in order to maintain a maximum data transfer rate, where $p$ is the maximum number of input variables and $q$ is the maximum number of output variables. The total buffer space required in either scheme is $(p+q)M$ words $= V_mM$ words, that is only half the buffer space required by the Simultaneous Access Scheme.

The buffer sizes are optimal in the sense that these are the minimum buffer spaces, hence incorporating minimum cost, that allow a maximum data transfer rate for all vector operations and all distributions of the variables. A distribution refers to a particular assignment of vectors to the modules. The maximum data transfer rate is the maximum rate
at which the processing system can accept and send data packets. In order to allow the processing system to achieve the maximum throughput it is necessary that the memory system be capable of supplying and receiving data at a rate greater than or equal to that of the processing system.

The maximum data rate for the processing system is dictated by the delays in the control signals for data transfer and by the delay in the transfer of control from one processor to another. These delays are discussed in more detail in Chapter IV. It is assumed here that the minimum time required to transfer a word is 'w' clock cycles. Thus the total time required to transfer p words of an input packet is pw clock cycles. In order to include the delays due to shifting of the control bits in the processing system after every packet transfer, the total time of transfer of an input packet is taken to be r1 where r1 > pw. Similarly, the q words of output packet can be transferred in the minimum time of 'qw' cycles. The total time for the transfer of an output packet is r2, where r2 > qw. As shown in Chapter II, the data transfer rate is determined by the larger of r1 and r2. We will denote the larger value by r_m, r_m = max(r1, r2).

The memory system can access M words in parallel in a memory cycle, M_C cycles. That is one word per M_C/M cycles. The time taken to fetch p elements of a packet and store q elements is (p+q) x \( \frac{M_C}{M} \cdot \frac{V_M M_C}{M} \). Here the time for fetch of input packet and the time for storing the output packet are added because these two operations are not overlapped in the memory. In order to assure that the processing system operates
at its maximum throughput, it is necessary that the memory system have a higher or the same throughput as the processing unit. Therefore, the time taken by the memory to fetch and store a packet should be less than the time taken for the transfer of a packet from the processors. That is,

\[
\frac{V_m M_c}{M} < r_m
\]

or

\[
M > V_m M_c
\]

-- E 3.1

Another condition assumed here is that \( M_c > w \) where \( w \) is time taken to transfer one word. This because if \( M_c < w \) then the memory is faster than a data bus and we need use only one memory module to keep up with the bus. That is, there is no need for parallel memories.

In order to establish the optimal buffer sizes, it is first shown that it is necessary to have buffer sizes greater than or equal to \( p \) and \( q \) in order to be able to transfer data at the maximum rate. It is then shown that buffers of sizes \( p \) and \( q \), for input and output, respectively, are sufficient to ensure that data can be transferred at the maximum rate. It follows that \( p \) and \( q \) are the optimal buffer sizes.

A. Necessary Condition

Theorem 3.2: For the Modified Simultaneous Access Scheme it is necessary that the total input buffer space per module be greater than or equal to \( p \) words and the total output buffer space be greater than or equal to \( q \) words, in
order to be able to transfer data at the maximum rate.

**PROOF:** The Modified Simultaneous Access Scheme uses switched buffers.

There is a separate buffer space for each variable in every module.

Therefore, there are \( p \) distinct input buffers and \( q \) distinct output buffers in each module. Since the minimum size of a buffer is one word, each module contains at least one word of buffer space per variable. Thus the total input buffer space is minimum \( p \) words and the total output buffer space is minimum \( q \) words.

**PROVED**

**Theorem 3.3** a) For the Independent Access Scheme, it is necessary that the total input buffer space per module be greater than or equal to \( Y_I \) such that

\[
Y_I = \left\lfloor p - \frac{(p-2)w}{M_C} \right\rfloor \quad \text{for} \quad p > 2 + \frac{M_C}{w}
\]

and

\[
Y_I = p \quad \text{for} \quad p < 2 + \frac{M_C}{w}
\]

in order to be able to transfer data at the maximum rate.

b) It is necessary that the total output buffer space per module be greater than or equal to \( q \) words.

**PROOF:** The Independent Access Scheme employs only one input buffer and one output buffer per module. The minimum buffer space required from the point of view of implementation is one word for input buffer and one word for output buffer. However,
from considerations of data transfer the buffer space required
is much larger, as shown below.

a) The proof is by counter example. It is shown that there
exists at least one variable distribution for which a buffer
size less than $Y_I$ will cause the data to be transferred at
a rate less than the maximum rate.

Consider the variable distribution where all the variables start in
the same module. In that case all the elements of a given packet reside in the same module. Let each module have an input buffer of $Y$ words
where $y = p$. A word can be transferred from the buffers to the processors in a minimum time of $w'$ cycles. If the transfer of a packet is
initiated at time $T_0$ and the data elements are transferred at the maximum rate, then the first element of the packet is transferred by the
time $T_0 + 2w$,

$$\vdots$$

the $p$th element is transferred by the time $T_0 + pw$. In order to transfer a data element at the time shown above, it is necessary that the
data element be available in the buffer before the initiation of the
transfer. That is, the first element of the packet must be available
in an input buffer by the time $T_0$, the second element must be available
in a buffer by the time $T_0 + w$,

$$\vdots$$

the $p$th element must be available in a buffer by the time $T_0 + (p-1)w$.

In the case of the example being considered, all the elements of an
input packet reside in the same module. Therefore, in order to be able to transfer a given data packet at the maximum rate, the first element must be available in the input buffer of some module by the time $T_0$, ..., C.1 the second element must be available in the same buffer by the time $(T_0+w)$, ..., C.2

C.2

the $p$th element must be available in the same buffer by the time $T_0 + (p-1)w$ ... C.p.

If any element is not available in the buffer (but has to be accessed from the memory) at the time stated for it above, then the data transfer would take more time since the transfer is delayed until the element is available in the buffer and can be sent to the processors. Hence, the data transfer would proceed at a rate slower than the maximum rate.

Consider an arbitrary packet, say the $j$th packet, containing the elements $X_1[j], X_2[j], ..., X_p[j]$. In case of the example being considered, all these elements reside in the same module, say $M_r$. If the buffer size is $p$ words and all the $p$ elements of the $j$th packet are available in the buffer at time $T_0$, then the $j$th packet would indeed be transferred at the maximum rate. This follows from the fact that the element $X_i[j]$ is available in the buffer at time $T_0$ which is less than or equal to $T_0 + (i-1)w$ for all $i$ such that $1 \leq i \leq p$. That is, $X_i[j]$ is available in the buffer before it is needed to be transferred. However, when the buffer size is $y$ words, $y < p$, a maximum of
only y elements can be available in the buffer at time $T_0$. The remaining $(p-y)$ elements have to be fetched from the memory after the transfer of the packet has been initiated. That is, at time $T_0$ the buffer contains the elements $X_1[j], X_2[j], \ldots, X_y[j]$. The elements $X_{y+1}[j]$ to $X_p[j]$ remain to be fetched. The element $X_{y+1}[j]$ cannot be fetched until there is an empty location in the buffer. The memory access for $X_{y+1}[j]$ is initiated after $X_1[j]$ has been transferred to the processors because that leaves an empty location in the buffer. $X_1[j]$ is transferred by the time $T_0 + w$. Then $X_{y+1}[j]$ is fetched from the memory $M_C$ cycles later by the time $(T_0 + w + M_C)$. $X_2[j]$ is transferred by the time $T_0 + w + w = T_0 + 2w_1$. Since $w < M_C$, $X_2[j]$ is transferred before the fetch of $X_{y+1}[j]$ is completed. Thus, after $X_{y+1}[j]$ is fetched into the buffer, there are only $(y-1)$ elements in the buffer. The next memory access is, therefore, initiated immediately following the completion of the access of $X_{y+1}[j]$. Then, $X_{y+2}[j]$ is accessed by the time $T_0 + w + M_C + M_C = T_0 + w + 2M_C$. Similarly, $X_{y+3}[j]$ is accessed by the time $T_0 + w + 3M_C$, and so on. $X_p[j]$ is accessed by the time $T_0 + w + (p-y)M_C$. In order to maintain data transfer at the maximum rate it is necessary that

- $X_{y+1}[j]$ be available in the buffer before time $T_0 + yw$ (from C.y+1),
  that is, $(T_0 + w + M_C) \leq T_0 + yw$ \hspace{1cm} -- E 3.5.1,

- $X_{y+2}[j]$ be available in the buffer before time $T_0 + (y+1)w$ (from C.y+2),
  that is, $(T_0 + w + 2M_C) \leq T_0 + (y+1)w$ \hspace{1cm} -- E 3.5.2,

  
  
- $X_p[j]$ be available in the buffer before time $T_0 + (p-1)w$ (from C.p),
that is, \( T_0 w + (p-y)M_c + T_0 + (p-1)w \) \( \text{-- E 3.5.1} \).

From E 3.5.1 we get \( \frac{y}{w} \geq \frac{M_c}{w} + 1 = \frac{M_c}{w} + (1-2) \),

from E 3.5.2 we get \( \frac{y}{w} \geq \frac{2M_c}{w} = \frac{2M_c}{w} + (2-2) \),

from E 3.5.3 we get \( \frac{y}{w} \geq \frac{3M_c}{w} - 1 = \frac{3M_c}{w} + (3-2) \),

from E 3.5.4 we get \( \frac{y}{w} \geq \frac{p - \frac{(p-2)w}{M_c}}{w} = \frac{M_c}{w} + (l-2) \),

where \( l = (p-y) \).

\[ y \geq \frac{M_c}{w} + (l-2), \]

\[ > \frac{M_c}{w} + (l-2) - \left( \frac{M_c}{w} - 1 \right) = (l-1) \frac{M_c}{w} - (l-3), \]

\[ > (l-1) \frac{M_c}{w} + (l-3) - \left( \frac{M_c}{w} - 1 \right) = \frac{(l-2)M_c}{w} + (l-4), \]

\[ \vdots \]

\[ > \frac{2M_c}{w} - (2-2) \]

\[ > \frac{M_c}{w} - (l-2). \]

That is, \( y \geq \frac{M_c}{w} + (l-2) \) satisfies all the inequalities E 3.5.1 to E 3.5.4.

Thus, it is necessary to have \( y \geq \frac{M_c}{w} + (l-2) \). Substituting \( l = (p-y) \) we get the condition \( y \geq p - \frac{(p-2)w}{M_c} \), in order to be able to transfer data at the maximum rate.

A buffer can contain only an integer number of words. So in order that \( y \geq p - \frac{(p-2)w}{M_c} \) and \( y \) be an integer, it follows that

\[ y \geq \left\lfloor p - \frac{(p-2)w}{M_c} \right\rfloor. \]
b) The output buffers store the results which are sent to the memory from the processors. Each output packet contains \( q \) elements. A processor sends out the output packets at the maximum rate of one element per \( w \) cycles. It is necessary that the buffers be able to receive and store these \( q \) elements at the maximum rate. Consider the case where all the output variables start in the same memory module, then all the \( q \) elements of a packet go to the output buffer of the same module. Since the processing time of an input packet depends on the operation being performed and the nature of the processors, it can be quite arbitrary. Thus, the result packet may be sent to a memory module when the module is fetching the input variables. In other words, all the output data elements have to be buffered immediately without the possibility of any element being stored. The output buffer thus must have \( q \) locations to hold the \( q \) elements of a data packet.

**PROVED**

We now show that \( p \) words and \( q \) words per buffer are sufficient for input and output buffer space per module, respectively, in order to maintain data transfer at the maximum rate.

**Lemma 3.3:**

a) If \( X_i[j] \), an element of the input variable \( X_i \), is fetched from a given module \( M_p \) at some time \( T_0 \), then the \( p \)th input element, after \( X_i[j] \), to be accessed from that module is \( X_i[j+M] \).
b) Similarly, if $X_i[j]$, an element of the output variable $X_i$, is stored in a given module $M_r$, at some time $T_0$, then the qth element, after $X_i[j]$, to be stored in $M_r$ is $X_i[j+M]$.

Note: An access, here, refers to a successful store or fetch operation.

This lemma follows from the facts

i) that the input and output variables are accessed in a cyclic sequence. That is the input variables are accessed in the cyclic sequence $X_1, X_2, \ldots, X_p, X_1, X_2, \ldots$ and the output variables are accessed in the cyclic sequence $X_{p+1}, X_{p+2}, \ldots, X_{p+q}, X_{p+1}, X_{p+2}, \ldots$.

ii) the vectors are stored in an interleaved fashion so the elements of $X_i$ in module $M_r$ are $X_i[j], X_i[j+M], X_i[j+2M], \ldots$.

**Lemma 3.4:** If the data elements are transferred at the maximum rate, then $X_i[j+M]$ will be transferred $M_r$ clock cycles after $X_i[j]$, for any variable $X_i$.

This lemma follows from the fact that there are $M$ packets between $X_i$ and $X_i[j+M]$ and it takes $r_m$ cycles to transfer each packet.

**Lemma 3.5:** Let the first element of variable $X_i$ in a module $M_\alpha$ be $X_i[j]$. The subsequent elements of $X_i$ in $M_\alpha$ are $X_i[j+M], X_i[j+2M], X_i[j+3M], \ldots$.

For both the IAS and the MSAS schemes, if the input buffer size is $p$ words and if the transfer of an element $X_i[j+M]$ is completed by the
time $T_1$, then the next element of $X_i$ in module $M_k$, that is, $X_i[j+(k+1)M]$ will be available in the buffer before the time $T_1 + Mr$, Therefore, $X_i[j+(k+1)M]$ can be transferred after the time $T_1 + Mr$, where $k = 0, 1, 2, \ldots$.

**PROOF:** The element $X_i[j+k\cdot M]$ is transferred from the input buffer at time $T_1$. The element $X_i[j+(k+1)M]$ is the $p$th input element after $X_i[j+k\cdot M]$ from Lemma 3.3. There are $(p-1)$ elements before $X_i[j+(k+1)M]$ and after $X_i[j+k\cdot M]$. Let these elements be $X_{i+1}[j_1], X_{i+2}[j_2], \ldots, X_p[j(p-i)], X_1[j(p-i+1)], \ldots, X_{i-1}[j(p-1)],$ where $j + k\cdot M \leq j_1 \leq j_2 \leq \ldots \leq j(p-1) \leq j + (k+1)M$.

--- E 3.2

The expression E 3.2 is the condition that ensures that the IAS scheme fetches elements into the buffers in an increasing order with respect to the indices. So $(p-1)$ elements have to be fetched before $X_i[j+(k+1)M]$ can be fetched.

At time $T_1$ there may be $y$ elements already in the buffer. The maximum time taken to fetch the remaining $(p-y)$ elements = $(p-y) M_c$. At worst $y = 0$, then, the time taken to fetch $p$ elements = $p M_c$.

The worst case time will be $p M_c$. Note that once the buffer is full, no more fetch is initiated.

The addressing scheme accesses input and output variables in the cyclic sequence $X_1, X_2, X_3, \ldots, X_i, \ldots, X_p, X_{p+1}, X_{p+2}, \ldots, X_{p+q}$. So in order to access the $p$ input elements of the variables $X_{i+1}, X_{i+2}, \ldots, X_p, X_1, X_2, \ldots, X_{i-1},$ the address sequence fol-
lowed by the memory is

$$X_{i+1}, X_{i+2}, \ldots, X_p, X_{p+1}, \ldots, X_{p+q}, X_1, X_2, \ldots, X_{i-1}.$$  

In between the p fetches, q store cycles are also executed. In that case, the total worst case time to fetch the p elements is $$pM_cm + qM_cm = V_mM_cm$$ cycles. Since $$X_{i+l}(k+1)M$$ is the pth element, it is fetched within $$V_mM_cm$$ cycles after time $$T_1$$. From the expression E 3.1, $$V_mM_cm \leq Mr_m$$ Therefore, $$X_{i+l}(k+1)M$$ is fetched before the time $$T_1 + Mr_m$$

PROVED--

This lemma shows that a p word buffer ensures that all the data elements of a variable are always available in the input buffer before they are needed to be sent to processors (except for the first element of that variable in every module, that is the first M elements of that variable).

It should be noted that the p elements held in the p word buffer are always one element of each input variable. This is assured in both the IAS and the MSAS schemes by accessing all the variables in a cyclic sequence. If a distinct one-word buffer is used per variable or a single p-word buffer is used, we are assured that the p elements are properly buffered. Thus Lemma 3.5 applies to both the IAS and the MSAS schemes.

Theorem 3.4: Assuming that the processing unit operates at the maximum throughput, and that $$Mr_m \geq V_mM_cm$$, then a p word buffer per module is sufficient to ensure that the memory system will transfer data to the processors at the maximum rate of one packet per $$r_m$$ cycles, after the first p memory cycles.
**Proof:** The proof is in two parts. If the $M$th packet is completely transferred by time $t_{Mp}$, then the first part of the proof shows that the memory system maintains data transfer at the maximum rate from the time $T_0 + pM_c$ to the time $t_{Mp}$. $T_0$ is the time at which the vector operation is initiated and $T_0 + pM_c$ is the time at which $p$ memory cycles are completed by every module. The second part uses Lemma 3.5 to show that all packets after the $M$th packet are transferred at the maximum rate. That is all data after the time $t_{Mp}$ proceeds at the maximum rate.

a) At the beginning of a vector operation, time $T_0$, all the buffer locations in every module are empty. All the modules start access in parallel and can fetch $p$ words each in parallel in $p$ memory cycles. So in $pM_c$ cycles the memory modules have fetched $p$ elements each. In the duration $pM_c$ some elements are sent out in packets to the processors. In the $p$ accesses, one element of each input variable is fetched in each module. Thus, in $M$ modules the first $M$ elements of each input variable are available in the buffers after $p$ accesses, that is, after the time $T_0 + pM_c$. The first $M$ elements of each variable form together the first $M$ input packets. Some of these packets are sent to the processors during the first $p$ memory accesses. The remaining packets are sent after the time $T_0 + pM_c$. Let $X$ be the number of remaining packets. All the elements of these $X$ packets are already in the buffer so they can be sent to the
processors at the maximum rate.

b) If the transfer of the first element of the first packet was
initiated at time $t_1$ and the last element of the Mth packet
was transferred at time $t_{Mp}$ then $t_{Mp} \geq t_1 + Mr_m$. -- E 3.5.1
This is because $Mr_m$ is the time taken to transfer M packets at
the maximum rate and the first $(M-x)$ packets may not be trans-
ferred at the maximum rate, Lemma 3.4. Consider the $(M+1)$st
packet. The first element is $X_1[M+1]$. From Lemma 3.5
$X_1[M+1]$ is available in the buffer before the time $t_1 + Mr_m$. So
it can be sent immediately after the transfer of the Mth
packet, that is, after $t_{Mp}$ by the time $t_M + w$. The second
element, the $(M+1)$st packet, is $X_2[M+1]$. Again by Lemma 3.5
$X_2[M+1]$ is available in the buffers before $t_2 + Mr_m$ where $t_2$
is the time at which $X_2[1]$ was transferred. Since the first M
packets may not have been transferred at the maximum rate
$t_2 \geq t_1 + w$ where $w$ is the time taken for transfer of an ele-
ment. Then from E 3.5.1 we get $t_{Mp} + w \geq t_2 + Mr_m$. This implies
that the element $X_2[M+1]$ is available in the buffer before it
is needed to be transferred. Hence, it can be transferred
immediately after the element $X_2[M+1]$. That is, $X_2[M+1]$ is
transferred by the time $t_{Mp} + 2w$. In general, using lemmas
3.4 and 3.5 repeatedly, we can show that all the elements of
(M+1)st packet and subsequent packets are always available in the buffers before they are needed to be sent to the processors, thus assuring a maximum data rate for the transfer of packets.

We have shown (1) that after the time $T_0 + pM_c$ the $(M+1)$ packet to $M$th packet are sent at the maximum rate, and (2) all packets after the $M$th packet are sent to the processors at the maximum rate.

These two can be combined to give the result that after the time $T_0 + pM_c$, data is transferred at the maximum rate from the memory.

**Theorem 3.5:** For both the IAS and MSAS schemes a q word output buffer space is sufficient to maintain the maximum rate of data transfer from the processors to the memory, given that the condition E 3.1, that is $M \nu_m \geq M_c \nu_m$, holds true.

**Proof:** Consider the output buffer of an arbitrary module $M_a$. It can hold q words. This buffer size is insufficient if at some time the buffer already has q elements and a processor sends another element to be stored in module $M_a$. Consider an instance when the buffer holds q elements. Let the first element (to be stored), among the q elements in the buffer, be $X_i[j]$. Then from Lemma 3.3, the $(q+1)$st element to be sent to $M_a$ will be $X_i[j+M]$. Let the time at which $X_i[j]$ is received from a processor be $T_1$. We will now show that $X_i[j]$ is always stored in the module before $X_i[j+M]$ is received from the processor, thus there are a maximum of $(q-1)$ occupied
buffer locations when \( X_i[j+M] \) is received.

The earliest \( X_i[j+M] \) can be received by the module \( M_a \) is when the processors send results at the maximum rate. From Lemma 3.4, \( X_i[j+M] \) will be received earliest at the time \( T_1 + M \cdot r_m \). Consider, now, the storing of \( X_i[j] \). From the addressing algorithms of IAS and MSAS schemes it follows that each variable is accessed (fetched or stored) once every \( V_m \) memory cycles. Then, \( X_i[j] \) will be stored within \( V_m \) memory cycles after the time of its being received in the buffer. That is \( X_i[j] \) will be stored before or by the time \( (T_1 + M_c \cdot V_m) \).

From E 3.1, \( M_c \cdot V_m \leq M \cdot r_m \), therefore
\[
T_1 + M_c \cdot V_m \leq T_1 + M \cdot r_m.
\]
That implies that \( X_i[j] \) is stored in module (thus leaving an empty buffer location) before \( X_i[j+M] \) is received. When \( X_i[j+M] \) is received there is at least one empty buffer location and \( X_i[j+M] \) is stored in the buffer without any conflict. In the MSAS scheme, the buffer location for the variable \( X_i \) will be empty and since the \((q+1)st\) element to be received is also an element of \( X_i \), that is \( X_i[j+M] \), it is buffered in that space without any conflict. Thus we have shown that it is never the case that more than \( q \) words are required to be stored at the same time in the output buffer of a module. In other words a \( q \)-word size is sufficient to maintain any data transfer rate less than or equal to the maximum data transfer rate of one packet per \( r_m \) cycles.

**PROVED**

From Theorems 3.2, 3.4 and 3.5 it follows that the optimal input buffer size for the MSAS scheme is \( p \) words (one word per variable) per
module, and the optimal output buffer size is \( q \) words per module. Similarly, for the IAS scheme the optimal output buffer size is \( q \) words per module, whereas a sufficient size for the input buffer is \( p \) words. It should be noted that \( p \) words is an optimal input buffer size for the IAS scheme for \( p < 2^{w-\frac{M}{w}} \cdot \left\{ \begin{array}{l} p < 2 + \frac{M}{w} \implies \left\lfloor p - \frac{(p-2)w}{M} \right\rfloor = p \end{array} \right\}. \)

In the following chapters it is assumed the memory system has sufficient buffer space to allow data transfer to proceed at the maximum rate. It was shown earlier that there is some initial time during which the packets are not sent to the processors at the maximum rate. There are some delays (depending on the variable distribution) in the transfer of elements during the first \( p \) memory cycle because the elements are not available in the buffers when required. The worst total initial delay is in the case of all variables starting in the same module. In this case even the first packet cannot be completely sent to the processors until the first \( p \) memory cycles are completed and the last element of the first packet is also available in the buffer. Thus the maximum initial delay is \( p \) memory cycles = \( pM \). Similarly, there is some delay associated with storing the last results after the processors have finished processing. Let the last result be received in a module at time \( T_2 \). There are at worst \( (q-1) \) elements ahead of the last element in the buffer. Since the module is no longer fetching any more data elements (since the vector operation is completed) the memory module can only be busy in storing the results. Then, at worst, the last element will be stored within \( q \) memory cycles after it is received by the buffer, that is, by the time \( T_2 + qM \). This shows that there is
Maximum delay of qMc in storage after the processors have completely finished processing.

The total time for accessing, processing and storing for a vector operation of length $Z$ is then at worst $pM_{c}+T_{p}+qM_{c}$ where $T_{p}$ is the processing time of a vector operation of length $Z$. The time $T_{p}$ was shown to be $R+Zr_{m}+r_{\min}+\left(\frac{Z}{N}\right)-1\left(R-(N-1)r_{m}\right)$ in Chapter II. Hence, the total time to execute a vector operation is given by $T$, where

$$T = (p+q)M_{c}+R+Zr_{m}+r_{\min}+\left(\frac{Z}{N}\right)-1\left(R-(N-1)r_{m}\right).$$  \hspace{1cm} \text{--- E 3.3}$$
CHAPTER IV
INTERCONNECTION SCHEMES

4-1 Introduction

In order to have a high throughput for a parallel memory system it is necessary to have some interconnection network that \( i \) collects the data from the memory system and transfers it to the processors in an efficient manner, and \( ii \) receives the data from the processors and stores it in the appropriate memory modules efficiently. Since the data elements to be sent to a processor may reside in different (and arbitrary) modules, the interconnection network must keep track of the modules from which the data is fetched. This problem of data alignment is discussed in detail in section 4-2. An interconnection network for the modified simultaneous Access Scheme is presented in section 4-3. It is shown that this network is modular and it allows the memory to transfer data as fast as the processing system can accept it. It is also shown that the same network, with minor modifications, can be used for the Independent Access Scheme. In section 4-4 it is shown that the same interconnection network can also be used to unscramble d-ordered vectors and skewed arrays.

It has been assumed in the earlier chapters that the processors operate independently and asynchronously. Section 4-5 describes the signalling scheme used by the processors to receive and transfer data on the busses in order to maintain an asynchronous behaviour with respect to the memory. In section 4-6 the details of implementation of the processor control loops are described. It is shown that the control loops can indeed be implemented with single control lines. It is shown that the control hardware required to allow processors to communicate with the memory under the control of the control loops is very simple.
The problem of data alignment of vectors in parallel memories was mentioned in chapters I and II. In this section the problem of data alignment is examined from the viewpoint of array processing in a stream-processing system. Consider an operation \( C = A \times B \), where \( A, B, C \) are vectors of length \( Z \). Let the memory system consist of \( M \) modules. Vectors \( A, B \) and \( C \) are stored in an interleaved fashion in these modules. In general the vectors \( A, B \) and \( C \) may start in different modules, as shown in Fig. 4.1. Although data elements \( a_i \) and \( b_i \) belong to the same data packet they may not reside in the same memory module. Consider an \( M \) memory-\( M \) processor system. If processor \( P_1 \) operates on the first data packet, \( P_2 \) on the second data packet and so on, then the packet \( <a_1, b_1> \) has to be sent to \( P_1 \), \( <a_2, b_2> \) to processor \( P_2 \) and so on. In order to accomplish the data packet transfer, the first \( M \) elements of \( A \) are accessed in parallel and transferred to the \( M \) processors as shown in Fig. 4.1. The interconnections required are \( M_1 \) to \( P_1 \), \( M_2 \) to \( P_2 \), \ldots etc. Next, the first \( M \) elements of \( B \) are transferred to the \( M \) processors as shown in Fig. 4.2. The required interconnections are \( M_j \) to \( P_1 \), \( M_{j+1} \) to \( P_2 \) and so on, which is different from the interconnection scheme required for the elements of \( A \). However, these interconnection schemes are necessary to insure that each processor gets the correct set of data elements. Ensuring that each processor gets the correct set of data elements, irrespective of which memory modules they reside in, is called data alignment. As mentioned in section 1.3, many schemes have been suggested for providing the required interconnections for \( N \) memory-\( N \) processor systems. A review of these schemes is given in [KUCK 75].

Data alignment is considerably simplified in the case of pipeline or
stream processing systems. It is necessary to generate the data packets in a sequence only. That is, it is necessary to transfer only one data element at a time. Consider the Simple Access Scheme described in section 3.2. A row of each vector is accessed in parallel and sent to the buffers serially on the data bus. Thus in case of the example considered earlier M data elements of A are fetched first and sent in sequence on the data bus. That is, the interconnections required from the memory data registers to the bus are first $M_1$ to the bus (for the transfer of $a_1$), next $M_2$ to the bus, next $M_3$ to the bus and so on till $M_M$ is connected to the bus, Fig. 4.3. In the meantime a row of B is fetched in parallel to the memory data registers. So in order to transfer the elements of B to the buffers in the correct sequence, the interconnections required are, first, $M_j$ to the bus, then $M_{j+1}$ to the bus, and so on. This is shown in Fig. 4.4. The interconnection paths in this case are the same as those used for the transfer of elements of vector A. The difference is that the data transfer is started from a different module in case of vector B as compared to vector A.

This scheme, then, needs only M interconnection paths but also needs some central control to

1) allow only one memory module to transfer data to the bus at one time,

2) to allow the M memory modules to transfer data in a cyclic sequence starting with any given memory module.

As mentioned earlier the simple access scheme has the disadvantage of additional delay caused by streaming M elements of each vector before any packets can be sent out.
First M elements of B:

\[ b_{m-j+2} \quad b_{m-j+3} \]

\[ b_1 \quad b_2 \quad b_{m-j+1} \]
In the case of the Modified Simple Access Scheme or the Independent Access Scheme, the requirements for data alignment are almost the same as those for the Simple Access Scheme. Fig. 4.5 illustrates the access of the two vectors A and B in case of the example considered earlier, for the modified Simultaneous Access Scheme. The vector elements are read into switched buffers in each memory module. From the buffers the data elements are collected into packets and sent to the processors. It is, then, necessary to transfer a complete data packet at one time on the data bus. First, a₁ is transferred from module M₁ to the bus, followed by b₁ which is transferred from module M_j to the bus. Next, a₂ is transferred from module M₂ followed by element b₂ from the module Mₖ₊₁, and so on. Thus the sequence in which the interconnection paths are used is M₁ to the bus, M_j to the bus, M₂ to the bus, Mₖ₊₁ to the bus, and so on. The modules get access to the bus in the sequence shown below.

\[ M₁ \rightarrow M₂ \rightarrow M₃ \rightarrow M₄ \rightarrow \ldots \]

\[ M_j \rightarrow Mₖ₊₁ \rightarrow M₃₊₂ \rightarrow Mₖ₊₃ \rightarrow \ldots \]

It can be seen that the sequence is formed of cyclic subsequences which are interlaced. In the example above the subsequences are

\[ M₁, M₂, M₃, \ldots, Mₖ \] and

\[ M_j, Mₖ₊₁, \ldots, Mₖ, M₁, M₂, \ldots, Mₖ₋₁ \] In section 4.3 an interconnection network is presented, which ensures the required data alignment by implementing the control for cyclic subsequences in the form of control loops.
Array Storage. The problem of data alignment is different when storage and access of arrays are considered. For example, consider a $4 \times 4$ array, $A$, stored in a 4 module memory. The array is stored row by row across the modules, as shown in Fig. 4.6.

Any row of the array can be fetched in one memory cycle. However, to access a column of $A$, say the first column, then the bandwidth is cut by one-fourth because of access conflicts as shown in the underlined elements in Fig. 4.6. Since all the elements of the first column reside in the first module it takes 4 memory cycles to fetch the complete column. However, all the elements of the forward diagonal reside in different memory modules, hence can be fetched in parallel in one memory cycle.

The desirability of fetching rows, columns and diagonals stems from the consideration of various common matrix operations [KUCK 68], [SENZIG 65]. It is of importance, then, to examine techniques for storing arrays which lead to fast access of rows, columns and diagonals. This problem has been extensively studied by [KRASKA 69], [BUDNICK 71], etc. One of the more efficient ways of storing of arrays is called the skewing scheme [KUCK 75]. A vector is said to have a skewing distance $\delta$ if each successive element of the vector is stored $\delta \pmod{M}$ memory modules away from the previous element. In case of array storage, each dimension of the array is treated as an independent vector. Let $\delta_i$ be the skewing distance in the $i$-th dimension. Then the storage scheme for a $k$-dimensional array is called a $(\delta_1, \delta_2, \ldots; \delta_k)$ skewing scheme. Considering the example of Fig. 4.6, if a $(1,1)$ skewing scheme is used, the elements will be stored in the memory as shown in Fig. 4.7A. Note that the data
Underlined elements belong to a column.

<table>
<thead>
<tr>
<th>a_{11}</th>
<th>a_{12}</th>
<th>a_{13}</th>
<th>a_{14}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
<td>a_{23}</td>
<td>a_{24}</td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
<td>a_{33}</td>
<td>a_{34}</td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
<td>a_{43}</td>
<td>a_{44}</td>
</tr>
</tbody>
</table>

Circled elements belong to the diagonal.

<table>
<thead>
<tr>
<th>a_{11}</th>
<th>a_{12}</th>
<th>a_{13}</th>
<th>a_{14}</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_{21}</td>
<td>a_{22}</td>
<td>a_{23}</td>
<td>a_{24}</td>
</tr>
<tr>
<td>a_{31}</td>
<td>a_{32}</td>
<td>a_{33}</td>
<td>a_{34}</td>
</tr>
<tr>
<td>a_{41}</td>
<td>a_{42}</td>
<td>a_{43}</td>
<td>a_{44}</td>
</tr>
</tbody>
</table>

FIG. 4.6
<table>
<thead>
<tr>
<th></th>
<th>A11</th>
<th>A12</th>
<th>A13</th>
<th>A14</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>a11</td>
<td>a12</td>
<td>a13</td>
<td>a14</td>
</tr>
<tr>
<td>A2</td>
<td>a24</td>
<td>a21</td>
<td>a22</td>
<td>a23</td>
</tr>
<tr>
<td>A3</td>
<td>a33</td>
<td>a34</td>
<td>a31</td>
<td>a32</td>
</tr>
<tr>
<td>A4</td>
<td>a42</td>
<td>a43</td>
<td>a44</td>
<td>a41</td>
</tr>
</tbody>
</table>

A. 

<table>
<thead>
<tr>
<th></th>
<th>A11</th>
<th>A12</th>
<th>A13</th>
<th>A14</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>a11</td>
<td>a12</td>
<td>a13</td>
<td>a14</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>a24</td>
<td>-</td>
<td>a21</td>
<td>a22</td>
<td>a23</td>
</tr>
<tr>
<td>A3</td>
<td>a32</td>
<td>a33</td>
<td>a34</td>
<td>-</td>
<td>a31</td>
</tr>
<tr>
<td>A4</td>
<td>-</td>
<td>a41</td>
<td>a42</td>
<td>a43</td>
<td>a44</td>
</tr>
</tbody>
</table>

B. 

**FIG. 4.7**
elements belonging to a column reside in different modules, as do the elements of a row. However, some data elements of a diagonal (circled elements) now conflict for access as they reside in the same module. If a \((2, 1)\) scheme is considered with \(M = 5\), their data is stored as shown in Fig. 4.7B. It can be seen that any four elements of a row, a column, or a diagonal reside in different memory modules. Thus a complete row, column, or diagonal can be fetched in one memory cycle. However, this scheme does waste some storage locations. It has been shown by KUCK [KUCK 75] that if we use \((\delta_1, \delta_2)\) skewing then in order to access the three types of \(n\)-vectors (vector of \(n\) distinct elements) the following conditions must hold:

\[
\begin{align*}
  m & \geq n \gcd (\delta_1, m) \\
  m & \geq n \gcd (\delta_2, m) \\
  m & \geq n \gcd (\delta_1 + \delta_2, m)
\end{align*}
\]

(rows),

(diagonals).

Thus, given the nature of data and available memory modules, these conditions can be used to select the appropriate scheme to obtain conflict-free access.

However, looking back at the example of Fig. 4.7B, if the first column is accessed in one memory cycle, the output words will be in the following order:

\[
a_{11} \quad a_{41} \quad a_{21} \quad \cdots \quad a_{31}
\]

It is clear that the output is not in a convenient sequential order so that the data elements may be picked off in a sequence to be sent to the processors. In other words, the output is scrambled. The output vector is called a 2-ordered vector since \(\delta = 2\). The problem of rerouting the
the data elements in order to unscramble a d-ordered vector has been studied by Swanson [SWAN 74]. He has shown that it is possible to reorder the vectors efficiently using "k-apart interconnection networks."

He has also derived upper and lower bounds for the number of routings necessary to achieve the final order, for a given d-ordered vector and a given number of memory units. An example of unscrambling for a 2-ordered vector using 3-apart interconnection is shown in Fig. 4.8. The solid lines represent the interconnection. Note that it takes 2 routings through the network to unscramble the vector. In general, only for certain values d and M can a given k-apart interconnection network unscramble a d-ordered vector.

In section 4-3 it is shown that the problem of unscrambling is considerably simplified by sending only one data packet at a time. Thus it is necessary to route only one element of a vector at any given time. It is shown that one simple network can handle efficiently both the data alignment problem and the problem of unscrambling a d-ordered vector for small values of d.
**2-ordered vector in registers**

0 4 1 5 2 6 3

Registers after first routing through network

0 5 3 1 6 4 2

Registers after second routing through network

**Fig 4.8**
4-3 The Interconnection Network

The functions of the interconnection network are

i) to align data elements and form an input packet,

ii) to transfer the input packets to the processors at some desired rate,

iii) to receive data packets from the processors as fast as they come,

iv) to store the data elements of an output packet into the correct modules without too much additional delay,

v) to allow the memory system to operate independently of the processing system.

It is desirable that the network introduce minimum additional delay in the transfer of data between the memory and the processing system. As mentioned earlier, the objective of the memory system is to be able to transfer data at the maximum rate dictated only by the bandwidth of the data busses and the processing system.

First, the interconnection network for the Modified Simultaneous Access Scheme is presented. It is shown that the same network, with some minor modifications, can be used for the Independent Access Scheme. Fig. 4.9 illustrates the network for the MSAS. Each module contains a set of switched buffers, one buffer for each variable. If the maximum number of variables permissible in the set of vector operations is \( V_{\text{max}} \) then the number of buffers in each module is \( V_{\text{max}} \). The buffers constitute the storage part of the network. It is necessary to hold the data temporarily in order to align the data and form packets, as explained earlier. The alignment and selection of data buffers in order to form the data
packets is done by use of the variable select lines and the variable control loops. The select lines and the control loops allow the selected data buffers to send or receive data on the data busses.

The detail of the selection mechanism is shown for a module in Fig. 4.10. The number of variable select lines and variable control loops is the same as the number of buffers per module. The control loops are single bit cyclic shift registers with only one logical bit circulating in each loop. A particular buffer is selected if the select line for the corresponding variable is asserted and the corresponding control loop has the control bit set to that module. This is illustrated in Fig. 4.10 by the dark lines. A variable select line serves to select the M buffers for that variable, one in every module, and the control loop for that variable selects one buffer out of the M buffers. The variable select line can be looked upon as a selection line for a given control loop. If the input data bus is one word wide, only one buffer may transfer data on it at a given time. Thus, only one select line may be asserted for transfer of data to the processors at a given time. Similarly, the output data bus is only one word wide, thus allowing only one select line to be asserted for data transfer from the processors to the memory. This is ensured by the Control Interface. However, if the data busses are wider, then more words can be transferred in parallel by asserting more than one select line.

*Packet formation and alignment.* This is best explained by an example. Consider the operation $A + B \times C$, where $A$, $B$ and $C$ are vectors. As explained earlier in Chapter III, the memory system has to generate data packets $<a_1, b_1>$, $<a_2, b_2>$, ..., $<a_2, b_2>$, where the length of the
FIG. 4.10
vectors is equal to Z. The memory receives the data packets \(<C_1>, <C_2>, \ldots <C_z>\) from the processors. Consider a four module memory where vector A starts in module 1, vector B in module 3 and vector C in module 4. Each module contains three buffers, one for each variable. Only three select lines and three control loops are used, one select line and one control loop is assigned to each variable. Thus, the A select line selects the A-control loop which in turn selects a particular A-buffer. Similarly, the B select line and the B-control loop select a B-buffer and the C-control line and the C control loop select a C-buffer. When the memory is accessed according to the MSAS scheme the data elements are available in the buffers as shown in Fig. 4.11.

In order to send the first packet to the processors, it is necessary to transfer \(a_1\) from module 1 followed by \(b_1\) from module 3. The control loops are initially set to the modules in which the corresponding variables start. Thus the control bit of the A-control loop is initially at module 1, the control bit of the B-control loop is at module 3 and the control bit of the C-control loop is at module 4. Then the data element \(a_1\) is selected by asserting the A-select line. After \(a_1\) is transferred, \(b_1\) is selected by asserting the B-select line. Note that the initial setting of the control loops takes care of the alignment for the first packet. Next, the second packet has to be sent to the processors. That is, \(a_2\) has to be transferred from module 2 and \(b_2\) from module 4. If the A-control loop and the B-control loop are shifted cyclically by one place (to the right, in Fig. 4.11) then the A-control loop selects module 2 and the B control loop selects module 4, as shown in Fig. 4.12 by the dark lines. Then, the data elements \(a_2\) and \(b_2\) are easily selected and trans-
ferred by asserting the A-select line followed by the B-select line. The control loops are shifted again by one place cyclically. The A and B control loops now point to module 3 and module 1 thus selecting elements $a_3$ and $b_3$. Thus the third packet can be easily sent to the processors. It is easily seen that shifting the control loops ensures that all the data elements of the next packet are selected.

Thus, in order to ensure data alignment it is only necessary to initially set the control loops to the appropriate modules and shift them by a place after the transfer of every packet. The additional delay introduced by the network is merely that of a single shift in the control loops. It should be noted that the A control loop selects the modules in the sequence $M_1, M_2, M_3, M_4, M_1, M_2, \ldots$ and the B control loop selects the modules in the sequence $M_3, M_4, M_1, M_2, M_3, M_4, \ldots$. These are exactly the sequences required for data alignment, as explained earlier in section 4-2.

Since the elements of a packet are always available in separate buffers (even if they are in the same module), they can be transferred in parallel if the data busses are wide enough. Clearly, in case of the example, Fig. 4.11, above, $a_1$ and $b_1$ can be transferred in parallel by asserting A and B select lines at the same time. However, this requires that the input bus be two words wide. Thus, the same network can be used on expanded systems (with larger busses) also.

The output packets are handled in the same manner by the control loops and the select lines. Since vector C starts in module 4, the C control loop is initially set to module 4. The transfer of output data packets on the output bus proceeds in parallel with data transfer on the input bus.
Whenever the element $C_1$ is available on the output bus, the C select line is asserted and element $C_1$ is stored in the C-buffer in module 4. The C-control loop is now shifted cyclically by one place, thus selecting module 1. The next element $C_2$ is then stored in module 1 and so on. Clearly, the elements of C are stored in the modules as desired. The same reasoning applies when the output packets consist of more than one element, each.

We can now state the steps taken for any given vector operation. Consider a vector operation with $p$ input variables, $I_1$ to $I_p$, and $q$ output variables, $O_1$ to $O_q$. If the address sequence for the input variable is $I_1, I_2, \ldots I_p$ then the top $p$ control loops are assigned to the input variables in the same sequence, Fig. 4.9. The next $q$ control loops are assigned to the output variables in the same sequence as defined by the output address sequence. The select lines are also assigned in the same order. Thus the top select line and the top control loop in Fig. 4.9 will select the leftmost buffers in every module. The steps executed for data transfer are:

(i) Initially the control bit of every control loop is set to the module in which the corresponding variable begins.

Input

(ii) The $I_1$-select line is asserted and an element of $I_1$ is transferred. If there is no data element available in the selected buffer then the select line is held high until the data element is obtained and transferred. After the completion of transfer of the element the select line is negated. Thus, after an
element of 1 is transferred the $I_1$-select line is negated. Next, the $I_2$-select line is asserted. An element of $I_2$, selected by the $I_2$ control loop, is transferred. After the completion of the transfer the $I_2$-select line is negated. Next, the $I_3$-select line is asserted and so on. Finally, the $I_p$-select line is asserted and an element of $I_p$ pointed to by the $I_p$-control loop is transferred, thus completing the transfer of a complete packet.

(iii) After the completion of the data packet transfer, the $p$ control loops, $I_1$ to $I_p$, are shifted cyclically by one place (to the right in Fig. 4.9).

(iv) The steps (ii) and (iii) are repeated until all the data packets have been transferred. The count of the number of packets transferred is kept by the Control Interface, as shown in Fig. 4.9.

For output

(ii) First the $O_1$-select line is asserted. The corresponding data element of variable $O_1$ is received from the processor and stored into the $O_1$-buffer pointed to by the $O_1$-control loop. If the data element is not immediately available then the current select line is held asserted until a data transfer is completed. After the completion of the data transfer the current select line is negated. After the $O_1$-select line is negated, $O_2$-select line is asserted and the next data element is transferred, and so on until $O_p$-select line is asserted.
and the final data element of the output packet is transferred from the processors to the buffers.

**iii)** After the completion of the transfer of one packet, all the output control loops are shifted by one place cyclically (to the right in Fig. 4.9).

**iv)** The steps **ii)** and **iii)** are repeated until all the output packets have been transferred.

**Transfer rate.**

The total time taken to transfer an input packet is the time taken to transfer p words on the data bus and the time taken to shift the control loops. If the time for a shift by one place is treated as a basic clock cycle and if the time taken to transfer a word from a buffer to the processors is 'w' clock cycles, then the total time taken for the transfer of an input packet, \( r_1 \), is given by the following expression, \( r_1 = (1 + p \cdot w) \) cycles. Here 'w' is determined by delays in signalling between the Control Interface and a processor and the bandwidth of the data bus in order to transfer a word to a given processor. The signalling required for a processor to accept a data word is discussed in section 4.5. Clearly, then the p words are transferred at the maximum possible rate, which is dictated by the bandwidth of the bus and the signalling to the processors. The only additional delay is of one cycle because of the time required to shift the control loops. However, once a packet is transferred to a processor, the I-bit in the processor control loop is also shifted to the next processor. Therefore no data transfer could be accomplished during that clock cycle, and the con-
trol loops in the interconnection network are shifted in that same cycle. Thus, the memory transfers words as fast as the processing system can accept the data. The assumption made here is that the data is available in the buffers at the same rate as the rate at which processor requests it. This assumption was shown to hold for MSAS and IAS memory schemes in Chapter III.

Similarly, the output packet can be transferred at best in $r_2 = (1 + q_w)$ cycles.

*Interconnection Network for Independent Access Scheme.*

The interconnection network for the Independent Access Scheme is illustrated in Fig. 4.13. The number of control loops and the number of select lines in the network are the same as before (Fig. 4.9). However, since each module has only one input buffer and one output buffer, the selection of a buffer can be done by more than one control loop. The detail of the selection mechanism is illustrated in Fig. 4.14. The input buffer of a module is selected when any of the input variable control loop points to the module and the corresponding select line is asserted. Similarly, the output buffer of a module is selected when any of the control loops for the output variables is set to the module and the corresponding select line is asserted.

The steps performed for data transfer are the same as for the MSAS scheme. Consider the example of the vector operation $C = A \times B$. The contents of the buffers after the first few memory accesses are illustrated in Fig. 4.15. The data elements $a_{\downarrow}$ and $b_{\downarrow}$ are available in modules $M_1$ and $M_3$, as before. The A control loop is set to module $M_1$, the B control loop
FIG. 4.13 INDEPENDENT ACCESS SCHEME
FIG. 4.14
is set to module $M_3$ and the C control loop is set to module $M_4$. Thus the first packet is transferred by asserting A-select line followed by the B-select line, as before. The two control loops are then shifted by one place. The modules $M_2$ and $M_4$ are selected by the loops. Thus, the second packet can be transferred and so on. The IAS scheme ensures that the correct data elements are always available at the front of the buffers and the control loops ensure that the data elements are selected from the correct buffers, as before.

The only disadvantage in performance, when compared with the MSAS scheme, is that the network shown in Fig. 4.13 allows only one word transfer at a time from the input (or output) buffers. Thus, if the bus width is increased to more than one word then the network cannot take advantage of the additional bandwidth. It is necessary to make some changes in the selection gates at the buffers, as shown in Fig. 4.16. The figure illustrates the selection logic for parallel transfer of all the input variables provided the bus is wide enough. Clearly the several data paths from the buffers will transfer data on different parts of the bus as shown. The input data bus as well as the output data bus can be treated as a collection of several busses.

The time taken to transfer input or output packets is the same as that for the MSAS scheme.
4.4 Unscrambling a d-ordered vector.

The unscrambling of a d-ordered vector is accomplished by shifting the corresponding control loop by d places after the transfer of every element of the vector. Consider the example of a 4 by 4 array shown in Fig. 4.18. The array is stored using a (2, 1) skewing scheme in a five module memory. As explained earlier in section 4.2, if the first column is accessed as a vector then the contents of the A-buffers in the five modules are $a_{11}, a_{41}, a_{21}, \ldots, a_{31}$. For simplicity the vector operation is assumed to have only this column vector as the input variable. Then, data elements have to be transferred to the processors in the order $a_{11}, a_{21}, a_{31}, a_{41}$. That is, it is necessary to select module 1, then module 3, then module 5 and finally module 2. The A-control loop is initially set to module 1, since the vector begins there, Fig. 4.19. Clearly, $a_{11}$ will be selected when the A select line is asserted. After $a_{11}$ is transferred the control loop is by two places, cyclically, to the right in Fig. 4.19. The control loop selects module 3. The data element $a_{21}$ can now be transferred by asserting the A select line. After the transfer of $a_{21}$ is completed the A control loop is again shifted by two places, cyclically. The control loop now selects module 5. It is clear that by shifting the control loop by two places every time the next desired element is selected and transferred, in effect unscrambling the original vector.

In the general case of a d-ordered vector it is only necessary to shift the corresponding control loop by 'd' places after the transfer of each data element of the d-ordered vector. Consider a vector operation
(2,1) Skewing storage scheme for arrays. Circled elements belong to the first column.

FIG. 4.18
A 2-ordered vector in a five module memory

FIG. 4.19
which involves three input variables A, B and C. If A is a $d_1$-ordered vector, B is a $d_2$-ordered vector and C is $d_3$-ordered vector, then in order to unscramble all of them, the A control loop must be shifted by $d_1$ places, the B control loop by $d_2$ places and the C control loop by $d_3$ places after the transfer of each input packet. An example of this is illustrated in Fig. 4.20 for $d_1 = 1$, $d_2 = 2$ and $d_3 = 3$, with all the three vectors starting in module 1 in a five module memory. The three control loops are initially set to module 1. The elements $a_1$, $b_1$ and $c_1$ are selected and transferred by asserting the A, B and C select lines. After transfer is complete, the A control loop is shifted by one place, the B-control loop by two places and the C-control by three places. The control loops new select $a_2$, $b_2$ and $c_2$, as shown in Fig. 4.20. It can be seen that after the transfer of every packet it is necessary to shift the A control loop cyclically by one place, the B control loop by two places and the C control loop by three places in order to select the correct elements for the next packet. The total delay in shifting the loops is given by the shift delay of the control loop for the vector with maximum skew. This is obvious since all the control loops are shifted in parallel. In the example above the shift delay is three cycles.

The example brings out three important points:

i) no changes in the network are required to unscramble $d$-ordered vectors for different values of 'd',

ii) no changes are required in the network in order to unscramble a collection of vectors with different skew, at the same time,

iii) the unscrambling operation does incorporate a delay of
'd\_max' clock cycles in the transfer time of a packet, 
where d\_max is the maximum skew.

The total time to transfer an input packet of p data elements is, 
then, given by r\_3 such that

\[ r\_3 = (d\_\max + w.p) \]

where 'w' cycles is the time taken 
to transfer one data element from (or to) the buffers to (or from) the 
processors; also referred to as a data cycle. If the skew was always 
one, as for normally interleaved vectors, the packet transfer time is 
given by r\_1 such that

\[ r\_1 = (1 + w.p), \] 
as shown in section 4-3. Therefore, 
unscrambling of vectors slows the data transfer rate by the factor

\[ \frac{r\_1}{r\_3} = \frac{(1 + w.p)}{(d\_\max + w.p)}. \]

This can be a significant factor for large values of d\_max. Thus, the net-
work will transfer data very inefficiently for vectors with large skew. 
However, for smaller values of d\_max such that \((d\_\max - 1) \ll w.p\), the 
factor \(\frac{r\_1}{r\_3} \approx 1\) and the system performance is the same as that for normally 
interleaved vectors.
Data Transfer

The transfer of data packets to the processors is controlled by means of data control lines from the processors to the control interface. The control interface translates the data requests from the processors to the appropriate sequence of variable select lines which are required to transfer the data elements from the memory. This was described in detail in the previous section. It is desirable that the operation of the processors be independent and asynchronous with respect to each other. This allows the processors to operate on their own clocks thus avoiding any clock skew problems. In order to ensure the independent and asynchronous mode of operation for each module, interlocked communication is used for data transfer.

The control signals for the control loops are shown in Fig. 4.21.

The data control lines are:

\( i \) DRQ - Data Request line; this line is used by the processors to request more data.

\( ii \) DAV - Data Available line; this line is used by the Control Interface to signal that a data word is available on the input bus.

\( iii \) DFIN - Data Finished line; this line is used by the Control Interface to signal that a complete data packet has been transferred.

\( iv \) DRY - Data Ready line; this line is used by a processor to indicate that it has a data word on the output bus ready to be transferred.
FIG. 4.21 DATA CONTROL SIGNALS

PIU = Processor Interface Unit

To Memory
v) DAC - Data Accepted line; this line is used by the Control Interface to indicate that it has transferred a word from the output bus to the memory.

First, consider the transfer of input packets. Let the I-bit of the input control loop for processors be at processor \( P_i \). The processor, \( P_i \), having the input control bit is allowed to initiate data transfer. If the processor is not busy in the processing state, then it indicates that it is free to receive a data packet by asserting the data request line, DRQ. Note that no other processor is allowed to assert DRQ since it doesn't have the control bit. The control interface, on reception of DRQ, initiates signals to the memory to fetch the required data elements for the next data packet, Fig. 4.22. Once these elements (one of these, if the bus width is one word) are available on the input bus, the Control Interface asserts DAV. This indicates, to the processors, that the data is available on the bus. \( P_i \) then transfers the data to its buffers from the bus. Only \( P_i \) is allowed to transfer the data because it has the input control bit. When the data transfer is complete the processor \( P_i \) negates DRQ. The Control Interface then negates DAV. This completes the transfer of b data elements if the width of the bus is b words. In all subsequent discussion the bus width will be taken as one word. This allows low hardware cost and a simpler interface design.

Having accepted one data element, the processor \( P_i \) will assert DRQ again. The control interface fetches the next data element and asserts DAV. Processor \( P_i \) transfers the word from the bus to its internal buffer and lowers DRQ. The control interface lowers DAV. This sequence of signalling is executed for the transfer of each data element. When the
FIG. 4.22 Data Control Signals (Behaviour)
complete data packet has been transferred, the control interface asserts DFIN instead of DAV (after the processor asserts DRQ). This indicates to the processor that the data packet has been transferred completely. The processor negates DRQ and shifts the control bit to the next processor, \( P_{i+1} \). The control interface then negates DFIN, and waits for a request from processor \( P_{i+1} \). The assertion of DFIN is also used by the processor to switch to execution mode. Note that the control interface keeps track of the number of data elements per data packet. Thus, the processor modules do not need any counting hardware.

The transfer of results from the processors to the memory is done by signalling on DRY and DAC. A processor, \( P_i \), can initiate output transfer if it has the output control bit. The data transfer is initiated by putting the word on the output bus and asserting DRY. This indicates to the control interface that a processor has put a word on the bus. The control interface then generates signals to transfer the word from the bus to a selected memory buffer. Once this transfer is completed the interface signals the acceptance of data by asserting DAC. The processor, on observing the assertion of DAC, will negate DRY. The control interface will in turn negate DAC. This completes the transfer of one word. This signalling sequence is repeated for each word that is transferred. When all the words of a result packet are transferred, the processor does not assert DRY but shifts the output control bit to the next processor, \( P_{i+1} \). There is no explicit signalling required to finish the data transfer in this case because a processor already knows how many data elements it has to send to the memory. The control interface has counting hardware which indicates how many data elements to expect. This count is
set initially at the time of sending the command packets to the processors. So after the transfer of the correct number of results the counter resets itself.

The signalling sequence ensures that the transfer of the next word is not initiated until the transfer of the current word is completed. This allows asynchronous data transfer since the signalling sequence requires no fixed time of operation or a common clock.
4-6 Processor Interface Unit

The processor control loops are implemented in the processor interface units. The function of an interface unit is to gate the data control signals to and from the processor for data transfer on a bus, whenever the interface unit has the control bit for that bus. The processor communicates with the interface on five control lines as shown in Fig. 4.23. The control lines are

i) PDRQ - processor data request. This signal informs the processor that it has the control bit and the signal is used as an inquiry for output from the processor.

ii) PDAV - processor data available. This signal is used by the processor to indicate that it has put an output word (a result) on the output bus. The interface translates this to the signal DRY.

iii) PDFIN - processor data finished. This is a signal from the processor to the interface to indicate that it has no more results to output and that the control bit should be shifted to the next processor.

iv) PDRY - processor data ready. This is a signal from the PIU to the processor to inform the processor that an input data word is available on the input data bus. This signal is essentially DAV from the Control Interface.

v) PDAC - processor data accepted. This is a signal from the processor to its PIU to indicate that it has
FIG. 4,23  PROCESSOR SIGNALS
accepted the data word on the input data bus and is ready for more. This signal is translated to DRQ by the PIU.

The PIU consists of two parts, one for controlling the transfer of input packets (IPIU) and the other part for controlling the transfer of output packets (OPIU) as shown in Fig. 4.24. Both the parts control transfer of data words except in opposite directions. Since the two parts perform the same functions, the same hardware is used for implementing them. Fig. 4.25 illustrates the implementation for the input part of PIU. The design is implemented using level logic and is clock independent. The lines ACTI and ACT0 form part of the control loop. The control bit is represented by the difference in the status of ACTI and ACT0. Thus, in order to shift the control bit to the next processor it is only necessary to change the status of the ACT0 line.

Input

When the IPIU receives a control bit, that is, there is change in the voltage level on ACTI line, it causes the IPIU to send out a request for data from the memory by asserting DRQ; a change on ACTI enables gates G1 and G3 thus asserting DRQ. When a word is available on the input bus the control interface asserts DAV, as explained earlier. This signal is received by every PIU. However, only the IPIU with the input control bit has gate G4 enabled thus asserting PRY for that processor. The processor transfers the word from the input bus to internal memory after it is through with the processing of any previous operation. The word is held on the data bus until the processor transfers it. Once the transfer of
FIG. 4.25  IPIU
of the word is complete the processor asserts PAC. This negates DRQ. The control interface in turn negates DAV, which negates PRY. Then, the processor negates PAC. This completes the transfer of a word. The IPIU now generates the request for another word by asserting DRQ; negation of PAC enables gate $G_4$ again thus asserting DRQ. The same sequence of signals is observed for the transfer of the next word and so on.

When the complete set of input words have been transferred, the control interface asserts DFIN instead of DAV, after receiving an assertion of DRQ to be negated. Note that no signal is sent to the processor. Where the control interface receives the negation of DRQ, it lowers DFIN. That changes the state of latch B, thus changing the status of ACTO. The status (voltage level) of ACTO matches that of ACTI, thus lowering gate $G_1$ hence all the remaining gates. The change in level on ACTO acts as the shift of the control bit to the next processor since the same line is ACTI for the next processor.

**Output**

The operation of the output processor interface unit (OPIU) is similar to that of IPIU only the direction of the signals is reversed. The implementation of OPIU is shown in Fig. 4.26. The circuit is identical to that used for IPIU. The lines DRY and DAC now serve as the equivalent signals PRY and PAC of IPIU. Similarly, PRQ, PAV and PFIN are employed in OPIU are equivalent to DRQ, DAV and DFIN of the IPIU. The transfer of output words requires the same signalling as for input word except the words are moved from the processor to the memory.
CHAPTER V
Applications

5-1 Introduction

In the preceding chapters it was shown that the Stream Processing System is a very efficient vector processor. It was shown in Chapter I and II that the system has a performance comparable to an equivalent pipeline or any other equivalent parallel processor configuration for vector operations. The vector operations under consideration have been n-ary operations which comprise a set of independent n-ary scalar operations. Examples of such operations are the Primitive Scalar Dyadic functions and the Primitive Scalar Monad functions of APL [GREY 73]. Specific examples of such operations are

Addition, \( A+B \), \( A \) and \( B \) are vectors,

Subtraction, \( A-B \),

Multiplication, \( A\times B \),

Division, \( A\div B \),

Maximum of \( A \) and \( B \), \( A\|B \),

Minimum of \( A \) and \( B \), \( A\&B \),

Base-A logarithm of \( B \), \( A@B \),

comparison, logical and other such operations. Note that all these operations are vector operations of the type \( A \ 'op' \ B \) which consist of the scalar operations, \( a_1 'op' b_1 \), \( a_2 'op' b_2 \), \ldots, \( a_z 'op' b_z \), where the scalar operations are independent of each other and can be executed in parallel.

Thus, if any algorithm can be broken into n-ary vector operations
of the types described above, then the algorithm can be efficiently implemented on the Stream Processing System. In this chapter some special parallel algorithms are presented and it is shown that they can be efficiently implemented on the Stream Processing Unit. The algorithms discussed in this chapter are for the solution of fast Fourier transform, polynomial evaluation, matrix transpose and linear recurrence problems. The algorithms generally require N processors (where N is the number of input data points) and some special interconnections like the Perfect Shuffle interconnection or the Inverse Shuffle interconnection to rearrange the data after a computation by the N processors. The significance of the algorithms discussed here is that they are highly parallel and, given N processors capable of operation in parallel, the solutions (to the problem at hand) can be obtained in $O(Lg_2N)$ time.

It is first shown in section 5-2 that the permutation of data elements obtained from the Perfect Shuffle interconnection is achieved in the Stream Processing System as a binary vector operation. In the case of fast Fourier transform this permutation requires no additional time; the permutation being performed during the time of transfer of data elements to the processors. Also, this approach allows the system to implement the Perfect Shuffle Interconnection for different values of N (N being a power of 2) without requiring any changes in the system. In section 5-3, the implementation of a parallel algorithm for evaluating a polynomial of degree N is described. The algorithm applies for values of N that are a power of 2 and requires the Perfect Shuffle interconnection. It is shown that this algorithm can be implemented as a series of vector operations on the Stream Processing System. Then,
the algorithm is extended to all values of N. The extension of the algorithm does not require a Perfect Shuffle interconnection.

In section 5-4, an implementation of the fast Fourier transform (FFT) is presented. It is shown that FFT algorithms can be implemented as a series of vector operations and that the execution time is close to O(Lg₂N). This algorithm also uses the Perfect Shuffle interconnection. Section 5-5 gives an algorithm for matrix transpose using the Perfect Shuffle interconnection. Section 5-6 discusses the class of problems which fall under the category of Linear Recurrences. The implementation of a parallel algorithm for the solution of linear recurrence problems requires an 'Inverse Shuffle' interconnection. It is shown that the 'Inverse Shuffle' interconnection is achieved in a manner similar to that for the Perfect Shuffle interconnection, that is, as a binary vector operation. This allows us to implement the parallel algorithm very efficiently.

Thus this chapter demonstrates that the Stream Processing System not only is a very effective vector processor but can also be used very efficiently in many special applications.
The perfect shuffle is an interconnection pattern which re-arranges the elements of a vector as shown in Fig. 5.1. The vector is split in two halves and recombined such that in the final vector the elements from the two halves alternate. If the indices on the left in Fig. 5.1 are mapped onto the indices on the right according to permutation P, then

\[ P(i) = 2i \quad 0 \leq i \leq \frac{N}{2} - 1 \]

\[ = 2i + 1-N \quad \frac{N}{2} \leq i \leq N-1, \text{ where} \]

\[ N = 2^m, \text{ m is an integer. This interconnection pattern has been shown to be very useful in implementing many algorithms on parallel processors} \]

[Pease 68], [Stone 71]. We will first show how the perfect shuffle can be performed in a stream processing system. Then the implementation of some important algorithms using the perfect shuffle is presented.

The perfect shuffle is implemented by splitting the original vector in two halves, then pairing the elements of the two halves and storing the resultant pairs in sequence in the result vector. With respect to Fig. 5.1 the two halves of the vector are \(<0, 1, \ldots, \frac{N}{2}-1>\) and \(<\frac{N}{2}, \frac{N}{2}+1, \ldots, N-1>\). These packets are not processed but simply sent back as the results. If all the elements are stored in the same vector then the resultant vector is \(<0, \frac{N}{2}, 1, \frac{N}{2}+1, \ldots, (\frac{N}{2}-1), (N-1)>\) which is the shuffled vector, as shown in Fig. 5.1. Since there is no processing done on the input packets during the shuffle operation, it is not necessary to send the input packets to the various processors. The packets have merely to be transferred from the input data bus to the output data bus through the Control Interface.
FIG. 5.1  PERFECT SHUFFLE INTERCONNECTIO

In terms of the control loops, one input control loop is initially set to the beginning of the given vector and another input control loop is set to the halfway point in the vector. The two halves of the vector are accessed independently and in parallel as two independent vectors. The results are stored using only one control loop. It has to be shifted after storing every element.

The vector operation has the length N/2 and a total of three variables. From E 3.3 the time for shuffle is given by T, such that

\[ T = 3M_c + N/2 \cdot r_1 + R_0 + r_1, \]

where \( r_1 \) = time to transfer an input or output packet (each packet has two elements).

\( R_0 \) = processing time = 0

\( M_c \) = memory cycle time

so

\[ T = 3M_c + N/2 r_1 + r_1 \]

\[ \approx 3M_c + N/2 r_1. \ldots E 5.1. \]

In some of the algorithms (FFT and SORT) it is necessary to operate on pairs of adjacent elements of the shuffled vector. The outputs of the operation are used as inputs for the next pass of operations. That is, they are again shuffled and operated upon in pairs and so on. An implementation of such algorithms using N/2 processors is illustrated in Fig. 5.2. The FFT algorithm using such an implementation is discussed in section 5.4.

The perfect shuffle with operations on pairs is accomplished in the stream processor by allowing the paired packets \( <0, N/2>, <1, N/2+1>, \ldots \).
FIG. 5.2 PERFECT SHUFFLE IMPLEMENTATION
\[ \ldots, <N/2-1, N-1> \] to be sent to the processors. The operations are performed in the processors. The results to the packets are obtained in the same sequence as that of the input packets. The results are stored as a single vector which is used as input for the next pass. Thus the total time is dictated by processor operations and there is no additional overhead for the shuffle in this case.

Thus we have seen that the control loops allow us to easily accomplish the perfect shuffle as a binary vector operation. Also in the cases where the perfect shuffle is followed by some operation on pairs of adjacent elements, the shuffle introduces no overhead. This is because the vectors have to be paired in any case to be sent to the processors for some operations. The shuffle is accomplished by pairing the halves of a vector in a certain fashion thus requiring no additional time.

The major advantage of the stream processing system is that the same algorithm can be used to perform a perfect shuffle for different values of \( N \) as long as \( N \) is a power of two. There is no reorganisation of processors or memories as required in use of hardwired shuffle interconnections.
5-3 Polynomial Evaluation

The natural lower bound on the minimum time for evaluation of $x^N$ is $\lceil \lg_2 N \rceil$ multiplication times, where $x$ is some real number and $N$ is an integer. The lower bound arises because it takes at least $\lg_2 N$ successive squarings of $x$ to compute $x^N$ when $N$ is a power of 2 [Stone 71]. The evaluation of a polynomial of degree $N$ requires computation of terms $x^2, x^3, \ldots, x^N$. Stone has shown that there exists an algorithm with a complexity proportional to that of the lower bound, $(\lceil \lg_2 N \rceil)$, for evaluating a polynomial of degree $N$ [Stone 71]. This algorithm has to be executed on a parallel processor with the perfect shuffle interconnection pattern in order to achieve the desired computational speed. We will first describe Stone's algorithm and show how it can be efficiently implemented on the stream processor. Next, we will present an extension of the algorithm which does not use the shuffle interconnection pattern and can be used for any value of $N$. Stone's algorithm requires that $N$ be a power of 2.

A. Consider a polynomial of degree $(N-2)$ with the coefficients $a_0, a_1, a_2, \ldots, a_{N-2}$ where $N$ is a power of 2. The value of the polynomial is $\sum_i a_i x^i$. We assume that the evaluation is performed on a parallel processor that is capable of executing $N$ operations simultaneously and has the perfect shuffle interconnection pattern. The following algorithm computes all the terms $a_i x^i$ in $\lg_2 N$ passes through the processor.

The algorithm is presented schematically in Fig. 5.3 for the case $N=8$. The coefficients $a_i, i=0, 1, \ldots, N-2$, are arranged in consecutive locations in a block of memory. From the memory the coefficients are transferred to registers in the processing units. A single number, $x$,
<table>
<thead>
<tr>
<th>Data</th>
<th>Mask</th>
<th>Data</th>
<th>Mask</th>
<th>Data</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>0</td>
<td>a₀</td>
<td>0</td>
<td>a₀</td>
<td>0</td>
</tr>
<tr>
<td>a₁</td>
<td>1</td>
<td>aₓ</td>
<td>0</td>
<td>aₓ</td>
<td>0</td>
</tr>
<tr>
<td>a₂</td>
<td>0</td>
<td>a₂</td>
<td>1</td>
<td>aₓ²</td>
<td>0</td>
</tr>
<tr>
<td>a₃</td>
<td>1</td>
<td>aₓ</td>
<td>1</td>
<td>aₓ³</td>
<td>0</td>
</tr>
<tr>
<td>a₄</td>
<td>0</td>
<td>a₄</td>
<td>0</td>
<td>aₓ⁴</td>
<td>0</td>
</tr>
<tr>
<td>a₅</td>
<td>1</td>
<td>aₓ</td>
<td>0</td>
<td>aₓ⁵</td>
<td>0</td>
</tr>
<tr>
<td>a₆</td>
<td>0</td>
<td>a₆</td>
<td>1</td>
<td>aₓ²</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x²</td>
<td>1</td>
<td>x⁴</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 5.3
is broadcast to all the processing units and a multiplication is performed at each unit. The current contents of a data register are replaced by the product of the current contents and the broadcast datum. It is also assumed that there is a masking capability in that the computation only takes place in those units which are identified by a '1' in the corresponding position in a mask register. If a mask bit is '0', the contents of the data register are left unchanged during an operation. Finally, the numeric value of x is sent as initial data to the Nth processing unit.

In Fig. 5.3, for the first operation the value of x is broadcast and the setting of the mask register causes every alternate unit to perform a multiplication. In the last unit the value computed is $x^2$. This value is loaded into the broadcast register for the next operation. The mask registers are updated as shown in the figure and the operation is repeated. Again the last unit contains the value $x^8$ which is put in the broadcast register for the next operation. The mask registers are updated and the next operation is performed. At the end of the third \([(\log_2 N) \text{ th}]\) operation all the terms of the polynomial have been computed and it is only necessary to take their sum

$$\sum_{i=0}^{6} a_i x^i$$

in one processor.

The number of passes required are $\log_2 N$ with summation at the end. The shuffle interconnection pattern is used to eliminate the need to store the mask vectors for all the passes thus saving storage space as well as
memory access time. Only the first mask vector \( v = (0, 1, 0, 1, \ldots, 0, 1) \) is stored and accessed. It can be sure that the next mask vector is obtained by shuffling \( v \). Again the third mask vector is obtained by shuffling the second mask vector and so on. The proof of this algorithm is given in [Stone 71].

This algorithm can be easily executed on the stream processor. The operation performed in each unit involves three operands, \( a_i \), mask, and the broadcast datum. In case of the stream processor, it is necessary to generate \( N \) packets with three elements in each packet and to stream them to the processors. In order to do that, the coefficients are stored as a vector in the memory. The numeric value of \( x \) is concatenated at the end of the coefficient vector. The first mask vector is also stored in the memory. The operation is regarded as a three-vector operation. A control loop is set to the start of the coefficient vector and another to the start of the mask vector. A third control loop points to the last element of the coefficient vector. This control loop is not shifted after the formation of each packet. This ensures that the same word is sent as the third element in each packet thus essentially accomplishing a broadcast of that word. Thus \( N \) packets are formed in sequence and sent to the processing units. The results form the new values in the coefficient vector. Each pass in Fig. 5 is now a single vector operation. At the end of the vector operation the mask vector is shuffled by splitting the vector in two halves, pairing and storing back the words, as explained in section 5.2.

The total number of passes, hence vector operations, is \( \log_2 N \) with
((\log_2 N)-1) shuffles. At the end it is required to sum the elements of the coefficient vector.

Each pass is a vector operation with three input vectors and one output vector. The length of the vector operation is \( N \). The time taken for each vector operation from E 3.3 is given by \( T_1 \) such that

\[
T_1 = 4M_C + (Nr_2 + R + r_2)
\]

where \( M_C \) = memory cycle time

\( R \) = processing time per processor

\( r_2 \) = packet transfer time. It is assumed here that the stream processor has enough processors to maintain the maximum data rate. That is, \( N_p \geq \frac{R + r_2}{r_2} \), where \( N_p \) = the number of processors.

The time taken for a shuffle, from E 5.1 is given by \( T_2 \), such that

\[
T_2 = \frac{rN}{2} + 3M_C.
\]

Then the total time for \( \log_2 N \) passes is given by \( T_3 \), such that

\[
T_3 = (\log_2 N)T_1 + [(\log_2 N) - 1]T_2
= (\log_2 N) \cdot [4M_C + Nr_2 + R + r_2] + [(\log_2 N) - 1] \left[ (3M_C + \frac{N}{2} \cdot r_1) \right]
= (\log_2 N)[4M_C + 3M_C + (R + r_2)] + N\log_2 N \left[ r_2 + \frac{r_1}{2} \right]
- \left[ 4M_C + \frac{3N}{2} \right].
\]

- E 5.2

If \( (Nr_2 + \frac{1}{2}) \ll 7M_C + R + r_2 \), then the time \( T_3 \) is \( O(\log N) \) as desired.

However, for large \( N \), there is a term, \( N\log_2 N \left[ r_2 + \frac{1}{2} \right] \), which increases as \( N\log_2 N \) and not as \( \log_2 N \).
B. It can be seen that Stone's algorithm requires the perfect shuffle interconnection only for generating the new masks. In the stream processor it results in additional delay, as shown by E 5.2. It is more efficient if each processing unit can generate the new mask while it generates the new coefficient value. Then, the new mask values are obtained as a result vector and it is no longer necessary to shuffle the mask vectors between the operations. Such an algorithm is presented below.

In Fig. 5.3 if all the mask values for each coefficient are put together in one word, the vector of these combined words is \((000, 100, 010, 110, 001, 101, 011, 111)\). The decimal equivalent of these words is \((0, 4, 2, 6, 1, 5, 3, 7)\) which is the reverse binary ordering of \((0, 1, 2, 3, 4, 5, 6, 7)\). Thus the correct mask bit for the \(i\)th coefficient for the \(j\)th pass can be obtained by taking the \(j\)th bit from the reverse binary representation of the number \(i\). For example, the mask bit for the second pass for the sixth coefficient is the second bit of 011 which is 1. So complete mask words are stored in the initial mask vector instead of mask bits. For the example of Fig. 5.3 the mask vector is initially \((0, 1, 2, 3, \ldots, 7)\). Each processor reads the mask word and takes the least significant bit as the mask bit for that pass. The least significant bit of the binary representation of \(i\) is the same as the first bit of the reverse binary representation. The mask word is then shifted right by one and sent as a result. This puts the next to least significant bit in the least significant position. So the next time around the correct mask bit is obtained. The algorithm is shown schematically
in Fig. 5.4. The underlined bits indicate the mask bits for that pass. The reason for modifying the mask words and sending them back as a result vector is that the operations on a particular coefficient may be performed by different processors in successive passes. The mask words are associated with the coefficients and not the processing units. Each pass is treated as a distinct vector operation of length N.

The algorithm can be stated as follows:

i) Initially the coefficients $a_0$ to $a_{N-2}$ are stored as a vector in the memory. The value of $x$ is concatenated as the $N$th element of the coefficient vector.

ii) The initial mask vector is $(0, 1, 2, \ldots, N-1)$.

iii) $N$ packets from the coefficient vector, mask vector and the broadcast datum are formed and sent to the processors in sequence, as before.

iv) The processors compute the modified coefficients and the new mask vectors.

v) $N$ modified coefficients and $N$ new mask words are sent to the memory as results.

vi) The results are stored as the new coefficient vector and the new mask vector.

vii) Steps iii) to vi) are repeated $(\lg_2 N) - 1$ times.

viii) The first $(N-1)$ elements of the final coefficient vector are summed to give the value of the polynomial.

The operations performed by each processor on an input packet are:

a) Obtain the mask bit from the mask word (select the least significant bit),

b) Perform a multiplication between the current coefficient and the
### Shift Right by One Bit

<table>
<thead>
<tr>
<th>Data</th>
<th>Mask</th>
<th>Data</th>
<th>Mask</th>
<th>Data</th>
<th>Mask</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>000</td>
<td>$a_0$</td>
<td>000</td>
<td>$a_0$</td>
<td>000</td>
<td>$a_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>001</td>
<td>$a_1$</td>
<td>000</td>
<td>$a_1$</td>
<td>000</td>
<td>$a_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>010</td>
<td>$a_2$</td>
<td>001</td>
<td>$a_2$</td>
<td>000</td>
<td>$a_2$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>011</td>
<td>$a_3$</td>
<td>001</td>
<td>$a_3$</td>
<td>000</td>
<td>$a_3$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>100</td>
<td>$a_4$</td>
<td>010</td>
<td>$a_4$</td>
<td>001</td>
<td>$a_4$</td>
</tr>
<tr>
<td>$a_5$</td>
<td>101</td>
<td>$a_5$</td>
<td>010</td>
<td>$a_5$</td>
<td>001</td>
<td>$a_5$</td>
</tr>
<tr>
<td>$a_6$</td>
<td>110</td>
<td>$a_6$</td>
<td>011</td>
<td>$a_6$</td>
<td>001</td>
<td>$a_6$</td>
</tr>
<tr>
<td>$x$</td>
<td>111</td>
<td>$x^2$</td>
<td>011</td>
<td>$x^4$</td>
<td>001</td>
<td>$x^8$</td>
</tr>
</tbody>
</table>

---

1st Pass 2nd Pass 3rd Pass Final Output Data

---

Fig. 5.4
broadcast datum to produce the new coefficient, if the mask bit is one. Else the new coefficient is the same as the current coefficient,
c) generate the new mask word by shifting the current mask word to the right by one bit.

The time taken for shuffles in Stone's algorithm is saved in this case. However, the amount of processing per processor is increased slightly because of steps a) and b) mentioned above. Generally the time taken to perform a shift or a mask operation is much smaller than that for a multiplication. Therefore, the processing time is taken to be the same as earlier. The number of output variables is increased from one to two since the mask words are outputted in this case. The total time for \( \lg N \) passes is then given by \( T_4 \), such that \( T_4 = \lg N \left[ 5M_c + \frac{N_r}{2} + R + r_2 \right] \) from E 5.2. The time for shuffles has been eliminated and the initial memory delay is \( 5M_c \) because there are 5 variables.

That is, \( T_4 = \lg N \left[ 5M_c + R + r_2 \right] + N \lg N \cdot r_2 \).

- E 5.3

This is a much better performance than obtained by directly using Stone's algorithm on the stream processor. In the above algorithm it is no longer necessary to perform the shuffle, hence it is not necessary that \( N \) be a power of 2. The algorithm will work correctly for any value of \( N \). Indeed a polynomial of degree \( N \) can be evaluated in time close to \( O(\lg^2 N) \) for small \( N \). The number of passes required for evaluation of individual terms of the polynomial is \( \lceil \lg N \rceil \). However, there is one modification required, that is, the last term of the mask vector should
have all '1's. Thus the mask vector will be \((0, 1, 2, \ldots, (N-3), (N-2), 2^m-1)\) where \(m = \lceil \log_2 N \rceil\). For example consider a polynomial of degree nine, as shown in Fig. 5.5. The mask vector is \((0, 1, 2, \ldots, 7, 8, 9, 15)\). Since the number of passes is equal to \(\lceil \log_2 N \rceil\), the total time taken for \(\lceil \log_2 N \rceil\) passes is given by

\[
T_4 = \left[5M_C + (N) r_2 + R + r_2 \right] \times \lceil \log_2 N \rceil.
\]

- E 5.4

Note that the vector length is now \((N+1)\).

The correctness of the sequence of operations can be shown as follows. The mask word associated with coefficient \(a_i\) is the index value \(i\). The algorithm is correct if and only if \(a_i\) is multiplied by \(x^i\).

Consider the binary representation of index \(i\),

\[
i = i_0 + i_1 \cdot 2^1 + i_2 \cdot 2^2 + \ldots + i_{m-1} \cdot 2^{m-1}
\]

where \(m = \lceil \log_2 N \rceil\). In Fig. 5.5 the \(j\)th coefficient \(a_{i_j}\), is multiplied by \(x^{2^j}\), \(j = 0, 1, \ldots, m-1\), if and only if \(i_j = 1\). That is, the \(i\)th coefficient after all the passes are completed is ultimately multiplied by

\[
x^{i_1 \cdot 2^1} \cdot x^{i_2 \cdot 2^2} \ldots \cdot x^{i_{m-1} \cdot 2^{m-1}} = x^{\sum j i_j 2^j} = x^i,
\]

as desired.
<table>
<thead>
<tr>
<th></th>
<th>1st pass</th>
<th>2nd pass</th>
<th>3rd pass</th>
<th>4th pass</th>
<th>Final Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Mask</td>
<td>Data</td>
<td>Mask</td>
<td>Data</td>
<td>Mask</td>
</tr>
<tr>
<td>( a_0 )</td>
<td>0000</td>
<td>( a_0 )</td>
<td>0000</td>
<td>( a_0 )</td>
<td>0000</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>0001</td>
<td>( a_1 )</td>
<td>0000</td>
<td>( a_1 )</td>
<td>0000</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>0010</td>
<td>( a_2 )</td>
<td>0001</td>
<td>( a_2 )</td>
<td>0000</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>0011</td>
<td>( a_3 )</td>
<td>0001</td>
<td>( a_3 )</td>
<td>0000</td>
</tr>
<tr>
<td>( a_4 )</td>
<td>0100</td>
<td>( a_4 )</td>
<td>0100</td>
<td>( a_4 )</td>
<td>0001</td>
</tr>
<tr>
<td>( a_5 )</td>
<td>0101</td>
<td>( a_5 )</td>
<td>0100</td>
<td>( a_5 )</td>
<td>0001</td>
</tr>
<tr>
<td>( a_6 )</td>
<td>0110</td>
<td>( a_6 )</td>
<td>0111</td>
<td>( a_6 )</td>
<td>0001</td>
</tr>
<tr>
<td>( a_7 )</td>
<td>0111</td>
<td>( a_7 )</td>
<td>0111</td>
<td>( a_7 )</td>
<td>0001</td>
</tr>
<tr>
<td>( a_8 )</td>
<td>1000</td>
<td>( a_8 )</td>
<td>0100</td>
<td>( a_8 )</td>
<td>0010</td>
</tr>
<tr>
<td>( a_9 )</td>
<td>1001</td>
<td>( a_9 )</td>
<td>0100</td>
<td>( a_9 )</td>
<td>0010</td>
</tr>
<tr>
<td>( x )</td>
<td>( x^2 )</td>
<td>( x^2 )</td>
<td>( x^4 )</td>
<td>( x^4 )</td>
<td>( x^8 )</td>
</tr>
</tbody>
</table>

Fig. 5.5
5-4 Fast Fourier Transform

The Fourier transformation is one of the most important mathematical aids to signal processing. For many reasons we should like to be able to compute Fourier transforms with digital machines. In order to accomplish this the time function \( f(T) \) is represented by the sequence of \( N \) samples \( f(nT), \ 0 \leq n \leq N-1 \), where \( T \) is the sampling interval in the time domain. Similarly, the frequency spectrum \( F(\omega) \) is represented by \( F(k\Omega), \ 0 \leq k \leq N-1 \), where \( \Omega \) is the chosen increment between samples in the frequency domain. The discrete Fourier transform, abbreviated DFT, of a sequence of \( N \) samples, \( f(nT), \ 0 \leq n \leq N-1 \), is defined as the sequence

\[
F(k\Omega) = \sum_{n=0}^{N-1} f(nT)e^{-j\Omega nk}, \quad k = 0, 1, \ldots, N-1,
\]

-- E 5.5

where \( \Omega = \frac{2\pi}{NT} \), [GOLD 69].

Unless otherwise specified, it is assumed that both \( f(nT) \) and \( F(k\Omega) \) are complex numbers. The DFT has properties parallel to those of the continuous Fourier Transform. A detailed discussion on the DFT, its properties, and its uses can be found in [GOLD 69] and [RABINER 75]. In many signal processing applications it is necessary (or advantageous) to have fast algorithm as well as fast hardware to evaluate the DFT of a time function. The direct evaluation of E 5.5 requires \( N^2 \) complex multiplications and additions, and for moderately large \( N \), say \( N \) greater than 1000, this direct evaluation is rather costly in computer time. In order to reduce this com-
putation time several special algorithms have been developed. Of major
interest are the algorithms called the fast Fourier transforms, abbrevi-
ated FFT \cite{COOLEY1965,COCHRAN1967} and \cite{COOLEY1967}. Although these algorithms do not apply when the number
of points in a sequence is a prime, the savings when \( N \) is highly com-
posite (has many factors) can be very significant. Where \( N \) is a power
of 2, the fast Fourier transform algorithms require a number of compu-
tations proportional not to \( N^2 \) but to \( N \log_2 N \). Thus, for an example
\( N = 1024 \), this is a computational saving of 99 percent. The
discussion in this section applies only to the case when the number of
samples is a power of two.

There are two classes of fast-Fourier-transform algorithms, and
each has many modifications. The two approaches are

\begin{enumerate}
\item decimation in time and
\item decimation in frequency.
\end{enumerate}

We will elaborate on 'decimation in frequency' because this method can
be implemented efficiently on the stream processing systems discussed
in the earlier chapters.

**Decimation in frequency:** This form of the algorithm was found indepen-
dently by Sande, Cooley, Stockham and others \cite{GOLD1969}.

Consider \( N \) samples of a function \( f(T) \), where \( N \) is divisible by 2. The
sample points are represented by the function \( f_k \), where \( k = 0, 1, \ldots, (N-1) \). From E 5.1 the \( N \)-point DFT, \( F_k \), is

\[
F_k = \sum_{k=0}^{N-1} f_k \omega^{nk} \quad \text{for } k = 0, 1, \ldots, (N-1),
\]
where \( \omega = e^{-j(2\pi/N)} \).

The function \( f_\ell \) can be separated into two sequences of \( N/2 \) points each, say \( g_\ell \) and \( h_\ell \), where \( g_\ell \) is composed of the first \( N/2 \) points of \( f_\ell \) and \( h_\ell \) is composed of the last \( N/2 \) points of \( f_\ell \). That is,

\[
g_\ell = f_\ell \quad \text{for } \ell = 0, 1, \ldots, N/2 - 1
\]
\[
h_\ell = f_\ell + N/2
\]

The \( N \)-point DFT, \( F_k \), may now be written in terms of \( g_\ell \) and \( h_\ell \).

\[
F_k = \sum_{\ell=0}^{N/2-1} \left[ g_\ell \omega^{\ell k} + h_\ell \omega^{(\ell+N/2)k} \right] \quad k = 0, 1, \ldots, (N-1)
\]

or

\[
F_k = \sum_{\ell=0}^{N/2-1} \left[ g_\ell + e^{-j\pi k} h_\ell \right] \omega^{\ell k}.
\]

-- E 5.2

We now consider the even- and odd-numbered points of \( F_k \) separately (therefore, the name decimation in frequency). Replacing \( k \) by \( 2k \) in E 5.2 we get

\[
F_{2k} = \sum_{\ell=0}^{N/2-1} (g_\ell + h_\ell) (\omega^2)^{\ell k}, \quad \text{and replacing } k \text{ by } 2k + 1 \text{ in E 5.2 we get}
\]

\[
F_{2k+1} = \sum_{\ell=0}^{N/2-1} \left[ (g_\ell - h_\ell) \omega^{\ell} \right] (\omega^2)^{\ell k}, \quad k = 0, 1, \ldots, N/2-1,
\]

which can be recognized as the \( (N/2) \)-point DFTs of the functions \( (g_\ell + h_\ell) \) and \( (g_\ell - h_\ell) \omega^{\ell} \). Thus all \( N \)-point DFT can be expressed in terms
of two computations of \((N/2)\)-point DFTs. Fig. 5.6 illustrates the reduction for \(N=8\). If \(N/2\) is even then an \((N/2)\)-point DFT can be reduced to two computations of \((N/4)\) point DFTs, and so on, until the algorithm is finally reduced to computation of 2-point DFTs \((N\) being a power of 2\)). This is illustrated in Fig. 5.7 for \(N=8\). The unmarked nodes represent the computation \((g_k + h_k)\) and the nodes with \(w_k\) associated with them represent the computation \((g_k - h_k)w_k\). The total amount of computation is proportional to \(N \lg_2 N\) which is the number of computation nodes in the graph. It is interesting to note that the signal flow graph has

i) \(\lg_2 N\) stages as shown in Fig. 5.7,

ii) \(N\) independent computation nodes per stage

iii) the outputs are in bit reversed order.

The \(N\) computations in every stage can be executed in parallel provided there are a sufficient number of functional units available. In order to use the same \(N\) functional units for every stage, it is necessary to reroute the data according to the interconnections shown in Fig. 5.7. It can be seen that the interconnections change from stage to stage.

The algorithm of Fig. 5.2 can be redrawn as shown in Fig. 5.8. Adjacent computation nodes in Fig. 5.8 can be combined into one functional unit since the nodes use the same two input values. Thus the algorithm can be redrawn as shown in Fig. 5.9. If should be noted that each functional unit performs two operations. If the inputs to a functional unit are \(g_k\) and \(h_k\), then the unmarked output corresponds to \((g_k + h_k)\) and the output marked with a coefficient \(w^k\) represents the result of \((g_k - h_k)w^k\). It should be noted that the interconnection pattern is the
Note: Both $W^r$ and $W^l$ are marked at a computation node to represent $(g_1W^r - h_1W^l)$.

FIG 5.6
FIG. 5.7 FAST FOURIER TRANSFORM ALGORITHM USING DIF.
FIG. 5.8  FAST FOURIER TRANSFORM (CONSTANT GEOMETRY)
same from stage to stage. This rearrangement is called the constant geometry form [GOLD 69]. The interconnection scheme between stages in Fig. 5.9 is the perfect shuffle. The algorithm can be implemented with only N/2 functional units and a perfect shuffle interconnection scheme as shown in Fig. 5.10. Thus the perfect shuffle interconnection pattern is sufficient for executing the transform algorithm on a parallel processor. This was first discovered by Pease [Pease 68].

In order to implement the FFT algorithm of Fig. 5.9 using the perfect shuffle scheme, as shown in Fig. 5.10, it is necessary to generate the correct set of "Twiddle" factors (terms involving powers of ω) for every stage of computation. The twiddle factors for N=8 are
\[
< \omega^0, \omega^1, \omega^2, \omega^3 > \quad \text{for the first stage},
\]
\[
< \omega^0, \omega^0, \omega^2, \omega^2 > \quad \text{for the second stage},
\]
and
\[
< \omega^0, \omega^0, \omega^0, \omega^0 > \quad \text{for the final stage}.
\]

In general, the twiddle factors for the jth stage are given by
\[
\left( \frac{i}{2^{(j-1)}} \right) \times 2^{(j-1)} \quad \text{for } i = 0, 1, \ldots, (N/2 - 1).
\]

In general, ω is a complex number. Therefore, the twiddle factors are assumed to be complex numbers when otherwise stated.

**Implementation using stream processors:** It was explained earlier that the perfect shuffle interconnection is accomplished in the stream processor by splitting the original vector into two halves and then pairing the two vectors. Since the FFT algorithm can be rearranged as shown in Fig. 5.9, it can be implemented using the perfect shuffle inter-
FIG. 5.9 FFT USING THE PERFECT SHUFFLE
FIG. 5.10  FFT IMPLEMENTATION
connection. The vector of initial coefficients is split in two halves and paired. In addition a twiddle factor is added to each pair. The resulting data packets are sent to the processors for computation. The results are stored as one vector. For example, consider the algorithm for N=8 as shown in Fig. 5.9. The original coefficient order is \( f_0, f_1, f_2, \ldots, f_7 \). It is split in two halves and paired to produce the packets \( < f_0, f_4 >, < f_1, f_5 >, < f_2, f_6 > \) and \( < f_3, f_7 > \). The twiddle factors to be added are \( < \omega^0, \omega^1, \omega^2, \omega^3 > \). The final packets are then \( < f_0, f_4, \omega^0 >, < f_1, f_5, \omega^1 >, < f_2, f_6, \omega^2 > \) and \( < f_3, f_7, \omega^3 > \). These packets are generated and sent in a sequence to the processors. The result packets are \( < f^1_0, f^1_1 >, < f^1_2, f^1_3 >, < f^1_4, f^1_5 > \) and \( < f^1_6, f^1_7 > \) which are stored in the memory as one vector, that is, \( < f^1_0, f^1_1, \ldots, f^1_7 > \). For the next stage of computation, this result vector is treated as the new input vector. It is split in halves and paired. The new twiddle factors, \( < \omega^0, \omega^0, \omega^2, \omega^2 > \), are added to the packets and the packets are sent to processors for the next stage of computation, and so on. Since the set of twiddle factors is different for each stage, the twiddle factors for each stage are stored in the memory as a distinct vector. This approach requires that \( \log_2 N \) twiddle vectors be generated and stored initially. This not only requires extra computation time in the beginning but also quite a large amount of storage.

If an N-point transform is being considered then each stage of the algorithm Fig. 5.8 is realized as a vector operation of length \( N/2 \).

The entire algorithm is completed by performing \( \log_2 N \) vector operations. The time taken for each operation given by \( T_S \), such that
$T_5 = (VM_c + (N/2)r_3 + R_2 + r_3)$ from E 3.3.

where $V =$ number of variables = 4

$r_3 = \text{time to transfer one data packet (three elements)}$

$R_2 = \text{processing time per processor,}$

and $N_r > (R_2 + r_3)/r_3$ .

The total time taken for the algorithm is then,

$$T_5 = \log_2 N \left[ wM_c + N \frac{r_3}{2} + R_2 + r_3 \right].$$

It was already noted that the scheme described above required a storage of $N/2 \log_2 N$ words for the twiddle factors. This can be significant for large $N$. We now present a scheme where it is necessary to store only $N/2$ twiddle factors. The twiddle factors for the next stage are computed in the processors during the current stage.

The initial vector of twiddle factors is $W = <\omega^0, \omega^1, \omega^2, \ldots, \omega^{N/2-1}, 0, 0, \ldots, 0>$. This vector contains the $N/2$ twiddle factors followed by $N/2$ empty locations. It is necessary to assign this space to store the modified twiddle factors generated by the processors. However, only the first $N/2$ values are used as input for every vector operation. The scheme is to square every twiddle factor after completing the 'butterfly' operation and send back two copies of the squared value. Thus if a processor receives the input packet $<g_k, h_k, \omega^r>$ the operations performed are

$$a) \quad g'_k = g_k + h_k,$$

$$b) \quad g'_{k+1} = (g_k - h_k)\omega^r$$

and $c) \quad \omega^r = (\omega^r)^2 = \omega^{2r}$.
The result packet contains \( g'_k, g'_{k+1}, \omega^{2\pi}, \omega^{2\pi} \). The new values of the twiddle factors are stored consecutively in the original vector \( W \). The twiddle factors in \( W \) are, now, \( \omega^0, \omega^0, \omega^2, \omega^2, \ldots, \omega^{N/2-2}, \omega^{N/2-2}, \ldots, \omega^{N-2}, \omega^{N-2} \). The last \( N/2 \) values are ignored. In the next vector operation, the first \( N/2 \) values of \( W \) are used. That is, the twiddle factors for the next operation are \( \omega^0, \omega^0, \omega^2, \omega^2, \ldots, \omega^{N/2-2}, \omega^{N/2-2} \) as desired by the algorithm, Fig. 5.8. Similarly, the next pass generates \( \omega^0, \omega^0, \omega^0, \omega^0, \omega^4, \omega^4, \omega^4, \ldots, \omega^{N/2-4}, \omega^{N/2-4}, \omega^{N/2-4}, \omega^{N/2-4} \), and so on.

We have thus avoided extra storage at the cost of increasing the processing time per processor. In order to get an idea of how much the processing time is increased, consider the three operations performed by each processor. All the three operations are in general complex operations.

If two coefficients are represented as \( g_k = (a + jb) \) and \( h_k = (c + jd) \) then the operations \( (g_k + h_k) \) and \( (g_k - h_k) \omega^r \) can be rewritten as

\[
\begin{align*}
  a' &= a + c \\
  b' &= b + d \\
  c' &= (a-c) \cos \frac{2\pi r}{N} + (b-d) \sin \frac{2\pi r}{N} \\
  d' &= (a-c) \sin \frac{2\pi r}{N} - (b-d) \cos \frac{2\pi r}{N}
\end{align*}
\]

where, \( g'_k = a' + jb' \),

\( g'_{k+1} = c' + jb' \), and

\( \omega^r = \cos \frac{2\pi r}{N} + j \sin \frac{2\pi r}{N} \).

That is a total of four multiplications and six additions (or subtrac-
Squaring $\omega^r$ involves computation of

$$\cos 2(\frac{2\pi r}{N}) \text{ and } \sin 2(\frac{2\pi r}{N}) \text{ because } \omega^{2r} = \cos 2(\frac{2\pi r}{N}) + j \sin 2(\frac{2\pi r}{N}).$$

Now,

$$\sin 2(\frac{2\pi r}{N}) = 2 \sin \frac{2\pi r}{N} \cos \frac{2\pi r}{N} \text{, and}$$

$$\cos 2(\frac{2\pi r}{N}) = 2 \cos^2 \frac{2\pi r}{N} - 1. \text{ These two computations require}$$

two multiplications and one subtraction. The multiplication by 2 can be accomplished by a left shift. Since multiplications are the major time consuming operations, the additional time required to compute $\omega^{2r}$ is 50% of time required for the 'butterfly' operations. This increase in processing time per processor is absorbed in the processing system by using more processors, such that

$$N_p \geq \frac{R' + r_4}{r_4} \text{ where } R' = \frac{3}{2} R_3.$$

Then, the total processing time is again given by Eq 3.3

$$T_6 = \frac{M C}{2} \left[ N \left( r_4 \right) + \frac{N r_4}{2} + R' + r_4 \right]. \text{ Now } V = 5 \text{ since there are}$$

three input variables and two output variables. $r_4$ is the time to transfer the larger of input or output packet.

The output packet has four elements, therefore

$$r_4 = \frac{4}{3} r_3.$$

$$\therefore T_6 = \frac{M C}{6} r_3 + \frac{3}{2} R_2 + \frac{4}{3} r_3.$$

We note that $T_6 > T_5$ but the increase in processing time is much smaller than 50%.
\[ T_6 = T_5 + \log_2 N \left[ M_c + \frac{N r_3^2 + R_2 + \frac{r_3}{3}}{6} \right] \]

and \[ \log_2 N \left[ M_c + \frac{N r_3^2 + R_2 + \frac{r_3}{3}}{2} \right] < \frac{T_5}{2}, \]

\[ \therefore T_6 - T_5 < \frac{T_5}{2} \]

The advantages of this scheme are

i) It requires only \( N \) storage locations for the twiddle factors as
against \( \frac{N}{2} \log_2 N \) if all the twiddle factors for every stage were to
be stored.

ii) It requires no special algorithm or mask vectors to generate the
twiddle factors for a given stage from the initial vector.

iii) The increase in processing time per processor is a constant, inde-
dependent of \( N \). Thus the increase in processing time per processor
can be absorbed by adding more processors. Then the Stream
Processing System will still operate at the maximum rate. Also,
the processors can be designed to optimize the computation time
for butterfly operations and the same improvement in performance
will apply to all the FFT algorithms irrespective of the value
of \( N \).
5-5 Matrix Transposition

One requirement of parallel processors that is rarely a requirement for serial processors is the necessity for rearranging data in order to take advantage of the opportunity for parallelism. Two dimensional matrix calculations are particularly susceptible to these requirements. In many problems it is necessary to have parallel access to both row and columns of a matrix. For example, in the case of matrix multiplication it is necessary to be able to access the rows of one matrix and columns of another matrix with equal speed. Generally, matrices are stored in the memories in row major order. This allows fast access of rows of a matrix but not columns, as explained earlier in section 4.2. One solution to this problem is to build an efficient mechanism for obtaining the matrix transpose. A fast algorithm for obtaining matrix transpose of an N by N matrix was developed by Stone [Stone 71]. He showed that the transpose of an N by N matrix can be obtained in exactly \( m \) perfect shuffles of the original matrix stored linearly in row-major order, where \( m = \lceil \log_2 N \rceil \). The original matrix is stored in the upper left submatrix of a \( 2^m \) by \( 2^m \) matrix. The \( 2^m \times 2^m \) matrix is stored as a linear vector in the data memory. This vector is shuffled \( m \) times. The resultant vector is the column-major order representation of the original matrix, hence the transpose of the matrix.

For example consider a 4 x 4 matrix \( A \),

\[
A = \begin{pmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\]
It is stored in the memory in row major order as the vector <\( a_{11}, a_{12}, a_{13}, a_{14}, a_{21}, a_{22}, a_{23}, a_{24}, a_{31}, a_{32}, a_{33}, a_{34}, a_{41}, a_{42}, a_{43}, a_{44} \)>

Fig. 5.11 illustrates \( m=2 \) shuffles of this vector. The resultant vector is <\( a_{11}, a_{21}, a_{31}, a_{41}, a_{12}, a_{22}, a_{32}, a_{42}, a_{13}, a_{23}, a_{33}, a_{43}, a_{14}, a_{24}, a_{34}, a_{44} \)>, which is the row-major order representation of \( A' \), where

\[
A' = \begin{array}{cccc}
\mathbf{a} & \mathbf{a} & \mathbf{a} & \mathbf{a} \\
11 & 21 & 31 & 41 \\
\mathbf{a} & \mathbf{a} & \mathbf{a} & \mathbf{a} \\
12 & 22 & 32 & 42 \\
\mathbf{a} & \mathbf{a} & \mathbf{a} & \mathbf{a} \\
13 & 23 & 33 & 43 \\
\mathbf{a} & \mathbf{a} & \mathbf{a} & \mathbf{a} \\
14 & 24 & 34 & 44
\end{array}
\]

It is obvious that \( A' \) is the transpose of matrix \( A \). This algorithm is very easily implemented in the Stream Processing System since it only requires \( m \) perfect shuffle to be performed on a given vector. It was shown earlier in section 5-2 that a perfect shuffle is easily implemented in the Stream Processing System as a binary vector operation requiring no processing time.
<table>
<thead>
<tr>
<th>Shuffle</th>
<th>Shuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁₁</td>
<td>a₃₁</td>
</tr>
<tr>
<td>a₁₂</td>
<td>a₁₂</td>
</tr>
<tr>
<td>a₁₃</td>
<td>a₁₂</td>
</tr>
<tr>
<td>a₁₄</td>
<td>a₃₂</td>
</tr>
<tr>
<td>a₂₁</td>
<td>a₃₃</td>
</tr>
<tr>
<td>a₂₂</td>
<td>a₁₄</td>
</tr>
<tr>
<td>a₂₃</td>
<td>a₃₄</td>
</tr>
<tr>
<td>a₂₄</td>
<td>a₄₂</td>
</tr>
<tr>
<td>a₃₁</td>
<td>a₁₃</td>
</tr>
<tr>
<td>a₃₂</td>
<td>a₄₁</td>
</tr>
<tr>
<td>a₃₃</td>
<td>a₂₂</td>
</tr>
<tr>
<td>a₃₄</td>
<td>a₄₂</td>
</tr>
<tr>
<td>a₂₁</td>
<td>a₂₃</td>
</tr>
<tr>
<td>a₄₂</td>
<td>a₄₃</td>
</tr>
<tr>
<td>a₄₃</td>
<td>a₂₄</td>
</tr>
<tr>
<td>a₄₄</td>
<td>a₄₄</td>
</tr>
</tbody>
</table>

4 x 4 matrix:

\[
\begin{array}{cccc}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44} \\
\end{array}
\]

↑
Row Major

↑
Column Major

Fig. 5.11
5-6 Linear Recurrence and Recursive Doubling

Recurrences are found often in computer design, numerical analysis and program analysis, so it is important to find fast, efficient ways to solve them. While not all recurrences are linear, the vast majority found in practice are, and fast algorithms have been developed for solving them on parallel processors [KOGGE 73], [TROUT 72], [KUCK 75]. The discussion presented here is mainly from Kogge and Stone's work.

The solution to a recurrence problem is a sequence \( x_1, x_2, \ldots, x_N \), where each \( x_i \) is a function of the previous \( m \) \( r \)'s, namely \( x_{i-1}, \ldots, x_{i-m} \). Formally, a recurrence problem consists of a set of recurrence equations:

\[
x_i = f_i(x_{i-1}, \ldots, x_{i-m}) \quad i = m+1, \ldots, N.
\]

In some recurrence problems the result of interest is only \( x_N \). Such recurrence problems produce scalar results. Examples of such problems are

a) computing an inner product of vectors \( A = (a_1, \ldots, a_N) \) and \( B = (b_1, \ldots, b_N) \). This can be written as a linear recurrence of the form

\[
X_i = X_{i-1} + a_i b_i \quad 1 \leq i \leq N
\]

where \( X \) is initially set to zero. After \( N \) iterations, the value of \( X \) is the inner product of \( A \) and \( B \).

b) computing the sum of \( N \) numbers, that is, computing \( a_1 + a_2 + a_3 + \ldots + a_N \). If \( X \) denotes the partial sum then the problem can be written
as the recurrence relation

\[ X = X + a_i, \ 1 \leq i \leq N, \] where \( X \) is initially set to zero.

In other problems it is necessary to obtain all the terms. The result
is a vector \( \langle X_1, X_2, \ldots, X_N \rangle \). A simple first order recurrence
problem is, given \( X_1 = b_1 \), find \( X_2, \ldots, X_N \), where

\[ X_i = a_i X_{i-1} + b_i. \]

Kogge and Stone have developed a fast algorithm for the general class of
first-order recurrence equations that can be placed in the form:

\[ X_1 = b_1 \]
\[ X_i = f_i(X_{i-1}) = f(b_i, g(a_i, X_{i-1})), \ 2 \leq i \leq N \]

where \( b_i \) and \( a_i \) are arbitrary constants and \( f \) and \( g \) are index-independent
functions that satisfy the following restrictions.

Restriction 1. \( f \) is associative. \( f(x f(y, z)) = f(f(x, y), z). \)

Restriction 2. \( g \) distributes over \( f \cdot g(x, f(y, z)) = f(g(x, y), g(x, z)). \)

Restriction 3. \( g \) is semiassociative, that is, there exists some
function \( h \) such that \( g(x, g(y, z)) = g(h(x, y), z). \)

Kogge and Stone have shown that all \( X_i \) for \( 2 \leq i \leq N \) can be computed in
time proportional to \([\log_2 N]\) on \( N \) processors.

We will discuss the algorithm for the following simple first order
recurrence problem. Let \( X_1 = b_1 \), and \( X_i = a_i X_{i-1} + b_i \). The problem is
to complete \( X_2, X_3, \ldots, X_N \) in \( O([\log_2 N]) \) time.

The basic algorithm involves a concept called recursive doubling,
which consists of breaking the calculation of one term into two equally
complex subterms. If we define a function

\[ Q(m,n) = \prod_{j=n}^{m} (\prod_{r=j+1}^{m} a_r)b_j. \]

where the vacuous product \( (\prod_{r=m+1}^{m} a_r) \) is given the value 1.

Then

\[ X_1 = b_1 = Q(1,1) \]
\[ X_2 = a_2X_1 + b_2 = a_2b_1 + b_2 = Q(2,1) \]
\[ X_3 = a_3X_2 + b_3 = a_3a_2b_1 + a_3b_2 + b_3 = Q(3,1) \]
\[ \vdots \]
\[ X_i = a_iX_{i-1} + b_i = Q(i,1) \]
\[ \vdots \]
\[ X_N = a_NX_{N-1} + b_N = Q(N,1) . \]

This can also be written as

\[ Q(1,1) = X_1 = b_1 \]
\[ Q(2,1) = X_2 = a_2Q(1,1) + Q(2,2) \]
\[ \vdots \]
\[ \vdots \]

That is, \( Q(2i,1) + X_{2i} = (\prod_{r=i+1}^{2i} a_r)Q(i,1) + Q(2i,i+1) . \)

In general, for any \( i,k \) such that \( 1 \leq k \leq i \leq N \) and any \( j \) such that \( 1 \leq j \leq k \)

\[ Q(i,i-k) = (\prod_{r=i-j+1}^{i} (a_r)) Q(i-j, i-k) + Q(i,i-j+1) \quad --E \, 5.1 \]

This expression gives us the recursive doubling since we have been able to break down the computation of \( Q(i, i-k) \) into two equally complex subterms \( Q(i-j, i-k) \) and \( Q(i, i-j+1) \). The algorithm is illustrated in Fig. 5.12 for \( N=8 \). The figure shows parallel computation of \( X_8 \). It can
\[ x_8 = Q(8,1) = a_8 a_7 a_6 a_5 \ Q(4,1) + Q(8,5) \]

\[ Q(8,5) = a_8 a_7 \ Q(6,5) + Q(8,7) \quad Q(4,1) = a_4 a_3 \ Q(2,1) + Q(4,3) \]

**Computation node**

\[ Q(8,7) = a_8 b_7 + b_8 \quad Q(6,5) = a_6 b_5 + b_6 \quad Q(4,3) = a_4 b_3 + b_4 \quad Q(2,1) = a_2 b_1 + b_2 \]

**FIG. 5.12  LINEAR RECURRENCE**
be seen that by using E 5.1 judiciously it is possible to break down the computation of \( X_8 = Q(8, 1) \) into \( Q(8, 5) \) and \( Q(4, 1) \) which can be further broken down into \( Q(8, 7), Q(6, 5) \) and \( Q(4, 3) \) and \( Q(2, 1) \), respectively. As shown in Fig. 5.12, \( Q(8, 7), Q(6, 5), Q(4, 3) \) and \( Q(2, 1) \) can be computed in parallel, then \( Q(8, 5) \) and \( Q(4, 1) \) can be computed in parallel. Therefore, \( X_8 \) can be evaluated in \( \log_2 N \) computation times.

In order to make the computations at the nodes identical, the computation performed at the nodes are as follows:

\[
\text{if} \quad a_i, b_i \quad \text{and} \quad a_j, b_j \quad \text{are the inputs to a computation node,}
\]

\[
\text{then the outputs are} \quad a'_i \quad \text{and} \quad b'_i
\]

where

\[
\begin{align*}
\begin{array}{l}
a'_i = a_i a_j \quad \text{and} \\
b'_i = a_i b_j + b_i
\end{array}
\end{align*}
\]

The former computes the \( \prod_{r=i-j+1}^{i} (a_r) \) part of E 5.1 and the latter computes \( Q(i, i-k) \). The algorithm for computing the values \( X_2, X_3, \ldots, X_7 \) is the same as that for \( X_8 \) except different values of the \( Q \) function are used. Kogge and Stone have shown that the computation of all the values \( X_2 \) to \( X_8 \) can be combined into one parallel algorithm as shown in Fig. 5.13. Stone has shown that the interconnection required from a stage to the next stage is essentially an "Inverse Shuffle" and the algorithm of Fig. 5.13 can be redrawn as shown in Fig. 5.14 [STONE 75]. An Inverse Shuffle is the same as the Perfect Shuffle interconnection with the direction of the arrows reversed. Although Fig. 5.14 shows only one output line from each processor, there are three outputs from each processor, \( a'_i, b'_i \) and the new mask word. A mask bit is needed at each processor for every operation because some of the operations are null operations, as shown in Fig. 5.13.
* SOLID NODE INDICATES A COMPUTATION NODE.

○ INDICATES NULL COMPUTATION.

FIG. 5.13
Inverse Shuffle

In order to implement this algorithm efficiently on the Stream Processing System, it is necessary to find an efficient and simple way to implement the 'Inverse Shuffle'. Indeed, the Inverse Shuffle can be implemented in the Stream Processing System in a manner very similar to the implementation of the Perfect Shuffle. Only one control loop is assigned to the input vector and two control loops to the output vector. One output control loop is set to the first location of the output vector and the second loop to the halfway location. The input control loop forms input packets of two elements each by selecting two consecutive elements for every packet. In order to accomplish that, the control loop has to be shifted after the selection of every element. With respect to Figure 5.15, the packets formed by the input control loop are \(<0, 1>, <2, 3>, \ldots, <N-2, N-1>\). These packets are transferred in sequence to the output bus (no processing is required). The output control loops treat the packets as result packets with the elements of two different variables in each packet. Thus the first element of each packet is stored by the first output control loop and the second element by the second control loop. The two result vectors are, then,

\(<0, 2, 4, 6, \ldots, N-2> and <1, 3, \ldots, N-1>\).

Since the starting location of the second vector was initially selected \(N/2\) locations away from the starting location of the first vector, the two vectors together form the final vector \(<0, 2, 4, 6, \ldots, N-2, 1, 3, \ldots, N-1>\). This is exactly the output vector obtained by the Inverse
FIG. 5.15 INVERSE SHUFFLE
Shuffle as shown in Fig. 5.15. Thus, the Inverse Shuffle is also implemented as a binary vector operation which requires no processing.

Implementation of the Parallel Algorithm for Linear Recurrence

Having seen how the Inverse Shuffle can be implemented, consider the original problem of implementing the algorithm of Fig. 5.14 in a Stream Processing System. Since each stage of the algorithm has N independent operations they can be treated together as a vector operation. Thus, each stage of the algorithm is executed as an n-ary vector operation where n is equal to five because each operation requires five inputs, \( a_i, b_i, a_j, b_j \) and a mask bit. There are two outputs, \( a^'_i \) and \( b^'_i \), from each processor in Fig. 5.14. However, in our implementation we will add one more operation, the computation of the next mask bit, which produces the third output, the new mask bit. Then, each processor has three outputs \( a^'_i, b^'_i \) and the new mask bit.

The scheme for generating the new mask bits is identical to the scheme used for polynomial evaluation in section 5-3. The mask bits for all the stages for each input packet are put together in a mask word. Then there are N mask words and each mask word contains \( \log_2 N \) bits (for \( \log_2 N \) stages) of significance. For each stage of computation the least significant bit of the mask word is taken as the mask bit. The new mask word is generated by shifting the current mask word to the right by one. This puts the next to least significant bit in the least significant position as required. The mask words for the example of Fig. 5.13 are
< 0 0 0 > ,
< 0 0 1 > ,
< 0 1 1 > ,
< 0 1 1 > ,
< 0 1 1 > ,
< 0 1 1 > ,
< 1 1 1 > ,
< 1 1 1 > ,
< 1 1 1 > ,
< 1 1 1 > .

In general, the mask words are

\[
\begin{align*}
(2^0-1) & , \\
(2^1-1) & , \\
(2^2-1) & \quad \text{2 terms} \\
(2^3-1) & \quad \text{2 terms} \\
(2^3-1) & \quad \text{2 terms} \\
(2^3-1) & \quad \text{2 terms} \\
& \quad \ldots \\
& \quad \ldots \\
& \quad \\
& \quad \\
\end{align*}
\]

Although each input packet has five elements, all the elements are obtained from three vectors from the memory. The three vectors are 

\[ A = <0, a_1, a_2, \ldots, a_N>, \quad B = <0, b_1, b_2, \ldots, b_N> \]

and the mask vector. One control loop is set to begin at the first location of A and another loop is set to the second location of vector A. Similarly, two control loops are set to the first two locations of vector B. Both the
vectors, A and B, are accessed twice. Thus, effectively five vectors are accessed for formation of packets. The packets sent to the processors are then 0, a₁, 0, b₁, mask>, <a₁, a₂, b₂, mask>, . . . <aᴺ⁻¹, aᴺ, bᴺ⁻¹, bᴺ, mask>. The processors receive the packets and perform the following three operations on each packet:

i) \( a'_i = a_ia_j \),

ii) \( b'_i = a_ιb_j + b_i \),

iii) New mask word = Right shift of the current mask word by one place.

The result packets are <a₁', b₁', new mask>, <a₂', b₂', new mask>, . . . <aᴺ', bᴺ', new mask>. These results are stored in the original A, B and the mask vector. The elements of A and B are stored from the second position onwards. The first elements of A and B are still 0. This is done to ensure that the first packet gets five elements, just as the rest of the packets. Once the results are stored the three vectors are shuffled through an Inverse Shuffle pattern. This completes the computation of the first stage.

The same procedure is repeated for the remaining \((\log_2 N) - 1\) stages at the end of which the vector B contains \(0, X_1, X_2, \ldots, X_N\). We have shown that parallel algorithms for linear recurrence can be effectively implemented on the Stream Processing System. It should be noted that the number of stages is still \(\log_2 N\). Each stage of the algorithm is implemented as a vector operation. The major advantage of this scheme is that it can be implemented with any number of processors. It has already been shown that for a given number of processors, \(N_p\), the Stream Processing System operates at the maximum parallelism, \(O(N_p)\).
CHAPTER VI

CONCLUSIONS

This chapter summarizes the results obtained in this thesis and indicates directions for future research.

6-1 Summary of Results

In this thesis we have introduced and developed the concept of Stream Processing. It was shown that a Stream Processing System has some of the versatility of a multiprocessor architecture and the simple memory requirements of a pipelined processor. The performance of a Stream Processing System with N processors was shown to be comparable to that of a general multiprocessor system consisting of N processors or a system containing an N-stage pipeline. It was shown that a stream processing unit can be implemented in an open-ended and modular fashion. This allows a system to contain an arbitrary number of processors without requiring any changes in the memory system, interconnection network or the systems programs. The analysis presented in this thesis provides a designer with the means to select an optimal number of processors required to achieve a desired performance for a given set of operations.

This thesis has also developed a modular and parallel memory system and an interconnection network which supplies data to the processing unit in a stream. An analysis of the memory scheme is presented. The analysis determines the conditions that have to be satisfied in order to ensure that (1) the memory operates at maximum bandwidth, and (2) the memory bandwidth is large enough to allow the processing unit
to operate at maximum throughput. It is shown that the addressing schemes presented in this thesis allow addresses to be indexed in the memory modules. So only the starting addresses of the given vectors have to be sent, initially, to the memory from the central control processor. This facilitates the implementation of vectors and arrays as primitive data types.

It is shown, in this thesis, that the interconnection network can be modularized and implemented as a part of the memory modules. It is further shown that one simple interconnection network is sufficient to handle data alignment as well as unscrambling of skewed vectors (for small skew). It is also shown that the buffering and the interconnection network allow the memory system and the processing unit to operate independently and asynchronously. The memory system and the interconnection network also allow efficient implementation of certain other interconnection patterns such as the Perfect Shuffle Interconnection.

Finally, the versatility and processing capability of the Stream Processing System is demonstrated by the implementation of some parallel algorithms. In particular, it is shown that parallel algorithms for the Fast Fourier Transform, polynomial evaluation, solution of Linear Recurrence problems can be implemented very efficiently. It is also demonstrated that algorithms can be implemented for a various number of input data points without requiring any changes in the number of processors, memory modules or changes in the interconnection structure.

It should be noted that this thesis provides a means for construc-
ting a low-cost system using microprocessors to achieve the capability of some of large vector machines for vector processing and some other special applications. The emphasis on modularity ensures that the same design can be efficiently used for solving problems with different processing and storage requirements.
6-2 Extensions and Future Work

The throughput of a Stream Processing System is bounded by the bandwidth of the data busses and the time associated with the control signals that are required to execute the transfer of a word. The minimum time required to completely transfer a word on a data bus will be called a 'data cycle'. This time is determined by the hardware sued for implementing the busses and the control circuitry (this time was taken to be 'w' clock cycles in Chapter IV). Then the memory can, at best, transfer one word per data cycle and receive, at best, one word per data cycle. The input data packet takes approximately p data cycles and the output packet takes q data cycles, where p is the number of input variables and q is the number output variables in the given operation. However large the number of processors, the processing unit can at best process one data packet per p data cycles, assuming p \geq q (the time taken is q cycles if q \geq p). We suggest two approaches that may be taken to increase the throughput of the system.

A. One natural way of increasing the throughput is to use wider busses. If a p-word input bus is used then a complete input packet can be transferred in one data cycle instead of p data cycles. Similarly, a q-word output bus allows an output packet to be transferred in one data cycle. Then, the processors can process, at best, one data packet per data cycle. This is an improvement in throughput of the processing unit by a factor of p. Note that this improvement is dependent on the operation being performed. If a monadic operation is being considered (there being only one input variable and one output variable), there is no increase in throughput. On the other hand, if the given operation has
three input variables (as in the case of polynomial evaluation) the improvement in throughput is by a factor of three.

This scheme does not require any changes in the interconnection network except that data lines go in parallel from every buffer location to different parts of the data bus, instead of being multiplexed onto the same part. It should be noted that the number of control loops required is the same as before and there is no change required in the control signals or the addressing schemes.

Besides the additional hardware cost the major disadvantage of such a scheme is that the bus capacity is not always fully utilized. In order to accommodate all the possible vector operations, the bus width has to be \( p_{\text{max}} \) words for the input bus and \( q_{\text{max}} \) for the output bus, where \( p_{\text{max}} \) and \( q_{\text{max}} \) are the maximum number of input variables and output variables, respectively, to occur in the set of all executable vector operations. Then, during operations that have \( p < p_{\text{max}} \) and \( q < q_{\text{max}} \), the total bus width is not utilized.

B. The second approach is to completely replicate the processing unit along with another set of two busses, as shown in Fig. 6.1. Data is stored in the memory in superwords. A superword contains as many elements as the number of processing units. In the case of the example of Fig. 6.1, each superword contains two data elements. Whenever a superword is accessed from a memory module one word is sent to the first processing unit and the second to the second processing unit. This requires that all the vectors be stored in superwords. For example, a vector \( A \) would be stored in the memory as \( a_1 a_2, a_3 a_4, \ldots \); a vector \( B \)
would be stored in the form \( b_1 b_2, b_3 b_4, \ldots \). An operation between A and B would send the packet \( a_1, b_1 \) to the first processing unit and the packet \( a_2, b_2 \) to the second processing unit at the same time. Thus, the first processing unit receives the odd numbered packets and the second processing unit receives the even numbered packets. Since the two processing units operate in parallel the throughput is doubled. This improvement is realized for all operations irrespective of the number of variables in each operation and the busses are always fully utilized.

This scheme can be implemented with only one set of control loops for the memory but it requires separate sets of control loops for the processing units. The buffers have to be increased to accommodate the superwords but there is no change required in the addressing schemes.

The limitations of such a scheme are

i) cost; clearly, doubling the throughput also doubles the cost of processing sections of the system, including the costs of the busses.

ii) this system needs special interfaces to allow communication between the two processing units. This need comes up in special vector operations like 'Reduction'. For example, summation of all the elements of a vector. Since this operation produces only one scalar result it has to be obtained in one processor. Thus it is necessary to route all the elements to a single processor. If the operation can be broken up into two separate summations that can be performed on the two
processing units, then it is still necessary to finally add the partial sums in one processor to obtain the final sum. The problem of intercommunication also turns up because of requirements of data alignment. Consider an operation of the type A[I] + A[I] + A[I+1]. Here elements A[I] and A[I+1] are stored in different parts of the same superword and will automatically go to different processing units. It is then necessary to make sure that each processing unit can obtain data from any part of a particular superword.

More work needs to be done to examine whether the intercommunication requirements between several processing units can be a real bottleneck. It also needs to be examined if the requirements of the communication interfaces are simple and easily implementable. Finally, it remains to be examined if such an approach is modular and cost-effective.

Another problem of interest is to analyse the performance of the system for a set of independent and different operations. That is, all processors do not execute the same operation. Processors may execute different operations requiring different amounts of processing time. Thus 'R' in E 3.3 is no longer constant. The Stream Processing System as described earlier executes only one vector operation at a time. A new instruction is not started until all the results of the current instruction are completely stored in the memory. Thus processors are idle when all the processing is finished and results are being stored. Further, if some of the vector operations have lengths less than the number of processors, then some of the processors remain idle during those vector operations. In order to better utilize the processors, it needs
to be examined if the free processors can execute another instruction and if execution of several vector instructions can be overlapped.

There are several problems associated with this approach that need to be examined. Firstly, if the performance of this system still remains comparable to other multiprocessor configurations. Secondly, if it is possible to still have the same interconnection structure and, if not, then how severe are the required modifications. Finally, it needs to be examined if addressing schemes such as MSAS and IAS can be modified to accommodate more than one vector operation.

It can be shown that the Left Priority Scheme is more efficient than the Two Loop Scheme for scheduling operations with different processing times. For example, consider a simple two-processor system. Let the first processor receive a task of length 12 time units at time $T_0$. The second processor receives a task of length, say 3 time units at time $T_0 + 1$. Then the second processor is free at time $T_0 + 4$ to receive another task. In the Two Loop Scheme, the second processor does not get the next task until the first processor has finished its current task and received a new task, that is, until the time $T_0 + 13$. Whereas, the Left Priority Scheme allows the second processor to receive the next task as soon as it finished the first task. A statistical analysis of the Left Priority Scheme for various distributions of task times remains to be done.
REFERENCES


[OMEN 74] Sanders Associates, "OMEN-60 Orthogonal Computers," from Appendix L [ENSLOW 74].


