

Cyclical Processor and Computer Architectures
for
Highly Parallel Applications

by

Fut-Suan Wong

B.Eng.(Hons.), University of Singapore, 1979
M.A.Sc., The State University of New York, 1980

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

DOCTOR OF PHILOSOPHY

In

THE FACULTY OF GRADUATE STUDIES

Department of Electrical Engineering

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

JANUARY, 1984

© 1984, F. S. Wong

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date April 10, 1984

Abstract

During the last few decades, the search for powerful computing machines has been one of the several endless pursuits among the scientific community. In this thesis, several novel architectural ideas for the designs of high-performance computing machines are presented, and the practicality and usefulness of cyclical architectures -- ones which have their hardware resources cyclically arranged -- in this respect are examined. These ideas are illustrated with the use of specific application examples including parallel sorting, packet-switched communications and the design methodology of a class of next-generation computers.

In the first part of our studies, the structure and control algorithms of a single-chip, recirculating systolic sorter (RSS), are presented. The correctness of the algorithms is proved, and general operational constraints are derived. This parallel sorter is highly amenable to VLSI implementations because of the simple control structure and the regular, repetitive and near-neighbour type of interconnections required. The number of quadruple comparators needed to sort N items is $N/4$, and the average sorting time is found to be bounded by $(\log N)^2$ and N . A hardware termination is incorporated into the control unit of the sorter, so that the sorting process can be terminated as soon as the input list is in the desired order.

In the second part of our studies, a novel loop-structured switching network (LSSN) is presented. It is intended for packet communications in large-scale systems consisting of hundreds to thousands of interconnected devices. With L loops -- where L is a power of two, it can connect up to $N=L(\log L)$ pairs of transmitters and receivers, using only $N/2$ two-by-two switches; in terms of switch counts and the amounts of wiring, this network is very advantageous when the value of N is large. It can be extended incrementally, and is free of the store-and-forward type of deadlocks which prevail in other cyclical, packet-switched networks. Our simulation results show that its average throughput rate and delay are close to that of other designs despite its relatively low switch count.

In the third part of our studies, a new design methodology for the next-generation computers is described. Our proposed system, the Event-Driven Computer (EDC) is primarily a data-driven system which has its computing resources arranged as a circular pipeline, and it is supplemented with control-driven activities. Such a combined approach is aimed at extracting the advantages of both the "pure" data-driven and control-driven computations while alleviating their shortcomings. Compared to other designs, an EDC has the merits of a simpler architecture, better resource utilization, array processing capabilities and a higher speed range.

As is shown by our studies, the properties of the cyclical architectures depend greatly on how the information packets interact with each other; deadlocks, for instance, will occur on systems such as the Loop-Structured Switching Network because of the asynchronous, circular requests of network resources by the packets (we have, however, presented a deadlock avoidance scheme), but will not occur on synchronous systems such as the Recirculating Systolic Sorter. In general, the resource utilization of the cyclical architectures are higher than that of the acyclic ones -- or equivalently, the cyclical architectures can handle larger amounts of information with relatively smaller areas -- they are therefore more suitable to the designs of very large-scale systems.

Key phrases:

computer architectures, next-generation computing, systolic arrays, parallel sorting, packet-switched networks, store-and-forward deadlocks, data-driven and control-driven computations.

Table of Contents	page
Abstract.....	ii
Table of Content.....	v
List of Figures.....	viii
List of Tables.....	x
Nonemclature.....	xi
Acknowledgements.....	xii
Chapter I. Introduction	
I.1. Background Information.....	1
I.2. Cyclical Architectures.....	4
I.3. Objectives and Scope of the Thesis.....	7
Chapter II. A Systolic Processor for Parallel Sorting	
II.1. Introduction.....	9
II.2. The Recirculating Systolic Sorter (RSS)	
A. Network Description.....	12
B. The Quadruple Comparator.....	14
C. The Comparison/Exchange/Shift Operations.....	16
II.3. The RSS Algorithms	
A. Algorithm I.....	20
B. Algorithm II.....	21
C. Examples.....	22
II.4. Operational Constraints	
A. Constraints on the Size of RSS.....	23
B. Marking Scheme A.....	26
C. Marking Scheme B.....	26
II.5. Analysis of the RSS Algorithms	
A. Analogy with the Odd-Even Transportation	

Sort.....	27
B. Correctness of the RSS Algorithms and Marking Schemes.....	29
C. Correctness of the Termination Method.....	40
D. Timing Complexities.....	41
II.6. Discussions.....	43
Chapter III. A Novel Loop-Structured Switching Network (LSSN)	
III.1. Introduction.....	48
III.2. Network Topology	
A. Addressing Scheme and Connection Function....	50
B. Routing Scheme.....	51
III.3. Network Properties.....	59
A. Network Conflicts.....	62
B. Deadlocks and Avoidance Method.....	65
C. Network Extensibility.....	68
III.4. Simulations and Performance Analysis.....	70
III.5. Discussions and Outlook.....	74
Chapter IV. Design and Evaluation of the Event-Driven Computer (EDC)	
IV.1. Introduction	
A. Background Information.....	81
B. Recent Developments.....	83
C. Overview of Our Approach.....	85
IV.2. The EDC Hardware Architecture	
A. Processing Modules.....	91
B. Storage Modules.....	96
C. Switches.....	98
IV.3. The EDC Information Structure	

A.	Machine Code Formats.....	104
B.	Packet Formats.....	107
C.	Program Organization.....	108
D.	Data Structures.....	111
E.	Process and Resource Management.....	113
IV.4.	The EDC Programming Language Structure	
A.	Statements and Program Blocks.....	114
B.	Language Constructs for Array Processing....	120
IV.5.	Performance Analysis	
A.	Flow Analysis of EDC.....	123
B.	Example.....	126
C.	Considerations for Generalized Computations.	131
IV.6.	Discussions and Outlook.....	132
Chapter V.	Conclusions	
V.1.	Summary of Results.....	138
V.2.	General Discussions.....	139
V.3.	Suggestions for Further Work.....	141
Appendix A		
Appendix B		
Appendix C		
References		
Vita		

List of Figures

page

1.	Fig.I.1.	The cyclical configuration.....	5
2.	Fig.II.1.	The Recirculating Systolic Sorter (RSS).....	11
3.	Fig.II.2.	The control unit of RSS.....	11
4.	Fig.II.3.	The Schematic diagram of a quadruple comparator.....	15
5.	Fig.II.4.	Symbols used for comparison and shift.....	16
6.	Fig.II.5.	The four operations performed by the quadruple comparators.....	19
7.	Fig.II.6.	An example to illustrate RSS Algorithm I and Marking Scheme A.....	24
8.	Fig.II.7.	An example to illustrate RSS Algorithm II and Marking Scheme B.....	25
9.	Fig.II.8.	The odd-even sorter.....	28
10.	Fig.II.9.	The three indexes: i , j , and J , and the initial marker position $M(i)$	30
11.	Fig.II.10.	The horizontal comparisons carried out on the RSS array.....	30
12.	Fig.II.11.	The number of comparison cycles versus the number of items to be sorted (Algorithm I).....	42
13.	Fig.II.12.	A general-purpose computer system with special-purpose chips attached [19].....	43
14.	Fig.III.1.	Assignment of loop and link labels on a LSSN which has 16 loops and 32 switches.....	53
15.	Fig.III.2.	Connection of transmitting and receiving devices on a LSSN with 16 loops.....	54
16.	Fig.III.3.	The schematic diagrams of a Type-A switch.....	55
17.	Fig.III.4.	A 16x16 baseline network	78
18.	Fig.III.5.	Effects of buffer size on the throughput and delay of a 64x64 LSSN.....	78
19.	Fig.III.6.	The throughput rates of a 64x64 baseline, a 64x64 LSSN and a 16x16 baseline, versus the inter- arrival time.....	79
20.	Fig.III.7.	The delay curves of a 64x64 baseline, a 64x64 LSSN and a 16x16 baseline, versus the inter- arrival time.....	80
21.	Fig.IV.1.	EDC system block diagram.....	87
22.	Fig.IV.2.	The connection diagram of EDC hardware architecture.....	90
23.	Fig.IV.3.	The schematic diagram of a PSN switch.....	100
24.	Fig.IV.4.	Parameter passing between the calling program M and the called program P.....	109
25.	Fig.IV.5.	The interactions between calling programs and a task program.....	110
26.	Fig.IV.6.	The physical and logical arrangements of the EDC memory system.....	111
27.	Fig.IV.7.	The implementation of a resource manager using a task program.....	113
28.	Fig.IV.8.	A "Begin/End" block and its data-flow graph.....	116
29.	Fig.IV.9.	An "IF" block and its data-flow graph.....	116

30.	Fig.IV.10. A "Match" block and its data-flow graph....	117
31.	Fig.IV.11. A "Loop" block and its data-flow graph.....	118
32.	Fig.IV.12. The statement, data-flow graph and machine code of a parallel vector operation.....	120
33.	Fig.IV.13. The statement, data-flow graph and machine code of a reduction operation.....	121
34.	Fig.IV.14. The statements of some alignment operations, and the data-flow graph and machine code format of the "SHIFT" operation.....	123
35.	Fig.IV.15. The MARPf,MATR and MARPc,max curves of the given example.....	129

	List of Tables	page
1.	Table II.1. Requirements of the RSS marking schemes..	40
2.	Table II.2. Complexities of sorting networks.....	47
3.	Table III.1. Scalar operations.....	135
4.	Table III.2. Compound operations.....	135
5.	Table III.3. The "Operand/Next instructions" fields of scalar operations.....	136
6.	Table III.4. The "Operand/Next instructions" fields of compound operations.....	136
7.	Table III.5. The formats of instruction packets.....	137
8.	Table III.6. The formats of result packets.....	137

Nomenclature

ADT	: Array Description Table
ARS	: Average routing steps
C	: Number of columns
CS	: Channel Selector
EDC	: Event-Driven Computer
GCU	: Global Control Unit
i, I	: Index (short form of "Instruction" when used as subscript)
j, J	: Index
k, K	: Index
IRs	: Instruction Registers
L	: Number of loops used in LSSN
LIT	: Linkage Information Table
LMS	: Local Memories
LSSN	: Loop-Structured Switching Network
M	: Number of Local Memories
MATR	: Maximum Average Throughput Rate
MARP	: Maximum Acceptance Rate of Packets
MIMD	: Multiple-Instruction and Multiple-Data (computer systems)
N	: Number of inputxp
P	: Number of processors
PDF	: Piece-wise Data Flow (computer)
PSN	: Packet-Switched Network
R	: Number of rows or Receiving Processors (short form of "Result" when used as subscript)
RL	: Request List
RP _s	: Receiving Processors
R _r	: Receiver
RSS	: Recirculating Systolic Sorter
SIMD	: Single-Instruction and Multiple-Data (computer systems)
SISD	: Single-Instruction and Single-Data (computer systems)
SM	: System Memory
SP	: Supervisory Processor
SUT	: Storage Utilization Table
SW	: Switch
T	: Number of Transmitting Processors
TP _s	: Transmitting Processors
Tr	: Transmitter

)

Acknowledgements

I sincerely thank my supervisor, Dr. M. R. Ito for his patient help and guidance during the course of my graduate program. I would also like to thank Dr. Chanson, Dr. Schrack and Dr. Vuong for their services as members of my supervisory committee.

I am also thankful to my past and current office mates, for making my stay on this campus a memorable one.

As for my financial support, I am grateful for the research assistantships provided by my supervisor, the teaching assistantships provided by the Department of Computer Science, and the awards provided by the Lee Foundation of Singapore.

Chapter I. Introduction

1. Background Information

The demand for high speed computation is ever-increasing, particularly among the scientific community engaged in large-scale computation such as weather forecasting, realtime battlefield assessment, artificial intelligence and simulations of very large and complex processes. While conventional computer systems can handle many of the current demands, they suffer from certain drawbacks -- ranging from software obesity to hardware inextensibility -- which severely restrict their usefulness in the design of the so-called "fifth-generation" computers [1] which are currently being planned for future very large-scale applications.

The first four generations of computers are commonly distinguished by their constituent technologies -- vacuum tubes, transistors, integrated circuits and, currently, very large-scale integration (VLSI). Central to the fifth-generation concept is a break with the conventional, or sometimes referred to as the Von Neumann, computer architecture that has prevailed in the first four computer generations [2].

Several classes of computer architectures have been proposed for the next-generation computers, including tree structures, square and cube arrays, pipelines, systolic arrays

[3], data-driven systems [4], demand-driven systems [5] and dynamic structures [55,57,61]. As of today, none of these architectures has yet evolved to become the single, dominant basis of research work in this area.

In this thesis, we will look into another interesting design methodology -- cyclical architectures -- for highly parallel applications, and several ideas based on the concept of cyclical architectures will be proposed. Our new designs will also incorporate the fundamental principles of systolic, packet communications, data-driven and control-driven systems.

Systolic systems are characterized by their data-flow pattern: rhythmic data movements analogous to the pulsations in the arteries caused by the recurrent contractions of the hearts. Because of their simple, highly repetitive structures, systolic systems are very amenable to VLSI implementations. The algorithms of many specialized applications such as the Fast Fourier Transform and matrix multiplications, have been proposed for systolic computation [3].

Packet communications are traditionally meant for computer systems which are geographically apart and interconnected via local networks; but recently, they have also been proposed for multiprocessor systems consisting of tens to thousands of closely interconnected processing and storage modules -- examples are data-driven computers in which instruction executions are triggered by the arrivals of input

operands which are encapsulated into the form of packets, and networks are used to convey these packets among the hardware modules.

Data-driven computers have recently received enormous attentions due to their simplicity in the explorations of asynchronous parallelism; but on the other hand, they do not take advantage of the simple control structure that exists in array computation, and also some inherently sequential activities do not conform naturally to the notion of data-driven computation.

In the conventional, control-driven computers, instruction executions are sequenced explicitly by control signals generated by the central processing units; in contrast to data-driven systems, they are more advantageous in handling array computation because they make use of the simple control structures which exist in array computation; but on the other hand, the exploitations of parallelism in control-driven systems are more difficult because explicit control signals are needed to specify the branching and merging of execution paths, which otherwise could be done implicitly in data-driven systems by operand packets which are sent among the instructions.

More details of these various systems will be provided in the following chapters.

We believe that in order to gain significant

improvement in the computation speed over existing computer systems, the new designs may have to depart from the prevalent sequential computation in both hardware and software to various extents. In other words, some of the existing development tools such as off-the-shelf components, compiler techniques, etc., may not be useful in our designs; for these reasons, we will only emphasize the architectural aspects but not any immediate implementation.

Throughout this dissertation, the term "processor" is used to denote a piece of passive hardware capable of only primitive operations; on other hand, "computer" refers to a fullfledged machine capable of executing high-level operations; "highly parallel" or "next-generation" applications are those containing large amounts of both synchronous and asynchronous, high and low-level computation which can be performed in parallel, such as those examples quoted in the beginning of this chapter.

2. Cyclical Architectures

The rationale of our advocacy of cyclical architectures is based on the behaviour of program executions. As exhibited in the execution cycles of instructions as well as "DO-LOOPS" which exist in nearly all scientific and business-oriented programs, the ways in which most programs are executed, are basically cyclical in nature. It is therefore very spontaneous to envision a class of architectures which

have their resources arranged into a cyclical configuration as follows:

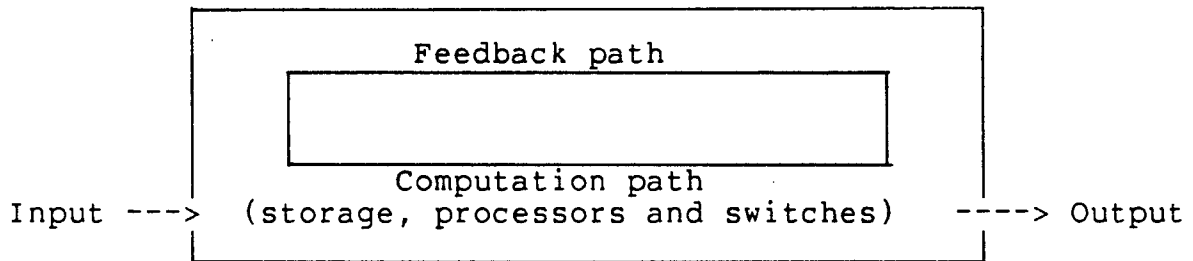


Fig.I.1. The cyclical configuration.

The main computation path in Fig.I.1 consists of both processing and switching elements, and either shift-registers or memory words are used for buffering and storage purposes. The information which goes through the feedback path are packets of either data, control signals or both, depending on the applications.

Current research involving such cyclical architectures could be broadly classified into three areas depending on the nature of the feedback signals:

- (1) Special-purpose processors attached to host computers: Examples are processors for the Fast Fourier Transform [6,7] and matrix transposition [9]. For this area of application, the purpose of feedback is to allow further interactions among the data elements and also to re-use the resources along the computation path.

- (2) Interconnection networks for processor-to-memory or processor-to-processor communications: Examples are single-staged shuffle-exchange networks [9,12] and multi-staged shuffle-exchange networks [20,21,22]. For this area of applications, the sole purpose of feedback is to re-use the resources; there is no interactions among the data.
- (3) Fullfledged, high-performance computers: With only a few exceptions [28,60], nearly all data-driven systems are based on the cyclical configuration [4,5]. For this area of applications, packets are fed back as the result of the completion of instruction cycles; and new instruction packets are brought into the computation path when certain result packets are received at the end of the feedback path.

If a system could be implemented with either the cyclical or the acyclic configuration, then the relative merits and demerits of the two configurations are as follows. In general, the cyclical configuration would give rise to better resource utilization than the acyclic one, because its resources could be used repeatedly by means of feedback; this feature would incur tremendous savings in system resources, especially when the size of the system is very large. Therefore, if the entire system is to be considered for fabrication on a single integrated-circuit chip, its cyclical configuration would be a better choice. On the other hand,

the control of the cyclical configuration is usually more difficult: in some systems, masking bits are needed to disable a subset of the processing resources [9]; while in others, feedback counts are required to separate the feedback signals from the incoming ones. If the cyclical configurations are used asynchronously (e.g., as packet-switched communications networks), then they would be susceptible to the store-and-forward type of deadlocks due to circular requests of resources. Another important characteristic of packet-switched, cyclical systems is their lack of responsiveness, because when interrupts occur, the computation path could already be congested with information packets such that the interrupts cannot be processed immediately.

3. Objectives and Scope of the Thesis

The main objective of this thesis is to advocate cyclical architectures as the basic design principle of a class of high-performance systems. Our ideas will be demonstrated by specific applications including parallel sorting, packet-switched communications and the design of a novel computer -- all of which are of current research interest. The advantages of our designs relative to others will be discussed, and the methods to resolve the various afore-mentioned demerits of cyclical architectures will be presented.

In Chapter II, we will present a recirculating systolic sorter (RSS) which is designed as a single-chip, parallel sorting module to be attached to a host computer. The sorting algorithms, design of the controller, and relative merits of the RSS will be detailed. Chapter III will describe a loop-structured switching network (LSSN) intended for communications in packet-switched, multiprocessing environments. The topology, properties and performance analysis of LSSN will be discussed, and the occurrence and resolution of deadlocks will be presented. Chapter IV will outline the design of the Event-Driven Computer (EDC) which is primarily a data-driven system supplemented with control-driven activities. The rationale of design, hardware and software organizations and performance of EDC will be addressed. General discussions and suggestions of further work will be given in Chapter V.

Chapter II. A Systolic Processor For Parallel Sorting

Abbreviations:

N: Number of input items
C, Column#: Number of columns
R, Row#: Number of rows
P, Comparator#: Number of comparators
i: Loop index
j: Moving position index
J: Fixed position index
M(i): Initial marker's position in loop i
t: Comparison cycle time
"*": A marker

1. Introduction

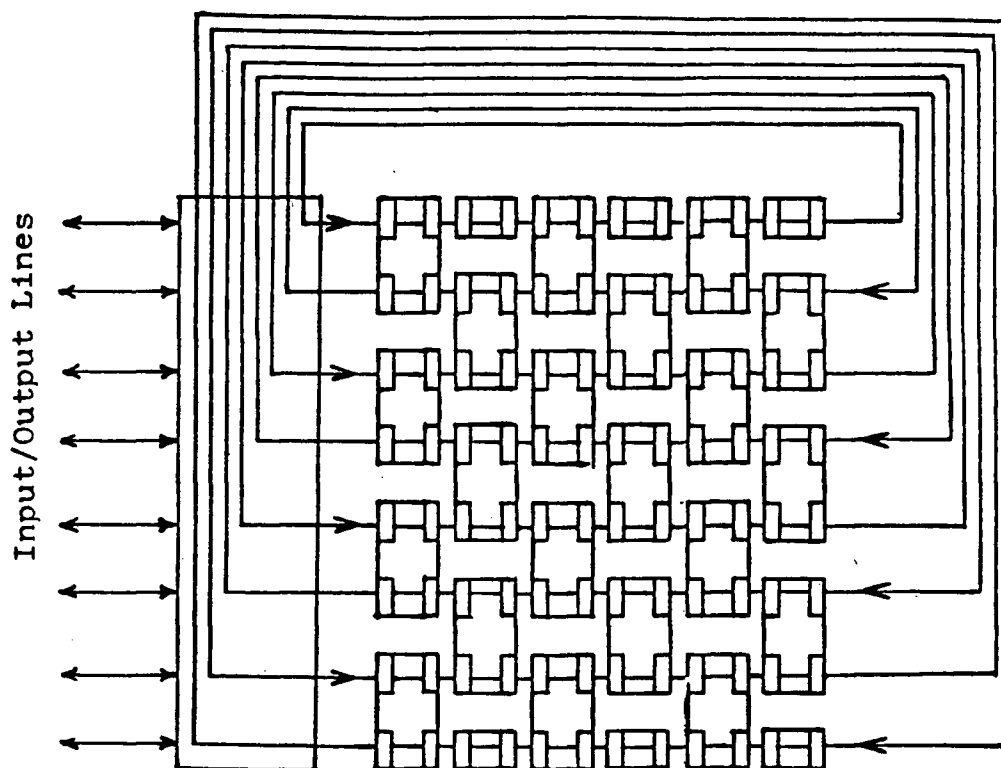
Sorting has been an important operation in business and computer engineering applications [13]. Many standard and novel sorting algorithms could be found in the literature [9-17]; some of them are optimal in time complexities, some in the number of comparators used while others lay emphasis on architectural designs, i.e., processor interconnections, data flow, control strategies and implementation technologies.

In this chapter, we present a parallel sorting network which embodies the concepts of both the cyclical architectures

and the systolic systems [3]. Systolic systems are characterized by their data flow pattern: once data are loaded from the memories, they and/or their intermediate results will move within the system along predetermined paths provided among the processing elements, and every element accepts and distributes data from and to its neighbours in a rhythmic fashion analogous to the pulsations in the arteries caused by the recurrent contractions of the heart.

A major advantage of such systems lies in the fact that processor-memory communications are involved only during the loading of the input data and unloading of the final results; therefore, there is no delay due to bus contentions and memory access conflicts during the computation time.

This study will demonstrate that a cyclical architecture coupled with systolic data movements can perform the useful task of sorting. Because of the highly regular interconnection, simple control and addressing structures, the area required by this design is very compact, and hence it is highly amenable to VLSI implementations. A description of the recirculating systolic sorter (RSS) will be given in Section 2, and the sorting algorithms, in Section 3. The constraints on RSS will be discussed in Section 4 while Section 5 will analyse the RSS algorithms and their timing complexities. The relative merits of RSS will be compared and discussed along with other designs in Section 6.



I/O switch Systolic array
Fig.II.1. The Recirculating Systolic Sorter (RSS).

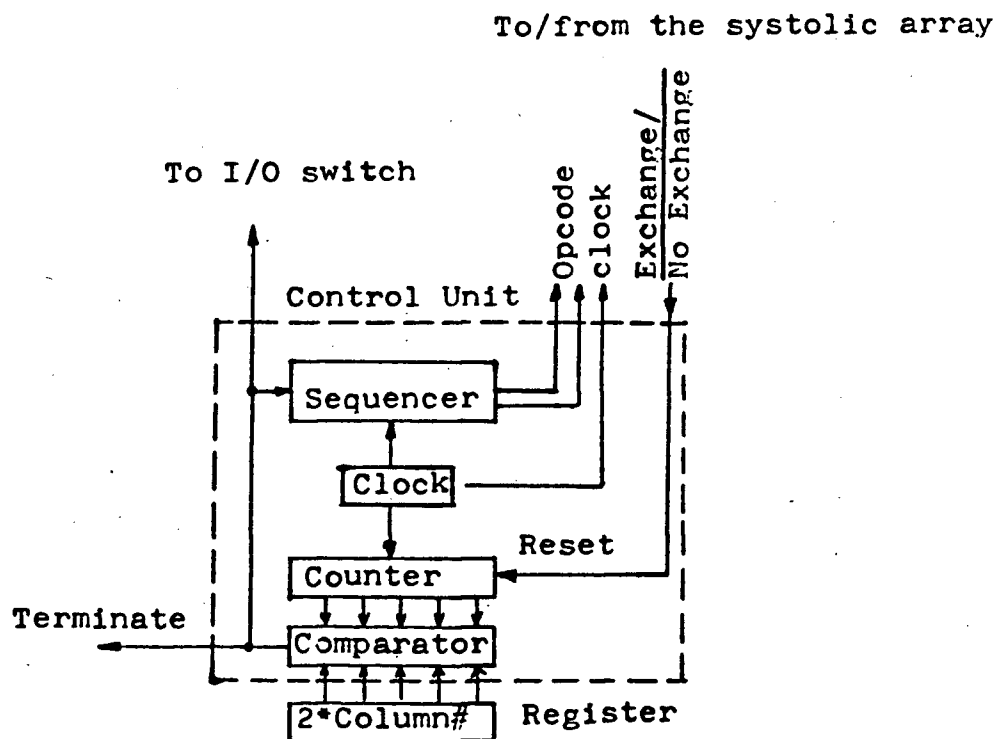


Fig.II.2. The control unit of RSS.

2. The Recirculating Systolic Sorter (RSS)

2.A. Network Description

A schematic diagram of the proposed sorter RSS is given in Fig.II.1. The RSS network consists of an array of "quadruple" comparators which are arranged into R rows and C columns. The whole array is articulated by $2 \cdot R$ circular loops as shown. Each of the quadruple comparators holds and sorts four input items during a comparison cycle, except those situated at the top and bottom and located in the odd-numbered columns of the array, where only either the upper or lower portion of these comparators is involved in the sorting process.

During the initial loading phase, all the loops are opened at the Input/Output switch and connected to the input lines; data items enter the network through the loops in a serial manner, with neighbouring loops shifted in opposite directions. After the network has been loaded, the input lines will be disconnected and all the loops will be closed. Before sorting commences, the comparator array has to be "marked" -- the sole purpose of which is to place a marker in a certain position within each loop, to indicate the beginning and end of that loop. The convention of marking adopted here is that the "head" of each loop will be associated with a marker, and the position on the right-hand side of the marker

will be regarded as the "tail" of that loop. The reader may refer to the examples given in Section 3 for illustrations; in these examples, asterisks are used to represent markers. Notice that the marking schemes -- i.e., the ways to place the markers on the array prior to the first cycle of sorting -- are different for the two examples, and they will be referred to as Scheme A and Scheme B respectively.

After the marking procedure, one of the proposed RSS algorithms will be applied to the array. During a comparison cycle, input data are compared and exchanged within the quadruple comparators. If a pair of data has to be exchanged, then their associated markers, if there are any, do not move with them but will remain where they are. However, between successive comparison cycles and when the data are shifted, the markers will be shifted along with the data with which they are associated.

A schematic diagram of the control unit used is presented in Fig.II.2. This unit generates the control signals (i.e., "Opcode" in Fig.II.2) to indicate one of the operations to be performed by the comparators: (1) Vertical-comparison; (2) Horizontal-comparison; (3) Diagonal-comparison and (4) Shift-operation. At the end of each comparison cycle, the control unit will test the status of the array (i.e., "Exchange/No-Exchange") to see whether any exchange has taken place during that cycle. It also has a cycle counter

which keeps track of the current number of consecutive "No-Exchange" cycles. In other words, the content of the counter is incremented upon entering a new cycle, and is reset whenever there is at least one exchange in that cycle; when the count reaches twice the number of columns (i.e., $\text{Count}=2*C$), a termination signal will be generated. At this stage, the input items have been sorted into a linear list. As demonstrated in the examples of Section 3, the first items of the sorted lists are accompanied by asterisks in the uppermost loops, and the last items are on the right-hand side of the asterisks in the lowest loops.

2.B. The Quadruple Comparator

The quadruple comparators have a higher logic density than the conventional, binary sorters used in other networks, but the number of input/output lines per comparator of the former is only slightly more than that of the latter. Fig.II.3 gives a sketch of the input/output configuration of a quadruple comparator.

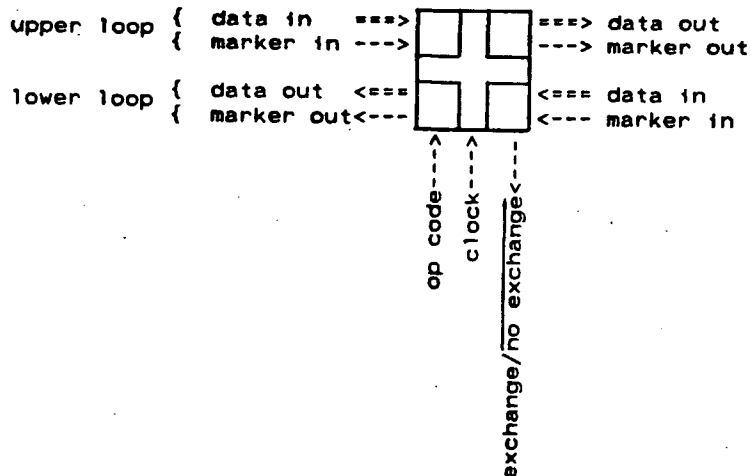


Fig.II.3. The schematic diagram of a quadruple comparator.

In addition to the two sets of input and two sets of output data lines, there are four single-bit lines used for shifting of markers along the two loops connected to the comparators: one line is for the clock signal, one line is used to indicate whether any exchange has taken place during the current comparison cycle, and two lines for the opcodes.

Essentially what a comparator unit accomplishes is the following. If it is located in an odd-numbered column, then it will push the smallest of the four data items which it holds to its upper-right neighbour; if it is in an even-numbered column, then it will retain the smallest and the largest items in its upper-right and lower-left positions

respectively. However, when markers are present inside the comparator, the situation becomes somewhat different and will be described in the next subsection.

2.C. The Comparison/Exchange/Shift Operations

For the convenience of illustration, the following symbols will be used throughout this chapter:

(i) Direction of comparison



(ii) Direction of shift



Fig.II.4. Symbols used for comparison and shift.

The direction of comparison is used to show the ordering of items after each comparison. In Fig.II.4(i), the solid arrow head indicates the position of the larger item for an ascending order; if, on the other hand, a descending order is desired, then the arrow head will indicate the smaller one. Without loss of generality, the ascending order will be assumed in this study. The open arrow of Fig.II.4(ii) is used to indicate the direction of movement for both the items

and the markers during the "Shift" operations.

The four operations performed by a comparator are depicted in Fig.II.5 and described below:

1. Vertical-comparison: The two items on the upper portion of the comparator are compared to the two at the bottom in parallel, with the directions of comparison pointing downward. The presence of markers is ignored.
2. Horizontal-comparison: Case(i) When no marker is inside the comparator: the two items on the right portion of the comparator are compared to the two on the left in parallel, with the directions of comparison pointing to the left; case(ii) When one or two markers are present: when a marker appears on the left portion of the comparator, the corresponding direction of comparison points to the right; otherwise it points to the left.

Note that in the horizontal comparisons, the direction of comparison always points from right to left according to the convention adopted, unless when both the head and the tail of a loop are involved in the comparison, i.e., when the marker appears on the left portion of the comparator, then the direction will be reversed. This reversal prevents the minimum and maximum items in a loop from crossing

over each other, and it is achieved by the action taken in Case (ii) above.

3. Diagonal-comparison: The two items on the upper portion of the comparator are compared to the two at the bottom in parallel, with the directions of comparison pointing downward and crossing each other.

At the first glance, the diagonal comparison involving the top-right and lower-left items seems redundant, because these two items are already in order after the vertical and horizontal comparisons; however, it is useful when two markers appear on the left portion of the comparator simultaneously. Furthermore, the top-left/bottom-right comparisons provides an exchange not provided by the combination of the vertical and horizontal comparisons.

4. Shift: Case(i) if the comparator is located in an even-numbered column, then its top two items will be shifted to the left and its lower two items to the right; case(ii) if the comparator is located in an odd-numbered column, its top two items will be shifted to the right and its lower items to the left.

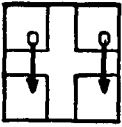
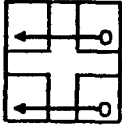
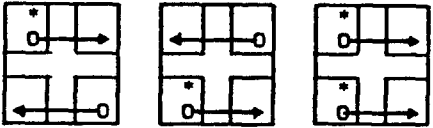
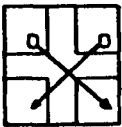
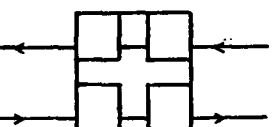
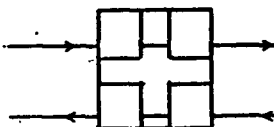
operation	actions	
1.Vertical_ Comparison time= t_c		
2.Horizontal_ Comparison	case(i) no marker is involved	case(ii) markers are involved
time= t_c		
3.Diagonal_ Comparison time= t_c		
4.Shift	case(i)for comparators in even columns	case(ii)for comparators in odd columns
time= t_s		

Fig.II.5. The four operations performed by the quadruple comparators.

3. The RSS Algorithms

3.A. Algorithm I

This algorithm involves only the "Vertical-comparison", "Horizontal-comparison" and "Shift" operations but not the "Diagonal-comparison", and is described in the following program fragment written in Pascal:

```

Program Recirculating-Systolic-Sorter;
:
:
Var
  Terminate          : boolean;
  Column#            : integer;
  Row#               : integer;
  Comparator#        : integer;
  Exchange           : boolean;
  Count-No-Exchange  : integer;
:
(*Initialization*)
:
While NOT Terminate do
(*enter next cycle of comparison*)
Begin
  for C:=1 to Comparator# do
    Begin
      Vertical-comparison;
      Horizontal-comparison;
    End;
  Check-Terminate;
  Shift;
End;
:

```

(Algorithm I)

The procedure "Check-Terminate" manipulates the following global variables:

1. "Exchange" - This boolean variable is always reset to be "False" before a new comparison cycle commences, and is set to be "True" if any exchange

takes place during the cycle.

2. "Count-No-Exchange" - This variable keeps track of the number of consecutive cycles which have no exchange, and is reset to zero whenever "Exchange" equals "True".
3. "Terminate" - This boolean variable controls the "WHILE-DO" loop, and is set to be "True" if the following condition is satisfied:

Condition(1) (for termination):

$$\text{Count-No-Exchange} \geq 2 * \text{Column\#}$$

3.B. Algorithm II

This algorithm is similar to Algorithm I except that the "Diagonal-comparison" operation is included in its "WHILE-DO" loop:

```

Program Recirculating-Systolic-Sorter;
:
:
While NOT Terminate do
(*enter next cycle of comparison*)
Begin
  For I:=1 to Comparator# do
    Begin
      Vertical-comparison;
      Horizontal-comparison;
      Diagonal-comparison;
    End;
  Check-Terminate;
  Shift
End;
:

```

(Algorithm II)

3.C. Examples

Two examples using three columns, three rows and eight comparators (i.e., $C=3$, $R=3$, $P=8$) are presented in Fig.II.6 and Fig.II.7. After the initial loading and marking procedures, Algorithm I and II are applied to the first and second examples respectively. The contents of the comparator array are shown for the first and the last two cycles. Both input lists are sorted into the ascending order. At the end of the last cycle, the minimum of each loop is indicated by the markers and the direction of increasing values is from right to left. All the numbers in a given loop are greater than or equal to those in the next loop above.

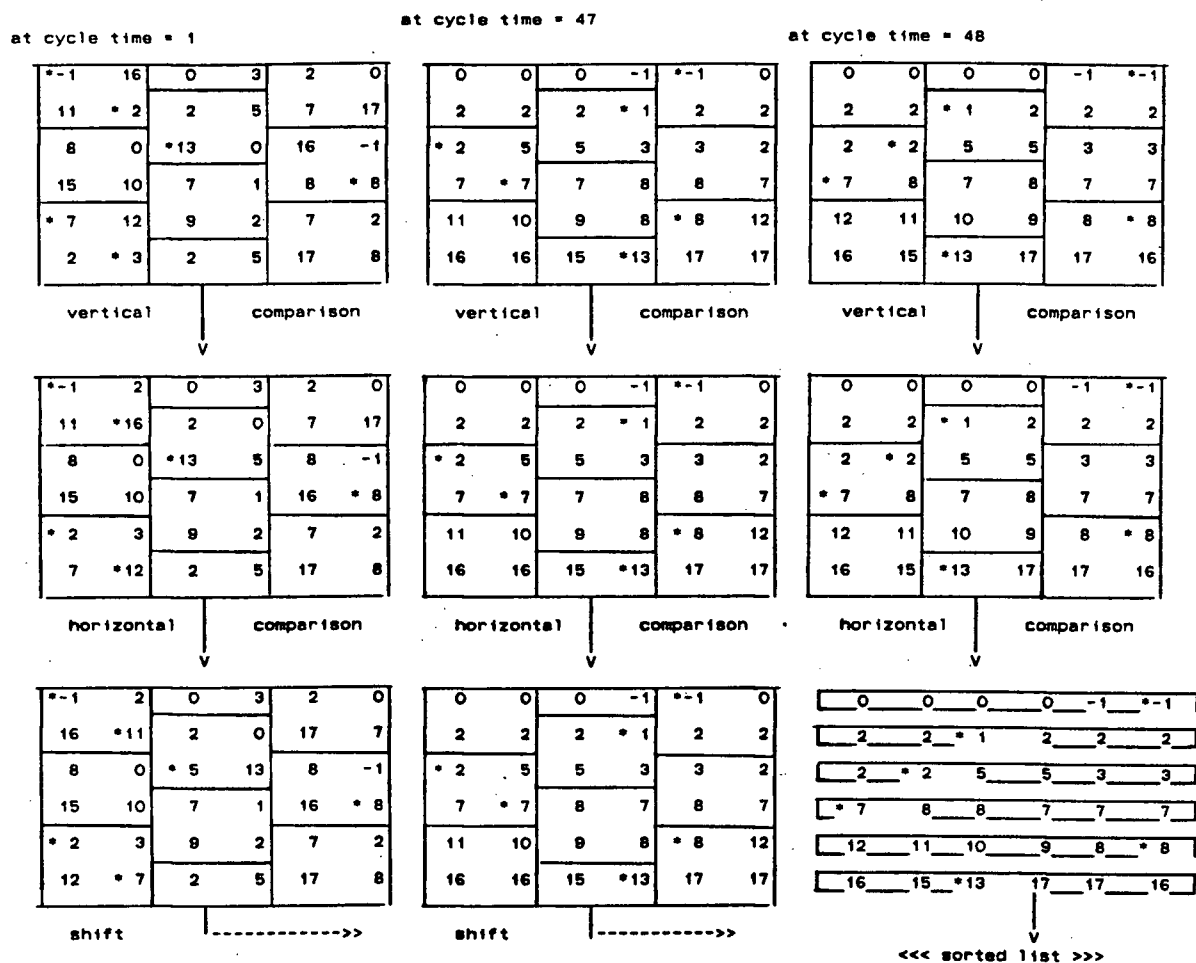


Fig.II.6. An example to illustrate RSS Algorithm I and Marking Scheme A.

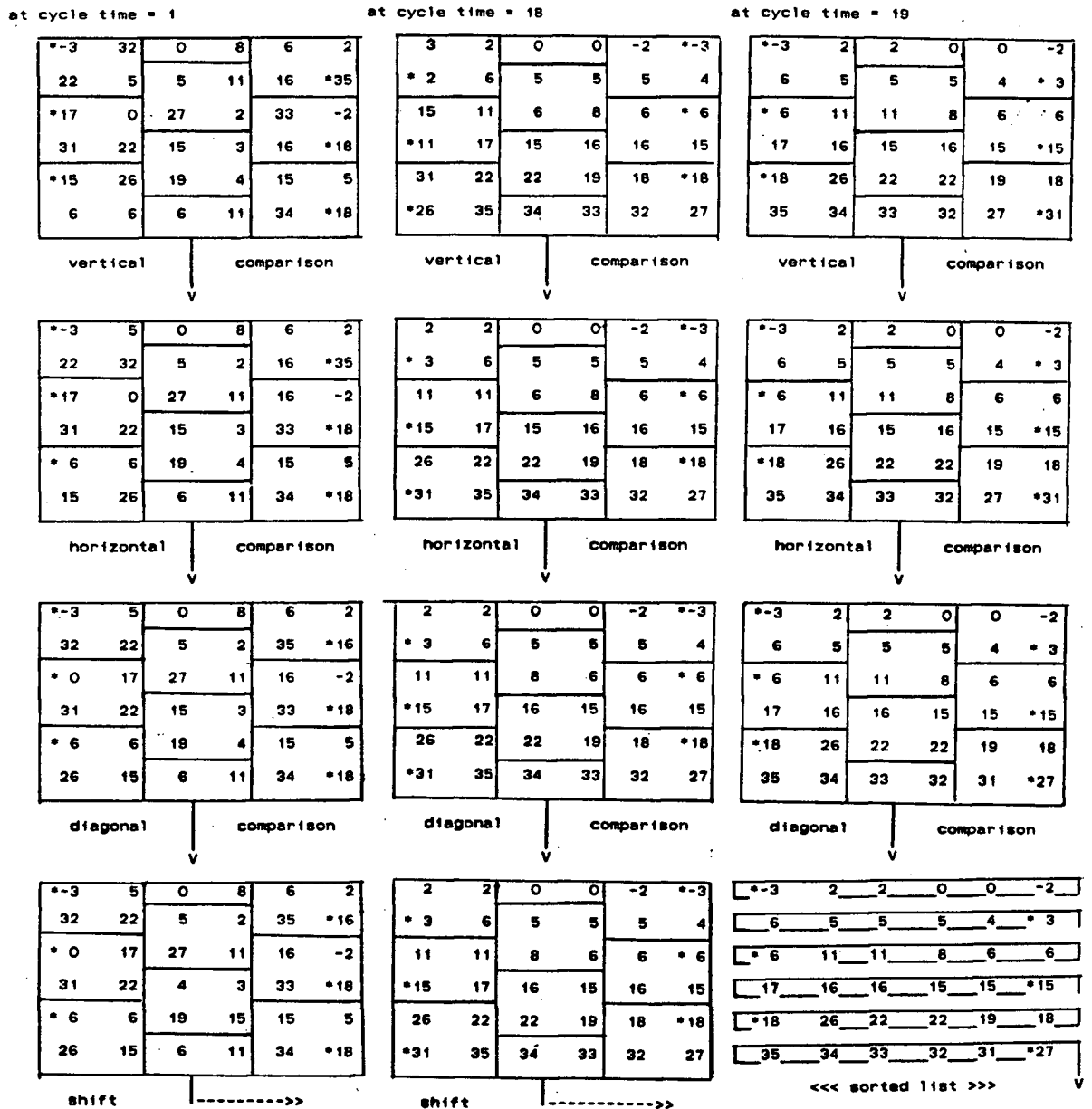


Fig.11.7. An example to illustrate RSS Algorithm II and Marking Scheme B.

4. Operational Constraints

4.A. Constraints on the Size of RSS

Most sorting networks impose certain constraints on the size of the networks. For examples, Batcher's bitonic sorter [13] requires that the number of its input lines be a power of two, and some mesh sorters [14,15] work on square arrays only. The basic constraint of the RSS array appears to be less stringent:

Requirement(1):

Column# >2

Row# >1

Further constraints may or may not be required depending on the marking schemes used: In Scheme A and B to be described below, Requirement(1) is sufficient to guarantee correct operations of both RSS algorithms when Scheme A is used, but an additional constraint -- which will be given later on -- on the size of RSS will be needed when Scheme B is used.

4.B. Marking Scheme A

It is observed from our simulation studies that only certain ways of marking the array can guarantee correct results, and one such way is given below.

Marking Scheme A:

The initial marker position, $M(i)$, of loop i is:

$$M(i) := 4*i - 2 + 1 - M(i-1)$$

where $i=1,2,\dots,2*\text{Row\#}-1$, $0 < M(i) \leq 2*\text{Column\#}$, and $M(0)$ can be any value in the range of $M(i)$.

Scheme A is applied to the example of Fig.II.6, where $M(0)=1$, $M(1)=3-1=2$, $M(2)=5-2=3$, $M(3)=9-3=6$, $M(4)=7-6=1$, $M(5)=3-1=2$, and the pattern repeats. If there are only two columns, then $M(3)=6\text{MOD}(2*\text{Column\#})=2$. The rationale behind this scheme will be explained in Section 5.B.

4.C. Marking Scheme B

In the second scheme, the markers are placed along the two sides of the comparator array, as demonstrated in

Fig.II.7. This method is simpler and we may use the Input/Output lines to insert the markers; and also, the retrieval of the final sorted list is easier than when Scheme A is used.

However, this scheme requires that the number of columns of the RSS array be twice an odd integer, or the next higher integer of that value:

Marking Scheme B:

$$\begin{aligned} M(i) &:= 1, && \text{for } i=\text{even} \\ &:= 2*\text{Column\#}, && \text{for } i=\text{odd}. \end{aligned}$$

Requirement(2) (for Scheme B only):

$$\begin{aligned} \text{Column\#} &:= 2*(\text{An odd integer}) && ,\text{or} \\ &:= 2*(\text{An odd integer}) + 1 \end{aligned}$$

5. Analysis of the RSS Algorithms

5.A. Analogy with the Odd-Even Transportation Sort

The RSS algorithms bear some resemblance to the Odd-Even Transportation Sort [11]; therefore, a brief explanation of the Odd-Even sorter would be helpful in analysing the RSS design:

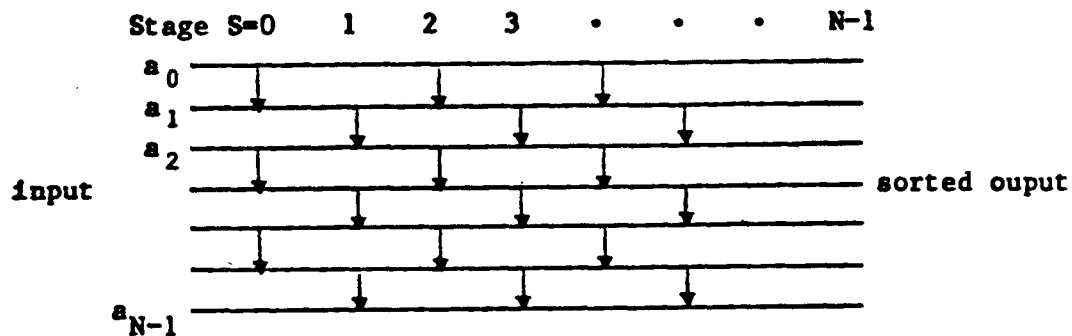


Fig.II.8. The Odd-Even Sorter.

In Fig.II.8, the appearance of an arrow indicates the presence of a conventional, binary sorter situated at that position. An item $a(j)$ will be compared to another item $a(j')$ at stage s if,

$$j' = j + (-1)^{j+s} \dots\dots\dots (II.1)$$

where all j' , j and s are greater than or equal to zero and are less than N , where N is the number of input lines.

The value of j' will alternate between $(j+1)$ and $(j-1)$ when s is incremented. This sorter guarantees correct sorting of N items in N cycles [11], but it requires a total of $N(N-1)/2$ sorters; it is therefore impractical if N is large.

5.B. Correctness of the RSS Algorithms and Marking Schemes

In this section, we will first prove that the RSS algorithms are correct, and then the two marking schemes will be derived. In Lemma (II.1), we will examine the effects of the RSS algorithms on each loop of the RSS array, temporarily ignoring the interactions among the loops; then Theorem (II.1) will show that with these interactions, a complete sorting process can be achieved.

Lemma (II.1): The Odd-Even Transportation Sort is performed on each of the RSS loop when either Algorithm I or II is applied to the RSS array.

Proof: Let us consider three indexes i, j and J on the RSS array. As demonstrated in Fig.II.9, $i(=0,1,..2*R-1)$ indexes the loops of the RSS array; $J(=1,2,..2*C)$ indexes the fixed positions of the array; and for loop i , $M(i)$ indicates the initial position of the marker of the loop, and $j(=0,1,...2*R-1)$ indicates the distance of a position away from the marker. Because the markers are shifted with time, j is therefore a function of time and is related to other indexes as follows:

$$j = [(M(i) + 2C - J) + (2C + (t * (-1) ** i) \bmod 2C)] \bmod 2C \quad \dots (II.2')$$

$$= [4C + M(i) - J + (t * (-1) ** i) \bmod 2C] \bmod 2C \quad \dots (II.2)$$

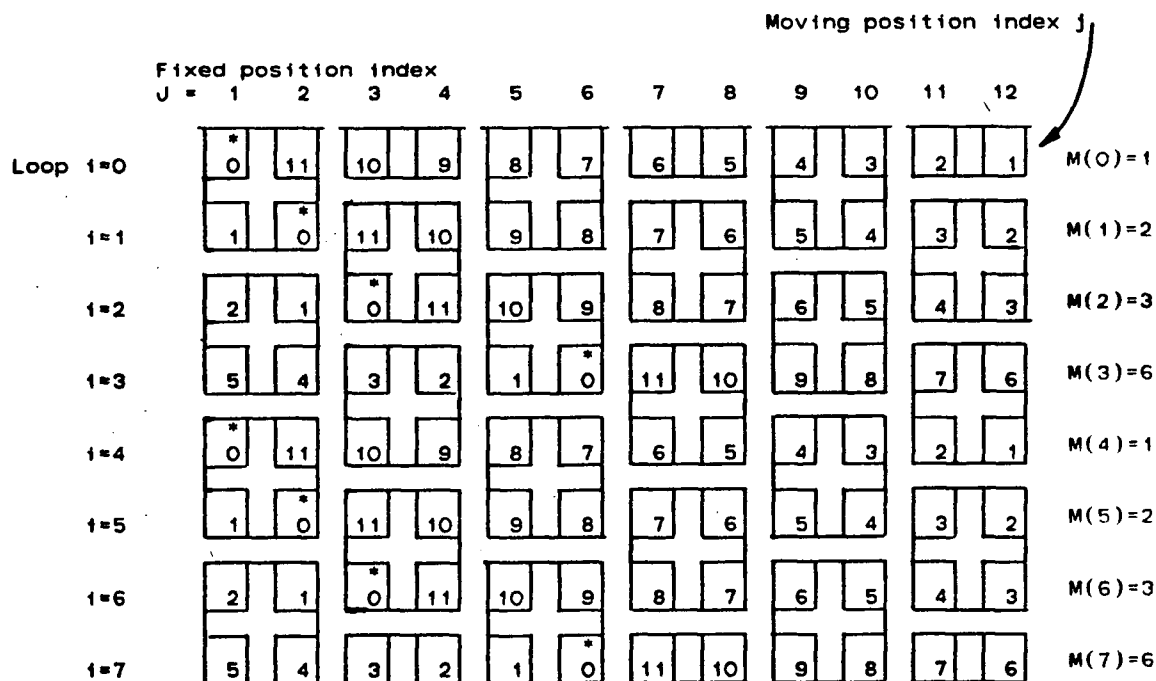


Fig.II.9. The three indexes: i , j , and J , and the initial marker position $M(i)$.

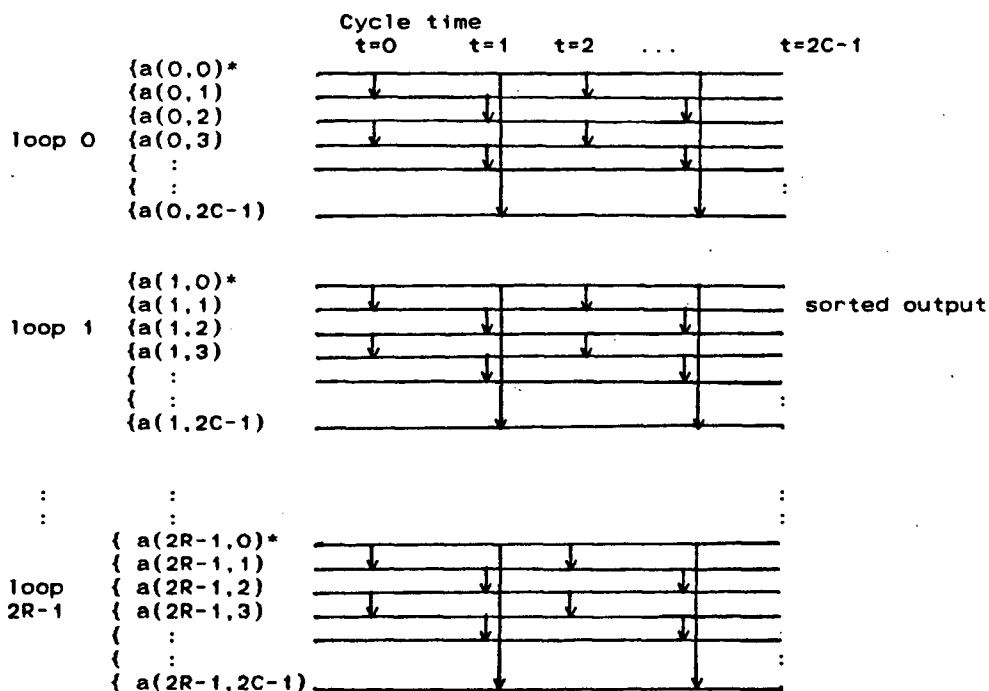


Fig.II.10. The horizontal comparisons carried out on the RSS array.

The first composite term in expression (II.2') shows the effect of the initial marker's position on j , and the second composite term is due to the effect of time indexed by t . The modulo functions are used to trim the values of t and j because both of them are repetitive with a period of $2C$. The reader may verify easily that expression (II.2) is correct from the example of Fig.II.9.

Having established the relationship among the indexes, we will now derive several expressions to relate a pair of data items $\{a(i,j), a(i',j')\}$ involved in a comparison. First, let us consider the horizontal comparison. In Fig.II.9, $a(i,J)$ is always compared to $a(i,J')$ where

$$J' = J - (-1)**J \quad \dots\dots\dots(II.3)$$

For item $a(i,j)$, j is related to other indexes as in (II.2):

$$j = [4C+M(i)-J+(t*(-1)**i) \bmod 2C] \bmod 2C$$

Similarly for item $a(i',j')$:

$$\begin{aligned} j' &= [4C+M(i')-J'+(t*(-1)**i') \bmod 2C] \bmod 2C \\ &= [4C+M(i')-J+(-1)**J+(t*(-1)**i') \bmod 2C] \bmod 2C \end{aligned}$$

For a horizontal comparison, i equals i' , therefore j' reduces

to:

$$j' = j + (-1)^{**}J \dots\dots\dots(II.3')$$

Again from expression (II.2):

$$J = -j + 4C + M(i) + (t * (-1)^{**}i) \text{MOD } 2C + 2KC$$

where K is any positive integer such that J will be positive. Substituting J into (II.3'), we obtain the expression for j', where a(i',j') will be compared to a(i,j) horizontally,

$$j' := -j + (-1)^{**}[j + M(i) + 2C + (t * (-1)^{**}i) \text{MOD } 2C] \dots\dots(II.4)$$

Within loop i, M(i) and $(-1)^{**}i$ are constants, therefore j' will alternate between $(-j-1)$ and $(-j+1)$ as t increases. By comparing (II.4) and (II.1), we can see that the "Horizontal-comparison" when coupled with the "Shift" operations, will perform the Odd-Even Sort as far as loop i is concerned, and therefore items within a loop can be sorted within $2 * C$ cycles. This point is further illustrated in Fig.II.10.

Q.E.D.

Theorem (II.1): The entire RSS array is capable of sorting with the combination of either Algorithm I or Algorithm II, and either Marking Scheme A or Scheme B.

Proof: In addition to the Odd-Even comparisons within a loop, the RSS also compares the items of any two adjacent loops by means of the "Vertical-comparison" and "Diagonal-comparison"; the purpose of these operations is to move smaller items upward and larger items downward. It is easily discernible that, if comparisons are provided between the head (i.e., $a(i+1,0)$) of one loop, and the tail (i.e., $a(i,2C-1)$) of the next higher loop, then odd-even sort will be carried out on the entire RSS array.

Therefore, the proof of this theorem is reduced to the proof that the "head-tail" comparisons are provided by the combination of the algorithms and the marking schemes. Let us consider the "Vertical-comparison" between a pair of items $\{a(i,j), a(i',j')\}$. In Fig.II.9, note that $a(i,j)$ will be always compared to $a(i',j')$ if,

$$i' = i - (-1)^{(i+[J/2])} \dots\dots\dots (II.5)$$

From expressions (II.2,4 and 5), we can obtain the position J where the head and tail of any two loops meet:

$$j = 2C - 1, \Rightarrow 4C + M(i) - J + (t * (-1)^i) \text{MOD } 2C = 2KC + 2C - 1 \dots (II.6)$$

$$j' = 0, \Rightarrow 4C + M(i') - J + (t * (-1)^{(i+1)}) \text{MOD } 2C = 2K'C \dots\dots (II.7)$$

Combining expressions (II.6 and 7), we obtain,

$$8C + M(i) + M(i') - 2J = (K + K') * 2C + 2C - 1$$

$$\Rightarrow J = K''C + (M(i) + M(i') + 1) / 2 \dots\dots\dots(II.8)$$

where K and K' are integers such that $0 \leq j < 2C$, and K'' equals either -1 , 0 or 1 because $1 \leq J \leq 2C$. Expression (II.8) means that the tail of loop i will be compared to the head of loop $(i+1)$ at either halfway between $M(i)$ and $M(i')$, i.e., $J = (M(i) + M(i') + 1) / 2$, or $J = (M(i) + M(i') + 1) / 2 + C$, depending on whether there is any comparator situated at these locations. From expression II.8,

$$M(i) + M(i+1) + 1 = 2(J - K''C)$$

$$\Rightarrow M(i) + M(i+1) = 2(J - K''C) - 1 \dots\dots\dots(II.9)$$

Expression (II.9) gives rise to another requirement for the marking of the RSS array:

Requirement(3):

(for marking schemes other than Scheme A and B)

$$M(i) + M(i+1) = \text{An odd integer}$$

This requirement will ensure that the tails and heads of the loops will be compared by the vertical comparisons.

It is automatically satisfied when Marking Scheme A or B is used, but has to be considered for other marking schemes.

Q.E.D.

Now we will derive Marking Scheme A. From (II.5),

$$\begin{aligned} i' &= i+1 \text{ and } i' = i - (-1)^{i+1} \lceil J/2 \rceil \\ \Rightarrow i + \lceil J/2 \rceil &= \text{An odd integer} \\ \Rightarrow \lceil J/2 \rceil &= (\text{An odd integer}) - i \end{aligned}$$

Case(i) at $i = \text{even}$, $\lceil J/2 \rceil = \text{odd}$; therefore,

$$\begin{aligned} \Rightarrow J &= 2 * (\text{An odd integer}), \text{ or} \\ &= 2 * (\text{An odd integer}) - 1 \quad \dots\dots\dots (\text{II.10}) \end{aligned}$$

from expressions (II.9 and 10),

$$\begin{aligned} \Rightarrow M(i+1) &= 4 * (\text{An odd integer}) - 2K^{\lceil J/2 \rceil - 1} - M(i), \text{ or} \\ &= 4 * (\text{An odd integer}) - 2K^{C-3} - M(i) \quad \dots (\text{II.11}) \end{aligned}$$

Case(ii) at $i = \text{odd}$, $\lceil J/2 \rceil = \text{even}$; therefore,

$$\begin{aligned} \Rightarrow J &= 2 * (\text{An even integer}), \text{ or} \\ &= 2 * (\text{An even integer}) - 1 \quad \dots\dots\dots (\text{II.12}) \end{aligned}$$

from expressions (II.9 and 12),

$$\begin{aligned} \Rightarrow M(i+1) &= 4 * (\text{An even integer}) - 2K^"C - 1 - M(i), \text{ or} \\ &= 4 * (\text{An even integer}) - 2K^"C - 3 - M(i) \dots \dots (II.13) \end{aligned}$$

We could obtain Scheme A by setting $K^"=0$ in expressions (II.11 and 14):

$$M(i+1) = 4 * i - 2 + 1 - M(i)$$

Or, equivalently,

$$M(i) = 4 * i - 2 + 1 - M(i-1)$$

which is Scheme A and where $1 \leq M(i) \leq 2C$, for $i := 1, 2, \dots, 2R$.

To derive Scheme B, let

$$\begin{aligned} M(i) &:= 1, & \text{for } i = \text{even} \\ &:= 2C, & \text{for } i = \text{odd} \end{aligned}$$

Case(i) At $i = \text{odd}$, from expressions (II.8 and 12),

$$\begin{aligned} J &= K^"C + (1 + 2C + 1) / 2 = 2 * (\text{An even integer}), \text{ or} \\ &= 2 * (\text{An even integer}) - 1 \\ \Rightarrow J &= K^"C + C + 1 \in \{3, 4, 7, 8, \dots\} \dots \dots (II.14) \end{aligned}$$

out of the three possible vaules of $K^"$, i.e., -1, 0, and +1,

only $K''=0$ can satisfy both expression (II.14) and $(1 \leq J \leq 2C)$;
therefore,

$$J=C+1 \in \{3,4,7,8,\dots\}$$

$$\Rightarrow C \in \{2,3,6,7,\dots\}$$

$$\Rightarrow C:=2*(\text{An odd integer}), \text{ or}$$

$$:=2*(\text{An odd integer})+1 \quad \dots\dots\dots(\text{II.15})$$

Case(ii) At $i=\text{even}$, from expressions (II.8 and 10),

$$J:=K''C+(2C+1+1)/2$$

$$=K''C+C+1 \in \{1,2,5,6,\dots\} \quad \dots\dots\dots(\text{II.16})$$

both $K''=0$ and $K''=-1$ can satisfy expression (II.16) and
 $(1 \leq J \leq 2C)$ simultaneously:

$$\text{when } K''=0, \quad C \in \{0,1,4,5,\dots\} \quad \dots\dots\dots(\text{II.17}')$$

$$\text{when } K''=-1, \quad C = \text{Any positive integer} \quad \dots\dots\dots(\text{II.17})$$

For all values of i , both expressions (II.15 and 17) can be
satisfied simultaneously by the requirement below:

$$C := 2*(\text{An odd integer}), \text{ or}$$

$$:= 2*(\text{An odd integer})+1$$

$$\Rightarrow C \in \{2, 3, 6, 7, \dots\}$$

which is Requirement (2) of Scheme B.

Now let us consider the diagonal comparisons, and we will show that Requirement (3) can actually be waived when Marking Scheme B is used with Algorithm II.

Again, from Fig.II.9, items $a(i, j)$ and $a(i', j')$ will be compared diagonally if,

$$J' = J - (-1)^{**} J \quad \dots\dots\dots (II.3)$$

$$i' = i - (-1)^{**} (i + \lceil J/2 \rceil) \quad \dots\dots\dots (II.5)$$

Converting J and J' into j and j' using expressions (II.2 and 3):

$$j = [4C + M(i) - J + (t * (-1)^{**} i) \text{ MOD } 2C] \text{ MOD } 2C$$

$$j' = [4C + M(i') - J + (-1)^{**} J + (t * (-1)^{**} i') \text{ MOD } 2C] \text{ MOD } 2C$$

The heads and tails of the loops will be compared by the diagonal comparisons if,

$$j = 2C - 1$$

$$j' = 0$$

$$i' = i + 1$$

Substituting these values into the above expressions, we get,

$$j=2C-1, \Rightarrow 4C+M(i)-J+(t*(-1)**i) \text{MOD } 2C=2KC+2C-1$$

$$j'=0, \Rightarrow 4C+M(i+1)-J+(-1)**J+(t*(-1)**(i+1)) \text{MOD } 2C=2K'C$$

Adding up the two expressions,

$$8C+M(i)+M(i+1)-2J+(-1)**J=(K+K'+1)*2C-1$$

$$\Rightarrow M(i)+M(i+1)=2J-2K''C-1-(-1)**J$$

$$=2(J-K'')C-1-(-1)**J$$

$$=2(J-K'')C, \text{ or } 2(J-K'')C-2$$

$$=\text{An even integer.}$$

The last result shows that Requirement (3) can be waived when Scheme B is used with Algorithm II, because if $M(i)+M(i+1)$ equals an odd integer, then the head-and-tail comparisons will be provided by the "Vertical-comparison", but if it equals an even integer, then it will be provided by the "Diagonal-comparison" as demonstrated above.

The various requirements for the marking schemes are summarized in Table.II.1.

Table II.1 - Requirements of the RSS marking schemes.

Marking Scheme	Algorithm I	Algorithm II
A	Requirement(1)	Requirement(1)
B	Requirements(1)&(2)	Requirements(1)&(2)
others	Requirements(1)&(3) and others to be derived from expressions II.5 and II.8.	

5.C. Correctness of the Termination Methods

If the RSS array is correctly marked and Requirements (1), (2) and (3) are duly met, then Condition (1) of Section (2.C) is sufficient to guarantee proper termination. The reason is that, as we can see from expressions (II.2,4 and 5), the comparison pattern repeats every 2C cycles; if there is no exchange in the most recent consecutive 2C cycles, then there will be no further exchanges in the subsequent comparisons, meaning that the sorting process must have terminated.

5.D. Timing Complexities

The RSS simulation program is listed in Appendix B for reference. Input parameters to the simulator include the numbers of rows and columns -- these two numbers determine the

total number of items to be sorted -- and the initial seed value for the generation of the input list; at the end of each simulation run, the simulator will produce the sorted list as well as the number of sorting cycles needed.

Fig.II.11 show the numbers of cycles needed by RSS Algorithm I to sort on arrays with various combinations of numbers of rows and columns. These simulation results indicate that with Algorithm I, the average number of cycles needed to sort a random set of N items is bound by the line N , and approaches $N/2$ as N increases.

When Algorithm II is used, the number of cycles needed will be much smaller -- due to the presence of diagonal comparisons, and the examples of Fig.II.6 and II.7 help illustrate this point. However, the actual speeds of Algorithm II may or may not exceed that of Algorithm I, because its comparison cycle include the diagonal comparison and hence its cycle time will be longer.

N_1 = Number of comparison cycles

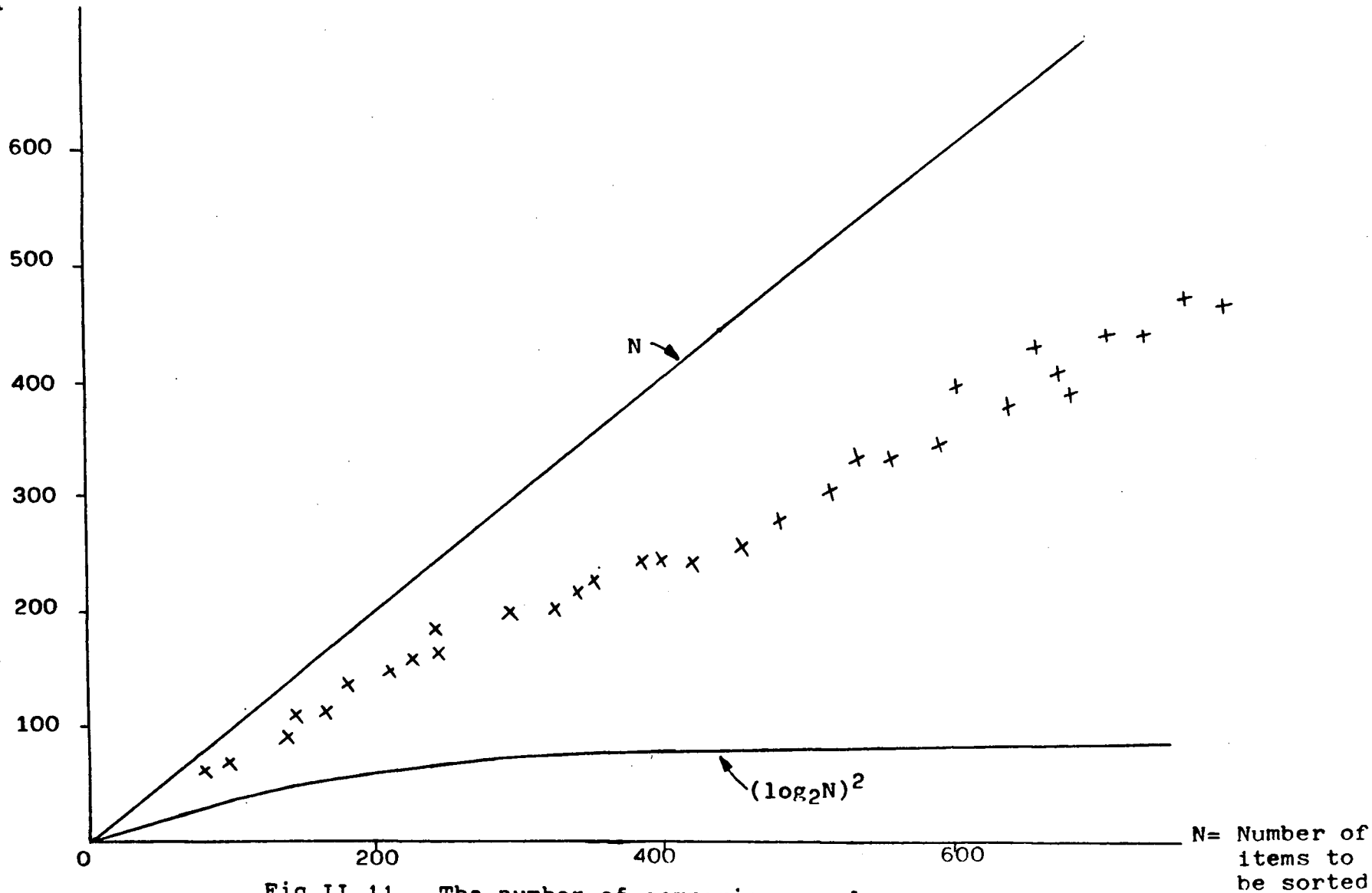


Fig.II.11. The number of comparison cycles versus the number of items to be sorted (Algorithm I).

6. Discussions

Since sorting is such a common and necessary operation in computer applications, there are dozens of sorting algorithms described in the literature. In this chapter, we have presented two similar sorting algorithms which apply the systolic idea to a cyclical architecture, and the functional design of a sorter (RSS) based on these algorithms has also been suggested. Our primary goal is to look into the design of a special-purpose VLSI chip that can be attached to a host computer such as the one envisioned by Foster and Kung [19]:

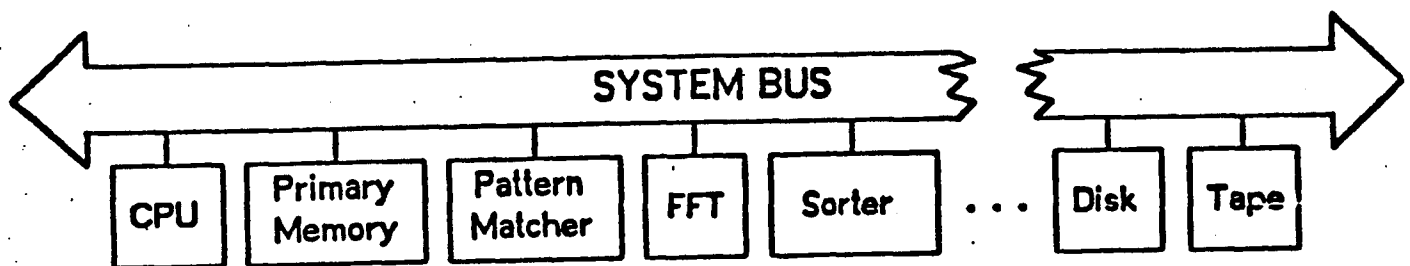


Fig.II.12. A general-purpose computer system with special-purpose chips attached [19].

Undoubtedly, the usefulness of the sorter is not limited to scientific computation, it could also be used in office information systems and relational data base machines.

With the stated goal in mind, we now compare our proposal with some existing ones using the following criteria: (a) time complexity; (b) hardware complexity and (c) control

complexity.

(a) Time Complexity: In Table.II.2, the sorting times of some existing algorithms could be divided into four categories, namely, $O(\log N)$, $O((\log N)^2)$, $O(N^{0.5})$ and $O(N)$, where N is the number of items to be sorted. Muller and Preparator's algorithm [10] is in the faster category, but it requires a discouraging number of comparators, $O(N^2)$. Batcher's bitonic sorter [13] and the perfect shuffle [9] are both in the $O((\log N)^2)$ category, and they are characterized by the shuffle-exchange type of interconnections. The two mesh sorting schemes sort N^2 items on a $N \times N$ mesh with approximately $O(N)$ time, therefore they belong to the $O(N^{0.5})$ category. Nassimi and Sahni's mesh sorting scheme [14] is based on Batcher's bitonic merge algorithm and it needs approximately $14N$ routing steps and $2 \cdot \log N$ compare-exchange steps on a $N \times N$ mesh; moreover, it requires that the input subfiles be pre-sorted. Thompson and Kung's mesh sorting scheme [15] needs roughly $6N + O((N^{2/3}) \log N)$ routing steps and $N + O((N^{2/3}) \log N)$ compare-exchange steps. The RSS algorithms belong to the $O(N)$ category, but because of their simpler control structures and near-neighbour type of data movements, their actual sorting times might be less than those of the mesh sorting schemes which require more complex control

and data movements.

- (b) Hardware complexity: Sorters with shuffle-exchange type of interconnections are not well-suited to VLSI implementations, because shuffle-exchange networks have a very low degree of regularity and modularity, and require wires of various lengths. It has been shown by Thompson [18] that at least $O((N^2)/(\log N)^2)$ chip area is required to lay out an N -vertex shuffle-exchange network -- this is a serious drawback when N is large. On the other hand, because the interconnection patterns required by the mesh and the RSS algorithms are highly regular and repetitive, these two types of sorters could be fabricated easily by replicating the circuits of a single comparator unit for the entire arrays.
- (c) Control complexity: The logic of the various operations (i.e., Horizontal-comparison, Vertical-comparison and Diagonal-comparison) can be hardwired into each of the quadruple comparator, and the control unit shown in Fig.II.1 simply broadcasts the sequence of these operations to all the comparators. The control structure required by the RSS algorithms is therefore comparable to that required by the Batcher's bitonic sorters, and is much simpler than that required by the mesh sorters. The

simplicity of the RSS controller is another important factor when the VLSI implementation is concerned.

Most other sorting networks impose certain non-trivial constraints on their sizes; for examples, the Batcher's sorter and the perfect shuffle network require that the number of input lines be a power of two, and the mesh sorting algorithms operate on square arrays. The constraints of the RSS algorithms (see Table.II.1) appear to be less stringent in this respect.

In summary, although the RSS design is not optimal in every aspect, it is highly amenable to VLSI implementations as far as its hardware and control complexities are concerned.

Table II.2 - Complexities of Sorting Networks+

Method	#Input	#Comparators	Time	Interconnection	Control
Batcher's Bitonic Sorter[13]	N	$O(N\{\log N\}^{**2})$	$O(\{\log N\}^{**2})$	high	low
Muller & Preparator's[10]	N	$O(N^{**2})$	$O(\log N)$	low	high
Perfect Shuffle[9]	N	$O(N)$	$O(\{\log N\}^{**2})$	high	low
Thompson & Kung's Mesh Sort[15]	NxN	NxN mesh	$O(N)++$	low	high
Nassimi & Sahni's Mesh Sort[14]	NxN	NxN mesh	$O(N)++$	low	high
RSS	N	$O(N)$	$O(N)++$	low	low

Notes: + In terms of amenability to VLSI implementations.

++ Please see discussions in Section II.6.

Chapter III. A Novel Loop-Structured Switching Network (LSSN)

1. Introduction

Many large-scale computer applications such as image processing, weather forecasting and ballistic missile defence systems, require execution rates of more than one billion instructions per second. With the advent of VLSI technologies, it is feasible and more flexible to construct such large-scale systems by interconnecting hundreds or even thousands of off-the-shelf processing and storage devices, to work in a co-operative manner. Although several existing networks can provide the required communication bandwidth among these devices, they are expensive to build and difficult to expand. For examples, the switch counts of a $N \times N$ crossbar and a $N \times N$ baseline [20] are $O(N^2)$ and $O(N \log N)$ respectively; for $N=1024$, the crossbar would require more than a million switches while the baseline would need about five thousand of them. Another disadvantage of using large number of switches is that of system reliability -- the networks are more likely to fail when more switches are used.

In this chapter, we introduce a novel loop-structured switching network (LSSN) which overcomes the above problems. The main feature of LSSN is its cyclical connections; and it only requires $N/2$ two-by-two switches for interconnecting N pairs of transmitting and receiving devices; therefore, it is

very attractive for large-scale, heterogeneous systems made up of many devices.

From the structural and functional points of view, LSSN is a packet-switched, multi-staged, blocking network with distributed control. In the next section, we will present its connection function, addressing and routing algorithms. In Section three, several important properties of LSSN will be revealed, and the causes of and method to avoid deadlocks will be discussed. In Section four, the results of our simulation studies and performance evaluations will be presented. Discussions and topics for further work are provided in Section five, and the LSSN simulation program are listed in Appendix C for reference.

2. Network Topology

Networks using $(\log L)$ stages of two-by-two switches -- where L is a power of two -- are well-known [20,21,22,25,26]. Traditionally, they are used to connect L input to L output terminals, i.e., for interconnecting L transmitters to L receivers. Feedback paths are sometimes provided to route the information back from the output side to the input side, thus forming loops. LSSN is also based on the concept of feedback loops, but it differs from others in that all its switches could be used as both entry and exit stations for data transmission and reception. With L loops -- where L is a power of two and at least equal to four -- LSSN can connect

up to $N=L\log L$ pairs of transmitters (Trs) and receivers (Rrs), using only $N/2$ switches -- this feature renders it attractive for large values of N . An example with $L=16$ and $N=64$ is illustrated in Fig.III.1.

2.A. Addressing Scheme and Connection Function

The following description can be easily understood if the readers refer to the example of Fig.III.1, in which all the switches have been set to the "Straight-through" connection; a loop is defined as a closed path in this configuration. For a LSSN with L loops, each of the loops is labeled with $L'=\log L$ bits of code, i.e., ℓ_L, \dots, ℓ_1 , which is the binary representation of an integer in the closed range $[0, L-1]$. The switches are arranged into L' stages each of which is labeled with $S'=\lceil \log L' \rceil$ bits of binary digits represented as $\Delta_S, \dots, \Delta_1$.

The output links of a switch at the s -th stage would be assigned the following addresses:

$$\text{Left Output link} = \Delta_S, \dots, \Delta_1 \ell_L, \dots, \ell_{s+1} 0 \ell_{s-1} \dots \ell_1$$

$$\text{Right Output Link} = \Delta_S, \dots, \Delta_1 \ell_L, \dots, \ell_{s+1} 1 \ell_{s-1} \dots \ell_1$$

These addresses are obtained by concatenating the stage and loop labels together, with the s -th bit of the address of the left output link set to "0" and that of the right output link

set to "1". One could verify this scheme on the example of Fig.III.1.

Consider a switch located in the s -th stage, and suppose one of its output links is part of the loop l_L, \dots, l_1 , then it realizes the following connection function:

$$\text{LSSN}_s(l_L, \dots, l_s, \dots, l_1) = l_L, \dots, \bar{l}_s, \dots, l_1, \text{ where } \bar{l}_s \text{ is the one's complement of } l_s.$$

The connection function states that at the s -th stage, any two loops with labels differing only in their s -th bits will be connected by a switch at that stage.

2.B. Routing Scheme

In LSSN, receivers (Rrs) are identified by using the address of the output links to which they are connected, whereas transmitters (Trs) need not be identified. To dispatch a message, a Tr will generate a packet which has the following format:

<f'f" ; destination address ; message >

where f'f" is a 2-bit field which will be referred to as the feedback count, and is initialized to zero when a packet is newly formed, and incremented whenever the packet goes through the feedback paths. Later on, it will be shown that this field would never require more than two bits regardless of the

network size. The address of the Rr and the actual message to be transmitted are also contained in the packet.

Two types of switches, namely Type-A and Type-B, will be considered in our studies, and their schematic diagrams are shown in Fig.III.3.

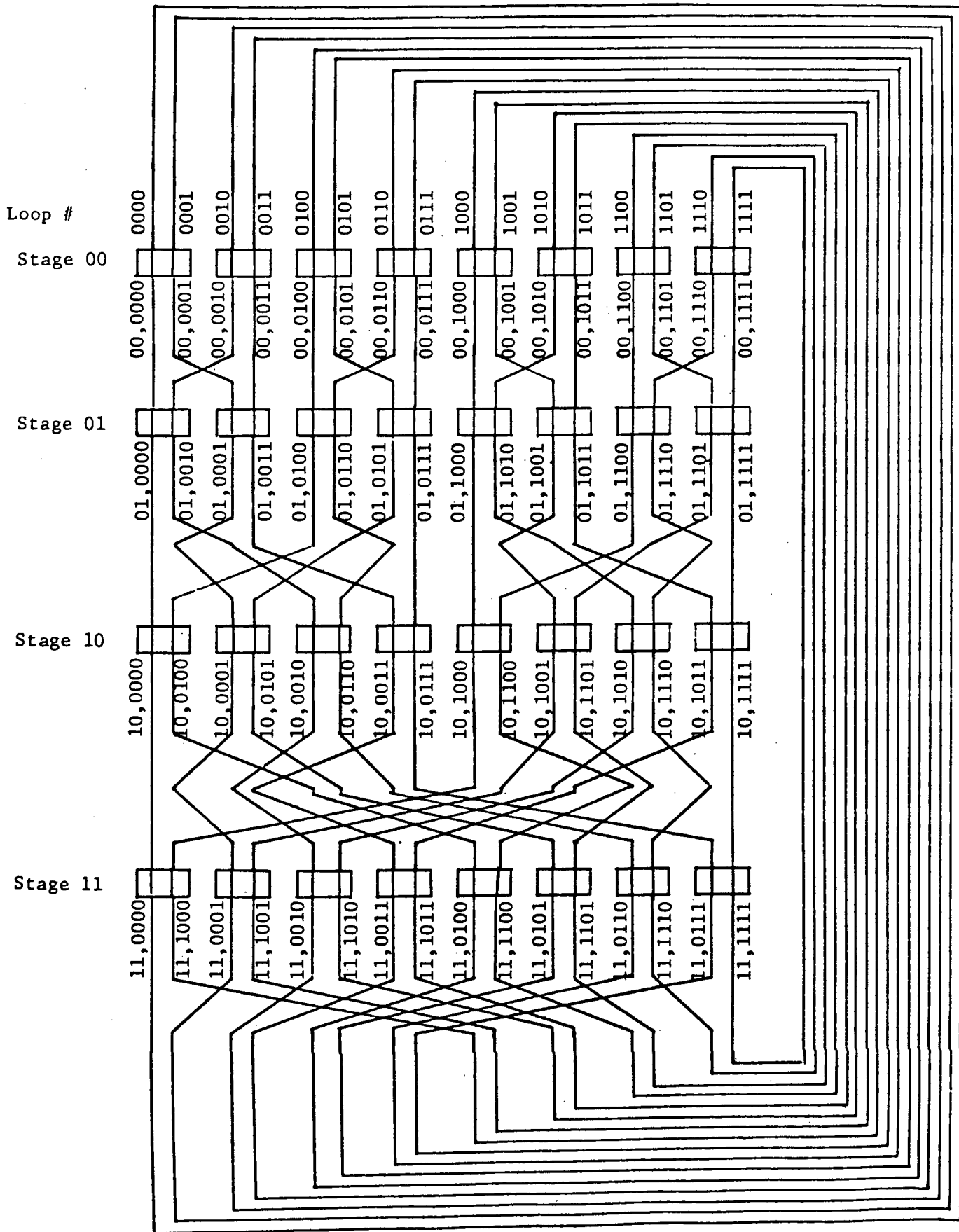


Fig.III.1. Assignment of loop and link labels on a LSSN which has 16 loops and 32 switches.

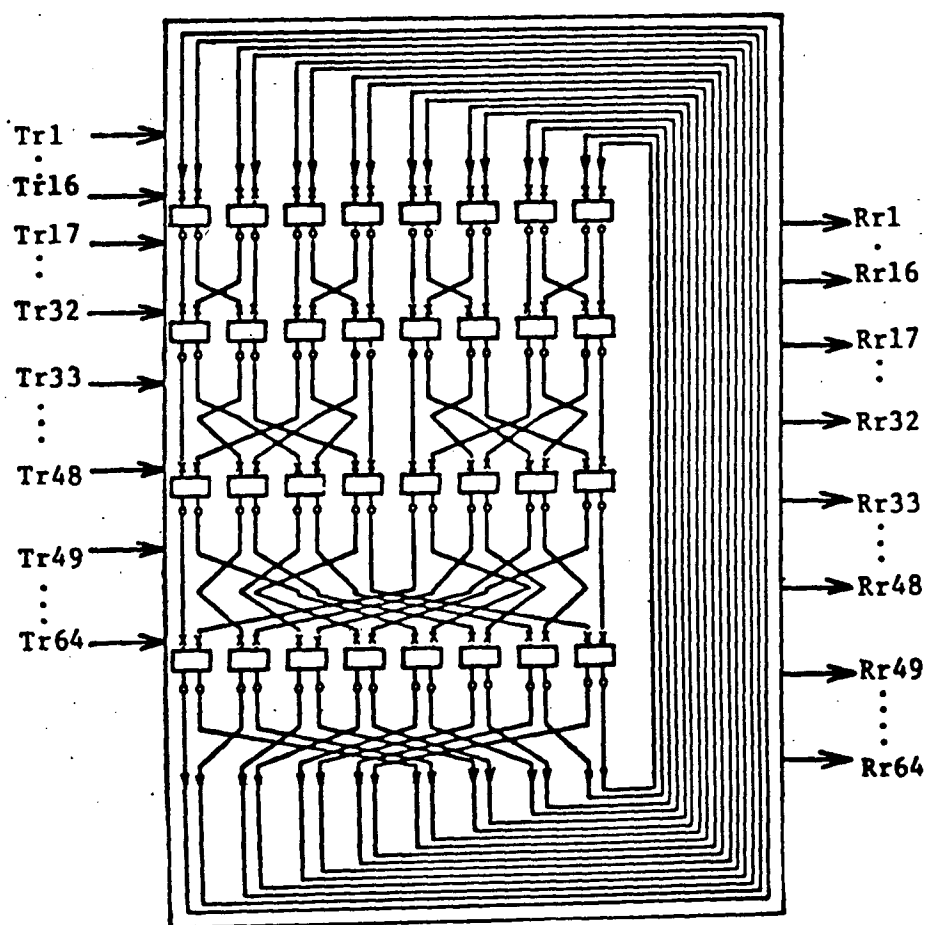


Fig.III.2. Connection of transmitting and receiving devices on a LSSN with 16 loops. (x= a hardwired connection for a transmitter, Tr; 0= a hardwired connection for a receiver, Rr.)

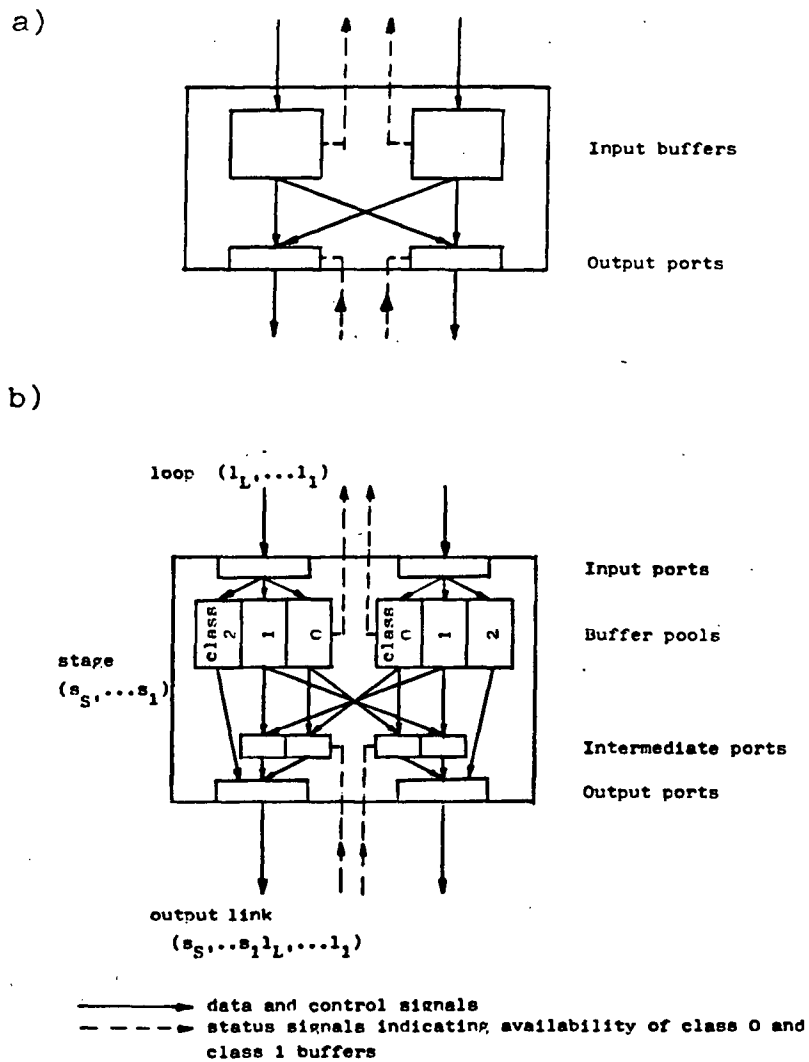


Fig.III.3. The schematic diagrams of a Type-A switch (a) and, a Type-B switch (b).

A Type-A switch is similar to those used in the conventional packet-switched networks, except that it has two built-in first-in-first-out buffers. When a packet enters a Type-A switch located in the s -th stage, it is first placed into one of its input buffers, and then switched to the left output port if the s -th bit of its destination address is a "0", or to the right output port if that bit is a "1".

As shown in Fig.III.3, a Type-B switch has a slightly more complicated internal structure than a Type-A switch. Its main features are the "structured buffer pools" which are made up of three classes of first-in-first-out (FIFO) buffers: Class-0, Class-1 and Class-2. It also contains four intermediate ports which are connected to the Class-0 and Class-1 buffers as shown. It has two sets of outgoing status lines indicating the availabilities of its Class-0 and Class-1 buffers to its preceding switches (which are connected to its left and right input links), and two sets of incoming status lines giving the same information from its succeeding switches (which are connected to its left and right output links). The Class- k buffer -- where k is in $\{0,1,2\}$ -- is used to accomodate packets with a feedback count of k ; the Class-2 buffer is connected to the output port directly while the Class-0 and Class-1 buffers are connected to the output port through the intermediate ports. The functions of these various mechanisms will be explained later on.

When a packet enters a Type-B switch located in the s -

th stage, it will undergo the following operations:

- (a) From an input port to the buffer pool: The packet will be placed into one of the buffers according to its feedback count;
- (b) From the buffer pool to the output port: Case(i) From a Class-0 and Class-1 buffers: If the s-th bit of the destination address of the packet is a "0", then the packet will be switched to the left intermediate port then to the left output port; else it will be switched to the right intermediate port then to the right output port. Case(ii) From a Class-2 buffers: The packet will be forwarded to the output port connected to the Class-2 buffer without switching and going through the intermediate ports.
- (c) From an output port to the exterior: At the output port, the destination address of the packet will be matched against that of the output link. If a match occurs, then a strobe signal will be sent to the receiver attached to that output link, and the packet will be removed from the output port by that receiver; else the next switch at the other end of the output link will be strobed and the packet will be forwarded to its input port. For the transmission between the last and the first stages via the feedback loops, the same operation will take

place, but in addition, the feedback count of those packets emerging from the output port of the last stage will be incremented.

These three operations will be collectively referred to as a single routing step for the Type-B switch. According to the descriptions above, the Type-B switch must contain the following features in addition to those depicted in Fig.III.3. First of all, the addresses of its output links must be made available to the matching operations (e.g., by storing the addresses inside the switch), and there must be some logic gates to perform the matching; the switch must be able to determine whether or not it is located in the last stage of the network by examining the labels assigned to its output links, because the feedback counts of those packets passing through it have to be incremented. Similar features must also be present in a Type-A switch, but those hardware involving the feedback counts may not be included.

Since the routing of packets is performed locally by each of the switches, LSSN has the advantage of not requiring a central controller. On the other hand, the lack of central control will give rise to conflicts among the packets for the shared network resources such as ports and links; the effects of such conflicts on LSSN when Type-A and Type-B switches are used will be detailed in the next section.

3. Network Properties

First, some useful theorems concerning the behaviors of LSSN with the presence of a single packet will be stated, then the various properties of LSSN with the presence of more than one packet will be examined. The proofs of all these theorems are given in Appendix A, and all the algorithms used are base-two.

Notice that even though the destination addresses $s_S, \dots, s_1 \ell_L, \dots, \ell_1$ carried by a packet consists of $(S'+L')$ bits of information, only the L' least significant bits are involved in the switching operation (i.e., Operation (b) of Section 2.B); and the first S' most significant bits, together with the L' least significant bits, are involved in the matching operation (i.e., Operation (c) of Section 2.B) only. This observation leads to the following lemma:

Lemma III.1: Consider a LSSN which has L loops and a packet which is destined for the address $s_S, \dots, s_1 \ell_L, \dots, \ell_1$, where $L'=\log L$ and $S'=\lceil \log L' \rceil$. The packet will be routed to the loop ℓ_L, \dots, ℓ_1 within L' steps of routing after its admission into the LSSN.

Example: Consider a LSSN with $L=16$, then $L'=4$ and $S'=2$. A packet destined for the address (101111) will be routed to the loop (1111) within 4 steps of routing regardless of where it is generated.

Lemma III.2: Consider a LSSN with L loops and a packet which is destined for the address $a_S \dots a_1 \ell_L \dots \ell_1$, where $L' = \log L$ and $S' = \lceil \log L' \rceil$. After the packet has been routed to the loop $\ell_L \dots \ell_1$, it needs at most another $(L' - 1)$ steps of matching along that loop to reach its destination.

Theorem III.1: In a LSSN with L loops, a packet will be delivered to its destination within $(2 \log L - 1)$ steps of routing regardless of where it is generated.

Example: In the example of Fig.III.1, $L=16$; therefore the maximum number of routing steps is $(2*4 - 1)=7$.

Theorem III.2: The average number of routing steps (ARS) needed to deliver a result packet in a LSSN with L loops is,

$$ARS(L) = (3 \log L - 1) / 2 + 2 / L - 1$$

Example: In the example of Fig.III.2, since $L=16$, therefore,

$$ARS(L=16) = (3 \log 16 - 1) / 2 + 2 / 16 - 1 = 4.625.$$

Corollary III.1: Any packet admitted into LSSN will go through the feedback path at most twice.

Example: In the example of Fig.III.1 and 2, if Tr49 -- which is attached to link 10 0000 -- sends a packet to Rr32 -- which is attached to link 01 1111, then this packet will go through the feedback paths twice: The first time through the loop (1000), and the second time through the loop (1111).

Corollary III.1 explains why the packets only have to carry two bits to indicate its feedback count $f'f''$, and also why each buffer pool of the Type-B switch is made up of three classes of buffers regardless of the network size L .

Theorem III.3: In a Type-B switch of a LSSN which has L loops, the probability that the destination address carried by a result packet will match the label of an output link of the switch, and hence the packet will be removed from the network is:

$$P_{\text{removed}} = 2L / \{3L \log L - L + 4\}$$

where the transmission pattern is such that each and every receiving port of the network is equally likely to receive that packet.

Theorem III.4: The maximum average throughput rate (MATR) of a LSSN with L loops is:

$$\text{MATR}(L) = 3/2 \times S_{R,SW} \times \log L \times L^{**2} / \{3L \log L - L + 4\}$$

where $S_{R,SW}$ is the maximum rate of transmitting Result packet between two switches via an output link.

3.A. Network Conflicts

When there are two or more packets in LSSN, they may contend for the same network resources such as input buffers, ports and data links, thus giving rise to conflicts.

If Type-A switches are used in LSSN, then there would be two types of conflicts:

- (a) A1 conflicts - which are the contentions due to the simultaneous requests made by packets in the two input buffers, for the same output port;
- (b) A2 conflicts - which are the contentions between an output port and the Tr sharing the same link, for the same input port of the switch at the end of the link.

A simple round-robin discipline can resolve both types of conflicts and will ensure fairness. A better alternative for A1 conflicts is to honor the input buffer which has more

waiting packets in it; and if both input buffers are equally occupied, then an arbitrary buffer will be chosen. As for A2 conflicts, the output ports perhaps should be given a higher priority over the Tr's so that those packets which are already admitted into the network could reach their destination faster (i.e., a faster response time could be obtained). These observations were obtained from the simulator listed in Appendix C.

As for Type-B switches, there are three possible types of conflicts (the reader may refer to Fig.III.3 for the following descriptions):

- (a) B1 conflicts - which are due to the simultaneous requests made by packets from the left and right buffer pools, for the same intermediate port;
- (b) B2 conflicts - which are the contentions among the the intermediate ports and Class-2 buffer for the same output port;
- (c) B3 conflicts - which are the contentions between an output port and the Tr sharing the same output link, for the input port at the end of the link. B1 conflicts could be resolved by a simple round-robin discipline: the conflicting packets are switched to the intermediate port alternately.

The resolution of B2 conflicts is more intricate. Our simulation studies showed that the round-robin discipline would give rise to unbearable propagation delay to certain packets, but much better performance, in terms of average throughput rate and delay, could be obtained with a priority-based policy (an explanation will be given in Section 3.B) which assigned the highest priority to the Class-2 buffer, and then the intermediate port connected to the Class-1 buffers, and finally the intermediate port connected to the Class-0 buffers. With this policy, packets in the Class-2 buffers are switched to the output port immediately when the output port becomes empty; as could be explained by Lemma 1 and 2, these packets will always remain in the same loops, therefore they need not go through any intermediate port. Furthermore, since they are assigned the highest priority in the use of the output port, they will not accumulate and hence the size of the Class-2 buffers is always bounded. When the Class-2 is empty, the "eligible" intermediate port with the next highest priority will be granted access to the output port. An intermediate port connected to the Class-k buffers is said to be "eligible" if it is non-empty, and if the incoming status lines indicate that the Class-k buffer of the next switch is not full. As for the connections between the last and first stage, an intermediate port connected to the Class-k buffer is "eligible" if it is non-empty and if the Class-(k+1) buffer of the next switch in the first stage is not full. The difference in the above definitions of

"eligibility" is discernible if one realizes that the feedback count of a packet is incremented whenever it goes through the feedback paths back to the first stage. The purpose of the status lines is therefore to help prevent the output ports and input ports from being clogged with packets which cannot be switched away immediately.

The priority-based policy would favor those packets originated at the lower stages, because the feedback counts of these packets are incremented sooner than those originated at the upper stages, and hence will be assigned higher priorities sooner. However, our simulation studies shows that this policy is superior than the round-robin discipline as far as the overall performance is concerned (an explanation will be offered at the end of next section).

The resolution of B3 conflicts is rather straightforward: the conflicting Tr and the output port are granted access alternately. But in addition, it is necessary for the Tr to check that the Class-0 buffer (not just the input port) at the entry point is not full before it can transmit. The availability of the Class-0 buffer has to be checked because newly admitted packets carry feedback counts of zero.

3.B. Deadlock and Avoidance Method

In conventional non-recirculating, packet-switched

networks, the blockage due to data path conflicts is temporary as long as there is a fair scheduling policy; whereas in a LSSN which uses Type-A switches, blockage might lead to deadlocks -- the situations in which certain loops are clogged with packets and no further switching can take place along these loops, and very soon the whole network will become impassable.

The deadlock problem in LSSN is attributable to the store-and-forward type of data movements and the cyclical requests of network resources. In a Type-A switch, if the packets coming out of its two input buffers always contend for the output ports, then the input buffers will be filled up rapidly; and if all the input buffers and output ports along a particular loop are filled with packets in transit, and if the first packets of all these input buffers are waiting for these occupied output ports to be freed, then this loop will enter a "single-loop" deadlock. A "multiple-loop" deadlock is produced in a similar manner but it involves more than one loop.

According to our simulation studies, the probability of deadlock could be reduced significantly by increasing the size of buffers and restricting the input load down to a certain level; but this approach does not eliminate deadlocks entirely, and moreover, it requires a deadlock detection scheme and a recovery procedure. Perhaps it is more

efficient to get around the deadlock problem by avoiding cyclical requests of the network resources; and Type-B switches are meant for such a purpose.

Our idea of using Type-B switches to prevent deadlocks is based on the concept of "structured buffer pools" put forward by Raubold and Haenle [29]. According to their method, buffer pools are divided into K classes, where K is the length of the longest path in the network concerned, and if a packet is of r routing steps away from its transmitter, then it may be placed into any Class- k buffer such that $k \leq r \leq K$. Clearly, their method has the drawback that K must be a function of the network size. We eliminate this drawback by classifying packets according to their feedback counts which has been proved to be bounded.

With the use of Type-B switches, the LSSN will be free of the store-and-forward type of deadlocks. A simple explanation is as follows: for packets entering the buffer pools, they will request buffers according to their feedback counts, therefore there is no circular request on the buffers; as for the shared links and input/output ports, these network resources are granted to the requesting packets on the condition that their occupations by the packets will always be temporary. With this idea in mind, now we will state the following theorem:

Theorem III.5: The LSSN which uses Type-B switches is deadlock

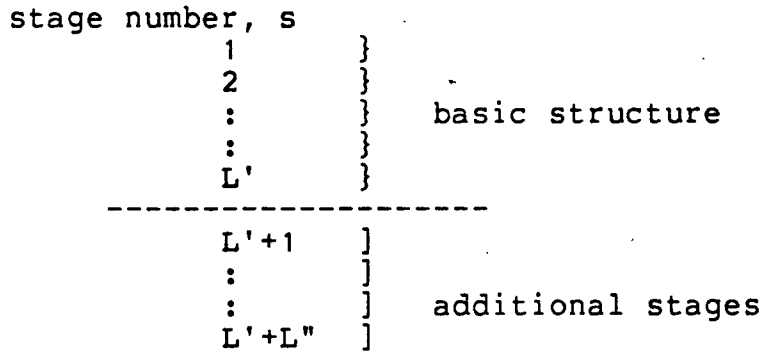
free.

An explanatory proof of Theorem III.5 is given in Appendix A.

3.C. Network Extensibility

Very often it is desirable to expand a network after it has been built; but usually such an expansion is difficult with most, if not all, existing designs. LSSN has the very useful property that it can be expanded incrementally by adding more stages to the basic structure without complicating the addressing and routing algorithms. Of course, to facilitate the expansion, there must be sufficient address lines to account for the added stages and devices.

One way to expand the basic structure while keeping it intact is to add the new stages to the bottom of it -- immediately after the last stage of switches and before the perfect shuffle takes place. Suppose there are L loops and L' stages originally, and we want to add L'' more stages, then the expanded LSSN would have a total of $(L'+L'')$ stages as shown:



Now the stages and R_r 's will be addressed using $S'' = \lceil \log(L' + L'') \rceil$ and $\{\lceil \log(L' + L'') \rceil + L'\} = \{S'' + L'\}$ bits of binary digits respectively. The addressing scheme and connection function for the newly added stages are:

$$\text{Left output link} = (\Delta_{S''} \dots \Delta_1 0 \ell_{L'-1} \dots \ell_1)$$

$$\text{Right output link} = (\Delta_{S''} \dots \Delta_1 1 \ell_{L'-1} \dots \ell_1)$$

$$\text{LSSN}_S(\ell_{L'}, \dots, \ell_1) = (\bar{\ell}_{L'}, \dots, \ell_1)$$

In words, all the new stages would be treated much the same as the last stage of the basic structure, i.e., the L' -th stage; and there is no shuffling among the output links of these new stages; and packets which are sent to them are routed according to the L' -th bits of the destination addresses of the packets.

In the expanded LSSN, the duty of incrementing the feedback counts is performed by the $(L' + L'')$ th stage rather than the L' -th; such a minor change has to be taken care of during the expansion. We do not intend to derive theorems from scratch for the expanded network because the validity of

Corollary III.1 and Theorem III.2 are discernible if the new stages are regarded as the subsidiaries of the L' -th stage, i.e., if stage L' through stage $(L'+L'')$ are considered as a single, compound stage. Lastly, we would like to point out that LSSN could also be expanded in the more expensive way by doubling its loop count.

4. Simulations and Performance Analysis

In our simulation studies, the throughput rate and delay are the two measures used for evaluations and comparisons. Throughput rate is defined as the average number of packets flowing through the network per unit time, and the delay of packets is defined as the average interval between their generations and receptions. The delay is made up of "entrance delay" and "propagation delay", where entrance delay is the average duration that a packet has to wait at the entry point, and propagation delay includes the time spent in queueing and switching within the network. Request interval is the varying parameter and is defined as the average time between the last successful transmission and the generation of the next packet.

In order to obtain some meaningful results and to facilitate the analysis later on, we have made the following assumptions:

- (a) The transmitting and receiving pairs are randomly selected out of the entire address space;
- (b) The transmission pattern is such that if the current request to transmit is in process or blocked, then the transmitter affected will not generate the next request;
- (c) Packets are removed immediately from the network when they arrive at their destination;
- (d) As for timing considerations, the amount of switching delay in going through a conventional binary switch was estimated to be five gate delays [30]: three for path selection and two for data transfer. Since Type-A switches would lead to deadlocks on LSSN, their analysis will not be included in our studies. A Type-B switch would need more delay than the conventional ones: three gate delays for path selection, two for data transfer from the input ports to the buffer pools, two from the buffer pools to the intermediate ports and another five for path selection and data transfer from the intermediate ports to the output ports -- a total of twelve gate delays. In the case of those packets which are inside Class-2 buffers, their switching delay are shorter because they do not have to go through the

intermediate ports. The main duty of the LSSN simulator is to compute the total delays of each individual packet by summing up its entrance, switching as well as queueing delays.

These assumptions are considered justifiable, and they have also appeared in the simulation studies of other packet switching networks (e.g. references [25,30]). In the LSSN simulator, a timer was associated with each packet entering the network, so as to record each type of delays that it will encounter. From time to time, the whole switching array was inspected to make sure that no packet would be subjected to substantial delay -- which is an indication of potential deadlocks. The LSSN under study had 16 loops and was fully connected with 64 pairs of transmitters and receivers. The effects of the buffer size on the network performance were first investigated, and it was confirmed that because the Class-2 buffers were given the highest priority in the B2 type of conflicts, the maximum requested size of the Class-2 buffers was bounded to two. For this reason, the size of the Class-2 buffers was fixed at two, and the sizes of the Class-0 and Class-1 buffers were varied from 4 to 14 (an arbitrary range). Our results (please see Fig.III.5) show that the variation of the sizes do not have a significant effect; the reason is that when more buffers were used, there would be more traffic introduced into the network, although the entrance delay of a packet is reduced, its propagation delay

would be increased; as a result, the total delay is not much affected.

In the second part of our study, we compared the performance of a 64x64 LSSN to that of a 64x64 baseline and then a 16x16 baseline. The baseline networks were considered because they are topologically equivalent to many existing networks [20]. We must emphasize that our comparison studies are not entirely fair because baseline-like networks could be used as either circuit-switched networks (e.g., The Star network [24]), or packet-switched networks, whereas LSSN is intended to be used as packet-switched networks only; furthermore, the LSSN switches have a much higher logic gate density than the conventional ones. For comparisons, we assumed that the switches used in the baselines had a buffer size of 16 (an arbitrary number); as for LSSN switches, the sizes of its Class-0, Class-1 and Class-2 buffers were fixed at seven, seven and two, respectively -- a total of 16 as well. The results obtained for this buffer size are presented in Fig.III.5; other buffer sizes would produce results very similar to these.

In Fig.III.5, all the measurements were scaled by the factor f which is the operating frequency of the networks. The value of f could be as high as 60 MHz if the switches are fabricated with TTL gates, or 400 MHz with ECL gates. A 64x64 baseline would need a total of 192 switches whereas a

64x64 LSSN and a 16x16 baseline would require 32 switches each. Not only the numbers of switches would contribute to the complexities of the networks, but the amounts of wiring have to be considered as well.

Our results show that if the request interval is very short, the throughput of the LSSN will be close to that of the 16x16 baseline, and its delay will be about three times higher; but when the request interval is longer than $40/f$, then both the throughput and delay of the LSSN will approach that of the 64x64 baseline. If the request interval is further increased, the delay of the LSSN will be reduced to that of the 16x16 baseline.

In summary, our results indicate that the performance of a 64x64 LSSN can match that of a 64x64 baseline with a significantly lower switch count and hence fewer wiring. This savings is even more substantial when the sizes of the networks considered are very large.

5. Discussions and Outlook

We have described a novel method to set up a communication network based on the concept of cyclical architectures; we have also presented the addressing and routing schemes, and several properties of the network. Although packet-switched, cyclically connected structures are

susceptible to the store-and-forward type of deadlocks when used asynchronously, we have suggested a deadlock avoidance scheme based on some unique features of our design.

The topology of our proposed network LSSN resembles that of some existing ones. If only L processors are attached to LSSN such that they all transmit packets through the first stage and receive packets from the last stage, then LSSN would be reduced to an indirect binary n -cube [21]. This similarity implies that those useful algorithms developed for the indirect binary n -cube could be adapted for LSSN easily. LSSN also resembles the last stage of the Batcher's bitonic sorter [13]; therefore, it is possible to perform Batcher Sort on LSSN provided there exists a masking scheme to disable some of the attached processors as data items are circulated around the network. LSSN could also be partially connected and used as an arbitrator or a distributor -- both of which are essential in the designs of data-driven computers [4,5,8,52,53].

LSSN can also be used to perform arbitrary permutations -- i.e., one-to-one mappings -- from the input side to the output side. Such permutations would require the presence of a central controller to compute the routing information; alternatively, those frequently used control patterns could be pre-computed and retrieved when needed. The simpler, Type-A switches could be used for such an application and they will not cause the network to deadlock as

long as only one permutation is performed at a time, and provided there are sufficient buffers inside the switches:

$$B(L) \geq \text{MAX}\{\text{MIN}(2^{\lceil 0.5 \log N \rceil}, N/2^{\lceil 0.5 \log N \rceil}), \\ \text{MIN}(2^{\lceil 0.5 \log N \rceil}, N/2^{\lceil 0.5 \log N \rceil})\}$$

$$= O(N^{0.5})$$

where $B(L)$ is the number of buffers of the Type-A switches needed to avoid deadlocks, and $N = L \log L$, where L is the number of loops; the worst-case delay to perform a one-to-one mapping is:

$$T_{\text{max}}(L) = 2^{\lceil 0.5 \log N \rceil} + N/(2^{\lceil 0.5 \log N \rceil}) - \log L - 1 \\ = O(N^{0.5})$$

and the average delay is:

$$T_{\text{avg}}(L) = O(\log N)$$

These results could be found in reference [23]; because both $B(L)$ and $T_{\text{max}}(L)$ are intolerably large for large values of N , we do not intend to include the analysis in this dissertation.

As is shown by our simulation results (Fig.III.6), the performance of the LSSN is not as good as that of the baseline

network for applications which have very short inter-transmission times; but if the transmitters are processors which send out data packets as the results of instruction executions -- i.e., if the processors compute and dispatch alternately -- then the LSSN will be an attractive design.

Our proposed system trades off external hardware complexities (e.g., component counts, wiring, etc.) with internal hardware complexities (i.e., logic gate per switch). High internal complexities can be easily achieved with today's technologies, but too many external components and wiring often renders the system difficult to manage -- this is the main motive behind our design.

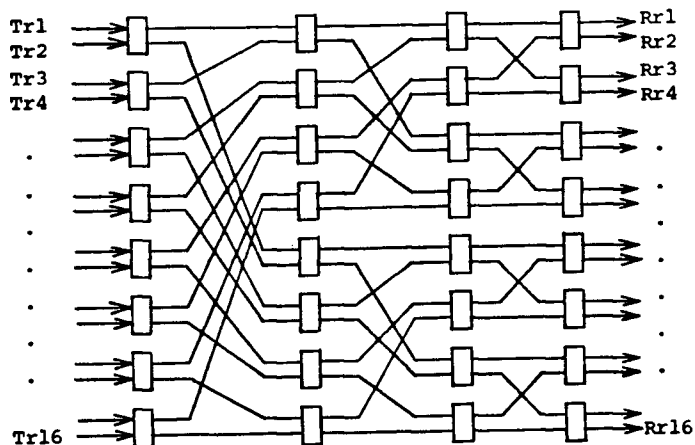


Fig.III.4. A 16x16 baseline network [20].

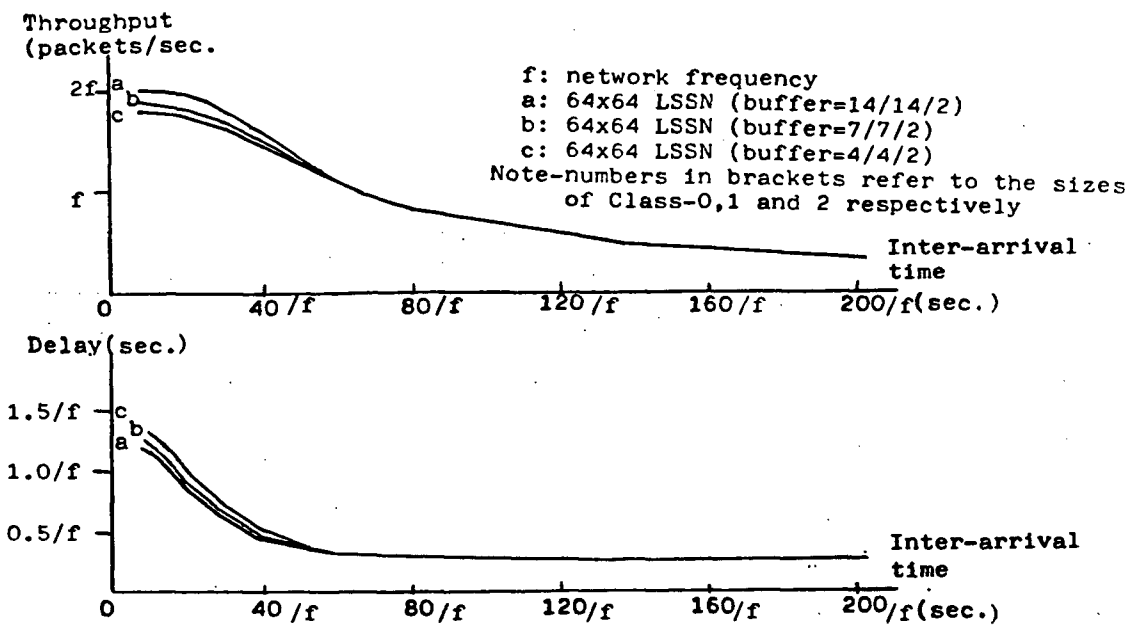
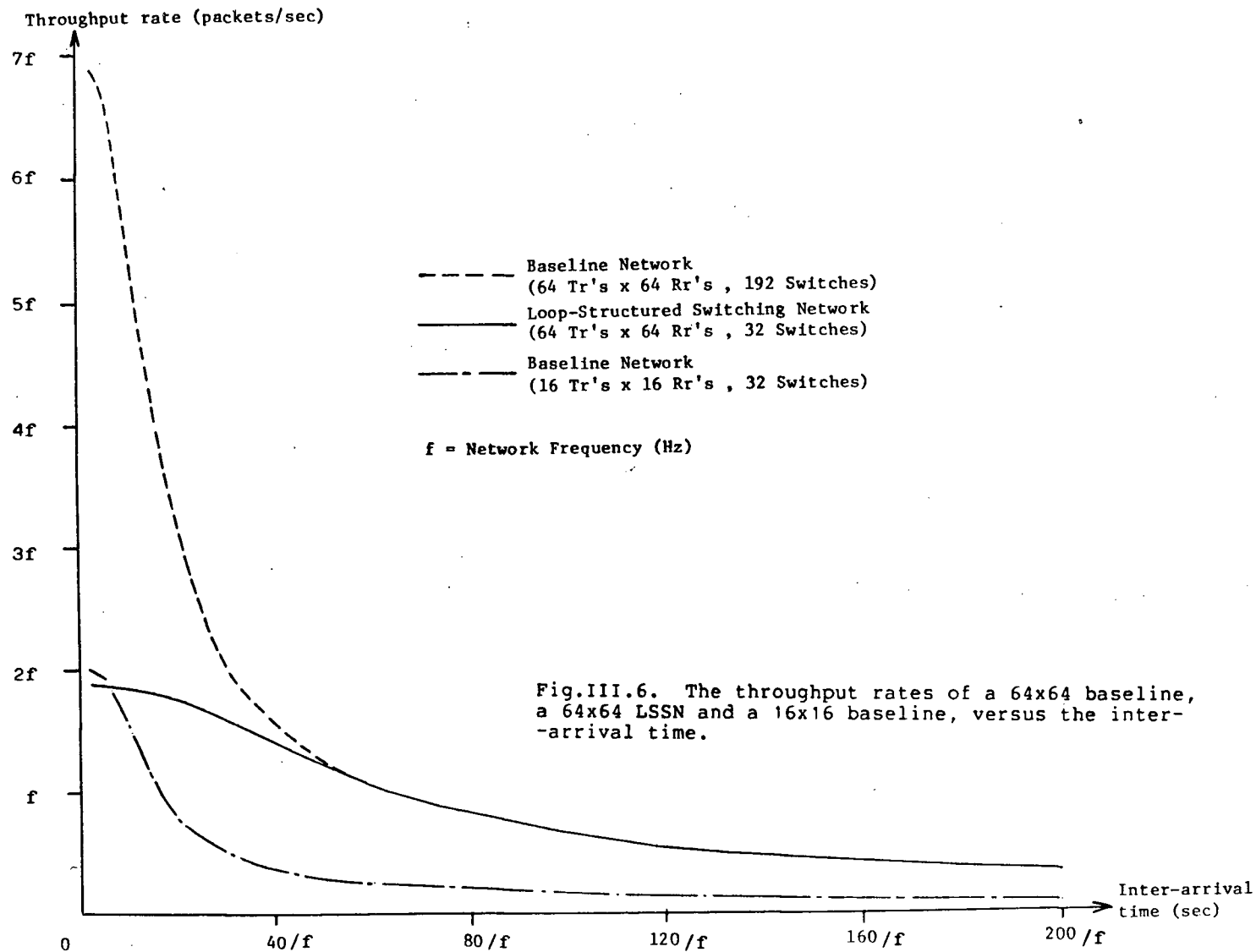


Fig.III.5. Effects of buffer size on the throughput and delay of a 64x64 LSSN.



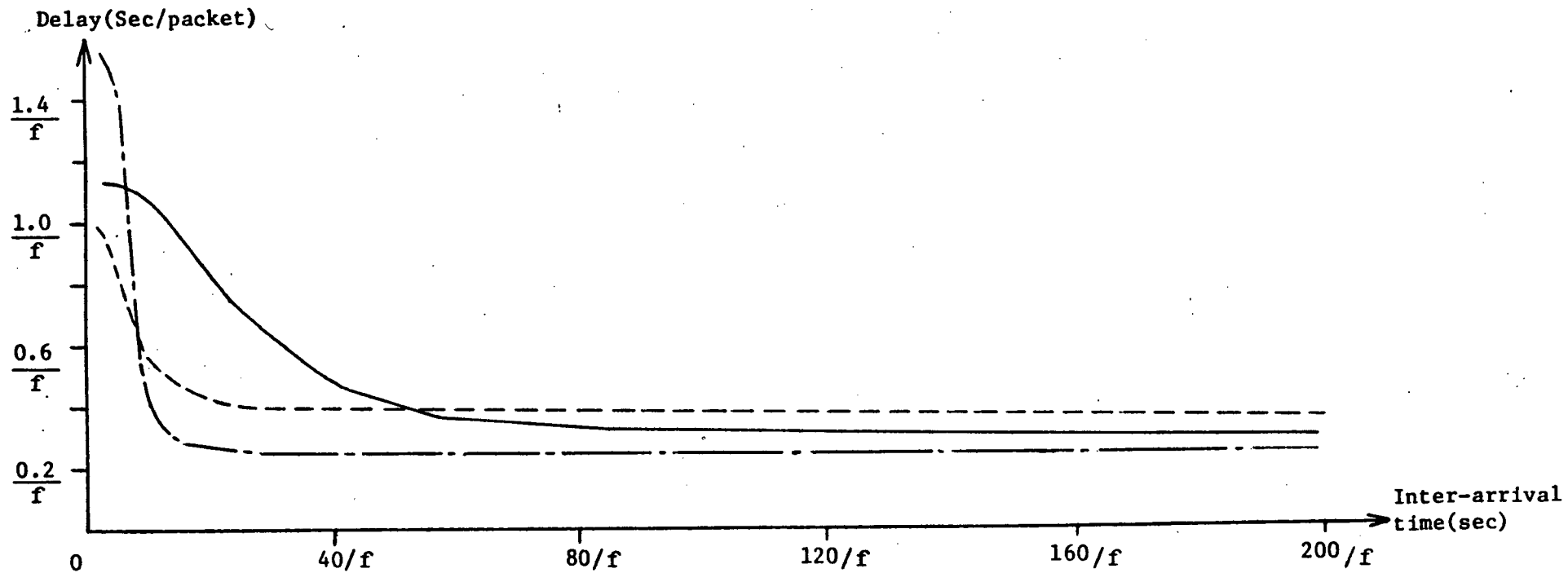


Fig.III.7. The delay curves of a 64x64 baseline, a 64x64 LSSN and a 16x16 baseline, versus the inter-arrival time.

Chapter IV. Design and Evaluation of The Event-Driven Computer (EDC)

1. Introduction

1.A. Background Information

In this chapter, we will examine how the concept of cyclical architectures could be applied to the design of a high-performance supercomputer; to start with, we will discuss the shortcomings of the conventional computer systems in this respect, and then the approach of our design will be identified.

Conventional computer systems are often referred to as Von Neumann, or sometimes as Harvard, machines, and they all have very similar "control" and "data" mechanisms. "Control" mechanisms refer to the methods for scheduling instructions for execution, and "data" mechanisms refer to the methods for passing data among instructions. Von Neumann computers are termed "control-driven" because their instruction executions are sequenced by control signals generated by the CPUs (Central Processing Units). In these computers, data are passed among instructions by writing and reading memory locations which are specifically assigned to these data.

A major drawback of using Von Neumann computers in a highly parallel environment is attributable to the need of, and difficulties in, specifying concurrency -- either the programmer or the compiler has to be careful with the

generation of control signals, to ensure that memory locations are not corrupted by wrongful information during read and write operations. Such a drawback is easy to overcome in a SIMD (Single Instruction stream Multiple Data streams [6]) system because there is only a single stream of executable instructions; whereas in a MIMD (Multiple Instruction streams Multiple Data streams) system, the asynchronous behavior of memory accesses among the various instruction streams often complicates the implementation of the control mechanisms. Another major drawback is attributable to the difficulty in "load partitioning" which often gives rise to uneven work load distributions among the processors and access bottlenecks in the memory modules. Moreover, system extensibility is always difficult to achieve, and increments in the number of processors are often not accompanied by proportionate improvement in the system performance.

The goal of research in "data-driven" computers [4] is aimed at alleviating the above shortcomings, and both their data and control mechanisms are implemented very much differently from those of Von Neumann systems: the data mechanism is such that data are passed from the producing instructions to the consuming ones directly without going through any intermediate storage, and the control mechanism is such that the consuming instructions would be readied for execution if, and only if, they have received all the required data and information. For a data driven computer, its control mechanism could therefore be implemented as

suboperations incorporated into its data mechanism, which could be easily implemented using well-known compilation techniques (e.g., data-flow analysis). The absence of an explicit control mechanism would ease the task of the programmer in specifying parallelism to a great extent. As a result, the data-driven approach is very appealing to multiprocessing and multiprogramming environments which contain large amounts of unstructured, asynchronous concurrency. However, the implicit control mechanism of data-driven systems does not conform to the notion of certain activities such as input and output operations, which are not necessarily ready for execution when their data have arrived. Further, the data mechanism of data-driven systems is very inefficient in handling large arrays because sending arrays among instructions for computation is both time and space consuming.

From the above discussions, it is clear that the data-driven and control-driven approaches are complements of each other; therefore, it is very natural to envision a class of computers which combine their control and data mechanisms for the purpose of better performance.

1.B. Recent Developments

This section will examine three existing proposals which adopt the combined approach:

- (1) Dependence-Driven system (1981) [32];
- (2) Combined system (1982) [33];
- (3) Piece-wise Data-Flow system (1983) [34].

The Dependence-Driven system is made up of a GCU (Global Control Unit) and several processor clusters, each capable of executing a high-level function. The compiler is expected to produce all the static information about the computation and the GCU will perform run-time scheduling. This system is best-suited for computation which could be heavily vectorized; however, the presence of scalar computation would cause some processing resources to stand idle during their executions, thus giving rise to under-utilization of these resources.

The Combined system integrates the concepts of the "pure" data-driven computation and those of the "multi-thread" control-driven computation. Trealeven et al [33] have shown how iterations, procedure calls and resource management are carried out on this system; not mentioned is how array operations are performed. If array operations are decomposed into individual packets each containing a participating array element, then there will be enormous amounts of overhead associated with the setting up and transmission of the packets, and also to synchronize the completions of the array operations.

Requa et al [34] have provided a rather detailed

description of the PDF system which possesses both SIMD and MIMD characteristics -- these two classes of computation are performed on different types of hardware modules which are not interchangeable. We believe that if both scalar and array operations could be carried out on the same type of hardware modules, then there will be fewer module types, and hence the system would be less expensive to design and easier to control. The PDF system avoids using any interconnection network by limiting the number of scalar processors to about eight; therefore, the speed of the PDF system is expected to have limited improvement over existing ones (please refer to the Introduction Section of [34]).

1.C. Overview of Our Approach

Our objective is to design a heterogeneous multiprocessor system which,

- (1) is capable of using hundreds to thousands of processors;
- (2) has a projected speed range of 100 to 1,000 MOPS (million operations per second);
- (3) possesses both SIMD and MIMD characteristics -- this is to be achieved by combining the principles of data-driven and control-driven computation;
- (4) is intended for next-generation applications, and is expected to depart from the prevalent, Von Neumann architectures.

In order to connect a large number of processors together and yet maintaining a high degree of flexibility, a large switching network would be included in our design. To achieve the desired speed range, the intended applications must possess a large amount of concurrency to keep the processors busy most of the time. Since the ratio of SIMD and MIMD instruction mix differs from application to application, in order to fully utilize its resources, the system must be able to maintain roughly the same level of performance regardless of the ratio of mix. We also believe that in order to attain a significant achievement toward ultra-fast computation, the new design may have to depart from the prevalent Von Neumann systems in both hardware and software; therefore, we only emphasize the architectural aspects of our design rather than any immediate implementation.

In our proposed system, there are two basic types of operations -- scalar and compound operations, both of which are scheduled for execution using data-driven principles; but suboperations within a compound operation are sequenced for execution in a control-driven manner. A compound operation is either a computational array operation, an array alignment operation or a block of sequential program. A sequential program is one which either requires a fast computation time and could run faster when executed solely by a single processor in a SISD (Single Instruction stream Single Data stream) mode than by many of them in a MIMD mode (due to

communications overhead), or is used to control an inherently sequential process such as printing on the line printer.

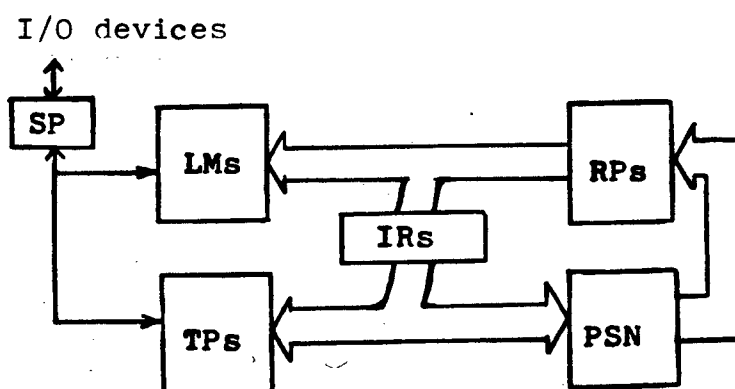


Fig.IV.1. The EDC system block diagram.

As shown in Fig.IV.1, an EDC consists of six basic parts: A Supervising Processor (SP); a bank of Local Memories (LMs); several Transmitting Processors (TPs) and Receiving Processors (RPs); a number of Instruction Registers (IRs) and a Packet Switching Network (PSN). The main duty of SP is to load and spread instructions and data into the bank of LMs, and to initiate the appropriate TPs to start execution -- both using the read/write links provided on the left of LMs and TPs. Compound operations (i.e., array computation, array alignments and executions of sequential programs) will involve the use of these read/write links as well: when a TP receives such a compound operation, it will become a subcontroller and request SP either for the control of several TP-LM pairs for array computation in a SIMD manner, or to initiate the execution of a block of sequential program on another TP in a SISD fashion.

Information flowing on the right of LMs and TPs are encapsulated into the form of either result or instruction packets: result packets are generated by TPs and are switched through PSN to RPs which will place them into the proper LMs; RPs are also responsible for the formation of instruction packets: they retrieve the executable instructions and data from LMs and buffer them into IRs to wait for free TPs for execution. Because each of the TPs will receive instructions from different instruction streams from time to time, the interpretation of "MIMD" in this case is somewhat different from the traditional one.

Other salient features of the EDC system include:

- (a) Fewer module types: Only a few types of hardware modules are used, although each type is intended to be used in large amounts. This would reduce the design costs and give rise to a simpler architecture which is easier to control than one which uses a lot of module types.
- (b) Interleaved instructions and skewed arrays: A subset of the LMs are used to store sequential programs which will be executed by their associated TPs using the read/write links in a SISD mode; while the majority of the LMs are meant for non-sequential programs which will be executed in a MIMD mode by TPs using the packet-switched network. For the latter case, the instructions will be interleaved into the

LMs concerned, thus randomizing and equalizing the access pattern of TPs and RPs; array elements will be skewed into these LMs using known storage techniques [36,37] which allow different portions of of an array to be referenced concurrently. These features would reduce the problem of memory access bottlenecks.

- (c) Overlapped array operations: While an array is being operated on using the read/write links, several array alignment operations could be carried out using the packet-switched network which provides a novel way to synchronize the completions of the alignment operations, and to signal those instructions dependent on them.
- (d) Extensibility: The EDC architecture is highly extensible. After an EDC system has been built, the numbers of TPs, RPs, IRs and LMs could be increased incrementally. Such an advantage is attributable to the extensibility of the network.

A more detailed schematic diagram of an EDC is shown in Fig.IV.2. The functional descriptions of the system hardware will be given in Section 2, and Section 3 will explain how information is stored and processed in an EDC. Section 4 will describe the nature of the programming language to be used, and Section 5 will examine the performance of an EDC. A comparison of an EDC with the three afore-mentioned designs and some suggested work will be given in Section 6.

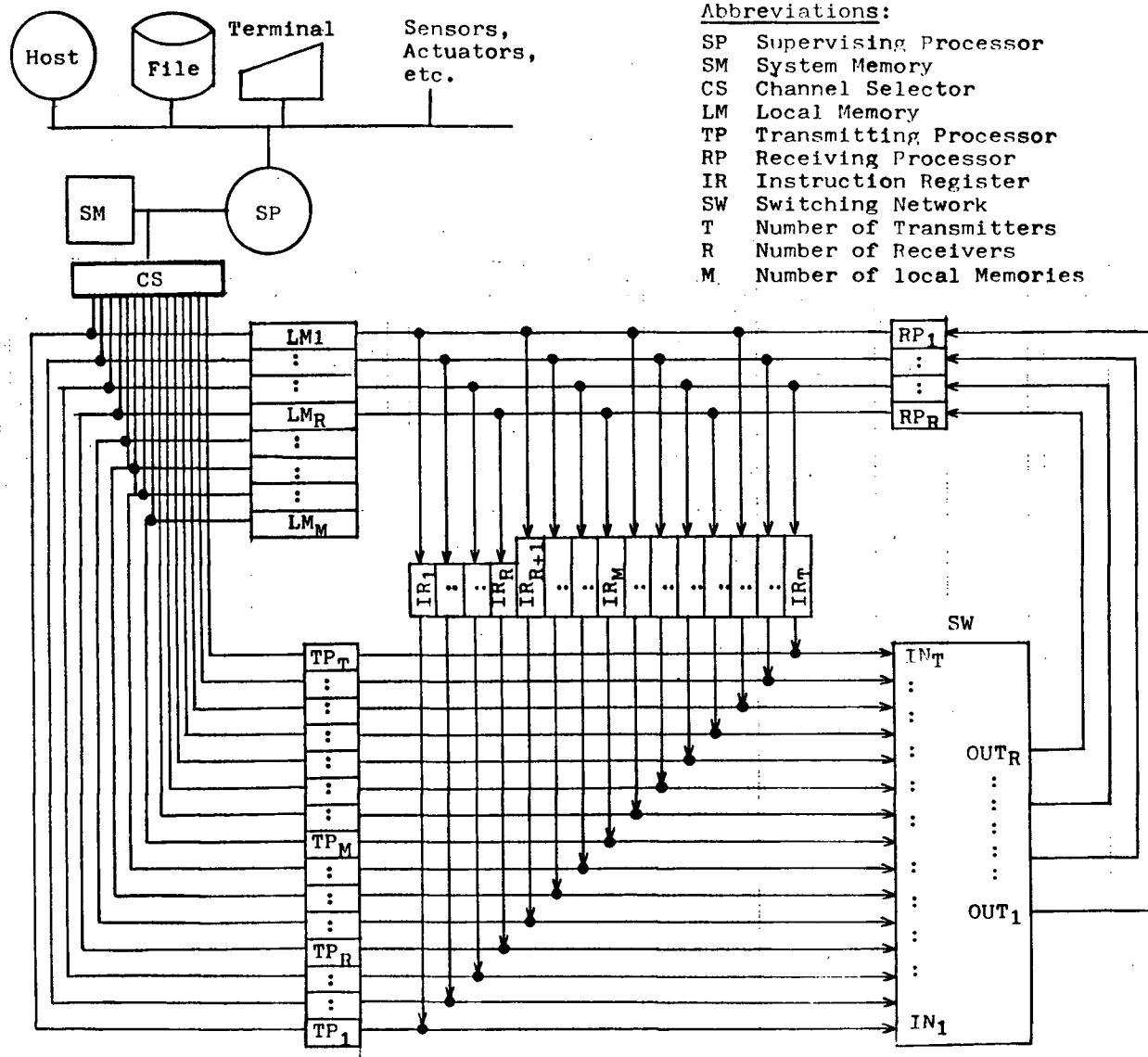


Fig.IV.2. The connection diagram of EDC hardware architecture.

2. EDC Hardware Architecture

2.A. Processing Modules

(1) Supervising Processor (SP)

SP is the master controller of the whole system and it oversees the executions of the following activities:

- (a) Program downloading and initialization: Programs are loaded from external sources such as the host computer or bulk memories, and stored in the System Memory (SM) initially. When a program is called for, SP will access a storage utilization table (SUT) which is located in SM, and allocate free memory pages to the called program which will then be fetched from SM and loaded into LMs. At the end of loading, SP will signal the TPs concerned to start execution.
- (b) Input and output operations: Input data will first go to the input buffer located in SM, and then proceed to the appropriate LMs. If the various parts of an array are to be referenced independently and concurrently, then it will be skewed into the first R LMs using techniques described by Budnik et al [36,37] for conflict-free accesses; the storage pattern will then be recorded in an array description table (ADT) located in SM for future references. Output data will be transferred from LMs to the output buffer which is also located in SM, and then

to the outside devices.

- (c) Process and resource management: SP also handles requests for process creations, interactions and terminations, procedure calls and the use of memories as well as other resources. A request list (RL) is maintained by SP to enqueue those requests that cannot be honored immediately.
- (d) Setting up of compound operations: Scalar operations need not go through SP and are executed by TPs autonomously; whereas compound operations have to be setup by SP. If the compound operation is an array operation, then SP will request a subset of the first R TPs to perform the operation under the demand and control of a subcontroller $TP(j)$, where $j > R$. In the case of a block of sequential program, SP will load it into $LM(k)$ -- where $R < k \leq M$ -- and request $TP(k)$ to execute it. In the former case, the choice of TPs will be specified by the subcontroller $TP(j)$ according to the compound operation it has received; in the latter case, the choice is arbitrary. The Channel Selector (CS) will be setup by SP to realize the above connections.
- (e) Other operating system tasks: SP may either execute these tasks directly, or regard them as application tasks and assign them to TPs. The choice depends on the nature of the O.S. tasks.

(2) Receiving Processors (RPs)

There are R RPs connected to the receiving side of the network PSN. A RP will continuously remove the arriving result packets from the network and update the contents of the LMs accordingly. The formats of the various types of result packets are listed in Table IV.6. A RP will respond to the content of a result packet as follows:

- (a) If it is an array element, then it will be stored into the memory location as specified by its destination address;
- (b) If it is a scalar operand, the base address of an array or a signalling token, then the receiving processor will update the instruction word given by the destination address of the packet, and it will then examine whether that instruction has received all the required information; if it has, then the instruction will be placed in an instruction register (IR) to wait for a free TP for execution (the selection of IR-TP pairs will be given in Section 2.B.(3)); otherwise no further action will take place.

(3) Transmitting Processors {TP(1) to TP(R)}

This group of TPs will execute both scalar and array operations. Any free TP belonging to this group will continuously check its associated instruction registers for

the addresses of executable instructions. If $IR(k)$ contains one, then $TP(k)$ will fetch the corresponding instruction from $LM(k)$ and execute it. The computed results together with the addresses of the next instructions will be packaged into result packets which are then forwarded to the network for distribution.

A subset of these TPs may undergo an array operation under the control of $TP(i)$ where $i > R$. When $TP(i)$ receives such a compound operation, it will generate and broadcast the control signals to these TPs via the Channel Selector (CS). As soon as these TPs have finished their current activities, they will respond by fetching the array elements from their LMs according to the broadcast signals. If the array operation is,

- (a) a computational activity, then these TPs will operate on the elements and then store the results back to the memories using the read/write links;
- (b) an alignment operation, then these TP will package the elements into result packets and forward them to the network for alignment. After the last element has been sent out, some of these TPs will be requested by SP to generate a synchronization token which will be forwarded to the network to indicate the end of transmission (this synchronization process will be described in Section 2.C(2)).

If a TP is not involved in or has just completed an array operation, it will resume its normal activities as mentioned in the beginning of this subsection.

(4) Transmitting Processors {TP(R+1) to TP(T)}

The main function of these TPs is to execute scalar operations; for those with LMs, they may be requested by SP to execute sequential programs as well.

Any free TP belongs to this group will continuously check its associated IR for executable instruction packets. Unlike the previous group, these TPs require that the actual instructions -- i.e., the opcodes, immediate operands and addresses of next instructions -- be available in the IRs, but not the addresses of the instructions, because these TPs do not have direct read/write links to access the first R LMs where the non-sequential programs are stored. The result computed by these TPs will be packaged into result packets which will then be forwarded to the network for distribution.

To initiate the execution of a sequential program, SP will select any free TP-LM pair of this group, and the program will be loaded into the LM, and the associated TP will be requested to execute it. Upon completion, that TP will either signal SP or produce a result packet to trigger other instructions via the network.

The number of TPs could be larger than or equal to

that of LMs, depending on the speeds of the various hardware modules and the intended applications.

2.B. Storage Modules

(1) System Memory (SM)

The aforementioned input and output buffers are located in SM which also contains application programs as well as system software such as I/O routines and interrupt service routines. While in SM, all the addresses of a program will remain in the relative form so that the program could be relocatable; when copied into LMs, these relative addresses will be translated into absolute ones by the TPs connected to the LMs, using the base address provided by SP. If a program is to be called repeatedly, then a copy of it will be kept in SM for replication purposes.

SM also contains those aforementioned tables, namely, the storage utilization table (SUT), the array description table (ADT), the request list (RL), as well as a linkage information table (LIT) which provides the linkage information between a calling program and its called programs.

(2) Local Memories (LMs)

LM(1) through LM(R) are used to contain interleaved

instructions and skewed arrays. Their left ports are connected to SP and TPs while their right ports to the RPs. Contentions between RPs and TPs could be resolved by granting their requests in an alternating manner. $LM(R+1)$ through $LM(M)$ are used to store sequential programs which are to be executed solely by the associated TP.

At times SP will interrupt the above activities for the loading and unloading of programs; such interferences could be reduced by increasing the size of LMs so that most of those frequently needed programs could reside in them.

(3) Instruction Registers (IRs)

IRs serve as buffers between RPs and TPs. As has been mentioned in Section 2.A(3) and (4), $IR(1)$ through $IR(R)$ contain only the addresses of executable instructions while $IR(R+1)$ through $IR(T)$ contain the actual instructions; therefore, the buffering capacities of these two groups of IRs are different.

Associated with each IR are two single-bit flags: the "Full/Not-Full" flag which indicates the status of the IR, and the "Autonomous/Slave" flag which indicates the operating mode of the connected TP. An autonomous TP is one which is ready to accept or is currently executing instructions from IRs, while a slave TP is one which is undergoing a compound operation under the control of another processor.

To schedule an executable instruction, $RP(i)$ will examine the flags of $IR(i+n*R)$ in the order of increasing n which is a non-negative integer, and the first IR which is not full and is connected to an autonomous TP will receive the instruction packet.

2.C. Switches

(1) Channel Selector (CS)

CS enables SP to select any of the TP - LM pairs to perform those activities mentioned in Section 2.A(1), namely, program loading, input and output activities and setting up of compound operations.

The implementation of CS is quite straight-forward and hence will not be discussed in this dissertation.

(2) Packet Switching Network (PSN)

Other conventional packet switching networks could be used in place of PSN, but they require at least $(N/2)\log N$ switches for a $(N \times N)$ connection, whereas PSN uses only $(N/2)$ switches; therefore, PSN is attractive when N is very large.

PSN is a modified version of Loop-Structured Switching Network (LSSN) which has been described in Chapter III, and its functions are:

- (a) to deliver result packets from TPs to RPs and LMs;
- (b) to perform hardware synchronization to signal the completion of array alignments.

It is the second function above which distinguishes PSN from LSSN. The topology and addressing scheme of PSN are the same as that of LSSN; but in order to perform hardware synchronization on the network, the PSN switches have to be different from the LSSN switches. Fig.IV.3 illustrates the schematic diagram of a PSN switch.

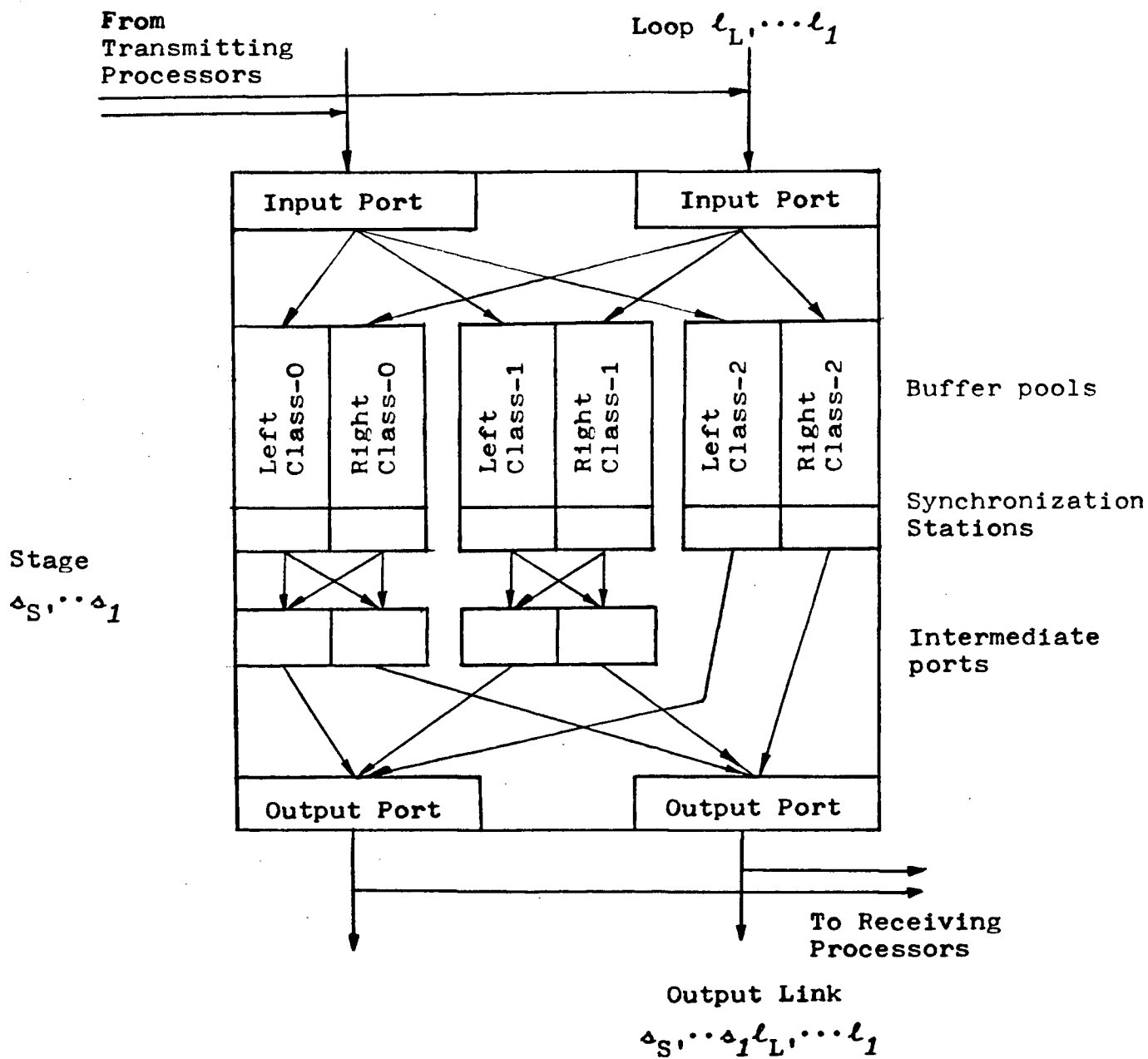


Fig.IV.3. The schematic diagram of a PSN switch.

In general, a result packet sent out by a transmitting processor would have the packet format as follows:

<Feedback Count; Destination Address; Result Type;
Result>

When a result packet enters the input port of a switch, it will be placed into the Class- i buffer inside the switch according to its feedback count $i \in \{0,1,2\}$, which is set to zero when the packet is initially generated, and is incremented whenever the packet goes through the feedback path. In Fig.IV.3, all types of result packets except the Synchronization packets, will bypass the Synchronization Stations when they emerge from the buffer pools. For a packet coming out of the Class-2 buffer, it will be forwarded to the output port immediately and directly when the latter becomes empty; packets coming out of the Class-0 and Class-1 buffers will be switched to an intermediate port to wait for their turns to be transferred to the output port. For a switch located in the s -th stage, the direction of switching is determined by the s -th bit of the destination address of the packet: if it is a "0", then the packet will be switched to the left intermediate port; else to the right one.

Because of the similarities that exist between the topologies of PSN and LSSN, those theorems developed for LSSN are also applicable to PSN. Theorem III.1 have shown that for a network with L loops, the maximum number of switches

that any packet would have to go through in order to arrive at its destination, is $(2\log L - 1)$. Consider the case in which a packet is admitted at the last stage of PSN and has to go through the maximum number of switches, $(2\log L - 1)$, then this packet will be removed from PSN when it emerges from a Class-2 buffer located in the $(\log L - 2)$ th stage -- which is the furthest destination any packet will have to go regardless of where it is originated; the significance of this observation will become obvious when we discuss the method of hardware synchronization on PSN. Another important property of PSN, as revealed by Lemma III.2, is that any packet which has already acquired a feedback count of 2 will always remain in the same loop for any of its further routing steps -- this explains why packets coming out of the Class-2 buffers in Fig. IV.3 need not go through the intermediate ports.

The purpose of the Synchronization Stations is to achieve the effect of hardware synchronization on PSN -- i.e., to signal the completion of array alignment operations so that other computation dependent on these operations may proceed. After all the elements involved in an alignment operation have been dispatched to PSN, each of the first L TPs (i.e., those TPs connected to the first stage of PSN) will be requested, by either SP or the subcontroller of the alignment operation, to forward a synchronization token in the form of a result packet. These packets will be treated much the same as other result packets except that they will be retained by the Synchronization Stations when emerging from the buffer pools;

a synchronization packet retained by the left (right) Class-1 station would have to wait for the arrival of another synchronization packet in the right (left) station of the same class, then both packets will proceed to the intermediate and output ports in a straight-through manner. Such a scheme would ensure that the synchronization packets will always lag behind the array elements which they are trailing, and that when these packets arrive at the Class-2 Synchronization Stations of the $(\log L - 2)$ th stage, all the array elements concerned must have been delivered to their destinations (as has been explained in the previous paragraph). Upon the arrivals of the synchronization packets, the Class-2 Synchronization Stations of the $(\log L - 2)$ th stage will transform them into signalling tokens by resetting their feedback counts to zero, and changing their result types (please refer to Table IV.6 for their formats); these signalling tokens will be retransmitted to trigger those instructions dependent on the completion of the array alignment operation, and their destination addresses are those originally carried by the synchronization packets.

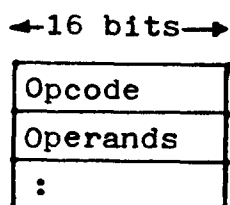
When a result packet arrives at an output port of a PSN switch, its destination address will be matched against that of the output link connected to the port. If a match occurs, then the RP connected to that link will be strobed and the result packet will be handed over to it; otherwise the packet will be forwarded to the switch situated at the other end of the link.

More details of PSN could be found in Chapter III which also explains how to expand the network incrementally -- a significant advantage of PSN over other conventional networks. Although a PSN switch has a much complex internal structure than a conventional binary switch, the savings in the number of switches as well as external wiring will offset such a disadvantage when the size of the network is large. With today's technologies, a high internal complexity could be easily achieved, but if a system involves too many external components, it will still be difficult to manage.

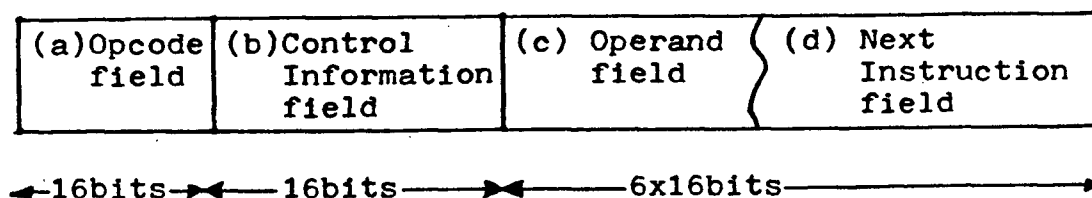
3. EDC Information Structure

3.A. Machine Instruction Formats

(1) Format for sequential programs: It is similar to that of conventional computer systems, and is arranged as one double-byte of opcode followed by either one or more double-bytes of operands.



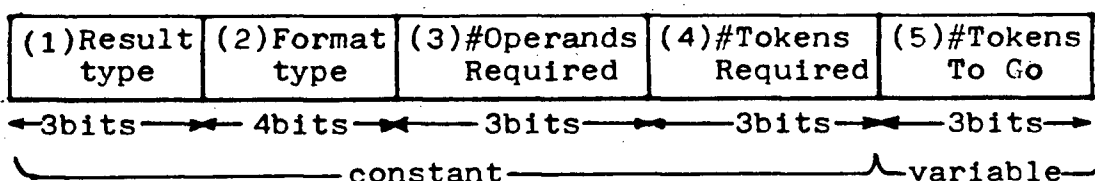
(2) Format for non-sequential programs: It is used to encode scalar and those encapsulated compound operations, and is made up of eight double-bytes which are divided into 4 fields:



The "Opcode" and "Control Information" fields are of one double-byte each, while the "Operand" and "Next Instruction" fields share the remaining six double-bytes.

(a) "Opcode" field: Tables IV.1 and 2 show the four categories of scalar and compound operations respectively, along with some typical examples and their data-flow graphs.

(b) "Control Information" fields: It is further divided into five subfields:



(1) "Result type": Specifies whether the computed result will be of single or double precision, a numerical or boolean value, or a signalling token.

(2) "Format type": Specifies the type of format used to accomodate operands and the addresses of those instructions dependent on the current instruction.

(3) "#Operand Required": Specifies the number of operands needed by the instruction.

(4) "#Tokens Required": Equals "#Operands Required" plus the total number of signalling tokens needed.

(5) "#Tokens To Go": Equals "#Tokens Required"

minus the number of tokens received.

When a RP receives an operand or a signalling token, it will decrement the "#Tokens To Go" of the receiving instruction; when this value reaches zero, the receiving instruction will be placed into an instruction register (IR) to wait for execution.

(c) & (d) "Operands" and "Next Instruction" fields:

The various types of formats used by scalar and compound operations are listed in Tables IV.3 and 4 respectively. These simple formats will meet almost all the computational needs; otherwise new formats could be added if necessary (a total of 4 bits are assigned to the "Format Type" field which could account for 16 formats).

In Table IV.3, "Opi" refers to the i-th operand of an instruction and "Nextj" refers to the address of the j-th next instruction, and "NextT" and "NextF" are the addresses of the next instructions when the result of a boolean operation is "True" and "False" respectively. Format No.8 is useful for those operations such as "Duplicate" and "Wait" which do not carry embedded operands.

In Table IV.4, "No. of elements" refers to the total number of array elements involved in the array operation, and "Stride" is the difference in the indexes of two neighboring array elements which take part in the operation. Both the

"No. of elements" and "Stride" are obtained from the loop control statements such as "DO I=1,64,2" or "FOR I=1to64step2 DO". In Table IV.4, "(V1)" is the base address assigned to the resulting vector V1, and "(V2)" and "(V3)" are those of the input vectors V2 and V3, respectively.

All compound operations except those "Reduction" ones would produce vectors which are too expensive (in terms of time and space) to be sent to each and every instruction requiring them; therefore, only the base addresses of the vectors will be sent. As for "Reduction" operations such as summation and product, their scalar results would be treated much the same as those produced by scalar operations.

Although the formats shown in Tables IV.3 and IV.4 have limited numbers of "Next Instructions" fields, their actual fan-outs could be extended infinitely by having one or more of their "Next Instructions" fields point to a number of "Duplicate" operators.

3.B. Packet Formats

There are two classes of packets that exist in EDC, namely,

- (a) Instruction packets: They flow from RPs to TPs and reside in IRs while waiting for execution. (Please refer to Table IV.5.)

- (b) Result packets: They are produced by TPs and are forwarded to RPs via the network PSN. (Please refer to Table IV.6.)

3.C. Program Organization

The EDC program organization is similar to those of the existing computer systems. Both the application and system software are made up of three types of program components:

- (1) Main programs: They are activated via external means such as the console and not to be called by other program components.
- (2) Procedures: They are activated by explicit calls from the program components. The calling programs use "Call" and "Distribute" operators and the called programs use "Distribute" and "Return" operators for parameter passing. As depicted in Fig.IV.4, when the "Call" instruction has gathered all its input tokens, it will be dispatched by a RP to a free TP which will then request the program code from SP. If the program code does not exist in LMs, then SP will allocate free memory pages to it and load it from SM to LMs, and its starting memory location will be returned to the requesting TP which will then proceed

with other computations. when the "Return" operator is executed, all the computed results will be routed back to the calling program and the memory pages assigned to the called program will be released.

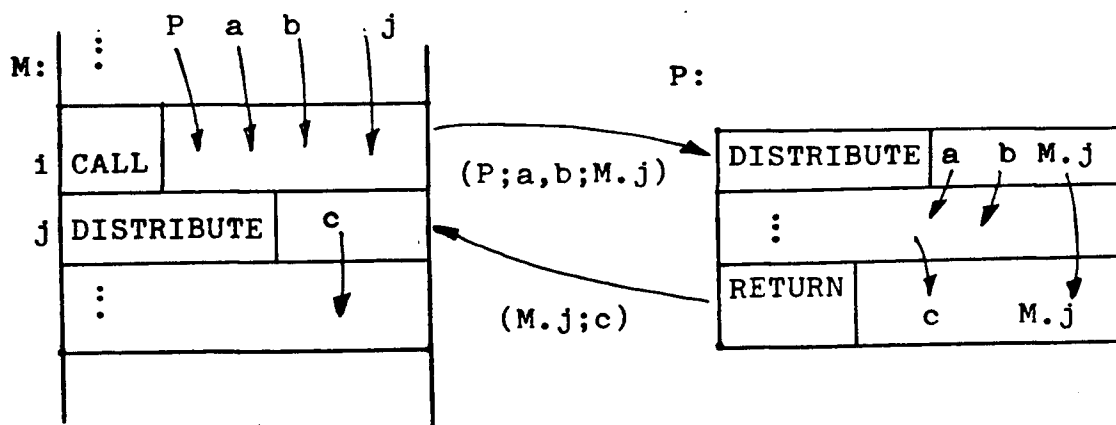


Fig.IV.4 Parameter passing between the calling program M and called program P. "a" and "b" are the input parameters and "c" is the returned result, and "j" is the return address.

- (3) Task programs: They are used to protect shared data and/or physical resources so as to ensure their proper use. A task program consists of one or more entry points whereby other programs could send data or signals to it, and therefore it is a means of providing communications and interactions among the various types of program components.

The implementation of parameter passing between a task and the calling programs is very much the same as that of procedure calling; the major difference is that a procedure is activated by an explicit call while a task program is

activated when the program which declares it comes into existence; also, a procedure terminates when the computed results are returned to the callers, whereas a task program may continue to serve other callers until an explicit termination statement is encountered, or the program which declared it has terminated.

```
e.g. Task t;
      :
      :
      Accept A(x:Real)Return(y:Real);
      :
      End A;
      :
      End t;
```

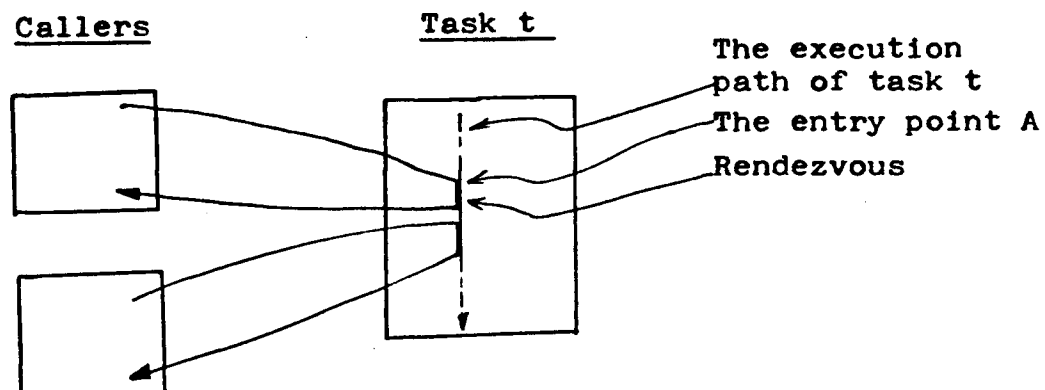


Fig.IV.5. The interactions between calling programs and a task program.

The example of Fig.IV.5 shows a task with a single execution path; but in general, a task could be "multi-threaded" -- i.e., made up of several concurrent execution paths.

The advantages of using task programs instead of low-level concurrency primitives such as semaphores [42] in

handling inter-program activities are ease of use and clarity. Furthermore, the implementation of tasks conforms to the principle of data-driven computation. Compared to other high-level constructs, a task is quite different from the "monitor" of Concurrent Pascal [38] but very similar to the "task" of Ada [44].

3.D. Data Structure

We only discuss arrays in this paper although some other more complicated structures [49,68] may also be considered in our design. The handling of arrays in an EDC is illustrated in Fig.IV.6.

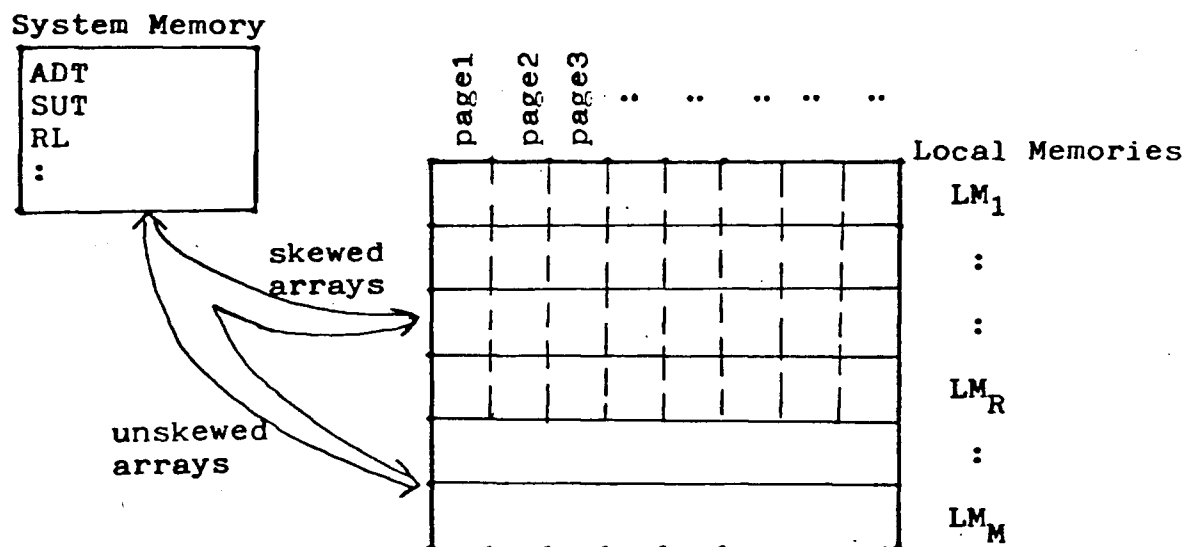


Fig.IV.6. The physical and logical arrangements of EDC memory system. The first R LMs are logically divided into pages as shown.

LM(1) to LM(R) are used to stored skewed arrays so

that they may be processed concurrently by TP(1) through TP(R). However, for reasons of efficiency or algorithmic constraints, an array may not be skewed but instead, will be either loaded entirely into a local memory LM(k) and processed by TP(k), or divided among several TP(k)-LM(k) pairs where $k > R$. The decisions concerning these arrangements could be made either statically at compile time, or dynamically by SP at run time. The array description table (ADT) and storage utilization table (SUT) must always be updated to reflect the storage patterns.

3.E. Process and Resource Management

In the EDC environment, a process is defined as either a main or task program in execution, and those procedures called and data structures owned by the program are regarded as parts of the process. The treatments of process creations and terminations are very similar to that of procedure calls: the request to create a process will be first placed on the Request List (RL) until it is removed by SP, which will then assign an unused identification number (ID) from the linkage information table (LIT) and free memory pages from the storage utilization table (SUT) to the process; SP will then load the memories allocated with the program code and initialize it to run. When the process terminates, SP will again update SUT and LIT accordingly.

As illustrated in Fig.IV.7, the management of hardware

and/or software resources could be implemented conveniently using a task program. The number of unused resources of a particular type (e.g., the number "N" of Fig.IV.7) is stored in a memory location which can only be accessed from within the critical region enclosed by the "Select" and "End Select" operators. In order to prevent malicious accesses to that memory location, only one request at a time would be allowed to enter the critical region to modify the number of the resources, and this is achieved with the use of a signalling token as shown. The content of that memory location is incremented whenever a "Release" request is honored and decremented whenever an "Acquire" request is granted.

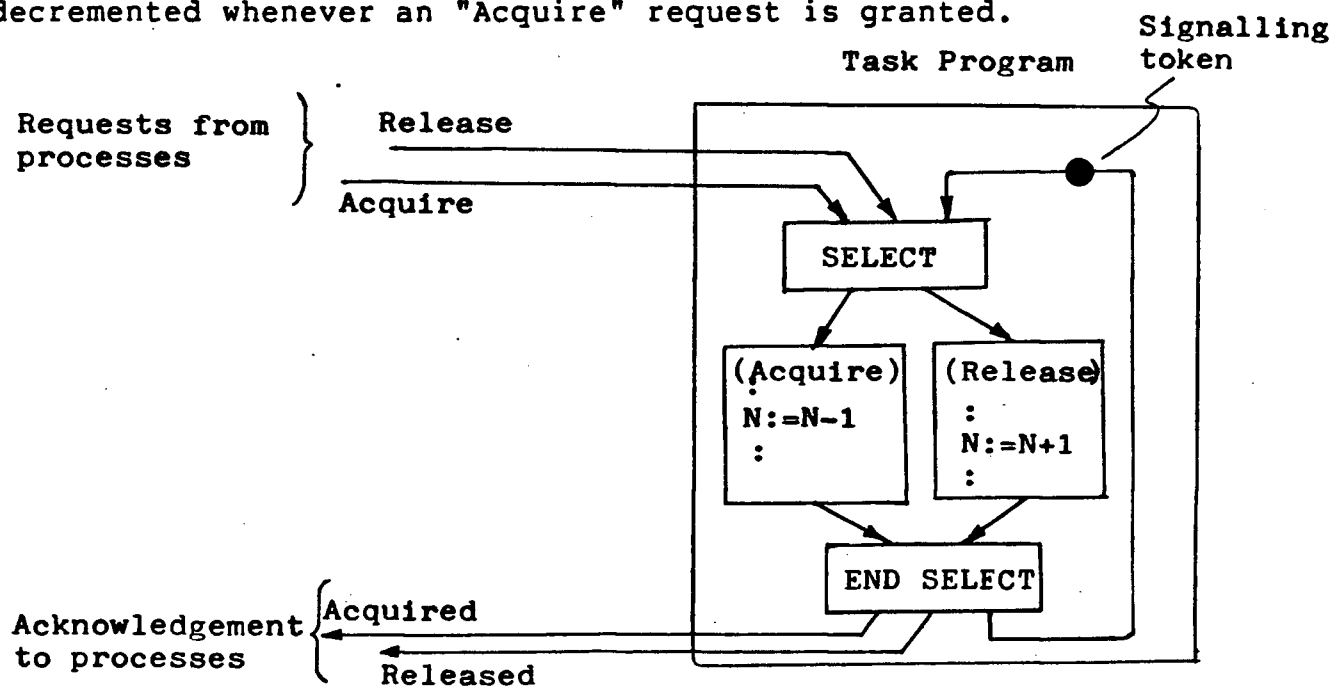


Fig.IV.7. The implementation of a resource manager using a task program.

The "Select" operator used in the resource manager of Fig.IV.7 does not conform faithfully to the data-driven principles, because its execution is triggered by the arrivals

of the signalling token plus at least one request -- not necessarily all of them. If there are several requests at the same time, then they will be enqueued when they arrive, and the selection policy for these requests could be either first-come-first-served or priority-based, depending on the implementation.

4. EDC Programming Language Structure

A program that runs on EDC takes the form of either a main program, a procedure or a task program, and it is composed of one or more "program blocks" which are collections of instructions that have no branching into or out of the blocks, except at the beginnings and endings. The advantages of using blocks are program clarity and that existing techniques of optimizing compilers could be used.

The objective of this section is to present some useful ideas concerning the design of the EDC programming language, and to illustrate how some language constructs are compiled into data-flow graphs which could be easily translated into machine code using the formats presented in Section 3.

4.A. EDC Statements and Program Blocks

(1) Declaration statements: Most of them are used to assist

the compiler in setting up the data-flow graphs and are not translated into executable operations. Exceptions are the declaration of task programs and arrays, which will be compiled into operations that will request SP for process creations and memory space respectively.

- (2) Assignment statements: In a conventional sequential program, variable names could be used repeatedly to represent different entities in different parts of the program without causing much confusion; however, such a "convenience" would often lead to obscurities in a concurrent environment. In the EDC system, the Single Assignment Rule (SAR) is used to avoid such confusions whenever necessary. The SAR simply states that a variable name must not be assigned more than one value within its scope; when applied to data-flow graphs, it means that each arc of the graphs could have atmost one source of origin.
- (3) "Begin/End" block: The "Begin" and "End" statements will be compiled into "Wait" operators as demonstrated in Fig.IV.8. The "Wait" operator is a means of imposing dependencies among program blocks so as to achieve the desired sequentiality not explicitly expressed by their data dependencies.

e.g. Begin
:
:
:
End;

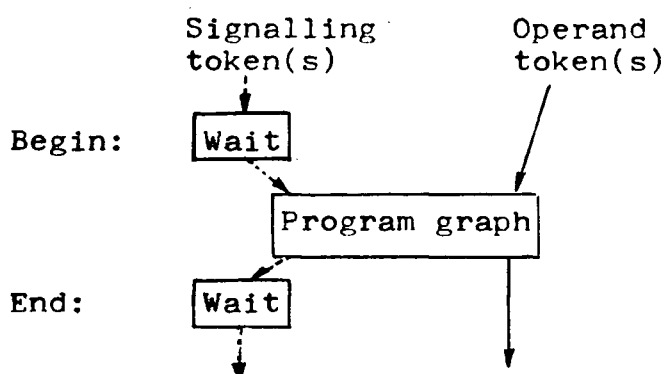


Fig.IV.8 A "Begin/End" block and its data-flow graph.

(4) "IF" block: It will be compiled into a boolean plus a number of "Switch"ing operators which are used to direct the flow of input operands into either the "IF" or "ELSE" part of the program. Sometimes certain emanating arcs have to be "grounded" in order to discard the unused operands after the conditional test.

e.g. IF(C1 \geq C2)THEN
:
:
ELSE
:
:
END;

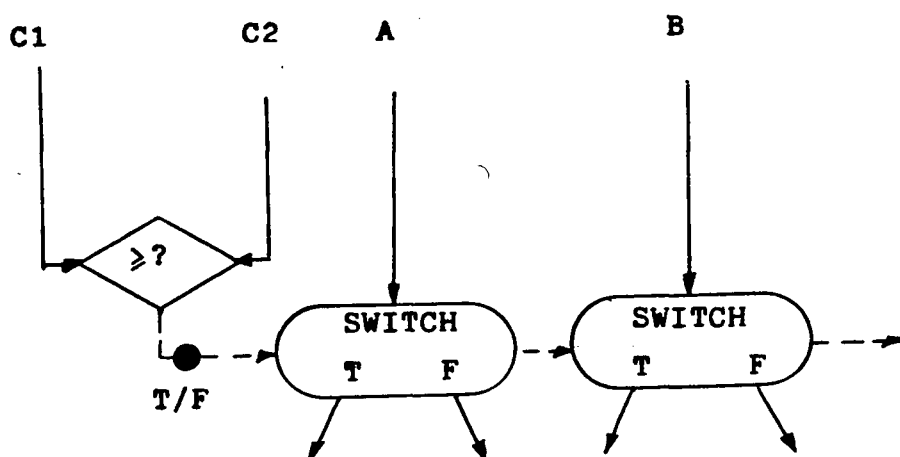


Fig.IV.9 An "IF" block and its data-flow graph.

(5) "Match" block: It will be compiled into a series of boolean and "Switch"ing operators as shown in Fig.IV.10. The input operand "C" will be matched against all the comparands "C1,C2," in parallel, and the "Switch"ing operators

will direct the operands to the part of the program which has a successful match. An "Else" part should be provided in case all the matches fail.

e.g. MATCH (C)
 CASE(C1)DO
 :
 CASE(C2)DO
 :
 ELSE
 :
 END;

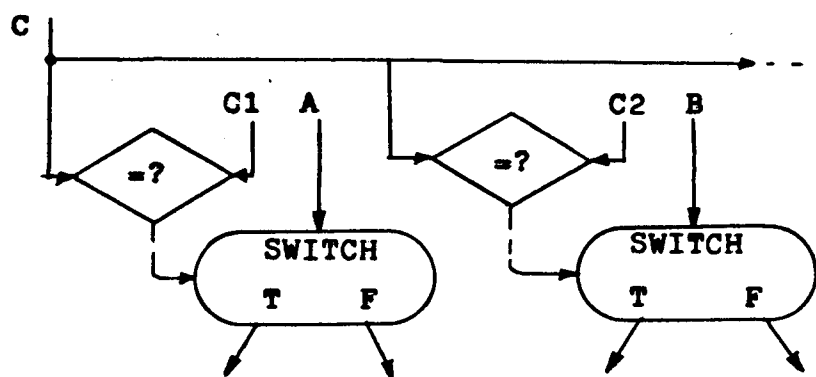


Fig.IV.10. A "Match" block and its data-flow graph.

- (6) "Loop" block: An EDC loop is different from the "For-all" and "Do-all" loops proposed in other data-driven languages [40]. Since parallel array computations in EDC are encoded as compound operations, it is not necessary to use "Loop"s for these computations; instead, loops are used for iterations and recurrence operations which exhibit data dependencies between two successive loop computations.

e.g. LOOP WHILE(C1=C2)DO

```

:
NEXT X := ...
NEXT C1:= ...
NEXT C2:= ...
:
LOOP EXIT
  XLAST := X;
:
END;

```

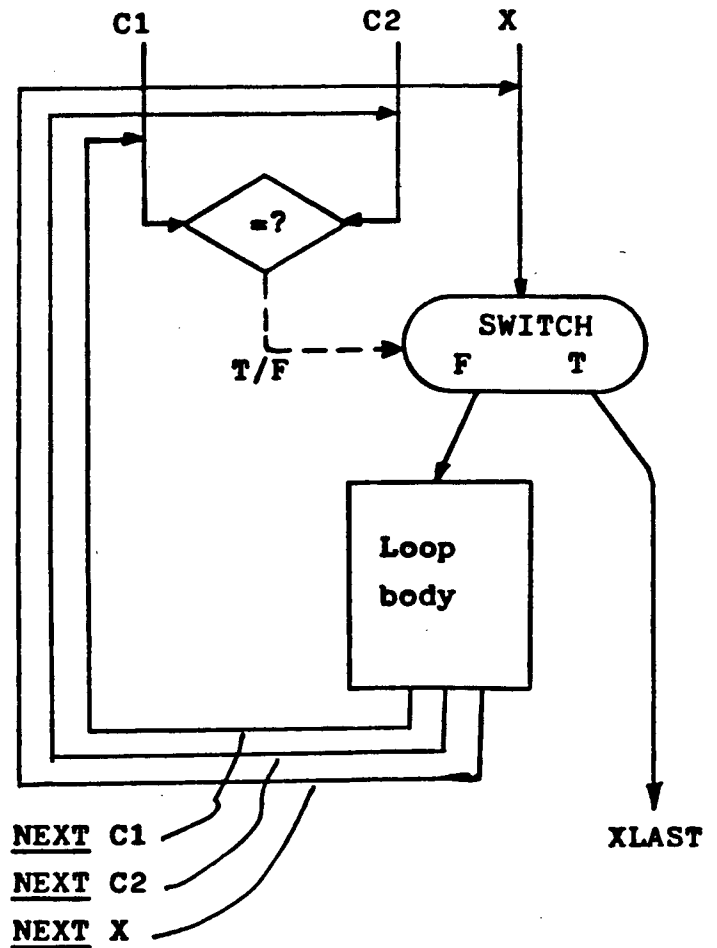


Fig.IV.11. A "LOOP" block and its data-flow graph.

As exemplified by Fig.IV.11, the data-flow graph of a EDC loop contains some arcs which have two sources -- one outside the loop and another one within the loop body -- implying that some variables are assigned more than once, thus violating the Single Assignment Rule. A common remedy [39,65] is to place prefixes such as "NEW" and "NEXT" in front of those variable names in question. Thus, "NEXT X" would be treated differently from "X" during a loop computation, but "NEXT X" will be updated as "X" at the boundary of two consecutive loop computations, and "NEXT X" will be an entirely new variable when the next loop computation commences.

When the conditional test associated with a loop is satisfied, the loop will be exited and then some of the computed results will be passed to the exterior of the loop by assigning them to names that are not used within the loop (e.g. "XLAST" of Fig.IV.11).

- (7) Prefix for sequentiality: As has been mentioned before, certain activities such as input and output are inherently sequential, and it is more convenient and efficient to execute their instructions in the order specified by the programs; and the "SEQ"quential prefix is meant for such purposes.

If an instruction block is prefixed with "SEQ", then signalling tokens would be used to enhance its data-

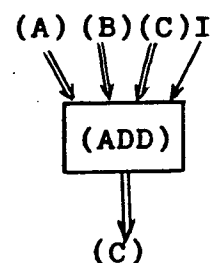
flow graph to achieve the desired sequentiality. If an entire program is prefixed with "SEQ", then it will be compiled into a sequential programs using the format mentioned in Section 3.A(1).

4.B. Language Constructs for Array Processing

Array operations are probably the richest source of synchronous parallelism and abound in scientific computations. This section will discuss how one-dimensional arrays are encoded and executed in EDC; arrays with a higher dimension will be reduced into one-dimensional arrays before their computations.

(1) Parallel Vector Operations: The range and stride of a parallel vector operation are indicated with the use of an "INDEX" set, which has to be declared prior to its use as follows:

```
e.g. DECLARE
      I: INDEX 1..64 STRIDE 2
      BEGIN
        :
        C(I) := A(I)+B(I);
        :
      END;
```



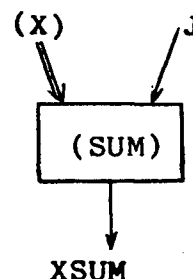
Opcode	Control	#Elements	Stride	Address of Output Array=(C)	Address of Input Array=(A)	Address of Input Array=(B)	Next2	Next1
(ADD)	Infor.	= 64	= 2					

Fig.IV.12. The statement, data-flow graph and machine code of a parallel vector operation. The machine code format is as described in Table IV.4.

Both the range and stride indicated in an index are regarded as input operands to the array operation, and they could be either constants or variables to be determined at run time. The base addresses of the input arrays (i.e., (A) and (B) of Fig.IV.12) are received from the preceding operations, and that of the output array (i.e., (c) of Fig.IV.12) is obtained from the memory manager prior to the execution of the operation, and is sent to the succeeding operations as an input operand. The execution of such operations has been described in Section 2.A.(1) and (3).

- (2) Reduction Operations: This is another type of array operations frequently encountered, and there are six of them, namely, "SUMmation", "PRODUCT", "MAXimum", "MINimum", "AND" and "OR".

```
e.g. DECLARE
      J: INDEX 1..1024
      BEGIN
        :
        xsum := SUM(x(J));
        :
      END;
```



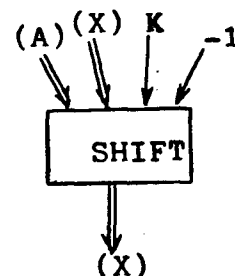
Opcode	Control	#Elements	Stride	Address of input Array=(X)	Next4	Next3	Next2	Next1
(SUM)	Infor.	=1024	=1					

Fig.IV.13. The statements, data-flow graph and machine code of a reduction operation. The machine code format is as described in Table IV.4.

When an array is to be reduced, its elements will be partitioned and loaded into several $LM(k)$'s -- where $k > R$ -- and each part of it will be processed by the associated TP independently; at the end of the computation, each of these TPs will forward its partial result in the form of a result packet, via the network to an instruction which will combine all the partial results. The number of TP-LM pairs used for a reduction operation depends on the size of the array and the speeds of the various hardware and software modules, and has to be optimized in order to obtain the shortest possible computation time.

- (3) Alignment operations: "SHIFT" and "ROTATE" are two alignment primitives: the "ROTATE" operator moves the array elements cyclically by the amount specified after the operator, while the "SHIFT" operator functions in a similar way except that there is no cyclic feedback of array elements, and zeroes are inserted into the positions vacated by the shifting operations. The direction of an alignment could be fixed arbitrarily; and if none of the operators is specified, then the "SHIFT" operation will be assumed.

e.g. $x := a(K \text{ SHIFT } -1);$
 $y := b(I \text{ ROTATE } 2);$
 $z := c(J -1);$



Opcode	Control	#Elements	Stride	Address of Output Array=(X)	Address of Input Array=(A)	Displace- ment= -1	Next2,1
SHIFT	Infor.	=1024	=1				

Fig.IV.14. The statements of some alignment operations, and the data-flow graph and machine code of the "SHIFT" operation.

5. Performance Analysis

5.A. Flow Analysis of EDC

In order to simplify the performance analysis of EDC and to arrive at some meaningful results, some assumptions will be made; later on, the justification of these assumptions will be discussed.

(1) Assumptions:

- (a) The problems to be processed by EDC comprise a large amount of concurrency which will keep all the TPs busy most of the time;
- (b) Initially all the computations are assumed to be of

scalar type; compound operations will be ignored temporarily;

- (c) The scalar operations are randomly distributed among the first R LMs, thus giving rise to approximately equal packet flow among the output ports of the network PSN.

(2) Constraints On Communications Loads:

As stated in Section 2.C(2), the maximum throughput rate (MATR) that can be delivered by a PSN with L loops is:

$$\text{MATR}(L) = 3/2 \times S_{R,SW} \times \log L \times L^2 / \{3L \log L - L + 4\} \dots (\text{IV.1})$$

(packets/second)

(3) Constraints On Processing Loads:

In order to prevent the instruction registers (IRs) from overflowing, the total processing speed of TPs must exceeds that of RPs, i.e.,

$$T \times \bar{S}_{I,TP} > R \times \bar{S}_{I,RP}$$

$$T > (\bar{S}_{I,RP} / \bar{S}_{I,TP}) \times R \dots (\text{IV.2})$$

where T and R are the numbers of TPs and RPs respectively, and $\bar{S}_{I,RP}$ is the average rate of producing instruction packets by

a RP, and $\bar{S}_{I,TP}$ is the average rate of consuming instruction packets by a TP.

For a PSN with L loops, we can connect up to a maximum of $N=L\log L$ pairs of TPs and RPs. If all the input ports are connected with TPs, then $T=N=L\log L$, and from expression (IV.2),

$$R < (\bar{S}_{I,TP}/\bar{S}_{I,RP}) \times L \times \log L \dots\dots\dots (IV.3)$$

If each RP is capable of accepting $\bar{S}_{R,RP}$ result packets from the network per second, then the maximum acceptance rate of result packets (MAR_P) by RPs per second is:

$$\text{MAR}_P = \bar{S}_{R,RP} \times R$$

Since most of the scalar instructions will be compiled into binary operations, meaning that on the average, the acceptance of every two result packets will cause one instruction to be readied for execution, i.e.,

$$\bar{S}_{I,RP} = \bar{S}_{R,RP}/2 \dots\dots\dots (IV.4)$$

If RPs are connected to all the output ports of the network, then $R=L\log L$ and the maximum acceptance rate of result packets of such a fully connected configuration (MAR_f) is:

$$\text{MARP}_f = R \bar{S}_{R,RP} = (L \log L) \bar{S}_{R,RP} \dots\dots\dots (\text{IV.5})$$

But the value of R is constrained by expression (IV.3); therefore, the maximum acceptance rate subjected to such a constraint is:

$$\text{MARP}_c < (\bar{S}_{I,TP} / \bar{S}_{I,RP}) \times L \log L \bar{S}_{R,RP}$$

Substituting $\bar{S}_{R,RP} = \bar{S}_{I,RP} / 2$ into the above expression,

$$\text{MARP}_{c,\max} = 2L \log L \bar{S}_{I,TP} \dots\dots\dots (\text{IV.6})$$

Expressions (IV.1, 5 and 6) show that the EDC performance depends on the network size L , the speeds of TPs, RPs and the switches.

5.B. Example:

Let us consider some typical values for the speeds of the various hardware modules, and then examine the EDC performance as a function of the network size, L .

We will assume that each TP and RP is capable of 3

MOPS (million operations per second) on the average -- this assumption is fair and has also appeared in other studies (for instance, see reference [30]). Since most of the scalar instructions require two result packets as their input operands, approximately half of the result packets coming out of PSN will not trigger their receiving instructions for execution; to process such a result packet, a RP would need one operation to retrieve the token count (i.e., "#Tokens To Go") of the receiving instruction, one operation to decrement it, one operation to store it back and another one to store the result token -- a total of four operations. The other half of the result packets would ready the receiving instructions for execution; to process such a result packet, a RP would require eight more operations to transfer an 8-byte instruction word from its associated LM to an IR, in addition to the four operations mentioned above. Therefore, the average number of operation needed to process a result is $(4 + (8+4))/2=8$, and hence the value of $\bar{S}_{R,RP}$ equals $(3 \times 10^6 / 8)$ packets per second. As for TPs, let us assume that the total number of operations needed for a TP to fetch an instruction from an IR, execute it, package the result into a result packet, and then forward it to the network, is around 40 (one might try other values); therefore $\bar{S}_{I,TP} = (3 \times 10^6) / 40$ packets per second. As for the speed of the PSN switches, $\bar{S}_{R,SW}$ equals " f/t_{min} " where " f " is the clocking frequency of the network, and " t_{min} " is the minimum number of clock pulses needed to transfer a packet from the output port of a switch

to the input port of the next switch. In this example, t_{min} is assumed to be 10 while f is taken to be 40 MHz; therefore, the value of $\bar{S}_{R,SW}$ equals $40 \times 10^6 / 10 = 4 \times 10^6$ packets per second.

The $MART$, $MARP_f$ and $MARP_{c,max}$ curves of this example are plotted against the network size L , and are shown in Fig.IV.15.

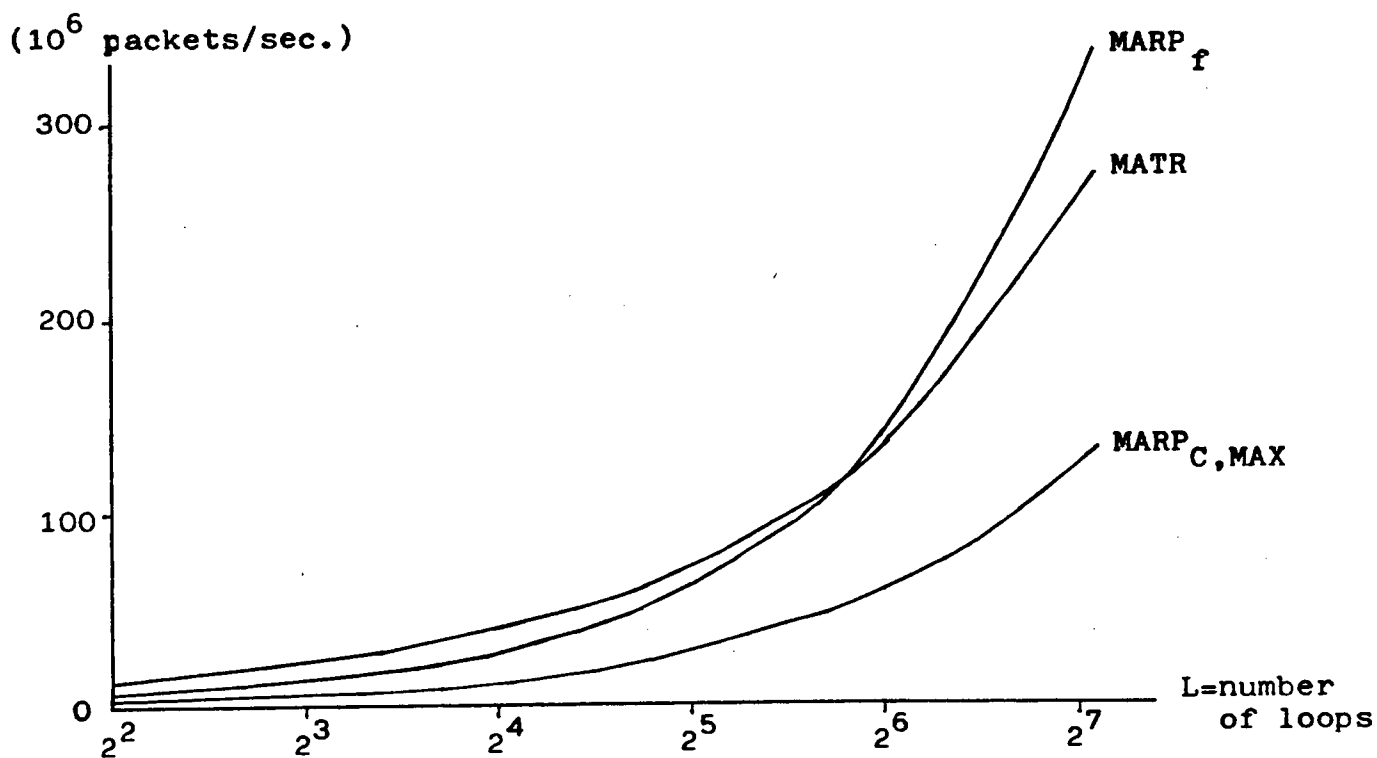


Fig.IV.15. The $MARP_f$, $MATR$ and $MARP_{c,max}$ curves of the given example.

In this example, the maximum throughput rate attainable by the EDC is limited by the $MARP_{c,max}$ curve which is the lowest among the three curves. Two observations could be obtained from Fig.IV.15:

- (a) The relative positions of the $MARP_f$ and $MARP_{c,max}$ curves suggest that in this example, it is not necessary to connect all the output ports of the PSN with RPs. For $L=64$, then $T=L\log L=64 \times 6=384$; from expression (IV.2):

$$R_{max} = (\bar{S}_{I,TP} / \bar{S}_{I,RP}) \times T = (3 \times 10^6 / 40) / (3 \times 10^6 / 8) \times 384 = 154.$$

- (b) The relative positions of $MART$ and $MARP_{c,max}$ curves indicate that for most of the time, the capacity of PSN will be higher than that required, and hence the extent and probability of traffic congestion in the PSN is expected to be low.

If there are always computations to keep all the TPs busy, then the raw speed attainable by the EDC is:

$$T \times 3 \times 10^6 = 384 \times 3 \times 10^6 = 1152 \text{ (MOPS)}$$

From Fig.IV.15, the maximum rate of flow of result packets in the system is:

$$MARP_{c,max}(L=64) = 57.6 \times 10^6 \text{ (packets/sec.)}$$

and the maximum rate of flow instruction packets is:

$$R\bar{S}_{R,RP} = R\bar{S}_{I,RP} / 2 = 154 \times (3 \times 10^6) / 8 / 2 = 28.9 \times 10^6 \text{ (packets/second)}$$

The curves of Fig.IV.15 are useful in estimating the size of the EDC architecture for a desired speed, and it also indicates which part of the architecture will be the most likely performance bottleneck after the system has been built.

5.C. Considerations for Generalized Computations:

If the values of $\bar{S}_{I,RP}$, $\bar{S}_{I,TP}$ and $\bar{S}_{R,SW}$ are different from those in the previous example, then expectedly, different throughput curves and conclusions would be obtained.

In general, computations in EDC will be made up of both scalar and compound operations which are mixed with unknown ratio, therefore, assumption (b) has to be removed for generality. The result of such a removal would give rise to a better performance because the execution of compound operations --i.e., array operations and sequential programs -- is control-driven, and requires simpler control structures and incur less communications overhead than scalar operations which are data-driven. Assumptions (a) and (c) are justifiable since EDC is intended for applications involving large amounts of concurrency; and the interleaving of instructions and skewing of arrays would spread the programs randomly and evenly among the TP-LM pairs.

6. Discussions and Outlook

A design methodology has been proposed for a class of next-generation supercomputers. Our proposal, which is named Event-Driven Computer (EDC), is primarily a data-driven system supplemented with control-driven activities. Although we do not emphasize the immediate implementation of EDC, most of its hardware are implementable with off-the-shelf components except the PSN switches which, however, could be easily fabricated with today's technologies. If the PSN (Packet Switching Network) consists of 64 loops, then approximately 400 TPs (Transmitting Processors) can be attached to the system, and the maximum rate of flow of instruction and result packets will be approximately 58 and 30 million per second, respectively; and the raw speed attainable by the EDC will exceed 1,000 MOPS. Since the proposed EDC language structure is similar to the existing ones [39,41,44,69], many of the techniques available today could be applied to its compiler design.

Compared to the Dependence-Driven System [32], the resources of EDC are better utilized: when a TP is not involved in an array operation, it could always get a scalar instruction from its associated IR (instruction register) and execute it. The array processing capabilities of EDC also distinguish it from the Combined System [33]. The speed range of EDC is expected to be many times higher than that of the PDF System [34].

Because only a few component types are used, the design costs of EDC are expected to be low. Since most of the programs are randomly distributed among the LMs thus equalizing the memory access load, the serious problem of memory bottlenecks could be reduced. As for array processing capabilities, array computation could be carried out by a subset of the first R TP-LM pairs using the read/write links provided; and the alignments of arrays could be performed using the PSN which provides a novel synchronization method to indicate the end of each array alignment.

Designing a supercomputer is not a simple task; we have presented some architectural ideas, but there are still several issues which deserve immediate attentions before the EDC concepts could become practical. Firstly, the detailed specifications of the EDC hardware have to be developed, and the division of labor between the compiler and hardware has to be clarified at the outset. Secondly, it is necessary to analyse the effects of run-time overhead (such as program loading) on the system performance, and remedies (such as increasing the size of LMs) have to be provided if the effects are severe. Thirdly, if several independent processes are run simultaneously, then an identification method is necessary (unique identification tags are often proposed for other data-driven systems [48]). Fourthly, the policies used by SP to schedule compound operations and processes play an important role in providing the processors with enough operations to

keep them busy; it is not sure whether there exist such policies and those found in the literature could be useful in this aspect. Fifthly, we have suggested that array computation be encoded as compound operations, but perhaps those operations on small arrays could be decomposed into scalar operations so that the load submitted to SP could be reduced; the criteria of such decompositions constitute another area of further study. Sixthly, it would be convenient to the users if more data structures -- such as records and lists -- are provided. Lastly, although studies on fault tolerance in packet communication architectures have been found in the literature [45], a specific study concerning EDC in this respect is indispensable. We feel that only after these above issues have been adequately dealt with, can a supercomputer then be built along the lines set forth for EDC.

Table III.1 - Scalar operations.

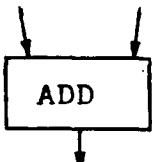
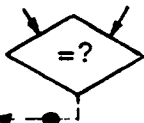
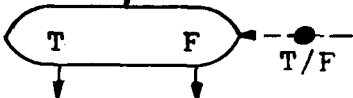
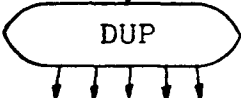

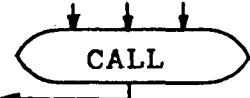
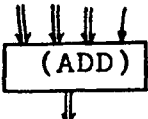
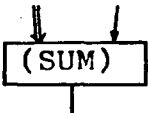
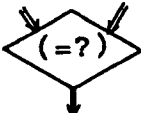
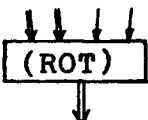
Scalar operation	Typical examples	Data-flow graph *
1.Arithmetic and logic	ADD, MUL,OR	
2.Boolean	EQUAL?	
3.Data transfer and control	SWITCH	
	DUPLICATE	
	WAIT	
4.Procedural calls	CALL,RETURN	

Table III.2 - Compound operations.

Compound operation	Typical examples	Data-flow graph *
1.Vector arithmetic and logic	(ADD), (MUL)	
2.Reduction	(SUM), (PRODUCT), (MAX), (MIN)	
3.Vector boolean	(EQUAL?)	
4.Alignment	(RIGHT SHIFT), (LEFT ROTATE)	

*See Section 4 on the use of these operators.

Table III.3 - The "Operand/Next instructions" fields of scalar operations.

Computation	Format No.	Operands/Next instructions					
Scalar Arithmetic, Logic and Procedure Call	1	Op1	Op2	Op3	Op4	Next2	Next1
	2	← Op1 →		← Op2 →		Next2	Next1
	3	Op1	Op2	Next4	Next3	Next2	Next1
	4	← Op1 →		Next4	Next3	Next2	Next1
	5	Op1	Next5	Next4	Next3	Next2	Next1
Boolean	6	← Op1 →		← Op2 →		Next _T	Next _F
	7	Op1	Op2	Next _T	Next _T	Next _F	Next _F
Data Transfer & Control	8	Next6	Next5	Next4	Next3	Next2	Next1

←16bit→ × 16 × 16 × 16 × 16 × 16 →

Table III.4 - The "Operand/Next instructions" fields of compound operations.

Computation	Format No.	Operands/Next instructions						
Vector Arithmetic and Logic	9	No. of elements	Stride	(V1)	(V2)	(V3)	Next2	Next1
Vector Boolean	10	No. of elements	Stride	(V1)	(V2)	(V3)	Next _T	Next _F
Alignments	11	No. of elements	Stride	(V1)	(V2)	Dis- place- ment	Next2	Next1
Reduction	12	No. of elements	Stride	(V2)	Next4	Next3	Next2	Next1

← 8-bit → × 8 × 16 × 16 × 16 × 16 × 16 →

Table III.5 - The formats of instruction packets.

Packet Content	Packet Format
a. Instruction Address	<Address of instruction>
b. Actual Instruction word	<Opcode;result & format types;operands; Next instruction addresses>

Table III.6 - The formats of result packets.

Result Type	Packet Format
a. Scalar Operand b. Array Element being aligned c. Base Address assigned to an array	<Feedback Count;Destination Address;Result Type;Result>
d. Signalling Token	<Feedback Count;Destination Address;Result Type>
e. Synchronization Token	<Feedback Count;Destination Address;Result Type>

Chapter V. Conclusions

1. Summary of Results

In Chapter II, we have presented a re-circulating systolic sorter (RSS) and two algorithms which work on RSS. The correctness of the algorithms has been proved and general operational constraints have been derived. This design is highly amenable to VLSI implementations due to the following attributes: (1) the simple control structure required by the algorithms; (2) the regular, repetitive and near-neighbour type of interconnections among the comparators; and (3) the systolic data movements. The sorting array is also well-suited for fabrication on shift-register type of storage and logic devices such as magnetic bubble memories (MBMs) and charge-coupled devices (CCDs), because of its closed-loop structure. The number of quadruple comparators needed to sort N items is $N/4$, and the average number of sorting cycles, as found by our simulation studies, is within the range $[(\log N)^2, N]$. A hardware termination method is incorporated into the control unit of the sorter, so that the sorting process can be terminated as soon as the input list is in the desired order.

Chapter III describes a novel loop-structured switching network (LSSN) intended for packet communications in highly parallel applications. With L loops, it can connect up to $N = L \log L$ pairs of transmitting and receiving devices, using only $N/2$ two-by-two switching elements. Therefore, it

is very cost-effective in terms of its component count. Its topology resembles that of the indirect binary n-cube network[21], but a much higher device-to-switch ratio can be achieved by LSSN because all the links between the switches could be used as both transmitting and receiving stations. It has the advantage of incremental extensibility, and it is free of the store-and-forward type of deadlocks which prevail in other cyclical packet-switched networks. Our simulation studies have shown that the average throughput rate and delay of LSSN are close to that of other designs despite its relatively low component count.

Chapter IV describes a new design methodology for the next-generation computers. Our proposal, the Event-Driven Computer (EDC) is primarily a data-driven, heterogeneous system which is supplemented with control-driven activities; such a combined approach is aimed at extracting the advantages of both the "pure" data-driven and control-driven systems while alleviating their shortcomings. Compared to other designs, EDC has the advantages of a simpler architecture, better resource utilization, array processing capabilities and a higher speed range. The LSSN of Chapter III has been modified for this application; with a configuration of 64 loops, this network can connect up to approximately 400 processors, and hence an execution speed of more than 1,000 million operations per second can be obtained by the EDC.

2. General Discussions

The main theme of this thesis is to demonstrate the practicality and usefulness of cyclical architectures in the designs of high-performance processors and computers. Our ideas have been illustrated through the use of specific application examples including parallel sorting, packet-switched communications and the design methodology of a novel, next-generation computer.

The ideas of feedback in our proposals are entirely different from that of process control, which uses feedback signals for correctional purposes (i.e., additive or multiplicative manipulations of the input signals); in the RSS arrays, feedback allows data items to be further compared among themselves until the whole input list is sorted; in the network LSSN, the sole purpose of feedback is to re-use the network resources until the packets are routed to their destinations, but there is no direct interaction (such as comparisons in the RSS arrays) among the information packets, other than competitions for the network resources; in the EDC, the arrivals of result packets in the feedback path signify the completion of one or more instruction cycles, and as a result, new instructions may or may not be brought into the computation path for executions, depending on the amounts of information they have gathered. The manners in which feedback packets interact with each other contribute greatly to the properties of the cyclical architectures. For

instance, the LSSN is susceptible to the store-and-forward type of deadlocks (we have, however, demonstrated how this problem can be solved), but the deadlock problem do not exist in the RSS, because data movements in the RSS network take place along specific paths and there is no data path conflict; in the case of the EDC, if there are always memory locations available in the Local Memories to await the result packets coming out of the network (i.e., if the programs are correctly written, compiled and loaded), then the EDC system should be deadlock-free. Another property of packet-switched, cyclical architectures is their lack of responsiveness -- interrupts cannot be processed immediately because the computation path could already be congested with packets when the interrupts occur; while in the EDC, direct read/write links connecting the Transmitting Processors and the Local Memories allow programs to be executed in a control-driven manner, without going through the PSN and the feedback path; therefore, fast execution and hence short response times could be expected. In general, resources in cyclical architectures are better utilized when compared to those in the acyclic systems.

3. Suggestions for Further Work

In this thesis, we have developed some ideas based on several new architectural concepts and demonstrated their practicality and usefulness. We have not implemented any of the proposals, because we feel that more related work has yet to be done. Specific topics for further research have been

suggested in the previous chapters; in particular, work concerning the detailed hardware specifications and fault tolerance studies of the three designs, should perhaps receive the utmost attentions, since our proposed systems are designed to make use of hundreds to thousands of interconnected processing and storage components, failures of single components will paralyse the entire systems, and these important issues are not included in our studies.

Appendix A

Lemma III.1: Consider a LSSN which has L loops and a packet which is destined for the address $\Delta_{S'} \dots \Delta_1 \ell_L \dots \ell_1$, where $L' = \log L$ and $S' = \lceil \log L' \rceil$. The packet will be routed to the loop $\ell_L \dots \ell_1$ within L' steps of routing after its admission into the LSSN.

Proof: Suppose the packet is admitted into the LSSN via the loop $\ell'_L \dots \ell'_s \dots \ell'_1$. According to the routing scheme, this packet will be routed to the loop $\ell'_L \dots \ell'_s \dots \ell'_1$ by a switch located in the s -th stage, where $s = 1, 2, \dots, L'$. Since the maximum value of s is L' , only L' steps of routing are required to route the packet to the aforementioned loop $\ell_L \dots \ell_s \dots \ell_1$.

Lemma III.2: Consider a LSSN with L loops and a packet which is destined for the address $\Delta_{S'} \dots \Delta_1 \ell_L \dots \ell_1$, where $L' = \log L$ and $S' = \lceil \log L' \rceil$. After the packet has been routed to the loop $\ell_L \dots \ell_1$, it needs at most another $(L'-1)$ steps of matching along that loop to reach its destination.

Proof: According to the routing scheme, the destination

address $s, \dots, s_1, \ell_L, \dots, \ell_1$ is one of the L' output links along the loop ℓ_L, \dots, ℓ_1 . After the packet has been switched to this loop, it will be removed by either the receiver attached to the link which is part of that loop, or one of the remaining $(L'-1)$ receivers attached to the same loop. In either case, at most $(L'-1)$ steps of matching are necessary.

Theorem III.1: In a LSSN with L loops, a packet will be delivered to its destination within $(2\log L - 1)$ steps of routing regardless of where it is generated.

Proof: This theorem is a result of Lemma 1 and 2.

Theorem III.2: The average number of routing steps (ARS) needed to deliver a result packet in a LSSN with L loops is,

$$ARS(L) = (3\log L - 1)/2 + 2/L - 1$$

Proof: Without loss of generality and for simplicity, we shall consider a transmitter (Tr) located in the first stage of the network. The routes from this Tr to the set of output links which can be reached without going through the feedback paths, as shown in Fig.III.2, is in the form of a "binary tree" which branches out toward the lower end of the network; the routes to the remaining set of output links would include the feedback paths, and is in the form of an irregular, "tapering" tree. The number of routing steps needed to reach

the receivers on these two trees are tabulated in Table III.1.

From Table III.1, the average number of routing steps needed for a Tr to reach any output link is therefore,

$$\begin{aligned}
 \text{ARS}(L) &= \{(2 \times 1 + 4 \times 2 + \dots + L \log L) + (L-2)(\log L + 1) + (L-4)(\log L + 2) + \dots \\
 &\quad + (L/2)(2 \log L - 1)\} / \{(2 + 4 + \dots + L) + (L-2) + (L-4) + \dots + (L/2)\} \\
 &= \{(L \log L - 2 \log L + L) + (L \log L - 4 \log L + 2L) + (L \log L - L/2 \log L + \\
 &\quad L(\log L - 1) + L \log L\} / \{L \log L\} \\
 &= \{L(\log L + (\log L + 1) + (\log L + 2) + \dots + (\log L + \log L - 1)) - \\
 &\quad \log L(2 + 4 + \dots + L/2)\} / \{L \log L\} \\
 &= \{L \log L(3 \log L - 1) / 2 - 2 \log L(2^{**}(\log L - 1))\} / \{L \log L\} \\
 &= (3 \log L - 1) / 2 + 2/L - 1
 \end{aligned}$$

Q.E.D.

Table.III.1 - The number of routing steps needed to reach the receivers of the "binary" and "tapering" trees.

Stage # of Rr	Binary tree		Tapering tree	
	#Rrs	#steps	#Rrs	#steps
stage 1	2	1	L-2	logL+1
stage 2	4	2	L-4	logL+2
:	:	:	:	:
stage(logL-1)	L/2	logL-1	L/2	2logL-1
stage(logL)	L	logL	0	2logL

Corollary III.1: Any packet admitted into LSSN will go through

the feedback path at most twice.

Proof: Consider a transmitter(Tr) which sends a packet at the s -th stage to a receiver(R_r) is of r routing steps away; and suppose there are L' stages in the network. The number of feedbacks, F , could be calculated as:

$$F(L) = \text{Quotient}((s+r-1)/L')$$

The reader may verify the correctness of this expression with a simple example on Fig.III.2. The maximum value of F is therefore,

$$F(L)_{\max} = \text{Quotient}((s_{\max} + r_{\max} - 1)/L')$$

Since $s_{\max} = L'$ and $r_{\max} = 2L'-1$ (from Theorem III.1), then

$$\begin{aligned} F(L)_{\max} &= \text{Quotient}((3L' - 2)/L') \\ &= 2 \end{aligned}$$

Q.E.D.

Theorem III.3: For a LSSN with L loops, the probability that the destination address carried by a result packet will match the label of an output link, and hence the packet will be removed from the network is:

$$P_{\text{removed}} = 2L / \{3L \log L - L + 4\}$$

where the transmission pattern is such that each and every receiving port of the network is equally likely to receive that packet.

Proof: Since the LSSN has L loops, it would have $\log L$ stages of switches and $L \log L$ pairs of transmitting ports (TPs) and receiving ports (RPs). Consider the case in which a TP in each stage of the network transmits a result packet to each and every RP in the network, then the number of packets transmitted by each stage of RPs is as tabulated in Table III.2.

In Table III.2, "Feedback Count" is the number of times the packets will go through the feedback paths in order to reach their destinations. The correctness of this table could be verified on the example given in Fig. III.3. From this table, the total number of packets that will be received by the RPs connected to a particular stage, say the last (i.e., $\log L$ -th) stage, is obtained by summing up the numbers across the corresponding row of the table, and it is:

$$N_{\text{matched}} = L \log L$$

and the total number of switching operations performed by the

same stage is:

$$N_{\text{total}} = N_{\text{matched}} + N_{\text{unmatched}}$$

where $N_{\text{unmatched}}$ is the number of packets which will not be removed by RPs of that stage because of unmatched destination addresses carried by them. In the case of the $(\log L)$ -th stage, $N_{\text{unmatched}}$ could easily be computed as the sum of products of the entries and their respective "Feedback count"s in Table III.2:

$$\begin{aligned} N_{\text{unmatched}} = & 1 \times \{ (L-2) + (L-4) + \dots + (L-L/2) + (L-L) \\ & + L + (L-2) + \dots + (L-L/4) + (L-L/2) \\ & + L/2 + L + (L-2) + \dots + (L-L/8) + (L-L/4) \\ & + \dots \\ & + 4 + 8 + 16 + \dots + (L-4) + (L-2) \} + \\ & 2 \times \{ (L-L/2) \\ & + (L-L/4) + (L-L/2) \\ & + (L-L/8) + (L-L/4) + (L-L/2) \\ & + \dots \\ & + (L-4) + (L-8) + \dots + (L-L/2) \} \end{aligned}$$

Let $N' = (L-2) + (L-4) + \dots + (L-L/2)$, then after re-arrangement,

$$\begin{aligned} N_{\text{unmatched}} &= N' \\ &+ L + N' \\ &+ L/2 + L + N' + (L-L/2) \\ &+ L/4 + L/2 + L + N' + (L-L/2) + (L-L/4) \\ &+ \dots \end{aligned}$$

$$\begin{aligned}
& +4+8+\dots+L/2+L+N'+(L-L/2)+(L-L/4)+\dots+(L-4) \\
& =N'\log L+(1+2+3+\dots+\log L-1)*L \\
& =\{L(\log L-1)-(2+4+8+\dots+L/2)\}*\log L \\
& \quad +\{L(1+\log L-1)(\log L-1)/2\} \\
& =\{L(\log L-1)-(L-2)\}\log L+\{L\log L(\log L-1)/2\} \\
& =\{3L(\log L)**2\}/2-\{3L\log L\}/2+2\log L
\end{aligned}$$

$$\Rightarrow N_{\text{total}} = \{3L(\log L)**2\}/2 - \{L\log L\}/2 + 2\log L$$

$$\Rightarrow P_{\text{removed}} = N_{\text{matched}} / N_{\text{total}}$$

$$\begin{aligned}
& = (L\log L) / \{ (3L(\log L)**2)/2 - (L\log L)/2 + 2\log L \} \\
& = \{2L\} / \{ (3L(\log L)**2)/2 - L + 4 \}
\end{aligned}$$

Q.E.D.

Theorem III.4: The maximum average throughput rate (MATR) of a LSSN with L loops is:

$$\text{MATR}(L) = 3/2 \times S_{R,SW} \times \log L \times L**2 / \{3L\log L - L + 4\}$$

where $S_{R,SW}$ is the maximum rate of transmitting Result packets

between two Switches via an output link.

Proof: Sincere there are $L \log L$ links in a L -looped LSSN, therefore, the maximum average rate of delivering packets to all the receiving ports could be formulated as:

$$\text{MATR}(L) = L \log L \times S_{R, SW} \times (1 - P_{\text{conflicted}}) \times P_{\text{removed}}$$

where $P_{\text{conflicted}}$ is the probability that an output link will not contain a packet due to conflicts within the switch concerned, and it could be computed with the illustrations below:

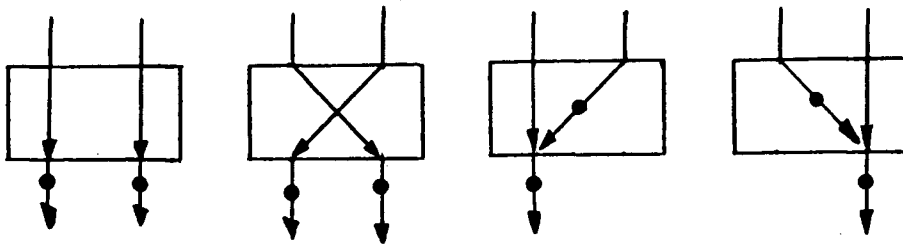


Fig.III.6.

On the average, 25% of the time an output link will not receive any packet due to conflicts in the switch, therefore,

$$1 - P_{\text{conflicted}} = 3/4$$

and also,

$$\begin{aligned} \text{MART}(L) &= 3/4 \times S_{R, SW} \times \log L \times L^2 / \{3L \log L - L + 4\} \\ &= 3/2 \times S_{R, SW} \times \log L \times L^2 / \{3L \log L - L + 4\} \end{aligned}$$

Q.E.D.

Theorem III.5: The LSSN which uses Type-B switches is deadlock free.

Proof: Type-B switches provide two essential features in avoiding deadlocks in LSSN:

- (a) The intermediate ports are used to hold packets with feedback counts of 0 and 1, such that they are not eligible to contend for the output ports if they cannot be switched to the next stage immediately, i.e., if the buffer pools of the next switch has no room to accept them.
- (b) The feedback counts of the packets emerging from the last stage are incremented so that when they are fed back to the first stage, they will request buffers of the next higher class.

The first feature ensures that links that are shared by packets with various feedback counts will not be clogged. The second feature prevents the formation of any cyclical request loop. With these two features, the path traversed by any packet in the network is "spiral" rather than "cylical" in shape, and the whole network could be conceived as several spirals interconnected in parallel, with the Class-0 buffers of the first stage as the heads of the spirals, and the Class-2 buffers of the last stage as the tails. Since there is no

cyclical request of resources, the network is therefore deadlock free.

Q.E.D.

Appendix B

```
PROGRAM RECIRCULATING-SYSTOLIC-SORT;
```

```
CONST
```

```
  INIT_SEED      = 3;  
  MARKER_COLUMN  = 1;
```

```
  RANGE          = 100.0;
```

```
  (* LARGEST RANDOM NUMBER TO BE SORTED *)
```

```
TYPE
```

```
  STORE_RECORD   = RECORD  
    MARKER : BOOLEAN;  
    ITEM : INTEGER  
  END;
```

```
  COMPARATOR_RECORD = RECORD  
    INIT_ROW, INIT_COLUMN : INTEGER;  
    AI, AJ, BI, BJ, CI, CJ, DI, DJ : INTEGER;  
    TEMPORARY : INTEGER  
  END;
```

```
  DATA_ARRANGE_TYPE = (RRANDOM, SSSEQUENTIAL);
```

```
VAR
```

```
  N_COMP, H_COMP, V_COMP : INTEGER;  
  LIMIT_NO_SWITCH : INTEGER;  
  ROW, COLUMN : INTEGER;  
  STORE : ARRAY (. 0 .. 50, 0 .. 50 .) OF STORE_RECORD;  
  COMPARATOR : ARRAY (. 1 .. 200 .) OF COMPARATOR_RECORD;  
  TOTAL_CYCLE, SWITCH_PER_CYCLE : INTEGER;  
  SEED : INTEGER;  
  J1, I, J, K, JJ, ODD_START, EVEN_START : INTEGER;  
  TEMPA, TEMPB, TEMPC, TEMPD : INTEGER;  
  TEMP_COL : INTEGER;  
  MARKERA, MARKERB, MARKERC, MARKERD : BOOLEAN;  
  TERMINATE : BOOLEAN;  
  DATA_ARRANGE : DATA_ARRANGE_TYPE;  
  DECR : INTEGER;  
  CONTINUOUS_NO_SWITCH : INTEGER;
```

```
PROCEDURE SETUP_NETWORK;
```

```
  BEGIN (* SETUP_NETWORK *)
```

```
    ROW := 2 * V_COMP;  
    COLUMN := 2 * H_COMP;  
    N_COMP := V_COMP * H_COMP - TRUNC (H_COMP / 2);  
    (* **NUMBER OF COMPARATORS** *)  
    LIMIT_NO_SWITCH := 2 * H_COMP + 4; (* TERMINATE IF NO SWITCHING  
                                         CONTINUOUSLY *)
```

```
    I := 0;
```

```
    J := 0;
```

```
    FOR K := 1 TO N_COMP DO
```

```
      WITH COMPARATOR (. K .) DO
```

```
        BEGIN
```

```
          (* ** EACH COMPARATOR WILL HOLD 4 ITEMS TO BE SORTED* *)
```

```
          INIT_ROW := I;
```

```
          INIT_COLUMN := J;
```

```
          I := I + 2;
```

```

        IF I >= 2 * V_COMP - 1 THEN
            BEGIN
                I := I - 2 * V_COMP + 1;
                J := J + 2
            END
        END
    END (* SETUP_NETWORK *);

FUNCTION RANDOM (VAR SEED : INTEGER) : INTEGER;
BEGIN (* RANDOM *)
    RANDOM := TRUNC ((SEED / 65536 - 0.1) * RANGE);
    SEED := (25173 * SEED + 13849) MOD 65536
END (* RANDOM *);

PROCEDURE CHECK_TERMINATE;
BEGIN (* CHECK_TERMINATE *)
    IF SWITCH_PER_CYCLE = 0 THEN
        BEGIN
            INCR (CONTINUOUS_NO_SWITCH);
            IF CONTINUOUS_NO_SWITCH >= LIMIT_NO_SWITCH THEN
                TERMINATE := TRUE
            END
        ELSE
            BEGIN
                SWITCH_PER_CYCLE := 0;
                CONTINUOUS_NO_SWITCH := 0
            END
        END
    END (* CHECK_TERMINATE *);

PROCEDURE INITIALIZE;
BEGIN (* INITIALIZE *)
    FOR I := 0 TO (ROW - 1) DO
        FOR J := 0 TO (COLUMN - 1) DO
            WITH STORE (. I, J .) DO
                BEGIN
                    IF DATA_ARRANGE = RRANDOM THEN
                        ITEM := RANDOM (SEED)
                    ELSE
                        BEGIN
                            ITEM := SEED;
                            SEED := SEED - DECR
                        END;
                    (* *****MARKING EACH LOOP***** *)
                    IF ((J = 2 * MARKER_COLUMN - 2) AND (I MOD 2 = 1)) OR ((J = 2 *
                        MARKER_COLUMN - 1) AND (I MOD 2 = 0)) THEN
                        MARKER := TRUE
                    ELSE MARKER := FALSE
                END;
            FOR K := 1 TO N_COMP DO
                WITH COMPARATOR (. K .) DO
                    BEGIN
                        TEMPORARY := 0;
                        AI := INIT_ROW;
                        AJ := INIT_COLUMN;
                        BI := AI;
                        BJ := AJ + 1;
                        CI := AI + 1;
                        CJ := AJ;
                        DI := AI + 1;
                        DJ := AJ + 1
                    END
                END
            END
        END
    END
END

```

```

    END;
    TOTAL_CYCLE := 0;
    SWITCH_PER_CYCLE := 0;
    CONTINUOUS_NO_SWITCH := 0;
    TERMINATE := FALSE
    END (* INITIALIZE *);

PROCEDURE VERTICAL_COMP;
BEGIN (* VERTICAL_COMP *)
    FOR K := 1 TO N_COMP DO
        WITH COMPATOR (. K .) DO
            BEGIN
                IF (STORE (. AI, AJ .).ITEM > STORE (. CI, CJ .).ITEM) THEN
                    BEGIN
                        TEMPORARY := STORE (. AI, AJ .).ITEM;
                        STORE (. AI, AJ .).ITEM := STORE (. CI, CJ .).ITEM;
                        STORE (. CI, CJ .).ITEM := TEMPORARY;
                        INCR (SWITCH_PER_CYCLE)
                    END;
                IF (STORE (. BI, BJ .).ITEM > STORE (. DI, DJ .).ITEM) THEN
                    BEGIN
                        TEMPORARY := STORE (. BI, BJ .).ITEM;
                        STORE (. BI, BJ .).ITEM := STORE (. DI, DJ .).ITEM;
                        STORE (. DI, DJ .).ITEM := TEMPORARY;
                        INCR (SWITCH_PER_CYCLE)
                    END
                END
            END
        END (* VERTICAL_COMP *);

PROCEDURE DIAGONAL_COMP;
BEGIN (* DIAGONAL_COMP *)
    FOR K := 1 TO N_COMP DO
        WITH COMPATOR (. K .) DO
            BEGIN
                IF (STORE (. AI, AJ .).ITEM > STORE (. DI, DJ .).ITEM) THEN
                    BEGIN
                        TEMPORARY := STORE (. AI, AJ .).ITEM;
                        STORE (. AI, AJ .).ITEM := STORE (. DI, DJ .).ITEM;
                        STORE (. DI, DJ .).ITEM := TEMPORARY;
                        INCR (SWITCH_PER_CYCLE)
                    END;
                IF (STORE (. BI, BJ .).ITEM > STORE (. CI, CJ .).ITEM) THEN
                    BEGIN
                        TEMPORARY := STORE (. BI, BJ .).ITEM;
                        STORE (. BI, BJ .).ITEM := STORE (. CI, CJ .).ITEM;
                        STORE (. CI, CJ .).ITEM := TEMPORARY;
                        INCR (SWITCH_PER_CYCLE)
                    END
                END
            END
        END
    END (* DIAGONAL_COMP *);
    (* DIAGONAL_COMP *)

PROCEDURE HORIZONTAL_COMP;
BEGIN (* HORIZONTAL_COMP *)
    FOR K := 1 TO N_COMP DO
        WITH COMPATOR (. K .) DO
            BEGIN
                TEMPA := STORE (. AI, AJ .).ITEM;
                TEMPB := STORE (. BI, BJ .).ITEM;
                TEMPC := STORE (. CI, CJ .).ITEM;

```



```

TEMPD := STORE (. DI, DJ .).ITEM;
MARKERA := STORE (. AI, AJ .).MARKER;
MARKERB := STORE (. BI, BJ .).MARKER;
MARKERC := STORE (. CI, CJ .).MARKER;
MARKERD := STORE (. DI, DJ .).MARKER;
TEMP_COL := TRUNC (INIT_COLUMN / 2);
IF ((NOT MARKERA) AND (TEMPB > TEMPA)) OR ((MARKERA) AND (TEMPB <
    TEMPA)) THEN
    BEGIN
        TEMPORARY := STORE (. BI, BJ .).ITEM;
        STORE (. BI, BJ .).ITEM := STORE (. AI, AJ .).ITEM;
        STORE (. AI, AJ .).ITEM := TEMPORARY;
        INCR (SWITCH_PER_CYCLE)
    END;
IF ((NOT MARKERC) AND (TEMPD > TEMPC)) OR ((MARKERC) AND (TEMPD <
    TEMPC)) THEN
    BEGIN
        TEMPORARY := STORE (. DI, DJ .).ITEM;
        STORE (. DI, DJ .).ITEM := STORE (. CI, CJ .).ITEM;
        STORE (. CI, CJ .).ITEM := TEMPORARY;
        INCR (SWITCH_PER_CYCLE)
    END
END
END (* HORIZONTAL_COMP *);

PROCEDURE DISPLAY;
BEGIN (* DISPLAY *)
    WRITELN;
    WRITELN;
    WRITELN ('NUMBER OF SWITCHING = ', SWITCH_PER_CYCLE : 5);
    WRITELN ('AT CYCLE TIME = ', TOTAL_CYCLE : 5);
    FOR I := 0 TO (ROW - 1) DO
        BEGIN
            IF (I MOD 2 = 0) THEN
                BEGIN
                    WRITELN;
                    WRITELN
                END;
            FOR J := 0 TO (COLUMN - 1) DO
                WITH STORE (. I, J .) DO
                    BEGIN
                        IF (J MOD 2 = 0) THEN WRITE (' ' : 1);
                        WRITE (ITEM : 4)
                    END;
                END;
            WRITELN
        END
    END (* DISPLAY *);

PROCEDURE TRU_DISPLAY;
BEGIN (* TRU_DISPLAY *)
    WITH COMPARATOR (. 1 .) DO
        BEGIN
            EVEN_START := AJ;
            ODD_START := CJ
        END;
    WRITELN;
    WRITELN;
    WRITELN ('AT CYCLE TIME=', TOTAL_CYCLE : 5);
    FOR I := 0 TO (ROW - 1) DO
        BEGIN

```

```

IF (I MOD 2 = 0) THEN
  BEGIN
    FOR J := EVEN_START TO EVEN_START + COLUMN - 1 DO
      BEGIN
        JJ := J MOD COLUMN;
        IF STORE (. I, JJ .).MARKER THEN
          WRITE (' *', STORE (. I, JJ .).ITEM : 2)
        ELSE WRITE (' ', STORE (. I, JJ .).ITEM : 2)
        END;
        WRITELN
      END
    ELSE
      BEGIN
        FOR J := ODD_START TO ODD_START + COLUMN - 1 DO
          BEGIN
            JJ := J MOD COLUMN;
            IF STORE (. I, JJ .).MARKER THEN
              WRITE (' *', STORE (. I, JJ .).ITEM : 2)
            ELSE WRITE (' ', STORE (. I, JJ .).ITEM : 2)
            END;
            WRITELN
          END
        END
      END (* TRU_DISPLAY *);

PROCEDURE SHIFT;
  BEGIN (* SHIFT *)
    FOR K := 1 TO N_COMP DO
      WITH COMPATOR (. K .) DO
        BEGIN
          IF (AI MOD 2 = 1) THEN AJ := (AJ + 1) MOD COLUMN
          ELSE AJ := (AJ + COLUMN - 1) MOD COLUMN;
          IF (BI MOD 2 = 1) THEN BJ := (BJ + 1) MOD COLUMN
          ELSE BJ := (BJ + COLUMN - 1) MOD COLUMN;
          IF (CI MOD 2 = 1) THEN CJ := (CJ + 1) MOD COLUMN
          ELSE CJ := (CJ + COLUMN - 1) MOD COLUMN;
          IF (DI MOD 2 = 1) THEN DJ := (DJ + 1) MOD COLUMN
          ELSE DJ := (DJ + COLUMN - 1) MOD COLUMN
        END
      END (* SHIFT *);

BEGIN (* PARA_SORT *)
  H_COMP := 6;
  V_COMP := 3;
  SEED := INIT_SEED;
  DATA_ARRANGE := RRANDOM;
  DECR := 1;
  WHILE (H_COMP <> -999) DO
    BEGIN
      WRITELN ('H_COMP/V_COMP/SEED/RAND/DECR=', H_COMP : 5, V_COMP : 5, SEED
        : 5, DATA_ARRANGE : 5, DECR : 5);
      WRITELN ('ENTER NEW VALUES/ -999 FOR TERMINATION');
      READLN (H_COMP, V_COMP, SEED, DATA_ARRANGE, DECR);
      IF (H_COMP <> -999) THEN
        BEGIN
          SETUP_NETWORK;
          INITIALIZE;
          TRU_DISPLAY;
          WHILE (NOT TERMINATE) DO
            BEGIN

```

```
        INCR (TOTAL_CYCLE);
        VERTICAL_COMP;
        HORIZONTAL_COMP;
        DIAGONAL_COMP;
        CHECK_TERMINATE;
        SHIFT
    END;
    TRU_DISPLAY;
    WRITELN;
    WRITELN ('NUMBER OF HORIZONTAL COMPARATORS =', H_COMP : 5);
    WRITELN ('NUMBER OF VERTICAL COMPARATORS =', V_COMP : 5);
    WRITELN ('NUMBER OF ITEMS SORTED=', ROW * COLUMN : 5);
    WRITELN ('NUMBER OF DOUBLE_COMPARISON/SHIFT CYCLES =', TOTAL_CYCLE,
        - LIMIT_NO_SWITCH : 5)
    END
END
END (* PARA_SORT *) .
```

Appendix C

```

PROGRAM LSSN;
(*DEADLOCK FREE
*USE CENTRAL BUFFERS FOR SIMULATION,ELSE STACK OVERFLOW
*CLASS_O,_1,_2 BUFFERS ARE PRIORITIZED:
    CLASS-2 = HIGHEST
    CLASS-1 = MIDDLE
    CLASS-O = LOWEST
*MAY 1983 *)
CONST
    N_SW=32;
    N_TR=64;
    N_RR=64;
    TR_INTL=10;
    T_SIMULAT=1000000;
    T_TRANSFER=2;
    T_DECIDE=3;
    T_SWITCH=2;
    FO_SIZE=7;
    F1_SIZE=7;
    F2_SIZE=2;
    FO12_SIZE=FO_SIZE+F1_SIZE+F2_SIZE;
    F_TOTAL=2*N_SW*FO12_SIZE;

TYPE
    BUFFER_RECORD=RECORD
        B_EMPTY:BOOLEAN;
        B_TR_TIME:INTEGER;
        B_DEST:INTEGER;
        B_FEEDBACK_COUNT:INTEGER;
    END;
    FIFO_RECORD=RECORD
        F_START,F_STOP:INTEGER;
        F_TOP,F_BOTTOM:INTEGER;
        F_EMPTY,F_FULL:BOOLEAN;
        F_COUNT:INTEGER;
    END;
    INPORT_RECORD=RECORD
        I_TIMER:INTEGER;
        I_EMPTY:BOOLEAN;
        I_TR_TIME:INTEGER;
        I_DEST:INTEGER;
        I_FEEDBACK_COUNT:INTEGER;
    END;

    OUTPORT_RECORD=RECORD
        O_TIMER:INTEGER;
        O_EMPTY,O_MATCHED:BOOLEAN;
        O_TR_TIME:INTEGER;
        O_DEST:INTEGER;
        O_FEEDBACK_COUNT:INTEGER;
        O_RR,O_NEXT_SW,O_NEXT_PT:INTEGER;
    END;
    SWITCH_RECORD=RECORD
        INPORT:ARRAY(.O..1.) OF INPORT_RECORD;
        FIFO:ARRAY(.O..1,O..2.) OF FIFO_RECORD;
        OUTPORT:ARRAY(.O..1.) OF OUTPORT_RECORD;
    END;
    TR_RECORD=RECORD
        T_EMPTY,T_BLOCKED:BOOLEAN;

```

```

T_DEST:INTEGER;
T_NEXT_SW,T_NEXT_PT:INTEGER;
T_TIMER:INTEGER;(*TRANSMIT WHEN TIMER REACHES CLOCK*)
END;

```

```

VAR
SWITCH:ARRAY(. 1..N_SW .) OF SWITCH_RECORD;
TR:ARRAY(. 1..N_TR .) OF TR_RECORD;
BUFFER:ARRAY(. 1..F_TOTAL .) OF BUFFER_RECORD;
FMAX:ARRAY(. 0..2 .) OF INTEGER;(* MAX USAGE OF FIFO*)
CLOCK:INTEGER;
SEED:INTEGER;
R_PACKET,T_PACKET:INTEGER;
TOTAL_DELAY:INTEGER;
MAX_DELAY:INTEGER;
TR_DELAY:INTEGER;(*BLOCKAGE AT ENTRANCE*)

```

```

FUNCTION RANDOM (VAR SEED:INTEGER) :REAL;
BEGIN
RANDOM:=SEED/65535;
SEED := (25173*SEED+13849)MOD 65536;
END;

```

```

PROCEDURE INITIALIZE;
VAR
TI,SI,IPI,OPI,CI,PI,BI:INTEGER;
TEMPI:INTEGER;
BEGIN
WRITELN('LAST STAGE DOES NOT MATCH LOOP NUMBER.JUST INCR FB#');
WRITELN('NUMBER OF SWITCHES=',N_SW:5);
WRITELN('NUMBER OF TR=',N_TR:5);
WRITELN('FIFO SIZE OF CL=0,1,2,TOTAL PER SWITCH=',FO_SIZE:5,
F1_SIZE:5,F2_SIZE:5,FO12_SIZE:5);
WRITELN('REQUEST RATE=',1/TR_INTL:10:5);
WRITELN('TR INTL= ',TR_INTL:7);
WRITELN('SIMULATION TIME=',T_SIMULAT:5);
FOR SI:=1 TO N_SW DO WITH SWITCH[SI] DO
BEGIN (*S*)
FOR IPI:= 0 TO 1 DO WITH INPORT[IPI] DO
BEGIN (*IP*)
I_TIMER:=0;
I_EMPTY:=TRUE;
I_DEST:=-1;
I_TR_TIME:=0;
I_FEEDBACK_COUNT:=0;
END;(*IP*)
FOR PI:=0 TO 1 DO
BEGIN
FOR CI:= 0 TO 2 DO WITH FIFO[PI,CI] DO
BEGIN (*BP,CL*)
F_EMPTY:=TRUE;
F_FULL:=FALSE;
F_COUNT:=0;
END;(*BP,CL*)
(**DETERMINE MEMORY LOCATIONS FOR EACH SWITCH'S BUFFER**)
TEMPI:=FO12_SIZE*(2*SI-2+PI)+1;
FIFO[PI,0].F_START:=TEMPI;
FIFO[PI,0].F_TOP:=TEMPI;

```

```

FIFO[PI,0].F_BOTTOM:=TEMPI;
FIFO[PI,0].F_STOP:=TEMPI+FO_SIZE-1;
FIFO[PI,1].F_START:=TEMPI+FO_SIZE;
FIFO[PI,1].F_TOP:=TEMPI+FO_SIZE;
FIFO[PI,1].F_BOTTOM:=TEMPI+FO_SIZE;
FIFO[PI,1].F_STOP:=TEMPI+FO_SIZE+F1_SIZE-1;
FIFO[PI,2].F_START:=FIFO[PI,1].F_STOP+1;
FIFO[PI,2].F_TOP:=FIFO[PI,1].F_STOP+1;
FIFO[PI,2].F_BOTTOM:=FIFO[PI,1].F_STOP+1;
FIFO[PI,2].F_STOP:=FIFO[PI,2].F_START+F2_SIZE-1;
END;
FOR OPI:=0 TO 1 DO WITH OUTPORT[OPI] DO
  BEGIN(*OP*)
    O_TIMER:=0;
    O_EMPTY:=TRUE;
    O_TR_TIME:=0;
    O_MATCHED:=FALSE;
    O_DEST:=-1;
    O_FEEDBACK_COUNT:=0;
    READLN(O_RR,O_NEXT_SW,O_NEXT_PT);
    END;(*OP*)
END;(*S*)
FOR TI:=1 TO N_TR DO WITH TR[TI] DO
  BEGIN(*T*)
    T_EMPTY:=TRUE;
    T_BLOCKED:=FALSE;
    T_DEST:=-1;
    T_NEXT_SW:= TRUNC((TI+1)/2);
    T_NEXT_PT:= (TI+1)MOD 2;
  REPEAT
    T_TIMER:=TRUNC(RANDOM(SEED)*2*TR_INTL);
  UNTIL T_TIMER>0 AND T_TIMER<2*TR_INTL;
  END;(*T*)
  FOR BI:=1 TO F_TOTAL DO
    WITH BUFFER[BI] DO
      BEGIN
        B_EMPTY:=TRUE;
        B_TR_TIME:=0;
        B_DEST:=-1;
        B_FEEDBACK_COUNT:=0;
      END;
    END;
  CLOCK:=0;
  TOTAL_DELAY:=0;
  TR_DELAY:=0;
  R_PACKET:=0;
  T_PACKET:=0;
  MAX_DELAY:=0;
  FMAX[0]:=0;
  FMAX[1]:=0;
  FMAX[2]:=0;
  (**MAX USAGE OF EACH CLASS OF BUFFERS*)
  END;(*INIT*)

```

```

PROCEDURE TR_GET_DEST;
VAR
  T: INTEGER;
BEGIN
  FOR T:= 1 TO N_TR DO WITH TR[T] DO
    BEGIN(*T*)

```

```

IF T_TIMER <= CLOCK AND T_EMPTY AND NOT T_BLOCKED THEN
  BEGIN(*TIMER*)
    REPEAT
      T_DEST := TRUNC(RANDOM(SEED)*65);
    UNTIL ( T_DEST IN (. 1..64 .) AND T_DEST <> T )
    T_EMPTY:=FALSE;
  END;(*TIMER*)
END;
END;(*TR_GET_DEST*)

```

```

FUNCTION ROUTE(SW, DT:INTEGER) : INTEGER;
VAR
  SR,DR:INTEGER;
BEGIN
  SR:=SW;
  DR:=DT;
  ROUTE:=1;(*INITIAL SETTING*)
  IF SR<=8 AND ( (DR-1) MOD 2 =0) THEN
    ROUTE:=0
  ELSE IF SR<=16 AND SR>8 AND ( TRUNC((DR-1)/2) MOD 2 =0) THEN
    ROUTE:=0
  ELSE IF SR<=24 AND SR>16 AND ( TRUNC((DR-1)/4) MOD 2=0) THEN
    ROUTE:=0
  ELSE IF SR<=32 AND SR>24 AND ( TRUNC((DR-1)/8)MOD 2=0) THEN
    ROUTE:=0
  END;
END;

```

```

PROCEDURE TR_TO_INPORT;
VAR
  TT:INTEGER;
  PACKET_COUNT:INTEGER;
BEGIN
  FOR TT:=1 TO N_TR DO WITH TR[TT] DO
    BEGIN(*T*)
      IF NOT T_EMPTY AND T_TIMER<=CLOCK THEN
        WITH SWITCH[T_NEXT_SW],INPORT[T_NEXT_PT] DO
          BEGIN(*READY TO TRANSMIT*)
            PACKET_COUNT:=FIFO[T_NEXT_PT,0].F_COUNT;
            IF ( I_EMPTY) AND I_TIMER<=CLOCK AND
              FIFO[T_NEXT_PT,0].F_COUNT<FO_SIZE AND
              FIFO[T_NEXT_PT,1].F_COUNT<F1_SIZE AND
              FIFO[T_NEXT_PT,2].F_COUNT<F2_SIZE THEN
              PACKET_COUNT<(FO_SIZE-1)*(TRUNC(T_NEXT_SW/8.4+1)*8/N_SW) THEN
                ( FIFO[0,0].F_COUNT < ( TRUNC(T_NEXT_SW/8.4)+1)) AND
                (FIFO[1,0].F_COUNT< ( TRUNC(T_NEXT_SW/8.4)+1)) THEN
                BEGIN(*GRANTED TO TRANSMIT*)
                  I_TIMER:=CLOCK+T_TRANSFER;
                  I_EMPTY:=FALSE;
                  I_DEST:=T_DEST;
                  I_FEEDBACK_COUNT:=0;
                  I_TR_TIME:=CLOCK;
                  T_EMPTY:=TRUE;
                  T_TIMER:=CLOCK+ TRUNC(TR_INTL*2*(RANDOM(SEED)));
                  T_DEST:=-1;
                  T_BLOCKED:=FALSE;
                  INCR(T_PACKET);
                END
              ELSE

```

```

        BEGIN
            T_BLOCKED:=TRUE;
            INCR(TR_DELAY);(*INCREMENT TOTAL TR_DELAY*)
        END;
    END;(*READY TO TRANSMIT*)
END;(*T*)
END;(*TR_TO_IMPORT*)

PROCEDURE TRANSFER( SS,BB,CC,PPRT,OOPT,CC_NEXT:INTEGER);
VAR
    ST,BT,CT,PTRT,OPT,CT_NEXT:INTEGER;
BEGIN(*GRANT CLASS-CL BUFFER*)
    ST:=SS;
    BT:=BB;
    CT:=CC;
    PTRT:=PPRT;
    OOPT:=OOPT;
    CT_NEXT:=CC_NEXT;
    WITH SWITCH[ST],FIFO[BT,CT],OUTPORT[OPT] DO
    BEGIN
        WITH BUFFER[PTRT] DO
        BEGIN
            DECR(F_COUNT);
            O_TIMER:=CLOCK+T_SWITCH+T_DECIDE;
            O_EMPTY:=FALSE;
            O_DEST:=B_DEST;
            IF O_DEST NOT IN (. 1..64 .) THEN
                WRITELN('????? WRONG DEST, LINE 257?????',O_DEST:5);
            IF O_DEST=O_RR THEN O_MATCHED:=TRUE
                ELSE O_MATCHED:=FALSE;
            O_FEEDBACK_COUNT:=CT_NEXT;
            O_TR_TIME:=B_TR_TIME;
            B_EMPTY:=TRUE;
            B_DEST:=-1;
            B_FEEDBACK_COUNT:=0;
            B_TR_TIME:=CLOCK;
            F_FULL:=FALSE;
            F_BOTTOM:=PTRT;
            IF F_TOP=F_BOTTOM THEN
                F_EMPTY:=TRUE
                ELSE F_EMPTY:=FALSE;
            END
        END (*CLASS_CL*)
    END;
END;

PROCEDURE OUTPORT_TO_IMPORT;
VAR
    S,OP:INTEGER;
BEGIN
    FOR S:=1 TO N_SW DO WITH SWITCH[S] DO
        FOR OP:=0 TO 1 DO
            WITH OUTPORT[OP],SWITCH[O_NEXT_SW].IMPORT[O_NEXT_PT] DO
                BEGIN(*S,OP*)
                    IF NOT O_EMPTY AND NOT O_MATCHED AND O_TIMER<=CLOCK THEN
                        BEGIN(*READY TO TRANSMIT*)
                            IF ( I_EMPTY) THEN
                                BEGIN(*GRANTED TO TRANSMIT*)
                                    I_TIMER:=CLOCK+T_TRANSFER;
                                END
                            END
                        END
                    END
                END
            END
        END
    END
END

```



```

        I_EMPTY:=FALSE;
        I_DEST:=O_DEST;
        I_TR_TIME:=O_TR_TIME;
        I_FEEDBACK_COUNT:=O_FEEDBACK_COUNT;
        O_TIMER:=CLOCK+T_TRANSFER;
        O_DEST:=-1;
        O_EMPTY:=TRUE;
        O_FEEDBACK_COUNT:=0;
        O_TR_TIME:=CLOCK;
    END
ELSE
    BEGIN(*BLOCKED*)
    (*****OUTPUT IS BLOCKED????????*****
    WRITE('?????OUTPUT BLOCKED????');
    WRITE('AT T,SW,NEXT SW,FB_COUNT');
    WRITELN( CLOCK:3,S:3,O_NEXT_SW:3,O_FEEDBACK_COUNT:4);
    O_EMPTY:=FALSE;
    END;
    END(*READY*)
ELSE
    IF O_MATCHED THEN(*REMOVE PACKETS*)
    BEGIN
        O_TIMER:=CLOCK;
        O_EMPTY:=TRUE;
        O_MATCHED:=FALSE;
        INCR(R_PACKET);
        WRITELN('RECEIVED AT TIME,DEST,RR,DELAY=',CLOCK:5,O_DEST:5,
        O_RR:5,CLOCK - O_TR_TIME:5);
        IF O_TR_TIME=O THEN WRITELN('??????? LINE 343????');
        TOTAL_DELAY:=TOTAL_DELAY+(CLOCK - O_TR_TIME);
        IF (CLOCK-O_TR_TIME)> MAX_DELAY THEN
            MAX_DELAY:=(CLOCK - O_TR_TIME);
        O_DEST:=-1;
        O_FEEDBACK_COUNT:=0;
        O_TR_TIME:=CLOCK;
    END;(*REMOVE PACKETS*)
    END;(*S,OP*)
END;(*OUTPORT_INPORT*)

PROCEDURE INPORT_TO_POOL;
VAR
SS,IP:INTEGER;
BEGIN
    FOR SS:=1 TO N_SW DO WITH SWITCH[SS] DO
        FOR IP:=0 TO 1 DO WITH INPORT[IP] DO
            BEGIN (*S,IP*)
                IF NOT I_EMPTY AND I_TIMER<=CLOCK THEN
                    BEGIN (*READY TO STORE PACKETS INTO BUFFER POOL*)
                        IF FIFO[IP,I_FEEDBACK_COUNT].F_FULL THEN
                            WRITELN('!!!!!!F FULL!!!!I S,P,CL=',SS:2,IP:2,I_FEEDBACK_COUNT:3);
                            WITH FIFO[IP,I_FEEDBACK_COUNT] DO
                                IF NOT F_FULL THEN
                                    BEGIN
                                        F_TOP:=(F_TOP+1);
                                        IF F_TOP>F_STOP THEN F_TOP:=F_START;
                                        WITH BUFFER[F_TOP] DO
                                            BEGIN
                                                INCR(F_COUNT);
                                                IF F_COUNT>FMAX[I_FEEDBACK_COUNT] THEN

```

```

        FMAX[I_FEEDBACK_COUNT]:=F_COUNT;
        B_EMPTY:=FALSE;
        B_DEST:=I_DEST;
        B_TR_TIME:=I_TR_TIME;
        B_FEEDBACK_COUNT:=I_FEEDBACK_COUNT;
        I_TIMER:=CLOCK+T_SWITCH+T_DECIDE;
        I_EMPTY:=TRUE;
        I_DEST:=-1;
        I_FEEDBACK_COUNT:=0;
        I_TR_TIME:=CLOCK;
        F_EMPTY:=FALSE;
        IF F_TOP=F_BOTTOM THEN F_FULL:=TRUE
            ELSE F_FULL:=FALSE;
    END
END
END
END (*S,IP*)
END;(*INPORT_BUFFER POOL*)

PROCEDURE POOL_TO_OUTPORT;
VAR
PTRP:INTEGER;(*POINTER OF STRUCTURED BUFFERS*)
TERMINATE:BOOLEAN;
SP,OPP,PP,CLP,CLP_NEXT:INTEGER;
CHECK_DEST:INTEGER;
CHECK_BIT:INTEGER;
OK_TRANSFER:BOOLEAN;
NOW_SCHEDULED:INTEGER;
BEGIN
FOR SP:=1 TO N_SW DO WITH SWITCH[SP] DO
FOR OPP:=0 TO 1 DO WITH OUTPUT[OPP] DO
    BEGIN(*S,OP*)
        IF O_EMPTY AND O_TIMER<=CLOCK THEN
            BEGIN(*READY TO ACCEPT PACKETS FROM CLASS_O,_1,_2 BUFFERS*)
                NOW_SCHEDULED:=0;
                TERMINATE:=FALSE;
                WHILE (NOW_SCHEDULED<6) AND (NOT TERMINATE) DO
                    BEGIN
                        CASE NOW_SCHEDULED OF
                            0: BEGIN PP:=0; CLP:=2; END;
                            1: BEGIN PP:=1; CLP:=2; END;
                            2: BEGIN PP:=0; CLP:=1; END;
                            3: BEGIN PP:=1; CLP:=1; END;
                            4: BEGIN PP:=0; CLP:=0; END;
                            5: BEGIN PP:=1; CLP:=0; END;
                            <>: BEGIN WRITELN('ERROR IN POOL TO OUTPORT!!!!!!!!');END;
                        END;
                        PTRP:=(FIFO[PP,CLP].F_BOTTOM+1);
                        IF PTRP>FIFO[PP,CLP].F_STOP THEN PTRP:=FIFO[PP,CLP].F_START;
                        (*REMOVE PACKET FROM BOTTOM OF BUFFER*)
                        CHECK_DEST:=BUFFER[PTRP].B_DEST;
                        CHECK_BIT:=ROUTE(SP,CHECK_DEST);
                        (*DETERMINE SWITCH BIT OF PACKET*)
                        (*IF FEEDBACK PACKET, THEN GO TO NEXT CLASS OF BUFFER*)
                        IF O_NEXT_SW IN ( . 1..8 . ) THEN
                            BEGIN
                                IF CLP<2 AND (((CHECK_DEST-1)MOD 16)=((O_RR-1) MOD 16))THEN
                                    CLP_NEXT:=2
                                ELSE
                                    IF CLP<2 THEN

```

```

        CLP_NEXT:=CLP+1
      ELSE CLP_NEXT:=CLP
    END
  ELSE
    CLP_NEXT:=CLP;
    IF O_NEXT_SW IN (. 1..8 .) AND CLP<2 THEN
      CLP_NEXT:=CLP+1;
    ELSE CLP_NEXT:=CLP;
    WITH SWITCH[O_NEXT_SW].FIFO[O_NEXT_PT,CLP_NEXT] DO
      IF (NOT FIFO[PP,CLP].F_EMPTY) AND
        (F_COUNT< (F_STOP-F_START)) AND
        (CHECK_BIT=OPP) THEN
        BEGIN
          OK_TRANSFER:=TRUE;
          TERMINATE:=TRUE;
          O_TIMER:=CLOCK+T_SWITCH;
          O_EMPTY:=FALSE;
        END
      ELSE
        WITH BUFFER[PTRP] DO
          BEGIN
            OK_TRANSFER:=FALSE;
            NOW_SCHEDULED:=(NOW_SCHEDULED+1);
            TERMINATE:=FALSE;
          END;
        END;
      END;
    END;(*WHILE*)
    IF OK_TRANSFER THEN TRANSFER(SP,PP,CLP,PTRP,OPP,CLP_NEXT);
  END;(*EMPTY*)
END;(*S,OP*)
END;(*POOL_TO_OUTPORT*)

PROCEDURE GROSS_DISPLAY;
BEGIN
  WRITELN('AT TIME=',CLOCK:5);
  WRITELN('  T_PACKET=',T_PACKET:7);
  WRITELN('  R_PACKET=',R_PACKET:7);
  IF R_PACKET > 1 THEN
    WRITELN('  AVERAGE DELAY=',TOTAL_DELAY/R_PACKET:10:5);
    WRITELN('  MAX DELAY= ',MAX_DELAY:5);
    WRITELN('  AVERAGE TR_DELAY= ', TR_DELAY/T_PACKET:10:5);
    WRITELN('  AVERAGE THROUGHPUT=', R_PACKET/T_SIMULAT:10:5);
    WRITELN('  UNDELIVERED PACKETS=', T_PACKET-R_PACKET:5);
    WRITELN('  MAX FIFO USAGE=',FMAX[0]:3,FMAX[1]:3,FMAX[2]:3);
    WRITELN('  TOTAL FIFO USAGE=',FMAX[0]+FMAX[1]+FMAX[2]:5);
  END;

PROCEDURE DETAIL_DISPLAY;
VAR
  SW,C,P,B:INTEGER;
BEGIN
  WRITELN('BUFFER DISPLAY OF SWITCH ARRAY');
  WRITE(' S PT CL B SRC DST SBIT STEP FB GEN_T TR_T BF_T DE_T DE_?');
  WRITELN(' F TOP F BOTTOM MAX_USED');
  FOR SW:=1 TO N_SW DO WITH SWITCH[SW] DO
    BEGIN
      FOR P:=0 TO 1 DO
        BEGIN
          FOR C:=0 TO 2 DO WITH BUFFER_POOL[P,C] DO
            BEGIN

```

```

        IF F_FULL THEN WRITELN('BUFFER FULL:S,P,C ',SW:3,P:3,C:3);
        IF NOT F_EMPTY THEN
            BEGIN
                FOR B:=0 TO FIFO_SIZE-1 DO WITH BUFFER[B] DO
                    IF NOT B_EMPTY THEN
                        BEGIN
                            WRITE(' ',SW:2,' ',P:2,' ',C:2,' ',B:2,' ',B_DEST:5);
                            WRITE(B_FEEDBACK_COUNT:3,' ');
                            WRITELN(B_GENERATE_TIME:5,B_TR_TIME:5,' ');
                            F_TOP:=6,F_BOTTOM:=6);
                        END;
                    END;
                END;
            END;
        WRITELN;
    END;
    END;
    END;
    END;

PROCEDURE DEBUG;
VAR
    B:INTEGER;
BEGIN
    WRITELN('DISPLAY OF BUFFERS');
    FOR B:=1 TO F_TOTAL DO WITH BUFFER[B] DO
        BEGIN
            IF NOT B_EMPTY THEN
                WRITELN(B:7,B_TR_TIME:5,B_DEST:5,B_FEEDBACK_COUNT:5);
            END;
        END;
    END;

BEGIN(*LSSN*)
    SEED:=8476;
    INITIALIZE;
    FOR CLOCK:=1 TO T_SIMULAT DO
        BEGIN
            TR_GET_DEST;
            INPORT_TO_POOL;
            POOL_TO_OUTPORT;
            OUTPORT_TO_INPORT;
            TR_TO_INPORT;
            IF CLOCK> T_SIMULAT -1 THEN GROSS_DISPLAY;
        END;
    END.
    (*INPUT FILE WHICH CONTAINS THE INTERCONNECTION PATTERN OF THE LSSN*)

```

References:

1. T. Moto-oka (editor) 1982. Fifth Generation Computer Systems. North-Holland Publishing Company.
2. IEEE Spectrum. Tomorrow's Computers. Vol.20, No.11, Nov. 1983.
3. H.T. Kung & C.E. Leiserson, "Systolic Arrays for VLSI," Dept. of Computer Sc., Carnegie-Mellon Univ., Tech. Rept. CS-79-103, Apr. 1983.
4. Computer, Vol. 15, No. 2, Feb. 1982. Special issue on data-flow computers.
5. P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architectures," ACM Computing Surveys, Vol. 14, No. 1, March 1982.
6. H.S. Stone, "Parallel Computers," in Introduction to Computer Architectures, edited by H.S. Stone et al, 1975, Science Research Associates, Inc.
7. W.R. Cyre & G.J. Lipovski, "On generating Multipliers for a Cellular Fast Fourier Transform Processor," IEEE Trans. on Computers, C-21, pp83-87, 1972.
8. D. P. Misunas, "A Computer Architecture for Data Flow Computation," MIT/LCS/TM-100, Cambridge, MA., 1975.
9. H.S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, C-20, pp153-161, 1971.
10. D.E. Muller & E.P. Preparata, "Bounds to Complexities of Networks for Sorting and Switching," J. Ass. Comput. Mach., Vol.22, pp195-201, Apr. 1975.
11. D.E. Knuth, The Art of Computer Programming, Vol.3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.
12. T. Lang & H.S. Stone, "A Shuffle-exchange network with simplified Control," IEEE Trans. on Computers, Vol.C-25, pp.55-65, Jan. 76.
13. K.E. Batcher, "Sorting Networks and their Applications," Proc. AFIPS 1968, Spring Joint Comput. Conf., pp307-314, Apr. 1968.
14. D. Nassimi & S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Trans. on Comput., V10-C28, No.1, pp2-7, Jan.1979.
15. C.D. Thompson & H.K. Kung, "Sorting on a Mesh-Connected Parallel Computer," Comm. of the ACM, Vol.20, No.4, pp263-271, Apr. 1977.
16. H.T. Kung, "Let's Design Algorithms for VLSI Systems," Dept. of Computer Sc., Carnegie-Mellon Univ., Tech. Rep., Jan. 1979.
17. F.S. Wong & M.R. Ito, "A Systolic Sorter and its Simulation Results," Dept. of E.E., The Univ. of British Columbia, Tech. Rep., Oct. 1982.

18. C.D. Thompson, "A Complexity Theory for VLSI," Ph.D. Thesis, Carnegie-Mellon Univ., Dept. of Computer Sc., 1979.
19. M.J. Foster & H.T. Kung, "Design of Special-Purpose VLSI Chips: Examples and Opinions," Dept. of Computer Sc., Carnegie-Mellon Univ., Tech. Rep., Sep. 1979.
20. C. Wu & T. Feng, "On a Class of Multistage Interconnection Networks," IEEE Trans. on Computers, Vol. C-29, No. 8, Aug. 1980, pp. 694-702.
21. M.C. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers, Vol. C-26, No.5, May 1977, pp.458-473.
22. Computer, Vol.14, No. 12, Dec. 1981. Special issue on interconnection Networks.
23. F.S. Wong & M.R. Ito, "A Novel Packet Switching Network," Tech. Rept., Dept. of E.E., The Univ. of Britihs Columbia, Canada, July 1982.
24. C. Wu, T. Feng & M.C. Lin, "Star: A Local Network System for Realtime Management of Imagery Data," IEEE Trans. on Computers, Vol. C-31, No. 10, Oct. 1982, pp. 923-933.
25. D.M. Dias & J.R. Jump, "Packet Switching Interconnection Networks for Modular Systems," in Computer, Vol. 14, No. 12, Dec. 1981, pp.42-53.
26. A.R. Tripathi & J. Lipovski, "Packet Switching in Banyan Networks," Proceedings of the 6th Annual Symposium on Computer Architectures, 1979, pp.160-167.
27. K.E. Batcher, "Sorting Networks and their Applications," Proceedings of AFIPS 1968, Spring Joint Computer Conf., pp.307-314, 1968.
28. F. S. Wong and M. R. Ito, "A Large-Scale Data-Flow Computer For Highly Parallel Signal Processing," Proceedings of the 1982 International Conference on Circuits and Computers, New York, Oct. 1982.
29. E. Raubold & J. Haenle, "A Method of Deadlock-free Resource Allocation and Flow Control in Packet Networks," Proceeding ICCS 1976, Toronto, Canada, Aug. 1976, pp.483.
30. G.H. Barnes & S.F. Lundstrom, "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems," in Computer, Vol. 14, No. 12, Dec. 1981, pp31-41.
31. K. S. Weng, "An Abstract Implementation for a Generalized Data Flow Language," MIT/LCS/TR-228, Cambridge, MA., 1979.
32. D. D. Gajski, D. J. Kuck and D. A. Padua, "Dependence-Driven Computation," Proceedings of the IEEE 1981 Compcon Spring, pp. 168-172.
33. P. C. Treleaven, R. P. Hopkins and P. W. Rautenback, "Combining Data Flow and Control Flow Computing," The Computer Journal,

34. J. E. Requa and J. R. McGraw, "The Piece-wise Data Flow Architecture: Architectural Concepts," IEEE Transactions on Computers, Vol. C-32, No. 5, 1983, pp. 425-437.
35. F. S. Wong and M. R. Ito, "A Loop-Structured Switching Network," Technical Rept., Dept. of E.E., The Univ. of British Columbia, 1982. (Accepted by IEEE Trans. on Computers.)
36. P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Trans. on Computers, Vol. C-26, 1971, pp. 1566-1569.
37. D. H. Lawrie and C. R. Vora, "The Prime Memory System for Array Access," IEEE Trans. on Computers, Vol. C-31, No. 5, 1982, pp. 435-442.
38. B. Hansen. The Architecture of Concurrent Pascal. Prentice-Hall, Inc. 1977.
39. W. B. Ackerman, "Data Flow Languages," Proc. of the 1979 National Computer Conference, 1979, pp. 1087-1095.
40. S. F. Lundstrom and G. H. Barnes, "A Controllable MIMD Architecture," Proc. of the 1980 International Conference on Parallel Processing, 1980, pp. 19-27.
41. D. Comte, N. Hifdi and J. C. Syre, "The Data Driven LAU Multiprocessor System: Results and Perspectives," Information Processing 80, S. H. Lavington (Ed.), North-Holland Pub. Co., 1980, pp. 175-179.
42. E. W. Disjkstra, "Co-operating Sequential Processes," in Programming Languages. F. Genuys (Ed.) Academic Press, 1968.
43. W. B. Ackerman and J. B. Dennis, "VAL -- a Value-oriented Algorithmic Language: Preliminary reference manual," MIT/LCS TR-218, Jan. 1979.
44. Reference Manual for the Ada Programming Language, Proposed Standard Document. US Department of Defense, 1980.
45. C. K. C. Leung, "Fault Tolerance in Packet Communication Computer Architectures," MIT/LCS/TR-250, 1980.
46. D. A. Adams, "A Computation Model with Data Flow Sequencing," Computer Science Dept., School of Humanities and Science, Stanford University, TR-CS17, Dec. 1968.
47. Arvind, K. P. Gostelow and W. E. Plouffe, "An Asynchronous Programming Language and Computing Machine," TR-114a, Dept. of Infor. and Comp. Sc., UC Irvine, Dec. 1978.
48. Arvind, V. Kathail and K. Pingali, "A Dataflow Architecture with Tagged Tokens," MIT/LCS/TM-174, Cambridge, Mar. 1980.
49. Arvind and R. E. Thomas, "I-Structure: An Efficient Data Type for Functional Language," MIT/LCS/TM-178, Sept. 1980.

50. J. D. Brock and L. B. Montz, "Translation and Optimization of Data Flow Programs," Proc. 1979 Intl. Conf. on Parallel Processing, Bellaire, Michigan, Aug. 1979, pp. 46-54.
51. A. L. Davis, "The Architecture of DDM1: A Recursively Structured Data Driven Machine," Univ. of Utah, Comp. Sc. Dept. TR-UUCS-77-113, 1977.
52. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Project MAC. MIT CSG Memo 102.
53. J. B. Dennis and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proc. of the ACM 1974 National Conference, pp. 402-409.
54. J. B. Dennis and K. S. Weng, "Application of Data Flow Computation to the Weather Problem," Proc. of the Symposium on High Speed Computer and Algorithm Organizations, April 1977, pp. 143-157.
55. S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions," in Computer, July 1978 issue.
56. S. P. Kartashev and S. I. Kartashev, "Supersystems for the 80's," in Computer, Nov. 1980 issue.
57. G. J. Lipovski, "On a Varistructured Array of Microprocessors," IEEE Trans. on Computers, Feb. 1977, pp. 125.
58. J. R. McGraw, "Data Flow Computing: The VAL Language," MIT/LCS TM-188, Jan. 1980.
59. L. B. Montz, "Safety and Optimization Transformation of Data Flow Programs," MIT/LCS/TR-240, Cambridge, Ma., Jan. 1980.
60. J. Rambough, "A Data Flow Multiprocessor," IEEE Trans. on Comp., Feb. 1977, pp. 138-146.
61. S. S. Reddi and E. A. Feustel, "A Restructurable Computer System," IEEE Trans. on Computers, Jan. 1978, pp. 1-20.
62. R. M. Shapiro and et al., "Representation of Algorithms as Cyclic Partial Ordering," Applied Data Research, Wakerfield, Mass., Report CA-7112-2711, Dec. 1971.
63. H. J. Siegel and et al., "A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems," National Computer Conference 1979, pp. 529-542.
64. M. R. Sleep, "Applicative Languages, Data Flow and Pure Combinatory Code," IEEE Compcon 1980, pp. 112-115.
65. P. C. Treleaven, "Exploring Program Concurrency in Computing Systems," in Computer, Jan. 1979, pp. 42-49.
66. C. G. Vick and et al., "Adaptable Architectures for Supercomputers," in Computer, Nov. 1980, pp. 17-36.
67. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labelling," National Computer Conference 1979, pp. 623-628.

68. D. P. Misunas, "Structure Processing in a Data Flow Processor,"
Proceedings of 1976 International Parallel Processing, Aug. 1976
pp. 100-105.
69. R. H. Perrott, "A Language for Array and Vector Processors,"
ACM Trans. on Programming Language and Systems, Vol. 1, No. 2,
Oct. 1979, pp. 177-195.