INFORMATION TO USERS

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

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

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

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

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

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

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

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR

7816139

MERSTEN, GERALD STUART
AN EXPERIMENTAL SIMULTANEOUS MULTIPROCESSOR ORGANIZATION.

CITY UNIVERSITY OF NEW YORK, PH.D., 1978

University Microfilms International 300 N. ZEEB ROAD, ANN ARBOR, MI 48106

© 1978

GERALD STUART MERSTEN

ALL RIGHTS RESERVED

AN EXPERIMENTAL SIMULTANEOUS MULTIPROCESSOR ORGANIZATION

BY

GERALD S. MERSTEN

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1978

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

April 24, 1978
date

Chairman of Examining Committee

April 24, 1978
date

Executive Officer

Prof. Se Jeung Oh - Mentor/Chairman

Prof. R. Mekel

Prof. D. Schilling

Prof. F. Thau

Supervisory Committee

The City University of New York

VITA

Gerald Stuart Mersten was born in Brooklyn, New York, on September 28, 1942. He received the Bachelor of Engineering (Electrical) and Master of Engineering (Electrical) degrees from The City College, The City University of New York in 1965 and 1969, respectively. In March 1973 he started his Ph.D. program at The City University of New York. In addition, Mr. Mersten was on the instructional staff of the Electrical Engineering Department of The City College of New York.

Since June 1965, he has been a digital computer engineer with the Guidance Systems Division (Navigation and Control Group), of The Bendix Corporation, in Teterboro, New Jersey. Mr. Mersten is responsible for the digital computer development group at Guidance Systems Division.

He is a member of Tau Beta Pi, Eta Kappa Nu, the Institute of Electrical and Electronics Engineers and its Computer Society.

Mr. Mersten has published three papers with S. J. Oh: Parallel Processing Utilizing Web Structures, Bit-Slice Microprocessor Emulation of an Aerospace Processor, and Achievable Error Detection Through Self-Test Software, at NAECON (National Aerospace & Electronics Conference) in Dayton Ohio, and published by the I.E.E.E. In addition, a fourth paper, "Bit-Slice Microprocessor Eumulation of an Aerospace Minicomputer" has won him "Best Paper Award, Based on Originality" at The 1977 Bendix Microprocessor Conference in Southfield, Michigan.

Abstract

AN EXPERIMENTAL SIMULTANEOUS
MULTIPROCESSOR ORGANIZATION
by
Gerald S. Mersten

Adviser: Professor Se Jeung Oh

A Simultaneous Multiprocessor Organization, abbreviated SAMSON, is proposed for surmounting the fundamental computational limitations of real time uniprocessor architectures. To maximize the degree to which the SAMSON multiprocessor exploits parallel processing, each SAMSON processor is itself capable of concurrent fetch, decode and execute operations. The proposed SAMSON architecture is a data-driven data flow processing system developed for the real time parallel processing of web structures. These webs are directed graphs for the parallel processing of computational expressions. The web structure is a generalization of the tree structure having reduced height (execution time) and minimal width (number of processors).

This work deals with the structure of the required multiprocessor interconnection mechanism required for communication with the external environment (Input/Output Subsystem) as well as the basic architecture of the SAMSON network.

In addition, the fault tolerant characteristics of both the SAMSON PE and the SAMSON system (configured as a fault tolerant system) are studied and achievable error detection through self-test software is evaluated.

ACKNOWLEDGEMENT

I wish to express my deepest gratitude to my adviser, Professor Se Jeung Oh for his guidance, inspiration, suggestions and his role as devil's advocate. I also wish to express my thanks to my Guidance Committee for their active interest and encouragement which has led this work to its successful completion.

I'd like to thank my colleagues and management at the Guidance Systems Division of The Bendix Corporation for their assistance and support throughout this period of my studies.

My thanks and appreciation to Miss Rose Russo for her diligence and excellent typing of this thesis.

My thanks to my mother for her inspiration throughout my scholastic endeavors.

I'd also like to express my special thanks to my dear wife, Arlene, without whose love, understanding and thoughtfulness this work could not have been completed. Finally, I'd like to especially thank my wonderful children, Cynthia and Beth, for being so patient, which at their tender age was not always easy.

TABLE OF CONTENTS

SECTION	TITLE	PAGE
1.0	INTRODUCTION	1
1.1	STATEMENT OF PROBLEM	2
1.2	SUMMARY OF RESULTS	4
2.0	SUMMARY OF PRIOR WORK	6
2.1	PRIOR MACHINE ARCHITECTURES	6
2.2	RELATED WORK	10
3.0	PARALLEL PROCESSING	12
3.1	INTUITIVE NOTIONS OF PARALLELISM	12
3.2	EXPLOITING PARALLELISM	15
3.3	DETECTION OF PARALLELISM	16
3.3.1	Parallelism in Sequential Programs	18
3.3.2	Theoretical Aspects of Parallelism	20
3.3.3	Conditions for Statement Independence	21
3.4	WEB STRUCTURES	23
3.5	SAMSON - A DATA FLOW MULTIPROCESSOR	36
3.6	MULTIPROCESSOR CHARACTERISTICS	44
3.7	THE INDIVIDUAL PROCESSING ELEMENTS (PEs) OF SAMSON	47
3.7.1	Introduction to the SAMSON PE	48
3.7.2	General Overview and Definitions	50
3.7.3	Emulated PE Description	56
3.7.4	Architecture of the SAMSON Microprocessor Based PE	59
3.7.5	PE Instruction Repertoire	69
3.8	SAMSON MEMORY ORGANIZATION	70
3.8.1	Local PE ROM Memory	73
3.8.2	Local PE RAM Memory	74
3.8.3	Local PE ROM Patch Organization	81

TABLE OF CONTENTS (cont'd)

SECTION	TITLE	PAGE
3.9	SAMSON MULTIPROCESSOR ARCHITECTURE	83
3.9.1	SAMSON Overview	85
3.9.2	Instruction Queue Preprocessor	86
3.9.3	Instruction Assignment Controller Option	89
3.9.4	Parallel Execution Monitor	90
3.9.5	Conflict Retry Algorithm	94
3.9.6	Dedicated SAMSON Configuration	100
3.9.7	Input/Output	101
3.10	SAMSON PERFORMANCE	102
4.0	COMMUNICATIONS FOR MULTIPROCESSORS	117
4.1	INTRODUCTION	117
4.2	MULTIPROCESSOR MODEL	119
4.2.1	Modeling Considerations	120
4.2.2	Modeling Assumptions	122
4.2.3	Unit Instruction Model	125
4.3	REQUIREMENTS OF THE INTERCONNECTION NETWORK	127
4.4	NETWORK CHARACTERISTICS	128
4.4.1	Notation	129
4.4.2	General Characteristics of One-to-One Type Interconnection Networks	130
4.4.3	General Characteristics of One-to-Many Type Interconnection Networks	133
4.5	PROGRAMMABLE CONNECTIVE NETWORK DESCRIPTIONS	138
4.5.1	Crossbar Switch	138
4.5.2	Binary Switches	140
4.5.3	Time Shared Bus Interconnections	150
4.5.4	Packet Switching (Time Slot, Loop) Network	153
4.6	NETWORK PERFORMANCE EVALUATIONS	155
4.6.1	Performance Measurement Parameter	155
4.6.2	Crossbar Performance	156
4.6.3	Binary Switch Performance	157
4.6.4	Time Shared Bus Performance	158

TABLE OF CONTENTS (cont'd)

SECTION	TITLE	PAGE
4.6.5	Packet Switching Performance	159
4.7	PERFORMANCE EVALUATION AND CONCLUSIONS	160
5.0	FAULT TOLERANCE OF THE SAMSON PROCESSOR	168
5.1	INTRODUCTION	169
5.2	THE FAULT-TOLERANT PROBLEM	171
5.2.1	Error Detection Problem	171
5.2.2	Error Recovery Problem	173
5.2.3	Reconfiguration Problem	174
5.3	ERROR DETECTION IN DIGITAL CIRCUITS	175
5.3.1	Path Sensitizing	175
5.3.2	Error Detection in Microprogrammed Central Processors	187
5.4	ACHIEVABLE SELF-TEST ERROR DETECTION FOR A CPU	189
5.4.1	Modeling of Failure Effects	190
5.4.2	General Test Philosophies	192
5.4.3	Theoretical Requirements of Self-Test	193
5.4.4	Objectives of Self-Test Software	194
5.4.5	Central Processor Architecture	196
5.4.6	Development of Comprehensive Self-Test Software	198
5.4.7	Self-Test Program	200
5.4.8	Self-Test Verification	202
5.4.9	Validation Philosophy	203
5.4.10	Results	204
5.4.11	Summary of Test Validation Procedure	206
6.0	FAULT TOLERANCE IN THE SAMSON DISTRIBUTED MULTIPROCESSOR NETWORK	208
6.1	INTRODUCTION	208
6.2	NETWORK ORGANIZATION	209
6.2.1	Characteristics of the PU	210
6.2.2	Characteristics of the Network	213

TABLE OF CONTENTS (cont'd)

SECTION	TITLE	PAGE
6.3	NETWORK ARCHITECTURE	214
6.4	DISTRIBUTED OPERATING SYSTEM (DOS)	225
6.5	ADDITIONAL FAULT TOLERANT FEATURES	228
6.6	SUMMARY	232
7.0	CONCLUSIONS	233
7.1	RESULTS	233
7.2	SUGGESTED FUTURE RESEARCH	234
BIBLIOGRAPHY		236
APPENDIX: A:	INSTRUCTION REPERTOIRE AND EXECUTION TIMES	253
B:	TRACE RESULTS	262
C:	DGN SIMULATION	281
D:	ONS ASSEMBLY LISTING	282
E:	STG TRACE	283

TABLE OF ILLUSTRATIONS

FIGURE	TITLE	PAGE
3.1-1	COMPUTATION OF A/X + B/X + CY = Z	14
3.3-1	EXAMPLE OF A DIRECTED PROGRAM GRAPH	19
3.4-1	SUMMATION OF a_i (i = 1,...,9)	28
3.4-2	MINIMUM NUMBER OF PEs REQUIRED FOR SUMMATION OF n NUMBERS	29
3.4-3	WEB FOR THE COMPUTATION OF X^{11}	32
3.4-4	WEB FOR THE COMPUTATION OF X^2 THROUGH X^{16}	32
3.4-5	COMPOUND WEB FOR THE COMPUTATION OF X^{18}, X^{24} AND X^{27}	33
3.4-6	GENERALIZED WEB FOR THE COMPUTATION OF X^n AND/OR X^2 THROUGH X^n	35
3.4-7	WEB FOR $P^7(X)$	37
3.4-8	WEIGHTED WEB FOR $P^4(X)$	38
3.4-9	$H[P^n(X)]$ vs n	39
3.4-10	WEB FOR $P^n(X)$ WHERE n = 1,2,...,36	40
3.5-1	ELEMENTARY DATA FLOW	42
3.7-1	PIPELINED FLOATING POINT ADDITION	54
3.7-2	TIME AND DATA-STATIONARY MICRO-CONTROLS	57
3.7-3	EMULATED PROCESSOR BLOCK DIAGRAM	58
3.7-4	SAMSON PE	61
3.7-5	INSTRUCTION STREAM PIPELINE FLOW DIAGRAM	63
3.7-6	INSTRUCTION STREAM PIPELINE FLOW DIAGRAM	65
3.7-7	SAMSON BRASSBOARD PE	68
3.8-1	PHYSICAL MEMORY	76
3.8-2	TYPICAL RAM ASSIGNMENT	78
3.8-3	ALTERNATING ROM/RAM	79
3.8-4	BACKING ROM	80
3.8-5	LOCAL PE PATCH ROM	82

TABLE OF ILLUSTRATIONS - (CONT'D.)

FIGURE	TITLE	PAGE
3.9-1	ADDRESSING FORMATS	84
3.9-2	SAMSON CONFIGURED WITH A COMMON MAIN MEMORY	91
3.9-3	REFERENCE TABLE AND CONFLICT RETRY BUFFER	98
3.9-4	TYPICAL REFERENCE TABLE	99
3.10-1	SPEED UP COMPARISON	108
3.10-2	EXECUTION TIME AND UTILIZATION COMPARISON	109
3.10-3	SPEED IMPROVEMENT FOR DIAGNOSTIC PROGRAM	113
3.10-4	COMBINED WEB SPEED IMPROVEMENT	115
3.10-5	COMPARISON: FIVE POLYNOMIAL COMPUTATION SCHEMES	116
4.2-1	GENERAL MODEL OF A PROCESSOR - MEMORY INTERCONNECTION NETWORK FOR A MULTIPROCESSOR SYSTEM	121
4.2-2	TYPICAL INSTRUCTION	123
4.2-3	A UNIT INSTRUCTION	123
4.4-1	LINEAR SKEWED MATRIX ELEMENT STORAGE	132
4.4-2	(A) PERFECT SHUFFLE OF 8 ELEMENTS (B) PERFECT SHUFFLE OF N ELEMENTS	134 135
4.4-3	MULTIWAY NEIGHBORHOOD INTERCONNECTION NETWORKS	137
4.5-1	CROSSBAR SWITCH	139
4.5-2	ONE-TO-MANY CONNECTION	139
4.5-3	4 x 4 BARREL SHIFTER	142
4.5-4	8 x 8 OMEGA NETWORK	143
4.5-5	8 - ITEM FLIP NETWORKS	147
4.5-6	REARRANGEABLE SWITCHING NETWORK	149
4.5-7	TIME-SHARED BUS INTERCONNECTIONS	151
4.5-8	PACKET SWITCHING TIME-SLOT OR LOOP NETWORK	154
4.7-1	NORMALIZED THROUGHPUT VERSUS NUMBER OF PROCESSORS	161
4.7-2	NORMALIZED THROUGHPUT VERSUS NUMBER OF PROCESSORS	162
4.7-3	NORMALIZED THROUGHPUT VERSUS NUMBER OF PROCESSORS	163
4.7-4	NORMALIZED THROUGHPUT VERSUS NUMBER OF PROCESSORS	164

TABLE OF ILLUSTRATIONS - (CONT'D.)

FIGURE	TITLE	PAGE
5.3-1	EXAMPLE FOR PATH SENSITIZING	177
5.3-2	SINGULAR COVERS	180
5.3-3	PRIMITIVE D-CUBES	180
5.3-4	PROPAGATION D-CUBE	180
5.4-1	SAMSON PROTOTYPE PROCESSOR BLOCK DIAGRAM	197
6.2-1	n^{th} ORDER SAMSON DISTRIBUTED MULTIPROCESSOR NETWORK	215
6.2-2	FOURTH-ORDER SAMSON DISTRIBUTED MULTIPROCESSOR NETWORK	216
6.3-1	PERFORMANCE DEGRADATION COMPARED WITH IDEAL	223
6.3-2	SAMSON THROUGHPUT IMPROVEMENTS OVER THE CONVENTIONAL SYSTEM	224

LIST OF TABLES

TABLE	TITLE	PAGE
3.7-1	Execution Times (in μ sec)	71
3.10-1	Average Normalized Execution Time versus the Number of PEs	107
3.10-2	Average Percentage Utilization versus the Number of PEs	107
3.10-3	Average Percentage Accumulator Usage	111
3.10-4	SAMSON Performance Improvement	114
4.7-1	Relative Performance of Interconnection Networks	165
5.3-1	Intersection Rules of D-Algebra	182
5.3-2	Singular Cover for Figure 5.3-1	185
5.3-3	Propagation D-Cubes for Figure 5.3-1	185
5.3-4	P-Sensitizing and Consistency Along the Path BDE	186
5.3-5	P-Sensitizing and Consistency Along the Path BF	186
5.4-1	Description of Self-Test Program	201
A-1	Instruction Repertoire	253
A-2	Instruction Execution Time	258
B-1	Normalized Execution Time for the Strapdown Guidance Program	262
B-2	Normalized Execution Time for the Omega Navigation Program	265
B-3	Normalized Execution Time for the Diagnostic Program	266
B-4	Hardware Utilization for the Strapdown Guidance Program	271
B-5	Hardware Utilization for the Omega Navigation Program	274
B-6	Hardware Utilization for the Diagnostic Program	275
B-7	Percent Accumulator Usage	280

1.0 INTRODUCTION

The broad intuitive concepts of simultaneous, concurrent, parallel operations has of late, inspired computer architects to develop systems with multiple processors in the hope of increasing the computational power of the computer. Intuitively, one can associate any of the following notions with these terms:

1. several devices working together in some manner on the same task,
2. several devices working together on different but related tasks,
3. several devices working at the same time, performing the same task but each with different data, and
4. one device performing more than one task at a time.

These different notions are, in part, responsible for the wide diversity possible within the area of concurrent processing. In a widely referenced paper, Flynn [1-1] has classified computer systems into four major categories:

- SISD, Single-Instruction Single Data Stream*: These are the conventional uniprocessor machines;
- SIMD, Single-Instruction Multiple Data Stream: Here a single-control unit broadcasts the single instruction to be executed by all active (participating) processing elements;
- MISD, Multiple-Instruction Single Data Stream: Here each processing element executes its unique instruction, independent of one another, on a single data stream;

*Stream as used here refers to the sequence of data or instructions as seen by the machine during the execution of a program.

- MIMD, Multiple-Instruction Multiple Data Stream: Here multiple independent instruction streams are executed by the multiple processing elements of the system, each on an independent data stream.

Some authors [1.2] have categorized array processors, pipeline processors and associative processors as MISD machines; however, this apparently was not Flynn's intention and these machines are not so categorized here. The well known computer architect Jean-Loup Baer [1.3] categorizes such architures as MIMD machines; it is this categorization, of such machines, which is used here.

On the classification of machine architectures as MISD processors Jean-Loup Baer has written [1.3] "To our knowledge, there exists no system which can be labeled uniquely this way". However, it is the purpose of this research to develop such an architecture.

1.1 STATEMENT OF PROBLEM

The major problem addressed by this research work is the development of a Multiple-Instruction Single Data (MISD) Stream multiprocessor* system for the purpose of surmounting the fundamental computational limitations associated with real time uniprocessor machines. Throughout this work, efficient utilization of the multiprocessor resources is considered a major goal. However, hardware utilization is considered secondary; the major goal in this research effort is the development of a high speed MISD multiprocessor.

*The ANSI Vocabulary of Information Processing defines a multiprocessor, broadly, as a system composed of two (or more) processing units under integrated control. A more restrictive attitude is to define multiprocessing as the simultaneous processing of two (or more) portions of the same program by two (or more) processing units.

The major problems in the efficient utilization of multiprocessors reside in the difficulty associated with the implementation of an integrated control mechanism within the operating system. For example, hardware architecture, software task splitting, and both hardware and software scheduling are areas where the presence of more than one processor element increases the supervisory complexity. In this research, primary emphasis is placed on hardware architecture (and hardware scheduling); however, the software operating system is briefly touched upon, where required.

In conventional multiprocessors, consisting of multiple processing units and multiple memory units, the interconnection mechanism is critical to the basic performance of the multiprocessor itself. In the MISD system developed here, the major impact of an interconnection mechanism is in the area of external input/output (I/O) communications and not within the MISD itself. Studying the conventional multiprocessor interconnection problem covers the MISD I/O interconnection problem; however, studying the latter does not necessarily include the former. Hence in Section 4 of this work, general interconnections for conventional multiprocessors have been studied.

Finally, in any network consisting of multiple processors the question of fault tolerance exists. Here we have addressed the problem of achievable error coverage within an individual processing element (Section 5) as well as the fault tolerant characteristics of a modified MISD multiprocessor (Section 6). The modified MISD could be

called an MTSD, Multiple-Task Single Data Stream multiprocessor.

1.2 SUMMARY OF RESULTS

The architecture of Simultaneous Multiprocessor Organization, abbreviated SAMSON, was developed (including working PE prototypes and a PE brassboard) capable of executing web computational graphs. The basic web structure developed, as part of this research, provides improved parallel processing results.

Web structures for the computation of: the summation of n numbers, the product of n numbers, the powers of x and polynomials of degree n were developed which produced improved execution times. Figure 3.4-2 is a plot of the improved execution time obtained for both the summation and product of n numbers. Also, the maximum execution time required to compute all of the powers of n (X^2 through X^n) was found to be $\lceil \log_2 n \rceil$ and required at most, only $n-2^{\lceil \log_2 n \rceil -1}$ or $2^{\lceil \log_2 n \rceil -2}$ processing elements, whichever is greater. For polynomials of degree n, execution times better than those obtained with the k-th order Horner's Rule, Estrin's Method, Tree Method or Folding Method were obtained. Furthermore, improved throughput performance of up to 635% were obtained (for sequential programs executing on the concurrent SAMSON architecture).

In addition, interconnection networks have been analyzed here utilizing the concept of the unit instruction. Specific conclusions regarding comparative throughput

performance were obtained; the crossbar and time shared bus structures demonstrated the highest throughput performance.

Furthermore, achievable error detection (through self-test software) of 100% was obtained and verified for the Processor Element through actual hardware testing. Finally, a distributed fault tolerant network was proposed having improved throughput, such as k-1 times that of the conventional redundant system (where k is the total number of tasks).

2.0 SUMMARY OF PRIOR WORK

The history of data processing has been characterized by a constant struggle for ever faster machines. At any given point in time, hardware technology imposes limitations on the maximum computational power obtainable with a single processor. Any attempts to circumvent these technological limitations must be accomplished through architecture improvements, such as the use of computational parallelism.

2.1 PRIOR MACHINE ARCHITECTURES

The CDC 6600 series and the IBM 360/90 series (SISD) machines achieve their computational power by overlapping the various sequential decision processes required in the execution of an instruction.

SIMD machines have been proposed by Unger [2.1], Slotnik [2.2], and by Crane and Githens [2.3] to name but a few. The SOLOMON machine [2.4] is the classical SIMD machine architure consisting of an array of 32 x 32 processing elements, each connected to its four nearest neighbors, under the global control of one central control unit. These processing elements have limited processing power; the central control unit processes a single program with each participating processor executing the same instruction stream on its own data stream. Depending upon the internal state and mode of each of these processing elements, they either participate or do not participate during an instruction execution. The internal state of the processing element is a function of the data it

is processing while the mode of the processing element is determined by the control unit. The principle means of control for each processing element are the mode commands. The major disadvantage of this machine organization and of global control structures in general, is its low efficiency when applied to typical general-purpose computations [2.5]. These computations require sequential execution; consequently, the central control unit utilizes only one (or relatively few) processing elements while all others stand idle. The SOLOMON machine has been superceded by the now famous ILLIAC IV.

The ILLIAC IV system [2.6], [2.7] alleviates the problem of the SOLOMON machine to some degree by: replacing the single global control unit with four such units, by providing I/O access to each processing element, and increasing the power of the processing elements. Each of the four global control units directly governs the operation of an 8 x 8 array of processing units. These global control units may function independently (64 processors per array) or in groups of two, three, or four arrays. These processing elements are all but devoid of local control; mode status and data dependent conditions being the only exceptions.

In such SIMD machines, the major problems revolve around finding algorithms suitable to the architecture of the machine [2.8]. In such a system (consisting of n processors), the performance would be at most $\log_2 n$ rather than n-times the performance of a single processing

element. Furthermore, if a computation is started with n processing elements, at the first conditional branch statement several of the processors will have data requiring one branch path, while others will have data requiring another path. The master control unit, however, can only command one of these two instruction streams. If branching continues we have an exponential type loss of computational resources. (Thus we have only logarithmic improvement in performance rather than the hoped for linear improvement with this class of multiprocessor).

The HOLLAND machine [2.9] is a distributed control machine consisting of an array of modules, each of which possesses some measure of local control and a limited capability for independent instruction execution. Within this structure the capability exists (to a limited extent) to execute independent programs simultaneously. HOLLAND proposed the organization to provide a basis for theoretical investigations, not as a practical device. Each module contains a storage register, operand registers and communication paths to its four nearest neighbors. An instruction stream is stored, one instruction per module with indicators to denote the predecessor and successor modules (one of the four neighbors). The instructions are executed in time sequence following the above predetermined path. The address of the operand is also stored in the module. The storage register can be loaded from an external source during the first machine cycle. During the second cycle the active module determines the operand location and establishes communications between the operand and the accumulation module.

In the final phase the instruction is executed. The proposed attribute of local control structures is the high degree of hardware utilization that can be achieved when many (small) computations are being executed concurrently; however, the HOLLAND machine architecture has fallen far short of its goal. The main problem associated with the local control organization of the HOLLAND machine is the spacial structure, of both the array and of the neighborhood communications, which results in serious path building problems.

The CDC STAR-100 and the TI Advance Scientific Computer (ASC) are based on a general pipeline approach. The instruction sets of these machines are heavily oriented towards vector operations. In both systems, several pipeline units can function in parallel. The ASC system has all identical units; hence all of these units are not mandatory. In the STAR-100 each unit performs a specific function, hence all of these units are required. These machines are here classified as MIMD processors.

The C.mmp (Carnegie Mellon multi-mini-processor) [2.10], [2.11] consists of 16 processors and 16 memory units. The C.mmp is not a geographically distributed network. In this MIMD multiprocessor configuration the processors are connected to the shared (main) memory units through a switching (interconnection) network. In this system one processor can interrupt another processor through an interrupt bus. This type network is generally referred to as a conventional (MIMD) multiprocessor.

The major drawbacks with such conventional multiprocessor architectures are path finding problems (interconnection of appropriate processors and memories) and the associated contention problems.

2.2 RELATED WORK

Recently, the concept of data flow processing has appeared in the literature [2.12] - [2.16]. In this type processor, instruction execution depends on the availability of the required operands. Because the execution of each instruction is independent, the data flow notation is convenient for expressing concurrency. Only in the past year have descriptions of machines of this type started to appear in the literature [2.17], [2.18]. The SAMSON multiprocessor system developed here is a data-driven data flow processor.

Multiprocessor interconnection networks have been described in the literature [2.19] with regard to hardware utilization, cost, etc. However, little has been formally done in the way of evaluating the effect of these networks on the overall system throughput, with the exception of the work by Danielsson and Gudmundsson [2.20].

The importance of error detectability of self-test software is due to the prevalent use of computers as an integral component in many real time control systems. In addition to these systems having to perform their desired control functions, it is necessary that they possess a high degree of self diagnostic capability and fail operability while possessing reconfigurability

characteristics. Numerous papers are appearing regularly in the literature since the first Fault-Tolerant Computing Conference in 1971.

Related work in the area of fault tolerance in real time computer networks has been underway at Stanford Research Institute (now SRI) since the early 1970s as part of their SIFT (Software Implemented Fault Tolerant) system development [2.21].

3.0 PARALLEL PROCESSING

The concept of multiprocessing and that of parallel processing are both extremely broad in scope. It is the intent of the following sections to: first introduce and develop this concept while placing some bounds on the notion of parallelism, then define the concept of parallelism within the confines of multiprocessing, then propose (introduce and develop) the concept of web structures and finally introduce the SAMSON multiprocessor capable of performing web computations. The web structure developed here transforms an arithmetic expression into a directed graph structure such that all operations at the same level can be performed in parallel while reducing both the number of levels within the structure (the execution time) as well as the number of processing elements required in the SAMSON multiprocessor.

3.1 INTUITIVE NOTIONS OF PARALLELISM

The concept of parallelism is very broad and includes a wide variety of notions. Intuitively these notions include: many devices working together in some way or other on the same task, devices working together on different but related tasks, as well as the notion of one device performing more than one task at a time. This broad notion of parallelism is, in part, responsible for the wide diversity possible within the area of parallel processing; i.e., SISD, SIMD, MISD and MIMD stream processing.

Now that these intuitive notions associated with the term parallelism have been briefly introduced we will formally define the concepts of parallelism by dividing it into two types [3.1], applied and natural parallelism.

> DEFINITION 3.1: Applied parallelism is the property that enables two or more identical operations within a set of computations to be processed concurrently on the same or distinct data bases.

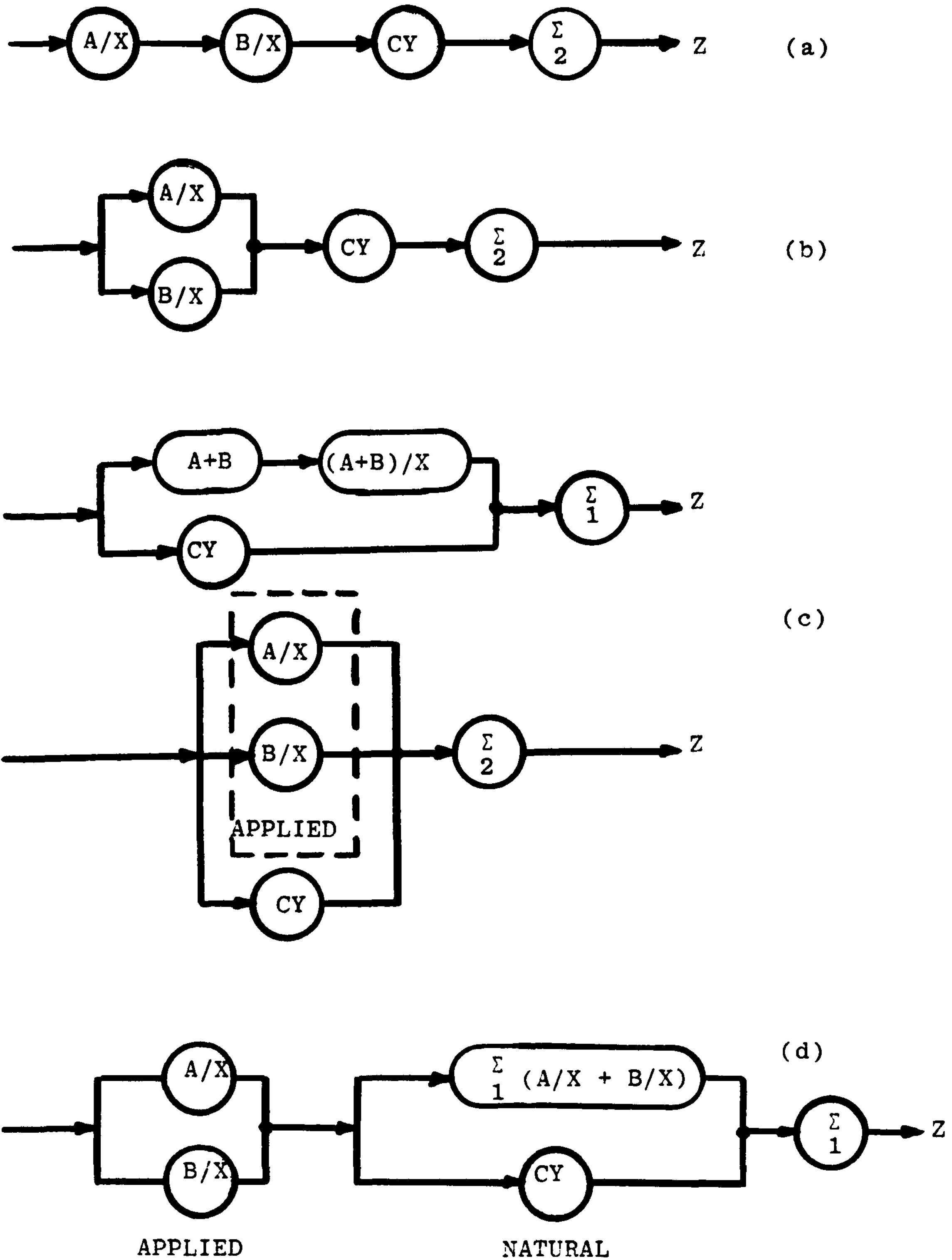
> DEFINITION 3.2: Natural parallelism is the property that enables two or more operations within a set of computations to be processed concurrently and possibly independently on the same or distinct data bases.

From these definitions, it is clear that applied parallelism is just a special case of natural parallelism. However, the distinction is made because of the important impact it has on computer organizations. Applied parallelism is efficiently handled by global control techniques since it provides common control for identical operations. Natural parallelism is efficiently handled by local control techniques since it provides independent processing for different operations. Figure 3.1-1 illustrates how the expression

$$A/X + B/X + CY = Z$$

is computed with and without the use of applied and natural parallelism.

FIGURE 3.1-1 Computation of the Expression A/X + B/X + CY=Z in (a) sequential steps, (b) utilizing applied parallelism, (c) utilizing natural parallelism, and (d) utilizing applied and natural parallelism.



3.2 EXPLOITING PARALLELISM

With the present state of today's technology, such as the development of LSI circuitry and microprocessors, it has become technologically and economically feasible to introduce multiple processors into the computer system to obtain a multiprocessing system capable of performing parallel processing [3.2].

Now that the technology exists to economically configure a multiprocessor system capable of processing many operations concurrently, we are faced with the problems of:

1. Exploiting parallelism so that computational resources are utilized efficiently, and,
2. Determining the parallel processing characteristics required of the multiprocessor.

In the exploitation of parallelism we are concerned with reducing the number of levels of arithmetic (and/or logical) operations and at the same time reducing the number of arithmetic operators at the level with the largest number of arithmetic operators. Obviously, reducing the number of levels of arithmetic operations decreases the number of sequential operations required to process an arithmetic expression and thus decreases the execution time associated with processing that expression. Likewise, reducing the number of arithmetic operators at a specific level decreases the number of processing elements (PEs) required at that level. A reduction in the number of arithmetic operators at

the level requiring the largest number of arithmetic operators reduces the total number of PEs required to process that arithmetic expression.

In this section, a web graph structure is introduced to facilitate obtaining improvements in the parallel computation of arithmetic and logical expressions.

In addition, the processing characteristics required of a multiprocessor system (to exploit the parallelism in an arithmetic expression), for the purposes of reducing both the execution time and the number of processing elements required, is discussed.

3.3 DETECTION OF PARALLELISM

Transforming an arithmetic expression into a tree representation is a well known process [3.3] - [3.8]. In this section, the web structure is introduced which is a generalization of the tree representation. The web structure is a representation of an arithmetic expression which displays the operations which can be performed concurrently. In developing the web structure, the primary interest was in obtaining a transformation that not only displays potential concurrency but also reduces both the execution time and the number of processing elements (PEs) required for the parallel processing of these arithmetic expressions.

A multiprocessor system can be thought of [3.9] as being composed of two major resources which are used to perform computations, variable resources and transformational resources.

DEFINITION 3.3: Variable resources (denoted v-resources) are utilized to preserve operand values and (temporary) results as well as instructions.

These v-resources are the source(s) (s-values) and destination(s) (d-values) of the transformational resources.

DEFINITION 3.4: The transformational resources (denoted t-resources) consist of a set of transform operators (t-operators) which transform the s-value(s) obtained from the v-resources into d-value(s) and transfer these d-values to the v-resources.

Each t-operator consists of a total ordering of the elemental transformation operators utilized by that t-operator. The particular transformational procedure to be performed by the t-resources are specified by an instruction.

DEFINITION 3.5: An instruction (I) is a specification of a t-resource and the v-resources, both the s-value(s) and the d-value(s), utilized by that t-resource.

In general, most arithmetic expressions are compound expressions requiring more than one instruction for

their specification. Such compound expressions are specified by tasks.

DEFINITION 3.6: A task (T) is partial or total ordering relation on a set of specified instructions.

3.3.1 Parallelism in Sequential Programs

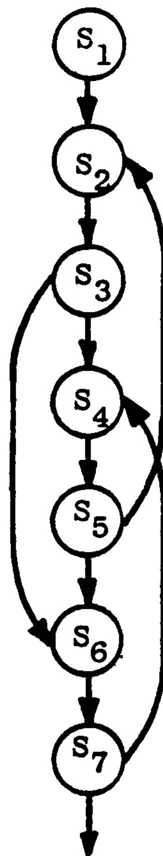
Consider a task consisting of a set of N instructions coded as a sequential program [3.10]; i.e., coded serially for a conventional uniprocessor where instructions are executed one at a time. Let the instructions I_i of this task be indexed by the assignment integer i where $1 \leq i \leq N$. Let the ordering operator $\circledcirc$ be interpreted such that if $I_i \circledcirc I_j$, then the execution of I_i precedes the execution of I_j.

In such a sequential program two types of program statements must be considered: (1) assignment statements (arithmetic and logical expressions) and (2) branch statements. Programs of this type are conveniently modeled by a directed program graph as shown in Figure 3.3-1. In the directed program graph, a node S_i represents the i-th program statement (instruction) and an arc (i,j) in the graph indicates the program flow from node S_i to node S_j. Nodes representing assignment statements and unconditional branch statements have only one outgoing arc while conditional branch statements have more than one outgoing arc. (The program flow from a conditional branch statement

FIGURE 3.3-1

AN EXAMPLE OF A

DIRECTED PROGRAM GRAPH



proceeds along only one arc at any given instant of time.) The execution route of the sequential program is specified by a sequence of arcs; i.e., (1,2)(2,3)(3,6)(6,7)(7,4), etc. The execution order of two nodes S_i and S_j is denoted by the ordering operator $\textcircled{<}$ and is specified by the execution route.

The execution order of program statements is generally unknown prior to execution of the program due to the existence of conditional branch statements. However, in a program segment containing only assignment statements, the execution order $S_i \textcircled{<} S_j$ is determined solely by the assignment order; i.e., $S_i \textcircled{<} S_j$ if and only if $i < j$.

3.3.2 Theoretical Aspects of Parallelism

For a partially ordered relation, the following situation may exist: $I_i \textcircled{<} I_j$ and $I_i \textcircled{<} I_k$, but $I_j \not\textcircled{<} I_k$ and $I_k \not\textcircled{<} I_j$. In this case more than one ordering relationship exists; i.e., I_i, I_j, I_k or I_i, I_k, I_j. At this point, the following observation can be made: (1) due to the ordering relations $I_i \textcircled{<} I_j$ and $I_i \textcircled{<} I_k$, I_i must be executed prior to either I_j or I_k, and (2) since I_j and I_k are not ordered with respect to one another, they can be executed in any order or concurrently and still preserve the task determinancy.

If the ordering relationship is total, then only one execution sequence exists; this sequence is the serial execution sequence. When the ordering relationship

is total, concurrent execution cannot occur. (Although, it may be possible to derive a partial ordering relation such that all the variables are read "correctly" and if all variables of S_i are read "correctly", then S_i is executed "correctly". Obviously, a program will be executed "correctly" if all its statements are executed "correctly".)

If I_i is a branch instruction (either conditional or unconditional), execution of I_i will cause certain subsequences of instructions to be executed more than once or not at all. If the branch instruction is a forward branch, then the destination instruction I_k ($k \neq i + 1$) is such that $i < k$; for this case the instruction string I_{i+1} through I_{k-1} will not be executed. If I_i is a backwards branch, then $i > k$ and the instruction string I_k through I_i is repeated (assuming this instruction string is branch free).

3.3.3 Conditions for Statement Independence

Two instructions I_i and I_j are said to be independent if and only if no d-value (dv) of I_i (denoted dv_i) is an s-value (sv) of I_j (denoted sv_j), no d-value of I_j is an s-value of I_i, and the d-values of I_i and I_j are disjoint. If the above conditions are not satisfied, then I_i and I_j are said to be dependent. If I_i and I_j are dependent, then a dependency exists between I_i and I_j.

Dependencies are here classified into the following three types: procedural dependencies (caused only by branch instructions, the uncertainty as to whether or not an instruction should be executed), data dependencies (dependent only on proper data values, such as observing the execution order to assure that the variables are read "correctly"; not caused by branch instructions) and procedural data dependencies (caused by a data value being dependent on the branch execution route).

Statements which satisfy the above conditions for statement independence are parallel executable [3.11]. The following definition of parallel executable statements follows directly from these requirements for statement independence.

DEFINITION 3.7: S_i and S_j are parallel executable statements if and only if

$$(dv_i \cap sv_j) \cup (dv_j \cap sv_i) \cup (dv_i \cap dv_j) = \Phi$$

where Φ is the empty set.

We are now able to make the following observations regarding parallel executable statements.

The precedence relationship which must be preserved among statements is related to both their execution order and their data dependency.

Parallel executable statements S_i and S_j are permitted to share only limited v-resources; only their s-values can be common resources. (Only $sv_i \cap sv_j$ is not required to be the empty set.)

The processing elements executing these parallel executable statements do not require direct communications between one another to execute these statements; the only reason for direct communications between these processing elements is to provide direct transmission of intermediate results to further speed up the multiprocessor system.

3.4 WEB STRUCTURES

The web structure is a graphical representation of an assignment statement which displays the operations which can be performed concurrently. This web structure can be used to obtain an improved lower bound on the number of processing elements required to process an expression without increasing the critical height of the graph structure.

A web, W, is a connected, directed graph structure consisting of: (1) a set of operands, (2) a set of nodes, $N=n_0,n_1,n_2,\ldots,n_f$ which represents the assignment operators (t-operators) of an assignment statement, E, and, (3) a set of arcs, the ordered pair node subscripts (q,r) which connect the nodes n_q and n_r of the graph. The web is a leveled structure whose levels are numbered from 0 to h in ascending order. Each

node at level r, $1 \leq r \leq h$, has one and usually two or more, inward arcs emanating from nodes at level(s) p and/or q, where $p < r$ and $q < r$. Each node at level r has one or more outward arcs to nodes at level(s) s, where $r < s$. The set of input operands are at the top level, level 0 in the structure, and they have only outward going arcs. In general, there is only one node at the bottom level, level h of the web; this node is the output node of E and is identified as the terminal node. (Sub-terminal nodes are permitted to exist at levels below h in the web structure; these sub-terminal nodes correspond to intermediate outputs. Multiple terminal nodes are also permitted in web structures and correspond to compound outputs.)

DEFINITION 3.8: Let the terminal node of a web W be at level h, then $H[W] = H$ is the height of the web. If W is a minimum height web, then the convention $H[W] = H$ is used to represent this height. This height is referred to as the critical height of the web.

We will adopt the convention that $h[n_i]$ represents the level of node n_i in the web.

DEFINITION 3.9: If there exists a string of arcs from n_i to n_j, where $h[n_i] < h[n_j]$, n_i is a predecessor of n_j and n_j is a successor of n_i. If $h[n_j] = h[n_i] + 1$ then n_j is the

immediate successor of n_i.

Obviously, the input operand set corresponds to the initial predecessor set, which is the predecessor set of the web.

With the above definitions established, we can define a syntactic web as follows:

DEFINITION 3.10: A syntactic web W [E] of an assignment statement E is a connected, directed graph whose nodes represent the operations of E to be performed on the datum. The datum is provided to the node on its input arcs by its predecessors. The datum result is provided by the node to its successor(s) on its output arc(s).

The nodes of W represent an operator for its predecessors and an operand for its successors.

DEFINITION 3.11: The height of the assignment statement E represented by the web W is h [E] = h[W]. H [E] is the convention for representing the minimum height of the assignment statement E.

DEFINITION 3.12: A binary web is a web whose nodes have a maximum of two inward arcs.

In the following paragraphs we discuss the number of processing elements, PEs, used in the web transformations of various expressions. The number of PEs required is equal to the depth of the web.

DEFINITION 3.13: The maximum depth, D [W], of a web W, is the maximum number of t-operators at any level of the web. The depth of W at level h is d [h].

The web structure described above is an n-ary web where each node can be at most an n-ary operator. In what follows, the term web will refer to binary webs.

For convenience, we shall use m to denote the number of PEs (i.e., the number of microprocessors) used by a web or at any level of the web (m=D [W] or m=d [h]). We shall use M to represent the minimum number of PEs required by a web of minimum height; M=D [W] for H [W] = H.

To provide truly useful information regarding the exploitation of parallelism, the construction of a syntactic web must depend on the operator weight, that is, the execution time required to perform each t-operator. In the simplest analysis only two basic operators (multiplication (*) and addition (+)) need be considered and they are assumed to have unit weight. The effect of differently weight operators is a direct extension of this work; one such web is presented here for completeness.

Summation of n Numbers

It is well known that the minimum number of levels $H[n,m]$ to add n numbers, $\sum_{i=1}^{n} a_i$, on m PEs is $\lceil \log n \rceil$; ($\log x$ means $\log_2 x$). Generally, the minimum number of PEs required to achieve this minimum computation time for a tree is given as $m = \lfloor n/2 \rfloor$. However, for the web structure, this is only an upper bound on M.

Consider the example where n=9. Here $m = \lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$ and $H[9,4] = \lceil \log 9 \rceil = 4$ for the tree, Figure 3.4-1a. However, for n=9 the web structure only requires M=3 while still maintaining $H[9,3] = 4$, Figure 3.4-1 b and c.

For these simple classes of computations, the web structure resembles a tree-like structure. This resemblance is to be expected in simple computations since trees are simplified, special cases of the generalized web structure. Figure 3.4-2 is a plot of the minimum number of PEs (M), required to add n numbers in the minimum height, $H[n,M]$, compared with $\lfloor n/2 \rfloor$ [3.5]. $H[n,M]$ is the minimum execution time for the summation of n numbers.

Multiplication of n Numbers

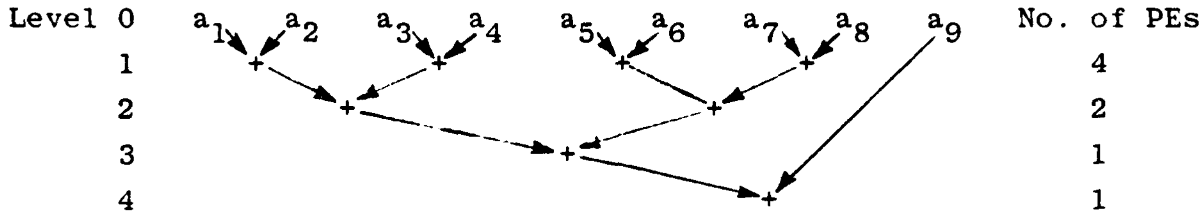
The results for the multiplication of n numbers are identical with those for the addition of n numbers.

FIGURE 3.4-1

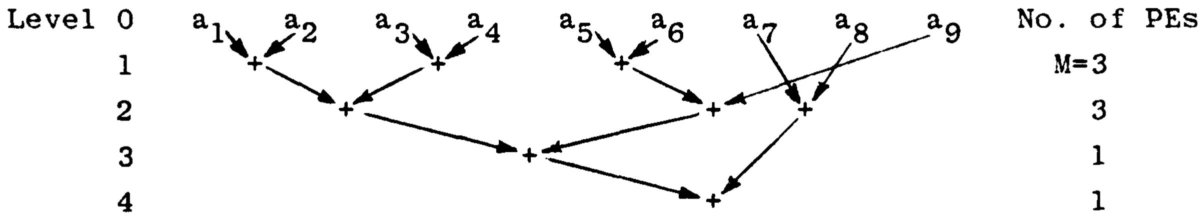
$\sum_{i=1}^{9} a_i$ (a) Tree: $m = \lfloor n/2 \rfloor = 4$, $H[9,4] = \lceil \log 9 \rceil = 4$

(b&c) Web: $M<m$; $M=3$, $H[9,3] = 4$

(a) Tree



(b) Web



(c) Web (with a reduction in the number of PEs at an earlier level)

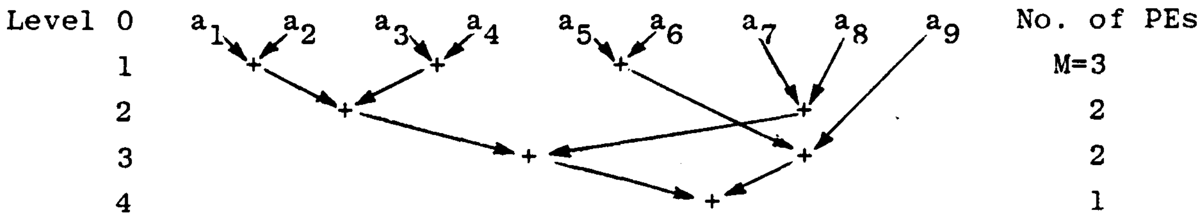
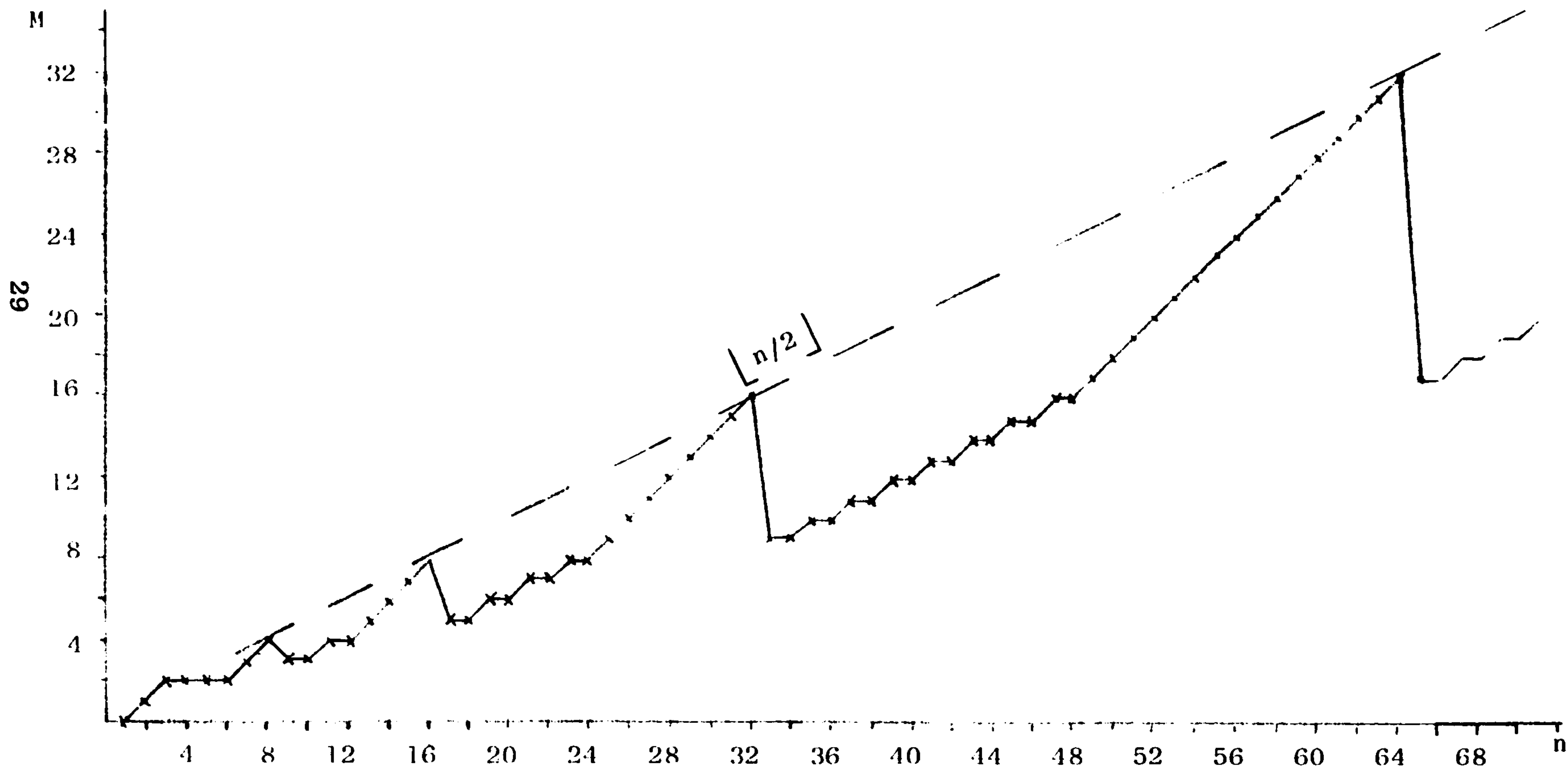


FIGURE 3.4-2

M vs n for $\sum_{i=1}^{n} a_i$ $(n = 1,2,\ldots,70)$



Minimum number of required PEs to add n numbers in the minimum height.

To avoid repetition, the case of multiplication of n numbers shall not be discussed here. It is sufficient to note that Figure 3.4-1 applies to the multiplication of n numbers if the t-operator * is substituted for the t-operator + everywhere. Likewise, Figure 3.4-2 applies equally well to $\prod_{i=1}^{n} a_i$.

Computation of Powers

In this section we will develop the web structure representation for the parallel computation of the powers of x, such as x^n where n is an integer, $n \geq 2$. We shall assume that the most powerful t-operator available for this class of computation is the multiplication operator; t-operators such as x^3, x^4, etc. are not included in the instruction repertoire of the PEs. We shall also assume that x is the only initial variable (initial predecessor). It will be clear from the web graph structure developed later in this section that the minimum number of levels required to compute x^n, assuming an adequate number of PEs (M), is $H[n,M] = \lceil \log n \rceil$ and that the minimum number of PEs necessary for this computation is sufficient (that is all that is necessary) for the web structure to obtain this minimum height. A web of height H [W] can be used to generate the powers of x (in the range x^2 through $x^{2^{H[W]}}$).

Let us first consider the computation of x^{11}. For this example, n=11 and H [11,M] = 4 when m $\geq$ 2, Figure 3.4-3. Here, the t-operator result will be substituted for the t-operator at each node. The only initial operand is x and it is located at level 0; at level 1 of the web the intermediate operand x^2 is generated. At level 2 the intermediate operand x^3 is generated and the intermediate operands x^5 and x^6 are generated at the third level. Finally, at level 4 the resultant x^{11} is produced.

A web with H [W] = 4 is sufficient to generate all of the values x^2 through x^{16}; Figure 3.4-4 depicts such a web. For this case M=8, the number of PEs at the bottom level (level 4). If all of these values of x^n (n=2,3,...,16) are not required as terminal or subterminal nodes then M $\leq$ 8. If only x^2 through x^{13} are terminal and subterminal nodes, then M=5, the number of PEs required at level 4. However, if only x^2 through x^{11} are required then M=4, the number of PEs required at level 3. In general, the number of PEs (M') is at most the number of PEs used at the bottom level (H[W]=H) or the number of PEs used at its immediate predecessor level. Figure 3.4-5 demonstrates a case where M' < M and furthermore, the number of PEs is not maximum in either of the last two levels. Figure 3.4-5 also demonstrates the web option which exists for generating intermediate subterminal and terminal nodes (i.e., $x^{18} = x^{12} * x^6$ rather than $x^9 * x^9$).

FIGURE 3.4-3 THE WEB FOR THE COMPUTATION OF X^{11}

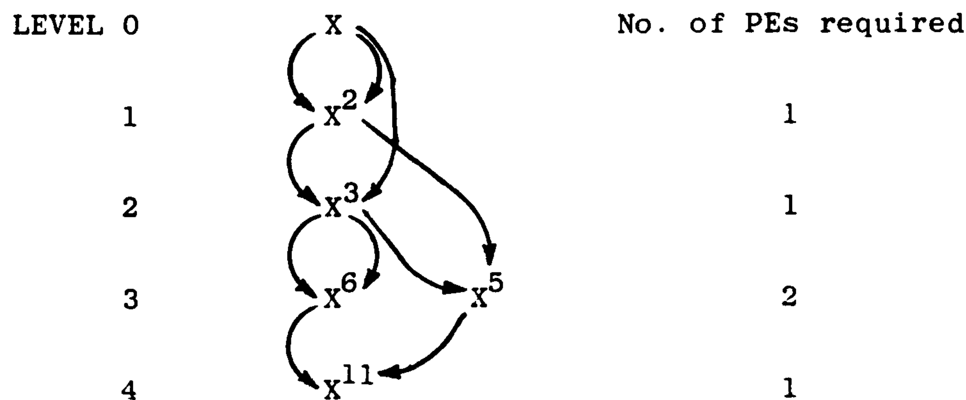


FIGURE 3.4-4 THE WEB FOR THE COMPUTATION OF X^2 THROUGH X^{16}

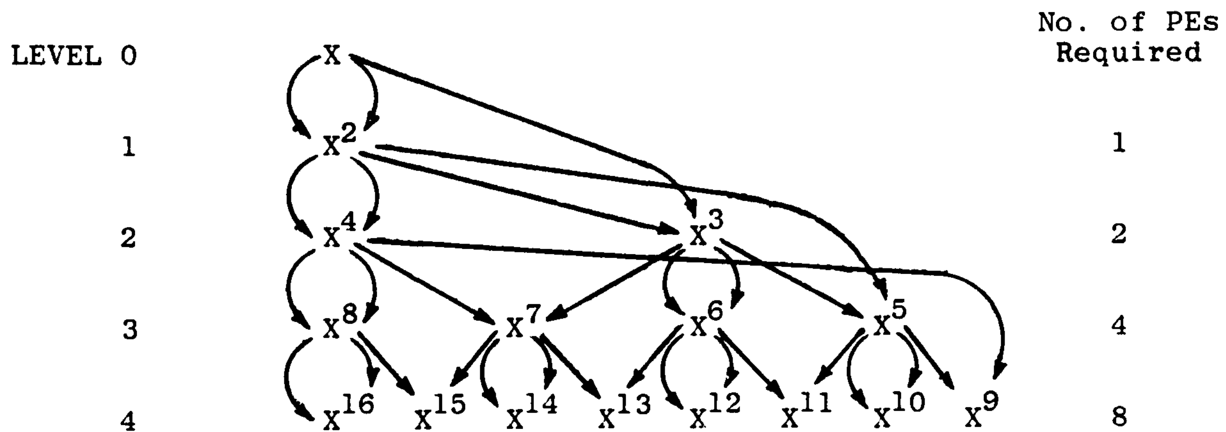
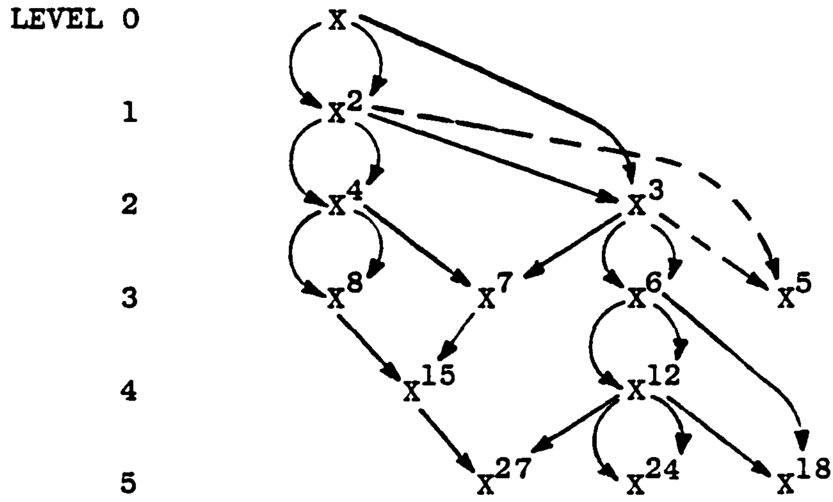


FIGURE 3.4-5

A COMPOUND WEB FOR THE COMPUTATION OF X^{18}, X^{24} AND X^{27}
(AND X^5 A SUBTERMINAL NODE IF REQUIRED)
NOTE WEB OPTION: $X^{18} = X^{12} * X^6$ RATHER THAN $X^9 * X^9$



The web of Figure 3.4-6 is a generalized web for x^n and/or x^2 through x^n. This web structure can likewise be used to directly obtain the web for x^j where $j \leq n$. At the i-th level of this generalized web, the powers of x are $2^{i-1}+1$ through 2^i and are computed using only its predecessors at levels i-1 and i-2. To achieve this degree of parallel processing, at this level of the web, at most 2^{i-1} PEs are required. The maximum number of levels required to compute x^2 thru x^n is $\lceil \log n \rceil$; this in general requires at most M' PEs where M' is the greater of: (1) $n-2^{\lceil \log n \rceil -1}$ or (2) $2^{\lceil \log n \rceil -2}$.

It is obvious that this web is optimum with regard to minimum height (computation time) and that a complete web of height $H[W] = \lceil \log n \rceil$ can be utilized to generate the powers of x (x^2 through $x^{2^{\lceil \log n \rceil}}$).

Computation of Polynomials

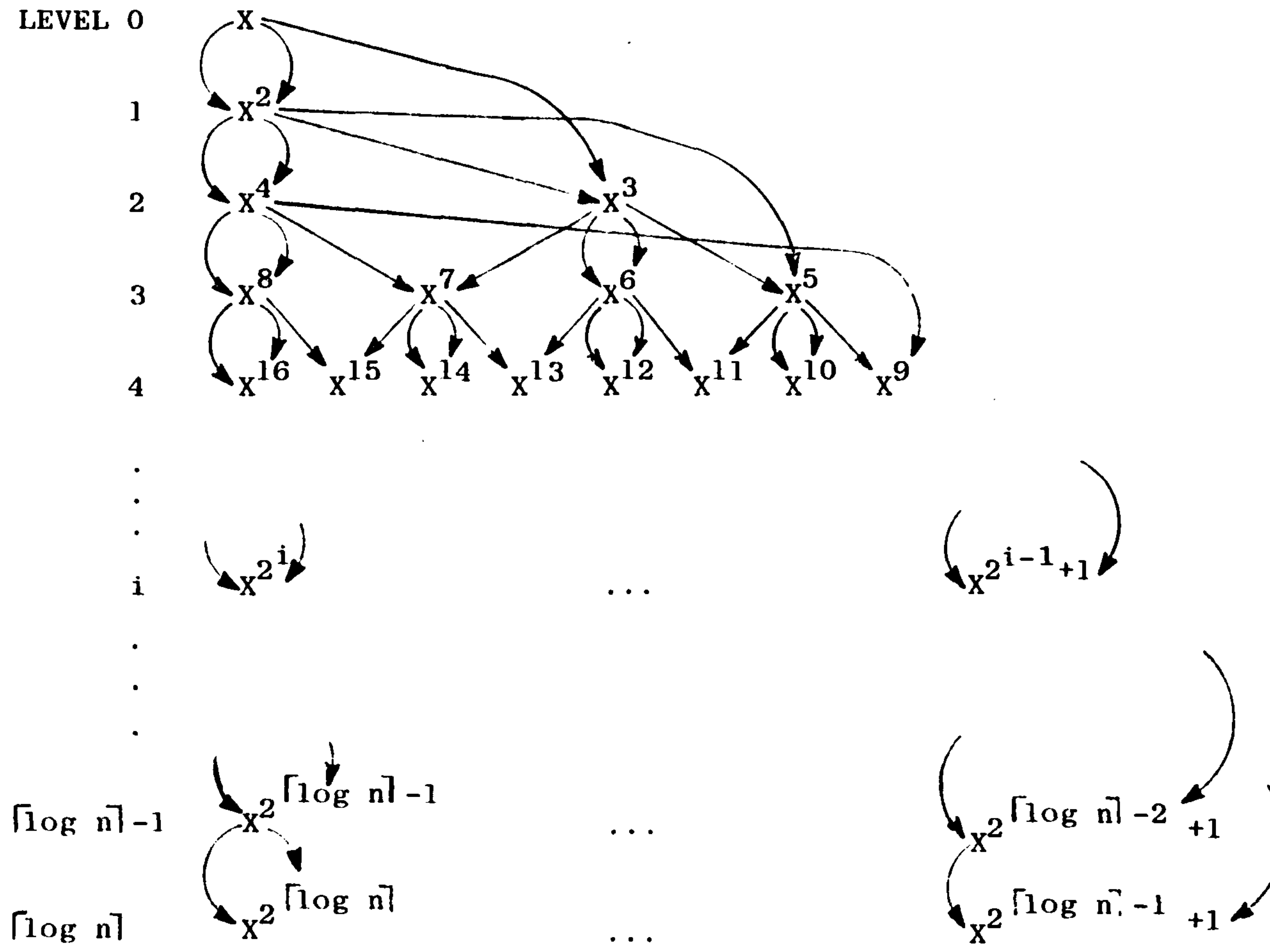
We will now consider the computation of polynomials $P^n(x)$, of degree n, utilizing the web structure.

$$P^n(x)=a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$$

We shall assume that the most powerful t-operators available for this class of computation are the multiplication and addition operators and that the only initial variables (initial predecessors) are a_i (i=1,2,...,n) and x. Furthermore, we shall assume that $P^n(x)$ is not available in factored form.

FIGURE 3.4-6

GENERALIZED WEB FOR THE COMPUTATION OF X^n AND/OR X^2 THROUGH X^n

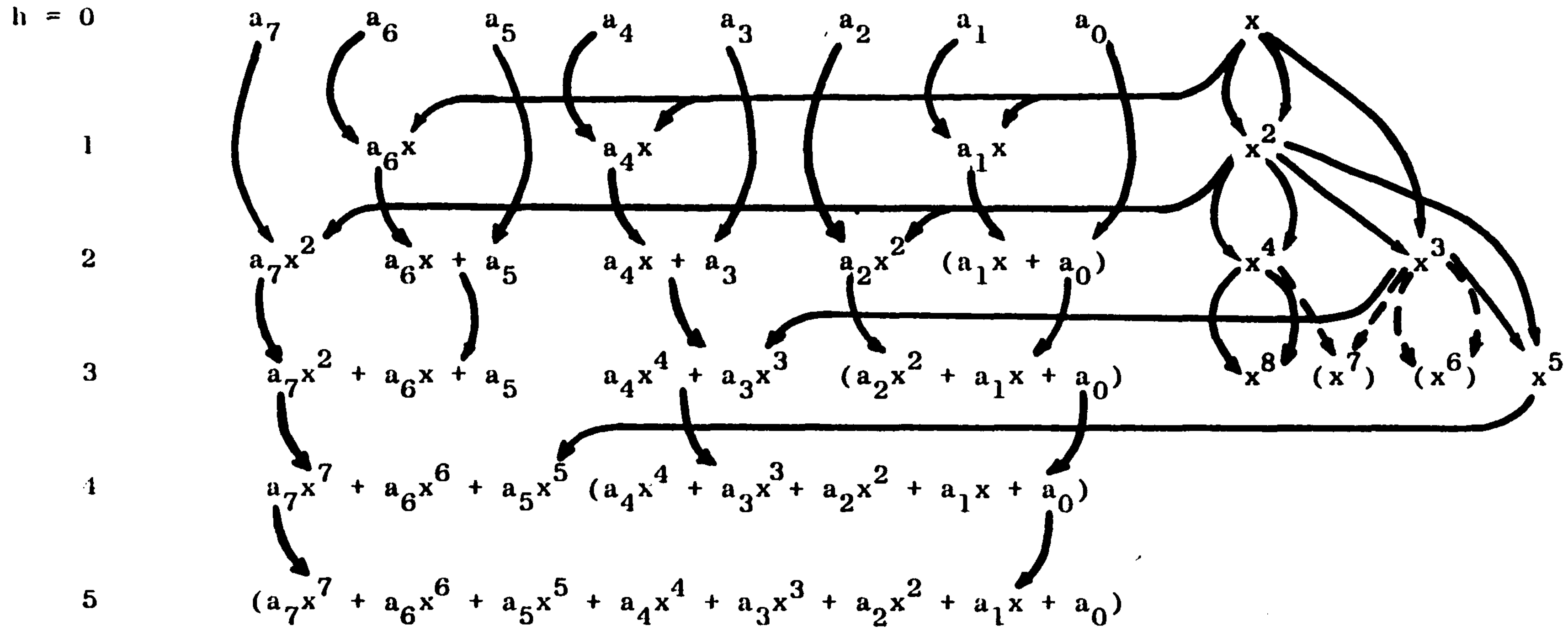


A modified form of the web used in the computation of the powers of x is utilized for the computation of polynomials. In Figure 3.4-7 a web is constructed for the polynomial $P^7(x)$; here $H[P^7(x)] = 5$. (It should be noted that X^6 and X^7 need not be generated to produce $P^7(x)$.) In Figure 3.4-8 a weighted web for $P^4(x)$ is presented; here the multiplication operator has a weight equal to three times that of the addition operator. (The addition operator execution time is t while the multiplication operator execution time is 3t.)

Figure 3.4-9 is a plot of H[W], execution time, as a function of n for the polynomial, $P^n(x)$ where $n = 1,2,\ldots,36$. The corresponding web is given in Figure 3.4-10. These web results are better in terms of computational speed than those obtained with the: k-th order Horner's Rule, Estrin's Method, Tree Method or Folding Method [3.5].

3.5 SAMSON - A DATA FLOW MULTIPROCESSOR

Recent studies of concurrent operations within computer structures have yielded a new form of program representation; this representation is known as a data flow representation. Data flow representations applied to programming languages have been described in the literature by Bährs [3.12], Dennis [3.13] and Kosinski [3.14], [3.15]. This representation of parallelism in programming languages combined with the proceeding research into parallel processing utilizing web structures sporned the concept from



Web for $P^7(x)$; $H\left[P^7(x)\right] = 5$

Figure 3.4-7

Figure 3.4-8

A Weighted Web for $P^4(x)$

$w_* = 3$ $w_+ = 1$ (hw = weighted height)

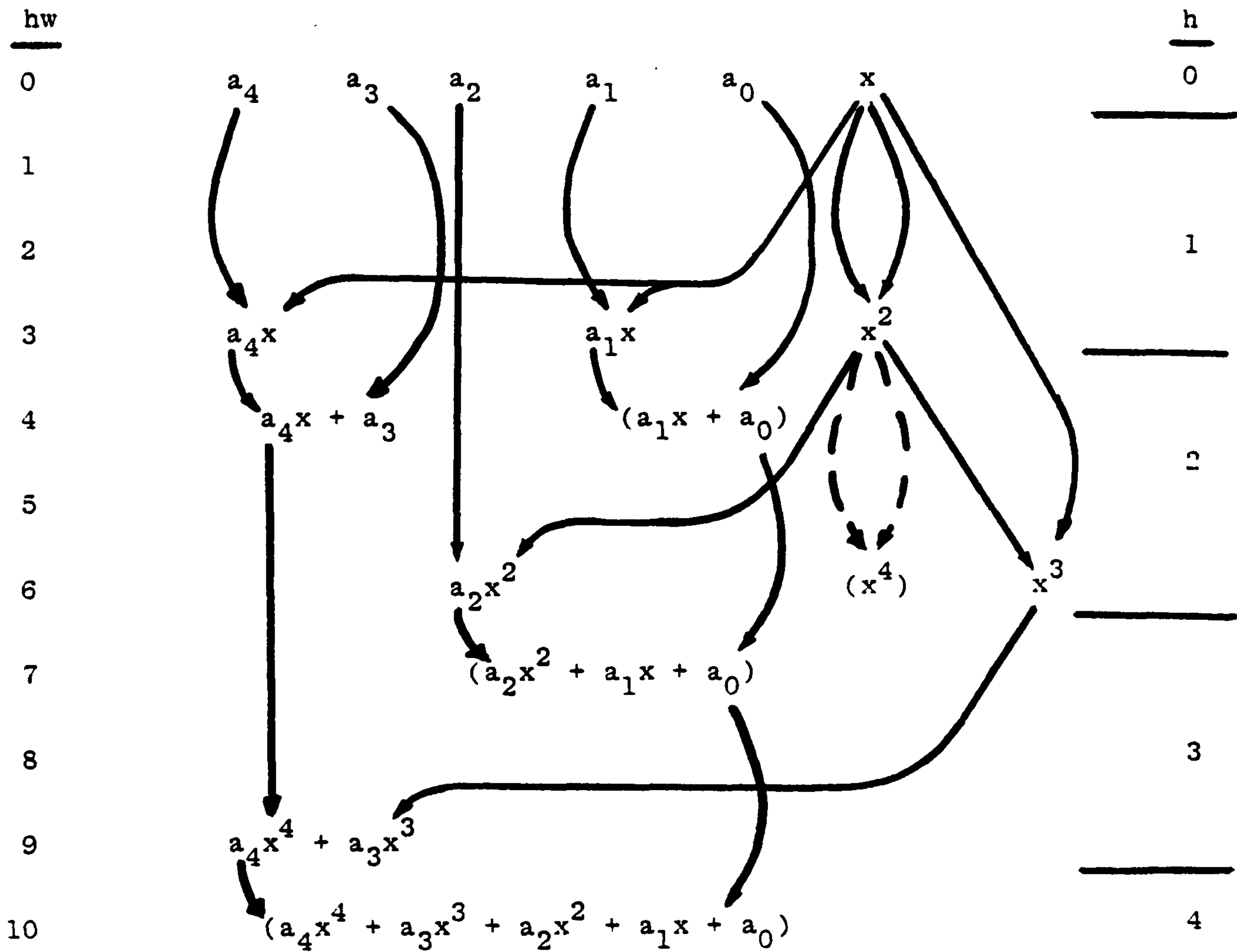
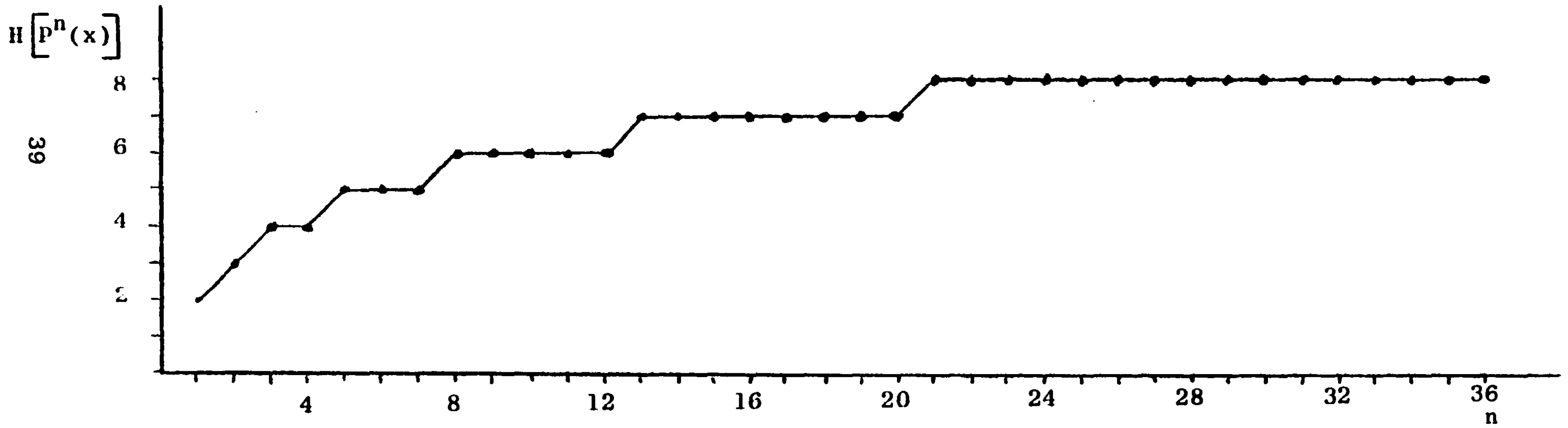


FIGURE 3.4-9

$H\left[P^n(x)\right]$ vs n

for $P^n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$



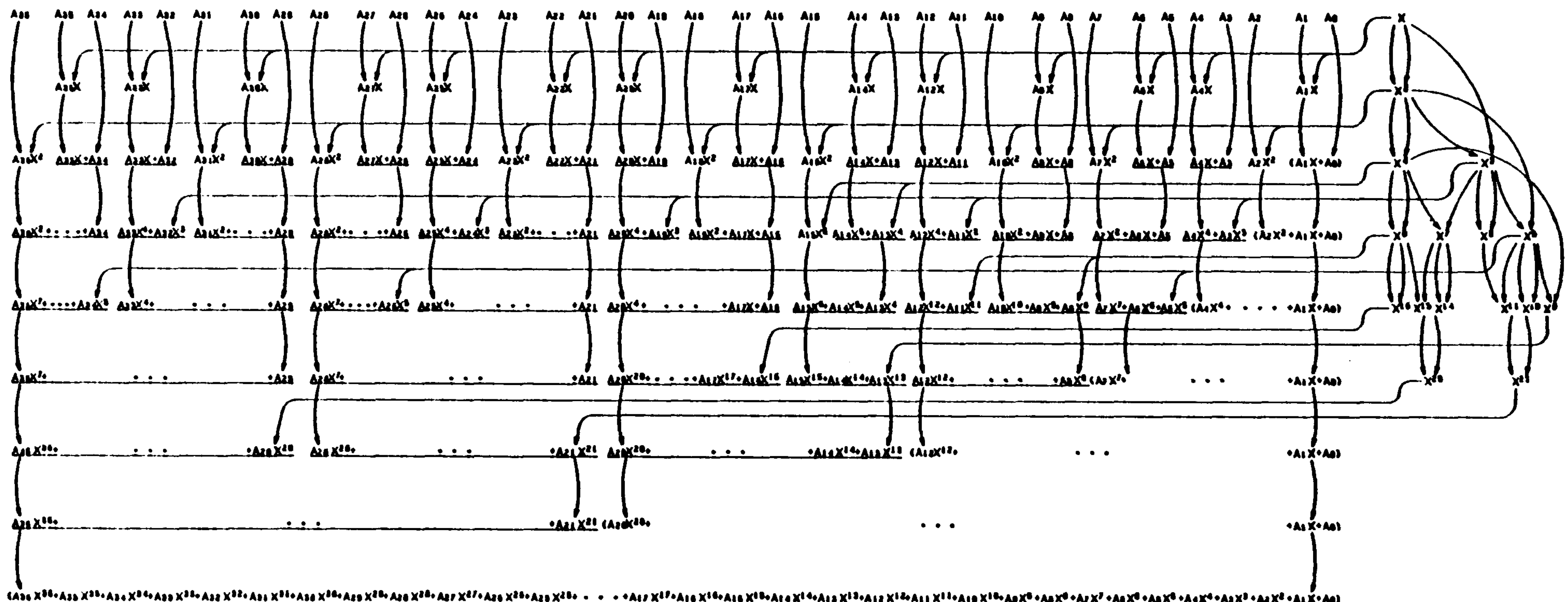
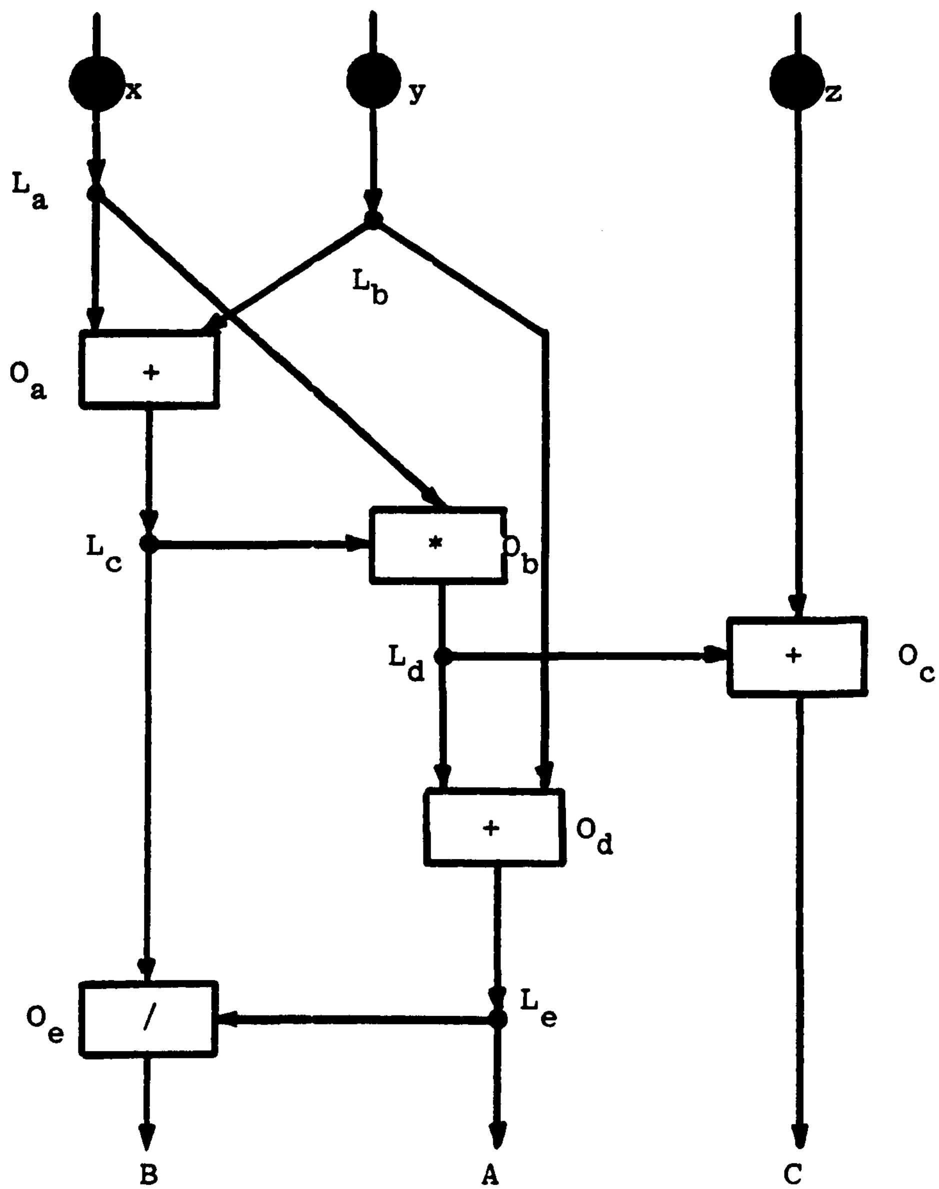


FIGURE 3.4-10
WEB FOR $P^n(x)$,
$n = 1, 2, \ldots, 36$

which the research into the SAMSON architecture developed.

Specifically, execution of an instruction is data-driven within the SAMSON multiprocessor; that is, each instruction is enabled for execution only when each required operand has been received by the executing PE (from the predecessor PE or alternately from the predecessor instruction). Hence, the SAMSON architecture executes elementary data-driven data-flow instruction streams and avoids the problems of processor switching and processor/memory interconnection. These problems are present when attempting to adapt conventional processors and memories in a multiprocessor system utilizing a massive interconnection network.

The SAMSON multiprocessor is designed to execute the elementary data-driven data-flow instruction stream which is graphically represented by the elementary data-flow web structure of Figure 3.5-1. The nodes (operators) of the web are connected by arcs along which the data values (conveyed by tokens) are transmitted [3.16]. A node is "enabled" only when all of the tokens are present on all of its input arcs. The enabled node may then fire at any time, removing the tokens on its input arcs, computing the result from the operands associated with the input tokens, and associating that result with a result token placed on its output arc. Results are sent to more than one destination by means of links which remove a token



Legend: node (operator)

links

tokens

ELEMENTARY DATA FLOW
Figure 3.5-1

on its input arc and places copy tokens on its output arcs. The nodes and links cannot fire unless there is no token present on the output arc of that node or link.

Figure 3.5-1 is an elementary data-flow web structure for the computation:

input:	x, y, z
computation	A= (x * (x+y)) + y
	B= (x + y)/A
	C= (x * (x+y)) + z
output:	A, B, C

The links La and Lb are initially enabled; the firing of La makes copies available at O_a and O_b. Likewise the firing of the L_b makes copies of y available at O_a and O_d. Once L_a and L_b have fired, in any order, O_a is enabled. After O_a has fired, completing the computation x+y, L_c becomes enabled. The firing of L_c enables the firing of O_b. After O_b has fired, L_d will fire. If a token exists on z then O_c and O_d are simultaneously enabled. The firing of O_c and O_d produce the results C and A, respectively. In addition, the firing of O_d enables L_e. When L_e fires, O_e is enabled; finally O_e fires and the result B is obtained.

It should be noted that once L_c fires, O_a can fire again provided O_a receives new tokens from L_a and L_b. Thus O_a can be producing a result corresponding to iteration n + 1 (the new token values of x and y) while

O_e is either (1) waiting to execute iteration n on values x and y (O_e is blocked), or (2) O_e is either ready (having received the input tokens but not having started execution) or, (3) O_a is executing iteration n.

3.6 MULTIPROCESSOR CHARACTERISTICS

The diversity of characteristics which can exist within a multiprocessing systems is due, in part, to the variety of notions associated with the concept of parallelism and, in part, to the inventiveness of the system architects. Rather than attempting to discuss the characteristics of the specific multiprocessing systems considered during this work, the multiprocessor will simply be defined in terms of those capabilities, generally agreed upon, which characterize it.

DEFINITION 3.14: A multiprocessor system must contain two or more, either symmetric or asymmetric, processing elements (although some definitions exclude asymmetric systems) all having access to the common shared memory as well as access to the shared I/O devices. In the multiprocessor system, there must be a single integrated operating system and, in addition, there must be intimate hardware and software interaction at all levels of the system.

To exploit the parallelism in an assignment statement for the purposes of improving the execution time, while minimizing the number of processing elements, it is desirable to impose additional requirements on the processing characteristics of the multiprocessor. These additional characteristics include: the ability to detect potential concurrency utilizing look ahead (hardware) monitors, the ability to execute multiple instructions concurrently upon the detection of this potential concurrency (while preserving the task determinacy) and the ability to communicate intermediate, intra-task results rapidly among multiple processing elements.

These features, in and of themselves, are inadequate to ensure significant improvements with a multiprocessor as compared with a uniprocessor architecture. Consider a multiprocessor containing unlimited resources (with regard to the number of processing elements); lack of a sufficient number of general registers would introduce dependencies on the actual machine code which are not dependencies in the higher level language statement of the program. Sixteen general purpose registers are provided in each SAMSON PE to minimize such dependencies. Likewise, inefficient use of the general purpose registers by the programmers, assuming an adequate number of such registers, or poor code generation by a compiler can create similar dependencies. (Giroux [3.17] reported a speed up factor of 25 being achieved for code carefully reworked.

This reprogramming effort took several years; such techniques will not be investigated here.)

Furthermore, unresolved addresses are a major cause of potential dependencies. Such dependencies have been ignored in the literature. In most third generation computers, absolute addresses are computed during run time by adding a displacement (contained within the instruction) to the contents of a base and/or index register. Although such addressing schemes permit shorter instruction word formats and greater run time flexibility in bulk data processing systems (with regard to memory management), they present a significant hazard for potential concurrent execution. Such dependencies are not apparent when dependency graphs are generated from assembly listings utilizing symbolic addresses. Consider a symbolic code segment:

Program	Comment
═══	
STO AO, I	STORE AO
═══	
ADD A1, J	ADD A1 with content of memory

The real address symbolically represented by I will have to be computed (assuming base and/or indexing is utilized, i.e., STD AO, A(x_n)) before it can be

compared with the real address symbolically represented by J (or any other memory address referenced after STO AO,I). Thus the fetch from J cannot be safely initiated. Where base, index, relative register or indirect addressing are not utilized such delays are not encountered. It should be noted that such delays would occur even if there were no actual conflict.

Thus memory addressing should be absolute wherever possible due to the delays caused by "register" and indirect addressing. However, provisions should exist so that the programmer can declare that the processing can proceed despite register and indirect addressing. However, when doing so it is the programmer's responsibility to guarantee that all instructions following that declaration (until the end of that declaration) are independent of the effective addresses utilized (e.g. they do not corrupt the effective address).

3.7 THE INDIVIDUAL PROCESSING ELEMENTS (PEs) OF SAMSON

Before proceeding with a description of the SAMSON system (including a discussion of the look ahead hardware monitors, the communications between PEs, and the throughput improvement achievable utilizing the SAMSON architecture) the individual PEs will be described. This is especially appropriate in the SAMSON multiprocessor since the architecture of each PE is that of an instruction stream pipeline organization which performs concurrent fetch, decode and

execute operations) utilizing bit-slice microprocessors and a pipelined microprogrammed sequence controller employing programmable read only memories (PROMs) to provide a flexible time-stationary microinstruction control flow. In addition, the PE incorporates an independent address processor and a broad micro-instruction word format to further enhance its execution speed. All of these features are combined in each of these PEs to achieve a high performance general purpose processing element capable of performing internal concurrent operations.

3.7.1 INTRODUCTION TO THE SAMSON PE

A description of the high performance microprocessor based PE emulation of the existing SAMSON prototype PE (which is itself a high speed, general purpose processing element) is presented here. The objective of this emulation was to improve the overall performance characteristics of the PE by more than three fold while maintaining software compatibility (with the SAMSON prototype PE). Hardware implementing this emulation has been built and tested, verifying the desired improvements. The resulting PE is completely software compatible with its predecessor, and also contains a powerful expanded instruction set. As an example of the speed improvement achieved, the interregister ADD execution time is less than 200 nanoseconds, as compared with one microsecond for the original PE. An overall improvement in throughput greater than three-to-one has been demonstrated for this PE (specifically, the throughput exceeds

1 MIPS*) without considering the additional efficiency obtainable through the use of the expanded instruction set. Additional improvements include: (a) half the weight, size and power dissipation, (b) reduced cost, (c) increased reliability, and (d) improved testability and fault isolation capability. The architecture of this new PE incorporates bipolar microprocessor technology, instruction stream pipelining, a pipelined microprogram sequence controller, and concurrent processing techniques. The resulting machine is an advanced state-of-the art real time PE that outperforms traditional processor organizations. The bit-slice microprocessor based PE utilizes an instruction stream pipeline organization which provides concurrent fetch, decode and execute operations. Instruction stream pipelining is accomplished through the use of independent units for the arithmetic computation, the address processing and the microprogram sequencing. These functional units are implemented utilizing 4-bit bit-slice microprocessors, bit-slice microprocessor support components, and standard MSI, LSI and PROM devices, respectively. A time-stationary micro-instruction control organization utilizing a PROM micromemory provides a highly efficient and versatile PE structure.

Unlike conventional pipeline processors (i.e., the

---

*MIPS, millions of instructions per second

IBM 360/91) whose arithmetic computational resources are divided into several sequential computational stages, each of which processes an independent set of data, an instruction stream pipeline processor is one that has several instructions in various phases of execution at the same time without necessitating independent, multiple computational resources. (This instruction stream pipeline processor, PE, requires only a single arithmetic computational unit).

This microprocessor based PE is not intended as a panacea for the EDP industry; rather, it is a small, low cost, low power, real time processor providing the high speed computational capability necessary to satisfy a broad spectrum of real time applications.

3.7.2 General Overview And Definitions

The microprocessor is a relatively new component which is emerging as a major tool for both logic and system designers. This microprocessor technology has now matured sufficiently so that it is both technologically and economically feasible to design and fabricate a high performance processing element, through the use of microprocessor bit-slice components, having the performance improvement desired here.

Traditionally, the speed, size and power dissipation improvements which have evolved in the processor area have resulted from technological gains obtained primarily through the use of new circuitry, i.e., logic

elements becoming significantly faster and significantly more sophisticated, while dissipating less power. In and of itself, the technology only provides a throughput improvement factor of approximately 1.8; however, this is not adequate to provide the high performance desired here. Perhaps technologies and circuitry presently in the development stages [3.18], [3.19] will provide this capability; unfortunately, these techniques will not be available in the immediate future. Consequently, in order to obtain significant performance improvements over traditional organizations, utilizing those components which are currently available, internal architectural changes to the PE structure were required [3.20], [3.21]. The resulting throughput improvement factor obtained with this PE architecture exceeds 3.8 times that of the predecessor PE.

The PE described in this paper is a microprogrammed, bit-slice microprocessor based machine which utilizes a pipelined, microprogrammed sequence controller and an instruction stream pipeline organization to provide concurrent fetch, decode and execute operations. This bit-slice microprocessor architecture has been successfully used to emulate* the existing SAMSON

*Here emulation is distinguished from simulation, such as: in an emulation one strives to equal or excel as compared with a simulation which simply tries to imitate or act like the original machine.

prototype PE, while providing improved performance characteristics. These improved performance characteristics include:

- o Upwards software compatibility (including an expanded instruction set)
- o Improved instruction execution times (i.e., a five-to-one improvement in the interregister ADD execute time) and,
- o Throughput improvement in excess of three-to-one (without considering the additional efficiency obtainable through use of the expanded instruction set).

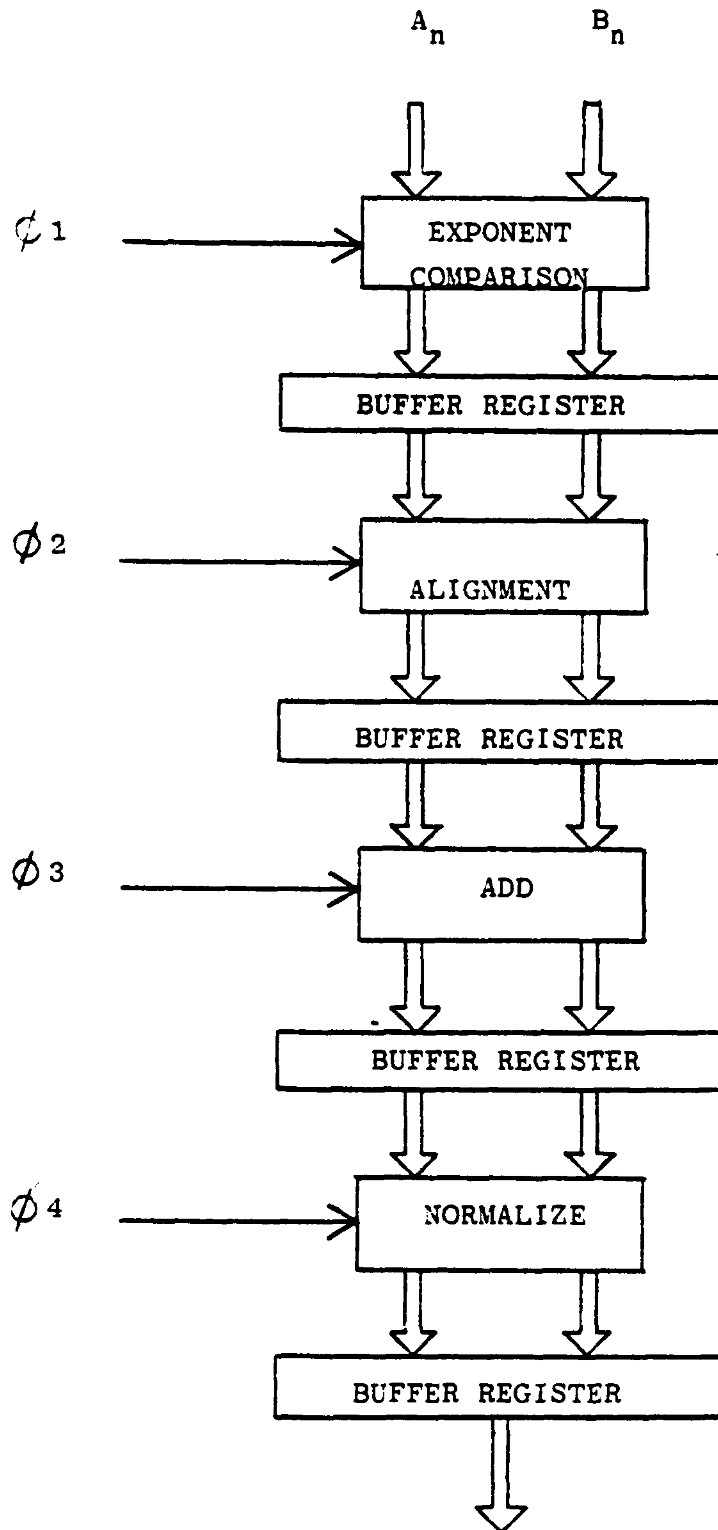
This PE is ideally suited for applications where the above factors, as well as cost, reliability and fault isolation are prime considerations.

Before proceeding further, the terms "microprogrammed", "pipelined" and "time-station micro control structure" will be defined.

The term "microprogram" was first introduced by M. Wilkes in 1951 in his paper, "The Best Way to Design an Automatic Calculating Machine" [3.22]. He used this terminology to describe the entire ensemble of micro-operations (the basic elemental micro-control

operations) used in his ordered logic matrix, which replaced the random, hardwired control logic within the CPU. The term microprogrammed as used here implies a machine employing a fixed micro-store (i.e., a read only memory) implementation of a particular machine architecture or one which is designed with the capability of being micro-coded (by the processor architect) to implement several machine languages and architectures. (This is in contrast to microprogrammable machines, utilizing writable control stores, which support user generated microprograms.)

Originally, the term pipeline was introduced to describe pipeline processors, the class of processors (including the CDC 6600 and the IBM 360/91) where computational operations were divided into several sequential stages; each of these stages processes an independent set of data at the same time. As a simple example, let us consider a floating point add instruction. This operation is easily partitioned into four basic operations; i.e., comparison of exponents, alignment, addition, and normalization. In performing this instruction, independent hardware blocks can be associated with each of these functions and interconnected by inter-stage buffers (Figure 3.7-1). The operands flow through this network in much the same way as fluid flows through a pipeline; hence the name pipeline processor [3.23].



PIPELINED FLOATING POINT ADDITION

FIGURE 3.7-1

Basically, a pipeline device is one that is capable of accepting new inputs before the processing of previously accepted inputs have been completed. This terminology has recently been adopted to describe devices organized as a one-word, word-parallel shift register, wherein the information generated in cycle i (which is required in cycle i+1) is placed on the doorstep of the pipeline (buffer) register during cycle i and captured by the pipeline register on the transition from cycle i to cycle i+1. It is this more recent interpretation of the term pipeline which is used throughout this paper when discussing the pipelined microprogram sequence controller.

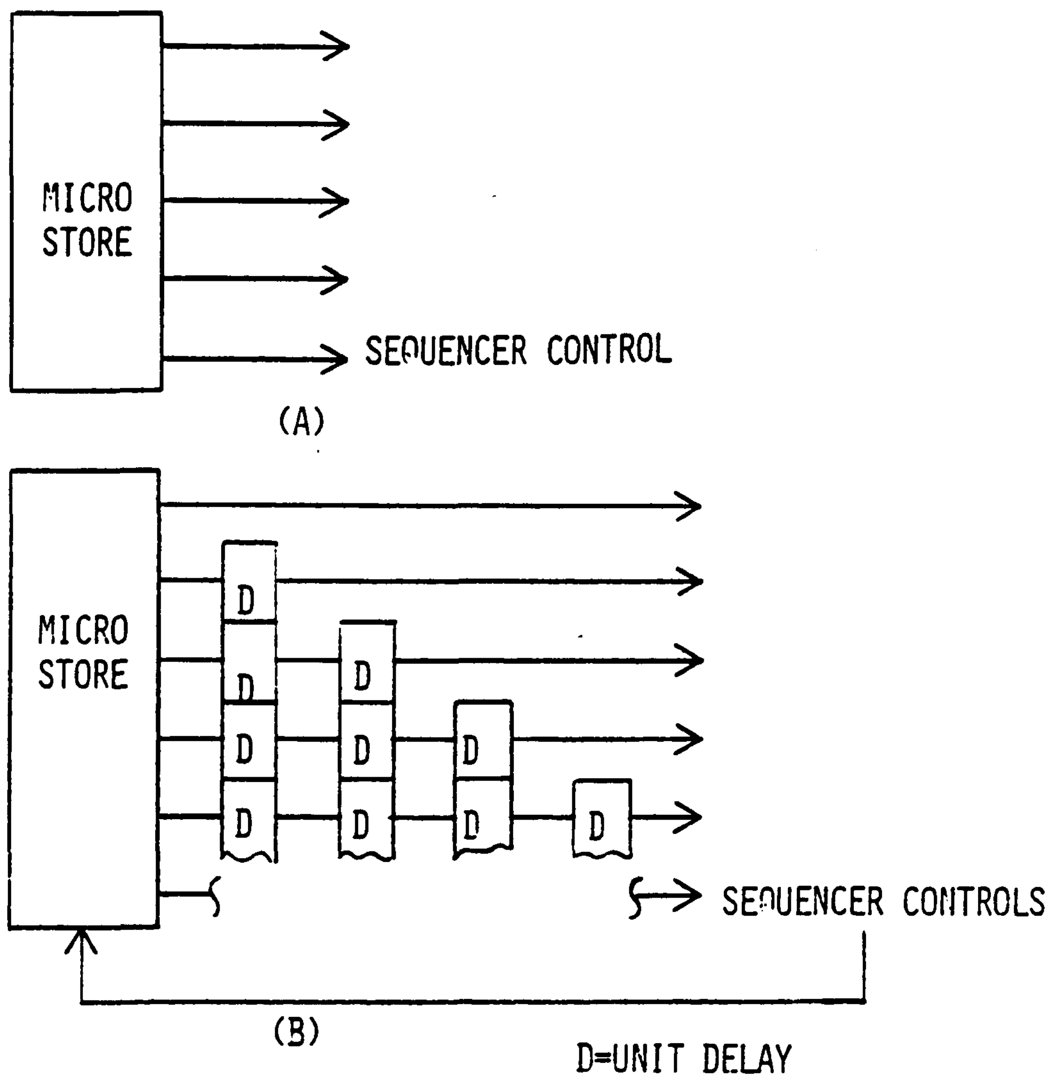
An instruction stream pipelined organization,(described later) like the pipelined processor [3.24], provides two extremely different types of control; namely, control of the processing of each independent data set at every stage of the pipeline as well as control of the unique routing of each datum. Such control can be achieved in a microprogrammed control structure in one of two ways. In the first of these approaches, the micro-instruction specifies all of the activities in the pipeline for that instance of time. Alternatively, the organization can be such that the micro-instruction flows through the pipeline with the data, providing control for several cycles. These two types of micro-program control are classified as time-stationary and data-stationary, respectively.

In time-stationary microprograms, each micro-instruction specifies all of the micro-controls of the pipeline for a single machine cycle. The entire state of the machine is defined by the current micro-instruction. For the case of data-stationary micro-programs, each micro-instruction specifies all of the paths for multiple micro-cycles to be taken for each datum. Thus, during any machine cycle, the state of the machine is determined by several currently active micro-instructions [3.25]. Both types of micro-control structures are shown in Figure 3.7-2.

3.7.3 Emulated PE Description

The SAMSON processing element which has been emulated is organized as a microprogrammed parallel general purpose machine operating on sixteen bit data words. This processor provides maximal computational capability in minimal size while utilizing standard MSI and LSI circuits. The essential elements of this processor are shown in the block diagram of Figure 3.7-3. They include: the 16 general purpose operational registers, arithmetic unit, micro-control unit, input/output unit, program counter, specialized single bit indicators, control logic and timing unit.

The sixteen general purpose registers (accumulators) are operated on primarily through the use of a powerful set of interregister instructions. Provisions are also made to utilize two of the registers as index registers during memory reference operations. In



(A) TIME-STATIONARY MICROARCHITECTURE
(B) DATA-STATIONARY MICROARCHITECTURE

TIME AND DATA-STATIONARY MICRO CONTROLS

FIGURE 3.7-2

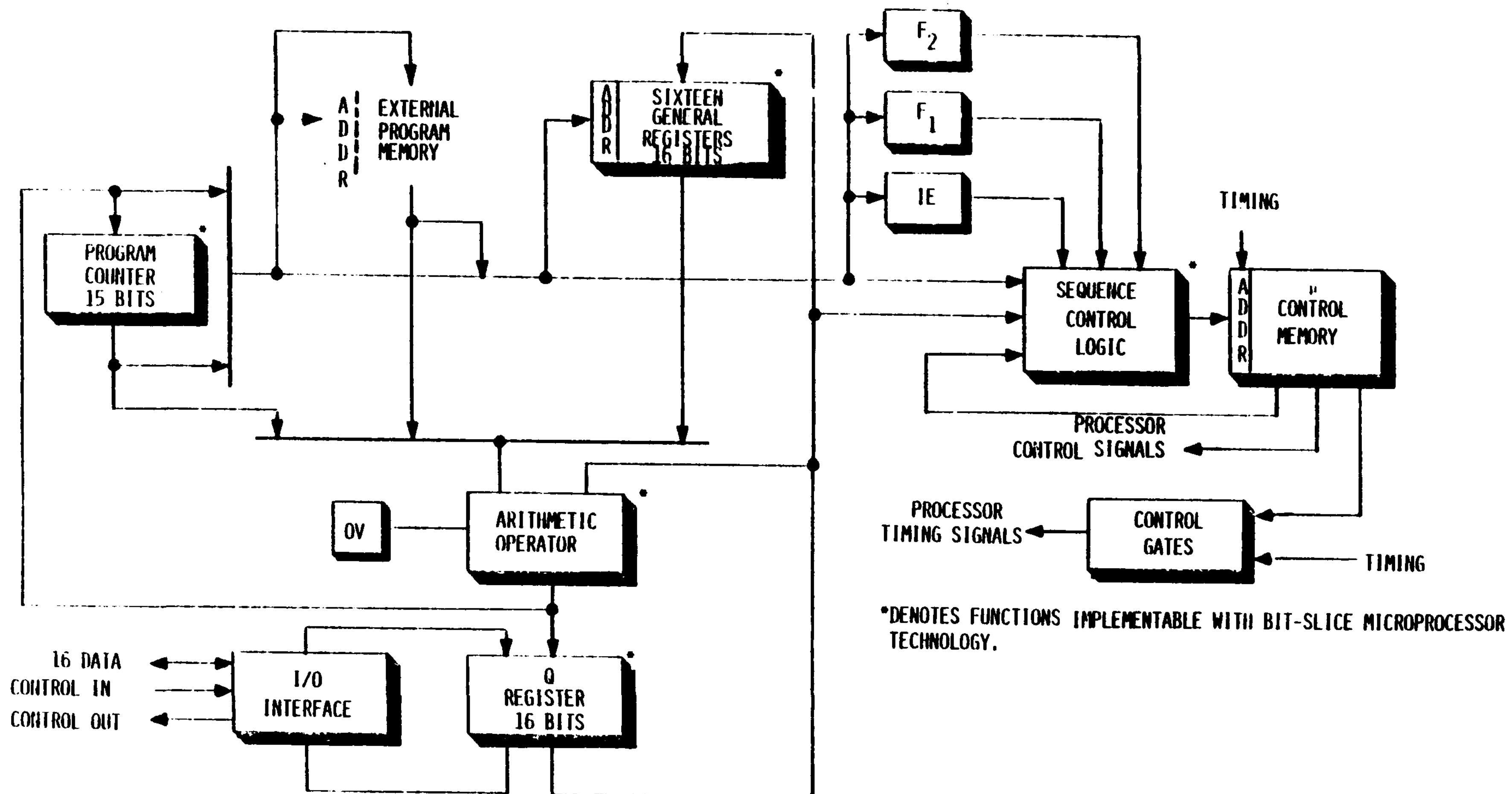


Figure 3.7-3

EMULATED PROCESSOR BLOCK DIAGRAM

addition, sequential registers are automatically linked for double length operations.

The arithmetic unit provides the capability to perform arithmetic and logical operations on the machine registers and memory. It provides this capability through the use of an adder/shifter together with an arithmetic unit internal register. Information from the program counter, program memory, input/output unit and the 16 general purpose registers is routed through the arithmetic unit under control of the micro-control unit.

The micro-control unit is the basic source of all processor control signals. The heart of this unit is a micro-store implemented with programmable read only memories (PROMs).

3.7.4 Architecture Of The SAMSON Microprocessor Based PE

The architecture of this microprocessor based PE is that of an instruction stream pipeline organization utilizing a pipelined microprogrammed sequence controller employing programmable read only memories (PROMs) to provide a flexible time-stationary micro-instruction control flow. In addition, the machine incorporates an independent address processor and a broad micro-instruction word format to further enhance its execution speed.

All of these techniques are combined in this machine to achieve a high performance general purpose PE which is small enough to fit on a single card. (This small size is essential if a network consisting of a relatively large number of PEs is to be developed.) This small size must be and is accomplished with readily available parts; no special purpose devices are required. The architectural structure, shown in Figure 3.7-4 consists of three independent units: the main processor which performs the arithmetic and logical computations, the address processor which performs the required address computations, and the microprogram sequence controller. The main processor, implemented with four 4-bit bit-slice microprocessors, provides the full parallel sixteen bit arithmetic operations as well as the internal arithmetic register and the sixteen general purpose registers. In the address processor, four 4-bit bit-slice microprocessor support components serve as the program counter, memory address register and address computational unit. Finally, the pipelined microprogrammed sequence controller, which consists of the micro-address control logic, the micro-memory and the micro-memory pipeline register, utilizes standard MSI, LSI and PROM devices.

A broad micro-instruction work format has been utilized to further maximize execution speed; the micro-cycle operates in less than 200 nanoseconds and simultaneously controls all of the following functions:

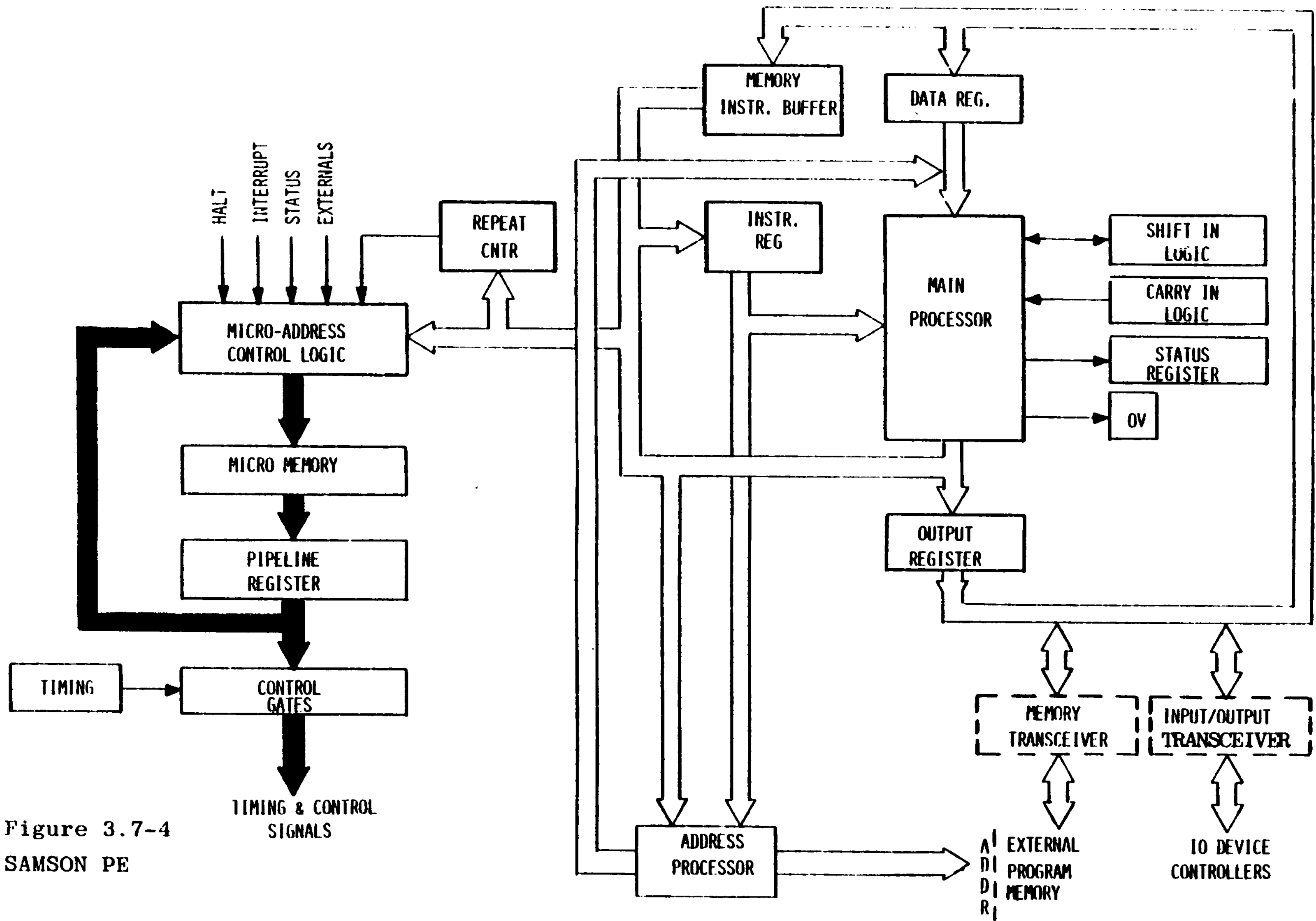


Figure 3.7-4
SAMSON PE

1) the setting of various multiplexers and data paths,
2) the arithmetic processing,
3) the address computation,
4) the memory and I/O operations, and
5) the microprogram sequence flow

The technique of instruction stream pipelining utilized in this processor provides the high speed execution of machine instructions by performing parallel activities within the execution of the instruction stream. In the instruction stream pipeline flow diagram, Figure 3.7-5, it is shown that the execution of an instruction can be considered as consisting of four sequential phases. Each phase is time overlapped with the other phases, causing several instructions to be in various stages of completion at any given time. Execution of a single instruction may require four, five, six or more sequential machine cycles; e.g., instruction address calculation, instruction fetch, instruction decode, (operand address calculation, operand fetch), and one or more arithmetic operations. If the first three of these phases of different instructions are overlapped (performed in parallel) then the actual execution time of each instruction is reduced [3.26].

DESCRIPTION OF INSTRUCTION
A EXECUTION:

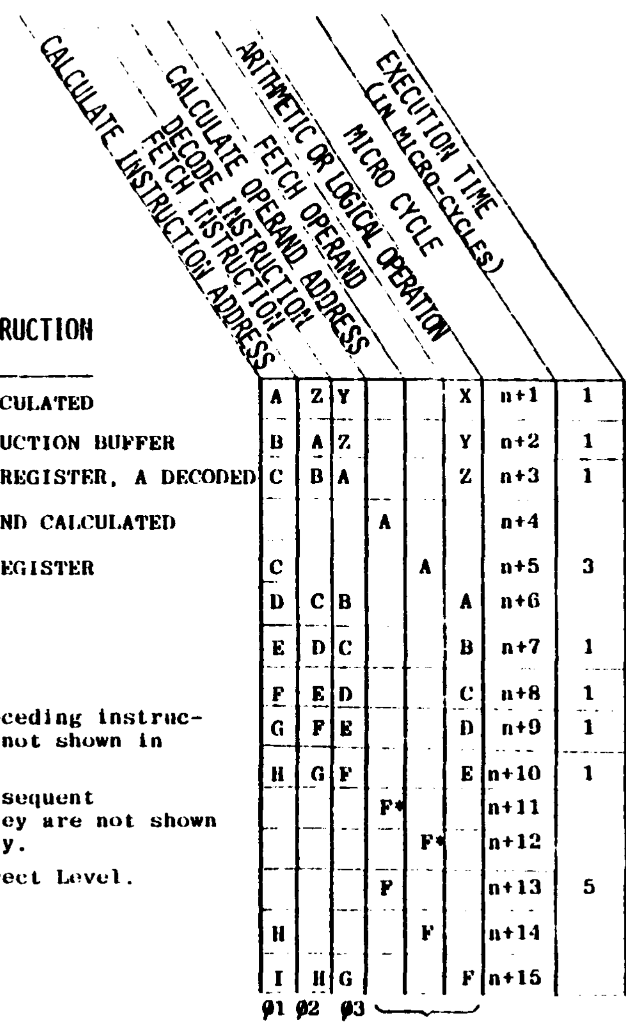
1. ADDRESS OF A CALCULATED
2. A → MEMORY INSTRUCTION BUFFER
3. A → INSTRUCTION REGISTER, A DECODED
4. ADDRESS OF OPERAND CALCULATED
5. OPERAND → DATA REGISTER
6. ADD OPERATION

NOTES:

a. X, Y & Z are preceding instructions, they are not shown in their entirety.

b. G, H & I are subsequent instructions, they are not shown in their entirety.

c.* Indicates Indirect Level.



Ø1	Ø2	Ø3	Ø4	Ø4	Ø4	MICRO CYCLE	EXECUTION TIME
A	Z	Y			X	n+1	1
B	A	Z			Y	n+2	1
C	B	A			Z	n+3	1
			A			n+4	
C				A		n+5	3
D	C	B			A	n+6	
E	D	C			B	n+7	1
F	E	D			C	n+8	1
G	F	E			D	n+9	1
H	G	F			E	n+10	1
			F*			n+11	
				F*		n+12	
			F			n+13	5
H				F		n+14	
I	H	G			F	n+15	

SAMPLE PROGRAM

INSTRUCTION	TYPE
X	INTERREGISTER
Y	INTERREGISTER
Z	INTERREGISTER
A	MEMORY REFERENCE ADD
B	INTERREGISTER
C	INTERREGISTER
D	INTERREGISTER
E	INTERREGISTER
F	INDIRECT ADDRESSED MEMORY REFERENCE ADD

Figure 3.7-5 INSTRUCTION STREAM PIPELINE FLOW DIAGRAM

In Figure 3.7-6, consider I_a the first instruction, i.e. initially, the instruction pipeline is empty (m+1 through m+3 non-existent); in the first phase, the address processor calculates the location of the first instruction, I_a. In the second phase, the instruction I_a is fetched from the external program memory and placed in the memory instruction buffer while the address processor generates the address of the next instruction. During the third phase, instruction I_a proceeds to both the microprogrammed sequence controller (where the instruction is decoded) and the instruction register. Simultaneously, instruction I_b enters the memory instruction buffer and the address processor generates the address of the next instruction. In the fourth and final phase the pipeline is full. The address of I_d is calculated, I_c is fetched from the program memory, I_b is decoded and I_a begins execution. Depending on the type of instruction, the execution phase may require one or more micro-cycles.

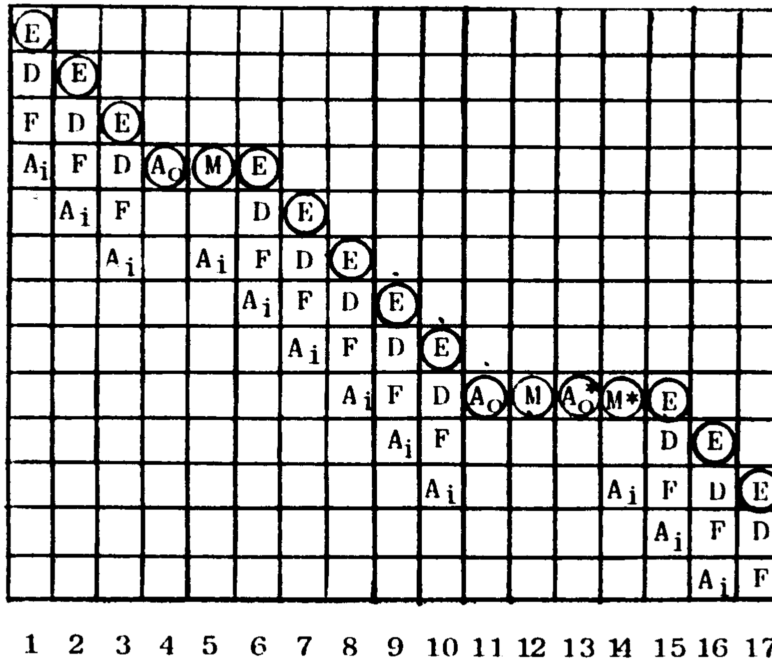
In the case of an interregister ADD instruction or any "one cycle instruction" only one execute micro-cycle (the arithmetic operation) is required. Other interregister instructions may require two or more execute micro-cycles.

In the case of a memory reference instruction, the execute phase consists of a minimum of three execute micro-cycles. The address processor calculates the operand address, rather than the next instruction

Figure 3.7-6 - INSTRUCTION STREAM PIPELINE FLOW DIAGRAM

SAMPLE PROGRAM

INSTRUCTION NO.	INSTRUCTION TYPE	EFFECTIVE EXECUTION TIME IN MICRO-CYCLES
m+1		1
m+2	INTERREGISTER	1
m+3	INTERREGISTER	3
m+4 (I_a)	MEM. REF. ADD	1
m+5 (I_b)	INTERREGISTER	1
m+6 (I_c)	INTERREGISTER	1
m+7 (I_d)	INTERREGISTER	1
m+8 (I_e)	INTERREGISTER	5
m+9	INDIRECT ADDRESSED MEM. REF. ADD	1
m+10	INTERREGISTER	1
m+11	INTERREGISTER	
m+12		
m+13		



TIME (MICRO-CYCLE)

LEGEND:
- A_i = INSTRUCTION ADDRESS CALCULATION
- F = INSTRUCTION FETCH
- D = DECODE INSTRUCTION
- A_o = OPERAND ADDRESS CALCULATION
- M = MOVE OPERAND TO DATA REGISTER
- E = EXECUTE OPERATION
- * = INDIRECT LEVEL

CIRCLED SYMBOLS INDICATE EXECUTE MICRO-CYCLES

address, during the first execute micro-cycle. In the second execute micro-cycle the operand is fetched into the Data Register and the address of I_c is regenerated (the address of I_c was destroyed when the operand address was calculated). In the third execute micro-cycle, assuming direct operand addressing, the appropriate execute operation is performed and the calculated address, fetch and decode phases of instructions I_d, I_c and I_b are performed, respectively.

If the instruction stream consists of a string of "one cycle instructions", e.g., I_b through I_e, the result of I_b emerges from the main processor as the address of I_e is calculated. Clearly, this architecture is capable of handling large streams of instructions at high data rates while making maximum utilization of the memory resources. In the instruction stream pipeline flow of the sample instruction mix, depicted in Figure 3.7-6, cycles 7 through 10 clearly demonstrates how the instruction stream pipeline improves the execution speed of a string of sequential interregister instructions. The actual execution time for these instructions is one micro-cycle each; the processing rate (once the pipeline is full) is thus four times that of the non-pipelined organization. From this example, it is obvious that maximum utilization of the memory is achieved (executing one instruction per memory cycle).

To interface with lower speed memories, as well as the

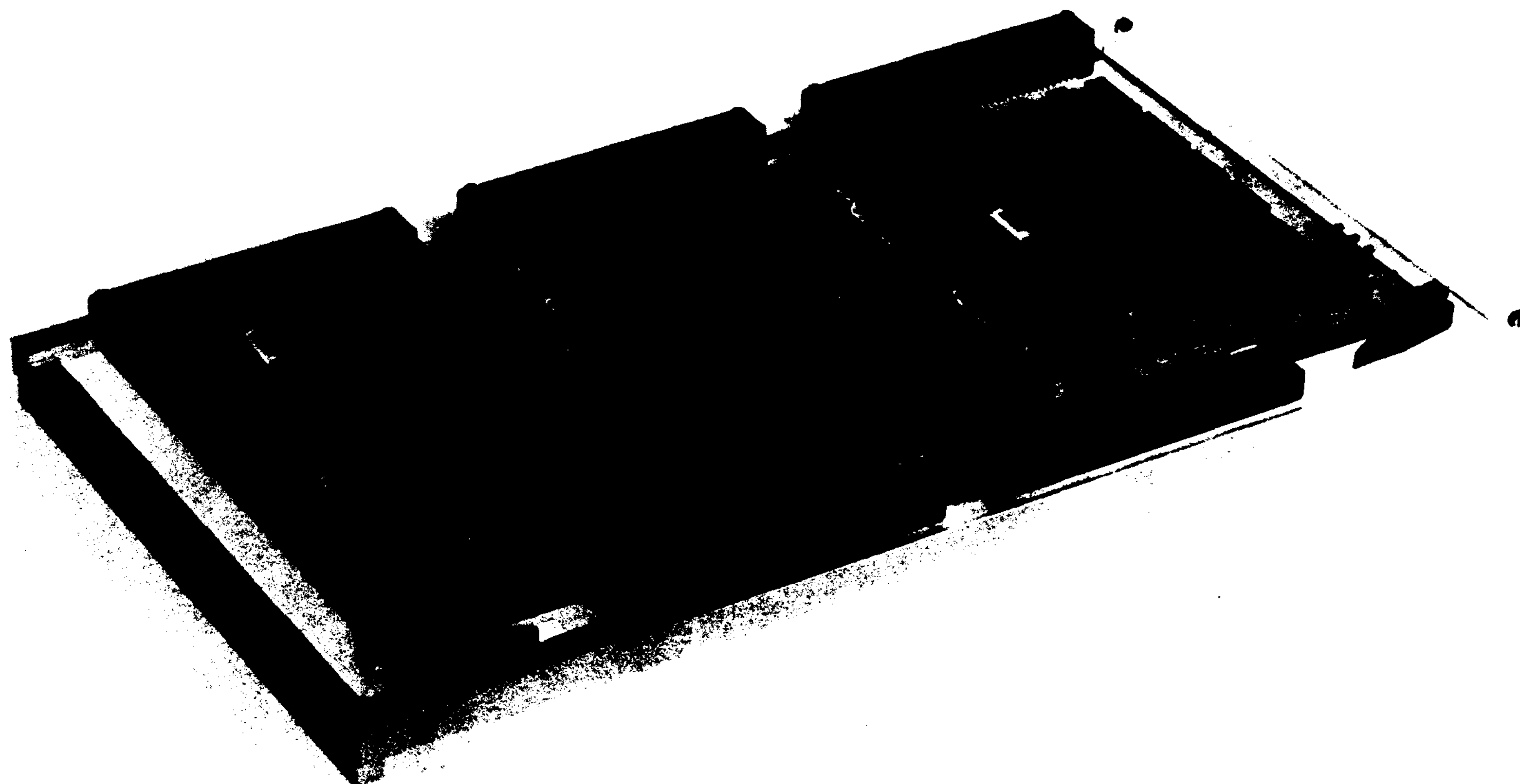
slower operating I/O devices, a request/response system is used to lengthen only those micro-cycles associated with low speed device communications. Thus, the system interface permits slow, moderate and high speed devices to communicate at their own maximum rate.

Performance Characteristics

The salient characteristics of the bit-slice microprocessor based machine shown in Figure 3.7-7 are:

TYPE: General Purpose, Digital Processing Element
Full Parallel Organization
Instruction Stream Pipeline Structure
Sixteen General Purpose Registers
Two Index Registers
Stack Pointer
Pipelined Microprogrammed Control Unit
Bipolar MSI, LSI and Microprocessor Technology
Upwards Software Compatibility With The Prototype PE

ARITHMETIC: Binary, Fixed Point, Fractional
16 Bit Single Precision Data Words
32 Bit Double Precision Data Words
Negative Numbers in 2's Complement Form



SAMSON BRASSBOARD PE
FIGURE 3.7-7

SPEED: 1.27 MIPS
200 nsec micro-cycle time
200 nsec memory cycle time
Throughput Improvement Factor
(vs. predecessor) - Greater than 3.8.

EXECUTION TIMES:	0.20 μsec	Interregister Add
	0.60 μsec	Memory Reference Add
	4.0 μsec	Multiply*

SIZE: One Card (86 microcircuits)

POWER: (Watts at 5 VDC): 15.1 Basic Instruction Set
16.4 Basic and Expanded Instruction Set

3.7.5 PE Instruction Repertoire

The basic instruction set, the instruction set of the prototype PE being emulated, is grouped into six classes: Memory Reference, Interregister, Shift, Skip, Control and Input/Output. All instruction word formats are constructed within the 16 bit word length. Table 3.7-1 specifies the execution times for some of the instructions in the basic instruction repertoire when executed on the bit-slice microprocessor based PE; instructions are listed by group

*Can be reduced with additional hardware to 1.4 μsec.

and mnemonic.

In addition, Table 3.7-1 also specifies the execution times of some of the expanded instructions included in this PE, which presently includes: Double Precision, Saturate Arithmetic, Logical, Stack and Multiple Register Memory Reference classes of instructions.

Appendix A provides a detailed listing of the execution time for all of the instructions.

3.8 SAMSON MEMORY ORGANIZATION

Each individual PE has its own local memory space where the minimal size is application dependent. The individual PEs are compatible with both core and semiconductor memories, including semiconductor ROM and/or RAM (read alterable memory).

In order to maintain maximal throughput, semiconductor RAM memory is utilized as the local RAM memory. In an all RAM configuration, the RAM local memory provides for storage of the operational program, constants, presets, results of intermediate calculations as well as, at least, a link to the executive program. In such a configuration, provisions are made for a hierarchical memory organization; a two-port local memory is utilized with one port connected to a low speed backing (core) memory. Mass storage requirements can be provided by extending the hierarchical structure

TABLE 3.7-1

EXECUTION TIMES (IN μSEC)

Instruction	Time
Memory Reference Instructions	
Add, Subtract, Load, Store, Jump	0.8
Interregister Instructions	
Add, Subtract, And, Or, Logical Complement & Arithmetic Complement	0.2
Multiply	4.0
Divide	7.75 (Av.)
Shift Instructions	
Left, Right, Long, Short, Arithmetic & Logical	0.4 + .2S
Rotate Left Long (Short)	0.4 + .2S
Skip Instructions	
Decrement and Skip if: Zero, Non-Zero Skip If: Greater Than, Greater Than or Equal, Less Than, etc.	0.8
Double Precision Instructions	
Add & Subtract	0.4
Arithmetic Complement	0.6
Multiple Register Instructions	
Load N Registers Store N Registers	1.2 + .2N
S= Number of places shifted,	$0 \leq S \leq 15$
N= Number of registers,	$1 \leq N \leq 16$

so that the backing memory is a two-port memory with one port connected to the slower speed mass memory (disc or drum). This memory hierarchy can be managed utilizing conventional replacement algorithms.

Alternatively, the operational program, constants, presets, and executive link can be stored in a ROM (PROM or EPROM) local memory to assure program integrity while utilizing a RAM local memory for temporary results. Here, the local memory is partitioned into a Scratch Pad (RAM) memory and a (ROM) Program memory. In this configuration, program replacement is managed in a completely different and unique manner. Power switching techniques, which are included to reduce the power dissipation of the Program memory, are utilized to select and/or replace active programs.

It is this latter memory configuration which provides maximum speed and minimal power. Obviously, speed is a primary consideration in a real-time multiprocessor. Perhaps, less obvious, is the need for minimal power. Clearly specific applications, i.e., aircraft, space, etc. require minimal power dissipation. However, in addition to these application requirements there is even a more fundamental need for a low power dissipation local memory. Since the multiprocessor may have a relatively large number (n) of PEs the total power dissipation due to the total memory (assuming equal size PE local memories) is n x local PE memory power.

3.8.1 Local PE ROM Memory

The local PE PROM memory segment is partitioned into multiple pages. (Assume a 16K word, by 16 bit, active ROM local memory implemented with 512 word 4 bit PROM devices.) The memory is partitioned into p pages, where

$$p = \frac{\text{local ROM memory size}}{\text{ROM component size}}$$

(hence, p = 32 pages in this example).

Rather than powering all p pages all of the time, only that page which is actively being addressed is powered. All other PROM pages are thus unpowered and the power dissipation is reduced to 1/p of what it would have been otherwise. (In the example under discussion the power would be 1/32 of an all powered local ROM memory. In addition, the reliability of the local ROM memory is increased as a result of these unpowered pages.) It should be noted that the present semiconductor ROM technology is adequate for developing a PE compatible power switched ROM local memory having a compatible access time. (Assuming w watts/ ROM device, the local ROM memory array power is only 4w).

The power switched local ROM memory can be further partitioned so that multiple application tasks, where each application task can be of the maximum memory address space size, can be accessed directly by each PE. In this configuration, only the active page of the active task memory segment is powered. Through

this power switching mechanism, it is possible for the PE to switch from task to task at a minimal overhead penalty. (Switching tasks takes the same amount of time as addressing a word within another page of the same task. It should be noted that the time to access a word within another page of an active task requires less than the PE required access time.)

The resulting logical address space (for the 32,768 word physical address space) is thus greater than 1 billion words.

3.8.2 Local PE RAM Memory

The local PE RAM memory in a ROM/RAM local memory is a segmented programmable read alterable memory. The segment (or block) size and logical address are both programmable. The local PE RAM utilizes a programmable address mapper to control both the segment size and logical address assignment of the physical RAM memory. This segmented programmable RAM provides maximum flexibility (with regard to task-to-task RAM requirements) while requiring a minimum of physical RAM memory (which not only reduces the total size of the multiprocessor memory system but also minimizes power dissipation). No special requirements (other than compatible access time) are necessary of the semiconductor RAM components utilized in this memory; however, to minimize total memory power it is desirable that these components have a standby mode (where power is reduced while retaining memory content)

as well as the normal operate mode. In addition, it is desirable but not mandatory that the RAM memory be configured as a two-port memory.

The programmable address mapper is programmed into a particular configuration by the executive mapper-loader routine utilizing the configuration data provided by the application program. In the following we will be considering a 1K physical RAM (in a total 16K memory) configured as 32 blocks (segments) with 32 words per block.

The address mapper monitors the addresses on the (higher order) memory bus-bits (starting with M05) which have been selected and subdivides the total (16K) address space into (512) blocks of 32 words each. If a block corresponds to the address space assigned as ROM memory, the address mapper output signal "ROM*" was programmed to the LLO state. If the block corresponds to the address space assigned as RAM, "ROM*" was programmed to the LL1 state and the address mapper addresses and enables the physical RAM. Figure 3.8-1 depicts a 4K ROM/1K RAM and shows the physical partitioning of the RAM into n blocks (n = 32 here). Consider a 16K local PE memory configured from three such modules and one 4K ROM module (depopulated of all RAM). Physical RAM on one module can be assigned by the address mapper to an address space within the same physical module, within the address space of another memory module or within

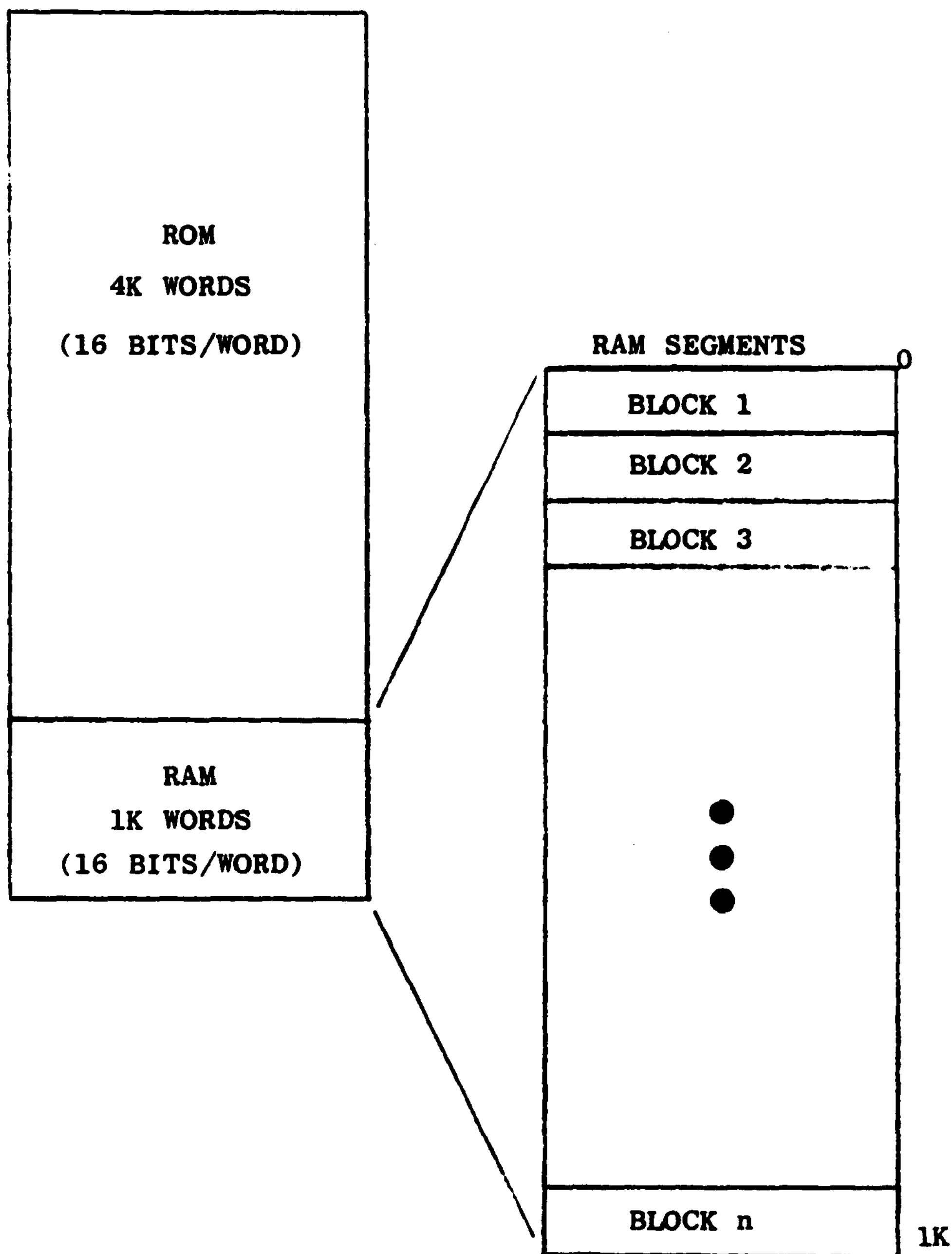


Figure 3.8-1
PHYSICAL MEMORY

the address space of even a non-existing memory module. Figure 3.8-2 shows a typical RAM assignment.

In order to efficiently handle the multilevel vectored interrupt capability of the PE, the local RAM memory has the capability to have alternating ROM words and RAM words. Within an address space assigned as RAM memory ("ROM*" was programmed to LL1), a block is specified as an alternating block by programming (via the executive routine) the mapper output signal "ALT*" to an LL0 state, Figure 3.8-3.

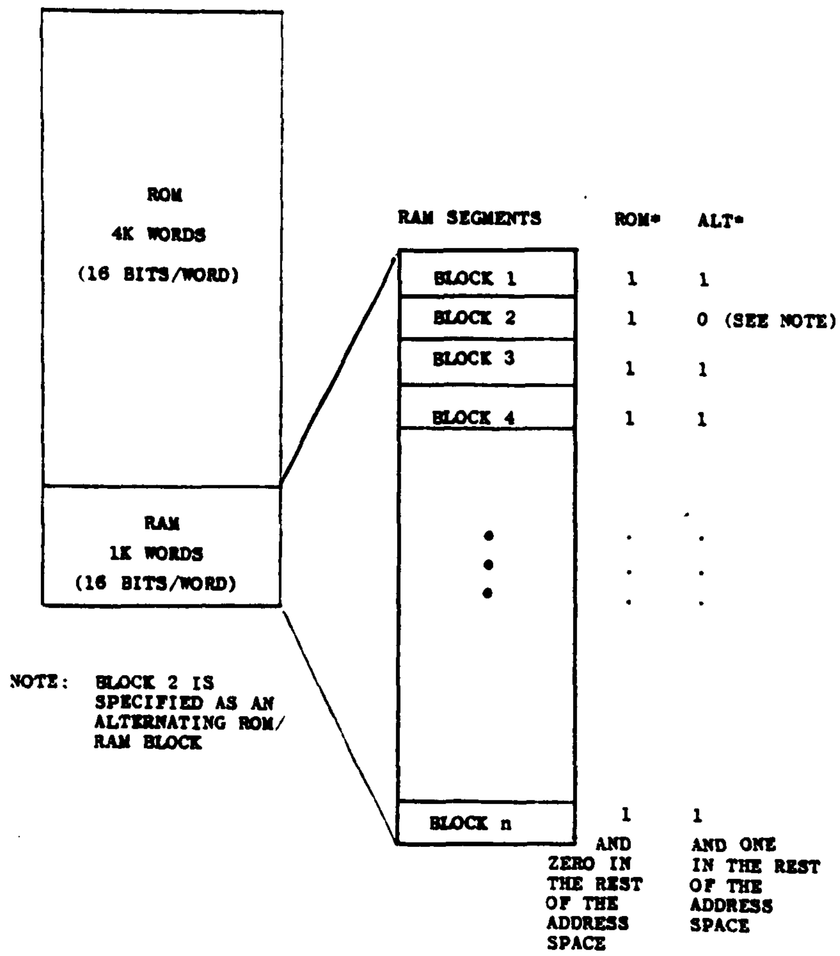
Rather than assigning RAM to the physical address space of ROM and making that physical ROM unaccessible to the PE, the "backing ROM" is assigned a PE accessible address by the address mapper. To utilize the backing ROM a free (unused) block of memory is utilized. Figure 3.8-4 depicts the backing ROM assignment for two different configurations.

Upon completion of a task (or reassignment of a PE to another task) the executive mapper-loader routine would reconfigure the programmable address mapper in accordance with the new configuration data provided by the new application task. If the PE has been temporarily reassigned, temporary reassignment storage allocation is provided for storage of this temporary data. Assuming the new configuration requires 64 blocks of 16 words per block the address mapper would monitor the newly selected memory bus bits (starting

PHYSICAL MEMORY		LOGICAL MEMORY	MEMORY 1 ROM*	MEMORY 2 ROM*	MEMORY 3 ROM
MEMORY CARD 1 ROM AND RAM	ROM 1A	ROM AND RAM	0	0	0
	RAM 1A		1	0	0
	ROM 1B		0	0	0
	RAM 1B		1	0	0
	ROM 1C		0	0	0
MEMORY CARD 2 ROM AND RAM	ROM 2A	ROM AND RAM	0	0	0
	RAM 2A		0	1	0
	ROM 2B		0	0	0
	RAM 2B		0	1	0
	ROM 2C		0	0	0
MEMORY CARD 3 ROM AND RAM	RAM 3A	ROM AND RAM	0	0	1
	ROM 3A		0	0	0
	RAM 3B		0	0	1
	ROM 3B		0	0	0
	RAM 3C		0	0	1
	ROM 3C		0	0	0
(MEMORY CARD 4 IF IT EXISTS, ROM ONLY)	(ROM 4A)	(ROM AND) RAM	0	0	0
	RAM 1D		1	0	0
	(ROM 4B)		0	0	0
	RAM 2C		0	1	0
	(ROM 4C)		0	0	0
	RAM 3D		0	0	1

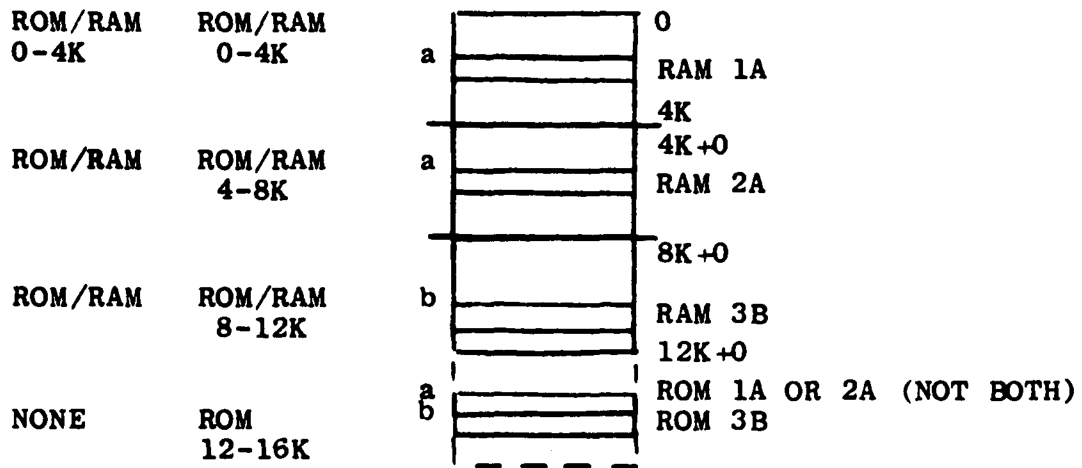
TYPICAL RAM ASSIGNMENT

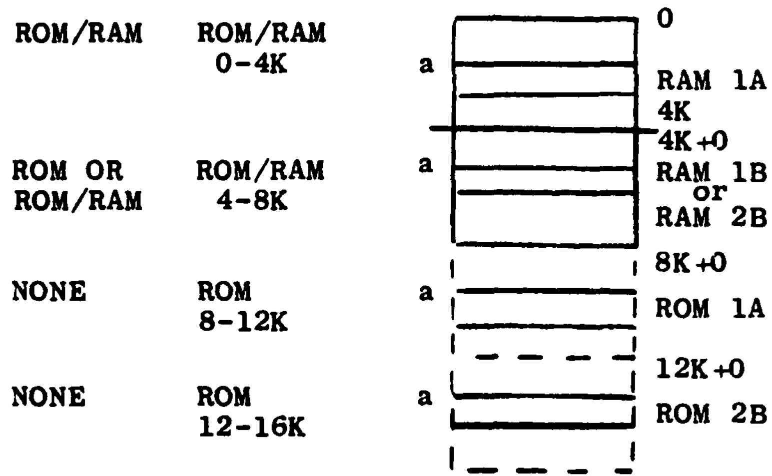
Figure 3.8-2



ALTERNATING ROM/RAM

Figure 3.8-3





BACKING ROM

Figure 3.8-4

with M03).

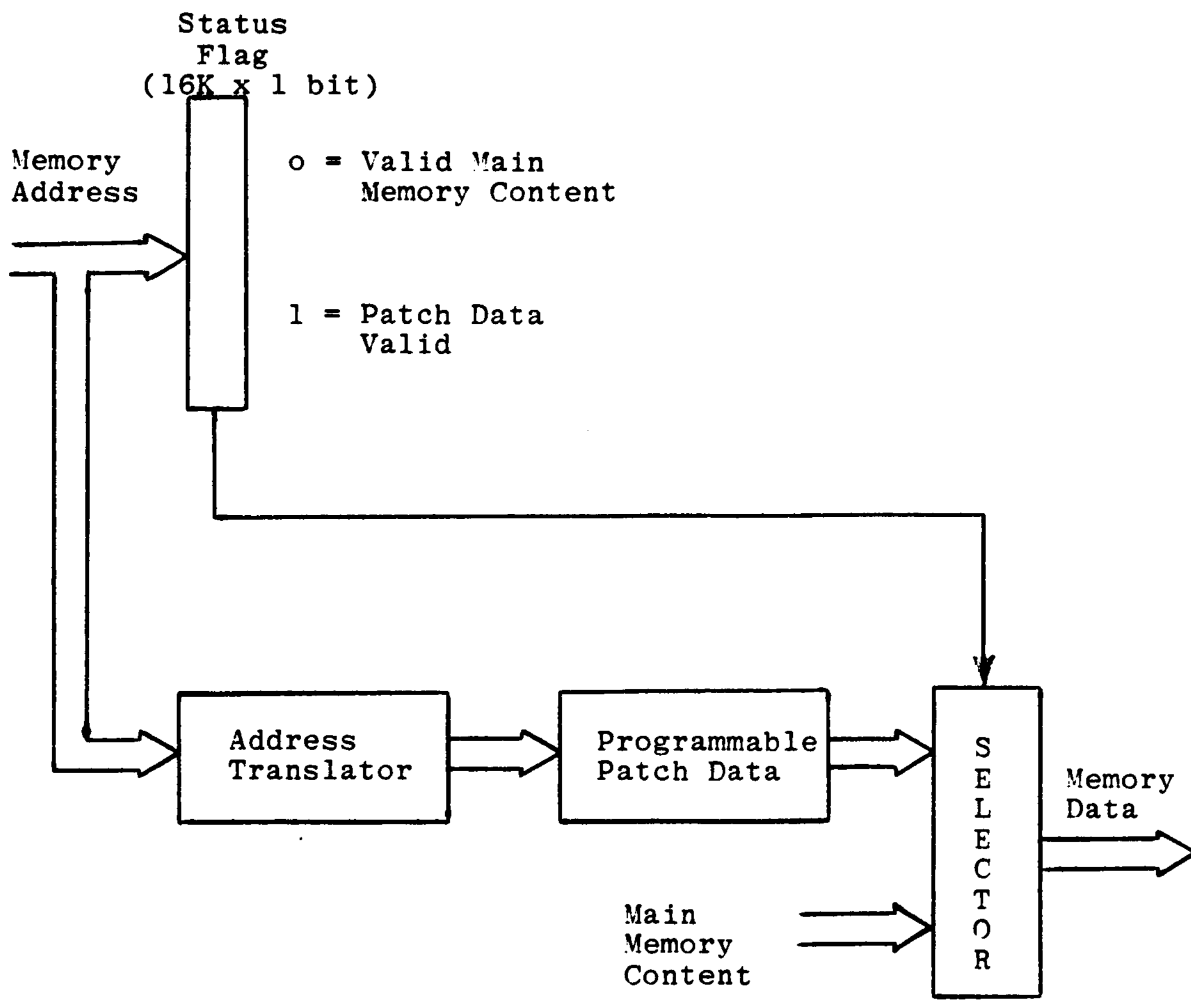
3.8.3 Local PE ROM Patch Organization

Although it is most desirable from a power dissipation point of view to utilize a ROM power switch memory configuration (once the executive routines and application programs have been established) there is one drawback to such a memory configuration. Generic software errors cannot be corrected easily and rapidly once the ROM memory elements have been programmed.

Hence, in order to obtain the best of both (low power as well as protection of program integrity due to the ROM and program flexibility due to the RAM) the PE local memory provides a programmable patch organization which enables modification of the ROM firmware without hardware alteration. The number of words which can be patched as well as the implementation to be utilized is dictated by the software and speed requirements of the specific application.

Independent of the application, the patched memory is programmed under control of the executive routine from either the hierarchical memory or from the external environment through the input/output system. Figure 3.8-5 depicts the local PE ROM patch organization (where the programming data paths have been omitted for clarity).

The memory address provided to the main local memory is also provided to the status flag and address



LOCAL PE PATCH ROM
(FOR A 16K PE MEMORY)
Figure 3.8-5

translator. The status flag, a 1 bit array, is accessed to determine whether the main local memory content is valid or whether the patch data should be utilized in lieu of the main memory content. Concurrently, the address translator converts the memory address to the address of the corresponding patch word while the main memory content is addressed. The status flag controls the selector so that either the main memory content or the corresponding patch word content is transmitted to the PE as the "actual" memory content.

3.9 SAMSON MULTIPROCESSOR ARCHITECTURE

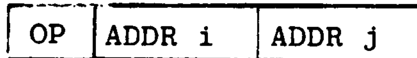
The SAMSON architecture is a flexible and viable organization capable of being configured from multiple identical PEs as well as multiple specialized sets of PEs (where each set of PEs performs a unique function). Furthermore, SAMSON can be configured so that a single common main memory system is utilized or, alternatively, each PE can have its own local memory.

The SAMSON architecture is not restricted to a PE single address instruction format. The single address format has been utilized here due to present technological benefits rather than intrinsic benefits to the SAMSON architecture. Figure 3.9-1 shows single address, double address and triple address formats; all of which are useable within the SAMSON architecture. A single address format is the highest level addressing format which has been used in SAMSON due to the higher speed

Single Address Format

$R_A \leftarrow R_A \circ M_i$

Double Address Format



$M_i \leftarrow M_i \circ M_j$

Triple Address Format

$M_k \leftarrow M_i \circ M_j$

$\circ$ denotes operation specified by OP (opcode)

ADDRESSING FORMATS

Figure 3.9-1

available through register manipulations (with the present state of the technology) and due to the resulting smaller word format required. Furthermore, this smaller word format is adequate for the majority of today's real time computing applications.

The SAMSON multiprocessor utilizes identical PEs; however, specialized PEs can be added to speed up particular operations should a specific real time application utilize these (slower) operators frequently.

As already indicated, the memory organization can be that of a single common main memory or multiple local memories. Furthermore, if a local memory configuration is utilized, a hierarchical memory organization is applicable. In addition, the local memory can be either contiguous or partitioned (i.e., consisting of a separate program and data memory), in which case program loading would be accomplished utilizing privileged instructions.

3.9.1 SAMSON Overview

As an instruction is peeled off of the memory, in a system configured with a common main memory, the SAMSON system dynamically schedules execution of this instruction by:

1) analyzing the op code and determining if a free PE capable of executing this instruction is available (if all PEs are identical the opcode need not be examined),

2) starting a fetch from the (data) memory assuming that the source operand(s) are not in conflict with any other instructions, either executing or waiting to be scheduled for execution in the Conflict Retry Buffer.
3) verifying this assumption simultaneously through the Parallel Execution Monitor (which identifies any conflicts), and
4) storing the instruction in the instruction queue from which the assigned PE will execute this instruction.

Alternatively, in a SAMSON multiprocessor system configured with PE local memories. the preconditioned instructions are preassigned and prescheduled. Execution here proceeds based upon the conflict free conditions as determined by the distributed Parallel Execution Monitor.

In the following sections the various components of the SAMSON multiprocessor are described in detail.

3.9.2 Instruction Queue Preprocessor

Either an interleaved fetching scheme or a pipelined fetching scheme can be utilized to form an instruction queue. The fetching scheme utilized determines the complexity and speed of the Instruction Queue Preprocessor (Fetching Unit). In a minimal configuration utilizing a pipeline fetch strategy the Instruction Queue Preprocessor would be distributed among each of the PEs; each would have a conventional memory

access controller, provided the memory cycle and access times are adequate to maintain the desired throughput. Alternatively, an interleaved strategy would be utilized with a common centralized main memory system and would require two or more memory units as well as a more complex memory access controller to manage this memory system. In such a configuration the memory access controller must maintain multiple queues (or stacks) of instructions, control allocation of fetched instructions to the appropriate queues as well as control access to the multiple memory units. In addition, in such a configuration the Instruction Queue Preprocessor must have its own replica of the program counter with incrementing and loading capability. Furthermore, this preprocessor must be able to interpret both the op code and address fields as well as having replicas of the index and base registers for calculating the effective memory address for branching, subroutining and operand fetching. The Instruction Queue Preprocessor must also monitor PE program execution progress to prevent exceeding the Instruction Queue capacity and to be able to handle interrupt calls.

It should be noted that the Instruction Queue Preprocessor does not provide the individual PEs with the raw operand address via the Instruction Queue. Instead, the Instruction Queue Preprocessor calculates the effective address,fetches and loads the appropriate operand into the Instruction Queue. If the PE being utilized has Vector assembly instructions, then the Instruction Queue Preprocessor would decompose the Vector assembly

instructions which it would provide to the Instruction Queue. Once started on the Vector assembly instruction there can be no interference with the index registers and memory locations utilized here by any other instructions. This necessitates inclusion of Test & Lock instructions which are executable by the Instruction Queue Preprocessor. Furthermore, instruction prefetching would be inhibited until the vector sequence approaches termination. The Test & Lock class of instructions is not the only instruction which is executable by the Instruction Queue Preprocessor. In addition, this unit is capable of executing branch instructions (both conditional and unconditional). Hence, this preprocessor unit will not pass on to the Instruction Queue the branch instructions which it executes. Whenever a branch is taken, the queue will simply be filled starting at the instruction just branched to. (Any unneeded instructions, past a branch instruction, in the Instruction Queue are simply flushed from the queue by resetting the queue pointer.)

When the SAMSON multiprocessor utilizes a pipeline fetch strategy, the Instruction Queue Preprocessor can be distributed among the PEs of the network. Here, the local PE program memories are preconditioned (preloaded and preassigned) with the appropriate program instructions or segments which would otherwise be dynamically assigned to the Instruction Queue at execution time by an interleaved Instruction Queue Preprocessor. This assignment would be made by the assembler software

at final assembly; hence, no speed or programming penalty is paid by the application user. Since high speed semiconductor memories having compatible memory cycle and memory access times are utilized, the Instruction Queue Preprocessor is all but non-existent (deteriorating into the conventional PE local memory controller).

3.9.3 Instruction Assignment Controller Option

In a SAMSON multiprocessor configured with an interleaved Instruction Queue Preprocessor utilizing dissimilar PEs (not all PEs being identical), the Instruction Assignment Controller would receive the preconditioned instructions (from the Instruction Queue Preprocessor fetching unit via the Instruction Queue) and determine which of the suitable processing elements is available for executing this instruction. The Instruction Assignment Controller simultaneously sends the fetched source operands and destination address to the Conflict Retry Monitor. (With dissimilar PEs, the processing elements might be divided into an arithmetic and logic (ALU) PE, multiply/divide PE, I/O PE, etc.)

With the pipeline fetch strategy described above being utilized by the SAMSON multiprocessor the Instruction Assignment Controller is not required in this SAMSON multiprocessor configuration. Similarly, with an interleaved strategy and identical PEs the Instruction Assignment Controller function is reduced to maintaining the free PE list.

Figure 3.9-2 depicts the SAMSON multiprocessor configuration utilizing a common main memory, an interleaved Instruction Queue Preprocessor, an Instruction Assignment Controller and three types of PEs (ALU PEs, MPY/DIV PEs and I/O PEs).

3.9.4 Parallel Execution Monitor

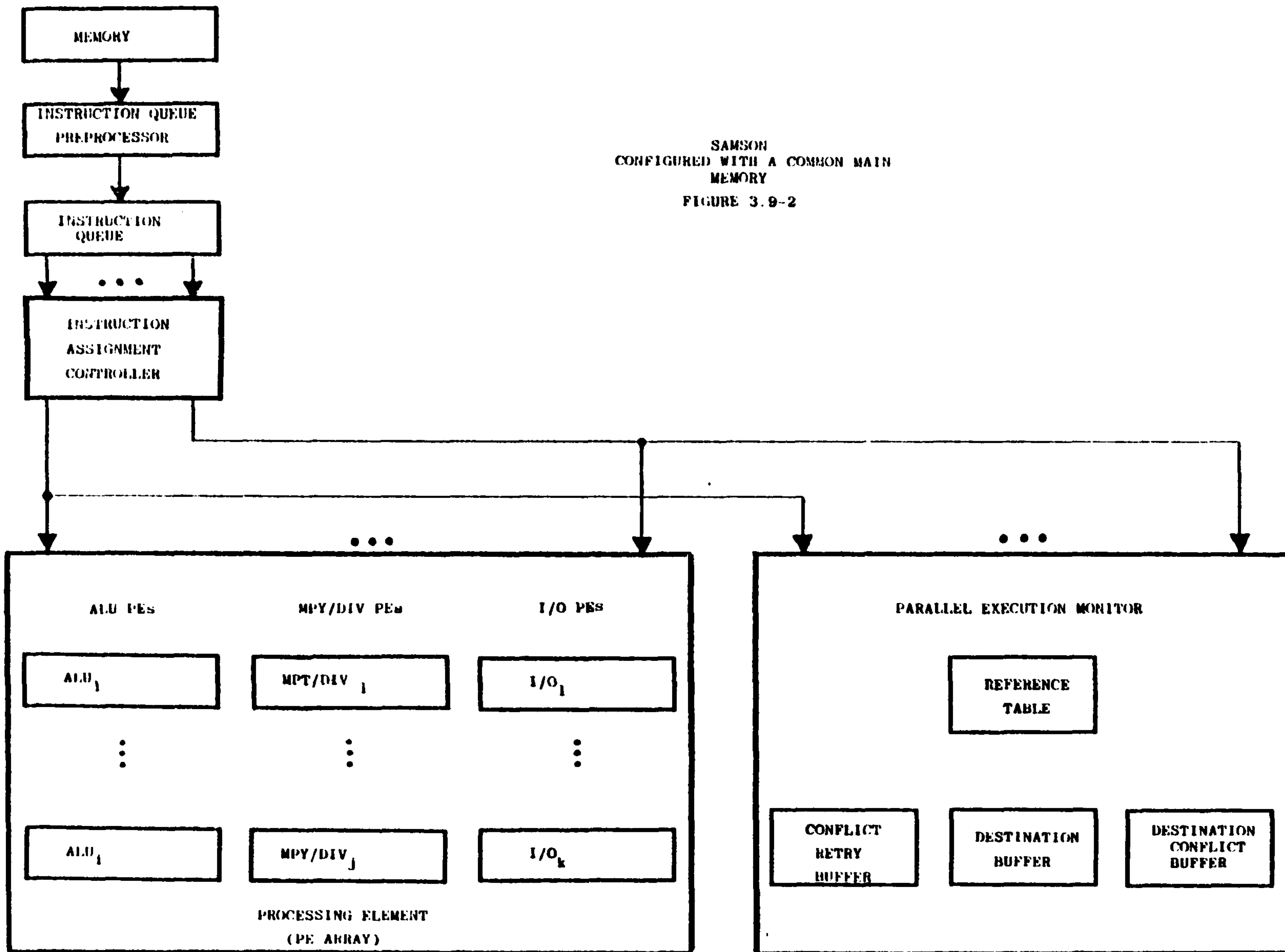
As indicated by Definition 3.7 parallel executable instructions are only permitted to share common source operands. Utilizing this definition it is possible to ascertain whether two intructions are in conflict. A conflict occurs whenever

(1) two instructions try to start execution and one (or both) of either of their operands corresponds to the result of the other, or if they both have the same destination, or

(2) an instruction trying to start execution has as one (or both) of its operands the result of an earlier instruction which has not completed execution yet, or its destination is the same as one still in the process of being executed.

The two conflicting instructions are said to be dependent, as described in Section 3.3.3 (the formulation of statement independence).

The Parallel Execution Monitor detects parallel executable instructions through the use of a



SAMSON
CONFIGURED WITH A COMMON MAIN MEMORY

FIGURE 3.9-2

hardware reference table [3.10]. The elements of the reference table $RT(X,S_i)$ are the variable names, X, and the statement labels, S_i. The entries of the reference table would be one of the following four values:

$$RT(X,S_i) = \begin{cases} 00 \text{ if X is not referenced in } S_i \\ 01 \text{ if X is read in } S_i \\ 10 \text{ if X is updated in } S_i \\ 11 \text{ if X is read and updated in } S_i \end{cases}$$

With this reference table information, the direct precedence relation between statements can be determined and hence concurrent execution of multiple instructions can proceed (and still preserve the task determinacy).

If a conflict is detected, the instruction trying to start execution will be temporarily stored in the Conflict Retry Buffer. Similarly, if the Instruction Assignment Controller cannot find an appropriate PE to assign an instruction to (for execution), then this instruction will be temporarily stored in the Conflict Retry Buffer. This buffer is capable of storing both operand addresses (of the conflicting operand) as well as operands (not causing the conflict).

Those instructions which are blocked are rescheduled when a PE becomes free, the source operand(s) are available and/or the destination address for a new datum of the same variable is available. When these conditions are satisfied the Instruction Assignment

Controller is notified that the present, ready-to-run instruction is no longer blocked and should be assigned to the appropriate PE for execution.

If an instruction can be executed, i.e., there is no conflict and an available PE exists, the executing PE receives the source operands and the opcode. The PE utilizes these source operand(s) and opcode to perform the designated operation under its own local control. Meanwhile, the Destination Buffer receives the destination register number or the destination effective address (if a three address format is utilized). The Destination Buffer is partitioned so that each word of this buffer corresponds to its companion PE.

Upon PE completion of an active instruction, the Parallel Execution Monitor determines whether the computed result can be stored into its destination location. If no conflict exists, the PE stores the result in the destination location; otherwise, the result is placed in the Destination Conflict Buffer. Assuming no destination conflict exists, the Parallel Execution Monitor attempts to retry conflicting instructions which have been waiting for this just-computed operand.

Those instructions which are non-conflicting can execute in any arbitrary order and indeed are executed in parallel by making use of the multiple execution resources (PEs) of the SAMSON multiprocessor.

Simple arithmetic and logical instructions are executable in 200 nanoseconds while the longest instructions (Multiply and Divide) presently require 4 and 7.75 (average) microseconds. (As previously indicated, these longer instruction times can be reduced dramatically with additional hardware; this was not done at this time in order to minimize research hardware costs.) Furthermore, there are no (or relatively few) unneeded memory references due to the multiple register architecture of the SAMSON PEs. In addition, the "multiple register load" and "multiple register store" instructions minimize execution overhead (which takes such a large fraction of the program execution time for initialization, loading of operands, and storing of results) associated with the expression or subroutine.

3.9.5 Conflict Retry Algorithm

The Parallel Execution Monitor utilizes the reference table described above to determine whether an instruction should be blocked (maintained on the Conflict Retry Buffer) or whether it has become ready (and should be moved to the Instruction Queue) so that it will be scheduled for execution.

Assuming a three address instruction format, the Parallel Execution Monitor would generate a reference table consisting of three entries per statement label, e.g. RT (X, Y, Z, S_i). Consider the case where the operation (op) specified by S_i is op = add; then S_i would read:

$$X \text{ op } Y \rightarrow Z \quad \text{or } X + Y \rightarrow Z$$

and the reference table information for these variables would be: X = 01, Y = 01, and Z = 10.

Consider instruction S_j, as the only other instruction, with variables X, Y and A. Let op = subtract (X op Y $\rightarrow$ A). Then the reference table for S_j would read: X = 01, Y = 01 and A = 10. Examination of the reference table indicates that the Conflict Retry Table need not be referred to in order to determine whether S_j can begin execution. To facilitate this look up operation, the reference table should be implemented with an associative memory. Thus one copy of the reference table (RT1) would be addressed with X, a second copy (RT2) would be addressed with Y, and finally the third copy (RT3) would be addressed with A. (With both single and double addressing formats only two copies of the reference table must be maintained.) RT1 would indicate that X is available, RT2 would indicate that Y is available and RT3 would indicate that it is available as a destination since it was not found in RT3.

It should be noted, that an instruction is started as soon as its source operands are available, independent of any conflict which may exist with regard to its destination. The result thus obtained is tagged with an iteration label and placed in the Destination Conflict Buffer. The iteration label is utilized to determine when the destination location should be updated to this new value. A simple tagging mechanism readily determines "when" this update should be performed; this tagging mechanism assigns an iteration number to all intervening instructions between the first and second update of this destination operand. Upon completion of the execution of all tagged instructions having this iteration number, the destination location would be updated to the newer destination value.

As instructions begin execution the reference tables are updated to reflect the present status of the executing instructions; new instructions are added and tagged as "execution". Likewise, as instructions complete execution the corresponding reference table entries are updated to delete the presently completed instruction. Blocked instructions are maintained on these reference tables but are tagged as being blocked. If S_k had Z, B and C as its variables, its reference table data might read: Z = 01, B = 01, C = 10. While S_i is blocked and/or executing, S_k will remain blocked since Z of S_i equals 10 (updated in S_i). While S_k is blocked, S_k will be placed in the

Conflict Retry Buffer. Upon completion of S_i, S_k will become ready and will move to the Instruction Queue. Figure 3.9-3 depicts the reference table data and Conflict Retry Buffer content when S_i and S_j are executing and S_k is blocked. Figure 3.9-4 shows a typical reference table.

The Conflict Retry Buffer is a FIFO (first in first out) buffer which maintains blocked instructions in the same chronological order in which they were received, thus preserving the task determinacy of two instructions having the same destination address.

These features can be centralized, as in the above discussion, (which has been done for the purpose of clarity) or they can be distributed among the PEs of the array. Since the fault tolerant characteristics of SAMSON are improved by distributing these features among the PEs, the distributed configuration should be utilized in those real time applications where fault survivability are critical.

Due to the importance of fault tolerance in any multiple processor configuration, special consideration has been given to this topic with regard to both the individual SAMSON PEs and the distributed fault tolerant SAMSON network. Refer to Sections 5 and 6, respectively, for a detailed treatment of these topics.

REFERENCE TABLE DATA
(THREE ADDRESS INSTRUCTION FORMAT)

							2 bit Status
S_i :	X	01	Y	01	Z	10	Executing (11)
S_j :	X	01	Y	01	A	10	Executing (11)
S_k :	Z	01	B	01	C	10	Blocked (01)

CONFLICT RETRY BUFFER

	1st Operand	Conflict Tag	2nd Operand	Conflict Tag	Destination Address	Conflict Tag	
S_k	Z	Yes	B	No	C	No	1st Blocked Instruction
-	-	-	-	-	-	-	
. . .							
-	-	-	-	-	-	-	nth Blocked Instruction

B would be the actual operand value while Z would be the operand address (denoted by the conflict tag). When S_i completes and stores the computed variable at Z, this value is copied into Z of S_k and the conflict tag is updated; at this time the reference table status for S_k would be updated to Ready (10).

REFERENCE TABLE & CONFLICT RETRY BUFFER
FIGURE 3.9-3

	S_1	S_2	S_3	S_4	S_5	S_6
A	10	00	00	00	00	00
I	01	01	11	00	00	01
J	01	00	00	11	01	01
X	00	11	00	00	01	00
Y	00	00	00	00	10	00
Z	00	00	00	00	00	10

Program Flow

S_1 : A = I + J
S_2 : X = X + I
S_3 : I = I - 1
S_4 : J = J - 1
S_5 : Y = J * X
S_6 : Z = I * J

Reference Table Rules

$$RT(x, S_i) = \begin{cases} 00 & \text{if } x \text{ not reference} \\ 01 & \text{if } x \text{ read} \\ 10 & \text{if } x \text{ updated} \\ 11 & \text{if } x \text{ read \& updated} \end{cases}$$

Note: The reference table is searched associatively using the variable name; therefore the first index is the variable name.

TYPICAL REFERENCE TABLE
Figure 3.9-4

3.9.6 Dedicated SAMSON Configuration

This dedicated configuration consists of multiple stored program computers (composed of a PE and local PE memory) each with a fixed assigned task and interconnected with appropriate data and control paths. Such a system is desirable in applications where:

1. The tasks are predetermined (as in real time control applications as compared with generalized batch processing),
2. operational speed is required which is greater than that which is obtainable from a single processor implementation,

3a. the program is partitioned such that multiple PEs can simultaneously perform individual subtasks, requiring only moderate intercomputer communications (and communication rates), or

3b. the algorithms utilized for individual tasks have been written for parallel processing execution.

In this configuration, instructions are preassigned to the various local PE memories so as to produce the same execution sequence as that which would be dynamically scheduled by the system previously described. Consequently, the Instruction Queue Preprocessor, Instruction Queue and Instruction Assignment Controller are eliminated from this configuration resulting in a minimal real time high speed multiprocessor suitable for aerospace applications where these features, as

well as minimal size and minimal power, are critical. Furthermore, in this configuration the Parallel Execution Monitor could be distributed among the PEs to provide fault tolerant characteristics otherwise not achievable.

3.9.7 Input/Output

The input/output requirements are significantly different for generalized applications and specific applications. Consequently, primary emphasis in this research effort has focused on the computational requirements of SAMSON rather than specific details of a specific I/O application. Furthermore, the communication mechanism to be utilized by the I/O is also highly application dependent. Some insight into this area can be gained from Section 4 which covers the performance of various interconnection schemes.

Despite these differing requirements, there is one common thread to all these I/O requirements. Although the system may have to interface with many I/O devices, only a limited number of the devices are actually communicated with during any computational interval. Consequently, utilization of an I/O bus structure is recommended since this reduces the size of the interconnection network without limiting the throughput of the network, assuming bus saturation does not occur. In general, bus saturation will not occur due to the limited number of actual I/O operations as compared with the amount of actual computation being performed. (It should be remembered that

SAMSON is primarily intended for computation bound problems.)

If utilization of a bus structure should result in bus saturation, dedicated direct memory access channels or dedicated input/output processors could be added to the network. The input/output PEs would receive and transmit data to and from the calculating PEs through memory block transfers.

3.10 SAMSON PERFORMANCE

The goal of the SAMSON system, and for that matter of all multiprocessor systems, is to maximize the number of times multiple instructions can be executed concurrently.

Several alternative techniques exist for determining the degree to which the system approaches this goal. These alternatives include:

1. Modeling the system; the model is then utilized to simulate the system performance utilizing actual data or to statistically evaluate its performance; or
2. Utilize the system itself and measure the actual system performance.

Here, the second performance evaluation technique was used to determine the degree of potential parallelism existing within uniprocessor programs; these uniprocessor programs were written as sequential programs and are not organized for parallel instruction execution. Obviously, programs written for execution on a multiprocessing sys-

tem would utilize considerably less execution time. Similarly, the system resources would be utilized much more when executing programs written for concurrent execution (as compared with the utilization when executing sequential uniprocessor programs).

In addition to the performance evaluation of these sequential uniprocessor programs, the performance (computation time) of the SAMSON network for the webs previously discussed were analytically determined and compared with the corresponding uniprocessor performance. Finally, SAMSON's computation time for the polynomial web of Figure 3.4-10 is compared with the results obtainable with the Folding Method, Tree Method, Estrin's Method and k-th order Horner's Rule.

The performance evaluation technique used on the uniprocessor programs provides actual performance measurements, whereas the results that would be obtained with the modeling technique would: (1) only be as good as the model, and (2) requires various assumptions regarding the system in order to formulate either statistical data or the required analytical equations. (See Section 4 of this work for a discussion of the parameters which must be accounted for in such a model. Skinner and Asher [3-27] proposed a discrete Markov chain model for a multiprocessor system where the processing and rewrite times were equal. Their analysis was performed under the restriction that the number of processors was less than or equal to two. However, for larger systems the complexity of the problem deterred them from further pursuit of an

analytic solution [3-28].)

The results obtained from these programs can be used as a general approximation of the potential improvement to be gained when sequential programs are executed on the SAMSON system. Furthermore, the improvement factor obtained here can be interpreted as an approximation of the absolute lower bound in the improvement that should be expected when such programs are written for concurrent execution on the SAMSON system.

Three real time uniprocessor programs were traced, utilizing the SAMSON PE, and analyzed to measure SAMSON's performance (on these uniprocessor programs). In determining SAMSON's performance our concern is only with the degree of potential parallelism that exists and not the function of the actual program. However, a brief explanation of the real time programs used, as well as an explanation of the real time simulation procedure utilized, is given here for completeness.

A 4K word strapdown inertial guidance program utilizing real input data was traced. The guidance program inputs data via I/O instructions; here they were simulated using the trace routine by means of an input data request message. In response, the data is provided by typing in the actual data values. In addition, this guidance program was periodically required to service an external interrupt. To simulate this operation the actual program code was modified slightly. Only one word in the actual program has to be modified per interrupt handler. The

modified line of code caused an interrupt service message to be printed. In actual operation, the interrupt service routine would be reached via a vector interrupt mechanism. For purposes of simulation, the interrupt service routine was manually selected in response to the interrupt service message.

Six segments of the resulting trace were examined to determine the degree of potential parallelism that exists in this uniprocessor program (written as a sequential program without being organized for concurrent execution).

Similarly, a major segment of a 12K word multilevel real time Omega Navigation program, utilizing data received from four stations, was traced and analyzed. This segment is associated with acquiring, synchronizing and locking onto the omega transmission pattern. This is done by utilizing pattern matching cross correlation techniques to determine the starting time of the omega signal. The program segment traced requires 1.5 seconds of real time to perform this task. This program was also written for uniprocessor execution (without being organized for concurrent execution).

Finally, a 1K word real time diagnostic program (also written for uniprocessor execution, without being organized for concurrent execution) was completely traced and analyzed.

Throughout this performance evaluation, a unit instruction

execution time is adopted for purposes of comparison; the only exception is in the evaluation of the weighted web in Figure 3.4-8 where the multiplication time is three times that of the addition time.

Tables B-1 through B-3 of Appendix B tabulate the normalized execution time required for the real time Strapdown Guidance Program, Omega Navigation Program and Diagnostic Program, using several different number of PEs. The normalized execution time is normalized with respect to the uniprocessor execution time. Tables B-4 through B-6 of Appendix B tabulate the corresponding hardware utilization of each individual PE as well as the total hardware utilization for these three programs (when executed with these various number of PEs).

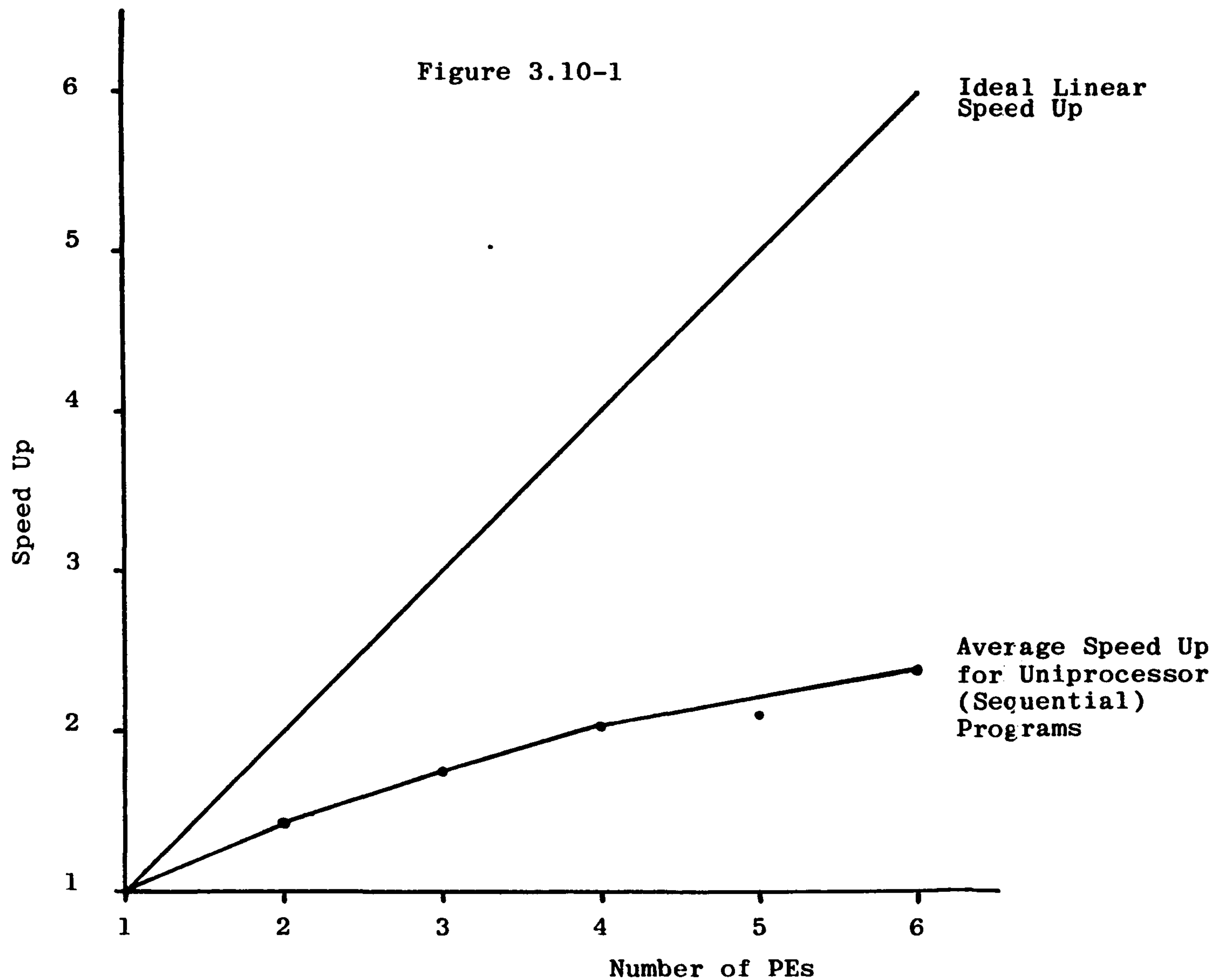
The resulting 25 sets of normalized execution times, versus the number of PEs utilized, are averaged and presented in Table 3.10-1 to find the norm, i.e. the expected overall system performance. The results obtained are as expected, i.e. the overall execution time decreases as the number of PEs increases. Figure 3.10-1 graphically compares these results with the ideal speed up. In Table 3.10-2 the average individual percentage PE utilization and overall hardware utilization, versus the number of PEs utilized, is tabulated. Hereto, the general trend of these results are as expected; the utilization factor decreases as the number of PEs increase. Figure 3.10-2 graphically compares execution time and utilization of the average uniprocessor performance vs the ideal performance.

Table 3.10-1
Average Normalized Execution Time
versus the Number of PEs

Number of PEs	Average Normalized Execution Time
1	1.00
2	.65
3	.57
4	.49
5	.48
6	.42

Table 3.10-2
Average Percentage Utilization
versus the Number of PEs

Number	Average % Individual PE Utilization								Average % Total
of PEs	1	2	3	4	5	6	7	8	Utilization
1	100	-	-	-	-	-	-	-	100
2	100	48.5	-	-	-	-	-	-	74.4
3	100	53	26.6	-	-	-	-	-	59.6
4	100	58.9	30.2	19	-	-	-	-	52
5	100	49.3	30.1	18.3	14.1	-	-	-	42.3
6	100	49.6	35.4	31.8	28.4	28.4	-	-	45.4
7	100	37.5	31	25.5	40.5	18	6.5	-	33.5
8	100	42	32	21	5	5	5	5	27



Linear speed up is the Ideal performance; it is difficult (if not impossible) to achieve due to the need for cooperation and the associated overhead.

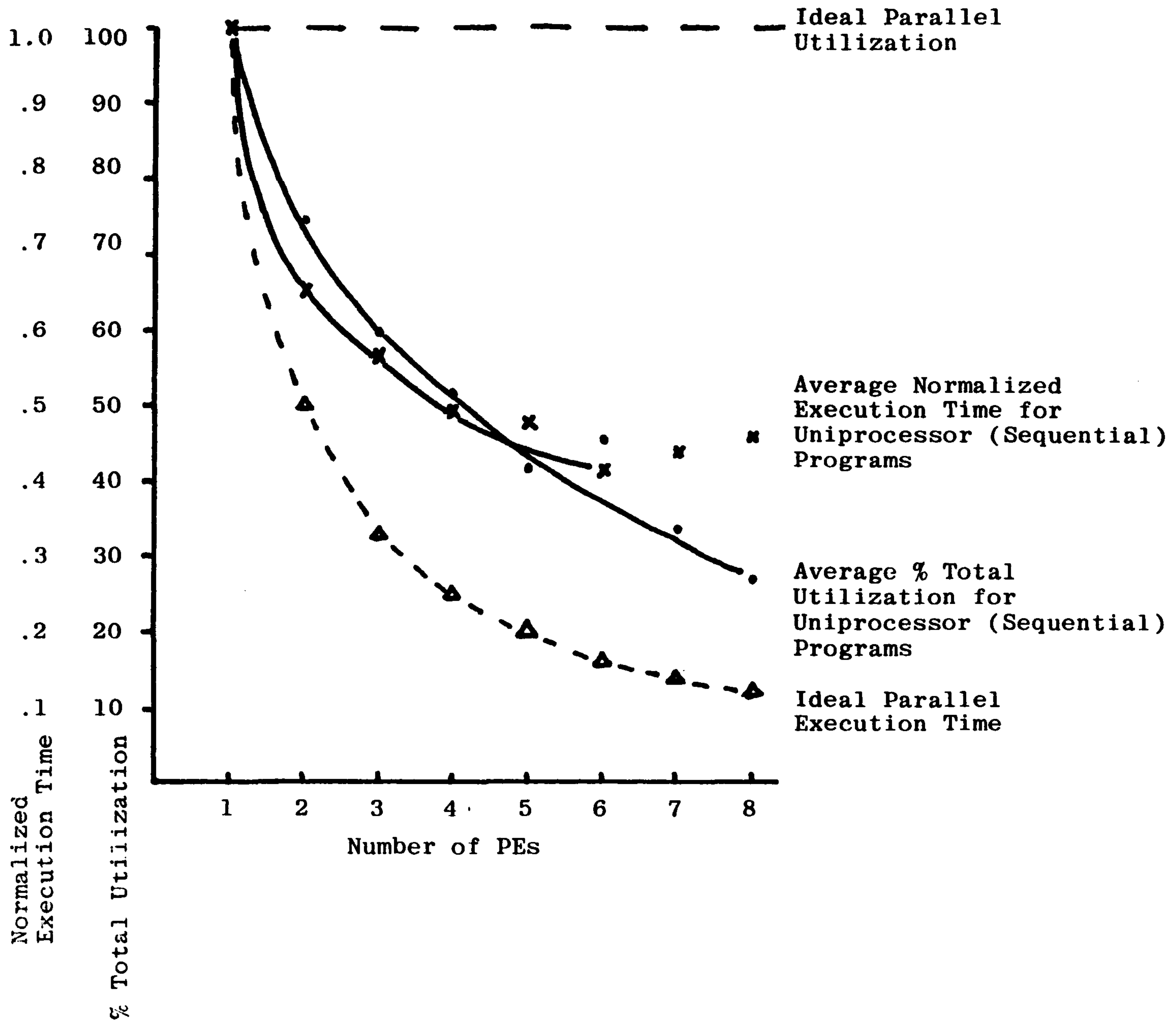


Figure 3.10.2
Execution Time and Utilization
Average Uniprocessor Performance vs Ideal Performance

In Table B-7 the accumulator usage is tabulated for the three programs analyzed; Table 3.10-3 is the average of these programs. Utilization of A0 is maximum; this is to be expected since this register performs three functions; i.e. general register for interregister operations, memory reference register and relative addressing register. Accumulator A1, like A0 also performs three functions; however, its utilization is lower than either A2 or A3 (which perform only two functions, general resistor for interregister operations and memory reference registers). The apparent explanation for this is: most programmers of uniprocessor systems utilize specific resources for specific software functions; here A0 was used extensively for relative addressing. Accumulators A2 and A3 were used more than A1 so that A1 could be kept available to resolve relative addressing bottlenecks, should they occur. The remaining utilization factors are as expected. (The increasing usage of the higher order numbered accumulators is apparently due to a "safety" phenomena, i.e. keeping the working registers away from temporary storage registers.)

The significance of this data results from the high usage of A0 through A3 and especially A0 (and A1). From this data it can be concluded that the potential for concurrent processing would be increased if all registers were memory reference registers. Furthermore, one can conclude that the relative addressing mode utilized in these programs would inhibit dynamic scheduling of the SAMSON system.

Table 3.10-3
Average Percentage
Accumulator Usage

Acc. No.	% Usage
A0	29
1	15.4
2	18
3	19.4
4	4
5	2.1
6	2.3
7	1.9
8	1
9	0.5
10	1.1
11	1.1
12	0.4
13	0.9
14	1.6
A15	1.2

It should be noted that the trace of the diagnostic program revealed that it could be executed as 18 independent tasks on the SAMSON system. These 18 tasks would then be executed in 13.6% of the original execution time. Furthermore, only eight (8) PEs are required to obtain this speed improvement. Figure 3.10-3 illustrates both the 635% speed improvement as well as the high hardware utilization obtained; seven of the eight PEs are utilized more than 90% of the time.

Table 3.10-4 lists the performance improvement obtained with the SAMSON system for the web computations depicted in Figures 3.4-1, 3.4-3, 3.4-4, 3.4-5, 3.4-7 and 3.4-8. The mean improvement obtained for this restricted set of web computations is a 121.4% speed improvement; the average speed improvement is 110%. The maximum improvement was obtained for the web computations X^{16} and $p^7(X)$ where the speed improvements were 275% and 280%, respectively; these results assume no other computation is being performed. Figure 3.10-4 illustrates the speed improvement resulting from executing two computations concurrently. The speed improvement obtained is 580%.

Figure 3.10-5 is a comparison of five concurrent polynomial computation schemes and indicates the improvement obtained with the web structure.

Figure 3.10-3

Speed Improvement
for Diagnostic Program

Sequential Execution

Task	Exec. Time
1	26
2	55
3	26
4	14
5	71
6	16
7	28
8	45
9	34
10	14
11	47
12	42
13	56
14	28
15	41
16	27
17	33
18	95

Total Exec Time = 698

Concurrent Execution
(Task (T) & Execution Time (E) given per PE)

	PE_1		PE_2		PE_3		PE_4		PE_5		PE_6		PE_7		PE_8	
	T	E	T	E	T	E	T	E	T	E	T	E	T	E	T	E
	18	95	2	55	1	26	8	45	9	34	10	14	5	71	7	28
			3	26	14	28	11	47	13	56	12	42	6	16	16	27
			4	14	15	41					17	33				
Total Exec Time:	95		95		95		92		90		89		87		55	
% Utilized:	100		100		100		97		95		94		92		58	

$$\text{Speed Improvement} = \frac{698-95}{95} = 635\%$$

Table 3.10-4

SAMSON Performance Improvement

(SAMSON vs. Uniprocessor)

Computation	Performance Improvement
$\sum_{i=1}^{9} a_i$	100%
X^{11}	25%
X^2	—
X^3	—
X^4	50%
X^5	33%
X^6	67%
X^7	100%
X^8	33%
X^9	100%
X^{10}	125%
X^{11}	150%
X^{12}	175%
X^{13}	200%
X^{14}	225%
X^{15}	250%
X^{16}	275%
X^{18}, X^{24} and X^{27}	120%
$P^7(X)$	280%
$P^4(X)$ $\omega_* = 3$, $\omega_+ = 1$	120%

Figure 3.4-6 x^2 through x^{16}									
#PEs	n								
	0	x							
[illegible]	1	x^2							
[illegible]	2	x^4				x^3			
[illegible]	3	x^8		x^7		x^6		x^5	
10	4	x^{16}	x^{15}	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^9
1	5								

Figure 3.4-7 $P^7(x)$

$a_7 \quad a_6 \quad a_5 \quad a_4 \quad a_3 \quad a_2 \quad a_1 \quad a_0 \quad x$

$a_6x \quad a_4x \quad a_1x \quad x^2$

$a_7x^2(a_6x + a_5) \quad a_4x + a_3) \quad a_2x^2 (a_1x + a_0) \quad x^4 \quad x^3$

$(a_7x^2+\dots+a_5) \quad (a_4x^4+a_3x^3) \quad (a_2x^2+\dots+a_0) \quad x^8 \quad x^5$

$(a_7x^7+\dots+a_5x^5) \quad (a_4x^4+ \quad \dots \quad +a_0)$

$(a_7x^7+ \quad \dots \quad +a_0)$

Using 10 PEs: 1 2 3 4 5 6 7 8 9 10

Utilization: 100% (1), 80% (2–5), 60% (6–9), 20% (10)

68% Average Utilization

Speed Improvement $= \frac{34-5}{5} = \frac{29}{5} = 580\%$

Figure 3.4-6	Sequential (Uniprocessor) Execution Time:	15
-7		19
	Uniprocessor Total Execution Time	34
	SAMSON Execution Time	5

Figure 3.10-4

Combined Web Speed Improvement

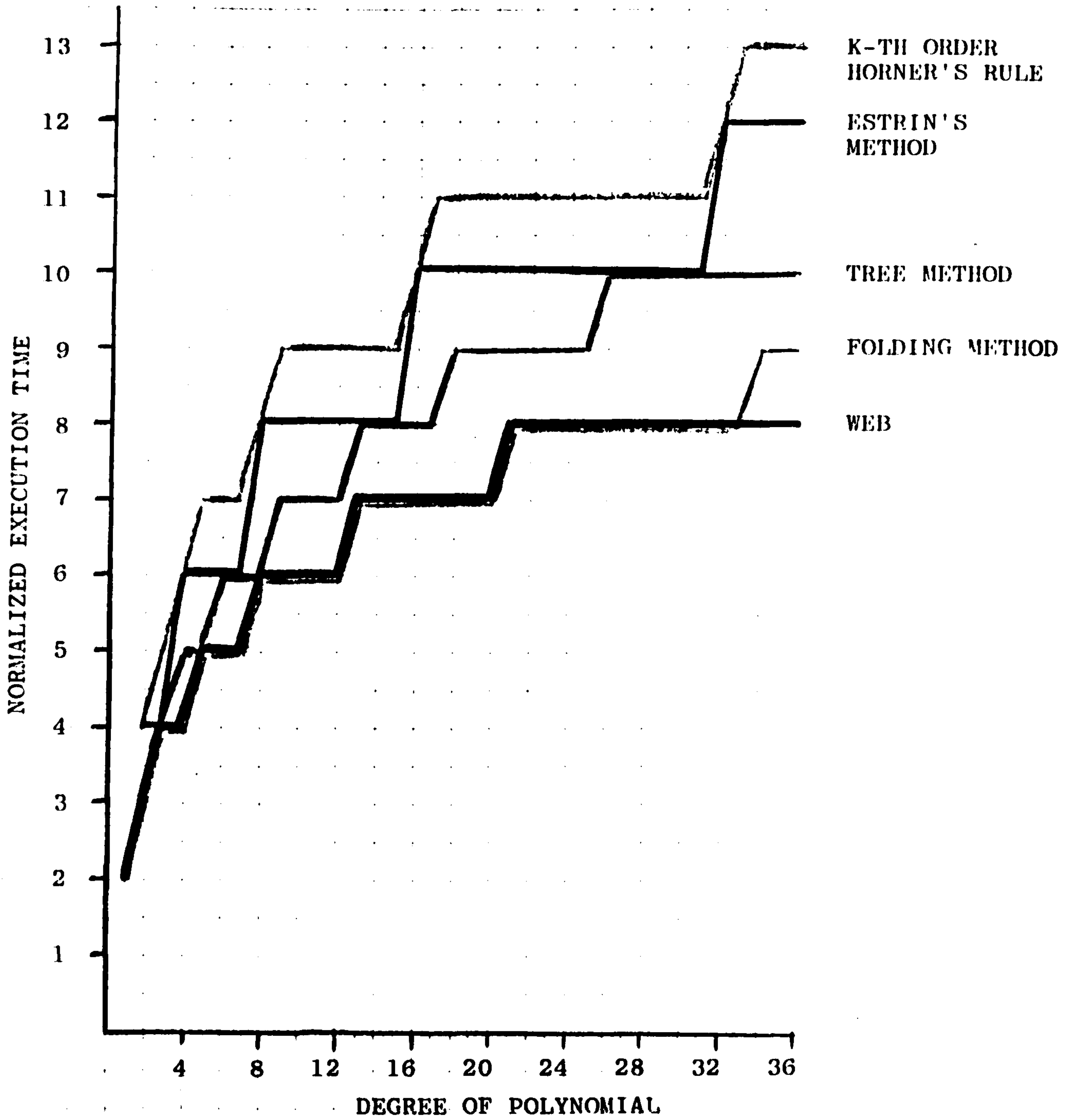


FIGURE 3.10-5
COMPARISON OF THE FIVE POLYNOMIAL COMPUTATION SCHEMES
(EXECUTION TIME VS. DEGREE OF POLYNOMIAL)

4.0 COMMUNICATIONS FOR MULTIPROCESSORS

4.1 INTRODUCTION

The interconnection of processing elements and memory units is an essential component of a multiprocessor system. However, the knowledge of how to efficiently interconnect such elements is still rudimentary [4.1].

In a conventional multiprocessor system, the efficiency of the system depends essentially on the efficiency of the communications between processing elements and memory units as well as the efficiency of intercommunication among the other system resources. In every case, communications between any of these units takes precious time. Hence, an efficient conventional multiprocessor architecture requires an interconnection technique which accomplishes this data transfer via a rapid communication mechanism. This communication can be managed in either firmware or hardware and either locally or globally [4.2], [4.3]; however, it must be performed at high speed while being both feasible, with today's technology, and economically practical [4.4] - [4.6].

The interconnection mechanism utilized in Input/Output communications must be performed at high data rates also. Furthermore, physically distributed processing systems such as those which might be used in real time computing application on board aircraft and spacecraft (which are distributed in order to provide fault tolerance in the event of explosions, lightning strikes, etc.) require communication at relatively high data rates.

Various approaches are possible for the intercommunications in a multiprocessor system, including communications via: internal buses, input/output buses, DMA channels, as well as various permutation networks. However, the performance of the multiprocessor will suffer significantly from an improper choice of the communication mechanism; the causes of this reduced efficiency may be:

- data strangling on the buses and/or channels;
- excessive delays due to switching and transfer times.

Thus, it is essential that the selection of the communication technique utilized by the multiprocessor system be based upon the relative performance of each of these approaches; rather than some ad hoc arbitrary selection.

In addition to the requirement for rapid communications, the interconnection network must provide sufficient communication between cooperating resources to enable data to be shared by all resources requiring this data. In the classical multiprocessor environment [4.7] a number of processors are connected by a switching system to the primary memory; however, the performance of such systems can be degraded by contention for both the switching paths and the memory units [4.8] as well as by the delays associated with the switching elements.

In this section, the performance achievable with various switching methods are studied and upper bounds on their

performance are analytically established. These results are compared in order to determine the interconnection network best suited for the real time multiprocessing I/O subsystem of the SAMSON system. Particular attention is paid to the real time requirements of these interconnection networks.

"Currently, one of the most active areas in computer architecture is the interconnection of computers to form systems...A discouraging aspect of this activity, however, is the almost total lack of published information describing the rationale for various designs, or comparing results achieved by various approaches" [4.9]. In every multiprocessor system and especially conventional multiprocessors, the interconnection network is a critical component [4.10].

4.2 MULTIPROCESSOR MODEL

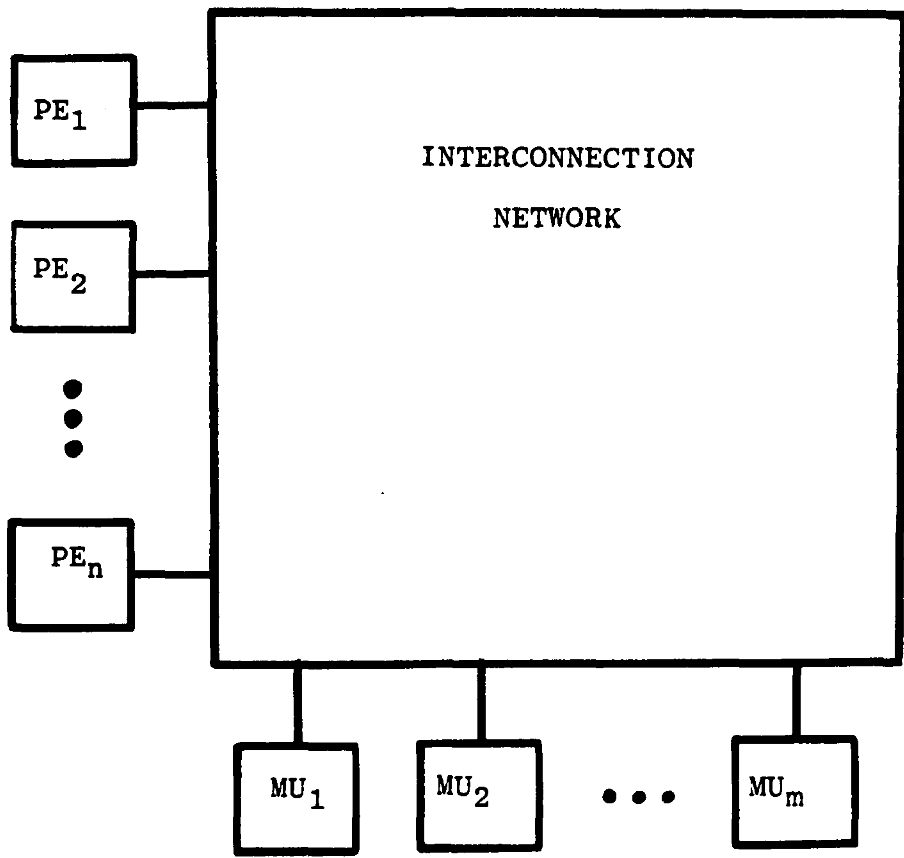
The principle aim in the design of a real time multiprocessor system is to increase the processing speed achievable by the computing system. In the following paragraphs, we will develop a general model of the processor-processor, processor-memory and memory-memory communications network utilizing the maximum throughput (T) attainable by the communications network as a measure of its performance. To facilitate this discussion our attention will be focused on, but not restricted to, processor-memory interconnections; these results can be, and are, applied directly to other interconnection networks.

The general structure being studied is the processor-memory interconnection network shown in Figure 4.2-1. The number of processing elements (PEs) is denoted by n and the number of memory units (MUs) is denoted by m. This multiprocessor system is referred to as an n x m system.

An ideal interconnection network provides the capability for maximum concurrent communications; i.e., all PEs can access different MUs simultaneously. The interconnection network provides the data paths such that each and every one of the n PEs can reliably [4.11] communicate with each and every one of the m MUs instantaneously. This ideal interconnection network is conflict-free due to the connecting network (although conflicts may still exist between PEs for access to a particular MU). Hence, in the ideal interconnection network a data path from PE(i) to MU(j) does not interfere with the connection of PE(k) to MU(l), provided $i \neq k$ and $j \neq l$; furthermore, these connections are established in zero time.

4.2.1 Modeling Considerations

The problem of modeling the performance of the interconnection network of a multiprocessor system is a complex one due to the large number of parameters that affect and characterize the behavior of such systems. These parameters include: the instruction mix, the access patterns (of the individual PEs to the MUs and the locations within the MUs accessed), access time, cycle time, processing time and execution time to name but a few. Furthermore, the processing time and the execution



GENERAL MODEL OF A PROCESSOR-MEMORY INTERCONNECTION NETWORK FOR A MULTIPROCESSOR SYSTEM

FIGURE 4.2-1

time are different for different instructions. Likewise, the instruction mix will differ for different applications resulting in different weighting factors for both the processing and execution times.

4.2.2 Modeling Assumptions

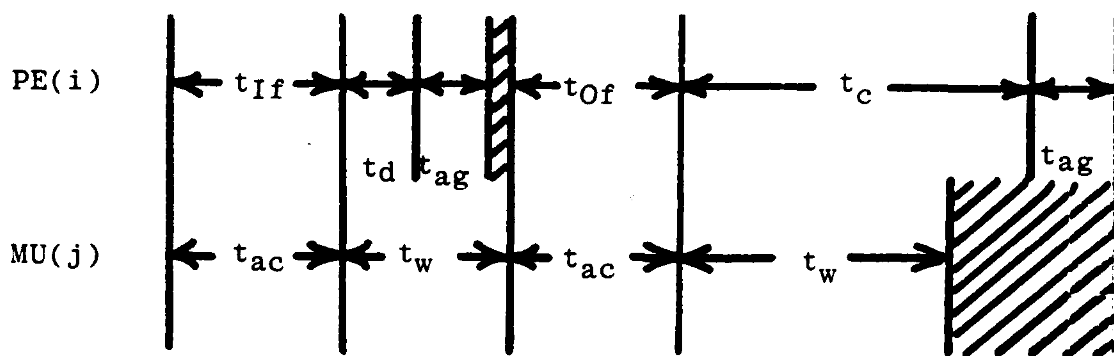
Figure 4.2-2 illustrates the timing associated with a typical instruction. It should be noted that the operand address generation time and the operand fetch and restore times may be non-existent as a function of instruction type. Furthermore, the instruction execution time may be less than, equal to or greater than the operand restore time as a function of both the instruction type and memory system utilized.

Due to the complexity of this problem, the exact detailed behavior of the communications within a multiprocessor system is extremely difficult to model. This will be demonstrated by briefly considering two of the parameters that affect the behavior of multiprocessor systems.

Instruction Mix: Instructions can be characterized by their relative frequency of occurrence in a program since the processor behavior varies for different instructions. The processor behavior varies since the processing time of an instruction differs from instruction type to instruction type.

Access Patterns: The access pattern, which is the trace of the memory units and memory locations accessed, likewise affects the multiprocessor performance since

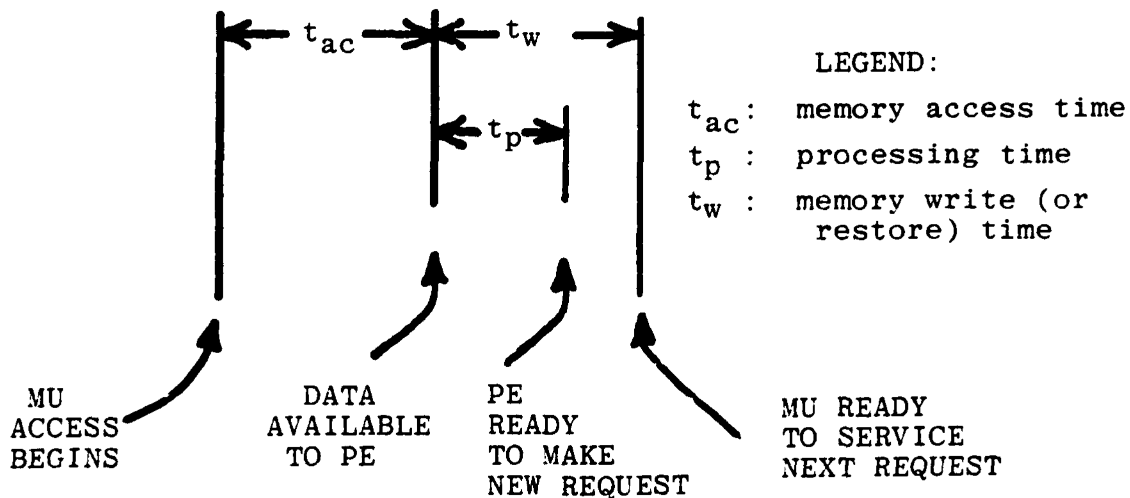
FIGURE 4.2-2 TYPICAL INSTRUCTION



LEGEND:

t_{If}: instruction fetch time
t_d : instruction decode time
t_{ag}: address generation time
t_{Of}: operand fetch time
t_e : instruction execution time
t_{ac}: memory access time
t_w : memory write (or restore time)

FIGURE 4.2-3 A UNIT INSTRUCTION



the MUs may be busy or ready when a request is generated by a PE and MUs may have quite different cycle times, access times and restore times.

Due to the complex timing relationship that exists between instruction timing and memory timing, the concept of the unit instruction [4.11], [4.12] is introduced. Figure 4.2-3 illustrates a unit instruction.

> Definition 4.1: A unit instruction consists of a single memory access followed by the processing time of the PE and the restore time of the MU.

The processing and restore times are overlapped (occurring simultaneously). In the unit instruction model, an operand fetch is simply considered as the occurrence of another unit instruction.

Hence, differences in instructions are not explicitly modeled. Processor behavior is modeled as a sequence of memory requests followed by the PE processing interval. At this level of modeling no distinction is made between the processing associated with instruction decoding and instruction execution. Here, the processing time characterizing the PE is the aggregate behavior of the real PE; hence, no compromising restrictions are placed on this model. In this model serial correlation between successive memory accesses will be ignored. This is not a serious assumption since data and instruction references are intermingled and are treated as separate "unit instructions". The effect of I/O activity is not

modeled explicitly since Strecker [4.12] has shown that if the rate of the I/O request is RIO, then a fraction RIO/m of the memorv access rate (of one MU) can be apportioned to I/O. (Here m is the number of MUs in the system.) Finally, no distinction is made in this model between read and write operations. This assumption places minimal restrictions on the model since these parameters are generally equal time intervals with today's computer technology.

4.2.3 Unit Instruction Model

The unit instruction [4.13] consists of a single memory cycle followed by the processing interval. The behavior of the various interconnection networks for the multiprocessor system considered in this work will utilize this unit instruction model to evaluate the relative performance of these interconnection networks.

The behavior of the multiprocessor system during the execution of the unit instruction as well as the notation utilized throughout this section is described below:

1. The PE generates a memory address and transmits it to the MU (this interval is part of the processing time, t_p);
2. The memory address propagates through the interconnection network eventually reaching the appropriate (that is, the correct) MU, where t_{sp} = switching network element propagation time (from the PE to the MU);
3. The MU then accesses the correct location, where t_{ac} = memory access time;

4. The accessed data propagates back from the MU through the switching network to the PE, where t_{s_m} = switching network element propagation time (from the MU to the PE);
5. Finally, the PE processes this data before generating any additional memory requests, where t_p = processing time (including address generation).

It should be noted that t_{s_p}, in general, is equal to t_{s_m} since the network is a symmetrical communication network. Hence, we will use the notation t_s to denote the switching network element propagation delay (see below).

Therefore, for a single processor of the multiprocessor system, the time per unit instruction, t_I, is:

$$t_I = t_{s_p} + t_{ac} + t_{s_m} + t_p$$

for $t_{s_p} = t_{s_m}$

$$t_I = 2t_s + t_{ac} + t_p$$

and the throughput of the multiprocessor system, T_{MS}, consisting of n processors is:

$$T_{MS} = \frac{n}{2t_s + t_{ac} + t_p}$$

It should be noted that for a uniprocessor system, where $t_s = 0$ and $n = 1$, $t_I = t_{ac} + t_p$

and the throughput of the uniprocessor system, T_{us}, is

$$T_{us} = \frac{1}{t_{ac} + t_p}$$

A thorough discussion of throughput considerations is presented later in Section 4.6.

In the case of processor to processor communications an interval, t_{PP_C} (the communication interval), equivalent to the unit instruction time exists. The components of this interval are identical to those of t_I where t_{ac} is interpreted as the data access time required within the responding PE and where t_p is the processing time within the requesting PE. Similar analogies exist for the memory to memory communications interval, t_{MM_C}. Hence,

$$t_I = 2t_s + t_{ac} + t_p,$$

$$t_{PP_C} = 2t_s + t_{ac} + t_p, \text{ and}$$

$$t_{MM_C} = 2t_s + t_{ac} + t_p.$$

4.3 REQUIREMENTS OF THE INTERCONNECTION NETWORK

The interconnection network must provide the capability for maximum concurrent communications with minimal interconnection delays and it must be such that it can be readily controlled by the resources requesting data via this network. In addition, the interconnection network must be able to resolve any conflicts which arise due to particular access patterns.

The interconnection network must provide the necessary data paths such that each and any of n' of the n PEs can communicate with each and any of m' of the m MUs in

an interval t_s. In the selection of an interconnection network, n' and m' should be maximized; i.e., n' approaches n and m' approaches m, while t_s should be minimized. In addition, the commands to the switching network should be in the form of memory bank numbers (or processor numbers) which should be a part of the address information produced by the requesting resource. Furthermore, since the memory (or processor) request patterns generated by the requesting resources are, in general, such that one or more conflicts may occur (e.g., more than one processor requesting service from the same memory), the switching network must be able to resolve such conflicts [4.14]. Alternatively, the switching network could provide the resource being requested sufficient information, regarding the multiple requests being received, so that the shared resource could itself resolve the conflict.

4.4 NETWORK CHARACTERISTICS

The basic interconnection (or transformational) paths required by a parallel processing system can, in general, be classified as:

- One-to-One Type Interconnection paths, and/or
- Global (One-to-Many Type) Interconnection paths.

Subclassifications within the One-to-One Type class include:

- Arbitrary Pair-wise,
- Linear Skewing (also referred to as Linear Shift, Shift or Skew Network), and
- Perfect Shuffle

while subclassifications within the Global class include:

- Neighborhood,
- Column Broadcasting, and
- Row Broadcasting.

It should be noted that the memory request patterns are not, in general, any of the orderly permutation patterns or geometric patterns. Therefore, permutation and geometric networks are of limited use for general purpose computations. Even in those applications where orderly permutations or geometric patterns are frequent occurrences, memory conflicts occur and corrupt the request patterns. Furthermore, the overhead penalty associated with control of parallel processors implemented with such networks is frequently excessive. Hence, such networks are of limited use. However, they can be of significant benefit in specific applications; in such special purpose applications they can be very efficient. In general, however, this is not the case. Even in applications where the majority of the computations benefit from such networks, the general overhead computations and control algorithms may suffer so dramatically that the penalties incurred exceed the benefits obtained from such networks. Consequently, some of the highly specialized networks within these subclasses are not considered here.

4.4.1 Notation

The PEs and MUs are considered as either the set of inputs, I, or the set of outputs, O, of a processor-memory interconnection network. In a processor-processor interconnection network, the PEs are considered to be

both the input and outputs; whereas, in a memory-memory interconnection network, the MUs are both the inputs and outputs.

During processor requests in a processor-memory interconnection network, the PEs are the inputs while the MUs are the outputs. (During the response to this request, the MUs can be considered the inputs and the PEs the outputs.)

In this regard, i is an input element of the input set I, and j is an output element of the output set O; i.e.,

$$i \varepsilon I \text{ and } j \varepsilon O$$

Here, the collection of PEs and MUs are the I and O sets (or O and I sets), respectively, with each i or j being a particular PE (or MU) or MU (or PE).

4.4.2 General Characteristics of One-To-One Type Interconnection Networks

The general characteristics of arbitrary pairwise networks, linear skewing networks and the perfect shuffle network are presented here to introduce the reader to the concept of One-to-One Type Interconnection Networks as well as potential applications of these networks.

4.4.2.1 Arbitrary Pairwise Interconnections

An arbitrary pairwise interconnection can be represented by:

$O_j = f(I_i)$ for any i and any j

where $i=1,2...,k \quad j=1,2...,l$

i.e., there are k elements of I and l elements of O and any pair can be connected via the interconnection network f.

The arbitrary pairwise interconnection network permits an arbitrary input i to be connected to any arbitrary output j in any pairwise (ij) manner, one at a time.

Applications of this arbitrary pairwise interconnection network include MIMD multiprocessors.

4.4.2.2 Linear Skewing Networks

Linear Skewing Networks, also referred to as Linear Shift, Shift or Skew Network, can be represented by:

$O_{j+s} = f(I_i)$
where s denotes the number of shifts
($s \geq 1$ and j+s is module k)

Applications of this type of interconnection network have classically been associated with matrix operations (i.e., matrix addition and subtraction) [4.15]. Consider the matrix A shown below, where a_{ij} are the elements of A,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

which is to be stored in m MUs for processing by a parallel processing system. If the elements a_{ij} are stored in (or fetched from) the corresponding MU_j in the normal matrix format, then one entire row can be accessed simultaneously; however, one column requires i sequential accesses. Obviously, a_{ij} can be operated on in transposed form; however, this would facilitate column operations at the expense of row operations. Storage in a linear skewed form, shown in Figure 4.4-1,

a_{11}	a_{12}	a_{13}	a_{14}
a_{24}	a_{21}	a_{22}	a_{23}
a_{33}	a_{34}	a_{31}	a_{32}
a_{42}	a_{43}	a_{44}	a_{41}
MU_1	MU_2	MU_3	MU_4

FIGURE 4.4-1 LINEAR SKEWED MATRIX ELEMENT STORAGE

permits simultaneous access to any entire row or any entire column, simultaneously. However, a skewing interconnection capability is required for both fetch and store operations to establish and maintain this linear skew storage (i.e., perform the necessary scrambling and unscrambling of the data elements) [4.16], [4.17].

4.4.2.3 Perfect Shuffle

The perfect shuffle [4.18] can be represented in the following manner (i=1,2,...,k and j=1,2,...,l)

$$O_j = f\ (I_i)$$

where $j=2i$ for $0 \leq i \leq \frac{k}{2} - 1$

where $j=2i+1-k$ for $\frac{k}{2} \leq i \leq k-1$

This network, shown pictorially in Figure 4.4-2 can be described as a transformation network which performs a right rotate of the input element numbers (assuming the element numbers are represented as binary numbers). Similarly, the output to input transformation can be represented by a left shift.

The most important application of this network is in the parallel processing computation of the Fast Fourier Transform (FFT).

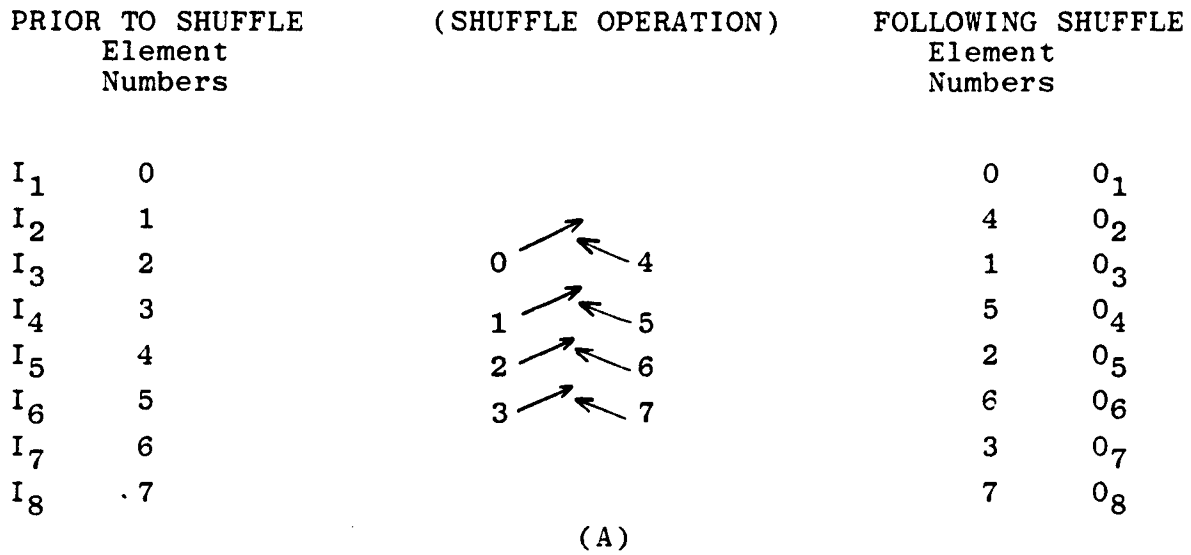
4.4.3 General Characteristics of One-To-Many Type Interconnection Networks

One-to-Many type interconnection networks are global in nature. They are most useful whenever data or instructions must be broadcasted to or shared by more than one resource; as in the case of some web structures, parsing and scheduling in general computational programs, and in the control of SIMD multiprocessors.

4.4.3.1 Neighborhood Interconnections

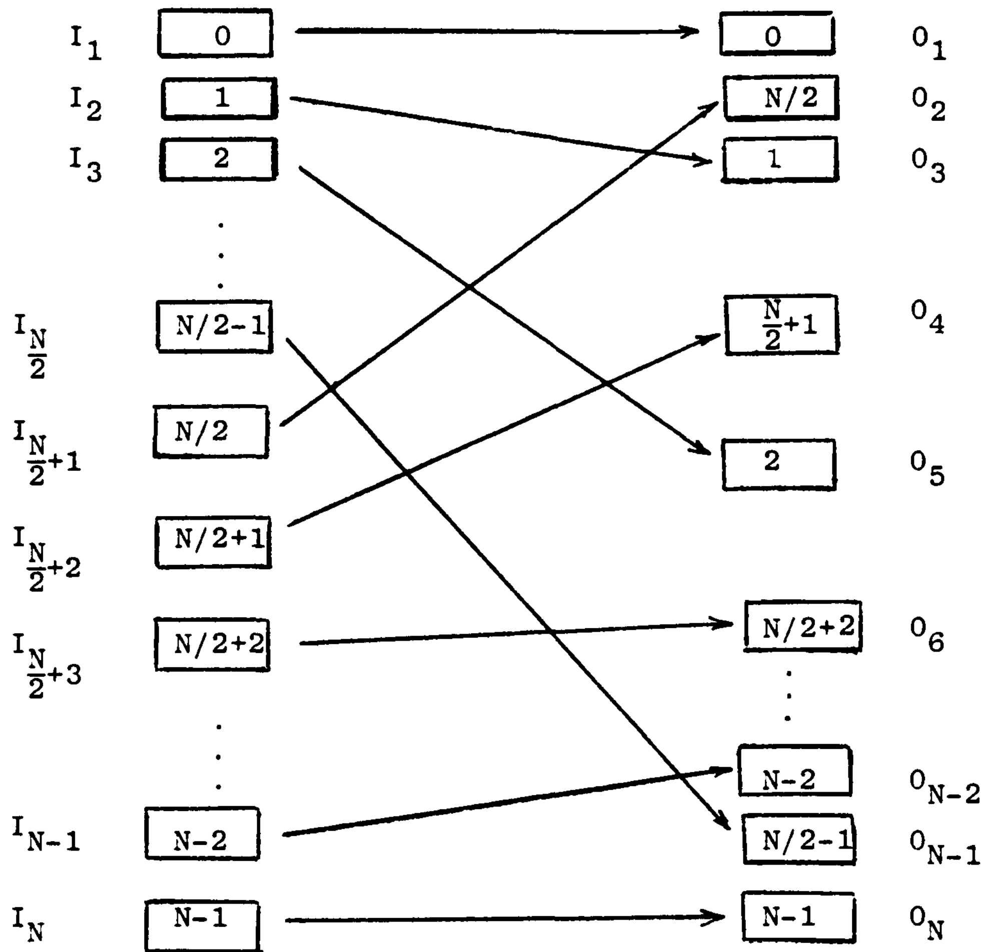
Various neighborhood interconnections can be created including: 4 way neighborhood communications, 8 way neighborhood communications, 10 way neighborhood

FIGURE 4.4-2 (A) PERFECT SHUFFLE OF 8 ELEMENTS



(A)

FIGURE 4.4-2 (B) PERFECT SHUFFLE OF N ELEMENTS



(B)

communication, etc. Figure 4.4-3 depicts 4 way, 8 way and 10 way neighborhood interconnection networks.

These neighborhood interconnection networks can be represented by:

4 way : $O_{jl} = f(I_{ik})$
where $jl = (i)(k+1),(i)(k-1),$
$(i+1)(k),(i-1)(k)$

8 way : $O_{jl} = f(I_{jk})$
where $jl = (i)(k+1),(i)(k-1),$
$(i+1)(k),(i-1)(k),$
$(i+1)(k-1),(i+1)(k+1),$
$(i-1)(k-1),(i-1)(k+1)$

10 way : $O_{jlm} = f(I_{jkn})$
where $jlm = (i)(k+1)(n),(i)(k-1)(n),$
$(i+1)(k)(n),(i-1)(k)(n),$
$(i+1)(k-1)(n),(i+1)(k+1)(n),$
$(i-1)(k-1)(n),(i-1)(k+1)(n),$
$(i)(j)(n+1),(i)(j)(n-1)$

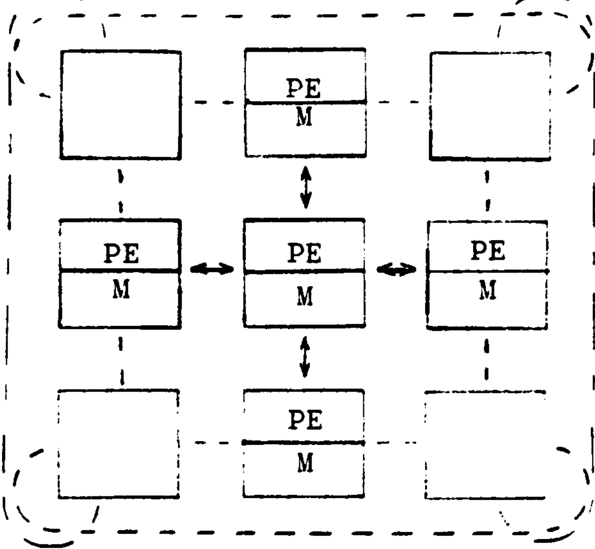
The four way neighborhood interconnection network is by far the most popular of these networks. It is useful for the solution of Laplace equations.

4.4.3.2 Row/Column Broadcasting

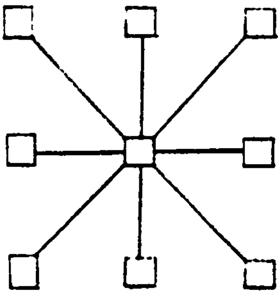
These networks can be represented by:

$$O_j = f(I_i)$$
$$\text{where } j=i,i+1,\ldots,i+k$$

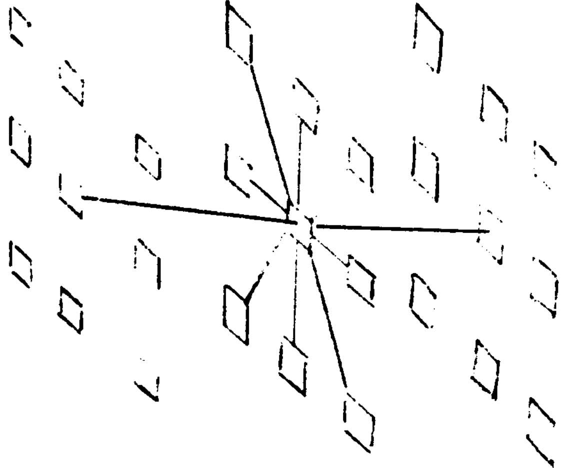
FIGURE 4.4-3: MULTIWAY NEIGHBORHOOD INTERCONNECTION NETWORKS
(A) 4 WAY; (B) 8 WAY; (C) 10 WAY



(A)



(B)



(C)

Typical applications of this type network would be for matrix multiplication where row and/or column operations are common place.

4.5 PROGRAMMABLE CONNECTIVE NETWORK DESCRIPTIONS

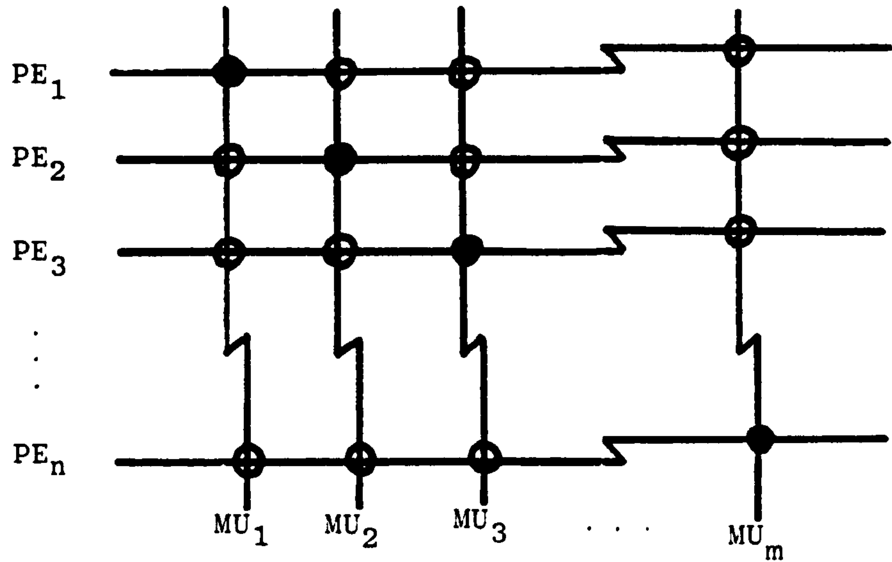
In this section, the programmable interconnection networks analyzed in this work are described in detail. These networks are the crossbar switch, various binary switches, time shared buses and packet switching (also referred to as time-slot or loop) networks.

4.5.1 Crossbar Switch

The programmable crossbar switch, shown in Figure 4.5-1 has the advantage of being able to make any arbitrary connection. The major disadvantage of this network is that it requires a relatively large amount of hardware.

This switching network can make any arbitrary connection in a minimal connecting interval (e.g., connection time interval, t_c). It is a relatively simple structure (in both its relay and integrated circuit, IC, form), easily constructed and simply controlled. Furthermore, considerable information regarding these structures are available due to the extensive studies already performed by such organizations as Bell Laboratories and others [4.19] - [4.21]. In addition to the above features, the crossbar switch can also perform One-to-Many type interconnections, as indicated by Figure 4.5-2.

FIGURE 4.5-1 CROSSBAR SWITCH



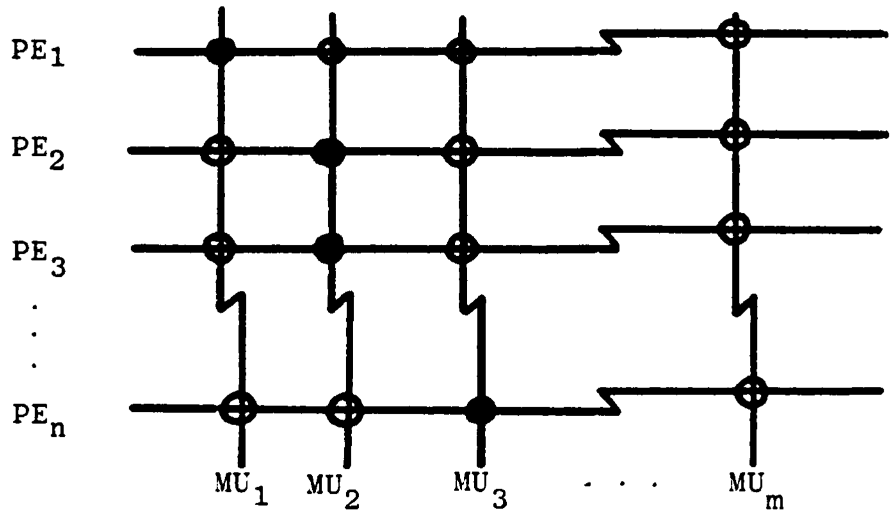


FIGURE 4.5-2 ONE-TO-MANY CONNECTION (MU_2 TO PE_2 & PE_3)

For an n x m bidirectional crossbar network, the number of gates required is:

2 nm 2 input gates, plus
m n input gates, plus
n m input gates

and can interconnect the input and outputs in $t_{g1}+t_{g2}$, where t_{g1} and t_{g2} are the propagation delay times of the 2 input and n or m input gates, respectively.

It should be noted that the crossbar switch provides sufficient concurrent interconnection paths so that all processors can access different memories simultaneously without any interference.

4.5.2 Binary Switches

Four binary switches are considered here; they are the Barrel Shifter, the Ω-network, the Flip-network, and Rearrangeable Switching Networks.

4.5.2.1 Barrel Shifter

The Barrel Shifter network allows any and all linear shift operations to be performed, where any linear shift of the inputs can be performed simultaneously on all inputs. Figure 4.5-3 is a 4x4 Barrel Shifter. The Barrel Shifter is composed of binary switch elements at each stage of the network; the binary switching element has two inputs and two outputs (except possibly the initial stage which requires only a single input). At each stage (column) of the Barrel Shifter, the inputs are from the same row (r) and the row immediately adjacent

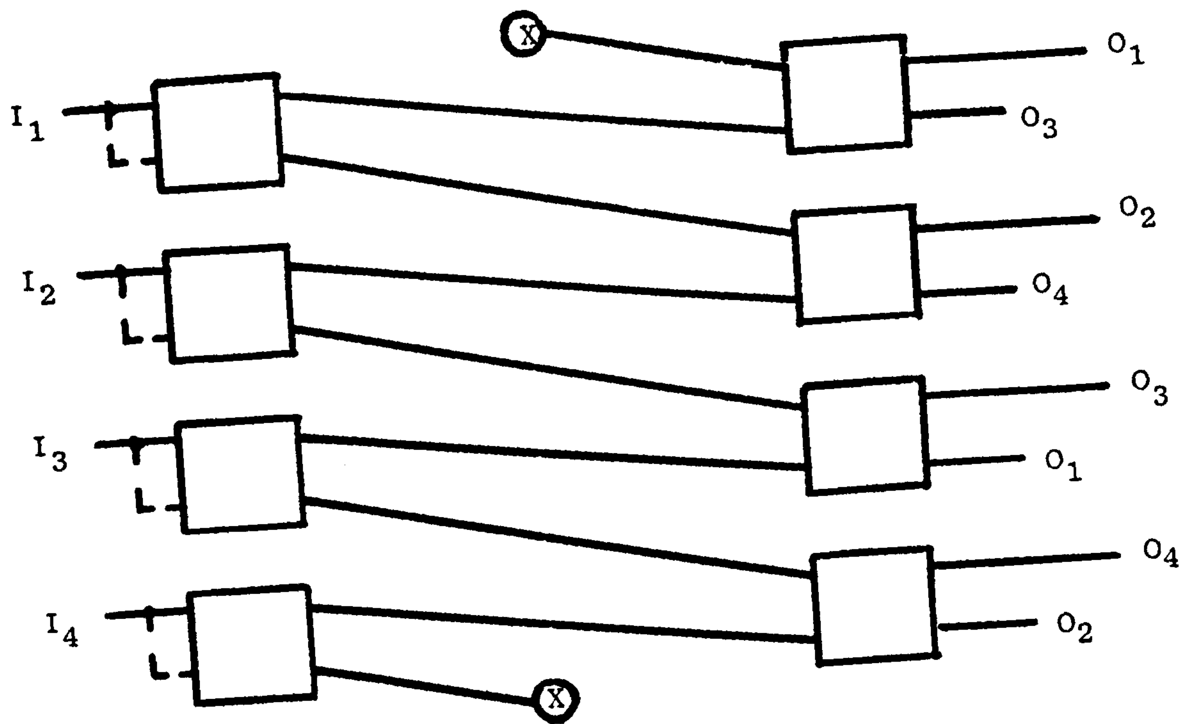
(r-1). The Barrel Shifter network of Figure 4.5-3 utilizes the adjacent row immediately above the present row; alternately the row immediately below (r+1) the present row can be utilized to generate a similar network. In the network depicted, the bottom row of the network is considered to be immediately above the top row. The 2N outputs of the final stages are connected as shown to produce an n x n interconnection network.

This network requires $\log_2 n$ stages with n binary switching elements per stage. Thus, a total of $n \log_2 n$ switching elements are required. Hence, a bidirectional network would require $2n \log_2 n$ switching elements. It should be noted that the binary switching element can be implemented with a 2x2 programmable crossbar switch.

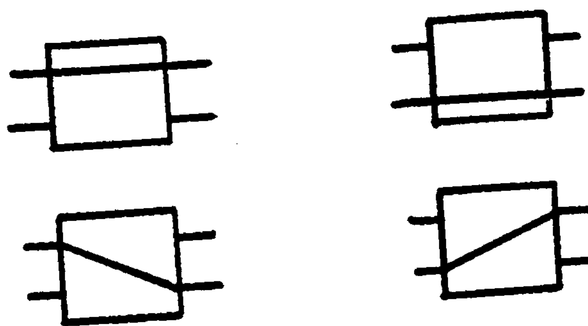
4.5.2.2 Omega Network

An 8x8 Ω-Network using 2x2 switching elements is shown in Figure 4.5-4. Higher order switching elements can be utilized; however, the interstage communications as well as the path-finding algorithms become much more complex.

The Ω network, using 2x2 switching elements, normally utilizes switching elements which allow only the states of the barrel shifter switching element to exist. However, this causes blocking (or interference) and path-finding problems. Here and in Lawrie [4.22] we have independently developed the Improved Omega Network which somewhat overcomes this problem by allowing the straight, interchange and both broadcast states to also exist within the switching element.



(A)



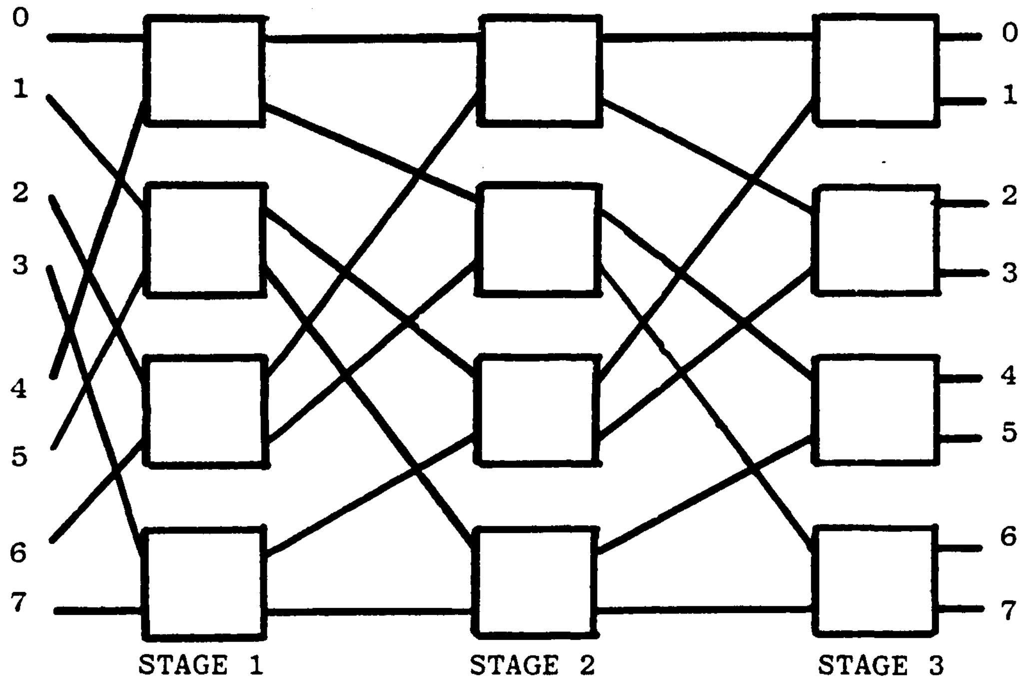
(B)

FIGURE 4.5-3 (A) 4x4 BARREL SHIFTER
(B) ALLOWED STATES OF SWITCHING ELEMENT

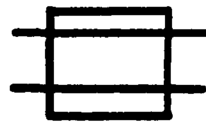
FIGURE 4.5-4

(A) AN 8x8 OMEGA NETWORK

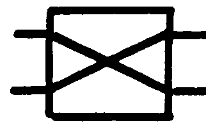
(B) ALLOWED STATES OF SWITCHING ELEMENT



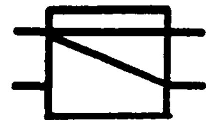
(A)



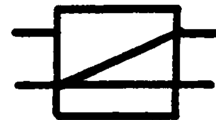
STRAIGHT



INTERCHANGE



UPPER BROADCAST



LOWER BROADCAST

(B)

The basic Ω-network is based on the operation of a perfect shuffle; it is capable of performing most of the "data" connections required by an array processor. However, this network cannot do any arbitrary permutation interconnection.

For the basic network, either input (but not both simultaneously) can be connected to either output (but not simultaneously). Thus, arbitration [4.9] must be provided when two input lines request the same output line. Ignoring such conflicts for now, it is obvious that the state of the stages can be controlled by the output element number, where the element number is represented in binary form $(b_1 b_2 \ldots b_n)$. The stage output of stage i is connected to the lower input if $b_i=1$ or to the upper input if $b_i=0$.

It is clear that with this network the number of permissible simultaneous connections are restricted. Interference can occur in several ways; i.e., two inputs that are applied to the same first stage switching element are contending for the same path through that stage. Whichever input receives the single data path (through that switching element)is interferring with the other input's interconnection path. Similarly, conflicting connective requirements can cause interference at every stage of the network; i.e., connection I_5 to O_7 and I_4 to O_3 interferes with I_3 or I_7 from reaching: O_6 due to second and third stage interference, O_4 and O_5 due to second stage interference and O_2 due to third stage interference.

Improving the switching element characteristics, so that the allowable states are those shown in Figure 4.5-4b, eliminates most of the interference; here all interference except the one to O_6 are eliminated. However. the extremely simple control algorithm described above must be replaced with one which is more complex.

The bidirectional nxn omega network consists of $2\log_2 n$ identical stages with n/2 switching elements per stage.

Other switching networks based on the well known properties of the perfect shuffle [4.18], [4.23], [4.24] have been developed by Benes [4.25] and Batcher [4.26]. The unidirectional omega network requires only $\log_2 n$ stages; this is one of the differences between this network and either Benes' or Batcher's. Benes' rearrangeable network requires $(2\log_2 n)-1$ stages. This network has the same capability as a crossbar but only n $\log_2 n$ gates (for an nxn network). The time to pass thru this network is on the order of $\log_2 n$ which is similar to that of the crossbar; unfortunately, this network is extremely difficult to set up in order to do any particular alignment. The Batcher sorting network requires $\log_2 n(\log n+1)/2$ stages. The logic of the switching element used in each of these networks is also somewhat different, but the complexity of the switching element is approximately the same as that required here.

4.5.2.3 Flip-Network

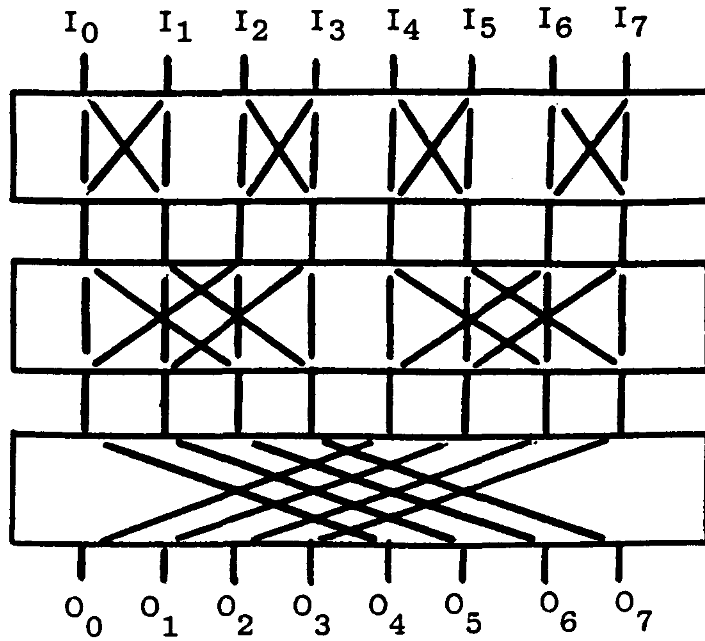
An n-Item Flip Network, where n=8 is shown in Figure 4.5-5. In Figure 4.5-5a the switching elements at each stage are different while the switching elements of Figure 4.5-5b are all identical.

An N bit ($N=\log_2 n$) flip control specifies one of the n possible flip permutations while the shift controls specify one of the $(N^2 + N + 2)/2$ shift operations [4-27]. This network which was developed for the STARAN Processor [4.28] requires only one pass through the network for some data manipulations. Some data manipulations, however, require multiple passes ($\log_2 n$ passes for n items while others require n passes for n items).

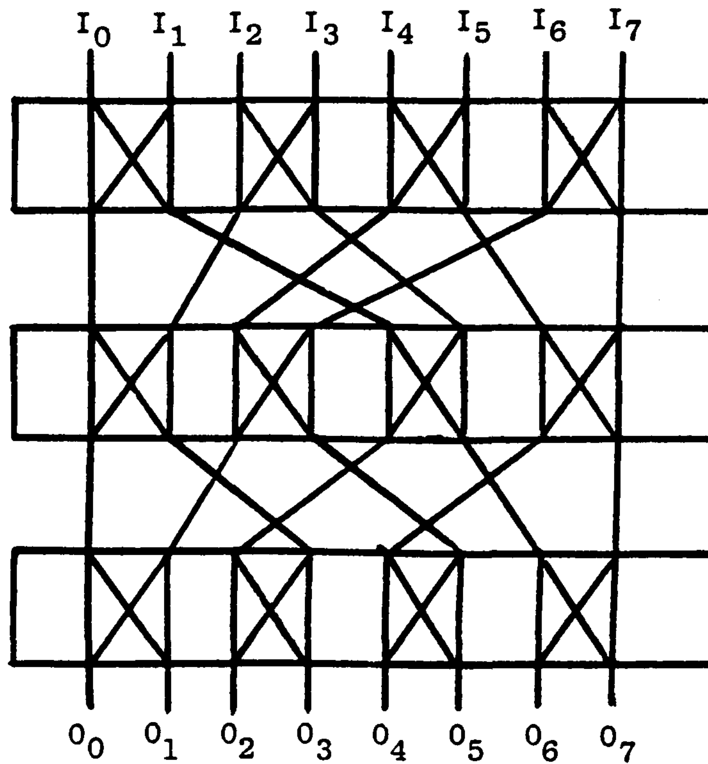
The n-Item Flip network has $\log_2 n$ identical stages when implemented as in Figure 4.5-5b. This is the same number of stages required by the Omega network; this however is not unexpected since the Flip network in this form is an Omega network with its input relabeled (shuffled), such as:

Omega	Flip
Io	Io
I4	I1
I1	I2
I5	I3
I2	I4
I6	I5
I3	I6
I7	I7

FIGURE 4.5-5
8-ITEM FLIP NETWORKS



(A)



(B)

4.5.2.4 Rearrangeable Switching Networks

The best known algorithm for setting up a particular alignment is a Rearrangeable Switching Network, RSN, which is due to Opferman and Tsau-Wu [4.29].

Unfortunately, this network requires on the order of $n\log_2 n$ time units to accomplish this interconnection. Comparing this with the crossbar switch which requires on the order of $\log_2 n$ time units to accomplish an interconnection,one must question the value of these RSNs for this application (in lieu of a speed penalty of a factor of n). Indeed, they are too slow to be practical in these high speed applications. (The Benes rearrangeable network discussed in Section 4.5.2.2, is an example of an RSN.) These networks are briefly described in this section for the purpose of completeness.

An nxn network, where n=dq such that d and q are integer factors of n, can be decomposed into an input and output stage each having n/d dxd networks, and a middle stage having d n/d x n/d networks as shown in Figure 4.5-6. The base-2 structure yields the most efficient network, when measured by the number of two-state switching elements required. In addition, the base-2 structure is relatively simple, consisting of 2x2 input and output stages (n-1 input stages and n-1 output stages). The middle stages are either $\frac{n}{d} \times \frac{n}{d}$ or they can be further decomposed [4.30] by means of an iterative process. If a base-2

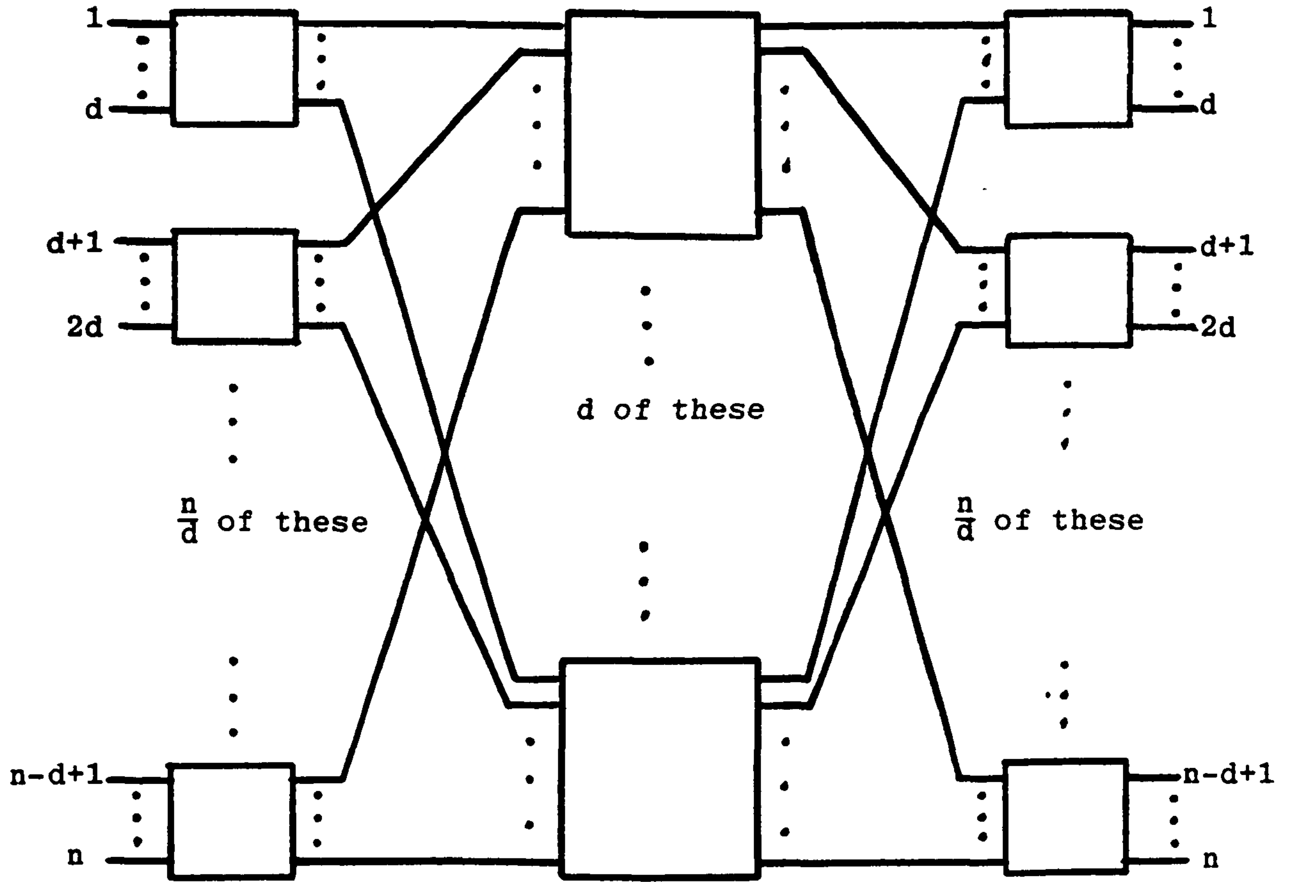


FIGURE 4.5-6 - REARRANGEABLE (n x n) SWITCHING NETWORK OF A GENERAL BASE-D STRUCTURE (RSN_d)

structure is carried throughout the network, then $n \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor} + 1$, 2 x 2, switching elements are required.

Minimum hardware is required when d is small; however, minimum propagation delay, which is proportional to the number of stages ($2 \log_d N-1$) occurs when d is large.

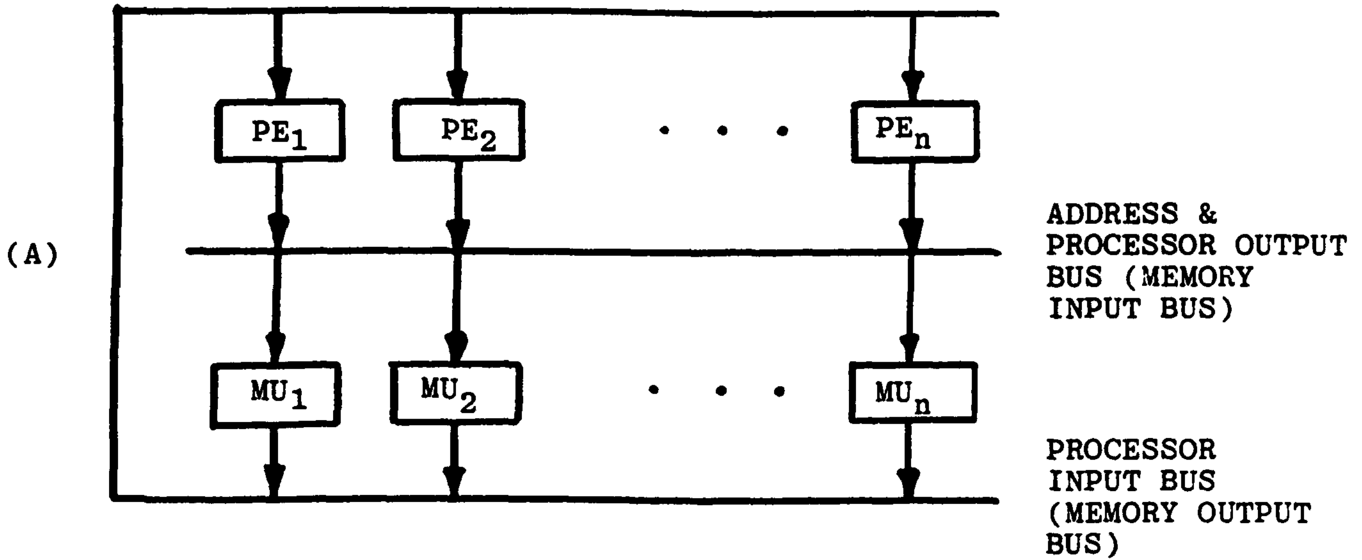
4.5.3 Time Shared Bus Interconnections

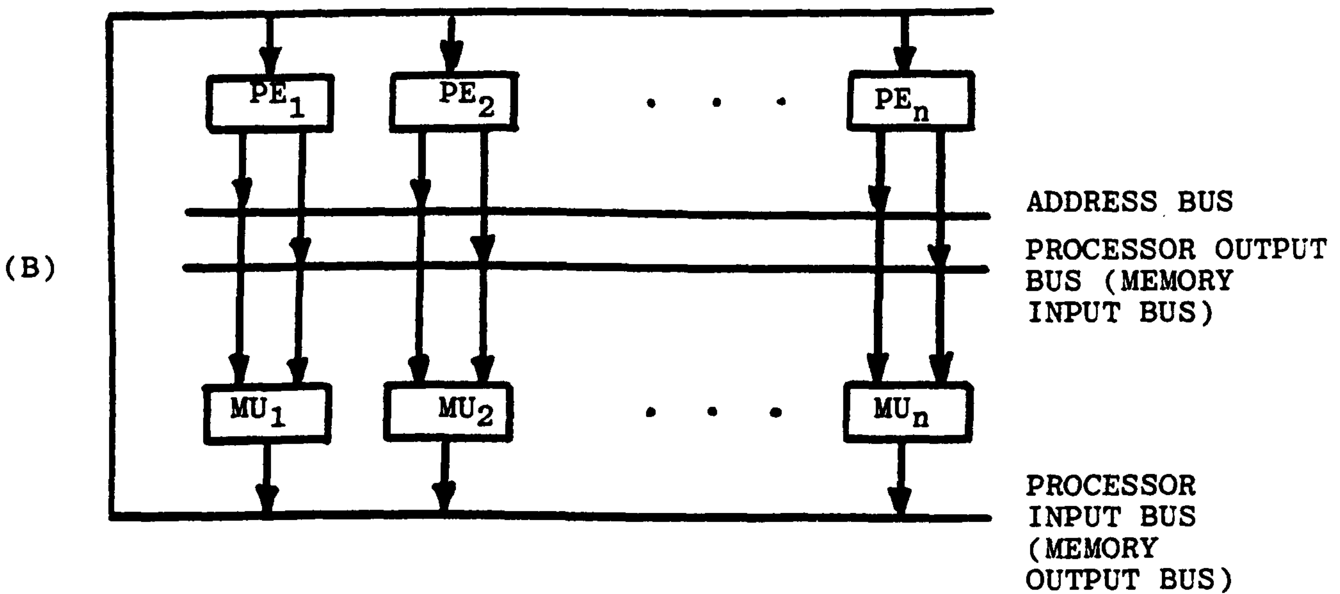
The time shared bus structure is quite common in uniprocessor computer application since it provides an economical and efficient [4.31] solution to the processor-memory and processor - I/O interface switching problem. However, as is frequently the case, this economical solution has a significant limitation associated with it; namely, the concurrency capability of this network is almost non-existent. This limitation is not significant in uniprocessor applications; however, it can be a significant shortcoming if improperly applied in a multiprocessor system.

In Figure 4.5-7 the lack of concurrent communications in a time shared bus structure is readily seen. One processor utilizes the address bus for one complete cycle when sending a request to a memory unit; likewise, the data bus (either the memory input bus for write operations or the memory output bus for read operations) is utilized for a complete cycle during the data phase.

FIGURE 4.5-7

TIME-SHARED BUS INTERCONNECTIONS





Both operations interfere with all other units using these buses during these intervals.

In Figure 4.5-7a, when the memory has completed its address acquisition phase it requires a complete additional cycle (T_D) to provide this data to the appropriate processor. During this T_D cycle, another (or the same) processor can utilize the address bus to provide a new request (to another memory unit). If a write operation was being performed by the first processor, it would interfere with the second processor's operation. The first processor should retain the bus so that it can provide the data to the memory unit, allowing this memory unit to complete its write cycle. In the case of a memory unit requiring an additional cycle before it is ready for the write data, the contention problem is simply delayed one cycle; the contention for the bus, however, still exists.

Consider the case where the cycle and access times of the memory units are not equal. If we allow a second read request prior to the first read data being returned, then the data from the second request could be returned prior to the data from the first request. In this case the processors must either know a priori which memory will respond at what times for all memory unit access patterns (and this may vary within a memory unit) or the data must be returned with additional information appended to it, which identifies which "data" it is or which processor the data is for.

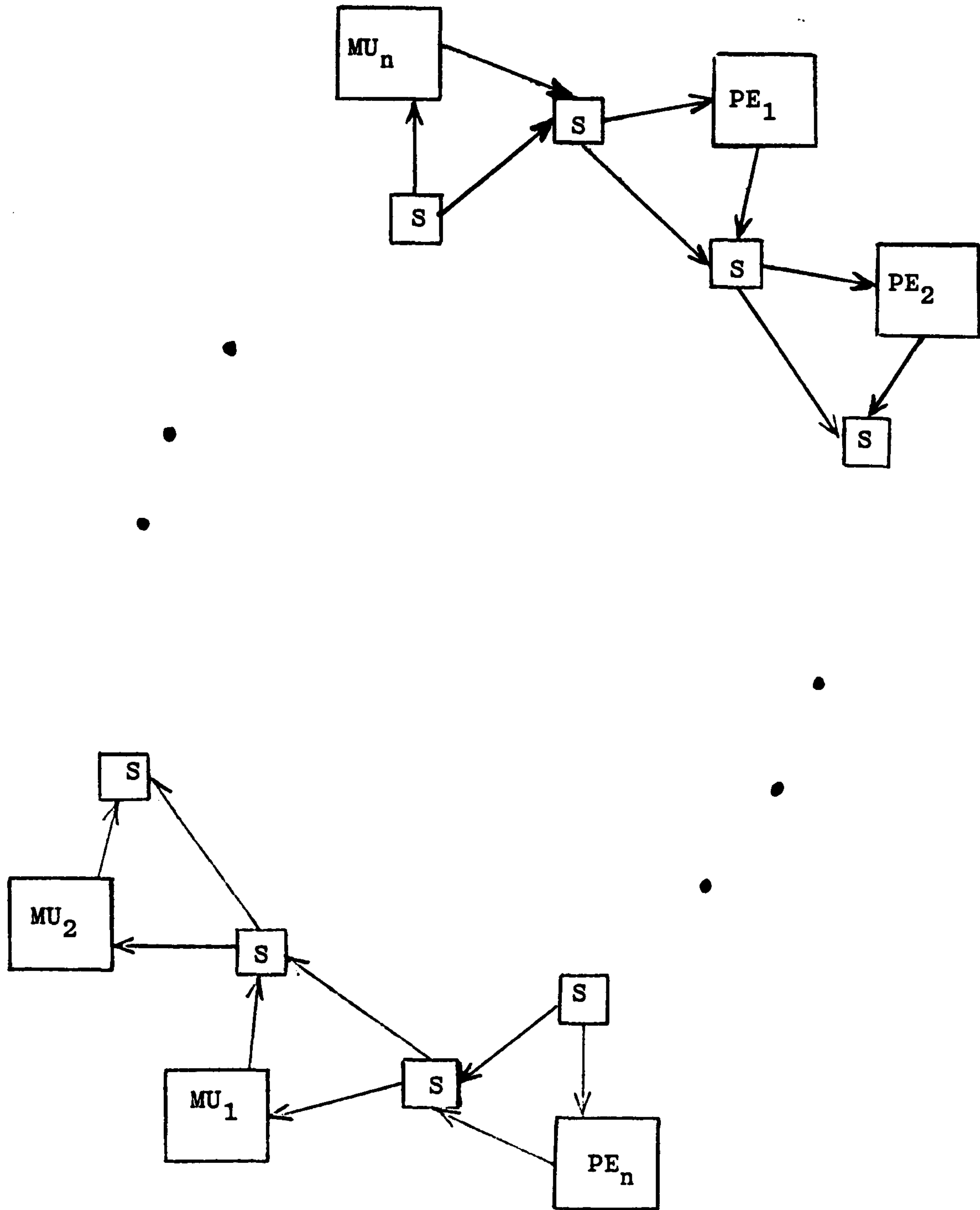
In any event, it should be clear that the ability of this type network to perform concurrent operations is non-existent or at least almost non-existent.

4.5.4 Packet Switching (Time Slot, Loop) Network

A packet switching, also referred to as a time-slot or a loop [4.32], [4.33] network is depicted in Figure 4.5-8. In this network, each processor request, processor response, memory request and memory response is considered to be a packet of information on a packet switched time slotted loop. The loop is divided into time slots, each either containing a packet of information or an empty time slot. The slots are shifted once around the loop during the loop cycle. The loop cycle time interval is equal to 2N slot time intervals.

In this type of network the data flowing on the loop must be tagged if the memory cycle time exceeds the slot time. This is necessary so that memory data, for example, can be associated with the address accessed and the processor requesting this data. If this tagging is not done, then the slot time must be lengthened so that it is compatible with the longest memory cycle time (assuming different cycle time memory units exist within the system). Obviously, lengthening the slot time interval is most undesirable since this is extremely inefficient with respect to utilization of available real time; maximum real time utilization is the main reason for a multiprocessor system.

FIGURE 4.5-8
PACKET SWITCHING, TIME-SLOT OR LOOP NETWORK



S = SWITCHING ELEMENT

4.6 NETWORK PERFORMANCE EVALUATIONS

In this section, we are concerned in particular with the performance achievable by the various programmable connective networks described in Section 4.5.

These connective networks provide the means for simultaneously connecting various resources of the multiprocessor system. Such capability has special significance in the multiprocessor system. These connective networks are extremely important, since resources have to be dynamically allocated and regrouped according to the state of these resources and since the particular needs of the system varies as the execution of those tasks currently being performed, proceed through the multiprocessor system.

In a multiprocessor system, it is important to note that there is, in general, no a priori distinction between inputs and outputs of the network; i.e. all terminals of the network are functionally equivalent.

4.6.1 Performance Measurement Parameter

One of the most important performance measurement parameters [4.34] of a real time processing system is its throughput rate. Throughput is defined as the number of outputs (which in this case is the number of instructions processed) per period of time. The throughput parameter directly reflects the processing power of a processor system. The higher the throughput rate, the more powerful the processing system.

Generally, a typical instruction mix or a bench mark program is utilized to determine the throughput of a processor. This is necessary since the execution time of an instruction differs from instruction type to instruction type. Computing the throughput using a typical instruction mix or benchmark is a simplified approach for accounting for the probability distribution of the instruction execution times. Effectively, throughput is the inverse of the average instruction execution time.

Utilizing the concept of the unit instruction eliminates the need for determining a typical instruction mix or an appropriate bench mark program. The throughput, in the case of the unit instruction model, is simply the inverse of the unit instruction time, t_I.

4.6.2 Crossbar Performance

The time per unit instruction for a multiprocessor system utilizing the crossbar switching network is:

$$t_I = 2t_s + t_{ac} + t_p$$

The factor $2t_s$ results from the fact that data must pass through the interconnection network twice. One piece of data is from the requesting resource and the other is from either the requesting or responding resource; i.e., address and data.

The throughput, T_{MS}, of a multiprocessor system having n PEs and n MUs connected via a crosbar is:

$$T_{MS_C} = \frac{n}{2ts + tac + tp}$$

4.6.3 Binary Switch Performance

The Barrel Shifter, the Omega-Network and the n-Item Flip Network all have $\log_2 n$ stages. Thus, the time per unit instruction for a multiprocessor system utilizing any of these networks is:

$$t_I = 2\ (\log_2 n)\ ts + tac + tp$$

and the throughput ($T_{MS_{BN}}$) of these binary switching network multiprocessors is

$$T_{MS_{BN}} = T_{MS_{BS}} = T_{MS_\Omega} = T_{MS_F}$$

$$= \frac{n}{2(\log_2 n)\ ts + tac + tp}$$

The rearrangeable switching networks described in Section 4.5.3.4 have d stages. Therefore

$$t_I = 2\ d\ ts + tac + tp$$

and

$$T_{MS_{RSNd}} = \frac{n}{2d\ t_s + tac + tp}$$

$T_{MS_{RSNd}}$ is maximized when d is minimized; for the minimum value of d, which is d=2

$$T_{MS_{RSN2}} = \frac{n}{4ts + tac + tp}$$

4.6.4 Time Shared Bus Performance

The average bus delay per unit instruction is $t_{BC/2}$, where t_{BC} is the bus cycle time; this assumes that the bus is not saturated. The worst case bus delay per unit instruction, assuming a non-saturated bus, is , obviously, t_{BC}. If the number of processors, n, exceeds the number of bus cycle subintervals, c_i, then bus saturation can occur and the bus structure becomes the system bottleneck. This situation arises when one PE is ready to utilize the bus and another PE is still utilizing the bus.

The bus cycle subinterval, tc_i, is equal to t_{BC}/c_i. Therefore, the bus saturates when $t_{BC} < n\ (tc_i)$.

Since $ts = tc_i$, the time per unit instruction is:

$$t_I = 2\ tc_i + tac + tp$$

and the throughput is:

$$T_{MS_{TS}} = \frac{n}{2tc_i + tac + tp} \quad \text{for } n \leq c_i.$$

If n is greater than c_i then the (average) throughput is the inverse of the (average) internal between allowable accesses onto the bus; this corresponds to bus saturation.

4.6.5 Packet Switching Performance

In the packet switched (time-slot, loop) network each unit instruction must transverse the entire loop, e.g., pass through each and every inner loop switch. With n PEs and n MUs, there are 2n inner loop switching elements. Hence, the time per unit instruction is

$$t_I = 4\ nts + tac + tp$$

and the throughput is:

$$T_{MS_p} = \frac{n}{4nts + tac + tp}$$

4.7 PERFORMANCE EVALUATION AND CONCLUSIONS

The upper bounds on the throughput for each of the various interconnection networks (with conflict free request patterns) were compared utilizing the following four sets of values of t_s (switching network element propagation time), t_{ac} (access time) and t_p (processing time):

	Set 1	Set 2	Set 3	Set 4
t_s	10 nsec	10 nsec	20 nsec	30 nsec
t_{ac}	130 nsec	160 nsec	450 nsec	500 nsec
t_p	130 nsec	90 nsec	350 nsec	400 nsec

The comparative graphs of the normalized throughput (T_N) vs the number of processors or memories (n), for data sets 1 through 4 are given in Figures 4.7-1 through 4.7-4. (All of these curves are normalized with respect to the throughput of a uniprocessor where $t_s=0$ and n=1.)

Table 4.7-1 shows the relative performance of each of these networks when normalized with respect to the ideal interconnection network.

Each of these networks exhibits a different performance dependency on n (where n is the number of processors and/or memories). Like the ideal network, the performance of the Crossbar and the Rearrangeable Switching $Network_2$ are linearly related to n. The Bus structure is also linear until bandwidth limiting (saturation). The Binary Switches on the other hand are proportional to $n/\log_2 n$ while the Packet Switched Network asymptotically approaches a constant (low performance) value.

FIGURE 4.7-1

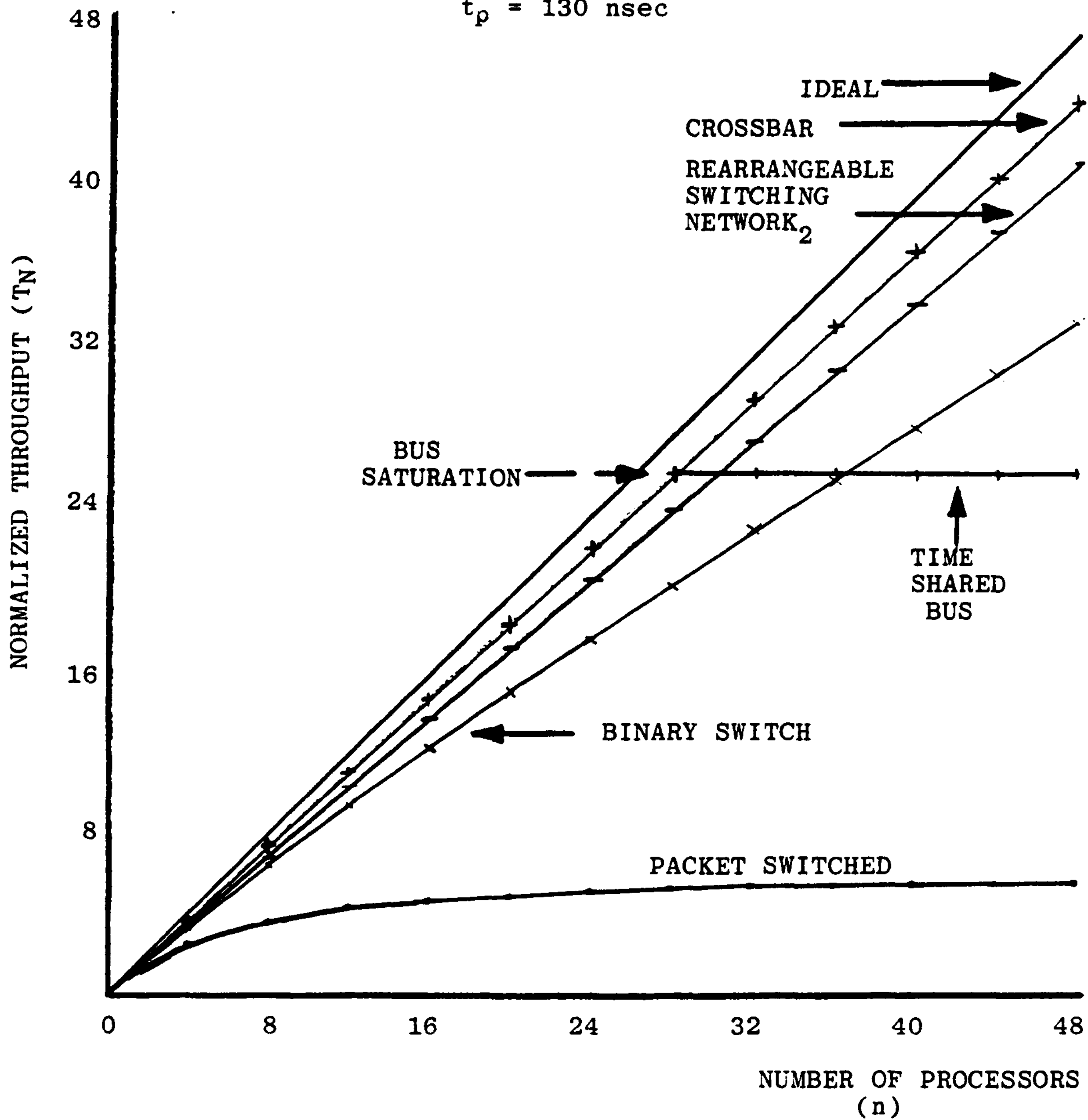


FIGURE 4.7-2

T_N vs n

t_s = 10 nsec

t_{ac} = 160 nsec

t_p = 90 nsec

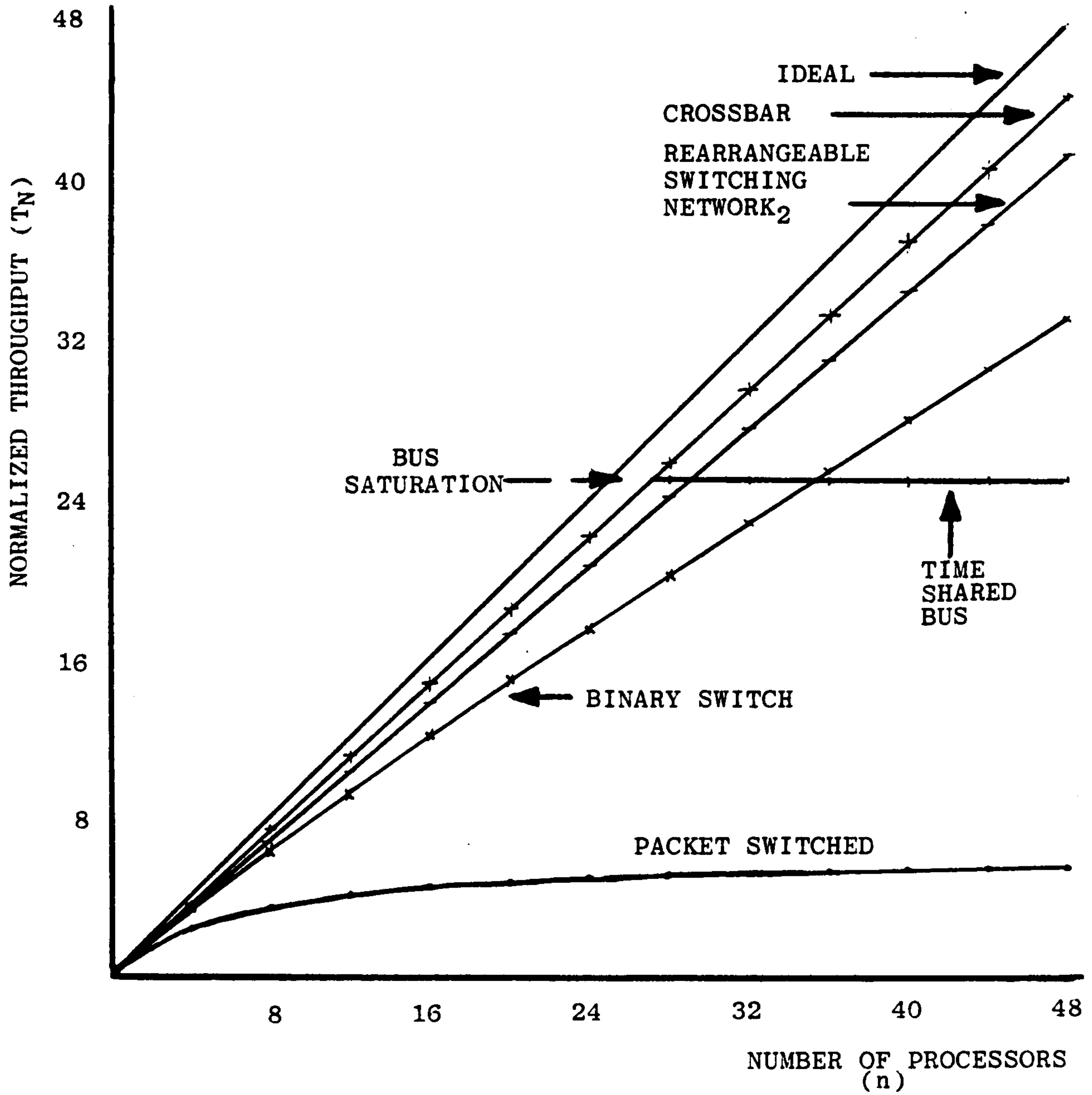


FIGURE 4.7-3

T_N vs n

t_s = 20 nsec

t_{ac} = 450 nsec

t_p = 350 nsec

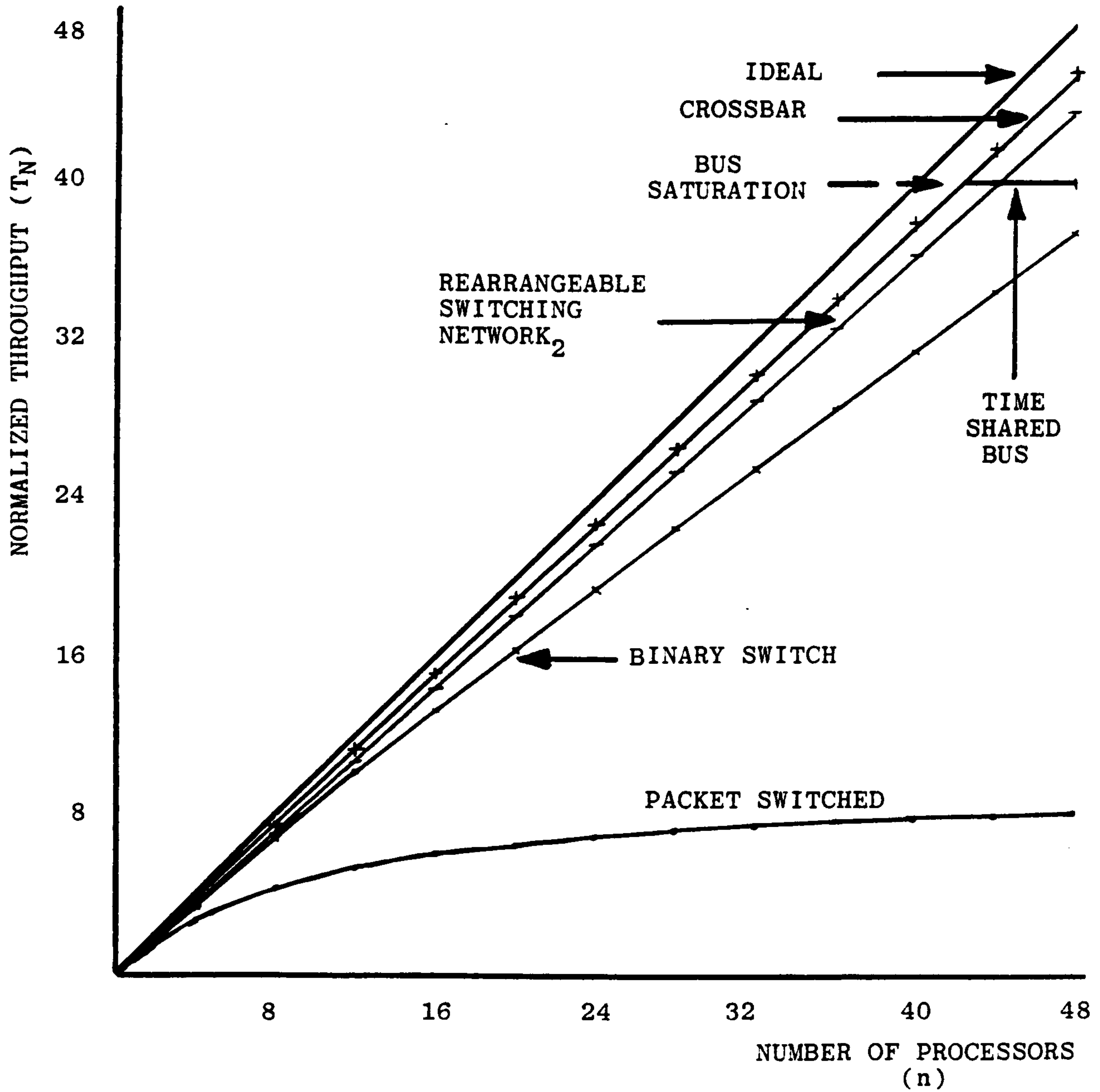


FIGURE 4.7-4

T_N vs n

t_s = 30 nsec

t_{ac} = 500 nsec

t_p = 400 nsec

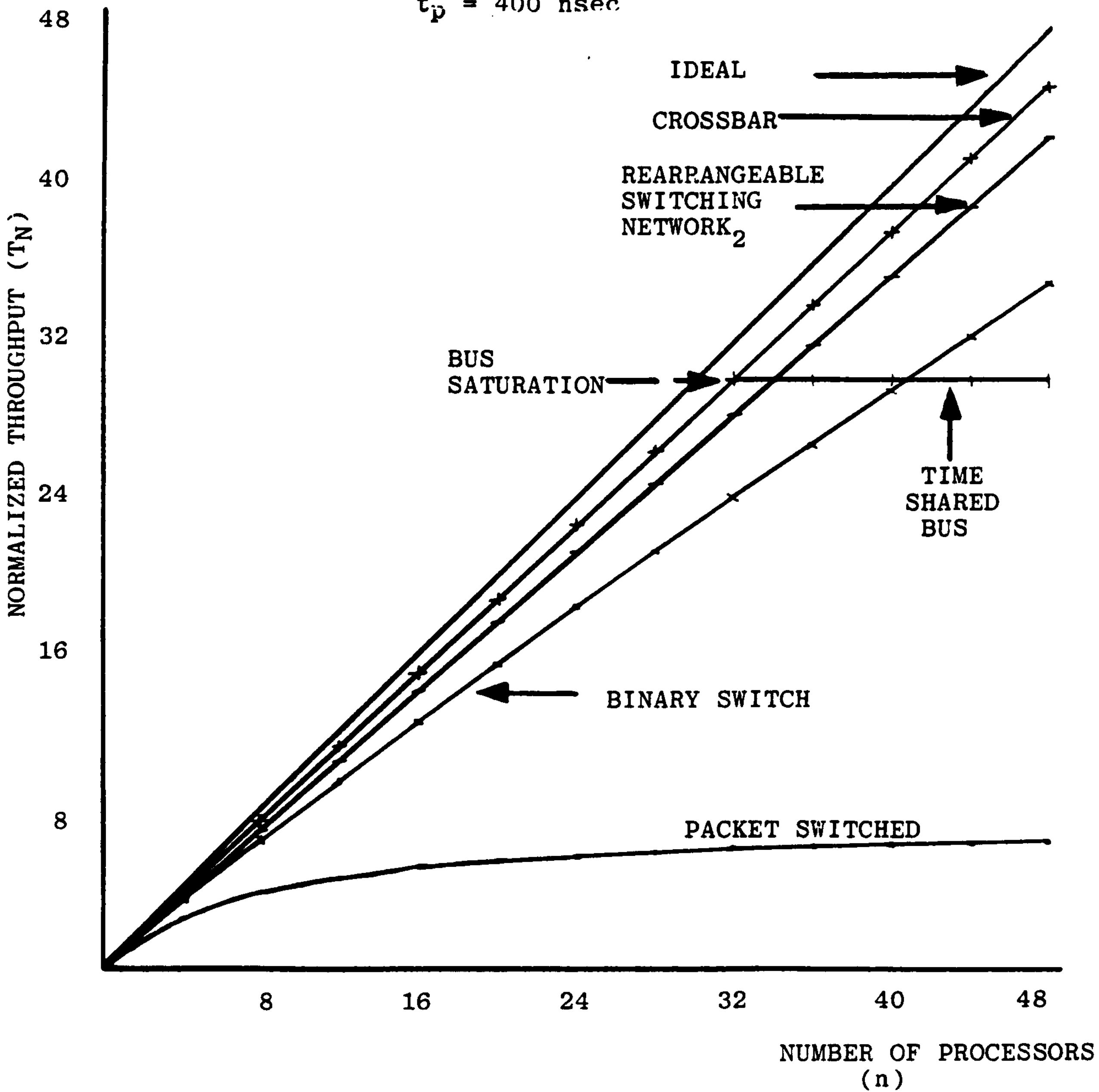


TABLE 4.7-1

RELATIVE PERFORMANCE OF INTERCONNECTION NETWORKS

t_s(nsec)	10	10	20	30
t_{ac}(nsec)	130	160	450	500
t_p(nsec)	130	90	350	400
$t_s/t_{ac}+t_p$	.040	.038	.025	.033
CROSSBAR	92.9%	92.6%	95.2%	93.8%
RSN_2	86.7%	86.2%	90.9%	88.2%
BINARY	69.9%	69.1%	78.2%	72.9%
BUS	92.9%	92.6%	95.2%	93.8%
BUS (SAT)	54.2%	52.1%	83.3%	62.5%
PACKET	11.9%	11.5%	17.2%	13.5%

The analysis of the behavior of the time shared bus reveals that there is little degradation in performance as compared to the ideal network (and no degradation in performance as compared to the crossbar) below bandwidth limiting, provided the ratio of t_s to $t_{ac}+t_p$ is small. These results agree with those obtained by Danielsson and Gudmundsson [4.35].

The crossbar is shown to approach the throughput performance of the ideal interconnection better than all other type networks. It is extremely interesting to note how poorly the packet switched network performed in comparison to the other configurations analyzed. Furthermore, the throughput performance of the Bus structure indicated that it was ideally suited for moderate size systems (and some large subsystems) where the effects of saturation would not limit its performance. Below the saturation level, this network's throughput performance equals that of the crossbar (surpassing all others). Since this configuration utilizes a minimal amount of hardware, it is ideally suited for I/O subsystem communications (provided fault tolerant characteristics are not a significant factor in the selection of the communication media).

It should be noted, despite the relatively high throughput performance of the crossbar, it is doubtful whether, with today's technology, it is economically practical (from a hardware viewpoint) to utilize a crossbar network rather than dedicated independent, isolated communication lines.

At this time, the time shared bus or dedicated, independent communications (providing such side benefits as fault tolerant communications)are more advantageous. However, as the technology continues to evolve the pendulum may swing in the direction of the time shared bus and the crossbar, especially in aerospace applications where the weight of the wiring used in dedicated communications is a critical factor.

From the results presented here only the crossbar and time-shared bus structure consistently approach 90% of the throughput of the ideal network. Since throughput performance improvement is the major goal of a multiprocessor, out of those interconnection networks analyzed, only the crossbar and time multiplexed bus (and of course dedicated, independent communciation lines) provide adequate throughput capacity so that bottlenecking and pathfinding do not become major multiprocessor drawbacks.

5.0 FAULT TOLERANCE OF THE SAMSON PROCESSOR

The highly modular PE structure of the SAMSON network facilitates the inclusion of fault-tolerant capabilities; these features can be provided in SAMSON at the module level. Error detection and recovery is provided in the SAMSON network through such features as redundant computations, reasonableness checks, task monitoring, and self-test software to name but a few.

Furthermore, the highly parallel structure of SAMSON (including the multiple processors, multiple memories and the interconnection network) allows reconfiguration of the system, upon detection of a fault*, to enable the processing task to continue. Such reconfiguration is accomplished in minimal time and with minimal degradation in performance; e.g., minimal effect on computational capability as well as minimal effect on the ability to detect and recover from further malfunctions.

In this section primary consideration is given to fault tolerance within a single SAMSON processor, while the next section is concerned with the fault tolerance of the

*An error within a computing system is defined here as that which generates an incorrect result. Such a malfunction of the computing system can be caused by various faults. (The terms malfunction and error are used synonymously throughout this work.) Faults are the various failures, such as transient, intermittent or permanent failures which occur. The classification of faults as transient, intermittent or permanent is a function of their rate of occurrence [5.1], [5.2]. (The terms fault and failures will, likewise, be used synonymously herein.)

SAMSON network. For clarity, certain network characteristics regarding fault tolerance are introduced and discussed in this section. Concepts regarding fault tolerance of general digital hardware are likewise discussed in this section.

5.1 INTRODUCTION

Despite the tremendous increase in the past few years in the reliability of the components utilized in digital computers, the chance of a failure, in any such system, still exists. In fact, due to the increased complexity of today's machines, the component reliability improvement has not significantly affected the overall system reliability. Furthermore, in many applications, either access to the machine is difficult (if not impossible), for purposes of repair, or a malfunction could have catastrophic results, as in spacecraft computer applications, air traffic control computer applications, and flight control computer applications. For these reasons, fault-tolerant computing techniques are a major goal of multiprocessor systems.

Most fault-tolerant systems are composed of a number of computers (three or more) executing independent copies of the identical program and comparing results. If a discrepancy exists, the majority disable the minority and processing continues with degraded error recognition and degraded or no recovery capability. Such an approach to the structure of a fault-tolerant computer system is extremely expensive, and has thus far been prohibitive, in all but critical applications. The problem with such

an approach is that system reconfiguration is accomplished at the highest system level [5.3]. The disabling of an entire computer upon detection of a malfunction significantly degrades the system performance, unless, of course, the number of redundant computers is very large. The cost of owning and maintaining such a system, having a large number of redundant, extra computers, is, in general, prohibitive. However, the SAMSON system has a highly modular structure, permitting system reconfiguration at a lower level. Upon detection of a fault, system reconfiguration is achieved by selecting a path around the failed component; i.e., selection of an alternate module (not necessarily an extra module) to perform (not necessarily replace) the function of the failed module.

In addition, the occurrence of a fault in a SAMSON component does not significantly affect the capability of the SAMSON system to detect and recover from other malfunctions which might occur, following the occurrence of the initial fault. This ability to detect subsequent errors and to further reconfigure the SAMSON system, in response to these additional malfunctions, is essentially due to the bypassing of faulty components.

The reconfiguration capability within the SAMSON system exists as long as there remains sufficient fault-free components (including processors, memories and interconnection paths) to perform the required computation in the required real time.

The processing power obtained in the SAMSON system far exceeds that obtained with redundant multiple computers. Furthermore, the additional cost of the SAMSON computer system (incorporating these fault-tolerant capabilities) is much less than the additional cost of a computing system utilizing redundant multiple computers when both approaches are compared with the cost of a non-fault-tolerant system.

5.2 THE FAULT-TOLERANT PROBLEM

The problem of fault-tolerant computing can be segmented into three essentially, independent problems; these three problem areas are error detection, recovery (continued operation) and reconfiguration (isolation and/or amputation).

When fault-tolerant capabilities are introduced into the computing system, the first consideration must be given to detecting the occurrence of an error, whether it is due to a hardware fault or some other system malfunction. Next, the computing system must be capable of successfully continuing the computational task(s) being performed; i.e., recovering from the error which would otherwise corrupt the task(s) in process. Finally, if a given error was caused by a hardware failure (as opposed to a transient disturbance),the failed module(s) must be isolated, and amputated from the system, to prevent corruption of the remaining fault-free modules; i.e., reconfiguration of the system in the presence of the fault.

5.2.1 Error Detection Problem

Various permanent faults as well as many types of transient and intermittent faults may occur in a computing

system which will cause an error in the computation. Some faults result in the "stuck-at" class of faults, while other errors may not be so readily recognizable. Those errors which are not so recognizable may cause an incorrect result to propagate through a computation and generate an incorrect data value, an extra data value, an incorrect address, a misdirected piece of information, etc. To detect such errors it is necessary to introduce some degree of redundancy; such as, execute several copies of the computational task and compare the results of these computations. From a real time detection point of view, it is desirable that these multiple copies execute simultaneously. However, from the point of view of total error detection coverage, it is desirable that these copies execute at different times; thus allowing detection of failures due to noise and other such bursts which could otherwise affect all redundant resources (multiple simultaneous malfunctions).

The error detection capability can be provided in the SAMSON system through such features as:

- error detection and correction codes
- reasonableness checks
- task monitoring
- wrap around checks
- redundant computation and comparison
- CPU self test
- memory tests
- I/O tests

Obviously, essential to all of these health checks, is the ability of a healthy processor to detect a failure in one or more of these tests within itself or another machine. Clearly, performing I/O tests utilizing a faulty processor will result in erroneous results. Consequently, self-testing and cross-testing of the processors is essential to fault-tolerant multiprocessor error detection in the SAMSON system. Later, in this work, self-testing rather than cross-testing receives greater attention since it is, by far, the more difficult of the two problems.

5.2.2 Error Recovery Problem

Having provided the necessary error detection facilities to permit the detection of an error, it is also necessary to provide a mechanism which allows graceful recovery from that error. Classically, a triple modular redundancy (TMR) scheme has been utilized, incorporating triplicated voters to select the majority result. However, TMR techniques require at least triplication of all hardware; at least three operational functional units (of each type) are required and furthermore different copies of these functional units are required by each of the separate copies of a given program.

In the SAMSON system, single computational units are primarily used; they are supported by reasonableness checks, self-test, etc., where possible to identify the occurrence of an error. Upon detection of the error, an alternate computational unit (as well as the original unit) is utilized to repeat the "corrupted" computation utilizing stored profiles of the "input" data values corresponding to this corrupted computation.

Only if the computation being performed is critical in nature will more than one copy of the program be executed. In general, sufficient (real time) response-time will exist to enable repeating the corrupted computation upon detection of the error; e.g., in flight control applications 57 msec [5.4] exists for real time error recovery. Consequently, in the majority of cases, including critical applications, at most two copies of any program is executed. Only in critical applications with severe (real time) response-time restrictions is it necessary to utilize triplicated computational techniques to continue the required computational task(s).

5.2.3 Reconfiguration Problem

Reconfiguration requires the isolation of a faulty module in order to prevent contamination of the computational facilities.

Reconfiguration is accomplished in the SAMSON system by maintaining an active library, within each processing element which reflects that processor's opinion of the health of all other resources. This distributed intelligence is utilized by the SAMSON processing elements to determine which modules are healthy and available for processing tasks as well as which modules are sick and must thus be ignored. Faulty modules are not physically amputated from the system; instead, they are simply ignored by the healthy resources.

5.3 ERROR DETECTION IN DIGITAL CIRCUITS [5.5],[5.6]

This section will familiarize the reader with some aspects basic to the concept of error detection as well as path sensitizing and the need for appropriate test sequences.

5.3.1 Path Sensitizing

Path sensitizing is the procedure utilized to generate a set (or sets) of test sequences for detecting a hard (non-intermittent) fault in a digital system. Hard faults are assumed to occur, one at a time, in the system and these faults are assumed to persist for the duration that the test sequence(s) is applied. A fault is said to be detected if the output (or one of the outputs) differs, in a detectable manner, from its expected value during (any of) the applied test sequence(s).

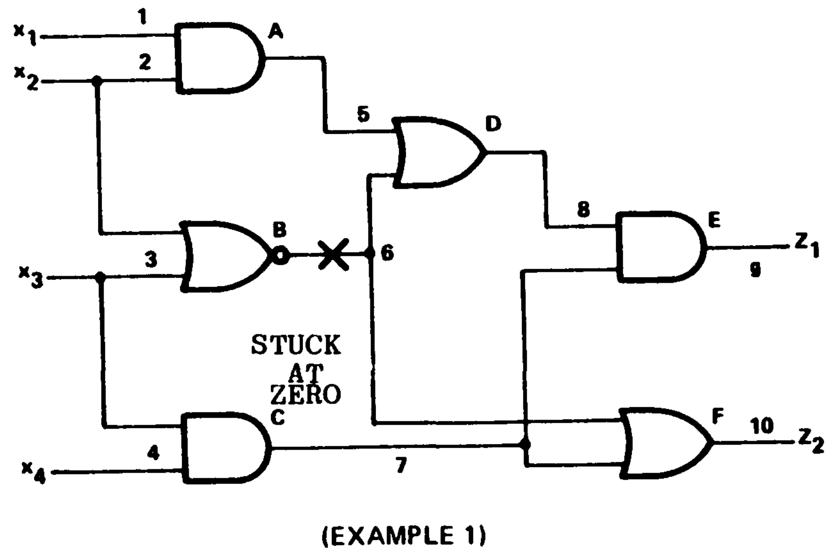
The procedure for path sensitizing consists of the following four steps:

1. Selection of a node as a potential faulty node in either the stuck-at-one or stuck-at-zero state.
2. The selected faulty node is assigned a sensitizing value opposite to the fault condition selected.
3. A path from the selected faulty node to the output(s) is selected for sensitization.
4. The inputs to the logic elements along the sensitizing path are assigned (non-faulted) state values so as to propagate the selected faulty node state along the sensitized path to the output(s).

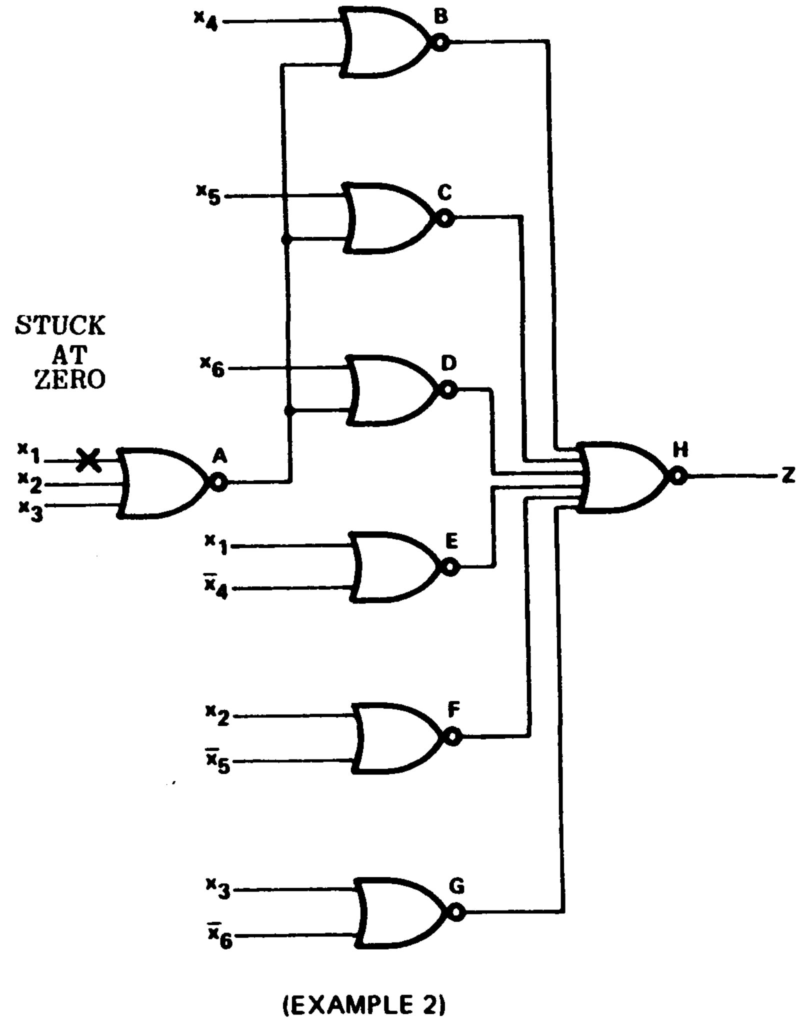
This procedure is illustrated by the examples given in Figure 5.3-1. In the first example, the test sequence is examined for its ability to detect a stuck-at-zero fault at the output of gate B. The path sensitizing and test sequence generation procedure is:

1. The output of node B in the stuck-at-zero state is selected as the faulty node.
2. The node (node B) is assigned the sensitizing value of one (logic level one, LL1). Since the output of a NOR gate can only be a LL1 when all its inputs are zero, the input signals X_2 and X_3 must both be made zero to make the faulty node a LL1, its sensitizing value.
3. The sensitizing path BF is arbitrarily chosen to propagate the fault.
4. The fault can be propagated through F if and only if the output of C is zero (logic level zero, LLO). Since the output of an AND gate is zero whenever any of its inputs are zero, and both X_2 and X_3 are already zero, the state of X_1 and X_4 are don't cares (represented by X). Therefore, the input test pattern X_1 X_2 X_3 X_4 = XOOX will detect this selected fault.

It can be shown that for the same fault, it is impossible to detect that fault along the path BDE. This is true since that path would require X_3 to be sensitized to both a LLO and a LL1 simultaneously (to propagate the fault to output E).



(EXAMPLE 1)



(EXAMPLE 2)

FIGURE 5.3-1 - EXAMPLE FOR PATH SENSITIZING

The second example is included to illustrate one of the drawbacks of sensitizing single paths using the path sensitizing method. Consider the problem of deriving a test sequence for detecting a stuck-at-zero on input X_1; the procedure is then:

1. The X_1 input of device A in the stuck-at-zero state is selected as the faulty node.
2. The faulty node is assigned the sensitizing value of one.
3. The sensitizing path ADH is arbitrarily chosen to propagate the fault.
4. To propagate through gate A, X_2=X_3=0; through gate D, X_6=0; and through gate H, only if the outputs of gates B=C=E=F=G=0. To make C=0, X_5 must equal 1. However, having X_2=0 and X_5=1 makes F=1. Due to this, and other similar contradictions, it is clear that the fault cannot be propagated to the output (Z) along path ADH. In fact, none of the single paths can be used to propagage the fault in this example.

This dilemma can only be solved by attempting to propagate the fault along paths ABH, ACH and ADH simultaneously. In order to do this, the path sensitizing procedure to obtain the required test sequence is:

1. Same as previously described.
2. Same as previously described (X_1=1).
3. The compound sensitizing paths ABH, ACH and ADH are selected.

4. To propagate through A, $X_2=X_3=0$; through B,C and D, $X_4=X_5=X_6=0$; and through H, requires E=F=G=0. The input values assigned ($X_1X_2X_3X_4X_5X_6$ = 100000) satisfy the requirements for E=F=G=0 without causing any contradiction. Hence, the stuck-at-zero fault on input X_1 can be detected with this test sequence.

Test sequences for hard faults can be obtained by sensitizing single and multiple paths utilizing a D-algorithm which utilizes the elements and rules of D-algebra.

5.3.1.1 D-Algebra

In the D-algebra, D represents a signal which is a LL1 in in the normal (non-failed) condition and a LL0 in the faulty condition. $\overline{D}$ represents the logical complement of D.

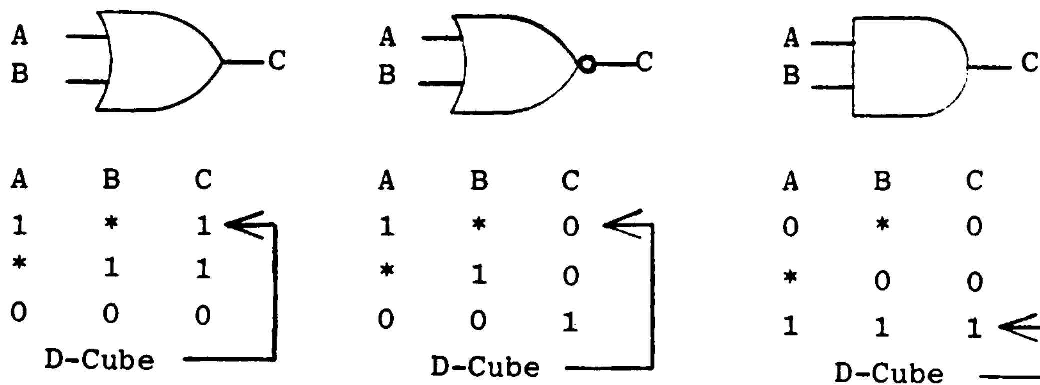
Definition 5-1: A D-Cube is a vector representation of the input and output values of a device.

Several D-cubes are shown in Figure 5.3-2.

Definition 5-2: A Singular Cover is a set of D-Cubes defining a particular device.

Figure 5.3-2 is the singular cover of several common gates.

Definition 5-3: A Primitive D-Cube is the set of D-Cube(s) wherein a faulty input causes the output of a device to be different from its normal value.

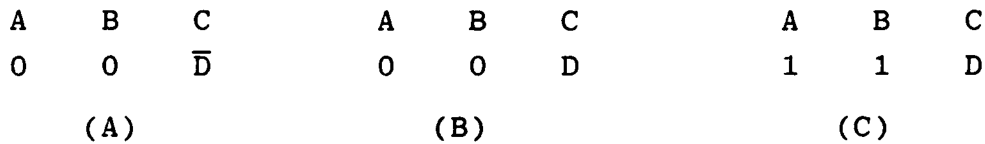


A	B	C
1	*	1
*	1	1
0	0	0

A	B	C
1	*	0
*	1	0
0	0	1

A	B	C
0	*	0
*	0	0
1	1	1

FIGURE 5.3-2 - SINGULAR COVER OF (A) OR GATE (B) NOR GATE (C) AND GATE, *=DON'T CARE



(A)

A	B	C
0	0	$\overline{D}$

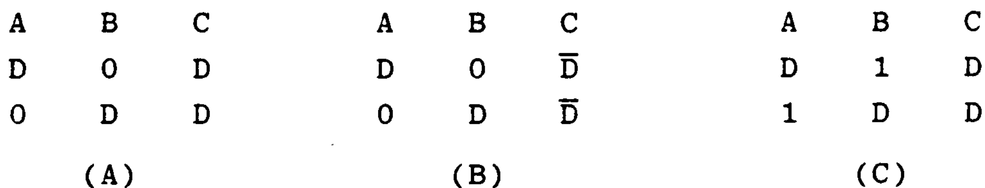
(B)

A	B	C
0	0	D

(C)

A	B	C
1	1	D

FIGURE 5.3-3 - PRIMITIVE D-CUBE OF (A) OR GATE (B) NOR GATE (C) AND GATE



(A)

A	B	C
D	0	D
0	D	D

(B)

A	B	C
D	0	$\overline{D}$
0	D	$\overline{D}$

(C)

A	B	C
D	1	D
1	D	D

FIGURE 5.3-4 - PROPAGATION D-CUBE OF (A) OR GATE (B) NOR GATE (C) AND GATE

Figure 5.3-3 illustrates the primitive D-cube of several common gates.

> Definition 5-4: A Propagation D-Cube is the set of D-Cubes which propagate a single fault at the inputs of a device to its output.

Figure 5.3-4 illustrates the propagation D-Cubes of those gates previously described.

Primitive D-Cubes can be intersected with propagation D-Cubes to generate test sequences by using the rules of D-algebra, Rules 1 and 2, presented in Table 5.3-1. In the rules of D-algebra, X represents a single element of a primitive D-cube and Y represents a single element of a propagation D-cube. Intersection of two D-cubes proceeds as indicated below.

<u>Intersection Procedure</u>

A. Starting with the first elements, intersect the elements of the two D-cubes using Rule 1 (of Table 5.3-1).
B. If a null (Φ), invalid, intersection is obtained for any element, re-initialize the intersection procedure starting with the first elements using Rule 2 (of Table 5.3-1).
C. If another null (Φ) intersection is obtained, the entire intersection of the two D-cubes is considered null.

As an example of this intersection procedure, the primitive D-cube (X) 11*DO$\overline{D}$ intersected with a propagation D-cube (Y)

TABLE 5.3-1

INTERSECTION RULES OF D-ALGEBRA

RULE 1: $X \cap Y$

		Y=PROPAGATION D-CUBE				
		D	$\overline{D}$	0	1	*
X=PRIMITIVE D CUBE	D	Φ	D	Φ	Φ	D
	$\overline{D}$	$\overline{D}$	Φ	Φ	Φ	$\overline{D}$
	0	0	0	0	Φ	0
	1	D	$\overline{D}$	Φ	1	1
	*	D	$\overline{D}$	0	1	*

RULE 2: $X \cap Y$

		Y=PROPAGATION D-CUBE				
		D	$\overline{D}$	0	1	*
X=PRIMITIVE D-CUBE	D	D	Φ	Φ	Φ	D
	$\overline{D}$	Φ	$\overline{D}$	Φ	Φ	$\overline{D}$
	0	0	0	0	Φ	0
	1	D	$\overline{D}$	Φ	1	1
	*	D	$\overline{D}$	0	1	*

$*10\bar{D}*D$ yields the D-cube $110D0\bar{D}$. The same primitive D-cube intersected with the propagation D-cube (Y) $*1*\bar{D}0\bar{D}$ yields a null intersection.

A D-cube can be intersected with a singular cover using the last three columns of Rules 1 and 2 where X is an element of the D-cube and Y is an element of the singular cover.

5.3.1.2 D-Algorithm Procedure

The D-algorithm procedure consists of the P-sensitizing algorithm and the consistency algorithm.

I. P-Sensitizing. The P-sensitizing algorithm is an algorithm for sensitizing a path from the faulty node to the output (or test) node. This algorithm consists of:

- First, choose a primitive D-cube of the fault and successively intersect it with the propagation D-cubes of the other devices along an arbitrary path (or paths).

- Then obtain the final D-cube where at least one output is specified.

- In the event a null intersection is encountered, repeat the P-sensitizing algorithm choosing a different path (or paths).
- If no path(s) can be found, it will be necessary to add intermediate outputs (test nodes) to the network and repeat the process.

II. Consistency Algorithm. The consistency algorithm is an algorithm used for backtracking from the device (gate) output to the inputs of the system to determine the required input states. This algorithm consists of:

- o First, intersect the final D-cube (from the P-sensitizing algorithm) with the singular covers of the devices utilized.
- o The intersection procedure is terminated when the required inputs are specified.
- o If any null intersections (inconsistency) exist, repeat the P-sensitizing algorithm along another path to obtain a new ensemble of D-cubes.

Example 3: The problem solved by path sensitizing in Example 1 (Figure 5.3-1) is repeated here using the D-algorithm. Table 5.3-2 shows the singular cover for Figure 5.3-1 while Table 5.3-3 shows the propagation D-cubes. Table 5.3-4 illustrates an unsuccessful attempt to sensitize the path BDE; a null intersection is obtained in the first step of the consistency algorithm. Table 5.3-5 demonstrates a successful P-sensitizing and consistency operation along path BF. The resulting test sequence is the same as that obtained previously, in Example 1.

TABLE 5.3-2 - SINGULAR COVER FOR FIGURE 5.3-1

		1	2	3	4	5	6	7	8	9	10
A	a	1	1			1					
	b	*	0			0					
	c	0	*			0					
B	d		0	0			1				
	e		*	1			0				
	f		1	*			0				
C	g			1	1			1			
	h			*	0			0			
	l			0	*			0			
D	m					0	0		0		
	n					1	*		1		
	o					*	1		1		
E	p							1	1	1	
	r							*	0	0	
	s							0	*	0	
F	t						0	0			0
	u						*	1			1
	v						1	*			1

TABLE 5.3-3 - PROPAGATION D-CUBES FOR FIGURE 5.3-1

		1	2	3	4	5	6	7	8	9	10
A	a_p	1	D			D					
	b_p	D	1			D					
B	c_p		0	D			$\overline{D}$				
	d_p		D	0			$\overline{D}$				
C	e_p			1	D			D			
	f_p			D	1			D			
D	g_p					0	D		D		
	h_p					D	0		D		
E	l_p							1	D	D	
	m_p							D	1	D	
F	n_p						0	D			D
	o_p						D	0			D

TABLE 5.3-4 - P-SENSITIZING & CONSISTENCY ALONG THE PATH BDE

P-Sensitizing	1	2	3	4	5	6	7	8	9	10
I	*	0	0	*	*	D	*	*	*	*
II = I $\cap$ g_p	*	0	0	*	0	D	*	D	*	*
III = II $\cap$ 1_p	*	0	0	*	0	D	1	D	D	*
Consistency										
IV = III $\cap$ g	*	0	ϕ	1	0	D	1	D	D	*

TABLE 5.3-5 - P-SENSITIZING & CONSISTENCY ALONG THE PATH BF

P-Sensitizing	1	2	3	4	5	6	7	8	9	10
I	*	0	0	*	*	D	*	*	*	*
II = I $\cap$ o_p	*	0	0	*	*	D	0	*	*	D
Consistency										
III = II $\cap$ 1	*	0	0	*	*	D	0	*	*	D

5.3.2 Error Detection in Microprogrammed Central Processors

The SAMSON processing elements are microprogrammed central processors and, as such, require special procedures since they are more than a collection of digital circuits, being a mixture of both hardware and "software" (namely, hardware and firmware).

The sophisticated SAMSON microprogrammed central processors are each a complex organization which utilizes advance digital circuits as well as advanced firmware techniques. Hence, error detection in microprogrammed central processors require the use of both the hardware path sensitizing technique (described above) as well as the firmware path sensitizing technique described below.

Firmware path sensitizing is application independent; firmware path sensitizing is a function of the hardware machine architecture and software machine architecture (instruction repertoire).

5.3.2.1 Firmware Path Sensitizing

The structure of a micro-sequence controller, of a micro-programmed machine, has the capability of controlling the micro address sequencing through all of the micro-code flow; the micro-code being the hardware equivalent of firmware (the micro-flow charting of the machine instruction repertoire). This micro-sequence controller, therefore, must have the ability to sequential address successive micromemory addresses as well as the ability to perform unconditional and conditional branches to arbitrary micromemory addresses as a function of various branch conditions.

For complete error detection within a microprogrammed central processor, it is essential that both the micro-sequencer hardware as well as the firmware (micro-code) branching within the various instructions (i.e., firmware paths) be exercised completely via appropriate test sequences.

This is accomplished by generating machine instruction sequences which exercise each micro address while path sensitizing each and every branch path. To accomplish this requires that those micro address locations which conditionally point to more than one location, be executed two or more times. No formal optimizing procedure has been developed for minimizing this test sequence; however, a formal procedure for developing and validating that the various branch paths are exercised has been developed.

Firmware Path Sensitizing Procedure:

1. Selection of a machine instruction to be sensitized.
2. The micro-flow chart, corresponding to the machine instruction, is examined to determine the number of conditional branches to be sensitized (and hence the minimal number of test sequences required).
3. For each conditional branch, a path from the selected branch point to the terminal micro-order is selected for sensitizing.
4. The inputs to the micro-instruction flow are selected so that the micro-program proceeds to the micro-instruction and micro-branch being sensitized.

Inputs are selected so that the wrong branch path or the wrong resultant is obtained as a result of any stuck-at-failures and furthermore the overall input test sequence executive distinguishes between these correct and incorrect outputs by verifying the results obtained.

In order to minimize the input test sequences associated with machine instructions having multiple micro-branch conditions, input sequences should be selected so as to maximize the number of branch paths exercised by the test vector.

The firmware path sensitizing procedure described herein has been utilized to develop a comprehensive set of test vectors for the SAMSON processor. The following sections describe the results obtained utilizing this procedure in the development of a comprehensive self-test procedure utilizing these test vectors.

5.4 ACHIEVABLE SELF-TEST ERROR DETECTION FOR A CPU

In the following paragraphs some general aspects of failure detection shall be examined for the purpose of exposing some of the difficulties involved in achieving near-perfect coverage. Specific attention is focused on comprehensive self-test coverage of a real time aerospace type central processor of the type suitable for the SAMSON system. This section emphasizes what is actually achievable through the use of self-test software and presents the results of a detailed self-test study, including data obtained on actual SAMSON hardware.

A major concern in real time (aerospace) computer applications (as well as in all computer applications) is the detection of failures in various (preflight and inflight) environments. Undetected failures in both computer and computer peripheral components can result in a significant reduction in mission reliability. Hence, failures must be detected with a coverage which is consistent with the reliability goals of the system. It has been shown that failure detection requirements are a function of the redundancy configuration and it is conceivable that the mission reliability goal, for a particular configuration, may require a (preflight) test efficiency of 99.9%, even with periodic testing [5.7].

In the following sections we examine some general aspects of failure detection, including an overview of well-known aspects of checking experiments and testing, for the purpose of exposing some of the difficulties involved in achieving near-perfect coverage. Specific attention is given to comprehensive self-test coverage of a SAMSON prototype PE. Failure detection requirements are not considered here, since this is highly application dependent; instead, emphasis is placed on what is actually achievable and by what means.

5.4.1 Modeling of Failure Effects

We take as our model, of the CPU, the <u>finite</u>, <u>sequential</u> machine [5.8] - [5.10]. We formalize the definition as follows:

Let $X = \{x_1, x_2, \ldots, x_m\}$ = set of inputs

$S = \{s_1, s_2, \ldots, s_n\}$ = set of internal states

$Y = \{y_1, y_2, \ldots, y_p\}$ = set of outputs

Then a finite sequential machine is the pair of functions f and g such that $y^i = f(x^i, s^i)$ and $s^{i+1} = g(x^i, s^i)$ where $x^i \varepsilon X$, $y^i \varepsilon Y$, $s^i \varepsilon S$ and the superscript, i, denotes the ith instant of time.

A machine with n states, m inputs and p outputs will be called an (n, m, p) machine. It can be shown that there are, at most, $\frac{(np)^{nm}}{n!}$ distinct (n, m, p) machines.

The finite sequential machine is a convenient model for representing the operation of a digital device, and a fortiori, for representing the effects of failures. Before proceeding further, we give the rationale for the present discussion. We start with a digital device which in its non-failed condition realizes a certain (n, m, p) sequential machine (M). If the machine fails, then it will behave like some other sequential machine (M'), not necessarily an (n, m, p) machine of the same order (the value of n, m and/or p may change). Because of the possible existence of these failures, M' may or may not be equivalent to M. It is the task of the diagnostician to make this determination; we place one restriction on the observer: internal states are not directly observable, the diagnostician is restricted to injecting inputs and observing the corresponding outputs. We can now define a failed machine.

Definition 5-5: Machine M' is a failed replica of machine M if and only if M' is not equivalent to M.

Thus far, we have not addressed the structure of the failed machine. To this end we make the following assumptions: (1) the set of inputs does not change, (2) the set of outputs does not change, and (3) the number of states does not increase.

The first two assumptions are relatively weak and impose minimum constraints on the failure modes. The third assumption is necessary in order to establish an upper bound on the number of (n, m, p) machines. This assumption appears to be reasonable and, in any case, is almost always invoked in the literature. (We note, in reference to this assumption, that, given an input sequence of any finite length there is a machine M', beginning in some state, which will yield the same output as M beginning in its initial state, provided we do not limit the number of states of M'.)

In summary we assume that a failed replica of an (n, m, p) machine is, again, a different (n, m, p) machine.

5.4.2 General Test Philosophies

There are several philosophies regarding testing of digital computers: (1) The computer is designed with dedicated additional hardware for the express purpose of detecting failures, usually through redundancy and comparison-type monitoring [5.11], [5.12], (2) Error detection coding of internal computer variables [5.13], [5.14]: these variables are coded in such a way that a failure or failures will very likely cause a recognizable change in the code, (3) Use of a self-test software program wherein the computed states (and variables) are tested against a stored table or a portion thereof [5.15], (4) Generate and output internal variables for comparison with similar variables in an identical computer (comparison-monitoring) [5.7], [5.16].

We will discuss here only the self-test philosophy assuming, as we do, that we are concerned at this point with a system consisting of a single processor which does not have special built-in failure detection capability.

5.4.3 Theoretical Requirements of Self-Test

In this section we examine the length of the input sequence necessary to completely test an (n, m, p) machine subject to the previously stated assumptions regarding the effects of failures.

For purposes of obtaining estimates we make the additional assumption that for each (n, m, p) machine, M', there is a set of component failures which will transform the original and non-failed machine, M, into M'.

It is undoubtedly true that the class of machines which are a replica of a failed machine is smaller than the class of all (n, m, p) machines. However, since we do not have sufficient information to significantly limit this class, we proceed on our assumption, which, in any case, presents the greatest difficulty to the diagnostician.

We now obtain a lower bound on the length of the input sequence required to test an (n, m, p) machine. The example is essentially due to Moore [5.8]. Consider a combination lock with combination $a_1, a_2, \ldots, a_{n-1}$, where each digit, a_i, could have assumed one of m values. Such a lock can be represented by an (n, m, 2) machine. The combination lock opens when the output equals unity and this can occur, starting in state s_1, if and only if the input sequence is $a_1, a_2, \ldots, a_{n-1}$. Now it is obvious that, in order to test the lock, it may be necessary to try all of the possible combinations [5.17] and that the number of combinations is m^{n-1}.

To illustrate the problems associated with the fault diagnosis of a sequential machine, we consider a random access memory (RAM). In order to fully appreciate the magnitude of the minimum length input sequence required, consider a typical 64-bit RAM, which is organized as 16, 4-bit words. The input is a 9-bit binary word (4 bits of which designate the input data, 4 bits the address, and a 1-bit input specifying read or write). The output is a 4-bit word. Thus, the RAM can be represented by a sequential machine with $n = 2^{64}$ states, $m = 2^9$ inputs, and $p = 2^4$ outputs. Here

we obtain $m^{n-1} = (2^9)^{2^{64}-1} \approx 2^{1.12x2^{67}} \approx 10^{3x10^{19}}$ as the minimum length of the input sequence required to test the device. The estimate of m^{n-1} is to be used when no advantage is taken of the unique structure of the device being tested. However, if the test (self-test) is designed for a particular device, it may be possible to do considerably better than m^{n-1}.

From this simple example we may conclude that an efficient and practical self-test must be designed to take advantage of the unique structure of each device being tested.

With regard to an upper bound on the length of the input sequence required (in view of the lower bound, the upper bound is of academic interest only), Moore [5.8] gives the estimate $\frac{n^{mn+2}p^n}{n!}$ for the "Moore" type machine.

5.4.4 Objectives of Self-Test Software

In the preceding section it was shown that a digital, sequential circuit could be represented by a sequential machine. The sequential machine representation leads to the conclusion that, if no advantage were taken of the unique structure of the device,

then the number of inputs required to completely test the device was so large as to render the test impractical. As a consequence, we must settle for something less than a complete test. We cite several factors which give cause for optimism: (1) The SAMSON processor consists of many types of combinatorial and sequential circuits whose inputs and outputs are directly accessible for fault-diagnosis, and, (2) Failure rates of hard-to-test failures and hard-to-test devices may be acceptably small.

It is generally agreed, although not universally accepted [5.18], that the most commonly encountered failures [5.19] are: (1) Stuck-at input or output bits, and (2) Stuck-at internal variables which prevent transitions to certain states (e.g., a stuck-at bit of a storage register).

The fault classes of primary concern, considered in this work, are the most commonly encountered failures: Input and Output Stuck-at-One and Stuck-at-Zero faults. Failures of this kind occur much more frequently than all other failures combined.

In a recent self-test software study [5.20] utilizing another self-test program, the effects of internal stuck-at-failures were investigated. That study utilized a gate level representation of the CPU to simulate the effects of these stuck-at-failures. The estimated difference in test coverage when the devices were simulated at the internal gate level as compared to the external level was less than a 1% improvement. On the basis of this result, limiting the class of faults to external stuck-at-faults is a reasonable restriction.

A complete test for these failures can be achieved by forcing each variable to the "1" and "0" state. It may be said that

the primary objective of this, and almost all, self-test algorithms is the detection of stuck-at-one and stuck-at-zero failures.

5.4.5 Central Processor Architecture

The central processor utilized in this study is organized as a microprogrammed parallel general purpose processor which operates on sixteen bit data words. The processor is constructed using significant quantities of large scale, LSI, and medium scale, MSI, integrated circuits thereby providing maximum computational capability in minimum size. The essential elements of the processor are shown in the block diagram of Figure 5.4-1; they include the: 16 general purpose operational registers, arithmetic unit, micro-control unit, input/output unit, program counter, specialized single bit indicators, control logic and timing unit.

The sixteen general purpose operational registers (accumulators) are operated upon primarily through the use of a powerful set of interregister instructions. Provisions are also made to utilize two of the registers as index registers during memory reference operations. In addition, sequential registers are automatically linked for double precision operations.

The arithmetic unit provides the capability to perform arithmetic and logical operations on the various machine registers and memory. It provides this capability through the use of an adder/shifter together with an arithmetic unit internal register. Information from the program counter, program memory, input/output unit and the sixteen general purpose operational registers is routed through the arithmetic unit under control of the micro-control unit.

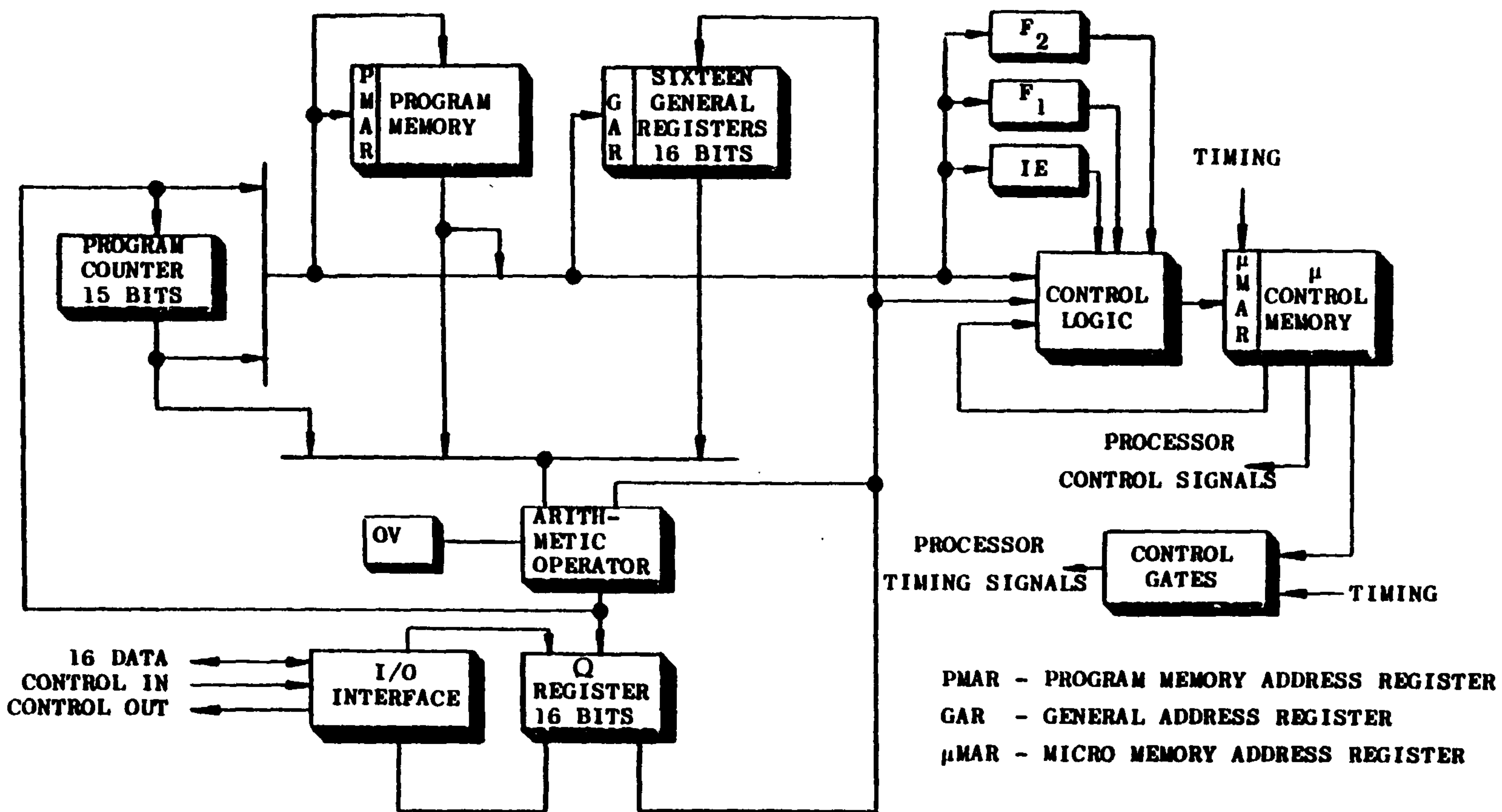


FIGURE 5.4-1
SAMSON PROTOTYPE PROCESSOR BLOCK DIAGRAM

The micro-control unit is the basic source of all processor control signals. The heart of this unit is a micro-control memory implemented with LSI semiconductor memories. The micro-control memory specifies which of the many possible machine operators are to be allowed during every phase of every machine instruction.

5.4.6 Development of Comprehensive Self-Test Software

To design an efficient and practical self-test one must take advantage of the unique structure of the processor being tested. This necessitates a thorough and detailed evaluation of the components, component interconnections, processor architecture and the micro-memory flow charts and microcoding of the machine. Only after such an evaluation is performed can a set of test vectors be generated to stress and exercise all of the above to the maximum extent in a reasonable length self-test program.

The primary purpose of the detailed examination of the components utilized is to determine an acceptable model of the internal structure and organization of these components. With this model available, the diagnostician can make an analytical and statistical determination of the effects of failures on the performance characteristics of individual components as well as groups of interconnected components. For the self-test software described here, component models were either obtained from the device manufacturer or developed from the component evaluation. (The simplest model which can be generated, assuming the manufacturer's model is either unobtainable or inadequate, is the NAND gate equivalent model.)

Utilizing the component models, a set of test vectors are generated to completely exercise not only the components but also the component interconnections, the processor architecture, the micro-memory flow and the instruction set. These test vectors are selected so that (wherever possible the internal nodes of the component model as well as) the external nodes of the component are forced to both the "1" and "0" state. In addition, these test vectors must be selected to exercise all of the component interconnections, processor architecture, all micro-memory locations and micro-memory branch paths and all of the machine instructions.

The criteria utilized to generate the appropriate set of test vectors is to determine a minimal set of test vectors that forces both internal and external variables to both the "1" and "0" state by controlling (only) the externally available variables to the appropriate values.

The test vectors obtained from the above procedure are then utilized in the self-test program to exercise and detect the class of faults under consideration (i.e., stuck-at-one and stuck-at-zero faults). To accomplish this, the self-test program must exercise the machine using these test vectors and assure the detection of these faults. Thus, the self-test software must be organized so that the intermediate variables as well as the final computed variables are verified for correctness. Verification of intermediate results limits the possibility of a single fault causing an undetected failure. In addition, the self-test program must include checks to assure that segments of the test program are not skipped (due to undetected failures). This requires execution verification of the various program segments. Similarly, the program code integrity must be assured; i.e., through the use of a memory sum check.

In addition to all of the above, the self-test program should provide fault isolation capability.

The resulting self-test program should then be optimized with respect to required memory size and execution time, while maintaining the self-test efficiency, so that it can be utilized in real time computer applications.

5.4.7 Self-Test Program

The self-test program consists of 24 blocks, 17 of which are test segments. Table 5.4-1 briefly describes the self-test program.

Test 1 is the bootstrap error detection test which partially tests those instructions used by the executive in error detection; they are only tested here to the extent necessary for minimal error detection.

In all test segments, no instruction is utilized (executed) within a segment unless that instruction has been previously tested in an earlier test segment (or unless that instruction is presently under test). In Tests 4 and 5 an internal sum is generated within the test and compared against a stored constant to assure complete execution of that segment.

The executive routine controls the program flow from test to test and generates an internal sum which is utilized to assure execution of each test segment. In addition, the executive provides fault isolation capability.

In the memory test a running check sum is generated for all memory locations utilized by the self-test program to assure the validity of the program being executed.

TABLE 5.4-1

DESCRIPTION OF SELF-TEST PROGRAM

BLOCK	TEST	BLOCK DESCRIPTION
1	-	Temporary Storage & Constants
2	-	Executive (Test Sequence Control)
3	-	Memory Test (Sum Check)
4&5	-	Program Control (Interrogation of Commands for Executive)
6	1	Bootstrap Error Detection Test Segment
7	2	Control & Skip on Indicator Instructions Segment
8	3	Complement Instructions & General Purpose Operational Registers A0 through A3 Segment
9	4	Immediate Instructions & Overflow Segment
10	5	Interregister Arithmetic Instructions & General Purpose Operational Register A4 through A15 Segment
11&12	6&7	Memory Reference Instructions (Base Addressing) Segment
13	8	General Purpose Operational Register Skip Instructions Segment
14	9	Logical Instructions Segment
15	10	Interregister "Move" Instructions Segment
16	11	Short Shift Instructions Segment
17	12	Long Shift Instructions Segment
18	13	Multiply Instruction Segment
19, 20 & 21	14(A, B,C)	Divide Instruction Segment
22	15	Memory Reference Instructions (Direct Addressing) Segment
23	16	Memory Reference Instructions (Indirect Addressing) Segment
24	17	Load/Store Instructions Segment

5.4.8 Self-Test Verification

A breadboard set-up was used for the purpose of validating this self-test software [9] on the processor hardware previously described. Because of the similarity of parts and the general structure of most single address mini-computers, the results of this study on the SAMSON (real time) processor are equally applicable to a wide class of processors and computers.

The self-test program tests: all internal CPU paths (buses), all 16 general purpose operational registers, the internal arithmetic register, limited testing of that portion of the main memory containing the self-test program, that portion of the program counter which is necessary to address the memory locations containing the self-test program, all arithmetic operators and all instructions (except I/O dependent instructions). It should be emphasized that this particular program was not designed to detect failures of the I/O and associated devices such as converters, multiplexers, I/O timing strobes, etc. This self-test program requires only 1,025 memory words* and 8,600 memory cycles to make one complete pass. At the rate of one microsecond per memory cycle, assuming a core memory (or two hundred fifty nanoseconds per memory cycle assuming a semiconductor memory, both conservative values), a complete pass requires 8.6 (or 2.15) milliseconds. If a fault is not detected, the program proceeds to a "GO" location; any other result indicates that a fault was detected. If a fault is detected and if the

*This is the number of actual memory words required; i.e., total self-test length minus unused reserved areas of memory.

processor can exercise sufficient control, the program proceeds to a "NO-GO" location and identifies the area of the failure.

5.4.9 Validaticn Philosophy

The validation was confined to a restricted class of failures which included grounded input and output nodes (stuck-at-zero failures). The eventual extension of the validation procedure to include the entire class of stuck-at failures was a paramount consideration; however, such testing might require destructive testing of several processors. Economic considerations precluded such destructive testing. Furthermore, it was understood that the present effort was the first step toward achieving this objective. Altogether, 350 pins, representing the entire complement of accessible nodes, were "failed". After each failure was injected the self-test program was initiated and the results tabulated. In the following paragraphs a brief description of the procedure and results is given.

The test was conducted by grounding all input and output nodes, one at a time. However, this did not result in the grounding of each individual input and output independently since, frequently, a single node fed two or more inputs via gating circuitry. As a result, the grounding of certain nodes actually resulted in the simultaneous failing of some inputs to a high (if the intervening gate was an inverter) and some inputs to ground.

It appears that, with expanded test facilities, this approach can be extended to include the following types of failures:

(1) Input and output nodes: stuck-at-one as well as stuck-at zero, and

(2) Common package failures - (a) device ground lead opening, (b) device V_{cc} lead opening, and (c) bridges (i.e., input to input, input to output, output to output).

There is another class of failures which are extremely difficult to simulate and, at present, no satisfactory method has been proposed for doing so. These failures are internal logic failures of the devices. Such failures result in either a stuck-at output failure or a restructuring of the internal state and transition branches. Internal failures resulting in stuck-at output failures are treated as described above. However, internal failures that cause restructuring of the device may not be seen at the output until a certain and unknown combination of inputs and internal states occurs. Since the resulting restructuring is unknown, simulation of such failures are extremely difficult.

5.4.10 Results

The self-test program used was designed to test all machine instructions except the I/O instructions and those skip instructions associated with external signals. All micro-memory words within the micro-memory are executed at least once except those associated with "power on", "console", "interrupt" and the above mentioned machine instructions.

Although the I/O bus (16 lines) is not explicitly tested by the self-test program, 13 or 14 of these nodes (as a function of when these nodes are failed) were detected as failures. The detection of these failures is due to the connection of the computer console during this validation procedure. The I/O bus is the data path to the computer console. This group of 16 signals should not be considered tested within the scope of this self-test program.

The four higher order program counter bits and the ripple carry into these program counter bits, represent those signals which could be included within the capability of this self-test program but are not. (Since four bit counter/register chips are used, only every fourth carry is available.) As previously indicated, such testing was not part of the original intent of this self-test program. It would be a relatively simple matter to add to or change the program to include coverage of these nodes; however, for the present, they illustrate a significant point.

The score card then reads (for the two card SAMSON processor):

	Control Unit	Arithmetic Unit	Total
Total Nodes Tested	193	185	378
Interrupt Nodes	3	1	4
Manual Halt Nodes	3	0	3
I/O Nodes	10	16	26
Valid Nodes Tested	177	168	345
Nodes Not Detected	0	5*	5*
+Efficiency (Node Ground Fault)	100%	97.0%	98.55%

*Program Counter-higher order bits and ripple carry.

As indicated, the upper program counter bits could be checked by adding to or modifying the self-test program to utilize these upper addresses. However, these results indicate the value of this program testing technique in developing effective self-test programs. If we had been unaware of the fact that these lines were untested, this approach would have identified these nodes as untested nodes.

+If all failures are equi-probable, then this quantity corresponds to test coverage.

Taking into account the original design objective of this self-test program, the score card actually reads:

	Control Unit	Arithmetic Unit	Total
Total Nodes Tested	193	185	378
Interrupt Nodes	3	1	4
Manual Halt Nodes	3	0	3
I/O Nodes	10	16	26
Higher Order Program Counter Nodes	0	5	5
Valid Nodes Tested	177	163	340
Nodes Not Detected	0	0	0
+Efficiency (Node Ground Fault)	100%	100%	100%

5.4.11 Summary of Test Validation Procedure

- The hardware validation procedure can be extended to include a large class of frequently encountered stuck-at failures.
- As conducted, the validation did not exercise the full potential of the self-test software. For instance, the self-test program checks the main memory by a memory sum test.
- Internal logic failures are difficult to simulate. Work in this area is described in a paper by R. L. Miga [5.21].
- Records of failure modes of digital devices must be maintained on a continuing basis, as they occur. Design defects should be distinguished from actual failures.

+If all failures are equi-probable, then this quantity corresponds to test coverage.

- Probabilities of device failures must be estimated from actual field data in order to accurately equate test efficiency to test coverage.
- From the above data, realistic failure modes of digital devices can be estimated. Failure modes with a high probability of occurrence must be recognizable by the self-test software.

6.0 FAULT TOLERANCE IN THE SAMSON DISTRIBUTED MULTIPROCESSOR NETWORK

A unique distributed multiprocessor network is discussed wherein each processing unit (PU, e.g. processing element, PE, plus memory) is identical and independent. The network configuration is that of co-equal processing units, i.e. no master-slave relationship exists. The SAMSON distributed multiprocessor network described herein is an integrated real-time parallel processing system. The basic architecture of the SAMSON distributed multiprocessor network results in an inherent fault tolerant organization.

6.1 INTRODUCTION

This section presents an overview of fault tolerant characteristics of the SAMSON [6.1] multiprocessor system which is a highly parallel experimental computer organization, utilized as a distributed multiprocessor network. The SAMSON distributed multiprocessor network is a unique real time processing system wherein each processing unit (PU) is identical and independent. The network configuration is that of co-equal PUs, cooperating in the performance of the overall system task. The SAMSON distributed multiprocessor network is capable of tolerating one or more transient or permanent failures and provides graceful degradation of performance even as the number of failures, m_f, becomes relatively large. This fault tolerant computing capability is provided in the SAMSON distributed multiprocessor network by its inherent ability for failure recognition, reconfiguration and recovery. Furthermore, these fault tolerant charac-

teristics are, to a great extent, independent of the fault tolerant characteristics of the individual PUs.

6.2 NETWORK ORGANIZATION

The SAMSON distributed multiprocessor is a network consisting of n PUs cooperating with one another to form an nth order network, wherein each PU is identical to, and independent of, every other PU in the network. The network configuration is that of multiple (n) co-equal partners, coordinating their activities in both the management and operation of the system as well as the solution of user related application tasks. The resulting organization is a fault tolerant network exhibiting graceful degradation of performance in the presence of single or multiple failures.

W.A. Curtin [6.2] indicated that ultrareliability meant that "no single element can perform so unique a function that its failure could disable the entire system, but rather should at most decrease the on-line capability". Despite this obvious fact, both special purpose processing elements and master control units have been and are still being incorporated within the structure of classical distributed networks and/or parallel processors. However, with the great potential for increased speed, reduced cost, reduced size, etc., available through the LSI and microprocessor microcircuit technology, this special purpose hardware no longer appears necessary (except in very specialized cases).

In the classical distributed network, a single point failure within either a master control unit or a special purpose processing element will either destroy the entire network or dramatically reduce its capability. The SAMSON distributed multiprocessor architecture, on the other hand, utilizes neither a master control unit nor any special purpose processing elements. This organization has thus eliminated one of the main fallacies of the classical distributed network; namely, the special purpose elements. Furthermore, a single point failure in an nth order SAMSON distributed multiprocessor network would, in general, only degrade the network by order 1.

In the presence of a single point failure, the nth order SAMSON distributed multiprocessor network is only degraded to order n-1; the network is a fail-soft fault tolerant network.

6.2.1 Characteristics of the PU

The SAMSON experimental system under development consists of four PUs (a 4th order network). The individual PUs, uniprocessors, are general purpose, parallel digital computers which provide the computational capability for both the network's operating system and a broad spectrum of user applications. The SAMSON uniprocessor features a full parallel sixteen bit arithmetic structure utilizing sixteen general purpose accumulators and a microprogrammed control unit; each uniprocessor has a

32K word (16 bits per word) memory capacity 6.3 .
The SAMSON prototype (PU) is constructed using conventional large scale, LSI, and medium scale, MSI, integrated circuits, thereby providing maximal computational capability in minimal size while providing a flexible hardware basis for experimentation, evaluation and concept validation.

Obviously, the network could be implemented so that each PU is itself a network of other PUs. Alternately, each PU could be a maxi-computer, a midi-computer, a mini-computer (its present status), a micro-computer (a micro-processor based PU has been developed as part of the continuing SAMSON development effort), or, for that matter, each PU could consist of multiple processing elements.

As an example of a PU consisting of multiple processing elements, consider an architecture wherein each PU is partitioned such that there is an input-output processor, or a separate input processor and a separate output processor, as well as a computational processor. As either an alternate partitioning or as an additional partitioning the operating system tasks can be segregated from the application tasks so that these tasks are performed in separate processors. With such an architecture, it might prove to be more cost effective to have the input and output processor(s) implemented as a microprocessor while implementing the computational processor as a mini-computer. In any event, for such an organization, it is both convenient and reasonable, although not essential, that the processing elements of the PU be software compatible.

Furthermore, each PU of the network could include fault tolerant features; i.e., error detecting and/or correcting codes [6.4] , [6.5] , arithmetic codes [6.5]-[6.7] , memory content and memory address verification [6.8] , [6.9] , voting [6.10] - [6.12] , sparing [6.13]- [6.15] , redundancy [6.11] , [6.16] , [6.17] , etc. Although such fault tolerant computing characteristics within the PU may enhance the overall fault tolerant characteristics of the network [6.11] , [6.18] , they are not essential to the SAMSON distributed multiprocessor architecture. The experimental system presently under development does not include such features; however, as part of the continuing development effort such attributes should be evaluated and it is expected that some of these features will be incorporated.

It should be noted that throughout the development of a fault tolerant computing system, it is essential that the architects provide adequate, but not excessive, fault tolerance. Furthermore, since these requirements may vary greatly from application to application, the computer architecture should be both flexible and modular, enabling the addition or deletion of such features as required by the specific applications.

Internal inter-PU communications is accomplished via uni-directional inter-PU communication links (ports). (See section 4 for an evaluation of various communication techniques.) The individual PUs have two dedicated internal output ports for transmitting inter-PU data; each PU outputs internal data on its own pair of

dedicated inter-PU output ports. In addition, each PU has n pairs of inter-PU input ports for receiving internal data. The alternative approach of using shared master, central, data bus [6.19] or buses was rejected for two reasons: (1) It would create access and control problems as well as transmission delays resulting from bus conflicts which are undesirable (and frequently intolerable in real time control applications); (2) It is definitely less fault tolerant than the approach utilized here. External communications is accomplished via the external I/O buses as a network option tailored to the requirements of the specific application.

In addition each PU has an interval timer, derived from its own master oscillator and a set of programmable watch dog timers.

6.2.2 Characteristics of the Network

The nth order SAMSON distributed multiprocessor network consists of n PUs, with 2n internal communication links for internal inter-PU communications and n_i I/O buses interfacing the individual PUs to their respective external subsystems. (The value of n_i is a function of the particular application requirements. In addition, the number of I/O buses per PU, n_p, is in general not equal to n_i and need not be constant for all PUs; i.e., $n_{pj} \neq n_{pk}$.) The PUs are externally interconnected to obtain the I/O function required by the particular application and to achieve the required fault tolerant characteristics for that computing system. An nth

order SAMSON distributed multiprocessor network is depicted in Figure 6.2-1. The 4th order network of Figure 6.2-2 corresponds to the experimental SAMSON distributed multiprocessor network being developed at The City College, The City University of New York. In this experimental network n_p is equal for all PUs (n_i=n_p=3).

We have previously indicated that in the presence of a single point failure, the nth order SAMSON distributed multiprocessor network is only degraded to order n-1. This degradation in performance need not exist; in an n out-of m order SAMSON network there would be no degradation of performance due to a single point failure. The SAMSON distributed multiprocessor network would simply function as an n out-of m-1 order network. The network would only degrade to order n-1 in the presence of m-n-1 failures.

6.3 NETWORK ARCHITECTURE

The SAMSON distributed multiprocessor network utilizes a distributed operating system, DOS, which is independent of the application program. The high degree of parallelism that exists within the network allows concurrent execution of various application tasks under the control of the DOS. The DOS, which exists in each PU, dynamically schedules the execution of these application tasks on the basis of a data driven [6.20] scheduler; that is, a task is enabled for execution only after the required operands have been provided, either by the

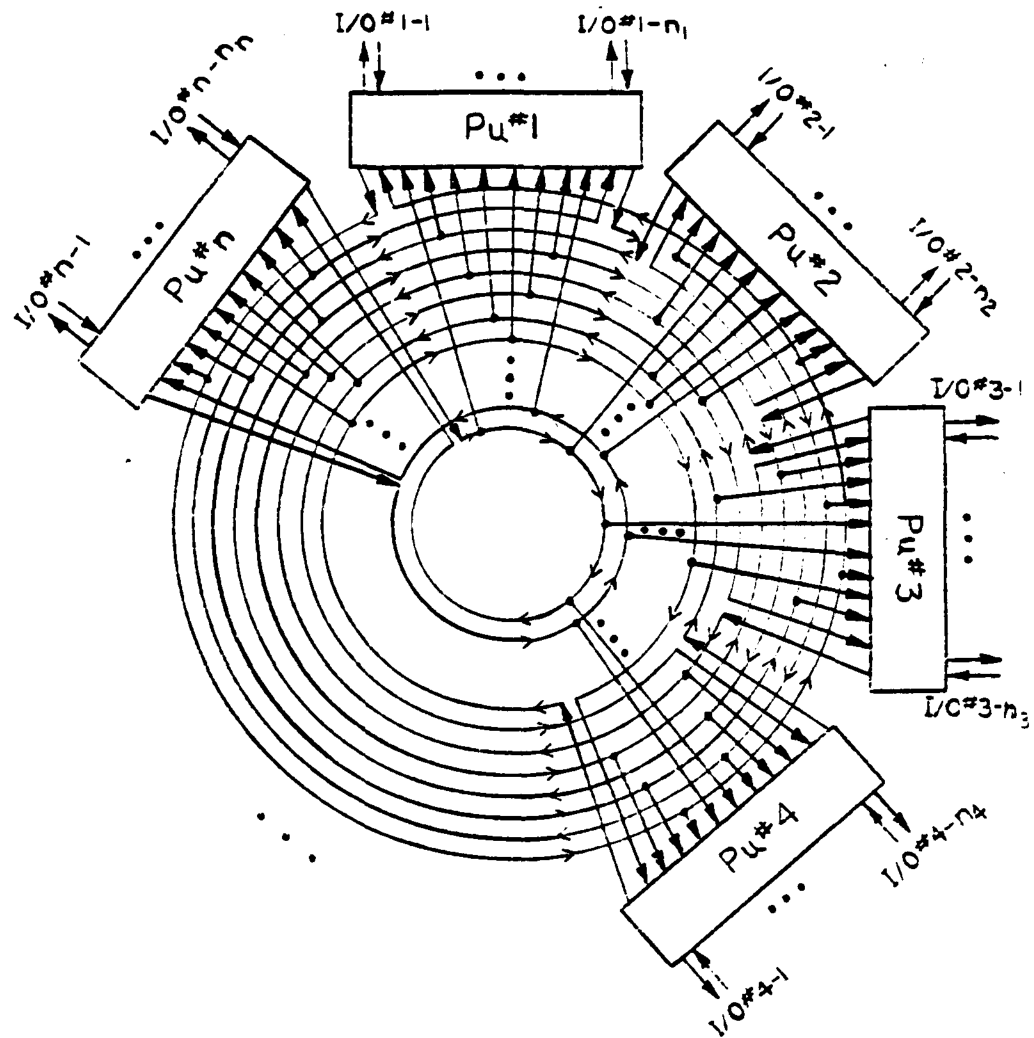


FIGURE 6.2-1
n^{th} - ORDER SAMSON
DISTRIBUTED MULTIPROCESSOR NETWORK

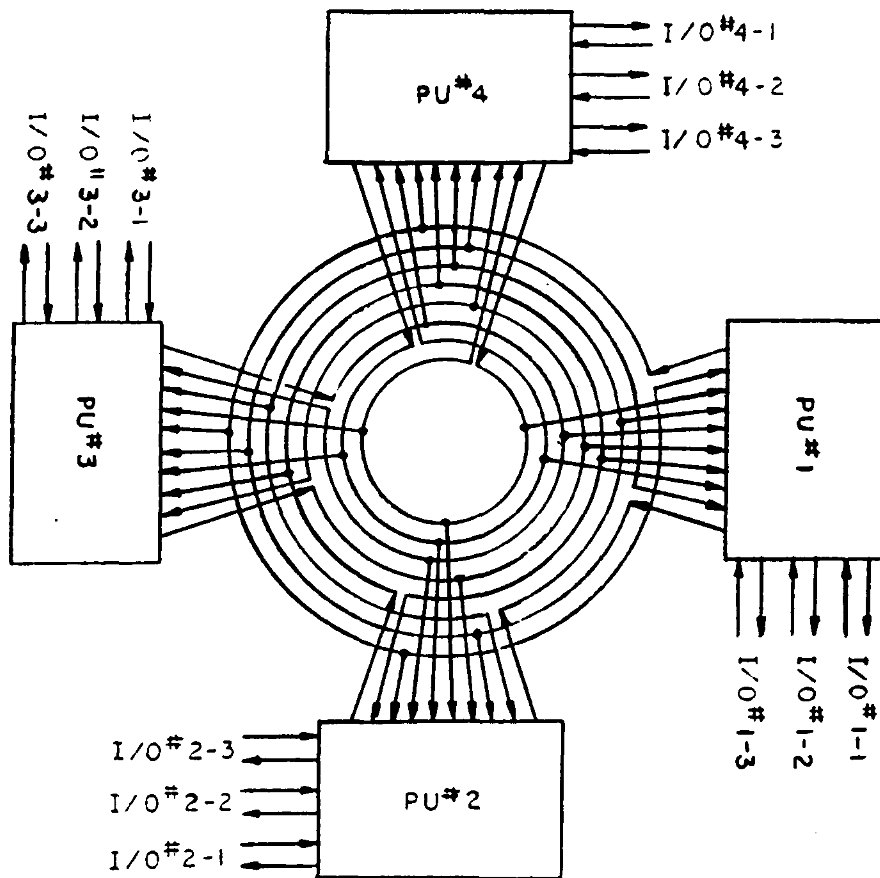


FIGURE 6.2-2

FOURTH ORDER SAMSON
DISTRIBUTED MULTIPROCESSOR NETWORK

execution of a predecessor task or by inputs from the I/O subsystem. Furthermore, the DOS places minimal constraints on the organization of application tasks. For example, the application tasks need not be structured to take advantage of the high degree of parallelism that exists in the SAMSON distributed multiprocessor network.

The DOS is responsible for initialization of the network, keeping track of the (health) status of the individual PUs, internal communications, monitoring the external I/O, scheduling tasks, resolving task conflicts, partially responsible for monitoring application tasks and is responsible for failure recognition, recovery and reconfiguration of the network.

System control is maintained by the DOS through messages transmitted on the inter-PU communication links. One such message is the "task initiate message". The PU initiating a task broadcasts this message to identify itself as the "application PU" which has accepted the task waiting and ready for execution. This message is a global message in that it is broadcasted to all participating PUs.

The "task initiate message" implicitly identifies the sending PU as the "application processor" and explicitly identifies the task, via a task identification number and sample number. In addition, the message contains error detection information.

Potential "application PUs" for a particular task are preassigned and conflicts resulting from simultaneous task initiations are dynamically arbitrated, on the basis of the PU priority number (PU number). The arbitrator is the DOS and arbitration is performed concurrently in the conflicting PUs.

Each task is preassigned to several PUs and, in general, a group of related tasks are preassigned to a cluster of PUs. It should be noted that the cluster need not consist of physically adjacent PUs. In fact, in specific applications, this non-physical adjacency of PUs would be a desirable fault tolerant feature; e.g., to avoid catastrophic failures in a fighter aircraft due to a single hit on the aircraft. In any event, the preassignment of each task to several PUs provides for flexible scheduling while enhancing the fault tolerance of the network. Although only a single copy of the task need be executed, provisions exist for multiple, simultaneous execution of the task. Consider the following classes of tasks: "normal" tasks, pseudo-critical tasks, critical tasks and "super-critical" tasks.

Definition 6.1: Normal Task - A task is defined as a "normal" task if the correct result is acceptable (actually a reasonable result is a more accurate description) in the presence of a single point failure in an interval of time approximating (exceeding at least) twice that required in a healthy environment.

The exact length of time required is a function of the specific application tasks being performed and the priority associated with reexecution of the task. It should be noted that the result is determined to be incorrect based on one or more of the following: the watch dog timer(s) monitoring the task have expired and the task is still incomplete, the task completes prematurely, the rate of change of the computed variable exceeds a specified (expected) limit, the result fails a reasonableness check. The recovery procedure associated with a fault during execution of a "normal" task is to roll-back and reschedule the task for reexecution by two or more PUs; one of the PUs which reexecutes the task is the original PU. If the new result obtained from the original PU is correct upon reexecution, the fault is identified as a transient fault. This procedure, of having the faulty PU re-execute the task, is utilized for all classes of tasks to identify transient faults. The DOS tabulates the number of transient faults occurring in each PU and declares a PU as faulty if the relative number of transient faults exceeds either a predetermined limit or a calculated weighted limit.

Definition 6.2: Pseudo-Critical Task - A task is classified as a pseudo-critical task if the correct result, in the presence of a single point failure, is acceptable in an interval of time approximately twice that obtainable without any failures; i.e.,

$$t_{spf} = t_{nf}(2+\varepsilon_1) \text{ where } \varepsilon_1 << 1 \qquad \text{(eq. 1)}$$

$t_{spf} \triangleq$ time for the correct result with a single point failure.

$t_{nf} \triangleq$ time for the correct result without any failures

In this case, the task is executed in at least two PUs (PU_i and PU_j) and the error is detected by comparing results. The correct result (or at least the reasonably correct result) can be determined by utilizing the mechanism previously discussed in regard to "normal" tasks. Alternatively, a roll-back procedure can be used where the task is rescheduled and reexecuted. In any event, the faulty PU (PU_i) reexecutes the task and the result is rechecked by PU_j by comparing the new PU_i result with the PU_j result. Transient faults are treated as above.

Definition 6.3: Critical Task - A task is classified as a critical task if the result must be correct in the presence of a single point failure and furthermore, the result is required in the presence of this failure within an interval of time approximating that obtainable without any failures; i.e.,

$$t_{spf} = t_{nf}(1+\varepsilon_2) \text{ where } \varepsilon_2 << 1 \qquad \text{(eq. 2)}$$

In this case three PUs (PU_a, PU_b, PU_c) execute the task. A 2 out of 3 majority volting mechanism is utilized here.

(If PU_a had been at fault, PU_a might identify both PU_b and PU_c as in error, PU_b would identify only PU_a, likewise PU_c would only identify PU_a. Each of these PUs would broadcast this information to the network and the other PUs which utilize the result(s) of this task would determine that PU_a were in error; they would ignore PU_a's result(s) as well as its error broadcast. Re-execution by PU_a utilizing stored past data values as well as new data values can be utilized to determine whether a transient fault or a data dependent fault had occurred.)

Definition 6.4: Super-Critical Task - A task is classified as a "super-critical" task if the result must be correct in the presence of multiple (m_f) simultaneous failures and furthermore, this result is required in approximately the same time as that obtainable without any failures; i.e.,

$$t_{mf} = t_{nf}(1+\varepsilon_3) \text{ where } \varepsilon_3 \ll 1 \qquad \text{(eq. 3)}$$

$t_{mf} \triangleq$ time for the correct result with m_f simultaneous failures

Simultaneous failures are considered here as those occurring within the interval of time required to perform the "super-critical" task. The number (m) of PUs required for a "super-critical" task is $m=2m_f+1$.

It should be noted that the judicious use of the appropriate class of tasks results in an improved overall throughput for the SAMSON distributed multiprocessor network. This feature makes the SAMSON network ideally suited for such applications which have heretofore utilized massive redundancy techniques; e.g., triple redundancy, dual-dual, etc. In such massive redundant systems, these systems have utilized several processors configured so that the entire job is processed by each processor. Both the hardware utilization and throughput is extremely inefficient, as compared with the total hardware and throughput available, since many of the processing tasks within the entire job need not be redundant, or do not need the same degree of redundancy as that required by a very limited number of processing tasks.

Equation 4 expresses the degradation in overall throughput for both the conventional redundant system and the SAMSON distributed multiprocessor network, as compared to that obtainable with the ideal system.

Degradation: $(N-1)\ T$ Conventional System (eq. 4a)

$(N-K)\ T$ SAMSON System (eq. 4b)

where:

- T is the throughput (or measure of the computational power of each processor)
- N is the number of processors in the conventional redundancy configuration
- K is the total number of tasks; here a task is considered one which utilizes all the available real time of the PU (the task itself may consist of multiple subtasks).

FIGURE 6.3-1

PERFORMANCE DEGRADATION COMPARED WITH IDEAL NORMALIZED WITH RESPECT TO THROUGHPUT

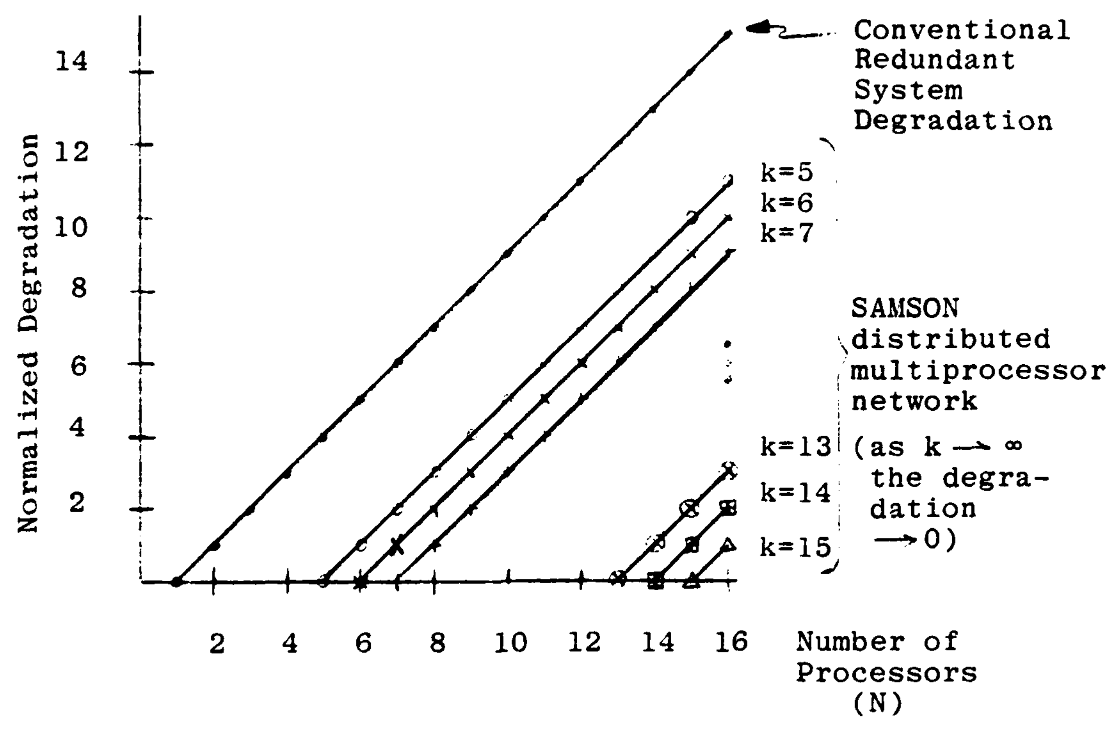
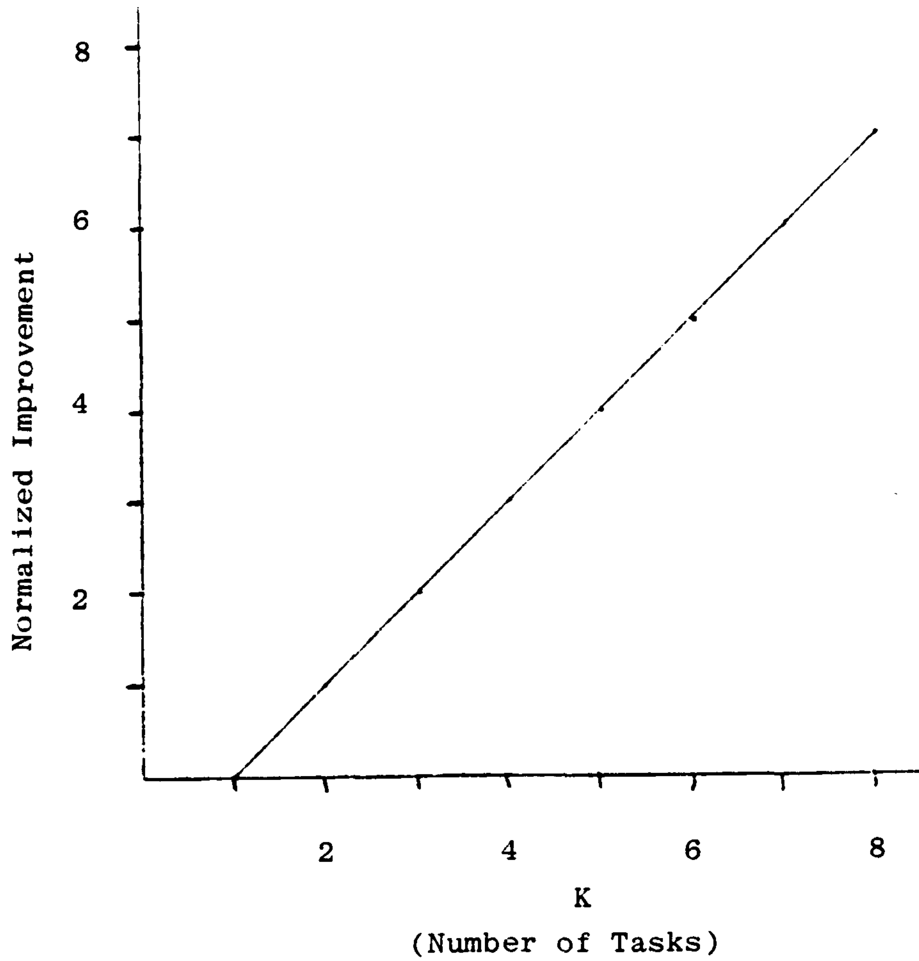


FIGURE 6.3-2
SAMSON THROUGHPUT IMPROVEMENT
OVER THE CONVENTIONAL SYSTEM
(NORMALIZED WITH RESPECT TO THROUGHPUT)

Improvement = (K-1) T



6.4 DISTRIBUTED OPERATING SYSTEM (DOS)

The DOS knows a priori all input parameters required by a task as well as the task classification. When all of the parameters required for execution are available, the DOS dynamically schedules the task for execution on the appropriate number of PUs, from those PUs having task copies. The "application PU(s)" assigned to execute the application task broadcasts a "task initiate message" when the task is initiated; likewise, a "task complete message" is broadcast upon completion of that task. The initiate message when received by a (task) monitor PU, causes the monitor PU to record in its own local memory the value of its own interval timer as well as the application task identification and the "application PU" number. In addition, a programmable watch dog timer is started by the monitoring PU. (This timer is of both lower resolution and accuracy than the interval timer.) Upon receipt of the "task complete message" assuming it is forthcoming, each of the monitoring PUs determines the present value of their interval timer and independently calculates the corresponding task processing interval, t_{TPI_i}, verifies that this value is within the acceptable bounds (both the upper and lower limit) previously established and recorded in the monitoring PUs local memory. If the value of t_{TPI_i} falls outside the recorded bounds (plus the tolerance for asynchronous operation), the monitoring PU(s) generate a global error interrupt and broadcasts an "error message" indicating that it has identified the "application PU" as faulty. If the "application PU" fails to generate a "task complete message", the monitoring PU watch dog timer will elapse and the monitoring PU

will likewise globally announce the "application PU" as faulty. It should be noted that, in addition to the other PUs, the "application PU" always monitors itself. Although the previous discussion dealt with an "application PU", the same situation holds true for a PU performing a network control task.

When a monitor PU determines that a PU is faulty, it takes no action other than globally announcing this fact. Each PU of the network upon receiving this information determines for itself the health status of the other PUs. It should be noted that a faulty PU need not be disconnected from the network since all messages from that PU are ignored. Ignoring the faulty PU rather than disconnecting it eliminates the need for switching networks otherwise not needed for normal system operation. It should be noted that in an n out-of m network utilizing unpowered spares such switching capability would exist and in such a network, the faulty PU could be disconnected. This is done here since no additional special hardware is required, furthermore, the power dissipation of the network is minimized, since all unnecessary activity is eliminated, and the network is easier to manage. Scheduling is done on the basis of a data driven task scheduler having a priori knowledge of the required input parameters.

Since the computer utilizes high speed random-access memory, program swapping and dynamic memory allocation are not required. If read only memories are utilized for the program memory, a program is loaded into a

working partition through power switching techniques; i.e., powering the selected segment and unpowering the previously selected segment. Once a program is "loaded" into a working partition, it remains there until a new program is requested by the DOS. In the event of a failure by one of the participating PUs, the DOS can reassign a healthy PU performing a lower priority task to the higher priority task being performed by the failed PU.

Since the possibility exists that a task may be interrupted at any time, due to the detection of a fault, all programs require special provisions for re-start after recovery from that fault. After completion of the system repair procedures, the PU performing a task may have been replaced; all register contents must be assumed to be lost as a result of the failure, thus necessitating periodic saving of the processor state vector (the multilevel buffering technique utilized for the roll-back procedure).

When roll-back is required, a task may have been performing one of three types of activities [6.21]: arithmetic computations, input-output or nonrepeatable events. The recovery procedure is different for each of these cases. If an arithmetic computation were being performed, the recovery action is to simply reestablish the working space in a healthy PU from the last valid processor state vector which was recorded at the established point of roll-back. If an input-output operation were active, the recovery is

slightly more complex. Here, the input-output media, including communications, must be reestablished to the point of roll-back before the processor state vector can be activated in the newly assigned PU. Execution of non-repeatable events can be the most complex activity when considering recovery. In some time-critical applications, the entire sequence may be aborted; this would correspond to a "wave off" on landing of an aircraft. In such applications, the recovery procedure must first access the user program to determine the proper recovery procedure in lieu of the application data. (Recovery procedures would be different as a function of position, altitude, etc.) Other time-critical applications necessitate the completion of the entire sequence once having passed a critical point in the procedure; this corresponds to the final stage of a limited control spacecraft landing. In such an application multiple PUs would be performing the task; upon detection of a failure within one of the PUs one (or more) additional PU(s) would be initialized to the present state of one (or more) of the functioning PUs and would start processing from that PSW (processor status word) state.

6.5 ADDITIONAL FAULT TOLERANT FEATURES

In addition to the above mentioned fault tolerant characteristics, the SAMSON distributed multiprocessor network has other such characteristics. The following paragraphs briefly cover these attributes.

The DOS monitors external I/O communications and verifies that data received by all participating PUs

is consistent. The input data is checked for reasonableness, where possible, by checking the rate of change of input variables, checking against limit values for these variables, etc. In addition external input data is exchanged between participating PUs to verify that the exact data is available in each PU; thus, multiple PUs can execute the same task, and in the absence of failures obtain identical results. This feature facilitates verification of proper PU performance.

Dynamic checking is thus provided at the interfaces. Furthermore, the interfaces provide a convenient location for error checking, including the use of error detecting and correcting codes. The dynamic checking at the interfaces tend to minimize the amount of circuitry required while preserving the data integrity essential for reliable recovery.

Additional fault tolerance results from the network's ability to perform related tasks simultaneously. Hence, complementary tasks (tasks implemented through independent procedures) can be processed concurrently; the results from these two tasks can be compared to detect faults. The complementary tasks can both be highly accurate or one can simply be a fast rough approximation of the other. It should be noted that this procedure detects both hardware as well as software malfunctions. These malfunctions can be either permanent and/or transient hardware errors or generic software errors. Such hardware errors are detectable and correctable through the monitoring, rollback and recovery procedures previously described. Generic errors, although

detectable, are not correctable; this ambiguity cannot be resolved in a real time environment. At this time, no clear solution (other than triple complementary execution) exists to this dilemma. The complementary task feature (assuming a fast rough approximation is utilized) provides a means whereby all tasks, or at least a large majority, can be processed despite the need for degraded performance in the presence of a relatively large number of failures.

Internal as well as external communications include error checking information. Internal communications is via the redundant inter-PU links, which are feedback to the transmitting PU to verify the integrity of the transmission. "Shared data variable" messages are communicated between participating PUs via the inter-PU links. Reasonableness checks are performed on shared variables in the same manner described above for input data. External communications is redundant, as required by the application. All output data is echoed to verify the integrity of these transmissions. As a function of the application, inputs can also be echoed to verify their integrity.

Self-test procedures [6.22] (see previous section) are periodically executed in each PU as a background task, to ensure the health of the overall network. The self-test procedure is partitioned into segments allowing execution of the overall procedure to progress segment by segment, as background processing time is available. In addition, health checking procedures

are executed and the results verified by another PU whenever the health of a PU is in question.

Fault recognition, recovery and reconfiguration are under the direct control of the DOS. Fault recognition is jointly accomplished by the DOS and the hardware facilities of the network. Fault recognition is reported by the DOS using global error messages. Recovery [6.23] from a fault is accomplished through rollback, rescheduling and reexecution of the task. The DOS saves the present as well as the most recent previous value(s) of each significant variable to facilitate rollback and reexecution procedures. Finally, reconfiguration is accomplished by the DOS wherein all PUs participate in the adaptive majority voting [6.12] mechanism which decides which PUs are failed or non-failed. These functions are implemented by the DOS P and R matrix. The P matrix contains each PU's opinion of the status of the other PUs and the majority opinion of each PU. The decision regarding PU_i is derived from the opinions of all other PUs, from the transient error record of PU_i, self-test results from PU_i and the health checks performed on PU_i. The P matrix decisions are utilized by the R matrix to decide which PU requests are to be ignored (or in the case of an n out-of m, with unpowered spares, which is to be turned-off). Once a PU has failed (determined by the P matrix), all activity by that PU is ignored using the R matrix results.

6.6 SUMMARY

The SAMSON distributed multiprocessor network possesses a high degree of fault tolerance, enabling it to achieve graceful degradation of performance despite permanent or transient failures. The network can even handle multiple failures without degrading system performance if it is configured as an n out-of m network.

The usual problems associated with processor switching and memory/processor interconnections are avoided through the use of an interconnection network which provides a high degree of inherent parallelism. In addition, the problems associated with central control units, special purpose processing elements and shared data buses are eliminated in the SAMSON distributed multiprocessor network.

Dynamic error detection and buffering of data enables expeditious and exact reruns of tasks. These features facilitate recovery of the system as well as reconstruction of the information necessary for continued operation. Furthermore, the distributed nature of the operating system, the use of a dynamic data driven scheduler and the ability for rapid reconfiguration, using the P and R matrix information to simply ignore a failed PU, results in a flexible fault tolerant system.

7.0 CONCLUSIONS

In this research work several major problems have been addressed, including:

- The architectural design, development, analysis and evaluation of a high speed, Multiple-Instruction Single Data (MISD), data driven, Simultaneous Multiprocessor Organization (SAMSON), including the detailed design, implementation, analysis and evaluation of the Processing Elements (PEs);
- Developing web structures for the computation of: the summation of n numbers, the product of n numbers, the powers of x, and polynomials of degree n;
- Characterizing and evaluating the throughput performance of multiprocessor interconnection networks;
- Achievable Error Detection; and
- Development of a distributed fault tolerant network.

7.1 RESULTS

A SAMSON data-driven multiprocessor system was developed in this work which has been shown to improve the computational execution time by as much as 635% (over that obtainable with a uniprocessor) while maintaining an overall hardware utilization of over 90%. In addition, the system demonstrated the desired decreasing execution time (with the increasing number of PEs) even for programs written for uniprocessor execution.

Likewise, the web results obtained in this work met the desired goals. Specifically, the web results for a polynomial of degree n are better (in terms of execution speed) than those obtained with the k-th order Horner's Rule, Estrin's Method, Tree Method or Folding Method.

The throughput performance analysis of the interconnection networks studied in this work supported the original research proposal assumptions that a timed-shared bus would meet the needs of the SAMSON system. In fact, the results obtained show that below bandwidth limiting there is little degradation in its performance as compared with the ideal network. These results agree with those obtained by Danielsson and Gudmundsson [7.1].

Furthermore, achievable error detection (through self-test software) of 100% within the constraints described (see Section 5.4.10) was obtained and verified for the Processor Element through actual testing on a SAMSON PE. Finally, a distributed fault tolerant network architecture was proposed having (k-1) times the throughput of a conventional Triple Modular Redundant (TMR) system (where k is the total number of tasks).

7.2 SUGGESTED FUTURE RESEARCH

Future research in the area of multiprocessing and fault tolerance within such networks is limited only by the imaginativeness of the researcher. Potential areas for future research include:

- Optimal Scheduling Strategies for Multiprocessors
- Dynamic Multiprocessor Scheduling
- Development of Techniques for Recognition of Tasks which can be Processed in Parallel
- Development of the Requirements for, as well as the Development of, an Efficient SAMSON Operating System
- Development of a Detailed Distributed Operating System (DOS) for the Fault Tolerant SAMSON Network

- Performance Evaluations of Fault Tolerant Multiprocessor Systems
- Performance Evaluation of Programs written for Parallel Processor Execution
- Development of a Detailed Queuing Model of the SAMSON system

BIBLIOGRAPHY

Section # . Reference # Reference

1.1 Flynn, M.J., Very High-Speed Computing Systems, Proceedings of the IEEE, Vol. 54, Dec. 1966.

1.2 Higbie, L.C., Supercomputer Architecture, Computer (IEEE Computer Society), Vol. 6, No. 12, Dec. 1973.

1.3 Baer, J-L., Multiprocessing Systems, IEEE Transactions on Computers, Vol. C-25, No. 12, Dec. 1976.

2.1 Unger, S.H., A Computer Oriented Toward Spacial Problems, Proc. IRE, October, 1958.

2.2 Slotnick, D.L., Borch, W.C., McReynolds, R.C., The Solomon Computer - A Preliminary Report, Proc. 1962 Workshop on Computer Organization, Washington, D.C.: Spartan, 1963.

2.3 Crane, B.A., Githens, J.A., Bulk Processing in Distributed Logic Memory, IEEE Transactions on Electronic Computers, Vol. EC-14, April, 1965.

2.4 Slotnick, D.L., The Solomon Computer, Fall Joint Computer Conference, 1962.

2.5 Murtha, J.C., Highly Parallel Information Processing Systems, Advances in Computers, 1966, 7.

2.6 Burroughs Corp., ILLIAC IV Systems Characteristics and Programming Manual, NASA CR-2159, Feb., 1973.

2.7 Bouknight, W.J., The ILLIAC IV System, Proceedings of the IEEE, Vol. 60, 4, April 1972, 369-388.

2.8 Flynn ,M.J., Podvin, A., Shared Resource Multiprocessors, Computer (IEEE Computer Society), March/April, 1972.

2.9 Comfort, W.T.,, A Modified HOLLAND Machine, Fall Joint Computer Conference, 1963.

2.10 Bell, G.C. and Wulf, W.A., C.mmp - A Multi-mini-processor, Fall Joint Computer Conference, 1972.

2.11 Bell, G.C. and Freeman, P., C.ai - A Computer Architecture for AI Research, Fall Joint Computer Conference, 1972.

2.12 Misunas, D. P., Performance Analysis of a Data-Flow Processor, Proceedings of the 1976 International Conference on Parallel Processing, August, 1976.

2.13 Rumbaugh, J., Data Flow Languages, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August, 1975.

2.14 Rumbaugh, J., A Data Flow Multiprocessor, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August, 1975.

2.15 Dennis, J.D., Packet Communication Architecture, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processings, August 1975.

2.16 Misunas, D.P., Structure Processings in a Data-Flow Computer, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August, 1975.

2.17 Schroeder, M.A., Meyer, R.A., A Distributed Computer System Using a Data Flow Approach, Proceedings of the 1977 International Conference on Parallel Processing, August, 1977.

2.18 Gurd, J., Watson, I., A Multilayered Data Flow Computer Architecture, Proc. of the 1977 Int. Conf. on Parallel Processing, August, 1977.

2.19 Harris, J.A., Smith, D.R., Hierarchical Multiprocessor Organizations, 4th Annual Symposium on Computer Architecture, March, 1977.

2.20 Danielsson, P.E., Gudmundsson, B.; Time-Shared Memory-Processor Interface, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August 1975.

2.21 Wensley, J.H., SIFT - Software Implemented Fault Tolerance, Fall Joint Computer Conference, 1972.

3.1 Koczela, L.J., The Distributed Processor Organization, Advances in Computers, 1968, 9, 283-353.

3.2 Kuck, D.J., "Parallel Processor Architecture - A Survey", 1975 Sagamore Conference Proc., pp. 15-39.

3.3 Baer, J.L., and D.P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computations", Proc. IFIP Congress 1968, pp. 340-346.

3.4 Stone, H.S., "One Pass Compilation of Arithmetic Expressions for A Parallel Processor", Comm. A.C.M., Vol. 10, No. 4 (April 1967), pp. 220-223.

3.5 Muraoka, Y., "Parallelism Exposure and Exploitation in Programs", PH.D Dissertation, Dept. of Computer Science, Univ. Illinois, Urbana, Ill., Feb. 1971.

3.6 Hsu, T.T., "On Parallelism, Scheduling and Data Communication in Parallel Processing Systems", Ph.D. Dissertation, Dept. of Elect. Eng., The City College of The City University of New York, August 1976.

3.7 Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing", Computer Surveys, Vol. 5, No. 1 (March 1973), pp. 31-80.

3.8 Brent, R., Kuck, D., Maruyama, K., "The Parallel Evaluation of Arithmetic Expressions Without Division", IEEE Trans. Comput., Vol C-22, No. 5 (May 1973), pp. 532-534.

3.9 Tjaden, G.S., "Hierarchical Properties of Concurrency", Proceedings of the 1976 International Conference on Parallel Processings, pp. 55-64.

3.10 Ramamoorthy, C.V., Leung, W.H., "A Scheme for the Parallel Execution of Sequential Programs, "Proceedings of the 1976 International Conference on Parallel Processing, pp. 312-316.

3.11 Bernstein, A.J., "Analysis of Programs for Parallel Processing", Trans. IEEE Vol. EC-15, No. 5 (Oct. 1966), pp. 757-762.

3.12 Bährs, A., Operation Patterns (An extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972.

3.13 Dennis, J.B., First Version of a Data Flow Procedure Language. Symposium on Programming, Institut de Programmation. University of Paris, Paris, France, April 1974, 241-271.

3.14 Kosinski, P.R., A Data Flow Programming Language. Report RC 4264, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., March 1973.

3.15 Kosinski, P.R., A Data Flow Language for Operating Systems Programming. Proceedings of ACM SIG PLAN - SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September 1973), 89-94.

3.16 Misunas, D.P., Performance Analysis of a Data-Flow Processor, Proceedings of the 1976 International Conference on Parallel Processing, August, 1976.

3.17 Giroux, E.D., A Large Mathematical Model Implementation on the Star-100 Computers, Symposium on High Speed Computer and Algorithm Organization (proceedings to appear).

3.18 Yu, M.L. and Saxema, A.M., Coherent A.C. Josephson Effect in a Bulk Granular Superconducting System, Brookhaven National Laboratory, New York, September, 1974.

3.19 Stotts, L.B., High Speed Optical Matrix Multiplier System, Department of the Navy, Washington, D.C., May 1975.

3.20 Ramseyer, R.R. and Van Dam, A., A Multi-Microprocessor Implementation of a General Purpose Pipelined CPU, The Fourth Annual Symposium On Computer Architecture, March 1977.

3.21 Rau, B.R. and Rossman, G.E., The Effect of Instruction Fetch Strategies Upon the Performance of Pipelined Instruction Units, The Fourth Annual Symposium on Computer Architecture, March 1977.

3.22 Wilkes, M.V., "The Best Way to Design an Automatic Calculating Machine", Proceedings of the Manchester University Computer Inaugeral Conference, London: Ferranti, July 1951.

3.23 Hallin, T.G., and Flynn, M.J., "Pipelining of Arithmetic Functions", IEEE Transactions on Computers, Vol. C-21, No. 8, August 1972.

3.24 Maekawa, M., and Boyd, D.C., Two Models of Task Overlap Within Jobs of Multiprocessing Multiprogramming Systems, Proceedings of the 1976 International Conference on Parallel Processing, August, 1976.

3.25 Ramamoorthy, C. V., and Li, H. F., Pipelined Processors - A Survey, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August, 1975.

3.26 Kogge, P. M., The Microprogramming of Pipelined Processors, The Fourth Annual Symposium on Computer Architecture, March 1977.

3.27 Skinner, C., Asher, J., Effect of Storage Contention on System Performance, IBM System Journal, Vol. 8, No. 4, 1969.

3.28 Bhandarkar, D.P., Analysis of Memory Interference in Multiprocessors, IEEE Transactions on Computers, Vol. C-24, No. 9, Sept. 1975.

4.1 Siegel, H.J., Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, Aug. 1975.

4.2 Nisnevich, L., and Strasbourger, E., Decentralized Priority Control in Data Communications, The 2nd Annual Symposium on Computer Architecture, Jan., 1975.

4.3 Franchi, P., Distribution of Functions and Control in PPCNET, The 3rd Annual Symposium on Computer Architecture, Jan., 1976.

4.4 Siegel, H.J., Single Instruction Stream - Multiple Data Stream Machine Interconnection Network Design, Proceedings of the 1976 International Conference on Parallel Processing, Aug., 1976.

4.5 Feug, T., Data Manipulation Functions in Parallel Processors and Their Implementation, IEEE Trans. on Computers. Vol. C-23, No.3, March, 1974.

4.6 Gecsei, J., Interconnection Networks from Three-State Cells, IEEE Trans. on Computers, Vol. C-26, No. 8, Aug., 1977.

4.7 Wulf, W.A., and Bell, C.G., C.mmp - A Multi-Mini-Processor, AFIPS, FJCC Proceedings, Vol. 41, Part II, 1972.

4.8 Hoogendoorn, C.H., Reduction of Memory Interference in Multiprocessor Systems, The 4th Annual Symposium on Computer Architecture, March, 1977.

4.9 Anderson, G.A., Jensen, E.D., Computer Interconnection Structures: Taxonomy, Characteristics and Examples, ACM Computing Surveys, Vol. 7, No. 4, Dec. 1975, pp. 197-213

4.10 Davidson, I.A., and Field, J.A., Design Criteria for a Switch for a Multiprocessor Computer System, 1975 Sagamore Computer Conference on Parallel Processing, Aug. 1975.

4.11 Bagai, I.A., and Lang, T., Reliability Aspects of Illiac IV Computer, Proc. of the 1976 International Conf. on Parallel Processing, Aug., 1976.

4.12 Strecker, W.D., Analysis of the Instruction Execution Rate in Certain Computer Structures, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburg, Pa., 1970.

4.13 Bhandarkar, D.P., Analysis of Memory Interference in Multiprocessors, IEEE Transactions on Computers, Vol. C-24, No. 9, Sept., 1975.

4.14 Misunas, D.P., Performance Analysis of a Data-Flow Processor, Proc. of the 1976 International Conf. on Parallel Processing, Aug., 1976.

4.15 Swanson, R.C., Interconnections for Parallel Memories to Unscramble p - Ordered Vectors, IEEE Transactions on Computers, Vol. C-23, No. 11, Nov., 1971.

4.16 Ruben, S., Faiss, R., Lyon, J., and Quinn, M., Application of a Parallel Processing Computer in LACIE, Proc. of the 1976 International Conf. on Parallel Processing, Aug., 1976.

4.17 Batcher, K.E., The Multidimensional Access Memory in STARAN, IEEE Trans. on Comp., Vol. C-26, No. 2, Feb. 1977.

4.18 Stone, H.S., Parallel Processing with the Perfect Shuffle, IEEE Transactions on Computers, Vol. C-20, No. 2, Feb., 1971.

4.19 Gaertner, W.W., Patel, M.P., Reddi, S.S., Retter, C.T., and Singh, I.M., High Resolution Image Processing on Parallel Computer System, Proc. of the 1976 International Conf. on Parallel Processing, Aug., 1976.

4.20 Gaertner, W.W., Architecture for a Highly Reliable Parallel Computer System, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, Aug., 1975.

4.21 Chen, C.J., and Frank, A.A., On Programmable Parallel Data Routing Networks via Crossbar Switches for Multiple Element Computer Architectures, Proc. of the Sagamore Computer Conference, Aug., 1974.

4.22 Lawrie, D.H., Access and Alignment of Data in an Array Processor, IEEE Transactions on Computers, Vol. C-24, No. 12, Dec., 1975.

4.23 Golomb, S.W., Permutation by Cutting and Shuffling, SIAM Rev., Vol. 3, Oct. 1961.

4.24 Pease, M.C., An Adaption of the Fast Fourier Transform for Parallel Processing, Journal of the Association on Computing Machines, Vol. 15, April, 1968.

4.25 Benes, V.E., Mathematical Theory of Connecting Networks and Telephone Traffic, New York: Academic, 1965.

4.26 Batcher, K.E., Sorting Networks and Their Applications, 1968 Spring Joint Computer Conference, AFIPS Conf. Proc., Vol. 32, 1968.

4.27 Batcher, K.E., The Flip Network in STARAN, Proc. of the 1976 International Conf. on Parallel Processing, Aug., 1976.

4.28 Bauer, L.H., Implementation of Data Manipulating Functions on the STARAN Associative Processor, 1974 Sagamore Computer Conference, 1974.

4.29 Opferman, D.C., and Tsao-Wu, N.T., On a Class of Rearrangeable Switching Networks Part I: Control Algorithms, Bell System Technical Journal, Vol. 50, No. 5, May - June, 1971.

4.30 Ramanujam, H.R., Decomposition of Permutation Networks, IEEE Transactions on Computers, Vol. C-22, No. 7, July, 1973.

4.31 Wittie, L.D., Efficient Message Routing in Mega-Micro-Computer Networks, The 3rd Annual Symposium on Computer Architecture, Jan., 1976.

4.32 Reames, C.C., and Liu, M.T., A Loop Network for Simultaneous Transmission of Variable-Length Messages, The 2nd Annual Symposium on Computer Architecture, Jan., 1975.

4.33 Reames, C.C., and Liu, M.T., Design and Simulation of the Distributed Loop Computer Network, The 3rd Annual Symposium on Computer Architecture, Jan. 1976.

4.34 Ramamoorthy, C.V., and Li, H.F., Pipelined Processors - A Survey, Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, Aug., 1975.

5.1 Avižiens, A., Design of Fault-Tolerant Computers, Fall Joint Computer Conference, Vol. 31, AFIPS, 1967.

5.2 Newnann, et al., A Study of Fault-Tolerant Computing: Final Report, SRI Project 1693, Standford Research Institute, Menlo Park, Calif., July, 1973.

5.3 Misunas, D.P., Error Detection and Recovery in a Data Flow Computer, Proceedings of the 1976 International Conference on Parallel Processing, Aug., 1976.

5.4 Szalai, K.J., Felleman, P.G., Gera, J., Glover, R.D., Design and Test Experience With a Triply Redundant Digital Fly-By-Wire Control System, AIAA Guidance and Control Conference, Aug., 1976.

5.5 Roth, J.P., Bouricius, W.G., and Schneider, P.R., Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits, IEEE Transactions on Computers, EC-16, No. 5, Oct., 1967.

5.6 Tasar, V., Analysis of Fault Detection Coverage of a Self-Test Software Program, Bendix Research Laboratories, Technical Report, BRL/TR-77-8524, Sept, 1977.

5.7 McGough, J., Moses, K., Platt, W., Reynolds, G., Strole, J., Digital Flight Control System Redundancy Study, Technical Report AFFDL-TR-74-83, Bendix Flight Systems Division, July, 1974.

5.8 Moore, E. F., Gedanken - Experiments on Sequential Machines, Automata Studies, Annals of Mathematics Studies No. 34, pp. 129-153, Princeton University Press, New Jersey, 1956.

5.9 Miller, R. E., Switching Theory, Vol. II, John Wiley and Sons, New York, 1965.

5.10 Wood, P. E., Switching Theory, McGraw-Hill, 1968.

5.11 Avižienis, A., Gilley, G. C., Mathur, F. P., Rennels, D. A., Rohr, J. A., Rubin, D. A., The STAR Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design, International Symposium on Fault Tolerant Computing, 1971.

5.12 Stiffler, J. J., The SERF Fault-Tolerant Computer Part I: Concept Design, International Symposium on Fault Tolerant Computing, 1973.

5.13 Avižienis, A., A Study of the Effectiveness of Fault-Detecting Codes for Binary Arithmetic, Technical Report No. 32-711, Jet Propulsion Laboratory, Pasadena, California, 1965.

5.14 Avižienis, A., Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design, International Symposium on Fault Tolerant Computing, 1971.

5.15 Mersten, G., Weilbacker, T., Computer Self-Test Program and Computer Validation, Technical Report, Bendix Guidance and Control Division, 1970.

5.16 Brosius, D. B., Jurison, J., Design of a Voter-Comparator-Switch for Redundant Computer Modules, International Symposium on Fault Tolerant Computing, 1973.

5.17 Hsieh, E. P., Checking Experiments for Sequential Machines, IEEE Transactions on Computers, October 1971.

5.18 Spillman, R. J., A Markov Model of Intermittent Faults in Digital Systems, International Symposium on Fault-Tolerant Computing, 1977.

5.19 Meyer, J. F., Fault Tolerant Sequential Machines, IEEE Transactions on Computers, October 1971.

5.20 Tasar, V., Analysis of Fault Detection Coverage of a Self-Test Software Program, Vol. I, Theory, Application, Results and Conclusions, Ph.D Thesis to be submitted to the University of Detroit, Department of Electrical Engineering.

5.21 Miga, R. L., A Method to Determine Percent Structural Error Detectability of Self-Test Software, Bendix Research Laboratories Technical Report, September 1976.

6.1 Mersten, G.S., An Experimental Simultaneous Multiprocessor Organization (SAMSON), Doctoral- Research Proposal, The City College, The City University of New York, March 1974.

6.2 Curtin, W.A., Multiple Computer Systems, Advances in Computers, 1963.

6.3 Jensen, E.D., A Distributed Function Computer for Real-Time Control, Second Annual Symposium on Computer Architecture, 1975.

6.4 Avižienis, A., A Study of the Effectiveness of Fault - Detecting Codes for Binary Arithemetic, Technical Report No. 32-711, Jet Propulsion Laboratory, Pasadena, California, 1965.

6.5 Brosius, J.P. Jr., Russel, B.J., Fault-Tolerant Plated Wire Memory for Long Duration Space Missions, International Symposium on Fault Tolerant Computing, 1973.

6.6 Avižienis, A., Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design, International Symposium on Fault Tolerant Computing, 1971.

6.7 Avižienis, A., Arithmetic Algorithms for Error-Coded Operands, International Symposium on Fault Tolerant Computing, 1972.

6.8 Stiffler, J.J., The SERF Fault-Tolerant Computer Part I: Concept Design, International Symposium on Fault Tolerant Computing, 1973.

6.9 Fletcher, J.C., Shared Memory for a Fault Tolerant Computer, U.S., Patent No. 3,950,729, April 1976.

6.10 Koczela, L.J., A Three Failure Tolerant Computer System, International Symposium on Fault Tolerant Computing, 1971.

6.11 Karosas, J., Žukauskas, K., The Theoretical Analysis of Redundant System Reliability, International Symposium on Fault Tolerant Computing, 1973.

6.12 Brosius, D.B., Jurison, J., Design of a Voter-Comparator-Switch for Redundant Computer Modules, International Symposium on Fault Tolerant Computing, 1973.

6.13 Avižienis, A., Gilley, G.C., Mathur, F.P., Rennels, D.A., Rohr, J.A., Rubin, D.A., The STAR Computer: An Investigation of the Theory and Practice of Fault Tolerant Computer Design, IEEE Transactions on Computers, November, 1971.

6.14 Avižienis, A., Rennels, D.A., RMS: A Reliability Modeling System for Self-Repairing Computers, International Symposium on Fault Tolerant Computing, 1973.

6.15 Carter, W.C., Bouricius, W.G., Jessep, D.C., Roth, J.P., Schneider, P.R., Wadia, A.B., A Theory of Fault Tolerant Computers using Standby Sparing, International Symposium on Fault Tolerance Computing, 1971.

6.16 Abraham, J.A., A Algorithm for the Accurate Reliability Evaluation of TMR Networks, International Symposium on Fault Tolerant Computing, 1973.

6.17 Mathur, F.P., Reliability Modeling, Analysis and Prediction of Ultrareliable Fault Tolerant Digital Systems, International Symposium on Fault Tolerant Computing, 1971.

6.18 Hopkins, A.L. Jr., A Fault Tolerant Information Processing Concept for Space Vehicles, International Symposium on Fault Tolerance, 1971.

6.19 Consolver, G., Ackley, D., Richard, M., McAfee, R., Shipchandler, T., Gylys, V., Distributed Processor/Memory Architectures Design Program, AFAL TR-75-80, Texas Instrument Incorporated, February, 1975.

6.20 Misunas, D.P., Performance Analysis of a Data Flow Processor, Project MAC, MIT, AD-A015 567, August, 1975.

6.21 Rohr, J.A., STAREX Self Repair Routines: Software Recovery in the JPL-STAR Computer. International Symposium on Fault Tolerant Computing, 1973.

6.22 Mersten, G.S., McGough, J.G., Oh, S.J., Achievable Error Detection through Self-Test Software, National Aerospace & Electronics Conference, May, 1978.

6.23 Carter, W.C., Jessep, D.C., Wadia, A.B., Schneider, P.R., Bouricius, W.G., Logic Design for Dynamic and Interactive Recovery. International Symposium on Fault Tolerant Computing, 1971.

7.1 Danielsson, P-E., Gudmundsson, B., Time-Shared Memory-Processor Interface, 1975 Sagamore Computer Conference on Parallel Processing, August, 1975.

APPENDIX A

TABLE A-1

INSTRUCTION REPERTOIRE

BASIC INSTRUCTION SET

MEMORY REFERENCE INSTRUCTIONS

(These can refer to any one of four accumulators)

Mnemonic	Description
ADD	Add Memory to Accumulator
SUB	Subtract Memory from Accumulator
CMP	Compare Memory with Accumulator and Skip 0 if Equal; Skip 1 if Greater; Skip 2 if Less
LOAD	Load Accumulator from Memory
STO	Store Accumulator in Memory
JU	Jump (Unconditional)
JSA0	Jump to Subroutine; Store return address in Accumulator Zero
JSA1	Jump to Subroutine; Store return address in Accumulator One
JSM	Jump to Subroutine; Store return address at start of Subroutine

INTER-REGISTER INSTRUCTION

(These can refer to any two of the 16 accumulators)

Mnemonic	Description
ADDR	Add Inter-Register
IAR	Immediate Add Inter-Register
SUBR	Subtract Inter-Register
CMPR	Compare Inter-Register, and Skip 0 if Equal; Skip 1 if Greater; Skip 2 if Less
MPY	Multiply
DIV	Divide
DECEQ	Decrement and Skip if zero
DECNE	Decrement and Skip if not zero

Table A-1 con't

Mnemonic	Description
TRA	Transfer
IR	Interchange Registers
AND	Logical AND
OR	Logical OR
LCM	Logical Complement
ACM	Arithmetic Complement

SHIFT INSTRUCTIONS
(These can refer to any of the 16 accumulators)

Mnemonic	Description
SLSL	Shift Left Short Logical
SRSL	Shift Right Short Logical
SLSA	Shift Left Short Algebraic
SRSA	Shift Right Short Algebraic
RLS	Rotate Left Short
SLLL	Shift Left Long Logical
	(Long - Double Word Length = 32 bits)
SRLL	Shift Right Long Logical
SLLA	Shift Left Long Algebraic
SRLA	Shift Right Long Algebraic
RLL	Rotate Left Long
SKGT	Skip of Accumulator greater than zero
SKLE	Skip if Accumulator less than or equal to zero
SKGE	Skip if Accumulator greater than or equal to zero
SKLT	Skip of Accumulator less than zero
SKEQ	Skip of Accumulator equal to zero
SKNE	Skip if Accumulator not zero
SSOV	Skip if Overflow Set

Table A-1 con't

Mnemonic	Description
SROV	Skip of Overflow Not Set
SSIE	Skip if Interrupt Enable Set
SRIE	Skip if Interrupt Enable Not Set
SSF1	Skip if Flag 1 Set
SRF1	Skip if Flag 1 Not Set
SSF2	Skip if Flag 2 Set
SRF2	Skip if Flag 2 Not Set
STIR	Skip if Interrupt Request True
SFIR	Skip if Interrupt Request False
STE1	Skip if External 1 True
SFE1	Skip if External 1 False
STE2	Skip if External 2 True
SFE2	Skip if External 2 False
STE3	Skip if External 3 True
SFE3	Skip if External 3 False

CONTROL INSTRUCTIONS

Mnemonic	Description
CONT	Modify Status of Flag 1, Flag 2, Overflow, and Interrupt Enable
CLAO	Clear Overflow and Specified Accumulator
CLA	Clear Specified Accumulator
NOP	No Operation
HALT	Halt

Table A-1 con't

INPUT/OUTPUT INSTRUCTIONS

(These can refer to any of the 16 accumulators)

Mnemonic	Description
OD	Output Data from Accumulator
OSR	Output Data from Accumulator and Skip if Ready
ID	Input Data to Accumulator
ISR	Input Data to Accumulator and Skip if Ready
OC	Output Control
ISW	Input Switch Register to Accumulator

EXPANDED INSTRUCTION

(These instructions are only available in the SAMSON-A PE)

DOUBLE PRECISION INSTRUCTIONS

(These can refer to any two adjacent pair of the 16 accumulators)

Mnemonic	Description
DACM	Double Precision Arithmetic Complement
DADDR	Double Precision Add
DSUBR	Double Precision Subtract
DMPY	Double Precision Multiply

LOGICAL INSTRUCTION

(This can refer to any two of the 16 accumulators)

Mnemonic	Description
EXOR	Exclusive OR

Table A-1 con't

MULTIPLE REGISTER MEMORY REFERENCE INSTRUCTIONS

(These can refer to any number of the consecutive (16) accumulators)

Mnemonic	Description
LDM	Load Multiple Registers from Memory
STM	Store Multiple Registers in Memory

STACK INSTRUCTIONS

Mnemonic	Description
JSS	Jump to Subroutine and Push Stack
RPS	Return from Subroutine and Pop Stack

SATURATE INSTRUCTIONS

Arithmetic instructions can be operated in the saturated mode, forcing results to be limited, in the case of an overflow, to the most positive or negative number representable.

Table A-2

INSTRUCTION EXECUTION TIMES: (TIME IN μSECS)

BASIC INSTRUCTIONS

MEMORY REFERENCE INSTRUCTIONS

INSTRUCTION TYPE	ADDRESSING MODE		
	BASE	REL P/A0/A1	INDIRECT
ADD	0.6+2T	0.8+2T	+0.4N+TN
SUB	0.6+2T	0.8+2T	+0.4N+TN
CMP A=M	1.0+2T	1.2+2T	+0.4N+TN
A>M	1.2+3T	1.4+3T	+0.4N+TN
A<M	1.4+4T	1.6+4T	+0.4N+TN
LOAD	0.6+2T	0.8+2T	+0.4N+TN
STO	0.6+2T	0.8+2T	+0.4N+TN
JU	0.6+3T	0.8+3T	+0.4N+TN
JSA0	0.8+2T	1.0+2T	+0.4N+TN
JSA1	0.8+2T	1.0+2T	+0.4N+TN
JSM	1.0+2T	1.2+2T	+0.4N+TN

NOTE: N IS THE NUMBER OF INDIRECT LEVELS.

INTERREGISTER INSTRUCTIONS

Instruction	Time
ADDR	0.2+T
IAR	0.8+T
SUBR	0.2+T
CLA0	0.2+T
CMPR A=D	0.6+2T
A>B	0.8+3T
A<B	1.0+4T
MPY	1.4+T
DIV	7.75+T AVER (7.6+T → 8.0+T)
TRA	0.2+T

INTERREGISTER INSTRUCTIONS (Cont'd)

Instruction	Time
IR	0.8+T
CLA	0.8+T
AND	0.2+T
OR	0.2+T
LCM	0.2+T
ACM	0.2+T

T = (Memory Cycle Time) - 130 nsec if the memory cycle time exceeds 200 nsec; otherwise, T=0

Table A-2 con't

SKIP INSTRUCTIONS

DECEQ	0.8+T	
DECNE	0.8+T	
SKGT	0.8+T:	LESS THAN
	1.0+T:	GREATER OR EQUAL
SKLE	0.8+T:	LESS THAN
	1.0+T:	GREATER OR EQUAL
SKGE	0.8+T	
SKLT	0.8+T	
SKEQ	0.8+T	
SKNE	0.8+T	
SKIP ON DIRECTION	0.8+T	

CONTROL INSTRUCTIONS

CONT	0.4+T
NOP	0.2+T

SHIFT INSTRUCTIONS

SLSL	0.4+0.2S+T
SRSL	0.4+0.2S+T
SLSA	0.4+0.2S+T
SRSA	0.4+0.2S+T
RLS	0.4+0.2S+T
SLLL	0.4+0.2S+T
SRLL	0.4+0.2S+T
SLLA	0.4+0.2S+T
SRLA	0.4+0.2S+T
RLL	0.4+0.2S+T

S IS THE NUMBER OF SHIFTS

INPUT/OUTPUT INSTRUCTIONS

ISW	0.4+T	
OD	1.0+T	
OSR	1.0+T:	NOT READY
	1.4+T:	READY
ID	1.0+T	
ISR	1.0+T:	NOT READY
	1.4+T:	READY
OC	0.8+T	

Table A-2 con't

LOGICAL

INSTRUCTION TYPE	
EXOR	0.2+T

DOUBLE PRECISION

INSTRUCTION TYPE	
DACM	0.6+2T
DADDR	0.4+T
DSUBR	0.4+T
DMPY	13.6+2T→20.2+2T

MULTIPLE REGISTER MEMORY REFERENCE INSTRUCTIONS

INSTRUCTION TYPE	BASE	REL P/A0/A1	INDIRECT
LDM	1.2+4T+0.2A+TA	1.2+4T+0.2A+TA	+0.4N+NT
STM	1.2+4T+0.2A+TA	1.2+4T+0.2A+TA	+0.4N+NT

STACK INSTRUCTIONS

INSTRUCTION TYPE	BASE	REL P/A0/A1	INDIRECT
JSS	1.6+3T	1.8+3T	+0.4N+NT
RPS	1.4+4T	-	-

NOTE: N=NUMBER OF INDIRECT LEVELS
A=NUMBER OF ACCUMULATORS

Table A-2 con't

EXPANDED INSTRUCTION EXECUTION TIMES

SATURATE INSTRUCTIONS

MEMORY REFERENCE

INSTRUCTION TYPE	BASE	NO OVFL REL P/A0/A1	INDIRECT
ADD	0.8+2T	1.0+2T	+0.4N+TN
SUB	0.8+2T	1.0+2T	+0.4N+TN

INSTRUCTION TYPE	BASE	OVFL REL P/A0/A1	INDIRECT
ADD	1.4+4T	1.6+4T	+0.4N+TN
SUB	1.4+4T	1.6+4T	+0.4N+TN

INTERREGISTER

INSTRUCTION TYPE	NO OVFL	OVFL
ADDR	0.8+3T	0.8+3T
SUBR	0.8+3T	0.8+3T
IAR	0.8+T	0.8+T
CLAO	0.8+3T	N/A
DADDR	0.6+T	1.4+3T
DSUBR	0.6+T	1.4+3T
SLSA	0.4+0.2S+T	0.8+0.2S'+T 1.0+0.2S+2T OVFL ON LAST SHIFT
SLLA	0.4+0.2S+T	1.0+0.2S'+3T

NOTE: N=NUMBER OF INDIRECT LEVELS
S=NUMBER OF SHIFTS
S'=NUMBER OF SHIFTS WHICH CAUSES OVFL TO OCCUR

Appendix B

Table B-1

Normalized Execution Time for the Strapdown Guidance Program

<u>Task</u>

STG1:

Number of PEs	Normalized Exec. Time
1	1
2	.65 → .68
3	.58 → .60
4	.58

STG2:

Number of PEs	Normalized Exec. Time
1	1
2	.64
3	.52
4	.49
5	.49
6	.41
7	.38

Table B-1
Normalized Execution Time
for the Strapdown Guidance Program

<u>Task</u>

STG3:

Number of PEs	Normalized Exec. Time
1	1
2	.66
3	.57

STG4

Number of PEs	Normalized Exec. Time
1	1
2	.81
3	.63

Table B-1
Normalized Execution Time
for the Strapdown Guidance Program

Task

STG5:

Number of PEs	Normalized Exec. Time
1	1
2	.50
3	.33
4	.33
5	.33
6	.17

STG5 Alternate:

Number of PEs	Normalized Exec. Time
1	1
2	.50
3	.33
4	.33
5	.38
6	.17

STG6

Number of PEs	Normalized Exec. Time
1	1
2	.65
3	.60

Table B-2
Normalized Execution Time
for the Omega Navigation Program

Task

ONS1:

Number of PEs	Normalized Exec. Time
1	1
2	.67
3	.44
4	.42
5	.41

Table B-3

Normalized Execution Time

for the Diagnostic Program

Task

DGN1:

Number of PEs	Normalized Exec. Time
1	1
2	.64

DGN2:

Number of PEs	Normalized Exec. Time
1	1
2	.72
3	.69
4	.68
5	.68
6	.66

DGN3:

Number of PEs	Normalized Exec. Time
1	1
2	.65

DGN4:

Number of PEs	Normalized Exec. Time
1	1
2	.81
3	.65

Table B-3

Normalized Execution Time for the Diagnostic Program

Task

DGN5:

Number of PEs	Normalized Exec. Time
1	1
2	.86

DGN6:

Number of PEs	Normalized Exec. Time
1	1
2	.81

DGN7:

Number of PEs	Normalized Exec. Time
1	1
2	.75
3	.68

DGN8:

Number of PEs	Normalized Exec. Time
1	1
2	.69
3	.67

Table B-3
Normalized Execution Time
for the Diagnostic Program

Task

DGN9:

Number of PEs	Normalized Exec. Time
1	1
2	.59
3	.56
4	.41

DGN10:

Number of PEs	Normalized Exec. Time
1	1
2	.71
3	.50

DGN11:

Number of PEs	Normalized Exec. Time
1	1
2	.64

DGN12:

Number of PEs	Normalized Exec. Time
1	1
2	.62
3	.55

Table B-3
Normalized Execution Time
for the Diagnostic Program

Task

DGN13:

Number of PEs	Normalized Exec. Time
1	1
2	.64
3	.61

DGN14:

Number of PEs	Normalized Exec. Time
1	1
2	.71
3	.64
4	.64
5	.64
6	.61

DGN15:

Number of PEs	Normalized Exec. Time
1	1
2	.66
3	.59
4	.49
5	.49
6	.49
7	.49
8	.46

Table B-3
Normalized Execution Time
for the Diagnostic Program

Task

DGN16:

Number of PEs	Normalized Exec. Time
1	1
2	.59
3	.52
4	.48
5	.44

DGN17:

Number of PEs	Normalized Exec. Time
1	1
2	.91

DGN18:

Number of PEs	Normalized Exec. Time
1	1
2	.67
3	.61

Table B-4
Hardware Utilization
for the Strapdown Guidance Program

Task

STG1:

Number of PEs	% Individual PE Utilization 1	2	3	4	% Total Utilization
1	100	-	-	-	100%
2	100	48	-	-	74
3	100	44→57	16→17	-	54→58
4	100	48→57	17	0→9	43

STG2:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	6	7	% Total Utilization
1	100	-	-	-	-	-	-	100
2	100	57	-	-	-	-	-	78
3	100	47	44	-	-	-	-	64
4	100	50	29	24	-	-	-	51
5	100	32	24	24	24	-	-	41
6	100	32	29	29	29	29	-	41
7	100	35	31	31	31	31	8	38

Table B-4

Hardware Utilization

for the Strapdown Guidance Program

Task

STG3:

Number of PEs	% Individual PE Utilization			% Total Utilization
	1	2	3	
1	100	-	-	100
2	100	51	-	76
3	100	59	16	59

STG4:

Number of PEs	% Individual PE Utilization			% Total Utilization
	1	2	3	
1	100	-	-	100
2	100	23	-	62
3	100	30	30	53

STG5:

Number of PEs	% Indiviudal PE Utilization						% Total Utilization
	1	2	3	4	5	6	
1	100	-	-	-	-	-	100
2	100	100	-	-	-	-	100
3	100	100	100	-	-	-	100
4	100	100	50	50	-	-	75
5	100	50	50	50	50	-	60
6	100	100	100	100	100	100	100

Table B-4
Hardware Utilization
for the Strapdown Guidance Program

Task

STG5

Alternate:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	6	% Total Utilization
1	100	-	-	-	-	-	100
2	100	100	-	-	-	-	100
3	100	100	100	-	-	-	100
4	100	100	100	100	-	-	100
5	100	88	88	88	88	-	90
6	100	100	100	100	100	100	100

STG6:

Number of PEs	% Individual PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	54	-	77
3	100	58	8	56

Table B-5
Hardware Utilization
for the Omega Navigation Program

Task

ONS1:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	% Total Utilization
1	100	-	-	-	-	100
2	100	50	-	-	-	75
3	100	74	56	-	-	77
4	100	72	58	7	-	59
5	100	75	60	7	4	49

Table B-6
Hardware Utilization
for the Diagnostic Program

Task

DGN1:

Number of PEs	% Individual PE Utilization 1	2	% Total Utilization
1	100	-	100
2	100	57	79

DGN2:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	6	% Total Utilization
1	100	-	-	-	-	-	100
2	100	39	-	-	-	-	70
3	100	37	8	-	-	-	48
4	100	38	6	4	-	-	37
5	100	35	6	4	2	-	29
6	100	36	6	4	2	2	25

DGN3:

Number of PEs	% Individual PE Utilization 1	2	% Total Utilization
1	100	-	100
2	100	53	77

DGN4:

Number of PEs	% Indiviudal PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	24	-	62
3	100	29	24	51

Table B-6
Hardware Utilization
for the Diagnostic Program

Task

DGN5:

Number of PEs	% Individual PE Utilization 1	2	% Total Utilization
1	100	-	100
2	100	17	59

DGN6:

Number of PEs	% Individual PE Utilization 1	2	% Total Utilization
1	100	-	100
2	100	23	62

DGN7:

Number of PEs	% Indiviudal PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	33	-	67
3	100	37	11	49

DGN8:

Number of PEs	% Individual PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	45	-	73
3	100	47	3	50

Table B-6
Hardware Utilization
for the Diagnostic Program

Task

DGN9:

Number of PEs	% Individual PE Utilization 1	2	3	4	% Total Utilization
1	100	-	-	-	100
2	100	70	-	-	85
3	100	47	32	-	60
4	100	64	43	36	61

DGN10:

Number of PEs	% Individual PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	40	-	70
3	100	57	43	67

DGN11:

Number of PEs	% Individual PE Utilization 1	2	% Total Utilization
1	100	-	100
2	100	57	79

DGN12:

Number of PEs	% Individual PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	62	-	81
3	100	70	13	61

Table B-6
Hardware Utilization
for the Diagnostic Program

Task

DGN13:

Number of PEs	% Individual PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	56	-	78
3	100	59	6	55

DGN14:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	6	% Total Utilization
1	100	-	-	-	-	-	100
2	100	40	-	-	-	-	70
3	100	39	17	-	-	-	52
4	100	39	11	6	-	-	39
5	100	33	11	6	6	-	31
6	100	35	12	6	6	6	27

DGN15:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	6	7	8	% Total Utilization
1	100	-	-	-	-	-	-	-	100
2	100	52	-	-	-	-	-	-	76
3	100	42	29	-	-	-	-	-	57
4	100	45	35	25	-	-	-	-	51
5	100	45	35	20	5	-	-	-	41
6	100	45	30	20	5	5	-	-	34
7	100	40	30	20	5	5	5	-	29
8	100	42	32	21	5	5	5	5	27

Table B-6

Hardware Utilization

for the Diagnostic Program

Task

DGN16:

Number of PEs	% Individual PE Utilization 1	2	3	4	5	% Total Utilization
1	100	-	-	-	-	100
2	100	64	-	-	-	85
3	100	71	21	-	-	64
4	100	69	23	15	-	52
5	100	75	25	17	8	45

DGN17:

Number of PEs	% Individual PE Utilization 1	2	% Total Utilization
1	100	-	100
2	100	10	55

DGN18:

Number of PEs	% Individual PE Utilization 1	2	3	% Total Utilization
1	100	-	-	100
2	100	48	-	74
3	100	53	10	55

Table B-7

% Accumulator Usage

Program	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	Total References
STG	45.0	18.5	4.6	17.8	.6	.2	.2	.2	.5	.5	.9	—	.2	.5	2.6	1.6	1249
DGN	24.0	24.0	16.3	11.7	7.8	2.9	3.0	2.2	.9	.9	1.6	1.1	.6	1.3	.6	1.1	938
ONS	17.9	3.6	33.2	28.7	3.7	3.3	3.7	3.3	1.7	.1	.8	—	—	—	—	—	19627

DGN SIMULATION

```
A0 =  131005     A1 =     146     A2 =  161665     A3 =  143520
A4 =       0     A5 =  177433     A6 =  177456     A7 =  177501
A8 =  177524     A9 =  177547     A10=  177341     A11=  1767C2
A12=  177341     A13=   53775     A14=  177341     A15=   45413
OVERFLOW = SET      INT. ENABLE = REST      FLAG1 = REST      FLAG2 = REST
TIME=     7C9.5
```

LINE	INST AC	OBJ WD	SOURCE CARD		EFF AD	(EF AD)	(A0)	(A1)	(A2)	(A3)	(REG)	(REG)	OV
442	1356	100442	SLLA	2, 2					107327	16500			NO
443	1357	105713	SSCV	1353	1360	40C2							
444	1360	4002	ADDR	0, 2			40334		1C7327				OV
445	1361	100440	SLLA	2, 0					107327	16500			NO
446	1362	105710	SSOV	1353	1363	4002							
447	1363	4CC2	ACDR	0, 2			147663		107327				NC
448	1364	3441	RLL	2, 1					16656	35201			
449	1365	4C02	ADDR	0, 2			166541		16656				NO
450	1366	3443	RLL	2, 3					166561	152010			
451	1367	4C02	ADDR	C, 2			155322		166561				NC
452	1370	3440	RLL	2, 0			155322		166561				
453	1371	4C02	ADDR	0, 2			1441C3		166561				NO
454	1372	10244C	SLLL	2, 0					166561	152010			
455	1373	4002	ADDR	0, 2			132664		166561				NO
456	1374	106440	SRLL	2, C					166561	152010			
457	1375	4002	ADDR	0, 2			121445		166561				NO
458	1376	104440	SRLA	2, 0					166561	152010			
459	1377	4C02	ACDR	0, 2			110226		166561				NC
460	1400	104441	SRLA	2, 1					173270	1C5004			
461	1401	4002	ADDR	C, 2			103516		173270				NO
462	1402	104443	SRLA	2, 3					177327	16500			
463	1403	4002	ACDR	0, 2			103045		177327				NO
464	1404	106441	SRLL	2, 1					77553	107240			
465	1405	4002	ADDR	C, 2			2620		77553				NO
466	1406	106442	SRLL	2, 2					17732	161650			
467	1407	31104	SUB	2, 1C4	104	17732			0				NC
468	1410	31505	SUB	3, 1C5	1C5	16165C				0			NO
469	1411	30106	SUB	0, 10C	1C6	262C	0						NC
470	1412	107042	SKNE	2, 1415	1413	1C7061			0				
471	1413	107C61	SKNE	3, 1415	1414	1C5001				0			
472	1414	105001	SKEC	0, 1416	1416	11142	0						
473	1416	11142	JSA1	142	142	1		1417					
474	142	1	TRA	0, 1			1417						
475	143	22243	ADD	0, ,REL	6	13332	14751						NO
476	144	62242	STO	0, ,REL	6	14751	14751						
477	145	17000	JSA1	0,1	1417	5C1C7		146					
478	1417	50107	LCAD	C, 107	1C7	1CC0CC	10000C						
479	1420	40	TRA	2, C					100000				
480	1421	1100	LCM	4, 0							4= 77777		
481	1422	124	TRA	5, 4							5= 77777		
482	1423	1C2121	SLSL	5, 1							5=177776		
483	1424	5402	MPY	0, 2			77777	177776	100000				
484	1425	60C4	SUBR	C, 4			C				4= 77777		NO
485	1426	6025	SUBR	1, 5				0			5=177776		NO
486	1427	107001	SKNE	0, 1431	143C	105021	0						
487	1430	105021	SKEQ	1, 1432	1432	5C04C		0					
488	1432	50C4C	LCAC	0, 40	4C	127C73	127073						
489	1433	5440	MPY	2, 0			127C73		50705	0			
490	1434	4040	ADDR	2, 0			127073		0				NC

```
830  2317   27317     0         STO   2,-1,1
831  2320   55001     0         LCAD  2,1,0       TRASFER STATUS OF STATIONS TO MACHINE REG 9   00857000
832  2321     222     C         TRA   9,2                                                        00858000
833  2322   14002     C         JU    2,0         RETURN                                         00859000
834  2323    2324  2324  LFFTR  LINK  **1                                                        00860000
835  2324    2137  2137         LINK  LF102       ADRESS OF 10.2 KC/S DATA (LEFT SIDE)           00861000
836  2325    3107  3107         LINK  LF136       ADRESS OF 13.6 KC/S DATA (LEFT SIDE)           00862000
837  2326    2327  2327  RTFTR  LINK  **1                                                        00864000
838  2327    3257  3257         LINK  RT102       ADRESS OF 10.2 KC/S DATA (RIGHT SIDE)          00865000
839  2330    3427  3427         LINK  RT136       ADRESS OF 13.6 KC/S DATA (RIGHT SIDE)          00866000
                         *                                                                        00868000
                         *****          UNPACK & POSITION DATA                                    00869000
                         *                                                                        00870000
840  2331       0     0  LAFAK  RES   1                                                          00871000
841  2332  104450     C         SPLA  2,8                                                        00872000
842  2333  102044     0         SLSL  2,4               RESCALE                                  00873000
843  2334  104070     0         SPSA  3,8                                                        00874000
844  2335  102064     C         SLSL  3,4               RESCALE                                  00875000
845  2336  112331  2331         JUA   UNPAR                                                      00876000
846             0     C  S2014  EQU   *                                                          00877000
                         *                                                                        00878000
                         *                                                                        00879000
                         *                                                                        00880000
                         ******              CROSS-CORRELATOR                                     00881000
                         *                                                                        00882000
                         *                                                                        00883000
                         *                                                                        00884000
847             0     C  S2015  EQU   *                                                          00885000
848  2337       0     0  CFCCR  RES   1                                                          00886000
849  2340   52147  2507         LOAD  0,STSIN     SET STARTING ADRESS (DATA)                     00887000
850  2341  150461  2555         LOAD  1,ACCFR     STORAGE AREA FOR CROSS-CORRELATOR OUTPUT       00888000 *****
851  2342   53563  2525         LCAC  3,FSYAC     CHECK IF SYNC ACHIEVED                         00889000
852  2343  105064     C         SKEC  3,**5       SKIP IF 'SYNC' NOT ACHIEVED                    00890000
853  2344   22162  2526         ADD   0,SLCT      MODIFY STARTING DATA ADRESS                    00891000
854  2345  107413     0         LAR   0,-5                                                       00892000
855  2346   22560  2526         ADD   1,SLCT      MODIFY STORAGE ADRESS                          00893000
856  2347  107433     0         LAR   1,-5                                                       00894000
857  2350   62162  2532         STO   0,STACR                                                    00895000
858  2351  160470  2556         STO   1,ACCRR+1                                                  00896000 *****
859  2352   52156  2530         LCAC  0,HIGHC     SET CROSS-CORRELATOR COUNTER                   00897000
860  2353   62156  2531         STO   0,HIGHC+1                                                  00898000
861             0     0  LAPL2  EQU   *                                                          00899000
862  2354   52145  2521         LCAD  0,APTRN     SET STARTING ADRESS (REPLICA)                  00900000
863  2355   62145  2522         STO   0,APTRN+1                                                  00901000
864  2356   52145  2523         LCAD  0,APTRN     INTERMEDIATE LEVEL COUNTER (=8)                00902000
865  2357   62145  2524         STO   0,APTRN+1                                                  00903000
866  2360   52173  2553         LCAD  0,ASCAL     POINTER TO AVERAGING FACTORS                   00904000
867  2361   62173  2554         STO   0,ASCAL+1                                                  00905000
868  2362   52150  2532         LCAC  0,STACR                                                    00906000
869  2363     251     C         TRA   10,9        SET STATION STATUS IN A10                      00907000
870             0     C  LAP10  EQU   *                                                          00908000
```

```
START AT OCTAL P=400*

PROGRAM  E.M.A.  1ST   LAST     LAST     2ND   LAST     LAST
COUNTER          ACC   READ     WRITE    ACC   READ     WRITE
                 NO.   DIST.    DIST.    NO.   DIST.    DIST.
                       1ST      1ST            2ND      2ND
                       ACC.     ACC.           ACC.     ACC.
   256           15       0        0     15       0        0
   257  R   261
   261
I/O INSTRUCTION, ADDR.=  15
 WHAT IS DATA (OCTAL REPR.)? TERMINATE WITH /(SLASH)
0/
                  0       2        2
   262            0       0        0
   263            0       0        0
   266            0       0        1     0       0        1
   267  W   132   0       0        0
   268  W   223   0       0        1
   269  R   222   0       0        2
   270            0       0        3
   271  W   253   0       0        0
   272  R   323   1      11       11
   273  W   242   1       0        0
   274  W   243   1       0        1
   275  R   336   1       0        2
   276  W   241   1       0        0
   277            0       5        6
   278  W   333   0       0        0
   279
INDIRECT: R       2
OPERAND AT:
        R   423
   424            2      19       19     2      19       19
   425            3      20       20     3      20       20
   426
INDIRECT: R       4
OPERAND AT:
        R   573   0       3        4
   427            0       0        0
   428           15      22       22     0       0        0
   429
INDIRECT: R       5
OPERAND AT:
        R   572   0       0        1
   430  R   461   1       9       10     0       0        0
   431           14      26       26     1       0        0
   432  R   450   1       0        1     0       1        2
   433  R   257   3       7        7     1       0        0
   434
   436           14       3        3
   437            1       2        3
   438  R   433
   433  R   253   3       4        4     1       1        1
   434
   436           14       4        4
   437            1       2        4
   433  R   433
```