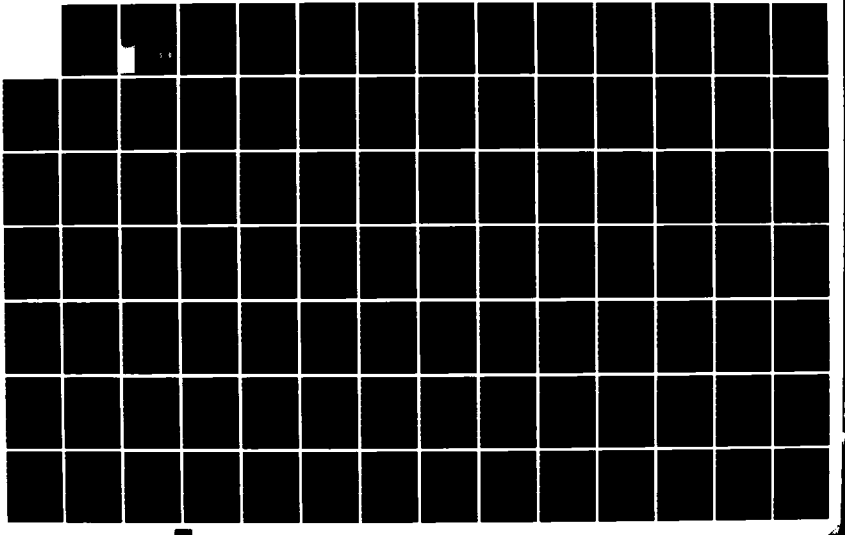
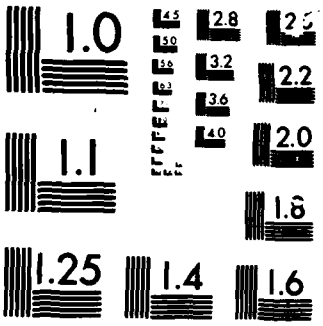
AD-A167 316 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR.. (U) PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING L J SEIGEL ET AL. 1/4

UNCLASSIFIED MAR 83 TR-EE-83-11 ARO-18790.17-EL-APP-A F/G 9/2 NL





MICROCOPY CHART

APPENDIX A for
Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation; Final Report for U.S. Army Research Office
Contract No. DAAG29-82-K-0101

Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation

1983 Progress Report

L.J. Siegel, H.J. Siegel, P.H. Swain,
G.B. Adams III, W.E. Kuhn III,
R.J. McMillen, T.A. Rice,
K.D. Smith, D.L. Tuomenoksa

TR-EE 83-11
March 1983

U.S. Army Research Office
Contract No. DAAG29-82-K-0101

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation 1983 Progress Report	5. TYPE OF REPORT & PERIOD COVERED: Progress Report: April 1, 1982 through March 31, 1983
	6. PERFORMING ORG. REPORT NUMBER: TR-EE 83-11
7. AUTHOR(s): L. J. Siegel, H. J. Siegel, P. H. Swain, G. B. Adams III, W. E. Kuhn III, R. J. McMillen, T. A. Rice, K. D. Smith, D. L. Tuomenoksa	8. CONTRACT OR GRANT NUMBER(s): Contract No. DAAG29-82-K-0101
9. PERFORMING ORGANIZATION NAME AND ADDRESS: School of Electrical Engineering, Purdue University, West Lafayette, IN 47907	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709	12. REPORT DATE: March 1983
	13. NUMBER OF PAGES: 292
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed computing, parallel processing, asynchronous computation, signal processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Research in the area of distributed computing systems for digital signal processing applications is described. The work involves the modeling of asynchronous parallel processes and computer systems for executing these processes. The objective of the work is to develop techniques by which the compatibility of an architecture and an algorithm can be evaluated. The three part effort addresses:

1. Modeling of asynchronous parallel computer system architectures;
2. Modeling of asynchronous parallel computational processes;
3. Evaluation of alternative architectures relative to classes of computational

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

processes.

The approach to the modeling of parallel processes and architectures is to examine the parallelism in a variety of one- and two-dimensional signal processing tasks. This includes a study of the ways in which different types of digital signal processing tasks can be executed on different types of architectures. The goal is to develop one set of features by which processes can be characterized, and another set of features by which parallel architectures can be characterized; and to use these features to obtain measures for the evaluation of process/architecture compatibility.

This research will contribute to the understanding both of how distributed computer systems can be designed for the execution of a class of tasks, and of how signal processing tasks can be decomposed for execution on a distributed computing system.

DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PARALLEL COMPUTATION

1983 PROGRESS REPORT

L. J. Siegel, H. J. Siegel, P. H. Swain,
G. B. Adams III, W. E. Kuhn III, R. J. McMillen, T. A. Rice,
K. D. Smith, D. L. Tuomenoksa

March 1983

U.S. Army Research Office
Contract No. DAAG29-82-K-0101

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

TR-EE 83-11

THE VIEWS, OPINIONS AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION. TR-EE 83-11

DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PARALLEL COMPUTATION 1983 PROGRESS REPORT

CONTENTS

Abstract

CHAPTER 1 - INTRODUCTION 1-1

CHAPTER 2 - MODELING ARCHITECTURES: CLASSIFICATION SCHEMES 2-1

CHAPTER 3 - MODELING ARCHITECTURES: MULTISTAGE INTERCONNECTION NETWORKS 3-1

CHAPTER 4 - HIGH LEVEL DESCRIPTIONS OF CONCURRENCY IN PROCESSES 4-1

CHAPTER 5 - FEATURES FOR DESCRIBING PROCESSES AND ARCHITECTURES 5-1

CHAPTER 6 - APPLICATION STUDIES 6-1

REFERENCES R-1

DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PARALLEL COMPUTATION

1983 PROGRESS REPORT

L. J. Siegel, H. J. Siegel, P. H. Swain
G. B. Adams III, W. E. Kuhn III, R. J. McMillen, T. A. Rice,
K. D. Smith, D. L. Tuomenoksa

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907

March 1983

TR-EE 83-11

ABSTRACT

Research in the area of distributed computing systems for digital signal processing applications is described. The work involves the modeling of asynchronous parallel processes and computer systems for executing these processes. The objective of the work is to develop techniques by which the compatibility of an architecture and an algorithm can be evaluated. The three part effort addresses:

1. Modeling of asynchronous parallel computer system architectures;
2. Modeling of asynchronous parallel computational processes;
3. Evaluation of alternative architectures relative to classes of computational processes.

The approach to the modeling of parallel processes and architectures is to examine the parallelism in a variety of one- and two-dimensional signal processing tasks. This includes a study of the ways in which different types of digital signal processing tasks can be executed on different types of architectures. The goal is to develop one set of features by which processes can be characterized, and another set of features by which parallel architectures can be characterized; and to use these features to obtain measures for the evaluation of process/architecture compatibility.

This research will contribute to the understanding both of how distributed computer systems can be designed for the execution of a class of tasks, and of how signal processing tasks can be decomposed for execution on a distributed computing system.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Evolving digital technology has made it possible to upgrade substantially the computational power available for solving complex data processing problems. Improvements in electronic device speed and central processing unit (CPU) architecture have brought about significant enhancements in system throughput. Largely untapped, however, is the potential for further gains offered by distributed processing. This potential has been demonstrated by the parallel processing systems which have appeared, such as Illiac IV [BaB68,Bou72], STARAN [Bat74,Dav74], and PEPE [Bla77,Wel77], but these represent only a very restricted class of multiprocessor systems. The "microprocessor revolution" has made feasible more general distributed system architectures which have recently begun to come under study [e.g.,Lie79,Nut77,Pea77,SiM78,SiS81a,SuB77, SwF77].

"A distributed processing system is one in which the computing functions are dispersed among several physical computing elements" [Lie79]. Given this definition, distributed processing can be further subdivided using terminology and definitions commonly accepted but nowhere formally standardized. The first subdivision is into "loosely coupled" and "tightly coupled." In tightly coupled systems, there is generally a higher degree of interaction and sharing among the processors to accomplish some specific task. In contrast to these tightly coupled systems, the nationwide ARPANET can be considered a loosely coupled system. The category of tightly coupled distributed processing can be further subdivided into synchronous processing and asynchronous processing. The synchronous subdivision includes SIMD systems, such as the Illiac IV [Bou72]. The asynchronous subdivision includes MIMD systems, such as the C.mmp

[WuB72]. The use of asynchronous distributed systems is the focus of the research described in this report.

One application area which stands to benefit greatly from the development of distributed processing is digital signal processing. One- and two-dimensional signal processing methods typically involve large amounts of computation, often required in real-time, and are generally of a character amenable to parallel processing.

To illustrate, consider the hypothetical missile detection and tracking system shown in Figure 1.1. In this highly simplified example, the system is composed of a large number of signal processing operations applied to both one- and two-dimensional inputs. Pictured are inputs from a variety of sources including seismic sensors used to detect missile launches and temporal sequences of satellite imagery which might be used to track suspected missiles. Both sets of signals are assumed noisy and are subjected to filtering before attempts are made to extract characteristic features and apply pattern recognition techniques to classify objects detected by the system. The figure suggests that once a suspected missile is detected, it is continuously tracked and the tracking information is analyzed to reinforce (or to quench) the identification of the object as a missile. Of course, it is entirely possible that many such objects will have to be tracked simultaneously. The ultimate output of the system might be a listing and/or a real-time display of suspected missiles detected, their locations, trajectories, and possible targets, or the system output might act as input to the control of a radar and/or defense system.

The performance of such a system, which must operate very reliably and in real-time, can be greatly enhanced through use of distributed processing. Many of the component operations can be implemented independently, some using special purpose processors, most themselves employing parallel or pipeline operations internally (e.g., MAP [Csp00], ASAP [Esl00], AP-120B [Flo00], FDP [GoL71], SPS-4 [Sig00]). System reliability can be enhanced through deliberate incorporation of redundancy in the form of

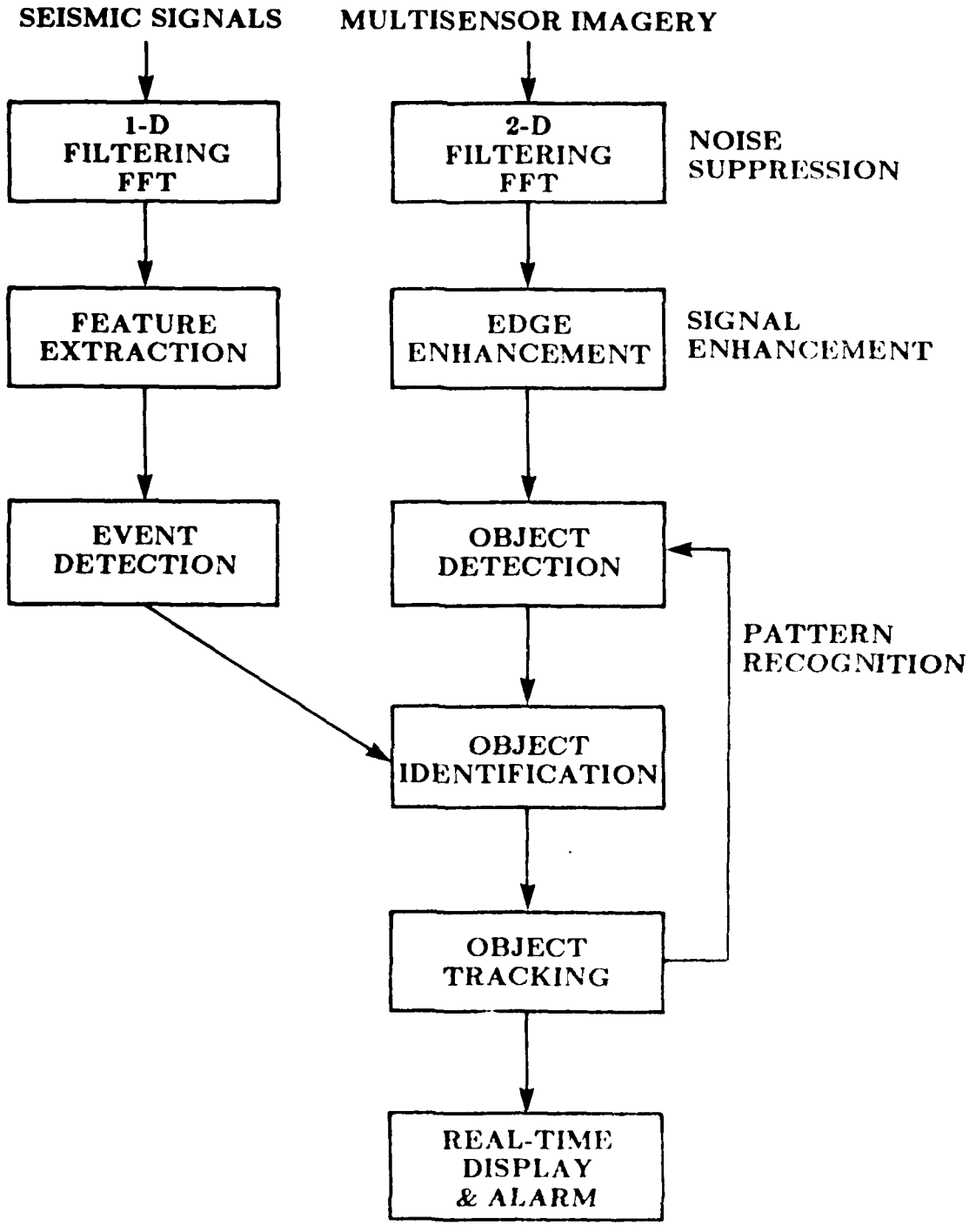


Figure 1.1 Hypothetical Missile Detection and Tracking System

additional parallel processing.

Effective implementation of complex high-performance signal processing tasks using distributed processing involves numerous difficult design choices. Each of the component signal processing operations could be carried out by a variety of alternative algorithms requiring different amounts of computation and producing results of differing quality. Each of the candidate algorithms could be implemented using any of a number of different uni- or multiprocessor architectures. Whatever the choices, the component subsystems will have to communicate with each other and be coordinated so that data and intermediate results continue to stream through the system without encountering bottlenecks.

Some rather specialized architectures have been developed for specific problems or narrow classes of computational processes [e.g.,Bat79,DuW74,Kru73,Mcc73]. Somewhat more general studies have been made of the capabilities of particular parallel architectures [e.g.,Bau74,Orc76,RoP77,RuF76]. The use of parallelism based on consideration of the task alone, independent of a particular architecture, has been examined in the Hearsay II speech understanding project [FeL77,Les75]. Some initial work has been done in the study of the relationships between algorithms and architectures [e.g.,Fos76,Gon78,KuP79]. However, much more work is needed in the area of deriving a fundamental understanding of the ways in which tasks can be structured as parallel processes and the ways in which to match parallel processes to parallel architectures. The analysis of the relationships between the problem domain of digital signal processing and the solution domain of special purpose distributed computing is the technological gap on which this research focuses.

1.2 Research Objectives

The general objective of this research is to identify the parameters which best characterize distributed processing architectures and asynchronous parallel processes as they apply to tasks from the signal processing problem domain so that the most suitable architecture for any given task can be determined. More specifically, a three part effort is being pursued:

1. Modeling of parallel processing architectures. Based on evolving technology, a wide variety of system architectures can be envisioned. What parameters best characterize distributed processing architectures? Examples of possible parameters include processor instruction set and interprocessor communications capabilities.

2. Modeling of asynchronous parallel computational processes. A computational task can be solved by a variety of algorithms. What parameters capture the salient features of these alternative algorithms? Candidates include parameters such as data dependencies between component processes and sizes of data bases required by the process.

3. Evaluation of alternative architectures relative to classes of computational processes. How can the relationship between an architecture and an algorithm be measured? In particular, how can the parameters from (1) and (2) be used to evaluate the compatibility of an algorithm and an architecture?

The approach to the modeling of parallel processes and architectures is through examination of the parallelism in a variety of one- and two-dimensional signal processing tasks. This includes a study of the ways in which different types of digital signal processing tasks can be executed on different types of architectures. These studies aim to develop one set of features by which processes can be characterized, to develop another set of features by which parallel architectures can be characterized, and to use these features to obtain measures for the evaluation of process/architecture compatibility.

1.3 Approach

1.3.1 Modeling Asynchronous Parallel Computer System Architectures

The two basic classes of multiple processor systems are SIMD and MIMD (see Section 2.1). In the SIMD mode all processors operate synchronously using the same instructions, while in the MIMD mode all processors operate asynchronously, each using its own independent instructions. The types of asynchronous distributed processing systems of interest here come under the MIMD category, which is more general and flexible than the SIMD mode. In order to study the parallel/distributed implementation of a "total task," as opposed to component algorithms such as the FFT, the MIMD mode is more applicable than SIMD. Most real-time total tasks involve computations not suitable for the SIMD mode, although an MIMD system designed to process a total task may include SIMD machines as component processors. With the generality that the MIMD mode provides come the problems of task decomposition and system control, which will be examined in the problem domain of signal processing.

Our approach involves developing modeling tools for constructing large-scale distributed multiprocessor systems for signal processing tasks. These systems may be for a single task or a class of tasks. Depending on computational speed requirements and volume of data to be processed, some tasks will justify the construction of special purpose distributed systems for their execution. In other cases, one special purpose system which can be used for a set of tasks will be appropriate. In either case, each processor in the system will operate independently except for the sharing of data and results. Furthermore, each processor may be different. For example, a distributed system might contain several 8-bit microprocessors, pipelined array processors, SIMD machines, and 16-bit microprocessors with special floating point hardware, all cooperating to solve a particular problem. The microprocessors and other components may include off-the-shelf and/or custom designed VLSI chips; the design of large-scale distributed systems

provides an excellent vehicle for exploiting the predicted low cost and high complexity of VLSI hardware.

To illustrate the design of a special purpose distributed system with heterogeneous processors, consider again the simple hypothetical missile detection and tracking system shown in Figure 1.1. Assume that the principal system performance criterion requires that subtasks be executed as quickly as possible. Furthermore, assume that new seismic signals and multisensor imagery are real-time inputs to the system. Given these assumptions, in order to maximize execution speed, it would be appropriate to design a special-purpose distributed processing system for this task. Each subtask in the figure would be executed by a different processor or set of processors. For example, the FFTs could be done by pipelined array processors, the edge enhancement by an SIMD machine, and the object identification by a set of cooperating microprocessors. In addition, by designing the system so that each subtask takes approximately the same amount of time, the data could be pipelined through the system, i.e., each subtask processor (or processors) could contain data from a different input data set. In this simple example the design choices are fairly obvious. For more complex (and realistic) tasks, it is much more difficult to identify the component subtasks and match them with suitable architectures in such a way as to meet performance requirements.

Distributed computer systems can be characterized in different ways. To describe the computational capabilities of a system, the types of features which might be considered include:

- interprocessor connection network
- instruction set of each processor
- shared memory method and size
- individual processor memory sizes
- number of processors
- processor speed

- processor precision

Each of the types of architectural features may be divided into subcategories for analysis.

The approach taken in this research is to study the ways in which specific types of digital signal processing tasks can be executed on different types of architectures. From this, the salient architectural features should be identifiable. Existing computer system classification schemes and notational methods for describing computer systems are the starting point of this investigation. Our initial work in this area is presented in Chapters 2 and 3 of this report.

1.3.2 Modeling of Asynchronous Parallel Processes

One broad method of characterizing the parallelism in a process is by identifying the degree of parallelism. A process can be classified as being in one of three categories: it may be inherently parallel, it may possess limited parallelism, or it may be inherently serial. An example of an inherently parallel process is the task of detecting the appearance of a blip on a display screen. This task can be accomplished by a large number of subprocesses executing in parallel, with each subprocess examining only a portion of the screen. The parallelism in this task is limited only by the size of the screen and by the amount of the screen which must be available to a process in order for it to detect a blip. An example of a process which can be characterized as having limited parallelism is the task of summing N numbers, given that only pairwise additions can be performed. Initially N/2 pairwise additions can be performed simultaneously, producing N/2 intermediate sums. At the next step, N/4 pairwise additions can be performed in parallel. N/4 of the processors (adders) employed in the first step are no longer of use. The summation can be performed in at best $\log_2 N$ addition steps, with only $N/2^i$ pairwise additions performed in parallel in the i-th step, $1 \leq i \leq \log_2 N$. Inherently serial

processes arise from time dependencies among portions of the process. For example, in the missile detection and tracking scenario, the edge enhancement cannot be performed until the two dimensional filtering of the image has been completed.

Although the "degree of parallelism" measure provides some insight into the characteristics of a task being considered for parallel implementation, it does not adequately describe all of the factors which will enter into a parallel implementation. Consider the task of finding a tank in an image. One portion of the process may involve multiple feature analysis, e.g., the detection of both straight line and curved line segments. These two subtasks may be performed in parallel, but in order to do so, problems such as the sharing of data and communication of results must be considered. For this reason, a more detailed means of describing the processes is needed. Examples of the types of features which may be included in such a description are:

- representation of the data dependencies between component processes
- maximum number of independent subprocesses
- inter-process communication requirements
- specification of the ways in which subprocesses may be generated
- number and sizes of data bases required by the process
- type and precision of data

As seen in Chapter 4 of this report, high-level parallel programming languages have begun to appear which, to a greater or lesser extent, have facilities for describing such features. Also in Chapter 4, it is noted that data dependency relations can be expressed in terms of Petri nets [PeB74], S-nets [Kry81], and related graphical methods. It is our aim to consider the suitability of these and other schemes for representing the data dependencies so that process characteristics may be compared to architecture characteristics. Further considerations along these lines are discussed in Chapter 5 of this report.

1.3.3 Evaluation

The final aspect of this work will be to integrate the information provided by the models of parallel processes and parallel architectures. The following examples of the interaction of algorithm and architecture communication features from the SIMD domain illustrate the type of analyses to be performed.

In the 2-D FFT algorithm in [SiM79b], in order to transpose an array, processor i must be able to send a data word to processor i+k, for all i, $0 \leq i < N$, simultaneously, and for a fixed k, $1 \leq k < N$. This can be modeled at the process level as single word transfers using "shift" connections, where shift(x) = x+k mod N, $0 \leq x < N$. From an architectural point of view, this corresponds to packet switching [ThM79] with single word packet size and a "uniform shift" permutation capability. Multistage cube networks [SiM81] can perform uniform shifts efficiently and can be implemented in a packet switch mode [McS80b].

Parallel image smoothing [SiS81a], where each processor is assigned a square subimage to smooth, requires connections from each processor to its neighbors to transmit subimage edge data. This can be modeled at the process level as multiword shifts for $k = +1, -1, N^{1/2}, -N^{1/2}$ (for the edge data on the right, left, bottom, and top edges of the subimage), and single word shifts for $k = -N^{1/2}-1$, $-N^{1/2}+1$, $+N^{1/2}-1$, and $+N^{1/2}+1$ (for data at the four corners of the subimage). From an architectural point of view, this corresponds to the eight nearest neighbor connection scheme, and either circuit switching or packet switching with variable size packets [ThM79]. A single stage network, as in the Illiac IV [Bou72,Sie79] but with an eight neighbor mesh connection pattern, would be most efficient, implemented so that network settings can remain unchanged for multiword transfers.

Other investigators have discussed, to a limited extent, the relationships between architectural features and algorithms. Foster [Fos76] has examined the particular architectural features needed by an associative processing system in order to execute

efficiently particular fundamental algorithm constructions. He defines algorithmic features such as mode of address, depth of nest, interpass coupling, and number of internal operands. He then uses these features to show which associative processor instructions are needed to support different types of algorithms.

Gonzales [Gon78] discusses the need for techniques that permit an evaluation of distributed computing systems. He suggests that the following elements are required: a set of attributes, a measure of the extent to which a distributed structure possesses each attribute, and a measure of the relative importance of each attribute to the task being computed. He then discusses, in a general sense, what the components of such a quantitative approach should be.

Kuck and Padua [KuP79] have used measurements on programs to describe the performance characteristics of some general architecture schemes on different types of programs. Their goal is to use the measurements in programs from a particular application area to guide the design of special purpose multiprocessor systems for that area. The approach is based on decomposing serial programs into blocks based on data dependencies.

The algorithm characterization which we have been working on for the Defense Mapping Agency (DMA) [SwS80] is aimed at providing information towards matching image processing algorithms to SIMD architectures.

This work that has been done provides us with a basis for our architecture and algorithm modeling, but there are significant differences from this previous work in both our approach and objectives. For example, Foster's study is limited to associative parallel processors. Gonzales suggests types of information that would be useful in obtaining a quantitative evaluation of an architecture, but does not explore the specific features which would enable an architecture and an algorithm to be compared. Kuck and Padua base their program measurements on automatic transformations on serial FORTRAN programs, rather than examining tasks at the subtask and algorithm level

as we propose to do. Our DMA work is limited to the analysis of specific image processing algorithms rather than tasks, and is limited to SIMD parallelism rather than the more general MIMD mode.

Our approach here is to develop techniques by which the parameters which characterize a system architecture and the parameters which characterize a computational process can be objectively compared. The effect will be to define a measure or measures by which the suitability of a particular architecture for a particular algorithm or set of algorithms can be evaluated. There are three ways in which this measure could be used:

(1) Given a set of algorithms, evaluate a variety of architectures. Used in this way, the measure provides a design tool for the development of special purpose systems.

(2) Given an architecture, evaluate alternative algorithms for a particular processing task. This provides a design tool for the development of software for a computer system.

(3) Given an architecture and algorithm, assess the performance of the algorithm on that system. The performance measures used would be determined by the specific requirements of the intended application.

Numerous techniques exist for evaluating the complexity of serial algorithms [AhH76,Knu73]. Although measures of complexity of a serial algorithm include implicit assumptions about the model of computation, the asymptotic time complexity of an algorithm will be the same for any canonic serial computer model [AhH76]. These techniques for evaluating the complexity of a serial algorithm are therefore not directly applicable to parallel algorithms, where the architecture must be considered as a variable affecting the execution of the algorithm. We shall investigate techniques for obtaining theoretical lower bounds on the asymptotic time complexity of asynchronous processes, given specified architectures. These techniques will be based on the parameters derived in the models for parallel processes and architectures.

In order to evaluate the suitability of an architecture for a task or set of tasks, measures of "goodness" are required. Two possibilities are raw computational speed and cost-effectiveness. Maximum speed with which a task can be processed would be of interest for determining a lower bound on execution time. This is especially important for real-time tasks. By adding considerations such as the monetary implementation cost of the computer system, measures of cost-effectiveness can be obtained. However, factors such as reliability, maintainability, and accuracy must also be considered. We have studied such measures for SIMD parallelism [SiS82]. Our future work will include examining ways in which to incorporate such performance measures into an evaluation scheme based on the models of processes and architectures.

1.3.4 Signal Processing Task Analysis

A key aspect of our approach to the modeling process is to examine the parallelism in a variety of typical one- and two-dimensional signal processing tasks for the purposes of identifying what features can best be used to relate parallel processes to parallel architectures. The types of tasks to be studied include problems from the areas of radar processing, image processing, statistical and syntactic pattern recognition, speech understanding, and speech coding. Figure 1.2 outlines the analysis process for a task. The subtasks represent the major computational units into which the task can be decomposed. Some of the subtasks may be executable in parallel; some may have data dependencies that dictate sequential execution. For each of the subtasks, sets of alternative algorithms for performing the subtask are identified, and for each algorithm, alternative implementations are considered. At each level in the analysis structure, features which characterize the components and interrelations among components at that level are abstracted.

Chapter 6 of this report contains an initial look at three prototypical signal processing tasks from the image processing domain.

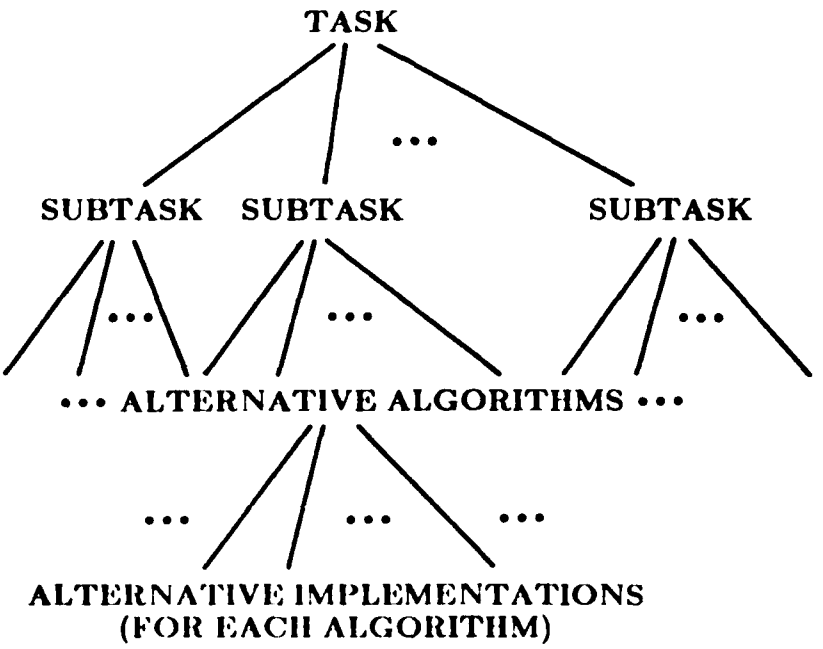


Figure 1.2 Task Analysis

1.4 Progress Summary

Our approach to date has combined both "top down" and "bottom up" strategies to the modeling problem. From a "top down" point of view, we have surveyed and analyzed high level representations of concurrency and existing architecture classification schemes. Starting with these known modeling tools, we are examining their applicability to problems of interest. From a "bottom up" perspective, we have considered the structuring of specific signal processing tasks for distributed execution in order to identify the salient attributes of the tasks and the corresponding architectures. From both approaches, we intend to extract features which characterize various aspects of distributed processes and architectures.

1.4.1 Modeling Architectures (Chapters 2,3)

Existing architectural classification schemes have been surveyed. These include general schemes, based on major system characteristics, and more detailed system description notations such as the processor-memory-switch (PMS) notation. Based on this survey, our approach to developing a comprehensive classification scheme will be to create a hierarchical classification system. A high level characterization will be followed by a more detailed description of the system's functional units and their organization. This scheme will involve combining and augmenting existing methods. Preliminary work has been done on the specification of this hierarchical system.

An extensive survey of multistage interconnection networks has been completed. From this, characteristic features of networks have been observed and a taxonomy of interconnection networks has been developed. Four aspects of multistage networks have been detailed: structural characteristics, distributed control schemes, implementation attributes, and fault tolerance.

1.4.2 High Level Descriptions of Concurrency in Processes (Chapter 4)

Concurrent programming languages and graphical representations are being studied for their usefulness in expressing the relevant characteristics of distributed processes. Languages considered include Ada, CSP, Concurrent Pascal, Path Pascal, Modula, and Edison. The languages have been examined with respect to their ability to represent some of the attributes which appear to be critical in designing concurrent implementations of signal processing tasks. These include provisions for local and global data, support of concurrent processes and the dynamic creation and termination of processes, specification of communication paths, and support of synchronization primitives.

Graphical representations are being studied for their ability to model various aspects of process synchronization. In asynchronous computation, the need for synchronization arises in various ways. A major process, after having spawned multiple subtasks, must resynchronize to coordinate the bringing together of results for use by the parent and/or subsequent processes. Our models must recognize data dependencies and include the overhead incurred by the resynchronization mechanism. We wish to develop models that take into account the different character of various synchronization methods, both in hardware and software. This is particularly relevant as it has been found that synchronization overhead may vary greatly for similar tasks executed on architectures that differ only in their synchronization mechanisms, so the software model should be accurate with various architectures. Another circumstance in which synchronization overhead must be considered is in the competition of multiple tasks for shared resources (peripherals, computation elements, shared data, etc.). Since the input to the systems under consideration is often random in nature, our models must include the stochastic behavior of this competition (the use of shared resources often cannot be prescheduled).

An extension of Synchronous Nets (which are themselves an extension of Petri nets) is being investigated for its utility in the asynchronous environment. Markov graphs or modifications thereof will be evaluated for use in stochastic modeling.

1.4.3 Features for Describing Processes and Architectures (Chapter 5)

Preliminary work has been done on integrating the information derived from the studies of concurrent languages, graph representations, architecture classification schemes, and task scenarios. Algorithm properties have been enumerated and candidate features to provide a "run time profile" have been identified. The features characterize such aspects of a task as uniformity of processing, global vs. local control, global vs. local data access, degree of parallelism, data set sizes, data types, and frequency of synchronization. Correspondences between algorithm/task features and architecture attributes are being developed.

1.4.4 Applications Studies (Chapter 6)

Parallel implementations of digital signal processing tasks are being designed in order to identify what features can best be used to relate parallel processes to parallel architectures. Three tasks from the image processing problem domain have been studied: contour extraction, shape recognition using Fourier descriptors, and computer vision. The contour extraction scenario involves edge detection, edge-guided thresholding, and contour tracing. It embraces both SIMD and MIMD subtasks, and is characterized by both local and non-local communications. The shape recognition task requires resampling, discrete Fourier transforms, global normalization operations, and library comparisons. It too can be implemented using a combination of SIMD and MIMD subtasks and requires a number of markedly different communications patterns. The computer vision task involves classification of the pixels of an image in order to delineate objects, followed by calculation of a number of parameters for each object,

including hole statistics, area, perimeter, and axis dimensions. A model of limited MIMD operation has been defined for this task, and simulations of the parallel vision algorithms have been performed. From these scenarios, significant attributes of the tasks which affect the parallel implementations have been observed.

CHAPTER 2

MODELING ARCHITECTURES: CLASSIFICATION SCHEMES

2.1 Introduction

Architectural classification and description schemes discussed in the literature vary considerably in the level of detail in which they treat systems. Consequently, the discussion of the different schemes is divided into subsections according to the level of detail. In Section 2.2, the general classification schemes of Feng [Fen72], Flynn [Fly66, Fly72], Kuck [Kuc80] and Shore [Sho73] are described. In Section 2.3, the more detailed structural system level description notations of Bell and Newell [BeN71] (PMS notation), Hockney and Jesshope [HoJ81], and Giloi [Gil81] are presented. In Section 2.4, some description and classification methods for computer subsystems are described. Included is Händler's ECS notation [Han77b, Han81] for describing processors and Ramamoorthy and Li's pipeline classification scheme. Siegel, McMillen, and Mueller's [Sim79a] taxonomy and parameters for describing networks and McMillen and Siegel's [McS80b] taxonomy for protocols are presented in Section 2.5. In Section 2.6, functional descriptions of computers are addressed in terms of instruction sets and data types supported directly. Bell and Newell's ISP notation [BeN71] and Giloi's Taxonomy [Gil81] are described. Finally, some recommendations toward combining a number of the more effective schemes together to produce a comprehensive, hierarchical architectural classification and description methodology are made in Section 2.7.

2.2 General Classification Schemes

2.2.1 Flynn's Classification

The oldest and most widely used scheme for classifying parallel computers has been proposed by Flynn [Fly66]. He divides computers into four groups depending on the number of concurrent instruction and data streams present. The simplest configuration is that used in a conventional serial processor which is classified *SISD* for *single instruction stream - single data stream.*

The first type of parallel system is classified as *SIMD, single instruction stream - multiple data stream.* Typically, an SIMD machine consists of a control unit, P processors, M memory modules (M is usually $\geq$ P), and an interconnection network. The control unit broadcasts instructions to all of the processors, and all active processors execute the same instruction at the same time. Thus there is a single instruction stream. Each active processor executes the instruction on data in its own associated memory module. Thus, there is a multiple data stream. The interconnection network sometimes referred to as an alignment or permutation network or switch, provides a communications facility for the processors and memory modules. A classic example of the SIMD organization is the Illiac IV [BaB68].

The second classification for parallel computers is *MISD* for *multiple instruction stream - single data stream.* In this type of an organization, a high bandwidth, dedicated execution unit is shared by a number of virtual machines. The virtual machines, operating independently on different programs, each have access to the execution hardware once per cycle. Thus there are multiple instruction streams and a single, interleaved data stream. An example of this is the peripheral processor units (PPM's) in the Control Data Corporation (CDC) 6600 [Tho70].

The *multiple instruction stream - multiple data stream* or MIMD organization is the last type defined by Flynn. An MIMD machine typically consists of P processors and M memories (M $\geq$ P), where each processor can follow an independent instruction

stream. As with SIMD machines, there is a multiple data stream and an interconnection network. Thus, there are N independent processors that can communicate among themselves. There may be a coordinator unit to oversee the activities of the processors.

Flynn allows pipelined computers to be placed in the SIMD category [Fly72]. However, some researchers [Han77b, HoJ81] believe that pipelined computers should be in a separate category and it has generally been the case that researchers using the term SIMD exclude pipelined processors. Since Flynn's classification is very broad, it is suitable for use only at the highest level in a hierarchical classification.

2.2.2 Feng's Classification

Feng's scheme for classifying computer architectures is based on the number of bits in a word and the number of words that are processed in parallel [Fen72]. These two simple measures of a system form a two dimensional feature plane in which a given computer is represented by a point in the plane. If x is the number of bits per word and y is the number of words operated upon in parallel, then (x,y) represents the computer. For example, the Illiac IV that was actually built is represented by (64, 64). STARAN [Bat74] would be represented by (256,1).

The intent of Feng's scheme is to distinguish among a variety of computer designs using two, easy to evaluate, features. The scheme is not, however, designed to expose the structure of a given computer. Thus, in the case of the STARAN, it could be concluded that it is a serial processor with an enormous word size. The fact that it is a powerful associative array computer is not readily apparent. Consequently, Feng's classification scheme is not well suited for use at a high level in the hierarchical scheme to be developed. On the other hand, the features of bits per word and words operated on in parallel are useful at a lower level of system description and will be incorporated there.

2.2.3 Shore's Classification

Shore divides computers into six classes or machines as illustrated in Figure 2.1 [Sho73]. His motivation in defining these machines is to refine Flynn's classification. Machine I consists of instruction memory (IM), a control unit (CU), a processing unit (PU), and a data memory (DM). The control unit fetches instructions from IM and the processing unit fetches and operates on words from DM. This is the organization of a conventional serial processor and corresponds to Flynn's SISD classification.

If the DM in machine I is rotated 90°, machine II is obtained. In this case, the processing unit accesses the same bit of all words in memory, i.e. a bit sice. STARAN is a good example of this type.

When parallelism is considered with respect to bits, Shore observes that there is little difference between machines of type I and II. If the data memories are square, performance will be the same in either case. In practice, however, there are typically *many more words* than there are bits per word. Consequently, machines of type II are capable of very high performance. It is partly for this reason that type II machines are considered "parallel" (since parallelism usually implies greater performance) whereas type I machines are not. Technically type II's can be classified as SIMD machines (and are considered as such by Hockney and Jesshope [HoJ81]) since there is one instruction stream and multiple (albeit bit serial) data streams. Machine type III is simply the combination of types I and II and has been called an *orthogonal* computer [Sho70]. Although the STARAN is capable of accessing words or bit slices from its 256 by 256 data memory, it does not have separate processors for dealing with the two formats. Thus it does not qualify as type III. The OMEN-60 series of computers do, however, belong to this class [Hig72].

Machine type IV is obtained by replicating PU-DM pairs. Each pair is connected to a single control unit. Since there are no connections between PU's, the only communication that can take place between PU's is through the CU. This configuration

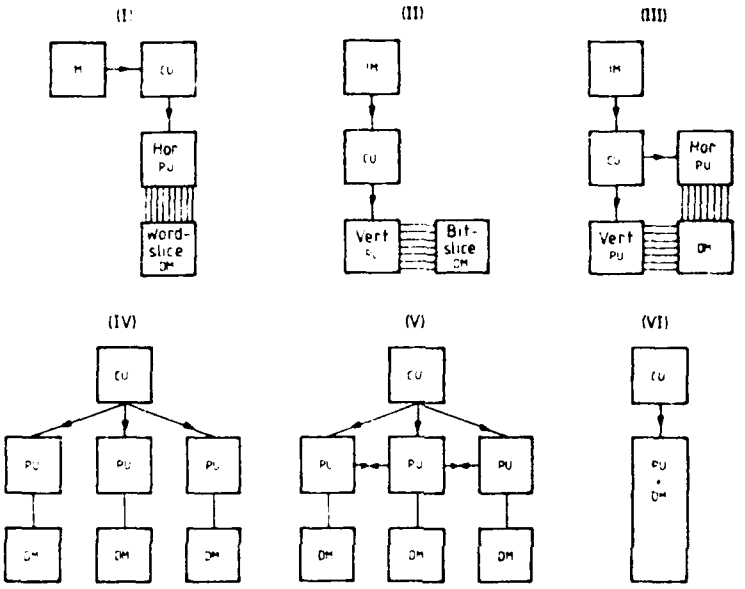


Figure 2.1 Block Diagram Representation of Six Machine Classes of Shore [Sho73]. Machine I, word serial, bit parallel; II, word parallel, bit serial; III, I and II combined, orthogonal computer; IV unconnected array; V, connected array; IV, logic-in-memory array (from [HoJ81]).

has previously been called an *ensemble*. PEPE [ViC78] has an architecture very similar to this except that there are three control units instead of one.

If near neighbor connections are added between the PU's in a type IV machine, a type V machine is obtained. The Illiac IV is such a machine. These computers are often referred to as *arrays* as opposed to ensembles due to the processors' ability to communicate with one another.

Since Shore's paper was written, considerable research has been done on interconnection networks that provide processor communication [cf. AnJ75, Fen81, Sim79a, Thu74]. Some of the methods that have been proposed are considerably more sophisticated than the linear near neighbor connections illustrated by Shore. Since the intended structure of the type V class is not altered by a more sophisticated connection scheme, it is considered appropriate to include array computers with interconnection networks in this class.

Finally, an array of logic in memory defines the type VI machine. Processing logic is associated with each bit of memory. Such machines are fully associative processors.

Illustrating just how "fuzzy" the line between serial and parallel processors is, Shore places the line between machine types III and IV and Hockney and Jesshope place it between I and II. The primary shortcoming of this scheme, as was the case with Flynn's, is the lack of provisions for pipelined designs. In addition, there is no provision for MIMD type organizations. The conclusion (as reached by Hockney and Jesshope [HoJ81]) is that Shore's machine types II - V are useful subdivisions of Flynn's SIMD category.

2.2.4 Kuck's Classification

Whereas Flynn categorized computers according to their instruction and data streams, Kuck proposes to classify them according to *instruction streams, instruction type, execution streams*, and *execution type* [Kuc78]. The instruction and execution

streams can be *single* or *multiple* and the instruction and execution types can be *scalar* or *array*. The number of instruction streams is determined by the number of programs that can be executed at once. It is assumed that each program requires a single register in some control unit to point to the next instruction to be executed. There is no restriction on the interaction between programs in the sense that they may be cooperating to achieve one final result. If the instructions used are essentially indistinguishable from those of a conventional uniprocessor, then the instruction type is scalar. If instructions explicitly refer to whole vectors or groups of operands (e.g. with base, limit, and increment as in the Texas Instruments Advanced Scientific Processor (TI ASC) [The74] or a vector register as in the CRAY-1 [Rus78]) then their type is array.

The number of *execution streams* is determined by the number of *operation types* that can be issued collectively at one time by all of the control units in the system. That is, regardless of the number of copies of an operation type issued, the number of execution streams depends only on the number of different types that can be issued simultaneously. Traditional operation types (usually specified by a single op-code in a uniprocessor) include store, load, fixed-point add, floating-point multiply, etc. The *execution type* is determined by the number of operands to which the instruction is applied. For example, in the Illiac IV, scalar type instructions are fetched by one control unit and broadcast to 64 execution units. Since only one instruction type is broadcast, there is a single execution stream. However, 64 operands are acted upon, so the execution type is array. The Illiac IV is thus classified as SISSEA or single instruction stream, scalar instruction type - single execution stream, array execution type.

To recapitulate, to characterize a system using Kuck's proposed scheme four parameters must be determined: the number of instruction streams (single or multiple), the instruction type (scalar or array), the number of execution streams (single or multiple), and the execution type (scalar or array). Since there are four parameters with two possible attributes each, 16 categories are defined and each is physically meaningful.

However, not all categories necessarily lead to desirable architectures. Table 2.1 lists the categories and many existing and proposed systems are shown as entries.

To a large extent, this classification scheme does a good job of separating different architectures into different categories while those in the same category are very similar in function if not in form. For example, in the SIAMEA category are the Burroughs Scientific Processor (BSP) [Sto77], CRAY-1, Control Data Corporation (CDC) 7600, and the TI ASC. The latter three all have multiple arithmetic pipelines whereas the BSP has 16 non-pipelined arithmetic units. Despite the difference in architecture, all machine's capabilities are approximately the same; in each case there can be 32 operands in some state of computation at one time (assuming dyadic computations are performed, i.e., there are two operands).

In the SISSEA category both the STARAN and Illiac IV machines are included, yet architecturally and functionally they are significantly different. Shore classifies the STARAN as type II (since it can be viewed as having a single processing unit with 256 bit registers) and the Illiac IV as type V. The simplest way to further classify these machines would be on the basis of the number of processor units they have: single and multiple respectively.

The PASM [SiS81a] and TRAC [SeU80] machines are reconfigurable and thus are listed in several categories. Their entries in categories not requiring all of their capabilities are shown in parentheses. PASM's most powerful mode in MISMEA and TRAC's is MIAMEA (i.e. TRAC's instruction set allows for explicitly referencing and operating on vectors [KaP80]).

Conceptually, data flow machines do not have instruction streams per se, only data streams. However, proposed implementations (e.g. [DeB80]) generally have some kind of instruction cells that receive operands, combine them with an instruction opcode and distribute the packets of data and instructions to execution units (the opcode may be simply the address of a single function arithmetic unit). Consequently

Table 2.1 Kuck's 16 Categories of Computer Architectures

		SINGLE EXECUTION		MULTIPLE EXECUTION	
	TYPE	SCALAR	ARRAY	SCALAR	ARRAY
SINGLE INSTRUCTION	SCALAR	PDP 11/45	ILLIAC IV STARAN (PASM) (TRAC)	CDC 6600 CPU	OMEN-60
	ARRAY	ZILOG Z80	CYBER 203/205	NONE KNOWN	CRAY-1 BSP CDC 7600 TIASC
MULTIPLE INSTRUCTION	SCALAR	CDC 6600 PPU	NONE KNOWN	BURROUGHS FMP DATA FLOW (PASM) (TRAC)	DENELCOR HEP PASM (TRAC)
	ARRAY	UNDESIRABLE DESIGN	NONE KNOWN	NONE KNOWN	PEPE CDC NASF TRAC PUMPS

there are multiple instruction streams which coincide with the execution streams.

There are four categories for which there are apparently no existing or proposed machines: SIAMES, MIAMES, MISSEA, and MIASEA. The first two categories imply an architecture in which single or multiple array type instructions are issued such that many operation types, to be applied to one pair of operands each, are specified by one array instruction. This represents an unusual instruction set which would be most likely found in a special purpose application, if it exists.

The MISSEA and MIASEA categories fall into the broader category of MISE machines - multiple instruction stream, single execution stream. This architecture implies that several instruction streams are interleaved on an instruction by instruction basis. An example in the MISSES category is the peripheral processing units (PPU's) in the CDC 6600. There are no known examples where this is done for array type instructions or arrays of operands. Indeed, this type of architecture is only appropriate when the execution units operate considerably faster than the rate at which instructions are issued by one control unit. In the MIASES category, it is very unlikely that streams of array instructions would need to be merged to keep a single execution unit busy.

In the SIASES category an architecture is implied in which array instructions are issued but execution takes place one operand (pair) at a time. This is useful for compressing the instruction set. An example is the Zilog Z80 microprocessor which has block move instructions. An array of data is to be moved, but it is done one element at a time. Such an instruction saves time since it only has to be fetched once.

Kuck's scheme is the most powerful of the general classification schemes examined. It is, however, more cumbersome to use. Thus, when no ambiguity will result, Flynn's scheme will be used in general discussions. When clarification is needed or a discussion is more detailed, Kuck's scheme will be used.

2.3 System Description Notations

System description notations are designed to indicate explicitly the architectural features of a given system. The three notations that are discussed here are (1) Giloi's notation [Gil81]; (2) processor-memory-switch or PMS notation [BeN71]; and (3) Hockney and Jesshope's notation [HoJ81] (which will be referred to as HOJ notation). Giloi's notation for describing architectures is not as detailed as the other two. PMS and HOJ notation are comparable in their descriptive power but there are important differences that will be described. All three notations are hierarchically organized and thus can be used to describe systems in varying levels of detail.

2.3.1 Giloi's Taxonomy

According to Giloi, [Gil81] there are two major features of a computer architecture: ***hardware structure*** and an ***operational principle***. The latter deals with the instruction set implemented and data types that can be directly manipulated. This aspect of the architecture will be discussed later (Subsection 2.6.2). The hardware structure is defined by the type of hardware resources and their number, an interconnection system, and a set of cooperation rules. Hardware resources include processors, memories, and peripheral devices; the interconnection system is comprised of all physical means by which hardware resources communicate; and cooperation rules govern communication and synchronization among the resources.

The hardware structure is specifically subdivided into a ***processor structure***, ***memory structure***, and ***communication structure***. The processor structure can contain a single or multiple processing sites. A single processing site consists of a conventional CPU, a multifunction processor, or a pipelined processor. Multiple processing sites can be arranged as an array of processing elements, a multiprocessor system, or a multicomputer system. Typically, a multiprocessing system consists of an array of processors which may each have some memory associated with them, whose primary purpose is

task execution. A multicomputer system consists of a number of computers that are linked together, where each computer contains a processor and its own memory. In addition, each may have I/O facilities and/or disk storage.

The memory structure consists of private and/or shared memory. The communication structure consists of memory sharing, a message switching bus, a message buffer memory, or a connection network. The cooperation rules used by multiple resources can implement a Master-Slave relationship, data flow, or cooperative antonomy.

Giloi's taxonomy is a useful way to view the components of a system at different levels of detail. However, there is no provision for explicitly indicating how the resources are connected together. Furthermore, the level of detail to which a system can be described is limited.

2.3.2 Bell and Newell's PMS Notation

Processor, memory, switch or PMS notation has been proposed by Bell and Newell [BeN71] for naming, describing, and interconnecting the parts of a computer system. The name of the notation comes from three of the primitives or basic component types used in the notation. There are a total of seven primitives:

M: *Memory* is a component used for storing information. It is not capable of altering the information it is given to store.

L: *Link* is a component that transfers information from one place to another without altering it.

S: *Switch* is a component that constructs links between other components. It has an associated set of L's that it enables or disables to make required connections.

D: A component that performs *data operations*. It can create and alter information. A classic example of this is an arithmetic unit.

K: *Control* is a component that evokes discrete operations of the other component in the system. With the exception of a processor (P), all the other primitives are passive and require activation by a K component.

T: *Transducer* is a component that transforms information without altering its meaning. For example, it might convert data from bit parallel to bit serial form.

P: *Processor* is a component that is capable of interpreting a program in order to execute a sequence of operations. Technically, it is not a primitive since it can be constructed from M, L, S, D, K, and T's. However it is such a fundamental part of systems, it is treated as a primitive.

The external environment is denoted by the symbol X. Components are connected with solid and broken lines (see Figure 2.2). Solid lines indicate flow of data and broken lines indicate transfer of control information. Lower case letters following a primitive symbol are used to differentiate between different types of a given primitive. For example, Pc is a central processor and Pio is an input/output processor. Mp is a primary memory and Ms is a secondary memory. A basic computer, C, is defined as:

$$C := Mp\text{-}Pc\text{-}T\text{-}X.$$

This notation is not usually linear. For example, if Pc is expanded into its components, the structure shown in Figure 2.2 results.

A considerable level of detail can be achieved by describing the attributes of a component in parentheses, adjacent to its symbol. For example, the processor in an IBM 370/165 can be represented as [BeN71]:

Pc (model: '165; cycle time: 80ns; data paths: 64 bits; cooling: water).

An attribute such as the type of cooling used is useful for documentation purposes, but

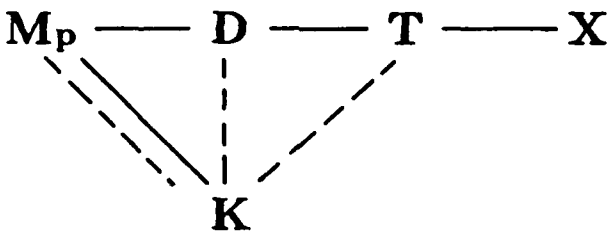


Figure 2.2 Basic Structure of a Computer Using Primitive Components of PMS Notation

is superfluous with regard to evaluating the architecture's ability to execute an algorithm.

A rigorous definition of PMS notation is given in the appendix in [BeN71]; it is too lengthy to include here. As an example of the capabilities of this notation, a detailed description of the Digital Equipment Corporation PDP-8 is shown in Figure 2.3. The main drawback to this notation is readily apparent from the figure: it is two dimensional. It does not lend itself to being embedded in text or stored in a computer (i.e. for analysis of the represented system's capabilities). The notation presented in the next section solves this problem.

2.3.3 Hockney and Jesshope's Notation

Hockney and Jesshope have developed a notation specifically designed to allow a one-line or few-line description of an algebraic style amenable to printing and entry into a computer [HoJ81]. HOJ notation represents a computer as a number of functional units that manipulate data, are connected by data paths, and are controlled by instruction units. Using their notation, a simple serial computer is represented by:

$$C = I[E-M].$$

This means that the computer, C, consists of a single instruction unit that controls the units in brackets. Those units are an execution unit, E, that performs arithmetic, connected to one memory unit, M, by a single path (the solid line). There are a total of 20 rule types that define symbols, govern their use, and how they are to be connected. A summary of them is given below along with some examples.

B: Boolean, integer, or fixed-point execution unit

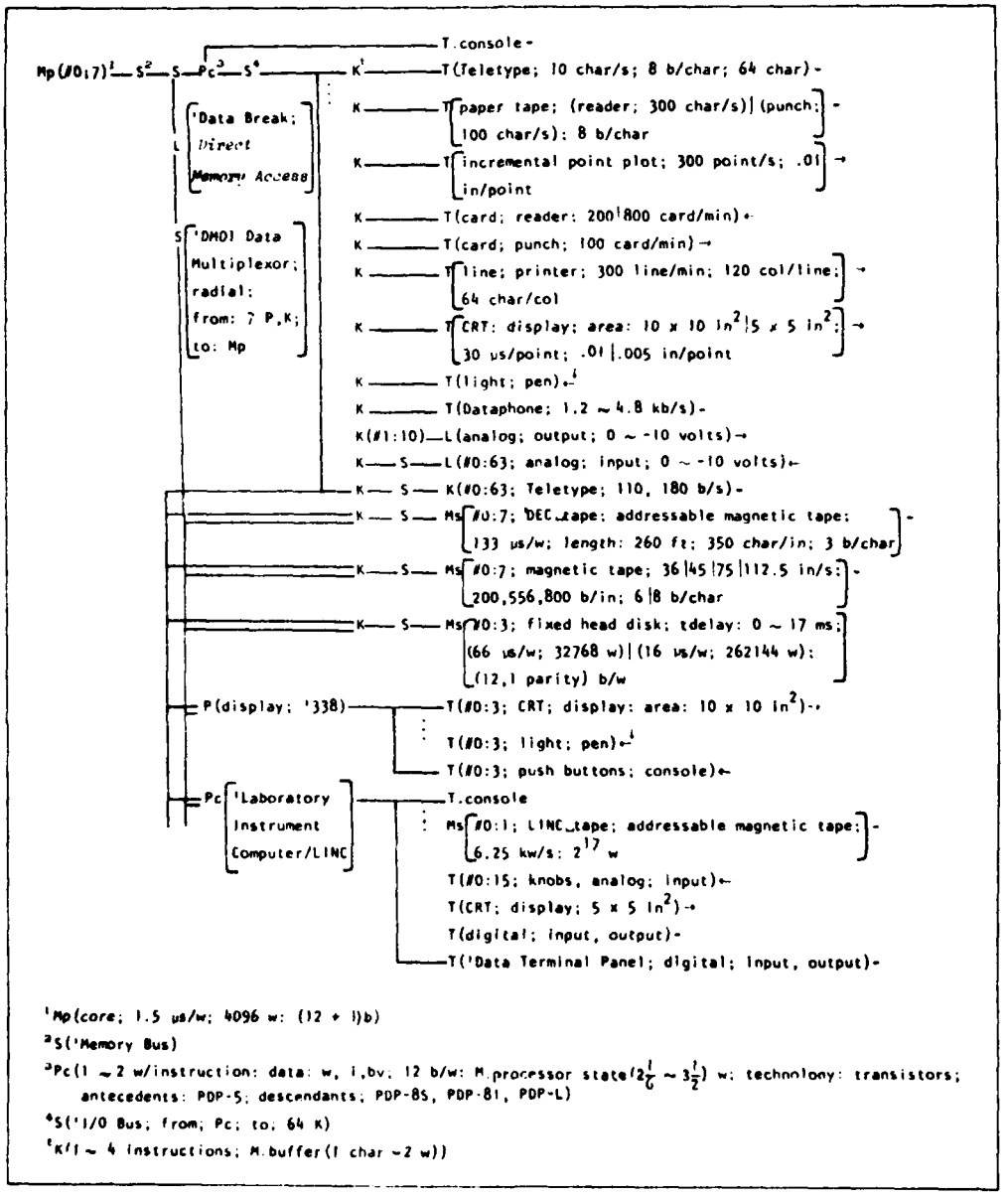


Figure 2.3. PMS Diagram of Digital Equipment Corporation PDP-8 (from [BeN71])

C: A computer containing at least one I unit.

Ch: An I/O channel that can operate independent of other units.

D: An I/O device (e.g. disk). Its type is specified in parentheses.

E: An execution unit (i.e. ALU). Can be specified as an F or B unit.

F: A floating-point execution unit.

H: A data highway or switching unit. It does not alter data other than to possibly reorder it (e.g. the Flip network in STARAN [Bat76]).

I: An instruction unit. It decodes a single instruction stream and sends commands to execution units within its scope of control. Instructions are sequenced with a single instruction counter.

IO: An interface to an I/O device.

M: A memory unit for storing data and/or instructions (e.g., registers, cache, main memory).

O: An orthogonal memory (two dimensional).

P: A processor with at least one execution unit but no instruction units.

U: An unspecified unit. To be used when the unit to be described fits into none of the above categories. A description is placed in parentheses adjacent to the U.

Xp: A pipelined unit (X is one of the above)

Iv: The I unit can interpret vector instructions. (v is the last symbol if the unit is also pipelined, i.e. Ipv).

Xn: n is any integer, used to differentiate between multiple units of the same type, e.g. Ipv1, Ipv2.

nX: n is an integer indicating the number of units of the same type that may operate simultaneously.

$n\overline{X}$: Indicates that multiple units of the same type are identical, e.g. $64\overline{P}$ represents 64 identical processors as in the Illiac IV [Bou72].

$\{\overline{expr}\}$: Indicates identical replication of a group of units defined by expr, e.g. $64\{\overline{E-M}\}$ represents $64\overline{P}$ in expanded form.

{X,...,Y}: A group of units that can operate concurrently.

{X,Y,Z}p: Units or operations (e.g. memory read) X, Y, and Z that operate or are executed in an overlapped (pipelined) fashion.

{X/.../Y}: A group of units that can only operate one at a time.

Example: {4F1/B1}; 4F1 = {F(+),F(*),F(1/x),F(sqrt)}; B1 = {B(+)/B(shift)}. This illustrates four floating point units that operate concurrently relative to each other, but sequentially with respect to a fixed-point unit. The floating point units consist of add, multiply, reciprocal, and square root units. The fixed point unit can perform an add or a shift but not both at the same time.

X_b: Subscript b is the number of bits on which unit X operates, X ≠ M or O.

nM_{w*b}: n is the number of one dimensional memory banks, each of which contains w words, b bits wide.

O_{w*b}: An orthogonal memory w words by b bits in size. It either delivers a b bit, word-slice or a w bit, bit-slice.

X^t: t is the characteristic time of the unit in nanoseconds unless otherwise noted. If X = I, t is the clock period, if X = E, t is the average execution time, and if x = M, t is the access time.

–: An unspecified connection.

->,<-: Unidirectional connections.

<->: Full duplex connection.

<-/->: Half duplex connection.

$\frac{t}{n*\{d+a\}}$: t is the transfer rate of n identical buses with d data bits and a address bits each.

Example: Complex communication structures can be defined in a nested fashion using the highway type: H3 = $\{\{\frac{}{16}>,<\frac{}{16}\}/<\frac{}{32}\}$. The bus denoted by H3 can be operated with 16 bits travelling each direction or with 32 bits travelling to the left. That is the direction of half of the bus can be reversed.

X - Y - Z: Series connection.

-X|: Connection on the left but not the right (| is optional if there is no ambiguity).

|X -: Connection on the right but not on the left.

-{-X-, -Y-}-: Units X and Y are connected in parallel.

-{X,Y}: Group of units that can operate concurrently, connected to a single bus in an unspecified way.

-{-X- / -Y-}-: Two parallel paths that can be used only *one at a time.*

Example: Very complex structures can be specified using this notation. The structure illustrated in Figure 2.4 is represented by the expression:

X1-{-X2|,-X3-{-X7/<-}-,-{-X4-,-X5-X6-}-{-X8-/->}-}-X9.

X-a: A unit can be attached to a *connection point* represented by a lower case letter. This is used for connecting very complex structures.

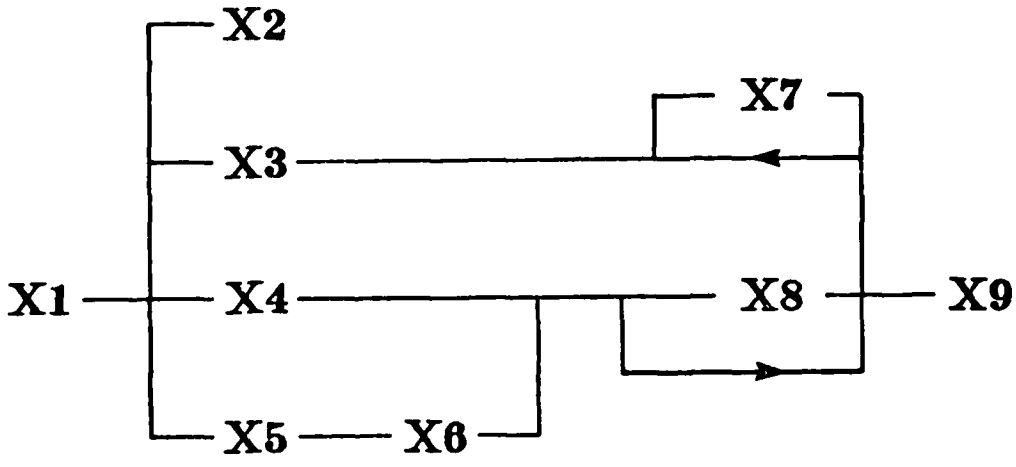


Figure 2.4 Hypothetical Arrangement of Nine Functional Units

$X-\{a,b,c\}$: Unit X is attached to connection points a, b, and c.

nX^{1-nn}: n units with first near neighbor connection, i.e. four nearest neighbors.

nX^{2-nn}: Second nearest neighbor connection is used, i.e., to the eight nearest neighbors.

nX^{0-nn}: There is no connection.

$nX \times mY$: Cross connection between nX units and mY units via a crossbar or switching network.

(Comment): Comments are placed in parentheses and used to clarify descriptions or how connections are to be made.

Example: A memory hierarchy could be represented by:

$$M1^{10}(\text{bipolar}) - M2^{500}(\text{CMOS}) - D1^{(1ms)}(\text{fixed head disk}).$$

A system with the Generalized Cube multistage interconnection network [SiM81b] could be represented as:

$$64\bar{E} \times (\text{Generalized cube}) \times 64\bar{M} .$$

$P = E - M$: Simplest form of a processor; there is no I unit.

$C = I[E - M]$: Simplest form of a computer. Since it contains an I unit, it can control other units. For example, $C[64\bar{P}]$ represents a computer with control over 64 identical processors.

$X[\ ]_a$: Control is asynchronous; $X \epsilon \{I,C\}$.

$X[\ ]_h$: Control is horizontal. One instruction controls several different units at each cycle.

$X[\ ]_l$: Control is lockstep or synchronous

$X[\ ]_r$: "Issue when ready." An instruction is issued when the execution unit and required registers are available, e.g., the CRAY-1 [Rus78].

A complete formal specification of HOJ notation in Backus Normal Form (BNF) is given in the appendix in [HoJ81].

This notation is very powerful and well suited to representing those features of an architecture that determine how well an algorithm will execute on it. Table 2.2 illustrates HOJ notation's ability to represent a wide variety of systems and its compatibility with Kuck's classification scheme. Kuck's 16 categories are listed and a representative structure is given for each in HOJ notation. Where an example system is included for a particular category, the HOJ representation is given specifically for that system.

The correspondence between Kuck's four features and notational components of HOJ notation is apparent from Table 2.2. The number of I units determines the number of instruction streams. The presence or absence of a "v" on an I unit determines whether the instruction type is array or scalar, respectively.

The number of E units determines whether the execution type is scalar or array. If all E units in the system are identical (i.e. there is a single $\bar{E}$, $\bar{F}$, $\bar{B}$, or $\bar{P}$ in the representation) then there is a single execution stream. If there are multiple, independent execution units (e.g., one I unit and multiple E units or multiple identical C units) then there are multiple execution streams. Because of this relationship between HOJ notation and Kuck's taxonomy, once a system has been described in HOJ notation (even at just a coarse level), it is very easy to classify.

2.4 Processor Description

Some research on classification methods has concentrated on specific portions of a computer system. Described below is Händler's Erlangen Classification Scheme (ECS) [Han77b, Han81] which concentrates on processors (computers according to HOJ

Table 2.2 Relationship Between Kuck's Taxonomy and Hockney and Jesshope's Notation

CATEGORY	EXAMPLE SYSTEM	REPRESENTATIVE STRUCTURE
SISSES	IBM 7090	$I[F_{36}-M]$
SIASES	ZILOG Z80	$Iv_8^{250}[B_8 \overline{-}_8 M_{18*8}] \overline{-}_8 M_{64K*8}$
MISSES	CDC 6600 PPU	$10I[E - M]$
MIASES		$10Iv[E - M]$
SISSEA	ILLIAC IV	$C1^{80}[64\overline{P}]_l^{1\ 00}$; $P=F_{64}^{600}-M_{2K*64}^{400}$
SIASEA	Cyber 205	$Iv^{20}[4\overline{F}p_{64}-512M_{16K*32}^{80}]$
MISSEA		$5I[3\overline{P}]_l$
MIASEA		$5Iv[3\overline{P}]_l$
SISMES	CDC 6600	$I[\{4F_{60}, 6B\} - 32M - 10\overline{P}]_r$
SIAMES		$Iv[10E - 20M]$
MISMES	BURROUGHS FMP	$C1 - C2 - 512\overline{C3}$ × (OMEGA NETWORK) × $521\overline{M}$; $C3=Ip^{40}[\{F,B\} - M_{32K*48}^{120}$ (ALSO 7 BIT SECDED)]
MIAMES		$256\overline{C}v$ × (ADM NETWORK) × $256\ \overline{M}$
SISMEA	OMEN-64	$I_{16}[64\overline{B}_1-O_{64*16}-E_{16}]_l$
SIAMEA	CRAY-1	$Iv^{12}[12Ep^{12}-16M^{50}]_r$; $12Ep=\{3Fp_{64}, 9B\}$
MISMEA	PASM	C1(HOST)–C2–$16\overline{C3}${× (MULTISTAGE NETWORK), –C4}; C3={C5$[64\overline{P}]_l$/C5 – $64\{\overline{I1}[\overline{P}]\}$}; C5=I2[B–{M(A)/M(B)}]; P=E–{M(A)/M(B)}; C4=$64\overline{C}$ (MSU)–5C(MEMORY MANAGEMENT); C(MSU)=C6 – D(DISK)
MIAMEA	TRAC	$16\overline{C}vI_8$ ×(SW-BANYAN, S=2, F=3)× (MIO); (MIO)={$64\overline{P}$, $16\overline{IO}$, D(DISK)}; P=B–M_{4K*8}

terminology since I units are considered) and their structure, ignoring memory and interconnection. The ECS takes all forms of pipelining explicitly into account. Also discussed is Ramamoorthy and Li's classification of pipelining methods [RaL77].

2.4.1 Händler's Erlangen Classification Scheme

The main purpose of the ECS is to account for all forms of parallelism in the system to be classified. As with the approach taken by Shore (see Subsection B.3), the basic unit of information considered is one bit. The processing hardware associated with one bit is called an *Elementary Logic Circuit (ELC)*. This is the lowest level of processing that is distinguished. The next higher level of processing is that of the *Arithmetic and Logical Unit (ALU)* and the highest is that of the *Program Control Unit (PCU)*. The PCU interprets program instructions (one at a time) and issues directives to one or more ALUs. Each ALU executes the directives or microinstructions. Microinstructions are made up of *microoperations which initiate elementary switching* operations that are performed by ELCs. The number of PCUs, ALUs per PCU and ELCs per ALU form a triple denoted by (k, d, w). A very simple early computer called MINIMA [Han77b] is classified by the triple (1, 1, 1). A conventional serial computer such as the IBM 701 is classified as (1, 1, 36). The Illiac IV is (1, 64, 64) and the STARAN is (1, 256, 1). This information is directly imbedded in HOJ notation. The classification (k, d, w) corresponds to the HOJ structure: kC; $C = I[d\bar{E}_w - M]$. (Note: this specification is not the same as $kI[d\bar{E}_w - M]$ because this implies that k I units have control over d E units. To be equivalent, braces need to be inserted: $k\{I[d\bar{E}_w - M]\}$.) The $-M$ means that the E units are connected to memory of an unspecified configuration. In the case of the Illiac IV, for example, the computer would be specified as: $C = I[d\{\overline{E_w - M}\}]$.

Händler notes that each of the components in the triple can be pipelined. Pipelining at the ELC level corresponds to arithmetic pipelining as in a pipelined floating

point unit. Pipelining at the ALU level is instruction pipelining and at the PCU level is called macro-pipelining. The number of units that operate concurrently in a pipe is indicated by the variables w', d', and k', respectively. Incorporating these parameters into the triple yields ($k \times k'$, $d \times d'$, $w \times w'$). This takes both horizontal (multiple units) and vertical (pipelined units) forms of parallelism into account.

In HOJ notation, ELC level pipelining is indicated by a lower case p that immediately follows any execution unit symbol (e.g. Fp). It does not indicate the number of stages in the pipe, though the number of bits is included. A useful addition to the notation is to include this information in the form: $X_{w*w'}$, where $X \epsilon \{E,F,B\}$, w is the number of bits and w' is the number of stages in the pipe. If w' is undetermined or not known, then the form Xp_w should still be used.

Händler measures the degree of instruction pipelining in terms of the number of independent function units available to execute an instruction. Thus, the CDC 6600 is classified as (1×1, 1×10, 60×1) since there are ten different arithmetic units. Instruction pipelining occurs because one instruction is decoded before the previous one is finished. If the second instruction does not need the same function unit and is independent of the first, it can begin execution immediately. Fetch, decode, and execute cycles are thus overlapped. The d' parameter measures the amount of hardware involved, but it does not indicate the level of speed up potentially possible or typically obtainable. In the case of the 6600, typical speed up is a factor of 2.6, not 10. For the purposes of this report, it would be more useful if d' measured the number of stages in an instruction pipe.

In HOJ notation, the parameters d and d' as defined by Händler are incorporated. The d' measure is implied by multiple, non-identical execution units, e.g., $d'E$, whereas d is implied by identical units, e.g., $d\bar{E}$. HOJ notation can easily be extended to allow the number of stages in an instruction pipe to be added as a subscript on an I unit in the form $I_{b*b'}$, where b is the number of bits and b' is the number of stages in the pipe.

Ip_b is to be used when b' is unknown or undetermined.

Macropipelining can be performed when a set of data is to be processed sequentially by more than one task. Each task can reside in a different processor. As the first task produces results, they are placed in a memory to which the second task also has access. The second task can begin processing intermediate results before the first task has finished, and so on. In the case where no results are available from a task until all results are, if there are several independent sets of data to be processed (e.g. a sequence of images) this method is still effective in speeding up the process time.

Macropipelining can be indicated explicitly or implicitly in HOJ notation. If the macropipelining factor is k', then it can be represented explicitly as k' serially linked computer/shared memory pairs. For example, for $k' = 3$:

$$C1 \rightarrow M \rightarrow C2 \rightarrow M \rightarrow C3 \rightarrow M .$$

Macropipelining is represented implicitly by the structure:

$$k'C \times (NETWORK) \times k'M .$$

Depending on the implementation details, as long as the computers operate independently, macropipelining can be achieved on this type of structure. This structure is more flexible than the previous since it is not limited to macropipelining.

The last aspect of ECS is a method of indicating combinations of different computers and those whose structure is reconfigurable. Two operators are used for the former: "+" (concurrence) and "×" (pipelining). The operators are used to connect triples. For example, (4, 1, 16) can also be represented as (1, 1, 16) + (1, 1, 16) + (1, 1, 16) + (1, 1, 16). The concurrence operator is most useful when the computers are different. A good example of the pipelining operator is illustrated in the representation of the CDC 6600: (10, 1, 12) × (1, 1 × 10, 60) (pipelining terms are omitted from a triple when pipelining is not present). The term on the left is for the peripheral processing units (PPUs) through which all programs must pass

to be executed by the main processor.

The "V" (logical OR) symbol is used to indicate different configurations of the same hardware. For example, the C.mmp system [WuB72] is represented as (16, 1, 16) V (1 x 16, 1, 16) V (1, 16, 16). For reconfigurable systems like PASM (SiS81a), TRAC[SeU80], and the Dynamic Computer [KaK79], this notation becomes cumbersome because of the large number of possible configurations. For example, the Dynamic Computer has a variable word width in multiples of 16, which is the number of bits contained in the basic computer. In a size N dynamic computer group there are 2^{N-1} ways to configure the system into from one to N independent virtual computers of varying word sizes. In addition, the independent virtual computers can be linked together in a wide variety of combinations. Expressing all the possibilities in ECS would be tedious and of questionable value.

The preferred approach is to represent reconfigurable systems in HOJ notation in sufficient detail to expose the legal configurations. For example, the structure for PASM in Table 2.2 shows two mutually exclusive configurations for a C3 computer. Since there are 16 C3 computers, there are 2^{16} = 64K (K = 1024) possible configurations. Thus the number of configurations can be derived from a sufficiently detailed HOJ representation.

2.4.2 Ramamoorthy and Li's Pipeline Classification Scheme

In [RaL77], Ramamoorthy and Li distinguish between two levels of pipelining that correspond to Händler's level 2 (instruction) and level 3 (arithmetic). They further distinguish between *unifunction* and *multifunction* pipes. The former is capable of performing only one kind of operation, e.g. multiplication. The latter can perform several different operations, e.g., floating point addition and subtraction. Multifunction pipes can be subdivided into *static* and *dynamic* categories. A static pipe can only perform one operation at a given time. Thus all instructions wishing to use a pipe at the same

time must use the same configuration. A unifunction pipe is static by definition. A dynamic multifunction pipe allows overlapped processing among instructions using different configurations. The control of a dynamic pipe is much more complex than that of a static pipe.

The last distinction is between *scalar* and *vector* arithmetic pipes. Scalar pipes accept operand pairs as they become available from the instruction unit. Vector pipes are augmented with hardware specifically designed to accommodate vectors or arrays of numbers stored sequentially in memory. They are usually equipped with registers for storing base addresses, offsets and vector length. The main advantage of a vector pipe over a scalar pipe is simplified address generation which leads to faster overall execution.

These additional parameters can easily be incorporated into HOJ notation either formally or informally. Informally they can be included as comments with the pipelined unit. For example:

$$\text{Iv}[\{\text{Fp}(+, -, *, \div, \text{dynamic, vector}),\text{Bp}(+, -, *, \text{static, scalar})\} - \text{M}]_{\text{r}} .$$

Whether the unit is uni- or multifunction is implicit in the list of functions it can perform. The specification can be made formal (and more compact) by replacing the "p" with two lowercase letters. The first is a "d" or an "s" and the second is a "v" or an "s." The previous example becomes:

$$\text{Iv}[\{\text{Fdv}(+, -, *, \div), \text{Bss}(+, -, *)\} - \text{M}]_{\text{r}} .$$

If any of the execution units are vector type, then the I unit has vector instructions. The converse, however is not true. For example, the structure $\text{Iv}[10\{\overline{\text{F}-\text{M}}\}]$ is possible.

2.5 Interconnection Network Description

In this section, ways to characterize two different aspects of interconnection networks are discussed. First, their structure is examined. A taxonomy used by Siegel, McMillen, and Mueller [Sim79a] is described. Also, parameters they used to compare a variety of networks are discussed. The second aspect of interconnection networks addressed is protocol. A taxonomy of network protocols proposed by McMillen and Siegel [McS80b] is presented. In the chapter that follows, multistage interconnection networks are surveyed extensively.

2.5.1 Siegel, McMillen, and Mueller's Taxonomy and Parameters

In their survey of interconnection methods, Siegel, McMillen, and Mueller [Sim79a] organize the networks to be discussed according to the taxonomy shown in Figure 2.5 (the single stage category has been added for completeness). The classifications are based on the differences between physical implementations. Before discussing the taxonomy, it is useful to define what is meant by "path" and "switching element." Anderson and Jensen [AnJ75] define a *path* as the "the medium by which a message is transferred between the other system elements" (e.g., wires or buses), and a *switching element* as "an entity which may be thought of as an 'intervening intelligence' between the sender and receiver of a message." Networks can be described by the type of switching elements used and the paths between switching elements.

As shown in Figure 2.5, interconnection networks can be separated into staged and direct path categories. In a *staged* network, a message typically passes through a number of switching elements on its way to its destination. These networks can always be designed so that the path between switching elements is dedicated to the pair connected [McS82b]. In a *direct path* network, aside from interfaces, a message typically traverses paths only in moving from one processor to another. In the case of a hierarchical organization (which is discussed further below) a message may shift levels

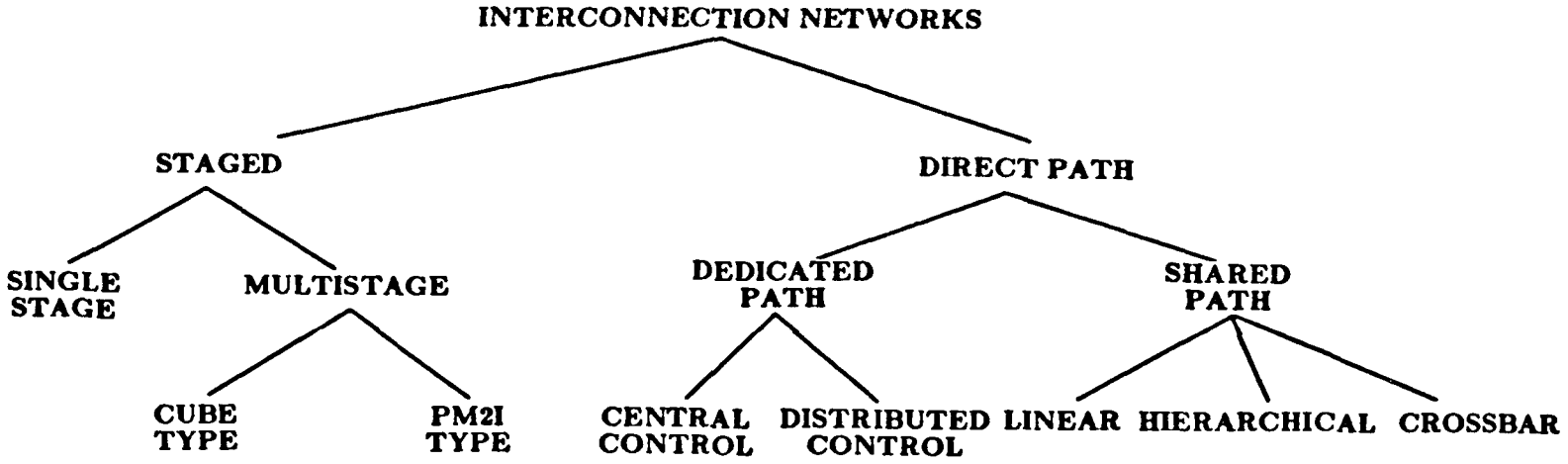


Figure 2.5 Interconnection Network Taxonomy

via a mapping element which is considered a switching element (e.g., the Cm* system [SwF77]). In this case, however, many switching elements usually share one path. In a direct path network, it is possible that a message would have to pass through intermediate processors on its way to its destination. Assume all the networks to be discussed have N inputs and N outputs.

Staged networks are subdivided into the single stage and multistage groups. A *single stage* network consists of one column of switching elements. Paths are arranged, connecting outputs to inputs, so that a message can be routed from any input to any output by recirculating enough times through the network. Such networks usually have a small upper bound on the number of passes required (e.g., $\log_2 N$). In this type of implementation, no intermediate processors are involved in handling a message between source and destination. Examples of this are in [ChY82, LaS76].

A *multistage* network consists of several columns of switching elements. Such networks usually have at least $\log_b N$ stages or columns where b is the number of input or output ports of one switching element. These networks can be further divided into cube type and PM2I type. These types and many examples are discussed in detail in the next chapter.

Direct path networks can be subdivided into dedicated path and shared path groups. A *dedicated path* network has direct links between pairs of processors. Typically, a message has to pass through several processors to reach its desired destination. Only in the case where all processors have a direct link to all other processors (fully connected) do messages totally avoid intermediate processors. An example of a system with a dedicated path network is CHOPP [SuB77]. Control of such networks can be centralized or distributed via routing tags (as is done in CHOPP).

A *shared path* network is typified by a bus where several devices (e.g., processors) share its use via time multiplexing. Shared path networks are subdivided into linear, hierarchical and crossbar configurations. A single bus or multiple buses at the same

level form *linear* shared paths. When several buses are combined so that high level buses carry traffic between low level buses the configuration is *hierarchical.* A switching element is usually required to switch traffic between levels. When there is a bus for each input and each output, totaling 2N buses, with a crosspoint (on/off switch) between every input and output bus, a *crossbar* structure results.

There are mathematical functions called interconnection functions (that will be described in detail in Chapter 3) that can be used to describe the pattern of connections used in a network. Two such functions are the "cube" and "PM2I" (plus-minus 2^i). It should be noted that different classes of networks can be based on the same family of functions (e.g., single stage, multistage, and linear direct path networks can all be based on cube functions).

Parameters that can be used to describe or quantify a network as described in [Sim79a] are summarized as follows. The *communications setup method* is the method used to establish an interprocessor communications path. *Delay* is the time it takes a network to transfer one data item from a source to the desired destination. The *ease of use* of a network is the degree to which connections are automatically established. The *cost* of a network is the asymptotic complexity of its implementation.

The *partitionability* of a network is its ability to divide the system into independent subsystems of different sizes. Partitionable systems may be characterized by any limitation on the subset of processors which may belong to a partition. Furthermore, a system may be *logically partitioned* using software techniques or *physically partitioned* using hardware switches within the network control structure. A network is *homogeneous* if it treats all processors similarly. *Modularity* is the ability of a network to be constructed from a small set of basic modules. *VLSI compatibility* is the suitability of a module to be implemented as an LSI chip, i.e., high-circuit complexity and low external connection requirements. The *extensibility* of a network is its ability to be extended to a larger size, i.e., the amount of modification needed to make the network function for

a larger number of inputs/outputs. *Fault tolerance* is measured in terms of a system's features which would allow the system to remain operational with faulty components (with possible degradation).

Let m be the number of processors which can transfer data simultaneously using the interconnection network. Then the degree of *simultaneity* supported by the interconnection network is $S = m/N$, $1 \leq m \leq N$. Permutations are one-to-one connections in which all processors participate. For networks with N inputs, N outputs, and S=1, let r be the number of permutations possible in a single pass through an interconnection network. Then the *connectivity* of the network is $C=r/(N!)$, $1 \leq r \leq N!$. The ability of a processor attached to the network to broadcast a single data item to all other processors can be measured by the *broadcast scope*. Let b be the maximum number of other processors which can receive data simultaneously from a given processor after one pass through the interconnection network. Then the broadcast scope is $B=b/(N-1)$. The *broadcast delay* is the number of transfers required for a complete broadcast. The *range* of a network can be measured by $R=x/(N-1)$, where x is the order of the set of processors (i.e., the number of processors) from which a single processor can choose to send data to in one pass through the network. The range can be further characterized by specifying the set of processors which can be sent data. Similarly the *domain* of a network can be measured by $D=x/(N-1)$, where x is the order of the set of processors a single processor can receive data from in one pass through the network, and can be further characterized by specifying the set of processors which can send the data.

The parameters likely to be the most useful in determining how well a computer using a given interconnection network can perform a given algorithm are delay, ease of use, partitionability, and simultaneously. Other parameters which may be useful are connectivity, broadcast scope and delay, range, and domain. They may be more useful in a different form, however. For example, a measure or characterization of the

network's ability to broadcast to a subset of all possible destinations may be more useful than the broadcast scope measure.

2.5.2 McMillen and Siegel's Protocols

There are two basic kinds of switching that can be used in an interconnection network, circuit switching and packet switching. *Circuit switching* is a mode of communication in which a complete path is established from an input port to an output port before any information is transmitted. *Packet switching* is a mode in which relatively small units of information called *packets* move from switching element to switching element as paths between switches become available. Packets do not require their entire path to be established prior to entering the network.

Some of the interconnection network types discussed in the last subsection are inherently limited to one mode of switching or the other. Single stage networks are packet switched due to the fact that several passes through the network may be required to reach the destination. Linear shared path networks are circuit switched since a circuit is always established between pairs of communicating devices. All of the remaining network types can be implemented either way. The following discussion of circuit and packet switching protocols (as presented in [McS80b]) only applies to those networks for which such an implementation is practical.

The options to be discussed are primarily concerned with packet switching. In circuit switching networks, the design of the switching elements is more straightforward. Design options in the circuit switching case are primarily concerned with: (1) implementation of the interface between the network and the devices it serves; and (2) protocol between sending device and receiving device upon establishment of the connecting circuit. Since the emphasis here is on switching element protocol, these issues are not discussed in detail. The options that will be discussed are shown in Figure 2.6. They include packet versus circuit switching, synchronous versus asynchronous request/grant

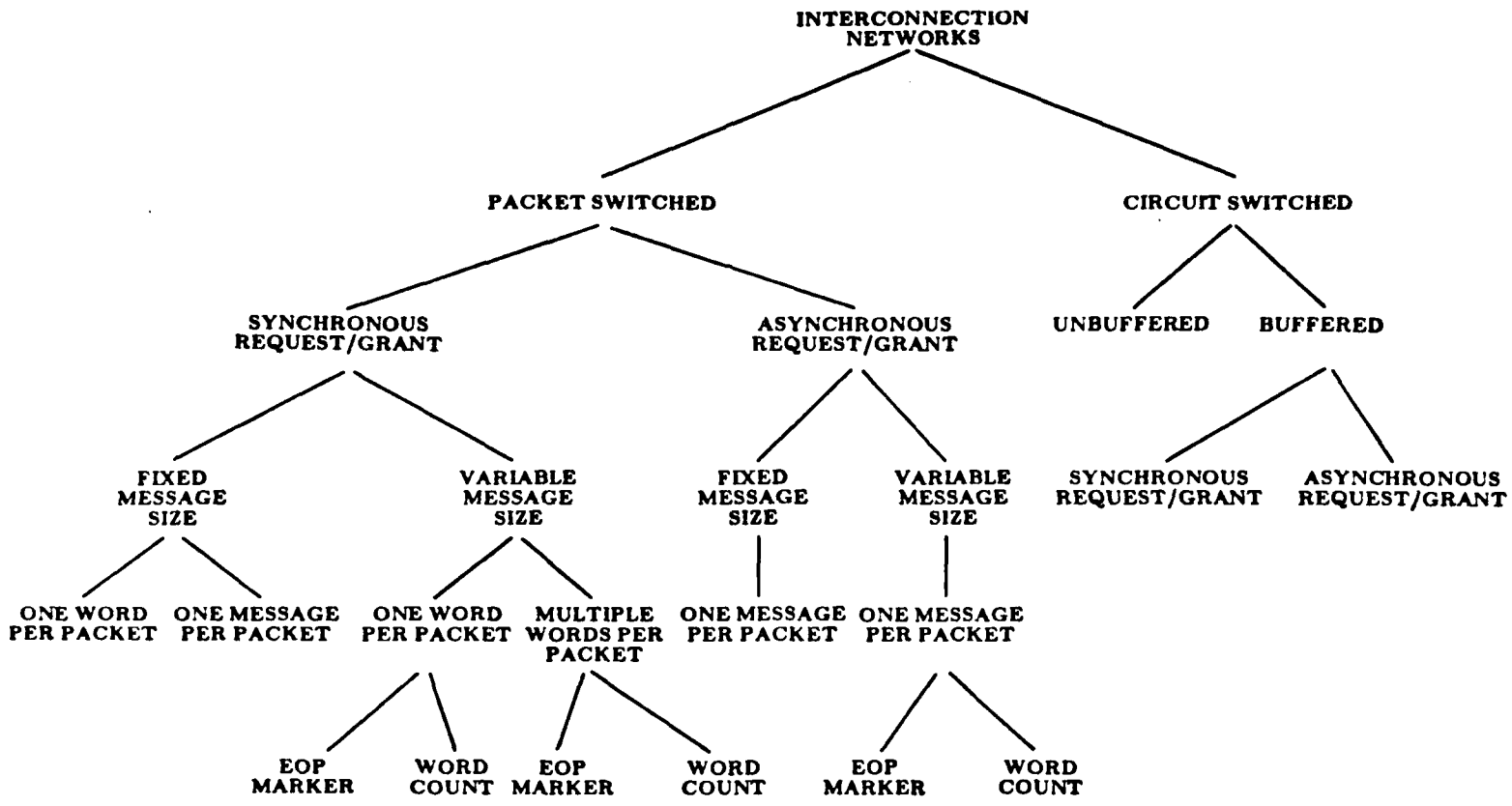


Figure 2.6 Taxonomy of Interconnection Network Protocols

(R/G) cycles, fixed versus variable message size, number of data items transferred per R/G cycle (packet size) and two methods for implementing a variable message size.

Throughout the remainder of this section, the use of the routing tags is assumed for networks with distributed control. In a packet switched network, the tags control the moving of packets from switch to switch. In a circuit switched network, the tags establish paths through the network. Since moving the tags through a circuit switched network (with distributed control) to establish paths is a special case of packet switching, the latter is discussed first. It is assumed that a packet switched network has distributed control.

In the following, a *message* is defined as a unit that is to be sent from one network user (device attached to a network input/output port) to another. It is composed of a routing tag and some number of data words. The data words are assumed to be the same width as the processors' data buses (i.e., the basic word width used by the processors). A *packet* is defined to be a unit that is transferred from one switching element to another in the network. Packets are delineated by control sequences that are performed by pairs of switches, in preparation for the transfer. If a message is larger than the packet size, it will be separated into a sequence of packets (possibly only the first of which contains the routing tag). This separation must either be performed by the network users (e.g., processors) or the network interface.

In a packet switched network, several functions the switching elements must perform can be identified. Consider a switching element labeled i. Let all other switching elements to which it can send a packet be in group $i+1$ and all those from which it can receive a packet be in group $i-1$. Switch i containing a packet must examine its routing tag and determine which switch in group $i+1$ is to receive the packet. (This assumes the packet contains a routing tag; when passing multi-packet messages , if there is only one copy of the routing information, it is stored in the switch.) Switch i must then *request* of the group $i+1$ switch permission to transfer the packet to it. If

switch i contains multiple packets (due to having multiple inputs), multiple requests may be sent to group i+1 switches from this switch. If two or more packets at switch i need to use the same output port (of the switch), only one of the packets can be processed at a time. If the packets in switch i use different output ports, multiple requests will be made to switches in group i+1. Thus in each switch, during the *request cycle*, routing tags are decoded, arbitration occurs if necessary, and requests are issued. Upon receiving requests from switches in i−1, switch i determines if it has the storage capacity to accommodate any of the packets. If so, appropriate *grant* signals are issued. This process occurs during the *grant cycle*. Finally, for those switches receiving grant signals, the *transfer cycle* effects the actual movement of data from one switching element to the next.

Given this scenario of the basic functions each switching element performs, the options in Figure 2.6 can be discussed. It is assumed that there is a single *network clock* connected to every switching element and that in one clock period or cycle, any one of request, grant, or transfer (of one data item no wider than the network data path width) may occur.

The first implementation option considered is whether or not to make the request/grant (R/G) cycles occur simultaneously in all switching elements. If so, the R/G cycles are called *synchronous* and, if not, they are called *asynchronous*. In the synchronous mode, all switches follow a fixed sequence of request, grant, transfer P words, request, grant, transfer P words, etc., where a number of network clock cycles are required to perform the transfer. Furthermore, in this mode the packet size P is fixed. The exact number of cycles required to move a packet is determined by the number of words in a packet, the word width relative to the network path width (e.g., a 16 bit word requires two cycles to traverse an 8 bit path), and some options to be discussed. If the total number of words to be transferred (e.g., the number of words in a message sent by a processor) is greater than P, one or more additional R/G cycles will

occur in each switching element through which the packet passes. In a system containing a synchronous network, the messages to be transferred must be segmented into packets of size P.

In the synchronous mode, when a device needs to send a message, it may have to wait a number of network clock cycles for the next request phase before requesting to enter the first packet. In the asynchronous mode, only one request and grant cycle is executed per message per switching element (i.e., the message size is equal to the packet size). On a given network clock cycle, any switching element may be executing a request, grant, or transfer cycle. The obvious advantage to the asynchronous mode is the smaller total number of network clock cycles required to transfer a packet from input to output. The advantage to the synchronous mode is that the number of connections between switching elements can be reduced and their control logic is less complex.

The next design choice concerns making the size of the message sent by the device attached to the network fixed or variable. Being able to choose one over the other is highly dependent on the expected communication transactions. A fixed message size is easier to implement than one which is variable. There is more overhead associated with a variable size message since information regarding the size must be included with the data. Two schemes for conveying the information are to include the exact count or to include an end of message marker. These schemes will be discussed later.

If the synchronous request/grant scheme is implemented, the packet size must be chosen. The packet size determines the minimum amount of storage required in each switching element. From a cost point of view, keeping storage requirements low is desirable. Transferring one word or a small number of words at a time will accomplish this. To minimize contention in the network, the storage capacity in each switch should be as large as possible. As soon as switch i is filled to capacity, for example, it will revoke the grant that corresponds to the switch that filled it in group i−1. If the

group i−1 switch was receiving packets bound for the currently blocked link it will begin storing them. When filled to capacity, it will in turn revoke any appropriate grants. If each switching element contains a minimum amount of storage, one conflict in a switch near the output can soon "tie-up" many links in the network. Thus overall throughput is worse than if the conflict had been contained to just one switching element. Simulation results reported in [DiJ81] for multistage networks verify that throughput increases significantly as the buffer size is increased (until a plateau is reached).

If the R/G cycle is synchronous but the message size is variable, an appropriate packet size must be chosen. The larger the packet, the better the throughput for lengthy messages, due to the lower ratio of R/G cycles to transfer cycles. On the other hand, there is the larger buffer requirement and worse fragmentation. Fragmentation results when the message size is not a multiple of the packet size. The last packet will contain unused data slots that are routed through the network.

In an asynchronous R/G cycle network, the packet size is the message size (regardless of whether the message size is fixed or variable). Thus, the asynchronous mode is the most efficient from a throughput point of view.

The last option represented in Figure 2.6 concerns how to implement variable message length. As mentioned earlier, this can be accomplished by including a word count or an end-of-packet (EOP) marker. The word count has an overhead of one additional word that immediately follows the routing tag.

In a circuit switched network, there are two approaches to establishing paths with routing tags. In the unbuffered case, the routing tag must be placed on the input data bus and held there until the path is complete. In the buffered case, establishing circuits is a special case of packet switching one word packets, i.e., routing tags only. Once a path is established, all buffers are bypassed to form a direct circuit. A variation on this form of circuit switching is called pipelining [SmS78]. In this case, data follows the tag

through the buffers. Any of these methods can be synchronous or asynchronous, but the difference in set-up time is negligible - two network clock cycles.

For a circuit switched network that is centrally controlled, all devices wishing service must submit their desired destination address to the controller during a request phase. For those requests granted, a request line must be held for the duration of the transmission.

The various options that have been discussed here can be incorporated into a network description notation. The notation should distinguish between circuit and packet switching, synchronous and asynchronous timing, fixed and variable packet size, and buffered and unbuffered circuit switching. Parameters should indicate packet size, amount of information transferred per cycle, and cycle time or delay for packet switching. Set-up time, switch node delay, and path width should be specified for circuit switching.

2.6 Functional Description

The discussion of notations and classification schemes so far has concentrated on the hardware structure of a computer system. In this section, operational characteristics are examined. Specifically, the kinds of operations that can be performed by the hardware and the data types that are supported directly. Bell and Newell's instruction - set processor (ISP) notation [BeN71] is described first. It is designed to complement their PMS notation that was described in Subsection 2.3.2. Then Giloi's taxonomy based on operational principles is explained. It is complementary to his hardware taxonomy that was presented in Subsection 2.3.1.

2.6.1 Bell and Newell's ISP Notation

ISP notation is designed so that any set of operations can be defined along with rules for interpreting a set of bits that represent a program. The program is a sequence

of operations. The set of operations can be divided into two groups. The first group consists of those needed to operate other system components: links, switches, memories, etc. (to use PMS terminology). The second group contains operations associated with D or data-operation components. These components actually transform information. Primitive forms of these components include add, subtract, multiply, divide, AND, OR, EXCLUSIVE-OR, etc. The D components are specialized according to the kind of data upon which they can operate, i.e., data-type. A data-type is defined by the referent of the bit pattern (e.g., that the bits refer to an integer in a given range) and a format (e.g., the most significant bit is the sign and the remaining bits are coefficients of sequentially decreasing powers of two in the binary representation of the integer). One processor may use several different data-types such as unsigned integer, signed integer, floating point and double precision floating point. Different operations are required for each data-type.

A processor is thus completely specified at the ISP level by its *instruction set* and its *interpreter.* These are defined in terms of *operations*, *data-types* and *memories.*

Each instruction in the instruction set is described by an *instruction-expression* of the form

condition → action-sequence.

The *condition* determines when the instruction is invoked. The *action-sequence* describes the transformations of data that takes place between memories (e.g., registers). The right arrow (→) represents the control action of a K unit or controller. The components of the action sequence eventually have the form

memory-expression ← data-expression.

The *memory-expression* describes which memory location is affected. The left arrow (←) corresponds to the transmission operation of a link and amounts to an assignment operator. The *data-expression* describes the information that is to be transmitted to the specified memory location. Data expressions generally are written in terms of

standard mathematical notation.

Action sequences can be concurrent or sequential. If the components of the action sequence are separated by semicolons only, they occur simultaneously. For example, in the sequence

$$Y_1 \leftarrow X_1; Y_2 \leftarrow X_2$$

all Xs are assumed to have defined values prior to execution of the action-sequence and upon execution, the Xs are transferred to the Y memories simultaneously. If the components of the action-sequence are separated by the term "next," they are sequential. For example, the sequence

$$Y \leftarrow Z; \text{ next } X \leftarrow Y$$

where X, Y, and Z are registers, causes the contents of Z to be copied into X and Y (this sequence is needed if there is no direct connection from Z to X).

Memory in the system (including registers) is given mnemonic names followed by the number of words in square brackets and the number of bits in angle brackets. The words and bits are specified by address and number, respectively, of the form a:b. The first number is "a" and the last is "b." For example, a 64K main memory with 16 bits is represented by

$$Mp[0:FFFF_{16}]<0:15>$$

where base 10 is the default. Where there is only one word of memory, as with a register, the square brackets and included information are omitted. For example, a 16 bit accumulator can be represented by

$$ACC<0:15>.$$

The bits can be named and enumerated, separated by commas, as with a status register:

STAT<E,F,H,I,N,Z,V,C>.

These are the status flags used in the Motorola 6809 microprocessor (entire state on stack, fast interrupt, half carry, irq interrupt mask, negative, zero, overflow, and carry-borrow). Bits can also be concatenated using the "□" operator. For example, the carry bit in the status register might be appended to the accumulator to form a new register for use in an arithmetic action-sequence:

CAC<C,0:15> := C □ACC.

The ":=" is used to define a new entity.

If a memory is multidimensional, several start/end address pairs can be used. For example, the Digital Equipment PDP-8 memory can be described as consisting of eight memory fields of 32 pages each, with 128 words per page and 12 bits per word. This is represented by

Mp[0:7][0:31][0:127]<0:11>.

Finally, a set of bits can have several names. A good example of this is to define fields within a register, such as an instruction register. In the PDP-8 the instruction format is [BeN71]:

Op<0:2> := instruction <0:2>

indirect_bit/ib := instruction <3>

page_0_bit/p := instruction <4>

page_address<0:6> := instruction <5:11>.

The "/" in the above is used to indicate equivalent symbols and is read "or."

With the basic notation and the form of expressions defined, some examples of instruction interpretation can be given. The following is a definition of the two's

complement add operation:

two's complement add/tad → (C□ACC → C□ACC + Mp[Z])

tad := (op = 1).

An abbreviation for the operation is defined along with the action that is to occur. Then, the opcode associated with the operation is defined. By defining all registers and action-sequences for each opcode, the functional characteristics for a given computer can be completely specified. An example from [BeN71] of the complete specification of the PDP-8 is shown in Figure 2.7, which illustrates how powerful the notation is. The formal specification for ISP can be found in the appendix of [BeN71].

ISP notation appears to be flexible enough to describe the function of any of the computers that have been discussed in this chapter. In the case of a computer like the Cray-1, since concurrent events can be described, pipelined operations can be defined. Also, vector registers are simply represented as multiword memories. For multiple computer systems, a description is given in ISP for each computer type.

2.6.2 Giloi's Taxonomy

In [Gil81], Giloi points out that most architectural classification schemes (e.g., those discussed in sections 2.2 through 2.5) are concerned solely with ***structural features***. To remedy this situation, he has developed a scheme that takes into account (1) how information is represented in the machine; (2) information access mechanisms; (3) control structures; and (4) communication structures. These features are all based on *operational principles* of the architecture.

A computer cannot be described by operational principles alone, so it is assumed that the taxonomy to be described is taken together with the structural features discussed in Subsection 2.3.1. Giloi's taxonomy is intended to be abstract and thus implementation independent. Therein lies the difference between his scheme and Bell and Newell's ISP notation.

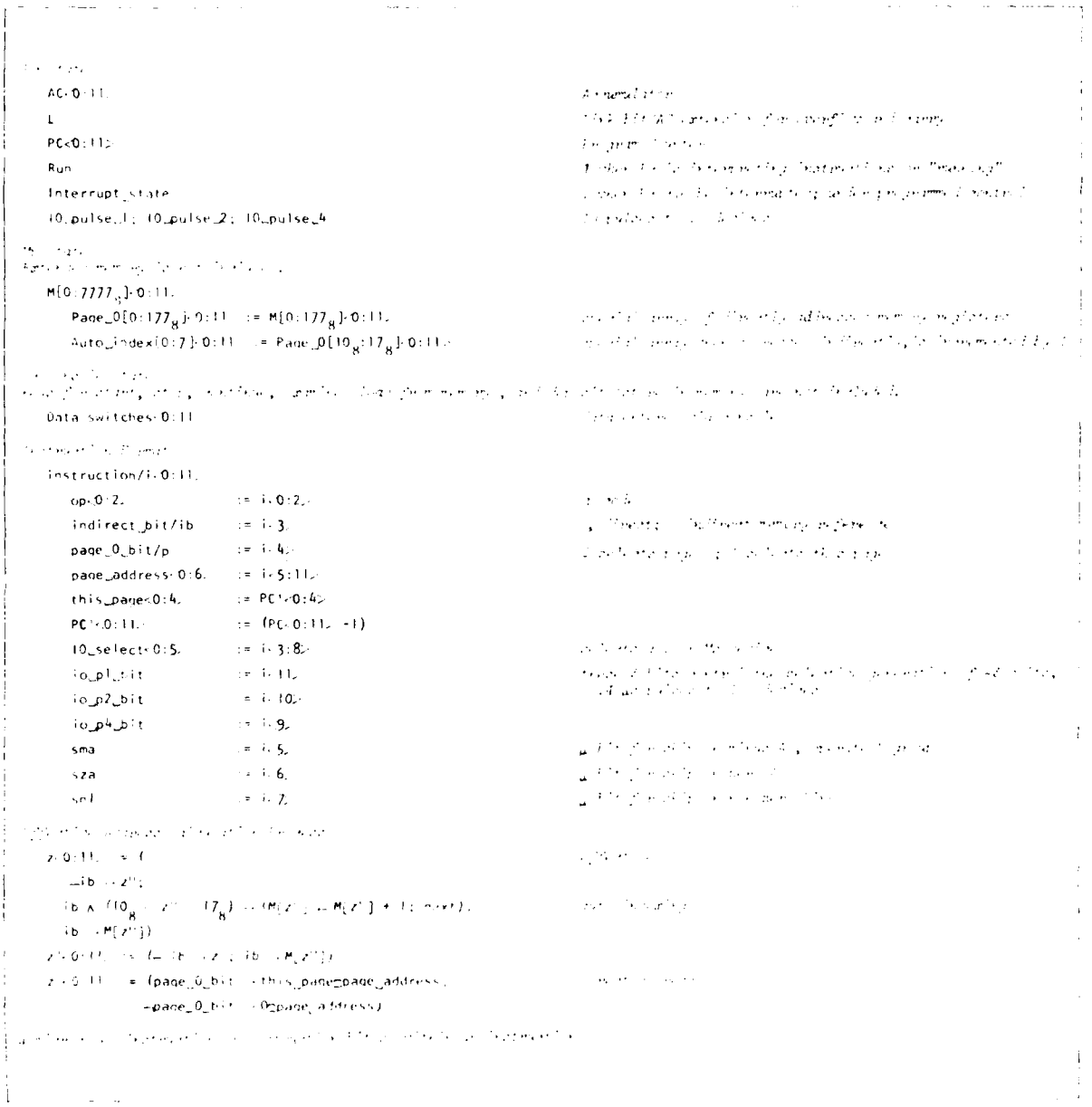


Figure 2.7 ISP Notation Representation of Digital Equipment Corporation PDP-8 (from [BeN71]) (continued on page 2-46)

```
Instruction Interpretation Process
  Run ∧ ¬ (Interrupt_request ∧ Interrupt_state) → (          no interrupt interpreter
     instruction ← M[PC]; PC ← PC + 1; next                  fetch
     instruction_execution);                                 execute
  Run ∧ Interrupt_request ∧ Interrupt_state → (              interrupt interpreter
     M[0] ← PC; Interrupt_state ← 0; PC ← 1)

Instruction Set and Instruction Execution Process
  Instruction_execution := (
     and (:= op = 0) → (AC ← AC ∧ M[z]);                     logical and
     tad (:= op = 1) → (L□AC ← L□AC + M[z]),                 two's complement add
     isz (:= op = 2) → (M[z'] ← M[z] + 1; next               index and skip if zero
                        (M[z'] = 0] → (PC ← PC + 1));
     dca (:= op = 3) → (M[z] ← AC; AC ← 0);                  deposit and clear AC
     jms (:= op = 4) → (M[z] ← PC; next PC ← z + 1);         jump to subroutine
     jmp (:= op = 5) → (PC ← z);                             jump
     iot (:= op = 6) → (                                     " in out transfer, microprogrammed to generate up to 3 pulses
        io_p1_bit → IO_pulse_1 ← 1; next                      to an IO device addressed by IO_select
        io_p2_bit → IO_pulse_2 ← 1; next
        io_p4_bit → IO_pulse_4 ← 1);
     opr (:= op = 7) → Operate_execution                     the operate instruction is defined below
                 )                                           end instruction execution

Operate Instruction Set
The microprogrammed operate instructions: operate group 1, operate group 2, and extended arithmetic are defined as a separate
 instruction set.
  Operate_execution := (
     cla (:= i.4_ = 1) → (AC ← 0),                           clear AC; common to all operate instructions.
     opr_1 (:= i.3_ = 0) → (                                 operate group 1
        cll (:= i.5_ = 1) → (L ← 0), next                    " clear link
        cma (:= i.6_ = 1) → (AC ← ¬ AC),                     " complement AC
        cml (:= i.7_ = 1) → (L ← ¬ L), next                  " complement L
        iac (:= i.11_ = 1) → (L□AC ← L□AC + 1), next         " increment AC
        ral (:= i.8_10_ = 2) → (L□AC ← L□AC * 2^rotate),     " rotate left
        rtl (:= i.8_10_ = 3) → (L□AC ← L□AC * 2^2^rotate),   " rotate twice left
        rar (:= i.8_10_ = 4) → (L□AC ← L□AC / 2^rotate),     " rotate right
        rtr (:= i.8_10_ = 5) → (L□AC ← L□AC / 2^2^rotate)),  " rotate twice right
     opr_2 (:= i.3_11_ = 10) → (                             operate group 2
        skip condition ≠ 0 (i.8_ = 1) → (PC ← PC + 1); next  " skip test
           skip condition = ((sma ∧ (AC < 0)) ∨ (sza ∧ (AC = 0)) ∨ (snl ∧ L))
        osr (i.9_ = 1) → (AC ← AC ∨ data switches);          " or in switches
        hlt (i.10_ = 1) → (Run ← 0)),                        " halt program
     EAE (:= i.3_11_ = 11) → EAE_instruction_execution)      go to EAE instruction set
```

Figure 2.7 (continued)

At the highest level, Giloi's taxonomy can be characterized as follows. The *operational principle* of a computer architecture defines its functional behavior in terms of an information structure and a control structure. The *information structure* consists of a set of abstract data types which specify the type and structure of information in the machine, its machine representation, and operations the machine can perform on it. The *control structure* is defined by control algorithms which interpret and transform information in the machine. This view of the function of a computer is very similar to that taken by Bell and Newell as described in the previous subsection.

A *machine data type* is defined by the triple (O,F,R), where O is a set of machine data objects of some type, F is a set of machine operations applicable to the objects in O, and R is a set of representations of the objects in O. Three major classes of object types that are distinguished are elementary types, set types, and structure types. Elementary data types can be characterized by such objects as instructions, descriptors, capabilities, reals, integers, characters, and semaphores. Classes of operations or functions that can be performed on elementary data types include (1) binding (i.e., load a data object with new information); (2) access (i.e., provide access to the contents of a data object); (3) decoding (i.e., interpret a data object); (4) value production (i.e., produce a value to become a data object); (5) test and set (i.e., test and/or change a semaphore); and (6) conversion (i.e., change an object type and/or representation). In a conventional computer, binding corresponds to assigning a value to a variable, access corresponds to evaluating an address, decoding corresponds to initiating control sequences, value production results from the usual arithmetic operations (add, subtract, etc.), and conversion is a change in format (e.g., from integer to real).

Corresponding to major object types, there are three major classes of object representations: (1) elementary; (2) set type; and (3) data structure. For *elementary data objects*, existing architectures utilize three kinds of *scalar representations*. The first is *generic* in which the scalar machine data object is a bit vector that represents

the value of the data object. The machine determines from the kind of function applied to the representation how to interpret it. The second is a ***self descriptive representation*** in which some bits of a bit vector form a ***tag field*** which denotes the object type. The remaining bits form a value representation of the data object. The third type of scalar representation is ***self identifying***. Some bits of a bit vector form a ***key field*** which denotes a class of which the object is an element. The remaining bits form a value representation.

The objects of a ***set type*** are linearly ordered sets of scalar data objects. All objects in the set have at least one common attribute (e.g., element type or access control attributes). If all elements in a set have the same type, it is said to be ***homogeneous***.

The objects of a ***structured machine data type*** are presented by the four-tuple

(<object identifier>, <structure specification>, <data set>, <attributes>).

The <structure specification> and <attributes> are represented by an object descriptor at the hardware level. The <data set> is represented by a set type object. Data items of a structured object are not individually named and cannot be referenced directly. Rather, they are accessed through the use of access functions. Examples of structured machine data types include a stack, tree (as used in reduction machines), and vector.

A complete and very detailed taxonomy based on these concepts is presented in [Gil81]. Its length is too great to be included here.

2.7 Conclusions

This survey of architectural classification schemes and description notations has shown that much work has been done, but at significantly different levels of detail. It is apparent that none of the schemes discussed is comprehensive. Hockney and Jesshope's notation has the most breadth of any single scheme, yet it does not address

interconnection networks in enough detail nor does it provide a functional specification for the system described. The combination of Bell and Newell's PMS and ISP notations comes closest to completely describing a computer system. However, as pointed out in Section 2.3, PMS notation is not as well suited to representing architectural features that determine an algorithm's performance as HOJ notation. The combination of Giloi's structural and functional taxonomies is very broad, but too abstract for the purpose at hand.

Taken as a whole, the elements of a comprehensive classification scheme/descriptive notation are embodied in the schemes presented in this chapter. Thus, the following approach to constructing a comprehensive scheme based on many of the results described here is proposed. It will be referred to as the CHACAD scheme for Comprehensive, Hierarchical Architectural Classification And Description scheme. As the name implies, it is hierarchical, and four levels are defined. Levels I and II are classification oriented and levels III and IV are description oriented. Level I is the most coarse and consists of four categories, based on Kuck's classification scheme (cf. Subsection 2.2.4). They are (1) single instruction stream, single execution stream (SISE); (2) single instruction stream, multiple execution stream (SIME); (3) multiple instruction stream, single execution stream (MISE); and (4) multiple instruction stream, multiple execution stream (MIME).

Level II has sixteen categories and is Kuck's complete scheme. These categories are derived from those at Level I by distinguishing between scalar and array instruction and execution types. Level III is an expanded version of Hockney and Jesshope's HOJ notation. It was illustrated in Subsection 2.3.3 that there is a close relationship between Kuck's categories and HOJ representations of systems in those categories. Thus, the transition from Level II to Level III is smooth. In Section 2.4 some additional notation was recommended for representing pipelined structures in more detail which should be included.

Level IV is the most detailed representation of a computer system. It is this level upon which future research should concentrate. Notation needs to be developed, compatible with HOJ notation, that describes the salient features of the system interconnection network and of system wide functionality. The structural taxonomy, parameters, and protocol taxonomy described in Section 2.5 should form the basis of the interconnection network notation. The notation should consist of two parts: (1) structural, describing how switching elements are connected; and (2) functional, describing how each type of switching element functions.

Bell and Newell's ISP notation (cf. Subsection 2.6.1) is recommended as a basis for the functional notation. However, it needs to be expanded to include, for example, parallel data types (e.g., a skewed array) that are manipulated by multiple processors.

Systems that have been proposed but not implemented may have undefined or only partially defined features (e.g., the instruction set). Thus the CHACAD scheme needs to be able to describe a system at the level available. At Levels I and II, parameters need to be identified that quantify system characteristics that can be determined at the respective levels. Providing these parameters will facilitate the evaluation of an algorithm's compatability with the system, albeit with a lower degree of confidence. In Chapter 5, some features or parameters that correspond to different levels of description will be discussed.

CHAPTER 3
MODELING ARCHITECTURES:
MULTISTAGE INTERCONNECTION NETWORKS

3.1 Introduction

Many different approaches to providing a communication capability in parallel processing systems have been proposed. These include the use of busses [Wid76], hierarchies of busses [SwF77], direct links [DeP78], single stage networks [Sto71], multistage networks [Bat76, Fen74, GoL73, Law75, Pea77, SiM81a, SiM81b, SiS78], and crossbars [WuB72]. These approaches have been surveyed [AnJ75, Fen81, SiM79a, Thu74] and were discussed in Chapter 2. In this chapter, multistage networks are examined in detail. In Section 3.2, seventeen networks that have been proposed and/or built are presented and discussed in the order in which they appeared in the literature. The networks are then placed into a family tree based on their structural relationship to one another. Two major classes of networks are identified. In Section 3.3, implementations for switching nodes or switching elements are surveyed and compared. Methods for distributing the control of multistage interconnection networks are discussed in Section 3.4. Finally, in Section 3.5, fault tolerant designs for these networks are examined.

This chapter was also supported in part by another research grant.

3.2 History of Multistage Networks

3.2.1 Introduction

A considerable amount of research has been done on multistage interconnection networks in recent years. The earliest efforts were in the context of telephone switching [Ben64,Clo53,Joe68,Wak68]. It was then realized that some of that work might be applicable, with suitable modifications, to computer communication [OpT71a,OpT71b]. Also, around that time, special purpose networks for number sorting called *bitonic sorters* were investigated [Bat68]. With experimentation into parallel processing or multiple computer systems such as Illiac IV [BaB68] and C.mmp [WuB72], interest in designing interconnection networks tailored to that application began to grow. Early work published in that vein was done by Lipovski on the *SW-structure* [Lip70]. That work was later refined and generalized by Goke and Lipovski who introduced a class of networks called *Banyans* [GoL73]. In [GoL73] it is pointed out that SW-banyans (S=F=2) (formerly SW-structures) are equivalent in topology to Batcher's bitonic sorter. In 1971, Stone published an influential paper on the *perfect shuffle* network [Sto71]. Though presented as a single stage network, it was later extended by Lang and Stone into a multistage version [LaS76]. Feng published work in 1974 on implementing networks for data manipulation [Fen74]. One such network has since come to be known as the *data manipulator*. At that same time Batcher was doing work on the *Flip* network used in STARAN [Bat74], but the details were not published until 1976 [Bat76]. Soon to follow was work by Lawrie on the *Omega* network [Law75] and by Pease on the *indirect binary n-cube* network [Pea77].

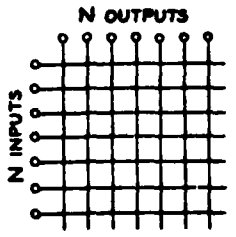
In April of 1978 Siegel and Smith published a paper comparing the data manipulator, Flip, Omega, and indirect binary n-cube networks [SiS78]. As a benchmark, they introduced the *Generalized Cube* network and showed that the Flip, Omega, and indirect binary n-cube networks were all topologically equivalent to it. They also relaxed some restrictions on the implementation of the data manipulator, calling their

version the *Augmented Data Manipulator (ADM)*, and proved that its capabilities were a superset of those of the Generalized Cube (and therefore all networks equivalent to the Generalized Cube). At nearly the same time, in August of 1978, Wu and Feng also published a comparison paper. They introduced the *baseline* network as a benchmark and showed that the Flip, Omega, indirect binary n-cube, SW-banyan with spread and fanout of two (a member of the Banyan class), and the reverse (or inverse) baseline were all topologically equivalent. At the same conference where this work was presented, the *HEP* system and network were introduced [Smi78].

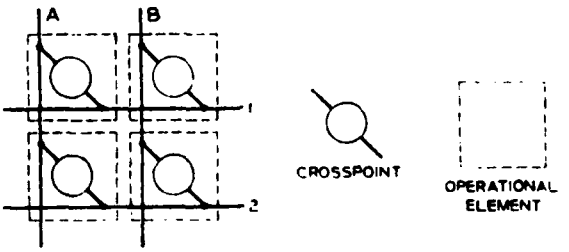
In the very recent past, the class of *Delta* networks has been introduced by Patel [Pat79], the reverse-exchange network has been investigated by Wu and Feng [WuF79a], and properties of the *inverse ADM (IADM)* network have been presented by McMillen and Siegel [McS80a]. In September 1980, Pradhan and Kodandapani published another comparison of multistage networks (also including single stage networks) [PrK80b]. They defined an equivalence relation and showed that the Flip, Omega, indirect binary n-cube, SW-banyan, and all of their inverses were equivalent under the defined relationship. The most recent introduction of a "new" multistage network is called the *Gamma* network, presented by Parker and Raghavendra [PaR82]. It will be shown later, however, that it is topologically equivalent to the IADM network.

3.2.2 Clos Networks

The early work on multistage interconnection networks was aimed at providing economical telephone switching capability. The most important constraint imposed on the network design is that any idle pair of input and output ports can be connected regardless of the existing connections. This is called the *non-blocking* property. The most obvious means for meeting this requirement is to build an NxN crossbar, shown in Figure 3.1(a). A connection is made from an input to an output by closing the crosspoint switch, illustrated in Figure 3.1(b), where the two busses intersect. The



(a)



(b)

Figure 3.1 (a) NxN Crossbar [Clo53]
(b) Crosspoint Switching Element [Joe68]

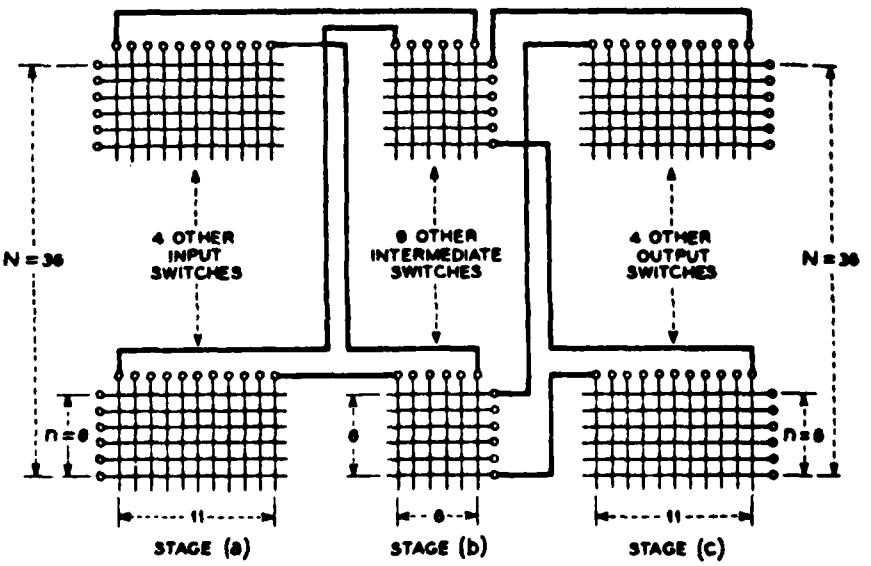


Figure 3.2 36x36 Three Stage Clos Network [Clo53]

major drawback to this scheme is that N^2 crosspoints are required. In 1953, Clos investigated a class of multistage networks with the non-blocking property but a lower cost [Clo53]. An example of a 36x36 three stage network is shown in Figure 3.2. The first and last stages have six 6x11 crossbars and the middle stage has eleven 6x6 crossbars. The number of crosspoints required is $6N^{3/2} - 3N = 1188$ for N=36. This compares with 1296 for a 36x36 crossbar. For values of N less than 36 the crossbar is cheaper. The larger N grows beyond 36, the greater the difference becomes. For N=1000, the difference is 1,000,000 versus 186,737.

The general three stage Clos network is shown in Figure 3.3. The first and last stages have r nxm crossbars and the middle stage has m rxr crossbars, where $n=N^{1/2}$. There are r·n inputs and outputs. A three stage Clos network is completely characterized by m, n, and r. Clos was able to show that for $m \geq 2n-1$, the network is strictly non-blocking [Clo53].

3.2.3 The Benes Network

A network is called *rearrangeable* if any idle pair of input and output ports can be connected after possibly rearranging some of the existing connections. Benes investigated a special case of the Clos networks in which only 2x2 crossbars were used [Ben65]. N is required to be a power of two. The Benes network is constructed recursively as follows. First construct a three stage Clos network with n=m=2 and r=N/2. Then, for each of the N/2 x N/2 middle stage crossbars repeat the procedure. This process continues until there are N/2 middle stage 2x2 crossbars. A size N=8 Benes network is shown in Figure 3.4. In general, an NxN Benes network has $(2\log_2 N)-1$ stages of N/2 2x2 crossbars or switching elements. This is a rearrangeable network. Since a 2x2 crossbar contains four crosspoints (see Figure 3.1(b)), a Benes network of size N requires $4N\log_2 N - 2N$ crosspoints. For N=1024, this is 38,912 versus 1,048,576 for the crossbar or 193,536 for a three stage Clos network, quite an improvement. The

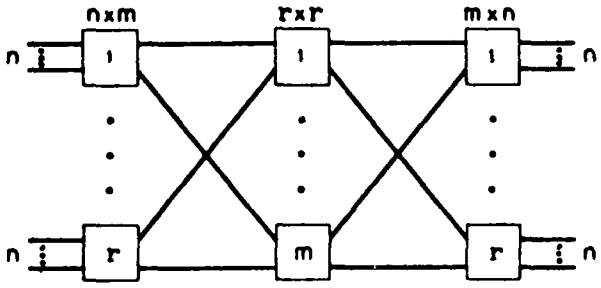


Figure 3.3 General Clos Network [Ben65]

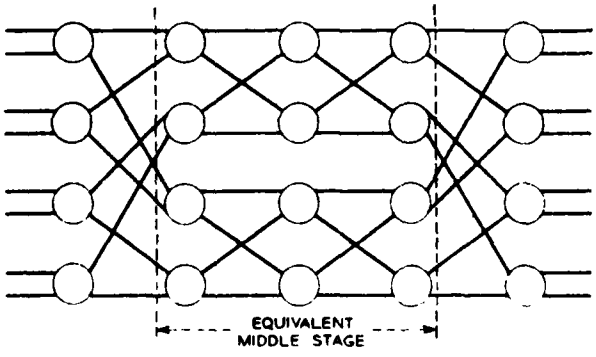


Figure 3.4 8x8 Benes Network [Ben65]

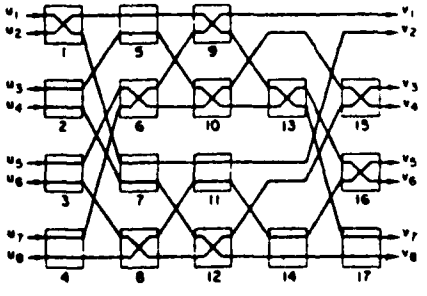


Figure 3.5 Benes Network as Modified by Waksman [Wak68]

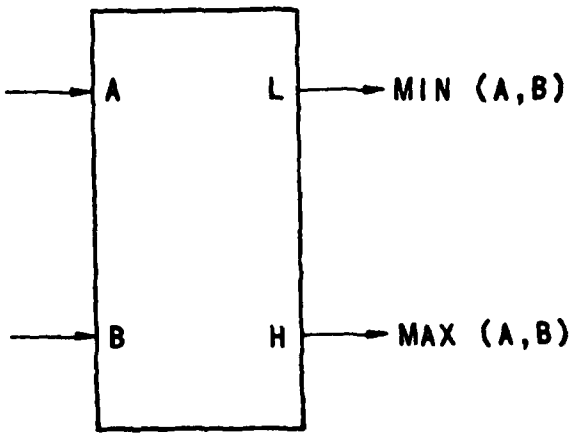
Benes network was refined slightly by Waksman who showed that a few of the switching elements could always be set to one state and therefore be removed [Wak68]. This is shown in Figure 3.5, where one possible connection of all inputs to all outputs is illustrated.

3.2.4 The Bitonic Sorter

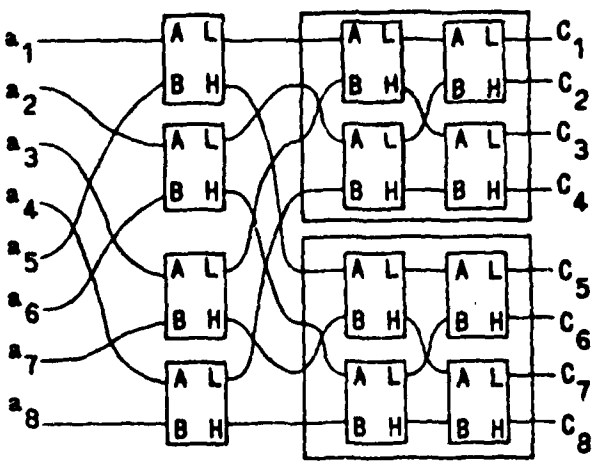
The bitonic sorter is a network designed by Batcher for efficiently sorting a bitonic sequence of numbers into a monotonic sequence [Bat68]. A bitonic sequence is the juxtaposition of two monotonic sequences (i.e. in non-decreasing or non-increasing order), one ascending, the other descending. Thus the bitonic sorter can be used to merge two monotonic sequences (which can always be combined to form a bitonic sequence) into one. This combined with other hardware can sort an arbitrary list of numbers. A single comparator element is shown in Figure 3.6(a) and an eight input bitonic sorter is shown in Figure 3.6(b). Notice how similar this network is to the first three stages of the Benes network (see Figure 3.4) if the A_2-A_6 comparator is swapped with the A_3-A_7 comparator (they are topologically identical). Batcher points out that one application of this network is interconnecting multiple computers.

3.2.5 Banyan Networks

Banyan networks are defined in terms of their graphical representation [GoL73]. A banyan is a directed graph composed of vertices and edges or arcs such that it is irreflexive, asymmetric, and intransitive (a Hasse diagram of partial ordering [Ber62]). A *base* in the graph is any vertex having no arcs incident into it and an *apex* is any vertex having no arcs incident out from it. The graph has the property that there is exactly one path from any base to any apex (and vice versa). A *regular banyan* is one in which the number of arcs incident into each vertex (fanout or F) and the number incident out from each vertex (spread or S) are constants. A *rectangular banyan* is a



(a)



(b)

Figure 3.6 (a) Two Input Sorting Element
(b) Eight Input Bitonic Sorter
[Bat68]

regular banyan in which the spread and fanout are equal. The TRAC prototype [ScU80] contains a regular banyan with spread = 2 and fanout = 3 as shown in Figure 3.7 (arcs shown undirected). In the figure there are four apexes and nine bases. Processors are connected to apexes and memory and I/O to bases. A structure that is equivalent to the bitonic sorter (Figure 3.6(b)) is the rectangular SW-banyan with S=F=2 shown in Figure 3.8 (with undirected arcs). To see the equivalence, replace all the sub-structures in Figure 3.8 that look like the banyan in Figure 3.9(a) with the switching node shown in Figure 3.9(b). This relationship is discussed in detail in [McS82c].

The SW-banyan can support the formation of data trees, shown in Figure 3.10(a) and instruction trees, shown in Figure 3.10(b). The data trees allow one processor to have access to more memory than that contained in just one memory module. The instruction tree allows several processors that have been linked together to form a larger processor and/or several processors working on vectors in SIMD mode, to receive the same instruction during their fetch cycle. The instruction tree structure is only active in the network during that cycle. The combined configuration in Figure 3.10 results in a two processor SIMD machine where each processor has two memory modules and one I/O port.

Another type of regular banyan is the CC-banyan, shown in Figure 3.11 for N=8. If apexes and bases are labeled from 0 to N−1, right to left and levels are labeled from n−1 to 0 from top to bottom, then vertex j, $0 \leq j < N$, at level i, $0 \leq i < n$, is connected to vertices j and $(j+2^i)$ *mod* N at level i−1.

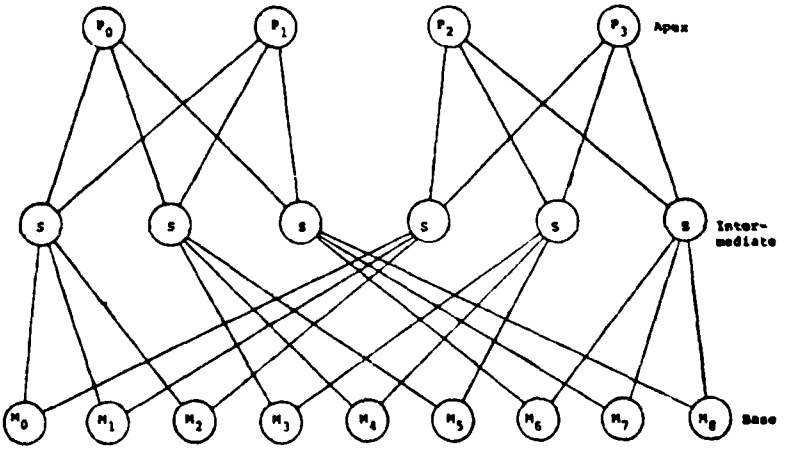


Figure 3.7 4x9 Regular Banyan with S=2, F=3 [JeB82]

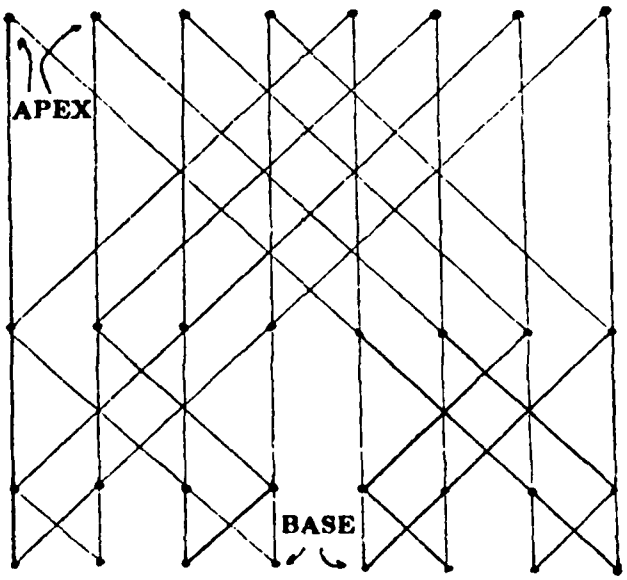
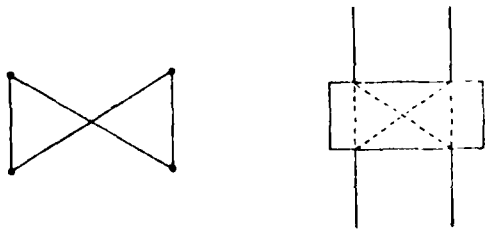
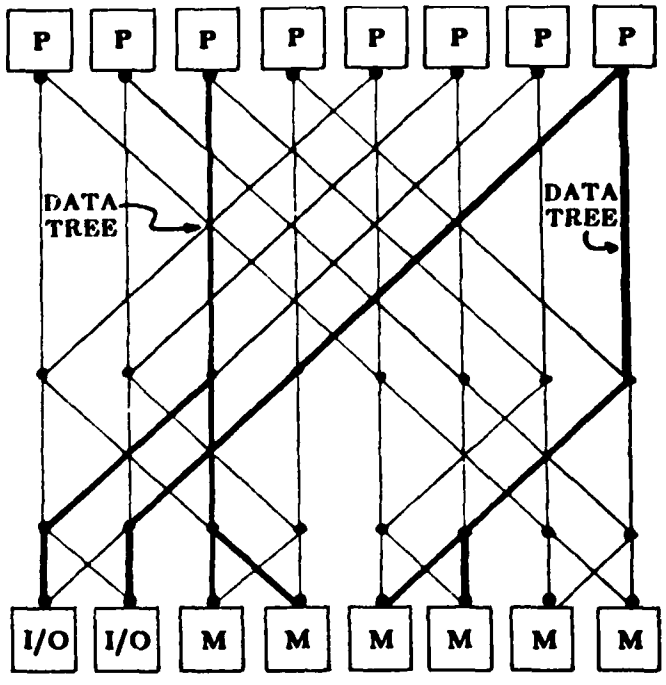


Figure 3.8 8x8 Rectangular SW-Banyan with S=F=2 [LiT77]

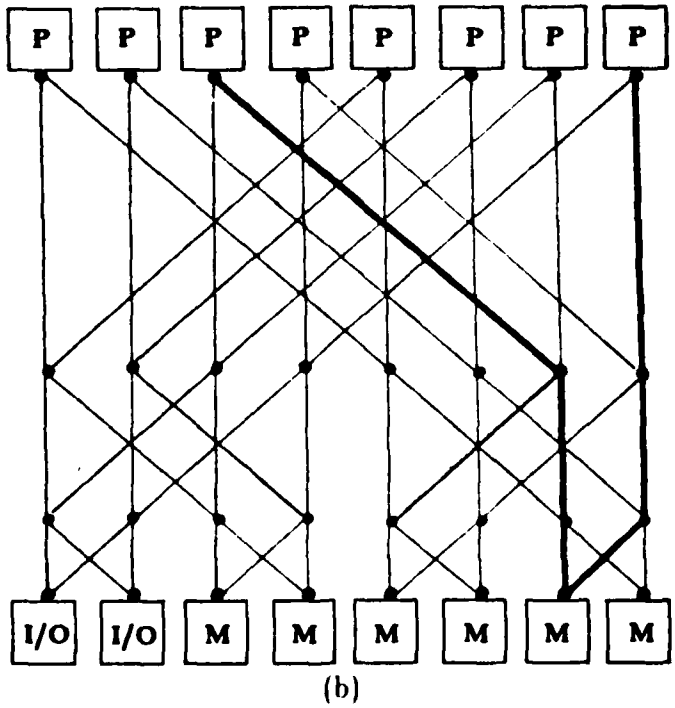


(a) (b)

Figure 3.9 (a) 2x2 SW-Banyan Network
(b) 2x2 Switching Element
[JeB82]



(a)



(b)

Figure 3.10 (a) Data Trees in the SW-Banyan (S=F=2)
(b) An Instruction Tree in the SW-Banyan (S=F=2)

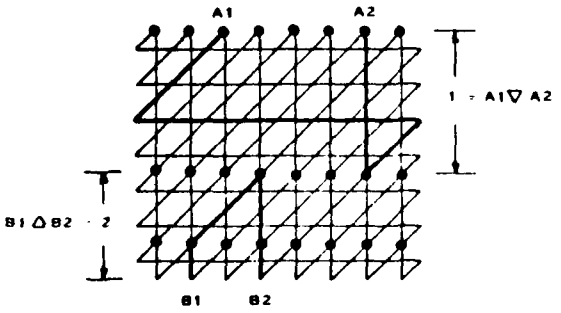


Figure 3.11 8x8 CC-Banyan Network [GoL73]

3.2.6 The Data Manipulator Network

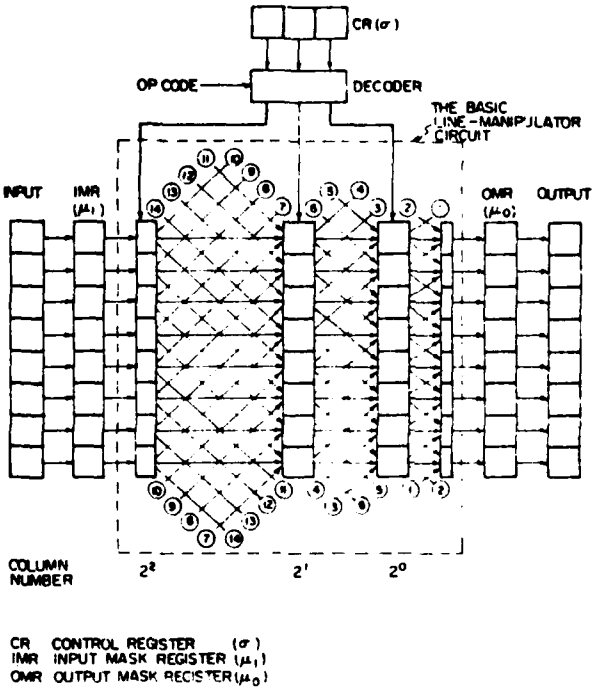
Feng's data manipulator is shown in Figure 3.12(a) with eight inputs and outputs. The network's structure is such that if there are $N=2^n$ I/O ports, then there are n stages or columns labeled $2^{n-1}, \ldots, 2^1, 2^0$, and an unlabeled output column. A cell (Figure 3.12(b)) at level k, $0 \leq k < N$, in column 2^i is connected to cells $k-2^i \bmod N$, k, and $k+2^i \bmod N$ in column 2^{i-1}. If the -2^i connections were removed from this network, it would be structurally equivalent to the CC-banyan shown in Figure 3.11.

Each column of the data manipulator receives three pairs of control signals and each cell is connected to one signal from each pair: $U_1^{2^i}$, $U_2^{2^i}$, $H_1^{2^i}$, $H_2^{2^i}$, $D_1^{2^i}$, $D_2^{2^i}$. U^{2^i} enables "Up" or -2^i links in stage 2^i, H^{2^i} enables "Horizontal" or straight links, and D^{2^i} enables "Down" or $+2^i$ links. Those cells whose i^{th} bit of their level, k, is a 0 are connected to control lines with subscript 1 and those whose i^{th} bit is a 1 are connected to control lines with subscript 2. Note that $U_1^{2^{n-1}}$ and $U_2^{2^{n-1}}$ are functionally identical to $D_1^{2^{n-1}}$ and $D_2^{2^{n-1}}$.

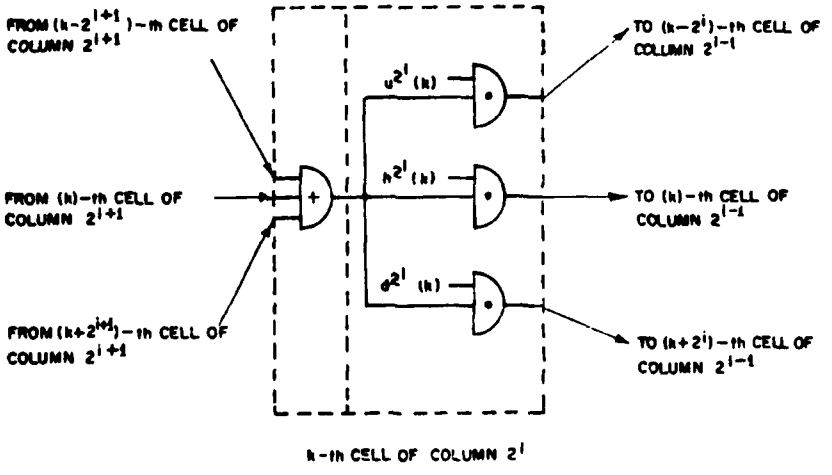
In [Fen74] it is shown that the network with this control scheme is able to perform the data manipulating functions of permuting, replicating, and spacing. Permuting is a rearrangement of the data at the input such that all items appear at the output in a new order and no two items go to the same output. Useful permutations include the shift, flip, shuffle, transpose, merge, mix, and bit reverse functions [Fen74]. Replicating is the copying of a group of elements. Spacing is any operation that moves the data without reordering it. For example, spreading and compressing.

3.2.7 The Flip Network

An 8x8 Flip network is shown in Figure 3.13(a) [Bat76]. A size N network has n stages labeled from input to output from 0 to n−1. At stage i, the inputs to that stage which differ in their i^{th} bit can switch positions. The network is centrally controlled and has two kinds of signals, one for flip permutations and one for shift permutations.

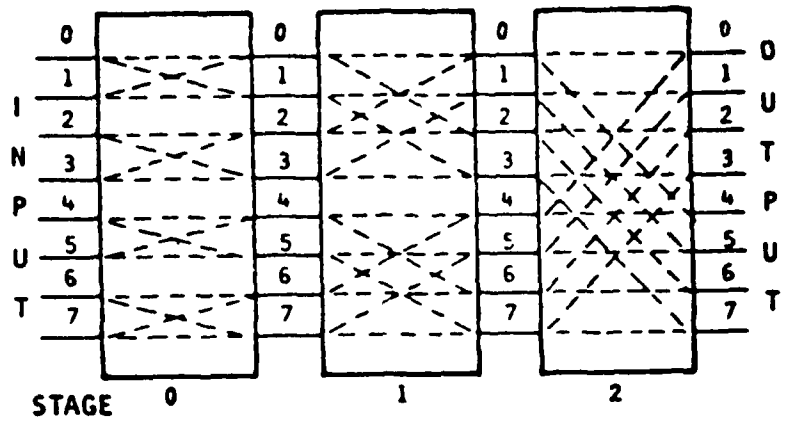


(a)

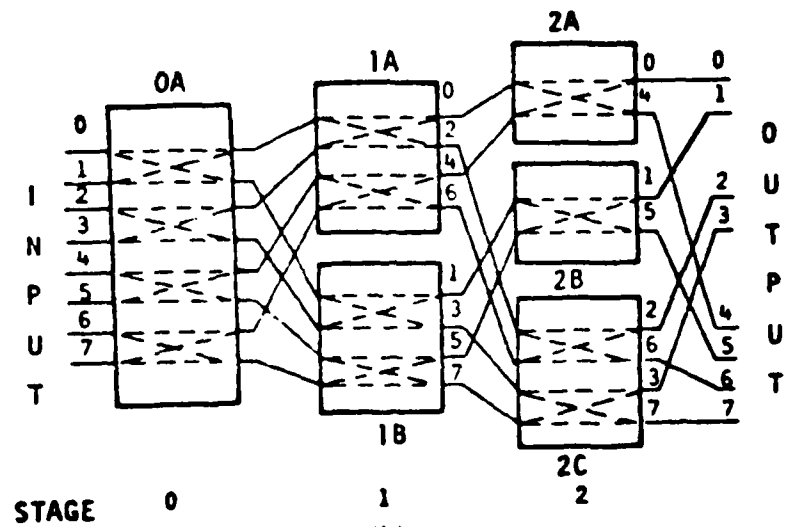


(b)

Figure 3.12 (a) 8x8 Data Manipulator Network
(b) Switching Element [Fen74]



(a)



(b)

Figure 3.13 (a) 8x8 Flip Network
(b) Flip Network Redrawn with Shift Control

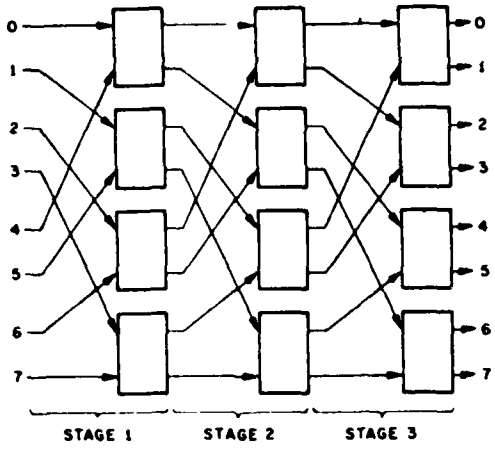
Let $F = f_{n-1} \cdots f_1 f_0$ be the flip control vector such that stage i gets f_i, $0 \leq i < n$. If $S = s_{n-1} \cdots s_1 s_0$ is an input address, then the flip network moves data at that address to output address $S \oplus F = s_{n-1} \oplus f_{n-1}, \ldots, s_1 \oplus f_1, s_0 \oplus f_0$. Since every value of F corresponds to a unique permutation, there are 2^n flip permutations the network can perform. When $F = 1 \cdots 11$, the data is flipped end for end (mirror permutation).

The shift control requires i+1 signals at stage i, $0 \leq i < n$, and is illustrated in Figure 3.13(b). The shift control allows data at input S, $0 \leq S < N$, to move to output $S+2^m \bmod 2^p$, $0 \leq m < p \leq n$. A shift of $2^m \bmod 2^p$ divides the 2^n data items into groups of 2^p items each and shifts each group down end-around 2^m places. For example, to obtain a shift of +1 mod 8, in Figure 3.13(b) the control signals should be 0A=1, 1A=1, 1B=0, 2A=1, 2B=0, and 2C=0. There are $(n^2+n+2)/2$ different shift permutations (including the identity).

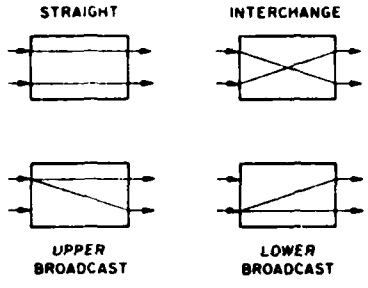
Notice that if the flip network in Figure 3.13(a) is rotated counterclockwise 90° it is structurally identical to the SW-banyan (S=F=2) in Figure 3.8. This is true in general, thus it is possible to implement the capabilities described for each network in either one.

3.2.8 The Omega Network

Lawrie's Omega network of size N=8 is shown in Figure 3.14(a) [Law75]. It has n stages, in this case numbered from 1 to n from input to output. Each stage has an identical structure; the links form a perfect shuffle permutation and connect to N/2 interchange boxes or switching elements. The perfect shuffle permutation [Sto71] moves an item from location S to location $(2S + \lfloor 2S/N \rfloor) \bmod N$. The interchange boxes have four states as shown in Figure 3.14(b). Two items can pass straight through or be exchanged, or an item on either input can be broadcast to both outputs. Due to the way the stages are labeled, at stage i, inputs that differ in the $n-i^{th}$ bit are compared and can be exchanged (the addresses paired can be obtained by setting all boxes in



(a)



(b)

Figure 3.14 (a) 8x8 Omega Network
(b) Four States of an Interchange Box
[Law75]

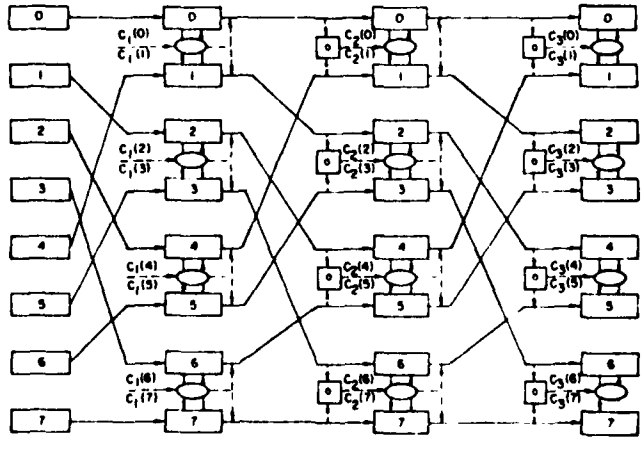


Figure 3.15 8x8 Expanded Shuffle-Exchange Network
[LaS76]

Figure 3.14(a) to straight and moving the input addresses on the established path throughout each stage). An Omega network can be controlled in a distributed fashion using routing tags which will be described in Section 3.4.

In [Law75] the ability of the Omega network to access vectors for processors (connected to the input) from matrices stored in memory (connected to the output) was investigated. It was shown that if a matrix is stored in memory in a skewed fashion, the Omega network provides conflict free access and alignment of rows, columns, diagonals, backward diagonals, and $N^{1/2}xN^{1/2}$ partitions in either row or column major order. It can also produce $N^{1/2}$-vector fanout and duplication functions.

3.2.9 The Extended Shuffle-Exchange Network

The extended shuffle-exchange network, shown for N=8 in Figure 3.15, and its capabilities are discussed by Lang and Stone in [LaS76]. It is a multi-stage version of the (single stage) shuffle-exchange network in [Sto71]. Structurally it is identical to the Omega network just discussed. The only difference is that it does not include the broadcast capability. In [LaS76], a simplified distributed control scheme is proposed that will be discussed in Section 3.4. This scheme is not quite a general as Lawrie's, however they show that it allows the network to perform some useful permutations. These include the uniform shift (S connects to S+k mod N, $0 \leq k, S < N$), unscrambling p-ordered vectors, and interchange of elements 2^{n-r} apart (which is used in some FFT algorithms). The network can also be used for partitioning 2^n processors into blocks of 2^r (with slight modification of the control algorithm).

3.2.10 The Indirect Binary n-Cube Network

Pease's indirect binary n-cube network is shown in Figure 3.16(a) for N=16 (n=4). Inputs and outputs are labeled from 1 to n [Pea77]. At stage i, input addresses that differ in their $i-1^{st}$ bit can be exchanged, as illustrated in the figure. This network also supports only two states, straight and exchange, in its switching elements (shown in Figure 3.16(b)). A hierarchical centralized control is proposed to set the switch states.

In [Pea77], it is shown that the network supports the communication requirements of a large array of processors working on massive numerical problems with a high degree of parallelism. Algorithms examined include those used in the solution of partial differential equations in two and three dimensions, the radix-2 FFT and other signal processing algorithms. Also, performing matrix operations, in particular matrix multiplication, is discussed.

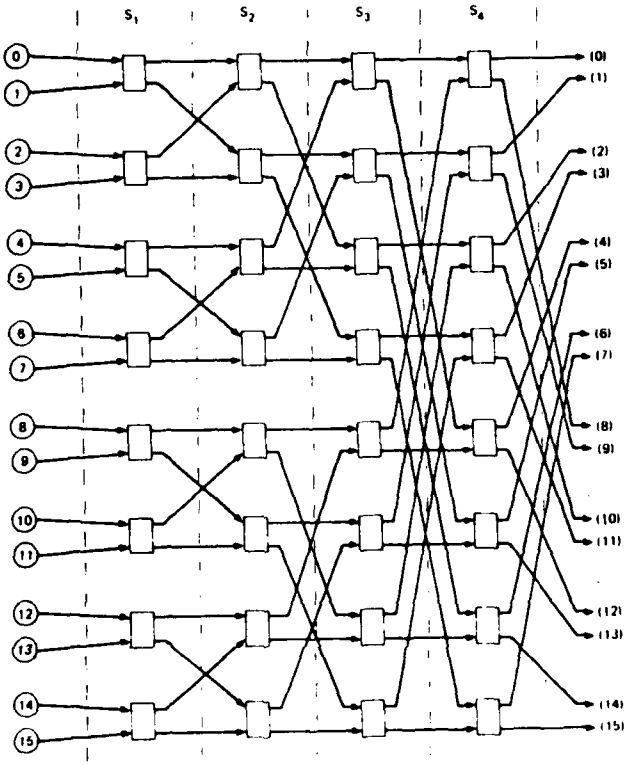
3.2.11 The Generalized Cube Network

The Generalized Cube network is a multistage cube type network topology that was introduced as a standard for comparing network topologies [SiS78]. The network has N inputs and N outputs, in Figure 3.17, N=8. The Generalized Cube topology has n stages, where each stage consists of a set of N lines connected to N/2 interchange boxes. Each interchange box is a two-input, two-output crossbar. The labels of the input/output lines entering the upper and lower inputs of an interchange box serve as the labels for the upper and lower outputs, respectively. Each interchange box can be set to one of the four legitimate states shown.

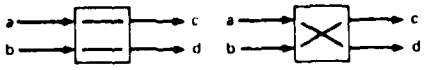
The connections in this network are based on the *cube interconnection functions* [Sie77]. Let $P = p_{n-1} \cdots p_1 p_0$ be the binary representation of an arbitrary I/O line label. Then the n cube interconnection functions can be defined as:

$$cube_i(p_{n-1} \cdots p_1 p_0) = p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_1 p_0$$

where $0 \leq i < n$, $0 \leq P < N$, and $\bar{p}_i$ denotes the complement of p_i. This means



(a)

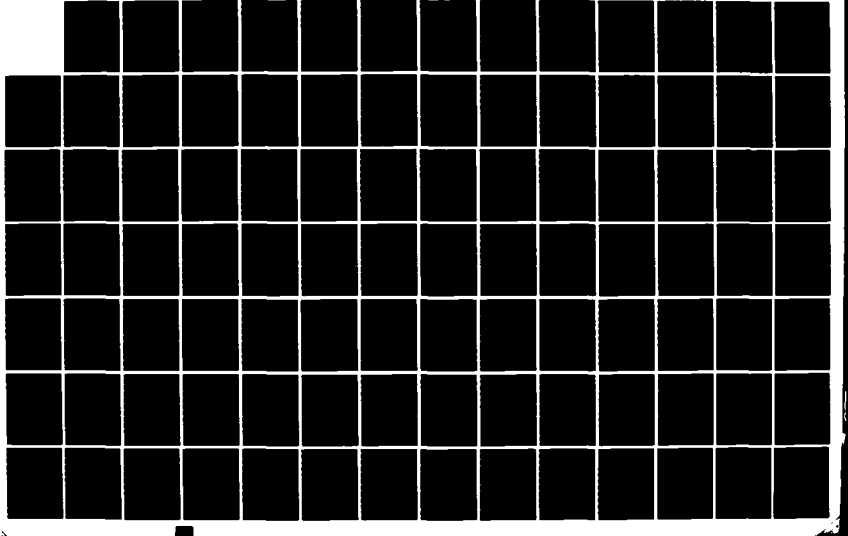


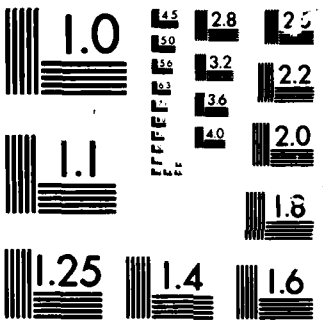
(b)

Figure 3.16 (a) 16x16 Indirect Binary 4-Cube Network
(b) Two States of an Interchange Box
[Pea77]

AD-A167 316 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR.. (U) PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING L J SEIGEL ET AL. 2/4

UNCLASSIFIED MAR 83 TR-EE-83-11 ARO-18790.17-EL-APP-A F/G 9/2 NL





MICROCOPY CHART

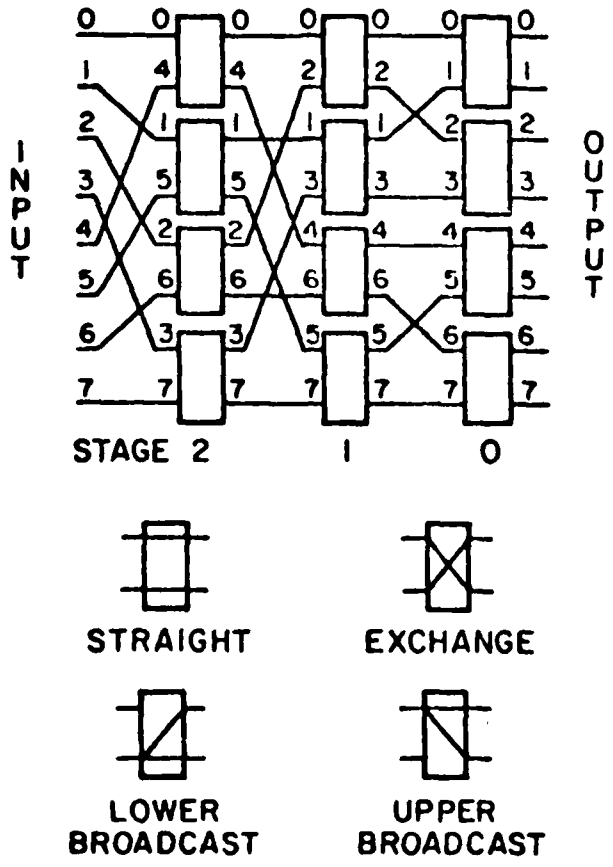


Figure 3.17 8x8 Generalized Cube Network. The four valid states of an interchange box are shown.

that the $cube_i$ interconnection function connects P to $cube_i(P)$, where $cube_i(P)$ is the I/O line whose label differs from P in just the i^{th} bit position. Stage i of the Generalized Cube topology contains the cube interconnection function. That is, it pairs I/O lines that differ in the i^{th} bit position. The other networks that have been discussed so far that are also based on the cube interconnection functions are the Benes, bitonic sorter, SW-banyan (S=F=2), Flip, Omega, extended shuffle-exchange, and the indirect binary n-cube. These networks are therefore referred to as *cube type* networks. Networks to be discussed that are also in this category are the baseline, reverse baseline, certain HEP networks, some Delta networks, and the reverse-exchange network.

3.2.12 The ADM and IADM Networks

The augmented data manipulator (ADM) network [SiS78] is shown in Figure 3.18 for N=8. It has the same structure as Feng's data manipulator discussed in Section 3.2.6 [Fen74]. In the ADM network, a *stage* consists of N *switching elements* or *nodes* and the 3N data paths that are connected to the inputs of a succeeding stage. At stage i of the ADM network, $0 \leq i < n$, the first output of node j is connected to the input of node $(j - 2^i) \bmod N$ of the next stage; the second output is connected to the input of node j; and the third output is connected to the input of node $(j + 2^i) \bmod N$. Because $(j - 2^{n-1})$ equals $(j + 2^{n-1}) \bmod N$, there are actually only two distinct data paths instead of three from each node in stage n−1 (in the figure, stage 2). There is an additional set of N nodes at the output stage. The *Inverse ADM (IADM)* network shown in Figure 3.19 is identical in structure to the ADM except that the stages are transversed from low order to high order (i.e. in the opposite order). The difference between these networks and the data manipulator is that the switching elements are controlled individually. This is done with routing tags that will be discussed in Section 3.4.

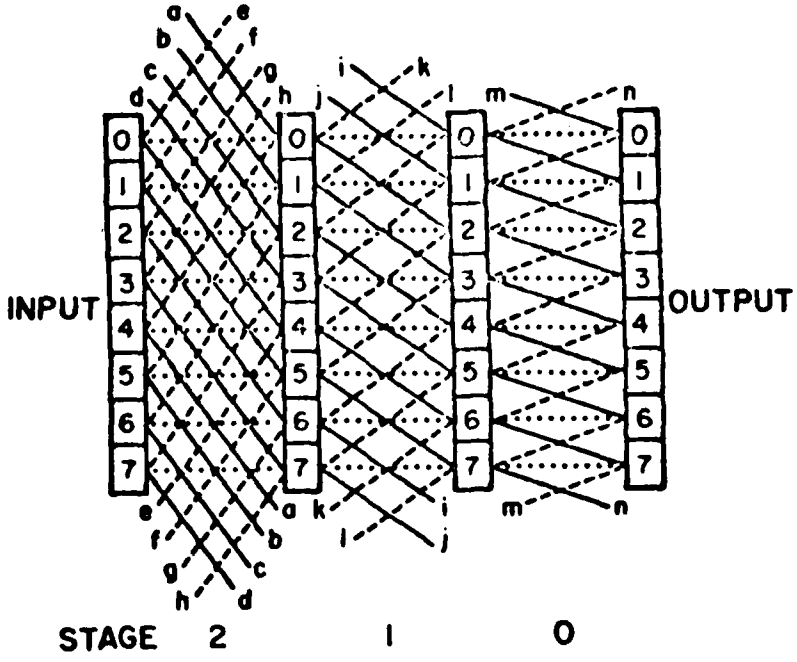


Figure 3.18 8x8 Augmented Data Manipulator Network. Lower case letters represent end-around connections.

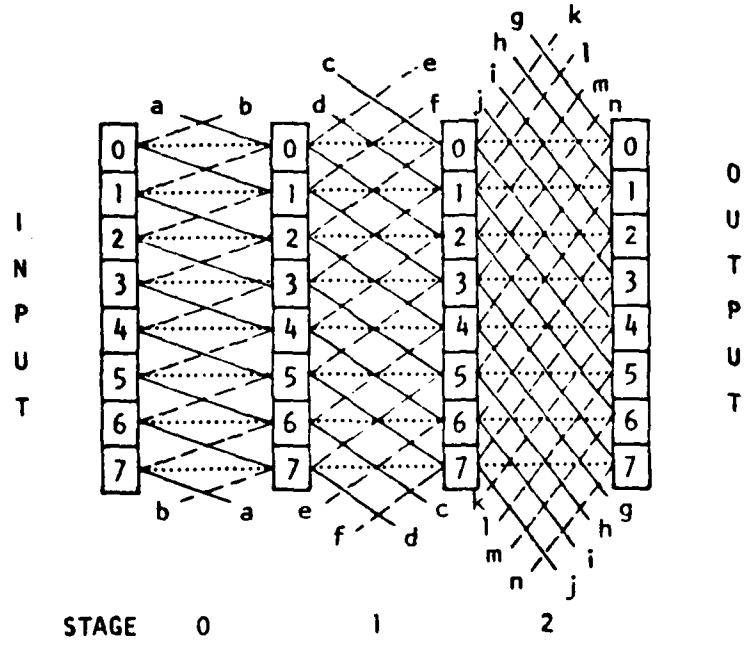


Figure 3.19 8x8 Inverse Augmented Data Manipulator Network

Both of these networks are based on the *PM2I (Plus-Minus 2^i) interconnection functions* [Sie77]. There are 2n of these functions defined by $PM2_{+i}(j) = j + 2^i \bmod N$ and $PM2_{-i}(j) = j - 2^i \bmod N$ for $0 \leq j < N$, $0 \leq i < n$, where $-x \bmod N = N - x \bmod N$. (Note $PM2_{+(n-1)} = PM2_{-(n-1)}$.) The *data manipulator* and the *Gamma* (to be discussed) networks are also based on the PM2I interconnection functions. Because they are all so closely related to the data manipulator, they will all be referred to as data manipulator type networks.

3.2.13 The Baseline Network

The baseline network was presented in [WuF78] by Wu and Feng as a standard for comparing network topologies. Its topology is generated in a recursive fashion. A column of N/2 2x2 switching elements form the first stage. The switching elements are numbered from 0 to (N/2)−1 with binary addresses of the form $p_{n-2} \cdots p_1 p_0$. The upper input and output lines are labeled $p_{n-2} \cdots p_1 p_0 0$ and the lower lines are labeled $p_{n-2} \cdots p_1 p_0 1$. The first stage is connected to two N/2 x N/2 subnetworks, C_0 (upper network) and C_1 (lower network). The upper outputs from the first stage are connected to C_0, ordered by switching element number and the lower outputs are connected to C_1 in the same order. For each of the subnetworks this process is repeated until the sub-subnetworks reach size 2x2. The result is shown in Figure 3.20 for N=16. The number of iterations required is n−1 resulting in an n stage network. The stages are labeled 0 to n−1 from side 1 to side 2. The network is controlled using routing tags that will be discussed in Section 3.4. The switching elements used here can assume only the straight and exchange states.

A reverse baseline is just the inverse network, which is equivalent to traversing a baseline from side 2 to side 1. The sides are not labeled as input or output because the network is defined to be bidirectional. Paths through the network are allowed to originate on either side and terminate on either side.

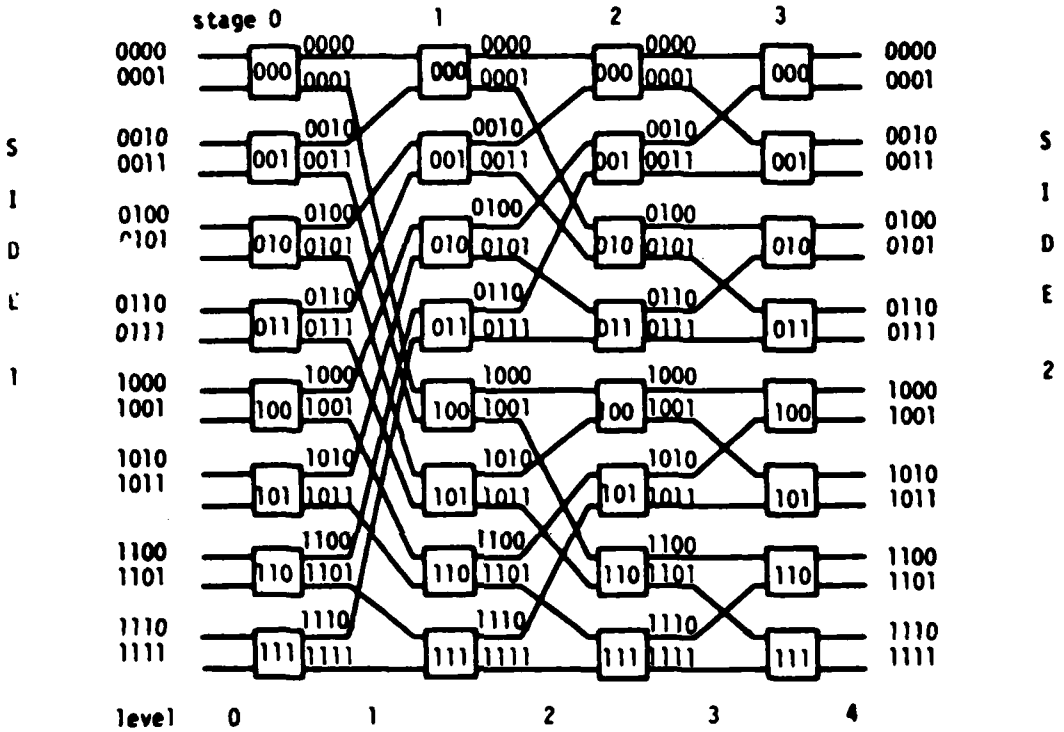


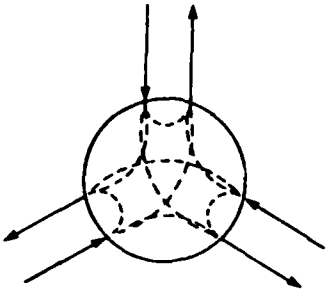
Figure 3.20 16x16 Baseline Network [WuF80]

3.2.14 HEP Networks

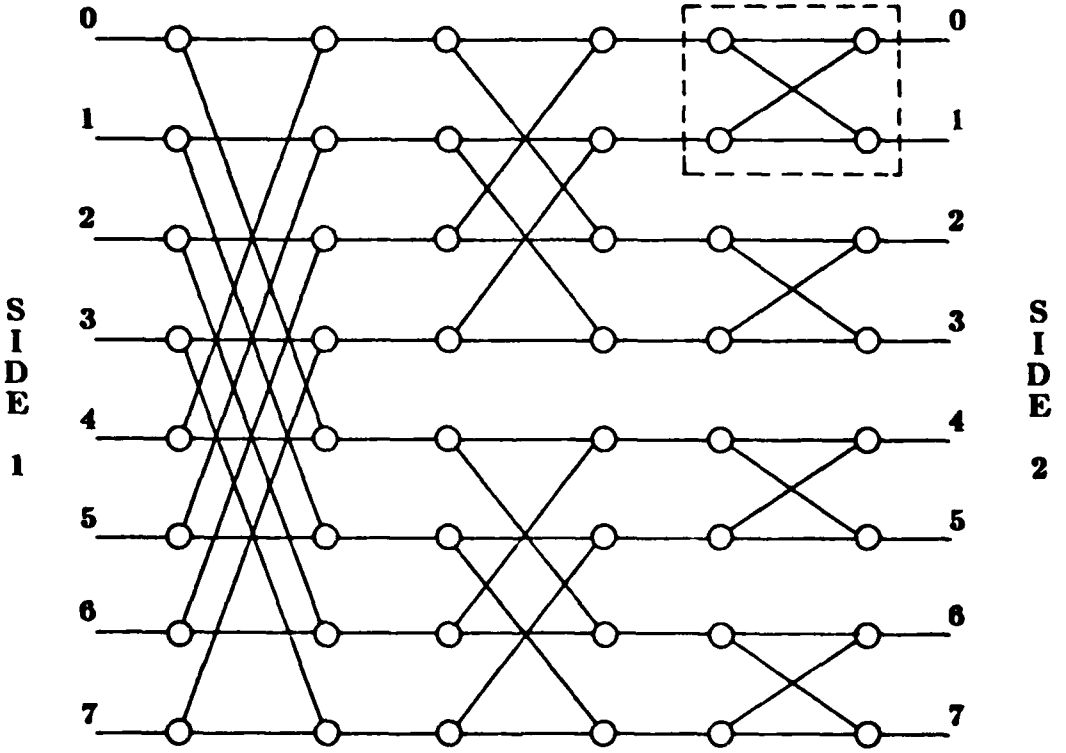
The HEP network is not defined according to any particular topology. Rather, the function of a switching element is defined and the network consists of any desired structure that can be obtained by interconnecting the switching elements. A switching element is shown in Figure 3.21(a). The dashed lines in the figure show possible paths through the switch. The switching element can be viewed as a three port, full duplex switch. Since any input can be connected to any output, it can also be viewed as a 3x3 crossbar switch. Any network constructed from these switching elements would be of the dual path bidirectional network type.

The HEP network is implemented as packet switched. The switch has the rather unique property that when two or more packets contend for the same output, one of them will be given access to it and the remainder will be intentionally mis-routed. This eliminates the need for buffering packets inside the switch. To compensate for the mis-route, a priority word that accompanies each packet is incremented. Packets with the highest priority are given preference when conflicts occur. Packets whose priority has reached the maximum (15) are handled specially. There is a path called an Eulerian circuit that traverses every port exactly once in each direction [Smi81a]. Packets with this priority are sent on such an Eulerian circuit, independent of their destination address. This guarantees (1) that the packet will reach its destination and (2) that maximum priority packets will not conflict with one another. This path is not necessarily optimal but it is guaranteed. To determine how a packet should be routed, a routing table associated with each output port (stored in every node) indicates the optimal path for all possible destination addresses. The table is written to the nodes when the system is initialized.

The need for routing tables is a direct result of the arbitrary way in which switch nodes can be connected. It is only the regular structure of the other networks that allows them to be controlled without using routing tables.



(a)



(b)

Figure 3.21 (a) HEP Switching Element
(b) One Possible 8x8 HEP Network

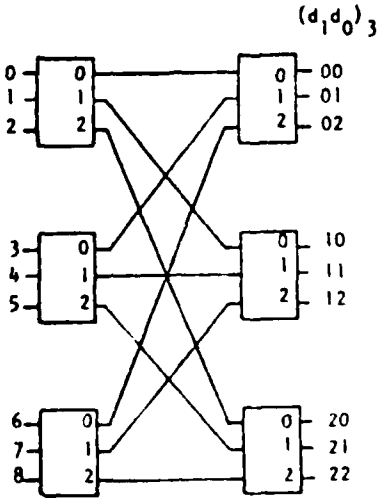
To facilitate comparison of the HEP network with the other multistage networks, the topology shown in Figure 3.21(b) is assumed. Note that the lines in the figure are actually bidirectional, consisting of a pair of unidirectional lines going opposite directions. Single lines are shown for clarity. Functionally, it is equivalent to the Generalized Cube network shown in Figure 3.17. The four HEP switching elements within the dashed lines in Figure 3.21(b) perform the same function as one interchange box in Figure 3.17, *however, no broadcast capability is included.* To see the structural equivalence, notice that the same addresses are paired at a given interchange box and its functional equivalent in the HEP network. For example, addresses 1 and 5 in stage 2.

3.2.15 Delta Networks

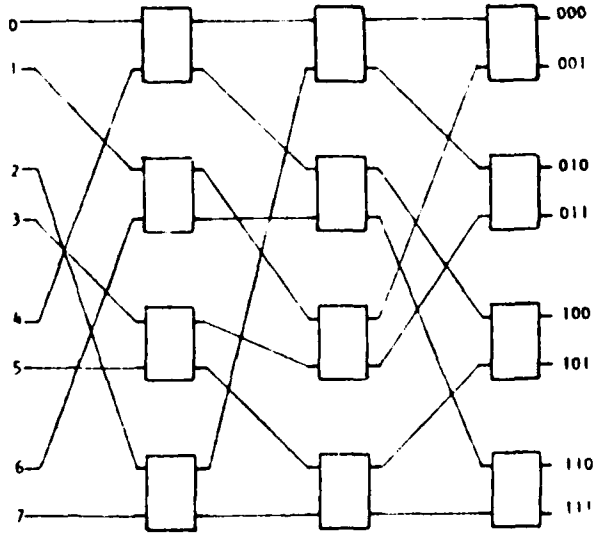
Delta networks are a class of networks introduced by Patel [Pat79]. They are constructed from bxb crossbars with outputs labeled from 0 to b−1. A b^nxb^n delta network contains nb^{n-1} bxb crossbars. Any network that can be constructed using the following rules is a member of the class: (1) No more than b^{n-1} crossbars can be used in one stage and no more than n stages are created; and (2) Each bxb crossbar that receives inputs from other bxb crossbars must have all its inputs connected to identically labeled outputs. A 3^2x3^2 delta network is shown in Figure 3.22(a) and an 8x8 is shown in Figure 3.22(b).

One property of delta networks is that there is exactly one path from any input to any output. Another property (from which the name is derived) is that they are *digit-controllable.* The setting of each crossbar is determined by a base b digit at each input. This control scheme will be described in detail in Section 3.4.

The Delta class of networks includes the bitonic sorter, all rectangular SW-banyans, the Flip, Omega, extended shuffle-exchange, indirect binary n-cube, Generalized Cube, baseline, and reverse baseline networks when they are implemented with



(a)



(b)

Figure 3.22 (a) 9x9 Delta Network
(b) 8x8 Delta Network
[Pat79]

bxb crossbars (typically b=2). It does not include any of the data manipulator type networks because they all have multiple paths from input to output. The data manipulator type networks are, however, digit controllable, as will be shown in Section 3.4.

3.2.16 The Reverse-Exchange Network

The reverse-exchange network was introduced in [WuF79a]. It is designed to perform arbitrary permutations of its inputs in two passes. A size N=8 version of this network is shown in Figure 3.23. Comparing it to the Omega network in Figure 3.14, it is clear that the two networks are topologically identical. Consequently, it is topologically equivalent to all the cube type networks (this was pointed out in [WuF79a]). The difference between this and the omega network is in how the inputs are labeled. The different labeling gives the reverse-exchange network different permuting capabilities.

The reverse-exchange network is related to the Benes̆ network in the following way (compare Figure 3.4 and 3.23). If the interchange boxes in Figure 3.23 are rearranged so that they are in order by box number, then these three stages are identical to the first three stages of the Benes̆ network in Figure 3.4. If the shuffle-exchange network is reversed so that the output becomes the input and vice versa, it is equivalent to the last three stages of the Benes̆ network. The algorithm used to control the reverse-exchange network is based on the work of Opferman and Tsao-Wu [OpT71a], Anderson [And77], and Lenfant [Len78] for controlling the Benes̆ network.

3.2.17 The Gamma Network

The Gamma network is shown in Figure 3.24 for N=8 [PaR82]. Its structure is identical to that of the IADM network discussed in Section 3.2.12. The difference between these networks is that the IADM switching elements connect one of their inputs to one output at a given time (or to multiple outputs for broadcasting); the

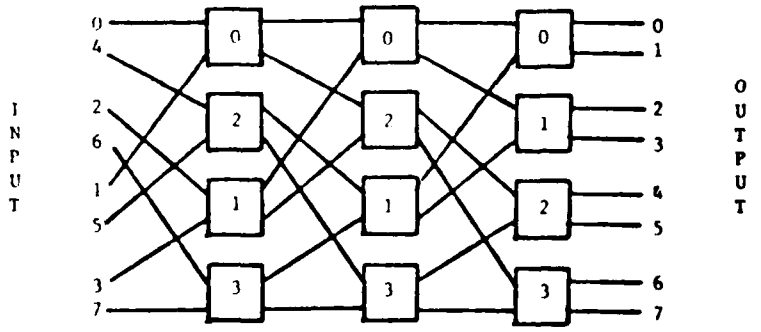


Figure 3.23 8x8 Reverse Exchange Network [WuF79a]

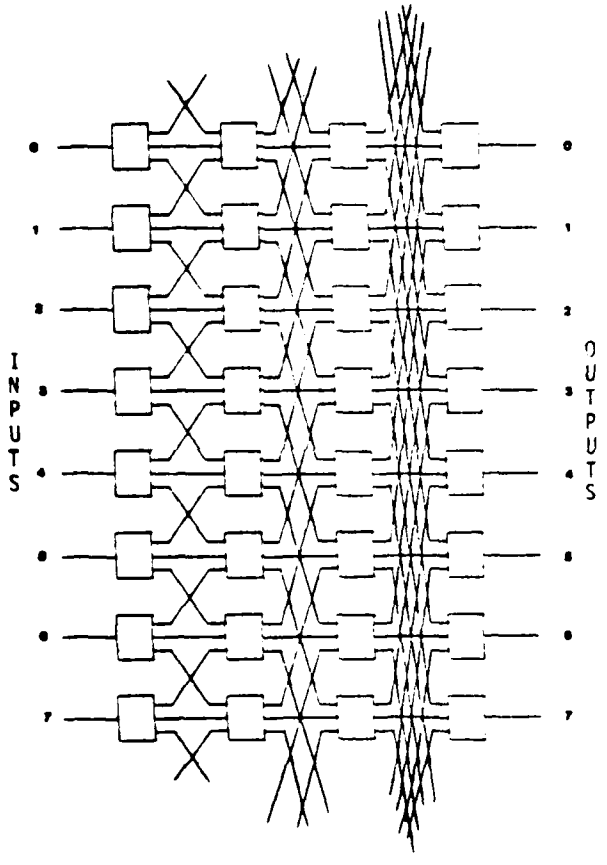


Figure 3.24 8x8 Gamma Network [PaR82]

Gamma network switching elements are 3x3 crossbars. In [PaR82], it is shown that the Gamma network can perform some permutation connections that the IADM cannot, e.g. the perfect shuffle.

3.2.18 Conclusions

The number of interconnection networks that have been proposed since 1953 (which for all intents and purposes was the birth of the multistage network), when Clos investigated cheaper ways to build a crossbar, is very large. Seventeen specific network topologies and seven different classes of networks have been surveyed here. Many of the networks discussed are very similar while others are quite different. To place all these networks into perspective, a family tree for multistage interconnection networks is shown in Figure 3.25. Each network or class has a date next to it to indicate when it was first presented in the literature. Four broad categories are defined: Permutation Networks, Multiple Path Networks, Single Path Networks and Fault Tolerant Networks. Fault tolerant networks will be discussed in Section 3.5, where the remainder of that family tree will be filled in.

Permutation networks are those that can connect their inputs to their outputs in any arbitrary way as long as no two inputs want the same output. For an N input, N output network, there are N! possibilities. The only networks discussed that can do this were those in the Clos class including the Benes and the Waksman modification of the Benes. The Benes network is the least expensive network in the Clos class.

Multiple path networks are those that have more than one path between a given input and output (with the possible exception that there is only one path when input=output). This category includes all of the members of the Permutation Network category listed and the data manipulator type networks. The former tend to have many more paths per input/output pair than the latter.

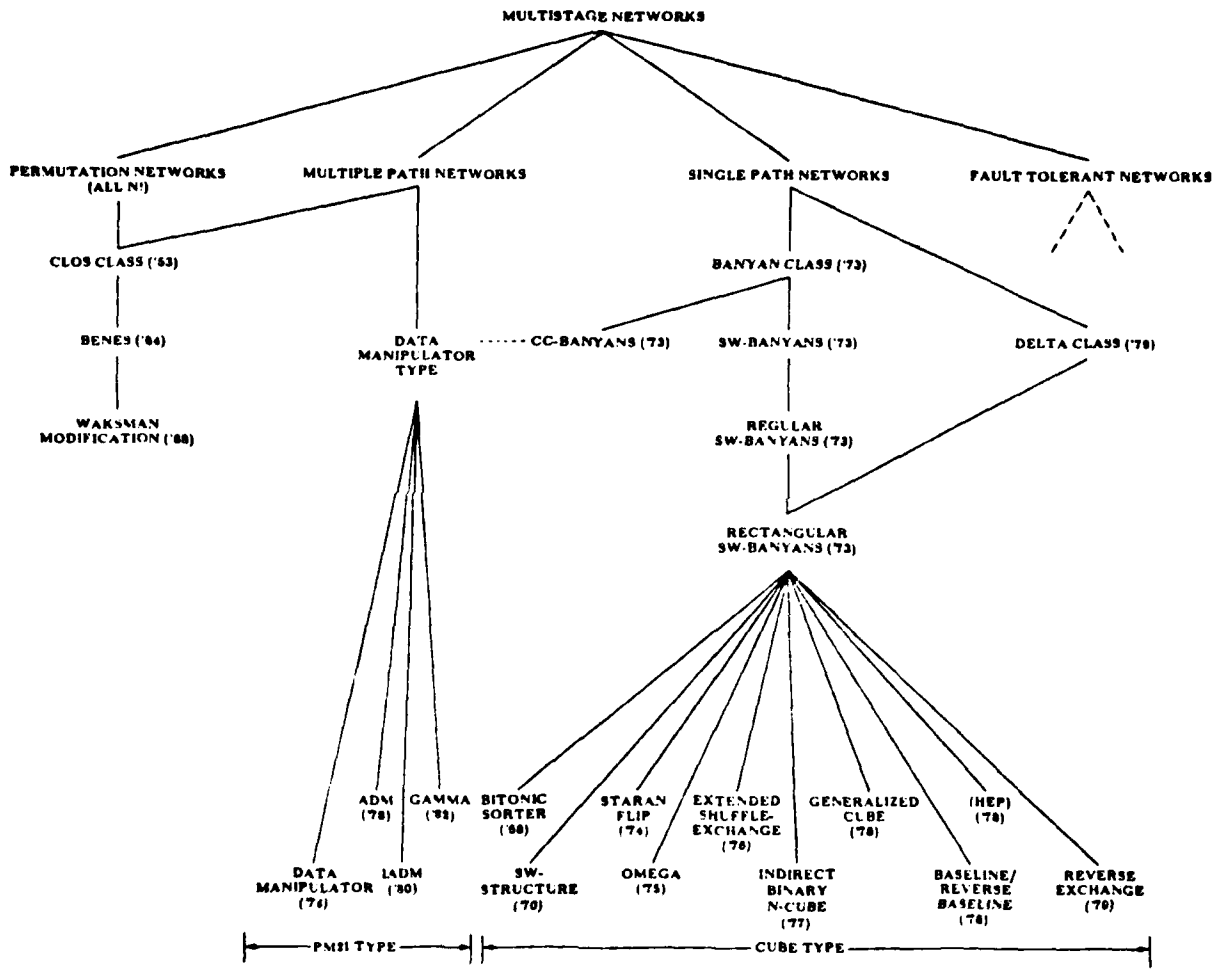


Figure 3.25 Multistage Interconnection Network Family Tree

The data manipulator type networks all have topologies constructed from straight, $+2^i$, and -2^i type connections. Thus they are generically referred to as PM2I type networks. Included are the data manipulator, Augmented Data Manipulator (ADM), inverse ADM (IADM) and Gamma networks. The ADM's capabilities are a superset of the data manipulator's. The ADM and IADM are comparable, and the Gamma network is the most powerful, having capabilities that are a superset of those of the IADM.

The single path networks have exactly one path between every arbitrary input/output pair. The two classes listed in this category are the Banyan class and the Delta class. The Banyan class is extremely general since it is defined in terms of unlabeled graphs. Each node (switching element) in the graph can have a different number of incoming and outgoing arcs (links) than the other nodes as long as some basic rules are followed. Of practical interest are the somewhat more structured subclasses called CC-banyans and SW-banyans. The Delta class is shown at this level in the tree because, in some qualitative sense, it is approximately as general as the CC-banyans and SW-banyans. The dashed line between the CC-banyans and the data manipulator type networks indicates that they are relatives. This is in the sense that some CC-banyans are based on straight and $+2^i$ type connections.

Within the class of SW-banyans is the subclass of regular SW-banyans. Regular implies that all the nodes (switching elements) are the same (i.e. have the same number of inputs and outputs as the other nodes). Within this class are the rectangular SW-banyans in which each node has the same number of inputs and outputs. Since the Delta class has switching elements with equal numbers of inputs and outputs, rectangular SW-banyans are also a subclass of the Delta class. Connections between switching elements in the Delta class are more general.

The class of rectangular SW-banyans contains more specific instances that have been discussed in the literature than any other class listed. Included (in historical order) are the bitonic sorter, SW-structure, STARAN Flip, Omega, extended shuffle-

exchange, indirect binary n-cube, Generalized Cube, Baseline, Reverse Baseline, some instances of HEP networks, and the reverse-exchange. HEP is shown in parentheses because not all HEP type networks belong to this class.

Two observations can be made regarding the consequences of membership in the rectangular SW-banyan class. First, when these networks are used in an MIMD mode, where random requests for connection come in, their performance is the same. In [Pat79] an analysis showed that all networks of the same size in the Delta class constructed from bxb crossbar type switching elements have the same performance. That is they have the same bandwidth and probability of accepting a request for access.

The second observation is with regard to use in SIMD mode. As was pointed out in [WuF79a], topological equivalence between two networks implies a one-to-one and onto mapping between the components of the networks and does not necessarily imply functional equivalence. Wu and Feng's proposed definition of *functional* equivalence is that two networks must have the same set of realizable connection capabilities using the same control information (with a possible mapping of the information to its proper location) [WuF79a]. It has been shown, however, that with an appropriate renumbering of inputs and outputs, it is possible to convert one network in this class into another [SiS78,WuF79a].

It can be concluded that any of the capabilities shown for one network can be built into another with suitable modifications (often minimal). Hence there is a significant body of literature describing a wide variety of things rectangular SW-banyan class networks can do. The capabilities discussed in this section included sorting bitonic sequences of numbers, partitioning resources, forming tree structures, accessing various vectors from matrices, and duplicating and spacing data out. Permutations that can be performed include flips, uniform shifts, and several useful to FFT algorithms.

3.3 Switching Element Implementations

3.3.1 Introduction

In most of the papers introducing the networks surveyed in the last section, the switching elements used to construct the network were functionally specified but no particular implementation was proposed. The simplest realizations of interchange boxes specified have been designed for use in circuit switched networks. Levitt, et al. proposed one of the first designs called a basic cell [LeG68]; Joel calls the same design a β-element [Joe68]; Smith and Siegel use a simple externally controlled multiplexer [SmS78, Smi81b]; and Patel suggests a slightly more complex, fixed priority, 2x2 crossbar implementation [Pat79]. Owing to advances in LSI technology, some significantly more sophisticated designs have been proposed (and implemented) recently. Ciminiera and Serra propose implementing whole subnetworks of circuit switched 2x2 crossbars and their associated control logic on one LSI chip [CiS81]. Premkumar, et al. describe in considerable detail, the design and implementation of 2x3 switching nodes capable of "simultaneous" (in the same network clock cycle) circuit and packet switching [PrK80a]. In the following, these various implementations will be discussed in more detail.

3.3.2 Early Switching Elements

The earliest suggested use of small crossbars for constructing large interconnection networks is attributed to Clos [Clo53]. His intended application, however, was for telephone switching networks. At the time, connecting large numbers of processors was unthinkable. The Benes network, a special case of the Clos networks is constructed from 2x2 crossbars [Ben65]. Benes' work is primarily concerned with the capabilities and control of the network as a whole, not the design of the 2x2 crossbars. Given that circuit switching is performed and the switches are controlled externally, the actual

implementation of the crossbars is understandably one of the less important aspects of the network. A 2x2 crossbar simply consists of two small 2-to-1 multiplexers.

One of the first discussions suggesting that networks of 2x2 crossbars be used to connect multiple computers is in [LeG68]. The motivation behind the work is the desire for ultra-reliable computer systems for aerospace applications. In these applications only a relatively small number of computers are involved. Levitt, et al. provide a circuit diagram (shown in Figure 3.26) of a simple 2x2 crossbar, so they can analyze the types of faults that could occur and how to circumvent them. The crossbars contain flip-flops that store the state of the switch. Altogether, the crossbar requires two flip-flops, five AND gates and two OR gates. One crossbar passes one bit of information, so a number of them in parallel are required to pass bytes or words of information.

Though Waksman [Wak68] presents interesting results on controlling Beneš networks, he apparently has little or no knowledge of hardware. There is a rather humorous remark regarding the implementation of 2x2 crossbars in the introduction which asserts, "Let the 'elementary cell' be the basic building block of such a network, which ... presumably can be constructed using a single flip-flop."

3.3.3 Omega and Indirect Binary n-Cube Switching Elements

The papers by Lawrie [Law75] and Pease [Pea77] each make recommendations regarding the design of network components but neither presents a specific implementation. Lawrie notes that his omega network of size N can be partitioned to form two identical subnetworks, which are themselves omega networks of size N/2. He recommends implementing the size N=4 subnetworks as one module containing four 2x2 crossbars or as one 4x4 crossbar. These modules are then interconnected with 2x2 crossbars to form a larger omega network. An example of this configuration is shown in Figure 3.27. He points out that a large network containing 4x4 crossbars has more powerful capabilities than that containing the four 2x2 crossbars. Lawrie assumes the

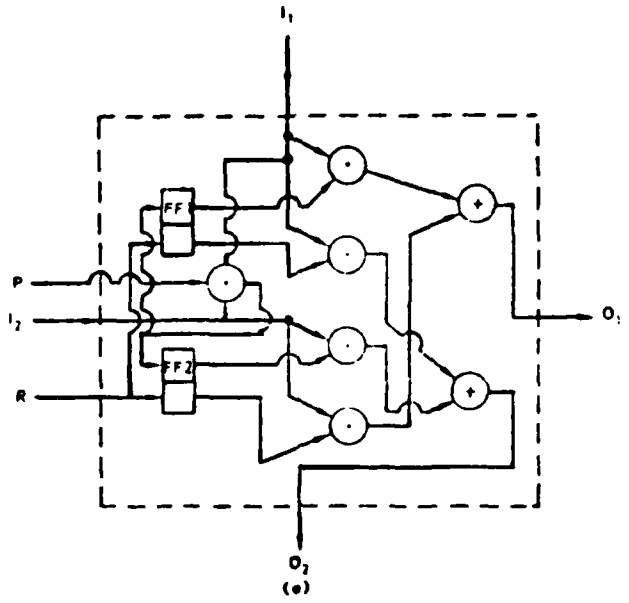


Figure 3.26 2×2 Crossbar Implementation [LeG68]

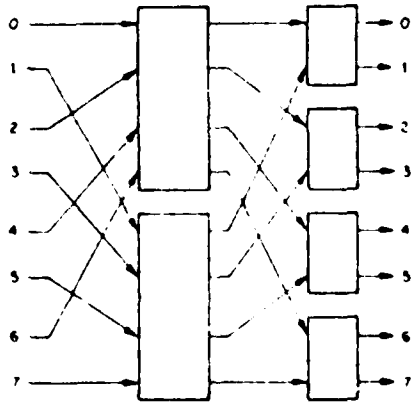


Figure 3.27 An Omega Network with 4×4 and 2×2 Switching Elements [Law75]

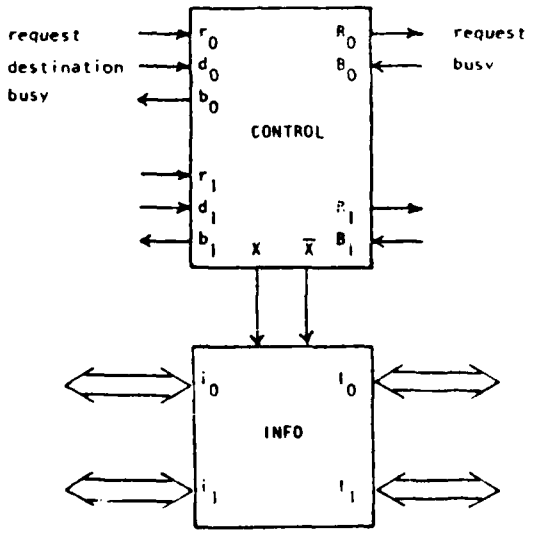
2x2 crossbars in an omega network have a broadcast capability. Implementing this capability increases the complexity of the control logic associated with each 2x2 or 4x4 crossbar [McA80]. The increase is not prohibitive, however. Pease assumes no such broadcast capability is present in the 2x2 crossbars which compose his indirect binary n-cube network. His only recommendation with regard to implementation is that each 2x2 crossbar be placed on one LSI chip.

3.3.4 Delta Network Switching Element

Patel has defined a class of delta networks which are used in a circuit switched mode to connect processors to memories [Pat79]. The networks are constructed from bxb crossbars. The only design presented is for 2x2 crossbars as shown in Figure 3.28. The crossbars are controlled by routing tags. For a 2x2 crossbar, one bit from each input, d_0 and d_1, determines its state. Based on the equations in the figure, 18 gates are required to implement the control logic and 6W gates for the INFO or path select logic, where W is the path width. The design shown is unidirectional. The priority scheme for each node is fixed; the upper input always has priority over the lower input. The result of this for an Omega network (a member of the class) is that one processor (at input 0) connected to the network will never be blocked while another (at input N−1) will only be able to establish a path consisting of links no other processor wants. All other processors will have non-equal probabilities of establishing their desired paths. The priority among processors can be randomized by connecting the two outputs of each 2x2 module to two different priority input ports at the next stage. Doing this, however, changes the permuting ability of the network in SIMD mode. It is better if the priority in each node alternates to assure all processors equal access to memories, on the average.

3.3.5 An Optimal Switching Element Size Study

Ciminiera and Serra performed a study whose goal was to determine an optimal packaging of some number of 2x2 crossbars onto one LSI chip [CiS80,CiS81]. They estimated logic and pin requirements for various size Omega networks. Shown in Figure 3.29 is a block diagram of a size four implementation. The block labeled 'C' contains four 2x2 crossbars. Routing tags control the switch setting. If an NxN omega network is implemented on one chip, the control unit will examine $\log_2 N$ bits of each tag. Details of this procedure can be found in Section 3.4.



$$X = r_0 d_0 + \overline{r}_0 \overline{d}_1 \qquad \overline{X} = r_0 \overline{d}_0 + \overline{r}_0 d_1$$

$$R_0 = r_0 \overline{d}_0 + r_1 \overline{d}_1 \qquad R_1 = r_0 d_0 + r_1 d_1$$

$$b_0 = \overline{X} B_0 + X B_1 \qquad b_1 = X B_0 + \overline{X} B_1 + r_0 d_0 d_1 + r_0 \overline{d}_0 \overline{d}_1$$

$$I_0 = I_0 \overline{X} + I_1 X \qquad I_1 = I_0 X + I_1 \overline{X}$$

Figure 3.28 2×2 Crossbar Implementation [Pat79]

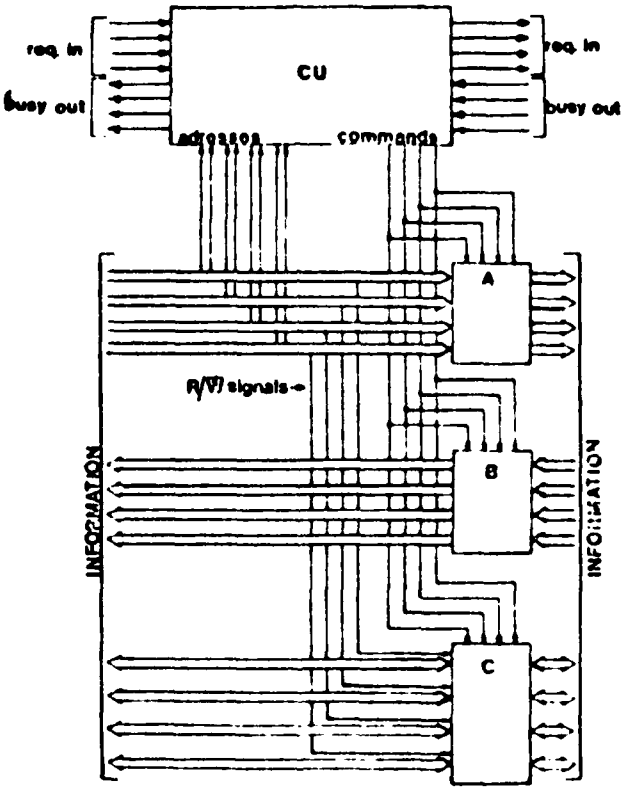


Figure 3.29 LSI Module of 4×4 Omega [CiS81]

In [CiS81], asynchronous circuit switching is assumed. When a tag is presented to the input of a chip, the control unit checks to see if the desired connection can be established. If connections existing in the chip do not block the new path, the states of the appropriate 2x2 crossbars are set and the "busy-out" signal corresponding to the "request-in" is turned off. The tag is then forwarded to the next chip via the newly established connection. Their analysis shows that if the maximum number of pins per chip is 60, an optimal configuration contains one 4x4 omega network, 6 bits wide. The network can be unidirectional or bidirectional. If the pin limit is doubled, an 8x8 omega network can be accommodated. If it is unidirectional, the optimal path width is 7 bits, or if bidirectional, 6 bits. Optimality is defined in terms of maximizing the logic per pin ratio without exceeding logic or pin limitations.

3.3.6 Banyan Switching Elements

One result noted in [CiS81] is that the pin limit is always exceeded before the logic limit. Aware of this fact, Tripathi and Lipovski [TrL79] suggest including a packet switching capability with nodes that also circuit switch. Doing so increases the logic in a node without a substantial increase in the pin count. This has been implemented in an SW-banyan network. Since the SW-banyan connects processors to memories, and because of memory access timing, they found that packet switching could be overlapped with circuit switching with a negligible time penalty. A circuit is only used in the latter part of a memory cycle, thus packets can be moved during the first part. Conversely, packets require time to negotiate a movement from one node to another, so this can be done while the circuit is in use. Though no design is presented in [TrL79], design issues for packet switched implementations are enumerated. Included are node-to-node protocols, buffering at the nodes, packet assembly/disassembly, error correction coding, acknowledgment, time-out, and retransmissions.

A detailed discussion of the design of nodes in an SW-banyan network is presented by Premkumar, et al., [PrK80a]. Two parameters determine the topology of the network, spread and fanout (S and F). An SW-banyan with S=2 and F=3 is shown in Figure 3.30. An N input SW-banyan has $\log_2 N$ levels. It is important to note that one node in the figure is *not* a 2x3 crossbar. One node can connect one of the incoming lines to one of the outgoing lines. Five nodes and the lines connecting them form the equivalent of one 2x3 crossbar as shown by the heavy lines in the figure. Interpretation of these graphs is discussed in detail in [McS82c].

The functional components in one 2x3 node are: (1) clock decode logic - a single network clock is converted into a six phase clock which determines all internal event sequences; (2) bus control logic - determines the current direction of the bidirectional circuit bus; (3) link control logic - establishes a path from one input to one output and arbitrates conflicts; (4) packet switch logic - implements the protocol for passing a packet from level to level; and (5) carry lookahead/priority logic - since all processors are bit-sliced, this logic allows several processors to be linked together to form one larger processor.

One packet is four bytes in length. The address or routing tag requires one byte and the data uses three. Packet switching is implemented such that two packets are processed simultaneously during the same cycle. The result is similar to having two parallel packet switched networks. Each node thus has two one byte buffers, one for each packet. Packets move as byte trains, progressing one level per clock cycle. A new packet can enter the same port every fourth cycle. Request and grant signals are used between nodes to negotiate the transfer of the first byte of a packet into the receiving node. Once the first byte is transferred, the remaining bytes will follow without interruption.

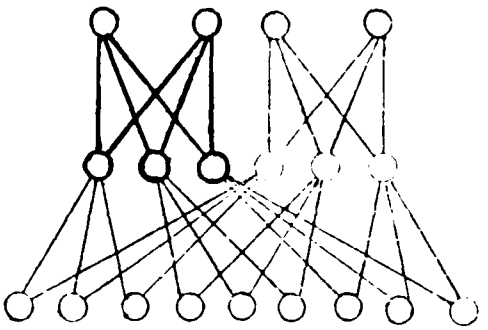


Figure 3.30 SW-Banyan with S=2, F=3. Heavy lines form a 2x3 crossbar.

3.3.7 Conclusions

The complexity of proposed switching elements varies considerably. The simplest are the telephone network crossbars that handle circuit switched serial communication lines. All of the centrally controlled circuit switching networks for connecting processors and memories tend to have very simple switching elements. However, the complexity of the controller is high since, in the general case, it must accept N requests for connections and generate $O(N \log_2 N)$ control signals. Packet switched networks have much more complex switching elements. Logic is needed to handle the buffering of packets and the protocol of transferring packets to other switching elements.

When control of the network is distributed among the switching elements, they become more complex regardless of whether they are circuit or packet switched. The switching elements in the SW-banyan network are very complex since they handle both circuit and packet switching. The two modes are time multiplexed, so a sizable number of control signals are needed to handshake with other switching elements and to keep track of the switch state. In the next section, distributed control methods are described, which will give an indication of the complexity of the control logic in a switching element.

3.4 Distributed Control Methods

3.4.1 Introduction

The Clos and Benes telephone switching networks have centralized control. The fastest known set-up algorithms for these networks were developed by Opferman and Tsao-Wu [OpT71a] and require $O(N\log_2 N)$ time for arbitrary sets of connections. For a large network, the time required to set the states of the switching elements is reasonable for telephone switching (e.g. 0.75 seconds), but not for computer communication. Lenfant found that he could speed up the set-up time in the Benes network for certain

classes of useful permutations called Frequently Used Bijections (FUB's) (a bijection is a permutation). Each FUB belongs to one of five classes and each class is characterized by up to four parameters whose size is a function of N. (For N=256, 39 bits encode the FUB.) There are n stages of control logic (one stage of control logic provides signals to two stages of the network). The FUB code is passed from stage to stage, so a packet switched network can be set up "on the fly," one step ahead of the data. This method is somewhat distributed but limited to the FUBs. There is no known method for completely distributing control of the Benes or Clos networks. Each network user cannot determine which switches it should use without knowing which ones all the other users have or will request when it requests access.

The reverse-exchange network is a $\log_2 N$ stage network proposed by Wu and Feng, designed to perform arbitrary permutations [WuF79a]. This is done by making two passes through the network. The first pass goes from input to output and the second, from output to input (see Figure 3.23). The effect of making two passes in this way is comparable to traversing the Benes network once. Thus, to control the reverse-exchange network, one of the algorithms for the Benes network must be used. To route arbitrary permutations, the control of the network cannot be fully distributed.

Some of the first designs for multistage interconnection networks used in computers had centralized control. They could do this efficiently because they were designed for SIMD operation and permuting data. This means that individual processors were not requesting network connections. Rather, the control processor issued an opcode to the network control unit specifying a particular permutation (or other) configuration to be established. Examples of this approach are described in [Bat76] for the Flip network and in [Fen74] for the data manipulator. The main advantage to this approach is that the switching elements are very simple (e.g. see Figure 3.12(b)) and therefore inexpensive and the control units are not very complex.

A network is considered to have *fully distributed control* if (1) each user calculates all the switch setting information required by the switching elements to be used for the current transmission and (2) each switching element can set its state based only on the control information associated with the transmissions it handles. The first network to be proposed that had fully distributed control was the bitonic sorter [Bat68]. Due to its designed purpose, its switching elements determine their state based on the transmitted data itself. This is therefore a rather trivial example of distributed control. However, the bitonic sorter can be modified to interconnect processors and memories and be controlled in a fully distributed way using routing tags. This is because it is almost functionally equivalent to the Generalized Cube network. Thus it can be modified to use the routing tag scheme that will be described for that network in Section 3.4.7.

Lawrie's routing tag scheme for the Omega network was the first of its kind to be proposed for fully distributing control of that network [Law75]. It is designed to route transmissions from one side of the network to the other. Lang and Stone then proposed a simplified version of Lawrie's scheme, for the extended shuffle-exchange network [LaS76]. Its capabilities are a subset of those of the Omega network. Their scheme is not as flexible, but it does allow some useful permutation connections to be set up. In [WuF78], Wu and Feng extended Lawrie's scheme for the Baseline network. Their scheme allows transmissions to be routed into and out of the same side of the network in addition to traversing it. Patel developed a general routing scheme that can be used by any of the networks in the Delta class [Pat79]. His scheme is identical to Lawrie's for any of the networks constructed from 2x2 switching elements (which includes the Omega network).

For the class of regular SW-banyan networks, Tripathi and Lipovski developed a general scheme whose capabilities are a superset of the Delta network scheme [TrL79]. It is more general because it deals with axb crossbar switching structures (as opposed to bxb only). The banyan scheme also includes a rerouting capability for avoiding faulty

switching elements. It does this by mis-routing a transmission and then allowing it to backtrack to a point from which it can move forward again to reach its desired destination.

The scheme discussed in [SiM81b] for the Generalized Cube network is designed for networks constructed from 2x2 crossbars or interchange boxes. It uses routing tags, but they are calculated in a way different than Lawrie's scheme and have different properties. They can, however, be used to control the Omega network or, with minor modifications, any of the other networks discussed that are topologically equivalent to the Generalized Cube.

In [McS82d] and [SiM81a] routing tag schemes are discussed for controlling the ADM and IADM networks in MIMD mode and SIMD mode respectively. These schemes are equally well suited to controlling the Gamma network and one of them can be readily used in the CC-banyan type shown in Figure 3.11 (see Section 3.4.8.1). Because these are all multiple path networks, the routing tag schemes are more sophisticated than those used by the cube type networks. The MIMD mode tag scheme has the ability to perform dynamic rerouting to avoid busy or faulty switching elements when possible, without backtracking [McS82d].

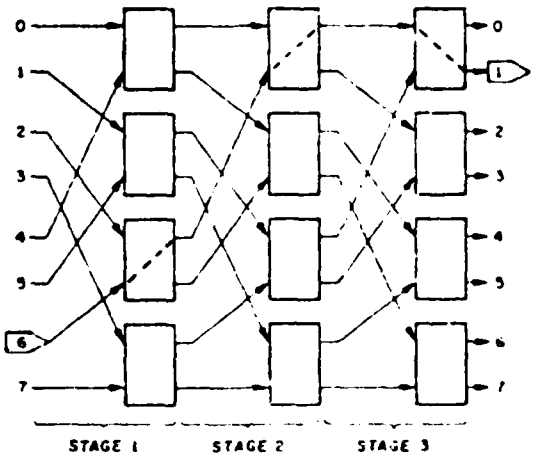
In the following sections the details of the fully distributed control schemes for the Omega, extended shuffle-exchange, Baseline, Delta, regular SW-banyan, Generalized Cube, ADM and IADM networks will be described. All these schemes use routing tags to distribute the control. If the network is packet switched, the tag is part of the header information in each packet. If it is circuit switched, the tag is held on the input data bus until a complete circuit is established. Further extensions to the basic schemes for the Generalized Cube, ADM and IADM networks are described in [McS82a, McS82d, SiM81a, SiM81b].

3.4.2 The Omega Routing Tag Scheme

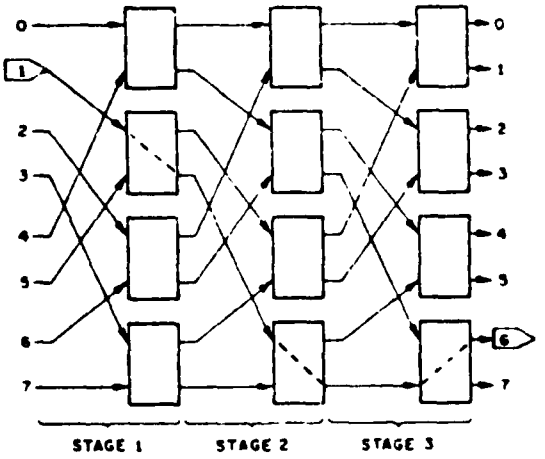
The routing tags defined by Lawrie in [Law75] are called destination tags. No computation is required on the part of the network users to generate the tag. The desired destination address, D, is itself the tag. Let $d_{n-1} \cdots d_1 d_0$ be the binary representation of D. The interchange box in stage i, $1 \leq i \leq n$, examines bit d_{n-i}. If $d_{n-i}=0$ the upper output is selected and if $d_{n-i}=1$, the lower output is selected. As an example, consider the path from input 6 to destination 1 in an Omega network of size N=8 (n=3), as shown in Figure 3.31(a). $D = d_2 d_1 d_0 = 001$; in stage 1, d_2 is examined and found to be 0 so the upper output is used. Similarly, in stages 2 and 3 the upper and lower outputs are used, respectively. If the user at destination 1 wants to send an acknowledgement or return message it must know the sender's address. Assuming that address 6 was transmitted with the message, D is set to 110 and the path shown in Figure 3.31(b) is established. The sequence traversed is lower, lower, upper. If the network is bidirectional, it can be readily verified that this scheme works in reverse (from output to input) as well.

3.4.3 The Extended Shuffle-Exchange Routing Tag Scheme

As was pointed out in Section 3.2.8, the extended shuffle-exchange network is identical to the Omega network in its topology and in the way inputs and outputs are numbered. Thus destination tags could be used to control it. However, Lang and Stone intended it to be used for routing permutation connections [LaS76]. To keep the control as simple as possible, they developed a scheme in which each input simultaneously enters a one bit tag. Each switching element combines the two control bits received using a Boolean operation (e.g. exclusive-or), sets its state according to the result, and passes the result on to the two switching elements in the next stage to which it is connected. The initial N bit input vector and the Boolean operator used define the overall permutation obtained. It was shown that the uniform shift



(a)



(b)

Figure 3.31 (a) Path from 6 to 1 in the Omega Network (N=8)
(b) Return Path from 1 to 6

permutations (input j, $0 \leq j < N$, is connected to output $j+s \bmod N$, s any integer) can be performed using the exclusive-or operation at the switching elements and an appropriate initial bit vector to determine s. Each input can calculate what its bit value should be knowing only the permutation to be performed. By replacing the exclusive-or with an equivalence operation at certain stages it was also shown that p-ordered vectors can be unscrambled (accessing of various vectors from a matrix stored in a skewed format [Law75]).

3.4.4 The Baseline Routing Tag Scheme

The Baseline network is defined to be bidirectional, with its I/O ports labeled Side 1 and Side 2 (see Figure 3.20) [WuF78]. Paths can be established from (1) Side 1 to Side 2, (2) Side 2 to Side 1, (3) Side 1 to Side 1, and (4) Side 2 to Side 2. The routing scheme used for cases (1) and (2) is exactly the same as Lawrie's destination tag scheme. The procedure for (3) and (4) is fairly complex and works as follows. Let $S = s_{n-1} \cdots s_1 s_0$ and $D = d_{n-1} \cdots d_1 d_0$ be the pair of I/O ports that want to communicate on, say, Side 1. Compute $C = c_{n-1} \cdots c_1 c_0$ as $S \oplus D$, the bitwise exclusive-or of S and D (i.e. $c_i = s_i \oplus d_i$, $0 \leq i \leq n-1$). If c_j is the most significant 1 in C then there are 2^j possible shortest paths between S and D with length $2(j+1)$. Next determine the set of addresses of the switching elements at which the path can reverse direction of travel, given by: $\{(z_{j-1}z_{j-2} \cdots z_0 s_{n-1}s_{n-2} \cdots s_{j+1})_j \mid z_i = 0 \text{ or } 1;\ 0 \leq i \leq j-1\}$. A member of this set is finally chosen using a conflict resolution procedure described in [WuF78].

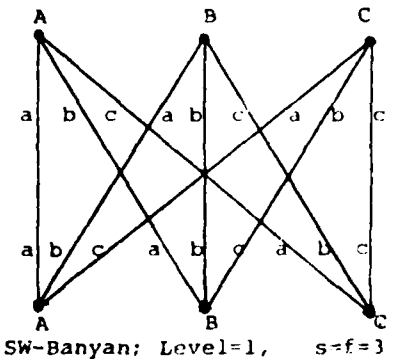
3.4.5 The Delta Class Routing Tag Scheme

The Delta network routing tag scheme is a generalization of Lawrie's destination tag scheme [Pat79]. For a $b^n x b^n$ network constructed from bxb crossbar switching elements, destination addresses are represented as base b numbers, i.e. $D = (d_{n-1} \cdots d_1 d_0)_b$. An example for b=3, n=2 is shown in Figure 3.22(a). Each of the switching element outputs is labeled from 0 to b−1. If the stages are labeled from n−1 to 0, input to output, then a switching element in stage i examines d_i. The requesting input is connected to the output labeled d_i. Clearly if b=2, the upper output is labeled 0 and the lower output is labeled 1 and this reduces to Lawrie's destination tag scheme.

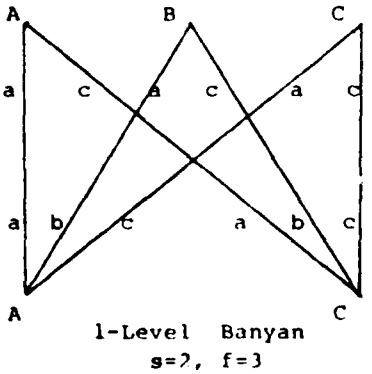
In the special case where $b=2^m$, the destination addresses can be represented in binary, as $b_{k-1} \cdots b_1 b_0$. Since m bits form a base b digit, the switching elements in stage i, $0 \leq i < k/m$, examine bits $b_{m(i+1)-1} \cdots b_{m*i+1} b_{m*i}$, which directly specify one of the outputs labeled 0 to 2^m-1.

3.4.6 The Regular SW-Banyan Routing Tag Scheme

The routing scheme for regular SW-banyans [TrL79] is slightly more general than the scheme just described for Delta class networks. For rectangular SW-banyans it is identical. To see this, examine Figure 3.32(a). An S=F=3 SW-banyan is shown with bases and apexes labeled A through C. The graph shown corresponds to a 3x3 crossbar. If the figure is rotated 90° clockwise and A=0, B=1, and C=2, a crossbar identical to those shown in Figure 3.22(a) is obtained. A two level S=F=3 SW-banyan is identical to the network in that figure. The routing tag described in [TrL79] is a destination tag and consists of sequences of letters instead of digits. It is represented as $D_L \cdots D_i \cdots D_1$ for an L level banyan, where D_i, $1 \leq i \leq L$, is a label. A node at level i chooses the node at the next level with label D_i. In an actual implementation, digits would be used instead of labels.



(a)



(b)

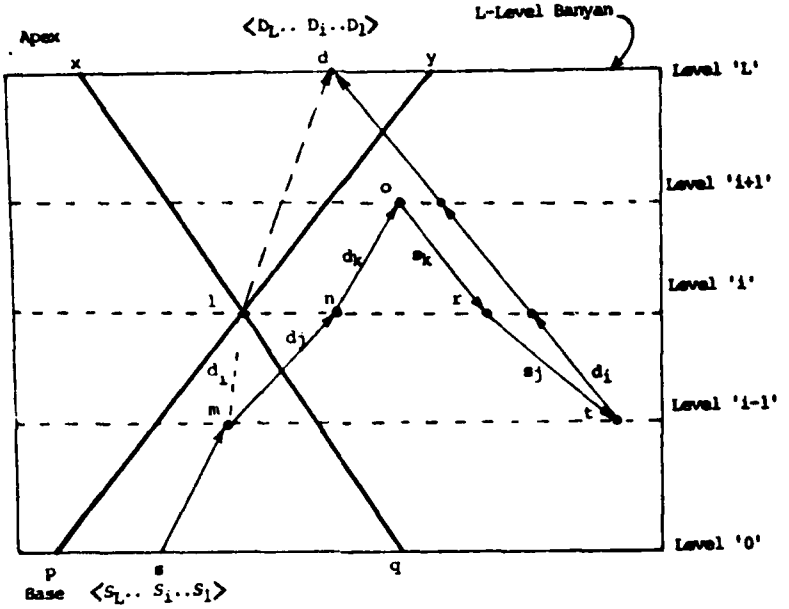
Figure 3.32 (a) Labeled S=F=3 SW-Banyan
(b) Labeled S=2, F=3, SW-Banyan
[TrL79]

The labelings for an S=2, F=3 regular SW-banyan are shown in Figure 3.32(b). This is the graph of a 2x3 crossbar. In this case, for connections from apex to base, the B label is invalid. The routing tags are the same, but not all labels lead to physical ports.

Rerouting to avoid faulty nodes is possible if backtracking is allowed. This is illustrated in Figure 3.33. Assume node l at level i has failed. This affects bases p through q when they want to communicate with any of apexes x through y (and vice versa). Suppose base $s = S_L \cdots S_i \cdots S_1$ wants to communicate with apex $d = D_L \cdots D_i \cdots D_1$, where S_i and D_i, $1 \leq i \leq L$, are labels as described above. Base s cannot go from node m to node l since node l is faulty. The scheme calls for the path to go through any node with a label other than D_i. From the new node n at level i, it must go to o at level i+1, turn around and go to node $r \neq n$ back at level i. It can then choose to go to any node in level i−1, say t. From t it can proceed directly to d as thought no reroute occurred. The values of m, l, n, o, r, and t are given in the figure. This rerouting procedure requires the traversal of four extra nodes.

3.4.7 The Generalized Cube Routing Tag Scheme

The routing tag scheme for the Generalized Cube network presented in [SiM81b] computes the (Hamming) distance between the input port number and desired output port number. Let S be the source address (input port number) and D be the destination address (output port number). Then the routing tag $T = S \oplus D$ (where "$\oplus$" means bitwise "exclusive-or"). Let $t_{n-1} \cdots t_1 t_0$ be the binary representation of T. An interchange box in the network at stage i examines t_i. If t_i=1, an exchange is performed and if t_i=0, the straight connection is used. If N=16, S=1011, and D=0110, then T=1101. The corresponding stage settings are exchange, exchange, straight, exchange. Because the exclusive-or operation is commutative, the incoming routing tag is the same as the return tag. Since the destination has the routing tag to the source,



Address of 'm' = $\langle S_L .. S_i, D_{i-1}, .. D_1 \rangle$

Address of 'l' = $\langle S_L .. D_i, D_{i-1}, .. D_1 \rangle$

Address of 'n' = $\langle S_L .. D_j, D_{i-1}, .. D_1 \rangle$

Address of 'o' = $\langle S_L .., D_k, D_j, D_{i-1}, .. D_1 \rangle$

Address of 'r' = $\langle S_L .., S_k, D_j, D_{i-1}, .. D_1 \rangle$

Address of 't' = $\langle S_L .., S_k, S_j, D_{i-1}, .. D_1 \rangle$

Figure 3.33 Rerouting in an SW-Banyan Network [TrL79]

it is easy to perform handshaking if desired. It can be readily verified that this scheme also works in creating a path from output to input. Thus it can be used in a bidirectional implementation and the inverse Generalized Cube.

This scheme can also be used in the Omega network. In the example in Figure 3.31(a), S=110, D=001, so T=111. The sequence traversed in that figure and in the return path in Figure 3.31(b) is exchange, exchange exchange. It should also be pointed out that the destination tag scheme could be used in the Generalized Cube network.

Consider an example of the use of the "exclusive-or" scheme in an SIMD environment. One interconnection function known to be admissible by the Generalized Cube network is a uniform shift of $+2^i$ mod N. That is, input port x is connected to output port $(x+2^i)$ mod N, where $0 \leq x < N$ and $0 \leq i < n$. Table 3.1 shows calculation of the routing tags and the pairings of tags used to set the state of each intechange box for N=8 and i=0 (i.e. a shift of +1 modulo 8). The rectangles around certain bits of each pair of tags indicate that those bits determine the state. Figure 3.34 shows the paths that will be taken through the network.

3.4.8 The ADM and IADM Routing Tag Scheme

3.4.8.1 MIMD Mode Communications. The routing tag scheme described here is used for both the ADM and IADM networks. All the properties to be discussed apply equally to both networks. This scheme can also be used in the Gamma network. In each network, a message can change its route at any stage. Since there are three possible paths that can be taken in each stage (except in stage n−1), $\log_2(3^{n-1} \cdot 2)$ is the theoretical lower bound on the number of bits required to represent any unique path through either network. The most general way to represent any unique path is with a *full routing tag* [SiM81a]. It is represented by $f_{2n-1}f_{2n-2} \cdots f_1f_0$. The high order n bits

Table 3.1 Permutation Function in the Generalized Cube Network. Source x is connected to destination x+1 mod 8. Boxes show bits paired in each stage; there are no conflicts.

Source	Destination	Routing Tag	Stage 2	Stage 1	Stage 0
000	001	001	[0] 0 1	0 [0] 1	0 0 [1]
001	010	011	[0] 0 1	0 [0] 1	1 1 [1]
010	011	001	[0] 1 1	0 [1] 1	0 0 [1]
011	100	111	[0] 1 1	1 [1] 1	0 1 [1]
100	101	001	[0] 0 1	0 [0] 1	0 0 [1]
101	110	011	[0] 0 1	0 [0] 1	1 1 [1]
110	111	001	[1] 1 1	0 [1] 1	0 0 [1]
111	000	111	[1] 1 1	1 [1] 1	0 1 [1]

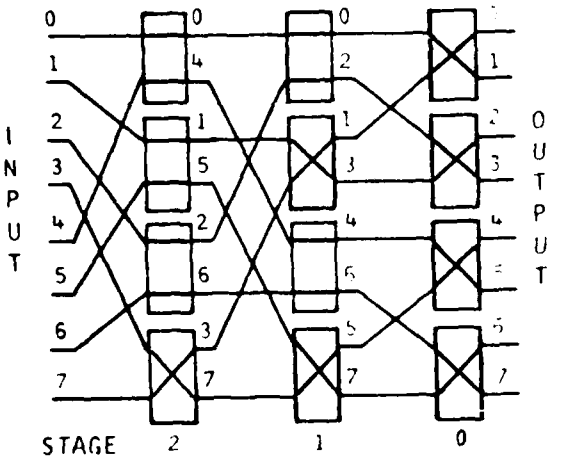


Figure 3.34 Uniform Shift of + 1 mod 8 in the Generalized Cube Network

of the 2n bit tag are sign bits and the low order n bits are magnitude bits. In stage i, if f_i=0, the straight link is used; if f_i=1 and f_{n+i}=0, the $+2^i$ link is used; and if f_i=1 and f_{n+i}=1, the -2^i link is used. The pair $f_{n+i}f_i$ corresponds to a signed digit as discussed in [PaR82].

Given a source address S and a full routing tag $F = f_{2n-1} \cdots f_1 f_0$, the destination address D is calculated as:

$$D = S + [(-1)^{f_{2n-1}} \cdot f_{n-1} \cdot 2^{n-1} + (-1)^{f_{2n-2}} \cdot f_{n-2} \cdot 2^{n-2} + \cdots + (-1)^{f_n} \cdot f_0 \cdot 2^0] \bmod N.$$

For example, in the ADM network, for N=16, if the source is 3 and the destination is 10, one possible value for F is 00101011. The path traversed is $+2^3$, straight, -2^1, $+2^0$. In Section 3.4.8.2, some methods for calculating full routing tags will be discussed.

If all the sign bits are the same, they can be collapsed into one bit. Thus the new tag only requires n+1 bits. Although all possible paths cannot be represented, a tag can be found to route a message between any source/destination pair. This is because there is more than one route between all source/destination pairs (source ≠ destination). To take advantage of the multiple routes, dynamic rerouting schemes can be employed [McS82d].

The n+1 bit routing tag scheme uses a relative addressing approach in which the information contained in the tag is the "distance" from the source to the destination, as opposed to the actual destination address. This scheme also provides a return tag which can be used if it is desired to send an acknowledgment.

Let $S = s_{n-1} \cdots s_1 s_0$ denote the source address and $D = d_{n-1} \cdots d_1 d_0$ denote the destination address, where s_i and d_i are the i^{th} bits of the respective unsigned addresses. An $n+1$ bit routing tag is formed by computing the *signed magnitude* difference between the destination and the source: $T = t_n t_{n-1} \cdots t_1 t_0 = D-S$. The *sign bit* is t_n where t_n=0 indicates positive and t_n=1 indicates negative. Bits $t_{n-1} \cdots t_1 t_0$ equal the

absolute value of D−S, the *magnitude* of the difference. A routing tag calculated in this manner is called a *natural routing tag.* In SIMD mode, if all N routing tags for a permutation are calculated in this way, then the permutation is said to be routed using *natural permutation routing tags.*

As an example, if N=16, S=1011, and D=0100, then T=10111. If the source and destination addresses are interchanged, the new tag is 00111. It is only necessary to complement the sign bit of an incoming tag to form a routing tag for a return or handshaking message.

To route a message through either network, stage i need only examine bits t_n and t_i in the routing tag. If t_i=0, the straight connection is used regardless of the value of t_n. If t_n=0 and t_i=1, the $+2^i$ link is used, and if t_n=1 and t_i=1, the -2^i link is used. In the previous example, with T=10111, if a message enters the ADM network at stage 3, the sequence of connections traversed from processor 11 to 4 is straight, -2^2, -2^1, -2^0. If a message enters the IADM network at stage 0, the sequence of connections is -2^0, -2^1, -2^2, straight. A route consisting of only straight or $+2^i$ connections is called *positive dominant* and a route consisting of only straight or -2^i connections is called *negative dominant.*

Given a source address S and a routing tag $T = t_n t_{n-1} \cdots t_0$, the value of the destination address D is calculated as:

$$D = [S+(-1)^{t_n}(t_{n-1}2^{n-1} + \cdots + t_0 2^0)] \bmod N.$$

Two tags, T_1 and T_2, are *equivalent* if and only if they route a message from the same source address to the same destination address; i.e., given $T_1(S) \rightarrow D_1$ and $T_2(S) \rightarrow D_2$, $T_1 \sim T_2$ if and only if $D_1 = D_2$.

A characteristic of the routing tag scheme is that for any arbitrary non-zero tag an equivalent routing tag can be computed that uses links of the opposite sign. The method for calculating equivalent tags is as follows. Let T′ denote the two's complement of T, then T′ ~ T (if T=0, T′ =T). To see this, let T_M denote the magnitude

bits of T. Also let t_n' be the sign bit of T'. For $T=t_nT_M$, $T\neq 0$, $T'=\bar{t}_nT_M'$. (Recall the two's complement of an n bit number T is evaluated by subtracting T from 2^n.) Assuming arithmetic is mod N,

$$T'(S) = S+(-1)^{t_n}T_M' = S-(-1)^{t_n}(2^n-T_M) = S+(-1)^{t_n}T_M = T(S).$$

For example, for N=16, if S=0110 and D=1101, then T=00111. The equivalent tag T' is 11001. In the ADM network, the first route is straight, $+2^2$, $+2^1$, $+2^0$. The equivalent route is -2^3, straight, straight, -2^0. A positive dominant tag can thus be converted to a negative dominant tag, and vice versa.

If the first node of the IADM network where a non-straight link is requested resides in stage i, if the $+2^i$ link is requested but blocked, then the -2^i link can be used, or vice versa. The equivalent tag can be formed and the message routed on the oppositely signed link. In the ADM network, if a straight link is blocked in the input stage, it can be avoided. As long as the low order n−1 bits of the tag are not all 0, the equivalent tag can be formed and the message sent on either non-straight link. Similarly, if a non-straight link at the input stage is blocked, the straight link can be used.

Return tags, to route from D to S, are formed by complementing the sign bit of the tag from S to D. Equivalent tags generate equivalent return tags.

Since the CC-banyan network shown in Figure 3.11 consists of $+2^i$ and straight links only, strictly positive dominant routing tags can be used to control it in a distributed fashion. Since all tags have the same sign, the sign bit, t_n, can be dropped from the tag.

In SIMD mode, a permutation is said to be routed using *positive dominant permutation routing tags* if those tags that are negative dominant in the set of natural permutation routing tags are converted to positive dominant. Similarly, a permutation is said to be routed using *negative dominant permutation routing tags* if those tags that are

positive dominant in the set of natural permutation routing tags are converted to negative dominant.

3.4.8.2 SIMD Mode Communications. In an SIMD environment, all the processors operate in lock-step and all active processors use the interconnection network simultaneously. There are two basic types of network settings: permutations, where each network input communicates with one network output, and broadcasts, where some network inputs are connected to multiple network outputs (but each output is connected to only one input). Here, the more common communication need in SIMD mode, that of permuting data, will be discussed.

First, the calculation of full routing tags for SIMD communications is examined. Then, some permutations performable by natural, positive dominant, and negative dominant permutation routing tags are described. Finally, the ability to perform a given permutation using different network settings is demonstrated.

Several permutation routing tag schemes were described previously in Section 3.4.8.1. A permutation is said to be *passable* by a network if the physical network structure (i.e., links and switches) allows the connections to be made. The use of full routing tags allows the ADM or IADM network to perform any passable permutation. The natural, positive dominant, and negative dominant permutation routing tags are limited in the permutations they can implement, but are, in general, more easily computed.

The full tags described in Section 3.4.8.1 are capable of representing any path in the network. A non-trivial problem however, is to find a way to calculate the tags for any ADM or IADM passable permutation so that the N paths specified do not conflict. Out of the N source/destination pairs in the permutation, a given pair may have many possible paths to choose from to complete the individual connection. Assuming the other N−1 paths have been specified, there will be only one path that does not conflict

with those already established. If the method used to calculate tags does not specify that particular path, the control scheme will not pass the permutation even though the network will.

If a permutation is performable by the ADM using natural (or positive dominant or negative dominant) permutation routing tags then the full routing tag can be easily generated. To convert a natural tag to a full tag it is only necessary to extend the sign bit, i.e., bits 0 to n−1 of the full are set to bits 0 to n−1 of the natural tag, and bits n to 2n−1 of the full tag are all set to the value of bit n+1 of the natural tag. The two's complement operation can still be used to produce an equivalent tag.

In [Sie79,SiS78] it was shown that the ADM can perform any permutation that the Generalized Cube network can. Similarly, the IADM can perform any permutation the inverse Generalized Cube can. This result can be used to generate full routing tags for either network based on the tags that would be used by the Generalized Cube network or its inverse. The way to compute the full routing tags from the Generalized Cube tags defined in Section 3.4.7 is given in Table 3.2, where s_i and d_i are the i^{th} bits of the source, S, and destination, D, and f_i and f_{n+i} are the i^{th} and $(n+i)^{th}$ bits of the full routing tag. To calculate the tag, set $f_{n-1} \cdots f_1 f_0$ to S OR D and $f_{2n-1} \cdots f_{n+1} f_n$ to $S\bar{D}$. For example, a full tag to establish a path in either network from 7=0111 to 11=1011, for N=16, is F = 01001100.

Permutations that are passable by the ADM (or IADM), but cannot be specified using natural permutation routing tags or as Generalized Cube (or its inverse) permutations are more difficult to handle. If it is known at compile time that such a permutation of data must be performed (and the permutation itself is known), the full routing tags can be precomputed and the execution of the SIMD algorithm will not be impeded. However, if it is necessary to perform an arbitrary passable permutation that is determined at execution time, this method can not be used. The efficient execution time computation of full routing tags to properly set the network to perform passable

Table 3.2 Full Routing Tag Settings for Performing a Generalized Cube Passable Permutation

s_i	d_i	Route	f_{n+i}	f_i
0	0	Straight	0	0
0	1	$+2^i$	0	1
1	0	-2^i	1	1
1	1	Straight	0	0

permutations is currently an open problem.

Now consider the capabilities of the natural, positive dominant, and negative dominant permutation routing tag schemes. Recall that these schemes can set the ADM to perform only a subset of the ADM passable permutations. However, they are easy to compute and require only $n+1$ bits each.

In [Len78], different classes of permutations known to be important in SIMD processing were defined. It was shown in [McA80] that positive or negative dominant permutation routing tags can correctly specify the connections required by two of these permutation classes. The first class is called the lambda permutation. Source address X is connected to destination address $jX+k \bmod N$, where j is an odd, positive integer, k is any integer and $0 \leq X < N$. The other permutation class is called delta and is comprised of uniform shifts in groups of 2^i where $0 < i \leq n$. This permutation can be thought of as the concatenation of 2^{n-i} networks of size 2^i in which the same uniform shift (mod 2^i) is being performed in each. It was also shown in [McA80] that natural permutation routing tags correctly specify the connections required to perform the perfect shuffle [Sto71] permutation (defined in Section 3.2.7) in the ADM network and the inverse perfect shuffle in the IADM network. Algorithms that can be used at compile time to determine if a given permutation is performable with natural, positive dominant, or negative dominant permutation routing tags were also presented.

An important property of the ADM and IADM networks is the existence of multiple paths between all non-trivial (i.e. source≠destination) source/destination pairs. Because of the nature of the multiple paths, there are many permutation connections that can be established in more than one way. The same pairing of inputs to outputs is achieved, but all messages (data items) have more than one path available to them. The constraint imposed is that all messages take paths that are mutually compatible (i.e. there must be no conflicts). Exploiting this property leads to some fault tolerance.

As an example, consider the uniform shift permutation defined earlier as connecting input X to the output whose address is X+k mod N, where k is some constant. This is shown in Figure 3.35 where k= +2 and N=16. For each input, the path taken consists of the straight, straight, $+2^1$, straight connections. As shown in Figure 3.36, the same uniform shift of +2 can be achieved if all paths consist of straight, $+2^2$, -2^1, straight connections. Other combinations are possible.

Intuitively, the "weights" of the links in a path can be summed, where $+2^i$ and -2^i links have weights $+2^i$ and -2^i, respectively, and straight links have a weight of 0. Thus, any set of paths for which the weights sum to +k mod N can be used to perform the uniform shift permutations. When a different network configuration can be used to perform a given permutation it is said that *redundant control settings* exist for that permutation.

For the example given above, if any (or all) of the straight links in stage 2 or any of the $+2^1$ links in stage 1 are faulty, the connections shown in Figure 3.36 can be used to avoid them. There is another configuration that can be used if any of the straight links in stage 3 are faulty (i.e., $+2^3$, -2^2, -2^1, straight). In this example, there is no way to avoid using the straight links in stage 0. If 2^0 was added to or subtracted from the sum of the weights, there would be no way to adjust the other weight values (i.e. change the path) to produce a total of +2. In general, if one path from an input (source) S to an output (destination) D has straight conditions at stages 0,1, $\cdots$,i, then all paths from S to D must have stages 0 to i set to straight. Further information about redundant control settings for permutations is in [SiS79].

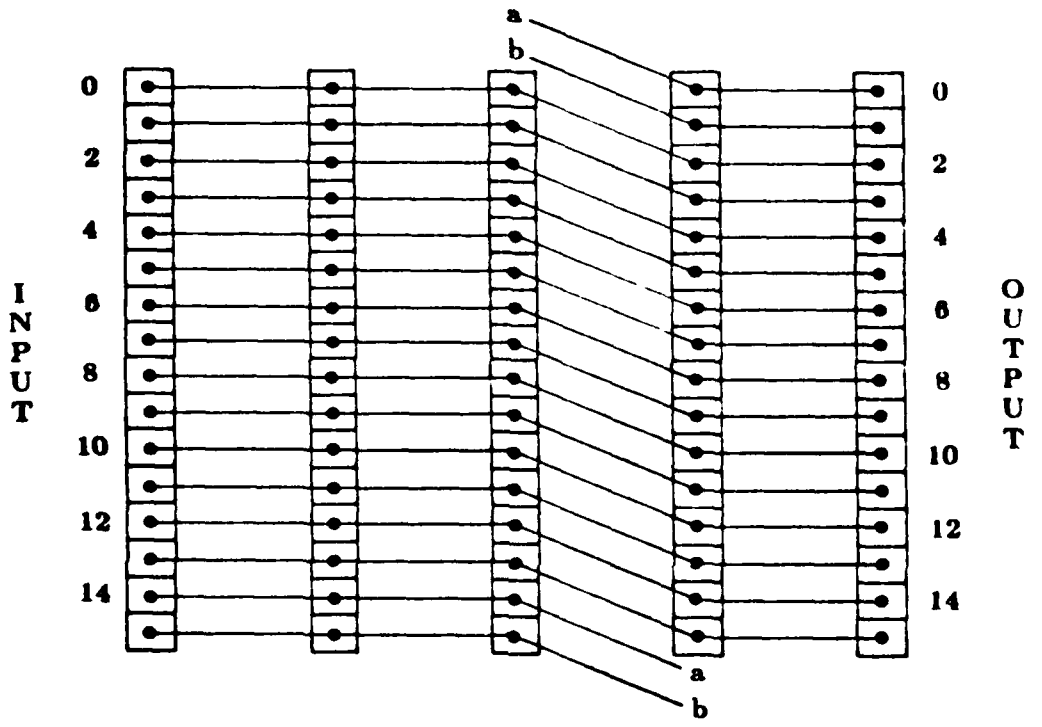


Figure 3.35 Uniform Shift of + 2 mod 16 in the ADM Network (N=16)

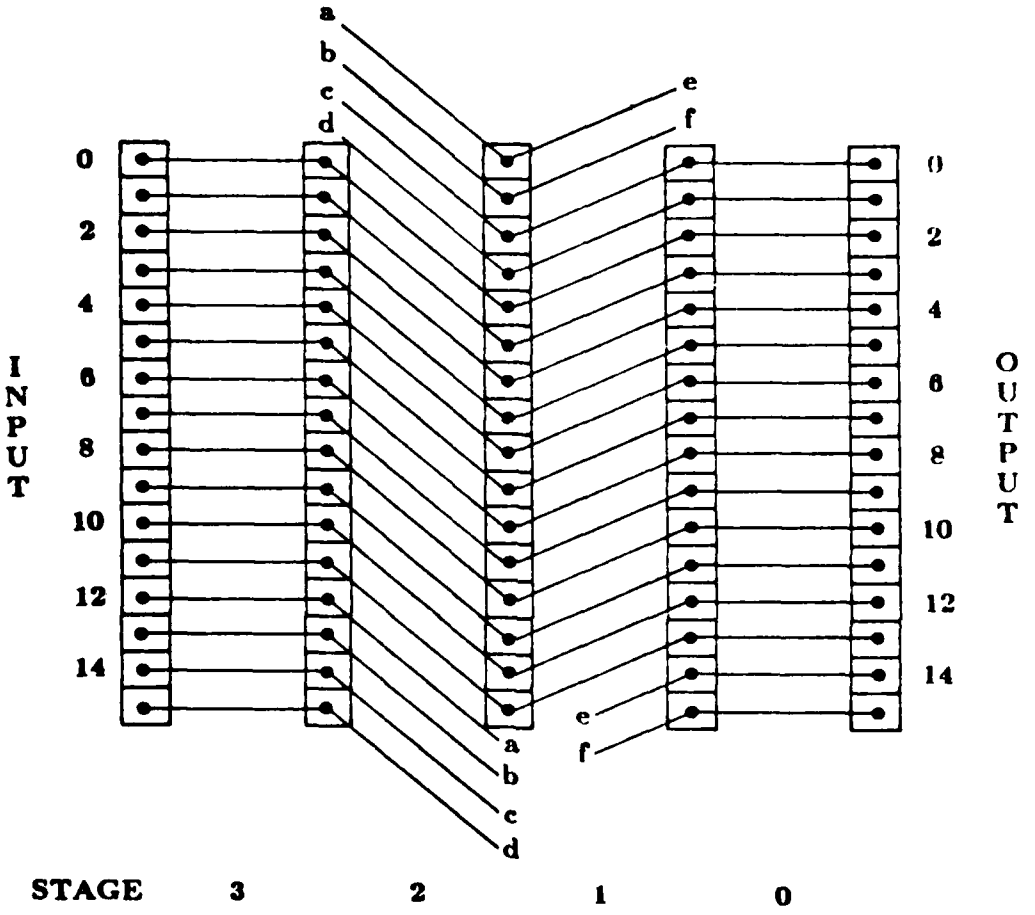


Figure 3.36 Equivalent Routing for the Uniform Shift of +2 mod 16 in the ADM Network (N=16)

3.4.9 Conclusions

The easiest way to fully distribute the control of a multistage interconnection network appears to be to use routing tags (it is certainly the most common). Each network user calculates the tags it will use, and each switching element determines its state according to the tags only it receives. The majority of the routing schemes discussed incorporate destination tags. The tag is simply the desired destination address; $n=\log_2 N$ bits are required for a network with N destination ports. Because of the highly structured nature of the networks that use them, each switching element can determine its state based on a subset of the tag bits. The networks that were discussed that use destination tags are the Omega, baseline, Delta, and regular SW-banyans (which means that networks not discussed, but in this class, can use destination tags).

The advantages to destination tags are: (1) that no computation is required to generate them (once the destination is known); and (2) misrouting can be detected by comparing the tag to the physical address where it arrives. The disadvantage is that for handshaking or sending return messages, the source address must accompany each message.

A different approach to routing tags is to compute the "distance" to be traversed. This was proposed for the Generalized Cube network where the Hamming distance was used. This tag also requires only n bits. A "distance" tag was also proposed for the ADM and IADM networks. In this case, the arithmetic distance is computed and represented in sign-magnitude form, which requires $n+1$ bits.

The distance tags are trivial to calculate, requiring an exclusive-or operation (for the Generalized Cube) or a subtraction (for the ADM/IADM), so this is not a major disadvantage. The advantage to this approach is that the tag contains all the information needed to determine the source address. For a return message, the Generalized Cube tag can be used without modification and the ADM/IADM tag can be used after complementing a single bit (the sign bit). The main disadvantage is that routing errors

cannot be directly detected. To add this capability requires sending the destination address with each message.

For the cube type networks, either the destination tag or the distance tag approach can be used. They are equally powerful and require the same number of bits. The choice depends upon specific system requirements. For the data manipulator type networks the distance tag approach is preferable. It can be shown that destination tags could be used. However, each switching element would require enough intelligence to compare the tag to its level in the network and then decide how to reduce the tag's distance from the desired destination. Considerably more logic would be required.

The scheme discussed for the extended shuffle-exchange network is interesting because it only requires one bit tags. However, it is special purpose, being designed for certain permutation connections in SIMD mode. Thus it is not generally suited to use in other non-functionally equivalent networks.

3.5 Fault Tolerant Designs

3.5.1 Introduction

There are two major aspects to fault tolerance: diagnosing faults and avoiding known faults (if such a capability exists). The literature on fault diagnosis of interconnection networks will be briefly summarized in the following and then some fault tolerant multistage network designs will be examined in more detail.

General multistage network fault diagnosis is discussed in [NaS80]. A method for diagnosing faults in the Benes network is presented in [OpT71b]. As discussed in [SoR80], certain kinds of faults can be tolerated in all stages of the Benes network except the center stage. A way to add an extra switching element to make the network tolerate any single fault is shown and will be described in Section 3.5.3.

Detecting and locating faults in the Baseline network are discussed in [WuF79b] and one of the results is extended in [FeK82]. The procedures described are generally applicable to the cube type networks. The method used is to generate test patterns that are propagated through the network. The emerging patterns are compared to precomputed, expected patterns. This requires no extra hardware in the network. Using a different approach, in [RaM80] four methods are described for diagnosing SW-banyan networks. Extra hardware is required and it is assumed that the switching elements can diagnose themselves and set a latch if faulty. It is claimed that two of the methods can be used for any multistage network.

In [FaP81] the diagnosis of multistage cube type networks is discussed. The interesting aspect of this work is that it is assumed that one whole stage of the network is implemented on one VLSI integrated circuit chip. Due to pin constraints, this is only realistic for small bit sliced networks. For example, a 32x32 network with bit slices one bit wide requires each chip to have 64 pins for data paths alone. Since control lines require two pins per data path, the count rises to 192. Systems with 64x64 networks and 60 bit path widths, built from discrete components, are being constructed [McS82a]. If they were built with one switching element per chip, they would require 7940 pin chips (including power, ground, clock, and reset)!

Most of the diagnosis studies implicitly assume the network is circuit switched (because of assuming the fault model in [LeG68]). However, diagnosing cube type packet switching networks is addressed in [Lim82]. In a more theoretical vein, a graph model is used in [MaM81] to determine the necessary and sufficient conditions for being able to diagnose t faults. An optimal assignment for a t-diagnosable network is presented. In [Agr82], methods for diagnosing multistage networks are surveyed.

In [ShH80] an interesting approach to fault tolerance analysis is taken. β-networks are defined as any interconnection network constructed from β-elements (after Joel [Joe68]), i.e. 2x2 crossbar switching elements. A β-network is considered fault tolerant

if any pair of I/O ports can communicate after a finite (but arbitrary) number of passes through the network. This is a much less restrictive definition than the usual, which requires communication to be possible in one pass. Several simple structures were analyzed in [ShH80]. The work was extended in [She82], where the shuffle-exchange, indirect binary n-cube, Benes, and double-tree (to be discussed in Section 3.5.2) networks were analyzed.

A 12 node network is shown in Figure 3.37, where a processor and memory is presumed to reside at each node. This structure was proposed in [Pra81] for parallel processing and was shown to be able to tolerate one arbitrary node or link failure. In [PrR81], the topology, routing tag schemes and fault diagnosis of similar structures are discussed and analyzed. Consideration is given to minimizing the maximum path length and keeping the interconnection complexity low. This kind of structure is well suited to a number of parallel processing problems. It does not, however, have the high bandwidth required by a number of large-scale systems such as PASM [SiS81], PUMPS [BrF82], the Ballistic Missile Defense (BMD) Agency test bed [McS82a], Burrough's Flow Model Processor [BaL81, Bur79], TRAC [SeU80], HEP [Smi78, Smi81a], and STARAN [Bat74]. Thus it is important to investigate adding fault tolerance to multistage interconnection networks, which do have the requisite bandwidth.

In the remainder of this section, a number of fault tolerant multistage interconnection networks that have been proposed are surveyed. Included is early work done on making permutation networks (full access networks) fault tolerant by adding a repair network to the output [LeG68]. A much less expensive approach discussed is the addition of a single extra switching element to the Benes network [SoR80]. A different kind of fault tolerance is achieved by adding an extra stage of switching elements to the network input (or output). This has been considered for the Generalized Cube network [AdS82a,AdS82b,SiM81b] and the baseline network [WuL82]. A different approach, also examined, is the inclusion of extra links between stages. This has been proposed

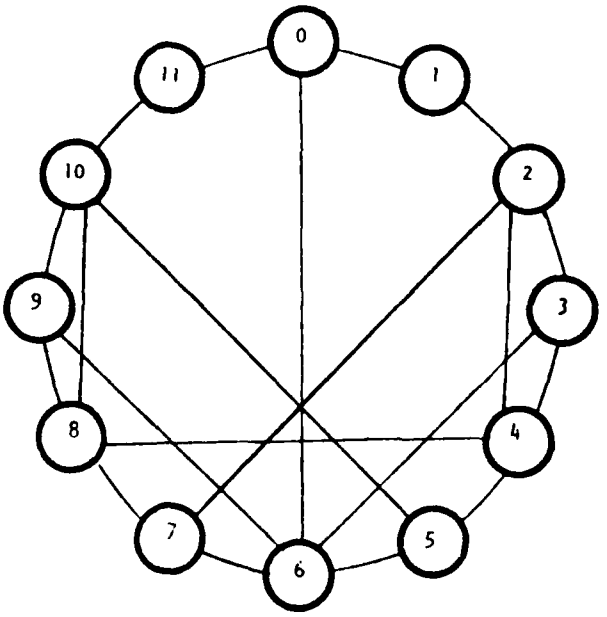


Figure 3.37 A One-Fault-Tolerant Interconnection Structure (N=12) [Pra81]

for the IADM network [McS82b] and for a cube based network [CiS82]. Finally, a novel scheme for the Omega network is described that uses error correcting codes [LiL82].

3.5.2 The Double-Tree Network

The double-tree network shown in Figure 3.38 for N=8 is designed to exchange any pair of inputs. It was proposed in [LeG68], for addition to any multistage permuting network constructed from 2x2 switching elements. Its intended use is to correct permuting errors. It was shown that any switching element in the multistage network stuck in the "straight" or "exchange" state has the potential effect of exchanging two outputs. This happens if the stuck-at-state is opposite to the needed state in that switch. Their justification for the "stuck-at" fault model was based on analyzing the failure modes of the switching element shown in Figure 3.26. If the double-tree network is appended to the output of the faulty network, then the exchanged outputs can be exchanged again, and corrected.

The double-tree network performs an arbitrary exchange in the following way. Any pair of inputs can be directed to one of the switching elements in the left half (shown by dashed lines in the figure) of the network. At the switch where they meet, they can be exchanged. The switch settings required to do this are then copied by reflection (about an imaginary vertical line through the switch labeled b_x) to the switches in the right half of the network, except for the switch that corresponds to the (left half) switch effecting the exchange. It is set to the opposite state. If the switch labeled b_x is where the input pair meets, it is set to exchange and all settings in the left half are mirrored to the right half. This kind of a network can be constructed for any value of N by "pruning" a corresponding double tree network for the next largest power of two greater than N.

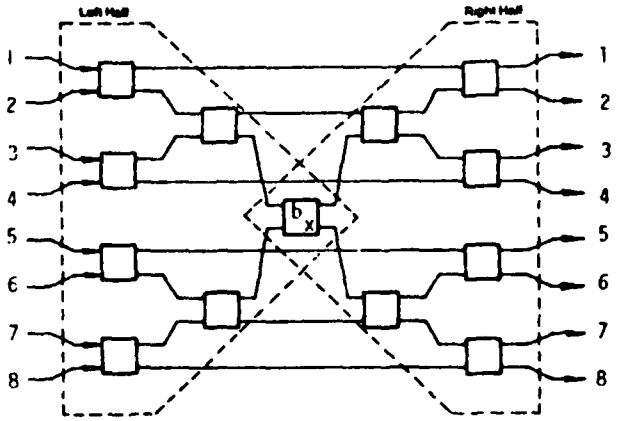


Figure 3.38 8x8 Double-Tree Network [She82]

It was shown in [LeG68] that when a double-tree network is combined with a full permutation network (e.g. the Benes̆ network), the composite network can accommodate a single stuck-at fault in any of the switching elements. It was also shown that this capability holds when the output column of the double-tree section is removed. The double-tree network that results from removing the output column is called the truncated double-tree (TDT). Finally, if p faults can be decomposed into separate pairwise exchanges, then a cascade of p TDT networks added to the permutation network can correct the faults.

It is important to note that this scheme is only designed to handle control line faults. If any fault occurs that alters data passing through the network, it cannot be corrected. This is because all switching elements participate in routing data and none can be avoided.

3.5.3 The Fault Tolerant Benes̆ Network

After analyzing the Benes̆ network, it was found in [SoR80] that the network can accommodate most single stuck-at faults (as defined in Section 3.5.2). This is because there are multiple paths between each input/output pair. It was shown that there are some permutations that cannot be performed under stuck-at faults anywhere in the center stage of the network. For example, any stuck-at exchange fault in the center stage prevents the identity permutation (input j to output j, $0 \leq j < N$) from being performed. Any stuck-at-straight fault in the center stage prevents a uniform shift of $+N/2$ mod N from being performed. It turns out that any center stage faults can be corrected with the addition of a single 2x2 switching element at the output of or input to the network. One connection to the switching element must come from a line labeled between 0 and $(N/2)-1$ and the other from a line labeled between $N/2$ and $N-1$. An example with the extra switching element at the output connected to 0 and 4 is shown in Figure 3.39 for $N=8$.

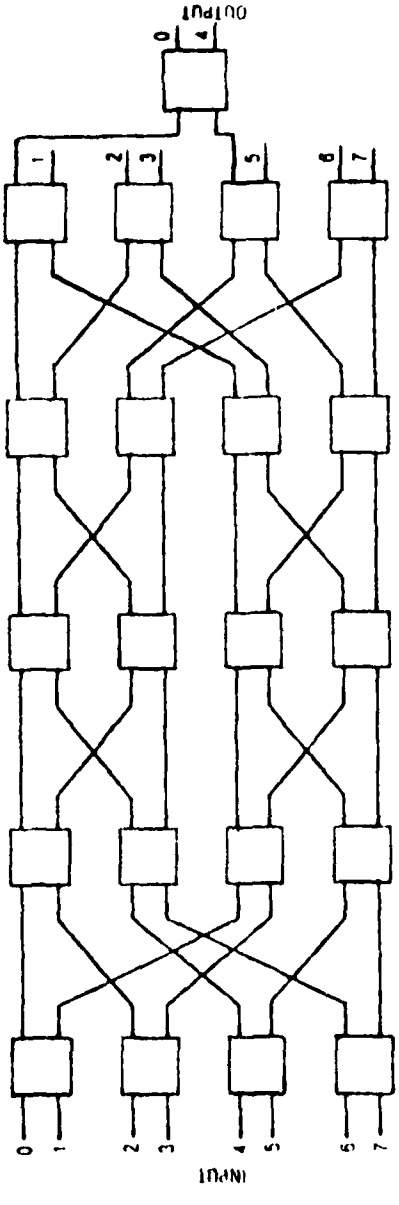


Figure 3.39 Fault-Tolerant Benes Network. A single switching element stuck at straight or exchange can be accommodated. [Agr82]

3.5.4 The Extra Stage Cube Network

The extra stage cube (ESC) is formed by adding an extra stage along with a number of multiplexers and demultiplexers to the Generalized Cube network [AdS82a,AdS82b,SiM81b]. Its structure is shown in Figure 3.40 for N=8. The extra stage, labeled stage n, is placed on the input side of the network and implements the $cube_0$ interconnection function. Thus, there are two stages in the ECS that can perform $cube_0$.

Stage n and stage 0 can each be enabled or disabled (bypassed). A stage is *enabled* when its interchange boxes are used to provide interconnection. It is *disabled* when its interchange boxes are being bypassed. Enabling and disabling in stages n and 0 is accomplished with a demultiplexer at each box input and a multiplexer at each output. Figure 3.41 shows an interchange box from stage n or 0 in detail. One demultiplexer output is connected to a box input, the other to one input of the corresponding multiplexer. The remaining multiplexer input is connected to the matching box output. The demultiplexer and multiplexer are configured such that they are either both connected to the interchange box (enabling it) or both disconnected from it, thereby shunting it (disabling it). All demultiplexers and multiplexers for stage n share a common control signal, as well as those for stage 0.

Stage enabling and disabling is performed by a system control unit. Normally, the network is set so that stage n is disabled and stage 0 is enabled. The resulting structure is that of the Generalized Cube network. If after performing fault detection and location tests a fault is found, the network is reconfigured. If the fault is in stage 0 then stage n is enabled and stage 0 is disabled. For a fault in a link or box in stages n−1 to 1, both stages n and 0 are enabled. A fault in stage n requires no change in the network configuration; stage n remains disabled. If a fault occurs in stages n−1 through 1, in addition to reconfiguring the network, the system informs each source device of the fault by sending it a fault identifier. The ESC can thus tolerate any single failure

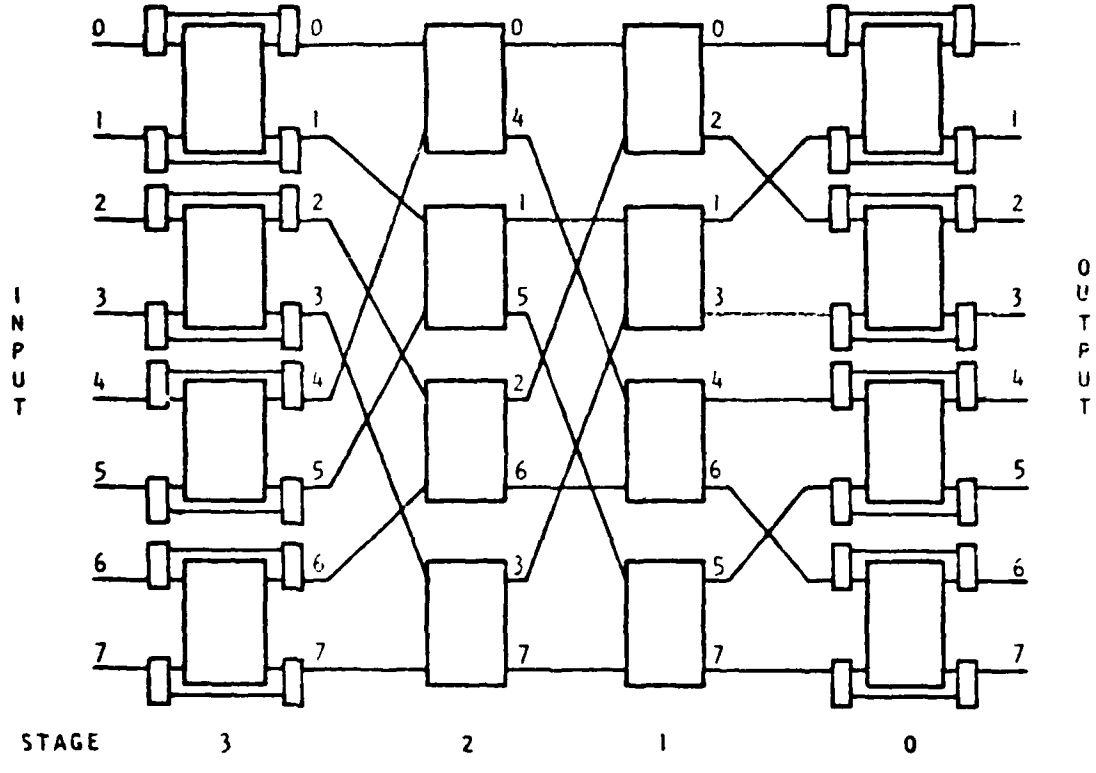


Figure 3.40 8x8 Extra Stage Cube Network. A single link or switching element (interchange box) failure can be tolerated.

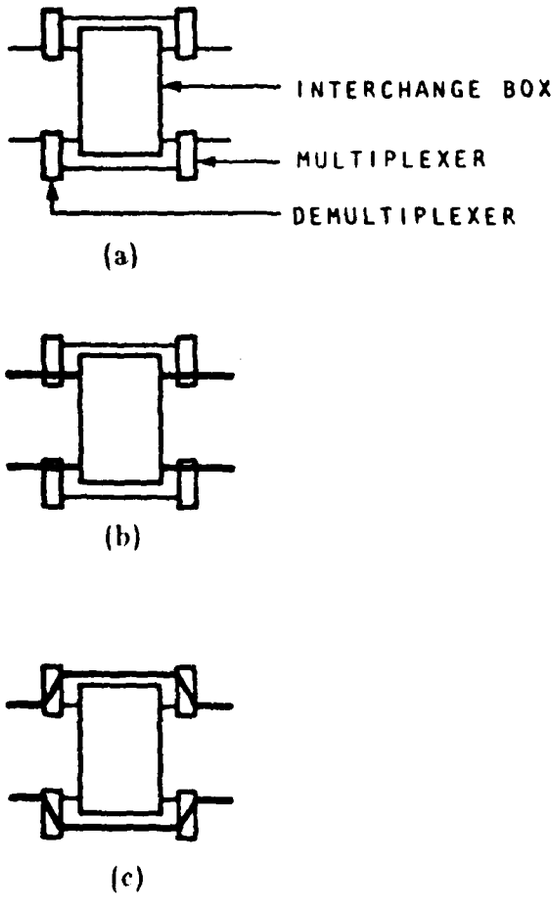


Figure 3.41 (a) Detail of Interchange Box. The multiplexer and demultiplexer are for enabling and disabling.
(b) Interchange Box Enabled
(c) Interchange Box Disabled

in the network. However, this applies only to MIMD mode communication. For the permutation connections required in SIMD mode, it is not one pass fault tolerant (two passes are required).

As discussed in Section 3.2.11, the Generalized Cube network, and therefore the ESC, compare addresses in stage i that differ in the i^{th} bit. Viewed a different way, this means that stage i, $0 \leq i < n$, determines the i^{th} bit of the output port address to which the path is to be established. Consider the path from source $S = s_{n-1} \cdots s_1 s_0$ to destination $D = d_{n-1} \cdots d_1 d_0$. If the route passes through stage i using the straight connection then the i^{th} bit of the source and destination addresses will be the same, i.e., $d_i = s_i$. If the exchange setting is used, the i^{th} bits will be complementary, i.e., $d_i = \bar{s}_i$. In the Generalized Cube, stage 0 determines the 0^{th} bit position of the destination is a similar fashion. In the ESC, however, both stage n and stage 0 can affect the 0^{th} bit of the output address. Using the straight connection in stage n performs routings as they occur in the Generalized Cube. The exchange setting makes available an alternate route not present in the Generalized Cube. In particular, the route enters stage n−1 at label $s_{n-1} \cdots s_1 \bar{s}_0$, instead of $s_{n-1} \cdots s_1 s_0$.

A related network is the extra stage baseline used in the Starnet system [WuL82]. It is shown in Figure 3.42 for N=8. Though it can tolerate single faults in the middle stages, since the input and output stages cannot be bypassed, it cannot tolerate arbitrary single faults. If it is modified by adding multiplexers and demultiplexers as is done in the ESC, it would have the same capabilities.

3.5.5 The F-Network

The F-network was proposed in [CiS82] and is shown in Figure 3.43 for N=8. It consists of n+1 columns of N switching elements each. A switching element selects one of four inputs to be connected to one of four outputs (or multiple outputs for broadcasting). The input column uses 1x4 switching elements and the output column uses

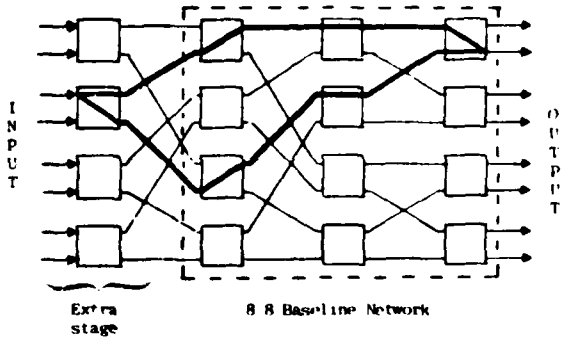


Figure 3.42 An 8x8 Baseline Network with Extra Stage. (The network cannot tolerate all arbitrary single faults.) [WuL82]

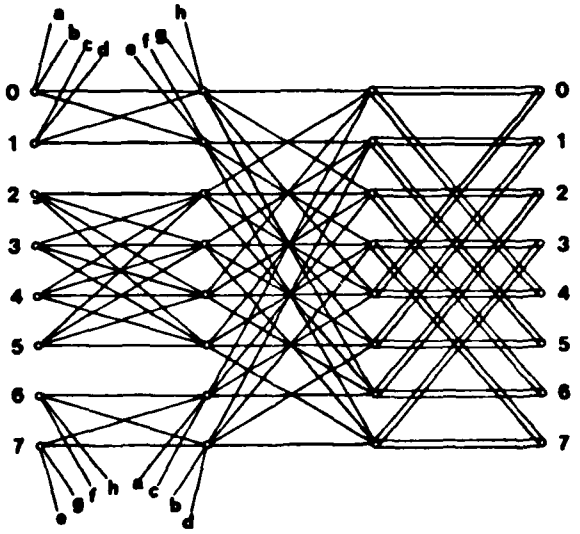


Figure 3.43 8x8 F-Network
[CiS82]

4x1 switching elements. Assume the columns are numbered from 0 to n, input to output. If $P_i = (p_{n-1} \cdots p_1 p_0)_i$ is the level of a switching element in column i, then P_i is connected to

$$P_{i+1} = (p_{n-1} \cdots p_{i+1} p_i p_{i-1} \cdots p_0)_{i+1},$$

$$Q_{i+1} = (p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_0)_{i+1},$$

$$R_{i+1} = (\bar{p}_{n-1} \cdots \bar{p}_{i+1} p_i p_{i-1} \cdots p_0)_{i+1}, \text{ and}$$

$$S_{i+1} = (\bar{p}_{n-1} \cdots \bar{p}_{i+1} \bar{p}_i p_{i-1} \cdots p_0)_{i+1}$$

in column i+1. Notice that p_{i+1} and Q_{i+1} are the choices available at the input to an interchange box in stage i of the Generalized Cube network. In the F-network the cube functions are ordered from $cube_0$ to $cube_{n-1}$, the reverse of the Generalized Cube. Since two other functions are available at each stage, the F-network's capabilities are a superset of the inverse Generalized Cube's.

It was shown in [CiS82] that at each column there are always two path choices available. Hence, the F-network can tolerate the failure of any single link or any switching element in columns 1 through n−1. If a switching element in column 0 fails, one input is isolated and in column n, one output is isolated. The network is very robust under multiple faults.

3.5.6 The IADM Network with Half-Links

It has been pointed out (see Sections 3.2.12 and 3.4.8.2) that the IADM network has multiple paths between nonequal source/destination pairs. Dynamic rerouting schemes have been developed to take advantage of that fact [McS82d]. The one drawback to dynamic rerouting is that it is not always possible to change paths. Specifically, this is the case when a straight link is required. To solve this problem, a scheme is presented in [McS82b] in which extra links are added to the IADM network. The links, called *half-links*, are added to each of stages 1 through n−1 (refer to Figure 3.19). At level j, $0 \leq j < N$, in stage i, the half links connect switching element j to switches $(j+2^{i-1})$ mod N and $(j-2^{i-1})$ mod N. The name half-link comes from the fact that $2^{i-1} = (1/2)2^i$, i.e., these links move routing tags half the distance of existing non-straight links.

The addition of half-links is motivated by the desire to route around a busy or faulty straight link. For example, suppose a tag wishes to route straight through stages i and i+1. Assume the straight link in stage i is unavailable. The tag can route $+2^i$ in stage i and then -2 in stage i+1. The net result is the same but the straight link in stage i is avoided.

It can be shown that, due to the additional links, a message in stage i, $0 \leq i < n-1$, can always route on either the $+2^i$, straight, or -2^i link if it is currently on a positive or negative dominant path. Otherwise it can always take the $+2^{i-1}$ or -2^{i-1} link (either half-link). As an example, examine Figure 3.44. All possible paths in the IADM network with half-links between source 9 and destination 31 are shown. The positive dominant routing tag associated with the 9 to 31 connection is T=010110. The positive and negative dominant paths are shown as solid lines. The alternate paths that are normally available in the IADM network are shown as dotted lines. Finally, the dashed lines indicate all the paths that are now available due to the inclusion of half-links. Examining the figure, it can be observed that there are three path

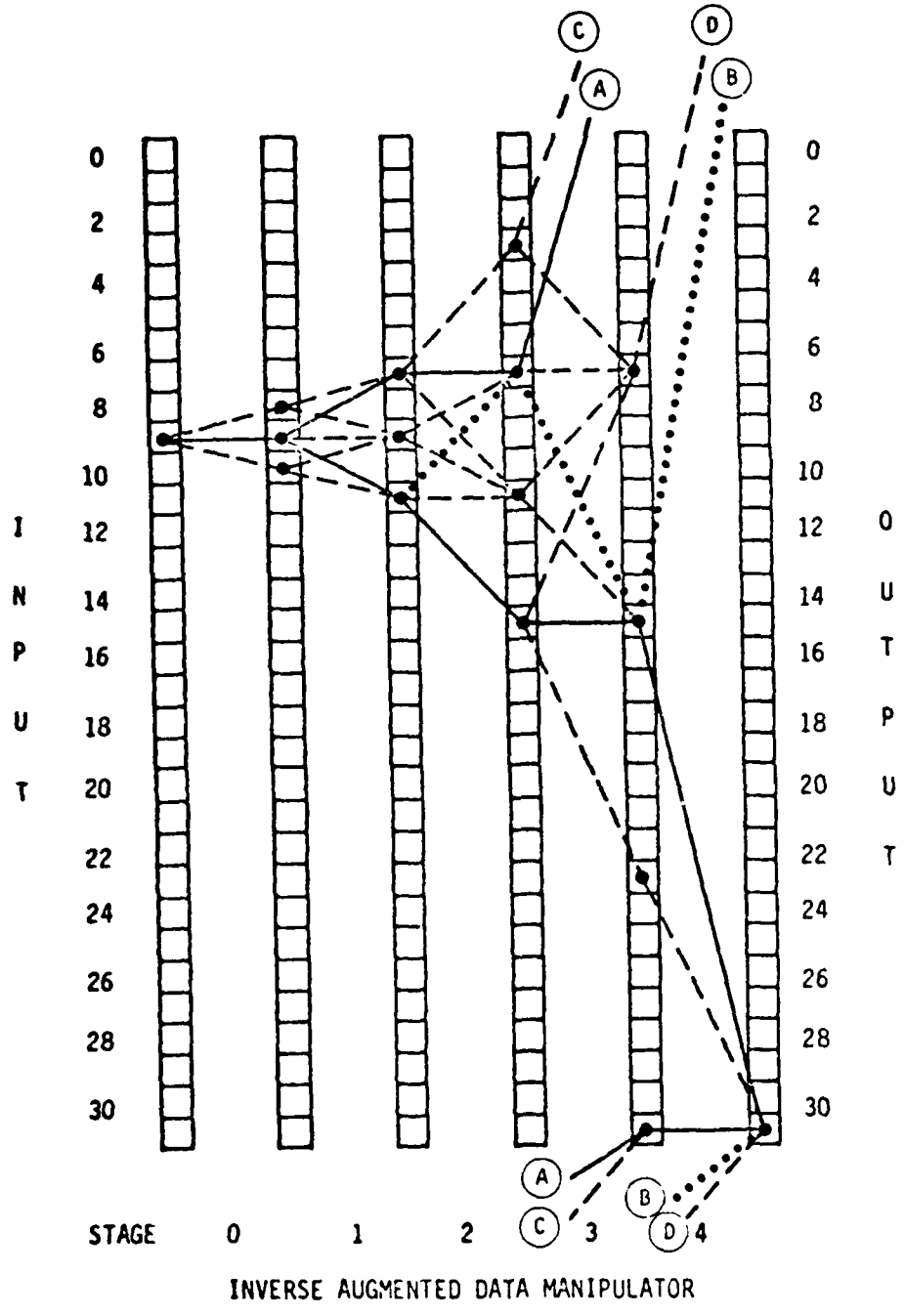


Figure 3.44: All Possible Paths Between Source 9 and Destination 31 in the IADM Network with Half-Links (N=32). T = 010110. Solid lines - positive and negative dominant paths; dotted lines - alternate paths available without half-links; dashed lines - paths available due to inclusion of half links.

choices at each switching element intersected by a solid line in stages 0 through n−2. Everywhere else there are two choices. There is always exactly one choice in stage n−1 (stage 4 in the figure) since the paths must converge here (the $+2^{n-1}$ and -2^{n-1} paths are connected to the same output and so are considered non-distinct paths).

To make this network fault tolerant, a fault look-ahead mechanism must be incorporated into each switching element. A separate latch associated with each of the five output links is included in every switch. If the outgoing link or switch to which it is connected is faulty, the corresponding latch is set to 1. Otherwise the latch is 0. The packet switching protocal (see Chapter 2, Section 2.5.2) must be modified so that any packet requesting this switch will be denied access if it is going to use a link whose fault latch is set to 1 (this can be determined from the packet's routing tag). With this scheme, it is shown in [McS82b] that the network can tolerate any *two* faults in links or switching elements (excluding the input and output columns which form the "hard-core").

To reduce the amount of hardware required to implement the IADM network with half-links, three options are available but with some sacrifice in fault tolerance. The first option is to remove stage 0, the second is to eliminate straight links in stages 0 through n−2, and the third is to do both. All of these modifications produce networks that can tolerate one arbitrary fault (again excluding input and output columns). Differences between the options result in variations in throughput.

A routing tag scheme is presented in [McS82b] for controlling the IADM with half-links and any of the networks produced by exercising the hardware reduction options. It is an extension of the scheme discussed in Section 3.4.8. It should be noted that modifying the ADM network according to the methods described in this section does *not* produce a fault tolerant network.

3.5.7 The Error Correction Coded Omega Network

A novel approach to designing a fault tolerant Omega network is proposed in [LiL82]. A wide path width is assumed and all data and routing tags are encoded with an error correcting code (e.g. a Hamming code). To see how the scheme works, visualize the network as being three dimensional. The x dimension is the direction of information flow, the y dimension is that of the inputs, and the z dimension is that of the parallel information bits. A network is normally viewed in the x-y plane. It is assumed that due to VLSI pin limitations, the switching elements are bit sliced, s bits per switch. If the network path width is w, there are w/s chips per switching element. There are also w/s NxN x-y switching planes. The trick then, is to encode groups of bits with an error correcting code and then to route each bit from one group through a different x-y plane. In this way, if a single chip fails, only one bit error is generated in each coded group and each can be corrected. Also, link failures can be easily tolerated. To avoid routing errors, the routing tag is encoded and any errors are corrected at each stage. In addition, the control section needs to generate three sets of control signals independently. Each switching element then votes to determine its state. Thus, no single error can cause a mis-route. A major advantage to this scheme is that it can handle spurious errors as well as hard failures (e.g. a line stuck at logical 1).

This scheme can be applied to any of the multistage networks that have been discussed. It works equally well for SIMD mode permutation connections and MIMD mode random connections.

3.5.8. Conclusions

To place these fault tolerant multistage interconnection networks into perspective, the family tree in Figure 3.25 is completed. The remainder of the tree headed Fault Tolerant Networks is shown in Figure 3.45. Two kinds of fault tolerance are considered: (1) SIMD mode, where only routing errors are dealt with (but are done so in one pass) and link or switching element failures that corrupt data cannot be tolerated; and (2) MIMD mode, where multiple paths are provided so that faulty links and switches can be avoided. Networks in the second category can be used in SIMD mode, but two passes are required when a fault is present.

The earliest approach to providing SIMD mode fault tolerance was to add a double-tree repair network to a permuting network. This network can be added to any of the cube type networks as well as the Benes̆ network. It restores their fault free permuting abilities in the presence of permanent routing errors. Another approach in this category is to add an extra switching element to the Benes̆ network.

To add fault tolerance to a network used in MIMD mode, one approach is to provide multiple paths between inputs and outputs.Two ways to accomplish this are (1) to add an extra stage of switching elements to the network; or (2) to add extra links between stages. The former is done in the Extra Stage Cube network and in the baseline with an extra stage. In the figure, the baseline is shown with a dashed line because it requires some modification to qualify as fault tolerant. The second method for providing multiple paths is used in the IADM network with half-links are added and in the F-network.

All of the approaches listed so far have drawbacks, a number of which are discussed in [LiL82] and will be enumerated here. The SIMD mode fault model is very optimistic. It assumes that the only kind of fault that can happen is that a switching element will get stuck in one of its valid states. Not considered are invalid states, link failures, and switching element failures that alter data. The MIMD mode approaches

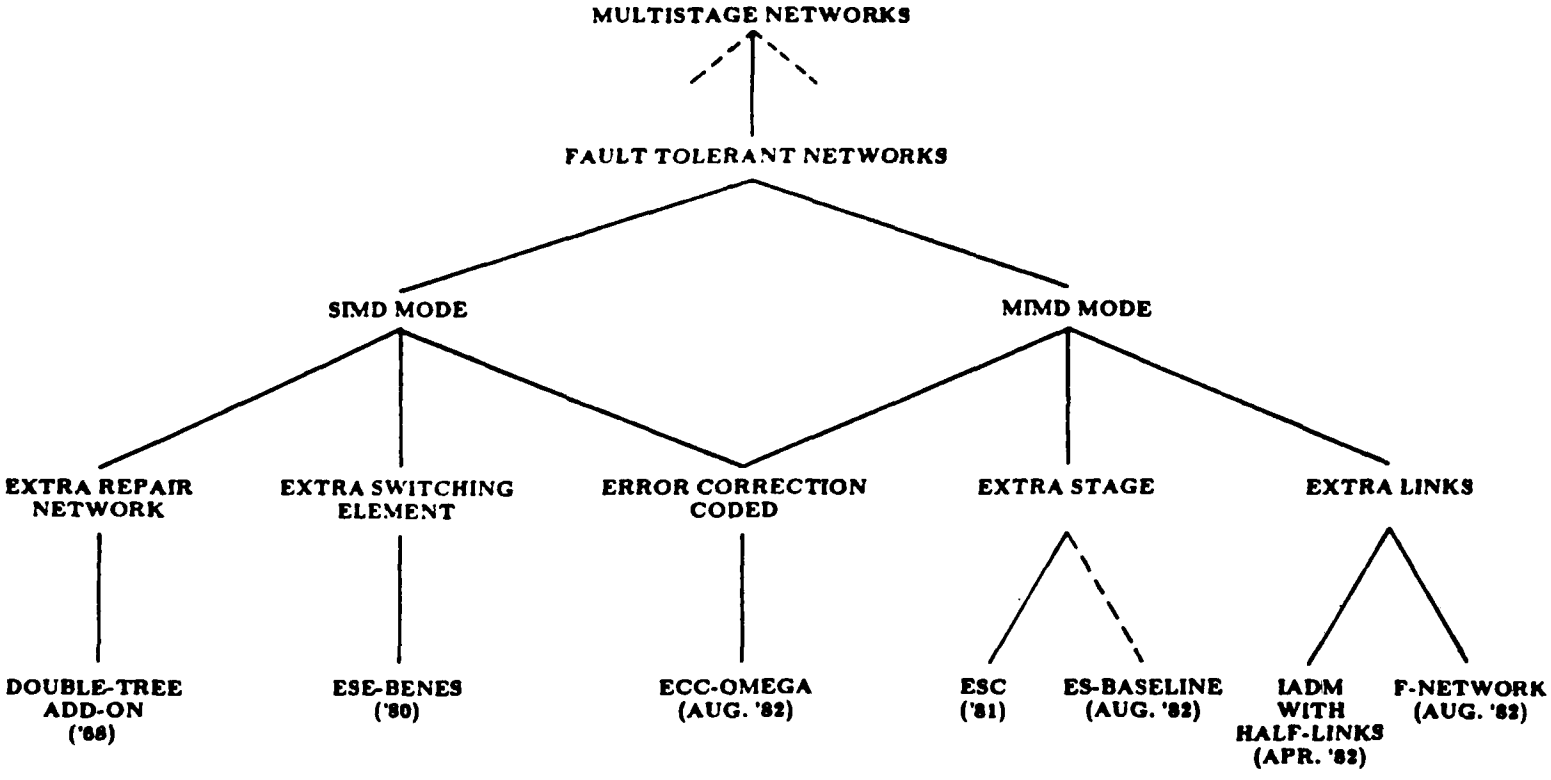


Figure 3.45 Fault Tolerant Multistage Network Family Tree

do deal with these problems but cannot route permutation connections in one pass (two are needed). If these approaches depend on periodic diagnosis to detect faults, transient errors will go undetected. The MIMD mode networks also have more complex routing tag schemes.

The solution proposed in [LiL82] to all these problems is the Error Correction Coded (ECC) Omega network. This is listed under both SIMD and MIMD mode in the figure. A crucial assumption required for this scheme to work is that the switching elements are bit sliced. Then different parts of one switching element can fail independently. For current technology, this is a reasonable assumption for many applications, due to the pin limitations of VLSI chips. The main drawback to this scheme is the large amount of extra hardware required since the parity bits increase the total path width significantly.

3.6 Conclusions

This chapter has presented a broad survey of multistage interconnection networks. In addition to examining the different topologies that have been proposed, different switching element implementations, distributed control schemes and fault tolerant designs have been discussed. For the most part, the networks included were those deemed to be suitable for use in large-scale systems. The other networks included were those originally proposed for telephone exchanges, from which modern parallel processing networks descended. These early networks not only have historical significance, but their structure and properties are highly related to the modern networks.

In this survey, wherever possible, the networks have been discussed independent of any particular system configuration in which they might be used. It is possible, however, that some configurations would require modification of some of the schemes described. For example, actual switching element design depends highly on whether a

unidirectional or bidirectional approach is taken; also upon whether packet or circuit switching is used. On the other hand, topology, control method and fault tolerance issues are more independent of such considerations.

In Section 3.2, seven classes and 17 different multistage networks were presented. It was found that the majority of the networks designed for parallel processing fall into one of two categories: (1) cube type and (2) PM2I (data manipulator) type. The cube type group has 10 members and the PM2I type has four members listed in Figure 3.25. This result should simplify the network notation scheme to be developed. For example, if the notation simply specifies the sequence of interconnection functions implemented by the network, nearly all of the networks listed in Figure 3.25 can be represented. The Omega and Generalized Cube networks are represented by $cube_{n-1}$, $cube_{n-2}$, . . . , $cube_1$, $cube_0$; the indirect binary n-cube by $cube_0$, $cube_1$, . . . , $cube_{n-1}$; and the ADM by $PM2_{\pm n-1}$, $PM2_{\pm n-2}$, . . . , $PM2_{\pm 1}$, $PM2_{\pm 0}$. Other interconnection functions need to be identified so that any network can be described this way. This represents the structural aspects of a network.

Based on the information in Sections 3.3 through 3.5, other parameters to be included in the notation can be identified. Implementation parameters should indicate whether the switching elements are crossbars or connect one input to one output at a time; this is in addition to the protocol related parameters discussed in Chapter 2, Section 2.5.2. Control parameters should indicate how many bits are used in a routing tag and how they are interpreted. Fault tolerance can be specified in terms of the number of link faults and switching element faults that can be tolerated.

A good approach to representing control and protocol information would be to use an ISP-like notation. For example, this would allow queue structures to be defined for packet switching and the interpretation of a routing tag would be analogous to decoding an instruction (but much less complex).

CHAPTER 4
HIGH LEVEL DESCRIPTIONS OF CONCURRENCY IN PROCESSES

4.1 Assessment of Parallel Programming Languages

4.1.1 Introduction

Recently, parallel systems have become practical and feasible. In order to program these systems efficiently, a number of parallel programming languages have been proposed and many of these have been implemented, both for simulation and actual use. Many of the languages have been developed with particular machines in mind, yet many are general purpose.

In terms of modeling distributed processing, parallel and concurrent programming languages provide one means of representing the parallelism in a process. Given a list of desired algorithm parameters, several languages are investigated here to see if these parameters can be determined given the program text and assumptions about run time parameters.

4.1.2 Language Features

Several important algorithm features have been investigated (see Chapter 5). From these, a preliminary list of minimal language features is proposed. In order to perform an effective analysis of a program written in a particular language, the language should include specifications of:

1a. size of local data declared

b. size of global data declared

2a. concurrent processes

b. dynamic process creation and termination

3a. description of communications paths

b. communications primitives

4 . synchronization primitives

5 . data types

6 . other global resource requests

These features are discussed below.

1a. *Size of Local Data Declared*

b. *Size of Global Data Declared*

Let any data that is accessed entirely by one processor be called local. Any data that is accessed by more than one processor is called global. A further distinction could be made for data that is accessed by only a few processors. This distinction is not valid here since similar performance penalties are incurred when any global access is made. The critical distinction is the number of actual accesses to global data, not the number of possible accesses to that data. Local data size determines the amount of memory per processor required on a target machine. A large global data base may indicate a need for a global shared memory or a fast interconnection network.

2a. *Concurrent Processes*

b. *Dynamic Proce · Creation and Termination*

A means for specifying concurrent processes is necessary. The analysis must know how the algorithm is to be divided among processors to match it to the number of processors available in a system. Most MIMD real time image processing tasks require a significant amount of dynamic task reconfiguration. To accomplish this a language must be able to specify the creation, execution, and termination of processes. (A subset of digital signal processing tasks involve a one time static enumeration of processes. These tasks could be described by declaring a static set of processes. However, for a language to effectively describe all algorithms in the problem domain, it must be able to express dynamic task changes.)

3a. *Description of Communications Paths*

b. *Communications Primitives*

Interprocessor communication is an important feature of any parallel language. The topology of communications paths is important since it determines the types of communications networks required. The implicit or explicit enumeration of communications allows analysis of the bandwidth necessary in any given network implementation.

4. Synchronization Primitives

If synchronization is required on a fine grain, it is imperative that the architecture support some quick method of synchronization. The granularity is determined by analyzing the frequency of occurrence of synchronization in the algorithm.

5. Data Types

The requirement on data types is dictated by digital signal processing applications rather than by the parallelism in the tasks. This enumeration of types determines the instruction set capabilities required of a given processor architecture.

6. Other Global Resource Requests

Other requests cover any special requirements of a given algorithm for global resources such as I/O devices and disk storage. These may be considered for specific algorithms, but in general, these requests will not be considered here.

4.1.3 Languages

Following is a description of how the features above are implemented in several languages. A discussion of trends and a summary of "ratings" is given at the end. This is not intended to be a complete summary of all parallel languages. It is merely a look at a cross section of more common or recent languages, and how well the languages support the features listed above. Comments about the ease of analyzing the implementations are inserted where appropriate.

Ada

Ada [DoD80] is a new programming language, sponsored by the United States Department of Defense. It includes mechanisms for MIMD parallel processing.

Concurrent processes in ADA are called **tasks.** The number of tasks is completely determined from the declarations. These tasks are initiated when the program unit in which they are declared is entered. Even though the number of tasks is free to vary, the number is easily extracted from the declarations in the current program unit. Elaborate facilities are available for termination of tasks.

Communications are performed by the **rendezvous** concept. In a multiprocessor system, this could be implemented easily with interprocessor communications. The input and output parameters plus some fixed overhead define the amount of data transferred for each rendezvous. Synchronization is defined with this same mechanism, except these rendezvous actions are defined with no input or output parameters. The communications bandwidth necessary can be determined from the fixed overhead of arranging the rendezvous.

Ada is strongly typed and all data formats and sizes can be easily determined from the program text and the **STANDARD** package, defined for a particular host architecture. For instance, the sizes of SHORT_INTEGER, INTEGER, and LONG_INTEGER can be defined to best fit a given architecture. Thus all data sizes and formats are easily determined. In Ada, all variables are assumed local to tasks unless otherwise indicated. Global variables are those that appear in **SHARED_VARIABLE_UPDATE** calls. In a multiprocessor implementation, this suggests some form of broadcast. These variables are the global data; all others are local.

Because of its completeness, Ada is a complex language to compile or analyze. The syntax does, however allow analysis of the indicated program features.

CSP

CSP [Hoa78] stands for Communicating Sequential Processes, and is a language proposed by Hoare to describe these communications. CSP, as proposed, is a syntax to describe interprocess communications. The serial processes themselves are written in a language that is implementation dependent. Therefore it is not a complete language specification in itself, although complete languages have more recently been implemented from the original definitions.

In CSP, processes are created when the program is begun. Thus, the maximum number of processes is known. Processes are permitted to terminate asynchronously. Presently, dynamic process creation is not supported. Since this is dependent on the implementation, this may become available in later developments. From the program, the number of processes can be determined but no variation of this number in time is permitted.

Communications is a fundamental part of CSP. All communications are invoked by asking to give or take data to or from another process on a defined **channel.** The communication operation is complete when both sender and receiver request the operation. Synchronization is accomplished in the same way, through the transfer of a dummy value, since neither process can continue past a communication until the operation is complete. The necessary number of transfers in a communications network can be determined from the explicit input or output statements.

Data type specifications are part of the given implementation. One would expect all implementations to use a typed structured language like Pascal in [Ada82] and C in [JaG82]. Thus types and sizes of data will be easily available for the analysis. All data is local to processes in CSP; no global data is permitted. All global data must be simulated with communications constructs.

Concurrent Pascal

Concurrent Pascal [Han77a] was developed for writing real time operating systems on uniprocessor machines. The power of the language allows its use with multiprocessor systems.

The data types are easily determined since the "conventional" serial portion of the program is written in Pascal. All variables are local to processes. Conventional global variables are replaced by **monitors.** These monitors define global variables and all possible permitted actions on these variables. Monitors guarantee exclusive access

through some semaphore technique or equivalent. The size of global data is determined by the monitor declarations and the access of these variables is indicated by a call to a monitor **entry.** Thus, these features are quantifiable. **Processes** are important constructs in Concurrent Pascal. As in CSP, these are statically initiated when the program is run. Thus the number of processes is fixed and can be determined from the number of processes declared in the program source.

As implied, all communication and synchronization are performed through monitors. These monitor calls can be considered predefined custom communication modules guaranteeing exclusive access to communication buffers. Through analysis of the monitors, the types of communications can be analyzed. Through analysis of the monitor entries called, the communications bandwidth can be estimated.

Concurrent Pascal is very structured, thus limiting some features such as process creation. However, this same structure makes it easy to analyze in order to produce reasonable quantifications of the indicated features.

Path Pascal

Path Pascal [CaK80] is an extension of Pascal which accounts for dynamic concurrent execution of processes, under restrictions of path expressions. Path expressions represent characteristics such as the amount of concurrency, precedence and ordering constraints, and restrictions on concurrent execution. Features for real time use have been added for interrupt handling and process priorities. Although Path Pascal was designed for uniprocessor systems, its extension to multiprocessor systems is direct.

Data types are complete as in standard Pascal. All variables declared within the context of a process are local. All other variables are global and require interprocessor communication to access them in a multiprocessor implementation. In fact, all interprocess communications are performed through these variables. Thus, communications can be quantified by the accesses to these global variables; the amount of data

transferred is related to the size of these variables.

Processes are clearly defined in Path Pascal. Restrictions on the execution of processes may be imposed with path expressions. These path expressions describe the synchronization constraints. Within these restrictions, processes are created when they are invoked much like a Pascal procedure. The difference is that the invoking process continues without waiting for the process to finish, thus producing a coroutine instead of a subroutine. Analysis of these invocations within the path constraints can produce the number of concurrent processes at any given time during execution. The path expressions can also implement a form of synchronization.

The path expressions introduce complexity into the analysis, but this complexity is traded off against the ability to invoke processes on a dynamic basis. The communications facilities are not explicit and must be implied from the use of global variables. Path Pascal emphasizes the concept of processes, while de-emphasizing communications issues.

Modula

Modula [Wir77] was developed by Nicholas Wirth (author of Pascal). It is intended for real time operating systems and primarily uniprocessor systems. A subset of the Pascal language is expanded to account for multiple processes, signals, and device dependencies.

Variables are strongly typed as in Pascal. Variables declared in the main program block are global to all processes, those declared within processes are local. Therefore, it is trivial to determine the type, size and scope of variables.

Separate processes may be declared only in the main program block. Processes are initiated much like a procedure call, but the calling routine never waits for a return. This limits the flexibility somewhat, but provides for easy analysis of the number of processes at any point in the program execution.

Communications are performed through the use of interface modules. Thus, the exact specification of the communications is programmed by the user in these modules. Analysis of these modules in a program will give a measure of the amount of communication per invocation of the module.

Signals are used to provide an explicit synchronization mechanism. They may be implemented in a more efficient manner than general communications. The analysis is straightforward since all signal actions are performed through the primitives **wait, send,** and **awaited.**

Modula provides a limited but useful dynamic process creation facility. The communications is not built in to the language, but is confined to interface modules. Signals provide explicit synchronization primitives. The language is well defined, and allows direct analysis.

Edison

Edison [Han81] is a language developed by Brinch Hansen (designer of Concurrent Pascal.) It is designed specifically for multiprocessor operation. In comparison to its predecessor Concurrent Pascal, it is more flexible while losing very little of the strict structure.

Based on Pascal, the typing of variables is complete. Both local and global variables are supported. Procedures containing a **cobegin** statement contain variables that are considered global. That is, processes starting with a **cobegin** statement in a given procedure have access to the variables in that procedure. Otherwise, all variables declared within procedures are local.

Separate processes are introduced in a program by a **cobegin** statement of the form:

```
cobegin
        1 do proc1
also
        2 do proc2
also
        3 do proc3
end
```

The number may optionally be used for processor binding. The "parent" process will not proceed until all the "children" have finished. This allows for great flexibility, while providing for easy analysis of the number of processes running in any portion of the program.

Communications are not explicit and analysis must be based on global variable accesses and parameters passed to processes. Variables that are shared may be accessed exclusively through use of the **when** statement. Communications are then defined with implicit use of these variables. Communications may also occur when a process is passed parameters or returns values. The analysis of communications must interpret these implicit communications.

Comments and Summary

In interpreting a language, the concepts of global store and interprocess communications are not distinct. A given MIMD system can simulate an interprocessor communications network through a global store, or can simulate a global store through interprocessor communications. Both of these concepts suggest information is transferred from the domain of one processor to the domain of one or more other processors. So in a sense, both are equivalent ideas. Some languages restrict specification of "communications" by not allowing global variables, while others provide no explicit

communications primitives. Synchronization may also be expressed as combinations of globally stored semaphores or communications protocols. Different languages emphasize different parallel constructs by providing primitives for those operations. The languages that look more conventional are those that use global variables for communications. The implication on a multiprocessor system with no global store is that every access requires communication to the processor where that variable is actually stored. This is much less efficient than a local memory access. If the program is written using a large number of global variables, it should not "match" well with a machine with no global store. Likewise if the program is written using communications primitives for all communications, it will not take advantage of a global store. Since a language may exclude use of one or the other operation, the language in which an algorithm is expressed influences the type of architecture on which the program will run best. Although it is the role of the compiler to implement the constructs used in an efficient way, the most versatile language is one that allows specification of transfers in either form.

The ability to alter the number of running processes dynamically is a desired feature of a parallel language. It may not be necessary for all problems, especially those on a small scale. However, as the problems become more complex, it is essential to have this capability in the language.

The languages Ada, Concurrent Pascal, Modula, and Edison all look like conventional languages (Pascal). They are similar in appearance and also in philosophy. Concerning process creation, in order of increasing capabilities, Concurrent Pascal is the most limited since all processes are started once. Modula declares dynamic processes cleanly in the main program block, but does not allow more than the first level of process declarations. Ada generates all the processes in a program unit when that unit is entered. Edison goes the furthest to allow processes to be started at any point in the program.

Ada and Concurrent Pascal provide explicit primitives for communications and synchronization in the form of a rendezvous or monitor. Modula provides interface modules for custom communications formats. Of these four more "conventional" languages, Modula is the only one to provide a signal mechanism specifically to facilitate synchronization.

CSP and Path Pascal have markedly different objectives from the four languages just discussed. CSP describes communications through the use of explicit operators. The analysis is simple and few assumptions need to be made. The number of concurrent processes cannot be altered dynamically in CSP. Whereas CSP is built around communications, Path Pascal concentrates on process ordering and concurrency. A path expression describes what ordering of processes is allowed. On communications, Path Pascal provides nothing but global variables.

4.1.4 Languages and Language Features

Following is a summary of how well the languages discussed above express the desired features.

1a. *Size Of Local Data Declared*

b. *Size Of Global Data Declared*

All the languages distinguish between local and global variables. Some are more explicit than others. Ada assumes variables are local to processes unless they are involved in SHARED_VARIABLE_UPDATE calls. CSP has only local variables. Concurrent Pascal limits global variables to user defined monitors. Path Pascal defines local variables within process definitions and global variables outside the scope of the processes. Modula has global variables in the main program block and local variables with processes. Edison has global variables in blocks that generate parallel processes,

but variables within processes are local.

2a. *Concurrent Processes*

b. *Dynamic Process Creation and Termination*

Path Pascal has the most the most complex and most thorough description of concurrent processes. Edison allows concurrency to be introduce dynamically at any level. Ada generates groups of processes when their given program unit is entered. Modula allows concurrent processes to be generated only from the main program. CSP and Concurrent Pascal provide a static enumeration of processes.

3a. *Description of Communications Paths*

b. *Communications Primitives*

CSP provides a built-in mechanism for communications, as well as a description of all possible communications channels. Other languages provide user definable communications mechanisms. Ada provides rendezvous, Concurrent Pascal provides monitors, and Modula provides interface modules. Path Pascal and Edison do not provide explicit communications mechanisms.

4. Synchronization Primitives

Modula provides explicit communications mechanisms. Path Pascal provides a complex mechanism for providing synchronization through path expressions. The other languages provide for synchronization with the same mechanisms that provide communications.

5. Data Types

All languages were modifications of Pascal, except one implementation of CSP based on C. With these languages, all variable are declared as a certain type. Thus analysis of variable types is straightforward.

4.1.5 Summary

Different languages approach MIMD parallel processing from different viewpoints. Regardless of the approach, one can assume the language is implemented as a series of primitive operations such as data transfers or synchronization mechanisms. Some language constructs may be implemented through global store or interprocessor communications. Analysis must consider this and account for all communications on a higher level. Some languages make it easier to extract the indicated features by making them more explicit. With a suitable amount of effort, all the indicated features can be extracted from the languages examined. Parallel or concurrent languages may therefore provide a useful means of representing some of the features of a distributed process. Moreover, the language constructs can be applied at both the general design and detailed implementation phases. In the design phase, a coarse decomposition of a task can be expressed in terms of general descriptions of data requirements, communications patterns, synchronization points, and process creation/execution/termination, using the representations provided by a concurrent language. In the implementation phase, the details of these attributes can be filled in. Concurrent languages can therefore serve as one possible modeling tool for describing distributed processes.

4.2 Graph Theory Applied to Modeling of Asynchronous Computation

4.2.1 Introduction

Our initial study of the application of graph theory for describing asynchronous computation has focused on Petri nets and their extensions. The literature contains many examples of the use of Petri nets to model various aspects of asynchronous computation. Many extensions are also described along with the limitations which they help to overcome and the tradeoffs that they impose. Here we summarize some of the most interesting results and suggest further extensions.

C. A. Petri developed the basis for Petri net theory in his Ph.D. thesis [Pet62]. His work drew the attention of people working on two important projects: A. W. Holt and others from the *Information-System Theory Project* of Applied Data Research Inc., (ADR) and also of J. B. Dennis' Computation Structures Group of Project MAC at M.I.T.

The final report of the ADR Project [HoS68] detailed much of the early theory, notation and representation of Petri nets developed and extended in the course of the project. Of particular relevance to this report is a paper [HoC70] showing that Petri nets could be used to model and analyze concurrent system components.

Dennis and others of M.I.T. have published many reports and dissertations on Petri nets [Den72,Hac72,Bak72]. Conference proceedings from the MAC *Conference on Concurrent Systems and Parallel Computation* in 1970 [Den70] and *Conference on Petri Nets and Related Methods* in 1975 have helped to develop and spread the ideas and results of Petri net theory.

4.2.2 Basic Petri Net Theory [Pet77]

A Petri net is a formal structure composed of a set of places P, a set of transitions T, an input function I, and an output function O. The input and output functions (defined relative to the transitions) relate the transitions and places. Thus, a Petri net structure C is a 4-tuple:

$$C = (P,T,I,O)$$

where, for example, sets of places and transitions, P and T, might be written

$$P = \{p_1,p_2,p_3,p_4,p_5\} \text{ and } T = \{t_1,t_2,t_3,t_4\}$$

The input and output functions are collections of places for each transition. Since a place may occur more than once as in input to or output from a given transition, the input and output functions generate "bags" rather than sets. (A bag is like a set except that an element in a bag may occur zero or more times, whereas recurrences of an element in a set are not significant.) We could write the I and O functions for an example net as:

$$I(t_1) = \{p_1\} \qquad O(t_1) = \{p_2,p_3,p_5\}$$
$$I(t_2) = \{p_2,p_3,p_5\} \qquad O(t_2) = \{p_5\}$$
$$I(t_3) = \{p_3\} \qquad O(t_3) = \{p_4\}$$
$$I(t_4) = \{p_4\} \qquad O(t_4) = \{p_2,p_3\}$$

The structure and arrangement of the Petri net may not be very obvious from its written description so the graphical representation is more commonly used. Places are represented as circles and transitions as bars. The input function is diagrammed by directed arcs from places to transitions while arcs from transitions to places represent the output function. Thus the Petri net defined by the formal structure C = (P,T,I,O) above may be shown graphically as in Figure 4.1.

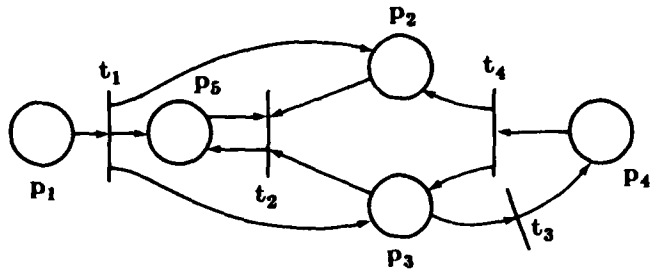


Figure 4.1 Petri Net Graph Equivalent to Given Example Structure [Pet77]

When modeling systems, events and situations with Petri nets, the places (circles) are used to represent conditions or the status of some element of a system. The transitions (bars) represent actions; in other words, the transitions are from one state to another.

To be able to show dynamic characteristics with Petri nets, a net is *marked.* The presence of a token in a place can be thought of as indicating that the condition represented by that place holds, the absence of a token indicating that the condition does not hold.

The earliest Petri nets were restricted to having either zero or one token in a place but now unless some specific limit is imposed, a place may have any non-negative integer number of tokens. A marking function μ yields the number of tokens in a place. The range of the marking function is the set of non-negative integers. For example if there are 3 tokens in place p_i, then $\mu(p_i) = 3$. A marking may also be written as a vector. For example, a net with places $P = \{p_1,p_2,p_3\}$ marked as:

$$\mu(p_1) = 1 \quad \mu(p_2) = 5 \quad \mu(p_3) = 0$$

has a marking vector $\mu = (1,5,0)$.

The activity or execution of a Petri net is made up of *transition firings* governed by *firing rules*. These rules have been defined in different ways at different times but there is now general agreement on the rules for basic Petri nets. They are as follows:

A transition may fire if it is enabled. A transition is enabled if there is at least one token in each input place. When a transition fires, a token is removed from each input place and a token is added to each output place.

Note that nothing has been said to indicate timing of execution. The basic definition of Petri nets says only that an enabled transition *may* fire, not that it must. Also a firing is defined as occurring instantaneously, i.e., in zero time. Thus the probability of two or more transitions occurring simultaneously is zero. A transition may be enabled for an indefinite length of time.

4.2.3 Modeling of Concurrency

Hwang and Briggs [HwB81] give an introduction to the use of Petri nets to model asynchronous concurrency and concern themselves with the formal definition of Petri net structures. They point out that Petri nets can be used to model the specific class of problems defined as "discrete-event systems with concurrent (parallel) events." See Examples 1, 2 and 3.

Example 1 - Petri Net Used to Show Asynchronous Concurrency [adapted from HwB81]

Figure 4.2 represents a computer system in which a processor is dedicated to serving two input devices that are gathering data.

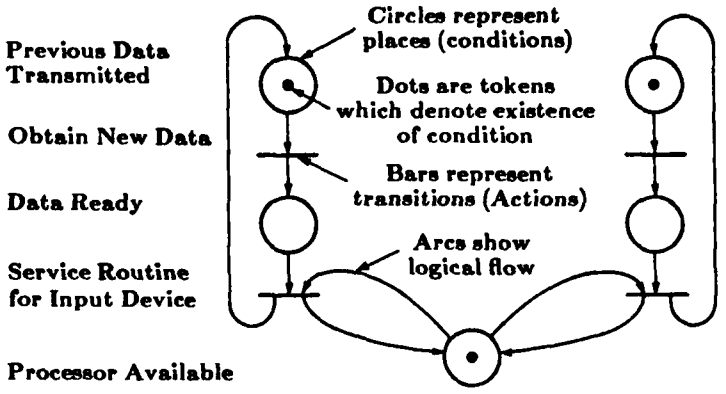


Figure 4.2 Modeling Asynchronous Concurrency

Example 2 - Petri Net Used to Model Various Constructs of High Level Language Including Explicit Concurrency [adapted from HwB81] See Figure 4.3.

```
program P0;
T0;
while P1 do          /* T2 */
      if P2 then
            T3;
      else
            T4;
      endif                /* P3 */
      cobegin              /* T5 */
            T6;
            T7;
            T8;
      coend          /* T9 */
endwhile
goto P0                    /* T1 */
```

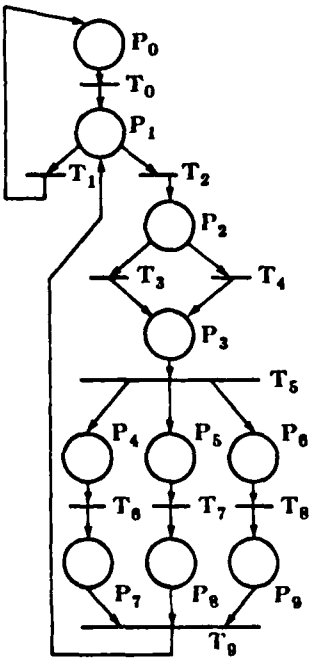


Figure 4.3 Modeling Concurrent High Level Language Program

Example 3 - Petri Net Model of Producer/Consumer Relations With Bounded Buffer [HwB81]

Figure 4.4 shows the relationship between a producer and a consumer with a buffer of fixed size between them. If the buffer is full, the producer is blocked.

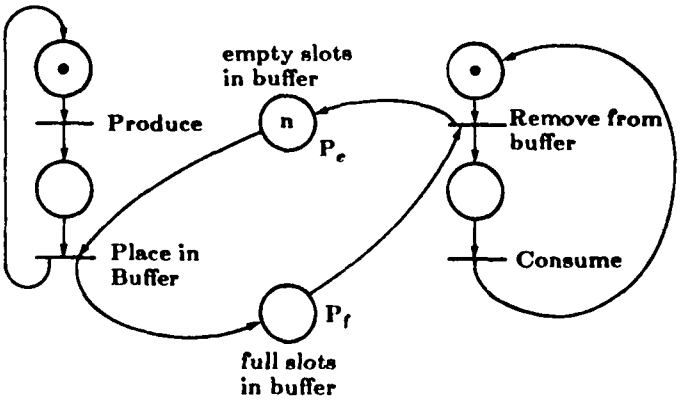


Figure 4.4 Model of Producer/Consumer Relations. The total number of tokens in P_e and P_f is the total number of slots in the buffer.

Peterson [Pet81] gives a more complete and general formal definition of Petri nets and gives examples of Petri nets used to model synchronization in various multi-task, multi-program or multi-processor systems. Examples 4 and 5 show Petri net models of synchronization primitives.

Example 4 - Mutual Exclusion for Critical Sections [Pet81]

Mutually exclusive critical sections are program segments which must not be executed concurrently; for example, sections of an operating system which allocate and de-allocate memory blocks.

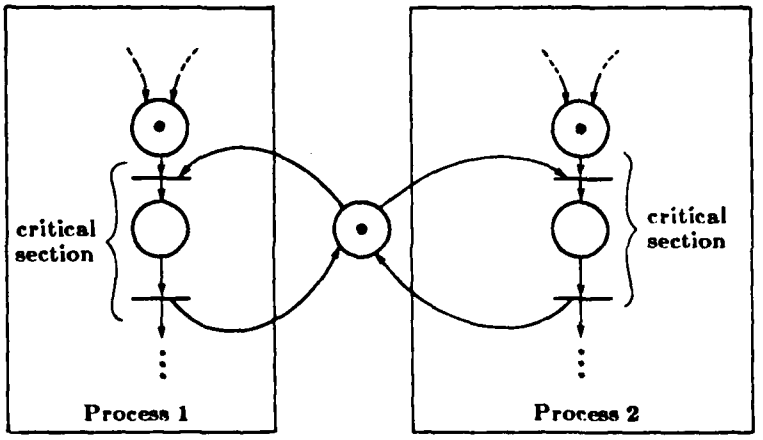


Figure 4.5 Modeling Mutually Exclusive Critical Sections

Example 5 - P and V Synchronization

The P and V operations on semaphores invented by Dijkstra [Dij65] can be modeled by Petri nets as follows. Extended P and V operations can be represented by allowing multiple tokens in the semaphore place S.

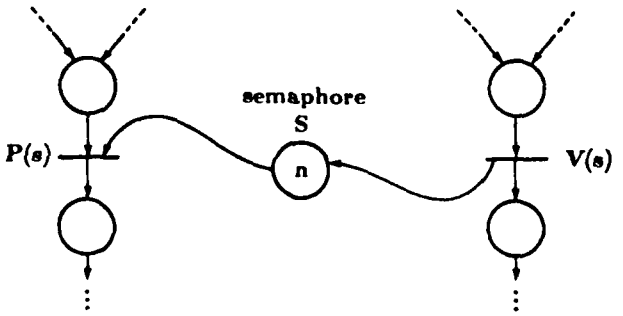


Figure 4.6 Modeling P and V Operations on Semaphores

Krygiel [Kry81] defines vector masks and vector places to yield a tool called synchronous nets (S-nets). These constructs are designed for and applied to the modeling of SIMD machines and offer a shorthand for Petri nets of large multiplicity. See Examples 6 and 7.

Example 6 - Vector Extensions to Petri Nets to Make Synchronous Nets

Vector mask places are made up of vector places and vector masks as shown by Figure 4.7.

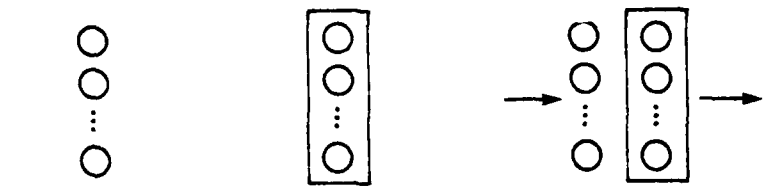


Figure 4.7 Vector Elements of S-Nets

The vector place represents aggregates of logically associated and homogeneous conditions whose initial and ceasing events are synchronized; for example, the conditions of an array of SIMD processors. The vector mask can model the participation or non-participation of elements of the vector place in firing of transitions. A vector place is holding if at least one of the elements of the vector mask is marked to 1 *and* each element of the vector place corresponding to the mask elements marked to 1 have a non-zero marking. The markings of the masks are chosen from a list of valid markings given by a descriptor of the immediately preceding transition. The choice of which possible marking is active is not determined by the model itself but by external means.

Example 7 - S-Net Representation of a SIMD Machine with Array of Three Computing Elements [Kry81]

The structure is given by:

$T=\{t_1,t_2,t_3,t_4,t_5,t_6\}$ $S=\{s_1,s_2,s_3\}$

$V=\{V_1,V_2,V_3,V_4\}$ $M=\{M_1,M_2,M_3,M_4\}$

$V_1=<v_{11},v_{12},v_{13}>$ $M_1=<m_{11},m_{12},m_{13}>$

$V_2=<v_{21},v_{22},v_{23}>$ $M_2=<m_{21},m_{22},m_{23}>$

$V_3=<v_{31},v_{32},v_{33}>$ $M_3=<m_{31},m_{32},m_{33}>$

$V_4=<v_{41},v_{42},v_{43}>$ $M_4=<m_{41},m_{42},m_{43}>$

$U=\{<V_1,M_1>,<V_2,M_2>,<V_3,M_3>,<V_4,M_4>\}$

$A=\{ <s_1,t_1>,<t_1,U_1>,<U_1,t_2>,<t_2,U_2>,<U_2,t_3>,<t_3,s_2>,$
$<t_1,U_3>,<U_3,t_4>,<t_4,U_4>,<U_4,t_3>,$
$<s_2,t_5>,<t_5,s_1>,<s_2,t_6>,<t_6,s_3> \}$

with initial marking K_0:

$K_0(s_1) = 1$ $K_0(s_2) = 0$ $K_0(s_3) = 0$

$K_0(V_1) = K_0(V_2) = K_0(V_3) = K_0(V_4) = <0,0,0>$

$K_0(M_1) = K_0(M_2) = <1,0,0>$ $K_0(M_3) = <1,1,1>$

$K_0(M_4) = <0,0,1>$

Figure 4.8 shows this structure graphically.

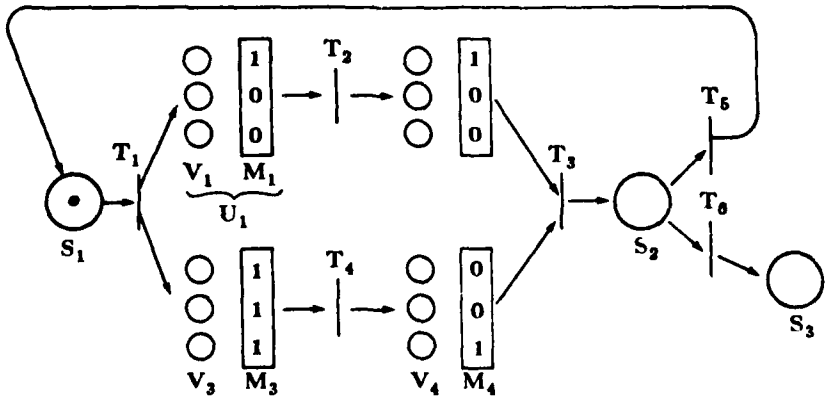
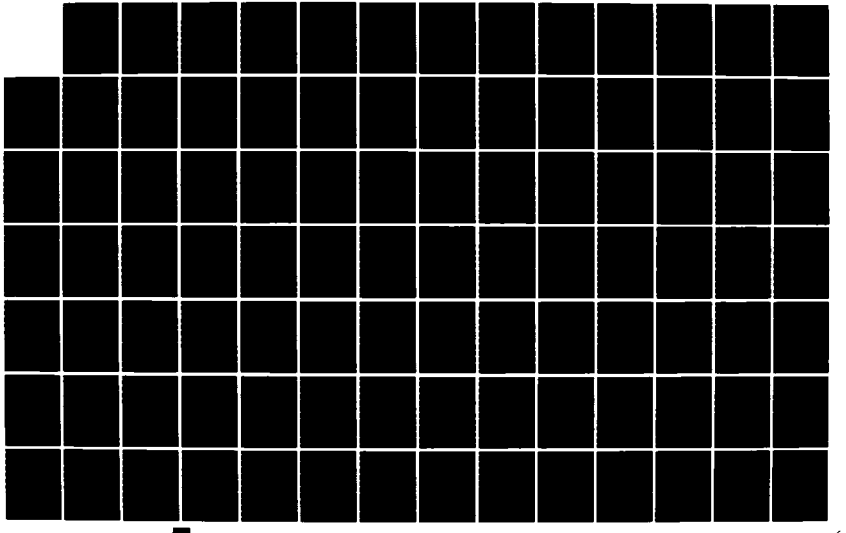
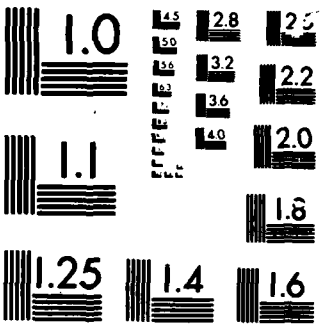


Figure 4.8 S-net Representation of an SIMD Machine (see [Kry81] for details)

AD-A167 316 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR..(U) PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING L J SEIGEL ET AL. 3/4

UNCLASSIFIED MAR 83 TR-EE-83-11 ARO-18790.17-EL-APP-A F/G 9/2 NL





MICROCOPY CHART

Baer [Bae82] shows some of the limitations in the modeling power of Petri nets and some of the tradeoffs involved in the use of various extensions. One such example is detailed in Example 8. Here the extension of inhibitors allows testing the emptiness of a place which is not directly testable with basic Petri nets. Baer further asserts that this extension gives Petri nets the power of Turing machines. The cost of this extension is that "liveness" (the freedom from potential deadlock), and "safety" (the boundedness of the number of tokens in a place), of the nets are no longer decidable.

Example 8 - The Inhibitor [AgF73]

The inhibitor allows testing the emptiness of a place. It is diagramed by an arc with a bar through it.

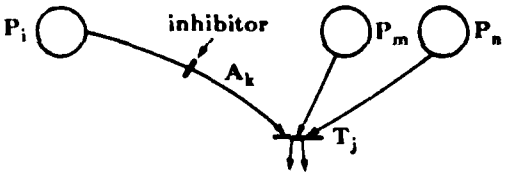


Figure 4.9a The Inhibitor

Transition T_j may fire if and only if it is enabled in the normal way *except* that any input place connected to it with an inhibitor ***must be empty,*** i.e., T_j may fire if and only if P_m and P_n are holding and P_i is empty. No tokens are removed by inhibitor arcs.

Baer [Bae82] shows that this extension allows the solution of a problem posed in [Kos73]: two producers P_1, P_2, two consumers C_1, C_2, and two buffers B_1, B_2. The two consumers are not allowed to access their buffers simultaneously (perhaps they use the same I/O channel) but C_1 has priority: C_2 can access B_2 only when B_2 is full and B_1 is empty. Unless the queues are bounded (that is, safeness constraints are imposed), Petri

nets cannot represent this situation. However, the extension of inhibitors makes this easy.

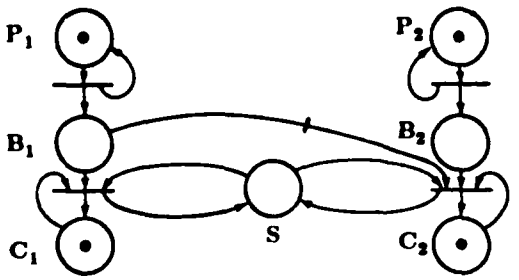


Figure 4.9b Producer, Consumer, Buffer Problem with Priority Shown by Inhibitor

Another tradeoff mentioned by Baer is in the external restriction of "safeness." Without this restriction many of the formal properties are undecidable. (Safe nets can be interpreted as finite state models.)

Disjunctive logic is allowed by an extension to the firing rules. Example 9 shows the use of an exclusive-or function in modeling an IF-THEN-ELSE construct.

Example 9 - Disjunctive Logic Extensions in Petri Nets [Bae82]

Disjunctive logic extensions modify the firing rules of basic Petri nets.

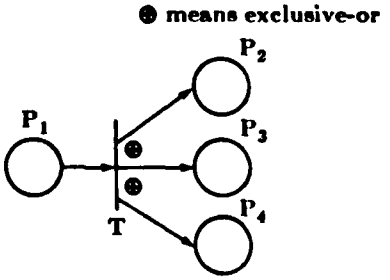


Figure 4.10a Exclusive-or Output

When transition T fires, only one of P_2, P_3 or P_4 will receive a token.

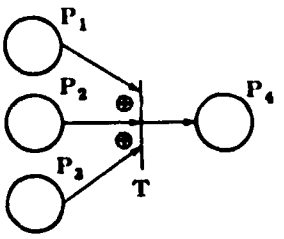


Figure 4.10b Exclusive-or Input

Transition T can fire if one and only one of P_1, P_2 and P_3 contains a token.

This extension also offers the descriptive advantage that a decision-making event is modeled as a transition, which simplifies the model of the IF-THEN-ELSE statement.

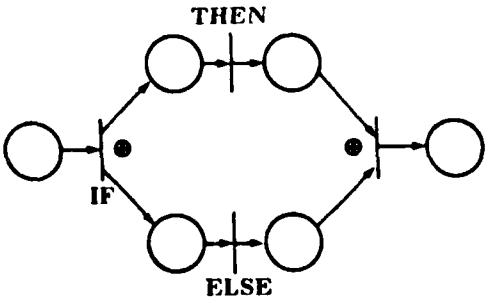


Figure 4.10c Modeling the IF-THEN-ELSE Construct

The class of nets which is always left in a predetermined state after execution is called "properly terminating" or PT-nets. The next extension discussed by Baer introduces token absorbers. These can be used to "soak up" stray tokens which makes it much easier to construct Petri nets of the PT class. Token absorbers are used to "kill"

redundant processes. Example 10 illustrates such a process.

Example 10 - Token Absorbers Yield Properly Terminating Petri Nets [Bae82]

When a transition with absorbers fires, all tokens in the places to which the absorbers are attached are removed. Figure 4.11 is a graph of a table search done in a dual processor environment. Each processor searches half the table. The token absorbers are shown as dashed lines.

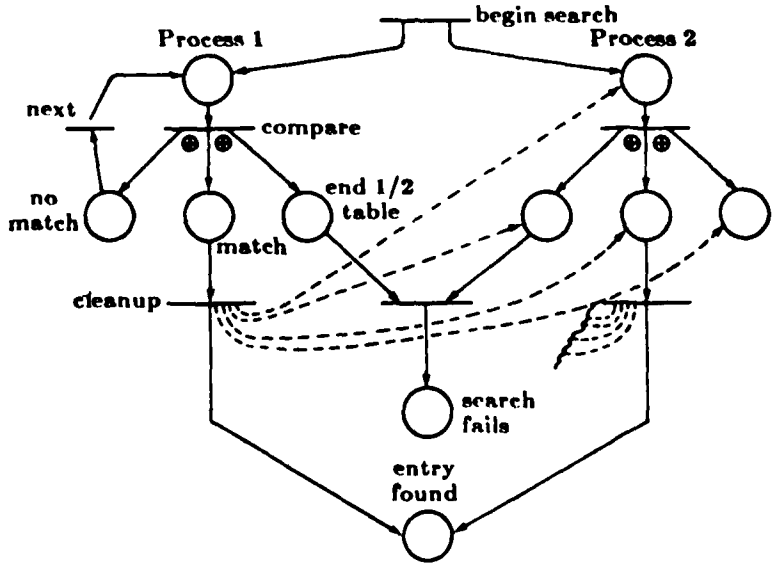


Figure 4.11 A Table Search Illustrating the Use of Absorbers

When the transition labeled "cleanup" fires, all remaining tokens from the other process are absorbed.

The absorbers from the "cleanup" transition of Process 2 are not shown in full but are symmetric to those emanating from Process 1.

Baer goes on to say that with large numbers of processes, the graphs would rapidly become intractable. He shows that colored Petri nets (suggested in [Jen79]) reduce this "spaghetti" effect by allowing tokens of several colors to move about the net independently, thus showing more than one process active on the same graph. They

also can model program re-entrancy and the instantiation of several identical processes [BaJ77].

Peterson [Pet81] examines the formal properties of conservation and coverability and their relation to the modeling power of Petri nets. Strictly speaking, conservation refers to maintaining the total number of tokens in a net. By extension, we are concerned with conservation of resources modeled by the net. Coverability addresses the problem of whether there exists a reachable marking with the number of tokens in given places greater than or equal to those in some specified marking. He also presents a system of matrix equations used to represent a Petri net of large multiplicity in a compact form. This form, it seems, would also lend itself to computer representation of a Petri net model. Peterson concludes with a study relating and comparing to Petri nets many of the existing systems for representing and modeling concurrent computations. He shows (here and in [PeB74]) that conversions between these systems are possible, allowing results from other systems to be extended to Petri nets. His bibliography is particularly well rounded and commented.

4.2.4 Conclusions

In summary, it is clear that various properties of Petri nets are of value in modeling asynchronous computation. The concepts of state-space and next-state functions for Petri nets are natural extensions from finite-state machines, state diagrams and next-state tables. Their particular value here is in the property that allows Petri nets to model both hardware and software functions in an asynchronous parallel environment; thus it should be possible to model the interaction between hardware and software.

The formal properties of safeness and reachability (the ability of one marking to be reached from another) allow these models to be used to check for hazards such as boundedness of the use of resources, and the property of liveness to check for possible

deadlock situations. Formal languages based on Petri nets [Pet81] and some of the properties above have been used to allow optimization in systems or programs represented by Petri nets [ShS70].

Since Petri nets have no explicit relation to the passage of time, except to give a partial ordering to events, there are areas of modeling which may require further extensions not yet found in the literature. Task graphs (graphs which represent precedence constraints of various possibly concurrent subtasks) have certain similarities to Petri nets and usually do have timing possibilities. The properties of a new class of nets with this added timing information need to be further studied.

When a conflict is modeled by a Petri net, there is usually no information about which of two (or more) transitions will fire. Another graph system with some similarities to Petri nets is the Markov-graph, which is a graphical representation of a Markov-chain [Pap65]. Examples of the use of Markov-graphs in concurrent systems is given in sections 7.3, 7.5 and 9.3 of [HwB81]. Here probabilities or probability functions are associated with the various possible transitions. The class of graphs which would result from the merging of Petri net and Markov-graph concepts should also be a powerful modeling tool and would probably be compatible with the extension to task-graph style timing information. Note that the timing might also be expressed as a probability function. The probabilities of transition firing might be functions of time-of-arrival of various tokens in the input (or other) places. More work will be done to explore the properties of these proposed graph concepts.

CHAPTER 5
FEATURES FOR DESCRIBING PROCESSES AND ARCHITECTURES

5.1 Introduction

The direct correlation of hardware attributes to features of software algorithms is a complex task. The approach taken here is to examine a variety of hardware configurations and concurrent programming languages as well as algorithms written in those languages. Salient features of computer architectures are identified and, where possible, metrics are proposed for quantifying or measuring them. Similarly, elements of the programming languages whose effectiveness (e.g., speed of execution) is affected by architectural features (or the lack thereof) are identified. Metrics are proposed for quantifying the extent to which a critical element is present in a given algorithm (e.g., the number of times a synchronization instruction is executed). To produce a correlation coefficient, equations are proposed for combining the architectural metrics with the algorithmic metrics. If architectures and algorithms being considered are specified in sufficient detail, each of the metrics can be evaluated and a set of correlation coefficients produced. In this way, the "best" architecture among those being considered can be chosen to execute a given algorithm, or the "best" algorithm can be chosen for a given architecture.

Preliminary results are presented in the following. A hierarchical approach is taken by defining high and low level features or characteristics. High level metrics are used in the absence of more detailed information. When more details are available, low level metrics, which are a refinement of their high level counterparts, are applied. The

format for presenting the metrics is as follows. Each section contains an algorithmic feature and its associated metric, and discussion on how the metrics can be used to determine compatibility between the features. It remains to evaluate the effectiveness of the features discussed by applying them to proposed architectures and existing algorithms.

5.2 Higher Level Characteristics

5.2.1 Uniformity

Algorithm: Uniformity of Processing

During the course of a computation, resource requirements vary. Resources include processors, memory, and the interconnection network. Appropriate features which relate to these resource requirements include:

* Degree of concurrency
* Frequency of interprocessor communications
* Structure of these communications
* Data type conversions

Metrics associated with each of these algorithmic features are discussed later. Here, uniformity or the degree to which they change is of interest.

Architecture: Reconfigurability

Architectural features which allow changing demands to be accommodated include:

* Total number of processing elements and ease of reconfiguring these elements into smaller groups
* Overall bandwidth of communications network and number of permutations as well as the amount of overhead involved in reconfiguring the network
* Ability to operate on different types of data and the ease of converting between these different formats

These metrics are intended to match dynamic algorithms with dynamically reconfigurable architectures. If an algorithm changes its requirements often, it is desirable to accommodate these changes quickly and efficiently. Algorithms that are more "static" in their needs do not require such capabilities, thus the additional cost of a reconfigurable system should produce a poorer match between these features.

5.2.2 Global Control

Algorithm: Global Control

Concurrent algorithms run on separate processing units independently. In order to guide the overall flow of the program, global control is required. This control is considered overhead in the computations if its execution cannot be overlapped with that of the non-control instructions. In some algorithms, it may even dominate the execution time. Several programming concepts are included in this category. Among them are:

* Synchronization (e.g. semaphores)
* Process generation (task spawning)
* Dynamic allocation of resources such as
    * Processing elements

    * Memories
    * Communication channels
* System reconfigurations

The proposed measurement for each of these is their frequency of occurrence. That is, if the number of control steps per total program steps at run time can be estimated, the amount of overhead incurred can be determined for any given architecture. An algorithm with few global control statements does not require elaborate hardware control mechanisms.

Architecture: Global Control Mechanisms

Various architectures have provided means for control activities on a global level. These special mechanisms reduce overhead for executing programs and provide facilities for the easy implementation of concurrent programming constructs. These mechanisms include:

* Test and set hardware
* Fetch and add hardware
* Process state management hardware
* Communications network
    * Structure
    * Flexibility
    * Bandwidth
    * Setup time
    * Extent of built-in control

* Control units

Many of these hardware features were designed to accommodate some software need. One possible measurement of fit is a matching of features with needs. Then bandwidth is considered in each case. Also, the architecture description needs to be complete enough so that the capabilities of providing for each software need can be measured. That is, if an architecture does not provide a particular mechanism, what time penality is paid for using an alternate, slower hardware mechanism?

Thus even in the case where an architecture does not directly provide for a control need, the overall performance can still be estimated. A further refinement considers maximum capabilities and insurance that worst case demands are met.

5.3 Lower Level Characteristics

5.3.1 Parallelism

Algorithm: Degree of Parallelism

Algorithms incorporate parallel constructs on many levels. These include explicit definition of concurrent processes as well as parallelism implicit in loops. The number of bits in each data type can be considered a measure of a rudimentary form of parallelism, at the word level. If the program is described as a graph model, the number of independent nodes gives a good measure of parallelism. Consider metrics describing the degree of parallelism at each of several levels including:

* Number of independent processes
* Number of subprocesses
* Width of data types

Let these parameters be indexed as a function of time. Then the maximum requirements as well as the average and deviation can be calculated. These measurements characterize the inherent parallelism in an algorithm.

Architecture: Parallelism Available

Primarily, three techniques have been used to achieve parallelism in architectures: (1) replication of execution units; (2) replication of processors; and (3) pipelining various functional units. A good architecture description should at least identify each of these types of parallelism.

Händler [Han77b, Han81] describes a scheme which include these ideas. He defines two types of parallelism: pipelining and replication. The three types of replication are: (1) multiple computers; (2) multiple execution units; and (3) parallel bits. Each of these functional units can be pipelined producing: (1) macro pipeline; (2) instruction pipeline; and (3) execution pipeline. A macro pipeline consists of a series of processing elements operating on data sets and passing results on to the next processing element. In this way each processing element can be matched closely to the appropriate sub-algorithm. Also pipelining within processing elements can be considered, for example, in the case where each component process is processing arrays of data. In general, a measure of fit for pipelined systems and algorithms at any level can be expressed. Let N be the number of "stages" in the pipe and X be the number of "items" to be processed. Stages may be adders or entire MIMD systems and "items" may be bits of data or large matrices. The efficiency of the system, E, is the average fraction of stages in use. E is given as

$$E = \frac{X}{S+N+X-1}$$

where S is a (possibly zero) pipe start up time. The optimal speedup is N since N stages are working. The performance, P, is NxE compared to a serial system. We will

normalize this measure to between 0 and 1, by replacing the N by $\frac{N-1}{N}$. This new measure rewards (i.e. produces a normalized number closer to 1) algorithms operating on long "vectors," since the time to fill the pipe is short relative to the time it is full. It also rewards pipes in general since more processing can be accomplished as long as "vectors" are involved. A combination of a long pipe and few data items produces a normalized number closer to 0. Hence, the performance metric due to pipelining is:

$$P = \frac{NX}{S+N+X-1} \quad \text{and}$$

$$P_{NORM} = \frac{(N-1)X}{N(S+N+X-1)}$$

A major source of potential speedup in MIMD systems is replication of processing elements. The efficiency of the pipeline concept is hampered by filling the pipe. In replication of elements, the efficiency is degraded only when all units are not kept busy. If X is the replication in the software and N is the replication in the hardware, then the efficiency, E, is of the form

$$E = \frac{X/N}{\lceil X/N \rceil}$$

where $\lceil A \rceil$ is the smallest integer greater than or equal to A. This measure varies between $\frac{1}{N}$ for a serial algorithm and 1 when the parallelism in the algorithm is a multiple of the parallelism in the architecture.

This measure considers only efficiency and does not take into account the overall throughput. Thus, it must be modified to indicate that twice as many *utilized* processors will operate twice as fast. That is, if 32 sub algorithms are involved, they will utilize 16 processing elements just as well as 32, yet the 16 element machine will require twice as much time. Each processor of the 16 element machine would have to run two sub-algorithms. In general, each element has to deal with $\lceil X/N \rceil$ or $\lceil X/N \rceil - 1$ sub-

algorithms. Thus multiplying the efficiency by $\frac{1}{\lceil X/N \rceil}$ accounts for degradation due to running more than one sub-algorithm on each processing element. So our performance measure will be given as:

$$P = \frac{X/N}{\lceil X/N \rceil} \times \frac{1}{\lceil X/N \rceil} = \frac{X/N}{\lceil X/N \rceil^2}$$

This measure of fit applies to any non-pipelined set of replicated units.

5.3.2 Data Types

Algorithm: Data Types

Assume that data can be in the form of fixed or floating point numbers of various sizes. The data required is separated into these categories and the operations on these types are numbered. The resulting measures are

$$\frac{\text{\# floating operations}}{\text{total \# operations}} \quad \text{and} \quad \frac{\text{\# fixed operations}}{\text{total \# operations}}$$

A more detailed analysis separates the count of operation by the number of bits in each format (e.g. 8, 16 and 32 bit fixed point and 32 and 64 bit floating point operation). The number of type conversions is also accounted for.

Architecture: Support of Data Types

Architectures may or may not support in hardware every data type required. If a particular operation needs to be performed in software, the measure of fit is reduced by a factor proportional to the slowdown.

Specifically, if the operations on data types are defined as OP_i, $1 \leq i \leq N$, where there are N possible operations, E_i is the efficiency of OP_i. $E_i = 1$ if the operation is directly supported. Otherwise, $E_i = \frac{1}{b}$, where $(\text{software time})_i = b(\text{hardware time})_i$. If

the number of occurrences of each operation is m_i and $M = \sum_{i=1}^{N} m_i$ then the measure of fit P is given as:

$$P = \sum_{i=1}^{N} \frac{m_i}{M} E_i = \frac{1}{M} \sum_{i=1}^{N} m_i E_i$$

This does not take into account the raw speed of the architecture. This needs to be considered as another measure. This also does not address the problem of knowing how long an operation would take in hardware when that operation is not available.

5.3.3 Local Storage

Algorithm: Local Storage Requirements L

Architecture: Size of Local Memories S

Local storage is defined as non-shared variables. If the capacity of local memory is enough to provide for all local variables, the fit is good. Otherwise, execution will be greatly slowed and the measure of fit should be degraded. This can be expressed by a boolean variable:

$$P = \begin{cases} 1 \text{ if } L \leq S \\ 0 \text{ if } L \leq S \end{cases}$$

A more refined measure can be given if more is known about how frequently the local variables need to be accessed and how much overhead is incurred by keeping local data in a remote or global store.

5.3.4 Local vs Global References

Algorithm: Local versus Global References

Architecture: Memory Structure and Access Timing

All memory references in a program can be categorized as either global or local accesses. Local memory is typically the fastest memory available (excluding registers or cache). Local memory is not shared by other processing elements, thus no contention occurs. If no local memory exists in an architecture, then local access time is the same as global access time. Global memory is accessible by more than one processing element. It is invariably slower than a local memory due to potential contention and more complex access logic. If a global memory as such does not exist, access time can be computed for requesting values from a processing element acting as a global store and transmitting the requested values through an interconnection network. This form of global variable access is much slower, yet should be considered as an alternative to a global store with direct hardware access. In either case, the access times for both local and global memory will be referred to as l and g respectively. g does not include delay due to contention. Assume the number of local and global accesses made during execution of a program are L and G respectively. The fraction of time, t_g, spent accessing global store computed as

$$t_g = \frac{g}{l\left(\frac{L}{L+G}\right)+g\left(\frac{G}{L+G}\right)} = \frac{g(L+G)}{lL+gG}$$

This assumes the global store is operating with little or no contention. To take contention into account, this metric is multiplied by a degradation factor G, typically a function of the global access rate. One way to characterize G is to assume that it is 1 until the access rate or memory load exceeds a threshold, C. That is $G = 1$ if $\frac{G}{L+G} < C$. Beyond the threshold, further loading decreases the performance. Assume that this

degradation is linear. Then the performance multiplier is of the form $1 - a\left[\frac{G}{L+G} - C\right]$. a is the slope, which is a function of the access scheme used. Possible values of C for a given communication scheme are:

$C = 1$	crossbar
$.5 \leq C \leq .75$	multistage interconnection network [Ove82]
$C < .5$	bus

The global access time g can now be modified by the loading function to give a rough estimate of the real access times $g' = gG$. The fraction of time spent on local and global accesses is

$$t_l = \frac{l(L+G)}{lL+g'G} \qquad t_g = \frac{g'(L+G)}{lL+g'G}$$

The numbers t_l and t_g lie between 0 and 1 and give an indication of program behavior. These parameters as a function of time indicate the program's memory access behavior and where bottlenecks may be occurring. It is desirable to keep t_g as small as possible. If t_g is above a given threshold, it will become significant in the measure of fit.

5.4 Summary

A preliminary set of measurable parameters has been presented. In most cases, mathematical expressions describing measures of fit were proposed. Future work includes expanding and refining the list of features, examining ways of combining the individual measures into a single correlation factor to determine overall fit, and "testing" the measures on sample parallel architectures and signal processing tasks. Work by Gonzales [Gon80] on determining the compatibility of an architecture and an algorithm will serve as a starting point for our research on deriving an overall metric.

CHAPTER 6
APPLICATION STUDIES

In contrast to the preceding chapters, the work described in this chapter takes a bottom-up approach. That is, specific signal processing tasks are given parallel implementations, and each task and its implementation are closely scrutinized to identify the salient attributes of the task and the requisite architecture. At this point, each task has been considered independently, but eventually the joint implications of the ensemble of tasks will be determined.

6.1 A Parallel Algorithm for Contour Extraction†

6.1.1 Introduction

This section presents a case study of a two-dimensional signal processing task known as contour extraction. Contour extraction is a difficult problem in image processing, but one of critical importance to many applications ranging from computer assisted cartography to industrial inspection. The task is used to develop model parameters descriptive of important computational needs. These needs are in turn examined to determine important architectural features which should be included in a system designed for contour extraction. The relating of these computation parameters and architectural features is a step toward developing a capability to evaluate candidate architectures in the light of classes of computational processes.

†The research in Section 6.1 was also supported by additional grants. Prof. O. R. Mitchell also contributed to this section.

In the past, edge information has been used to improve threshold selection [Mil79] in the contour extraction process. A new scheme for determining threshold values has been developed by Suciu and Reeves [SuR82]. This scheme has been incorporated in an image shape analysis method directed toward classifying small well-defined regions, such as buildings and airplanes, which has been investigated by Mitchell, Reeves, and Fu [MiR81]. A processing scenario (composed of serial algorithms) which produces interpretation results from digitized imagery using these methods has been implemented at Purdue University on a VAX 11/780. In this application, image sizes are typically 5000-by-5000 *pixels (picture elements).* The image is analyzed in 256-by-256 pixel subimages which are processed independently. To insure that each object (which has a maximum dimension of 127 pixels) will be completely contained within at least one subimage, it is necessary to overlap the subimages.

The serial method of [MiR81] yields good results, but is computationally intensive, incurring long execution times. The time required to complete the processing scenario can be reduced by exploiting its inherent parallelism. In this work, a processing scenario composed of parallel algorithms which allows the problem to be completed with significantly reduced execution time is considered. In addition to decreasing the processing time, the parallel scenario does not place a limit on the maximum size of an object. Once it has been constructed, requirements which the parallel scenario imposes on the architecture of a parallel computer system such as PASM [SiS81a] are studied.

A parallel computer system model is given in Section 6.1.2. In Section 6.1.3 the object shape analysis problem [MiR81] is defined and the parallel scenario is overviewed. In Sections 6.1.4 and 6.1.5 the parallel algorithms which compose the scenario are presented, and they are evaluated in Section 6.1.6. The implications the scenario has concerning system architecture are considered in Section 6.1.7.

6.1.2 An SIMD/MIMD Model

The system model which will be used for implementing the contour extraction task is PASM. *PASM*, a partitionable SIMD/MIMD machine, is a large-scale dynamically reconfigurable multimicrocomputer system being designed at Purdue University [SiM78,SiS81a]. Image processing and pattern recognition tasks are the target problem domain for PASM, and the requirements of these applications are being used to guide design decisions. PASM is intended to be a flexible research machine, and it has more capability than is necessary to cope with the example image processing scenario discussed in this chapter. In particular, PASM's capability to be partitioned to operate as many independent SIMD/MIMD machines of varying sizes is not needed for this scenario.

The rest of this section is a brief overview of PASM to provide background for the following sections. A block diagram showing the basic components of PASM is given in Figure 6.1. The *System Control Unit* is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit (PCU)* contains $N = 2^n$ processors, N memory modules, and an interconnection network. The *PCU processors* are microprocessors that perform the SIMD and MIMD computations. The *PCU memory modules* are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. PASM is being designed for N = 1024. An N = 16 prototype based on Motorola MC68000 processors is planned [KuS82].

The PCU is organized as shown in Figure 6.2. A pair of memory units is used for each PCU memory module so that data can be moved between one memory unit and secondary storage while the PCU processor operates on data in the other memory unit (double-buffering). Each memory unit is of substantial size (e.g., 64K words). A processor and its associated memory module form a *PCU processing element (PE)*. The

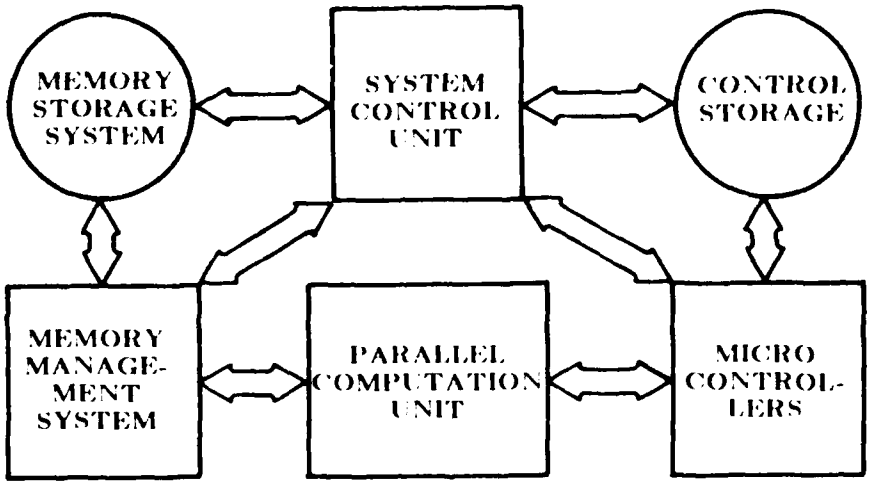


Figure 6.1 Block Diagram Overview of PASM

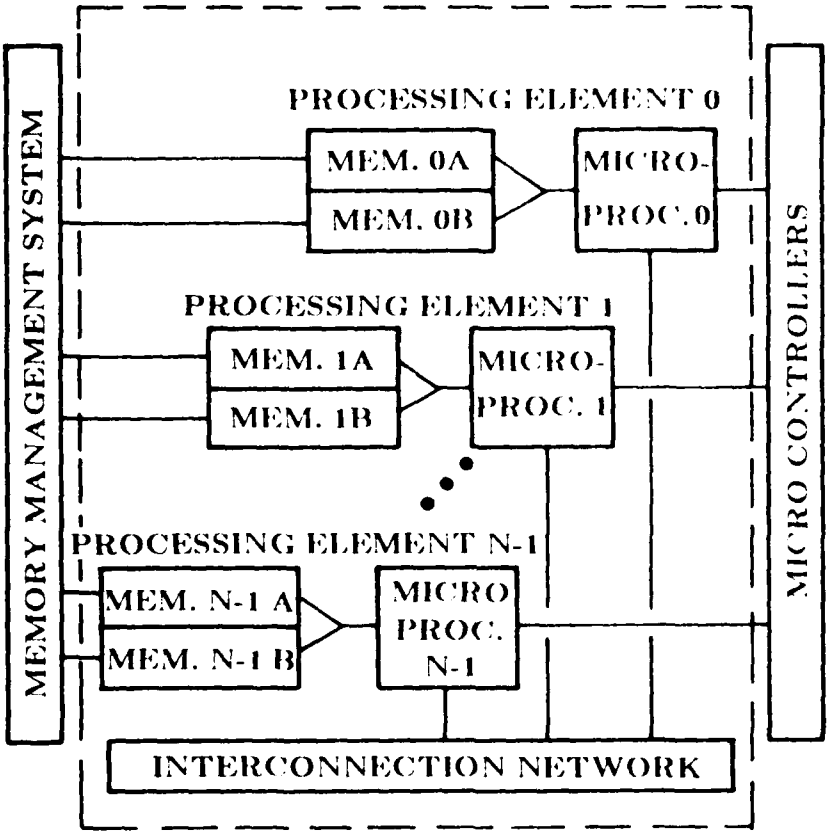


Figure 6.2 PASM Parallel Computation Unit

PCU PEs are addressed (numbered) from 0 to N−1. The *interconnection network* provides a means of communication among the PEs. PASM will use either a Extra Stage Cube type [AdS82b,SiM81b] or Augmented Data Manipulator type [McS82d,SiM81a] of multistage network. The *Memory Management System* controls the loading and unloading of the PCU memory modules from the multiple secondary storage devices of the *Memory Storage System*.

The *Micro Controllers (MCs)* are a set of microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. *Control Storage* contains the programs for the MCs.

6.1.3 Image Processing Task

6.1.3.1 Problem Definition and Serial Algorithms

The first stage of the shape analysis scenario of [MiR81] is to identify boundaries of potential objects using edge-guided thresholding [SuR82]. Edge-guided thresholding (EGT) uses adaptive thresholding to allow contour extraction where gray level variations would not allow global thresholding to be effective. The image is segmented by selecting several gray level thresholds and tracing the resulting contours. Classification is accomplished by comparing the contours with prototype object models using either Fourier descriptors [WaW80] or standard moments [Hu62,Tea80].

An overview of the serial image processing scenario follows (further details are given later in this section). Segmentation is simplest when there is little background information, i.e., the objects of interest cover a significant portion of the image. To achieve this with a very large image, the image can be divided into subimages. A subimage size twice the largest dimension of an object is chosen, and each subimage is processed independently. Subimages are located so that they overlap neighboring subimages 50 percent in both the horizontal and vertical direction. This insures that

an object will be completely contained in at least one block. However, it is necessary to perform the image processing computations four times for each pixel. The advantage of this method is that it eliminates the need to trace contours across subimage boundaries (simplifying the algorithms) and significantly reduces the amount of main memory required (subimages are discarded after processing).

Potential thresholds for a subimage are selected using edge-guided thresholding, which selects thresholds based on an edge-matching criterion. Using the Sobel edge operator [DuH73], an *edge image* is generated in which gray levels indicate the magnitude of the gradient. A figure of merit which indicates how well a given thresholded gray level image matches edges in the edge image is then computed for every possible threshold. Using thresholds with high figures of merit, a requantized version of the gray level image is generated. A median filter [GaW81] may then be applied to remove isolated noise artifacts. The contours for all potential objects not touching the subimage boundary (i.e., completely contained within the subimage) are extracted for further shape analysis. Very short and very long contours may not be retained if they represent objects outside the range of interest. The boundary of each object (contour) is stored as a sequence of x-y coordinates.

6.1.3.2 Parallel Scenario

In this section a parallel formulation of the contour extraction scenario is presented. This parallel scenario will be used as an application example for determining the execution environment which must be provided by the architecture of an SIMD/MIMD parallel processing system such as PASM. The specific context of the contour extraction scenario would depend on the application. The contour extraction scenario may be preceded by image processing such as rectification. Subsequent use of the extracted contours depends on the particular end application. Highlighting contours of an image requires essentially no further processing, while shape analysis and

classification may involve significant additional calculation beyond contour extraction.

An M-by-M pixel image is represented by an array of M^2 pixels, where the value of each pixel is assumed to be an eight-bit unsigned integer representing one of 256 possible gray levels. To implement contour extraction on an SIMD/MIMD machine of 1024 PEs, assume that the PEs are logically configured as a 32-by-32 grid, on which the M-by-M image is superimposed, i.e., each processor has an M/32-by-M/32 subimage (see Figure 6.3(a)). For M = 5120, each PE stores a 160-by-160 subimage. Each pixel is uniquely addressed by its i-x-y coordinates, where x and y are the x-y coordinates of the pixel in the subimage contained in PE i.

Two important parallel algorithms of the contour extraction scenario are edge-guided thresholding and contour tracing. The edge-guided thresholding algorithm, which is discussed in Section 6.1.4, is used to determine a set of optimal thresholds for each subimage. The contour tracing algorithm, which is considered in Section 6.1.5, uses the set of optimal thresholds to segment the image and trace the contours, generating an i-x-y sequence for each contour.

The parallel algorithms described yield a significant reduction in execution time because the multiplicity of processors allows all of the subimages to be processed simultaneously. Since the parallel contour tracing algorithm is able to trace contours over subimage borders, it is not necessary to overlap the subimages, and each pixel is processed only once. The parallel algorithms can result in improved information extraction since the subimages can be smaller (assuming a large number of PEs), yielding a better choice of thresholds within each subimage. In addition, the parallel algorithms do not require an object to be contained in a single subimage.

The parallel scenario could be implemented on a serial computer system with virtual memory [Den70a]. The disadvantage of this approach is that when a contour spans more than one subimage, the linking of partial contours residing in different

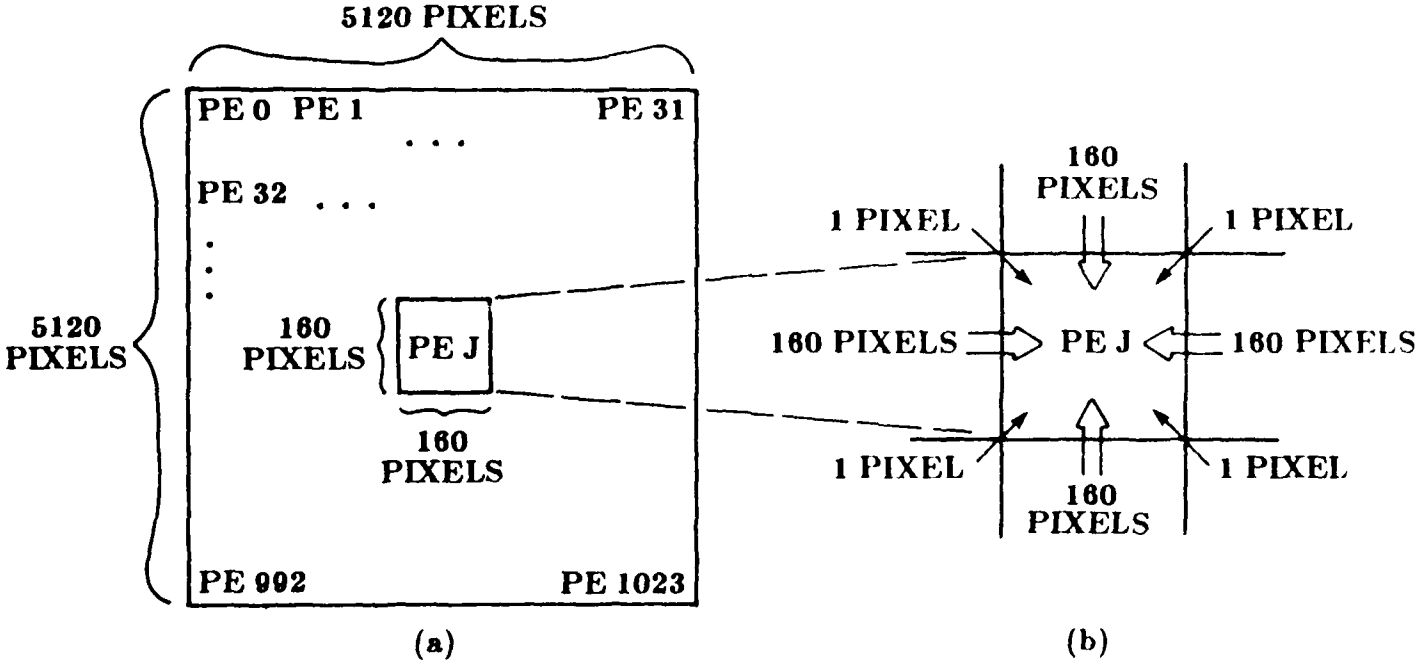


Figure 6.3 (a) Data Allocation for a 5120-by-5120 Image Using 1024 PEs
(b) Data Transfers Needed to Apply Sobel Edge Operator

subimages requires that a representation of the subimages, as well as any contour information, be accessible. This may result in significant delay due to paging subimages into primary memory. Paging overhead does not occur on a parallel system since the entire image is stored in primary memory. Thus, it is the multiplicity of primary memories in a parallel system such as PASM (the large primary memory space) that makes the non-overlapping subimage approach practical.

6.1.4 Edge-Guided Thresholding

The first major procedure of the example scenario is *edge-guided thresholding (EGT)* [SuR82], which is used to identify boundaries of possible objects. Edge-guided thresholding selects threshold levels based on an edge-matching criterion instead of the classical technique of image histogram local minimum values [PrM66]. Frequently, EGT gives better results than the histogram method because it is able to detect small regions not discernibly represented in the histogram [SuR82].

The EGT algorithm operates on each subimage independently, and consists of three major steps. First an edge image is generated. Then a figure of merit is computed for every possible threshold. Finally, local maxima (peaks) in the figure of merit function determine the threshold levels.

The Sobel edge operator is used to generate the edge image in the example scenario. SIMD parallelism is the most advantageous form of parallelism for the Sobel algorithm. This can be shown by analysis of the operator itself. Let the image I be M-by-M and I(i,j) be a gray level image pixel, where $0 \leq i, j \leq M-1$. The Sobel procedure (ignoring image edge pixels for clarity) is the following.

for i = 1 to M−2 do

for j = 1 to M−2 do

$$sx(i,j) = \frac{1}{4}\Big|(I(i-1,j-1)+2*I(i-1,j)+I(i-1,j+1)) - (I(i+1,j-1)+2*I(i+1,j)+I(i+1,j+1))\Big|$$

$$sy(i,j) = \frac{1}{4}\Big|(I(i-1,j-1)+2*I(i,j-1)+I(i+1,j-1)) - (I(i-1,j+1)+2*I(i,j+1)+I(i+1,j+1))\Big|$$

$$g(i,j) = \sqrt{sx(i,j)^2 + sy(i,j)^2}$$

The value g(i,j) represents the gradient at pixel (i,j), and these values form the edge image.

The algorithm is particularly well suited for SIMD parallelism because all pixels are processed identically. This complete synchronization aids the PE-to-PE communication necessary when subimage border pixels within each PE must be processed. In the case of this algorithm, transmission delays incurred due to PE-to-PE data transfers can be overlapped with data processing to reduce total execution time. All PEs will simultaneously request the same border pixel relative to their subimages. For example, when processing begins (with the upper left corner subimage pixel) all PEs will request (from the PE to their upper left) the pixel immediately above and to the left of their upper left corner pixel (if this pixel is within the complete image). This transfer of data from upper left neighbors can occur for all PEs simultaneously. A total of 4*(160 + 1) = 644 parallel transfers are needed for a 5120-by-5120 pixel image, as shown in Figure 6.3(b). The candidate interconnection networks for PASM can support these parallel transfers from any neighboring PE. The result of the Sobel operator is the edge image. High edge image pixel values indicate the presence of an edge.

The next step of the EGT algorithm is to compute a figure of merit value for each possible gray level. The figure of merit is a measure of how well the edges generated by a given threshold match the edges detected by the Sobel operator. Specifically, the figure of merit is determined as follows.

1. The local maximum and minimum pixel values over a 3-by-3 window are determined for each gray level image pixel.
2. For each possible threshold value (i.e., all gray levels) the center pixel of the 3-by-3 window is tested to see if it is an *edge point*. It is an edge point if the threshold is greater than or equal to the local minimum and less than the local maximum.
3. The mean of the edge image pixels corresponding to the gray level image pixels found to be edge points at a given threshold is the figure of merit for that threshold.

The figure of merit calculation has portions suited to both SIMD and MIMD parallelism. Steps 1 and 2 can be done efficiently in SIMD mode since all pixels are processed similarly. Step 3 is executed only on the gray level image pixels which are edge points. To do this, the PEs operate in MIMD mode, each sequencing through the edge points in its subimage. Since the number of such pixels may vary, some PEs may complete Step 3 before others.

The greater the mean of the edge points in Step 3, the better the match between threshold-generated boundaries and the edges detected by the Sobel operator. To avoid the assignment of a high figure of merit to a small number of noise pixels, a bias can be added to the denominator when calculating the mean. This has the effect of lowering the figure of merit if only a small number of pixels are above the threshold. The gray levels associated with local maxima (peaks) in the figure of merit function are chosen for image segmentation. Typically, three to six levels are chosen. The next step of the

scenario is contour tracing.

6.1.5 Contour Tracing

In this section an approach to performing contour tracing using MIMD parallelism is presented. Initially, each PE contains a list of threshold values, $\{T_1,T_2,...,T_t\}$, for its subimage which have been selected using edge-guided thresholding. The number of thresholds for any given PE is denoted by t and can differ for each PE. The contour tracing algorithm has two phases. In Phase I, the subimage is segmented within each PE and all local contours (both closed and partial) are traced and recorded. In Phase II, the partial contours traced during Phase I are connected.

A contour table is constructed in each PE containing an entry for every contour, whether partial or closed, which is located in the subimage associated with that PE. Each contour table entry contains the following fields: (a) a contour identification number, (b) the threshold value which generated the contour, (c) the number of pixels in the contour, (d) a flag indicating if the contour is closed or partial, (e) a pointer to the array containing the i-x-y sequence of the contour, (f) a flag indicating whether the partial contour has been connected (for use in Phase II), (g) the physical address of the PE which linked the contour, (h) the physical PE address and identification number denoting the partial contour blocking extension of the contour, and (i) a locked/unlocked semaphore. Contour table entries g, h, and i are discussed below. Each PE also contains a ***partial contour list***. This list has an entry for each partial contour containing the i-x-y coordinates of its two end points and a pointer to its contour table entry.

In Phase I there is no PE-to-PE communication. Each PE considers its threshold values T_i, $1 \leq i \leq t$, independently. Its subimage is segmented using each threshold level T_i. To create the segmented image for threshold T_i, pixels in the original image

which have a value greater than or equal to T_i are assigned a value of one, while those which are less than the threshold are assigned a value of zero.

The rows of the segmented image are scanned beginning with the top row. Scanning stops when a pixel with value one is found which has a zero-valued neighbor to either side. This pixel is marked as the *start point* of a new contour, and its i-x-y coordinates are stored. Consider this pixel as the center pixel of the 3-by-3 window in Figure 6.4. The contour is traced in a counterclockwise direction generating a sequence of i-x-y coordinates. Beginning with the neighboring pixel in position five (see Figure 6.4) and incrementing by 1 modulo 8 to determine the next pixel, the algorithm looks for a pixel which has a value of one. The algorithm stores the direction, p, of this new pixel and appends its i-x-y coordinate to the contour sequence. Treat this new pixel as the center point of the 3-by-3 window in Figure 6.4. The algorithm then looks for the next pixel in the contour beginning with the pixel in position (p + 5) modulo 8 (to produce a counterclockwise trace). Tracing continues until the start point or a point of indecision is reached. If all of the neighbors of a start point are zero, that pixel is an isolated point and is ignored.

A *point of indecision* occurs when information from an adjacent subimage is required to determine the direction of the contour. When a point of indecision is reached, it is recorded as an *end point,* and the algorithm returns to the start point to trace the contour in a clockwise direction until another point of indecision is reached. When tracing in the clockwise direction, the new contour pixels are inserted onto the front of the i-x-y sequence. Each pixel in the contour is marked in the thresholded image so that the contour will not be retraced.

Consider the following contour tracing example based on Figure 6.5. A 10-by-20 image is divided into two 10-by-10 subimages; each subimage is loaded into one of two PEs. The local threshold value T_1 is applied to the subimage in each PE. Each PE i

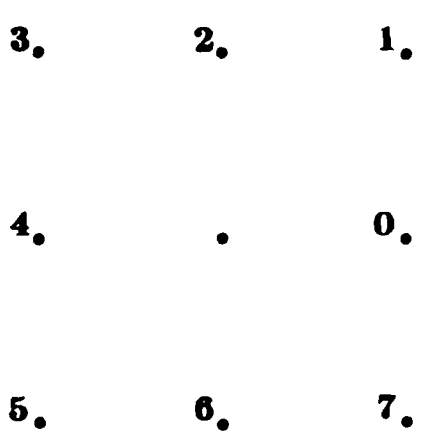


Figure 6.4 Naming Convention for the Neighbors of the Center Pixel in a 3-by-3 Window

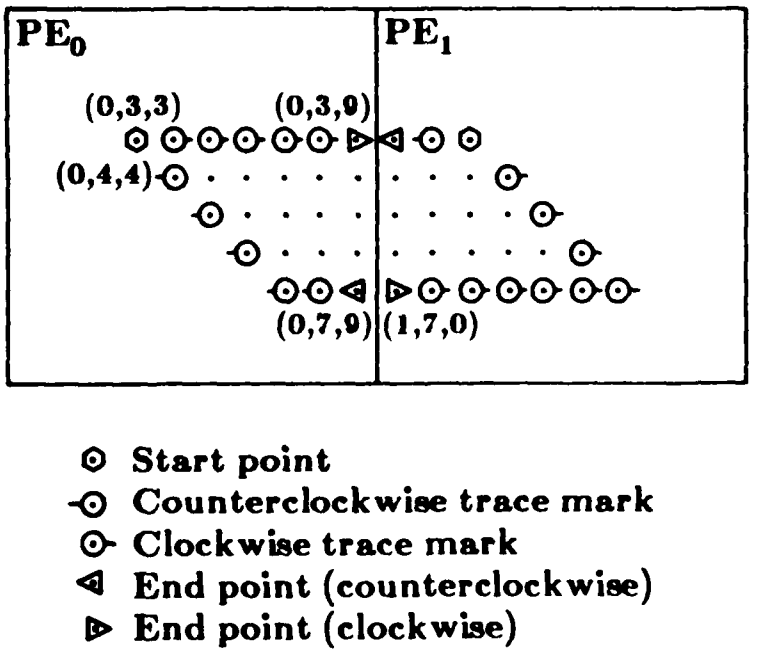


Figure 6.5 Example of Phase I Contour Tracing for a 10-by-20 Image

begins scanning its respective subimage at pixel (i,0,0), for a one (indicated by a dot) with a zero on either side. PE 0 locates the edge of a segmented object at pixel (0,3,3). Pixel (0,3,3) is the start point for the new contour. PE 0 traces the contour of the object counterclockwise to a point of indecision at pixel (0,7,9), which is recorded as an end point. Pixel (0,7,9) is a point of indecision since pixels (1,6,0), (1,7,0), and (1,8,0) of the subimage in PE 1, which could extend the contour, are not in the subimage contained by PE 0. PE 0 then traces the contour in the clockwise direction beginning at pixel (0,3,3), reaching a point of indecision at pixel (0,3,9). After the clockwise trace, the first pixel in the i-x-y sequence describing the contour is (0,3,9). PE 0 resumes scanning at pixel (0,3,4) and finds no other contours in its subimage. Note that, for example, pixel (0,4,4) is not a start point for a new contour since it was marked during the trace of the first contour. Similarly, a partial contour is located in PE 1 with (1,7,0) as the first pixel in its i-x-y sequence. Once a PE has scanned the segmented image generated by threshold T_i, *it repeats the process for threshold* T_{i+1}. After all threshold values in a PE have been considered, Phase I is complete.

In Phase II, each PE attempts to connect its partial contours to partial contours which are located in neighboring PEs. There are two alternatives for determining when a PE can enter Phase II. With the first, PEs are allowed to start Phase II processing *after all have completed Phase I. With the second*, a PE enters Phase II immediately after completing Phase I. However, it can only attempt to extend contours into subimages of PEs which are also in Phase II. If all neighboring PEs are still in Phase I, the PE must wait. The latter approach may reduce the total scenario execution time since the PE with the longest Phase I time may well not be the one with the longest Phase II time. The first alternative requires time equal to the sum of the longest times in each phase.

Since multiple PEs can contain portions of the same contour, there must be a rule to determine which PEs have priority to attempt to close a contour. The rule is each

PE attempts to extend only its partial contours which have both end points bordering subimages to the left and/or above. For example, in Figure 6.6, partial contours A, B, C, and D are considered by the PE, while E, F, and G are not. For each given partial contour (generated by a threshold T_i), the PE attempts to extend it into the neighboring PE from the counterclockwise end point (as described below).

In order for a PE to extend a contour, it must be able to access and modify contour tables which are located in other PEs. As a result, a mechanism to prevent one PE from using a contour table entry while another PE is in the process of using that entry must be provided by the system and used by the contour tracing algorithm. Any section of code which modifies a contour table entry is a *critical section* [Dij68]. The only table entry fields which can be modified by another PE are the flag which indicates if the partial contour has been connected and the physical address of the PE which linked the contour (fields (f) and (g)). While a critical section is being executed on a given table entry, that entry is locked, so no other processor can modify it.

A *semaphore* is a variable whose value indicates whether or not a critical section can be entered [Dij68]. There is a semaphore for each contour table entry which can take on a value of zero or one. Before a PE enters a critical section (for a given contour), the processor performs a *P-operation* [Dij68] on the given contour to determine if it is unlocked. If the semaphore for the contour table entry is one, the processor sets the semaphore to zero (locking the contour table entry so that no other processor can access it) and enters the critical section, free to modify the contour table entry. When the processor completes modification of the contour table entry (i.e., the critical section ends), it performs a *V-operation* [Dij68] on the semaphore for the contour, setting the semaphore to one. The contour table entry is then unlocked. On the other hand, if the semaphore is initially zero, the processor receives a message indicating that the partial contour is locked.

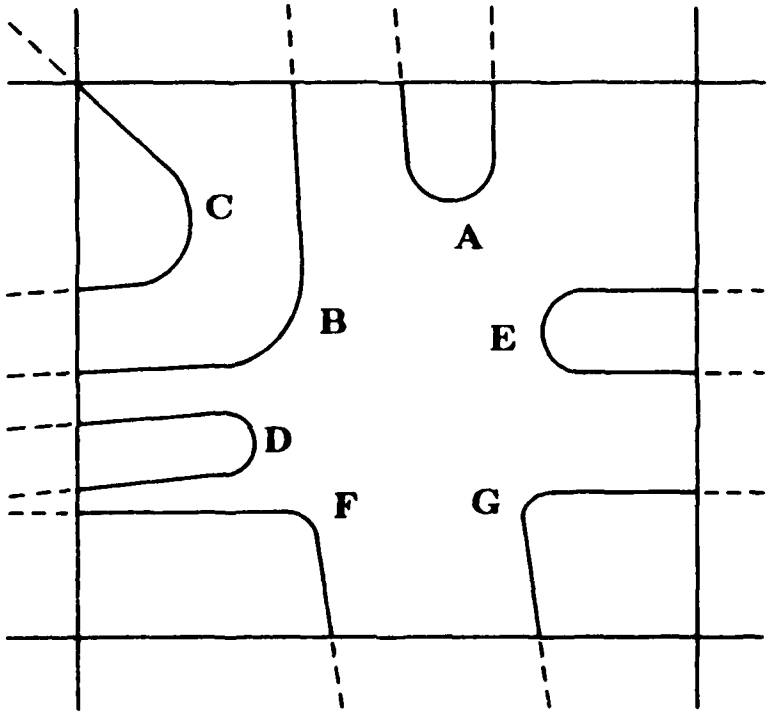


Figure 6.6 Phase II Connection Precedence. Partial contours A, B, C and D are considered by the PE; E, F, and G are not.

If the end point of a given partial contour is not at a corner of its subimage, there are three pixels, located in the adjacent subimage, which can possibly extend the contour. The PE accesses the partial contour list for the adjacent subimage (see Section 6.1.7). Considering the possible extending pixels one at a time in counterclockwise order, the PE checks the partial contour list to determine if any partial contours in the adjacent subimage have the possible extending pixel as an end point. If such a partial contour exists, the PE performs a P-operation on the contour table entry pointed to by the partial contour list. If the contour was unlocked, the i-x-y sequence for the contour is transferred (discussed in Section 6.1.7) to the PE containing the given partial contour and then concatenated to its i-x-y sequence, forming a new, extended partial contour. If there is more than one partial contour with the same end point which can extend the given contour, the partial contour which was generated by a threshold value closest to that for the given contour is selected.

If the end point of a given partial contour is a corner point of its subimage, there are five pixels located in adjacent subimages which can possibly extend the contour. Since these five pixels are located in three different subimages, the PE attempting to extend the given partial contour must check for continuation in each of the upper-left adjacent subimages (in a counterclockwise order).

Note that regardless of where partial contour end points lie, the search for pixels to extend the contour can be widened beyond the three or five pixels here to allow for threshold value discontinuities at subimage boundaries. Thresholds could be interpolated across subimage boundaries to allow partial contours with non-adjacent end points to be joined.

Assume that PE i has a partial contour which it is responsible for extending. If a continuation of the partial contour is not found in the partial contour list for the adjacent subimage, PE i probes into the adjacent subimage to determine if an extension of

the partial contour can be generated by the threshold, T, it (PE i) used to trace its partial contour. If so, PE i extends its partial contour by accessing the data from the adjacent PE. Instead of creating an entire segmented subimage for the threshold T, PE i dynamically thresholds pixels as needed. This contour generation using T is done since it is possible that the partial contour in the adjacent PE was not located in Phase I because different threshold values were used, or the contour fell along the edge of the subimage (see the split between PEs 2 and 3 in Figure 6.8).

Once PE i locates a partial contour in an adjacent subimage which continues the given contour and has stored the concatenated contour in its contour table, it repeats the process, if necessary, by following the contour to the next PE until the contour is closed or cannot be extended. A limit is placed on the maximum contour length to guarantee algorithm termination in the event of a pathological image.

Consider the example in Figure 6.7 where a 12-by-12 pixel image is divided between four PEs. After Phase I, PE 0 contains partial contours A with end points (0,1,5) and (0,5,1) and B with end points (0,4,5) and (0,5,3); PE 1 contains partial contour C with end points (1,4,0) and (1,1,0); and PE 2 contains partial contour D with end points (2,0,1) and (2,0,3). Since both end points for contour C border the subimage to the left, PE 1 attempts to extend contour C in Phase II. Similarly, PE 2 attempts to extend contour D since its end points border the subimage above.

PE 1 attempts to extend C in the counterclockwise direction, i.e., from pixel (1,1,0). It first locks its contour table entry for C. It then examines the contour table of PE 0 and determines that A can be linked to C. If the table entry for A is unlocked (i.e., the semaphore value is zero), PE 1 locks it (performs a P-operation) and appends the i-x-y sequence of A to the i-x-y sequence of C. It also sets the flag which indicates that A has been linked and records that PE 1 performed the linkage.

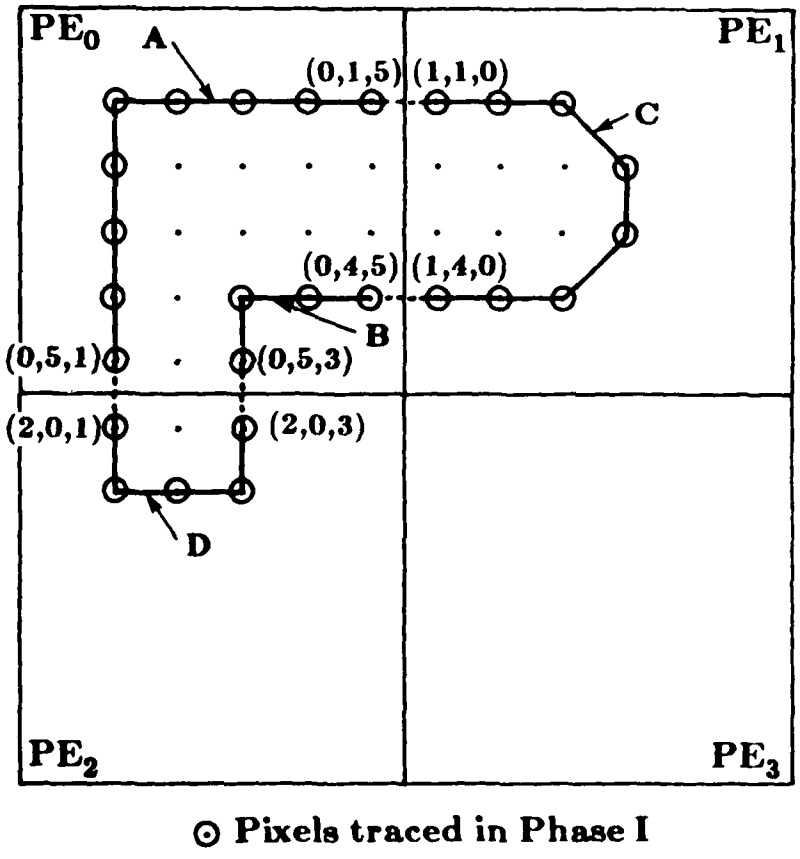


Figure 6.7 Example Where Two PEs Attempt to Connect the Same Contour

Independently of the actions of PE 1, PE 2 attempts to extend contour D (from pixel (2,0,3)). As did PE 1 with A, PE 2 appends B to D. If PE 2 attempts to extend the result, DB, while PE 1 is in the process of extending C into PE 0, it will find C locked. PE 2 then abandons its attempt to close the contour, since PE 1 is also attempting to do it, and unlocks partial contour DB. This allows PE 1 to access DB after it has appended A to C. Therefore, the closed contour CADB is ultimately traced completely and stored by PE 1. If PE 1 had completed linking A to C before PE 2 completed linking B to D, the closed contour would have been completely traced by PE 2. Deadlock is the situation when each of two or more PEs are halted while waiting for the other(s) to continue [Sto80]. If a PE is blocked due to a lock then (1) not allowing a PE to wait for access to a locked contour table entry of another PE, and (2) requiring the blocked PE to unlock its affected partial contour prevents deadlock.

If PE 1 and PE 2 had completed their first linking operation ***simultaneously***, both would have abandoned tracing the contour (i.e., no PE would link the contour CADB). To insure that the linking of a contour will not be abandoned by all PEs, the following protocol is used. Assume PE i is blocked from extending a contour X by PE j, which has higher positional precedence (i.e., $i < j$). In that case, PE i unlocks contour X and sends a message informing PE j that PE i has abandoned its attempt to further extend contour X. If PE j had also abandoned the contour, this message would cause PE j to try again. The message sent from PE i to PE j contains the identification number of contour X and the value i. After receiving the message, PE j searches its contour table to determine if it abandoned X. To do this it uses field (h) of the contour table. For the above example PE 2 would link the partial contours since it has higher precedence.

Deadlock with multiple contours cannot occur since each PE considers only one contour at a time and does not abandon the attempt to extend that contour until that PE has closed the contour or has relinquished control to another PE to close that contour.

When Phase II of the algorithm is complete, the i-x-y sequence for each contour in the image will be contained in exactly one of the PEs which contained part of the contour originally. In the example given in Figure 6.5, PE 1 will contain i-x-y sequence for the contour.

As a final example, a 30-by-20 image is divided into six 10-by-10 subimages; each subimage is loaded into one of six PEs. In Figures 6.8 and 6.9 the results of Phase I and II processing are shown, respectively. Even though the entire object in PE 5 was located within the subimage, the left edge of the object was not traced in Phase I since PE 5 could not determine whether the object continued into the next subimage. On the other hand, a closed contour was found in Phase I for the object in PE 4 since the object did not include any border pixels of the subimage.

6.1.6 Algorithm Evaluation

Serial algorithm subimage size is chosen to be twice the maximum allowed object dimension so that overlapping of subimages guarantees that each object appears in its entirety in some subimage. With this property, partial contours never need to be considered; all objects are found as closed contours within a subimage. The advantage of the serial approach (Section 6.1.3.1) over the parallel approach (Section 6.1.3.2) is that partial contour extension is not necessary. The disadvantages of the serial approach when compared to the parallel approach are threefold. First, the maximum size of an object of interest must be established so that subimage size is known. This choice is constrained by the fact that EGT performance tends to degrade with increasing subimage size. Thus, there is a practical limit on the maximum object size. Second, each pixel is processed for contour extraction four times. Finally, thresholding (including EGT) tends to perform less well when objects are small relative to the image (in this case, subimage) size. The parallel algorithms do not limit maximum object size, process each pixel just once, and may improve threshold accuracy by allowing ready use of

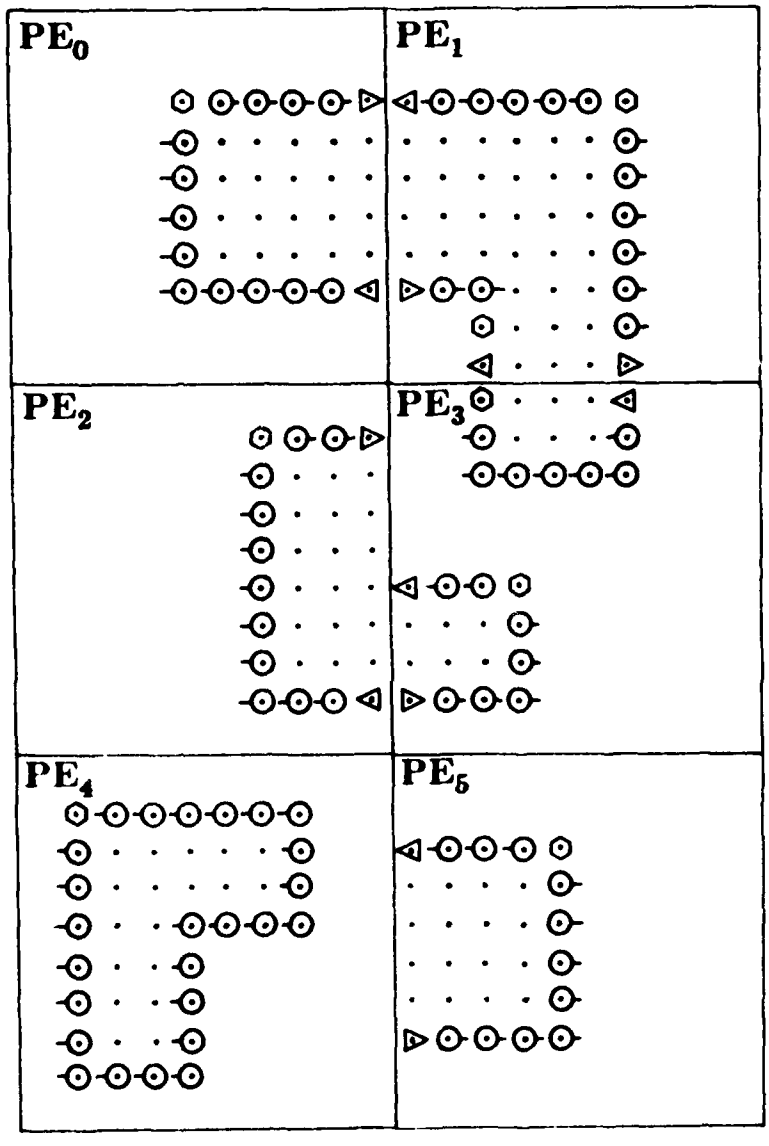


Figure 6.8 Results of Phase I of Contour Tracing for a 30-by-20 Subimage

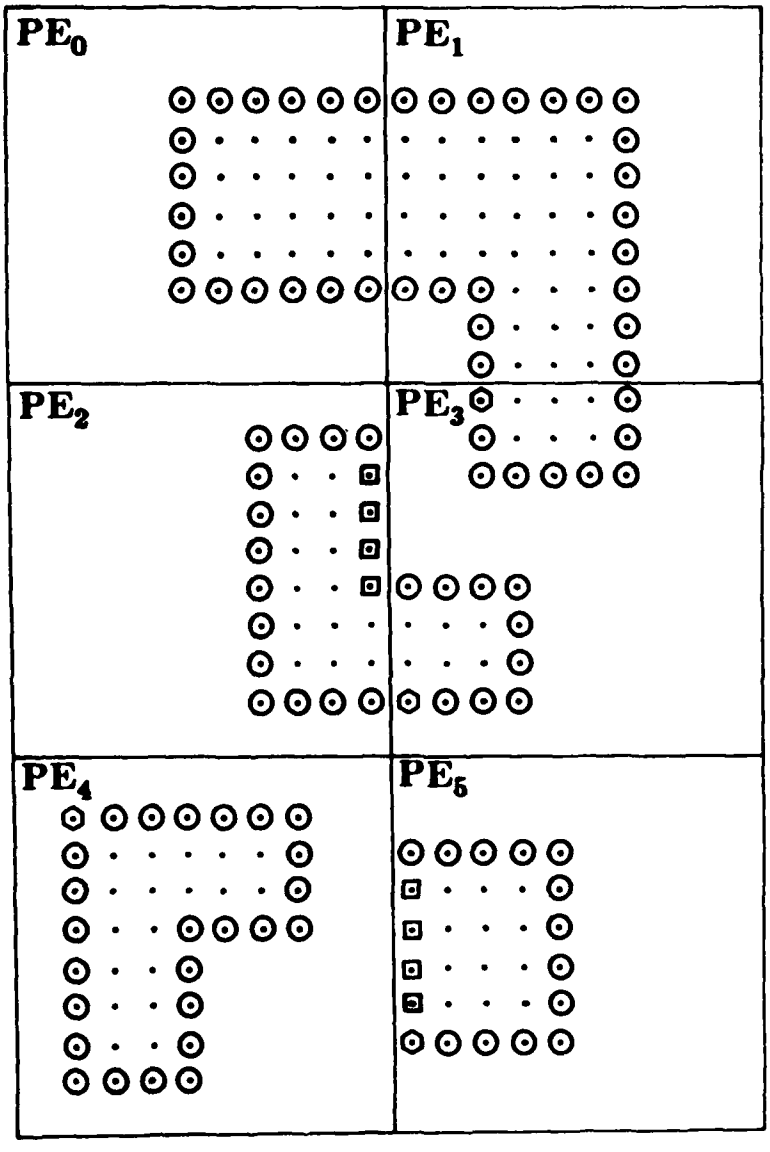


Figure 6.9 Results of Phase II of Contour Tracing for a 30-by-20 Subimage

small subimages. Thus, parallel systems can allow the full benefits of adaptive thresholding via EGT to be more readily realized.

Speedup is the usual rationale for employing parallel processing techniques, and the example parallel scenario has the potential for significant speedup. However, the speedup is data dependent. This is because the PE workload may be highly varied during contour tracing due to uneven distribution of contours throughout the image being processed. While it may be possible to implement load sharing for this portion of the scenario (with certain overhead costs), inequities reducing actual speedup are almost certain to remain.

Overall, the parallel algorithms presented are strong contenders to replace serial methods in some applications. One such is quality control inspection of printed circuit boards. In this application, large object handling capability is needed for following long circuit traces, and sufficient speedup is necessary for timely response. Other applications involve military environments where real-time processing is crucial.

6.1.7 Architectural Implications

The study of a parallel formulation of an image processing scenario involves both the design of individual parallel algorithms and the determination of a method to integrate them into a single job. This leads to an understanding of necessary and useful hardware attributes for a parallel machine intended to execute that scenario. For the example scenario, aspects of each algorithm which have an architectural impact other than those pertaining to the processors will be listed. Processor specific considerations (e.g., instruction set) are not treated because they are similar for serial and parallel machines.

The Sobel edge detection algorithm step of EGT requires data that is, by vast majority, local to each PE. When non-local data is required, nearest neighbor PEs

comprise the set of data sources. Local maxima and minima calculation on 3-by-3 windows mimics the characteristics of the Sobel operator, but with more memory references. Edge point detection is similar in these regards to the previous steps.

The figure of merit calculation for EGT is different in kind from the previous steps. Only local data is required, and processing time is data dependent. MIMD operation is preferable to SIMD, even if edge point detection and figure of merit calculations are merged into a one-pass operation.

Phase I of contour tracing requires only local data, but execution time is data dependent. Phase II makes heavy use of non-local data and has data dependent execution time. Both phases are suited to MIMD mode.

Now the architectural requirements for a parallel machine performing the example scenario can be considered. Probably the most basic need for the system if it is to support the scenario well, is to be capable of dynamically switching between SIMD and MIMD operation, as can PASM. With only SIMD capability, vast inefficiency would occur in later stages of the scenario. Having only MIMD mode is a less serious handicap, but will lengthen execution time for the Sobel operator and determining local maxima and minima, due to the need for explicit synchronism and data sharing. Thus, the capability to dynamically switch between SIMD and MIMD modes is important so that each subsequent portion of the scenario can be executed in the most appropriate operational mode.

An interconnection network is needed to perform permutations involving eight nearest neighbors in SIMD mode. In MIMD mode, it is used for eight nearest neighbors and for somewhat arbitrary one-to-one connections (when transferring partial contour information between non-adjacent PEs). Both types of connection needs must be performed efficiently by the network. The networks proposed for PASM can do so.

The PE-to-PE transfer of information must be efficient, or the parallel algorithms will be slowed. One method to perform PE-to-PE communication is by using direct memory access (DMA). DMA is a method for storing or retrieving data without processor intervention. There are several ways to implement this capability. In one, a PE extending a partial contour sends an interrupt to the remote PE containing the extension of the partial contour along with the identifier of the needed partial contour. The remote PE then enters a DMA handling routine. This routine computes the local memory address range of the requested partial contour i-x-y sequence and sends this information along with the requesting PE number to special DMA hardware. The DMA hardware then autonomously retrieves the information from local memory and performs necessary network interfacing to send the data to the requesting PE. DMA hardware accesses to local memory can be via cycle stealing. Another implementation of DMA capability is through an intelligent network interface unit (NIU). Requests for data from remote PEs would be received, interpreted, and discharged by the NIU without local PE processor intervention. The NIU would combine DMA capability with network protocol support. VLSI technology may allow ready fabrication of sophisticated NIUs. Thus, such a DMA capability would be worthwhile to include in a system such as PASM.

6.1.8 Summary

Considering an entire scenario in the light of parallelism is a useful approach for matching image processing tasks and parallel architectures. A number of observations were made and conclusions drawn from the example image processing scenario. In particular, the parallel scenario was found to embrace both SIMD and MIMD subtasks, involve significant PE-to-PE data transfer, and contain both nearest-neighbor and non-adjacent PE communication patterns. Parallel formulation of the algorithms lead to several advantages including speedup, elimination of object size constraints, and

potential for improved accuracy.

These observations indicate that parallel contour extraction could be useful in industrial inspection and military applications. They suggest desirable system architecture features, including SIMD/MIMD capability with dynamic mode switching, dedicated PE-to-PE communication support hardware, and arbitrary PE-to-PE interconnection capability. These requirements are consistent with the capabilities of PASM.

This study is an initial effort towards relating parallel computation characteristics and parallel architecture features to develop an ability to evaluate alternative architectures relative to classes of computational processes. Contour extraction is a key component of many image processing tasks. The results and insight described in this chapter will be extended by drawing together our work in Fourier descriptors. This will give a more varied image processing task from which to extract more detailed process models and guide architecture model evolution.

6.2 Fourier Descriptors

6.2.1 Introduction

Fourier descriptors [WaM79, WaM80] have been proposed as a method of performing shape analysis for applications such as object recognition and tracking. The task of computing normalized Fourier descriptors has been structured for MIMD execution. The objective of this work is to identify the characteristics of the Fourier descriptor task which influence its parallel implementation, and the attributes of the parallel architecture which best supports the task.

6.2.2 Algorithm Overview

Given the contour of an object in a two dimensional plane as input, a series of frequency domain coefficients are computed which describe the image. These are the Fourier descriptors, which are further processed in a normalization procedure so that they can be compared to a library of these descriptors. The end effect is the identification of the object as well as a reasonable estimate of its orientation in space [WaM79]. The algorithm has been proven effective in identifying and tracking aircraft in flight [WaM80].

Input to the program consists of the chain code representation of the contour of an image in a two dimensional plane. This chain code input is then converted to X-Y coordinates of the image. After optional smoothing, the image is resampled at equally spaced intervals on the contour. If an FFT transformation is later employed, this resampling must be done with the number of samples equal to a power of two. Then a complex Fourier transform is performed on the resampled points. This produces a Fourier descriptor (FD).

The second logical division of the algorithm normalizes this descriptor. The goal is to scale and orient the contour by rule such that the FD from an unknown contour will always normalize to the correct library representation. Different normalizations have been proposed, but Wallace's algorithm [WaM79] is the one investigated here. The normalization is accomplished as follows: The H most significant complex coefficients of the FD (H is typically 32), are denoted as $A(-\frac{H}{2} + 1)$ through $A(\frac{H}{2})$. This frequency domain representation of the contour is normalized by taking out information relating to the relative position of the contour, its size, its starting point, and its orientation. This is accomplished by three steps

Step 1: Set A(0) = 0.

This takes out all "DC" positional information.

Step 2: Divide A(i) by $|A(1)|$, $-\frac{H}{2} + 1 < i < \frac{H}{2}$

This normalizes the size of the image such that $A(i) \leq 1$.

Step 3: Multiply the A(i) by $e^{j[(i-k)u + (1-i)v]/(k-1)}$

k is the coefficient with second largest magnitude (next to A(1)).

u and v are the phases of A(1) and A(k) respectively.

k is the coefficient with the second largest magnitude.

This simultaneous application of the rotation and starting point shift operations finds one of the normalizations satisfying u = v = 0. If k = 2, this normalization is unique. Otherwise the phase and starting point of the normalization must be shifted to account for the $|k - 1| - 1$ other possible normalizations. Then the correct normalization must be chosen based on some other criteria. The criterion examined here chooses the correct normalization as the one which maximizes

$$\sum_{i=-\frac{H}{2}+1}^{\frac{H}{2}} Re[A(i)]\,|\,Re[A(i)]|$$

This algorithm will be examined closely for parallel constructs. To accomplish this, it will be divided into distinct tasks, and each examined individually. To achieve further parallelism, the tasks could be pipelined to increase real-time throughput. This global parallelism will not be investigated at this time. The major emphasis will be on describing parallel implementations of each section and discovering what features of a parallel architecture would most affect the performance of each task.

Some basic assumptions about the architecture are made. The primary emphasis is on MIMD systems although SIMD systems will be considered as a component of a larger MIMD system. Thus the major thrust will be with MIMD systems. Interprocessor communication networks will be investigated if they affect execution speed. Local memory is a commonly proposed means for reducing memory conflicts, and thus will be examined when it is relevant. Access to a global memory system will also be considered.

6.2.3 Decomposition into Parallel Algorithms

In this section, parallel algorithms are described for each of the subtasks required for generating normalized Fourier descriptors. The algorithms are for conversion from chain code to X-Y coordinates, filtering, resampling, Fourier transform calculation, and FD normalization.

1. Input Conversion

For now, it will be assumed that the contour of the image is entered in chain code representation (see Figure 6.10). The location of point p_i is dependent upon the points

p_0 through p_{i-1}. This presents immediate problems for parallel implementation. Also, the algorithm requires equally spaced sampling along the contour, which is not the case in general since chain code line segments may have lengths differing by a factor of $\sqrt{2}$ (see Figure 6.10).

Representation	Meaning	Length
0	right 1 unit	1
1	right 1 unit, up 1 unit	$\sqrt{2}$
2	up 1 unit	1
3	left 1 unit, up 1 unit	$\sqrt{2}$
4	left 1 unit	1
5	left 1 unit, down 1 unit	$\sqrt{2}$
6	down 1 unit	1
7	right 1 unit, down 1 unit	$\sqrt{2}$

Figure 6.10. Typical chain code input

Also, it may be desirable to resample the contour so that the number of samples is a power of two. This is a requirement if a FFT is used in the next step. Chain code inputs of practical use contain from a few hundred to a few thousand points. This is typically a variable number which depends on the relative size, shape, and perspective of the object being identified. Define the number of chain code inputs as C, and the number of processors as P. The effects of these parameters on processing speed will be investigated given a suitable parallel algorithm.

Chain code input is inherently serial since each input is merely an offset from the previous input. Consider a parallel algorithm for this normally serial task. Initially,

assume that any processor has access to all memory. Also, initially assume that $C = P^2$. Thus, the C inputs can be divided among $\sqrt{C}$ processors and each processor will be responsible for $P = \sqrt{C}$ chain code inputs. This is illustrated for an example having P=5 and C=25 in Figure 6.11. The contour consists of input points XY(i), $0 \leq i < 25$.

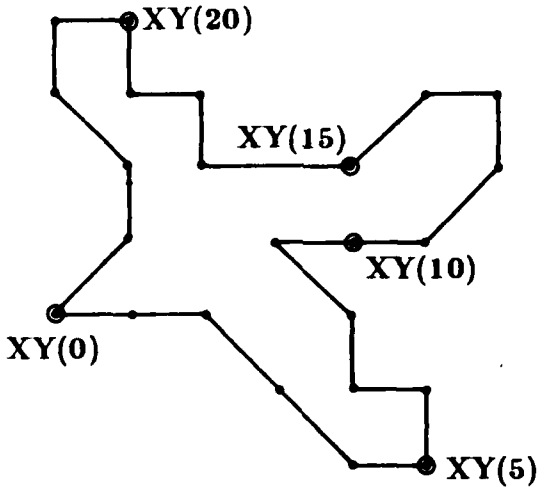


Figure 6.11. Example of a 25-point Contour Divided into 5 Segments, Corresponding to C=25, P=5

The inputs can be viewed as logically forming a two dimensional array as shown below. The array of input points will be denoted as CCIn(0..C-1).

CCIn(0)	CCIn(1)	CCIn(2)	···	CCIn(P−1)
CCIn(P)	CCIn(P+1)			·
CCIn(2P)				·
·				·
·				·
·				·
CCIn((P−1)P)	·	·	·	CCIn(P^2−1)

We can then define parallel operations in which each processor acts on a row or column of this array. The parallel algorithm follows.

Initially each row of the input is processed by a separate processor. This is equivalent to dividing the contour into P continuous pieces and giving each processor a piece to work on. Each processor can then assume that it has the "first" section of the contour and assign the first point the coordinates (0,0). It can then compute the X-Y coordinates of the rest of its points starting from this reference. Given P chain code inputs in each segment, each processor assumes the first point, then generates X-Y coordinates for P additional points. Thus, the last point generated in $processor_p$ corresponds to the first point for $processor_{p+1}$ (the point assumed to be (0,0)). With this completed, X-Y coordinates for all the input points have been generated. However, each row of the array (each segment of the contour) has a different origin in the X-Y plane.

Now a correction step may be employed. Denote the X-Y coordinates of an input point as XY(0..C-1). Since the origin is arbitrary, set it at the point XY(0), that is XY(0) $\equiv$ (0,0). Then our previous step correctly computed the coordinates of XY(0) through XY(P). To correct the coordinates of XY(P) through XY(2P) in the second segment, add to each of these the coordinates of XY(P) computed in the first segment. Then to correct points XY(2P) through XY(3P) in the third segment, add on the (newly corrected) XY(2P) from the second segment. This correction must be done in order, for each row of the square or each continuous segment of the contour. To clarify this, examine a program segment aimed at computing XY coordinate from chain code inputs. Assume this is executing concurrently on P processors and that there are C input points where $C = P^2$. Assume the existence of the functions CtoX() and CtoY() which convert one chain code input to the proper increment in the X or Y coordinate according to a conversion table. Let **SYNC** be a synchronization instruction which insures that all processors have completed up to this step. Finally, each processor knows its "number" p ($0 \leq p \leq P - 1$). The algorithm is expressed in Flock Algol, an algorithmic language proposed in [SiS81b]. The algorithm is executed asynchronously

in each of the P processors.

```
/* Global variable definitions */
/* All processors have access to these variables */
CCIn(0..C-1)   /* Chain code input */
X(0..C-1)      /* X coordinates */
Y(0..C-1)      /* Y coordinates */

/* Local variable definitions */
/* Each processor has a separate copy of these variables */
RowStart       /* Starting index */
index          /* Computed index */
p              /* Initialized to the processor number */
Col            /* Column to work on in correction step */

RowStart←P*p

/* Each segment starts at the origin */
X(RowStart)←0
Y(RowStart)←0

/* Compute X-Y coordinates from this origin */
FOR i←0 THROUGH P−2 DO
BEGIN
        index←RowStart+i
        X(index+1)←X(index)+CtoX(CCIn(index))
        Y(index+1)←Y(index)+CtoY(CCIn(index))
END

SYNC
/* After synchronization, compute the first point of the next segment */
index←RowStart+P−1
X(index+1)←X(index)+CtoX(CCIn(index))
Y(index+1)←Y(index)+CtoY(CCIn(index))

SYNC

Col←p+1
RowStart←0

/* Correct each point according to offset from actual origin */
FOR i←1 THROUGH P−1 DO
BEGIN
        X[RowStart+Col]←X[RowStart+Col]+X[RowStart]
        Y[RowStart+Col]←Y[RowStart+Col]+Y[RowStart]

        RowStart←RowStart+P /* Move to the next row */
        SYNC
END
```

In order to estimate the amount of computation performed, some assumptions about the number and types of statements will be made. In the first **FOR** loop, the X and Y increments are chosen based on a test of the input value. This will be considered one "test." The assignment of the next X and Y outputs will be assumed to require one complex add and one complex assignment. In the second **FOR** loop, each step consists of one complex addition. The original serial scheme consisted of C tests, C complex additions, and C complex assignments. The parallel approach executes in the time for $\sqrt{C}$ tests and $\sqrt{C}$ assignments and $2\sqrt{C} - 1$ complex additions. Assuming the dominant operation is the additions, the speedup is approximately given as

$$S \simeq \frac{C}{2\sqrt{C} - 1} \simeq \frac{C}{2\sqrt{C}} = \frac{1}{2}\sqrt{C}\,.$$

Since $P \equiv \sqrt{C}$ in the ideal case, then

$$S \simeq \frac{1}{2} P$$

The tests were not included in the count, and the addition in the second loop could be performed directly in memory, so the second **FOR** loop would probably be shorter in execution time. It was assumed that the two loops were comparable in execution time, thus, the speedup estimate is conservative. Simulations are under way to determine the speedups more exactly.

Consider the kinds of memory references required in the above algorithm. If the data is viewed as a matrix with P data points on a side, each processor operates on a row and then a column of that matrix. In a parallel system with global memory, the store is typically divided into several memory units. Optimum efficiency comes about when each processor is accessing a different memory unit during a given memory cycle, since each memory unit can deliver only one word per memory cycle. The most obvious way to split up the data is to put each segment of the contour (row of the matrix) in a separate memory unit. During the first half of the conversion, each processor acts

on a row, so the memory system operates with ideal efficiency. During the second half of the conversion, every processor acts on the same row simultaneously. This creates a large bottleneck at the memory unit containing that row. Kuck discusses this problem in [Kuc77] and suggests skewed storage techniques to eliminate these bottlenecks, at the cost of more complex address computations in array accesses. There is an overhead involved in every array access, thus reducing the speedup.

Another common model of parallel systems involves processors that can access only local memories. It is assumed that these accesses can occur without contention from other processors. In this model, all communications between processors take place through a communications network. In a system with local memory, the cost of parallel computation can be computed by considering the additional code and the number of interprocessor communications steps.

The algorithm below has been rewritten and restructured to use local memories only and to minimize parallel overhead by minimizing the number of communications steps. Recursive doubling is a method of computing accumulated sums across processors [Sto80]. An example showing the use of recursive doubling to obtain a different accumulated sum in each processor is shown in Figure 6.12. Here the doubling is done to produce correction values for all the segments of the contour at once. To show this in a program segment, assume a call to rec_dbl(val) uses the value val, takes care of all the communications to perform the recursive doubling, and then returns the partial sum. That is, if val_i is the value in processor i, $1 \leq i < p$, and rec_dbl is called in processor p, it will return the sum of val_0 through val_p.

$$\text{rec\_dbl}(val_p) = \sum_{i=0}^{p} val_i$$

The call to rec_dbl function assumes that the execution of the function will be synchronized with the other processors and that synchronization and transfers will be performed in parallel. The restriction that $P \equiv C^2$ is relaxed. The only assumption made

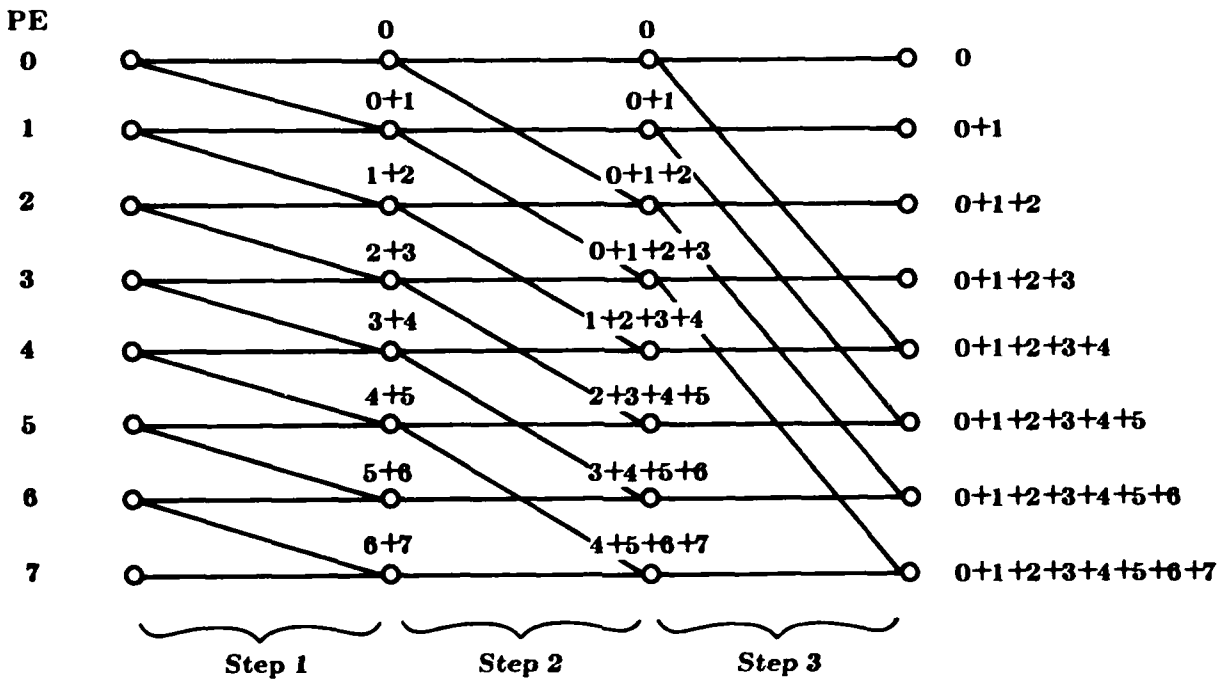


Figure 6.12. Recursive Doubling Example for 8 PEs, Requiring $\log_2 8 = 3$ steps

is that there are D input points in each processor's local memory in the array CCIn(0..D-1). Using these constructs and the constructs of Flock Algol, the algorithm is given as follows:

```
/* Local variable definitions */
P               /* Number of processing elements */
D               /* Number of data points in each processor */
CCIn(0..D-1)    /* Input chain code for one contour segment */
X(0..D)         /* X coordinates for this contour segment */
Y(0..D)         /* Y coordinates for this contour segment */
sumx            /* Partial sum of all X coordinates */
sumy            /* Partial sum of all Y coordinates */
p               /* The processor number */


X(0) = Y(0) = 0;
sumx = sumy = 0;

/* Compute X-Y coordinates for all points */
FOR i←1 THROUGH D-1 DO
BEGIN
        X(i+1)←X(i)+CtoX(CCIn(i))
        Y(i+1)←Y(i)+CtoY(CCIn(i))
END

/* Compute correction factors in parallel */
sumx←rec_dbl(X(D)) /* log2P transfer steps and 2log2P synchronization steps */
sumx←sumx-X(D) /* Only consider offset from previous segments */
sumy←rec_dbl(Y(D)) /* log2P transfer steps */
sumy←sumy-Y(D) /* Only consider offset from previous segments */

/* Correct each segment locally */
FOR i←1 THROUGH D-1 DO
BEGIN
        X(i)←X(i)+sumx
        Y(i)←Y(i)+sumy
END
```

Here the number of input points is C=P*D. The overall computational complexity is proportional to $2D+2\log_2 P$. If the assumption is kept that $C=P^2$, then D=P Since the correction stage is simpler than the original conversion, and the recursive doubling is only of the order $2\log_2 P$, speedups of at least $\frac{P}{2}$ are reasonable to expect. This modified algorithm also has the distinct advantage that is is independent of the

number of points in each processor, thus it is more versatile for handling a variable number of input points.

2. Filtering

The filtering of the image is an optional step to remove some of the quantization noise. This could be done easily in parallel by giving each processor a section of the contour. Given a filtering window width W, each processor will need to access W/2 points from each adjoining section. This could be accomplished by at most an additional W transfer steps. If a memory system is used where accesses to adjacent memories are allowed, it is important that "wrap-around" can occur.

Overall, in this portion of the algorithm speedups on the order of P can be expected. Significant deviations from this could come about depending on the window size W (a predetermined value). For a large W, the number of accesses to data in adjacent processors may be significant. Then, properties of the parallel system would have more effect on the speed of processing. These properties include methods of memory accesses and interconnection between processors/memory.

3. Resampling

The input outline needs to be resampled since the Fourier descriptor algorithm requires equal distances between input samples. From chain code input, the diagonal segments are longer by a factor of $\sqrt{2}$. If the points are divided into P continuous sections after resampling, there will be approximately the same number of points in each group. The computation of the total length of the contour can be sped up considerably. The length within each group can be computed with a speedup of P and the partial and total sums across the groups can be computed in $\log_2 P$ steps using a recursive doubling technique [Sto80]. If the filtering step is not employed, the lengths can be

computed more directly from the original chain code inputs. Now that the total length, the partial length up to the local segment, and the length within the local segment are known, a new "starting point" for each segment can be computed. A starting point is a point known to be in the *resampled* image. As soon as each processor has computed a starting point within its segment, it can individually compute the resampled points in its own section. The resampling operation consists of a convolution of the original points on the contour with a window function known as the interpolation kernel [Pra78]. The resampling is completed when every processor reports that its section is resampled. Although SIMD algorithms to perform resampling have been developed [WaS82], this algorithm is well suited to asynchronous operation since each processor can start once it knows its starting point, and each processor may have a different number of resampled points to compute.

Except during the recursive doubling, each processor operates primarily on local data. The only non-local data needed is from the adjacent segment (for resampling the end point) and typically consists of a single point. Hence, the memory conflict problem does not exist, and the algorithm is well suited to any parallel system. During the recursive doubling activities, the interconnection network may be used. Thus, any architecture in which the communications facilities can easily support these kinds of transfers should run this algorithm well. Alternatively, if these transfers need be performed in some other longer way, the degradation in performance should be small, since they account for only a small fraction of the computation time.

A serial resampling algorithm is very irregular. It is desired to follow the contour and mark points at equal spacing to be in the resampled contour. Except for the task of computing a starting point in the resampled contour for each segment, the operations performed are identical to the serial algorithm. This is well suited to a MIMD machine, the operations performed in any processor are directly dependent on the shape of the contour segment assigned to that processor. The number of resampled points

within any segment may differ by as much as $\sqrt{2}$ since the original allocation divided the contour by the number of original sample points, not by the length of each segment. The overhead is again minimal. Communications facilities are the major requirement of the MIMD architecture, yet these do not account for a significant amount of the processing time.

4. Fourier Transform Calculations

Once the contour has been resampled, the one-dimensional Fourier transform of the (new) contour is computed. It may be easier to compute the Fourier transform directly instead of using an FFT since for applications such as shape recognition, only the H most significant frequency coefficients are used, where typically H = 32 [WaM79]. If the entire input contour could be accessed by all processors with equal ease, then up to 32 processors could each compute the DFT complex coefficient for each of the 32 most significant coefficients. With no global memory this would take C' broadcast steps to distribute all the data points where C' is the number of resampled data points. Thus, a quick broadcasting ability would improve execution speed. Then, the Fourier transform could be computed locally as defined:

$$A(k) = \sum_{m=0}^{C'-1} C'(m) e^{-j(\frac{2\pi}{C'})mk}$$

where k has a different value in each processor. This clearly can be accomplished in $C'(\frac{H}{P})$ steps. It is not unreasonable that $\frac{H}{P}$ may be 1 (P = 32) in which case the algorithm can achieve a speedup of P. In general, an FFT cannot accomplish similar gains because it computes C' coefficients, all but H of which are ignored. C' may be on the order of 256 to 2098 while H = 32.

Using the approach cited, the major limiting factor may be the time necessary to broadcast all input data to all processors. From then on, speedup on the order of P

can be expected. Simulations are being performed to determine the ranges where the DFT or FFT is faster.

5. FD Normalization Procedure

For the moment, assume H = P = 32. A(0) is set to 0 and all values are scaled by $|A(1)|$. This requires one broadcast and 1 parallel division. To find which coefficient is largest, the magnitude can be computed in parallel, then the comparison will take $\log_2 P$ (=5) transfers and comparisons. The speedup would be on the order of only $\frac{P}{\log_2 P} = 6.2$ for this small section, using recursive doubling. Then depending on parameters of A(0) and A(k), the starting point and origin are shifted appropriately. This is done once if k = 2, otherwise it is done $|k-1|$ times. Speedups can be estimated from the operations involved in shifting the origin or starting points. Either of these can be computed easily by multiplying each coefficient by a complex factor. This factor is the same across processors for the origin adjustment and it is computed individually for the starting point adjustment. No communication or synchronization is needed, so any MIMD system should handle these well. The speedups then will be $S \simeq P = 32$ for these shifting operations.

When more than one of these normalizations are done, the "correct" normalization is computed as the one with the maximum sum $\sum_{i=1}^{H} |\mathrm{Re}[A(i)]|\,\mathrm{Re}[a(i)]$. These terms can be computed in parallel with optimal speedup and then the sum can be formed in $\log_2 H$ steps and compared on a single processor to each of the other normalizations.

Instead of dealing with a few hundred or a few thousand data points, where the number of data points varies depending on the input contour, this procedure deals with only H (H≃32) data points. Thus, care must be observed in estimating speedups since synchronization overhead may be a significant factor in execution speed. For this part

of the processing, an SIMD system may be better suited to the operations being performed. In general, the ability to perform limited synchronous tasks in an asynchronous system is an important factor to consider in this step. Overall, a somewhat lesser speedup in this section may not significantly affect the execution time since it is dominated by other sections.

6.2.4 Summary

A parallel implementation of the calculation of normalized Fourier descriptors has been presented. Issues addressed in the design include selection of SIMD versus MIMD processing for the component algorithms of the task, effects of global versus local memories, number of processors used, and interconnection requirements of the various portions of the task. These aspects of processing are included in the algorithm/architecture feature sets being developed.

6.3 Computer Vision

6.3.1 Introduction

With the advent of practical parallel architectures and the use of robotics becoming more prevalent in industry, it becomes of interest to see how these fields can be tied together to achieve higher levels of technological automation. One way in which this can be achieved is through the use of computer vision. By applying parallel computing to the computationally intensive task of computer vision, one might be able to achieve computational speedups large enough to benefit production systems. Other applications include applying the computer vision techniques to digital photogrammetry [Kea76], or applying parallelism to robotics. Although some hardware does exist to perform vision tasks in real-time, the hardware approach does not allow easy expansion or modification of the parameters measured. Moreover, existing real-time hardware is restricted to operations on binary images (number of gray levels equals two: black and white). Significantly more computation is needed for images having more (e.g., 256) gray levels.

In this work, the specific focus will be upon deriving and analyzing a flexible parallel computer vision system. The basic parameters that it is desired the system derive are based upon the SRI vision module [SRI79]. Additional parameters are based upon the usage of Fourier descriptors [WaM80]. This work provides "bottom up" information towards the identification of features which characterize the match between processes and architectures. By examining the attributes of the vision task and the architectures which appear well suited to it, the salient attributes of both the task and the architecture can be observed.

6.3.2 Definitions for Parallel Simulation

The machine model assumed is a variation of the PASM multimicroprocessor system [SiS81a]. This scheme assumes a number of Processing Elements, or PEs under the management of a Control Unit. The number of PEs is a power of two. Each of the PEs has a unique "address" between 0 and N-1 where N is the number of PEs. In addition, there exists some type of mechanism to allow all of the PEs to transfer data to other PEs simultaneously. For the computer vision task, the only transfer patterns that are being assumed are uniform modulo shifts (the highest PE connects to the lowest PE) and power of two distance transfers. An example of the former case is a uniform modulo shift of three, where PE_0 transfers to PE_3 and PE_i transfers to $PE_{(i+3) \bmod N}$. Note that the transfers wrap around from the high numbered processors to the low numbered processors. In general, the modulo shift can be of any positive or negative integer increment. An example of a power of two distance transfer is a transfer of distance 2 for 8 PEs. For this case, the following pairs exchange data: (0,2), (1,3), (4,6), (5,7).

In the model, each PE will contain the same code to run but will execute the code on a different subimage. (This is not a necessity for PASM.) However, within each PE, the code can run in a Multiple Instruction stream - Multiple Data stream (MIMD) form. This allows different PEs to execute different parts of a conditional statement concurrently, whereas in a strict Single Instruction stream - Multiple Data stream (SIMD) machine, only one of the parts of a conditional could be executed at a time. Note that this is not full MIMD performance, as it is required that the code in each PE be the same. This aids in insuring synchronization and thus helps enforce data coherence, i.e., insuring that a PE grabs the correct version of a variable from another PE. Thus, it does not matter if the separate processors take different times to execute their code, as they will be forced to synchronize at transfers to insure coherence.

The validity of the assumed model comes about from two directions. First, the idea of splitting the image among several PEs is valid since each subimage thus formed is still a valid image and the same types of operations are still needed on the pixels of each subimage. Second, since the actual quantities of the various operations that will be performed on each subimage may vary, asynchronous operation may allow higher PE utilization than strictly synchronous operation.

Synchronization can take place in one of two ways. First, synchronization is required at all data transfer points. This is done because data transfers often involve the same variable for all of the PEs. Explicit synchronization will also be possible by one of the simulation language constructs that requires that all PEs finish a section of code before any can move to the next section.

In order to develop parallel software, one must choose from one of two major approaches. Either the software can be of a generally descriptive nature to illustrate the parallelism (or lack thereof) inherent in a task, or the software can be designed to be compilable and testable, either by parallel execution or via serial simulation. Due the computational intensity and intricacy of the task at hand, the most reliable way to insure correctness is via testing. This will insure that typical problem cases are being handled by correctly by testing the software for a variety of images. A set of test images, some with multiple objects, was used for debugging and for analyzing computational speedup. Therefore, the software was designed so that it could be compiled and tested.

The actual programming was done in a modified version of 'C' [KeR78]. This language was chosen for its simplicity of developing parallel data structures and the high degree to which one can manipulate system information (such as memory areas). The latter played a large part in the simulating of parallel data transfers. The actual conversion of the serial 'C' language to a parallel language was done via macros and support subroutines. These features were designed to facilitate the development of

parallel code without the user having to know the specific details of the serial implementation. Thus, one can simply use the macro file without knowing its details and can then write parallel code.

The major points of this implementation are as follows. A construct of the form

$$\text{in\_pe}\left\{\ \text{codeblock;}\ \right\}$$

executes the enclosed block of code in each of the PEs. The prefix "PE." prepended to a variable indicates that the variable is local to a PE. All other variables are assumed to be global (e.g., the control unit has one copy of the variable). Global variables are used for such operations as loop control and overall conditional testing. There are also versions of the "in_pe" construct that allow the code to be executed in a limited subset of the PEs (mask() and recur_in_pe{}). These schemes use an address mask; an address mask is a matching format that the PE address must match for execution to occur in that PE.

Interprocessor communication is accomplished via a "transfer" subroutine,

transfer(destination_address,source_address,offset)

where the addresses are the variable addresses in the current PE. The transfer routine uses these addresses along with information about the size and structure of the PE data space to simulate the transfer via a memory-to-memory move. Recursive transfers and broadcasts (where one value is transferred to the all of the PEs) are similar. Synchronization is needed at transfer points to insure data coherence (otherwise, one PE might grab an incorrect version of a variable).

Adaptations of this basic scheme are possible. First, the limited MIMD ability could be converted to strict SIMD operation. The code would be the same, but the execution time would increase. This occurs because typically there would be idle PEs when conditionals are being executed since only one of the options of a conditional could be

executed at a time. As a second adaptation, the requirement for identical code in each PE could be removed. This could be useful for PEs that have to do special tasks (e.g., PEs processing an external border of an image). However, synchronization becomes a major concern at this point.

6.3.3 Overview

In this section, an overview of the procedures followed by the parallel computer vision software is provided. More detailed descriptions will be presented in the next section.

To facilitate testing of the system, it was desired that there be a simple way a user could enter an image into the system. The method chosen was to develop a simple scheme whereby the user could use a terminal with cursor control to draw an image on the screen and enter that image into the data memory. This section of the code used a small subsection of the "curses" [Arn] utilities available on the test system.

After an image has been entered into the data memory, the first task is to classify the image. This consists of transforming an image consisting of edge and non-edge pixels into an image with edge, internal, and external pixels. An internal pixel is a pixel that represents a point on an object, whereas an external pixel represents a point external to an object (such as the external background or a hole in the object).

After the inside and the outside of the image have been identified by the classification step, the holes in the image need to be located. A hole is defined as an area outside the object. Thus, the background also fits the definition of a hole. These holes are identified so that later merging can be easily accomplished. This capability is needed since holes that are initially thought to be separate may actually be joined. This separation can occur in one of two ways. First, a hole within a PE subimage might be of such a shape that it is initially thought to be multiple holes. This is due to the scanning pattern. The second way in which one hole might be thought to be

multiple holes is if the holes crosses a PE subimage boundary. In this case, each PE would think its section of the hole was separate.

Along with locating the holes, the areas of these holes are computed and recorded. This is done at the same time as the original hole identification since the data search patterns are quite similar. For purposes of isolating the object parameters, the background is defined to have an area of zero.

Once the inside of the object is known, it is a simple matter to determine the center of mass of the object. This is done by computing the local moments in each PE and then combining the results. Although, in and of itself, the center of mass is not a particularly useful parameter, it is used to normalize some of the perimeter statistics to be derived later.

To find the perimeter, all that needs to be done is find edge points that are adjacent to the background. Once this has be done, it is a simple matter to find the distances from the perimeter points to the center of mass. These distances are used to calculate the average, minimum, and maximum perimeter distance from the center of mass.

Finally, using the already determined perimeter, a description of this perimeter is produced in the form of a list of coordinate pairs. This list can then be used to determine Fourier descriptors or other similar parameters. Provisions have been made for the processing of images that contain multiple (non-overlapping) objects.

6.3.4 Detailed Descriptions of the Parallel Software

Image Initialization

To be able to test the system easily, a simple method by which a user could enter an image into the system was developed. The user executes the vision program and then uses a standard keyboard to direct the cursor and draw an image border. The user also has the option of turning the cursor on and off to allow him/her to draw

unconnected borders (such as an internal border). The connection pattern for the drawing is an eight neighbor scheme. That is, from a point, the user can direct the cursor in any of the four horizontal and vertical directions as well as along the diagonals between these directions. After the user has created the image to his/her satisfaction, an exit command automatically starts the image processing on the given image.

The produced image can be easily saved for later testing and can be reloaded and modified if desired in place of drawing a new image. The user also has the option of either saving the results in a text file or of simply viewing the results as they are produced.

The image is divided among the PEs with each of the PEs having an equally dimensioned stripe (either horizontal or vertical) of the image. Each PE then operates upon the section of the image contained in its local memory, communicating with other PEs when further information is needed. Thus, the granularity (unit of operation) in each PE is on the subroutine level.

Internal / External Classification

The classification scheme implemented is a two-pass method. A fixed number of passes has the advantage of having deterministic timings. The software could easily be modified to allow for indeterminate numbers of passes. As the results show later, this section of the software demonstrates good speedup. Thus, the assumption of a two-pass classifier gives a conservative speedup estimation. This is because if more passes were used, each pass would exhibit the same good speedup. As a result of the classification in this section, each pixel is labeled as being on the inside of the object, outside the object, or on the border. The first pass traverses the image from the upper left to the lower right. The initial classification of a pixel is based upon the two neighboring points (to the left of the current point and above the current point) that have already been classified. The method tries to classify the new point as external if either of the

previous points is external. If the adjacent points are both edges (border pixels), then information on the length of the edge and the previous region classifications are used to make the classification.

The second pass traverses the image from the lower right to the upper left (backwards as compared to the forward pass). This pass uses the four major compass points in relation to the current point to attempt to correct any classification errors. Again, the bias is toward external classification.

This section of the vision software also uses several schemes to insure robustness. Besides the previously mentioned ability to reclassify points on the second pass, the software also looks for the specific case of tracing an edge. In addition, several trouble patterns are checked for to prevent major misclassifications. Figure 6.13 illustrates the classification procedure. The first part of the figure is the image before classification (border only). The edges are represented by '2.' The other two parts are the image after the first and second passes of the classification. Internal points are represented by '1' and external points are represented by '0.' An example of a reclassification on the second pass is illustrated by the outlined areas in Figures 6.13b and 6.13c.

In the parallel implementation, each PE works with its own stripe of the image data. The communication between PEs is limited to the values of the border elements of a subimage. One such transfer will take place for each border element on one of the sides of the subimage. These transfers will be uniform modulo shifts of distance one.

Identifying Image Holes

After the object has been separated from its surroundings by the classification section, the holes in the image need to be identified. This operation is initially performed separately within each PE. This is done by creating a template of the same size as the initial local section of the image. These templates are arrays that are of the same size as the subimage in the PE. Each template location will contain an identifier that

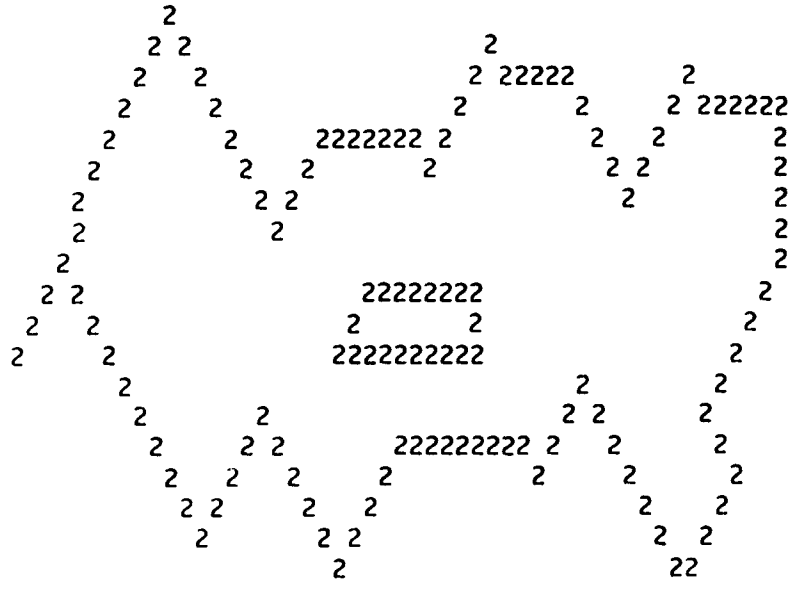


Figure 6.13a Initial Image

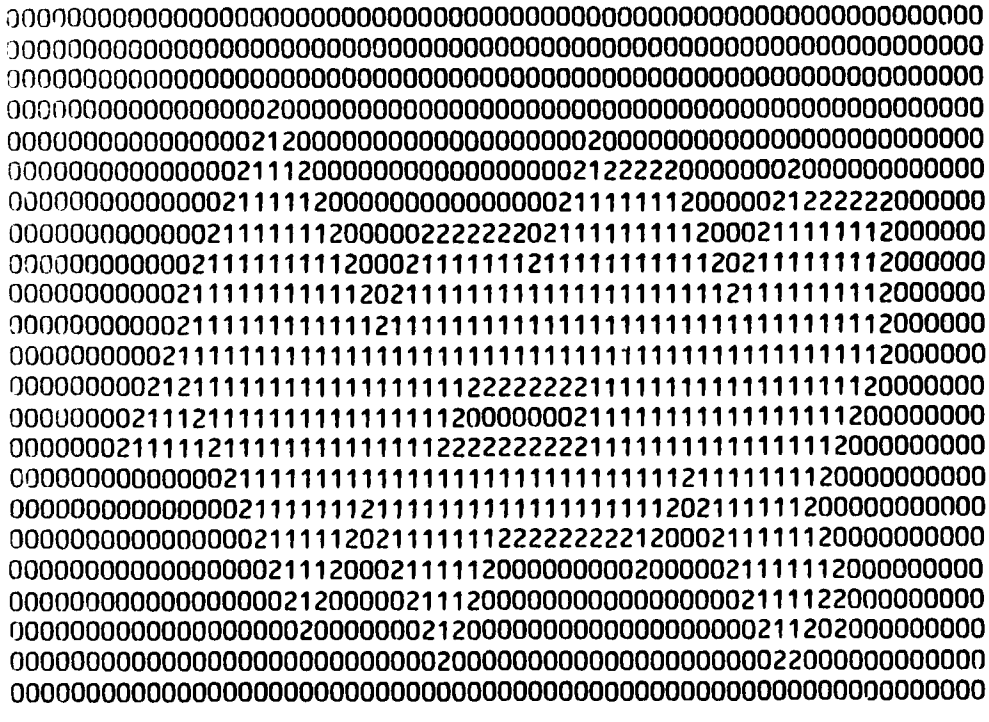


Figure 6.13b Classification: First Pass

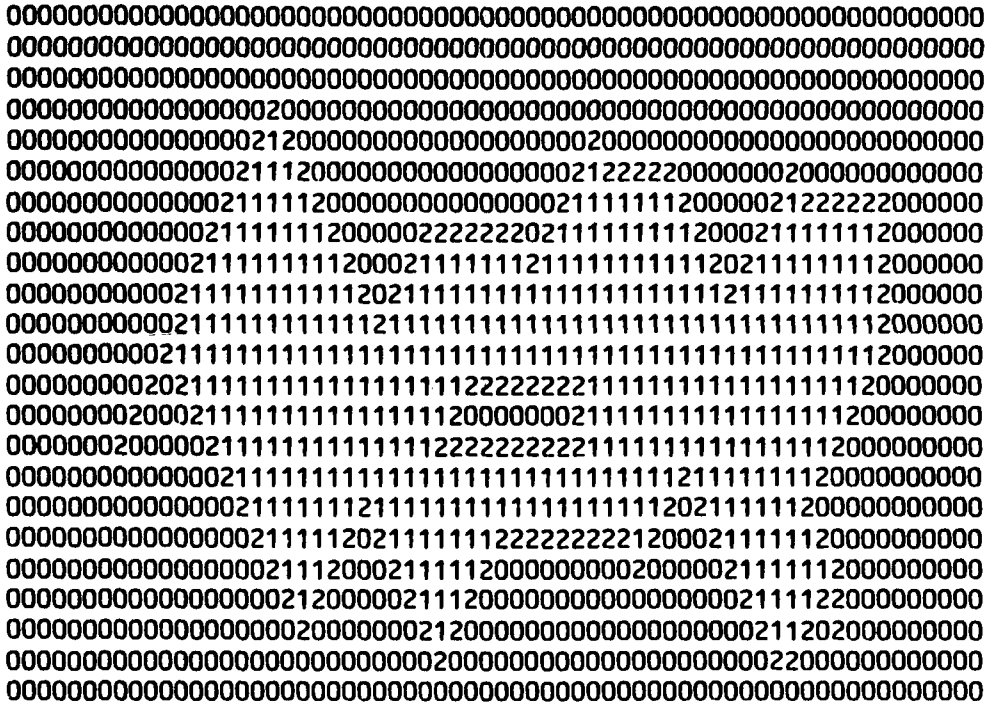


Figure 6.13c Classification: Second Pass

indicates the local hole number for the corresponding subimage point (zero for non-hole points). Each time an external point is located that is not adjacent to a previous hole, a new hole identifier is used and entered for that point in the template. If the external point is adjacent to a previous hole, then the previous identifier is continued. A two neighbor scheme is used for all of the pixels except those on one of the subimage borders. Since the points on one edge will only have points from the previous row (or column, in the case of horizontal stripes) to base a decision upon, a one neighbor scheme is used. Experimentation showed that no accuracy problems are encountered due to the small number of neighbors being used in the classification. There is also the possible case of an external point being adjacent to two different previous hole identifiers. This is effectively a merging of holes. The software accounts for this via a set of parameters that keep track of merged holes and their statistics without having to go back and relabel the hole template.

These operations are performed totally within a PE: no communication with other PEs is needed. Each PE owns the information about its own holes. This information is transferred to other PEs during hole merging (described later).

Figure 6.14 shows the internal hole identifiers for each PE (the sections of the image within each PE are separated for easier examination). Hole identifiers that are adjacent are considered common. That is, only one of the identifiers contains the information for the hole. All of the others contain a pointer to the "master" information.

Once the holes have been identified within each PE, they need to be merged across the PE borders. This is done by transferring the borders of the PE hole template to adjacent processors and looking for matching holes. The areas are merged at the same time that holes are joined. The scheme used is that if a hole has only one edge on a PE border, then the statistics for that hole are transferred to that adjacent PE. This finally results with each hole being "controlled" by one PE. For purposes of easy identification and to separate holes within an object from the background, the border background is

```
      PE 0              PE 1              PE 2              PE 3
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1111111111111111 1111111111111111 1111111111111111
1111111111111111 1 111111111111111 1111111111111111 1111111111111111
1111111111111111    1111111111111 111111 111111111 1111111111111111
11111111111111       111111111111 11111        1111 111 111111111111
1111111111111         11111111111 1111          111 11          111111
111111111111           11111        1            11 1           111111
11111111111             111                       1             111111
1111111111               1                                      111111
1111111111                                                      111111
111111111                                                       111111
11111111 2                                                     3333331
11111111 333                    33 22222                       44444441
1111111 44444                                                  555555551
11111111111111                                                 6666666661
111111111111111                                    4           77777777771
1111111111111111        4                         555           7777777771
1111111111111111 2     555          333333333 66666            777777771
1111111111111111 22   66666       7 3333333333333333          777777771
1111111111111111 222 8888888    97 3333333333333333 2     8 777777771
1111111111111111 222222222222 ::7 3333333333333333 22  999999999991
1111111111111111 2222222222222227 3333333333333333 2222222222222221

Hole determination: 2 total holes in image
Total Hole Areas: 7
```

Figure 6.14 Image Hole Determination

defined as having an area of zero. The process of merging is illustrated in Figure 6.15.

This method of merging holes across PEs is deterministic in that the maximum number of passes needed can be determined by the types of images being examined. For example, the more an object tends to spiral (a spring, for example, as compared to a wheel), the more passes that will be needed. For analyzing performance, a fixed number of passes (more that necessary for the images being considered) was assumed. It was found that this section provides poor speedup. Thus, the net result of the fixed number of passes is again to provide a conservative estimate of the computational speedup of the algorithm. The information that needs to be transferred from each PE is placed on a transfer stack. These stacks are then transferred. All of these are transfers to logically neigboring PEs. The amount of information transferred is highly dependent upon the actual image.

Computing Image Hole Areas

The areas to be computed can easily be tabulated at the same time as the hole identifiers are placed in the template. This reduces the amount of computation necessary since the area computation is divided among the PEs. To handle the merging of holes, either within a PE of between PEs, an indirection table to point to the actual hole area is used.

Locating the Center of Mass

After the points that comprise an object are known, the center of mass of the object can be easily determined. In this system this step is performed by computing the moments in each PE separately and then summing across PEs using recursive doubling [Sto80]. After the center of mass has been determined, it is broadcast to all PEs since this information will be needed at a local PE level in later processing. This scheme requires that each PE know its absolute position in the configuration since the

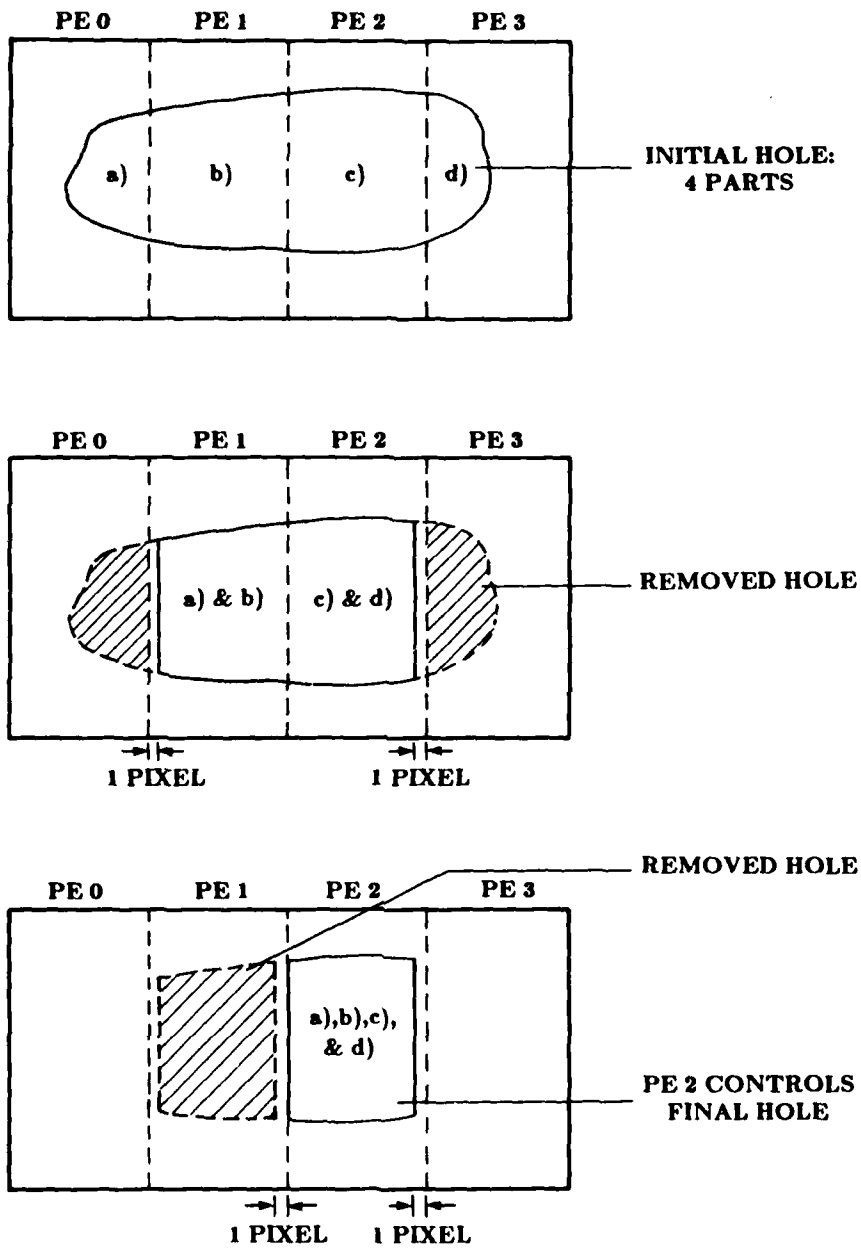


Figure 6.15 Hole Merging Example

weighting of one of the moments in each PE is dependent upon the PE address. For example, if the stripes are in the vertical direction, the the x axis will be split among the PEs. Moments that involve the absolute distance along the x axis would depend upon the PE address. To get the center of mass, $\log_2 N$ sets of transfers will be needed. This would be followed by one broadcast, for an effective total of $1 + \log_2 N$ transfers.

Perimeter Identification and Perimeter Statistics Determination

Identifying the perimeter is straightforward once the external background hole has been identified. This hole has area zero by definition. An edge point next to an external hole (or next to another perimeter point) is a perimeter point. Since the area of holes is determined through an indirection table, all one needs to do is see if the hole has zero area. When a perimeter point is located in a PE, a counter in that PE is also incremented so that the total perimeter can be determined by a simple application of recursive doubling to accumulate the total across the PEs.

After the perimeter has been identified, it is a simple matter to find the distances between the perimeter points and the previously determined center of mass. This is done by scanning through the image template looking for perimeter points. Each PE scans its stripe of the image. For each perimeter point so found, the radial distance from the perimeter point to the center of mass is determined. A running sum is kept of these distances, along with the minimum and the maximum distances. When the entire image has been scanned, recursive doubling is used to find the average, minimum, and maximum such distances. Three stages of recursive doubling transfers will be needed, one set for each of the perimeter statistics being gathered. This results in a total of $3\log_2 N$ transfers.

Figure 6.16 shows the identified perimeter for an image. The perimeter is noted by "B," as compared to "2" for a non-perimeter edge point.

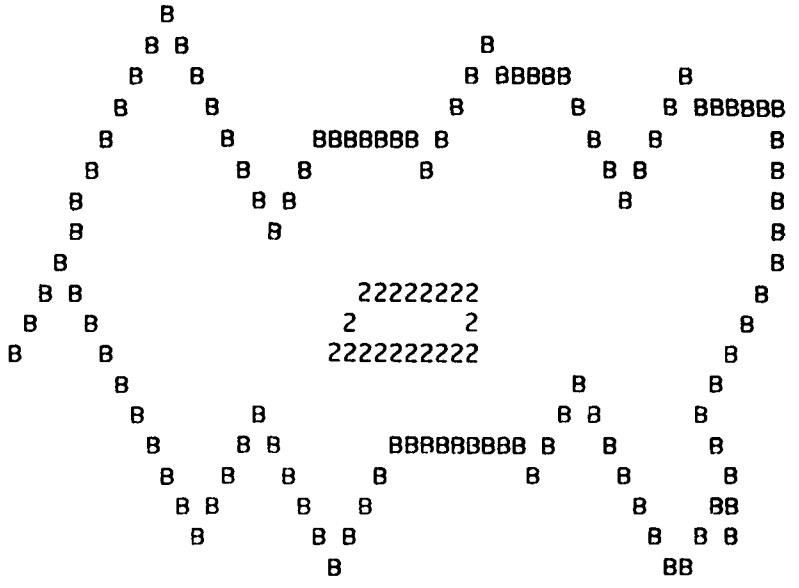
Data Preparation for Fourier Descriptors

As an illustration of some of the higher level functions that can be performed once the basic parameters have been extracted, the image can be converted into the information necessary to calculate Fourier descriptors (refer to Chapter 6.2 of this report). This information is simply an ordered list representation of the perimeter of the object. Each entry in this list consists of a set of coordinates representing a perimeter point.

The vision software begins this step by forming the perimeter nodes into a multiply-linked list. This is done to facilitate the removal of false perimeter points (spikes). This converts the perimeter into a traceable contour. Next, these linked-lists are transferred to one PE which completes the processing. This processing includes converting the lists into partial ordered lists and then combining these lists. Other schemes, such as forming the partial lists in each PE separately, were found to induce such a large amount of overhead in transfers that any advantages in parallelism were lost. The final contours in the single PE are then broadcast to the remainder of the PEs in preparation of the Fourier descriptor calculations. If the perimeter is equally distributed among the PEs, 75% of the partial ordered listings will need to be transferred. Each of the objects in one of these lists contains ten data fields. If the perimeter is not equally distributed, then the perimeter could be gathered into the PE with the largest number of perimeter points. Thus, if there are P perimeter points, a maximum of 7.5P transfers would be needed. An example of a contour listing for the image from Figure 6.16 is given in Table 6.3.1.

Multiple Object Images

The software that has been described up to this point has treated the contents of the image field as one object. If there is more than one object in the image field, the same software can still be used, but the results will be a composite of the information for the separate objects. However, it is not exceedingly difficult to separate the



Total Object Perimeter: 109
Center of Mass: (33,11)
Perimeter Statistics: distances from Center of Mass
MIN = 3, MAX = 24; AVG = 12

Figure 6.16 Object Perimeter Determination and Center of Mass Statistics

Table 6.3.1:
Final Contour Representation.

(Read across, then down)

(7,31)	(7,30)	(7,29)	(7,28)
(7,27)	(8,26)	(9,25)	(10,24)
(9,23)	(8,22)	(7,21)	(6,20)
(5,19)	(4,18)	(3,17)	(4,16)
(5,15)	(6,14)	(7,13)	(8,12)
(9,11)	(10,11)	(11,10)	(12,11)
(13,12)	(14,13)	(15,14)	(16,15)
(17,16)	(18,17)	(19,18)	(20,19)
(19,20)	(18,21)	(17,22)	(16,23)
(17,24)	(18,25)	(19,26)	(20,27)
(21,28)	(20,29)	(19,30)	(18,31)
(17,32)	(17,33)	(17,34)	(17,35)
(17,36)	(17,37)	(17,38)	(17,39)
(17,40)	(18,41)	(17,42)	(16,43)
(15,44)	(16,45)	(17,46)	(18,47)
(19,48)	(20,49)	(21,50)	(21,51)
(20,52)	(19,53)	(19,54)	(18,54)
(17,53)	(16,52)	(15,53)	(14,54)
(13,55)	(12,56)	(11,57)	(10,57)
(9,57)	(8,57)	(7,57)	(6,57)
(6,56)	(6,55)	(6,54)	(6,53)
(6,52)	(5,51)	(6,50)	(7,49)
(8,48)	(9,47)	(8,46)	(7,45)
(6,44)	(5,43)	(5,42)	(5,41)
(5,40)	(5,39)	(4,38)	(5,37)
(6,36)	(7,35)	(8,34)	(7,33)

information for the separate objects.

Once the contours of the image have been determined, the software knows how many separate objects are in the image. This involves the classification, hole and area identification and merging, and perimeter determination steps described above. That is, the number of contours will equal the number of objects in the image given that the objects do not overlap and that no object is inside of another (such as a bolt within a wheel rim). To process the items individually, all that needs to be done is remove the objects corresponding to the undesired contours and reprocess the image. This should be done for each object in the image. The individual processing involves all the the previous sections, from classification through perimeter determination and perimeter statistics.

To remove an object from the image, its perimeter points (which are known from the contour) are marked to be removed. Two passes are made over the image (similar to the initial classification) to convert internal, perimeter, and edge points bordering the removal points to removal points themselves. This is similar to the erosion scheme used by CLIP4 [ReO82]. A final pass is made of the image to convert all removal points to external points, effectively erasing the object from the image.

If the program detects multiple images, it will still give the composite results, but it will also sequentially erase all but one of the objects and then process the remaining object. This additional processing is identical to the main processing sequence, except the checks for multiple objects are omitted.

Parallel Utility Software

Along with the macros used to help define a parallel version of 'C,' several subroutines were needed to simulate such operations as uniform shift transfers, recursive doubling transfers, and address mask computation. The most major of these operations are the transfer routines. These routines use the address of a source and destination within

a current PE to obtain the proper address in the destination PE. This new address is used for the actual transfer of information. Thus, inter-PE transfers are simulated by memory operations. Recall that synchronization is needed at transfer points.

Parallel Computer Vision Software

The actual programs used for the parallel computer vision software were compiled and run on an 11/780 Dual Vax at Purdue University. Figure 6.17 contains an example of the vision software output.

6.3.5 Analysis

In order to determine the validity of applying parallel architectures to computer vision, a comparison of the parallel software with serial software is needed. This simulation was performed on an 11/780 Dual Vax [GoM82]. An estimation of the computational speedups were derived by an examination of the structure of the parallel software as described above (see Table 6.3.2). It is assumed that the parallel architecture is capable of performing some limited type of MIMD operation. However, the software is such that in strict SIMD operation is needed, then the speed will be reduced by a factor of close to a small power of two (due to the nested conditional statements). For example, for a two level conditional, which effectively has four states, then the SIMD machine will need to execute each of these conditions separately. If the code executed in the conditional sections is of approximately the same complexity, then the SIMD machine would need four times as long to execute the code for all conditionals than the MIMD machine would.

```
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000002000000000000000000000000000000000000000000000000000
00000000000000021200000000000000000020000000000000000000000000000000
00000000000000021112000000000000000021222220000000200000000000000000
00000000000000021111200000000000000021111111200000212222220000000000
00000000000002111111200000022222202111111111200021111111200000000000
00000000000021111111112000211111121111111111120211111111200000000000
00000000002111111111112021111111111111111111121111111112000000000000
00000000021111111111111211111111111111111111111111111112000000000000
00000000211111111111111111111111111111111111111111111112000000000000
00000000202111111111111111112222222211111111111111111120000000000000
00000002000211111111111111120000000211111111111111111200000000000000
00000020000021111111111111122222222211111111111111112000000000000000
00000000000000211111111111111111111111111111121111111120000000000000
00000000000000021111111211111111111111111112021111112000000000000000
00000000000000002111120211111112222222221200021111112000000000000000
00000000000000000211120002111112000000000200000211111200000000000000
00000000000000000021200000211120000000000000000002111122000000000000
00000000000000000002000000021200000000000000000002112020000000000000
00000000000000000000600000000200000000000000000000220000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
```

Hole determination: 2 total holes in image

Total Object Perimeter: 109

Total Hole Areas: 7

Center of Mass: (33,11)

Perimeter Statistics: distances from Center of Mass
MIN = 3, MAX = 24; AVG = 12

1 Object(s) in Image.

Figure 6.17 Example of Vision Software Output

Table 6.3.2
Computational Performance Results

Algorithm Division	Approx. Speedup	Serial Time	Time Proportions
class()	N(I/(I+N-1))	15.36	0.3531
holes()	N/((N-1)(SPIFAC+1))	15.79	0.3630
areas()	(called by holes)	N/A	N/A
center()	N	1.64	0.0377
pstats()	N	10.71	0.2462

I = Image Border (I by I image)
N = Number of PEs
SPIFAC = How many times a section of the object in the image can switch directions in crossing the image (for example, the letter "Z" would have a SPIFAC of 2). For the images analyzed, SPIFAC = 6

To provide coherence with the "curses" input method, images were 64 by 23. The image was divided into 64/N by 23 stripes. With this division method, the speedups are still primarily determined by N and I.

Total Weighted Speedup:

$$S(N,I) = \frac{0.3531NI}{I+N-1} + \frac{0.3630N}{(N-1)(SPIFAC+1)} + 0.0377N + 0.2462N$$

$$= N\left(\frac{0.3531I}{I+N-1} + \frac{0.3630}{(N-1)(SPIFAC+1)} + 0.2839\right)$$

$$\lim_{N\to\infty} S(N,I) = 0.3531I + \frac{0.3630}{SPIFAC+1} + 0.2839N$$

Thus, a reasonable maximum speedup for design assumptions is:

$$S(N,I) < 0.35I$$

For I = 1024,

S(N,I) ˜ 71withN ˜ 128

S(N,I) ˜ 135withN ˜ 256

S(N,I) ˜ 247withN ˜ 512

S(N,I) ˜ 433withN ˜ 1024

Simulation demonstrated that the major problem with the parallel implementation is basically of one form: the amount of transfers needed reduce the effectiveness of the parallelism. This can occur when the amount of information that is needed to make a proper decision (such as for hole merging) is large. This problem can manifest itself in several forms, such as algorithms that are inherently serial or that require data from the entire image. Such tasks might better be performed in one PE or in the Control Unit.

The actual results for the major section of the software are presented in Table 6.3.3. The proportions of time required by different sections of the code were determined by executing a serial version of the algorithm. The problem of non-determinism in speedups was handled by using deterministic versions of non-deterministic routines. These routines were designed in such a way as to provide a conservative estimate of the speedup. With the current simulation, there is no way to account for major synchronization delays due to greatly different execution times. Thus, these results should only be used an an order of magnitude indicator.

One must remember that these analyses assume that the serial simulations of parallel operations (such as inter-PE transfers) are approximately a factor of N slower (for simulating an N PE system). If a transfer is assumed to take approximately the same time as a memory access (no provision for operational overlap), then this assumption is quite valid.

The proportions of time that the serial algorithm sections take are used to provide a weighting of the parallel speedup results. This way, a section with low speedup that requires only a small fraction of the serial processing time will not falsely lower the overall speedup. Similarly, a section with high speedup that requires only a small fraction of the serial processing time will not falsely raise the overall speedup.

Table 6.3.3

Experimental Speedup Results

Algorithm	Serial Time	N=2 Time	N=4 Time	N=2 Speedup	N=4 Speedup
classification	15.36	9.47	6.02	1.62	2.55
holes and areas	15.79	13.47	17.11	1.17	0.92
center	1.64	1.11	0.66	1.48	2.48
perimeter	10.71	5.61	2.79	1.91	3.84
overall	43.50	29.64	26.86	1.47	1.62

Compare these with the analytical speedups of 1.24 for N=2 and 2.35 for N=4 (obtained from the speedup formula). Thus, there is no advantage in having N too large. Again, the assumption of no overlap between processing and transfers is conservative. Other software could be added to the program and analyzed in a similar fashion.

An example of this would be producing the contours. Simulations of methods tried so far have produced poor results due to the serial nature of the algorithm. However, it must be noted that the rest of the processing can be done in parallel without incurring losses in computational speed when the contours need to be found. That is, even though the parallel method is poor, the overall algorithm can take advantage of parallelism without having to increase processing time to regroup data to find the

contours. Again, the major overhead is transfers. If any type of full or partial global memory is available, the speedups should improve for this algorithm.

6.3.6 Possible Additional Features

The purpose of this simulation was to determine the feasibility of performing computer vision tasks on a specific class of parallel architecture. It can easily be expanded and used as a model for an actual vision system. In such a system, there are other features that one might wish to add in order to improve the robustness of object identification. Some of these additional parameters are simply combinations of previous parameters. An example of such a parameter is the factor of roundness (how circular the image is), which is computed by dividing 4π times the area by the square of the perimeter. The area of the object could also be calculated at the same time that the second classification pass is made. This area could be combined with the internal hole area to provide a total of the areas occupied by the object. The ratio of hole area to total area is similarly obtainable.

There are other parameters that would require additional computation in the main processing sequence to determine. This class of parameters would include such features as second moments, ratios of major and minor axes, finding the bounding rectangle, and line fitting. Others could be added based upon the specific task at hand.

Finally, one needs to consider the non-ideal cases where either multiple objects in the image overlap or the objects are not entirely contained within the borders. Much information for the latter case can be obtained from processing the object as usual and then applying statistical methods to determine possible matches with known objects. The other case is not as simple - some type of image reduction would be necessary if it was determined that an object was not known. Such software could selectively reduce protrusions of an object until a known object was found.

6.3.7 Architectural Considerations

A specific type of architecture has been assumed throughout this simulation and analysis. At this point, this restriction will be removed and the tasks considered will be examined to explore the advantages of other types of parallel architectures.

First, consider the attributes which the vision task requires. Each processing unit would need access to a substantial amount of memory to store a large enough subsection of the image. Also, arithmetic operations are necessary (not just simple logical operations).

One type of architecture that has been used for simpler image processing is a large scale bit-processing machine. Although such a machine might be adaptable to the variety of tasks at hand, the correspondence is not obvious.

If the approach of tracing a contour is taken (i.e., one PE traces exactly one contour), two major problems arise. First, how does one select the contour to trace? That is, is one contour split up among processors, and if so, how is this division performed? Also, what happens in the case of multiple contours? Second, such a scheme might have serious problems utilizing a reasonable number of processing units.

The final major approach to be examined here is region growing. However, this approach is not far different from the methods already used. It thus appears that the original is an acceptable initial choice. The concern now becomes to determine how this model could be improved. This redesign will be performed in an attempt to reduce the major cause of efficiency loss - interprocessor transfers.

By examining the algorithms, is it seen that a set memory area (the memory assigned to one PE) is not needed by more than two PEs in a given processing section. If the memory is dual ported, with one write channel and two read channels, then the need for transfers can be virtually eliminated. In such an approach, the memory that was previously the exclusive responsibility of a specific PE would still be connected to that PE via the write channel and one of the read channels. However, the other read

channel would be connected to a memory redirection network that would be setable by the Control Unit when a new type of access pattern is needed. This redirection network could either be bidirectional or (more practical) two unidirectional networks, one direction being used to transmit the memory accesses and the other being used to return the data. The advantage of using two unidirectional networks is that information can be flowing in both directions at the same time without the need for redirection or buffering. This would allow the memory to be accessed in an interleaved manner, further improving system performance. When this scheme is compared with the number of transfers needed in some of the processing steps (such as in hole merging and Fourier descriptor preparation), the possible savings are quite evident.

6.3.8 Conclusions / Recommendations

This software has demonstrated the advantages of applying parallel architectures to computer vision specifically and image processing in general. Because of the modular design of this software, it is quite possible to expand the processing sequence to include other common image processing techniques. Such actions will help provide a clearer picture of the advantages and disadvantages of parallel computing. In general, it has been shown that increases in performance (such as overall speedup) of near to I for an I by I image are obtainable. Performance of this order of magnitude is also attainable with a moderate number of PEs (N up to I).

REFERENCES

[AdS82a] G. B. Adams III and H. J. Siegel, "A multistage network with an additional stage for fault tolerance," *15th Annual Hawaii Int'l. Conf. System Sciences,* Vol. 1, Jan. 1982, pp. 333-342.

[AdS82b] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Computers,* Vol. C-31, May 1982, pp. 443-454.

[Ada82] J. Adamo, "Pascal + CSP, merging Pascal and CSP in a parallel processing oriented language," *Third Int'l. Conf. on Distributed Computing Systems,* Oct. 1982, pp. 542-547.

[AgF73] T. Agerwala and M. Flynn, "Comments on capabilities, limitations and 'correctness' of Petri nets" *Proc. 1st Symp. Computer Architecture,* July 1973, pp. 81-86.

[Agr82] D. P. Agrawal, "Testing and fault-tolerance of multistage interconnection networks," *Computer,* Vol. 15, Apr. 1982, pp. 41-53.

[AhH76] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley Publishing Co., Reading, Mass., 1976.

[AnJ75] G. A. Anderson, and E. D. Jensen, "Computer interconnection structures: taxonomy, characteristics, and examples," *ACM Computing Surveys,* Vol. 7, Dec. 1975, pp. 197-213.

[And77] S. Anderson, "The looping algorithm extended to base 2^t rearrangeable switching networks," *IEEE Trans. Communications,* Vol. COM-25, Oct. 1977, pp. 1057-1063.

[Arn00] K. Arnold, "Screen Updating and Cursor Movement Optimization, A Library Package," UNIX* Version 4.1bsd.

[BaB68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Trans. Computers,* Vol. C-17, Aug. 1968, pp. 746-757.

---

*UNIX is a trademark of Bell Telephone Laboratories, Inc.

[BaJ77] J.-L. Baer and J. Jensen, "Simulation of large parallel systems: modeling of tasks" *Proc. 3rd Int'l. Symp. on Modeling and Performance Evaluation,* Oct. 1977, pp. 53-73.

[BaL81] G. H. Barnes and S. F. Lundstrom, "Design and validation of a connection network for many-processor multiprocessor systems," *Computer,* Vol. 14, Dec. 1981, pp. 31-41.

[Bae82] J.-L. Baer, "Techniques to exploit parallelism" in *Parallel Processing Systems, An Advanced Course* D. J. Evans, ed., Cambridge University Press, Cambridge, England, 1982, pp. 75-99.

[Bak72] H. Baker, "Petri nets and languages," Computer Structures Group Memo 68, Project MAC, MIT, 1972.

[Bat68] K. E. Batcher, "Sorting networks and their applications," *AFIPS 1968 Spring Joint Computer Conf.,* April 1968, pp. 307-314.

[Bat74] K. E. Batcher, "STARAN parallel processor system hardware," *AFIPS 1974 Nat'l. Computer Conf.,* May 1974, pp. 405-410.

[Bat76] K. E. Batcher, "The flip network in STARAN," *1976 Int'l. Conf. Parallel Processing,* Aug. 1976, pp. 65-71.

[Bat79] K. E. Batcher, "MPP -- a massively parallel processor," *1979 Int'l. Conf. Parallel Processing,* Aug. 1979, p. 249.

[Bau74] L. H. Bauer, "Implementation of data manipulating functions on the STARAN associative array processor," *1974 Sagamore Computer Conf. Parallel Processing,* Aug. 1974, pp. 209-227.

[BeN71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples,* McGraw-Hill, New York, 1971.

[Ben64] V. E. Beneš, "Optimal rearrangeable multistage connecting networks," *Bell System Technical Journal,* Vol. 43, No. 4, Part 2, July 1964, pp. 1641-1656.

[Ben65] V. E. Beneš, *Mathematical Theory of Connecting Networks and Telephone Traffic,* Academic Press, New York, N.Y., 1965.

[Ber62] C. Berge, *The Theory of Graphs,* John Wiley and Sons, Inc., New York, N.Y., 1962, p. 12.

[Bla77] C. E. Blakely, "PEPE application to BMD systems," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 193-198.

[Bou72] W. J. Bouknight, et al., "The Illiac IV system," *Proc. IEEE,* Vol. 60, No. 4, April 1972, pp. 369-388.

[BrF82] F. A. Briggs, K. S. Fu, K. Hwang, and B. W. Wah, "PUMPS architecture for pattern analysis and image database management," *IEEE Trans. Computers,* Vol. C-31, Oct. 1982, pp. 969-983.

[Bur79] Burroughs Corporation, *Final Report - Numerical Aerodynamic Simulation Facility Feasibility Study,* prepared under contract NAS2-9897, Paoli, Pa., March 1979.

[CaK80] R. H. Campbell and R. B Kolstad, "An overview of Path Pascal's design and Path Pascal user manual," *SIGPLAN Notices,* Vol. 15, No. 9, Sept. 1980, pp. 13-24.

[ChY82] P.-Y. Chen, P.-C. Yew, and D. H. Lawrie, "Performance of packet switching in buffered single-stage shuffle-exchange networks," *3rd Int'l. Conf. Distributed Computing Systems,* Oct. 1982, pp. 622-627.

[CiS80] L. Ciminiera and A. Serra, "LSI implementation of modular interconnection networks for MIMD machines," *1980 Int'l. Conf. Parallel Processing,* Aug. 1980, pp. 161-162.

[CiS81] L. Ciminiera and A. Serra, "Modular interconnection networks with asynchronous control," *14th Annual Hawaii Int'l. Conf. System Sciences,* Vol. 1, Jan. 1981, pp. 210-218.

[CiS82] L. Ciminiera and A. Serra, "A fault-tolerant connecting network for multiprocessor systems," *1982 Int'l. Conf. Parallel Processing,* Aug. 1982, pp. 113-122.

[Clo53] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal,* Vol. 32, No. 2, March 1953, pp. 506-424.

[Csp00] CSP Inc., Burlington, Mass.

[Dav74] E. W. Davis, "STARAN parallel processor system software," *AFIPS 1974 Nat'l. Comp. Conf.,* May 1974, pp. 17-22.

[DeB80] J. B. Dennis, G. A. Boughton, and C. K. C. Leung, "Building blocks for data flow prototypes," *7th Annual Int'l. Symp. Computer Architecture,* May 1980, pp. 1-8.

[DeP78] A. M. Despain and D. A. Patterson, "X-tree: a tree structured multi-processor computer architecture," *5th Annual Int'l. Symp. Computer Architecture*, Apr. 1978, pp. 144-151.

[Den70a] P. J. Denning, "Virtual memory," *Computing Surveys*, Vol. 2, Sept. 1970, pp. 153-188.

[Den70b] J. Dennis (editor), *Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation*, ACM, New York, 1970.

[Den72] J. Dennis, "Concurrency in software systems," Computation Structures Groups Memo 65-1, Project MAC, MIT, 1972; Also *Advanced Course in Software Engineering*, Springer-Verlag, Berlin, 1973, pp. 111-127.

[DiJ81] D. M. Dias and J. R. Jump, "Analysis and simulation of buffered delta networks," *IEEE Trans. Computers*, Vol. C-30, Apr. 1981, pp. 273-282.

[Dij65] E. Dijkstra, "Solution of a problem in concurrent programming," *Communications of the ACM*, Vol. 8, 1965, pp. 569-570.

[Dij68] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages*, edited by F. Genuys, Academic Press, Inc., New York, 1968, pp. 43-112.

[DoD82] *Military Standard: Ada Programming Language*, United States Department of Defense, Dec 1980.

[DuH73] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.

[DuW74] M. J. B. Duff, D. M. Watson, and E. S. Dentsch, "A parallel computer for array processing," *Information Processing 74*, pp. 94-97.

[Esl00] E. S. L., Inc., Sunnyvale, Calif.

[FaP81] K. M. Falavarjani and D. K. Pradhan, "Fault-diagnosis of parallel processor interconnection networks," *11th Annual Int'l. Symp. Fault-Tolerant Computing*, June 1981, pp. 209-212.

[FeK82] T. Feng and I. Kao, "On fault-diagnosis of some multistage networks," *1982 Int'l. Conf. Parallel Processing*, Aug. 1982, pp. 99-101.

[FeL77] R. D. Fennell and V. R. Lesser, "Parallelism in artificial intelligence problem solving: a case study of Hearsay II," *IEEE Trans. Computers*, Vol. C-26, No. 2, Feb. 1977, pp. 98-111.

[Fen72] T.-Y. Feng, "Some characteristics of associative/parallel processing," *Proc. 1972 Sagamore Comp. Conf.*, Syracuse Univ., Aug. 1972, pp. 5-16.

[Fen74] T. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Computers*, Vol. C-23, Mar. 1974, pp. 309-318.

[Fen81] T. Feng, "A survey of interconnection networks," *Computer*, Vol. 14, Dec. 1981, pp. 12-27.

[Flo00] Floating Point Systems, Inc., Portland, Ore.

[Fly66] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, Vol. 54, Dec. 1966, pp. 1901-1909.

[Fly72] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Computers*, Vol. C-21, Sept. 1982, pp. 948-960.

[Fos76] C. C. Foster, *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, New York, 1976.

[GaW81] N. C. Gallagher, Jr. and G. L. Wise, "Passband and stepband properties of median filters," *IEEE Trans. Acoustic, Speech, Signal Processing*, Vol. ASSP-29, Dec. 1981, pp. 1136-1141.

[Gil81] W. K. Giloi, "A complete taxonomy of computer architecture based on the abstract data type view," *IFIP Workshop on Taxonomy in Computer Architecture*, June 1981, pp. 19-38.

[GoJ80] M. J. Gonzalez, Jr., and B. W. Jordan, Jr., "A framework for the quantitative evaluation of distributed computer systems," *IEEE Trans. Computers*, Vol. C-29, No. 12, Dec. 1980, pp. 1087-1094.

[GoL71] B. Gold, I. L. Lebow, P. G. McHugh, and C. M. Rader, "The FDP, a fast programmable signal processor," *IEEE Trans. Computers*, Vol. C-20, No. 1, Jan. 1971, pp. 33-38.

[GoL73] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *1st Annual Int'l. Symp. Computer Architecture*, Dec. 1973, pp. 21-28.

[GoM82] G. H. Goble and M. H. Marsh, "A dual processor VAX* 11/780," *IEEE 9th Annual Symp. on Computer Architecture*, Apr 1982, pp. 291-298.

---

*VAX is a trademark of Digital Equipment Corp.

[Gon78] M. J. Gonzales, Jr., "Quantitative evaluation of distributed computer systems," *2nd Rocky Mt. Symp. on Microcomputers: Systems, Software, Architecture*, Aug. 1978, pp. 125-130.

[Hac72] M. Hack, "Analysis of Production Schemata by Petri Nets," M.S. Thesis, Department of Electrical Engineering, MIT, 1972.

[Han75] P. Brinch Hansen, "The programming language Concurrent Pascal," *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 199-207.

[Han77a] P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[Han77b] W. Händler, "The impact of classification schemes on computer architecture," *1977 Int'l. Conf. Parallel Processing*, Aug. 1977, pp. 7-15.

[Han81] W. Händler, "Standards, classification, and taxonomy; experiences with ECS," *IFIP Workshop on Taxonomy in Computer Architecture*, June 1981, pp. 39-75.

[Han81a] P. Brinch Hansen, "Edison -- a multiprocessor language," *Software -- Practice and Experience*, Vol. 11, No. 4, April 1981, pp. 325-361.

[Han81b] P. Brinch Hansen, "The design of Edison," *Software -- Practice and Experience*, Vol. 11, No. 4, April 1981, pp. 363-396.

[Han81c] P. Brinch Hansen, "Edison programs," *Software -- Practice and Experience*, Vol. 11, No. 4, April 1981, pp. 397-414.

[Hig72] L. C. Higbie, "The OMEN computers: associative array processors," *COMPCON '72 Digest*, 1972, pp. 287-290.

[HoC70] A. Holt and F. Commoner, "Events and conditions" Applied Data Research, New York 1970; Also *Record of Project MAC Conf. on Concurrent Systems and Parallel Computation*, New York: ACM, 1970.

[HoJ81] R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms*, Adam Hilger Ltd, Bristol, 1981.

[HoS68] A. Holt, H. Saint, R. Shapiro, and S. Warshall, "Final report of the information system theory project," Technical Report RADC-TR-68-305 Rome Air Development Center, Griffiss AFB, New York, 1968.

[Hoa78] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, Aug. 1978, pp. 666-677.

[Hu62] M. K. Hu, "Visual pattern recognition by moment invariants," *IRE Trans. Info. Theory*, Vol. IT-8, Feb. 1962, pp. 179-187.

[HwB81] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, Class Notes, 1981, to be published McGraw-Hill, 1984.

[JaG82] M. Jazayeri, G. Ghezzi, D. Hoffman, D. Middleton and M. Smotherman, "Design and implementation of a language for communicating sequential processes," *Third Int'l. Conf. on Distributed Computing Systems*, Oct. 1982, pp. 173-180.

[JeB82] R. M. Jenevein and J. C. Browne, "A control processor for a reconfigurable array computer," *9th Annual Int'l. Symp. Computer Architecture*, Apr. 1982, pp. 81-89.

[Jen79] K. Jensen, "Coloured Petri nets and the invariant method" Technical Report DAIMI PB-104 C. S. Dept., Aarhus University, Denmark. 1979.

[Joe68] A. E. Joel, Jr., "On permutation switching networks," *Bell System Technical Journal*, Vol. 47, No. 5, June 1968, pp. 813-822.

[KaK79] S. I. Kartashev and S. P. Kartashev, "A multicomputer system with dynamic architecture," *IEEE Trans. Computers*, Vol. C-28, Oct. 1979, pp. 704-720.

[KaP80] R. N. Kapur, U. V. Premkumar, and G. J. Lipovski, "Organization of the TRAC processor-memory subsystem," *AFIPS 1980 Nat'l. Computer Conf.*, May 1980, pp. 623-629.

[KeF78] J. Keng and K. S. Fu, "A special purpose architecture for image processing," *1978 IEEE Comp. Soc. Conf. Pattern Recognition and Image Processing*, June 1978, pp. 287-290.

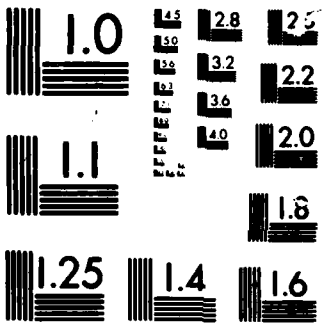
[KeR78] B. W. Kernigham and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Inc, Englewood Cliffs, NJ, 1978.

[Kea76] T. J. Keating, "Analytical photogrammetry from digitized image densities," *XIII Congress*, Int'l. Soc. for Photogrammetry, 1976.

[Knu73] D. E. Knuth, *The Art of Computer Programming*, Vols. 1 (1968) and 3 (1973), Addison-Wesley Publishing Co., Reading, Mass.

AD-A167 316 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR.. (U) PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING L J SEIGEL ET AL. 4/4

UNCLASSIFIED MAR 83 TR-EE-83-11 ARO-18790.17-EL-APP-A F/G 9/2 NL



MICROCOPY CHART

[Kos73] S. Kosaraju, "Limitations of Dijkstra's semaphore primitives and Petri Nets," *Operating System Review*, Vol. 7, No. 4, Oct. 1973, pp. 122-126.

[Kru73] B. Kruse, "A parallel picture processing machine," *IEEE Trans. Computers*, Vol. C-22, No. 12, Dec. 1973, pp. 1075-1087.

[Kry81] A. Krygiel, "Synchronous nets for single instruction stream - multiple data stream computers," *Proc. 1981 Int'l. Conf. Parallel Processing*, Aug. 1981.

[KuP79] D. J. Kuck and D. A. Padua, "High-speed multiprocessors and their compilers," *1979 Int'l. Conf. Parallel Processing*, Aug. 1979, pp. 5-16.

[KuS82] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int'l. Conf. Parallel Processing*, Aug. 1982, pp. 353-362.

[Kuc77] D. C Kuck, "A survey of parallel machine organization and programming", *ACM Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 29-59.

[Kuc78] D. J. Kuck, *The Structure of Computers and Computations*, Vol. I, John Wiley and Sons, Inc., NY, 1978.

[Kuc80] D. J. Kuck, "High speed machines and their compilers," in *Parallel Processing Systems - An Advanced Course*, Cambridge Univ. Press, Cambridge, D. J. Evans, Ed., Sept. 1980, pp. 193-214.

[LaS76] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," *IEEE Trans. Computers*, Vol. C-25, Jan. 1976, pp. 55-65.

[Law75] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Computers*, Vol. C-24, Dec. 1975, pp. 1145-1155.

[LeG68] K. N. Levitt, M. W. Green, and J. Goldberg, "A study of the data commutation problems in a self-repairable multiprocessor," *AFIPS 1968 Spring Joint Computer Conf.*, April 1968, pp. 515-527.

[Len78] J. Lenfant, "Parallel permutations of data: a Benes network control algorithm for frequently used permutations," *IEEE Trans. Computers*, Vol. C-27, July 1978, pp. 637-647.

[Les75] V. R. Lesser, "Parallel processing in speech understanding systems," in *Speech Understanding*, D. R. Reddy, ed., Academic Press, New York, 1975.

[LiL82] J. E. Lilienkamp, D. H. Lawrie, P. Yew, "A fault tolerant interconnection network using error correcting codes," *1982 Int'l. Conf. Parallel Processing*, Aug. 182, pp. 123-125.

[LiT77] G. J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," *1977 Int'l. Conf. Parallel Processing*, Aug. 1977, pp. 165-174.

[Lie79] B. H. Liebowitz, *Distributed Processing Overview*, notes and viewgraphs from tutorial presented at the First Int'l. Conf. on Distributed Computing Systems, Oct. 1979.

[Lim82] W. Y. Lim, "A test strategy for packet switching networks," *1982 Int'l. Conf. Parallel Processing*, Aug. 1982, pp. 96-98.

[Lip70] G. J. Lipovski, "The architecture of a large associative processor," *AFIPS 1970 Spring Joint Computer Conf.*, May 1970, pp. 385-396.

[MaM81] J. Maeng and M. Malek, "A comparison connection assignment for self-diagnosis of multiprocessor systems," *11th Annual Int'l. Symp. Fault-Tolerant Computing*, June 1981, pp. 173-175.

[McA80] R. J. McMillen, G. B. Adams III, and H. J. Siegel, "Permuting with the augmented data manipulator network," *18th Annual Allerton Conf. Communication, Control, and Computing*, Oct. 1980, pp. 544-553.

[McS80a] R. J. McMillen and H. J. Siegel, "MIMD machine communication using the augmented data manipulator network," *7th Annual Int'l. Symp. Computer Architecture*, May 1980, pp. 51-58.

[McS80b] R. J. McMillen and H. J. Siegel, "The hybrid cube network," *Distributed Data Acquisition, Computing, and Control Symp.*, Dec. 1980, pp. 11-22.

[McS82a] W. C. McDonald and R. W. Smith, "A flexible distributed testbed for real-time applications," *Computer*, Vol. 15, Oct. 1982, pp. 25-39.

[McS82b] R. J. McMillen and H. J. Siegel, "Performance and fault tolerance improvements in the inverse augmented data manipulator network," *9th Annual Int'l. Symp. Computer Architecture*, Apr. 1982, pp. 63-72.

[McS82c] R. J. McMillen and H. J. Siegel, "A comparison of cube type and data manipulator type networks," *3rd Int'l. Conf. Distributed Computing Systems*, Oct. 1982, pp. 614-621.

[McS82d] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Computers*, Vol. C-31, Dec. 1982, pp. 1202-1214.

[Mcc73] B. H. McCormick, "The Illinois pattern recognition computer - ILLIAC III," *Pattern Recognition: Introduction and Foundations*, J. Sklansky, ed., Dowden, Hutchinson, and Ross, Inc., Stroudsburg, PA, 1973.

[MiR81] O. R. Mitchell, A. P. Reeves, and K. S. Fu, "Shape and texture measurements for automated cartography," *1981 IEEE Comp. Soc. Conf. Pattern Recognition Image Processing*, Aug. 1981, pp. 367.

[Mil79] D. L. Milgram, "Region extraction using convergent evidence," *Computer Graphics and Image Processing*, Vol. 11, 1979, pp. 1-12.

[NaS80] J. J. Narraway and K. So, "Fault diagnosis in inter-processor switching networks," *1980 Int'l. Conf. Circuits and Computers*, Oct. 1980, pp. 750-753.

[Nut77] G. J. Nutt, "Microprocessor implementation of a parallel processor," *4th Annual Symp. Computer Architecture*, Mar. 1977, pp. 147-152.

[OpT71a] D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangeable switching networks - Part I: control algorithm," *Bell System Technical Journal*, Vol. 50, No. 5, May-June 1971, pp. 1579-1600.

[OpT71b] D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangeable switching networks - Part II: enumeration studies and fault diagnosis," *Bell System Technical Journal*, Vol. 50, No. 5, May-June 1971, pp. 1601-1618.

[Orc76] S. E. Orcutt, "Implementation of permutation functions on Illiac IV-type computers," *IEEE Trans. Computers*, Vol. C-25, No. 9, Sept. 1976, pp. 929-936.

[Ove82] A. L. Overvig, "The Simulation of the Generalized Cube Interconnection Network", MSEE Thesis, Purdue University, Aug. 1982, p. 251.

[Pap65] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 1965, pp. 528-535.

[PaR82] D. S. Parker and C. S. Raghavendra, "The Gamma network: a multiprocessor network with redundant paths," *9th Annual Int'l. Symp. Computer Architecture*, Apr. 1982, pp. 73-80.

[Pat79] J. H. Patel, "Processor-memory interconnections for multiprocessors," *6th Annual Int'l. Symp. Computer Architecture*, April 1979, pp. 168-177.

[PeB74] J. L. Peterson and T. H. Bredt,"A comparison of models of parallel computation," *Information Processing 74*, 1974, pp. 466-470.

[Pea77] M. C. Pease, "The indirect binary n-cube multiprocessory array," *IEEE Trans. Computers*, Vol. C-26, No. 5, May 1977, pp. 458-473.

[Pet62] C. Petri, "Kommunikation mit automaten" Ph.D. Dissertation, University of Bonn, West Germany, 1962 (In German); Also "Communication with automata;" (translated by C. F. Greene Jr.) Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1, Rome Air Development Center, Griffiss AFB, New York, 1966.

[Pet77] J. L. Peterson, "Petri nets," *ACM Computing Surveys*, Vol. 9, No. 3, 1977, pp. 223-252.

[Pet81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 31-78.

[PrK80a] U. V. Premkumar, R. Kapur, M. Malek, G. J. Lipovski, and P. Horne, "Design and implementation of the banyan interconnection network in TRAC," *AFIPS 1980 Nat'l. Computer Conf.*, June 1980, pp. 643-653.

[PrK80b] D. K. Pradhan and K. L. Kodandapani, "A uniform representation of single- and multistage interconnection networks used in SIMD machines," *IEEE Trans. Computers*, Vol. C-29, Sept. 1980, pp. 777-791.

[PrM66] J. M. S. Prewitt and M. L. Mendelsohn, "The analysis of cell images," *Annals N.Y. Academy of Science*, Vol. 128, 1966, pp. 1035-1053.

[PrR81] D. K. Pradhan and S. M. Reddy, "A fault-tolerant communication architecture for distributed systems," *11th Annual Int'l. Symp. Fault-Tolerant Computing*, June 1981, pp. 214-220.

[Pra78] W. K. Pratt, *Digital Image Processing*, Wiley, New York, 1978, pp. 93-120.

[Pra81] D. K. Pradhan, "Interconnection topologies for fault-tolerant parallel and distributed architectures," *1982 Int'l. Conf. Parallel Processing*, Aug. 1981, pp. 238-242.

[Pyl81] I. C. Pyle, *The ADA Programming Language*, Prentice Hall Int'l., London, 1981.

[RaL77] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *ACM Computing Surveys*, Mar. 1977, pp. 61-102.

[RaM80] B. D. Rathi and M. Malek, "Fault diagnosis of networks," *Distributed Data Acquisition, Computing, and Control Symp.*, Dec. 1980, pp. 110-119.

[ReO82] D. E. Reynolds and G. P. Otto, "Software tools for CLIP4," Dept. of Physics and Astronomy, University College, London, 1982.

[RoP77] D. Rohrbacker and J. L. Potter, "Image processing with the STARAN parallel computer," *Computer,* Vol. 10, No. 8, Aug. 1977, pp. 54-59.

[RuF76] S. Ruben, R. Faiss, J. Lyon, and M. Quinn, "Application of a parallel processing computer in LACIE," *1976 Int'l. Conf. Parallel Processing,* Aug. 1976, pp. 24-32.

[Rus78] R. M. Russell, "The Cray-1 computer system," *Communications of the ACM,* Vol. 21, Jan. 1978, pp. 63-72.

[SRI79] Stanford Research Institute, "Machine intelligence research applied to industrial automation," Menlo Park, Ca, 94025, Aug. 1979.

[SeU80] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 Nat'l. Computer Conf.*, June 1980, pp. 631-641.

[ShH80] J. P. Shen and H. P. Hayes, "Fault tolerance of a class of connecting networks," *7th Annual Int'l. Symp. Computer Architecture,* Apr. 1980, pp. 61-71.

[ShS70] R. Shapiro and H. Saint, "A new approach to optimization of sequencing decisions" *Annual Review in Automatic Programming,* Vol. 6, Part 5, 1970, pp. 257-288.

[Sho82] J. P. Shen, "Fault tolerance analysis of several interconnection networks," *1982 Int'l. Conf. Parallel Processing,* Aug. 1982, pp. 102-112.

[Sho70] W. Shooman, "Orthogonal processing," *Parallel Processor System Techniques and Applications,* New York, Spartan Books, 1970.

[Sho73] J. E. Shore, "Second thoughts on parallel processing," *Comput. and Elect. Eng.*, Vol. 1, Pergamon Press, 1973, pp. 95-109.

[SiM78] H. J. Siegel, P. T. Mueller, Jr., and H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor system," *1978 Int'l. Conf. Parallel Processing,* Aug. 1978, pp. 9-17.

[SiM79a] H. J. Siegel, R. J. McMillen, and P. T. Mueller, Jr., "A survey of interconnection methods for reconfigurable parallel processing systems," *AFIPS 1979 Nat'l. Comp. Conf.*, June 1979, pp. 529-542.

[SiM79b] L. J. Siegel, P. T. Mueller, Jr., and H. J. Siegel, "FFT algorithms for SIMD machines," *17th Annual Allerton Conf. Communication, Control, and Computing,* University of Illinois, Oct. 1979, pp. 1006-1015.

[SiM81] H. J. Siegel and R. J. McMillen, "The cube network as a distributed processing test bed switch," *2nd Int'l. Conf. Distributed Computing Systems,* Apr. 1981, pp. 377-387.

[SiM81a] H. J. Siegel and R. J. McMillen, "Using the augmented data manipulator network in PASM," *Computer,* Vol. 14, Feb. 1981, pp. 25-33.

[SiM81b] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," *Computer,* Vol. 14, Dec. 1981, pp. 65-76.

[SiS78] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," *5th Annual Int'l. Symp. Computer Architecture,* Apr. 1978, pp. 223-229.

[SiS79] H. J. Siegel, L. J. Siegel, R. J. McMillen, P. T. Mueller, Jr., and S. D. Smith, "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," *1979 IEEE Comp. Soc. Conf. Pattern Recognition and Image Processing,* Aug. 1979, pp. 214-224.

[SiS81a] H. J. Siegel, L. J. Siegel, F. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers,* Vol. C-30, Dec. 1981, pp. 934-947.

[SiS81b] L. J. Siegel, H. H. Siegel, P. H. Swain, et al., "Parallel image processing/feature extraction: interim report for fiscal 1981, volume I. algorithms", School Elec. Eng., Purdue Univ., West Lafayette, IN, Tech. Rep. TR-EE 81-35, Oct. 1981.

[SiS82] L. J. Siegel, H. J. Siegel, and P. H. Swain, "Performance measures for evaluating algorithms for SIMD machines," *IEEE Trans. Software Engineering,* Vol. SE-8, July 1982, pp. 319-331.

[Sie77] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Computers,* Vol. C-26, Feb. 1977, pp. 153-161.

[Sie79] H. J. Siegel, "Interconnection networks for SIMD machines," *Computer,* Vol. 12, June 1979, pp. 57-65.

[Sig00] Signal Processing Systems, Waltham, Mass.

[SmS78] S. D. Smith and H. J. Siegel, "Recirculating, pipelined, and multistage SIMD interconnection networks," *1978 Int'l. Conf. Parallel Processing*, Aug. 1978, pp. 206-214.

[Smi78] B. J. Smith, "A pipelined, shared resource MIMD computer," *1978 Int'l. Conf. Parallel Processing*, Aug. 1978, pp. 6-8.

[Smi81a] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," *SPIE*, Vol. 298, Real Time Signal Processing IV, Aug. 1981, pp. 241-248.

[Smi81b] S. D. Smith, "LSI design considerations for multistage interconnection networks for parallel processing systems," *14th Annual Hawaii Int'l. Conf. System Sciences*, Vol. 1, Jan. 1981, pp. 219-227.

[SoR80] S. Sowrirajan and S. M. Reddy, "A design for fault-tolerant full connection networks," *1980 Conf. Information Sciences and Systems*, Princeton Univ., Mar. 1980, pp. 536-540.

[Sto71] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Computers*, Vol. C-20, Feb. 1971, pp. 153-161.

[Sto77] R. A. Stokes, "Burroughs scientific processor," *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Eds., Academic Press, New York, 1977.

[Sto80] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture*, 2nd edition, edited by H. S. Stone, Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[Sto82] P. D. Stotts, Jr., "A comparative survey of concurrent programming languages", *SIGPLAN Notices*, Vol. 17, No. 10, Oct. 1982, pp. 50-61.

[SuB77] H. Sullivan, T. R. Bashkow, and K. Klappholz, "A large scale homogeneous, fully distributed parallel machine," *4th Annual Symp. Computer Architecture*, Mar. 1977, pp. 105-124.

[SuR82] R. E. Suciu and A. P. Reeves, "A comparison of differential and moment based edge detectors," *1982 IEEE Comp. Soc. Conf. Pattern Recognition and Image Processing*, June 1982, pp. 97-102.

[SwF77] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a modular, multi-microprocessor," *AFIPS 1977 Nat'l. Comp. Conf.*, June 1977, pp. 637-644.

[SwS80] P. H. Swain, H. J. Siegel, and J. El-Achkar, "Multiprocessor implementation of image pattern recognition: a general approach," *5th Int'l. Conf. Pattern Recognition*, Dec. 1980, pp. 309-317.

[Tea80] M. R. Teague, "Image analysis via the general theory of moments," *Journal Optical Soc. Am.*, Vol. 70, Aug. 1980, pp. 920-933.

[ThM79] K. J. Thurber and G. M. Masson, *Distributed-Processor Communication Architecture*, Lexington Brooks, Lexington, MA, 1979.

[The74] D. J. Theis, "Vector supercomputers," *Computer*, Vol. 7, No. 4, Apr. 1974, pp. 52-61.

[Tho70] J. E. Thornton, *Design of a Computer, the Control Data 6600*, Scott, Foresman and Co., Glenview, IL, 1970.

[Thu74] K. J. Thurber, "Interconnection networks - a survey and assessment," *AFIPS 1974 Nat'l Computer Conf.*, May 1974, pp. 909-919.

[TrL79] A. R. Tripathi and G. J. Lipovski, "Packet switching in banyan networks," *6th Annual Int'l. Symp. Computer Architecture*, Apr. 1979, pp. 160-167.

[ViC78] C. R. Vick and J. A. Cornell, "PEPE architecture - present and future," *AFIPS 1978 Nat'l. Computer Conference*, June 1978, pp. 981-1002.

[WaM79] T. P. Wallace and O. R. Mitchell, "Local and global shape description of two- and three-dimensional objects," School Elec. Eng., Purdue Univ., West Lafayette, IN, Tech. Rep. TR-EE 79-43, Sept. 1979.

[WaM80] T. P. Wallace and O. R. Mitchell, "Analysis of three-dimensional movement using fourier descriptors," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, No. 6, Nov. 1980, pp. 583-588.

[WaS82] M. R. Warpenburg and L. J. Siegel, "SIMD image resampling," *IEEE Trans. Computers*, Vol. C-31, Oct. 1982, pp. 934-942.

[WaW80] T. P. Wallace and P. A. Wintz, "An efficient three-dimensional aircraft recognition algorithm using normalized Fourier descriptors," *Comput. Graphics and Image Processing*, Vol. 13, 1980, pp. 99-126.

[Wak68] A. Waksman, "A permutation network," *Journal of the ACM*, Vol. 15, Jan. 1968, pp. 159-163.

[Wel77] H. 0. Welch, "Numerical weather prediction in the PEPE parallel processor," *1977 Int'l. Conf. Parallel Processing,* Aug. 1977, pp. 186-192.

[Wid76] L. C. Widdoes, Jr., "The Minerva multi-microprocessor," *3rd Annual Int'l. Symp. Computer Architecture,* Jan. 1976, pp. 34-39.

[Wir77a] N. Wirth, "Modula: a language for modular multiprogramming," *Software -- Practice and Experience,* Vol 7., No. 1, Jan.-Feb. 1977, pp. 3-35.

[Wir77b] N. Wirth, "The use of Modula," *Software -- Practice and Experience,* Vol. 7, No. 1, Jan.-Feb. 1977, pp. 37-65.

[Wir77c] N. Wirth, "Design and implementation of Modula," *Software -- Practice and Experience,* Vol 7., No. 1, Jan.-Feb. 1977, pp. 67-84.

[WuB72] W. A. Wulf and C. G. Bell, "C.mmp--a multi-miniprocessor," *AFIPS 1972 Fall Joint Computer Conf.,* Dec. 1972, pp. 765-777.

[WuF78] C. Wu and T. Feng, "Routing techniques for a class of multistage interconnection networks," *1978 Int'l. Conf. Parallel Processing,* Aug. 1978, pp. 197-205.

[WuF79a] C. Wu and T. Feng, "The reverse-exchange interconnection network," *1979 Int'l. Conf. Parallel Processing,* Aug. 1979, pp. 160-174.

[WuF79b] C. Wu and T. Feng, "Fault-diagnosis for a class of multistage interconnection networks," *1979 Int'l. Conf. Parallel Processing,* Aug. 1979, pp. 269-278.

[WuF80] C. Wu and T. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Computers,* Vol. C-29, Aug. 1980, pp. 694-702.

[WuL82] C. Wu, W. Lin, and M. Lin, "Distributed circuit switching starnet," *1980 Int'l. Conf. Parallel Processing,* Aug. 1982, pp. 26-33.

END

FILMED

6-86

DTIC