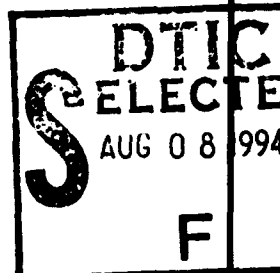


TEC-0044

AD-A282 960



Image Understanding Architecture Prototype Evaluation and Development



Charles C. Weems
Allen R. Hanson
Martin Herbordt
Deepak Rana

Edward M. Riseman
James Burrill
Michael Scudder

University of Massachusetts
Computer Science Department
Box 34610
Amherst, MA 01003-4610

June 1993

33198

94-24859



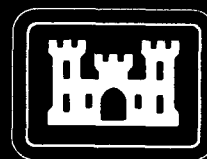
Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 5

Prepared for:
Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

94 8 05

Monitored by:
U.S. Army Corps of Engineers
Topographic Engineering Center
7701 Telegraph Road
Alexandria, Virginia 22315-3864



US Army Corps
of Engineers
Topographic
Engineering Center

T

E

C



**Destroy this report when no longer needed.
Do not return it to the originator.**

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The citation in this report of trade names of commercially available products does not constitute official endorsement or approval of the use of such products.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 1993		3. REPORT TYPE AND DATES COVERED Final Report Sep. 1989 - Sep. 1992
4. TITLE AND SUBTITLE Image Understanding Architecture Prototype Evaluation and Development			5. FUNDING NUMBERS DACA76-89-C-0016	
6. AUTHOR(S) Charles C. Weems, Edward M. Riseman, Allen R. Hanson James Burrill, Martin Herbordt, Michael Scudder, Deepak Rana				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Massachusetts Computer Science Department Box 34610 Amherst, MA 01003-4610			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Advanced Research Projects Agency 3701 North Fairfax Drive, Arlington, VA 22203-1714 U.S. Army Topographic Engineering Center 7701 Telegraph Road., Alexandria, VA 22315-3864			10. SPONSORING/MONITORING AGENCY REPORT NUMBER TEC-0044	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The primary goal of this research was to complete the development and proof-of-concept prototype of a 1/64th slice of a parallel architecture to support image understanding, and to test and evaluate the architecture so the next generation Image Understanding Architecture (IUA) can be specified. The majority of the hardware effort has taken place at Hughes Research Laboratories, Malibu, California, although UMass has principle responsibility for designing the IUA architecture and has undertaken some smaller portions of hardware development. The majority of the software effort took place at UMass. Software efforts included development of a detailed software simulator for the IUA, a FORTH interpreter, and Apply and C compilers for the low-level processor of the IUA. Also developed were run-time support libraries, diagnostics, and vision algorithms. Some effort was spent implementing the ARPA IU Benchmark and evaluating the results of the exercise. Included in this report is a summary of accomplishments, an overview of the IUA design and the trade-offs addressed in its development, the design of the intermediate level interconnection network, the design of the IUA simulator and the ARPA benchmark.				
14. SUBJECT TERMS Image Understanding Architecture (IUA), Knowledge-Based Vision, Real-Time Computer Vision, Software Simulator, Parallel Processor			15. NUMBER OF PAGES 83	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNLIMITED	

Table of Contents

Abstract.....	ii
Table of Contents	iii
List of Figures	v
List of Tables.....	vi
Preface.....	vii
Executive Summary	1
1. Introduction.....	8
2. Overview of the Image Understanding Architecture (IUA).....	10
3. Tradeoffs in the Development of the Low-Level Processor.....	10
3.1 Neighbor Communication	10
3.2 Memory	12
3.3 Response and Activity Control	12
3.4 Response Count	12
3.5 Input/Output	14
3.6 Changes to the CAAPP Specification for the Second Generation.....	16
4. Design of the ICAP Interconnection Network	17
4.1 The Parallel Communication Switch	18
4.2 The Second Generation ICAP Communication Network.....	21
5. The IUA Simulator and Software Environment.....	24
5.1 CAAPP-specific Parallel Extension to C.....	31
5.2 Apply.....	32
5.3 Libraries	33
6. The Image Understanding Benchmark.....	33
6.1 Review of Previous Vision Benchmark Efforts.....	34
6.2 Benchmark Task Overview.....	37
6.3 Benchmark Philosophy and Rationale.....	40
6.4 Results and Analysis.....	43
6.5 Sequential Solution.....	44
6.6 Alliant FX-80 Solution.....	48
6.7 Image Understanding Architecture.....	48
6.8 Aspex ASP.....	49
6.9 Sequent Symmetry 81.....	53
6.10 Warp.....	54
6.11 Connection Machine.....	54
6.12 Intel iPSC-2.....	58
6.13 Comparative Performance Summary.....	59
6.14 Recommendations for Future Benchmarks.....	61
6.15 Benchmark Efforts Under This Contract.....	63

7. Routing in the CAAPP.....	65
7.1 Outline of the Mesh Greedy Routing Algorithm.....	65
7.2 The MGRA on the Basic Mesh-Connected Processor Model.....	66
7.3 The MGRA with Longer Queues.....	66
7.4 The MGRA on a Mesh with Broadcast Buses.....	67
7.5 The MGRA on a Mesh with Reconfigurable Buses.....	67
7.6 The Four Channel MGRA on the Basic Model.....	67
7.7 The Coterie Greedy Routing Algorithm.....	68
7.8 Many-to-One Routing.....	68
8. Conclusions	69
9. References	72
Appendix: Polymorphic Multiple Processor Networks.....	Follows 75

List of Figures

1. IUA Overview	11
2. Coterie Network	13
3. Memory Architecture	14
4. I/O Architecture	15
5. A 64 x 64 Network Built with PARCOS Chips	20
6. PARCOS Chip Organization	20
7. IU Benchmark Model	39
8. IU Benchmark Intensity Image	39

Accession For	
NTIS - DRAVI	☒
DTIC - TAB	☐
Unannounced	☐
J. H. ...	
By	
Distribution	
Availability Codes	
Dist	Avail and/or Special
A-1	

List of Tables

1. Tanque Verde Benchmark Suite.....	3 5
2. Tasks from the First DARPA Image Understanding Benchmark.....	3 5
3. Distribution List for the Second DARPA Benchmark.....	3 7
4. Steps that Compose the Integrated Image Understanding Benchmark.....	4 2
5. Sun-3/160 Results.....	4 5
6. Sun-3/260 Results.....	4 6
7. Sun-4/260 Results.....	4 7
8. Alliant FX-80 Single Processor Results.....	5 0
9. Alliant FX-80 Results with Eight Processors.....	5 1
10. Image Understanding Architecture Results.....	5 2
11. Aspex ASP Results.....	5 5
12. Sequent Symmetry 81 Results.....	5 6
13. Results for the Warp.....	5 7
14. Results for the Connection Machine on the Low-Level Portion.....	5 8
15. iPSC-2 Results for Median Filter and Sobel Steps.....	5 9
16. Distribution of Processing Time for Data Set Sample.....	6 0
17. Distribution of Processing Time for Data Set Test.....	6 0
18. Distribution of Processing Time for Data Set Test2.....	6 0
19. Distribution of Processing Time for Data Set Test3.....	6 1
20. Distribution of Processing Time for Data Set Test4.....	6 2
21. Additional Distribution of the Second DARPA Benchmark.....	6 4

PREFACE

This research is sponsored by the Advanced Research Projects Agency (ARPA) and monitored by the U.S. Army Topographic Engineering Center (TEC) under Contract DACA76-89-C-0016, titled "Image Understanding Architecture Prototype Evaluation and Development". The ARPA Program Manager is Dr. Oscar Firschein, and the TEC Contracting Officer's Representative is Ms. Lauretta Williams.

Executive Summary

The goals of the Image Understanding Architecture Prototype and Evaluation and Development contract were to complete the construction of a proof of concept prototype of a heterogeneous parallel processor to support machine vision, and to develop software support and demonstration applications for the machine with the purpose of evaluating the architecture so that specifications and a design for a second generation of the architecture could be developed. These goals have been essentially achieved, although the schedule for the program was delayed for various reasons. Primarily, the delays were caused by problems with the MOSIS integrated circuit brokerage service. Ours was one of the first major research efforts to try building large chips via MOSIS. As with any pioneering effort, we encountered difficulties which have since been resolved. The chip fabrication delays also caused slippage in other portions of the project schedule, which in turn required the timetable to be stretched, which then resulted in some additional costs. Delays in receiving funding and legal issues relating to nondisclosure agreements also incurred some delays.

In the area of software, as required by the contract, parallel programming languages were developed for the CAAPP level of the architecture, based on the FORTH and C programming languages, and on the application-specific Apply language. The software environment for the system was enhanced, including adding functionality to the debugger for the CAAPP and ICAP levels, adding greater support for I/O, developing an improved memory management system for the CAAPP level, extending the simulators to provide more support for timing and instruction set utilization analysis, and converting the Sun-Tools-based programmer's interface to use X-windows.

Various vision algorithms were implemented for the architecture, using the simulators. A parallel algorithm was developed for the Nagin-Kohler region segmentation, but it was determined that the algorithm was not practical to implement -- the original algorithm depended greatly on computations and modes of communication that were not well supported in the prototype IUA (in particular, the heavy use of floating point at the low level and the movement of large blocks of data between processors at the intermediate level). Although the result of the algorithm implementation was negative, the experience had value in that it pointed out areas of the ICAP architecture that could be improved in the second generation design.

The design of a communication network for the Intermediate Level of the second generation IUA was completed. A brief summary of the design is included in this report, and the full Ph.D. thesis that resulted in the design and several alternatives is attached as an appendix.

As can be seen from the list of activities below, a significant portion of our effort has been spent on cooperating with DARPA and strategic computing contractors to share our results and explore avenues for transferring technology.

In the area of hardware, Hughes essentially completed a 1/64th scale proof-of-concept prototype of the Image Understanding Architecture. It contains 4096 CAAPP processing elements and 64 ICAP processors. A simple array control unit has been built that allows the machine to be tested and demonstrated with small but representative applications. The machine is fully functional in that all of the memory, communication channels, and processing elements operate correctly, although at a slower speed than was our original target.

Although the IUA prototype hardware was not completed by Hughes until after the funding for this contract was cut off, a no-cost extension of the program eventually saw the hardware finished. UMass worked with Hughes to implement the IUA software tools and simulator environment, first, through their IMS tester, and then, through their simple array controller on the IUA prototype.

Delivery of the prototype was accepted at Hughes Research Laboratories rather than having the hardware physically delivered to the University of Massachusetts, for two reasons: First, because of delays in the completion of the prototype and the expectation that development of a second generation system would begin about one year into this program, it was decided that Hughes should continue debugging of the prototype hardware to identify any further problems that might affect the development of the second generation. Second, it was determined that the prototype hardware was, at that early stage of operation, too fragile for use in a university laboratory. Another factor affecting the decision was that the University would have had to develop an interface board for its IMS tester to act as the controller for the IUA, but Hughes was willing to build the simple controller mentioned above in return for being allowed to use the prototype hardware in the development of the second generation system. At this point, Hughes is nearly done with the prototype system and we have asked them to put it into a condition that will allow us to take physical delivery. In the meantime, we have accomplished the goals of this program through the use of our extended simulators, and through the use of the hardware at Hughes.

The use of a multiprocessor to support high-level vision processing was examined through consulting with several parallel efforts. These included the implementation of a parallel common LISP and a general blackboard system with that compiler, for the Sequent Symmetry multiprocessor. Another parallel effort with which we consulted was the development of a pose estimation procedure for the Sequent. Although we were unable to directly instrument any of these systems as planned, we obtained valuable data from their own experience and instrumentation. We also evaluated a prototype Xenologic X-1

symbolic (Prolog) processor board as a potential high-level processor for the IUA.

UMass worked with Hughes and DARPA to develop a preliminary design for the second generation of the IUA. That design involved quadrupling the number of CAAPP processing elements on a chip and switching from the TMS-320C25 processor to the TMS-320C30 processor at the ICAP level. The newer TI processor provides double the number of bi-directional communication channels (two), adds a DMA controller, supports 32-bit integer and floating-point operations and operates at 16 MIPS instead of 5 MIPS. Most of the design effort involved the communication network for the ICAP level and the array control unit for the CAAPP level.

All of the goals of the contract effort were thus either completed or nearly completed. The near completion of some of the goals resulted from a decision by DARPA to cut off funding before the last of the contract options was completed. (Many of those goals were later completed under a separate DARPA contract that was awarded after the termination of this effort.)

The major accomplishments and activities performed during the period of this report are as follows:

1. UMass completed debugging of the IUA implementation of the DARPA benchmark. Its performance was enhanced as part of this process. Included in the benchmark were parallel algorithms for connected component labeling, k-curvature (determination of points of high curvature on a boundary), boundary tracing, grouping of corners into rectangles, rectangle parameter extraction, tree-based model matching, windowed Hough transform, windowed statistical measures, median filter, and Sobel edge extraction.
2. UMass implemented the Weymouth-Overton edge-preserving smoothing operator on the CAAPP simulator, using FORTH. Weymouth-Overton is the first step in the Nagin-Kohler region segmentation algorithm. Analysis of the Weymouth-Overton algorithm showed that its heavy dependence on floating point makes its performance unsuitable for real-time applications on the CAAPP (low-level IUA processor). A parallel algorithm for Nagin-Kohler was developed, but it was found that it requires distributed-control routing in the ICAP (intermediate-level IUA processor), which the prototype IUA does not support. We could emulate such routing through feedback and broadcast via the array controller, but doing so would sequentialize the processing. Also, Nagin-Kohler requires the transfer of histograms between successively larger spatial regions, and the 5 M-bit/second links of the TMS-320C25 will present a bottleneck to such large

transfers. It was thus determined that there was no useful point in actually carrying out the implementation of the algorithm, since its development alone had served the desired purpose of identifying areas in the ICAP communication network design that require work.

3. A generalized permutation routing algorithm has been developed for the CAAPP by UMass. Three different versions of the algorithm have been developed, and it is possible to determine the best version to use prior to executing the route. One version is for dense general permutation routes, another is for dense local routes, and another is for sparse routes. The performance varies from a few microseconds to a few milliseconds, depending on the density and the pattern of collisions during the route. That performance ranges then from 100 times faster to 10 times slower than a Connection Machine 2, but without having a dedicated router.
4. UMass developed an Apply compiler for its TI Explorer-based simulation system. The compiler was then reimplemented on the Sun-based simulator environment. Apply is an application-specific language, based on Ada syntax, that supports the development of parallel local-neighborhood convolution operators. It was originally developed at Carnegie Mellon University (CMU) for the Warp systolic array.
5. UMass purchased and installed an IMS ASIC test system to act as an interim controller for the IUA, following the lead of Hughes. The system will also be used to test custom chips (the feedback concentrator and ICAP parallel communication switch) developed at UMass.
6. UMass developed an interface to its IMS test system to support the transfer of instructions to the IMS so that it can act as an interim IUA controller. This development was a rather lengthy process because, in spite of assurances to the contrary, IMS did not support a GPIB interface to the Sun-4 platform at the time. Although the interface was theoretically simple, it was complicated by undocumented timing peculiarities in the IMS controller, which had to be identified through experimentation.
7. Articles about the CAAPP routing algorithm, the feedback concentrator chip, and the IUA in general were submitted to the International Conference on Pattern Recognition.
8. UMass began development of a revised DARPA benchmark. The monolithic code of the original version was divided into a set of separate subroutines, and a model and image generator was built to avoid the cost of distributing large data sets.

9. A new benchmark of eleven image processing routines from CMU was coded for the CAAPP and analyzed with the IUA simulator. Performance figures were reported to CMU and later published in a trade journal showing that the CAAPP alone was significantly faster than all other machines in nearly every test.
10. Drs. Weems, Riseman, Nash, and Shu briefed Lt. Col. Erik Mettala at DARPA about the IUA program.
11. An article on the CAAPP routing algorithm was submitted to the Symposium on Parallel Architectures and Algorithms. An article on the feedback concentrator was submitted to the Application Specific Array Processors conference.
12. Drs. Weems and Riseman participated in the 1990 DARPA IU Working group meeting in Scottsdale, AZ.
13. Dr. Weems gave a presentation on the IUA at the 1990 Joint Directors of Laboratories Meeting at Ft. Belvoir.
14. The Web library of Apply image processing routines was compiled, except for certain routines that depend on Warp assembly language, and multiresolution operations.
15. Drs. Weems, Riseman, Hanson, Nash, Shu, and Grinberg met with Lt. Col. Erik Mettala, Tice DeYoung, and Rand Waltzman at DARPA to plan the development of the next generation of the IUA.
16. Dr. Weems gave a presentation at a workshop associated with the first European Conference on Computer Vision. Dr. Riseman attended a joint DARPA/ESPRIT working group meeting associated with the same conference.
17. Dr. Weems gave a presentation on the IUA at the 1990 Departmental Research Review for Industry.
18. Drs. Weems, Riseman, and Mr. Burrill attended the DARPA IU Environments Workshop in Malibu. While there, a meeting was also held with Hughes staff to discuss IUA hardware and software development issues.
19. Documentation for the Apply compiler was completed.
20. Dr. Weems wrote two book chapters on the IUA and its programming environment, and on the reconfigurability of the IUA. He also wrote an article on the Architectural Requirements of Image Understanding for The

Proceedings of the IEEE, an article on the DARPA IU Benchmark for the Journal of Parallel and Distributed Computing, and began to assemble articles for a special issue of Machine Vision and Applications.

21. Drs. Weems, Riseman, Hanson, Nash and Shu, and Mr. Rana and Mr. Herbordt attended the 1990 International Conference on Pattern Recognition. Drs. Weems, Nash, and Shu also met to discuss IUA development strategy.
22. UMass has developed an X-Windows version of the IUA simulator to enhance both portability and remote access.
23. An article on Multi-guage emulation using the Coterie network was published in the 1990 Frontiers of Massively Parallel Processing conference.
24. Dr. Weems lead the writing of a five-author paper on the impact of Strategic Computing on the DARPA IU community for IEEE Expert.
25. An article on the CAAPP routing algorithm was written for the 1990 IU Workshop.
26. UMass designed a data-parallel extension to C that uses high-level language syntax, but does not completely hide the underlying architecture.
27. UMass was contacted by MasPar to explore cooperation in the development of a compatible data-parallel C compiler.
28. UMass worked with Hughes in the design of an array controller for the second generation IUA (which could also be used to drive the prototype system, with modest modifications).
29. UMass developed a microcode assembler for the array controller design to aid in its evaluation.
30. UMass developed a set of routines for self-test and diagnosis to be used with the IUA prototype.
31. Mr. Rana gave a presentation at the ASAP conference on the feedback concentrator chip.
32. Dr. Weems gave a presentation at the 1990 DARPA IU Workshop.
33. Dr. Weems gave a presentation at the first DARPA UGV Workshop.
34. Dr. Weems gave a presentation at the 1990 DARPA VLSI and High Performance Computing Contractor's Workshop.

35. Mr. Herbordt attended the 1990 Frontier's conference as a co-author of the Multi-guage paper.
36. The X-Windows simulator was ported from the Sun to a DecStation 5000 running Ultrix and DecWindows.
37. Work was begun on parallel line and curve grouping algorithms that will utilize the intermediate level of the IUA. Work was also begun on a motion analysis parallel algorithm that will operate on the low-level processor.
38. UMass developed a series of designs for the ICAP communication network that add successively greater sophistication and flexibility. The designs use the same hardware interface and require only that a single circuit board be replaced to upgrade from one to the next. We thus proposed to develop the simplest for initial use in the second generation IUA and upgrade as funding and technology permit.
39. UMass assisted Hughes in transporting the IUA software tools and environment to the IUA prototype, and several applications were transported as demonstrations of the hardware's functionality. These include the routing algorithm and parts of the DARPA IU Benchmark.

1. Introduction

Computer vision using color imagery requires a processor capable of accepting 23 megabytes of input per second, and interpreting it to construct a three dimensional model of the sensor's environment. An interpretation may require hundreds of objects of many different types to be identified. Vision researchers [Hanson, 1986] have shown that pattern recognition techniques, by themselves, are inadequate for this task. In fact, most of what we "see" in natural scenes is really inferred from partial information. In addition to sensory and knowledge-based processing it is useful to introduce a level of symbolic processing. Thus, vision researchers tend to classify algorithms and representations into three levels: low (sensory), intermediate (symbolic), and high (knowledge-based).

While it may be argued that a general-purpose processor can fulfill the requirements of vision, the goal of real-time performance necessitates the use of special-purpose processors. Another key to achieving real-time performance is processing at all levels simultaneously, which leads to the idea of linking together three different parallel processors. But because of the massive amount of communication between levels, and the requirement for flexible, top-down control, the different parallel processors must be designed from the start to be tightly coupled with each other. This analysis leads to the concept of the Image Understanding Architecture (IUA).

The primary goal of the IUA project (Contract DACA-76-86-C-0015) was to build a proof-of-concept prototype of a 1/64th slice of a parallel architecture to support real-time, knowledge-based image understanding, and develop the software support environment that will be needed to utilize the hardware. This follow-on effort (Contract DACA-76-89-C-0016) is intended to support initially the continued development of software while the hardware is being completed, including the extension of the software environment, and the development of additional parallel vision algorithms, and then the evaluation of the hardware and architecture so that specifications and the design of a second generation of the architecture can be developed.

The majority of the hardware effort has taken place at Hughes Research Laboratories, Malibu, California, under a subcontract that was part of the preceding effort. However, UMass has principle responsibility for the specification of the architecture. UMass also undertook some smaller portions of the hardware development (the feedback concentrator for the low and intermediate level arrays, and the communications switch for the intermediate level array) in the preceding effort, and lead the effort under this contract to design a new communication network for the intermediate level of the IUA. The majority of the software effort took place at UMass, although Hughes was also

involved in some software development, both in support of their hardware efforts, and in the form of algorithm development for specific applications on the IUA.

Our software efforts have included enhancement of the software simulation of the IUA and the FORTH interpreter for the low-level processor of the IUA. We have also developed a C compiler and an Apply compiler. The remainder of the software effort has been in the development of run-time support libraries, diagnostics, and vision algorithms. Effort was also spent on improving IUA performance on the Integrated DARPA Image Understanding Benchmark and starting to develop a new version of the benchmark.

2. Overview of the Image Understanding Architecture (IUA)

The Image Understanding Architecture [Weems, 1989] consists of three different, tightly coupled parallel processors (Figure 1). The low- and intermediate-levels are controlled by a dedicated Array Control Unit (ACU) that takes its directions from the high level. As Figure 1 indicates, each of these processors provides a different granularity and different modes of parallelism. We have built a 1/64th slice of the IUA as a proof-of-concept demonstration. The discussion that follows describes the full IUA, except where it is noted that a feature pertains only to the prototype.

At the high level the IUA is purely a MIMD parallel processor. In our original proposal, the high level was called the General Purpose Processor Array (GPPA) but has since been renamed the Symbolic Processing Array (SPA) to avoid confusion with scientific parallel processors. The low level, called the Content Addressable Array Parallel Processor (CAAPP) operates in pure SIMD or multi-associative mode, and the intermediate level operates in SPMD or MIMD mode. In the original proposal, the intermediate level was called the Intermediate and Communication Processor (ICP), but was later renamed the Intermediate Associative and Communication Processor (ICAP) to reflect the emphasis of associative processing on its design.

Briefly, the multi-associative and SPMD processing modes differ from the familiar SIMD and MIMD modes as follows. In multi-associative mode, the PE's execute a single instruction stream, but are arranged into disjoint groups, with each group able to locally broadcast values, and compute its own summary values in parallel with other groups. In SPMD (Single Program, Multiple Data) the processors execute the same program with autonomous instruction pointers so that they can branch independently.

3. Tradeoffs in the Development of the Low-Level Processor

Our analysis of low-level vision algorithms showed that the majority would best be served by a mesh-connected array, augmented with the features of an associative processor (i.e. global broadcast with local partial matches, activity control with global override, and dedicated response hardware) [Weems, 1984].

3.1 Neighbor Communication

One tradeoff is a four-way versus an eight-way mesh. We found that few algorithms take advantage of an eight-way mesh, and the increase in performance is small unless operations take place on eight inputs at once. Even

then, the improvement does not justify the resultant cost of tripling the number of I/O pins on the processor chip and at circuit board boundaries.

In addition to local communication, several low-level vision algorithms require communication between processors that are spatially distant in the mesh. We chose to enhance the mesh itself, so that no additional connectivity is required between processors. This scheme is similar to those proposed by [Kumar, 1985], [Miller, 1988], and [Li, 1987], and is a generalization of the propagate operation in the CLIP-4 [Duff, 1978], and the "flash-through" mode of the ILLIAC III [McCormick, 1963]. In our scheme, any contiguous group of processors in the mesh can be connected by a bus. For example, each region in an image could be electrically isolated from its neighbors, allowing local broadcast and some/none operations to occur simultaneously in all regions (Figure 2). An important result is that maximum or minimum values can be determined within regions, which is used to label connected components. Our simulations show that this takes roughly 50 microseconds on a 512 by 512 array, assuming a 100 nanosecond cycle. This Coterie Network, as it is called, has many other uses, including matrix arithmetic, FFT, convex hull computation, simulating a pyramid processor, etc.

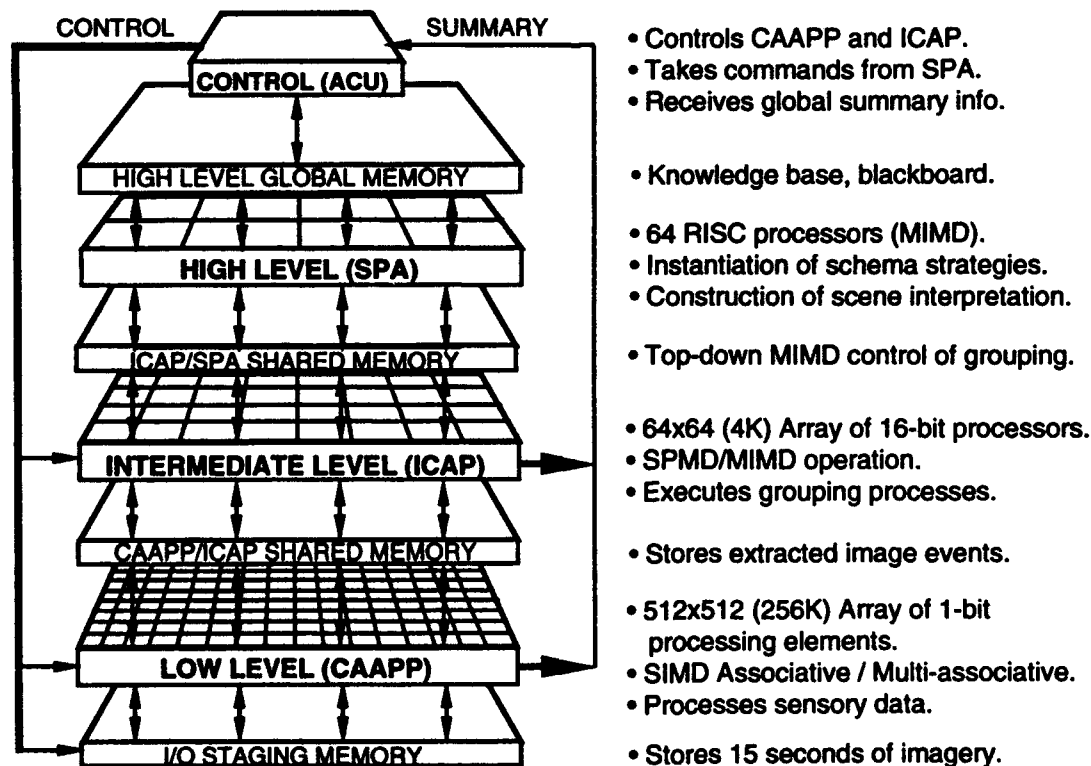


Figure 1. IUA Overview

3.2 Memory

The two obvious options for expanding processor memory are to add memory to the processor chip, or to use external memory. We anticipate chips with more than 64 (1024 is feasible) processors, and increased clock rate, which favors on-chip memory. However, we also saw the need for at least tens of thousands of bits per processor, which only off-chip memory could support. Our solution to this dilemma is to do both. Each processor contains an explicitly managed data cache on the chip. In our current implementation, this contains 320 bits, but the architecture provides for expansion to 1024 bits. Two pages of the cache also perform corner-turning (the transformation of bit-serial data into bit-parallel formats). The external memory is dual-ported with the intermediate-level processor, and is the primary data path between processing levels. The low-level, bit-serial data must therefore be "corner-turned" on its way to the backing store, so that the intermediate-level processor can work with it directly in bit-parallel formats. Each low-level processor has access to 32K bits of external memory. Backing store transfers take 16 instruction cycles per byte.

The corner-turning pages have eight-bit data paths, providing a factor of eight speed-up over the bit-serial data path for movement of data between locations in these pages. The wider data path is also useful for aligning mantissas in floating-point operations. The registers that control the Coterie Network switches are also attached to the 8-bit data path, allowing the entire network to be reconfigured with a single instruction.

3.3 Response and Activity Control

Traditional associative processor designs use a response flag to both control local activity and provide summary information to the central control. An opposite approach is to separate response from activity by providing a register for each. Both split and combined activity and response can be provided by allowing writes into either of the two registers or both simultaneously. This small change provided a 20 percent speed improvement in equality comparisons between a local value and a broadcast value. A similar change allows inequality tests to be performed in about 50% less time, by permitting the response register to be loaded with an operand at the same time that a result is stored in the activity register.

3.4 Response Count

Our analysis showed that counting processors with a specific bit set is frequently done in bursts. For example, summing a set of values in the array involves counting the ones in each bit position (each processor loads one bit at a time into its response register; a sum is developed by counting the bits in each position

and scaling the counts appropriately before summing them). Another example is creating a histogram of a set of data which could require 256 counts for an 8-bit field (each processor compares its value to a broadcast value, the bucket number, and if the values match the processor sets, its response bit, the table of counts, then corresponds to all of the buckets of the histogram).

For real-time applications, a count must be developed very quickly. One technique proposed by Foster [Foster, 1971], uses a pyramid of adders. Within the processor chip, Foster's scheme is used to produce a response count at the end of each instruction cycle. A special instruction latches the count into a shift register so that it can output serially. The processors are able to overlap computation with output of the current count.

A custom VLSI chip was designed with 64 serial inputs, one serial output, and six parallel outputs. One cycle after the low order bits of a set of partial counts are input, the low order bit of the result appears at the serial output. The high order bits of the result appear in the parallel outputs. The process can be repeated to sum 64 inputs of any bit length with the low order portion of the result being shifted out serially and the high order six bits available in parallel. Two levels of the chips are cascaded to form a count for the entire array. Only 1.6 microseconds are required to count the response registers in an array of 262,144 processors, using 65 copies of a single custom chip. The same chip also provides the Boolean summaries some/none, some/all, and exactly one responder. It can be thought of as a general purpose 64-input reduction unit.

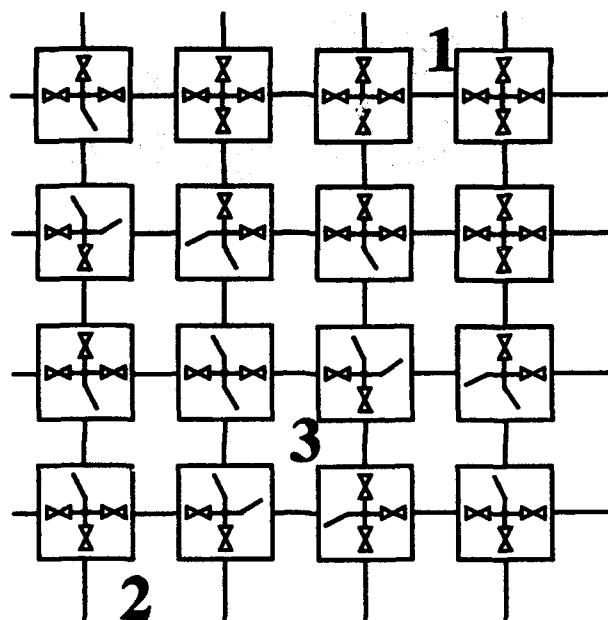


Figure 2. Coterie Network

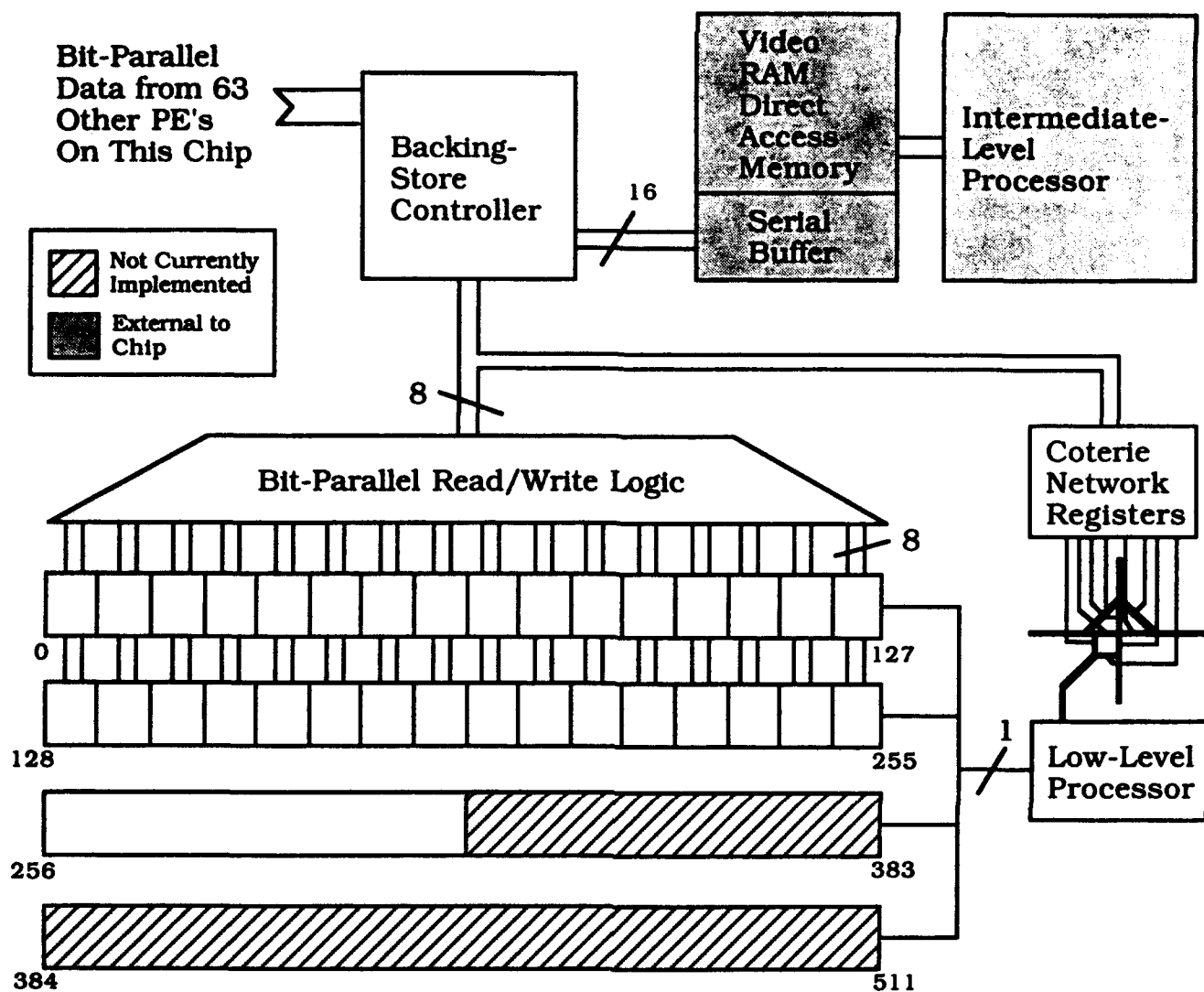


Figure 3. Memory Architecture

3.5 Input/Output

Our original I/O scheme was similar to that used in the MPP [Batcher, 1980], with data shifted in from an array edge. With parallel mesh communication, the time required to fill a 512 by 512 array with an eight-bit image, assuming a 100 nanosecond clock is 410 microseconds. Initially we felt that this would be fast enough, but later realized that even a pause of this length could interfere with real-time deadlines. Also, in a multisensory task, the amount of I/O is much more significant.

Another consideration is that vision systems occasionally fall behind and need to subsample the incoming stream of frames in order to catch up. It is also desirable to be able to retrieve previous frames. We were thus faced with

redesigning the I/O after beginning to design the new chip, which greatly limited our options.

Our solution for the prototype is to associate an additional Video RAM (VRAM) with each processor chip, connected to the South edge of the chip's mesh network (Figure 4). A special instruction tri-states the North edge of the chip during I/O. Then the data in the serial buffer of the VRAM is shifted in from the South, and stored by the processors, using the corner-turning circuitry. To output to the VRAM, the Coterie Network is used to send data from the North edge of the chip to the South, where it is streamed into the VRAM's serial port. Transfers to and from this staging memory take only 8 microseconds for an array-sized 8-bit image. The memory is large enough to hold sixteen seconds of imagery, providing a reasonable time window.

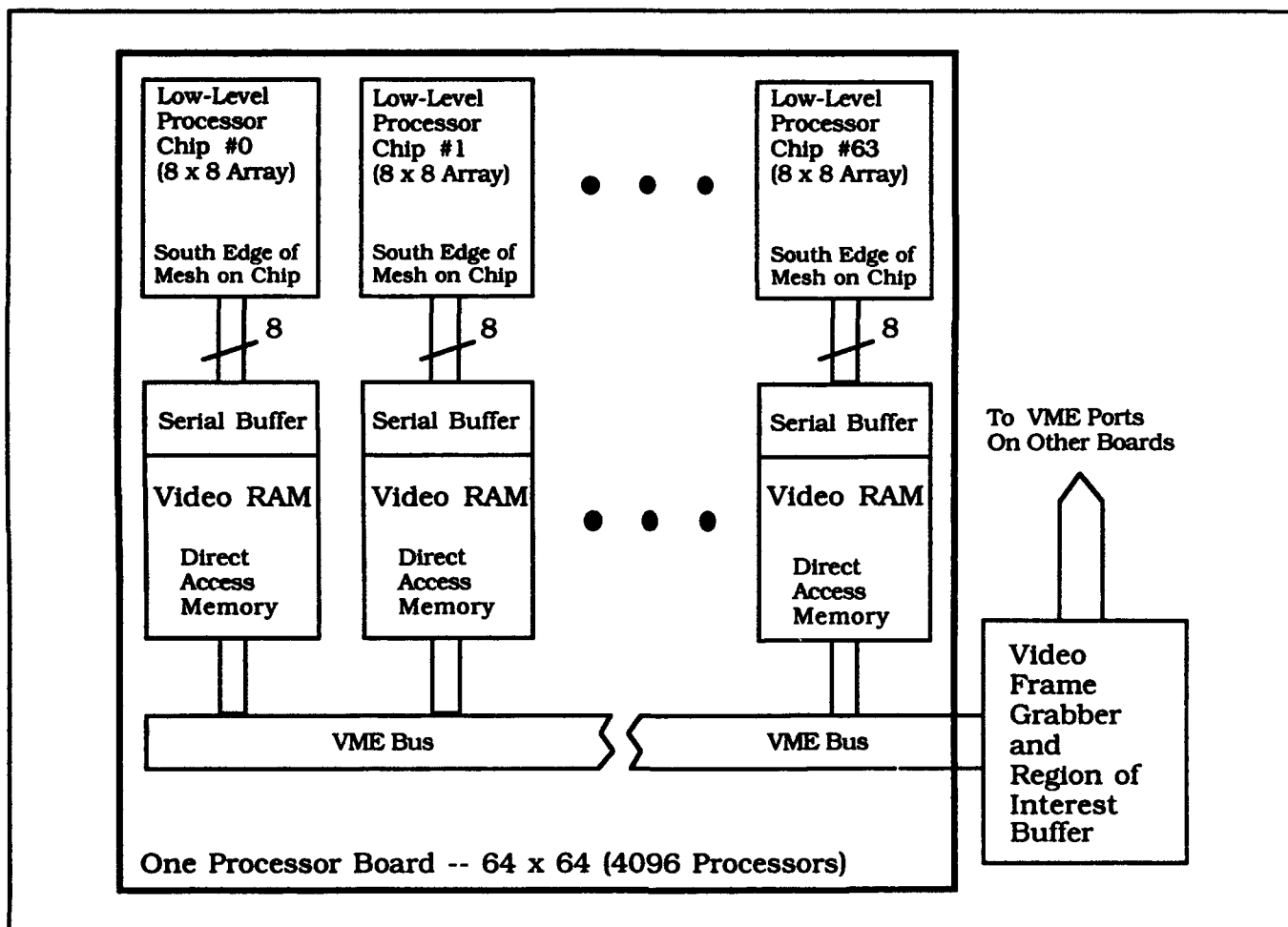


Figure 4. I/O Architecture.

The direct access portion of the VRAM is connected to a VME port, and appears as a block of memory in the VME address space. In the full-scale system, each processor card could have its own VME port so that parallel I/O to the staging memory can take place. In our prototype, the VME port can be connected to a frame grabber that moves data in and out through a region-of-interest buffer.

3.6 Changes to the CAAPP Specification for the Second Generation

Based on the results of our algorithm development and analysis efforts, we have recommended several changes to the CAAPP level of the IUA. These involve I/O, backing store access, and the number of processors per ICAP. We have also identified other areas that could be improved, but concluded that the required changes would not be supported by the technology available at the time of the construction of the second generation machine.

The input mechanism for the prototype requires an awkward series of steps in which the data is shifted in over the mesh network with bits of pixels distributed over multiple processors and then gathered to their destinations via the corner-turning hardware. Output involves scattering the bits with the corner-turning hardware and then routing them back to the input edge with the Coterie network. While this results in acceptable performance for I/O, it makes programming more difficult and it means that the HCSM can be used only for I/O and not as additional general purpose memory.

Thus, in the second generation CAAPP, we have specified that HCSM will appear as a second bank of backing store memory. The only physical difference between HCSM and CISM will be that the host (or I/O subsystem) will have access to the direct-access side of the HCSM VRAM while the ICAP has access to the direct-access side of the CISM VRAM. From the point of view of the CAAPP chip, the only difference will be in the addresses of the two banks. Thus, I/O becomes a memory-mapped DMA operation and the programmer simply accesses I/O data as memory locations. In addition, because backing-store transfers take place automatically under the control of a finite state machine, the array control unit only needs to issue one instruction to cause data in the HCSM to be loaded into the chip. Also, if an application requires more memory, it can use the HCSM to store intermediate results that are not required by the ICAP.

Because the prototype uses a 16-bit processor for the ICAP level, the backing store transfers between the CAAPP and CISM corner turn the data between bit-serial and bit-parallel bytes. The limited memory makes it necessary to pack bytes from two CAAPP processors into one ICAP word. With the decision to switch to the 32-bit TMS320C30 processor in the second generation ICAP and to quadruple the number of CAAPP processors per ICAP, this scheme encounters scaling problems. In particular, packing bytes from four CAAPP processors into

one ICAP word requires a costly reformatting of the data in order for the ICAP to efficiently access bytes in order. The reformatting is compounded by the use of 16-bit and 32-bit quantities in the CAAPP, whose bytes are then distributed among different ICAP words and packed with bytes from other CAAPP processors. Thus, a new backing-store transfer scheme has been specified, in which a 32-bit transfer register is added to the CAAPP. Data from the CAAPP's on-chip memory is transferred into the register in 8-bit blocks, and a backing-store transfer operation directs the finite state machine to reformat the data (packed or unpacked) as it is shifted out to the serial side of the CISM or HCSM VRAM. The reverse of this process is used for transfers in from CISM or HCSM.

As mentioned above, we also decided to quadruple the number of CAAPP processors per ICAP processor. The change goes hand-in-hand with the move to the TMS320C30 because the more powerful ICAP processor is able to handle larger numbers of extracted tokens. Also, increasing the number of processors on the CAAPP chip allows us to build systems with more PE's, thus reducing the requirement for high virtualization factors when processing large images.

Other changes that were deemed desirable but not appropriate for the current technology were wider data and communication paths and support for floating point. Adding any of these would significantly change the pitch of the processors on the chip, requiring a complete redesign and probably preventing the quadrupling of the number of processors (which will have a greater effect on performance). Increasing the data path width would necessitate a proportional increase in the on-chip memory, since the processors would go through their data much faster. Even in the current design, there is not enough on-chip memory, and so it makes no sense to increase the need for memory when there is not enough room for expansion. Wider communication paths would speed up routing and local communication, but would also require a substantial increase in pins which are even more limited than chip area. Thus, it was decided to postpone significant changes to the processing element architecture until the third generation.

4. Design of the ICAP Interconnection Network

The ICAP operates in two distinct modes of control. When working with the SPA, the ICAP operates in MIMD mode, but when interacting with the CAAPP, it is most efficient to keep the ICAP processors roughly synchronized. The remainder of this section will focus on the latter mode, which is called SPMD (Single Program, Multiple Data), and has a programming model that is similar to SIMD, except that branches can be performed simultaneously rather than sequentially. During SPMD operation, the ACU manages the stages of processing through barrier synchronization points. Communication between ICAP processors also

occurs synchronously via a reconfigurable network that is managed by the ACU. A typical scenario involves the ICAP processors communicating via one connection pattern, then synchronizing at a barrier and waiting for the ACU to reconfigure the network before releasing them.

The ICAP connection network is used to set up a connection pattern between the N output ports of the processors and the N input ports of these same processors. The connection network can be programmed on-line, to make a direct link from the output port of any processor to the input port of one or more processors. We have built a custom VLSI chip, called the PARallel COmmunication Switch (PARCOS), which is capable of both point-to-point and broadcast communication, allowing the connection network to realize any of N^N mappings of its input ports onto its output ports. All of the processors can send and receive data on their links at the same time. These links can be changed by the ACU at any time.

The 64-input, 64-output connection network for the IUA prototype uses 2 stages of 32 x 32 PARCOS chips. The PARCOS chips are connected to make a 64 x 64 crossbar switch with broadcast capability as shown in Figure 5. A detailed discussion of the network can be found in [Rana, 1988]. While these chips have been constructed and tested, they have not yet been installed in the IUA prototype because neither the Hughes nor the UMass budgets included the cost for fabricating the circuit-boards to hold them. UMass assumed Hughes would build the boards because it was up to them to decide on the physical construction of the system, including circuit-board form factors (which are necessary to determine costs). Hughes assumed that UMass had budgeted construction of the board because UMass was designing the custom VLSI chips. Fortunately, the system was designed to also function without the communication network (communication via the host instead), so this merely affects communication performance rather than functionality. We still hope to eventually construct the boards and install them, using other funds, once the prototype is physically installed at UMass.

4.1 The Parallel Communication Switch

The PARCOS chip consists of a communication matrix of 32 bit-serial inputs and 32 bit-serial outputs, a control memory, a set of registers and associated read/write circuitry. The PARCOS chip organization is shown in Figure 6. Multiple PARCOS chips can be used to build larger connection networks, such as the 64 x 64 network in the IUA prototype.

The communication matrix of PARCOS consists of 32 tree-structured multiplexers, each of which is a 1 of 32 multiplexer. All 32 input lines are connected in parallel to each of the 32 multiplexers. With this architecture, multiple outputs can be connected to the same input, providing broadcast mode

capability. For any multiplexer, path selection at any level of the tree is done with a single bit of a control word. Thus, 5 control bits are required to select 1 of 32 inputs for each multiplexer, or $32 \times 5 = 160$ bits for configuring the entire communication matrix.

The PARCOS control memory consists of 32 control words, where each control word contains the 32 bytes of 5 bits required for one configuration. The on-chip control memory is therefore constructed so that PARCOS can hold up to 32 of the most frequently used connection patterns for larger networks built out of this chip. The control memory is called the Connection Pattern Cache (CPC), because it is analogous to storing the most frequently used pages in a memory system cache.

The connectivity information for the communication matrix is stored serially into the control words. To write connectivity information in a control word of the CPC, first a row number is set in the Row Select Register (RSR). RSR is mapped into the chip's memory space, allowing the address bus in PARCOS to select the register, and the binary value on the data lines determines the row number. Next, 32 5-bit bytes are written into addresses 0 - 31. The memory location's address is the output port number and its contents determine which input port it is connected to. If only a subset of links need to be modified, this can be done by selectively writing only into locations corresponding to those links.

Reswitching the configuration of the communication matrix from one stored connection pattern in a control word to another requires a single write instruction, where the address of a new control word is placed in the RSR, and the control word's contents are loaded into the Control Pattern Register (CPR), activating a new connection pattern. Notice that the CPR allows a control word to be modified in the CPC without disturbing an existing configuration in the communication matrix. In many cases this feature allows the time to write a new connection pattern from the ACU into the CPC to be hidden while the processors are working on an algorithm.

PARCOS is implemented on a single 84 pin, 50,000 device, VLSI chip. It is a full custom design, built out of a 2 micron, P-Well, double metal, scaleable CMOS technology available through MOSIS. Each CPC memory bit is a 6 transistor static RAM cell. The worst case delay in broadcast mode from one input to 32 outputs is less than 50nS.

In addition to the centrally-controlled PARCOS design, we developed a series of enhancements to add capabilities for distributed-control synchronous routing, and then distributed-control asynchronous routing. Because the advent of the TMS320C30 with its two bidirectional channels, and built-in DMA controller obviated the need for these designs in the second-generation IUA after they were completed, we have not included them in this report. For the lengthy

discussion of the designs, and an analysis of their performance, the reader is referred to the thesis attached as an appendix to this report.

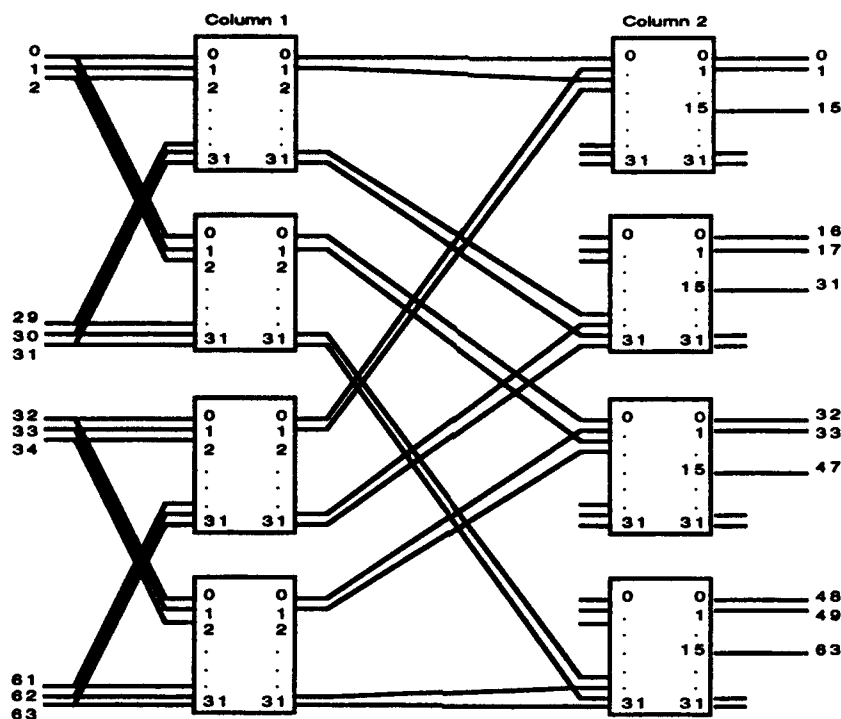


Figure 5. A 64 x 64 Network Built with PARCOS Chips.

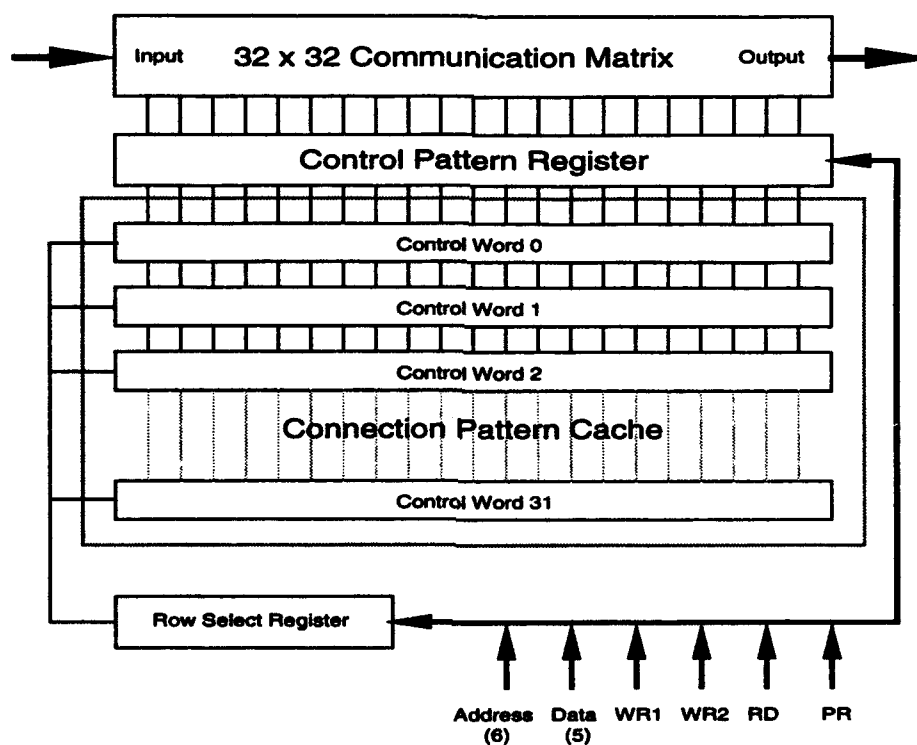


Figure 6. PARCOS Chip Organization

4.2 The Second Generation ICAP Communication Network

This section starts with an analysis of the ICAP router scheme for the second generation IUA (referred to here as IUA Generation II) that was originally proposed by Hughes. Thereafter an alternative to this scheme is explored in order to improve the performance of the ICAP interprocessor communication.

For comparative purposes, it is assumed that the ICAP messages are fixed in size at 64 bits. A fixed message size will simplify the router hardware and the routing algorithm. Since the TMS320C30 is a 32 bit processor, a multiple of 32 bits is the most feasible choice for the message size. A message size of 32 bits may be too small because several bits are required for overhead, such as source and destination addresses and control bits. A message size significantly larger than 64 bits will be inefficient when the ICAPs have short messages to transmit, while larger messages take longer to send.

First the design proposed by Hughes is presented as background. There are 64 ICAP processors in the system that are arranged in groups of four. Each group is called a Super Node (SNode) and is organized in a manner such that the four processors in it share a common memory (SMem) through their expansion bus (using a standard interlock scheme). Each SNode is treated as one unit in the ICAP router. There are 2 serial channels on each processor, allowing a total of 8 serial channels for each SNode. By using a binary hypercube topology, one can construct a 256 SNode system. The 16 SNodes in the IUA Generation II are connected in a dual 4-dimensional binary hypercube topology. The serial channels in the processors allow a handshaking mode such that a pair of channels on two processors can be directly connected. For processor to processor communication, the SMem serves as an intermediate storage space in a store-and-forward protocol. It is possible to set up a DMA mode such that either a word is read from the SMem and put on a serial output channel (in the handshaking mode, a serial channel has to be preprogrammed as an input or as an output channel) or a message is read through a serial input channel and is stored in the SMem. It should be noted that there is only one DMA controller in every processor and it needs certain preprogramming to be set up for reading from or writing to SMem. As such, the DMA controller on every processor can be used for only one serial channel at a time.

The following points summarize the performance of this design:

- (1) In a 4-dimensional binary hypercube configuration, the maximum SNode to SNode distance is 4.

- (2) Each serial channel has a maximum data rate of 1Mbyte/sec (the maximum clock frequency of the channels is 8MHz or 125nS for a 32MHz input clock frequency to the processors).
- (3) It will take $64 \times 125\text{nS} = 8$ microseconds to send or receive a 64 bit message through the processor serial channels. Taking into account the overhead for DMA transfer to and from the SMem at the two ends, it will take about 10 microseconds per message per hop.
- (4) Therefore, the best processor to processor communication time is about 10 microseconds.
- (5) The lower bound on the communication time between any two processors without any conflicts corresponds to the diameter of the hypercube, or about 40 microseconds.
- (6) The worst case time for many to many communication will take at least $4 \log N$ steps or $16 \times 10 = 160$ microseconds. Another example of this time is in many-to-one communication where 63 processors send a message to the 64th processor. On every SNode, only 4 serial channels are configured as input channels (the other 4 channels are configured as output channels). It will thus take 16 steps on 4 channels to receive 63 messages.

Before we present our alternative design, we begin by again noting that each serial channel in a processor is capable of only a 1 Mbyte/sec data rate (when configured in handshaking mode, a channel can send data in only one direction at a time. The other direction is used for acknowledgment). The cumulative capacity of all of the serial channels on one SNode is thus 4 Mbyte/sec. Remember that there is only one DMA controller on every processor. When either a source or a destination is in the internal memory of the processor and the other destination or source is on the expansion bus (SMem in this case), processor DMA is capable of up to a 22 Mbyte/sec data rate. Thus, there is a clear bottleneck in the SNode, which is its serial ports. Looking at it in another way, the processor DMA capability is underutilized.

The first change we recommend is to put all 8 processors on a motherboard on a common bus. Even in this case, the DMA bus has much higher capacity than the serial ports.

Second we recommend a change to the interconnection topology of the network on the backplane. In the dual binary hypercube topology, there are 12 serial channel connections on the backplane for every motherboard. By having only two additional serial channels (a total of 14), one can have a fully connected

topology between the motherboards. The benefit of this topology will be apparent in the following paragraphs.

Now, every motherboard has only one SNode. Each SNode has a total of 16 serial channels. Seven of these channels are configured as output channels and are connected to seven SNodes on other motherboards. Seven additional serial channels are configured as input channels and connected to other SNodes. We recommend connecting the remaining 2 channels on the 8 SNodes in a daisy-chain to an outside connector. This connector can be used later for diagnostic purposes or for direct I/O with the processors.

The following points summarize the performance of the second design:

- (1) The maximum SNode to SNode distance is 1.
- (2) The best communication time between any two processors without conflicts is 10 microseconds.
- (3) The worst case time is in many to one communication. At most there can be 8 outgoing messages from a motherboard queued on a single serial channel, which will take 80 microseconds.

Notice that between the two designs, the worst case communication time has improved by a factor of 2. Also notice that the average case communication time has improved by a factor of 2 to 4. In synchronous routing, if there are no conflicts, the message that has to travel the maximum distance determines the routing time. In the hypercube, the diameter will determine the performance.

A third design has been developed that requires custom VLSI hardware. The basic concept of the design is to bypass the TMS320C30 communication channels and to provide a router chip that operates in DMA mode, transferring data to and from the SMem of different processors. We estimate that such a design could achieve almost 900 MBytes/sec aggregate transfer capacity with a latency of 16 instruction cycles, which is reasonably balanced against the 13 cycles required to initiate a transfer. Because the IUA Generation II schedule does not provide enough time to construct a custom VLSI router chip, a plan has been formulated to allow such a chip to be retrofitted to our proposed design. For the lengthy discussion of the third design and more detailed presentations of the other designs, the reader is referred to the thesis attached as an appendix to this report.

5. The IUA Simulator and Software Environment

The simulator for the Image Understanding Architecture provides a way of testing the design of, and developing the software for, the IUA. The simulator originally was developed on a VAX running VMS, then was ported to a TI Explorer and to a Sun running SunOS and Sunviews. It now runs under X and is currently installed on Sun and DEC workstations and on a Sequent Symmetry multiprocessor. Several versions of the simulator exist on each machine and differ in the size of the IUA being simulated. The larger the IUA, the slower and larger is the simulation. The simulator supports IUA configurations with various numbers of Mother Boards as shown in the following table.

Mother Boards	1	4	16	64
CAAPP PEs	4096	16384	65536	262144
ICAP Processors	≤ 64	≤ 256	≤ 1024	≤ 4096

Even with the smallest complement of Mother Boards, a complete IUA is usually not simulated due to limitations on the real memory available on the host computer and the desire to avoid page faults when running simulations. (Even the smallest configuration of the IUA contains 42 MB of RAM.) The amounts of CAAPP-ICAP Shared Memory (CISM), ICAP-SPA Shared Memory (ISSM), and ICAP Data Memory (IDM) are thus limited to that needed by the problem being run on the simulator. For the same reasons, the ICAP Program Memory (IPM) is considered to be read-only so that it need not be duplicated for all the processors.

The simulator has been constructed in a modular fashion so that the various parts may be replaced easily for different needs such as allowing substitution of a 64 processor ICAP simulator module for a 16 processor ICAP simulator module when the primary simulation is at the ICAP level instead of at the CAAPP level. The user's view of the simulator is presented by the "user console," which contains separate windows such as a control panel, display window, and programming terminal.

The user console is a window which is roughly 600 rows by 800 columns in size, leaving sufficient room to view other windows (such as a command window or editor) on the screen. This display is split into a left and right side. The left side is the Control Panel and the right side is the Display Window.

The Control Panel contains displays of the 1-bit CAAPP registers arranged as blocks of 64 x 64 pixels with one pixel per processor. For the smallest simulator this is the complete set of processors. For larger versions of the simulator, this is a sub-window into the n by n array of CAAPP processors. Associated with each

register display is a button. Clicking this button with the mouse causes the display to be shown enlarged in the Display Window. Other buttons in the Control Panel cause other displays to appear in the Display Window or bring up pop-up windows for special operations such as loading and saving files. Having all of the registers shown at the same time allows the programmer to see the state of the CAAPP processors arranged in correspondence to images stored in the array.

The Display Window consists of a foreground and background display. The background display is always the Programming Terminal. The foreground display may or may not be present and shows the display selected by clicking a button in the Control Panel. The foreground display leaves the bottom sixth of the Programming Terminal always visible. The Programming Terminal allows entry of commands and input to a running program. Output from a program can also be shown on the terminal.

The foreground displays include processing element (PE) registers, PE memory, a gray level display, coterie network display, ICAP display, and PE instruction display. All of the displays can be selected and manipulated by the user program running in the simulated ACU.

The PE register display presents a binary image indicating the status of one selected PE register from all PEs. The user can zoom in or out from a 2 by 2 processor display up to a display showing all the processors being simulated, even for a 512 x 512 simulation. Scroll bars on the sides indicate what portion of the complete array is being shown. Individual PE registers may be set or cleared by clicking with the mouse inside of the display window. The Control Panel simultaneously shows the current row-column processor location selected by the mouse.

The PE memory display shows one location in every PE Memory as a binary image. This display has the same functionality as the PE Register display.

The gray display shows a contiguous range of bits in PE Memory as a gray scale image. The user can zoom in or out from a 2 by 2 processor display up to a display showing all the processors being simulated. (Scroll bars on the sides indicate what portion of the complete array is being shown.) The user may select from 1 to 32 bits in the range to be displayed, although the actual screen representation may be limited by the available graphics hardware. The gray display may be changed to an inverse gray or false color mode.

The user may select a 3 by 3 pixel array with the mouse to be shown as numeric values in a pop-up window in either signed integer, unsigned integer, or IEEE floating point format. The location in PE memory or CISM memory that is

sampled may be different from the location shown in the Gray Display, allowing an image to be used to guide exploration of other values that may not be visually informative when shown as an image. The Gray Display can be overlaid with red, green, and blue pixel maps. The overlay can be any of the PE registers or locations in PE memory.

The Coterie Display shows a graphic representation of the state of the switches and the electrical charge in the network. Currently, the Coterie Display is limited to 32 by 32 PEs. A particular PE can be dragged to the center of the display with the mouse. Scroll bars on the side indicate what portion of the complete network is being displayed. Using function keys, individual switches can be opened or closed in the Coterie Network.

The ICAP display shows the complete set of registers for one ICAP processor. Also shown are all of the registers on the same Daughter Board (except for the CAAPP PE registers). When this display is selected, a pop-up window appears which may be used to select a particular Daughter Board, and a range of locations in ICAP Data Memory (IDM), to be displayed. A range of locations in ICAP Program Memory (IPM), surrounding the current value of the program counter, is also shown. Up to four breakpoints may be set for the program running in the ICAP processors. Both the IDM range and the breakpoints may be selected using symbolic expressions.

The PE Instruction Display is a scrollable display of the last 2048 instructions sent to the CAAPP by the ACU. The instructions are shown both in hex and in symbolic form.

The Daughter Board Simulator simulates all of the PEs, CISM, ISSM, and glue logic. While the IUA is made up of multiple Daughter Boards, the Daughter Board simulator does not simulate them one at a time. Instead, for efficiency, the PEs are simulated as a vector of processors. At those places where the geometry of the IUA is apparent, the simulation applies a board-by-board approach. For example, the instruction "zero the Z registers" is done for all PEs in a tight loop while the backing-store operations are done board-by-board.

The Daughter Board simulator receives instructions in the same form that the real Daughter Boards receive instructions from the ACU. The instructions are sent on a simulated bus as 32 bit signals and are then decoded. This provides two benefits over a more tightly coupled scheme. First, the instruction stream on the bus is easily captured and can be used in exercising circuit boards under test. Second, using a bus allowed quick construction of versions of the simulator that could utilize a parallel processor. In fact, we use the same Daughter Board code for both the uniprocessor and multiprocessor implementations. A compile time switch is used to select the data partitioning parallel code which resides in

only one subroutine. The parallel code is coupled with synchronizing check points that are no-ops in the non-parallel versions.

The computation of execution time is also simplified through the use of the simulated bus. Because the majority of instructions take a single clock cycle, an accumulator is used to count the instructions sent over the bus. An ACU overhead cost is also added as appropriate. (Micro-controller routines have lower per-instruction overhead.) For those operations, such as backing-store-read, where the result is not available immediately, the code that simulates the individual instruction adds the worst case time to the accumulator. (In the multiprocessor version of the simulator, only one processor does the addition.) For the real machine, these variable length processes will be handled either by using feedback or by fixed time idle loops in the ACU for the worst case. For the simulator, we took the second approach since we did not want to simulate the backing-store finite state machine at the level needed to provide the correct timings using the feedback method.

The Coterie Network provided special problems in the simulation because it is really an analog circuit using electrical charge propagation. As the cost of an analog simulation of the network is prohibitive, we simulated the charge propagation digitally; permitting us to execute one cycle of 100 ns in approximately 18 milliseconds for a 512 by 512 PE simulator using 9 processors on the Sequent multiprocessor. For the DARPA Integrated IU benchmark, this would have otherwise required approximately 24 days of wall-clock time to run only one of the five test cases. An analysis of the problem showed that the configuration of the network was being changed infrequently with respect to the number of network operations performed. We thus modified the simulator to record a list of connected PEs whenever the network is reconfigured, and this information is used to accelerate the simulation of subsequent network operations using that configuration. The result is that the complete set of 5 IU benchmark test cases can be run in just two and one half days on the Sequent.

The Programming Terminal is an interactive interpreter that allows entry of commands and programs to directly manipulate the processing arrays. The IUA Simulator has been designed so that this module may be easily replaced by other modules. (Currently, the only module available is for interpreting the FORTH language.) A module for Lisp could be provided as well. The interface consists of input and output streams from the simulator, procedure calls for issuing instructions to the CAAPP bus, and other procedure calls for changing the displays.

We felt that it was very important to provide an interactive environment so that the edit, compile, test loop would be very fast. The programmer can rapidly

prototype code interactively and then reimplement the tested algorithm as an "ACU Macro" if desired.

FORTH is a threaded language. A few simple constructs are combined into ever more powerful constructs. Each construct is called a word. FORTH was selected because its interpreter is small and fast. We also had access to the source code for a FORTH implementation. New features such as floating point operations were built, in addition to interfaces to the IUA Simulator. FORTH provides a quick way of changing a program and re-trying it, or of just entering instructions. We provided a FORTH-based assembler for the CAAPP PE instructions so that the user is able to enter an instruction such as

```
A := B 'AND Y !!
```

and have it assembled and sent on the CAAPP bus. Each of the symbols (A, :=, B, etc.) are FORTH words that place information on the FORTH stack. This information is processed by the FORTH word !! to produce the CAAPP bus instruction. The special words provided to interface with the simulator allow FORTH to control all aspects of the simulation and act as the ACU.

The low level processing portion of the DARPA Integrated IU benchmark was written mostly in FORTH with some of the simple ACU Macros such as ADD-FIELDS written in C. The major problem encountered with the use of FORTH was its flat name scoping, which prevented the FORTH code from being completely modular due to name conflicts (no vocabulary facility is available).

The following example is the definition of a FORTH word that adds two integer fields of the same length in PE memory.

```
( 1 ) : ADD-FIELDS ( length f1 f2 -- )      ( Add field f2 to field f1 )
( 2 )   Z := ZERO !!
( 3 )   2 ROLL 1 - 0 DO
( 4 )       1 PICK I + >R X := R > A!
( 5 )       0 PICK I + DUP >R := X '+' R > A!
( 6 )   LOOP ;
```

Line 1 defines the word ADD-FIELDS. This word takes three arguments off of the FORTH stack. The top stack value is the PE Memory address of the second operand. The next stack value is the first operand/result field PE Memory address. The third value is the length of the fields in bits. This argument protocol is documented with the comment in line 1. Line 2 clears the Z (carry) register in all PEs. Line 3 is a FORTH indexed loop. The 2 ROLL picks up the length value from the FORTH stack. If this value were 8, then line 3 would be equivalent to 7 0 DO which would loop over the values 0, 1, 2, 3, 4, 5, 6, 7. The end of the loop is

specified in line 6 which also ends the FORTH word. Line 4 generates the PE instruction "load the X register of the active PEs with the value from memory location $f2 + I$ " where I is the loop index. The FORTH return stack is used as a temporary holding place for the value $f2 + I$. Line 5 generates the PE instruction

$M[f1+I] := X + M[f1+I] A!$

which causes the X register, the Z register, and memory location $f1 + I$ to be added in all active PEs. The result is placed in memory location $f1 + I$ and the carry goes to the Z register. For the FORTH statement

2 10 20 ADD-FIELDS

the following PE instructions would be issued to add a pair of 2-bit values at locations 10..11 and 20..21, with the result being stored back in locations 10..11:

$Z := ZERO !!$

$X := M[20] A!$

$M[10] := X + M[] A!$

$X := M[21] A!$

$M[11] := X + M[] A!$

The ICAP simulator is structured to simulate one instruction from the first ICAP processor, one instruction from the next and so forth. Because the ICAP processors run in MIMD mode, these instructions will probably be different. As we did not want to pay the price of a gate level simulator, we chose to implement a functional simulator for the TMS320C25.

Because the simulation is on an instruction level, code written to use timing loops is not valid as the simulator will not maintain synchronization between the various ICAP processors. A disadvantage of this approach is that the timings are not exact. Our timing model uses the average time for an instruction with data in on-chip memory and instructions in off-chip memory. Our experience shows that approximately 90 percent of the data references are to the on-chip memory. With the TMS320C25 there is a large benefit to using a small amount of contiguous data memory, which is due not only to the on-chip data memory, but also to the addressing modes supported.

The full IUA will have 4096 ICAP processors. Each one will have 128k bytes of data memory and 128k bytes of program memory. This is a gigabyte of memory and is far more than the real memory available on any of the machines we have used to run the simulator. In order to prevent page faults which would have drastically increased the elapsed time for any simulation, we reduced the number of ICAP processors and the amount of IDM and IPM available in the

simulator. The IUA simulator can be easily re-configured as to the number of ICAPs being simulated (independently of the size of the CAAPP array) and the size of IDM. The IPM is shared as read-only memory among all the ICAP processors. Even though the ICAP processors run independently, they all have the same programs loaded in IPM; which restricts the programmer to writing code that is not self-modifying.

Because there is a single system clock, the CAAPP and ICAP instruction streams are in approximate synchronization with roughly a two to one execute rate. Therefore, the IUA simulator executes two CAAPP instructions and then one instruction for each ICAP processor. Since an ICAP instruction may take more than one cycle, a particular ICAP processor is held up if its clock shows more time than the CAAPP has used.

The ICAP code is loaded into IPM by the ACU. The ACU can write the same program in every ICAP IPM directly using the CAAPP bus. The ACU can also write directly into any off-chip IDM location. Because the bus is a one-to-many bus, all ICAPs receive the same data and programs. However it is also possible for the ACU to write a loader program into IPM that causes IPM to be loaded from ISSM by each individual ICAP processor. Since each ICAP processor sees a separate part of ISSM, each can be loaded with a different program in the actual machine. This mode of operating is not supported by the IUA simulator as there is only one IPM shared by every ICAP processor in the simulation, to reduce the space requirements. In practice, this restriction did not present any problems because MIMD operation can be simulated by loading multiple programs into the single IPM and having the ICAP processors branch to the appropriate code.

Code for the ICAP processors can be written in either C or Assembler. The IUA simulator provides a loader for either special absolute code or for the TI COFF loader text format.

While the full-scale IUA has one ACU and 64 SPA processors, the prototype hardware has one ACU and just one SPA. In the simulator, the ACU and SPA are the same computer, and the simulator follows this simpler model regardless of how many PEs or ICAP processors are simulated. The ACU/SPA is the interactive module linked into the Programmers Terminal, which is currently the FORTH interpreter.

A separate module called the ACU Macros is also part of the ACU. The ACU Macros are procedures written in C for standard operations on the CAAPP such as ADD-FIELDS or FIND-GREATEST. On the real IUA, these macros will be stored in the micro-code memory of the Micro-controller that feeds the bus to the Daughter Boards. The ACU will request that a macro be executed with specified parameters, the micro-controller will execute the macro, substituting the

parameters at the clock rate of the CAAPP, while the ACU program sets up the next request. The approach taken in the simulator has been to identify which sequences of code should be in the micro-code memory and what capabilities the micro-controller must have to efficiently execute those sequences. The ACU calls the macros via a special simulator interface procedure.

Also available to the user are "User macros". These macros allow the user to provide procedures written and compiled in C, which can either be special code for an application or candidates for an ACU Macro. Thus, the interactive Programmer's Terminal can still be used while taking advantage of the benefits of C. The cost to the user is that the simulator must be relinked whenever a change is made in the C code.

5.1 CAAPP-specific Parallel Extension to C

In addition to FORTH, C may be used to program the ACU and thus the CAAPP PEs. We use a special pre-processor to convert CAAPP PE instructions written in a high-level like syntax into procedure calls that communicate with the PEs. The programmer must still be cognizant of the structure of the CAAPP PEs. As an example, a C program for doing an ADD-FIELDS in the PEs looks like

```
void add_fields(length,f1,f2)
int length;          /* The length of the field */
int f1;              /* Address of source and result field */
int f2;              /* Address of other field */
{
    int i;
    !! Z := ZERO !!
    for (i = 0; i < length; i++)
    {
        !! X := M[f2 + i] A!
        !! M[f1 + i] := X + M[i] A!
    }
}
```

The !! signals the beginning of a parallel instruction to the preprocessor and the other !! or A! terminates the instruction. The statement

```
!! X := M[f2 + i] A!
```

is converted to a call to a subroutine that sends a 32-bit instruction to the simulated CAAPP bus. The sense of this style of programming is that the programmer is writing machine-specific programs in C.

Using C is not difficult because the programmer rarely has to work at such a low level. A large library of routines for doing the mundane arithmetic operations is available. These routines work with variable size fields, and require only that the programmer determine the size of a result from the sizes of the arguments. For example, multiplying a four bit unsigned field with a seven bit unsigned

field produces a 11 bit result. Other library routines for standard vision applications also exist.

Because we use variable-size fields in CAAPP processing, changing field sizes initially required all of the PE memory to be remapped by the programmer, which is a very error prone process. We solved the problem by providing dynamic allocation of PE memory via allocation code in the ACU. When we have a true compiler for the CAAPP, a return to static mapping will be possible because the compiler will be able to optimize the mapping without error.

5.2 Apply

Apply is a special-purpose image processing language developed at CMU for use on the WARP systolic array. Apply is very good for expressing low level image operations that are applied in a window on the image. Examples include gaussian filtering, pixel averaging, median filters, etc. We have implemented Apply for the CAAPP. It produces output that is acceptable to the C preprocessor described above. The Apply compiler produces code that uses the CAAPP PE memory as cache memory and the CISM as the real memory. This is an approach that we believe will generally be used in the implementation of higher level languages.

There is a large body of Apply code (the Web library) available for the WARP, and having Apply for the CAAPP makes these routines available on the CAAPP. The following is an example of Apply code for performing a simple smoothing operation using a 3 x 3 window around each image pixel.

```
-- inimg - a 3 x 3 window centered around the
--      output pixel (outimg).
--      The border value is set to 128, which is a mid-range value.
--      -1,-1 | -1, 0 | -1, 1
--      -----
--      0,-1 | 0, 0 | 0, 1
--      -----
--      1,-1 | 1, 0 | 1, 1
```

```
procedure smooth(
  inimg: in array (-1..1, -1..1) of byte border 128,
  outimg: out byte) is
```

sum, i, j: integer; -- These local variables are local to each pixel in the image.

```
begin  sum := 0;          -- sum receives the total of the input image window.
  for i in -1..1 loop
    for j in -1..1 loop
      sum := sum + inimg(i,j);
    end loop;
  end loop;
  outimg := (sum + 4) / 9; -- add 4 to round result and divide by number of pixels in the input window.
end smooth;
```

The major limitation of Apply is that only spatially local operations fit the Apply paradigm. The DARPA Integrated IU benchmark, for example, relied more on global communication among processors for which Apply was inappropriate.

5.3 Libraries

We currently have an extensive library of arithmetic subroutines for the CAAPP, including byte, integer, and floating point arithmetic, and many of the standard transcendental functions. Thus, we have the basis for a compiler run-time library. We also have implemented many vision subroutines, including various convolutions, filters, edge-preserving smoothing, convex hull, expand and contract morphological operators, connected component labeling, boundary tracing, windowed Hough transform, etc. In addition, we have implementations of the DARPA Integrated IU Benchmark, the Abingdon Cross Benchmark, and an optical ray-tracing application (as an example of a non-vision application).

The C compiler for the ICAP implements the "standard" C of Kernighan and Ritchie. No special provisions were made for the IUA. To make the job of the ICAP programmer easier we have provided an ICAP monitor which services requests for CISM, ISSM, and PARCOS operations. With the monitor, an ICAP process can request ACU operations, respond to network operations, access CISM in a simplified manner, and send and receive information through ISSM. In addition to the compiler run-time library, we have a library that supports communication via the serial ports, and synchronization with the ACU. From the DARPA Benchmark, we also have a model-matching algorithm that runs on the ICAP.

6. Image Understanding Benchmark

While traditional supercomputing benchmarks may be useful in estimating the performance of an architecture on some types of image processing tasks, those benchmarks have little relevance to the majority of the processing that takes place in a vision system [Duff, 1986]. Nor has there been much effort to define a vision benchmark for supercomputers, since those machines in their traditional form have usually been viewed as inappropriate vehicles for knowledge-based vision research. However, now that parallel processors are becoming readily available, and because they are viewed as being better suited to vision processing, researchers in both machine vision and parallel architecture are taking an interest in performance issues with respect to vision. We begin by summarizing the work that has been done in the area of vision benchmarks to date, then we examine the DARPA IU Benchmark developed under this effort.

6.1 Review of Previous Vision Benchmark Efforts

One of the first parallel processor benchmarks to address vision-related processing was the Abingdon Cross benchmark, defined at the 1982 Multicomputer Workshop in Abingdon, England [Preston, 1986]. In that benchmark, an input image was specified that consisted of a dark background with a pair of brighter rectangular bars, equal in size, that cross at their midpoints and are centered in the image, and with Gaussian noise added to the entire image. The goal of the exercise was to determine and draw the medial axis of the cross formed by the two bars. The results obtained from solving the benchmark problem on various machines were presented at the 1984 Multicomputer Workshop in Tanque Verde, Arizona, and many of the participants (including members of the UMass IUA group) spent a fairly lengthy session discussing problems with the benchmark and designing a new benchmark that it was hoped would solve those problems.

It was the perception of the Tanque Verde group that the major drawback of the Abingdon Cross was its lack of breadth. The problem required a reasonably small repertoire of image processing operations to construct a solution. The second concern of the group was that the specification did not constrain the assumed information that could be used to solve the problem. In theory, a valid solution would have been to simply draw the medial lines since their true positions were known. Although this was never done, there was argument over whether it was acceptable for a solution to make use of the fact that the bars were oriented horizontally and vertically in the image. A final concern was that no method was prescribed for solving the problem, with the result that every solution was based on a different method. When a benchmark can be solved in different ways, the performance measurements become more difficult to compare because they include an element of programmer cleverness. Also, the use of a consistent method would permit some comparison of the basic operations that make up a complete solution.

The Tanque Verde group specified a new benchmark, called the Tanque Verde Suite, that consisted of a large collection of individual vision-related problems. Table 1 contains the list of problems that was developed. Each of the problems was to be further defined by a member of the group, who would also generate test data for their assigned problem. Unfortunately, only a few of the problems were ever developed, and none of them were widely tested on different architectures. Thus, while the simplicity of the Abingdon Cross may have been criticized, it was the respondent complexity of the Tanque Verde Suite that inhibited the latter's use.

Standard Utilities	High Level Tasks
3x3 Separable Convolution	Edge Finding
3x3 General Convolution	Line Finding
15x15 Separable Convolution	Corner Finding
15x15 General Convolution	Noise Removal
Affine Transform	Generalized Abingdon Cross
Discrete Fourier Transform	Segmentation
3x3 Median Filter	Line Parameter Extraction
256 Bin Histogram	Deblurring
Subtract Two Images	Classification
Arctangent(Image1/Image2)	Printed Circuit Inspection
Hough Transform	Stereo Image Matching
Euclidean Distance Transform	Camera Motion Estimation
	Shape Identification

Table 1: Tanque Verde Benchmark Suite

In 1986, a new benchmark was developed at the request of the Defense Advanced Research Projects Agency (DARPA). Like the Tanque Verde Suite, it was a collection of vision-related problems, but the set of problems that made up the new benchmark was much smaller and easier to implement. Table 2 lists the problems that comprised the first DARPA Image Understanding Benchmark. A workshop was held in Washington, D.C., in November of 1986 to present the results of testing the benchmark on several machines, with those results summarized in [Rosenfeld, 1987]. The consensus of the workshop participants was that the results cannot be compared directly for several reasons. First, as with the Abingdon Cross, no method was specified for solving any of the problems. Thus, in many cases, the timings were more indicative of the knowledge or cleverness of the programmer, than of a machine's true capabilities. Second, no input data was provided and the specifications allowed a wide range of possible inputs. Thus, some participants chose to test a worst-case input, while others chose "average" input values that varied considerably in difficulty.

11x11 Gaussian Convolution of a 512x512 8-bit Image
Detection of Zero Crossings in a Difference of Gaussians Image
Construct and Output Border Pixel List
Label Connected Components in a Binary Image
Hough Transform of a Binary Image
Convex Hull of 1000 Points in 2-D Real Space
Voronoi Diagram of 1000 Points in 2-D Real Space
Minimal Spanning Tree Across 1000 Points in 2-D Real Space
Visibility of Vertices for 1000 Triangles in 3-D Real Space
Minimum Cost Path Through a Weighted Graph of 1000 Nodes of Order 100
Find all Isomorphisms of a 100 Node Graph in a 1000 Node Graph

Table 2: Tasks from the First DARPA Image Understanding Benchmark

The workshop participants pointed out other shortcomings of the benchmark. Chief among these was that because it consisted of isolated tasks, the benchmark did not measure performance related to the interactions between the components of a vision system. For example, there might be a particularly fast solution to a problem on a given architecture if the input data is arranged in a special manner. However, this apparent advantage might be inconsequential if a vision system does not normally use the data in such an arrangement, and the cost of rearranging the data is high. Another shortcoming was that the problems had not been solved before they were distributed. Thus, there was no canonical solution on which the participants could rely for a definition of correctness, and there was even one problem for which it turned out there was no practical solution. The issue of having a ground truth, or known correct solution was considered very important, since it is difficult to compare the performance of two architectures when they produce different results. For example, is an architecture that performs a task in half the time of another really twice as powerful if the first machine's programmer used integer arithmetic while the second machine was programmed to use floating point, and they thus obtained significantly different results? Since problems in vision are often ill-defined, it is possible to argue for the correctness of many different solutions. In a benchmark, however, the goal is not to solve a vision problem but to test the performance of different machines doing comparable work.

The conclusion from the first DARPA benchmark exercise was that a new benchmark should be developed that addresses the shortcomings of the preceding benchmarks. Specifically, the new benchmark should test system performance on a task that approximates an integrated solution to a machine vision problem. A complete solution with test data sets should be constructed and distributed with the benchmark specification. And, every effort should be made to specify the benchmark in such a way as to minimize the opportunities for taking shortcuts in solving the problem. The task of constructing the new benchmark, to be called the Integrated Image Understanding Benchmark, was assigned to the vision research groups at the University of Massachusetts at Amherst, and the University of Maryland.

Following the 1986 meeting, a preliminary benchmark specification was drawn up and circulated among the DARPA image understanding community for comment. The benchmark specification was then revised, and a solution was programmed on a standard sequential machine. In creating the solution, several problems were discovered and the benchmark specification was modified to correct those problems. The programming of the solution was done by the group at the University of Massachusetts and the code was then sent to the group at the University of Maryland to verify its validity, portability, and quality. The group at Maryland also reviewed the solution to verify that it was general in

nature and neutral with respect to any underlying architectural assumptions. The Massachusetts group developed a set of five test cases, and a sample parallel solution for a commercial multiprocessor.

In March of 1988, the benchmark was released, and made available from Maryland via network access, or by sending a blank tape to the group in Massachusetts. The benchmark release consisted of the sequential and parallel solutions, the five test cases, and software for generating additional test data. The benchmark specification was presented at the DARPA Image Understanding Workshop, the International Supercomputing Conference, and the Computer Vision and Pattern Recognition conference [Weems, 1988]. Over 25 academic and industrial groups, listed in Table 3, obtained copies of the benchmark release. Nine of those groups developed either complete or partial versions of the solution for an architecture. A workshop was held in October of 1988, in Avon Old Farms, Connecticut, to present those results to members of the DARPA research community. As with the previous workshops, the participants spent a session developing a critique of the benchmark and making recommendations for the design of the next version.

International Parallel Machines	Hughes AI Center
Mercury Computer Systems	University of Wisconsin
Stellar Computer	George Washington University
Myrias Computer	University of Massachusetts*
Active Memory Technology	SAIC
Thinking Machines*	Eastman Kodak
Aspex Ltd.*	University College London
Texas Instruments	Encore Computer
IBM	MIT
Carnegie-Mellon University*	University of Rochester
Intel Scientific Computers*	University of Illinois*
Cray Research	University of Texas at Austin*
Sequent Computer Systems*	Alliant Computer*

Table 3: Distribution List for the Second DARPA Benchmark

* Indicates Results Presented at the Avon Workshop

The remainder of this section summarizes those results and recommendations, following a brief review of the benchmark task and the rationale behind its design.

6.2 Benchmark Task Overview

The overall task that is to be performed by this benchmark is the recognition of an approximately specified 2 1/2 dimensional "mobile" sculpture in a cluttered environment, given images from intensity and range sensors. The intention of

the benchmark designers is that neither of the input images, by itself, is sufficient to complete the task.

The sculpture to be recognized is a collection of two-dimensional rectangles of various sizes, brightnesses, two-dimensional orientations, and depths. Each rectangle is oriented normal to the Z axis (the viewing axis), with constant depth across its surface, and the images are constructed under orthographic projection. Thus, an individual rectangle has no intrinsic depth component, but depth is a factor in the spatial relationships between rectangles. Hence the notion that the sculpture is 2 1/2 dimensional.

The clutter in the scene consists of additional rectangles, with sizes, brightnesses, two-dimensional orientations, and depths that are similar to those of the sculpture. Rectangles may partially or completely occlude other rectangles. It is also possible for a rectangle to disappear when another rectangle of the same brightness or slightly greater depth is located directly behind it.

A set of models is provided that represent a collection of similar sculptures, and the recognition task involves identifying the model which best matches the object present in the scene. The models are only approximate representations of sculptures in that they allow for slight variations in component rectangle's sizes, orientations, depths, and the spatial relationships between them. A model is constructed as a tree structure where the links in the tree represent the invisible links in the sculpture. Each node of the tree contains depth, size, orientation, and intensity information for a single rectangle. The child links of a node in the tree describe the spatial relationships between that node and certain other nodes below it.

The scenario that the designers imagined in constructing the problem was a semi-rigid "mobile", with invisible links, viewed from above, with bits and pieces of other mobiles blowing through the scene. The state of the system is that previous processing has narrowed the range of potential matches to a few similar sculptures, and has oriented them to correspond with information extracted from a previous image. However, the objects in the scene have since moved, and a new set of images has been taken prior to completing the matching process. The system must make its final choice for a best match, and update the corresponding model with the positional information extracted from the latest images.

The intensity and depth sensors are precisely registered with each other and both have a resolution of 512 x 512 pixels. There is no averaging or aliasing in either of the sensors. A pixel in the intensity image is an 8-bit integer gray value. In the depth image a pixel is a 32-bit floating-point range value. The intensity image is noise free, while the depth image has added Gaussian noise.

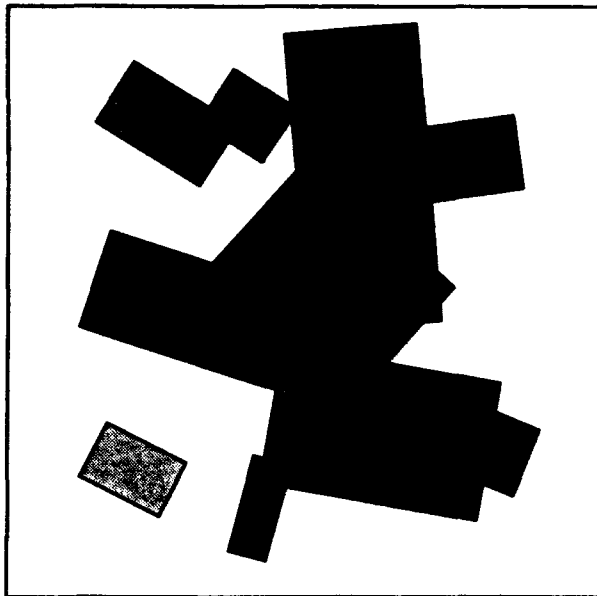


Figure 7: Intensity Image of Model Alone

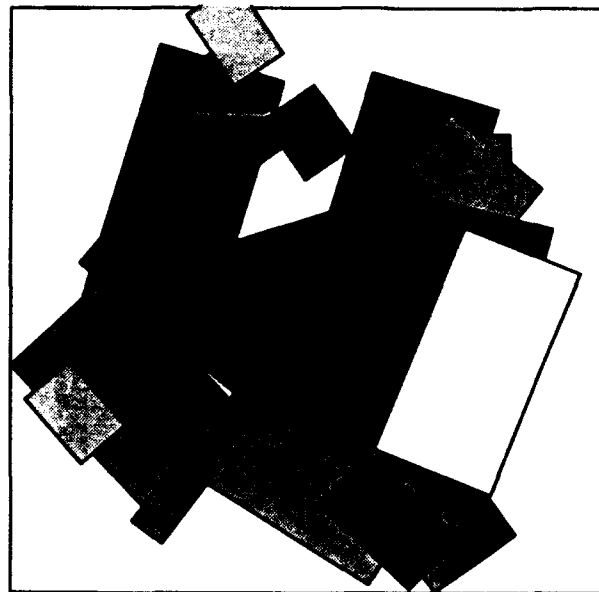


Figure 8: Image of Model with Clutter

A set of test images is created by first selecting one of the models in a set. The model is then rotated and translated as a whole, and its individual elements are then perturbed slightly. Next, a collection of spurious rectangles is created with properties that are similar to those in the chosen model. All of the rectangles (both model and spurious) are then ordered by depth and drawn in the two image arrays. Lastly, an array of Gaussian-distribution noise is added to the depth image array.

Figure 7 shows an intensity image of a sculpture alone, and Figure 8 shows the sculpture with added clutter.

Processing in the benchmark begins with some low-level operations on the intensity and depth images, followed by some grouping operations on the intensity data that result in the extraction of candidate rectangles. The candidate rectangles are used to form partial matches with the stored models. For each model, it is possible that multiple hypothetical poses will be established. The benchmark then proceeds through the model poses, using the stored information to probe the depth and intensity images in a top-down manner. Each probe can be thought of as testing an hypothesis for the existence of a rectangle in a given location in the images. Rejection of an hypothesis, which only occurs when there is strong evidence that a rectangle is actually absent, results in elimination of the corresponding model pose. Confirmation of the hypothesis results in the

computation of a match strength for the rectangle at the hypothetical location, and an update of its representation in the model with new size, orientation, and position information. It is possible for the match strength to be as low as zero when there is no supporting evidence for the match and a lack of strong evidence that the rectangle is absent, as in the case of a rectangle that is entirely occluded by another. After a probe has been performed for every unmatched rectangle in the list of model poses, an average match strength is computed for each pose that has not been eliminated. The model pose with the highest average match strength is selected as the best match, and an image is generated that highlights the model in the intensity image. Table 4 lists all of the steps that make up the complete benchmark task.

The benchmark specification requires that this set of steps be applied in implementing a solution. Furthermore, for each step, a recommended method is described that should be followed whenever possible. However, in recognition of the fact that some methods simply may not work, or will be extremely inefficient for a given architecture, implementors are permitted to substitute other methods for individual steps. When it is necessary for an implementation to differ from the specification, the implementor is expected to supply a justification for the change. It is also urged that, if possible, a version of the implementation be written and tested with the recommended method so that the difference in performance can be determined.

6.3 Benchmark Philosophy and Rationale

In writing an integrated image understanding benchmark, the goal is to create an interpretation scenario that is an approximation of an actual image interpretation task. One must remember, however, that the benchmark problem is not an end in itself, but is a framework for testing machine performance on a wide variety of common vision operations and algorithms, both individually and in an integrated form that requires communication and control across algorithms and representations. This benchmark is not intended to be a challenging vision research exercise, and the designers feel that it should not be. Instead, it should be a balance between simplicity for the sake of implementation by participants, and the complexity that is representative of actual vision processing. At the same time, it must test machine performance in as many ways as possible. A further constraint on the design was the requirement that it make use of as many of the tasks from the first DARPA benchmark as possible, in order to take advantage of the previous programming effort.

The job of the designers was thus to balance these conflicting goals and constraints in developing the benchmark scenario. One result is that the benchmark solution is neither the most direct, nor the most efficient method of solving the problem. However, making the solution more direct would have

eliminated several of the algorithms that are important in testing certain aspects of machine performance. On the other hand, increasing the complexity of the problem to necessitate the use of those algorithms would have required significant additional processing that is redundant in terms of performance evaluation. Thus, while the benchmark solution is not a good example of how to build an efficient vision system, it is an effective test of machine performance both on a wide variety of individual operations and on an integrated task. Moreover, having taken a lesson from the Tanque Verde Suite, the benchmark design attempts to minimize the effort required of the participants, while maximizing the information obtained.

The great variety of architectures to be tested is itself a complicating factor in the design of a benchmark. It was recognized that each architecture may have its own most efficient method for computing a given function. However, the purpose of the benchmark requires that the tasks and methods be well defined so that the results from different machines will be comparable. Otherwise the results will include a significant factor that depends on the cleverness of the programmer. Thus, the benchmark specification requests that participants do not take shortcuts in the solution, and that they use the recommended methods whenever possible. It should be noted that the recommended methods are not always the most efficient techniques because they were chosen to be as widely implementable as possible. Thus, while the processing time for a given step or for the entire task may not be the best performance that a machine can muster, it will be comparable to the results from others. Participants were also encouraged to develop timings for more optimal solutions, in addition to the standard solution, if they so desired.

The designers also recognize the tendency for any benchmark to turn into a horse race. However, that is not the goal of this exercise, which is to increase the scientific insight of architects and vision researchers into the architectural requirements for knowledge-based image interpretation. To this end, the benchmark requires a much more extensive set of instrumentation than simple execution times. Participants are required to report execution time for individual tasks, for the entire task, for system overhead, input and output, system initialization and loading any precomputed data, and for different processor configurations if possible. Implementation factors that are to be reported include an estimate of the time spent implementing the benchmark, the number of lines of source code, the programming language used, and the size of the object code. Machine configuration and technology factors that are requested include the number of processors, memory capacity, data path widths, integration technology, clock and instruction rates, power consumption, physical size and weight, cost, and any limits to scaling-up the architecture. Lastly, participants

are asked to comment on any changes to the architecture that they feel would contribute to an improvement in performance on the benchmark.

Low-Level, Bottom-Up Processing	
Intensity Image	Depth Image
Label Connected Components	3x3 Median Filter
Compute K-Curvature	3x3 Sobel and Gradient Magnitude
Extract Corners	Threshold
Intermediate Level Processing	
Select Components with 3 or More Corners	
Convex Hull of Corners for Each Component	
Compute Angles Between Successive Corners on Convex Hulls	
Select Corners with K-Curvature and Computed Angles Indicating a Right Angle	
Label Components with 3 Contiguous Right Angles as Candidate Rectangles	
Compute Size, Orientation, Position, and Intensity for Each Candidate Rectangle	
Model-Based, Top-Down Processing	
Determine all Single Node Isomorphisms of Candidate Rectangles in Stored Models	
Create a List of all Potential Model Poses	
Perform a Match Strength Probe for all Single Node Isomorphisms (see below)	
Link Together all Single Node Isomorphisms	
Create a List of all Probes Required to Extend Each Partial Match	
Order Probe List According to the Match Strength of the Partial Match Being Extended	
Perform a Probe of the Depth Data for Each Probe on the List (see below)	
Perform a Match Strength Probe for Each Confirming Depth Probe (see below)	
Update Rectangle Parameters in the Stored Model for Each Confirming Probe	
Propagate Veto from Rejecting Depth Probe Over the Corresponding Partial Match	
When No Probes Remain, Compute Avg. Match Strength for Each Remaining Model Pose	
Select Model with Highest Average Match Strength as the Best Match	
Create the Output Intensity Image, Showing the Matching Model	
Depth Probe	
Select an X-Y Oriented Window in the Depth Data that will Contain the Rectangle	
Perform a Hough Transform Within the Window	
Search the Hough Array for Strong Edges with the Approximate Expected Orientations	
If Fewer than 3 Edges are Found, Return the Original Model Data with a No-Match Flag	
If 3 Edges are Found, Infer the Fourth from the Model Data	
Compute New Size, Position, and Orientation Values for the Rectangle	
Match-Strength Probe	
Select an Oriented Window in the Depth Data that is Slightly Larger than the Rectangle	
Classify Depth Pixels as Too Close, Too Far, or In Range	
If the Number of Too Far Pixels Exceeds a Threshold, Return a Veto	
Otherwise, Select a Corresponding Window in the Intensity Image	
Select Intensity Pixels with the Correct Value	
Compute Match Strength Based on Number of Correct vs. Incorrect Pixels in the Images	

Table 4: Steps that Compose the Integrated Image Understanding Benchmark

6.4 Results and Analysis

Due to limitations of time and resources, only a few of the participants were able to complete the entire benchmark exercise and test it on all five of the data sets. In almost every case, there was some disclaimer to the effect that a particular architecture could have shown better performance given more implementation time or resources. It was common for participants to underestimate the effort required to implement the benchmark, and several who had said they would provide timings were unable to complete even a portion of the task prior to the workshop. Despite requests to groups that did not attend the workshop, that they submit belated results to be included in this report, not one new benchmark report has been received. Thus, the results presented here are those that were provided by the workshop participants. In a few cases, the results have been updated, corrected, or amended since they were originally presented.

Care must be taken in comparing these results. For example, no direct comparison should be made between results obtained from actual execution and those that were derived from simulation [Carpenter, 1987]. No matter how carefully a simulation is carried out, it is never as accurate as direct execution. Likewise, no comparison should be made between results from a partial implementation and a complete one. The complete implementation must account for overhead involved in the interactions between subtasks, and even for the fact that the program is significantly larger than for a partial implementation. Consider that a set of subtasks might appear to be much faster than their counterparts in a complete implementation simply because less paging is required to keep the code in memory. It is also unwise to directly compare the raw timings, even for similar architectures, without considering the differences in technology between systems. For example, a system that executes a portion of the benchmark in half the time of another is not necessarily architecturally superior if it also has a clock rate that is twice as high or if it has twice as many processors.

In addition to the technical problems involved in making direct comparisons, there are other considerations that must be kept in mind. For example, every participant expressed the view that given more time to tune their implementation, the results for their architecture would improve considerably. What is impressive in many cases is not the raw speed increase obtained, but the increase with respect to the amount of effort required to obtain it. While this has more to do with the tools available for developing software for an architecture than with the architecture itself, it is still important in evaluating the overall usefulness of the system. Another major consideration is the ratio of cost to performance, since many applications can afford to sacrifice a small amount of performance in order to reduce the cost of the implementation. In

other applications, the size or weight or power consumption of a system may be of greater importance than all-out speed. One of the purposes of this exercise has been merely to assemble as much of this data as possible so that the performance results can be evaluated with respect to the requirements of each potential application of an architecture.

Thus, in what follows, there is no single best architecture and there are no winners or losers. Each has its own unique merits and drawbacks, of which none are absolute. To play down the direct comparison of raw timings, the results for each architecture will be presented separately. The order of presentation is random, except that the sequential solution is presented first to provide a performance baseline, and then complete parallel implementations are presented, followed by partial implementations. Results that were based on theoretical estimations are not included in this report. The timings in all of the tables are in seconds, for the sake of consistency. Where a timing is zero, it indicates that the processing time was less than the resolution of the timing mechanism employed. Blanks in the tables indicate values that were omitted from the reports that were supplied by the implementors.

6.5 Sequential Solution

The sequential solution to the benchmark was developed in C on a Sun-3/160 workstation. The solution contains roughly 4600 lines of code, including comments. The implementation was designed for maximum portability and has been successfully recompiled on several different systems. The only portion that is system dependent is the actual result presentation step, which uses the graphics primitives provided for drawing on the workstation's screen. The implementation differs from the recommended method on the Connected Component Labeling step by using a standard sequential method for computing this well-defined function. The sequential method is designed to minimize array accesses and their corresponding index calculations, which is not a problem for array processors, but incurs a significant time penalty on a sequential machine.

Timings have been produced for the sequential code running on all five data sets, and on three different machine configurations. The configurations are a Sun-3/160 (a 16 MHz 68020 processor) with 8MB of RAM, a Sun-3/260 (a 25 MHz 68020) with 16MB of RAM, and a Sun-4/260 (a 16MHz SPARC processor) with 16MB of RAM. The extra RAM on the latter two machines did not affect performance, since the benchmark was able to run in 8MB without paging. The 3/260 was equipped with a Weitek floating-point co-processor, while the 3/160 used only the standard 68881 co-processor. Table 5 shows the results for the Sun-3/160, Table 6 shows the Sun-3/260 results, and Table 7 gives the execution times for the Sun-4/260. The timings were obtained with the standard system clock utility which has a resolution of 20 milliseconds on the Sun-3 systems, and 10 milliseconds on the Sun-4.

Data Set	Sample		Test		Test2		Test3		Test4	
	User	System	User	System	User	System	User	System	User	System
Total	794.94	2.94	335.96	2.10	326.84	2.40	549.30	2.52	550.26	2.90
Overhead	4.02	1.06	4.06	0.88	4.50	1.14	4.60	1.04	4.58	0.94
Miscellaneous	2.24	0.04	2.18	0.04	2.16	0.06	2.12	0.02	2.10	0.02
Startup	0.02	0.00	0.04	0.00	0.02	0.04	0.00	0.02	0.02	0.00
Image input	0.60	0.68	0.58	0.54	1.32	0.78	1.50	0.74	1.42	0.66
Image output	0.24	0.30	0.30	0.28	0.06	0.24	0.06	0.24	0.08	0.26
Model input	0.92	0.04	0.96	0.02	0.94	0.02	0.92	0.02	0.96	0.00
Label connected components	27.40	0.38	27.46	0.36	28.12	0.28	27.86	0.36	27.88	0.36
Rectangles from intensity	6.42	0.08	4.00	0.14	4.34	0.04	5.36	0.08	5.10	0.24
Miscellaneous	2.06	0.06	1.84	0.02	1.94	0.02	1.94	0.02	1.92	0.06
Trace region boundary	0.52	0.02	0.28	0.02	0.38	0.00	0.42	0.00	0.38	0.06
K-curvature	1.62	0.00	0.80	0.00	0.82	0.00	1.22	0.00	1.10	0.00
K-curvature smoothing	1.26	0.00	0.62	0.00	0.70	0.00	0.96	0.00	1.02	0.02
First derivative	0.46	0.00	0.22	0.02	0.24	0.00	0.28	0.02	0.22	0.02
Zero-crossing detection	0.26	0.00	0.06	0.00	0.04	0.00	0.18	0.00	0.24	0.02
Final corner detection	0.20	0.00	0.16	0.02	0.18	0.02	0.28	0.02	0.16	0.04
Count corners	0.00	0.00	0.00	0.02	0.02	0.00	0.00	0.02	0.00	0.00
Convex hull	0.02	0.00	0.00	0.02	0.00	0.00	0.02	0.00	0.04	0.00
Test for right angles	0.00	0.00	0.02	0.02	0.00	0.00	0.04	0.00	0.00	0.00
Final rectangle hypothesis	0.02	0.00	0.00	0.00	0.02	0.00	0.02	0.00	0.02	0.02
Median filter	246.06	0.60	118.62	0.26	92.58	0.28	90.70	0.22	90.66	0.24
Sobel	135.3	0.18	133.14	0.16	135.92	0.18	135.12	0.16	135.14	0.28
Initial graph match	24.4	0.06	24.94	0.06	26.02	0.02	68.30	0.14	67.48	0.14
Match data rectangles	0.14	0.00	0.10	0.02	0.08	0.02	0.26	0.04	0.24	0.00
Match links	0.22	0.00	0.06	0.00	0.08	0.00	0.74	0.00	0.58	0.02
Create probe list	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.00
Partial match	24.04	0.06	24.78	0.04	25.86	0.00	67.28	0.10	66.64	0.12
Match strength probes	24.02	0.06	24.74	0.02	25.82	0.00	66.64	0.10	65.82	0.12
Window selection	0.02	0.00	0.02	0.00	0.00	0.00	0.12	0.02	0.10	0.02
Classification and count	24.0	0.06	24.72	0.02	25.82	0.00	66.50	0.06	65.70	0.08
Match extension	326.54	0.50	11.46	0.12	18.72	0.20	202.58	0.32	204.68	0.44
Match strength probes	72.88	0.10	3.28	0.00	5.80	0.06	47.82	0.06	42.00	0.06
Window selection	0.08	0.00	0.00	0.00	0.00	0.00	0.08	0.02	0.10	0.00
Classification and count	72.80	0.10	3.28	0.00	5.80	0.06	47.72	0.02	41.88	0.06
Hough probes	253.32	0.38	8.16	0.12	12.84	0.12	153.76	0.22	161.98	0.36
Window selection	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.02	0.02	0.02
Hough transform	252.20	0.36	8.10	0.12	12.78	0.12	151.86	0.16	160.34	0.28
Edge peak detection	1.08	0.02	0.06	0.00	0.06	0.00	1.76	0.00	1.54	0.02
Rectangle parameter update	0.04	0.00	0.00	0.00	0.00	0.00	0.04	0.02	0.04	0.00
Result presentation	24.80	0.00	12.28	0.04	16.64	0.02	14.78	0.00	14.74	0.02
Best match selection	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00
Image generation	24.80	0.00	12.28	0.04	16.64	0.02	14.76	0.00	14.74	0.02
Statistics										
Connected components	134		35		34		114		100	
Right angles extracted	126		99		92		210		197	
Rectangles detected	25		21		16		42		39	
Depth pixels > threshold	21256		14542		12898		18584		18825	
Elements on initial probe list	381		19		27		400		249	
Hough probes	55		3		5		97		93	
Initial match strength probes	28		20		15		142		142	
Extension mat. str. probes	60		3		5		110		97	
Models remaining	2		1		1		2		1	
Model selected	10		1		5		7		8	
Average match strength	0.64		0.96		0.94		0.84		0.88	
Translated to	151,240		256,256		257,255		257,255		257,255	
Rotated by (degrees)	85		359		114		22		22	

Table 5: Sun-3/160 Results

Data Set	Sample		Test		Test2		Test3		Test4	
	User	System	User	System	User	System	User	System	User	System
Total	293.42	5.96	130.48	2.06	116.96	2.56	191.38	3.38	192.38	3.20
Overhead	2.26	0.66	2.46	0.58	2.76	0.68	2.50	0.94	2.72	0.72
Miscellaneous	1.28	0.00	1.24	0.00	1.24	0.02	1.22	0.02	1.22	0.00
Startup	0.02	0.00	0.00	0.00	0.00	0.02	0.00	0.04	0.02	0.00
Image input	0.30	0.50	0.50	0.50	1.00	0.48	0.76	0.72	0.92	0.54
Image output	0.18	0.14	0.26	0.08	0.06	0.16	0.06	0.14	0.08	0.18
Model input	0.48	0.02	0.46	0.00	0.46	0.00	0.46	0.02	0.48	0.00
Label connected components	14.14	0.38	14.20	0.26	14.10	0.36	14.46	0.12	14.40	0.26
Rectangles from intensity	3.60	0.14	2.36	0.02	2.44	0.04	3.12	0.04	2.90	0.08
Miscellaneous	1.28	0.02	1.12	0.00	1.22	0.02	1.26	0.00	1.08	0.00
Trace region boundary	0.28	0.02	0.20	0.00	0.18	0.00	0.14	0.02	0.26	0.04
K-curvature	0.82	0.02	0.44	0.02	0.42	0.00	0.68	0.00	0.48	0.02
K-curvature smoothing	0.78	0.02	0.26	0.00	0.42	0.02	0.50	0.00	0.56	0.00
First derivative	0.20	0.02	0.16	0.00	0.10	0.00	0.18	0.06	0.26	0.00
Zero-crossing detection	0.02	0.02	0.04	0.00	0.06	0.00	0.18	0.00	0.14	0.00
Final corner detection	0.20	0.00	0.12	0.00	0.04	0.00	0.18	0.00	0.04	0.00
Count corners	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Convex hull	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.04	0.00
Test for right angles	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.04	0.00
Final rectangle hypothesis	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02
Median filter	112.50	1.20	59.86	0.42	42.64	0.46	42.64	0.34	42.72	0.54
Sobel	38.96	2.04	38.12	0.38	37.90	0.44	38.02	0.74	38.14	0.42
Initial graph match	6.10	0.06	6.06	0.02	6.38	0.20	17.02	0.30	16.80	0.14
Match data rectangles	0.08	0.00	0.06	0.00	0.04	0.00	0.14	0.02	0.12	0.00
Match links	0.10	0.00	0.04	0.00	0.04	0.00	0.30	0.00	0.26	0.00
Create probe list	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Partial match	5.92	0.06	5.96	0.02	6.30	0.20	16.58	0.28	16.42	0.14
Match strength probes	5.90	0.06	5.94	0.02	6.30	0.20	16.34	0.22	16.04	0.14
Window selection	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.02	0.02	0.00
Classification and count	5.90	0.06	5.94	0.02	6.30	0.18	16.24	0.18	16.02	0.10
Match extension	109.18	1.28	3.78	0.14	6.02	0.22	69.32	0.76	70.42	0.74
Match strength probes	17.54	0.02	0.78	0.00	1.40	0.00	11.60	0.06	10.20	0.10
Window selection	0.04	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.04	0.00
Classification and count	17.50	0.02	0.78	0.00	1.40	0.00	11.56	0.06	10.16	0.08
Hough probes	91.44	1.26	3.00	0.12	4.62	0.20	57.30	0.66	59.80	0.64
Window selection	0.04	0.00	0.00	0.00	0.00	0.00	0.04	0.02	0.02	0.00
Hough transform	90.64	1.24	2.98	0.12	4.60	0.20	56.40	0.64	59.00	0.62
Edge peak detection	0.76	0.02	0.02	0.00	0.02	0.00	0.82	0.00	0.78	0.02
Rectangle parameter update	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00
Result presentation	6.68	0.00	3.64	0.00	4.72	0.00	4.30	0.20	4.28	0.02
Best match selection	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Image generation	6.68	0.00	3.64	0.00	4.72	0.00	4.30	0.02	4.28	0.02
Statistics										
Connected components	134		35		34		114		100	
Right angles extracted	126		99		92		210		197	
Rectangles detected	25		21		16		42		39	
Depth pixels > threshold	21256		14542		12898		18584		18825	
Elements on initial probe list	381		19		27		400		249	
Hough probes	55		3		5		97		93	
Initial match strength probes	28		20		15		142		142	
Extension mat. str. probes	60		3		5		110		97	
Models remaining	2		1		1		2		1	
Model selected	10		1		5		7		8	
Average match strength	0.64		0.96		0.94		0.84		0.88	
Translated to	151,240		256,256		257,255		257,255		257,255	
Rotated by (degrees)	85		359		114		22		22	

Table 6: Sun-3/260 Results

Data Set	Sample		Test		Test2		Test3		Test4	
	User	System	User	System	User	System	User	System	User	System
Total	117.21	3.80	40.19	2.45	38.88	2.06	78.41	2.64	80.15	2.69
Overhead	2.49	1.85	2.34	1.58	2.43	1.36	2.62	1.46	2.66	1.45
Miscellaneous	1.23	1.20	1.17	0.81	1.24	0.70	1.45	0.77	1.43	0.74
Startup	0.02	0.03	0.00	0.05	0.03	0.02	0.01	0.05	0.01	0.06
Image input	0.33	0.48	0.27	0.58	0.33	0.47	0.35	0.46	0.38	0.47
Image output	0.10	0.11	0.12	0.10	0.05	1.11	0.05	0.10	0.09	0.09
Model input	0.52	0.02	0.50	0.02	0.50	0.04	0.50	0.04	0.49	0.04
Label connected components	4.39	0.35	4.29	0.27	4.31	0.23	4.36	0.26	4.33	0.28
Rectangles from intensity	1.01	0.09	0.68	0.00	0.67	0.04	0.86	0.10	0.87	0.04
Miscellaneous	0.31	0.05	0.32	0.00	0.27	0.02	0.33	0.05	0.32	0.02
Trace region boundary	0.06	0.01	0.04	0.00	0.04	0.01	0.04	0.00	0.03	0.00
K-curvature	0.21	0.00	0.05	0.00	0.11	0.00	0.08	0.00	0.08	0.00
K-curvature smoothing	0.22	0.00	0.16	0.00	0.15	0.00	0.21	0.01	0.22	0.00
First derivative	0.12	0.00	0.09	0.00	0.06	0.00	0.14	0.00	0.08	0.00
Zero-crossing detection	0.04	0.01	0.01	0.00	0.00	0.01	0.02	0.00	0.04	0.00
Final corner detection	0.04	0.01	0.01	0.00	0.03	0.00	0.02	0.02	0.06	0.00
Count corners	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02
Convex hull	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.00
Test for right angles	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00
Final rectangle hypothesis	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00
Median filter	30.33	0.20	14.47	0.17	11.14	0.16	11.16	0.14	11.15	0.19
Sobel	11.21	0.95	11.26	0.17	11.17	0.10	11.11	0.30	11.15	0.30
Initial graph match	3.41	0.01	3.36	0.10	3.53	0.01	10.01	0.09	9.83	0.11
Match data rectangles	0.03	0.00	0.00	0.03	0.02	0.00	0.05	0.01	0.04	0.02
Match links	0.07	0.00	0.01	0.01	0.02	0.00	0.22	0.01	0.18	0.00
Create probe list	0.03	0.00	0.02	0.00	0.01	0.00	0.12	0.00	0.12	0.01
Partial match	3.28	0.01	3.33	0.06	3.48	0.01	9.62	0.07	9.49	0.08
Match strength probes	3.27	0.10	3.33	0.60	3.47	0.01	9.44	0.07	9.30	0.08
Window selection	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.04	0.01
Classification and count	3.15	0.00	3.23	0.06	3.38	0.01	8.85	0.05	8.65	0.02
Match extension	60.98	0.26	2.06	0.12	3.35	0.08	36.18	0.23	38.10	0.26
Match strength probes	9.89	0.02	0.45	0.00	0.79	0.00	6.63	0.02	6.06	0.02
Window selection	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.01	0.01	0.00
Classification and count	9.60	0.00	0.44	0.00	0.78	0.00	6.12	0.00	5.56	0.02
Hough probes	50.99	0.21	1.61	0.12	2.56	0.08	29.32	0.20	31.77	0.22
Window selection	0.03	0.00	0.00	0.00	0.01	0.00	0.09	0.01	0.07	0.00
Hough transform	50.65	0.12	1.60	0.11	2.54	0.07	28.86	0.08	31.32	0.12
Edge peak detection	0.15	0.00	0.01	0.00	0.01	0.00	0.24	0.02	0.21	0.00
Rectangle parameter update	0.03	0.01	0.00	0.00	0.00	0.00	0.03	0.01	0.06	0.00
Result presentation	3.37	0.01	1.67	0.00	2.24	0.00	2.07	0.00	2.02	0.00
Best match selection	0.06	0.00	0.02	0.00	0.02	0.00	0.10	0.00	0.04	0.00
Image generation	3.31	0.01	1.65	0.00	2.22	0.00	1.97	0.00	1.98	0.00
Statistics										
Connected components	134		35		34		114		100	
Right angles extracted	126		99		92		210		197	
Rectangles detected	25		21		16		42		39	
Depth pixels > threshold	21254		14531		12892		18579		18822	
Elements on initial probe list	381		19		27		389		248	
Hough probes	55		3		5		93		92	
Initial match strength probes	28		20		15		142		142	
Extension mat. str. probes	60		3		5		105		97	
Models remaining	2		1		1		2		1	
Model selected	10		1		5		7		8	
Average match strength	0.64		0.96		0.94		0.84		0.88	
Translated to	151,240		256,256		257,255		257,255		257,255	
Rotated by (degrees)	8.5		35.9		11.4		2.2		2.2	

Table 7: Sun-4/260 Results

6.6 Alliant FX-80 Solution

The Alliant FX-80 consists of up to eight computational elements and up to twelve I/O processors that share a physical memory through a sophisticated combination of caches, buses and an interconnection network. The computational elements communicate with the shared memory via the interconnection network which links them to a pair of special purpose caches that in turn access the memory over a bus that is shared with the I/O processor caches. The FX-80 differs from the older FX-8 primarily in that the computational elements are significantly faster.

Alliant was able to implement the benchmark on the FX-80 in roughly one programmer-week. The programmer who built the implementation had no experience in vision and, in many cases, did not even bother to learn how the benchmark code works. The implementation was done by rewriting the system dependent section to use the available graphics hardware, compiling the code with Alliant's vectorizing and globally optimizing C compiler, using a profiling tool to determine the portions of the code that used the greatest percentage of CPU time, inserting compiler directives in the form of comments to break implicit dependencies in four sections of the benchmark, and recompiling the new version of the code. Alliant provided results for five configurations of the FX-80, with 1, 2, 4, 6, and 8 computational elements. In order to save space, only two of the configurations are represented here. Table 8 shows the execution times for a single FX-80 computational element, and Table 9 shows the results for an FX-80 with eight elements. Another point that was noted by Alliant is that the C compiler is a new product and does not yet provide as great a degree of optimization as their FORTRAN compiler (a difference of up to 50% in some cases). They expect to see significantly better performance with later releases of the product.

6.7 Image Understanding Architecture

Because the IUA was still under construction at the time, the simulator was used to develop the benchmark implementation. The benchmark was developed over a period of about six months, but much of that time was spent in building basic library routines and additional tools that were generally required for any large programming task. A 1/64th scale version of the simulator (4096 low-level, 64 intermediate-level, and one high-level processor) runs on a Sun workstation, and was used to develop the initial benchmark implementation. The implementation was then transported to a full-scale IUA simulator running on a Sequent Symmetry multiprocessor. Table 10 presents the results from the IUA simulations with a resolution of one instruction time (0.1 microsecond). There are several points to note about these results. Because the processing of different steps can be overlapped in the different processing levels, the sum of

the individual step timings does not always equal the total time for a segment of the benchmark. Some of the individual timings represent average execution times, since the intermediate level processing takes place asynchronously and individual processes can vary in their execution time. For example, the time for all of the match-strength probes is difficult to estimate since probes are created asynchronously and their processing is overlapped. However, the time for a step such as match extension takes into account the span of time required to complete all of the subsidiary match-strength probes. Lastly, it should be mentioned that the intermediate-level processor was greatly underutilized by the benchmark (only 0.2% of its processors were activated), and the high-level processor was not used at all. The low-level processor was also idle roughly 50% of the time while awaiting requests for top-down probes from the intermediate level.

6.8 Aspex ASP

The Associative String Processor (ASP) is being built by the University of Brunel and Aspex Ltd. in England [Lea, 1988]. It is designed as a general purpose processing array for implementation in wafer-scale technology. The processor consists of 262,144 processors arranged as 512 strings of 512 processors each. Each processor contains a 96-bit data register and a 5-bit activity register. A string consists of 512 processors linked by a communication network that is also tied to a data exchanger and a vector data buffer. The vector data buffers of the strings are linked through another data exchanger and data buffer to another communication network. One of the advantages of this arrangement is a high degree of fault tolerance. The system can be built with 1024 VLSI devices, or 128 ULSI devices, or 32 WSI devices. Estimated power consumption is 650 watts. The processor clock and instruction rate is 20 MHz. Architectural changes that would improve the benchmark performance include increasing the number of processors (improves performance on K-curvature, median filter, and Sobel), increasing the speed of the processors and communication links (linear speedup on all tasks), and adding a separate controller to each ASP substring (gives approximately an 18% increase overall).

Because the system is still under construction, a software simulator was used to implement and execute the benchmark. The benchmark was programmed in an extended version of Modula-2 over a period three months by two programmers, following a three month period of initial study of the requirements and development of a solution strategy. A Jarvis' March algorithm was substituted for the recommended Graham Scan method on the convex hull. Table 11 lists the benchmark results for the ASP.

Data Set	Sample		Test		Test2		Test3		Test4	
	User	System	User	System	User	System	User	System	User	System
Total	204.858	2.531	102.700	1.861	93.311	1.828	136.759	3.049	139.130	3.032
Overhead	7.968	0.776	7.925	0.777	7.897	0.775	7.900	0.764	7.895	0.763
Miscellaneous	0.627	0.030	0.585	0.033	0.559	0.033	0.554	0.030	0.554	0.031
Startup	0.030	0.031	0.029	0.033	0.029	0.031	0.029	0.032	0.029	0.029
Image input	5.692	0.515	5.691	0.051	5.691	0.505	5.697	0.509	5.690	0.504
Image output	1.039	0.175	1.039	0.179	1.038	0.183	1.039	0.171	1.040	0.177
Model input	0.580	0.021	0.058	0.017	0.580	0.018	0.580	0.017	0.580	0.019
Label connected components	16.917	0.268	16.830	0.258	16.800	0.253	16.948	0.247	16.930	0.259
Rectangles from intensity	2.760	0.590	1.791	0.267	1.874	0.252	2.312	0.681	2.286	0.643
Miscellaneous	1.005	0.231	0.928	0.097	0.931	0.094	0.986	0.255	0.983	0.239
Trace region boundary	0.312	0.078	0.172	0.021	0.183	0.019	0.255	0.062	0.221	0.054
K-curvature	0.592	0.037	0.287	0.017	0.308	0.017	0.438	0.045	0.432	0.045
K-curvature smoothing	0.362	0.037	0.176	0.018	0.188	0.017	0.269	0.045	0.264	0.044
First derivative	0.158	0.037	0.077	0.017	0.082	0.016	0.119	0.045	0.117	0.043
Zero-crossing detection	0.170	0.037	0.076	0.017	0.099	0.017	0.135	0.045	0.133	0.043
Final corner detection	0.135	0.042	0.060	0.022	0.069	0.022	0.103	0.051	0.101	0.049
Count corners	0.006	0.037	0.003	0.017	0.002	0.017	0.007	0.044	0.006	0.042
Convex hull	0.013	0.026	0.006	0.017	0.006	0.017	0.015	0.042	0.015	0.040
Test for right angles	0.006	0.013	0.005	0.011	0.004	0.009	0.009	0.022	0.008	0.021
Final rectangle hypothesis	0.003	0.013	0.003	0.011	0.002	0.009	0.006	0.022	0.005	0.021
Median filter	77.294	0.170	43.652	0.160	31.886	0.163	31.919	0.154	31.880	0.166
Sobel	26.147	0.001	26.079	0.001	26.063	0.001	26.128	0.001	26.129	0.001
Initial graph match	2.458	0.088	2.397	0.063	2.569	0.055	7.117	0.368	7.011	0.373
Match data rectangles	0.067	0.023	0.051	0.012	0.046	0.014	0.129	0.047	0.111	0.041
Match links	0.067	0.002	0.024	0.004	0.022	0.004	0.262	0.013	0.214	0.023
Create probe list	0.002	0.001	0.002	0.001	0.002	0.001	0.005	0.001	0.006	0.003
Partial match	2.321	0.062	2.320	0.046	2.499	0.036	6.722	0.307	6.680	0.307
Match strength probes	2.305	0.045	2.303	0.032	2.486	0.024	6.502	0.228	6.429	0.229
Window selection	0.009	0.032	0.003	0.011	0.002	0.008	0.020	0.076	0.020	0.077
Classification and count	2.299	0.015	2.298	0.011	2.482	0.008	6.471	0.076	6.397	0.076
Match extension	68.025	0.385	2.149	0.083	3.817	0.091	42.243	0.600	44.806	0.584
Match strength probes	7.139	0.096	0.311	0.005	0.568	0.008	4.600	0.168	4.216	0.155
Window selection	0.009	0.032	0.000	0.002	0.001	0.003	0.15	0.056	0.014	0.052
Classification and count	7.125	0.032	0.310	0.002	0.566	0.003	4.576	0.056	4.193	0.052
Hough probes	60.754	0.202	1.833	0.068	3.241	0.071	37.330	0.301	40.320	0.312
Window selection	0.008	0.030	0.001	0.002	0.001	0.003	0.014	0.051	0.014	0.051
Hough transform	60.259	0.082	1.806	0.061	3.210	0.061	36.650	0.097	39.604	0.110
Edge peak detection	0.474	0.031	0.026	0.002	0.030	0.003	0.642	0.050	0.681	0.050
Rectangle parameter update	0.008	0.030	0.000	0.002	0.001	0.003	0.015	0.051	0.014	0.051
Result presentation	3.269	0.002	1.860	0.002	2.388	0.002	2.177	0.002	2.174	0.002
Best match selection	0.003	0.001	0.001	0.001	0.001	0.001	0.004	0.001	0.002	0.001
Image generation	3.266	0.001	1.859	0.001	2.387	0.001	2.174	0.001	2.172	0.001
Statistics										
Connected components	134		35		34		114		100	
Right angles extracted	126		99		92		210		197	
Rectangles detected	25		21		16		42		39	
Depth pixels > threshold	21266		14542		12888		18572		18813	
Elements on initial probe list	374		19		27		389		248	
Hough probes	55		3		5		93		92	
Initial match strength probes	28		20		15		142		142	
Extension mat. str. probes	60		3		5		105		97	
Models remaining	2		1		1		2		1	
Model selected	10		1		5		7		8	
Average match strength	0.65		0.96		0.94		0.84		0.88	
Translated to	151,240		256,256		257,255		257,255		257,255	
Rotated by	85		359		114		22		22	

Table 8: Alliant FX-80 Single Processor Results

Data Set	Sample		Test		Test2		Test3		Test4	
	User	System	User	System	User	System	User	System	User	System
Total	57.177	2.935	31.056	2.082	30.872	2.043	50.357	3.577	50.153	3.467
Overhead	7.940	0.847	7.903	0.825	7.897	0.813	7.891	0.820	7.899	0.822
Miscellaneous	0.601	0.042	0.558	0.039	0.558	0.039	0.553	0.041	0.560	0.058
Startup	0.030	0.056	0.029	0.047	0.029	0.042	0.029	0.043	0.029	0.033
Image input	5.690	0.549	5.695	0.541	5.691	0.532	5.690	0.542	5.690	0.536
Image output	1.039	0.173	1.040	0.172	1.038	0.177	1.039	0.173	1.039	0.173
Model input	0.580	0.023	0.580	0.021	0.580	0.017	0.580	0.017	0.580	0.017
Label connected components	6.930	0.295	6.864	0.272	6.849	0.270	6.979	0.273	6.992	0.272
Rectangles from intensity	2.776	0.686	1.799	0.314	1.882	0.295	2.329	0.785	2.309	0.751
Miscellaneous	1.010	0.277	0.931	0.120	0.934	0.113	0.994	0.303	0.990	0.290
Trace region boundary	0.312	0.084	0.172	0.023	0.183	0.022	0.227	0.071	0.224	0.063
K-curvature	0.594	0.042	0.287	0.020	0.308	0.019	0.438	0.051	0.433	0.049
K-curvature smoothing	0.364	0.042	0.176	0.019	0.189	0.019	0.270	0.052	0.267	0.050
First derivative	0.159	0.042	0.077	0.019	0.083	0.019	0.120	0.051	0.120	0.050
Zero-crossing detection	0.171	0.049	0.077	0.020	0.100	0.019	0.136	0.051	0.135	0.050
Final corner detection	0.136	0.048	0.060	0.028	0.070	0.025	0.103	0.057	0.130	0.055
Count corners	0.007	0.041	0.003	0.019	0.003	0.019	0.008	0.050	0.007	0.052
Convex hull	0.014	0.030	0.007	0.019	0.007	0.019	0.016	0.047	0.016	0.045
Test for right angles	0.006	0.016	0.005	0.013	0.004	0.010	0.010	0.025	0.009	0.023
Final rectangle hypothesis	0.004	0.015	0.003	0.013	0.002	0.010	0.007	0.026	0.005	0.023
Median filter	9.890	0.223	5.637	0.220	4.111	0.212	4.110	0.214	4.109	0.209
Sobel	3.798	0.001	3.789	0.001	3.787	0.001	3.795	0.001	3.795	0.001
Initial graph match	2.455	0.123	2.399	0.094	2.569	0.086	7.130	0.485	7.014	0.459
Match data rectangles	0.068	0.048	0.052	0.028	0.046	0.033	0.131	0.102	0.112	0.083
Match links	0.068	0.004	0.024	0.009	0.022	0.009	0.263	0.030	0.213	0.020
Create probe list	0.002	0.001	0.002	0.001	0.002	0.001	0.005	0.001	0.006	0.004
Partial match	2.317	0.070	2.322	0.055	2.499	0.043	6.732	0.351	6.682	0.351
Match strength probes	2.301	0.050	2.304	0.037	2.485	0.027	6.509	0.259	6.429	0.263
Window selection	0.004	0.017	0.004	0.012	0.002	0.009	0.023	0.087	0.025	0.087
Classification and count	2.294	0.017	2.298	0.012	2.482	0.009	6.473	0.085	6.390	0.087
Match extension	20.105	0.455	0.786	0.107	1.376	0.122	15.926	0.739	15.845	0.702
Match strength probes	7.121	0.111	0.311	0.006	0.567	0.009	4.609	0.195	4.219	0.185
Window selection	0.010	0.037	0.001	0.002	0.001	0.003	0.019	0.065	0.016	0.065
Classification and count	7.105	0.037	0.310	0.002	0.565	0.003	4.580	0.066	4.193	0.060
Hough probes	12.847	0.243	0.468	0.086	0.799	0.099	10.996	0.378	11.350	0.366
Window selection	0.008	0.033	0.001	0.002	0.001	0.003	0.014	0.057	0.014	0.057
Hough transform	12.353	0.110	0.441	0.078	0.767	0.086	10.315	0.151	10.629	0.140
Edge peak detection	0.472	0.034	0.026	0.002	0.030	0.003	0.645	0.057	0.682	0.057
Rectangle parameter update	0.009	0.033	0.000	0.002	0.001	0.003	0.013	0.056	0.014	0.057
Result presentation	3.265	0.002	1.859	0.002	2.382	0.002	2.178	0.002	2.173	0.002
Best match selection	0.003	0.001	0.001	0.001	0.001	0.001	0.004	0.001	0.002	0.00
Image generation	3.262	0.001	1.858	0.001	2.381	0.001	2.174	0.001	2.171	0.001
Statistics										
Connected components	134		35		34		114		100	
Right angles extracted	126		99		92		210		197	
Rectangles detected	25		21		16		42		39	
Depth pixels > threshold	21266		14542		12888		18572		18813	
Elements on initial probe list	374		19		27		389		248	
Hough probes	55		3		5		93		92	
Initial match strength probes	28		20		15		142		142	
Extension mat. str. probes	60		3		5		105		97	
Models remaining	2		1		1		2		1	
Model selected	10		1		5		7		8	
Average match strength	0.65		0.96		0.94		0.84		0.88	
Translated to	151,240		256,256		257,255		257,255		257,255	
Rotated by	85		359		114		22		22	

Table 9: Alliant FX-80 Results with Eight Processors

Data Set	Sample	Test	Test2	Test3	Test4
Total	0.0844445	0.0455559	0.0455088	0.4180890	0.3978859
Overhead	0.0139435	0.0139435	0.0139435	0.0139435	0.0139435
Miscellaneous	0.0092279	0.0092279	0.0092279	0.0092279	0.0092279
Startup	0.0038682	0.0038682	0.0038682	0.0038682	0.0038682
Image input	0.0000020	0.0000020	0.0000020	0.0000020	0.0000020
Image output	0.0000020	0.0000020	0.0000020	0.0000020	0.0000020
Model input	0.0008302	0.0008302	0.0008302	0.0008302	0.0008302
Label connected components	0.0000596	0.0000596	0.0000596	0.0000596	0.0000596
Rectangles from intensity	0.0161694	0.0125489	0.0134704	0.0131378	0.0129635
Miscellaneous	0.0003227	0.0002421	0.0002010	0.0006216	0.0002421
Trace region boundary	0.0033792	0.0015472	0.0018672	0.0010912	0.0012832
K-curvature	0.0038256	0.0019936	0.0023136	0.0015376	0.0017296
K-curvature smoothing	0.0005525	0.0005525	0.0005525	0.0005525	0.0005525
First derivative	0.0003777	0.0003777	0.0003777	0.0003777	0.0003777
Zero-crossing detection	0.0000108	0.0000108	0.0000108	0.0000108	0.0000108
Final corner detection	0.0000118	0.0000118	0.0000118	0.0000118	0.0000118
Count corners	0.0000020	0.0000020	0.0000020	0.0000020	0.0000020
Convex hull	0.0036694	0.0019109	0.0015290	0.0025947	0.0026463
Test for right angles	0.0006122	0.0006009	0.0005906	0.0006421	0.0006421
Final rectangle hypothesis	0.0067877	0.0067877	0.0078821	0.0067877	0.0064229
Median filter	0.0005625	0.0005625	0.0005625	0.0005625	0.0005625
Sobel	0.0026919	0.0026919	0.0026919	0.0026919	0.0026919
Initial graph match	0.0121876	0.0076429	0.0066834	0.1124236	0.0822296
Match data rectangles	0.0029096	0.0015672	0.0013264	0.0134885	0.0106136
Match links	0.0088872	0.0056950	0.0049762	0.0985542	0.0712324
Create probe list	0.0000968	0.0001299	0.0001130	0.0009252	0.0008618
Partial match	0.0033786	0.0077033	0.0068704	0.1828976	0.1534418
Match strength probes	0.0009275	0.0011460	0.0012285	0.0025175	0.0212640
Window selection	0.0002100	0.0003000	0.0002700	0.0005700	0.0004800
Classification and count	0.0001043	0.0001490	0.0001341	0.0002831	0.0002384
Match extension	0.0300650	0.0017674	0.0024856	0.0899214	0.1277396
Match strength probes	0.0026500	0.0001146	0.0004095	0.0543250	0.0071766
Window selection	0.0006000	0.0000300	0.0000900	0.0012300	0.0016200
Classification and count	0.0002980	0.0000149	0.0000447	0.0006109	0.0008046
Hough probes	0.0068430	0.0003251	0.0005092	0.0084591	0.0109868
Window selection	0.0000675	0.0000045	0.0000090	0.0001755	0.0002385
Hough transform	0.0053010	0.0002223	0.0003036	0.0044499	0.0053477
Edge peak detection	0.0011745	0.0000783	0.0001566	0.0030537	0.0041499
Rectangle parameter update	0.0003000	0.0000200	0.0000400	0.0007800	0.0010600
Result presentation	0.0022826	0.0009452	0.0011944	0.0029768	0.0029766
Best match selection	0.0000404	0.0000403	0.0000405	0.0000406	0.0000397
Image generation	0.0022352	0.0009185	0.0011396	0.0029464	0.0029464
Statistics					
Connected components	134	35	34	114	100
Right angles extracted					
Rectangles detected	31	23	19	60	55
Depth pixels > threshold					
Elements on initial probe list					
Hough probes	44	5	8	84	100
Initial match strength probes	24	20	15	81	80
Extension mat. str. probes	20	1	3	41	54
Models remaining	3	1	1	2	1
Model selected	10	1	5	7	8
Average match strength	0.45	0.86	0.84	0.81	0.84
Translated to	151,240	256,256	257,255	257,255	257,255
Rotated by	85	359	113	23	23

Table 10: Image Understanding Architecture Results

Timings were not provided for several of the steps in the model matching portion of the benchmark, possibly because a different method was used. Startup and model input times were not listed separately, perhaps because those operations are done outside of the simulation. The miscellaneous time under overhead accounts for the input and output of several intermediate images. The miscellaneous time under the section that extracts rectangles from the intensity image accounts for the output and subsequent input of data records for corners and rectangles. No indication was given of whether any data rearrangement took place as part of these I/O operations.

6.9 Sequent Symmetry 81

The Sequent Computer Systems Symmetry 81 multiprocessor consists of multiple Intel 80386 microprocessors, running at 16.5 MHz, connected via a shared bus to a large shared memory. The particular configuration used to obtain these results included 12 processors (one of which is reserved by the system), each with an 80387 math coprocessor, and 96 MB of shared memory. The test system also contained the older A-model caches, which induce a considerably greater level of traffic on the shared bus than the newer B-model caches. An improvement of 30 to 50 percent in the overall performance is possible with the new caching system. Sequent was to have provided timings for a system with the improved cache, but they have not yet done so. The timings presented in Table 12 were obtained by the benchmark developers at UMass as part of their effort to ensure the portability of the benchmark to different systems.

About a month was spent developing the parallel implementation for the Sequent. The programmer who did the work was familiar with the benchmark, but had no previous experience with the Sequent system. Part of the development period was spent back-porting modifications to the sequential version of the benchmark in order to enhance its portability. The low-level tasks were directly converted to a parallel implementation by dividing the data sets among the processors in a manner that completely avoided write-contention. About half of the development time was spent adding the appropriate data locking mechanisms to the model-matching portion of the benchmark, and resolving problems with timing and race conditions. It was only possible to obtain timings for the major steps in the benchmark, because the Sequent operating system does not provide facilities for accurately timing individual child processes. The benchmark was run on configurations of from one to eleven processors, with the optimum time being obtained with eight or nine processors. Additional processors resulted in an overall reduction in performance, which was due to a combination of factors. As the data sets were divided among more processors, the ratio of processing time to task creation overhead decreased so

that the latter came to dominate the time on some tasks. We also believe that some of the tasks reached the saturation point of the shared bus at about eight or nine processors since one run that was observed on a B-model cache system showed performance to improve with more processors. The table shows the performance obtained for a single processor running the sequential version of the benchmark, to provide a comparison baseline, and the performance on the optimum number of processors for each data set.

6.10 Warp

The CMU Warp is a systolic array consisting of ten high speed floating point units in a linear configuration [Kung, 1984]. Processing in the Warp is directed by a host processor, such as the Sun-3/60 workstation that was used in executing the benchmark. The benchmark implementation was programmed by one person in two weeks, using a combination of the original C implementation and subroutines written in Apply and W2. The objective of the implementation was to obtain the best overall time, rather than the best time for each task. While it would seem that the latter guarantees the former, consider that the Warp and its host can work in parallel on different portions of a problem. Thus, even though the Warp could perform a step in one second that requires four seconds on the host, it is better to let the host do the processing if it would otherwise sit idle while the Warp is computing. Thus, the Warp implementation of the benchmark exploits both the tightly-coupled parallelism of the Warp array, and the loosely-coupled task-level parallelism present in the benchmark.

Table 13 lists the results for the Warp. Timings were not provided for a few of the steps, but the totals include all of the processing time. The Miscellaneous category under Overhead is the time required for downloading code to the Warp array at various stages of the processing. A figure for the total system time was provided, rather than a breakdown of system time by task. The overall Total includes the system time, which is listed on the line below the Total. Note that sums of the times for the individual steps will not equal the Total time because of the task-level parallelism that was used.

6.11 Connection Machine

The Thinking Machines Connection Machine model CM-2 is a data-parallel array of bit-serial processors that are linked by an N-dimensional hypercube router network [Hillis, 1986]. In addition, for every 32 of the bit-serial processors, a 32-bit floating-point coprocessor is provided. Connection Machines are available in configurations of 8192, 16384, 32768, and 65536 processing elements. Results were provided for direct execution on the three smaller configurations, and extrapolated to the largest configuration.

Data Set	Sample	Test	Test2	Test3	Test4
Total	0.1307200	0.0359600	0.0398100	0.1130700	0.1188200
Overhead	0.0008200	0.0008200	0.0008000	0.0008000	0.0008000
Miscellaneous	0.0002560	0.0002560	0.0002560	0.0002560	0.0002560
Startup					
Image input	0.0000512	0.0000512	0.0000512	0.0000512	0.0000512
Image output	0.0000512	0.0000512	0.0000512	0.0000512	0.0000512
Model input					
Label connected components	0.0392000	0.0228000	0.0228000	0.0348000	0.0313000
Rectangles from intensity	0.0033100	0.0029200	0.0028800	0.0031900	0.0033500
Miscellaneous	0.0000761	0.0000860	0.0000842	0.0000795	0.0000734
Trace region boundary	0.0000047	0.0000047	0.0000047	0.0000047	0.0000047
K-curvature	0.0007800	0.0007800	0.0007800	0.0007800	0.0007800
K-curvature smoothing	0.0004500	0.0004500	0.0004500	0.0004500	0.0004500
First derivative	0.0000320	0.0000320	0.0000320	0.0000320	0.0000320
Zero-crossing detection	0.0000045	0.0000045	0.0000045	0.0000045	0.0000045
Final corner detection	0.0000018	0.0000018	0.0000018	0.0000018	0.0000018
Count corners	0.0000400	0.0000380	0.0000380	0.0000530	0.0000380
Convex hull	0.0003300	0.0002820	0.0002820	0.0003300	0.0003300
Test for right angles	0.0008800	0.0008400	0.0008400	0.0009500	0.0009200
Final rectangle hypothesis	0.0004500	0.0003800	0.0002900	0.0007600	0.0007000
Median filter	0.0007200	0.0007200	0.0005100	0.0006100	0.0005100
Sobel	0.0006240	0.0006240	0.0006240	0.0006800	0.0006240
Initial graph match	0.0000090	0.0000090	0.0000090	0.0000090	0.0000080
Match data rectangles					
Match links					
Create probe list					
Partial match					
Match strength probes					
Window selection	0.0001200	0.0001320	0.0001080	0.0005500	0.0006400
Classification and count	0.0009500	0.0008850	0.0008650	0.0015400	0.0016000
Match extension	0.0835200	0.0001470	0.0001400	0.0002650	0.0002590
Match strength probes					
Window selection	0.0003000	0.0000240	0.0000360	0.0009200	0.0009800
Classification and count	0.0030000	0.0004050	0.0003520	0.0047200	0.0054500
Hough probes					
Window selection	0.0002880	0.0000240	0.0000360	0.0005800	0.0007300
Hough transform	0.0790000	0.0054000	0.0104000	0.0610000	0.0690000
Edge peak detection	0.0007700	0.0000640	0.0000990	0.0015400	0.0017600
Rectangle parameter update	0.0002160	0.0000090	0.0000100	0.0002340	0.0002360
Result presentation	0.0008500	0.0004400	0.0004700	0.0004700	0.0010300
Best match selection	0.0000250	0.0000150	0.0000150	0.0000280	0.0000150
Image generation	0.0007200	0.0003200	0.0003500	0.0008400	0.0009100
Statistics					
Connected components		34	33	113	99
Right angles extracted		99	92	210	197
Rectangles detected		21	16	42	39
Depth pixels > threshold		14533	12891	18582	18817
Elements on initial probe list					
Hough probes		3	5	97	93
Initial match strength probes		20	15	142	142
Extension mat. str. probes		3	5	110	97
Models remaining		1	1	2	1
Model selected		1	5	7	8
Average match strength		0.96	0.93	0.84	0.87
Translated to		256,256	257,255	257,255	257,255
Rotated by		359	114	22	22

Table 11: Aspex ASP Results

Data Set	Sample		Test		Test2		Test3		Test4	
	Single	Eight	Single	Eight	Single	Nine	Single	Eight	Single	Nine
Total	889.66	251.33	300.34	73.88	282.71	77.87	562.15	174.96	578.14	139.72
Overhead	5.84	6.00	5.57	5.93	5.62	5.87	5.75	5.86	5.65	5.90
System time	3.60	9.40	2.00	5.40	2.10	6.40	2.80	7.60	2.90	8.80
Label conn. components	19.27	12.68	19.34	15.83	19.29	16.01	19.60	16.84	19.58	16.89
Rectangles from intensity	4.18	1.45	2.62	0.92	2.74	1.92	3.42	1.42	3.38	1.89
Median filter	239.24	31.00	114.12	15.25	85.81	11.08	85.83	11.45	85.79	11.11
Sobel	110.89	15.00	113.21	15.46	110.80	14.83	110.84	15.20	110.81	14.73
Initial graph match	18.52	3.08	18.53	3.76	19.90	4.35	52.53	7.21	51.63	7.17
Match data rectangles	0.17	0.04	0.11	0.03	0.09	0.03	0.26	0.13	0.22	0.06
Match links	0.19	0.24	0.06	0.20	0.06	0.65	0.74	0.29	0.59	0.78
Create probe list	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Partial match	18.15	2.80	18.35	3.52	19.74	3.66	51.52	6.78	50.81	6.32
Match extension	470.90	161.34	16.16	5.97	24.08	9.38	271.07	103.99	288.21	69.10
Result presentation	20.82	20.78	10.80	10.76	14.47	14.43	13.11	12.99	13.09	12.93

Table 12: Sequent Symmetry 81 Results

The development team at Thinking Machines spent about three programmer months converting the low-level portion of the benchmark into 2600 lines of *LISP, which is a data-parallel extension to Common LISP. There was not enough time to implement the intermediate and top-down processing portions of the benchmark before the workshop, and other projects have taken priority over completing the benchmark since then. However, there was also some concern as to whether the Connection Machine would be the best vehicle for implementing the other portions, since they are more concerned with task parallelism than data parallelism. It was suggested that if the model data base included several thousand models to be matched, then an appropriate method might be found to take advantage of the Connection Machine's capabilities.

Table 14 summarizes the results for the Connection Machine on the low-level portion of the benchmark, with times rounded to two significant digits (as provided by Thinking Machines). A 32K-processor CM-2 with a Data Vault disk system and a Sun-4 host processor was used to obtain the results. The results that were supplied were for only one data set, and did not indicate which one was used. It is interesting to note that several of the tasks saw little speedup with the larger configurations of the Connection Machine. Those tasks involved a collection of contour values that had been mapped into 16K virtual processors, which are enough to operate on all of the contour points in parallel, and so there was no advantage in using more physical processors than virtual processors. It was suggested that the Connection Machine might thus be used to process the contours for several images at once in order to make use of the larger number of processors. On the other hand, for those tasks that are pixel oriented, 256K virtual processors were used and therefore a proportional speedup can be observed as the number of processors increases.

Data Set	Sample	Test	Test2	Test3	Test4
Total	43.60	20.30	22.30	58.10	55.30
System	3.00	2.30	2.50	4.30	4.90
Overhead					
Miscellaneous	3.56	2.24	2.30	5.52	7.30
Startup	5.76	6.04	5.96	5.88	6.00
Image input	3.52	3.72	5.40	5.34	5.34
Image output					
Model input	1.30	1.18	1.02	1.08	1.06
Label connected components	3.98	4.04	4.60	4.54	4.56
Rectangles from intensity					
Miscellaneous					
Trace region boundary					
K-curvature	3.14	2.24	2.20	2.272	2.54
K-curvature smoothing	1.38	0.64	0.78	0.98	0.90
First derivative	0.42	0.24	0.28	0.34	0.40
Zero-crossing detection	0.32	0.06	0.12	0.14	0.22
Final corner detection	0.16	0.10	0.12	0.22	0.20
Count corners	0.02	0.02	0.04	0.06	0.06
Convex hull	0.02	0.00	0.02	0.08	0.06
Test for right angles	0.00	0.00	0.02	0.02	0.02
Final rectangle hypothesis	0.04	0.00	0.02	0.02	0.04
Median filter	10.70	8.70	1.38	1.40	2.00
Sobel	0.48	0.48	0.72	0.94	0.92
Initial graph match	0.42	0.24	0.22	1.22	1.38
Match data rectangles	0.20	0.16	0.16	0.40	0.68
Match links	0.22	0.08	0.06	0.82	0.70
Create probe list					
Partial match					
Match strength probes					
Window selection					
Classification and count					
Match extension	24.80	3.64	4.58	38.60	41.20
Match strength probes	9.10	2.64	2.86	13.60	13.50
Window selection	0.02	0.02	0.02	0.24	0.18
Classification and count	9.00	2.56	2.82	13.20	13.10
Hough probes	15.30	0.96	1.68	23.30	25.80
Window selection	0.02	0.00	0.02	0.12	0.06
Hough transform	12.80	0.88	1.44	19.30	20.00
Edge peak detection	2.38	0.08	0.22	3.80	5.58
Rectangle parameter update	0.02	0.00	0.00	0.00	0.08
Result presentation	2.60	2.26	2.52	2.24	2.26
Best match selection	0.02	0.00	0.00	0.02	0.02
Image generation	2.54	2.20	2.46	2.16	2.18
Statistics					
Total match strength probes	91	23	20	247	239
Hough probes	58	3	5	97	95

Table 13: Results for the Warp

Configuration	8K	16K	32K	64K
Total (low level tasks only)	1.26	0.91	0.71	0.63
Overhead				
Miscellaneous				
Startup	0.10	0.10	0.10	0.10
Image input	0.155	0.155	0.155	0.155
Image output				
Model input				
Label connected components	0.34	0.21	0.14	0.10
Rectangles from intensity				
Miscellaneous				
Trace region boundary	0.44	0.30	0.23	0.17
K-curvature	0.019	0.019	0.018	0.018
K-curvature smoothing	0.0056	0.0055	0.0062	0.0055
First derivative	0.00038	0.00037	0.00037	0.00037
Zero-crossing detection	0.00021	0.00020	0.00019	0.00019
Final corner detection	0.0058	0.0053	0.0053	0.0053
Count corners	0.018	0.016	0.016	0.016
Convex hull	0.041	0.038	0.039	0.038
Test for right angles				
Final rectangle hypothesis				
Median filter	0.082	0.041	0.025	0.015
Sobel	0.052	0.026	0.014	0.008

Table 14: Results for the Connection Machine on the Low-Level Portion

6.12 Intel iPSC-2

The Intel Scientific Computers iPSC-2 is a distributed memory multiprocessor that consists of up to 128 Intel 80386 microprocessors that are linked by a virtual cut-through routing network which simulates point-to-point communications. Each of the microprocessors can have up to 8 MB of local memory, and an 80387 arithmetic coprocessor. The benchmark implementation for the iPSC-2 was developed by the University of Illinois at Urbana-Champaign using C with a library that supports multiprocessing. The group had only enough time to implement the median filter and Sobel steps of the low-level depth image processing. However, they did run those portions on five different machine configurations, with 1, 2, 4, 8, and 16 processors, and on four of the five data sets. Table 15 presents their results, which are divided into user time and system time (including data and program load time, and output time).

Configuration	1		2		4		8		16	
	User	System	User	System	User	System	User	System	User	System
Median Filter										
Sample	176.47	0.00	87.93	11.52	43.46	11.23	22.27	3.1	11.14	3.82
Test	75.45	0.00	37.72	10.88	18.99	10.84	9.66	3.15	4.84	3.87
Test2	60.84	0.00	30.36	11.48	15.25	11.45	7.63	3.73	3.81	4.19
Test3	60.83	0.00	30.36	11.12	15.25	11.23	7.63	3.49	3.82	4.03
Sobel										
Sample	78.63	0.00	39.32	3.53	19.68	3.00	9.84	2.37	4.92	2.91
Test	80.82	0.00	40.42	3.47	20.25	2.89	10.15	2.43	5.10	2.82
Test2	80.82	0.00	40.42	1.46	20.25	1.99	10.15	1.87	5.10	2.50
Test3	78.63	0.00	39.31	2.62	19.68	2.51	9.84	2.17	4.92	2.69

Table 15: iPSC-2 Results for Median Filter and Sobel Steps

6.13 Comparative Performance Summary

As mentioned above, the direct comparison of raw timings is not especially useful. We have attempted to provide as much information about each benchmark implementation as is necessary for others to make informed and intelligent comparisons of the results. For example, a valid comparison of architectural features should take into account the technology, instruction rate, and scalability of the processors that were actually used to obtain the results. On the other hand, a comparison that seeks to establish the currently available machine with the best cost to performance ratio should look at the timings with respect to both the programming effort required and the price of the hardware. The authors hope to develop and publish some direct comparisons of architectural features, once a few more implementations are added to the sample and a reasonably broad set of scaling functions is established.

In the meantime, one interesting comparison that can be immediately drawn from the data, which requires no scaling for technology, is the relative amount of processing time that each architecture expends on each portion of the benchmark. This function, which is just the percentage of the total time taken for each step, provides an indication of those tasks that each architecture excels at and those that it struggles with. Tables 16 through 20 compare the efforts for the different architectures on each of the major benchmark steps, for the five data sets. It should be noted that the data sets Test and Test2 require very little model matching effort since they involve very simple models. The other three data sets involve more complex models, which is easily seen in Tables 16, 19, and 20. Only the complete implementations are listed, since a total time for the benchmark is required to compute the values in the tables. Blanks in the tables represent information that was missing from the reports by the different groups.

Architecture	Sun-3	Alliant	IUA	ASP	Sequent	Warp
Overhead	0.6	14.6	16.5	0.6	2.3	
Label connected comp.	3.5	12.0	0.1	30.0	4.9	9.1
Rectangles from intensity	0.8	5.8	19.1	2.5	0.6	
Median filter	30.9	16.8	0.7	0.6	11.9	24.5
Sobel	17.0	6.3	3.2	0.5	5.8	1.1
Initial graph match	3.1	4.3	14.4	0.0	1.2	1.0
Match data rectangles	0.0	0.2	3.5		0.0	0.5
Match links	0.0	0.1	10.5		0.1	0.5
Create probe list	0.0	0.0	0.1		0.0	
Partial match	3.0	4.0	4.0		1.1	
Match extension	40.9	34.2	35.6	63.9	61.9	56.9
Result presentation	3.1	5.4	2.7	0.7	8.0	6.0

Table 16: Distribution of Processing Time for Data Set Sample

Architecture	Sun-3	Alliant	IUA	ASP	Sequent	Warp
Overhead	1.5	26.4	30.9	2.3	3.1	
Label connected comp.	8.2	28.9	0.1	63.4	9.4	19.9
Rectangles from intensity	1.2	6.4	27.5	8.1	0.9	
Median filter	35.2	17.7	1.2	0.2	34.6	42.9
Sobel	39.4	11.4	5.9	1.7	34.4	2.4
Initial graph match	7.4	7.5	16.8	0.0	6.0	1.2
Match data rectangles	0.0	0.2	3.4		0.0	0.8
Match links	0.0	0.1	12.5		0.1	0.4
Create probe list	0.0	0.0	0.3		0.0	
Partial match	7.3	7.2	16.9		5.8	
Match extension	3.4	2.7	3.9	0.4	5.9	17.9
Result presentation	3.6	5.6	2.1	1.2	5.8	11.1

Table 17: Distribution of Processing Time for Data Set Test

Architecture	Sun-3	Alliant	IUA	ASP	Sequent	Warp
Overhead	1.7	26.5	30.6	2.0	3.2	
Label connected comp.	8.6	21.6	0.1	57.3	9.8	20.6
Rectangles from intensity	1.3	6.6	29.6	7.2	1.3	
Median filter	28.2	13.1	1.2	1.3	26.9	6.2
Sobel	41.3	11.5	5.9	1.6	34.8	3.2
Initial graph match	7.9	8.1	14.7	0.0	6.7	1.0
Match data rectangles	0.0	0.2	2.9		0.0	0.7
Match links	0.0	0.1	10.9		0.3	3.7
Create probe list	0.0	0.0	0.2		0.0	
Partial match	7.9	7.7	15.1		6.4	
Match extension	5.7	4.6	5.5	0.4	9.2	20.5
Result presentation	5.1	7.2	2.6	1.1	7.9	11.3

Table 18: Distribution of Processing Time for Data Set Test2

Architecture	Sun-3	Alliant	IUA	ASP	Sequent	Warp
Overhead	1.0	16.2	3.3	0.7	1.6	
Label connected comp.	5.1	13.4	0.0	30.8	4.9	7.8
Rectangles from intensity	1.0	5.8	3.1	2.8	0.7	
Median filter	16.5	8.0	0.1	0.5	13.2	2.4
Sobel	24.5	7.0	0.6	0.6	17.1	1.6
Initial graph match	12.4	14.1	26.9	0.0	8.1	2.1
Match data rectangles	0.1	0.4	3.2		0.1	0.7
Match links	0.1	0.5	23.6		0.1	1.4
Create probe list	0.0	0.0	0.2		0.0	
Partial match	12.2	13.1	43.7		58.3	
Match extension	36.8	30.9	21.5	0.2	50.9	66.4
Result presentation	2.7	4.0	0.7	0.4	3.5	3.9

Table 19: Distribution of Processing Time for Data Set Test3

Architecture	Sun-3	Alliant	IUA	ASP	Sequent	Warp
Overhead	1.0	16.3	3.5	0.7	1.6	
Label connected comp.	5.1	13.5	0.0	26.3	5.1	8.2
Rectangles from intensity	1.0	5.7	3.3	2.8	0.7	
Median filter	16.4	8.1	0.1	0.4	13.5	3.6
Sobel	24.5	7.1	0.7	0.5	17.5	1.7
Initial graph match	12.2	13.9	20.7	0.0	8.2	2.5
Match data rectangles	0.0	0.4	2.7		0.0	1.2
Match links	0.1	0.4	17.9		0.2	1.3
Create probe list	0.0	0.0	0.2		0.0	
Partial match	12.1	13.1	38.6		8.0	
Match extension	37.1	30.9	32.1	0.2	49.8	74.5
Result presentation	2.7	4.1	0.7	0.9	3.6	4.1

Table 20: Distribution of Processing Time for Data Set Test4

6.14 Recommendations for Future Benchmarks

At the conclusion of the Avon workshop, a panel session was held to discuss the benchmark, ways it could be improved, and future benchmark efforts. The general conclusion of the participants was that the benchmark is a significant improvement over past efforts, but that there is still work to be done.

One of the major complaints was the sheer size and complexity of the benchmark solution. The sample solutions are a considerable help in this regard, but a great deal of work is still required to transport them to parallel architectures. Several people expressed the opinion that a FORTRAN version should be made available

so that the benchmark would be taken up by the traditional supercomputing community. It was pointed out that most groups don't have the time or resources to implement such a complex benchmark, and that it would be almost impossible to tune it for optimum performance as is done with smaller benchmarks. A counter-argument was voiced that most vision applications are not highly tuned, and that the benchmark might therefore give a more realistic indication of the performance that could be expected. Suggestions for reducing the size of the benchmark included removing one of the top-down probes (although there was no consensus on which one should be removed), and simplification of the graph matching code through increased generality.

On the other hand, several people complained that the benchmark task was too small. The groups that had benchmarked data-parallel systems all indicated that they would like to see data sets involving thousands of models so that they could exploit more data parallelism, rather than being forced into a task parallel model. Of course, those who had benchmarked multi-tasking systems took the opposite view. It was then suggested that an interesting variation on the benchmark would be to provide a range of data sets with model-bases ranging through several orders of magnitude. Such data sets would provide another dimension to the performance analysis, and thus some insight into the range of applications for which an architecture is appropriate. Beyond simply increasing the size of the model-base, several of the vision researchers expressed a desire to see a broader range of vision tasks in the benchmark. For example, motion analysis over a succession of frames would test an architecture's ability to deal with real-time image input and would help to identify those with a special ability to pipeline the stages of an interpretation. However, there was an immediate outcry from the implementors that the benchmark is already too complex. It was then suggested that an optional second level of the benchmark could be specified that would be based on the basic task, but extended to include image sequences and motion processing.

An important observation was made that the complexity of the benchmark was not the issue, but the cost of implementation. It was suggested that the benchmark might be more palatable if it was reorganized to be built out of a standard set of general purpose vision subroutines. Even though a group might still have to implement all of those routines, they would then at least have a library that could be used for other applications, over which they could amortize the cost. The benchmark specification would then be a framework for applying the library to solve a problem, and could involve separate tests for evaluating the performance and accuracy of the individual subroutines.

Part of the discussion focused on the fact that the benchmark does not truly address high-level processing. However, as the benchmark designers were quick

to point out, there is no consensus among the vision research community as to what constitutes high-level processing. Until agreement can be reached on what types of processing are essential at that level, it will be pointless to try to design a benchmark that includes the high level. It was also noted that the current top-down direction of low-level processing by the benchmark has some of the flavor of the high-level control of intermediate- and low-level processing which many people feel is necessary. In the end, it was decided that the community is not yet ready to define high-level processing to the degree necessary to build a benchmark around it.

Another point was that a standard reporting form should be developed, and that the sequential solution should output its results to match that form. Although the benchmark specification included a section on reporting requirements, the sequential solution did not precisely conform to it (partly because many of the reporting requirements were for aspects of the implementation that went beyond the timings and statistics that were to be output). In fact, most of the groups followed the example of the reporting format for the sequential solution, rather than what was requested in the specification. It was also noted that because the benchmark allows alternate methods to be used whenever dictated by architectural considerations, the reporting format can not be made completely rigid.

The conclusion of the panel session was to let the benchmark stand as specified for some period of time, in order to allow more groups to complete their implementations. Then a new version of the benchmark should be developed with the following features: It should be a reorganization of the current problem into a library of useful subroutines and an application framework. A set of individual problems should be developed to test each of the subroutines. A broader range of data sets should be provided, with the size of the model-base scaling over several orders of magnitude, and perhaps a set of images of different sizes. The graph matching code should be simplified and made more general purpose. A standard reporting format should be provided, with the sample solutions generating as much of the information as possible. Lastly a second level of the benchmark might be specified that extends the current problem to a sequence of images with motion analysis. The second level would be an optional exercise that could be built on top of the current problem to demonstrate specific real-time capabilities of certain architectures.

6.15 Benchmark Efforts Under This Contract

We have continued to supervise the distribution of the IU Benchmark under this effort. This supervision includes the shipping of approximately 13 tapes containing the benchmark code and data sets to the sites listed in Table 21. In addition, following an upgrade of the UMass network connection, it became

practical to distribute the benchmark via anonymous FTP. An account was established to provide such access and we have not kept track of all of the people who have copied the benchmark via that means. To date, not one of the recipients has returned performance results to us. In one or two cases we have heard that the benchmark has been implemented on machines, but the performance was so poor that the manufacturers decided not to submit a report. In other cases we have been told that the benchmark was simply too complex, and that the Image Understanding market did not justify the expenditure of the effort required to carry out the implementation. Most benchmarking by manufacturers is conducted by their marketing departments, who must justify the effort by the number of sales that the benchmark results will generate. Thus, it isn't surprising that poor performance and a small market would lead to a lack of implementation reports. We have also been informed that the benchmark was used as a canonical vision task in a Ph.D. thesis study. While we are gratified by such use, we always caution that the benchmark as a whole is not a representative vision task -- it merely provides a representative environment in which to execute the individual routines that comprise it.

Center for Night Vision and Electro-Optics
International Parallel Machines
Tjipto Santoso
Booz-Allen&Hamilton Inc.
University of South Carolina
North Carolina State University
Univ. of California Irvine
University of Bristol
University of Washington
MasPar
Intel Scientific Computers (second copy)
University of South Florida
National Institute of Standards Technology

Table 21: Additional Distribution of the Second DARPA Benchmark

Work began on the recommended changes to the benchmark, and we recoded all but one of the benchmark routines (the graph matcher) as separate subroutines that could be called by a benchmark driver program. We also developed a model and image data generator that will simplify distribution of the test data for the models. The generator can either output random data or can input a file that describes how to construct the model and the imagery. Thus, users can automatically generate new test sets, and we can still specify required test sets for purposes of comparison.

The IUA implementation of the benchmark was further debugged and its performance was enhanced under this effort. That performance is reflected in the tables above.

7. Routing in the CAAPP

The critical problem in creating practical online SIMD mesh routing algorithms is minimizing both the number of communication steps and the size and complexity of the queues required at each PE. Currently, the best available algorithms for likely array sizes require $16n$ routing steps with queue size 1; if priority queues of size $2q - 1$ are allowed, the number of routing steps required is reduced to $14n/q + 2n$. We have developed an algorithm (the MGRA discussed below), based on wormhole routing, that has routed a large number of communication patterns (i.e. all patterns tried besides a synthetically constructed worst case) in $5n$ routing steps with a FIFO queue of size 2. We also show that the MGRA can be modified for meshes with broadcast buses and reconfigurable broadcast buses to route in a similar number of routing steps but with a queue size of 1. A second algorithm (the CGRA) uses reconfigurable broadcast buses in implementing cut-through routing. Using the CGRA, sparse patterns are routed in a small constant number of communication steps.

7.1 Outline of the Mesh Greedy Routing Algorithm

The basic algorithm, we call the *mesh greedy routing algorithm* or MGRA, runs as follows. Every PE emulates a local section of two communication channels, X and Y, by using the nearest neighbor mesh and space allocated in on-chip memory. The X-channel and Y-channel are arbitrarily chosen to run in directions parallel to the rows and columns respectively. Conceptually, the algorithm runs as follows: PEs inject packets into the network, which are sent through the X-channel a distance of one PE per routing step until the correct X coordinate (column) is reached. At this point the packet is moved from the X-channel to the Y-channel. The packet then travels through the Y-channel until the destination is reached. The X- and Y-moves are interleaved so that each occurs during every iteration. Packets travel in only one direction in each channel and wraparound is used; because the packets have only unit length (are made up of single *flits*), having single X- and Y-channels does not cause deadlock. If the packet has reached the correct X coordinate but the section of the Y-channel being emulated at that PE is occupied, then the packet is "blocked," as are all the other packets contiguously behind that packet in the X-channel. Y-channels are never blocked, so overall progress is assured.

Initially, each iteration contains two data movement instructions, one in the X-channels and one in the Y-channels. After every iteration, however, the controller checks to see if there are still packets in the X-channels. If none remain, a second phase of the MGRA is begun where only Y-channel moves are executed. During this phase, the controller checks the Y-channels for packets: when none remain, the algorithm is terminated.

The critical problem in this algorithm is how to handle the collisions that occur when a packet needs to switch from an X- to a Y-channel but finds the Y-channel already occupied. If the X-channel packet is simply left in place, that packet runs the danger of being overwritten by a packet arriving in the next iteration. Naive approaches are to emulate queues within each PE as in the MIMD greedy routing algorithm, or to include a notification step. In the latter approach, each PE with a blocked packet sends a message to the packets contiguously behind it informing them that they too are blocked and should not proceed in the next iteration. However, both alternatives yield $\theta(N)$ algorithms: the former approach requires queues of length $\theta(n)$, while the notification step requires n data transfers. How we deal with collisions is the key difference among the implementations on the MGRA in the different models.

7.2 The MGRA on the Basic Mesh-Connected Processor Model

In the basic mesh-connected processor model we handle collisions by emulating queues of size two in the X-channels. The details are as follows: we adapt the MGRA by adding another buffer to the X-channel; we call the two buffers X-head and X-tail. The algorithm now has some additional steps interposed: instead of transferring packets directly from X-channel buffer to X-channel buffer, a PE moves packets from its X-head to the X-tail of its neighbor, and then internally from its X-tail to its X-head. Of course, in either case, PEs only send packets if the destination buffer is clear. The correctness can be seen intuitively from the fact that the "blocked" information travels back down the train of contiguous packets at the same rate that incoming packets become compressed in the trailing queues.

7.3 The MGRA with Longer Queues

Here we consider the MGRA with FIFO queues of arbitrary length. We did not attempt to implement priority queues, a structure that---although it would enable an optimal algorithm in terms of communication steps---would have unacceptable overhead.

Queues can be emulated on the basic model in several ways. However, when a queue size greater than two (all we needed above) must be supported, the algorithms are slightly more complicated. The intuitive method is to use a circular buffer and a bit vector. A somewhat simpler method using no bit vector is to simply keep the queue "justified" to one end of the buffer.

If hardware support for local indexing is available, then FIFO queues can be implemented directly. As mentioned earlier, however, the queue and dequeue operations are often substantially slower (by about a factor of 10) than the PE to PE move operations. Therefore even if local indexing is available, longer queues

should only be used if the reduction in the number of communication operations is greater than the slowdown in memory access.

7.4 The MGRA on a Mesh with Broadcast Buses

Even queues of length two create unwanted overhead through internal move instructions. In this section we show how the MGRA can be modified to use broadcast buses to reduce the X-channel queue size to one. This has two advantages: 1) internal moves are eliminated and 2) nearest neighbor transfers are faster for some mesh architectures when the packet memory address in the source PE is identical to the memory address in the destination PE. The problem again is dealing with collisions: we must keep from overwriting packets that are blocked because of occupied Y-channels. We use the following solution: since a packet can only be overwritten by another packet in an X-channel, PEs with blocked packets broadcast that status to their rows. Therefore, if any packet in a row is blocked, then no packet in that row proceeds.

7.5 The MGRA on a Mesh with Reconfigurable Buses

The obvious disadvantage of the above method is that some packets are needlessly prevented from proceeding. When the broadcast buses are reconfigurable, however, it is possible retain the queue size of one while blocking only those packets that could overwrite a blocked packet. The method is as follows. Each PE containing a packet in its X-channel closes its East and West switches, while all PEs open their North and South switches. If the PE contains a blocked packet, then the West switch is opened. In this way circuits are formed along the horizontal buses that are made up of contiguous PEs containing packets in their X-channels; if there is a PE with a blocked packet within the circuit, it will be in the leftmost PE.

7.6 The Four Channel MGRA on the Basic Model

One way to cut down on the number of blocked packets is to double the number of channels emulated so that the approach is more similar to that in [Dally87]; in this case the overhead per iteration is also roughly doubled. We call these new channels X2 and Y2. When a packet reaches the last row (column) of the torus, but has not yet reached its destination column (row), the packet switches channels to X2 (Y2) and wraps around. This scheme does indeed cut down on the congestion, but was not found to be worth the overhead.

A much more effective scheme is for the additional channels X2 and Y2 to route packets in the opposite directions of X1 and Y1, respectively. Packets are injected into the X1 or X2 (Y1 or Y2) channels so that they always travel by a shortest path from source to destination. This is advantageous when the

maximum shortest path is significantly less than n (in Manhattan distance). The complexity of the channel emulation is significantly more than doubled, however, as there are now 4 ways that X- and Y-channels can interact rather than 1. We call this variation the 4-channel MGRA. Again, this algorithm does not deadlock because the packets are transferred in their entirety.

7.7 The Coterie Greedy Routing Algorithm

Reconfigurable buses can also be used to emulate cut-through routing. Cut-through routing was developed by Kermani and Kleinrock [Kermani 1979] as a hybrid of store-and-forward and point-to-point circuit switched routing. Each packet is routed through the network to the furthest available PE toward its destination, where it is queued in its entirety. The advantage is that packets need not wait for the entire circuit between source and destination to be free before transfer, while also avoiding the need to be queued at every intermediate PE. The implementation of cut-through routing on reconfigurable buses involves substantially greater overhead per iteration than the MGRA, but as we shall see below, this is more than compensated by the decreased number of iterations when the communication pattern is sparse.

In this version of the greedy algorithm, which we call the Coterie Greedy Routing Algorithm (CGRA), the X- and Y-channels are emulated not only by the nearest neighbor connections, but also by the reconfigurable broadcast buses. The major consequence is that rather than moving packets just one PE at a time, all of the open space between occupied PEs is traversed in a single iteration of the algorithm. The basic idea is to create circuits having the property that the rightmost PE (bottom-most if these are Y-channels) contains a packet, while all other PEs in the circuit do not. The occupied PE then broadcasts its packet to the circuit, where it is read either by the destination or by the leftmost (topmost) PE.

7.8 Many-to-One Routing

A *combine* operation has been created by augmenting the routing algorithms as follows: Instead of simply moving the packets that have arrived at their destinations from the Y-channel(s) to the output buffer, a binary operator is interposed. For example, sum-combine adds the value in the packet to the value already in the output buffer. Many-to-one routing is implicit in the combine operation; more congestion is therefore likely to occur than in permutation routing. To deal with this situation, intermediate combining at the point of collision may optionally be executed. The cost is an increase in overhead of an extra compare and arithmetic operation for each iteration, but there are situations where intermediate combining is worthwhile. One example is the degenerate case where the entire array is combined at one destination: the complexity of that combine operation is reduced from $O(N)$ to $O(n)$.

8. Conclusions

The three-level structure of the Image Understanding Architecture supports the necessary hierarchy of abstractions for the different representations and operations that we believe are needed to generally solve the vision problem. Each level is constructed to perform a suite of tasks most appropriate for that level of abstraction.

The CAAPP is optimized to perform local operations on neighborhoods of pixels and to provide feedback to the higher levels of processing about the state of the computation and statistics about low-level data. It excels at very tightly-coupled fine-grained parallelism. The mapping of one pixel onto each processor ensures that the maximum amount of parallelism available in the low-level vision tasks will be utilized. The reconfigurable nature of its Coterie network supports communication among groups of processors that correspond to image events. The global summary feedback capabilities and I/O subsystem of the CAAPP make it especially well suited for real-time applications. Of course, its unique feature is its interface to the ICAP level.

The ICAP is designed to support the necessary tasks of building an intermediate symbolic representation of the image, and operating on that representation. These operations need two primary capabilities: data manipulation and communication. The data representations used by the CAAPP need to be transformed by the ICAP into a more accessible format, and then passed to neighboring ICAP cells to perform merging and grouping operations.

The high level tasks which perform knowledge-based inference and manipulation of object models are run in the SPA. To support distributed artificial intelligence processing, powerful processors are needed with large amounts of memory. The communication between processes will primarily be in terms of a blackboard system managed by the processes themselves. As these processes run and make requests via the ACU to the ICAP (and sometimes directly to the CAAPP) they will extract information about the image and post the results of their analysis on the blackboard for other processes to use. The end result will be an interpretation of the image achieved by cooperation of the set of object processes.

The IUA simulator, parallel language extensions to FORTH and C, Apply compiler, debugger, subroutine libraries, and applications provide an environment for developing code and demonstrating the IUA prototype. The simulator, in particular, permits the programmer to directly observe the status of algorithms running in the machine as they are executing. The wealth of visual information its displays provide is very helpful to both experienced and novice programmers.

The IUA prototype hardware has been completed, and is fully functional, although it operates at a speed that is less than originally specified and has remained at Hughes to support further development. The sources of the reduced performance have been determined, and could be corrected if funding were available. The hardware has been demonstrated running several vision algorithms. The software environment for the IUA simulators has been transported to the prototype through a combined effort at Hughes and UMass.

A second generation of the IUA has been specified, based on our experience with developing the prototype and in using our simulators. The new design quadruples the number of processors in a CAAPP chip, enhances I/O, changes the ICAP processor to a 32-bit DSP, and gives the ICAP a full-connect network linking up to eight groups of eight processors that share a memory. Designs have been developed for interconnection networks based on the original 16-bit DSP chip and on the new 32-bit chip. The latter designs include a custom VLSI chip architecture that is estimated to support a 900 MB/S aggregate transfer rate.

The DARPA Integrated Image Understanding Benchmark is another step in the direction of providing a standard exercise for testing and demonstrating the performance of parallel architectures on a vision-like task. While not perfect, it is a significant improvement over previous efforts in that it tests performance on a wide variety of operations within the unifying framework of an overall task. The benchmark also goes a long way toward eliminating programmer knowledge and cleverness as a factor in the performance results, while providing sufficient flexibility to allow implementors to take advantage of special architectural features.

Complete implementations have only been developed for a handful of architectures to date, but it is hoped that others will be added to the sample. In the meantime, it is possible to draw a few general conclusions from the data that has been gathered. It is clear that a tremendous speedup is possible for the data parallel portions of the interpretation task. However, every one of the architectures in the sample devoted the greatest percentage of its overall time to the model matching portion of the benchmark on those data sets that involved complex models. One conclusion might be that this portion of the task simply doesn't permit the exploitation of much parallelism. However, when the model matching step is viewed at an abstract level, it appears to be quite rich with potential parallelism, but, in the form of task parallel direction of limited data parallel processing. While this style of processing can be sidestepped by increasing the size of the model-base so that the entire task becomes data parallel in nature, the inclusion of true high-level processing will force us back to dealing with this processing model. Thus, one potential area for research that the benchmark points out is the development of architectures, hardware and

programming models to support task parallelism which can direct data parallel processing in a tightly coupled manner.

The CAAPP routing algorithms developed have demonstrated that a reconfigurable mesh is capable of supporting permutation routes with performance that is comparable to a Thinking Machines CM-2 which has a dedicated router. The advantage of the CAAPP approach is that it is simpler, lower in cost, and does not have a hardware-imposed upper bound on scaling.

9. References

- [Annexstein, 1990] F. Annexstein and M. Baumslag: A Unified Approach to Offline Permutation Routing, 2nd ACM Symp. on Parallel Algorithms and Architectures, 1990.
- [Batcher, 1980] Batcher, K. E., Design of a Massively Parallel Processor, IEEE Trans. Comp., Vol. C-29, No. 9, September 1980.
- [Beveridge, 1989] J.R. Beveridge, J. Griffith, R.R. Kohler, A.R. Hanson, E.M. Riseman: Segmenting Images Using Localized Histograms and Region Merging, International Journal of Computer Vision, 2, 1989.
- [Carpenter, 1987] Carpenter, Robert J., Performance Measurement Instrumentation for Multiprocessor Computers, Report NBSIR 87-3627, U.S. Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology, Gaithersburg, MD, August, 1987, 26pp.
- [Dally, 1986] W.J. Dally and C.L. Seitz: The Torus Routing Chip, Distributed Computing, 1 (3), 1986.
- [Dally, 1987] W.J. Dally and C.L. Seitz: Deadlock Free Routing in Multiprocessor Interconnection Networks, IEEE Trans. on Comp., 36 (5), 1987.
- [Duff, 1978] Duff, M.J.B., Review of the CLIP Image Proceeding System, Proceedings of the National Computer Conference, 1978, AFIPS, pp. 1055-1060.
- [Duff, 1986] Duff, M.J.B., How Not to Benchmark Image Processors, in Evaluation of Multicomputers for Image Processing, L. Uhr, K. Preston, S. Levialdi, and M.J.B. Duff, Eds., Academic Press, Orlando, FL, 1986, pp. 3-12
- [Foster, 1971] Foster, C. C., Stockton, F. D., Counting Responders in an Associative Memory, IEEE Transactions on Computers, Vol. C-31, No. 12, Dec. 1971, pp. 1580-1583.
- [Graham, 1972] R.L. Graham: An Efficient Algorithm for Determining the Convex Hull of a Planar Set, Information Processing Letters, 1, 1972.
- [Hanson, 1986] Hanson, A. R., Riseman, E. M., A Methodology for the Development of General Knowledge-Based Vision Systems,. In: Vision, Brain, and Cooperative Computation, M. Arbib and A. Hanson (Eds.), MIT Press, Cambridge, 1986.
- [Herbordt, 1990a] M.C. Herbordt, C.C. Weems, D.B. Shu: Routing on the CAAPP, Proceedings of the 10th Int. Conf. on Pattern Recognition, 1990.

- [Herbordt, 1990b] M.C. Herbordt, C.C. Weems, J.C. Corbett: Message Passing Algorithms on a SIMD Torus with Coteries, Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures, 1990.
- [Hillis, 1986] Hillis, Daniel W., The Connection Machine, MIT Press, Cambridge, 1986
- [Kermani, 1979] P. Kermani and L. Kleinrock: Virtual Cut-Through: A New Computer Communication Switching Technique, Comp. Networks, 3, 1979.
- [Kumar, 1985] Kumar, V.K.P., Raghavendra, C.S., Array Processor with Multiple Broadcasting, Proc. 12th Annual Symp. Computer Architecture, Association for Computing Machinery Press, 1985.
- [Kung, 1984] Kung, H.T., and Onat Menzilcioglu, Warp: A Programmable Systolic Array Processor, Proc. SPIE Symp., Vol. 495, Real-Time Signal Processing VII, Aug. 1984
- [Lea, 1988] Lea, R.M., ASP: A Cost-effective Parallel Microcomputer, IEEE Micro, October, 1988, pp. 10-29
- [Leighton, 1989] F.T. Leighton, F. Makedon, I. Tollis: A $2n - 2$ Step Algorithm for Routing in an $n \times n$ Array With Constant Size Queues, 1st ACM Symp. on Parallel Algorithms and Architectures, 1989.
- [Li, 1987] H. Li and M.Maresca: Polymorphic-Torus Network, Proc. International Conference on Parallel Processing, 1987.
- [Little, 1989] J.J. Little, G.E. Bluelloch, T.A. Cass: Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine, IEEE Trans. on PAMI, 11 (3), 1989.
- [McCormick, 1963] McCormick, B.T., The Illinois Pattern Recognition Computer -- ILLIAC III, IEEE Trans. on Elect. Computers, Dec., 1963, pp. 791-813.
- [Miller, 1988] R. Miller, V.K. Prasanna Kumar, D. Reisis, Q.F. Stout: Meshes With Reconfigurable Buses, Proc. of the MIT Conference on Advanced Research in VLSI, 1988.
- [Nassimi, 1979] D. Nassimi and S. Sahni: Bitonic Sort on a Mesh-Connected Parallel Computer, IEEE Trans. on Comp., 28, 1979.
- [Nassimi, 1980] D. Nassimi and S. Sahni: An Optimal Routing Algorithm for Mesh-Connected Parallel Computers, J. of the ACM, 27. 1980.

[Preston, 1986] Preston, Kendall Jr., Benchmark Results: The Abingdon Cross, in Evaluation of Multicomputers for Image Processing, L. Uhr, K. Preston, S. Levialdi, and M.J.B. Duff, Eds., Academic Press, Orlando, FL, 1986, pp. 23-54

[Rana, 1988] Rana, D., Weems, C.C., and Levitan, S.P., "An easily reconfigurable circuit switched connection network", Proc 1988 IEEE Int Symp on Circuits and Syst, June 1988, pp 247 - 250.

[Rana, 1990] Rana, D., Weems, C.C., A Feedback Concentrator for the Image Understanding Architecture, Proc. 1990 IEEE Intl. Conf. on Pattern Recognition, Atlantic City, NJ, June 1990.

[Rosenfeld, 1987] Rosenfeld, Azriel R., A Report on the DARPA Image Understanding Architectures Workshop, Proceedings of the 1987 DARPA Image Understanding Workshop, February 1987, Los Angeles, CA, Morgan Kaufmann Publishers, Los Altos, CA, 1987, pp. 298-302

[Scudder, 1990] Scudder, M., Weems, C.C., An Apply Compiler for the CAAPP, COINS TR #90-60, University of Massachusetts, 1990.

[Thompson, 1977] C.D. Thompson and H.T. Kung:Sorting on a Mesh Connected Computer, Comm. of the ACM, 20 (4), 1977.

[Valiant, 1981] L.G. Valiant and G.J. Brebner: Universal Schemes for Parallel Computation, 13th ACM Symp. on the Theory of Computing, 1981.

[Weems, 1984] Weems, C. C., Image Processing on a Content Addressable Array Parallel Processor}, Ph.D. Dissertation Computer and Information Science Department, University of Massachusetts at Amherst, September 1984.

[Weems, 1988] Weems, Charles C., Allen Hanson, Edward Riseman, and Azriel Rosenfeld, An Integrated Image Understanding Benchmark: Recognition of a 2 1/2 D "Mobile", Proceedings of the 1988 DARPA Image Understanding Workshop, March, 1988, Cambridge, MA, Morgan Kaufmann Publishers, Los Altos, CA, 1988, pp.

[Weems, 1989] Weems, Charles C., Steven P. Levitan, Allen R. Hanson, Edward M. Riseman, David B. Shu, and J. Gregory Nash, The Image Understanding Architecture, International Journal of Computer Vision, Vol. 2, Kluwer Academic Publishing, Boston, MA, 1989, pp. 251-282.

[Weems, 1990a] Weems, C.C., Rana, D., Reconfiguration in the Low and Intermediate Levels of the Image Understanding Architecture, in Reconfigurable SIMD Parallel Processors, Hungwen Li (ed.), Prentice Hall, Englewood Cliffs, N.J., 1990. Also, COINS TR# 90-10.

[Weems, 1990b] Weems, C.C., Rana, D, Shu, D.B., Nash, J.G., A Progress Report on the Development of the Image Understanding Architecture, Proc. IEEE Intl. Conf. on Pattern Recognition, Atlantic City, NJ, June, 1990.

[Weems, 1990c] C.C. Weems and J.R. Burrill: The Image Understanding Architecture and its Programming Environment, in Parallel Architectures and Algorithms for Image Understanding, V.K. Prasanna Kumar, ed. Academic Press, Orlando, Florida, 1990.

[Weems, 1991] Weems, C.C., Riseman, E.M., Hanson, A.R., Rosenfeld, A., The DARPA Image Understanding Benchmark for Parallel Computers, Journal of Parallel and Distributed Computing, To Appear.

Appendix: Polymorphic Multiple-Processor Networks

A Ph.D. dissertation by Deepak Rana

Completed September, 1991

**Department of Electrical and Computer Engineering
University of Massachusetts at Amherst**

Based on research supported in part by DARPA Contracts DACA76-86-C-0015 and DACA76-89-C-0016, monitored by the U.S. Army Topographic Engineering Center.

POLYMORPHIC MULTIPLE-PROCESSOR NETWORKS

A Dissertation Presented

by

DEEPAK RANA

**Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of**

DOCTOR OF PHILOSOPHY

September 1991

Department of Electrical & Computer Engineering

©Copyright by Deepak Rana 1991
All Rights Reserved

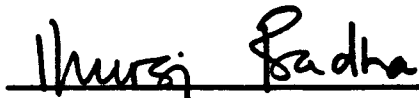
POLYMORPHIC MULTIPLE-PROCESSOR NETWORKS

A Dissertation Presented

by

DEEPAK RANA

Approved as to style and content:



Dhiraj K. Pradhan, Chair



Charles C. Weems, Member



Steven P. Levitan, Member



Maciej J. Ciesielski, Member



Lewis E. Franks, Department Head

Department of Electrical & Computer Engineering

To My Family.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Professor Charles Weems. His guidance, suggestions, and endless editorial help were crucial to this work. I would like to thank Professor Dhiraj Pradhan for his constant encouragement and support. Thanks to Professor Maciek Ciesielski for his comments. I am particularly thankful to Professor Steve Levitan for his invaluable advise during my graduate studies.

I would like to thank Professors Ed Riseman and Al Hanson for their moral and financial support.

Thanks to the VISIONS staff, in particular Bob Heller, Janet Turnbull, and Laurie Waskiewicz for all the help. Special thanks to Mike Rudenko; my close friend at UMass, for his sympathetic ear and advise.

Last but not least, I would like to thank my wife Aud and my son Jay for their love and support.

ABSTRACT

POLYMORPHIC MULTIPLE-PROCESSOR NETWORKS

September 1991

Deepak Rana, B. S., Delhi College of Engineering

M. S., Ph.D., University of Massachusetts

Directed by: Professors Dhiraj K. Pradhan and Charles C. Weems

Many existing multiple-processor architectures are designed to efficiently exploit parallelism in a specific narrow range, where the extremes are fine-grained data parallelism and coarse-grained control parallelism. Most real world problems are comprised of multiple tasks which vary in their range of parallelism. In order to be more effective, future multiple-processor architectures must be "flexible" in supporting multiple forms of parallelism. Machine vision in general, and intermediate-level vision in particular, is an excellent example for demonstrating multi-modal parallelism, which is chosen as the application domain for this thesis.

This thesis addresses issues related to communication in "flexible" multiple-processor systems. The specific problem addressed is to determine the communication requirements of the intermediate level (ICAP) processors of the Image Understanding Architecture (IUA), explore the design space of potential solutions, develop a network design that meets the requirements, demonstrate the feasibility of constructing the design, and show both analytically and empirically that the design meets the requirements.

The approach is to first investigate the computational characteristics of the vision tasks to be run at the ICAP level. The communication and control requirements of the ICAP router are extracted from the computational characteristics. These requirements are divided into logical groups, and an evolving series of network architectures is developed that cumulatively support or address these groups. A number of custom VLSI chips are designed, and analytical study of the networks is carried out to demonstrate the feasibility of their construction.

The major contributions of this thesis are: (1) Crossbars and other dense networks are viable design alternatives even for large parallel processors (2) Central control is viable for reasonably large network sizes, which is contrary to conventional wisdom (3) It is shown that by using a special search memory to implement part of the Clos and Benes network routing algorithm in hardware, it is feasible to quickly reconfigure these networks so that they may be used in fine-grained, data-dependent communication (4) The feasibility of constructing easily reconfigurable communication networks for "flexible" multiple-processor systems is shown. These networks can quickly reconfigure their topologies to best suit a particular algorithm, can be controlled efficiently (in SIMD as well as MIMD mode), and can efficiently route messages (especially with low overhead in SIMD mode) (5) During the course of this investigation it was discovered that, flexible communication as well as shared memory support is much more critical for supporting intermediate-level vision than providing a variety of fixed communication patterns. This observation may also have implications for general-purpose parallel processing, and (6) It was also discovered that supporting a symbolic token database at the intermediate level is a more fundamental requirement than supporting particular algorithms.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	v
ABSTRACT	vi
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER	
1. INTRODUCTION	1
1.1 Machine vision	2
1.2 The Image Understanding Architecture	4
1.3 Problem statement	6
1.4 Our Approach	8
1.5 Major contributions	12
1.6 Outline	13
2. BACKGROUND AND RELATED WORK	16
2.1 Overview of parallel processing	16
2.1.1 Architectural Classification	16
2.1.2 Our terminology for identifying Parallel Systems	24
2.2 Overview of interconnection networks	25
2.2.1 Static interconnection networks	26
2.2.2 Dynamic interconnection networks	30
3. ARCHITECTURAL REQUIREMENTS OF INTERMEDIATE-LEVEL VISION	42
3.1 Low-level vision	43
3.1.1 Histogram-Based Region Segmentation	44
3.1.2 Straight-line extraction by gradient orientation	48

3.1.3 Straight-line extraction by edge grouping	50
3.1.4 Other low-level algorithms	50
3.2 High-level vision	52
3.2.1 The Schema system	52
3.2.2 Knowledge Sources	57
3.3 Intermediate-level Symbolic Representation (ISR) Database	64
3.3.1 Intermediate Symbolic Representation (ISR)	64
3.4 Architectural characteristics of the ICAP	69
3.4.1 As an attached processor to the CAAPP	70
3.4.2 As an attached processor to the SPA	72
3.4.3 Intermediate-level vision tasks	73
3.5 Requirements of the ICAP communication network	77
4. GENERATION 1.0: CENTRAL ROUTING CONTROL	78
4.1 Parallel Communication Switch I	79
4.2 IUA GEN I ICAP communication network	84
4.2.1 ICAP communication network architecture	84
4.2.2 Network setup and re-switching	84
4.3 Larger crossbar networks	87
4.3.1 Network architecture	87
4.3.2 Network setup and re-switching	89
4.4 Larger non-blocking networks	91
4.4.1 Network architecture	92
4.4.2 Network setup and re-switching	95
4.5 Larger rearrangeably non-blocking networks	98
4.5.1 Network architecture	98
4.5.2 Network setup and re-switching	101
4.6 Analysis of PARCOS I	101

4.6.1 Hardware Cost	102
4.6.2 Pinout requirements	107
4.6.3 Power dissipation	108
4.6.4 Time to set up and reconfigure the crossbar	110
4.6.5 Latency and throughput of PARCOS I	110
4.7 Comparison of various networks	119
4.7.1 Hardware cost	119
4.7.2 Time to set up and reconfigure	126
4.7.3 Latency and throughput	126
4.8 Conclusions	127
4.8.1 Goals achieved	128
4.8.2 Goals remaining	128
5. GENERATION 1.5: CENTRAL ROUTING CONTROL	129
5.1 The IUA feedback concentrator	130
5.1.1 The IUA Feedback Concentrator Architecture	131
5.1.2 The feedback concentrator building-block chip	134
5.1.3 The IUA feedback concentrator operation	139
5.1.4 Sample algorithms	139
5.1.5 Second generation IUA feedback concentrator	142
5.2 Data dependent synchronous communication	145
5.2.1 Crossbar network	145
5.2.2 Non-blocking network	151
5.3 Data dependent asynchronous communication	158
5.3.1 Asynchronous crossbar network	158
5.3.2 Asynchronous non-blocking network	162
5.4 Multicast communication	164
5.4.1 Asynchronous communication	165
5.4.2 Synchronous communication	167
5.5 Analysis and comparison of networks	168

5.5.1 Hardware cost	168
5.5.2 Time to set up and reconfigure	170
5.6 Conclusions	172
5.6.1 Goals achieved	172
5.6.2 Remaining goals	173
6. GENERATION 2: DISTRIBUTED ROUTING CONTROL	174
6.1 PARCOS II	174
6.2 Changes in the intermediate level	179
6.2.1 Architectural requirements	181
6.2.2 Design constraints	182
6.3 IUA GEN II communication network	185
6.3.1 Some TMS320C30 features	186
6.3.2 Design proposed by Hughes Research Laboratories	193
6.3.3 Stage I design	197
6.3.4 Stage II design	199
6.4 IUA GEN II+ communication network	206
6.5 Larger networks	216
6.5.1 Based on point to point topology	216
6.5.2 Based on multiple buses	217
6.5.3 Hybrid scheme	217
6.6 Summary	217
7. RESULTS AND CONCLUSIONS	221
7.1 Summary of research	221
7.2 Future research	224
7.2.1 High performance systems	225
7.2.2 Application specific parallel architectures and algorithms	227
7.3 Conclusions	228
BIBLIOGRAPHY	230

LIST OF TABLES

Table	Page
1.1 Taxonomy of Communication Networks	11
3.1 Basic image processing operations	51
3.2 Characteristics and requirements of the intermediate level	76
4.1 Various networks built out of 8×8 switch	121
4.2 Various networks built out of 16×16 switch	122
4.3 Various networks built out of 32×32 switch	123
4.4 Various networks built out of 64×64 switch	123
4.5 Various networks built out of 128×128 switch	124
4.6 Various networks built out of 256×256 switch	124
4.7 Various networks built out of 512×512 switch	126
5.1 Number of cycles for routing	170
6.1 Taxonomy of Communication Networks	179
6.2 Interlock Operations	190
6.3 Best times for shared memory operations	216

LIST OF FIGURES

Figure	Page
1.1 A block diagram of the IUA	5
1.2 Road map of the thesis	14
2.1 Flynn's classification scheme	17
2.2 PE - PE organization of a SIMD computer	20
2.3 PE-Memory organization of a SIMD computer	21
2.4 Organization of switched systems	23
2.5 Some static network topologies	27
2.6 Shared bus organization	31
2.7 Crossbar network	32
2.8 An 8×8 Omega network	35
2.9 An 8×8 Data Manipulator Network	36
2.10 Centrally controlled asynchronous network	40
3.1 Organization of Region Segmentation	45
3.2 House scene part-of network	53
3.3 Overview of VISIONS system components	55
3.4 IUA and the VISIONS system components	56
3.5 Frame and token hierarchy in ISR	66
4.1 Organization of the PARCOS I chip	80
4.2 Organization of a multiplexer tree	81
4.3 Microphotograph of the PARCOS I chip	83
4.4 IUA GEN I ICAP Communication Network	85
4.5 Building larger crossbar networks	88
4.6 A 3-Stage Clos network	93
4.7 A 3-Stage 512-input 512-output Clos network	94

4.8	A 4-Stage modified Clos network	96
4.9	A 5-Stage Clos network	97
4.10	A 3-Stage Benes network	99
4.11	A 2×2 connector based reconfigurable network	100
4.12	One half of the MUX tree	103
4.13	Worst delay path in PARCOS I	111
4.14	Driver for the selector trees	113
4.15	Selector tree and buffer	116
4.16	Equivalent circuit for a selector tree	117
4.17	Normalized board area vs switch size	125
5.1	Node Some/None and Count network	133
5.2	Motherboard Some/None and Count Networks	135
5.3	Global Some/None and Count Networks	136
5.4	Schematic of the Concentrator Chip	137
5.5	Microphotograph of the concentrator chip	138
5.6	Schematic of the daughterboard concentrator chip	143
5.7	Schematic of the motherboard Concentrator Chip	144
5.8	First synchronous crossbar network	146
5.9	Second synchronous crossbar network	148
5.10	Synchronous non-blocking network	152
5.11	Hardware for finding MSS	156
5.12	Asynchronous Crossbar network	159
5.13	Asynchronous non-blocking network	163
5.14	Counter-example	166
6.1	Block diagram of PARCOS II	176
6.2	Block diagram of one cell	177
6.3	Checkplot of one cell	180
6.4	Schematic of ICAP communication network	183
6.5	TMS320C30 Block Diagram	187
6.6	Serial channels in TMS320C30	188
6.7	DMA Controller	189

6.8 Multiple TMS320C30 sharing global memory	192
6.9 HRL SNODE structure	195
6.10 16-Node Dual-Hypercube	196
6.11 SNODE on a motherboard	200
6.12 Block diagram of PARCOS III	202
6.13 SNODE for shared memory	208
6.14 Block diagram of PARCOS III+	209
6.15 Multiple bus based scheme	218
6.16 Hybrid scheme	219

CHAPTER 1

INTRODUCTION

As the fundamental limits of signal speeds in uniprocessor systems are being approached, the search for computationally efficient multiple-processor or parallel architectures has become increasingly important. The availability of more sophisticated and reliable hardware in recent years has enabled many new developments in this area.

Many existing multiple-processor architectures are designed to efficiently exploit a specific form of parallelism, where the extremes are fine-grained data parallelism and coarse-grained control parallelism. Real-world problems often comprise multiple tasks with varying forms of parallelism. One motivating factor of this thesis is that, in order to be more effective, future multiple-processor architectures will have to be "flexible" in supporting multiple forms of parallelism. Machine vision in general, and intermediate-level vision in particular, is an excellent example for demonstrating multi-modal parallelism, which is chosen as the application domain for this thesis.

Data communication is considered by many to be the key bottleneck to successful exploitation of parallelism. To alleviate this bottleneck, a "good" interconnection network for PE-PE communication or for PE-Memory communication is required. We define a "good" communication network as one that meets the performance requirements of a given application while also satisfying its engineering constraints. As such, a communication network architecture is intimately tied to the overall multiple-processor architecture (such as control, mode of operation etc.) and a measure of "goodness" can only be defined in relation to the context of its use.

The objective of this thesis is to explore the suitability and feasibility of construction of VLSI-based, easily reconfigurable, communication networks for "flexible" multiple-processor systems outlined above and in specific, for intermediate-level vision as the application.

The outline of the rest of this chapter is as follows. In the next section we provide a brief motivation of what machine vision is and why it needs parallelism at different levels, followed by an overview of the Image Understanding Architecture, which is a particular hardware solution for machine vision. In section 1.3 we state the problem addressed in

this thesis followed by an outline of our approach for solving it. Major contributions of the research are outlined in section 1.5 followed by an outline of the rest of the chapters.

1.1 Machine vision

This section provides a brief motivation of what machine vision is, and why it needs parallelism at different levels. More details can be found in [Hanson 86, 87; Weems 84,89,91].

Computer vision deals with extracting information about a scene by analyzing images of that scene. It has many applications, in areas such as document processing, remote sensing, radiology, microscopy, industrial inspection, and robot guidance. These applications, and the challenge to process a vast amount of data using a diverse set of complex operations, have long tantalized researchers to build a *vision machine*.

Machine vision is regarded as one of the most computationally intractable problems, requiring a very broad spectrum of techniques ranging from signal processing to knowledge-based symbolic computing in artificial intelligence. A typical scenario with video input might require that an interpretation of a changing scene be updated as video frames arrive at 30 frames per second. Each video frame might contain about three-quarters of a million color-intensity pixels. Many researchers believe that it takes 1,000 to 10,000 operations on each pixel to interpret the whole image. This would require on the order of $10^{10} - 10^{11}$ instructions per second just to keep up with the input. Of course, based on the task, not all operations have to be repeated on every frame, and some vision tasks may not require such frequent updating of an interpretation. Nevertheless, the enormity of computational requirements of a *vision machine* should be apparent.

One goal of machine vision is the construction of a symbolic description of the environment depicted in an image. Such an interpretation involves not only labeling certain regions in an image, or locating a single object in the viewed scene, but often requires the construction of three-dimensional models of the surroundings, with associated identification in the image of the two-dimensional projections of these models (see [Hanson 86] as one example). There are generally three levels of computational abstraction required, with two of these levels obvious: Processing of sensory data and processing of world knowledge. The necessity of an intermediate level of processing has been motivated in [Boldt 87; Draper 89; Weems 91] among others.

Because of the inherent ambiguities that are present in images of natural scenes, it is rarely possible to construct an interpretation directly from the pixel data with classical

image processing techniques such as contrast enhancement, and computer vision techniques of edge detection, region segmentation, and feature extraction. Additional knowledge must be used to reduce local ambiguities and to infer portions of objects that are missing in an image due to effects such as occlusion or shadows. Inference via stored knowledge and the reduction of ambiguity from low-level sensory processing are a part of what is referred to as high-level or knowledge-based vision processing.

The successful functioning of an interpretation system involves hypothesizing scene and object parts from low- and intermediate-level abstractions. These hypotheses are used to access symbolic knowledge structures (called schemas) which capture object descriptions and contextual (relational) constraints derived from prototypical scene situations.

The intermediate level bridges the gap between the low and high levels. The basic unit of information at the intermediate level is a symbolic description of an image event extracted or derived from the image data, and is referred to as a token. Examples of token classes (or token types) are lines, regions, surfaces and, in general, any extractable sensory event that is useful for image interpretation. One class of operations at the intermediate-level may then involve grouping these token events into more complex structures such as rectangles, planes, sets of parallel lines, and textured regions. Another class of operations may involve transforming and projecting 3-D scene independent models from the high-level for 2-D matching with the events extracted from the low-level. The third class of operations at the intermediate-level involves the actual matching of projections of scene independent models from the high-level with the grouped set of tokens derived from the low-level.

The algorithms used in image analysis and understanding are, in general, characterized by potentially massive parallelism. There are several tasks that must be performed on the raw sensory input repeatedly, to construct a symbolic description of the environment depicted in the image. Each of these tasks has a great potential for spatial and temporal parallelism. In general, the degree of exploitable parallelism is high but dynamically variable, based upon the input image and the stage of processing. For example, given a sequence of frames from a video camera, for low-level processing, spatial decomposition of each image frame provides a natural way of generating parallel tasks (spatial parallelism), to be repeated on each frame (temporal parallelism). Further, tasks for low-level processing form a good mix of static and dynamic (input data dependent) algorithms. Therefore, a great deal of parallelism can be exploited, without apriori knowledge of the input image, whereas data dependent parallelism can be exploited only by efficient, dynamic utilization of resources. For high-level analysis operations, parallelism may be based, to a great extent, on input image characteristics.

The Image Understanding Architecture (IUA) is a massively parallel, multi-level system, representing a hardware implementation of the three levels of abstraction discussed above. An overview of the IUA is provided next.

1.2 The Image Understanding Architecture

A massively parallel, multi-level system for supporting real-time image understanding applications and research in knowledge-based computer vision, called the Image Understanding Architecture (IUA) [Weems 89], is currently being developed at the University of Massachusetts at Amherst and Hughes Research Laboratories, Malibu, California. A block diagram of the IUA is shown in figure 1.1. The IUA integrates parallel processors operating simultaneously at three levels of computational granularity in a tightly-coupled architecture. Each level of the IUA is a parallel processor that is different from the other two levels, in order to best meet the processing needs at each of the corresponding levels of abstraction in the image interpretation process. Communication between levels takes place via parallel data and control paths. The processing elements within each level can also communicate with each other in parallel, via a different mechanism at each level that is designed to meet the specific computational, communication, and control needs of each level of abstraction.

The low-level, called the Content Addressable Array Parallel Processor (CAAPP), is a 512×512 array of bit-serial processors intended to perform low-level image processing tasks, on the input image pixels. The CAAPP architecture is especially oriented toward associative processing with an emphasis on fast global summary feedback mechanisms supported in hardware. The CAAPP processing elements are linked through a four-way communication grid that is augmented with a *Coterie Network* that allows certain types of long-distance communication to take place quickly [Herbordt 90]. Currently the CAAPP operates in SIMD mode under the control of a dedicated Array Control Unit (ACU). Studies are underway to support Multi-SIMD processing mode where the CAAPP can be divided into disjoint subgroups, each receiving a different instruction stream.

The intermediate-level, called the Intermediate and Communication Associative Processor (ICAP), is a collection of 4096 fast Digital Signal Processor (DSP) chips. The ICAP is designed for retrieving, comparing, and matching tokens, computing geometric relationships between tokens, and constructing new tokens that describe more abstract entities. For example, the recognition of a house roof in an image may require the ICAP to group together long, straight, parallel lines, and then to extract parallelograms that are candidate

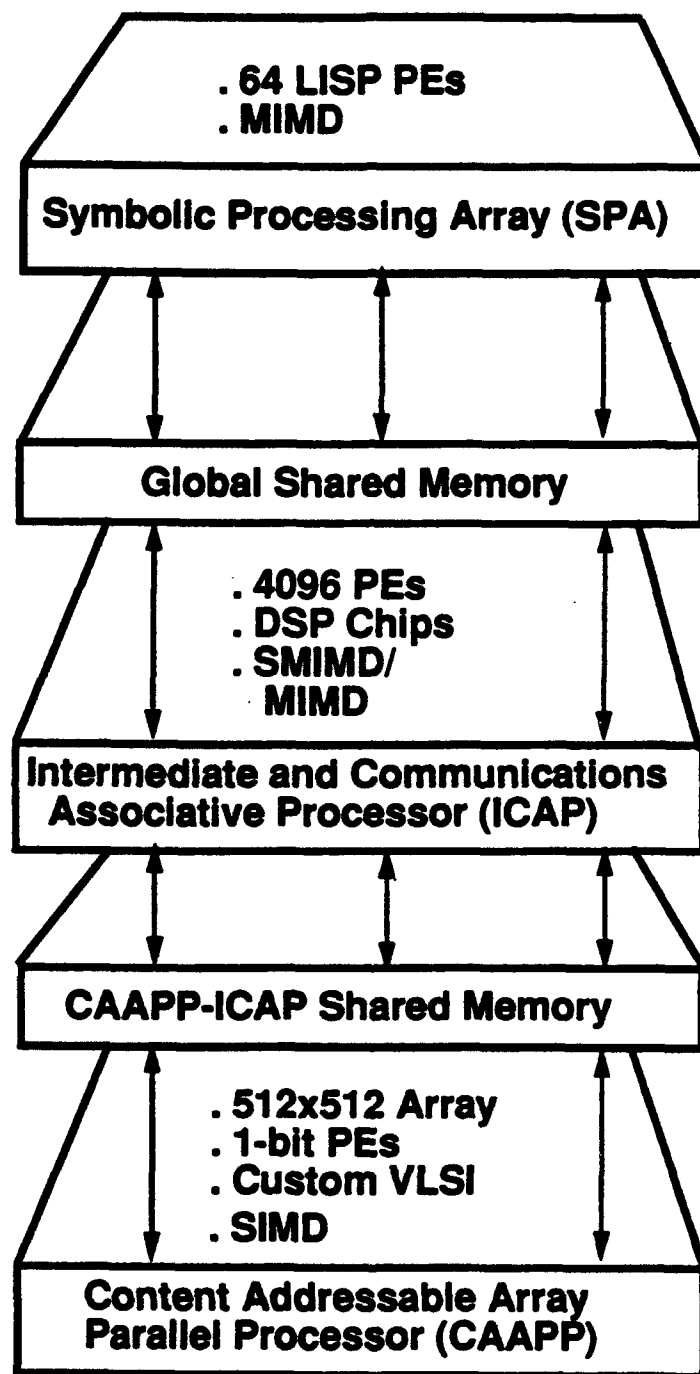


Figure 1.1. A block diagram of the IUA

roof outlines. Should the need arise, the results of further processing in the CAAPP can be integrated with the representation in the ICAP, because the ICAP representation is in approximate registration with the original image events in the CAAPP. The ICAP serves as a database for the symbolic interpretation process running on the high level. Control of the ICAP is provided by the ACU in a synchronous-MIMD mode, and by the high-level in pure MIMD mode. The ICAP communicates with the CAAPP through a CAAPP-ICAP shared memory under the ACU's control. The individual processors in the ICAP communicate with each other through a dynamic interconnection network, that is the subject of this thesis.

At the high-level, called the Symbolic Processing Array (SPA), a set of 64 processors, capable of executing LISP programs, support computation involving inference, hypothesis generation and verification, analysis of uncertainty, model-based processing, and indirect control of processing at lower levels. The SPA processors operate in an MIMD mode. The processors communicate with each other through a large shared memory. The SPA communicates with the ICAP through an ICAP-SPA shared memory. The ACU takes its directions from the SPA.

Construction of a proof-of-concept prototype of 1/64th of the IUA has recently been completed by the University of Massachusetts at Amherst and Hughes Research Laboratories, Malibu, California.

1.3 Problem statement

The problem addressed in this thesis is to determine the communication requirements of the intermediate level processors of the Image Understanding Architecture (IUA), explore the design space of potential solutions, develop a design that meets the requirements, demonstrate the feasibility of constructing the design, and show both analytically and empirically that the design meets the requirements.

Before discussing the characteristics and requirements of the intermediate level processors of the IUA, a brief note is in order. Machine vision is highly dynamic and evolutionary in nature and the development of parallel architectures for machine vision is a nascent field. Therefore, the following requirements are as they were understood at the start of the IUA effort and it was known that they would evolve as the project proceeded.

The general characteristics of communication between the intermediate level processors are:

- Varying communication load
- Varying computational granularity
- Iterative processing with massive temporal parallelism (Cycle through different algorithms repeatedly, using different data sets)
- A mix of static (and known apriori), and dynamic (data-dependent) interprocessor communication, and
- A mix of local, and non-local interprocessor communication

In addition to the above communication characteristics, the following are the control requirements of the ICAP, which directly affect the architecture of the network:

- Centrally controlled SIMD-like processing
- Centrally controlled Synchronous-MIMD (SMIMD)
- MIMD

In other words, both central and distributed control with different granularities of control interaction with respect to data volume.

The aforementioned ICAP modes require further explanation. The ICAP is comprised of TMS320C30 processors. Each C30 has its own program and data. As such, the MIMD mode of operation for the ICAP is self explanatory. Some operations, such as FFT and matrix arithmetic, are characterized by SIMD-like fine grained computation and communication, i.e. data is exchanged between processors after every few computations. To perform these operations efficiently, the ICAP must operate in a tightly synchronous manner (enforced by the central controller) and must have very low overhead interprocessor communication. Essentially, the ICAP is being operated like a SIMD processor. If the communication patterns are known apriori, they can be computed off-line and stored in the communication network, thereby eliminating the cost of setting them up between computation steps. We call this mode of ICAP operation as centrally controlled SIMD-like processing. SMIMD mode of ICAP operation is also characterized by *staged computation* (alternating stages of computation and communication), however, unlike SIMD-like processing, the computational granularity is coarser. Interprocessor communication is data dependent and cannot be known apriori. But periodically, the ICAP processors synchronize under central control, and make requests to communicate with specific processors at the same time. This mode of operation provides some

freedom to the ICAP without incurring the synchronization overheads of MIMD processors. Examples of these modes of ICAP operation will be provided in Chapter 3.

Architectural requirements of the ICAP communication network

Based on the architectural characteristics of the ICAP that are required to efficiently support intermediate-level vision, the architectural requirements of the ICAP communication network are summarized as follows:

- It should have low latency, high bandwidth, and high common access throughput, especially in real-time applications
- It should have the ability to support low-overhead SIMD-like synchronous routing, under central control
- In SIMD-like routing, it should be equally efficient in supporting both regular and irregular communication patterns. In other words, it should not have a bias towards one communication pattern over others
- It should have the ability to support data-dependent synchronous routing under the SMIMD mode of ICAP computation
- It should have the ability to support data-dependent asynchronous routing under the MIMD mode of ICAP computation, and
- Most importantly, it should have the capability of rapid reconfiguration to efficiently support all of these requirements

It should be noted that the above combination of requirements is greater in breadth than those of typical multiprocessors. As pointed out earlier in this section, it was understood that as the IUA project proceeded and the architectural requirements for machine vision became better known, the preceding list could be modified to best suit later IUA designs.

1.4 Our Approach

We follow a three-step approach for solving this problem, in which we develop an evolving series of network architectures that cumulatively support or address specific sets of requirements for the interconnection network.

The following points should be noted about the network design:

- A static interconnection network is not appropriate because such a network is optimal for only one form of interprocessor communication, based on the locality of its connections.
- A fully connected network is only of theoretical interest because the number of I/O ports required on each of the processors for an N processor system is $N - 1$, making their implementation impractical even for moderate size networks.
- A packet-switched network requires the additional overhead of generating packet headers and buffering packets. The communication granularity at the intermediate level is often characterized by frequent, short bursts of small messages between processors. Thus, depending upon the task, a packet-switched network could lower the system throughput to unacceptable levels because of the relatively high overhead. Therefore, as much as possible, the overhead of generating and decoding packet headers should be paid only in data dependent routing.

Once a statically-connected network is ruled out, the network architecture of choice is a *switched-system* or a *multistage network*. Further, to reduce the latency in the network in terms of the number of links and nodes between an input-output pair, we choose moderately large *crossbar-based* nodes (switching elements), instead of the popular 2×2 switches. To reduce the complexity of the network hardware due to packet buffering and forwarding circuitry and reduce its latency, we use a circuit switching scheme. In order to entirely eliminate a bias towards any specific communication pattern, we restrict the design space of potential solutions to networks that are capable of efficiently supporting all N^N possible mappings of their inputs onto their outputs. Such networks realize all $N!$ I/O permutations in one pass, and all N^N I/O mappings in two passes (one pass in crossbar networks). For small networks, it is feasible to implement a crossbar switch built with modular units. However, to implement larger networks, we shall explore strictly non-blocking, and rearrangeably non-blocking topologies, which efficiently support the N^N mappings property.

We should point out that a *Combining network* as proposed in many architectures such as the NYU Ultracomputer [Gottlieb 83], Connection machine [Hillis 85; Tucker 88], Non-Von [Shaw 85], and many Pyramid-based machines, is not suitable for the ICAP. These machines use simple, restricted communication models (Fetch and Add in Ultracomputer, Send with different options in the Connection Machine, Some/None-like associative operations in the Non-Von and Pyramid architectures). On the other hand, the ICAP interprocessor

communication model is not very regular, and the information within messages is not sufficiently uniform to apply the same combining rule across the entire ICAP. For example, a single message might contain addresses and constant values, or different processors may have different information types in their messages. Therefore, we shall not consider the case of a combining network for the ICAP in this thesis. The following are the three designs that we do explore:

Central control, apriori communication

This design addresses the requirements of the communication network when the ICAP is operated in centrally-controlled, SIMD-like manner, i.e. when the interprocessor communication is fixed and known apriori. This occurs in situations where various communication patterns are used repeatedly on different data sets (temporal parallelism).

Networks of this type are built using copies of a building block custom VLSI Parallel Communication Switch (PARCOS I) chip. In addition to an $n \times n$ crossbar, the PARCOS I chip contains a control memory that allows the networks built with the chip to store a large number of network configurations (patterns). With a single instruction, the network configuration can be changed from one stored pattern to another. Any of the stored patterns is incrementally modifiable, without interrupting processing taking place under an existing network configuration. This scheme eliminates header generation and routing (control) overhead for fixed, apriori communication.

Central control, data-dependent communication

This design addresses the requirements of the ICAP when interprocessor communication is data-dependent and cannot be determined apriori, i.e. when the ICAP operates in Synchronous-MIMD (SMIMD) or MIMD mode.

The design permits various networks to be built with simple custom hardware in addition to the PARCOS I chips, to provide an interim solution to the problem of supporting data-dependent interprocessor communication in the ICAP. The networks built with this design use central routing control which, in general, is serial in nature. Therefore, this design is not optimal, but it serves as a stepping stone to the third design.

Distributed routing control, data-dependent communication

This design deals with distributed or parallel routing control in the ICAP communication network. Interprocessor communication is data-dependent and fine-grained, and the ICAP can be operating in either SMIMD or MIMD mode.

The networks under this design use a new custom, VLSI, building-block chip, called PARCOS II, which implements a self-routing crossbar switch and is capable of arbitrating between its inputs in unit time, in order to route them to their respective outputs. Using copies of PARCOS II, various self-routing networks such as Clos, Hypercube etc. can be built.

To place the capabilities of these designs in perspective, we use a table as shown below.

Table 1.1. Taxonomy of Communication Networks

Patterns	Central control		Distributed control	
	Synchronous	Asynchronous	Synchronous	Asynchronous
Computed				
Off-line				
On-line				

Table 1.1 shows a taxonomy of communication networks. The entries in a row correspond to whether a communication pattern (or the switch setting in the network) is computed off-line or on-line. The entries in a column correspond to whether the processors communicate synchronously or asynchronously under the respective form of control. For example, a shared bus with a central arbiter is a case of central control, asynchronous, on-line routing. Batcher's *Bitonic Sorter* [Batcher 68] is an example of distributed control, synchronous, on-line routing. Off-line computation of switch settings in a Benes network, as proposed by [Nassimi 82], is an example of central control, synchronous, off-line routing, and so on. After each of the three design's analyses, we will repeat this table showing which capabilities have been achieved up to that point.

1.5 Major contributions

The major contributions of the research are:

- It will be shown that crossbars and other dense networks are viable design alternatives, even for large parallel processors.
- It is shown that central control is viable for reasonably large network sizes, which is contrary to conventional wisdom.
- By using a special search memory to implement part of the Clos network routing algorithm in hardware, it is shown that it is feasible to quickly reconfigure these networks so that they may be used in fine-grained, data-dependent communication.
- The feasibility of building easily reconfigurable communication networks for "flexible" multiple-processor systems is shown. These networks can quickly reconfiguring their topologies to best suit a particular algorithm, can be controlled efficiently (in SIMD as well as MIMD mode), and can efficiently route messages (especially with low overhead in SIMD mode).
- During the course of this investigation, it was discovered that, flexible communication, as well as shared memory support, is much more critical for supporting intermediate-level vision than providing a variety of fixed communication patterns. This observation may also have implications for general-purpose parallel processing.
- It was also discovered that supporting an Intermediate Symbolic Representation (ISR) database at the ICAP level is a more fundamental requirement than supporting particular intermediate-level vision algorithms.

The following section surveys these contributions and indicates where each is elaborated in subsequent chapters.

1.6 Outline

A "road-map" of the thesis is shown in figure 1.2. In Chapter 2, we provide an overview of parallel processing and interconnection networks. In Chapter 3, we discuss the characteristics and architectural requirements of intermediate-level vision. We investigate the requirements of known tasks in the VISIONS laboratory at the University of Massachusetts, to be run on the IUA, along with predicted requirements based on representative tasks proposed by other researchers in a more abstract form. From this, we summarize a "minimal set" of computational requirements for the processes running on the ICAP. Finally, from these computational requirements, we extract the architectural (communication and control) requirements of the ICAP communication network.

In Chapter 4, we discuss the first stage design: Central routing control with apriori communication. We describe the architecture of the PARCOS I chip in detail, and show how this chip can be used for building various networks. In particular, we show how various sizes of crossbar, non-blocking Clos, and rearrangeably non-blocking Benes networks can be built with PARCOS I. PARCOS I is analyzed in detail with respect to its hardware requirements, pinout requirements, power dissipation, time to set up and reconfigure, and its latency and throughput. Then we compare the various networks that can be built with PARCOS I in terms of their hardware cost, time to set up and reconfigure, and latency and throughput, to show, given various constraints (for example, pinout and board area), which networks are feasible for construction.

In Chapter 5, we discuss the second stage design: Central routing control with data-dependent communication. This design uses a network controller, which is used for setting up links in the network for communication between processors. In addition, the IUA feedback concentrator mechanism (which already exists in the IUA to support other functions) can be usefully employed in this design. We show the design for the building block chip and the optimal architecture for this mechanism in a separate section. Then we describe four networks that can be built using this scheme to support data-dependent routing at the ICAP level. This design has the ability to save an established communication pattern (which is established on-line) in the control memory of the network for later use. Even though our network controller is serial, it can achieve network set up times comparable to many parallel control schemes for non-blocking and rearrangeably non-blocking networks, by using a special hardware search memory. We discuss how to support *multicast* communication in these networks, followed by analysis and evaluation of the overall design.

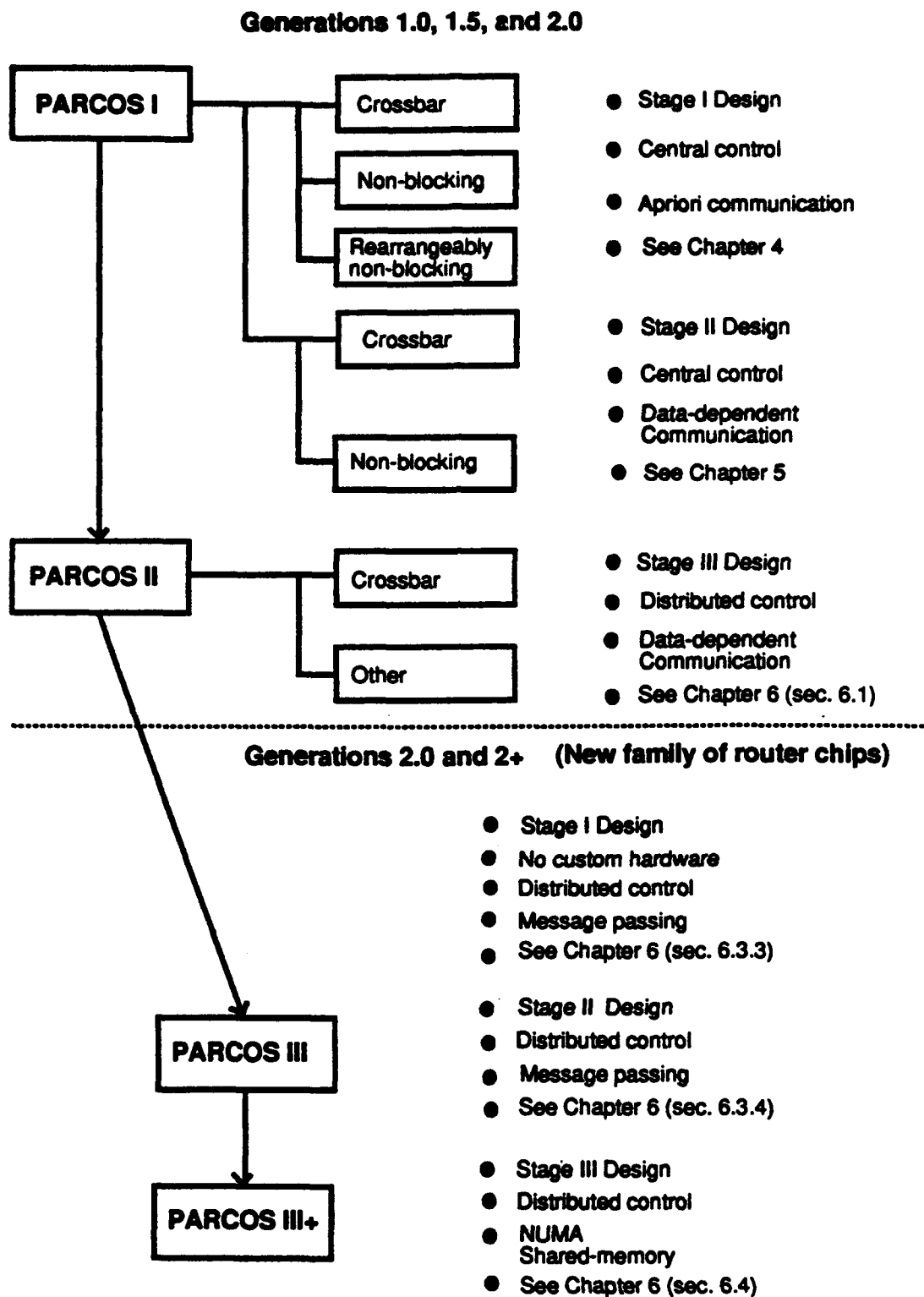


Figure 1.2. Road map of the thesis

In Chapter 6, we discuss the third stage design: Distributed routing control with data-dependent communication. Our original plan was simply to extend the previous designs to include this capability in hardware. A custom VLSI building block chip, called PARCOS II is described in detail, which can be used for building larger self-routing networks. Due to the dynamic and evolutionary nature of machine vision, during the course of this research new intermediate-level vision requirements evolved along with different IUA design constraints. This made it necessary to develop an entirely new network family using a different methodology than that used for the first three stages. The new intermediate-level vision requirements along with the new design constraints are discussed, followed by a description of a series of networks that were designed to address these requirements.

In Chapter 7, we present our conclusions and discuss future directions for the research.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides an overview of parallel processing and interconnection networks. Various schemes have been proposed to classify computer architectures, but none of them covers the entire functional model of the ICAP level of the IUA. In Section 2.1.1 we provide an overview of the three most widely used computer architecture classification schemes. Since none of these schemes cover the ICAP, we define our own terminology for identifying a parallel processor such as the ICAP in Section 2.1.2. A myriad of schemes have been proposed for interconnection networks for PE - PE communication or for PE - Memory communication in parallel processing systems. In Section 2.2 we provide an overview of the field of interconnection networks. We will be using a combination of more than one of these schemes as our approach to ICAP interprocessor communication in the following chapters.

2.1 Overview of parallel processing

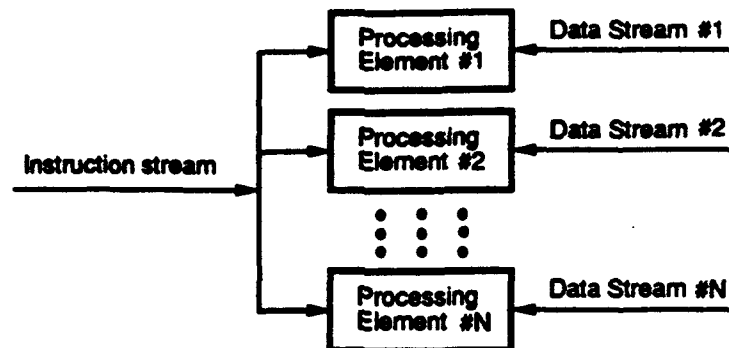
Parallel processing refers to simultaneous or concurrent computing activity on more than one node of a computer system. Parallel computers are those systems that emphasize parallel processing.

2.1.1 Architectural Classification

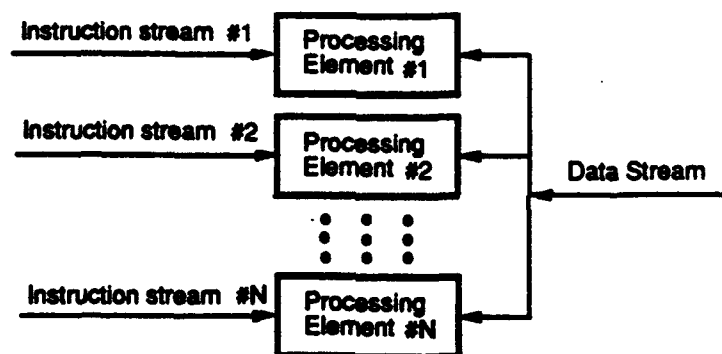
We begin by considering three important classification schemes for parallel processing architectures. Flynn's classification [Flynn 66] is based on the multiplicity of instruction schemes and data streams in a computer system. Feng's scheme [Feng 72] is based on serial versus parallel processing. Handler's classification [Handler 77] is determined by the degree of parallelism and pipelining in various subsystem levels. All three schemes distinguish a serial computer from parallel computers, with Flynn's scheme being the most widely used.



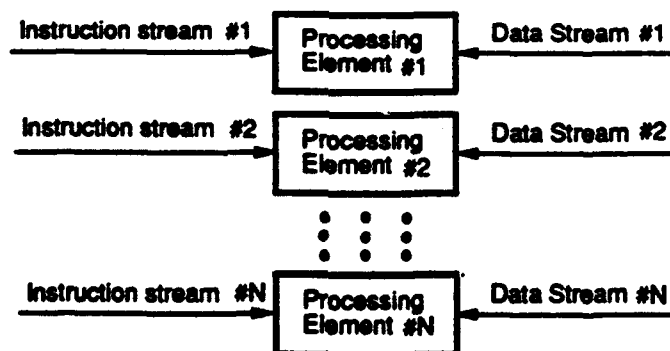
a) SISD computer



b) SIMD computer



c) MISD computer



d) MIMD computer

Figure 2.1. Flynn's classification scheme

Shown in figure 2.1, Flynn's classification scheme treats a computing process as the execution of a sequence of instructions on a set of data and, in general, digital computers may be classified into four categories, according to the multiplicity of *instruction* and *data streams*. An *instruction stream* is a sequence of instructions as executed by the machine; a *data stream* is a sequence of data including input, partial results or temporary data, called for by the instruction stream. Computer organizations are characterized by the multiplicity of the hardware provided to service the instruction and data streams. The following are Flynn's four machine organizations.

- Single instruction stream, single data stream (*SISD*),
- Single instruction stream, multiple data stream (*SIMD*),
- Multiple instruction stream, single data stream (*MISD*), and
- Multiple instruction stream, multiple data stream (*MIMD*).

According to Flynn's classification, *SISD* is the classic serial computer. The remaining three organizations classify parallel computers. The *MISD* organization is the least frequently found, and perhaps the least understood. It seems that there are few cases in which multiple operations can be performed on a single datum with any benefit in a digital system.

Parallel computers are often divided into three architectural configurations [Hwang 84].

- Pipeline computers and Systolic Arrays.
- Array processors and Associative processors.
- Multiprocessor systems.

Pipelining is used to exploit *temporal* parallelism in many computers. The concept of a pipeline computer is similar to a manufacturing assembly line. To achieve pipelining, one must subdivide the input task into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are streamed into the pipe and get executed in an overlapped fashion at the subtask level. Based upon the type of processing, Handler [Handler 77] defines three classes of pipelining: Arithmetic pipelining; Instruction pipelining; and Processor pipelining.

In *arithmetic pipelining* the Arithmetic Logic Unit (ALU) of a computer can be segmented for pipeline operations in various data formats. For example, to add two normalized floating point numbers, a pipelined floating point adder can be designed with four functional stages

[Hwang 79]. Suppose each stage has a time delay of 90nS and the interface latch has a delay of 10nS. If the pipeline can be kept full, one result can be generated every 100nS as compared with 360nS for a non-pipelined floating point adder. Some examples of arithmetic pipelines are the four stage pipes used in Star-100 [Hintz 72], the eight stage pipes used in the TI-ASC [Watson 72], the 14 stage pipeline used in the Cray-1 [Russell 78], and the 26 stage pipes in the Cyber-205 [Hwang 84].

In *instruction pipelining*, a stream of instructions is executed in an overlapped fashion. This technique is also known as *instruction lookahead*. Normally, the process of executing an instruction in a digital computer involves four major steps: instruction fetch (IF) from the main memory; instruction decoding (ID) (identifying the operation to be performed); operand fetch (OF), if needed in the execution; and then execution (EX) of the decoded operation. In a non-pipelined computer, these four steps must be completed before the next instruction can be issued. In an instruction pipeline, the four stages can be arranged into a linear cascade. An *instruction cycle* consists of multiple pipeline cycles. The operation of all stages is synchronized under a common clock control. For a serial computer with a multifunction ALU, it takes four pipeline cycles to complete one instruction. In an ideal case, if one instruction can be fed into the instruction pipeline on every cycle, an output result can be produced from the pipeline on every cycle, thereby increasing the throughput of the machine by a factor of four. A major impediment to instruction pipelining is branches in the code. Many systems are substantially augmented to alleviate this problem. Most modern high performance computer systems incorporate instruction pipelining.

Processor pipelining refers to processing of the same data stream by a cascade of processors, each of which performs a specific task. The data stream passes through the first processor with results stored in a memory block, which is also accessible by the second processor. The second processor then passes the refined results to the third processor, and so on. Systolic Arrays are considered an example of processor pipelining by many researchers.

The SIMD machine organization in Flynn's classification corresponds to the array processor and associative processor configurations of parallel computers in Hwang's classification. A synchronous ensemble of multiple processing elements (PEs) under the supervision of one control unit (CU) is called an array processor. By replication of PEs, one can achieve spatial parallelism. The PEs are synchronized to perform the same function at the same time. A data-routing mechanism may exist among the PEs. Scalar and control-type instructions are directly executed in the CU. The CU broadcasts instructions in parallel to all the PEs in the array. There are two variations to the design of PE arrays. In the first variation, shown in figure 2.2, each PE comprises an ALU with registers and a local memory. The

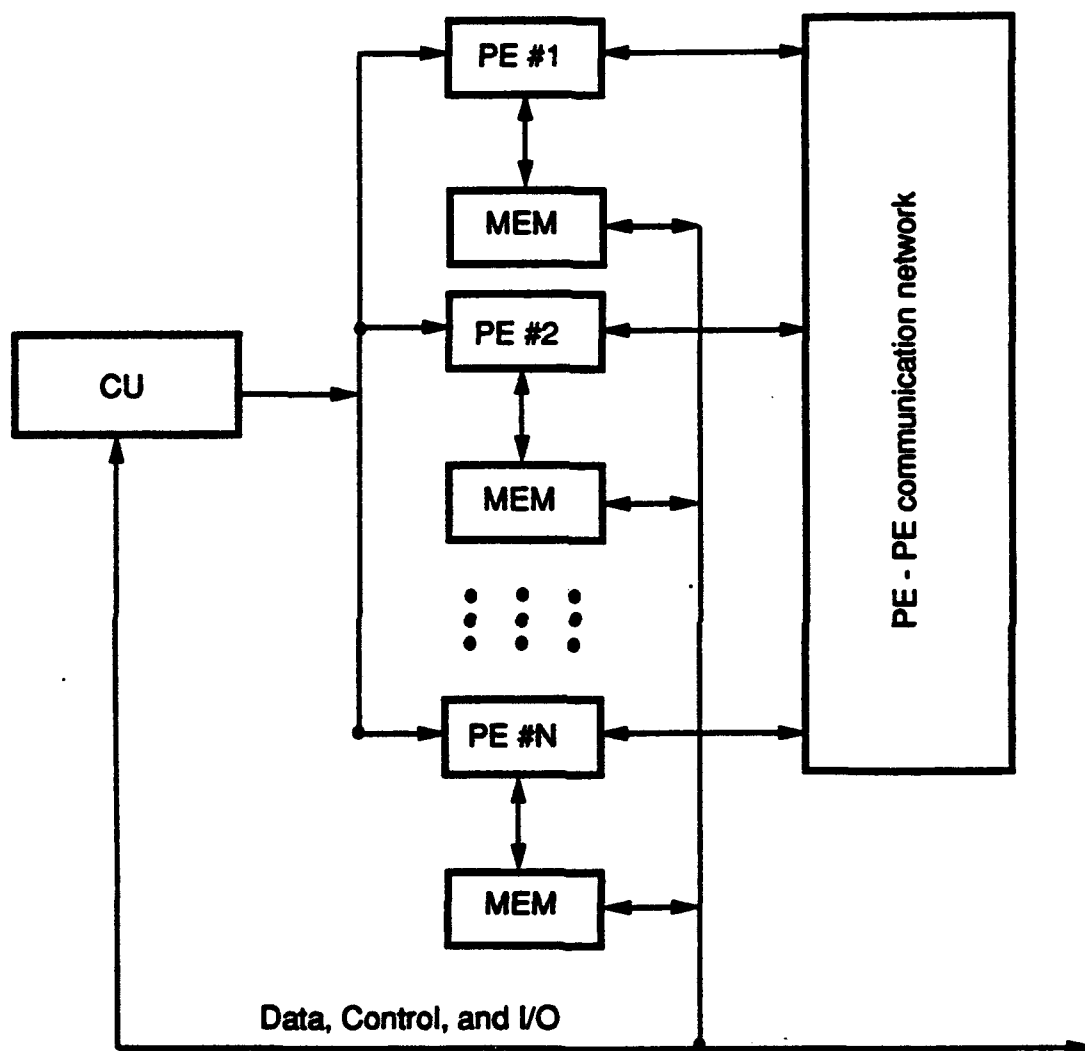


Figure 2.2. PE - PE organization of a SIMD computer

PEs are connected by a data routing network. The interconnection pattern to be established for specific computations is under program control from the CU. Vector instructions are broadcast to the PEs for distributed execution over different component operands fetched directly from the local memories. Instruction fetch from the control memory, and decoding is done by the control unit. The PEs are passive devices without instruction decoding capabilities. In the second variation, shown in figure 2.3, each PE comprises an ALU with

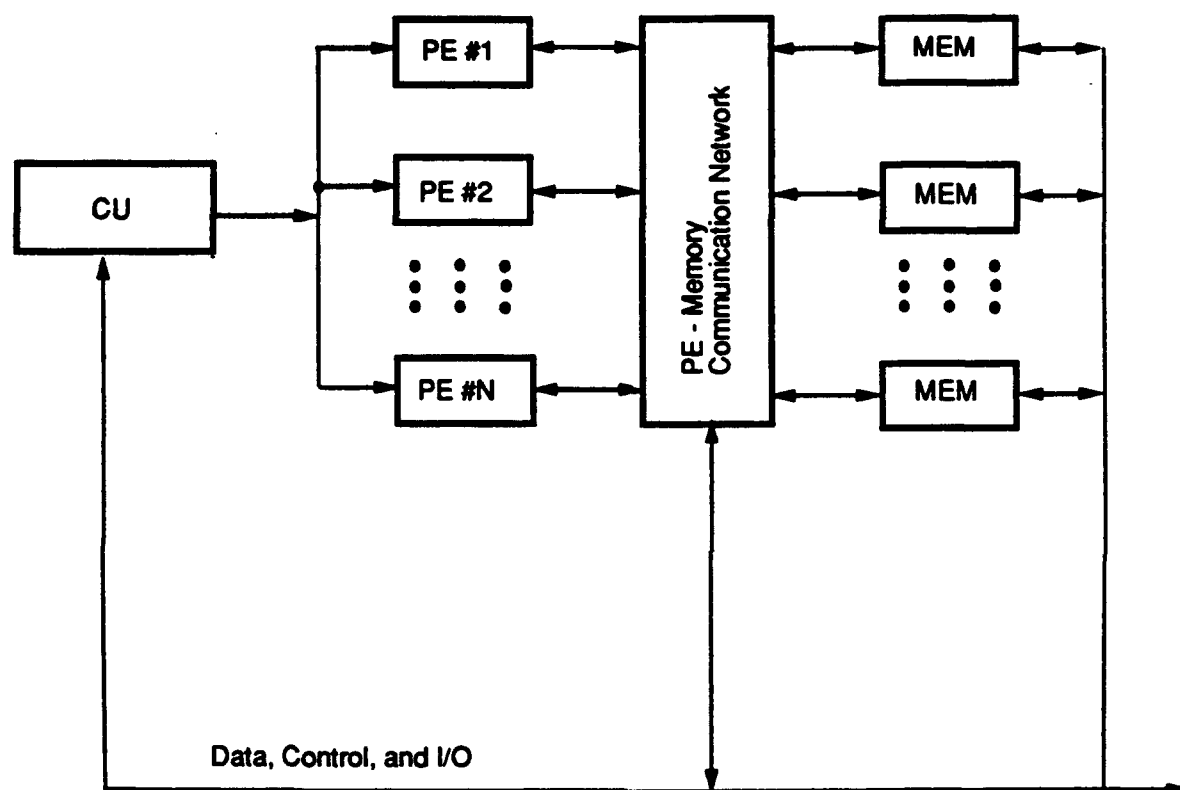


Figure 2.3. PE-Memory organization of a SIMD computer

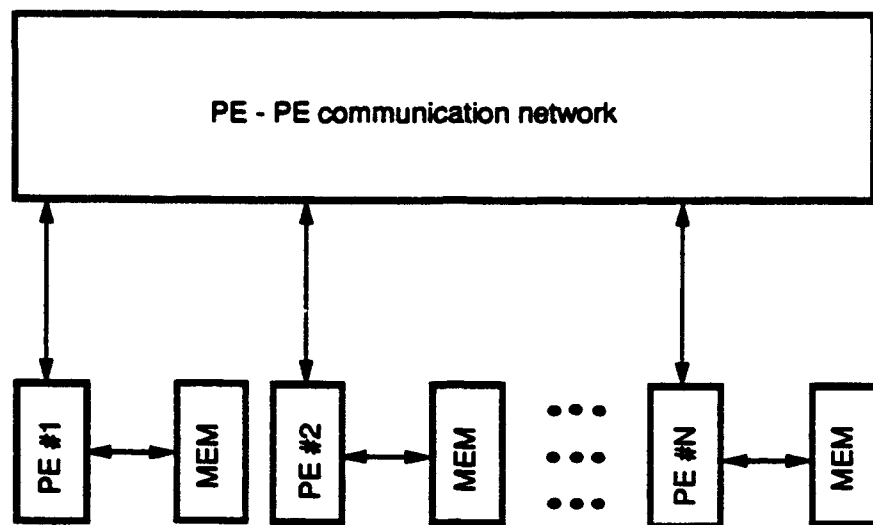
registers, and the local memories attached to the PEs are now replaced by parallel memory modules, shared by all the PEs through a data alignment network. There are N PEs and P memory modules in the second variation. The two numbers are not necessarily equal. The data alignment network is a dynamic path switching network between the PEs and the parallel memories. Such an alignment network is desired to allow conflict free access to the shared memories by as many PEs as possible, and is under program control from the CU. Some examples of array computers in the first category are the Spatial Computer [Unger 58], SOLOMON [Slotnick 62], MPP [Batcher 80], ILLIAC IV [Bouknight 72], Clip-4 [Duff 78], and DAP [Hunt 81]. Some examples of array computers in the second category are Burroughs Scientific Processor (BSP) [Kuck 82], and the Orthogonal Computer [Shooman

60]. A good review of array processing can be found in [Zakharov 84].

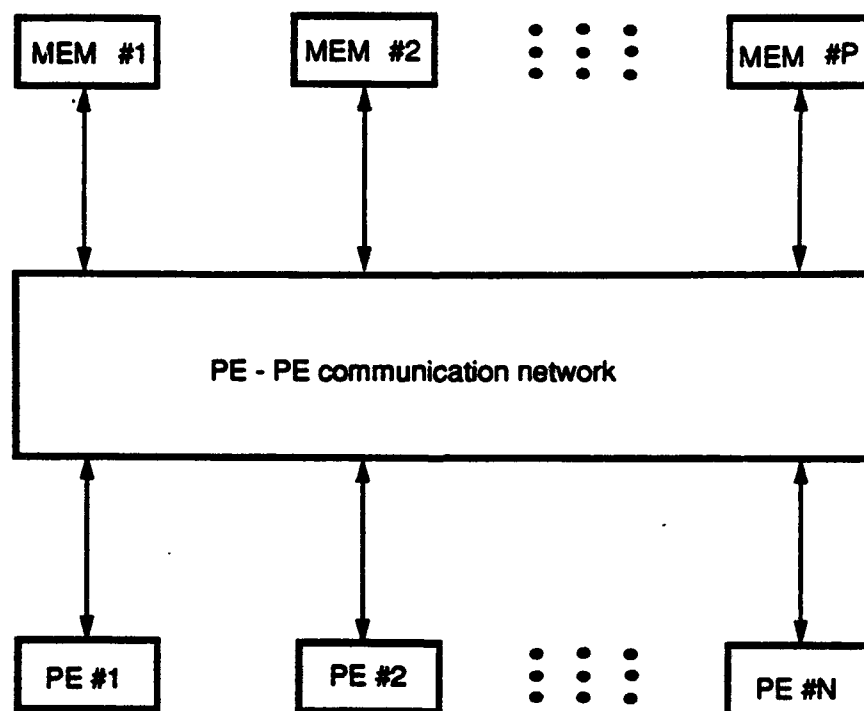
Associative processors are SIMD class parallel processors. Researchers in computer architecture have always distinguished them from the mainstream of parallel processors, which does not seem so much due to differences in hardware organization as to differences in applications. Indeed, many mainstream parallel processor systems are capable of operating as associative processors with some loss of efficiency. The reason that they are not used as such is mostly that associative processing is not the purpose for which they were designed. Associative processors on the other hand, have most often been designed for applications that chiefly use the associative facet of the architecture. Some examples of Associative processors are Staran [Batcher 77], and PEPE [Berg 72]. An overview of associative processing can be found in [Weems 84].

The fourth machine organization in Flynn's classification is MIMD machines. Often such systems are called *multiprocessors*. Multiprocessors are suitable for a broader class of computations than are array computers, because multiprocessors are inherently more flexible. For example, if a computation has no specific vector structure or other natural iterative structure, then it is unlikely to be suitable for a vector or array processor, but it may be suitable for a multiprocessor. A multiprocessor need only have potential parallelism that can be exploited by independent instruction streams. Also, a multiprocessor can handle local conditional branches easily due to the autonomy of the tasks on the processors, whereas an array processor which has only one task running at any time, must handle these local branches serially. Therefore, in many applications, it is relatively easy to fit a computation onto a multiprocessor system. However, to attain high-efficiency computation in a multiprocessor system, one has to solve the problems of task synchronization and task scheduling. Most importantly, the entire system must be controlled by a single integrated operating system providing interactions between processors and their programs at various levels.

There are two major subdivisions of MIMD machines: *Switched-systems* and (statically connected) *Networks* [Hockney 85]. In a Switched-system, shown in figure 2.4, a switch unit (also called communication network) comprising of one or more stages of dynamic switching elements is used to connect together a number of processors and memory modules. Within Switched-systems, there are two subdivisions: *Shared-memory* and *Distributed-memory* systems. In a Shared-memory system, a number of processors, each with its own instruction stream, are connected via the switch unit to a number of independent memory modules. In this way the memory is shared by all the processors through a large common address space or common name space. A small local memory (cache) usually exists in each



Distributed memory



Shared memory

Figure 2.4. Organization of switched systems

processor to alleviate the memory latency problem and to reduce memory access contention. Such systems are also called *Tightly-coupled* multiprocessors [Hwang 84; Stone 80], or *Paracomputers* [Schwartz 80]. Some examples of Switched-systems are the RP3 [Pfister 85], NYU Ultracomputer [Gottlieb 83], TRAC [Sejnowski 80], and BBN Butterfly [Growther 85].

The alternative to a common global shared memory is to attach the memory directly to the processors and produce a Distributed-memory system. In this case each memory module is connected as local memory to a processor. The role of the switch is now to interconnect the processors, and there are no memory modules connected directly to the switch. It should be noted that examples of pure Shared-memory or Distributed-memory Switched-systems are rare. Many of the large Switched-systems use both types of memory.

The second division under MIMD machines is *Network-based* or *Statically-connected* systems. In such systems, a number of processors with their own local memory are connected together using a fixed topology communication network. Such systems are Distributed-memory systems in the sense that all the memory is distributed throughout the system as local memory. The processors communicate via the interprocessor communication network and the communication latency between two processors depends on whether they are locally connected to each other or are connected through one or more layers of the communication network. Such systems are also called *Loosely-coupled* multiprocessors [Hwang 84; Stone 80], *Ultracomputers* [Schwartz 80], or *Multicomputers* [Seitz 85; Athas 88]. Some examples of such systems are the COSMIC CUBE [Seitz 85], Intel Hypercube [Rattner 85], and JPL MARK II [Tuazon 85]. An overview of MIMD computing can be found in [Hockney 85].

2.1.2 Our terminology for identifying Parallel Systems

The functional model of the ICAP covers more than one architectural class in the existing architectural taxonomies. Also, some modes of ICAP operation are not even defined in these taxonomies, such as synchronous-MIMD. As such, none of the following terms can be used to define the ICAP: Array processor, Associative processor, Tightly-coupled multiprocessor or Multiprocessor, Loosely-coupled multiprocessor or Multicomputer, etc. Therefore, in this thesis, we will use the term *Multiple-processor system* to define a parallel system such as the ICAP. We will use the term *Multiple-processor systems* collectively for array processors, associative processors, multiprocessors, and multicomputers. In other words, we will use the term multiple-processor for a parallel processor that comprises multiple PEs, a mechanism for inter-PE communication, or communication between PEs and one or more global shared

memory modules, and a mechanism for issuing instructions to the PEs from one or more sources. We distinguish multiple-processor systems from the general class called parallel processors by the fact that the class called multiple-processor does not include arithmetic pipelining, attached processors such as floating point units, or overlapped instruction execution. The multiple-processors are subsumed by parallel processor, and themselves subsume array processors, associative processors, multiprocessors, and multicomputers, in our architectural classification.

2.2 Overview of interconnection networks

After establishing a general overview of parallel processing, we provide an overview of a critical area of research in parallel processing, and the research area of this thesis: Data communication (interconnection) networks.

Until not long ago, computer system architects were primarily concerned with the computing aspects of parallel processing; the communication aspects such as the volume of data or messages transferred, synchronization etc., received relatively little attention. Lately, however, there has been a growing belief that in multiple-processor systems the communication aspects are at least as important as the computing aspects. Some examples of work on communication can be found in [Cvetanovic 87; Gannon 84; Gentleman 78; Levitan 87; Lint 81]. Data communication is the key to successful exploitation of parallelism. It has been stated that "the most critical system control mechanisms in a distributed computer are clearly those involved with interprocess and interprocessor communication" [Jensen 78].

In light of the above, computer architects have designed data communication or connection networks based upon two broad approaches: Point to point or *static interconnection networks* between multiple PE's; and *dynamic interconnection networks* based upon time-sharing a common link (bus) and/or multiple stages of switching elements to establish communication paths between one or more sources and one or more destinations. The sources and the destinations may either be processors, memories, or a combination of the two. There are examples where a hybrid of these approaches have been used (for example, Cedar [Gajski 83; 86], NETRA [Sharma 85], and PM4 [Briggs 79]).

2.2.1 Static interconnection networks

In multiple-processor systems with a static interconnection topology, the PEs are connected such that each is directly linked to a fixed number of PEs called its neighbors (A PE comprising a processor and some local memory is the most popular arrangement, though other variations in which a PE is either just a processor or a memory are possible). Many multiple-processor architectures with a static interconnection topology have been proposed and some have been built to solve various problems. Among these are the linear arrays [An-nartone 87; Kung 83; Kung 80], the two or more dimensional meshes [Batcher 80; Bouknight 72; Duff 78; Holland 59; Slotnick 62; Thompson 77], the doubly twisted torus [Sequin 81], the binary tree [Horowitz 81], the orthogonal trees [Nath 83], pyramid processors [Bode 85; Hanson 80; Rosenfeld 86; Tanimoto 83; Uhr 72, 87], the hypercube [Hays 86; Hillis 85; Rattner 85; Seitz 85; Tuazon 85; Tucker 88], the Shuffle-Exchange [Stone 71], the Cube Connected Cycles [Preparata 81], the De Bruijn graph [Samatham 89] and the group graphs [Akers 89].

Routing in static interconnection networks

In multiple-processor systems with a static topology, if PE i has to send a message to PE j , it can do so by directly sending it to j , if j is one of its neighbors. Otherwise, the message will have to travel through one or more PEs in a path between i and j . There are two methods of routing messages between non neighbor PEs: *Non-adaptive routing*, and *adaptive routing*. In non-adaptive routing, PE i sends the message to one of its neighbors which in turn sends the message to one of its neighbors and so on, such that at each step the message moves closer to its final destination than at the previous step. In adaptive routing, the load (traffic) conditions on various PEs and links of the network are also taken into consideration, such that a message may not always move closer to its destination, but an attempt is made to minimize the average delay for all the messages in the system. Typically, a message comprises a header, followed by an information packet and a trailer.

Both of the above schemes have their advantages and disadvantages. For example, in a non-adaptive routing scheme, the routing protocol can be very simple such that there is very little overhead to forward a message at an intermediary PE. Also, the routing protocol can be completely distributed, such that any PE needs to have only local information about its input and output links. The chief disadvantage of non-adaptive routing is its inability to

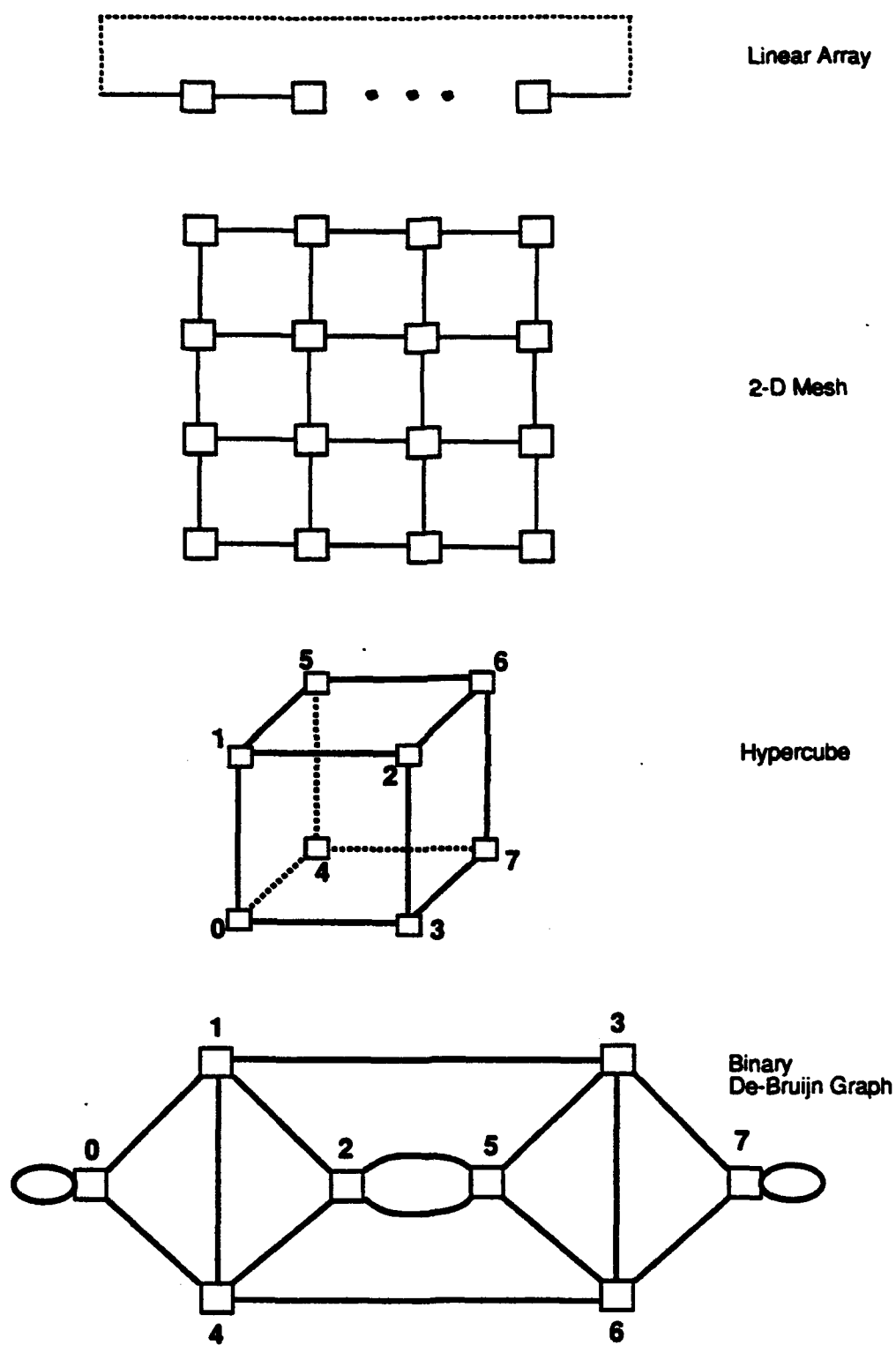


Figure 2.5. Some static network topologies

handle local congestion, which may result in a slowdown of the whole system. On the other hand, adaptive routing can solve the local congestion problem in many cases. However, the price paid in adaptive routing is the increased overhead in the forwarding protocol at the intermediary PEs and the overhead of providing non-local information to each potential intermediary PE at each routing step.

A variation of these schemes is a routing mechanism proposed by Kermani and Kleinrock, called the *Virtual cut-through* [Kermani 79]. In the above mentioned schemes, a message hops as a single unit from one PE to another. Virtual cut-through is similar, with the difference that as soon as the message header arrives at an intermediary PE, if the appropriate outgoing link is free, then transmission of the message to the adjacent PE begins before it is received completely at the intermediary PE. The message is buffered at the intermediary PE, only if its output link is busy. Therefore, in a virtual cut-through scheme, the delay due to unnecessary buffering in front of an idle link is avoided.

A variation of virtual cut-through is a scheme proposed by Dally and Seitz, called *Wormhole routing* [Dally 87; Dally 86]. In Wormhole routing, messages are not buffered as a unit at an intermediary PE when its output link is blocked. Instead, a message comprises a series of words, called *flits*, where each flit is of the same bit width as the physical links that carry them. Therefore, if a header flit gets blocked at an intermediary PE, the flit behind the header flit is buffered at the previous PE and the flit before that is buffered at the one before that and so on until it reaches the source PE, which in turn does not push the next flit into the message path.

Some problems with static interconnection networks

One thing should be clear from the preceding discussion of message routing in statically connected multiple-processor systems: *Locality of communication is highly desirable*. The latency in message transfer will depend upon the distance (the number of links traversed) between two communicating PEs. This could become a serious synchronization and performance bottleneck in staged algorithms where computation in the PEs and the communication in the network cannot be overlapped. Further, depending on the interconnection topology, there can be severe congestion in the network if many PEs must communicate simultaneously with remote PEs.

Some schemes to improve static interconnection network routing

Many researchers have tried to alleviate the problem of non local communication in static interconnection topologies. The basic ideas behind these improvements are taken from the dynamic interconnection networks, which will be covered in later sections. We discuss the case of improvements on a mesh topology, which is one of the popular static topologies. Arguably many static topologies can be considered to have been studied to alleviate communication problems in meshes. Some examples are the pyramid computer [Tanimoto 83] and the mesh-of-trees [Nath 83; Leighton 84]. Other approaches have considered providing one or more broadcast buses, or reconfigurable buses in addition to the static mesh topology. Bokhari and Stout proposed a mesh connected computer with a global bus [Bokhari 84; Stout 83; Stout 86], to speed up some computations. A mesh with multiple broadcast buses is considered in [Kumar 87], where a PE in each row and column of a mesh is connected to a row and a column bus. A polymorphic torus architecture has been proposed [Li 87; Li 89], based on a torus which has a crossbar switch at each PE to make any arbitrary connection between the east, west, north and south bus ports. Weems [Weems 84] and Miller et al. [Miller 87] have both proposed mesh-connected processors with a reconfigurable broadcast bus. A switch is inserted in the bus link between every two neighboring PEs, that can be closed or opened by the PE at each end of the link. These switches allow the broadcast bus to be divided into sub-buses, where each sub-bus can function as a smaller reconfigurable broadcast bus or reconfigurable mesh. The lowest level of processors in the Image Understanding Architecture (IUA) [Weems 89], which consists of a mesh, augmented by the *Coterie Network*, that provides an additional feature of some crossover capability over reconfigurable mesh architectures. The PEs can be clustered into arbitrary non-overlapping coterie where each coterie has its own private bus for broadcasting. If several PEs try to write on this bus, a logical OR of their values gets written onto the bus. Finally, the Configurable Highly Parallel (CHiP) processor [Berman 83; Cuny 84; Hedlund 82; Snyder 82; Snyder 84], consists of a collection of PEs placed at grid points in the plane, and a switch lattice with programmable switches interposed between the processors. The PEs have local storage for program and data. Depending upon the chosen lattice, by closing appropriate switches, different interconnection patterns between the PEs can be obtained. Switches can have local memory to store interconnection patterns that can be controlled by the program running on the machine. Thus, various interconnections between PEs are accomplished by using different stored patterns.

2.2.2 Dynamic interconnection networks

The second broad approach to providing connection networks for multiple-processor systems is based on time-sharing common links between one or more sources and one or more destinations. These connection networks are often called *dynamic interconnection networks*, or simply *dynamic networks*. Two parameters are often used to characterize dynamic networks [Stenstrom 88]. The first parameter is the *bandwidth* of the network, which is a measure of the capacity of the network to fulfill the communication rate of its inputs and outputs. The second is the *common access throughput*, meaning the maximum number of simultaneous requests that can pass through the interconnection network at the same time. There are three major configurations of dynamic networks: *Shared bus*, *Crossbar switch*, and *Multistage network*.

Shared bus interconnection networks

Due to its simplicity, the *shared bus* is perhaps the most popular dynamic network. Some examples of shared bus organization are shown in figure 2.6. A shared bus has a fixed bandwidth, and the common access throughput for a bus is only equal to one; it can handle only one request at a time. The shared bus can be used for either PE - PE or for PE - Memory communication. In commercial systems, the shared bus has primarily been used in tightly coupled or shared memory systems, for PEs to access shared memory modules.

A bus arbiter selects one of several requesting PEs to get exclusive access to the shared bus. An obvious way to increase the bandwidth and the common access throughput is to use several parallel, independent buses capable of handling several memory requests at the same time.

Typically, the bus latency time (or bus cycle time) is shorter than the memory access time. To take full advantage of the bandwidth of the bus, designers often implement a bus protocol that allows pipelining of several memory requests. While one memory module executes a load, another processor can send a memory request to another memory module. Examples of shared bus multiple-processor systems are; Sequent Balance [Thakkar 88], Encore Multimax [Schanin 86], Alliant FX/Series [Perron 86], CM* [Swan 77], SPUR [Hill 86], and Dragon [Monier 85].

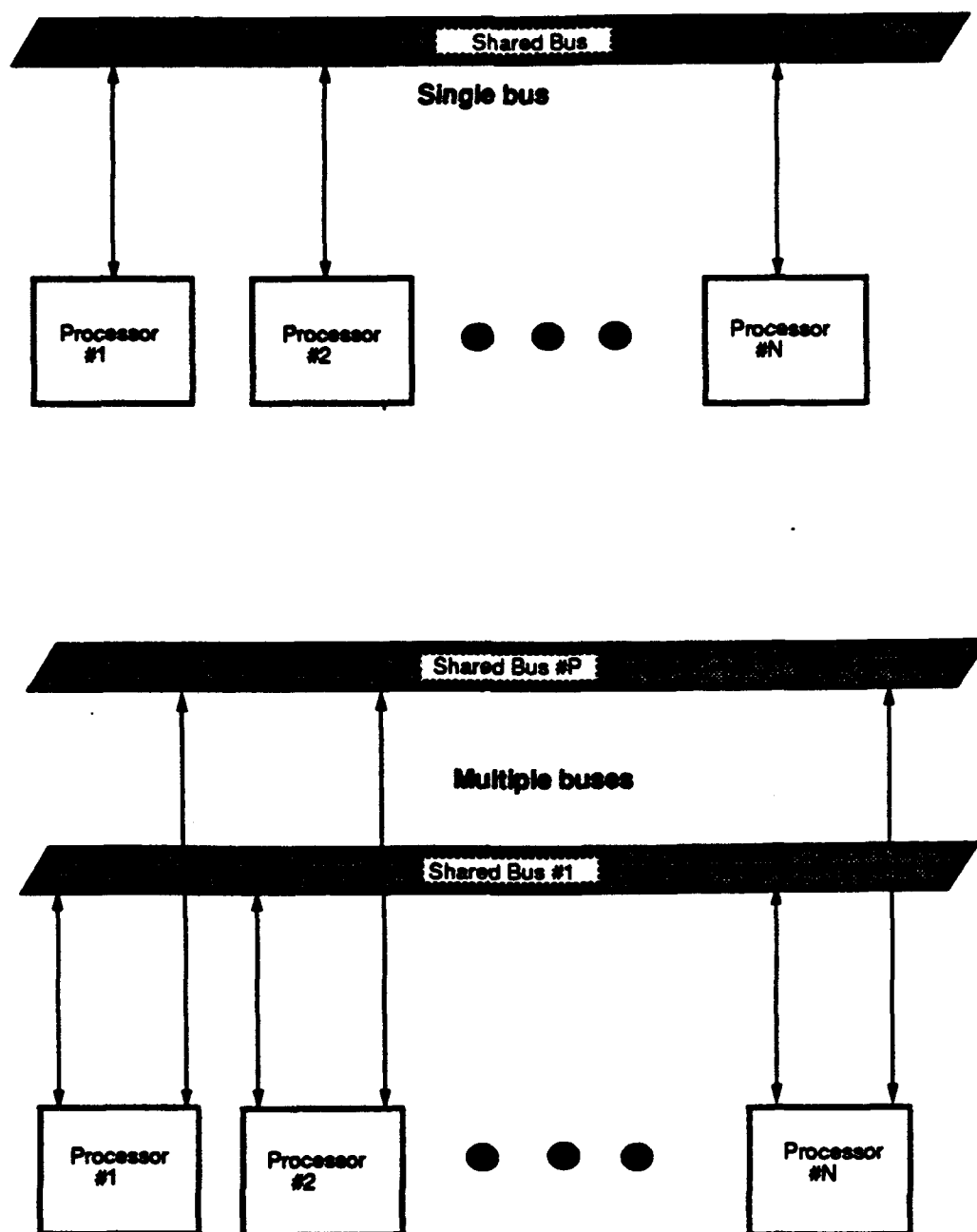


Figure 2.6. Shared bus organization

Crossbar interconnection network

A second configuration of dynamic networks is a crossbar switch. A crossbar switch can be viewed as a number of vertical and horizontal links interconnected by a switch at each intersection. The number of vertical and horizontal links equals the number of processors and memory modules, respectively in a shared memory multiple-processor. A crossbar network is shown in figure 2.7.

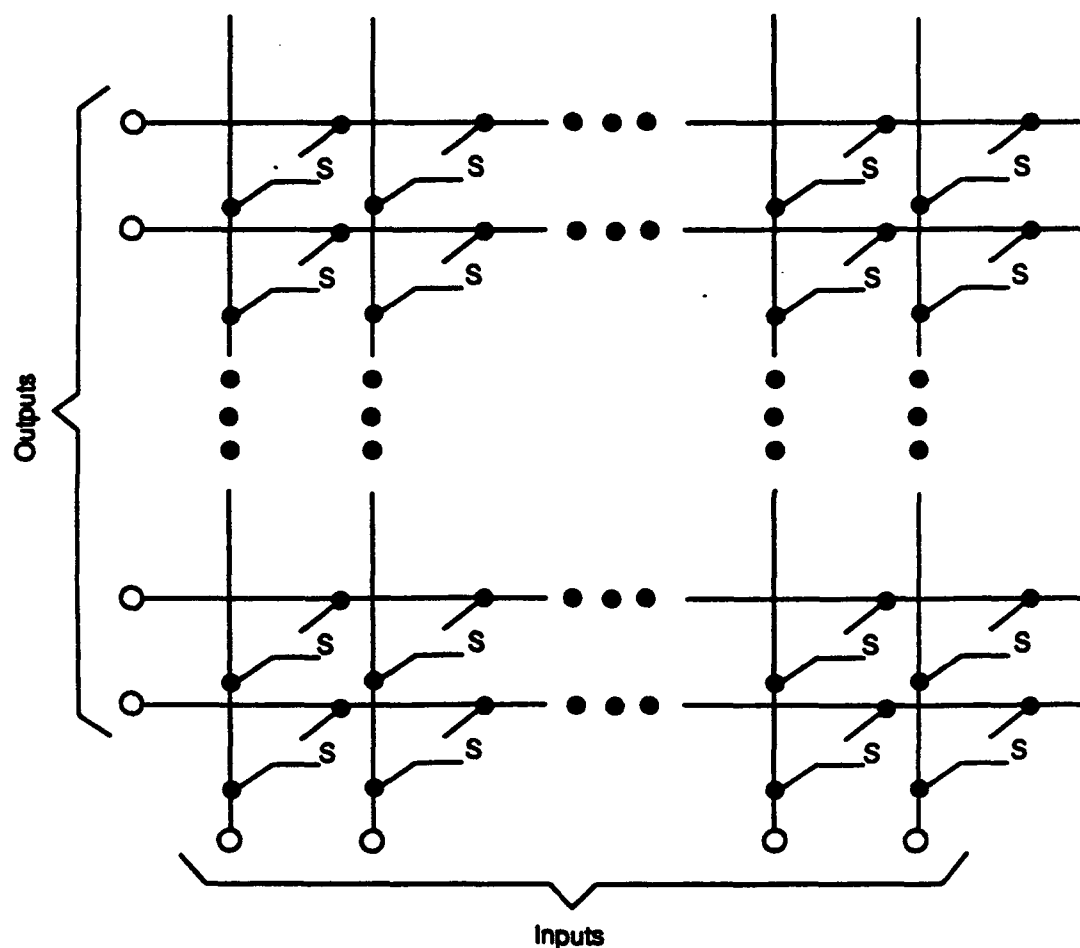


Figure 2.7. Crossbar network

The crossbar switch is the ultimate solution to bandwidth and the common access throughput for high performance multiple-processor systems. It is a modular network in that the bandwidth is proportional to the number of processors. The common access throughput is the same as the number of processors, assuming an equal or greater number

of memory modules.

The single most often cited disadvantage of the crossbar switch is that the number of switches required to implement it grows as N^2 , given N processors and N memory modules. Until a few years ago, some researchers even claimed "In fact, considering the current low costs of microprocessors and memories, a crossbar would probably cost more than the rest of the system components combined" [Patel 81]. There are numerous references to actual implementations of relatively large crossbar switches, for example [Barber 88; Shin 88; Rana 89-1; Rana 89-2; Cooperman 89]. With currently available technologies, it is feasible to implement a crossbar switch on a single chip with up to a few hundred single-bit inputs and outputs. Some examples of multiple-processor systems using a crossbar interconnection network can be found in Cedar [Gajski 83; Gajski 86], Aquarius [Srini 85], the Las Alamos project [Trujillo 82], and the Burroughs Scientific Processor (BSP) [Kuck 82].

Multistage Interconnection Networks

Multistage interconnection networks form the third major dynamic network configuration. These networks comprise multiple stages of switching elements, called *nodes*. There are several possible designs for the nodes. Nodes between adjacent stages are connected by links. The two terminal stages are also connected to the input ports and the output ports of the network. A path or connection between an input port and an output port is created by suitably selecting an alternating sequence of nodes and links.

Circuit Switching and Packet Switching are the two most popular schemes for communication in dynamic networks for multiple-processor systems. In circuit switching, a complete path is established from an input port to an output port before any information is transmitted. The path can be either a direct electrical connection or a direct logical path through gates. Circuit-switched networks are characterized by both the set-up time (for establishing the path) and the transmission time (for moving information over the path). The transmission time is typically faster than the set-up time. Packet switching in multiple-processor systems is a mode in which relatively small, fixed size units of information, called packets, move from stage to stage as paths between the stages become available. They do not require an entire path to be established prior to entering the network.

Classification of Multistage Networks

Even though most of the multistage interconnection networks can be used either with circuit switching or packet switching, their performance may vary drastically with the switching scheme. Multistage networks can be divided into two categories, one each for packet-switched and circuit-switched schemes. In the packet switched category, multistage networks can be divided into three broad classes [McMillen 84]: Single path networks, multiple path networks, and permutation networks. In the circuit switched category, Benes classified multistage networks into four classes based upon their ability to establish connections [Benes 62; also Broomell 83; Marcus 77]: Blocking networks, rearrangeable nonblocking networks, wide sense nonblocking networks, and strictly nonblocking networks.

First we consider the three types of packet-switched networks. The *single path networks* have exactly one path between any arbitrary input-output port pair. There are two classes of networks in this category: Banyan class [Goke 73], and the Delta class [Patel 79; 81]. The Banyan class is extremely general and was originally presented as a "Hasse diagram of a partial ordering in which there is only one path from any *base* to any *apex*". A *base* is a vertex having no arcs incident into it, an *apex* is any vertex with no arcs out from it, and all other vertices are called *intermediates*. A large class of networks can be built using this scheme, but among the more practical are the regular SW-banyans, and the CC-banyans (In the SW-banyan structure, there are an equal number of vertices at each stage, and all *intermediate* vertices have the same degree. CC is an acronym for Cylindrical Crosshatch, since a CC-banyan can be neatly laid out as a crosshatch pattern on the surface of a cylinder). The regular SW-banyan is equivalent to the Delta class. Examples of single path networks that are equivalent to this class are the Bitonic Sorter [Batcher 68], Staran Flip [Batcher 76], Omega [Lawrie 75], Extended Shuffle-Exchange [Lang 76-1], Indirect Binary N-cube [Pease 77], Generalized cube [Siegel 78, 85], Baseline/Reverse Baseline [Wu 78], Reverse exchange [Wu 79] and the Butterfly [Pease 68]. An 8×8 Omega network is shown in figure 2.8.

Multiple path networks are those that have more than one possible path between any arbitrary input-output port pair. This category includes all permutation networks and the PM2I-type networks [Siegel 85]. However, multiple path networks are often characterized from the point of providing a limited (usually 2) alternate paths between an input and an output. Examples of this class are the Data Manipulator network [Feng 74], Augmented Data Manipulator (ADM) network [Siegel 78], and the Gamma network [Parker 82]. An 8×8 Data Manipulator network is shown in figure 2.9.

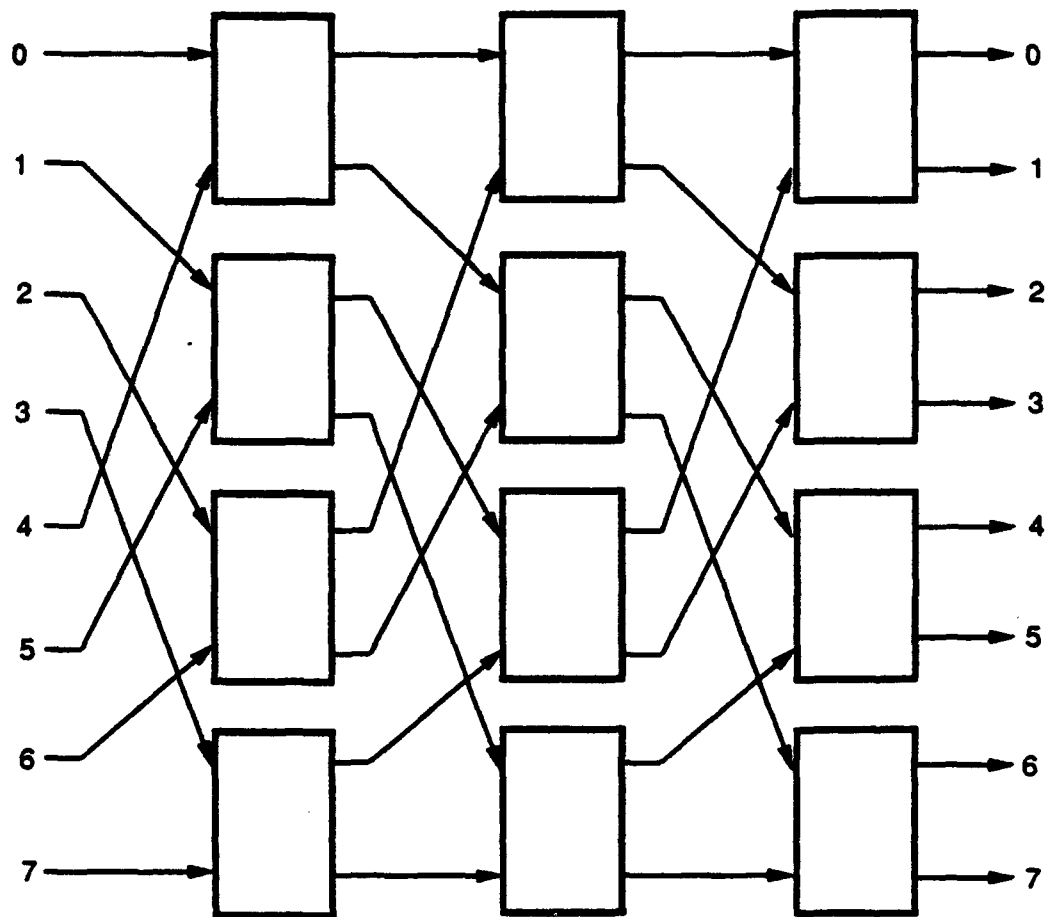


Figure 2.8. An 8×8 Omega network

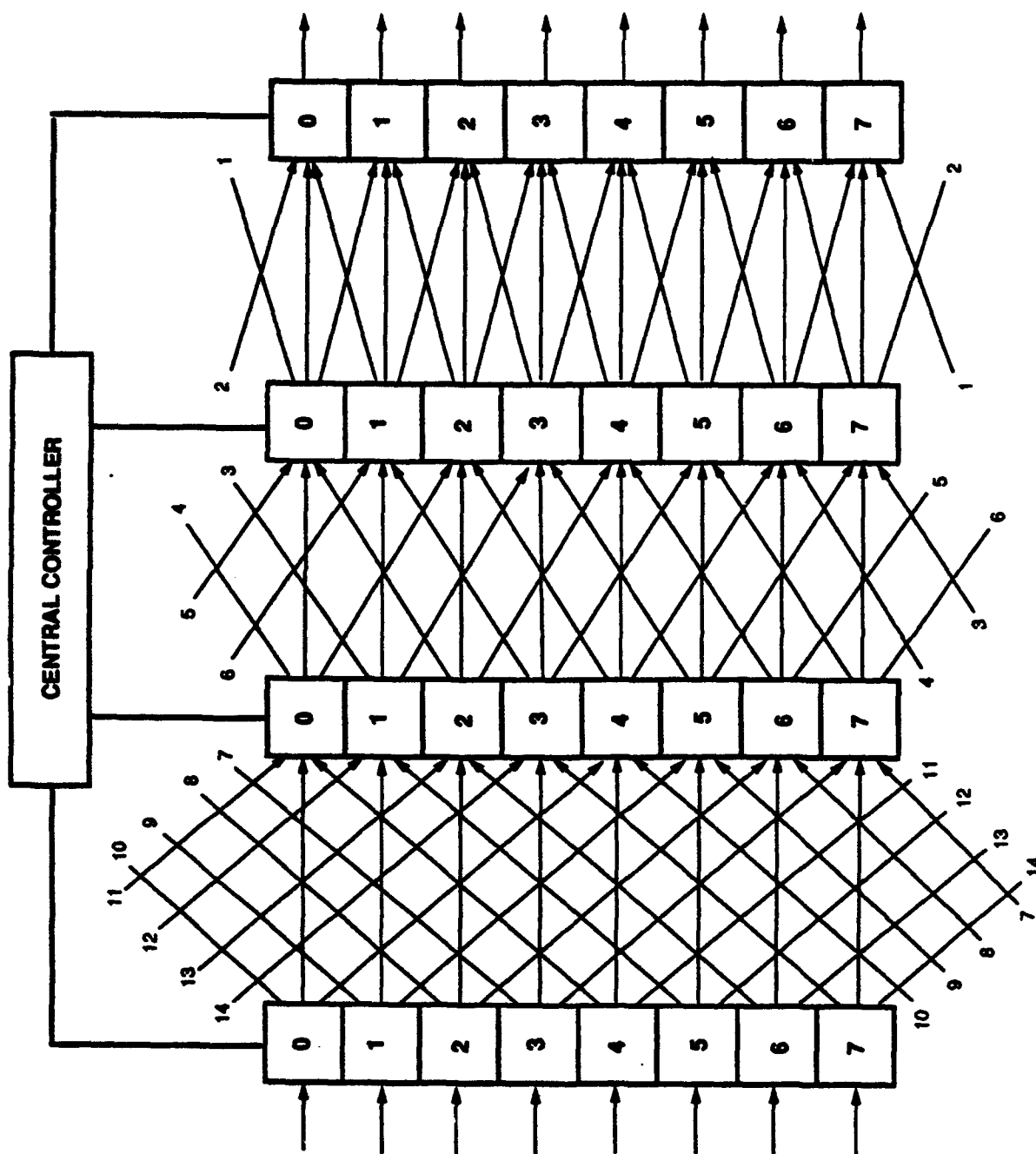


Figure 2.9. An 8 x 8 Data Manipulator Network

Permutation networks are those that can connect their inputs to their outputs in any arbitrary way as long as no two inputs go to the same output. For an N input, N output network, there are $N!$ possible connection patterns. Examples of permutation networks are Clos networks [Clos 53], Benes network [Benes 62], Waksman's modification [Waksman 68], the Bitonic Sorter [Batcher 68], and the cellular interconnection arrays [Kautz 68].

Now we turn to the four types of circuit switched networks. *Blocking Networks:* A network is said to be blocking if it can realize only a subset of the $N!$ permutations of the input ports onto the output ports. For example, as a circuit switching topology, the Omega network has limited capabilities. An 8-input 8-output Omega network comprises a total of twelve 2×2 switching elements, arranged in three stages of four switches each. This network will allow only $2^{12} = 4096$ different states for the 12 two-state elements as opposed to the $8! = 40320$ possible permutations of 8 inputs to 8 outputs that can be realized with a permutation network. All the examples cited in the single-path network category except Batcher's Bitonic sorter, fall into the blocking network class.

Rearrangeable Networks: A network is said to be rearrangeable or rearrangeable nonblocking when a desired connection between an unused input and an unused output port may be temporarily blocked, but can be established if one or more existing connections are rerouted or rearranged. Examples in this class are Waksman's permutation network [Waksman 68], the Bitonic sorter [Batcher 68], and the cellular interconnection array [Kautz 68].

Wide-sense nonblocking network: A network is said to be nonblocking in the wide sense or wide-sense nonblocking if any desired connection between an unused input and output port can be established immediately without interference from already existing connections, *provided* that the existing connections have been inserted using some routing algorithm peculiar to the network. If the algorithm has not been followed, some attempted connections may be blocked. Benes networks [Benes 62-1] are an example of this class.

Strictly nonblocking networks: A network is said to be nonblocking in the strict sense or strictly nonblocking if any desired connection between an unused input-output pair can be established immediately without interference from any arbitrary existing connections. Clos networks [Clos 53] are examples of this class.

Control and routing in multistage networks

The control of an interconnection network is as important as its implementation. It is a complex problem, and often depends upon the overall objectives and design of the multiple-processor system. There are two extremes of network control: *Centralized* or *Common*, and *Distributed*. The routing method in an interconnection network is primarily determined by the overall multiple-processor architecture, and its intended application. There are two extremes to it: *Synchronous* and *Asynchronous*.

Centralized control is used extensively in telephone crossbars to set up connections for all users. In multiple-processor systems, central control is typical in SIMD or Array processors. Essentially, the network is used either as a PE to PE data permutation network, or a PE to Memory data alignment network. A control processor issues an opcode to the network control unit specifying a particular permutation (or other) configuration to be established [Batcher 76; Feng 74; Pease 77]. An example of centralized control is the Data Manipulator network shown in Figure 2.9.

Distributed control is typical in MIMD or multiprocessors. In such networks, the requesting input port provides the routing tag in the message header [Lang 76-1; Lawrie 75; McMillen 82; Siegel 81; Tripathi 79; Wu 78]. The intermediary nodes in each stage determine their setting by examining the tag.

Synchronous routing is typical in SIMD or Array processors. The operation of such machines is usually divided into alternating sequences of computation and communication. The central control unit provides the synchronization points for all the PEs to start sending messages simultaneously in the beginning of a communication cycle, and waits until all the PEs have sent their messages to provide a synchronization point to begin a computation cycle.

Asynchronous routing is typical in MIMD machines and computer networks. Asynchronous routing is most often accompanied by distributed control. In this scheme, computation and communication are not interleaved. Instead, at any time an arbitrary PE can request to communicate with any other PE or memory module irrespective of the state of the other PEs.

Now we turn to a few examples to illustrate the preceding points.

Examples

Pease [Pease 77] proposed a multistage network called the indirect binary n -cube network, which is a typical example of a centralized control, synchronous routing network. For an N processor PE to PE network, it comprises $\log N$ stages of 2×2 switching elements or nodes. By closing the switches of the i th stage in cross-connected manner and switches in all other stages in direct-connected manner, direct paths are created from the output ports of all PEs X to the input ports of PEs $(X + 2^i) \bmod N$. If the PEs are viewed as connected in a virtual binary hypercube structure, this would have the effect that the PEs are communicating in the i th dimension of the hypercube. Pease showed how this network could be used in computing the FFT and a wide range of other numeric problems.

Lawrie [Lawrie 75] proposed a distributed routing scheme based on the destination tags in his Omega network, shown in figure 2.8. It should be noted that most other single path networks with distributed routing schemes use similar methods. This network is most suitable for asynchronous routing. 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 $d_{n-1} \dots d_1 d_0$ be the binary representation of D . The switching elements in stage i examine bit d_{n-1} of the destination address. If $d_{n-1} = 0$ the upper output is selected in the 2×2 node, otherwise the lower output is selected. By successively examining appropriate bits in each stage, a path can be created from the requesting input to the desired output. If the network is bidirectional, it can be readily verified that this scheme also works in reverse (from output to input).

Batcher [Batcher 68] proposed the Bitonic sorting network, which can be used as a distributed control, synchronous routing network in multiple-processor systems. This network comprises stages of 2-input 2-output comparison nodes. When two inputs are presented at the inputs of the node, the smaller of the pair appears at the upper output of the node, and the other appears at the lower output. Batcher showed that by suitably connecting these stages, the whole network can sort any arbitrary sequence at the inputs to appear at the outputs.

There are examples of centrally controlled asynchronous routing [Payne 86] where an interconnection network may consist of a control matrix and a switch (or communication) matrix. Each input and output port is connected in parallel to both matrices as shown in figure 2.10. The two matrices may be implemented in different technologies to take specific advantages of appropriate technologies. Here, the control matrix operates essentially as a

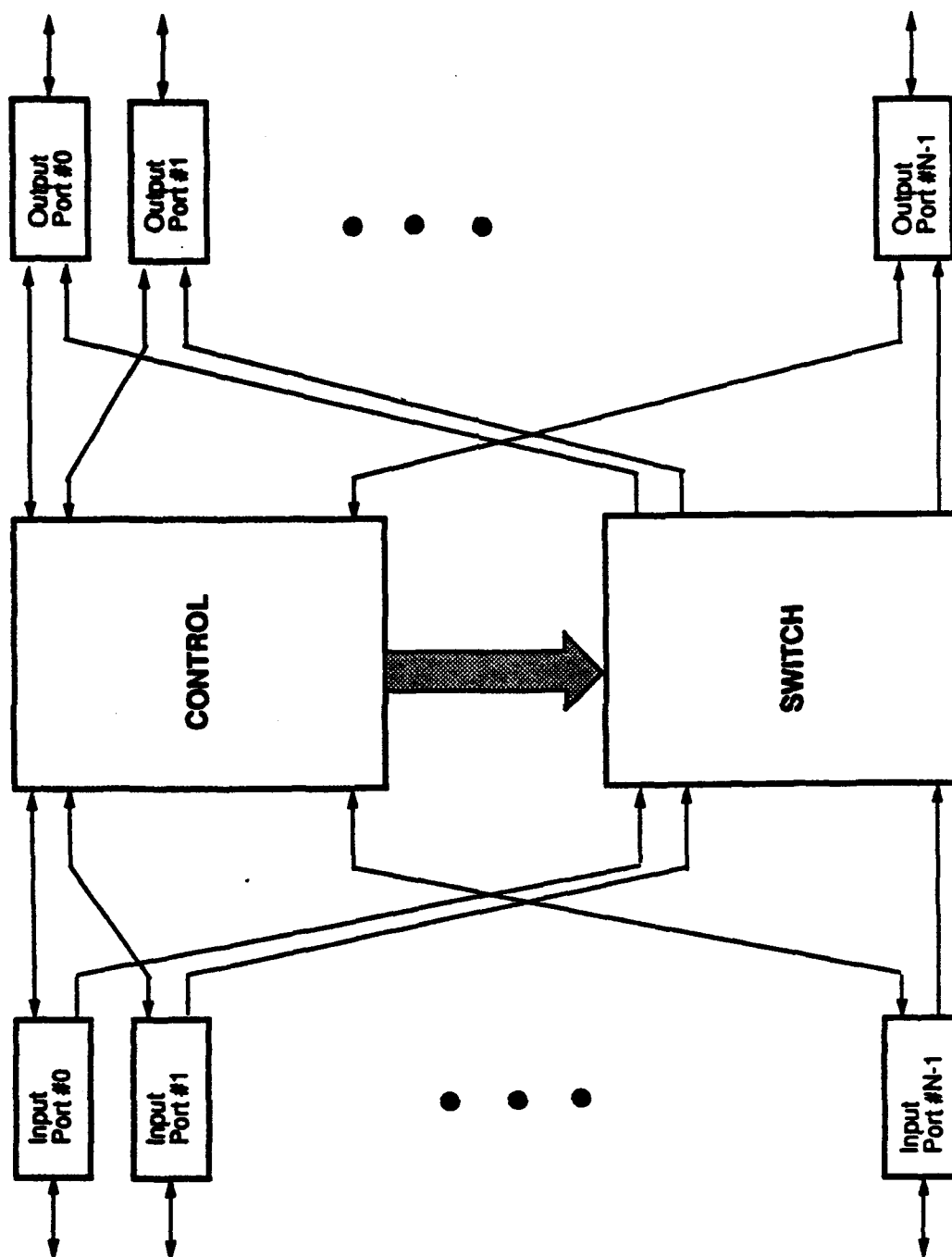


Figure 2.10. Centrally controlled asynchronous network

watchdog control processor for the switch matrix and continuously scans any path segment from the PEs. As soon as a PE makes a request, the control matrix makes a suitable connection in the switch matrix for the requesting PE to communicate with the desired PE.

In the next section, we provide an overview of the architectural characteristics and requirements for computation and communication at the ICAP level of the IUA.

CHAPTER 3

ARCHITECTURAL REQUIREMENTS OF INTERMEDIATE-LEVEL VISION

This chapter addresses the architectural characteristics and requirements of interprocessor communication at the ICAP level of the Image Understanding Architecture. We view interprocessor communication in a multiple-processor system as an integral part of the underlying parallel architecture that is, the communication requirements can only be defined in the context of the requirements of the underlying architecture. The architectural requirements are, in turn, a direct consequence of the intended application domain for which the parallel system is being built. Therefore, to ascertain the requirements of the ICAP communication network, we first investigate the computational characteristics of the vision tasks to be run at the ICAP level. From these characteristics, we extract the architectural (communication and control) requirements of the ICAP communication network.

The tasks in a parallel processing environment can be largely characterized in terms of their data distribution and control mechanisms – interprocessor communication is often an implied aspect of both. Therefore, while investigating the computational characteristics of the tasks to be run at the ICAP level, we are specifically interested in the manner in which the data is distributed amongst the processors during various stages of computation, and the various mechanisms that are used. This will give us information about the data and control granularities embodied in the tasks; most importantly, control and data dependencies.

We are primarily interested in the computational characteristics of the tasks to be run at the ICAP level, but there are two problems. First, it is impossible to look at the tasks for the ICAP in isolation, because many of the tasks are ill-defined and transcend the processor boundaries of the CAAPP, ICAP, and SPA. Second, the IUA is primarily intended as a vision research tool and, therefore, different researchers may choose to divide and map the tasks differently onto different levels of the IUA. These two problems make it necessary for us to study the computational characteristics of the intermediate-level tasks in two stages: First, from the point of view of the well-established image interpretation tasks that have been identified in the VISIONS [Hanson 86] laboratory at the University of Massachusetts, and second, from the literature where other vision researchers have made use of a wide range of techniques for image interpretation. From the general literature, we

take a representative sample of the work being done. Additionally, because of the non-unique mapping of intermediate-level vision tasks onto the ICAP level, it is necessary to view them in the light of low- and high-level vision tasks.

One point should be noted before we go any further. Machine vision research is highly dynamic and evolutionary in nature, and the development of parallel architectures for machine vision is a nascent field. Most of the computational models and tasks that exist for machine vision are geared towards experimentation and further research, and are not meant to be the "final solution." Therefore, our approach in this chapter is to integrate the requirements of known tasks in the VISIONS lab onto the IUA, with predicted requirements based on representative tasks proposed by other researchers in a more abstract form. This will give us a "minimal set" of computational requirements for the processes running on the ICAP level of the IUA. To address the problem of unforeseen computational requirements, we will try to make the communication network more capable than what is currently sufficient. As knowledge-based machine vision becomes better understood, and the IUA is redesigned, the ICAP communication network can be redesigned accordingly.

The rest of this chapter is organized as follows. Section 3.1 provides an overview of some of the low-level vision tasks defined in the VISIONS group. Next, we review some of the tasks for high-level vision, which involves an overview of the Schema system [Draper 89]. Schemas at the SPA level treat the low- and intermediate-level vision tasks as a set of knowledge sources. Some of the low- and intermediate-level knowledge sources along with the schema system are discussed in section 3.2. A discussion of some of the intermediate-level vision tasks is provided in section 3.3. After the discussion of the intermediate-level vision tasks, we turn to the architectural characteristics of the ICAP level of the IUA in section 3.4. Finally, from this we extract the architectural (communication and control) requirements of the ICAP communication network in section 3.5.

3.1 Low-level vision

In a 2-D image understanding paradigm, two primary goals of low-level vision are *feature extraction*, and *segmentation*. Sometimes the term *segmentation* is used loosely to cover not only the normal definition of partitioning an image into connected, non-overlapping sets of pixels, but also low-level line extraction algorithms. Currently three segmentation algorithms are used in the VISIONS environment:

1. Histogram-based region segmentation.

2. Straight-line extraction by gradient orientation, and
3. Straight-line extraction by edge grouping.

These algorithms are discussed next.

3.1.1 Histogram-Based Region Segmentation

Region segmentation is used to decompose an image into non-overlapping regions, where pixels in each region have some common property such as brightness, color, or texture. Domain specific knowledge in the form of top-down control may be introduced to resolve various ambiguities in the segmentation task. The region segmentation algorithm used in the VISIONS environment is based on analysis of histogram peaks and valleys in local subimages and is comprised of two sub algorithms: a *localized histogram-segmentation algorithm* and a *region-merging algorithm* [Beveridge 89]. Either can be used by itself to produce a region segmentation, or they can be combined into a single system as in figure 3.1.

The local histogram region segmentation algorithm

There are five phases to the local histogram-segmentation algorithm:

(i) *Preprocessing: Edge-Preserving Smoothing and Scaling*

In the first phase, the initial image is processed via edge-preserving smoothing and scaling. The goal in smoothing the image is to remove minor variations so that histograms will be distinctly peaked and regions more uniform, and to do so without blurring the boundaries of adjacent regions. This is accomplished with an edge preserving smoothing algorithm developed by Overton and Weymouth [Overton 79]. The value of a pixel in one iteration of smoothing becomes the weighted average of itself and its neighbors within a fixed size mask, where the weights are decreasing functions of spatial separation and intensity difference between pixels.

Scaling the data adjusts its dynamic range, and hence the number of buckets in the histograms, which directly affects the histogram analysis.

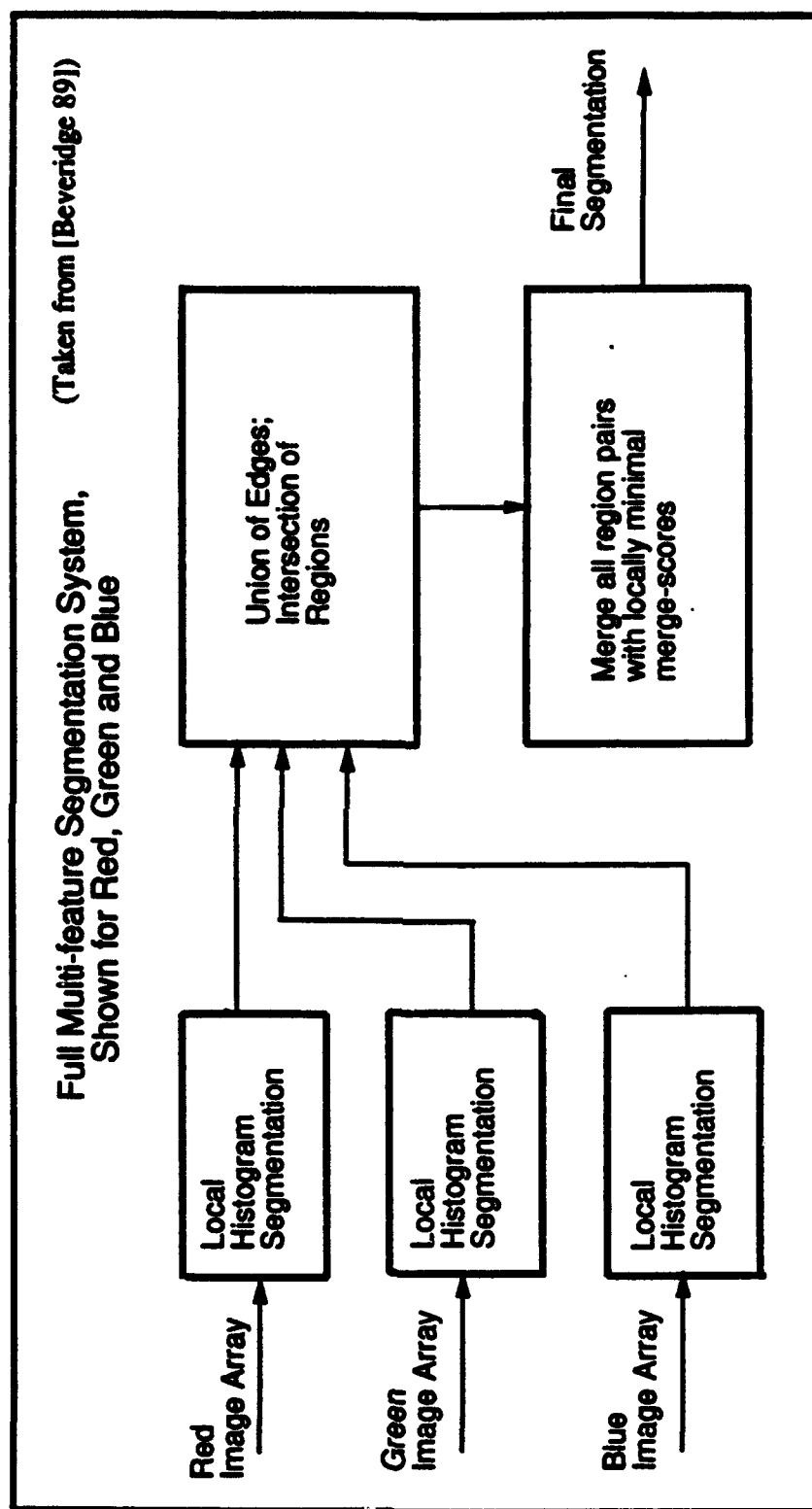


Figure 3.1. Organization of Region Segmentation

(ii) Localized Sector Segmentation

In the second phase, a histogram of pixel values is formed for each subimage called a **sector**. Typically, the dimensions of a sector are 16×16 or 32×32 pixels. A small overlap of about 25% is allowed in the sectors to help correct for situations in which a sector boundary happens to divide an area into two parts that should be a single region. If this occurs, the cluster corresponding to the smaller part may be missed in the localized histogram.

Next, within the histogram of each sector, "significant" clusters are identified by means of a *peak-valley* analysis. A cluster is defined by a histogram peak and its two neighboring valleys. Only those clusters are chosen that satisfy three measures: Peak height, Peak to valley ratio, and Peak distance. The peak height is simply the value of the histogram at the peak. The peak to valley ratio is the peak height divided by the height of the higher of the two neighboring valleys. The peak distance is the gray-level difference between two peaks.

Once the clusters and surrounding valleys have been identified, the pixels are given the labels of the cluster to which they belong. Since each cluster may give rise to multiple regions, a connected components algorithm is applied independently to each sector to uniquely label individual regions.

(iii) Adding Clusters from adjoining Sectors

The third phase involves adding clusters that might have been missed in the second phase by examining clusters in adjoining sectors. Although sector overlap reduces the problem of losing portions of regions protruding in from an adjacent sector, there are still cases where such regions are missed. Peaks in the central sector are matched with the labels (peaks) associated with regions along its boundary in the four neighboring sectors. This process is carried out iteratively (typically, twice). The peaks from the neighboring sectors that are not within a minimum *peak-distance threshold* of the central sector clusters are considered as potential candidates to be added to the central sector. Next, for the candidate peaks from the neighbors, peaks are added which show evidence of their presence in the central sector. A histogram is constructed from the original pixel data over the half of the central sector closest to the adjoining boundary of the sector containing each candidate peak. A cluster peak (i.e. a local maximum) within the minimum peak-distance threshold of the candidate peak will serve as evidence for its presence. Otherwise, the candidate peak is replaced by the largest local maximum within this range, and this shifted peak is added to the set of

peaks for the central sector.

After this step, the augmented set of peaks in the central sector form the basis for a new set of clusters. To arrive at the new set of clusters, valleys between peaks are determined and the original peak-valley analysis for extracting clusters is applied with cluster selection parameters changed to achieve more sensitive cluster selection.

(iv) Removal of Sector Boundaries

The fourth phase involves removing artificial sector boundaries that are created by the algorithm. A local and global *merge-score* strategy is employed to remove artificial boundaries between regions that are both locally and globally similar.

(v) Postprocessing: Small-region suppression

The fifth and final phase is a postprocessing step which removes very small fragmented regions. Very small regions can be removed by either merging them with adjacent regions with which they share the longest common boundary, or by merging them with adjacent regions that are closest in gray level.

Region merging algorithm

The second part of the region-segmentation algorithm is a region-merging algorithm. The region-merging algorithm alleviates the problem of very small regions formed due to textured portions of the input image. Also, if the histogram-based segmentation algorithm is used over more than one feature, the region-merging algorithm can be applied to merge the results of multiple segmentations.

The region-merging algorithm computes a *merge-score* between pairs of regions which is a measure of the similarity between them. A lower merge-score indicates a greater propensity to merge. Two regions are merged if their paired merge-score is lower than any other pair associated with either region. The region-merging algorithm is applied iteratively until all merge-scores are higher than a threshold.

In general, many region characteristics could be used in the region-merging algorithm to determine whether two regions should be merged. The following three characteristics

are used in the VISIONS environment via a weighted product of the individual measures: (i) *Similarity*, (ii) *Size*, and (iii) *Connectivity*.

The most commonly used characteristic in the region-merging algorithm is similarity. Many measures can be used here such as color, texture, intensity, etc. In some domains there may be a nominal best size for regions. Very large regions may not represent relevant image structure, yet too many small regions may place an increased computational burden on later interpretation process. This measure is dependent on the domain. The connectivity measure reflects the heuristic that a pair of regions sharing a relatively large common boundary are good candidates for merging. This measure discourages the formation of regions with small necks connecting larger areas.

Next, we discuss the first of the two line extraction algorithms used in the VISIONS environment.

3.1.2 Straight-line extraction by gradient orientation

The abstraction of sharp, linear intensity changes into intermediate-level symbolic tokens (lines) is an important task in image interpretation. Once representative lines corresponding to image intensity changes are extracted, many attributes that are useful in the later interpretation process can be derived.

One approach to extracting straight lines from intensity images and the associated edge parameters is described in [Burns 86]. In this approach, the intensity surface is segmented into contiguous pixels with similar gradient orientation, called *line-support regions*. Using a planar fit to these support regions, representative lines are extracted. A number of attributes can be extracted from each line-support region, its corresponding weighted planar fit, and the representative line.

There are three steps in this approach to extracting straight lines:

(1) *Grouping pixels into line-support regions*

The first phase in this step involves deriving a gradient-image by convolving the intensity image with two different 2×2 masks [Burns 86]. Corresponding to each pixel in the intensity image, there is a vector in the gradient-image that encodes the local gradient-magnitude and the gradient-orientation.

In the second phase of this step, the local gradient-orientations are grouped into regions. This is similar to segmenting the gradient-image, except that gradient-orientation (angle) is used as the basis for grouping. The 360 degree range of gradient orientation is divided into two sets of overlapping partitions, each with eight 45 degree intervals. The overlapping partitions are used to avoid problems with possible fragmentation of a line-support region if the distribution of gradient-orientation crosses a partition boundary. Thus, if one of the 45 degree partitions starts at 0 degrees, then the second partition starts at 22.5 degrees. Every pixel of the input image is assigned one of the 8 labels for each partition on the basis of the corresponding vector in the gradient image.

The third phase of this step is merging the two sets of labels from the second phase in such a way that a single line in the image is associated with a single line-support region. The region considered best for pixel association (on the basis of one of the two labels from the two overlapping partitions) is the one that provides an interpretation of the line that is longer. The following scheme is used to select such a region for every line. First, the lengths of the lines are determined for each line-support region. Since each pixel is a member of exactly two regions (one in each of the overlapping partitions), each pixel votes for the longest interpretation of the line (i.e. the longest line-support region). Finally, the percentage of voting pixels within each line-support region is the "support" of that region. Typically the regions selected are those that have support greater than 50%.

(2) Interpreting the line-support region as a straight line

Once input image pixels have been grouped into line-support regions, each region represents a potential candidate for a straight line. In order to extract a representative straight line, a plane is fit to the intensity surface of the pixels in each line-support region. The parameters of this plane are obtained by a weighted least-squares fit to the feature value on the intensity surface corresponding to each line-support region. The pixels are weighted by the local gradient magnitude so that those in rapidly changing portions of the intensity surface dominate the fit.

Once a planar fit to the intensity surface is available, the constraint on the orientation of the representative line is that it should be perpendicular to the gradient of the fitted plane. Burns' approach for locating the line along the projection of the gradient is to intersect the fitted plane with a horizontal plane representing the average intensity of the region weighted by local gradient magnitude.

(3) Attribute extraction and filtering

The line-support regions and the planar fit of the associated intensity surface provide the basis for extracting a variety of attributes beyond the basic orientation and position parameters. Length, width, contrast, and straightness can be calculated. Based upon these line attributes, the large set of lines can be filtered to extract a set with specific characteristics such as short textured-edges, or to select a "working-set" of long lines at different levels of intensity.

Next, we discuss the second straight-line extraction algorithm used in the VISIONS environment.

3.1.3 Straight-line extraction by edge grouping

Boldt [Boldt 87; Weiss 86] developed a hierarchical algorithm for grouping collinear line segments into progressively longer segments on the basis of geometric properties of the hypothesized group as well as the similarity of image features along both sides of the component lines. This algorithm is mentioned here as a low-level vision algorithm because, in a restricted form, it can be viewed as an alternative to Burns line extraction algorithm [Burns 86]. In its general form however, this algorithm is a part of perceptual grouping [Hanson 86] and hence, an intermediate-level vision task. This kind of grouping is also employed in the ISR (to be discussed later). We will further discuss Boldt's grouping scheme in the section on perceptual organization and grouping.

3.1.4 Other low-level algorithms

There are numerous additional low-level algorithms that different researchers use as part of low-level vision. These include a wide variety of image processing, and other numeric and non-numeric algorithms. Table 3.1 lists some important image processing operations taken from a survey by Preston [Preston 89].

In addition, there are a number of algorithms that are used in low-level vision. Their scope and extent would make it impossible to list all of them here, thus we list only a few. Extensive details can be found, for example, in [Ballard 82; Hanson 78; Horn 86; Marr 82; Rosenfeld 82].

Table 3.1. Basic image processing operations

Utilities	Geometric
Storage allocation	Scaling/rotation
Program control	Rectification
Formatters	Mosaicing/registration
I/O control	Map projection
Test-pattern generators	Gridding/masking
Transform	Arithmetic
Noise removal	Point
Fourier analysis and other spectral transforms	Line (vector)
Power spectrum	Matrix
Filtering	Complex number
Cellular logic	Boolean
Measurement	Decision theoretic
Histogramming	Feature select (training)
Statistical	Classify
Numerical and geometric	Evaluate results

- Edge-preserving smoothing.
- Different forms of thresholding.
- Different kinds of convolution.
- Hough Transform.
- Euclidean-Distance Transform.
- Subtracting two images.
- Convex Hull
- Voronoi diagram.

Next, we discuss high-level vision.

3.2 High-level vision

High-level vision comprises two components: Scene-independent as well as scene specific knowledge and interpretation strategies. Scene-independent knowledge is used to create models of the scenes and objects in the scenes. These models could either be domain independent or specific to the domain of the scene. From these models, 2-D and 3-D representations of the particular scene and its objects are derived. The objective of high-level vision is to try to match these stored models and representations to the abstract tokens stored in the Intermediate Symbolic Representation (ISR) database. The tokens in the ISR represent abstractions of significant events extracted from the input image. This entire process of matching first passes through the initial hypothesis generation phase, and then multiple phases of refinement until a "reasonable" interpretation of the scene can be generated by matching different objects in the scene to one of the internally instantiated *Semantic-networks* (to be discussed in the following paragraphs) of knowledge. When no match can be found, no reasonable interpretation of the image is possible. Different strategies may alter the course of action. The high-level in the VISIONS environment is embodied in the Schema system [Draper 89], which is discussed next.

3.2.1 The Schema system

The schema system implements a knowledge-based approach to image interpretation and is comprised of two components: Object knowledge, and Control knowledge. Object knowledge can encompass domain independent as well as domain specific information such as 3-D structures and models, 2-D appearances of these models, and geometric and co-occurrence relationships between different objects and object parts. Control knowledge encompasses information for efficient extraction and organization, strategies for matching this information in the form of abstract tokens from the lower levels, to stored models and their projections, and control of processing in the entire system to ensure consistency in the evolving interpretation. The schema system incorporates control as a key part of the knowledge base. In this subsection, we will sometimes use the term knowledge collectively for object knowledge and control knowledge.

Knowledge in the schema system is represented by a hierarchical structure organized as a semantic network of schema nodes. The object knowledge in the schema system is encoded as a part-of graph for the object classes for a given domain. An example part-of

graph for a house scene knowledge base is shown in figure 3.2. In addition to the object

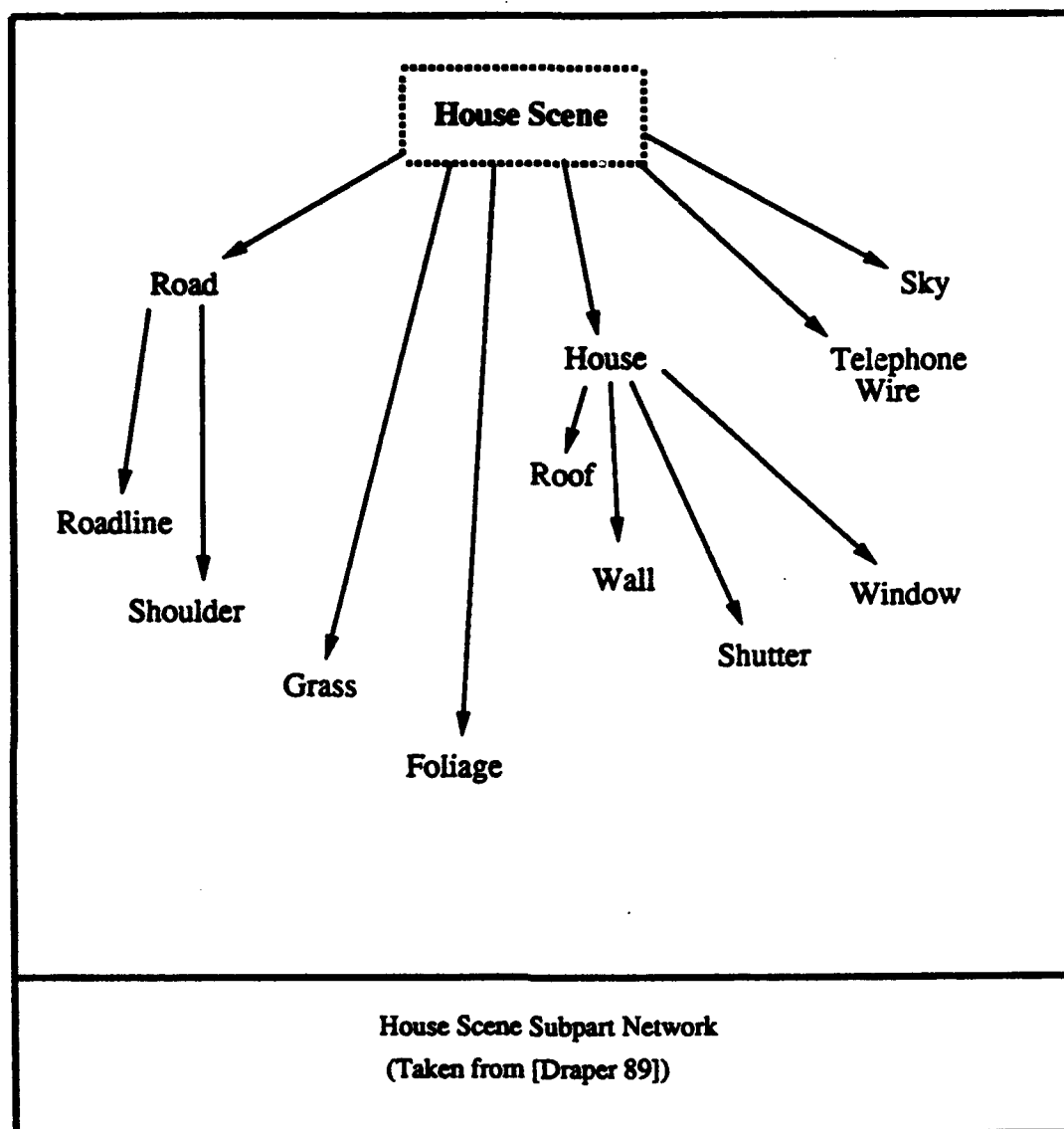


Figure 3.2. House scene part-of network

knowledge, each schema has control knowledge describing object recognition techniques in the form of hypothesis generation and verification processes called interpretation strategies. There could be multiple interpretation strategies associated with each schema.

In order to make the schema system computationally as well as storage efficient, the knowledge in it is divided into long term memory (LTM) and short term memory (STM) across the levels of hierarchy. This differentiates the system's permanent apriori knowledge base (which could reside in secondary storage as sleeping processes) from the representation

derived from the sensory data of a specific image during the interpretation process (only those schemas that are necessary for the specific image are activated. For example, *house-1* may be an instance of a two-story Victorian which is a subclass of the general class called *house*). Both, the STM and the LTM are multilevel organizations of 2-D and 3-D symbolic tokens such as points, lines, regions, surfaces and volumes, and abstract semantic tokens of objects and scenes. A node in the STM is an instance of a node in the LTM. For each hypothesized instance of an object class in an image, a *schema instance* is created. A schema instance is a copy of the schema which could be executed as a separate process with its own state. To gather support for the presence of a hypothesized object in a specific input image, each schema can invoke knowledge sources at the lower levels. Knowledge sources are the only means by which schemas interact with the lower levels, and will be discussed shortly. The relationship of the schema system with the rest of the VISIONS environment is illustrated in figure 3.3.

Schema Implementation

The schema system has been designed such that active schemas (in the STM) are viewed as a set of concurrent processes with coarse-grain communication. The manner in which the schema system, along with the rest of the VISIONS system, is expected to run on the IUA is illustrated in figure 3.4.

Schemas are run in a decentralized control manner. The control links are indirectly established as part of the schema hierarchy. This would make schema system implementation particularly suitable on an MIMD system.

While each schema is an expert subsystem for recognizing its associated object, it often requires knowledge about other objects in the scene. Additionally, schemas may have to exchange information about partial interpretations to arrive at a consistent final interpretation of the entire scene. The inter-schema communication cannot be determined apriori. Replication of information to all schemas is inefficient and introduces potential problems associated with data consistency in shared memory systems. Therefore, information sharing is implemented using a *global blackboard* in the schema system. A global blackboard (logical shared memory) allows schema instances to publish their contributions to the incrementally developing interpretation and to access the public contributions of other schemas. A schema can post messages to the global blackboard without knowing who will read them and likewise, reading a message requires no information about when the message was or will be

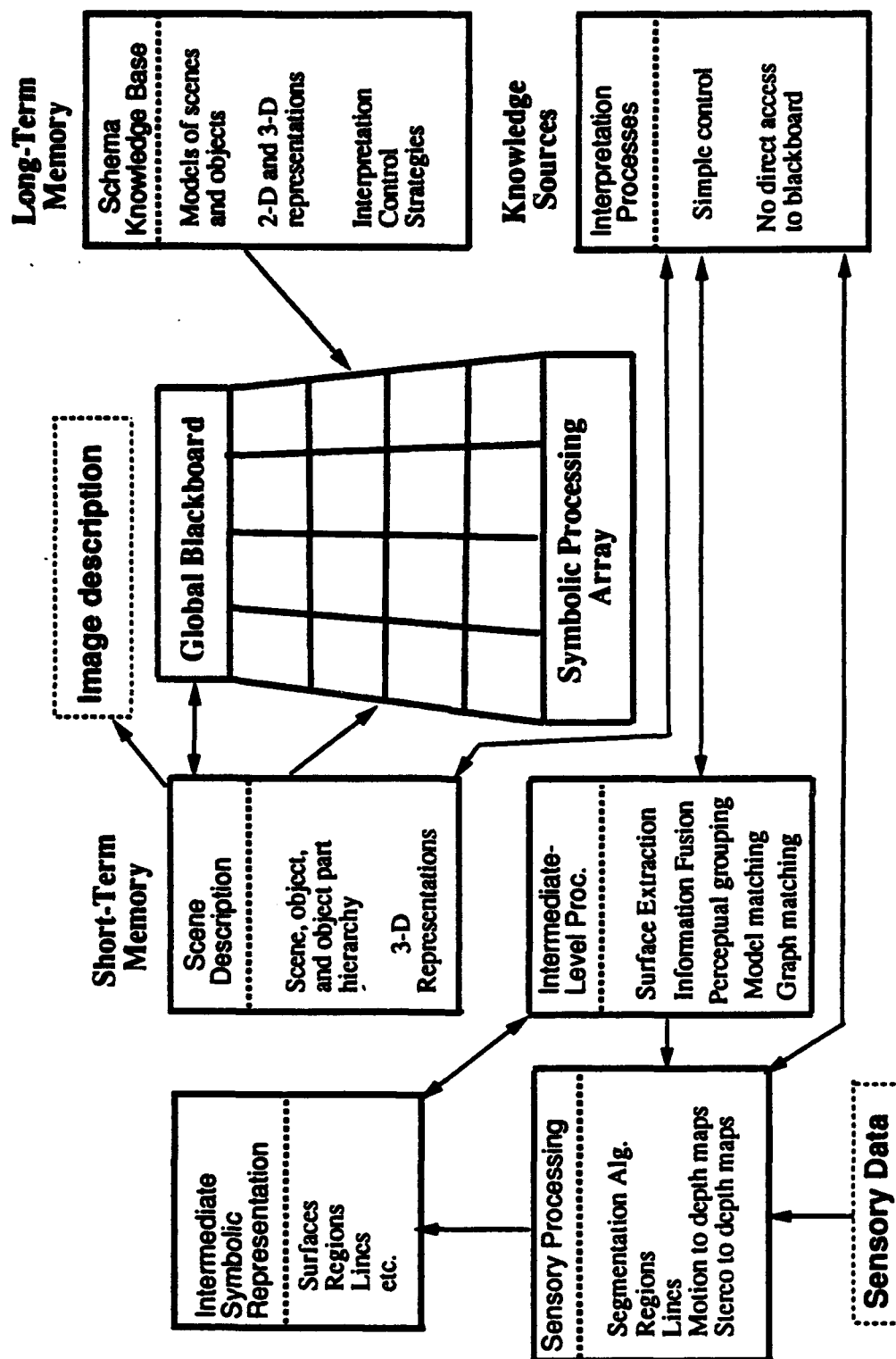


Figure 3.3. Overview of VISIONS system components

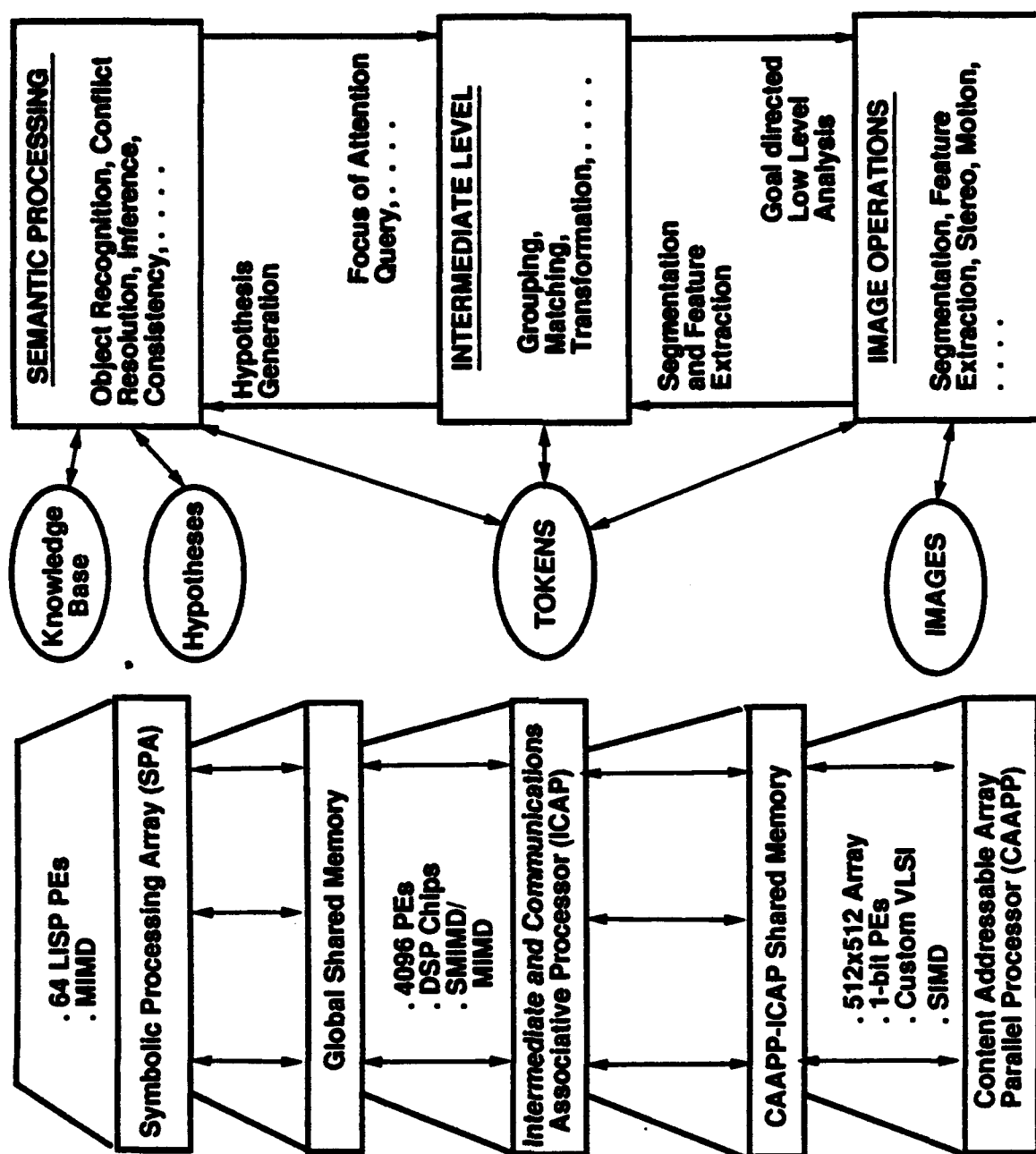


Figure 3.4. IUA and the VISIONS system components

posted, or who posted it.

Each schema is comprised of the following four parts: (i) Part-of graph, (ii) Internal hypothesis, (iii) Interpretation strategies, and (iv) Local blackboard.

As discussed above, the part-of graph portion of a schema encodes an expected representation of the specific object in the scene, derived from a scene independent model. The hierarchy of objects in a specific image are matched against a hierarchy of schema instantiations.

Each instantiated schema develops and maintains an internal hypothesis about possible instances of the specific object in the scene that it is designed to detect. An internal hypothesis consists of (1) tokens (such as points, lines, regions etc.) representing image events; (2) a set of endorsements from different knowledge sources; and (3) any other information, such as conflicts or confidences supplied by multiple interpretation strategies.

Interpretation strategies address the question of what to do next. For example, if a schema detects a house, it might initiate one or more schemas to search for roof, shutter, wall, and windows etc. Each schema might initiate more than one interpretation strategy. Each strategy in turn embodies one control link in the chain of schemas. In addition, interpretation strategies encode knowledge about which knowledge source to apply to gather evidence for the specific object in the scene.

A schema instance may contain many internal hypotheses, each of which must be available to all of its active interpretation strategies. A local blackboard provides the mechanism for all active strategies to share evidence or conflicts about a specific object's presence in the image. The local blackboard is accessible to all the strategies making up a schema instance, but only those strategies. Local blackboard messages can be highly schema specific and private only to the specific schema, which prevents the requirements of a large system from imposing undesirable restrictions on its subsystems.

The schema system interfaces with the rest of the VISIONS environment through knowledge sources. Knowledge sources are discussed in the next section.

3.2.2 Knowledge Sources

Knowledge sources are general purpose programs or processes which could vary over a wide range in their complexity and functionality. Knowledge sources' relationship with the rest of the VISIONS environment is illustrated in figure 3.3. Interpretation strategies

associated with the active schema instances in the short term memory call one or more knowledge sources to carry out a specific task for them. As such, the knowledge sources provide the means for top-down control from the high level to the lower levels. In addition, the knowledge sources provide the means for generating more abstract image descriptions for the high level, by taking more primitive tokens from the lower levels, which comprises the bottom-up processing of the input data.

From the point of view of the high level, various tasks at the low, and intermediate levels, such as image segmentation algorithms, perceptual grouping algorithms etc., can be viewed as knowledge sources. There is a qualitative difference in the manner that computation takes place at the low and intermediate levels in the VISIONS environment and the IUA. Accordingly, the knowledge sources are grouped into two categories; Low level knowledge sources, and intermediate level knowledge sources, depending upon what level of processing they are supposed to affect. When *meta-knowledge* issues are considered, the high level can also be viewed as a set of KSs with distributed control.

Next we turn to the two types of KSs.

Low-level Knowledge Sources

Low-level knowledge sources primarily process the numerical pixel arrays, and form the low-level part of the VISIONS environment. Low level KSs exploit spatial parallelism in the input pixel arrays. In the IUA, they are most suitable for a SIMD or data parallel implementation. The control structure in applying low level KSs is known apriori and usually follows a single thread. Many low level KSs are initially run with their default parameter settings. Thereafter, as the schemas detect different image contents, low level KSs can be selectively re-run with more sensitive parameters.

Currently, the following low level KSs have been identified in the VISIONS environment [Draper 89]. In the future, more KSs will be added to the system.

- Region Segmentation by Localized Histograms
- Straight-Line Extraction by Gradient Orientation
- Straight-Line Extraction by Edge Grouping
- Straight Bounding-Line Extraction
- Region Feature Extraction

• Straight-Line Feature Extraction

The Region feature KS computes features of regions obtained by region segmentation, and straight lines obtained by gradient-oriented line extraction. Computed features are divided into five categories: Color, Texture, Shape, Size, and Location. This KS provides the schema interpretation strategies measures such as contrast, bounding rectangle, hue, area, aspect ratio, centroid, parameter, line density (number of lines with more than certain length, contrast, slope etc.) of different regions.

The Straight-line Feature KS computes features for a straight line such as length, orientation etc., given the lines' end points.

Intermediate-Level Knowledge Sources

Intermediate-level knowledge sources are part of the intermediate-level processing in the VISIONS environment, and operate on the tokens generated by the low-level KSs, producing more complex abstractions for the interpretation strategies at the high level. The tokens generated by the low-level KSs are stored in the ISR. Therefore, many intermediate-level KSs rely heavily on the ISR for their efficient implementation.

To constrain the combinatoric explosion that can result from grouping a large number of low-level tokens according to multiple relations, the intermediate-level KSs are applied under the control of the interpretation strategies of the schemas that invoke them. In other words, intermediate level KSs are model driven; they are applied in response to the models that schemas want to match with some scene objects during various stages of image interpretation. The control structure for the intermediate level KSs follows from the models and hypothesis instantiated by the schemas during the interpretation cycle. The invocation and control of the intermediate level KSs should be contrasted with the low level KSs, which are typically invoked in a rather simple bottom up manner.

Intermediate level vision is currently an active area of research in the image understanding community and the VISIONS group. Therefore, many new developments may take place in the near future. Currently available intermediate level KSs are divided into five categories by Draper et al. [Draper 89] and are discussed next.

(i) Feature-based classification

In this category, there are two KSs: Initial hypothesis system (IHS), and Exemplar extension (EXEMPLAR). In the first phase of image interpretation, one or more segmentation algorithms (low level KSs) are applied to the image, and the tokens generated by these algorithms are stored in the ISR ¹. The segmentation algorithms are knowledge-free with respect to the environment depicted in the image, therefore their output may need further refinement by re-applying them with modified parameters, once there is an approximate idea as to what kind of image environment and objects are depicted in the image. The IHS knowledge source is meant to do precisely this. The IHS is the first step in image interpretation after tokens have been stored in the ISR, and is used in the initial bottom-up phase of image interpretation to generate a rank-ordered set of hypotheses about one or more objects in the scene. This work is discussed in detail in [Hanson 87-2; Lehrer 87]. The combination of IHS and EXEMPLAR is called the object hypothesis system and can be viewed as creating "islands of reliability" from which knowledge-driven processing may be initiated.

The way IHS works is as follows. In the early interpretation phase, when little or nothing is known about the environment depicted in the scene, the objective is to generate one or more reliable hypotheses for significant image events. These hypotheses will then provide a basis for further top-down control as context and expectations are validated with further knowledge-dependent processing. Initially, the IHS is given a set of training images. The training images are hand-labeled following region segmentation. The feature values of the objects corresponding with the regions in the training images are also known. From these training images, the IHS derives feature value constraints for each region, giving the IHS a method to correlate region feature values to the objects' stored feature values. Next, the image to be interpreted is supplied to the IHS together with its region segmentation. Two tasks can be performed by the IHS: apply IHS to a specific region and it returns a rank ordered set of objects (e.g. the IHS returns that a particular green patch could be grass, hedge, tree, ... sky. In other words the green patch is most likely to be grass, and least likely to be sky), or apply IHS to a specific object and it returns a rank ordered set of regions (e.g. if the object is roof, the IHS returns a rank ordered set region.1, region.2, ...). From one of these two outputs, attention can be focused on the most likely hypothesis to guide further processing.

¹The ISR is organized in such a way that it stores image tokens at multiple levels of abstraction in a hierarchical manner. At the lowest level, ISR stores tokens generated by the segmentation algorithms.

The functioning of the EXEMPLAR knowledge source is based on the presumption that the appearance of many objects such as grass, sky etc. will vary very little in an image (although they may vary more between images). An EXEMPLAR knowledge source takes an exemplar region from a specific image and returns other regions from the same image that appear similar.

(ii) Perceptual organization and Grouping

One important component of intermediate level vision is perceptual organization and grouping algorithms (Also called perceptual organization and grouping KSs). The objective of these KSs is two-fold: reduce the amount of data to be handled by the high level, and bridge the semantic gap between the low- and high-level vision components. In essence, these KSs are used to organize the input image information in such a way that an overview or abstraction of the image structure is available to the high level as well as quick access to finer levels of detail in the image.

Perceptual organization and grouping algorithms operate on symbolic tokens generated by the low-level vision algorithms (KSs). In current algorithms, the input data comprises various line and region tokens. Various perceptual (relational and spatial) constraints are used by these algorithms to organize the input tokens into more abstract entities. The perceptual constraints can either be derived from the input image in a bottom up manner, or from the knowledge base at the high-level in a top down manner. Data reduction in the context of these algorithms comprises two parts: Filtering data, and abstraction or grouping of data. In data filtering, the objective is to ignore some of the information present in a set of input tokens. The second part involves the grouping of multiple tokens to form an aggregate which is then represented by a single token.

The grouping algorithm for regions was discussed in section 3.1.1 as part of the histogram-based region segmentation algorithm [Beveridge 89]. Next we return to the perceptual line grouping algorithm for straight lines that was described briefly in section 3.1.3 . A similar theme has been used in [Reynolds 87] for perceptual grouping of rectilinear lines.

Hierarchical grouping of line segments

The framework for this algorithm in the VISIONS environment was developed by Boldt and Weiss [Boldt 87; Weiss 86]. They originally used the algorithm to extract straight lines. Since then, the algorithm has also been used in extracting curved lines [Dolan 89]. We will refer to the grouping algorithm developed by Boldt and Weiss as the line grouping algorithm.

The basis for the line grouping algorithm is the observation that in many cases a straight line can be viewed as a sequence of line segments such that the successive segments are roughly collinear and similar in contrast, and the entire sequence passes a straightness test. These criteria depend on the relative scale of the features. For example, long line segments can be separated by a larger gap than short ones. A sequence of line segments which is not straight at one scale can be part of a longer sequence that passes the same straightness test at a larger scale.

The input to the line grouping algorithm is a set of edges derived from the input by any process. For example, the edges could be derived with Burns' algorithm. In actual practice however, Boldt and Weiss used zero crossings in the output from applying a Laplacian operator on the input image, as the initial edge detection operator. The rest of the line grouping algorithm deals with grouping these edges in a multiple cycle process.

Each cycle of the line grouping algorithm consists of two steps: *linking* and *merging* or replacement. The linking step is used to improve the efficiency of the algorithm and reduces the burden of exhaustive search on the merging step, by searching for potential candidate line segments. During the merging or replacement step, two or more lines are merged and replaced by a single longer line segment based on a global straightness criterion. These two steps are repeated alternately on longer line segments, resulting in a hierarchical grouping.

The linking step consists of searching for pairs of lines that satisfy certain geometric and non-geometric measures that make them candidates for grouping. With suitable thresholds (that may vary with the scale), these measures are converted into a binary relation for pairs of lines to derive a *link graph*. In this graph, the vertices represent line segments at the particular level of hierarchy or scale, and the arcs represent potential line segment pairs that can be merged. The geometric measures used are collinearity and proximity (the two lines must be within a certain threshold called the *linking radius*, the endpoints must be close, the lines must be approximately collinear, and they must not overlap too much). The non-geometric measure is that the line segments should have similar contrast (gradient magnitude of the edge operator). The same linking process is repeated for each endpoint of every line segment. The result of this step is a directed graph as mentioned above. It

should be noted that, depending upon the scale during the line grouping algorithm, different thresholds are used for these measures.

The merging or replacement process consists of searching for potential line segments in the link graph and replacing a sequence of such segments by one longer segment. For every vertex (line segment) in the link graph, the replacement algorithm computes all sequences of line segments containing the vertex that are within a *search radius* or *perceptual radius*, which bounds the length of a sequence of line segments that is tested for straightness. Each sequence of line segments is approximated by a straight line, and if the straightest path for the vertex (line segment) in question passes a straightness threshold, that sequence is replaced by a straight line. The linking and replacement process is repeated at different scales, with the linking and perceptual radii increasing by a constant factor from one scale to the next.

At any scale, an unmerged line segment is copied to the next level of the hierarchy, until it fails to merge too many times (four times in the original implementation).

(iii) Constraint-based Graph Matching

This KS has multiple purposes. As the name suggests, it tries to find graph isomorphisms. For example, it can be used to match a *data graph* derived from an input image whose nodes represent image tokens and whose arcs represent token attributes and relations, with a *pattern graph* derived from stored object models and their attributes and relations. To be computationally feasible, this KS must be applied under suitable constraints so that the data graph and the pattern graph are small.

(iv) Token Relations

The Token Relations KS takes various line, region, and object tokens from the ISR and computes a variety of relationships between them. For example, it can be used to calculate region-line intersection, spatial relations between regions, etc.

(v) Knowledge-Directed Resegmentation

Knowledge-Directed Resegmentation is discussed in [Kohl 87]. It is also called the Goal-Directed Intermediate-level Executive (GOLDIE). Basically, GOLDIE invokes region segmentation and line extraction KSs with modified parameters that are more suitable for a

particular environment, under the control of the schema interpretation strategies. GOLDIE can be used to apply these low level KSs either to an entire image or to a subimage.

Next, we discuss what is perhaps the most important component of the intermediate-level vision: the intermediate token database.

3.3 Intermediate-level Symbolic Representation (ISR) Database

Intermediate level vision bridges the semantic gap between low- and high-level vision. It comprises two major components: A database system for storing intermediate symbolic representations of significant image events at multiple levels of abstraction, and a numeric as well as symbolic processing engine for various intermediate-level vision algorithms. Many aspects of the intermediate level in relationship to various knowledge sources have already been discussed, here we focus on the database system.

3.3.1 Intermediate Symbolic Representation (ISR)

The intermediate symbolic representation (ISR) provides database support for various intermediate-level vision algorithms. The ISR is significantly different from conventional textual and image databases, however, we shall not provide a comparative study. Instead, this section outlines the salient features of the ISR and its various components. Wherever necessary, ISR features will be compared with other databases. The details of ISR can be found in [Brolio 89; Draper 90]. Brolio et al. [Brolio 89] discuss the details of the older version of the ISR called ISR1, whereas Draper et al. [Draper 90] discuss the details of the most recent version of the ISR, called ISR2. ISR2 subsumes ISR1, and in our discussion, we shall refer to ISR2 as ISR.

In comparison to many database systems where data usually resides on disk, the ISR is an in-core database. Tokens that are used during an interpretation cycle are always kept in the memory, primarily to increase the efficiency and speed of the ISR. However, mechanisms are provided for explicitly storing and retrieving a particular core image of the ISR to and from the disk.

The ISR contains two primary types of objects: Tokens and Frames. A token in the VISIONS terminology is equivalent to a record in database systems. Each primitive token in the ISR corresponds to an image event extracted from the low level, such as a line segment

or a region. Associated with each token is a *set of features*, which are simply different properties or attributes of the token. For example, a line token's features could be its length, endpoints, orientation etc. Since each image is likely to have a number of primitive tokens such as lines and regions, a further structuring is imposed on the ISR, which is to group all tokens that share the same set of features. Such a group is called a *tokensequence*. An example is the tokensequence of all the regions from a region segmentation. Grouping tokens of the same type in a tokensequence is useful for subsequent perceptual grouping algorithms, because these algorithms often carry out operations on a set of tokens of the same class. Thus, a token is a class of object, of which there are many instances in a tokensequence. Tokensequences are the fundamental information storage mechanism in the ISR. Because of the large number of possible instances of a token type in a tokensequence (several thousand in many cases), it is infeasible to assign a name to each instance of a token type. Instead, all the instances are automatically assigned an index, since numbering is the simplest automatic naming scheme. A tokensequence can be viewed as a two-dimensional array. Each row in the array represents a token, and each column represents a property or attribute of that token. This 2-D structuring simplifies algorithms that access tokens by indexing, or associatively by certain feature values. Since this array can either be sorted by one of the features or by index, indexed retrieval is trivial in only one case (indexing by the token number, in case of ISR). Other cases, where additional computing is required, are called "dynamic indexing".

The second primary object type in the ISR is the frame. Frames are the building blocks of the hierarchical structure of the ISR. A frame is a named object, each with its own set of features. Frames are created by the user to group together different types of information at various levels of abstraction. For example, the user might create a frame that stores the parameters used in running the GOLDIE knowledge source for region segmentation. The relationship between tokensequences and frames is such that every tokensequence is part of a frame, and every frame has at most one tokensequence. Figure 3.5 shows the organization of the ISR. At the lowest level are the tokens². Two tokensequences are shown in the figure corresponding to line tokens and region tokens in the Road1 image. These tokensequences are part of the two frames: Regions and lines. Each of these frames might contain additional features such as average size of the regions, average length of the lines etc. The Regions and Lines frames are in turn, shown to belong to frame Road1. Road1 is an individual image of a general environment represented by frame Road, which might contain many individual road images. In addition, Road might contain additional information in the form of features

²Tokens don't necessarily have to be at the bottom level of the ISR. The user might create abstract tokens and make them part of higher-level frames.

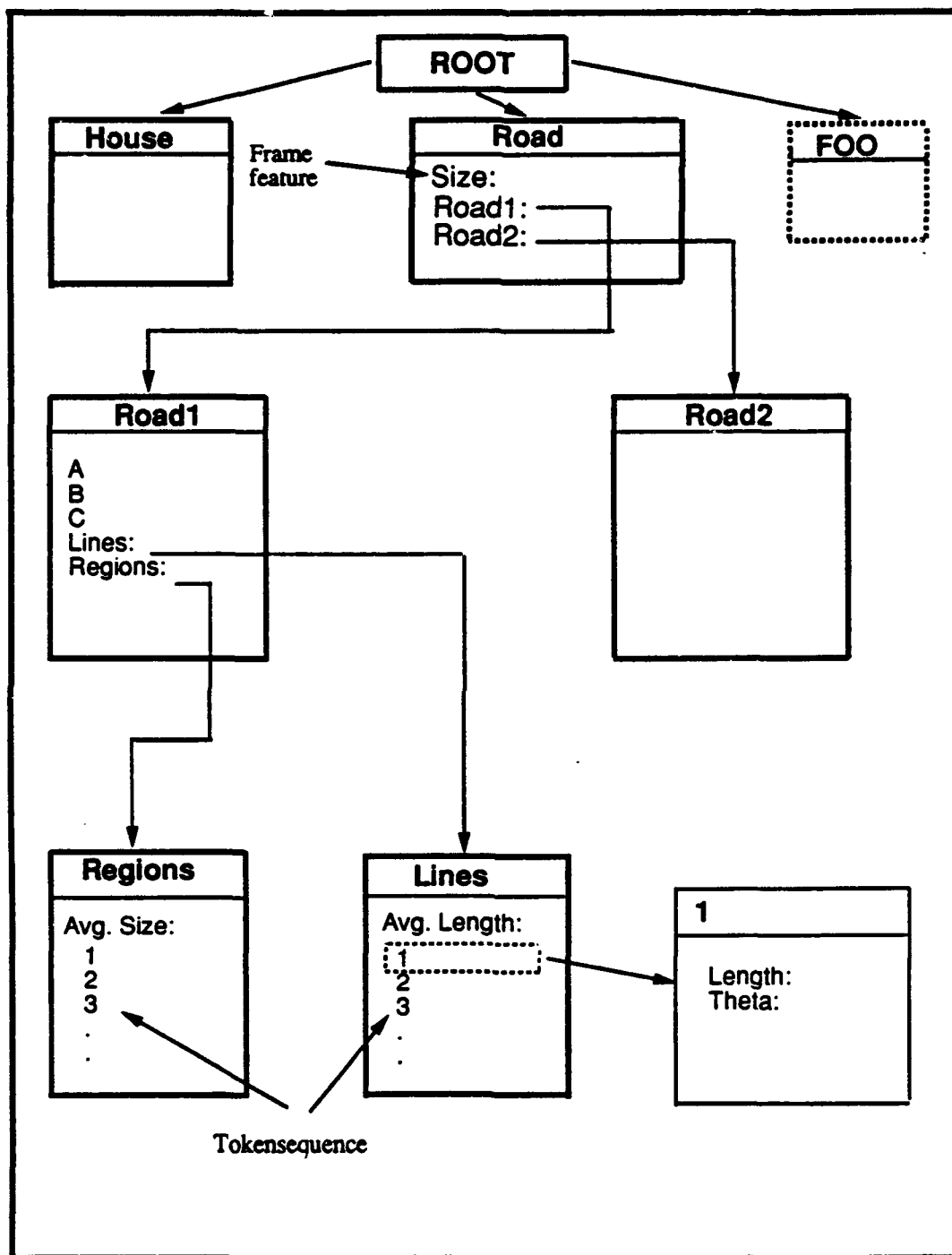


Figure 3.5. Frame and token hierarchy in ISR

that apply to all images and thus need not be repeated. It should be apparent from this discussion that the ISR is organized in a hierarchical manner. The ISR is however created from its root down. At the system initialization, there is an empty frame called ROOT. From here, the user can create a frame FOO and give ROOT a feature FOO whose value is a pointer to the new frame. In the same way, more frames can be attached to either ROOT or its children and so on. This mechanism calls for dynamic storage allocation as the frames and features are created.

Features

Features are where data values are stored in the tokens and frames. There are two different types of features, *frame features*, and *token features*. The frame feature is straightforward; it is a single value for a particular feature name in a frame. Examples of frame features in figure 3.5 are Avg Size in the Regions frame, Lines in the Road1 frame, Size in Road frame etc. Referring back to the organization of a tokensequence as a 2-D array, we find that each token feature (e.g. length) there is a value for each token in a tokensequence. Thus a frame feature is characterized by a single value, whereas a token feature is characterized by a sequence of values.

Two important components of features; their *data types* and *facets* are discussed next.

Feature data types

The ISR features are one of seven data types: Boolean, Integer, Real, String, Array, Pointer, and Handle. The first four data types are self explanatory. The array data type is supported in the ISR only as far as storage and retrieval. No predicates are allowed on the Array data type. A Pointer points to an object. For example, the feature Lines in the frame Road1 points to another frame called Lines. A Handle in the ISR is another form of pointer. There are various types of Handles depending upon what they are pointing to, but for most purposes they are identical. In general, pointers are used for constructing the hierarchical structure of the ISR, and Handles are used to refer to different entities in different frames in conjunction with operations defined with them. For example, in associative access of token features, a range for index or feature values could be specified with a handle. The most obvious types of handles are frame handles and token handles which refer to frames and tokens respectively. A *frame feature handle* points to the value of a frame feature. The most

useful handle is the *token feature handle*, which points to a sequence of values corresponding to a feature in a tokensequence in a frame.

All the ISR feature data types are augmented with an additional pair of values: Uncalculated, and Undefined. These special values are used to denote that a feature value has not been calculated yet or is undefined for this object. For example, a feature whose value is expensive to compute, might leave that value as Uncalculated until it is actually needed. In some cases a feature value cannot be computed, such as the orientation of a point line, so the value is set to Undefined.

Feature facets

In addition to a name and a value or sequence of values, every feature has five additional facets or components: data type, documentation, if-needed, if-getting, and if-setting. Feature data types were discussed above. Documentations are strings supplied by the user. The last three facets are lists of *demons* that should be triggered when these feature values are accessed. Demons are attached procedures or functions that are activated whenever a value is requested, or modified, or when a value is needed that has not yet been computed. The if-needed facet contains functions that can be used to calculate the value of the slot³ corresponding to the feature. The if-getting facet is a list of functions to be invoked whenever a feature value is accessed. The if-setting facet is a list of functions to be invoked whenever a feature value is changed. The if-setting facet can also be used to block the storage of a new value. Next we discuss some important operations performed in the ISR.

Tokensequence and Associative access

One important mechanism for accessing tokens is by their values rather than by their index. This associative access mechanism is further supplanted in the ISR with a number of set operations (union, intersection, set-difference) over different subsets of token features to assist the high-level strategies in operating over sets.

A *tokensubsequence* (TSS) is a set of tokens, which is a subset of a tokensequence. A TSS is a type of handle, which in conjunction with one of the facets (if-needed etc.)⁴, can be used

³A slot is a form of feature value where, in place of a data value, a number of procedures or functions are stored to carry out operations and return a value whenever that slot is accessed.

⁴Note that if-needed is a demon.

for a variety of associative operations on different token sequences.

Virtual features

Virtual features are useful, for example, in transforming lines from a rectangular coordinate system to a polar coordinate system. With suitable handles, a set of virtual features can be created that are computed only when desired.

Spatial queries

Often it is more useful or even necessary to group data into "buckets" according to spatial location in order to perform efficient retrieval. Knowledge sources such as line extraction and region segmentation make use of spatial queries.

The simplest form of static spatial indexing used in the ISR is the regular grid. A grid is laid down over the image, and a token is stored in each cell of the grid that it touches or intersects. This can be done by making a grid frame containing information about the grid such as the cell size, and a set of tokens, each of which represents one cell of the grid. The token features are TSS's of different frames, denoting which token from each frame intersects which cells. Variations of this grid scheme are possible.

After discussing the ISR, we next discuss the architectural characteristics and requirements of the ICAP level of the IUA.

3.4 Architectural characteristics of the ICAP

As a parallel processor, the ICAP tasks can be divided into the following categories:

- As an attached processor to the CAAPP
- As an attached processor to the SPA, and
- Intermediate-level vision tasks

3.4.1 As an attached processor to the CAAPP

As an attached processor to the CAAPP, the ICAP performs tasks in one of the following categories:

- Exchange data or control with the CAAPP
- Share load with the CAAPP

Exchange data or control with the CAAPP

The ICAP interacts with the CAAPP to either exchange data with it, or pass control information down. For example, the CAAPP would pass tokens, generated by the application of one of the low-level vision algorithms of section 3.1, to the ICAP during initial bottom-up processing. The ICAP would either pass local control information or data to the CAAPP during later stages of image interpretation. For example, the Goal-Directed Intermediate-level Executive (GOLDIE) knowledge source (also called Knowledge-directed resegmentation) modifies parameters for CAAPP region and line segmentations to be more suitable for a particular environment, under the control of the schema interpretation strategies. In these cases, the amount of data exchanged between an ICAP PE and the CAAPP PEs under it may vary between a single bit to hundreds of bytes. In the initial bottom-up processing phase, the most suitable way for the ICAP to retrieve tokens from the CAAPP is to operate in synchronous mode. Again, depending upon the operation, it might be desirable to either operate the ICAP in a fine-grained SIMD-like mode, or in a medium grained SMIMD mode. During later stages of image interpretation, where local conflicts or ambiguities are resolved, it might be desirable to operate the ICAP PEs in a less centralized manner. Depending upon the task, the ICAP could operate in SMIMD or pure MIMD mode.

To summarize, while exchanging data or control with the CAAPP, it is desirable to be able to operate the ICAP in all of the modes outlined in Chapter 1.

Share load with the CAAPP

The ICAP can be used to compliment the CAAPP in low-level processing. The CAAPP uses bit-serial processing elements with limited memory. Many low-level vision tasks use algorithms such as Fourier transforms (FFT); for example, in texture analysis. Often, these tasks require floating point arithmetic. The FFT is a staged (alternating computation and communication), fine-grained, communication and computation intensive algorithm. It requires apriori interprocessor communication, but the distance (in terms of PE index) between the processors that exchange data varies as 2^i ($0 \leq i \leq \frac{\log N}{2}$, where N is the smaller of the number of processors or the number of points in the FFT) between stages of computation. A 1K fixed-point FFT on the CAAPP takes over 10mS. We currently do not have a figure for a floating point FFT at the CAAPP level. However, to get an idea of the time a floating point FFT might take at the CAAPP level, it should be noted that a floating point operation takes on the order of 100 times as long as for a fixed point operation on a CAAPP PE. A 1K fixed-point FFT on a single TMS320C25 takes less than 3mS. The second generation IUA (to be discussed in Chapter 6) uses TMS320C30 floating-point DSP chips at the ICAP level. A 1K floating-point FFT on a single TMS320C30 also takes less than 3mS. Notice from table 3.1 that many low-level vision algorithms might use matrix arithmetic; for example, image rotation. In many cases, because of the large dynamic range of intermediate results, floating point arithmetic may be required.

Therefore, such operations may be off-loaded to the ICAP by the CAAPP. FFT and matrix-based computations are fine-grained operations. They use regular, apriori interprocessor communication, that depend on the stage of computation. A SIMD-like synchronous mode with minimum communication setup overhead is highly desirable for the ICAP to efficiently support these operations. Also, notice that a wide variety of numeric and graph algorithms useful in low-level vision tasks have been demonstrated on shuffle-exchange, various meshes, hypercube, and similar regular networks [Chu 89; Dekel 81; Kuhn 80]. Many of these networks are incompatible with one another; for example, meshes are very inefficient in emulating shuffle-exchange type networks [Snyder 82]. Therefore, merely adding a capability for SIMD-like synchronous operation to the ICAP is not sufficient. Its communication network must be efficient in supporting a wide variety of (sometimes incompatible) communication patterns.

In addition to complimenting the CAAPP in floating-point operations, the ICAP can also be used to share computational load with the CAAPP in various low-level algorithms. For example, in the region segmentation algorithm of section 3.1.1, the ICAP could be

used in the second part of the region merging algorithm, where computation as well as communication is often non-uniform and dependent on the regions extracted in the first part. A SMIMD mode with limited autonomy or an MIMD mode with full autonomy for the ICAP are desirable in this case. Similarly, Boldt's algorithm in section 3.1.3 involves data dependent and non-uniform communication in the linking and merging steps (it depends on the graph structure formed in the linking step of the previous phase). A SMIMD or MIMD mode of operation of the ICAP is desirable to support operations in this algorithm.

To summarize, while sharing load with the CAAPP, it is desirable to be able to operate the ICAP in all of the modes outlined in Chapter 1.

3.4.2 As an attached processor to the SPA

As an attached processor to the SPA, the ICAP performs tasks in one of the following categories:

- Exchange data or control with the SPA, and
- Share load with the SPA

Exchange data or control with the SPA

The SPA interacts with the ICAP to either exchange data with it or pass control information or queries down. For example, the ICAP would pass abstract tokens generated at the intermediate level either as a result of grouping processes or as response to specific requests from the high level. The SPA would pass control information for knowledge-directed processing and grouping operations, and queries in terms of grouping and matchings in the ISR tokens. The amount of data exchanged between an SPA and the ICAP PEs under it may vary between a few bytes to thousands of bytes. In the initial bottom-up processing phase, the most suitable way for the ICAP to pass abstract tokens to the SPA is to operate in a SMIMD mode with limited autonomy. During the later stages of image interpretation, where the ICAP responds to queries from the individual PEs of the SPA, it is desirable to operate the ICAP in an MIMD mode such that multiple simultaneous requests may be satisfied. The interprocessor communication at the ICAP level in these cases, in general, is data dependent and irregular.

To summarize, while exchanging data or control with the SPA, it is desirable to be able to operate the ICAP in SMIMD and MIMD mode with low latency interprocessor communication.

Share load with the SPA

The SPA is a collection of symbolic processors, that are not geared towards "number crunching." As pointed out earlier, one task of high-level vision is to construct 3-D models from domain independent knowledge and generate scene-specific 2-D projections of these models. These projections involve polynomial evaluation as well as floating point matrix computations. Notice that polynomial evaluation can be performed using FFT techniques [Sedgewick 88]. To generate these projections efficiently, the SPA might offload these computations onto the ICAP.

As in the case of sharing load with the CAAPP, while performing these operations, a SIMD-like or a SMIMD mode with low communication setup overhead is most suitable for the ICAP.

3.4.3 Intermediate-level vision tasks

In the knowledge-based image understanding paradigm in the VISIONS group, the ICAP level roughly maps to the intermediate level of vision. Projections (2-D) of the instantiated models at the high level meet here with the extracted image events from the low level. Often, the image events (tokens) from the low level must be further grouped together or organized before they can be matched with the models from the high level. Similarly, models at the high level may have to be transformed before they can be matched with the events derived from the scene. Intermediate-level vision tasks for the ICAP can be divided into the following categories:

- Grouping
- Model transformation
- Matching, and
- Distributed server for the SPA

Central to all ICAP operations is the symbolic token database, ISR, where the operands are stored.

Grouping

Once the initial tokens are generated by the CAAPP, the ICAP uses them in organizing the lowest level frames of the ISR. In many cases, the ICAP may even be involved with the CAAPP in generating the lowest level tokens; for example, in region segmentation and in Boldt's algorithm. The initial organization of the ISR can be carried out in a synchronous manner under central control. The interprocessor communication can be either apriori or data dependent. The lowest level ISR tokens are edges, regions, flow vectors, etc., where individual items are small in size, but there can be a very large number of them in a frame. As such, a SIMD-like and a SMIMD mode of operation with low communication overhead are desirable for the ICAP.

The next step after initial organization of the ISR, is to apply various perceptual organization and grouping algorithms to the primitive tokens, to generate more abstract tokens or collections for matching. The groups of tokens may be organized in the form of various data structures, such as lists, trees, or graphs. One example of grouping is in Boldt's algorithm, where collinear line segments may be grouped across the entire image to extract longer lines that are partially occluded. The line segments are organized as a graph structure in this algorithm. Another example of grouping is in the feature-based classification knowledge sources of section 3.2.2. The basic idea is to group the tokens with respect to a multitude of common properties such as color, texture, label etc. Thereafter, based on some feature value or an example token (e.g. a region in the EXEMPLAR KS in section 3.2.2), it is possible to classify other events in the image as having that property or appearing similar. This type of operation is performed in generating the initial hypothesis about the scene and its objects, and in servicing queries from the high level in the later stages of knowledge directed interpretation. The algorithms used for these tasks are similar to graph traversal and graph automorphism algorithms.

To perform grouping operations effectively, the ICAP must provide efficient means for the access and organization of tokens. Since tokens might be grouped across an entire image, and their density varies, the ICAP must support non-uniform communication and computation. To be able to support this kind of computation, the ICAP must provide semi-autonomous (SMIMD) as well as fully autonomous (MIMD) modes of operation with low communication

overhead.

Model transformation

This task was partially discussed in section 3.4.2. Part of model transformation takes place at the SPA level. Further, because of the ambiguities and missing data in the input image, it may be necessary to instantiate a large number of models for one scene. We stated earlier that certain model transformations based on FFT techniques and matrix arithmetic are best suited for SIMD-like synchronous processing at the ICAP. When a large number of models are involved for a single scene, it might be desirable to initially operate the ICAP in MIMD mode where each PE handles a subset of the models, and subsequently operate in a loosely synchronized mode to combine the results of all transformations and matchings.

Matching

Matching is a crucial part of intermediate-level vision. The basic idea here is to try to match instantiated models from the high level with the tokens generated by grouping tasks. In general, the token structures derived from the image will have both missing pieces and extra components in comparison with the instantiated models from the high level. In addition, the structures derived from the image may provide more than one alternative for each element, or even an associated level of uncertainty that must be considered in the matching process. For these reasons, rather than forming an isomorphism, matching is usually treated as an optimization problem. Depending upon the representation of the instantiated models and the image events, the matching may be graph-based or matrix-based (as in linear programming).

Computation and communication in matching is fine-grained in nature. Since the image tokens may be spread across an entire image, the communication is data dependent and non-uniform in general.

Distributed server for the SPA

In order to verify existing hypotheses, establish new ones, or resolve conflicts during

the later stages of image interpretation, the schema system at the high level may request additional knowledge-directed processing at the lower levels. Since the schema system operates in a shared memory MIMD mode, the requests from the high level are non-deterministic in time. In response to the requests from the high level, the ICAP basically performs operations in one of the classes outlined so far. To serve potentially multiple requests simultaneously from the high level, it is desirable to operate the ICAP as a distributed server, or in an MIMD mode. It should be noted, however, that in response to a query from the high level, the entire ICAP or some portion of it may have to carry out complex computations on a group of tokens associated with multiple ICAP processors. Therefore, as a follow up to the query from the high level, all or some of the ICAP may have to operate in synchronous manner (SIMD-like or SMIMD).

Weems et al. [Weems 91] have carried out a study of architectural requirements of image understanding with respect to parallel processing. To further add to the requirements of the ICAP level, we reproduce table 3.2 from the cited paper.

Table 3.2. Characteristics and requirements of the intermediate level

Computation	Communication
Medium grained	Medium messages
16-bit integer arithmetic	Global broadcast
32-bit integer arithmetic	Subset broadcast
32-bit floating point arithmetic	Global summary
Symbolic processing	Down to low level
Record processing	Local neighbors
Compare/matching	Over lists
Graph processing	Over graphs
Geometry	Collection (grouping)
Matrix arithmetic	Dense routing
	Up to high level
Control requirements	Data types and structures
Medium grained	8-bit bytes
Related threads	16- and 32-bit integers
Independent threads	32-bit floating point
Associative select	Records and arrays
Synchronous	Linear lists
Asynchronous	Linked lists
Central control	Trees
Distributed control	Graphs
High level control	Symbolic formulas

Having discussed the characteristics and requirements of the ICAP level, we discuss the architectural requirements of the ICAP communication network to support them.

3.5 Requirements of the ICAP communication network

Based on the architectural characteristics and requirements of the ICAP level to efficiently support intermediate-level vision, we summarize the architectural requirements of the ICAP communication network as follows.

- It should have low latency, high bandwidth, and high common access throughput, especially in real-time applications
- It should have the ability to support low-overhead SIMD-like synchronous routing, under central control
- In SIMD-like routing, it should be equally efficient in supporting both regular and irregular communication patterns. In other words, it should not have a bias towards one communication pattern over others
- It should have the ability to support data-dependent synchronous routing under the SMIMD mode of ICAP computation
- It should have the ability to support data-dependent asynchronous routing under the MIMD mode of ICAP computation, and
- Most importantly, it should have a capability for rapid reconfiguration to efficiently support all of the above requirements

Recall that these requirements were also discussed in the first chapter. In the next chapter, we discuss the first stage of our solution for providing a "good" communication network for the ICAP.

CHAPTER 4

GENERATION 10: CENTRAL ROUTING CONTROL

In this chapter we address the requirements of a multiple-processor network when the target parallel processor is used in a SIMD mode (or SIMD-like, as in the case of the ICAP) or in a synchronous MIMD (SMIMD) mode. As discussed in Chapter 2, there is an obvious problem with a SIMD or a SMIMD multiple-processor system with a fixed static interconnection network in that it is optimal for only one interprocessor communication pattern. On the other hand, when dynamic interconnection networks are superimposed on static networks, a penalty is paid for non-local communication in the setup and the routing control of the superimposed dynamic network. Examples of such mechanisms can be found in the Connection Machine [Hillis 85], the MasPar [Grondalski 87; MasPar 90], and the CAAPP level of the IUA [Weems 89; Herbordt 90].

We alleviate this problem in stages. In this chapter we address the requirements of a multiple-processor network when the interprocessor communication patterns in the target parallel processor are fixed and known apriori. Examples of tasks that use fixed interprocessor communication patterns are the Fast Fourier Transform (FFT), convolution, template matching, matrix algebra, some sorting algorithms, some search algorithms, and many scientific problems dealing with partial differential equations and finite element analysis. Tasks of these kinds often have unique optimal communication patterns. For example an FFT data flow graph is incompatible with a 2-D mesh. If the target multiple-processor system is connected in a 2-D mesh topology, each data exchange stage of the FFT graph may require $\theta(\sqrt{N})$ steps instead of $\theta(1)$ if the parallel system was connected in a butterfly topology. Similarly a bit reversal permutation is incompatible with a shuffle-exchange network [Stone 87, pp 313].

To alleviate the problem of latency due to mismatch between the communication pattern of a task graph and the network of the system in the restricted context when all of the communication patterns are fixed and known apriori, we describe the construction of three classes of dynamic connection networks. The three classes are crossbar, strictly non-blocking, and rearrangeably non-blocking networks. All three classes can efficiently support the N^N mappings property defined in Chapter 1.

To reduce the latency in the network in terms of the number of links and nodes between an input-output pair, we choose moderately large crossbar-based nodes (switching elements) instead of the 2×2 switches often used for constructing these networks. In the next subsection, we describe the architecture of a crossbar-based node called the PARallel COmmunication Switch I (PARCOS I). In subsequent subsections we describe how PARCOS I can be used for the construction of networks in the three classes, and then provide an analysis and evaluation of the design.

4.1 Parallel Communication Switch I

In this section we describe the architecture of a VLSI chip that was designed as the building block for the ICAP communication network in the first generation IUA. The design concepts used in the VLSI chip, which we call PARCOS I, form the basis for construction of the three classes of networks to be discussed later in this chapter. It should be noted that even though the following discussion pertains to a PARCOS I chip of a specific size, the basic concepts can be used to build PARCOS I chips of different sizes. Various limitations to the design will be discussed in a later section.

The PARCOS I chip organization is shown in figure 4.1. The chip consists of a communication matrix of 32 bit-serial inputs and 32 bit-serial outputs, a control memory, a set of registers and associated read/write circuitry.

The communication matrix of PARCOS I consists of 32 tree-structured multiplexers, each of which is a 1 of 32 multiplexer. All 32 input lines are connected in parallel to each of the 32 multiplexers. With this architecture, multiple outputs can be connected to the same input, providing *broadcast* mode capability. Figure 4.2 illustrates one multiplexer tree. It will be noted that there are two multiplexer trees, one made out of n-channel MOSFETs and the other made out of p-channel MOSFETs, with their outputs connected together. By properly sizing the two types of MOSFETs, we achieved near equal delays for both low-to-high and high-to-low transitions at the output. For any multiplexer, path selection at any level of the tree is done with a single bit of a control word. The true value of the control word bit is used to select a path in the n-channel multiplexer tree, and its complement value is used for the p-channel multiplexer tree. Thus, 5 control bits are required to select 1 of 32 inputs for each multiplexer, or $32 \times 5 = 160$ bits for configuring the entire communication matrix.

The PARCOS I control memory consists of 32 control words, where each control word contains the 32 bytes of 5 bits required for one configuration. The on-chip control memory

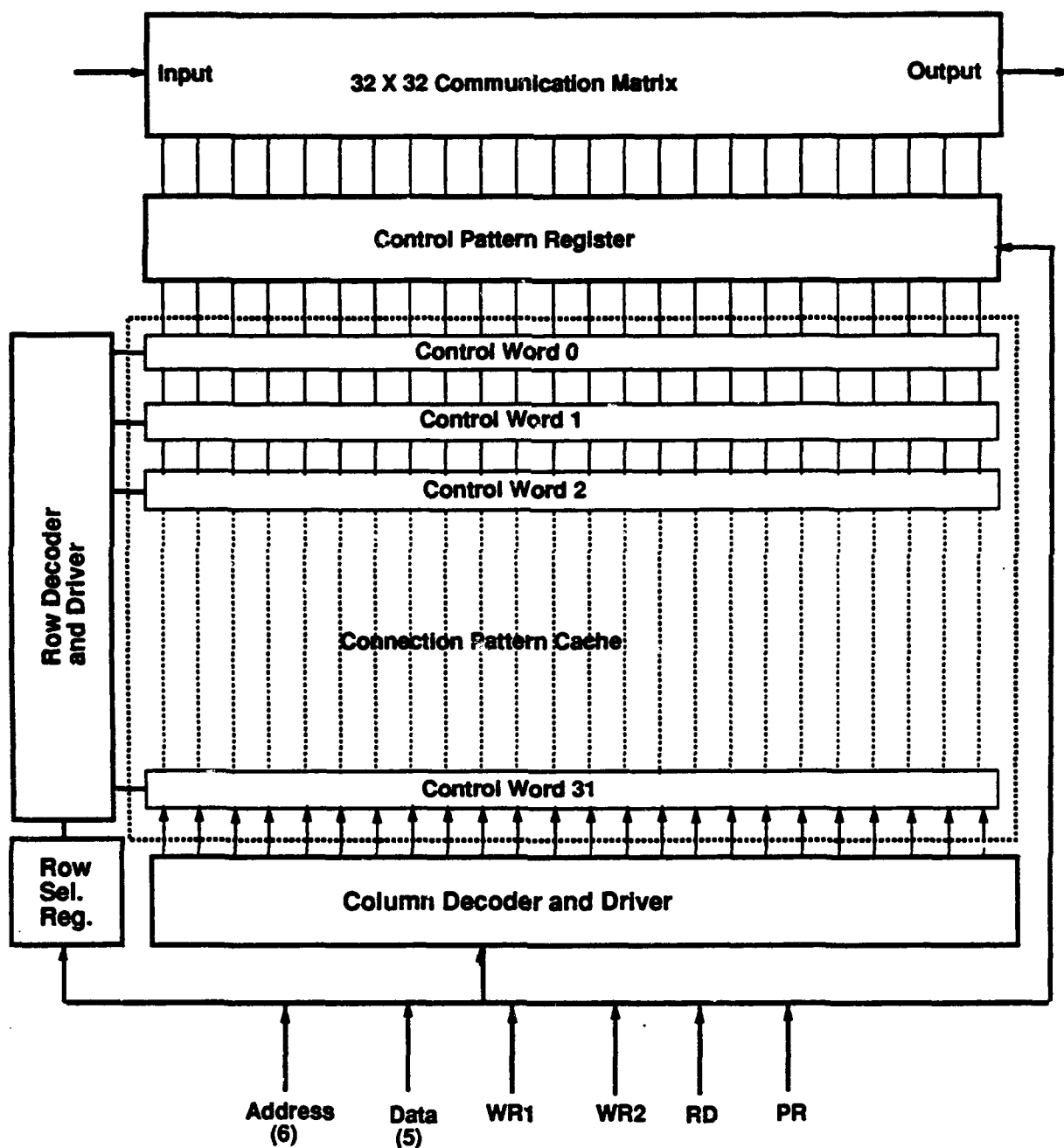


Figure 4.1. Organization of the PARCOS I chip

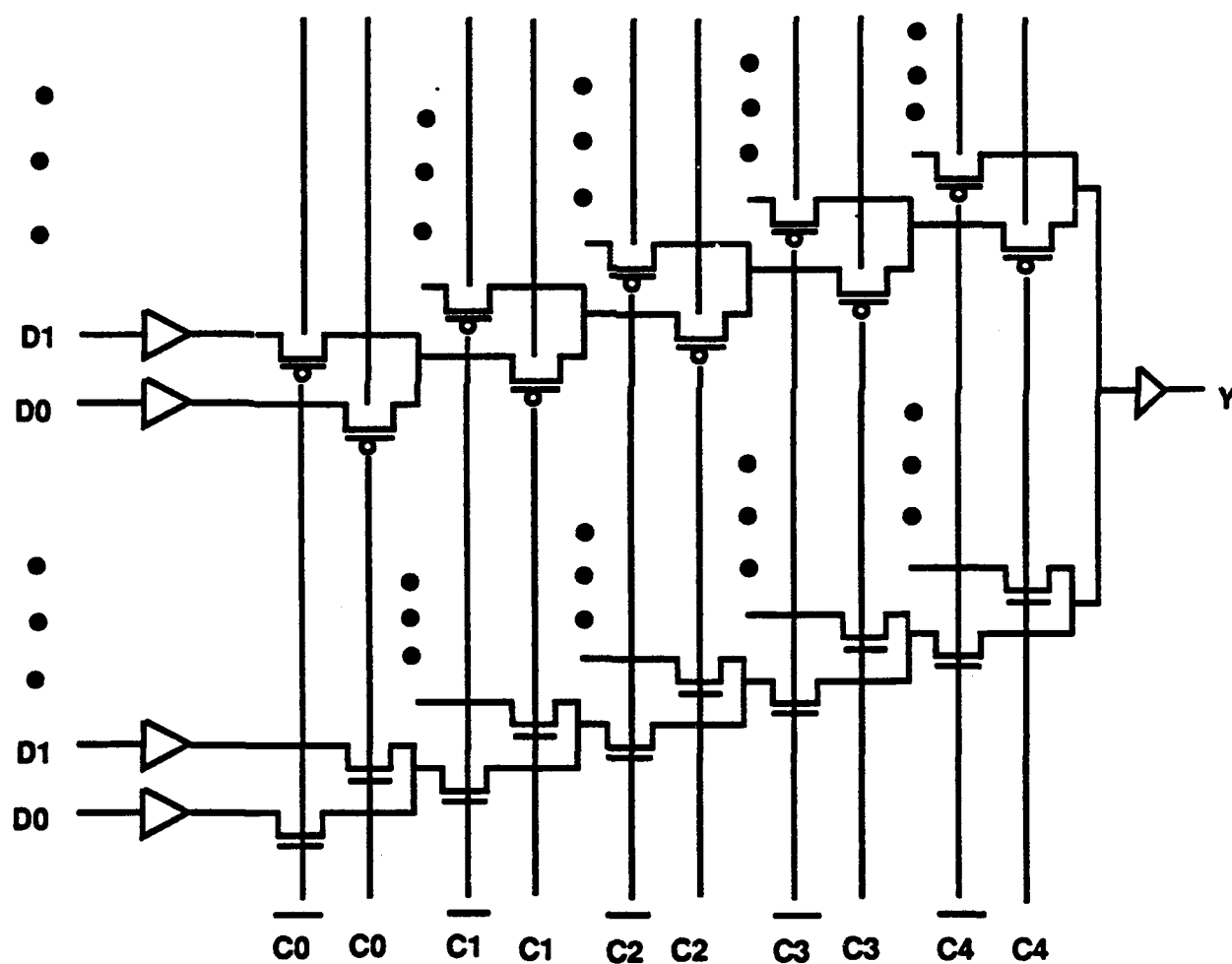


Figure 4.2. Organization of a multiplexer tree

is therefore constructed so that PARCOS I can hold up to 32 of the most frequently used connection patterns for larger networks built out of this chip. The control memory is called the Connection Pattern Cache (CPC), because it is analogous to storing the most frequently used pages in a memory system cache.

The connectivity information for the communication matrix is stored serially into the control words. To write connectivity information in a control word of the CPC, first a row number is set in the Row Select Register (RSR). RSR is mapped into the chip's memory address space, allowing the address bus in PARCOS I to select the register as a memory location. In PARCOS I, the RSR is mapped to address 32. When data is written to address location 32, the binary value on the data lines determines the row number selected in the RSR. Once a row number is selected in the RSR, 32 5-bit bytes are written into addresses 0 - 31. The memory location's address is the output port number and its contents determine which input port it is connected to. If only a subset of links need to be modified, this can be done by selectively writing only into locations corresponding to those links.

Reswitching the configuration of the communication matrix from one stored connection pattern in a control word to another requires a single write instruction, where the address of a new control word is placed in the RSR, and the control word's contents are loaded into the Control Pattern Register (CPR), activating a new connection pattern in the communication matrix. Notice that the CPR allows a control word to be modified in the CPC without disturbing an existing configuration in the communication matrix. In many cases this feature allows the time to write a new connection pattern from the central controller into the CPC to be hidden while the processors are working on an algorithm.

PARCOS I is implemented on a single 84 pin, 50000 transistor VLSI chip. It is a full custom design, built out of a 2 micron, P-Well, double metal, CMOS technology available through the MOSIS fabrication service. Each CPC memory bit is a 6 transistor static RAM cell. The RAM cell provides both the true and the complement value of its contents on the two bit lines, eliminating the need for separate circuitry to generate the complement value for the multiplexer tree. The memory cycle time has been measured to be 100nS. The worst case delay in broadcast mode from one input to 32 outputs is less than 50nS. A microphotograph of the chip is shown in figure 4.3.

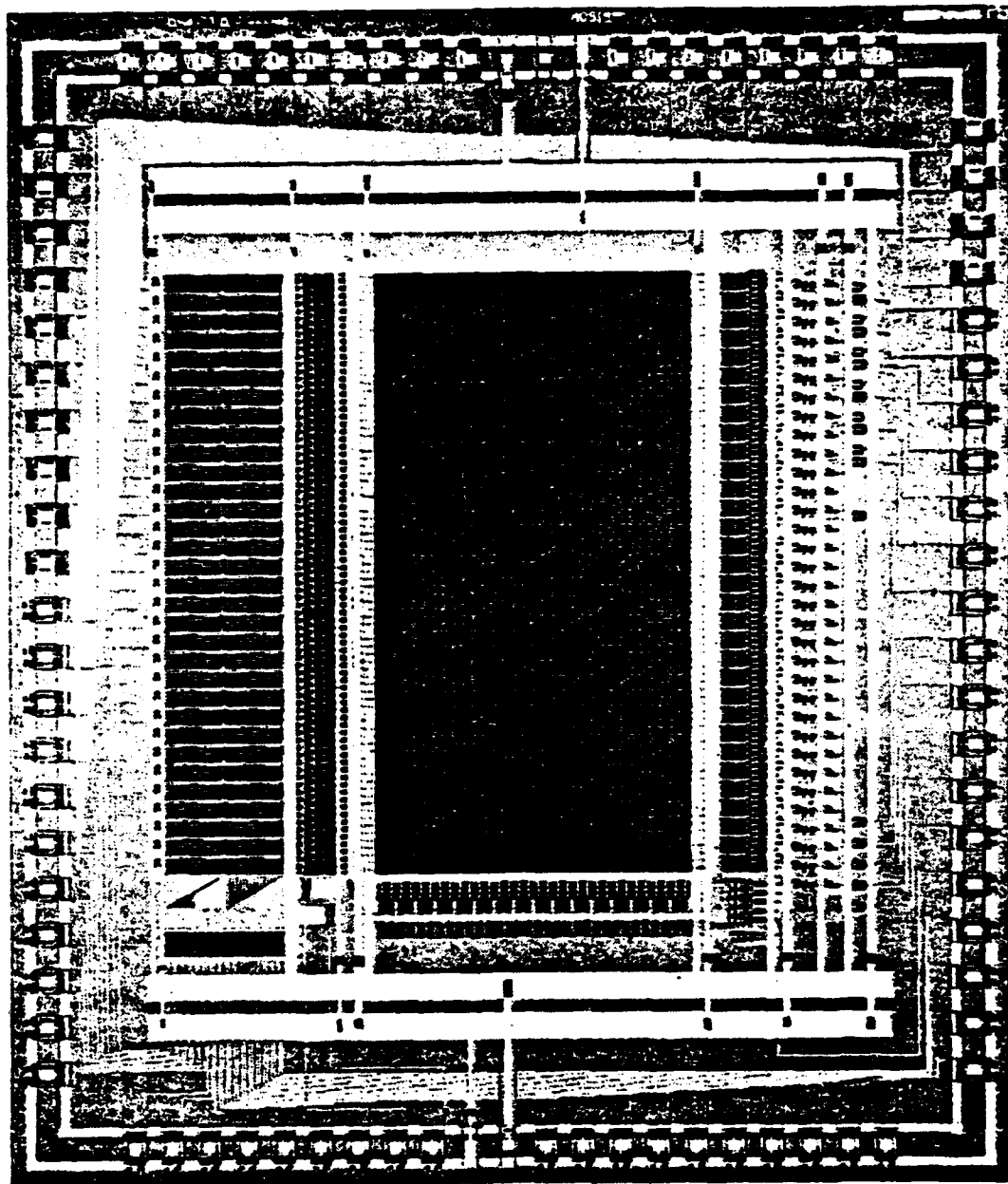


Figure 4.3. Microphotograph of the PARCOS I chip

4.2 IUA GEN I ICAP communication network

In this section we describe how to build larger crossbar networks with PARCOS I chips. This is the first of the three classes of dynamic connection networks that we will explore in this chapter. In this section we describe the architecture of a 64×64 crossbar network built with 32×32 PARCOS I chips. The 64×64 network is used as the ICAP interprocessor communication network in the first generation IUA called *IUA GEN I*.

4.2.1 ICAP communication network architecture

The ICAP level of the IUA GEN I is a parallel processor, built with 64 fast digital signal processing (DSP) chips (Texas Instruments TMS320C25). Each DSP chip has one serial input port and one serial output port, each of which is capable of a 5M-bit/sec data rate. These serial ports provide the basis for interprocessor communication within the ICAP level of the IUA GEN I and as such they form the set of data sources and sinks that are linked by the network described here.

The 64-input, 64-output connection network for the IUA GEN I uses 2 stages of 32×32 PARCOS I chips. The PARCOS I chips are connected to make a 64×64 crossbar switch with broadcast capability as shown in figure 4.4. The second column of PARCOS I chips is essentially used as a set of 2-input 1-output multiplexers. This connection structure provides a simple addressing scheme for setting up a connection pattern. The PARCOS I chip is capable of broadcasting, allowing the connection network to realize any of the possible N^N mappings of its input ports onto its output ports in a single pass. All of the processors can send and receive data on their links at the same time. These links can be changed at any time by the IUA's central controller called the Array Control Unit (ACU).

4.2.2 Network setup and re-switching

The on-chip control memory in the PARCOS I chips permits storage of up to 32 of the most frequently used input-output connection patterns for the whole network. The method for programming a new pattern in the IUA GEN I ICAP connection network is as follows. In the network of figure 4.4, two 5-bit bytes are used to specify which input an output is connected to. The first (most significant) byte selects an input to two of the PARCOS I chips

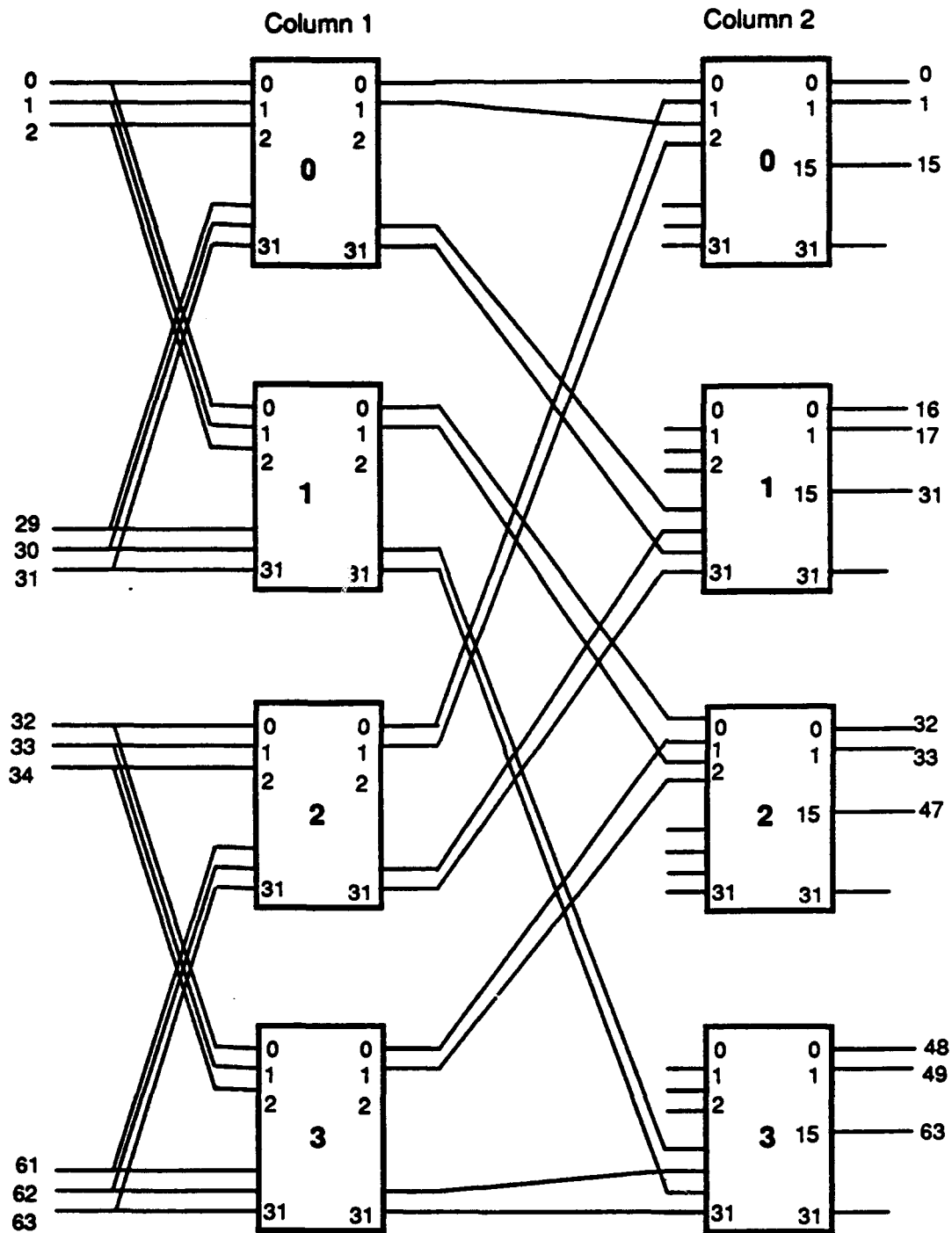


Figure 4.4. IUA GEN I ICAP Communication Network

in the first column. In the second column, the second byte selects one of the two resulting outputs from the first column. The RSRs of all 8 chips are mapped to the same memory address such that writing a row number at this address selects the same row in the control memory of all of the chips. As an example, suppose that it is desired to connect input 30 to output 33. In base two this mapping would be (011110) $\rightarrow$ (100001). The instruction issued by the ACU to the connection network board looks like:

Instruction Code	Input Address	Output Address

The two most significant bits of the output address are used by hardware to select one of the PARCOS I chips in Column 2 (chip #2 is selected in this case). The 4 least significant bits of the output address are used to select the proper output in the selected PARCOS I chip in Column 2. As mentioned earlier, the PARCOS I chips in the second column are used as a set of 2-input 1-output multiplexers. In any of the 4 chips in the second column, an output X is connected to only one of the two inputs: $(2 * X)$ or $(2 * X + 1)$ which is determined by the most significant bit of the input address. Output 0001 is connected to input 0010 in chip #2 in Column 2 in this example. Notice that this selection can be done by simple hardware. Also, the most significant bit of the input address (0 in this case) is used to select one of the two PARCOS I chip pairs 0-1 or 2-3 in Column 1 (PARCOS I chip pair 0-1 is selected in the example). The 5 least significant bits of the output address select the output port in the chosen PARCOS I chip pair, whereas the 5 least significant bits of the input address select the desired input to the PARCOS I chip pair in the first column. All of the address and data values can be derived and written into the proper crossbars by the hardware with a single instruction from the ACU. Thus, in one instruction time the ACU can set up a connection from one output of the connection network to a desired input. Filling a control word (Storing a complete connection pattern) requires that 64 connections be specified, each with a separate write instruction from the ACU. Another word can be selected for writing by loading a new row number into the RSR and repeating the above operation. If only a subset of a connection pattern in the CPC needs to be changed, the ACU can selectively change just those links to be modified. Selective rewriting is faster than rewriting the whole pattern and is advantageous in cases where, as an algorithm progresses, the communication pattern changes incrementally. Since all of the row-select registers are mapped to the same address, a new connection pattern stored in the CPC can be activated with just one instruction from the ACU.

4.3 Larger crossbar networks

In this section we describe how to build a crossbar network of arbitrary size out of a smaller PARCOS I chip and how to program it.

4.3.1 Network architecture

Instead of restricting the PARCOS I chip to a fixed size, we assume that it implements an $n \times n$ crossbar function. The way to build larger $N \times N$ crossbar network is outlined in figure 4.5. The $N \times N$ crossbar comprises two logical stages. For $N = m \times n$, the first logical stage comprises an $m \times m$ matrix of PARCOS chips. Groups of n inputs are fed in parallel to m PARCOS chips in every row. In the second logical stage of the bigger crossbar, one output from every PARCOS chip in a column is fed to a m -input 1-output MUX made out of PARCOS chips. Therefore there are N m -input 1-output multiplexers in the second logical stage.

The first logical stage of the $N \times N$ crossbar requires $m \times m = m^2$ PARCOS chips. The number of chips required in the second logical stage and the whole crossbar are calculated as follows. We assume for the sake of simplifying notation that N , n , and m are all powers of 2.

- For $1 < m \leq n$, each PARCOS chip can be used as n/m m -input 1-output multiplexers. Thus the total number of PARCOS chips required in the second stage is $N/n/m = N \times m/n = m^2$. The total number of PARCOS chips required for the entire crossbar then is $2 \times m^2$.
- For $n < m \leq n^2$, each MUX in the second logical stage will be constructed as a two physical stage MUX. The first physical stage of every MUX (closer to the inputs) will require m/n PARCOS chips. The second physical stage of every MUX (closer to the outputs) will implement m/n -input 1-output MUX. The total number of PARCOS chips required for the entire crossbar will be

$$(m^2 + N * m/n + N/n/m/n) = 2 * m^2 + m^2/n = \left(2 + \frac{1}{n}\right) (m^2)$$

- For $n^2 < m \leq n^3$, the total number of PARCOS chips required for the entire crossbar is

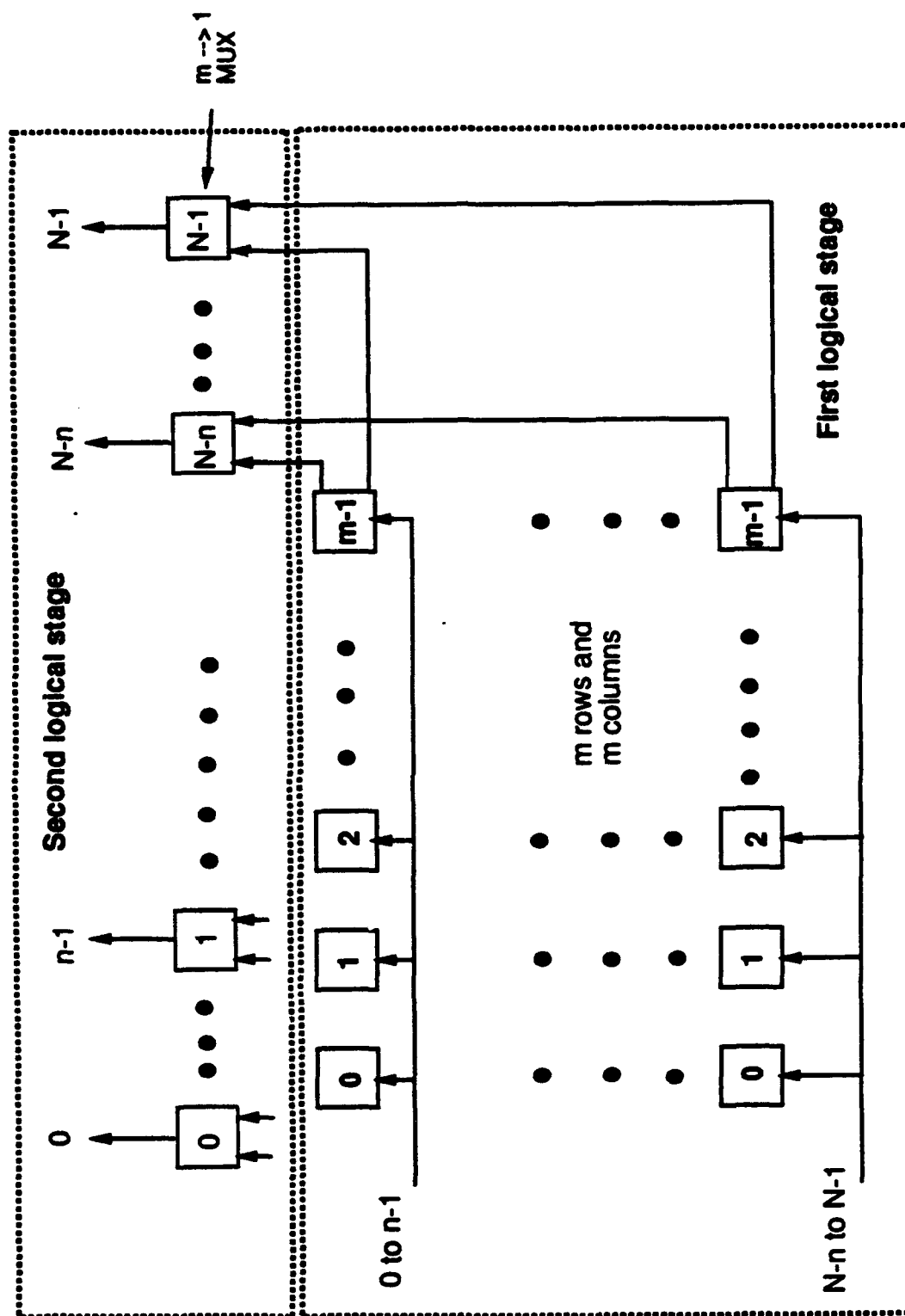


Figure 4.5. Building larger crossbar networks

$$2 * m^2 + m^2/n + m^2/n^2 = \left(2 + \frac{1}{n} + \frac{1}{n^2}\right) (m^2)$$

- In general for $n^s < m \leq n^{s+1}$, the total number of PARCOS chips required is

$$m^2 \left(2 + \frac{1}{n} + \frac{1}{n^2} + \dots + \frac{1}{n^s}\right) = 2 \times m^2 + m^2 \left(\frac{n^s - 1}{n^s(n - 1)}\right)$$

It should be noted that in the first logical stage, groups of n inputs are fed in parallel to m PARCOS chips. For very large m , the loading on the input signals may be excessive. This problem can be addressed by suitably buffering the input signals.

For various values of n and m , the number of PARCOS chips and the board area required to implement larger crossbars are compared with other networks in a following section.

4.3.2 Network setup and re-switching

The method for programming a connection pattern in these networks is similar to the 64×64 network described earlier.

A connection pattern can be broken into two parts: The first logical stage, and the second logical stage. The method for deriving address and data for the first logical stage is the same for any size network, whereas a slightly different scheme is employed for the second logical stage, depending on the network size.

The scheme for deriving address and data for the first logical stage is as follows. Refer to the block diagram in figure 4.5.

- For brevity, let us assume that N , n , and m are all powers of 2, such that

$$N = 2^x \implies \log(N) = x$$

$$n = 2^y \implies \log(n) = y$$

$$m = 2^z \implies \log(m) = z$$

$$x = y + z$$

- The z most significant bits of the input address select one of the m rows.
- The y least significant bits of the input address are used as data in the selected row.
- The y least significant bits of the output address are used as the address in the selected row.

The scheme for deriving the address and data for the second logical stage is as follows.

For $1 < m \leq n$

The second logical stage of the crossbar comprises one physical stage. Each PARCOS chip can be used as n/m m -input 1-output multiplexers. There are m^2 PARCOS chips in this stage.

- The $2z$ most significant bits of the output address are used by hardware to select one of the m^2 PARCOS chips.
- The remaining $x - 2z$ least significant bits of the output address are used to select the proper output (address) in the selected PARCOS chip in the second logical stage.
- In any of the m^2 chips in this stage, an output X is connected to one of the following m inputs:
 $m * X, m * X + 1, m * X + 2, \dots, m * X + m - 1$
 This is determined by the z most significant bits of the input address. This selection can be done by simple hardware.

For $n < m \leq n^2$

The second logical stage of the crossbar comprises two physical stages. The first physical stage (closer to the inputs) implements $N \times m$ n -input 1-output multiplexers. For every output, m PARCOS chips are required in this stage. The second physical stage (closer to the outputs) implements N m/n -input 1-output multiplexers. Each PARCOS chip can implement $N/m/n = n^2/m$ m/n -input 1-output multiplexers.

- The $2z - y$ most significant bits of the output address are used by hardware to select one of the m^2/n chips in the second physical stage.
- The next $2y - z$ (n^2/m m/n -input MUX per chip) most significant bits of the output address are used to select the proper output (i.e. address) in the selected PARCOS chip in the second physical stage.
- In any of the m^2/n chips in the second physical stage, an output X is connected to one of the m/n inputs
 $m/n * X, m/n * X + 1, m/n * X + 2, \dots, m/n * X + m/n - 1$

which is determined by the $z - y$ most significant bits of the input address. This can be done by simple hardware.

- The first physical stage is organized as N groups of m PARCOS chips each. The z bits of the output address are used to select one of these groups. Notice that each PARCOS chip in this stage is used as an n -input 1-output MUX. Therefore output is taken from port #0 in all of these chips (i.e. address 0 is chosen for writing in the chips).

- If the input address is

$$I = i_{2n-1}i_{2n-2}\dots i_{2n-1}i_{n+1}i_n i_{n-1}\dots i_1 i_0$$

then $J = i_{2n-1}i_{2n-2}\dots i_n$ is used as data in the selected group (i.e. input # J is connected to output #0).

This scheme can be extended to even bigger crossbars. However, it should be noted that such designs are impractical. For example, if $n = 32$ and $m = 32 \times 32 = 1024$, a 32K-input 32K-output crossbar will require 2,129,920 32 $\times$ 32 PARCOS I chips. PARCOS I chip has an area of $1.2 \times 1.2 = 1.44$ sq. inch. Thus the total board area required will be $(2,129,920 \times 1.44)/(36 \times 36) = 2,366.6$ sq yards. About 14,200 12" $\times$ 18" (rough size of a VME board) PCBs will be required to build this network.

4.4 Larger non-blocking networks

A crossbar is an ideal network for interprocessor communication in multiple-processor systems. But, as pointed out in the preceding section, there is a practical limit to the size of crossbar network that can be built from smaller crossbars. In order to build larger connection networks that can efficiently support the N^N mappings property, it is necessary to explore designs that use less hardware than a crossbar.

The most often cited disadvantage of a crossbar network is that it requires $\theta(N^2)$ switches. From our experience in constructing building block chips for larger communication networks, we contend that the greatest impediment to building larger networks is the number of chips, boards, and pinout requirement on the chips and the boards rather than the number of transistors required in building them. For example, the 32 $\times$ 32 size of PARCOS I was limited by the maximum number of pins on the available package rather than the number of transistors in it. This issue will be discussed more in a following section. For now, it is sufficient to state that minimizing the number of transistors is not the criteria chosen for building larger networks in this thesis. In addition to the N^N mappings property, we are

interested in the following requirements for the larger network: (1) Minimize the number of switching stages between an input and an output, (2) The design should be modular so that it can be built from a minimum of different chip types, and (3) It must be possible to quickly establish an input-output connection pattern. These three criteria further help us in choosing a network in this and the next section.

In the non-blocking network category, there are only two fundamental explicitly known constructions as proposed by Clos [Clos 53] and by Cantor [Cantor 71]. Cantor's work was guided by minimizing the number of switches in the network. Further extension was carried out by Pippenger [Pippenger 78]. An optimal non-explicit (i.e. merely indicating a proof of its existence but unable to provide a method of constructing it) construction of a non-blocking network was provided by Bassalygo and Pinsker [Bassalygo 73]. An overview of these issues can be found in [Thurber 78, Thurber 79]. When we apply our criteria of minimizing the number of switching stages between an input and an output in addition to modular design, the Clos network is the design of choice. For an equal number of inputs and outputs, a Clos network can be built from multiple copies of a single building block chip. In fact, if we use the same building block chip (i.e. a small crossbar) to construct a Cantor network, the resulting topology will be *identical* to the Clos network. Before we describe the network, an additional note is in order. Within the framework of this chapter, *all* interprocessor communication patterns in the target parallel processor are fixed, known a priori, and employ central routing control. Thus, we don't really need a non-blocking network. A rearrangeably non-blocking (or permutation) network will suffice. However in the chapter dealing with data dependent asynchronous routing, a strictly non-blocking network is required for ideal performance. Thus for the sake of completeness, we include non-blocking networks in this chapter.

4.4.1 Network architecture

A 3-stage N -input N -output Clos network is shown in figure 4.6. We denote the network by $N(n, m, r)$. The first stage in the network comprises r switches that implement a $n \times m$ crossbar function. The second stage comprises m switches that implement an $r \times r$ crossbar function. Finally the third stage comprises r switches implementing an $m \times n$ crossbar function. Clos [Clos 53] showed that for $m \geq 2n - 1$ the network is non-blocking in the strict sense. If we let $m = 2n$, then by using modular $m \times m$ switches, one can build a 3-stage Clos network with up to $(m/2 \times m) = m^2/2$ inputs and outputs. For example, by using 96 copies of a PARCOS I chip with $m = 32$, one can build a 512-input 512-output network as shown in figure 4.7. It should be noted that in the input stage, 16 of the 32 inputs to PARCOS I are

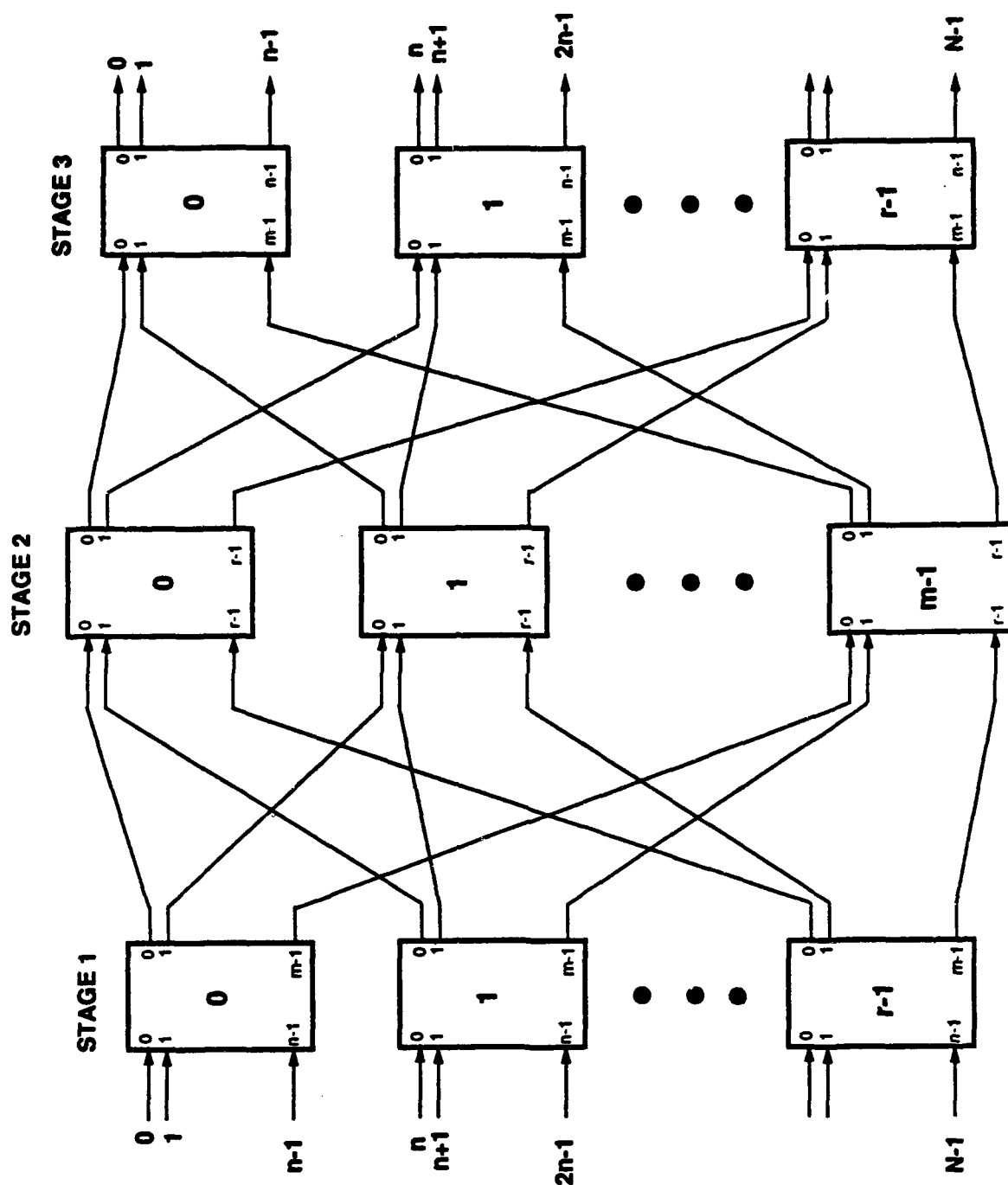


Figure 4.6. A 3-Stage Clos network

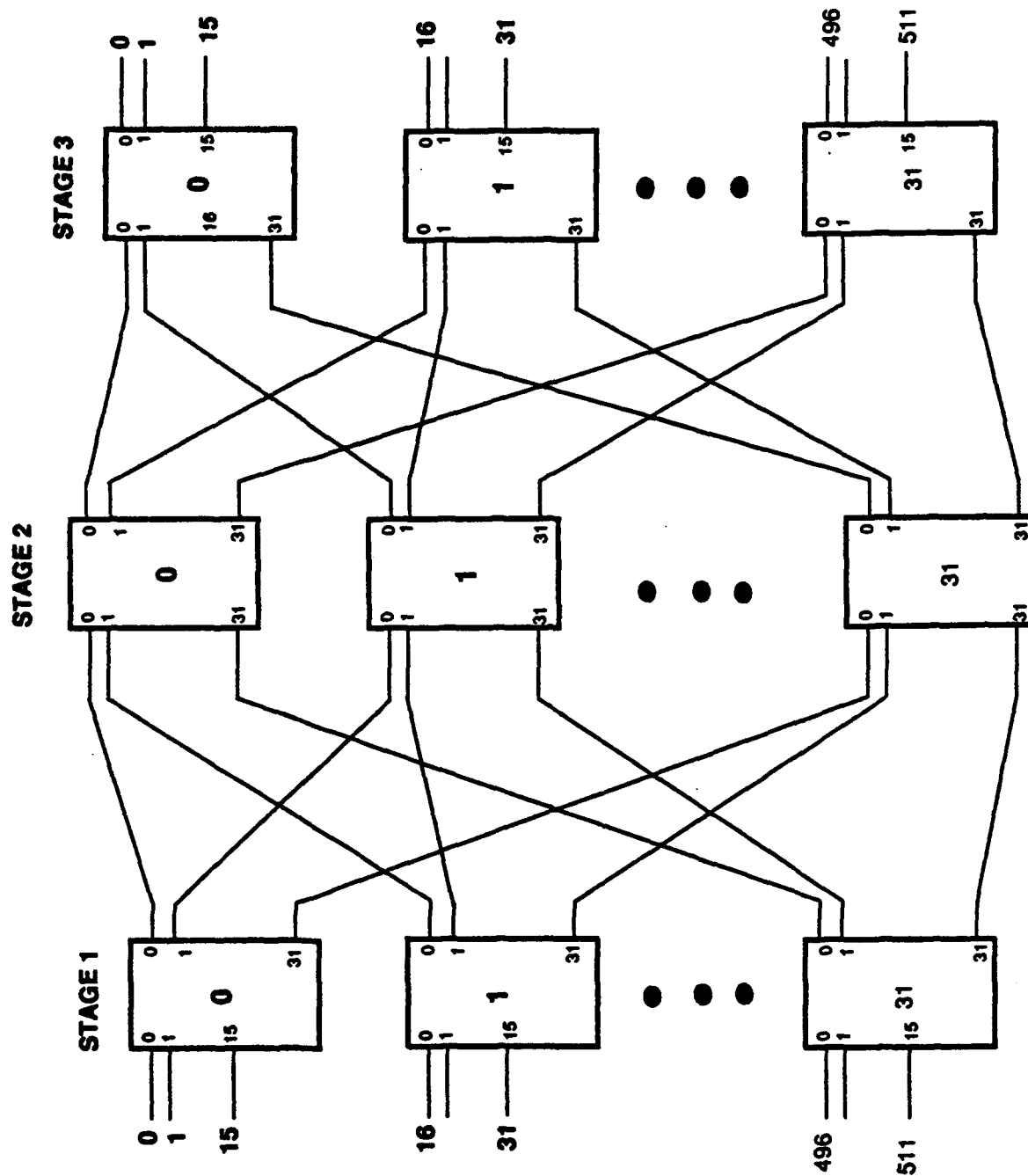


Figure 4.7. A 3-Stage 512-input 512-output Clos network

left unconnected. Similarly 16 of the 32 outputs from PARCOS I in the output stage are left unconnected. We have developed a modified 4-stage design that extends the size of the Clos network by a factor of 2. By connecting 256 PARCOS I chips in this topology, a 1024-input 1024-output network can be constructed as shown in figure 4.8. Larger networks can be constructed with a 5-stage design. For example, by connecting 4096 PARCOS I chips in a 5-stage design, an 8K-input 8K-output network can be constructed as shown in figure 4.9.

4.4.2 Network setup and re-switching

Programming a connection pattern in these networks can be considered as comprising two parts. In the first part, address and data values for various stages are derived from the list of input-output address pairs. In many off-line routing schemes, this is also referred to as generating routing tags from an input-output mapping. In the second part, these derived data values are actually written into the derived address locations. Within the framework of this chapter where all connection patterns are known apriori, we are not interested in the time and storage costs of the first part of programming a connection pattern, which can be carried out off-line in the ACU.

For an N-input N-output non-blocking network with $m = 2n = r$, the second part of programming a connection pattern can be carried out in N steps. For the 3-stage Clos network in figure 4.6 the PARCOS I chips in each stage can be partitioned in the address range:

$$\{0, 1, \dots, n-1\}, \{n, n+1, \dots, 2n-1\}, \dots, \{(m-1)n, (m-1)n+1, \dots, (m-1)n+n-1\}$$

Corresponding to each input-output address pair, three data values are written simultaneously at three addresses in the input, middle, and the output stage respectively (notice that $m = 2n$, but only half of these locations have values stored in them). The same scheme is employed for the 4-stage modified Clos network, where groups of 4 PARCOS chips in stage 3 and stage 4 are addressed in the same way as a larger crossbar by using schemes outlined in a previous subsection. Similarly, groups of 2 PARCOS chips in stage 1 are addressed as a single large crossbar. For a 5-stage Clos network, 5 data values are written simultaneously in the 5 stages for each input-output pair. In the next chapter we show how to derive these address data pairs.

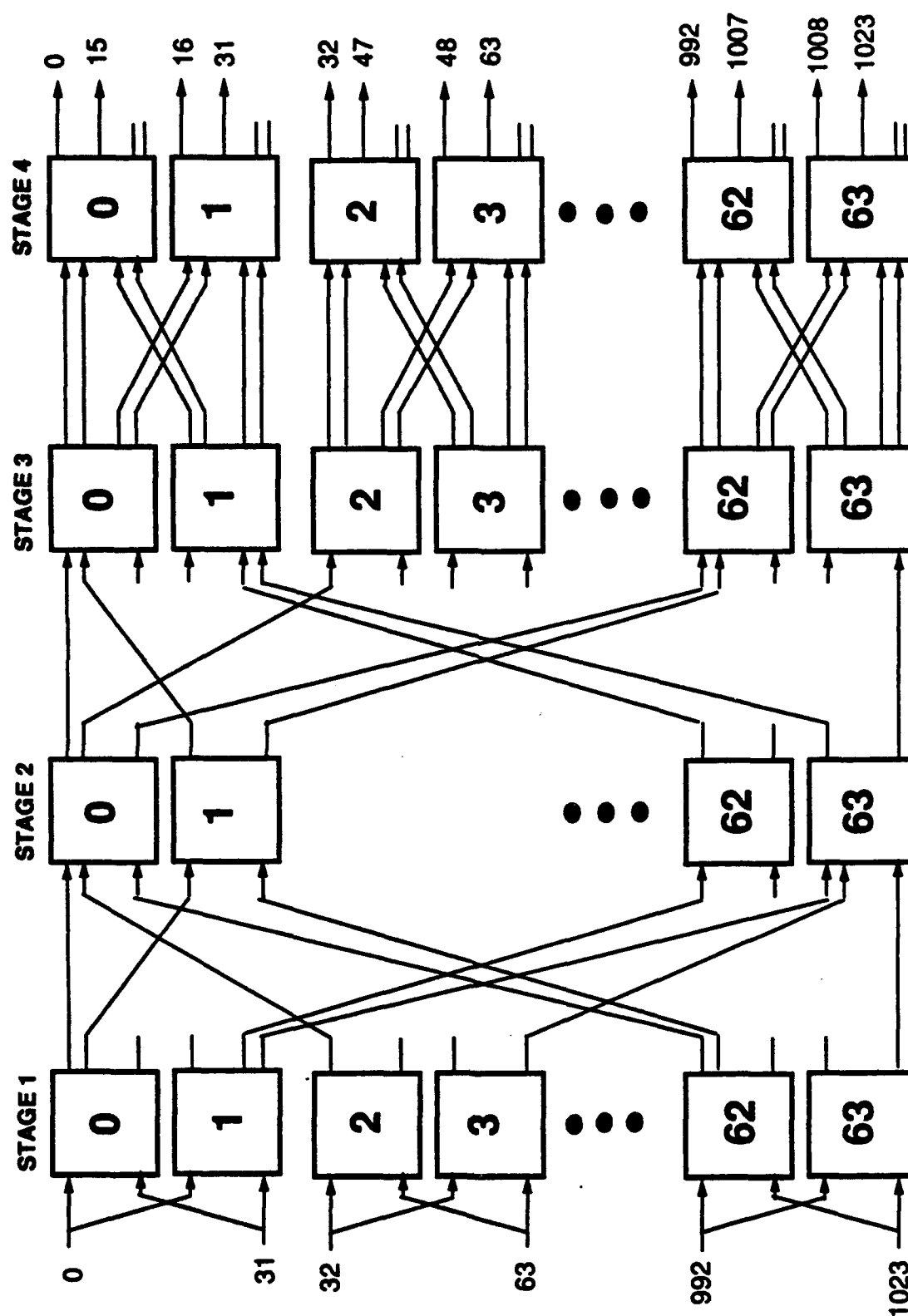


Figure 4.8. A 4-Stage modified Clos network

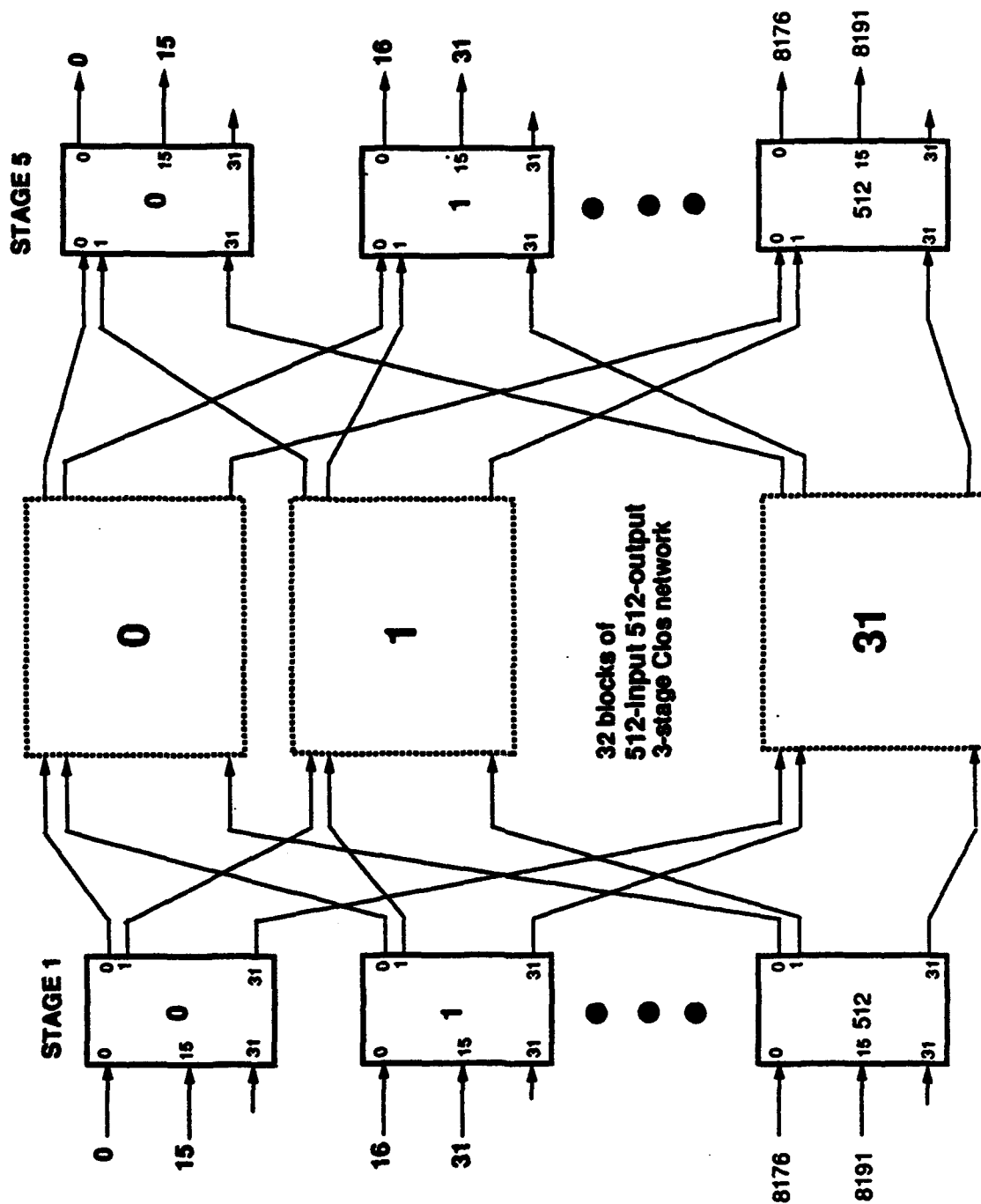


Figure 4.9. A 5-Stage Clos network

4.5 Larger rearrangeably non-blocking networks

Two fundamental approaches have been followed in the design of rearrangeably non-blocking networks. The first approach is a "rectangular switch" based design, and the other approach is based on 2×2 connectors. The "rectangular switch" approach is based on Benes networks. Benes networks in turn are derivatives of Clos networks. The 2×2 connector-based approach is driven by the need to minimize the total number of switches (transistors) in the network at the expense of higher latency.

We shall not consider 2×2 connector-based designs in this thesis. First, as discussed earlier, our experience in constructing building block chips for communication networks shows that the number of switches (transistors) is not the constraining factor. Second, the three criteria stated at the beginning of section 4.4 for selecting a network architecture further rule out 2×2 connector based approaches.

The original work on "rectangular switch" networks was carried out by Clos. These networks were discussed previously in section 4.4. Benes extended these ideas in the context of mathematics of switching systems. The design we consider in this section is the rearrangeably non-blocking network proposed by Benes [Benes 65]. It should be noted that in the context of rearrangeably non-blocking networks, the work by Masson together with Jordan [Masson 72,76] is a generalization of Benes' work. An overview of these issues can be found in [Thurber 79].

4.5.1 Network architecture

A 3-stage N -input N -output Benes network is shown in figure 4.10. Notice the similarity between the 3-stage Benes network and the 3-stage Clos network. In the notation of the 3-stage Clos network, for $m \geq n$, the network is rearrangeable. The network in figure 4.10 is the optimal case of a 3-stage rearrangeable network with $m = n = r$. For example, by connecting 96 copies of the PARCOS I chip with $m = 32$, one can build a 1024-input 1024-output network. In a manner similar to Clos networks, even larger networks can be constructed with a 5-stage design. For example, by connecting 5120 PARCOS I chips in a 5-stage design, a 32K-input 32K-output Benes network can be constructed.

Before we end this subsection, a brief note on 2×2 connector-based approaches is in order. Figure 4.11 shows the original extension to the Benes network proposed by Waksman

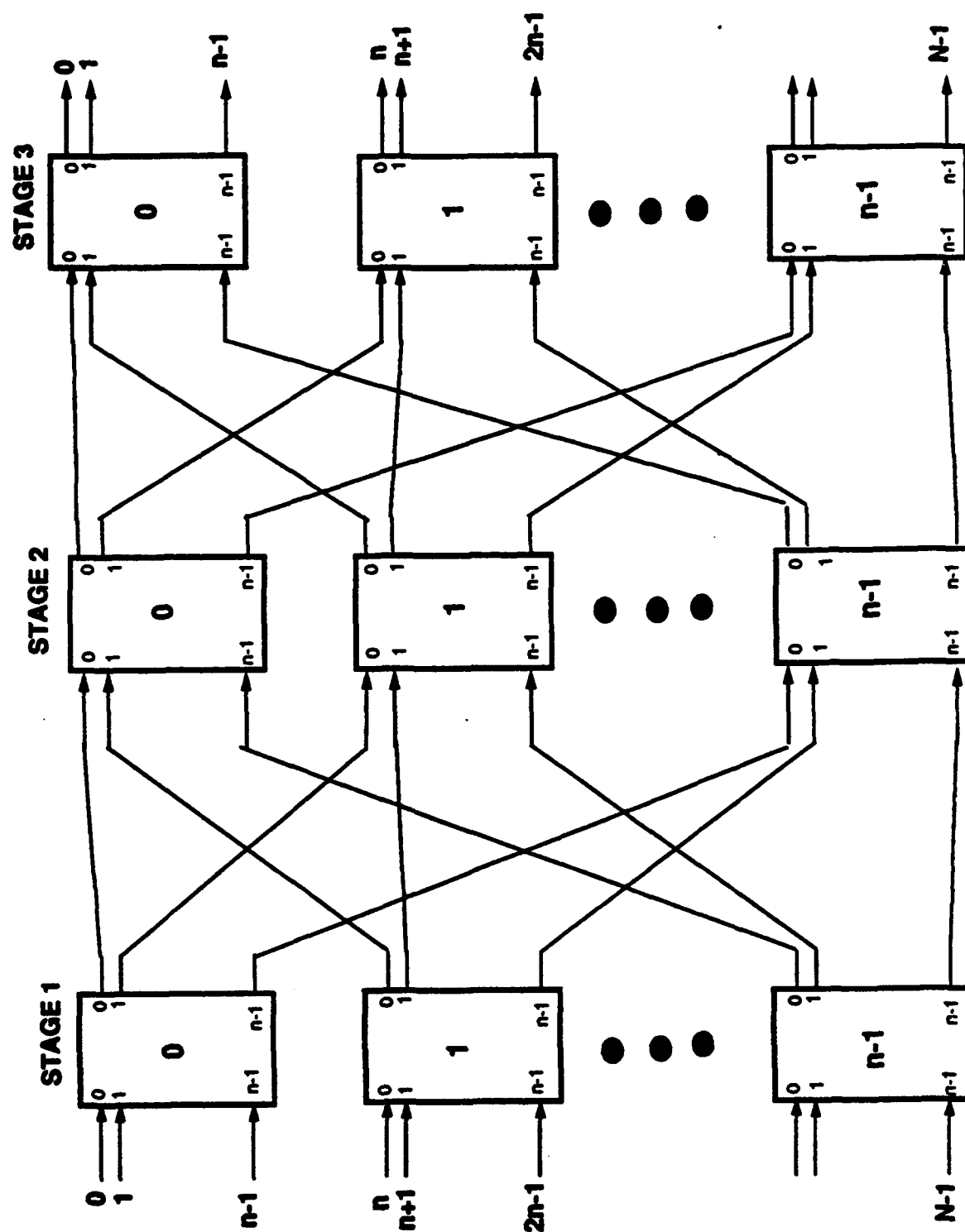


Figure 4.10. A 3-Stage Benes network

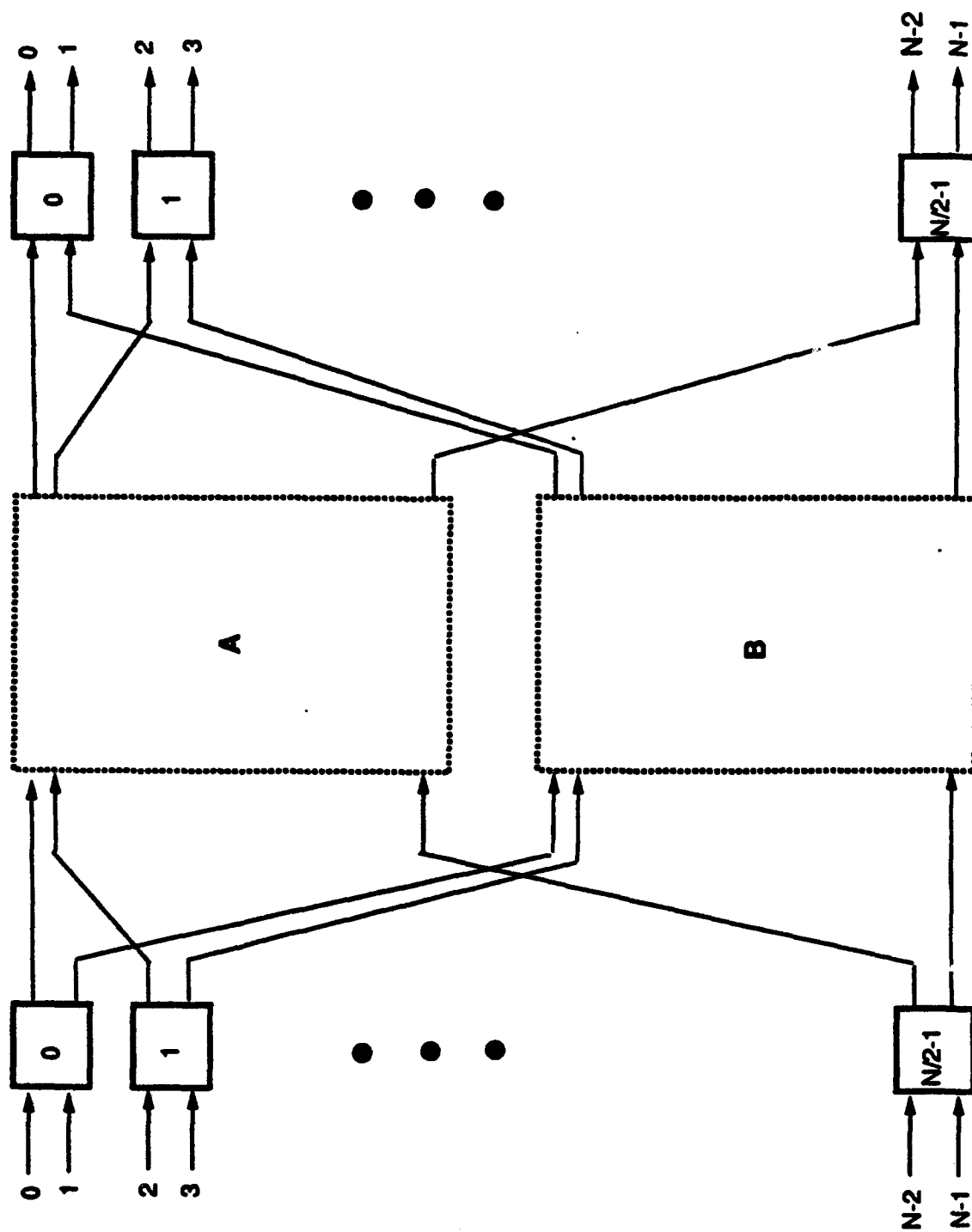


Figure 4.11. A 2×2 connector based reconfigurable network

[Waksman 68]. In Clos network terminology, Waksman has $m = 2$, $n = 2$, and $r = N/2$. The two larger blocks in the middle stage are recursively decomposed into 2×2 connectors. Another approach for deriving 2×2 connector-based networks is to start with the Benes network with $m = n = r$ and recursively decompose each $n \times n$ switch into 2×2 connectors. Notice that these two schemes lead to complementary wiring schemes between adjacent stages of 2×2 connectors. Both approaches will result in $2\log(N) - 1$ stages of 2×2 switches. If any of these networks is broken into two along the middle stage, then either half of these networks will have a "butterfly" or its complementary connection topology.

4.5.2 Network setup and re-switching

Programming a connection pattern in Benes networks is similar to programming Clos networks. In fact, both classes of networks can use the same scheme for the first part of programming a connection pattern. It is only in asynchronous routing, where the non-blocking capability of a Clos network is required, that their programming differs. The first part of programming a connection pattern affects network setup time in data dependent routing where processors may request to communicate with other specific processors, and will be discussed in the next chapter.

For an N -input N -output Benes network, the second part of programming a connection pattern can be carried out in N steps as in a Clos network.

4.6 Analysis of PARCOS I

In this section we discuss the hardware cost, power dissipation, latency and throughput of PARCOS I in detail. We analyze the hardware cost (die area) of implementing an arbitrary size PARCOS I crossbar on a single die and show that the maximum size PARCOS I that can be implemented on a single die is pin-limited rather than area-limited. The design's power dissipation is analyzed to support this argument. Its latency and throughput are also analyzed to project what can be expected from bigger switches using better CMOS technologies.

4.6.1 Hardware Cost

In this section we analyze the hardware cost (die area) of the PARCOS I design, and show that the number of pins in the VLSI package is a greater impediment to constructing a larger PARCOS I on a single chip than the number of transistors required. To support this argument we analyze the area requirements and power dissipation of PARCOS I to project the largest PARCOS I crossbar that can be constructed on a single die using one of the currently available technologies.

Two approaches can be used in making our projection. The first approach is to purely derive the die area analytically. We make assumptions regarding the design rules, feature size, and area requirements of various components to arrive at the final result. We shall not pursue this approach for two reasons. First, there is a tendency to make inaccurate estimates of area for various components (e.g. placement and routing constraints in a specific case may make the chip area significantly greater than using "standard" measures of the areas for its leaf cells and interconnect). Second, area estimates depend directly on the technology and process used for implementing the chip. The opposite approach is to empirically compute the VLSI area by designing (and perhaps fabricating) various PARCOS I chips. This approach provides the most accurate estimate of area but is impractical because of the cost involved.

We follow an intermediate approach of extrapolating the area from the actual implementation of the 32×32 PARCOS I. First we estimate the die area of larger PARCOS I chips as if they were built in the same manner as the 32×32 PARCOS I. Next we point out improvements that can be carried out to reduce its die area.

A block diagram of the 32×32 PARCOS I chip is shown in figure 4.1, and its checkplot is shown in figure 4.3. For the time being let us ignore the die area required for global wiring from the pads. The 32×32 PARCOS I layout area is determined as follows: The width of the layout is determined by the width of the communication matrix. In the layout, the CPC width was intentionally increased to match the communication matrix width. The communication matrix width is comprised of 32 1 of 32 multiplexers shown in figure 4.2, and an input buffer. In the actual design, using a 2 micron technology ($\lambda = 1\mu$), this width is about $1000 + 32 \times 150 = 5800\lambda$ or roughly 6000 microns. A checkplot of one half of one of the multiplexers is shown in figure 4.12. Because of the binary tree structured layout, the width of a multiplexer will increase proportional to $\log_2$ of the number of inputs. Therefore, for a 128×128 PARCOS I, the multiplexer portion of the communication matrix will have a width of

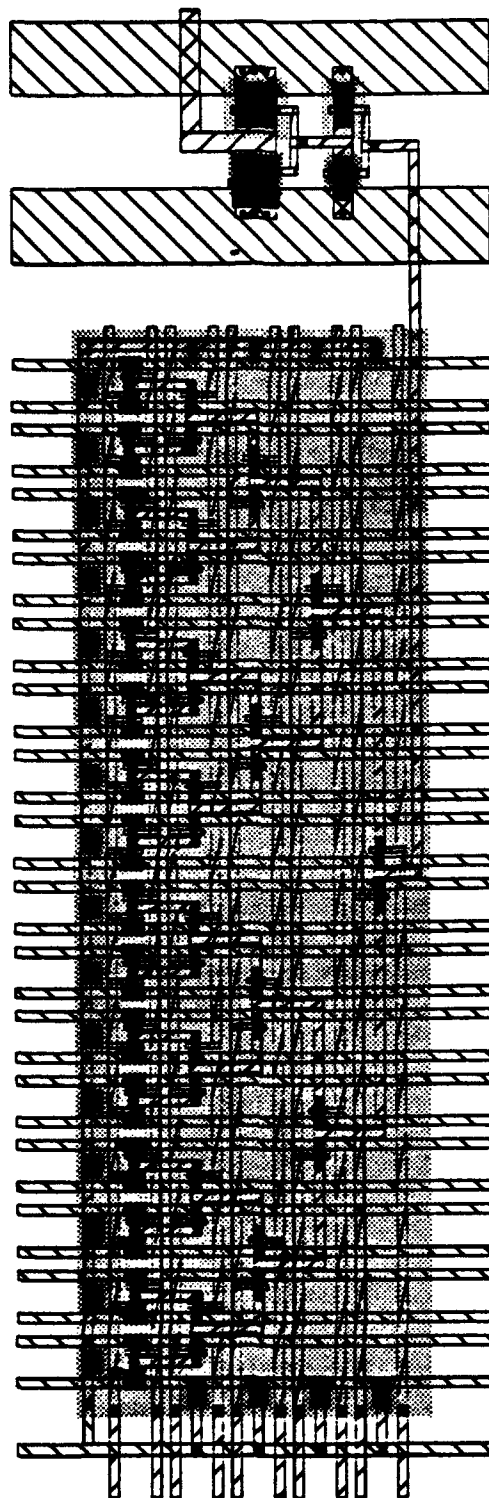


Figure 4.12. One half of the MUX tree

$$128 \times \left\{ \frac{150}{\log_2 32} \times \log_2 128 \right\} = 26,880\lambda \quad (4.1)$$

The input buffer is comprised of 32 chained inverters to drive the multiplexer inputs, and a jumper to split the 32 inputs to the multiplexers into two groups to feed the upper and the lower selector trees. The jumper in the 32×32 PARCOS I chip is 500λ wide. Jumper width increases proportional to the number of inputs with a constant of proportionality less than one. Therefore, the jumper width in 128×128 PARCOS I will be less than $500 \times 4 = 2000\lambda$. The chained inverter driver's width in the current design is 274λ . Each inverter chain is comprised of 4 inverters of monotonically increasing size to drive the large load in the communication matrix. This is done to minimize the delay in the driver. In the ideal case, to drive a load C_l , $n = \log_e(C_l/C_o)$ inverter stages are required, where C_o is the capacitance of a minimum size inverter. Each successive inverter is $e (=2.718)$ times larger than the previous one. In the practical case, n has to be an integer, and in many cases an even number. Also, because of the drain island capacitance effect, n is smaller than $\log_e(C_l/C_o)$. Detailed discussion of this issue can be found in [Lee 84; Lewis 84; Mead 82; Moshen 79; and Shoji 88]. A 128×128 PARCOS I will represent about four times the load of a 32×32 PARCOS I to the input buffer. If $n_1 = \log_e(C_l/C_o)$ inverters are required in the input buffer of a 32×32 PARCOS I, $n_2 = \log_e(4 \times C_l/C_o)$ inverters will be required for a 128×128 PARCOS I chip. This gives $n_2 = n_1 + 1.39$. For a moment let us assume that it is possible to have 5.39 inverters. The width of the driver for a 128×128 PARCOS I chip is calculated as follows.

Let $w_1, w_2, \dots$ represent the widths of 1-stage, 2-stage, ... inverter chains for the driver. Then

$$w_1 = k$$

$$w_2 = k + ke = k(1 + e)$$

...

...

...

$$w_n = k(1 + e + e^2 + \dots + e^{n-1})$$

where k is the proportionality constant and depends on the CMOS technology used.

This series is solved to give

$$w_n = \frac{k(e^n - 1)}{(e - 1)}$$

We know that $w_4 = 274$. Then

$$\frac{w_{5.39}}{w_4} = \frac{(e^{5.39} - 1)}{(e^4 - 1)} \quad (4.2)$$

or

$$w_{5.39} = \frac{274(e^{5.39} - 1)}{(e^4 - 1)} = 1115.5\lambda \quad (4.3)$$

Therefore the total layout width of 128×128 PARCOS I will be $26880 + 2000 + 1115.5 = 29995.5\lambda$. Currently available technologies with $\lambda = 0.4\mu$ will result in a layout width of $29995.5 \times 0.4 = 11998.2\mu$ or about 12 mm. Currently available die widths of the order of 15-18 mm allow plenty of room for the pads and the global wiring. Two additional points should be noted here. First, $\lambda = 0.25\mu$ technologies are on the verge of commercial viability, and will further allow us to reduce the width of the PARCOS I layout, allowing up to a 256×256 PARCOS I design to be implemented on a single die. Second, the current 32×32 PARCOS I is designed *very* conservatively. A more aggressive layout would allow us to reduce the multiplexer width by at least 30% resulting in further reduction of the layout width. Next we address the height of the PARCOS I layout.

The height is the sum of the heights of the communication matrix, control pattern register, connection pattern cache, and column decoder and drivers. Respectively, this amounts to

$$1149 + 464 + 2905 + 1322 = 5840\lambda$$

The height of the communication matrix is directly proportional to the number of inputs. Therefore, in a 128×128 PARCOS I, the communication matrix will be

$$1149 \times 4 = 4596\lambda \quad (4.4)$$

high. The control pattern register comprises 32 bytes of 5 read bits. A read bit is comprised of bit line precharge circuitry, followed by a sense amplifier, followed by a buffer to drive the control lines of the multiplexers in the communication matrix. The only aspect that changes with a larger communication matrix is the buffer in the read bit, which is about 150λ high and whose growth is similar to the driver circuit in the communication matrix. Therefore, for a 128×128 PARCOS I, the height of the control pattern register will be approximately

$$464 + 150(e^{1.39} - 1) = 916.2\lambda \quad (4.5)$$

The height of the connection pattern cache is independent of crossbar size. For now let us assume that there are just 32 control words. The column decoder and driver comprises

an address and data line buffer, followed by bit line drivers. The height of this portion is completely determined by the number of control words in the connection pattern cache. The total layout height of a 128×128 PARCOS I design with 32 control words will thus be

$$4596 + 916.2 + 2905 + 1322 = 9739.2\lambda \quad (4.6)$$

At $\lambda = 0.4\mu$, the layout height will be $9739.2 \times 0.4 = 3895.7\mu$, or about 3.9 mm. Therefore, layout height is not a restriction in designing bigger PARCOS I chips.¹ If we allow a maximum layout height of about 10mm, then there is room for more than 200 control words in the CPC. It should be noted, however, that the static RAM design used in the 32×32 PARCOS I is unoptimized. For example, the CPC contains $32 \times 32 \times 5 = 5120$ bits of RAM and occupies an area of $3 \times 5\text{mm}^2$. By using a 0.8μ ($\lambda \approx 0.5\mu$) double polysilicon CMOS technology in 1988, it was possible to make a 1Mb SRAM with a 15nS access time and a die area of $6.15 \times 15.21\text{mm}^2$ [Sasaki 88]. Taking into account the scaling effect of λ , it is possible to improve our design by a factor of at least 8. Therefore, including 1K control words in the CPC is feasible using current technology.

Before we end this subsection, a brief note on the silicon area requirements of arbitrary size PARCOS I is in order. If we assume a fixed number of words in the CPC, its height is independent of the communication matrix size. The width of PARCOS I layout is determined by the width of the communication matrix. The width of the communication matrix is dominated by the width of the multiplexer; the input buffer makes a minor contribution to the communication matrix width. From equation 4.1, it can be seen that for an $n \times n$ communication matrix, its width is proportional to $n \log_2 n$ ($150/\log_2 32$ is the proportionality constant). From equation 4.4, it can be seen that PARCOS I height is proportional to n . Therefore, PARCOS I layout area is proportional to $n^2 \times \log_2 n$. It should be noted that this is the die area. As will be seen in the next section, PARCOS I design is pin limited rather than silicon area limited, and the package and board areas are much bigger than the silicon die area. Therefore, the seemingly large growth rate of the die area is of no consequence in the hardware (board area) cost. Next, we discuss the pinout requirements of PARCOS I.

¹Notice that for a 256×256 PARCOS I chip, the layout height will be

$$\begin{aligned} 1149 \times 8 + (464 + 150(e^{2.08} - 1)) + 2905 + 1322 \\ = 9192 + 1514.7 + 2905 + 1322 = 14934\lambda \end{aligned}$$

At $\lambda = 0.4\mu$, the layout height will be $14934 \times 0.4 = 5973.6\mu$, or about 6 mm

4.6.2 Pinout requirements

While providing pads for signals in a die, certain design rules have to be followed. For example, the pads are required to have a minimum size to guarantee a reliable bond to the wires, e.g. in Pin Grid Arrays (PGA), or the carrier, e.g. Tape Automated Bonding (TAB). Additionally, the pads should have certain minimum spacing. The minimum pitch in wire bonding varies from 0.16 mm to 0.30 mm in current state-of-the-art technologies. TAB can offer lead spacings as small as 0.08 mm to 0.12 mm. There is a limit, however, to which the lead spacing can be reduced (and consequently provide more signals to the die). As the feature size reduces and the operating frequency increases, the power dissipation in the die increases. Higher operating frequency (in turn higher *slew rate* of the signals), closely spaced thinner conductors, and a larger number of signals on a die play a havoc with chip reliability. Signal coupling (due to radiation and capacitive coupling), signal deterioration (due to higher lead inductance), and dI/dt noise generated in the power lines when a large number of output pads are driven simultaneously, limit the maximum number of pins that can be allowed on a die. The current state of the art allows up to 400 pins on a die. Not all of these pins are used for signals. For example, in a 180 pin grid array package of TMS320C25, 24 pins are used for distributing power. In a 325 pin grid array package of TMS320C30, 69 pins are used for distributing power. These carriers use a single die. In another example of a high density interconnect (but using multiple dies) such as Thermal Conduction Module (TCM) [Blodgett 82], 500 out of the total 1800 pins are used for distributing power. The increase in the percentage of power supply pins as the total number of pins on a carrier increases independent of whether it is a single chip or a multichip carrier. We showed in the previous subsection that it is reasonable to implement a 256×256 PARCOS I switch on a single die (from the standpoint of silicon area). Assuming a signal to power supply pin ratio of 3, such a die will require of the order of 700 pins. Assuming a signal to power supply pin ratio of 3.5 for a 128×128 PARCOS I, such a die will require of the order of 350 pins. A discussion of issues related to pinout requirements can be found in [Bakoglu 90; Davidson 82].

Therefore, we argue that the maximum size PARCOS I becomes pin limited before becoming hardware limited. Specifically, current technology is only capable of supporting a 128×128 PARCOS I crossbar. Next we discuss the power dissipation in PARCOS I.

4.6.3 Power dissipation

Estimation of power consumption of a CMOS VLSI chip is a difficult task, especially at an early stage of the chip's design. To quote Shoji [Shoji 88, pp 294], "The power of a CMOS VLSI chip is the hardest parameter to estimate. The author's experience is that the accuracy of chip power estimate based entirely on the logic design specification is only within 40% accuracy. There is no way to estimate accurately how often the internal circuits charge or discharge. It is easier and is more accurate to estimate the power consumption of a chip from a similar chip designed earlier, by applying a modification factor,".

We will follow a similar approach in estimating power dissipation in PARCOS I. The actual observed power dissipation in the 32×32 PARCOS I will be used to project dissipation in larger switches.

The 32×32 PARCOS I is comprised of two major blocks. The CPC and a 32×32 communication matrix. The CPC is essentially a static RAM and is active only during reading or writing into control words to modify a communication pattern. The total CPC capacity is $32 \times 32 \times 5 = 5K$ bits. The actual power dissipation in the 32×32 PARCOS I was measured under two conditions: Reads and writes to the CPC with no activity in the communication matrix, and communication through the matrix without any reads or writes into the CPC. In the second case it was found that the maximum dissipation occurred in cases where the input-output mapping in the communication matrix was a permutation. The measurements were taken on a Tektronix DAS 9200 system. The probes for observing the output signals from PARCOS I essentially provided about 10pF capacitive load. The supply current to the chip was directly measured using a multimeter that provided the average DC current. We project the two currents (i.e. only CPC active and only communication matrix active) for larger size PARCOS I chips separately and then project their dissipation together.

It can be shown that when driving a capacitive load, the power dissipation is proportional to the capacitance of the load. The power dissipated in the CPC is essentially in precharging and transitions related to reads and writes in the word and bit lines. Assuming the same number of control words in the larger PARCOS I, the power dissipation in PARCOS I will increase proportional to $n \log(n)$, where n is the number of inputs. The standby current in the 32×32 PARCOS I chip is 3.5mA and the current during reads and writes to the CPC is 18.8 mA. Therefore the current in a 128×128 PARCOS I during CPC reads and writes will be about

$$18.8 \times \frac{128 \log_2(128)}{32 \log_2(32)} = 105.28 \text{ mA} \quad (4.7)$$

It should be noted that the SRAM used in the design was about 10 times larger than state-of-the-art cells. Using a better SRAM design will reduce the power dissipation.

The PARCOS I current during permutation connections in the communication matrix is 18.1mA. A substantial part of this current is dissipated in the pad output drivers. The power dissipation in driving the DAS 9200 probes is about

$$P_D = \left[\frac{1}{2} \times N \times C_l \times V_{dd}^2 \times f \right] \text{ Watts} \quad (4.8)$$

here $N = 32$

$C_l = 10 \times 10^{-12} F$

$f = 20 \times 10^6 \text{ Hz}$, and

$V_{dd} = 5V$

This gives

$$P_D = \frac{1}{2} \times 32 \times 10 \times 10^{-12} \times 5^2 \times 20 \times 10^6 = 0.08 \text{ Watts} \quad (4.9)$$

which is $0.08/5 = 16 \text{ mA}^2$. In other words, a large part of the power is spent in driving the output pads. This allows us to use a linear relationship between chip power dissipation and the number of outputs in estimating power dissipation in larger PARCOS I chips. Therefore in a 128×128 PARCOS I the current due to communication activity will be about

$$18.1 \times \frac{128}{32} = 72.4 \text{ mA} \quad (4.10)$$

The combined current assuming both activities (CPC writes and communication) are taking place simultaneously in a $n \times n$ PARCOS I is given by

$$C_{sb} \times \frac{n}{32} + (C_{cpc} - C_{sb}) \times \left(\frac{n \times \log_2(n)}{32 \times \log_2(32)} \right) + (C_{mus} - C_{sb}) \times \frac{n}{32} \quad (4.11)$$

Where C_{sb} is the standby current, C_{cpc} is the current during CPC activity, and C_{mus} is the current during communication activity in 32×32 PARCOS I chip. Also, it is assumed that

²It should be noted that output signals didn't change state in every cycle because of the DNRZ format used. If the ratio of the state change to clock cycles is also taken into account (but difficult to calculate in general) this number will be smaller

the standby current scales proportionally to the size of larger PARCOS I chip. For $n = 128$ the current will be

$$3.5 \times \frac{128}{32} + (18.8 - 3.5) \times \frac{128 \times \log_2(128)}{32 \times \log_2(32)} + (18.1 - 3.5) \times \frac{128}{32} = 160.9mA \quad (4.12)$$

This gives a power dissipation of $160.9 \times 5 = 804.5mW$ in the worst case.

It should be noted that in practical situations, the power dissipation due to output drivers will be substantially less than the worst case, and also the current in the CPC will be less because a read or write will not take place in every cycle. A power dissipation of $804.5mW$ allows plenty of margin for scaling down of the feature size and increasing the number of control words in the CPC. Next we discuss the time required to set up the communication matrix and reconfigure it.

4.6.4 Time to set up and reconfigure the crossbar

The time to set up or reconfigure the switch matrix is completely determined by the speed of the CPC and was discussed in detail earlier in this chapter. However, a brief note is in order. The latest state-of-the-art CMOS SRAM chips with capacities as great as 1 Mbit allow read or write times as short as 10-15nS. Assuming that this number will decrease even further as newer CMOS technologies with smaller feature sizes become available, it can be assumed that switching at a rate of more than 100M connections per second might be feasible in the near future. This point will be addressed further in the next chapter.

4.6.5 Latency and throughput of PARCOS I

The latency and throughput of PARCOS I is determined by the input-output path through the communication matrix. The maximum delay occurs in PARCOS I with broadcast mode, where one input is connected to all 32 outputs. A block diagram of a worst-case delay path in PARCOS I is shown in figure 4.13. The path passes through the following elements: Input pad, top-level chip wiring to the inverter chain driver, driver, fanout to 32 pairs of selector chains, buffer, wiring to the output pad, and the output pad. We calculate the delay in each of these segments separately to point out where the delay can be improved in designing future versions PARCOS I.

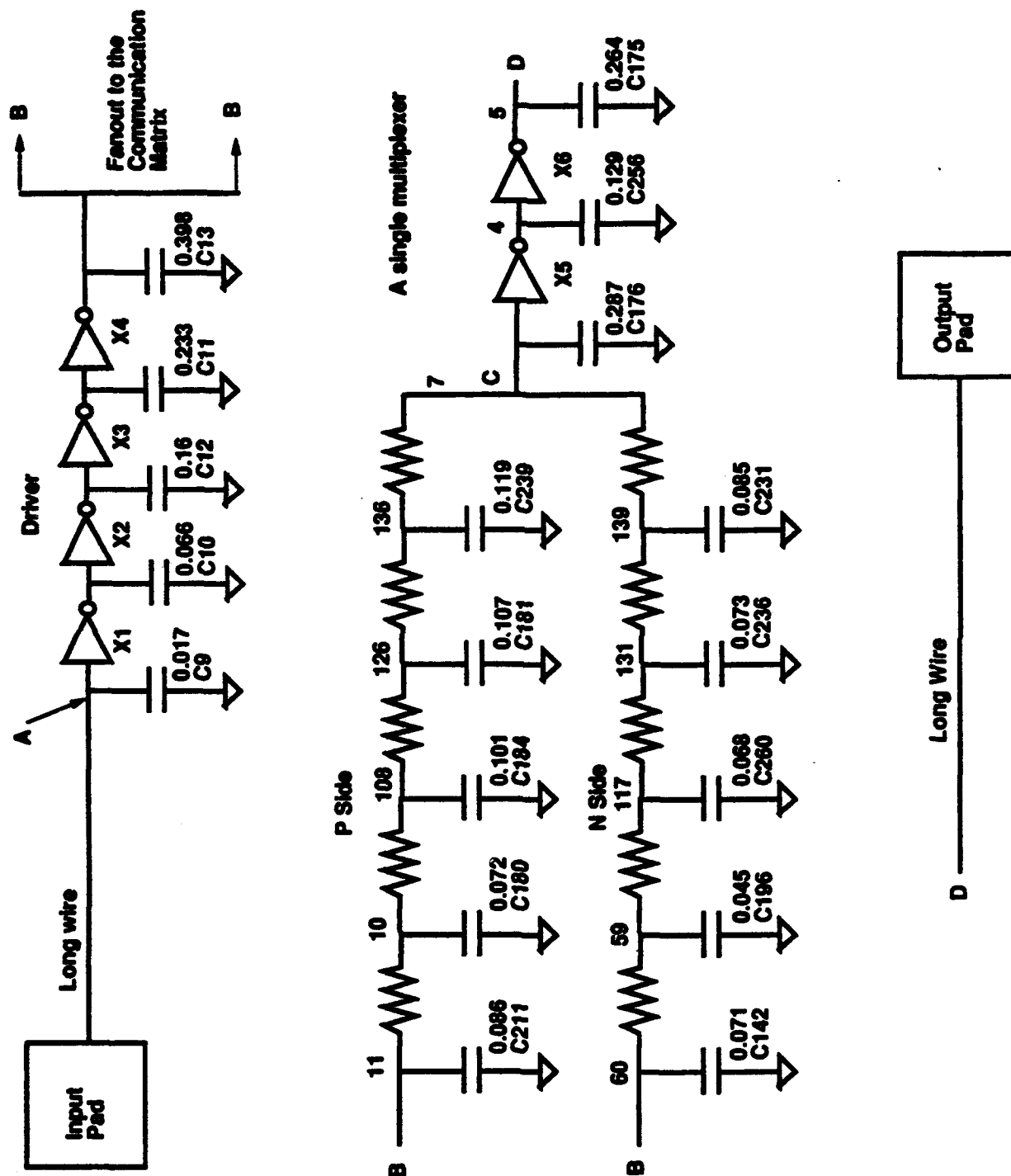


Figure 4.13. Worst delay path in PARCOS I

Wire segment: The longest wire from an input pad (as well as an output pad) is about 2000μ long and 8μ wide. Using the actual fabrication parameters, in which sheet resistance is 0.051 ohms/sq and area capacitance is $0.056 \text{ fF}/\mu^2$ (layout to substrate + edge capacitance) for Metall in 2μ P-well CMOS process, the capacitance of the wire is about

$$0.056 \times 10^{-15} \times 2000 \times 8 = 0.9 \text{ pF}$$

and its resistance is about

$$0.051 \times \frac{2000}{8} = 13 \text{ ohms}$$

and the time constant is 11.5 pS . The delay in the wires is thus negligible. However, the capacitance of the wires has implications for the delays in their drivers.

I/O Pads: From the data provided by MOSIS and our own experience with the measured delay in previous fabrications, the combined delay in an input-output pad pair is about 12 nS .

Driver: A detailed circuit diagram of the driver is shown in figure 4.14. Various capacitance values were extracted using Spice³. It can be seen from the figure that the nfets and the pfets are smaller than the optimum necessary for equal rise and fall times in the inverters X_1, X_2, X_3 , and X_4 , which was mainly done to save die area. For a moment, let us assume that the rise and the fall times in the inverters are the same. We calculate the delay in the driver as follows (details of the method used can be found in [Shoji 88, pp 366]).

The cascaded inverters in figure 4.14 consist of individual stages that have pullup delay time T_U and pulldown delay time T_D given by

$$T_U(i) = \tau_P \frac{P_{i+1} + N_{i+1} + f(P_i + N_i)}{P_i} \quad (4.13)$$

and

$$T_D(i) = \tau_N \frac{P_{i+1} + N_{i+1} + f(P_i + N_i)}{N_i} \quad (4.14)$$

Where P_i and N_i are the sizes of the pfet and the nfet of the i th stage. τ_P is the time constant defined by the product ($R_P C$) of the gate capacitance of a unit-size FET (the nfet and pfet

³The capacitance values shown are for a 3μ process. The 32×32 PARCOS I chip was fabricated using a 2μ P-well technology. These capacitance values cannot be scaled easily to get values for a 2μ technology. The capacitance values extracted by Spice account for all the capacitances in the circuit including gate capacitance, Miller capacitance and drain island capacitance. Out of these, only gate capacitance can be scaled, and then only if the gate oxide thickness is the same in both processes. In general even the gate oxide thickness changes with the feature size

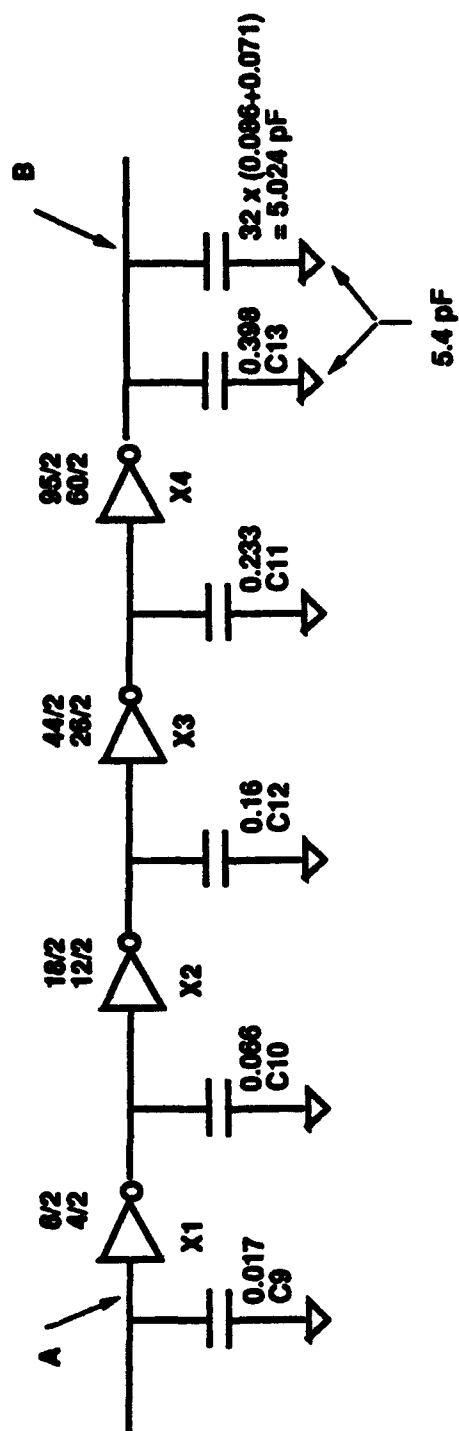


Figure 4.14. Driver for the selector trees

are assumed to be the same size) and the equivalent resistance of a unit-size pfet. τ_N is defined in the same way for nfets. Parameter f is the ratio of drain island capacitance to the gate capacitance. It varies between 1-2. If we let the pullup and the pulldown delays be the same, we get

$$\frac{\tau_P}{P_i} = \frac{\tau_N}{N_i} = \frac{\tau}{S_i} \quad (4.15)$$

Where $S_i = P_i + N_i$ is the size of the gate, and $\tau = \tau_P + \tau_N$ is the gate pair delay which is determined from the ring oscillator frequency for the fabrication technology. The ring oscillator frequency for the 32×32 PARCOS I process was 26.5 MHz for 31 stages with a pullup to pulldown size ratio of 2. This gives a pair delay for a minimum size inverter (pullup 12/2, pulldown 6/2) of

$$\frac{1}{31} \times \frac{1}{26.5 \times 10^6} = 1.22nS = \tau \quad (4.16)$$

Substituting these values in formulae for the delays in the individual stages of the driver, we get a total delay of

$$T_{D-Driver} = \tau \left[\left(\frac{S_2}{S_1} + f \right) + \left(\frac{S_3}{S_2} + f \right) + \left(\frac{S_4}{S_3} + f \right) + \left(\frac{(C_L/C_0)S_1}{S_4} + f \right) \right] \quad (4.17)$$

The term C_L/C_0 accounts for the "load" of 32 MUXs in terms of a minimum size inverter. $C_L = 32 \times (0.086 + 0.071) = 5.024$ pF⁴. From Spice simulation and data from other similar technologies, we know that $C_0 = 0.07$ pF (The output mode capacitance of an inverter with PU = 6/2 and PD = 4/2 with a single fanout. S_1 has PU = 6/2 and PD = 4/2). The variable f is the ratio of drain island capacitance to the input gate capacitance, which varies between 1 and 2 in practice.

$$T_{D-Driver} = \tau \left[\frac{18 + 12}{6 + 4} + \frac{44 + 26}{18 + 12} + \frac{95 + 60}{44 + 26} + \frac{(5.02/0.07) \times (6 + 4)}{95 + 60} + 4 \times f \right] \quad (4.18)$$

using a value of $f = 1$ (as suggested in the literature for technologies such as ours) this gives

$$T_{D-Driver} = 1.22(3 + 2.33 + 2.21 + 4.63 + 4) = 19.73nS \quad (4.19)$$

The design can be optimized further by resizing the transistors so that every stage has the same delay. In that case, the delay in the driver will be about

⁴These numbers are from the circuit extraction of the actual layout

$$T_{D-Driver} = \tau n \left[f + \left(\frac{C_L}{C_o} \right)^{\frac{1}{n}} \right] \quad (4.20)$$

or

$$T_{D-Driver} = 1.22 \times 4 \left[1 + \left(\frac{5.024}{0.07} \right)^{\frac{1}{4}} \right] = 19.1 nS \quad (4.21)$$

Notice that our design is nearly optimal for 4 stages. The optimal design will need

$$n = \frac{\log_e \left(\frac{C_L}{C_o} \right)}{1 + \frac{1}{n}} = \frac{\log_e \left(\frac{5.024}{0.07} \right)}{1 + \frac{1}{n}} = 3.1 \quad (4.22)$$

or 3 stages. Using a 3 stage design, the delay would have been

$$1.22 \times 3 \times \left[1 + \left(\frac{5.024}{0.07} \right)^{\frac{1}{3}} \right] = 18.9 nS \quad (4.23)$$

In order to obtain the desired logic sense, however, an even number of stages was required.

Multiplexer: A detailed circuit diagram of the multiplexer is shown in figure 4.15. The delay in the inverters X_5 and X_6 can be calculated in a manner similar to the driver and is

$$T_{D-Buffer} = \tau \left[\frac{34 + 17}{12 + 8} + \frac{\left(\frac{0.9}{0.07} \right) \times \frac{6+4}{2}}{\left(\frac{34+17}{3} \right)} + 2 \times f \right] \quad (4.24)$$

$$= 1.22(2.55 + 3.78 + 2) = 10.16 nS$$

The delay in the pass transistor selector tree is calculated as follows. As before, the details of the method can be found in [Shoji 88, pp 275]. Referring to figure 4.15, the transistors in the selector tree can be assumed to be in the ohmic region. Without going into the details of this derivation, it can be shown that the circuit in figure 4.15 is equivalent to the circuit in figure 4.16 where

$$R_A = R_0 + R_1 + R_2 + R_3 + R_4 \quad (4.25)$$

$$R_B = R_5 + R_6 + R_7 + R_8 + R_9 \quad (4.26)$$

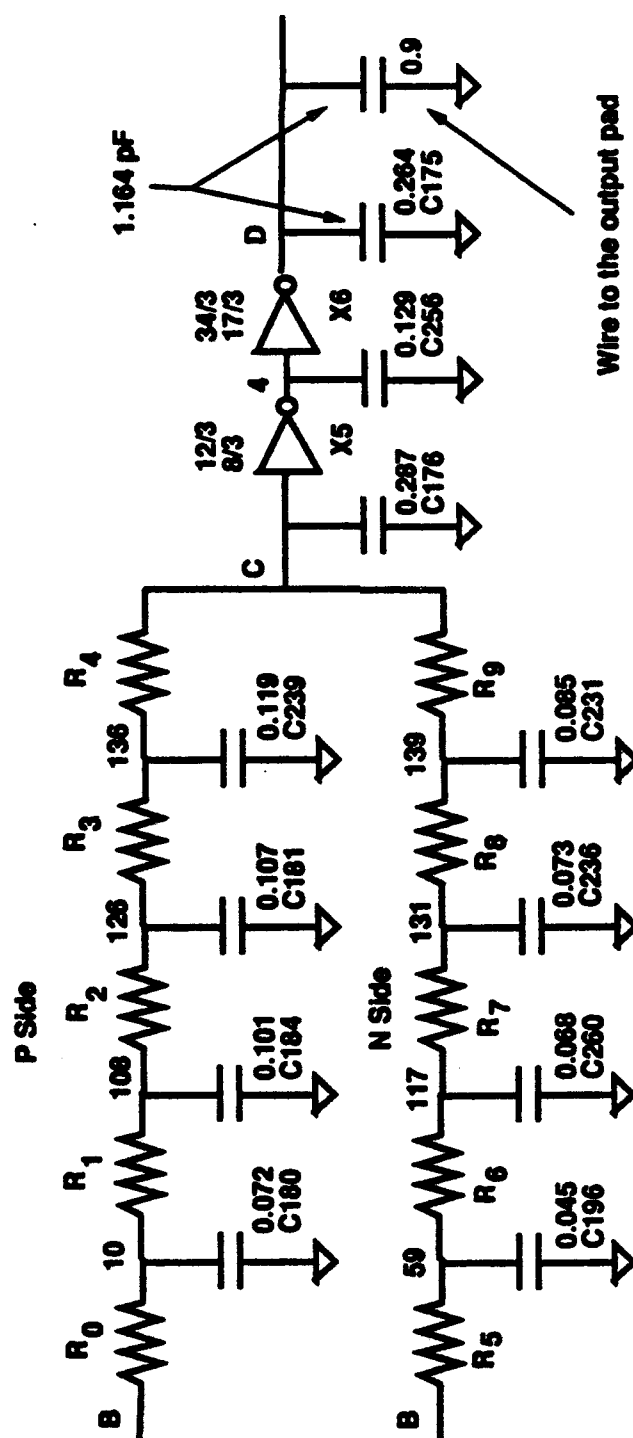


Figure 4.15. Selector tree and buffer

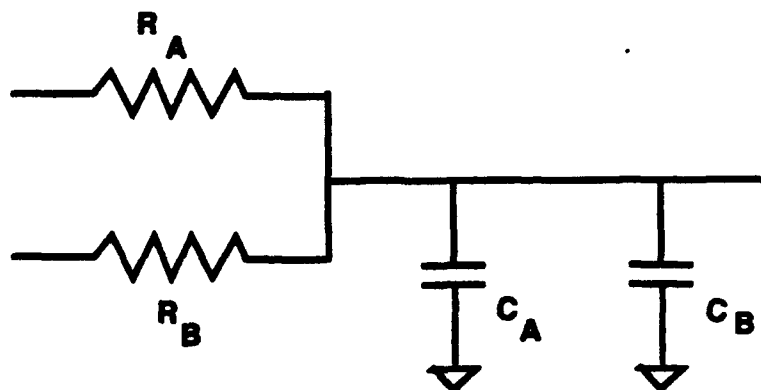


Figure 4.16. Equivalent circuit for a selector tree

$$C_A = \frac{1}{R_A} \left[\begin{array}{l} R_0 (C_{180} + C_{184} + C_{181} + C_{239} + \frac{1}{2}C_{176}) + \\ R_1 (C_{184} + C_{181} + C_{239} + \frac{1}{2}C_{176}) + \\ R_2 (C_{181} + C_{239} + \frac{1}{2}C_{176}) + \\ R_3 (C_{239} + \frac{1}{2}C_{176}) + \\ R_4 (\frac{1}{2}C_{176}) \end{array} \right] \quad (4.27)$$

and

$$C_B = \frac{1}{R_B} \left[\begin{array}{l} R_5 (C_{198} + C_{280} + C_{236} + C_{231} + \frac{1}{2}C_{176}) + \\ R_6 (C_{280} + C_{236} + C_{231} + \frac{1}{2}C_{176}) + \\ R_7 (C_{236} + C_{231} + \frac{1}{2}C_{176}) + \\ R_8 (C_{231} + \frac{1}{2}C_{176}) + \\ R_9 (\frac{1}{2}C_{176}) \end{array} \right] \quad (4.28)$$

Using an approximate resistance value for P-channel of 15000 ohms/sq (/sq is a dimensionless unit) and 6000 ohms/sq for N-channel [Mukherjee 86, pp 156] and noting that the pfets in the selector tree are 6/2 and the nfets are 4/2 in size, we get

$$R_0 = R_1 = R_2 = R_3 = R_4 = 15000 \times \frac{2}{6} = 5000 \text{ ohms}$$

and

$$R_5 = R_6 = R_7 = R_8 = R_9 = 6000 \times \frac{2}{4} = 3000 \text{ ohms}$$

Thus

$$R_A = 5 \times 5000 = 25000 \text{ ohms}$$

$$R_B = 5 \times 3000 = 15000 \text{ ohms}$$

$$C_A = \frac{5000}{25000} \left(\frac{0.287}{2} \times 5 + 0.119 \times 4 + 0.107 \times 3 + 0.101 \times 2 + 0.072 \right) = 0.36 \text{ pF}$$

and

$$C_B = \frac{3000}{15000} \left(\frac{0.287}{2} \times 5 + 0.085 \times 4 + 0.073 \times 3 + 0.068 \times 2 + 0.045 \right) = 0.29 \text{ pF}$$

Thus the total delay in the selector tree is

$$\begin{aligned} T_{D-Selector} &= \left(\frac{R_A \times R_B}{R_A + R_B} \right) \times (C_A + C_B) \\ &= \left(\frac{25000 \times 15000}{25000 + 15000} \right) \times (0.36 + 0.29) \times 10^{-12} = 6.1 \text{ nS} \end{aligned} \quad (4.29)$$

From above, the total latency of a worst-case path through the 32×32 PARCOS I is

$$\begin{aligned} T_{D-PARCOS I} &= T_{D-Pads} + T_{D-Driver} + T_{D-Selector} + T_{D-Buffer} \\ &= 12 + 19.73 + 6.1 + 10.16 = 48 \text{ nS} \end{aligned} \quad (4.30)$$

This figure is in close agreement with the delay measured in the fabricated chip, which is 42 nS for a high-to-low transition at the output and 52 nS for a low-to-high transition at the output.

The throughput of the chip is the inverse of its latency multiplied by the number of outputs. It should be noted that the throughput can be increased by buffering the signals inside the PARCOS I chip (currently a path from an input pad to an output pad is fully combinational). The delay in the driver can be reduced by using one driver each for the P-selector tree and the N-selector tree. It should be noted, however, that the delay in the selector tree is the smallest component of the total delay. It can be shown that the delay in a pass transistor selector tree increases in proportion to the square of the number of links it contains. Therefore, for a 128×128 PARCOS I, the delay in the selector tree will be about

$$6.1 \times \frac{7^2}{5^2} = 12 \text{ nS}$$

The contribution of the MUX delay in larger versions of PARCOS I will be small. Further, the effect on delay of smaller feature size is more than linear. For example, a MOSIS 1.6μ CMOS technology has a ring oscillator frequency of about 72 MHz for 31 stages and a 1.2μ technology has a ring oscillator frequency of more than 90 MHz. These figures should be compared with 2μ technology where the ring oscillator frequencies are in the range of

26-35MHz. Thus, the effect of reduced τ due to smaller feature size technologies will provide more than a linear improvement in the PARCOS I latency. We project that, using 1.2μ technology, it is feasible to design a 128×128 PARCOS I chip with a latency of less than 25 nS.

4.7 Comparison of various networks

In this section we analyze and compare the three classes of networks (Crossbar, Clos, Benes) described earlier in this chapter, in terms of (1) Hardware cost, (2) Time to setup and reconfigure, and (3) Latency and throughput.

4.7.1 Hardware cost

Here we are primarily interested in determining the circuit board area required for implementing networks of various sizes. Traditionally, multistage networks have been compared with respect to the number of transistors (or switches) in them. We showed earlier that the number of pins in a VLSI package is a greater impediment to constructing larger size networks on a single chip than is the number of transistors. Some interesting observations can be made concerning the hardware cost of networks when the measure of cost (board area) in terms of VLSI package area is chosen instead of the number of transistors in the circuit. Also the size of the building block chip plays an important role in the total board area for these networks. These results are presented in two forms: Hardware costs of various size networks in terms of a fixed size PARCOS I chip and the effect on the hardware cost of varying the PARCOS I size.

Table 4.1 through 4.7 list the number of chips and the normalized board area required for building various sizes of crossbar, 3-stage Clos, and 3-stage Benes networks using different sizes of PARCOS I packages. For calculating the normalized board area, we assume that the different PARCOS I chips can be packaged in equal-pitch PGAs. PGA area is proportional to the number of pins in the package. The area of a hypothetical 8×8 PARCOS I chip is chosen as one unit. Using a chip area of an 8×8 PARCOS I as the measurement results in some inaccuracies in the board area calculations for larger PARCOS I designs because larger packages don't necessarily use the same lead pitch or even the same packaging technology. Therefore, in practice, the board areas for networks using larger PARCOS I chips will be less

than those listed in the tables. From a known area of 1.44 sq inch for a 32×32 PARCOS I package, we can use an approximation of 0.5 sq inch for the area of an 8×8 PARCOS I, which allows us to get at least a rough estimate of the board areas for larger networks. For example, from table 4.5, the board area for an 8K-input 8K-output Clos network using 128×128 PARCOS I chips will be about

$$6144 \times 0.5 = 3072 \text{ Sq inch}$$

Using 18×12 " boards (roughly the size of a VME board), this will require

$$\frac{3072}{18 \times 12} = 14.32$$

or about 15 boards.

A brief note is in order regarding the blank entries in the tables for Clos and Benes networks. For example, in table 4.3 there is only one optimal 3-Stage Clos network construction (512-input 512-output) using 32×32 PARCOS I chips, and is the largest 3-stage Clos network that can be built with 32×32 PARCOS I chips. For network sizes smaller than 512×512 , a 32×32 PARCOS I is actually being used as two or more smaller PARCOS I switches. The same situation occurs for the blank entries in the Benes network columns.

The reason for computing estimates based on PARCOS I sizes larger than 128×128 is as follows. With currently available PGAs, it is feasible to implement a 128×128 PARCOS I on a single die. A rule of thumb is that the clocking rate on a board is about an order of magnitude less than on a die. Therefore, it is important to construct as large a switch as possible on a single package. Using packaging technologies such as the Thermal Conduction Module (TCM) [Blodgett 82] and multichip modules using Direct Die Mounting methods (see [Bakoglu 90] for an overview), it is feasible to construct larger PARCOS I modules without paying the full penalty of routing signals on board (a module substrate can be clocked faster than a printed circuit board but not as fast as a die). Such PARCOS I modules will further reduce the board area and in turn allow potentially smaller latencies in the entire network. Also, more driver area and power are required for a signal to go from one package to another on a PC board than on the same die or a module substrate. Together, these factors dictate that the number of packages should be minimized in a design.

An additional note is in order as to why we are not considering Clos and Benes networks with more than 3 stages. Using 262144 copies of a 256×256 PARCOS I chip, it is possible to build a 4M-input 4M-output, 5 stage Clos network. Such a network will require a board area of about

$$262144 \times 32 \times \frac{1}{2} = 4,194,304 \text{ Sq inch}$$

About 19,418 $12 \times 18''$ (rough size of a VME board) PCBs will be required to build this network. Therefore, 5 and more stage Clos network designs are impractical. The same applies to 5 or more stage Benes networks. In addition, the control algorithm for 5 or more stage Clos and Benes networks are a lot more time consuming, which limits the usefulness of these networks in large scale multiple-processor systems.

Table 4.1. Various networks built out of 8×8 switch

Network Size	Number of packages required			Relative area ($8 \times 8 \text{ switch} = 1 \text{ unit}$)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
8×8	1	-	-	1	-	-
16×16	8	12	6	8	12	6
32×32	32	24	12	32	24	12
64×64	128	-	24	128	-	24
128×128	544	-	-	544	-	-
256×256	2176	-	-	2176	-	-
512×512	8704	-	-	8704	-	-
$1K \times 1K$	35072	-	-	35072	-	-
$2K \times 2K$	140288	-	-	140288	-	-
$4K \times 4K$	561152	-	-	561152	-	-

Table 4.2. Various networks built out of 16×16 switch

Network Size	Number of packages required			Relative area (8×8 switch = 1 unit)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
16×16	1	-	-	2	-	-
32×32	8	12	6	16	24	12
64×64	32	24	12	64	48	24
128×128	128	48	24	256	96	48
256×256	512	-	48	1024	-	96
512×512	2112	-	-	4224	-	-
$1K \times 1K$	8448	-	-	16896	-	-
$2K \times 2K$	33792	-	-	67584	-	-
$4K \times 4K$	135168	-	-	270336	-	-
$8K \times 8K$	541696	-	-	1083392	-	-
$16K \times 16K$	2166784	-	-	4333568	-	-

Figure 4.17 is a graph showing the relationship of the board area required to construct larger crossbar, 3 stage Clos, and 3 stage Benes networks using various sizes of PARCOS I chips. For example, the ratio of the board areas for a 1-K input 1-K output crossbar network built with 32×32 PARCOS I chips to the same size network built out of 512×512 PARCOS I is

$$\frac{8192}{512} = 16$$

This apparent disparity comes to light only if we use a more practical measure of hardware, the number of packages and their size, rather than the silicon area. There is no correct way of comparing the board areas of a specific size Clos or Benes network built with PARCOS I switches of different sizes. As pointed out earlier, there is only one optimal 3-stage Clos network defined for a specific size of PARCOS I. Therefore, there is only one point each for a specific size 3 stage Clos or a specific size 3 stage Benes network in the graph.

Table 4.3. Various networks built out of 32×32 switch

Network Size	Number of packages required			Relative area ($8 \times 8_{\text{switch}} = 1 \text{ unit}$)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
32×32	1	-	-	4	-	-
64×64	8	12	6	32	48	24
128×128	32	24	12	128	96	48
256×256	128	48	24	512	192	96
512×512	512	96	48	2048	384	192
$1K \times 1K$	2048	-	96	8192	-	384
$2K \times 2K$	8320	-	-	33280	-	-
$4K \times 4K$	33280	-	-	133120	-	-
$8K \times 8K$	133120	-	-	532480	-	-
$16K \times 16K$	532480	-	-	2129920	-	-
$32K \times 32K$	2129920	-	-	8519680	-	-

Table 4.4. Various networks built out of 64×64 switch

Network Size	Number of packages required			Relative area ($8 \times 8_{\text{switch}} = 1 \text{ unit}$)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
64×64	1	-	-	8	-	-
128×128	8	12	6	64	96	48
256×256	32	24	12	256	192	96
512×512	128	48	24	1024	384	192
$1K \times 1K$	512	96	48	4096	768	384
$2K \times 2K$	2048	192	96	16386	1536	768
$4K \times 4K$	8192	-	192	65536	-	1536
$8K \times 8K$	33024	-	-	264192	-	-
$16K \times 16K$	132096	-	-	1056768	-	-
$32K \times 32K$	528384	-	-	4227072	-	-
$64K \times 64K$	2113536	-	-	16908288	-	-

Table 4.5. Various networks built out of 128×128 switch

Network Size	Number of packages required			Relative area ($8 \times 8_{\text{switch}} = 1_{\text{unit}}$)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
128×128	1	-	-	16	-	-
256×256	8	12	6	128	192	96
512×512	32	24	12	512	384	192
$1K \times 1K$	128	48	24	2048	768	384
$2K \times 2K$	512	96	48	8192	1536	768
$4K \times 4K$	2048	192	96	32768	3072	1536
$8K \times 8K$	8192	384	192	131072	6144	3072
$16K \times 16K$	32768	-	384	524288	-	6144
$32K \times 32K$	131584	-	-	2105344	-	-
$64K \times 64K$	526336	-	-	8421376	-	-

Table 4.6. Various networks built out of 256×256 switch

Network Size	Number of packages required			Relative area ($8 \times 8_{\text{switch}} = 1_{\text{unit}}$)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
256×256	1	-	-	32	-	-
512×512	8	12	6	256	384	192
$1K \times 1K$	32	24	12	1024	768	384
$2K \times 2K$	128	48	24	4096	1536	768
$4K \times 4K$	512	96	48	16384	3072	1536
$8K \times 8K$	2048	192	96	65536	6144	3072
$16K \times 16K$	8192	384	192	262144	12288	6144
$32K \times 32K$	32768	768	384	1048576	24576	12288
$64K \times 64K$	131072	-	768	4194304	-	24576

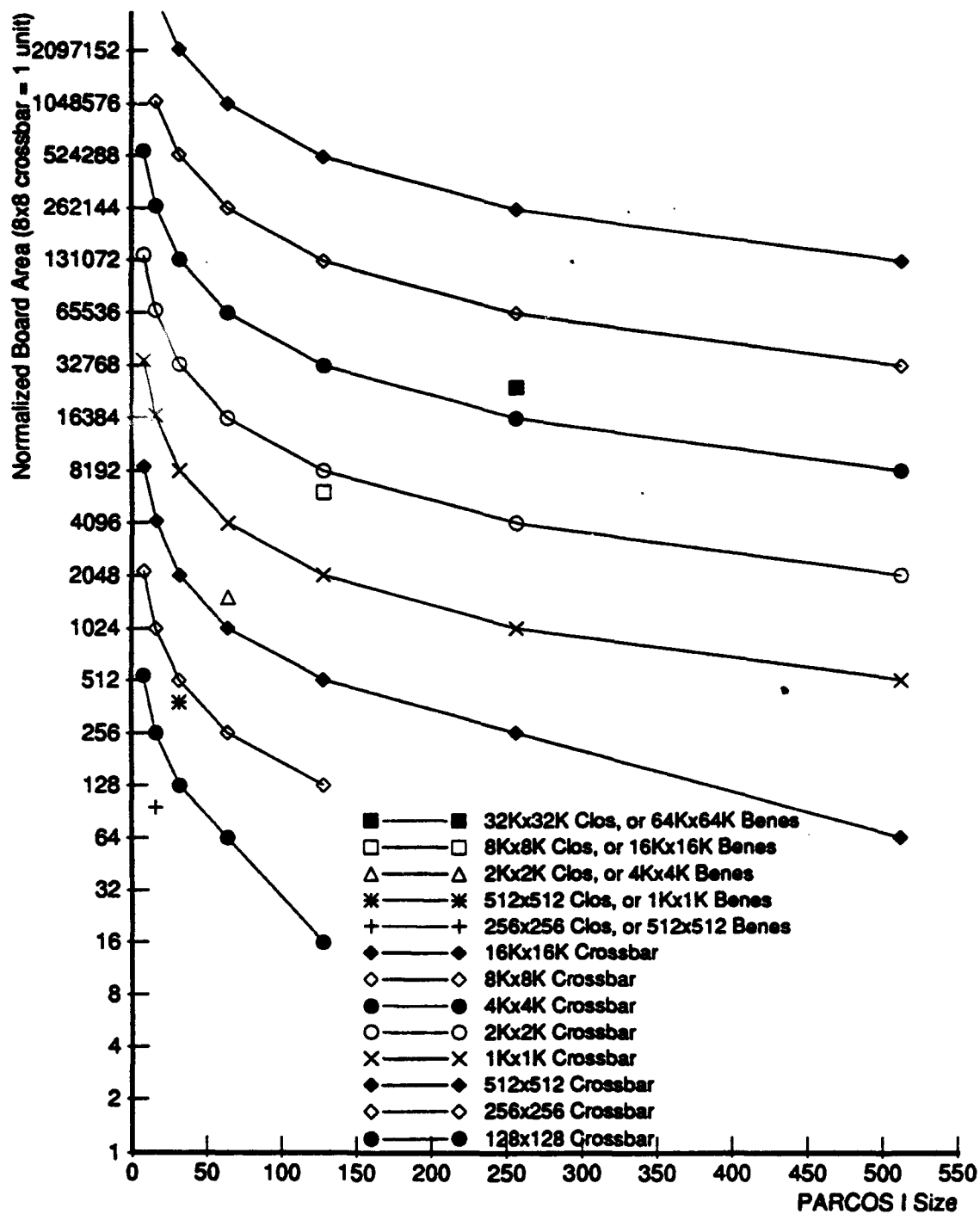


Figure 4.17. Normalized board area vs switch size

Table 4.7. Various networks built out of 512×512 switch

Network Size	Number of packages required			Relative area (8×8 switch = 1 unit)		
	Cross-bar	3-St Clos	3-St Benes	Cross-bar	3-St Clos	3-St Benes
512×512	1	-	-	64	-	-
$1K \times 1K$	8	12	6	512	768	384
$2K \times 2K$	32	24	12	2048	1536	768
$4K \times 4K$	128	48	24	8192	3072	1536
$8K \times 8K$	512	96	48	32768	6144	3072
$16K \times 16K$	2048	192	96	131072	12288	6144
$32K \times 32K$	8192	384	192	524288	24576	12288
$64K \times 64K$	32768	768	384	2097152	49152	24576

4.7.2 Time to set up and reconfigure

The setup time or the time to program an input-output connection pattern is comprised of programming the individual PARCOS I memories, and was discussed earlier in this chapter for the three classes of networks. Crossbar networks do not require any "precomputing," however, Clos and Benes networks do. The cost of "precomputing" will be discussed in the next chapter. If we ignore the "precomputing" time as part of the setup time in these networks, then all three classes of networks can be programmed in N steps for an N -input N -output system. The time to reconfigure from one stored connection pattern to another is equal to one memory cycle time in all of these networks.

4.7.3 Latency and throughput

The PARCOS I communication matrix is a fully combinational circuit. As was discussed in a previous subsection, it is reasonable to expect a 128×128 PARCOS I to have a latency of less than $25nS$. When building larger Crossbar, Clos or Benes networks with PARCOS I building-block chips, two approaches can be followed: Allow a fully combinational circuit from the input to the output of the entire network, or buffer the signals at the individual stages. Clearly the second choice is preferable for high throughput. Suppose a 3-stage Clos network is designed using PARCOS I. The latency of such a network using the first approach will be at least $75nS$ and the throughput per serial link will be less than

$$\frac{1}{75 \times 10^{-9}} = 13.33 \text{ Mb/sec}$$

Using the second approach, however, the latency will still be around 75nS but the throughput will be about

$$\frac{1}{25 \times 10^{-9}} = 40 \text{ Mb/sec}$$

PARCOS I can be designed to have this buffering capability at its inputs and/or at its outputs. Possibilities for buffering signals inside PARCOS I to further increase its throughput were discussed in subsection 4.6.5.

4.8 Conclusions

The major conclusion of this chapter is that crossbar and other dense networks, such as Clos and Benes, are viable design alternatives, even for large scale multiple-processor systems. Such networks, in general, have been considered impractical to build.

To arrive at this conclusion, we presented the architecture of a VLSI PARallel COMmunication Switch (PARCOS I) chip that was designed and fabricated as the building block for the ICAP communication network, as well as a variety of other communication networks. Next, we demonstrated how PARCOS I can be used in building large crossbar, Clos, and Benes networks. To further lend support to the feasibility of constructing such networks, we analyzed the PARCOS I architecture with respect to its hardware cost (die area), pinout requirements, power dissipation, time to set up and reconfigure the switch, and its latency and throughput. An interesting conclusion of this study is that even though the layout area grows proportional to $n^2 \times \log_2 n$, the PARCOS I design is pin limited rather than silicon area or power dissipation. This warrants the design of large crossbars on a single chip or module, limited only by the packaging technology. Finally, we analyzed and compared the aforementioned networks with respect to their hardware cost, time to set up and reconfigure, and the latency and throughput. It was shown that it is practical to build large crossbar networks with up to a few K inputs, or large Clos and Benes networks with up to a few tens of K inputs, using state of the art technologies. The following is a summary of the goals achieved in this thesis so far and the remaining goals.

4.8.1 Goals achieved

The objective of the generation 1.0 design was to address the requirements of a multiple-processor network when the target parallel processor is used in a SIMD mode or in synchronous MIMD (SMIMD) mode. In this chapter we addressed the requirements of a multiple-processor network when the interprocessor communication patterns in the target parallel processor are fixed and known apriori. We showed that under central routing control, these networks can be programmed and reconfigured in the best possible time.

4.8.2 Goals remaining

This design does not work well in situations where the communication patterns are not known apriori (e.g. in data dependent communication), which are common in MIMD parallel processors and some SIMD processors such as the CM2 and the MassPar. In the next chapter we will extend the design to partially address the requirements of such systems.

CHAPTER 5

GENERATION 1.5: CENTRAL ROUTING CONTROL

In this chapter we address the requirements of a multiple-processor network when the interprocessor communication cannot be determined apriori, but, once established, the pattern is likely to be used later. Also, the target parallel system is operated in a SMIMD or a MIMD mode.

To support fine-grained low latency self routing in the networks discussed in the previous chapter, we follow a two stage approach. First, in this chapter we discuss a series of designs that can be built by using simple additional custom hardware besides PARCOS I chips, to provide an interim solution to the problem of supporting data dependent interprocessor communication at the ICAP level of the IUA. These designs use central routing control which in general, is serial in nature. Therefore, these designs are not optimal. We call these designs, Generation 1.5 design - i.e. these designs besides being interim solutions for the ICAP communication network, serve as stepping stones to the Generation 2 design.

Even though Generation 1.5 designs are inadequate to satisfy all of our goals, they serve two important purposes. First, they are valuable exercise because they show that it is possible to construct networks that fit the above mentioned section of the requirements space, and provide insight that will lead to the self-routing designs. Second, they provide backup solutions to the ICAP communication network, in case the Generation 2 design should encounter problem or delays.

The routing control in Generation 1.5 design is central. We will first develop simple extensions of the existing designs to support synchronous data-dependent routing. In data dependent synchronous routing, a custom VLSI chip can be used to determine whether all of the communication requests have been fulfilled. This chip was originally designed to construct the IUA feedback concentrator mechanism. After developing extensions to support synchronous data-dependent communication, we will further extend the design to support asynchronous data-dependent routing.

The rest of this chapter is organized as follows. The IUA feedback concentrator and its building block chip are described first as a self-contained section. In section 5.2 we discuss

the extensions to support synchronous data-dependent routing, followed by discussion of extensions to support asynchronous data-dependent routing. Multicast is a useful capability in multiple-processor systems. Issues related to supporting multicast operations in these networks will be discussed in section 5.4. The designs developed in this chapter will be analyzed and compared in section 5.5, followed by an evaluation of Generation 1.5 design.

5.1 The IUA feedback concentrator

Recall from section 1.2 that the control for the CAAPP and the ICAP is provided by the SPA, using associative processing techniques, and the CAAPP level is especially oriented towards associative processing with an emphasis on fast global summary feedback mechanisms supported in hardware. This section provides the details of the IUA feedback concentrator and its building block chip. The IUA feedback concentrator implements the associative processing primitives in the IUA, to be discussed shortly. In data dependent synchronous routing, the IUA feedback concentrator can be used to determine whether all of the communication requests have been fulfilled. The details of the feedback concentrator and its building block chip are organized as this self contained section and can be skipped, without losing continuity in this thesis.

Many parallel algorithms, when mapped onto multiple-processor systems, require gathering data from all of the processors to generate a global value. In SIMD parallel processors, this global value can be used by the central controller either as data in scalar processing or as input to control flow decisions. Many low- and intermediate-level vision algorithms are characterized by this kind of processing. Making a global decision in fine-grained SIMD parallel processors is equivalent to a test-and-branch operation in a uniprocessor, and should have a similar time cost. Thus a fast global feedback mechanism is required in the system.

All of the above observations apply to the IUA. Fine-grained control at the low and the intermediate levels in the IUA is achieved by using associative processing techniques. Foster [Foster 76] has identified four key processing capabilities in associative computation:

- (1) Global broadcast/local compare/ activity control.
- (2) Select a single responder.
- (3) Some/None response, and
- (4) Count-responders.

We present the architectural details of the last two of these capabilities as they are

embodied in the IUA feedback concentrator, and demonstrate their power in parallel processing by giving a few example algorithms.

The rest of this section is organized as follows. Next we present the details of the feedback concentrator mechanism as implemented in a custom VLSI chip, which uses an innovative combination of circuit techniques to achieve high speed. A brief description of the chip is provided thereafter. Next we compare the performance of our architecture with a mesh connected processor without the feedback mechanism for three common low-level vision tasks, followed by a brief overview of the plans for the feedback concentrator for the next generation of the IUA.

5.1.1 The IUA Feedback Concentrator Architecture

The hardware comprising the two lower level processors of the IUA is built from 4096 CAAPP chips and 4096 ICAP processor chips, along with memory chips and the associated interface and I/O circuitry. Each CAAPP chip contains 64 bit-serial CAAPP PE's, their local memory, and interface logic. A CAAPP chip together with an ICAP processor constitutes what we call a node. The system is divided into 64 motherboards each containing 64 nodes.

Each CAAPP PE has a response register called the X register, and an activity register called the A register. The fastest method of counting the number of responders would be to feed the 1-bit output from the X registers of the $512 \times 512 = 262,144$ CAAPP PEs to a hardware adder and generate a 19 bit sum for the ACU. One technique, proposed by Favor [Favor 64] for counting the number of responders, uses a pyramid of full adders to generate the least significant bit of the count of the inputs. Next, the carries generated at various stages of the adder pyramid are fed to a second smaller pyramid to generate the second least significant count bit. Successively smaller adder pyramids are chained together to develop the full count. In this method each stage waits for one count bit to be formed before feeding any carries from the current level to the next. Foster [Foster 71] improved upon Favor's scheme such that at any stage of the current adder pyramid, as soon as 3 carries are available, they are fed to the next level adder pyramid. Foster's scheme, called a carry-shower adder, results in a significant speedup over Favor's scheme. For example, Favor's scheme has a delay of 21 full adders for 64 inputs, whereas Foster's scheme has a delay of 10 full adders for 64 inputs. In another study, Swartzlander [Swartzlander 73] corrected Foster's lower bound formula and showed that the theoretical lower bound on the delay is 9 full adders for 64 inputs. But the best known actual circuit has a delay of 10 full adders. Further, Swartzlander proposed a "faster" scheme for the counting hardware by

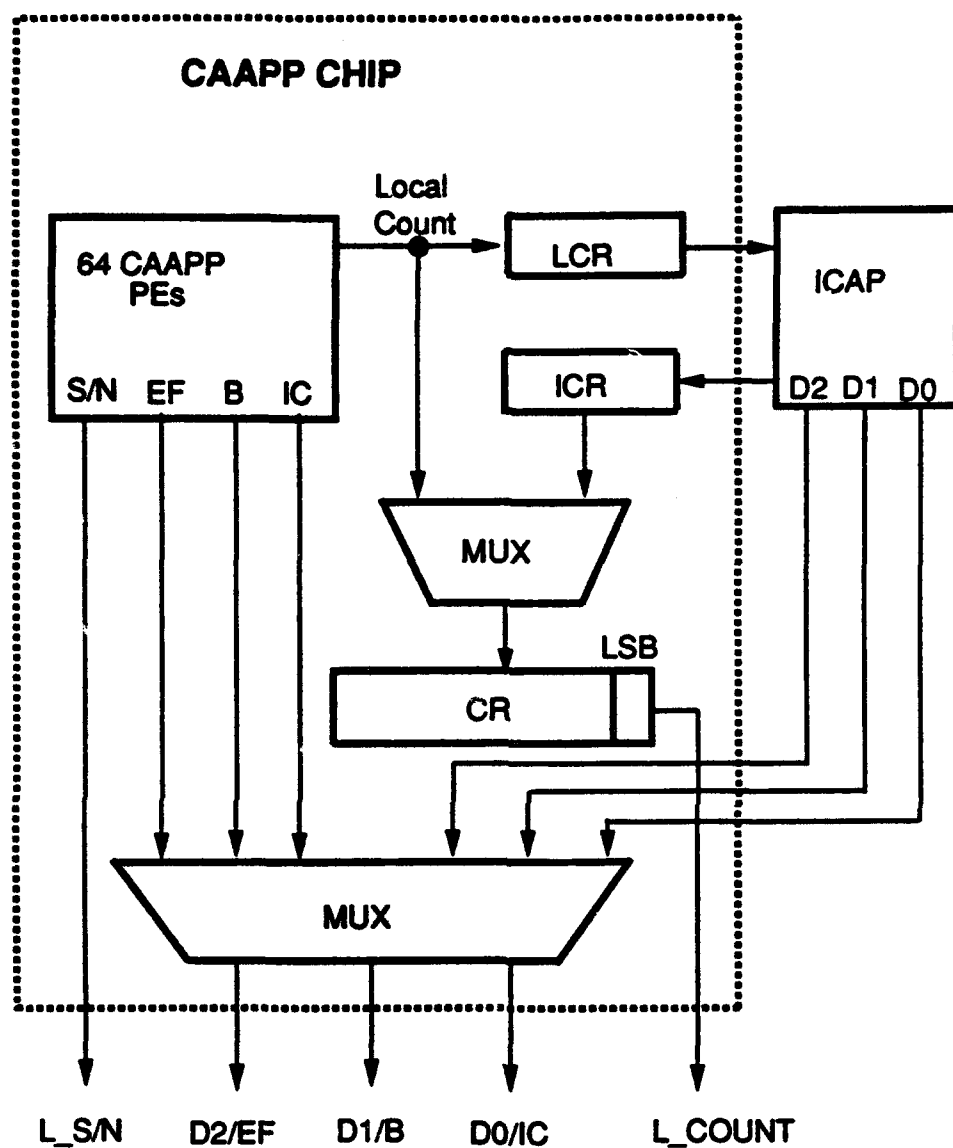
using Read-Only Memory (ROM). However, his scheme is suitable for only small numbers of inputs, and he assumed that the delay in a ROM is independent of its size. The fan-in from 262,144 processors to a single sum is too great to be practically realized this way.

In the IUA design, the pin constraints on the processor chip, and space and pin limitations for the external circuitry required that the chip-level counts be output serially. Figure 5.1 illustrates the logical organization of the Some/None and the Count-responder mechanisms on one node. Local Some/None is the logical sum of the X register of the 64 CAAPP PEs on a node, generated at the end of every CAAPP instruction by using Foster's scheme within the processor chip. A special instruction, called `latch_count`, allows either the CAAPP count or an 8-bit value from the associated ICAP processor to be loaded in the node count register (CR). The value in the CR can be read out serially from the CAAPP chip. A separate instruction, called `latch_local_count`, allows the CAAPP count to be loaded in the local count register, where it can be read by the corresponding ICAP processor. Three additional general purpose signals $D_0 - D_2$ are provided by the ICAP and are multiplexed under program control with three special purpose feedback signals from the CAAPP chip.

From the nodes, one option is to use an instruction to shift out each bit of the count register. However, this idles the processors during output of the count. Our analysis showed that typically some short operation is performed between the counts in a burst. The operation may be as simple as loading another bit in the response register, but is likely to be a comparison (for example, in computing a histogram). Thus, it was decided to use a finite state machine that automatically shifts the count out of the chip, least significant bit first, one bit per cycle. Output begins as soon as a count is latched. The processors are thus able to overlap computation with the development of the current count.

There are many ways to sum the node counts. One method would be to feed the 8-bit values from 4096 nodes to a hardware adder such as a Carry Save Adder (CSA) [Cavanagh 84]. This approach is infeasible because of the hardware cost and the number of wires.

Our final solution is to trade a small amount of the speed of counting responders for a substantial saving in the hardware and the number of wires – we serialized the entire adding process. This scheme is shown in figures 5.2 and 5.3. Figure 5.2 illustrates the Some/None and the Count-responder mechanism on one motherboard. The local Some/None (LS/N) outputs from the 64 nodes are fed into the motherboard Some/None tree. (The $D_0 - D_2$ feedback signals are treated in the same way.) The local counts (L.Count) from the count register (CR) of the 64 nodes are serially fed into the motherboard count responder tree. Figure 5.3 illustrates the global Some/None and Count-responder mechanisms for the full IUA, which concentrate the outputs of the motherboard level networks in a pipelined



D2, D1, D0 : ICAP Status Lines
EF, B, IC : ICAP/CAAPP Status Lines
S/N : Local Some/None line on a Node
LCR : Local Count Register
ICR : ICAP Count Register
CR : Node Count Register

Figure 5.1. Node Some/None and Count network

manner and are similar in design.

Before describing the functioning of the feedback concentrator, we discuss the design of a custom VLSI chip that was built to implement all four blocks of figures 5.2 and 5.3.

5.1.2 The feedback concentrator building-block chip

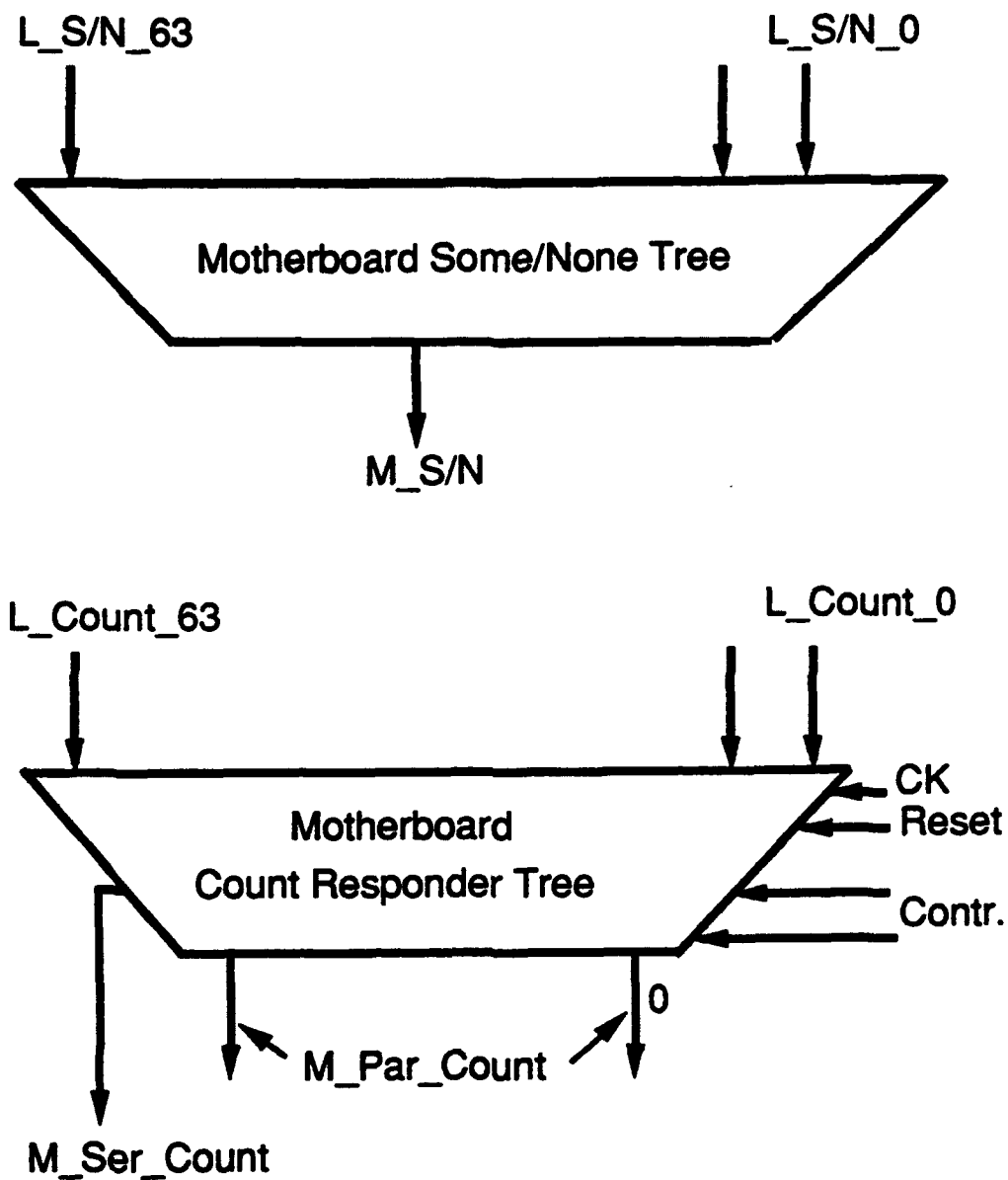
A schematic of the concentrator chip is shown in figure 5.4. It comprises four hardware blocks: a carry-shower adder, registers, an auxiliary logic unit, and a carry-select adder. The CAAPP PE's operate with a 100nS cycle time, therefore the delay from the inputs to the outputs was constrained to 100nS. Additionally the delay through the second adder was constrained to 25nS. Also, the technology was limited to a MOSIS 2-micron CMOS process and an 84-pin package.

The carry shower adder is designed using a full adder cell and is similar to the one proposed by Foster, as discussed in the previous section. The carry-shower adder generates a 7 bit sum from 64 (one-bit) inputs and has a delay of about 50nS.

Another hardware block of the concentrator chip is a pair of registers D_Reg_1 and D_Reg_2, whose sizes are 7- and 6-bits respectively. The clock-to-output delay in the registers is about 6nS.

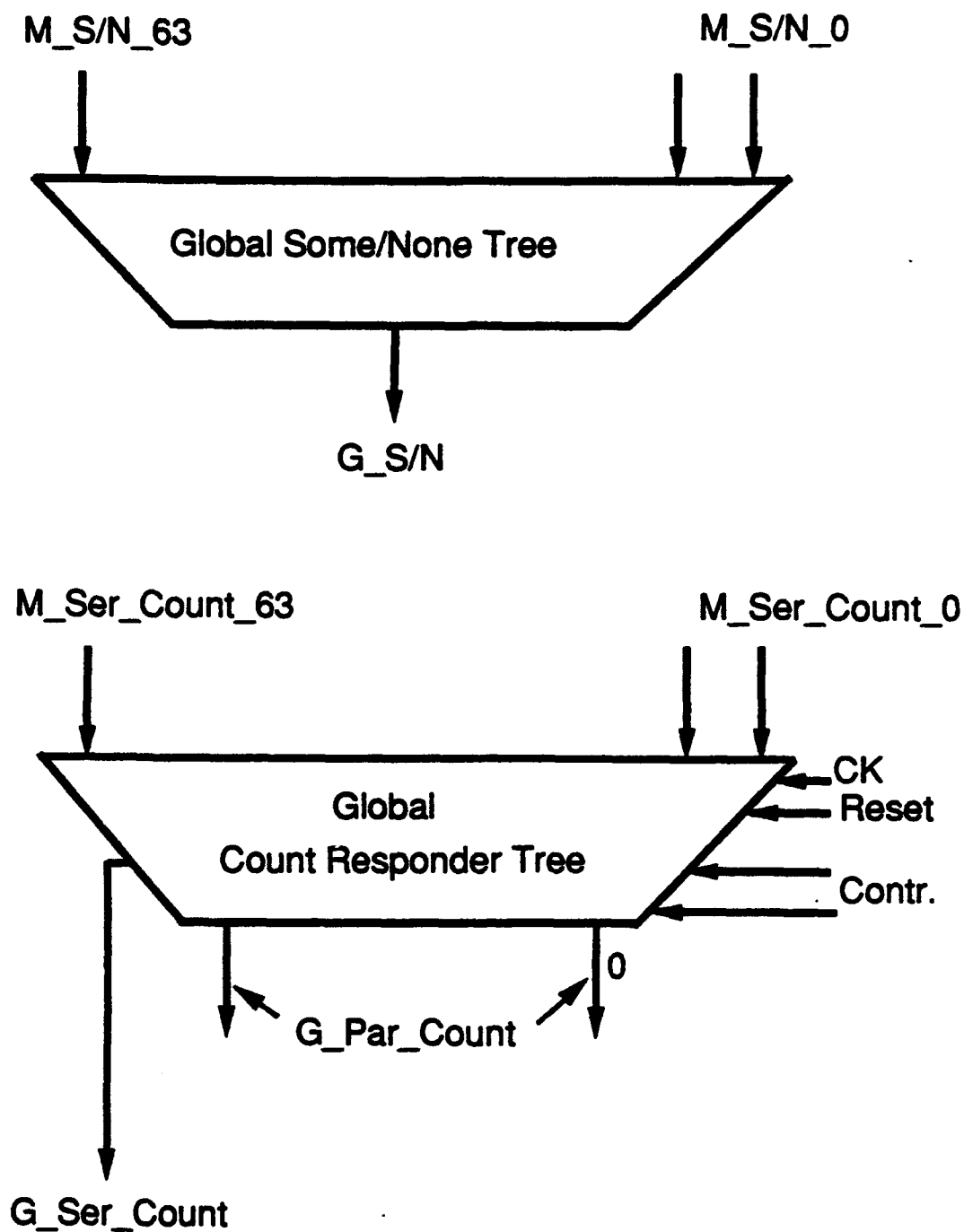
The auxiliary logic unit is used to generate the logical AND, logical OR and logical EXOR of the 64 inputs to the concentrator chip. The inputs to the auxiliary logic unit are taken of the 7 binary outputs of D_Reg_1. The logical OR is used for building the Some/None circuitry at the motherboard and the global level. The logical AND and the logical EXOR are provided for future use. The auxiliary logic unit has a delay of less than 10nS.

The last major block of the concentrator chip is a 7 bit carry-select adder. Its design was particularly critical to the overall speed and timing of the chip. Subtracting 6nS for the D register delay and 4nS for the delay in the output pads from the 25nS goal left a maximum of 15nS of allowable delay in the 7 bit adder. A ripple carry adder using 7 adder cells is not adequate, because each cell would have a 5nS delay for a total of 35nS. Also, a 7 bit carry look-ahead adder cannot be constructed in the available technology with a delay of less than 15nS. We achieved the desired speed by trading more hardware (VLSI chip area) for speed, through the use of a 7 bit carry-select adder [Cavanagh 84]. A microphotograph of the feedback concentrator chip is shown in figure 5.5.



(Networks for other 3 inputs are similar to L_S/N)

Figure 5.2. Motherboard Some/None and Count Networks



(Networks for other 3 inputs are similar to M_S/N)

Figure 5.3. Global Some/None and Count Networks

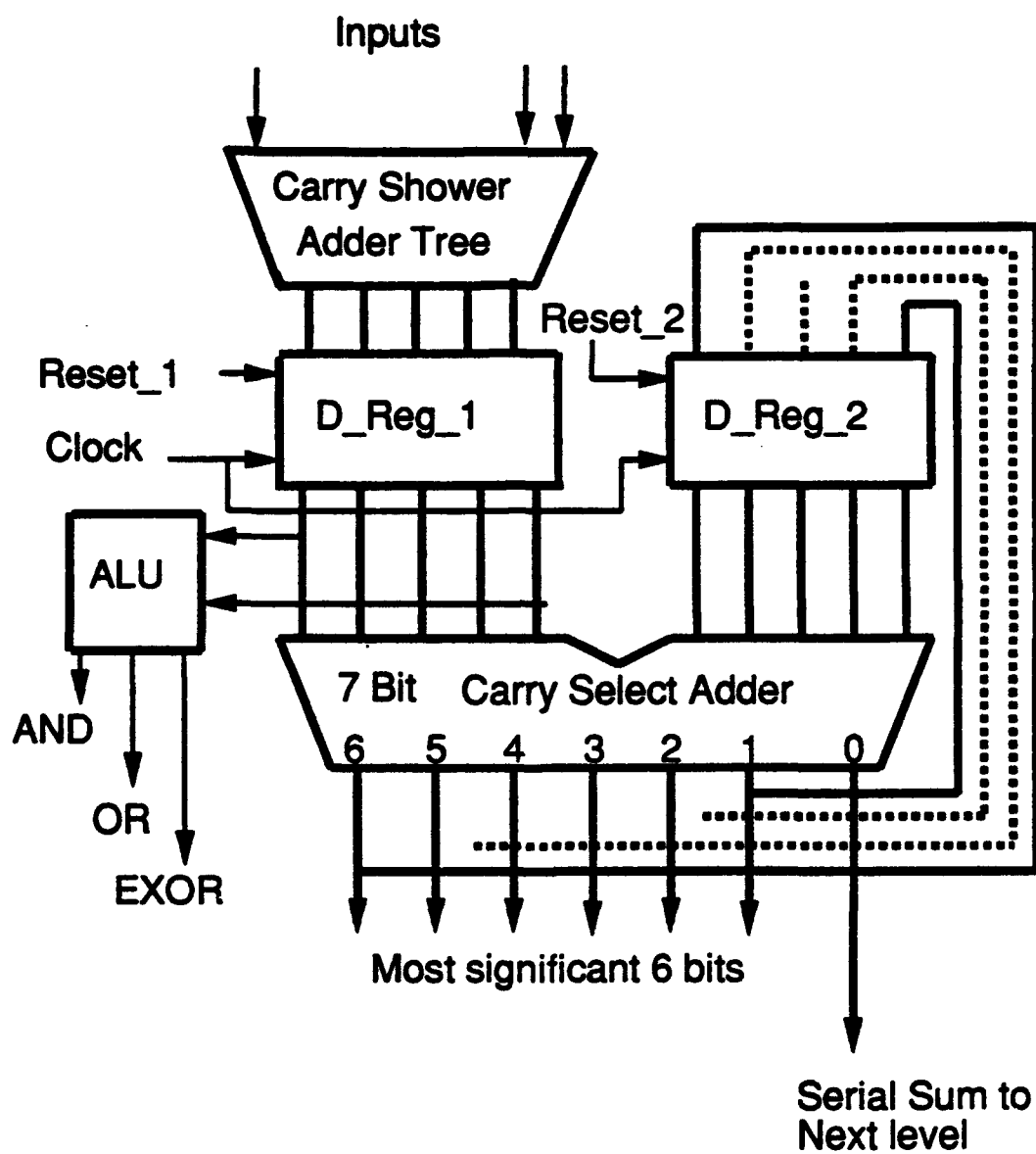


Figure 5.4. Schematic of the Concentrator Chip

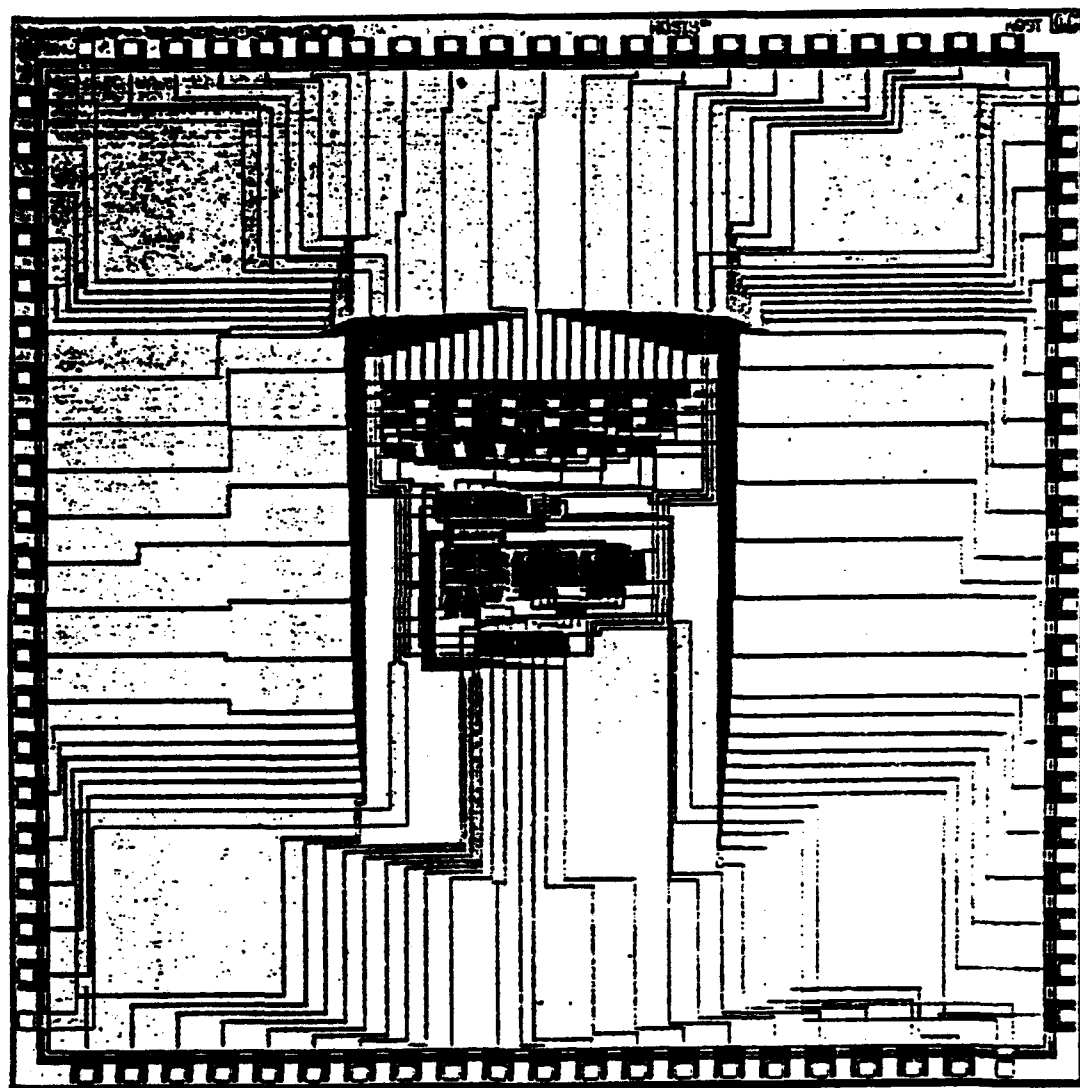


Figure 5.5. Microphotograph of the concentrator chip

5.1.3 The IUA feedback concentrator operation

When the concentrator chip is used for computing global Some/None, it functions as follows. The logical sum (L.S/N) of the response registers (X registers) on a node stabilizes at the end of the instruction cycle (this operation is carried out in every CAAPP instruction cycle). In the second cycle, a logical sum of the 64 nodes is computed for every motherboard. By the end of the second cycle, the M.S/N from the 64 motherboard concentrator chips is ready at the inputs of the global concentrator chip. Meanwhile, in the second cycle, the L.S/N from the next instruction passes through the motherboard concentrator chips in a pipelined manner. By the end of the third cycle (or $0.3\mu\text{S}$), a global Some/None is available, the second L.S/N is at the inputs of the global concentrator and a third L.S/N is at the inputs of the motherboard concentrator.

When the concentrator chip is used for counting the responders in the CAAPP or for summing 8-bit values from the ICAP, it functions as follows. One cycle after the low-order bits of the chip-level counts are input, the 7-bit motherboard-level result appears. If another set of bits are input, the high-order 6-bits are recirculated and added to the next result through the carry-select adder. The low-order bit is output to the next stage of the count circuit. The process can be repeated to sum 64 inputs of any length with the low-order portion of the result being shifted out serially and the high-order 6 bits available in parallel one cycle after the last set of bits is input. To serially output the entire result, zeros are input after the last set of bits. By the end of the third cycle, the least significant bit of the final count appears at the serial output of the global concentrator chip, and by the end of 16 cycles ($1.6\mu\text{S}$), the last of the low-order bits is output (assuming 8 bits are output by the daughterboards) and the high-order portion of the final count is available in parallel.

5.1.4 Sample algorithms

In this section we discuss three sample algorithms, used extensively in low-level vision tasks. We provide their exact running time for a 512×512 image on the same size CAAPP array, and then compare these times with another 512×512 mesh-connected SIMD architecture but without the Some/None and the Count-responders mechanisms. The algorithms presented here are merely intended to demonstrate the power of the two feedback mechanisms in the extreme case. There is a great body of literature on other architectures for low-level vision, that provide speedups between these two extremes. We will not compare

these architectures with the CAAPP.

Both the mesh connected processor (MCP), and the CAAPP are assumed to have the same 100nS machine cycle time. For the CAAPP, the feedback response is stored in the array control unit (ACU), whereas the feedback response from the MCP is stored in the top rightmost PE (Possibly to be offloaded later. The top right PE is chosen merely for convenience).

A. Some/None

As mentioned earlier, it takes 3 cycles on the CAAPP or $0.3\mu\text{S}$ for this operation. On the MCP, in the worst case, a 1 from the lower left corner PE will have to be shifted to the upper right hand corner PE. Assuming that the MCP has an instruction that allows a PE to get a value from a neighbor PE's register, OR it with its register value, and store the result in its local register (which is possible in CAAPP), all in one cycle, it will take $2 \times 512 \times 0.1 = 102.4\mu\text{S}$ on the MCP for this instruction.

B. Count Responders

Here we want a count of PEs with 1 stored in their registers. As mentioned earlier, this instruction takes 16 cycles or $1.6\mu\text{S}$ in the CAAPP. In the MCP, the final count length is 19 bits. First we accumulate the counts of the rows in the rightmost PE's. The variable 'count' is kept in each rightmost column PE's memory. By successively shifting right the values in the rows and accumulating in the rightmost PEs, we get a maximum length of 10 for the 'count' variable. For the initial two cycles, the 'count' variable can be a maximum of 2 bits, for the next 4 cycles it can be of 3 bits, for the next 8 cycles it can be of 4 bits, and so on. This gives us a formula to compute the time for computing sums in the rows; which is:

$$2 \times 2 + 3 \times 4 + 4 \times 8 + \dots + 9 \times 256 + 10 \times 1 = 4106$$

or $410.6\mu\text{S}$.

Next the values in the rightmost PEs are accumulated from bottom to top in the rightmost column. The time taken for this operation is given by

$$11 \times 2 + 12 \times 4 + \dots + 18 \times 256 + 19 \times 1 = 8705$$

or $870.5\mu\text{S}$.

The total time taken is thus $1281.1\mu\text{S}$.

It should be noted that the count can be used to quickly compute other statistical

measures, such as mean, median, mode, standard deviation etc., and to compute a histogram of an array of data.

C. Find greatest value

In this algorithm, the goal is to determine the greatest value in a given memory field of the PEs. In the CAAPP, the algorithm begins by loading the high-order bit of a given field into the response register of all cells. The ACU then tests the Some/None output of the CAAPP. If any PEs have their high-order bit set, then they are candidates for the maximum value, in which case any cells that have a 0 in their high-order bit are then deactivated. However, if no cells have their high-order bit set, then none are deactivated because they are still potential candidates. This process repeats with each successively lower-order bit in the field. When the low-order bit has been processed, only those cells that have the maximum value will remain active. For each iteration, the ACU saves the Some/None response so that the maximum value is available in the ACU at the conclusion of processing. A pseudo-code algorithm is shown below.

```

For Bit_Num := Field_Length-1 Down to 0 Do {Begin with the high-order bit}
  Response := Field[Bit_Num]                {Put bit in response register}
  If Some                                     {If any cell has a 1 in this bit}
    Then
      Activity := Response                    {Then turn off activity in cells }
                                              {with a 0 in this bit}

```

This algorithm takes 40 CAAPP instruction cycles or 4.0 μ S for an 8 bit value.

For the MCP, first we find the maximum in each row, and put it in the rightmost PEs. Next we find the maximum in the right column and put the value in the top PE. The basic operation is to compare the fields of two neighbors, and put the greater value in the right hand side PE's memory during the row phase, and in the upper PE's memory during the column phase. It will take some k multiples of 8 instructions for each compare. The value of k for the MCP will depend upon the specific architectural implementation and its instruction set. When this algorithm is emulated on the CAAPP, the value of k is 4. Thus, the total run time for the algorithm on the MCP will be of the order of

$$4 \times 8 \times 2 \times 512 \text{ Cycles}$$

or 3276.8 μ S for the operation.

There are numerous algorithms where a hardware Some/None and Count-responder mechanism will give significant speedups. Our objective in this section was merely to demonstrate a few low-level algorithms that make use of the feedback concentrator mechanism.

5.1.5 Second generation IUA feedback concentrator

Since the development of the IUA prototype started in 1986, VLSI technology has improved substantially in terms of minimum feature size, speed and packaging. The connector technology has also kept pace with the VLSI technology in terms of I/O pin density. These technological improvements have enabled us to start developing the second generation IUA with a substantially higher packaging density.

A smaller version of the second generation IUA, called the *IUA prototype II* comprises 8 daughterboards on a single motherboard. Each daughterboard holds 8 CAAPP chips, 8 ICAP processors, memory chips and associated interface and I/O circuitry. Each CAAPP chip contains 256 CAAPP PEs. Thus there are 64 ICAP processors and 16K CAAPP PEs in the *IUA prototype II*. The full IUA system is expected to have up to 32 motherboards.

In a way similar to the first generation IUA, the Local Some/None is generated at the end of every CAAPP instruction in the *IUA prototype II*. The 8 Some/None lines on every daughterboard are fed in parallel to an 8-input OR gate. The 8 Some/None lines from the daughterboards are fed to an 8-input wired OR backplane trace on the motherboard. The total delay from the generation of the Local Some/None to the generation of motherboard level Some/None is less than 100nS.

The scheme for counting CAAPP PE responders in the *IUA prototype II* is as follows. The 9-bit chip-level counts from 8 CAAPP processors on every daughterboard are serially fed, 2 bits at a time, into a daughterboard concentrator chip. A block diagram of the daughterboard concentrator chip is shown in figure 5.6. A single phase edge triggered flip-flop is used that allows a simpler design of the concentrator chip. The outputs from 8 daughterboards are serially fed, 2 bits at a time, into a motherboard concentrator chip. A block diagram of the motherboard concentrator chip is shown in figure 5.7.

The count scheme for the *IUA prototype II* works in a manner similar to figures 5.2, 5.3, and 5.4. It takes 5 cycles to serially feed the 9-bit CAAPP counts into daughterboard concentrator chips. One additional cycle is used to latch the most significant bit of daughterboard counts at the output of the daughterboard concentrator chip. One more cycle

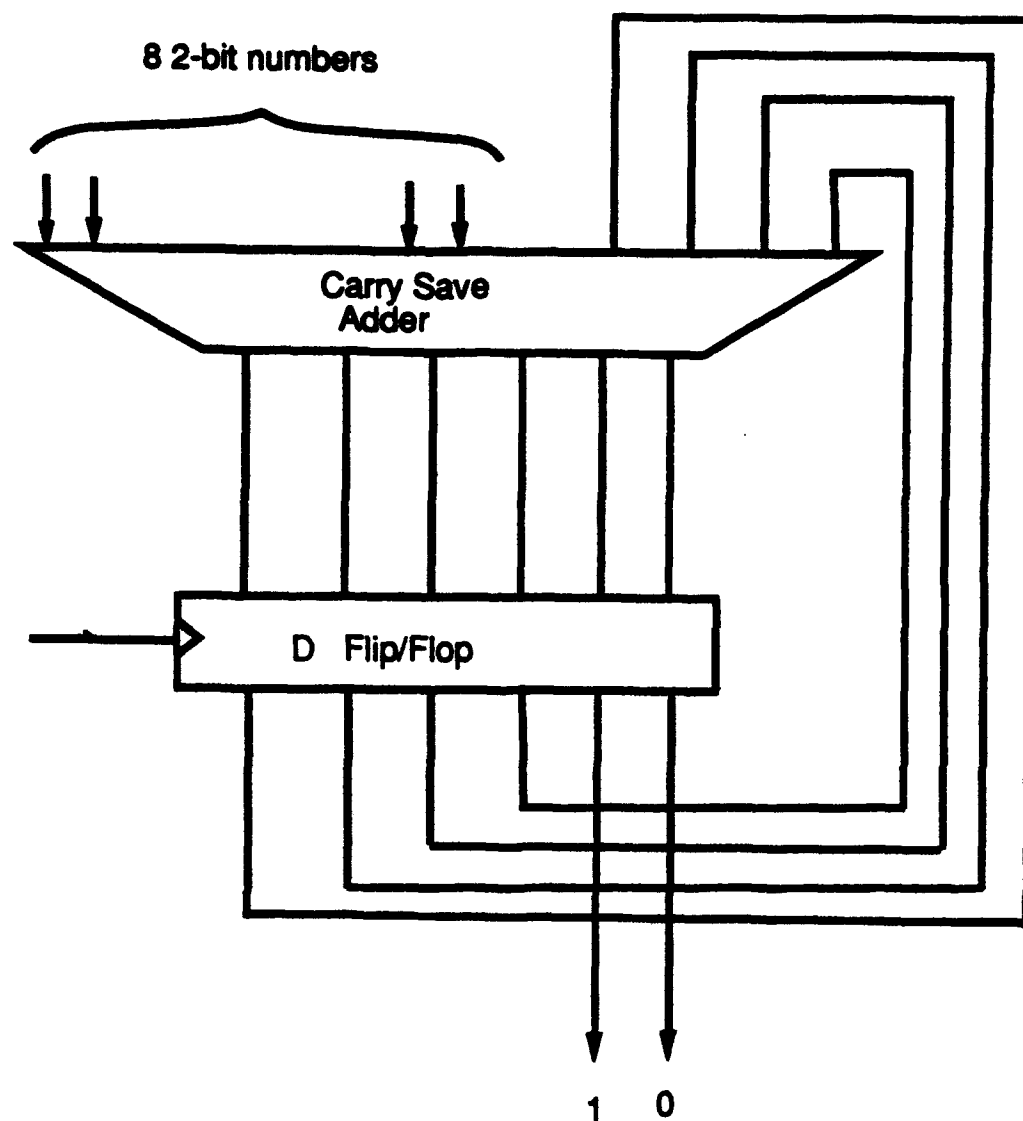


Figure 5.6. Schematic of the daughterboard concentrator chip

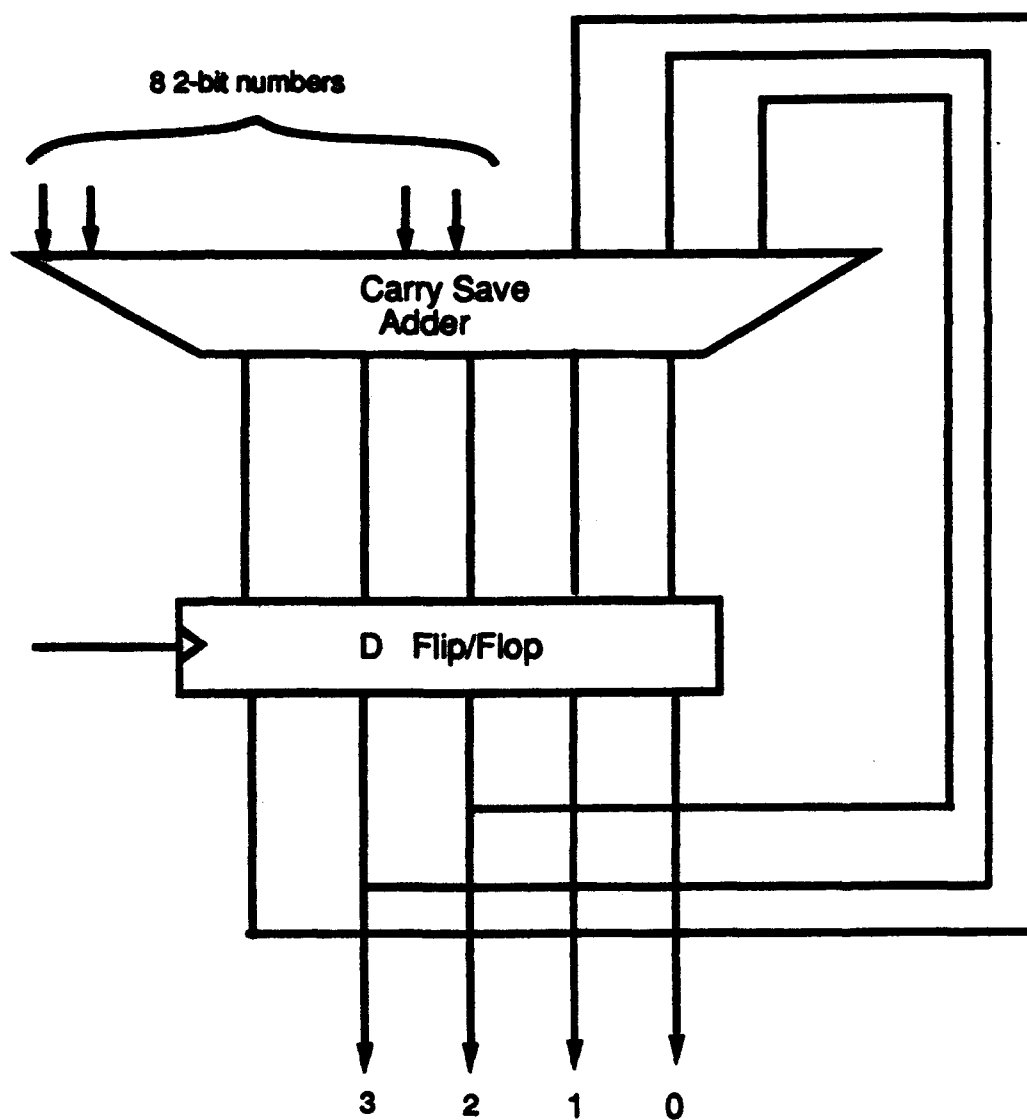


Figure 5.7. Schematic of the motherboard Concentrator Chip

is required to transmit the last bit from the daughterboards to the motherboard concentrator chip through the backplane. Finally, the motherboard concentrator chip introduces an additional 1-cycle delay in producing a 16-bit motherboard level count. Thus the total delay is 8 cycles. The delay in the new concentrator chip is low enough to allow us to clock the count circuitry at twice the CAAPP instruction rate. Thus in terms of the CAAPP instruction cycle, the delay in the count_responders operation for the *IUA prototype II* is 4 cycles. It should be noted that if the same scheme is followed for an IUA system with 1M CAAPP PEs, the delay in counting responders will be only 6 CAAPP instruction cycles which still compares favorably with a uniprocessor test-and-branch instruction.

5.2 Data dependent synchronous communication

In this section we consider centrally controlled networks to support data dependent synchronous communication that can be based upon the networks discussed in the previous chapter. That is, these new networks will be extensions of the earlier networks and will thus include their functionality as well. The processing that takes place in a system is comprised of alternating steps of computation and communication. Such processing is also called *Staged Computation*. In the following subsections, we discuss two of these networks: Crossbar and 3-stage Clos.

5.2.1 Crossbar network

In the previous chapter we showed the feasibility of constructing crossbar networks for a moderate number of processors. We will show in this section that in addition to providing low latency and the highest possible *common access throughput*, a crossbar also has simple network setup and reswitching schemes, and will also discuss two crossbar network designs. The designs differ in the way they handle contention, and in the way they implement some of the routing functions in software vs hardware.

Network architecture

Figure 5.8 is a block diagram of the first of the two designs for building a crossbar network, whose links are computed on-line, using PARCOS I chips. It is comprised of three

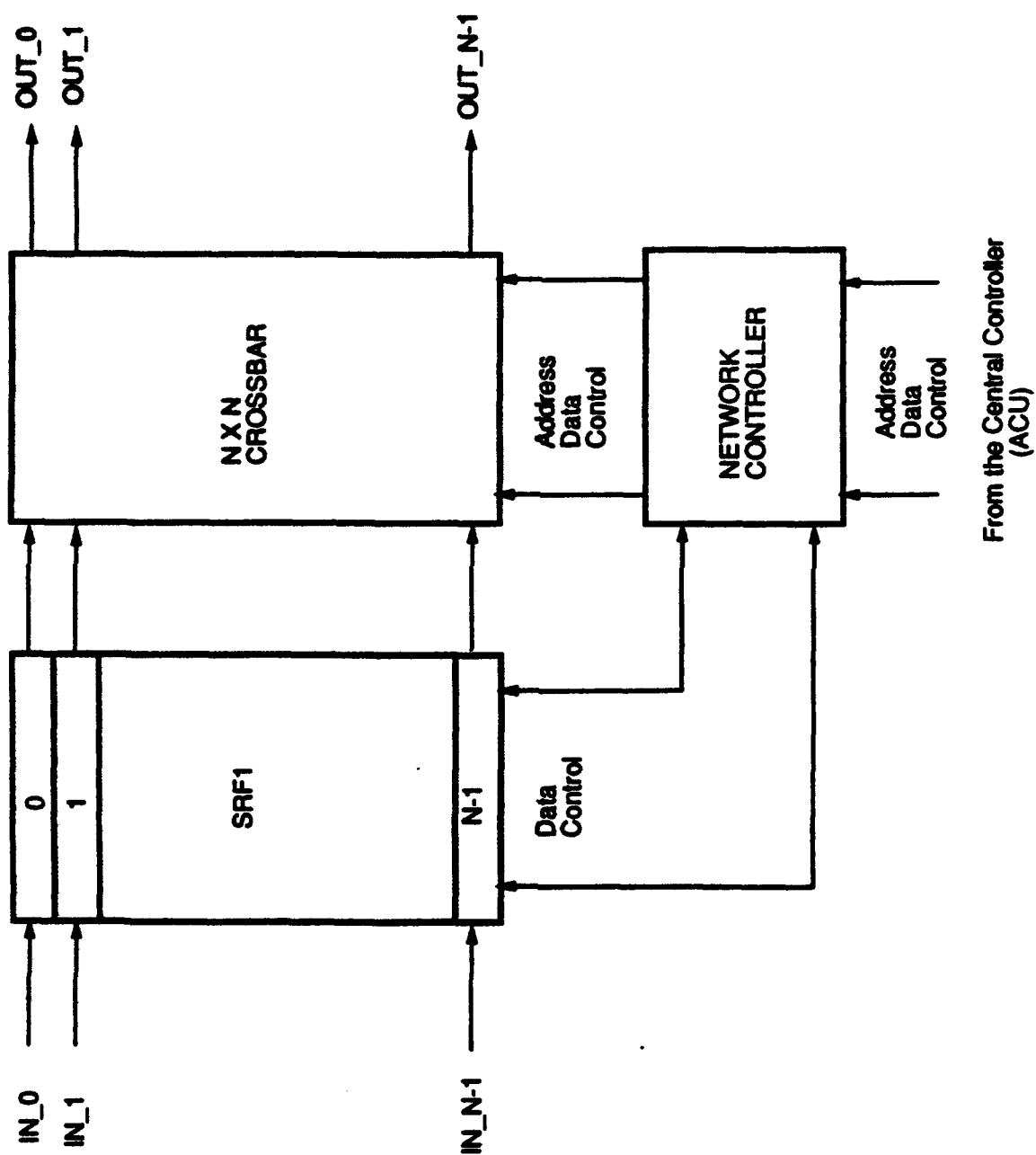


Figure 5.8. First synchronous crossbar network

basic building blocks. The first block is the $N \times N$ crossbar, built with one or more PARCOS I chips as in the previous chapter. Second, a network controller is interposed between the Array Control Unit (ACU) and the crossbar. As will be seen below, the network controller performs extremely simple operations and can be operated at up to ten times the clock rate of the ACU, allowing the network to be set up much faster than if the ACU were to perform the same operations. Further, the ACU is freed to perform other useful functions concurrently while the network is being set up. The third block is a shift register file, SRF1. All of the processors can simultaneously write a destination port number into SRF1 in a serial manner (bit serially if the processor ports are bit-serial. If the processor ports are wider than one bit, that can also be accommodated.) and the network controller can access this file, one word at a time, as a shift register or FIFO.

Figure 5.9 is a block diagram of the second crossbar design. It is similar to the first except that a second shift register file, SRF2, has been added on the output side of the $N \times N$ crossbar. The purpose of SRF2 will be explained shortly.

Network setup and re-switching

Both of these crossbar designs are restricted to synchronous routing. Therefore, all of the processors have to communicate simultaneously. Establishing a communication pattern in the networks of figures 5.8 and 5.9 requires multiple steps. We first describe the network setup and re-switching schemes for the network of figure 5.8.

First, the ACU interrupts all of the processors in the system and directs them to write the addresses of their respective destination processors into SRF1. Keep in mind that the entire system operates with a centralized clock. Writing into SRF1 will take $\log_2(N)$ cycles, where N is the number of processors in the system. The time to interrupt the processors varies with the processor design. For TMS320C30s (the processors used in IUA GEN II), it takes 4 cycles. Thus the total time for the first step is

$$\log_2(N) + 4 \text{ Cycles}$$

In the second step the ACU issues an instruction to the network controller to select a specific control word in the row select registers (RSR) of the PARCOS I chips and start building a connection pattern. The network controller then writes this word address in the RSR of all of the PARCOS I chips to select the desired row. Next it shifts out N words from SRF1, one word at a time, using a counter that it presets to $N-1$, and decrementing

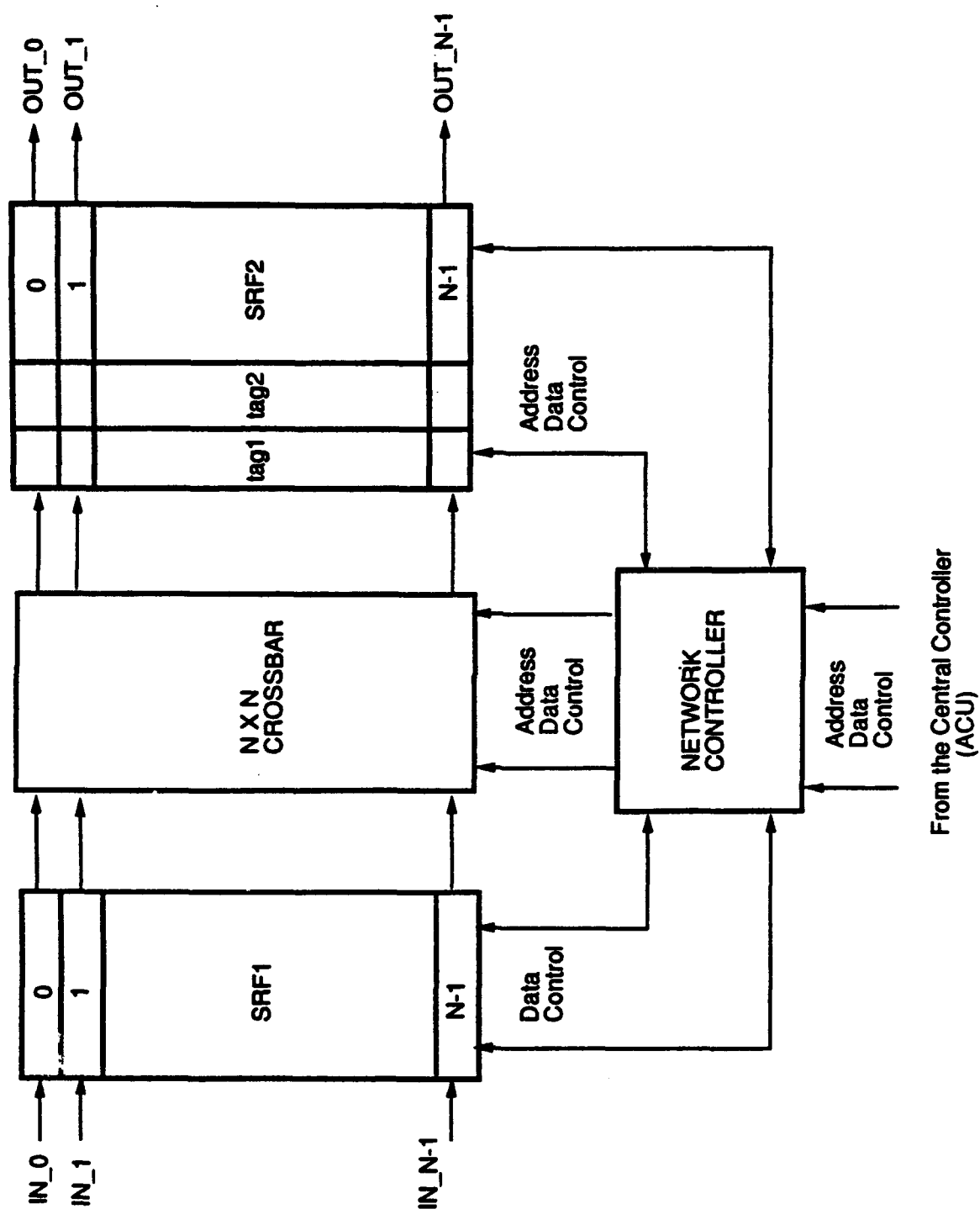


Figure 5.9. Second synchronous crossbar network

for each shift. The counter value corresponds to data (i.e. input port number) for the selected control word in the $N \times N$ crossbar and the associated value shifted out of SRF1 corresponds to the address (i.e. the output port number) for the crossbar. The SRF1 is designed in such a manner that as the last word is shifted out, the connections behind it are shorted across SRF1 (i.e. input IN_x is directly connected to the crossbar rather than through the $\log_2(N)$ stages of the shift register). Thus, after a communication pattern is established, the SRF1 is logically eliminated from the communication path. SRF1 can be easily designed to incorporate this feature. Or alternately, this function can be performed by the network controller such that it directs all inputs to SRF1 to bypass their respective $\log_2(N)$ bit shifters. After N shifts out of SRF1 and N writes into the $N \times N$ crossbar memory, a connection pattern is established in the selected control word. Next, the network controller issues an instruction to load this control word into the PARCOS I control pattern register (CPR), thus establishing the communication paths. If multiple processors request to communicate with the same processor, the last request overwrites previous requests. Retries must be handled in software with this design (The second crossbar design partially alleviates this problem).

The second step of routing thus takes 2 cycles per link ((1) Shift a word out of SRF1 and (2) Write it in the $N \times N$ crossbar). These two steps can be pipelined to reduce the setup time. Thus the total time for this step is

$$1 + N + 1 + 1 = N + 3 \text{ Cycles}$$

The two additional cycles in the total account for selecting a word number to be filled and loading the filled control word into the CPR. The total time taken for establishing a communication pattern is therefore,

$$N + 3 + \log_2 N + 4 = N + \log_2 N + 7 \text{ Cycles}$$

Thus, for a system with 64 processors, 77 cycles would be needed, and for a 4096 processor system, 4115 cycles would be needed. It should be noted, however, that during the second ($\theta(N)$) step the ACU can be independently executing other operations. This is especially useful in the IUA, where the ACU performs other functions such as issuing instructions to the low-level processors.

The crossbar design in figure 5.9 addresses some of the shortcomings of the first design. The first step in setting up the second network is identical to the first design and thus takes

$$\log_2(N) + 4 \text{ Cycles}$$

The second step, is considerably different from the first design, in part because shift register file SRF2 differs from SRF1 in two respects: The network controller can read or write into SRF2 like a standard RAM, and every word of SRF2 has two additional tag bits that are reset to 0 in the first step of setting up the network. When the network controller shifts a word out of SRF1, it uses the contents of the word as an address into SRF2 to fetch the first tag bit (tag_1). If the fetched tag is 0, it indicates that the destination port is free, and the network controller uses the contents of the corresponding SRF1 word as the address for PARCOS I and the current counter value as the data (input port number) to set up the link in the $N \times N$ crossbar. The network controller simultaneously uses the same address and data to write into SRF2 word and set tag_1 to 1. On the other hand, if the destination port is busy (tag_1 is 1), the network controller uses the counter value as the address into SRF2 and sets the second tag bit, tag_2 , to 1.

Thus the second step of setting up the network takes 3 cycles per link ((1) Shift a word out of SRF1, (2) Fetch tag_1 , and (3) Write in crossbar and SRF2.). These three steps can be pipelined to reduce the setup time. The total time for this step is

$$1 + N + 2 + 1 = N + 4 \text{ Cycles}$$

Two of the extra cycles account for selecting a word in the RSR of PARCOS I and loading a control word into the CPR to enable the communication pattern.

In the third step, the ACU instructs the processors to read their corresponding words and tag bits from SRF2. From this information, each processor knows (1) if its request to connect to a destination processor was fulfilled or not, and (2) if it is going to receive a message and from which processor. At this point the feedback concentrator mechanism can be used in two ways: (1) By using the Some/None mechanism the ACU can determine if the communication requests of all processors have been met or not. If not, the ACU can instruct the processors and network controller to create another pattern for the remaining processors. In general, if the communication fan-in (i.e. P is the maximum of the number of messages sent to a common processor) is P , P communication patterns must be created to satisfy all of the communication requests. (2) The Count_responder mechanism can be used to determine the degree of communication contention in the system. This information can be used to guide process migration in instrumentation of system performance.

Notice that by suitable modification the network controller can incorporate all of the functions performed by the feedback concentrator. In our case there is no need for this because the feedback concentrator already exists in the IUA to perform other functions.

The third step of network setup also takes $\log_2(N) + 4$ cycles, so the total cost of network

setup (i.e. filling one control word) is

$$N + 2 \times \log_2(N) + 12 \text{ Cycles}$$

For a system with 64 processors, the total time is thus 88 cycles; with 4096 processors, it is 4132 cycles. Thus for a 64-processor system, the second design takes 15% longer to set up a pattern, but detection and resolution of communication contention is considerably simplified.

5.2.2 Non-blocking network

As discussed earlier, there is a practical limit to the size of crossbar network that can be built from smaller crossbars. In order to build larger connection networks we consider the non-blocking 3-stage Clos network in this section. We are considering only 3-stage Clos networks for three reasons. First, we want to minimize the network latency. Second the control algorithms for 5 or higher stage Clos or Benes networks are considerably more time consuming than for the 3-stage networks, which makes them impractical. Third, as discussed in the preceding chapter, we contend that the size of network that requires, for example a 5-stage Clos design, is impractical to construct in the near future.

An additional note is in order. Within the framework of this subsection, where all processors request to communicate simultaneously, a Benes network will suffice. However, we will not examine the Benes network for the following reasons. First, in a following section dealing with data dependent asynchronous routing, a strictly non-blocking network is required. Second, as discussed in the previous chapter, the pinout on the circuit board and chips are the real impediments to building larger networks and the Benes network is no better than the Clos network in this regard. Third, the Benes network utilizes a more complex and time consuming control algorithm than a Clos network.

Network architecture

Figure 5.10 is a block diagram of the non-blocking network design using PARCOS I chips. It is similar to the crossbar design in figure 5.8 and is comprised of three building blocks. The $N \times N$ Clos design was discussed in the preceding chapter, and contains 3 stages of PARCOS I chips. Each stage is comprised of $m \times m$ PARCOS I switches. Only half of the

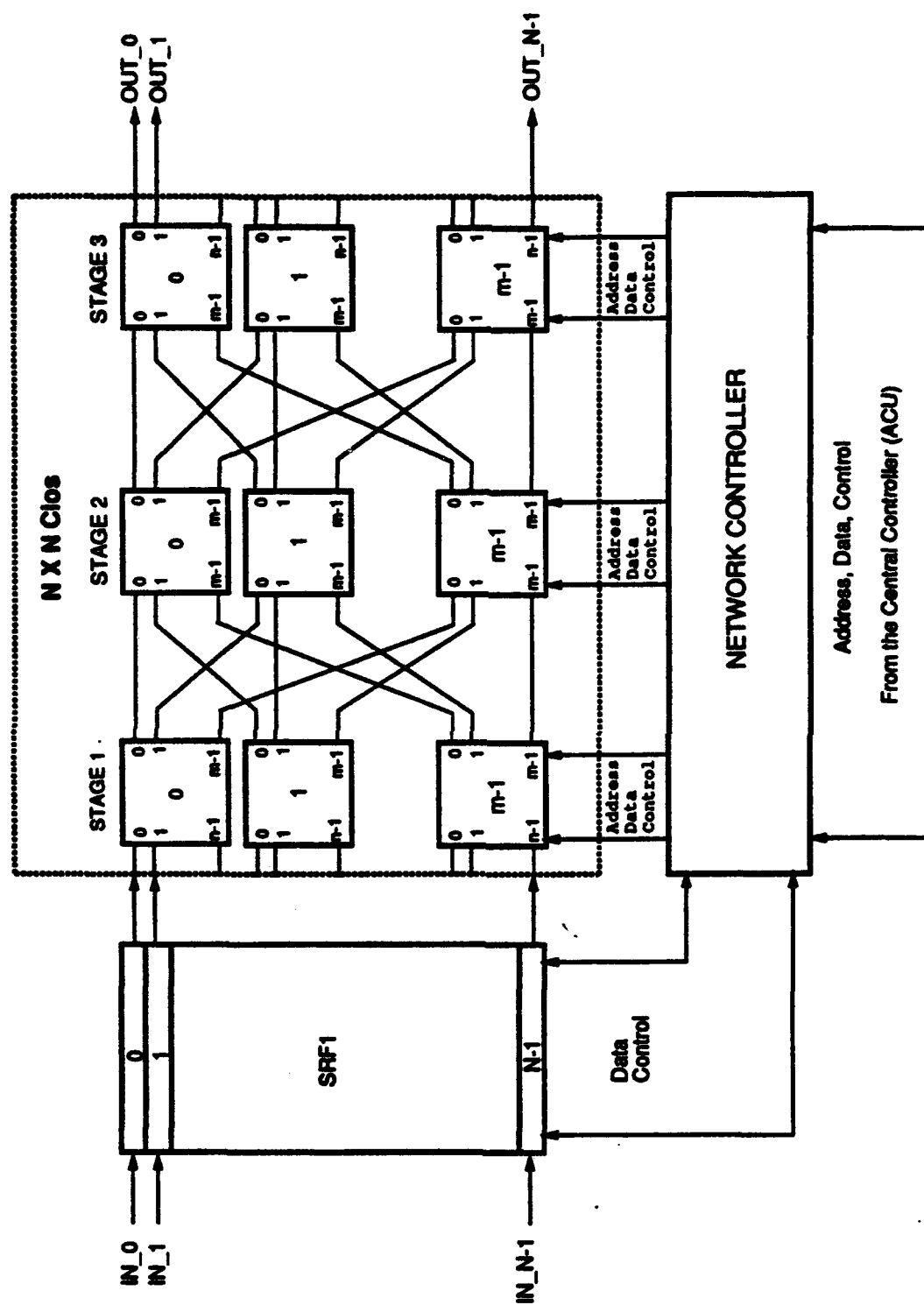


Figure 5.10. Synchronous non-blocking network

m inputs are used in each chip in Stage-1. Similarly only half of the m outputs are used in each chip in Stage-3. With $3 \times m$ PARCOS I chips one can build an $m^2/2$ -input, $m^2/2$ -output 3-stage non-blocking Clos network. For $m = 32$, a 512×512 Clos network can be built with 96 copies of the 32×32 PARCOS I chip. The shift register file SRF1 is similar to that of figure 5.8. All of the processors can simultaneously write a destination port number into SRF1 in a bit serial manner and the network controller can access this file, one word at a time, as a FIFO. A network controller is interposed between the ACU and the $N \times N$ Clos network to control it under the supervision of the ACU. The network controller of figure 5.10 is considerably different from the one in figure 5.8, and will be discussed below.

Network setup and re-switching

Establishing a communication pattern requires the following steps, which can be pipelined to reduce the total number of cycles needed to establish a communication pattern in the network.

The first step in establishing a communication pattern is identical to the crossbar network and takes

$$\log_2 N + 4 \text{ Cycles}$$

The second step is considerably different from the crossbar network. First, the network controller selects a specific control word in all of the PARCOS I chips of the $N \times N$ Clos. Next, it shifts out N words from SRF1, one word at a time. As before, the network controller has a counter that it initializes to $N-1$ and decrements after each shift. The counter value corresponds to an input port number in the network and the associated value shifted out of SRF1 corresponds to an output port number of the network.

As in the crossbar networks of figure 5.8 and 5.9, two schemes can be used to handle contention. In the first scheme, the network controller maintains an internal array that is initially reset. As the connections are established, the network controller fills in values corresponding to the input-output mapping. Before the network controller writes an entry in this array it checks whether the requested output port is available. If the output port is not available, it simply ignores the request and goes back to the first step¹. This scheme takes one additional cycle for every link. The second scheme is identical to that of figure 5.9. We will not explain these two schemes further because their derivation is obvious from

¹Notice that this is different from the crossbar where the last request overwrites all previous requests

the crossbar design. However, the programming of a link from an input to an output is considerably different from the crossbar design.

Programming one link in the $N \times N$ Clos network involves writing one data value in a PARCOS I chip in each of the three stages of the network. All three data values can be written simultaneously. The three addresses and the data values are derived as follows. Suppose the input and the output ports are numbered $0, 1, 2, \dots, N - 1$, and chips in each of the three stages are numbered from 0 to $m - 1$. Inputs $0, 1, \dots, n - 1$ are in the chip number 0 in Stage-1. Inputs $n, n + 1, \dots, 2n - 1$ are in chip number 1 in Stage-1 and so on. Similarly outputs $0, 1, \dots, n - 1$ are from chip number 0 in Stage-3. Outputs $n, n + 1, \dots, 2n - 1$ are from chip number 1 in Stage-3 and so on. For a specific link to be established, let the input port number be X ($0 \leq X \leq N - 1$) and the output port number be Z ($0 \leq Z \leq N - 1$). Suppose that input X is on the chip numbered ISS (Input Stage Switch. Switch is a common terminology used in the field of these networks rather than chip. $0 \leq ISS \leq m - 1$) in Stage-1. The output Z is on the chip numbered OSS (Output Stage Switch, $0 \leq OSS \leq m - 1$) in Stage-3. The task of establishing a communication link from input X to output Z involves selecting a chip MSS (Middle Stage Switch) in Stage-2 through which neither any other input from ISS is connected to any output, nor any output in OSS is connected to any input. It was shown previously that in a non-blocking Clos network, at least one such MSS always exists. Suppose the input X on ISS is numbered x_i ($0 \leq x_i \leq n - 1$) and the output Z on OSS is numbered z_o ($0 \leq z_o \leq n - 1$). Further let us assume that when the link from input X to output Z has been established, the input x_i is connected to output x_o ($0 \leq x_o \leq m - 1$) in ISS , which is connected to input y_i ($0 \leq y_i \leq m - 1$) in MSS , which is connected to output y_o ($0 \leq y_o \leq m - 1$) in MSS , which is connected to input z_i ($0 \leq z_i \leq m - 1$) in OSS , which is connected to output z_o in OSS (which is output Z).

We know that

$$ISS = X \text{ div}(n) \quad (5.1)$$

and

$$OSS = Z \text{ div}(n) \quad (5.2)$$

Notice that we are assuming that n and m are powers of 2. ISS and OSS are simply the $\log_2(m)$ most significant bits of X and Z respectively.

$$x_i = X - n \times ISS \quad (5.3)$$

and

$$z_o = Z - n \times OSS \quad (5.4)$$

Thus z_i and z_o are the $\log_2(n)$ least significant bits of X and Z respectively.

From the construction of the 3-stage Clos network we know that the m inputs to OSS in Stage-3 are from outputs numbered OSS ($0 \leq OSS \leq m - 1$) from each chip in Stage-2. Therefore

$$y_o = OSS \quad (5.5)$$

and we know that the m inputs to MSS in Stage-2 are from outputs numbered MSS ($0 \leq MSS \leq m - 1$) from each chip in Stage-1. Therefore

$$z_o = MSS \quad (5.6)$$

Similarly output from ISS are connected to inputs numbered ISS one in each chip in Stage-2. Therefore

$$y_i = ISS \quad (5.7)$$

Similarly

$$z_i = MSS \quad (5.8)$$

The only unknown quantity on the right hand side of equations 5.1 through 5.8 is MSS , which is determined with special hardware in the network controller. A block diagram of that hardware is shown in figure 5.11. There are two major blocks MEM1 and MEM2, each of which is an $m \times m$ 1-bit memory array. Before a routing cycle starts, all bits in the two memory arrays are set to 1. In MEM1, columns correspond to a specific output number (y_o) on a chip in Stage-2, whereas a row corresponds to a specific chip in Stage-2. Similarly, in MEM2, columns correspond to a specific input number (y_i) on a chip in Stage-2, whereas a row corresponds to a chip in Stage-2.

The objective in finding MSS is to find a row that has a 1 in column y_o in MEM1 and column y_i in MEM2. This is done as follows: The OSS derived in the first step of the routing algorithm is used to enable column y_o in the MEM1 array. At the same time, all rows are

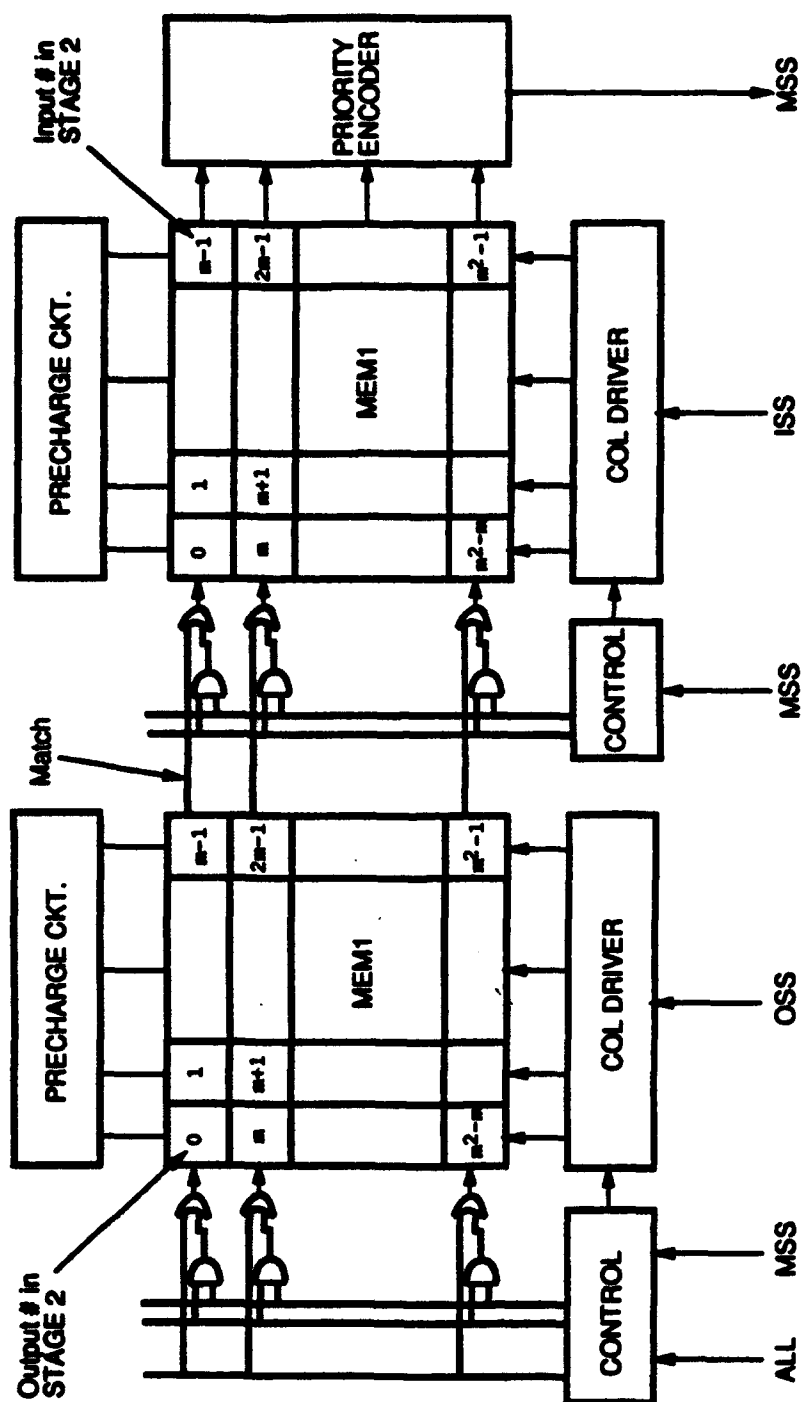


Figure 5.11. Hardware for finding MSS

enabled in the array by asserting the ALL signal². Corresponding to each row in MEM1 there is a Match output on the right hand side of the array. This output is set to 1 if there is a 1 in the bit corresponding to column *OSS* in a given row. At any time, more than one match line may be high. The *ISS* derived in the first step of the routing algorithm is then used to enable column *y*_i in the MEM2 array. Then, only those rows in MEM2 that had a match in MEM1 are enabled, which results in an output of 1 on the corresponding Match lines of MEM2. Each match in turn corresponds to PARCOS I chip in Stage-2 that has a free input as well as a free output, and is thus a candidate for making a connection from *X* to *Z*. Since we need only one one of them, a priority encoder is used to select the first available chip. Notice that obtaining *MSS* by using *ISS* and *OSS* can be done in one cycle.

Once *MSS* is known, the three writes (one in each stage of the $N \times N$ Clos) can be carried out in one additional cycle. In the same cycle, a 0 is written in column *OSS* and row *MSS* of MEM1 as well as in column *ISS* and row *MSS* of MEM2 to indicate that this input-output pair of *MSS* is now occupied. Thus, the second step of routing takes 4 cycles per link ((1) Shift a word out of SRF1, (2) Determine whether the output is free, (3) Find a match in MEM1 and MEM2, and (4) Write as necessary to establish the link). These four steps can be pipelined to reduce the setup time. Thus the second step of the network setup takes a total of

$$N + 5 \text{ Cycles}$$

In the second design, a third step must be added in which the ACU interrupts the processors and directs them to read their corresponding words and the tag bits from SRF2. In that case, the total time cost for network setup is

$$N + 2 \times \log_2 N + 13 + 2 = N + 2 \times \log_2 N + 15 \text{ Cycles}$$

Notice that in our case, the serial setup time of $\theta(N)$ matches with a parallel setup time of $\theta(N)$ in Benes networks as reported in many algorithms [Carpinelli 87; Lee 85; Lee 87]. This is somewhat a case of comparing apples and oranges. We are comparing the serial setup time of a non-blocking Clos network with the parallel setup time of a rearrangeably non-blocking Clos (also called Benes) network. Therefore this shows another disadvantage of Benes network. We are not aware of any other reports on serial or parallel setup times for non-blocking Clos networks.

Next we turn to methods for incorporating data dependent asynchronous routing in these two networks.

²For brevity we are not describing the obvious steps of precharging, reading and writing, etc.

5.3 Data dependent asynchronous communication

In this section we consider extensions to the networks of the previous section to support data dependent asynchronous communication. It is imperative to incorporate this mode of communication if the target system is to efficiently support MIMD-mode computation. The network control is still central and hence serial in nature, but if the amount of reconfiguration in the network is small, the controller can support this mode of communication. Finkel [Finkel 87] shows three case studies: Palindrome-generating puzzle, a general package for recursive tree search, and the Simplex method for linear optimization, where this mode of computation and communication is involved. Also, notice from section 3.4 that a variety of vision tasks are particularly suited for this mode of computation at the ICAP level.

5.3.1 Asynchronous crossbar network

To incorporate asynchronous communication, the circuitry peripheral to the $N \times N$ crossbar must be redesigned. Each input port to the network now must have additional signals for making a connection request, disconnecting a connection, and for reading a Ready signal from the network. Notice that our objective is still to have a design that supports fine-grained, low-latency communication. Therefore, we are not interested in designs where the connect and disconnect requests are encoded in special bit strings, a technique used in long-haul as well as some short-haul computer networks. Such networks are, in general, not concerned with the kinds of communication granularities and latencies required in multiple-processor systems.

Figure 5.12 is a block diagram of the asynchronous crossbar network. The block diagram is very similar to figure 5.8 except that the shift register file, SRF1, has been replaced by an interface unit we call the Front-End. The network controller can read any of the words in the Front-End as if they are stored in a standard RAM. Associated with each port in the Front-End are two flags that are set by the requesting processor through the lines REQ and REQCLR. The network controller can reset any of these flags. There is an additional Ready line associated with every input port, which indicates to the requesting processor when its request (connect or disconnect) is satisfied. A logical OR of the REQ flag of every input to the Front-End is provided via the signal S/N_1 (*Some/None₁*) to the network controller. A logical OR of the REQCLR flag of every input to the Front-End is provided through the signal S/N_2 to the network controller. Additionally, there are two priority encoders in the Front-End

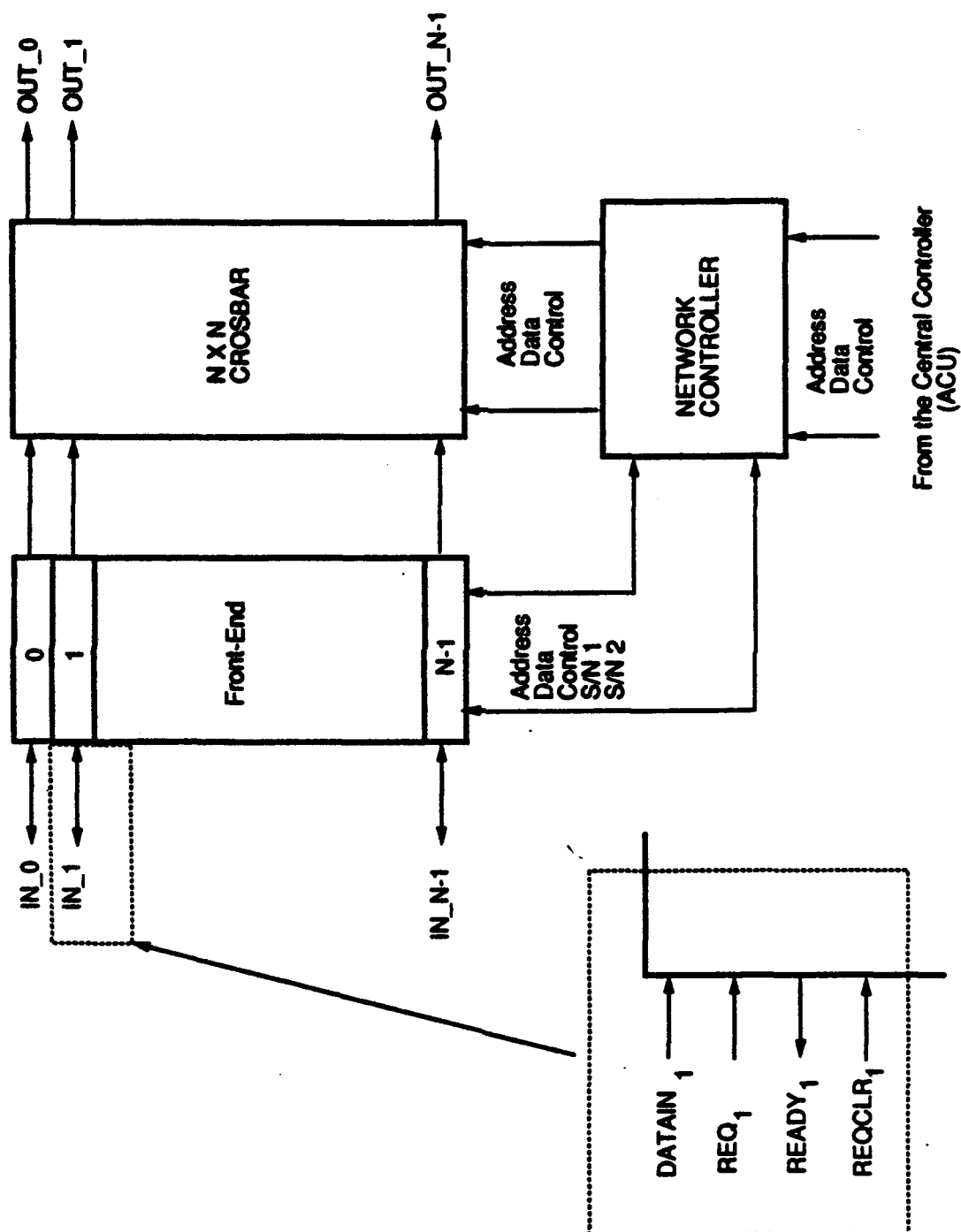


Figure 5.12. Asynchronous Crossbar network

whose inputs are the REQ and REQCLR flags from the input ports. In conjunction with S/N_1 or S/N_2 and one of the priority encoders, the network controller can identify the processor's connect or disconnect requests. Once a request is satisfied by the network controller, it resets the appropriate S/N_1 or S/N_2 bit so that other processors may be serviced.

Recall that the entire system is being operated under a central clock. This restriction can be easily removed from the network by providing an additional clock line in parallel with every input port. A processor makes a request to connect or disconnect a communication link to a destination processor by asserting its REQ or REQCLR line and sending the destination port address through its DATAIN line. All REQ, REQCLR and DATAIN signals must conform to the timing restrictions imposed by the design. The request step takes $\log_2(N)$ cycles. Once a processor makes request, it cannot make another request until the network controller has satisfied it by asserting the READY signal. Once a connect request is satisfied, the link is retained by the sending processor until it requests to disconnect the link. This approach has a potential starvation problem, which can be avoided by having a watchdog timer such that if a processor's request is not satisfied in certain predetermined time, the request is canceled and removed from the Front-End. This is only necessary for the connect request. Notice that by making multiple connect requests, processors can create *multicast* communication patterns. Notice that there is a deadlock situation in this scheme. For example, if processors A and B both want to *multicast* to processors C and D, and each has one of the two lines, they will be deadlocked. This case is treated no differently than where a source processor requests to communicate with a destination processor and the destination processor port is busy. Using a watchdog timer, the source processor must remove its request, if it is not satisfied within a predetermined time. Notice that this can also be incorporated in SRF1 with every input port. In *multicast* communication, software has to handle situations where only partial requests can be satisfied.

As soon as an address for a request is latched in the Front-End, the corresponding REQ or REQCLR flag is set and the respective S/N_1 or S/N_2 line is asserted to the network controller. It is obvious that the network controller should give higher priority to disconnect requests. In conjunction with S/N_1 or S/N_2 and the corresponding priority encoder, the network controller can determine the source and the destination processor number in one cycle.

If it is a disconnect request, the network controller resets the REQCLR flag for the requesting processor, disables the corresponding link in the selected control word in the $N \times N$ crossbar and pulses the READY line to indicate satisfaction of the request to the processor, all in one cycle. The network controller can be designed in such a manner that

it waits until there are no other disconnect requests pending before it writes the modified control word in the control pattern register (CPR), thus establishing the new communication pattern. Notice that reloading the CPR between processor communication has no effect on the state of the links that don't change. Additionally the PARCOS I design can be modified so that in this mode of communication, it is not necessary to explicitly load a control word into the CPR after every link change. The CPR can be operated in a transparent mode such that any change in a control word is automatically reflected in the communication matrix. Alternately, the network controller can reload the CPR with every request. The two schemes have different network re-switching times. Therefore a disconnect request takes either one or two cycles depending upon the scheme used. Notice that the network controller can service a disconnect request every 2 or 3 cycles. From the beginning of a disconnect request, the shortest time to service it, using the scheme where CPR is reloaded after every disconnect request is

$$\log_2(N) + 3 \text{ Cycles}^3$$

Notice that if PARCOS I is designed in such a manner that explicit loading into the CPR is not required after every link change, then the network controller can service multiple disconnect requests at a rate of one every two cycles. If the S/N_2 line is set in the beginning of a cycle, the network controller starts a disconnect cycle. The corresponding priority encoder provides the address of the requesting processor which is used to fetch a word from the Front-End. During next cycle, the link is disabled in the $N \times N$ crossbar, the REQCLR flag is reset and the READY line is asserted.

To service connect requests, the network controller maintains an array with the busy/free status of every output. Alternatively this array could store the network's input-output map (obtained from reading SRF1), to be read by the ACU for load-balancing or instrumentation purposes. In either case, the network controller reads the status of the destination port from the array in one cycle. If the port is free, the controller then resets the REQ flag in the Front-End, updates the status array and programs the link in the crossbar, all in the next cycle. During the third cycle, the controller loads the new control word into the CPR enabling the communication pattern, and signals the READY line to indicate service completion to the requesting processor. As discussed previously (with respect to a disconnect request), this step can be eliminated by redesigning the PARCOS I chip. From the beginning of a connect request, the shortest service time is

³This is latency and not throughput.

$$\log_2 N + 4 \text{ Cycles}$$

With pipelining, the network controller can service connect requests at a rate of one every 4 cycles as long as there is no contention. If the destination port is busy, the controller must disable the corresponding REQ flag from the priority encoder in the Front-End so that other requests can be serviced. The network controller can re-enable all disabled REQ flags in the Front-End after the next (or a cycle of, depending upon the scheme) disconnect request is serviced.

5.3.2 Asynchronous non-blocking network

Figure 5.13 is a block diagram of the non-blocking Clos network to support data dependent asynchronous routing in large systems. The operation of this network is quite similar to the asynchronous crossbar design, and therefore we will only briefly explain the differences.

The Front-End in figure 5.13 is identical to that for the crossbar network and performs the same functions. The method for a processor to make a request is also identical to that for the crossbar. However, unlike the crossbar network, non-blocking Clos networks as well as rearrangeably non-blocking Benes networks *cannot* support arbitrary *multicast* communications. This will be explained in the following section. We assume that it is requesting processor's job to ensure that it does not request simultaneous connections to multiple processors. As before, the network controller can determine the source and destination ports in one cycle.

The network controller in this design maintains an N -row table corresponding to every output of the network. Each row entry is comprised of a tag bit that indicates whether the port is free and an integer entry MSS , indicating the the number of the Stage-2 chip being used for a connection to this output if it is busy. Thus, an additional cycle is required to determine whether a requested output port is free and to read the MSS number for a disconnect request.

For a disconnect request, the network controller resets the REQCLR flag for the requesting processor, disables the corresponding link in the selected control word in the $N \times N$ Clos network, writes 1 at appropriate locations in MEM1 and MEM2, resets the busy flag in its internal N -row table, and signals the READY line to indicate service completion to the

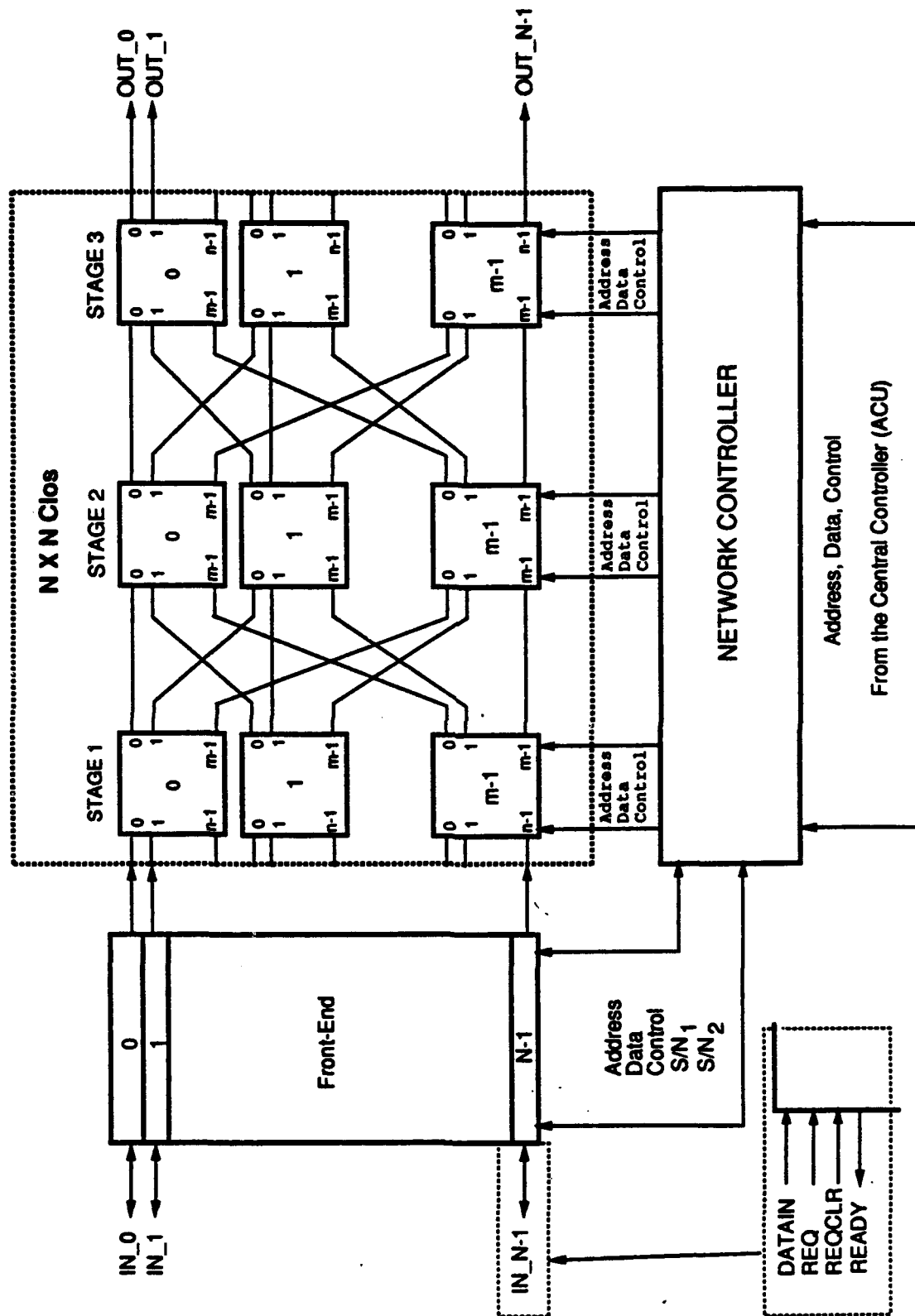


Figure 5.13. Asynchronous non-blocking network

requesting processor, all in one cycle.

If the network controller reloads the CPR after every disconnect request, then the shortest time to service a disconnect request is

$$\log_2 N + 4 \text{ Cycles}$$

For a connect request where the network controller has determined that the requested output port is free, the same steps are followed as in the case of the synchronous non-blocking network, to establish the link, except for the difference in the fourth cycle, and an additional fifth cycle. In the fourth cycle, in addition to writing in the Clos network, the network controller resets the REQ flag for the requesting processor as well as updates its internal N-row table. The fifth (additional) step is to reload the CPR and pulse the READY line for the requesting processor. From the beginning of a connect request the shortest time to service is

$$\log_2 N + 5 \text{ Cycles}$$

If, however, the requested destination port is busy, the network controller follows an approach identical to that of the asynchronous crossbar network.

Next we will briefly discuss *multicast* communication in non-blocking Clos and rearrangeably non-blocking Benes networks.

5.4 Multicast communication

Broadcast communication can be a useful capability in multiple-processor systems. For example, multiplication of two $N \times N$ matrices on an N -processor system requires that the contents of one of the matrices be broadcast to all processors.

One straightforward way of broadcasting in point to point communication topologies is by message replication at intermediate nodes. For bounded degree networks, this operation takes $\theta(N^{1/2})$ steps and in unbounded degree networks it takes $\theta(\log N)$ steps, assuming a single source and N destinations. In disjoint subset broadcasts (also called *multicast*), these times can be much longer depending upon the network architecture and the routing algorithm.

In multistage networks, *multicast* communication is possible if the individual switches support multicast communication. An extensive amount of work has been done in this area.

Many networks, based on 2×2 switches, have been proposed that allow N^N mappings of their inputs onto their outputs. A survey can be found in [Thompson 78].

We discuss two cases of supporting *multicast* communication in 3-stage Clos and 3-stage Benes networks: Asynchronous communication, and Synchronous communication.

5.4.1 Asynchronous communication

It is not possible to support multicast communication in asynchronous 3-stage Clos as well as asynchronous 3-stage Benes networks, which is easily established by a counter-example given in figure 5.14. A multicast from input 0 to outputs 0,2,4,and 6 is shown. If the scheme in figure 5.14 is followed for creating this subset first, input 1 cannot be connected to any output. Masson and Jordan [Masson 71; Masson 72] have shown a generalized 3-stage network design with X inputs and Y outputs (X is not necessarily equal to Y) that supports multicast communication. In the case where $X = Y$, their theorem can be stated as

Theorem 1: Let $N(n, m, r)$ be a 3 stage network as shown in figure 4.6, where the first stage in the network comprises r switches that implement an $n \times m$ crossbar function, the second stage comprises m switches that implement an $r \times r$ crossbar function, and the third stage comprises r switches implementing an $m \times n$ crossbar function. This network is strictly non-blocking in supporting N^N mappings (such networks are also called generalized connection networks. $N = n \times r$) if

$$m \geq n \times r + n - 1$$

This is for a strictly non-blocking generalized 3-stage connection network (as encountered in asynchronous routing). For a rearrangeable generalized 3-stage connection network (as encountered in synchronous networks of generation 1.0 design and in the next subsection) $m \geq n \times r$ is sufficient. Notice that even in this case the 3-stage Clos as well as the 3-stage Benes network cannot support the N^N mappings property (Because a 3-stage Clos network has $m = r = 2 \times n$).

We will not go into the details of the proof of these results, as they can be found in the cited papers [Masson 71; Masson 72]. In our case, we have $m = r = 2n$. Therefore, it is obvious that neither the 3-stage Clos nor the 3-stage Benes network can support asynchronous multicast communication in circuit switched configuration.

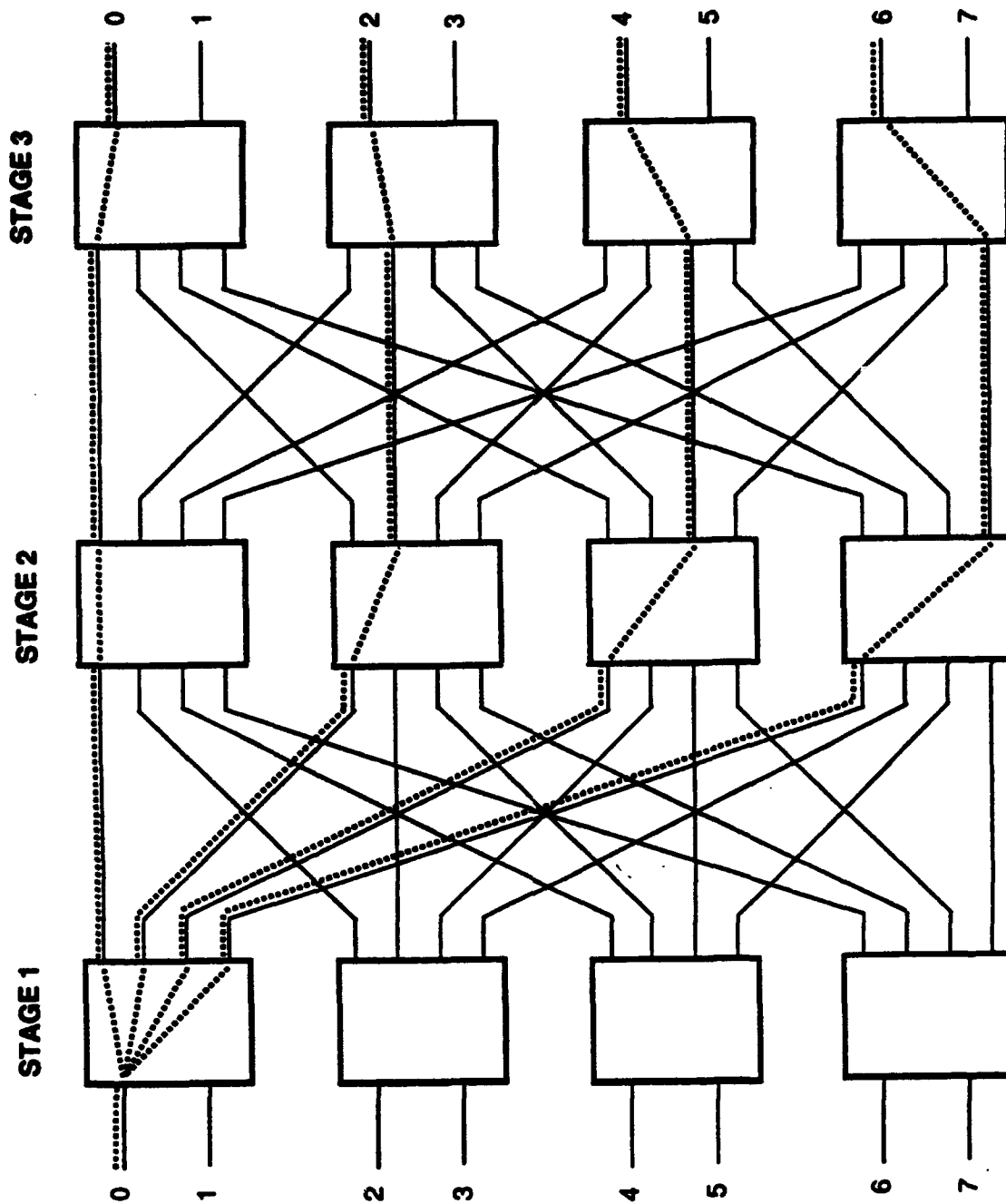


Figure 5.14. Counter-example

5.4.2 Synchronous communication

From the above theorem, it is clear that even synchronous 3-stage Clos as well as synchronous 3-stage Benes networks cannot support multicast communication. However, a different approach has been described for supporting multicast communication in synchronous networks built with rectangular ($n \times m$, n and m are not necessarily equal) crossbar switches, by using more than three stages of these switches. It remains to be proven whether four stages of switches are sufficient to support multicast communication. Kumar [Kumar 88] showed how to support multicast communication using two back-to-back 3-stage Benes networks. He further showed that rather than using two physical Benes networks, the same can be achieved by making two passes of the messages through one 3-stage Benes network (with the processors acting as intermediate storage between the two passes). The first pass is comprised of generating replicated copies of the input requests corresponding to the cardinality of the output set that each input is connected to, in a monotonically increasing ordering of the input numbers (in other words, look at input 0 first. If it is connected to 4 different outputs, generate 4 copies of this message from input 0 to outputs 0-3 in the first pass. Then, say input 1 is connected to 7 different outputs. Generate 7 copies of the messages from input 1 to outputs 4-10 in the first pass, and so on). The second pass is comprised of permuting these copies to the appropriate outputs. Thompson's generalized connection network is designed along similar lines [Thompson 78].

Many serial and parallel routing schemes exist for the control of synchronous Benes networks [Andersen 77; Carpinelli 87; Chow 80; Lee 87; Nassimi 82; Opferman 70; Ramanujam 73; Tsao-Wu 74; Waksman 78; Wilkerson 87]. Some of these algorithms assume a specific construction of the Benes network. The optimum worst-case serial time is $\theta(N^2)$ and the optimum worst-case parallel time is $\theta(N)$. The replication algorithm of Kumar's [Kumar 88] two-stage scheme has the same complexity as the permutation algorithm. Notice that all of these algorithms assume apriori communication patterns.

We can handle synchronous multicast communication more efficiently in our non-blocking network design. Using a serial network controller, we achieve speeds equivalent to many parallel control algorithms for rearrangeable networks. As before, this is somewhat a case of comparing apples and oranges. We are comparing the serial set up time of a Clos network with the parallel setup time of a Benes network. Therefore this shows another disadvantage of the Benes network.

5.5 Analysis and comparison of networks

In this section we analyze and compare the four networks (Synchronous crossbar, synchronous 3-stage Clos, asynchronous crossbar, and asynchronous 3-stage Clos) described earlier in this chapter, in terms of (1) Hardware cost, and (2) Time to set up and reconfigure.

5.5.1 Hardware cost

Here, we are primarily interested in determining the hardware implications of the four designs discussed in this chapter - i.e. the hardware cost involved in building them, in terms of the number of different custom chips and board area. Recall that the hardware costs of the blocks containing crossbar and 3-stage Clos networks were discussed in the previous chapter and do not change in these designs. Of greater interest here are various blocks added to incorporate data-dependent routing.

Crossbar network

In synchronous crossbar design, let us consider the second design in figure 5.9. The network controller is a pipelined design. It can be built with off the shelf parts, but for maximum performance, it will be a custom VLSI chip. The shift register file, SRF1 is similar to *corner turning memories* [Batcher 77; Batcher 80]. On one side, a number of input ports can simultaneously shift an address into SRF1, in a bit-serial manner. On the other side, the network controller can shift these words out, one word at a time. Assuming a limit of 400 pins on current PGAs, it is feasible to implement a 128-input SRF1 on a single custom VLSI chip. From figures 5.8 and 5.9, notice that it is not necessary to route signals to the $N \times N$ crossbar via SRF1. Inputs $IN_0 - IN_{N-1}$ can be routed in parallel to two places: SRF1 and the $N \times N$ crossbar. After the network controller sets up a communication pattern in the crossbar, it can simply disable SRF1. This will allow a single chip design of SRF1 with more than 256 inputs. Let us assume that it is 256. Thus an $N \times N$ crossbar network will require $\lceil \frac{N}{256} \rceil$ custom SRF1 chips. Identical approach can be followed for SRF2. Rather than routing outputs from the $N \times N$ crossbar via SRF2, they can be multiplexed with the outputs from SRF2 under network controller supervision. Using 8 32×32 PARCOS I chips from the previous design, along with one custom VLSI chip each for the network controller,

SRF1, and SRF2, it is feasible to design a 64-input 64-output ICAP communication network for IUA GEN II prototype.

The network controller in the asynchronous crossbar network is again a single VLSI chip. The Front-End design requires 4 pins for every input port. Using a scheme identical to SRF1 (for data pin), the pins required for input to the crossbar from the Front-End can be saved. This will allow a single chip in the Front-End to support 64 inputs. For an $N \times N$ network, $\lceil \frac{N}{64} \rceil$ custom Front-End chips will be required. By using 8 32×32 PARCOS I chips and one custom chip each for the network controller and the Front-End, ICAP communication network for IUA GEN II prototype can be fabricated on a single board.

Clos network

In synchronous Clos network design of figure 5.10, SRF1 is identical to that for a synchronous crossbar network. The network controller in figure 5.10 even though, is considerably different from that in synchronous crossbar, can still be built on a single custom VLSI chip. The hardware for finding MSS as shown in figure 5.11 is implemented inside the network controller. For example, by using 48 128×128 PARCOS I chips, along with 4 SRF1 chips and a network controller chip, a $1K$ -input $1K$ -output Clos network can be built. MEM1 or MEM2 in the network controller for such a network will be 128×8 bits each. Using sizes for these chips from the previous chapter, such a network will require

$$(48 + 4 + 1) \times 16 \times \frac{1}{2} = 424 \text{ Sq. inch}$$

Using 18×12 " boards, this will require

$$\frac{424}{18 \times 12} = 1.96$$

or 2 boards.

In asynchronous Clos network design of figure 5.13, using a scheme for the Front-End similar to that for the asynchronous crossbar, a 64-input Front-End can be implemented on a single chip. Notice that Front-End has to be designed in such a manner that it is possible to extend its size (for S/N_1 , S/N_2 , and priority encoder) by using multiple chips. The network controller design is similar to that for the synchronous Clos network. For example, by using 192 128×128 PARCOS I chips along with 64 Front-End chips, and a network controller, a $4K$ -input $4K$ -output asynchronous Clos network can be built. MEM1 and MEM2 in the

network controller for such a network will be 128×64 bits each. Using the size of these chips from the previous chapter, such a network will require

$$(192 + 64 + 1) \times 16 \times \frac{1}{2} = 2056 \text{ Sq. inch}$$

Using 18×12 " boards, this will require

$$\frac{2056}{18 \times 12} = 9.5$$

or 10 boards.

Next, we turn to the time for setting up and reconfiguring these networks.

5.5.2 Time to set up and reconfigure

Recall that the latency and throughput for the same topologies were discussed in the previous chapter. The extensions proposed in this chapter have no effect on these aspects of performance. Of greater interest here is the time required for setting up a communication link as well as a complete pattern. We chose the worst-case time for the comparison purposes. The processor is assumed to be TMS320C30 (or one that requires 4 cycles for interrupt).

Table 5.1. Number of cycles for routing

Operation	Synch. Crossbar	Synch. Clos	Asynch. Crossbar	Asynch. Clos
One link	$2\log N + 11$	$2\log N + 12$	$\log N + 4$	$\log N + 5$
N links	$N + 2\log N + 12$	$N + 2\log N + 15$	$4N + \log N$	$5N + \log N$

Table 5.1 lists the routing times in terms of network controller cycles. It is obvious from the discussion of the network controller that its design is relatively simple and pipelined. By extrapolation from currently available RISC processors and memory technologies, we assume that the cycle time can be 25nS using a MOS technology and 15nS using a bipolar technology. As an example, a synchronous crossbar of size $1K \times 1K$ can be fully configured in

$$25 \text{ nS} \times [1024 + 2 \times \log(1024) + 12] = 26.4 \mu\text{S}$$

using a MOS technology, or in

$$15 \text{ nS} \times [1024 + 2 \times \log(1024) + 12] = 15.8\mu\text{S}$$

using a bipolar technology.

If we assume the processors operate at 20 MIPS (50nS per instruction), the network setup time is equivalent to 550 instruction times using a MOS technology or 330 instruction times using a bipolar technology.

As another example, a $4K \times 4K$ synchronous Clos network can be configured in

$$25 \text{ nS} \times [4096 + 2 \times \log(4096) + 15] = 103.4\mu\text{S}$$

using a MOS technology, or in

$$15 \text{ nS} \times [4096 + 2 \times \log(4096) + 15] = 62.0\mu\text{S}$$

using a bipolar technology.

For the asynchronous case, the best time to service a connect request in $1K \times 1K$ crossbar is

$$25 \text{ nS} \times [\log(1024) + 4] = 350 \text{ nS}$$

using a MOS technology or is

$$15 \text{ nS} \times [\log(1024) + 4] = 250 \text{ nS}$$

using a bipolar technology. This is on the order of 5 to 7 processor instruction cycles, depending on the technology used in the network. However, in the case where N simultaneous, non-conflicting connect requests are made, it will take

$$25 \text{ nS} \times [4 \times 1024 + \log(1024)] = 102.6\mu\text{S}$$

using a MOS technology or

$$15 \text{ nS} \times [4 \times 1024 + \log(1024)] = 70.6\mu\text{S}$$

using a bipolar technology. These times are on the order of 140 to 200 instruction cycles. Remember that redesigning PARCOS I, so that it doesn't require reloading the CPR after

every link modification, will reduce the asynchronous network routing times by N cycles. This results in a reduction of about 25% in the above two times. Even so, this design is restricted to MIMD systems where network reconfiguration requests do not occur in bursts or are more evenly distributed in time.

5.6 Conclusions

The major conclusion of this chapter is that central control is a viable solution for reasonably large network sizes, in supporting data dependent routing, which is contrary to conventional wisdom.

To arrive at this conclusion, we presented the architecture of centrally controlled crossbar and Clos networks. We further subdivided the problem and showed how synchronous (i.e. all processors make communication request at the same time. This mode is used in SMIMD mode of operation at the ICAP level) as well as asynchronous (i.e. for MIMD mode of computation) data dependent routing can be carried out in these networks. It was shown that by using a special search memory to implement part of the Clos network routing algorithm in hardware, it is possible to carry out incremental routing in constant number of steps. Further, by pipelining this routing algorithm, it was shown that a complete assignment (i.e. setting N links for an $N \times N$ network) can be carried out in $N + \text{Constant}$ number of steps. This is the best serial time known so far. It was shown how multicast communication can be supported in these networks. Finally, we analyzed and compared these networks with respect to their hardware cost and time to setup (incremental as well as complete assignment) and reconfigure. The following is a summary of the goals achieved in thesis so far and the remaining goals.

5.6.1 Goals achieved

The objective of the generation 1.5 design was to address the requirements of a multiple-processor system that is operated in such a manner that the interprocessor communication is data dependent and cannot be known apriori. We showed that under central routing control it is possible to provide such a capability by extending the generation 1.0 network designs. These extensions are low cost solutions that, because of their serial nature, cannot be used for arbitrarily large networks. A network controller was interposed between the

central controller (ACU) and the switching network to carry out network operations under ACU supervision. It was shown how a feedback concentrator mechanism can be used to aid the network controller. The network controller can be operated concurrently with the ACU and thus, in many cases, the network setup time can be overlapped with other ACU activities.

5.6.2 Remaining goals

These designs are not effective in fine-grained communication where the amount of network reconfiguration is high (also called statistical switching). There is considerable debate in the community whether it is justifiable to assume that large-scale multiple-processor systems require support for dense irregular communication. Certainly, this is necessary to support a distributed, shared memory of computation. Thus, we are still interested in exploring ways to further extend these designs to support low latency, parallel self-routing.

CHAPTER 6

GENERATION 2: DISTRIBUTED ROUTING CONTROL

This chapter deals with the last of the requirements of multiple-processor systems: Distributed or parallel control of communication. Our original plan was simply to add this capability to the crossbar and 3-stage non-blocking Clos networks. Recall from section 1.3 that we understood at the beginning of the IUA effort that machine vision is highly dynamic and evolutionary in nature, and the development of parallel architectures for machine vision is a nascent field. Therefore, the requirements of the ICAP communication network outlined in sections 1.3 and 3.5 were as they were understood at the start of the IUA effort and it was known that they would evolve as the project proceeded. During the course of this research new intermediate-level vision requirements evolved along with different IUA design constraints. This provided the opportunity to develop an entirely new network family using a different methodology than that used for the first three stages.

The organization of this chapter is as follows. In the next section we describe the architecture of PARCOS II chip that implements a self-routing 6×6 crossbar network, and addresses the original plan for building larger self-routing networks. Next, we discuss the changes in the architectural requirements of the intermediate level and the new IUA design constraints. We tackle the problem in stages, presenting three designs. Thereafter, we discuss what might be involved in building larger networks with these capabilities, followed by conclusions.

6.1 PARCOS II

In this section we describe the architecture of a self-routing communication network building block chip called PARCOS II, which implements a self-routing 6×6 crossbar switch. The switch is capable of arbitrating between its inputs in unit time in order to route them to their respective outputs. The broader objective is to use either the individual cells from this chip to build large crossbar networks on a chip or to directly replicate this crossbar switch on

a chip to build larger crossbar networks. Once a crossbar switch (using one or more chips) of a specific size is obtained, copies of this switch can be used to build various networks such as Clos, Hypercube etc.

A block diagram of the chip is shown in figure 6.1. It is comprised of a 6×6 array of building block cells. The I/O interface to each processor using a network built with this chip has 5 lines, namely DATAIN, DATAOUT, CHREQ, CHACK and CHREQCLR. Because of constraints on the pin count for the particular (low cost) fabrication run employed, this design uses a central clock. Making the design completely asynchronous, i.e. eliminating the central clock, requires only one additional pin for every I/O port, and can be readily incorporated at a future date.

The function of this switch (we will sometimes interchangeably use the term switch instead of chip, because the ideas discussed here pertain to the design and are not specific to whether the switch is implemented on a single or multiple chips) is to establish connections in parallel between DATAIN lines connected to the requesting processors and DATAOUT lines connected to the destination processors. A processor requests a connection to another processor by asserting its CHREQ signal to PARCOS II and shifting in a 3-bit destination processor address via its DATAIN line. After the destination processor address has been input, the channel request line - CHREQ is pulled low. If the requested output is busy, a 0 is returned on the CHACK line. In the present design, if the requested output is busy, the requesting processor will have to retry by repeating the request. (We are currently working on a design in which a request for a busy output is queued. As soon as the requested output becomes available, the requesting processor is then notified by the CHACK signal and a link is established.) If the requested output is free, a 1 is returned on the CHACK line and the DATAIN line from the requesting processor is connected to the DATAOUT line for the destination processor. After a clock cycle delay, the requesting processor is free to send data. The path from DATAIN to DATAOUT is circuit switched and fully combinational. After all data transmission is over and the source processor no longer requires the connection, it sends a channel request clear signal on its CHREQCLR line, thereby releasing the connection.

A block diagram of an individual cell of PARCOS II is shown in figure 6.2. The first step in establishing a connection between a source and a destination processor is to shift in the address of the destination processor into the 3-bit register under the control of the finite state machine (FSM). The outputs of the FSM are 3 enable signals for the 3-bit register and one enable signal for the decoder. The FSM has four states, encoded by two bits. When the state of the FSM is 00, the outputs EN0, EN1, EN2, and EN3 are 0001. In this state, none of the bits in the 3-bit register are enabled to accept data from DATAIN. At the positive edge

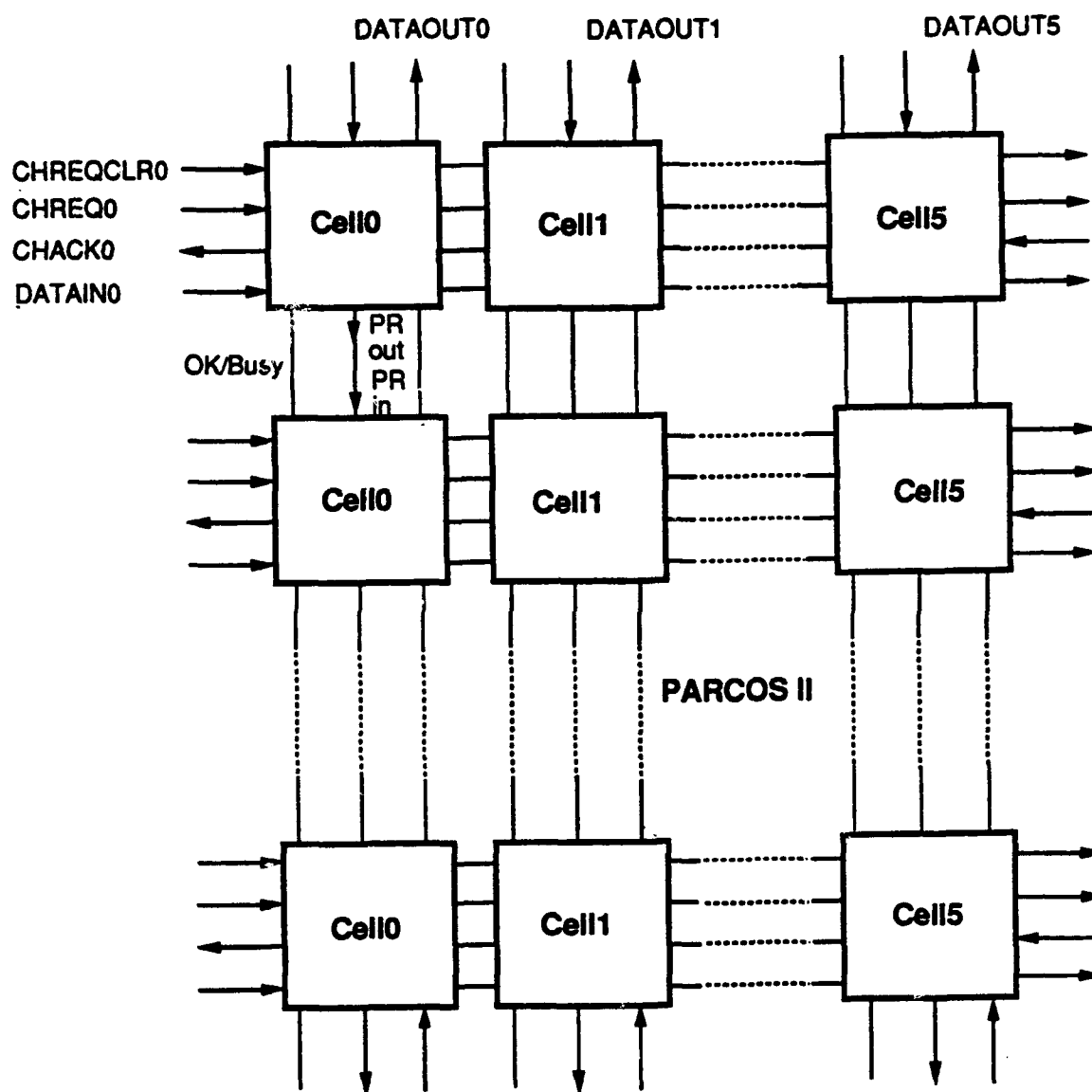


Figure 6.1. Block diagram of PARCOS II

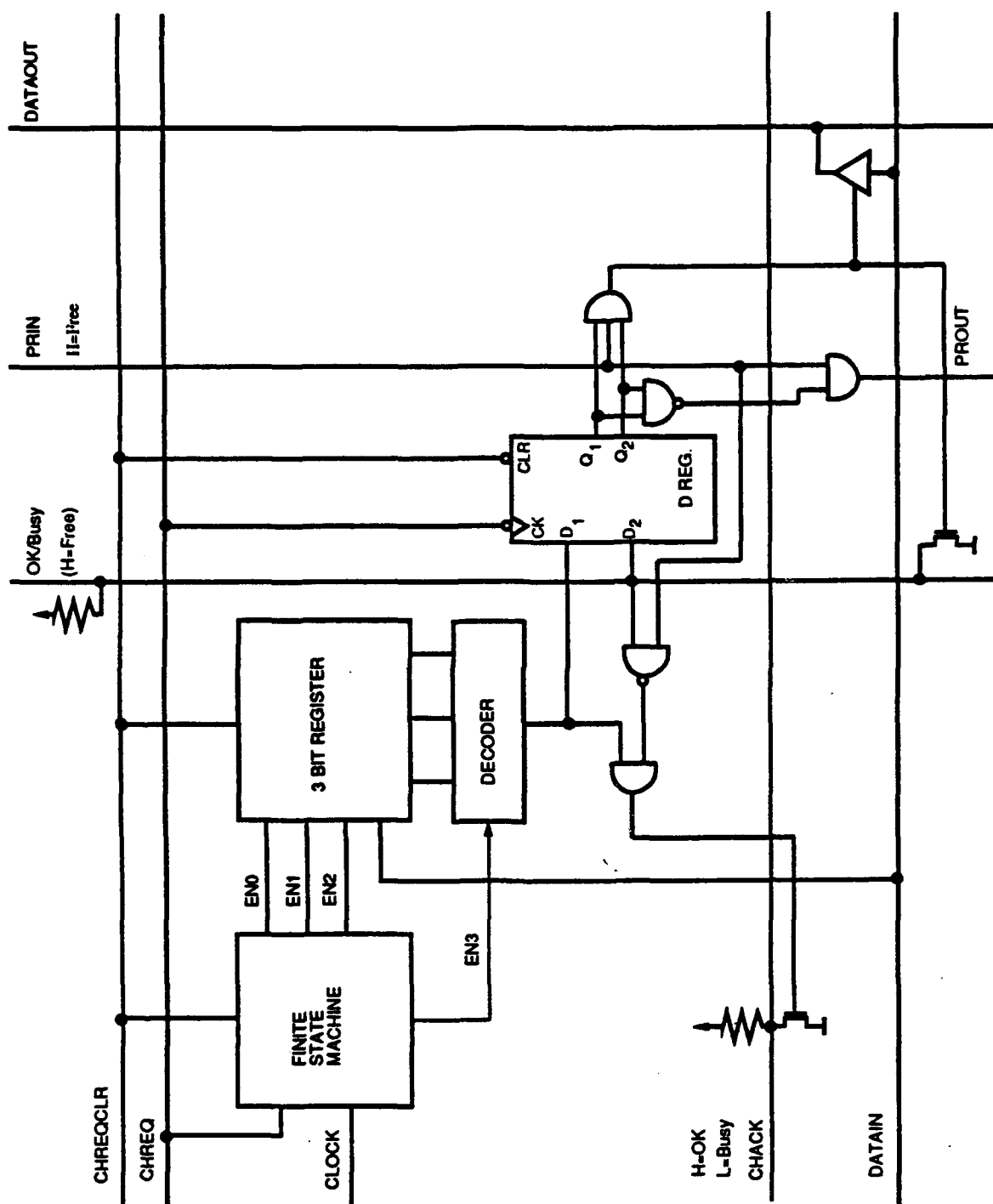


Figure 6.2. Block diagram of one cell

of the CHREQ signal, i.e. when a particular source processor begins a request, the state of the FSM changes to 01. The outputs in this state are 0010. Now the least significant bit (LSB) of the destination address is shifted from DATAIN into the LSB (bit 0) of the 3-bit register. At the next rising edge of the clock, the state changes from 01 to 10. The outputs of the FSM are now 0100 and the second bit of the destination address is shifted into bit 1. At the next rising edge of the clock, the state of the FSM changes to 11 and the outputs change to 1001. The most significant bit (MSB) of the destination address is shifted into bit 2 and simultaneously the DECODER is enabled. At the next clock edge, the state of the FSM becomes 00 and all three of the bits are disabled while the DECODER remains enabled. The state of the FSM remains 00 until the next rising edge of the CHREQ signal, i.e. when the next request is made. Note also that a CHREQCLR signal brings the FSM to state 00 regardless of its current state.

Depending on the destination address that was shifted into the 3-bit register, the output of the appropriate DECODER in the row corresponding to the requesting processor is asserted high during the third cycle of the address input, after which, the CHREQ line is pulled low. At the falling edge of CHREQ, the output of the decoder is latched at the output Q_1 of the D Register. Additionally, the state of the OK/Busy line is latched at output Q_2 of the D Register. The OK/Busy line is used by all of the cells in a column to arbitrate for a common output. At this point, depending upon the status of the OK/Busy and PRIN lines, the connection between DATAIN and DATAOUT may be established. At the same time an acknowledge signal CHACK will be generated for the requesting processor. Note that if more than one processor requests a connection to the same output in the same clock cycle, the OK/Busy line will not be sufficient to arbitrate. The PRIN and PROUT lines provide arbitration between inputs in this situation. As can be seen from figure 6.2, arbitration is priority-based.

The original goal of this design was to be able to queue all unfulfilled requests without requiring processors to retry. The design can be used to achieve this for requests made during the same cycle. If multiple requests are made for a common output during the same cycle, then the outputs Q_1 and Q_2 of the respective cells will be 1. However, the top cell will be connected first and all other cells' PRIN lines will be low. As soon as the first processor is done with its communication and asserts CHREQCLR, its PROUT line goes high and the second requesting processor is connected to this column and a CHACK signal is generated for it. The same process is repeated until the last request is fulfilled. If, during a waiting period, a processor wants to cancel its request, it can do so by asserting its CHREQCLR signal and it will be deleted from this queue. If a request for an output is made while it is busy, the requesting processor's Q_2 will be 0 after the CHREQ signal goes low and thus there is no way to automatically fulfill this request after the existing communication terminates.

This situation can be addressed in a later design.

If the CHACK signal is high, i.e. the channel is free, then data may be transmitted via the DATAIN line. At the end of data transmission, the CHREQCLR signal is asserted (a one cycle duration return-to-1 pulse) to disconnect the link and free it for other processors.

A checkplot of one of the cells is shown in figure 6.3. All 36 cells in PARCOS II are similar to figure 6.3 except that the DECODER and pullup resistors are used only in the left column and top row. This chip was fabricated by the Massachusetts Microelectronics Center (MMC or M²C) using a 2 μ N-well double metal CMOS technology on a 40 pin package. Unfortunately, due to some problems in the standard pad frame library, the chip could not be tested. The simulation results using SPICE indicate that the design can be clocked at 25MHz. This design is extensible to larger crossbar switches, either on a chip or with multiple chips.

This approach to self routing multistage networks was discontinued at this point because of the new architectural requirements and design constraints on the ICAP level of the IUA. We discuss these changes in the next section. However, it should be noted that this design achieves our original goals for communication in the ICAP. In terms of the taxonomy of communication network outlined in Chapter 1, thus far we have achieved all of the checked capabilities in table 6.1 below

Table 6.1. Taxonomy of Communication Networks

Patterns Computed	Central control		Distributed control	
	Synchronous	Asynchronous	Synchronous	Asynchronous
Off-line	✓	-	✓	-
On-line	✓	✓	✓	✓

The designs presented in the remainder of this chapter provide additional capabilities that were not required in the first generation ICAP architecture.

6.2 Changes in the intermediate level

Our original plan in this research was to demonstrate the feasibility of incorporating fast self-routing into the networks outlined in chapters 4 and 5. We discontinued pursuit of this path beyond the design and fabrication of the building block PARCOS II chip for the reasons

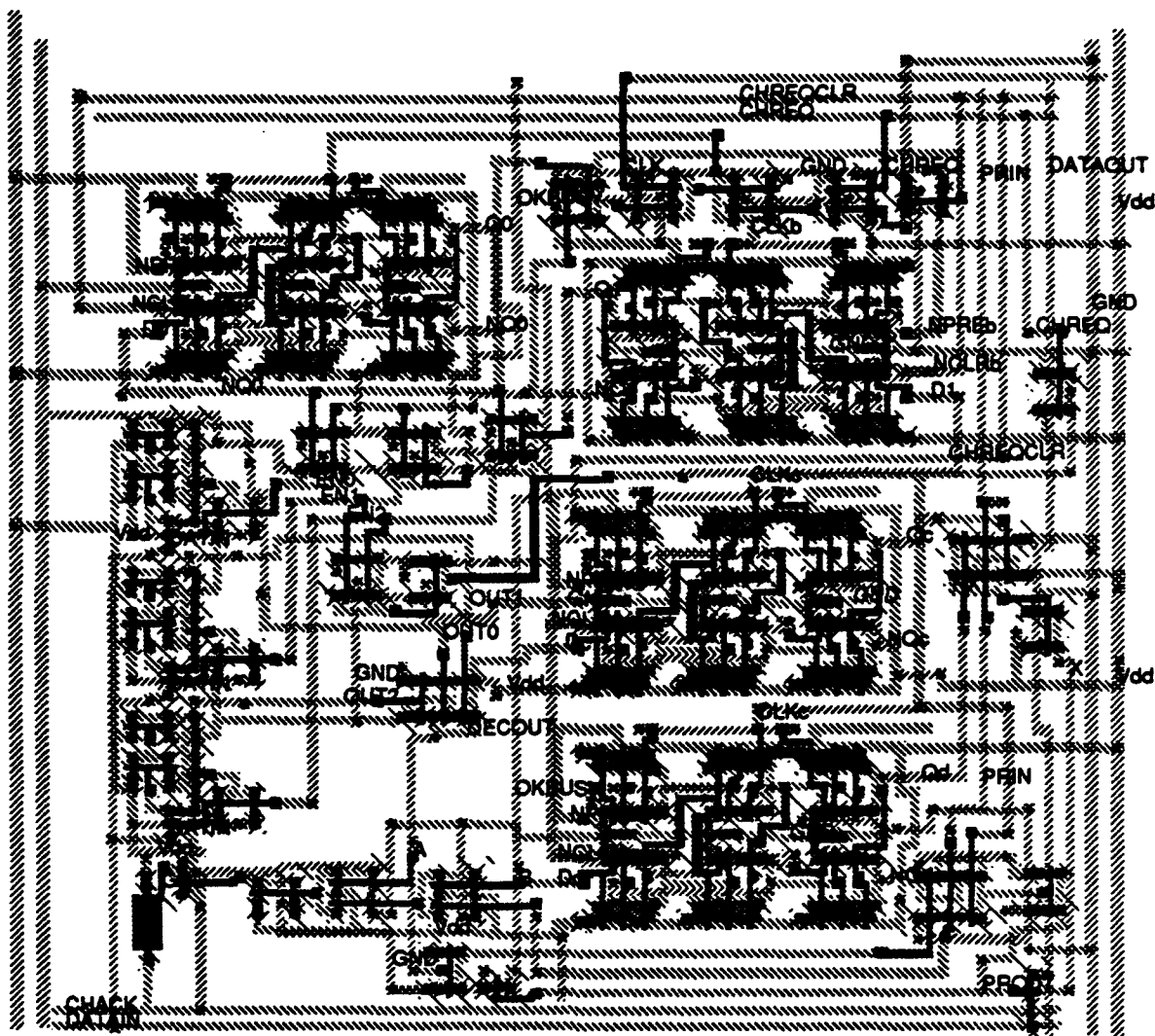


Figure 6.3. Checkplot of one cell

stated in the previous section. It was noted in Chapter 1 that any optimal solution must satisfy an application's requirements while also satisfying its design constraints. The new architectural requirements of the intermediate-level vision require a different solution, and the design constraints require a different methodology for satisfying these requirements.

In the next subsection, we discuss new observations with respect to the current approach to processing at the intermediate-level of vision and the architectural requirements they impose on the ICAP design. Thereafter, we will discuss the new design constraints on the IUA. The new constraints make it necessary to solve the problem with a different methodology.

6.2.1 Architectural requirements

In IUA GEN I, the ICAP communication network was designed to act as a data alignment network in case of apriori and fixed interprocessor communication patterns, or as a message passing network in case of data-dependent interprocessor communication. It did not have hardware mechanisms to support shared memory access. Of course, shared memory can be emulated by a message-passing system, but at the cost of speed (latency) and efficiency. In bottom-up processing in the IUA, a SIMD or SMIMD mode of computation is used because a large part of this kind of processing is inherently synchronous or staged. In top-down processing however, in addition to these two modes, the computation as well as the communication network require operation in MIMD mode such that different ICAP processors can simultaneously perform different tasks for the high-level system.

The view of intermediate-level vision in the VISIONS group at the University of Massachusetts was outlined in chapter 3. In that view, the ISR token database has become a significant part of intermediate-level vision and the ICAP level will play a critical role in its efficient implementation.

ISR is very different from traditional databases such as transaction processing systems, CAD, hypertext etc. A detailed discussion of the ISR was provided in chapter 3. Some additional properties of the ISR that evolved during last 5-6 years, as the IUA project progressed, are as follows:

- *Fine Grained:* In addition to coarse grained data structures, the ISR must support fine-grained storage, access, manipulation, and computation of tokens.
- *Distributed:* Not only may the database be distributed over multiple processors, but a

token set or even a token may be distributed over more than one processor.

- *Concurrent:* The objective of the ISR is not merely to perform storage and retrieval, but also to perform (potentially concurrent) computation on data items. In other words, in a parallel implementation, it will be an integrated parallel processing and parallel database system. For example, the current ISR implements what is called lazy evaluation, wherein evaluation of functions may not take place until their results are accessed. Thus, a function evaluation may be performed in parallel at any point between the call and access of its result.
- *Dynamic modification/reconfiguration:* Individual or grouped entries are modified dynamically and the all or part of the database may be reconfigured on-line.
- In addition to storage and retrieval, the ISR involves massive amount of data manipulation and computation.

These characteristics are most efficiently served by a shared memory programming model with a common name space at the ICAP level. The specific issues involved in implementing concurrent ISR on the IUA are currently under study by other researchers.

Note that although our message passing implementation of the ICAP communication network can emulate a common memory name space, it does not provide the low latency and efficiency required for real time image understanding and scene interpretation.

6.2.2 Design constraints

Recall that the generation 1.0 and generation 1.5 ICAP communication networks are based on the multistage network design outlined in figure 6.4. The actual IUA GEN I prototype hardware with 64 ICAP processors is organized as 16 motherboards on a backplane bus. The communication network resides on an additional motherboard to which 64 sets of wires, one set for each ICAP processor, are routed on the backplane. Each set is comprised of 6 wires (data input, data output, clock input, clock output, frame sync input, and frame sync output). The ICAP communication network requires $64 \times 2 + 4 = 132$ wires¹ on the backplane.

During the course of development of the IUA GEN I, plans were underway for the future generations of the IUA. The VLSI and packaging technologies improved to an extent that

¹This is because data input and data output signals from 64 processors have to be wired individually to the router board. The remaining four signals are common to all 64 processors, thus they require only one wire each.

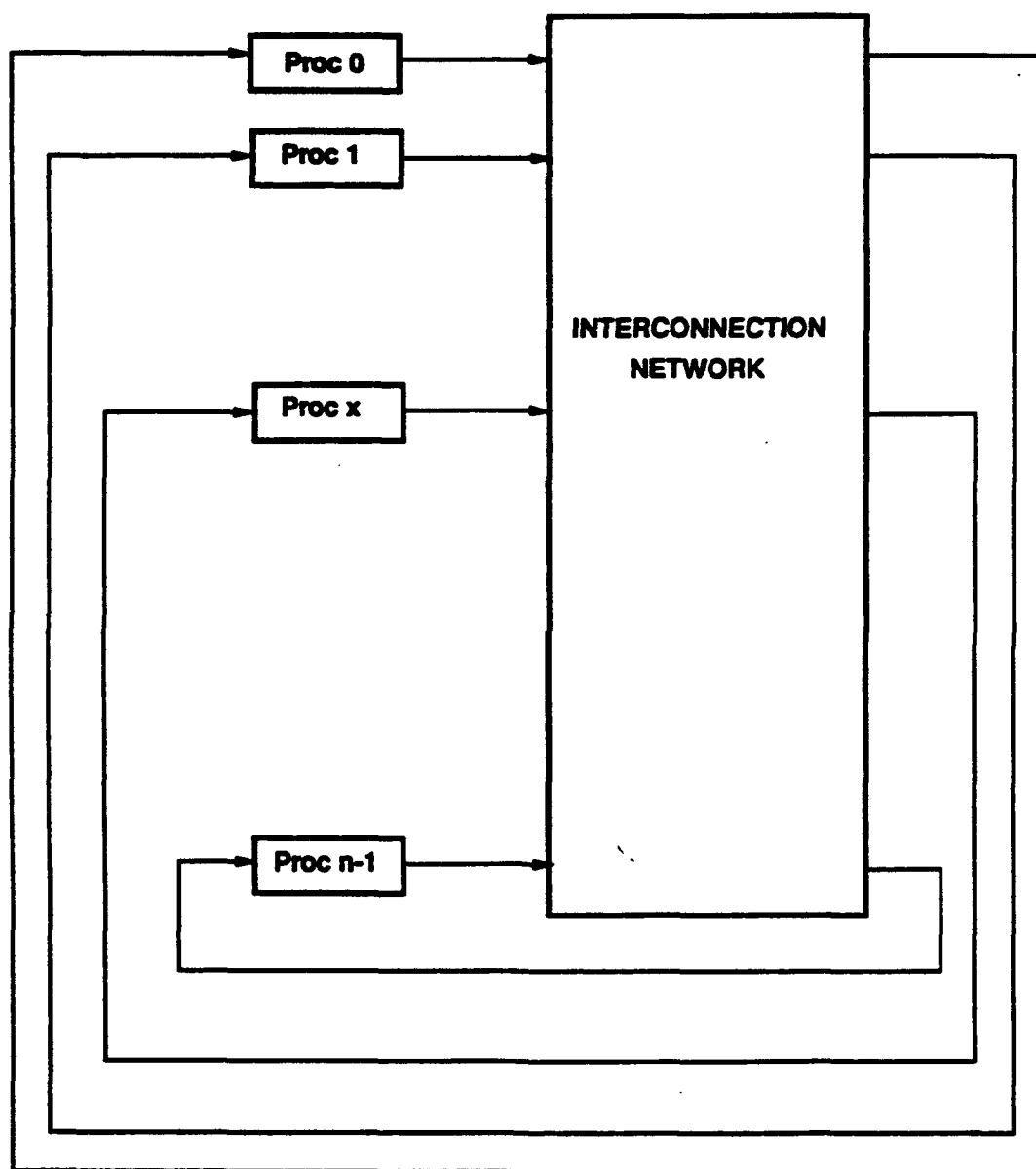


Figure 6.4. Schematic of ICAP communication network

allowed us to incorporate 256 CAAPP PEs on a single chip as compared to 64 PEs in the first design. Thus the CAAPP can be made four times larger using the same number of chips. At about the same time, the next generation of Texas Instruments DSP processors became available. The new TMS320C30 processor provides a number of additional and improved features over the TMS320C25 that was used in IUA GEN I. With respect to the ICAP communication network, each TMS320C30 (henceforth referred to as a C30) has a pair of 1 MB/S serial channels, each with handshaking capability. Serial channels from two C30s can be directly connected without any external hardware or global control, allowing them to communicate under program control of the two processors. In addition, the serial channels in the C30s have DMA capability such that packets of any length can be transferred from the memory of a source processor to the memory of a destination processor concurrently with computation. For this as well as other reasons (such as increased memory address space), it was decided to switch from the C25 to the C30 in the IUA GEN II.

The IUA GEN II prototype is organized as 8 motherboards on a backplane. Each motherboard holds 8 CAAPP chips and 8 ICAP processors. Thus, it will have 64 ICAP processors and $8 \times 8 \times 256 = 16K$ CAAPP PEs. Together, the two communication channels on each C30 require 12 wires. Thus, 8 C30s would require $8 \times 12 = 96$ pins on a motherboard connector for the ICAP communication network alone. Similarly, if we were to use a multistage network as in the IUA GEN I, it would require $64 \times 12 = 768$ traces on the backplane for the ICAP communication network alone.

Pin constraints on the motherboard connectors do not allow us to route all 16 serial channels from the 8 C30s to the backplane. Similarly, constraints on the backplane traces do not allow us to route all 128 serial channels from 64 C30s to a separate network board. It should be noted from the construction of the crossbar and Clos networks in Chapter 4 that it is not possible to reduce the number of traces on the backplane by distributing the network switches across 8 motherboards. For example, the cardinality of a cut-set [Liu 68, pp 188] in figure 4.6, when the 3-stage Clos network is cut along a vertical line between stage 1 and stage 2, or stage 3 and stage 3, is $m \times r = 2N$. Therefore, say if this network is put on two motherboards (with unequal number of chips on each), the number of traces required on the backplane for communication between the two boards alone will be $2N$ (in addition to traces required for routing signals to and from non-local board to a processor). The cardinality of a cut-set, when this network is cut along a horizontal line is $\theta(r^2)$, which is $\theta(N)$. Similar observation can be made for the crossbar network. In addition, splitting the network across a larger physical space would potentially increase clock skew and reduce its effective data rate.

We therefore started to pursue a new methodology for the ICAP communication network. It was clear that it must be based on a point to point topology in order to reduce the number of traces on the backplane and the motherboard connectors. In the following sections we describe this effort. It should be noted that the work is in progress and hence some of the results are preliminary in nature.

6.3 IUA GEN II communication network

We decided to address the problem of developing a new ICAP communication network in two phases. The first phase addresses only the design constraints of the IUA GEN II and the second phase further addresses the architectural requirements. In other words, we decided to first design an ICAP communication network that supports message passing and is implemented as a point to point network. In the second phase, mechanisms are added to the network to support fine grained, shared memory access. This section deals with the first phase and the next section deals with the second phase.

The reasons for choosing a two phase approach were: (1) This research is to be part of a deliverable project with a hard deadline, and (2) The second phase ICAP communication network requires complex and performance-critical hardware. Within a university environment, it is a high risk project. Therefore, it was necessary to have a backup scheme which would allow the ICAP processors to communicate with each other using easily built hardware. Therefore, it must be possible to upgrade the simpler scheme with additional hardware after the system has been fabricated, and possibly in the field.

Even the first phase is subdivided into two stages. The first stage deals with an ICAP communication network design for 64 TMS320C30s in the IUA GEN II prototype with no custom VLSI hardware, which is the ultimate backup scheme. If everything else fails, this scheme can be used for ICAP communication. The second stage uses a custom VLSI PARCOS III chip and the design is optimized for the IUA GEN II prototype. Issues related to extending the design to larger networks will be discussed in a separate section. Next we discuss some of the important features of C30 that are exploited in our design. Thereafter, we discuss the original Hughes Research Laboratories (HRL) proposal for an ICAP communication network in the IUA GEN II prototype, our first stage design, and our second stage design

6.3.1 Some TMS320C30 features

The TMS320C30 has three important features that make it well suited for multiple-processor systems. These features are outlined in figures 6.5 through 6.8 as adapted from [TI 90].

Figure 6.5 is a block diagram of the processor, which we will not examine in depth. The details can be found in [Papamichalis 88; TI 90]. One aspect of the architecture to be noted in this figure is that the C30 has two address and data buses, called the primary bus and the expansion bus, which allow a pair of simultaneous accesses to take place. The expansion bus is primarily intended for use with peripherals. For ease of use and programming, the expansion bus address space is mapped within the primary bus address space.

The first of the three features of TMS320C30 mentioned above is a pair of bidirectional serial channels as shown in figure 6.6. The two channels are totally independent, but identical in function, with a separate set of memory mapped control registers controlling each one. A serial channel can be configured to transfer 1 to 4 bytes per word and any number of words per message. The clock for each channel can originate either internally or externally, which, in conjunction with a special handshake mode, allows two or more TMS320C30s to communicate without any external hardware or clock. On a TMS320C30, one end of these serial channels is connected to the outside world through I/O pins and the other end is connected to the 32 bit data and 24 bit address peripheral bus².

The second of the three important features of the C30 is an on-chip DMA controller as shown in figure 6.7. The on-chip DMA controller can read from or write to any location in the memory map without interfering with the operation of the CPU. Therefore, the TMS320C30 can interface to external peripherals (the two serial channels in our case) without reducing throughput to the CPU. The DMA controller contains its own address generator, source and destination registers, and transfer counter. One end of the DMA controller is connected to the peripheral bus and the other end is connected to an internal dedicated DMA bus (DMAADDR and DMA³DATA)³ to minimize conflicts between the CPU and the DMA controller. A DMA operation consists of a block transfer of one or more words to or from memory. Since the serial channels are memory mapped, the DMA controllers on two connected processors can be programmed so that a block of words from any location in the source processor can be

²The external expansion bus is a subset of the internal peripheral bus. See figure 6.5.

³DMAADDR and DMA³DATA is one of the 4 busses internal to C30. Multiple internal busses allow multiple data accesses in one cycle (C30 is a superscalar architecture).

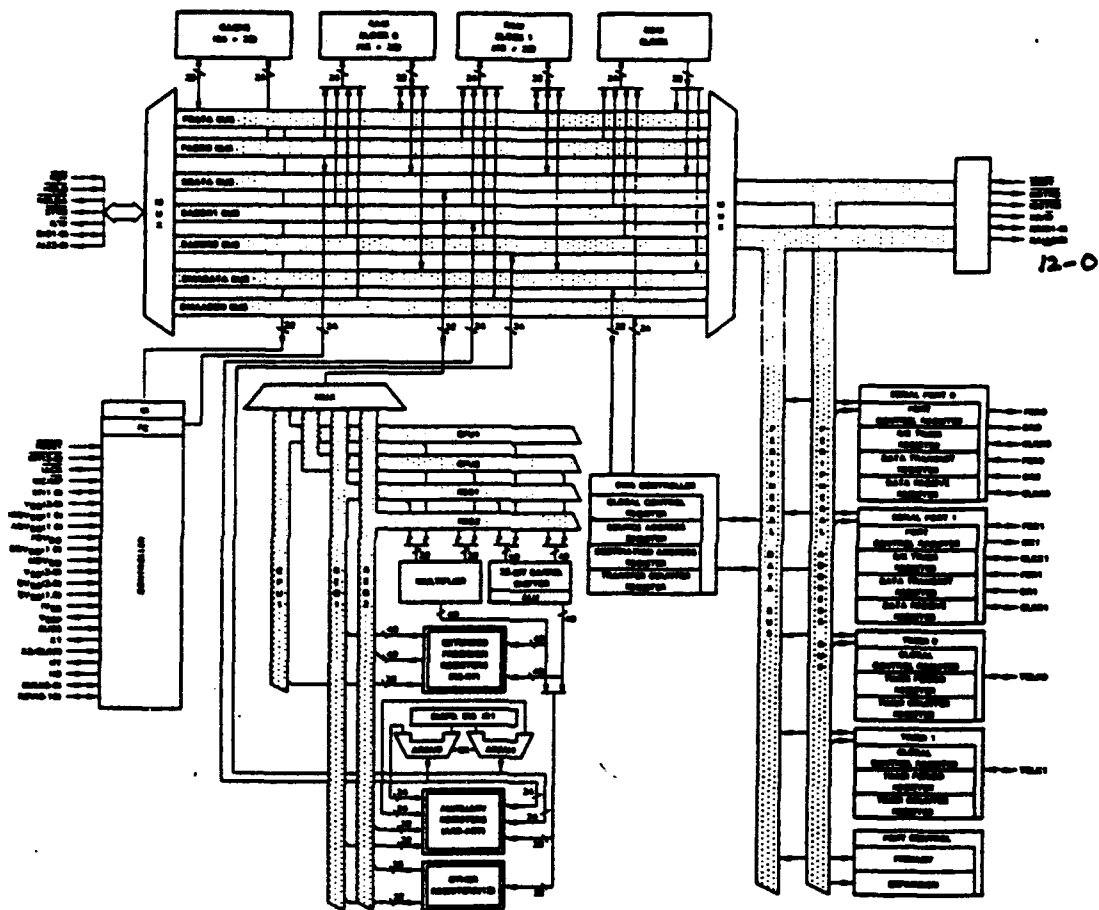


Figure 6.5. TMS320C30 Block Diagram

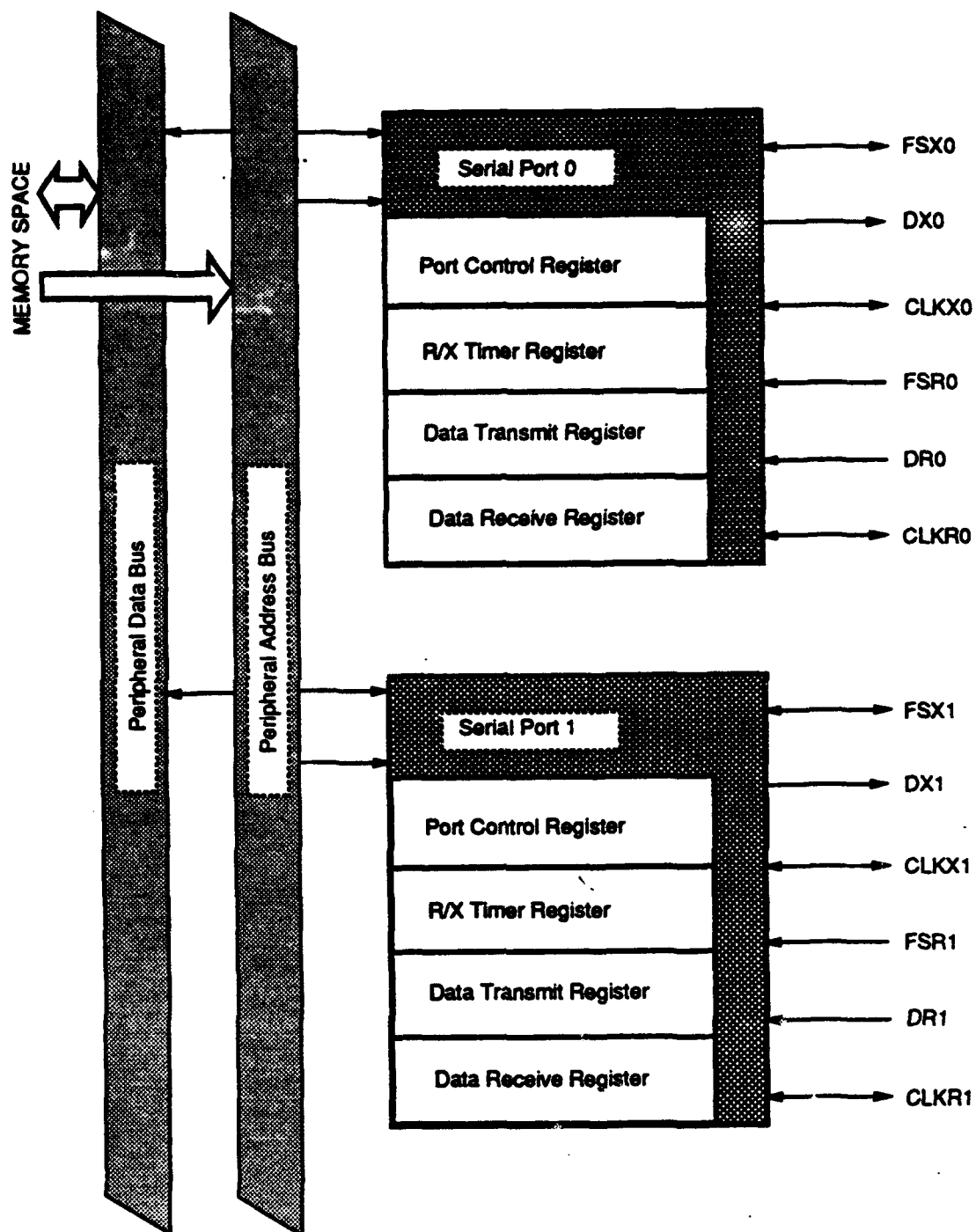


Figure 6.6. Serial channels in TMS320C30

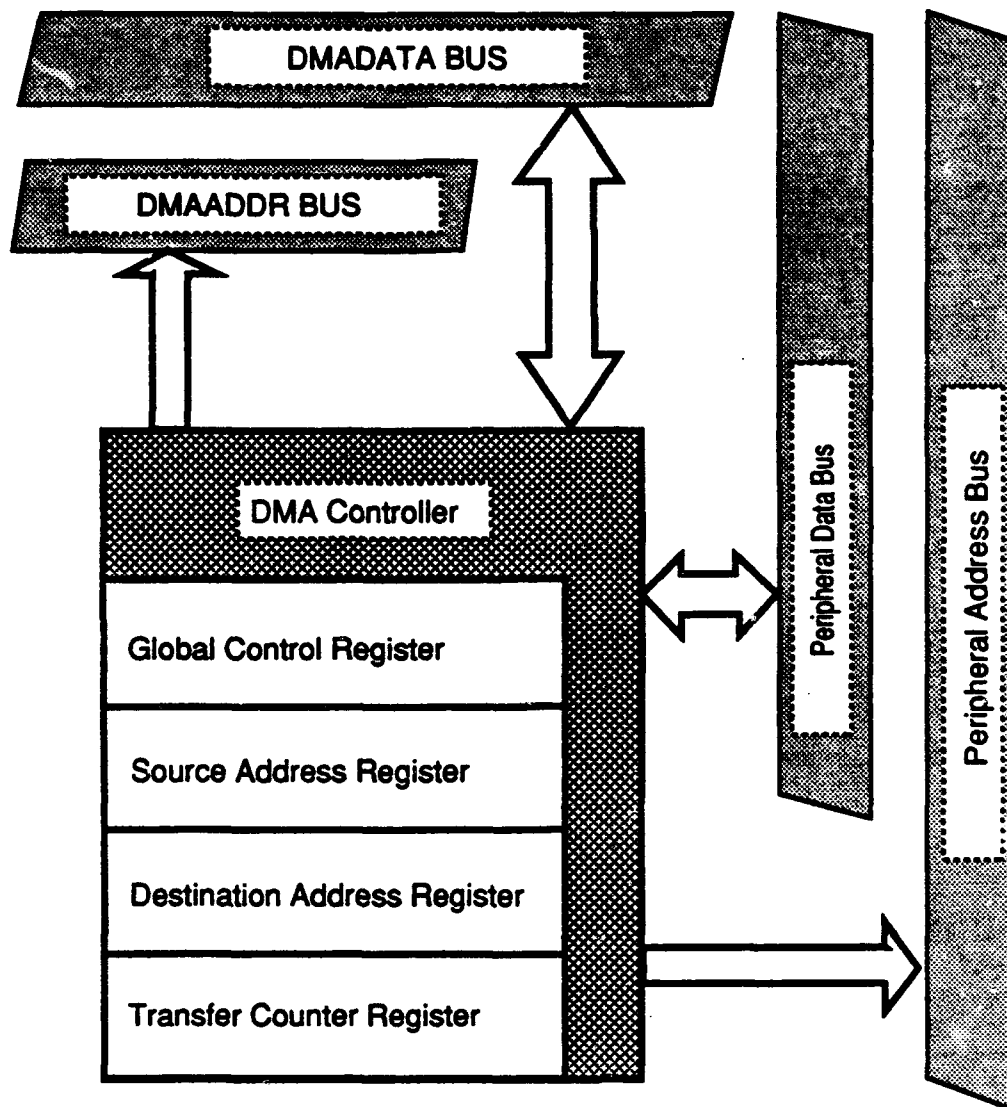


Figure 6.7. DMA Controller

moved to any location in the destination processor concurrently with processing.

The third important feature of The C30 is a set of five instructions and two external flag pins that allow multiple TMS320C30s to share a common memory. This scheme is shown in figure 6.8 and the five instructions are listed in table 6.1. The instructions and flag pins provide the necessary arbitration, handshaking and synchronization mechanisms to allow multiple C30s to share a global memory, in what is called interlocked mode. By configuring external flag zero (XF0) as an output pin and XF1 as an input pin, XF0 can signal an interlock operation request (read, write, or modify shared location), and XF1 can send an acknowledge signal for the requested operation.

Table 6.2. Interlock Operations

Mnemonic	Description	Operation
LDFI	Load floating-point value into a register, interlocked	Signal interlocked src → dst
LDII	Load integer into a register, interlocked	Signal interlocked src → dst
SIGI	Signal, interlocked	Signal interlocked Clear interlock
STFI	Store floating-point value to memory, interlocked	src → dst Clear interlock
STII	Store integer to memory, interlocked	src → dst Clear interlock

The LFI and LDII instructions are similar except for the type of data involved and perform the following actions.

1. Simultaneously set XF0 to 0 and begin a read cycle
2. Execute LDF (load floating point value) or LDI (load integer value) instruction and extend the read cycle until XF1 becomes 0 and a ready is signaled⁴.
3. Leave XF0 set to 0 and end the read cycle⁵.

The STFI and STII instructions are similar to each other except for the type of data involved and perform the following operations.

⁴The ready signal is generated by the arbitration logic to the shared memory. See figure 6.8. Interlocked mode can be used on either primary or secondary bus. In IUA GEN II, primary bus is used for shared memory.

⁵XF0 is set high during the STFI or STFII cycle.

1. Simultaneously set XF0 to 1 and begin a write cycle
2. Execute a STF (store floating point value) or STI (store integer value) instruction and extend the write cycle until a ready (by the arbitration logic) is signaled.

The fifth instruction, SIGI, is used to signal other processors in a multiprocessor configuration, and functions as follows:

1. Set XF0 to 0.
2. Idle until XF1 is set to 0.
3. Set XF0 to 1 and end the operation.

These five operations allow implementation of shared memory operations such as a busy-waiting loop, manipulation of shared counter to support a simple semaphore mechanism, or to synchronize multiple TMS320C30s. For example, a busy waiting loop can be implemented as shown below and described in [TI 90]. Location LOCK is the interlock for a critical section of code, and a nonzero value in that location means the lock is busy.

```

        LDI    1, R0           ; Put 1 in R0
L1:     LDHI   @LOCK, R1       ; Interlock operation begun
        ; Contents of LOCK → R1
        STHI   R0, @LOCK      ; Put R0 (=1) into LOCK, XF0=1
        ; Interlocked operation ended
        BNZ, L1               ; Keep trying until LOCK=0

```

To access a critical section when several processors are connected to a shared memory, semaphores may be used. The example in figure 6.8 shows two processors sharing a global memory, but can be readily extended to more processors. Two primitive, indivisible operations are defined on semaphores:

```

V(S):   S = S+1
P(S):   P:  if (S=0), go to P
        else S = S-1

```

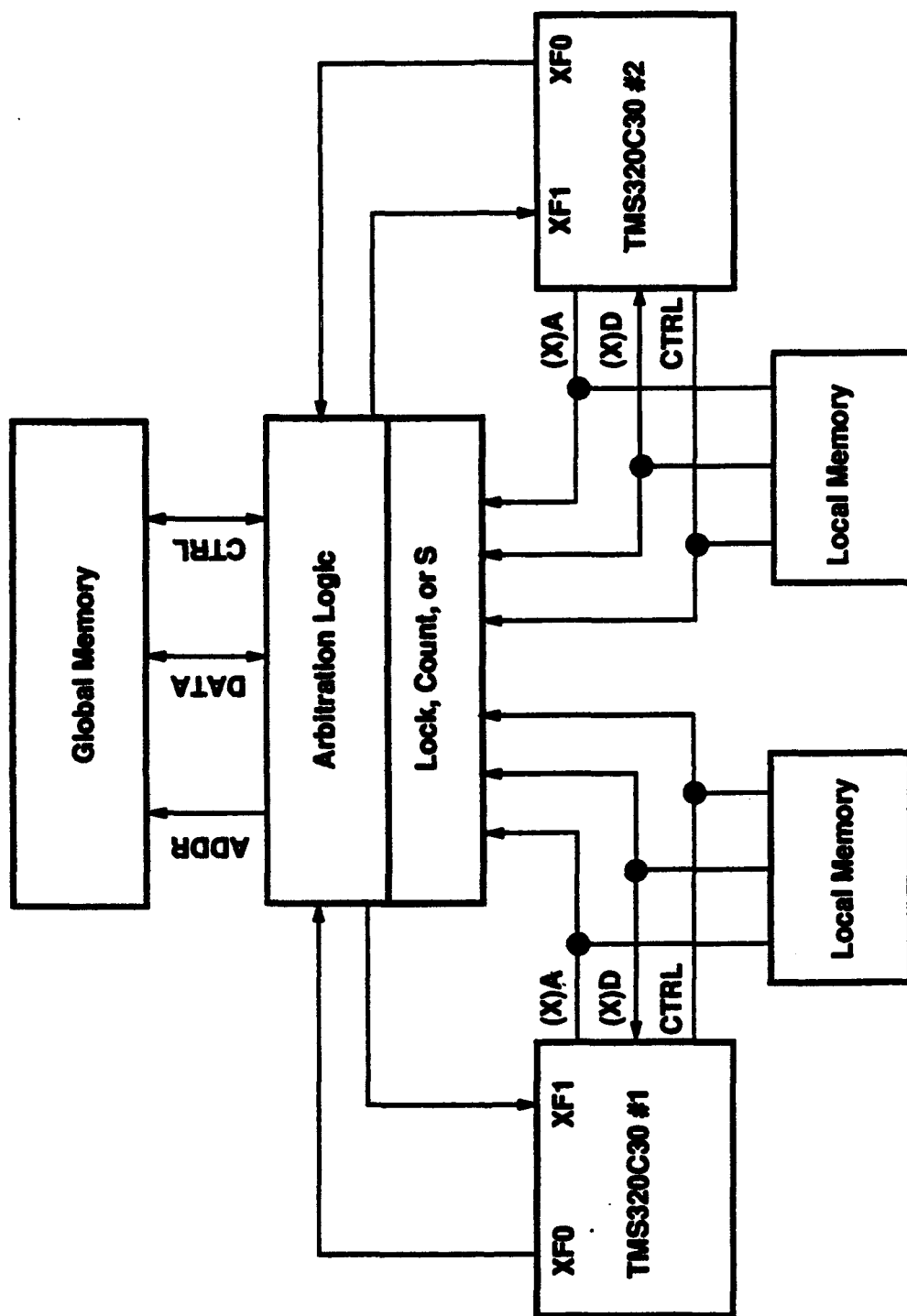


Figure 6.8. Multiple TMS320C30 sharing global memory

Prior to entering a critical section, a P operation is performed on a common semaphore, say S, which is initialized to 1. The first processor performing P(S) will be able to enter its critical section. All other processors are blocked because S has become 0. After leaving its critical section, the processor performs V(S), thus allowing another processor to execute P(S) successfully. The codes for these operations are shown below.

```

V: LDH   @S, R0      ; Interlocked read of S begins (XF0 = 0)
      ; Contents of S → R0
      ADDI 1, R0      ; Increment R0 (=S)
      STH   R0, @S    ; Update S, end interlock (XF0 = 0)

P: OR     4, IOF      ; End interlock (XF0 = 1)
      LDH   @S, R0    ; Interlocked read of S begins
      ; Contents of S → R0
      BZ    P         ; If S=0, go to P and try again
      SUBI 1, R0      ; Decrement R0 (= S)
      STH   R0, @S    ; Update S, end interlock (XF0 = 1)

```

Having discussed the serial channels, the DMA operation and how to implement shared memory on a cluster of TMS320C30s, we next discuss the design proposed by HRL to implement a message passing system on the 64 TMS320C30s of the IUA GEN II prototype.

6.3.2 Design proposed by Hughes Research Laboratories

There are 64 ICAP processors (TMS320C30) in the IUA GEN II prototype. In order to connect these processors in some meaningful point to point topology with diameter less than $\theta(N)$, more than two communication channels are required on each processor. In theoretical computer science, an open extremal problem in graph theory still exists, called the (d, k) graph problem [Cattermole 77] and consists of maximizing the number of nodes n of an undirected regular graph (d, k) of degree d and diameter k . A survey of approaches to the problem can be found in [Memmi 82]. This problem has obvious implications in point to point multiple-processor communication networks. It has been shown that there exist regular graphs with diameter k and degree d with $\theta(d^k)$ nodes (the constant is very close to 1).

Space constraints on the motherboards and the requirement that there should be a backup scheme without any custom VLSI hardware, do not permit a separate custom VLSI I/O processor to be attached to every ICAP processor in order to increase the number of channels beyond the two supplied. Therefore, a scheme had to be devised to solve this problem without custom hardware.

Before we discuss the design proposed by HRL, a brief note is in order. If we had chosen to connect the 64 TMS320C30s in a 6-dimensional hypercube topology using a pair of serial channels per link (via an attached I/O processor per node), it would have required 48 serial channels or $48 \times 6 = 288$ pins on the motherboard connector for the ICAP communication network alone. The diameter in that case would have been 6. As will be seen shortly, the HRL' solution achieves lower latency. Also, the motherboard connector in the IUA GEN II prototype is comprised of 3 128 pin connectors. Therefore, each motherboard has only $3 \times 128 = 384$ pins. Even if we ignore the pins used for ground connections⁶, only $384 - 288 = 96$ pins remain to support all other system functions. Obviously, such an arrangement is unworkable.

The solution proposed by HRL is shown in figure 6.9. Using the shared memory support mechanisms provided by the TMS320C30 architecture, groups of 4 ICAP processors are connected in shared memory clusters called supernodes or SNODEs. Each SNODE is treated as one unit in the ICAP communication network. There are 8 communication channels in each SNODE and there are a total of 16 SNODEs in the IUA GEN II prototype. By using a binary hypercube topology, one can construct a 256 SNODE system. The 16 SNODEs are connected in a 16-node Dual-Hypercube topology as shown in figure 6.10. Simply stated, instead of one communication link between two adjacent SNODEs, there are two. Using the handshake mode of the TMS320C30 serial channels, one of these channels is programmed for outgoing messages and the other channel is programmed for incoming messages. For end-to-end TMS320C30 communication, the shared memory (SMEM) of an SNODE serves as intermediate storage space in a store-and-forward protocol. It is possible to set up a DMA mode such that either a word is read from the SMEM and put on a serial output channel (in handshaking mode, a serial channel has to be prepogrammed as an input or an output channel) or a message is read through a serial input channel and is stored in the SMEM without interrupting the CPU. By suitably programming the DMA controllers and the serial channels on two connected ICAP processors, it is possible to transfer a message of

⁶To alleviate signal coupling and other noise related problems, every alternate signal is grounded on PCB connectors. This is especially critical in signals that are broadcast to multiple boards such as address and data lines. Such signals induce large dI/dT noise and thus alternate ground signals reduce this as well as common mode noise.

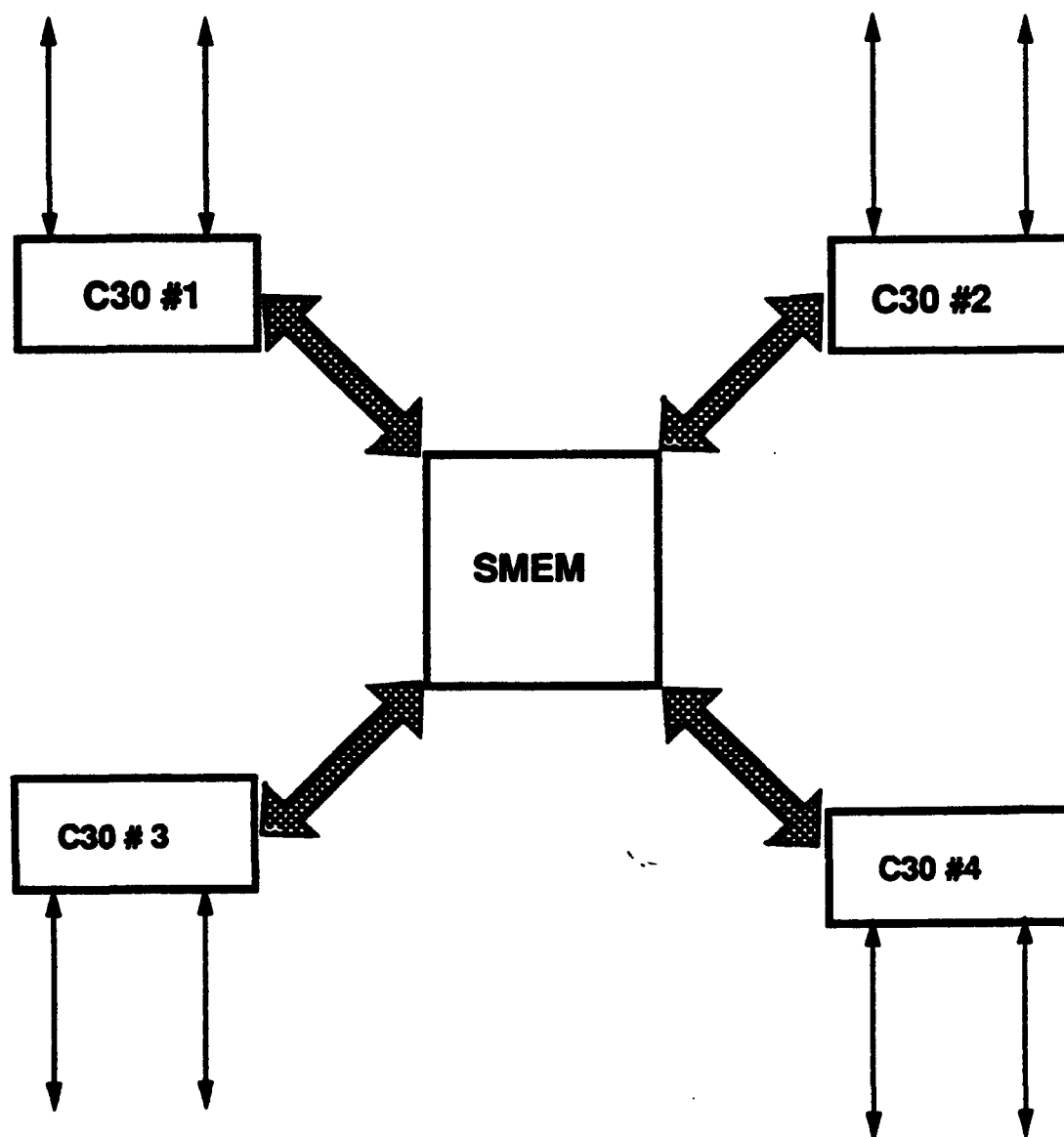


Figure 6.9. HRL SNODE structure

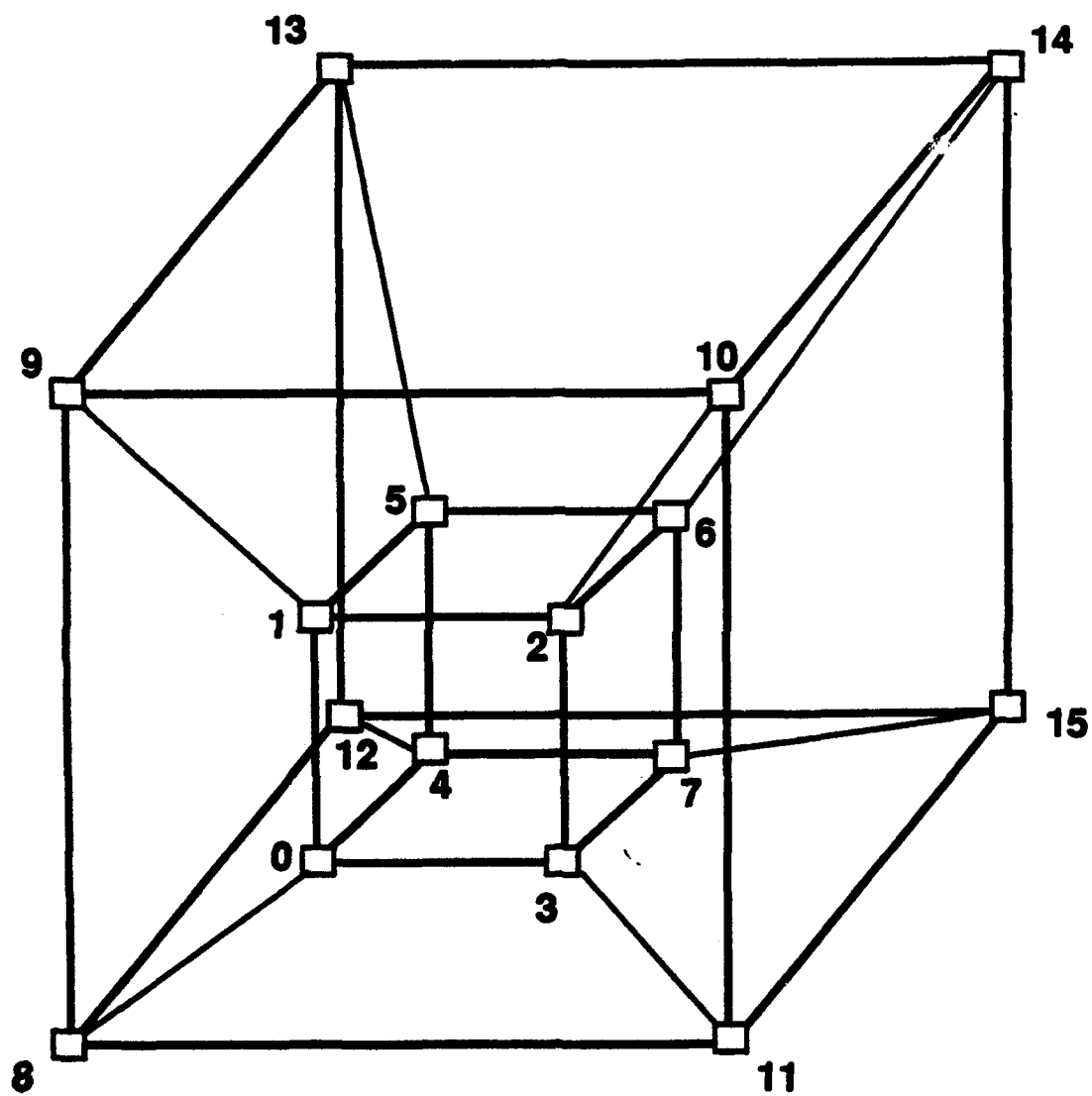


Figure 6.10. 16-Node Dual-Hypercube

one or more words from anywhere in the SMEM of the source processor to anywhere in the SMEM of the destination processor without interrupting processing. It should be noted that there is only one DMA controller in every TMS320C30 and it must be preprogrammed for reading from or writing into SMEM. As such, the DMA controller on a TMS320C30 can be used for only one serial channel at a time.

Performance

In a 4-dimensional binary hypercube configuration, the maximum SNODE to SNODE distance is 4. Each serial channel has a maximum data rate of 1MB/sec (the maximum clock frequency of the serial channels is 8MHz or 125nS for a 32MHz input clock frequency to the C30). Assuming a message length of 64 bits, it will take $64 \times 125 = 8\mu\text{S}$, to transmit a 64-bit message through a C30 serial channel.

For non-neighbor communication, SMEM in the intermediate SNODEs acts as temporary storage for messages in transit. A processor in an intermediate SNODE that receives a message must interrupt another processor on the same SNODE to forward this message to the next SNODE. Interprocessor interrupt on a SNODE is implemented by shared, memory mapped registers. If a processor writes a 1 at a specific shared address, it activates the hardware interrupt of a particular C30.

In addition to the store and forward overhead, the overhead of setting up DMA and serial channel transfer modes at the two ends and starting the transfer must be considered. We assume that it will require 30 additional processor cycles ($2\mu\text{S}$), resulting in a communication time of $10\mu\text{S}$ between two processors on adjacent SNODEs. The best time between any two ICAP processors corresponds to the diameter of the hypercube and is $40\mu\text{S}$. The worst case time for many to many or many to one communication will take at least $\log^2 N$ steps or $16 \times 10 = 160\mu\text{S}$. For example if 63 C30s send a message to the 64th C30 at once, it will take 16 steps to receive the 60 messages (from remote SNODEs) via the four input channels on the destination SNODE.

Next we discuss our Stage I design and show how it improves upon this performance.

6.3.3 Stage I design

In the HRL proposal, there are two SNODEs with 4 C30s each on every motherboard. Of

the 16 serial communication channels on each motherboard, 12 are routed to other motherboards through the backplane. The remaining four channels are used for interconnecting the two SNODEs on a motherboard. Each serial channel requires six pins on the motherboard connector. Thus, there are $12 \times 6 = 72$ pins on the motherboard connector for the ICAP communication network. When configured in handshaking mode, a serial channel can send data in only one direction at a time, the other direction being used for acknowledgment. Each serial channel is capable of only a 1MB/S data rate. The cumulative capacity of all of the serial channels on one SNODE in HRL design is 4 MB/S. When either a source or a destination for DMA is in the internal memory of the C30, as it is in the case of the memory mapped registers, and the other end of the transfer is on the primary bus (SMEM in this case), the DMA timing for transfer of one four-byte word is given by

$$(2 + C_r + 1) \times \frac{2}{f}$$

Where f is the system clock frequency (32 MHz in our case) and C_r is the number of wait states in the SMEM (3 in our case). Thus each C30 in the IUA GEN II prototype is capable of transferring

$$\frac{4}{(2 + C_r + 1) \times \frac{2}{f}} = \frac{4 \times 32 \times 10^6}{(2 + 3 + 1) \times 2} = 10.7 \text{ MB/S}$$

Therefore, an SNODE can support communication between more than four C30s.

The Stage I design incorporates several changes. First, all eight C30s on a motherboard reside in single SNODE, so that there is only one SNODE on every motherboard. Even so, the DMA of one processor has a much higher capacity than all of the serial channels in an SNODE combined. The second modification is to change the interconnection topology of the network on the backplane from a hypercube to fully connected between motherboards. Each SNODE on a motherboard has 16 serial channels. Seven of these channels are configured as output channels and seven as input channels, connected to the seven SNODEs on other motherboards. The remaining 2 channels on each SNODE are connected to neighboring SNODEs in a daisy-chain that terminates at an outside connector. This connector is used for diagnostic purposes or for direct input of data to the C30s or for output of data from them. This connector can also be used for instrumentation purposes.

In the Dual-Hypercube topology, there are 12 serial channel connections on the backplane. By having only two additional serial channels (a total of 14), one can have a fully connected topology between the motherboards. The benefit of this topology will be apparent in the following paragraphs and in the next design.

Performance

The maximum SNODE to SNODE distance in the Stage I design is 1. Using the same figures for transmission of a 64-bit message as in the HRL design, the best communication time between any two ICAP processors without conflicts in the Stage I design is $10\mu\text{S}$. The worst case time is observed in many to one or one to many communication. At most there can be 8 outgoing messages from a motherboard queued on a single serial channel, which will take $80\mu\text{S}$. Notice that between the HRL design and the Stage I, the worst case communication time has improved by a factor of 2. Also notice that the average case communication time has improved by a factor between 2-4 (worst case vs. the best case). The average time will depend on the communication pattern.

6.3.4 Stage II design

The Stage II design requires custom VLSI hardware. Within the time constraints imposed by the delivery schedule for the IUA GEN II system, it may not be possible to deliver functional Stage II hardware in time. Also, as discussed earlier in this chapter, an additional constraint was imposed on the ICAP communication network design that there should be a backup scheme that does not require custom VLSI hardware. Therefore, Stage II has been developed as an add-on to the Stage I design that can be installed without modifying the layout of the backplane or the motherboards.

The organization of the Stage I and Stage II designs is shown in figure 6.11. The dotted area is the communication daughterboard on every motherboard. Converting the system from the Stage I design to Stage II involves replacing this daughterboard. The serial channel signals along with XF0 and XF1 from eight C30s are routed to the daughterboard and from there, 7 pairs of channels are jumper-connected to the backplane connector. The eighth channel pair is wired to an external connector for diagnostic purposes as discussed above. The arrangement so far is functionally equivalent to the Stage I design.

For now, let us call the custom VLSI hardware for Stage II design, PARCOS III. For interprocessor communication, the Stage II design does not use the serial channels. Instead, all messages are encoded as memory mapped writes to the shared memory. PARCOS III implements part of the SNODE shared memory which is used to implement message passing queues on the send and receive sides of PARCOS III, and for queue management. PARCOS III is additional master of the SMEM bus and implements an interlock instruction set identical

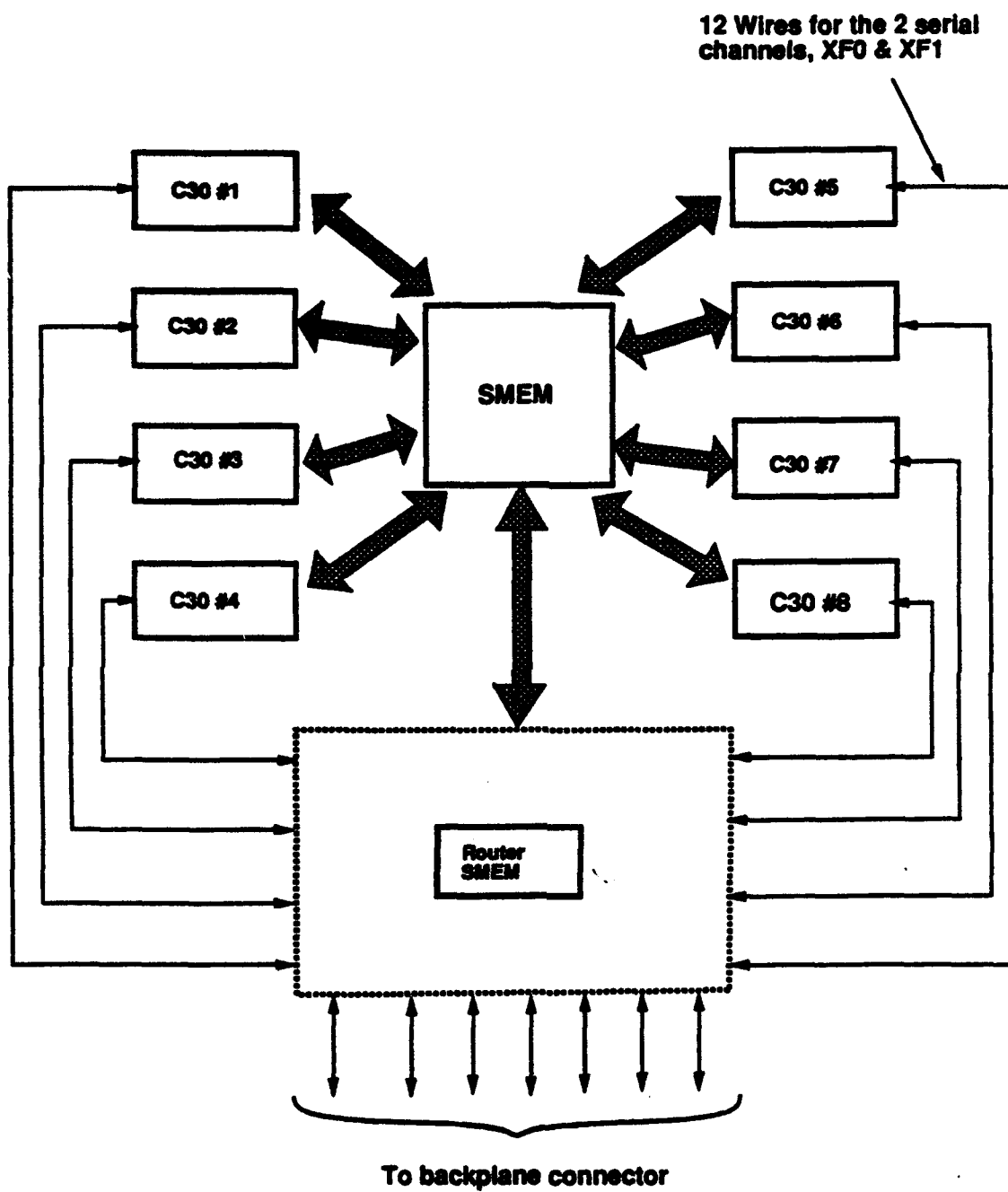


Figure 6.11. SNODE on a motherboard

to that of the processors.

The message passing mechanism implemented in the Stage II design is to take a 64 bit (2 word) message written into a PARCOS III queue by a source processor, deliver it to the destination SNODE PARCOS III, which in turn writes these 2 words at a specific location in the destination SNODE's shared memory and interrupts the destination processor.

Architecture

Figure 6.12 is a block diagram of PARCOS III. Only one of the seven pairs of channels is shown in the diagram. The other pairs are identical except for their address mappings. In any SNODE a number (to be defined shortly) of memory locations are reserved for the ICAP communication network. Seven 1-bit Empty/Full registers, one corresponding to each send queue in PARCOS III, are memory mapped in the shared address space. These registers are set by the corresponding port controller # n ($0 \leq n \leq 6$) to indicate whether the corresponding send queue is empty or full. Before sending a message, a source processor has to read this bit in interlocked mode. The actual size of the send queue has yet to be determined, subject to simulations with actual data. We chose a size 8 which would take care of the case in which every processor on an SNODE simultaneously sends a message to a common SNODE.

Each item in the send queue is 68 bits in length. Of these bits, 64 are the actual message and the remaining bits represent the number of the processor on the destination SNODE (Currently we have only 8 processors on an SNODE. A 4-bit address allows future extension of up to 16 ICAPs on a single SNODE). The send queues are mapped into the shared address space so that in order to write in to one of them, a processor performs a write to a specific location. The send queue on an SNODE transmits data to the connected receive queue on another SNODE via the 6 wires employed in the Stage I design. Four of these wires are used for sending data in 4-bit nibbles. One wire is used to gate writing of the nibble into the receive queue, and the last wire is used to determine whether the receive queue is empty or full. A receive queue is similar to a send queue in design but different in functionality. Each pair of queues in a channel has its own controller. All seven of these controllers operate under the control of the PARCOS III controller. A message in the receive queue is stored at a location in SMEM determined by the destination processor number. If that location is not empty (because the processor has not yet removed its previous message), the blocked message is inserted at the end of the queue to prevent other received messages for other processors on the same SNODE from being blocked. If there are messages in multiple receive

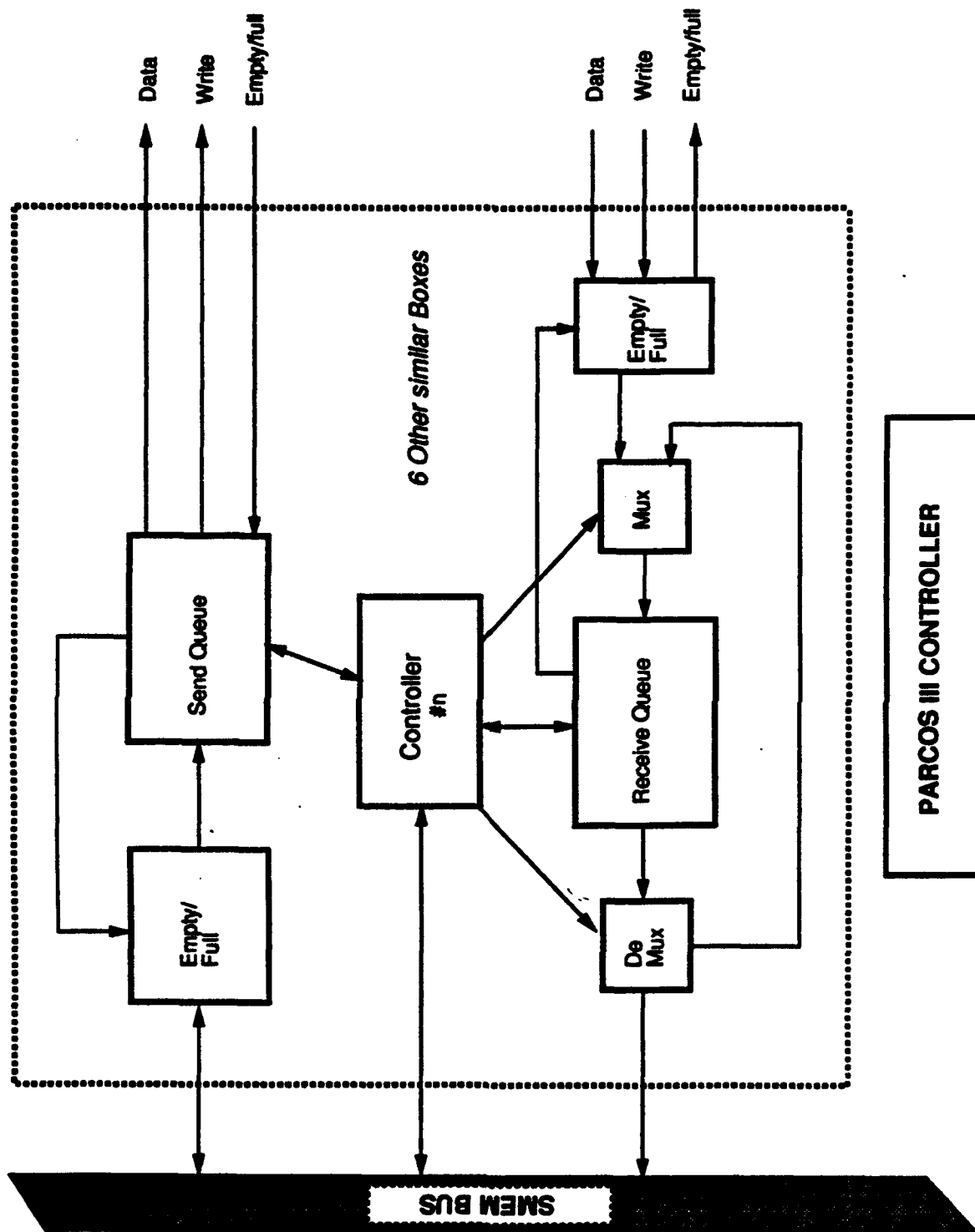


Figure 6.12. Block diagram of PARCOS III

queues then the respective controllers are polled (and operated) in a round-robin fashion by the PARCOS III controller. Writing into send queues⁷ is governed by the arbitration mechanism on the SNODE itself.

Operation

An ICAP processor performs the following operations in order to send a 64 bit message under the Stage II design.

1. The ID of the appropriate send queue is derived from the destination processor number. In interlocked mode, the processor checks if the send queue is empty by reading the corresponding Empty/Full bit.
2. If the queue is full, it releases the SMEM bus and retries later.
3. If the queue is empty, it writes two words at specific locations, associated with the queue and then releases the SMEM bus. There are 7 sets of 8 pairs of these locations, one set for each SNODE send queue with each pair assigned to a specific processor in the destination SNODE. Depending upon which of the 8 pairs is written to, the port controller will extract the 4 bit⁸ destination processor number and attach it to the 64 bit message in the send queue.

If a send queue has one or more messages, it tries to send them to the corresponding receive queue, depending on the Empty/full signal from the receive queue. There are two hardware approaches to transmitting data from the send queue to the receive queue. In a centralized clocking scheme, a write signal is merely used as a gating signal for the message. The clocking out and in of the data is performed using the system clock. Clock transitions can be used to transfer a 4-bit nibble at the rising edge of the clock and another at the falling edge. In a non-centralized clock scheme, the write signal is actually used for clocking every nibble out of and in to the queues. The first scheme provided twice the data transfer rate of the second approach.

As soon as there are one or more messages in a receive queue, its corresponding port controller, under control of the PARCOS III controller, tries to deliver the messages to their

⁷Note that only one ICAP can write a message at a time in any SNODE shared memory.

⁸Note that in IUA GEN II, only three bits are sufficient.

appropriate locations in the SMEM. To deliver a message, the port controller performs the following operations:

- 1 From the destination processor number, it determines where the message is to be delivered. In interlocked mode, it checks if the destination processor has removed the previous message by reading a predetermined location. There are 8 sets of locations with 3 words each, one set for each processor on an SNODE. The first word in a set is used by the port controller as well as the corresponding processor for handshaking. Every time the processor removes a 64-bit message from the other two words, it sets the first word to 0. Every time a port controller writes a message in the last two words, it sets the first word to 1 so that no other port controller will overwrite the message before the processor removes it.
- 2 If the destination buffer is full, then the port controller moves the message from the front of the receive queue to the end and releases the SMEM bus.
- 3.a If the destination buffer is empty, the port controller sets the first word to 1 and writes the remaining 2 words containing the actual message.
- 3.b The port controller writes in the memory mapped interrupt register to indicate receipt of a message to the destination processor.
- 3.c The port controller releases the SMEM bus.

When the destination ICAP receives an interrupt, it removes the message and sets the first word to 0 in interlocked mode.

Having discussed the operation of the Stage II design we next discuss its performance.

Performance

The best communication time between any two processors on different SNODEs is calculated as follows:

The time for the first step, in which the source processor writes a message in to the send queue, is comprised of:

1. In interlocked mode, check if the send queue is empty or full. This takes 5 cycles. (Recall that there are 3 wait states in the SMEM.)

2. Write two words in the send queue. This takes 8 cycles.
3. Release the SMEM bus, which takes 4 cycles.

Thus the total time for the first step is 17 cycles. With a 32 MHz system clock, each instruction cycle being half the clock rate, the cycle time is 62.5 nS. Therefore, 17 cycles will take

$$17 \times 62.5 = 1062.5 \text{ nS}$$

Or 1.06 μ S. Note that with a zero wait state SMEM, this step would have taken 5 cycles or 312.5 nS.

A message from the send queue to the receive queue is broken into 4-bit nibbles. Assuming the approach based on a centralized clock, a 68-bit (17-nibble) packet will take

$$\left(\frac{1}{2} \times 17 \times \frac{1}{32 \times 10^6} \right) = 265.5 \text{ nS}$$

to be transmitted between PARCOS III chips.

The time for delivering a message from the receive queue to the SMEM is comprised of the following steps:

1. Check if the destination buffer is empty. This will take 5 cycles.
2. If empty, mark this buffer full and write the 2 words of the message. This will take 12 cycles.
3. Set the appropriate bit in the memory mapped interrupt register. This will take 4 cycles.
4. Release the SMEM bus. This will take 4 cycles

Thus the total time for this step is 25 cycles or

$$25 \times 62.5 = 1562.5 \text{ nS}$$

Note that with a zero wait state SMEM this step would have taken 7 cycles or 437.5 nS.

Thus, the total time for communication is:

$$1062.5 + 265.5 + 1562.5 = 2890.5 \text{ nS}$$

or $2.89\mu\text{S}$, with 3 wait state SMEM, or

$$312.5 + 265.5 + 437.5 = 1015.5 \text{ nS}$$

or $1.02\mu\text{S}$, with a zero wait state SMEM.

It is not realistic to analytically calculate average or worst case latency in this design, because they depend on the actual computation taking place in the processors. For example, the communication latency due to delay on the SMEM bus will depend upon the activity of the other processors which might be using it for computation. Similarly, the latency in the send and the receive queues will depend upon the computation and communication in other processors, which, in MIMD processing, is in general, nondeterministic. This analysis is therefore subject to simulation with actual tasks on the ICAP.

Note that between the Stage I and Stage II designs, the best communication time has improved by a factor of 3 to 10, depending upon the speed of the SMEM.

In the next section, we discuss schemes for supporting shared memory at the ICAP level in the IUA. The design in the next section was carried out independently of the other designs in this chapter. Therefore, we refer to it as the IUA GEN II+ communication network.

6.4 IUA GEN II+ communication network

This section describes the second phase of the new ICAP communication network design. We discuss mechanisms that are added to the network of the Stage II (or they can be considered additions to Stage I) design to support fine grained, shared memory operations. The field of shared memory multiple-processor designs is very rich, but different systems have been, in general, influenced by the application space as well as the architectural constraints. It should be noted that in our case, the logical name space is shared and not the physical address space. The ICAP in IUA GEN II+ is therefore, a NUMA (Non-Uniform Memory Access time) machine. Another example of NUMA architecture is the BBN Butterfly [LeBlanc 88]. In order to keep our design relatively simple, we decided to support five operations, which can be visualized as an extension of the Stage II design. The 5 operations are:

- **Send**

Send two words from the local SNODE shared memory to the remote SNODE shared memory.

- **Send.int**

Send two words from the local SNODE shared memory to the remote SNODE shared memory. Interrupt a specific remote processor when completed.

- **Get**

Retrieve two words from the remote SNODE shared memory to the local SNODE shared memory.

- **Get.int**

Retrieve two words from the remote SNODE shared memory to the local SNODE shared memory. Interrupt a specific processor in the local SNODE when completed.

- **Fetch & Replace (F & R)**

Fetch a word from a specific location in the remote SNODE shared memory, replace it by the value supplied, store the fetched word at a specified location in the local SNODE shared memory, and interrupt a specified local processor when completed.

By using F & R in conjunction with multiple Send or Get operations, a number of shared memory primitives such as a critical section guarded by semaphores can be implemented.

The organization of an SNODE that supports shared memory is shown in figure 6.13. We will refer to the custom VLSI router as PARCOS III+. PARCOS III+ is another master on the SMEM bus and always snoops on the bus signals. In addition, it implements an interlock instruction set like that of the C30s.

All 5 of the operations outlined above are encoded as memory mapped writes to the part of the shared memory that is implemented in PARCOS III+. The architecture of PARCOS III+ is discussed next.

Architecture

Figure 6.14 is a block diagram of PARCOS III+. Only one of the 7 pairs of channels is shown in the diagram. The other pairs are identical except for their address mappings. Comparing this figure to figure 6.12, it can be seen that rather than writing into the send queue, a processor requests one of the 5 operations by writing a three-word instruction in one of the task queues. There are seven 1-bit Empty/Full registers, one corresponding to each task queue in PARCOS III+, that are memory mapped in the shared address space.

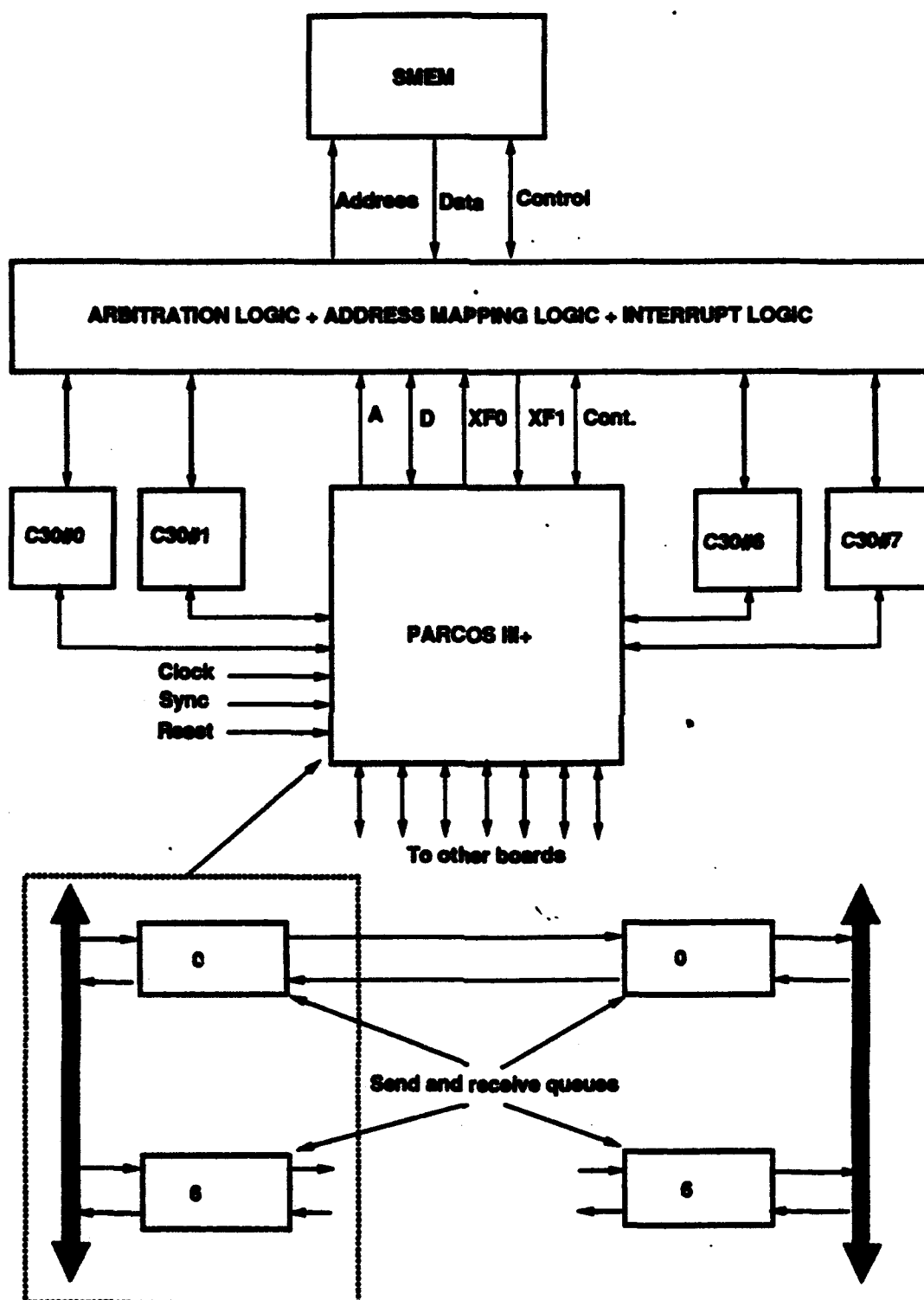


Figure 6.13. SNODE for shared memory

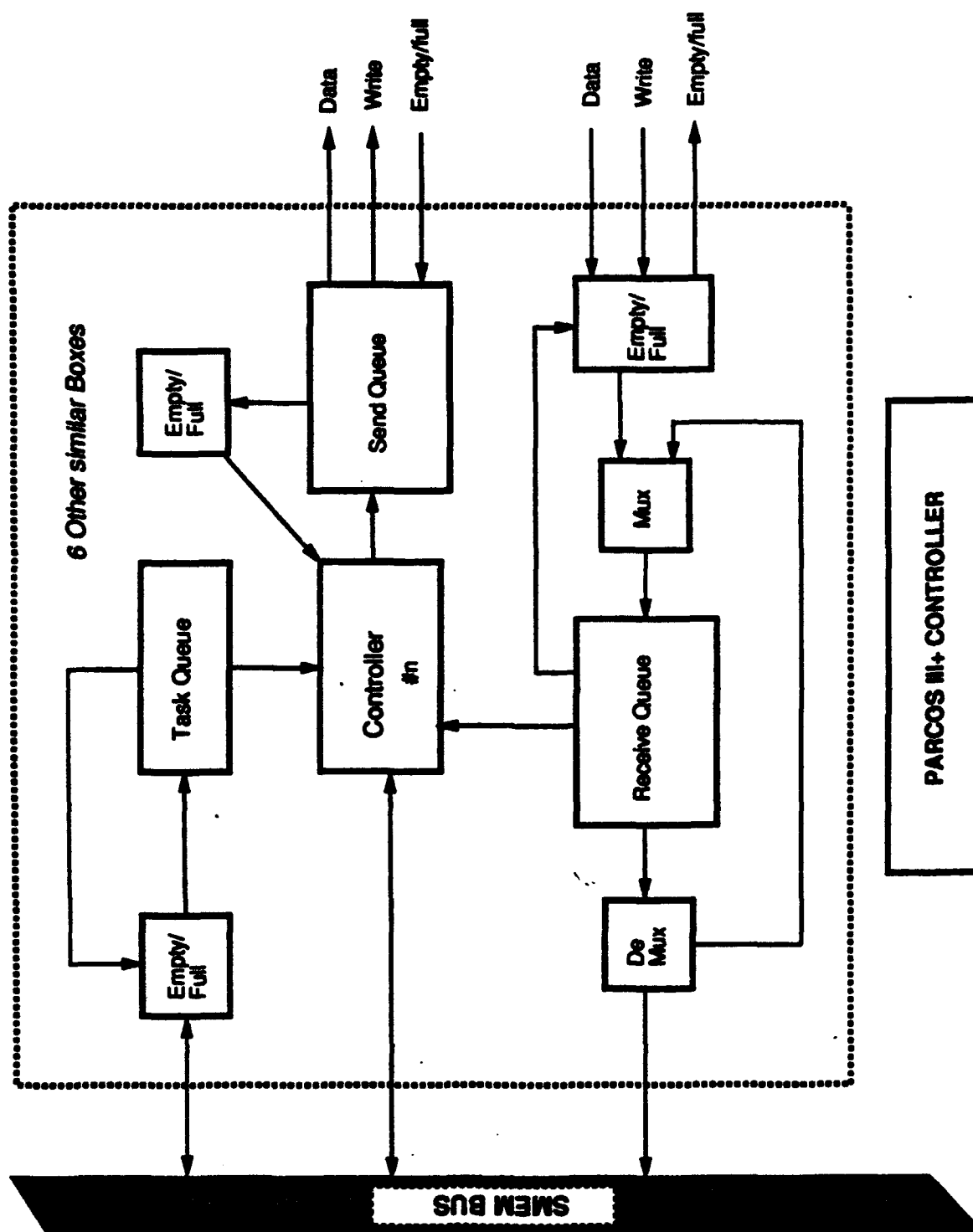


Figure 6.14. Block diagram of PARCOS III+

These registers are set by the corresponding port controller #n ($0 \leq n \leq 6$) and the task queue to indicate whether the queue is empty and thus able to receive this request. Before making a request, a source processor has to check this bit in interlocked mode. The size of the send queue must be determined from simulations with actual data.

Each item in a task queue is three words in length. The first word contains the operation, the source processor number, the destination processor number, and a value (for the F&R operation). The second word contains the address for the remote SNODE shared memory. The third word contains the address for the local SNODE shared memory. The format of the first word is:

OPCODE	Processor #	Replace value

The 3-bit OPCODE field indicates one of the five operations. The four processor # bits⁹ indicate the source or the destination processor to be interrupted, depending upon the operation. The remaining 25 bits can be used as the replacement value in an F&R operation¹⁰. The second and third words contain 32-bit addresses, even though there are currently only 1M words in SNODE shared address space. The task queues are mapped into the shared address space so that a processor merely has to perform memory writes to three contiguous locations.

A port controller continuously monitors the task and receive queues and performs operations requested via either of these. The communication between a send queue and a receive queue is performed just as in the Stage II design.

Operation

A processor executes the following steps to perform one of the five shared memory operations.

1. From the destination SNODE number, it determines which task queue is associated with the desired SNODE. In interlocked mode, it checks if the queue is empty by reading the corresponding Empty/Full bit.

⁹Recall from Stage II design, that only 3 bits are sufficient for IUA GEN II

¹⁰It is possible to specify a 32-bit value by modifying this instruction to a four word instruction.

2. If the task queue is full, it releases the SMEM bus and retries later.
3. If the task queue is empty, it writes a three word entry in the queue, and releases the SMEM bus.

If there are one or more requests in the task queue and the send queue is empty, the port controller tries to satisfy the requests one by one. From the first word of a request in the task queue, it determines the operation requested and simultaneously puts the word in the send queue. If the requested operation is Get, Get_int, or F&R, it moves the second and the third word from the task queue to the send queue and proceeds to the next request. If the requested operation is Send or Send_int, it moves the second word from the task queue to the send queue, and uses the contents of the third word in the task queue as the starting address of the two words which will be fetched from the local SMEM; it then moves them to the 3rd and the 4th words in the send queue, before proceeding to the next request.

Process of communication between the send and receive queues is identical to that of the Stage II design. Each packet in the send queue is either 3 or 4 words long depending upon the operation. The first word in the send queue has the same format as in the task queue. If the requested operation is Get, Get_int or F&R, the second and the third words are those copied from the task queue. If the requested operation is Send or Send_int, the second word contains the address in the remote SNODE's shared memory and the remaining two words contain the actual value to be stored in the remote locations.

As soon as there are one or more messages in a receive queue, the corresponding port controller tries to satisfy them, by performing the following steps.

1. From the first word of the message it determines the operation requested.
2. If the requested operation is Send or Send_int, it uses the second word as an address into the local SNODE shared memory and, using interlocked mode, stores the remaining two words starting at the specified address. For a Send_int operation, it additionally writes in the appropriate interrupt register bit. Lastly, it releases the SMEM bus.
3. If the requested operation is Get, Get_int or F&R, it checks whether the send queue is empty or full. If the queue is full, it moves the request from the front of the receive queue to its end.
4. If the send queue is empty, it uses the first word to determine the operation and does the following:

- (a) If the requested operation is Get or Get.int, it uses the second word as the address in the local SMEM and fetches two words in interlocked mode. Meanwhile, it transfers the first word in the receive queue to the first word in the send queue, modifying the Opcode from Get to Send. Following this first word, it moves the third word from the receive queue to the second word in the send queue. Once the two words are fetched from local SMEM, it places them in the 3rd and 4th words in the send queue.
- (b) If the requested operation is F&R, the second word in the receive queue is used as the address in the local SMEM to fetch the word there, and replace it with the value supplied in the first word of the receive queue. The Opcode part of the first word is changed to Send.int and put in the first word in the send queue. The third word from the receive queue is moved to the second word in the send queue and the fetched value from the local SMEM is placed in the third word. In order to reduce controller complexity, we chose to include an additional dummy fetch in the F&R operation so that same number of fetches are performed for all operations.

Having discussed the operation of this design we now discuss its performance.

Performance

The best case times for performing the five operations discussed in this section are calculated as follows.

The time for the first part, in which the source processor writes a request into the task queue is comprised of:

1. Check if the send queue is empty, in an interlocked mode. This takes five cycles.
2. Write three words in the task queue. This takes 12 cycles.
3. Release the SMEM bus. This takes four cycles.

Thus the total time for the first part is 21 cycles. With the same cycle time as before, 21 cycles will take

$$21 \times 62.5 = 1312.5 \text{ nS}$$

Or $1.31\mu\text{S}$. Note that with a zero wait state SMEM this would have taken six cycles or 375 nS.

Assuming that the port controller operates at the same rate as the C30s, it will take eight cycles to test and move three words from the task queue to the send queue for the Get, Get_int, or F&R operations (we assume two cycles for fetching and testing). Thus the time is

$$8 \times 62.5 = 500 \text{ nS}$$

It will take 14 cycles to fetch two words from the local SMEM and move two other words from the task queue to the send queue for the Send or Send_int operations. Thus the time is

$$14 \times 62.5 = 875 \text{ nS}$$

with a 3 wait state SMEM or

$$8 \times 62.5 = 500 \text{ nS}$$

with a zero wait state SMEM.

Moving a request from the send queue to the receive queue will take

$$\left(\frac{1}{2} \times 24 \times \frac{1}{32 \times 10^6} \right) = 375 \text{ nS}$$

for the three-word (24 nibbles) of a request or

$$\left(\frac{1}{2} \times 32 \times \frac{1}{32 \times 10^6} \right) = 500 \text{ nS}$$

for a four-word request.

For a port controller, the time for acting on a request in the receive queue is comprised of the following steps:

1. From the first word in the receive queue, find the operation requested. This takes 2 cycles.
2. If the requested operation is Send, it will take 12 cycles to put the two words in the SMEM.
3. If the requested operation is Send_int, it will take 16 cycles to put the two words in the SMEM and set the bit in the interrupt register.

4. Releasing the SMEM bus will take another 4 cycles.
5. If the requested operation is Get or Get.int, it will take 16 cycles with 3 wait state SMEM or 10 cycles with zero wait state SMEM to move the fetched word to the send queue and modify the Opcode.
6. If the requested operation is F&R, it will take 20 cycles with 3 wait state SMEM or 11 cycles with zero wait state SMEM.

The Get, Get.int and F&R operations are not complete yet. These three instructions must move the fetched data value to the originating SNODE and interrupt a processor if necessary. The times for these steps are as follows:

$$500 + 62.5(2 + 12 + 4) = 1625 \text{ nS}$$

or 1.63 μ S for a Get instruction using 3 wait state SMEM, or

$$500 + 62.5(2 + 6 + 1) = 1062.5 \text{ nS}$$

or 1.06 μ S using zero wait state SMEM;

$$500 + 62.5(2 + 16 + 4) = 1875 \text{ nS}$$

or 1.88 μ S for a Get.int or an F&R instruction using 3 wait state SMEM, or

$$500 + 62.5(2 + 7 + 1) = 1125 \text{ nS}$$

or 1.13 μ S using a zero wait state SMEM.

At this point all operations are over. With 3 wait state SMEM the total time for a Send operation will be

$$62.5(21 + 14) + 500 + 62.5(2 + 12 + 4) = 3812.5 \text{ nS}$$

or 3.81 μ S. With zero wait state SMEM a Send operation will take

$$62.5(6 + 8) + 500 + 62.5(2 + 6 + 1) = 1937.5 \text{ nS}$$

or 1.93 μ S.

With 3 wait state SMEM a Send_int operation will take

$$62.5(21 + 14) + 500 + 62.5(2 + 16 + 4) = 4062.5 \text{ nS}$$

or 4.06 μ S. With zero wait state SMEM a Send_int operation will take

$$62.5(6 + 8) + 500 + 62.5(2 + 7 + 1) = 2000 \text{ nS}$$

or 2.0 μ S.

With 3 wait state SMEM, the total time for a Get operation will be

$$62.5(21 + 14) + 375 + 62.5(2 + 16 + 4) + 500 + 62.5(2 + 12 + 4) = 5562.5 \text{ nS}$$

or 5.56 μ S. With zero wait state SMEM, a Get operation will take

$$62.5(6 + 8) + 375 + 62.5(2 + 10 + 1) + 500 + 62.5(2 + 6 + 1) = 3125 \text{ nS}$$

or 3.13 μ S.

With 3 wait state SMEM, the total time for a Get_int operation will be

$$62.5(21 + 14) + 375 + 62.5(2 + 16 + 4) + 500 + 62.5(2 + 16 + 4) = 5812.5 \text{ nS}$$

or 5.81 μ S. With zero wait state SMEM, a Get_int operation will take

$$62.5(6 + 8) + 375 + 62.5(2 + 10 + 1) + 500 + 62.5(2 + 7 + 1) = 3187.5 \text{ nS}$$

or 3.19 μ S.

With 3 wait state SMEM, the total time for a F&R operation will be

$$62.5(21 + 14) + 375 + 62.5(2 + 20 + 4) + 500 + 62.5(2 + 16 + 4) = 6062.5 \text{ nS}$$

or 6.06 μ S. With zero wait state SMEM, a F&R operation will take

$$62.5(6 + 8) + 375 + 62.5(2 + 11 + 1) + 500 + 62.5(2 + 7 + 1) = 3250 \text{ nS}$$

or 3.25 μ S.

Table 6.3. Best times for shared memory operations

Operation	3 wait state SMEM	zero wait state SMEM
Send	3.81 μ S	1.93 μ S
Send_int	4.04 μ S	2.00 μ S
Get	5.56 μ S	3.13 μ S
Get_int	5.81 μ S	3.19 μ S
F&R	6.06 μ S	3.25 μ S

The total times for all these operations are tabulated in table 6.2. Analysis of the average and worst case latency in these operations is subject to simulation using actual tasks on the ICAP¹¹.

Next we very briefly discuss issues related to building larger networks.

6.5 Larger networks

A huge body of literature exists that deals with shared memory systems. The design of these systems, in general, has been influenced by the application space as well as the architectural constraints. We feel that at this moment, exploring designs for larger networks for the ICAP is very speculative in nature, because the application space (parallel processing as related to intermediate-level vision) and the architectural constraints of the future generation IUAs are not well defined. Nonetheless, one of the following schemes might be used for building larger ICAP communication networks.

6.5.1 Based on point to point topology

By redesigning SNODEs, it is possible to have a 64-bit wide SMEM bus. By pipelining SMEM access and operating the SMEM bus at 10MHz, it is possible to achieve an effective throughput of 80 MB/S for each SNODE shared memory. This would allow increasing the number of processors on a SNODE from 8 to 16. The fan-in on the SMEM bus will not allow a substantially larger number of processors on an SNODE. In addition, access to the SMEM is serial, therefore, increasing the number of processor will increase the average latency.

¹¹An analysis could be developed using queuing theory, but such an analysis is of doubtful value without simulation data to validate it.

It was seen in sections 6.3 and 6.4 that the majority of the delay in message passing or shared memory access occurs inside SNODE. For example, with a zero wait state SMEM, a Send operation in section 6.4 took $1.93 \mu\text{S}$. Out of this time, only $0.5 \mu\text{S}$ was spent in actual message transmission between SNODEs. Therefore, by improved SNODE design and using 7-Dimensional hypercube topology with *virtual cut through* routing between PARCOS III+, it is possible to build a $2^7 = 128$ SNODE, or 2K processor ICAP system with latencies of the order of twice those in IUA GEN II+ prototype. It is feasible to incorporate 8 pairs of channels in each PARCOS III+, and thereby, support up to 4K processors in the ICAP.

6.5.2 Based on multiple buses

An outline of this scheme is shown in figure 6.15. Each PARCOS III+ on an SNODE is connected to multiple buses. This scheme offers potentially lower latencies than hypercube topology in sparse communication. An obvious extension to this scheme is by using hierarchical buses.

6.5.3 Hybrid scheme

An outline of this scheme is shown in figure 6.16. This scheme is advantageous if there is sufficient spatial locality in interprocessor communication. Clusters of SNODEs are connected via self-routing crossbar switches (PARCOS II can be used here). The same crossbar switches are also connected to one or more buses. SNODEs in the same cluster communicate via their respective crossbar switch. Inter-cluster SNODE communication takes place via one of the global buses. An obvious extension to this scheme is by using hierarchical crossbars instead of buses to connect the clusters.

6.6 Summary

This chapter dealt with two different issues. The first issue relates to parallel or distributed control of communication in message passing multiple-processor systems to support fine-grained MIMD tasks. In section 6.1 we presented the architecture of a building block PARCOS II chip that implements a self-routing crossbar network. Individual cells

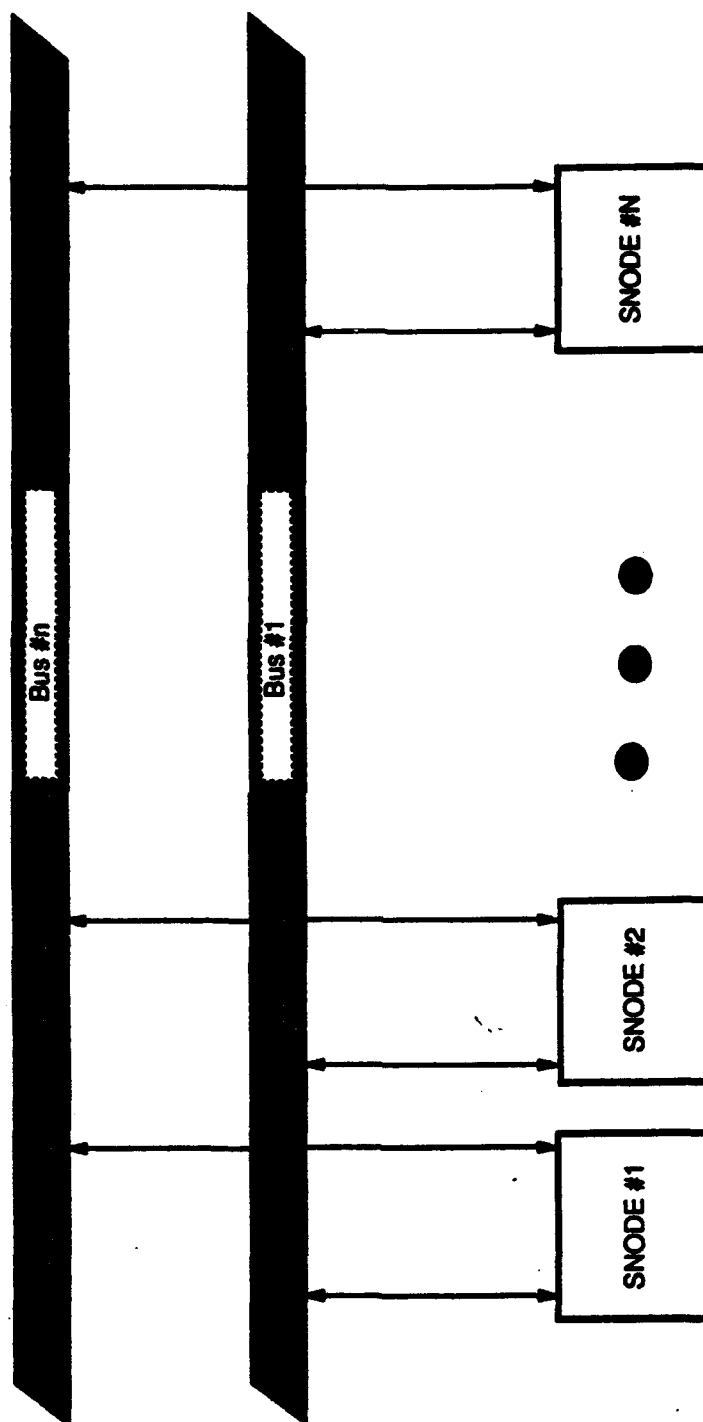


Figure 6.15. Multiple bus based scheme

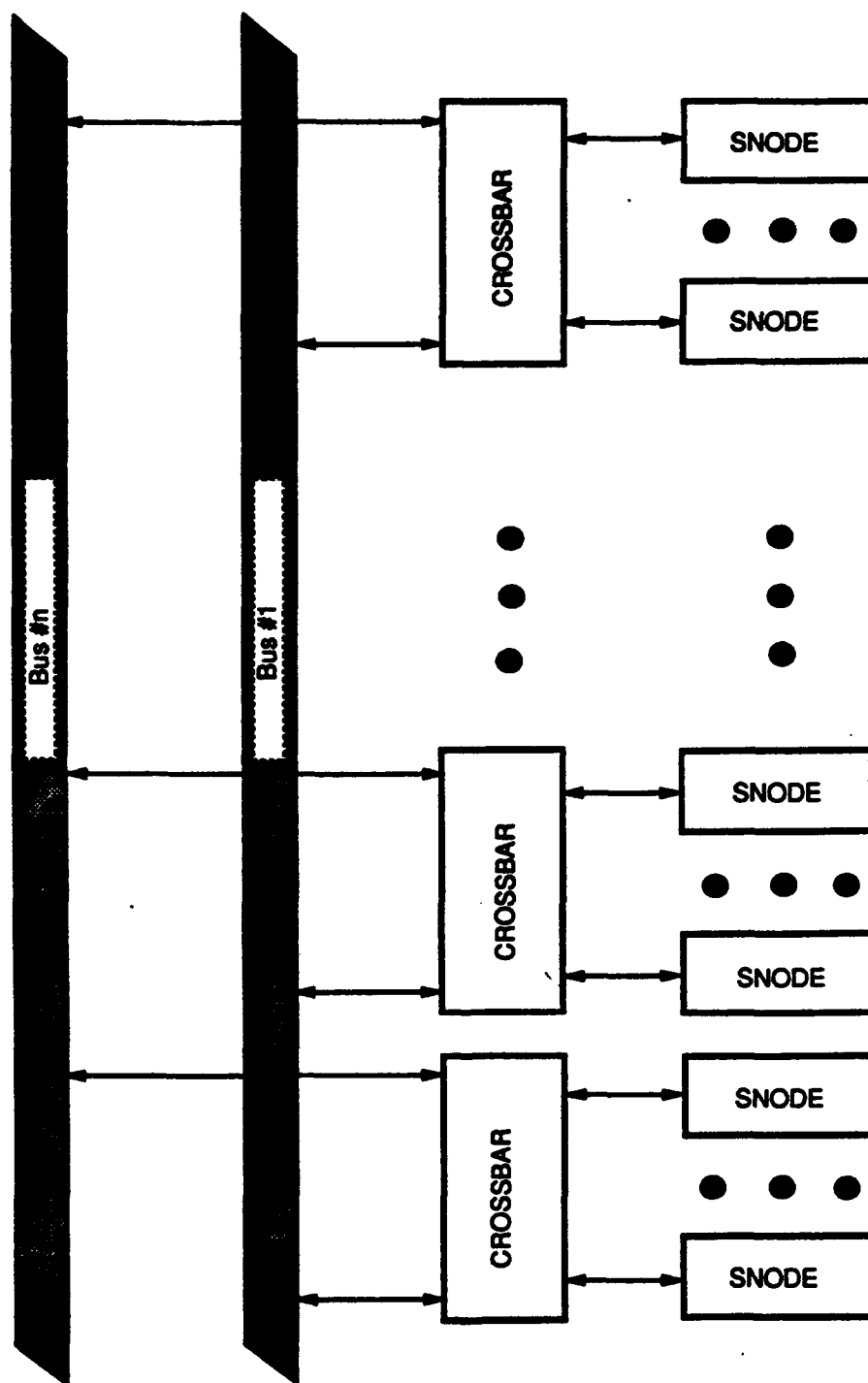


Figure 6.16. Hybrid scheme

from this chip can be used to build larger crossbar networks on a chip, or this chip can be used as a building block in constructing a variety of self-routing networks.

The second part of this chapter dealt with providing NUMA shared memory support at the ICAP level. During the course of this investigation, it was discovered that supporting a symbolic token (ISR) database at the ICAP level is a more fundamental requirement than supporting particular intermediate-level vision algorithms. It was also discovered that flexible communication and shared memory support are much more critical for supporting intermediate-level vision than providing a variety of fixed communication patterns. To this end, we developed an evolving series of ICAP communication network architectures in sections 6.3 through 6.5. We first elaborated the additional requirements and design constraints at the ICAP level that evolved during the course of this research. In section 6.3.3 we described Stage I design that is intended to be the backup solution if Stage II or Stage III design fails - it does not require any custom hardware and partially addresses the aforementioned requirements. In section 6.3.4 we described Stage II design that uses a custom VLSI PARCOS III chip and improves upon the performance of Stage I. In section 6.4 we described Stage III design that follows a different path from Stage II and provides mechanisms for NUMA shared memory. Finally in section 6.5, we discussed some schemes that might be used for building larger ICAP communication networks.

CHAPTER 7

RESULTS AND CONCLUSIONS

7.1 Summary of research

This thesis has addressed issues related to Processor-Processor or Processor-Memory communication in multiple-processor architectures. A central theme has been that the three main aspects of a multiple-processor system, its target application, its architecture, and its communication mechanisms greatly influence each other. Another theme has been that real-world problems are often comprised of multiple tasks with varying forms of parallelism (where we define the extremes to be fine-grained data parallelism and coarse-grained control parallelism). In order to be more effective, future multiple-processor architectures will have to be "flexible" in supporting multiple forms of parallelism. Machine vision in general, and intermediate-level vision in particular, is an excellent example for demonstrating multi-modal parallelism, which is chosen as the application domain for this thesis.

The specific problem addressed has been to determine the communication requirements of the intermediate level processors (ICAP) of the Image Understanding Architecture (IUA), explore the design space of potential solutions, develop a network design that meets the requirements, demonstrate the feasibility of constructing the design, and show both analytically and empirically that the design meets the requirements.

The approach has been to first investigate the computational characteristics of the vision tasks to be run at the ICAP level. The architectural (communication and control) requirements of the ICAP communication were extracted from the computational characteristics. The requirements were then divided into logical groups, and an evolving series of network architectures was developed that cumulatively supports or addresses these groups.

Chapter 2 provided an overview of parallel processing and interconnection networks. Various schemes have been proposed to classify computer architectures, but none of them covers the entire functional model of the ICAP level of the IUA. Therefore, a new terminology was defined for identifying a parallel processor such as the ICAP. A myriad of schemes have

been proposed for interconnection networks for PE - PE communication or for PE - Memory communication in parallel processing systems. An overview of the field of interconnection networks was provided. A combination of more than one of these schemes was used as the solution to the ICAP interprocessor communication in the following chapters.

Chapter 3 addressed the architectural characteristics and requirements of interprocessor communication at the ICAP level of the IUA, which in turn, were derived from the computational characteristics of the tasks to be run at the ICAP level. There are two problems with this approach. First, it is impossible to look at the tasks for the ICAP in isolation, because many of the tasks are ill-defined and transcend the processor boundaries of the CAAPP, ICAP, and SPA. Second, the IUA is primarily intended as a vision research tool and, therefore, different researchers may choose to divide and map the tasks differently onto different levels of the IUA. These two problems make it necessary to study the computational characteristics of the intermediate-level tasks in two stages: First, from the point of view of the well-established image interpretation tasks that have been identified in the VISIONS [Hanson 86] laboratory at the University of Massachusetts, and second, by using a representative sample from the literature where other vision researchers have utilized a wide range of techniques for image interpretation. Additionally, because of the non-unique mapping of intermediate-level vision tasks onto the ICAP level, they are viewed in the light of low- and high-level vision tasks.

The general communication characteristics and control requirements that directly affect the architecture of the ICAP communication network were found to be:

- Varying communication load
- Varying computational granularity
- Iterative processing with massive temporal parallelism (Cycle through different algorithms repeatedly, using different data sets)
- A mix of static (and known apriori), and dynamic (data-dependent) interprocessor communication, and
- A mix of local, and non-local interprocessor communication
- Centrally controlled SIMD
- Centrally controlled Synchronous-MIMD
- MIMD

Based on the architectural characteristics and requirements of the ICAP level to efficiently support intermediate-level vision, the architectural requirements of the ICAP communication network were defined and are again summarized here:

- It should have low latency, high bandwidth, and high common access throughput, especially in real time applications
- It should have the ability to support low-overhead SIMD-like synchronous routing, under central control
- In SIMD-like routing, it should be equally efficient in supporting both regular and irregular communication patterns. In other words, it should not have a bias towards one communication pattern over others
- It should have the ability to support data-dependent synchronous routing under the SMIMD mode of ICAP computation
- It should have the ability to support data-dependent asynchronous routing under the MIMD mode of ICAP computation, and
- Most importantly, it should have a capability for rapid reconfiguration to efficiently support all of the above requirements

These requirements were broken into logical groups and an evolving series of ICAP communication architectures were developed in the succeeding chapters that cumulatively support or address the requirements.

Chapter 4 discussed the first stage design, which is used when the ICAP is operated in SIMD-like manner, i.e. when the interprocessor communication is fixed and known apriori. A family of networks can be built using copies of the building-block custom VLSI PARCOS I chip. In addition to an $n \times n$ crossbar, the PARCOS I chip contains a control memory that allows the networks built with the chip to store a large number of network configurations and with a single instruction, the network configuration can be changed from one stored pattern to another. Any of the stored patterns is incrementally modifiable without interrupting processing, using an existing network configuration. This scheme eliminates header generation and routing (control) overheads in case of fixed apriori communication. The major conclusion of this chapter was that crossbar and other dense networks, such as Clos and Benes, are viable design alternatives, even for large scale multiple-processor systems. Such networks, in general, have previously been considered impractical to build.

Chapter 5 discussed second stage design, which addresses the requirements of the ICAP when the interprocessor communication is data-dependent and cannot be determined apriori. The ICAP can be operating in a Synchronous-MIMD (SMIMD) or a MIMD mode. This design is such that various networks can be built with simple custom hardware in addition to PARCOS I chips, to provide an interim solution to the problem of supporting data dependent interprocessor communication at the ICAP level of the IUA. The networks built using this design use central routing control. Even though the network controller, which is used for the routing control, is serial, by using a special hardware search memory it can achieve network set up times comparable to many parallel control schemes for non-blocking and rearrangeably non-blocking networks. Nonetheless, because of its serial nature, this design is suitable only for smaller networks. The major conclusion of this chapter was that central control is a viable solution for reasonably large network sizes, in supporting data dependent routing, which is contrary to conventional wisdom.

Chapter 6 dealt with two different issues. The first issue relates to parallel or distributed control of communication in message passing multiple-processor systems to support fine-grained MIMD tasks. The architecture of a building block PARCOS II chip that implements a self-routing crossbar network was presented. Individual cells from this chip can be used to build larger crossbar networks on a chip, or this chip can be used as a building block in constructing a variety of self-routing networks. The second part of this chapter dealt with providing Non-Uniform Memory Access (NUMA) time shared memory support at the ICAP level. During the course of this investigation, it was discovered that supporting a symbolic token (ISR) database at the ICAP level is a more fundamental requirement than supporting particular intermediate-level vision algorithms. It was also discovered that flexible communication and shared memory support are much more critical for supporting intermediate-level vision than providing a variety of fixed communication patterns. To this end, the additional requirements and design constraints at the ICAP level that evolved during the course of the research were elaborated, followed by the description of an evolving series of ICAP communication network architectures to address these requirements.

7.2 Future research

Because of its diverse nature, this research has many useful future implications in the areas of high performance systems and application specific parallel architectures and algorithms. The following is a brief discussion on each of them.

7.2.1 High performance systems

As a continuation of this research, the following three critical issues in high performance (multiple-processor) system architecture¹ need further attention:

Processing Element design

The design of the PE is most crucial to the *raw power* of a multiple-processor system. To this end, important concerns include the datapath width, memory address space, clock speed, I/O bandwidth, etc. The operation mode (SIMD, MIMD, etc.) of the multiprocessor influences the PE's instruction set, local control, communication interface, I/O subsystem, testing and diagnostics, etc. The following two areas deserve further attention.

Close coupling of computation and communication subsystems

Often, the communication mechanisms are considered separately from the design of the PEs in multiple-processor systems. In many cases this introduces inefficient synchronization and communication overhead in interprocessor communication. There is a need to explore PE designs that incorporate the computation and communication subsystems in a unified manner to alleviate the interprocessor communication problem. The ultimate goal is to achieve communication latencies comparable to memory access time and communication setup overheads comparable to memory access overheads.

Multi-mode capability

Adding capabilities to PEs to efficiently support different operation modes (for example, SIMD and Synchronous-MIMD) for multiple-processor systems is a necessity. Alternate approaches based on "multiprocessor-network" have been proposed to tackle the same problem, by connecting a number of specialized multiprocessors through a high bandwidth communication network. The individual multiprocessors are expected to tackle specific tasks of the bigger problem in the most efficient manner. Arguably, many of the themes in the research on "flexible-multiprocessors" (with multi-mode capability) and "multiprocessor-

¹These issues become even more complex if the target application requires a "flexible-multiprocessor" architecture.

network" based approaches are similar.

Multiple-processor communication networks

The topic of this thesis is communication networks for multiple-processor systems. With respect to this, the following two areas deserve further attention.

Easy reconfigurability

Completing the designs discussed in chapter 6 is one potential area of future research. For example, how to extend NUMA shared memory architecture for the ICAP to larger size machines.

Alternate technologies

Integrated optics might alleviate the bandwidth limitations and other problems associated with multicast operations. Also, this technology will be helpful in system clock distribution, because it has minimal skew and it does not suffer from the fan-out problem associated with conventional technologies. Additionally, wafer scale integration and similar technologies might alleviate some of the communication latency problem.

Control Mechanisms

Control mechanisms are crucial to the overall efficiency of a multiple-processor system, broadly these mechanisms cover the synchronization of various subsystems in a multiple-processor, and program flow control. They vary with the target application, operational mode(s), and the architecture of the multiple-processor system. The following two areas deserve further attention.

Associative processing techniques

The IUA has shown great effectiveness by using fine-grained control based on associative processing techniques. It was shown that the IUA feedback concentrator can be used effectively in the second message passing network design. Some other interesting issues are: the effectiveness of these techniques in other multiple-processor operation modes such

as Multi-SIMD, and their usefulness in general computing.

Control mechanisms for "flexible-multiprocessor" architectures

It is important to further explore control mechanisms for a multiple-processor architecture that would allow it to operate efficiently over a wide range of modes. The question to be addressed is, for example, whether it is possible to automatically switch multiple-processor control from local/global mode, to global/local mode, based on the requirements of a task, without a centralized control mechanism.

7.2.2 Application specific parallel architectures and algorithms

The architecture of a multiprocessor and its target application greatly influence each other. For example, an application may have specific requirements in terms of the programming language, data storage, I/O method and bandwidth, real-time constraints, and computational characteristics and requirements. These requirements put specific demands on the multiple-processor system design. On the other hand, a *specific* multiprocessor architecture essentially defines a set of capabilities and limitations. In order to design efficient parallel architectures that meet an application's performance requirements while also satisfying the design constraints of the application, it is imperative to better understand their interdependence.

A two pronged approach can be followed to tackle this problem. In the first part, one would investigate the interdependence of parallel algorithms and parallel architectures in a more general sense without regards to a specific application. In the second part, one would deal with specific applications.

Interdependence of parallel algorithms and parallel architectures

Complexity analysis of serial algorithms allows programmers to compare various algorithms, and predict their performance on actual machines. The analysis of parallel algorithms, however, is not a straight forward extension of the serial case. Unlike a uniprocessor, a parallel architecture greatly influences the design and analysis of parallel algorithms. Approaches based on abstract machine models such as *PRAM*, *CRCW*, *CREW*, etc., give only a crude estimate of the actual performance of a parallel algorithm. On the

other hand, designing optimized parallel algorithms for *particular* parallel architectures represents a huge investment, since it is not easy to port a parallel algorithm from one architecture to another. A better understanding of the interdependence of parallel algorithms and parallel architectures will lead to better algorithm and architecture designs. This research would involve experimentation, with the ultimate goal of carrying out qualitative and quantitative analysis of parallel algorithms from an architectural point of view. The following questions deserve specific attention.

Are there uniquely preferred architectures for specific algorithms/tasks

Another way of looking at this question is: What are the characteristic computational, communication, and control requirements of specific tasks, and what architectural features are required to best support these requirements? An alternate question is: What kinds of algorithms/tasks are best suited for a *specific* parallel architecture. For example, many data dependent computations may not be suitable for globally controlled SIMD architectures. On the other hand, many fine-grained *staged* computations may not be suitable for MIMD architectures with relatively large synchronization and communication overheads.

Programming environment and language

How does the programming environment and language influence the design and performance of a parallel architecture? This question is often considered separately from the actual tasks and algorithms that will run on a parallel architecture. In real world problems, the importance of this issue is often as great as the speedup a parallel architecture might offer. The question is of particular importance from a systems point of view, since the programming environment is the interface between a parallel algorithm and the architecture that it runs on. Every interface has potential overhead and thus it influences the efficiency of a parallel architecture.

7.3 Conclusions

Many existing multiple-processor architectures are designed to efficiently exploit parallelism in a specific narrow range (where the extremes are fine-grained data parallelism and coarse-grained control parallelism). Most real world problems are comprised of multiple tasks which vary in their range of parallelism. Extreme examples are found in AI problems

that deal with sensory data, such as speech, machine vision, and robotics. We argue that, in order to deal with real world and advanced research problems more efficiently, future multiple-processor architectures must be "flexible" in supporting parallelism over a wider range.

Communication is crucial to the overall performance of a multiple-processor system. To alleviate the bottleneck of interprocessor communication, a "good" interconnection network for PE-PE communication in a message passing system or for PE-Memory communication in a shared memory system is required. It is a very complex task to design a "good" interconnection network, since it depends on many factors that interact with each other to determine the performance of a multiple-processor system. A "good" interconnection network can be defined as one that meets the performance requirements of a given application while also satisfying its engineering constraints. That is, a measure of "goodness" can only be defined for an interconnection network in relation to the context of its use.

In order to run as an efficient integrated system, a communication network must be designed in close cooperation with the system architecture. In other words, the architecture of a communication network is intimately tied to the overall multiple-processor architecture (such as control, operation mode, etc.), it cannot be treated as just a means for creating links (with certain desirable properties) between processors, or processors to memory modules. The central theme and the main result of this thesis is that it is indeed feasible to design networks for the aforementioned "flexible" multiple-processor systems.

BIBLIOGRAPHY

- [Adams 82] George B. Adams and Howard J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers*, vol C-31, no 5, May 1982, pp. 443-545.
- [Akers 89] Sheldon B. Akers and B. Krishnamurthy, "A group theoretic model for symmetric interconnection networks," *IEEE Transactions on Computers*, Vol. 38, No 4, April 1989, pp 555-566.
- [Al-Sadoun 85] H.B. Al-Sadoun et al, "Interconnecting off-the-shelf microprocessors," *1985 National computer conference, AFIPS*, pp 175-181.
- [Andersen 77] Steiner Anderson, "The looping algorithm extended to base 2^k rearrangeable switching networks," *IEEE Transactions on Communications*, vol COM-25, no 10, October 1977, pp 1061-1063.
- [Annaratone 87] Mario Annaratone, et al, "The WARP computer: Architecture, Implementation, and performance," *IEEE Transactions on Computers*, Vol. C-36, No. 12, December 1987, pp 1523-1538.
- [Athas 88] William C. Athas and Charles L. Seitz, "Multicomputers: Message-passing concurrent computers," *IEEE Computer*, Aug 1988, pp 9-24.
- [Bakoglu 90] H.B. Bakoglu, *Circuits, Interconnections and Packaging for VLSI*, Addison-Wesley Publishing Company, 1990.
- [Ballard 82] D. Ballard and C Brown, *Computer Vision*, Prentice Hall Inc, Englewood Cliffs, NJ, 1982.
- [Barber 88] F.E. Barber et al, "A 64 x 17 Non-blocking crosspoint switch," *1980 IEEE International Solid-State circuits conference*, pp 116-117, 322.
- [Bassalygo 73] L.A. Bassalygo and M.S. Pinsker, "On the Complexity of Non-Blocking Switching Networks without Rearrangement," in *Problems in Information Transmission*, Plenum Publishing Corporation, New York, 1973, pp 84-87.
- [Batcher 68] Kenneth E. Batcher, "Sorting networks and their applications," *Spring Joint Computer Conference, AFIPS*, 1968, pp 307-314.
- [Batcher 76] Kenneth E. Batcher, "The Flip network in STARAN," *International Conference on Parallel Processing*, Aug 1976, pp 65-71.
- [Batcher 77-1] Kenneth E. Batcher, "STARAN series E," *International Conference on Parallel Processing*, 1977, pp 140-143.
- [Batcher 77-2] Kenneth E. Batcher, "The Multi-Dimensional-Access Memory in STARAN," *IEEE Transactions on Computers*, vol C-26, Feb 1977, pp 174-177.

- [Batcher 80] Kenneth E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol. C-29, No. 9, Sept 1980, pp 836-840.
- [Beetem 85] John Beetem, Monty Denneau, and Don Weingarten, "The GF11 Supercomputer," *International Symposium on Computer Architecture*, 1985, pp 108-115.
- [Benes 62-1] V.E. Benes, "Algebraic and topological properties of connecting networks," *Bell Systems Technical Journal*, July 1962, pp 1249-1274.
- [Benes 62-2] V.E. Benes, "On rearrangeable three-stage connecting networks," *Bell Systems Technical Journal*, Vol XLI, No. 5, Sept 1962, pp 1481-1492.
- [Benes 65] V.E. Benes, *Mathematical theory of connection networks and telephone traffic*, Academic Press, New York, 1965
- [Bently 79] J.L. Bently, "Decomposable searching problems," *Information processing letters*, Vol. 8, No. 5, June 1979, pp 244-251.
- [Berg 72] R.O. Berg et al, "PEPE - An overveiw of architechture operation and implemen-tation," *Proceedings of the National Elect Conference*, 1972, pp 312-317.
- [Berman 83] Francine Berman , "Parallel computations with limited resources," *Conference on Information Sciences and Systems*, John Hopkins University, 1983.
- [Beveridge 89] J. Ross Beveridge, et al, "Segmenting Images Using Localized Histograms and Region Merging," *International Journal of Computer Vision*, Vol 2, 1989, pp 31-347.
- [Blodgett 82] A.J. Blodgett and D.R. Barbour, "Thermal Conduction Module: A high Per-formance Multiplayer Ceramic Package," *IBM Journal of Research and Development*, vol. 26, no 1, Jan 1982, pp. 30-36.
- [Bode 85] A. Bode et al, "Multi-Grid oriented computer Architecture," *International Con-ference on parallel processsing*, 1985, pp 89-95.
- [Bokhari 84] Shahid .H. Bokhari, "Finding maximum on an array processor with a global bus," *IEEE Transactions on Computer*, Vol. C-33, No 2, Feb 1984, pp 133-139.
- [Boldt 87] Michael Boldt and R. Weiss, "Token based extraction of straight lines," *COINS Tech Report 87-104, Computer and Information Science Dept., University of Mas-sachusetts*, Amherst, MA 01003, Oct 1987.
- [Bouknight 72] W.J. Bouknight et al , "The Illiac IV System," *Proceedings of the IEEE* Vol 60 No 4, April 1972, pp 369-388.
- [Briggs 79] F.A. Briggs, et al, "PM4 - A Reconfigurable Multiprocessor system for Pattern Recognition and Image Processing," *AFIPS National Computer Conference*, 1979, pp. 255-266.
- [Brolio 89] John Brolio, et al, "ISR: A Database for Symbolic Processing in Computyer Vision," *IEEE Computer*, December 1989, pp 22-30.
- [Broomell 83] G. Broomell and J.A. Heath, "Classification categories and historic develop-ment of circuit switched topologies," *ACM Computing Surveys*, Vol. 15, No. 2, June 1983, pp 95-133.
- [Burns 86] J. Brian Burns, Allen Hanson, and Edward Riseman, "Extracting Straight Lines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol PAMI-8, no 4, July 1986, pp 425-455

- [Cantor 71] D.G. Cantor, "On Non-Blocking Switching Networks," *Networks*, vol 1, no 4, 1971, pp. 367-377.
- [Carpinelli 87] J.D. Carpinelli and A. Yuvuz Oruc, "Paralle setup algorithms for Clos networks using a Tree-Connected computer," *Second International Conference on Supercomputing*, 1987, pp 321-327.
- [Cattermole 77] K.W. Cattermole and J.P. Summer, "Communication networks based on the product graph," *Proceedings of the Institution of Electrical Engineers*, Vol 124, Jan 1977, pp 38-48.
- [Cavanagh 84] Joseph J. F. Cavanagh, *Digital Computer Arithmetic*, McGraw Hill, New York, 1984.
- [Chen 80] T.C. Chen, "Overlap and pipeline processing," in *Introduction to Computer Architecture*, H.S. Stone (Ed), SRA 1980.
- [Chow 80] Yuan-Chieh Chow, R.D. Dixon, Tse-Yun Feng, and Chuan-Lin Wu, "Routing techniques for rearrangeable interconnection networks," In *Proc. Workshop on Interconnection networks for parallel and distributed processing*, H.J. Siegel (Ed), 1980, pp 64-69
- [Chu 89] Jeff Chu and Geog Schnitger, "The communication compexity of several problems in matrix computation," *Symposium on Parallel Algorithms and Architectures*, 1989, pp 22-31.
- [Clos 53] Charles Clos, "A Study of Non-Blocking Switching Networks," *Bell Systems Technical Journal*, vol 32, no 2, March 1953, pp 406-424.
- [CM-2 87] Thinking Machines Corporation, Connection Machine Model CM-2 technical summary, *Thinking Machines Tech rep HA 87-4*, Cambridge, MA, April 1987
- [Cuny 84] J.E. Cuny and L. Snyder , "Testing the coordination predicate," *IEEE Transactions on Computer*, vol C-33, No 3, March 1984, pp 201-208.
- [Cvetanovic 87] Zarka Cvetanovic, "The effects of Problem Partitioning, Allocation, and Granularity on the performance of Multiple-Processor Systems," *IEEE Transactions on Computers*, vol C-36, No 4 April 87, pp 421-432.
- [Dally 86] William J. Dally and Charles L. Seitz, "The Torus routing chip," *Distributed Computing*, Springer Verlag, 1986, No 1, pp 187-196.
- [Dally 87] William J. Dally and Charles L. Seitz, "Deadlock-Free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, vol. c-36, No 5, May 1987, pp 547 - 553.
- [Davidson 82] E.E. Davidson, "Electrical Design of a High Speed Computer Package," *IBM Journal of Research & Development*, Vol 26, No 3, May 1982, pp 349-361.
- [Dekel 81] E. Dekel, David Nassimi, and Sartaj Sahni, "Parallel Matrix and Graph Algorithms," *SIAM Journal of Computing*, Vol 10, no 4, Nov 1981, pp 387-403.
- [Dolan 89] John Dolan and Rich Weiss, "Perceptual Grouping of Curved Lines," *Proceedings of DARPA Image Understanding Workshop*, Palo Alto, CA, 1989.
- [Draper 89] B.A. Draper, R.T. Collins, J. Brolio, J. Griffith, A.R. Hanson, and E.M. Riseman, "The Schema System," *International Journal of Computer Vision*, Vol 2, March 1989, pp 209-250.

- [Draper 90] B.A. Draper, et al, "ISR2 User's Guide," *COINS Tech Report, 90-, Computer and Information Science Dept, University of Massachusetts, Amherst, MA 01003.*
- [Duff 78] M.J.B. Duff, "Review of the CLIP image processing system," *Proceedings of the National Computer Conference, AFIPS, 1978, PP 1055-1060.*
- [Duff 86] M.J.B. Duff (Ed), *Intermediate-Level Image Processing*, Academic Press, New York, 1986.
- [Favor 64] J. Favor, "A method for obtaining the exact count of responses using Full and Half adders," *AP-111770, Goodyear Aerospace Corporation, Akron, Ohio, Oct 1964.*
- [Feng 72] T.Y. Feng , "Some characteristics of Associative/Parallel Processing," *proceedings 1972 Sagamore computer conference, Syracuse University 1972, pp 5-16.*
- [Feng 74] T.Y. Feng, "Data Manipulating functions in parallel processors and their implementations," *IEEE Transactions on Computers*, vol C-23, March 1974, pp 309 - 318.
- [Finkel 87] Raphael A. Finkel, "Large-garin parallelism - Three case studies," in *The characteristics of parallel algorithms*, L. Jamieson et al (Eds), MIT press, Cambridge, MA 1987.
- [Flynn 66] Michael J. Flynn , "Very high-speed computing system," *Proceedings of the IEEE* vol 54, 1966, pp 1901-1909.
- [Foster 71] C. C. Foster, "Counting responders in an Associative Memory," *IEEE Trans. on Computers*, Dec 1971, pp 1580 - 1583.
- [Foster 76] C. C. Foster, *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, New York, 1976.
- [Gajski 83] Daniel Gajski et al, "Cedar," in *Tutorial on supercomputers: Design and applications*, Kai Hwang (Ed), IEEE Press, pp 251-275.
- [Gannon 84] Dannis G. Gannon and J.V. Rosendale, "On the impact of communication complexity on the design of parallel numeric algorithms," *IEEE Transactions on Computers*, Dec 1984, pp 1180-1194.
- [Gentleman 78] W.M. Gentleman, "Some complexity results for matrix computation on parallel processors," *Journal of the ACM*, Jan 1978, pp 112-115.
- [Goke 73] L.R. Goke and G.J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *First Annual symposium on computer architecture*, 1973, pp 21-28.
- [Gottlieb 83] A. Gottlieb et al, "The NYU ultracomputer - Designing an MIMD shared memory parallel computer," *IEEE Transactions on computers*, vol C-32 No 2, Feb 1983, pp 175-189.
- [Grondalski 87] Robert Grondalski, "A VLSI chip set for a massively parallel Architecture," *IEEE International Solid-State Circuits Conference*, 1987, PP 198-199.
- [Growther 85] W. Growther et al , "Performance measurements on a 128-node butterfly parallel processor," *International conference on Parallel Processing*, 1985, pp 531 - 540.
- [Handler 77] W. Handler , "The impact of classification schemes on computer architecture," *International Conference on Parallel Processing*, 1977, pp 7-15.

- [Hanson 78] A.R. Hanson and E.M. Riseman, *Computer Vision Systems*, Academic Press, New York, 1978.
- [Hanson 80] Allen R. Hanson and Edward M. Riseman, "Processing cones: a computational structure for scene analysis," In *Structured Computer Vision*, S. Tamimoto and A. Klinger (Eds), *Academic Press, New York 1980*.
- [Hanson 86] Allen R. Hanson and Edward M. Riseman, "The VISIONS Image Understanding System," *COINS Tech Report, 86-62, Computer and Information Science Dept, University of Massachusetts, Amherst, MA 01003*.
- [Hanson 87-1] Allen R. Hanson and Edward M. Riseman, "Summary of progress in image understanding research at the University of Massachusetts," *COINS Tech Report, 87-20, Computer and Information Science Dept, University of Massachusetts, Amherst, MA 01003*.
- [Hanson 87-2] Allen R. Hanson and Edward M. Riseman, "From image measurements to object hypothesis," *COINS Tech Report, 87-129, Computer and Information Science Dept, University of Massachusetts, Amherst, MA 01003*.
- [Hays 86] John P.Hays, Traver Mudge and Quinton F. Stout, "A microprocessor-based hypercube supercomputer," *IEEE MICRO*, October 1986, pp 6-17.
- [Hedlund 82] K.S. Hedlund and L. Snyder, "Wafer scale integration of Configurable, Highly parallel processors," *International Conference on Parallel processing*, 1982, pp 262-264.
- [Herbordt 90] Martin C. Herbordt, Charles C. Weems, and James C. Corbett, "Message-Passing algorithms for a SIMD Torus with Coteries," *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, July 1990.
- [Hill 86] Mark Hill et al, "Design Decisions in SPUR," *IEEE computer*, vol 19, No 11, Nov 1986, pp 8-22.
- [Hillis 85] W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Massachusetts, 1985.
- [Hockney 85] R. W. Hockney, "MIMD computing in the USA - 1984," *Parallel Computing*, North Holland, vol 2, 1985, pp 119-136.
- [Holland 59] John Holland, "A universal computer capable of executing an arbitrary number of sub-programs simultaneously," *Eastern Joint Computer Conference AFIPS*, 1959, pp 108-113.
- [Horn 86] B.K. Horn, *Robot Vision*, MIT Press, Cambridge, MA, 1986.
- [Horowitz 81] Ellis Horowitz and A. Zorat, "The binary tree as an interconnection network: Applications to multiprocessor systems and VLSI," *IEEE Transactions on Computers*, Vol C-30, No. 4, April 1981, pp 247-253. Also in *Proceedings, workshop on interconnection networks for parallel and distributed processing*, April 21-22, Purdue University, 1980, pp 1-10.
- [Hunt 81] D.J. Hunt, "The ICL DAP and its applications to image processing," In *languages and architectures for image processors*. M. J. B. Duff, S. Levialdi (eds). *Academic Press: London 1981*.
- [Huntz 72] R.G. Huntz, and D.P. Tate, "Control Data STAR-100 Processor design," *Proc COMPCON*, Fall 1972, pp 1-4.

- [Hwang 79] Kai Hwang, *Computer Arithmetic: Principles, Architecture and Design*, Wiley, New York, 1979.
- [Hwang 84] Kai Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill, New York, 1984.
- [Jensen 78] E. Jensen, "The Honeywell experimental distributed processor - An overview," *IEEE Computer*, vol 11, pp 28-38, Jan 1978.
- [Kautz 68] W.H. Kautz, et al, "Cellular Interconnection Arrays," *IEEE Transactions on Computers*, vol C-17 No 5, May 1968, pp 443-451.
- [Kermani 79] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, North Holland Publishing Company, vol 3, 1979, pp 267-286.
- [Kogge 81] P.M. Kogge, *The architecture of pipeline computers*, McGraw-Hill, New York 1981.
- [Kohl 87] C.A. Kohl, A.R. Hanson, and E.M. Riseman, "A Goal-Directed Intermediate-Level executive for Image Understanding," *Proceedings of the International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987, pp 811-814
- [Kuck 82] David J. Kuck and R.A. Stokes , "The Burroughs scientific processor (BSP)," *IEEE Transactions on Computers*, May 1982, pp 363-376.
- [Kuck 86] David J. Kuck et al, "Parallel supercomputing today and the Cedar Approach," *Science*, vol 231, Feb 1986, pp 967-974.
- [Kuhn 80] Robert Kuhn, "Transforming algorithms for single stage VLSI Structures," *Proceedings of Workshop on Interconnection networks for parallel and distributed processing*, H.J. Siegel (Ed), 1980, pp 11-19.
- [Kumar 87] V.K.P. Kumar and C.S. Raghavendra, "Array processors with multiple broadcasting," *Journal of parallel and distributed computing*, Vol 4, 1987, pp 173-190.
- [Kumar 88] Manoj Kumar, "Supporting Broadcast connections in Benes Network," *IBM Research Division, Tech Report RC 14063*, T.J. Watson Research Center, Yorktown Heights, NY 10598, Oct 1988.
- [Kung 80] H.T. Kung, "The structure of parallel algorithms," in *Advances in computers*, Vol 19, Academic Press, 1980, pp 65-112.
- [Kung 83] H.T. Kung and C.E. Lieserson, "Systolic arrays for VLSI," *15th ACM Symp on theory of computing*, 1983.
- [Lang 76-1] Thomas Lang and Harold S. Stone, "A shuffle-exchange network with simplified control," *IEEE Transactions on Computers*, vol c-25, No 1, Jan 1976, pp 55-66.
- [Lang 76-2] Thomas Lang, "Interconnection between processing and memory modules using the Shuffle-Exchange network," *IEEE Transactions on Computers*, Vol C-25, May 1976, pp 496-503.
- [Lawrie 75] D.H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, Vol C-24, No 12, Dec 1975, pp 175-189.
- [LeBlanc 88] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *International Symposium on Computer Architecture*, 1988, pp 161-172.

- [Lee 84] C.M. Lee and H. Soukup, "An algorithm for CMOS timing and area optimization," *IEEE Journal of Solid State Circuits*, vol SC-19, Oct 1984, pp 781-787.
- [Lee 85] K.Y. Lee, "On the rearrangeability of $2(\log_2 N - 1)$ stage permutation networks," *IEEE Transactions on Computers*, vol c-34, May 1985, pp 412-425.
- [Lee 87] K.Y. Lee, "A new Benes network control algorithm," *IEEE Transactions on Computers*, vol c-36, no 6, June 1987, pp 768-772
- [Lehrer 87] Nancy Lehrer, George Reynolds, and Joey Griffith, "A method for initial hypothesis formation in image understanding," *COINS Tech Rep 87-04, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, 1987.*
- [Leighton 84] F.T. Leighton, "Parallel computation using meshes of trees," *1983 workshop on Graph-Theoretical concepts in computer science*, Trauner Verlag, Linz, pp 200-218.
- [Levitan 87] Steven P. Levitan, "Measuring communication structures in parallel architectures and algorithms," in *The characteristics of parallel algorithms*, L. Jamieson et al (Eds), MIT press, Cambridge, MA 1987.
- [Lewis 84] E.T. Lewis, "Optimization of device area and overall delay for CMOS VLSI design," *Proceedings of the IEEE*, vol 72, June 1984, pp 670-689.
- [Li 87] Hungwen Li and M. Meresco, "Polymorphic-Torus: A new architecture for vision computer," *IEEE workshop on computer architectures for pattern analysis and machine intelligence*, 1987, pp 176-183.
- [Li 89] Hungwen Li and M. Meresco, "Polymorphic-Torus Architecture for computer vision," *IEEE Transaction on PAMI*, vol 11, No 3, March 1989, pp 233-243.
- [Lint 81] B. Lint and T. Agerwala, "Communication issues in the design and analysis of parallel algorithms," *IEEE Transactions on Software Engineering*, March 1981, pp 174-188.
- [Liu 68] C.L. Liu, *Introduction to Combinatorial Mathematics*, McGraw Hill, New York, 1968.
- [Markus 77] M.J. Markus, "The theory of connecting networks and their complexity: A review," *Proceedings of the IEEE*, vol 65, No 9, Sept 1977, pp 1263-1271.
- [Marr 82] David Marr, *Vision*, W.H. Freeman, San Francisco, CA, 1982.
- [MasPar 90] MasPar Computer Corporation, "MP-1 Hardware Manual," 1990.
- [Masson 71] G.M. Masson and B.W. Jordan, "Realization of a class of multiple connection assignments with asymmetric three stage networks," *Proc Fifth Annual Princeton Conference on Information sciences and systems*, 1971
- [Masson 72] G.M. Masson and B.W. Jordan, "Generalized Multi-Stage Connection Networks," *Networks*, Vol 2, 1972, pp 191-209
- [Masson 76] G.M. Masson, "On Rearrangeable and non-blocking switching networks," *Proceedings of 1976, International Computer Communication Conference Record*, also in "Tutorial: Distributed Processor Communication Architecture," K.J. Thurber (Ed), IEEE Computer Society.

- [McMillen 83] Robert J. McMillen and Howard J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Transactions on Computers*, vol C-31, Dec 1982, pp 1202-1214.
- [McMillen 84] Robert J. McMillen, "A survey of interconnection networks," *IEEE GLOBE-COM*, 1984, pp 105-113.
- [Mead 82] Carver Mead and M. Rem, "Minimum propagation delays in VLSI," *IEEE Journal of Solid-State Circuits*, vol SC-17, Aug 82, pp. 773-775.
- [Memmi 82] Gerard Memmi and Yves Raillard, "Some new results about the (d, k) graph problem," *IEEE Transactions on Computers*, Vol C-31, no 8, Aug 1982, pp 784-791.
- [Miller 87] R. Miller et al , "Parallel computations on reconfigurable meshes," *Tech Rep IRIS No 229, Dept of EE - Systems and Institute for Robotics and Intelligent Systems, University of Souther California*, Los Angeles, CA, March 1987.
- [Monier 85] L. Monier and P. Sindhu, "The architecture of the Dragon," *Proceedings 30th IEEE Computer Society International Conference*, 1985, pp 118-121.
- [Moshen 79] A.M. Moshen and C.A. Mead, "Delay-Time optimization for driving and sensing of signals on high-capacitance paths of VLSI systems," *IEEE Journal of Solid-State Circuits*, vol SC-14, April 1979, pp 462-470.
- [Mukherjee 86] Amar Mukherjee, *Introduction to nMOS & CMOS VLSI System Design*, Prentice Hall, 1986.
- [Nassimi 82] David Nassimi and Sartaj Sahni, "Parallel algorithms to setup the Benes permutation network," *IEEE Transactions on Computers*, vol c-31, no 2, Feb 1982, pp 148-154.
- [Nath.83] D.D. Nath, S.N. Maheshwari and P.C.P. Bhatt, "Efficient VLSI networks for parallel processing based on orthogonal trees," *IEEE Transactions on Computers*, Vol C-32, No 6, June 1983, pp 569-581.
- [Opferman 70] D.G. Opferman and N.T. Tsao-Wu, "On a class of rearrangeable networks, Part I: Control algorithms," *Bell Systems Technical Journal*, vol 50, no 5, May-June 1971, pp 1579-1600
- [Overton 79] K. Overton and T.E. Weymouth, "A Noise Reducing Preprocessing Algorithm," *Proceedings of IEEE Conference on Pattern Recognition and Image Processing, Chicago, IL*, 1979, pp 498-507.
- [Papamichalis 88] Panos Papamichalis and Ray Simar, "The TMS320C30 Floating-Point Digital Signal Processor," *IEEE MICRO*, December 1988, pp 13-29.
- [Parker 82] D.S. Parker, and C.S. Raghavendra, "The Gamma network: A multiprocessor network with redundant paths," *9th Annual Symposium on Computer Architecture*, 1982, pp 73-80.
- [Patel 79] Janak H. Patel, "Processor-memory interconnections for multiprocessors," *6th Annual Symposium on Computer Architecture*, 1979, pp 168-177.
- [Patel 81] Janak H. Patel, "Performance of Processor-Memory interconnections for multiprocessors," *IEEE Transactions on Computers*, Vol C-30, No 10, Oct 1981, pp 771-780.
- [Payne 86] William A. Payne, "Complexity and performance of statistically switched interconnection networks," *PhD dissertation, Illinois Institute of Technology*, May 1986.

- [Pease 68] Marshall C. Pease, "An adaption of the Fast Fourier Transform for parallel processing," *Journal of ACM*, vol 15, pp 252-264, April 1968.
- [Pease 77] Marshall C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers*, Vol C-26, No 5, May 1977, pp 458-473.
- [Perron 86] R. Perron, "The architecture of the Alliant FX/8 computer," *Proceedings COMPCON*, May 1986, pp 390-396.
- [Pfister 85] G. Pfister et al, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *International conference on Parallel Processing*, 1985, pp 764-771.
- [Pippenger 78] Nicholas Pippenger, "On rearrangeable and Non-Blocking Switching Networks," *Journal of Computer and System Science*, vol 17, no 4, Sept 1978, pp. 145 - 162.
- [Preparata 81] Franko P. Preparata and J.E. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation," *Communications of the ACM* May 1981, pp 300-309.
- [Preston 89] Kendall Preston, "The Abingdon Cross Benchmark Survey," *IEEE Computer*, July 1989, pp 9-18.
- [Ramanujam 73] H.R. Ramanujam, "Decomposition of permutation networks," *IEEE Transactions on Computers*, vol c-22 no 7, July 1973, pp 639-643
- [Rana 88] Deepak Rana, Charles C. Weems, and Steven P. Levitan, "An easily reconfigurable circuit switched connection network," *IEEE International Symposium on Circuits and Systems*, June 1988, pp 247-250.
- [Rana 89] Deepak Rana, and Charles C. Weems, "The ICAP parallel processor communications network," *IEEE International Symposium on Circuits and Systems*, June 1989, pp 126-129.
- [Rana 90a] D. Rana, and C.C. Weems, "The IUA Feedback Concentrator," *Proc 1990 IEEE International Conference on Pattern Recognition*, Atlantic City, New Jersey, June 1990.
- [Rana 90b] D. Rana, and C.C. Weems, "A Feedback Concentrator for the Image Understanding Architecture," *Proc 1990 Application Specific Array Processors*, Princeton, New Jersey, July 1990.
- [Rattner 85] Justin Rattner , "Concurrent processing: A new direction in scientific computing," *National Computer Conference, AFIPS*, 1985.
- [Reynolds 87] G. Reynolds and R. Beveridge, "Searching for geometric Structures in Images of Natural Scenes," *Proceedings of the DARPA Image Understanding Workshop*, Los Angeles, CA, Jan 1987
- [Rosenfeld 82] Azriel Rosenfeld and Avinash Kak, *Digital Picture Processing*, Academic Press, New York, 1982.
- [Rosenfeld 86] Azriel Rosenfeld, "The prism machine : an alternative to the pyramid," *Journal of Parallel and Distributed computing*, Vol 3, pp 404-411.
- [Russell 78] R.M. Russell, "The Cray-1 computer system," *Communications of the ACM*, Jan 1978, pp 63-72.

- [Samatham 89] M.R. Samatham, and D.K. Pradhan, "The De-Bruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI," *IEEE Transactions on Computers*, vol. C-38, no 4, April 1989, pp 567-581.
- [Sasaki 88] Katsuro Sasaki, et al, "A 15nS 1Mb CMOS SRAM," *IEEE International Solid State Circuits Conference*, 1988.
- [Schanin 86] D. Schanin, "The design and development of a very high speed system bus - The Encore Multimax Nanobus," *Proceedings Fall Joint Computer Conference*, Nov 1986, pp 410-418.
- [Schwartz 80] Jack T. Schwartz, "Ultracomputer," *ACM Transactions on Programming languages and systems*, vol 2 No 4, Oct 1980, pp 484-521.
- [Sedgewick 88] Robert Sedgewick, "Algorithms," *Addison Wesley*, Reading, Massachusetts, 1988.
- [Seitz 85] Charles L. Seitz, "The Cosmic Cube," *Communication of the ACM*, Vol 28, No 1, Jan 1985, pp 22-33.
- [Sejnowski 80] M.C. Sejnowski et al, "An overview of the Texas Reconfigurable Array Computer," *National Computer Conference, AFIPS*, 1980, pp 631-641.
- [Sequin 81] Carlo H. Sequin, "Doubly twisted torus network for VLSI processor arrays," *Proc 8th annual International Symposium on Computer Architecture*, 1981, pp 471-480.
- [Sharma] M. Sharma, et al, "NETRA: An architecture for a large scale multiprocessor vision system," *Workshop on computer architecture for pattern analysis and image database management*, Miami Beach, Florida, Nov 1985, pp. 92-98.
- [Shaw] David. E. Shaw, "NON-VON's applicability to three AI task areas," *International Joint conference on Artificial Intelligence*, August 1985, pp 61-72.
- [Shin 88] H.J. Shin and D.A. Hodger, "A 250 Mb/s crosspoint switch," *1988 IEEE International Solid-State circuits conference*, pp 114 - 115, 321.
- [Shoji 88] Masakazu Shoji, *CMOS Digital Circuit Technology*, Prentice Hall, New Jersey, 1988.
- [Shooman 60] W. Shooman, "Parallel computing with vertical data," *Eastern Joint Computer Conference, AFIPS*, 1960, pp 108-113.
- [Siegel 78] Howard J. Siegel and S.D. Smith, "Study of multistage SIMD interconnection networks," *5th Annual Symposium on Computer Architecture*, 1978, pp 223-229.
- [Siegel 81] Howard J. Siegel and Robert J. McMillen, "The multistage cube: a versatile interconversion network," *IEEE Computer*, vol 14, Dec 1981, pp 65-76.
- [Siegel 85] Howard J. Siegel, "Interconnection Networks for large-scale parallel processing," *Lexington Books*, 1985.
- [Slotnick 62] D.L. Slotnick et al, "The Solomon computer," *National Computer Conference, AFIPS*, 1962, pp 97-107.
- [Snyder 82] Lawrance Snyder, "Introduction to Configurable Highly Parallel Computers," *IEEE Computer*, Jan 1982, pp 47-56.
- [Snyder 84] Lawrance Snyder, "Parallel programming and the Poker programming environment," *IEEE Computer*, July 1984, pp 27-36.

- [Srini 85] Vason P. Srini, "An architecture for doing concurrrent systems research," *National Computer Conference, AFIPS*, 1985, pp 267-277.
- [Stenstrom 88] P. Stenstrom, "Reducing contention in shared-memory Multiprocessors," *IEEE Computer*, Nov 1988, pp 26-37.
- [Stone 71] Harold S. Stone , "Parallel processing with perfect shuffle," *IEEE Transactions on Computers*, Vol C-20, No 2, Feb 1971, pp 153-161.
- [Stone 80] Harold S. Stone , "Parallel computers," in *introduction to computer architecture*, H.S. Stone (Ed), SRA 1980.
- [Stone 87] Harold S. Stone, *High-Performance Computer Architecture* Addison-Wesley Publishing Company, 1987.
- [Stout 83] Quinton F. Stout, "Mesh connected computers with broadcasting," *IEEE Transactions on Computers*, Vol C-32, 1983, pp 826-830.
- [Stout 86] Quinton F. Stout, "Meshes with multiple buses," *26th IEEE Symp. on Foundations of Computer Science*, 1986, pp 264-273.
- [Swan 77] R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "CM* - A modular, multi-microprocessor," *National Computer Conference, AFIPS*, 1977, pp 637-644.
- [Swartzlander 73] E.E. Swartzlander, "Parallel Counters," *IEEE Trans on Computers*, Nov 1973, pp 1021-1024.
- [Tanimoto 83] Steven Tanimoto , "A pyramidal approach to parallel processing," *Proceedings 10th Annunal International Symposium on Computer Architecture*, Stockholm, June 1983.
- [Thakkar 88] Srikant Thakkar, "The Balance Multiprocessor System," *IEEE Micro*, Feb 1988, pp 57-69.
- [Thompson 77] Clark D. Thompson and H.T. Kung, "Sorting on a mesh connected parallel computer," *Communications of the ACM*, Vol 20, 1977, pp 263-271.
- [Thompson 78] Clark D. thompson, "Generalized Connection Networks for paralle processor interconnections," *IEEE Transactions on Computers*, vol c-27, no 12, December 1978, pp 1119-1125.
- [Thurber 78] Kenneth J. Thurber, "Circuit Switching Technology: A State of the Art Survey," *IEEE COMPCON*, Fall 1978.
- [Thurber 79] Kenneth J. Thurber and Gerald M. Masson, *Distributed-Processor Communication Architecture*, Lexington Books, 1979.
- [TI 90] TMS320C30 User's Guide, Texas Instruments, 1990.
- [Tripathi 79] A.R. Tripathi and G.J. Lipovski, "Packet switching in Banyan networks," *6th Annual Symposium on Computer Architecture*, 1979, pp 160-167.
- [Trujillo 82] V.A. Trujillo, "System architecture of a reconfigurable multiprocessor research system," *International Conference on Parallel Processing*, 1982, pp 350-352.
- [Tsao-Wu 74] Nelson T. Tsao-Wu, "On Neiman's algorithm for the control of rearrangeable switching network," *IEEE Transactions on Communications*, vol COM-22, no 6, June 1974, pp 737-742.

- [Tuazon 85] J. Tuazon et al, "Caltech/JPL mark II hypercube concurrent processor," *International Conference on Parallel Processing*, 1985, pp 666-673.
- [Tucker 88] Lewis W. Tucker and G. G. Robertson , "Architecture and applications of the connection machine," *IEEE computer*, August 1988, pp 26-38.
- [Uhr 72] Leonard Uhr , "Layered recognition cone network that preprocesses classify and describe," *IEEE Transactions on computers*, Vol C-21, 1972, pp 758-768.
- [Uhr 87] Leonard M. Uhr , *Parallel Computer Vision*, Academic Press, Orlando, Florida, 1987.
- [Unger 58] S.H. Unger, "A computer oriented toward spatial problems," *Proceedings of the IRE*, Oct 1958, pp 1744-1750.
- [Waksman 68] Abraham Waksman, "A permutation network," *Journal of ACM*, vol 15, Jan 1968, pp 159-163.
- [Watson 72] W.J. Watson, "The TI-ASC: A highly modular and flexible supercomputer architecture," *Fall Joint Computer Conference, AFIPS*, 1972, pp 221-228.
- [Weems 84] Charles C. Weems, "Image processing on a content addressable array parallel processor," *PhD dissertation, COINS Tech Rep 84-14, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003*, 1984.
- [Weems 89] Charles Weems, Steve Levitan, Allan Hanson, Edward Riseman, David Shu, and J. Greg Nash, "The Image Understanding Architecture," *International Journal of Computer Vision*, March 1989, pp 251-282.
- [Weems 91] Charles C. Weems, "The Architectural Requirements of Image Understanding with respect to Parallel Processing," *Proceedings of the IEEE*, (To appear), 1991.
- [Weiss 86] R. Weiss and M. Boldt, "Geometric Grouping Applied to Straight Lines," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, Miami, FL, June 1986, pp 489-495.
- [Wilkerson 87] R. Wilkerson, "A routing algorithm for the three stage rearrangeable Clos networks," *ACM Computer Science Conference*, 1987, pp 235-238.
- [Wu 78] C. Wu and T. Feng, "Routing techniques for a class of multistage interconnection networks," *International Conference on Parallel Processing*, 1978 pp 197-205.
- [Wu 79] C. Wu and T. Feng, "The reverse-exchange interconnection networks," *International Conference on Parallel Processing*, 1979, pp 160-174.
- [Zakharov 84] Vladamir Zakharov, "Parallelism and array processing," *IEEE Transactions on Computers*, Vol C-33, No 1, Jan 1984, pp 45-78.