ETL-0499

The image understanding architecture project, first annual report

Charles C. Weems
Steven P. Levitan
Allen R. Hanson
Edward M. Riseman
David B. Shu
J. Gregory Nash
James Burrill
Michael Rudenko

University of Massachusetts
Computer & Information Science Department
Amherst, Massachusetts 01003

April 1988

Approved for public release; distribution is unlimited.

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ADA203816

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ETL-0499

6a. NAME OF PERFORMING ORGANIZATION
University of Massachusetts

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
U.S. Army Engineer Topographic Laboratories

6c. ADDRESS *(City, State, and ZIP Code)*
Computer & Information Science Department
Amherst, Massachusetts 01003

7b. ADDRESS *(City, State, and ZIP Code)*
Fort Belvoir, VA 22060-5546

8a. NAME OF FUNDING / SPONSORING ORGANIZATION Defense Advanced Research Projects Agency

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
DACA76-86-C-0015

8c. ADDRESS *(City, State, and ZIP Code)*
1400 Wilson Boulevard
Arlington, VA 22209-2308

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
6E20	86		

11. TITLE *(Include Security Classification)*
The Image Understanding Architecture Project, First Annual Report

12. PERSONAL AUTHOR(S) Charles C. Weems, Steven P. Levitan, Allen R. Hanson, Edward M. Riseman, David B. Shu, J. Gregory Nash, James Burrill, Michael Rudenko

13a. TYPE OF REPORT
Annual

13b. TIME COVERED
FROM 9/23/86 TO 9/22/87

14. DATE OF REPORT *(Year, Month, Day)*
1988 April

15. PAGE COUNT
184

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Image Understanding Architecture
Knowledge-Based Vision
Real-time Computer Vision
Software Simulator
Parallel Processor

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report presents the results of the Image Understanding Architecture (IUA) project for the first year of its two-year contract period. The purpose of the IUA project is to design and construct a next-generation parallel processor that specifically addresses the needs of real-time computer vision applications.

The current effort involves the construction of a proof-of-concept, 1/64th scale prototype IUA system (hardware and software) that will serve as the basis of research leading to the design and construction of the next generation IUA system. The work is being performed jointly by the University of Massachusetts at Amherst, and Hughes Research Laboratories in Malibu.

Included in this report are a summary of accomplishments during the first year, an overview of the IUA design, a collection of algorithms, a discussion of a vision processing scenario as it is expected to take place on the IUA, a summary of the performance. (OVER)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Rosalene M. Holecheck

22b. TELEPHONE *(Include Area Code)*
202-355-3022

22c. OFFICE SYMBOL
CEETL-RI-I

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. Abstract (Continued)

figures for the IUA on the DARPA IU Benchmark Exercise, a detailed description of the architecture of the bottom level of the IUA, documentation for the IUA software simulators, and a report of the hardware design efforts at Hughes.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SUMMARY

This report presents the results of the Image Understanding Architecture (IUA) project for the first year of its two-year contract period. The purpose of the IUA project is to design and construct a next-generation parallel processor that specifically addresses the needs of real-time computer vision applications.

The current effort involves the construction of a proof-of-concept, 1/64th scale prototype IUA system (hardware and software) that will serve as the basis of research leading to the design and construction of the next generation IUA system. The work is being performed jointly by the University of Massachusetts at Amherst, and Hughes Research Laboratories in Malibu.

Included in this report is a summary of accomplishments during the first year, an overview of the IUA design, a collection of algorithms, a discussion of a vision processing scenario as it is expected to take place on the IUA, a summary of the performance figures for the IUA on the DARPA IU Benchmark Exercise, a detailed description of the architecture of the bottom level of the IUA, documentation for the IUA software simulators, and a report of the hardware design efforts at Hughes.

PREFACE

This document reports the results of efforts at the University of Massachusetts and Hughes Research laboratories during the first year of the DARPA sponsored Image Understanding Architecture project. These efforts include both hardware and software development.

Because a great deal of documentation has already been written as part of these efforts, we have chosen to assemble the majority of this report from those documents. The report begins with a brief introduction and overview of the project and the major accomplishments in the first year. The remainder of the report is a lengthy set of appendixes that document the work in detail.

The reader is thus referred to the main text for a management summary of the project. If the reader's interests are in the technical accomplishments, then the appendixes are more appropriate. The table of contents covers all of the material in the appendixes as well as the main text.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS - CONTINUED

IMAGE UNDERSTANDING ARCHITECTURE ANNUAL REPORT

Introduction

The primary goal of the Image Understanding Architecture project is to build a proof-of-concept prototype of a 1/64th slice of a next-generation vision architecture, and develop the software support environment that will be needed to utilize the hardware. The majority of the hardware effort is taking place at Hughes Research Laboratories, although UMass has principal responsibility for the design of the IUA architecture. UMass has also undertaken some smaller portions of the hardware development (the feedback concentrator for the low and intermediate level arrays, communications router for the intermediate level array, and a simple version of the array control unit). The majority of the software effort is taking place at UMass, although Hughes is 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.

During the first year of this program we have focused on developing a complete hardware specification for the architecture, fabrication of the necessary custom chips, and the construction of software simulators for the architecture. Appendix 1 is an overview of the IUA that has been submitted for publication in the *International Journal of Computer Vision.* Appendix 2 is our detailed functional specification document for the lower level (CAAPP) array and its interfaces to the other parts of the IUA. Appendix 3 is a summary of our chip fabrication efforts, with photomicrographs and testing reports for all of the chips that have been built. Appendix 4 summarizes our efforts in building software simulators for the IUA on the TI-explorer and Sun-3 workstations, as well as on the VAX.

Summary of Project Accomplishments

The major accomplishments during the first year of the contract are as follows.

1. Development of a detailed functional specification for the CAAPP and ICAP levels of the IUA. (In the prototype the SPA processor and host are the same physical device. Our current plan is to use a Sun workstation to fulfill this function.)

2. Development of software for the IUA. We have implemented IUA simulators under three different environments: Unix on a Sun-3, VAX, VMS, and a TI-Explorer. The VAX-based simulator is designed to be portable to other systems, and is in fact the basis for the Sun-based simulator. The simulator on the Sun has been integrated with the workstation's windowing system

but retains the same capabilities as the VAX-based version. The VAX- and Sun-based simulators implement the Array Control Unit (ACU), CAAPP array, Coterie Network, and Backing Store. The ICAP array portion of the simulators is still under development (we originally planned to use Texas Instrument's simulator for the TMS320C25 processor, but have been unable to obtain it).

The Explorer-based simulator runs on an Odyssey co-processor board that contains four TMS32020 processors. It implements the ACU, CAAPP array, Coterie Network, Backing Store, and a subset of the ICAP processor instruction set (that portion of the TMS320C25 instruction set which is supported by the TMS32020). The Explorer-based simulator is not portable, however, it is roughly 20 times faster than the other simulators. (The increased speed is a direct result of the special hardware that the Odyssey co-processor provides.)

3. Design and construction of a 32 processor custom VLSI CAAPP chip. The design included 32 processors and memory, but did not include the Coterie Network or Backing Store Controller. Due to problems in the fabrication process, none of the chips were functional. However, Hughes was able to microprobe the dies and evaluate selected portions of the design.

4. Design of a 64 processor custom VLSI CAAPP chip. The design includes (in addition to the processors), the corner-turning memory, nearest neighbor network, and the Coterie Network, but not the Backing Store Controller. The chip has been submitted for fabrication and is due back in December of 1987. (Note: This has been delayed until at least the end of February for parts from a backup run. Parts from the primary vendor are not expected until at least April.)

5. Design and construction of the Feedback Concentrator chip. This chip provides the Some/None and Done responses from the CAAPP and ICAP arrays to the ACU. These chips also failed to function after the first fabrication run. The problem was traced to an incorrect specification of the standard pad frame when the chip was submitted to MOSIS. The error resulted in the frame being reduced in size and incorrectly bonded. The problem has been corrected and the design has been re-submitted, with chips due back in December of 1987.

6. Design and construction of the ICAP Router chips. These chips provide a 32 by 32, bit-serial crossbar switch. The switch is centrally controlled by the ACU. It stores up to 32 input to output connection patterns in an on-chip memory called the Connection Pattern Cache (CPC). As its name implies,

the CPC is used to store the 32 most frequently used processor connection patterns. The ACU can load new patterns into the CPC at run time. (Patterns can be completely rewritten, or changed incrementally.) As with the 32-processor CAAPP chips, the router chips encountered fabrication problems (excessive spread in the metal layers) that kept them from functioning. We are currently exploring options to obtain more reliable fabrication through MOSIS.

7. We participated in the DARPA IU-Benchmark exercise, providing timing estimates for both the IUA and the Encore Multimax. We have also worked with the University of Maryland to specify the next version of the benchmarks. These be ready for distribution by, and will presented at the IU Workshop in April of 1988. The specification will also be presented at the International Conference on Supercomputing, and at the IEEE Conference on Computer Vision and Pattern Recognition.

Changes in Project Goals

It should also be noted that the version of the IUA that is currently under construction is significantly more powerful than the design that was originally proposed. We have incorporated a number of enhancements that were developed during the year that elapsed between approval of the proposal and the start of funding. We elected to do this because we felt that the capabilities of the delivered system could be considerably expanded at no additional cost to the government. However, doing so has made the IUA prototype much more desirable so that there is a good deal of interest in building copies. Because our original budget was simply for building a proof-of-concept prototype (not a production-quality, fieldable system), we have submitted a second proposal that would fund additional systems-integration engineering to improve hardware reliability, construction of a sophisticated ACU, and optionally the construction of several copies of the prototype.

Given this general background, we can list the specific changes that have occurred in the program, and those that will occur under the expanded funding. There are three major areas of change: Hardware, software, and responsibilities. For each of these areas we will describe what was initially proposed, what we currently plan to deliver, and what it should be possible to deliver with the proposed additional funding.

Hardware. The initial proposal had a custom VLSI chip containing both the CAAPP and ICAP processors. The CAAPP cells had a simpler architecture, and were connected to their off-chip neighbors by a 4-way time-multiplexed network. Each CAAPP PE had 128 bits of RAM, of which 8 were shared with the ICAP.

The CAAPP processing elements were restricted to a one bit data path. The ICAP processors were initially SIMD, with about 4K of RAM. Each ICAP PE had an 8-bit data path, and a simple ALU (logical operations plus addition). The interface to the SPA was through a multiplexed controller I/O port. That is, each SPA processor could temporarily become the controller for a custom bit-slice processor, capable of issuing sequences of instructions at full rate, and of performing simple computations and making limited processing decisions. This was judged to be adequate for experimentation and demonstration purposes, since it would be able to execute individual vision algorithms with limited reliance on the host. However, it would depend upon the host for control of the overall vision processing strategy.

The hardware that we currently plan to deliver has a greatly enhanced processor, and a downgraded controller. The CAAPP and ICAP are separate chips, with only the highly regular CAAPP architecture being fabricated in custom VLSI. Each CAAPP PE will have 320 bits of RAM. Of these, two 128 bit banks provide a double-buffered swapping mechanism that can access a 32-K bit RAM backing store that is associated with each PE. The entire backing store is shared with the ICAP processor associated with each group of 64 CAAPP processors. Communications between CAAPP processors is now provided via a full mesh (no multiplexing), and an augmented mesh called the Coterie Network (or Some/None Mesh). This second network allows construction of wired-OR busses between contiguous processor groups. An example benefit that is obtained with the augmented mesh is a factor of 1000 speed increase over a standard mesh in performing connected component labeling operations. The CAAPP processors also have an 8 bit data movement path, in addition to the single bit data path that runs through the ALU. The ICAP processors now run in MIMD mode, and in a closely controlled mode called synchronous-MIMD. Each ICAP processor is a full 16 bit computer with 128K program memory and 128K local data memory (in addition to accessing the 256K backing store). The ICAP ALU is quite sophisticated, with single cycle multiply and accumulate operations. The interface to the SPA is now through a shared memory (128K per ICAP processor), which allows globally controlled processing to continue in the CAAPP and ICAP while the SPA is looking at low and intermediate level results. The controller has been reduced in complexity to permit construction at UMass. In its current form it will be a large instruction buffer with start and stop address registers, and a program counter. The host will simply load an instruction sequence and then start the processors. Upon completion, the host will be interrupted and it can examine the contents of feedback registers. This controller will be adequate for limited experimentation and for demonstrating the capabilities of the architecture. However, it is far from an ideal programming environment.

With the proposed additional funding, the hardware will change in two ways. The controller will be upgraded to a fully programmable processor that can support a complete programming environment; and the controller and processor will undergo a system integration period in which the system will be debugged, and its reliability greatly increased. The controller will contain two separate processors. One will be a 68020 with a full support environment that has access to the array. This processor provides a rapid prototyping facility that will allow users to quickly and easily take advantage of the processing power of the IUA. However, the 68020 can only issue instructions to the IUA at roughly 10% of the IUA's maximum rate. Attached to the 68020 is a custom, bit-slice, horizontal microcode engine. The micro-engine is a complete processor that can drive the IUA at its peak rate for long periods, however, it requires that code be written in micro-assembly language to obtain best results. Once an algorithm is coded for the micro-engine, it can be executed by a call from the 68020. Thus, as the library of micro-coded routines grows, the performance obtained by users of the 68020 will improve. With this controller architecture, the IUA prototype will be transformed from a proof-of-concept demonstrator into an immediately useful vision processor.

Software. We initially proposed to deliver a functional simulator of the IUA for use by the IU community, and some additional software that would be for local use only (because it would require the IUA prototype hardware. in order to be effective). This latter collection of software includes a debugger, an interface to the host processor, subroutine libraries, a mini-code generator (for automatic generation of convolution operations), and some IU algorithms. Equivalent software for use with the simulator was also to be provided.

Following our initial change in plans we felt that the same software could be delivered, although it would not be as full featured as we had originally planned. The increased complexity of the architecture, for example, made the simulator more difficult to write. Hence we elected to simplify the debugger and make it an integral part of the simulator, rather than an application that would run on the simulator. Because we must now write a system kernel for the ICAP processors, we have reduced our effort in coding IU algorithms.

With additional funding for the controller, we can divert some of our controller budget into the software effort and make some modest improvements in the quality and features of the deliverables. The change will be small because at this point we have already expended some of the controller budget in designing the simple controller, and much of the remaining money will be needed to develop specifications for the new controller. With additional software money, we could hire another programmer and significantly enhance the software effort. This would probably make the difference between a minimally adequate software environment and a useful program development environment.

Responsibilities. In our initial proposal the division of labor was simple: UMass was responsible for the system architecture and software development, and Hughes Research Labs in Malibu was responsible for hardware development, including the controller and system integration.

After our initial change in plans, UMass was responsible for the architecture, software, construction of the controller, ICAP router, array feedback concentrator chips, and the system integration. Hughes was to build the processor, and to provide some support in the integration effort.

Under the expanded effort, UMass would return to being responsible for the architecture and the software. Hughes would remain in charge of processor construction, and would be responsible for construction of the controller and for final system integration and delivery.

In summary, the IUA project has gone through two major phases and is hopefully about to enter a third phase. The first phase was based on a simple version of the IUA concept, for which the development contract was awarded to build a proof-of-concept prototype slice of the hardware and a collection of supporting software packages. The second phase involved making significant enhancements to the architecture while scaling back the controller effort so as to maintain the same overall budget. The third phase is intended to convert the enhanced IUA prototype slice from a proof-of-concept device into a reproducible, fieldable system.

Conclusions

Except for some VLSI fabrication delays that resulted from various problems with the vendors under contract to MOSIS, the IUA project is proceeding as planned and on schedule. The fabrication delays could, however, lead to some serious delays in the later portions of the IUA project. We have compensated by pushing forward our design and development efforts while awaiting the return of various test parts. If those parts are free of major design errors, then we should be able to continue to maintain our schedule. However, there is always the chance that a major error will be found that requires a significant redesign effort, which would substantially delay completion of the hardware.

The architectural design and software development components of the project are proceeding as planned. We have also taken on the task of specifying the next generation of the DARPA IU Benchmark and programming a solution to the benchmark on a standard (sequential) computer system, which follows from our involvement in the original benchmark exercise. We hope to be able to program most, if not all, of the new benchmark on our IUA simulators.

We have submitted a supplemental proposal for construction of a sophisticated controller for the IUA prototype. If this effort is funded, it will allow us to convert the prototype from a proof-of-concept demonstrator into a reproducible, production quality system. The availability of such a system will greatly enhance our capability to perform research on parallel computer vision algorithms, which will in turn guide the design of the next generation IUA.

Appendix A:
Image Understanding Architecture Overview

The Image Understanding Architecture

Charles C. Weems, Steven P. Levitan*
Allen R. Hanson, Edward M. Riseman

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

David B. Shu, J. Gregory Nash
Hughes Research Laboratories
3011 Malibu Canyon Road
Malibu, CA 90265

ABSTRACT

This paper provides an overview of the Image Understanding Architecture (IUA), a massively parallel, multi-level system for supporting real-time image understanding applications and research in knowledge-based computer vision. The design of the IUA is motivated by considering the architectural requirements for integrated real-time vision in terms of the type of processing element, control of processing, and communication between processing elements.

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 distinctly different from the other two levels, designed to best meet the processing needs at each of the corresponding levels of abstraction in the 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 communication needs of each level of abstraction.

An associative processing paradigm has been utilized as the principle control mechanism at the low and intermediate levels. It provides a simple yet general means of managing massive parallelism, through rapid responses to queries involving partial matches of processor memory to broadcast values. This has been enhanced with hardware operations that provide for global broadcast, local compare, Some None response, responder count, and single responder select. To demonstrate how the IUA may be used for vision processing, several simple algorithms and a typical interpretation scenario on the IUA are presented.

* Current address: Department of Electrical Engineering, Benedum Hall, University of Pittsburgh, Pittsburgh, PA 15261

We believe that the IUA represents a major step towards the development of a proper combination of integrated processing power, communication, and control required for real-time computer vision. A proof-of-concept prototype of 1/64th of the IUA is currently being constructed by the University of Massachusetts and Hughes Research Laboratories.

1. INTRODUCTION

The motivation for building a "vision machine or image understanding architecture stems from the need to support a diverse set of complex operations on massive amounts of data at high speeds, and to provide an environment for experimentation that minimizes the software effort that is necessary to build a vision system.

Machine vision is one of the most computationally intractable domains of artificial intelligence research. 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 contains three quarters of a million color-intensity data values which comprise the picture elements (pixels) of the image. Performing a single operation on each of these pixels requires executing about 23 million instructions per second just to keep up with the input. Of course, the computation needed to perform image interpretation is much more than one operation on every pixel in an image. Many researchers believe it is in the range of 3 to 4 orders of magnitude more computation. Although "real-time" interpretation does not mean that a full interpretation must be completed with each new frame (because much of the previous interpretation may be re-used, and some vision tasks may not require such frequent updating of an interpretation), nevertheless a very large computational rate is required.

The construction of a symbolic description of the environment depicted in an image involves a variety of algorithms and data structures that must be employed at different levels of computational granularity. There are basically three types of computation 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 the other papers of this issue [Draper, 1988. Boldt, 1988].

Image interpretation may thus be characterized as involving three different levels of processing, each with its own specific class of information. Additionally, those levels must be able to interact through bottom-up transfers of information and top-down control of processing. The low, intermediate, and high levels of representation and processing, together with our understanding of how these levels interact, provides the basis for the design of our Image Understanding Architecture (IUA) presented in section 4.

2. ARCHITECTURAL CONSIDERATIONS FOR KNOWLEDGE-BASED VISIONS

In this section a variety of considerations and issues relating to vision processing are presented. The different forms of data and levels of processing have different architectural requirements. In particular we will elucidate the communication issues between parallel processing elements at each level and between levels. This communication will involve both data transfer and control information necessary to support bottom-up and top-down processing.

2.1 Architectural Requirements for Vision

In order to provide a focus to the ensuing discussion we present without further motivation general requirements for architectures to support vision in an integrated manner:

- The ability to transform pixel data (i.e. an image) into a set of meaningful symbols that describe it.

- The ability to process both pixel and symbol data in parallel.

- The ability to simultaneously maintain the low, intermediate, and high-level representations.

- Fast I/O and processing rates for huge amounts of sensory and intermediate level data.

- The ability to select particular subsets of data for varying types of processing.

- Summary feedback and evaluation mechanisms that allow focusing of attention and data-directed processing, without having to dump processing results to some "host" for external evaluation.

Beyond these requirements, two significant architectural implications can be derived from an understanding of our approach to the computer vision problem:

- Multiple levels of representation and stages of processing are essential and require very different types of processing elements.

- Fine grained and high speed communication and control is required both among the processes at each level and between the different processing levels.

The following discussion examines the computational requirements needed at the different processing levels and then moves on to discuss the communication and control issues involved in vision processing.

2.2 Processing Characteristics for Multi-Level Architectures

The three levels of abstraction discussed above have been mapped into three levels of architectural requirements with each level having the appropriate architecture for the set of tasks at that level. The most straightforward level of processing from an architectural view is the low level, close to sensory data, where uniform computation is generally applied, usually at each pixel, across the image. Here, we need to perform operations such as convolution (e.g. smoothing, edge detection, etc.) and [illegible] algorithms for extraction of lines, regions, and surfaces with their associated attributes, as well as feature matching between frames in motion and stereo processing. Since each pixel, or a neighborhood around each pixel, must be processed (usually repeatedly), the typical organization is SIMD. Some of these operations may also require processing over larger image neighborhoods such as in correlation between stereo and motion frames.

In addition to high-speed low-level operations, there is a need to be able to quickly load the data and to test the results of partial processing. To perform multiple iterations of lower-level operations under control of higher-level processes requires the fast, concise transmission of data upward and control information downward, as discussed in the next section.

At the intermediate-level our concerns are for comparing and manipulating the symbolic tokens associated with the extracted image events, such as regions, lines, and surfaces. This often involves grouping these token events into more complicated structures such as rectangles, planes, sets of parallel lines, and textured regions. Although there has not been as large a body of research on grouping and organizing surface patches, we believe that the processing issues for surfaces at the intermediate-level are equivalent to those for regions and lines. The types of operations needed at this level involve comparing token attributes and relations and indexing tokens by the values of their attributes and relations which is referred to as associative retrieval. The processes are used to partition, aggregate, and in general transform these tokens into structures that more naturally match the object representation.

The intermediate level also requires specialized processors [Duff, 1986]. Large numbers of line and region fragments can be generated by even the most effective low-level algorithms. To perform grouping operations (e.g., merging and splitting of regions, or linking and reorganizing lines) one needs a large amount of "less local" communication. Fragments of lines must be matched and merged across large fractions of the image. Similarly, regions and surfaces need to be merged and compared with others from possibly non-contiguous areas.

Data reduction through abstraction is fundamental to intermediate-level processing. For it to be done quickly requires architectural support for both inter-level and intra-level communication as well as a flexible repertoire of data manipulation instructions. Thus, the intermediate level must operate as a server for the queries made as a result of object and scene processing at the high level, support data reduction processes, and provide control and evaluation mechanisms for the lower level processes.

Interpretation operations at the semantic level must in some manner construct complex descriptions of the scene from the data provided by the low and intermediate processing and stored world knowledge. The computational and communication needs of these processes should be provided by symbolic processors, that form a third tier of processing power directed toward solving the image understanding problem. Support must be provided for semantic processing, including the formation and verification of object hypotheses, manipulation of 2-D and 3-D object models, mechanisms for focus of attention, complex control strategies for application of multiple knowledge sources, and inferencing under uncertainty. This type of processing involves extensive symbolic computation. Note that the geometric manipulation of three-dimensional models, may require significant amounts of numeric computation.

2.3 Communication Between Processing Levels

A central characteristic of image interpetation is the bi-directional flow of communication and control up and down through all representation and processing levels. The ability to suppor these characteristics allows multiple image operations to be performed. processing results to be evaluated. and algorithms to be re-applied with different parameters on different parts of the image. Specific communication operations must be performed in microseconds or in tenths of microseconds in order for different interpretation strategies to be tried dynamically within one or a few frame times.

In the upward direction. the communication consists of image abstraction and segmentation results from multiple algorithms, and possibly from multiple sensory sources. It also involves the communication of a set of attributes describing the token associated with each extracted image event; these are stored in a symbolic representation at the intermediate-level. From the intermediate- to high-level the upward communication consists of grouped collections of lower-level abstractions, plus descriptions and responses to queries about token attributes and relations. In addition. summary information and statistics allow processes at the higher-levels to assess the success of lower-level operations. In the downward direction the communication consists of control knowledge-directed processing and grouping operations, commands for selecting subsets of the image for specifying further processing in particular portions of the image, modification of parameters of lower-level processes. and requests for additional information in terms of the intermediate representation.

2.3.1 Associative Communication and Control

From the preceding discussion, it can be seen that communication between processing levels involves the rapid transfer and evaluation of information from lower to higher levels, and the ability to exercise fine grained control from higher to lower levels. Based on our experience with highly parallel algorithms [Levitan. 1987]. we believe that the best way to meet these requirements of high speed. fine grained communication and control at the low and intermediate levels is with associative processing techniques. There are four processing capabilities that are key to associative computation [Foster. 1976]:

- Global Broadcast/Local Compare/Activity Control
- Some/None Response
- Count Responders
- Select a Single Responder

Associative processing can best be understood by an example of a single controller (a teacher) interacting with an associative array (a class of students). If the teacher needs to know if any student in a class has a copy of a particular book, the teacher can simply state, "If you have the book, raise your hand." The students each make a check, in parallel, and respond appropriately. This corresponds to a broadcast operation of a controller and a local comparison operation at each pixel in an array, to check for a particular value. Both operations assume that the local processors have sufficient computational capability to perform the comparison.

Query and response is just the first part of associative processing. Thus far, we have described only a content addressable ("If your hand is up, I'm talking to you.") scheme. We must be able to conditionally generate symbolic tags based on the values of data and use those tags to constrain further processing. It should be possible to broadcast a command that selects pixels or tokens via constraints on the values of their attributes. This capability becomes more interesting as the controller performs multiple logical operations on pixels or tokens (line, region, etc.) that have multiple tags based on their attributes (including spatial location), and relationships to other pixels or tokens. As processing continues only subsets of the data are involved in any particular operation, but note that all pixels or tokens with a given set of properties are being processed in parallel.

The ability to associate tags with values is half the battle for high speed control. We also need to get responses back from the array quickly. Forcing the teacher to ask each student if they have their hand up defeats the process. The teacher can see immediately if any of the students have their hands up. Similarly, a Some-response/No-response (referred to as Some/None) output from the pixel array allows the controller to immediately determine cetain logical properties about the data in the array, and therefore the state of processing in the array, *without looking sequentially at the data values themselves.*

Additionally, fast hardware to perform a count of the responders allows the controller to obtain summary information about the state of the data in the array. For example, the ability to sum values gives us the power to rapidly compute statistical measures such as mean and variance. By using the properties of the radix representation of numeric values in the array, we can also use the counting hardware to sum the values in the array. Because they are supported in hardware, the Some/None and Response Count operations allow programs to be executed that can *conditionally* perform operations based on the overall state of the computation.

A single-responder-select operation provides one mechanism for transferring non-summary information from a lower to a higher level in the vision processing hierarchy. A higher level processing element may select a single cell or token in a lower level, based upon a partial match of its information fields, and read it out. Selecting a single responder (referred to as "Select-First") is useful when a group of tokens have been associatively selected by matching some of their attributes to broadcast values, and it is then desired to read out other attributes values for individual members of the set of selected tokens. Another purpose is to select a single cell as representative of a group of cells which can then be used to store data about that group. For instance one cell associated with a single pixel in a region could store the mean and variance for the entire region.

By implementing the four operations discussed above in specialized hardware, associative processing can be used as the communications paradigm between each pair of levels and as the control paradigm for the low and intermediate levels in the hierarchy. Constraints can be broadcast for selecting pixels, or symbolic tokens for selective processing. In this way higher levels control lower levels. It is possible to test and/or count the response that comes after processing data to allow conditional branching for the next step of processing in a given algorithm. Thus, the lower levels provide feedback to higher ones.

2.3.2 Parallel Data Transfer and Control Between Levels

As mentioned above the single-responder-select operation is useful when a small number of data values must be copied to a higher level. However, when a large number of values must be transferred between levels, a parallel data path between adjacent levels is more appropriate. A simple means of achieving this is by connecting spatially collocated processors in adjacent pairs of levels with independent data paths so that inter-level data transfers may be performed in parallel.

The parallel data transfer mechanism permits the use of a general processing strategy in which each level is used to transform a lower-level representation into a higher-level representation. All or part of this new representation may then be extracted by the next higher level of processing which then treats the level below it as an associative data base.

Since the parallel transfer mechanism also permits the passing of control information from higher to lower levels, the granularity of control can be made to vary. This would allow, for example, initial processing to take place in a coarse-grained control mode with a single controller (SIMD) and later processing to take place in a fine-grained control mode where multiple controllers are active (Multi-SIMD, or MIMD). The former is useful for generating initial hypotheses about an image, while the latter is better suited to resolving multiple local conflicts between those hypotheses, and to filling in details of the interpretations.

As can be seen from the preceding discussion, a vision architecture requires a complex and unique combination of computational power, communication, and control. The remainder of this section examines existing parallel architectures with regard to these requirements.

3. REVIEW OF EXISTING ARCHITECTURES

In order to meet the requirements of image interpretation, it is clear that the power of parallel processing must be utilized. Existing parallel architectures, however, do not

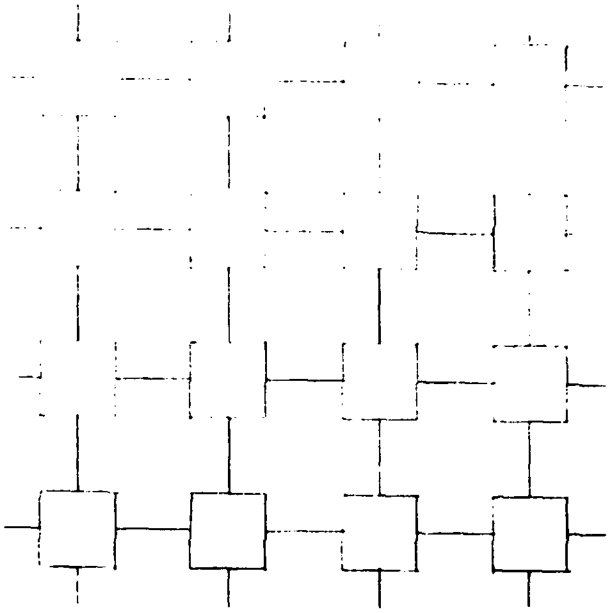


Figure 1. A 4-Connected Mesh of Processors

simultaneously address the varying computational needs of computer vision, although any given architecture may perform quite well on some portion of the vision problem. This section will briefly review six general classes of existing architectures and discuss their advantages and disadvantages with respect to solving the vision problem. The six classes are mesh-connected arrays, pyramids, hypercubes, mesh-plus-hypercube-connected arrays, shared memory systems, and systolic arrays.

3.1 Mesh-Connected SIMD Arrays

These machines are an obvious choice for image processing applications. A typical machine of this class has thousands of small processors that are connected to their nearest 4, 6, or 8 neighbors (Figure 1). Processors respond in SIMD mode to instructions that are broadcast by a central controller.

The advantage of this architecture is that images map quite naturally onto its structure. When the image size matches the size of a mesh-connected architecture, maximum parallelism can be obtained for operations that involve computations on individual pixels or small neighborhoods of pixels.

This type of architecture has several disadvantages for vision processing. Foremost of these is a lack of MIMD processing capability that is necessary to support intermediate- and high-level vision. Different algorithms must be applied as different hypotheses are pursued across the image. In addition, there are many low- and intermediate-level vision algorithms that involve the grouping or matching of image structures which are spatially distant in an image. In a mesh-connected architecture, however, communication of information across long distances is very time consuming. Lastly, in order to economically build an array with thousands of processors, the individual processors must be quite small. For example, typical processors have a 1-bit ALU, a small amount of memory, and communication links to its immediate neighbors. Although such processors are adequate for performing pixel operations, they lack the power that is needed for manipulating intermediate-level tokens; for example, computing trigonometric functions on floating-point values is very slow on a 1-bit processor.

Some examples of mesh connected architectures are CLIP-4 [Duff, 1978], MPP [Batcher, 1980], DAP [Hunt, 1981], GRID [Arvind, 1983], and GAPP [Davis, 1984]. Each of these machines has its own special features, but all of them have the same general form. One additional characteristic of all of these machines is that they do not strongly emphasize global summary feedback mechanisms that permit rapid evaluation of processing results. Although it is possible for these machines to compute summary values or to transfer their results to another machine for evaluation, doing so is a much slower operation. This reflects the fact that they are used primarily for low-level processing or image enhancement where results are interpreted by humans rather than forming the first stage of an autonomous vision system.

3.2 Pyramid Processors

An extension of the mesh-connected array concept is the multi-resolution pyramid or quad-tree. This structure has been proposed in a variety of forms [Uhr, 1972; Hanson, 1974, 1980; Rosenfeld, 1986; Tanimoto, 1983], but the essential idea is that an image-sized mesh-connected array is augmented by layers of successively lower resolution mesh-connected arrays. Except for those processing elements at the bottom level, each processing element in a pyramid is typically connected to four processors in the array below it, hence the term "pyramid" (Figure 2).

The pyramid processor provides the capability for quickly changing the resolution of an image, which can significantly improve the execution speed of many low-level algorithms (especially those that depend upon communication between cells that are spatially distant in an image). However, pyramid processors are more difficult to build than mesh-connected arrays because they have a more complex arrangement of communication links and require one-third more processing elements than a mesh-connected array of the same image resolution. Although at least two pyramid processors have been built at universities [Uhr, 1987], none have yet been built commerically.

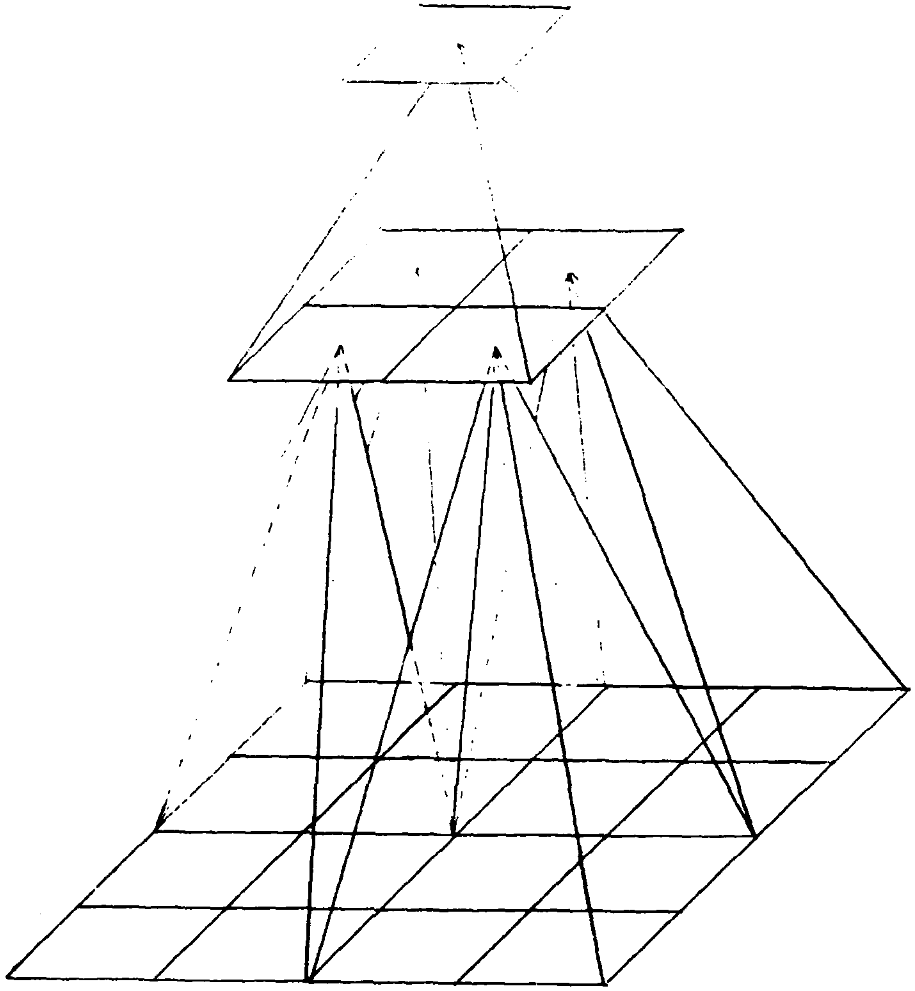


Figure 2. A Pyramid of Processors

Even though a pyramid architecture has multiple levels of processing elements, it should be noted that it implements an image resolution hierarchy, whereas computer vision requires an architecture that implements a hierarchy of abstraction levels. In a typical pyramid architecture, all of the processors are identical and execute in SIMD fashion. A vision machine, on the other hand, clearly requires different types of processors at different levels, and a variety of modes of parallelism including both SIMD and MIMD. At successively higher levels of a vision machine, the number of processing elements may decrease, but their complexity will increase so that, physically, the circuitry at each level of a hierarchical vision machine will not decrease in direct proportion to the number of processing elements (and could even increase). This contrasts with a pyramid architecture, in which the amount of circuitry decreases by a factor of four with each successively higher level.

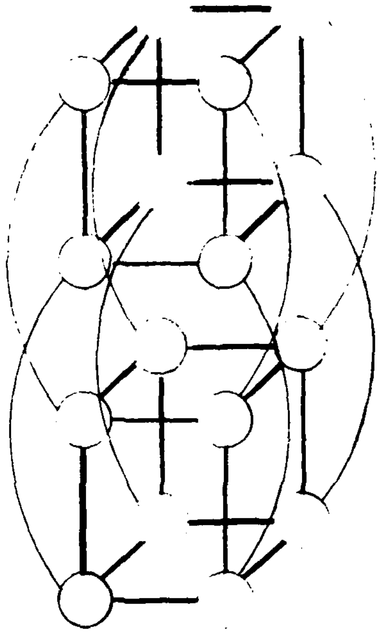


Figure 3. A Hypercube of Dimension 4

3.3 Hypercube Processors

Machines of this class consist of processors connected by communication links whose arrangement is topologically equivalent to an n-dimensional cube. A hypercube machine of dimension d will have 2^d processors, each with d communication links to other processors. Each processor is at most d-1 links away from any other processor (Figure 3). Machines of this class range in size from tens to hundreds of processors, although at least one system with a thousand processors has been proposed [NCube, 1985]. A processor in a machine of this type is typically an 8,16, or 32 bit microprocessor with local memory of at least 64K-bytes and as much as 2.5 M-bytes.

One advantage of the hypercube architecture is that it provides rapid communication between all processors because the network has a small diameter. Another advantage is that many popular network topologies, such as the mesh and tree, can be embedded in the hypercube (though not always efficiently). The main disadvantage is that for large numbers of processors, the hypercube can become rather costly. For example, a 512 × 512 array would require an 18-D hypercube with 18 links per node. This is double the number of links found in a pyramid and 4.5 times the number in a mesh. It should also be noted that the hypercube cannot be expanded as easily as simpler topologies, because a larger hypercube requires that links be added to each node. Thus, if a hypercube node is designed with d links, it can only be used to build a system with up to 2^d nodes. The nodes of meshes and pyramids, on the other hand, do not inherently limit the overall size of a system. Whereas meshes and pyramids are typically operated in SIMD mode, hypercubes have also become popular for MIMD architectures.

Some examples of hypercube architectures are the Intel Hypercube [Rattner, 1985], NCube [NCube, 1985], and Cosmic Cube [Seitz, 1985].

3.4 Mesh-Plus-Hypercube-Connected Arrays

In addition to the pyramid approach, one way to eliminate the limitation on local neighborhood communication in a mesh-connected array is to add a set of hypercube communication links between the processors. Although a mesh may be embedded topologically in a hypercube network, the result is usually slower and more cumbersome than a true mesh for image processing operations. Thus, this type of architecture combines the advantages of the two topologies and has the same disadvantages as the pure hypercube.

An example of this class of machine is the Connection Machine [Hillis, 1986], which is technically a SIMD array processor, but whose hypercube router contains additional computational capability beyond that of the basic array.

3.5 Shared Memory Multiprocessors

Typical machines of this class are MIMD parallel processors in which each processing element is a 16- or 32-bit microprocessor that has access to a large global memory. In some cases the processing elements may also have a smaller amount of local memory. There are two general methods for linking processors to memory. The simplest method involves a high-speed bus to which all of the processors and memories are attached (Figure 4). Such machines typically have fewer than 30 processors (although systems with hierarchical bus architectures are being built that will have larger numbers of processors). A more complex method involves the use of a multistage switching network that provides links between processors and memory on a demand basis (Figure 5). Machines of this type have been built with up to 128 processors.

The advantage of a shared memory is the ease of implementing a large shared data structure, such as a blackboard [Erman, 1980, Nii, 1986, Draper, 1988]. Thus, this class of machine is suited for high-level vision tasks. Although a shared memory machine can perform image processing operations in parallel by dividing the image up among the processors, it is far less effective than a SIMD array. In addition to the much lower level of parallelism, this reduced effectiveness stems from the MIMD nature of the machine class. Each parallel operation must be created as a new process on the system, incurring some amount of overhead. Also, whenever processes must interact, they suffer a time penalty

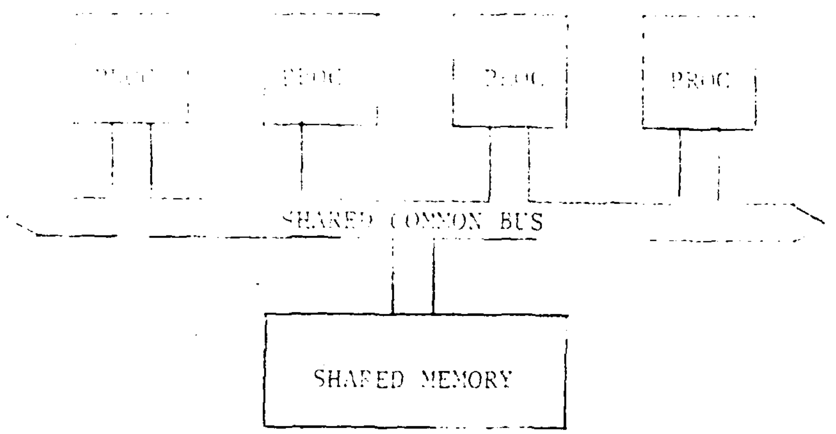


Figure 4. Common Bus Shared Memory Multiprocessor

in synchronizing with each other. Because the actual image processing operations execute relatively quickly when they are divided among multiple processors, overhead grows to dominate the total processing time. In many cases, increasing the number of concurrent processes beyond a certain point actually causes an increase in execution time.

Two examples of the bus architecture shared memory are the Encore Multimax [Encore, 1986] and the Sequent Balance [Sequent, 1986] machines. Examples of shared memory systems that use multistage networks are the BBN Butterfly [Crowther, 1985] and the IBM RP-3 [Pfister, 1985].

3.6 Systolic Processors

These machines take their name from physiology. The term "systolic" refers to the contraction of the heart that rhythmically forces the blood forward. A typical systolic processing element has a small set of inputs and outputs. On each machine cycle it takes values from its inputs, performs an operation on them, and passes the results to its outputs. These systolic elements are chained together to form a systolic array. Data is pumped into one end of the array, passes through the elements in serial fashion, and the results emerge at the other end of the array. The systolic array can also be thought of as a pipeline with a series of pumping (or processing) stations. Once the pipe is filled with data, all of the processing stations are operating on values in parallel (Figure 6).

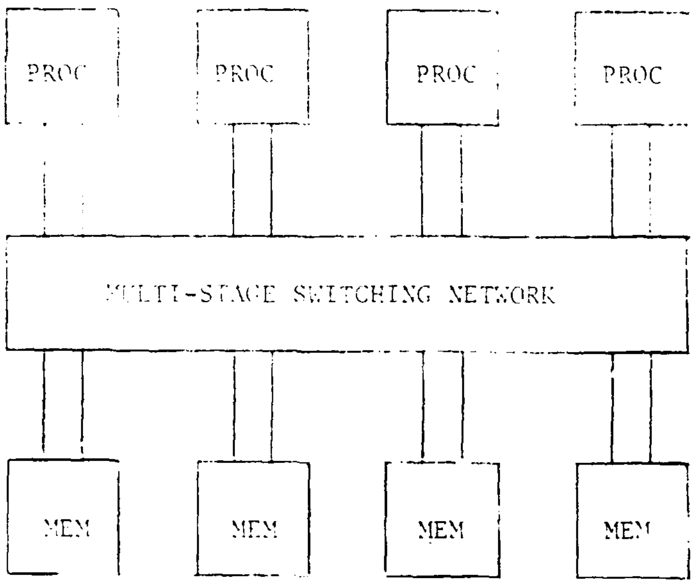


Figure 5. Switching Network Shared Memory Multiprocessor

Systolic array elements can be either general-purpose programmable function units or special-purpose fixed function units. The primary advantage provided by programmable systolic architectures is high performance for low cost. They are well suited for image processing tasks, and can also work well with any application that involves large arrays of data. The main disadvantage of the systolic array is that evaluation of the processing results must wait until all of the data has passed through the array. If a systolic array processes an image in one frame time, then this restriction has the effect of only allowing the controlling process to make a decision and change the array's programmed functions once each frame time. In a systolic array, it is thus more difficult for a vision system to quickly adapt its processing strategy to the actual characteristics of an image.

Of course, systolic arrays are not well suited for high-level vision because most high-level processing involves data structures that are more complex than numerical arrays. Systolic arrays are also not designed to support the MIMD style of processing that is characteristic of high level processing. One example of a systolic array that is being used for image processing is the CMU WARP [Annaratone, 1987].

4. OVERVIEW OF THE IMAGE UNDERSTANDING ARCHITECTURE (IUA)

The Image Understanding Architecture represents a hardware implementation of the

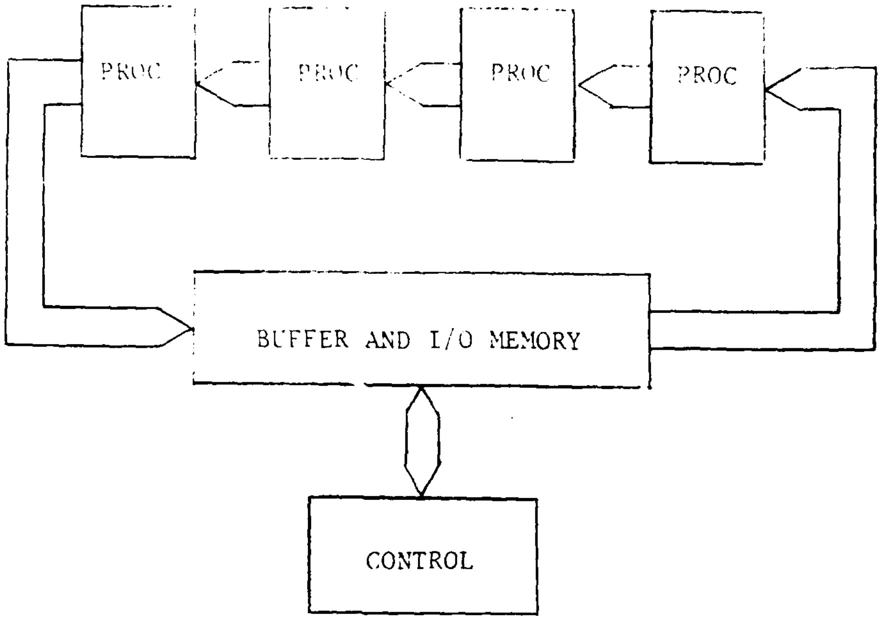


Figure 6. A Systolic Array

three levels of abstraction inherent our view of computer vision. It consists of three different, tightly coupled parallel processors. These are the Content Addressable Array Parallel Processor (CAAPP)* at the low level, the Intermediate Communications Associative Processor (ICAP) at the intermediate level, and the Symbolic Processing Array (SPA) at the high level (Figure 7). The CAAPP and ICAP levels are controlled by a dedicated Array Control Unit (ACU) that takes its directions from the SPA level. In each layer of the IUA the processing elements are tuned to the computational granularity and algorithms required by that particular level of abstraction. For example, it is inappropriate to try to run concurrent LISP at the lowest level, because processing there is primarily concerned with fast pixel operations, and should be tuned for real-time image processing. At the highest level, on the other hand, symbolic AI processing will be the main objective, so the high-level processors are selected for their ability to run LISP code efficiently. Figure 8 shows the relationship between the IUA and the UMass VISIONS system, which is discussed elsewhere in this issue [Draper, 1988, Weiss, 1988].

* The term "content-addressable" is a synonym for "associative" and is an alternate term that now is not as widely used as it was when some of our work began [Foster, 1976, Weems, 1984a.]

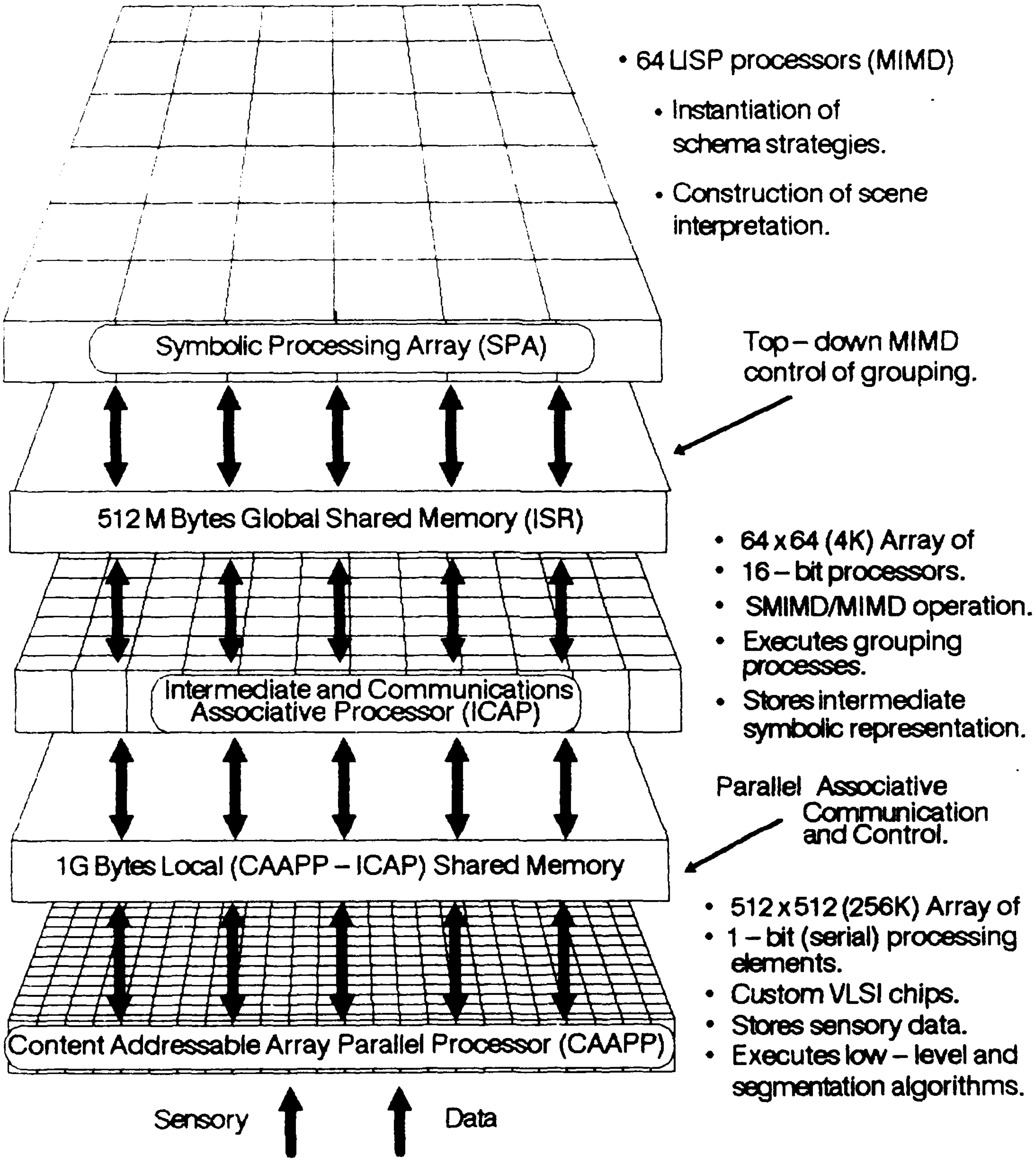

• 64 LISP processors (MIMD)
• Instantiation of schema strategies.
• Construction of scene interpretation.
Symbolic Processing Array (SPA)
Top – down MIMD control of grouping.
512 M Bytes Global Shared Memory (ISR)
• 64 x 64 (4K) Array of
• 16 – bit processors.
• SMIMD/MIMD operation.
• Executes grouping processes.
• Stores intermediate symbolic representation.
Intermediate and Communications Associative Processor (ICAP)
Parallel Associative Communication and Control.
1G Bytes Local (CAAPP – ICAP) Shared Memory
• 512 x 512 (256K) Array of
• 1 – bit (serial) processing elements.
• Custom VLSI chips.
• Stores sensory data.
• Executes low – level and segmentation algorithms.
Content Addressable Array Parallel Processor (CAAPP)
Sensory
Data

Figure 7. IUA Overview

At the high level, the IUA is purely a MIMD parallel processor. Additionally, the intermediate and low levels of the IUA may be treated in a variety of modes of parallelism

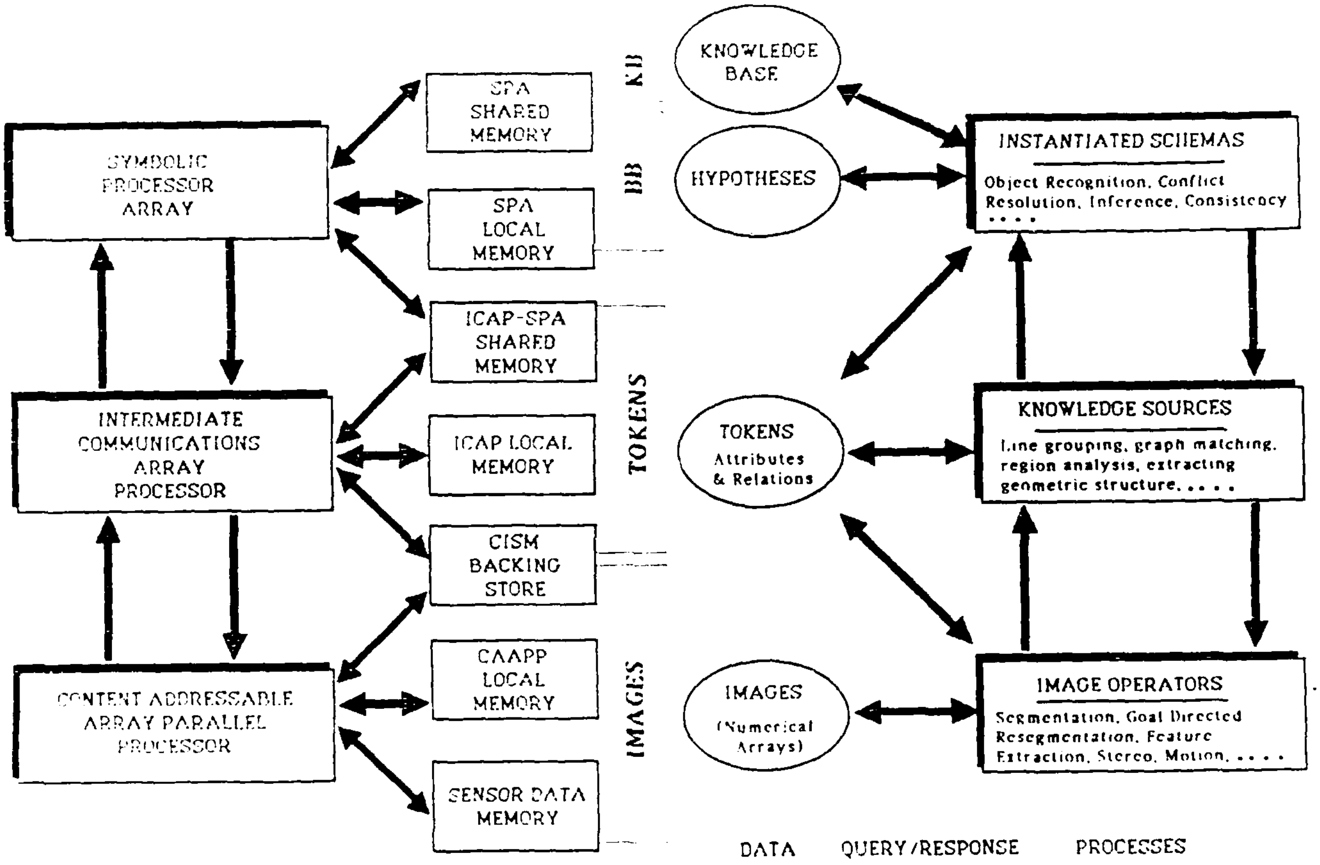


Figure 8. IUA Compared to UMass Visions System

to allow multiple hypotheses from the SPA to be evaluated in parallel at the lower levels. These include the CAAPP operating in pure SIMD or local-SIMD mode, and the ICAP operating in synchronous-MIMD or pure MIMD mode. A brief explanation of how the local-SIMD and synchronous-MIMD modes differ from the familiar SIMD and MIMD modes is required. In local-SIMD mode, the CAAPP cells excute in disjoint SIMD groups, with each group able to operate on locally broadcast values, and to locally compute its own summary values in parallel with all other groups. This allows different parameters to be employed in processing disjoint portions of the image, with all portions of an image being processed simultaneously. Local-SIMD differs from Multi-SIMD processing in that all processors receive the same instruction stream, whereas with Multi-SIMD the disjoint groups receive separate instruction streams. The prototype IUA does not support Multi-SIMD processing, but we are exploring several ways of adding that capability to the full scale IUA. In synchronous-MIMD mode, the programming paradigm is more like SIMD than MIMD: the ICAP processors execute the same program but have their own instruction pointers so that they can branch independently, and globally synchronize for each stage of processing. Synchronous-MIMD has the advantage of being as simple to program as a SIMD system but without the time penalty, usually encountered in SIMD systems, of having to sequentially execute all of the paths in a branching control structure.

We are currently building 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 specifically noted that a feature pertains only to the prototype.

4.1 The CAAPP (low) Level

The CAAPP is a 512 × 512 square grid array of custom 1-bit serial processors intended to perform low-level image processing tasks. The CAAPP is similar in many ways to CLIP-4 [Duff, 1978], MPP [Batcher, 1980], DAP [Hunt, 1981], GRID [Arvind, 1983], GAPP [Davis, 1984], and the Connection Machine [Hillis, 1986]. However, its architecture is especially oriented towards *associative processing* with an emphasis on fast global summary feedback mechanisms supported in hardware. The CAAPP is also specifically designed to interact with the ICAP in a tightly coupled fashion for both bottom-up and top-down processing. Thus, the CAAPP has been tailored to permit flexible control, to provide rapid feedback to the controlling processes so that they may exercise control in response to actual image properties, and to integrate fully into a hierarchically organized vision architecture.

The CAAPP processing elements are linked through a four way (S,E,W,N) communications grid which is augmented with circuitry that allows certain types of long distance communication to take place quickly. Each processor can execute an instruction in 100 nanoseconds and contains 5 one-bit registers, an ALU, data routing circuitry, and 320 bits of RAM that acts as an explicitly managed cache memory. Each element has access to a 32K-bit backing store memory that is dual-ported with the ICAP. The backing store is also referred to as the CAAPP-ICAP shared memory (CSM). The architecture of a CAAPP cell is shown in Figure 9, and a table of CAAPP instructions is shown in Figure 10. The custom VLSI chip layout containing 64 CAAPP PE's is shown in Figure 11. The

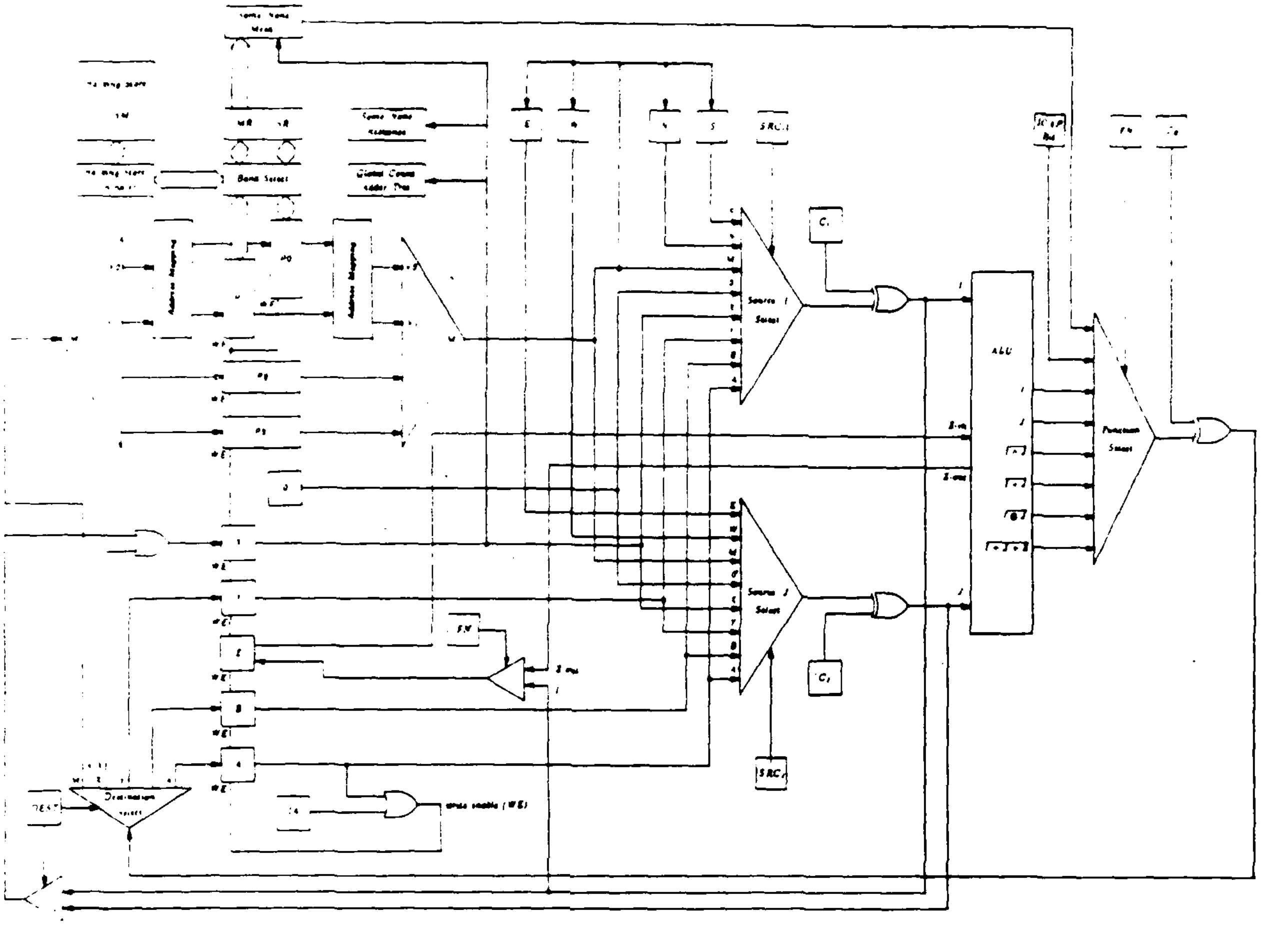


Figure 9. CAAPP Cell Architecture

CAAPP chip is being built in 2-micron CMOS via the DARPA MOSIS facility and will contain roughly 107,000 transistors.

The key to integrating the CAAPP into the IUA is its combination of associative feedback and control mechanisms. One of the principle feedback mechanisms in the CAAPP is the array-wide logical OR output, called Some/None, which indicates whether any CAAPP cells are in a given state represented by the response bit. At the end of each instruction cycle the global controller receives a Some/None signal for the full array, while the ICAP processors receive the Some/None indication for that portion of the CAAPP array connected to each of them.

A count of all responding cells is also available at the global controller. The counting operation is used to gather statistics about an image and the results of processing. For example, through counting we may quickly determine the mean and standard deviation of an attribute value for a given set of processors (see section 6.4). Each ICAP processor receives the count for the 8 × 8 subarray of the CAAPP associated with it.

31		27		23		18		13	9	8–0
0	INH	C_r	Ftn	C_i	S_i	C_j	S_j	$Dest$	0	$Address$

INH	(Inhibit)
0	Non-inhibit — always active
1	Inhibit if A = 0
2	Inhibit if A = 0 or S/N = Some
3	Inhibit if A = 0 or S/N = None

$C_{i,j,r}$	
0	TRUE
1	Complement $S_{i,j}$, *Result*

	S_i	S_j	Ftn	$Dest$	
0	zero-reg	zero-reg	$Some/None \Rightarrow Dest$	X_{pc}	0
1	C	C	$S_i \Rightarrow Dest$	A, X	1
2	S	E	$S_j \Rightarrow Dest$	$A, X \Leftarrow S_i$	2
3	N	W	$\overline{S_i \wedge S_j} \Rightarrow Dest$	$A, X \Leftarrow S_j$	3
4	Y	Y	$\overline{S_i \vee S_j} \Rightarrow Dest$	Y	4
5	X	X	$\overline{S_i \oplus S_j} \Rightarrow Dest$	X	5
6	B	B	$\overline{S_i + \overline{S_j}} + \overline{Z} \Rightarrow Dest$	B	6
7	A	A	$ICAP\ C \Rightarrow Dest$	A	7
8	memory	memory	$S_i \Rightarrow Z$	memory	8
9	—	—	$memory \Rightarrow MR$	—	9
10	—	—	$memory \Rightarrow MR, SB$	—	10
11	—	—	$MR \Rightarrow memory$	—	11
12	—	—	$MR, SB \Rightarrow memory$	—	12
13	—	—	—	—	13
14	—	—	—	—	14
15	—	—	—	—	15

Figure 10. CAAPP Cell Instructions

Communication among CAAPP cells may take place in four different ways. One way is through global feedback and rebroadcast. This method is used when all or most of the CAAPP processors must be told the value of one of the processors (e.g. broadcasting the maximum value so that all cells can normalize their values). A second way is via the ICAP; in some cases it is more efficient to transfer CAAPP data to the backing store and let the ICAP move it across the array and place it in the backing store of the appropriate CAAPP cell. The third way uses the nearest neighborhood (S,E,W,N) mesh, which allows

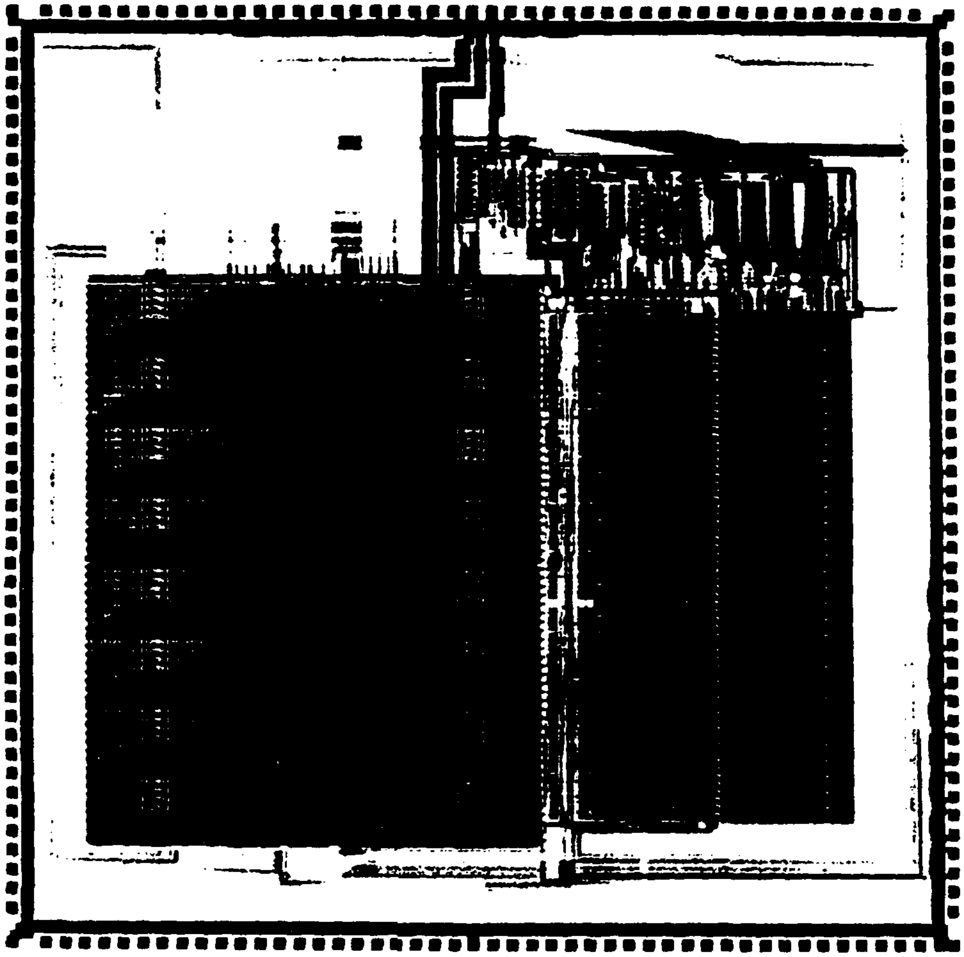


Figure 11. 64 Element CAAPP Test Chip

a CAAPP processor to read a bit from up to two of its neighbors at once. This is similar to the network employed in other mesh-connected SIMD parallel processors. The remaining communication mechanism is described in the next section.

4.2 The Coterie Network

The fourth means of communication among CAAPP processors involves a new and powerful variation on the nearest neighbor mesh called the Coterie network. This is similar to the reconfigurable buses proposed by Kumar [Kumar, 1987], Miller and Stout [Stout, 1986] and the polymorphic torus proposed by Li [Li, 1987], but differs in that it allows general reconfiguration of the mesh, and multiple processors to write to the mesh at the same time. By adding the simple switch network shown in Figure 12, it is possible, under program control, to create independent groups of processors that share a local associative Some/None feedback circuit. The isolated groups of processors can then respond to globally broadcast instructions in a locally data-dependent fashion,(i.e. local SIMD) which permits parallelism to be employed with more flexibility. For example,

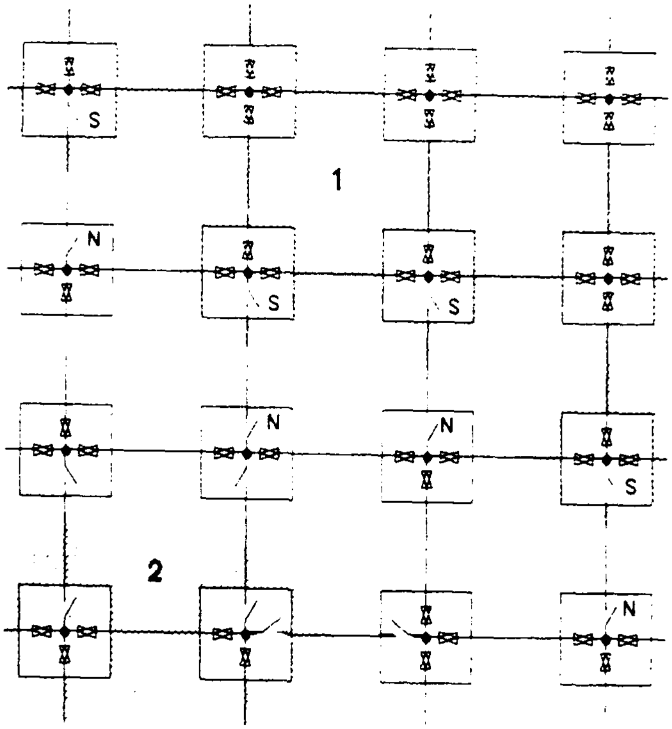


Figure 12. The Coterie Network

suppose that an image is divided into a large number of regions, and that we wish to determine some attribute for each of the regions. In a typical SIMD architecture this would be done by sequentially selecting each region for analysis or in parallel by complex communication between neighbors where the attribute is computed via a propagating wave that checks region labels at each step. However, using the coterie network in the CAAPP, in many cases, all regions can perform their own local evaluation in parallel without having to check region labels after one initial step of neighbor comparison.

The name Coterie Network is based upon the similarity of the isolated processor groups to a "coterie": that is, a group of people who associate closely because of common purpose, interests, etc. [Random, 1987].. The isolated groups of processors are thus referred to as coteries. Note that the Coterie Network is separate from the nearest neighbor mesh, which we refer to as the SEWN Mesh.

Creation of a set of coteries typically begins with opening all of the switches that link processors. Using the SEWN Mesh, the processors compare their own values with the values of their neighbors. They then close the switches that connect them to neighbors with similar properties, leaving open the switches that would connect them to dissimilar neighbors. Of course similarity can be defined by an operation such as a global broadcast of a threshold and a local comparison. In this way, processors with similar properties establish independent coteries. It should be fairly obvious to the reader that, among other things, each region of a segmentation could be a coterie of cells. Because the CAAPP processors can save and restore the switch settings that make up a set of coteries, it is possible to reconfigure the Coterie Network from one processor interconnection pattern to another by broadcasting a single instruction.

Within a coterie, there is a mesh of wire that all of the processors are connected to. Each active CAAPP processor may be instructed to output a bit onto its coterie's mesh and then read whatever bit value is currently on the mesh within its coterie. When more than one processor in a coterie tries to output a bit onto the mesh, the value that appears on the wire is the logical OR of the output bits of all of the processors in the coterie. The shared mesh is thus functionally equivalent to the global Some/None feedback circuit except that its output is locally formed and only available within a coterie. In addition to its associative feedback function, the coterie mesh can be used to broadcast a value from a single processor to every member of the coterie as in a Broadcast Protocol Multiprocessor [Levitan, 1984]. This is accomplished by first selecting a single processor within each coterie, using an associative search operation in parallel within all coteries. Subsequent instructions for placing a value onto the mesh will only be performed by these selected cells. However, all of the cells will perform the operations for reading the value that is on the mesh. In this way, the coterie network can be used for local broadcasting of data values. The local feedback and broadcast processes can occur in every coterie in parallel.

4.3 Inter-level Communication Between the CAAPP and ICAP

The principal mechanism for transferring data between the CAAPP and ICAP is the CISM (or backing store). Each ICAP processor has access to the 256K-byte block of memory that also acts as the 32K-bit backing store for each of the 64 CAAPP cells associated with an ICAP processor. Swapping between the CAAPP and CISM is accomplished by dual-porting a portion of the on-chip CAAPP memory. When data is moved between the CAAPP and the CISM it goes through an automatic corner turning mechanism that provides bit-serial data access to the CAAPP and byte-parallel access to the ICAP.

4.4 The ICAP (Intermediate) Level

The ICAP is designed to manipulate tokens (symbolic descriptions of extracted image events and their associated attributes) at the intermediate level and to support data base functions that allow access to these tokens by grouping processes running on the ICAP and by symbolic interpretation processes, running on the SPA processors. 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 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 is a square grid (64 × 64) array of Texas Instruments TMS320C25 16-bit digital signal processor chips. Each of the 4096 ICAP processors consists of a CPU, 256K bytes of local RAM, 384K bytes of dual-ported memory for interacting with the CAAPP and SPA, and network communications hardware. The ICAP processors operate at 5 million instructions per second and can perform a 16-bit multiply-and-accumulate operation in a single instruction time. In addition to its speed, a digital signal processor was chosen at the ICAP level because its instruction set and arithmetic capabilities are well suited for performing computations in spatial geometry. Three-dimensional geometric projections, computing distances, and matching operations are common operations that may be needed at the intermediate level of vision processing.

Control of the ICAP is provided by the ACU (in Synchronous-MIMD mode) and by the SPA (in MIMD mode). Once sensory events have been extracted and represented symbolically at the intermediate level in the ICAP (and continue to evolve as grouping operations take place), each of the SPA processors may then query the ICAP in parallel to establish and verify hypotheses. The ICAP provides three different global OR outputs available to the controller that can be used to determine the status of processing in the ICAP array. The choice of meaning for each signal is left up to the programmer. For example, the programmer may choose to have them indicate completion of a task in the ICAP array with or without exceptions. Another use is as an associative Some/None mechanism. A global summation mechanism is also provided that uses the global count hardware in the CAAPP to form a sum of an 8-bit value from each ICAP processor.

The horizontal links between the ICAP processors provide the intra-level communications necessary for grouping and merging processes to operate on token attributes and token relations within the intermediate symbolic representation. In the IUA prototype, which has only 64 ICAP processors, the bit-serial I/O links between the processors are connected through a centrally controlled 64 × 64 bit-serial crossbar switch. Thus it is possible to establish any point-to-point network topology in the prototype ICAP. Various methods of extending the ICAP communications network to the full-size ICAP array are under consideration.

4.5 Inter-level Communication Between the ICAP and SPA

The ICAP-SPA Shared Memory (ISSM) provides the principal communication path between the top two levels of the IUA. It is viewed as an I/O device by each ICAP processor. A given ICAP processor can write (or read) values to (from) an I/O buffer in the ISSM. The ICAP then initiates a block transfer between the I/O buffer and a page of its choosing in the ISSM RAM. An ICAP processor may only access the 128 K-byte segment of ISSM that is associated with it. However, each SPA processor has global access to the entire ISSM for all of the ICAP processors. This structure allows processes in the SPA to access the results of ICAP processing regardless of their spatial locations in the array.

4.6 The SPA: High Level Processing

The SPA processors will run a LISP-based blackboard system [Erman, 1980, Nii, 1986, Draper, 1987, 1988] through which the various knowledge-based processes can communicate while cooperatively constructing an interpretation of an image and determining the relationships of the various imae components to stored knowledge. From the point of view of the blackboard system, the CAAPP and ICAP will appear as knowledge sources at different levels of abstraction. Knowledge-based processes in the system can activate different processes in the CAAPP and ICAP either for the full array or for independent sub-arrays. Thus, the SPA processors operate in MIMD mode with communication through the blackboard. The detailed architecture of the SPA has not yet been fully defined. In the first prototype of the IUA, which is a 1/64th vertical slice of the full IUA, the SPA will be a single Motorola M68020 class processor, augmented with a symbolic co-processor. A separate research investigation within the UMass VISIONS project is currently exploring the implementation of cooperative algorithms and data structures using a commercially available shared-memory multiprocessor at the SPA level [Draper, 1988]. This experience is providing additional direction to the future scaling up of the IUA at the SPA level.

Currently, the full SPA is envisioned as consisting of 64 or more processors, each capable of running LISP. Each processor will have some local memory and will have access to a global shared memory that will include the ISSM and the blackboard. The shared memory decouples the SPA processors from the locality of information in the image.

4.7 The ACU: Controlling the CAAPP and ICAP

One major design goal for the Array Control Unit (ACU) was to maximize the rate at which instructions are issued to the CAAPP. This meant that the overhead for controlling loops, branches, and subroutine calls in the ACU had to be minimized. A second major design goal for the ACU was to minimize the cost of implementing a complete development environment for it. Preferably, the ACU would execute a commonly used instruction set so that software could be transported from an existing machine.

Clearly, the first goal required a custom processor, while the second goal dictated an off-the-shelf processor. The solution to this dilemma was to incorporate both into the ACU design. Thus, the ACU contains two separate processors that can issue instructions to the CAAPP (and control the ICAP as described below). The two processors are referred to as the Macro-controller and the Micro-controller.

The Macro-controller is a Motorola M68020-based system that brings with it the wide range of software tools that are available for that processor. It can issue instructions to the CAAPP in two ways. The simplest way is to take direct control of the instruction bus and write out data values that will be interpreted as instructions by the processor arrays: Even at its maximum rate, however, the 68020 can only issue instructions at about one-tenth of the rate that the CAAPP can execute them. The second method for the Macro-controller to issue instructions is to issue subroutine calls to the Micro-controller.

The Micro-controller is a custom processor, driven by horizontal microcode. It is capable of issuing an instruction to the CAAPP every 100 nanoseconds, with minimal overhead for loop, branch, and subroutine control. The Micro-controller will have a large library of CAAPP routines in its program memory, any of which can be called by the Macro-controller. When the Micro-controller completes execution of a CAAPP routine, it returns a status flag to the Macro-controller which may then issue a new call.

The routine-calling mechanism permits the user to write applications in a high-level language for the Macro-controller, and yet obtain good peak instruction rates for operations on the CAAPP. Although this does not provide 100% utilization of the CAAPP, it is reasonable to expect 50% utilization in many cases, which should be adequate for most research and development situations.

Although the only source of instructions for the CAAPP is the ACU, the ICAP processors each have their own program memory. The ICAP program memory is loaded with a large library of service routines upon system initialization. The way in which the ACU issues instructions to the ICAP is by storing a user program into ICAP program memory and then issuing an interrupt to the ICAP that causes it to jump to the user program. (The program is broadcast to all of the ICAP program memories in parallel.) An ICAP user program is typically just an execution script (written in C, Forth, or assembly language) of calls to the ICAP library. Thus, the ACU and ICAP interact very little when a program is running in the ICAP; the exception is when the ICAP program reaches a global synchronization point - this must be mediated by the ACU. The ACU can also set the ICAP to operate in MIMD mode, by turning control over to a task queuing program in the ICAP processors. The queuing program reads execution scripts from the ISSM according to a predefined protocol. When the ICAP is executing in MIMD mode, it depends upon the SPA to provide coordination of any required synchronization between ICAP processors.

The ACU thus supports the close interaction between the CAAPP and ICAP during the initial phases of interpreting an image. However, the ACU also permits the CAAPP and ICAP to work independently, with the ICAP taking directions from the SPA as the high level interpretation processes come into play. This allows the CAAPP to concurrently perform additional-low level processing, such as integrating information from other sensors or starting to process the next image.

5. IUA PROGRAMMER'S MODEL

Just as the IUA is a hierarchical architecture, the IUA programming model is also hierarchical. At the lowest level, a programmer may write assembly language primitives for the CAAPP, ICAP, and SPA. More typical, however, will be high-level programs for the SPA. Between these two extremes are programs for the ACU that control the CAAPP and ICAP arrays in cooperation with the SPA.

SPA programs are written for a shared-memory multiprocessor model, typically using a high level language with extensions to support concurrency. Because the SPA processors are all identical, any process may execute on any processor and have access to the entire intermediate level database (ISSM) and the blackboard. The ACU is interfaced to the SPA in such a way that it acts as a special SPA processor that has control over the CAAPP and ICAP arrays. Thus, ACU programs communicate with SPA processes via the same mechanisms that SPA processes use to communicate with each other.

To the application developer, the IUA is simply programmed with a set of concurrent processes, some of which perform high-level vision tasks and some of which service requests for CAAPP and ICAP processing. The server processes run in the ACU and view the CAAPP and ICAP as a pair of attached array processors.

The software interface to the CAAPP and ICAP takes two forms. An ACU program will usually just call library subroutines that cause the arrays to perform predefined operations. However, if an operation is required that isn't in the library, then a new subroutine must be written. CAAPP routines can be written inline as part of an ACU program via a series of calls to a subroutine that issues individual CAAPP instructions. CAAPP routines can also be written in assembly language, or a high level language and compiled as linkable modules. ICAP routines must be precompiled and downloaded to the ICAP program memory because the function of the ACU is primarily to issue SIMD instructions to the CAAPP. The ACU merely coordinates execution of routines that are already stored in the ICAP processors whenever they are operating in synchronous-MIMD mode, or interacting closely with the CAAPP.

When an application requires maximum performance from the CAAPP and ICAP, it must be micro-coded for execution on the micro-controller. The micro-controller is designed to be a complete processor, capable of executing general purpose programs. Thus, for real-time applications, a typical development scenario would involve rapid prototyping of an implementation on the Macro-controller, followed by migration of the high-level code into Micro-controller instructions. In the future, tools and compilers will be developed for the Micro-controller that will aid the code migration processes. However, it is reasonable to assume that the Macro-controller development environment will always be more attractive. Thus, it is expected that the two-stage development process will remain as the standard approach to implementing real-time applications on the IUA.

The IUA *programming* environment currently exists as a set of software simulators.

The ACU, CAAPP, and backing store portions of the simulator are available on VAX and SUN systems, while the full IUA simulator is running on an Explorer LISP workstation augmented by a TI Odyssey parallel signal co-processor board containing four TMS32020 processors. The simulators can be programmed in a variety of languages; LISP, Forth, C, etc., may all be used to write ACU programs that drive the CAAPP. The ICAP processors, however, are currently programmable only in Forth and assembly language, although a C compiler is under development by TI. The simulated SPA on the Explorer may be programmed in LISP or Prolog.

6. SAMPLE ALGORITHMS

While the purpose of this section is to provide a sense of the types and range of operations that can take place on the IUA, it is by no means a complete discussion of all of the system's capabilities. The algorithms presented here are specifically intended to demonstrate the various forms of communication that occur within and between the CAAPP and ICAP levels. A processing scenario that involves the SPA level is outlined in section 7.

This section will begin with several fairly simple but detailed algorithms in order to show how the IUA is programmed. The algorithmic notation used is very close to one of the programming languages employed with the IUA software simulators. However, the notation is simplified to improve the clarity of the presentation. Macro operations are also used where their machine language implementation is not an important element of the algorithmic method. For example, adding two 8-bit quantities in the CAAPP is actually performed bit-serially by a sequence of instructions, but a typical program will make use of the standard macro for addition to perform this operation. Following the detailed algorithms, several more algorithms will be sketched whose complexity makes them too lengthy to be presented here in great detail. The concepts behind the algorithms are worth considering, however, because they demonstrate additional capabilities of the IUA.

6.1 Select Greatest Responding Value

This algorithm (Figure 13) demonstrates the use of the associative Some/None feedback from the CAAPP array. The goal is to select, from among all active cells, the cell or cells that have the greatest value in a given field of their memory. In addition, that value is to be made available in the ACU for subsequent processing.

The algorithm begins by loading the high order bit of a given field into the response register of all active cells. The global controller then tests the Some/None output of the array. If any cells have their high order bit set, then they are candidates for the maximum value, in which case, any cells that have a zero 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 all still potential candidates. This process repeats with each successively lower order bit

```
-- Beginning with the high-order bit
FOR Bit := Field Length - 1 DOWN TO 0 DO
        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}
```

Figure 13. Finding a Maximum Value in the CAAPP

in the field. When the low order bit has been processed, only those cells that contain the maximum value will remain active. For each iteration, the controller saves the Some/None response so that the maximum value is available in the controller at the conclusion of processing. This takes 24 CAAPP instruction cycles (2.4 microseconds) for an 8-bit value.

6.2 Label Connected Components

The local associative Some/None operation provided by the Coterie Network is demonstrated by this algorithm. Because finding a maximum uses only broadcast and Some/None feedback, it can be performed locally within a coterie and in parallel with every other coterie. This leads to the simple algorithm for computing a connected component labeling of an image shown in Figure 14.

The algorithm begins by loading each processor with its address in the array from the backing store memory, which serves to give each processor a unique number. Next, the Coterie Network switches are opened between processors that are on region boundaries (i.e. between pairs of processors that have different values), establishing a coterie for each image region. Lastly, all regions in parallel determine their local maximum address value. Note that this is the same algorithm as for finding a maximum value in the entire array except that the coterie Some/None response is used in place of the global Some/None response to control the setting of activity. As part of finding the maximum, every processor in a coterie stores the maximum address value for all cells in its coterie in its own memory. Because this value is different for every region, the result is that each connected group of processors is assigned a unique label that is common to every processor within a group. From our electrical simulations of the Coterie Network, we calculate that this algorithm will take approximately 50 microseconds to execute.

6.3 Histogram

The preceding algorithms use only the Some/None Response form of feedback. The

```
Load Processor Addresses

Coterie Switches := (Open, Open, Open, Open)                         {Initialize coterie switches}
FOR Neighbor := North TO West DO                                     {Initialize flag for each neighbor}
    Equal := True
    FOR Bit Num := Field Length -1 DOWN TO 0 DO                      {For each bit in field}
        Equal := Equal AND (Neighbor.Field Bit Num = Field Bit Num)   {Compare own bit with neighbor}
    IF Equal                                                         {If field value matches neighbor, bit for bit}
        THEN                             {Then close the coterie switch to connect with that neighbor}
            Coterie Switch [Neighbor] := Closed
FOR Bit Num := Address Length -1 DOWN TO 0 DO
                                                                     {Find maximum addresses in coteries}
    Response := Address Bit Num                                      {Put bit in response register}
    IF Coterie 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}
        [illegible] Bit Num := Coterie Some                          {Save bit values for component label}
```

Figure 14. Connected Component Labelling using the Coterie Network

response count is of equal importance in many of our algorithms. For example, we can form a histogram of any numerical feature in the CAAPP using the response count. This is quite simple to do: For each bucket in the histogram we associatively select those cells whose values fall within the range of the bucket by broadcasting the minimum and maximum value of the range, comparing with the cell's value, and appropriately setting the response register to 0 or 1; then a count of the responding cells gives the histogram bucket value. Thus, the time to form the histogram is proportional to the number of buckets in the histogram (typically about 1.6 microseconds per bucket), and is independent of the number of values in the array.

6.4 Compute Average Value

Figure 15 gives a CAAPP algorithm that uses the response count to compute the mean of the values stored in selected cells. The algorithm begins by summing the values in the selected cells. Starting with the high order bit position of the values to be summed, each bit of the values in the selected cells is separately counted. The counts are each added to the overall sum after being appropriately scaled by a power of two. The algorithm concludes by setting each cell's response bit equal to its activity bit so that the response count will be the number of active cells, and dividing the sum by that count to get the mean of the values.

6.5 The Sobel Edge Operator

Of course, in addition to processing that is oriented around associative feedback, the CAAPP is able to perform the usual image processing and low level vision algorithms that do not depend upon feedback to the controller. For example, smoothing operators such as Gaussian convolution, edge detectors such as the Sobel and Canny operators, local pixel comparisons, line curvature, border following, etc. all use the mesh connected operations

```
                                                              {Initialize sum}
Sum: 0
FOR Bit Num: High Order DOWN TO Low Order DO   {Count each bit in field and add to sum, scaling appropriately}
        Response: Field Bit Num
        Sum: Sum*2 + Response Count
                                                      {Count number of active cells}
Response: Activity
Mean: Sum Response Count                                    {and compute mean}
```

Figure 15. Computing the Mean of Values in Selected CAAPP Cells Using the Response Count Operation

that are typical of this class of machines.

To demonstrate communications between neighboring CAAPP PEs, the algorithm for performing a Sobel operation is shown in Figure 16. The Sobel computes the local X and Y gradient magnitude at each pixel in an image. These X and Y magnitude vectors subsequently can be combined to form the orientation and magnitude of the local gradient at each pixel. Pixels with large gradient magnitudes are frequently associated with strong edges or lines in an image, and therefore are likely to be of use in interpreting an image.

The Sobel operator requires that the image be convolved with two different 3 × 3 masks. One mask computes the gradient magnitude in the X direction while the other computes the magnitude in the Y direction. The masks are:

```
{Compute X Magnitude}
    Double Own := Own Value + Own Value
    Int_Result := Double_Own + North(Own_Value)
    Int_Result := Int_Result + South(Own_Value)
    X Magnitude := East(Int Result)
    X Magnitude := X Magnitude - West(Int Result)
{Compute Y Magnitude}
    Int_Result := Double_Own + East(Own_Value)
    Int Result := Int Result + West(Own Value)
    Y_Magnitude := North(Int_Result)
    Y_Magnitude := Y_Magnitude - South(Int_Result)
```

Figure 16. Sobel Algorithm for the CAAPP

X magnitude:

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

Y magnitude:

$$\begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{matrix}$$

In the CAAPP, the X magnitude is computed by first having each cell double its own value, and then add the values of its North and South neighbors. This intermediate result is then used to compute the actual X magnitude by subtracting the intermediate value of each cell's West neighbor from the intermediate value of its East neighbor. A similar sequence of operations is used to compute the Y magnitude.

Assuming that the operation is being applied to 8-bit integer pixels, it would require 100 CAAPP instruction cycles (10 microseconds) to compute the components of the Sobel operator for the image. The gradient magnitude and orientation can be computed from these components by applying the standard formulas as a sequence of CAAPP arithmetic operations.

```
Label Connected Components             {Each component is given a unique label that is equal to the maximum cell address
                                        within the component. The label is stored in a field called Component Label in
                                        each cell                                                                        }
Start ICAP Corner List Builder;        {A process is started in the ICAP that will respond to each Signal ICAP operation
                                        (except the first) by picking up a corner from the backing store, and adding it to
                                        the list for the appropriate component region                                    }
Activity := 1                                                                                     {Turn on all cells}
Leader := Address = Component Label                                          {Identify the leader for each coterie}
Backing Store.Write(Leader);                                                  {Copy leader tags to backing store}
Response := Leader
Latch Local Count                                                           {Count of leaders in each CAAPP chip}
Signal ICAP                            {The ICAP processor associated with each CAAPP chip reads the local count to
                                        determine the number of coteries for which it will collect corners. Each ICAP
                                        processor also scans its portion of the backing store to determine the addresses of
                                        the leaders in the coteries associated with it                                   }
Activity := Response := Corner Tag                                                    {Activate all corner cells}
WHILE Some DO                                                 {Select corner with greatest address in each coterie}
    FOR Bit Num := Address Length - 1 DOWNTO 0 DO
        Response := Address[Bit Num]
        Next Corner[Bit Num] := Coterie Some!    {Store each bit of the greatest address in all members of the
                                                  coterie (including the leader) by ignoring the activity bit     }
        IF Next Corner[Bit Num] THEN
            Activity := Response                                   {Turn off cells that are less than the max}
    Corner Tag := False                                         {Disable the greatest corner once it's found}
    Backing Store.Write(Next Corner)                                  {Copy the address to the backing store}
    Signal ICAP                                  {The ICAP processor associated with each CAAPP chip picks
                                                  up the address stored in each coterie leader location in its por-
                                                  tion of the backing store and saves it in a list               }
    Activity := Response := Corner Tag!                                           {Activate remaining corners}
END WHILE
```

Figure 17. Creating Lists of Border Corners

6.6 Create Border Corner Lists

In addition to performing associative Some/None operations, the Coterie network may be used to pass messages across the mesh by providing direct links between non-adjacent processors. In the algorithm shown in Figure 17, it is assumed that some corner detection operation has been performed on the borders of regions in the image. The result is a sparse set of processors that are labelled as corners (ie. those processors whose Corner_Tag field is set to true). One useful feature that can be extracted for each region is a list of the positions of its corners. The following algorithm forms the corner lists for all regions in parallel. It begins by making each region an independent coterie, using the connected components labelling algorithm presented earlier. A single cell in each coterie is selected as the coterie leader. In this case, the chosen cell is the member of the coterie whose cell address equals the region's component label. The leader is responsible for collecting the corners for its region, and passing them to the ICAP which stores them in a list.

The first corner is determined by selecting the corner cell in each coterie with the maximum address. As part of this process, the coterie leader learns that cell's address and passes it to the ICAP processor associated with the CAAPP chip containing the leader. The selected corner cell is then shut off and the process is repeated so that the next corner is selected. The loop ends when there are no more corners to select, at which point every corner will have been passed to the ICAP by its coterie leader.

This algorithm causes the corner lists to be created in reverse raster-scan order, which is adequate if the regions are simple convex figures. However, for more complex regions, it may be difficult to reconstruct the shape of a region given a corner list in this order. A better arrangement is to list the corners in clockwise (or counterclockwise) boundary traversal order (i.e. the order in which corners would be encountered as the cells at the boundary of the region are traversed, starting from some arbitrary boundary point). The coterie network can also be used to accomplish this task. Although the basic concept for doing this is quite simple, in practice it is complicated by considerations of regions that are one-pixel wide and regions that completely enclose other regions. Because the algorithm is more complex, it is only discussed here in general terms; the detailed discussion will be left to a future paper.

As in the preceding algorithm, the first step is to label connected components. After connected component labelling has been performed, each member of a component examines its neighborhood to determine whether any neighboring cell has a different component label. Any cell that has a neighbor belonging to a different region is at the boundary of its own region. In the simplest case, the cells that are at a region's boundary form a chain that are at a region's boundary for a chain that is a one-pixel wide closed loop. Each cell in the chain will have two neighbors; one in the clockwise direction and the other in the counterclockwise direction around the loop. The cells that make up a boundary chain can then set their coterie switches so that the chain becomes a separate coterie. (Some of the complexity of the actual algorithm stems from the situations in which a boundary chain is not a simple closed loop and how the different cases are handled.)

A leader is selected for each of the boundary-chain coteries. Each leader opens the coterie switch connecting it to its counterclockwise neighbor in the loop. The loop is thus transformed into an open figure with the leader at one end. Every cell in the chain that is also tagged as a corner now opens the switch connecting it to its clockwise neighbor in the chain. Next, each leader broadcasts a bit to its coterie. Because the corner cells have broken the coterie, the bit will only reach the first corner clockwise along the boundary from the leader. That corner is then activated and transmits its address back along the coterie to the leader which subsequently passes the address to the ICAP. The active corner then closes the switch to its clockwise neighbor and deactivates itself so that further broadcasts from the leader will pass through it. The process is repeated until all corner cells have been read out to the ICAP. Note that all region boundaries are being processed simultaneously through their coterie chains.

6.7 Region Adjacency Graph

A similar algorithm involves collecting a list of adjacent region labels for each region in an image. This algorithm begins by having every boundary cell get the label of its neighbor that is in another region. using the SEWN mesh. Each region then performs a coterie-select-greatest operation on these region labels, and a region label is output to the ICAP via the coterie leader. All boundary cells that have the selected label are then shut off and the test is repeated to obtain the next region label. The process is complete when there are no more labels to output. Because a region label is the address of the coterie leader for a region. and the ICAP processors are spatially collocated with respect to the CAAPP cells. each ICAP processor can directly compute the ID number of the other ICAP processors that contain descriptions of adjacent regions.

6.8 Rule-Based Region Merging

Once the ICAP processors have collected the information required to describe a specific type of image event, for example lines or regions, the ACU can broadcast rules (or constraints) to the ICAP that cause it to take some action. Given that the ICAP contains an attribute list for each region consisting of its size, average intensity, list of border corners, and list of adjacent regions, the ACU could broadcast a rule to the ICAP that is equivalent to the statement: "If a region is below size X, and is adjacent to one or more regions that exceed size Y, it should be merged with the adjacent region whose intensity differs least from its own, but only if the intensity difference is less than threshold Z."

Such a compound rule will actually take the form of a processing script that is downloaded to the ICAP processors from the ACU via a broadcast to the ICAP program memories. The script will actually be a series of calls to library routines that have been pre-stored in the ICAP. In this case, the script would select regions larger in size than threshold Y; and have their associated processors transmit their size, intensity, and ID to all ICAP processors that are associated with an adjacent region. The ICAP processors then compare each region that is smaller than X to the size and intensity of each region for which information was received. If the condition for a merge is met, then an ICAP processor transmits all of the information for the smaller region to the processor that is responsible for the larger region. The processor that contains the larger region adds the smaller region's information to its database. The processor that contains the smaller region transmits the new region label to the CAAPP by writing the label into the backing store for the region's coterie leader. The ACU then instructs all coterie leaders that have received new labels to broadcast the new label to their coterie and then resign as coterie leader. The cells that are on the common boundary between the two regions then close their coterie switches so that the two coteries are merged into one. The ICAP processor that was responsible for the smaller region then deletes the region's information from its database.

A variety of interprocessor communication topologies can be supported in the ICAP, due to the flexibility of the centrally controlled network switch. For example, topologies such as the mesh, ring, hypercube, or shuffle can be built with the ICAP switch. In this algorithm the mesh topology would be used because the communication tends to be local in nature. (Information that is to be sent between non-adjacent ICAP processors is relayed by those processors that are between them.) In order for an ICAP processor to communicate with its four neighbors, it sends to one neighbor while receiving from the neighbor in the opposite direction. Each time the ICAP is ready to switch directions, it signals the ACU which changes the ICAP connection pattern and resynchronizes the serial ports on all of the ICAP processors. Thus, all ICAP processors might initially transmit to the North and receive from the South. When all transmissions to the North are finished, the ACU changes the connection pattern so that all messages will be transmitted to the West and received from the East, and then signals the ICAP processors to start transmitting again.

7. A VISION PROCESSING SCENARIO FOR THE IUA

The discussion that follows describes one possible sequence of operations on the IUA that could be used to form an interpretation from an image. This is actually a gross oversimplification of how the UMass VISIONS system is used to interpret an image [Draper, 1987]. It would be impossible to provide a complete discussion of the full interpretation process on the IUA within the space available in this paper. Our purpose here is merely to show the types of processing and interactions that can take place in the IUA in a context that is larger than a single algorithm.

The processing is initiated with a region segmentation of the image. The first step is to apply an edge-preserving smoothing operator. We use an algorithm which involves a few iterations of a 3×3 window convolution on the CAAPP. The next step is a region segmentation [Beveridge, 1987 also in this issue], which uses local histograms within 16×16 windows. In brief, each ICAP computes a histogram for the 8×8 tile of CAAPP cells associated with it. This is done by broadcasting a series of value range comparisons and performing local count operations in the CAAPP. The ICAP simply records the local count for each range corresponding to buckets in the histogram. The ICAPs then merge their histograms through communication with their horizontal neighbors in the same 16×16 window. (Actually, the algorithm utilizes windows that overlap by 4 pixels in each direction, forming 24×24 pixel histograms and requiring a bit more complex communication at the CAAPP and ICAP levels than we have room to discuss here.) Next, the ICAPs search their histograms for peaks and valleys, applying various criteria for defining clusters of values. The ICAPs communicate with their neighbors to consistently extract labels of peaks to be associated with pixels, and generate a cluster label plane that is returned to the CAAPP through the backing store. The CAAPPs form connected components within 16×16 windows, and then perform region merging to remove the artificial seams of the 16×16 windows in order to produce the final segmentation.

Another part of the interpretation process involves line extraction. We use the Burns straight line algorithm [Burns, 1986] which begins by applying two 3×3 convolution windows to compute the Sobel gradient operator on the original image in the CAAPP (this step can actually be done in parallel with ICAP processing for the region segmentation). Edges are assigned coarse orientation labels by broadcasting a table of orientation ranges to the CAAPP processors. When a processor's value falls within an orientation range that is being broadcast, it stores the associated orientation label. Using the Coterie Network, a connected components labelling algorithm is run on the orientation label plane producing regions of pixels with similar gradient orientations, each with a unique label. Short lines (ie. regions with a small set of pixels of similar orientation) that result from this process are associatively selected and then saved for later use as a texture measure. The parameters describing the remaining "long" lines are transferred to the backing store so that the ICAP has access to them. The ICAP processors then compute for each region the parameters of a representative line by fitting a planar intensity surface to the pixel values in the region. The ICAP array links collinear segments that may have resulted from excessive fragmentation of longer lines in the original image. The result is a set of tokens that describe straight lines of various lengths which correspond to events in the image.

The next phase of processing results in the construction of a feature database for the extracted tokens-the Intermediate Symbolic Representation (ISR)-that is stored at the ICAP level. The CAAPP and ICAP together compute various feature values that describe the regions and lines that have been extracted from the image. For example, some features associated with a line might be its length, orientation, the contrast across it, adjacent regions, end points, etc.

At this point, the ICAP essentially takes over the processing. Our simulations indicate that the preceding operations take on the order of 20 milliseconds to perform. During the remainder of the scenario the CAAPP is free to receive another image and begin to do similar or other types of low-level processing in a pipeline fashion. This could involve stereo or motion analysis, or simply preparing the next frame for merging with the token database.

Next, the ICAP applies sets of object constraints retrieved from the knowledge base in the SPA shared memory to the tokens in the ISR that are resident in ICAP memory in order to form initial object hypotheses. Constraints are minimum and maximum values on token attributes that form a range on accepting tokens as object hypotheses [Hanson, 1987]. For each hypothesis, a score and threshold are generated within each ICAP for its token set. During this phase the ICAP processors are essentially running in MIMD mode.

The next major step involves geometric grouping of lines based on collinearity, parallelism, and orthogonality to abstract more complex geometric structures. The ICAP returns to synchronous-MIMD operation in order to facilitate the exchange of information between ICAP processors.

The last phase uses the results of the previous two phases to extend hypotheses, detect conflicts between them, and resolve those conflicts. It is at this point that the SPA takes a greater role. The ICAP processors transfer selected token labels to the memory that is shared with the SPA. The different object schemas [Draper, 1987] in the SPA apply various grouping strategies to the tokens by issuing commands to the ICAP processors that refer to the token labels and object verification strategies. Through a global blackboard the schemas incrementally attempt to resolve conflicts and find a consistent set of hypotheses with proper spatial and spectral relationships. Algorithms from the previous phases may be selectively repeated with different parameters and for different goals as different strategies for object verification are applied in different areas of the image to arrive at a consistent interpretation of the scene.

8. IMAGE UNDERSTANDING BENCHMARK PERFORMANCE

Figure 18 shows estimated execution times for the IUA on a suite of image understanding tasks that comprise the first DARPA Image Understanding Benchmark [Rosenfeld, 1987]. It should be noted that these tasks only exercise the CAAPP and ICAP levels of the IUA and, except for one instance, the tasks do not require inter-level communication. As such, they can only be viewed as isolated processing steps that will not give any real test of the integrated processing required for an interpretation. An Integrated Image Understanding Benchmark has been defined to address the issue of inter-level communication [Weems, 1988].

The times that are shown in Figure 18 are based, as much as possible, on execution of programs with in the instruction-level simulator. In several instances, however, fully simulating execution of the task was too time consuming, and we thus elected to simulate key portions of the task and extrapolate the total execution time.

The first two tasks begin with an 8-bit digital image of size 512 × 512, pixels. The first task involves convolution with an 11 × 11 mask and detection of zero crossings in the resultant image. The second task requires lists of the border pixels, as defined by the zero-crossings, to be output. Both of these tasks are performed by the CAAPP level of the IUA.

The next pair of tasks begins with a 1-bit digital image of size 512 × 512 pixels. In the first of these two tasks, all connected components which have an image value of 1 are grouped into regions that are assigned unique positive-integer labels. The second task involving the binary image requires the formation of the Hough transform of the input image, with the result being an array of size 360 × 512 where the first dimension represents theta and the second represents rho. Both of these tasks are also performed exclusively by the CAAPP.

The next three tasks begin with a set of 1000 real coordinate pairs defining 1000 points in a plane. The output of the first task is the convex hull of the set of points. The second task results in the Voronoi diagram for the points. The third task requires construction of the minimal spanning tree across the set of points. The ICAP level of the IUA was used exclusively for the convex hull and minimal spanning tree tasks. For the Voronoi diagram

	Task	Execution Time (milliseconds)	Processor
1.	Convolution, zero crossing	0.2	CAAPP
2.	Output border list	0.2	CAAPP
3.	Connected components	0.05	CAAPP
4.	Hough transform	27.0	CAAPP
5.	Convex hull	15.0	ICAP
6.	Voronoi diagram	50.0	ICAP/CAAPP
7.	Minimual spanning tree	124.0	ICAP
8.	Visibility of vertices	290.0	ICAP
9.	Minimum cost path	1.0	ICAP

Figure 18. Image Understanding Benchmark Performance

task we elected to use an algorithm that required the CAAPP and ICAP to work together (Other algorithms exist that could be run at either the CAAPP or ICAP level alone.) To facilitate execution by the CAAPP, we also chose to convert the real input representation (for which no precision was specified) into an internal 24-bit fixed-point representation. In the other tasks, it was assumed that a real value was in IEEE 32-bit floating point representation.

The vertex visibility task begins with 1000 triples of triples of real coordinates in the range of 0 to 1000. These coordinates specify the vertices of 1000 opaque triangles in three-dimensional space. The task requires a list of all vertices that are visible from the origin to be output. This task is performed using only the ICAP level of the IUA.

The last task in the benchmark required that a minimum cost path be found between two points in a graph with 1000 vertices of order 100 where the edges of the graph are weighted by non-negative real values. Only the ICAP level of the IUA was used in performing this task.

As can be seen from the figure, the IUA performed quite well on this suite of tasks. These timings do not take into consideration the fact that two third of the IUA was left idle in every case but one. The IUA could easily execute several of these tasks concurrently without even making use of the SPA level.

9. CURRENT IMPLEMENTATION STATUS

At present a 1/64th scale demonstration prototype of the IUA is being built at Hughes Research Labs. This vertical slice of the machine is scheduled for completion in late 1988

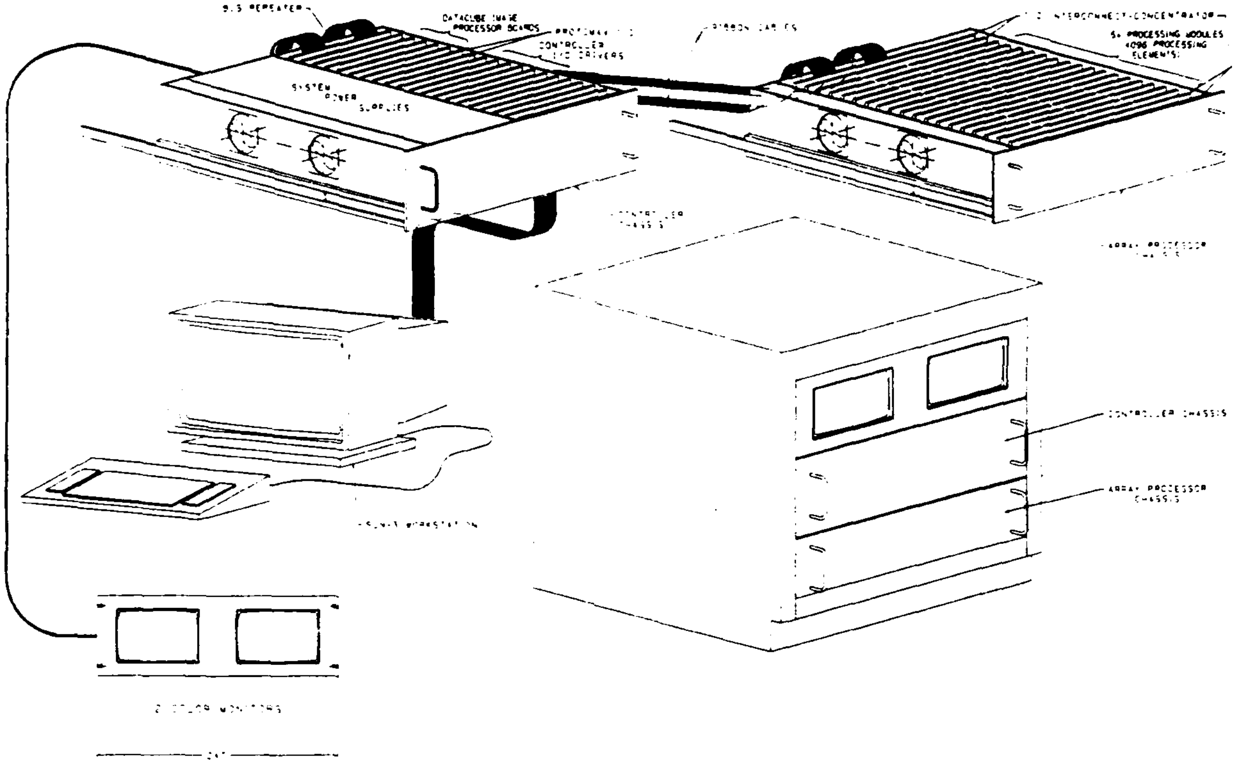


Figure 19. Image Understanding
Architecture Test Bed

and will include 4096 CAAPP cells, 64 ICAP cells, a single SPA processor and the ACU. The entire prototype will plug into a single-user workstation that will serve as a host (Figure 19).

The prototype will physically consist of a 16 by 20 inch mother-board with 69 daughter-boards attached to it. Sixty-four of the daughter-boards are 4.5 by 3.5 inches and the remaining three are 20 by 3.5 inches in size. The 64 smaller boards contain the ICAP and CAAPP processors and their memories. The five larger daughter boards provide the controller interface, feedback concentration, ICAP communications network switching, and clock and signal buffering. The mother-board also includes a dual ported frame-buffer memory that allows high speed image input and output.

Each processor daughter-board will contain a single custom VLSI chip, a TMS320C25, 256K bytes of static RAM, 384K bytes of dual-ported dynamic ram, and tri-state bus buffers. The single custom chip holds the 64 CAAPP processors with their local memories, the backing store controller, a refresh controller for the dynamic RAM, and arbitration logic for the various devices that must access the bus of the associated ICAP processor.

10. FURTHER DEVELOPMENT

Several refinements are being considered for future versions of the machine, in particular to the ICAP communication structure and the CAAPP to ICAP control interface. One possibility being explored is to give the ICAP the ability of interacting with a local CAAPP controller. The CAAPP would then be able to execute in Multi-SIMD mode at the chip level: each 8 × 8 section could then execute a separate instruction stream as an isolated SIMD processor. Another extension would be to augment the ICAP circuit switched network that is centrally controlled with a distributed control routing network. Finally, we are also examining the implications of augmenting the SPA shared memory with a Content Addressable Input Output Memory (CAIOM) [Levitan, 1984] to facilitate blackboard management.

11. CONCLUSION

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 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.

Just as the IUA is a hierarchical system, the programmer's model of the IUA is also hierarchical. At the lowest level, a programmer may write assembly language support routines for the CAAPP, ICAP, and SPA (CAAPP support routines are actually written in microcode for the Micro-controller portion of the ACU). At the intermediate level of the programmer's model are applications written for the Macro-controller portion of the ACU. The highest level of the model, where the majority of applications are written, takes the form of concurrent processes that execute in the SPA using a shared-memory multiprocessing model of computation.

A collection of algorithms has been presented that demonstates some of the ways that the CAAPP and ICAP levels of the IUA may be programmed. In particular, processing modes have been demonstrated that use global associative processing; local associative processing via the Coterie Network; horizontal communication via the SEWN mesh, coterie network, and ICAP centrally switched network; vertical communication via the local count and CISM; and the SIMD, Multi-SIMD, and Synchronous-MIMD forms of parallelism.

Beyond simply testing our hardware design, our ultimate goal for the prototype is to provide a powerful interim development environment for image understanding and parallel processing research. A simulated parallel processor has simply been too slow to permit any significant amount of experimentation. Once our prototype is running, we will be able to perform more total computation in the first few minutes of execution time than we have been able to do in our previous five years of simulation.

12. ACKNOWLEDGEMENTS

This research was supported, in part, by the Advanced Research Projects Agency of the Department of Defense via Contract No. F49620-86-C-0041 monitored by the Air Force Office of Scientific Research and under Contract No. DACA76-86-C0015 monitored by the U.S. Army Laboratories. We would also like to thank Dr. Caxton C. Foster for his long-term contribution of ideas in content addressable memories and processing that provide a foundation to this work.

13. REFERENCES

[Annarontone, 1987] Annarontone, M., Arnold, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., and Webb, J.A., The Warp Computer: Architecture, Implementation, and Performance, IEEE Trans. on Computers, Vol. C-36, No. 12, December, 1987.

[Arvind, 1983] Arvind, D.K., Robinson, I.N., and Parker, I.N., A VLSI Chip for Real-Time Image Processing, Proc. IEEE International Symposium on Circuits and Systems, 1983, pp. 405-408.

[Batcher, 1980] Batcher, K. E., Design of a Massively Parallel Processor, *IEEE Trans. Comp.*,Vol. C-29, No. 9, September 1980.

[Beveridge, 1987] Beveridge, J.R., Griffith, J., Kohler, R.R., Hanson, A.R., Riseman, E.M., Segmenting Images Using Localized Histograms and Region Merging (in preparation).

[Burns, 1986] Burns, J.B., Hanson, A.R., Riseman, E.M., Extracting Straight Lines, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:425-455, 1986.

[Crowther, 1985] Crowther, W., Goodhue, J., Starr, E., Thomas, R., Milliken, W., Blackadar, T., Performance Measurements on a 128-Node Butterfly Parallel Processor, Proc. of the Intl. Conf. on Parallel Processing, 1985, pp. 531-540, IEEE Computer Soc. Press, Washington, DC.

[Davis, 1984] Davis, R., Thomas, D., Geometric Arithmetic Parallel Processor-Systolic Array Chip Meets the Demands of Heavy-Duty Processing, Electronic Design, October 31, 1984, pp. 207-218.

[Draper, 1987] Draper, B.A., Collins, R.T., Brolio, J., Griffith, J., Hanson, A.R., Riseman, E.M., "Tools and Experiments in the Knowledge- Directed Interpretation of Road Scenes", Image Understanding Workshop Proceedings, Morgan Kaufmann, Los Altos, CA, 1987.

[Draper, 1988] Draper, B.A., Collins, R.T., Brolio, J., Griffith, J., Hanson, A.R., Riseman, E.M., The Schema System: Knowledge Based Vision (in preparation).

[Duff, 1978] Duff, M.J.B., Review of the CLIP Image Proceeding System, Procedings of the National Computer Conference, 1978, AFIPS, pp. 1055-1060.

[Duff, 1986] Duff, M.J.B. (ed.), Intermediate-Level Image Processing, Academic Press, London, 1986.

[Encore, 1986] Encore Computer Corp., promotional literature, Marlborough, MA, 1986.

[Erman, 1980] Erman, L., et al., The Hearsay-II Speech-Understanding System: Integrating

Knowledge to Resolve Uncertainty, *Computing Surveys*, 12:213-253, 1980.

[Foster, 1976] Foster, C. C., *Content Addressable Parallel Processors*, New York: Van Nostrand Reinhold, 1976.

[Hanson, 1984] Hanson, A. R. and Riseman, E. M., Preprocessing Cones: A Computation Structure for Scene Analysis, COINS Technical Report 74C-7, University of Massachusetts at Amherst, September 1974.

[Hanson, 1978a] Hanson, A. R., Riseman, E. M., VISIONS: A Computer System for Interpreting Scenes. In: *Computer Vision Systems*, A. R. Hanson, and E. M. Riseman (Eds.), New York: Academic Press, 1978, pp. 303-333.

[Hanson, 1978b] Hanson, A. R., Riseman, E. M., Segmentation of Natural Scenes. In: *Computer Vision Systems*, A. R. Hanson, and E. M. Riseman (Eds.), New York: Academic Press, 1978, pp. 129-163.

[Hanson, 1980] Hanson, A. R., Riseman, E. M., Processing Cones:A Computational Structure for Scene Analysis for Image Analysis. In: *Structured Computer Vision*, S. Tanimoto and A. Klinger (Eds.), New York: Academic Press, 1980.

[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.

[Hanson, 1987] Hanson, A.R., Riseman, E.M., From Image Measures to Object Hypotheses, Submitted to IEEE PAMI.

[Hillis, 1986] Hillis, D.W., The Connection Machine, MIT Press, Cambridge, 1986.

[Hunt, 1981] Hunt, D.J., The ICL DAP and its Application to Image Processing, in Languages and Architectures for Image Processors (M.J.B. Duff, S. Levialdi eds.), Academic Press, London, 1981.

[Levitan, 1984] Levitan, S. P., *Parallel Algorithms and Architectures: A Programmers Perspective*, Ph.D. Dissertation, Computer and Information Science Department, also, COINS Technical Report 84-11, University of Massachusetts at Amherst, May 1984.

[Levitan, 1987] Levitan, S. P., Measuring Communication Structures in Parallel Architectures and Algorithms. In: *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, and R. Douglass (Eds.), MIT Press, Cambridge, 1987.

[Nagin, 1982] Nagin, P.A., Hanson, A.R., Riseman E.M., Studies in Global and Local Histogram-Guided Relaxation Algorithms. IEEE Trans. Pattern Analysis and Machine Intelligence, 4:263-277, 1982.

[NCube, 1985] NCube Corp, Promotional literature, Beaverton, OR, 1985.

[Nii, 1986] Nii, H.P., The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, AI Magazine, Vol 7, Number 2, pp.38-53.

[Parma, 1980] Parma, C. C., Hanson A. R., and Riseman, E. M., Experiments in Schema-Driven Interpretation of a Natural Scene, COINS Technical Report 80-10, University of Massachusetts at Amherst, April 1980. Also in: *NATO Advanced Study Institute on Digital Image Processing*, R. Haralick and J. C. Simon (Eds.), Bonas, France, 1980.

[Pfister, 1985] Pfister, G., Brantley, W., George, D., Harvey, S., Kleinfelder, W., McAuliffe, K., Melton, E., Norton, V., Weiss, J., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, Proc. of the Intl. Conf. on Parallel Processing, 1985, pp. 764-771, IEEE Computer Soc. Press, Washington, DC

[Random 1987] The Random House Unabridged Dictionary of the English Language, Flexner, S.B., Hauck,L.C.H., Eds., Random House, N.Y., 1968.

[Rattner, 1985] Rattner, Justin, Concurrent processing: A new direction in scientific computing, Proceedings of the National Computer Conference, 1985.

[Reynolds, 1984] Reynolds, G., Irwin, N., Hanson A. R., and Riseman, E. M., Hierarchical Knowledge-Directed Object Extraction Using a Combined Region and Line Representation, *Proc. of the Workshop on Computer Vision: Representation and Control*, Annapolis, Maryland, April 30 - May 2, 1984, pp. 238-247.

[Rosenfeld, 1986] Rosenfeld, A., The prism machine: An alternative to the pyramid, J. Parallel and Distributed Computing 3:404-411.

[Rosenfeld, 1987] Rosenfeld, A., A Report on the DARPA Image Understanding Architectures Workshop, Proceedings of the 1987 DARPA Image Understanding Workshop, Morgan Kaufmann, Los Altos, CA, 1987.

[Seitz, 1985] Seitz, Charles L., The Cosmic Cube, Communications of the ACM, 28-1, January 1985, pp 22-33.

[Sequent, 1986] Sequent Computer Systems, promotional literature, Beaverton, OR, 1986

[Tanimoto, 1983] Tanimoto, S., A Pyramidal Approach to Parallel Processing, *Proc. 10th Annual International Symp. on Computer Architecture*, Stockholm, Sweden, June, 1983

[Uhr, 1972] Uhr, L., Layered Recognition Cone Networks that Preprocess, Classify and Describe, *IEEE Trans. Comp.*,21:758-768, 1972.

[Weems, 1984a] Weems, C. C., *Image Processing on a Content Addressable Array Par-*

allel Processor. Ph.D. Dissertation Computer and Information Science Department, also, COINS Technical Report 84-14, University of Massachusetts at Amherst, September 1984.

[Weems, 1984b] Weems, C. C., Levitan, S. P., Foster, C. C., Riseman, E. M., Lawton, D. T., and Hanson, A. R., Development and Construction of a Content Addressable Array Parallel Processor (CAAPP) for Knowledge-Based Image Interpretation, *Proc. Workshop on Algorithm-Guided Parallel Architectures for Automatic Target Recognition*, Leesburg, VA, July 16-18, 1984, pp. 329-359.

[Weems, 1985] Weems, C. C., The Content Addressable Array Parallel Processor: Architectural Evaluation and Enhancement. *Proc. IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, New York, October 7-10, 1985. pp. 500-503.

[Weems, 1988] Weems, C.C., Hanson, A.R., Riseman, E.M., Rosenfeld, A., An Integrated Image Understanding Benchmark: "Recognition of a 2 1/2D "Mobile", Proc. IEEE Conf. on Computer Vision and Pattern Recognition, Ann Arbor, MI, June 5-9, 1988.

[Weymouth, 1986] Weymouth, T. E., *Using Object Descriptions in a Schema Network for Machine Vision*, Ph.D. Dissertation, Computer and Information Science Department, also, COINS Technical Report 86-24, University of Massachusetts at Amherst, 1986.

Appendix B:
IUA Functional Specifications

IUA Functional Specification March 10, 1988

1 Terminology

- SPA — Symbolic Processing Array Processor

  This is one of the processors in the 64 grid array of high level processors that does schema processing of the data processed by the CAAPP and ICAP levels of the machine.

- Mother Board

  The Mother Board provides connections for 64 Daughter Boards. This consists of 64 ICAPs and 4096 CAAPP PEs.

- Daughter Board

  The Daughter Board contains one CAAPP Chip, one ICAP, and associated memory and control.

- ICAP — Intermediate Communications Associative Processor

  This is one of the processors in the 4096 grid array of intermediate processors that transform the data processed by the CAAPP level of the machine. The ICAP PE is a Texas Instruments TMS320C25 DSP microprocessor using the Harvard Architechure.

- CAAPP Chip — Content Addressable Array Parallel Processor VLSI Chip

  The CAAPP Chip contains a 64 grid array of CAAPP PEs and logic used to control the memories on the CAAPP Daughter Board.

- CAAPP PE — CAAPP Processing Elements

  Each PE is a bit-serial processor designed to do low level processing on an individual pixel of an image. Neighborhood operations are also provided. All 512 × 512 PEs run in SIMD mode.

- GCU — Global Control Unit

  The Global Control Unit provides a control and queuing server for the SPAs so that they can cause processes to run at the CAAPP and ICAP levels of the machine. This is accomplished by providing the ACU with a list of processes to execute.

- ACU — Array Control Unit

  The array control unit provides the SIMD control at the CAAPP level and, therefore, is responsible for sending every instruction to all the CAAPP Chips and CAAPP PEs in the system. It is also capable of providing the array of ICAPs with requests to run processes at a near SIMD mode. User programs are executed by the ACU.

- BSM — Backing Store Memory

  The Backing Store Memory provides a large memory for backing up the small memory available to each CAAPP PE. It also provides a path for communication between the ICAP on each Daughter Board with the PEs on the Daughter Board.

- IPM — ICAP Program Memeory

  This is the memory used by the ICAP for ICAP Programs.

- IDM — ICAP Data Memory

  This is the memory used by the ICAP for data storage.

- ISSM — ICAP/SPA Shared Memory

  This memory is accessible by the SPAs as direct memory and is writable and readable by the ICAPs. There is a portion of this memory on each Daughter Board.

- IB — Instruction Bus

  The Instruction Bus connects the Array Control Unit with each Daughter Board. This is a broadcast bus from the ACU to each Daughter Board. A bi-directional bus is not used so that no time need be spent for hand-shaking.

- DB — Daughterboard Bus

  TheDaughterboard Bus connects the ICAP with its memories.

- HB — Host Bus

  Connects the host to the motherboard.

- CIOB — CAAPP I/O Bus

  Connects the external I/O subsystem to the CAAPP.

- SB — SPA Bus

  The SPA Bus connects each SPA with its memory.

2 Instruction Hierarchy

A program residing in the Array Control Unit (ACU) may be composed of CAAPP PE, CAAPP Chip, ACU Macro, and ACU Local instructions.

- ACU Local Instructions

  These instructions control the Array Control Unit. They perform operations on local registers, provide control logic, etc.

- ACU Macro Instructions

  These instructions cause a sequence of CAAPP PE, and CAAPP Chip instructions to be issued on the Instruction Bus (CB). The sequence is controlled by the ACU Micro-sequencer and it issues these instructions to the bus at a high sustained rate.

- CAAPP Chip Instructions

  These instructions are passed down the Instruction Bus un-modified and control functions on each CAAPP Chip such as Backing Store Memory operations.

- CAAPP PE Instructions

  These instructions are passed down the Instruction Bus un-modified. Each instruction is executed by each PE in every CAAPP Chip (under the control of the activity field).

- IUA Motherboard Instruction

  These instructions control functions on the motherboards such as chip select and router connections.

3 Array Control Unit

The following status outputs are available from the Daughter Boards. These status lines are ORed together so that while there are 64 daughter boards, there is only one line returning to the ACU. The status is set when ANY daughter boards have set their corresponding line.

- S — Some/None

  This line is logical sum of all the CAAPP PE X registers.

- B — Backing Store Complete Status

  This line is set when each CAAPP Chip completes its Backing Store Memory operation.

- IC — ICAP Comparand

  This is the logical sum of the ICAP Comparand sent by each ICAP to the CAAPP Chip on its Daughter Board.

- EF — memory test error flag

  This line is set by any CAAPP Chip which detects an error when storing data from the ACU into ICAP memory.

- D_0, D_1, D_2 — ICAP Done Status Outputs

  There is one done status bit corresponding to each ICAP interrupt level.

- H — ICAP (DB Bus) Hold Acknowledge

  This line is set when the ICAP on each Daughter Board has been disconnected from its memory (DB) bus.

Only five status lines return to the ACU. The H status and S/N status are always available. The D_n status outputs share the same lines with the EF, BSM, and IC status outputs. The ACU selects which status outputs are gated on these lines by sending an instruction on the Instruction Bus.

4 UMIUA Motherboard

The full UMIUA contains an array of 8 × 8 Motherboards. Each UMIUA Motherboard provides the connections for an 8 × 8 array of Daughter Boards. The Motherboard also contains the logic used for selecting individual Daughterboards, routing messages between ICAPs, and I/O control for image data[1].

4.1 Motherboard Memory

Each Motherboard contains the following memories:

- Video Memory
- Column Chip Select Register

  This 8 bit register has the column enabled bit for each column of Daughter Boards on the Motherboard. The corresponding bit drives an enable line to each CAAPP Chip in the column. If the enable line is not high, the output of each PE ALU on the CAAPP Chip is not routed to the destination. This results in the PEs remaining inactive for every instruction until the line is set high.
- Row Chip Select Register

  This 8 bit register has the row enabled bit for each row of Daughter Boards on the Motherboard. The corresponding bit drives an enable line to each CAAPP Chip in the column. If the enable line is not high, the output of each PE ALU on the CAAPP Chip is not routed to the destination. This results in the PEs remaining inactive for every instruction until the line is set high.
- Router Connection Register

[1] Some of this logic is physically placed on special boards connected to the Motherboard

4.2 Motherboard Instructions

31		28		23	19	15		7	0
1	1	0	*Ftn*	P_1	P_2	P_3	P_4	P_5	

	Ftn	P_1	P_2	P_3	P_4	P_5
0	Router_Write		Connection			Address
1	Router Start					
2	Chip_Select		Board	MBL		MBH
.	.					
.	.					
31	Extended					

Figure 1: Motherboard Chip Instructions

These instructions are executed by logic on each Motherboard:

- Router_Write(Connection,Address)

  This instruction writes 16 bits of connection data to one of 256 connection registers on each Motherboard.

- Router_Start()

  This instruction causes the data buffered by each ICAP to be routed through the router network. It essentially generates a synchronization signal detected by logic on each CAAPP Chip so that the chip knows when to start sending and receiving data bit serially from its 16 bit buffer.

- Chip_Select(Board,MBL,MBH)

  This instruction loads the Column Chip Select Register or Row Chip Select Register on the Motherboard specified by the **board** argument in conjunction with **MBL** and **MBH**. The **board** argument specifies the row or column. The **MBH** and **MBL** arguments specify the bits to be placed in the registers.

board	Selects	**board**	Selects
0000	C_0	0100	R_0
0001	C_1	0101	R_1
0010	C_2	0110	R_2
0011	C_3	0111	R_3

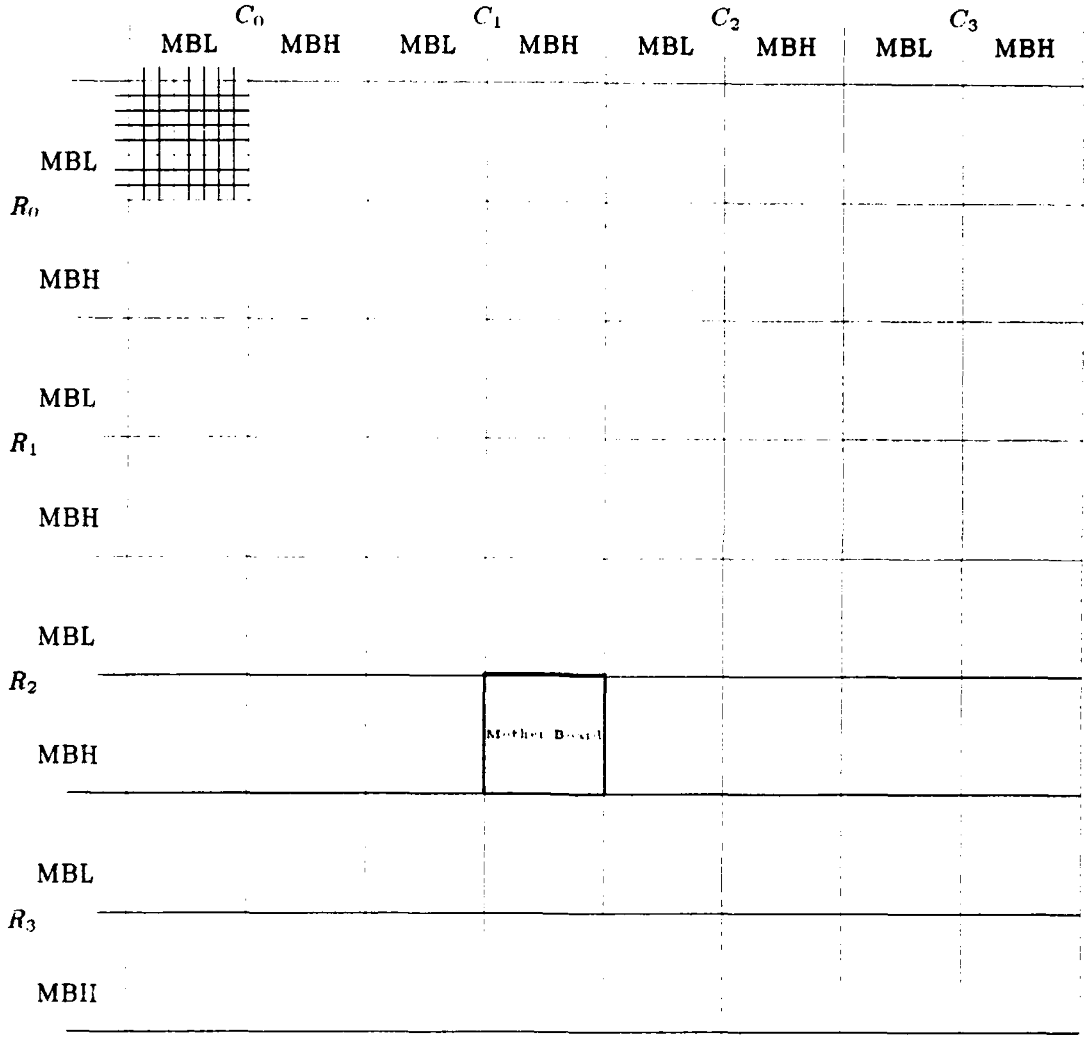


Figure 2: Motherboard Chip Select

5 UMIUA Daughter Board

The UMIUA Daughter Board contains the CAAPP Chip with 64 CAAPP PEs, the ICAP memory, ISSM, and the logic for integrating the CAAPP Chip, the ICAP, and the memories that are on the daugther board. One part of this logic is the set of finite-state-machines that control these memories.

5.1 Daughter Board Memory

The Daughter Board contains the following memories:

- IPM

  This contains the program memory for the ICAP DSP computer. There are 64k 16-bit words in this memory. It is writable both by the ICAP and by the Array Control Unit via the Instruction Bus and the Daughterboard Bus.

- IDM

  This memory contains the data memory for the ICAP DSP computer. There are 64k 16-bit words in this memory. This memory is writable both by the ICAP and by the Array Control Unit via the Instruction Bus and the Daughterboard Bus.

ICAP Adress Range	Maps to
0000_{16}–$F9FF_{16}$	ICAP Data Memory
$FA00_{16}$–$FBFF_{16}$	Reserved.
$FC00_{16}$–$FDFF_{16}$	Backing Store Memory
$FE00_{16}$–$FFFF_{16}$	Reserved

- Backing Store (BSM)

  The Backing Store Memory is arranged as 128k 16-bit words of memory using video RAM memory chips. The backing store memory can be thought of as 256 blocks of 512 words each. At any particular time, one of these blocks is available as ICAP data memory and is directly addressable by the ICAP. The particular block selected for ICAP access is specified by the ICAP Backing Store Select (IBS) register which is loaded by the ICAP referencing I/O port F_{16}. Transfers between the CAAPP chip and the backing store are accomplished through the serial side of the video RAM memory. Blocks are transferred to/from the serial side and the random access memory. The BSM is refreshed by logic on the Daughter board.

- ISSM

  The ICAP SPA Shared Memory is also organized using video ram memory chips. The dynamic access side is available as SPA memory while the ICAP does transfers using the serial side. The ISSM is refreshed by logic on the Motherboard.

Register	Size	Use
	Bits	
Backing Store related		
BSA	4	Backing Store Address
BSE	4	Backing Store End Address
BSP	5	Backing Store PE Counter
BSCS	4	Backing Store Column Select
BSBA	8	Backing Store Block Address
BSB	64	Backing Store Buffer
BSD	1	Backing Store Done
ICAP related		
IBS	8	ICAP Backing Store Select
ITCA	8	ISSM Transfer Control Block Address
ITCS	8	ISSM Transfer Control Column Select
TRF	2	ISSM Transfer Control Mode
ID	3	ICAP Done
LCR	8	Local Count Register
ICR	8	ICAP Count Register
IC	1	ICAP Comparand
ACU related		
AR	16	Address Register
CR	8	Count Register
PE related		
PR	1	Page Register

Figure 3: CAAPP Chip Registers

6 CAAPP Chip

The CAAPP Chip also contains logic that controls some functions on the chip that are not associated with a particular CAAPP PE. The most notable of these functions is the Coterie Network which requires some additional logic to control.

6.1 CAAPP Chip Registers

Each CAAPP chip has the following registers:

- BSA

  The Backing Store Address register contains the byte number of the next byte in the CAAPP PE memory to be transferred. After each byte is transferred, this register is incremented.

- BSE

  The Backing Store End Address register contains the byte number of the last byte to be transferred. The transfer continues until the BSE matches the BSA.

- BSP

  The Backing Store PE register contains the number of the next CAAPP PE pair for which the transfer is being done. PE bytes are transferred in pairs — (0, 32), (1, 33), etc.

- BSCS

  The Backing Store Column Select contains the column select value for the serial buffer of the Backing Store Memory Video RAM. It is incremented by one every time the BSA register is incremented.

- BSBA

  The Backing Store Block Address register contains the block number of the block in the Backing Store Memory Video RAM to/from which the serial buffer will be transferred.

- BSB

  The Backing Store Buffer is used for assembling data to be transferred to/from the backing store memory (BSM) serial buffer from/to the CAAPP PE V_0 memory.

- BSD

  The Backing Store Done register latches the Backing Store Done status for return to the ACU.

- IBS

  This register holds the block number of the block to which ICAP data accesses to addresses $FC00_{16} \ldots FDFF_{16}$ go. This register is set or read by the ICAP referencing I/O port F_{16}.

- ITCA

  This register holds the block number of the block in the ISSM to which the ICAP will be transferring data. This register is set using the ITA port of ICAP IO memory.

- ITCS

  This register contains the column in the serial buffer of the ISSM Video RAM memory which will be used by ICAP data transfers. This register is written using the ITA port of ICAP IO memory.

- TRF

  This register records the type of ISSM transfer that must be done when the SPA bus is tri-stated at some later time. This register is written using the ITC port of ICAP IO memory.

mode	action
00	do nothing
01	Pseudo-write
10	Write
11	Read

- ID

  This register latches the three ICAP done bits for access by the ACU.

- LCR

  The Local Count Register holds the CAAPP chip response count so that it is available to the ICAP.

- ICR

  The ICAP Count Register holds an eight bit value written to by the ICAP so that it may be gated into the response count network.

- IC

  The ICAP Comparand (IC) register is setable by the ICAP as an IO operation. This bit is latched on the chip and is available to all the PEs on the chip vis the $ICAP_C \Rightarrow Dest$ instruction.

- AR

  The Address Register contains the address in the ICAP IO memory, IDM, or IPM that will be used for transfers when these memories are being written to by the ACU.

- CR

  The Count Register is gated onto the response count network. The register is loaded from either the ICAP Count Register or with the CAAPP chip local count.

- PP

  The Page register controls which physical page of PE memory is referenced as virtual page 0 and virtual page 1 for PE references, backing store transfers, and Mesh Register loading.

6.2 CAAPP Chip Instructions

31			28	23	19	15		7	0
1	0	1	*Ftn*	P_1	P_2	P_3	P_4	P_5	

	Ftn	P_1	P_2	P_3	P_4	P_5
0	Reset		Mode			
1	BSM_Read		Start	End	CS	Address
2	BSM_Write		Start	End	CS	Address
3	Interrupt_ICAP		Interrupt			
4	Select Response		Source			
5	Set PR		Page			
6	Latch_Count		Source			
7	Latch_Local Count					
8	Inject_GCN		Direction			
9	.					
10	Hold ICAP					
11	Connect_To		Memory		Address	
12	Write				Data	
13	Release_ICAP					
14	Gather		Group/Page	Byte		Address
15	Scatter		Group/Page	Byte		Address
.	.					
.	.					
31	Extended					

Figure 4: CAAPP Chip Instructions

These instructions are executed by logic on the CAAPP Chip:

- Reset(Mode)

  The Reset instruction may reset all CAAPP PEs and/or all ICAPs. The following functions are performed by a reset:

Mode			Effect
x	x	1	The CAAPP PEs are reset.
			PR is set to zero:
			addresses 0 . . . 127 are mapped to P_0,
			addresses 128 . . . 255 are mapped to P_1.
			LCR, ICR, and CR count registers are set to zero.
			Various other internal states are reset.
x	1	x	The ICAPs are reset by a reset line to each ICAP.
1	x	x	The backing store controllers are reset.
			BSS, BSA, BSE, BSP are set to 0.

- BSM_READ(Start,End,CS,Address)

The **BSM_Read** instruction transfers a 512 word by 16-bit block from the Backing Store Memory (BSM) to the serial buffer in the Backing Store, and then transfers bytes from that serial buffer into PE memory.

The **start** field is loaded into the Backing Store Address (BSA) register. It specifies the beginning byte number in V_0 of PE memory that will receive the bytes from the serial buffer.

The **end** field is loaded into the Backing Store End Address (BSE) register and specifies ending byte number in V_0 of PE memory.

The **address** field specifies which of the 256 512-word blocks from the BSM is to be transferred.

The **CS** field is placed into the Backing Store Column Select (BSCS) register. The serial buffer is viewed as 16 columns with 1 byte per PE in each column. The **CS** field specifies the starting column to be transferred into PE V_0.

- BSM_Write(Start,End,CS,Address)

The **BSM_WRITE** instruction transfers bytes to the serial buffer from PE memory and then transfers the 512 16-bit block from the serial memory to a block in the Backing Store Memory (BSM).

The **start** field is loaded into the Backing Store Address (BSA) register. It specifies the beginning byte number in V_0 of PE memory that is transferred to the serial buffer.

The **end** field is loaded into the Backing Store End Address (BSE) register and specifies ending byte number in V_0 of PE memory.

The **address** field specifies which of the 256 512-word blocks in the BSM to which the serial buffer is written.

The **CS** field is placed into the Backing Store Column Select (BSCS) register. The serial buffer is viewed as 16 columns with 1 byte per PE in each column. The **CS** field specifies the starting column to be written to from PE V_0.

- Interrupt_ICAP(Interrupt)

This instruction raises an interrupt line to the ICAP. The interrupt line raised is either Int0, Int1, or Int2 depending on the value of the **Interrupt** field.

- Select Response(source)

  This instruction is used to select which Daughter Board status lines are to be gated back to the ACU.

source	Status selected
0	$D_0, D_1, D_2, H, S/N$
1	$EF, BSMD, IC, H, S/N$

- Set_PR(Page)

  This instruction specifies the page to which CAAPP PE instruction addresses 0...255 map. The **Page** field is a value from 0 to 1. The Page (PR) register is set to this value.

- Latch Count()

  This instruction latches either the CAAPP count (**Source** = 0) or the ICAP Count (ICR) register (**Source** = 1) to the Count (CR) register on the CAAPP chip.

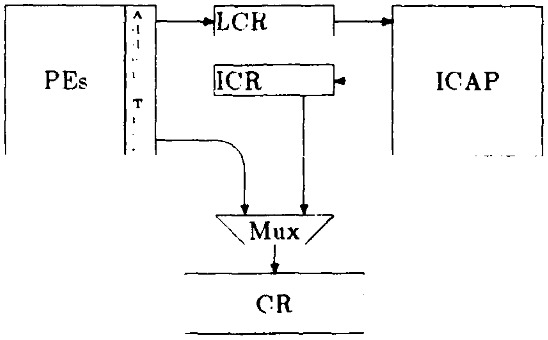


Figure 5: Count Network

- Latch_Local_Count()

  This instruction latches the CAAPP count to the Local Count (LCR) register so that it may be read by the ICAP.

- Inject GCN(Direction)

  This instruction activates t. : Coterie Network. The direction specifies in which direction the mesh is activated. See Figure 6.

Direction Field		Direction of Mesh
0	NE	North East
1	SE	South East
2	SW	South West
3	NW	North West

- Hold ICAP()

  This instruction places the ICAP into *ICAP Hold* mode so that the bus can be used by the Array Control Unit to write into memories on the ICAP Bus using the **connect to**, **write**, and **release ICAP** instructions. The amount of time required for the bus to become available varies depending on what state each individual ICAP is in. Because of the way the ICAP enters *ICAP Hold* mode it is necessary to insert some other instructions between the **hold_ICAP** and the first **write** instruction, otherwise the **write** will be ignored. This instruction raises $\overline{ACUICAPHold}$. An ICAP Hold Acknowledge signal is returned.

- Connect_To(Memory,Address)

  This instruction sets up the registers needed for transfering data to memories on the ICAP Bus. The **Write** instruction can be used to transfer data into one of the various data memories on the daughter board. The **memory** field controls whether the I/O address space, IDM data space, or IPM program space is to be used and is loaded into the Memory Select (MS) register. The **address** field value is loaded into the Address (AR) register which is used to specify where the data will be written.

  The **memory** argument specifies which memory is used for subsequent transfers.

Instruction Bit			
18	17	16	Memory selected
0	0	0	ICAP Program Memory
0	0	1	ICAP Data Memory
0	1	X	ICAP I/O Space
1	0	0	Test Mode for ICAP Program Memory
1	0	1	Test Mode for ICAP Data Memory
1	1	X	Test Mode for ICAP I/O Space

  Writing to the IBS register is disabled.

  No access from the ACU to the BSM is allowed as this would slow down access speed due to the different type of memory used (Video RAM memory) and is not necessary because the ACU can have the ICAP do the transfers.

- Write(Data)

  This instruction writes the **data** field value to the selected memory at the location specified by the Address (AR) register. After the write, the AR register is incremented.

  Writes to IDM above address $FA00$ write into memory that is not accessible by the ICAP.

  No read is provided due to the problems of reading from many (Daughter Boards) to one (ACU) and the need to make the Instruction Bus as fast as possible. Data can be accessed at higher levels by writing it to the the ISSM.

- Release ICAP()

  This instruction releases the ICAP and the DB bus.

- Gather(group,page,byte,address)

This instruction causes 64 individual bits in 64 separate PEs to be gathered together into 8 bytes stored in eight PEs. The **group** argument specifies which set of eight PEs receives the bytes and is specified by bits 19...17 of the instruction. The **page** argument specifies whether V_0 or V_1 is used to receive the bytes and is specified in bit 16 of the instruction. The **byte** argument specifies which byte in the selected page receives each byte of the Gather operation. The **address** argument specifies where in page P_2 of each PE the bits are obtained.

- Scatter(group,page,byte,address)

  This instruction causes eight bytes stored in eight PEs to be scattered into 64 individual bits in 64 separate PEs. The **group** argument specifies from which set of eight PEs the bytes are obtained and is specified by bits 19...17 of the instruction. The **page** argument specifies whether V_0 or V_1 is used to obtain the bytes and is specified in bit 16 of the instruction. The **byte** argument specifies which byte in the selected page is used to obtain each byte of the Scatter operation. The **address** argument specifies where in page P_2 of each PE the individual bits are placed.

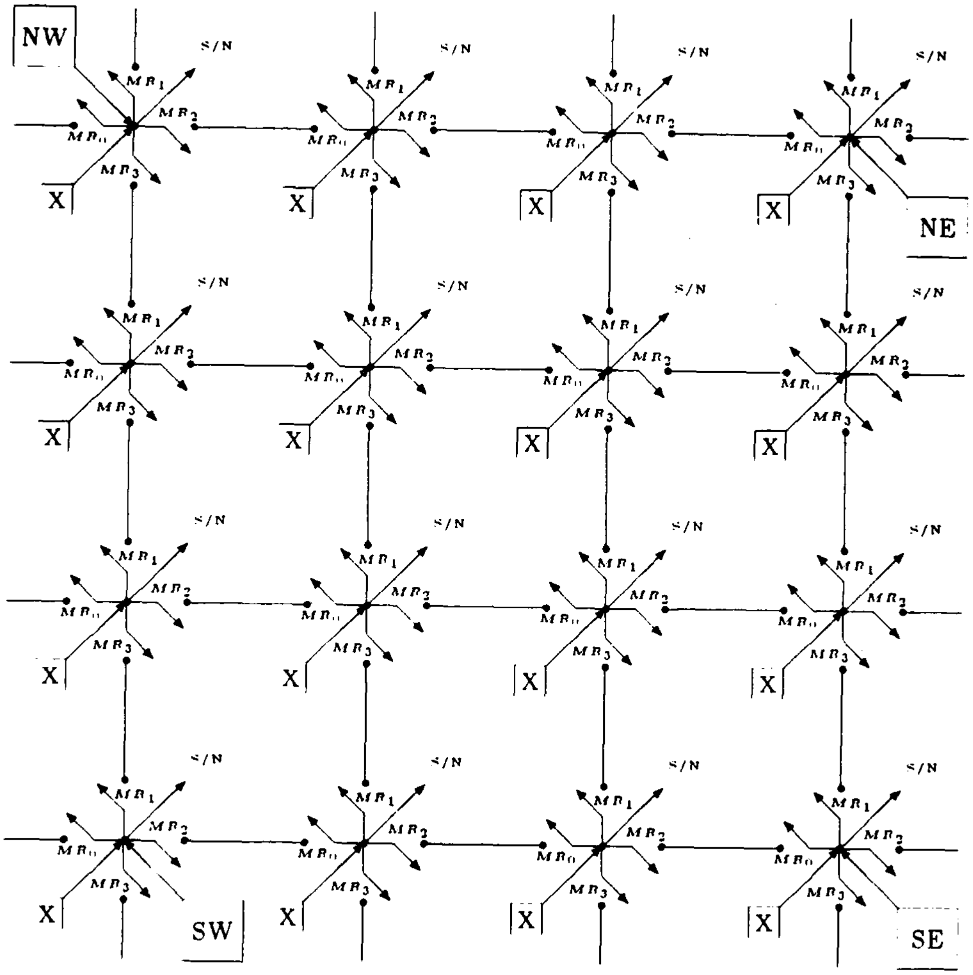


Figure 6: Coterie Network

Register	Size Bits	Use
ZERO	1	constant 0
A	1	Activity
B	1	General Purpose
C	-	Local Comparand
X	1	Status
Y	1	General Purpose
Z	1	Carry
N	-	North
E	-	East
S	-	South
W	-	West
MR	4	Mesh Register
SB	4	Static Buffer

Figure 7: CAAPP PE Registers

6.3 CAAPP PE

Each CAAPP Chip contains 64 CAAPP PEs arranged in an 8 × 8 array. Each PE is a bit-serial processor which has its own local memory and a set of registers.

6.3.1 CAAPP PE Memory

The memory for each PE is arranged as a set of 4 128-bit pages providing a total of 512 bits of memory for each PE[2]. The physical pages are denoted as P_0, P_1, P_2, P_3.

An address is given as a 9-bit field — the top two bits specify the page while the botton 7 bits of address specify the bit in the page. The page specified in the address field in a CAAPP PE instruction is denoted as V_0, V_1, V_2, V_3. This virtual page is mapped into a physical page by the Page (PR) Register.

The backing store transfers access V_0 memory and are also effected by the PR register.

6.3.2 CAAPP PE Registers

Each CAAPP PE has a set of these registers:

- ZERO

  This register is a hard-wired 0 bit.

- A

  This register has the activity bit. This register in conjunction with the inhibit (INH) field of a CAAPP PE instruction controls whether an individual PE executes a particular CAAPP PE instruction.

[2] The original implementation provides less than the 4 total pages (320 bits) due to chip space constraints.

- B

  This register is a general purpose register.

- C

  This register is reserved for future use as a local comparand bit when local control logic is incorporated into the CAAPP chips.

- X

  This is a general purpose register which is used in Some/None and Response Count operations. It is read by the Coterie Network (see Figure 6). The use of the X register by the Coterie Network is not affected by the INH field of any CAAPP PE instruction.

- Y

  This is a general purpose register.

- Z

  The Z register is the carry register and is set by the add instruction. The Z register can be set to 0 by doing an add with S_i and S_j selecting the ZERO. It can be set to 1 by doing the same add and complementing S_i and S_j. The Z register may be read again adding the ZERO to itself and placing the result in some other place.

- N

  This is the value of the currently addressed memory bit of the PE to the immediate north. The address field of the CAAPP PE instruction is used to select the memory bit of the neighboring PE to be read. References to P_0 or P_1 will read garbage.

- E

  This is the value of the currently addressed memory bit of the PE to the immediate east. The address field of the CAAPP PE instruction is used to select the memory bit of the neighboring PE to be read. References to P_0 or P_1 will read garbage.

- S

  This is the value of the currently addressed memory bit of the PE to the immediate south. The address field of the CAAPP PE instruction is used to select the memory bit of the neighboring PE to be read. References to P_0 or P_1 will read garbage.

- W

  This is the value of the currently addressed memory bit of the PE to the immediate west. The address field of the CAAPP PE instruction is used to select the memory bit of the neighboring PE to be read. References to P_0 or P_1 will read garbage.

- MR

  The Mesh Register controls the Coterie Network and is used in 4-bit memory transfers. The contents of the MR Register can only be transferred to/from V_0 and V_1.

- SB

  This register is used only in conjunction with the MR register fnr 8 bit parallel memory transfers in the CAAPP PE memory. The MR register receives/transmitts the low address four bits. The contents of the SB Register can only be transferred to/from V_0 and V_1.

6.3.3 CAAPP PE Instructions

These instructions are executed by each PE on the CAAPP Chip in SIMD mode.

Either one or both sources specified by S_i and S_j may be complemented before they are used in the operation. The result may also be complemented before it is placed in the destination specified by *Dest*. The X_{pc} destination places the result in the X register and pre-charges the Coterie Network.

If S_i is N or S or if S_j is E or W, the *Address* field is used to access the operand from the appropriate neighbor PE. *Only one address may be specified in an instruction regardless of whether or not both operands or the destination access memory.*

For $Ftn \geq 8$, C_r has no effect. And, *Dest* should be set to 8 to avoid placing garbage into a register.

If the destination is A, X, then both registers receive the same result. If the destination is $A, X \Leftarrow I$, then the A register receives the result while the X register receives the value specified by I. If the destination is $A, X \Leftarrow J$, then the A register receives the result while the X register receives the value specified by J. If I or J is A, the value set into X is the value of A before it is modified by the instruction.

The operation may or may not be performed at a particular PE depending on the value of the INH field and the value in the A register.

Note that it is possible to select and access pairs of neighboring PEs with a single operation by accessing N, S, E, or W. Pairwise access excludes opposing directions. For example, it is not possible to access both North (N) and South (S) at the same time.

31			27	23		18		13	9	8 0
0	INH	C_r	Ftn	C_i	S_i	C_j	S_j	$Dest$	0	$Address$

INH	(Inhibit)
0	Non-inhibit — always active
1	Inhibit if A = 0
2	Inhibit if A = 0 or S/N = Some
3	Inhibit if A = 0 or S/N = None

$$I \Leftarrow C_i \bigoplus S_i$$
$$J \Leftarrow C_j \bigoplus S_j$$

	S_i	S_j	Ftn	$Dest$	
0	ZERO	ZERO	$Coterie \Rightarrow R$	X_{pc}	0
1	C	C	$I \Rightarrow R$	A, X	1
2	S	E	$J \Rightarrow R$	$A, X \Leftarrow I$	2
3	N	W	$\overline{I \bigwedge J} \Rightarrow R$	$A, X \Leftarrow J$	3
4	Y	Y	$\overline{I \bigvee J} \Rightarrow R$	Y	4
5	X	X	$\overline{I \bigoplus J} \Rightarrow R$	X	5
6	B	B	$\overline{I + J + Z} \Rightarrow R$	B	6
7	A	A	$ICAP\_C \Rightarrow R$	A	7
8	memory	memory	$I \Rightarrow Z$	memory	8
9	—	—	$memory \Rightarrow MR$	—	9
10	—	—	$memory \Rightarrow MR, SB$	—	10
11	—	—	$MR \Rightarrow memory$	—	11
12	—	—	$MR, SB \Rightarrow memory$	—	12
13	—	—	—	—	13
14	—	—	—	—	14
15	—	—	—	—	15

$$Dest \Leftarrow C_r \bigoplus R$$

Figure 8: CAAPP PE Instructions

- $GCN \Rightarrow R$

  The state of the Coterie Network is sensed at each PE and loaded into the destination register. The Coterie Network is controlled by the Mesh Register (MR), the value of each PE's X register, and whether the Mesh was pre-charged. C_r is effective for this operation.

- $I \Rightarrow R$

  The contents of the source designated by the S_i field are loaded into the destination.

- $J \Rightarrow R$

  The contents of the source designated by the S_j field are loaded into the destination.

- $\overline{I \bigwedge J} \Rightarrow R$

  The complement of the logical product of the sources designated by S_i and S_j are placed in the destination.

- $\overline{I \bigvee J} \Rightarrow R$

  The complement of the logical sum of the sources designated by S_i and S_j are placed in the destination.

- $\overline{I \bigoplus J} \Rightarrow R$

  The complement of the logical difference of the sources designated by S_i and S_j are placed in the destination.

- $\overline{I + J + Z} \Rightarrow R$

  The complement of the binary sum of the sources designated by S_i, S_j, and the Z register are placed in the destination. The Z register is set to 1 or 0 depending on whether or not there was a carry out from the addition.

- $ICAP\ C \Rightarrow R$

  Puts the output of the ICAP comparand output into the destination.

- $I \Rightarrow Z$

  Transfer the contents of the S_i source to the Z register. C_i is effective. C_r is not effective.

- $memory \Rightarrow MR$

  This instruction transfers four bits from each PE V_0 or V_1 memory to the Mesh (MR) register. The address field of the instruction is used to specify the 4 bit nibble. Only bits 7...2 of the address field are used because nibbles are only accessed on 4-bit boundaries.

- $memory \Rightarrow MR, SB$

  This instruction transfers eight bits from each PE V_1 memory to the Mesh (MR) register, Static Buffer (SB) register pair. The address field of the instruction is used to specify the 8 bit byte. Only bits 7...3 of the address field are used because bytes are only accessed on 8-bit boundaries. The low address four bits are transferred to the MR Register.

- $MR \Rightarrow memory$

  This instruction transfers four bits from each PE Mesh (MR) register to V_1 memory. The address field of the instruction is used to specify the 4 bit nibble. Only bits 7...2 of the address field are used because nibbles are only accessed on 4-bit boundaries.

- $MR, SB \Rightarrow memory$

  This instruction transfers eight bits from each PE Mesh (MR) register, Static Buffer (SB) register pair to V_1 memory. The address field of the instruction is used to specify the 8 bit byte. Only bits 7...3 of the address field are used because bytes are only accessed on 8-bit boundaries. The low address four bits are transferred from the MR Register.

6.4 Finite State Machines

These finite state machines are implemented on the CAAPP Chip. They control access sequences for the various memories, bus contention arbitration, and other functions.

6.4.1 ICAP Hold Arbiter

This finite state machine decides which asynchronous process obtains the ICAP Bus. The process raises a hold request line. When it is given control of the ICAP Bus, it receives a Hold Ack signal. It keeps its hold request line up until it is done with the bus.

6.4.2 Backing Store Controller

This finite state machine implements the **write** BSM and **read** BSM CAAPP instructions. When the CAAPP Chip Instruction decode logic recognizes these two instructions, it gates the arguments into the *BSA*, *BSE*, *BSCS*, and *BSBA* registers and lowers the *BackingStoreDone* signal to the ACU. It then sets the *READBSM* or *WRITEBSM* flipflops.

In this logic it is assumed that a carry out of the BSP register is added into the least significant bit of the BSA register.

Many of the signal names in this section refer to signals on the Video RAM chip (see Mitsubishi M5M4C264P).

6.4.3 ICAP-ISSM Read/Write

The ICAP-ISSM Read/Write finite state machine is used by the ICAP to read data from the ISSM or write data into it. The ISSM is arranged as 256 256-word blocks in Video RAM memory. Data must first be placed into the serial buffer of the Video RAM memory before it can be moved into the RAM that is referenced as SPA memory. The reverse is also true.

Data is transferred to and from the ICAP bus and the serial access memory of the Video RAM chip by a finite state machine on the CAAPP chip. To transfer the serial access memory to and from the Video RAM chip RAM memory requires that the SPA bus be tri-stated. This is also required by refresh logic. Logic on the ACU board periodically tri-states the SPA bus and does the refresh. Also periodically, logic on the ACU board sends a signal (TRG) to each CAAPP Chip specifying that the transfers may be done — the SPA bus is tri-stated at this point. Each CAAPP Chip that has a transfer pending does the transfer at this point.

Three ICAP I/O ports are used:

- ITA — this port is used for writting the ITCA and ITCS registers. The ITCA register contains the block number of the block in Video RAM memory. The ITCS register contains the column select for the serial buffer of the Video RAM memory.
- ITC — this port is used to initialize the ISSM serial/RAM transfer. A transfer to ITC causes the *TRF* register to be set. The mode is remembered so that when *ITRG* comes up, the proper sequence may be done. The value of the mode is latched into the TRF register.

mode	action
00	do nothing
01	Pseudo-write
10	Write
11	Read

- ITT — this port is used to actually transfer 16 bits of data to and from the serial buffer and the ICAP.

The transfers are accomplished by ICAP I/O operations.

```
Write data to ISSM:
  Output Block Number, Column Select to port ITA.
  Output Pseudo-write mode to port ITC.
  Wait on BIO Signal.
  Loop
    Output Data to port ITT.
  Loopend
  Output Write mode to port ITC.
  Wait on BIO signal.

Read data from ISSM:
  Output Block Number, Column Select to port ITA.
  Output Read mode to port ITC.
  Wait on BIO signal.
  Loop
    Input Data from port ITT.
  Loopend
```

7 ICAP

7.1 ICAP I/O Ports

The ICAP uses I/O operations to control various processes and access results. An ICAP I/O operation raises IS which is seen by logic on the CAAPP Chip. This logic uses the address lines from the ICAP to select one of various functions. The data lines are use to specify the data needed in the operation or to return the results.

$A_{3...0}$	Name	$R/\overline{W}$	Action
0			
1			
2			
3			
4			
5			
6			
7			
8	ID	0	$ID \Leftarrow D_{2...0}$
		1	Daughter Board Status $D_{9...0}$
9	IC	0	$IC \Leftarrow D_6$
		1	$D_6 \Leftarrow IC$
10	ICR	0	$ICR \Leftarrow D_{7...0}$
11	LCR	1	$D_{7...0} \Leftarrow LCR$
12	ITT	0	Transfer $D_{15...0}$ to ISSM
		1	Transfer from ISSM to $D_{15...0}$
13	ITC	0	Activate ICAP-ISSM Read/Write Mode in $D_{1...0}$
14	ITA	0	$ITCA \Leftarrow D_{7...0}$ $ITCS \Leftarrow D_{15...8}$
15	IBS	0	$IBS \Leftarrow D_{7...0}$

Figure 9: ICAP Port Assignments

Because the CAAPP PEs run twice as fast as the ICAP, the Some/None bits in the Daughter Board status register can change faster than the ICAP can read them. This isn't really important, because the ICAP can get a latched some/none value from the count. If someone wants to program the CAAPP and ICAP to interact using the S/N bits in the status register, then they have to be aware that some handshaking will need to be done to guarantee that the ICAP gets valid results.

Bit	Status
0	D_0 — ICAP Done Status
1	D_1
2	D_2
3	S/N — CAAPP Chip Some/None
4	BSMD — Backing Store Memory Done
5	EF — Memory test Error Flag
6	IC — ICAP Comparand
7	S/N — CAAPP Chip Some/None
8	Y — Chip Select (row)
9	X — Chip Select (column)

Figure 10: Daughter Board Status

Appendix C:
Test Reports and Photographs of Chips Fabricated Through MOSIS During Year 1

TEST REPORT FOR THE FEEDBACK CONCENTRATOR CHIPS

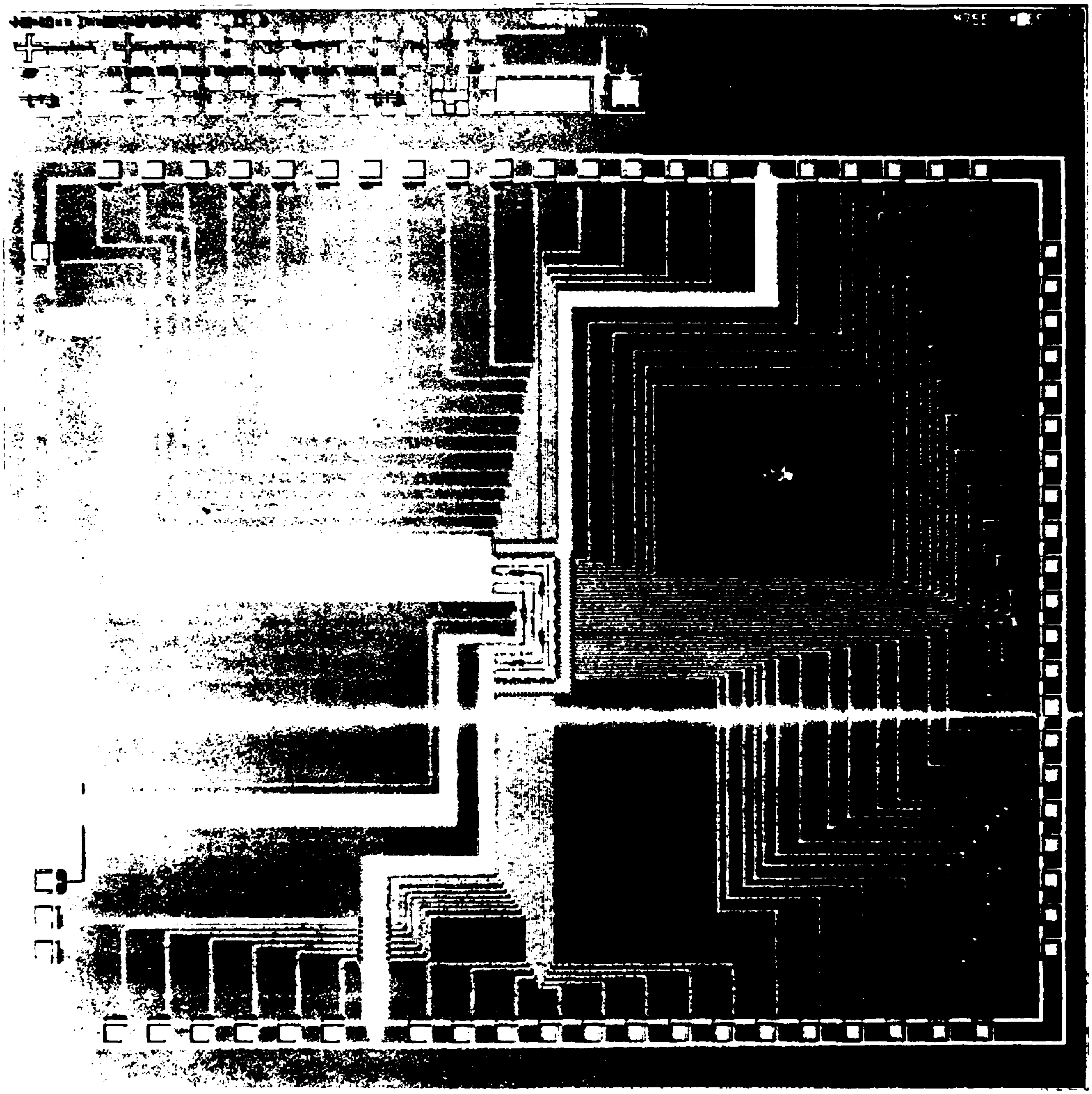
P-Name:	64OR
Fab-ID:	M75EPE1 $\lambda = 1.5\mu$ (NMOS)
Source·	UMASS

This chip implements a data feedback concentrator.

A total of 21 packaged parts were received. They all were in good physical condition.

This chip was designed in NMOS with $\lambda = 2.0\mu$. During submission to MOSIS, we made a mistake in specifying parameter "min-lambda", and as a consequence, the lambda was scaled down by MOSIS to 1.5 microns. This resulted in a reduction in pad sizes to two thirds of the minimum required for safe bonding. All chips had many wire bonds on the die shorted to the outer power supply ring. The problem has been corrected and the chip has been redesigned in CMOS.

Figure 1. Feedback Concentrator Chip



TEST REPORT FOR THE ICAP ROUTER CHIPS

P-Name:	ROUTER
Fab-ID(1):	M77PCE1 (VTI) $\lambda = 1.0\mu$ (SCN)
Fab-ID(2):	M77NCJ1 (UTMC) $\lambda = 1.0\mu$ (SCP)
Source:	UMASS

This chip was assigned to two different runs by MOSIS. The circuit implements a 32-input 32-output crossbar switch with a control memory called the Connection Pattern Cache (CPC).

VTI RUN:

A total of 24 parts were received. They were all in good physical condition. Bonding was acceptable in most of the cases, but in some cases, the solder bonds on the dies were dangerously close to the outer power supply ring of the standard frame.

About 12 parts were tested and none of them worked. There were logical bridging faults between the outputs, which were at different places in different chips. We suspected that these bridging faults between the outputs were due to *electrical shorts* between Metal1 and Metal1, as well as between Metal2, and Metal2 in the multiplexure array. These were confirmed to be shorts by visually inspecting a few parts with a stereo microscope. There were cases in which 4 λ wide metal1/2 wires separated by 4 λ appear to be essentially one wider wire.

Besides the bridging faults, there were many output stuck-at-0 and output stuck-at-1 faults. We suspect that these were due to shorts between different layers in the chip.

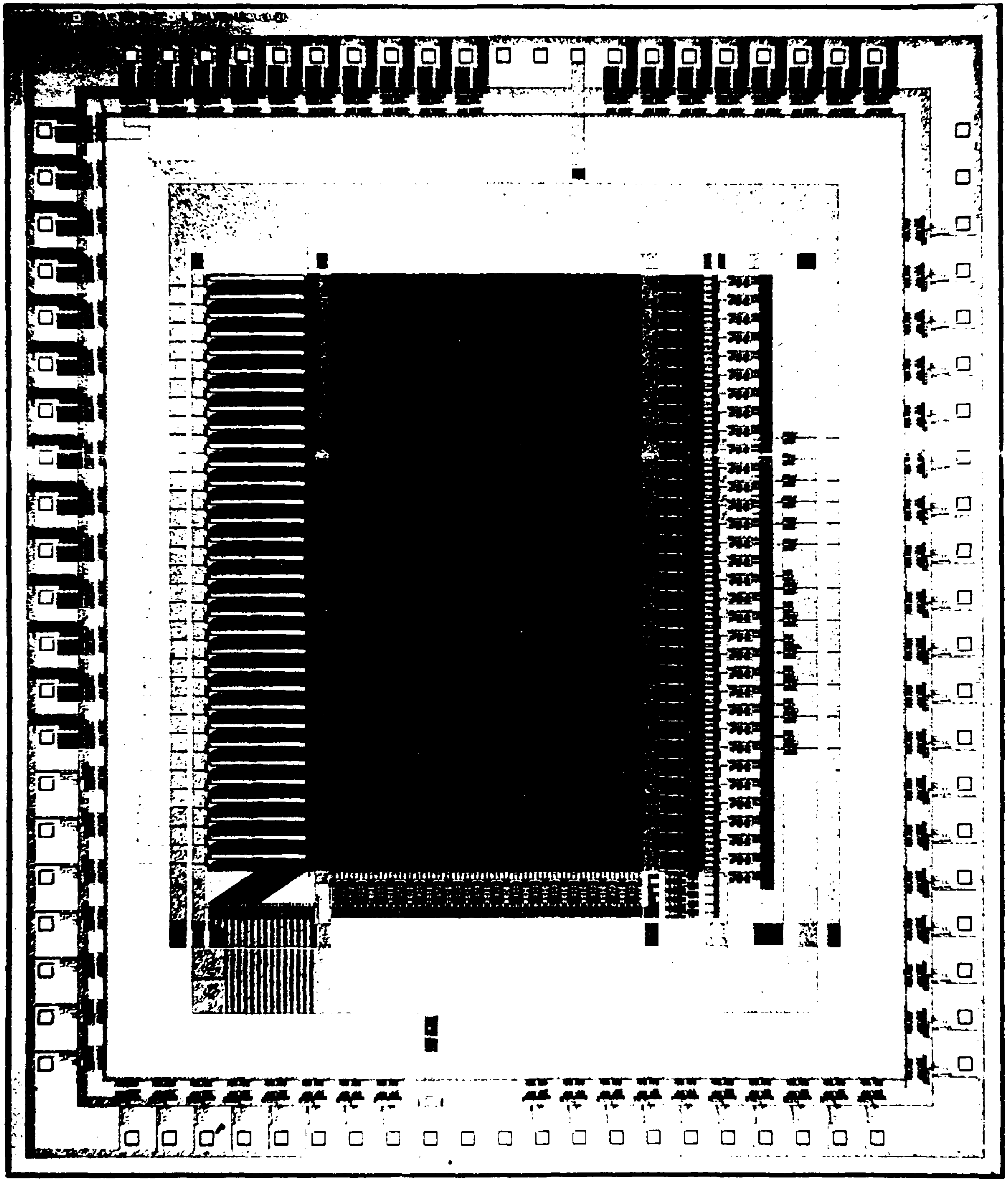
UTMC RUN:

A total of 24 parts were received. They were in good physical condition. Bonding of the parts was good.

About 10 parts were tested and none of them worked. We cannot provide as much feedback about this run because all of the outputs in all of the chips were logically stuck-at-1. By visual inspection with a stereo microscope, we found that the geometries on the die were much better defined than the VTI run, but metal wires separated by *minimum* distance still appeared dangerously close in many places.

In consultation with MOSIS, these problems have been traced to the fact that VTI and UTMC do not planarize their wafers, which leads to excessive spreading in the Metal2 layer. We have arranged through MOSIS to use National Semiconductor and Hewlett-Packard for future fabrication runs, which should avoid this problem.

Figure 2. ICAP Router Chip



TEST REPORT FOR THE 32-ELEMENT CAAPP CHIPS

P-Name:	CAAPP32
Fab-ID(1):	M72O (VTI) $\lambda = 1.0\mu$ (SCN)
Fab-ID(2):	M72K (National) $\lambda = 1.0\mu$
Source:	Hughes

This chip was assigned to two different runs by MOSIS. The circuit implements 32 CAAPP processors and their associated on-chip memories.

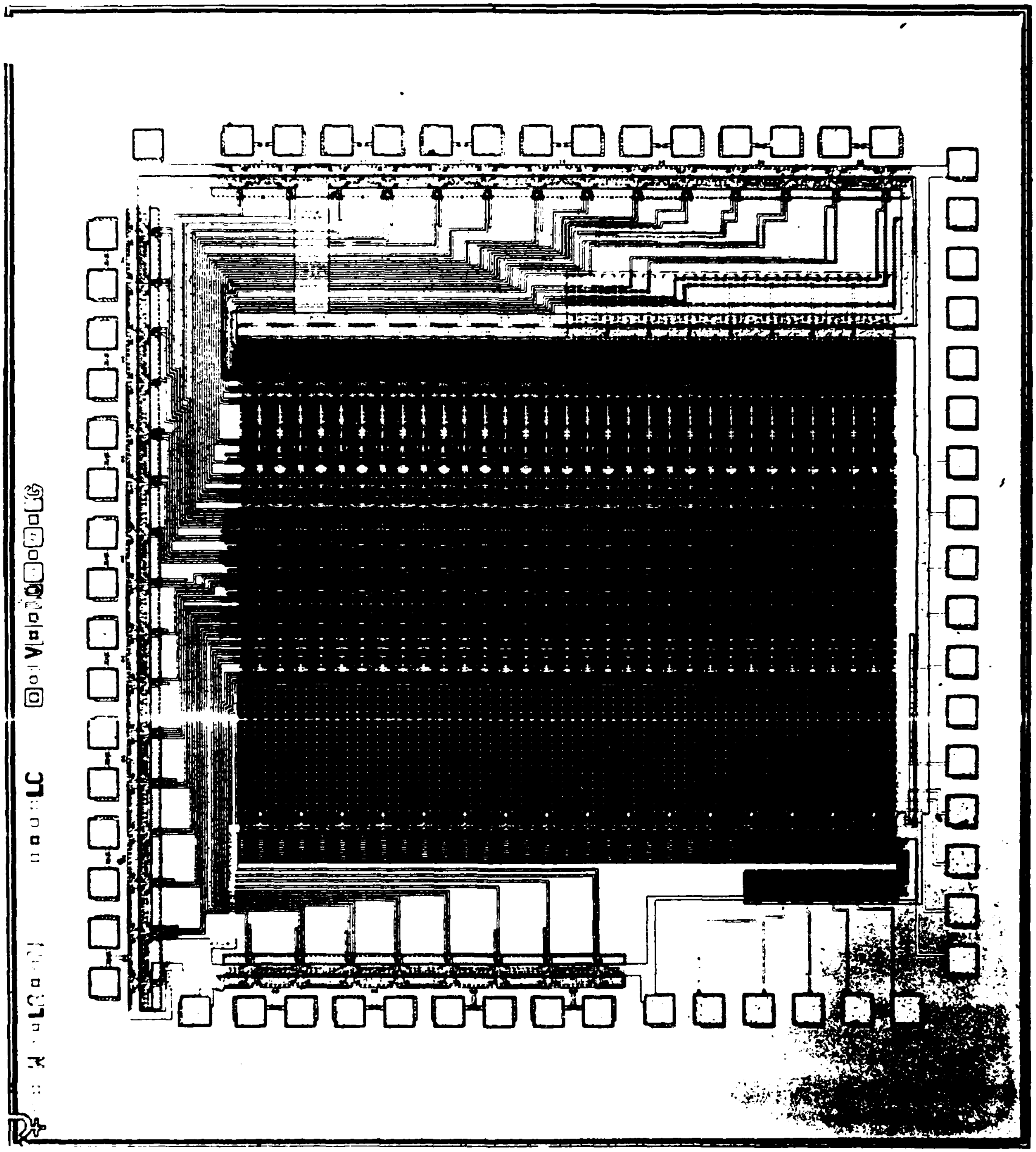
VTI RUN:

All of the chips from the VTI run were inoperative due to excessive metal spreading that resulted in numerous shorts. Working with MOSIS, Hughes was able to remove the passivation of several chips and cut some of the metal lines. This allowed micro-probe testing of the ALU and the dynamic RAM. The ALU functioned correctly, but only a small portion of the dynamic RAM was functional.

NATIONAL RUN:

This run was not tested by Hughes because the decision had been made, based on the results of the VTI run, to completely redesign the chip using the MOSIS design rules (instead of the VTI rules) and to submit the chip for fabrication at National. (Hughes obtained National's proprietary rule set but elected not to use it for this project).

Figure 3. 32 Element CAAPP Chip



Appendix D: Documentation for IUA Simulators

Part 1: Texas Instruments Explorer Version of Simulator

IUA Software Simulator Specification

Michael Rudenko

1 Overview

The primary function of a simulator is to provide a user with a working model of a target machine on a host machine. An appropriately structured simulator might prove useful for:

- design verification
- hardware debug
- hardware performance evaluation
- application program development
- algorithm performance evaluation

There are a variety of levels at which a machine's behavior may be modeled. If every gate and every signal to or from a gate is represented individually, then the simulator is referred to as a "structural" model. An even finer level of granularity may be achieved by representing the predicted behavior of the semiconductor technology selected for the construction of the circuits. At the other end of the spectrum, one of the coarsest levels of granularity is to represent the machine as it would appear to an applications programmer. This would entail combining a register transfer model of the architecture together with a model of the system's input/output structures. Such a representation is generally referred to as a "behavioral" model.

The level of granularity at which a machine is simulated is a function of the type of information desired, and the simulation run-time cost one is willing to tolerate. If one intends to use the simulator for design verification, hardware debug, or hardware performance evaluation, a structural model is essential. Such a model could potentially be quite large and slow. If the intent of the simulator is to act as an instruction set interpreter for the target architecture for purposes of developing programs and evaluating algorithms, a structural model simulation would probably take longer to execute than one might reasonably expect to tolerate. In this case, the simpler, faster behavioral model would be more appropriate.

1.1 Hardware Structural Model

A hardware structural model is an idealized representation of the manner in which the parts of a machine will function under some specified conditions. A crude level of structural modeling would be to represent the machine in terms of registers, busses, data paths, arithmetic logical units, shifters, and so on. A finer grained structural model would include modeling of the combinational and sequential circuit networks which go to make up the detailed logic design of the architecture. In any event, the simulator will at a minimum need to provide its user with the ability to track of all of the inputs, outputs and current values for all signals and locations which have been modeled. Additionally, provisions probably need to made to allow the user to dynamically examine and alter values, set trace and watch points, and step the clock.

1.2 Hardware Behavioral Model

The behavioral model version of a simulator would enable a user to develop programs for a target machine in such a fashion that the user would be able run the program unaltered on a host machine under control of the simulator. The purpose of such a "software" simulator would be to allow development, debug, and execution of code before any target machine hardware is available. The simulator environment would probably also allow interactive reading and writing to target machine registers and memory locations as well as single instruction execution control. The structure of a behavioral model simulator generally parallels the structure of a register transfer model of the architecture, with some statistics gathering and control capabilities included.

2 IUA Simulation

The IUA is to be composed of custom and off-the-shelf components. The custom logic for the IUA resides in two places: on the CAAPP chips and on the "glue" logic PLD chips. These components will require structural level support to assist in design verification and hardware debug. The remainder of the components are assumed to function as specified by their vendors, and thus should not require structural level simulators.

A behavioral level simulator will be written for as much of the IUA as is feasible. The primary purpose of this simulator will be to provide a vehicle for program development and algorithm performance evaluation.

2.1 Structures

The major modules which are to be simulated are:

1. CAAPP processing elements/mesh network
2. CAAPP control unit
3. backing store memory and controller
4. switch circuit network
5. associative functions unit
6. ICAP processors/local memory
7. ICAP routing network
8. ICAP/SPA shared memory

The ordering of these modules represents the expected sequence in which they will be developed. Additionally, the following features will be present in the simulator:

- specify the number of CAAPP cells present
- load/dump of bit, byte, or word image planes
- load/dump of CAAPP registers
- histograms of memory fields
- register response counts
- read/write instruction cycle counter

3 Run-time Environment

There will be three modes of simulator operation:

- program controlled
- interactive
- pure

3.1 Program Controlled Simulation

In program controlled simulation, the simulator subroutines will be available as library routines which the user's program can call however desired. The user is responsible for initializing the simulator data base, and calling whatever routines are necessary in order to execute machine operations. Various machine state statistics may be obtained by calling appropriate library routines. The user program thus takes on the role of a global controller with statistics gathering capabilities.

3.2 Interactive Simulation

In interactive simulation mode, the user will interact with the simulator library under the *VISIONS* system. This mode enables the user to examine and manipulate image data which has been processed by the simulator, with the full functionality of the *VISIONS* system environment. So, rather than writing a program which is compiled, linked to the simulator and then run - possibly followed by starting up the *VISIONS* system and examining the resultant image, the user can invoke simulator routines simply by calling them from the *VISIONS* system prompt.

3.3 Pure Simulation

In pure simulation mode, the user must write a program just as it would appear for execution on the actual hardware of the target system. This code will pass through the cross-assembler, and the object code will be "executed" by the simulator program. No meta-simulator statistics gathering routines may be called.

CAAPP PE Instructions

Register Names

- Refer to §5.3.3 of the *CAAPP Functional Specification*
- Registers which can be either a source or destination: **X, Y, A, B**.
- Additional source registers: **ZERO, S, N, E, W**. The S and N registers can only be used for *Si*, and the E and W registers can only be used for *Sj*.
- Additional destination registers: **X–PC, A–X, A–X–I, A–X–J**.
- Memory locations appear as numeric values: 0, 17, etc. The programmer can store memory locations in variables or symbolic constants, and use them in place of writing a numeric value. Additionally, any numeric expression which evaluates to a positive integer (less than 512) can appear in an instruction.
- The carry bit for arithmetic operations is called **Z**. It may also be written to directly via the Z := SI . . . instruction.
- The 4-bit Mesh–Register is called **MR**, and the 8-bit Mesh–Register/Static–Buffer is called **MR–SB**.
- Except for MR and MR–SB, any source register can be complemented by preceeding it with a tilde (˜) followed by a space, e.g. Y := ˜ X 'AND B !!. When the MR or MR–SB are used as sources, their complemented forms are ˜**MR** and ˜**MR–SB** (no space after the tilde).
- Destinations are never complemented. Instead, each function has a complemented form.

Function Names

- Refer to §5.3.3 of the *CAAPP Functional Specification*
- The CAAPP PE functions are: **GCN, VIA–SI, VIA–SJ, 'NAND, 'NOR, 'NXOR, 'N+, ICAP–C, SI**.

- Each function has a complemented form: ˜ **GCN**, ˜ **VIA–SI**, ˜ **VIA–SJ**, **'AND**, **'OR**, **'XOR**, **'+**, ˜**ICAP–C**, ˜**SI**.

- The functions relating to the MR and MR–SB registers are implemented via assignment statements, e.g. 12 := MR !! would copy mesh–register bit 0 to memory bit 12, mesh–register bit 1 to memory bit 13, etc.

- The "SI" and "˜SI" functions refer to loading the Z–register with the contents of the SI–register, e.g. Z := SI X !! would load the contents of the X–register into the Z–register.

Activity Control

- Every instruction must end in one of four activity control flags: **!!**, **A!**, **S!**, **N!**. These correspond to: execute always, inhibit if A=0, inhibit if A=0 or S/N=Some, inhibit if A=0 or S/N=None, respectively.

CAAPP Chip Instructions

- Refer to §5.2 of the *CAAPP Functional Specification*

- CAAPP Chip Functions ("Mode 1"): **RESET, BSM–READ, BSM–WRITE, ROUTE, SET–PR, LATCH-CNT, LATCH-LCL-CNT, INJECT-GCN, LATCH-CAAPP-S/N, HOLD-ICAP, CONNECT-TO, WRITE, RELEASE-ICAP**.

- Each function is followed by up to four numeric parameters, separated by spaces, and always ended with "!!", e.g. BSM–READ 0 4 0 17 !! transfers data from backing store block 17, column 0, to bytes 0 to 4 of PE memory. Numeric parameters may be replaced with user defined constants or variable values or any expression which evaluates to an integer.

ACU Simulator Utilities

- **AA** — turns on activity register (A) for all pe's
- **ZZ** — clears the carry bit register (Z) for all pe's
- **DISPLAY** — displays fields in an instruction
- **STOP-CAAPP** — stops CAAPP simulator
- **STOP-BS** — stops Backing Store simulator
- **DRAW-GREY** — displays greyscale/dither patterns
- **SHOW-GREY** — displays pixel value to dither pattern correspondence
- **UNIFORM-GREY** — distribute 32 dither patterns uniformly for 256 levels
- **SET-GREY** — set a range of pixel values to some dither pattern
  - *required arguments*: **low-value high-value dither-pattern#**
- **MVIEW** — examines pe-memory
  - *required arguments*: **starting-address length**
  - e.g. **17 8 mview** would display 1024 8-bit numbers whose low order bit is bit 17
- **MPIC1** — graphically displays pe-memory
  - *required arguments*: **starting-address length**
  - e.g. **17 8 mpic1** would produce a 32 by 32 display of 8-bit pixel values whose low order bit is bit 17
- **MPIC** — graphically displays 1 bit of pe-memory
  - *required arguments*: **address**
  - e.g. **17 mpic** would produce a 32 by 32 display of bit 17 for each pe
- **VIEW** — displays contents of a register

  - *required arguments*: **register**
  - choices are: X-REG, Y-REG, A-REG, B-REG, Z-REG

- **PIC** — graphically displays a register
  - *required arguments*: **register**
  - choices are: X-REG, Y-REG, A-REG, B-REG, Z-REG
- **MRVIEW** — graphically displays the mesh register connections (and X-register)
- **PEVIEW** — displays all registers and memory bits for 1 pe
  - *required arguments*: **pe#**
- **FILL-IP-BUF** — fills image plane buffer
  - *required arguments*: **pe#i-data pe#i+1-data ... pe#i+k-data k i. k** is the number of values being loaded, and **i** is the starting pe number.
  - e.g. **0 2 2 17 0 5 32 FILL-IP-BUF** would load the 5 values 0, 2, 2, 17, and 0, into the image plane buffer for pe 32, 33, 34, 35, and 36, respectively.
- **IP-TO-PE** — loads contents of image plane buffer into pe memory
  - *required arguments*: **bits values start-pe pe-mem-start**, where **bits** is how many pe-memory bits to use, **values** is how many values to tranfer from the image plane buffer to pe memory, **start-pe** is which pe to begin loading to, and **pe-mem-start** is which bit in pe memory is the low order bit of the data.
- **CLR-IP-BUF** — fills the image plane buffer with zeros.
- **LOAD-IMAGE-FILE** — loads the 32 by 32 8-bit image plane (level 6) into pe-memory locations 256 - 263. The image plane must be named: IMAGE-IN.DAT
- **STORE-IMAGE-FILE** — stores the 32 by 32 8-bit image plane in pe-memory locations 256 - 263 to file: IMAGE-OUT.DAT
- **INIT** — intialize execution cycles to 0
  - execution cycles are accumulated in the variable CYCLES in the ACU

- **GET–S/N** — places value of some/none register on the stack
- **GET–CNT** — places value of count register on the stack
- **TEST–PE** — test a bit of a register for one pe
  - *required arguments*: **pe# base-addr**
  - legal base-addr values: X-REG, Y-REG, A-REG, B-REG, Z-REG
- **SET–PE** — set a bit of a register for one pe
  - *required arguments*: **pe# base-addr**
  - legal base-addr values: X-REG, Y-REG, A-REG, B-REG, Z-REG
- **CLR–PE** — clear a bit of a register for one pe
  - *required arguments*: **pe# base-addr**
  - legal base-addr values: X-REG, Y-REG, A-REG, B-REG, Z-REG
- **MR–SET** — set a bit in the mesh register for a pe
  - *required arguments*: **direction pe#**
  - legal directions: W–LINK, N–LINK, E–LINK, S–LINK
- **ZERO–FIELD** — clears field in memory
  - *required arguments*: **starting–address length**
- **COMPLEMENT** — complements a field
  - *required arguments*: **starting–address length**
- **EQUAL** — global compare of a value with a field in all active pe's
  - *required arguments*: **comparand starting–address length**
  - Activity register is set for those pe's equal to the comparand
- **NOT–EQUAL** — global compare of a value with a field in all active pe's
  - *required arguments*: **comparand starting–address length**

  - Activity register is set for those pe's not-equal to the comparand

- **LESS** — global compare of a value with a field in all active pe's
  - *required arguments*: **comparand starting-address length**
  - Activity register is set for those pe's less than the comparand

- **LESS-EQUAL** — global compare of a value with a field in all active pe's
  - *required arguments*: **comparand starting-address length**
  - Activity register is set for those pe's less than or equal to the comparand

- **GREATER** — global compare of a value with a field in all active pe's
  - *required arguments*: **comparand starting-address length**
  - Activity register is set for those pe's greater than the comparand

- **GREATER-EQUAL** — global compare of a value with a field in all active pe's
  - *required arguments*: **comparand starting-address length**
  - Activity register is set for those pe's greater than or equal to the comparand

- **LEAST** — find active pe's with minimum value in a field
  - *required arguments*: **starting-address length**
  - Activity register is set for those pe's containing the minimum value comparand

- **GREATEST** — find active pe's with maximum value in a field
  - *required arguments*: **starting-address length**
  - Activity register is set for those pe's containing the maximum value comparand

- **COPY** — copy one field to another
  - *required arguments*: **dest-addr source-addr length**

- **ADD2** — add one field to another
  - *required arguments*: **dest-addr source-addr length**

- both the X and Z registers get the carry bit

- **ADD3** — add one field to another
  - *required arguments*: **dest–addr source-addr source-addr length**
  - both the X and Z registers get the carry bit
- **MULTIPLY** — multipy one field by another
  - *required arguments*: **dest–addr source-addr length source-addr length**

ICAP Simulator Utilities

- **BS!** — store a word in the backing store
  - *required arguments*: **data address chip#**
- **BS@** — fetch a word from the backing store
  - *required arguments*: **address chip#**
- **F** — signify that the next argument is a floating point number
  - *required arguments*: **fp#**
  - e.g. **F 3.1416** would convert the string "3.1416" to a floating point format understood by the ICAP. This word can only be used outside of colon definitions.
- **FPCREATE** — create a variable of type floating point
  - *required arguments*: **name**
  - e.g. **FPCREATE FOO** sets aside memory to hold one floating point number, address is "FOO". This word can only be used outside of colon definitions (just like the standard CREATE).
- **FPVAR** — create a floating point variable and intialize it
  - *required arguments*: **fp#**
  - *command format*: **FPVAR fp# name**

  - e.g. **FPVAR 3.1416 FOO** would create a floating point variable called "FOO" and intialize it to 3.1416.

- **WFP** — display the floating point number on top of the stack
- **FPSWAP** — swap the two top two floating point numbers on the stack
- **FPDROP** — drop the floating point number on the stack
- **FP!** — store the floating point number on top of the stack
  - *required arguments*: **fp# (ie exponent low-order-mantissa high-order-mantissa sign address**
- **FP@** — fetch a floating point number and place it on top of the stack
  - *required arguments*: **address**
- **F*** — multiplies the top two floating point numbers on the stack
- **F+** — adds the top two floating point numbers on the stack
- **F/** — divides top floating point numbers on the stack by the floating point number under it

Sample Explorer Displays from Simulator

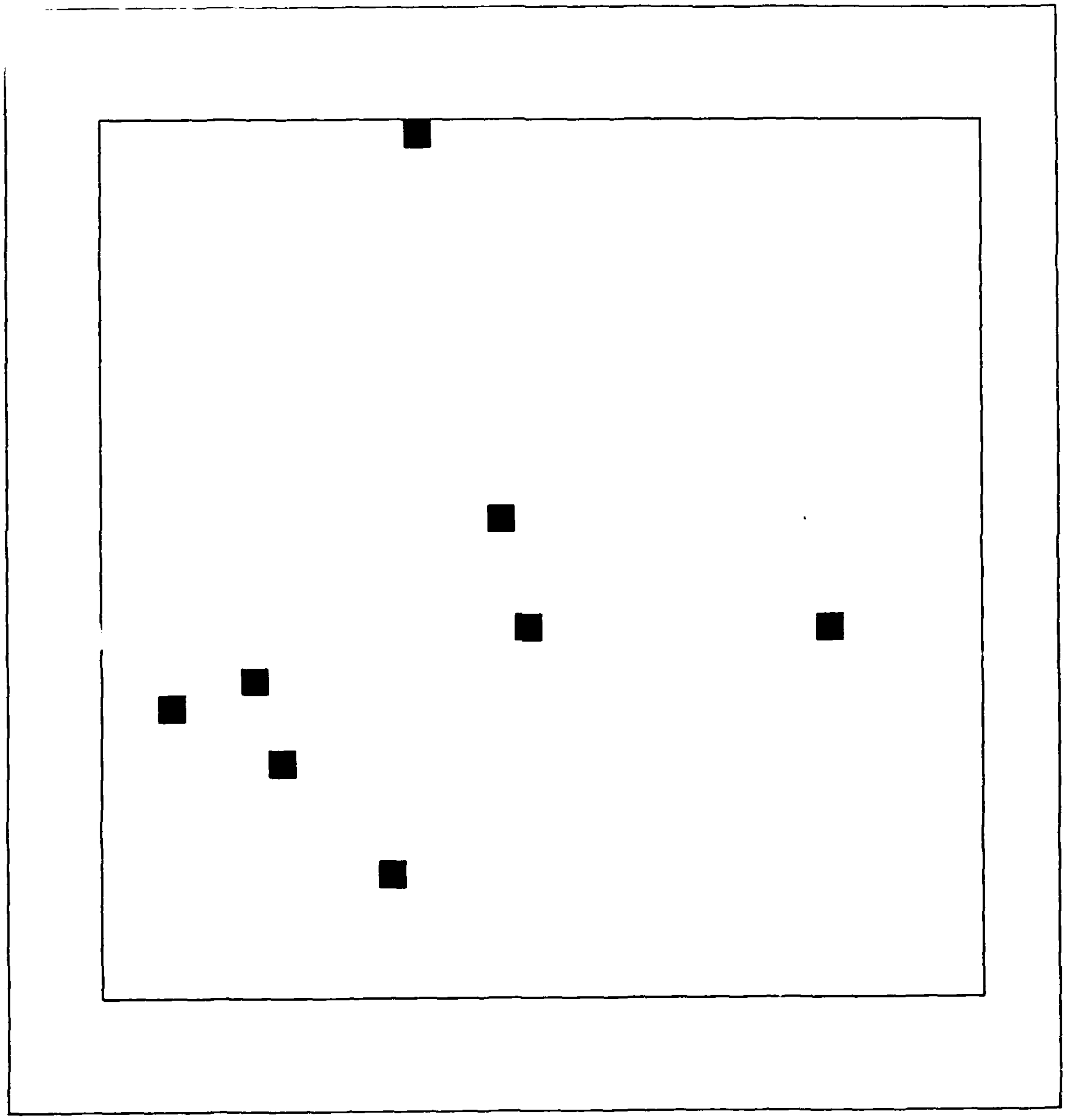


Figure 1. Display of X-Register Image

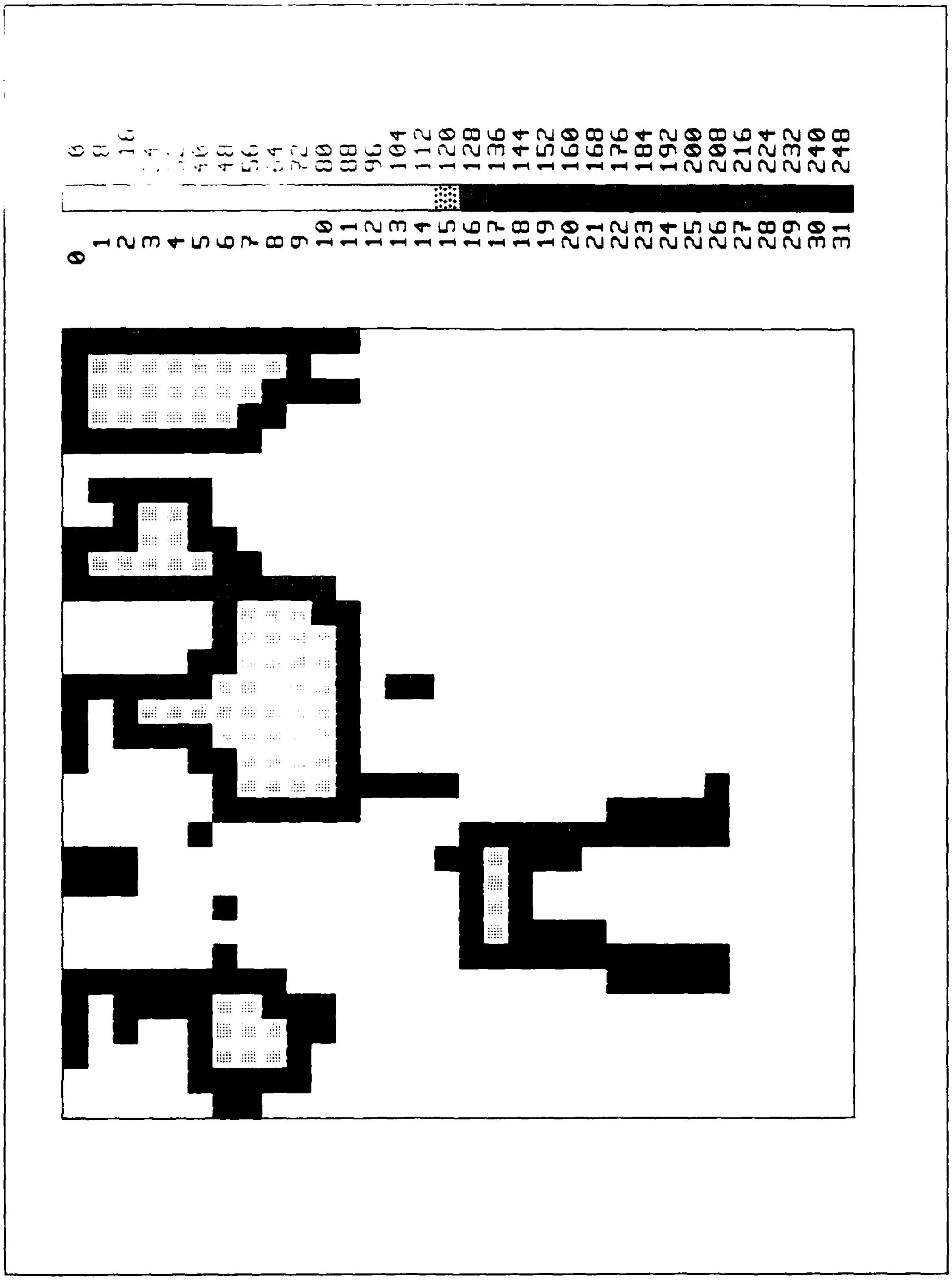


Figure 2. Display of Memory

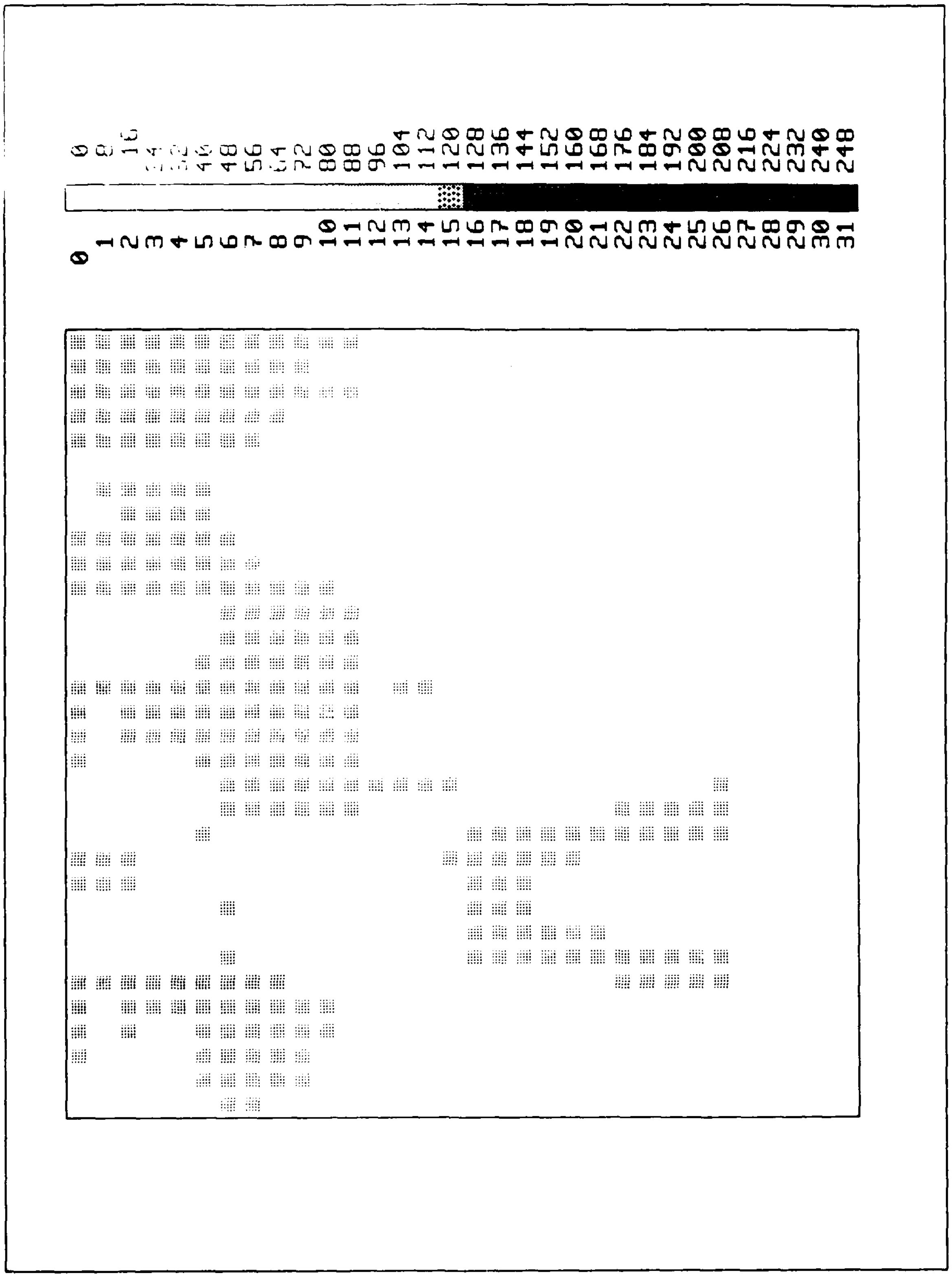


Figure 3. Thresholded Display of Memory

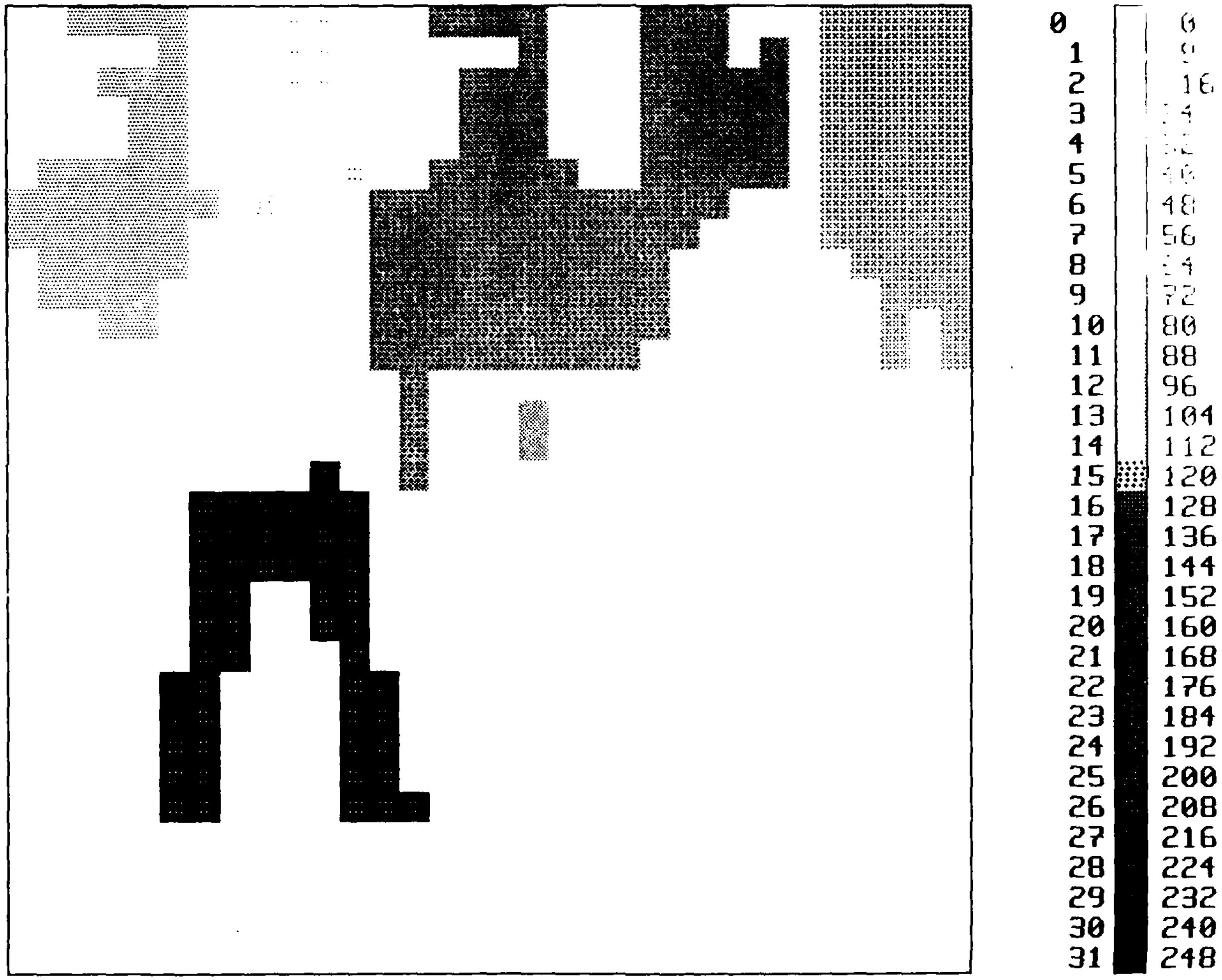


Figure 4. Display of Labeled Components

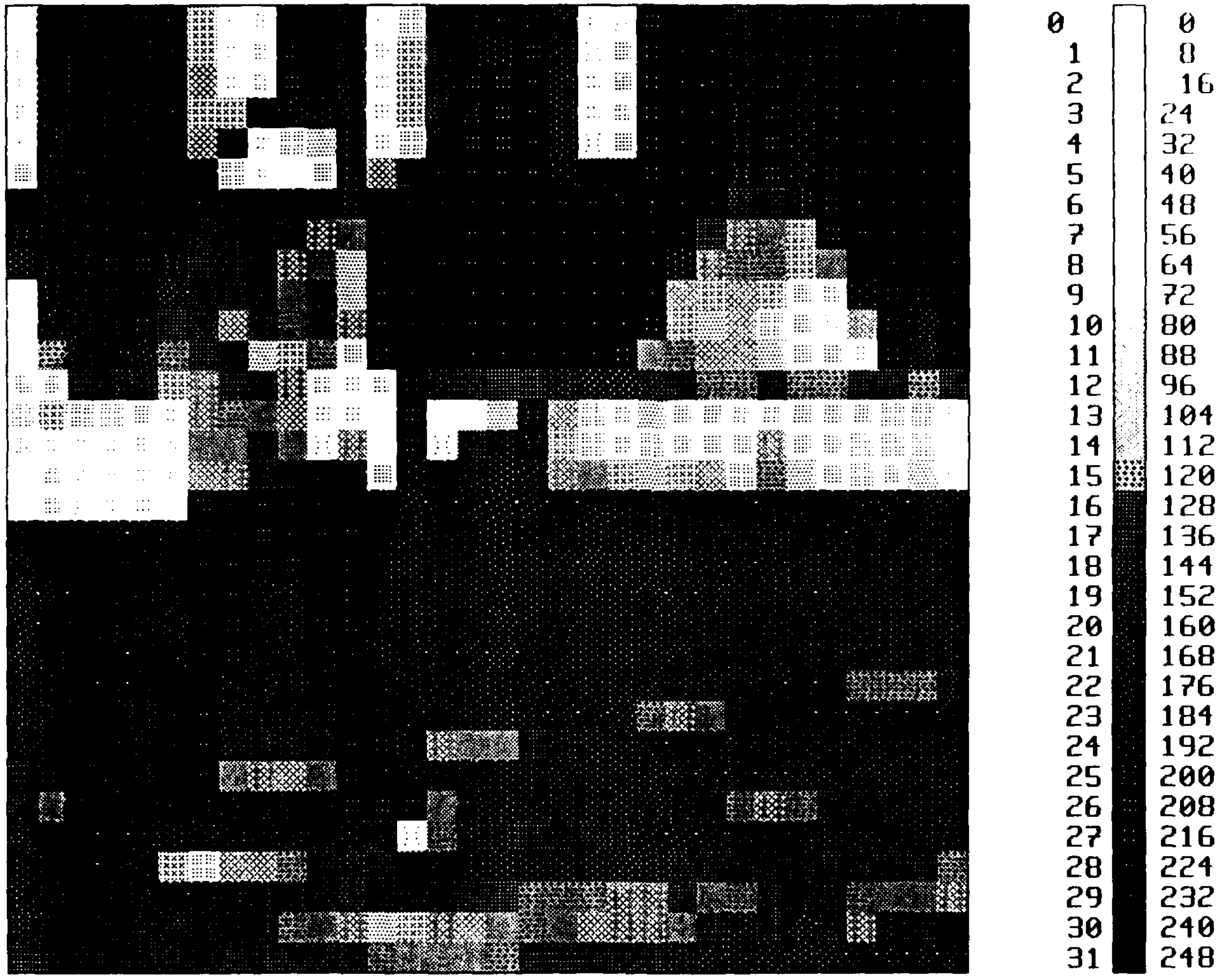


Figure 5. Display of Image in Memory

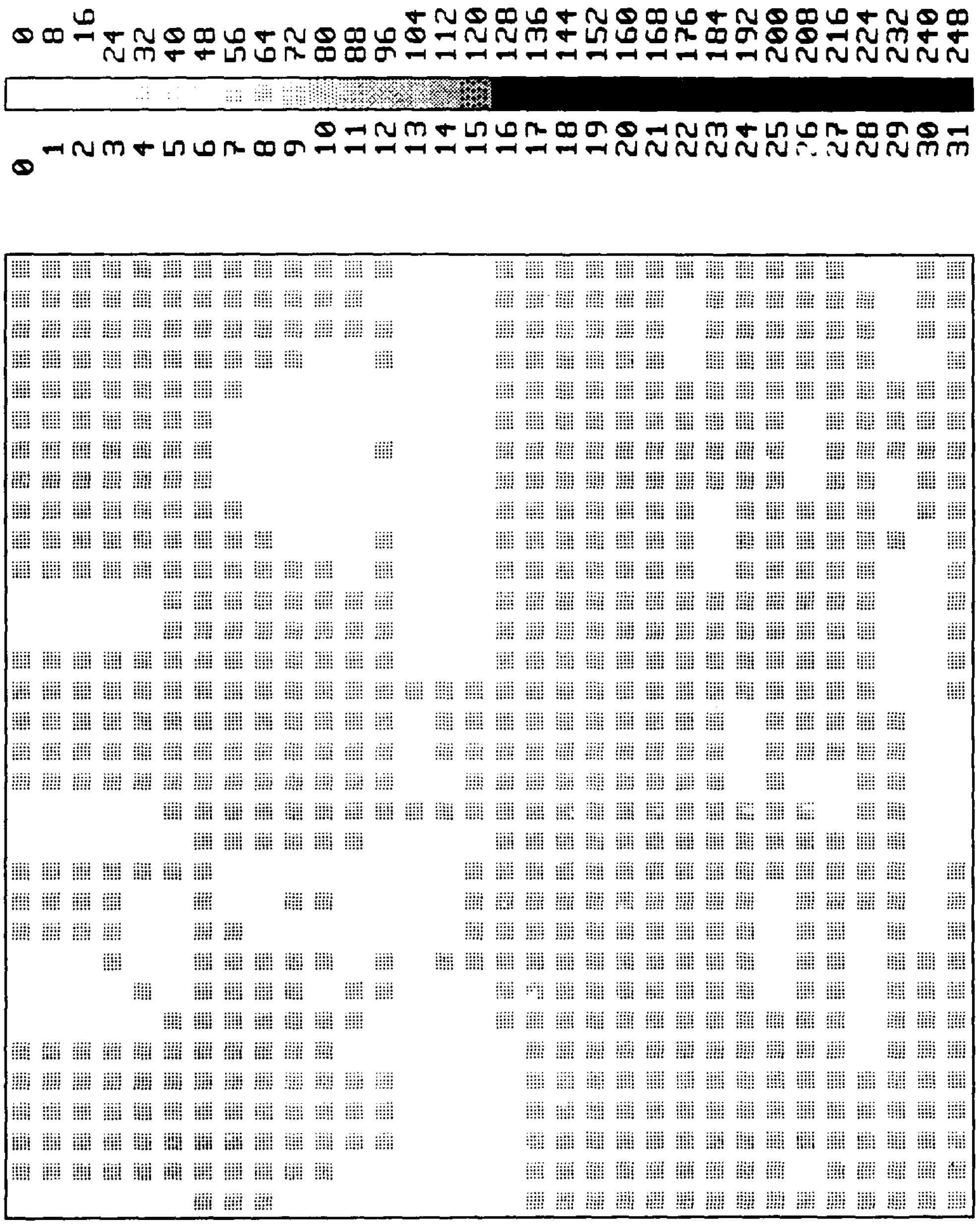


Figure 6. Thresholded Display of Image

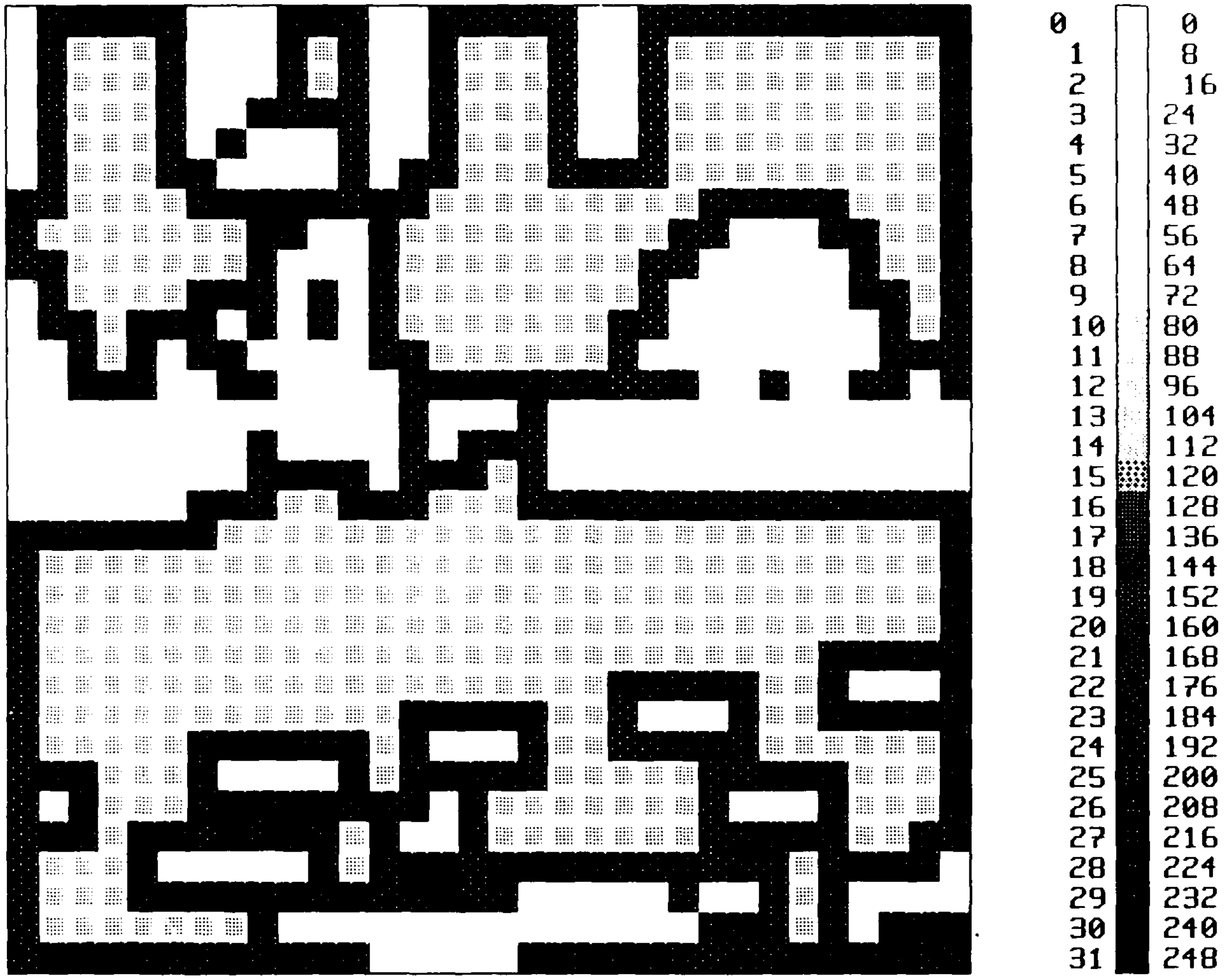


Figure 7. Display of Region Boundaries

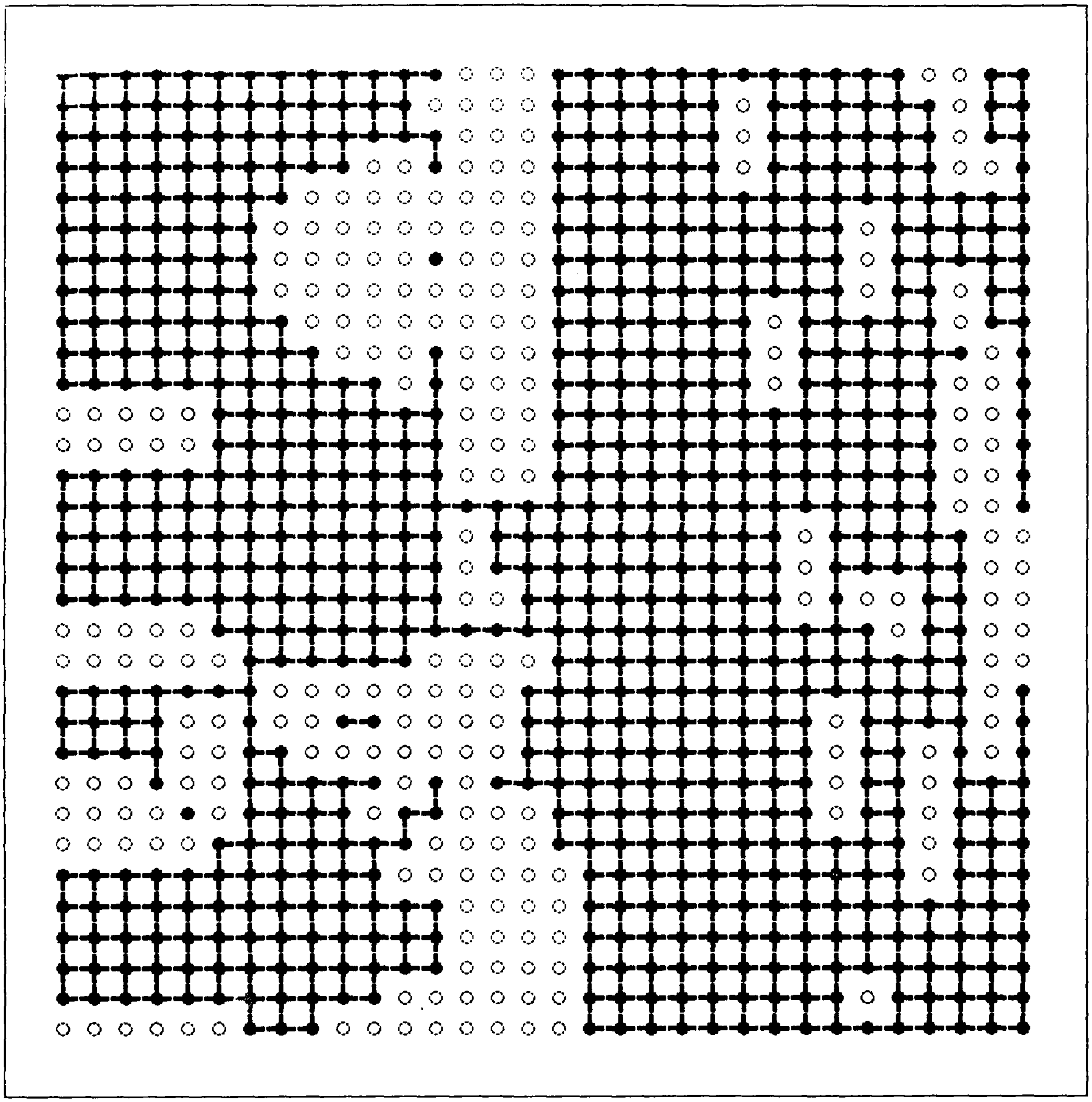


Figure 8. Display of Coterie Network

Appendix D: Documentation for IUA Simulators

Part 2: Sun-3 Version of Simulator

UMIUA Simulator for the SUN

The University of Massachusetts Image Understanding Architecture simulator was made out of several parts previously constructed. These parts were integrated into the SUN Workstation windowing system running under Unix. The parts include:

FORTH: a portable, C-coded figFORTH interpreter written by Allan Pratt and completed April 1985. The FORTH interpreter is used for controlling the UMIUA simulator and substitutes for the ACU in the actual machine.

daughter board: a simulator for the UMIUA motherboard including 64 by 64 CAAPP processing elements.

acu macros: a set of pre-coded CAAPP routines callable from FORTH. These correspond to the routines that will be executed from the ACU micro-programmed controller.

umiua: this is the high level control for the simulator system and includes the drivers for the SUN Window displays. The displays show all the UMIUA registers for all 4096 PEs and the PE instructions executed so far. You may change the registers in individual PEs and show PE memory as an 8 level gray image. The high level control also allows you to transfer images to/from PE memory from/to files under Unix.

screen_dump: this is the program that outputs the display of the simulator that appears on the SUN workstation under SUNTOOLS. You can dump just the simulator or the entire screen. The resulting file can be in SUN rasterfile format, LN03 text file using sixel mapping, or an LLVS plane.

readit: this is a set of image IO routines for *transferring* images to/from the PE memory and Unix files.

Using the UMIUA Simulator

To use the simulator, run the executable **umiua**. The simulator will look for two files **/usr/bin/forth.core** and **/usr/local/forth.block**. If they are not found, the simulator outputs an error message and looks for the files in your working catalog. Once the files have been loaded, the simulator will bring up its display window. You will see two sub-windows. The left sub-window is the control panel and register display. The right sub-window is the FORTH display. Under each SunWindows *button* in the control panel is displayed each PE register.

By positioning the mouse cursor over a button in the left display and pressing the left mouse button, the selected register will be displayed in the right sub-window over part of the FORTH display. This is called the Display window. Selecting the NONE button will remove this display and restore the complete FORTH display.

If you have a register displayed on the right, you may change the contents of an individual PE register by moving the mouse to the Display sub-window and placing it over the position for a single PE. Pressing the left mouse button sets the register to one while pressing the middle mouse button sets the register to zero. When the mouse is moved from the Display window, the register display in the control panel is updated.

The **mem** button displays the location in PE memory specified by the memory address field in the Display window.

The **chip** button displays the CAAPP Chip registers in the Display window while the **inst** button displays the chip and PE instructions executed since the last reset (or until the instruction buffer cycles). The **chip** display also displays the current PE memory allocation map for each physical page of PE memory.

Also included in the control panel is a button marked **gray**. Activating this button causes a multi-level display of PE memory in the display sub-window. The PE memory displayed starts at the address specified by the **Memory Address** field. The number of levels displayed is specified by the **Bits** field. Only 4 levels or less will be displayed regardless of the value in the bits field until you move the mouse to the display sub-window and press the left mouse button. Then, up to eight levels will be displayed. If **Bits** is greater than 8, the highest addressed eight bits will be displayed.

To change the **Memory Address** or **Bits** fields, move the mouse cursor over the field and press the left mouse button. The character cursor will blink in that field. Then, type in the value you want. Use the Delete key to delete existing characters.

The **File Operations** button brings up a pop-up window. You may move this window to any position on the screen using the standard SUN Windows procedures. This window is used to specify operations that are *outside* of the UMIUA architecture. Using this window you can read and write parts of PE memory as LLVS planes. You must specify the PE address, number of bits, and the file name. Then select the appropriate button. You may also use this window to cause a dump of the display to a file for printing out on some other device. You may select one of three different formats: SUN standard RASTERFILE format, LN03 sixels file, and an LLVS plane. (Note - the SUN standard RASTERFILE is incapable of displaying anymore than two levels - black and white.) You can also change the FORTH.BLOCK file. Be sure to do a FLUSH in FORTH after changing this file.

The control panel buttons **Read** and **Write** cause an image to be transferred to and from PE memory starting at the PE address specified in the **Memory Address** field. The number of bits transferred is specified by the **Bits** field which must be less than or equal to 32. When reading, if the number of bits per PE in the file is less than the number of bits specified, zero bits will be moved in. The file affected is specified by the **File** field corresponding to the *button*.

Using FORTH

The right sub-window is the FORTH window. In it are displayed the input sent to the FORTH interpreter and the output generated by your FORTH programs. Everything displayed here is also sent to the standard output file of the window in which you called the simulator (allowing re-direction).

The FORTH interpreter is written entirely in portable C and FORTH. The features include:

- Block file I/O.
- Execution tracing and single-stepping.
- Breakpoint detection and dumping the FORTH stack at the breakpoint.
- Saving and automatic restoration of the FORTH environment.
- Ability to convert the block file to a line-editor-compatible file, and back.
- Interface words for the daughter board simulator and the display driver.

When you start the UMIUA simulator, FORTH will tell you the number of blocks in the block file (either the default or the one you specified with -f). You can see a block with the LIST command: "3 LIST" will list block number 3. Block zero is special: because of a bug in the FORTH standard model, you can't see block zero until you have accessed many other blocks.

Don't use tabs; FORTH doesn't recognize them as whitespace. Everything in the FORTH world is upper-case. Use the CAPS key.

You can leave the UMIUA simulator by typing BYE in the FORTH sub-window.

FORTH Words for the Simulator

Three words have been implemented as primitives in FORTH to control the simulator:

- **CAAPPX (n n -- n)**

  CAAPPX calls the simulator. The top two words on the FORTH stack are combined into a 32 bit UMIUA instruction and sent to the simulator just as the ACU will send a 32 bit bus signal to the mother board.

- **CAAPPD (--)**

  CAAPPD causes the main control to update the control sub-window and the display sub-window.

- **ACU (n1 n2 n3 n4 mac -- n)**

  ACU executes an ACU Macro "mac" using n1, n2, n3, and n4 as the values needed.

The value placed on the FORTH stack after CAAPPX and ACU have been executed is the PE cycles used since the last reset.

PE Instructions in FORTH

To make use of the simulator more convenient, a PE instruction interpreter has been constructed in FORTH allowing you to enter PE instructions in a mnemonic form. Thus,

```
A := VIA-SI ~ ZERO !!
```

can be used to set the every PE A register to one. The syntax is very simple:

```
Destination  := [{{~} $S_i$} op {~} $S_j$ | {op {~}} $S_i$] [!! | A! | S! | N!]
```

PE Register Names

These registers can be either a source or destination: X, Y, A, B. Additional source registers are: ZERO, S, N, E, W. The S and N registers can only be used for S_i, and the E and W registers can only be used for S_j. Additional destination registers are: X-PC, A-X, A-X-I, A-X-J.

Memory locations appear as numeric values: 0, 17, etc. The programmer can store memory locations in variables or symbolic constants, and use them in place of writing a numeric value. Additionally, any numeric expression which evaluates to a positive integer (less than 512) can appear in an instruction.

The carry bit for arithmetic operations is in the Z register. It may also be written to directly via the `Z := SI ...` instruction. The 4-bit Mesh-Register is called MR, and the 8-bit Mesh-Register/Static-Buffer is called MR-SB.

Except for MR and MR-SB, any source register can be complemented by preceding it with a tilde (~) followed by a space, e.g.

`Y := ~ X 'AND B !!.`

Destinations are never complemented. Instead, each function has a complemented form.

PE Function Names

The CAAPP PE functions are: S/N, VIA-SI, VIA-SJ, 'NAND, 'NOR, 'NXOR, 'N+, ICAP-C, SI. Each function has a complemented form: ~S/N, ~VIA-SI, ~VIA-SJ, 'AND, 'OR, 'XOR, '+, ~ICAP-C, ~SI. The functions relating to the MR and MR-SB registers are implemented via assignment statements, e.g. 12 := MR !! would copy mesh-register bit 0 to memory bit 12, mesh-register bit 1 to memory bit 13, etc. The `SI` and `~SI` functions refer to loading the Z-register with the contents of the SI-register, e.g. `Z := SI X !!` would load the contents of the X-register into the Z-register.

PE Activity Control

Every instruction must end in one of four activity control flags: !!, A!, S!, N!. These correspond to: execute always, inhibit if A=0, inhibit if A=0 or S/N=Some, inhibit if A=0 or S/N=None, respectively.

CAAPP Chip Instructions

RESET mode !!	Reset the PEs, BSM, and/or the ICAP
BSM-READ start end column block !!	Read from BSM
BSM-WRITE start end column block !!	Write to BSM
SELECT-RESPONSE source !!	
SET-PR page !!	Set the PR register
LATCH-CNT source !!	Get the ICAP count or PE count
LATCH-LCL-CNT !!	
INJECT-GCN direction !!	Inject Global Coterie Network
HOLD-ICAP !!	
CONNECT-TO memory address !!	
WRITE data !!	
RELEASE-ICAP !!	
GATHER group page byte address !!	Corner turning bits to bytes
SCATTER group page byte address !!	Corner turning bytes to bits

Each function is followed by up to four numeric parameters, separated by spaces, and always ended with "!!", e.g. `BSM-READ 0 4 0 17 !!` transfers data from backing store block 17, column 0, to bytes 0 to 4 of PE memory. Numeric parameters may be replaced with user defined constants or variable values or any expression which evaluates to an integer. The most important is RESET which places the chip in the proper mode for computation and should be done first, e.g. `RESET 5 !!`.

ACU Macros

ACU macros simulate the micro-code sequencer that is provided by the ACU. Because of the way they are implemented in the UMIUA, these macros provide a higher instruction issue rate than can be achieved by simply issuing each PE instruction directly. These macros also provide better performance in the simulator as they do not require the FORTH interpreter for execution of each individual PE instruction. The following is a list of the FORTH words that execute the macros:

ADDFIELDS	(length field1 field2 –)	add field2 to field1
ADDCONSTANT	(length field constant result –)	
SUBFIELDS	(length field1 field2 –)	subtract field2 from field1
SUBCONSTANT	(length field constant result –)	
STORECONSTANT	(length field constant –)	
ANDFIELDS	(length field1 field2 result –)	
ANDCONSTANT	(length field constant result –)	
ORFIELDS	(length field1 field2 result –)	
ORCONSTANT	(length field constant result –)	
XORFIELDS	(length field1 field2 result –)	
XORCONSTANT	(length field constant result –)	
NOTFIELD	(length field –)	
RIGHTSHIFTCOUNT	(length field count-field count-length –)	
RIGHTSHIFT	(length field shift –)	
LEFTSHIFTCONSTANT	(length field count-field count-length –)	
LEFTSHIFT	(length field shift –)	
MOVEFIELD	(length from to –)	
ADD1	(length field –)	
SUBTRACT1	(length field –)	
SETFLOATFMT	(exponent-size mantissa-size guard-size –)	
GETMSB	(exponent-address –)	
NORMALIZE	(mantissa-address exponent-address length –)	
ROUND	(mantissa-address exponent-address –)	
F+	(field1 field2 work-area –)	floating add field2 to field1
F-	(field1 field2 work-area –)	floating subtract field2 from field1
F*	(field1 field2 work-area –)	multiply field1 by field2
1/	(field work-area –)	form the reciprocal of field
F/	(field1 field2 worrk-area –)	floating divide of field1 by field2
COMPARECONSTANT	(length field constant –)	A register set to those that are equal

Other Useful Simulator Words

SET-DISPLAY-INT (interval –) Selects automatic display update after *interval* PE instructions. Use zero for no automatic update.

SET-PE pe !! Set current PE index.

DEPOSIT data bits address !! Deposit bits of data at address for current PE, increment current PE index.

EXAMINE bits address !! Examine bits at address for current PE, increment current PE index. **bits** put in CYCLE.

GET-CNT (– count-register)

GET-S/N (– some/none)

DISP (–) Display the fields of the last instruction sent to the simulator.

INIT (–) reset CYCLES counter

CYCLES @ Get PE cycles used since last RESET.

The following return -1 if the request is not satisfied.

PEM-INIT (page – address) Initializes storage allocation logic for physical page.

PEM-ALLOC (bits page – address) Allocates bits in physical page. Returns the address on the stack.

PEM-FREE (bits address) Un-allocates bits at address in physical page.

PEM-ALLOC-NIBBLE (bits page – address) Same as PEM-ALLOC except that the returned address is on a nibble boundary.

PEM-ALLOC-BYTE (bits page – address) Same as PEM-ALLOC except that the returned address is on a byte boundary.

PEM-RESERVE (bits page address – address) Allocates bits at address in page.

The following FORTH words use the previous ones to allocate temporary storage in PE memory.

PE-SHIFT (length loc dir –) Does a neighborhood shift in direction (`dir`) specified (0 - north, 1 - east, 2 - south, 3 - west). `loc` must be a PE address in P2, `length` is the number of bits shifted. P2 must be initialized using PEM-INIT first.

SPIRAL-OUT (length loc radius dir ftn –) Does a neighborhood shift (see PE-SHIFT) in a spiral pattern with a the specified **radius** (1 - nine shifts, 2 - 25, etc). The location of the data to be shifted (`loc`) can be in any page. P2 must be initialized using PEM-INIT first. The starting direction is `dir`. `ftn` specifies a FORTH word that is called each time after the shift. For example

```
8 0 1 0 ' SUM-WINDOW SPIRAL-OUT
```

would call SUM-WINDOW 9 times and pass to it three arguments - the index (0 .. 8), 8, and the location in P2 where the data is after each shift. SUM-WINDOW might be defined as

```
: SUM-WINDOW ( INDEX LENGTH FIELD -- )
8 SWAP ADDFIELDS . SPACE ;
```

The sum of the field at 0, length 8, of all nine PEs would be left in location 8 and the index would be printed after each sum.

Other FORTH Information

Included with the interpreter is a block file containing an UNTHREAD utility and a screen editor with key-binding and cursor-addressing.

There are two utility filters, b2l and l2b. These convert files from block format to line format and back (b2l - block to line). A line-format file is one suitable for editing with such as **vi** or **emacs**: it consists of a header line for each screen, followed by 16 newline-separated lines of text for that screen, followed by the next screen. *There must always be sixteen lines between headers in the line file, or l2b won't work properly.* You may use l2b to create the block file **forth.block** from **forth.line**. Use:

```
l2b < forth.line > forth.block
```

to do this. Note that **forth.block** is the default block file used by FORTH.

If you use the standard block file, you can load the UNTHREAD utility (see below) with **1 LOAD** (because it starts on block 1), and the editor with **10 LOAD**.

When you have been working in FORTH for a while, you will have developed words which you'd like to save, without having to reload them from the block file all the time. The word (SAVE) will save the current core image on a file, normally **forth.newcore**, and exit. The -s flag on the **umiua** command line changes the save file name.

When you start the UMIUA simulator, the core file (either **forth.core** or the one specified with -c) is checked to see if it is a saved image or a bootstrapped image. If it's a saved image, execution begins from the spot from which it left off.

Summary of Umiua Command Line Options

-t[n] trace; n is a digit from 0 to 9, default 0.

Each tim. through the inner interpreter, a line will be printed out showing the current stack pointer, the top n stack elements (topmost at the left), the current interpretive pointer (ip), an indent to reflect the current nesting depth (actually the return-stack depth), and the name of the word about to be executed. N is the *trace depth*; see DOTRACE below.

-d[n] debug; n is a digit from 0 to 9, default 0.

Like -t, -d prints out the trace line each time through the inner interpreter. Unlike -t, it will then wait for input from the terminal. If you hit newline (Return, CR, etc.) it will proceed. If you type any key followed by newline, it will dump the current memory image to the dump file, usually "forth.dump", and then continue.

-n no setbuf.

If -n is present, the setbuf(stdout,NULL) call which makes stdout unbuffered instead of line-buffered will not be executed. This is useful if you intend to do debugging with -t, -d, or the TRON command (see below).

-p xxxx breakpoint.

Breakpoints are enabled, and one is set at address xxxx (in hex). Each time through the inner interpreter, the ip address is checked against this breakpoint address. If they match, "Breakpoint" is printed, along with the current stack pointer and the entire contents of the stack, with the topmost element at the left.

-c corename set the core file name

The memory image will be read from this file instead of the default, which is "forth.core".

-b blockname set the block file name

This file will be used as the block file for disk reads and writes, instead of the default block file, which is `forth.block`.

-s savename set the core-save file name

This file will be created or overwritten upon execution of the (SAVE) primitive. It will contain a core image (just like forth.core or the -c name) which reflects the current status at the time of the save. If this file is used as input (renamed to `forth.core` or used in -c later), the FORTH system will restart right where it left off. Note that -c and -s MAY use the same name.

Departures from the figFORTH-79 Standard

There are two features of the FORTH-79 standard which are unimplemented in this system: the **<BUILDS . . . DOES>** construct and VOCABULARIES. They do not affect the operation of FORTH except that the dictionary is simply a flat stack structure and defining-words using DOES doesn't work.

Extensions from the figFORTH-79 Standard

- **CASE c1 OF action1 ENDOF c2 OF action2 ENDOF default-action ENDCASE**

  This construct comes from Doctor Dobb's Journal, Number 59, September, 1984, in an article by Ray Duncan.

- \ (backslash)

  This word begins a comment which extends to the end of the current line. This is only meaningful while loading, and causes an error if you are not loading. It amounts to an open-paren where the end of the line is the closing paren. This, too, comes from Dr. Dobb's Journal, Number 59, in a different article, by Henry Laxen.

- **TRON, TROFF, DOTRACE**

  These provide tracing facilities for C-FORTH. TRON takes one parameter, which is the number of stack elements to show. TROFF disables tracing. DOTRACE traces once, using the most recent depth value (from TRON or -tn on the command line). TRON will have no effect (but will still consume its argument) if the FORTH system was compiled without the TRACE flag set. DOTRACE, however, will still trace one line as advertised.

- **REFORTH**

  This word is useful when loading screens, to get the user's input. It reads one line from the terminal (with QUERY) and interprets it (with INTERPRET), then returns to the caller. This is used in the editor (see below) to read in the user's terminal type. This is yet another construct from Ray Duncan's article in Dr. Dobb's Journal, Number 59.

- `ALIAS` usage: `ALIAS NEW OLD`

  This word is used to change the meaning of an existing word, after it has been used in defining other words. Say you want to change EMIT so it forces a CR when the current column (user var OUT) equals the number of characters per line (constant C/L). The current definition of EMIT is:

  ```
  : EMIT (EMIT) 1 OUT !+ ;
  ```

  You could define a new word, NEWEMIT, as follows:

  ```
  : NEWEMIT
    OUT @ C/L = IF CR THEN
    (EMIT) 1 OUT +! ;
  ```

  and use ALIAS NEWEMIT EMIT to make all previous references to EMIT execute NEWEMIT instead. This is accomplished by making the definition of EMIT read:

  ```
  : EMIT NEWEMIT ;
  ```

  (Note that CR sets OUT to zero, so this really would work.)

User's Guide for the Editor

This is a screen editor for FORTH screens. It will alter the file `forth.block` in the current directory, or whatever the installer set the default blockfile to be, or whatever you specify on the command line with the `-b file` switch.

You call a screen up with the EDIT command:

```
3 EDIT
```

will begin editing screen 3. The file starts with screen 0, but, due to a bug in figFORTH which has been carefully preserved (:-), you can't see screen 0 until you've edited several others – as many as fit in memory, in fact. This number is 5 at the outset, but can be as low as 2 or as many as your installer's whim dictated. In any case, stick with screens numbered from 1 to the size of the block file as displayed when you started FORTH.

When you start editing, the screen will clear, and the screen you asked for will appear. FORTH "screens" are, by convention and convenience, sixteen lines by sixty-four characters, for a total of 1K per screen. At the bottom of the screen being edited will be the word SCREEN and this screen's number.

The keys match WordStar cursor movement keys: `^E`, `^D`, `^X`, `^S` move one character or line at a time while `^A` and `^F` move one word backward and forward, and `^R` and `^C` move one SCREEN backward and forward. If you have not disabled `^C` as your Unix interrupt character, use ESC-C (escape is a meta-prefix like in Emacs). `^H` can be used for backspace too.

To see all the key bindings, use the FORTH command DESCRIBE-BINDINGS. To make a binding, do:

1. Type `BIND-TO-KEY function <CR>` where function is the name of the command you want bound to a key, and `<CR>` means press return/newline.
2. The computer will print `KEY:` at which time you should strike the key you want to have that function bound to. If you want something bound to ESC plus something, just press escape followed by the key, exactly as you intend to use the key in practice.

To see the binding for one key, use the command DESCRIBE-KEY. Again, you will be asked for the key you want described. Strike the key just as you normally do, with ESC before it if necessary.

The initial bindings are on screens 27 and 28 of the standard `forth.block` file.

The editor is always in replace mode. That is, when you strike a key, it overwrites whatever was under the cursor. There is *no* insert mode.

As you go around changing things, you are only changing the image of the screen which FORTH keeps in memory; you are not changing the block file. You can mark the current screen for updating to the blockfile with `^U`. This only marks the screen; it won't be written to the blockfile until the space is needed again, or you explicitly FLUSH the memory buffers. Use ESC-F (press escape, then press capital F) to mark the current screen for updating, and then FLUSH all marked screens (including this one, now) to the block file. Once you've done that, the change is permanent.

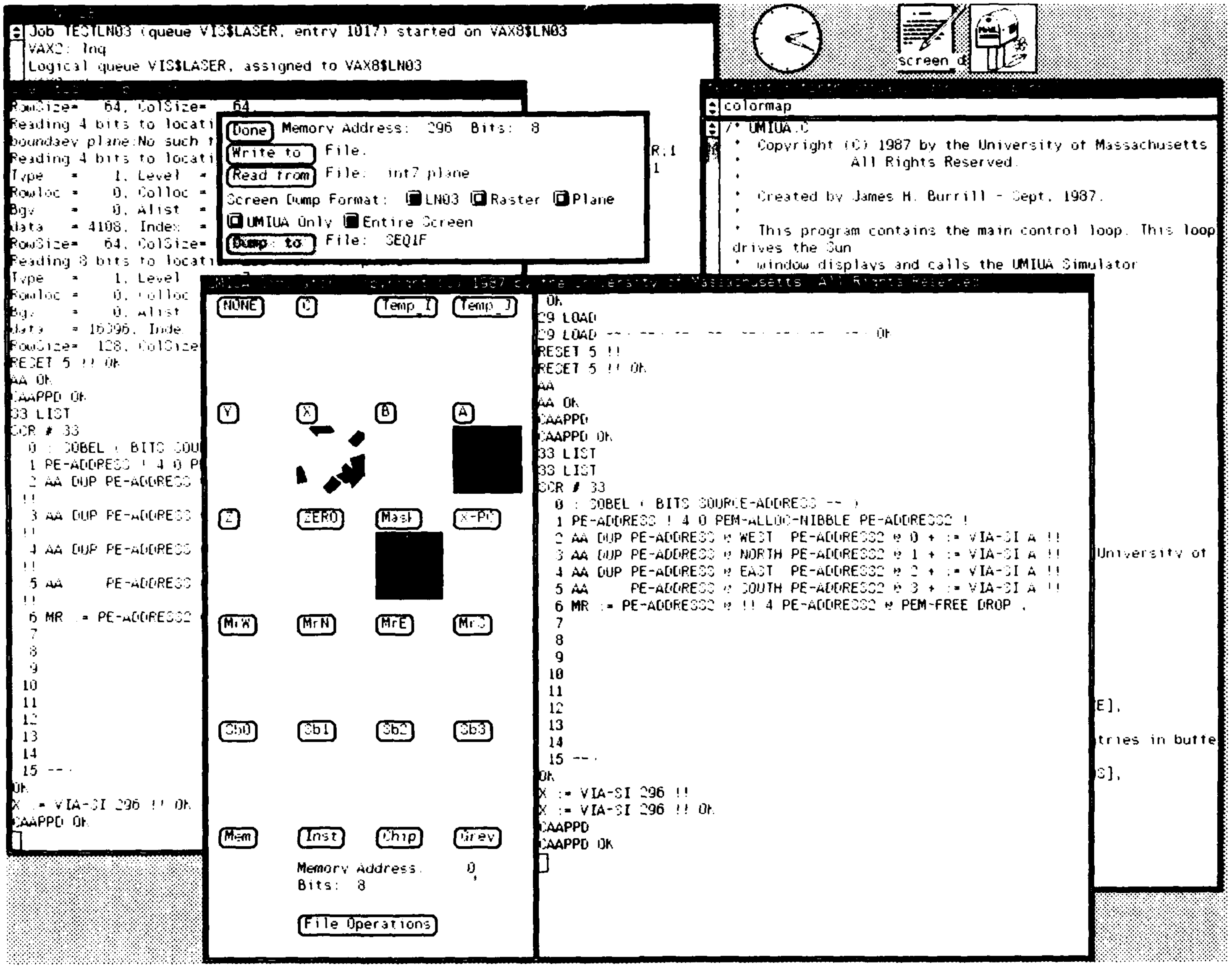
If you really mean to make a given change to a screen, mark it as updated (`^U`) right away. Countless hours have been wasted changing screens without marking them for writing, and then having that memory reused losing the changes.

You can leave the editor with ESC-Q (for Quit). Be sure to do an ESC-F before you do, if you want to save what you have. You can never be sure if your changes will get written out if you don't.

Summary of Files

makefile	create the **umiua** executable from the object files
b2l	filter to convert block-files into line-files for editing
l2b	filter to convert line-files into block-files for FORTH
umiua	the main executable
umiua.o	the control program and display driver object file
daughter_board.o	the CAAPP Chip simulator object file
acu_macros.o	the ACU micro-controller macro instructions object file
readit.c	file transfer source code
readit.o	file transfer object file
forth.o	the FORTH interpreter object file
screen_dump.o	the screen dump object file for SUN Windows
forth.block	This is the (default) block-file used by FORTH for its editing- and load-screens
forth.line	This file usually resembles forth.block, but is in a format suitable for editing with emacs or vi: a header line, followed by sixteen lines of trailing-blank- truncated, newline-terminated text for each screen. If one of forth.line and forth.block is out of date with respect to the other, you can bring it back up to date by using b2l or l2b.
forth.core	This contains the core image for the FORTH environment.
forth.newcore	This is the file for holding core images saved with the (SAVE) primitive. If FORTH is started with `-c forth.newcore`, the image is restarted right where it left off.
*.plane	These are LLVS image files.

Sample Sun-3 Displays from Simulator



```
Job TESTLN03 (queue VIS$LASER, entry 1017) started on VAX8$LN03
VAX2: Inq
Logical queue VIS$LASER, assigned to VAX8$LN03
```

```
RowSize=   64, ColSize=   64
Reading 4 bits to locati
boundaev plane:No such f
Reading 4 bits to locati
Type    =    1, Level   =
Rowloc  =    0, Colloc  =
Bgv     =    0, Alist   =
data    = 4108, Index   =
RowSize=   64, ColSize=
Reading 8 bits to locati
Type    =    1, Level
Rowloc  =    0, Colloc
Bgv     =    0, Alist
data    = 16396, Inde
RowSize=  128, ColSize
RESET 5 !! Ok
AA Ok
CAAPPD Ok
33 LIST
SCR # 33
  0 : SOBEL ( BITS SOU
  1 PE-ADDRESS ! 4 0 P
  2 AA DUP PE-ADDRESS
 !!
  3 AA DUP PE-ADDRESS
 !!
  4 AA DUP PE-ADDRESS
 !!
  5 AA      PE-ADDRESS
 !!
  6 MR  = PE-ADDRESS2
  7
  8
  9
 10
 11
 12
 13
 14
 15 --
Ok
X := VIA-SI 296 !! Ok
CAAPPD Ok
```

Done Memory Address: 296 Bits: 8

Write to File.

Read from File: int7 plane

Screen Dump Format: ■ LN03 □ Raster □ Plane

□ UMIUA Only ■ Entire Screen

Dump to File: SEQ1F

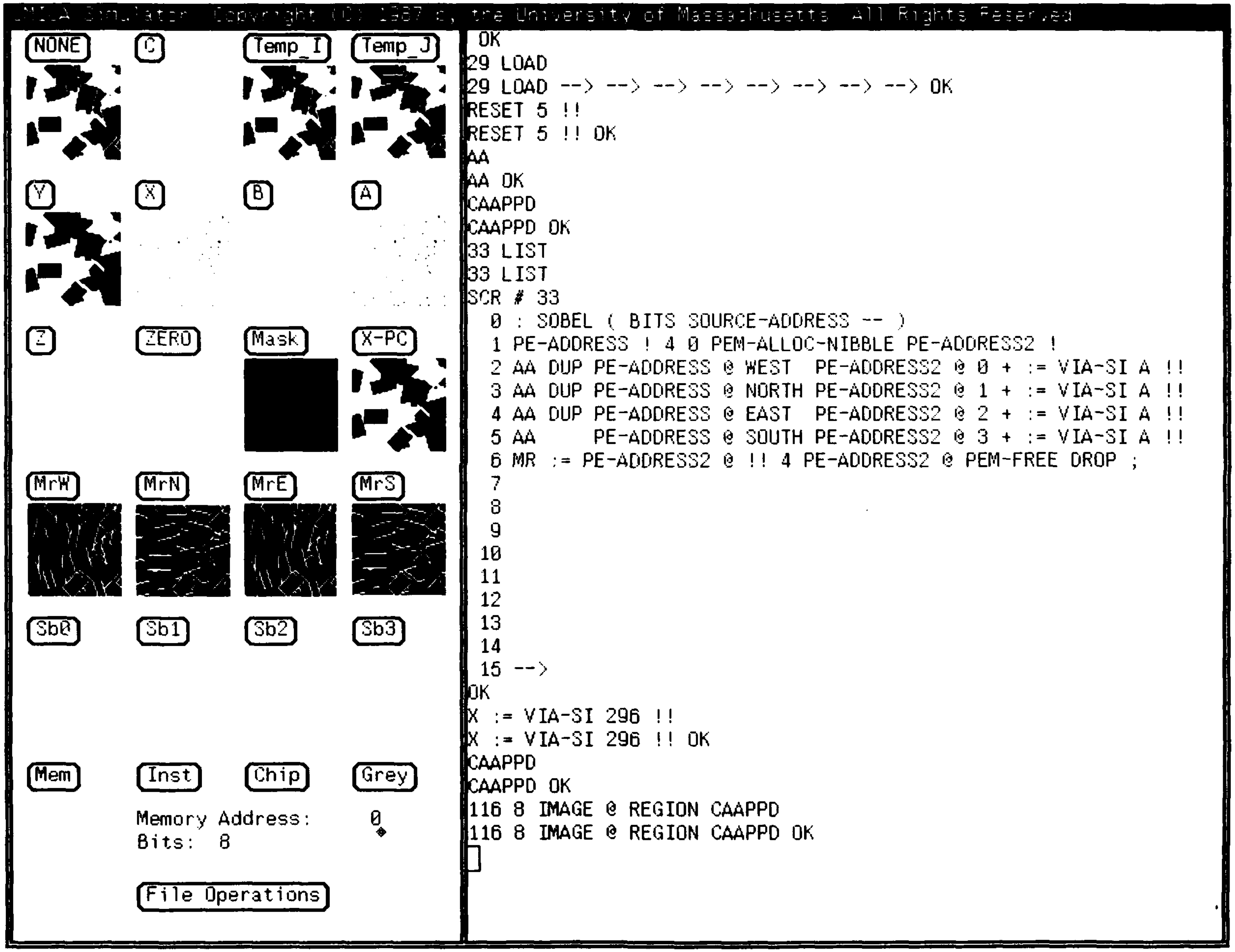
```
colormap
/* UMIUA.C
 *  Copyright (C) 1987 by the University of Massachusetts
 *           All Rights Reserved.
 *
 *  Created by James H. Burrill - Sept. 1987.
 *
 *  This program contains the main control loop. This loop
drives the Sun
 *  window displays and calls the UMIUA Simulator
```

Memory Address: 0

Bits: 8

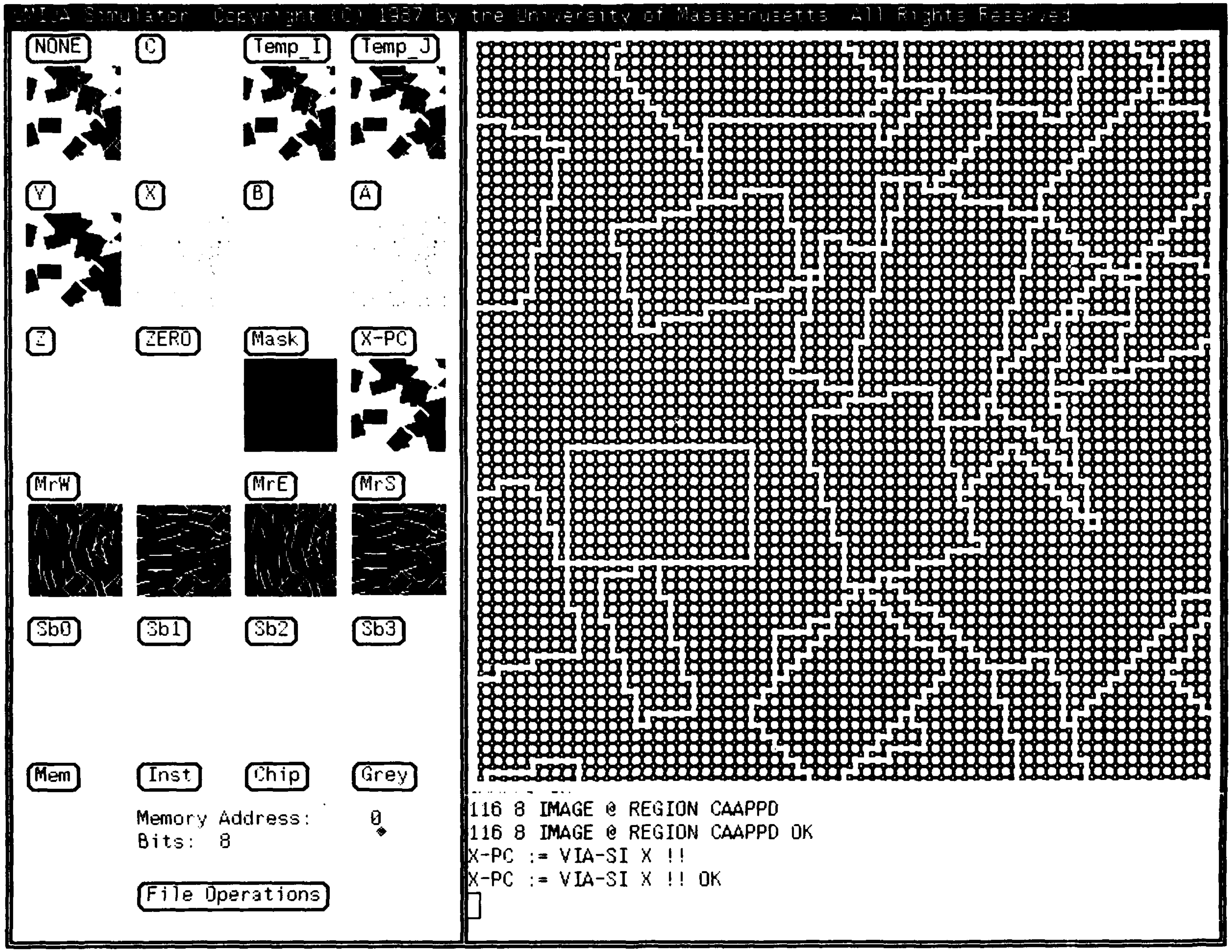
```
 Ok
29 LOAD
29 LOAD -- -- -- -- -- -- -- -- Ok
RESET 5 !!
RESET 5 !! Ok
AA
AA Ok
CAAPPD
CAAPPD Ok
33 LIST
33 LIST
SCR # 33
  0 : SOBEL ( BITS SOURCE-ADDRESS -- )
  1 PE-ADDRESS ! 4 0 PEM-ALLOC-NIBBLE PE-ADDRESS2 !
  2 AA DUP PE-ADDRESS @ WEST  PE-ADDRESS2 @ 0 + := VIA-SI A !!
  3 AA DUP PE-ADDRESS @ NORTH PE-ADDRESS2 @ 1 + := VIA-SI A !!
  4 AA DUP PE-ADDRESS @ EAST  PE-ADDRESS2 @ 2 + := VIA-SI A !!
  5 AA     PE-ADDRESS @ SOUTH PE-ADDRESS2 @ 3 + := VIA-SI A !!
  6 MR := PE-ADDRESS2 @ !! 4 PE-ADDRESS2 @ PEM-FREE DROP .
  7
  8
  9
 10
 11
 12
 13
 14
 15 --
Ok
X := VIA-SI 296 !!
X := VIA-SI 296 !! Ok
CAAPPD
CAAPPD Ok
```

Figure 1. Display of Simulator Environment



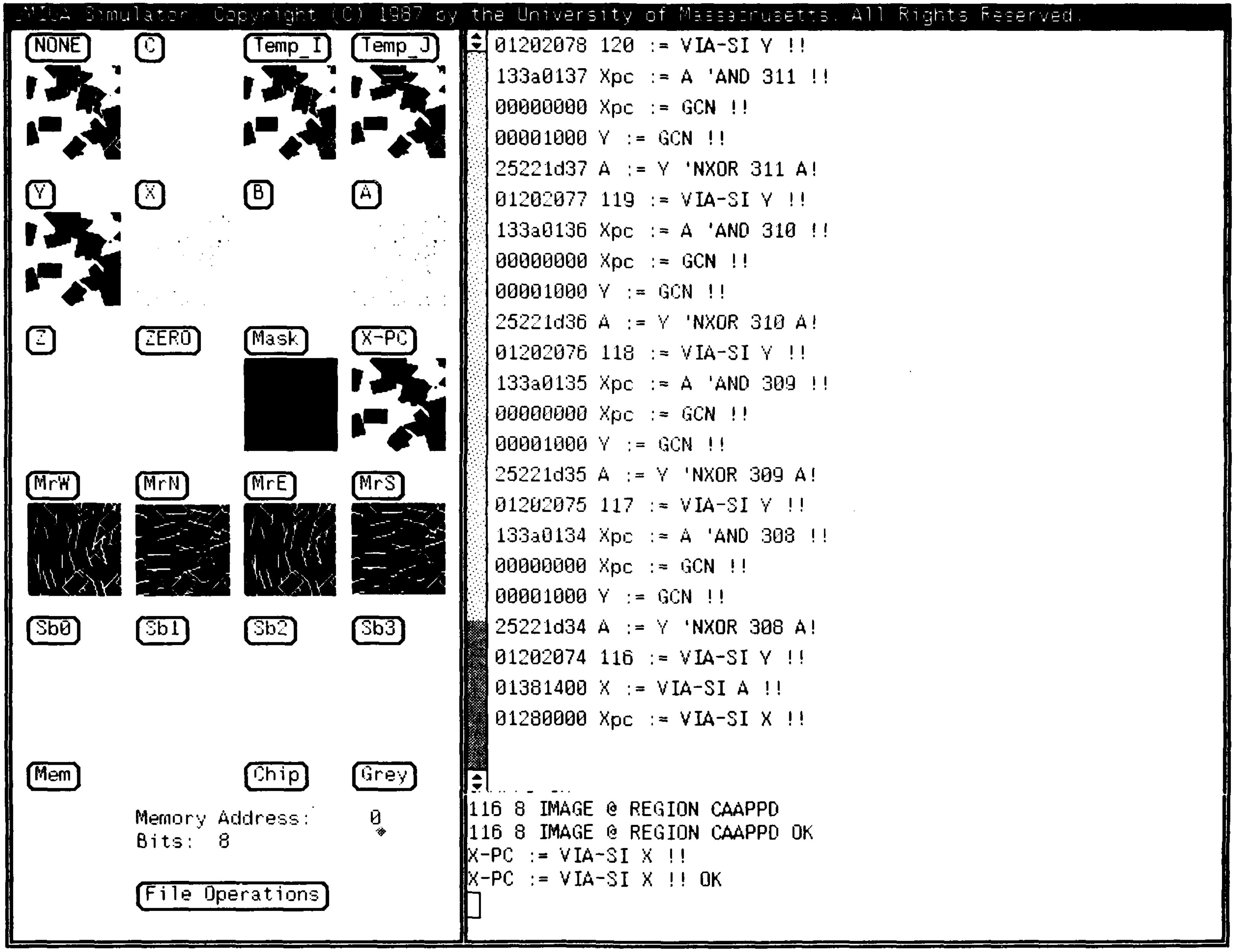
```
 OK
29 LOAD
29 LOAD --> --> --> --> --> --> --> --> OK
RESET 5 !!
RESET 5 !! OK
AA
AA OK
CAAPPD
CAAPPD OK
33 LIST
33 LIST
SCR # 33
  0 : SOBEL ( BITS SOURCE-ADDRESS -- )
  1 PE-ADDRESS ! 4 0 PEM-ALLOC-NIBBLE PE-ADDRESS2 !
  2 AA DUP PE-ADDRESS @ WEST  PE-ADDRESS2 @ 0 + := VIA-SI A !!
  3 AA DUP PE-ADDRESS @ NORTH PE-ADDRESS2 @ 1 + := VIA-SI A !!
  4 AA DUP PE-ADDRESS @ EAST  PE-ADDRESS2 @ 2 + := VIA-SI A !!
  5 AA     PE-ADDRESS @ SOUTH PE-ADDRESS2 @ 3 + := VIA-SI A !!
  6 MR := PE-ADDRESS2 @ !! 4 PE-ADDRESS2 @ PEM-FREE DROP ;
  7
  8
  9
 10
 11
 12
 13
 14
 15 -->
OK
X := VIA-SI 296 !!
X := VIA-SI 296 !! OK
CAAPPD
CAAPPD OK
116 8 IMAGE @ REGION CAAPPD
116 8 IMAGE @ REGION CAAPPD OK
```

Figure 2: Display of Program Screen



```
116 8 IMAGE @ REGION CAAPPD
116 8 IMAGE @ REGION CAAPPD OK
X-PC := VIA-SI X !!
X-PC := VIA-SI X !! OK
```

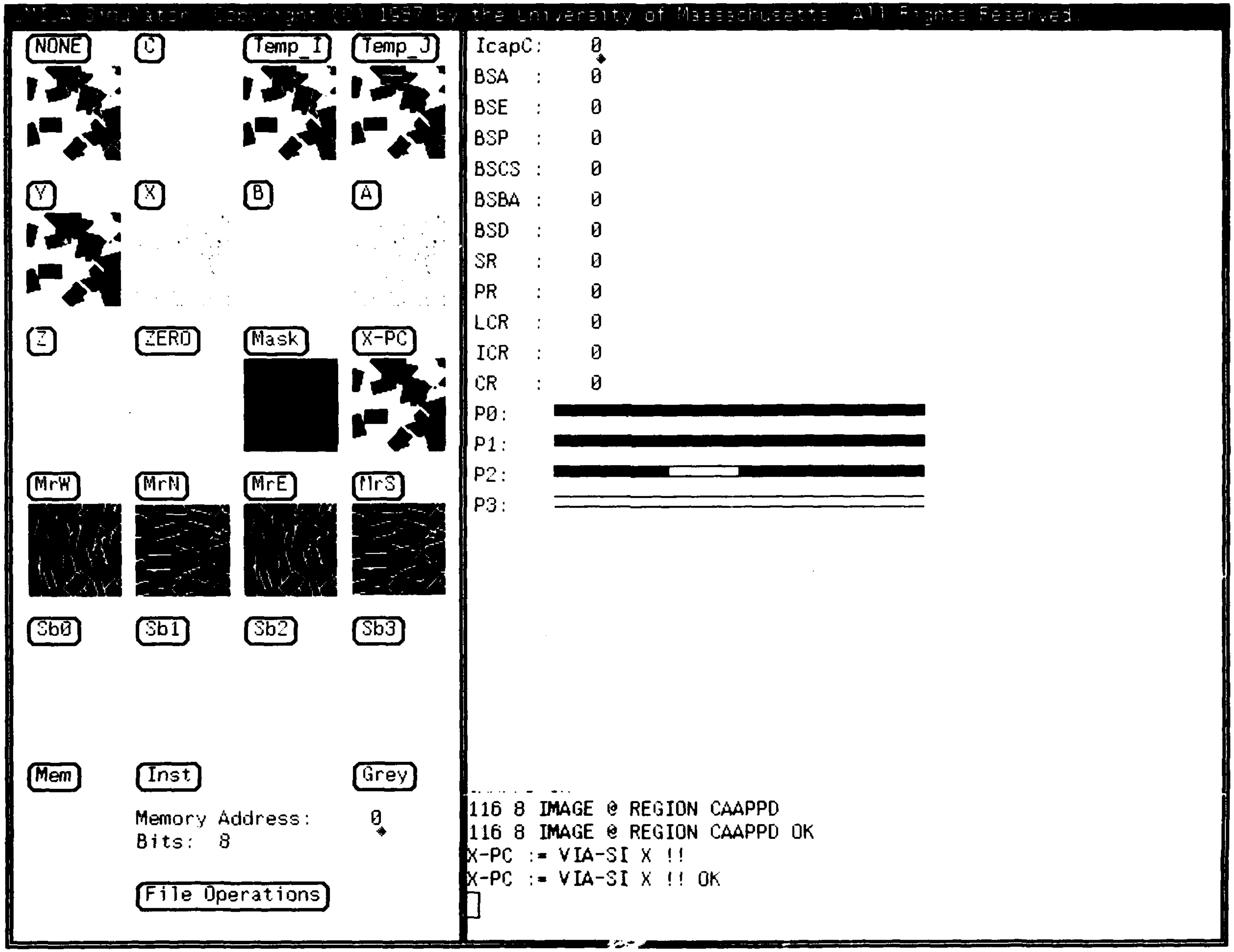
Figure 3. Display of Coterie Network



```
01202078 120 := VIA-SI Y !!
133a0137 Xpc := A 'AND 311 !!
00000000 Xpc := GCN !!
00001000 Y := GCN !!
25221d37 A := Y 'NXOR 311 A!
01202077 119 := VIA-SI Y !!
133a0136 Xpc := A 'AND 310 !!
00000000 Xpc := GCN !!
00001000 Y := GCN !!
25221d36 A := Y 'NXOR 310 A!
01202076 118 := VIA-SI Y !!
133a0135 Xpc := A 'AND 309 !!
00000000 Xpc := GCN !!
00001000 Y := GCN !!
25221d35 A := Y 'NXOR 309 A!
01202075 117 := VIA-SI Y !!
133a0134 Xpc := A 'AND 308 !!
00000000 Xpc := GCN !!
00001000 Y := GCN !!
25221d34 A := Y 'NXOR 308 A!
01202074 116 := VIA-SI Y !!
01381400 X := VIA-SI A !!
01280000 Xpc := VIA-SI X !!
```

```
116 8 IMAGE @ REGION CAAPPD
116 8 IMAGE @ REGION CAAPPD OK
X-PC := VIA-SI X !!
X-PC := VIA-SI X !! OK
```

Figure 4. Display of Instruction Log Buffer



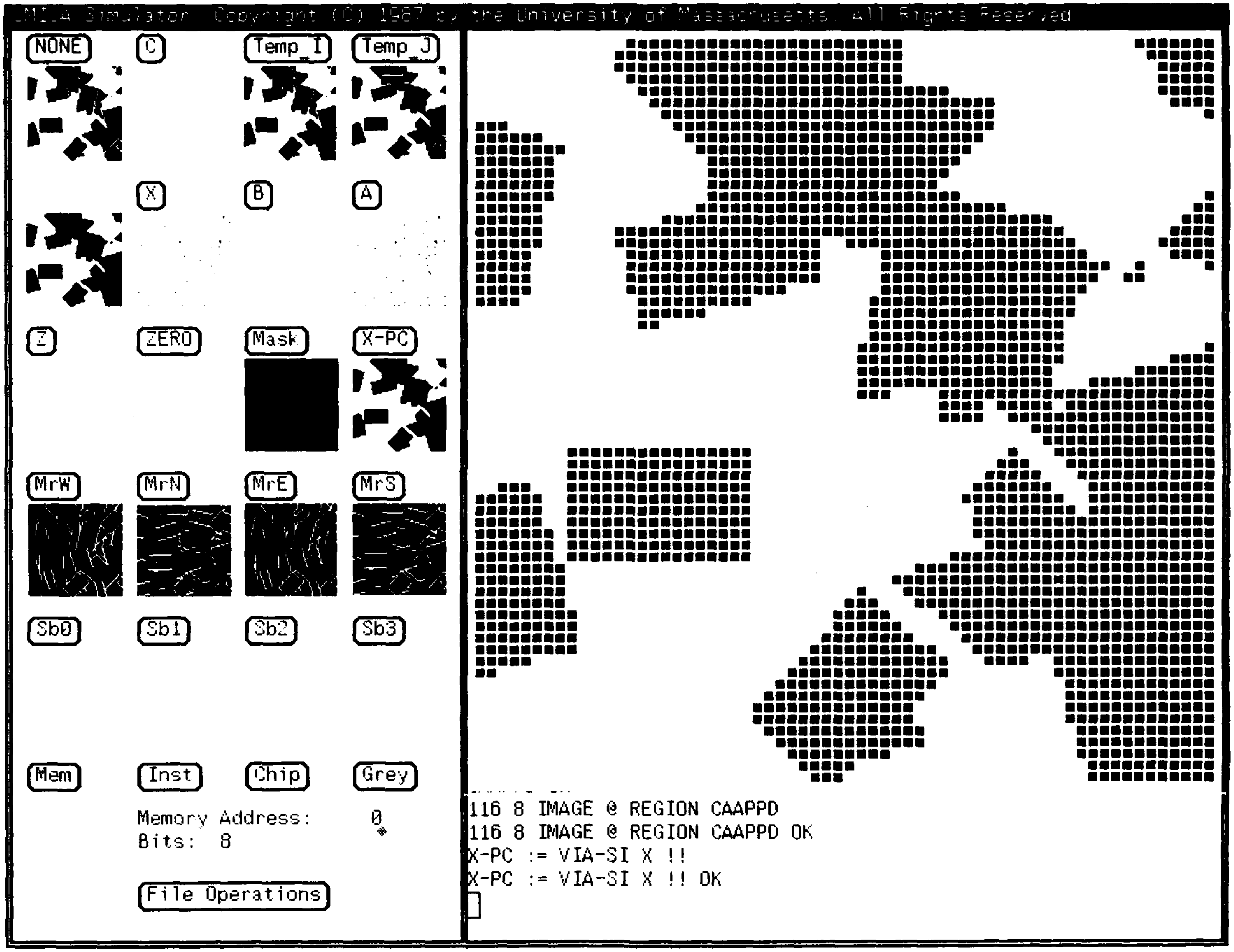
IUA Simulator Copyright (C) 1987 by the University of Massachusetts All Rights Reserved.

```
Memory Address:    0
Bits:  8
```

```
IcapC:     0
BSA   :    0
BSE   :    0
BSP   :    0
BSCS  :    0
BSBA  :    0
BSD   :    0
SR    :    0
PR    :    0
LCR   :    0
ICR   :    0
CR    :    0
P0:
P1:
P2:
P3:
```

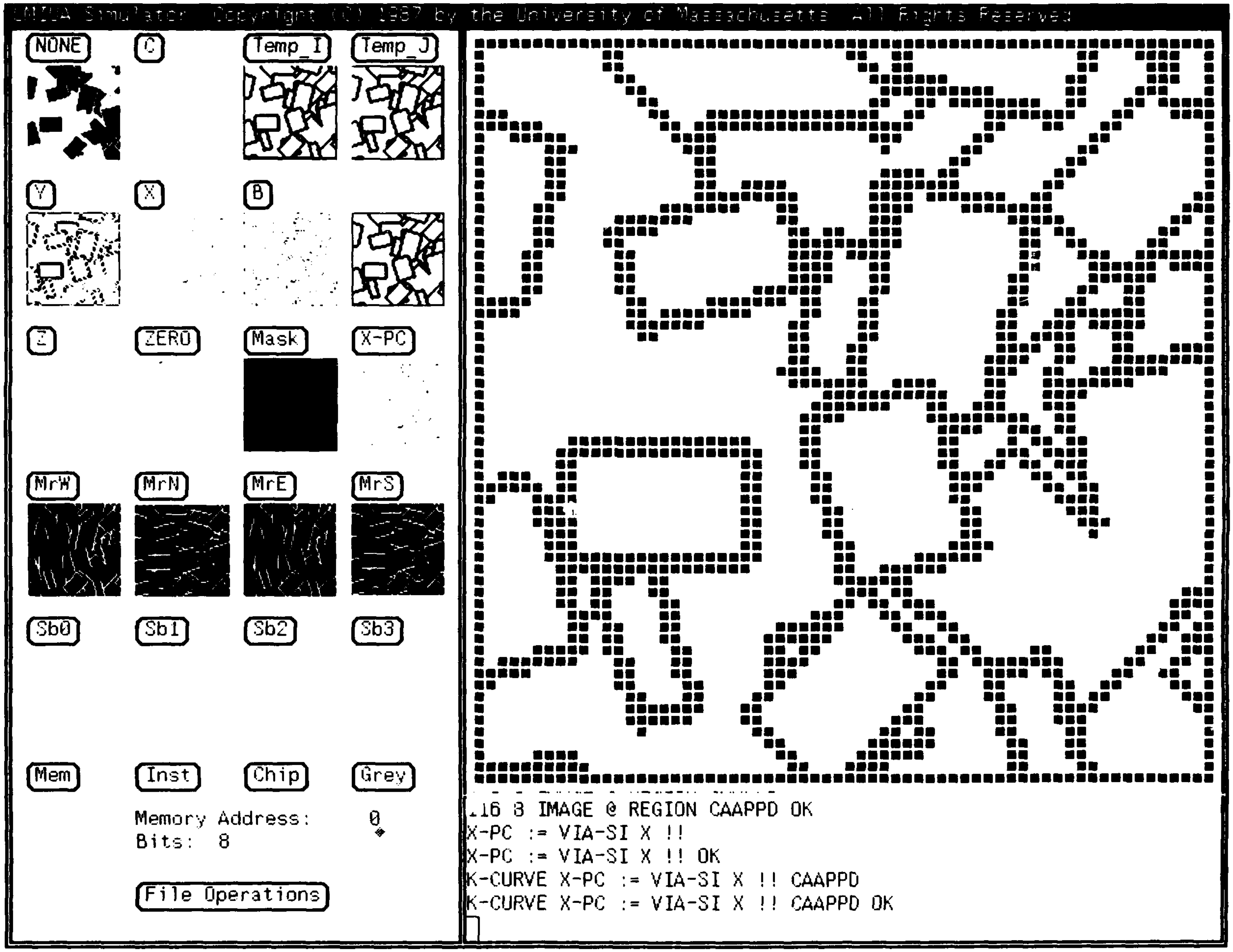
```
116 8 IMAGE @ REGION CAAPPD
116 8 IMAGE @ REGION CAAPPD OK
X-PC := VIA-SI X !!
X-PC := VIA-SI X !! OK
```

Figure 5. Display of ACU Registers and Memory Allocation



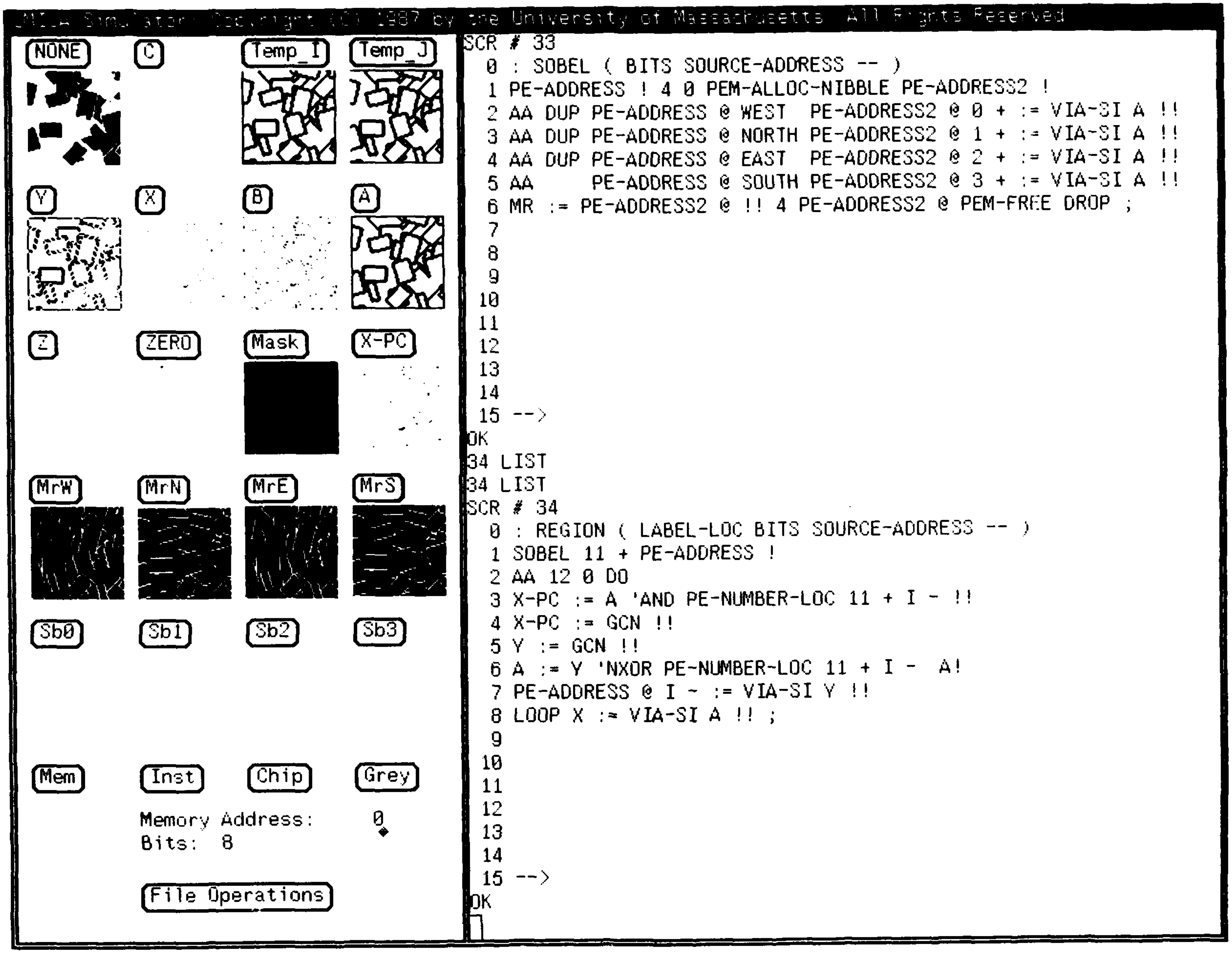
```
116 8 IMAGE @ REGION CAAPPD
116 8 IMAGE @ REGION CAAPPD OK
X-PC := VIA-SI X !!
X-PC := VIA-SI X !! OK
```

Figure 6. Display of Y Registers



```
.16 8 IMAGE @ REGION CAAPPD OK
X-PC := VIA-SI X !!
X-PC := VIA-SI X !! OK
K-CURVE X-PC := VIA-SI X !! CAAPPD
K-CURVE X-PC := VIA-SI X !! CAAPPD OK
```

Figure 7. Display of Activity Pattern



```
SCR # 33
  0 : SOBEL ( BITS SOURCE-ADDRESS -- )
  1 PE-ADDRESS ! 4 0 PEM-ALLOC-NIBBLE PE-ADDRESS2 !
  2 AA DUP PE-ADDRESS @ WEST  PE-ADDRESS2 @ 0 + := VIA-SI A !!
  3 AA DUP PE-ADDRESS @ NORTH PE-ADDRESS2 @ 1 + := VIA-SI A !!
  4 AA DUP PE-ADDRESS @ EAST  PE-ADDRESS2 @ 2 + := VIA-SI A !!
  5 AA     PE-ADDRESS @ SOUTH PE-ADDRESS2 @ 3 + := VIA-SI A !!
  6 MR := PE-ADDRESS2 @ !! 4 PE-ADDRESS2 @ PEM-FREE DROP ;
  7
  8
  9
 10
 11
 12
 13
 14
 15 -->
OK
34 LIST
34 LIST
SCR # 34
  0 : REGION ( LABEL-LOC BITS SOURCE-ADDRESS -- )
  1 SOBEL 11 + PE-ADDRESS !
  2 AA 12 0 DO
  3 X-PC := A 'AND PE-NUMBER-LOC 11 + I - !!
  4 X-PC := GCN !!
  5 Y := GCN !!
  6 A := Y 'NXOR PE-NUMBER-LOC 11 + I -  A!
  7 PE-ADDRESS @ I - := VIA-SI Y !!
  8 LOOP X := VIA-SI A !! ;
  9
 10
 11
 12
 13
 14
 15 -->
OK
```

Figure 8. Display of Sobel Program

Appendix E:
Report of Efforts at Hughes

Image Understanding Architecture Program

1.0 Introduction

This report is a summary of the design/architecture activities over the past year as part of a DARPA/ETL funded program to build a feasibility prototype of the Image Understanding Architecture (IUA). At the beginning of the program the basic multi-level architecture had been defined; thus much of the effort has revolved around the IUA system "micro-architecture" definition performed jointly by UMASS and Hughes Research Labs, and the associated implementation of this micro-architecture. Key issues were the need for:

- MSIMD-like operation (e.g., for connected component labeling)
- Local summation (e.g., Hough transform, statistics)
- High density local memory for large problems
- Efficient usage of the CAAPP PE
- Macro instruction expander to support a virtual MSIMD capability
- Low cost solution to high component density/pin-out restrictions
- Fast, simple means for I/O
- Efficient, transparent utilization of resources
- Powerful controller scalable to a custom chip
- Support for corner turning and floating point operations
- Efficient means for communication of data between layers

This report summarizes our approach to achieving these various goals. The basic ingredients in the micro-architecture were:

- Some/none local mesh
- Local count using Some/none
- Backing store with dual port VRAM chips
- Double buffering
- Provision for future local controller
- Motherboard/daughterboard combination
- Intelligent frame buffer distributed among CAAPP chips
- Efficient CAAPP design (108,000) transistors in ≈280 x 260 mils^2
- Byte wide data path

The basic IUA includes 4096 CAAPP cells, 64 ICAP cells, a single SPA processor, and an array control unit. The entire prototype will plug into a single-user VME based workstation that will serve as a host. This prototype consists of one motherboard containing 64 daughterboards, a concentrator board, an interconnect board and four driver boards. The daughterboard contains the custom CAAPP chips, the TI TMS320C25 Digital Signal Processing (DSP) chip for the ICAP level, plus 256K bytes of static (SRAM) and 448K bytes of dynamic video memory (VRAM). Each of the 64 bit-serial processing elements (PEs) on the CAAPP chip contain 320 bits of local data storage. This custom VLSI chip was designed in 2-micron CMOS with 2 metal layers and contains approximately 108,000 devices. The concentrator board will be used to provide summary information from the some/none lines plus status information from the TI processors at ICAP level. The interconnect board will contain special crossbar switch chips that allow the TI processors to communicate in a bit serial mode. Since the purpose of the prototype slice was not simply demonstration of a concept, but ultimately to produce a research machine, it was decided to pay special attention to providing sufficient memory to deal with full image arrays. Clearly, a machine with the computing power of 64 5-MIP DSP chips and 4096 10-MHz bit-serial processors could not be utilized efficiently without large amounts of memory with a high bus bandwidth. Therefore, enough chips were put on the daughterboards to provide 44 MBytes of memory for the entire prototype.

2.0 Micro-Architecture of the IUA

Intended to be able to support the entire range of vision algorithms, IUA includes a host and three heterogeneous parallel processor arrays: CAAPP (512x512), ICAP (64x64), and SPA (8x8), corresponding to fine, medium and coarse-grained architectures. Tightly integrated with these various processors are an evenly distributed set of dual-port video-RAM shared memories (SM) between adjacent layers. These are labeled as HCSM, CISM, ISSM, and SHSM, corresponding to HOST, CAAPP, ICAP, and SPA. HCSM and CISM can only be shared locally, while ISSM and SHSM can be shared globally via a switch network. Each type of PE also has its own local memory. The overall memory structure is shown in Figure 1. In general, as the level of abstraction in the analysis process increases, the center of activity moves up the hierarchy. These dual port memories act as a conduit of information between the different levels. The communication within the CAAPP layer is full-mesh with gated connection capability, 2-D bit serial crossbar for ICAP, and shared memory via crossbar for SPA. An example of a future crossbar interconnection network between ICAP processors based on a 64x64 crossbar switch chip is shown in Figure 2. A program residing in the Array Control Unit (ACU) mediates system activity. Programs may be composed of CAAPP PE, CAAPP Chip, ACU Macro, and ACU Local instructions. The ACU can perform functions such as holding the ICAP, loading routines to it, and interrupting it to service routine.

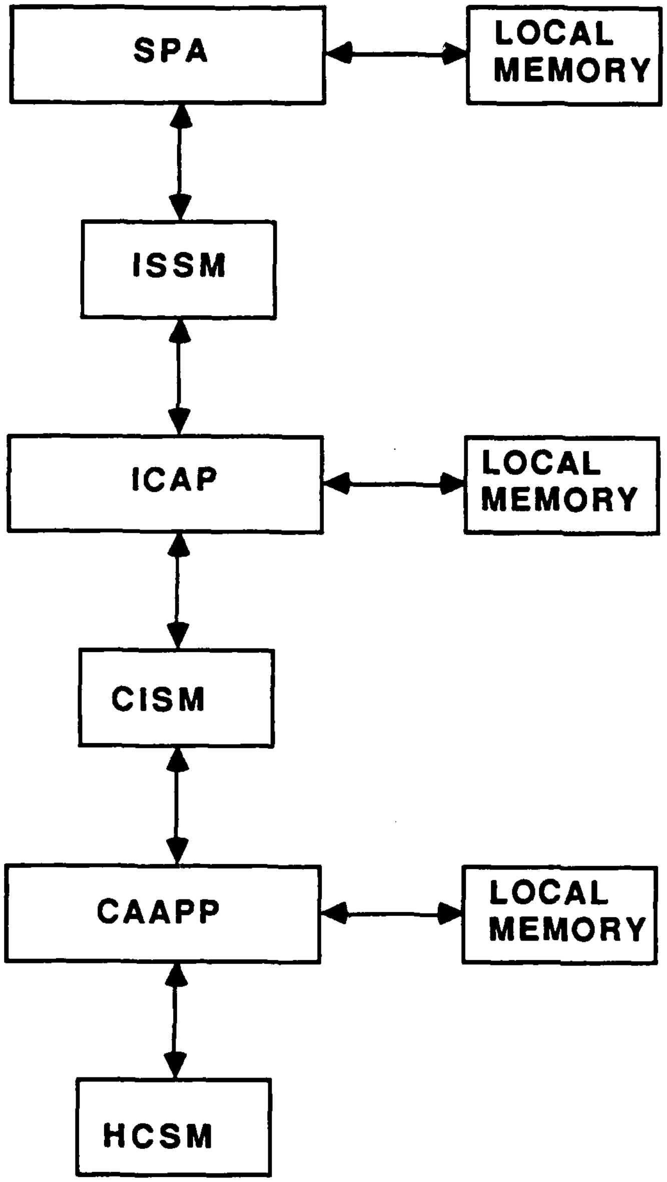
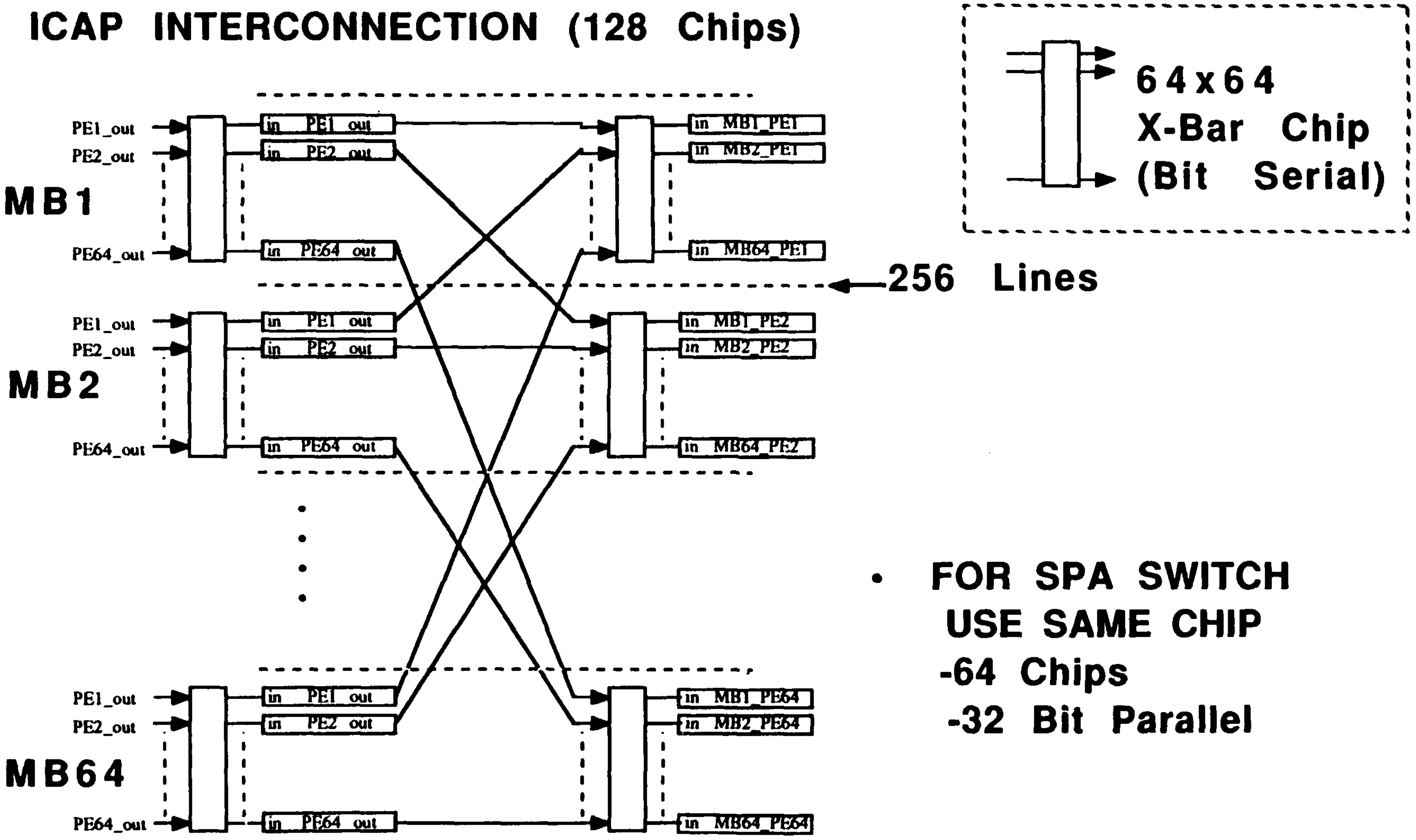


Figure 1. Memory Organization of IUA

ICAP INTERCONNECTION (128 Chips)



- FOR SPA SWITCH
  USE SAME CHIP
  -64 Chips
  -32 Bit Parallel

Figure 2. Two dimensional bit serial crossbar network for the ICAP

There is a total of eight status bits available from the daughterboards. Three done bits correspond to three ICAP interrupt levels. The hold acknowledge is a bit signifying that the ICAP is tri-stating all its buses to allow the ACU to load the routines. There is an error flag to indicate that there is memory error during loading. The backing store status indicates transfer is complete. There is an ICAP general purpose flag, and a some/none flag corresponding to the logical sum of all the CAAPP PE X-registers. Finally, there is a global response count feedback to the ACU.

3.0 Enhanced IUA/CAAPP architecture

3.1 Introduction

There are four major areas of CAAPP architecture enhancement carried out under this contract: memory, I/O, interconnection network, and processor. For each of these areas we will describe the issues, the strategies that have been implemented to address them cost-effectively, and the resulting performance benefits.

3.2 Memory

One of the major reasons for developing a parallel architecture is to avoid "Von-Neumann bottlenecks." At 10-MHz clock speeds, each CAAPP PE could potentially require a large amount of memory. With the original 128 bits of memory, each PE would need extra space to save intermediate results, or need new data which might not be available locally. Consequently, data must be supplied from a system bus and suffer bandwidth bottlenecks. Thus, a PE memory size of 320 bits was implemented on-chip. The key to achieving this was an area-efficient 3-transistor dynamic RAM memory cell and a space efficient layout. A one-transistor cell design would have provided higher density, but the additional design cost and lack of detailed knowledge of available process technology eliminated this possibility. A static cell, on the other hand, would have been far too expensive in terms of space, with the six transistor technology available.

An alternative to increasing the on-chip memory is to use an external backing store. If implemented directly this would entail an additional 64 pins to the CAAPP chip since there are 64 PEs on a chip. Because efficient use of chip pins is so important, we decided to use time multiplexing. With a 4-way time-multiplexed buffer, we could reduce the number of extra pins to 16.

Until now, however, only static random access memory (SRAM) has had the required speed to keep up with the time multiplexing scheme. The drawbacks of using high speed SRAM are that they are expensive, use board area inefficiently, and have low density compared with the dynamic random access memory (DRAM). With the recent availability of low-cost high density dual port video-RAMs (VRAM), we could achieve a design allowing each CAAPP PE to have access to a 32K-bit RAM backing store by using 256-K-bit VRAMs. In addition, CAAPP memory could later be increased to 128 K bits per PE by using the 1-Mbit VRAM chips now available. The serial port of the video RAM has an internal 256-bit shift register and can shift out data to the CAAPP chip every 25-40 ns. The serial port does not need an address for each access, unlike a dual port SRAM.

The availability of the dual port capability makes it possible for the ICAP to share the CAAPP PE backing store via the random access port. By implementing strategies such as this, along with page-mode efficiency and providing zero-wait-state VRAM access to the ICAP, we can very cost-effectively closely couple two heterogeneous parallel architectures at a very fine grain with high speed. This is a very important result, since the ICAP is designed to manipulate the tokens and object frames at the intermediate level in a way which requires its approximate registration with the original image events in the CAAPP (there is a 64 to 1 processor ratio between spatially co-located CAAPP and ICAP elements).

Another important consideration was that of the footprint on the PC daughterboards. Traditional DIP packages use a considerable amount of space. However, with the VRAM approach the ZIP ("zig-zag" in line package) package, 100 mils in width, was available. This offered 2.5 times more board density than the standard DIP.

The 4-way time-multiplexed buffer is connected to allow access to 256 bits of RAM from each CAAPP PE. The dual port allows 256 bits of the 320 bits of each CAAPP PE to be shared with the ICAP, a big improvement over the original 8 bits to be shared out of 128 bits. The 320 bits of RAM is also divided into multiple banks to provide a double-buffered swapping mechanism such that the ICAP can access the CAAPP memory while CAAPP PEs are active. The transfer between backing store and CAAPP PE memory is through block mode. A transfer controller keeps track of whether the specified number of bytes from CAAPP PE memory of a given address has been transferred to the backing store at a given memory location. It employs a cycle stealing technique in addition to double buffering. Since the internal transfer bus width is 64 bits, compared with the external 16 bit bus, cycle stealing means a CAAPP PE can have access to all three banks of memory while the backing store transfer operation is in progress.

The strategy of sharing the CAAPP PE memory with the ICAP, though, has one drawback. While a CAAPP PE is a bit-serial machine with a 1-bit data path to access memory, the ICAP, by contrast, accesses data in a 16-bit word format. In addition, most image pixels have an 8-bit byte format. One way to address the problem is to convert the first two 128-bit banks of CAAPP memory to have an 8-bit data path, and let the 4-way time-multiplexed buffer pack and unpack 8-bit pixel bytes to 16-bit words that are compatible with the ICAP. This approach has many architectural benefits and one implementation drawback. First, it offers the CAAPP PE an 8-times speed increase over load/store operations, and improves the floating point performance because alignment shifts can be made in 8-bit increments instead of 1-bit at a time. Second, it supports the I/O corner turning circuitry, which requires one bit from each of eight boundary CAAPP PEs to turn one byte in a given PE so that an external image can be loaded or stored via mesh network with an effective bandwidth of 4K x 10 Mbyte/sec (i.e. 40 billion bytes/sec). The details will be discussed later in the I/O section. Third, it supports the 8-bit control registers used by an augmented mesh called the Gated Connection Network (GCN) (or Some/None "Coterie" mesh) discussed later. The use of these control registers in "long-distance" communication reduces the diameter of the GCN to "1" by properly short-circuiting the involved CAAPP PEs. The 8-bit data path allows the GCN to be set-up with one instruction instead of eight. This speed-up is significant when a binary-tree graph is embedded on the GCN to process some function such as local response count summation in logarithmic steps. Each level of the tree uses new settings in the control registers. Finally, it provides a future upgrade path for adding 8-bit full adders for each CAAPP PE on the existing 8-bit data path, which could improve the multiplication and floating point operation speed by eight times.

The only drawback of the above approach is the VLSI implementation. The floor plan for the chip is as shown in Figure 3(a). Since all CAAPP PEs are designed to share the same control lines, obtained from instruction decoder shown on the top right, the PEs are made long and thin, stacked one on top of the other as shown on the upper right portion of the figure. All control lines run vertically through the stack of processors. Because the layout size of the CAAPP PE's full mesh communication network in the horizontal direction is fixed, as shown next to the PE's ALU, to minimize the network real estate the vertical pitch of CAAPP PE is designed as small as possible. For 2-micron CMOS, the pitch is only 79μ, as indicated by the arrow. The small vertical pitch is also dictated by the speed requirement to minimize control signal skew to the bottom PEs, and by the constraints associated with the Coterie network on the left side of the floor plan. With the existing 3 control/feedback signals over the memory cell, it is extremely difficult to run 8 additional horizontal data paths at this 79μ pitch in addition to the necessary contacts/vias, assuming a design

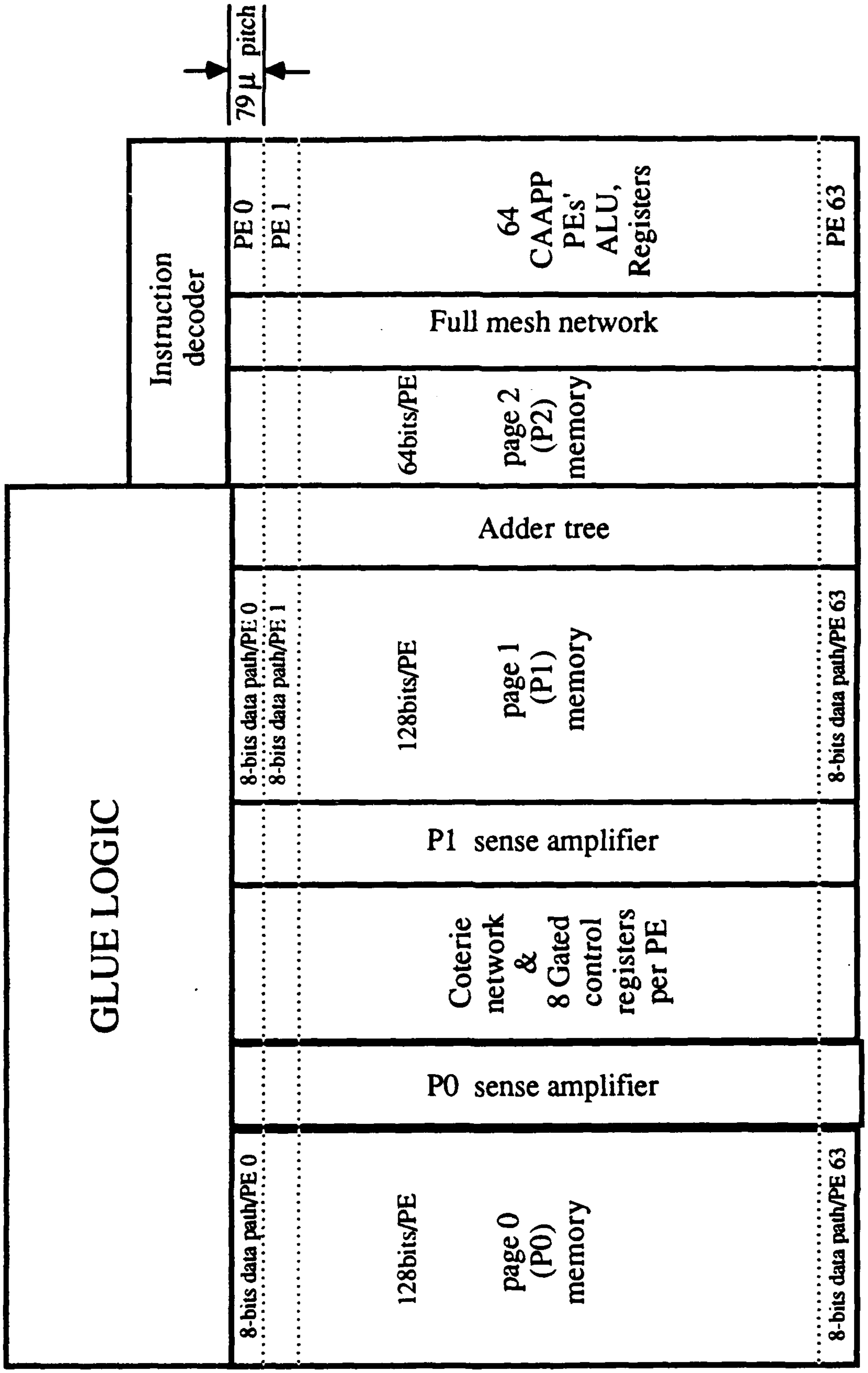


Figure 3(a). CAAPP Chip Floor Plan.

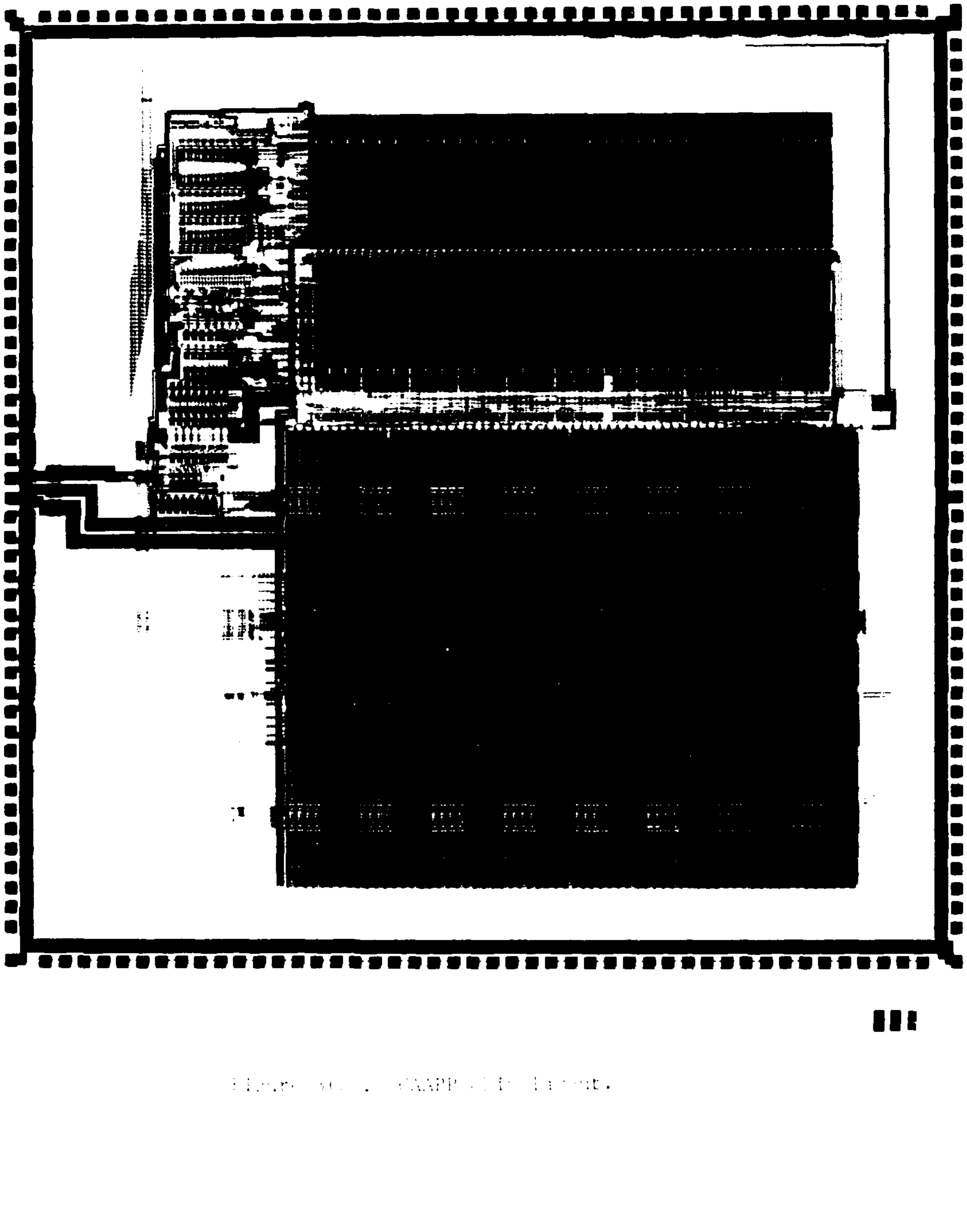


Figure [illegible]. CAAPP chip layout.

rule of 7μ per "metal-2" line. Even worse, there are six masters trying to access it: the CAAPP PE ALU, the backing store 4-way time-multiplexed buffer, the double buffering of two banks, the corner turning circuitry, the Coterie control registers, and possible future provision for an 8-bit full adder. How to arbitrate the data path access was quite challenging. Based on the finished layout, more than 70% of the chip is consumed by memory, which occupies most of the left hand side of the floor plan as indicated by page 0, 1, and 2 memory.

3.2 Input/Output

In the past, designers of systems with parallel architectures often mistakenly ignore the bottlenecks caused by loading the input data into the parallel machine. Recently, system designers are using a variety of strategies to address the problem of implementing a cost-effective I/O channel for a mesh-connected parallel machine. Most are feeding data from the edge of the mesh communication network and then shifting the data to the rest of the processors. This strategy has two drawbacks. When the parallel machine is loading or storing data, it can't execute programs. In addition, the loading time is proportional to the size of the network and the shift speed of the processing elements.

Our solution to this problem is to use dual port VRAM chips distributed evenly among the processors of the parallel machine. This simultaneously provides compatibility with other VRAM memory in the IUA system, and also a high density, low footprint implementation. One block of dual port video RAM is connected directly to each custom CAAPP chip via a shared full mesh connection such that the next image can be loaded from camera or disk while the current image is being processed by the CAAPP array. The serial port of the video RAM block (8 bits wide) is connected to the south side of the mesh network of each CAAPP chip. The random access port is connected to a DMA controller which is in turn connected to I/O or disk channel and host system bus. These blocks of video RAM are called the Host-CAAPP shared memory (HCSM), and are viewed by the host as ordinary memory which can be loaded from disk like any other system memory. However, these memories ultimately will have direct access to the resources of 512 x 512 CAAPP PEs. The effective bus bandwidth between these memories and the CAAPP PEs is approximately 4K x 10 MByte/sec (i.e. 40 billion bytes/sec) since there are 4K-Byte blocks of video RAM, one for each CAAPP chip, and clock speed is 10-MHz.

Normally, the serial ports of the video RAM are disabled so that off-chip neighboring CAAPP PEs can communicate with each other. When the next frame of the image is finished loading at the random access port of the HCSM, or an "intelligent memory operation" is requested by the host, the data will be transferred to the serial port and shifted out. In the mean time, the output of the serial port will be enabled while the output driver of the north side of the mesh network of each CAAPP chip will be disabled. Using an existing CAAPP PE instruction for loading data from the south neighbor, the eight bits of data from serial port will be loaded into the eight south boundary CAAPP PEs, one bit per PE (each CAAPP chip contains an 8 x 8 grid of PEs). Off-chip neighboring CAAPP PEs can't communicate with each other during this time, because the output drivers of the north boundary CAAPP PEs are disabled intentionally to relinquish control of the off-chip mesh network to the VRAM. After eight CAAPP PE loading instructions, eight bytes (64 bits) of data are evenly distributed over all 8x8 CAAPP PEs, one bit per PE. Using the corner turning circuitry, one bit from each of eight CAAPP PEs in a row turns to one byte in a given PE of that row; eight rows will perform this operation simultaneously with one corner turning instruction. The CAAPP routines to perform this I/O operation are shown in Appendix C. By implementing strategies such as this, IUA can now support real-time vision without adding any extra CAAPP chip pins except for two control signals.

3.3 Interconnection Network:

With availability of area-efficient 180-pin VLSI pin-grid-array (PGA) packages and reasonable foundry yields for large chip sizes (300 x 300 mil^2), the communication between CAAPP chips was enhanced from a 4-way time-multiplexed network to a non-multiplexed full mesh to support GCN or Coterie network. This second network allows construction of wired-OR buses between contiguous processor groups. An example of the benefit that is obtained with the "augmented" mesh is a speed increase over a standard mesh by a factor-of-1000 in performing operations such as connected component labelling, and extraction of region features such as orientation, size, width, length, and aspect ratio of the minimum bounding rectangle of a given region. The details can be found in a book chapter [1] which is included in Appendix A.

The Coterie can connect a large number of processors with the compactness of a mesh architecture while achieving the same or better communication efficiency as the hypercube architecture for a wide spectrum of vision algorithms. For example, some of the consensus function such as MINIMUM, MAXIMUM, AND (i.e. ALL) and OR (i.e. ANY) of the entire image, take logarithmic steps to process in a binary-tree graph intrinsic to the hypercube architecture. It takes a unit step in our Coterie network [1].

As shown in Figure 4, each CAAPP PE has 8 control gates <N, E, W, S, NW, NE, H, V> corresponding to the 8 directions of north, east, west, south, northwest, northeast, horizontal, and

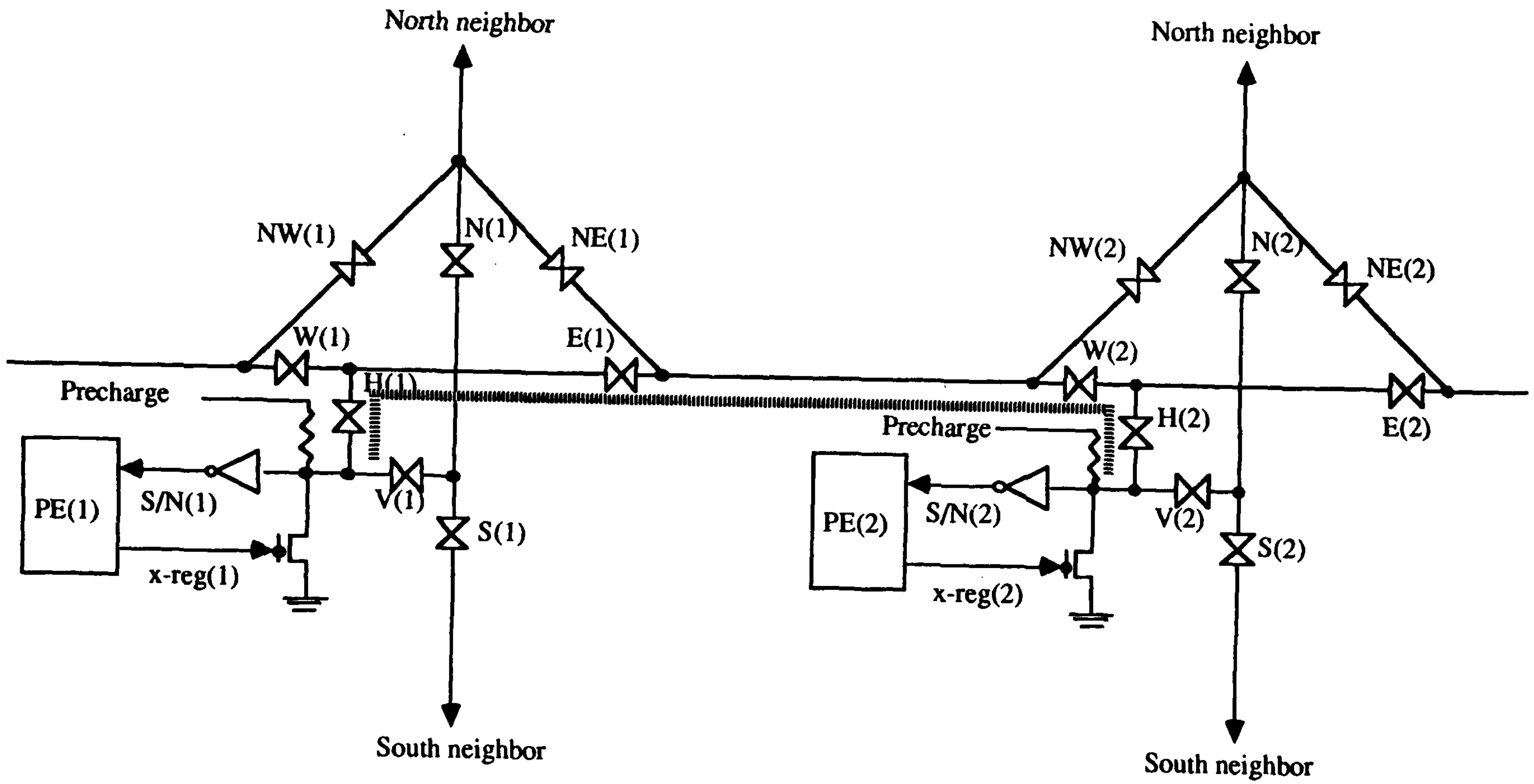


Figure 4. Gated connections for PE(1) and PE(2)

vertical, respectively. They are driven by 8 corresponding GCN control registers (GCR), which can be loaded from any byte/nibble of the first two pages of PE RAM (256 bits) with one instruction. This same instruction can also be used to shift 8 or 4 bits of PE RAM to facilitate floating point operations. GCR(N)=1 means north control gate is closed; GCR(N)=0 implies it is open. The voltage level at the GC network is determined by the corresponding output of PE x register (x-reg), and can be stored via an inverter in individual PE memory locations; the signal after the inverter for PE(i) is called the some/none (S/N(i)), which means if any of the x-regs of the gate connected PEs is equal to one, then the some/none signal will be equal to some (i.e. S/N(i)= 1). For example, as shown in Figure 4, if gates H(1), E(1) of PE(1) and H(2), W(2) of PE(2) are closed, then S/N(1) and S/N(2) are equal to the wired-or of x-regs of PE(1) and PE(2), which are connected as the result of the closed gates. The some/none signal S/N(i) may be used to control (e.g. disable) the activity of PE(i), if so desired. One of the nice features of a bit serial processor is that for any decision based on local summary results there are only two-way branches. For example, if some/none equals "some" then do something; otherwise it can shut itself off. Therefore, different groups of gate-connected PEs can operate differently, because each group may have different local results even if they receive the same instructions. This effectively turns an SIMD machine into one with MSIMD-like capabilities. In other words, it meets the requirement that there be local feedback of summary information corresponding to data dependent concurrent processing.

For extreme cases, there may be only one transistor activated by one x-reg to sink all the charge in the electrically connected network. Two design techniques have been introduced to improve the rise and fall time of the GCN. First, it only allows a zero level voltage to pass between chips; if the GCN has a high level voltage across the chip boundary, then the two chips are electrically disconnected. Therefore, any internal x-reg needs to sink only the GCN, consisting of no more than 64 PEs. If the boundary buffer sees a zero, then it will sink and pass the zero level voltage to the neighboring chip. This is efficiently implemented because of the need for having external buffers to drive the chip at the motherboard interface. Assuming 10-15 nsec delay to cross chip boundaries, for 64 chips in a row the maximum propagation delay in a full-up (512 x 512) system is approximately 1 to 2 μsec. A second technique decouples the horizontal and vertical GCN such that any internal x-reg needs to sink only the nodes corresponding to 8 PEs. Although the details are somewhat complex, the basic idea is that any zero level voltage on a vertical set of connected nodes will automatically pull down the horizontal GCN via an extra buffer and vice versa. The GCN shown in Figure 4 is just a logical view of the net; the actual circuitry and layout are much more complicated, due to numerous constraints such as the 79 μ vertical pitch distance allocated to each CAAPP PE.

3.4 Processor

The activity of the CAAPP processor is now not only controlled by the Activity bit (A-register) but also by the state at the GCN such that simultaneous data-dependent processing of multiple PE groups can be achieved. Each group of PEs can be operated independently from other groups at the same time. The speed-up over standard SIMD architecture is proportional to the number of PE groups. The PE group may consist of all PEs in a connected region, or PEs in a given column or row, or connected nodes in a network graph. For example if there are 100 regions in an image, the speed-up will be 100.

4.0 Daughterboard Design

The daughterboard is intended to be a glue-free cost-effective building block for the full IUA system. It includes five basic components: a CAAPP chip containing PEs and all the necessary daughterboard glue; a backing store VRAM block for CAAPP and ICAP Shared Memory (CISM); a TI TMS320C25 5-MIP DSP chip for the ICAP; a SRAM block for ICAP's program and data memory; and another VRAM block for ICAP and SPA Shared Memory (ISSM). Since the full IUA system requires 4K of these daughterboards, they must be built very efficiently to reduce the manufacturing costs.

Each daughterboard is 3" x 4" in size and contains more than 700 plated-through holes for IC pins due to PGA and high pin-count chips. Figure 5 shows the physical dimensions of each component and its floor plan. The VRAM with 100-mil-wide ZIP packages stands out as the most efficient board area user, because within this small area we are able to pack 640 Kbytes of memory. Due to the large number and density of high pin-out chips, it is extremely difficult to route all these holes/pins within the printed circuit board (PCB) of 3" x 4" with constraints that only allow two traces to pass through the 100-mil clearance between two holes. Our solution was to send all global signals, such as the 32-pin instruction bus, the 32-pin ISSM address/data bus, and the 32 pins of mesh network over the motherboard to other boards. Only local signals such as those between ICAP and its memory will be routed and "descrambled" within the daughterboard PCB. The routing and descrambling of the global signals are done on the address/data driver board and instruction driver board located at the motherboard. The mesh network is routed via motherboard. Routing difficulty is the major reason we took the strategy of daughterboard/motherboard. It is nearly impossible under current technology to route and

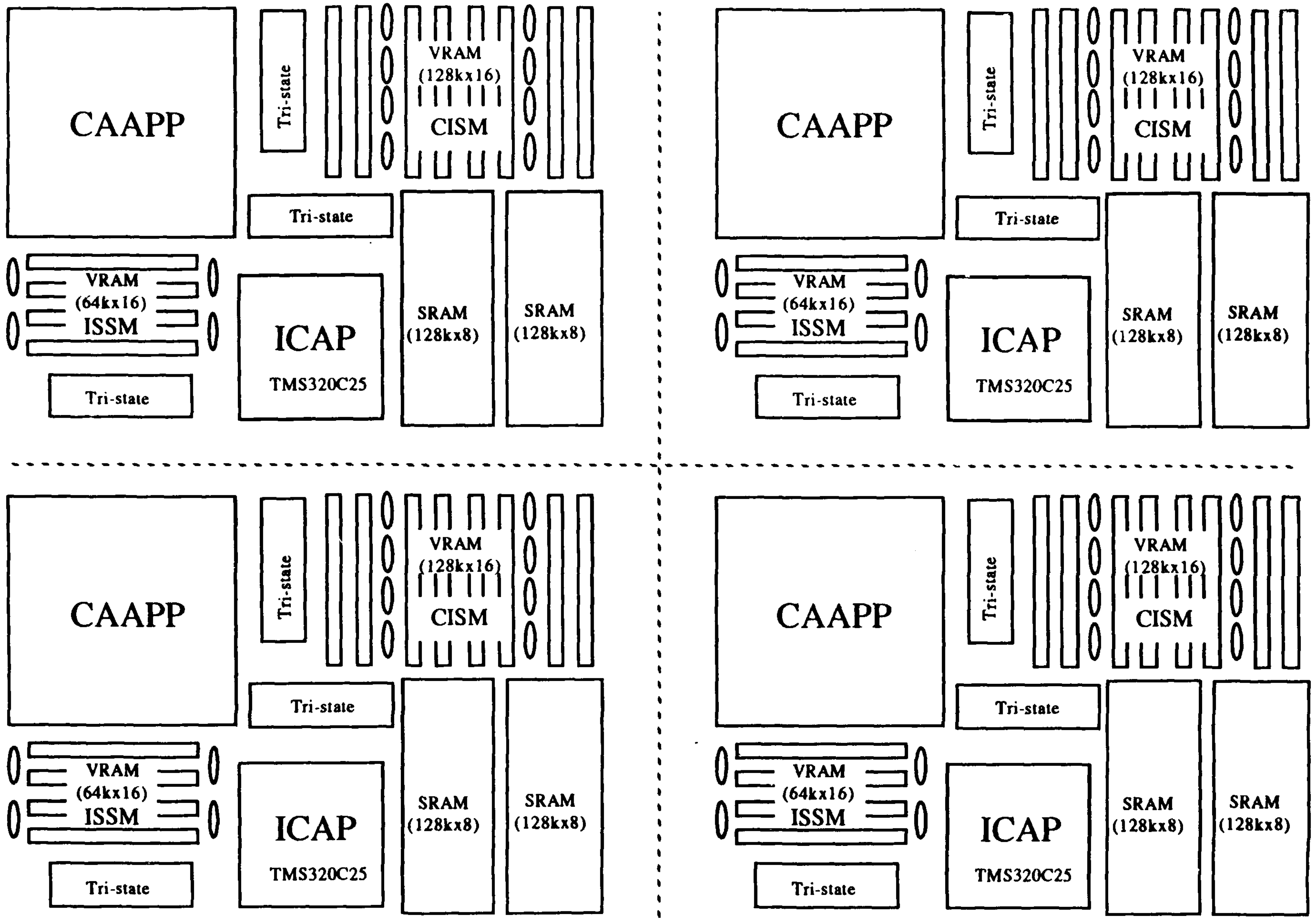


Figure 5. Daughter board floor plan

descramble all the global and local signals in a reasonably sized single motherboard without using the daughterboard approach.

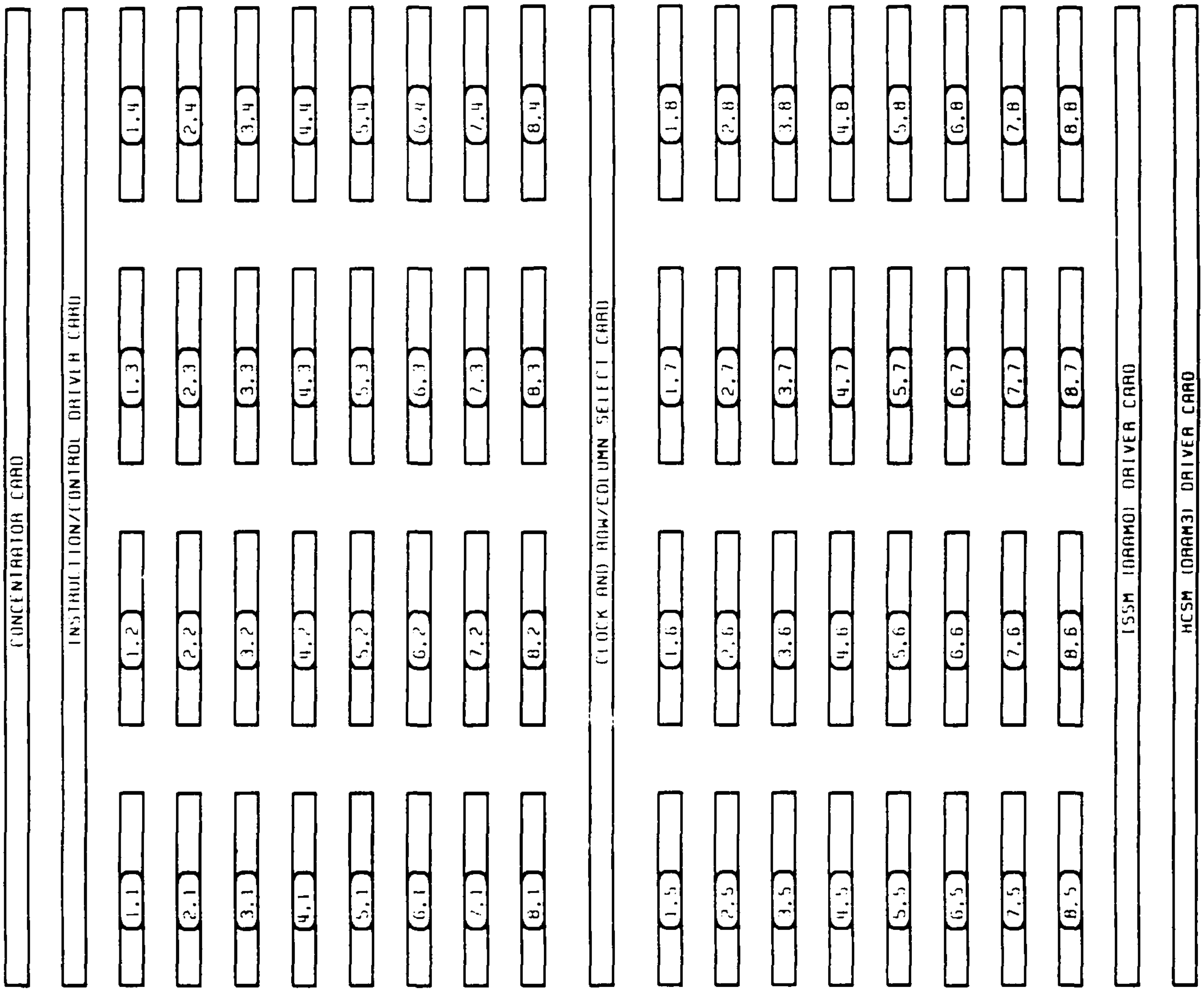
At a 5 MIP speed, each ICAP would consume lots of memory. Thus, we used the two latest 1-Mbit 32-pin DIP SRAM modules to provide 128K x 16 of program and data memory. The selection of TI TMS320C25 Digital Signal Processing (DSP) chips for the ICAP level provides us with fast general-purpose number crunchers, built-in serial communication ports for networking, multi-processor synchronization, low-power CMOS technology, large memory space, and a high level C-language compiler. It can satisfy the IUA medium-grain processor requirements with minimal development effort so that we can concentrate on optimizing custom CAAPP chip design.

In both the daughterboard and motherboard, the board areas are used efficiently and packed with dense chips. As a result, all the chips must be CMOS technology; also, forced air-cooling by fan is required. There is one 64-pin ribbon cable connector on the top edge of a daughterboard, and one 96-pin DIN connector on the bottom edge. The daughterboard will be plugged into the motherboard in parallel with the air-flow so that neither the ribbon cable on top nor the vertical daughterboard will block the air-flow.

5.0 Motherboard Design

To minimize the size of the motherboard and simplify its design, there are no IC chips on the motherboard except VRAM blocks of HCSM, which sit over the mesh net connecting two adjacent 96-pin DIN connectors 0.8" apart. As shown in Figure 6, daughterboards are arranged as a 4 x 16 matrix. Consequently, there are four boards per row with total of 16 rows. There is one ribbon cable running over the top of each of the four columns of boards, carrying the instruction bus and address/data bus for the ISSM, as shown in the middle four columns of Figure 7(a).

In addition to 64 daughterboards, there are six special "quad-boards," which are connected to each of the four columns. These are the interconnect board, concentrator board, instruction/control driver board, and other boards for clock and row/column select drivers, ISSM driver, and HCSM driver. Each quad-board is connected to the mother board through four 96-pin DIN connectors on the bottom card edge, and supports four 64-pin ribbon connectors on the top with ribbon cables running over it. The interconnect card provides a bit serial crossbar network for all 64 ICAPs, and can be easily upgraded for more advanced interconnection topology. The concentrator card, shown in Figure 7(b), gathers the some/none, the ICAP hold acknowledge, the response count,



IU SYSTEM CARD POSITIONS

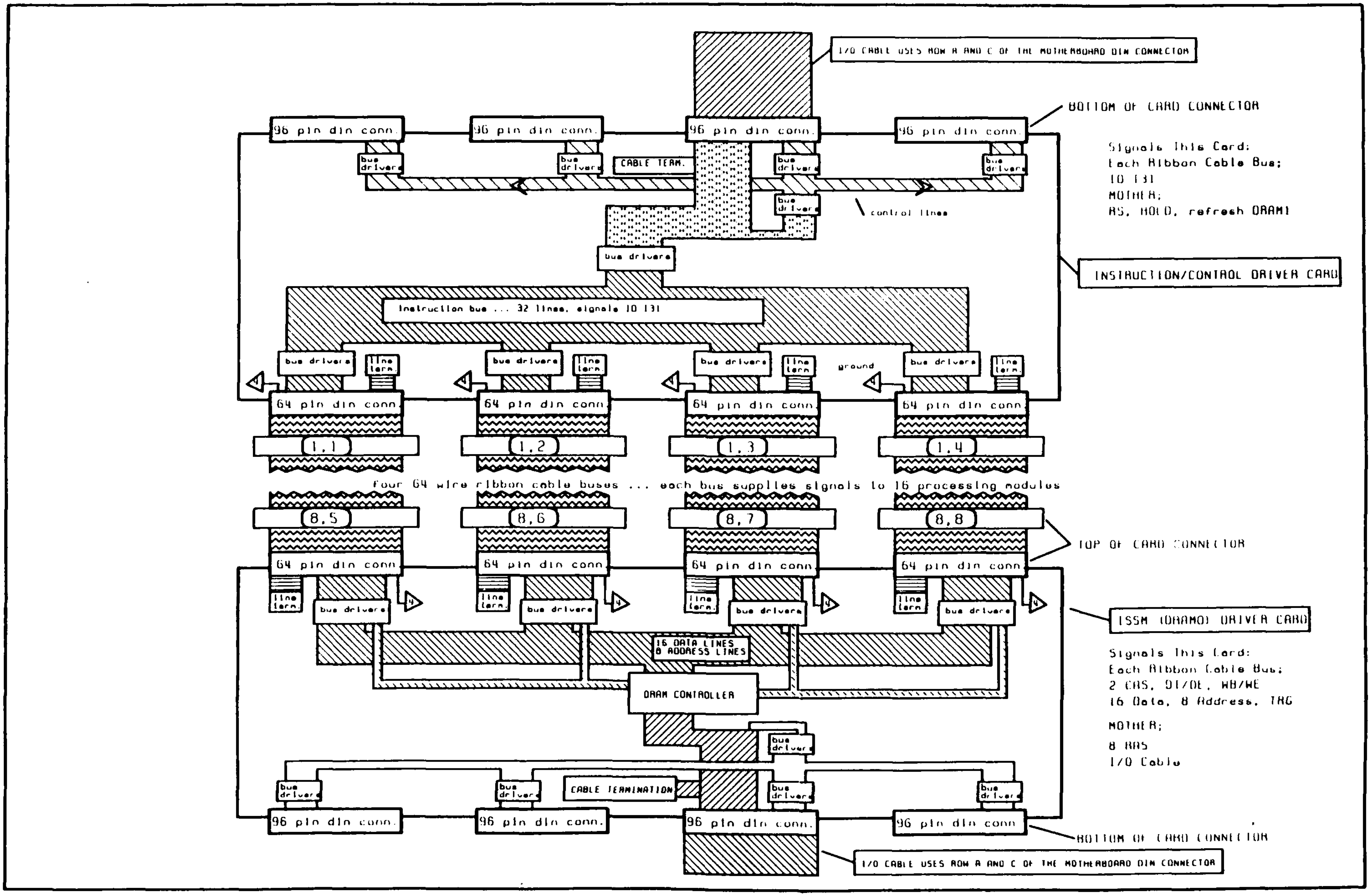
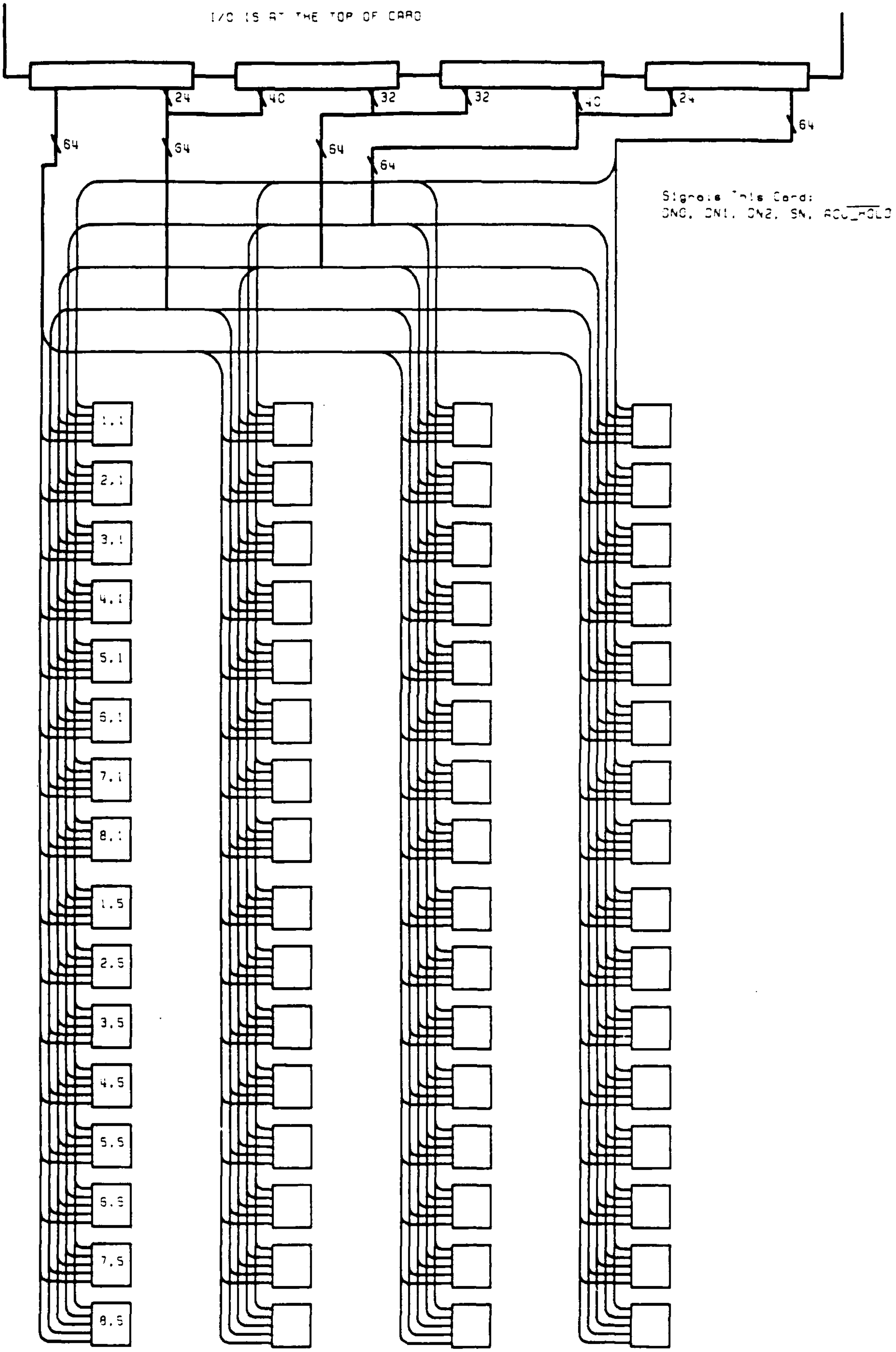


Figure 7(a). Four ribbon cables over four columns of daughterboards.



Signals This Card:
DN0, DN1, DN2, SN, ACQ_$\overline{\text{HOLD}}$

Figure 7(b). CONCENTRATION CARD

and three status bits from each of 64 daughterboards for a total of 64 x 6 = 384 signal wires, which are wire-wrapped at the four 96-pin DIN connectors on the bottom edge of the quad-board. The output to the controller is via the top ribbon connectors.

The other four quad-boards are drivers. For example, Figure 7(a) shows two driver boards with four columns of ribbon cables running in between; beneath each column of ribbon cables there are 16 daughter boards. The instruction driver card on the top of the diagram illustrates the four 96-pin DIN and four 64-pin ribbon connectors. It receives instructions from one of the 96-pin DIN connectors, buffers it, and then drives downward 4 columns of daughterboards via 64-pin ribbon connectors and cables that terminate on the ISSM driver board at the other side of the motherboard. The ISSM driver board on the bottom of the diagram receives bus signals from the VME interface in the controller chassis via one of the 96-pin DIN connectors on the bottom. The address/data buses are then fed through VRAM controller circuitry. The output bus/control signals then drive upward 4 columns of daughterboards via 64-pin ribbon connectors and cables that terminate on the instruction driver board on the other side of mother board. Obviously, 64 daughterboards are located between the instruction and ISSM driver boards. The function diagram of the ISSM and HCSM driver boards are shown in Figures 7(c) and 7(d,e), respectively. The clock and row/column select card is shown in 7(f).

With huge numbers of signals routed through the motherboard along with high clock speeds, we must use the combination of wire-wrap and multi-layer PCB techniques for routing and careful design to avoid signal crosstalk. For every daughterboard, there are card-guides screwed in the reinforced metal bars located under the motherboard to maintain mechanical integrity.

6.0 Architecture Implementation

6.1 Extended CAAPP Chip

Using VTI CAD tools, we designed the CAAPP test chips of 32 PEs, each of which had 64 bits of memory, and submitted these to MOSIS (2 micron CMOS N-well) in January. All chips returned were bad because of excessive metal-1 to metal-2 shorts. Working with MOSIS to remove the passivation of a die, we were able to probe on the chip to determine whether there were metal-1 to metal-2 shorts based on the I/V curves. In some areas of the die where the metal shorts were absent, we also cut some metal-2 lines to probe-test the function of the ALU parts of the PEs, and they operated correctly. We used a 5 micro-seconds-long pulse to test the dynamic RAM; the read and write operations functioned correctly, excepting only small portions of the 64

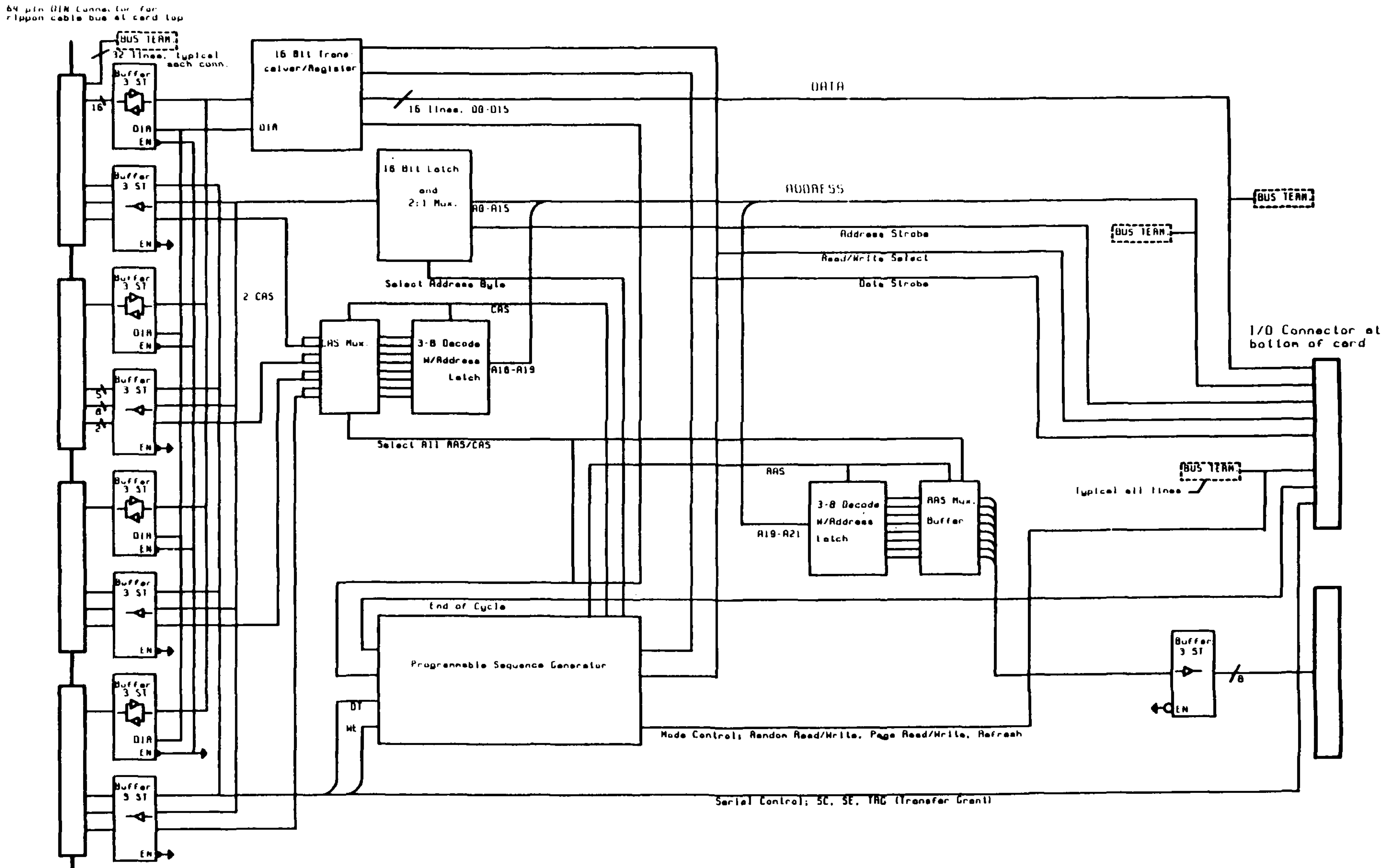

64 pin DIN Connector for
rippon cable bus at card top
BUS TERM
32 lines, typical each conn.
16
Buffer 3 ST
DIA
EN
16 Bit Transceiver/Register
DIA
DATA
16 lines, D0-D15
16 Bit Latch and 2:1 Mux.
A0-A15
ADDRESS
BUS TERM
BUS TERM
Address Strobe
Read/Write Select
Data Strobe
Select Address Byte
2 CAS
CAS
CAS Mux.
3-8 Decode W/Address Latch
A16-A19
I/O Connector at bottom of card
5
8
2
Select All RAS/CAS
RAS
BUS TERM
Typical all lines
3-8 Decode W/Address Latch
A19-A21
RAS Mux. Buffer
End of Cycle
Programmable Sequence Generator
Buffer 3 ST
8
EN
DT
Wt
Mode Control; Random Read/Write, Page Read/Write, Refresh
Serial Control; SC, SE, TRG (Transfer Grant)

Figure 7(c). ISSM (DRAMO) driver card functional diagram.

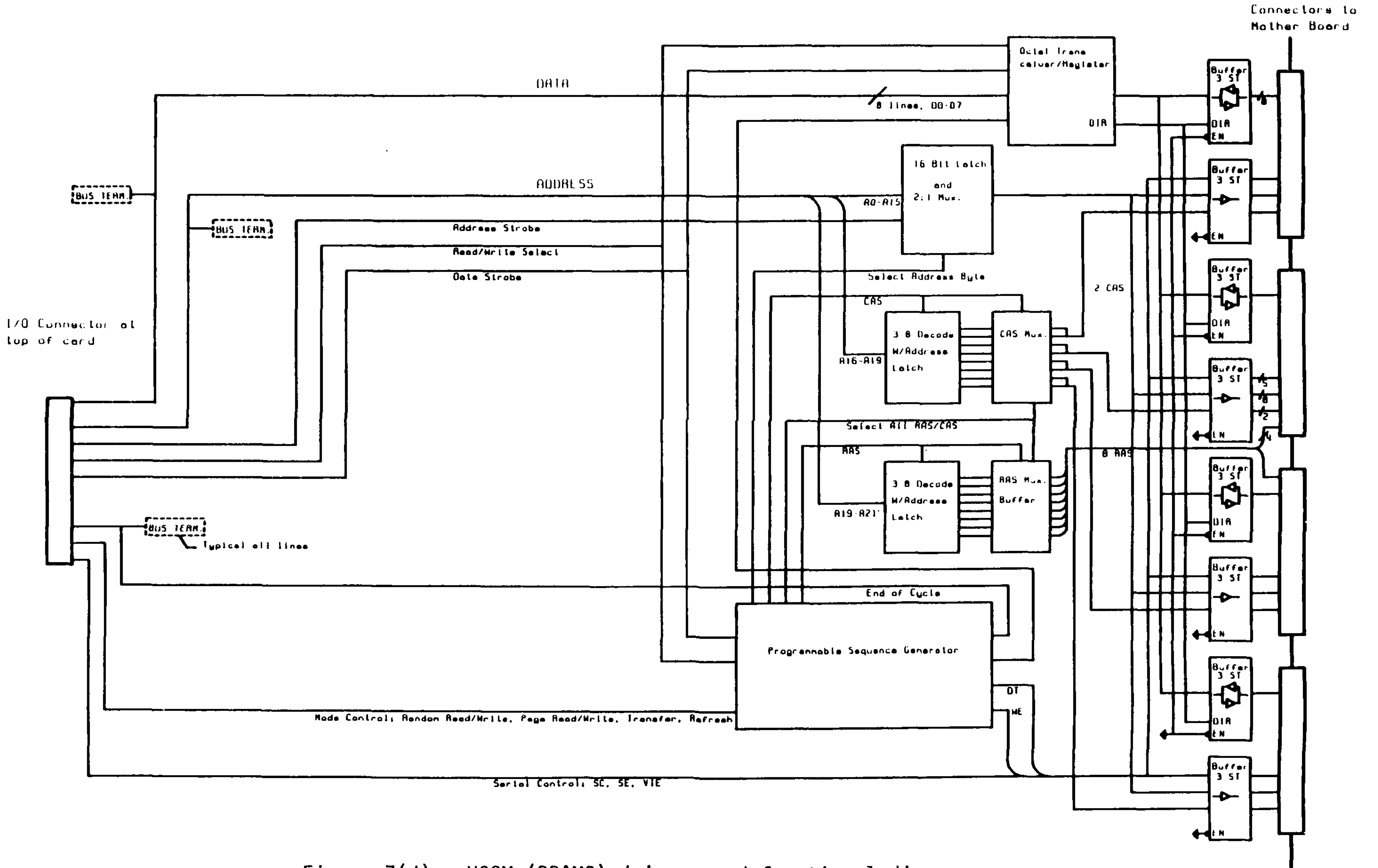


Figure 7(d). HCSM (DRAM3) driver card functional diagram.

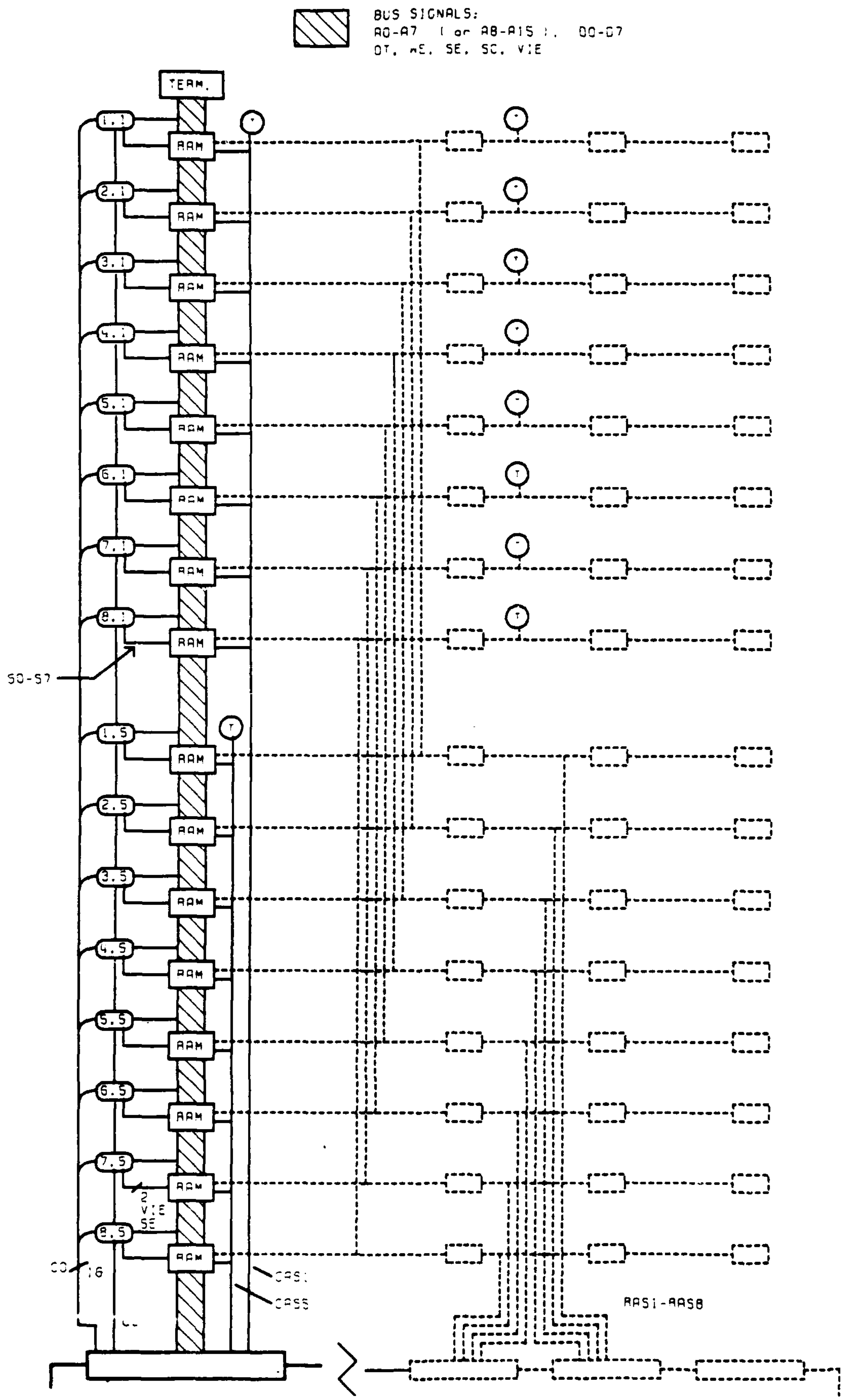


Figure 7(e). HCSM (DRAM3) driver card.

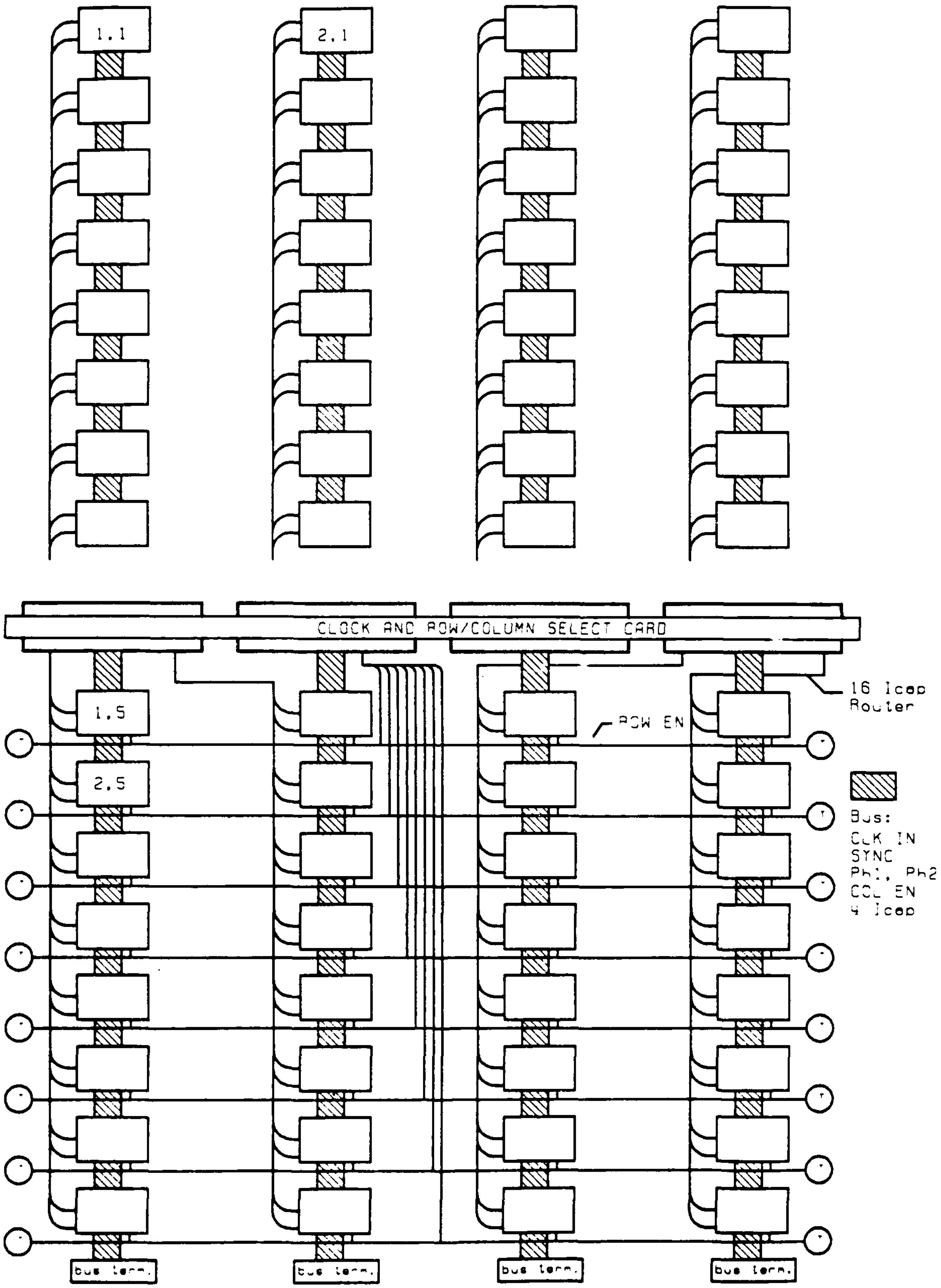


Figure 7(f). Clock and row/column select card.

memory bits, probably because the VTI design rules we were using were different from the MOSIS design rules. In summary, we still were able to test the chip function at the cell and control level.

We then completely redesigned the CAAPP chip using MOSIS design rules instead of those of VTI. The newly designed CAAPP chip with 64 PEs contained two extra pages for each PE. This added 256 bits of memory, supporting a double-buffered swapping mechanism to the 32k bit RAM backing store. This included the 8-bit data movement path, in addition to the single bit data path that runs through the ALU. These two pages are also shared with the ICAP and are the principal communication path between the CAAPP layer and ICAP layer. Additional circuitry for the Coterie network was also included. The newly designed CAAPP chip was sent to MOSIS in September, and required special handling because of large pin counts of 180 pins. The propagation delay of the Coterie network through each chip is about 15 nsec, based on Spice simulation with VTI 2 μ CMOS technology ; the delay per chip-crossing is about 15 nsec at 30 Pf loading.

The final version of the CAAPP PE chip was sent to MOSIS in December. The latest version contains additional corner turning circuitry to provide an interface to the intelligent frame buffer so that the CAAPP can send and receive directly from an external video I/O subsystem or the host VME in of support real-time vision processing. The corner turning circuitry moves one bit of Page 2 memory from 8 PEs to one byte of Page 0/1 memory of a given PE. Every subsection of the chip, such as the instruction decoder, memory, adder tree, PE registers, ALU and interconnection network have been separately extracted from the layout design and verified by the VTI simulation before the submission to MOSIS. The total transistor counts is 107,986.

6.2 Daughterboard

The design and schematics have been completed and a block diagram is shown in Figure 8(a). Figure 8(b), (c) illustrate the function diagrams for data and address bus, respectively. The first version schematics is shown in Appendix B. The breadboard, shown in Figure 9(a,b), consists of one CAAPP custom chip, backing store video RAM block, one TI TMS320C25 DSP chip and SRAM block for its program and data memory; it was completed for easy IMS testing. A second breadboard is now under construction and includes VME interface and control registers, VRAM controller and refresh circuits, two CAAPP custom chips, one video RAM block with one side of the dual port connected to the CAAPP chips via sharing the mesh network, the other side connected to the VME interface to the Sun workstation host so that a small image can be loaded

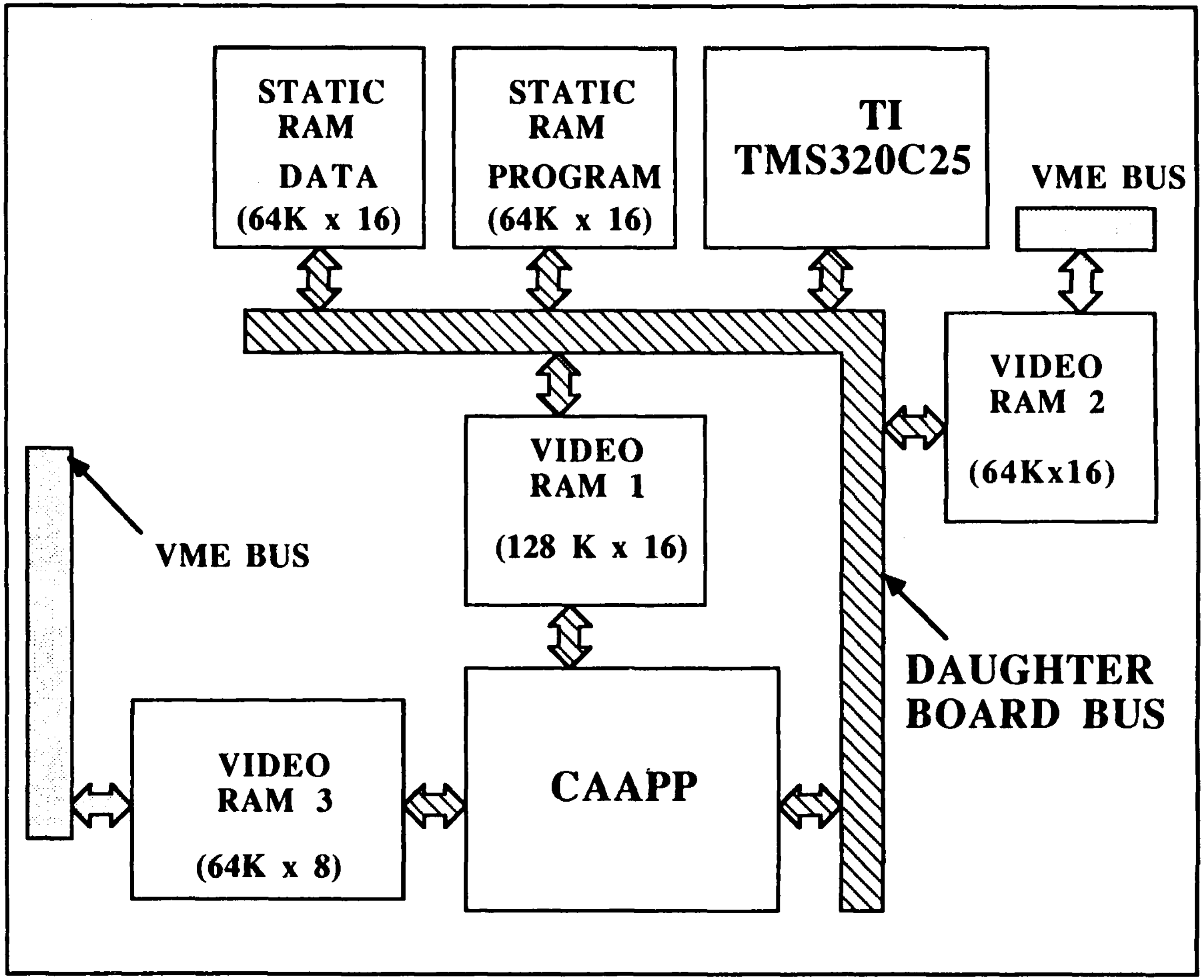


Figure 8(a). High level organization of daughterboard

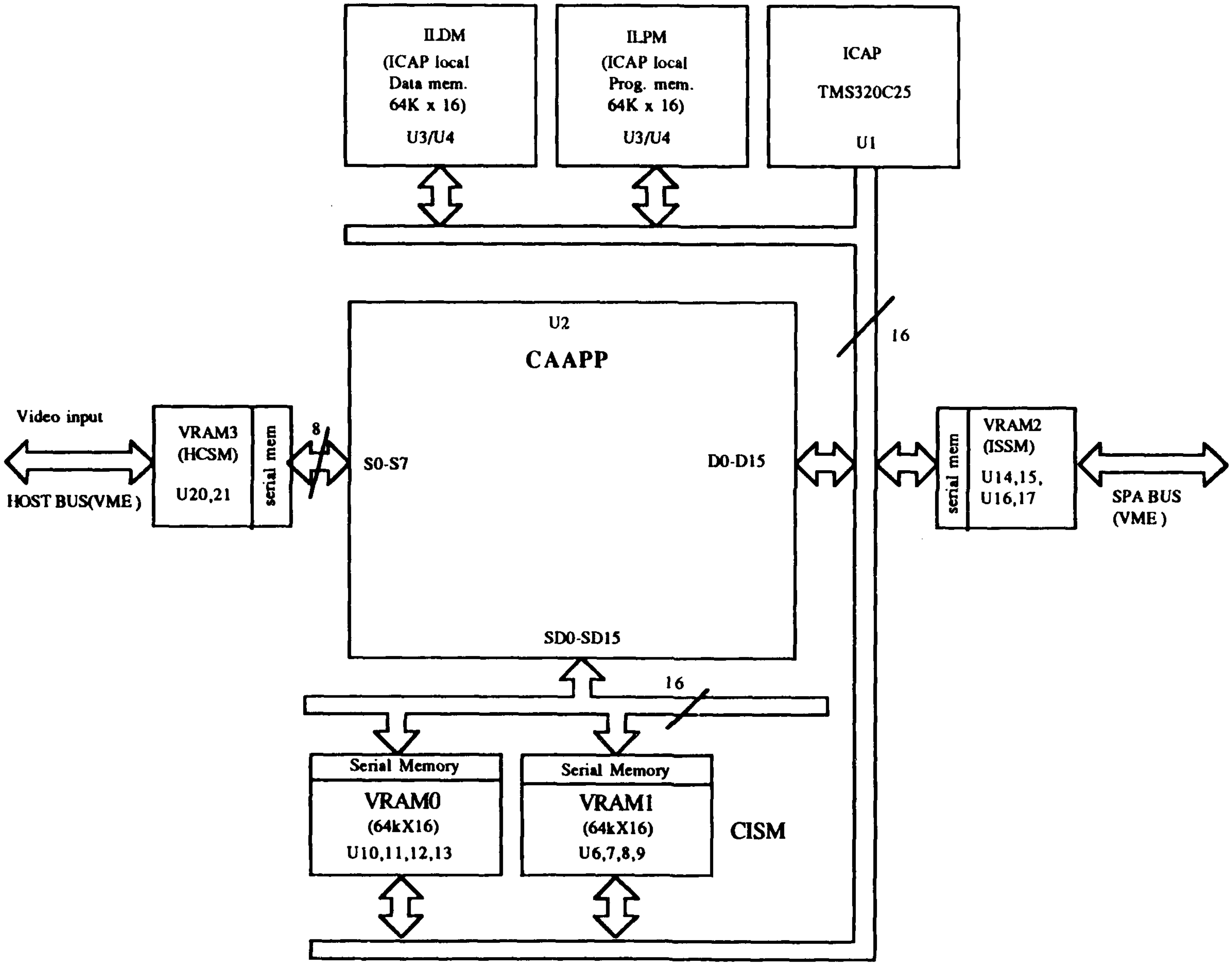


Figure 8(b). DAUGHTERBOARD DATA BUS

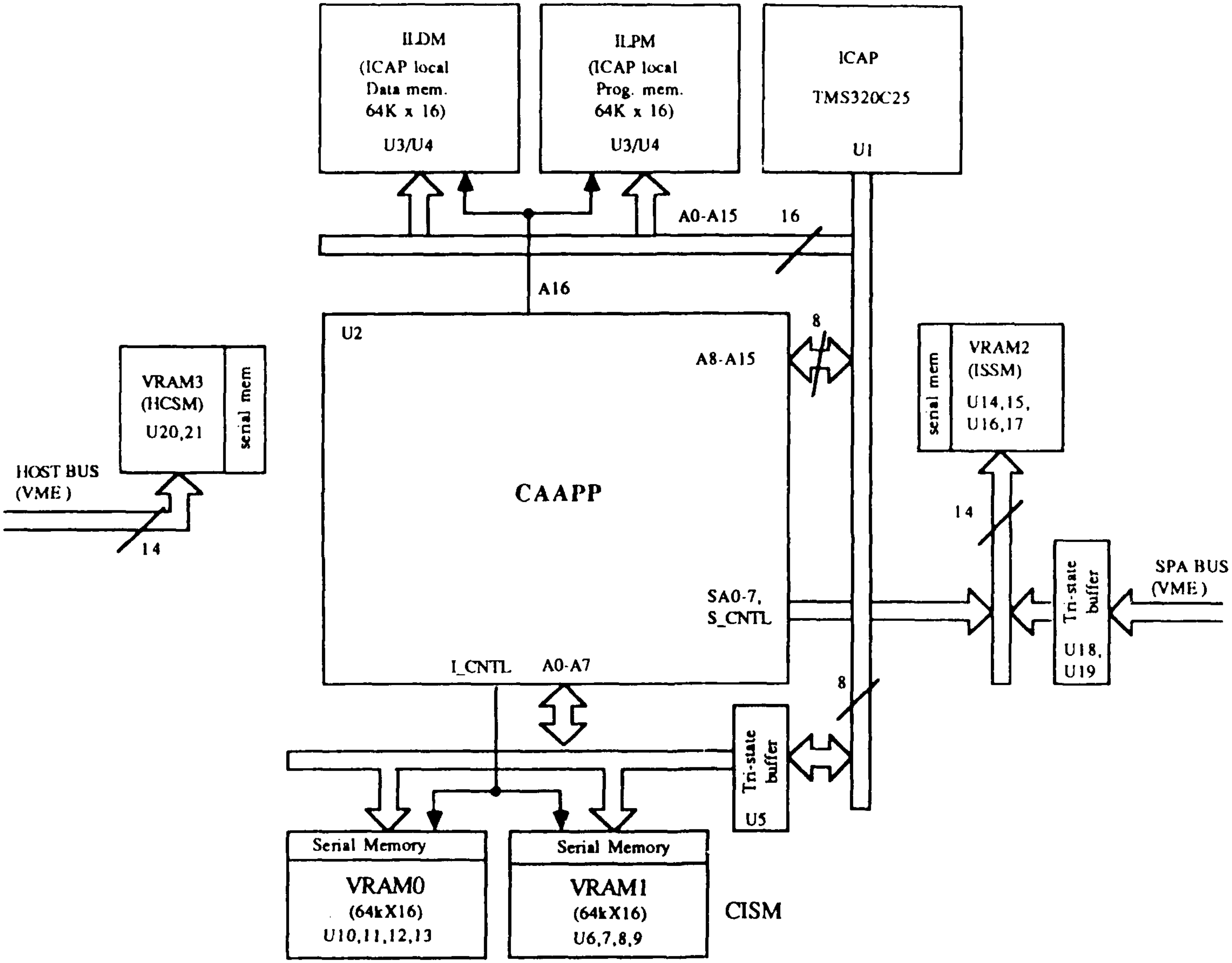


Figure 8(c). DAUGHTERBOARD ADDRESS BUS

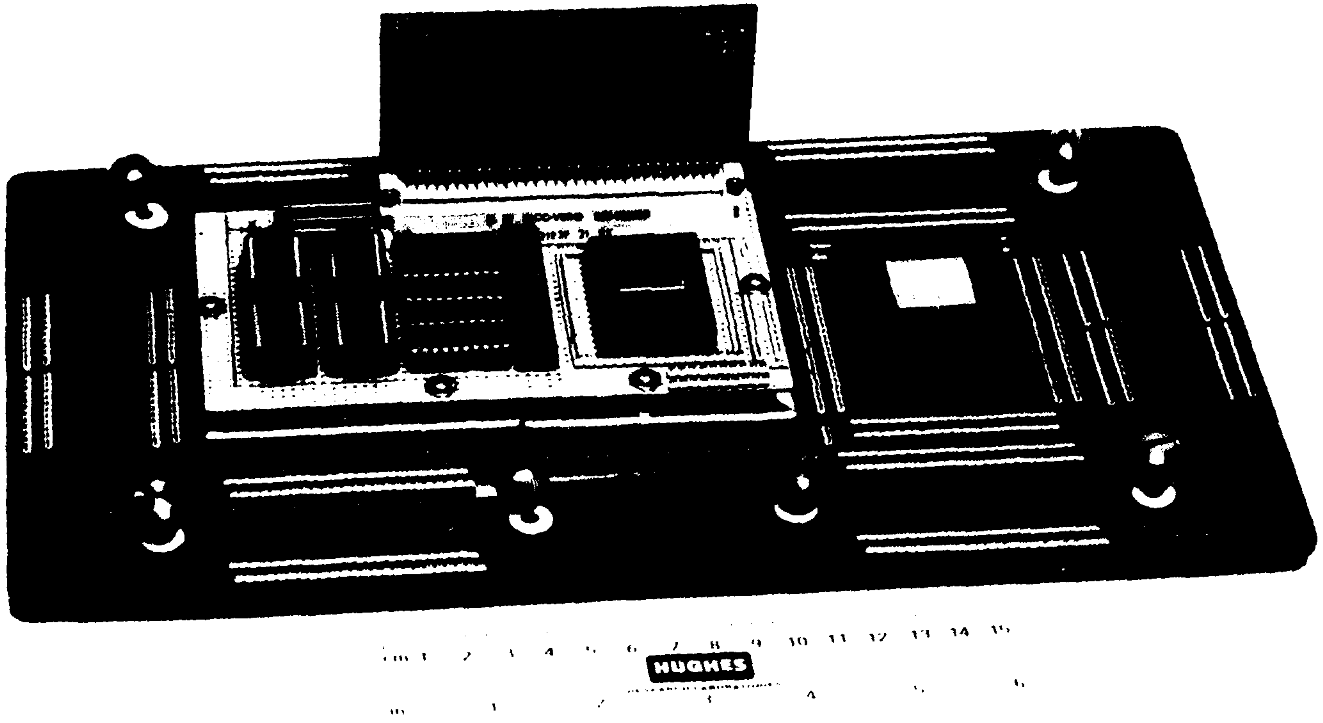
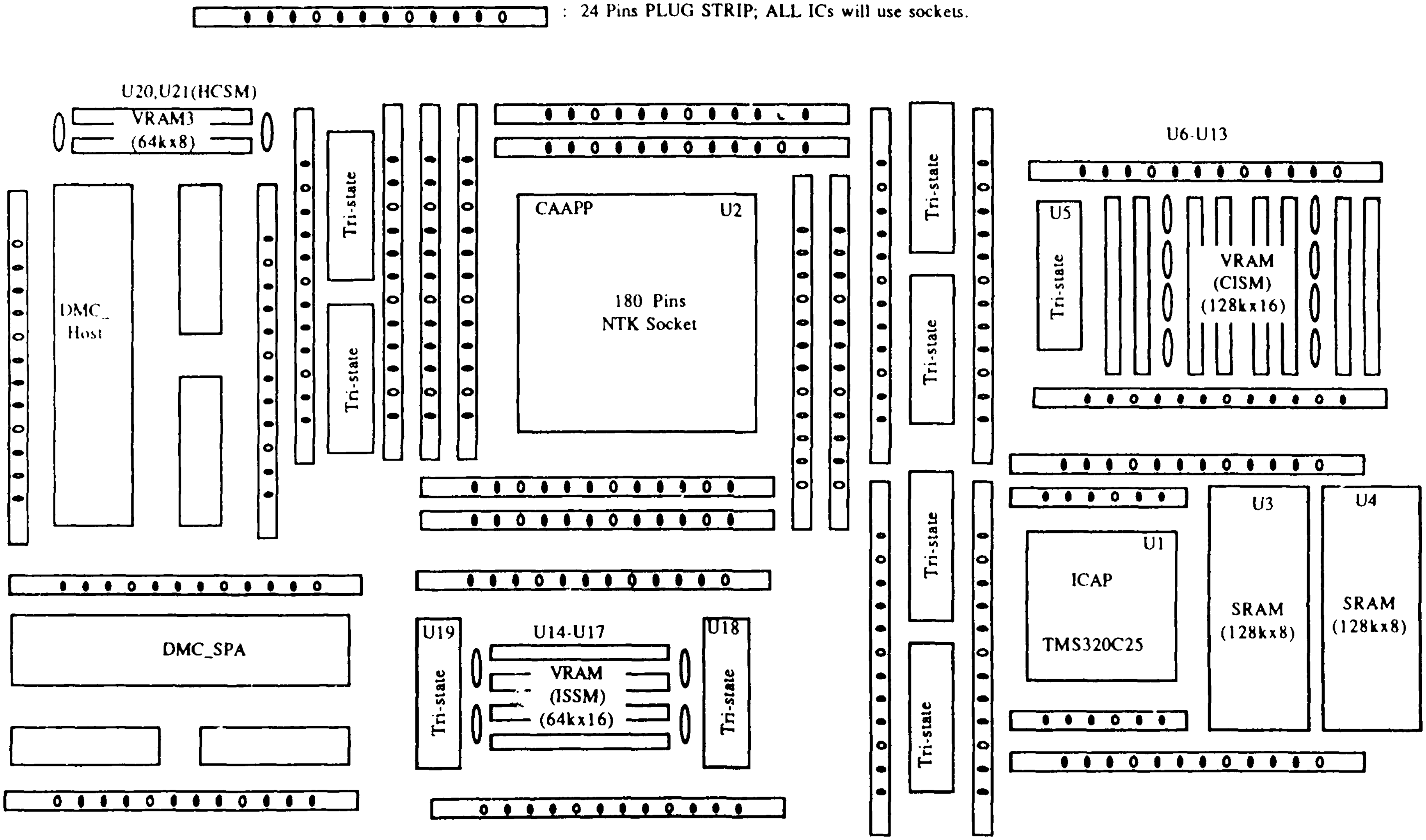


Figure 9(a). Daughterboard breadboard.



Note: · mall gauge wires
- Different color for different groups of signals
- ZIP socket need adjustment

Figure 9(b). Daughter board breadboard

from disk into the CAAPP to perform a proof-of-concept demonstration. All CAAPP instruction routines will be stored in the IMS tester. The interface between tester and Sun host is via control registers in VME interface.

The IMS test system has a large number of pattern generator channels and acquisition channels. Therefore, it is capable of issuing sequences of instructions at full rate. In addition, it has the capability to branch on detection of appropriate signals; thus, we can use it to emulate the controller. The 2-chip breadboard will allow us to demonstrate that the basic architecture is sound before we start the 64 chip production run. Since the CAAPP array architecture is based on a local mesh connection, if two neighboring chips can talk with each other, then the whole array can talk together. The only difference between the 2-chip breadboard and the 64-chip prototype is the array size and propagation delay. Moreover, the 2-chip version has a higher level of reliability than the 64-chip version connected via daughterboard/motherboard configuration. Thus, it will be easier to spot any architecture flaws. Consequently, the 64-chip prototype should be a more robust implementation.

The breadboarding benchmark demonstrations will be taken from various possibilities: segmentation, component labelling, extraction of region features such as orientation, size, width, length, aspect ratio of the minimum bounding rectangle of a given region, finding the minimum cost path, and computing minimum spanning tree. All these benchmarks are computational intensive and non-trivial and can verify that IUA is a superior machine for IU.

6.3 Motherboard

The design and schematics have been completed. In addition the structure has fleshed out in terms of physical pieces such as the chassis, fan, enclosure, connector and cables as shown in the prototype drawing in Figure 10. It includes three components: first, the array processor chassis containing mother and daughter board; second, a controller chassis for housing an array controller (ACU), the VME interface to HCSM and ISSM, and slots for commercially available image acquisition and display boards; third, two color monitors for input image and output image display.

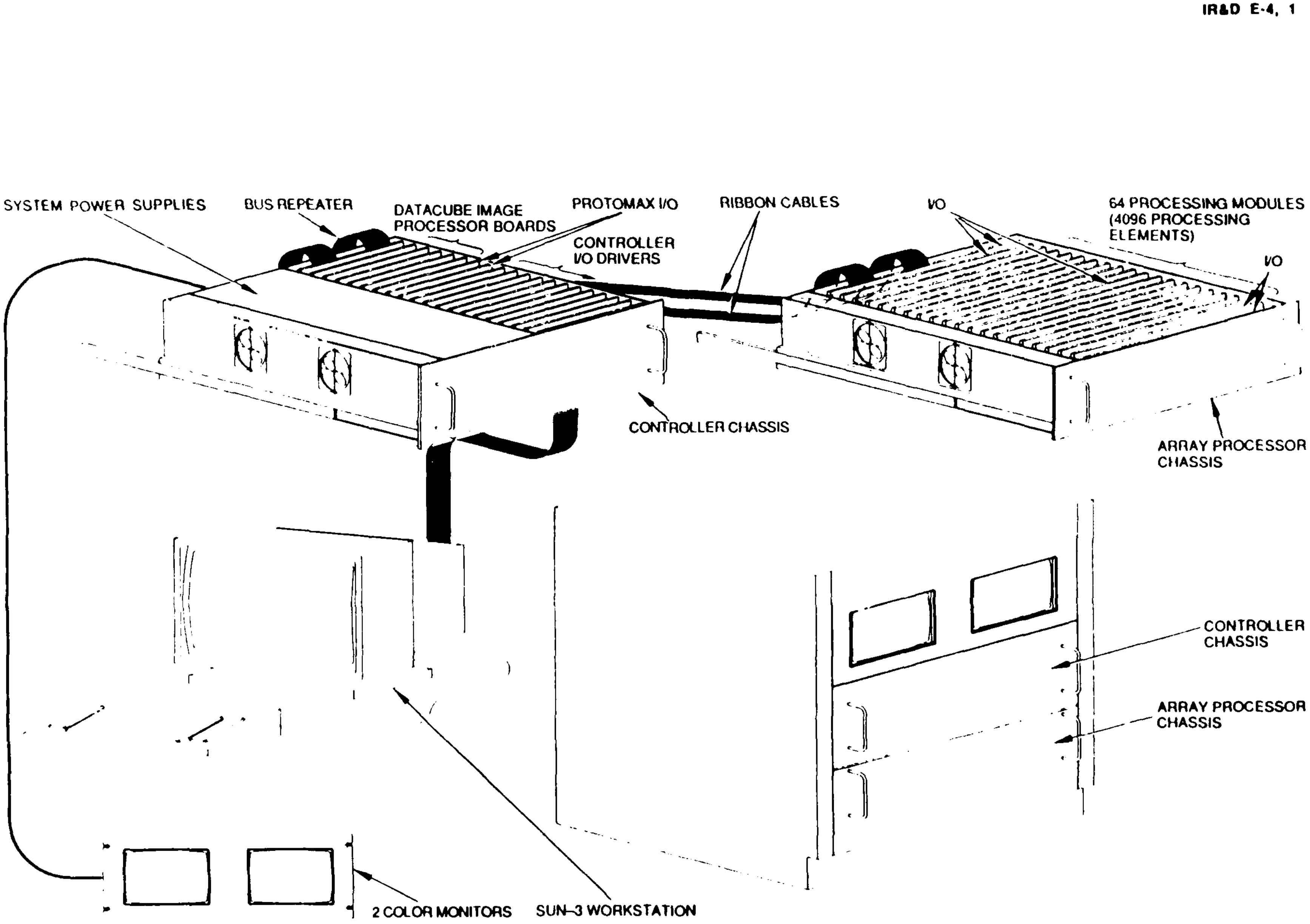


Figure 10. Image Understanding Architecture test bed.

7.0 Milestones/Schedule

7.1 Introduction

The program schedule and milestones are shown in Figures 11(a) and 11(b). As can be seen from Figure 11jor tasks have been started. Note that the concentrator board design, which we haven't started (a subtask of 5) is a relatively simple design and does not represent a lot of effort.

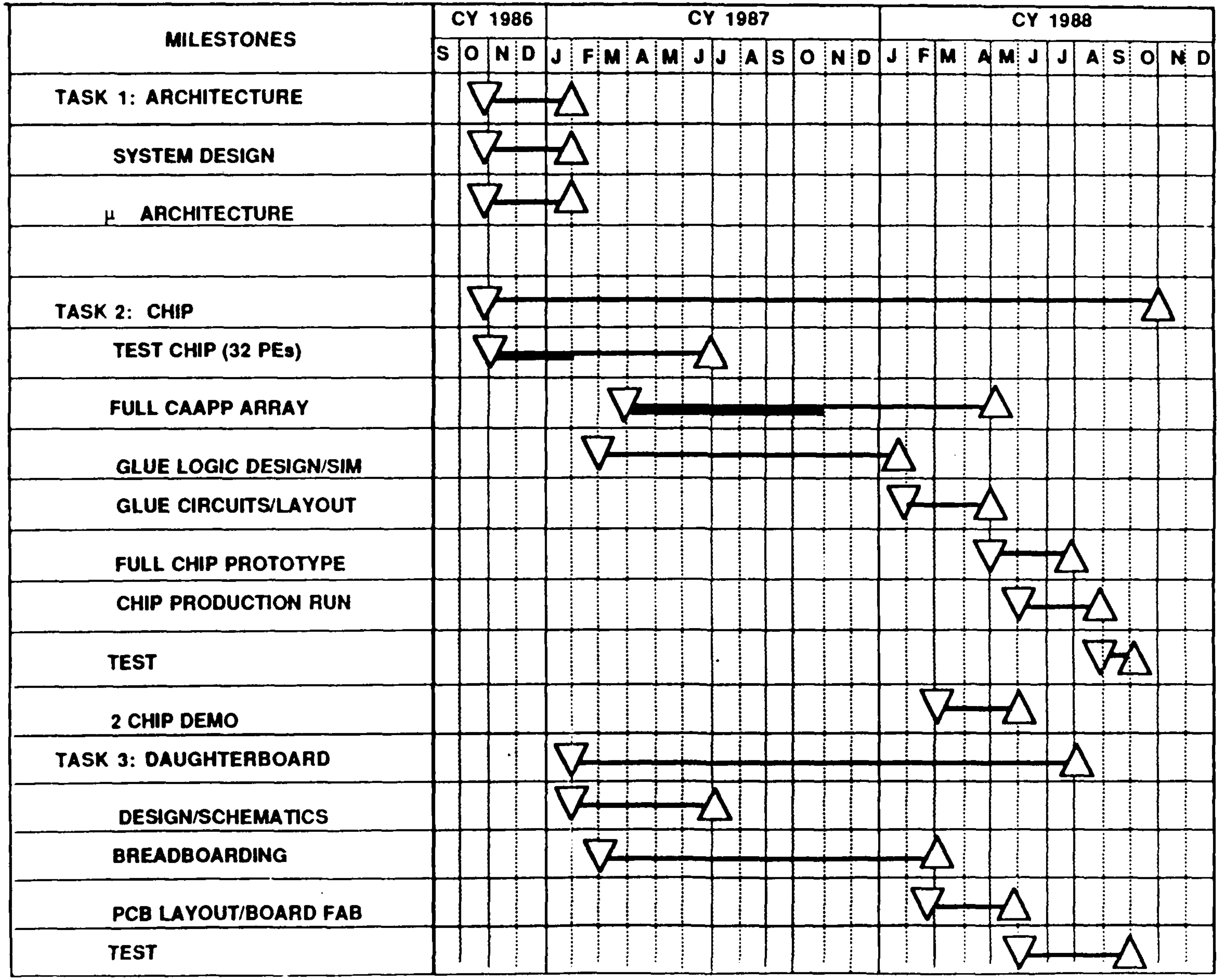
The critical time line is that for CAAPP chip design/fabrication (Task 2), which sets the progress rate for the program. This effort has been slowed down by approximately three months due to problems with MOSIS. The CAAPP has been designed and laid out; however, it still might be necessary to go through another design modification fabrication cycle if we find errors in the design during testing. (More detail on this is provided in Section 7.2.) Neither the daughterboard or motherboard are on this critical time line; however, although we have finished the schematic design stage for both the motherboard and daughterboard, we cannot proceed further until the pin-out decisions have been made for the glue circuitry on the chip. Once this is done, we can begin layout of the daughterboard. When the pin-outs are known for the daughterboard, we can begin layout of the motherboard.

We have added one demonstration at an early stage in the program to provide program proof-of-concept . This will be a two-chip demonstration with prototype versions of the CAAPP chip. (With normal yields we should get at least two working chips out of the prototype run.) For our VME interface to our Sun workstation the two-chip demonstration will use the breadboarded version of our interface card. Almost everything in terms of programs, software and hardware interface needed in the full 64 x 64 can be demonstrated with the two-chip breadboard. Thus, we expect that the full 64 x 64 demonstration should be straightforward. We will use our IMS test system to perform the functions of a primitive controller.

7.2 MOSIS/Chip Fabrication

Our first chip design was shipped to MOSIS for fabrication in January 1987. This chip contained the basic PE logic in a 32-element array along with 64 bits of linear memory. The purpose of this chip was to test the logic and memory cells and to verify the 2μ VTI CMOS process for our future designs. This particular lot of wafers experienced both logistical problems that delayed its progress and technical problems. After receiving it in June 1987 we found that there were metal-metal shorts that prevented direct testing of the circuitry. However, with an ultrasonic probe we were able to do enough testing to verify that the PE logic worked and that the 3-transistor RAM cell

IUA PROJECT MILESTONES FOR HUGHES



Fiqure 11(a).

IUA PROJECT MILESTONES FOR HUGHES

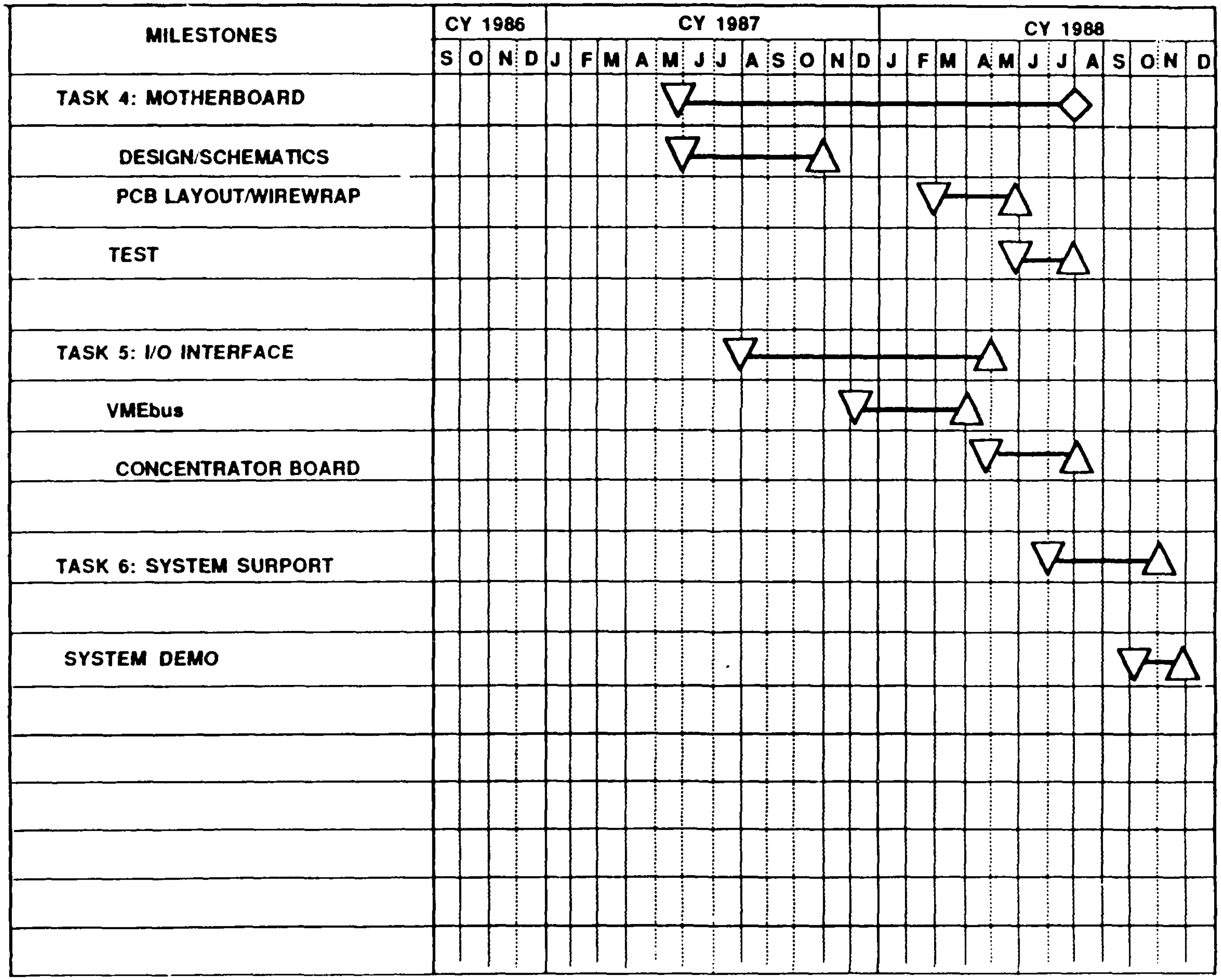


Figure 11(b).

worked. However, we could not get consistently good results for all the RAM cells. We felt that this was probably due to the faulty process and some incorrect design rules.

In discussions with MOSIS in July it was decided that we could go with a single vendor for all our prototyping. We chose National, since MOSIS claimed they were having good results with them. A full 64-PE array was submitted to MOSIS the first week in September for the National run; however, this run was canceled a week later. In the first week of October we submitted the same 64-PE design with a few modifications, based on simulations we had performed. However, in late November this run was also terminated as a result of process/mask problems associated with National's change from a 4" to a 6" line in different facilities. Since MOSIS had not solved all their problems related to the National process in time for the December run, MOSIS decided to send all designs back to VTI as a back-up and then send the same designs to National when the problems had been solved. It appears that the National run was sent off in the middle of December. The next National run is not scheduled until February, so that if there are more problems with the December National run, our schedule could be further delayed. Of course the VTI back-up run could produce good results if VTI has solved its processing problems.

We have also looked into the possibility with MOSIS of having our design run on HP's 1.6μ CMOS line. However, at this point there is not a fixed schedule for the next run and special permission is required for this run. To process our chip at these dimensions by scaling the chip down would probably introduce additional risk into the effort. Also, there would be a moderate amount of extra design effort involved to re-wire the required up-scaled pads and driver circuitry. On the other hand if the chip is run through the HP line unchanged, the mask biasing would not be ideal for our design since we have 2μ design rules instead of 1.6μ. However, we favor the latter approach in terms of keeping the risk down. Tentatively, we will try to arrange such a run with MOSIS.