IMAGE PROCESSING ON A CONTENT ADDRESSABLE ARRAY PARALLEL PROCESSOR

Charles C. Weems, Jr.

COINS Technical Report 84-14

September 1984

This research was supported in part by the Army Research Office under grant number DAAG 29-79-G-0046, the Defense Advanced Research Projects Agency under contract N00014-82-K-0464, General Electric Company (Flexible Automation Systems Program, Corporate Research and Development), and Digital Equipment Corporation.

IMAGE PROCESSING ON A CONTENT ADDRESSABLE ARRAY PARALLEL PROCESSOR

A Dissertation Presented

By

Charles Chilton Weems Jr.

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

DOCTOR OF PHILOSOPHY

September 1984

Department of Computer and Information Science

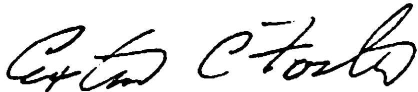
IMAGE PROCESSING ON A CONTENT
ADDRESSABLE ARRAY PARALLEL PROCESSOR

A Dissertation Presented

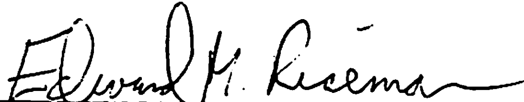
By

Charles Chilton Weems Jr.

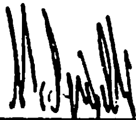
Approved as to style and content by:



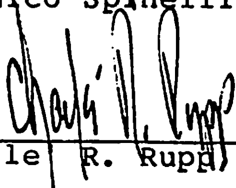
Dr. Caxton C. Foster, Chairperson of Committee



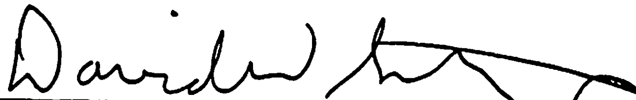
Dr. Edward M. Riseman, Member



Dr. D. Nico Spinelli, Member



Dr. Charles R. Rupp, Member



Dr. David W. Stemple, Graduate Program Director
Department of Computer and Information Science

Charles Chilton Weems Jr.

©

All Rights Reserved

Research supported in part by:
Army Research Office DAAG 29-79-G-0046.
DARPA N00014-82-K-0464.
General Electric Company, Flexible Automation Systems Program, Corporate Research and Development.
Digital Equipment Corporation.

Dedicated To The Memory Of

Herman Gotloeb Herrmann

ACKNOWLEDGEMENT

There are many people who have helped me along the road to writing this dissertation. In particular I would like to thank my committee: Nico Spinelli, for his helpful suggestions, especially with regard to reviewing the literature; Charle' Rupp, for teaching me most of the practical things I know about VLSI; and Edward Riseman, for his many suggestions and a great deal of help with the computer vision aspects of this work. Most of all I would like to thank my advisor, mentor and friend, Caxton Foster. Over the past five years I have learned a great many things from him, some technical in nature and some not. I feel very proud to be able to call myself one of his students.

In addition to my committee, there are many people who have directly assisted me in this research. I would like to thank Al Hough, Daryl Lawton, Martha Steenstrup, Jeff Bonar, Raj Wall, John Adler, Steve Weiss, Ken Estabrook, Kurt Rudahl, and Madhura Kirloskar for their contributions. I would especially like to thank Steve Levitan. He has been a constant source of ideas, encouragement, help, sage advice and moral support for these five years. I am most indebted to Bill Verts, my very good friend and fellow Oregonian, who has kept me from going insane on many an occasion.

I would also like to thank the people who have really made this all possible, the secretaries and staff who do so much to keep life at the University running smoothly: Ruth Morrell, Rose Korowski, Renee Stephens, Louise Till, Bonnie Cichy, Barbara Gould, Janet Turnbull, Susan Parker, Renita Ballard, Rick Newton, Skip Rochfort, and Joey Griffiths.

Three of my former teachers, Robert Anderson, Gene Enfield and Harry Goheen have had a particularly strong influence on the course of my academic career. They are the people who are most responsible for my choosing the path that I have taken.

Most of all, I would like to thank my Mother for her support, encouragement and love.

ABSTRACT

Image Processing on a Content
Addressable Array Parallel Processor

August, 1984

Charles Chilton Weems Jr., B.S., M.A., Oregon State
University, PhD., University of Massachusetts

Directed by: Professor Caxton C. Foster

We present the design of a Content Addressable Array Parallel Processor (CAAPP) for image processing and low to intermediate level vision processing. This new architecture combines: a) associative processing including global broadcast and response to and from the array of cells, and b) array processing via local square neighborhood computation. This combination of capabilities can be used to close the feedback loop between high and low level vision processing, by providing appropriate and effective mechanisms for bidirectional information flow between the CAAPP and its host symbolic processor. A number of algorithms are presented which demonstrate this.

The CAAPP design consists of a square array of 512 by 512 processors, with its own controller, driven by a host processor. The basis of the hardware design is a custom VLSI chip which contains 64 bit-serial processors. We have taken a pragmatic view of fabrication technologies (VLSI,

packaging, etc.), approaching the design very conservatively. The architecture does, however, represent a genuine increase in processing power over the best machines now available. The design of a test chip with 16 processors is presented to demonstrate the feasibility of construction with current technology.

The experience needed to achieve an effective CAAPP design was attained by iterative evaluation and redesign. The evaluations consisted of developing algorithms with a subsequent analysis of architectural effectiveness. Two iterations of the development process are presented here. The first is our experience with a commercially available system. The second is the analysis of our first CAAPP design, based on simulations that gave us statistics for static and dynamic instruction set usage. The result is three new designs, based on the capabilities of three potential fabrication technologies.

Several of the parallel algorithms developed as part of the evaluation process are, themselves, of considerable interest. These include a method of decomposing rotational and translational motion parameters from an optic flow vector field, an order square-root of N sorting algorithm, and a way of performing LISP garbage collections that insures the capability of real-time response.

TABLE OF CONTENTS

ACKNOWLEDGEMENT v
ABSTRACT vi
LIST OF FIGURES xiii
LIST OF TABLES xv

CHAPTER

I. INTRODUCTION AND OVERVIEW 1

The Vision Problem 2
The Architecture Problem 8
The Engineering Problem 11
The Experience Problem 13
The Contributions of this Research 14

II. A REVIEW OF ASSOCIATIVE AND PARALLEL PROCESSORS .19

Introduction .19
Definitions of Associativity 20
What CAM's are Used For 21
A Database Search Example 27
Finding Greatest and Least 31
Some Drawbacks of CAM's 32
Content Addressable Parallel Processors 34
Some CAPP Operations 36
The Add Comparand Algorithm 37
The Add Fields Algorithm 38
Typical CAPP Applications and Environments . . .41
A Survey of Associative Processors 42
The Semionics REM 44
A Review of Parallel Processing and Processors .51
SIMD Parallel Processors for Image Processing .54

III. THE SEMIONICS REM: AN EXPERIMENTAL CASE STUDY IN CONTENT ADDRESSABLE MEMORY 59

Introduction .59
Postage Stamp Data Base 61
Architectural evaluation 62
Real Time LISP Interpreter 62
Architectural evaluation 64
Cryptanalysis of Simple Substitution Ciphers . .65
Architectural evaluation 67

Real Time Tune Recognition 67
Architectural evaluation 69
John Conway's Game of Life 69
Architectural evaluation 72
Text to Speech Synthesis 73
Architectural evaluation 78
Summary of Findings 78

IV. DESIGN FOR A CONTENT ADDRESSABLE ARRAY PARALLEL PROCESSOR 85

Introduction 85
Design Constraints 86
Design Goals 92
The CAAPP 101
The controller 101
The distribution of 262,144 processing elements 104
Processing elements 107
The registers 111
Neighbor communications 113
Broadcast comparand and memory data 113
Activity control 115
Some/none and response count 115
The array edge circuitry 121
The Processing Element Instruction Set 122
Memory operations 124
Register operations 125
Communications operations 126
Special operations 128
The CAAPP Simulator 129
Circuit Board Layout 132
The Special Purpose CAAPP Integrated Circuit . . 134
CAAPP chip pin assignments 135
CAAPP chip overall floorplan 135
Processing element floorplan 141
Support circuitry 143
Processing element circuitry 162
Summary statistics 188
Memory simulation and fabrication 189
Design Conclusions 194

V. APPLICATIONS AND ANALYSIS 195

Introduction 195
Basic Operations 199
Count responders 199
Exact match 202

Greater and less than searches 204
Greatest and least searches 206
Select first 208
Add and subtract constant 214
Add and subtract fields 215
Multiply by constant 217
Divide by comparand 219
Multiply fields 221
Divide fields 223
Simple Image Processing Operations 228
Game of life 230
Gaussian smoothing image convolution . . . 234
Sobel edge extracting operation 245
Histogram 251
Rotate 3-D model and extract frontal surface 254
Higher Level Image Processing Operations . . . 262
Region growing and labelling 264
Decomposition of rotational and translational motion parameters from optic flow 269
Other Applications 288
Center of mass 289
Geocorrection of satellite images 293
Square grid sort 299
Real time execution of LISP programs . . . 305
Simulation of neural networks 311
Graph processing and semantic networks . . 313

VI. EVALUATION AND REDESIGN 319

Introduction 319
Statistical Summary of Algorithms 321
Discussion of Instruction Set Usage 325
Summary of Findings 338
Other Possible Architectural Enhancements . . 341
Memory size 341
Power reduction 343
Faster response count 344
Expansion of the ALU inputs 347
Ability to complement ALU inputs 348
Full width interchip communications 349
Long distance communications between cells 350
Vector broadcast registers 354
Direct selection of individual cells by the controller 355
Build chips from 16 cell blocks of processors 355

Dual ported cell memory 356
Multiple ALU result destinations 358
Multiple controllers connected to subarrays 358
Evaluation of the Proposed Enhancements . . . 363
Conservative Second Design 368
Design constraints 368
Overview of the design 369
Chip and processing element design features 369
Design advantages and limitations 377
Implementation analysis 378
Comparison to the previous design 381
Intermediate Second Design 381
Design constraints 381
Overview of the design 382
Chip and processing element design features 383
The new response count mechanism 388
Design advantages and limitations 392
Implementation analysis 393
Comparison to the previous designs 395
Advanced Second Design 397
Design constraints 397
Overview of the design 398
Chip and processing element design features 399
Design advantages and limitations 402
Implementation analysis 404
Comparison to the previous designs 404

VII. CONCLUSIONS 408

VIII. FURTHER RESEARCH 416

. . . .

BIBLIOGRAPHY 420

APPENDIX A: A HISTORICAL REVIEW OF ASSOCIATIVITY . . . 431

Early Conceptions of Associativity 431
MEMEX 433
Cryotron Catalog Memory 433
Associative Processor and APP 436
GAP or the RADC 2048 word CAM 438
NEBULA 441
ASP 444
STARAN 449
RADCAP or SIMDA 454
General Comments 456

APPENDIX B: A HISTORICAL REVIEW OF SIMD PARALLEL PROCESSORS 460

von Neumann's Cellular Computer 460
The Spatial Computer 461
Holland's Machine 462
The Orthogonal Computer 464
SOLOMON I 465
SOLOMON II 474
ILLIAC III 474
ILLIAC IV 472
PEPE 474
OMEN 478
CLIP 3 480
CLIP 4 482
Massively Parallel Processor 485
MIT Connection Machine 489
Other Machines 491
Summary 492

LIST OF FIGURES

1. Array Topology for all Edge Treatments 96
2. Overview of the CAAPP System 103
3. Off-Chip Communications Network 108
4. Organization of One Processing Element 117
5. On-Chip Response Count Logic 119
6. CAAPP Chip Floorplan 137
7. Test Chip Floorplan 139
8. Processing Element Floorplan 142
9. Super Buffer Circuit 145
10. Checkplot of Super Buffer 146
11. Checkplot of Super Buffer Column 147
12. Address Select Timing Generation Circuit 149
13. Checkplot of Boundary Between Instruction and Address Decoders 150
14. Sample Section of Address Decoder Logic 153
15. Sample Section of Address Decoder Stick Diagram . 154
16. Sample Section of Instruction Decoder Logic . . . 155
17. High Power Buffer Circuit 156
18. Checkplot of High Power Buffer 158
19. Some/None Logic 159
20. External Neighbor Preselect Circuit 160
21. Checkplot of External Neighbor Preselect 161
22. Memory Cell and Driver Circuits 165
23. Checkplot of a Pair of Memory Cells 166
24. Checkplot of Memory Driver 168
25. XOR-Select Section Circuits 169
26. Checkplot of XOR-Select Section 170
27. Neighbor Select Circuit 172
28. On-Chip Neighbor Network Cell Segments 174
29. Off-Chip Neighbor Network Cell Segments 175
30. Off-Chip Neighbor Network Termination 176
31. ALU Function Selector and Neighbor Networks . . . 178
32. ALU Circuit 179
33. Checkplot of ALU and Function Selector 180
34. Register Driver Circuit 182
35. Register Circuit Part One 183
36. Register Circuit Part Two 184
37. Checkplot of Registers 185
38. Checkplot of Register Driver 186
39. Photomicrograph of Memory Cell and Driver from Test Fabrication 193
40. Weight values for the Sobel Operation 246
41. Sample Flow Field 276
42. Difference Field for Sample Flow Field 277

43. Rotational Template Selected by the Algorithm . . 279
44. Translational Template Selected by the Algorithm . 280
45. Sample Flow Field With Rotational Template Subtracted 281
46. Response to Poorly Matched Translation Template . 282
47. Translation Template Generating Example of Poor Response 283
48. Sample Flow Field With Random Spike Noise Added . 284
49. Difference Field for Sample Flow Field With Noise 285
50. Sample Noisy Flow Field With Rotational Template Subtracted 286
51. Multicontroller Processing Array 361
52. Floorplan of Conservative Second Design Chip . . . 371
53. Communication Network for Conservative Second Design Chip 372
54. Conservative Second Design Processing Element . . 373
55. Intermediate Second Design Processing Element . . 387
56. Intermediate Second Design Instruction Set 389
57. Intermediate Second Design Response Count 390
58. Advanced Second Design Processing Element 401
59. Advanced Second Design Instruction Set 403

LIST OF TABLES

1. Semionics REM Instruction Set 50
2. Summary of Findings from REM Case Study 79
3. Summary of Design Constraints 87
4. Summary of Design Goals 99
5. CAAPP Chip Instruction Set 123
6. Circuit Board Connections 133
7. CAAPP Chip Pin Assignments 136
8. Instruction Decoder Patch Table 151
9. Summary Statistics for Test Chip 187
10. Algorithms Developed for the CAAPP Evaluation . . 196
11. Instruction Occurrence Counts for Sample Algorithms 322
12. Algorithm Count for Each Instruction 323
13. Percent of Algorithms Using Each Instruction . . . 324
14. Average Number of Occurrences Per Algorithm . . . 326
15. Macro Usage Statistics 365
16. Enhancements Ranked by Speed Merit 366
17. Conservative Second Design Instruction Set 376
18. Pin List for Conservative Chip and Circuit Board I/O Lines 380
19. Intermediate Chip Pin List 394
20. Intermediate Second Design Circuit Board Connections 396
21. Advanced Chip Pin List 405
22. Advanced Second Design Circuit Board Connections . 406

Preface

"Consensus is a necessary condition for a successful political system."

---Enid Bak

"Any system is characterized by action or behavior that is oriented toward specific goals."

---Talcott Parsons

High quality service at the front line has to start with a concept of service that starts with top management and finds its way into the structure and operation of the organization. The role of management is to build and maintain the culture, set expectations of quality, provide a motivating climate, furnish the necessary resources, help solve problems, remove obstacles and make sure high-quality job performance pays off.

CHAPTER I

INTRODUCTION AND OVERVIEW

"And generally it is good to commit the beginnings of all great actions to Argus with his hundred eyes, and the ends to Briareus with his hundred hands; first to watch and then to speed..." -- Francis Bacon: Of Delays

The focus of this dissertation is the design of a highly parallel computer architecture for image processing and computer vision. There are three basic problem areas that must be integrated in order to produce an effective machine design. These three areas are: Computer vision, computer architecture and engineering. From the computer vision area it is necessary to determine what is required of a machine that can be used to facilitate the solution of the vision problem. From the computer architecture area it must be determined what machine structures should be selected to meet the requirements set forth by the vision area. The engineering problems that must be addressed are the limitations of current fabrication technology, and what can actually be built within those limitations. This involves not only VLSI circuit design, but also integrated circuit packaging technology, circuit board fabrication technology, power and signal distribution techniques, heat dissipation problems, radio frequency electrical shielding, circuit

board and integrated circuit connector technology, and even mechanical mounting technology for housing the completed system.

The Vision Problem

Returning to the first of the problem areas, computer vision, a close examination will reveal that specifying the processing requirements is not a simple task. The difficulty is that the computer vision problem itself is far from being solved, and is currently a rapidly evolving area of research. At this point in time, nobody can give a detailed algorithmic specification for a general vision interpretation system. Rather than a set of specific processing requirements, then, it is only possible to give a list of general features that must be present in any machine that is to be used to significantly advance the vision problem. It is believed that if such machines are built, they will greatly facilitate research and define many issues in machine vision development.

In the discussions that follow, there will be little direct reference to image processing which, usually, refers to the enhancement and classification of images. In general, the computer vision problem subsumes the tasks

performed in normal image processing. The computer vision problem can be summarized as the automatic transformation of an image to a symbolic form that represents a description and an understanding of the content of the image. This process may be referred to as an iconic to symbolic transformation.

From our perspective, the computer vision problem will be described as involving three levels of processing. These are referred to as the low, intermediate and high levels. The low level consists mainly of operations on pixels and neighborhoods of pixels, similar to the types of operations performed in standard image processing tasks. The intermediate level provides an interface between the low and high levels of representation, that is, between an iconic pixel-based representation and the symbolic elements representing visual knowledge. In the UMASS VISIONS system [84, 85], which is the environment in which most of this research was conducted, the intermediate level consists of a symbolic description of the two dimensional image in terms of regions and line segments, and their associated attributes. In some systems this level would consist of representations of surfaces, or more generally, "intrinsic" features of the physical environment that are in registration with the image.

The high level processing relates the symbolic two-dimensional representations of the intermediate level to object descriptions, stored in a knowledge base of information about the three-dimensional world. The result is a symbollic representation of the content of a specific image in terms of the general stored knowledge of the object classes and the physical environment. Communication between these levels is by no means unidirectional. In most cases, recognition of an object or part of a scene at the high level will establish a strategy for further processing and probing at the low and intermediate levels, in order to pull out additional features under the guidance of a partial interpretation.

Based on this general view of the vision problem, a key requirement is a flow of communication and control both up and down through all levels of visual representation. In the upward direction, the communication consists of summary information and statistics that allow processes at the higher levels to evaluate the success of lower level operations, and also the passing of actual symbols. In the downward direction the communication consists of commands for selecting subsets of the image for specifying further processing in particular portions of the image, and requests for information in terms of the intermediate

representation.

The long range goal, of course, is for all of this to be done in "real time". Real time can best be defined as whatever time the processing may take without causing the system to fail to meet its time dependent goals. By this definition, real time is purely application dependent. For example, vehicle navigation in a dynamic environment requires control decisions to be made roughly once each second. This, however, depends upon the speed of the vehicle and the speeds of independently moving objects in the environment. If the vehicle is being visually guided, then several frames may need to be processed in one second in order to determine parameters of motion. Therefore, many of the low level operations necessary to achieve this must be performed at video rate. Specifically, the time for scanning one video frame (one thirtieth of a second) will be used as a basis for speed comparisons. It should be noted that this was simply chosen as a convenient time period for discussion and comparison of operating speeds in relation to real time. Many applications will, as mentioned above, not impose such strong time restrictions.

It is obvious that many applications require some form of massive parallel processing. Consider a serial machine, with a one microsecond instruction time, processing a 512 by

512 image (roughly a quarter of a million pixels). Even if an image operation can be performed in 20 instructions for each pixel, the total time will be five seconds for the operation. Vision processing might require hundreds of such operations, so it is quite obvious that a serial processor will take too long to generate a result even for an application that requires far less than video rate processing. This is due to the well known "von Neumann bottleneck", in which a large memory must be accessed and processed one cell at a time.

From the above description of the computer vision problem, a set of general requirements for a computer vision architecture can be deduced. Not the least of these is the ability to process images in real time, preferrably with a series of many operations being performed in a frame time. As a separate point, this implies that the machine must be able to load (and possibly dump) a complete image in well less than a frame time (or in parallel with the actual processing of a previous frame). Loading a 512 by 512 by 16 bit image in under one frame time represents a rather high data transfer rate. Since a great number of low level operations will be needed to support processing at the higher levels, the speed requirement would tend to indicate the need for a pixel per element class, mesh connected

cellular array processor. It is generally recognized that these provide the greatest speed in performing low level image operations.

Most important of the architectural requirements, however, is that a general vision machine should provide mechanisms for communicating information and control both up and down through the three levels of representation. The machine must be able to provide the necessary summary information quickly, so that it can try a variety of processing approaches to produce the best results. This type of communication will be necessary to permit the autonomous transformation of an image to a set of meaningful symbols. For this reason, the mechanisms that provide the summary information must be applicable to both pixel and symbol data.

A key issue in achieving an effective architecture, is the ability to maintain the low and intermediate representations, pixels and symbolic region, line and surface representations simultaneously in the same machine. The necessity of dumping an image out, for evaluation by a sequential program, must be avoided at all cost. It is simply too time consuming to transfer the volume of information contained in an image. Even if it took no time to dump the information, the time required for serial

evaluation would still be too great. Dumping an image for outside evaluation simply defeats the entire purpose of having a special parallel processor for computer vision. Instead, the computer vision machine must be able to provide enough feedback to the controlling processor to allow all of the operations to take place within the vision machine itself.

The Architecture Problem

The problem facing the computer architect is to design a machine that is specialized for vision processing but which is sufficiently general that new approaches, to the various aspects of vision, can be implemented on it. It is quite simple to build special purpose machines that implement particular image processing algorithms with great speed. However, as mentioned above, computer vision research is a dynamic, rapidly changing area. New algorithms are constantly under development and experimentation. At this stage of our understanding of the problem, for a vision architecture to be considered a contribution, it must be sufficiently fast and general to allow complex experimentation up to the interpretation level.

The basic architectural issues to be addressed for

vision stem from the requirements of the problem: the ability to process both pixel and symbol data, a fast processing rate, the ability to select particular subsets of the pixels for special processing, feedback mechanisms that allow focussing of attention and data-directed processing (without having to dump the image for external evaluation), and the ability to transform an image into a set of meaningful symbols that describe it.

The general solution that we have developed and which will be presented in this dissertation is a machine that is a fusion of mesh connected cellular array processors and associative or content addressable parallel processing capabilities. The review chapter will more fully explain the benefits of combining these two sets of capabilities. To briefly summarize, however, previous research has shown that a mesh connected cellular array is a structure that is extremely well suited to performing basic local image processing tasks. With one processing element per pixel, such a machine can perform very quickly many of the basic image processing operations, including both the pixel and local neighborhood classes of operations. The problem with the cellular arrays that have been proposed is that they generally do not provide for selective processing of pixel subsets (such as collections of regions or line segments

based on location or particular attributes of color, texture, size, shape, etc.), nor do they supply feedback to the controller. In other words, they do not provide the necessary bidirectional communication between symbolic processing and pixel processing. An image is simply loaded, some operations are applied to it, and then the image is returned for external sequential processing or human presentation.

Research on content addressable parallel processors (CAPP), on the other hand, has always emphasized selecting and processing arbitrary subsets of the data elements, providing feedback to the controller and doing whatever is necessary to keep from having to move data in and out of the processor. This is because the time required for loading the data, which is roughly equivalent to the time to serially process the data with one operation, must be included in the total processing time. In order to claim any significant speed increase over a serial processor, a CAPP must be able to average the data load time with a large number of parallel operations. One way of achieving this is to reduce the number of times that the data must be transferred in and out, by eliminating the need to externally evaluate the results of processing. This can be done by providing global summary mechanisms that feed back

to the controlling processor, to allow it to perform the evaluation of the processing without removing the data from the processor. It will be seen that the combination of features provided by cellular arrays and associative processors is exactly what is dictated by the requirements of the vision problem. Thus, the end result of this particular bit of research is the design of a Content Addressable Array Parallel Processor (CAAPP) for computer vision.

The decision to build a computer vision machine based on a CAAPP architecture does not solve all of the architectural problems, however. It would be quite easy to build a CAAPP that is not well suited for vision. Therefore, determining the fundamental operations and processing element characteristics required to produce a good vision processor is the basis for much of the research that will be presented here.

The Engineering Problem

This essentially boils down to: Given what is desirable in a vision machine, how much of it can actually be built. It would be desirable, for example, to give each CAAPP processing element the full power of a minicomputer.

Realistically, however, this simply isn't feasable in a processor per pixel machine. Besides the prohibitive cost, the complexity that such a design would entail will lead to very unreliable hardware.

The approach that has been taken, in the course of this research, has been to develop sets of constraints on the designs which keep them within the limits of buildability for particular given technologies. The technologies involved are integrated circuit fabrication (VLSI), integrated circuit packaging, circuit board construction, connectors for wiring between circuit boards, backing storage, image displays, and so on. For all but one of the designs presented here, the technology employed is mature and reliable. The guiding philosophy has been that it is dangerous to both push the limits of architecture and technology at the same time. To do so is a sure formula for failure.

Thus, it is the engineering problems which will temper the potential solutions for the vision and architecture problems with realism. Throughout this dissertation the vision, architecture and engineering problems are continually being played off against each other to produce a good set of tradeoffs in the final design.

Some of the specific engineering problems that must be addressed are: number of componenents on a VLSI chip, heat dissipation, number of connections on chips and circuit boards, size of circuit boards, lengths of wires, signal propagation times, switching times, interfacing, power distribution, reliability, and cost. Proper evaluation of a design with respect to these can only be done through actual construction. Thus, it will be noted that part of this research involved actually designing VLSI chips to permit this evaluation to be done.

The Experience Problem

There is one more problem area which encompasses all of these three areas. This is the general lack of experience in designing real architectures for vision. Because no one has yet built a true, general vision machine, the initial foray into this realm is highly speculative. For this reason, the approach taken with this research has been to go through several iterations of design and evaluation in order to build a base of expertise on which the final conclusions can rest. This has included experimentation on, and evaluation of, an actual hardware system, as well as the use of several simulators.

The Contributions of this Research

This research began by addressing the experience problem. Before designing a CAAPP system of our own, it was felt that the best course of action would be to spend several months developing expertise by programming a previously existing CAPP system. In this case, a Semionics REM provided the hardware environment. The REM and the results of our experience with it, as well as several local hardware modifications are described in chapter three. It was this experience that heavily influenced the details of the design work that followed. Let us consider the most important of the conclusions from this case study. The some/none and response count report mechanisms are very important because they provide a way for a controlling processor to quickly get a summary of the global state of the processing array. Specifically, some/none indicates whether any or none of the processing elements is in a given state, and the response count is an integer representing the number of processing elements in a given state. Other conclusions include the desirability of having more processing elements with smaller memories in order to increase parallelism, that nearest neighbor connections were valuable both because they allow certain types of algorithms

to be performed and because, given smaller memories in the cells, they provide a means of simulating larger memories, and finally that bit serial processing elements are more cost effective than bit parallel elements.

From these findings an initial design for a CAAPP was developed. The machine was designed as a 512 by 512 array of processing elements, connected by a four way mesh, with hardware some/none, response count, and response resolution (the ability to select a single element for processing when several have responded to a query). Other global report mechanisms were developed in software, including maximum and minimum value, mean, standard deviation, and center of mass. The response count mechanism is particularly innovative in that it is reasonably fast (about 1000 times faster than a serial count) but requires very little special hardware. This is because it takes advantage of the square grid communications network. The processing element design was kept very simple in order to enhance reliability, and consisted of five single bit registers, 32 bits of memory, a simple ALU and some data routing logic.

After designing the machine architecture, it was necessary to show that it could be built. This involved a VLSI design effort that resulted in a one-quarter size test chip with sixteen processing elements. (The eventual goal

was to put 64 processing elements on a single chip.) Although the chip was never fabricated, due to problems of obtaining access to the foundry for which it was designed, various electrical simulations indicate a high probability that it will work. This exercise also addressed the experience problem, giving us a good background in VLSI design, and a solid feel for what could and could not be done with mature fabrication technology. The first CAAPP design, the VLSI implementation of the test chip, and an analysis of the chip are presented in chapter four.

Besides showing that the first design could be cast in silicon and evaluated from a hardware viewpoint, it was decided that a software simulator should also be built that would allow various applications to be developed in order to also evaluate the architecture from a software viewpoint. Roughly 30 such applications were then programmed for the CAAPP, ranging from simple arithmetic macro operations to image enhancement processes. Each of these was evaluated for timing and instruction set usage. These are presented in chapter five. An algorithm for extracting rotational and translational motion parameters from an optic flow field is particularly interesting.

From the instruction set usage statistics an analysis of weak points, delays and bottlenecks in the architecture is

developed in chapter six. This chapter also presents a variety of solutions, forming a shopping list of potential hardware enhancements. The effectiveness of each of these enhancements is then evaluated by developing a figure of merit based on the speed increase that each would provide for a set of frequently used instruction sequences (weighted by frequency of use for each macro), versus a cost measure based on difficulty of implementation for each of the enhancements.

In the last part of chapter six, three sets of design constraints are presented to guide the design of three more advanced CAAPP architectures. One of these is identical to the original set of constraints imposed on the first design. The resulting machine is in some sense the best that we could have done with the original design, given more experience. The second set of constraints is slightly relaxed to take into account three years of progress in what is considered mature technology. The resulting machine design represents what we would build now if given the opportunity. The third set of constraints is even further relaxed to permit the use of the best available technology. The resulting design is probably impractical to build today, but may be buildable in a few years time. The set of enhancements chosen for inclusion in each design is based

upon the speed and cost evaluations discussed in the preceding paragraph. The result is a set of designs that are well tuned for three different levels of technological constraints, to best serve the needs of vision processing. Thus, the final result of this research is a CAAPP architecture that has been carefully tuned to serve as a general computer vision machine, providing the necessary processing power, communication paths, and control mechanisms to bridge the gap between the low, intermediate, and symbolic levels of image representation.

CHAPTER II

A REVIEW OF ASSOCIATIVE AND PARALLEL PROCESSING

Introduction

This chapter presents the basic concepts of associative and parallel processing. It also examines some existing architectures in order to give the reader a feel for what is currently the state of the art. For readers that are interested, appendices A and B provide a broader presentation of the literature review from which this chapter is extracted.

The emphasis here will be oriented more towards associative processing than parallel processing because most people are more aware of the basic concepts of parallel processing than they are of associativity. The purpose of this chapter is to give the reader a feel for what each of these areas has to contribute to the vision problem. In general it will be seen that in associative processors what has been emphasized are control and feedback mechanisms, while parallel processors have tended to emphasize the topology of the communications network that links the processing elements. By combining the strengths of these two lines of research, a new architecture can be formed

which does a much better job of satisfying the needs of computer vision processing than either a pure associative machine or a pure parallel array machine.

It is also the purpose of this chapter to show how associativity can be used to perform in-cell operations that would require, on a typical parallel processor, special logic in each processing element. Thus, associativity can also be considered as a means of reducing the circuit complexity of processing elements without reducing their computational power. This is very important, since reducing circuit complexity will allow more processing elements to be built for a given cost, and therefore permit greater parallelism. Of course, simplified circuitry also enhances reliability.

Definitions of Associativity

The glossary of the International Federation for Information Processing defines an "associative store" as: "a store whose registers are not identified by their name or position but by their content [64]." The glossary also gives a clarifying example in which it is noted that the retrieval of any item in such a store would be accomplished by performing a content search on all of its registers in

parallel with a single operation.

In a normal computer memory a pattern of bits, called the address, is presented to the memory. This activates logic circuits which "decode" the address and select one of the cells in the memory. Data bits can then be read from or written to that cell depending upon a control bit (usually called the read/write or R/W control line).

By contrast a Content Addressable Memory (CAM), sometimes called an Associative Memory, is accessed by broadcasting a data value to all of the cells in the memory. This activates logic in each of the cells which compares the data bits stored in the cell to the bit pattern being broadcast. If the values match, then the cell is selected. In a typical CAM, however, selecting a cell does not usually make it possible to read or write its contents.

What CAM's are Used For

At this point, the logical question that arises is "What good is the ability to select a memory cell if you already know what's in it?" The answer lies in the other name for CAM: Associative Memory. A typical CAM cell is larger than a normal memory cell and is also divided into fields. The

fields contain "associated" pieces of data. The data that is broadcast to all of the CAM cells is compared against only a subset of these fields. Matches, however, select the entire cell. Thus the main use of a CAM is to search a list of records, by a key field, in order to extract associated data. In fact, CAMs are also referred to as "search memories".

The next question that is usually asked is "What good is having a record selected if you can't read or write the contents?" In fact it is possible to read the contents (and sometimes also to write into the cell) but not directly. The reason for this is based in the underlying nature of searching lists of data. The problem is that a list may contain more than one element with identical key fields. Because each cell in a CAM has its own comparison logic and selection status register (called the "response store"), all of the matching cells will be selected at once. This makes it impossible to then simply read the contents of the selected cells -- which one of the cells would the data value be taken from? Before the data can be read out, another operation must be performed to choose one of the selected cells. This is called the "find first" operation because it usually just selects the "first" of the selected data cells. The operation is also called "select first" and

is sometimes referred to as "response resolution". Once this is done, the cell's contents can be read. Of course, if only one cell is selected by the initial search, then its contents could be read directly. Determining the number of cells that have been selected is an operation that is often available in CAM systems.

Although many CAM's do provide a mechanism for counting the number of cells selected by a comparison, the operation is usually either very slow or, if fast, requires very expensive hardware. A less expensive alternative that almost all CAM's have is a single status bit that indicates whether any or none of the cells was selected. This is usually called the some/none bit. It is most frequently used to test for completion of processing of all of the selected cells. This takes the form of a "while" loop that saves the response bits in a set of holding bits (called the "candidate" store), then finds the first responder, processes it, turns off the corresponding candidate bit and loads the remaining candidates back into the response store. The loop ends when the some/none bit indicates that there are no more responders -- all of them having been processed.

This essentially encompasses the basic features of a typical CAM. Almost every CAM ever designed has the

components and operations described above in one form or another as a part of it. The only detail yet to be mentioned is the mechanism by which the subset of fields is selected for comparison. This depends upon the overall organization of the memory. The actual CAM memory cells may be either bit parallel or bit serial. In the bit parallel memory, every bit of every cell contains logic for comparing against broadcast bits as well as logic for combining the results of the comparisons to set the response bits. In bit serial memory, there is only one set of logic elements for comparison in each cell. The search value is then broadcast, one bit at a time, to the cells. As each bit is broadcast, the comparison logic in the cells checks that bit against the corresponding bit in the cells memory. The result of all of the comparisons is accumulated so that if all of the bits match then the response store bit is set. In a bit parallel memory then, the fields are selected by storing a bit pattern in a register called the "mask" register. The mask is then broadcast along with the search value (which is often called the "comparand") and only the comparison logic for bits that are not masked is activated to produce a response value. In the bit serial memory, bits that are not to be included in the comparison are simply never fetched up by the comparison logic, and thus they do not contribute to the response value.

The reason for the two different memory organizations is cost. Bit serial CAMs are much less expensive than bit parallel CAMs because the bit parallel CAM requires several times as many logic elements for the same number of storage bits. The bit serial CAM can use standard memory elements with the addition of only one set of comparison logic for each memory cell. The bit parallel CAM must have a set of comparison logic elements for each bit of every memory cell plus logic for combining the results of all of the bit comparisons in each cell. The result is that if a bit parallel CAM has N bits in each memory cell, it will contain roughly N times as many logic elements as a bit serial CAM with the same number of cells. On the other hand, the bit serial CAM will be N times slower than the bit parallel CAM for operations that can be applied to an entire field or cell at once. We will see, however, that many CAM algorithms are inherently bit serial in nature. In the case of these operations, the bit parallel CAM has no advantage and may, in fact, be slightly slower than a bit serial CAM, due to the necessity of changing the mask for every operation. Even less expensive and still slower CAM implementations exist in which the cells are processed serially as well. Such CAMs are no faster than general purpose computers but are useful as pre-prototype design

test beds in the development of parallel CAMs.

In summary, we can take a look at a list of the typical elements of an associative memory that was given by Minker [81] in his 1971 survey of papers on CAMs:

1) the memory array, which provides the data storage itself;

2) the comparand register, which contains the data to be compared against the contents of the memory array for searches; it may provide a shifting register for some input/output operations, and it can play an intermediate role in the transfer of data between the memory array and a general purpose computer, depending upon the configuration in which the processor is employed;

3) the mask register, which is used to contain data specifying portions of words for operations involving only word portions;

4) the resolver, which is used to determine the location of response bits in the response store;

5) the search logic, which causes the search commands received by the memory to be executed properly -- search operations are generally accomplished in a bit serial, word parallel fashion starting at the most significant bit; and

6) the response store, which receives vectors indicating which data satisfy a given search criterion and which can execute logical operations (such as shifting and Boolean operations on these vectors).

A Database Search Example

To illustrate the use of CAM in a common area of application, we will examine an example of a database query system.

Suppose we have a database stored in a CAM that represents a catalog of astronomical objects. Each entry in the database will consist of the designation of the object, its type, subtype, celestial coordinates, distance, proper motion and system association. Not all of the fields will contain entries for all of the objects.

The simplest query would be to recall all of the information about a particular object. Because each object has a unique designation, we could simply broadcast the object's designation thus selecting the record that contains information pertaining to it and read it out directly.

Another query that we could make would be to ask the

designation of an object located at particular celestial coordinates. In this case we would broadcast values for two fields (right ascension and declination). Celestial coordinates do not always specify a unique object, however, so we will have to use a some/none bounded loop to read out all of the responders.

It is possible that the preceding query could give no response. This would occur if the coordinates we broadcast were not quite accurate and thus gave a position in the sky where no object is present. We can solve this through the use of an inexact matching criterion. One simple way to achieve this is to compare against just the high order bits of the stored numbers. By ignoring some of the low order bits, we reduce the exactitude of the matching operation and are thus checking a range of acceptable values. Doing this in the above example would essentially increase the size of the "window" in space where we are searching for the object. If the search fails again, the precision of the comparison could be further reduced (increasing the size of the window) and another search performed. The process could then be repeated until an object is found.

Often we make conjunctive queries to databases. For example, we might ask for a list of all stars that have high proper motion and are either white dwarfs or K class stars

that are in binary systems. This is a fairly complex query but is easily handled in a CAM. First we set all of the response bits to one. Next we perform a set of queries, each of which will turn off response bits for cells that don't match but will leave the response bits for matching cells unchanged. In this case we would first query for all objects that are stars, then for all objects that have high proper motion. This would give us a list of all stars that have high proper motion. Next we store a copy of the response bits in the candidate store. Now we query for all white dwarfs, giving us a list of all white dwarf stars that have high proper motion. Then we exchange the candidate and response bits. Again we have a list of stars with high proper motion. To this list we apply the query for K class and then the query for membership in a binary system. We now have a list of binary K class stars with high proper motion in the response store, and a list of white dwarf stars with high proper motion in the candidate store. The last step uses some logic that we haven't discussed previously but which is usually present in CAMs -- candidate-response combinational logic. In addition to operations for exchanging and copying values between the candidate and response stores, operations are usually provided for combining the stores through logical AND and OR operations and for inverting, setting or clearing the bits

in either store. Thus the last step is to combine the two lists with a logical OR operation, putting the result in the response store; which gives us the desired list of records.

Because any conjunctive query can be written as a boolean expression it can thus be put into canonical form. This allows us to compute it with just the AND, OR and NOT operations between the candidate and response stores. Note that although the two stores and combinational logic are sufficient for computing any query, they are not necessarily the most efficient means. Some compound queries could require that certain searches be repeatedly performed. With multiple candidate bits, these search lists could be temporarily saved and then recalled when needed. Of course this is only important in bit serial CAMs where searches take time proportional to the length of the fields being compared against. In bit parallel CAMs, the search time would be exactly the same as the time to transfer bits between candidates and responders. Thus bit serial CAMs often have additional hardware for multiple response bits or, alternatively, a mechanism for storing and retrieving response bits from the cell's memory. Using memory bits to store temporary response patterns does reduce the amount of memory available for data storage but bit serial CAMs can usually afford to have more bits per cell because the memory

cells are less expensive than those used in bit parallel CAMs.

Finding Greatest and Least

Another type of query that can readily be done in CAM is a search for records which have the greatest value in some field. For example, we might want to know what object (or objects) have the greatest proper motion. To do this we begin by testing the high order bit of the field to see if it is set to one. If the some/none response bit indicates that some of the records match this comparison, then we know that the highest value must have that bit set (if none of the records have the high order bit set, then we know the converse is true). We then set the high order bit of the comparand to match what we have discovered to be true of the greatest value and the next highest bit to one. We then compare against the two high order bits of the field. If there are some responders then we know that the highest value has its two high order bits matching the combination in the comparand. If there are no responders, then we know that the second highest bit must be zero. We set the comparand correspondingly and then set the next highest bit to one and repeat the comparison. This goes on for all of

the bits in the field. Upon completion, the comparand will hold the greatest value in that field and all of the records which have that value will be selected. Note that this operation is inherently bit serial -- even on a bit parallel CAM, the mask would be used to add one bit at a time to the comparison.

In addition to finding the greatest value, the CAM can also be used to find the least value, all values greater than a given value, all values less than a given value, and (by combining the latter two) all values within a specified range. This last type of query could be applied to the astronomical data base to find all of the objects in a given rectangular section of the sky. (First we would select a range in one dimension and then apply a range selection in the other dimension.) If a CAM is also provided with a mechanism for quickly counting the number or responders, it can easily be queried for counts of data elements, mean, median and mode values, and so on. Foster [40] outlines these algorithms in his book.

Some Drawbacks of CAM's

The point of the foregoing discussion is to show that a CAM is a very versatile searching machine and that under the

right conditions it can perform those searches far faster than a general purpose computer. What are the right conditions? The main condition is that the entire database must fit in the CAM at one time and that it must be allowed to remain there through a large number of searches. The reason for this is that the time required to serially load the CAM is essentially the same as that required for a general purpose computer to serially search the database. Thus the CAM is advantageous only if the overhead of loading time can be averaged with many fast search operations. The report back mechanisms, such as some/none and response count, are an outgrowth of this need to keep a data set in memory for long periods of time. Except in the case where the entire data set cannot fit into the CAM at one time, the major reason for unloading a CAM is to allow the controlling processor to gather some overall statistical information about it (such as a count of the number of responding cells). With the inclusion of hardware mechanisms that provide global summary information about the data set, it becomes possible to leave the data in memory until processing has been completed. This will be further discussed in another section, later on.

Another drawback of the CAM is that although it is a parallel architecture in that it searches all of the records

contained in it at once, it does not provide for processing records in parallel once they have been selected. Each selected record must be serially read out and processed individually. The CAM is essentially a parallel search Read Only Memory (ROM). In fact, the addition of one simple operation, the ability to write in parallel to all of the selected cells, is all that is needed to make a CAM capable of processing a database in parallel as well as searching it.

Content Addressable Parallel Processors

Many CAMs have been designed with the feature of being able to write data into all of the cells selected by the response store. This "multiwrite" operation, (Bird [10]) considerably extends the capabilities of an associative memory. By proper use of the mask register (to achieve bit serial operation in a bit parallel CAM) with the multiwrite operation, it is possible, for example, to perform bit serial addition of a constant to all of the memory locations selected by the response store. The increase in computational power is so significant that Foster [40] distinguishes such CAMs as a separate class of devices, called Content Addressable Parallel Processors (CAPP).

The CAPP is a Single Instruction stream Multiple Data stream (SIMD) parallel processor under Flynn's [36] classification system. A single control unit broadcasts commands to all of the memory cells, each of which may be considered to be a simple processor. The cells then act in unison on the data that they contain. Because the cells may be individually disabled by not being selected by the response store, "local control" can be simulated to a certain extent. This takes the form of the central control unit sequentially selecting each group of cells that require different operations to be performed on them and issuing the appropriate commands to perform those operations. The simulation of local control is limited to situations where there are only a few distinct groups of cells. If the number of groups grows too large, the speedup gained from parallelism is quickly lost because each group must be processed sequentially by the global controller. In the limiting case, the CAPP is reduced to processing all of the cells sequentially. Thus a CAPP can be applied to problems which require a limited amount of independent processing within the cells.

Some CAPP Operations

To illustrate how a CAPP may be used as an SIMD parallel processor, several algorithms will be presented that perform operations commonly found in SIMD architectures. These algorithms are paraphrased from Foster [40] and the reader is encouraged to examine this reference for descriptions of additional algorithms and applications of CAPPs.

One of the simplest operations found in most large parallel processors is the increment, or add one, operation. This adds one to all or a selected subset of the data elements in memory. On the CAPP, this can also be done in parallel on all (or a subset) of the data values, but bit serially. The algorithm works as follows: Every cell has a field which is to be incremented and one extra bit designated to hold the carry (this carry bit is just another bit of the cell memory that isn't being used for anything else). We begin by setting the carry bit to one and then copy the lowest order bit of the data field to the response store. If the bit is one, the situation is that the carry and the bit are one, so the result is that the bit becomes zero and the carry remains one -- thus we multiwrite these values into the appropriate bits of the selected cells. Next we invert the bits in the response store, thus

selecting all of the cells with the data bit equal to zero. In this case, the carry will become zero and the data bit will become one -- which is what we write into the cell's memory. Those cells for which the carry has become zero will not require any further processing. For the next bit we thus select all cells that have the carry bit equal to one and for these we load the next higher order data bit into the response store. The same combination of values (bit=0, carry=1) are again written into the cells. Instead of simply inverting the response store values, however, we must directly select the cells that have carry equal to one and the data bit equal to zero (inverting the response store would select all of the cells that have carry=0 or bit=0). Into these we set the carry bit to zero and the data bit to one, just as we did for the lowest order bit. This process repeats until all of the carry bits are zero or until all of the bits in the field have been processed. (If some carry bits remain set after all of the bits have been processed, then overflow has occurred and can be handled in whatever manner is appropriate.)

The Add Comparand Algorithm

The above algorithm can be generalized to allow addition

of any value to all of the cells at once. Again we start with the low order bit of the cells and also of the value to be added. In this case the carry bit is cleared to zero in all of the cells, then we enter a loop to perform the bit serial addition. In the case where the current data bit from the value to be added is zero, we first select all cells with their current data bits also equal to zero. Into these we write a zero for the carry and a one for the data bit. Next we select all cells which have both the current data bit and carry bit equal to one. For these, we simply change the data bit to zero (and leave the carry equal to one). For the case where the current bit of the value being added is equal to one, we would instead select cells that have a data bit of one and a carry of zero. These would then be set so that the data bit is zero and the carry bit is one. Next, cells with both bits equal to zero would be selected and their data bits changed to one. The process is repeated for all of the bits in the values proceeding from lowest order to highest order.

The Add Fields Algorithm

We can further generalize this to perform addition of two fields within the cells. For example, if one field

contains some integer value and another field contains a second integer, we could add these fields together in all of the cells (or a subset). The algorithm adds the two numbers together, placing the result back into one of the addend fields. These will be referred to as the "result" and "other" fields. The procedure starts by setting all of the carry bits to zero. Then for each bit in the two values the following four steps are performed: Select cells with both addend bits equal to one and carry equal to zero, then put zero in the corresponding result bit and set the carry to one. Select cells with the result field bit equal to zero, the other equal to one and carry equal to zero, then put one into the result bit. Next select cells with both addend bits equal to zero and carry equal to one, setting the result bit to one and the carry to zero in this case. Lastly, select cells with the result bit equal to 1, the other bit equal to zero and carry equal to 1, setting the result bit to zero. At the conclusion of the loop, the result field will contain the sum of the original two values. This algorithm can also be modified so that the result appears in a third field.

Analogous algorithms exist for decrementing, subtracting a value and subtracting two fields. Algorithms also exist for multiplication, division and logical operations. There

is no need to describe these here, the point is that a CAPP is capable of performing standard arithmetic operations on all cells in parallel. The time required for these operations is greater than that for fully parallel arithmetic hardware, but the cost and complexity involved in placing such hardware in each cell of a large CAPP is prohibitive. The result is that the loss in speed can be compensated by the greater parallelism of having many more cells than would otherwise be possible. It is interesting to note that all of the arithmetic operations described above are inherently bit serial, just as was the "find maximum" search.

One more useful addition is the ability to move response bits between cells. This usually takes the form of a circular shift register containing all of the response bits. This hardware adds the capability for intercell processing. Being able to do this permits such operations as relaxation, which is often used in signal processing. It will be shown later how this same mechanism, in the form of a square mesh connecting the response bits, can be used to perform relaxation type operations in the image processing domain.

Typical CAPP Applications and Environments

Most associative memory and CAPP systems have been designed to be driven by general purpose computers. Dugan [28] lists four ways in which an associative memory may be connected to a general purpose computer:

1) peripheral device -- connected on a normal transfer channel in the same manner as a disk or drum.

2) multiprocessor -- a device that has its own instruction repertoire and can operate simultaneously with a central processor.

3) integrated -- the associative memory is embedded as a part of core memory and can operate upon data in both an associative manner and in a conventional manner; and

4) special I/O search controller -- the associative memory is used to coordinate, control, and optimize search operations employing peripheral devices, and thereby to assist the overall computation process by decreasing the imbalance between memory speeds.

With each of the above, variants are possible through different logical designs of the associative memory. These may range from devices that are capable of simple equality searches to devices that contain full parallel hardware,

thereby permitting complex operations and turning the memory into a full fledged parallel processor.

A Survey of Associative Processors

Preliminary to this research, an extensive survey of Associative Processors, that have been described in the literature, was undertaken. For interested readers, the details of some of the more interesting and historically important of these machines are presented in Appendix A. The findings of the survey are summarized here.

Associative and Content Addressable Parallel Processors have two features which typify them. The first of these is to be able to select arbitrary subsets of the data elements for processing, through globally broadcast queries or computational probes. The second feature is some form of global report back mechanism that allows the controlling processor to evaluate the results of a computation or query without having to read out the data. The processors examined provided a wide variety of report back mechanisms, including some/none, response count, first responder (response resolution), and maximum or minimum responders. Note that the computer vision problem requires just these sorts of capabilities: The ability to get feedback from

processing and then selectively process arbitrary subsets of the image data. This two way communication, between the controller and the processing elements, closes the feedback loop that is essential to autonomous interpretation of images.

The purpose of the broadcast and report features is to allow the data to be kept in memory for as long as possible. This is because the efficiency of a CAPP as a parallel processor is greatly reduced whenever it becomes necessary to transfer the data set in or out of the CAPP. CAPP's are only cost effective, as compared with serial machines using hashing algorithms, under four conditions: 1) if the data set is one for which it is difficult to compute a hashing function (because, for example, the range of key values is extremely large with data values very unevenly distributed in an unpredictable fashion); 2) if the data set is one for which searches frequently result in the selection of many data values that can be processed in parallel; 3) when the search criteria are extremely varied or even created dynamically as a result of other processing of the data; 4) when the data itself is very dynamic.

These restrictions severely limit the number of applications that are suitable and cost effective for associative memories and processors. The result has been

the characterization of associative systems as a solution in search of a problem. It should be noted, however, that the four restrictions listed above almost perfectly describe the conditions encountered in vision processing. It would seem that vision is a problem for which associative processing is not only well suited but actually required. Furthermore, the feedback mechanisms that have been developed for CAPP's, to avoid the necessity of unloading the data set for evaluation, are just what is needed to allow autonomous vision processing. Lastly, the vision problem itself does away with the problem that causes the most inefficiency in typical CAPP applications: mismatching of data set size with processor size. In a typical CAPP application, the data set is usually either too small (thereby wasting processing power) or too large (necessitating frequent loading and dumping of subsets of the data). In low to intermediate levels of the vision problem, the size of the data set is fixed at the size of an image. Thus it is possible to build a CAPP that is exactly the right size, so that maximum processing efficiency can be obtained.

The Semionics REM

In order to gain experience with associative machines

and their applications, the first part of this research involved the use and evaluation of a CAPP system that had actually been implemented in hardware. The particular CAPP that was studied was based on the Semionics Recognition Memory (REM). The system is described here as an example of a hardware CAPP. The next chapter will describe the applications developed for the REM and the conclusions from the architectural evaluation.

The laboratory, which became known as the Cam LOgic Study and Evaluation Team (due to the diminutive space in which it resided), was equipped with a Cromemco Corporation System Three microprocessor enclosure that contained the following hardware: A 22 slot S-100 bus motherboard, a double drive single sided, single density eight inch floppy disk drive, a Zilog Z-80 microprocessor based CPU running at two or four megahertz (switchable -- usually run at two MHz), 48 kilobytes of random access memory, parallel and serial input-output cards, a Vector Graphic Corporation bit mapped graphics display and a Votrax S-100 speech synthesizer. Later in the project the graphics card was replaced by a Cromemco "TV-Dazzler". Analog and custom I/O cards were also added later. External to this a Digital Equipment Corporation Decwriter II acted as the console device and a color monitor served as a graphics display.

The associative memory consisted of 2Ø cards of Semionics Recognition Memory (REM) mounted in an external S-1ØØ bus card cage that was connected as an extension to the motherboard in the Cromemco System Three. This card cage was provided with a separate power supply and cooling fans. The Cromemco CDOS operating system, a variant of the popular Digital Research Corporation CP/M operating system, provided the development environment. Applications were written in Z-8Ø assembly language and Pascal or a combination of the two.

Of principle interest here is the architecture of the Semionics REM. Each REM board consists of 4 kilobytes of memory, organized into 16 "superwords" (called swords) of 256 bytes. Since the Z-8Ø processor can only broadcast eight bit values over the S-1ØØ data bus, each byte of a sword must be operated on sequentially. Thus, a REM board may be thought of as 256 CAM's, each one byte wide by 16 deep, which share a common response bit (or as one 256 byte wide by 16 deep CAM which must be accessed sequentially, one byte at a time across its width). Multiple boards may be placed in one system to increase the number of swords and thus allow more operations to take place in parallel.

In theory the REM could be enlarged to any size. However, it must initially be loaded by address. Because

the address range of the Z-80 is only 64 kilobytes, this is an upper limit. Because of practical considerations, such as providing space for standard program memory, this space is actually limited to about 16 kilobytes. In order to get around this, the Semionics REM is designed to allow bank switching when it is being accessed as standard RAM. In this scheme, all of the REM boards occupy the same segment of address space. When it is desired to access a particular bank, a code for that bank is output to a reserved I/O port, causing the desired memory bank to become active and all of the other banks to become hidden. During CAM operations, however, all of the REM boards are active and look at the bus in parallel.

In addition to the 16 superwords and response bits, each REM board has 16 "candidate" bits. The candidates may be used with the response bits through exchange, AND and OR operations. No provision is made for shifting the contents of either the response or candidate bits from sword to sword, nor are the response or candidate bits directly accessible to the CPU. To perform either shifting or access operations requires that the bits be transferred into the sword memory and read out by the CPU. There are also no some/none, find first or response count operations provided by the standard REM. These were added as a local

modification in the form of wiring each response bit to a data port so that the CPU could poll the boards in the REM system serially.

REM may occupy any segment of memory in a system. However an additional four kilobyte block of address space must remain empty in any REM system. It is this empty block that provides the mechanism for executing CAM instructions. Since the Z-8Ø processor does not directly support any type of co-processor operation, the REM boards are set up to recognize a memory write into this empty block (called the memory hole) as an instruction code directed at them. When the upper four bits on the address bus specify a location in the hole, the REM boards decode the lower 12 bits of the address to determine the CAM operation to be performed, and the byte column of the swords to which it is to be applied. The signals on the data bus (ie. the byte which the processor is attempting to write into memory) provide the comparand value for the operation. Some REM instructions do not make full use of this information -- the Set Tags instruction, for example, ignores the data bus and column in the sword.

Because the REM is a bit parallel CAM, rather than bit serial, it uses a bit mask to perform operations that use less than a full byte of memory. Just as the bank switching

is done through an I/O port, so is the setting of the mask register. There is only one mask in any REM system. The desired mask is output to the mask I/O port and is received in parallel by all of the REM boards. Zeroes in the mask will block access to the corresponding bits in a byte of REM, while ones permit access. Table 1 lists the Semionics REM instruction set.

During the first phase of the study the REM was organized into four banks of 12 kilobytes each, 48 swords per bank, for a total of 192 swords. Later on with the addition of more REM this was reorganized into 20 banks of four kilobytes each, 16 swords per bank, for a total of 320 swords.

In summary, the Semionics REM provided us with the opportunity to explore a genuine, hardware implemented CAPP architecture. Once fully assembled, the REM provided a 320 word (256 bytes per word) CAPP with moderately fast some/none, find first and response count operations. These report back mechanisms were quite useable even though they took on the order of 50 instruction times to execute. The experience gained from developing applications on the REM formed much of the basis for our own design work.

Mnemonic	Operation
SETT	Set the tag bit in all SWORDS
CLRS	Clear the secondary tag bit in all SWORDS
EQC	Exact match one byte in all SWORDS with T and S set
IEQ	Set tags and match
EQU	Exact match one byte in all SWORDS with T set
MWR	Multiwrite to all SWORDS with T set
MWNR	Multiwrite to all SWORDS with T cleared
MWRC	Multiwrite to all SWORDS with S and T set
MWNRC	Multiwrite to SWORDS with T cleared and S set
EXST	Exchange S and T
ORST	Logical OR S and T with result to S
ANDST	Logical AND S and T with result to S
CPLT	Complement T
GEQ	Greater or equal match on one byte of SWORDS
GRT	Greater than match on one byte of SWORDS
LEQ	Less than or equal match on one byte of SWORDS
LST	Less than match on one byte of SWORDS

Table 1

Semionics REM Instruction Set

A Review Of Parallel Processing And Processors

Parallel processing is by no means a new concept. It dates back in one form or another even into prehistory. Whenever humans worked together toward a common goal they were engaging in parallelism. Flynn [36] has characterized parallel processing as falling into four categories based upon the data and instruction streams involved. These are Single Instruction stream Single Data stream (SISD -- typical serial processors), Single Instruction stream Multiple Data stream (SIMD), Multiple Instruction stream Single Data stream (MISD) and Multiple Instruction stream Multiple Data stream (MIMD). All of these can be found in natural human activities that have gone on for centuries.

Human beings most often work in an MIMD mode -- each of us works independently of others on our own particular problems, but none the less we are all working at the same time in parallel, often contributing toward a common goal. This is the way that a modern business office works. It is the way that any government bureaucracy functions and also how science progresses on many fronts at once. In this MIMD model we can think of each person as an SISD processor.

Less frequently we find people working in SIMD mode. Because each person is an independently acting entity it is

difficult for people to act in synchrony. This does happen, however, when an attempt is made to combine physical power efficiently. Oarsmen in the Roman galleys worked in this manner as do the rowers in an eight person crew shell today. In both of these cases there is someone in charge providing a common source of synchronization, either by beating a drum or shouting orders. This is equivalent to the synchronous clocking found in modern electronic SIMD processors.

The least frequently found and perhaps the least understood of the categories is MISD. It seems that there are very few cases in which multiple operations can be performed on a single datum with any benefit. Many people have included pipeline processors, which are analogous to production lines found in industrial settings, in the MISD category. Just as many people claim that this is a misnomer. Argument for this classification is that at each station along the line a different operation is performed, but that all of these are acting on the same stream of data. The counter argument is that the operations are not being performed on the same datum in parallel, but sequentially. The real conclusion that must be drawn here is that Flynn's classification scheme, while conveniently simple on the surface, is difficult to apply to actual

systems. For example, in the discussion of CAPP's in appendix A it is noted that there are four different ways of performing operations in such machines (Serial by Word and Serial by Bit (SWSB), Serial by Word and Parallel by Bit (SWPB), Parallel by Word and Serial by Bit (PWSB) and Parallel by Word, Parallel by Bit (PWPB)) and yet Flynn's notation only divides these into two groups (SISD for SWSB and SWPB, SIMD for PWSB and PWPB).

Despite the inadequacies of the classification method, however, one model of MISD processor will satisfy the critics as being genuinely of this class. In this model a single datum is broadcast in parallel to all of the processors which then perform independent operations on it, producing results which can then be collected in some manner. In human affairs this scenario is most often encountered in voting. Each person is presented with the same data but uses different filtering mechanisms and algorithms to come to some decision regarding the information. Everyone then votes in parallel, following which the votes are combined in some fashion to produce an overall result. This may take the form of a yes/no response (as in deciding a ballot measure), a selection from choices (as in electing a candidate), a numerical evaluation (as in Olympic judging) or some other form. In the future we will

most likely see MISD applied to situations requiring fault tolerance -- one example being the computer control system on the current generation of space shuttles in which five computers constantly monitor the same inputs and vote to select a proper response.

SIMD Parallel Processors for Image Processing

It is generally agreed that computer vision requires parallel processing in some form. The quantity of data in an image is simply too great and the processing required is too complex for a serial machine to function, in a reasonable amount of time. Using Flynn's taxonomy, this leaves three types of processor for consideration (SIMD, MISD, and MIMD). The MISD class can also be eliminated because the vision problem is not easily mapped onto such a structure. There is considerable debate over whether vision best belongs on an MIMD or an SIMD processor. It has been suggested that some combination is actually the best choice [106]. Although probably correct, the specific form that the combination must take is still unclear. There is no doubt, however, that for low level image processing tasks the type of architecture that provides the greatest speed is an SIMD cellular array; specifically, the class of architectures in

which there are a large number of processing elements, each of which is designed to operate on a single pixel. Typically, the processing elements in such an architecture are arranged in a hexagonal or square grid, connected by a four, six or eight way communications network. The communications network provides a means of performing image operations that involve groups or neighborhoods of pixels.

At least half of the purpose of this research has been to demonstrate that, properly augmented, a cellular array architecture can be used to not only perform low level image processing, but also to be an effective means of transforming image data into a symbolic representation. (The other half of the research has been to show that such a machine can indeed be built and, through analysis of simulations, to develop a well tuned design.) The specific form of the augmentation is to add the two main features of associativity described above, namely, the ability to select arbitrary subsets of the data elements for processing, based upon some globally broadcast computation or criterion, and a set of mechanisms by which the controlling processor can evaluate the results of processing.

It may seem strange that no processor has yet been designed that incorporates all of these features. An extensive survey of the literature, however, has shown this

to be the case. The interested reader is referred to Appendix B which summarizes this survey by discussing the more prominent machines and designs that were examined. Two of these machines are worth mentioning here, because they come very close to providing the desired combination of features, and because, having been actually built, they represent the current state of the art.

The first of these machines is the CLIP-4, constructed at University College, London [120]. It is a 96 by 96 array of bit serial processing elements, each with 32 bits of memory, connected by an eight way mesh. CLIP-4 provides one main feedback mechanism: a fast response count. There is no some/none output although presumably for this number of processors the count requires no longer to develop and can be used instead. (For larger processing arrays, the time to count becomes considerably longer than the time required to form the logical OR of a bit pattern.) CLIP-4 also does not provide for directly selecting subsets of the processing elements, although this can be simulated by using data values in the memory to mask operations in subsets of the cells. Finally, the small size of the CLIP-4 array requires the image to be broken into subimages which must then be processed sequentially, for vision processing. The machine has demonstrated, however, the usefulness of cellular arrays

for image processing.

The second of the two machines is the Massively Parallel Processor (MPP), built for NASA by Goodyear Aerospace [5]. This is considered to be the most advanced parallel image processor currently in existence. It contains 16,384 processing elements arranged in a 128 by 128, four connected, mesh. Each MPP element is a bit serial processor with 1024 bits of memory. Because the memory is in chips separate from the processing elements, it can be expanded (up to 65,536 bits) at a later date. The MPP elements include a special register that controls processing activity in the cells, allowing easy selection of arbitrary subsets of the elements. The MPP also provides feedback to the controller in the form of a some/none output formed from the data bus. There is, however, no response count mechanism. The array size is also just a little too small for vision processing, although the MPP does include special mechanisms for processing larger images by combining the results of processing on subimages. Even so, real time vision will require the capability for processing a full 256 by 256 or 512 by 512 image in parallel. The MPP is currently being used for satellite image processing where real time is not so severely constraining.

The important point to note about both of these

processors is that neither provides quite the full combination of associative and cellular array features. On the other hand, they both come considerably closer to doing so than any of their predecessors. This would seem to indicate a trend that leads to the type of processor, the CAAPP, proposed in this dissertation. Cellular array processors prior to CLIP and MPP probably neglected the associative features because they had so few processing elements that polling all of them serially was not particularly time consuming. CLIP and MPP are really among the first of their type to have enough processing elements to bring the feedback problem into focus. There is no doubt that in increasing the size of cellular array processors beyond that of CLIP and MPP, designers would eventually reach the same conclusions that the designers of associative processors reached fifteen years ago, and would add the same sorts of response mechanisms. Thus, the concept of the CAAPP would seem to be a natural development that is simply slightly ahead of its time.

CHAPTER III

THE SEMIONICS REM:

AN EXPERIMENTAL CASE STUDY IN CONTENT ADDRESSABLE MEMORY

Introduction

The author's first introduction to content addressable machines was an association with the Nebula CAPP project at Oregon State University. The development phase of the project had been completed a few years prior to the author's arrival. However, the hardware remained functioning and available for use up until shortly before the author moved to the University of Massachusetts. Here another project to study associative architectures was just getting under way. Under the direction of Dr. Caxton Foster a laboratory had just been established which included a microprocessor based associative processor.

The function of this laboratory was to permit researchers to experiment with the implementation of algorithms on an operating associative processor. This had two purposes. The first was to see how a variety of problems could be solved on such a machine, and to evaluate the suitability of the architecture for different problem domains. Recall that associative processors had been called

"a solution in search of a problem". This part of the research was involved with finding problems. The second purpose was to develop an experience base from which further design work could proceed. Recall the experience problem discussed in the introduction to this dissertation.

The research project took the form of a small core of graduate students under the direction of Dr. Foster, along with a larger number of graduate students participating in seminars, developing as many applications for the hardware as the time would permit. For each application a description of the suitability of the hardware along with suggestions for improvement was written or orally presented. Some thirty different applications were developed in this way, with about a dozen actually being programmed to completion. The others were either partially coded or developed as algorithms with estimations of how well they would fit the architecture. This chapter presents six of those applications that form a representative sample.

For a description of the hardware, the reader is referred back the the section entitled "The Semionics REM" in chapter two.

Postage Stamp Data Base

The first application that everyone seems to think of for a CAM is a database. This is no exception for the REM. A database system was written for the REM that was tuned for free form queries about postage stamps. The idea was that a collector could sit down at the console with a stamp in hand and by describing its appearance find out what the official reference number is for it. The system essentially looked for descriptive keywords that it knew about (such as shapes, colors, sizes, values, pictorial subjects and so on) and would check the database stored in the REM for matches. The interesting aspect of the system was that it could also aid the user by suggesting the information that, if supplied, would most narrow down the list of possibilities. The test system was restricted to Irish postage stamps, although it could easily have been expanded to cover any set of data given sufficient amounts of REM storage.

The determination of the most discriminating descriptive data was done by selecting all of the swords that matched the given data. The other description fields for those swords would then be searched and the number of different values counted. The field or fields with the most different values would then be recommended to the user as being most helpful. This was possible because of the fast search

capability of the REM.

Architectural evaluation. It was found that the REM was particularly well suited to database problems in which a data record was large enough to use most of the sword memory. Smaller records would be inefficient and invite splitting the sword into segments that would be tested sequentially. Larger records would make it difficult to link the data (because of the lack of a response store shift operation) and would not leave enough room for storage of intermediate response sets.

The absence of both the Find First and Response Count operations caused a significant slowing of the searches for the most discriminating data field. The speed of the report back operations was improved somewhat by the addition of local hardware modifications.

Real Time LISP Interpreter

Levitan and Bonar [13] developed this application as an experiment to see if a CAM might be used to make real time programming possible with the LISP programming language. LISP is heavily based on the use of recursive function calls and dynamic linked data structures. As such, programs that

are written in the language go through large fluctuations in the amount of memory that they use. The LISP programmer is not responsible for returning storage space to the free storage pool when it is no longer needed. This is done "invisibly" by the LISP system whenever the pool becomes empty. Unfortunately this is a very time consuming task. The data structures are very complex and just because one element no longer points at another does not mean that the now parentless element is really an orphan. It may well have been adopted by other parent elements. The only solution is to search all of memory to see if there are indeed no other nodes which point to it. If this is the case then it is indeed an orphan and can be returned to free storage. Because all of memory must be searched, most LISP systems put off the chore for as long as possible. When the time finally comes to collect garbage, it is done for all of the potentially free nodes at once. Batched garbage collection minimizes the number of pauses that occur in processing but it does not eliminate them altogether. It is these naps that prevent LISP from being used in real time applications. A garbage collection pause could occur at some critical moment and the system will be unable to react in time to prevent a failure. Because much of the research in artificial intelligence is carried out with LISP, this has hindered use of these techniques in real time

applications.

Levitan and Bonar essentially found that a CAM could be used to eliminate garbage collection pauses. This is possible because the ability of CAM to be searched in one operation allows garbage collection to take place as each element is unlinked. When a node in a data structure is orphaned by one parent, its address is simply broadcast to all of the memory. If no other CAM cell responds to the query then it is known that the node can be returned to free storage. Otherwise it is not truly an orphan. Thus, a CAM can guarantee that a free cell will be returned in a fixed amount of time, if one exists.

Architectural evaluation. The REM was found to be moderately successful at implementing real-time LISP. The biggest problem was that the swords were too big. The LISP cells did not need anywhere near 256 bytes to store links and data. Eight bytes (64 bits) would have been a much more reasonable size. In order to make more efficient use of the memory the swords were divided into segments that were processed sequentially. This increased the search time, but still kept the total garbage collection time to within a guaranteeable upper bound. The most useful change would have been to arrange the REM with more words of shorter length. The LISP application also used the some/none

operation. The absence of a hardware implementation of this slowed the searching as well. The local hardware modifications for report back operations provided a faster way of performing this. A true, single instruction time, some/none operation would have provided an even greater speed increase.

Cryptanalysis of Simple Substitution Ciphers

Although Wall [121] did not implement this on the REM, it was tested on a CAM simulator that was installed on a Control Data Corporation 170 series mainframe computer. At that time this was considered to be in the supercomputer class of machines. The application is of interest because it demonstrates another problem to which a CAM can be applied. The reason that this was not done on the REM is that the memory was simply too small to hold all of the information in the data base.

A simple substitution cypher uses a one to one mapping of the alphabet onto a rearrangement of itself. Each letter in a message is replaced by the corresponding letter in the scrambled alphabet. A variety of methods may be used to unscramble such messages. Typically these are based on replacing the most frequently occurring letters of the

cipher text with various arrangements of those that most frequently appear in english until recognizable words begin to appear from which further letters can be determined. The problem here for a computer implementation is the idea of "recognizable words". The human reader who occasionally works a crossword puzzle has no trouble in guessing the meaning of the combination "d*ss*rt*tion". For a computer, however, this means a time consuming search of the entire "D" section of the dictionary. Wall used the search capabilities of the CAM to speed up this process.

Even though the CAM can search very quickly, the problem still arises that there may be several words that match. This is especially true at the start of the process when none of the letters are known with certainty. There is also the problem of what to do when a letter assignment results in a nonexistent word. To solve this, Wall implemented the search using frequency of word occurrence in standard english as a measure of fitness. Given multiple words, the most frequently used was tested first. If a search failed to turn up any words that matched a combination then the algorithm backtracked and tried substituting a less frequently used word for one that was previously tried. Thus the process took the form of a depth first search tree with backtracking. The interesting feature of this approach

is that to implement this system on a conventional memory would require very complex data structures or considerable amounts of search time. With the CAM, essentially all that was needed was to label each test word in the partially solved cipher text with a number indicating how recently it had been substituted. If it became necessary to backtrack, the most recently substituted word would be selected and the CAM queried to find the next most frequently used word that met the same criteria.

Architectural evaluation. The features that were shown to be most useful by this application are: Many cells with memory up to 16 bytes (128 bits), fast some/none, and a fast Find First operation.

Real-Time Tune Recognition

This application was developed by Hough [62] as a demonstration of the ability of a CAM to do real-time pattern matching. The basic idea was to read a series of notes being played on a keyboard and at the same time to be searching the REM for matching notes. As soon as enough notes had been entered to narrow the possibilities to a single tune, the program would print out the name of the song. This became known as the computerized "Name That

Tune" contestant.

The problem was interesting because the input was taken from a rather poor source: The audio output of an electronic toy instrument called a "Bee Gee's Rhythm Machine" (a product of the Mattel Toy Company). This meant that the program had to devote much of its processing time to rejection of noise and keyboard bounce, sampling zero crossings to determine the frequency of a note and so on. Because the oscillator was unstable with respect to temperature and voltage fluctuations, the program used relative intervals to discriminate between tunes. All of this meant that the CPU was simply too busy to take time for standard memory search. Although a hashing scheme might have been employed successfully, the database was constantly having new tunes added to it. This implied a complex hash function would be necessary and, again, would probably have taken too long to compute. With the REM, however, the searches were very fast. Only the slow software based response count was a problem and this was converted to a some/none test, followed by a Find First, Discard First and another some/none test. This method was a bit faster than the full count and was acceptable since it was only necessary to determine if there was one or more than one responder. The tune recognition problem successfully

demonstrated the usefulness of the REM in search intensive real-time applications. It should be noted that the program failed whenever a rhythm background was accidentally selected during a run.

Architectural evaluation. Once again it was found that the length of the REM swords was much longer than necessary to hold the data. If the database of tunes had ever grown beyond the number of swords in the system (320), segmentation and sequential searching of parts of the swords would have been required. It is doubtful that the real-time performance of the program could have been maintained if sequential searching of sword segments had to be done. The application would also have benefitted from fast hardware Response Count, Some/None and Find First operations.

John Conway's Game of Life

This application was developed by the author [123] in order to exercise the parallel processing capabilities of the REM. Specifically it was desired to see if a local neighborhood computation on array elements could be implemented on the REM. This would test the suitability of the REM hardware in dealing with shifting response patterns, performing arithmetic and logical operations, and aiding

updates of a graphics display.

The basic idea in Game of Life is to represent the growth of a culture of cells given certain simple rules about how cells are born and how they die. The cells live on a square grid (although some versions of the game use a hexagonal grid), one cell per array position. The eight positions surrounding any cell form its neighborhood. Whether a cell lives or dies depends upon how many live neighbors it has: too many and it starves to death, too few and it dies of loneliness. If a live cell has two or three live neighbors, it lives on to the next generation. If an empty grid position has exactly three neighbors a cell will be born in it in the next generation. The algorithm for computing the next generation simply has each grid position take a count of the live cells in its neighborhood and then the appropriate tests are made on this value to determine the contents of the cell in the next generation.

In this case the REM was broken into 128 segments of one byte each. The remaining 128 bytes in each sword were not used. Only the first 128 swords were used to produce a 128 by 128 grid. This application was developed prior to the enlarging of the REM from 192 to 320 swords. Had the extra swords been available, however, there still would have been some problem in trying to work with a 256 by 256 array. Two

of the byte columns of the swords were needed in order to implement the shifting operations and to store the intermediate response patterns necessary for all of the arithmetic and logical operations. In actuality this would have been possible to implement, though inconvenient and somewhat slower.

The shifting of response patterns was accomplished in software by transferring the responders to a byte column of the swords and serially rewriting them to their shifted positions. All of the arithmetic and logical operations had to be repeated 128 times each in order to cover the 128 segments into which the array was broken. This meant that parallel processing was only being used for one dimension of the array.

One of the more interesting features of the algorithm used was that it applied the search capability of the REM in updating the display. Once a new generation of cells was computed, the array was updated and all of the cells that changed value were tagged in parallel. These were then selected and the Find First operation was repeatedly applied to extract only those cells that had changed and thereby the processor had only to update the corresponding pixels in the graphics memory. This greatly increased the screen update speed for patterns that did not change large areas of the

screen, and helped to minimize memory contention between the CPU and the display generator.

Because so much of the algorithm had to be implemented with serial software, and so few elements were processed in parallel, the overall speed of execution was only about 10 times faster than a good RAM based implementation. For patterns that changed little from generation to generation there was a significant increase in screen update speed although this was still not outstandingly fast. The slowness here can be attributed to the software implementation of the Find First operation.

Architectural evaluation. This application served to demonstrate the unsuitability of the REM for array operations involving local neighborhoods. It did, however, show that there is no inherent reason why a CAPP could not be used for such applications given the appropriate hardware. This would have to include a fast Find First operation, a means of quickly shifting response patterns between neighbors, multiple response candidate registers to hold intermediate patterns, addition of an ALU of some sort to speed up arithmetic and logical operations, and far more processing elements with smaller memories. It is interesting to note that the processing required to implement Conway's Game of Life is very similar to a number

of image processing tasks. Specifically this is really just a simple form of three by three window convolution. It was this realization that in part lead to the concept of the Content Addressable Array Parallel Processor (CAAPP) for computer vision and image processing.

Text to Speech Synthesis

This was another application developed by the author. The method used to synthesize speech was developed by Elovitz, et. al. [29], at the Naval Research Laboratory (NRL). The technique involves the production of phoneme codes for a Votrax speech synthesizer, directly from entered text, via a set of letter to sound rules. The NRL algorithm was converted to the REM system to test the suitability of CAM for implementing rule based algorithms.

The form of each of the rules used in the speech synthesis algorithm included four parts: a prefix, a root, a suffix and the translation that would be generated if a match was found. The prefix and suffix parts can contain special symbols that represent a class of letters (such as voiced consonants, vowels, sibilants, etc.). Some of these special symbols also indicate a variable quantity, ie. "zero or more consonants". The prefix and suffix parts can

also be empty. The root contains one or more literal characters which must be matched. The translation contains a string of phoneme abbreviations taken from the International Phonetic Alphabet (IPA). The translation to IPA was done in order to build a rule set that is independent of a particular speech synthesizer. It is a simple matter to convert the IPA text to Votrax codes. In fact the same program can accomplish this just by loading a second set of rules and retranslating the original output.

A sample rule from the set of about 330 that NRL developed would look like: #^:[I]^+='IH'. This translates into: One or more vowels, followed by a single consonant, followed by zero or more consonants, followed by the letter "I", followed by a single consonant and then a frontal vowel (E, I, or Y) produces the IPA sound "IH". The root is surrounded by square braces and the equality symbol separates the suffix from the translation.

There are two ways to implement the naval algorithm on a CAM. One way is to load the text into the CAM and then to broadcast the rules sequentially. As each word hears a rule called out that applies to some part of it, that part gets translated. The other way is to put the rules into the CAM and transmit the words sequentially, selecting the "best" response to translate the word. The problem with each of

these methods is the variable length nature of the prefix and suffix. The CAM is good at matching patterns with fixed lengths, even if parts are missing or variable. It is much more difficult to match an unbounded pattern (such as "zero or more consonants") because matches may vary in length. To the CAM this represents a case of getting different cells to respond to different criteria at the same time. This is simply not possible.

The solution to this problem involves placing upper bounds on the variant parts and then expanding the rule set to include all of the possible combinations. In the above sample rule, for example, a search of an unabridged english dictionary might find that no word has more than four consonants in the prefix. Rules would then be generated for the cases of one, two, three and four consonants. All of these rules now have a fixed length and can be used in a CAM search.

Of the two methods for implementation, the one that stores the rules rather than the text, in CAM, was chosen. This was done because it was desired to have the capability of generating speech in real time. The batch processing nature of the text in CAM method precluded this. In actuality the program that was developed did not generate speech during the conversion because the NRL rules were set

up for the two phase translation (Text to IPA, then IPA to Votrax). The point was, however, to see if this could be done if not to actually do it.

For the REM implementation the rule set was not expanded as described above. There were two reasons for this decision. One was that there was not enough time to search a large dictionary for all possible applications of some 330 rules. (This is, perhaps, a job for a large CAM.) Secondly, the REM did not provide enough storage space to hold the expanded rule set. Instead, the NRL rules were loaded into the REM and only the root parts were searched to determine potential matches. Each responding rule was then read out and tested separately to see if the prefix and suffix matched. This usually meant that only three or four rules had to be searched to find one that worked.

The REM was divided into segments for each different size of root, the largest being seven characters in length. Each word was first compared against the segment containing the longest roots that were no longer than the word itself. If the comparison produced a match, the matching rules would be read out with the Find First operation and the prefix and suffix parts would be tested. If none of the rules with matching roots would work, the segment with the next smaller roots would be searched. This would continue until the

single letter roots were tested. Because of the nature of the rules, single letter rules were guaranteed to match. Whenever a match was found, the translation was output and the part of the word that had been translated was discarded. The process would then be reapplied to the remainder of the word.

The resulting speed was a bit hard to estimate because of the access time required to write the output to the floppy disk. Overall, however, the speed was equivalent to about one word every half second. This is about one half of the normal rate of human speech. For comparison purposes the algorithm was recoded using a simple hashing scheme (the search would go directly to the group of rules whose roots began with the same letter as the remainder of the word). This was found to be about ten times slower than the REM version. It is interesting to note that the REM implementation of the NRL algorithm was about twice as fast as the NRL implementation, despite the fact that the NRL version was run on a Digital Equipment Corp. PDP-10 (a 36 bit mini-mainframe) while the UMass version was run on a 2 MHz Z-80 based (8 bit) microprocessor. Some of the speed difference may be due to the fact that the NRL system was written in FASBOL (a compiling version of SNOBOL) while the UMass version was written in Pascal. On the other hand, the

UMass version essentially copied the NRL version of the algorithm by implementing the underlying SNOBOL functions as Pascal procedures. Had an expanded version of the rules been generated and the REM expanded to accomodate them, the application would most likely have run with significantly better than real time speed.

Architectural evaluation. Once again it was found that the sword length of the REM is too long and that more words of shorter length are needed. The algorithm was also significantly slowed by the software Find First operation. A fast some/none test and response count would also have helped to increase the execution speed.

Summary of Findings

Table 2 lists the key architectural enhancements that were found to be desirable as a result of the Semionics REM case study.

Except for the Postage Stamp Data Base application all of the applications studied found that 256 bytes was far too large a word size for a CAM. Most of the applications would have benefitted from having far more words of much smaller size. This varied from a single byte for the Game of Life

Need more PE's with smaller memories.
- Gives greater parallelism.
- Avoids sequential processing of memory segments.

Need intercell communication.
- Linear connection topology is essential.
- Square grid connection will allow image operations.

Larger memories can be simulated by communication.
- Easier to combine memories through intercell communication than to split larger memories into segments.

Need instantaneous Some/None report back.

Need fast Response Count and Select First operations.

Multiple candidate bits are desirable.

Separate response and activity control are desirable.

Bit serial processing is more cost effective.
- Allows use of standard memory cells.
- Avoids frequent changes of a bit mask.

Addition of an ALU to each PE is desirable.

Table 2

Summary of Findings from REM Case Study

to 21 bytes for the Text to Speech Synthesis problem. The Tune Recognition application used the entire REM word to store each of the tunes in the database. It was found, however, that most of the tunes could be recognized in only a few notes. Presumably as the collection of tunes grew the number of notes needed would also increase. Still, 256 bytes is far in excess for most applications.

Although not mentioned in the application discussions, a common complaint of the programmers was that the algorithms were more complex because the response tags determined both the swords that would respond to CAM operations and those that would respond to counts, some/none and find first. It was often desirable to separate these functions. To do so meant storing the response tags into the sword and then setting up a response set for the other function. This indicated a need for a response bit and a separate "activity" bit for each CAM cell. The response bit would take part in the queries from the CPU (Find First, etc.) and the activity bit would control the set of cells that would take part in an operation. If the two had to be the same, then simply copying one into the other would accomplish this.

Almost all of the applications found a need for fast report back operations that would allow the CPU to query the

status of the CAM. Especially important was the some/none operation. This was used regularly as both an integral part of comparisons (inequality tests) and to determine the result of a comparison. The Find First operation was important for sequentially processing multiple responders. Although this type of processing is to be avoided because it eats into the speed gained from parallelism, it is nonetheless necessary in many cases. The Response Count operation was used least frequently of the three report back operations. This may have been in part due to the fact that it was so slow and thus the programmers tended to avoid it. On the other hand it was also noticed that the main use for this information was guiding higher level processing strategy. For example, selecting the postage stamp feature that would most narrow the database search. Thus, a very fast (single instruction time) Some/None was seen as necessary. A slightly slower Find First and Response Count, on the order of a few instruction times, was seen as acceptable although an equally fast version would be desirable.

Another problem that was encountered was that the single response bit required the use of sword memory to hold intermediate search results. The lack of multiple response bits was a problem for any compound query. The first part

of a query would produce a response set that had to be stored in the sword in order to be combined with other parts that were computed later. The candidate bits of the REM were generally unusable as temporary storage because many of the changes to the response bits had side effects that would modify the candidates. The lack of independence between the response and candidate bits was most frustrating because had they been independent, the candidates would have done the job quite satisfactorily. It was usually felt that taking byte columns out of general purpose use in order to store response patterns was a poor practice. Doing so meant that the sword columns could no longer be treated homogeneously, ie. that there were "special" columns that always had to be avoided. Thus, there was a strong sentiment among the programmers that a CAM should contain multiple "registers" for holding temporary response patterns and that these should not be a part of the cell memory.

For those applications that performed arithmetic operations on the cells (such as the Game of Life), it was found that the REM was generally inadequate. The problem arose in attempting to perform bit-serial arithmetic through the bit mask. Simulating bit-serial operation required loading a new mask for every operation, greatly slowing the computation speed. As the REM made no other provision for

arithmetic operations, the necessity of frequently changing the mask value was found to be one of its worst performance points. Because CAM applications most frequently need to do arithmetic operations on operands of varying sizes, bit-serial arithmetic is the best choice. Thus some provision should be made to allow bit-serial operation without having to constantly change masks. Bit-parallel operations are best applied for exact match comparisons, but for most other operations serial-by-bit processing is necessitated by the different field sizes. Even for exact match queries it may be necessary to frequently change the mask if the fields being tested are not exact multiples of the mask width. Thus, it was concluded that a bit-serial CAM is preferrable to a bit-parallel CAM for most applications. The choice of bit-serial operation for the CAM cells has economic implications as well. A bit-serial CAM word can be made up of standard memory components with a single one-bit processing element. The use of standard memory cells gives a much lower construction cost. It also means that more resources can be expended on increasing the power of the processing elements. This will then make up, at least in part, any loss of speed as compared to bit-parallel operation.

Lastly, the Game of Life application pointed out the

need for local neighbor communications. This capability makes it possible for data elements in different cells of a CAPP to be combined. Otherwise, the only operations that are possible are those that deal with the data values as isolated quantities. Being able to pass data between neighbors also makes it possible to combine cells to simulate larger cell memories. The ability to simulate cells with larger memories, through intercell communication, will be important given the judgement that it is better to build more CAPP cells with smaller memories. In such a CAPP with neighbor communications it will still be possible to implement applications that require large memories. For many applications, a linear shift network connecting all of the response bits will be adequate. As the Game of Life pointed out, however, there is a great potential for image processing applications on a CAPP that has response communications arranged as a square grid. It was partly this potential that lead to the CAAPP design that is presented in the next chapter.

CHAPTER IV

DESIGN FOR A CONTENT ADDRESSABLE ARRAY PARALLEL PROCESSOR

Introduction

In this chapter the design of a Content Addressable Array Parallel Processor (CAAPP) [122, 125] will be presented. The design is based upon the conclusions drawn as a result of the preliminary research presented in the preceding chapter. It is also partly based upon several months of discussions with various researchers working in the computer vision domain. It represents a first attempt at specifying a machine capable of supporting computer vision at near real-time speeds. It is also meant to demonstrate that such an architecture is buildable without stretching technology beyond what was available at the time (1981) and that in fact it could be built with technology that was not even state of the art. In chapter 6 we will present modifications to this design based upon three years of experience with it.

The presentation begins by detailing a set of constraints that were imposed on the design in order to keep the construction complexity as low as possible. It then moves on to establish the goals of the design. Following

this the design presentation begins, proceeding top down, that is, from the overall system view to the processing elements and their instruction set. Next, the circuitry and VLSI layout of the major components of the special purpose CAAPP integrated circuit are presented. These were developed for a quarter size test version of the chip, but adequately demonstrate that such a device can indeed be fit into the limits of silicon real estate. The silicon requirements of the chip are further discussed in a section devoted to a statistical summary of the chip. Lastly, the results of fabrication run on the memory circuits of the chip are discussed.

Design Constraints

There were nine major constraints imposed on the design process as an attempt to, in some way, give it a reasonable chance of success. The constraints are listed in table 3. One of the most difficult problems a computer architect faces is keeping a design within the limits of what is buildable. It is so very easy to think, "it would be so simple to just add this one more feature". This invariably leads to a machine that has everything imaginable including the kitchen sink. The result is a development project that

No more than 100 printed circuit boards.

Less than 100 connections per circuit board.

Less than 5000 integrated circuits.

No more than 10 MHz clock rate.

For the custom built CAAPP integrated circuits:

- No more than 40 pins per integrated circuit.
  Allows use of standard packages.
- Less than 2 watts of power per integrated circuit.
  Allows forced air cooling.
- Three micron, single metal, NMOS fabrication process.
  Or any other comparable mature technology.
- Less than 40,000 devices in the integrated circuit.
- Less than 350 mil by 350 mil die size.

Table 3

Summary of Design Constraints

is plagued by problems, cost over-runs and slipping completion dates. In the end it is best to build a little less grandiose design. One then has a working machine to experiment with while waiting for the technology to catch up with "the design that could have been". The guiding philosophy reflected in the following list of constraints is: Don't push the limits of both technology and design at one time.

To minimize overall system complexity, the first constraint on the list limits the machine to no more than 100 printed circuit boards. Machines have been built with many more circuit boards than this, but all have had to deal with troublesome engineering problems. The most notable of these is that such a large system must occupy considerable space and thus signals must travel some distance to cross this space. In a high speed system this makes synchronization, clock distribution and so forth into non-trivial problems. The real goal in limiting the number of circuit boards was to end up with a machine that would fit into the space of two standard 19 inch wide equipment racks.

The next constraint on the list is designed to reduce problems associated with wiring connections. All of the circuit boards would have to be built with less than 100

connections. This is because connectors have been a basic source of weakness in previous large designs. They are subject to wear, corrosion, poor and intermittent contact, stray capacitance and so on. Also, the fewer the number of long wires running around inside of a machine, the less one must deal with crosstalk, RF-leakage, RF-interference, wire defects, and signal delays.

To further control the complexity, a limit of 5000 was placed on the total number of integrated circuits in the system. This is roughly in the same range as the number found in a mainframe size computer system. Although this is a fairly large number, it must be remembered that this is a super-computer class machine and so some complexity is necessary to achieve the performance goals.

The chips themselves were constrained to having no more than 40 pin connections. In fact, the goal of the designers was to keep the number to under 28. There were several reasons for these limitations. First, the largest commonly available packages at the time the design was begun were those with 40 pins. The goal of less than 28 pins stemmed from the reduced package size that this would provide. This in turn would allow the circuit boards to be reduced in size. The smaller the machine, of course, the less problems with signal distribution. Actually the most favored package

was a narrow 22 pin configuration that, while not widely used, was standardized and much smaller than even a normal 24 pin package. The other reason for limiting the pin count of the chips is that these represent further connections and thus potential trouble sources.

The power level for the chips was constrained to two watts each with a design goal of one to one and one half watts. This was done in order to keep the cooling requirements for the machine within the capabilities of forced air type cooling. This is always much easier to deal with in a development project than chilled water or freon coolant. It also will tend to minimize hot spots within the cabinetry. These can lead to repeated failures of the circuitry. Because the machine would be made, for the most part, of special purpose chips, repair time could run to several months if repeated failures exhausted the limited supply of replacement parts. It is also simply unacceptable to have a machine that fails on a regular basis.

It was decided that the chip design would count on access to only a reasonably good fabrication process. This was meant to prevent the design from becoming dependent on the promise of sub-micron, four metal, high speed CMOS (or some equally exotic, for that time, process). It was thus decided that the initial chip design would be limited to

what could be built in single metal layer, three micron feature size, NMOS technology. If better fabrication technology became readily available it would be possible to redesign to take advantage of it. Showing that the chip could be built with less exotic technology only serves to strengthen the claim that the whole machine can be built without pushing both architecture and technology.

The chips are further constrained to a maximum of 40,000 devices per chip with a design goal of less than 30,000 devices. At the time these constraints were developed, state of the art was about 70,000 devices. It was felt that halving this number would lead to a design that could fit into a reasonable silicon die size. Although the design is mostly memory elements and repeated sub-units (which should be easy to compress), it must be remembered that the designers had little VLSI design experience at that point and could not be expected to produce a particularly space efficient layout.

In addition to the limit on device count, the maximum die size was specified as that necessary to hold a 350 mil by 350 mil body area plus wiring pads. This is a fairly large die size, but not to unreasonable. Again, it should be pointed out that the goal is to build a supercomputer and that a few of the constraints had to be left less

restrictive in order to accomplish the goals of the project.

Finally, it was decided that the clock rate for the machine would not be greater than 10 megahertz. Although 50 MHz fabrication processes were available at the time, the lower clock rate was chosen to keep the effort within reason (faster operating speeds require more thorough simulation and care in the layout), and to reduce problems with radio frequency noise, crosstalk and so forth. Further it was considered that any clock rate between one and 10 MHz would be acceptable, as long as the prototype would run reliably at the selected speed.

Design Goals

One may wonder how, given all of the constraints imposed in the previous section, it can be possible to design a machine that is to be a supercomputer class image processor. The answer is: by carefully selecting the design goals. Recall that the main purpose of imposing the constraints in the first place was to keep the resulting design within reason. The following list of goals was chosen because they represented those aspects of the design that were most desirable and which were reasonable to build

with the available technology.

The first goal is really intended to relieve some of the workload of the designer. It is desired that the machine be interfaced to and driven by a host processor of some sort. The preferred host is one suitable for a software development environment, such as a mini-mainframe. This might be a Digital Equipment Corp. VAX-11/780 or similar machine. The host would communicate with a dedicated controller that in turn directs processing in the array. This might be an augmented microprocessor or a custom built high speed device. In either case, the functions it performs are limited to control and interface operations. This saves the designer from having to build in the capabilities of a general purpose computer in order to support software development.

It is desired that the machine be capable of accepting or dumping (preferably both at once) an image in the same time as it takes to generate the image with standard video equipment. This video frame time is one thirtieth of a second for American standards. This may seem like a long time, but recall that a 512 by 512 pixel image contains 262,144 data elements. At 16 bits each, this gives a transfer rate of 16 million bytes per second. This is faster than mass data transfers supported by most processors

that are in the class intended to serve as host. A faster transfer time is desirable to increase the amount of time available for processing the image before the next one will be ready. This will, however, necessitate adding dedicated mass storage devices, image sources, etc. to the processor in order to feed it at the faster rate.

Because image generating hardware is mostly designed to produce square tesselated pixel fields, the CAAPP should be designed to support communications between neighboring cells as if they are arranged in a square grid. It should also be possible to treat the cells as if they are laid out linearly because, as the applications from the previous chapter demonstrate, there are many cases which can make use of either configuration (or, for that matter, a system with no neighbor communications at all).

It was also decided that the CAAPP, when being treated as a square grid, should provide a variety of ways of dealing with the edges. The simplest of these is to treat the edges as if they are a ring of dead cells, and which is thus called the "dead edge" treatment. Another treatment is "cylindrical wrap", in which the elements on an edge are connected to the corresponding elements on the opposite edge. The third treatment is called either "skewed cylindrical wrap", "zig-zag wrap", "spiral wrap", or "snake

shift", and has the edges connected as for cylindrical wrap but shifted over one element. Spiral wrap permits the entire array to be connected linearly (satisfying the preceding goal). By allowing the pairs of edges to independently choose the way that they connect, it is possible to have nine different combinations of edge treatment, each of which gives a different overall topology to the array. These can range from a flat grid to a doubly skewed torus. The effective geometry of the array for each of the preceding connection schemes is shown in figure 1.

One goal that became a basic design philosophy for the machine was that there should be one processing element for each image pixel. The REM experience with having to segment the storage and repeat processing steps for each was a strong influence in this decision. It was found that a system that had more than one datum per cell was much more difficult to program, which meant that many researchers were unwilling to devote the time and effort required to implement many of the more interesting applications. The "pixel per element" style of programming was conceptually much simpler to deal with. It also lead to greater processing speed. Thus, it was decided that a primary design goal would be to build a machine that has 512 by 512 processing elements in order to deal with the sort of image

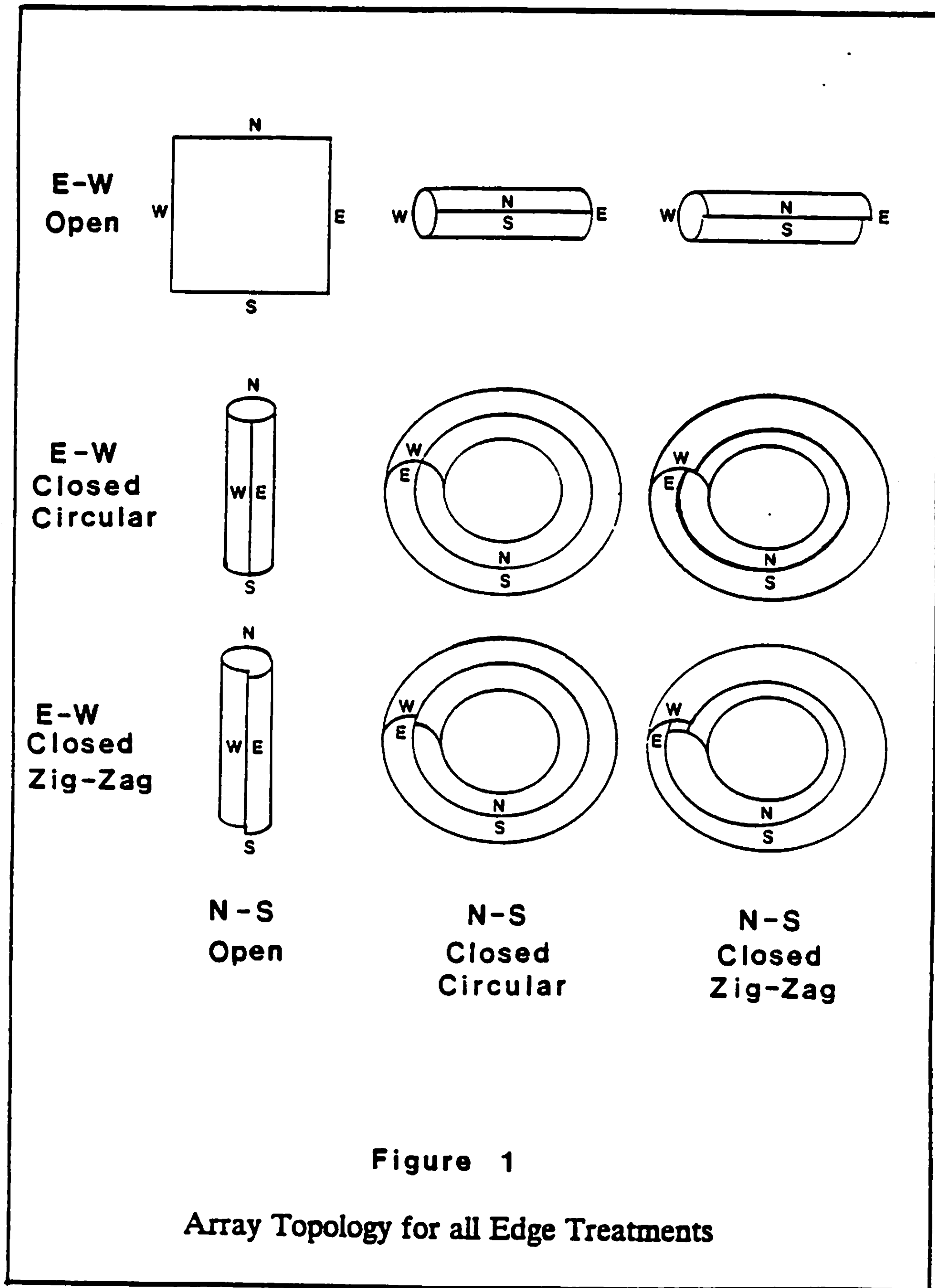


Figure 1

Array Topology for all Edge Treatments

resolution necessary for computer vision.

As per the findings, of the Semionics REM case study, the mode of processing was chosen to be bit serial. As discussed in the review chapter, this allows the use of standard memory cells in the chips, thereby reducing space requirements. It also allows an ALU to be included in each processing element. The goal for the ALU design was to provide AND, OR, NAND, NOR, full add (including automatic carry generation), exclusive or, equality, and complement functions.

It was decided that 32 bits of memory per processing element would be the initial target. This size was selected because it became apparent that each chip would have to contain 64 processing elements, giving a total of 2048 memory bits per chip. At that time, 4096 bits was the standard size for static RAM chips, with 16,384 bits just becoming popular. The feeling was that 2048 bits of memory with 64 processing elements would be more than enough to fill the silicon real estate. The choice of 32 bits was rationalized by the fact that much image processing is done with six or eight bit pixels, permitting four or five operand fields to be placed in memory at once. The experience with the REM showed that 32 bits is sufficient for most applications. It was realized that 32 bits would

be insufficient for processing color images. However, color work could be done at a lower resolution by combining processing elements through the communications network. The REM work had also shown that the programmers were more comfortable with having to combine smaller memories to form the equivalent of larger memories, than they were with the reverse process. The fact that combining small memories was found to be easier than segmenting large ones, was the basis of the general conclusion that it was better to have many more cells with smaller memories. It can be easily seen that this design incorporates that idea.

The REM work had also pointed out the need for separate response and cell activity control bits, which was also included as a design goal. It was also important that it be possible to transfer data between these and to combine them with logical operations. All of these ideas were goals of the design, and are summarized in table 4.

Besides separating the response and activity functions into separate tag bits, it was a conclusion of the preliminary research that there should be more than one of each of these. Thus, another design goal was to provide at least two extra tag bits that could be used for holding temporary values from the response or activity bits.

Host driven architecture.

Video frame time (33 ms) image load/dump time.

512 by 512 array of processing elements.

Square grid communications.

Linear array communications.

Versatile edge to edge connections.

32 bits of memory per processing element.

Bit serial processing elements.

Processing elements include an ALU with:
AND, OR, NAND, NOR, NOT, XOR, EQUAL, full add

Separate response and activity bits.

Multiple response and activity candidates.

Instantaneous Some/None report back.

Fast Response Count and Select First operations.

Table 4

Summary of Design Goals

The final set of design goals follow exactly the recommendations from the preliminary research. The machine would have to provide an instantaneous Some/None feedback and reasonably fast Response Count and Find First operations.

This set of design goals was established entirely with the set of design constraints in mind. The result, however, is still an impressive architecture that is worthy of being called a supercomputer. It contains 262,144 processing elements: an order of magnitude greater in number than the next largest machine, the MPP. Other than the MPP it is the only machine which incorporates associative processing capabilities with a square grid topology. It is even more strongly oriented toward the associative side of the processing because of its emphasis on response report back functions and the separation of response from activity control. It is also the first design which attempts to provide a processing element per pixel for a medium resolution image. At a tenth of a microsecond per instruction this machine will have an effective processing rate of 2.6 trillion instructions (albeit bit serial) per second (TIPS).

The CAAPP

The controller. One of the first decisions that was made in the design process was to make the controller a secondary issue. This was done for several reasons. First among these was that the controller presented no particular architectural challenges, which meant that it could be built with standard technology at a later date. The goal of the design effort was primarily to show that the processor array could be built -- the controller was really a side issue to that problem. Secondly, the controller did present a number of design challenges. Though architecturally simple, it would have to be interfaced with a high speed bus on an existing machine; all in all, a good way for the project to get bogged down in irrelevant details that would only delay the real purpose. Thirdly, it did not make sense to fully specify the controller until after the array had been completely designed. It was anticipated that the array design would have to make compromises between the goals and the constraints and these could well lead to changes in the design of the controller. Thus, the time spent on controller design would be wasted anyway. The result was that the design effort was directed almost entirely at construction details for the array, which lead to a concentration of effort on the actual processing elements

and the design of the special purpose VLSI circuits.

The controller was left specified as a vague entity that was a high-speed supplier of instructions for the array, an interface to a variety of I/O devices and to the host processor. This is shown in figure 2. It was felt that the controller might go through several development phases. In the beginning, when it would primarily be a means of exercising the array for testing, it could simply be a high speed memory with an instruction counter and a stop pointer. Instructions would be loaded into it and then it would be turned loose, dumping the memory as instructions to the array. The feedback data would simply be passed to the host for processing. Later on this might be expanded to include a high speed 16 bit microprocessor that could be programmed to repeatedly refill the memory as a way to simulate looping, and to make decisions based on the feedback information. In the end it could wind up being a custom built, high speed, bit slice machine with dedicated functions for controlling the array. The more sophisticated controller might include a Read Only Memory (ROM) filled with code for commonly used operations. It would be difficult, beforehand, to know what operations should be included in such a ROM. Only experience with real applications on the actual machine would provide this

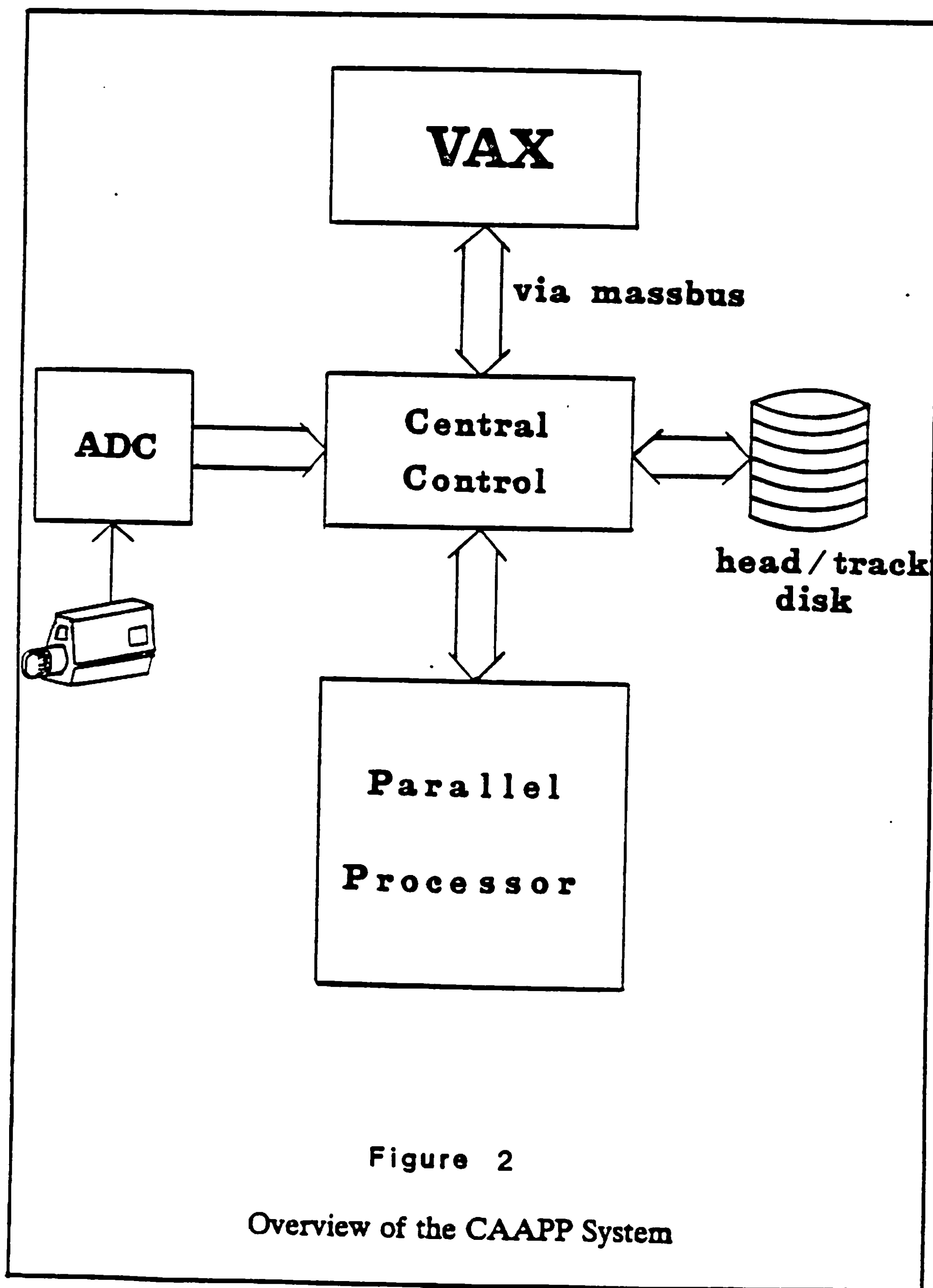


Figure 2

Overview of the CAAPP System

knowledge.

The distribution of 262,144 processing elements. An obvious question in designing such a large array of processors is: How does one distribute 262,144 processing elements so that they fit on less than 5000 chips with less than 40 pins, and less than 100 circuit boards with less than 100 connections. A very symmetrical solution is to have 64 circuit boards, each with 64 of the special purpose integrated circuits, each with 64 processing elements. The real problem then is not how to distribute the processors so that they fit the available space, but how to get them to communicate over so few wires.

The first thing to do is to examine the interface boundaries and to determine the geometry that produces the minimum number of processing elements that lie on them. In this case the area is 64 processors (or chips) and the minimum figure is an eight by eight square with a perimeter of 32. Any other figure will place more processors on an interface boundary and thus increase the number of communication lines that must cross the boundary, which of course translates into more connections.

The problem is still not solved, however. This gives each chip 32 communications lines (leaving only 8 for all of

the remaining functions) and each of the circuit boards 256 connections. Considering just the circuit boards for a moment, if communications lines are run in ribbon cable with a ground wire between each signal wire (to reduce crosstalk), each circuit board would have a nearly three foot wide swath of ribbon cable connected to it. Such a large number of wires and connections is clearly a source of potential unreliability and is thus unacceptable. How can the number of communications lines be reduced? The answer is to time multiplex the communications. Each chip will have only a single communication line for each of the four neighboring chips that it must reach (N, S, E, W), which gives each chip a total of four such lines and each circuit board a total of 32 such lines, leaving plenty of room within the design constraints for the other necessary signals.

It should be noted that the pin and connector limitations automatically precluded the use of an eight way interconnection scheme (N, S, E, W, NE, SE, NW, SW). With eight way connections the number of wires at each boundary is tripled, resulting in 96 wires at the edges of the circuit boards. In addition, the only reason for an eight way interconnection pattern is to speed access to the cells in the corners of the local neighborhood. With a

multiplexed system, this makes little difference in the overall speed, which leads to the unfortunate drawback of the multiplex solution: communications are much slower. In this case it takes at least eight instruction times to transfer a row of data bits across a chip boundary. Eight operations must be done in order to shift any pattern of bits as a whole among the processing elements of the array. Still, one such parallel shift is far faster than a uniprocessor could transfer so much data. It seems a reasonable compromise in order to get such a reduction in the number of communication lines.

The first scheme that comes to mind for implementing multiplexed communications of this sort is to provide a register at each edge of a chip that accepts in parallel all eight of the bits that are being sent by the edge cells and then shifts this out bit serially. On the other end of the wire, a similar register receives the bits serially and then stores them in parallel to the edge elements. There are two problems with this scheme. One is that the registers take up considerable space on the chip. The other is that ten instruction cycles are needed, rather than eight, to shift a pattern in the array. A minor problem is that the load and store operations require two more special purpose instruction codes that could otherwise be used for array

operations. Lastly, it means that edge cells must be somehow treated differently -- that is there will be certain instructions that only they will respond to, which leads to more design complexity.

A better solution is to provide a special topology for the neighbor communications lines. The basic grid connects all of the cells in a chip into a rectangular dead-edged array. By adding connections between opposite edges that are shifted one cell over, (analogous to the snake shift edge treatment for the entire array), the cells are formed into a single long linear array in either direction. The off chip communications network can be seen in figure 3. When the ends of this network are connected to the ends of matching networks on adjacent chips, it becomes possible to transfer bits between chips by simply shifting them along the chain. When they fall off the end of one of the chips, they enter the first position in the next chip. By shifting eight times (the number of cells on an edge), the effect is that the entire pattern in the array is moved over exactly one row or column, which eliminates the need for extra registers, special instructions and special treatment of the edge cells. It also reduces the data transfer time to the theoretical maximum for the given multiplex ratio.

<u>Processing elements</u>. Each of the CAAPP processing

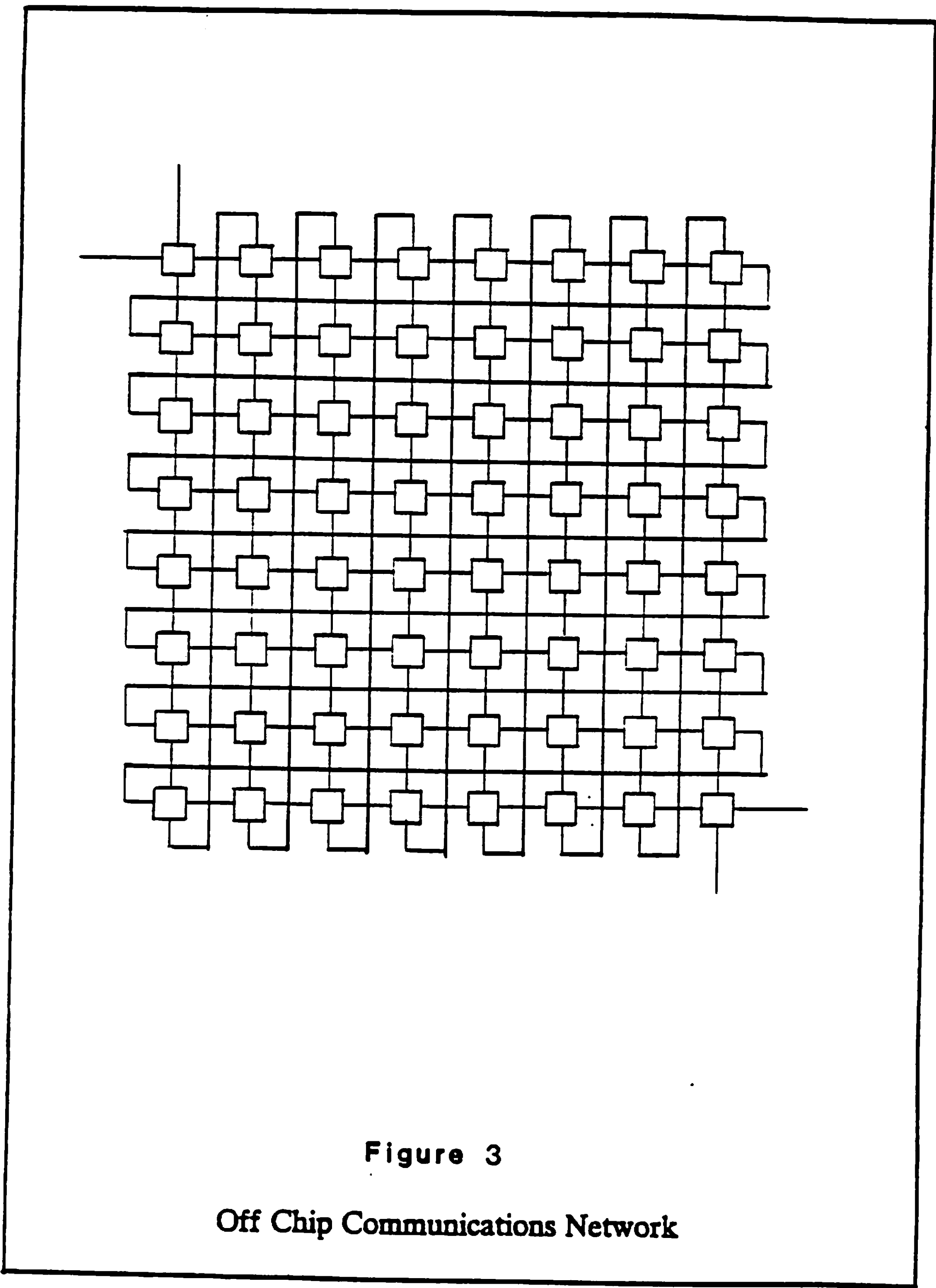


Figure 3

Off Chip Communications Network

elements is a small bit serial processor with local data memory. There is no provision for program memory in the element because the overall design of the processor is as an SIMD machine. Also the elements must be kept as simple as possible in order to fit so many of them onto a chip. In order to get different sets of elements to execute different sequences of instructions it is necessary to set up different patterns for the activity control bits, one for each subset of elements, and to sequentially load these and execute the desired set of operations for the subset that the pattern activates.

As per the design goals, each processing element is specified as having 32 bits of local memory, an ALU, several registers, access to neighboring cells, access to a globally broadcast data bit and provision for feedback to the controller. It was decided that for the initial design, the memory bits would be static cells with no special read/write circuitry. This meant that each could be treated simply as a flip flop that would provide and accept standard logic levels. The alternative was to design the cells as dynamic elements that would have to be periodically read and refreshed. This would save considerable space but would complicate use of the chips. Another alternative was to shrink the cells so that they would provide marginal logic

values on read out that would then have to be amplified before they could be used by the rest of the circuitry. Because there were no suitable simulation tools available to the design effort at that time, the decision was made to go with the most robust type of memory cell.

The ALU in each processing element was designed to provide all of the functions specified in the design goals. Consideration was given to providing all 16 logic functions for two input bits, but the realities of space limitations quickly squelched the idea. The functions directly supported by the ALU are NAND, NOR and the complemented full add operation. The inputs to the ALU are the contents of three of the registers in the cell. The ALU also passes the value of the registers directly through. The design of the ALU is such that it is always generating these functions. Following the ALU is a one of N selector circuit that chooses which of the functions is to be used. This is passed on through some additional selector circuitry and eventually reaches an exclusive OR gate that takes one of its inputs from the controller. The result is that when the controller raises this line, the selected function is complemented. When the line is low, the function retains its original logic sense. Thus, the number of ALU functions is doubled by this simple arrangement. This also allows the

use of inverted logic elements, which are more space efficient, in the ALU. The resulting set of ALU functions is then AND, OR, NAND, NOR, full add, complemented full add, any register value, and the complement of any register value. The only remaining function is the equality test. This is obtained by setting the carry register to zero and applying the complemented full add operation to the bits being compared. (Complemented full add with no carry input is equivalent to inverse XOR, which is the equality test.)

The registers. Much consideration was given to the number of registers that each cell should have and what functions should be assigned to each. At one point a stack mechanism was even considered for the response and activity bits. It was obvious that at least three register bits would be needed. The design already called for separate response and activity stores as well as a dedicated carry bit. The design goals also called for more temporary storage bits for intermediate response and activity patterns. The latter was especially important because the small cell memory would most likely be fully taken up with data, leaving no room there for temporary storage. The reality was, however, that register bits would be expensive in terms of space. Although their design is essentially the same as the memory bits, the memory bits share a single wire

for reading. Because multiple registers must be available at one time, they each required their own output line. (The ALU itself requires simultaneous output by three of the registers, plus the activity register output is used to control the cell activity at all times.) Thus a compromise was made by providing two additional registers. One of these was intended to provide a backing store for the response bit (and to act as the second data input to the ALU). The other would be primarily a backing store for the activity bit. These two registers could, however, be used as general purpose temporary storage if so desired. Further, the carry bit could even be used to hold a temporary value as long as no addition operation was selected. The automatic carry, though always being generated by the ALU, need not be written into the register unless the add operation was actually selected by the function selector. This allows other data to occupy the carry bit at other times. The registers were eventually given the names X, Y, Z, A and B. The X register is the response bit, acts as one of the data inputs to the ALU and also presents its value to the neighboring cells at all times. The Y register is the backing store for response patterns and the second data input to the ALU. The Z register holds the carry bit. The A register contains the activity mask. When this is one, the cell is active and

responds to all instructions. When zero, the cell only responds to a special subset of the instructions. This is necessary, for example, to allow the A bit to be set back to one after being turned off. The B register is the backing store for the activity mask.

Neighbor communications. As previously mentioned, the X register is continuously transmitting it's value to the neighboring cells. This means that each cell is continuously receiving values from all four of its neighbors at once. Actually there are eight such inputs because there are two different configurations of neighbors. One of these is the dead-edged grid and the other is the snake shift network. It was found that the least expensive way of implementing these two connection schemes, even though they share a significant common element in their organizations, was to build them separately. This will be further explained in the discussion of how the chip was constructed. The result is that each cell sees the outputs of eight X registers. Since only one value can be selected for writing into a register, a selector circuit (one of eight) had to be provided. This too feeds into the complementing exclusive OR gate, allowing the complemented value of a neighboring response bit to be read.

Broadcast comparand and memory data. The remaining data

elements that can be selected for writing back into a register or to memory are the broadcast comparand bit from the ALU, and one of the 32 memory bits. The design does not provide for memory being read, and stored back to itself for two reasons. One of these is that memory can only be transferred to a register -- it cannot be directly input to the ALU. Without any form of modification, the only other use for fetching and writing memory in the same instruction is to move bits around. This would require two addresses to be input to the chip at once and would thus expand the number of pins required to get the instructions into the chip. This was considered to be a large expenditure of resources for a gain of limited value. It is much more cost effective to restrict the elements to single address operations and thus use two instructions to move a memory bit, rather than one.

These two data sources are the last stage of the selection process that begins with the ALU functions, register values and neighbor values. Of all of these, only one is selected by each instruction. The selected value is then input to the globally controlled complementing XOR gate and written back into one of the registers or a memory bit. Thus data begins at one of the sources (possibly an ALU result), passes through a selection network, is possibly

complemented and then goes back to a register or memory for storage until it is needed again. The only exception to this process is that an addition instruction will cause the carry bit (Z) to directly take the carry result via a dedicated communication line, at the same time that the result of the add is being stored into another register.

Activity control. Actually there is another case where a data bit might not follow this path. If the activity bit (A) in a cell is zero, the entire process takes place as before, but all of the storage elements in the cell are inhibited from performing the final write operation. This prevents any change of state in the cell and thus it acts as is it had been deactivated. A special group of instructions is provided which ignores the status of activity. This is done by passing the output of the A register through an OR gate with a second input from the instruction decoder. For any of the special instructions, this input is made high and the cell is forced to perform the operation no matter what the state of its A bit. Without this it would be impossible to restore activity to the cell. Other operations that must ignore activity include the neighbor transfers -- without this, a transfer would result in the array being filled with "holes" in the data near inactive cells.

Some/none and response count. The feedback to the

controller is all that remains to be discussed in the overall description of the processing elements. (The next section presents the instruction set.) The some/none circuit is very easily implemented in NMOS as a 64 input NOR gate, the output of which is complemented before being sent off chip to be combined with all of the other chips. The inputs to this NOR gate are, of course, the values from the 64 X registers on the chip. Figure 4 presents an overview of how all of the components fit together to form a processing element.

The response count was a more difficult matter to implement. The objective is to provide a count of the response bits on each chip to be summed externally. A number of ways have been proposed for doing this [44], but the simplest in this situation was to use the snake shift network on the chip (hereafter called the off-chip shift network) to feed a counting register. One of the off-chip outputs is disconnected from the outside world and fed to the input of a seven bit counter register. (Seven bits are needed because values from zero to 64, inclusive, must be represented.) To simplify matters, the input to the same network is reconnected to the input of this register. Counting the number of responders is then a matter of clearing the register and executing a circular shift of all

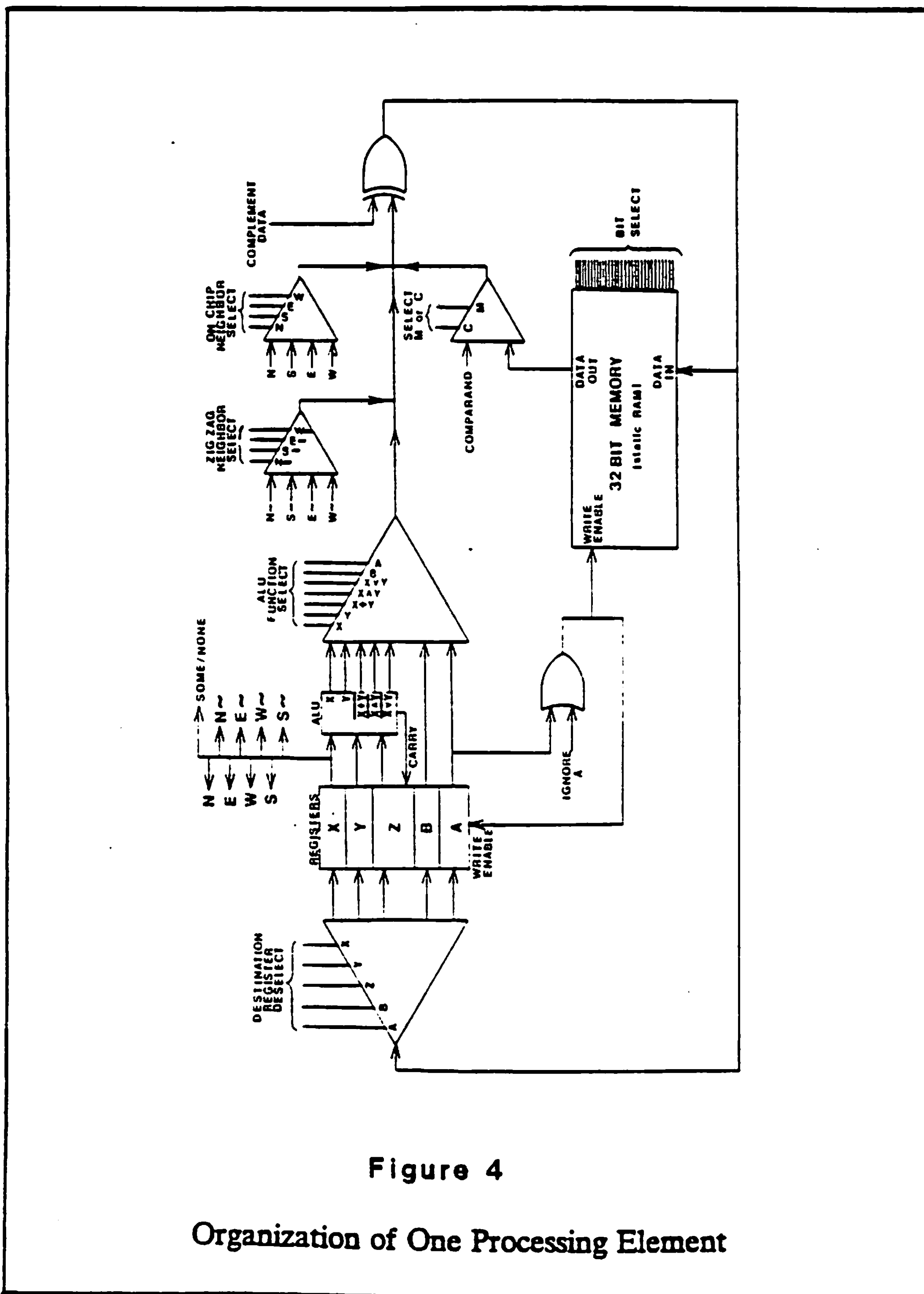


Figure 4

Organization of One Processing Element

64 response bits past the input of the counter. At the end of this operation, all of the response bits wind up back at their original locations and their count is stored in the register.

The most interesting part of this scheme is the way in which the count is passed out of the chip. The method actually results in a count being formed for each column of chips in the array, but without making any part of the on chip circuitry dependent on the overall size of the array. This last point is very important. Although the designers were willing to build array size dependencies into the chips as long as they did not prohibit building arrays smaller than the goal, there was a strong belief that the best strategy would be to make the chips entirely independent of array size.

The column summing response count scheme involves adding only a full adder to the circuitry described above, and making the count register capable of shifting its contents out from the low order end. This is diagrammed in figure 5. The idea involves a pipelined addition of all of the on chip counts, proceeding from the top chip in each column to the bottom chip. The full adder outputs its result to the chip below via the south neighbor communication line. It also produces a carry which is recirculated back to one of the

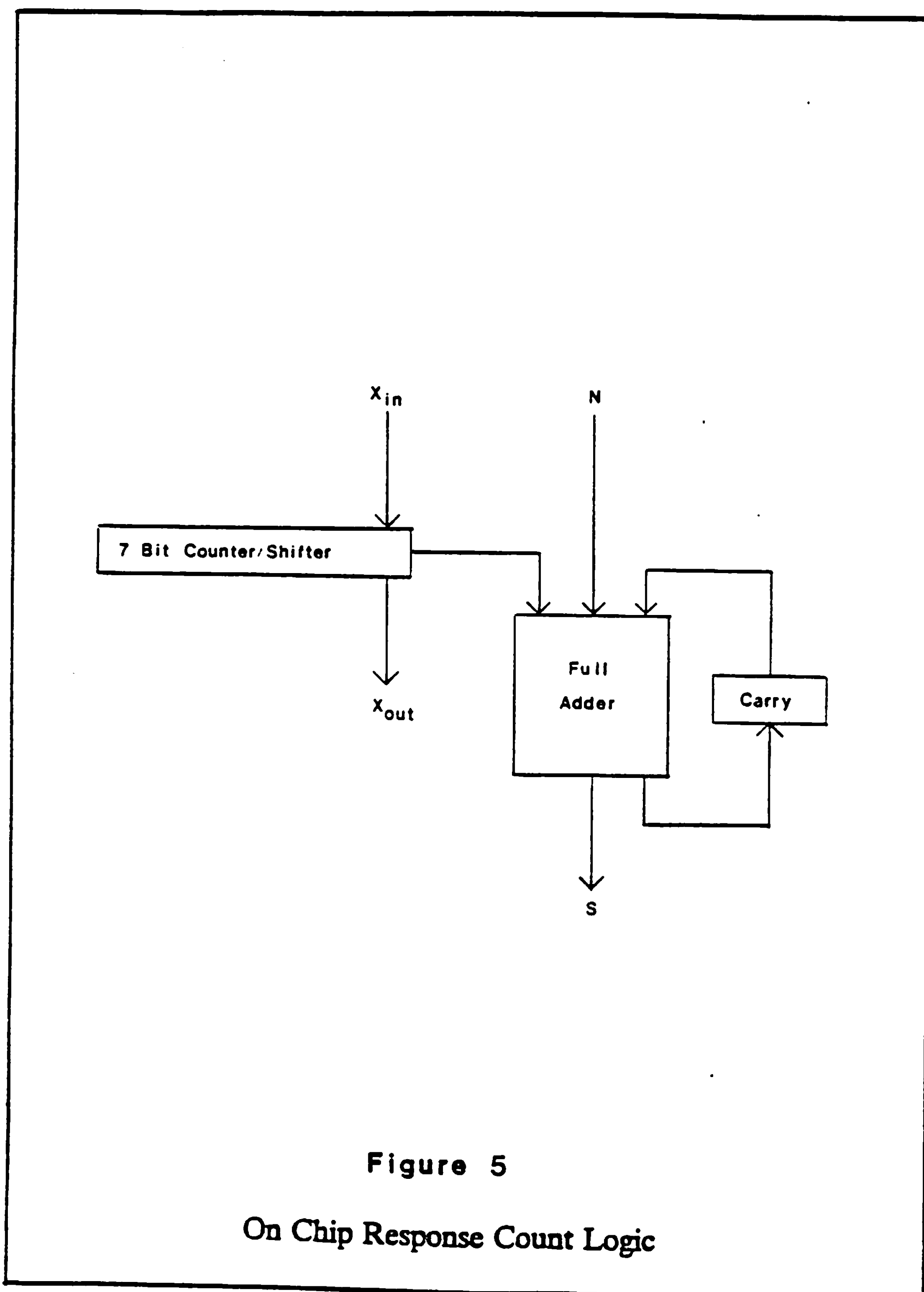


Figure 5

On Chip Response Count Logic

inputs. Another of the inputs is taken from the low order bit of the response count register. The third input comes from the chip above, via the north neighbor communication line. Thus the adders form a chain that begins with a null input at the top of the column and works its way down to the bottom, where a register is connected that will receive the column sum.

The summing proceeds as follows: The controller selects the top row of chips in the array (activating the top chip in each column via a chip select input). The adders on this row take the zero value from above, add it to the low order bit of the count register and produce a result which is output to the chip below. The response count register shifts down one bit, end off, so that the next highest order bit is now in position for the adder input. The second row of chips is then activated. The adders are triggered and the second row forms the sum of the low order bit of its own response count register and the same bit from the chip above. Meanwhile, the chip above is forming a result that represents the second order bit from its register. Now the registers in both rows of chips are shifted down and the third chip row is activated. This proceeds until the bottom row of chips has been activated and the last of its register bits has been added into the sum; at which point the columns

have been summed into registers at the bottom edge of the array. These can then be added in a similar manner proceeding right to left. As was previously pointed out, no part of the chip circuitry in this scheme is dependent upon the overall array size. Only the registers at the bottom edge of the array will reflect this, and they are not a part of the special CAAPP chips. They can be implemented with off-the-shelf components.

The array edge circuitry. This brings up one more point about the overall design of the architecture. The array of 64 circuit boards containing the CAAPP chips will have to be surrounded by another group of 32 boards that will perform a variety of functions. Among these is the control of array edge treatments, the summing of the column response counts, forming the overall some/none report back, controlling row and column selection of chips in the array and so on. These should be buildable with standard components. They raise the total circuit board count to 96, leaving four boards on which to build the controller. Of these 32 boards, 16 are likely to contain most of the required circuitry while their opposite numbers simply provide buffering and termination circuitry and a way to connect into the far side of the array.

The Processing Element Instruction Set

The instruction set that is recognized by the special purpose CAAPP integrated circuits is presented in table 5. This is divided into roughly four groups. The first group consists of operations involving memory. The second contains operations that transfer values between registers, possibly involving the ALU. The third group is used for communicating between the processing elements. The last group is made up of special instructions.

The notation used in the table deserves some explanation. Most of the operations take the form of a Pascal assignment statement. Thus there will be a destination storage element to the left of a ":=" assignment operator and a data source to the right of the assignment. A key to the source and destination abbreviations appears to the right of the memory operations. When a source is followed by an exclamation point (!) this indicates that the assignment will ignore the activity bit. Such operations are referred to as "jam transfers". If a source includes a tilde (~), then it indicates that the source will come from the off-chip (snake shift) network. The definitions of the special instructions are given below the register operations section.

Memory Operations

OP-CODE	R/W	ADDRESS

Op	R/W	FUNCTION
0	0	M:=C
0	1	A:=M
1	0	M:=B
1	1	B:=M
2	0	M:=X
2	1	X:=M
3	0	M:=Y
3	1	Y:=M
4	0	M:=A!
4	1	A:=M!
5	0	M:=B!
5	1	B:=M!

! - Transfer ignores activity
A - Activity register
B - Secondary Activity register
C - Comparand
M - Memory
X - Main tag register
Y - Secondary tag register
Z - Carry register
N - Data from North
E - Data from East
W - Data from West
S - Data from South

Register Operations

OP-CODE	DEST	SOURCE

OP CODE 6 - NORMAL
OP CODE 7 - INVERT SOURCE

Source	Destination 0	Destination 1	Destination 2	Destination 3
0	X:=A!	A:=A!	B:=A!	Y:=A!
1	X:=B	A:=B	B:=B	Y:=B
2	X:=X	A:=X	B:=X	Y:=X
3	X:=Y	A:=Y	B:=Y	Y:=Y
4	X:=$\overline{X-Y}$	A:=$\overline{X-Y}$	B:=$\overline{X-Y}$	Y:=$\overline{X-Y}$
5	X:=$\overline{X \wedge Y}$	A:=$\overline{X \wedge Y}$	B:=$\overline{X \wedge Y}$	Y:=$\overline{X \wedge Y}$
6	X:=$\overline{X \vee Y}$	A:=$\overline{X \vee Y}$	B:=$\overline{X \vee Y}$	Y:=$\overline{X \vee Y}$
7	X:=C	A:=C	B:=C	Y:=C
8	X:=N	A:=N	B:=N	Y:=N
9	X:=E	A:=E	B:=E	Y:=E
10	X:=W	A:=W	B:=W	Y:=W
11	X:=S	A:=S	B:=S	Y:=S
12	X:=N~!	A:=C!	X:=CN~!	Z:=C
13	X:=E~!	A:=B!	X:=CE~!	(6)Z:=X (7)X:=Z
14	X:=W~!	A:=X!	X:=CW~!	(6)SCRR! (7)CRCR!
15	X:=S~!	A:=Y!	X:=CS~!	(6)SCRC! (7)PANS!

~ - Zig zag shift with data transfer in and out of chip
SCRR - Shift and count responders by rows
CRCR - Clear response count register
PANS - Pipelined add North to South
SCRC - Shift and count responders by columns

Table 5

CAAPP Chip Instruction Set

There are also two boxes in the figure which show the positions of various bits in the instruction. For the memory operations, the operation code occupies the upper three bits. This is followed by the read/write select bit and then the five address bits. For the register operations, the operation code also occupies the upper three bits. Only the combinations 110 (6) and 111 (7) are allowed for these. Code 6 gives the operation specified in the table while code 7 causes the source value to be inverted before it is stored in its destination. The source and destination fields are somewhat misnamed because not all instructions adhere to the same codes.

<u>Memory operations</u>. There are 12 different memory operations. Half of these cause data to be written to memory while the others read from memory. These are moderately symmetrical with only the first pair not being so. The first memory instruction allows the controller to write a value into memory directly via the broadcast comparand bit (C). Because there is no way to reverse this one to many process, the corresponding read operation was made to load A with a memory bit under the control of activity. This serves to form the AND of the current activity pattern and a pattern stored in memory. Note that the reverse of this is also a useless instruction as writing

activity to memory under control of activity is just the same as writing with C equal to one into memory under activity control.

The remainder of the instructions are symmetric. These permit (under activity control) loading and storing B, X, and Y, and ignoring activity, to load and store A and B. This latter group of instructions is intended to allow activity masks to be directly stored and retrieved from memory. There will undoubtedly be cases where the two bits (A and B) are not sufficient to hold all of the patterns. In this case they will have to be "swapped out" to memory. By providing activity-ignoring memory transfers for both of them it is possible to save a pattern first to B and then to memory (much like stacking variables in a subroutine call), and then to fetch it back up to either A or memory.

Register operations. The register operations are divided into two groups: Those which are straight transfers and those which involve the ALU. The instruction set provides for directly transferring any of the registers A, B, X, and Y to any other of these registers. Transfers with A as the source all ignore activity for the same reason that there is no M:=A operation. There is an extra set of operations for A which allow it to take C, B, X or Y as a jam transfer source. This permits A to be transferred to

and from any of the registers and to be set or cleared by the controller via the comparand bit. Each of the registers may also take C as a data source under activity control. This permits them to receive data from the controller directly. Last among this group is a small set of operations for moving data to and from the carry bit. The X register is the only place to which the carry can be transferred, but it can receive data from both X and C. This latter is provided as a means of setting and clearing the bit.

Communications operations. There are three groups of communications operations. The first of these provides for data transfers via the on-chip (square grid) network. The data source for all of these is the X register of some neighboring cell. Only the direction from which the data is to come need be specified (N, S, E, W). The destination may be any one of the four main registers (X, Y, A, B). Cells that are on the edges of the chip array see an input of zero from beyond the edge. Because the data can be complemented, however, a transfer may also bring ones in from the edge. These operations are intended to be used in setting up activity patterns that are replicated in each of the chips. This is an integral part of the Find First operation as it provides a means of selecting individual cells within a

chip. Recall that the finest level of selection that the controller has otherwise is the row and column chip select.

The second group of communications operations provides for transfers via the off-chip network. In normal use, each of these operations will be executed eight times in a row. This results in an entire bit pattern in the X registers being moved one cell in a given direction. It was considered that these operations could be done by the chips with a single instruction, ie. that a single operation code would lead to all eight transfers being done invisibly. Had this been done, the chip would have needed some sort of state counter and sequencer. Also the controller would have had to go into a special mode that would transmit eight clock cycles without issuing a new instruction. It was felt that the inconvenience to the programmer would be minor (especially with the implementation of a few macros) compared to the additional design complexity. Note that these operations take place ignoring activity while the preceding group are under activity control. This was done because it seemed that some operations might be able to take advantage of the activiity mask with the on-chip network (the A bit could always be saved to B and set to one temporarily). On the other hand, the unusual topology of the off-chip network made it seem unlikely that there would

ever be any advantage to allowing it to be controlled by activity. Because the instruction set is rather compact, there wasn't enough room to cleanly insert a second set of these operations that could do so. Therefore the ones that would be used more frequently were chosen instead.

The final group among the communications operations also uses the off-chip network, but in a different way. Instead of shifting data in from neighboring chips and out to other chips, the data is shifted in from the broadcast comparand line and is lost at the other end. This provides another way for the controller to build a pattern within the chips. This will be used most frequently to turn on a single element in a chip. The controller shifts in a string of zeroes with a single embedded one. This string then occupies one edge of the chip array. In three or less on chip shifts that one can then be transferred to the desired position. This is because each cell in the chip array is no more than three jumps away from at least two edges.

Special operations. There are four special operations provided in order to count the responders. The first that is used in any response count is the CRCR instruction. This clears the response count register and the carry bit in the associated full adder. The second operation is the counting of the response bits. There are two ways of doing this: by

rows or by columns. Normally this won't make any difference. If a count of part of the chip is desired, however, then these two can be used to select any contiguous set of rows or columns for counting and even partial rows or columns. SCRR shifts and counts responders by rows while SCRC shifts and counts responders by columns. The last instruction on the list is the Pipelined Add North to South (PANS) which performs the chip column summing operation described above.

The CAAPP Simulator

Once the instruction set for the CAAPP was finalized, a software simulator was built to allow researchers to test applications and to aid in gathering statistics of instruction set usage. The simulator was also designed to provide timing estimates for the applications so that it would be possible to see just how much the CAAPP could speed up processing.

The instruction set of the simulator was slightly different than that of the CAAPP chip. It was meant to represent what a programmer would see in writing programs for the controller. For this reason the off chip shift instructions were condensed to a single operation, as was

the response count. A some/none test was also provided as an instruction although it was simply a logic output from the chips. The special operation codes were simply removed from the set. This meant that it would not be possible to perform the partial chip counts with the simulator, but it was felt that the cost of implementing this in software outweighed the possible uses for this feature.

The simulator follows closely the actual design of the processor except that it only implements a 64 by 64 subarray of the processing elements. This was done to reduce the time required for a simulation. The size of the array can be changed simply by redefining a few constants and recompiling the program. It was found, however, that even for a 64 by 64 array a uniprocessor is very slow at simulating the CAAPP. This is at least partly due to the fact that such a machine is not well suited to simulating bit serial operations. It is at least as much due to the fact that a uniprocessor just cannot provide the processing power required to simulate a parallel machine at any reasonable speed. If it could, then there would be no point to building parallel machines. For a test run on Conway's Game of Life, the simulator required nearly a minute of CPU time on a Cyber 175. As it turned out, many of the more interesting applications could not be run on the simulator

in any reasonable amount of time. A second simulator was built which was integrated into the UMass VISIONS system [84, 85] that was more optimized for speed than accuracy of simulation. Even this ran sufficiently slowly that many of the applications could not be tested in less than many days of processing time on a VAX-11/780. This alone points to the desperate need for parallel processors in the research community. Without actual parallel hardware on which to test applications, it is simply impossible for research to progress beyond a certain point except as pure speculation.

The first simulator was originally developed for the Cyber series of machines running NOS 2.1. The language used was the Aarhus University version of Pascal 6000 which provides facilities for separate compilation of packages. The simulator was designed as a package that would then be imported by an application program. The instructions were implemented as procedure calls and functions (for response count, some/none and find first). To use the package the application must call an initialization routine to set up the statistics gathering element of the simulator. At the conclusion of the run the program must also call a wrap up routine that prints out a series of tables giving the statistics for the application. The simulator has also been translated into VAX pascal for the VMS operating system.

The second simulator is designed as a set of user calls in the UMass VISIONS system. Because of the special nature of the implementation it is not presented here.

Circuit Board Layout

Although the circuit boards for the array were not laid out, a tentative overall design was developed for them. It was decided that they would each contain the 64 special purpose chips plus the necessary buffer circuitry for communicating off of the card. Many of the signals could be run on a backplane bus and so one edge of the card would have a standard edge connector that could be inserted into a motherboard mounted at the back of the card cage. The only wires that needed to be run in ribbon cable were the 32 that provide communications between boards. These connectors would be mounted at the opposite end of the board from the edge connector. That part of the board would stick out beyond the end of the cage to allow easy access to the ribbon cable connections. The connectors would be positioned so as to allow use of minimum length ribbon cable.

Table 6 presents the list of connections to each of the circuit boards. Most of these are self explanatory, but a

Number of Lines	Function
8	Bidirectional North Neighbor Communications
8	Bidirectional South Neighbor Communications
8	Bidirectional East Neighbor Communications
8	Bidirectional West Neighbor Communications
8	Chip Column Select
8	Chip Row Select
4	Op Code
5	Bit Address or Sub Op Code
1	Broadcast Comparand Data
1	Some/None Output
2	Clock phases
1	Power
1	Ground
63	

Table 6

Circuit Board Connections

couple deserve mention. The Some/None output is an open collector wired OR with all of the boards in a row. The special cards at the array edge provide termination for these lines and also form them into a final input to the controller. There are two Clock inputs in order to allow fine tuning of the two phases of the system clock. Presumably, once the prototype has been built and debugged, on chip phase generation circuitry can be added to eliminate one of these lines if any copies of the machine are built. Note that the total of 63 lines is well below the original design constraint. This leads to a design for the array that has only 4032 connectors.

The next section presents the design of the CAAPP chip which Dr. Foster has christened the good chip Titanic.

The Special Purpose CAAPP Integrated Circuit

Once the design of the processing elements and the instruction set had stabilized, effort turned to proving that the circuitry would fit into the space allotted for it. The only way to do this was to actually design the silicon layout. Two passes were made at this. The first was part of a seminar that was intended to teach VLSI design. The second pass, which will be presented here, was

part of a project course. The circuits to be developed in the course had an opportunity for fabrication on a line owned by a nearby company. Although this never happened, part of the circuitry was eventually fabricated via the MOSIS facility. The results of this test run are described at the end of the section.

CAAPP chip pin assignments. Before proceeding to the design of the chip itself it would be helpful to see the pin-out for it. Table 7 lists these. Mostly due to the communications multiplexing it was possible to keep the total pin count to less than 22 pins. Recall that this allows the use of an especially small package. Note that as with the circuit boards, both clock phases are input separately to allow fine tuning of the duty cycle. Because the design expects the two phases to be non-overlapping, this also provides for adjusting the gap between rising and falling edges. The arrangement of the pins is based purely on speculation as to what will give the simplest routing for wires on the circuit board. This may change entirely at a later date.

CAAPP chip overall floorplan. The overall floorplan for the production chip is shown in figure 6. The chip that was actually designed for test purposes had a much smaller body area allocated to it. For this reason it was cut back to

Pin	Function
1	West Neighbor Communications (Bidirectional)
2	Chip Select 1
3	South Neighbor Communications (Bidirectional)
4	Op Code Bit 1
5	Op Code Bit 2
6	Op Code Bit 3
7	Op Code Bit 4
8	Comparand in
9	Some/None out
10	Clock Phase 1
11	Ground
12	East Neighbor Communications (Bidirectional)
13	Chip Select 2
14	Spare (Test)
15	Address Bit 5
16	Address Bit 4
17	Address Bit 3
18	Address Bit 2
19	Address Bit 1
20	North Neighbor Communications (Bidirectional)
21	Clock Phase 2
22	Power

Table 7

CAAPP Chip Pin Assignments

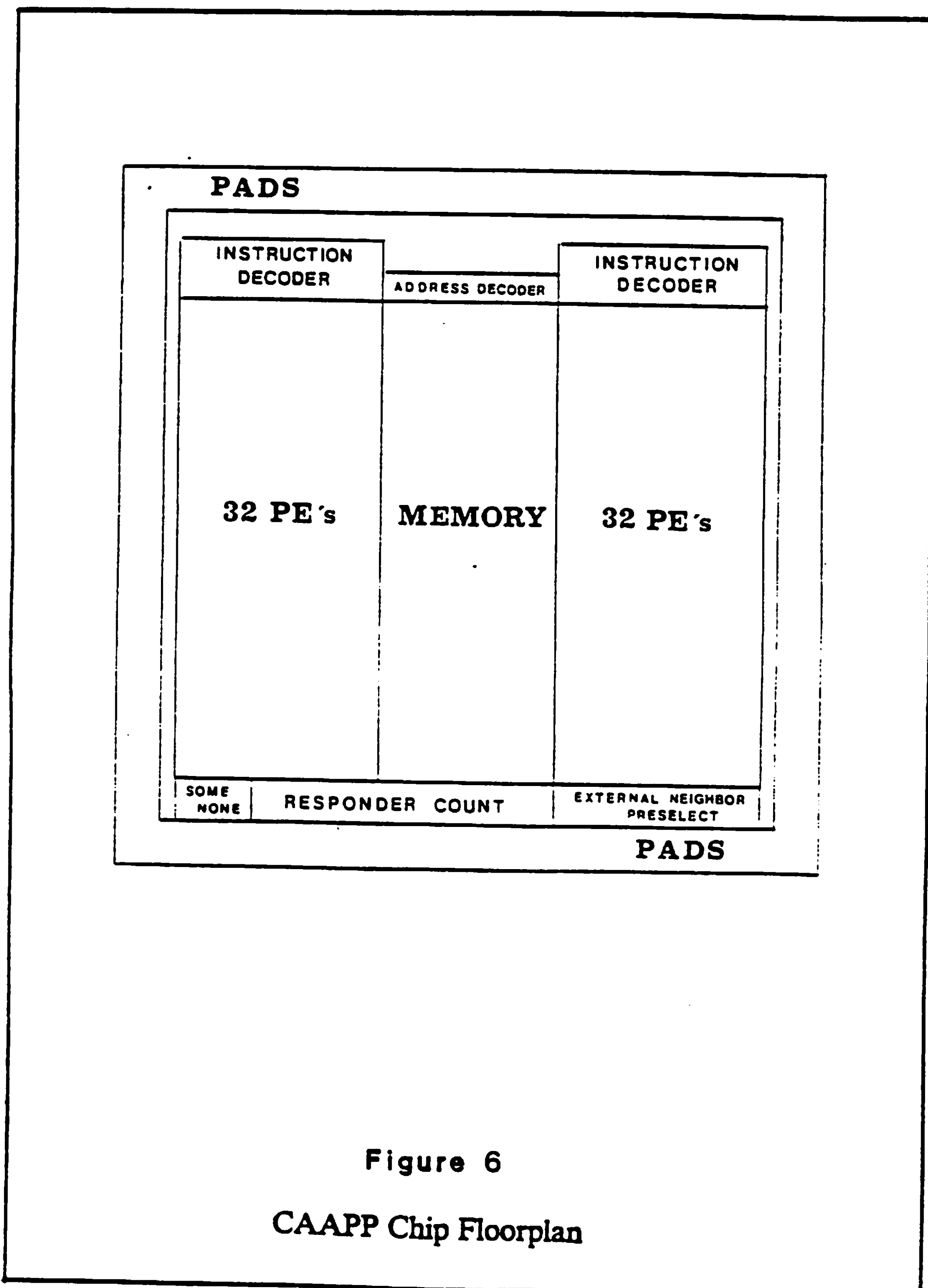


Figure 6

CAAPP Chip Floorplan

one quarter of the size of the production chip. The floorplan of the test chip is shown in figure 7. The first pass at designing the chip took the obvious approach of breaking the body area up into an eight by eight grid and attempting to fit the processors into each of these squares. It was soon realized that this was extremely wasteful of space. The problem with a design of that type is that it requires eight replications of the control and address signals across the chip. In a moderately sophisticated processing element architecture there are enough of these lines to occupy about one half of the width of the chip even if they are packed as closely as possible to one another.

A better approach to the design is to make the processing elements long and thin, stacking them one on top of the other. This allows all of the elements to share the same control lines. These run vertically through the stack of processors. The only complication that this presents is the arrangement of the communications network. The problem is how to maintain the square grid topology when the array is essentially a linear string of processors. As with most problems in topology this can be solved by imagining the grid as drawn on a sheet of very elastic rubber. Take the array by opposite corners and stretch it until the

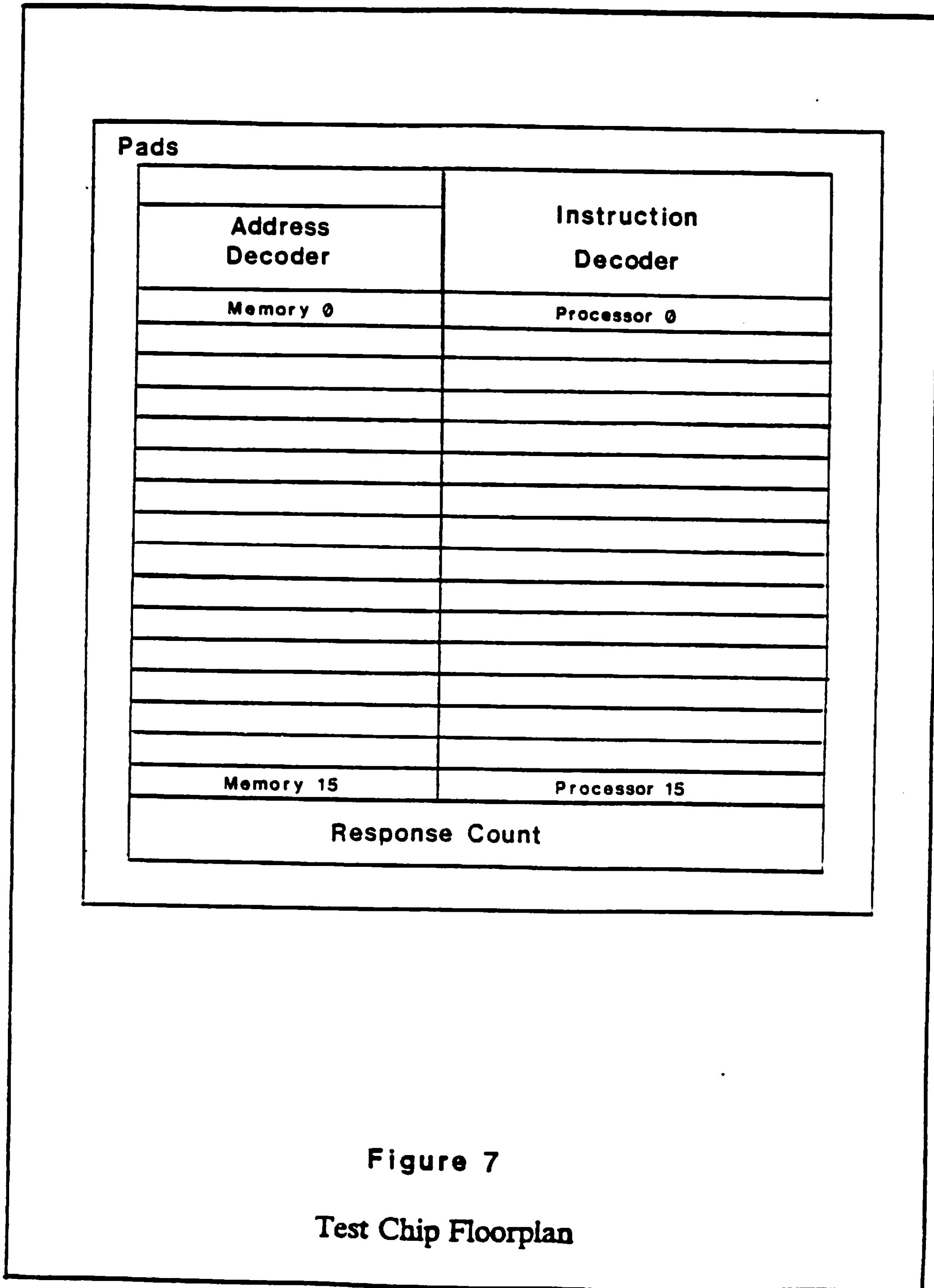


Figure 7

Test Chip Floorplan

processing elements are all in a line. The lines between them then represent the routing strategy for the communications wires on the chip. This is further detailed in the section that describes the communications networks.

In the floorplan for the production chip, the processor stack is split in two and half of it is mirror imaged to the opposite half of the chip. This gives two stacks of 32 processors. In one of those bizarre quirks of VLSI design, it turns out to be much more space efficient to simply replicate the instruction decoder a second time (also mirror imaged) than to try to route all of the wires from a single decoder to both of the processor stacks. Only one copy of the address decoder is necessary because the memory strips have a much narrower pitch than the processing elements (about half as tall) and can therefore be interleaved with strips from the opposing set of elements. This actually provides a much more dense packing of memory cells than any of the alternatives that were considered (such as making the memory cells taller and narrower to more closely match the processor pitch).

The 16 cell test chip was designed specifically as a quadrant of the full production chip in order to show that the latter could also be built. The design rules that were used for the test chip development were essentially a

slightly augmented Mead and Conway [79] set. The process provided a single layer of metal, poly, diffusion and buried contacts. There were two types of ion implant. Lambda was three microns. Minimum metal width was two lambda, with two lambda separation. Diffusion and poly were both two lambda minimum width with one lambda separation between like wires and one half lambda between each other. The minimum gate region was two lambda by two lambda.

Processing element floorplan. Figure 8 shows the floorplan of an individual processing element. Because the element is so long and narrow, it is broken into two pieces in the figure. The top section appears to the right of the bottom section in the actual chip. Again, the reason for the long narrow layout is to maximize the sharing of control signals among the processing elements. The organization of the subparts of the element correspond to the logical flow of data through the architecture. Data begins at the right in one of the five registers. (To the right of these is the some/none circuit which takes its input from the X register.) These values pass through the ALU which generates the three functions (NAND, NOR, NSUM) continuously in addition to passing along the values of the registers. One (or possibly none if a non-register or ALU instruction is being processed) of these is selected by the ALU function

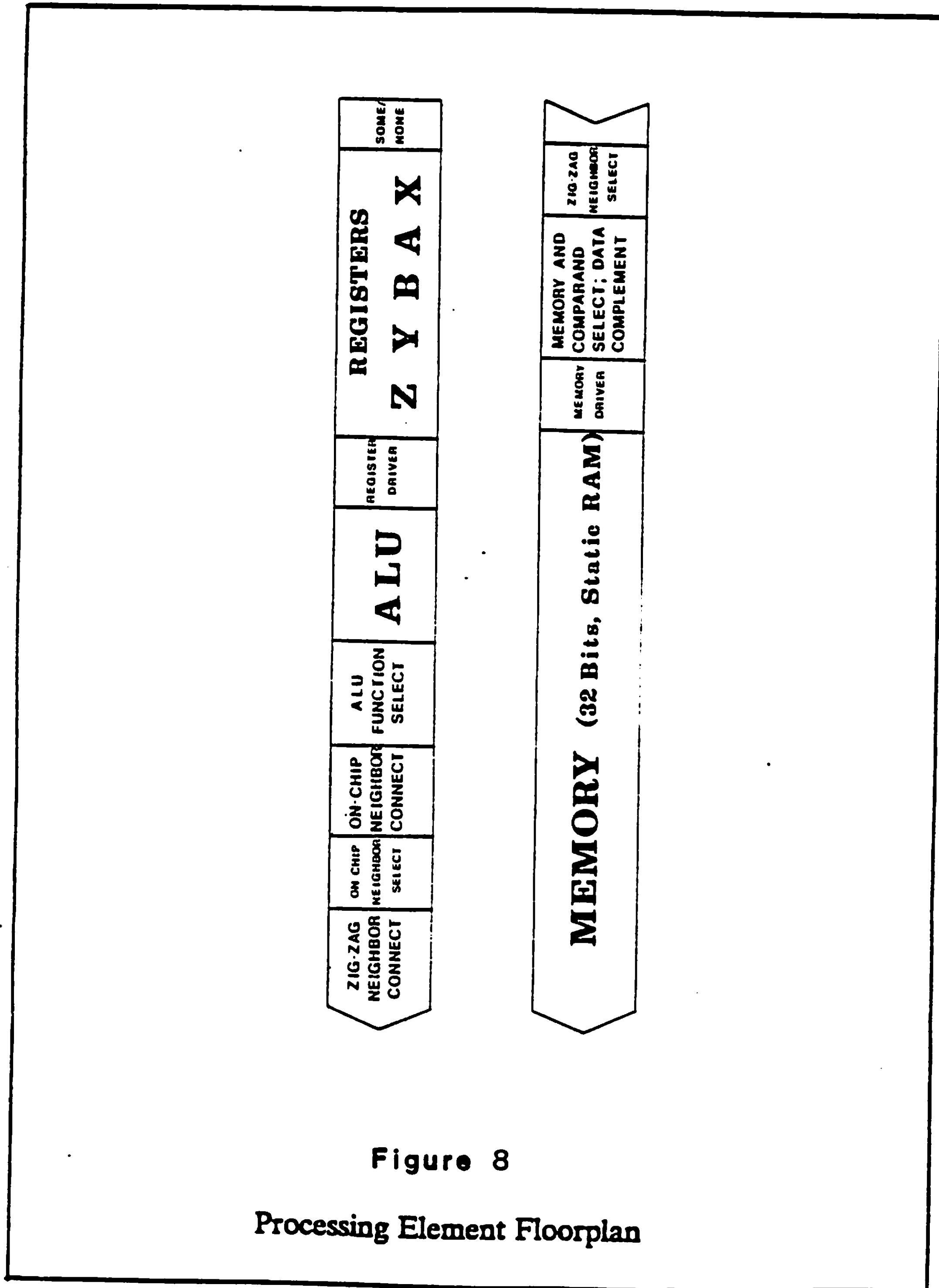


Figure 8

Processing Element Floorplan

selector. The output of the X register is tapped at this point and distributed to the neighboring cells via the two communications networks. Conversely, their X values are also available in this area, one (or possibly none) of which may be selected by the two neighbor selectors. Finally, the broadcast data bit or the output of memory may be selected. Only one of these will actually be selected, with all of the other data sources excluded. That final value may be complemented as it passes through the XOR gate in the "data complement" section. To the left of this is the memory driver and the memory cells. Note that the memory is at this end of the floorplan because it is not needed prior to the final selection stage in the data flow path through the architecture. The final result branches at this point and can either go left to the memory or right for the long trip back to the registers where it may be stored.

<u>Support circuitry</u>. Here begins the examination of the actual circuitry and layout that went into the chip design. In the first part of this examination the circuitry that is not a part of the processing elements will be presented. This includes the instruction decoder, address decoder, drivers for signal distribution, some/none generation, external neighbor preselect and response count.

Beginning with the instruction and address decoders it

will first be necessary to examine the circuits which provide the input signals to these. The decoders require both true and complemented forms of the the inputs, however, these are only provided in true form as they enter the chip. The signals in question are the five address lines, the four operation code lines and the two chip enable lines. Also, because of the long wires which these must drive and the fast switching time required for the signals, a high power driver circuit is necessary. This is provided by the dual output super buffer shown in the circuit diagram of figure 9 and the silicon layout of figure 10. A strip of these is inserted into the decoders roughly every 32 stages in order to maintain signal levels. As there are 68 stages in the instruction decoder and 32 stages in the address decoder this works out to three sets of super buffers overall. Figure 11 shows how these super buffers fit together to match the pitch of the decoder circuits.

The decoders themselves are essentially the AND plane of a programmed logic array. Because the encoding of the instructions was arranged to simplify the decoding process (by grouping similar instructions together), it was possible to almost eliminate the OR plane. The NOR-NOR form was used in the implementation but many of the signals required only the first stage of this while the others required only a

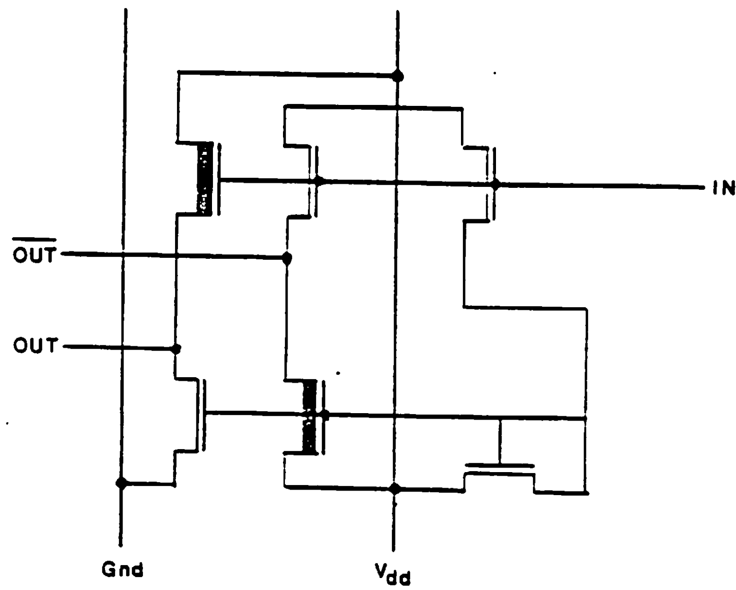


Figure 9

Super Buffer Circuit

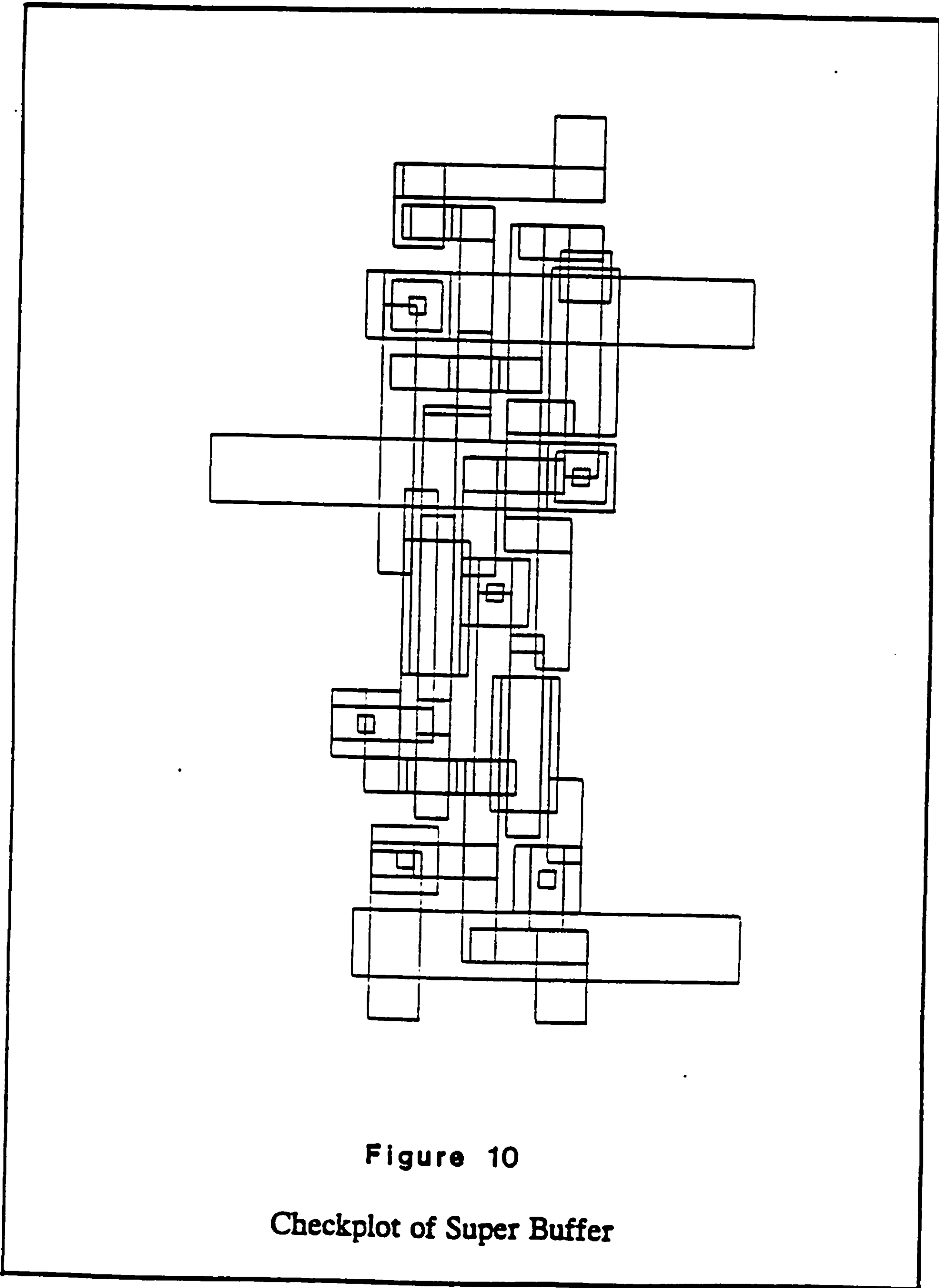


Figure 10

Checkplot of Super Buffer

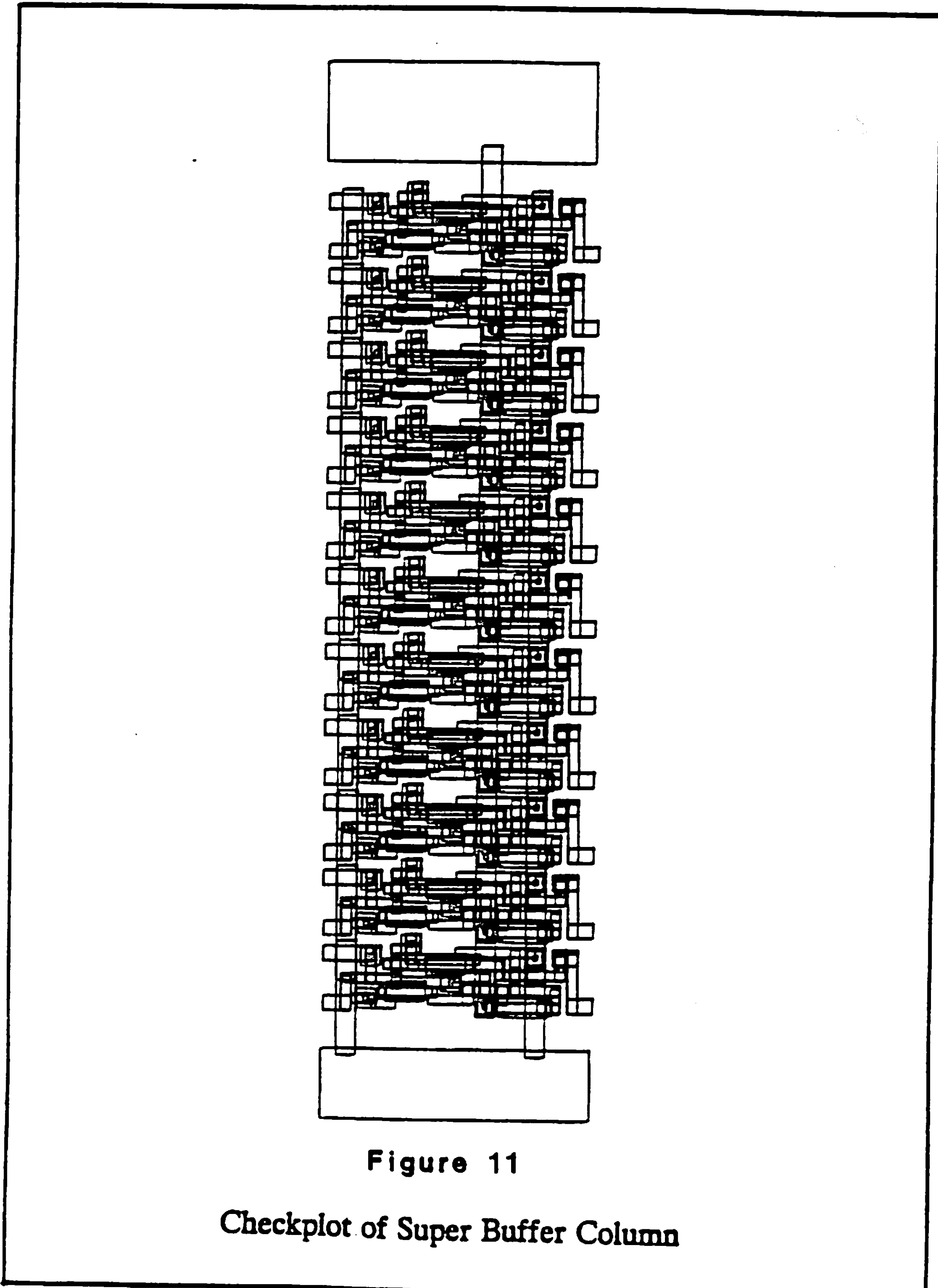


Figure 11

Checkplot of Super Buffer Column

simple NOR gate with between one and four inputs to provide the final output (all of the outputs were, of course, inverted). The address decoder did not require any second stage NOR circuitry. It's output did, however, have to be gated by an appropriate write timing signal. The circuit for generating this is shown in figure 12. The actual layout of the timing circuitry can be seen at the bottom, center in figure 13 which shows the section of the layout where the address and instruction decoders meet. This also gives a good view of sample segments of the decoders, final signal formation circuits and how the super buffers fit into the scheme of things.

The decoders themselves were designed modularly. A generic decoder cell was first specified for each of the two decoders and then replicated the appropriate number of times. The functions for the decoders were then generated as a set of patches that were layed over this to form the appropriate pattern of true and complement pass transistors. These would pull the NOR circuit to ground if the input pattern matched. The patches for the address decoder were generated by a binary count routine while those for the instruction decoder were table driven. Table 8 shows the arrangement of the patches. The top section of the table covers the right half of the decoder while the

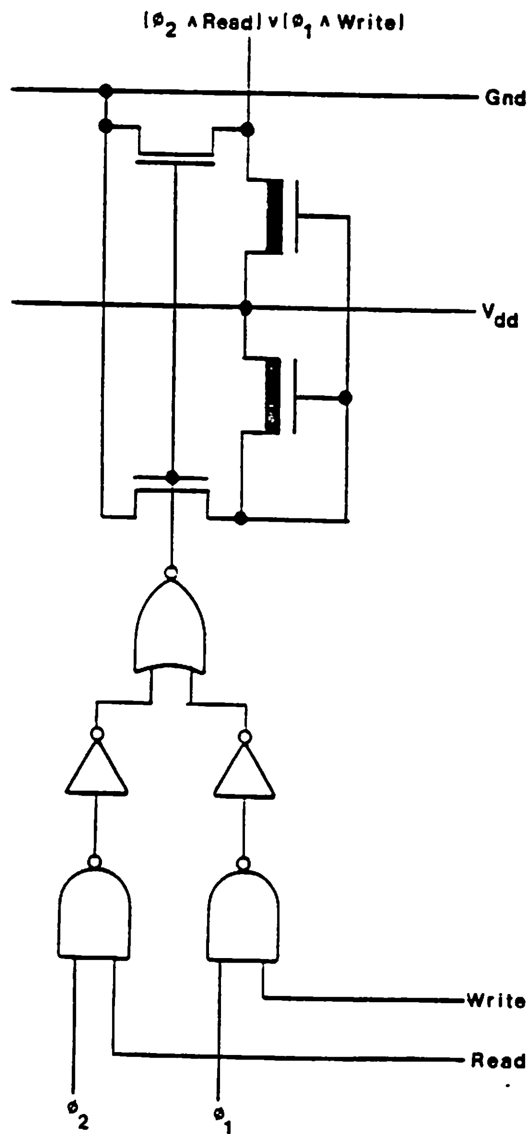


Figure 12

Address Select Timing Generation Circuit

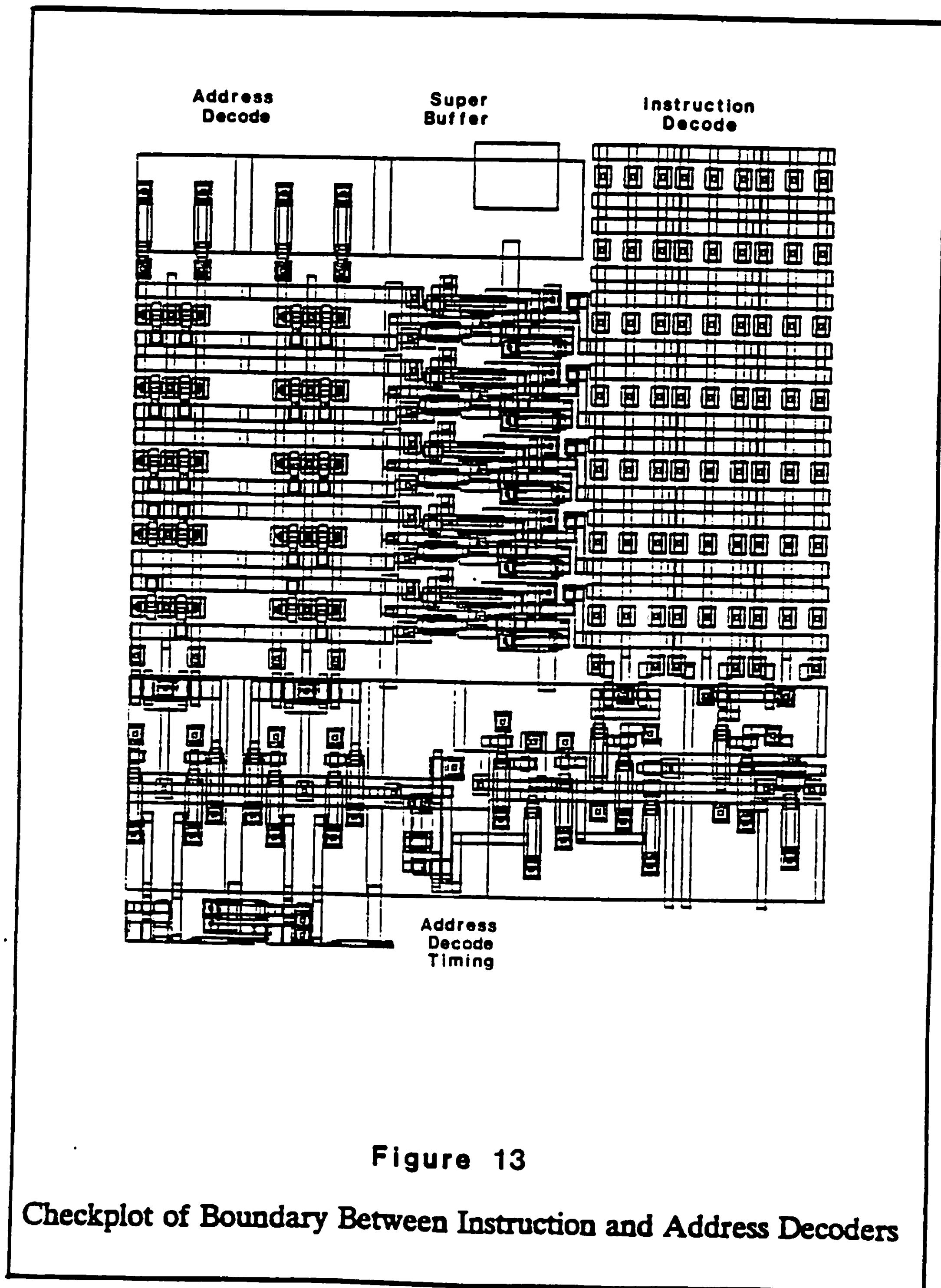


Figure 13

Checkplot of Boundary Between Instruction and Address Decoders

	Memory Write	Complement	Memory Select	Comparand Sel	S~ Select	W~ Select	E~ Select	N~ Select	S Select	W Select	E Select	N Select	Y Select	B Select	Nand Select	A Select	Nor Select	X Select	Sum Select
CE_1	11	1	11	111	11	1	11	1	1	1	1	1	111	111	1	11	1	1111	1
CE_2	11	1	11	111	11	1	11	1	1	1	1	1	111	111	1	11	1	1111	1
OP_4	01	1	01	110	11	1	11	1	1	1	1	1	110	11X	1	11	1	1110	1
OP_3	X0	1	X0	110	11	1	11	1	1	1	1	1	111	110	1	10	1	1111	1
OP_2	XX	1	XX	XX0	X0	X	X0	X	X	X	X	X	XX1	XX1	X	X0	X	0XX0	X
OP_1	00	X	11	XX0	X1	X	X1	X	X	X	X	X	0X0	0X0	X	X0	X	10X0	X
A_5	XX	X	XX	1XX	01	0	01	0	X	X	X	X	1XX	1XX	X	XX	X	11XX	X
A_4	XX	X	XX	10X	11	1	11	1	1	1	1	1	10X	10X	0	0X	0	110X	0
A_3	XX	X	XX	11X	11	1	11	1	0	0	0	0	10X	10X	1	0X	1	110X	1
A_2	XX	X	XX	01X	11	1	01	0	1	1	0	0	11X	00X	0	0X	1	011X	0
A_1	XX	X	XX	01X	11	0	10	0	1	0	1	0	11X	11X	1	0X	0	100X	0

	Ignore A	Data $\wedge \phi_2$	Z Select $\wedge \phi_2$	Z ← Carry	Z ← Data	Y ← Data	B ← Data	A ← Data	X ← Data	C~ Shift	SCRR	SCRC	CRCR	PANS	NE Tri-State	SW Tri-State
CE_1	11111	1111111	1	1	11	111	111	11	1111	1	1	1	1	1	1	1
CE_2	11111	1111111	1	1	11	111	111	11	1111	1	1	1	1	1	1	1
OP_4	11111	0X11111	1	1	11	110	11X	1X	1110	1	1	1	1	1	1	1
OP_3	01111	X011111	1	1	11	111	110	10	1111	1	1	1	1	1	1	1
OP_2	XXXXX	XXXXXX0	1	X	X0	XX1	XX1	X0	1XX0	X	0	0	1	1	X	X
OP_1	X10XX	110X111	1	X	11	111	111	01	1101	1	1	1	1	1	0	0
A_5	X0XXX	XXX0111	1	X	11	11X	00X	1X	100X	0	1	1	1	1	0	0
A_4	X1110	XXXXX01	1	0	11	0XX	0XX	XX	11XX	1	1	1	1	1	1	1
A_3	X1110	XXXX011	1	1	11	10X	10X	XX	11XX	1	1	1	1	1	1	1
A_2	X0010	XXXXXXX	0	0	00	XXX	XXX	XX	0XXX	X	1	1	1	1	0	1
A_1	XXXX0	XXXXXXX	1	0	01	XXX	XXX	XX	1XXX	X	0	1	0	1	X	X

Table 8

Instruction Decoder Patch Table

bottom section covers the left half. The positions in the table marked "X" have no patches as the state of that instruction line does not matter for that stage. The ones indicate patches to the true input and the zeroes indicate patches to the complemented input line. In the figure showing the layout, it can be seen how the patches have been inserted into the address decoder while the instruction decoder has not yet had the patches added. Figure 14 shows the logic diagram for a sample section of the address decoder while figure 15 shows the stick diagram for the same section. The logic for a sample section of the instruction decoder is shown in figure 16. The layout of the entire decoder section will not be shown here because the resolution required to see any detail in it would produce a figure several feet wide.

The outputs of the decoders must, like their inputs, also be buffered. Because the design was prepared for the full strip of 32 processing elements, and it was felt that these would present a greater drain (and certainly a longer wire length) than the decoder stages, a special high power buffer was designed to provide better switching characteristics. This was essentially a super buffer with a 2:24 gate ratio output transistor. In order to conserve space, this transistor was implemented with a circular gate

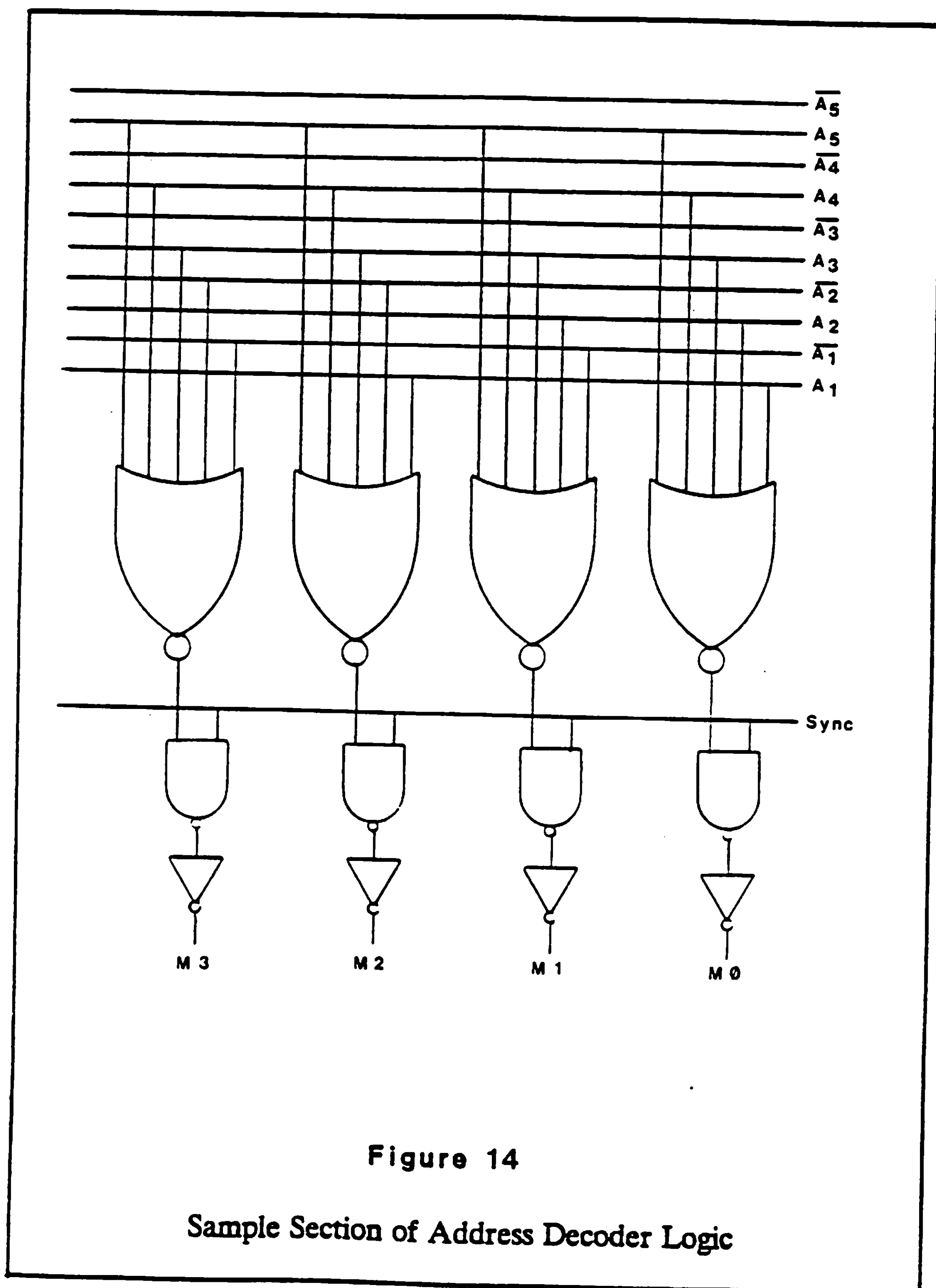


Figure 14

Sample Section of Address Decoder Logic

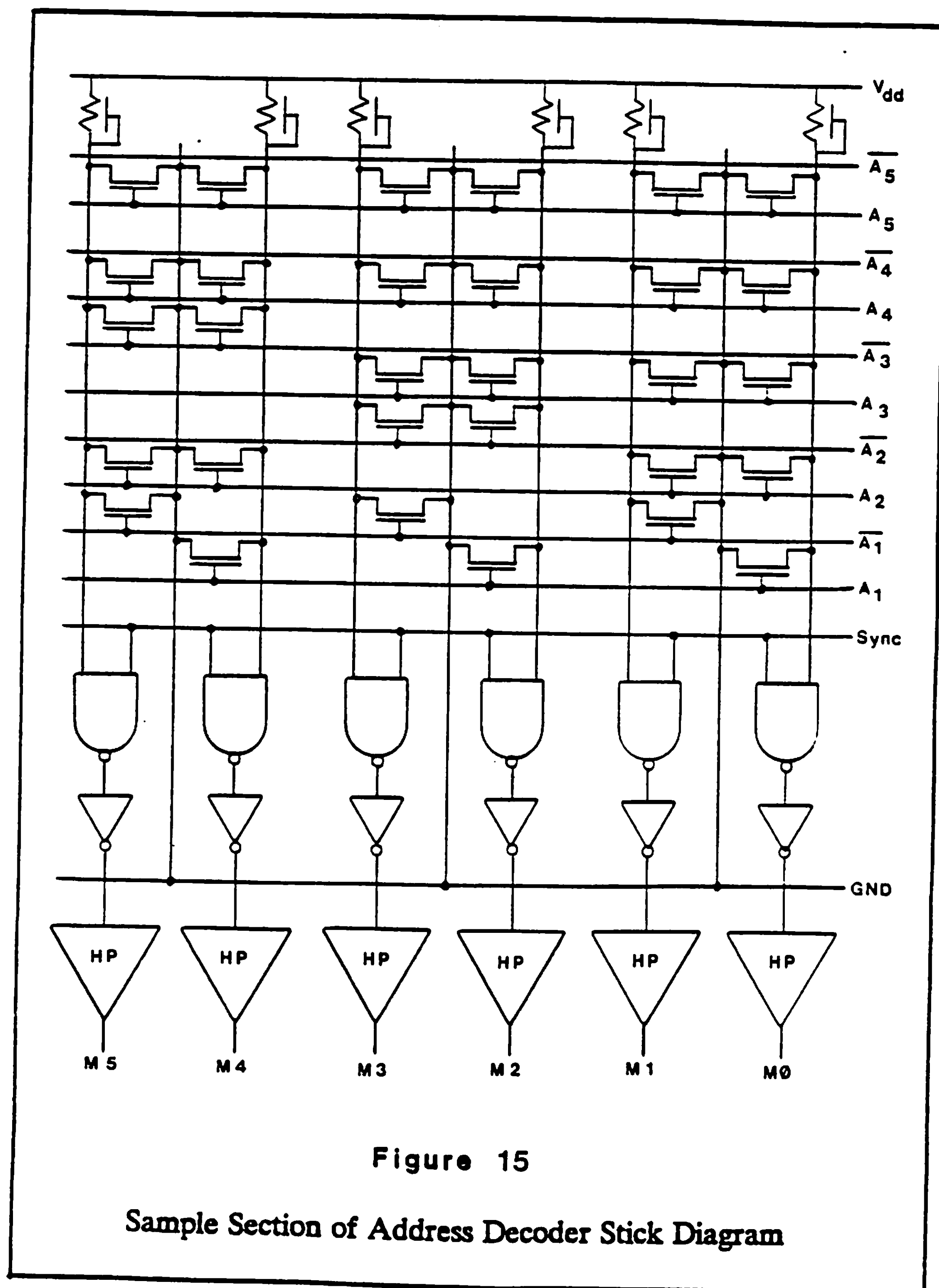


Figure 15

Sample Section of Address Decoder Stick Diagram

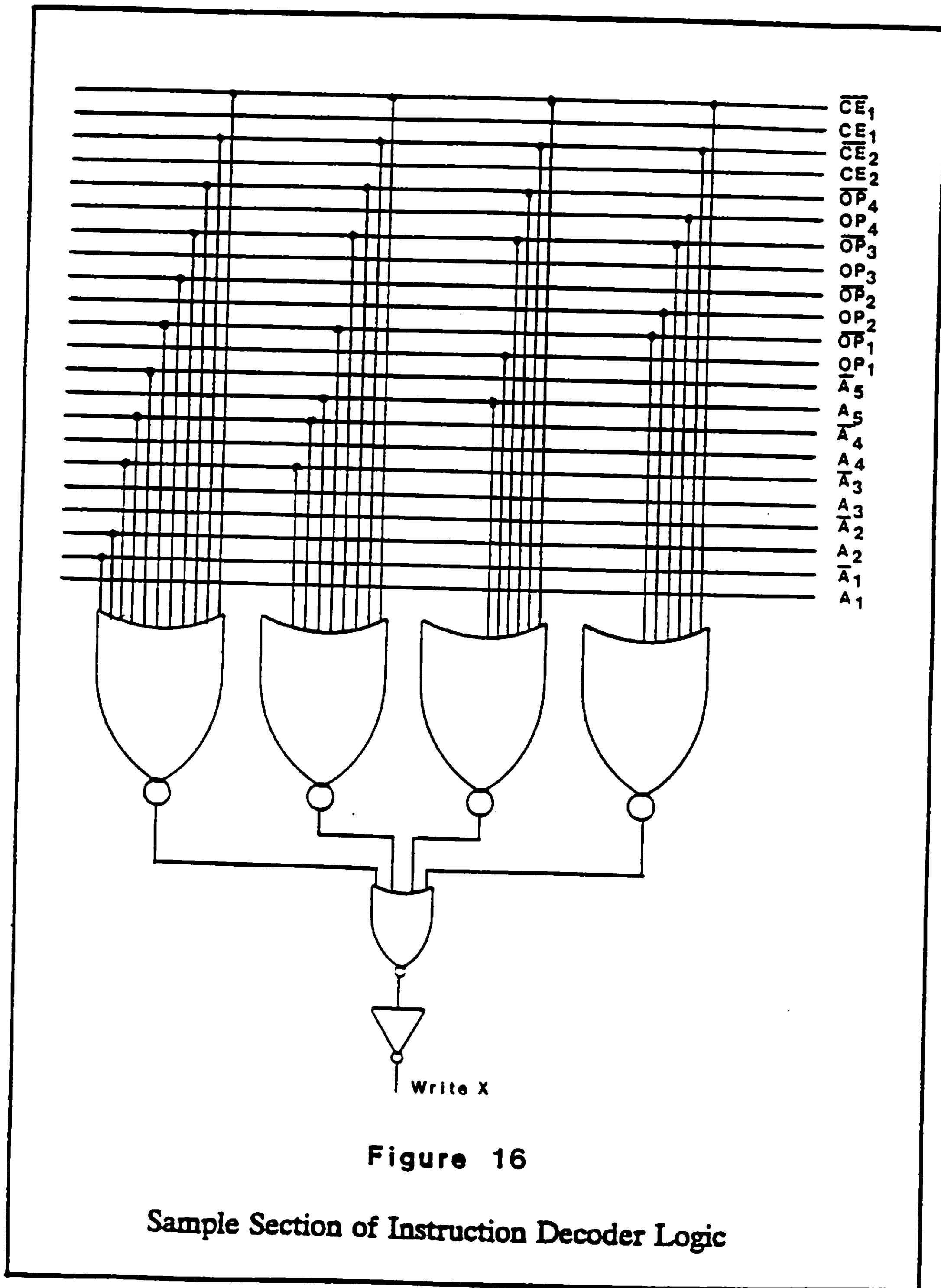


Figure 16

Sample Section of Instruction Decoder Logic

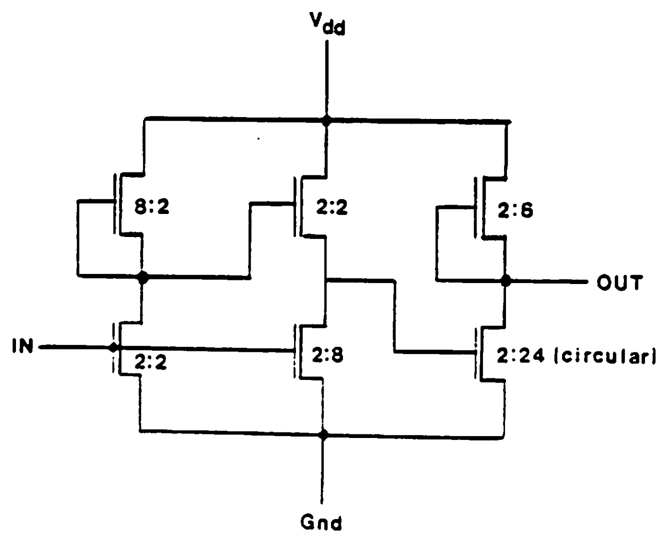


Figure 17

High Power Buffer Circuit

region. The circuit diagram for this is shown in figure 17 and the VLSI layout in figure 18.

The remaining support circuits deal with input and output to the chip via the communication lines and the feedback to the controller. The some/none circuit is very simple to implement in NMOS. The desired effect is the OR of all of the response bits. This is implemented as a a big NOR circuit that runs the length of the processing element stack, at the edge furthest from the memory. At the end of the NOR chain, an inverter converts the output to the proper logic sense. A stick diagram for this is shown in figure 19.

The off chip neighbor network is used for three separate functions. The first of these is the shifting of response patterns between chips. In addition the network is used to shift in values broadcast by the controller to establish patterns (with bits shifted off the opposite end) and as the basis of the circular shift that performs the response count. In order to fit all three of these functions into the one network, the ends of the net are brought out to a common point and there interfaced to the outside world via some logic that sets up the proper configuration for whatever instruction is performed. A stick diagram for this is presented in figure 20 and the layout is shown in figure

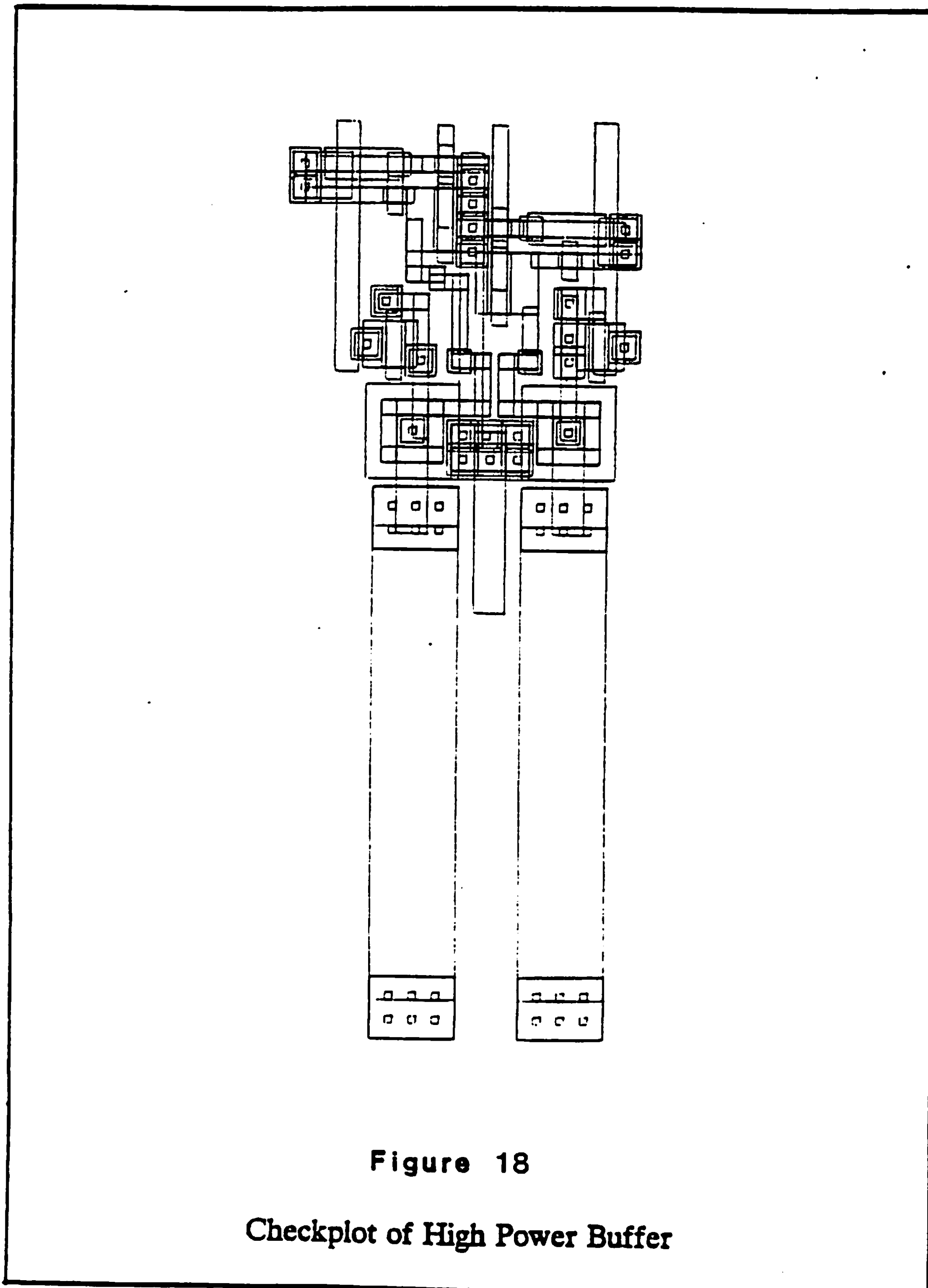


Figure 18

Checkplot of High Power Buffer

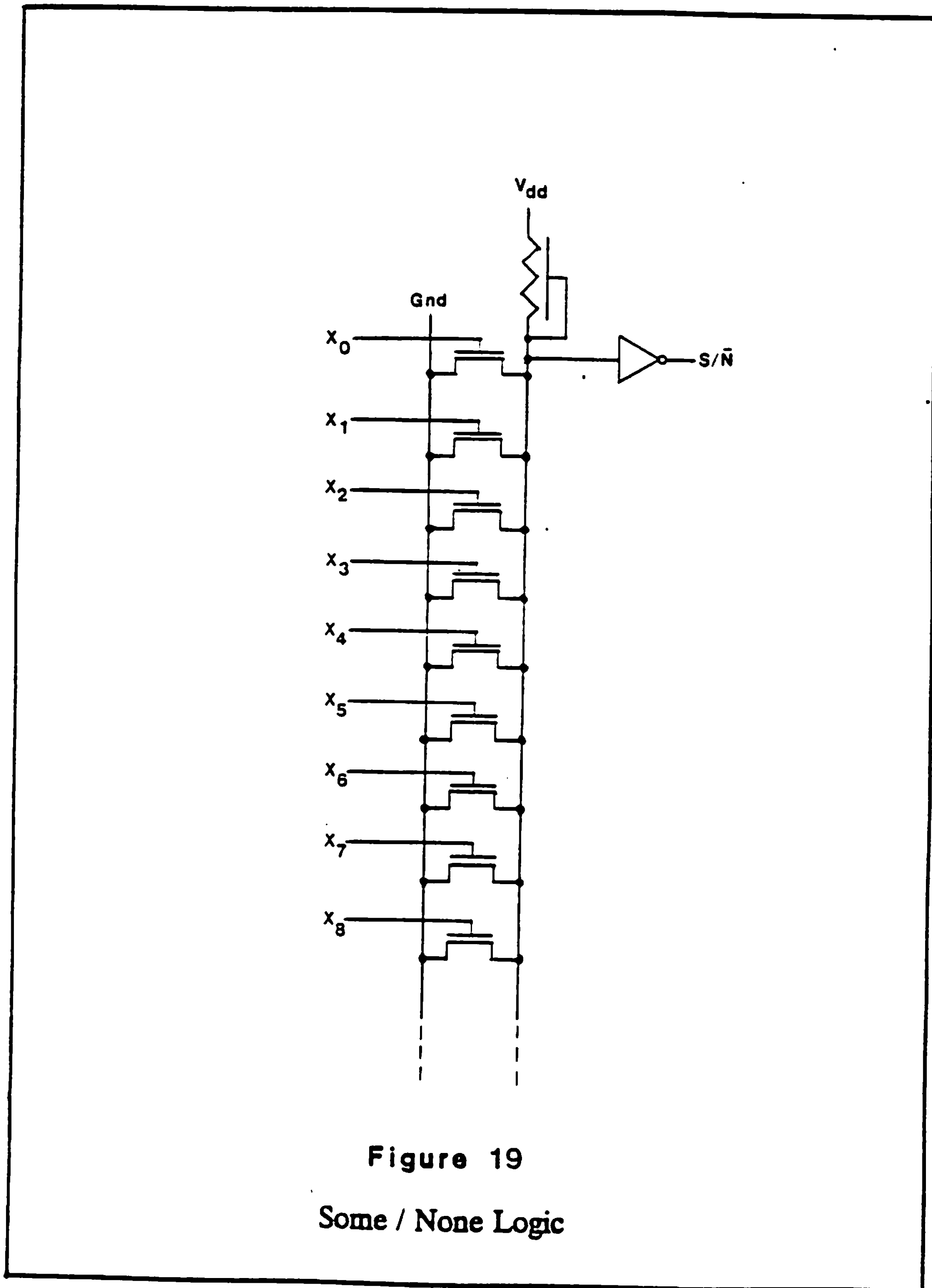


Figure 19

Some / None Logic

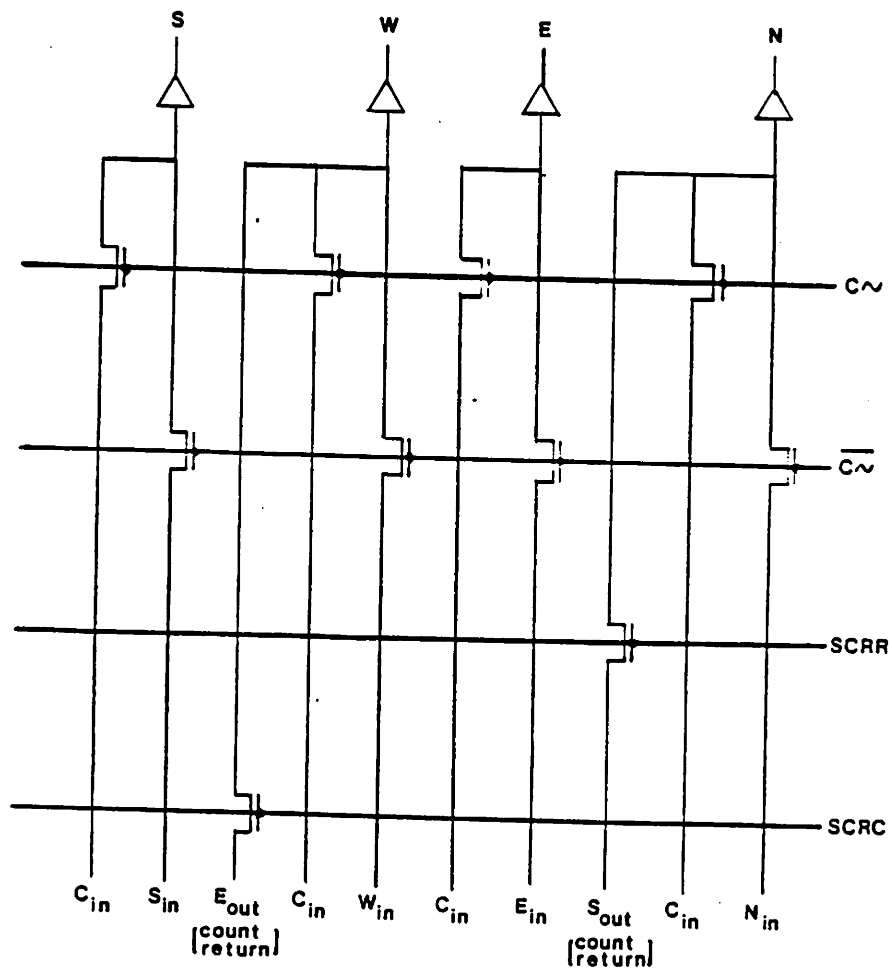


Figure 20

External Neighbor Preselect Circuit

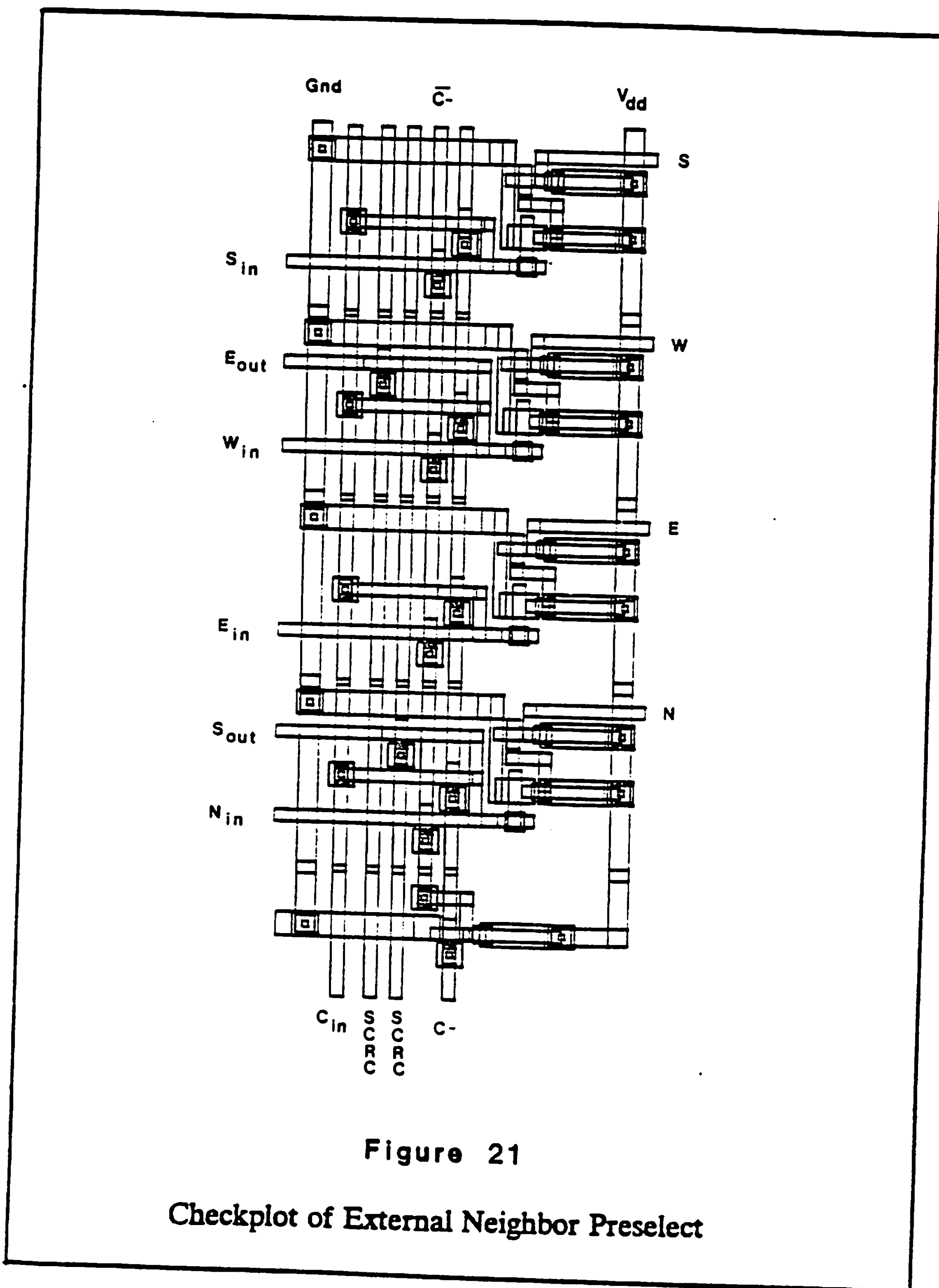


Figure 21

Checkplot of External Neighbor Preselect

21.

The response count circuit was neither designed nor laid out due to time constraints in the test chip development project. At the point where the counter would tap the circular shift registers a line was brought out to one of the test chip pads so that an external counter could be substituted. Although counters are not trivial to design there is sufficient space left that it should be possible to fit it in. Much more concern was given to the design of the processing elements as each extra square lambda of space in those translates into 64 square lambda in the final design (16 in the test chip). It was assumed that if those could be kept small enough then there would be sufficient empty space left to implement the one-of-a-kind circuits.

Processing element circuitry. The processing elements are divided into five main sections. These are the memory (and its driver circuits), the registers, the ALU and its function selector, the communications networks and their selectors, and the final data source selection and complement section. Each of the cells is identical except for the communications sections. These vary depending upon where in the topology the cell appears. The common elements of the processors were all defined with standard positions for the connections to the communications sections. These

were then replicated 16 times (32 for the full size chip) with the correct sequence of communication segments patched into place. Fortunately there was also some repetitiveness in the communication sections that could be taken advantage of.

The memory and its driver circuit are very simple in comparison to the designs used in standard memory chips. There are two reasons for this. The first is that standard cells are not designed to be used in this way. In a standard RAM, the cells are typically arranged so that there are orthogonal selection lines which select a single bit. The cells also share read and write lines because only one is selected at a time. The CAAPP memory, however, selects all of the cells in a column of bits for parallel reading or writing with their own processing elements. The second reason is that the design team did not have much prior experience in memory design and little access to the tools necessary to develop a more sophisticated design. Memory design is still essentially a black art that pushes process parameters to their limits to squeeze out the last square micron of space from the cells. This necessitates construction of sophisticated sense amplifiers to detect and amplify the output of the cells and so on. It was felt that the most realistic approach for the design, in the absence

of proper tools, was to overbuild to be safe. The advice that was most frequently given in the design was, "If you're not sure, drive the daylights out of it." The result might waste some power unnecessarily but it would at least stand a chance of working.

The memory cell itself was a very robust six transistor design. The stick diagram of a single cell with the memory driver circuit is shown in figure 22. This is a standard static memory configuration of cross-coupled inverters with inputs for writing zero or one. Each cell gets Vdd and ground, plus its select signal from vertical busses that pass through every cell in the same position. The design was able to conserve some space by combining pairs of cells, mirror imaged, to allow sharing of power and ground. The checkplot for a pair of cells is shown in figure 23. Each pair of cells also overlaps to share a vertical wire. One of the biggest problems with this design is that it forces the data to run on long lengths of diffusion. This is because the vertical signals must run in metal, preventing the horizontal lines from using that layer. Of all of the aspects of the chip, the fact that the memory data lines had to be run in diffusion was the most worrisome, so much so that when the opportunity later arose, the circuit was simulated and test fabricated independently. The results of

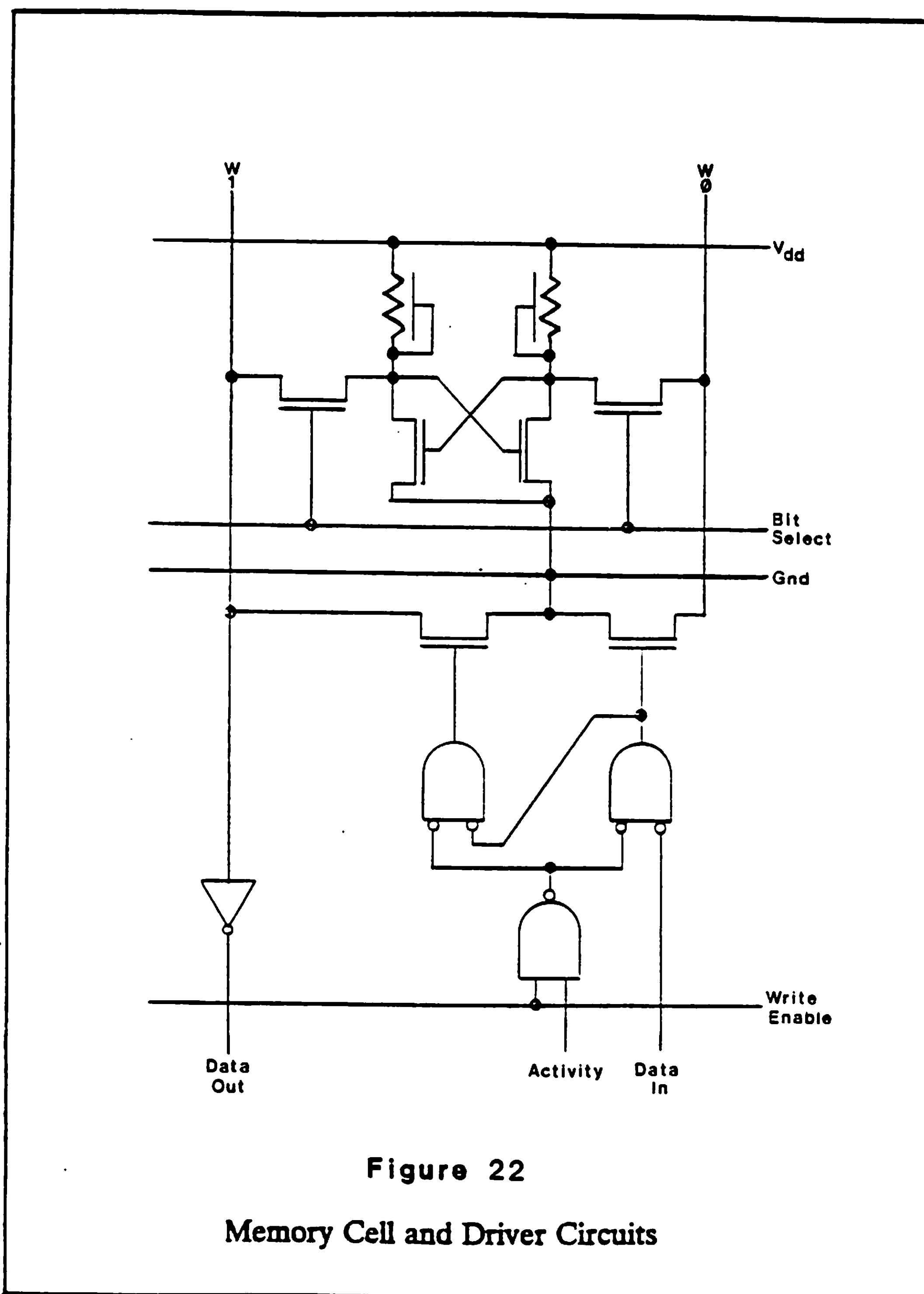


Figure 22

Memory Cell and Driver Circuits

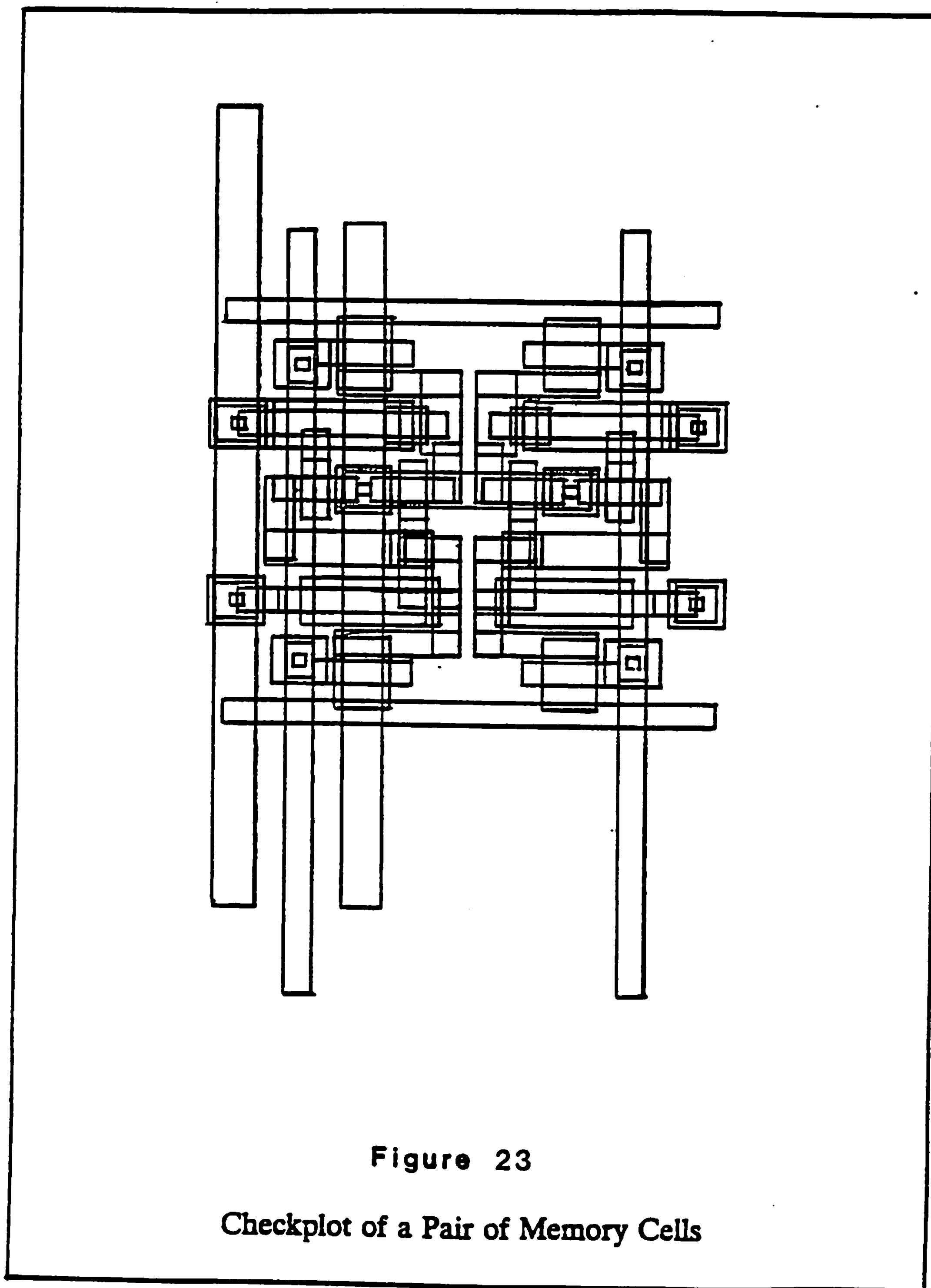


Figure 23

Checkplot of a Pair of Memory Cells

this testing are presented in the last section of this chapter.

The memory driver circuit is simply the output of a logic network that splits the data line into separate write zero and write one signals. This also combines the activity signal to inhibit writing when the cell is inactive. The read circuitry is simply an inverter. The checkplot for this is shown in figure 24.

Moving from left to right across the processing element, the next section is the final data select and the complement data circuit. This is essentially the final two stages of the data selector circuit, followed by a level restoring buffer that enters one side of an exclusive OR gate. The other side of the XOR is taken from a line that runs vertically from the instruction decoder. The selector circuits are fairly standard. Signals come in vertically (in this case, memory and comparand), are tapped off and gated by a pass transistor. The output of the pass transistor is to a common data bus that is shared by all of the other selectors. Avoidance of bus contention is enforced by the instruction decoder which will allow only one pass transistor among all of the selectors to be turned on at any time. The circuit diagram for this section is shown in figure 25, and a checkplot appears in figure 26.

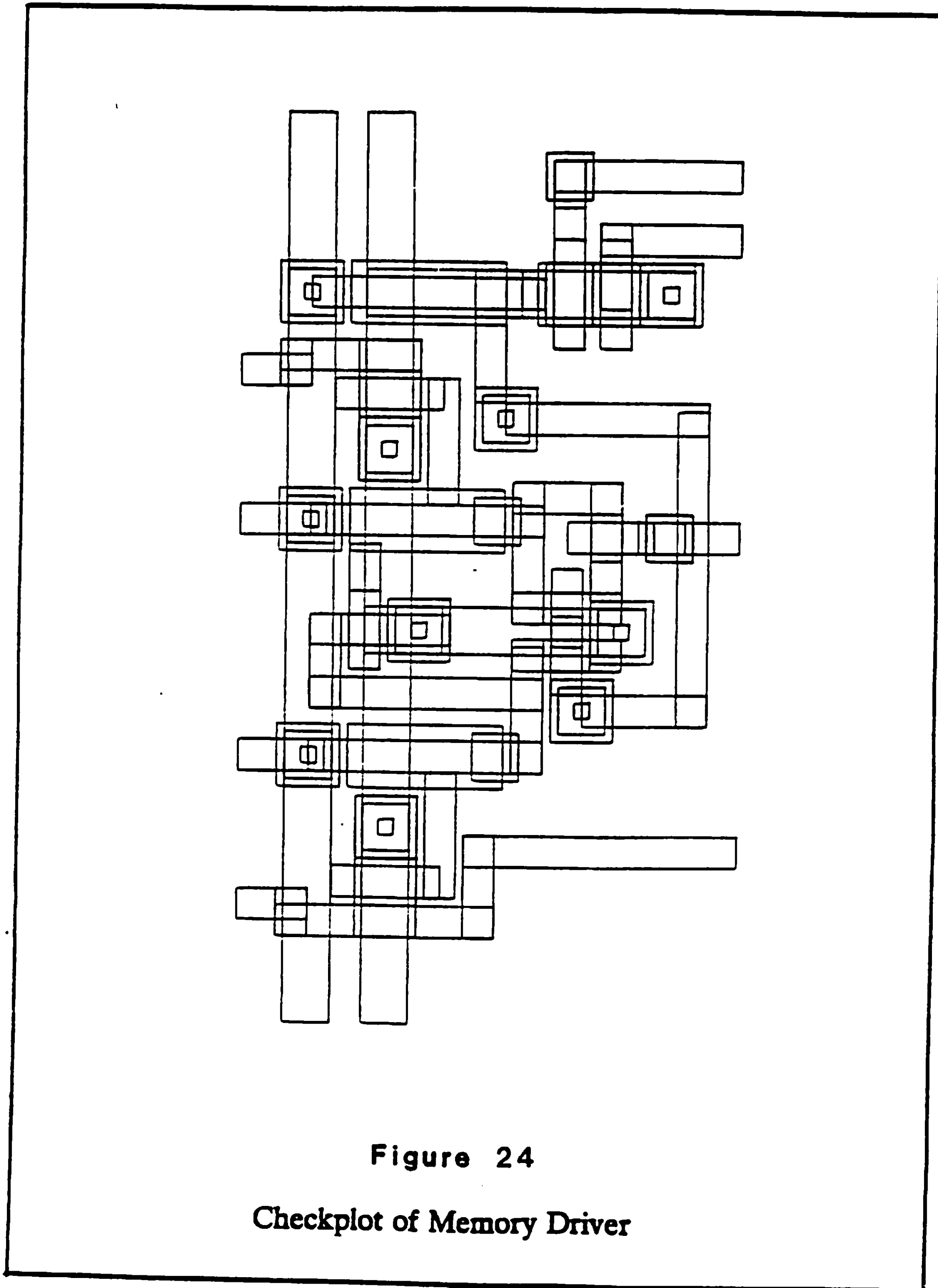


Figure 24

Checkplot of Memory Driver

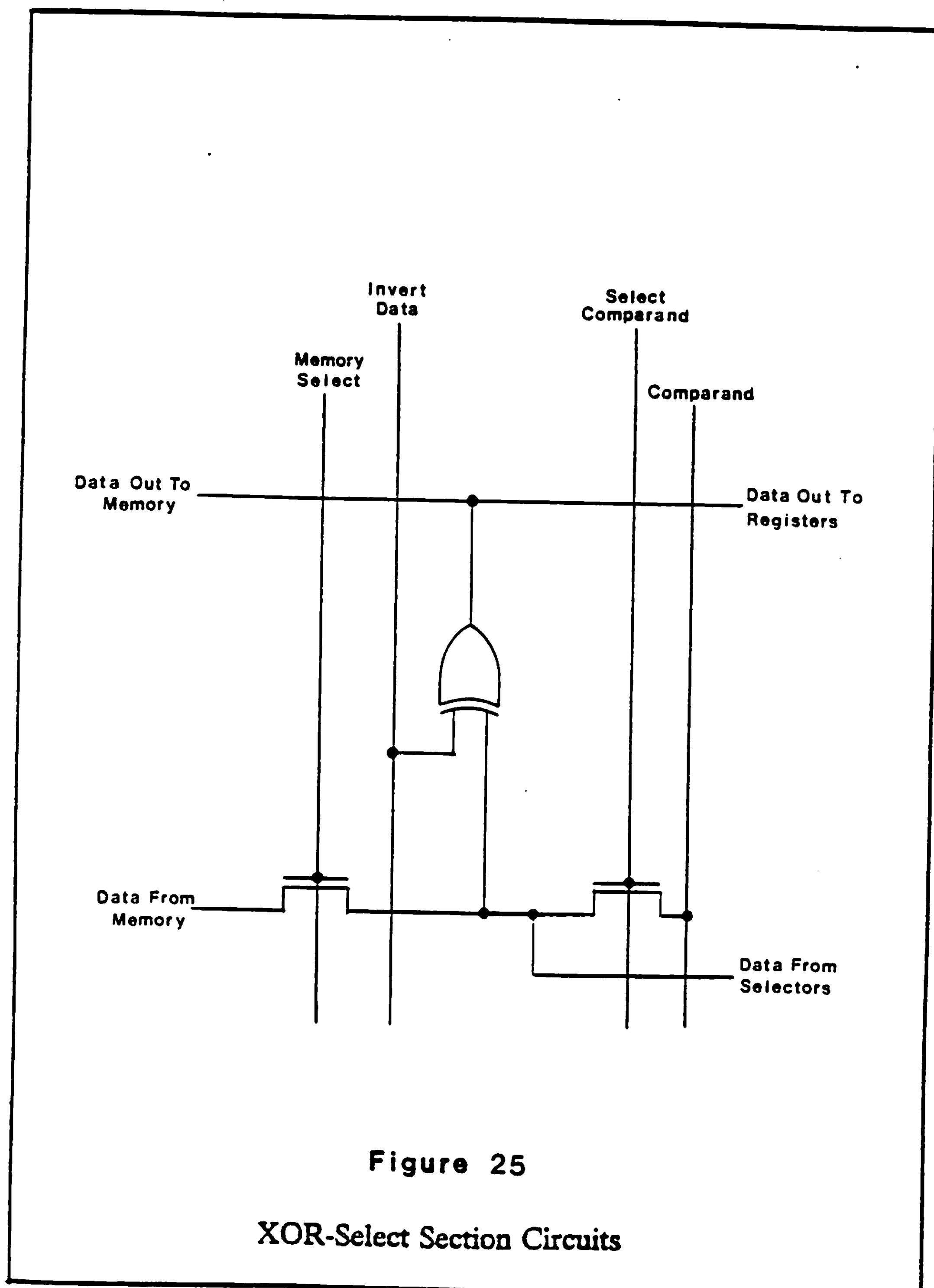


Figure 25

XOR-Select Section Circuits

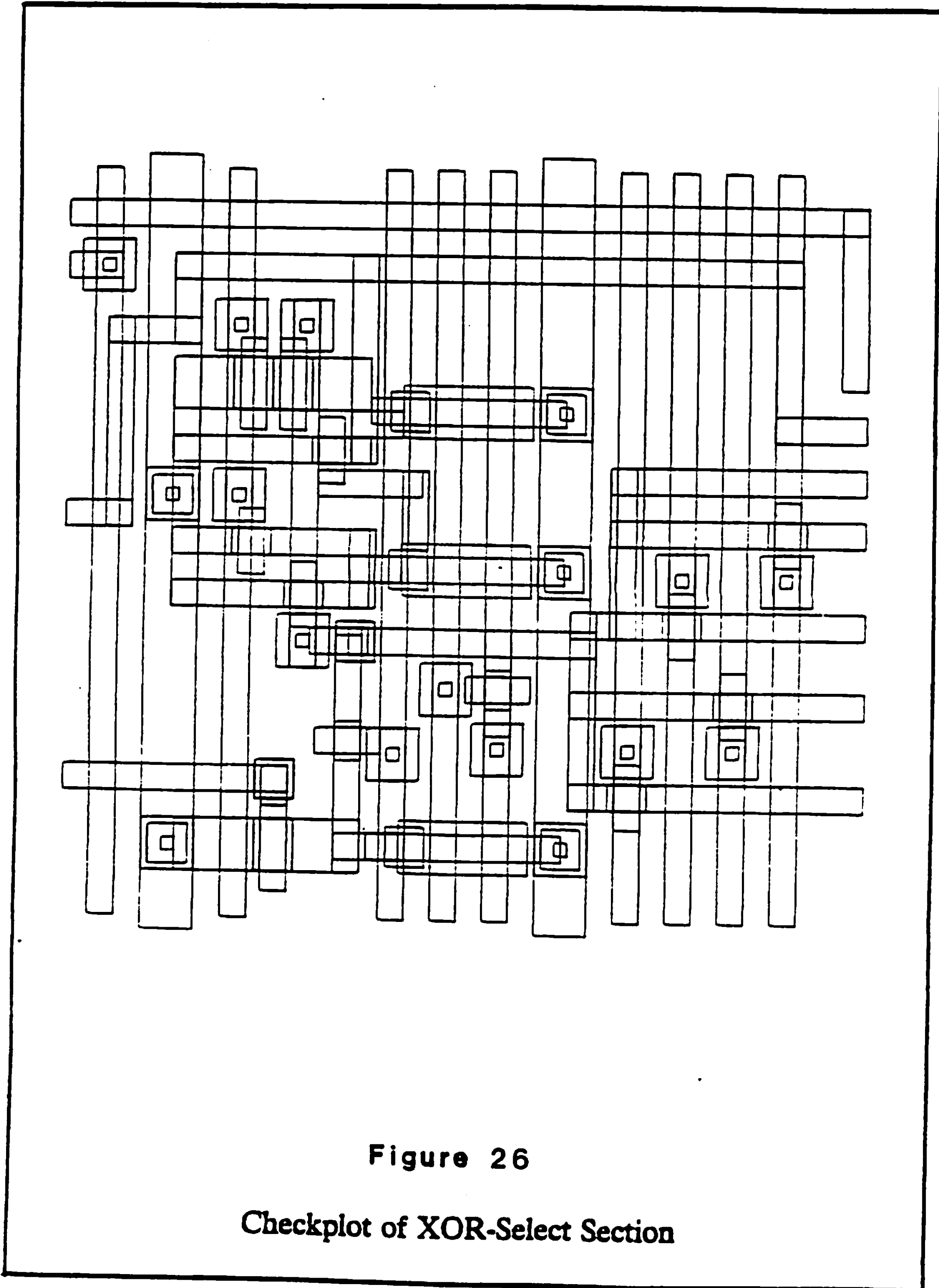


Figure 26

Checkplot of XOR-Select Section

The next section to the right is the off-chip neighbor selector, followed by the off-chip neighbor network, then the on-chip neighbor select and the on-chip neighbor network. These are very similar in construction. Despite this it was found that less space was taken up by having the two separate networks. This is because trying to force both topologies into a single network would have required special switching circuitry for the edges of the net. The position and space that this would occupy would spring the pitch of the cells and thus waste a tremendous amount of area. Each of the selectors is of the same form as the one previously described: Vertical signals are tapped and and gated through pass transistors that are also controlled by vertically running signals. The outputs of the pass transistors are connected to the common data bus with the instruction decoder preventing contention. This is diagrammed in figure 27.

There are four different types of neighbor network segments for each of the two nets. Each net also has a top and bottom termination segment. The segment that is selected for a particular cell depends on which column of the grid topology the cell lies in. It should be noted that for an eight by eight grid, there would be eight of these segments for each network. It should also be mentioned that

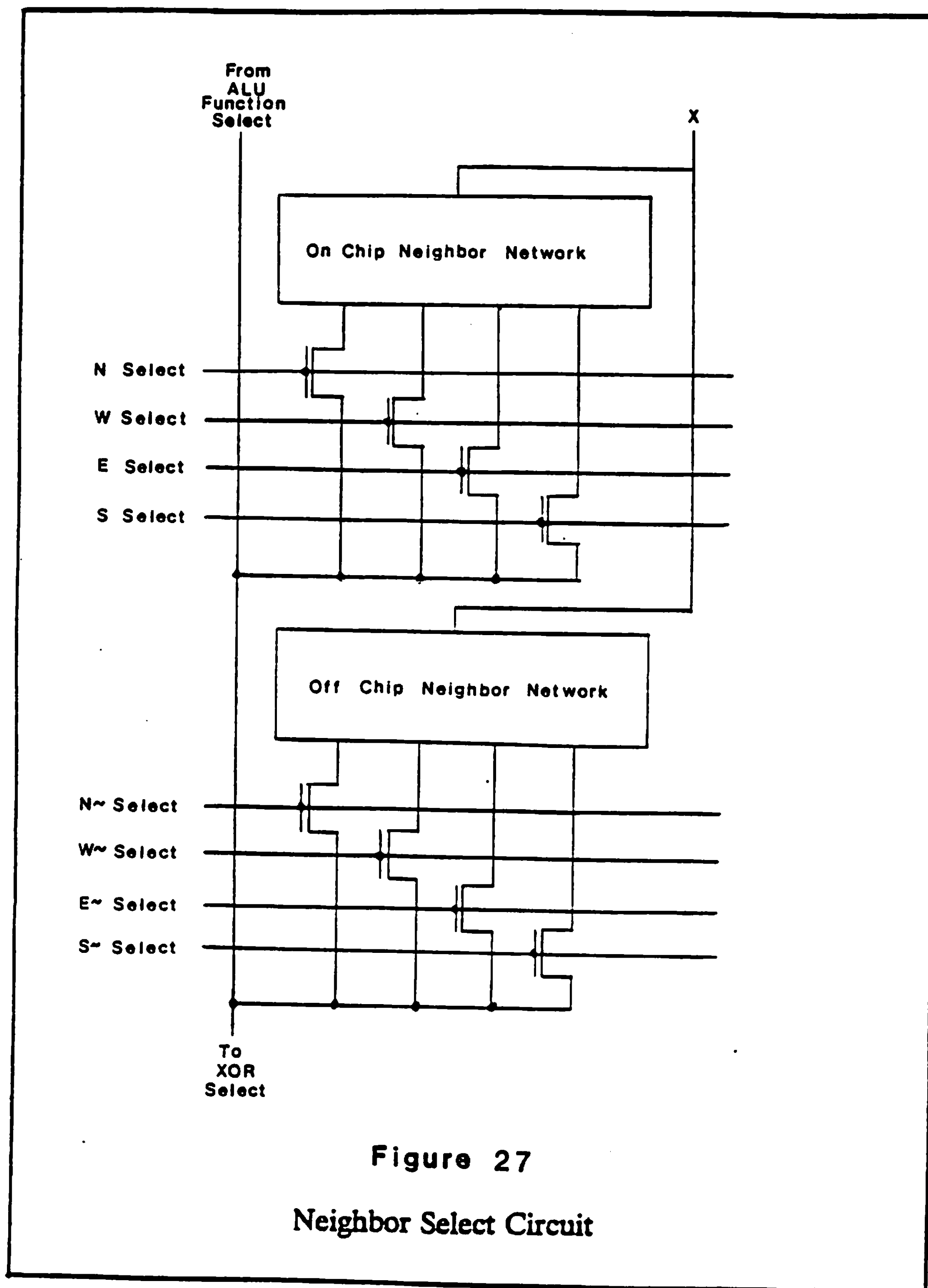


Figure 27

Neighbor Select Circuit

the top processing element in the stack is in the Northwest corner of the grid and the bottom cell is in the Southeast corner. Scanning down through the elements is equivalent to traversing the grid in lexicographic order. Figure 28 shows the four types of segments for the on-chip network and the terminations. These are arranged vertically in the processor stack in the order: Top, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, bottom. Figure 29 gives the four types of segments for the off-chip network and figure 30 shows the termination circuits for these. The order in which these appear in the stack is: TopZ, 1Z, 2Z, 3Z, 4Z, 1Z, 2Z, 3Z, 4Z, 1Z, 2Z, 3Z, 4Z, 1Z, 2Z, 3Z, 4Z, bottomZ. (The Z stands for Zig-Zag network in order to avoid confusion between the two networks.)

All of the segments for each network are designed with a common interface to the rest of the processor and to each other. They can therefore be dropped into the stack in any order if a different topology was desired, for example. This also permits the processor design to remain undisturbed when the larger network is implemented. (It also allows the rest of the processor to be modified without affecting the network design, so long as the interface is maintained.)

To the right of this part of the processing element is the ALU function selector and the ALU itself. The function

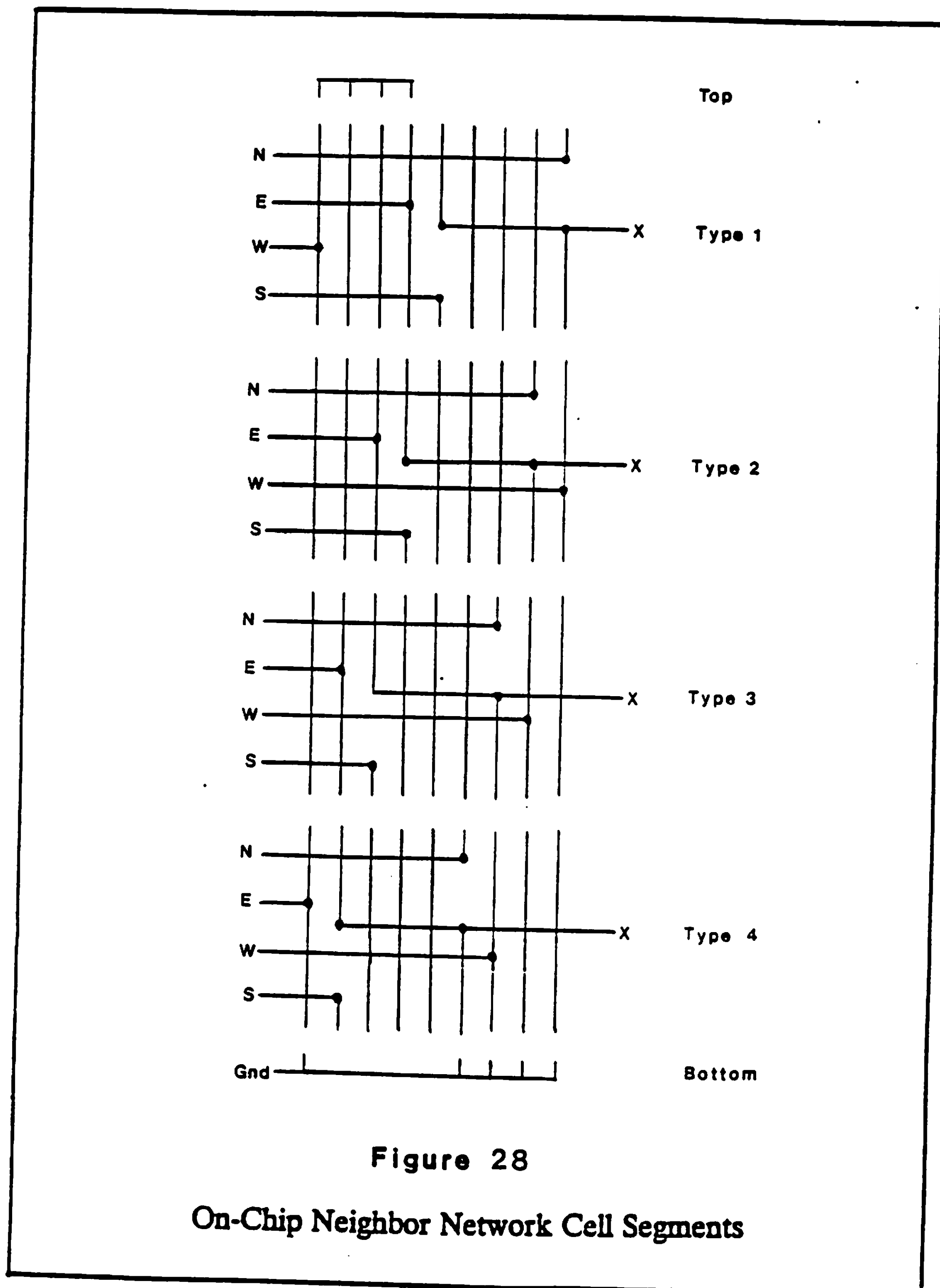


Figure 28

On-Chip Neighbor Network Cell Segments

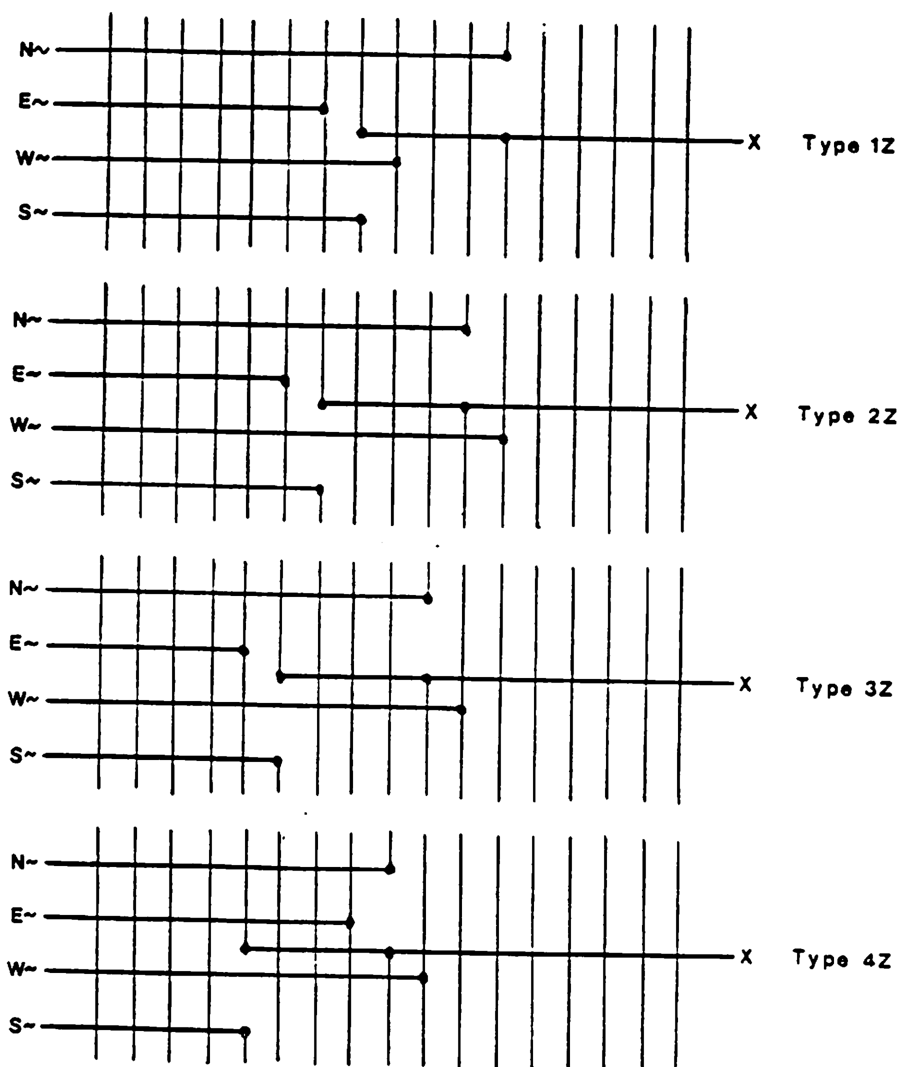


Figure 29

Off-Chip Neighbor Network Cell Segments

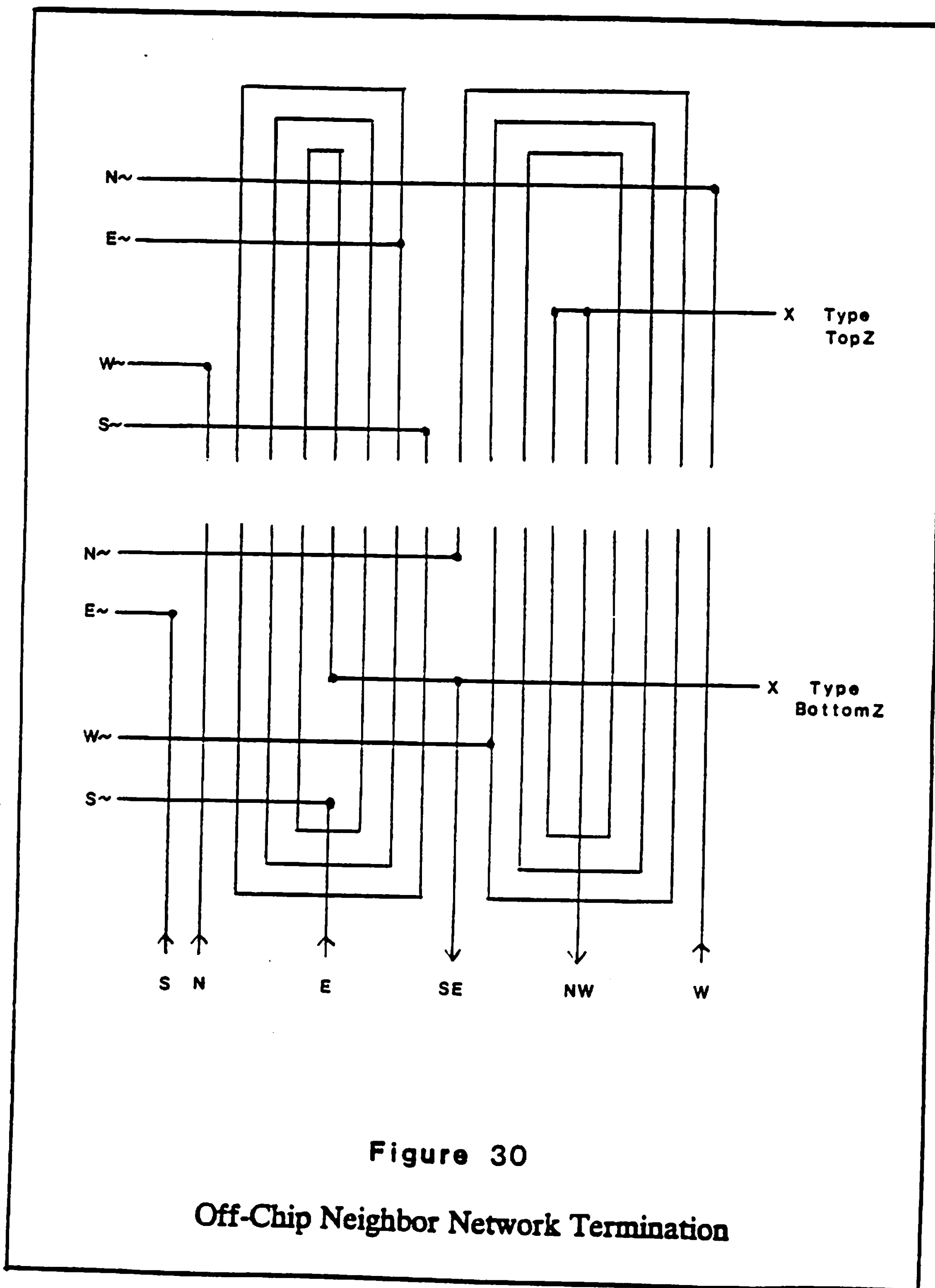


Figure 30

Off-Chip Neighbor Network Termination

selector and its relationship to the neighbor networks is diagrammed in figure 31. This is exactly the same type of selector as the other three. The ALU itself contains three elements. Leftmost is the NOR gate that produces the NOR of X and Y. Next comes the NAND gate and then the full adder. The adder circuit is a standard three input, two output design. The inputs are in true form with the outputs inverted. The X, Y and Z signals enter from the registers at the right. The result exits to the left while the return carry goes back to the right on a dedicated line directly to an inverter and then to the Z input. The X, Y, A, and B values also pass through the ALU to the function selector. The circuit for this is shown in figure 32 and the layout of both the ALU and function selector is shown in the checkplot of figure 33.

The last section in the processing element is the register driver and the five registers. These are arranged with the driver to the left, followed by the Z register, then Y, B, A, and X. This ordering was selected for several reasons. The Z register needed to be close to the ALU and driver because it required special extra circuits to be able to accept both carry and result inputs (recall that Z can take X and C as sources). There was sufficient room to fit this in at the left end and it reduced the distance that the

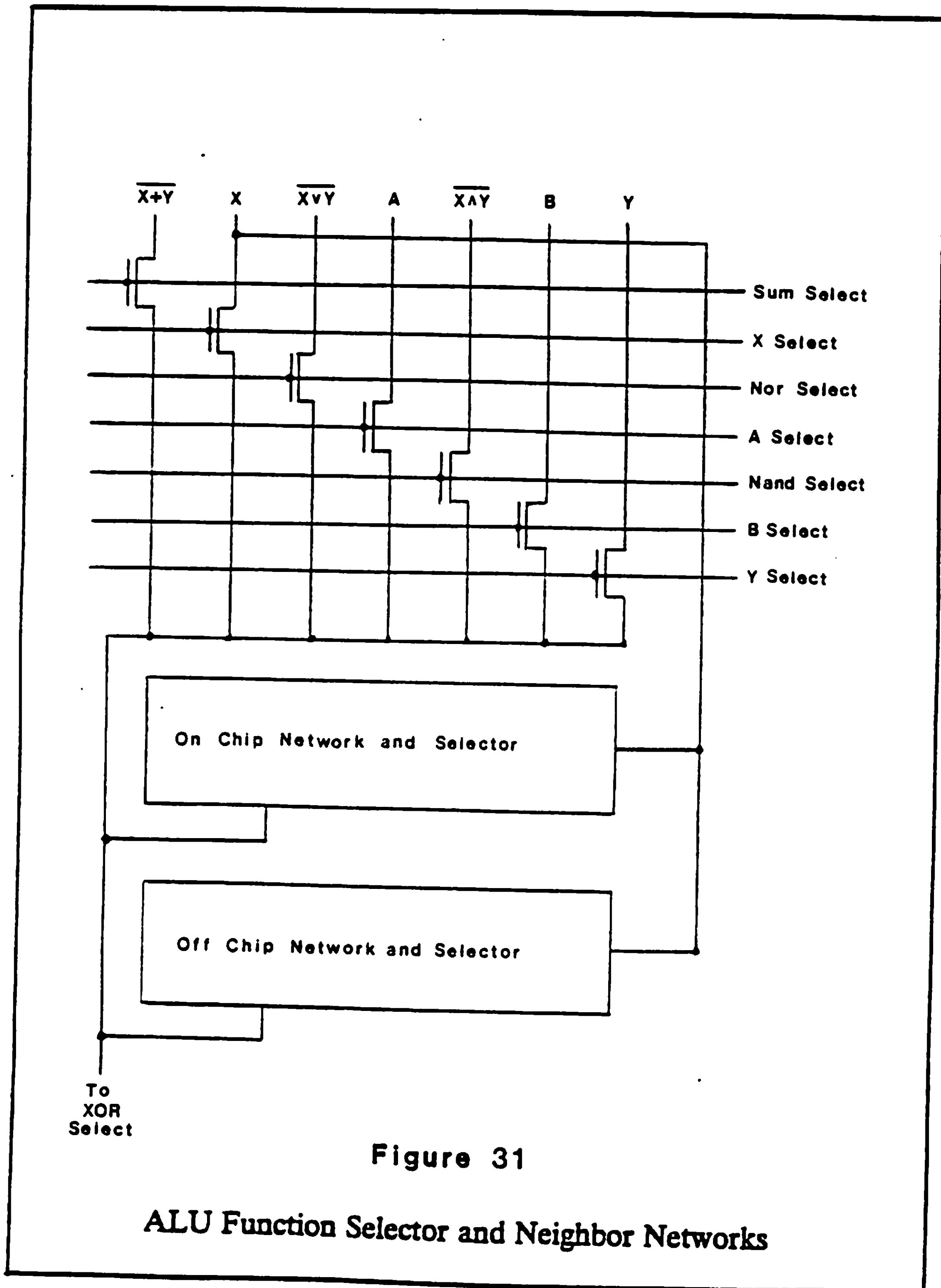


Figure 31

ALU Function Selector and Neighbor Networks

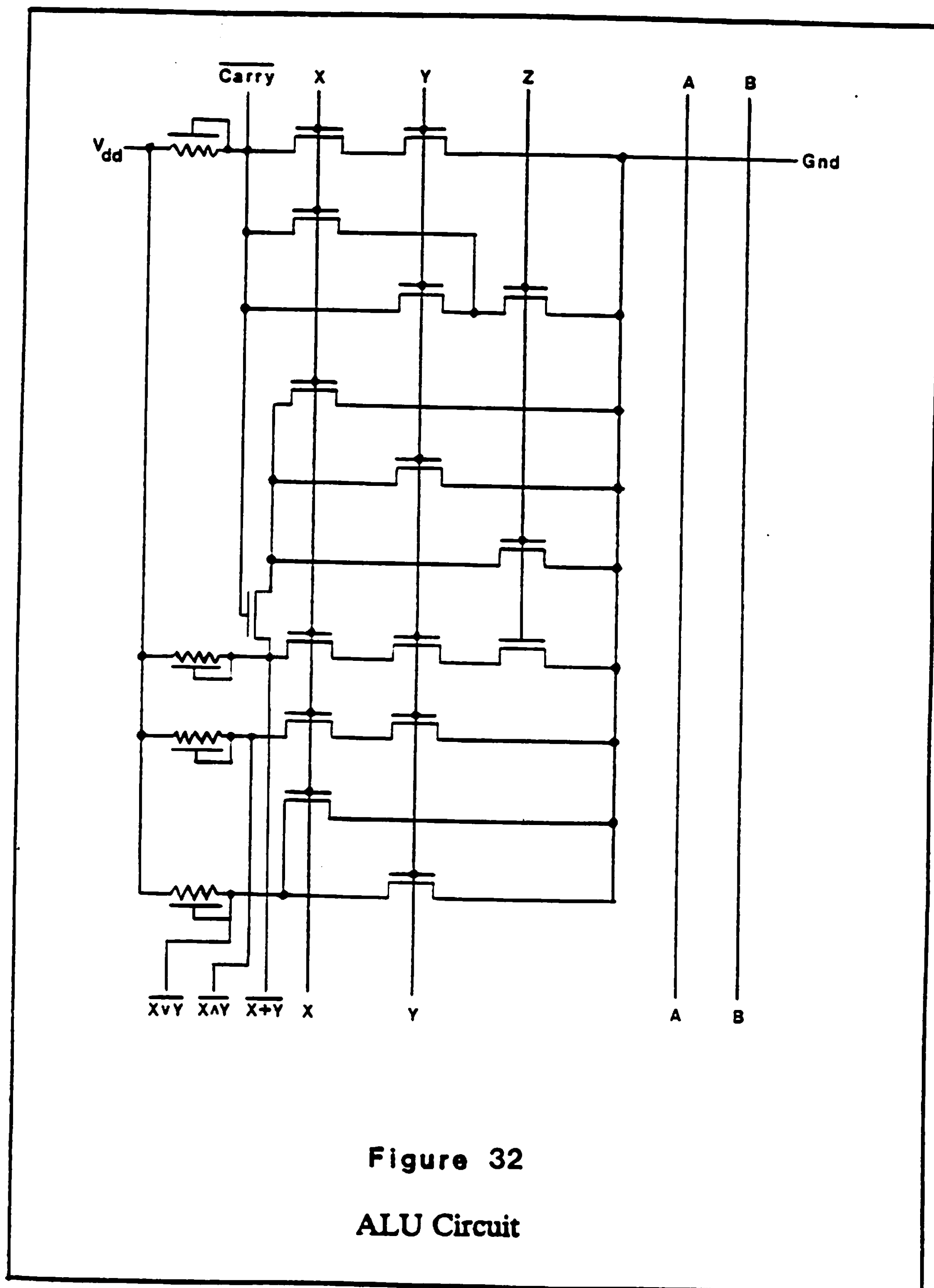


Figure 32

ALU Circuit

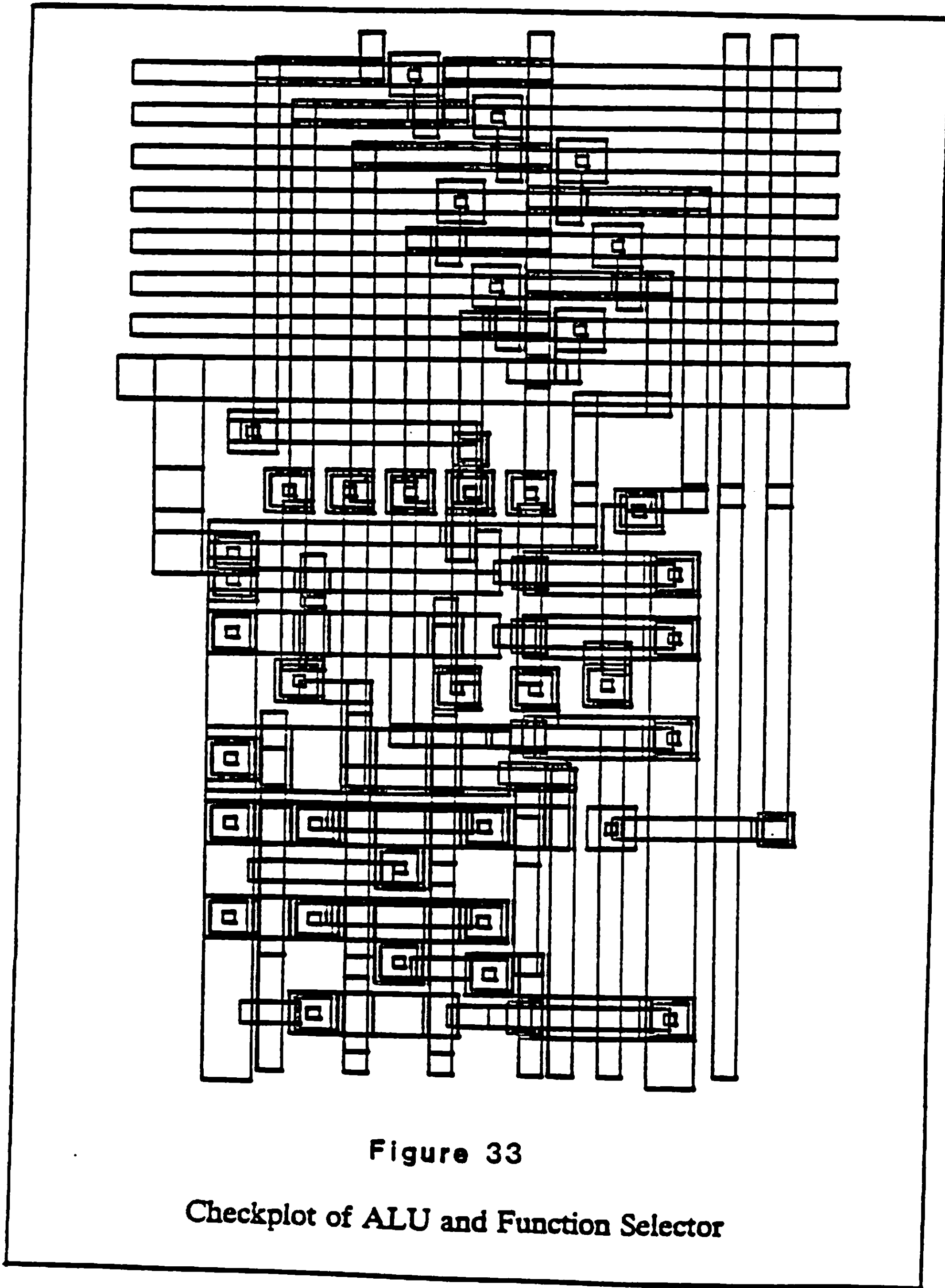


Figure 33

Checkplot of ALU and Function Selector

extra wires had to run. The X register had to be at the right end because this put it closest to the some/none circuitry and avoided having to run that extra wire for any distance. The placement of the rest of the registers was more or less arbitrary except that it allowed their outputs to pass out to the proper connection points at the ALU with a minimum of jogging around obstructions and crossing each other.

The register cells themselves are similar in design to the memory elements, except that a super buffer output stage was used because of the long diffusion lines that they would have to drive. Recall that the memory bits share a common output bus while the registers must all output in parallel. The register driver circuit creates the write enable signal by combining the output of the activity bit with the ignore activity signal from the instruction decoder. This is passed to both the registers and the memory. The driver also selects which data source will be read by the Z register. The individual registers themselves are selected by signals coming down from the instruction decoder. The driver circuit is shown in figure 34 while the circuitry for the registers is split between figures 35 and 36. Figure 37 shows the checkplot of the registers and figure 38 shows their driver.

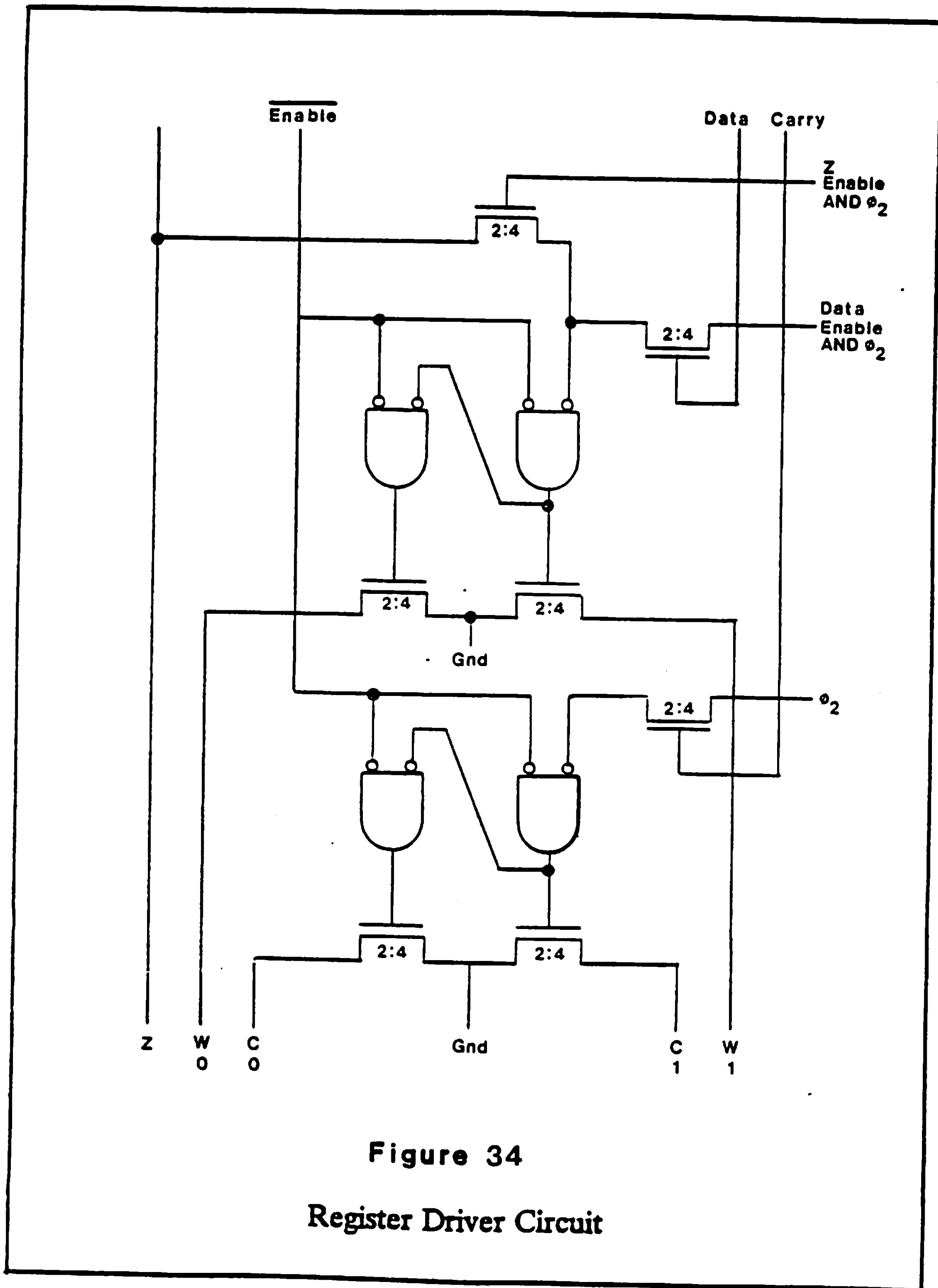


Figure 34

Register Driver Circuit

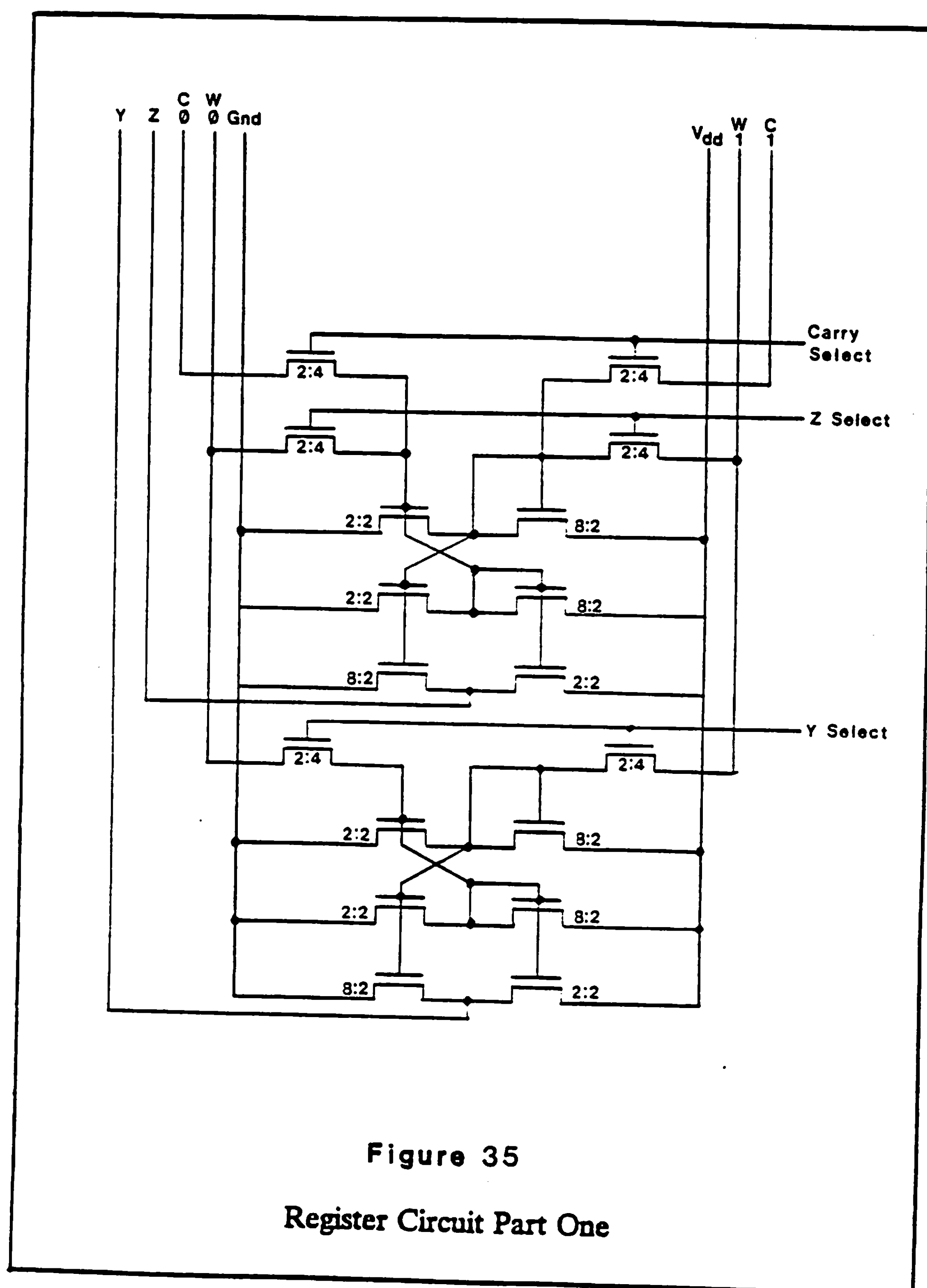


Figure 35

Register Circuit Part One

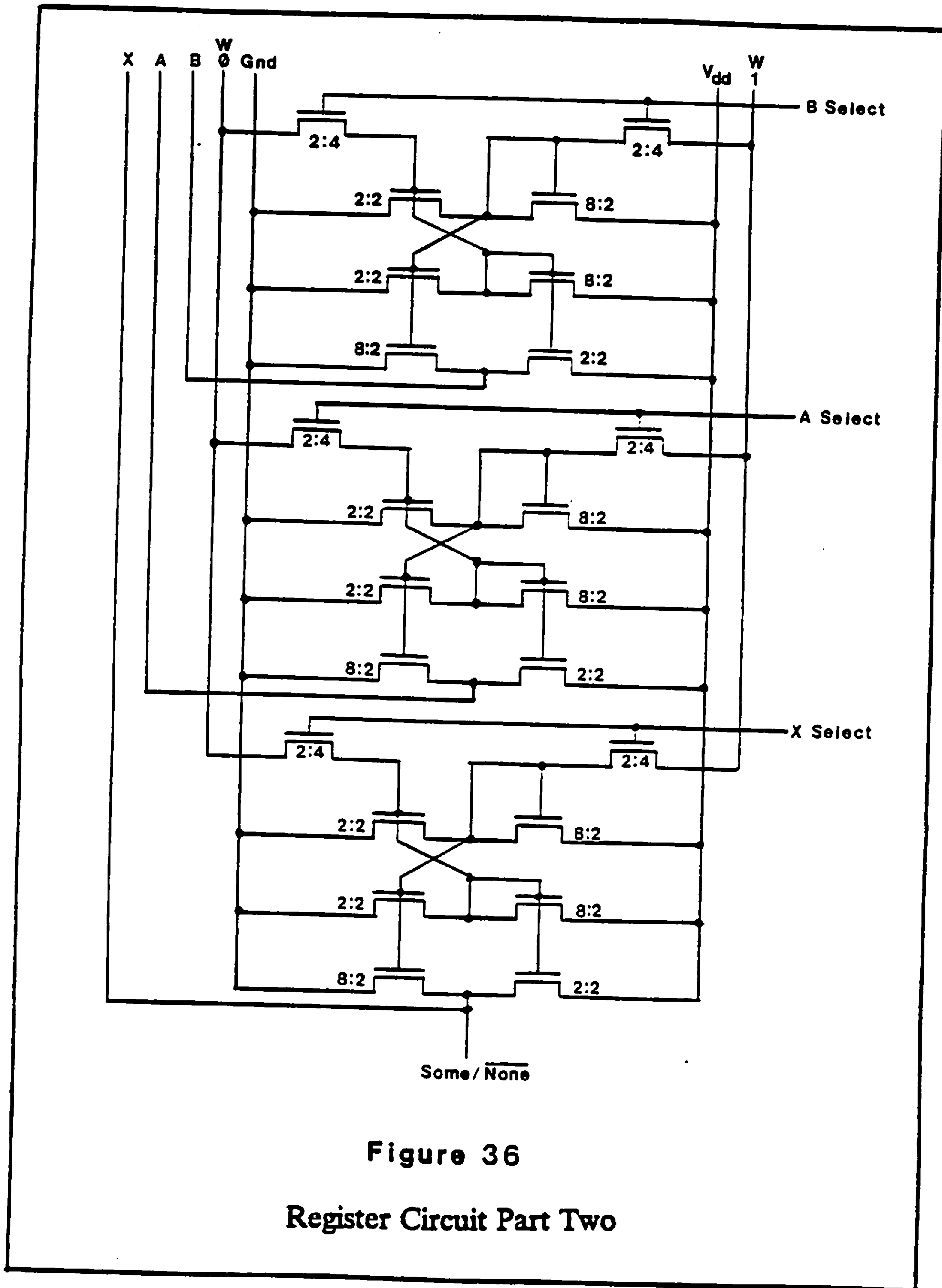


Figure 36

Register Circuit Part Two

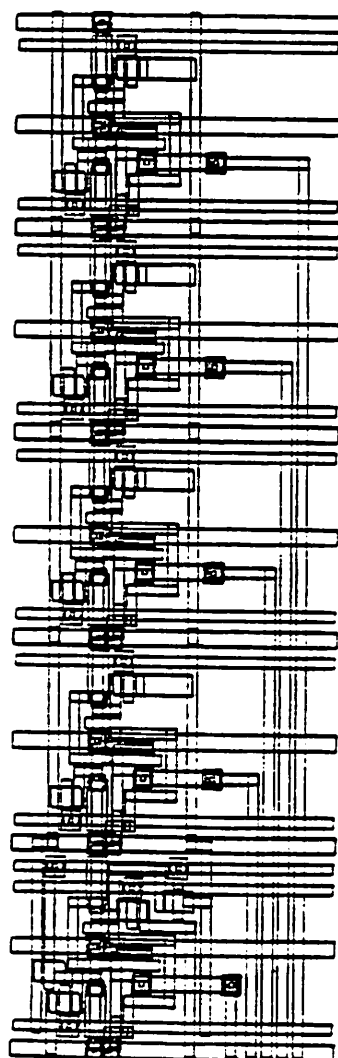


Figure 37

Checkplot of Registers

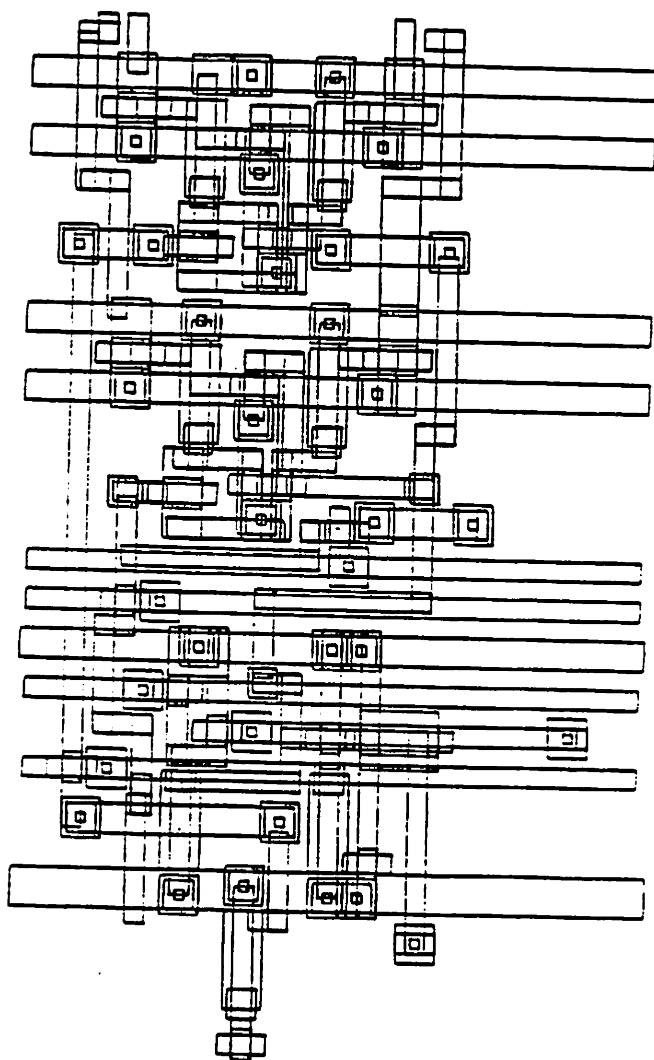


Figure 38

Checkplot of Register Driver

Total Body Area: 1466 lambda square (2,149,156 square lambda)
173 mil square (at 3 micron lambda)

Cell Area: 1,677,500 square lambda (78 percent of body area)

Transistor Type	Number	Area (square lambda)
2x8 Pullup	1682	26912
2x2 Pulldown	2902	11624
8x2 Pulldown	231	3696
4x2 Pulldown	1605	12840
2x2 Pullup	151	640
2x6 Pulldown	64	768
24x2 Pulldown	64	3072
6x2 Pulldown	48	576
2x4 Pullup	1	8
	6752	60100 (2.80 % of body area 3.58 % of cell area)

Designed Transistors: 170
Regularity (mean repetition factor): 39.72

Power Dissipation
100% Duty Cycle

Pullups	1527 x 240 uW	= 366.48 mW	= 73.30 mA at 5VDC
Super Buffers	28 x 1200 uW	= 33.60 mW	= 6.72 mA at 5VDC
High Power Buffers	64 x 2400 uW	= 153.60 mW	= 30.72 mA at 5VDC
		553.68 mW	=110.74 mA at 5VDC

Operating Duty Cycle

Pullups (50%)	183.24 mW	= 36.65 mA at 5VDC
Super Buffers (50%)	16.80 mW	= 3.36 mA at 5VDC
High Power Buffers (2 off)	148.80 mW	= 29.76 mA at 5VDC
	348.84 mW	= 69.77 mA at 5VDC

Table 9

Summary Statistics for Test Chip

Summary statistics. Table 9 presents a statistical summary of the test chip. The figure of 1466 lambda for the width of the body area translates to a little over 173 mils. Doubling this gives a figure of 346.3 mils for the length of the full scale chip. Note, however that the test chip leaves considerable area unused in that dimension. Also note that the decoders will not have to be duplicated for the full design. Thus there should be more than enough room for the full size chip in this dimension. In the other dimension (width), the test chip uses most of the available space. Again, however, this will not all be duplicated since the design calls for interleaving the narrower memory strips which can then share the address decoder. Thus the overall size is likely to be closer to 300 by 300 mils for the production chip. This is well within the original constraint.

The total device count for the test chip is 6752 transistors. Of these, 256 are in the address decoder and 840 are in the instruction decoder. This gives an estimated count for the full chip of 24,560 transistors. This will be slightly greater as the response counter has yet to be added, but it is still well within the design constraint of 40,000 devices. It is interesting to note the nearly 40 percent regularity factor for the design. This is extremely

high as compared to most other VLSI circuits, with only pure memory elements typically being greater. The regularity is expected to increase slightly for the full size chip. Also the figure of 3.58 percent active area is not at all unreasonable, especially for a green design team.

The power dissipation for the chip at operating duty cycle is estimated at about 349 milliwatts. Of this, roughly 43 percent is due to the fact that the logic sense of the memory select lines requires all but two of the high power buffers to be turned on at any time. Although time did not permit a redesign, if ever the chip was to be built for fabrication, this would have to be changed. Even so, the total power for the 64 element chip (because the address buffers would also be shared) would be approximately 950 milliwatts. With the reversed logic sense on the select lines, this would drop to about 800 milliwatts. Again, this is well within the original design constraint of 2 watts per package and should be easily cooled by forced air.

Memory simulation and fabrication. As was mentioned above, the test chip never went to fabrication. Near the end of the project the offer of a fabrication run was withdrawn by the local company. The final wiring of the instruction decoder to the processing elements and of the I/O lines to the pads and external neighbor preselect was

never completed. It was felt that the rest of the project work had adequately demonstrated the buildability of the chip and that there was no point in continuing since the primary objective had been met.

There was still considerable concern that the memory design might not work. When the opportunity arose to have the circuit analyzed and fabricated (via MOSIS) as part of a senior project, it was immediately taken. The two seniors, Estabrook and Weiss [31], used the SPICE circuit simulator to determine the operating characteristics of the memory strip. They simulated both the memory cell as an independent entity and having to drive the long diffusion line that formed the read bus. It was felt that there was no problem with writing into the memory cells because a super buffer was being used to drive the write lines. The cell itself performed very well, accepting data and regurgitating it when told to, at better than the speed goal for the design.

As was expected, however, they found that the cell was unable to drive the long diffusion line. The charge on the read line was so great that the cell could barely pull it down. This was better than the worst possible case, however. There was considerable worry that the charge on the read line might actually overwrite the value in memory

when the gate was opened for reading. (Or as they refer to it, the tail would wag the dog.) This was not the case, however. A series of experiments then began to find a read circuit that could detect the small change in level accompanying a read operation.

A simple sense amplifier was eventually chosen that was capable of doing this. The amplifier consisted of two cross coupled inverters, forming a flip flop that is tied to the two memory I/O lines. This is essentially a memory bit with the pass transistors for selection removed. The regenerative nature of this circuit pushes the read line to its appropriate level. This lead to a problem in that the sense amp tended to hold the write-one line low after reading a one. This made the next transition to the high state for this line too slow. To overcome this the design was modified to provide a simple form of precharging. A transistor was connected with its source and drain to the I/O lines so that they would be brought to an equilibrium state. This unclamps, or resets, the sense amp. It also brings the line voltages close to the transition voltage for the readout inverter so that the inverter switches more quickly to the appropriate level.

The spice simulations were run on the modified cell with six nanoseconds of precharge time. The results showed that

a read/write cycle could be performed in as little as 50 nanoseconds or at a frequency of 20 MHz. This is double the design goal of 10 MHz. It was felt that because this part of the chip was the most likely to bring the speed down to a lower value, that the rest of the chip could probably be made to function at the same speed. The addition of the sense amplifier and precharge circuit does not add significant complexity to the design. The total additional circuitry amounts to the equivalent of adding one extra memory bit to each processing element. Nor is the sense amplifier of exotic design, using purely standard design rules. This leaves plenty of room for improvement should it later be decided to push the memory technology a little further.

Unfortunately the fabrication of the memory element did not go as well as the simulations. In converting the design to MOSIS design rules a number of errors were introduced. Close examination of the photomicrograph of figure 39 will reveal a number of obvious design rule violations. The most obvious of these is that the select lines are shorted to Vdd in each of the memory cells. It can also be seen that the chip select inputs are shorted together (upper right corner). Further the input line can be seen to short to Vdd as it passes under a cut between metal and diffusion at the

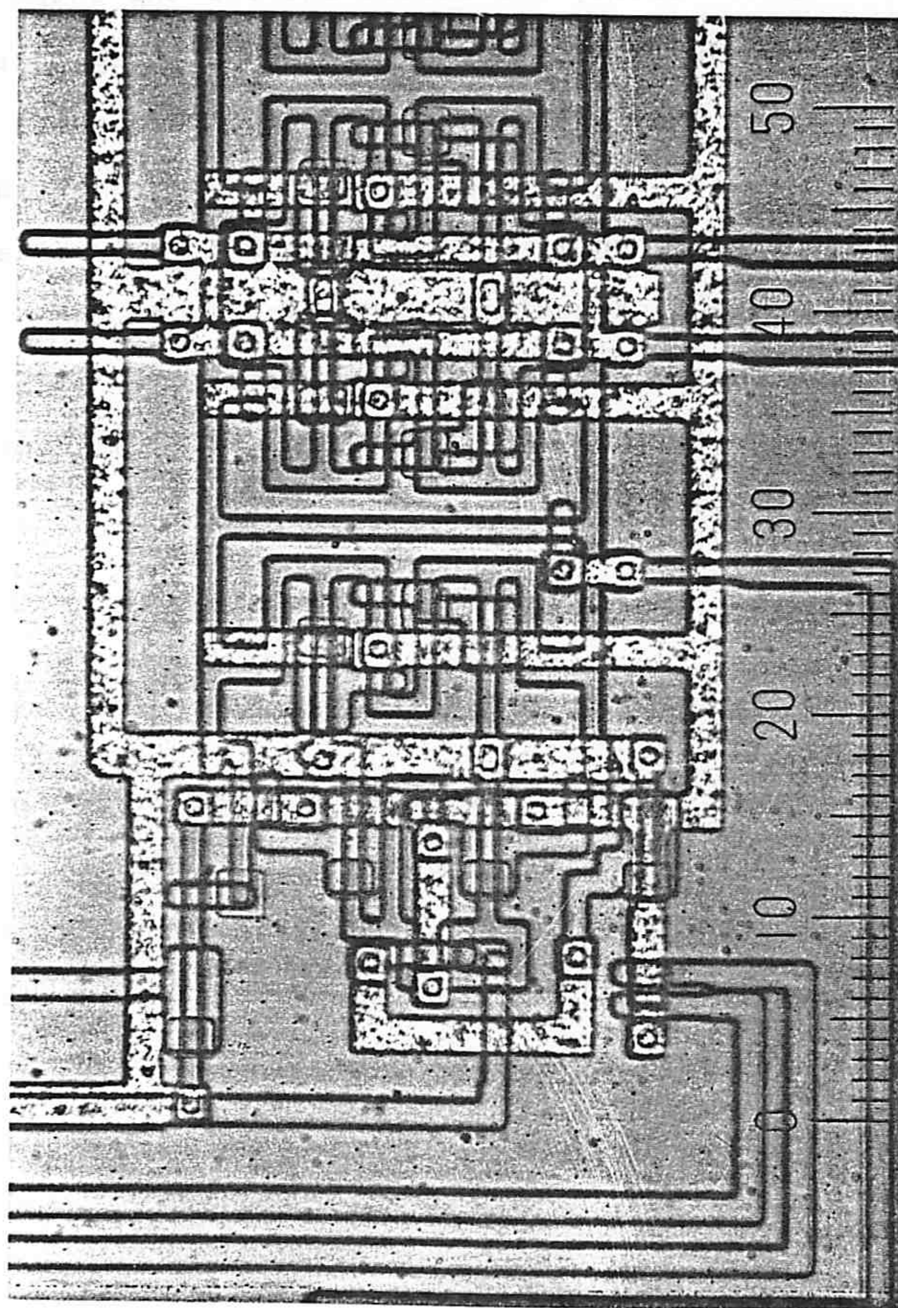


Figure 39

Photomicrograph of Memory Cell and Driver from Test Fabrication

lower right. Lastly the buried cut for the output inverter (same area) is at the Vdd end of the gate region rather than the ground end, where it should be. This means that the output is also tied to Vdd. (If the input to the inverter could go high, then it would simply form a voltage divider between Vdd and ground with the ratio of the lengths of diffusion on either side of the cut. This might lower the output level slightly.) Empirical analysis confirms that all of this is true.

Design Conclusions

It has been shown that the Titanic CAAPP can indeed be designed to meet all of its goals without violating any of the design constraints. This shows that such a machine may well be built without simultaneously pushing the limits of construction technology.

CHAPTER V

APPLICATIONS AND ANALYSIS

Introduction

This chapter will be a presentation of about 25 programs and algorithms for the architecture described in the preceding chapter. Timing and instruction utilization estimates will be given for each of these, as well as general comments on the suitability of the architecture for the particular algorithm. The next chapter will be an overall evaluation of the architecture in the form of a summary of this chapter, followed by a discussion of desirable changes and enhancements. The algorithms are listed in table 10. Because the code for some of the algorithms is too lengthy to be presented here, the table notes for each of the algorithms whether or not the algorithm was actually coded. The table also indicates, for each algorithm, whether or not it makes use of intercell communications, the some/none report mechanism, the response count mechanism, broadcast data, and local cellular processing. Of course, all of the algorithms use the instruction broadcast mechanism.

Algorithm	Micro Coded	Broad-cast Data	Local cell ALU	Inter-cell Comm.	Some None	Fast Count
Basic Macro Operations						
Response count	Yes	No	No	No	No	Yes
Exact match	Yes	Yes	Yes	No	No	No
Greater than	Yes	Yes	Yes	No	No	No
Less than	Yes	Yes	Yes	No	No	No
Select greatest	Yes	No	Yes	No	Yes	No
Select least	Yes	No	Yes	No	Yes	No
Select first	Yes	No	Yes	Yes	Yes	No
Add constant	Yes	Yes	Yes	No	No	No
Subtract constant	Yes	Yes	Yes	No	No	No
Add fields	Yes	No	Yes	No	No	No
Subtract fields	Yes	No	Yes	No	No	No
Multiply constant	Yes	Yes	Yes	No	No	No
Divide constant	Yes	Yes	Yes	No	No	No
Multiply fields	Yes	No	Yes	No	No	No
Divide fields	Yes	No	Yes	No	No	No
Simple Image Processing Operations						
Game of life	Yes	Yes	Yes	Yes	No	No
Gaussian smoothing	Yes	Yes	Yes	Yes	No	No
Sobel edge extract	Yes	Yes	Yes	Yes	No	No
Histogram	Yes	Yes	Yes	No	No	Yes
Rotate model view	Yes	Yes	Yes	Yes	Yes	No
Higher Level Image Processing						
Connected components	Yes	Yes	Yes	Yes	Yes	No
Motion parameters	No	Yes	Yes	Yes	Yes	Yes
Other Application Areas						
Center of mass	Yes	Yes	Yes	Yes	No	Yes
Geocorrection	Yes	Yes	Yes	Yes	Yes	No
Square grid sort	No	No	Yes	Yes	Yes	No
Real time LISP	No	Yes	Yes	No	Yes	No
Neural networks	No	Yes	Yes	Yes	Yes	No
Semantic networks	No	Yes	Yes	Yes	Yes	Yes

Table 10

Algorithms Developed for the CAAPP Evaluation

What table 10 actually points out is which of the algorithms use purely the associative aspect of the processor, which algorithms use purely the square grid aspect, and which algorithms use both aspects of the processor. Purely associative algorithms are indicated in the table by a "no" in the Intercell Communications column. For those algorithms with a "yes" in the Intercell Communications column, if the corresponding entry in both the Some/None column and the Fast Count columns is "no", then the algorithm uses purely the square grid aspect of the CAAPP. Those algorithms with a "yes" in the Intercell Communications column and a "yes" in either or both of the Some/None and Fast Count columns use both the associative and the square grid aspects of the CAAPP.

The collection of algorithms will be divided into four parts: basic operations, simple image processing, higher level image processing and other application areas. The first section will contain programs for the CAAPP that will most likely be contained as microcoded instruction sequences in the controller's ROM. These are presented here because their expected frequency of use makes timing analyses of them important in developing timing estimates for the more complex algorithms to be presented later. The second section presents several low level image processing

applications including a Gaussian smoothing convolution, the Sobel edge detection operation, histogramming and hidden surface removal. The third section develops two somewhat more complex image processing applications: region growing and labelling, and motion flow field extraction. The last group of algorithms is intended to give a sampling of some of the other application areas to which the CAAPP could be applied. The areas that will be examined are geocorrection of satellite images, center of mass calculation, and a sorting algorithm. In the conclusion it will be pointed out how the techniques developed for some of these algorithms can be used in a variety of other areas.

The notation that will be used in the following algorithm presentations is a combination of a structured language similar to Pascal and the instructions listed in the table of CAAPP instructions presented in the previous chapter. As a review, remember that: X, Y, Z, A and B will refer to the registers in the CAAPP cells: X is the response bit, Z is the carry bit, A is the activity bit, Y and B are temporary storage. M(I) will refer to the I-th bit of memory in a cell. C refers to the broadcast comparand bit. Occasionally C(I) will be used to refer to the I-th bit of a value that is being broadcast bit serially by the controller. The constants 1 and Ø will also refer to the

broadcast comparand but further specify its exact value. An operation with a trailing exclamation point (!) will ignore the value of the activity bit (A) and take place on all cells. N, S, E, W refer to the X register of the neighboring cell in that direction. If a direction is followed by a tilde (~), the zig-zag connection network will determine the cell that is the nearest neighbor in that direction (as opposed to the on-chip connection network). If a zig-zag operation is preceded by a "C", a "1" or a "Ø", then the input from outside of the chip is the broadcast comparand bit and the output is discarded (normally input is from a neighboring chip and output is likewise to a neighboring chip). When preceded by a minus sign (-), a value is to be logically inverted before it is operated on. The symbols "^", "v" and "+" will be used to represent logical "and", "or" and "full adder sum" (X + Y + Z) respectively.

Basic Operations

Count Responders. This operation returns a count of the number of elements which have their X registers set to one. It uses a special set of hardware and three special CAAPP instructions to accomplish this. The operation begins by

clearing a counter register on each of the chips. The X registers are then zig-zag shifted cyclically through this counter. The result is that the X register values all end up back in their original places and the counter contains the number of one-bits in the X registers on the chip. The next phase of the operation is to perform a pipelined addition of all of the counter registers within each column. This is done by streaming the contents of the counters into a chain of full adders that run the length of a column of chips. On the first step of this phase the top chip in each column transmits the low order bit from its counter to the chip below it. On the second step the chip just below the top sums the bit that it just received with its own low order counter bit and passes the result to the chip below itself. At the same time, the top chip will pass its second lowest order counter bit to the chip below it. On the third step, the chip in the third row sums the bit from above with its lowest order counter bit and passes the result to the chip below, the chip in the second row sums the bit from above it with its second lowest order counter bit and passes the result to the chip below, and the chip in the top row passes its third lowest order counter bit down. Processing proceeds in this fashion until the full sums of all of the counter registers in each column of chips streams out the bottom of the column into special holding registers

on the edge of the entire array. A similar sort of pipelined addition is then performed on these registers to produce the full sum which is the count of responders. The algorithm along with timing estimates is now presented:

```
A := 1!
CRCR                (* Clear response count register *)       0.1 us
For I := 1 to 64 do                                            0.1 us
    SCRR            (* Shift and count responders *)
Turn off all chip row select lines                             6.4 us
Turn on all chip column select lines                           0.1 us
For I := 1 to 64 do                                            0.1 us
   begin
      Turn on row select line I
      PANS          (* Pipeline add North to South *)
   end
For I := 1 to 8 do (* Empty the pipe *)                       12.8 us
   PANS             (* Pipeline add North to South *)
For I := 1 to 64 do                                            0.8 us
   PAWE             (* Pipeline add West to East *)
                                                               6.8 us
                                                           ----------
                                                              26.8 us
```

It is interesting to note that this algorithm is only two orders of magnitude slower than the optimum (0.1 us -- single instruction cycle) despite the small amount of hardware included in the CAAPP to provide the count capability. Recall from chapter 4 that the only hardware specifically included for the purpose of forming a response count is a seven-bit counter and a full adder on each chip and some special logic on the bottom edge circuit boards. It is also interesting to note that adding the counter and

adder to the chips does not make them dependent upon the size of the overall array or, conversely, the array size is not limited by the response count circuitry on the chips. This presents a good hardware cost versus operating speed tradeoff.

Exact Match. The exact match operation is one of the most basic in any CAM or CAPP system. The algorithm that will be presented takes advantage of the special properties of the activity mask. In order to do this without destroying the mask, the B register will be used as temporary storage for the A register during the operation. This is an example of the primary intended use of the B register. The result will be that the X registers of the active cells that match the comparand will be set to one and all of the others that were originally active will be set to zero. Inactive cells will be unaffected except that their B registers will be changed when the current mask is saved. In the following, "S" is the starting (low order) bit position of the field being matched and "L" is its length.

```
B := A!                                                  Ø.1 us
For I := S to S+L-1 do
    begin
        X := M(I)                                    L * Ø.1 us
        Y := C                                       L * Ø.1 us
        Z := Ø                                       L * Ø.1 us
```

```
      X := -(X + Y)(* NOT XOR is equality test *) L * Ø.1 us
      A := X                                      L * Ø.1 us
   end
A := B!                                               Ø.1 us
                                            ----------------
Total                                       L * Ø.5 + Ø.2 us

Sample time for 8 bit comparison                      4.2 us
```

If it is not essential to preserve the status of the X and Y registers of inactive cells, approximately one fifth of this time can be saved by directly assigning -(X + Y) to A in the loop and then jam transferring A to X at the end of the loop. The time would then be (L * Ø.4 + Ø.3) for the exact match (3.5 us for 8 bits).

In case the reader is wondering why the above algorithm does not directly assign -(X + Y) to A: The value is first assigned to X in order to give the response bit the proper value in case the following assignment to A shuts off activity in the cell. The alternative is to ignore X during the matching computation and then to use the final values of A and B to compute the proper value for X. This latter requires rearranging the registers to do the computation. The result is that for fields 5 bits or less in length the above algorithm is faster. For longer fields it is more efficient to rearrange registers and compute X after the loop terminates. The timing for this alternate approach is (L * Ø.4 + Ø.7 us).

Greater and less than searches. These two searches (along with greater than or equal and less than or equal) are most often used in applications where some threshold is used as a selection criterion. They can also be used together to do bandpass or band reject selections. The searches assume that the bit slice to be compared is in M(S) through M(S+L-1) with the latter being the high order bit. The result of the search will be that, among those cells that are initially active (A = 1), X will be set to one if the slice is greater (or less) than the broadcast comparand, zero otherwise. The contents of B will be lost, but A will be restored to its original value in each cell. The basic idea behind this algorithm is that X will be set to the M(I) bit and Y to the C bit (for a greater than search, for less than the values are loaded into the opposite registers). If the two aren't equal then X will already contain the appropriate logic value so we just turn off activity. To see how this works, take the example in which all of the cell's bits have matched the comparand's bits up to a particular bit position. At the point they differ, either X or Y may be one but whichever it is, the other register must contain zero. If X contains the one, it means that the cell's value is greater than the comparand and so the cell's response bit should be set. This would mean setting the X bit to one in most cases but in this case it is already one,

so nothing needs to be done. The same applies in the case that Y contains the one and X the zero. This means that the cell's value is less than the comparand and thus its response bit should contain zero -- which is exactly the existing condition. Thus all that has to be done is to turn off activity in the cell in order to preserve the state of its response bit while the remaining bits in other cells are compared. If the X and Y bits are equal then activity remains turned on and processing proceeds to the next bit position. At the end of the slice, we wrap up the operation by setting X in all remaining active cells to zero (since those cells equal the comparand). If the search is also to accept equality then we just set the remaining X registers to one. The algorithms are:

Search for all cells greater than comparand:

```
B := A!                                                     0.1 us
For I := S+L-1 downto S do
   begin
      X := M(I)                                         L * 0.1 us
      Y := C                                            L * 0.1 us
      Z := 0                                            L * 0.1 us
      A := -(X+Y)   (* Not XOR is equality test *)      L * 0.1 us
   end
X := 0              (* X := 1 for >= search *)              0.1 us
A := B!                                                     0.1 us
                                                      ----------------
Total                                               L * 0.4 + 0.3 us

Sample 8 bit comparison                                     3.5 us
```

Search for all cells less than comparand:

```
B := A!                                                   0.1 us
For I := S+L-1 downto S do
   begin
      X := C                                          L * 0.1 us
      Y := M(I)                                       L * 0.1 us
      Z := 0                                          L * 0.1 us
      A := -(X+Y)   (* Not XOR is equality test *)    L * 0.1 us
   end
X := 0              (* X := 1 for <= search *)            0.1 us
A := B!                                                   0.1 us
                                                ----------------
Total                                           L * 0.4 + 0.1 us

Sample 8 bit comparison                                   3.5 us
```

It should further be explained that the reason for setting Z to zero is to ensure that the result of the (X + Y) operation is indeed purely an exclusive OR. This is another example of economization in the circuitry: Making a full adder also serve as an exclusive OR gate.

<u>Greatest and least searches</u>. These two searches select all cells from among those currently active which have values equal to the highest (or lowest) value among the active cells in the array. The activity mask is preserved but B is lost in all of the cells. It is interesting to note that these searches are faster than either Exact Match or Less or Greater Than searches. This is exactly the opposite of what one would expect. In fact we shall see in the next chapter that this is purely an artifact of the architecture and instruction set of the PEs. In the

following the variables S and L are the starting bit of the bit slice and its length, respectively.

Find Greatest:

```
B := A!                                                 0.1 us
For I := S+L-1 downto S do
   begin
      X := M(I)                                     L * 0.1 us
      if Some                                       L * 0.1 us
         then A := X                                L * 0.1 us
   end
A := B!                                                 0.1 us
                                              ----------------
Total                                         L * 0.3 + 0.2 us

Sample 8 bit comparison                                 2.6 us
```

Find Least:

```
B := A!                                                 0.1 us
For I := S to S+L-1 do
   begin
      X := M(I)                                     L * 0.1 us
      if Some                                       L * 0.1 us
         then A := -X                               L * 0.1 us
   end
A := B!                                                 0.1 us
                                              ----------------
Total                                         L * 0.3 + 0.2 us

Sample 8 bit comparison                                 2.6 us
```

The above algorithms take advantage of the separation of the response bit (X) and the activity bit (A) into two distinct functions. It is interesting to note that these operations are optimal, ie. there are no simple

architectural or instruction set enhancements that could make them any faster. The timings given are maxima. Depending upon the data set, these algorithms could run in as little as (L * Ø.2 + Ø.2 us) or 1.8 microseconds for an 8 bit comparison. The minimal time data set is, however, a degenerate case. The average data set will approach the maximum time.

Select First. This operation is used when a CAAPP query selects multiple responders and it is desired to process them individually. It results in a single one of the responders being active, with activity turned off in the remainder of them. The algorithm presented below can be applied when the CAAPP already has some activity pattern present and will select a responder from the subset of responders that are in active cells. (It is a simple matter to change the algorithm to operate on all responders.) The algorithm also returns the <X,Y> location of the selected cell to the controller. The reason for calling this operation "Select First" is that the cell selected will be the first active responder encountered if the CAAPP array is searched in lexicographic order (left to right, top to bottom). This is, as was mentioned in the previous chapter, the order in which the CAAPP cells appear when they are treated as a linear array.

The following algorithm is divided into five parts. The first part simply saves the original response and activity patterns and prepares the response store for the search. The second part performs a binary search on the rows of chips in the array to find the topmost with at least one responder. It is interesting to note that the associative nature of the CAAPP allows this type of search to be performed even though we are not dealing with an ordered list of search keys. The third part of the operation uses a one cell per column activity mask and the On-Chip shift network to find the row of cells within the selected row of chips. The fourth part performs another binary search, this time by columns, to find the leftmost chip in that row which contains a responder in the selected row of cells. The fifth part determines which column of cells within the chip contains the leftmost responder. The algorithm is:

```
Turn on all chip column selects                                 Ø.1 us
Turn on all chip row selects                                    Ø.1 us
if None
   then exit                                                    Ø.1 us
B := A!      (* Save activity pattern in B *)                   Ø.1 us
A := 1!                                                         Ø.1 us
Z := X       (* Save response pattern in Z *)                   Ø.1 us
A := -B                                                         Ø.1 us
X := Ø       (* Clear responders in inactive cells *)           Ø.1 us
A := 1!                                                         Ø.1 us
Y := X       (* Save masked responders in Y *)                  Ø.1 us
                                                               --------
                                                                1.Ø us
```

```
Low_Row := 0                                                  0.1 us
High_Row := 31                                                0.1 us
For K := 1 to 7 do    (* Binary search to find top *)
   begin              (* chip row with a responder *)
      Turn on Low_Row through High_Row
         chip row selects                                7 * 0.1 us
      if Some                                            7 * 0.1 us
         then High_Row :=
            Low_Row + (High_Row - Low_Row) div 2         7 * 0.4 us
         else
            begin
               Mid := (High_Row - Low_Row) div 2         6 * 0.3 us
               Low_Row := High_Row + 1                   6 * 0.2 us
               High_Row := Low_Row + Mid                 6 * 0.2 us
            end
   end
                                                    ----------------
                                                    4.4 us to 6.2 us

I := 1                                                        0.1 us
X := 0                                                        0.1 us
X := -N    (* Sets top row of X bits to one *)                0.1 us
A := X     (* Top row is now active        *)                 0.1 us
X := Y     (* Restore masked response pattern *)              0.1 us
while None do                                                 0.1 us
   begin
      X := 1                                             7 * 0.1 us
      X := 0N~!   (* Move row of ones down *)            7 * 0.1 us
      A := X!                                            7 * 0.1 us
      X := Y                                             7 * 0.1 us
      I := I + 1                                         7 * 0.1 us
   end
                                                       -------------
(* Topmost row of cells is now selected *)             0.7 to 4.1 us

Low_Col := 0                                                  0.1 us
High_Col := 31                                                0.1 us
For K := 1 to 7 do  (* Binary search for first column *)
   begin
      Turn on Low_Col through High_Col
         chip column selects                             7 * 0.1 us
      if Some                                            7 * 0.1 us
         then  High_Col :=
            Low_Col + (High_Col - Low_Col) div 2         7 * 0.4 us
         else
            begin
               Mid := (High_Col - Low_Col) div 2         6 * 0.3 us
               Low_Col := High_Col + 1                   6 * 0.2 us
               High_Col := Low_Col + Mid                 6 * 0.2 us
            end
```

```
   end
                                                          -------------
                                                          4.4 to 6.2 us
(* The leftmost chip with a responder in the
   topmost row of cells is now selected      *)

J := 1                                                         Ø.1 us
X := Ø                                                         Ø.1 us
X := -W   (* Puts a one in the leftmost X of the row *)Ø.1 us
A := -X!                                                       Ø.1 us
X := Ø    (* Clears all of the other X bits *)                 Ø.1 us
A := -A!  (* That cell is now the only one active *)           Ø.1 us
X := Y                                                         Ø.1 us
while None do                                              7 * Ø.1 us
   begin
      X := 1                                               7 * Ø.1 us
      X := ØW~!  (* Move one cell to the right *)          7 * Ø.1 us
      A := X!    (* Make it the active one     *)          7 * Ø.1 us
      X := Y                                               7 * Ø.1 us
      J := J + 1                                           7 * Ø.1 us
   end
                                                          -------------
                                                          Ø.8 to 4.9 us

                                                          =============
                                                         11.3 to 22.4 us
```

The above timing analysis requires some further explanation. The minimum time for the binary search sections occurs when Some is true every time through the loop. This can occur when the first responder is in the top row of chips. The maximal time is obtained when Some is false for all but the last pass through the loop. (Because no search is performed on an array with no responders, at least one pass through the loop must have Some true.) Thus the true branch can be executed at most 7 times while the false branch can at most go through 6 iterations. The maximal case then counts 6 passes through the false branch

and one through the true branch. The timings for the on-chip searches assume for the minimal case that the while loop is skipped entirely, and for the maximal case that it must execute for the full seven times required to shift the activity pattern across the chip. Of course, even when the loop does not execute at all, one instruction time must be counted for the loop control test.

After the above algorithm has been run, the first responder is the only active cell in the single active chip. It should be noted that cells in other (deselected) chips may still have their activity bits set. The <X,Y> address of the cell is available in the controller as <X = High_Col * 8 + J, Y = High_Row * 8 + I>. If the <X,Y> location is not needed, the speed of the algorithm can be improved slightly by deleting the code associated with the I and J counters. The binary search segments of the algorithm will most likely be implemented with the loops expanded into a non-looping branching structure that includes all of the 64 possible results. This will save time in that the new bounds of the search will not have to be computed each time (they can be stored as constants for each of the branches) and because some of the searches only require six steps to complete instead of seven. (The algorithm given above forces all of the searches to take seven steps, in order to

simplify the coding for presentation purposes. In some cases a single row or column of chips is selected by the sixth iteration and the seventh is just a waste of time.) Use of the branching version of the binary search will reduce the times for this algorithm to a minimum of 5.Ø microseconds and a maximum of 13.8 microseconds.

Because the algorithm saves the original response and activity patterns in Z and B, respectively, it is easy to perform a "Discard First" operation and restore these patterns with the first responder removed. The code to do this is:

```
Z := Ø
Turn on all chip row selects
Turn on all chip column selects
A := 1!
X := Z
A := B
```

This takes Ø.6 microseconds. Once this has been done, the Select First operation can be reapplied to select the next responder. In this way a set of responders can be processed serially. Of course this defeats the parallelism of the CAAPP and thus should be avoided as much as possible. Whenever the set of responders is larger than 24ØØ, the time required to process them all serially will be longer than the 33 milliseconds in one video frame time.

Thus for large sets of responders requiring serial processing it will be faster to dump the contents of the array and allow the host processor to do the processing.

Add and Subtract Constant. These two algorithms are very straightforward on the CAAPP. This is because the design includes a full adder in each cell specifically to facilitate arithmetic operations. The two's complement of the broadcast constant is formed directly in the cell during the subtraction. This takes no more time than would otherwise be required and eliminates the need for doing this in the controller. In the following, "S" is the starting (low order) bit position of the field and L is its length. These algorithms do not change the activity pattern.

```
(* Add Constant *)

Z := Ø                                                   Ø.1 us
for I := Ø to L-1 do
   begin
      X := M(S + I)                                  L * Ø.1 us
      Y := C(I)                                      L * Ø.1 us
      X := X + Y                                     L * Ø.1 us
      M(S + I) := X                                  L * Ø.1 us
   end
X := Z                                                   Ø.1 us
M(S + L) := X                                            Ø.1 us
                                                 ----------------
                                                 Ø.3 + Ø.4 * L us

Sample time for 8 bit field                              3.5 us
```

```
(* Subtract Constant *)

Z := 1                                              0.1 us
for I := 0 to L-1 do
   begin
      X := M(S + I)                             L * 0.1 us
      Y := -C(I)                                L * 0.1 us
      X := X + Y                                L * 0.1 us
      M(S + I) := X                             L * 0.1 us
   end
X := Z                                              0.1 us
M(S + L) := X                                       0.1 us
                                          ----------------
                                          0.3 + 0.4 * L us

Sample time for 8 bit field                         3.5 us
```

In the above algorithms the memory bit just above the high order bit of the field is used to store the overflow condition. If the last carry (Z) value is one, then the addition resulted in an overflow or the subtraction resulted in an underflow. This representation leaves the resulting number in proper form. That is, the combination of the overflow bit and the field represent the proper result that would be obtained if the field were long enough to contain it. In many cases there is no need to save the overflow bit in memory. Quite often, in fact, it will be desired to simply transfer it into the A register and perform some appropriate operation on the offending cells (such as clearing their result fields). In either case, however, the timings will be the same. (It takes as long to transfer X to A as it does to transfer it to memory.)

<u>Add and Subtract Fields</u>. These two algorithms are very

similar to the preceding two. Instead of getting the value for the second operand from the broadcast constant input, the value is taken from a second field in the memory. Thus only one instruction is changed and the timings are identical. The overflow bit is also treated in the same way. The variables "S1" and "S2" are used to represent the starting (low order) bit positions of the two fields. The result will go in the field beginning at S1. The activity pattern is not changed by these algorithms.

```
(* Add Fields *)

Z := Ø                                              Ø.1 us
for I := Ø to L-1 do
   begin
      X := M(S1 + I)                            L * Ø.1 us
      Y := M(S2 + I)                            L * Ø.1 us
      X := X + Y                                L * Ø.1 us
      M(S1 + I) := X                            L * Ø.1 us
   end
X := Z                                              Ø.1 us
M(S1 + L) := X                                      Ø.1 us
                                            -----------------
                                            Ø.3 + Ø.4 * L us

Sample time for 8 bit field                         3.5 us


(* Subtract Fields *)

Z := 1                                              Ø.1 us
for I := Ø to L-1 do
   begin
      X := M(S1 + I)                            L * Ø.1 us
      Y := M(S2 + I)                            L * Ø.1 us
      Y := -Y                                   L * Ø.1 us
      X := X + Y                                L * Ø.1 us
```

```
        M(S1 + I) := X                                  L * 0.1 us
     end
  X := Z                                                    0.1 us
  M(S1 + L) := X                                            0.1 us
                                                  ----------------
                                                  0.3 + 0.5 * L us

  Sample time for 8 bit field                               4.3 us
```

Multiply by constant. This algorithm uses the standard technique of "shift and add" adapted to bit-serial operation. In this case the algorithm has been designed such that the result will appear in a separate field. (It is possible to write a multiplication algorithm that recycles some of the memory originally taken up by the multiplier field.) The result field will be equal in length to the sum of the lengths of the two operands. If this extended precision is not required, the algorithm can be modified to discard the low order bits in order to conserve memory space. In the following "SR" and "SM" are the starting (low order) bits of the result and multiplicand fields, respectively. "LC" and "LM" are the lengths of the constant and the multiplicand field. "C(I)" refers to the I-th bit of the constant multiplier. The activity pattern is not affected by this algorithm.

```
  (* Multiply by Constant *)

  for I := 0 to LC + LM-1 do
     M(SR + I) := 0 (* Clear result field *)(LC + LM) * 0.1 us
```

```
for I := Ø to LC-1 do
   begin
      if C(I) = 1                                       LC * Ø.1 us
         then
            begin
               Z := Ø                                   LC * Ø.1 us
               for J := Ø to LM-1 do
                  begin  (* add constant shifted *)
                     X := M(SM + J)           LC * LM * Ø.1 us
                     Y := M(SR + J + I)       LC * LM * Ø.1 us
                     Y := X + Y               LC * LM * Ø.1 us
                     M(SR + J + I) := Y       LC * LM * Ø.1 us
                  end
               X := Z                                   LC * Ø.1 us
               M(SR +LM + I) := X                       LC * Ø.1 us
            end
                                   ----------------------------------
maximum time (all C bits=1) Ø.1(LC+LM)+Ø.4(LC)+Ø.4(LC*LM) us
minimum time (all C bits=Ø)            Ø.1(LC+LM)+Ø.1(LC) us

if LM = LC (which is frequently the case):

maximum time (all C bits = 1)       Ø.6(LM) + Ø.4(LM*LM) us
sample time for 8 bits                                30.4 us
minimum time (all C bits = Ø)                     Ø.3 * LM us
sample time for 8 bits                                 2.4 us
```

The timing analysis of the above is based upon the fact that the shift and add algorithm does nothing when a bit of the multiplier is zero. In a bit-serial machine, every zero bit in the multiplier eliminates one bit-serial addition operation. Because this is where the most time is spent, the savings are great when the multiplier has only a few bits equal to one. In the worst case (all of the bits are one) the algorithm takes time proportional to LM squared. In the best and next best cases (all bits zero or all but one set to zero) the time is linear with respect to LM. The time can be reduced somewhat if the field sizes of the

constant multiplier and the multiplicand differ. In that situation there is no reason to pad the fields with extra high order zeroes in order to make their lengths equal. The algorithm, because of its bit-serial nature, can easily deal with fields of differing lengths and need not spend extra time manipulating extraneous high order zeroes.

Divide by comparand. Bit-serial division is much more complex than multiplication. The following algorithm uses the standard technique of aligning the divisor and then comparing and subtracting from the dividend while adding appropriate powers of two to the quotient. Three of the previously presented algorithms will be used here: Compare Greater or Equal, Subtract Constant, and Add Constant. In the following, "SQ" and "EQ" refer to the starting (low order) and ending (high order) bit positions of the quotient. "SD" and "ED" refer to the starting and ending bit positions of the dividend. "LF" is the length of these fields. The algorithm leaves the remainder in the dividend field.

```
(* Divide by Constant *)

for I := SQ to EQ do     (* Clear quotient *)
   M(I) := Ø
if (Constant = Ø) or (Constant >= 2**(LF-1))          LF * Ø.1 us
   then exit                                               Ø.1 us
```

```
K := 0                                                   0.1 us
while C(LF) <> 1 do     (* Align divisor *)
   begin
      Shift Constant left                       (LF-1) * 0.1 us
      K := K+1                                  (LF-1) * 0.1 us
   end
B := A!                                                  0.1 us
for I := K downto 0 do
   begin
      for J := LF-1 downto 0 do  (* Set activity if *)
         begin                   (* >= divisor     *)
            X := M(SD + J)                    LF * LF * 0.1 us
            Y := C(J)                         LF * LF * 0.1 us
            Z := 0                            LF * LF * 0.1 us
            A := -(X + Y)                     LF * LF * 0.1 us
         end
      X := 1                                       LF * 0.1 us
      A := B!                                      LF * 0.1 us
      A := X                                       LF * 0.1 us
      Z := 1                                       LF * 0.1 us
      for J := 0 to LF-1 do   (* Subtract divisor *)
         begin                (* from dividend    *)
            X := M(SD + J)                    LF * LF * 0.1 us
            Y := -C(J)                        LF * LF * 0.1 us
            X := X + Y                        LF * LF * 0.1 us
            M(SD + J) := X                    LF * LF * 0.1 us
         end
      X := Z                                       LF * 0.1 us
      M(ED + 1) := X                               LF * 0.1 us
      Z := 1                                       LF * 0.1 us
      Y := 0                                       LF * 0.1 us
      for J := SQ+I to EQ do   (* Add 2**I to   *)
         begin                 (* quotient      *)
            X := M(J)                   (LF-1)/2 * LF * 0.1 us
            X := X + Y                  (LF-1)/2 * LF * 0.1 us
            M(J) := X                   (LF-1)/2 * LF * 0.1 us
         end
      X := Z                                       LF * 0.1 us
      M(EQ + 1) := X                               LF * 0.1 us
      Shift Constant right (* Realign divisor  *)LF * 0.1 us
      A := B!              (* Restore activity *)LF * 0.1 us
   end
                           ------------------------------
                           1.35(LF*LF) + 0.2(LF) + 0.1 us
 Sample time for 8 bits                             94.1 us
```

As can be seen from the timing analysis for the above, bit-serial division is quite slow. Only the maximum time is

given. This assumes that the constant is one and must therefore be shifted and subtracted for every bit position of the dividend field. In actual use the algorithm will usually be faster because the constant will be larger and require less shifting in order to be aligned with the dividend. In fact a check by the controller for a divisor equal to one, simply setting the quotient equal to the dividend in that case, would greatly improve the worst case performance (but would also complicate the timing analysis). The one half term in the timing analysis deserves a word of explanation. This came about because the particular loop which generated it begins (in the worst case) by executing LF iterations. On each successive iteration of the outer loop, however, this number decreases by one. The average number of executions is thus (LF-1)/2. It should be noted that the final formula always results in a whole number of machine cycles despite this. As a side effect, this algorithm also leaves the original activity pattern intact upon completion.

Multiply Fields. This algorithm is similar to the multiply by constant algorithm in that it uses the technique of shifting and adding. Instead of using a broadcast constant as the multiplier controlling the shifts and adds, it uses a second field within each CAAPP cell for this.

Thus for each bit position in the multiplier field, the algorithm must activate only those cells that have the selected bit equal to one. Once this is done, addition of the shifted multiplicand can take place. In the following, "SR", "SM", and "SF" refer to the starting (low order) bit of the result, multiplier and multiplicand fields, respectively. "LM" and "LF" refer to the lengths of the multiplier and multiplicand fields.

```
(* Multiply Fields *)

B := A!        (* Save activity pattern *)                   Ø.1 us
for I := Ø to LM+LF-1 do
   M(SR+I) := Ø                                    (LM+LF) * Ø.1 us
for I := Ø to LM-1 do
   begin
      A := M(SM + I)    (* Activate cells with *)  LM * Ø.1 us
      Z := Ø            (* multiplier bit I = 1*)  LM * Ø.1 us
      for J := Ø to LF-1 do
         begin    (* Add shifted multiplicand *)
            X := M(SF + J)                    LF * LM * Ø.1 us
            Y := M(SR + J + I)                LF * LM * Ø.1 us
            Y := X + Y                        LF * LM * Ø.1 us
            M(SR + J + I) := Y                LF * LM * Ø.1 us
         end
      X := Z                                       LM * Ø.1 us
      M(SR + LF + I) := X                          LM * Ø.1 us
      A := B!           (* Restore activity *)     LM * Ø.1 us
   end
                  ----------------------------------------------
                  Ø.4(LM*LF) + Ø.1(LM+LF) + Ø.5(LM) + Ø.1 us

If LM = LF then:                 Ø.4(LF*LF) + Ø.7(LF) + Ø.1 us
Sample for 8 bits:                                      31.3 us
```

It should be noted that in the case where the sizes of

the multiplier and multiplicand fields are different that the smaller field should be the multiplier. This is because the overhead time for the outer loop is repeated for as many bit positions as are in the multiplier field. In the timing analysis expression this can be seen as the "0.5(LM)" term. All of the other terms are independent with respect to which field is which. The algorithm also leaves the original activity pattern in place upon completion (an unintentional side effect of the method employed).

Divide Fields. This algorithm, like the preceding one, performs an arithmetic operation between two fields within each CAAPP cell. It uses the same general technique for division as does the divide by constant algorithm. In this case, however, because the divisor is stored in each cell the alignment with the dividend must take place in the cells. This is further complicated by the fact that each cell may have a different divisor which will require a different shift factor for alignment. In the multiplication algorithms it was possible to avoid actually shifting operands by simply applying a bias to the bit address while doing the additions. This was only possible because the same amount of shift was applied to each cell. In this case, the shift factors being different, the divisor will have to actually be shifted.

Once the divisors are aligned, they are compared against the corresponding dividends and those cells in which the divisor is less than or equal to the dividend are made active. The divisor is then subtracted from the dividend, a suitable power of two is added to the quotient and then all of the divisors are shifted right in preparation for the next iteration. When all of the divisors have been shifted back to their original positions and the last compare, subtract and add cycle has been performed, the algorithm terminates with the remainder in the dividend field and activity restored to its original pattern. In the divide by constant algorithm a shift count was used to determine the bias for the single bit that would be added to the quotient on each iteration of the main loop. Although this could be used here, the combination of differing counts in each cell and having to decrement them bit-serially adds considerable complexity to the algorithm. Instead a fourth field is introduced, called the adder, which contains a single one bit that is shifted left when the initial divisor alignment is done and which then is shifted right with each iteration of the main loop. In this way the field always contains the appropriate constant to be added to the quotient, and provides a simple way of keeping track of when a divisor has shifted back to its original position. This reduces the overall time for the algorithm (versus using a counter) and

only slightly increases the memory requirements (in the counter arrangement, the counter must be stored twice -- once as a backup and also as a working copy). For an 8 bit division, the method used here requires only 2 bits more than the counter based method.

In the following, "SD", "SF", "SQ" and "SA" represent the starting (low order) bit positions of the divisor, dividend, quotient and adder fields, respectively. "ED", "EF", EQ", and "EA" represent the ending (high order) bit positions of the same fields. "LD" is the length of each field, except for the "ZD" field which is a single bit that stores a tag pattern indicating which cells have non-zero divisor fields.

```
B := A!                                                     Ø.1 us
for I := 1 to LD-1 do   (* Align divisor and   *)
   begin
      A := M(ED)     (* put a 1 in proper bit *)LD-1 * Ø.1 us
      M(SA + I -1) := 1   (* of adder field  *)LD-1 * Ø.1 us
      A := B!                                  LD-1 * Ø.1 us
      X := M(ED)                               LD-1 * Ø.1 us
      A := -X                                  LD-1 * Ø.1 us
      for J := ED downto SD+1 do
         begin  (* Shift divisor left *)
            X := M(J-1)                   (LD-1)**2 * Ø.1 us
            M(J)  := X                    (LD-1)**2 * Ø.1 us
         end
   end
M(EA) := 1                                                  Ø.1 us
A := B!                                                     Ø.1 us

for I := SQ to EQ do  (* Clear quotient *)
```

```
   M(I) := Ø                                                  LD * Ø.1 us

Y := 1         (* Find non-zero divisors *)                        Ø.1 us
Z := Ø                                                             Ø.1 us
for I := SD to ED do
   begin
      X := M(I)                                               LD * Ø.1 us
      X := X + Y                                              LD * Ø.1 us
   end
X := Z                                                             Ø.1 us
M(ZD) := X                                                         Ø.1 us

repeat
   A := B!                                                    LD * Ø.1 us
   for I := LD-1 downto Ø do
      begin  (* find dividends >= divisors *)
         X := M(I + SF)                                    LD**2 * Ø.1 us
         Y := M(I + SD)                                    LD**2 * Ø.1 us
         Z := Ø                                            LD**2 * Ø.1 us
         A := -(X + Y)                                     LD**2 * Ø.1 us
      end
   X := 1                                                     LD * Ø.1 us
   A := B!                                                    LD * Ø.1 us
   A := X                                                     LD * Ø.1 us
   A := M(ZD)  (* and divisors <> Ø *)                        LD * Ø.1 us

   Z := 1  (* Subtract divisor from dividend *)               LD * Ø.1 us
   for I := Ø to LD-1 do
      begin
         X := M(SF + I)                                    LD**2 * Ø.1 us
         Y := M(SD + I)                                    LD**2 * Ø.1 us
         Y := -Y                                           LD**2 * Ø.1 us
         X := X + Y                                        LD**2 * Ø.1 us
         M(SF + I) := X                                    LD**2 * Ø.1 us
      end
   X := Z                                                     LD * Ø.1 us
   M(SF + LD) := X                                            LD * Ø.1 us

   Z := Ø   (* Add adder to quotient *)                       LD * Ø.1 us
   for I := Ø to LD-1 do
      begin
         X := M(SQ + I)                                    LD**2 * Ø.1 us
         Y := M(SA + I)                                    LD**2 * Ø.1 us
         X := X + Y                                        LD**2 * Ø.1 us
         M(SQ + I) := X                                    LD**2 * Ø.1 us
      end
   X := Z                                                     LD * Ø.1 us
   M(EQ + 1) := X                                             LD * Ø.1 us

   A := B!  (* Shift adders right *)                          LD * Ø.1 us
```

```
    for I := SA to EA-1 do
       begin
          X := M(I + 1)                                  LD**2 * 0.1 us
          M(I) := X                                      LD**2 * 0.1 us
       end

    Y := 1  (* Set activity in cells with *)                LD * 0.1 us
    Z := 0  (* non-zero adders            *)                LD * 0.1 us
    for I := SA to EA do
       begin
          X := M(I)                                      LD**2 * 0.1 us
          X := X + Y                                     LD**2 * 0.1 us
       end
    X := Z                                                  LD * 0.1 us
    A := X                                                  LD * 0.1 us

    for I := SD to ED-1 do
       begin  (* Shift divisors right *)
          X := M(I + 1)                                  LD**2 * 0.1 us
          M(I) := X                                      LD**2 * 0.1 us
       end

    X := 1  (* Make all active cells responders *)LD * 0.1 us

until None (* No active cells indicates all of *)LD * 0.1 us
           (* the adders have shifted to zero  *)
A := B!    (* Restore activity pattern *)                 0.1 us
                                       ----------------------------
                                       2.1(LD**2) + 1.8(LD) + 0.5 us

Sample time for 8 bits:                                 149.3 us
```

As can be seen from the sample time, division of one field by another is a very slow process. The timing analysis gives the worst case time. This will frequently be the actual time because only one of the divisors need be equal to one in order to force the main loop to go through the maximum number of iterations. It might also be worthwhile to have a special test prior to entering the loop for small divisors (1 and 2, for example) and to quickly handle these cases in order to reduce the number of

iterations through the lengthy body of the main loop.

Unlike the previous arithmetic algorithms this one does not restore the activity pattern as a side effect. For the sake of consistency an extra instruction has been added specifically to restore the original activity pattern upon termination.

Simple Image Processing Operations

This section is devoted to implementations of some of the most common low level image processing and computer vision operations, which when implemented on the CAAPP will be viewed as the next higher level set of operations programmed in terms of the basic CAAPP operations. The algorithms in the previous section were strongly oriented towards the associative aspect of the architecture and almost completely ignored the topological arrangement of the processing array. The algorithms in this section will take the opposite approach, although through the low level operations the higher level algorithms will indirectly make use of the associativity.

The combination of associative and array processing is a very important feature of the architecture. In the case of

the low level algorithms, however, the associativity is mostly used as a substitute for having special dedicated logic to perform the low level functions in hardware. (Recall the CAPP examples in chapter 2.) The use of associativity to provide low level functions is no minor point for it is exactly this that makes it possible to fit so many processors onto a chip and thus to build an array with one processor per pixel. On the other hand, we shall see later that some very powerful operations result from combining associative operations with array operations in a different way. In those cases the associative operations will be employed at the same logical level as the array operations instead of simply being used for support.

We shall begin this section with a return to the "Game of Life" that was presented in Chapter 3 and see how this is implemented on the CAAPP. This will then be generalized to show how the CAAPP can do image convolution operations. We shall see two convolution based algorithms: A Gaussian smoothing filter and a Sobel edge detecting operation. These are common operations used in the preprocessing of images for computer vision systems. Another common image segmentation operation is the histogram, and we will see how the response count operation can be used to provide this.

An operation common to image generation is the rotation

of a model of an object and the generation of an image of the visible surface that this presents. This involves both a translation of the axes of the coordinate space in which the model is represented and the removal of all hidden surfaces. For rotation about a single axis this proves to be quite simple on the CAAPP and will serve as the first example of combining associative and array processing at the same level.

Game of Life. This problem was presented in chapter 3 where it was initially demonstrated that it could benefit from a CAM implementation. It was this problem that first motivated the author to consider the combination of associative processing and pixel per element array processing. Hence it should not be surprising that this algorithm would be used as a principal test of the architecture and that it would execute very quickly.

The following algorithm is highly optimized to take advantage of special properties of the Life grid. For example, knowledge of how the counting of neighboring cells can progress allows the use of partial field additions to speed up the counting. This algorithm uses bit zero of the memory to hold the status of the cell and bits one through four to hold the count of live neighbor cells. The "Shift_X" operations are used to represent the series of

eight Zig-Zag shifts necessary to move the entire array of response (X) bits in a particular direction. These thus count for 0.8 microseconds each in the timing analysis.

```
(* Conway's Game of Life *)

(* Initialize *)

A := 1!                                                   0.1 us
for N := 1 to 4 do  (* Zero counter *)
   M(N) := 0                                          4 * 0.1 us
X := M(0)     (* Load status into X *)                    0.1 us

(* Send to North *)

Shift_X_North                                             0.8 us
M(1) := X                                                 0.1 us

(* Send to Northwest *)

Shift_X_West                                              0.8 us
B := X̄                                                    0.1 us
Z := 0                                                    0.1 us
Y := M(1)                                                 0.1 us
Y := X + Y                                                0.1 us
M(1) := Y                                                 0.1 us
X := Z                                                    0.1 us
M(2) := X                                                 0.1 us
X := B                                                    0.1 us

(* Send to West *)

Shift_X_South                                             0.8 us
B := X̄                                                    0.1 us
Z := 0                                                    0.1 us
Y := M(1)                                                 0.1 us
Y := X + Y                                                0.1 us
M(1) := Y                                                 0.1 us
X := 0                                                    0.1 us
Y := M(2)                                                 0.1 us
Y := X + Y                                                0.1 us
M(2) := Y                                                 0.1 us
X := B                                                    0.1 us
```

```
(* Send to Southwest *)

Shift_X_South                                    Ø.8 us
B := X̄                                           Ø.1 us
Z := Ø                                           Ø.1 us
Y := M(1)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(1) := Y                                        Ø.1 us
X := Ø                                           Ø.1 us
Y := M(2)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(2) := Y                                        Ø.1 us
X := Z                                           Ø.1 us
M(3) := X                                        Ø.1 us
X := B                                           Ø.1 us

(* Send to South *)

Shift_X_East                                     Ø.8 us
B := X̄                                           Ø.1 us
Z := Ø                                           Ø.1 us
Y := M(1)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(1) := Y                                        Ø.1 us
X := Ø                                           Ø.1 us
Y := M(2)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(2) := Y                                        Ø.1 us
Y := M(3)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(3) := Y                                        Ø.1 us
X := B                                           Ø.1 us

(* Send to Southeast *)

Shift_X_East                                     Ø.8 us
B := X̄                                           Ø.1 us
Z := Ø                                           Ø.1 us
Y := M(1)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(1) := Y                                        Ø.1 us
X := Ø                                           Ø.1 us
Y := M(2)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(2) := Y                                        Ø.1 us
Y := M(3)                                        Ø.1 us
Y := X + Y                                       Ø.1 us
M(3) := Y                                        Ø.1 us
X := B                                           Ø.1 us
```

```
(* Send to East *)

Shift_X_North                  Ø.8 us
B := X                         Ø.1 us
Z := Ø                         Ø.1 us
Y := M(1)                      Ø.1 us
Y := X + Y                     Ø.1 us
M(1) := Y                      Ø.1 us
X := Ø                         Ø.1 us
Y := M(2)                      Ø.1 us
Y := X + Y                     Ø.1 us
M(2) := Y                      Ø.1 us
Y := M(3)                      Ø.1 us
Y := X + Y                     Ø.1 us
M(3) := Y                      Ø.1 us
X := B                         Ø.1 us

(* Send to Northeast *)

Shift_X_North                  Ø.8 us
B := X                         Ø.1 us
Z := Ø                         Ø.1 us
Y := M(1)                      Ø.1 us
Y := X + Y                     Ø.1 us
M(1) := Y                      Ø.1 us
X := Ø                         Ø.1 us
Y := M(2)                      Ø.1 us
Y := X + Y                     Ø.1 us
M(2) := Y                      Ø.1 us
Y := M(3)                      Ø.1 us
Y := X + Y                     Ø.1 us
M(3) := Y                      Ø.1 us
X := Z                         Ø.1 us
M(4) := X                      Ø.1 us

(* Evaluate Count *)

X := M(1)                      Ø.1 us
X := -X                        Ø.1 us
Y := M(3)                      Ø.1 us
X := X ^ Y                     Ø.1 us
Y := M(4)                      Ø.1 us
Y := -Y                        Ø.1 us
X := X ^ Y                     Ø.1 us
B := X                         Ø.1 us
Y := M(2)                      Ø.1 us
Y := -Y                        Ø.1 us
X := X ^ Y                     Ø.1 us
Y := M(Ø)                      Ø.1 us
X := X ^ Y                     Ø.1 us
```

```
Y := B                                    0.1 us
B := X                                    0.1 us
X := M(2)                                 0.1 us
X := X ^ Y                                0.1 us
Y := B                                    0.1 us
X := X v Y                                0.1 us
M(0) := X                                 0.1 us
                                        ---------
                                         17.4 us
```

This translates into 1915 Life generations in one video frame time, or 57471 generations per second, or over 3 million generations in one minute. By contrast, a one microsecond per operation serial processor would take over 9 days to run 3 million generations on a 512 by 512 field. Except as a demonstration (and the exploration of certain issues in automata theory) this is not an especially useful program, but it obviously generalizes to convolution algorithms.

Gaussian smoothing image convolution. Raw images often contain high frequency noise. This can be induced by limitations of the imaging system, by errors in image digitization and even by failures in storage media when images are held for later processing. Noise can also be a part of the actual image in the form of texture or specular reflection. The presence of high frequency noise can impair other image processing and analysis operations. Thus it is often desirable to first apply a low pass filter to an image in order to smooth out the noise. The Gaussian smoothing

operation described in this section is typical of those used to accomplish this.

The Game of Life involves a set of very simple image convolutions. More general convolution operators were further generalized with the design of the CAAPP.

A discrete, two dimensional, convolution is based on a mask of multipliers that each cell applies to its local neighborhood, forming the sum of the pairwise products of the cell's neighbors with their corresponding mask values. Typically the sum is then scaled in some manner, and the resulting value is used to update the value in the cell. The update operation must, of course, be performed after all cells have finished examining their neighborhoods. On a parallel processor this examination can be performed simultaneously by all of the cells on the grid, as can the update operation. Thus, the algorithm for the convolution can be described as the actions of a single cell with the understanding that each action is performed simultaneously by all of the cells.

There are two different ways of approaching the problem of examining the neighborhood. The one that first comes to mind is that each cell "looks" at each cell in its neighborhood, gathering what information it needs to perform

an update. In practice this involves moving data from each cell in the neighborhood into the "central" cell where some function is then applied and the result stored for the update phase of the convolution. The problem with this is that the data must, in the cases of all but the four nearest neighbors (on a four way connected grid), pass through other cells before it reaches the central cell. For example, when the neighborhood is seven by seven cells, data from the outer ring of cells must pass through at least two other cells before reaching the center cell. Because movement of data takes time, this "passing through" is rather inefficient. The solution is to have the data stored in the intermediate cells on its way through to the center, thus taking advantage of the fact that those cells will also need to know the values in order to compute the function of their neighborhoods. Although this will work, the algorithm becomes rather messy since it must now take into consideration the actions of several cells at once and how these relate to each other. It also becomes a complex problem to determine an optimal set of data collection paths as the neighborhood's diameter varies.

It turns out that the other approach to examining the neighborhood greatly simplifies these problems. This approach takes the opposite view of the collection process

-- in a sense it is its dual. Instead of each cell collecting all of the data from its neighborhood, each cell distributes its own data to every cell in the neighborhood. Because every other cell is also doing this, the end result is that the central cell (and hence all cells) gets the data it needs from all of the cells in the neighborhood. The problem of establishing an optimal distribution path is trivial for a square array of odd diameter: it is simply a rectangular spiral out from the center cell. For even diameter square neighborhoods the problem is only slightly more difficult because the center cell is actually half of a cell width off center in two directions. In this case it is simply required that the appropriate choice of initial direction and of clockwise or counter clockwise spiral be made to select the optimal path.

The only point that requires mentioning is that, because this is a distribution processs rather than a collection process, the function mask for the convolution must be mirrored across the central cell. For example, when the cell's value is being stored in its north neighbor, the south mask multiplier must be applied. This is easily seen when it is realized that the central cell is to the south of the cell to its north. The mirroring of the convolution mask is actually quite easy to accomplish: Simply step

through the mask array in exactly the opposite direction that the distribution path takes through the neighborhood array.

An example of this process can be seen in the algorithm for the Gaussian smoothing operation. In this convolution the center cell accumulates the sum of its neighbor's values, weighted inversely with distance. This sum is then normalized. The function mask is simply a three by three array of multiplier values that will be used to weight the values stored in the neighbors. The center cell will have a weight of four, the corner cells will have weights of one and the N, S, E, W cells will be weighted by two. If we define the neighborhood as an array M(i,j), Ø<=i<=2, Ø<=j<=2, where M(1,1) is the central cell, then the following algorithm will perform the convolution (M(Ø,Ø) is northwest):

```
i := 1
j := 1
sum := value * M[i,j]
move value north
i := i + 1
sum := sum + value * M[i.j]
move value east
j := j + 1
sum := sum + value * M[i.j]
move value south
i := i - 1
sum := sum + value * M[i,j]
```

```
move value south
i := i - 1
sum := sum + value * M[i,j]
move value west
j := j - 1
sum := sum + value * M[i,j]
move value west
j := j - 1
sum := sum + value * M[i,j]
move value north
i := i + 1
sum := sum + value * M[i,j]
move value north
i := i + 1
sum := sum + value * M[i,j]
value := sum * normalizing factor
```

It should be noted that the time required to perform a convolution using the parallel processor is independent of the size of the image (assuming the image is no larger than the array) and only dependent upon the area of the convolution mask. Since the CAAPP does cell level arithmetic bit-serially, the size of the data values also affects the speed of the algorithm.

The following program gives the list of instructions required to make the CAAPP perform the convolution given in the above example. In this case we have again, as with the Life algorithm, taken advantage of special characteristics of the values in the mask to help direct the shift and add process for the required multiplications. The variables used in the program are SV, SS and ST, the starting (low order) bit positions of the value, sum and temporary storage fields; ES, the end (high order bit) of the sum field; and

LF, the length of the value and temporary fields.

```
(* Gaussian smoothing convolution *)

(* Sum := value * 4 *)

for I := Ø to LF-1 do
   begin
      X := M(SV + I)                                    LF * Ø.1 us
      M(SS + I + 2) := X                                LF * Ø.1 us
   end
for I := SS + LF +1 to ES do
   M(I) := Ø                                     (LR-LF) * Ø.1 us

(* Send to north *)

Z := Ø                                                   Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(SV + I)                                    LF * Ø.1 us
      Shift_X_North                                     LF * Ø.8 us
      M(ST + I) := X                                    LF * Ø.1 us
      Y := M(SS + I + 1)                                LF * Ø.1 us
      Y := X + Y                                        LF * Ø.1 us
      M(SS + I + 1) := Y                                LF * Ø.1 us
   end
X := Ø                                                   Ø.1 us
for I := SS + LF+1 to ES do
   begin
      Y := M(I)                                (LR-LF-1) * Ø.1 us
      Y := X + Y                               (LR-LF-1) * Ø.1 us
      M(I) := Y                                (LR-LF-1) * Ø.1 us
   end

(* Send to Northwest *)

Z := Ø                                                   Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                                    LF * Ø.1 us
      Shift_X_West                                      LF * Ø.8 us
      M(ST + I) := X                                    LF * Ø.1 us
      Y := M(SS + I)                                    LF * Ø.1 us
      Y := X + Y                                        LF * Ø.1 us
      M(SS + I) := Y                                    LF * Ø.1 us
```

```
   end
X := Ø                                                  Ø.1 us
for I := SS + LF to ES do
   begin
      Y := M(I)                               (LR-LF) * Ø.1 us
      Y := X + Y                              (LR-LF) * Ø.1 us
      M(I) := Y                               (LR-LF) * Ø.1 us
   end

(* Send to West *)

Z := Ø                                                  Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                               LF * Ø.1 us
      Shift_X_South                                LF * Ø.1 us
      M(ST + I) := X                               LF * Ø.1 us
      Y := M(SS + I + 1)                           LF * Ø.1 us
      Y := X + Y                                   LF * Ø.1 us
      M(SS + I + 1) := Y                           LF * Ø.1 us
   end
X := Ø                                                  Ø.1 us
for I := SS + LF + 1 to ES do
   begin
      Y := M(I)                             (LR-LF-1) * Ø.1 us
      Y := X + Y                            (LR-LF-1) * Ø.1 us
      M(I) := Y                             (LR-LF-1) * Ø.1 us
   end

(* Send to Southwest *)

Z := Ø                                                  Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                               LF * Ø.1 us
      Shift_X_South                                LF * Ø.8 us
      M(ST + I) := X                               LF * Ø.1 us
      Y := M(SS + I)                               LF * Ø.1 us
      Y := X + Y                                   LF * Ø.1 us
      M(SS + I) := Y                               LF * Ø.1 us
   end
X := Ø                                                  Ø.1 us
for I := SS + LF to ES do
   begin
      Y := M(I)                               (LR-LF) * Ø.1 us
      Y := X + Y                              (LR-LF) * Ø.1 us
      M(I) := Y                               (LR-LF) * Ø.1 us
   end

(* Send to South *)
```

```
Z := Ø                                        Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                     LF * Ø.1 us
      Shift_X_East                       LF * Ø.8 us
      M(ST + I) := X                     LF * Ø.1 us
      Y := M(SS + I + 1)                 LF * Ø.1 us
      Y := X + Y                         LF * Ø.1 us
      M(SS + I + 1) := Y                 LF * Ø.1 us
   end
X := Ø                                        Ø.1 us
for I := SS + LF+1 to ES do
   begin
      Y := M(I)                 (LR-LF-1) * Ø.1 us
      Y := X + Y                (LR-LF-1) * Ø.1 us
      M(I) := Y                 (LR-LF-1) * Ø.1 us
   end

(* Send to Southeast *)

Z := Ø                                        Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                     LF * Ø.1 us
      Shift_X_East                       LF * Ø.8 us
      M(ST + I) := X                     LF * Ø.1 us
      Y := M(SS + I)                     LF * Ø.1 us
      Y := X + Y                         LF * Ø.1 us
      M(SS + I) := Y                     LF * Ø.1 us
   end
X := Ø                                        Ø.1 us
for I := SS + LF to ES do
   begin
      Y := M(I)                   (LR-LF) * Ø.1 us
      Y := X + Y                  (LR-LF) * Ø.1 us
      M(I) := Y                   (LR-LF) * Ø.1 us

(* Send to East *)

Z := Ø                                        Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                     LF * Ø.1 us
      Shift_X_North                      LF * Ø.8 us
      M(ST + I) := X                     LF * Ø.1 us
      Y := M(SS + I + 1)                 LF * Ø.1 us
      Y := X + Y                         LF * Ø.1 us
      M(SS + I + 1) := Y                 LF * Ø.1 us
   end
```

```
X := Ø                                                    Ø.1 us
for I := SS + LF+1 to ES do
   begin
      Y := M(I)                               (LR-LF-1) * Ø.1 us
      Y := X + Y                              (LR-LF-1) * Ø.1 us
      M(I) := Y                               (LR-LF-1) * Ø.1 us
   end

(* Send to Northeast *)

Z := Ø                                                    Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                                 LF * Ø.1 us
      Shift_X_North                                  LF * Ø.1 us
      Y := M(SS + I)                                 LF * Ø.1 us
      Y := X + Y                                     LF * Ø.1 us
      M(SS + I) := Y                                 LF * Ø.1 us
   end
X := Ø                                                    Ø.1 us
for I := SS + LF to ES do
   begin
      Y := M(I)                                 (LR-LF) * Ø.1 us
      Y := X + Y                                (LR-LF) * Ø.1 us
      M(I) := Y                                 (LR-LF) * Ø.1 us
   end

(* Normalize *)

for I := Ø to LF-1 do
   begin
      X := M(SS + I + 4)                             LF * Ø.1 us
      M(SV + I) := X                                 LF * Ø.1 us
   end
               ---------------------------------------------
               1Ø.7(LF) + 1.3(LR-LF) + 1.2(LR-LF-1) + 1.6 us

Because LR-LF = 4                            1Ø.7(LF) + 1Ø.4 us

Sample time for 8 bit pixels:                           96.Ø us
```

This represents a capability for performing over 340 of these convolutions or their equivalent in a single video frame time. Convolutions with more general and or larger masks will take longer than this time. A very rough worst

case estimate of the time required for such convolutions can be obtained from the formula: T = P(Ø.8N + Ø.2M + Ø.1) + Ø.3M(N**2 * P + N + 1). Where T is the time in microseconds, N is the number of bits in a pixel value, M is the number of bits in a mask value and P is the number of pixels in the mask area. This is a worst case time which assumes that all of the bits in all of the mask values are ones (since this gives the slowest multiply speed). In actuality, if it were known that all of the values are ones there is a shortcut that makes the timing comparable to the minimum. For the sake of obtaining an upper bound, however, we will ignore this. Under normal circumstances T will be about half of the value obtained from the formula. This also assumes a totally general square mask where the values are not known ahead of time.

If constants are to be used for the mask values a significant speed increase can be obtained by optimizing the multiplication operations for those values. Thus, for example, a convolution on 16 bit values with 8 bit mask values could be applied over at most a seven by seven mask in one video frame time, given the worst case situation. For normal situations it should be possible to convolve over a 1Ø by 1Ø area. Given constant mask values that allow considerable optimization, even a 25 by 25 area could be

convolved in one video frame time.

As a final note, this method is not restricted to square masks but can be further generalized for use with any mask shape. All that is required for this is an algorithm for efficiently shifting the center cell's value so that it covers the mask area. Thus it should be possible to easily adapt it to such mask shapes as annuli and disjoint areas. In fact the next algorithm will be a convolution with two masks which ignore a central stripe through the neighborhood.

Sobel edge extracting operation. Another common operation in image processing and computer vision systems is the reduction of an image to a set of lines that represent the edges in an image. The Sobel operator is one way of approaching this problem. It applies a convolution with a pair of masks (shown in figure 40) that result in each cell containing a gradient vector with components of magnitude and orientation. The resulting field of vectors is an approximation to the intensity gradient (first derivative) of the image. Cells which have strong gradient vectors are on or near points in the image that change abruptly in intensity and are thus likely candidates for being edges of regions.

-1	0	1
-2	0	2
-1	0	1

F(x)

1	2	1
0	0	0
-1	-2	-1

F(y)

Figure 40

Weight Values for the Sobel Operation

The following algorithm uses the same basic method as that of the convolution presented in the previous section. The vectors developed by the Sobel operator are in cartesian form. Rather than convolve each of the two masks required to generate the X and Y magnitudes, the algorithm distributes the center cell's value only once, computing both of the resultant values simultaneously. Even though a third of each of the two masks is empty, the non-empty areas of the masks overlap in such a way as to make it most efficient to distribute the value over the entire neighborhood. Only half of the neighbors need to apply both masks, the others use just one of the masks. In the following, SF, ST, XR and YR are the starting (low order) bits of the value, temporary, X-result and Y-result fields respectively. LF and LR are the lengths of the data (value, temporary) and result (X and Y) fields, respectively.

```
(* Sobel edge extraction operator *)

for I := Ø to LF-1 do    (* South *)
   begin
      X := M(SF + I)                                    LF * Ø.1 us
      Shift_X_South                                     LF * Ø.8 us
      M(ST + I) := X                                    LF * Ø.1 us
      M(YR + I + 1) := X                                LF * Ø.1 us
   end
M(YR + LF + 1) := Ø                                          Ø.1 us

Z := Ø                                                       Ø.1 us
for I := Ø to LF-1 do    (* Southwest *)
```

```
   begin
      X := M(ST + I)                                  LF * Ø.1 us
      Shift_X_West                                    LF * Ø.8 us
      M(ST + I) := X                                  LF * Ø.1 us
      M(XR + I) := X                                  LF * Ø.1 us
      Y := M(YR + I)                                  LF * Ø.1 us
      Y := X + Y                                      LF * Ø.1 us
      M(YR + I) := Y                                  LF * Ø.1 us
   end
X := Ø                                                     Ø.1 us
for I := LF to LR-1 do
   begin
      Y := M(YR + I)                             (LR-LF) * Ø.1 us
      Y := X + Y                                 (LR-LF) * Ø.1 us
      M(YR + I) := Y                             (LR-LF) * Ø.1 us
      M(XR + I) := Ø                             (LR-LF) * Ø.1 us
   end

Z := Ø                                                     Ø.1 us
for I := Ø to LF-1 do    (* West *)
   begin
      X := M(ST + I)                                  LF * Ø.1 us
      Shift_X_North                                   LF * Ø.8 us
      M(ST + I) := X                                  LF * Ø.1 us
      Y := M(XR + I + 1)                              LF * Ø.1 us
      Y := X + Y                                      LF * Ø.1 us
      M(XR + I + 1) := Y                              LF * Ø.1 us
   end
X := Ø                                                     Ø.1 us
Y := M(XR + LF + 1)                                        Ø.1 us
Y := X + Y                                                 Ø.1 us
M(XR + LF + 1) := Y                                        Ø.1 us

Z := Ø                                                     Ø.1 us
for I := Ø to LF-1 do    (* Northwest *)
   begin
      X := M(ST + I)                                  LF * Ø.1 us
      Shift_X_North                                   LF * Ø.8 us
      X := -X                                         LF * Ø.1 us
      M(ST + I) := X    (* store complement *)        LF * Ø.1 us
      X := -X                                         LF * Ø.1 us
      Y := M(XR + I)                                  LF * Ø.1 us
      Y := X + Y                                      LF * Ø.1 us
      M(XR + I) := Y                                  LF * Ø.1 us
   end
X := Ø                                                     Ø.1 us
for I := LF to LR-1 do
   begin
      Y := M(XR + I)                             (LR-LF) * Ø.1 us
      Y := X + Y                                 (LR-LF) * Ø.1 us
```

```
        M(XR + I) := Y                              (LR-LF) * Ø.1 us
     end

  Z := 1                                                    Ø.1 us
  for I := Ø to LF-1 do
     begin
        X := M(ST + I)                                 LF * Ø.1 us
        Y := M(YR + I)                                 LF * Ø.1 us
        Y := X + Y                                     LF * Ø.1 us
        M(YR + I) := Y                                 LF * Ø.1 us
     end
  X := 1                                                    Ø.1 us
  for I := LF to LR-1 do
     begin
        Y := M(YR + I)                              (LR-LF) * Ø.1 us
        Y := X + Y                                  (LR-LF) * Ø.1 us
        M(YR + I) := Y                              (LR-LF) * Ø.1 us
     end

  Z := 1                                                    Ø.1 us
  for I := Ø to LF-1 do   (* North *)
     begin
        X := M(ST + I)                                 LF * Ø.1 us
        Shift_X_East                                   LF * Ø.8 us
        M(ST + I) := X                                 LF * Ø.1 us
        Y := M(YR + I + 1)                             LF * Ø.1 us
        Y := X + Y                                     LF * Ø.1 us
        M(YR + I + 1) := Y                             LF * Ø.1 us
     end
  X := 1                                                    Ø.1 us
  Y := M(YR + LF + 1)                                       Ø.1 us
  Y := X + Y                                                Ø.1 us
  M(YR + LF + 1) := Y                                       Ø.1 us

  Z := 1                                                    Ø.1 us
  for I := Ø to LF-1 do   (* Northeast *)
     begin
        X := M(ST + I)                                 LF * Ø.1 us
        Shift_X_East                                   LF * Ø.8 us
        M(ST + I) := X                                 LF * Ø.1 us
        Y := M(XR + I)                                 LF * Ø.1 us
        Y := X + Y                                     LF * Ø.1 us
        M(XR + I) := Y                                 LF * Ø.1 us
     end
  X := 1                                                    Ø.1 us
  for I := LF to LR-1 do
     begin
        Y := M(XR + I)                              (LR-LF) * Ø.1 us
        Y := X + Y                                  (LR-LF) * Ø.1 us
        M(XR + I) := Y                              (LR-LF) * Ø.1 us
```

```
   end
Z := 1                                                  Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                               LF * Ø.1 us
      Y := M(YR + I)                               LF * Ø.1 us
      Y := X + Y                                   LF * Ø.1 us
      M(YR + I) := Y                               LF * Ø.1 us
   end
X := 1                                                  Ø.1 us
for I := LF to LR-1 do
   begin
      Y := M(YR + I)                          (LR-LF) * Ø.1 us
      Y := X + Y                              (LR-LF) * Ø.1 us
      M(YR + I) := Y                          (LR-LF) * Ø.1 us
   end

Z := 1                                                  Ø.1 us
for I := Ø to LF-1 do   (* East *)
   begin
      X := M(ST + I)                               LF * Ø.1 us
      Shift_X_South                                LF * Ø.8 us
      M(ST + I) := X                               LF * Ø.1 us
      Y := M(XR + I + 1)                           LF * Ø.1 us
      Y := X + Y                                   LF * Ø.1 us
      M(XR + I + 1) := Y                           LF * Ø.1 us
   end
X := 1                                                  Ø.1 us
Y := M(XR + LF + 1)                                     Ø.1 us
Y := X + Y                                              Ø.1 us
M(XR + LF + 1) := Y                                     Ø.1 us

Z := 1                                                  Ø.1 us
for I := Ø to LF-1 do  (* Southeast *)
   begin
      X := M(ST + I)                               LF * Ø.1 us
      Shift_X_South                                LF * Ø.8 us
      X := -X                                      LF * Ø.1 us
      M(ST + I) := X   (* complement *)            LF * Ø.1 us
      X := -X                                      LF * Ø.1 us
      Y := M(XR + I)                               LF * Ø.1 us
      Y := X + Y                                   LF * Ø.1 us
      M(XR + I) := Y                               LF * Ø.1 us
   end
X := 1                                                  Ø.1 us
for I := LF to LR-1 do
   begin
      Y := M(XR + I)                          (LR-LF) * Ø.1 us
      Y := X + Y                              (LR-LF) * Ø.1 us
      M(XR + I) := Y                          (LR-LF) * Ø.1 us
```

```
   end
Z := Ø                                                  Ø.1 us
for I := Ø to LF-1 do
   begin
      X := M(ST + I)                               LF * Ø.1 us
      Y := M(YR + I)                               LF * Ø.1 us
      Y := X + Y                                   LF * Ø.1 us
      M(YR + I) := Y                               LF * Ø.1 us
   end
X := Ø                                                  Ø.1 us
for I := LF to LR-1 do
   begin
      Y := M(YR+I)                           (LR-LF) * Ø.1 us
      Y := X + Y                             (LR-LF) * Ø.1 us
      M(YR + I) := Y                         (LR-LF) * Ø.1 us
   end
                              ------------------------------
                              11.9(LF) + 2.5(LR-LF) + 2.7 us

Because (LR-LF) = 2:                     11.9(LF) + 7.7 us

Sample time for 8 bit pixels:                     1Ø2.9 us
```

A time of 1Ø3 microseconds translates to roughly one three-hundredth of a video frame time, which leaves plenty of time for other operations to be performed.

<u>Histogram</u>. A particularly useful probe for evaluating images is the histogram. This is used in such processes as contrast adjustment, region splitting, region formation, straight line detection, etc. Typically a histogram is formed that represents the frequency with which a local image event or image feature occurs, and peaks in the histogram often signify the possibility that a global event of interest may have taken place. Let us consider the case of region formation for illustrative purposes. Peaks in a histogram of pixel values (intensity or R, G, B) indicate

those values that occur most often, and are thus likely to be part of contiguous regions with the same values. Valleys in the histogram indicate values that occur infrequently and are thus likely to be found on edges or transition zones.

The following algorithm uses the "Select Less Than" algorithm and the "Count Responders" algorithm, both given above. The algorithm simply selects ranges of values, starting with the lowest range and working up to the highest range. A particular range of values is called a bucket because in a serial algorithm each pixel would be individually examined and then dropped into the appropriate "bucket" (an accumulating counter variable). In the following, Max is the greatest value, Size is the number of values in the range assigned to each bucket and High holds the current maximum value for the bucket being filled. Buckets is the number of buckets in the histogram and the array Bucket[1..Buckets] holds the values that make up the histogram.

```
(* Histogram *)

Y := 1                                                          0.1 us
Size := Max div Buckets                                         0.1 us
for I := 1 to Buckets do
   begin
      High := I * Size                                Buckets * 0.1 us
      Select Cells < High                     Buckets * (0.4*L + 0.3) us
```

```
      X := X ^ Y                                    Buckets * 0.1 us
      Bucket[I] := Response Count                  Buckets * 26.8 us
      X := -X                                       Buckets * 0.1 us
      Y := X ^ Y                                    Buckets * 0.1 us
   end
                                      ---------------------------------
                                      Buckets * (0.4*L + 27.5) + 0.2 us

Sample time for 256 buckets
   and L = 8 bits:                                         7859.4 us

Sample time for 64 buckets (L = 8)                         1964.8 us
```

For eight bit pixels the formula above gives a time of 30.7 microseconds per bucket. This is only about 30 times faster than could be achieved by a fast serial processor. This lower speedup factor can be attributed to the fact that the response count hardware of the CAAPP was designed with both speed and reduced complexity and cost in mind. Even if the response count took only 0.1 microsecond to compute, the time for the 256 bucket histogram would only be reduced by a factor of eight (to a little under one millisecond). To speed up the histogram any further will require increasing the speed of the other parts of the algorithm. Because the select-less-than operation consumes the most time outside of the actual count, architectural enhancements to increase the speed of the inequality test will help the most. Another way of increasing the speed of the histogram would be to change the architecture so that the count could be formed in parallel with other operations in the array. Currently all array operations must wait for the count to be completed.

It should be possible to change the architecture so that the response count logic can latch a copy of the response set and then the array can go on to prepare the next response set in parallel with the formation of the count. In the histogram algorithm, the select-less-than operation for the next bucket could then take place in parallel with the count of the previous bucket. Thus the histogram time would be reduced to just the time required to count each bucket.

Rotate 3-D model and extract frontal surface. In this section we will switch from image processing operations to an image generation example. A common operation in the computer aided design of an object is to rotate the object and produce an image of the surface that is then turned toward the observer, which seems like a simple operation until it is realized that for complex objects this means that various parts of the object may be occluding others. Determining which parts are visible and which are hidden becomes a time consuming problem. For a simple rotation about any single axis this is, however, a simple problem for the CAAPP.

In most computer aided design systems an object is modelled as a collection of polygons that define an approximation to its surface, which is done to reduce the size of the data set needed to represent the object. In

order to perform operations such as slicing or cutaway viewing the design system must interpolate the appearance. The processing can become rather complicated when an object has internal detail that must be included. With the CAAPP, on the other hand, an object can be modelled directly in a 512 by 512 by 512 cube (actually the third dimension can be any size, but it is easiest to talk about a cubic volume). The entire physical structure of the object is mapped into a series of planes that cut through it in the X-Z direction. Each cell of this model contains a set of values describing the material that would occupy the same volume in the actual object. For example the cell might contain values for the color, transparency, reflectivity and index of refraction for the material. It would then be possible to slice the object in any way and apply a function representing the desired lighting situation and extract the image of the exposed surface.

In the CAAPP such a model is operated on one plane at a time. Each plane along the Y axis is loaded into the CAAPP, manipulated and then replaced by the next. As can be seen from just the time required to load 512 planes of data at 30 milliseconds each, the minimum time to process a model will be 15.36 seconds. Although this is not exceptionally fast, as compared to many computer aided design systems, the

additional versatility embodied in a direct representation model will make this approach useful in some situations. The important point to note is the simplicity of the algorithm employed. It is interesting to see how, using such a simple technique, the CAAPP can develop a result so quickly as compared to a serial processor.

Before the frontal surface algorithm can be presented one more elementary operation must be developed. This is the operation of loading the X-Y coordinate grid into the cells of the CAAPP. Each of the two coordinates is a nine bit number (log2(512) = 9) in which the upper 6 bits represent the chip address and the lower 3 bits represent the column or row address within a chip. The lower 3 bits are first developed by passing a set of masks across the chips, first in one direction then in the other, using the on chip neighbor communications network. The high order six bits are loaded by setting an appropriate series of activity patterns in the chip row and column select lines and writing ones into the corresponding memory bits. In the following, SX and SY are the starting (low order) bit positions of the X and Y coordinates, respectively. EX and EY represent the corresponding ending (high order) bit positions of those fields. The two fields are placed contiguously in memory with X below Y because this allows the entire 18 bit

combination of the two fields to be used as the address of the cell when the array is treated as a linear structure.

```
(* Load X-Y coordinate grid *)

Turn on all chip row selects                          Ø.1 us
Turn on all chip column selects                       Ø.1 us
A := 1!                                               Ø.1 us
for I := SX to EY do
   M(I) := Ø                                   18 * Ø.1 us

X := 1                                                Ø.1 us
for I := 1 to 7 do
   begin
      X := E                                    7 * Ø.1 us
      X := -X                                   7 * Ø.1 us
   end
M(SX) := X                                            Ø.1 us
X := 1                                                Ø.1 us
for I := 1 to 3 do
   begin
      X := E                                    3 * Ø.1 us
      X := E                                    3 * Ø.1 us
      X := -X                                   3 * Ø.1 us
   end
M(SX + 1) := X                                        Ø.1 us
X := 1                                                Ø.1 us
X := E                                                Ø.1 us
X := E                                                Ø.1 us
X := E                                                Ø.1 us
X := E                                                Ø.1 us
X := -X                                               Ø.1 us
M(SX + 2) := X                                        Ø.1 us

X := 1                                                Ø.1 us
for I := 1 to 7 do
   begin
      X := N                                    7 * Ø.1 us
      X := -X                                   7 * Ø.1 us
   end
M(SY) := X                                            Ø.1 us
X := 1                                                Ø.1 us
for I := 1 to 3 do
   begin
```

```
      X := N                                               3 * Ø.1 us
      X := N                                               3 * Ø.1 us
      X := -X                                              3 * Ø.1 us
   end
M(SY + 1) := X                                                 Ø.1 us
X := 1                                                         Ø.1 us
X := N                                                         Ø.1 us
X := N                                                         Ø.1 us
X := N                                                         Ø.1 us
X := N                                                         Ø.1 us
X := -X                                                        Ø.1 us
M(SY + 2) := X                                                 Ø.1 us

for I := 1 to 6 do
   begin
      Set chip column select register to 'DDDDDD'          6 * Ø.1 us
      Set position I of column register to '1'             6 * Ø.1 us
      M(SX + 2 + I) := 1                                   6 * Ø.1 us
   end
Turn on all chip column selects                                Ø.1 us

for I := 1 to 6 do
   begin
      Set chip row select register to 'DDDDDD'             6 * Ø.1 us
      Set position I of row register to '1'                6 * Ø.1 us
      M(SY + 2 + I) := 1                                   6 * Ø.1 us
   end
Turn on all chip row selects                                   Ø.1 us
                                                             --------
                                                              12.7 us
```

This is a reasonably short time to load an 18 bit field into the entire array. In actuality the above algorithm will be a little faster because its implementation in controller microcode will unravel the last two loops and use constants for selecting the chip rows and columns. This will save 1.2 microseconds and bring the time down to 11.5 microseconds. If cell rows and columns were individually selectable the time could be reduced to 5.6 microseconds. If the cells had their on-chip addresses somehow hardwired

into them the time would be 3.8 microseconds. Note, however, that the latter modification would either require an extended address space in the cells or would reduce the amount of general purpose memory available by 6 bits.

Now that we have this operation defined we can proceed to the algorithm for rotating an object and extracting a frontal view. As stated above, the CAAPP processes the model one plane at a time. This algorithm will load each plane, rotate it about the <0,0> axis and then return the edge on view of that slice of the model. Because of the associative nature of the CAAPP there is no need to actually move the data in order to accomplish the rotation. Instead the trigonometric formula for vector rotation is applied to the X-Y coordinate grid (as loaded by the previous algorithm). Then for each plane in the 3-D model we load the plane in, select each X coordinate in the rotated coordinate grid one at a time and for each of these apply the Select Greatest operation to find the Y coordinate containing a model element that is nearest the front of the plane.

By assembling the array of pixels that this scanning operation produces into a row for each plane, an image of the rotated object results. In the following, X1, Y1 are the fields containing the original X-Y grid. X2 and Y2

contain the rotated grid. X3 is a temporary field used to compute the rotated coordinates. Theta is the angle of rotation. Xmin and Xmax contain the least and greatest values of X in the rotated coordinate system. SM and EM are the starting (low order) and ending (high order) bits of the model value. The field MP indicates whether there is a model value present in each cell. Note that the coordinate grid only has to be rotated once. After that each plane is simply overlayed onto it.

```
(* Rotate model and extract frontal view *)

Store X-Y coordinates in X1, Y1                              12.7 us
X2 := X1 * Cos(Theta)    (* 9 bit multiply const *)    37.8 us
Y2 := Y1 * Sin(Theta)    (* 9 bit multiply const *)    37.8 us
X2 := X2 - Y2            (* 9 bit subtract fields*)     3.9 us
Y2 := Y1 * Cos(Theta)    (* 9 bit multiply const *)    37.8 us
X3 := X1 * Sin(Theta)    (* 9 bit multiply const *)    37.8 us
Y2 := Y2 + X3            (* 9 bit add fields     *)     3.9 us

if Theta <= 90
   then
      begin
         Xmin := -512 * Sin(Theta)
         Xmax := 512 * Cos(Theta)
      end
else if Theta <= 180
   then
      begin
         Xmin := 512 * Cos(Theta) - 512 * Sin(Theta)
         Xmax := 0
      end
else if Theta <= 270
   then
      begin
         Xmin := 512 * Cos(Theta)
         Xmax := -512 * Sin(Theta)
```

```
        end
  else
     begin
        Xmin := 0
        Xmax := 512 * Cos(Theta) - 512 * Sin(Theta)
     end                                                  0.5 us

  for I := 0 to 512 do
     begin
        load model plane I                           K1 * 30000.0 us
        for XL := Xmin to Xmax do
           begin
              A := 1!                                   K2 * 0.1 us
              Exact match X2 = XL      (* 9 bits *)  K2 * 4.7 us
              A := X                                    K2 * 0.1 us
              A := M(MP)                                K2 * 0.1 us
              Find least Y2            (* 9 bits *)  K2 * 2.9 us
              A := X                                    K2 * 0.1 us
              for J := SM to EM do
                 begin
                    X := M(J)                           K3 * 0.1 us
                    Image_Bit[I,XL,J] := Some           K3 * 0.1 us
                 end
           end
     end
                              --------------------------------------
                              30000(K1) + 8(K2) + 0.2(K3) + 172.2 us

  K1 = 512
  K2 = (Xmax - Xmin) * 512  (Maximum = 724 * 512 = 370688)
  K3 = LM * (Xmax - Xmin) * 512

  Thus T =  (4096 + 102.4(LM)) * (Xmax - Xmin) + 15360172.2 us

  Sample time for 8 bit model:                       21291180.2 us
                                               (21.291182 seconds)
```

It should be noticed that over half of this time is spent in loading the model planes into the CAAPP. If it were possible to load the memory with a new plane simultaneously with the processing of the current one, there would be about 9 seconds of free time during which further processing could be performed on the model (such as application of lighting

effects, cutaway viewing and so on). As mentioned above, this time seems a bit long as compared to the best computer aided design systems, but the versatility afforded by the modelling scheme should make it possible to perform operations (such as analysis of deformation under stress) that would take a serial processor much longer to compute with the same accuracy and level of detail. One of the features of the advanced designs presented in the next chapter is a faster image load time, which should greatly speed up this algorithm. It is, of course, the data transfer time that presents the greatest problem in any direct cubic representation processing: The amount of data contained in a 512 by 512 by 512 cube is so great that it takes a considerable amount of time to move it into and out of the array. That the transfer can be done as quickly as it is, in this design, is really a credit to the architecture. If the CAAPP were to be seriously applied to image generation problems, however, more compact model representation mechanisms would have to be developed.

Higher Level Image Processing Operations

In the preceding section we examined some of the

operations used in low level image processing. In this section we will see some examples of how the CAAPP can be used to perform more complex image processing tasks. The first that we will examine is the connected components region labelling operation. The second operation that we will look at is the extraction of a field of flow vectors from a sequence of images which are obtained from a moving sensor.

Because these operations are more complex than those presented earlier, they will be presented using language that hides the mass of details required to implement them. This is equivalent to using procedure or subroutine calls in a normal program. The result is that the timing for an algorithm will seem to be much greater than its length would imply, due to the fact that the elementary operations require more time to execute. If all of the details were presented directly in these algorithms, they would have been several pages in length and the overall design completely hidden. Although slight variations may be used, these elementary operations have all been described in the preceding sections and the reader should consult them if any questions arise.

In writing serial programs the IF-THEN-ELSE statement is the easiest way to represent a two-way branch in the control

flow. In the CAAPP, both branches of an IF-THEN-ELSE will be executed whenever it one encountered because some of the elements will respond to the THEN part and some will respond to the ELSE part. In the CAAPP a branch in control flow is implemented by a change in the activity mask for the array. The old mask must be saved and then restored when the branch rejoins the main control flow path. For the ELSE clause of an IF-THEN-ELSE statement the mask must be restored to its original pattern and the inverse conditional test applied to establish the pattern that is the reverse of that used in the THEN branch. Note that this parallel IF-THEN-ELSE is really implementing the branch at the processing element level (as opposed to controller level branches). It is possible that there will be no elements that require processing by one or the other of the branches and this could be checked for with a Some/None test. With a CAAPP this large, however, the probability that at least one element will need to be processed by each branch is so great, that more time is likely to be lost in Some/None testing than would be lost due to the occasional needless execution of a branch.

Region growing and labelling. One very basic image processing operation is the labelling of regions via connected components algorithms, where adjacent pixels are

given the same label if they are similar, or have no edges between them.

First the algorithm will be presented in pseudo-code form to make it easier to understand. In the following algorithm the status bit holds a one if a cell is part of an edge and zero otherwise. The label for a cell is initially the 18 bit concatenation of its X and Y coordinates on the grid. This is the simplest way of starting each cell with a unique label and thus ensures that the regions that eventually result will have unique labels also.

```
(* Region Growing / Labelling Pseudocode *)

Copy status bit from each neighbor into own memory.
Load X-Y grid into cells.
repeat
   Clear Label-Changed bit
   for each neighbor (N,S,E,W) do
      if Neighbor-Status is not Boundary
         then
            if Own-Label is greater than Neighbor's-Label
               and Own-Status is not Boundary
               then
                  Copy Neighbor's-Label into Own-Label
                  Set Label-Changed bit
until no Label-Changed bits are set
```

In the following, "Status" is the same as Own-Status, "Change" is the same as Label-Changed and "Label" is the same as Own-Label above. The names "S_Status", "E_Status", "W_Status", "N_Status" and "Neighbor_Status" refer to the

four bits in local memory where copies of the the status bits from the cell's four neighbors are stored. (The status bits are stored locally to increase the speed with which they can be accessed.) There is one additional field, called "Temp" which is used to hold a copy of a neighbor's label field for comparison purposes and to save time in case the neighbor's label must be copied into the cell's own label field.

```
(* Region Growing and Labelling *)

X := M(Status)                                                  Ø.1 us
X := -X          (* Set up NON-edge activity pattern *)         Ø.1 us
M(Status) := X                                                  Ø.1 us
Shift_X_North                                                   Ø.8 us
M(S_Status) := X                                                Ø.1 us
X := M(Status)                                                  Ø.1 us
Shift_X_South                                                   Ø.8 us
M(N_Status) := X                                                Ø.1 us
X := M(Status)                                                  Ø.1 us
Shift_X_East                                                    Ø.8 us
M(W_Status) := X                                                Ø.1 us
X := M(Status)                                                  Ø.1 us
Shift_X_West                                                    Ø.8 us
M(E_Status) := X                                                Ø.1 us
Load X-Y Grid                                                  12.7 us

repeat
   M(Change) := Ø                                           D * Ø.1 us
   for Neighbor := N to W do
      begin
         A := M(Neighbor_Status)!                       D * 4 * Ø.1 us
         Copy Label from Neighbor to Temp              D * 4 * 18.Ø us
         Compare Label > Temp                           D * 4 * 7.5 us
         A := X                                         D * 4 * Ø.1 us
         A := M(Status)                                 D * 4 * Ø.1 us
         Copy Temp to Label                             D * 4 * 5.4 us
         M(Change) := 1                                 D * 4 * Ø.1 us
```

```
      end
    X := M(Change)                                       D * Ø.1 us
until None                                               D * Ø.1 us
                                                   -------------------
                                                   D * 125.5 + 16.9 us
```

In the above timing analysis, D represents the number of iterations through the repeat loop required to complete the labelling of all of the cells in the image. If, for each region, we compute the distance from the cell whose original label eventually becomes the label for the region containing it, to the cell furthest from it in the region, then the greatest of these distances is equal to D (This might be considered as form of diameter measure for the region.). This distance is somewhat hard to estimate. As a general rule, the distance is the number of cells through which the label passes when traversing the path to the most distant cell. In actuality it will usually be less than this number because a label will pass through two cells per iteration in many cases. In the best case the distance will be half of the number of cells on the path.

The reason for storing the inverse of the neighbor's status bits is that what we a really want to do is create an activity mask that excludes all edge cells. Among other things, this prevents the edge cells from changing labels. Once the regions have been labelled, it is easiest to label the edges with some function that combines the labels of the

regions that they separate. It would be impractical to try to label edges with an algorithm like the above because so many of the edges are connected together that the result would be a few very long many branched edges and a few very short edges (the latter enclosing isolated regions). Because of these long edges, the execution time of the algorithm would be quite long.

The maximum execution time for the region growing algorithm would be something on the order of 16.5 seconds. This would be for a degenerate case in which the distance across a region is equal to half of the number of cells in the image. In this case the "edge" would be a rectangular spiral starting at the center and expanding with a single cell width gap between each successive wrap. The "region" would then be the corresponding spiral formed by the cells in the gap and would be over 131 thousand cells long. Another such worst case pattern would be a zig-zag figure, connecting every other row or column. A more likely time is based on D being equal to the width of the image. This gives a time of 64.2729 milliseconds, or just under two video frame times. If the region width is no more than half the width of the image, the time will be under a frame time (32.1449 milliseconds).

One strategy that might help reduce the execution time

for this algorithm would be to preprocess the largest regions associatively. Based upon a histogram analysis it should be possible to identify some feature (such as a particular color) that occupies a large amount of the image. All of the cells that contain this feature could then be prelabelled as a single region. This might be done for some small number of features. Region growing could then be done for the remainder of the image. It should also be possible to write a version of the region growing that would grow for up to some fixed distance and then "un-label" any areas that haven't been filled in completely. These could then be labelled separately according to some feature. However it is done, it should be possible to reduce the time for this algorithm to the point where a fully labelled segmented image can be developed from raw data in under a video frame time. This is, of course, one of the primary requirements for a system that can perform real time image analysis.

Decomposition of rotational and translational motion parameters from optic flow. The major goal in motion processing is the recovery of motion parameters of the sensor and each independently moving object in the field of view. The computation of environmental depth of visible surfaces then follows in a relatively straightforward

manner. This generally involves two stages of processing: computation of a feature displacement field, followed by inference of motion parameters and environmental depth.

The set of image displacements from two or more images is an approximation to optic flow. During this stage of the processing one faces the well-known correspondence problem, which involves the matching of corresponding image points of an environmental feature in the pair of images. The second stage involves inference of environmental information from the optic flow or the displacement field. This becomes a problem of separating the rotational and translational components of the flow field.

Rotation of the sensor induces image displacements that are a function only of the rotational parameters and image position; in particular the feature displacement between images is not a function of the depth of its environmental surface point.

The translational motion of the sensor, on the other hand, carries all of the environmental cues. For purely translational motion, the image displacement paths are determined by radial flow lines emanating from a single point in the image plane; that point is the intersection of the translational axis with the image plane (also referred

to as the focus of expansion -- FOE). The size of the displacements along these paths are a function of environmental depth and the distance from the FOE. Thus the problem of general motion becomes one of decomposing the rotational and translational effects of motion, and then using the image displacements from the instantaneous component of translational motion to compute depth. The following algorithm is the result of the author's collaboration with M. Steenstrup and D. Lawton [112].

Our algorithm is an exhaustive search procedure which uses a set of rotational and translational flow field templates to find a component pair that can account for the motion depicted in a given flow field. A total of 1000 rotational and 200 translational templates are used. These are generated from 100 axes which are uniformly distributed with respect to a unit hemisphere, and all pass through the origin of the sensor coordinate system, ie. the orientation of the axes of translation and rotation are specified as directions on the unit sphere. Each flow field consists of 16 by 16 vectors and is stored on a 2 by 2 chip segment of the CAAPP. Each flow field template thus occupies 256 CAAPP cells. The division of the CAAPP into 2 by 2 cell groups facilitates addressing of the flow field templates. Each cell contains the horizontal and vertical components of a

flow vector, each specified to 10 bits of precision.

The algorithm steps will be described at a high level and not expanded to single instruction detail. The algorithm is sufficiently complex that it is too slow to run on the actual CAAPP instruction level simulator. Instead it has been simulated by a special purpose program that took time saving short cuts while remaining constrained to macro operations that could be performed on the CAAPP. The timing estimate was built up from known timings for the macro operations and the timing of the simulation.

The basic concept of the algorithm is to exhaustively search all of the combinations of rotational and translational components represented by the template sets, selecting the pair that gives the best correlation with the input flow field. For the template sets that we have chosen, this represents a total of 200,000 combinations that must be searched. With a serial machine, such a search would take far too much time. With the CAAPP, however, the array processing capabilities can be used to perform the necessary computations on many templates in parallel, while the associative processing capabiities can be used to quickly determine which template gives the best match.

In our algorithm, the rotational templates are each

represented by a 16 by 16 element array of X and Y vector components. The entire set of 200 rotational templates is loaded into the CAAPP at the start of the algorithm by using 256 by 200 CAAPP cells. The input flow field, also represented by a 16 by 16 array of flow vectors (a sparsely sampled field across a full image), is then broadcast to the array so that 200 copies of it are overlayed on top of the 200 rotational templates. Each of the 256 input flow field vectors is sequentially broadcast to the array to accomplish this loading. The rotational templates are then subtracted, in parallel, from the overlayed copies of the input flow field, and the slope of each of the resulting vectors is computed, also in parallel. The CAAPP then contains a set of 200 vector arrays that represent potential rotational matches, ie. that represent potential translation fields after each rotational field has been subtracted. The 1000 translational templates are then sequentially broadcast to the array in the same manner as the input flow field, ie. so that the 200 copies of each translational template are overlayed onto the 200 putative translational vector arrays contained in the CAAPP.

Each of the translational templates consists of a 16 by 16 array of slope values. Once a translational template is broadcast to the CAAPP, the difference between the slopes is

computed in each CAAPP cell and then scaled. Cells in which the slope difference does not exceed a certain value are tagged as matches. In each of the 16 by 16 vector arrays, the number of matches is then counted by collecting the match bits, through neighbor shifts, in the lower right cell of each 16 by 16 array. If the number of matches exceeds a certain value (75 percent of the elements, or 192 matches, in our implementation), then the rotational and translational template pair for that array is tagged as a candidate for the decomposition. The tagged arrays then have their variance computed (the variance of the scaled slope difference field). A find least search is then performed to determine which of the candidates has the minimum variance. The minimum variance and the identities of the rotational and translational templates that produced it are then saved.

For each successive translational template that is broadcast to the array, the resulting minimum variance value is checked to see if it is less than the saved minimum variance value. If it is less than the saved value, then it and its corresponding template identifiers will replace the saved values. Thus, after all of the translational templates have been broadcast to the array, the saved values will be the identities of the rotational and translational

templates which, together, gave the best match to the input flow field. From the identities of the templates, the actual motion parameters are then returned by table lookup.

To illustrate this process: Figure 41 shows an artificially generated, noise free, sample flow field. Figure 42 shows the array after the rotational templates have been subtracted and a translational template that matches well has been applied. Each of the 200 squares in the figure represents one of the 16 by 16 arrays. Each of the dots in a square represents a cell in which the slope difference has been tagged as a match. Thus, squares that are nearly black represent template pairs that correlate well with the input flow field. Note that the cyclic appearance of the correlations is simply an artifact of the arrangement of the rotational templates as they are loaded into the array; thus rotational templates which are similar are not necessarily adjacent, and in this example the set of rotational templates which respond strongly are all close to each other.

Figure 43 shows the rotational template that the algorithm selected for the sample flow field, while figure 44 shows the translational template that the algorithm selected. As a check on the correctness of these choices, figure 45 shows the input flow field with the selected

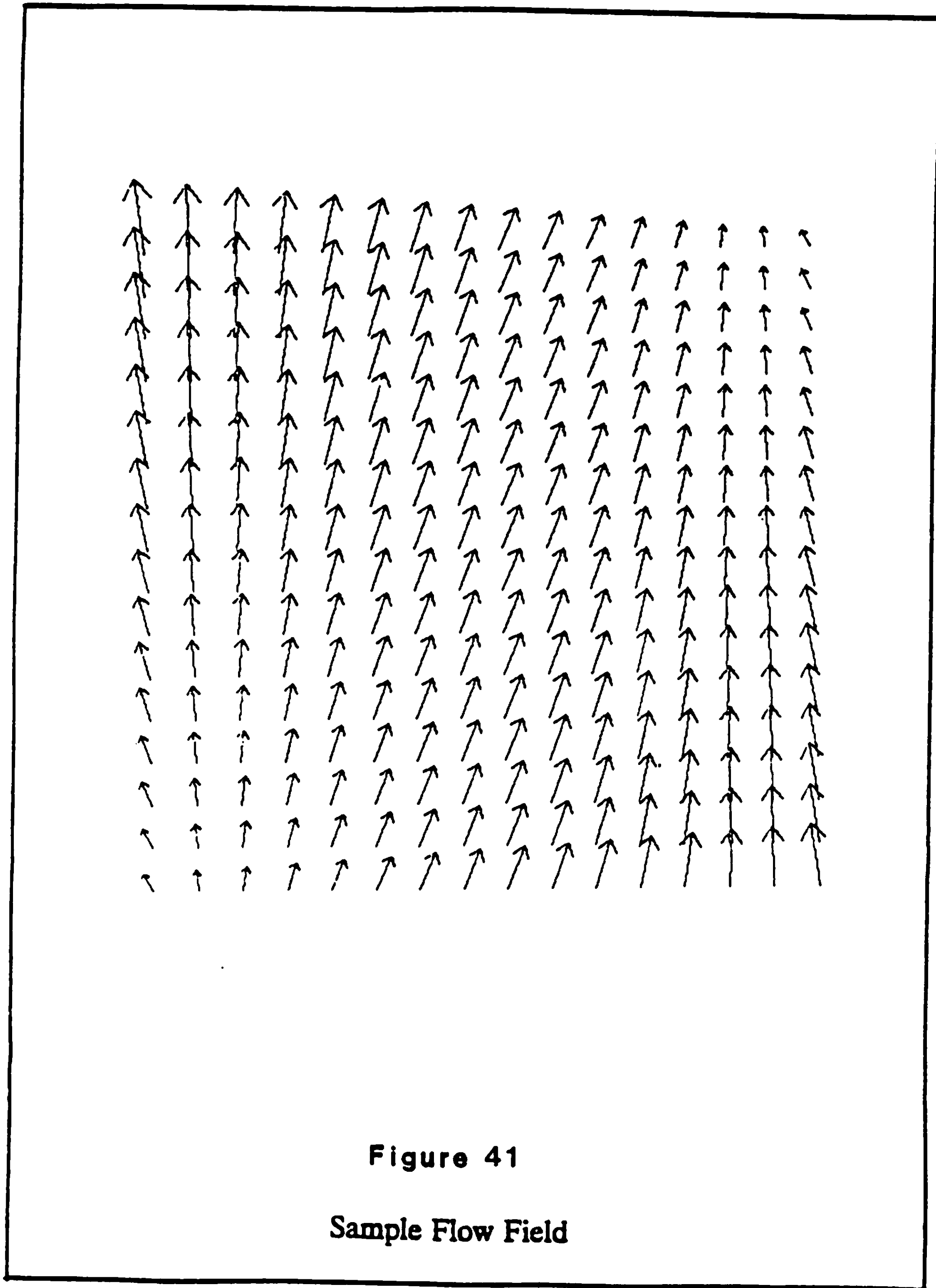


Figure 41

Sample Flow Field

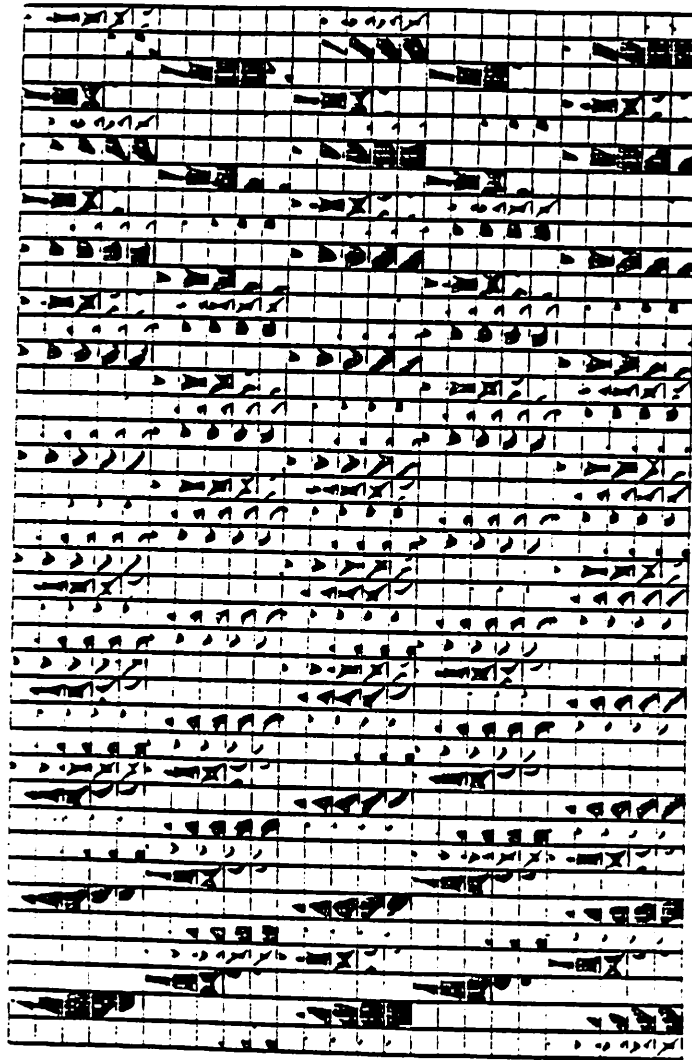


Figure 42

Difference Field for Sample Flow Field

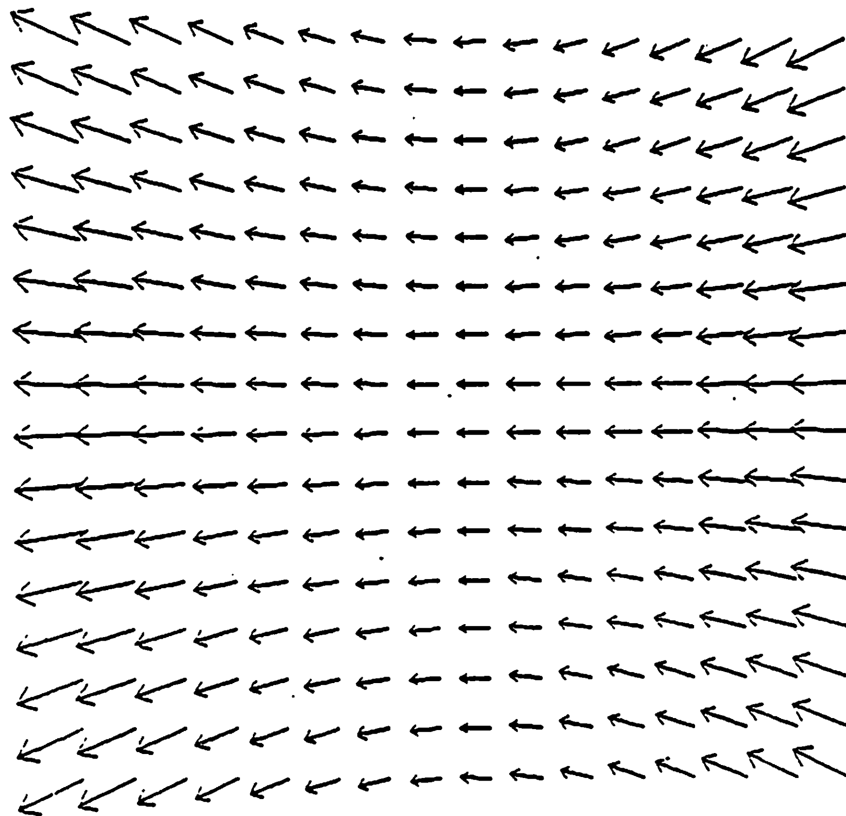


Figure 43

Rotational Template Selected by the Algorithm

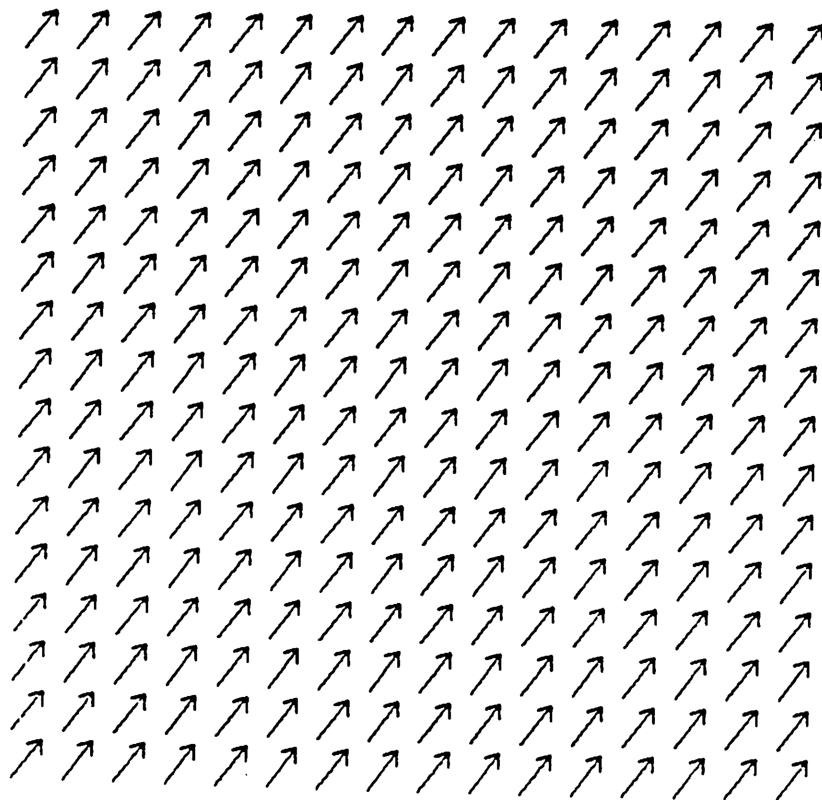


Figure 44

Translational Template Selected by the Algorithm

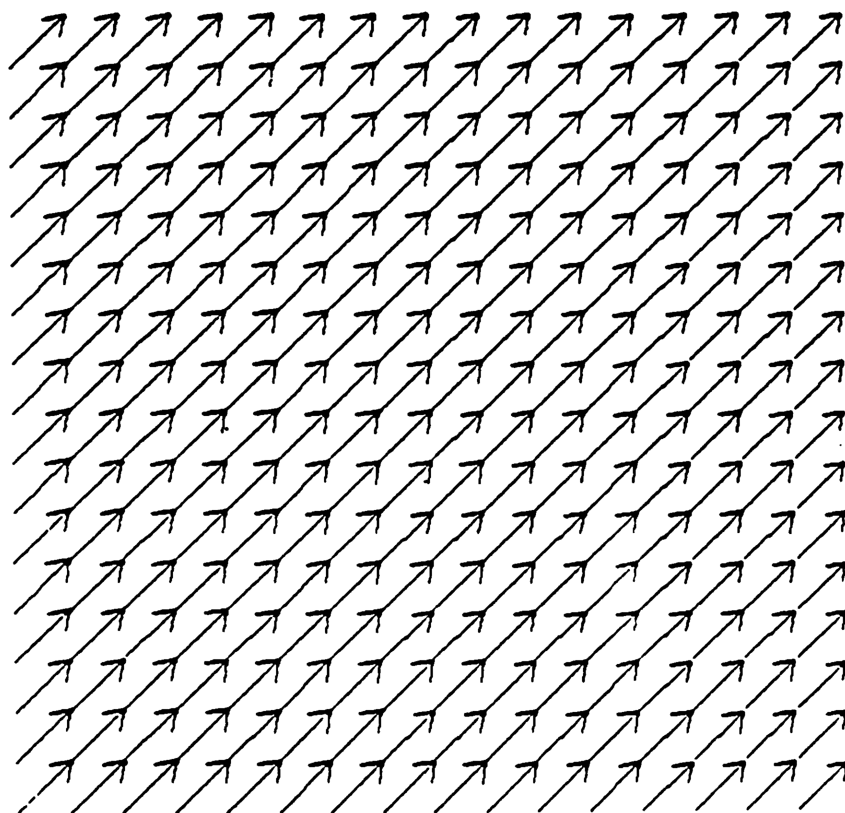


Figure 45

Sample Flow Field With Rotational Template Subtracted

rotational template subtracted. Comparing this to the selected translational template shows that the algorithm did indeed select the best pair of templates.

For comparison purposes, figure 46 shows the CAAPP response for a translational template (shown in figure 47) that correlates poorly with the input flow field.

One of the features of this algorithm is that it is fairly robust with respect to random spike noise in the input flow field. Figure 48 shows the original input flow field with random spike noise added. Figure 49 shows the CAAPP response in the case of the best translational template being broadcast. Note that correlation patterns are very similar to those in figure 42, although not quite as strong. Figure 50 shows the noisy input flow field with the selected rotational flow field subtracted. Comparing this to figure 45 shows that the algorithm has still succeeded in selecting the best pair of templates.

Experiments were performed using a wide variety of motions and simulated environments. In all of the cases that were examined the translational template closest to the actual translational motion was selected. The rotational template selected was always close to the actual rotational motion, but was sometimes not the closest template that

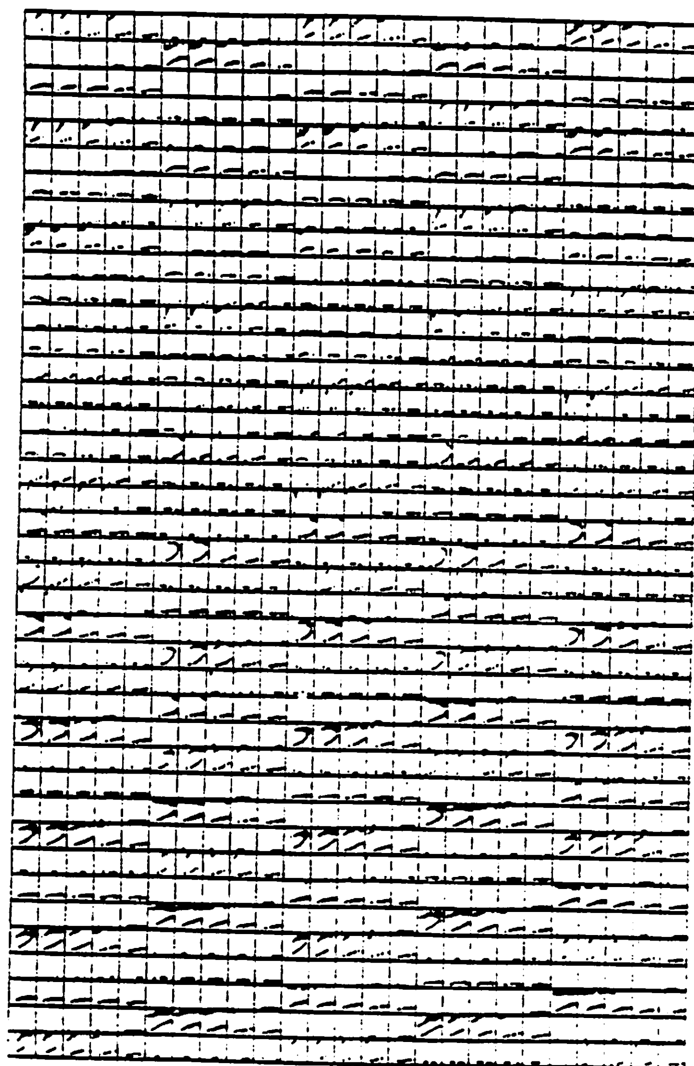


Figure 46

Response to Poorly Matched Translation Template

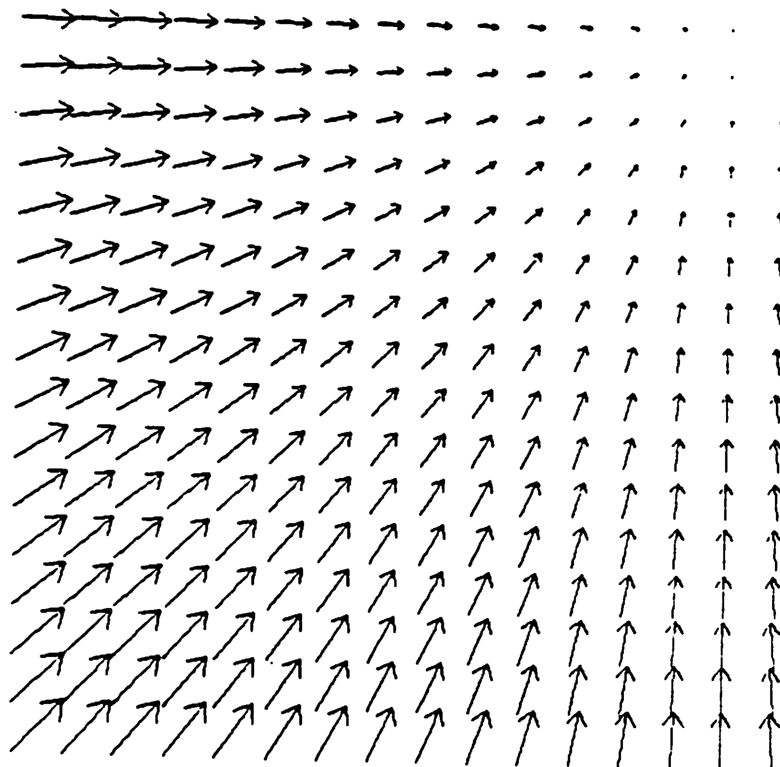


Figure 47

Translation Template Generating Example of Poor Response

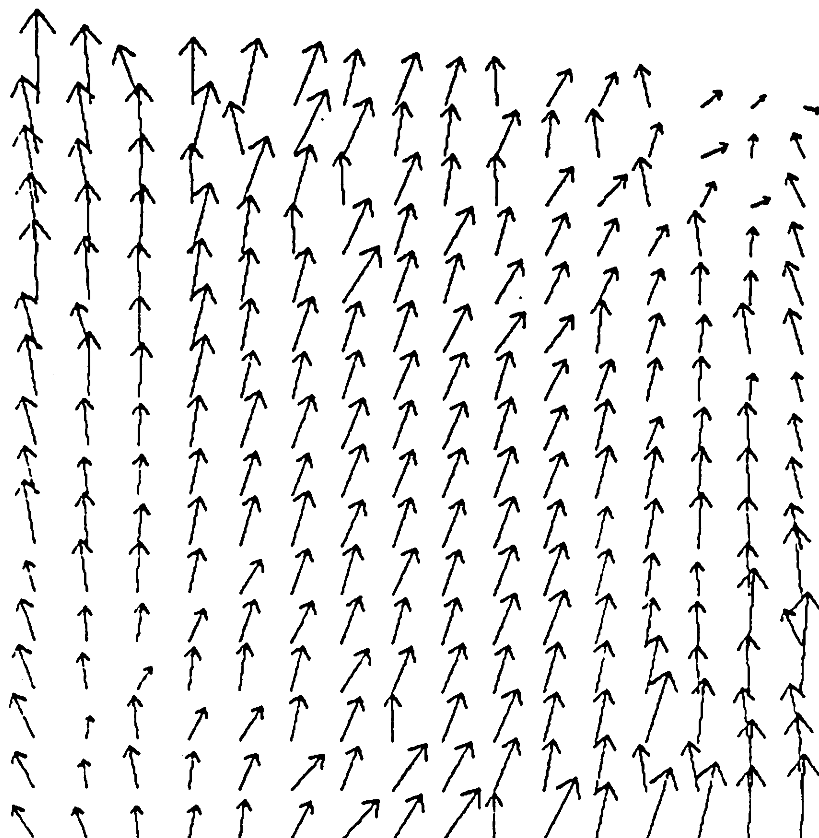


Figure 48

Sample Flow Field With Random Spike Noise Added

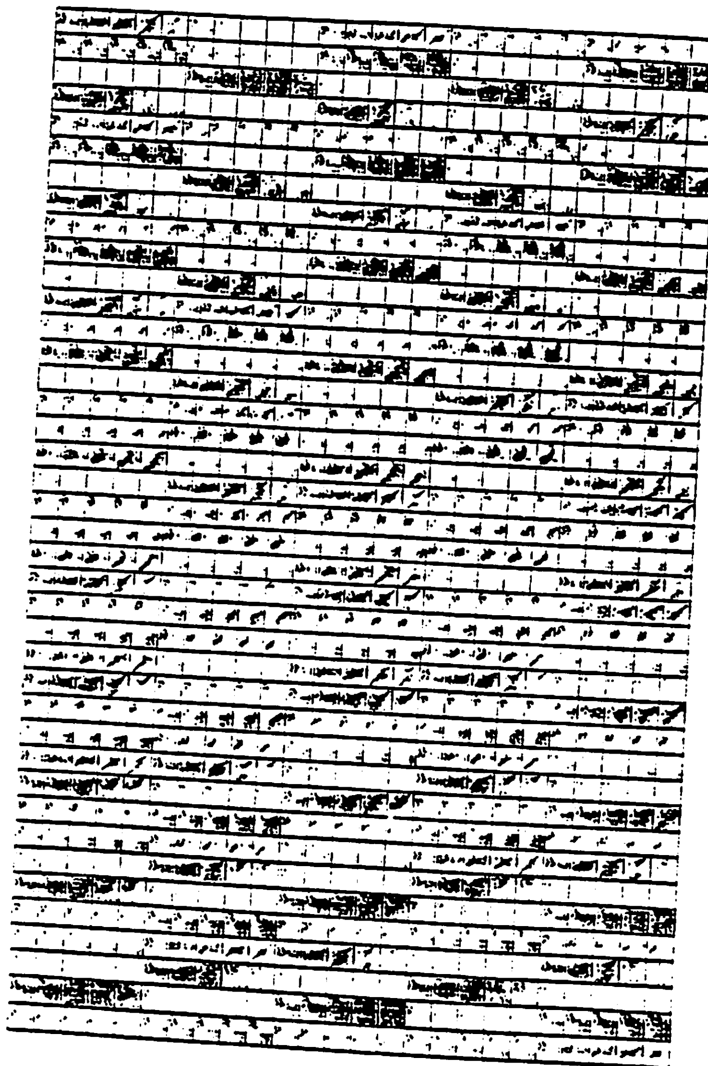


Figure 49

Difference Field for Sample Flow Field With Noise

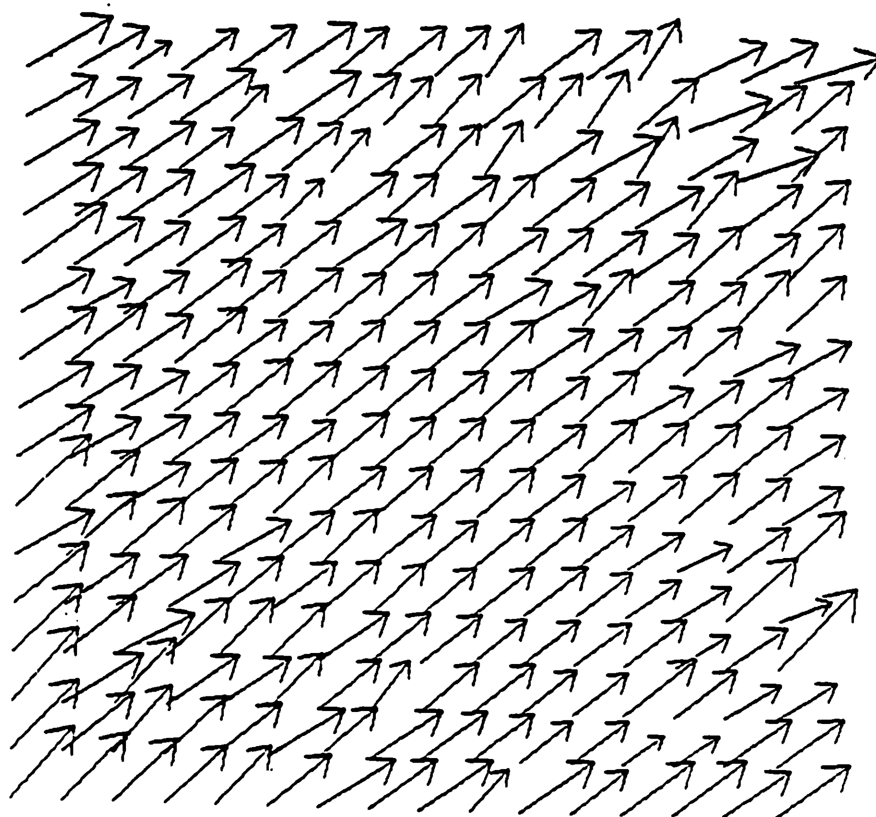


Figure 50

Sample Noisy Flow Field With Rotational Template Subtracted

could have been chosen. The procedure proved to be resistant to limited Gaussian noise as well as to limited random spike noise in the original flow field. Applying independent motion to points at random depths produced results similar to those obtained in the noise experiments. The algorithm's performance degraded slightly if each flow vector component was specified by eight bits instead of by ten.

The CAAPP timing calculations revealed that the algorithm could perform the rotational-translational decomposition in slightly more than one fourth of a second. This is about 7.5 video frame times. Although this is a bit slow for real time motion analysis in scenarios where the camera is moving rapidly it is less than about an order of magnitude away from the necessary speed. If the camera is undergoing gradual changes in its motion then it should be possible to reduce the number of trial templates after the original parameters are determined, because the system will then be refining current motion parameters in an updating process, instead of a startup mode. This will be true even if the camera is moving at high speed. The number of trial templates required depends only upon the rate of change in the translational motion of the camera. This is because all of the rotational templates are stored in the CAAPP at

initialization and remain there throughout operation of the algorithm.

Other Applications

In this section we will look into some areas for application of the CAAPP outside of image processing and computer vision. It should first be pointed out that because the CAAPP is in part a large associative memory, it can be used in the same way as any normal CAM. The use that always comes to mind for a large CAM is database management and indeed the CAAPP would be more than adequate for such an application. What will mostly be examined in this section, however, are applications that take greater advantage of the unique features of the CAAPP. Those that will be looked at are finding the center of mass of an object, geocorrection of satellite images, sorting data in the array, real time execution of LISP programs, simulation of neural networks and graph processing. The first two of these will be developed in detail while the third will be presented as a high level algorithm. For the remainder of these applications it will be pointed out how the CAAPP might be applied. The purpose of this section is to give the reader a taste of the kinds of other applications the CAAPP can be

used for, rather than to solve specific problems.

Center of Mass. This algorithm was originally developed by C. Foster and presented in [125]. It is presented here because it is an excellent example of how the combination of the square grid topology and the fast response count of the CAAPP can be used together to solve problems that would be difficult to solve for a machine that had only one or the other of these features. The timing analysis presented here is also more detailed than Foster's and includes some corrections.

A fairly common problem in engineering is the determination of the center of mass of an object. This is usually more difficult than the problem of locating the centroid (the centroid being the geometrical center of the object). This is due to variations in density within the object. In dealing with models of objects the simplest way to find the center of mass is to divide the object into very small areas by means of a cartesian grid, then use the weight of each area as a multiplier to find the weighted sum of all of the areas on the grid, finally dividing this by the number of areas to get the coordinates of the center of mass. This essentially amounts to finding the weighted average of the coordinate space enclosed by the object.

In developing the algorithm for the CAAPP we will restrict the coordinate space to two dimensions for simplicity. For three dimensional objects a technique such as that used in the hidden surface algorithm described above could be used. This problem which is very time consuming on a conventional computer is quite easy for the CAAPP and runs very quickly. The object may either be loaded into the CAAPP or it may be generated in the CAAPP by some other operation (for example, once region growing has been performed on an image, it might be useful to compare the centers of mass of the major regions with their centroids). The algorithm first loads an X-Y grid into the CAAPP. Next the total weight of the object is computed. This sum is determined by loading each bit of the value specifying the weight of a cell into the X register and counting the number of bits. The bit count is then multiplied by the appropriate power of two to shift it into position and is added to the total sum. The same technique is used to compute the weighted sums later on. The X coordinate is then multiplied by the weight, and all of the cells in the object are again summed. The Y coordinate is treated likewise. Finally, the two weighted sums are each divided by the total weight of the object to produce the X and Y coordinates of the center of mass.

In the following algorithm, it is assumed that the object is already present in the CAAPP and that M(D) contains a one bit in cells that hold the object, and zeroes elsewhere. There will be five other fields in each cell. These are the Weight, X_Weight, Y_Weight, X_Coordinate and Y_Coordinate. For space conservation purposes the X_Weight and Y_Weight as well as the X_Coordinate and Y_Coordinate fields can be the same. Only one of each of the pair of values needs to be in the CAAPP at any time. The following was written as if they are separate fields simply to make the algorithm easier to understand. This does not affect the timing analysis. The constants SW, SXW and SYW are the starting (low order) bit positions of the Weight, X_Weight and Y_Weight fields respectively. The LW, LXW and LYW constants represent their lengths. The remainder of the variables are held in the controller.

```
(* Center of Mass *)

X_Sum := 0                                              0.1 us
Y_Sum := 0                                              0.1 us
W_Sum := 0                                              0.1 us
Load X-Y Grid                                          12.7 us
A := M(D)!                                              0.1 us

for I := 0 to LW-1 do
   begin
      X := M(SW + I)                               LW * 0.1 us
      W_Sum := W_Sum + Response Count             LW * 27.0 us
   end
```

```
Multiply Weight by X_Coord giving X_Weight  3.7(LW) - 3.5 us
for I := Ø to LXW - 1 do
   begin
      X := M(SXW + I)                             LXW * Ø.1 us
      X_Sum := X_Sum + Response Count            LXW * 27.Ø us
   end

Multiply Weight by Y_Coord giving Y_Weight  3.7(LW) - 3.5 us
for I := Ø to LYW - 1 do
   begin
      X := M(SYW + I)                             LYW * Ø.1 us
      Y_Sum := Y_Sum + Response Count            LYW * 27.Ø us
   end

X_Bar := X_Sum / W_Sum                                  Ø.1 us
Y_Bar := Y_Sum / W_Sum                                  Ø.1 us
                        ---------------------------------------
                        34.5(LW) + 27.1(LXW) + 27.1(LYW) + 6.3 us

Because LXW = LYW = LW+9                    88.7(LW) + 494.1 us

Sample time for 8 bit weight:                          12Ø3.7 us
```

The reason that an activity mask is used for the object, and only the weight of its cells is summed, is that this method allows several objects to be present in the CAAPP at one time and yet makes it possible to determine the center of mass of each one individually. This might well be the case if the "objects" are regions in an image, for example. If only a single object is present in the CAAPP, then we could indeed sum the weight of all of the cells regardless of the area occupied by the object. Note that this algorithm is just as applicable to objects that consist of disjoint pieces as to those that consist of a single piece. Thus it could be used to determine the center of mass of a planetary system, for example. It should be pointed out

that this technique need not even be restricted to "objects". It will work just as well if the data is a scattering of points with associated vectors. The ability to sum the contents of a CAAPP field can also be used for statistical analysis of the contents of the CAAPP -- providing a quick means of determining the mean and standard deviation of a data set. All of this simply serves to point out the importance of having a response count mechanism in the CAAPP.

Geocorrection of Satellite Images. Whenever a satellite photographs the surface of the Earth it is, like a mapmaker, converting a curved surface into a flat representation. The mapmaker has the advantage, however, of working from data that represents a stationary view of the surface. The satellite, on the other hand, has to contend with the rotation of the Earth under it, its own orbital motion, its own rotational motion and a certain amount of skew in the angle at which it makes the photograph. This, combined with side effects of the optical system used to make the photograph, produces a picture that has significant blurring in the form of stretching of the image. Because the conditions under which the satellite took the image are usually known, it is possible to create a function that can determine the displacement of each pixel in an image from

its proper position. The following algorithm will show how the CAAPP could be used to both compute this displacement in parallel and to correct the image accordingly.

Because the functions for geocorrection are reasonably complex but are still simply functions of the X and Y coordinate grid decomposable into arithmetic operations that we have already demonstrated in previous sections, the actual computations will not be shown here. Of greater interest is the mechanism for rearranging the image pixels to match the corrected coordinate system. There are two ways of doing this. The simpler one will be discussed here. The more complex will be described following the presentation of the algorithm. The technique used in the algorithm essentially involves moving the the pixels of the image horizontally and vertically, along with their destination coordinates, over a fixed coordinate grid. When a pixel arrives at the fixed coordinate that matches its destination coordinate, it stops moving and drops into place. If another pixel has already occupied that location, the newly arrived pixel will be averaged with it. This may occur because of the phenomenon known as "pixel aliasing". Because of roundoff error in the computation of the corrected coordinates, more than one pixel may be assigned to a location. This will be minimized by the choice of the

best possible correction function, however it is still likely to occur. The result is that the final picture may have some empty pixels. In this case there are several alternative actions but the simplest would be to have all of the empty pixels take the average of their neighbors as their own values. This gives these pixels values that are close to those they should have had and does not reduce the resolution of the rest of the image (as would, for example, application of a Gaussian filter). The method of moving the pixels in the CAAPP is to shift pixels that are not in the proper column to the right until they reach whatever column they belong in. Upon reaching the proper column, those pixels that are not in the proper row advance upwards until they reach that row. At that point the pixel has reached its new home and settles down by either dropping into place or being averaged with a preceding value. Because the CAAPP provides for circular wrap from edge to edge, a pixel that is initially above or to the right of its proper position will simply "go around the horn" before reaching its destination. Obviously, the worst case for this algorithm is that in which the image correction simply shifts all of the pixels one cell down and to the left. A modification of the algorithm might perform a quick analysis of the average displacement of the pixels and adapt the shifting direction to give the best speed. As presented, the algorithm is

sufficiently complex without the addition of such speed enhancements. This will give an upper bound on the timing that should be easy to improve upon.

In the following XC, YC, DXC and DYC refer to the fixed X coordinate, fixed Y coordinate, destination X coordinate and destination Y coordinate respectively. OV, FV, NSV, ANSV, NSDY, FVE and OVF hold the original value of the cell, the final value of the cell, a value temporarily in the cell because it is moving North from somewhere to the South, an activity bit indicating that the cell contains a pixel moving vertically, the value of the destination Y coordinate for the cell moving vertically, an activity bit indicating that the final value field of the cell is still empty, and an activity bit indicating that the original value field is still full, all respectively. The algorithm terminates when all of the original values have moved into final position.

```
(* Satellite Image Geocorrection *)

Load Image                                                30000.0 us
Load X-Y Grid into XC, YC fields                             12.7 us
Apply deformation function to XC, YC, giving
   DXC, DYC grid                                               DF us

A := 1!                                                       0.1 us
M(FVE) := 1                                                   0.1 us
M(OVF) := 1                                                   0.1 us

repeat
```

```
   A := 1!                                                  512 * Ø.1 us
   Exact Match XC = DXC (* 9 bits *)                        512 * 4.7 us
   A := X                                                   512 * Ø.1 us
   Exact Match YC = DYC (* 9 bits *)                        512 * 4.7 us
   A := X                                                   512 * Ø.1 us
   Copy OV to FV   (* LV bits *)                       512 * LV * Ø.3 us
   M(OVF) := Ø                                              512 * Ø.1 us
   A := -A!                                                 512 * Ø.1 us
   Exact Match XC = DXC (* 9 bits *)                        512 * 4.7 us
   A := X                                                   512 * Ø.1 us
   Copy OV to NSV  (* LV bits *)                       512 * LV * Ø.3 us
   Copy DY to NSDY (* 9 bits  *)                            512 * 2.7 us
   M(OVF) := Ø                                              512 * Ø.1 us
   M(ANSV) := A!                                            512 * Ø.1 us
   A := 1!                                                  512 * Ø.1 us
   repeat
      Shift NSV North  (* LV bits *) 512 * 512 * LV * 1.Ø us
      Shift ANSV North                         512 * 512 * 1.Ø us
      Shift NSDY North (* 9 bits  *)           512 * 512 * 9.Ø us
      A := M(ANSV)                             512 * 512 * Ø.1 us
      Exact Match YC = NSDY  (* 9 bits *)      512 * 512 * 4.7 us
      A := X                                   512 * 512 * Ø.1 us
      B := A!                                  512 * 512 * Ø.1 us
      M(ANSV) := Ø                             512 * 512 * Ø.1 us
      A := M(FVE)                              512 * 512 * Ø.1 us
      Copy NSV to FV  (* LV bits *)  512 * 512 * LV * Ø.3 us
      A := B!                                  512 * 512 * Ø.1 us
      X := M(FVE)                              512 * 512 * Ø.1 us
      X := -X                                  512 * 512 * Ø.1 us
      A := X                                   512 * 512 * Ø.1 us
      Set FV to (FV + NSV)/2   512 * 512 * Ø.4 * LV + Ø.3 us
      A := 1!                                  512 * 512 * Ø.1 us
      X := M(ANSV)                             512 * 512 * Ø.1 us
   until None                                  512 * 512 * Ø.1 us

   Shift OV East  (* LV bits *)                        512 * LV * 1.Ø us
   Shift DY East  (* 9 bits  *)                             512 * 9.Ø us
   Shift DX East  (* 9 bits  *)                             512 * 9.Ø us
   Shift OVF East (* leaves OVF in X *)                     512 * 1.Ø us
until None                                                  512 * Ø.1 us
                -----------------------------------------------
                1.8(LV)*512**2 + 1.6(LV)*512 + 419Ø883.4 + DF us

                           472678.4(LV) + 419Ø883.4 + DF us

Sample time for LV = 8 bits                        797231Ø.6 + DF us
```

For an average DF this will work out to about 8

seconds. It should be noted that this is an absolute worst case time. As mentioned above, there are modifications to the algorithm that will speed it up considerably. For example, the one described above would reduce the worst case time to about one fourth of this (or about 2 seconds). The average time should be under half of the worst case time and so it is reasonable to estimate that an average image could be corrected in about one second.

The simple algorithm given above does not take into account the effects of deformations which push pixels off of the grid. In this case they are simply wrapped around to the opposite edge of the image. This may result in some problem if there are overlapping pixels at those points that really belong there. The easiest solution for this is to use the overflow bit from the DXC and DYC computations as a flag that either prevents the pixels from moving, or allows them to move but puts them into a separate final value field once they reach their destination (it may even be possible to reuse the Original Value field at that point).

The complex version of this algorithm requires less memory space and should also run a little faster. The basic idea is very much the same, but instead of keeping both sets of coordinates in memory and comparing them, only one set representing the distance of a pixel from its destination is

used. Each time a pixel moves, its distance vector is decremented. When both components of the vector equal zero, it has reached its destination. Using some complicated and messy control structures it would also be possible to take advantage of the fact that as pixels move across the grid, the maximum length of the vector components decreases and so shorter and shorter field shifts could be used to further save time. The timing analysis for this would, however, be very messy.

Square Grid Sort. Before presenting this algorithm it should be noted that sorting on a pure associative processor is about as useful as an icebox is to an Eskimo. Sorting is most frequently done on serial processors in order to make searching or merging lists less time consuming. With an associative processor, each datum can be located instantly. If it is truly desired to read the contents out in order, the find greatest and find least operations can be used to produce a datum every few microseconds without first sorting the array. Why then are we interested in sorting on the CAAPP? The novelty of this is not in the ability to produce an ordered list, but in the fact that it provides an efficient way to move data around in the array so that similar items can be grouped together in physical proximity with respect to the communications network. This is useful

in applications that require intercell communications such as semantic networks and simulation of neural nets. By applying a function to the communication rates and addresses of cells to which messages are sent a sorting key can be computed. Periodically, then, the array can be sorted to bring cells that communicate frequently into closer proximity, thereby reducing the travel time for the messages.

The concept of the sorting algorithm that was developed for the CAAPP is quite simple. Its implementation is a bit messy however. Rather than show this at the detailed level and attempt to extract a timing analysis, the high level version of the algorithm will be presented and a rough estimate of the timing will be made.

This sort is somewhat similar to one that was developed by Habermann [56] for use on parallel processors arranged as a linear string of elements. Habermann's sort was in turn based upon the standard compare and swap exchange sort often referred to as the bubble sort. The basic idea is that each cell examines, in turn, each of its neighbors and, if it discovers that it is out of the proper ordering with any of them, exchanges its value for theirs. A problem arises in that all of the cells cannot be doing this in parallel. If they did then we would see, for example, one cell trying to

exchange with another while the other is trying to exchange with a third. Which cell ends up with the value in the middle? To avoid this, the grid is divided into strips vertically and horizontally. The strips are each two cells wide and run all the way across the grid. There are two subsets of these strips in each of the directions (four subsets in all). The two subsets running horizontally overlap by one cell as do the subsets running vertically. For the vertical direction, one of these subsets consists of all of the pairs of columns formed by an even numbered column on the left and odd numbered column on the right (cell columns are numbered sequentially, left to right, starting with zero). The other subset consists of all pairs of column with an odd numbered column in the left position of the pair and an even numbered one to the right. These are called the even and odd column pairs, respectively. The even column pair set thus consists of the column pairs <0,1>, <2,3>, <4,5>,..., <510,511>. The odd column pair set consists of columns <1,2>, <3,4>, <5,6>,..., <509,510>. The same applies for the rows.

Once the grid is divided into strips, it becomes possible for exchanges to take place. All exchanges are restricted to taking place within (and actually across) whatever set of strips is activated. By activating only one

set at a time, this ensures that only pairs of cells will participate in exchanges. The overlap between the subsets of stripes will allow values to cross from stripe to stripe as exchanges are alternated between pairs of stripes.

Defining the ranking of cells in a two dimensional array is also a bit of a problem. Such schemes as diagonalization are not without merit. Arbitrarily, however, the ordering that was chosen is the same as the ordering for the CAAPP when treated as a linear array. This is the same as english reading or lexicographic order, namely left to right and top to bottom. The way that the algorithm is presented will result in the lowest order cell having the greatest value, but this can just as well be reversed by reversing the inequalities in the comparisons. This definition leads to a problem in moving data around the array. In order to get values to advance upwards upon reaching the leftmost column of the array it is necessary to introduce diagonal end around links. These connect the zero column cell of row I+1 with the 511 column cell of row I and vice versa. This is already provided for by the zig zag edge connect of the CAAPP and presents no hardware problems. When exchanging with diagonal neighbors it will be noted, however, that the exchange includes end wrap around. The horizontal and vertical exchanges, however, do not.

In the following algorithm there is no reference to specific fields in the CAAPP memory. It is assumed that the comparisons take place on the proper bits and that the exchanges move the appropriate data. The timing will differ depending upon the length of the comparison fields and the amount of data that must be shuttled across to the neighboring cell in an exchange. This is an expensive operation because of all of the inter-cell communication that must take place. The CAAPP requires about one full microsecond per bit that must be transferred between cells. The constant N in the following is the number of cells in the array.

```
(* Square Grid Sort *)

for I := 1 to SQRT(N)-1 do
   begin
      A := M(Even_Columns)!
      Compare East Neighbor > Own Value
      A := X
      Exchange Values with East neighbor

      A := M(Even_Rows)!
      Compare South Neighbor > Own Value
      A := X
      Exchange Values with South Neighbor

      A := M(Even_Rows)!
      Compare Southwest Neighbor > Own Value
      A := X
      Exchange Values with Southwest Neighbor, edge wrap around

      A := M(Even_Rows)!
      Compare Southeast Neighbor > Own Value
```

```
    A := X
    Exchange Values with Southeast Neighbor, edge wrap around

    A := M(Odd_Columns)!
    Compare East Neighbor > Own Value
    A := X
    Exchange Values with East Neighbor

    A := M(Odd_Rows)!
    Compare South Neighbor > Own Value
    A := X
    Exchange Values with South Neighbor

    A := M(Odd_Rows)!
    Compare Southwest Neighbor > Own Value
    A := X
    Exchange Values with Southwest Neighbor

    A := M(Odd_Rows)!
    Compare Southeast Neighbor > Own Value
    A := X
    Exchange Values with Southeast Neighbor
end
```

A rough timing estimate for this is 511 * (12 * LC + 8 * LV + 6) microseconds, where LC is the length of the comparison field and LV is the length of the value field. For eight bits each, this gives a total sorting time of 84826 microseconds or about 85 milliseconds. Normally, of course, the fields will be considerably larger, but it is interesting to note that the time simply grows linearly with field size. The most interesting aspect of this sort is the order associated with it, namely square root of N. One of the maddening things about developing this sort is the difficulty of proving it; as yet there is no analytical proof for it. It has been shown empirically to work for all arrays up to size N = 16. Beyond this the number of

combinations required to fully test an array size grows quite large. The largest array for which it seems feasible to exhaustively test the sort is N = 49 (7 by 7). This requires applying the sort to 2**49 different binary arrays. The estimated order of the algorithm is the result of this testing. In actuality the average time for the sort seems to be well below half of the maximum time.

The number of compare and exchange operations associated with this sort is 8 * (SQRT(N)-1). The factor eight derives from the number of neighbors that surround a cell, and thus with which it must compare and exchange. If we introduce the variable D to represent the dimensionality of the array, then the formula becomes (3**D - 1) * (N**(1/D) - 1). This raises the interesting possibility of generalizing the sort to operate on a D dimensional orthogonal grid. It can be seen from this last formula that if N is allowed to grow much more quickly than D, that is if N is always large with respect to D, then a nearly D-th root of N sorting time will be achieved. Much as I would like to prove this, I am afraid that it will have to be left for future research.

<u>Real time execution of LISP programs</u>. Bonar and Levitan [13] have shown that a CAM may be used to insure real time behavior of programs written in the LISP programming language. LISP programs are highly dynamic in their use of

memory space. As such it is a common occurence that a LISP program will run out of storage space in main memory. When this happens a procedure called "garbage collection" takes place. This involves searching memory for cells that are no longer pointed to by any active structures. (The linked list is a basic structure in LISP.) It is usually most efficient to simply collect all such garbage cells at one time and return them to the free storage pool. Because all of memory may have to be searched for even a single free cell, garbage collection introduces the potential for arbitrary delay times in a LISP program. Because of the highly dynamic nature of LISP it is also nearly impossible to predict when these delays will occur. In a real time application these "naps" can be disastrous since they may come during critical control periods. Because most artificial intelligence applications are written in LISP, this has effectively prevented their use in environments requiring real time response.

With a CAM, however, it is possible to avoid naps entirely so long as there is any memory left at all. The garbage collection problem can be reduced to the question of how to tell if a cell is truly garbage. In LISP a cell can have more than one parent node in the linked list structure. Simply because one of a cell's parents releases

it does not mean it is now free -- it may have other parents that still need it. The only way to tell if this is true, in an addressed memory system, is to search the pointer fields of all cells in order to tell if any of them point to the cell in question. This is why garbage collection is often put off as long as possible. Doing so allows a single search to check all of the potentially free cells at once. Still, this is a time consuming process.

On a CAM, however, a search operation can be performed very quickly. This allows garbage collection to be performed on the fly, as each cell is needed. Whenever a parent node frees a cell, a simple search of CAM for any cells which point to the cell in question will determine if indeed that cell can be labelled as garbage. Essentially, the usual overhead of garbage collection is distributed to these global search operations. Each search takes time proportional to the number of bits in the linkage fields of the cells. When a free cell is needed, a simple Select First operation can be applied to the garbage status bit of all cells to select a free cell. If the selected cell has non-null pointer values then it is also necessary to examine the cells to which these point to see if they are also garbage. It should be noted that this last operation can be interrupted and stacked for processing at some later time --

the essential operation of retrieving a free node has taken place. Although Bonar and Levitan originally implemented these ideas on the Semionics REM described in chapter 5, the CAAPP can be used equally as well for this. The operation that frees a node can be done in about 20 microseconds on the CAAPP. Retrieving a free node and freeing any garbage subnodes can be done in about 55 microseconds on the CAAPP (of which about 40 microseconds is spent in freeing the subnodes).

This leads to some interesting possibilities for image processing. Most notably it means that it is possible for the same CAAPP cells to be used to perform symbolic processing as are used for the initial image processing chores. Once regions and segments have been grown in an image, they can be reduced to a few essential descriptive statistics. The remaining image pixel cells are then no longer needed and can be converted into free storage for symbolic processing. Each seed cell for an image feature can then become the head node of a linked structure that describes how it is related to other image features. This has been called by Nudd, the iconic to symbolic transformation. This appears to be one of the key steps on the road to image understanding within artificial intelligence systems. The fact that the associative nature

of the CAAPP can guarantee a minimum real time response in symbolic processing means that the entire image understanding process can be performed under some set of real time constraints.

To see how this iconic to symbolic transformation might proceed, take for example the building of the relations list for a set of line segments in an image. The segments are first extracted with a Sobel style operation with some additional thinning and gap filling. Then an interest operator is applied that finds points of high curvature. This can be evaluated with the response count and then adaptively refined until a reasonable number of interesting points has been extracted. Between these points lay line segments of lower curvature. The selected points send "runners" out in both directions along these segments. The runners carry their starting addresses and also keep a count of the number of cells through which they pass. When two runners collide, they form a seed point for a segment of low curvature that connects two points of high curvature. The seed point has available the end points of the segment it represents and thus the length of the straight line that connects them; and also the number of cells that lie along the segment. From this it can compute the degree and direction of curvature of the segment. It now becomes

possible to start building representations of how the segment is related to other segments. This might take the form of selecting all segments with lengths (or curvatures) withing a certain range (perhaps directed by histogram analysis). For this set, a first responder is selected and read out by the controller. Its values are then broadcast to the rest of the cells in the set. These are the seed cells which represent the segments. Because the segments have been condensed to a seed cell representation, the rest of the cells that formerly contained the pixels that represented the image form of the segment can now be used as a free storage pool for the relationships. Thus the seed cells take the broadcast information and pass it to neighboring cells to be stored. This can be in the form of a LISP structure, with pointers, but also has the advantage of placing the information within close proximity of the list header (the seed cell) with respect to the communications network. It therefore becomes possible to perform direct as well as linked traversals of the relational structure. Once the segment information has been read out, broadcast and stored as a relational network, it becomes possible to make very complex and sophisticated queries and to get answers very quickly. For example, a query might ask if there are any structures which contain multiple segments that have roughly the same length,

orientation and curvature. A series of feature models could be broadcast to the structures in order to select those that are candidates for matching a particular object. Any query that requires all of the image features to have knowledge of certain aspects of their surroundings becomes possible. The same sort of technique can be used for image regions.

It should be noted that although the iconic to symbolic transformation has been described as if it is a single, time consuming phase of the processing, it need not be done that way. The relational structures can be built on a purely by-demand basis. That is, the host processor can keep a semantic network that represents the relationships that have been built in the CAAPP up to the current time, and from this determine if it is possible to make a particular query. If it is not possible, then the net can be used to determine what information must further be extracted and broadcast in order to build the needed relations. Thus only the relations necessary to the understanding of each particular image need to be constructed. Once they have been built, then they are available for any future queries. It may even be possible that the host can analyze the net to determine relations that can be discarded to conserve memory space.

<u>Simulation of neural networks</u>. While one approach to

the development of intelligent systems is based upon the modelling of the apparent external behavior of living organisms, there is another approach that considers how organisms function internally to try to understand what structures are involved in producing the apparent behavior. This involves modelling the neurophysiology of living organisms. The CAAPP can be used to build and examine fairly large models of neural networks. The primary feature of the CAAPP that permits it to do this is its communications network.

Depending upon the amount of detail in the model, one or more CAAPP cells may be used to represent each neuron. The communications lines connected to each neuron cell group may be used to represent axons that connect the neuron to other neurons. Because each communication line can carry multiple messages in either direction, only one link is needed between any two cells in order to represent any combination of axonal connections between the neurons they represent. Because neurons may connect to many other neurons that are a considerable distance away, it will often be necessary to use some of the CAAPP cells simply as way stations for passing messages between distant neurons. If used appropriately, these can represent relative delay times for signal propagation along axons.

This mechanism should be capable of achieving a considerable amount of parallelism. It will be most limited by the number of different types of neurons used in the model. This is because the CAAPP is an SIMD machine and must therefore process each type of neuron sequentially for each simulation cycle. For a model made from homogeneous neurons, the maximum parallelism is attained. The worst case is for a model in which no two neurons are the same. This is no better than fully serial processing could provide, and is even a bit worse due to the bit serial nature of the processors. It may be possible to structure the model so that all of the neurons have some phase of processing in common, with only a subset of the operations requiring sequential processing.

Graph processing and semantic networks. Before discussing how the CAAPP can be applied to graph processing it is appropriate to review a little terminology of graph theory. A graph consists of a set of nodes called "vertices" and a set of links between the vertices called "arcs". The arcs may have directions associated with them, in which case the graph is called a "digraph" or directed graph. It is also possible for arcs to have values or costs associated with them. In this case the graph is said to be "weighted". A graph may be disjoint. That is the graph may

consist of disconnected subsets. A "path" through a graph is any series of arcs and vertices that links a given pair of vertices.

Graphs are frequently used to represent networks such as the telephone communication network, the highway network, electrical power networks and so on. A common problem that graph analysis is used to solve is the determination of a minimal cost path through such a network. On the CAAPP this problem can be approached in much the same way as the neural network simulation. Each network vertex is represented by one or more cells and communications paths are established between these cells to represent the arcs of the graph. These paths need only be virtual in the sense that they can be represented by a linkage address, stored in the cell at the head of the arc, which gives the location of the cell at the other end of the arc. In this case, communication between vertices can use a mechanism similar to that which was used in the satellite image geocorrection problem. Specifically, each message sent from one vertex to another consists of some information with a destination address attached. The message then wanders across the CAAPP grid in a directed fashion, by comparing its destination address to its current location at each iteration, until it reaches the destination point. Parallelism is achieved whenever an

application can make use of more than one message transmission at a time. For example, one way of establishing a minimal cost path is to perform a breadth first search. Because the CAAPP can activate and evaluate all of the paths at the same level simultaneously, it takes no longer to reach a particular depth in the search than it would for a depth first search to evaluate a single path to the same depth.

Another use for graphs is the codification of knowledge in the form of a semantic network. In this structure nodes represent objects, concepts and events. Arcs are used to represent relationships between the nodes. For example, "Lisa" might be linked to "Mathew" by a "sister-of" relation. In this type of application it is common to ask questions that are similar in form to the question of "Who are all of the ancestors of this person?" In the case where the node representing the person is linked by "offspring-of" relations to all of the nodes representing parents of the person, then it is an easy manner to trace back to the parents in parallel and then to transitively trace back along the parents "offspring-of" arcs and so on until all of the known ancestors have been located. The problem is more difficult, however, if only the relationships for "parent-of" are present. In this case the backwards links

do not exist. These can be created by searching all of the cells for nodes that point to the node in question with "parent-of" links and then having them pass their addresses to the offspring node so that it can create the backwards link.

In the CAAPP there are two ways of creating links. For cases in which many nodes require this search to be performed, a method like that used in the MIT Connection Machine [59] can be applied. This consists of waves of communication that are propagated outward from all of the nodes that are initiating searches. As these waves expand outward across the CAAPP they reach nodes that satisfy the search criterion and which thus store the return address of the node that originated the search. The nodes can either wait to respond until all of the waves have covered the CAAPP, or they can respond immediately (the latter requires more complex control and is a bit slower than the former, which requires more memory). The response takes the form of messages which are sent back to the originating node to establish the reverse link. This method uses a mechanism that is similar to that used for the geocorrection algorithm to get the messages to their destinations. The second method is more like that used in the iconic to symbolic transformation described above. It is best suited for

smaller sets of nodes. In this method each node is selected individually, read out from the CAAPP by the controller and rebroadcast to the array. This establishes very quickly the responder nodes which can then use the message transmission mechanism to pass their addresses to the appropriate node to establish the linkage. The first method is exactly that used by the Connection Machine and it is interesting to note that the CAAPP is capable of performing any operation that can be done on that architecture. There is some loss of speed due to the communications multiplex employed in crossing chip boundaries in the CAAPP. The designers of the Connection Machine chose to optimize their design for intercellular communication. Were the CAAPP to be redesigned with a full interconnection path between chips it could achieve speeds comparable to that of the Connection Machine. In addition to these capabilities, however, it is important to note that the associativity and feedback mechanisms in the CAAPP architecture give it the ability to perform the second type of search and moreover, the ability to examine the current overall status of the array to decide which of the two methods is most appropriate.

To summarize, the CAAPP has been demonstrated as being capable of performing all of the standard CAM and array processor functions. It has also been shown how it may be

used to perform low level and intermediate level image processing tasks. Methods for implementing some symbolic processing on the CAAPP have been described. Furthermore a method has been outlined by which it should be possible to directly transform a reduced iconic image representation into a symbolic representation within the CAAPP array. Thus it may be possible for this single architecture to perform all of the steps in the image understanding process from enhancing the raw image to manipulating a symbolic representation of the image to extract its meaning. It has also been shown that the CAAPP is useful in a number of other application areas including modelling of CAD objects and dynamic systems.

CHAPTER VI

EVALUATION AND REDESIGN

Introduction

In this chapter an evaluation of the architecture presented in chapter 4 will be presented. This evaluation will be based upon a statistical analysis of the instruction usage in the algorithms presented in chapter 5. Further evaluation will take into account an examination of several of the simpler algorithms to determine specific problems that slow execution. Following the evaluation a set of enhancements to the architecture will be described. Most of these changes will be based on the concept of eliminating lumpiness in the distribution of instruction usage as a means of reducing bottlenecks and generally smoothing and distributing the flow of information within the processors. This should maximize processor throughput and produce optimal utilization of all of the parts of the processors. These descriptions will include estimations of the difficulty of implementation for each change.

The proposed enhancements will be assembled into three new designs. The combination of enhancements selected for

each design will be based upon different relaxations of the original design constraints. The first design will include all of the enhancements that can be added without going beyond the original set of constraints. The second design will allow moderate relaxation of the packaging pin-out limitation while still minimizing the number of communications lines between circuit boards. The third design will be unrestricted with respect to pin-out and communications, and will have its silicon space restrictions moderately relaxed. These will be referred to, respectively, as the conservative, intermediate and advanced designs. The presentation of each of these designs will consist of a description of the constraints being relaxed, the set of enhancements that can then be incorporated, the resulting logic for each processing element, a description of the new instruction set that this presents, an analysis of pin count and an estimate of the space requirements for the new chip design, any overall changes to the architecture, and a commentary on the overall complexity, capabilities, benefits, limitations and special features of the new design.

Statistical Summary of Algorithms

Table 11 gives the total number of occurrences of each of the CAAPP integrated circuit instructions in the algorithms that were coded in chapter 5. The instruction counts were obtained by tallying the appearances of each instruction in the code for each algorithm. Where simpler operations were referred to in more complex algorithms (for example, the use of Load X-Y Coordinate Grid in the Center of Mass algorithm), the simple operation was expanded and counted as if it had been written out in long form at those points. Although this collection of algorithms does not cover the entire range of possible applications for the CAAPP, it does represent those types that are likely to be used frequently. Thus these instruction counts are a reasonable indication of expected overall instruction set usage for the CAAPP. In the discussion a few exceptions to this will be noted and explained.

In table 12, the number of algorithms in which each instruction appeared at least once is given. Table 13 translates this into a percentage of the total number of algorithms. This is a rough indicator of the generality of each instruction. That is the breadth of application areas in which it may be useful. For example, the M:=X

	+		+	-		+	-		+	-		+	-
M:=C	38	X:=A!			A:=A!		2	B:=A!	20		Y:=A!		
A:=M	9	X:=B	6		A:=B			B:=B			Y:=B	2	
M:=B		X:=X		28	A:=X	20	2	B:=X	8		Y:=X	1	
B:=M		X:=Y	4		A:=Y			B:=Y			Y:=Y		5
M:=X	98	X:= $\overline{X+Y}$		19	A:= $\overline{X+Y}$	2	4	B:= $\overline{X+Y}$			Y:= $\overline{X+Y}$		61
X:=M	81	X:= $\overline{X \wedge Y}$		6	A:= $\overline{X \wedge Y}$			B:= $\overline{X \wedge Y}$			Y:= $\overline{X \wedge Y}$		1
M:=Y	63	X:= $\overline{X \vee Y}$		1	A:= $\overline{X \vee Y}$			B:= $\overline{X \vee Y}$			Y:= $\overline{X \vee Y}$		
Y:=M	70	X:=C	77		A:=C			B:=C			Y:=C	14	2
M:=A!		X:=N	35	1	A:=N			B:=N			Y:=N		
A:=M!	2	X:=E	35		A:=E			B:=E			Y:=E		
M:=B!		X:=W		1	A:=W			B:=W			Y:=W		
B:=M!		X:=S			A:=S			B:=S			Y:=S		
		X:=N~!	9		A:=C!	17		X:=CN~!	1		Z:=C	58	
X:=Z	24	X:=E~!	5		A:=B!	25		X:=CE~!			Z:=X	1	
CRCR!	4	X:=W~!	8		A:=X!	2	1	X:=CW~!	1		SCRR!	4	
PANS!	8	X:=S~!	10		A:=Y!			X:=CS~!			SCRC!		

Table 11

Instruction Occurrence Counts for Sample Algorithms

	+		+	−		+	−		+	−		+	−
M:=C	12	X:=A!			A:=A!		2	B:=A!	13		Y:=A!		
A:=M	6	X:=B	1		A:=B			B:=B			Y:=B	1	
M:=B		X:=X		8	A:=X	8	2	B:=X	1		Y:=X	1	
B:=M		X:=Y	1		A:=Y			B:=Y			Y:=Y		5
M:=X	16	X:=$\overline{X+Y}$		9	A:=$\overline{X+Y}$	1	4	B:=$\overline{X+Y}$			Y:=$\overline{X+Y}$		7
X:=M	21	X:=$\overline{X \wedge Y}$		2	A:=$\overline{X \wedge Y}$			B:=$\overline{X \wedge Y}$			Y:=$\overline{X \wedge Y}$		1
M:=Y	8	X:=$\overline{X \vee Y}$		1	A:=$\overline{X \vee Y}$			B:=$\overline{X \vee Y}$			Y:=$\overline{X \vee Y}$		
Y:=M	10	X:=C	14		A:=C			B:=C			Y:=C	9	2
M:=A!		X:=N	5	1	A:=N			B:=N			Y:=N		
A:=M!	2	X:=E	5		A:=E			B:=E			Y:=E		
M:=B!		X:=W		1	A:=W			B:=W			Y:=W		
B:=M!		X:=S			A:=S			B:=S			Y:=S		
		X:=N~!	4		A:=C!	8		X:=CN~!	1		Z:=C	19	
X:=Z	12	X:=E~!	4		A:=B!	13		X:=CE~!			Z:=X	1	
CRCR!	2	X:=W~!	4		A:=X!	1	1	X:=CW~!	1		SCRR!	2	
PANS!	2	X:=S~!	4		A:=Y!			X:=CS~!			SCRC!		

Table 12

Algorithm Count for Each Instruction

	+		+	-		+	-		+	-		+	-
M:=C	50	X:=A!			A:=A!		8	B:=A!	54		Y:=A!		
A:=M	25	X:=B	4		A:=B			B:=B			Y:=B	4	
M:=B		X:=X		33	A:=X	33	8	B:=X	4		Y:=X	4	
B:=M		X:=Y	4		A:=Y			B:=Y			Y:=Y		17
M:=X	67	X:=$\overline{X+Y}$		38	A:=$\overline{X+Y}$	4	17	B:=$\overline{X+Y}$			Y:=$\overline{X+Y}$		29
X:=M	88	X:=$\overline{X \wedge Y}$		8	A:=$\overline{X \wedge Y}$			B:=$\overline{X \wedge Y}$			Y:=$\overline{X \wedge Y}$		4
M:=Y	33	X:=$\overline{X \vee Y}$		4	A:=$\overline{X \vee Y}$			B:=$\overline{X \vee Y}$			Y:=$\overline{X \vee Y}$		
Y:=M	42	X:=C	58		A:=C			B:=C			Y:=C	38	8
M:=A!		X:=N	21	4	A:=N			B:=N			Y:=N		
A:=M!	8	X:=E	21		A:=E			B:=E			Y:=E		
M:=B!		X:=W		4	A:=W			B:=W			Y:=W		
B:=M!		X:=S			A:=S			B:=S			Y:=S		
		X:=N¯!	17		A:=C!	33		X:=CN¯!	4		Z:=C	79	
X:=Z	50	X:=E¯!	17		A:=B!	54		X:=CE¯!			Z:=X	4	
CRCR!	8	X:=W¯!	17		A:=X!	4	4	X:=CW¯!	4		SCRR!	8	
PANS!	8	X:=S¯!	17		A:=Y!			X:=CS¯!			SCRC!		

Table 13

Percent of Algorithms Using Each Instruction

instruction appeared in 66.7 percent of the algorithms for a total of 98 occurrences, while the X:=Z instruction that only occurred 24 times appeared in 50 percent of the algorithms. Thus the two instructions have almost equal generality even though the latter is only used about one fourth as often as the former.

Table 14 presents the average number of occurrences per algorithm for each instruction. This was obtained by dividing the values in table 11 by those in table 12. This gives an indication of the average frequency of use for each instruction. This provides a corrected measure of instruction usage frequency. For example, even though M:=X has the highest number of raw occurrences, it can be seen from the figure that Y:=X+Y is actually the most frequently used instruction on average.

Discussion of Instruction Set Usage

We will examine the instruction usage statistics by looking at the average use values. The number of occurrences per algorithm for each instruction provides several natural groupings of the instructions, which simplifies the discussion. Each of these groups will be examined in turn, starting with the group of instructions

	+		+	-		+	-		+	-		+	-
M:=C	3.2	X:=A!			A:=A!		1.0	B:=A!	1.5		Y:=A!		
A:=M	1.5	X:=B	6.0		A:=B			B:=B			Y:=B	2.0	
M:=B		X:=X		3.5	A:=X	2.5	1.0	B:=X	8.0		Y:=X	1.0	
B:=M		X:=Y	4.0		A:=Y			B:=Y			Y:=Y		1.3
M:=X	6.1	X:= $\overline{X+Y}$		2.1	A:= $\overline{X+Y}$	1.0	1.0	B:= $\overline{X+Y}$			Y:= $\overline{X+Y}$		8.7
X:=M	3.9	X:= $\overline{X \wedge Y}$		3.0	A:= $\overline{X \wedge Y}$			B:= $\overline{X \wedge Y}$			Y:= $\overline{X \wedge Y}$		1.0
M:=Y	7.9	X:= $\overline{X \vee Y}$		1.0	A:= $\overline{X \vee Y}$			B:= $\overline{X \vee Y}$			Y:= $\overline{X \vee Y}$		
Y:=M	7.0	X:=C	5.5		A:=C			B:=C			Y:=C	1.6	1.0
M:=A!		X:=N	7.0	1.0	A:=N			B:=N			Y:=N		
A:=M!	1.0	X:=E	7.0		A:=E			B:=E			Y:=E		
M:=B!		X:=W		1.0	A:=W			B:=W			Y:=W		
B:=M!		X:=S			A:=S			B:=S			Y:=S		
		X:=N~!	2.3		A:=C!	2.1		X:=CN~!	1.0		Z:=C	3.1	
X:=Z	2.0	X:=E~!	1.3		A:=B!	1.9		X:=CE~!			Z:=X	1.0	
CRCR!	2.0	X:=W~!	2.0		A:=X!	2.0	1.0	X:=CW~!	1.0		SCRR!	2.0	
PANS!	4.0	X:=S~!	2.5		A:=Y!			X:=CS~!			SCRC!		

Table 14

Average Number of Occurences Per Algorithm

that occur most frequently when averaged over all of the algorithms that they occur in. Note that average occurrences figure can be somewhat misleading. Consider that instructions which are used in many algorithms will have a much lower average occurrence figure, while those that occur many times in just a few algorithms will have much higher values. Nonetheless, most architects would consider the more widely used instruction to be of greater importance in evaluating an architecture. Thus, the average instruction occurrence number should not be considered as a figure of merit, but rather as a guide that points out anomalies in the instruction usage. The anomalies must then be subjected to careful analysis, which is the method that has been employed here.

According to table 14 the most frequently used instruction is Y:=X+Y. In examining the uses of this instruction it becomes apparent why this is so. The add operation underlies every arithmetic function performed in the CAAPP cells. The reason that the Y register is the most popular destination for summation is that the X register is frequently occupied with some datum that is either being communicated to or from neighbors or being transmitted to the controller via Some/None or a response count.

The second and third most frequently used instructions

are the transfers to and from memory, respectively, involving the Y register. This is strongly related to the high frequency of use of the Y:=X+Y operation. In most cases of arithmetic operations in the cells, X is loaded with one operand (frequently a broadcast bit) and Y is loaded with a second operand (usually a bit from memory). The result is computed and stored in Y, then immediately transferred back into memory, usually to the same location from which the original Y bit was taken. This indicates that X and Y are simply being used as holding registers for input to the ALU of broadcast and memory bits. What is really happening for most of the arithmetic functions in the cell is that an M:=M+C operation is being performed in a roundabout way. This is true for all of the algorithms that perform arithmetic or comparisons with a broadcast comparand and a value in memory. Almost every algorithm examined in the last chapter does this, with the exception of those that do strictly memory to memory operations (such as Add Fields).

The next most frequently performed operations are the on-chip transfers of neighboring (North and East) X register values into each cell's own X register. This is an unusual case because the operations have only one real use: In the Load X-Y Coordinate Grid algorithm they are each used seven

times in order to shift a pattern of bits into the cells to establish the on-chip segment of the address in each direction. Thus their frequency of use is an artifact of the aspect of the design that limits direct external enable control to the 64 cell group on a chip.

Following the on-chip neighbor transfers, it can be seen that the next most frequently used instruction is transferring the contents of the X register to a memory location. This instruction actually occurs the most frequently according to the raw count. This stems from a variety of uses that combine to produce this number. One frequent use for this instruction is saving bits into memory that have been transferred in from neighboring cells. This points out the utility that would be provided by an instruction that would fetch a bit from memory, send it out to a neighbor and then store the bit being received from the corresponding neighbor in the opposite direction, into the same memory cell. For example, this would increase the speed of the Sobel and Gaussian Filter operations and especially the Geocorrection algorithm. Another use for this instruction is storing the overflow bit (the last value of Z at the end of an arithmetic operation). The value in Z must be transferred to X before it can go anywhere else. This is an indication of the need for instructions which

allow Z to be transferred directly to memory. Some of the arithmetic operations do put their results into X and these are then stored into memory. This again points to the usefulness of being able to directly write from the ALU into memory.

Transfers between X and B appear to be very popular according to the averages, but this is because they are used several times in a single algorithm. The actual usage is not terribly significant.

Next on the list is transferring a broadcast comparand bit into X. Although nearly 60 percent of the algorithms use this instruction, they all use it in the same way: C is loaded into X simply to put it into a position from which it can be operated upon by the ALU. This may be either an arithmetic operation, or a logical operation (as in the case of comparisons). This is evidence of the need for somehow making C selectable as a direct input to the ALU.

At this point there is an abrupt drop in the occurrences per algorithm figures. The group of instructions which have average usage figures between three and four represent a transition from more general instructions to those that are specific to only a few applications. The first of these, for example, is X:=Y, which has an average of four

occurrences because it is used four times in only a single application. A similar case is the PANS instruction which is only used in counting responders.

The third among this group is X:=M which is just the opposite of the first two. It is, however, the second most used (in terms of raw occurrences) of all of the instructions and is the most broadly used, appearing in 87.5 percent of the applications. (The fact that it is used in so many aalgorithms is, of course, the reason that it has a lower average occurrence figure.) This is because X is the "busiest" of the registers. It is used for passing information to and from neighbors, as the response bit for tests made by the controller, and as one of the two main inputs to the ALU. In all of these cases it is common that values from memory will be loaded into X. This forms something of a bottleneck in processing on the cell. Often times the registers had to be rearranged in the algorithms to allow some piece of data to pass through X on its way to somewhere else. Then once X was again free, the registers had to be restored to their original state. This bottleneck can be alleviated by allowing memory to be selected as an ALU input, and as a source for neighbor communications. This would result in X being used primarily as the response bit, and to a lesser degree as a temporary holding bit for

intermediate results of logical operations. This is much more in keeping with the original intent of the design.

Next most frequent in this group is the X:=-X instruction. The popularity of this operation stems at least in part from the fact that memory cannot be loaded or stored in complement form. A common pattern in the algorithms was the loading of a bit from M into X, followed by a complement X instruction. It was less frequent to complement X before storing because most of the other operations allow the result to be complemented. The other time that this was used was when a logical operation required an inverted input and for one reason or another, X was not available in this form (for example, because it had previously been used as an input to another operation in true form). This would tend to indicate a need for selectable inversion of the data sources. Having this would eliminate the extra step currently required to put the data in proper form.

Following this comes the M:=C instruction. The primary purpose of this is to initialize a field in memory with some broadcast value. This is a fundamental operation for which there is no shortcut.

Another instruction from this subset which has a high

occurrence count and breadth, and therefore only a moderately high occurrence per algorithm figure, is the Z := C operation. This is used to initialize Z for arithmetic operations. A second function is to keep Z cleared during comparisons. This accounts for much of the usage of this instruction. The reason for doing this is that the ALU does not provide a pure exclusive OR operation (the logical function used in most comparisons). As a simplifying measure, the design makes use of the full adder in the ALU to provide the equivalent of an exclusive OR by clearing the carry bit (Z) after each operation. This effectively doubles the time required for the simpler comparison operations. A carry inhibiting add operation (XOR) would significantly improve the performance of the processor in performing comparison searches.

The last of this group is the X:=X^Y instruction. This is a basic logical operation used to combine response patterns. It is also used in circumstances where the desired effect is equivalent to having an instruction that performs an M:=M^C operation, to allow a broadcast mask to be applied to a memory field. This is further evidence for the utility of allowing M and C to be selected as inputs to the ALU with direct write back into memory.

The remaining instructions occur on average less than

three times per application. At this level generality (breadth of use) becomes more important. Consider the X:=Z operation which occurs 24 times among 50 percent of the algorithms. This is certainly a more important operation than Y:=B which occurs twice in one application, even though they both occur two times per algorithm on average.

The two instructions from this group which are most widely used are A:=B! and B:=A!. These are used for saving and restoring an activity mask. Essentially they can be thought of as providing a single level of subroutine call capability. When one of the routines is entered it saves the current activity pattern and frees the A register for unrestricted use within the subroutine. Upon completion the original pattern is restored so that processing may continue as before. If more levels are required, the contents of B may be saved to memory in a stack like manner before calling another subroutine. Sometimes it would also be useful to be able to swap A and B, as in the execution of a parallel IF-THEN-ELSE. Currently it is required that one of the registers be temporarily saved somewhere else for them to exchange contents.

The next most widely used instruction in this group is X:=Z. For the most part this is done to put the content of Z into a position from which it can be stored to memory.

Again, this two instruction operation can be shortened to a single instruction. This would simply involve provision for storing Z directly to memory.

The next most widely used in this group are the X:=X+Y and Y:=C instructions. The former is used for arithmetic and comparisons whenever X is not otherwise occupied by its communication and response functions. It is most often a substitute for an M:=M+C operation or an M(I):=M(I)+M(J) operation (in the case of field to field arithmetic). Although two address operations add considerable complexity to the hardware, it is not unreasonable to have an M(I):=X+M(I) operation that assumes X is previously loaded with M(J). The Y:=C instruction is used to load a broadcast constant into Y prior to some arithmetic or logical operation. If an M:=M+C instruction became available then Y:=C would have reduced significance.

At 33.3 percent, the A:=X and A:=C! instructions are the next most widely used. The former is most heavily used in the comparisons where it is necessary to freeze the value in the response register (X) when a certain condition is detected in some set of cells, allowing other cells to then proceed with the comparison. The only way to speed this up would be to allow simultaneous writing of a result into X and A. The A:=C! instruction has the purpose of resetting

the activity mask to all active following some operation that turns off a group of cells. This is a fundamental operation in the CAAPP for which there is no substitute.

The A:=M operation is the next most widely used of this group, occurring in 25 percent of the algorithms. This is another fundamental operation. It is used to recall a previously stored activity mask in such a way that it is logically ANDed with the current mask. This is often a result of a query of the form: "Find all cells that are currently active which also responded to this previously tested criterion."

The remainder of the instructions that were used at all fall into a group that are used infrequently and not very widely. These include the off-chip neighbor transfers, which were avoided as much as possible because they are costly in terms of time. If made faster, the off-chip transfers would undoubtedly be used much more. Because the off-chip shifts often occur in the innermost nested loops of the algorithms, any increase in shift speed has a significant effect on the overall algorithm speed. Also among the group of infrequently used instructionas are the special response count operations CRCR! and SCRR!, some of the logical operations and register to register transfer instructions. The utility of these is reasonably evident --

there are situations in which they are needed and which would require execution of a sequence of other instructions and additional storage if they did not exist. It can be surmised that in an even larger sample of algorithms these might produce more impressive statistics.

The final instruction group in this analysis is made up of those instructions that do not appear anywhere in any of the algorithms. The subset of these that most stands out is the set of instructions which perform on-chip transfers from neighboring cells to the A, B and Y registers. On-chip transfers are always between X registers in the algorithms presented, because the on-chip transfer is used to pass a pattern that is introduced at the edge across the rest of the cells in the chip. Since X is the only communication source, it is most efficient to simply keep the data in the X registers rather than passing it to another register from whence it will only need to be loaded into X in order to continue on its way. Cross register on-chip communication is thus a rarely used feature and could easily be eliminated from the architecture.

Another group that is rarely used is the inverted logical operation instruction set. Some of this can be attributed to the natural tendency to think in terms of ANDs and ORs rather than NANDs and NORs. It is most obvious,

however, that the B register is never the destination of an ALU result. This is because its primary use is as a backing store for A. This subgroup could be eliminated from the instruction set if necessary, although doing so would even further reduce the symmetry of the register operations.

A third group that stands out is the set of instructions that transfer values between B and memory. This probably results from the algorithm's failure to use multiple levels of subroutines that each require their own activity pattern. Presumably more complex algorithms will then use instructions from this set to stack these patterns.

The rest of the instructions are either oddballs (like SCRC! and the No-Ops) that are included for purposes of simplifying the instruction set encoding, or are left out because of quirks in the selection of algorithms or the style of this particular programmer.

Summary of Findings

From the above discussion a collection of problems and potential solutions can be gathered. The most significant of these is the bottleneck at the inputs to the ALU. This causes about a 67 percent reduction in speed for arithmetic

and logical operations with respect to the optimum performance. Instead of a single X:=M+C operation, a series of three instructions must be performed for each bit position in the operands. This is further slowed by the fact that there is no provision for directly writing back into the same memory location from which the memory operand is taken. The addition of a read-modify-write capability along with M and C ALU inputs would reduce memory-comparand ALU operation time by 75 percent. This would essentially give the optimum speed of one instruction cycle per bit. Memory-memory ALU operations would take 50 percent less time (two instruction cycles per bit) than they currently do.

Restricting communications to values in the X register also creates a bottleneck. With a read-modify-write memory it would be possible to directly transfer memory bits between neighbors. For on-chip shifts this amounts to a 67 percent saving in time. For off-chip transfers the time saved is 20 percent. This could also be extended to permit direct transfers of activity. Currently a shift of activity requires transferring A to X before the shift and then back to A afterwards. The same reduction in time would be obtained as for the memory transfers.

Comparisons would be much faster if a carry inhibited add (XOR) operation was available. This would provide about

a 20 percent speed-up over the current time. The M and C inputs to the ALU would provide an additional 40 percent increase in speed, which is the optimal time for inequality comparisons. The exact match, however, can be made another 20 percent faster with the addition of an exchange A and B instruction. The A-B exchange would reduce the total time for an exact match to approximately one instruction cycle per bit (there is a 3 instruction cycle overhead associated with each exact match operation).

A small amount of overhead in the arithmetic operations can be eliminated by the addition of an instruction that allows Z to be directly stored to memory. This eliminates an intermediate transfer to X that is currently necessary in order to save the overflow status after an arithmetic operation. More importantly the current situation forces X to be kept clear of other data (such as a response pattern) in order to allow the overflow status to pass through on its way to memory. This is just one more bottleneck in the information flow patterns within the cell.

Further time can be saved in the subtraction and division operations if provision is made for complementing memory at the input to the ALU. Currently the memory operand must be loaded to a register in true form and then recirculated to that register via the ALU to put it in

complement form prior to being operated upon. Allowing complemented ALU input would reduce the subtraction time to the same level as the addition operation.

The cross-register on-chip neighbor transfer operations can be eliminated from the instruction set. Only transfers from X to X, M to M and A to A between cells are necessary. It may be desirable to provide some means of complementing a value in transit but this is probably not worth any extra hardware investment.

Other Possible Architectural Enhancements

This section will examine other aspects of the algorithms presented in chapter five to determine other changes to the architecture that will improve overall performance. Each of these changes will be discussed in terms of the expected improvement and the difficulty of implementation.

Memory size. First on this list is expanding the size of memory. Few of the algorithms placed any restriction on the number of memory bits that were available. Careful analysis reveals that several of the more interesting applications (such as Satellite Image Geocorrection) are

impossible to implement with only 32 bits of memory; actually they can all be implemented, but the implementation in some cases restricts the size of a data element to a useless amount. At least double the number of bits are necessary for some of the applications. Even 64 bits is somewhat restrictive. Thus a cell memory size of 128 bits is suggested. Obviously the more the merrier, however there are limits to the amount of memory that can be fit onto the available silicon real estate.

Currently the chip has 32 bits of memory per cell. This is done by interleaving the memory stripes that belong to the two columns of processing elements (see chapter four for the device floorplan), which allows the sharing of the address decoder between the two columns. These memory cells are, however, a very conservative, static, single metal layer design. By redesigning the cells for a two metal layer process, using a sense amplifier to detect output and possibly going to a dynamic design it should be possible to reduce the overall cell size by two thirds. This would allow the extra memory to reside in an area only 30 percent larger than it currently occupies. Because the memory currently takes up about 30 percent of the total space, this would result in roughly a 10 percent overall increase in required space. Although this is not unreasonable, it

should be noted that the design of sophisticated memories is a nontrivial problem. Accomplishing this will require access to considerably more sophisticated design and simulation equipment than is currently available in our development environment. With the tools at hand, perhaps the best that could be expected is a 64 bit memory. Nonetheless it is probably worthwhile to pursue the 128 bit design. An alternative is to design the processor chip without memory and to then build a special purpose memory chip that would be mounted next to it in the same package. This would entail some 128 connections between the two chips. Not only is this impractical, but it would make it much more difficult to achieve the desired instruction cycle time.

Power reduction. A simple change that has nothing to do with the algorithms involves reversing the logic sense on the bit select lines for the memory. Because of some shortsightedness in doing the design, and the time constraints involved, the current memory sense is such that all but one of the high powered drivers is active at all times. The only inactive driver is the one that is selected for accessing a particular memory bit. Should the memory be redesigned, this should be taken into consideration. This will require reversing the logic sense of the address

decoder as well, but the savings in heat dissipation should more than make up for this in the long run.

Further reduction in heat dissipation could be achieved by converting the chip design to Complementary MOS (CMOS) technology. This would require a considerable investment in time to accomplish. Because the current design is within the limits of forced air cooling capacity, there is no overwhelming reason to make this investment at this time.

Faster response count. Even though the response count is within two orders of magnitude of the optimum, there are some applications which could benefit from a considerably faster count. Specifically adaptive histogram guided segmentation of images would benefit. Currently a 256 bucket histogram requires 8 milliseconds to develop. This results in a limit of four histograms per frame time. Changing the bottom row circuitry to use a tree of adders instead of a pipelined addition (the PAWE instruction) would reduce the per bucket time by 6.7 microseconds. This would provide just over a 20 percent speed increase and would allow an additional histogram to be performed in one video frame time. A further 6.3 microseconds could be shaved off of the time by using an adder tree to build the on-chip count instead of using the shift and count method. The combination of this with the bottom row adder tree and some

speed increase that will be obtained as a result of enhancements proposed in the previous section, should result in a doubling of the number of histograms that can be performed in a frame time. The complexity involved in implementing these changes is not insignificant. The bottom row adder tree will be quite large but it can be built with off the shelf components. The on-chip adder tree is another matter. This will require an output from each of the 64 processors to converge on a combinational circuit. Though not difficult to design by itself, this circuit must somehow be packed into a geometry that is not natural for it. The routing of all of the wires that converge on it is another problem. The width of 64 metal lines on a chip is considerable (on the order of half of the width of a processing element). On the other hand, there may be some way to distribute the tree so that only a few lines need to be run for any distance. This change is probably worth some exploratory design work but may not be feasible in the end.

To increase the speed of the response count any further requires overcoming the time to perform the North to South Pipelined Addition (PANS) operation. This accounts for 12.8 microseconds per histogram bucket. The problem with most schemes for beating this time is that they require the chips to have some design feature that is dependent on the overall

size of the array. There are, however, a couple of solutions that can overcome this. The simplest is to reduce the length of the pipeline. This is done by having the column counts be collected at the bottom edge of each board, then an expansion of the adder tree that develops the final count can be used to quickly sum these. This will reduce the column summing time to 1.8 microseconds. With adder trees on either side of this it should be possible to develop a response count in 2.0 microseconds. This will allow 64 histograms to be performed in one frame time. An additional benefit of this technique is that if the on-board adder tree output is made available to the controller, it becomes possible for the controller to extract counts for each of the 64 board sized sub-arrays in the image. This would be useful in the Flow Field Extraction problem which requires many counts of small neighborhoods.

An alternative, to forming the final sum of the circuit board level counts with an adder tree, is to poll the boards and form the sum through a series of additions. This is probably less expensive than building a large adder tree, and not a great deal slower (because the number of boards is relatively small compared to the number of cells). The optimum would, of course, be a full adder tree running from the cell level all the way up to the controller. Even so,

this would probably require about Ø.2 microseconds settling time for the fastest logic elements (for example high speed Emitter Coupled Logic -- ECL). With less exotic elements, this time is likely to approach Ø.5 microseconds and possibly more. Thus there is little reason to consider implementing such a scheme.

Even if it were possible to count responders in less time, it is not clear that this is at all useful. Consider that the minimum time for an eight bit exact match, even with all of the other architectural enhancements described in the previous section, is still about one microsecond. Therefore even if an independently operating Ø.1 microsecond count mechanism were available, the minimum time for a 256 bucket histogram would still be 256 microseconds. It is difficult to think of any application in which a long series of meaningful response sets could be computed, for counting, at a rate of more than one set per microsecond. This is conceivable for a short series of counts, but in these cases the total lost time is probably insignificant.

Expansion of the ALU inputs. Moving on to another area for possible improvement, it has already been shown that there is a considerable bottleneck at the inputs to the ALU. The simplest solution to this is to allow one of the ALU inputs to select between X and memory, and the other input

to select between Y and the broadcast comparand. This requires a fairly small increase in circuit complexity, consisting essentially of a two way selector on each of the ALU inputs. This is at least partially offset by the fact that the memory and comparand select circuits in the current design can then be eliminated. Expanding the bottleneck even further, it may be useful to allow any combination of the five registers, memory or comparand to be selected by the two ALU inputs. This would require the addition of two larger selector circuits to the processing element. Taking this to its logical extreme, the neighbor values could also be input to the selectors. This would only be feasible if full width communication was provided for between chips. Then it would be possible, for example, to form the AND of the North and South neighbors. This would reduce polling of the local neighborhood to three operations per bit in some cases (of course most of the speedup here is gained by not having to multiplex the communications at chip boundaries). In addition to the pin-out problems that this produces, the selectors that are required for the ALU inputs would be considerably larger. This would, however, be partially offset by the elimination of several of the current selectors and one of the interconnection networks.

<u>Ability to complement ALU inputs</u>. If the ALU inputs are

expanded, it would be reasonable to also provide for selectively complementing them. This is because the complemented inputs considerably expand the set of operations that the ALU can perform. Following a selector with an XOR gate that allows complementing is an efficient way to provide complement forms for multiple data sources.

Full width interchip communications. Full width communications between chips is another very desirable feature. This would speed up all of the local neighbor operations, such as the Sobel operator, by a considerable amount. It would also eliminate the need for the second communications network on the chip (the zig-zag network) with the result being a substantial savings in silicon area. Unfortunately this also requires 28 additional pins on the package, with 28 matching pads on the chip. The space saved is thus probably more than taken up by the wiring pads. The total pin count for a full width communication design will exceed the original design restrictions and force the use of more exotic packaging technology. Although this is now becoming more widely available, there is still the problem of how to run over 300 wires off of each circuit board. This adds up to around 20 thousand wires running between boards in the full array. Still, the benefits that this would produce may justify some

exploration of alternative circuit board interconnection technology.

Long distance communications between cells. Should full width communications become feasible, an interesting avenue of exploration is the idea of long distance communication across the grid. This mechanism would greatly improve the timing of algorithms that need to transmit information between distant points in the array. This would include the Satellite Image Geocorrection problem, Graph Processing, Neural Network Simulations, Semantic Network Processing and Logical Inferencing. The basic idea of this scheme is that each row (and column) of processors forms an interruptible bus. Through some query or computed activity pattern a group of "bus talkers" is established. All of the other elements in the array are listeners. For that matter, the talkers can also listen. Consider a row of processing elements as an example bus. The leftmost talker loads a value into its communication register (probably X, although it might be advantageous to set aside another register expressly for this purpose). It must also select a communication direction (all talkers transmit in the same direction at any one time), for example, East. To its East are some number (possibly zero) of listeners and either another talker or the edge of the array (if the array is

cylindrically wrapped, the next talker may be itself). At this point, some subset of the listeners is selected and copies the bit being transmitted into their local memory. The talker to the East may be one of these listeners. Each talker in effect breaks the bus. Data can pass from one talker no further than to the next talker because the next talker is itself using the far side of the bus to transmit its own information. Nonetheless, if there are only a few talkers on the bus, their information can travel a long distance before it is stopped. The reason that the talkers themselves can be listeners is to permit relays of information.

To see how this works, consider the geocorrection problem. Each pixel has the address of its new location. Each cell also has its own address. The difference in X coordinates is computed and a find greatest operation is performed. These cells then become the talkers. Presumably there are only a few of these, although degenerate cases exist where most of the pixels have to move the same distance. The talkers transmit their destination addresses and pixel values bit serially to all of the listeners (including the next talker down the line). The listeners then compare the received address values to their own addresses. Those that match in the one direction become

talkers in one of the two orthogonal directions. They relay the destination address and pixel value along in that direction until it reaches its correct location and settles down. They then talk in the original direction to the original talkers, letting them know that they need not relay the value any further. Then the original talkers that have not been so notified become active again and relay the values they received from the West to all of the listeners to the East, and the process is repeated until all of the values have reached their destinations. Note that this process speeds up as it progresses because some of the talkers will have dropped out, thereby lengthening the busses. After this the find greatest search can again be applied to select the next group of pixels. In the end, all of the pixels will be selected that have only a short distance to move. Even if these are the most numerous, it will have little effect on the algorithm (and in fact improves its execution speed) because they will not have to transmit as far in order to reach their destinations.

In the original algorithm the timing depended upon the maximum distance that a pixel had to travel to reach its destination. With this scheme the timing depends upon the frequency of occurrence of the distances. If the pixels with the longest distances to travel are few in number, they

will nonetheless reach their destinations very quickly. In semantic networks it is often the case that a few nodes initiate activity and that it is desired to spread this initial activity quickly so that many other nodes may act on it in parallel. This can be quite readily accomplished via this mechanism. Of course, if the number of transmitting nodes is small, they could simply be extracted by the controller and broadcast to the array. There are a number of graph theoretic problems that also find a natural mapping onto the long distance type of communications network. Levitan [72] has explored a number of these algorithms on broadcast protocol architectures which are similar to this. It is interesting to note that the zig-zag (or skewed torus) wrapparound for the array would permit a single talker cell to transmit information to the entire array without having to go through the controller.

In terms of circuitry, this mechanism is relatively simple to construct but may be difficult to run with any speed. Each cell is simply provided with a talker/listener control register (possibly X although a dedicated register may be desirable), a pass transistor that is controlled by this register, a buffer to amplify the signal beyond the pass transistor, and I/O connections on either side of the transistor to permit transmission and reception. There

would actually be two sets of this hardware, one for rows and the other for columns. The problem is that each of the buffers introduces a gate delay time. Over long distances these will inevitably add up to well over the time allotted for an instruction cycle. Thus it may be necessary to have some special controller state that creates a long period instruction. This sort of complexity often leads to timing troubles and would require extensive simulation to get right.

Vector broadcast registers. If long distance communication becomes available, there is another change that is relatively simple but very useful. At the top and left edges of the array a vector of registers could be added. The array could then be set into a mode in which all of the processors are listeners and the registers are the talkers (only one of the vectors could talk at once, of course). This would permit array operations in which vectors are transmitted via rows or columns to the processors. Thus it becomes possible to quickly multiply a two dimensional array by a vector of numbers. This has a number of interesting applications, such as synthesis of holograms, and linear programming. Initial indications are that this would reduce the simplex method from cubic to linear time for constraint sets that can be fit into the

array.

Direct selection of individual cells by the controller. It should be mentioned that any system which employs full width communications between chips ought also to include full X-Y select capability at the cell-by-cell level. This adds even more pins to the chips (16 select lines are needed as opposed to two in the current design), which only serves to strengthen the point that this little group of changes will require exotic packaging and interconnection technologies. Direct cell selection capablility would, however, increase the speed of the select first and the load X-Y grid operations.

Build chips from 16 cell blocks of processors. Coming back down to Earth, there are a few remaining changes that are simple to implement and which provide a variety of features. One of these is intended to improve chip yield and perhaps allow fault tolerant versions of the CAAPP to be built. This involves using a larger die onto which six copies of the 16 cell design are placed. This provides 96 processors on a chip. At time of manufacture, all of these are tested for proper functional operation and speed. Many of the 16 cell groups will have faulty cells, however, as long as there are four usable groups (67 percent), the chip can be used. This will require either electrical link

fusing or laser trimming to eliminate extra connections to the faulty groups. It is also conceivable that there would be a market for chips with fewer than 64 cells.

The idea of patching out bad quadrants could be extended to construct a fault tolerant version of the machine. Some of the chips, though very few, will have 96 usable processing elements. If special provisions are made for dynamically selecting two of the groups to be ignored, then it should be possible to switch in a backup group when a fault is detected in any of the active groups. Furthermore this is a double fault system in that as many as two groups can fail before a chip becomes unusable. Even so, it may be possible to ignore an isolated faulty processor in some applications. In image processing, for example, the array is sufficiently fine grained that even if its fault affects computation in its local neighborhood, an isolated faulty processor will cause no more damage to the image than, say, noise in the imaging system due to a burned spot on the sensor (and probably a good deal less). If extreme fault tolerance is desired, the array could be built with an extra row of chips on each board and, for that matter, an extra row of boards which could be selected should a full chip or board be detected as faulty.

<u>Dual ported cell memory</u>. Another change, this one

affecting overall processing speed, is to dual port a segment of cell memory. This would most likely be either 16 or 32 bits in size. The purpose here is to allow the controller to access a segment of the memory while the processors are operating on the remainder. Currently the array must sit idle while data is being loaded or dumped. This would permit a new image or data field to be loaded or dumped while another is being processed. Thus a certain amount of pipelining is possible. With the current design, approximately a video frame time is required to load the array. Thus it takes a minimum of two frame times plus processing time to pass an image through the processor. During the two frame times, no other processing is accomplished. With the dual ported memory, the total processing time does not decrease, however the throughput increases because the array can be active during loading and dumping. This increases the throughput from roughly ten images per second to thirty images per second.

In terms of implementation complexity, this change has some significant problems. Any tinkering with the memory design, especially of the sort that makes some of the cells act differently, is just asking for trouble. It might be easier to double port all of the memory and provide a mechanism for selecting 16 or 32 bit banks that are under

the control of the array load/dump circuitry. This will also add complexity to the address decoder circuitry. The potential benefits, however, make it worthwhile to seriously explore possible solutions to these problems.

Multiple ALU result destinations. A simpler change, though not so obviously beneficial, is to permit any combination of the registers to be a destination for an ALU result. This is mostly useful for clearing several registers at once -- something that is occasionally done at initialization. This would, however, substantially enlarge the instruction set with only limited benefit. An increase in the instruction set size also means an increase in the pin count on the chips.

Multiple controllers connected to subarrays. The last change on this list of possibilities is another of the more complex variety. The main bottleneck in the Iconic to Symbolic Transform problem is that the image features must all pass through the central controller in order to be rebroadcast to the array. This is a variation on the problem known as the "von Neumann bottleneck". The CAAPP eliminates one side of the bottleneck: The controller can only fetch one datum at a time but it can broadcast it in parallel to a large number of storage cells. For operations that take place solely in the array, there is no bottleneck,

because each datum has its own processor. In applications which require processing of combinations of data that are distant from each other in the array, there is a bottleneck. Either the data must be fetched individually by the controller (bottleneck) and rebroadcast, or the data must traverse a long and time consuming communications path (bottleneck) through the array. Even the long distance communication scheme does not entirely solve this problem because it provides very little speed increase when large numbers of cells need to communicate.

The solution in this case is to provide multiple controllers that can read data in parallel from the array and rebroadcast these or pass them via shortcuts to their destinations. The optimum would be to have controllers assigned to groups of cells according to some response pattern in the cells. Unfortunately this isn't feasible. It would require a separate instruction and address decoder for each controller on every chip. Even if silicon real estate were unlimited, such a chip would have to have hundreds of pins in order to talk to all of those controllers at once. A more reasonable solution is to divide the array into subarrays and assign a controller to each. For example, a controller might be provided for each board in the CAAPP array. In this situation, the local

controllers would then be able to fetch and broadcast directly all communications that originate on the board which have destinations that are also on the board. Those that pass off of the board would have to be routed to the other controllers, or to the central controller for general broadcast to the entire array. This arrangement is diagrammed in figure 51.

One interesting result of this design is that the local controllers can also read the board level response counts. This would then permit them to confer and develop a strategy for adaptive enhancement on a region by region basis and to apply different enhancements to different areas in parallel. Again, the problem of regions that cross control boundaries would have to be handled specially.

Carrying this a step further, a number of interesting computer vision algorithms have been proposed which make use of a hypothetical pyramid architecture. For the above configuration, only an additional 21 controller type processors would be required to complete the pyramid. This would give a total of 85 such processors plus the central controller. (The central controller is a special processor which can talk to the entire CAAPP array and the pyramid, while the peak processor on the pyramid can only talk to the level below it, and the central controller.) With this many

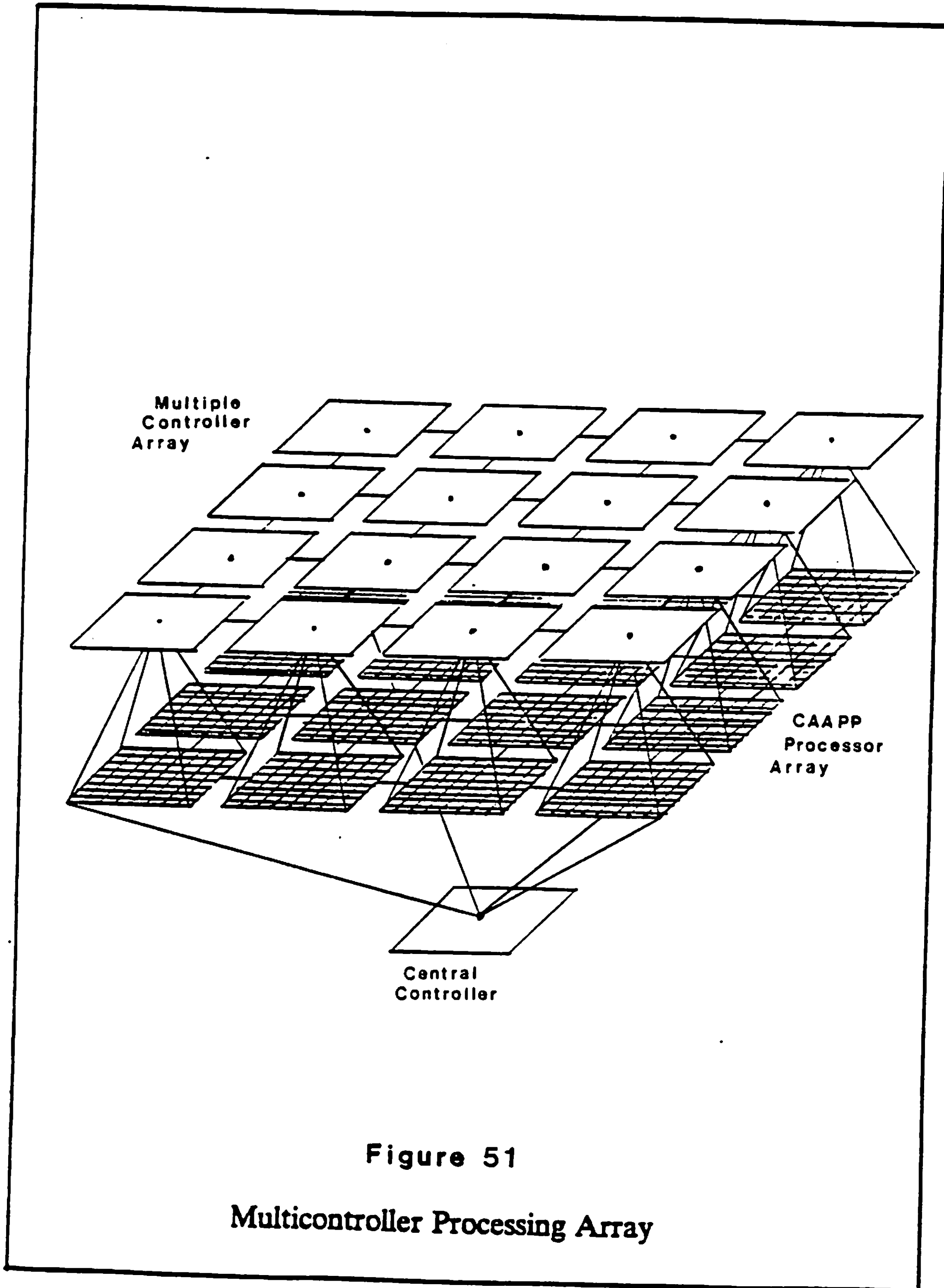


Figure 51

Multicontroller Processing Array

controller class processors it would probably be most cost effective to use a standard microprocessor chip to implement this. The microprocessor would have to be a fast (10 MHz) part with at least a 32 bit internal architecture and an external 16 bit data path.

The cost and complexity of the multicontroller array is far from trivial. However, this method of implementing a pyramid is far simpler than trying to build such a structure on a single flat piece of silicon. Most pyramid architectures use a smaller interlevel ratio, such as four to one, rather than the 4096 to one ratio that this design employs. Rather than having processors on higher levels act as controllers, these other designs have them act as simply another plane of cells, so that all of the levels are homogenous. Cells on higher planes collect or distribute information from or to the cell groups below them, which requires a large number of communication paths between levels. Because current packaging technology cannot provide enough pins to support this communication, it is necessary to put several levels of the pyramid on each chip. Even for just a few levels on a chip, the pyramid is a complicated geometry to try to wire on the silicon plane. It also seems that there is no reason to have all of the processors in the pyramid be identical. In fact it is desirable to have the

processing power increase for higher levels in the structure, otherwise a throughput bottleneck is likely to result. Thus, the multicontroller array concept, despite its cost and complexity, would seem to be more feasible to construct and more useful for vision processing than a homogenous pyramid architecture.

Evaluation of the Proposed Enhancements

Obviously no buildable design can possibly accomodate all of the 27 enhancements that have been proposed in these last two sections. Depending upon the set of restrictions imposed on the implementation, however, different subsets can be incorporated into the design. What follows is a set of three such designs which progressively relax the restrictions. These will be called the Conservative, Intermediate and Advanced designs.

The particular choice of which subset of enhancements would be included in each of the designs was based on a comparison of the estimated cost of the enhancement with the estimated benefits. Each enhancement was assigned a cost factor in the range zero to four. This was based on a partly subjective evaluation of the amount of silicon space, number of pins, difficulty of implementation and additional

power requirements that would result from it. Factor zero indicates an enhancement that has no net cost associated with it. Factor four indicates that the enhancement would push the technology to the limit or beyond.

The chief benefit examined with respect to each enhancement was the resulting increase in execution speed. This was based on an evaluation of how the enhancement would speed up a set of 16 macro operations that are used frequently in the algorithms of chapter five. For each of the macros a usage count was taken among the algorithms. Due to the wide range of values, this was scaled by taking the base ten logarithm to produce a weight that ranges from 1 to 6. The number of algorithms in which each macro is used (independent of the number of times used) was also determined. For each macro and enhancement pair the above weight was divided by the cost of the enhancement, then multiplied by the percent speedup that the enhancement would bring to the macro. For each enhancement these values were summed across all of the macros and then divided by the number of macros to which the enhancement applied. The result was a figure of merit that contrasts cost with speed. Table 15 shows the macros, their usage counts, the weight factor, and the number of algorithms in which the macro appeared. Table 16 shows the list of enhancements

Rank	Macro	Number Algorithms	Count	Weight
1	Memory to memory neighbor	6	275584	6
2	Exact match	2	264192	6
3	Find max or min	1	262144	6
4	Select first	1	262144	6
5	Inequality match	3	5376	4
6	Shift field in cell	1	2048	4
7	Load or dump image	1	512	3
8	Count responders	2	280	3
9	Add or subtract constant	2	160	3
10	Load X-Y grid	4	4	1
11	Multiply constant	1	4	1
12	Add or subtract fields	1	2	1
13	Multiply fields	1	2	1
14	Divide constant	0	0	1
15	Divide fields	0	0	1
16	Life	0	0	1

Algorithms

3 x 3 Gaussian filter
3 x 3 Sobel
256 Bucket histogram
Rotate model and extract front view
Region growing and labelling
Center of mass
Geocorrection
Sort array

Table 15

Macro Usage Statistics

Rank	Enhancement	Cost Factor	Number Algorithms	Speed Merit
1	Fast Count	2	1	141.0
2	Build chips as four quadrants of 16 cells, double number of communications lines	1	5	132.9
3	Bottom row adder tree	1	1	76.2
4	Dual port memory	4	1	75.0
5	Multiple controllers	4	1	71.6
6	Add true XOR to ALU	1	3	70.8
7	ALU can take A and Y, X and M, C and Y, C and M as inputs	1	9	64.4
8	Full communication between chips and cell level select control	4	5	52.6
9	A can take ALU result while X takes one of the operands	2	1	45.8
10	ALU can take M, any register or C as either of its inputs	2	9	32.3
11	Output to neighbors can be either A, X or M	1	2	32.2
12	A and X can both take ALU result at same time	2	2	29.9
13	Read-modify-write memory	1	8	27.8
14	ALU can take M, any register, any neighbor or C as either of its inputs	4	9	18.2
15	Any register or M can be output to neighbors	2	2	16.2
16	Generalize ignore activity	1	1	13.2
17	Two address memory operations	4	3	6.8
18	Generalize Z transfers	1	7	3.1
19	Selectively complement ALU inputs	2	2	3.0
20	Exchange A and B instruction	2	0	0.0
21	Long distance communications	4	0	0.0
22	Vector broadcast	4	0	0.0
	128 bit memory	1		
	256 bit memory	3		
	Eliminate cross register shifts	0		
	Change chips to CMOS	3		
	Reverse memory select sense	0		

Table 16

Enhancements Ranked by Speed Merit

with their estimated cost factors, number of algorithms to which they apply, and the resulting figure of merit.

It is important to note in this analysis that those enhancements which have no effect on speed are unfairly represented by the figure of merit. For example, it is highly desirable that the memory size be increased, even though there is no increase in speed resulting from this. These enhancements have been listed separately at the bottom of the table with only their estimated cost factors shown.

Despite this analysis, the constraints placed on each of the three new designs essentially dictated a choice based on the cost factor alone. The conservative design uses all of the enhancements that have a cost factor of zero or one. The intermediate design was restricted to those of cost two and below. The figure of merit was used only to exclude the last of the cost two enhancements from the design. Only with the advanced design did the figure of merit play an important part. The advanced design uses all of the cost zero through three enhancements (except Exchange A and B) and the third and fourth ranked cost four enhancements. (The first of the cost four enhancements has as its purpose the increase of image load and dump speed. This is adequately taken care of by the third ranked enhancement -- full communications and select control. The latter is much

more generally applicable and thus was chosen first despite its slightly lower rank. The second ranked of the cost four enhancements, multiple local controllers, is not included in the advanced design because it adds considerable cost and complexity to the overall design, and since it has no effect on the chip design, the decision to exclude this enhancement can be reversed at a later date without penalty.) All of the lower ranked cost four enhancements have been excluded from the advanced design. The three designs are presented in the following sections.

Conservative Second Design

Design constraints. This design conforms to the original set of constraints for the first design while attempting to improve processing throughput as much as possible. To review these constraints, the CAAPP must consist of no more than 100 circuit boards, each board should have a maximum of 100 off-board connections, the VLSI chips are restricted to 40,000 devices, have a pin out of no more than 40 pins and must dissipate less than two watts. This essentially restricts the choice of design enhancements to those with an estimated cost factor of zero or one. This is the most that can be included without a relaxation of the

constraints. Basically what this new design does is to tune the original design and take advantage of the fact that there was some room to spare within the constraints.

Overview of the design. The same overall design for the CAAPP has been retained in this new version. The machine is still intended to be a co-processor with a mini-mainframe acting as a host. The only change to the controller is the instruction set which it broadcasts to the processing array. The array itself is still the same size and tentatively arranged as an eight by eight array of circuit boards, each with 64 of the special purpose chips. The same signal distribution and response count circuitry is used except that the final summing of the response count is done with an adder tree to speed this up slightly.

Chip and processing element design features. The major changes to the CAAPP are in the special purpose VLSI circuits. Instead of having a single eight by eight array of processors arranged as two columns of 32 processing elements (as in the first design), four quads of 16 elements are used. This has several benefits. Firstly it makes it much easier to implement the high-yield/fault-tolerant version discussed in the preceding section. Secondly it reduces the space required by the neighbor communications network. Instead of trying to connect all of the processors

with both networks, only 16 at a time are connected by the on-chip style network (here renamed the on-quad network). Thus only four wires are needed to connect the quads on a chip. Thirdly, the time is reduced for an off-chip (off-quad) transfer by 50 percent to 0.4 microseconds per bit. The new floor plan for the chip is shown in figure 52. Figure 53 shows the layout of the communication network on the chip.

Most of the changes on the chip are in the processing elements themselves. First of all, the memory has been enlarged to 128 bits per cell. This will be done by shrinking the cell size, using a sense amplifier circuit to read the output (thereby allowing a reduction in the driving capability of the memory bits). Use of a two metal layer process will allow considerable reduction in cell size as well. If necessary a dynamic memory circuit may be employed to further reduce cell size and device count.

Figure 54 shows the structure of the new processing element. The most important features to note are that the ALU has an expanded set of inputs, the output to the neighbors may select from three different sources, the Z register now has full transfer capability for any other destination as well as a write inhibit input to allow exclusive OR operations. Although it is not visible in the

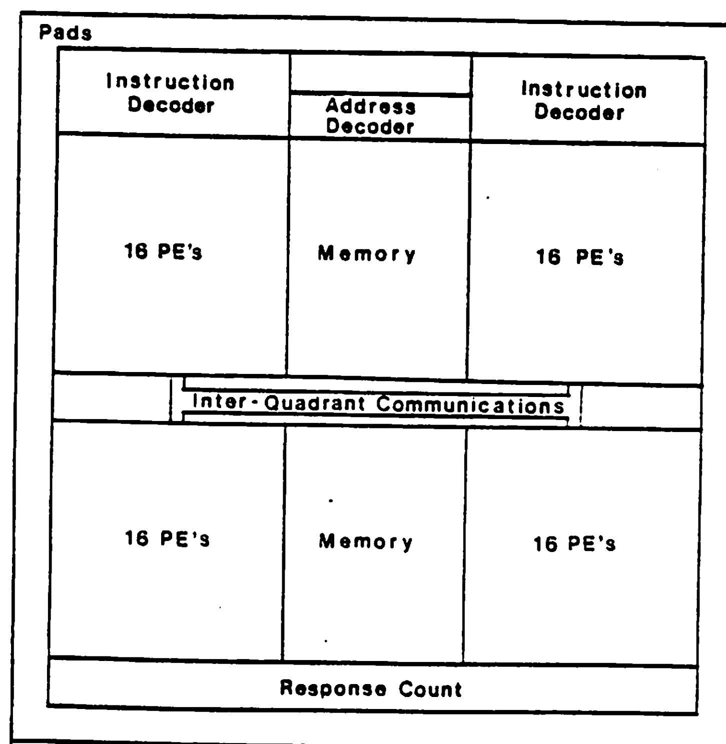


Figure 52

Floorplan of Conservative Second Design Chip

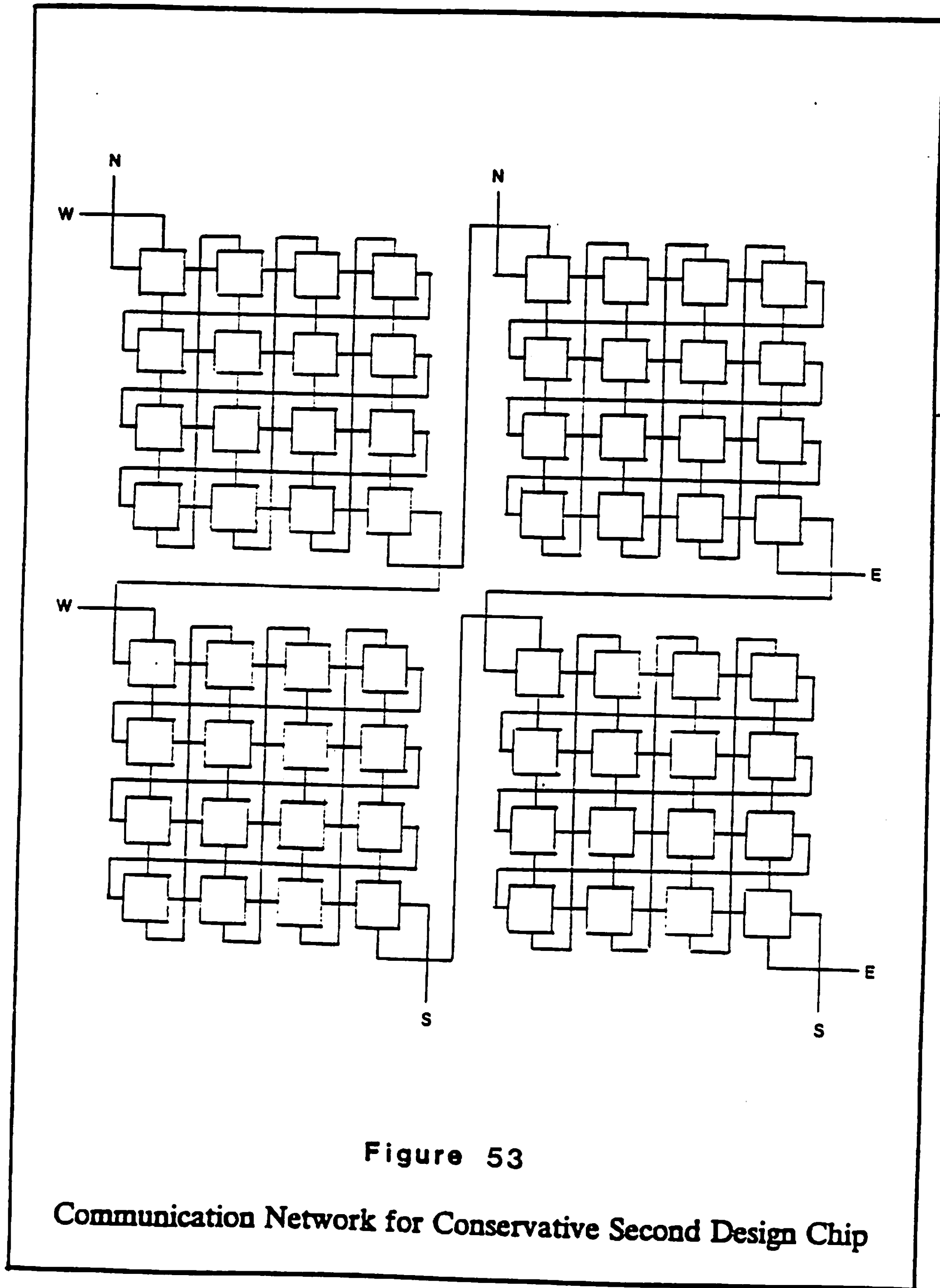


Figure 53

Communication Network for Conservative Second Design Chip

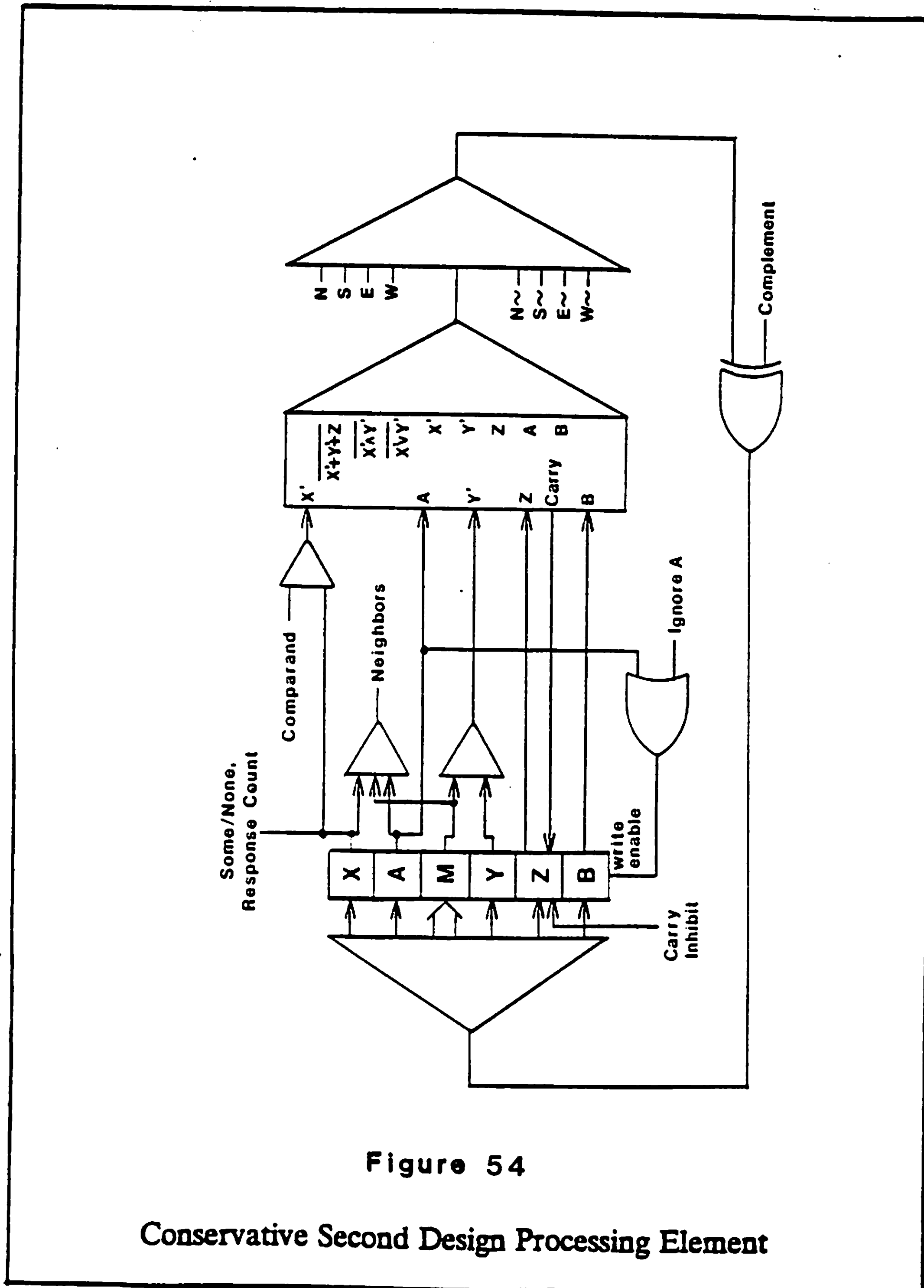


Figure 54

Conservative Second Design Processing Element

figure a read-modify-write capability has been added for operations involving memory as both a source and destination.

The instruction set for the processing element has been somewhat enlarged but simplified. The useless instructions noted in the preceding section have been eliminated and more useful instructions added. The goal here was to increase the overall symmetry of the instruction set so that there would be fewer special cases. In essence the instruction set now allows any of the ALU outputs, register values, memory bits, or the comparand to be transferred into any register or memory location. Each of these transfers may take place under the control of, or independent of the activity mask with the transferred value being left in true form or complemented. The only exceptions to this are the memory operations which, for the most part, can only take place under control of the activity mask. Transfers to and from memory involving the A and B registers may also ignore activity. It is further possible to store the broadcast bit into memory while ignoring activity. The reason for excepting memory from the general pattern is that this provides a few operation codes that can be used for the neighbor transfer operations by making the address field of the instruction do double duty. This keeps the instruction

set size to a minimum, with a corresponding reduction in the number of pins required to carry the instruction lines into the chip.

Neighbor communications permit the value of the X register to be transferred to neighboring X or Y registers via either the on-quad or the off-quad neighbor networks. The activity bit (A) may be transferred via the on and off quad networks only to the A registers in neighboring cells. The zig-zag comparand shifts into X are retained with the addition of A as a destination, to allow external insertion of activity and response patterns onto the quads; such as in the Load X-Y Coordinate Grid algorithm (each quad is individually selectable in this design). Provision is also made for a read-modify-write transfer from a memory bit to the corresponding memory bit in a neighboring cell via the off-quad network. Read-modify-write to a neighboring cell can only be done ignoring the activity pattern and with true logic sense. All of the other neighbor transfers have the full set of optional modes of operation.

Table 17 presents the instruction set for the processing elements. The instruction set provides 60 memory based operations, 127 non-memory transfer instructions, each with the full four modes of operation, and four special operations. These are the response count operations CRCR,

Source	X	A	B	Y	Z	M
	Destination					
X	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
A	!:+-	!:+-	!:+-	!:+-	!:+-	! +-
B	!:+-	!:+-	!:+-	!:+-	!:+-	!:+-
Y	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
Z	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
C	!:+-	!:+-	!:+-	!:+-	!:+-	!:+-
M	:+-	!:+-	!:+-	:+-	:+-	:+-
$\overline{X + Y + Z}$	!:+-	!:+-	!:+-	!:+-	$X := \overline{X \oplus Y}$	:+-
$\overline{C + Y + Z}$	!:+-	!:+-	!:+-	!:+-	$X := \overline{C \oplus Y}$	:+-
$\overline{X + M + Z}$	!:+-	!:+-	!:+-	!:+-	$X := \overline{X \oplus M}$	:+-
$\overline{C + M + Z}$	!:+-	!:+-	!:+-	!:+-	$X := \overline{C \oplus M}$	:+-
$\overline{X \wedge Y}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{C \wedge Y}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{X \wedge M}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{C \wedge M}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{X \vee Y}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{X \vee Y}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{X \vee M}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
$\overline{C \vee M}$	!:+-	!:+-	!:+-	!:+-	!:+-	:+-
X(NSEW)	!:+-			!:+-		
X(~NSEW)	!:+-			!:+-		
A(NSEW)		!:+-				
A(~NSEW)		!:+-				
C(~NSEW)	!:+-	!:+-				
M(~NSEW)						! +

! = Ignore activity
+ = True logic
: = Activity controlled
- = Complemented logic

Table 17

Conservative Second Design Instruction Set

SCRR, SCRC, and PANS. For the purpose of counting responders an extra connection is provided that links the quads into a single long shift register via the off-quad network. There are also four special instructions in the set which do not follow the basic source-destination transfer arrangement. Where there should be operations for Z:=X+Y+Z, Z:=C+Y+Z, Z:=X+M+Z and Z:=C+M+Z, a different set has been provided. The eliminated operations would automatically force Z to accept the new carry and thus there is a conflict as to whether Z should accept the result or the carry. There is also the problem that it is desirable to provide an exclusive OR operation for use in the comparison algorithms. One possibility would be to have Z take the result, ignoring the carry. The problem is that Z would still have to be cleared before each bit of the operand is tested. Furthermore, the comparison operations all require that the result be put in the response bit (X). Thus, the best choice is to have this set of operations inhibit writing of the carry into Z and to send the result to X.

Design advantages and limitations. The advantages of this new design are that it reduces the ALU input bottleneck and the neighbor communication bottleneck, permits read-modify-write operations on memory, doubles the speed of neighbor transfers, and streamlines and simplifies the

instruction set. The limitations that are still present are a response count that is only a little faster, there is no operation for swapping A and B, the array must still sit idle during response counts and bulk loads and dumps of data sets, communication between cells is still four times slower than the optimum, the controller still does not have direct selection control for the individual cells, long distance communications are still time consuming and the controller can only read one element out of the array at a time.

Implementation analysis. In terms of complexity of implementation, the new processing element is only slightly more complex than the original design. The extra selector circuits that are required for the ALU and neighbor network inputs are offset by the removal of their counterparts from to post-ALU selector. Thus the space requirement and component count are only slightly greater. There is actually a reduction in space from the original design in the communications network region. The biggest increase in complexity is in the expanded memory. This quadruples the number of devices in that area. Switching to a dynamic memory circuit could divide that number by a factor of three, thus giving an overall increase in memory devices of 33 percent. Further shrinkage of the cell size (by use of two metal layers, for example) could bring the space

requirements down to only 10 to 15 percent more than the original design allowed. Thus the total space for the chip should remain nearly the same. The total device count should still be well under 40,000 and the power dissipation should only moderately increase (recall that there will be a considerable reduction in the power dissipated by the bit select drivers once the logic sense is inverted).

The most obvious increase in complexity is going to be the ten extra pins required to carry all of the communication and control signals. This brings the total pin count (see table 18 for listing) to 31. Although this is still under the limit of 40 pins, it requires a considerably larger package than the original chip design. Furthermore this leads to a violation of the constraints in another area: There will have to be 115 wires connected to each of the circuit boards in the array. Actually this is not such a major violation that the design should be abandoned. Of the 115 wires, 51 can be connected to a back-plane bus structure. The remaining 64 wires are the communication lines that run between the boards. These are divided into four groups (one for each neighboring board) of 16 wires. Most likely these signals will be run in ribbon cable. A board with 51 backplane connections and 64 ribbon cable connections is certainly within the capabilities of standard

Chip Pinout

Op Code	6
Address	7
Communications	8
Quadrant Enable	4
Comparand In	1
Some/None Out	1
Clock	2
Power/Ground	2
Total	31

Circuit Board Connections

North Communications	16
South Communications	16
East Communications	16
West Communications	16
Column Select	16
Row Select	16
Op Code	6
Address	7
Comparand In	1
Some/None	1
Clock	2
Power/Ground	2
Total	115

Table 18

Pin List for Conservative Chip and Circuit Board I/O Lines

technology to implement reliably. The number of circuit boards in the array is then kept the same, although their size will most likely increase due to the increased size of the IC packages.

Comparison to the previous design. To provide a quick comparison of the speed of this design to the original design, a recoding of the Sobel algorithm with the new instruction set gives a time of 6.3*LF + 3.8 microseconds or 54.2 microseconds for an eight bit pixel. This is nearly double the speed of the old design, which took 103.2 microseconds for the same operation.

Intermediate Second Design

Design constraints. This design slightly relaxes both the space and pinout constraints on the chip, and the number of connections between boards, while increasing the symmetry in the instruction set and further expanding the capabilities of the ALU. Specifically the number of pins is allowed to grow to 64, the space on the chip may be increased by 15 to 20 percent (or correspondingly the feature size may be reduced to allow a comparable number of devices to be fit into the same area). The connections between boards may be expanded to 100 on a backplane bus

plus 64 running between boards in ribbon cable or a comparable technology. The power dissipation per chip is still fixed at two watts to permit forced air cooling, and the number of boards is still limited to 100. This level of relaxation in the constraints is mostly based upon improvements in technology that have taken place in the two and a half years following the initial design. Smaller 64 pin packages are now readily available, and feature sizes have indeed been reduced. The relaxation in circuit board I/O lines recognizes the somewhat arbitrary limit that was placed on the first design and has been adjusted to correspond to standard connector sizes. Thus, the constraints will still lead to a very conservative design that should be readily buildable with high reliability. The result is that almost all of the enhancements that were estimated at cost factor two and below can be included in this design. The only one that is excluded is the Exchange A and B instruction, simply because it adds considerable asymmetry to the instruction set.

Overview of the design. Once again the same basic configuration has been retained for the overall design of the CAAPP. It is still planned as an array of 64 circuit boards, each with 64 special purpose chips, giving an array of 512 by 512 processing elements, and a single controller

connected to a mini-mainframe host. The major changes are again mostly within the VLSI circuits, although the polled-by-card response count circuit outlined earlier is included to increase the speed of that operation by about an order of magnitude to 2.5 microseconds per count with successive counts available each following 1.6 microseconds. This will allow a 256 bucket histogram to be computed in 410.5 microseconds or roughly one eightieth of a video frame time. The new response count circuit will be described in detail after the new processing element architecture is presented.

Chip and processing element design features. As in the conservative design, the memory per cell is expanded to 128 bits. The chip is similarly built as four quads of 16 processing elements, each with on and off quad communication networks. This leads to the same off-quad transfer time of 0.4 microseconds per bit. The overall processing element structure of data sources feeding an ALU followed by a function selector that returns a result to one of the storage elements in the cell, has been retained. This is about as far as the similarity between the two designs goes. As to the original design, there is only a vague resemblance between the old processing elements and these new ones.

The biggest change in the processing element structure is in the circuitry between the data sources and the ALU. Each of the two inputs to the ALU is preceded by a selector circuit that allows any of the data sources to be read in either true or complement form. The result is that any pairwise combination of X, -X, Y, -Y, Z, -Z, B, -B, A, -A, comparand, complemented comparand, a memory bit, and a complemented memory bit can be operated on by the ALU. The inputs to the ALU are hence renamed I and J as they no longer are tied to any particular data source. The I input to the ALU has special significance: It is the source for data to be transmitted to the neighbors. Sending data from I to the neighbors allows direct access to any storage element in any of the four neighboring cells. The read-modify-write cycle for memory to memory operations is retained from the conservative design. The J input also has a special feature in that it can be fed directly back to X. Provision has been made so that X may be a destination for J at the same time that A is a destination for the ALU result. The purpose of this dual destination mechanism is to permit the operations (A:=-(M+C); X:=M and A:=-(M+C); X:=C). Examination of the inequality test algorithms in chapter five will reveal that this combination permits such comparisons to be performed at a rate of one instruction time per bit, which makes inequality tests just as fast as

the exact match operation.

Another feature is a carry inhibited form of the addition operation (XOR) that can have any destination. Inhibiting the carry is done by inhibiting the writing of the carry output of the ALU into Z. Tentatively, the design allows the ALU result to then be stored into Z. It may turn out that the circuitry for this requires too much space, in which case the Z:=I+J operations may be thrown out and replaced by no-ops.

The response count circuitry on the chip has been converted to an adder tree design with independent control of the latching and output of the count. Independent control allows the array to execute instructions at the same time that a latched response set is being counted. The response count register can latch a new value after 0.8 microseconds, however, it must wait another 0.8 microseconds before it can transmit the value to the circuitry outside of the chip. This gives a time of 1.6 microseconds during which the array can prepare a new set of responders without making the controller wait to siphon off the data. The count time corresponds to the time required to compare a broadcast value against a 12 bit field in cell memory. Thus, for an eight bit comparison, the array will have to wait four instruction cycles for the controller to finish

counting. For a 16 bit comparison the controller will have to wait four cycles for the array to finish computing. This would seem to be a reasonable tradeoff. The architecture of the processing element is presented in figure 55.

The instruction set for this design is considerably enlarged over the conservative and original designs. The total instruction set results in 5,432,448 possible control combinations. Taking the logarithm (base two) of this number gives a minimum instruction plus address size of 23 bits. Unfortunately, packing all of the instructions into a field of this size will result in an extremely complicated encoding scheme. This has two drawbacks. One is that this would make programming the CAAPP very difficult -- if only for the assembler and compiler writer. The other is that the instruction decoder on the chip would have to increase correspondingly in complexity. In the original design, the encoding was simple enough that it was possible to use what was essentially the NOR plane of a Programmed Logic Array (PLA). With a more complex encoding a full PLA would probably be required and would be so large that there would barely be any room left on the chip for the processing elements. The solution to this problem is to "waste" a few more bits by enlarging the size of the instruction to allow a simpler encoding. By expanding the combined instruction

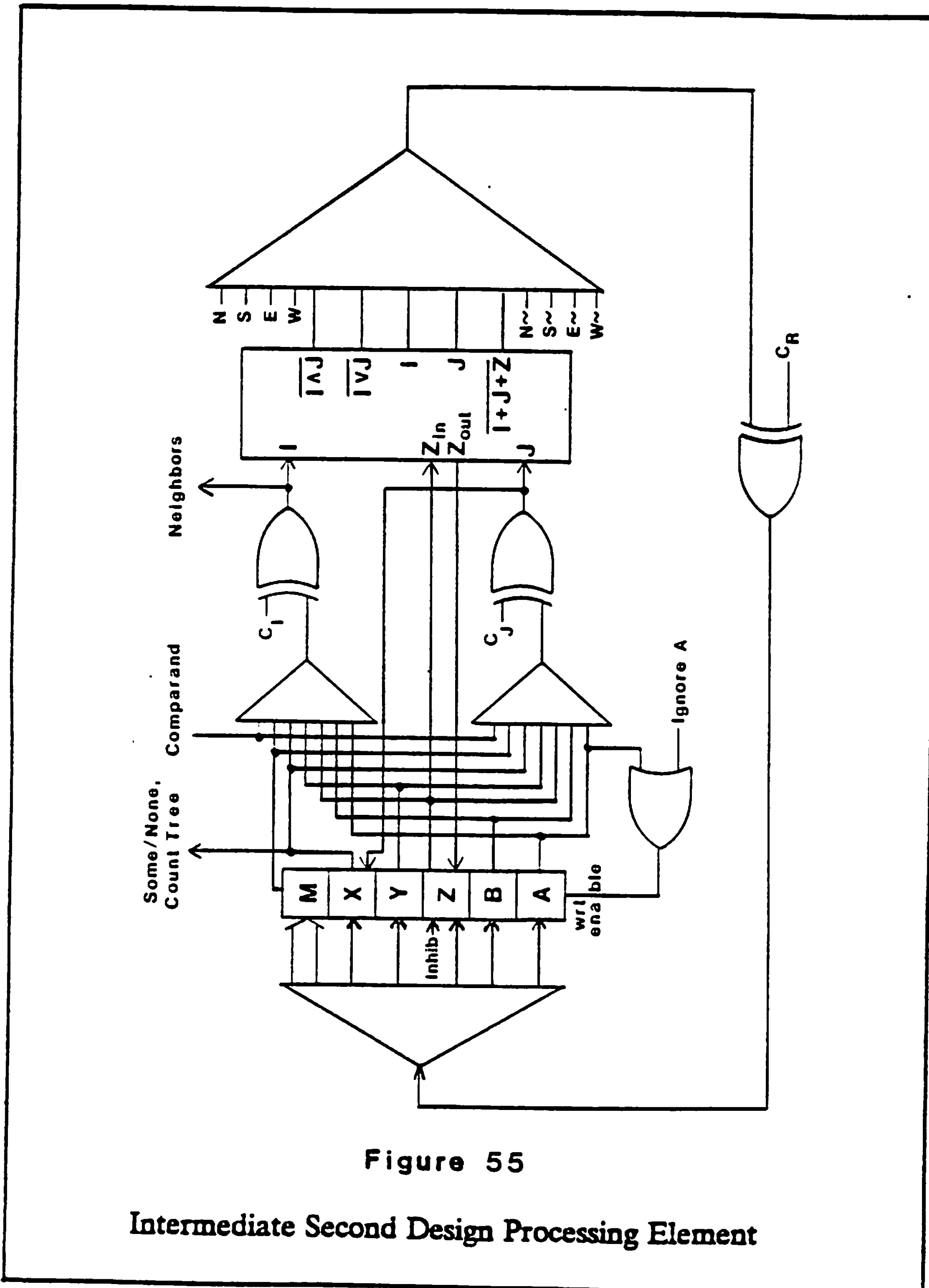


Figure 55

Intermediate Second Design Processing Element

and address space to 25 bits (an addition of two beyond the minimum required) the encoding presented in figure 56 becomes possible. (Note that only the two comparand shifts that were found to be useful in the statistics have been implemented.) This is very simple both to build a decoder for and in which to write instructions.

The new response count mechanism. The new response count scheme uses the idea of adder trees within the chips and on the circuit boards to develop a board level response count. Rather than use the Pipelined add North to South (PANS) scheme, however, another method is employed to develop the final count. The response count register of each board is connected via tri-state bus drivers to a backplane bus. Unlike the other bus lines on the backplane, however, this bus is broken at the end of each row of circuit boards. Thus, there are eight such busses, each with eight cards. A set of three select lines is provided to determine which column of cards gets to talk on the busses at any one time. The busses are connected to the controller through a full selector and an adder tree which sums all of the selected rows to a single value. The value is then input to a high speed adder which has its own output as its second input. This is diagrammed in figure 57.

A response count then proceeds as follows: The response

I	
Ø	Activity Controlled
1	Ignore Activity

C (i,j,r)	
Ø	True
1	Complement

M/R	
Ø	Register Destination
1	Memory Destination

Source (i,j)	
Ø	None
1	M
2	X
3	Y
4	Z
5	B
6	A
7	C

Destination	
Ø	None
1	A,X
2	X
3	Y
4	Z
5	B
6	A
7	A, X ← J

Function	
Ø	CW~
1	I
2	J
3	$\overline{I \wedge J}$
4	$\overline{I \vee J}$
5	$\overline{I \oplus J}$
6	$\overline{I + J + Z}$
7	CN~
8	N
9	S
1Ø	E
11	W
12	N~
13	S~
14	E~
15	W~

Figure 56

Intermediate Second Design Instruction Set

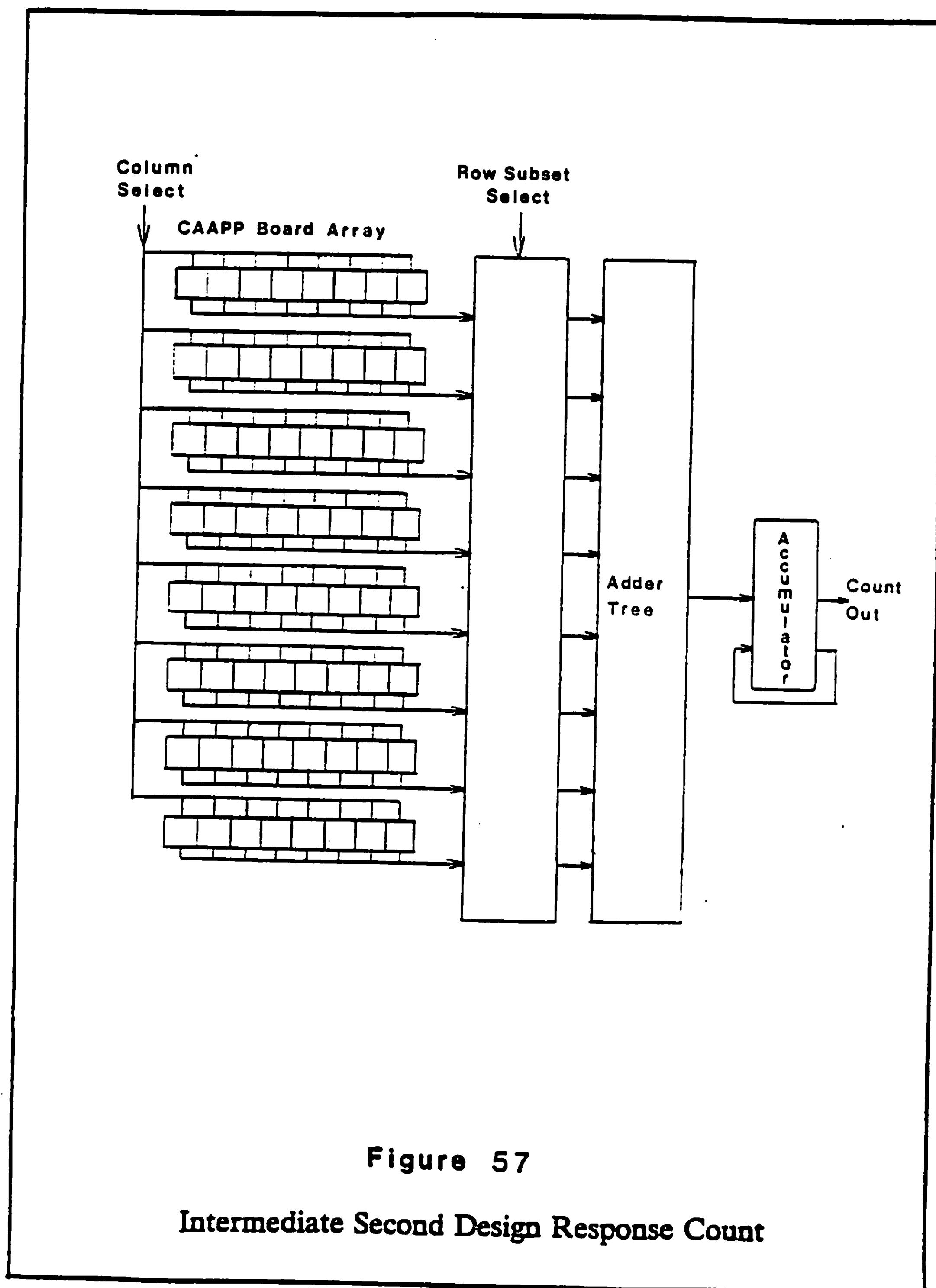


Figure 57

Intermediate Second Design Response Count

count register on each of the chips latches the count of the current response pattern (Ø.1 us) and then shifts this bit serially to an external register (Ø.7 us). Note that as soon as the count is latched, the array can go back to work -- shifting control is independent of array processing. The output of the external register feeds another adder tree which is connected to the 13 bit board response count register. After another instruction time (Ø.1 us) this value is latched. At this point the response count register on the chip can begin feeding its next count out to the external register. Meanwhile the controller begins polling the columns of board response count registers by selecting them one after the other to be output onto the eight busses. Each column of response counts passes through the selector, then the adder tree and finally to the summing register (Ø.1 us for the select, followed by Ø.1 us to develop the sum) at which point it is accumulated into the previous sum.

The total count time is thus 2.5 microseconds of which 1.6 microseconds is required for the polling operation. As this can be carried on independently of the rest of the operation, successive counts can be obtained at the maximum polling rate. Note that the combination of being able to select any set of board rows for input to the final adder

and being able to select any sequence of columns for reading allows the controller to count any subset of the boards. This is useful, for example, when the array contains four lower resolution images which are being compared. It is then possible to extract a count for each image independently of the others. The subcounts can be formed for any size of sub-array, ranging from a single board (64 by 64) up to the full 512 by 512 array.

Design advantages and limitations. Let us summarize the benefits of this intermediate second design: It permits faster response count and comparison operations than either the original or conservative designs. It also greatly expands the data source and destination combinations for both the ALU and the neighbors and provides for complemented data sources as well as the destination. It also adds the capability to write into one or two registers simultaneously with a memory write. It retains a number of the benefits of the conservative design that are not present in the original design. These include the faster communications between chips, the carry inhibit addition (XOR), the expanded memory and a more symmetric instruction set.

The limitations of the design are that neighbor communications are still not at maximum speed, the response count is not up to maximum speed, long distance

communications are still time consuming, the controller can still only read one value from the array at a time, there is still no swap A and B instruction (although this has been reduced to requiring only two instruction cycles), the array must still sit idle during bulk loads and stores of data sets, and the controller still does not have direct control of cell selection.

Implementation analysis. In terms of complexity of implementation, the silicon space required for the new processing elements will increase a considerable amount. The two new selectors are considerably larger than those proposed for the conservative design. There are also two more exclusive OR gates required to complement the ALU inputs. The increase in memory size will be the same as for the conservative design. The instruction decoder for the chip will be both taller and wider than in either of the previous designs. This will be necessary to accomodate the 25 inputs and the additional control outputs for the selectors and XOR gates. Overall, however, the required real estate should not exceed the budgetted increase.

If anything drives the circuit space beyond the maximum in this design, the wiring pad area will be the thing to do so. The total number of pins required for the chip is now up to 45 (listed in table 19). This is still well below the

Instruction	25
Communications	8
Quadrant Enable	4
Comparand In	1
Some/None Out	1
Read/Hold Count	1
Shift/Wait Count	1
Clock	2
Power/Ground	2
Total	45

Table 19

Intermediate Chip Pin List

maximum of 64 and is in fact not much more than the original limit. The number of circuit board connections is still within the specified limit with a total of 146 wires (listed in table 20). Of these, 64 are in ribbon cable (providing communication between the boards) and the remaining 82 can be run on the back plane bus. The number of boards still conforms to the limit. The total number of devices and power dissipated per chip are a bit difficult to compute, but neither should be larger than the corresponding figures for the conservative design by any great percentage. Most of the additional devices will be in the selectors (which are very simple) and in the instruction decoder.

Since there was a substantial safety margin for the conservative design, presumably this design will not violate the constraints. Under this assumption, the intermediate design should not be particularly hard to construct as long as smaller packages (such as quad inline) are used to conserve circuit board space. The only requirements are that the design team be equipped with state of the art design tools and have access to current fabrication technology.

Comparison to the previous designs. Once again, for comparison purposes, a recoding of the Sobel operation in the new instruction set gives a timing of 4.9*LF + 2.8

North Communications	16
South Communications	16
East Communications	16
West Communications	16
Column Select	16
Row Select	16
Instruction	25
Comparand In	1
Some/None Out	1
Read/Hold Count	1
Shift/Wait Count	1
Board Count Latch	1
Board Count	13
Board Count Select	3
Clock	2
Power/Ground	2
Total	146

Table 20

Intermediate Second Design Circuit Board Connections

microseconds. For eight bit pixels this gives a time of 42 microseconds. This is a 59 percent reduction from the original design and 22.5 percent faster than the conservative design. The increase in speed over the conservative design is mainly due to the ability of the intermediate design to store into both memory and a register at one time. (42 microseconds is 1/785 of a video frame time) Of course the most noticable speed increase for this machine is seen in the histogram algorithm. This is now ten times as fast as was previously possible.

Advanced Second Design

Design constraints. For this design, the constraints have been considerably relaxed. This will be the optimum performance design. Despite the relaxation in design constraints, this machine should be buildable with current technology. It will, however, require the very best tools, technology and people to make it work reliably. Given another three to five years of improved technology, this machine will probably be relatively easy to build. Specifically, the pin out constraint has been raised to 100 pins, the maximum device count on a chip is 100,000, the power dissipation limit is kept at two watts (coolable by

forced air), the circuit board connection limit is raised to 256 ribbon cable type connections and a 200 contact back plane connector, the number of circuit boards allowed is still 100. The result is that all of the cost factor three and below enhancements can be included, plus two of the cost factor four enhancements. The two most costly enhancements that were chosen for inclusion are full width communications between chips with direct select control for the individual cells, and making the neighbors directly selectable as input sources to the ALU. The latter is really only costly by itself. Once full communications have been provided between chips, the additional cost is small to make the neighbors go directly to the ALU.

Overview of the design. The advanced design retains most of the features of the intermediate design while adding full width communications between chips to speed up neighbor transfers, full select control of the cells, expansion of the ALU inputs to take data directly from the neighbors, and double the memory per cell.

At the top level, the controller still is connected to a host processor and directs operation of the array. The same response count circuitry is used as in the intermediate design. The side and bottom edge cards will also have to be expanded to provide routing circuitry for the increased

number of communications lines and select control lines at each edge.

Chip and processing element design features. In the chips, the biggest changes are the increased memory and the elimination of the zig-zag neighbor connections. Although not part of the computational circuitry, another very noticable change upon examination of the chip will be the large number of wiring pads. Because of the increased component count, the chip will have to be constructed with CMOS technology to keep power dissipation within the specified limit.

The biggest change in the processing elements is that the neighbors are now directly accessible as inputs to the ALU. This is made possible by the full width communications between chips. In previous designs the multiplexing at chip edges meant that a cell's neighbor was always at least four jumps away. In the advanced design, neighbors are all directly connected. This necessitates enlarging the ALU input selectors even more than for the intermediate design, and the addition of yet another selector to determine the data source that is to be transmitted to the neighbors. Fortunately, the selector that follows the ALU is correspondingly reduced in size so that the net gain in selector devices is about six pass transistors and a buffer

(in the neighbor data source selector). This is diagrammed in figure 58.

The advanced processing element retains the secondary return path to X that was first presented in the intermediate design. It also retains the same carry inhibit addition operation (XOR), the ability to complement the ALU inputs, the ability to write into a register and memory at the same time and the adder tree form of response count with independent control of the count register. The output of the register is fully parallel, however, instead of bit serial. The board response count register is replaced by a circular queue of 32 registers in a special purpose chip. This is done to allow the array to quickly dump a series of response patterns (as in summing a field, such as with the Center of Mass algorithm) without having to wait for the 1.6 microseconds it takes the controller to form the final count. The array can then go on to some other computation while the controller collects the series of counts.

The instruction set for the advanced design is considerably larger than for the preceding designs. This is due to the increased number of data sources. Thus there are a total of 75,989,760 possible instruction and address combinations. Taking the logarithm (base two) of this shows that the minimum number of bits required to represent these

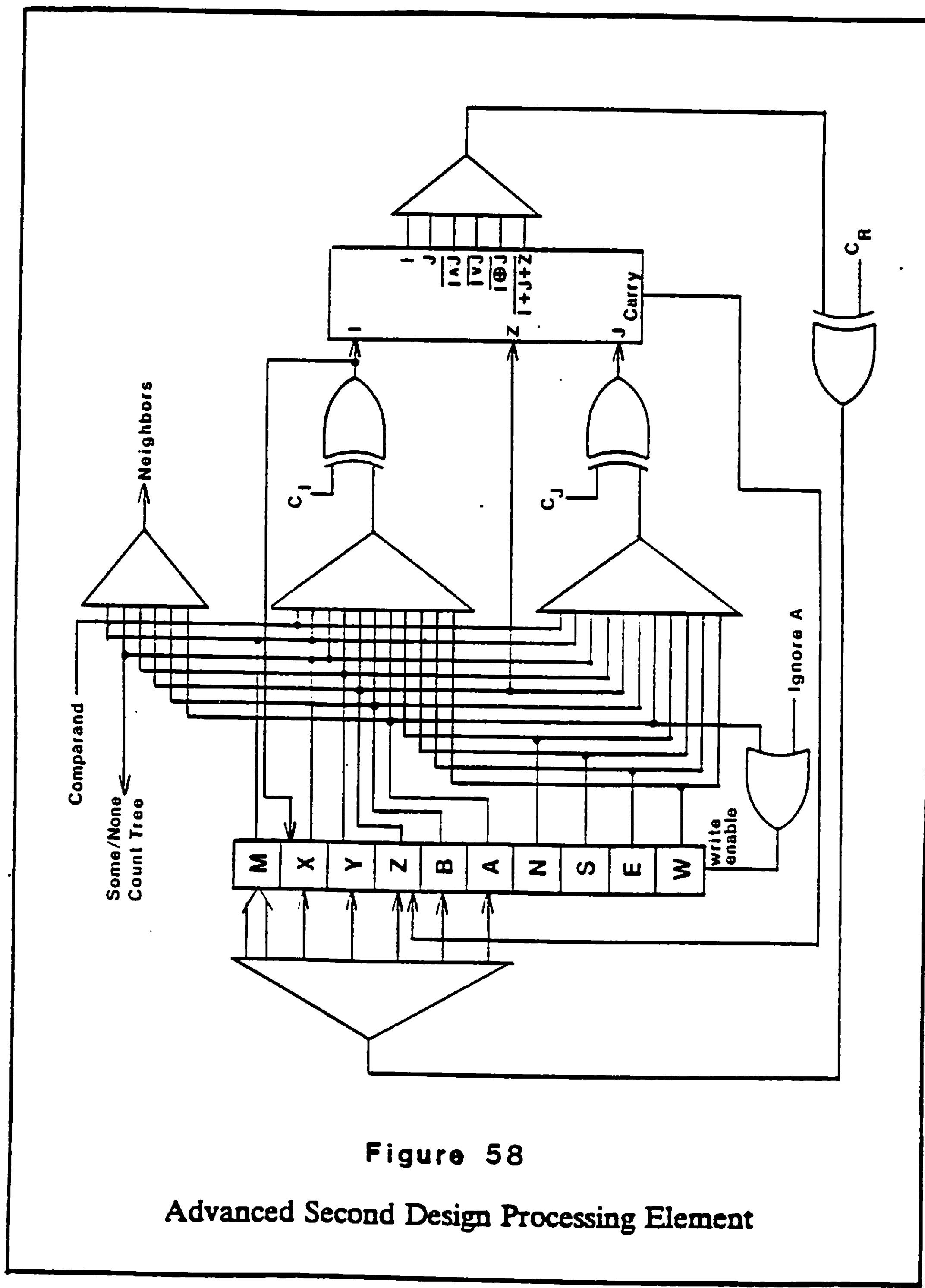


Figure 58

Advanced Second Design Processing Element

combinations is 27. As with the intermediate design, an encoding that packs everything into this number of bits will be so complex that the on chip instruction decoder will leave no room for the processing elements. In order to simplify the encoding, three extra bits are added to form a 30 bit instruction. This is shown in figure 59.

Design advantages and limitations. The benefits that are gained by this design are fast communications between neighbors, the ability to take data from two neighbors at once (thereby reducing the number of instructions required to poll a local neighborhood to only three), faster load and dump time for data planes (a 16 bit plane can now be shifted in from the edge in one fortieth of a frame time), further decoupling of the response count from the array operations, double the amount of in cell memory, the controller now has full selection control of cells.

The remaining limitations are that the controller can still only read out one datum from the array at a time, there is no provision for long distance communications, the swap A and B operation still takes two instruction times, and the array must still sit idle during bulk loads and stores (although the time for these is now relatively short).

I	
Ø	Activity
1	Ignore

C[i, j, r]	
Ø	True
1	Complement

M	
Ø	None
1	Address

Source[i, j]	
Ø	None
1	M
2	X
3	Y
4	Z
5	B
6	A
7	C
8	N
9	S
10	E
11	W
12	None
13	None
14	None
15	None

Src_n	
Ø	None
1	M
2	X
3	Y
4	Z
5	B
6	A
7	C

Dest	
Ø	None
1	A, X
2	X
3	Y
4	Z
5	B
6	A
7	A, X ← I

Fn	
Ø	None
1	I
2	J
3	$\overline{I \wedge J}$
4	$\overline{I \vee J}$
5	$\overline{I \oplus J}$
6	$\overline{I + J + Z}$
7	None

Figure 59

Advanced Second Design Instruction Set

Implementation analysis. In terms of complexity of implementation, this machine presents a real challenge. The chips themselves contain, as a rough estimate, about 70,000 devices. This is mostly due to the increased memory size. Perhaps the biggest problem, however, will be where to put the 92 wiring pads required by chip. These are listed in table 21. This leads to the number of connections on each circuit board which is itemized in table 22. Of these 338 lines, 256 are run directly between boards while another 182 can be run on the back plane bus. Nonetheless, this leads to a total of 21,632 wiring connections for the array alone. Obviously, construction of this machine will require very reliable connector technology, the best VLSI fabrication technology and considerable engineering expertise. Although buildable now, the design will be more cost effective if built after a few more years. As numerous projects can testify, pushing the limits of both architecture and technology at the same time is a sure formula for trouble.

Comparison to the previous designs. As a final comparison to the other three designs, the Sobel operation has once again been recoded for the new instruction set. Using the ability to poll two neighbors at once gives an total time of 1.0*LF + 0.8 microseconds or 8.8 microseconds

Instruction	30
Communications	32
Cell Enable	16
Read/Hold Count	1
Count Out	7
Comparand In	1
Some/None Out	1
Clock	2
Power/Ground	2
Total	92

Table 21

Advanced Chip Pin List

North Communications	64
South Communications	64
East Communications	64
West Communications	64
Row Select	64
Column Select	64
Instruction	30
Read/Hold Count	1
Latch Board Count	1
Count Out	13
Board Select	3
Comparand In	1
Some/None Out	1
Clock	2
Power/Ground	2
Total	338

Table 22

Advanced Second Design Circuit Board Connections

for eight bit pixels. This is nearly 12 times faster than the original design, six times faster than the conservative design and a little less than five times faster than the intermediate design. It is interesting to note that the time required for this machine to apply the Sobel edge operator to a 512 by 512 image of eight bit pixels is about the same as is required by many serial processors to perform a single floating point multiply. This could perhaps become the defining mark of a true image processing architecture: the ability to perform a meaningful operation on a complete image in the same time that a serial machine takes to perform a single floating point operation. These might then be called "image operand class" architectures.

CHAPTER VII

CONCLUSIONS

The first and most important conclusion to be drawn from this research is that it is possible to build a pixel-per-element class associative processing machine with technology that is currently available and mature. The fact that mature technology can be used is especially important because its use will reduce the construction problems of the machine to those that are associated with the architecture. This is not to say that the machine will be easy to build. It will, however, require no more effort than the construction of any prototype of similar complexity -- roughly about the same as for a small mainframe computer system.

The second and almost equally important conclusion is that this line of research is almost impossible to continue without having actual parallel hardware. Many of the algorithms developed for exercising the initial design were near the limits of simulation on a uniprocessor. Because many of the techniques that work best on a parallel processor are of the brute force variety, testing their effectiveness is simply too computationally expensive. Researchers are not willing to wait for three days to get

each new evaluation of a change in a set of parameters. Without the interest of researchers in other fields it is almost impossible to develop a sufficient sample of applications on which to base the next design iteration. Thus until one of these machines or something similar is built and an expertise in its use developed, there is no way to go forward with the design of new architectures except via pure speculation. Even the designs presented here are rather speculative, but at least they have a basis in some experience.

The machine should be replicable. In order to develop a broad expertise in parallel processing there should be a standard parallel processor architecture that can be installed in a large number of facilities. Thus the machine must be kept reasonable in cost, the design must be robust, and it must be maintainable. When computers were one-of-a-kind machines, the growth in their development was very limited. It was the advent of multiple copy machines which led to the development of high level programming languages, operating systems and diverse applications. As long as parallel processors remain one-of-a-kind machines, their development will also be stifled. Not until a machine exists in multiple copies will there be enough experience on which to base the same sorts of development.

In order to accomplish the above it is essential that the design not be too ambitious. The use of mature technology will help, but there are limits to which it can be pushed for the support of an architecture. Current connector technology, for example, is probably about three to five years away from making a pixel per-element-machine, that has full grid connectivity for all of the elements, buildable with high reliability. Until that time it would be best to concentrate on machines that use four to one or eight to one multiplex in communicating between chips. Even once the appropriate connector technology is available, it may be several more years before it is cost effective for such a large application as a full scale CAAPP. This would tend to indicate either the conservative or intermediate designs, presented here, as good choices for construction. The intermediate design provides a good tradeoff of constraints versus goals for the currently mature technology.

A number of important considerations for the design of any parallel image processor, especially those for real-time vision applications, have come out of this research. The most important of these is that the combination of associativity and its related report back capabilities with a square grid array architecture results in a machine with

considerably more power than either purely associative or purely array machines. It was found in developing the applications for the original design that there are three types of algorithms for a machine that presents associative plus square-grid combination. There are those algorithms that are purely associative in nature. There are also algorithms that use only the topological aspect of the architecture. And, there is a third group of algorithms which require both of these. Some examples of this third class are Center of Mass, Flow Field Extraction, Network Operations and the Iconic to Symbolic Transform. This last algorithm demonstrates another important point about this combination -- it makes it possible for a single machine to be used at different levels of abstraction in parallel image processing, from simple enhancement to symbolic processing.

Another interesting conclusion that was drawn from the algorithms research is that many tasks can be performed in two ways in the CAAPP. These two approaches may be characterized as "local propagation" and "global selection". Combining these methods will often result in the task being performed much more quickly than if one of the methods is used alone. For example, in the region labelling problem it was found that the optimum speed would be obtained by first sequentially selecting and labelling

individual large image features using the global feedback and broadcast mechanism. This might be done via some histogram-guided algorithm. Once the large features have been extracted, the smaller features can then be grown in parallel by a local propagation algorithm such as connected component labelling. Because the only regions that remain to be labelled are small, the local propagation mechanism can quickly cover them.

Because there are frequently a large number of small regions in an image, the local propagation mechanism gives a high level of parallelism by labelling all of them in parallel. Similarly, the global selection mechanism achieves high parallelism for the large features because it labels their many pixels at once, via broadcasting. Global selection would, however, be a very inefficient way of labelling many small regions, just as local propagation would take a long time to label large regions. Thus, by combining the two approaches we get a method that, overall, is much more efficient.

There are other design considerations: Pixel-per-element machines are easier to program than those which require folding of the image or data field into separately processed segments. Obviously this is only possible for medium resolution images at this time. High resolution images,

such as those from the Landsat Thematic Mapper, are currently beyond the technology for construction of pixel-per-element machines. The Landsat images do not generally, however, require full image enhancement and analysis in a video frame time, as do vision applications. Without pixel-per-element processing, real time vision is nearly impossible. This illustrates well one of the fundamental differences between standard image processing machines and vision processors.

A fast Some/None and nearly as fast Response Count and Find First are important features of a vision processor. The fast Response Count is especially important for histogram guided image operations such as region segmentation. For that matter, it is important to note that the response count operation is largely responsible for turning what would otherwise be a fairly standard image processor into a vision processor. The ability for higher level processes to statistically probe an image in order to determine the course of further low level processing (without having to read out the image for analysis) appears to be a fundamental aspect of real time vision processors.

The separation of response from activity control adds flexibility to the processing capabilities. Multiple registers provide backing storage for response and activity

patterns without having to disrupt memory. The analysis of instruction usage statistics for lumps in the distribution provides a way of locating bottlenecks in the data paths of the processing element. Elimination of bottlenecks can then greatly increase data throughput. Specifically it was found that memory and comparand must have direct access to the ALU inputs; also that a read-modify-write memory cycle eliminates a bottleneck at the output of the ALU; finally that giving the response bit, activity bit and memory direct access to the neighbor communication network will reduce the load placed on the X register by communications operations.

As always is the case, it is desirable to have more memory associated with a processor. A memory size of 128 bits seems to be reasonable, given the current technology available. If this is too small, a good communication network can be used to simulate larger cells (with lower overall resolution) without too much trouble. It was further found that a bit serial processing element is more cost effective for a machine of this size, than a bit parallel element with a mask.

For the VLSI designer, the conclusions that are most important are that the layout of the processors as long thin strips perpendicular to the control paths is more space efficient than building a square grid of processors. This

is due to the need for fewer repetitions of the control paths in the former, and was found during our initial examination of alternative layout schemes. (In a personal communication with Dr. K.E. Batcher, of Goodyear Aerospace, he concurred, saying tht the designers of the MPP chip had also found this to be true.). Also, higher yield may be obtainable by modularizing the chip into quadrants. This also presents the possibility of highly fault tolerant construction.

The final conclusion about this research is that the proposed processor represents a genuine increase in computing power over the best systems currently available. It provides the ability to transform a raw image into a symbolic representation in real time. Also, there is the fact that it can be built and replicated, which leads to the possibilities for spurring the development of high level parallel languages and the subsequent explosion in applications. In addition to all of this, however, there is something simply awe inspiring about the thought of a machine the size of two equipment racks which can perform 145 billion 16-bit fixed point additions per second.

CHAPTER VIII

FURTHER RESEARCH

The first point in the conclusions leads directly to the first possibility for further research. One of the CAAPP designs should be built so that real experience with parallel hardware can be obtained in preparation for the next generation of parallel machines. Construction of a CAAPP will entail considerable research and engineering.

CAAPP applications form the largest area for potential research. Of special interest are the areas involved with higher level analysis of images. The iconic to symbolic transform is as yet ill defined -- it is unclear how much relational information and what type of information the symbols need to carry. There is also the possibility of doing logical inferencing in the architecture. The computer aided design and image generation areas need further exploration. There is also the area of dynamic process simulation: fluidics, weather prediction, aerodynamic analysis, etc. Of course standard array processing techniques can also be applied. The potential exists for synthetic hologram generation. The list is probably endless and all of it will lead to broadening of the knowledge base from which future parallel architectures can be designed and

on which the first truly general high level parallel programming languages can be developed. Recall that the early languages for serial machines were really just a collection of commonly used operations.

This leads to another area of research: The study of parallel processing operations to extract common features that can become programming language structures. For example, the work on the Game of Life and the various convolutions showed that there is a need for a way of describing the geometry and functions associated with a convolution mask so that a program can automatically generate the optimum distribution path for the data, and minimize the arithmetic involved.

In chapter six it was noted that the other end of the bottleneck between the controller and the array could be widened by the addition of multiple local controllers and perhaps even a hierarchical structure such as a pyramid. The implications of such a scheme on the types of processing and the way that data would move through it need to be further explored. This opens up not only the possibility of reducing the bottleneck, but of a whole new class of algorithms.

The implications of the long distance communication

system proposed in chapter six also need to be examined. This may have a significant impact on the way that symbolic processing is carried out in the machine. It will also tend to reduce the need for transmitting values through the controller in order to distribute them around the array. This could further reduce the bottleneck to the controller. It may also be that another group of applications may be found that are only feasible to implement with this scheme.

Another architectural change which was proposed in chapter six is the addition of vector broadcast registers to the edges of the array. Recall that this mechanism would provide a set of registers at the north and west edges of the CAAPP array, with one vector element per CAAPP row (or column). These vectors could then be broadcast, in parallel, across the array. Essentially the vector broadcast mechanism is an extension of the current comparand broadcast, simply allowing vectors to be transmitted in addition to scalars. Although the vector broadcast is easiest to implement for a full width interchip communications machine, it can be built for a multiplexed version as well. This mostly entails additional programming complexity and a reduction in speed from the non-multiplexed version. Vector broadcast would permit implementation of a large number of classical array operations directly on the

machine. Before this is implemented, however, some study must be put into the problem to determine its cost effectiveness.

Lastly, there is an annoying little problem which so far has defied attempts at analytical proof: the square grid sort presented in chapter five. The author is personally interested in laying to rest the question of whether or not this algorithm actually sorts in the predicted square root of N time, and whether it is extensible to higher dimensional arrays with the corresponding dimensional root of N sorting time. A positive result would have some very interesting implications for sorting theory. A negative result would at least provide some peace of mind.

B I B L I O G R A P H Y

1 Anderson, John R., Bower, Gordon H., "Human Associative Memory", V.H. Winston and Sons, Washington D.C., 1973

2 Ball, J.R. et al, "On the Use of the SOLOMON Parallel Processing Computer", Proceedings of the Fall Joint Computer Conference, 1962, pp. 137-146, AFIPS

3 Barnes, George H., et al, "The ILLIAC IV Computer", IEEE Transactions on Computers, Vol. C-17, No. 8, August 1968, pp. 746-757

4 Barr, A., Feigenbaum, E.A.(eds.), "The Handbook of Artificial Intelligence", William Kaufman, Los Altos CA, 1981

5 Batcher, Kenneth E., "Bit Serial Parallel Processing Systems", IEEE Transactions on Computers, Vol. C-31, No. 5, May 1982, pp. 377-384

6 Bear, J.I., "De memoria et reminiscentia", in W.D. Ross (Ed.) "The Works of Aristotle, Vol. 3", Oxford, Clarendon Press, 1931

7 Bell, C.G., Mudge, J.C., McNamara, J.E., "Computer Engineering: A DEC View of Hardware Systems Design", Digital Press, Bedford MA, 1978

8 Bell, C.G., Newell, A., "Computer Structures: Readings and Examples", McGraw-Hill, NY, 1973

9 Berg, R.O., Schmitz, G.L., Nuspl, S.J., "PEPE -- An Overview of Architecture, Operation and Implementation", Proceedings of the National Elect. Conference, 1972, pp. 312-317

10 Bird, R.M., Cass, J.D., Fuller, R.H., "Study of Associative Processing Techniques", RADC-TR-66-209, Vol. 1, Sept. 1966

11 Boles, J.A., "The Logical Design of the Nebula Computer", Oregon State Univ. PhD. Thesis, June 1968

12 Boles, J.A., Rux, P.T., Weingarten, F.W., "Nebula: A Digital Computer Using a 20MC Glass Delay Line Memory", Communications of the ACM, Vol. 9, July 1966, pp. 503-508

13 Bonar, J.G., Levitan, S.P., "Real Time LISP Using Content Addressable Memory", Proceedings of the 1981 International Conference on Parallel Processing, Bellaire, MI, August 1981

14 Bouknight, W.J., et al., "The Illiac IV System", Proceedings of the IEEE, Vol. 60, No. 4, April 1972, pp. 369-382

15 Bush, Vannevar, "As We May Think", Atlantic Monthly, Vol. 176, July 1945, p. 101

16 Charlesworth, A.E., "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family", IEEE Computer Magazine, September 1981, pp. 18-27

17 Chiang, Shi-Kuo, "The Computation of Window Operations on a Parallel Organized Computer -- A Case Study", IEEE Transactions on Computers, Vol. C-22, No. 1, Jan. 1973

18 Comfort, W.T., "A Modified Holland Machine", 1963 Fall Joint Computer Conference, November 1963, pp. 481-488

19 Control Data Corp., "CDC STAR-100 Reference Manual", CDC Pub. No. 60256000, 1972

20 Conway, Lynn, "The MIT '78 VLSI System Design Course", Xerox Corp., Palo Alto CA, 1979

21 Cordella, L.P., Duff, M.J.B., Levialdi, S., "An Analysis of Computational Cost in Image Processing: A Case Study", IEEE Transactions on Computers, Vol. C-27, No. 10, Oct. 1978

22 Cray Research Inc., "The Cray-1 Computer System", Cray Research Publication No. 2240008 B, 1977

23 Davis, Robert L., "The ILLIAC IV Processing Element", IEEE Transactions on Computers, Vol. C-18, No. 9, September 1969, pp. 800-816

24 DeFiore, Casper R., Berra, P. Bruce, "A Quantitative Analysis of the Utilization of Associative Memories in Data Management", IEEE Transactions on Computers, Vol. C-23, No. 2, February 1974, pp. 121-132

25 Deutsch, E.S., "On Parallel Operations on Hexagonal

Arrays", IEEE Transactions on Computers, Vol. C-19, No. 1Ø, Oct. 197Ø, pp. 982-983

26 Digby, David W., "A Search Memory for Many-to-Many Comparisons", IEEE Transactions on Computers, Vol. C-22, No. 8, Aug. 1973, pp. 768-772

27 Duff, M.J.B., "Review of the CLIP Image Processing System", Proceedings of the National Computer Conference, 1978, AFIPS, pp. 1Ø55-1Ø6Ø

28 Dugan, J.A., Green, P.S., Minker, J., Shinole, W.E., "A Study of the Utility of a Hybrid Associative Memory Processor", Proc. of the 21st ACM National Conference, 1966, pp. 347-36Ø

29 Elovits, H.S., "Automatic Translation of English Text to Phonetics by Means of Letter to Sound Rules", Naval Research Laboratory, AD-AØ21929, Jan. 1976

3Ø Ermann, R. M., Grosky, W. I., "Some Computational and System Theoretic Properties of Regular Processor Networks", Georgia Institute of Technology

31 Estabrook, K., Weiss, S., "TITANIC (Part 2); Will It Float?", Univ. of Mass. Senior Project, 1983

32 Estrin, G., Fuller, R.H., "Some Applications for Content Addressable Memories", Proceedings of the Fall Joint Computer Conference, 1963, pp. 495-5Ø8, AFIPS

33 Etchells, R. David, Nudd, Graham R., "Software Metrics for Performance Analysis of Parallel Hardware", Proceedings of the DARPA Image Understanding Workshop, 1983, pp. 137-147

34 Ewing, Richard G., Davies, Paul M., "An Associative Processor", Proceedings of the Fall Joint Computer Conference, 1964, AFIPS

35 Fernstrom, Christer, "The LUCAS Associative Array Processor and Its Programming Environment", Univ. of Lund, Sweden, 1983

36 Flynn, Michael J., "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, Vol. C-21, No. 9, September 1972, pp. 948-96Ø

37 Foster, Caxton C., "A View of Computer Architecture", Communications of the ACM, Vol. 15, No. 7, July 1982,

pp. 557-565

38 Foster, C.C., "Determination of Priority in Associative Memories", IEEE Transactions on Computers, Vol. C-17, No. 8, Aug. 1968, pp. 788-789

39 Foster, C.C., "Some Simple Algorithms for Content Addressable Memories", COINS Tech Note TNCS-00016, Univ. of Mass., July 1970

40 Foster, C.C., "Content Addressable Parallel Processors", Van Nostrand Reinhold, New York, 1976

41 Foster, C.C., "A Simulated Associative Memory", COINS Technical Note No. TNCS-00023, Univ. of Mass., Dec. 1970

42 Foster, C.C., "Controlled Shifting in the Response Store of a Content Addressable Parallel Processor", Unpublished

43 Foster, C.C., "A Gap Crossing Shifter for the Response Store of a Content Addressable Memory", Unpublished

44 Foster, Caxton C., Stockton, Fred D., "Counting Responders in an Associative Memory", IEEE Transactions on Computers, Vol. C-31, No. 12, Dec. 1971, pp. 1580-1583

45 Foster, Caxton C., "Parallel Execution of Iterative Algorithms", Goodyear Aerospace Report GER-11857, 1965

46 Foster, Caxton C., "Computer Architecture", Van Nostrand Reinhold, New York, 1976

47 Fuller, R.H., "Content Addressable Memory Systems", UCLA Dept. of Engineering Report No. 63-25, 1963

48 Fuller, R.H., "Associative Parallel Processing", Proceedings of the Spring Joint Computer Conference, 1967, AFIPS

49 Fuller, R.H., Bird, R.M., "An Associative Parallel Processor with Application to Picture Processing", Proceedings of the Fall Joint Computer Conference, 1965, AFIPS

50 Gilbert, R., "Microprocessors as CAM Cells", Univ. of Mass. Master's Thesis. 1976

51 Golay, Marcel, J.E., "Hexagonal Parallel Pattern Transformations", IEEE Transactions on Computers, Vol. C-18, No. 8, August, 1969, pp 733-740

52 Goodyear Aerospace Corp., "Hybrid Associative Computer Study, Volume Two -- Appendices", RADC Technical Document Report No. RADC-TDR-65-Volume Two, 1965

53 Goos, G., Hartmanis, J.(eds.), "Lecture Notes in Computer Science: Parallel Processing", Springer-Verlag, NY, 1975

54 Gray, Stephen B., "Local Properties of Binary Images in Two Dimensions", IEEE Transactions on Computers, Vol. C-20, No. 5, May 1971, pp. 551-561

55 Gregory, J., McReynolds, R., "The SOLOMON Computer", IEEE Transactions on Electronic Computers, December 1963, pp. 771-781

56 Habermann, A.N., "Parallel Neighbor Sort. Or, The Glory of the Induction Principle", NTIS document number AD-759248

57 Hall, J. Storrs, "System Design and Programming for a CAM Based General Purpose Computer", Rutgers Technical Report LCSR-TR-16, 1981, New Brunswick, NJ

58 Hillis, W.D., "The Connection Machine", A.I. Memo No. 646, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, September 1981

59 Higbie, Lee, "Applications of Vector Processing", Computer Design Magazine, April 1978

60 Hobbs, L.C., et. al. (Ed.), "Parallel Processor Systems, Technologies and Applications", Spartan Books, New York, 1970

61 Holland, J.H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously", Proceedings of the 1959 Eastern Joint Computer Conference, AFIPS, pp. 108-113

62 Hough, A.A., "Tune Recognition Using a Content Addressable Memory", Univ. of Mass. Master's Thesis, 1982

63 Hwang, Kai, "Computer Arithmetic: Principles, Architecture and Design", John Wiley and Sons, NY,

1979

64 Joint Technical Committee on Terminology, "IFIP-ICC vocabulary of information processing", North-Holland Publishing Co., Amsterdam, 1966

65 Klir, George J., "Introduction to the Methodology of Switching Circuits", D. Van Nostrand, NY, 1972

66 Kohonen, T., "Content Addressable Memories", Springer-Verlag, New York, 1980

67 Kruse, Bjorn, "A Parallel Picture Processing Machine", IEEE Transactions on Computers, Vol. C-22, No. 12, Dec. 1973, pp. 1075-1087

68 Kruzela, Ivan, "An Associative Array Processor Supporting a Relational Algebra", Univ. of Lund, Sweden, 1983

69 Kuck, David J., "ILLIAC IV Software and Application Programming", IEEE Transactions on Computers, Vol. C-17, No. 8, August 1968, pp. 758-770

70 Kuck, David J., "Parallel Processor Architecture -- A Survey", Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, pp. 15-39

71 Kung, H.T., Sproull, B., Steele, G. (eds.), "CMU Conference on VLSI Systems and Computations", Computer Science Press, Rockville MD, 1981

72 Levitan, Steven P., "Algorithms for a Broadcast Protocol Multiprocessor", 3rd International Conference on Distributed Computing Systems, Miami and Ft. Lauderdale, FL, Oct. 18-22, 1982

73 Love, Hubert H., "The highly parallel supercomputers: definitions, applications and predictions", AFIPS Conference Proceedings for the 1980 NCC, Vol. 49, pp. 181-190

74 Lowery, Michael R., Miller, Allan, "General Purpose Very Large Scale Integration (VLSI) Chip for Computer Vision With Fault Tolerant Hardware", SPIE Vol. 281, pp. 342-345, 1981

75 Mano, M.M., "Computer System Architecture", Prentice-Hall, Englewood Cliffs NJ, 1982

76 Matney, Roy M., "Parallel Computing Structures and Algorithms for Logic Design Problems", Management Information Services, Detroit, MI, 1970

77 McKeever, B.T., "The Associative Memory Structure", Proceedings of the Fall Joint Computer Conference, 1965, pp. 371-388, AFIPS

78 McCormick, Bruce H., "The Illinois Pattern Recognition Computer -- ILLIAC III", IEEE Transactions on Electronic Computers, Dec. 1963, pp. 791-813

79 Mead, C.A., Conway, L.A., "Introduction to VLSI Systems", Addison Wesley, Reading, MA, 1980

80 Mill, J., "Analysis of the phenomena of the human mind", Longmans, Green, Reader and Dyer, London, 1869, p. 264

81 Minker, Jack, "An Overview of Associative of Content Addressable Memory Systems and a KWIC Index to the Literature: 1956 - 1970", ACM Computing Reviews, Oct. 1971

82 Montalvo, Fanya S., Foster, Caxton C., "An Algorithm for Intercell Communication in a Tesselated Automaton", COINS Dept. UMASS, Technical Note No. TN/CS/00037, Amherst, MA

83 Nagel, H.T., Carroll, B.D., Irwin, J.D., "An Introduction to Computer Logic", Prentice-Hall, Englewood Cliffs NJ, 1975

84 Nagin, Paul A., Hanson, Allen R., Riseman, Edward M., "Region Extraction and Description Through Planning", COINS Technical Report 77-8, Univ. of Mass., May 1977

85 Nagin, Paul A., Hanson, Allen R., Riseman, Edward M., "Region Relaxation in a Parallel Hierarchical Architecture", COINS Dept., Univ. of Mass.

86 Nassimi, David, and Sahni, Sartaj, "Bitonic Sort on a Mesh Connected Parallel Computer", IEEE Transactions on Computers, Vol. C-27, No. 1, Jan. 1979, pp. 2-7

87 Natarajan, N.K., Thomas, Paul A.V., "A Multiaccess Associative Memory", IEEE Transactions on Computers, Vol. C-18, No. 5, May 1969

88 Newman, W.M., Sproull, R.F., "Principles of

Interactive Computer Graphics", McGraw-Hill, NY, 1971

89 Nickodemus, W.A., (Ed.) "Progress Report on the Nebula Computer", Oregon State Univ. CC-66-1, Jan. 1966

90 Nudd, Graham R., "Image Understanding Architectures", AFIPS Conference Proceedings, Vol. 49, 1980, pp. 377-390

91 Onoe, M., Preston, K., Rosenfeld, A. (eds.), "Real-Time/Parallel Computing: Image Analysis", Plenum Press, NY, 1981

92 Ramamoorthy, C.V., Turner, James L., Wah, Benjamin W., "A Design of a Fast Cellular Associative Memory for Ordered Retrieval", IEEE Transactions on Computers, Vol. C-27, No. 9, Sept. 1978, pp. 800-815

93 Reeves, Anthony P., "A Systematically Designed Binary Array Processor", IEEE Transactions on Computers, Vol. C-29, No. 4, April 1980, pp. 278-287

94 Reeves, Anthony P., Bruner, John D., "Efficient Function Implementation for Bit Serial Parallel Processors", IEEE Transactions on Computers, Vol. C-29, No. 9, Sept. 1980

95 Richard, F., "Compiling Techniques for Associative Processors", Univ. of Mass. PhD Dissertation, 1977

96 Rupp, Charle' R., PEPE lecture notes, Univ. of Mass. Oct. 1975

97 Russell, Richard M., "The Cray-1 Computer System", Communications of the ACM, Vol. 21, No. 1, January 1978, pp. 63-72

98 Rux, P.T., "A Glass Delay Line Content Addressed Memory System", IEEE Transactions on Computers, Vol. C-18, No.6, June 1969, pp. 512-520

99 Rux, P.T., "Design and Evaluation of a Glass Delay Line Content Addressable Memory System", Oregon State Univ., Feb. 1968

100 Rux, P.T., "Evaluation of Three Content Addressable Memory Systems Using Glass Delay Lines", Oregon State Univ., July 1967

101 Rux, P.T., Weingarten, F.W., Young, F.W., "Serial

Associative Memories", IEEE Computer Group Repository No. 62-72, Mar. 1967

102 Savitt, D.A., Love, H.H., Troop, R.E., "ASP: a new concept in language and machine organization", Proceedings of the Spring Joint Computer Conference, 1967, AFIPS

103 Shooman, W., "Parallel Computing with Vertical Data", Proceedings of the 1960 Eastern Joint Computer Conference, AFIPS, pp. 108-113

104 Siegal, Howard Jay, Siegel, Leah J., Kemmerer, Frederick C., Mueller Jr., Phillip T., Smalley Jr., Harold E., Smith, S. Diane, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", IEEE Transactions on Computers,Vol. C-30, No. 12, Dec. 1981, pp. 934-946

105 Siegel, Leah J., Siegel, Howard J., Feather, Arthur E., "Parallel Processing Approaches to Image Correlation", IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982, pp. 208-218

106 Siegel, Leah J., Siegel, Howard Jay, Swain, Philip H., "Performance Measures for Evaluating Algorithms for SIMD Machines", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, July 1982, pp. 319-330

107 Sklansky, Jack, Cordella, Luigi P., Levialdi, Stefano, "Parallel Detection of Concavities in Cellular Blobs", IEEE Transactions on Computers, Vol. C-25, No. 2, Feb. 1976, pp. 187-195

108 Slade, A.E., McMahon, H.O., "A Cryotron Catalog Memory", Proceedings of the Eastern Joint Computer Conference, 1956

109 Slotnick, D.L., Borck, W.C., McReynolds, R.C., "The SOLOMON Computer", Proceedings of the Fall Joint Computer Conference, 1962, pp. 97-107, AFIPS

110 Stamopoulos, C. D., "Parallel Algorithms for Joining Two Points by a Straight Line Segment", IEEE Transactions on Computers, Vol. C-23, No. 6, pp. 642-646

111 Stamopoulos, Charlambos D., "Parallel Image Processing", IEEE Transactions on Computers, Vol. C-24, No. 4, April 1975, pp. 424-433

112 Steenstrup, M.E., Lawton, D.T., Weems, C.C., "Determination of Rotational and Translational Components of a Flow Field Using a Content Addressable Parallel Processor", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, June 1983

113 Stevenson, David K., "Numerical Algorithms for Parallel Computers", AFIPS Conference Proceedings, Vol. 49, 1980, pp. 357-361

114 Stillman, Neil J, Defiore, Casper R., "Associative Processing of Line Drawings", Proceedings of the Spring Joint Computer Conference, 1971, pp 557-562, AFIPS

115 Stone, H.S. (ed.), "Introduction to Computer Architecture", Science Research Associates, Chicago IL, 1975

116 Svensson, Bertil, "Lucas Processor Array: Design and Applications", Univ. of Lund, Sweden, 1983

117 Thurber, Kenneth J., "Large Scale Computer Architecture", Hayden Books, New Jersey, 1976

118 Thurber, Kenneth J., Patton, Peter C., "The Future of Parallel Processing", IEEE Transactions on Computers, Vol. C22, No. 12, Dec. 1973, pp. 1140-1143

119 Unger, S.H., "A Computer Oriented Toward Spatial Problems", Proceedings of the IRE, October 1958, pp. 1744-1750

120 University College, London, CLIP-IV lecture notes

121 Wall, Rajendra S., "Decryption of Simple Substitution Cyphers with Word Divisions Using a Content Addressable Memory", Cryptalogia, Vol. 4, No. 2, April 1980, pp. 109-115

122 Weems, C.C., Levitan, S.L., Foster, C.C., "Titanic: A VLSI Based Content Addressable Parallel Processor", Proceedings of the IEEE International Conference on Circuits and Computers, Sept. 1982

123 Weems, C.C., "Life is a CAM Array Old Chum", Unpublished

124 Weems, C.C., "An Algorithm for a Simple Image Convolution on the Titanic Content Addressable Parallel Array Processor", COINS Technical Report No. 83-07, Univ. of Mass., 1983

125 Weems, C.C., Levitan, S.L., Foster, C.C., Lawton, D.T., Steenstrup, M.E., Wall, R.S., "Titanic Information Kit", COINS Technical Report No. 83-32, Univ. of Mass., 1983

126 Weingarten, F.W., Rux, P.T., Boles, J.A., "On an Associative Memory for the Nebula Computer", Oregon State Univ., 1964

127 Whitesitt, J.E., "Boolean Algebra and its Applications", Addison Wesley, Reading MA, 1961

128 Winograd, S., "On the Speed Gained in Parallel Methods", IBM Research Note RC 4670, Yorktown Heights, NY, Jan. 1974

129 Yau, S.S., Fung, H.S., "Associative Processor Architecture -- Survey", Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, pp. 1-14

130 Zobrist, Albert L., Thompson, William B., "Building a Distance Function for Gestalt Grouping", IEEE Transactions on Computers, Vol. C-24, No. 7, July 1975, pp. 718-728

APPENDIX A

A HISTORICAL REVIEW OF ASSOCIATIVITY

Early Conceptions of Associativity

Aristotle can generally be credited with the first published account of associativity in human memory. In his essay "On Memory and Reminiscence" he recognizes one type of rememberance in which one probes ones memory with a particular thought that leads through a chain of associations to some desired recollection. Specifically he states: "Accordingly, therefore, when one wishes to recollect, this is what he will do: He will try to obtain a beginning of a movement whose sequel shall be the movement he desires to reawaken [6]."

Aristotle's theory of associativity in the human mind went essentially unchanged until the mid seventeenth century when a number of British philosophers and psychologists began to build upon it with various theories of the underlying mechanism. This group of theories is collectively called British Associationism by Anderson and Bower [1]. The particular theories range from "Human Nature" by Thomas Hobbes (1650) to "The Senses and Intellect" by Alexander Bain (1859). Other British associationists

included John Locke, Bishop Berkeley, David Hume, David Hartely, John and James Mill. The common thread which runs through these theories is that complex ideas are composed from associations of simpler ideas. For example, Mill states that

"The word, man, we shall say, is first applied to an individual; it is first associated with the idea of that individual, and acquires the power of calling up the idea of him; it is next applied to another individual, and acquires the power of calling up the idea of him; so another, and another, till it has become associated with an indefinite number, and has acquired the power of calling up an indefinite number of those ideas indifferently. What happens? It does call up an indefinite number of the ideas of individuals, as often as it occurs; and calling them up in close connexion, it forms them into a species of complex idea." (Mill [80])

By the 1930`s John Watson`s theory of behaviorism had strongly modified theories of associativity, especially among American psychologists and philosophers. This lead, for example to the Stimulus-Response theory of behavior which described memory as associations between stimuli (both external and internal) and physical responses.

MEMEX

It was in this climate that the first mechanical associative memory was described by Vannevar Bush in an article for Atlantic Monthly magazine in 1945 [15]. The memory, called MEMEX, was never intended to be built but was simply a thought exercise illustrating how a machine might be constructed to automate information retrieval. The machine was described as an extension to human memory that would help people cope with the ever growing body of general knowledge. Bush envisioned three ways of accessing information stored in the machine: sequentially, by successive selection of subclasses, and by association. Bush asserted that the mind works in an associative manner and further stated that "Selection by association, rather than by indexing, may yet be mechanized." The hardware for the processor included data stored on microfilm, a form of graphics terminal and dry photocopying for hardcopy.

Cryotron Catalog Memory

The first functional associative memory was described by Slade and McMahon [108] in 1957. Called a Cryotron Catalog

Memory, it was based on the principle that the temperature at which metal becomes superconductive varies slightly under the influence of a magnetic field. This was implemented as a set of tantalum wires wrapped in coils of niobium wire and immersed in liquid helium. The application of a current to a coil causes the wire that it surrounds to take on a slight resistance. The ratio of this to the superconductive resistance (which is zero) is of course infinite. The reason for selecting such unusual materials for the components is that tantalum becomes resistive under a very slight magnetic field while niobium requires a much stronger field to become resistive. Thus the coils can remain superconductive even when the "gate" wires are resistive. This permits the construction of a flip-flop circuit by cross coupling coils and gates in recirculating superconductive loops. When one coil is energized, it quenches the current flowing through the gate wire that it surrounds. Because this gate is connected in series with the coil that controls the gate of the circuit containing the energized coil, no magnetic field will be created in the series connected coil. Thus the entire circuit containing the energized coil will remain superconductive and therefore the current in the circuit will recirculate indefinitely. Because either of the two coils can be energized to select one of the two loops, the circuit is bistable. These

flip-flops can then be combined to form a memory in which the presence of a bit pattern can be detected, but not read out. When an interrogating pattern of currents is passed through the memory, some of the bits become resistive. If an exact match occurs all of the bits in a word become resistive. The words are wired so that all of the bits in each word form legs of parallel circuits. The words, on the other hand, are wired in series. Thus an exact match will result in the overall memory becoming resistive. In the author's system this condition was simply indicated by a voltmeter that registered the resulting voltage drop. The original article states that the authors had successfully constructed a twenty bit memory as an initial test prior to building a 200 bit feasability test model. The access time for the device was 10 microseconds, however the write time was 500 microseconds. This slow operating speed, combined with other difficulties, eventually put an end to research in cryotrons. There are two important points to note about this system. The first is that the device was actually a bit parallel, word parallel associative memory in the modern sense. The second point to note is that this first associative machine used a some/none response circuit as an (and indeed its only) output.

Associative Processor and APP

During the mid 1960's at least two designs were presented that made use of plated wire memory for CAM. The two that we will look at here are the Associative Processor [34] and the Associative Parallel Processor (APP) [10,47,48,49]. Both of these were sponsored by the Air Force and have very similar designs. The Associative Processor preceded the APP and was described as having 512 words of 96 bits in its associative array. Although no exact figures were given to describe the size of the APP memory, reference was made in the articles to an algorithm which would use 2000 words of associative memory. The access time of the Associative Processor was given as 0.1 microsecond while the APP was described as having a 0.8 microsecond cycle time.

Both the Associative Processor and the APP were designed to operate on data in a bit serial fashion. To facilitate this, they both had sets of registers for counting through bit positions of two separate fields. These consisted of a counter and a limit register for each field. Both machines provided two sets of these registers so that two data fields could be manipulated at once. The Associative Processor additionally provided a third bit counter register with no matching limit register. Both machines also had separate, dedicated processing control units which used standard

random access memory for instruction storage.

The Associative Processor and the APP differ most in their methods of implementing the bit serial processing. In the Associative Processor the CAM consists of standard random access memory where all words are accessed in parallel, one bit at a time. All operations are therefore performed bit serially. In the APP, however, each bit contains a comparator and thus the equality comparison operation can be performed in parallel both by word and by bit. Arithmetic and logical operations are all performed bit serially through the use of a mask using a method that the machine's designers call "sequential state transformation". Examples of this method can be seen in the algorithms for addition described in Chapter 2. In general, the technique treats each combination of states for the bits being operated on bit serially. Thus for an operation involving two one-bit operands there would be four states that would have to be selected separately by comparison searches and then operated on. The number of states remains the same for multi-bit operands as long as the function being applied to them treats each pair of bits separately (as in a logical AND). If the bits interact, additional states will be required to represent the interaction (as with arithmetic addition which requires states to represent

the various carry conditions between bits). The number of states that must be processed can be reduced in many cases by processing redundant states together, ignoring impossible states or taking advantage of shared operations required by some different states.

GAP or the RADC 2048 word CAM

Another plated wire memory CAM system that was actually built in the mid 1960's was the Goodyear Associative Processor (GAP). This was generally called the RADC 2048 word CAM because it was actually built for the Air Force Rome Air Development Center (RADC) and consisted of 2048 words of 50 bits each. Two of these bits had special uses. One of them was used for parity checking while the other indicated whether a cell was busy or not. The CAM operated in word parallel, bit serial fashion. The 2048 words were divided into two banks of 1024 words, called the upper and lower banks. The two banks serially shared a 1024 element response store. The actual implementation of this sharing scheme had 2048 response bits, but in order to reduce complexity and cost there were only 1024 memory driver circuits for the response store. Thus the sharing really only involved the driver circuits with the result that speed

was reduced by a factor of two.

The GAP also contained mask and comparand registers (the latter was called the "data" register) and its own control unit. The GAP control unit was capable of branching and indexing and could thus execute small programs autonomously. The rest of the installation consisted of a CDC 1604 computer with a CDC 818 disc acting as a host system. The GAP itself was primarily intended as a study vehicle to help pave the way for more advanced associative processors being considered by RADC. Because of this there was nothing terribly innovative about the design. Except for some of the architectural aspects resulting from cost cutting, the GAP was a very standard CAM. The interesting aspect of this system was the evaluation study that was based upon it. The "Hybrid Associative Computer Study"[52], as it was called, evaluated the performance of the GAP when applied to several interesting problems. The study also evaluated the performance of four alternate architectures (variations of the GAP architecture) on these problems in comparison to the GAP.

In the study the GAP was referred to as "HB" while the other four designs were known as "H1", "H21", "H22" and "H23". H1 was the GAP specially modified for the automatic abstraction problem. The other three involved various

modifications to enhance performance on the spelling correction problem. The H1 design added a high speed local instruction and mask store so that the control unit did not have to get its instructions and masks from the 1604 memory. A tailored instruction set was also provided and all excess hardware (index registers, adders, accumulators, etc.) was removed. H21 was like H1 but with the indexing capability reinstalled and some special instructions for spelling checking. H22 was an expanded associative processor, containing 10,000 words of 96 bits but with the 1024 element response store still shared by the cells. The H23 was the architecture most optimized for spelling checking. It was like the H22 but with a full 10,000 element response store, no instruction register (all commands came directly from the 1604) and a very specialized instruction set.

The study identified three facilities that are desireable in the control unit of a CAM. These were indexing and looping facilities to allow the CAM to execute autonomously, a fast data load and dump mechanism that operates independently of the host processor, and a means of retaining a mask value from instruction to instruction so that new mask values need not be specified for each command. The study also showed that when a special purpose

CAM is applied to a problem for which it is not intended that the processing speedup will often be negligable. The speedups attained in the study ranged from factors of three to 186. (Neither of the problems was well suited to processing by CAM because in both the data was loaded and searched but not processed in any way. The memory was thus idle 70 to 80 percent of the time. The designs that relied on the 1604 for their instruction streams also spent about 50 percent of the time waiting for the host when running the spelling checking problem.)

NEBULA

Another interesting device that has been used to implement CAM is the glass delay line. This device consists of a block of glass which has a pair of acoustic transducers mounted on it. One of the transducers injects a high frequency acoustic signal at one end of the block while the other detects this signal at the other end of the block. Because the time required for this signal to pass through the block is long compared with the switching time of electronic digital circuits, the delay line can be used as a memory element. This is done by applying a string of bits as the signal entering the delay line. Because each of the

acoustic wavefronts progresses through the glass at the same speed as all of the others, the bit pattern presented at one end of the block is preserved as it passes through the glass and will reemerge at the other end. In a typical configuration it will then be recirculated to the transmitting transducer via an electrical circuit that passes through the processing unit of a computer. The total storage capacity of the delay line is equal to the length of the traversal time multiplied by the maximum data rate. The latter is dependent upon the the acoustical properties of the glass and the transducers (primarily the maximum signal frequency).

This type of memory was used in the implementation of the Nebula computer and CAM at Oregon State University [11, 12, 89, 98, 99, 100, 101, 126] in 1967. Both the controlling processor and the CAM used glass delay lines for data storage. Each delay line had a delay time of 100 microseconds and was capable of storing 2048 bits. The controlling processor memory contained 4096 words of 34 bits (68 delay lines) and the CAM contained 2048 words of 34 bits (34 delay lines). The unusual word size is based on 32 bits of data, one parity bit and one "spare" bit. In the CAM, the spare bit becomes the response bit. In the actual implementation, a thirty-fifth delay line was added to the

CAM to provide an extra control bit called the "use bit", although this wasn't accessable to the programmer.

Although 100 microseconds seems rather long for a CAM search operation, it should be remembered that this was only 50 to 100 instruction times for an average processor in 1967. Even today an access time of 48 nanoseconds per word is quite respectable. The word serial, bit parallel nature of the architecture led to some interesting developments. One of these was that both searches and operations could be performed in one memory cycle. This meant that the response bit did not need to be used to store the result of every search. Many searches select words for just one operation. Another result of the word serial nature of the memory was that counting the number of responders required only one memory cycle time. Also, the fact that only one processing element was required for all 2048 words meant that more resources could be put into its construction and thus it could have a considerably more powerful repertoire of instructions than a word or bit parallel machine of comparable cost. The machine was actually "designed to be changed" in that it was quite easy to add new instructions to the set. This was because the instructions were essentially function modules that were attached to the recirculating data path. Another advantage to the word

serial mode of processing is low cost. The entire Nebula system cost only $50,000 to construct, with most of the cost attributable to the glass delay line memory modules.

The Nebula control processor began operation in 1966 and the CAM came on line late in 1967. In 1975 a 16K word core memory replaced the delay lines in the controlling processor. The machine was dismantled in 1978 so that it could be moved to a new location but due to a lack of funding it was never reassembled. During the 12 years it was in operation a number of software packages were developed for it including an assembler, a FORTRAN compiler which had special instructions and subroutine packages to support the use of the CAM, an ALGOL subset compiler and a multiuser operating system.

ASP

In 1967 Savitt, Love and Troop [102] proposed a system called the Association Storing Processor (ASP) which consisted of both a language for manipulating complex networks of associations and a processor designed to run the language. Although it was never built, ASP incorporates a number of interesting features that are being implemented in at least one machine currently under construction (the MIT

Connection Machine).

The ASP system is based upon the idea of manipulating data structures that explicitly include the associations between various pieces of information stored in the system. The basic unit of data in ASP is the "relation". This consists of three components, an ordered pair of items and a link which specifies the type of association between them. Simple items and links are character strings. Sample links are "is_an_example_of", "is_a_type_of", "is_assigned_to", etc. Thus we might have ASP relations "Pufton_House is_an_example_of House", "Colonial is_a_type_of House" or "VAX4 is_assigned_to Pufton_House". An item in a relation may itself be an association between two items and have the form of a relation. Such an item is called a "compound item" in the ASP system.

An instruction in the ASP system specifies a conditional transformation to be performed on the data. This involves a search of the data for matches to one set of relations and, if this succeeds, the replacement of the located elements by another set of relations. ASP instructions themselves are expressed as structures of relations and are stored with the data. A special item and several special link labels are used to represent instructions. Each instruction includes items called "name", "control structure", "replacement

structure" and "go to". The name item identifies the individual instruction. The control structure item is a relation that is to be searched for. The replacement structure is the relation that will replace any relations found to match the control structure. The go to item specifies the name of the next instruction to be executed. The control and replacement structures may be arbitrarily complex. ASP also provides for matching structures by implication. This allows, for example, indirect matches to be made between a set of stored relations and a single control structure relation on the basis of the data relation link having transitive properties.

The hardware of the ASP system executes searches and replacements over all of the data in parallel. In order to do this it incorporates what the authors call a "context addressed memory". The context addressed memory is a CAPP in which the cells are arranged as a square grid with communication lines running between the cells South to North and East to West. Thus each cell can accept data from either its South or East neighbor and transmit data to either its North or West neighbor. The authors propose that the array be 1024 by 1024 in size (1,048,576 cells). The memory of each cell is divided into five equal sized fields. Three of these store data and instruction information and the other

two hold tags. Each cell also has its address permanently wired into it. The cell logic performs comparison operations and generates and propagates the inter-cell communication signals. In addition it records the results of a comparison and performs Boolean operations on the results of successive comparisons.

There are eight basic memory operations in ASP. The first of these is the "search" operation. Its function is identical to the content addressing function in a normal CAM. Second is the "context address" operation. This selects in parallel all cells which are specified in a particular data field of a set of transmitting cells. To accomplish this, the selected cells transmit the contents of the data field (the field contains the address of another cell) to their West or North neighbors. The choice of neighbor depends upon the direction the signal must go in order to reach the destination cell. The signals are propagated in parallel along the appropriate path (North to the proper row, then West to the proper column -- wrapping around to the opposite edge in both cases if necessary) to the destination cell, where is sets the "match tag". The third operation is the "box car". This operation selects, in parallel, all members of a transmitting set of cells which specify, in a particular data field, members of a

second set of cells. This works in a way that is very similar to the "context address" operation except that the address of the transmitting cell is carried along with the signal as it is propagated to the destination cell. When the destination is reached, if the destination cell's match tag is set it then uses the address attached to the signal to send a reply to the transmitting cell. If the destination cell's match tag isn't set, it discards the address and takes no further action. The fourth type of operation is the "read". This causes the comparand register of the memory to have its contents replaced by the contents of a cell which has its match tag set. This is accomplished by having every selected cell "box car" to the cell with the lowest address. Instead of passing its address, however, each cell attaches the contents of the specified field to the signal. The cell with the lowest address has as its North neighbor the comparand register, to which it subsequently passes the data. The operation can read either the first responder only, or it may stream all of the responders to some external device. The fifth operation is the "write". This operates in exactly the same way that the multiwrite operation functions in a CAPP. The sixth operation is the "mass write". The mass write is used to simultaneously write the addresses of one set of cells into another set of cells which are tagged as empty. In this

case, all of the selected cells transmit their addresses which propagate to the nearest cell that is empty. The address is then written into that cell. The seventh operation, "reset", clears the match tags in all of the cells. The last operation is called "pulse". It resets a "sequence" tag in the cell's logic if the match tag is not set, and simultaneously resets the match tag in all other cells. The pulse function is used to perform AND and OR functions of a sequence of states of the match tag.

In the case that a propagating signal collides with another in any of the above operations, the East-West signal is allowed to continue and the South-North signal is not allowed to enter the row. The South-North signal must then propagate all of the way around the array before it again reaches its destination row and has another chance to enter it. Recognizing that this could lead to lengthy delays, the authors added a second grid of "routing cells" to the basic grid. They found a 32 by 32 array to be optimal for a 1024 by 1024 basic array. This reduces the maximum propagation distance by an order of magnitude to 186 cells.

STARAN

Perhaps the best known associative processor is the

STARAN, manufactured by Goodyear Aerospace Corp. This machine was based on Goodyear's previous work with the GAP. Although the STARAN, like the GAP, was first built for the Air Force (RADC) it was also made commercially available. STARAN was considerably more powerful than any of its predecessors -- leading Goodyear to proudly declare in its advertisements that STARAN embodied "a new way of thinking". The machine was also very expensive: A 2048 word STARAN was initially priced at about $2.6 million (now considerably less). Surprisingly enough, the considerable complexity of the machine was a result of cost cutting measures. All of this combined to form what Foster called "an architecture that the most generous hearted would label baroque and the small minded might call grotesque" [40].

Part of the expense of STARAN was a result of the hierarchical memory used for program storage in the control unit. This consisted of a few pages of fast bipolar memory (which was then very expensive) that was paged from slower magnetic core pages. The system also shared several pages of control memory with a host computer via direct memory access. This allowed fast communication between the host and STARAN. It also required a rather complex mechanism for managing the memory. This will not be described here because it has little bearing on the associative segment of

the architecture.

In addition to the control memory, STARAN's main elements are the Associative Processor Control (APC) and the Associative Processor Array (APA). The APC is essentially the same as the control unit of any general purpose computer system but with some additions to facilitate associative processing. One such addition is hardware to directly support looping with a counter as a loop control variable (as with a FOR or DO loop) via a single instruction. Another enhancement is a set of field pointer and counter register pairs, similar to those of the APP described above. The APC also contained a register, called the Common Register, that was similar to a normal comparand register in a bit parallel CAM.

The STARAN Associative Processor Array (APA) has a number of interesting features. Perhaps the most interesting of these is the logic used to access the array. One of the cost cutting measures used in the STARAN design was to have the same logic elements perform both bit slice and individual word memory accesses. This was accomplished through a set of elements called the Flip Network (FN). Because of this design, the APA actually consists of one or more square arrays of bits (256 by 256) each with its own FN. A STARAN can be configured with as many as 32 of these

square arrays. Its maximum size is thus 8192 words of 256 bits.

Each 256 bit word has 3 additional bits associated with it, called the M, X and Y bits. The M bit determines whether a word will respond to a multiwrite operation. The Y bit is the response store which holds the results of search operations. The X bit is a temporary storage location that participates in some logical and arithmetic operations. As mentioned above, there is also a vector of 256 bits called the Common register. This can be used, via the FN, as both a means of communicating with all of the bits of an individual word (or a bit slice of all selected words) or as a means of communicating with the X or Y registers of all words in parallel. The Common register can also act as a shift register and thus provides the ability for communication between words within each 256 word block of the APA.

The APC instruction set is microprogrammed in the same sense that the Digital Equipment Corp. PDP-8 minicomputer is. Various bit slices of an instruction would specify different interpretations of other bit slices, leading to a very rich instruction set. In general, almost any combination of register to register or register to memory transfer is permitted, each with a wide variety of options

for how the transfer is to be carried out. Part of the register to register transfer instruction set includes transferring the result of one of sixteen boolean operations between either X or Y and the Common register back into X or Y. There are also instructions for finding the first responder to a search, resetting the first responder and reporting whether there are any responders. The STARAN APC does not provide an instruction for counting responders however this could be accomplished by transferring the response store to the common register and counting the number of one bits contained in it. Of course the APC also has a non-APA instruction subset that closely resembles that of any typical general purpose computer. This includes arithmetic, logical, looping, branching and I/O operations.

Even today, STARAN is considered to be a fast processor. For example, a 32 bit fixed point bit serial addition requires 22.4 microseconds to execute. On a 2048 word STARAN this would result in throughput equivalent to 90 million instructions per second (MIPS) on a serial processor. Although there are quite a few computer systems that are now commercially available which achieve higher throughput, the recent announcement of a VLSI based successor to STARAN that is small enough for airborne applications will no doubt make this architecture's

influence be felt for a long time to come.

RADCAP or SIMDA

At about the same time that Goodyear was building STARAN for the Air Force Rome Air Development Center (RADC), Texas Instruments was also building another associative processor for RADC. This was originally called the RADCAP, but was later renamed SIMDA (the name RADCAP was then given to the RADC STARAN installation in which the host computer was a Honeywell 645 running the MULTICS operating system). SIMDA, like STARAN, was also completed in 1972. It is nearly as complex a design as STARAN and thus only its more salient features will be described here.

A SIMDA cell consists of a 256 word by five bit processing element memory (PEM) and a processing element (PE) which includes a four bit arithmetic-logic unit. One of the five bits in each memory word is used for parity checking. Each PE also contains two four bit arithmetic registers, two registers for microprogram control, and two status registers. The basic unit of communication within SIMDA is, as can be guessed, the four bit "byte".

Each SIMDA PE can perform 48 different operations

including arithmetic, logic and register transfer instructions. Arithmetic instructions operate on 16 bit values, thus these operands must consist of four words of PEM. PEM words are selected globally for all of the cells. The cells themselves can be accessed both by address and associatively. SIMDA included special logic elements for finding minimum and maximum values in PEM words, for selecting a first responder and for counting the number of responders.

The overall array consists of 32 blocks of 32 cells for a total of 1024 cells. Blocks can be independently connected to either the control unit or to I/O channels. This allows processing and I/O to occur concurrently.

SIMDA also provides a rather elaborate scheme for communication between PE's. Each PE can communicate directly with PE's that have addresses up to eight less or seven more than its own address. It can also communicate with all PE's that are a multiple of 16 away and those that are a multiple of 128 away. The configuration is such that no PE is more than 3 levels of communication away from any other.

General Comments

Even though associative systems have been built which are also parallel processors, researchers in computer architecture have always distinguished them from the mainstream of parallel processor architectures. This distinction is not strongly justified by differences in hardware organization so much as by differences in applications. Indeed, many mainstream parallel processor systems are quite capable of operating, with some loss of efficiency, as associative processors. The reason that they are not used as such is mostly that associative processing is not the purpose for which they were designed. As a result, their users are not oriented toward looking for applications which use this facet of the processor's capabilities. Those users who do so will often encounter deficiencies in the processor that make it difficult to effectively use the associativity.

Associative processors, on the other hand, have most often been designed for applications that chiefly use the associative facet of the architecture. In this regard, they have often run into trouble for lack of applications in which they can be used more effectively than, for example, a pipeline processor or a general purpose processor using a

sophisticated hashing table algorithm. The reason for this is that an associative processor is only efficient when the data that it is working on can remain in the memory for a large number of associative operations. The time required to load or unload an associative memory can be quite long in comparison to most other operations. When this time is added as overhead to the associative processing, efficiency is greatly reduced.

If only simple operations are to be performed on a block of data after it is loaded, the associative processor is not significantly faster (and is considerably more expensive) than a pipelined processor that operates by serially streaming the data through a special purpose function unit. (This is easy to see if one considers that the time to stream the data through a processor is roughly equal to the time required to serially load the same data into a memory.)

Even if many associative operations are to be performed on a data set, an associative processor may not be as cost effective as a general purpose processor using a hashing table algorithm. With a sophisticated hashing scheme it is quite possible for a general purpose computer to search a data set with a speed that is within an order of magnitude of that of an associative processor. There are four

situations in which an associative processor will be sufficiently faster to justify its greater cost: 1) If the data set is one for which it is difficult to compute a hashing function (because, for example, the range of key values is extremely large with data values very unevenly distributed in an unpredictable fashion). 2) If the data set is one for which searches frequently result in the selection of many data values that can be processed in parallel. 3) When the search criteria are extremely varied or even created dynamically as a result of other processing of the data. 4) When the data itself is very dynamic.

These restrictions severely limit the number of applications that are suitable and cost effective for associative memories and processors. This has lead to the common description of associative systems as a solution in search of a problem. Indeed, except for research machines, large systems are used only in "cost is no object" applications such as air traffic control and military radar tracking.

For further information about associative processors and processing the reader is referred to: DeFiore [24], Estrin [32], Foster [38, 39, 41, 42, 43, 44, 45], Digby [26], Gilbert [50], Hall [57], Kohonen [66], Kruzela [68], McKeever [77], Natarajan [87], and Ramamoorthy [92], Richard

[95], Yau [129].

APPENDIX B

A HISTORICAL REVIEW OF SIMD PARALLEL PROCESSORS

von Neumann's Cellular Computer

The first design for an electronic, digital parallel processor came from John von Neumann who is also credited with originally designing the stored program computer architecture on which most modern computers are based. His parallel processor design is referred to as von Neumann's Cellular Automata. It was an attempt to mathematically model neurons as simple entities. The design consisted of a two dimensional array of cells that could communicate with their North, South, East and West neighbors. Each cell had 29 states which represented the excitation state of a cell, its inputs and outputs. A set of transition rules specified the next state a cell would take in the following stage of processing. Although the machine was clumsy to program, it was shown by von Neumann that it could be made computationally universal in the same way that a Turing machine is.

The Spatial Computer

Another early parallel processor was the Spatial Computer described by Unger [119] in 1958. This is, in general form, the prototype for most of the major SIMD parallel processors that would follow in the ensuing 25 years. The Spatial Computer was another square array of processing elements. Unlike von Neumann's highly theoretical model of the neuron, Unger's processing elements were quite down to Earth. They consisted simply of a one bit register, a few bits of memory and a small amount of logic. In fact, each cell required only 11 flip flops, 170 gate inputs and had only 30 logic gates. The whole array was controlled by a central unit that would broadcast instructions to the processing elements. The central control was not able to read from the cells, except for a some/none type response line that was the logical OR of the registers of all of the cells. It was, however, possible to load and unload the cells directly to memory external to the array. This machine was in some sense a bit serial CAPP although it did not provide the usual multiwrite facility. In the Spatial Computer there was no actual cell activity mask -- a pattern could be built in the accumulators (the one bit register) of all of the cells by arithmetic operations with memory bits and this could then be stored

back into the memory. The mechanism is thus equivalent to multiwrite and, through clever use of arithmetic, to comparison searching.

Holland's Machine

In 1959 John Holland [61] proposed "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously". Although this was not a typical parallel processor architecture and was never built, it introduced the important concept of activity control in each of the processing elements. In both von Neumann's and Unger's machines all of the processing elements responded to all of the instructions or time steps issued by the central control. (It was possible that a cell would not be changed by an instruction but this is not quite the same.) In Holland's machine a cell could actually be independently disabled such that it would only respond to instructions that caused it to be enabled.

Holland's machine, like von Neumann's, was a theoretical study. Its main purpose was to explore concurrent execution of multiple programs from the viewpoint of automata theory. Thus there was no specification for input, output or other practical aspects of the machine design. Although it was

specified as a rectangular two dimensional array of processors, it was not constrained to this dimensionality.

The basic idea behind Holland's architecture was that processing and communications between programs would take the form of path building between processors. Many programs would be running on the machine simultaneously, independently and asynchronously (although the processors themselves would be synchronized by a centrally generated clock signal). Programs could then fork (or spawn) other programs that would act independently and perhaps join (rendezvous with) the original process at some later time. Independent programs could also send messages to each other. All of this was done by having programs build communication paths through the network. This meant that many of the processors would only be used for passing messages between programs even though all of the processors would be capable of executing complex programs. An interesting result of this is that to optimize communications between programs the programs might actually migrate to different places in the machine.

The Holland machine did not lend itself to efficient use by most programs. It was consequently difficult to program in any way that made use of its unique features. Comfort [18] has proposed a modified Holland machine that is easier

to program, less expensive to construct and provides for more efficient utilization of the hardware. However, because we are primarily interested in SIMD processors here (Holland's and Comfort's machines are MIMD), we will now leave them and move on.

The Orthogonal Computer

In 1960 Shooman [103] proposed an architecture that combined aspects of both conventional SISD and SIMD computers. The basis of this design was an "orthogonal memory". This memory could be accessed in two modes: serial by word and parallel by bit (SWPB) or parallel by word and serial by bit (PWSB). The term "orthogonal" derives from the fact that these two modes each select a slice of bits in the memory but the slices are perpendicular to each other. No specific size for the memory was specified although the possibility was noted that a subset of the memory might not be accessable in both directions.

The machine included two processors: the horizontal unit (SWPB) and the vertical unit (PWSB). The horizontal unit was really just a conventional SISD processor. The vertical unit contained temporary storage and carry registers and a mask register that served as an activity control. A central

control unit managed and coordinated the two processing units.

Although the bit serial feature of the orthogonal memory would make it possible for the computer to perform associative operations it was not actually designed for this purpose. The orthogonal design was intended to provide a machine organization that allowed parallel processing to take place whenever a serial algorithm could take advantage of it. Thus it would be used in applications that are, for the most part, serial but which have some array or search operations that could be done in parallel. This design eventually led to the OMEN computer series which will be covered later.

SOLOMON I

Slotnick, et al., [2, 55, 1Ø9] presented the design for the Simultaneous Operation Linked Ordinal MOdular Network (SOLOMON) in 1962. This was the first of the series of computers that led to the ILLIAC IV parallel processor. Its purpose was to do matrix manipulation in complex problems that involved a high degree of parallelism. The design was correspondingly complex.

The overall design of SOLOMON consists of six units: program memory, control unit, central control sequencer, input-output sequencer, PE matrix and input-output equipment.

The control unit is the only unit which accesses program memory. The purpose of the control unit is to manage and coordinate the operation of the entire SOLOMON system. It can distinguish between four types of instructions: PE matrix commands, control I/O commands, PE matrix I/O commands and program control commands.

The major objective of the sequencer is to provide control signals for commands involving the PE matrix. These commands are fetched by the control unit and passed to the sequencer. The sequencer also provides control signals to the I/O executor during I/O transfers to the PE matrix. The control signals consist of both decoded command signals and memory addressing signals. The memory address control is provided by two digit counters and two matrix switches.

The I/O executor is used to control peripheral equipment I/O to and from the PE matrix. This is primarily for high speed transfer between the matrix and auxiliary storage as a means of manipulating arrays larger than the matrix. Devices that were to provide the auxilliary storage included

multichannel magnetic tape and a high speed magnetic drum. The PE matrix can also communicate with a secondary I/O system through an I/O buffer. The secondary I/O system includes interface and data format conversion hardware to which devices such as magnetic tape units, card readers and punches, printers or paper tape units could be connected. Both the primary I/O system and the secondary I/O buffer can communicate with the PE matrix in two ways. One of these is by transferring data to and from PEs at one edge of the matrix. The design called for the ability to communicate via any of the four edges of the matrix. The other means of communication was by a method called "geometric control". This allowed rows or columns of elements within the matrix to connect directly to the outside. The purpose of this was to eliminate multiple transfers required to move information from the edges into the center of the array.

The PE matrix consists of a 32 by 32 array of processing elements. Each PE contained 4096 bits of magnetic core storage and an arithmetic unit capable of bit serial arithmetic and logical operations. Each PE was also able to communicate bit serially with its North, South, East and West nearest neighbors in the matrix. A fifth source of input to each PE was a broadcast bit that could be transmitted to all of the elements by the control unit. The

memory in each PE is divided into two "frames". Each of these frames is made up of sixty four 32-bit words, physically organized as a stack. Memory access is bit serial with a special bit selection register providing the control signals. Each of the two frames has its own bit selection register and thus they are capable of accessing different bit slices in parallel. Placing an initial offset value into the bit selection register (which is also a counter) also provides a mechanism for implementing variable length words. Frame one data can be loaded to the ALU or transmitted to neighboring cells. Frame two data can only be routed to the full adder in the ALU. Storage into the memory can take three forms: the sum from the full adder can be written into frame one with the frame two operand rewritten to frame two, the sum can be stored into frame two with the frame one operand rewritten into frame one, or information can be exchanged between frames. Each PE also contains a processing mode register. Any command issued by the central control will only be executed by processing elements that are in a mode that matches the mode control signals broadcast by the central control. The control unit may activate any combination of four states permitting a command to be executed by individual elements in different states.

Although its bit serial nature made it possible for SOLOMON to perform associative operations, there is no mention that it was ever intended to be used in this way. The lack of SOME/NONE response reporting (and also of response count) would have hampered many important associative algorithms.

SOLOMON II

The SOLOMON design was revised in 1964 to enhance its speed and reduce complexity. The major change was fixing the word length at 24 bits and providing 24 bit fixed point bit parallel arithmetic and boolean logic operations. Processing element memory was consolidated into a single random access store instead of the two stack frames. Two 24-bit registers (called P and Q) and a one bit carry register were added to each of the processing elements.

ILLIAC III

Although ILLIAC III (the Illinois Pattern Recognition Computer) [78] comes numerically between SOLOMON II and ILLIAC IV in the family tree, it bears only marginal resemblance to either of these machines. ILLIAC III was

actually a special purpose parallel processor designed to analyze bubble chamber photographs. It could, however, be used to process almost any two dimensional digitized data field. Operations that it was intended to support included line thinning, gap filling in lines, identification of line end points, determination of bends in lines and determination of points of intersection on lines.

ILLIAC III had five major components. These were the oscilloscope scanners, output complex, arithmetic unit, taxocrinic unit and the pattern articulation unit. The scanners provided rapid entry of digitized information. The output complex connected ILLIAC III to the outside world. The arithmetic unit performed statistical and stereoreconstruction computations. The taxocrinic unit was to organize abstract graph output in list form. The pattern articulation unit (PAU) was a 32 by 32 array of processing elements which reduced the input to a segmented image. The PAU is the unit of primary interest in this discussion.

The processing elements of the PAU are arranged in a square grid and can communicate with either their nearest eight neighbors (N,S,E,W,NE,SE,NW,SW) or with their six nearest neighbors (rhombic interconnection). Either of the two connection systems may be selected. Each PE contains two one bit registers and I/O control logic. Associated

with the PAU is a unit called the Transfer Memory (TM). The TM is arranged as a 32 by 32 array of 54 bit words. The PE array can transfer data to and from any of these 54 bit planes in the TM. The combination of the TM with the PAU is capable of bit serial associative operations, although it is not particularly efficient at performing them. The lack of SOME/NONE and response count also adds to this. One interesting feature of the PE is an operation called "flash through". Depending upon the state of the PE, certain flash through operations will cause the cell to become transparent to the communication grid. Thus it will provide a direct link between neighboring cells on opposite sides of itself. By chaining cells in the flash through mode, ILLIAC III provides a mechanism for long distance communication across the PAU. The PE communication registers (used to hold incoming data from neighbor cells) also have an interesting twist. They are also "bubble registers". That is they are capable of squeezing all one bits to one end of the register and all zeroes to the other end. The one bits can then be easily counted and a threshold applied to the count to select points in a threshold guided segmentation.

ILLIAC IV

The ILLIAC IV [3, 14, 23, 69] was probably one of the best known parallel processing systems ever built. It was even featured twice in the National Geographic Magazine. Both of these articles were devoted to advances in technology. The first was printed shortly after ILLIAC IV was completed. The second was printed only last year and noted that ILLIAC IV was being dismantled because it had become obsolete. The ILLIAC IV design followed from Slotnick's work on SOLOMON. It was originally intended to consist of four quadrants of 64 processors for a total of 256 processors. Owing to a variety of problems, only one of the quadrants was ever built. The architecture was primarily designed at the University of Illinois and the fabrication was mostly performed by the Burroughs Corporation.

The ILLIAC IV had as its major system components a Burroughs 6500 general purpose computer system (serving as a host processor), an I/O controller and associated hardware, and the four array quadrants. The quadrants were each organized as an 8 by 8 square array (N, S, E, W interconnection) with switchable edge wraparound or connection to another quadrant. The quadrants could operate

independently or together as a 16 by 16 element array.

Each quadrant had its own Control Unit (CU) and 64 processing units. Each processing unit contained a Processing Element (PE), Memory Logic Unit (MLU) and Processing Element Memory (PEM). The CU could directly access the PEMs through the MLUs. All instructions were broadcast to the PEs by the CU. Unlike its predecessors, ILLIAC IV was a fully parallel machine. The PEs were capable of full 64 bit parallel floating point arithmetic. The PE instruction set was patterned after large general purpose computers and included a mode register which allowed program direction of control flow within individual PEs.

Each PEM contained 2048 words of 64 bits, with the possibility of expansion to 8192 words. ILLIAC IV was one of the first machines to rely primarily upon semiconductor memory and was actually a significant driving force in the development of that technology. The PEs each had five 64 bit registers: A, B, C, R, S. The A register served as an accumulator. The B register was for holding a second operand and also provided the most direct means of communicating with external data sources. The C register was used by certain instructions to save carries from the adder. The R register was used for communicating with other PE's and could serve as temporary storage. The S register

was used for storage of an operand within the PE. The PE also had an 8 bit mode register which would control some of the operations in the PE as well as store PE faults and test results, and a 16 bit index register (X). Two of the mode register bits were used to protect the A, S and X registers and also the Memory Information Register (MIR). Two more mode bits stored faults (underflow, overflow, etc.) and the remaining four were used as temporary storage for test results.

ILLIAC IV was eventually installed at NASA Ames Research Center. It was used in a variety of applications including Eulerian flow, Lagrangian flow, neutron transport, implicit relaxation solutions to differential equations, weather analysis and prediction, matrix computations, linear programming, Fourier transforms and multichannel filter design. Much of the current day interest in parallel processing and architectures owes its impetus to the tantalizing work done on this machine.

PEPE

The Parallel Element Processing Ensemble (PEPE) [9, 96] originated at Bell Laboratories and was subsequently implemented by the System Development Corporation and

Honeywell Inc. Although PEPE was designed as an associative architecture it has generally been classified with mainstream parallel processing systems. This is due to the powerful processing capabilities of its processing elements. PEPE is called a parallel processing "ensemble" as opposed to an array because there are no inter-PE communication links. PEPE was designed specifically for radar processing applications involving ballistic missile tracking and identification.

PEPE consists of a collection of associatively addressed processing elements which are directed by three different control units. The reason for associatively addressing the processing elements is to reduce the input output bottleneck encountered in many parallel processors. The three PEPE control units are the Correlation Control Unit (CCU), Arithmetic Control Unit (ACU) and the Associative Output Control Unit (AOCU). Each control unit is mirrored by a corresponding processing unit within each PEPE processing element (Correlation Unit (CU), Arithmetic Unit (AU), Associative Output Unit (AOU)).

The CU/CCU combination work to associatively assign input data to one or more processing elements. The CCU treats the CUs as an associative memory. Each new datum received by the CCU is compared against the predicted data

in the CU portion of the PEs. The datum is then input to the CU's which match. The matching can take the form of an exact match or a series of "in limits" matches.

The AU/ACU combination perform the arithmetic processing on data within each PE. Individual processing elements may choose not to respond to AU commands depending upon their internal status. Each AU is a fully parallel 32 bit arithmetic processor. The AU is capable of both integer and floating point operations and includes hardwired algorithms for multiplication, division and square root. An 8 bit content addressable tag register provides associative access to the AUs. Circuitry is also provided for maximum and minimum element searches on the accumulator contents of all of the AUs in the ensemble.

The AOU/AOCU combination is provided as a means of transmitting data from the PEs to the control units without interrupting processing in the AUs. The AOU contains a content addressable tag register, activity stack and maximum/minimum element search hardware like that in the AU.

In addition to the three elements mentioned above, each PEPE processing element contains a 1024 word memory (32 bits per word) which is called the Element Memory (EM). The EM is accessible to the three processing unis (CU, AU, AOU). These

units share the memory on a cycle stealing basis with conflict resolution taking place in the control units. The control units were also provided with their own data and program memories. The CCU and ACOU each have 2048 words of program memory and 2048 words of data memory. The ACU has 32K of program memory and 3K of data memory.

One of the interesting features of PEPE is the amount of autonomy given to each of the three processing controllers. Each of these has its own program and data memory (called sequential memory), instruction fetch and decoding circuitry. Because the controllers direct processing within independent subunits of the processing elements, each of the three controllers can thus independently execute a stored program. This was done to maximize the design goal of offloading all parallel operations from the host computer.

PEPE also required considerable ingenuity in fabrication. A 300 element system requires 46 KW of power (9000 Amps) due to the extensive use of Emitter Coupled Logic (ECL) for high speed. The result was a sophisticated water cooled packaging design utilizing multichip carriers. PEPE was eventually constructed in a 288 element configuration with a CDC-7600 acting as the host computer.

OMEN

OMEN [117] is actually a series of computers developed by the Sanders Corporation for high speed signal processing applications. The design is based on Shooman's Orthogonal Computer concept. The typical OMEN system consists of a DEC PDP-11 minicomputer (which acts both as a controller and host), an Orthogonal Memory (OM) and a Vertical Arithmetic Unit (VAU). The differences between the four different OMEN processors in the series are basically defined by the options selected for the Vertical Arithmetic Unit. The OMEN was developed without government funding, using off the shelf components -- a rarity in parallel processors.

The OM, PDP-11 CPU and VAU communicate via the DEC UNIBUS. To the PDP-11 the OM simply appears as a segment of memory from 8K to 128K bytes long. Access to the OM is controlled by a unit called the "scoreboard" -- patterned after the CDC-6600 scoreboard but much simpler. It keeps track of activity in the various units and coordinates them and their accesses to the OM. OMEN uses a three level programming structure consisting of instructions stored on a stack, vertical (16 bit wide) microinstructions which implement the stack instructions and horizontal "nanoinstructions" (80 bits wide) which implement the microinstructions.

The OMEN VAU contains 64 processing elements. Thus the OM is organized as a 64 bit by 16K bit array. In the long dimension, however, direct bit slice access is not provided. The PDP-11 byte slice accessing is used for this and thus a bit slice operation requires fetching a full byte and masking off unnecessary bits. Each PE in the VAU consists of three major parts: the skew logic, register file and arithmetic logic unit. The skew logic is the same for all of the OMEN series processors. It provides pair interchange, perfect shuffle and barrel shift connections between PEs. The main difference between the four OMEN series processors is the size of the register file and the ALU. Register file sizes can be 8, 37 or 133 bits per PE. Either of two ALUs may be chosen. One is bit serial while the other is fully bit parallel and provides hardware floating point arithmetic.

A typical OMEN signal processing configuration is envisioned as a combination of an OMEN61 (37 register file bits, serial ALU) and an OMEN64 (133 register file bits, parallel ALU) with the two OMs connected via a 64 bit wide Parallel Memory to Memory Bus (PMMB). The smaller OMEN would then perform I/O, display control and data formatting operations for the OMEN64. Although FORTRAN and BASIC are available for programming OMEN (through the use of macro

libraries), the primary language developed for the processor is a compiling version of APL-360.

CLIP 3

This machine was built by the Image Processing Group of the Department of Physics and Astronomy at University College, London, in 1973 [27]. The Cellular Logic Image Processor (CLIP) was built to process binary images (pixels having values of 1 or 0). The overall design of the processor included a serial control unit with 256 words of 24 bits for program storage and an array of 192 (16 by 12) processing elements.

Each of the processing elements in CLIP 3 contained two I/O registers, neighbor communication logic, a function generator and 16 bits of memory. Both six and eight neighbor communication networks were built into the array and either could be selected under program control. Thus CLIP 3 provided an ideal base for comparing the effectiveness of hexagonal grids versus eight way (N, S, E, W, NE, SE, SW, NW) square grids. incoming information from neighboring cells is automatically summed and thresholded according to globally broadcast signals which determine the set of neighbors to be included in the summation and the

threshold level. The output of the threshold circuit is ORed with the value in one of the two I/O registers (the B register) and becomes one of the two inputs to the function generator. The second input to the function generator is the value of the other I/O register (the A register). The function generator provides two outputs, called N and D. The N output is transmitted to all of the neighboring cells. The D output goes to one of the 16 local memory bits, any of which can then in turn be loaded into the A or B registers. The A and B registers of all of the cells are also connected to a pair of oscilloscopes that display the contents of the array. Neighbor inputs from off the edges of the array to cells on the edges can be set to either all ones or all zeroes.

The machine operated at a basic cycle time of 30 nanoseconds for operations on data within cells. Operations that required data to be transmitted between neighboring cells required 60 ns. Maximum propagation time across the entire array was 3.12 microseconds. Although CLIP 3 was very limited in processing power, it was also very inexpensive in comparison to most parallel processor systems. Total cost for the machine was on the order of $10,000. CLIP 3 provided its designers with considerable experience which they then directed into the design of its

successor machine.

CLIP 4

This machine [120] is the successor to CLIP 3. It is a considerably enlarged design, containing 9216 (96 by 96) processing elements, each with 32 bits of memory. The neighbor interconnection network of CLIP 4 is eight way (N, S, E, W, NE, NW, SE, SW) only, the hexagonal connection scheme having been dropped. (Rectangular pixels are more easily generated with existing video equipment.) Each processing element has the same compliment of registers (A, B, C and the D storage element) as CLIP 3. The neighbor value summing logic has been removed, however, with neighbor inputs now individually gated (any subset of the eight inputs) and ORed together (possibly with the output of the C register) to form one of the inputs to the function generator. The A register is the other input to the function unit. (The B register can be combined with the neighbor input via a NOR-NOR circuit.)

CLIP 4 is still basically designed to work with binary (one/zero) valued pixels. Five common modes of operation are described for the machine. "Simple Boolean Operation" puts the bits of one image in the A registers, the bits of

another image in the B register and stores some function of these two in one of the 32 D bits. "Simple Propagating Operations" take the OR of some subset of the neighbor values and apply a function to this and the contents of the A register, storing into a D bit. "Labelled Propagating Operations" performs propagation for those cells with the value one on their B registers. "Bit Plane Arithmetic Operations" provide for arithmetic between images with gray level pixels. In this mode the two images are stored in bit slices of the D store and are then fetched up a bit at a time. The C bit provides for storage of carry information from bit to bit. "Binary Column Arithmetic Operations" provide for manipulating images stored in a different manner. In this mode, the bits representing an image are stored in multiple processing elements (one bit per element) within a column of 96. Carry operations then take place automatically as a neighbor to neighbor propagation.

Data is loaded into CLIP 4 through a set of shift registers. An image is first shifted into a 9216 bit register. This then breaks into 96 separate registers of 96 bits. Each of these is then connected to one row of the array and discharges its contents into the array by shift and propagate operations. The machine also has an adder tree which provides a rapid count of the number of cells

with their A registers set to one. No some/none signal is provided however. The basic cycle time of CLIP 4 is 400 nanoseconds. The average instruction execution time is 9 microseconds plus 1.2 microseconds per cell for propagation beyond the immediate neighborhood. Data entry time is 4 milliseconds per bit plane (although synchronization with video equipment increases this figure to 10 milliseconds).

Although CLIP 4 can be used associatively, it is not really designed for this. The lack of some/none and of an activity control mechanism for each PE that is separate from data values makes certain associative operations difficult to implement. The most interesting feature of CLIP 4 (and of CLIP 3) is the dual output function generator. This allows the PE to compute one value for itself while computing a totally different value to be presented to its neighbors. For example, a PE could compute and store a value based on some other value that is passing through it as part of a long distance propagation. Thus the propagate and compute operations require only one instruction time instead of two. CLIP 4 has been in operation since 1979 and is also now commercially available.

Massively Parallel Processor

This machine (known as MPP) [5] was contracted by NASA Goddard Space Flight Center to Goodyear Aerospace in 1979 and became operational in mid 1983. The MPP is designed to provide high speed processing of satellite imagery. It is currently the largest parallel processing system in the world, consisting of 16,896 processing elements. The PEs are arranged in a 128 by 132 rectangular grid. Only a 128 by 128 element square is used out of this, with the extra four columns provided to enhance hardware fault tolerance. Each processing element can communicate with four of its neighbors (N, S, E, W). Three treatments are provided for each pair of array edges. They can be stitched together so that the array forms a cylinder (or torus), stitched so that the rows and/or columns form a single long linear array or they can be left unconnected. This is controlled by a topology control register in the array control unit. Data transfer to and from the array is mediated by a staging memory. The staging memory can also be used to rearrange array data to facilitate Fast Fourier Transform (FFT) operations.

Each MPP processing element is a bit serial processor with six 1 bit registers (A, B, C, G, P, and S), a shift

register with programmable length, a RAM, data bus, full adder and some additional combinational logic. The A, B, and C registers, the full adder and the shift register are used for bit serial arithmetic operations. The G register acts to mask operations within the PE as an activity control. The P register is used for logic and routing operations. The S register is used for input and output of array data. The data bus can select its source from B, C, P, S, a RAM bit or the equivalence function applied to P and G.

Bit serial arithmetic on the MPP is mainly based on additions of operands passed through the full adder. The A and P registers are summed with the carry maintained in the C register. The A register can draw the bits of its operand either from the shift register or the RAM. The P register can take its operand bits from the RAM or the P register of any neighboring processing element. The result of the addition can be sent to either the shift register or the RAM. Other arithmetic operations are performed with the usual adder-based algorithms. The only other hardware provided for this purpose is an inverter which allows the complement of one of the operands to be loaded directly into the P register.

The data bus of the PE can also be fed into a tree of OR

gates to provide a Some/None type indication to the control unit. There is no special hardware for counting the number of PE's with a particular register set to one. Each PE has 1024 bits of RAM, expandable to 65,536 bits. This large storage is achieved by having the memory reside on chips separate from the special processing element chips. This means that standard RAM chips can be used for the memory of the MPP processing elements. Because there are only eight processing elements on each of the PE chips, only a small overhead in pinouts is incurred by the off-chip memory. The PE chips have 52 pins and a complexity of about 8000 transistors. There are 2112 such chips in the array, occupying some 88 printed circuit boards.

Input and output to the array (from the staging memory) takes place through the S registers in all of the PEs. Columns of data are shifted into the S registers starting from the West edge of the array. Once all of the S registers have been filled with a bit plane, they are transferred to a bit plane of the RAM. The operation is repeated for each bit plane in an image. This provides a transfer rate of 160 million bytes per second.

Control for the MPP is divided into four parts. One of these is a host computer. In the case of the prototype this is a DEC VAX 11/780 mini-mainframe. Another part of the

controller is the PE Control Unit (PECU). This controls the processing in the array unit processing elements. The third part is the I/O Control Unit (IOCU) which manages data transfers to and from the array and also controls the staging memory. The last part is the Main Control Unit (MCU) which acts as an interface between the host and the other two units. The MCU has its own program and data memory and supervises the overall MPP operation. It performs all scalar arithmetic itself and queues all array processing into the PECU. Once the MCU initiates an operation on the PECU or IOCU those controllers can operate independently of each other. The overall result of this division of control is that the MPP can perform scalar, array and I/O operations concurrently.

The I/O staging memory actually consists of three banks of memory: the main stager, input substager and output substager. The main stager can have 4, 8, 16 or 32 banks of 16K, 64K or 256K words of 64 bits (plus 8 parity bits). The substagers are fast 128 bit by 1024 bit ECL multidimensional access memories. These are very similar in design to the STARAN flip network.

The MPP has a basic cycle time of 100 nanoseconds. It can thus sum two arrays of 16,384 sixteen bit values in about 5 microseconds. This provides an effective throughput

of 3 billion fixed point operations per second. Floating point multiplication is performed at an effective rate of 200 million floating point operations per second (MFLOPS).

MIT Connection Machine

This is perhaps best described as a modernized Holland Machine (see above). It is an attempt to develop a machine that will allow semantic network type operations to be carried out on a network of asynchronously communicating processors. At the time of this writing no detailed information is available concerning the actual hardware. The only paper available [59] is dated 1981 and is basically a description of the concept.

The overall description of the Connection Machine is that it will consist of a million processing elements (although the prototype may be only 128K elements) connected by some form of twisted torus or grid. Each processing element will contain a "a few registers, a state vector and a rule table". Each cell is essentially a Finite State Automaton (FSA), taking inputs and current state information which are passed through the rule table to provide a new state and some outputs. The rule tables are shared among multiple cells, although the method for doing this is not

described. Because the virtual connection scheme of the network is a LISP tree, each cell has three connection registers to hold the addresses of the cells it will connect to. There are also "two or three extra registers for storage of addresses and numbers" in each cell. Because each address in a million cell machine is 20 bits long, each processor will have a total of about 150 bits of storage. (Status requires "ten to fifty bits".) Note that the cell design inherently limits the maximum size of the machine because of the size of the address registers.

Operations on the Connection Machine take the form of messages passed between connected cells. Unconnected cells become connected through a mechanism called "message waves". A cell that needs to establish a connection to another cell will start a propagating wave message that will expand out from it, eventually reaching all cells. The message includes the return address of the initial cell. When the message reaches a cell with the desired characteristics, that cell starts another message propagating back along the return path. The return message carries the address of the sought-after cell back to the original cell. At this point the two cells have each other's addresses and a communication path is established. The original cell then starts a second wave message

propagating to catch up with and cancel the original wave to minimize activity in the network. The cancellation wave travels at twice the speed of the original wave and thus eventually catches up. (Note that this really means that the first message must always travel at half speed through the network.) Additional wave messages may be required if multiple cells respond to the original wave, in order to provide a handshaking operation to eliminate spurious communication links. The designers also claim that the Connection Machine is very fault tolerant because only non-defective cells will ever be linked to. They do not, however, indicate how cells are identified as defective. Nor do they explore the implications of defective cells within the context of the hardware topology. No processing speed estimates have been given for the machine.

Other Machines

The preceeding set of machine descriptions is by no means a complete list of parallel processors. It is simply meant to give the reader an idea of some of the types of machines that have been designed. The focus is on machines that are significantly related to the work presented in this dissertation. Other machines, also of interest, include:

the Stanford Wafer Scale Vision Processor Project [74], Kruse's Parallel Picture Processing Machine [67], Reeves' Binary Array Processor [93, 94], Howard and Leah Siegel's PASM [104], the CRAY-1 computer system [22, 97], the LUCAS associative processor [35, 116] at the University of Lund in Sweden, the Control Data Star-100 [19] and Cyber 205 computers, and the Floating Point Systems AP-120B and 164 family of processors [16].

Summary

We have looked at a wide variety of parallel processing systems. These vary from unconnected ensembles of processing elements to such odd arrangements as a multiply twisted torus although the most common is a rectangular array connection network to allow communication with N, S, E, W neighbors. Perhaps the characteristic most commonly shared among these designs is the use of a host computer and separate array control processor. This stems from the recognition that the power of a parallel processor is wasted on scalar operations and that speed gains due to parallelism can become quite negligible if these are not performed on a separate host system or scalar processing element. We can thus see that parallel processors are best applied to

problems in which a host can offload sufficient parallel processing tasks to keep the processing array busy. Parallel processors also suffer from the same I/O problems as associative processors: efficiency drops dramatically when data must be loaded and unloaded with few processing operations inbetween. Just as with associative processors then, parallel processors work best when data can be loaded that require many computations to be performed before output is necessary.

Also noticeable in the above discussion is a trend toward larger and larger processing arrays. Early machines had only a few dozen processing elements. The current record is 16,384 with million element machines being proposed. As the machines grow larger a variety of problems arise. One of these is increasing the I/O data transfer rate so that the processing array does not become I/O bound. Another problem is fault tolerance and reliability. It will be difficult to keep 1 million processors of any complexity all operational for any length of time. Yet another problem is packaging and intercommunication between cells. The more cells of an array that are put on a chip, the more communication lines are required to connect the chip to the rest of the array. Whenever communication lines are added to a chip design the circuit area for the chip

must be drastically reduced. (This is due to the relatively large size of wiring pads as compared to logic elements on a chip.) One proposed solution to this problem is to build wafer sized chips that would allow the entire array to reside on one piece of silicon. Although wafer scale integration will eventually come about in one form or another, it will be several years before the problems (heat dissipation, packaging, thermal expansion, yield, fault tolerance, etc.) are solved.

For further information on VLSI and computer architecture in general, readers are advised to consult any of the following references: Foster[37], Stone[115], Mano[75], Whitesitt[127], Nagel[83], Klir[65], Hwang[63], Bell[7, 8], Kung[71], Mead[79], Conway[20], and Goos[53], Chiang [17], Deutsch [25], Ermann [30], Etchells [33], Golay [51], Higbie [58], Hobbs [60], Kuck [70], Love [73], Matney [76], Montalvo [82], Nassimi [86], Siegel [106], Stevenson [113], Thurber [117, 118], Winograd [128].