

**An Integrated Computing Structure
for Pixel-Parallel Image Processing**

by

Jeffrey Carl Gealow

Bachelor of Science in Electrical Engineering
Massachusetts Institute of Technology, June 1989

Master of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, June 1990

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 1997

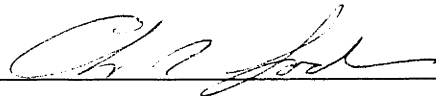
© 1997 Massachusetts Institute of Technology. All rights reserved.

Signature of Author _____



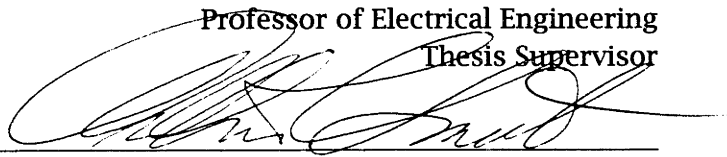
Department of Electrical Engineering and Computer Science
16 May 1997

Certified by _____



Charles G. Sodini, Ph.D.
Professor of Electrical Engineering
Thesis Supervisor

Accepted by _____



Arthur Clarke Smith, Ph.D.
Professor of Electrical Engineering
Graduate Officer

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 24 1997

ARCHIVES

LIBRARIES

An Integrated Computing Structure for Pixel-Parallel Image Processing

by

Jeffrey Carl Gealow

Submitted to the Department of Electrical Engineering and Computer Science
on 16 May 1997 in partial fulfillment of the requirements for the degree of
Doctor of Science in Electrical Engineering and Computer Science

Abstract

A new integrated circuit architecture provides a basis for microcomputer systems capable of performing low-level image processing tasks in real time. Compact logic units are pitch-matched to DRAM columns to form dense blocks of processing elements. The processing elements are interconnected to form a rectangular array, with each processing element assigned to a single pixel.

A prototype integrated circuit provides a 64×64 array. Efficient logic circuits minimize power dissipation and area per processing element. Interface circuits extend the processing element interconnection network across chip boundaries so that multiple devices may be used together to form processing element arrays matching the size of large images. Serial-access memories provide a fast and efficient means of transferring image data to and from the device.

A demonstration system employs four prototype devices to form a 128×128 processing element array. The system is managed by a desktop computer. The control path provides high processing element array utilization without complex control hardware. The data path reorders image data in real time so that the processing element array may be used with standard imagers. A programming framework implemented using the C++ programming language separates application development from device and system details.

Operating in the demonstration system with a 60-ns clock cycle, fully functional devices dissipate 300 mW. Several low-level image processing tasks have been implemented including median filtering, smoothing and segmentation, and optical flow computation. All have been successfully performed in real time at rates exceeding thirty frames per second. Careful analyses show that the prototype devices require far less power per pixel than microprocessor and digital signal processor devices.

Thesis Supervisor: Charles G. Sodini
Title: Professor of Electrical Engineering

Acknowledgements

I am very grateful to the many people who have contributed to my doctoral studies. I hope that they will share in the joy and pride I feel as I finish the work.

I thank my thesis supervisor, Charles Sodini, for his guidance, support, and encouragement. I have benefited greatly from his technical expertise and from the expertise of others in his research group. I thank my thesis readers, Anant Agarwal and Jacob White, for their interest in my work and for their many helpful suggestions. My work was aided by my association with the Vision Chip Project, led by John Wyatt, and by my frequent interaction with Ichiro Masaki, who graciously shared his application expertise. I am grateful to Hae-Seung Lee for his thoughtful advice.

I was very fortunate to have Frederick Herrmann as one of my colleagues. I enjoyed working with him to develop our system architecture and software framework and I learned much from our technical discussions. I will always be grateful to him for introducing me to awk.

I thank Steven Decker, Jennifer Lloyd, and Joseph Lutsky for their frequent assistance and for their companionship.

Gary Hall designed and built the controller for our demonstration system. He and Michael Perrott provided invaluable assistance in the final stages of the chip design. Daphne Shih developed the format converter board used for image I/O. Zubair Talib is developing a second-generation processing element array board; I look forward to seeing a demonstration.

I am grateful to Myron Freeman and Thomas Lohman for all they have done to maintain and improve the computer resources on which I relied. I thank Patricia Varley for her gracious help with many part and supply orders.

I thank my parents, Jon and Rita Gealow, for their advice, support, and assistance. Most of all I thank my wife, Mary Gealow, for her unfailing confidence in me, for her prayers, for her patience, and for all the help she has provided and all the sacrifices she has made.

This work was supported by grants from the International Business Machines Corporation, the Defense Advanced Research Projects Agency (contract MIP-9117724), and the National Science Foundation (contract MIP-9423221) and by a National Science Foundation Graduate Fellowship.

Contents

List of Figures	9
List of Tables	12
Transistor Notations	15
1 Introduction	17
1.1 Image Processing System	17
1.2 Processing Element Implementation	18
1.3 Processing Element Operation	19
1.4 Previous Work	22
1.5 Thesis Organization	23
2 Integrated Processing Element Array	25
2.1 Design Overview	25
2.2 Twin Cell Dynamic Memory	27
2.3 Rows	30
2.3.1 Wordlines	30
2.3.2 Wordline Driver	32
2.3.3 Address Decode Circuits	32
2.4 Columns	34
2.4.1 Sense Amplifier	34
2.4.2 Write Driver and Latch A	38
2.4.3 Interconnect Logic and Function Generator	38
2.4.4 Latches B, C, D, and E	40
2.4.5 Write Logic	40
2.5 Timing	40
2.6 Processing Element Interconnection	44
2.7 Interchip Communication	44
2.8 Image Input/Output	46
2.9 Perimeter Circuits	48
2.9.1 Line Driver	48
2.9.2 Pad Circuit	50
2.10 Design Process	54
2.11 Summary	55

3	System Design	59
3.1	Data Path	59
3.1.1	Image Organization	59
3.1.2	Corner-Turning	60
3.1.3	Design	62
3.2	Control Path	63
3.2.1	Control Strategy	64
3.2.2	Instruction Selection	67
3.3	Programming Framework	69
3.4	Summary	71
4	Experimental Results	73
4.1	System Construction	73
4.2	Control Path Characterization	74
4.3	Integrated Circuit Testing and Characterization	76
4.3.1	Preliminary Functional Verification	76
4.3.2	Hold Time Characterization	78
4.3.3	Performance Characterization	80
4.4	System Performance	81
4.4.1	Median Filtering	81
4.4.2	Smoothing and Segmentation	84
4.4.3	Optical Flow	86
4.5	Summary	86
5	Architectural Alternatives	87
5.1	Simplified Processing Element	87
5.2	Comparison Methodology	90
5.3	Bit-Parallel Processing Element	90
5.4	Field-Programmable Gate Array	92
5.5	Microprocessor	95
5.6	Digital Signal Processor	97
5.7	Summary	98
6	Conclusion	101
6.1	Summary	101
6.2	Future Work	102
	References	105
A	Chip Data	111
A.1	Supplies and Signals	111
A.2	Pads	115
A.3	Packaging	117

CONTENTS	9
B Layout Design Rules	121
C Array Code	125
D Processing Element Timing Control	127

List of Figures

1.1	Pixel-parallel image processing system	18
1.2	Logic pitch-matched to dynamic memory	19
1.3	Processing element	20
1.4	Instruction format	20
2.1	Chip organization	26
2.2	Twin cell dynamic memory	27
2.3	Twin cell schematic	28
2.4	Twin cell layout	29
2.5	Twin cell cross section	29
2.6	Wordline driver	33
2.7	Address predecoder	33
2.8	Address decoder	34
2.9	Sense amplifier, write driver, and latch A	35
2.10	Processing element interconnect logic and function generator	39
2.11	Latches B, C, D, and E and write logic	41
2.12	Processing element timing	42
2.13	Wordline detector	43
2.14	Interconnecting processing elements to form a square array	45
2.15	Interchip communication multiplexer	46
2.16	Interchip communication control	47
2.17	Serial-access memories	49
2.18	Serial-access memory cell	50
2.19	Low-crosstalk line driver circuit	51
2.20	Line driver simulation	51
2.21	Input/output pad circuit	53
2.22	Chip photomicrograph	57
3.1	Image organization	60
3.2	Multidimensional access memory	61
3.3	Shuffler	62
3.4	Format converters	63
3.5	Controller	66

3.6	Instruction selection	68
3.7	Programming framework	70
4.1	Test and demonstration system	74
4.2	Four-chip processing element array board	75
4.3	Control path performance	77
4.4	Median filtering, smoothing, and smoothing and segmentation	82
4.5	Simplified smoothing and segmentation code	85
5.1	Simplified processing element	88
5.2	Characteristics of architectures	100
A.1	Clock timing	113
A.2	Pad locations	115
A.3	Die orientation	117
A.4	Bonding diagram	118
A.5	Pin assignments	119
D.1	Processing element timing control, part 1	128
D.2	Processing element timing control, part 2	129
D.3	Processing element timing control, part 3	129
D.4	Processing element timing control, part 4	129

List of Tables

1.1	Neighbor difference procedure	21
1.2	Processing element array performance	22
2.1	Twin cell electrical parameters	30
2.2	Polysilicon wordline rise time	31
2.3	Metal wordline rise time	31
2.4	Properties of horizontal processing element interconnection lines	45
2.5	Layout area per processing element	55
2.6	Estimated power dissipation	56
4.1	Hold time characterization results	79
4.2	Calculated cell leakage data	80
4.3	Chip characteristics	81
4.4	Application performance	83
5.1	Sum procedure	89
5.2	Sum procedure for simplified processing element	89
5.3	Comparison of image processing devices, 3×3 median filtering	93
5.4	Comparison of image processing devices, Sobel filtering	94
5.5	Comparison of image processing devices, Laplacian convolution	95
5.6	Comparison of image processing devices, 5×5 convolution	96
5.7	Comparison of image processing devices, 3×3 separable convolution	99
5.8	Comparison of image processing devices, 3×3 convolution	99
A.1	Operation codes	113
A.2	Function generator truth table	114
A.3	Operating conditions	114
A.4	Pad assignments	116
C.1	Neighbor difference code	126

Transistor Notations

A notation of the form $n(w/l)$ indicates a set of n transistors, each with drawn channel width w and drawn channel length l , that have common drain, gate, source, and bulk connections. A notation of the form $n(w)$ indicates a set of transistors with drawn channel length $0.6 \mu\text{m}$.

A notation of the form w/l indicates a single transistor with drawn channel width w and drawn channel length l . A notation of the form w indicates single transistor with drawn channel length $0.6 \mu\text{m}$.

Chapter 1

Introduction

Typical low-level image processing tasks require thousands of operations per pixel for each input image. Traditional general-purpose computers are not capable of performing such tasks in real time. Yet important features of traditional computers are not exploited by low-level image processing tasks. Since identical operations are performed for each pixel, the flexibility of sequential processing is of little value. Since storage requirements are limited to a small number of low-precision integer values per pixel, large hierarchical memory systems are not necessary.

The mismatch between the demands of low-level image processing tasks and the characteristics of conventional computers motivates investigation of alternative architectures. The structure of the tasks suggests employing an array of processing elements, one per pixel, sharing instructions issued by a single controller.

To build pixel-parallel image processing hardware for microcomputer systems, large processing element arrays must be produced at low cost. Integrated circuit designers have had tremendous success creating dense and inexpensive semiconductor memories. They handcraft circuits to perform essential functions using very little silicon area, then replicate the circuits to form large memory arrays. This thesis shows how the same technique may be applied to create a dense integrated processing element array.

1.1 Image Processing System

The integrated processing element array provides the foundation for a desktop demonstration system. Figure 1.1 shows the structure of the system. Each processing element (PE) includes the memory and logic necessary to perform operations for a single pixel. A two-dimensional network connects the processing elements.

The host computer governs the processing element array through the control path. The processing elements receive instructions issued by the controller. The instructions are delivered through a high-bandwidth point-to-point channel. The controller is managed by the host computer over the computer's backplane bus.

Analog signals from an imager such as a video camera are converted to digital data

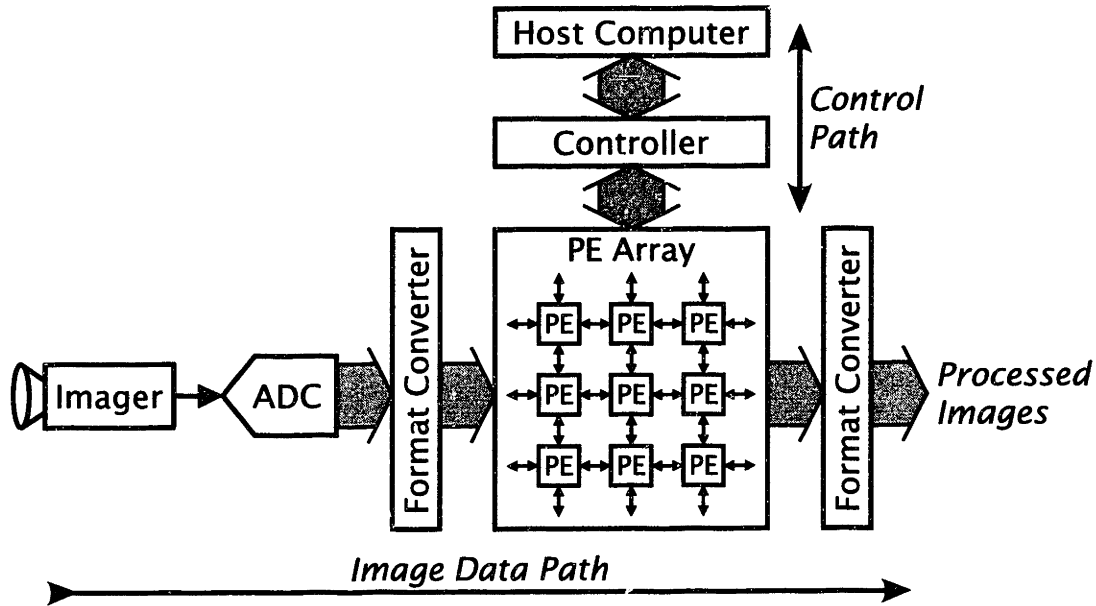


Figure 1.1: Pixel-parallel image processing system.

by an analog-to-digital converter (ADC). A format converter reorders the image data for efficient transfer to the processing element array. Another format converter may be used to reorder output data for subsequent use.

The system is designed for a large class of low-level image processing tasks. Tasks in this class share three characteristics: they require uniform processing of all pixels, they can be performed by processing elements sharing common instructions, and they produce output values that directly correspond to individual pixels. Examples include median filtering, image convolution, pixel-level optical flow computation, and template matching.

1.2 Processing Element Implementation

Processing elements are implemented using logic pitch-matched to DRAM cells. As shown in Figure 1.2, logic units are placed above and below 128-bit DRAM columns. The layout pitch of the logic units is exactly double the memory column pitch. A processing element comprises a logic unit and a DRAM column. There are no column decoders—logic circuits are connected directly to the bitlines. The pitch-matched processing element implementation maximizes the bandwidth between memory and logic and minimizes processing element area [1].

Figure 1.3 is a functional representation of the processing element design. One-bit-wide logic is combined with a 128-bit DRAM column. Three latches, A, B, and C, provide inputs to a function generator. Eight control signals, f_{7-0} , select between the 256 three-input Boolean functions. Latch D holds write data. Latch E provides a local write-enable

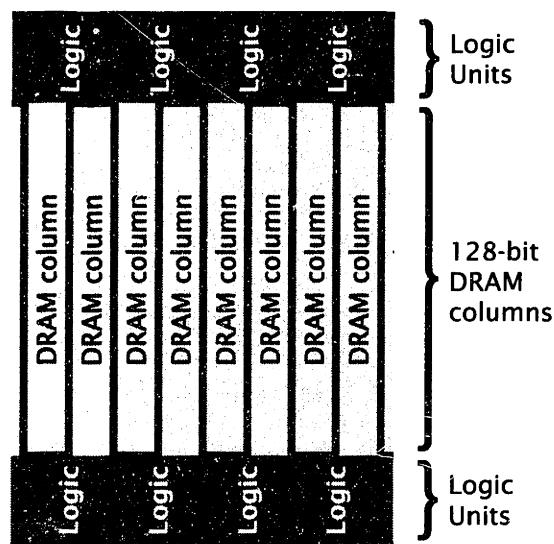


Figure 1.2: Logic pitch-matched to dynamic memory.

signal.

Processing elements are interconnected to create a two-dimensional rectangular array, matching the structure of image data. Latch A controls the output signal for nearest-neighbor communication. Input signals from adjacent processing elements are combined with the function generator result according to control signals f_W , f_E , f_S , and f_N . With the interconnection network extended across chip boundaries, multiple chips may be used to form processing element arrays matching the size of large images.

1.3 Processing Element Operation

Figure 1.4 shows the instruction format for the processing element array. Five bits specify latch load signals, four bits specify the interconnect control signals, eight bits specify a Boolean function and seven bits specify a memory address. The remaining three bits specify an operation code which controls memory and interchip communication activity.

Execution of an instruction begins with application of the function generator and processing element interconnection logic. The result may be loaded into latch B, C, D, or E. Next, a memory operation may be performed. Finally, the result of a memory read operation may be loaded into latch A. Specified interchip communication activity takes place throughout an instruction.

Because processing element data paths are only one bit wide, arithmetic and data movement operations generally require several instructions per bit. A bit-serial "neighbor difference" procedure, $D \leftarrow M - M^{north}$, typifies primitive operations. This procedure computes the difference between a four-bit value in the DRAM column and the corresponding value in the north neighbor's DRAM column. The result is saved in the

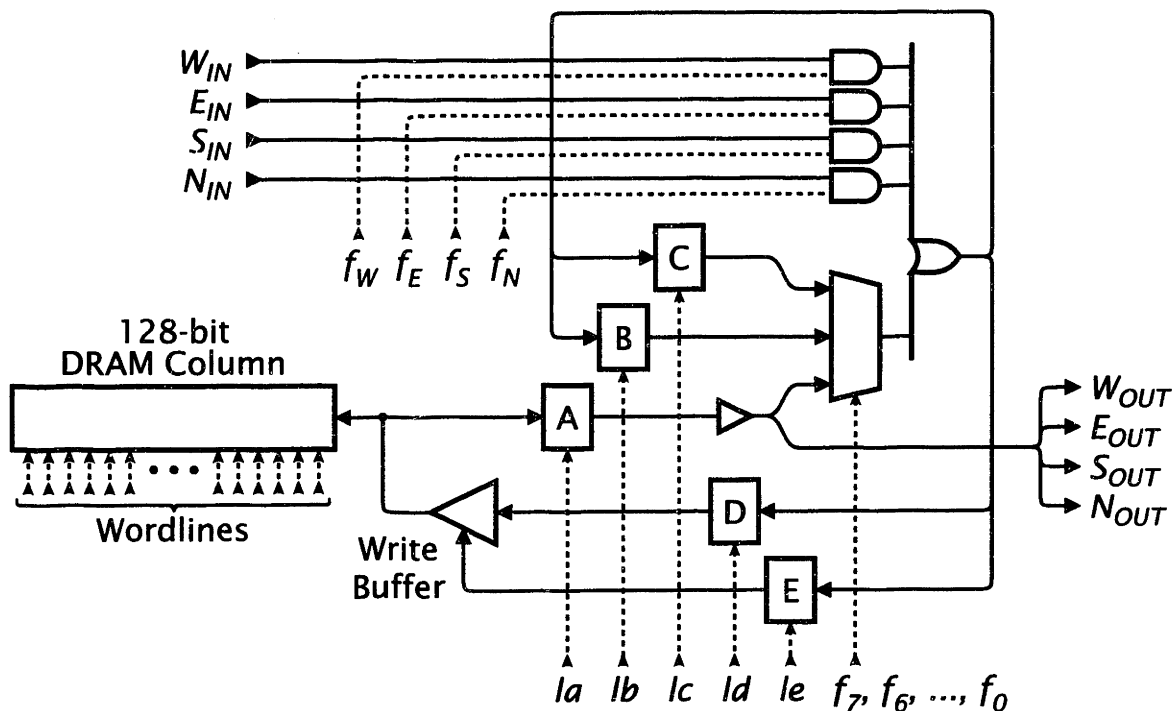


Figure 1.3: Processing element. Solid lines represent data signals. Dashed lines represent control signals.

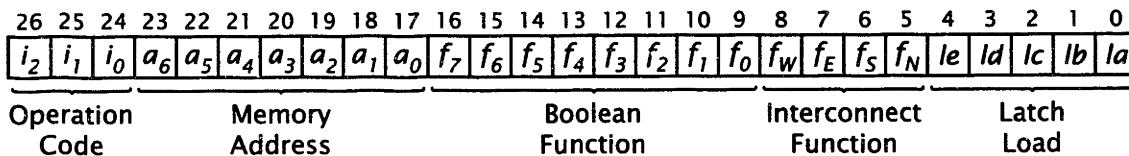


Figure 1.4: Instruction format.

Table 1.1
Neighbor Difference Procedure, $D \leftarrow M - M^{north}$

	Interchip Communication	Logic Activity	Memory Activity
1	—	$E \leftarrow 1$	$A \leftarrow Mem[M_0]$
2	north \Rightarrow south	—	—
3	—	$B \leftarrow A^{north}$	—
4	—	$D \leftarrow A \oplus B$	$Mem[D_0] \leftarrow D$
5	—	$C \leftarrow \bar{A} \wedge B$	$A \leftarrow Mem[M_1]$
6	north \Rightarrow south	—	—
7	—	$B \leftarrow A^{north}$	—
8	—	$D \leftarrow A \oplus B \oplus C$	$Mem[D_1] \leftarrow D$
9	—	$C \leftarrow (\bar{A} \wedge B) \vee (\bar{A} \wedge C) \vee (B \wedge C)$	$A \leftarrow Mem[M_2]$
10	north \Rightarrow south	—	—
11	—	$B \leftarrow A^{north}$	—
12	—	$D \leftarrow A \oplus B \oplus C$	$Mem[D_2] \leftarrow D$
13	—	$C \leftarrow (\bar{A} \wedge B) \vee (\bar{A} \wedge C) \vee (B \wedge C)$	$A \leftarrow Mem[M_3]$
14	north \Rightarrow south	—	—
15	—	$B \leftarrow A^{north}$	—
16	—	$D \leftarrow A \oplus B \oplus C$	$Mem[D_3] \leftarrow D$

local DRAM column. Table 1.1 presents the procedure.¹

Instruction 1 loads latch E, enabling the write buffer, and loads the least significant bit of the operand, M_0 , from memory into latch A. Instruction 2 transfers latch A values from processing elements on the south edges of the arrays in each chip to latches in adjacent chips serving north edges of the arrays. Instruction 3 transfers the north neighbor's M_0 value to latch B. If the north neighbor is on an adjacent chip, the value is taken from one of the latches loaded by the previous instruction. If the north neighbor is on the same chip, the value is taken directly from the neighbor's latch A. Instruction 4 computes the least significant bit of the difference, D_0 , and stores the result in memory. Instruction 5 computes a borrow value, loads the result in latch C, and reads the second bit of the operand, M_1 , from memory into latch A. Instructions 6 through 8 complete processing for the second bit position. Instructions 9 through 12 and 13 through 16 handle the third and fourth bit positions, respectively. Note that instructions involving interchip communication take place after instructions loading latch A and before instructions using the processing element interconnection logic. This insures the integrity of data transferred between chips.

¹Appendix C, Table C.1 provides low-level specifications of all the array instructions used in the neighbor difference procedure.

Table 1.2

Processing Element Array Performance. 64×64 array. 60-ns clock cycle. Parallel objects are represented by upper case letters. Scalar objects are represented by lower case letters.

Operation	Notation	Instructions per n -bit Operation	8-bit Operations per Second
write	$A \leftarrow x$	$n + 1$	7.6 G
add	$A \leftarrow A + b$	$2n$	4.3 G
copy	$A \leftarrow B$	$2n$	4.3 G
absolute value	$A \leftarrow A $	$2n$	4.3 G
sum	$S \leftarrow A + B$	$3n$	2.8 G
signed product	$P \leftarrow D \times r$	$2n^2$	0.5 G
unsigned product	$P \leftarrow D \times R$	$4n^2 - n - 1$	0.3 G
move	$A \leftarrow B^{neighbor}$	$3n$	2.8 G
neighbor difference	$D \leftarrow M - M^{neighbor}$	$4n$	2.1 G

Because processing element data paths are only one bit wide, the number of instructions required to perform arithmetic operations increases with the size of the operands and result. Processing elements execute one array instruction per clock cycle. Table 1.2 shows the number of array instructions required to perform several basic operations. It also shows the number of eight-bit operations performed per second by a 64×64 processing element array operating with a 60 ns clock cycle. As shown in the table, with many processing elements working in parallel, bit-serial processing may be used in a high-performance system.

1.4 Previous Work

Several reported devices integrate both logic and memory to form arrays of processing elements. In most of these devices, each processing element comprises a relatively large block of memory and a large processing element. For example, the BLITZEN chip [2] provides an 8×16 processing element array. Each processing element has a one-bit-wide two-input function generator, a full adder, six 1-bit registers, a 32-bit shift register, and a 32×32 SRAM array.

Two reported devices employ logic pitch-matched to memory columns. Both provide arrays of 64 processing elements with simple linear processing element interconnection networks. In one device, each processing element includes four 32-bit SRAM columns [1]. In the other device, each processing element includes a 2k-bit SRAM column [3].

1.5 Thesis Organization

This chapter introduced pixel-parallel image processing and the design and operation of the integrated processing element array. The remaining chapters discuss the primary contributions of this work: 1) compact low-power logic circuits pitch-matched to DRAM cells, 2) integration of a large two-dimensional processing element array with an interconnection network that may be extended across chip boundaries, 3) efficient control and data paths for pixel-parallel image processing, and 4) the implementation and demonstration of a fully functional real-time system.

Chapter 2 thoroughly describes the design of the integrated processing element array. Chapter 3 discusses system-level issues, including controller architecture and image I/O. Chapter 4 reports experimental results and describes demonstration applications. Chapter 5 compares the circuit and system architecture with other architectures. Chapter 6 summarizes major accomplishments and presents ideas for future research.

Appendix A provides detailed chip documentation for board-level designers. Appendix B details the rules used for layout design. Appendix C provides a detailed example of code for the processing element array. Appendix D provides detailed timing control diagrams.

Chapter 2

Integrated Processing Element Array

This chapter describes the core of the pixel-parallel image processing system: the integrated processing element array. Sections 2.1 and 2.2 introduce the design and describe the memory cell. Sections 2.3 and 2.4 describe circuits pitch-matched to memory rows and columns. Section 2.5 describes internal timing strategy and implementation. Sections 2.6 and 2.7 deal with communication between processing elements. Section 2.8 discusses image I/O and Section 2.9 presents line driver and pad circuits. Section 2.10 describes the design process used to develop the integrated processing element array.

2.1 Design Overview

At the time the integrated processing element array was designed, the most suitable technology available for prototype fabrication was the Hewlett-Packard CMOS14TB technology. CMOS14TB, offered by the MOSIS Service [4], is a “standard” 3.3-V *n*-well CMOS technology with three metal layers. It is most often used for ASIC projects. Following design rules developed by MOSIS, the minimum drawn polysilicon width is 0.6 μm and the minimum drawn polysilicon spacing is 0.9 μm .

As shown in Figure 2.1, the chip has eight blocks of 512 processing elements. At the center of each block is a twin cell DRAM array with 128 rows and 512 columns. Sense amplifiers are placed at the bottom of the DRAM array. Arrays of 256 logic units are placed above and below the DRAM array.

The chip has five supplies. Interface circuits use a 3.3-V supply, V_{HH} , and a ground supply, V_{LL} . Input and output levels are compatible with standard 3.3-V parts. Wordline drivers and platelines use a distinct 3.3-V supply, V_{PP} . Internal circuits use a 2.5-V supply, V_{DD} , and a ground supply, V_{SS} .

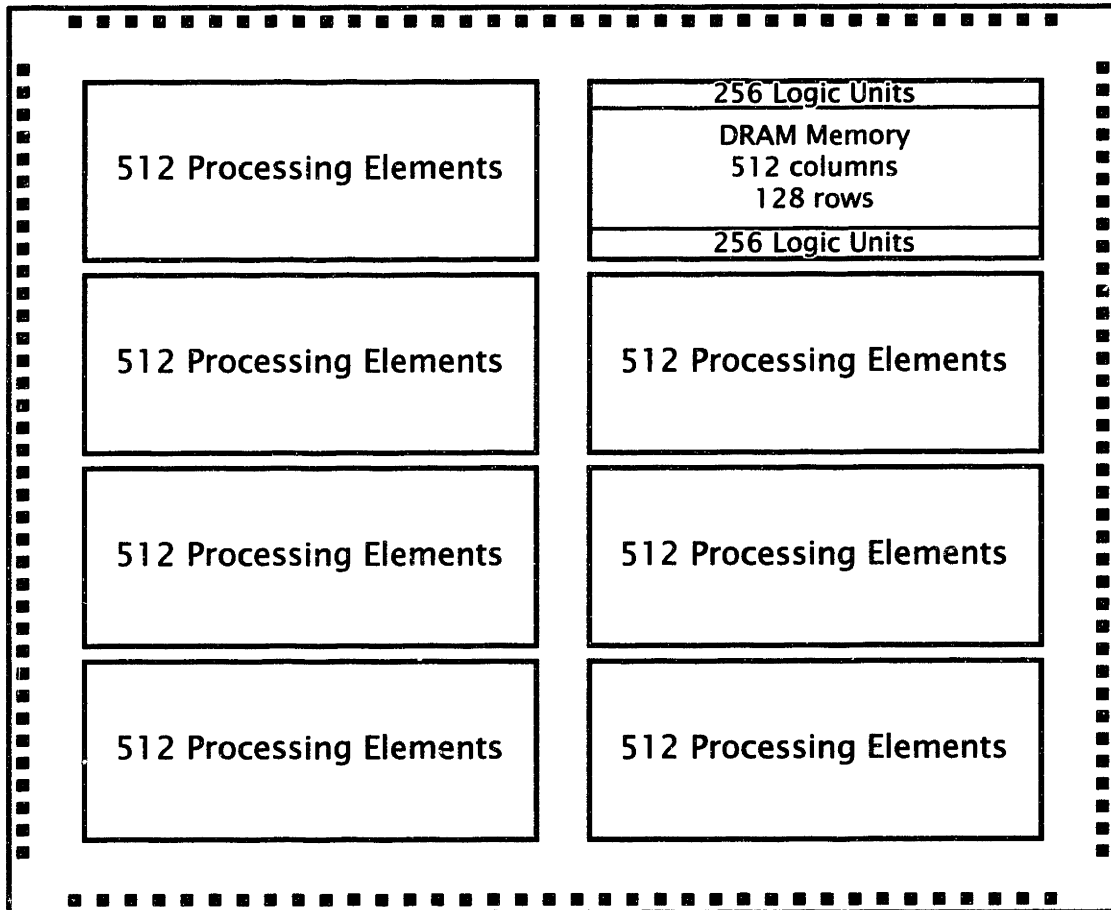


Figure 2.1: Chip organization.

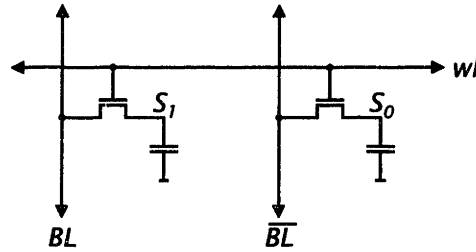


Figure 2.2: Twin cell dynamic memory.

2.2 Twin Cell Dynamic Memory

A dynamic twin cell structure is used to implement processing element memory. Twin cell structures were used in 4Kb and 18Kb DRAM devices [5, 6, 7]. More recently, a twin cell structure was used to implement a word redundancy array in a 16Mb DRAM device [8].

In a twin cell dynamic memory, two identical cells are used to store each bit. As shown in Figure 2.2, each cell consists of a transistor and a storage capacitor. A 0 is stored by establishing a high potential at storage node S_0 and a low potential at storage node S_1 . Conversely, a 1 is stored by establishing a high potential at S_1 and a low potential at S_0 . A read operation begins with the bitlines, BL and \overline{BL} , precharged to a common potential. Driving the wordline, wl , high causes charge sharing between the bitlines and storage nodes. The initial potential difference between the two storage nodes results in a differential signal on the bitlines.

The charge capacity of the cell is defined by

$$Q_C \equiv C_S(V_{SH} - V_{SL})$$

where C_S is the capacitance of the storage node, V_{SH} is the stored high potential, and V_{SL} is the stored low potential. Assuming the wordline is driven sufficiently high to equalize the bitline and storage node potentials, the nominal twin cell read signal magnitude is

$$\Delta V = \frac{Q_C}{C_{BT} + C_S} = \frac{V_{SH} - V_{SL}}{C_{BT}/C_S + 1}$$

where C_{BT} is the total capacitance of the bitline nodes.

A large charge capacity is desirable not only to provide a large read signal, but also to obtain a low soft error rate. Soft errors occur when electron-hole pairs produced by energetic particles upset stored data. A cell's sensitivity to energetic particles depends on the critical charge: the amount of charge needed to change the state of the cell. The critical charge is equal to the difference between the cell charge capacity and the amount of charge needed for proper read operation. Soft error rates increase exponentially with decreasing critical charge [9].

Three primary objectives in the design of dynamic memory cells are small cell area, large charge capacity, and small bitline capacitance. A smaller cell allows integration

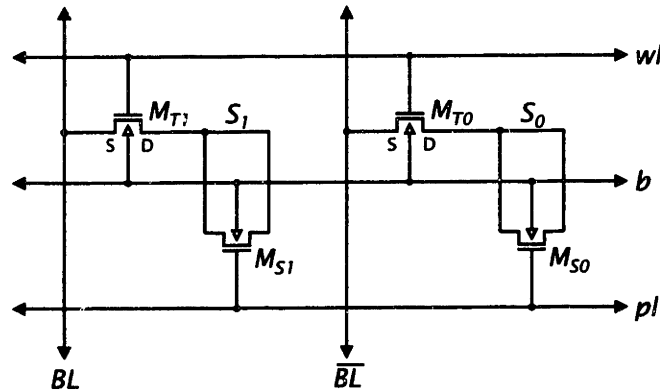


Figure 2.3: Twin cell schematic.

of more memory per unit area. Larger charge capacity and smaller bitline capacitance increase the read signal magnitude, making faster read operations possible. Larger charge capacity decreases the soft error rate.

To provide a large charge capacity in a small cell area, a structure with a large capacitance per unit area is required. In standard CMOS technologies, the largest capacitance per unit area is the oxide capacitance between channels and gates.

Figure 2.3 shows the twin cell used to implement processing element memory. The storage capacitors are formed using n -channel devices. The plateline, pl , is tied to a high-potential supply, so that there are inversion layers under the gates of the storage devices, M_{S0} and M_{S1} . The principal storage capacitance is the parallel combination of the oxide capacitance between the channel and the plateline and the capacitance between the channel and the p -channel silicon bulk.

Figure 2.4 shows the twin cell layout. Figure 2.5 shows a cross section extending through the center of a bitline. A similar cell structure has been employed in memories with one cell per bit [10, 11, 12, 13]. The bitlines run vertically in first-level metal (metal1). The wordline runs horizontally in polysilicon (poly). Second-level metal (metal2) is used to provide a low-resistance wordline shunt, $mw1$.

The bitlines are formed using metal to minimize bitline capacitance. First-level metal area and perimeter capacitances are roughly an order of magnitude lower than diffusion area and perimeter capacitances. With first-level metal claimed by the bitlines, the wordline must be formed using polysilicon. The contacts used to connect the bitlines to the sources of the transfer devices, M_{T0} and M_{T1} , are shared by adjacent cells, minimizing area per cell and bitline capacitance per cell. Table 2.1 presents calculated electrical parameters for the cell.

To maximize density, most DRAM devices are implemented using only one cell to store each bit. Each cell is connected to one bitline. For proper read operation, a reference potential must be established on the complementary bitline. This may be accomplished by setting the bitline precharge potential to an appropriate value or by employing a dummy cell. Both approaches require very careful analysis of cell charac-

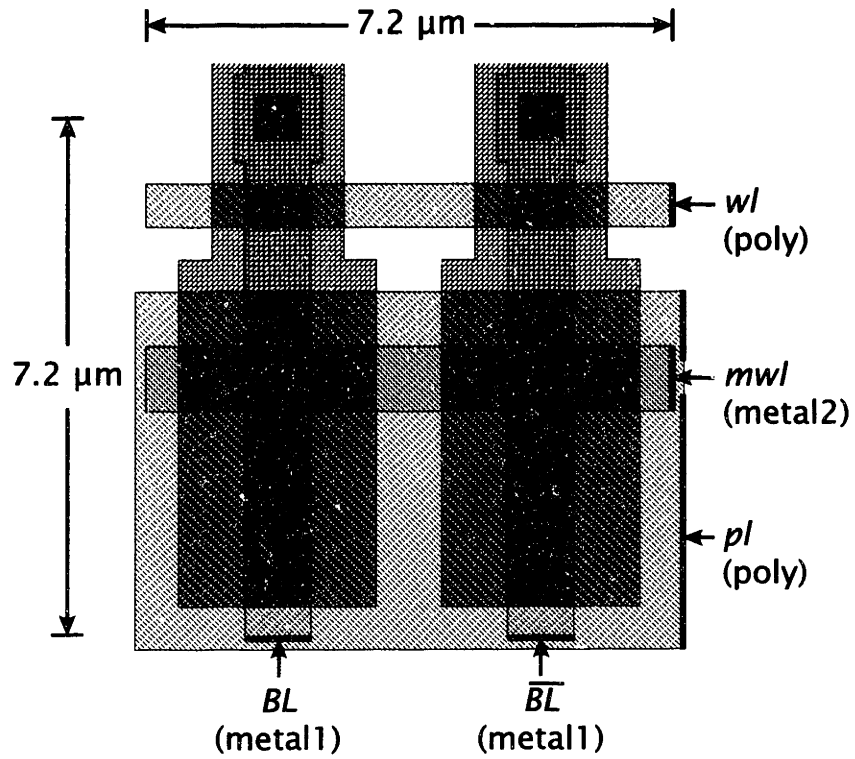


Figure 2.4: Twin cell layout.

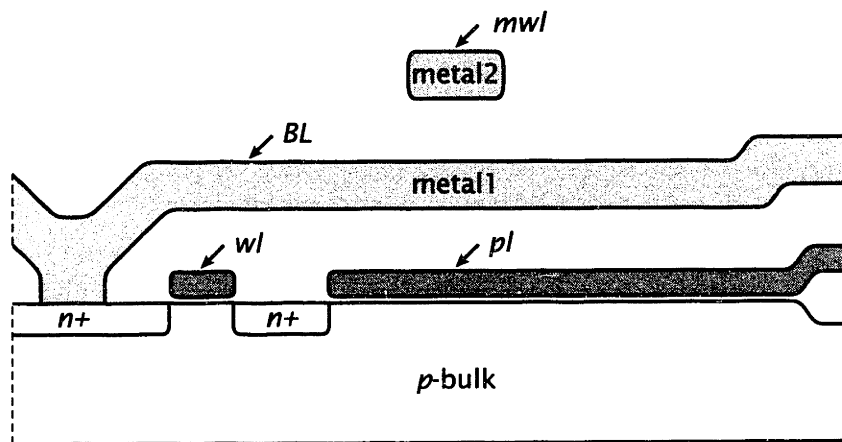


Figure 2.5: Twin cell cross section.

Table 2.1
Twin Cell Electrical Parameters

storage capacitance	C_S	45 fF
bitline capacitance	$C_{BL/cell}$	3 fF
wordline capacitance	$C_{wl/twincell}$	12 fF
bitline-wordline capacitance	C_{BL-wl}	1 fF
polysilicon wordline resistance	$R_{wl/twincell}$	100 Ω
metal wordline resistance	$R_{mwl/twincell}$	600 m Ω

teristics to insure that the reference potential will provide roughly equal 0 and 1 read signals. The detailed technology information needed to complete a precise analysis was not available. Therefore, a design with one cell per bit was not attempted.

2.3 Rows

In all memory cell schematics of the previous section, the wordlines are drawn horizontally and the bitlines are drawn vertically. This convention corresponds to the physical structure of the memory array. Each row of cells shares a wordline; each column of cells shares a pair of bitlines. This section discusses row design: the structure of the wordlines, the wordline driver circuit, and the address decode circuits that control the wordline drivers.

2.3.1 Wordlines

Polysilicon wordline resistance sets a practical limit on the number of columns between connections of the polysilicon wordline to the metal wordline. If there are too many columns between connections, the voltage near the center of polysilicon wordline spans will not respond quickly to the wordline driver. Similarly, metal wordline resistance sets a practical limit on the total number of columns per wordline. If there are too many columns per wordline, the far end of the metal wordline will not respond quickly to the wordline driver.

Wordline behavior may be evaluated analytically. The time required for the output voltage of a circuit to go from ten percent of its full transition to ninety percent of its full transition is called the rise time. In response to a step potential input, the rise time of a distributed RC network is approximately $0.9RC$ [14, pages 198-202], where R is the total resistance and C is the total capacitance.

In response to a step potential input applied to connections between a polysilicon wordline and a metal wordline, the rise time at the center of polysilicon wordline spans is

$$t_{wl,10-90} = 0.9 \cdot \left(\frac{m}{2}\right)^2 \cdot R_{wl/twincell} \cdot C_{wl/twincell}$$

Table 2.2
Polysilicon Wordline Rise Time

columns per <i>mw</i> - <i>wl</i> connection <i>m</i>	rise time (ns) $t_{wl,10-90}$
16	0.1
32	0.3
64	1.0
128	4.0
256	16.1

Table 2.3
Metal Wordline Rise Time

columns per metal wordline <i>n</i>	rise time (ns) $t_{mw,10-90}$
128	0.1
256	0.4
512	1.7
1024	6.8

where m is the number of columns between connections. The calculated rise times for several values of m are presented in Table 2.2. The final design has 64 columns between connections, corresponding to a 1.0 ns polysilicon wordline rise time. Fewer connections would result in an unacceptably large rise time. More connections would increase the size of the array but would not provide a significantly smaller rise time.

In response to a step potential input applied to one end of a metal wordline, the rise time at the other end is

$$t_{mw,10-90} = 0.9 \cdot n^2 \cdot R_{mw}/twincell \cdot C_{wl}/twincell$$

where n is the number of columns per metal wordline. The calculated rise times for several values of n are presented in Table 2.3. The final design has 512 columns per metal wordline, corresponding to a 1.7 ns metal wordline rise time. Shorter metal wordlines would require a considerable increase in chip area. A design with 256 columns per metal wordline would be implemented as sixteen blocks of 128 processing elements. The number of address decoders and logic timing circuits would be twice the number in the eight-block design. Chip area would be roughly 4% larger.

Speed is one of two critical issues influenced by wordline structure. The other issue is crosstalk. There are significant parasitic capacitances between bitlines and wordlines. Therefore, bitline transitions may cause transient voltage changes on wordlines. If crosstalk magnitude is too large, a bitline transition may produce a large positive transient on an inactive wordline that partially turns on transfer devices, altering the state of storage nodes.

Crosstalk magnitude at a transfer device depends on the resistance between the gate of the transfer device and the wordline driver. If the resistance is very high, crosstalk magnitude is limited only by the ratio of the capacitance between bitlines and wordlines to the capacitance between wordlines and bulk silicon. If the resistance is lower, crosstalk magnitude is limited by the product of the resistance, the capacitance between bitlines and wordlines, and the speed of bitline transitions.

Maximum total crosstalk magnitude is the sum of the crosstalk-induced voltage from

the ends of a polysilicon wordline span to the center of the span, v_{wl} and the crosstalk-induced voltage at the far end of the metal wordline, v_{mwl} . The wordline current per cell produced by a bitline transition is $C_{BL-wl} \cdot dv_{BL}/dt$, where dv_{BL}/dt is the rate of change of the bitline voltage. If all cells are subjected the same bitline transition, the crosstalk component corresponding to the polysilicon wordline is

$$v_{wl} = \left[1 + 2 + \dots + \frac{m}{2} \right] \cdot R_{wl/twincell} \cdot C_{BL-wl} \cdot \frac{dv_{BL}}{dt},$$

and the crosstalk component corresponding to the metal wordline is

$$v_{mwl} = [1 + 2 + \dots + n] \cdot R_{mwl/twincell} \cdot C_{BL-wl} \cdot \frac{dv_{BL}}{dt}.$$

Fortunately, the wordline structure chosen according to speed concerns also keeps crosstalk quite low. With $m = 64$ columns between polysilicon wordline connections and $n = 512$ columns per metal wordline, a very rapid bitline transition, $dv_{BL}/dt = 1$ V/ns, would produce crosstalk components $v_{wl} = 38$ mV and $v_{mwl} = 67$ mV. Furthermore, as described in Section 2.4, bitline swings are generally balanced. The potentials of the two bitlines in each column change in opposite directions and the crosstalk effects cancel.

2.3.2 Wordline Driver

The wordline driver circuit, shown in Figure 2.6, has three primary components: a level converter, large pull-up and pull-down devices, $M_{P1,N1}$, and a pull-down device, M_{N6} , that is part of a wired-OR structure. The large pull-up and pull-down devices are separated from the level converter by an inverter, $M_{P2,N2}$. Pull-up current is supplied by V_{PP} , a 3.3-V supply.

The wordline driver input, in , is driven by an address decoder. The address decoders use the 2.5-V supply, V_{DD} . The wordline driver must therefore include a level converter. The level converter is formed using cross-coupled p -channel devices, $M_{P3,4}$, two n -channel pull-down devices, $M_{N3,4}$, and an inverter $M_{P5,N5}$. The n -channel pull-down devices are sized to quickly overpower the cross-coupled p -channel devices so that short-circuit currents are low and transitions are fast. Devices $M_{P3,N3}$ are larger than $M_{P4,N4}$ because they must drive the gates of devices M_{P2} and M_{N2} .

2.3.3 Address Decode Circuits

Address decode circuits use the binary addresses specified by array instructions to produce 128 wordline driver inputs. A simple implementation would use a column of 128 seven-input AND circuits. Lines carrying the address input signals and their complements run through the column. Each AND input is connected to an input signal or its complement. The connection pattern establishes the mapping between input addresses and wordlines. Each of the fourteen address lines serves 64 AND inputs.

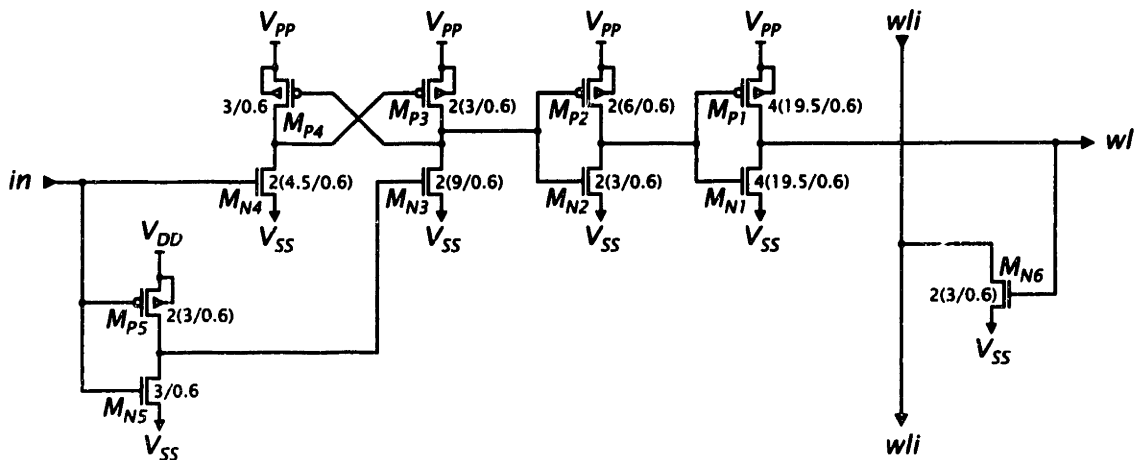


Figure 2.6: Wordline driver.

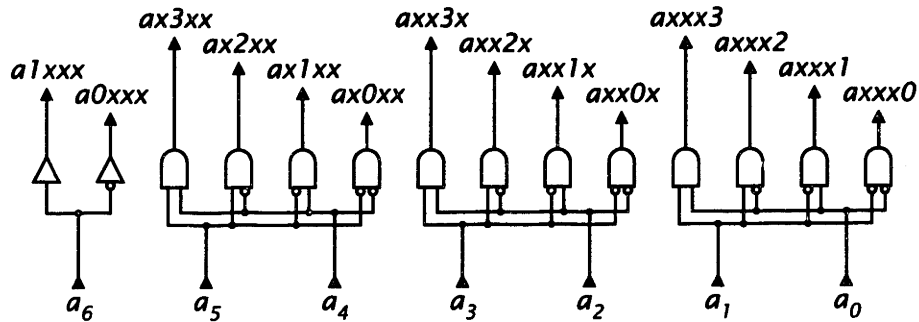


Figure 2.7: Address predecoder.

The actual implementation uses a predecoding scheme to reduce the complexity of the pitch-matched circuits and to reduce the loading on the address lines [15]. As shown in Figure 2.7, binary address signals a_{6-0} are used to generate fourteen “predecoded” address signals. Twelve of the predecoded signals are produced by three 2-to-4 decoders. The remaining two signals represent a_6 and its complement.

The address decoder, shown in Figure 2.8, is pitch-matched to memory rows. A single decoder serves four wordlines; a column of 32 decoders serves a full memory array. Lines carrying the predecoded address signals run through the column of address decoders. The layout pitch of small two-input AND circuits matches the row pitch. The layout of the four-input AND gate fits within four row pitches. Each of the twelve address lines carrying signals from the 2-to-4 decoders serves only 32 AND inputs.

Each two-input AND circuit takes input from the four-input AND and from one of the four address signals generated from a_1 and a_0 . Each four-input AND circuit takes input from one of the four address signals generated from a_3 and a_2 , from one of the four address signals generated from a_5 and a_4 , from one of the two address signals generated from a_6 , and from a timing signal wl .

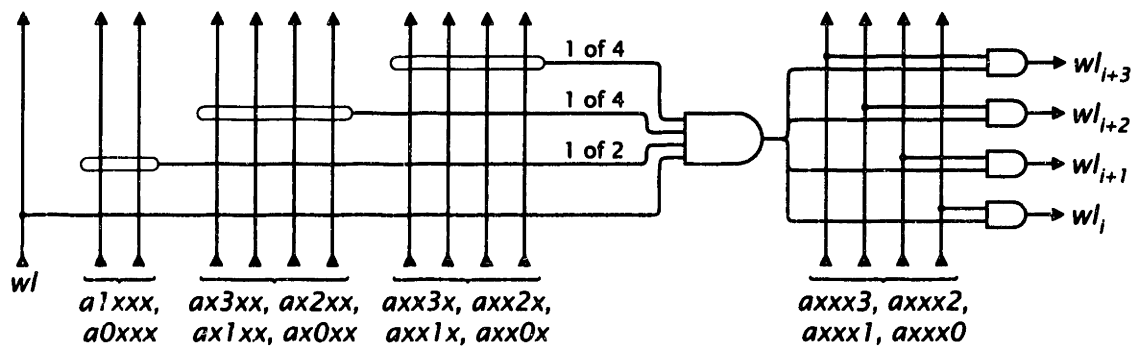


Figure 2.8: Address decoder.

2.4 Columns

The logic unit, pitch-matched to DRAM columns, is perhaps the most distinct element of the integrated processing element array. This section covers the sense amplifier and all of the circuits comprising the logic unit. The circuits are physically arranged in the following order: 1) sense amplifier, 2) write driver, 3) latch A, 4) interconnect logic, 5) function generator, 6) latches B, C, D, and E, and 7) write logic. The discussion follows the same order.

2.4.1 Sense Amplifier

Wordline activation produces a small read signal on the bitlines. The signal must be amplified, and the original state of the storage capacitors must be restored. Before the next memory operation, the bitlines must be precharged to a common potential. The sense amplifier performs these functions.

Figure 2.9 shows the sense amplifier. At the beginning of a read operation, if a wordline is high it is driven low. Then the source of the p -channel cross-coupled pair, sap , is driven low and the source of the n -channel cross-coupled pair, san , is driven high, turning off the cross-coupled devices. Signals eq and pc are driven high, equalizing and precharging the bitlines through devices M_{N3} , M_{N4} , and M_{N5} . After the bitline potentials settle, signals eq and pc are driven low, isolating the bitlines from each other and from node vpc . Next, the selected wordline is driven high, producing a signal on the bitlines. Finally, sap is driven high and san is driven low, amplifying the signal through the two cross-coupled pairs.

Devices M_{P1} , M_{N1} , M_{P2} , and M_{N2} are each implemented using two transistors with common drain, gate, source, and bulk connections. The two transistors comprising each device share a common n -diffusion drain. The n -diffusion source of one transistor is to the left of the drain. The source of the other transistor is to the right. This arrangement suppresses mismatches due to processing variations and minimizes layout area [16].

A pair of bitlines is equalized through device M_{N3} . The purpose of devices M_{N4} and

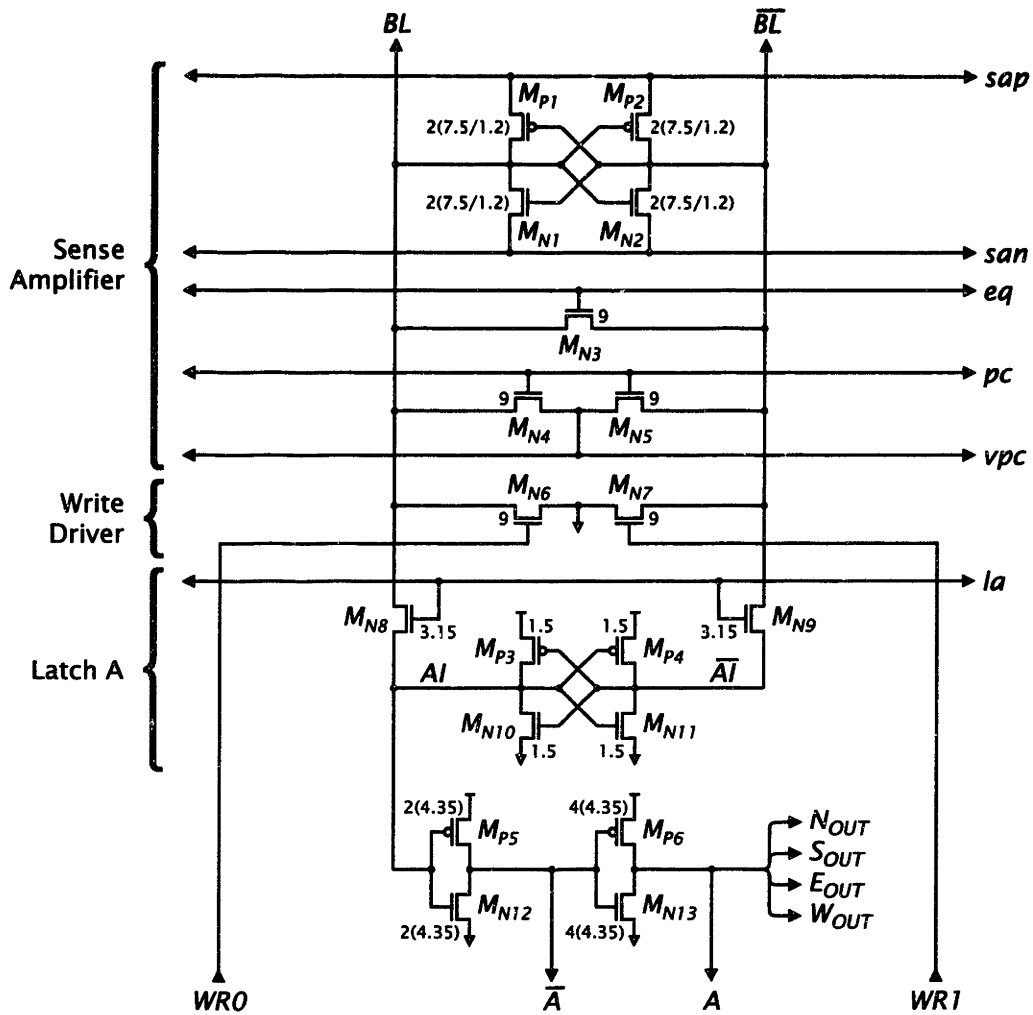


Figure 2.9: Sense amplifier, write driver, and latch A.

M_{N5} is to equalize the potentials of all bitlines, so that all sense amplifiers begin signal amplification at the same time. If the precharged potentials varied from column to column, amplification would start earlier in some columns than others. Crosstalk from columns where amplification starts early, to columns where amplification starts late, could cause read errors.

The sense amplifier controls bitline operating voltages. The maximum *sap* voltage and the minimum *san* voltage are the maximum and minimum bitline voltages. The bitline precharge voltage may be controlled by driving *vpc* to the desired level. The choice of operating voltages affects the charge capacity of the memory cells, power dissipation, speed, and layout area.

Charge capacity considerations are relatively simple. As explained in Section 2.2, the charge capacity of the cell is proportional to the difference between the high and low stored potentials. These potentials depend on maximum *sap* and minimum *san* voltages.

As a bitline is pulled down through the sense amplifier, there is an increase in the gate-source voltage of the transfer device connected to the bitline and the active wordline. The corresponding storage node potential follows the bitline potential. Thus, with *san* driven down to the V_{SS} supply, the stored low potential is 0 V.

When a bitline is pulled up through the sense amplifier, the transfer device turns off as the difference between the wordline potential and the storage node potential approaches the device's threshold voltage. Body effect increases the threshold voltage. Given the wordline high level, 3.3 V, driving bitlines higher than the V_{DD} supply, 2.5 V, would not increase the stored high potential. So *sap* is driven up to the V_{DD} supply. The stored high potential is $3.3 \text{ V} - V_T$, where V_T is the threshold voltage of the transfer device.

Power dissipation, speed, and layout considerations are more complex. They are all relevant to the choice of the bitline precharge potential, V_{PC} .

A large component of memory power dissipation is the dissipation due to bitline charging and discharging. Bitline power dissipation may be calculated by summing the amount of charge taken from the V_{DD} supply. To precharge the bitlines, the bitline potentials may be equalized, then charged or discharged to the precharge potential. The charge taken from the supply is

$$Q_{pc} = \begin{cases} 2(V_{PC} - V_{DD}/2)C_{BL}, & \text{if } V_{PC} > V_{DD}/2 \\ 0, & \text{if } V_{PC} \leq V_{DD}/2. \end{cases}$$

where C_{BL} is the bitline capacitance. During signal amplification the sense amplifier pulls up one bitline to V_{DD} . The charge taken from the supply is

$$Q_{amp} = (V_{DD} - V_{PC})C_{BL}.$$

The total amount of charge used per cycle is given by

$$Q_{BL} = Q_{amp} + Q_{pc} = (V_{DD}/2 + |V_{PC} - V_{DD}/2|) C_{BL}.$$

Thus bitline power dissipation per column is given by

$$P_{BL} = \frac{V_{DD}Q_{BL}}{t_{cycle}} = \frac{V_{DD}(V_{DD}/2 + |V_{PC} - V_{DD}/2|)C_{BL}}{t_{cycle}},$$

where t_{cycle} is the cycle time.

Precharging the bitlines to $V_{DD}/2$ clearly minimizes bitline power dissipation. But this advantage had to be balanced against other considerations: signal development speed, sense amplifier complexity, required support circuits, and other components of memory power dissipation. A 16Mb DRAM design team found that these other considerations outweighed the bitline power savings [8],

For DRAM cells implemented using n -channel devices, lower bitline precharge potentials provide faster signal development. Transfer devices are turned on as soon as the difference between the active wordline potential and the precharge potential exceeds the transfer device threshold voltage. Once the active wordline is at its maximum voltage, the gate-source voltages of the transfer devices are greater when the bitline voltage is lower.

With bitlines precharged to $V_{DD}/2$, both n -channel and p -channel cross-coupled pairs are active during initial signal amplification and source nodes of both pairs must be controlled. With bitlines precharged to ground, only the p -channel cross-coupled pair is active during initial signal amplification. Thus, mismatches between the n -channel cross-coupled devices do not produce significant amplifier offset. So the n -channel devices may be relatively small devices with minimum gate length. The sources of the n -channel cross-coupled devices may be connected directly to ground.

Lower bitline precharge potentials allow use of smaller precharge and equalization devices. With bitlines precharged to $V_{DD}/2$, the gate-source voltage on the equalization and precharge devices is only $V_{DD}/2$. With bitlines precharged to ground, the gate-source voltage on the equalization and precharge devices is V_{DD} .

With smaller devices, sense amplifier designs using bitlines precharged to ground require less area than $V_{DD}/2$ designs. In addition, with smaller devices and no need to vary the source node voltage of the n -channel cross-coupled pair, the collection of circuits that drive sense amplifier control signals dissipates less power.

In a conventional DRAM device with a short access time, precharging bitlines to $V_{DD}/2$ generally requires a precharge voltage generator. Bitlines are precharged at the end of operation cycles. That way, precharge time does not contribute to access time. The $V_{DD}/2$ bitline precharge voltage may be established by equalizing the bitline potentials, but there is no guarantee that one operation cycle will be immediately followed by another. So, unless bitlines are actively held at $V_{DD}/2$, bitline voltages may drift above or below $V_{DD}/2$ before the beginning of an operation cycle. Such changes undermine sense amplifier operation. The $V_{DD}/2$ voltage generators necessary to prevent bitline voltage drift require power and layout area.

For the cell design, operation timing, and circuit structures employed in this work, the bitline power savings of a $V_{DD}/2$ design outweighs all of the other considerations.

If the DRAM cell array were formed using p -channel devices, signal development with bitlines precharged to $V_{DD}/2$ may have been too slow. But, with the n -channel array, signal development time is less than the latency of the corresponding timing circuits. A voltage generator is not necessary since bitlines are equalized at the beginning of instructions. The sense amplifier requires somewhat more area and somewhat more complex driver circuits than would be necessary with bitlines precharged to ground. But the area cost is very small relative to the area of the memory column, and the additional power dissipated in the driver circuits is far less than the bitline power savings.

2.4.2 Write Driver and Latch A

As shown in Figure 2.9, the write driver comprises two devices, M_{N6} and M_{N7} . The devices are sized so that they will overpower devices M_{P1} and M_{P2} . During write operations, WRO goes high to write a 0 and $WR1$ goes high to write a 1.

When control signal la is high, latch A captures the result of read operations. As shown in Figure 2.9, the latch is implemented using the same structure as a six-transistor SRAM cell. This structure requires only one control signal, maintains its state during periods of inactivity, and does not require much layout area.

Latch A drives the input of a series of two inverters, $M_{P5,N12}$ and $M_{P6,N13}$. The output of inverter $M_{P5,N12}$, \bar{A} , goes to the function generator. The output of inverter $M_{P6,N13}$, A , goes to the function generator and to four neighboring processing elements.

2.4.3 Interconnect Logic and Function Generator

The function generator and processing element interconnect logic are combined in a single dynamic gate, shown in Figure 2.10. At the beginning of instructions, control signal \bar{fp} is low, precharging node FI through device M_{P7} . The twelve evaluate signals, f_{7-0} , and $f_{N,S,E,W}$ are all low, so there is no static power dissipation. Later, \bar{fp} goes high, isolating FI . Then, the evaluate signals specified by the instruction go high. After an interval long enough for a single NAND stack to pull down node FI , the specified evaluate signals return low.

Signals from four neighboring processing elements go to four two-transistor NAND stacks. If a neighbor input signal is high and the corresponding evaluate signal goes high, the evaluation node, FI , is pulled down through a NAND stack.

The function generator evaluate signals, f_{7-0} , correspond to output values of a truth table. Signals from latches A, B, and C enable only one of the eight four-transistor NAND stacks. The function generator result is determined by the corresponding evaluate signal. If the evaluate signal goes high, node FI is pulled down through the stack. Otherwise, FI stays high unless pulled down by the interconnect logic.

In general, large dynamic gates suffer from charge sharing between the evaluation node and internal nodes. To avoid charge sharing problems, the data input signals from neighboring processing elements and from latches A, B, and C must be stable before

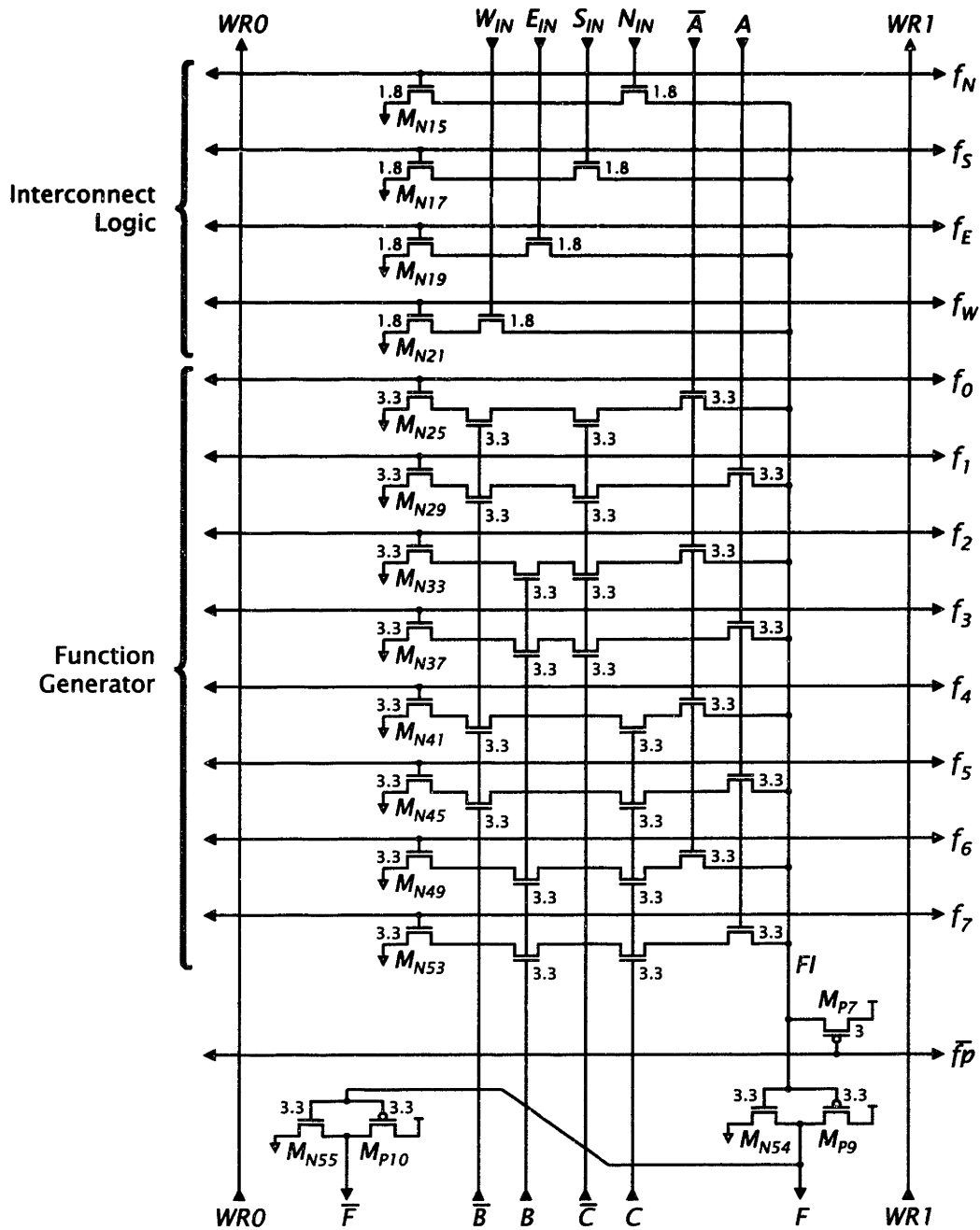


Figure 2.10: Processing element interconnect logic and function generator.

control signal \overline{fp} goes low. That way, internal nodes in the NAND stacks are precharged along with node FI . Transistors connected to evaluate signals are placed at the bottom of the NAND stacks so that they do not isolate any of the internal nodes from node FI . To avoid logic errors, the data input signals must remain stable until all the evaluate signals return low.

2.4.4 Latches B, C, D, and E

As shown in Figure 2.11, Latches B, C, D, and E are all implemented using the same six-transistor SRAM structure as for latch A. Internal nodes of the latches, B , \overline{B} , C , \overline{C} , D , \overline{D} , and E , are directly connected to function generator and write logic devices, avoiding the area requirements of buffers. Despite the small size of the p -channel devices, simulations indicate that the rise and fall times of the internal nodes are under 2 ns. The results also indicate that short-circuit currents in the latches are small relative to total latch current. The function generator evaluate signals and the write logic control signal, wr , are all low when the latches change state. Thus, latch transients do not cause short-circuit current in the function generator or write logic.

2.4.5 Write Logic

The write logic, shown in Figure 2.11, is quite simple. Two three input NAND gates control the write signals $WR1$ and WRO . Both write signals are low whenever the control signal wr is low and whenever the latch E holds value 0. When latch E holds value 1 and wr goes high, the write signals depend on the value held in latch D. If latch D holds value 1, $WR1$ goes high and WRO stays low. If latch D holds value 0, WRO goes high and $WR1$ stays low.

2.5 Timing

The timing of the control signals for the wordline drivers, sense amplifiers, and logic circuits must be coordinated to insure proper processing element operation. Several general techniques are available to implement timing control: input clocks, custom timing logic, and dummy circuits. Control with input clocks allows timing to be optimized for the characteristics of fabricated devices, but each additional input clock increases the complexity of board-level circuits. Control with custom logic is often simple, but logic delays cannot be reliably matched to processing element circuit speed. Dummy circuits provide excellent matching, but sometimes it is not possible to construct suitable structures. The final design combines all three techniques.

Figure 2.12 shows the timing for the processing element circuits. Clock signals from two chip input pads are used to generate four internal clock signals, $p00$, $p01$, $p11$, and $p10$. Instructions begin on the rising edge of $p00$. Bitline potentials are equalized and precharged at the beginning of read and write instructions. The dynamic logic gate and

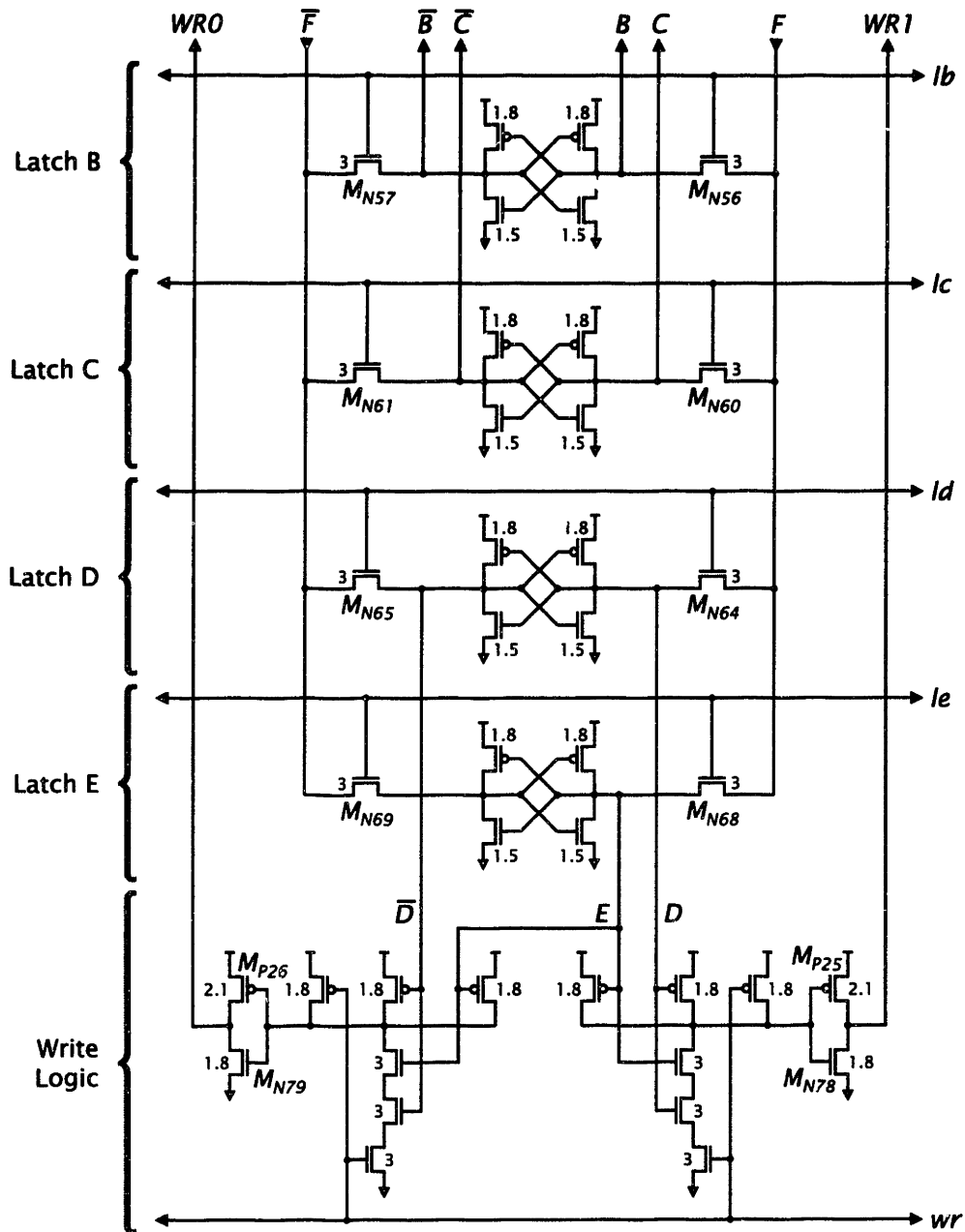


Figure 2.11: Latches B, C, D, and E and write logic.

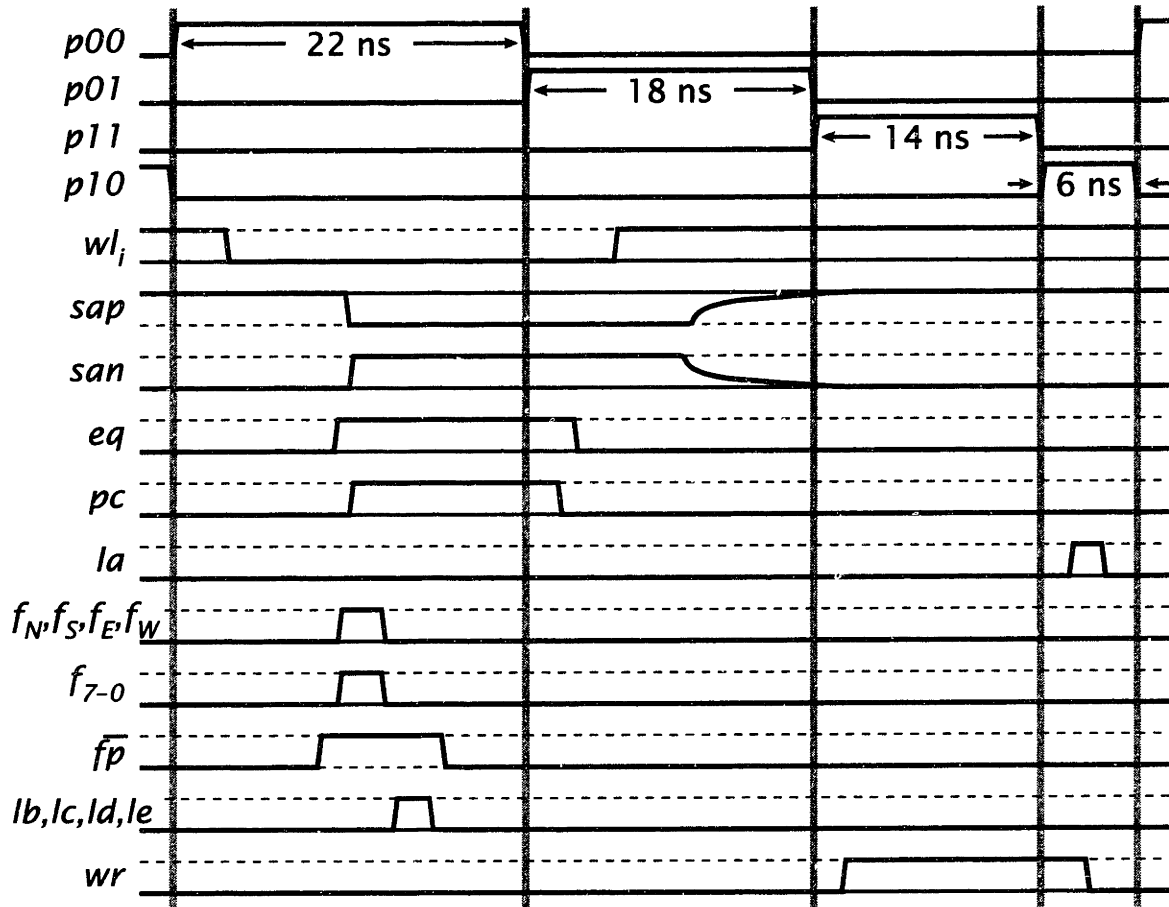


Figure 2.12: Processing element timing.

latches B, C, D, and E are active during the same interval. Signal $p01$ triggers the sensitive portion of memory operations. Equalization is completed, the selected wordline, wl_i , is driven high, and the sense amplifier is activated. During write instructions, $p11$ activates the write driver. During read instructions, $p10$ activates latch A.

The wired-OR circuit shown in Figure 2.13 is used to detect wordline transitions. The circuit is active when signal wlx is high. Each of the 128 wordline drivers includes an n -channel pull-down device connected to node wli . Devices M_{N5-12} are sized so that the current through device M_{P5} is less than one-sixteenth the saturation current of one pull-down device. Devices M_{P5} and M_{P6} are sized so that the maximum current through M_{P6} is less than one-fourth the saturation current of one pull-down device. When the selected wordline goes high, the corresponding pull-down device turns on. Node wli goes low. The output, \bar{wlq} , follows wli . When the selected wordline goes low the pull-down device turns off, node wli is pulled up through M_{P6} , and node \bar{wlq} goes high. When signal wlx is low, the current through devices M_{P5} and M_{N5-12} stops and wli is held low by device M_{N4} .

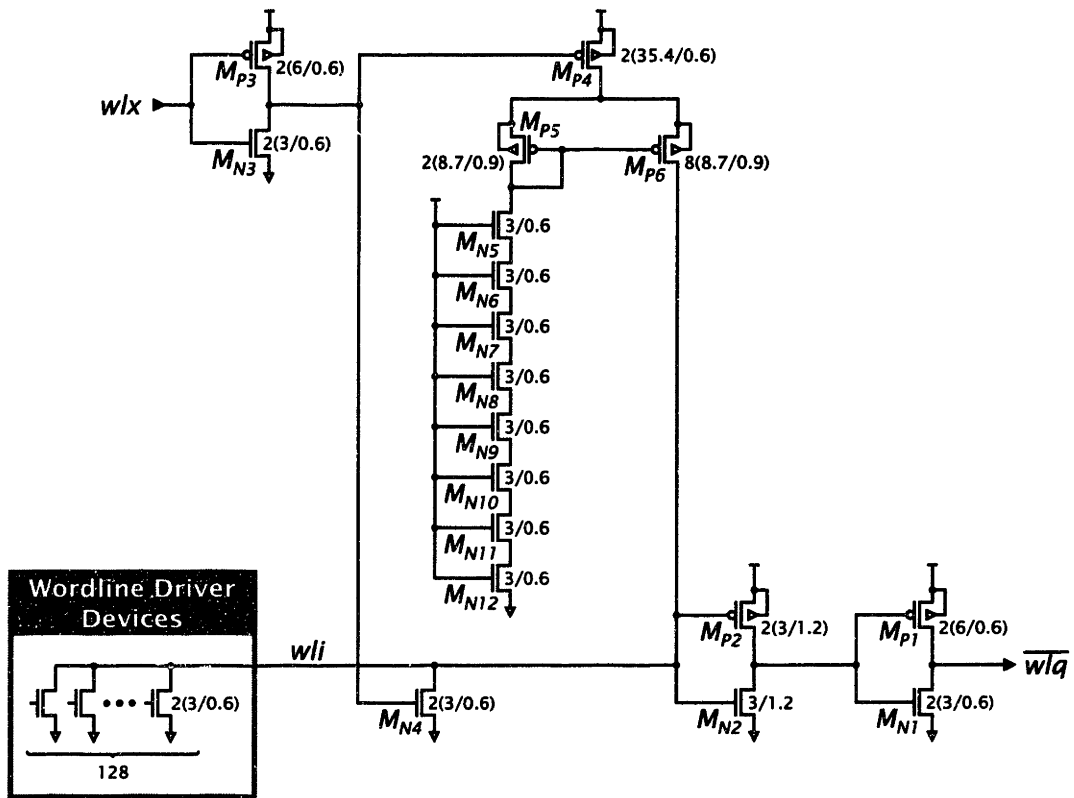


Figure 2.13: Wordline detector.

A dummy dynamic gate corresponds to the interconnect logic and function generator. A dummy latch corresponds to latches B, C, D, and E. While $p00$ is high, these dummy circuits are used to control the timing of signals f_{7-0} , $f_{N,S,E,W}$, \overline{fp} , lb , lc , ld , and le . Detailed timing control diagrams are provided in Appendix D.

2.6 Processing Element Interconnection

With logic units placed above and below DRAM columns, a block of 512 processing elements has two large rows of logic units. The complete chip, with four pairs of two blocks, has only eight rows of logic units. Thus, the 4096 processing elements on the chip are in a 512×8 arrangement.

The I/O bandwidth required to extend the processing element interconnection network across chip boundaries is proportional to the perimeter of the on-chip array. The perimeter of a 512×8 array is over four times greater than the perimeter of a 64×64 array. Even for a 64×64 array, interchip communication requires a large number of pads and considerable power. Thus, the physical arrangement of the processing elements is not a good logical organization.

Fortunately, the 512×8 arrangement can be “folded” to form a 64×64 array. Figure 2.14 illustrates the technique. Processing elements in an 8×2 physical arrangement are interconnected to form a 4×4 array. In the physical structure, a horizontal connection passes through each processing element. These connections correspond to the horizontal connections in the 4×4 array. For example, the connection between processing elements 0,0 and 0,1 passes through processing element 1,0.

The 512×8 arrangement of the 4096 processing elements on the chip is folded to form a 64×64 array. Eight horizontal connections pass through each processing element. These connections account for almost 15% of the total logic unit layout area, so the implementation of the connections demanded careful consideration.

FastCap, a numerical capacitance extraction program, was used to evaluate the layout of the lines forming the connections. Both first-level metal and polysilicon were considered. Results are shown in Table 2.4. The distance between metal and bulk silicon is greater than the distance between polysilicon and bulk silicon. So the parasitic capacitances between metal lines and ground are relatively small. But metal lines are much thicker than polysilicon lines and thus parasitic capacitances between adjacent metal lines are relatively large. Considering all parasitic capacitances, for the small line pitch, metal lines exhibit larger worst-case effective capacitance. Therefore, horizontal connections are implemented using polysilicon lines.

2.7 Interchip Communication

For the 64×64 array, simply extending the on-chip wiring to connect processing elements between chips would require 512 pads: one input pad and one output pad for

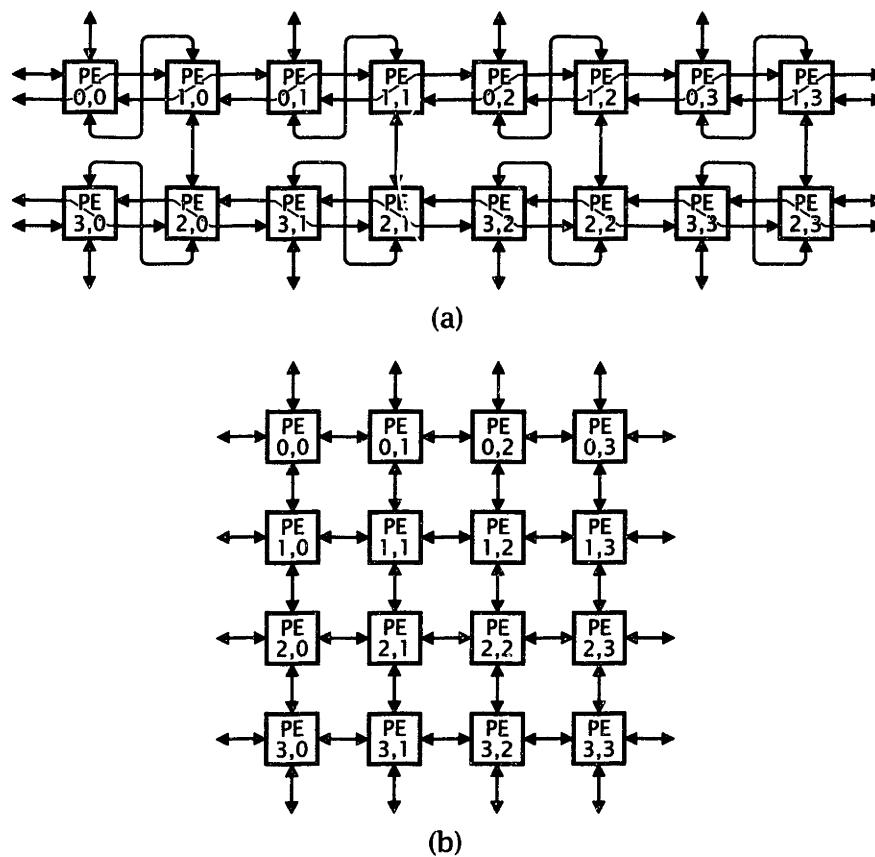


Figure 2.14: An 8×2 block of processing elements, (a), interconnected to form a 4×4 array, (b).

Table 2.4
Properties of Horizontal Processing Element Interconnection Lines

		Actual Polysilicon Implementation	Alternative Metal Implementation
line width		$0.76 \mu\text{m}$	$1.14 \mu\text{m}$
line spacing		$1.52 \mu\text{m}$	$1.14 \mu\text{m}$
line-bulk capacitance	C_{LB}	20 fF	11 fF
line-line capacitance	C_{LL}	1 fF	5 fF
nominal effective capacitance	$C_{LB} + 2C_{LL}$	22 fF	21 fF
worst-case effective capacitance	$C_{LB} + 4C_{LL}$	24 fF	30 fF

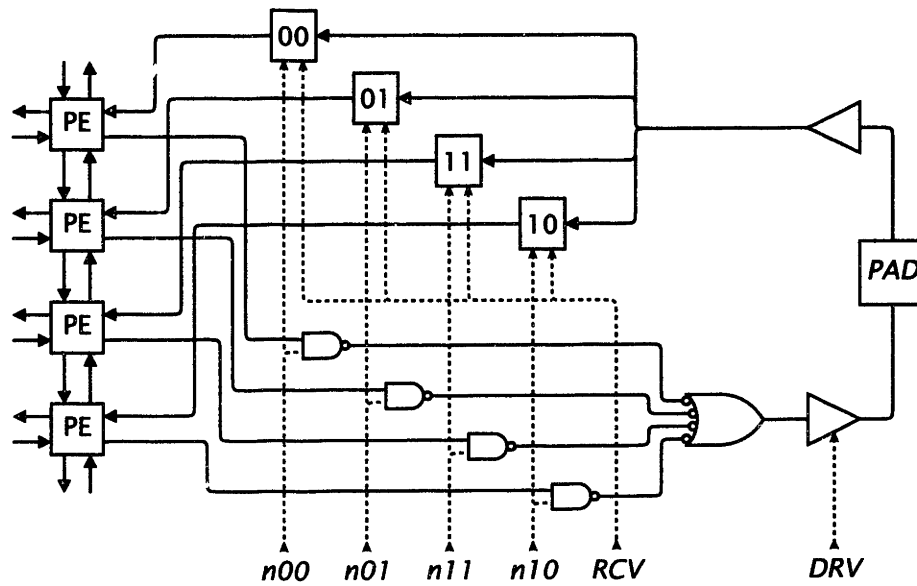


Figure 2.15: Interchip communication multiplexer.

each of the 256 perimeter processing elements. Bidirectional pad circuits and multiplexers are used to implement interchip communication with only 64 pads.

As shown in Figure 2.15, a single pad serves four perimeter processing elements. Processing element signals are multiplexed using four non-overlapping internal clock signals, $n00$, $n01$, $n11$, and $n10$. The four clock signals are generated using clock signals from two chip input pads. Each of the four internal clock signals is high during one-quarter of the clock cycle. When control signal DRV is high, signals from the four processing elements are driven to an adjacent chip through the pad. When control signal RCV is high, signals from processing elements on an adjacent chip are captured by latches 00 , 01 , 11 , and 10 .

As shown in Figure 2.16, four signals, $NORTH$, $SOUTH$, $EAST$, and $WEST$, are used as DRV and RCV signals in the multiplexer circuits. The four signals correspond to communication direction. For example, when signal $NORTH$ is high, circuits serving the bottom edge of the array drive data to an adjacent chip and circuits serving the top edge of the array receive data from an adjacent chip. To minimize power dissipation the interchip communication circuits are only active when necessary.

2.8 Image Input/Output

Dedicated serial-access memories transfer image data to and from the processing element array. As shown in Figure 2.17, the memories are incorporated into the processing element interconnection network. The 64×64 processing element array is conceptually divided into four 64×16 groups. Sixty-four serial-access memory cells are placed at

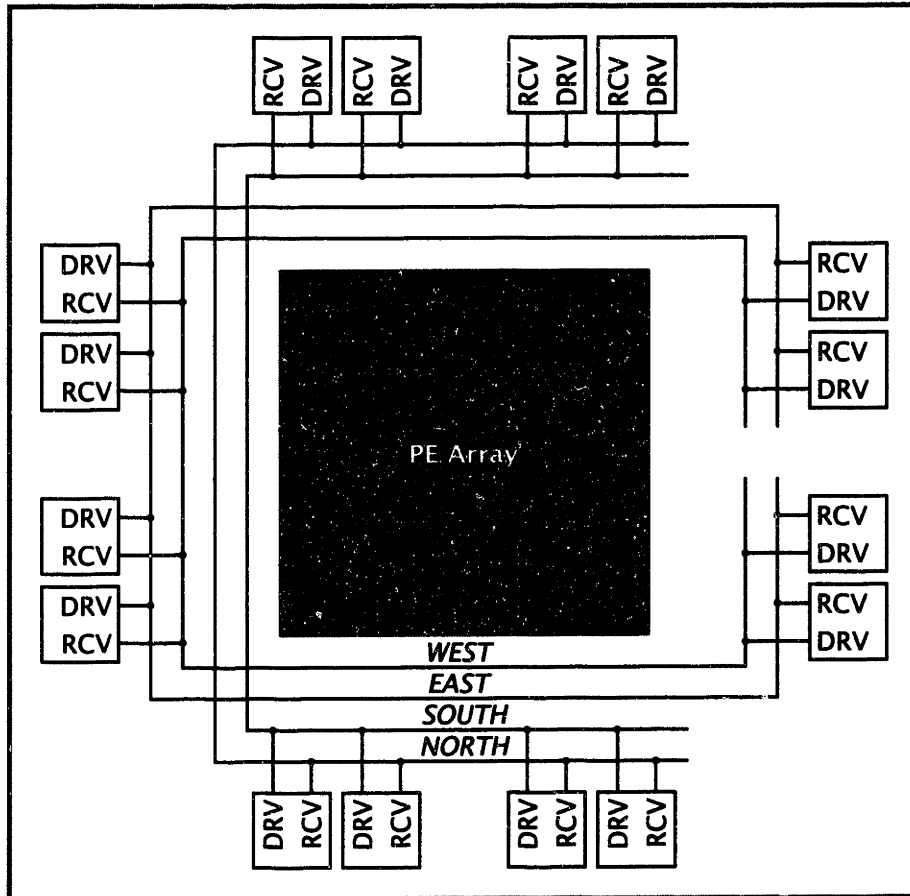


Figure 2.16: Interchip communication control. Operation of the multiplexers inside the perimeter of the chip is governed by signals *WEST*, *EAST*, *SOUTH*, and *NORTH*.

the bottom of each group. Each serial-access memory cell is connected to a processing element in the bottom row of the group. The serial-access memory cells are connected together to form two serial access memories.

Figure 2.18 is a functional representation of the serial access memory cell. The cell is comprised of two latches and a multiplexer. The cell sits between two connected processing elements. Input signal N_{IN} is from latch A of the processing element above the cell. Input signal S_{IN} is from latch A of the processing element below the cell. Output signals N_{OUT} and S_{OUT} go to the processing elements above and below the cell, respectively. During normal operation, control signal STR is low, so the multiplexer passes the S_{IN} input to the N_{OUT} output.

The signal from the preceding cell is captured by latch 1 when control signal $SCK1$ is high. The signal from latch 1 is captured by latch 2 when control signal $SCK2$ is high. Alternate $SCK1$ and $SCK2$ pulses move data through the cell. The control signals for the cell are independent of the control signals for the rest of the chip, thus image data may be transferred through the serial-access memories using clock frequencies different from the operating frequency of the processing element array. In addition, image data may be transferred while the processing element array is active.

To transfer image data from the cell to the processing element array, control signal STR goes high so the multiplexer passes the signal from latch 1 to the N_{OUT} output. To transfer data from the processing element array to the cell, control signal SLD goes high and latch 2 captures the N_{IN} input.

2.9 Perimeter Circuits

This section describes two of the more critical perimeter circuits, the line driver circuit and the I/O pad circuit.

2.9.1 Line Driver

Forty-one third-level metal lines carry address signals, control signals, and clock signals from circuits near the perimeter of the chip to the address and control circuits located between blocks of processing elements. These lines, running vertically through the center of the chip, are over 7 mm long. With minimum width and spacing, the lines fit over the address and control circuits. But the capacitive coupling between neighboring lines causes crosstalk. As a line switches state, the drivers for neighboring lines must sink or source current. Since driver output conductance is finite, the current produces a transient voltage change.

The metal lines can be modeled by capacitors between each line and ground, and capacitors between neighboring lines. Crosstalk magnitude depends on the size of the capacitance between neighboring lines relative to the size of the capacitance between each line and ground. The capacitance between neighboring lines is about 500 fF. The

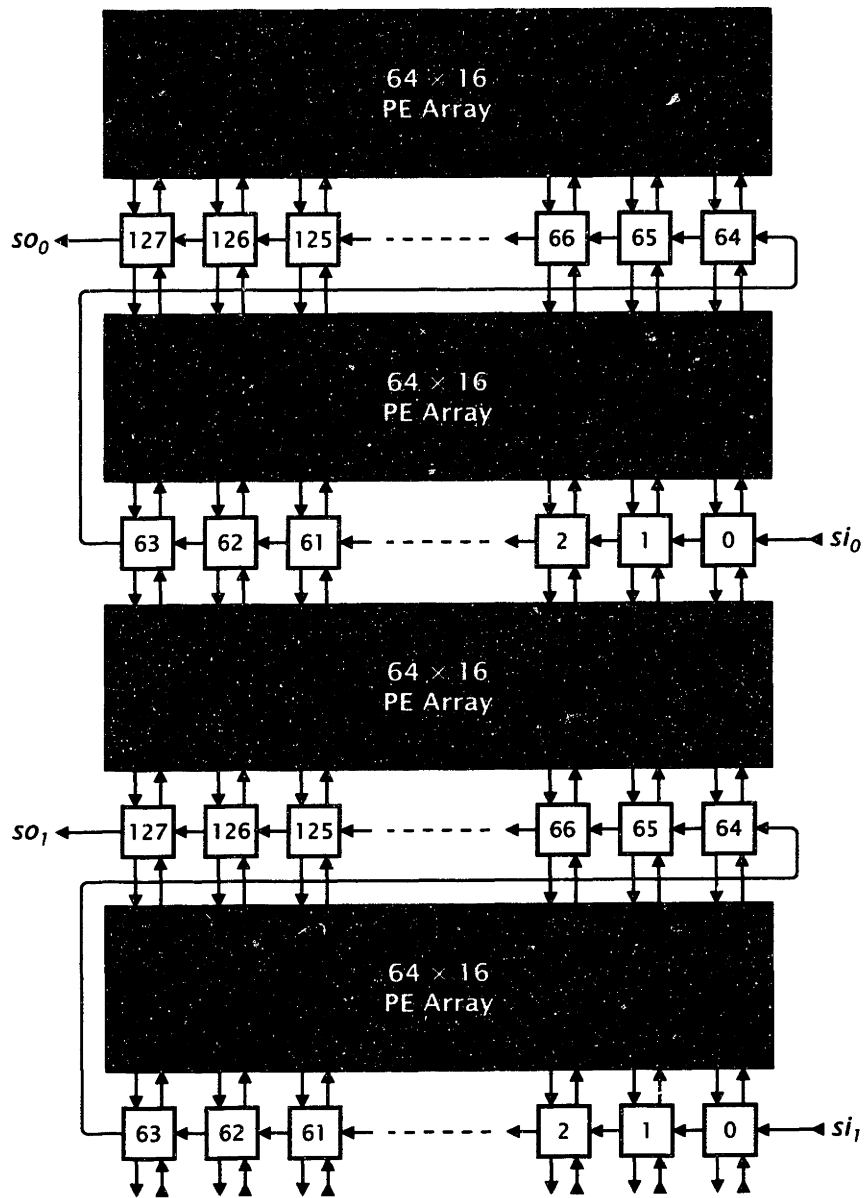


Figure 2.17: Serial-access memories.

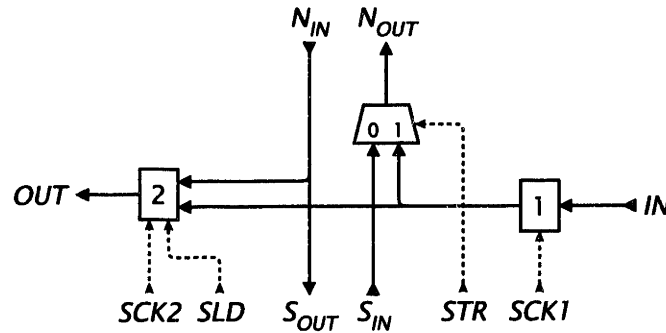


Figure 2.18: Serial-access memory cell. Solid lines represent data signals. Dashed lines represent control signals. When $SCK1$ is high, latch 1 captures signal IN . When $SCK2$ is high, latch 2 captures the output of latch 1. When SLD is high, latch 2 captures signal N_{IN} .

capacitance between each line and ground is about 300 fF. Crosstalk is therefore a significant concern.

Many of the metal lines carry internal clock signals. Glitches on these lines due to crosstalk would result in logic faults. With a simple cascaded-inverter driver circuit, crosstalk would be unacceptably large. Reducing the strength of the driver doesn't help: transitions are slowed, reducing currents, but driver output conductance is lower, so currents produce larger voltage changes.

A more complex driver circuit, shown in Figure 2.19, lowers crosstalk using feedback. The circuit has two pairs of driver devices, $M_{P1,N1}$ and $M_{P2,N2}$. Devices $M_{P2,N2}$ are three times larger than devices $M_{P1,N1}$. When the input, IN , changes state, M_{P2} and M_{N2} are both turned off. The output, OUT , is pulled up by M_{P1} or pulled down by M_{N1} . After the output transition is complete, M_{P2} or M_{N2} is turned on to assist M_{P1} or M_{N1} . Thus, the output conductance of the driver is much greater when the driver is holding its output steady than when it is switching its output.

Simulations with the simple cascaded-inverter driver and the low-crosstalk driver demonstrate the advantage of the latter design. Figure 2.20 shows results of simulating a driver holding a line low while other drivers switch neighboring lines low to high. With cascaded-inverter drivers, capacitance between neighboring lines produces a 750-mV glitch. With low-crosstalk drivers, crosstalk does not exceed 400 mV.

2.9.2 Pad Circuit

The chip has three types of signal pads: three-state pads, input-only pads, and output-only pads. Input-only and output-only pad circuits are derived from a three-state circuit. Two special supplies are used for the pad circuits: V_{HH} , a 3.3-V supply, and V_{LL} , a ground supply. These supplies carry pad input and output currents. They are distinct from V_{DD} and V_{SS} , the 2.5-V and ground supplies used for internal circuits.

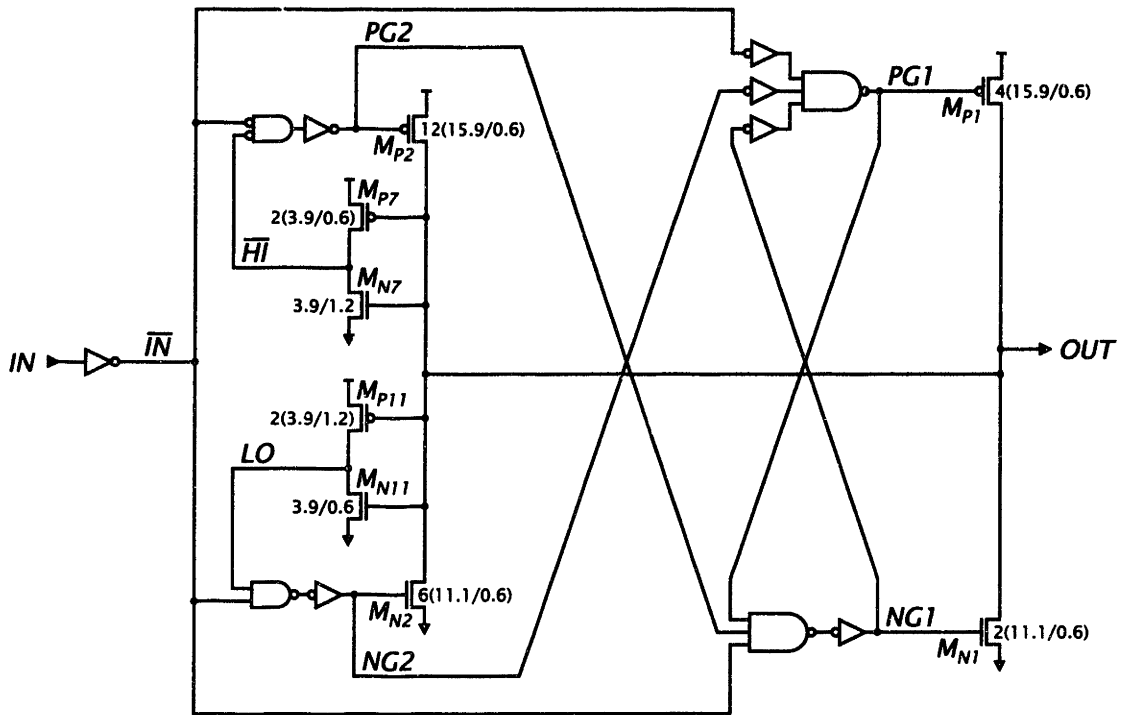


Figure 2.19: Low-crosstalk line driver circuit.

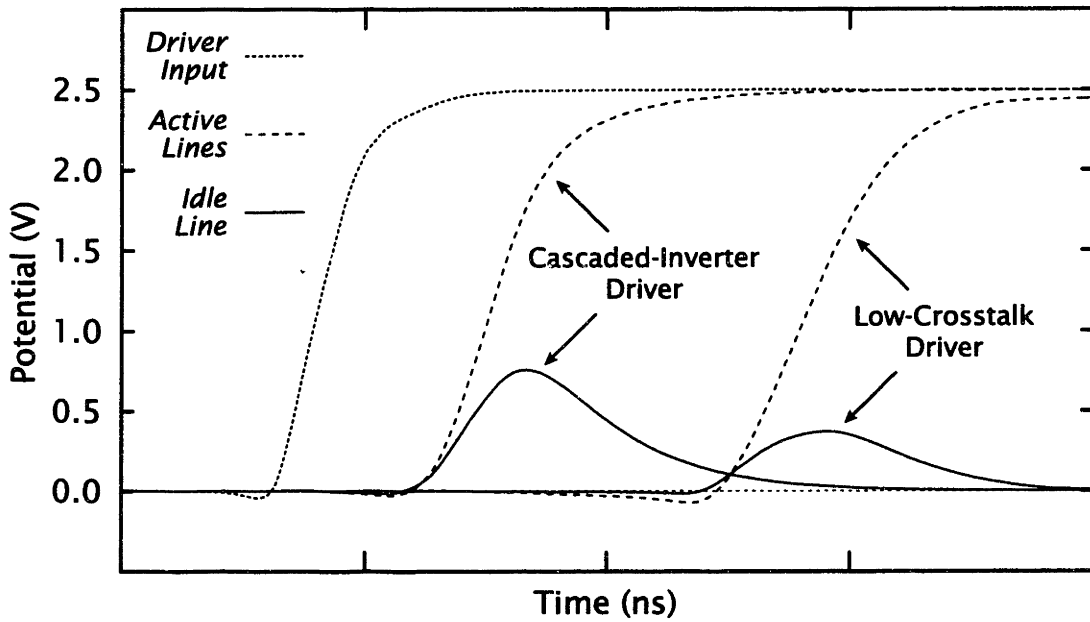


Figure 2.20: Line driver simulation.

The three-state pad circuit, shown in Figure 2.21, is based on a pad circuit developed for low-voltage chips [17]. The pad is tied to a pull-up device, M_{P1} , and a pull-down device, M_{N1} . The pad pull-up device is controlled by input signal \overline{ONE} through a level converter. The level converter comprises cross-coupled p -channel devices, $M_{P6,8}$, two n -channel pull-down devices, $M_{N6,8}$, and an inverter $M_{P10,N10}$. The pad pull-down device is controlled by input signal \overline{ZERO} through an identical level converter. When \overline{ONE} is low and \overline{ZERO} is high, the pad is pulled high. When \overline{ONE} is high and \overline{ZERO} is low, the pad is pulled low. When both \overline{ONE} and \overline{ZERO} are high, both M_{P1} and M_{N1} are off and the pad may be used to receive input signals. The fourth possible input condition, \overline{ONE} and \overline{ZERO} both low, is not useful, and would result in large currents through the pad pull-up and pull-down devices.

The relative strength of the cross-coupled p -channel devices and the n -channel level converter pull-down devices is critical. The n -channel devices must overpower the p -channel devices. The maximum gate-source voltage on the p -channel devices is 3.3 V. The maximum gate-source voltage on the n -channel devices is 2.5 V. The n -channel devices are sized so that they have over three times the saturation current of the p -channel devices, taking into account the difference in gate-source voltage.

The devices that drive the gates of pad pull-up and pull-down devices are sized so that the pull-up and pull-down devices are turned off more quickly than they are turned on. The speed difference reduces the short-circuit current that results when \overline{ONE} and \overline{ZERO} change simultaneously. The pull-up and pull-down devices are sized to insure that pad rise and fall times are less than 5 ns when the total load is 30 pF.

Two inverters, $M_{P13,N13}$ and $M_{P12,N12}$, buffer input signals from the pad and perform level conversion. The first inverter, $M_{P13,N13}$, is protected from electrostatic discharge by an n -well resistor, a thick oxide transistor with a second-level metal gate, and the diodes formed by the drain-bulk junctions of the pad pull-up and pull-down devices.

The stacks of n -channel and p -channel devices, M_{N14-21} and M_{P14-17} , were not present in the first fabrication run. They have been added to the circuit for future runs. The devices hold the pad potential low or high when all connected drivers are off. In doing so, they prevent the large short-circuit currents that would flow through inverters $M_{P13,N13}$ and $M_{P12,N12}$ if the pad potential drifted midway between V_{HH} and V_{LL} . The sizes of the devices and the number of devices in each stack were chosen so that the magnitude of the output current is about 100 mA when the pad voltage is 0.8 V and when the pad voltage is 2.0 V. These voltages are common input-low and input-high voltages for 3.3-V parts. Devices M_{N15-21} and M_{P15-17} could be replaced with a single long n -channel device and a single long p -channel device. Stacks of devices were employed so that the circuit can be more accurately simulated with models optimized for short channel lengths.

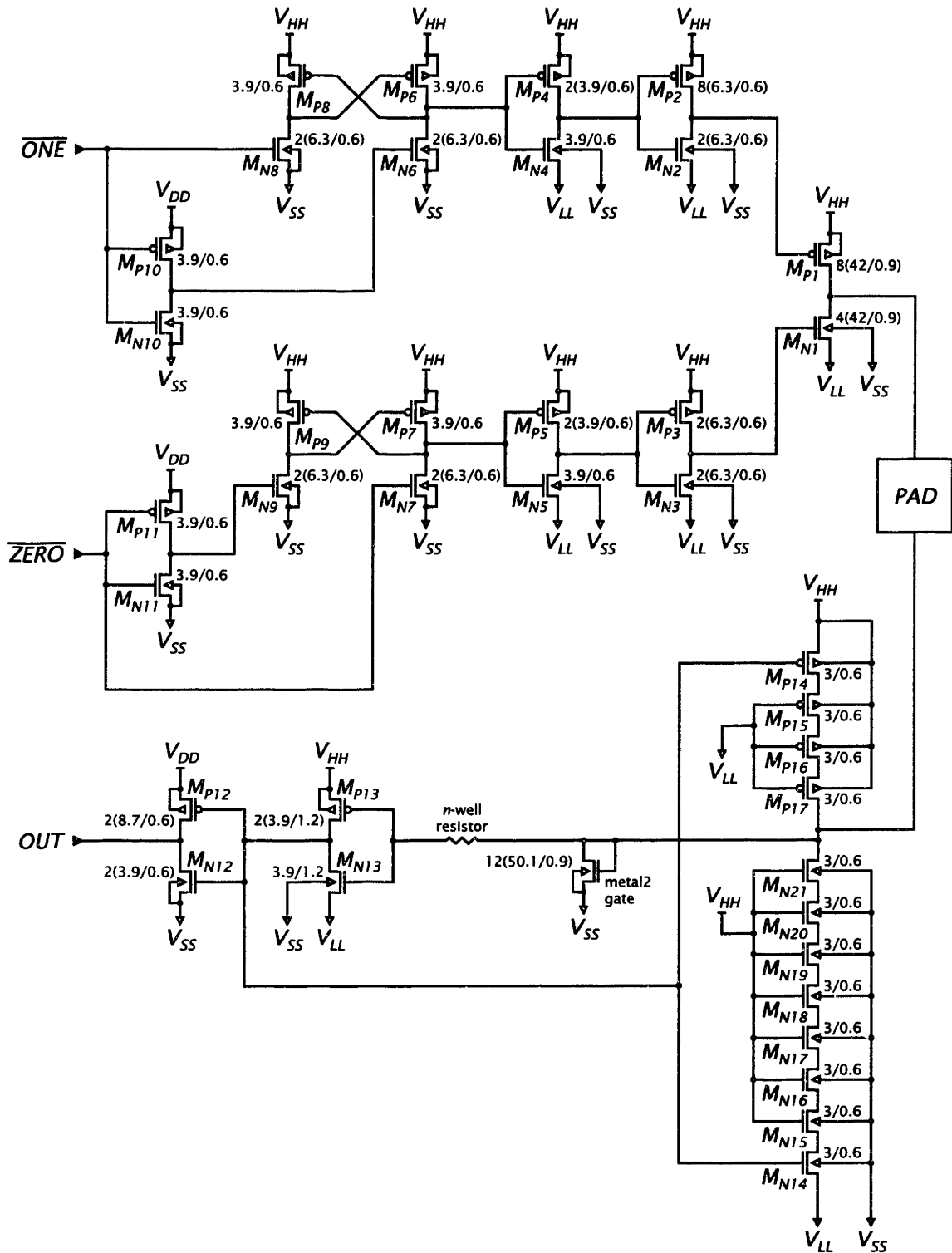


Figure 2.21: Input/output pad circuit.

2.10 Design Process

A large collection of software was employed to complete the integrated circuit design work. As preliminary architecture plans were completed, Cadence Design Systems made a large suite of tools available to students at the M.I.T. Microsystems Technology Laboratories. The design flows used for the integrated circuit work were based on many of the Cadence tools.

The pitch-matched circuits required full-custom design. It was seldom difficult to conceive circuits that simply performed the required functions, but it was often difficult to create circuits that also met the pitch constraints. The design of each circuit began with schematic drawings on paper. Then time-consuming layout development was performed using Magic, a layout editor developed at the University of California at Berkeley [18]. Stable technology files provided by MOSIS were used with a few simple modifications. Magic provides a fine user interface and continuous design-rule checking. These features make it an excellent tool for creative work. However, there are no convenient facilities for verifying the consistency of Magic layout data and corresponding schematic diagrams.

When each circuit layout was completed, a Magic format conversion facility was used to produce a layout description in Stream format. Then a Cadence format conversion facility was used to read the Stream data and incorporate the layout into a Cadence Design Framework II library. The Cadence Virtuoso layout editor was used to clean up the results of the translation process and to add connectivity information.

After finishing the work with Virtuoso for each circuit, a schematic diagram was prepared using the Cadence Composer schematic editor. Then the Cadence Diva design rule checker, layout parameter extraction, and layout versus schematic tools were used to verify the circuit design. The Cadence Analog Artist environment was used to produce a netlist from the extracted layout for the Meta-Software HSPICE circuit simulator. Simulation results were used to verify circuit functionality and adjust transistor sizes.

Arranging and verifying groups of processing elements required a different design flow. Starting with a group of four processing elements, a block of 512 processing elements was implemented using six levels of hierarchy. Schematic diagrams for each level were laboriously prepared using Composer. Groups of cells were physically arranged and connectivity information was added using Virtuoso. Each level was verified using Diva.

The Cadence SKILL programming language was used to physically arrange blocks of 512 processing elements and add connectivity information. SKILL was used in the same manner for pad circuits. Lines connecting perimeter circuits to the blocks of processing elements were routed by hand using Virtuoso.

Full-chip verification was performed using the Diva tools on a Sun Ultra 1 Model 140 workstation with 196MB RAM and 2.5GB disk space for virtual memory and temporary files. A hierarchical design rule check took about seventeen hours. The consistency of the layout and the schematic diagrams was verified, with memory cells removed, in

Table 2.5
Layout Area per Processing Element

Processing Element Memory		7794 μm^2
DRAM Column	6635 μm^2	
V_{PL} Bypass Capacitor	570	
Sense Amplifier	589	
Processing Element Logic		3437 μm^2
Write Driver	220 μm^2	
Latch A	452	
Interconnect Logic	810	
Function Generator	890	
Latches B, C, D, & E	672	
Write Logic	393	
Serial-Access Memory		224 μm^2
V_{DD} Bypass Capacitor		1247
Other (timing & driver circuits, pads, etc.)		6520

about six hours.

2.11 Summary

This chapter described the implementation of the processing element array. Compact logic circuits are pitch-matched to DRAM cells to form dense processing elements. Table 2.5 shows the the total layout area used by each component of the processing elements. Processing element logic accounts for less than 20% of the total area. Operation of the logic and memory is coordinated with only two clock inputs, no more than are needed by DRAM chips.

To minimize power dissipation, bitlines are precharged to $V_{DD}/2$ and internal circuits operate with a 2.5-V supply. Actual power dissipation depends not only on the instruction mix, but also on the input data. Table 2.6 provides power dissipation estimates for typical operation.

The processing elements are interconnected to form a 64×64 array. The processing element interconnection network extends across chip boundaries. A regular wiring pattern maps the physical processing element arrangement to a square array, minimizing the bandwidth required for interchip communication. To reduce the number of pads, interchip communication signals are multiplexed. Serial-access memories, incorporated into the interconnection network, provide an efficient means of transferring image data to and from the array.

The final design was submitted to MOSIS in July 1996. Parts were received in October 1996. Figure 2.22 is a photomicrograph of the chip. Of the 144 pads, 40 are used for

Table 2.6
Estimated Power Dissipation

Processing Element Memory		120 mW
Processing Element Logic		104
Latch A	34 mW	
Interconnect Logic	13	
Function Generator	33	
Latch B	5	
Latch C	6	
Latch D	4	
Latch E	< 1	
Write Logic	9	
Interchip Communication		44 mW

supply connections, 64 are used for interchip communication, and 8 are used for image I/O. The remainder are used for clocks and array instructions. Chips are packaged in ceramic pin grid arrays.



Figure 2.22: Chip photomicrograph.

Chapter 3

System Design

This chapter describes how the integrated processing element array is applied to form the real-time image processing system. Section 3.1 discusses the data path, focusing on the architecture of the format converters. Section 3.2 discusses the control path, including the controller architecture. Section 3.3 discusses system software.

The work presented in this chapter is the product of partnerships with other graduate students. The design of the integrated processing element array described in this thesis overlapped the development of an integrated array employing associative memory cells [19, 20]. The control path and system software designs support both integrated circuit architectures [21]. The first controller implementation [22] was used to test and demonstrate the associative processor. The format converter implementation [23] and an improved controller implementation [24] are part of the current system.

3.1 Data Path

The integrated processing element array presents two major data path design challenges. One is common to multiple-chip image processing arrays. The other is common to bit-serial components. This section discusses the two challenges and the actual data path design.

3.1.1 Image Organization

A digital image may be viewed as a three dimensional array of bits, with one dimension corresponding to image rows, one dimension corresponding to image columns, and the third dimension corresponding to one-bit components of the pixel representation. To build a large frame buffer, it is natural to use one chip per bit-plane. Pixel data may be transferred to and from the memory one pixel at a time with the work equally distributed among chips operating simultaneously. In contrast, to build a large image processing array, it is natural to divide images into blocks of pixels and use one chip per block.

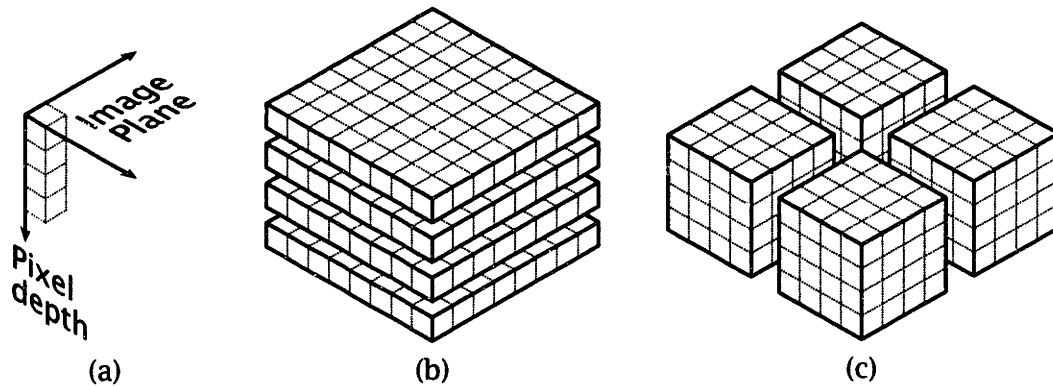


Figure 3.1: Image organization. (a) A single four-bit pixel. (b) Planar partitioning: one chip per bit-plane. (c) Spatial partitioning: one chip per block.

That way, processing tasks may be performed with the work equally distributed among chips operating in parallel.

The two organizations are illustrated in Figure 3.1. Planar partitioning, typical of frame buffers, divides a 4-bit 8×8 image into four bit-planes, each handled by a single chip. Spatial partitioning, appropriate for parallel image processing, divides the image into four blocks, each handled by a single chip.

Most imagers, designed for compatibility with television systems, provide data row-by-row, starting with the top row and ending with the bottom row. Within a row, imagers provide pixel values sequentially from left to right. Planar partitioning works well with this order. But with spatial partitioning, only one chip would be active at a time. Thus, the I/O bandwidth of each chip would need to match the output bandwidth of the imager. To efficiently transfer image data to multiple-chip processing element arrays, input data must be reordered and divided into multiple sequences, with each sequence serving a single chip. In that way, the I/O bandwidth of each chip may be a fraction of the I/O bandwidth of the imager.

3.1.2 Corner-Turning

The integrated processing element array, like other bit-serial components, operates on data one bit-plane at a time. Bit-parallel components, including video digitizers, video encoders, and microprocessors, operate on data one value at a time. In a practical system, the processing element array must communicate with bit-parallel components. One might argue that functions performed by bit-parallel components could be performed by bit-serial components. But the argument is moot. Bit-serial components are simply not available.

To efficiently transfer data to and from the processing element memory, “corner-turning” hardware is required. Input image data provided one pixel at a time must be presented to the processing element array in bit-plane order. Processed data from the

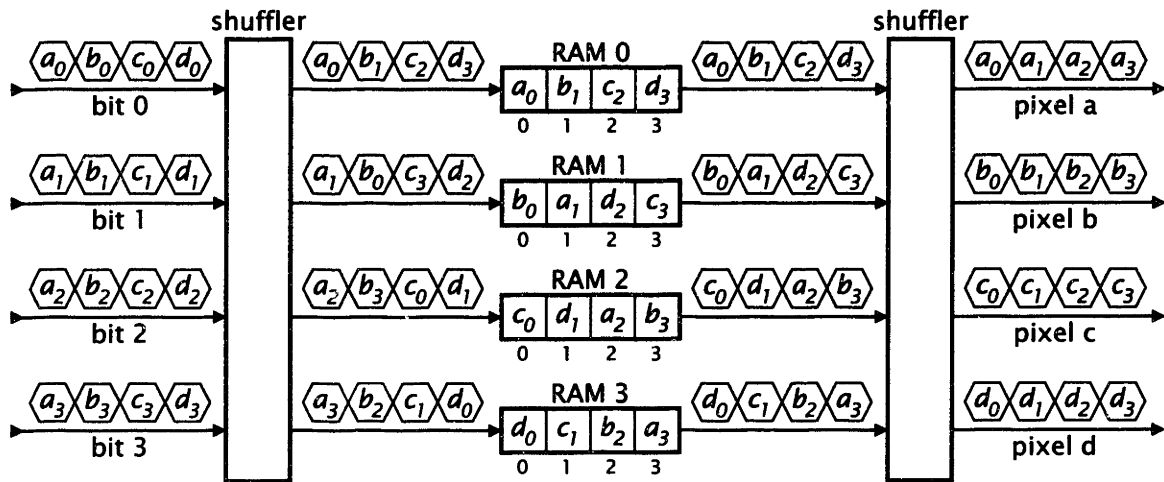


Figure 3.2: Four-bit multidimensional access memory.

array in bit-plane order must be presented to subsequent system components one pixel at a time. The STARAN bit-serial supercomputer system included a multidimensional access (MDA) memory to perform corner-turning [25]. The MDA memory architecture provides a basis for data path design.

Figure 3.2 shows a small MDA memory that reorders 4-bit, 4-pixel image data. The MDA memory comprises four 1-bit random-access memories and two “shufflers.” It accepts data one pixel at a time and provides data one bit-plane at a time.

The MDA memory accepts sequences of four pixel values, a_{3-0} , b_{3-0} , c_{3-0} , and d_{3-0} . When a is presented, a shuffler steers bit 0 to RAM 0, bit 1 to RAM 1, bit 2 to RAM 2, and bit 3 to RAM 3. RAM 0 stores pixel a data in location 0, RAM 1 stores pixel a data in location 1, RAM 2 stores pixel a data in location 2, and RAM 3 stores pixel a data in location 3. When b is presented, the shuffler steers bit 0 to RAM 1, bit 1 to RAM 0, bit 2 to RAM 3, and bit 3 to RAM 2. RAM 1 stores pixel b data in location 0, RAM 0 stores pixel b data in location 1, RAM 3 stores pixel b data in location 2, and RAM 2 stores pixel b data in location 3. Values c and d are stored as indicated in the figure.

After four values are stored in the RAMs, the MDA memory provides a sequence of four bit-plane data, $\langle a_0, b_0, c_0, d_0 \rangle$, $\langle a_1, b_1, c_1, d_1 \rangle$, $\langle a_2, b_2, c_2, d_2 \rangle$, and $\langle a_3, b_3, c_3, d_3 \rangle$. The memories read bit 0 data from location 0, bit 1 data from location 1, bit 2 data from location 2, and bit 3 data from location 3. A shuffler steers data to the appropriate output lines.

As shown in, Figure 3.3, a 4-bit shuffler may be implemented using two levels of logic, with four selectors in each level. An n -bit shuffler requires $\log_2 n$ levels of logic, with n selectors in each level. The logic required to produce addresses for the four RAMs is simpler. When storing data, the addresses for RAMs 1, 2, and 3 can be produced by inverting appropriate bits of the RAM 0 address. For example, bit 0 of a RAM 2 address is the same as bit 0 of the corresponding RAM 0 address. Bit 1 of a RAM 2 address is

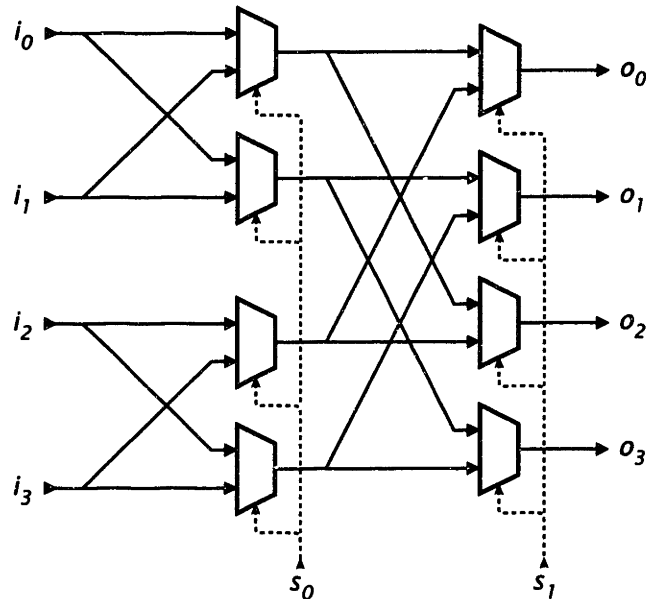


Figure 3.3: Four-bit shuffler. Four selectors are controlled by signal s_0 and four by signal s_1 .

the inverse of bit 1 of the corresponding RAM 0 address.

3.1.3 Design

Two features guided data path design. First, the data path supports real-time image processing using a standard CCD camera to provide input images, and using an NTSC display to show processed images. Second, the data path supports complete functional testing of the integrated processing element array using the host computer to provide input images and to verify processed images. The data path is designed for 8-bit gray-scale pixels. Aside from the processing element array, the two format converters are the principal components of the data path. They handle four-chip 64×64 processing element arrays and sixteen-chip 256×256 arrays.

As shown in Figure 3.4, the format converters are implemented using an MDA memory structure. The input format converter accepts image data from an NTSC video digitizer or from the host computer through the VMEbus interface. The input format converter provides raw images to the processing element array. The output format converter accepts processed images from the array and provides image data to an NTSC video encoder and to the host computer through the VMEbus interface. The digitizer is connected to a camera. The encoder is connected to a display.

The format converters are more sophisticated than plain MDA memories. The input format converter performs corner-turning and, at the same time, divides data into multiple sequences, with each sequence serving a single serial access memory. The output

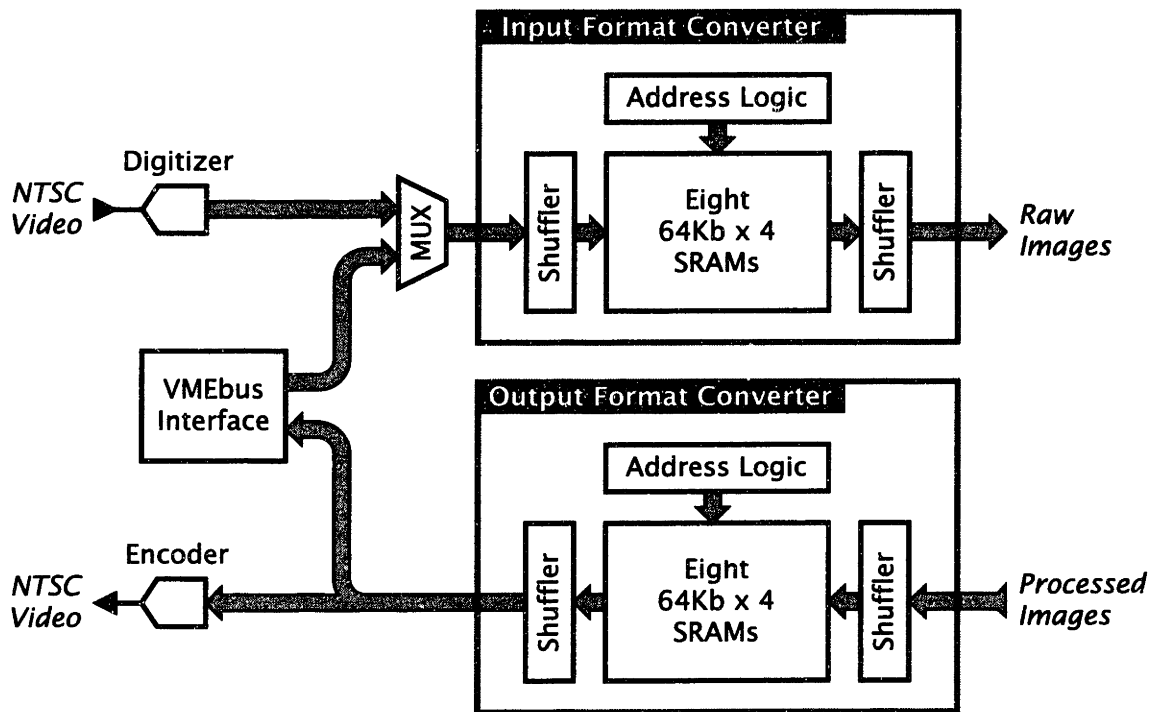


Figure 3.4: Format converters.

converter performs corner-turning and produces a standard row-by-row sequence of pixel values.

Only one multidimensional memory structure is needed for each format converter. The input format converter accepts data while the camera provides valid picture information and provides data to the processing element array during the vertical blanking interval. The output format converter accepts data from the array during the vertical blanking interval and provides data to the encoder when valid picture information is required. Because the vertical blanking interval is a small portion of the field interval, a much higher data rate is required for transfers to and from the array than for transfers from the digitizer or to the encoder. Therefore, while 8-bit paths are used for connections from the digitizer and to the encoder, 32-bit paths are used for connections between the array and the format converters.

3.2 Control Path

The RAM-based processing element array described in this thesis and the associative processing element array are both single instruction stream, multiple data stream (SIMD) designs. While the processing elements include both memory and logic, they do not have control structures. Instead, the processing elements share instructions received from a single controller. The two array designs each have only one, minimally encoded,

instruction format. The associative processing element array is designed to execute an 83-bit instruction every 100 ns. The RAM-based processing element array executes a 27-bit instruction every 60 ns. Therefore, to support the two designs, the control path must deliver nearly 1Gb per second to the array. Flaws in control strategy may result in a system which does not efficiently utilize the processing element array. Whenever the control path falls behind, the array sits idle.

3.2.1 Control Strategy

To minimize system cost, one might consider using software executed on a host computer to directly generate low-level instructions executed by the processing element array. In this approach, low-level instructions are transferred over the host's backplane bus. Unfortunately, there are two serious problems with direct sequential control. First, the host computer may not be able to compute instructions rapidly enough to keep the array busy. Second, the rate at which the host can deliver instructions to the array is limited by bus transaction delays.

To reduce the bus bandwidth and host computation required to sustain array activity, SIMD supercomputers typically employ a multi-level control hierarchy. In a conventional two-level design, a microcontroller is placed between the host and the processing element array. The microcontroller executes *microcode*, interpreting *macroinstructions* issued by a host and transferred over the host's backplane bus. The microcontroller produces the instructions executed by the array. Typical macroinstructions produce many array instructions. Thus, the demands on the host computer and bus are reduced. The Connection Machine [26] and the Massively Parallel Processor [27] employ two-level designs. The associative string processor (ASP) testbed developed by Aspex Microsystems [28] and the Vastor processor [29] employ three-level control hierarchies, adding an additional level of interpretation between the host computer and the microcontroller.

The microcontroller must handle a variety of different macroinstruction formats. Macroinstructions generally include both an operation code and one or more arguments. Typical macroinstructions specify a basic arithmetic, comparison, or data movement operation. Arguments identify fields of processing element memory or scalar values. The number of arguments and the types of arguments included in a macroinstruction depends on the operation. For example, the *add immediate* operation, $A \leftarrow A + b$, involves a field, A , and a scalar value, b , while the *sum* operation, $S \leftarrow A + B$, involves three fields, A , B , and S .

After decoding a macroinstruction, the microcontroller must produce a proper series of array instructions. Usually, bit-serial procedures may be generated using a fixed pattern of array instructions for intermediate bit positions. But handling the lowest and highest bit positions is often a more difficult task. Consider the sum operation. Corresponding instructions for intermediate bit positions vary only in the effective locations of active memory cells. But the least significant bit position must be handled differently since there is no carry data from a previous bit position. More importantly,

accommodating field length differences requires some sophistication. If the length of the sum field exceeds the lengths of the addend fields, sums must be sign-extended. If the lengths of the operand fields differ, the array instructions produced for the highest bit positions must provide the effect of sign-extending the shorter operands.

While the conventional hierarchical control strategy may be suitable for SIMD supercomputers, it is not appropriate for less expensive systems. One drawback is that to generate array instructions at a rate commensurate with the speed of the processing element array, a sophisticated, fast microcontroller is needed. The microcontroller design must include one or more functional units to perform the arithmetic and logical operations involved in producing array instructions. Given the amount of computation generally required to decode macroinstructions and produce array instructions, the frequency of the controller clock must be several times that of the array clock.

A second drawback to conventional hierarchical control is the burden of providing appropriate software support. With both a control path and one or more functional units, the microcontroller amounts to a special-purpose computer. Separate programs are required for the host and the controller. Getting these two programs to work together can be more than twice as hard as writing a single program. The microcontroller also requires its own set of development tools, including a compiler and a debugger.

In both the direct control strategy and the conventional hierarchical control strategy, sequences of array instructions are computed on-the-fly during program execution. In the first strategy, the instructions are computed by a host computer, limiting array utilization. In the second strategy, the instructions are computed by a microcontroller, necessitating sophisticated hardware. In order to achieve high array utilization without a complex microcontroller, run-time instruction computation must be avoided.

Fortunately, run-time instruction computation is not needed for real-time image processing. The same low-level tasks are repeated for each image. An efficient control strategy exploits this property. Sequences of array instructions are generated by the host computer before processing begins and are stored in a controller. Macroinstructions are reduced to simple calls telling the controller which sequence to send to the processing element array. The controller can be simplified because it does not have to decode and interpret complex macroinstructions.

Figure 3.5 illustrates the controller architecture. Sequences of microinstructions, generated by the host, are held in the control store. Each microinstruction includes two array instructions, a sequencer instruction, and a branch address. To initiate a sequence of microinstructions, the host computer writes the starting address of the sequence into the opcode register. The sequencer steps through the control store, producing one array instruction every clock cycle. The select register and the associated multiplexer are used to support scalar variables.

Basic sequences, generated by library functions, perform simple arithmetic, comparison, and data movement operations. These sequences are generally ten to sixty array instructions long. Application implementations often concatenate basic sequences to form larger sequences, reducing the amount of interaction between the host computer

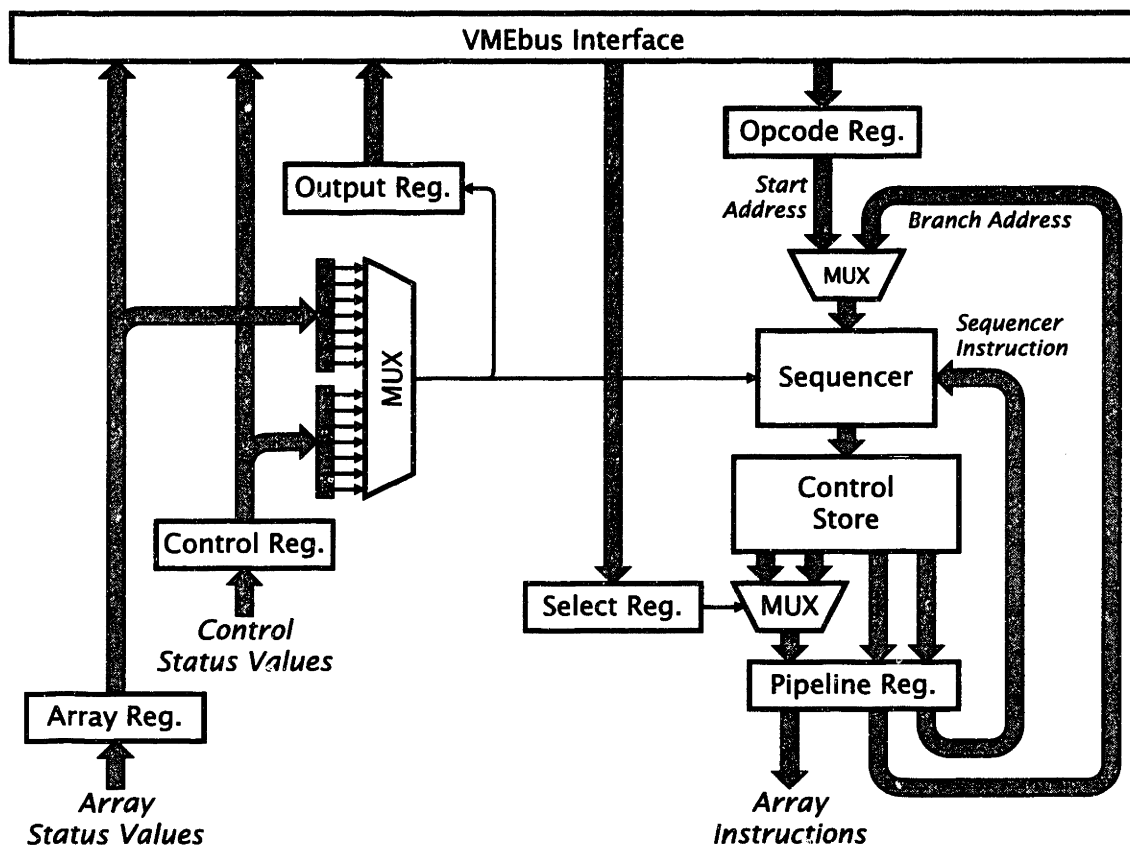


Figure 3.5: Controller.

and the controller.

The array register and the output register provide feedback paths from the array to the controller and to the host computer. Array status values are stored in the array register. Within the controller, the status values may be used by conditional branch instructions. In addition, using the array register, the host computer can directly obtain array status information. Alternatively, status values may be accumulated in the output register, a serial-in, parallel-out shift register. This feature is often employed to assemble the results of bit-serial associative operations.

The control strategy has several important features:

- *Simple controller hardware.* Since array instructions are generated by the host computer, the controller need not perform arithmetic and logical operations. Thus, the controller does not include a data path. Instead, array instructions are merged into the control path. Furthermore, the controller need not operate with a higher clock frequency than the processing element array.
- *High array utilization.* During processing, the host computer need not issue individual instructions to the processing element array. Thus, provided that sequences are sufficiently long, array utilization is not unacceptably limited by host speed or by bus transaction delays.
- *Unified software development.* All system software can be created and compiled on the host computer with existing software development tools.

3.2.2 Instruction Selection

SIMD processing involves two kinds of data: scalar data stored in the host computer or controller, and parallel data (fields) stored in the processing element array. A SIMD system must provide a mechanism for incorporating scalar values into operations performed by the processing element array. Examples of operations employing scalar values include addition of a scalar value to a field, identification of field elements with values less than a scalar value, and assignment of a scalar value to a field.

The host computer may handle scalar constants as it produces sequences of array instructions. But, by definition, values of scalar variables are not fixed before processing begins. Consider the addition of an n -bit scalar variable, stored in the host computer, to an n -bit field. In real time, the host computer must evaluate the scalar variable and invoke an appropriate sequence of array instructions to perform the addition. A different sequence is used for each of the 2^n possible scalar values. But the amount of controller memory necessary to store all 2^n sequences is unacceptable (unless n is very small).

Fortunately, the number of stored sequences may be reduced by dividing the add immediate operation into n steps, with each step handling one bit position. For each bit position, only two scalar values may be encountered, 0 and 1. Thus, instead of storing 2^n sequences, one can store $2n$ shorter instruction sequences, two for each

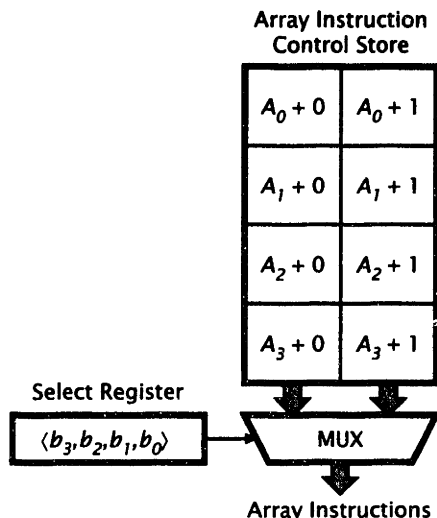


Figure 3.6: Instruction selection. $A \leftarrow A + b$. $A = \langle A_3, A_2, A_1, A_0 \rangle$, $b = \langle b_3, b_2, b_1, b_0 \rangle$.

bit position. Other arithmetic and comparison operations, performed using bit-serial procedures, can be decomposed in a similar manner.

Typical arithmetic and comparison operations employing scalar values require sequences of one to three array instructions per bit position. Using the host computer to call such short sequences of instructions individually would result in poor array utilization due to bus transaction delays. A simple hardware mechanism called instruction selection provides an alternative. Instead of calling sequences individually, the host computer writes scalar values to the select register in the controller and the controller picks the appropriate sequences.

Figure 3.6 shows how instruction selection is employed to add a 4-bit scalar variable, b , to a parallel variable, A . Before processing begins, four pairs of instruction sequences are stored in the controller. Each microinstruction contains a pair of array instructions. During program execution, the value of b is loaded into the select register, a parallel-in, serial-out shift register. As the sequencer steps through the control store, the shift register output selects one array instruction from each microinstruction. After the last microinstruction for each bit position, the select register value is shifted to the right. Suppose the value of b is binary 1101. The first sequence executed is the one at the top right, which adds 1 to the least significant bit. The second sequence (on the left) propagates the carry to the second bit, and the final two sequences (on the right) add 1 to the third and fourth bit positions. A distinct multiplexer is not necessary. Instead, the multiplexer function is implemented by using the select register output as one bit of the array instruction control store address.

In practice, some fields of array instructions may not depend on scalar values. Thus, it may not be necessary to include two complete array instructions in each microinstruction. For example, each microinstruction may include two values for the array

instruction field specifying function generator operations, but only one value for all other instruction fields. Of course, such optimizations depend on details of the array architecture, instruction format, and sequence implementation.

Array architectures may be based on processing elements with a 1-bit immediate input, eliminating the need for instruction selection. Instead, the select shift register output may be used as one bit of the array instruction. This approach was rejected because it would degrade associative processor performance. For example, using an immediate scalar input, the associative processor would require five instructions per bit position to perform the add immediate operation. Using instruction selection, only three instructions per bit position are necessary.

3.3 Programming Framework

Very fine-grained parallel processors with one-bit-wide processing element logic present challenging software problems. Instruction sets are very primitive: simple parallel arithmetic, comparison, and data movement operations require sequences of many array instructions. In addition, different processing element array implementations employ different memory structures, provide different instructions, and require different computational algorithms. To facilitate application development and maintenance, a programming framework was created. The framework hides details of array and controller implementations, allowing programmers to focus on application issues.

The programming framework uses the C++ programming language [30]. C++ provides facilities for defining new types (classes) that act in the same way as built-in types. In effect, these facilities allow the language to be augmented to support additional concepts. The framework is implemented as a library of C++ classes. Figure 3.7 outlines key components of framework.

Application code need not be tied to a particular array implementation, controller implementation, or host computer. The current system software includes support for the controller and the processing element array described in this thesis. The controller architecture is labeled "MIT." The array is labeled "HDPP" (high-density parallel processor). The software also includes support for the associative processor. The associative processor architecture, employing content-addressable memory cells, is labeled "CAPP" (content-addressable parallel processor). In the future, support may be added for other array and controller architectures without altering existing system code. Array and controller architectures are represented by abstract classes. No objects of the architecture classes may exist, but the classes may be used as base classes for derived classes.

The programming framework supports use of three data types in processing element arrays: signed integers, unsigned integers, and ternary values. In the future, support may be added for rational numbers. Support for floating-point numbers is not anticipated. With a small amount of memory per processing element and no floating-point hardware, the array architectures are not intended for applications employing floating-point numbers.

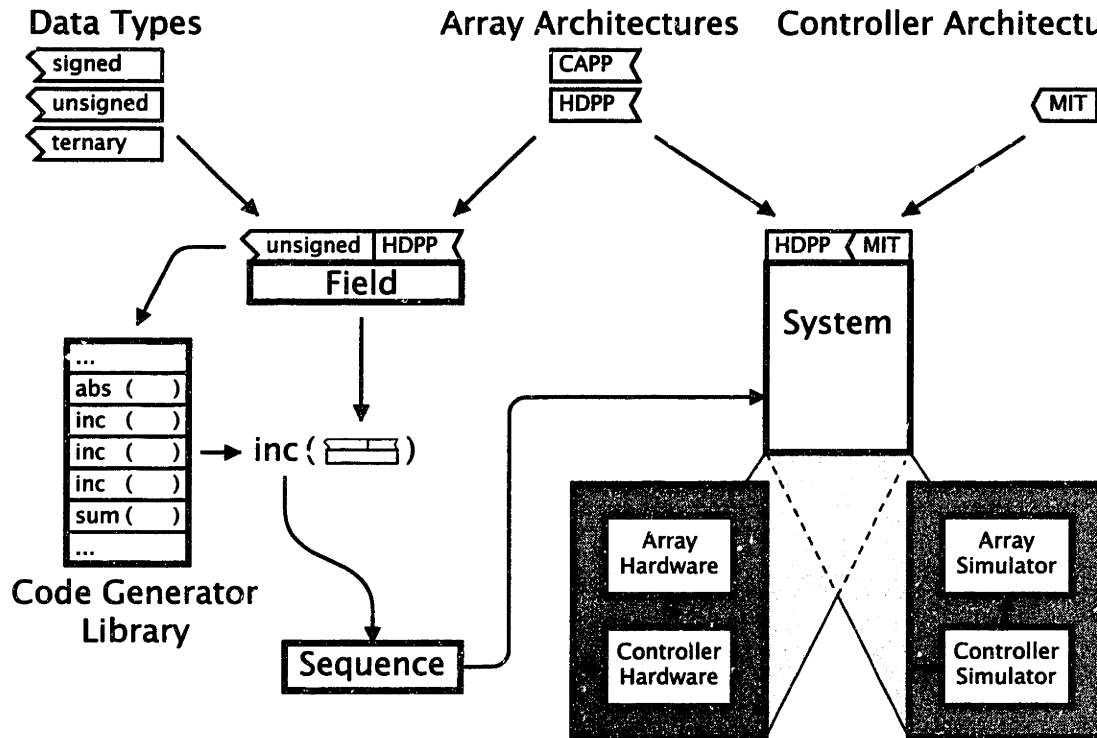


Figure 3.7: Programming framework.

Parallel variables, stored in the processing element array, are represented by *field* classes. Three field classes are derived from each array architecture class. One class represents fields containing unsigned integers, a second class represents fields containing signed integers, and a third class represents fields containing ternary values. Field class implementations include management of processing element memory.

A processing element array coupled with a controller is represented by a *system* class. Systems classes are implemented using an interface to array and controller hardware or using array and controller simulation code. The simulation option provides a valuable tool for architecture development work. An object of class *sequence* represents a set of controller and array instructions which may be loaded into a region of the control store on the controller and invoked within an application. Application programs employ processing element arrays by directing the execution of sequences by systems.

A *code generator* is a function used to produce sequences. In general, code generators have one or more field parameters. For example, *inc* has a single field parameter and returns a sequence that, when executed by a system, increments the values contained in the field. Employing code generators, application code need not include low-level array and controller instructions.

An organized code generator library provides support for using the same application code with different array architectures. Code generators that produce sequences which perform identical tasks are given the same name. For example, all code generators

which produce increment sequences are given the name `inc`. This practice is called overloading. When application code uses an overloaded name, the compiler selects the appropriate function by comparing argument types with parameter types. Application code can be adapted for use with different architectures by changing field types to reflect the appropriate association between fields and array architectures. There is no need to modify code generator calls or sequence execution patterns.

3.4 Summary

The pixel-parallel image processing system comprises four main components: the processing element array, the format converters, the controller, and the programming framework. This chapter described the latter three components. The format converters, implemented using multidimensional access memory structures, provide an efficient interface between the bit-serial processing element array and conventional bit-parallel components. The controller design, using simple hardware, issues instructions to the processing element array at rates matching the speed of the array. The programming framework hides low-level system details, allowing software developers to focus on implementation issues.

Chapter 4

Experimental Results

This chapter is devoted to practical matters: circuit and system testing, characterization, and performance. Section 4.1 discusses the construction of the system used to test the integrated array and to demonstrate pixel-parallel image processing. Section 4.2 and Section 4.3 provide results of control path and array characterization. Section 4.4 presents measured performance of the complete system for three typical low-level image processing operations.

4.1 System Construction

Figure 4.1 shows the principal components of a system used to test and demonstrate the integrated circuit devices. A Sun SPARCstation IPX, a UNIX workstation, manages the system. A Performance Technologies PT-SBS915 adapter links the workstation to a VMEbus chassis. The adapter comprises an SBus board installed in the workstation, a VMEbus board installed in the chassis, a cable that connects the boards, and device driver software for the workstation.

Two format converters are integrated on a single VMEbus board along with a video digitizer and a video encoder [23]. The digitizer accepts an NTSC video signal from a CCD camera. The encoder provides an NTSC video signal to a display. Four chips on a board fitting the VMEbus chassis provide a 128×128 processing element array. The array board accepts raw images from one format converter and provides processed images to the other format converter. A controller, implemented as a VMEbus board, issues instructions for the processing element array [24]. The host computer communicates with the format converter board and the controller board through the VMEbus adapter.

The controller board employs a 16-bit microprogram sequencer, three SRAM modules, two complex programmable logic devices, ten register chips, and some clock driver, bus interface, line driver, and line receiver parts. Sequencer instructions and branch addresses are stored in one of the three $64\text{Kb} \times 32$ SRAM modules. Array instructions are stored in the other two modules. The controller and the processing element array oper-

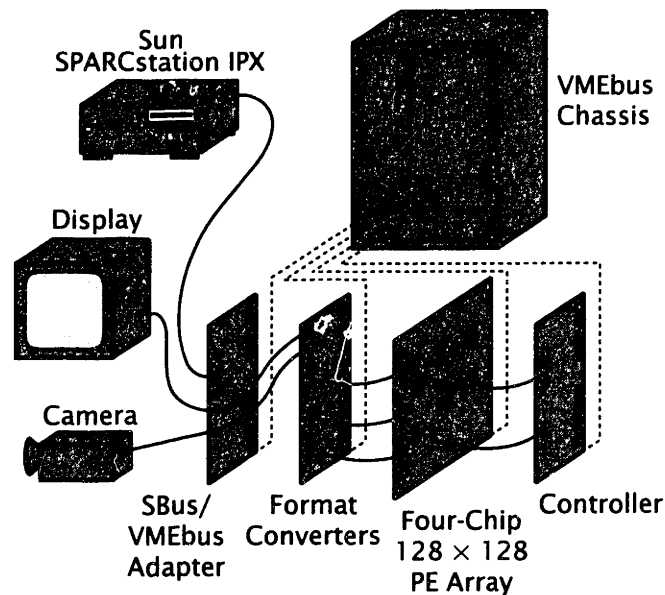


Figure 4.1: Test and demonstration system.

ate synchronously. The controller is fully functional with clock cycles 50 ns or longer.

The format converter board employs two field-programmable gate arrays (one for each format converter), sixteen $64\text{Kb} \times 4$ SRAM chips (eight for each format converter), a digitizer, an encoder, and some clock driver, bus interface, line driver, and line receiver parts. The format converter board operates at about 25 MHz, using a clock signal produced by the digitizer. The serial-access memories used to transfer data to and from the processing element array and the format converter board operate synchronously.

The processing element array board is shown in Figure 4.2. The principle design objective was to facilitate testing and characterization of processing element devices. Physical size was not a primary concern. The board has four sockets for four chips, the minimum number required to fully test interchip communication. The board is powered by Hewlett-Packard 6628A and 6629A supplies. Four $50\text{-}\Omega$ BNC connectors are used for clock inputs provided by Hewlett-Packard 8112A and 8161A pulse generators. The board is connected to the format converter board through right-angle headers on the left edge. The board is connected to the controller board through four straight headers.

4.2 Control Path Characterization

A simple test routine was developed to characterize the performance of the control path. The routine creates a sequence of array instructions, loads the sequence into the controller's control store, then repeatedly calls the sequence. Since the routine does not require use of status values or images from the processing element array, the control path may be characterized independent of processing element devices. Control

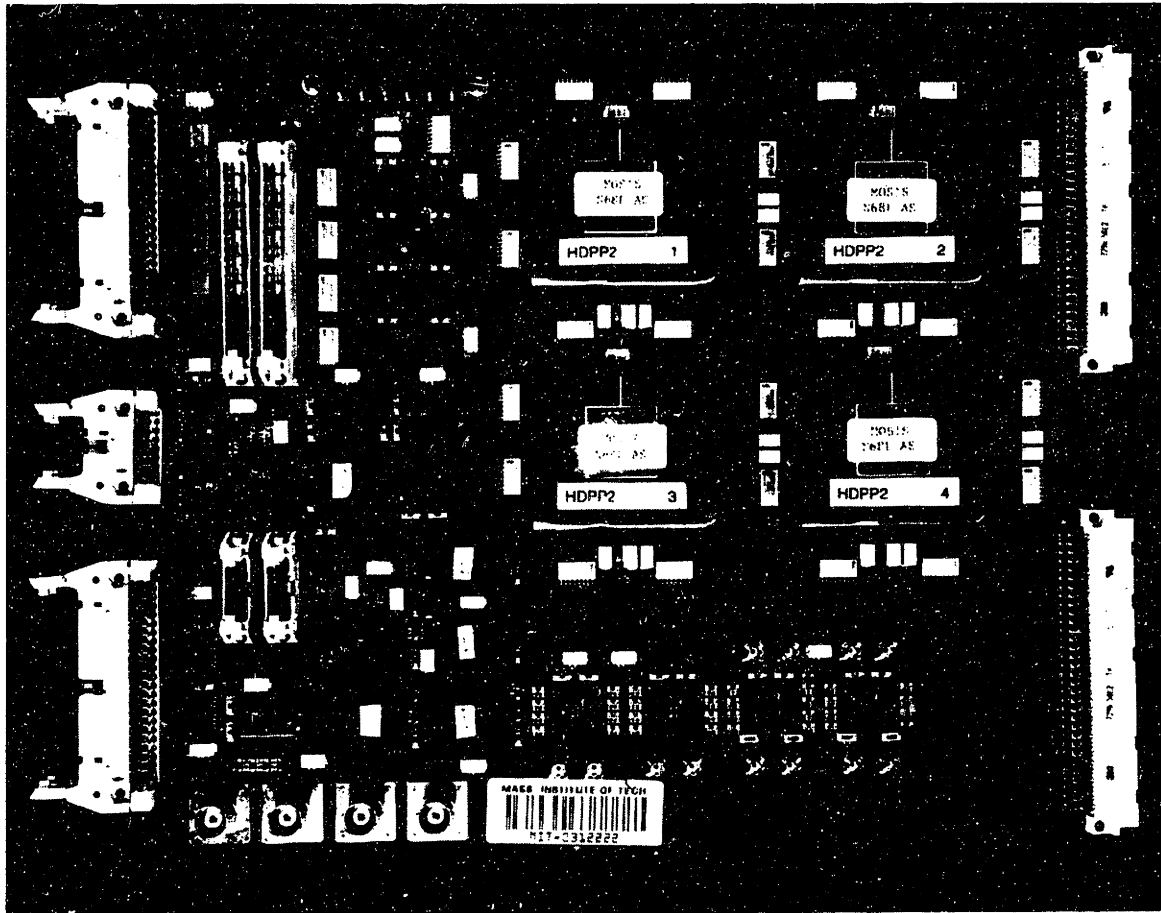


Figure 4.2: Four-chip 128×128 processing element array board.

path performance was measured with a 60-ns controller clock cycle, corresponding to the maximum speed of the processing element devices. Results were collected for sequences with lengths ranging from 1 to 1024 instructions.

Figure 4.3(a) shows the experimental results. Note the knee in the curve. For short sequences, about 2.2 μ s elapses each time the sequences are called by the host computer. This reflects the amount of time required to transmit starting addresses from the host computer to the controller. For longer sequences, the time required to issue each sequence exceeds the time required to transmit starting addresses. Thus, execution time increases with sequence length.

Figure 4.3(b) shows the relationship between sequence length and array utilization. If sequences are short, the controller finishes delivering each sequence of instructions to the array before receiving the next starting address. As a result, array utilization is low. If sequences are longer, the controller receives the next starting address before completing delivery of each sequence. Thus, array utilization is limited only by a few extra clock cycles which the controller needs to initiate each sequence. Typical sequences, over thirty instructions long, are executed efficiently.

4.3 Integrated Circuit Testing and Characterization

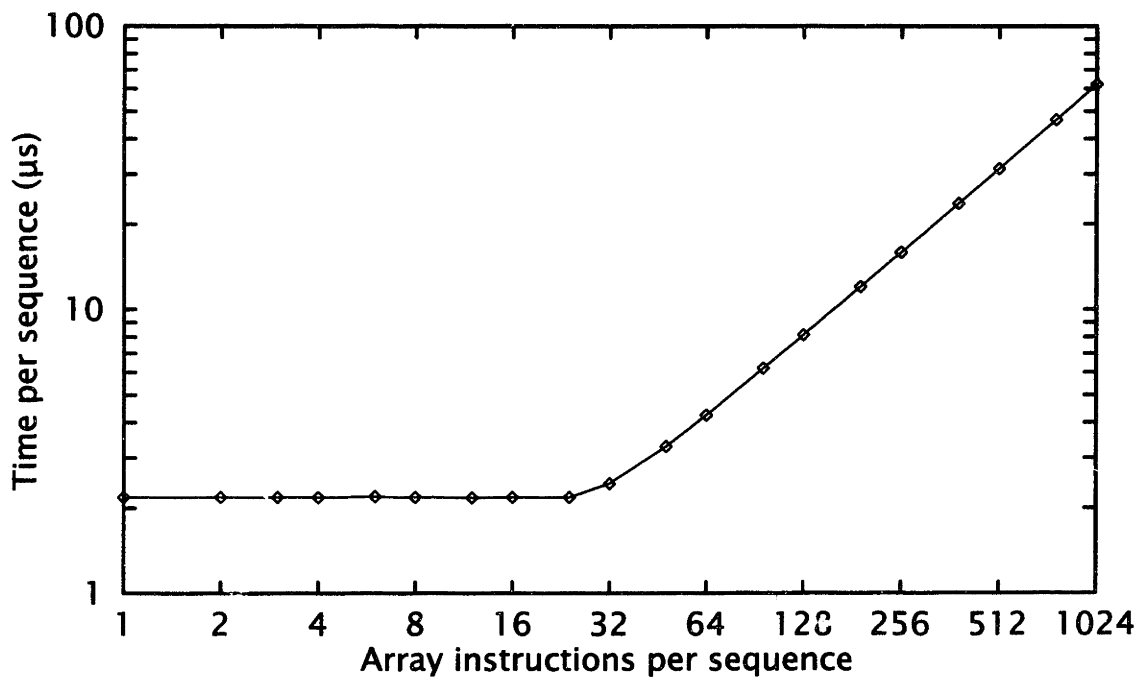
Integrated processing element devices were tested in the system. Through the VMEbus, the data path board provides test images from the host computer to the array and provides processed images from the array to the host computer. Initial testing work was devoted to identifying and correcting problems and to verifying chip functionality. The processing element array was operated with a relatively long, 100-ns clock cycle. Four basic test programs were employed. Later work focused on hold time and performance characterization.

4.3.1 Preliminary Functional Verification

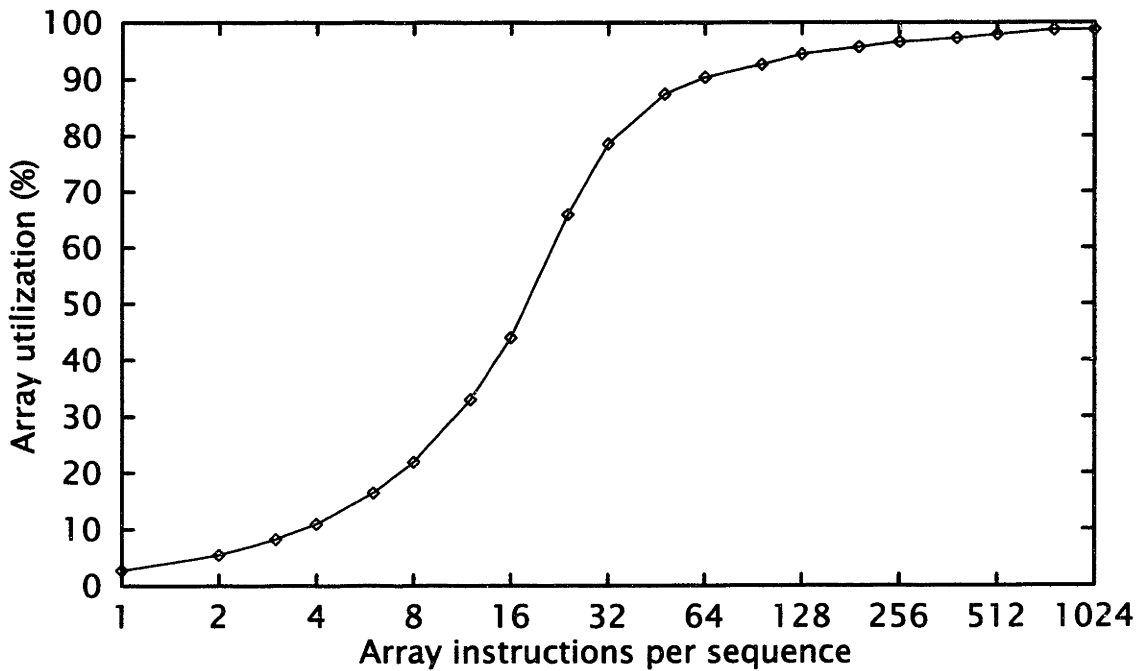
The first test program checks the serial-access memories. Data from the format converter board are clocked through the memories and back to the format converter board. The data are never transferred into processing element memory. The test program revealed several system software bugs, all of which were easily eliminated. The program also identified one logic error on the format converter board, one on the controller board, and one on the array board. Programmable logic device configurations were revised to fix the errors.

The second test program exercises the processing elements. Complete images from the host computer are transferred to and from the processing element array. The transfers use not only the serial-access memories, but also some of the processing element memory and some of the processing element logic.

The image I/O test program revealed a mistake in the instruction decode logic located along the perimeter of the chip. The signal from Ia input pad goes to the input of a



(a)



(b)

Figure 4.3: Control path performance. 60-ns clock cycle. (a) Execution time per sequence versus sequence length. (b) Array utilization versus sequence length.

latch. The output of the latch goes to two small blocks of logic. One of the blocks determines whether latch A should be loaded in each processing element. The other block determines whether any memory activity is necessary. When la is high, latch A should be loaded and a read should be performed. When la is low, latch A should not be loaded and the memory should be active only if a refresh or write operation is specified. The mistake inverted the result of the latch A logic block.

The software function that generates the bit patterns specifying array instructions was modified, circumventing the logic mistake. The la value is inverted, so that the latch A logic block produces desired results. The instruction code for a refresh operation is used whenever a read is performed. Thus, the memory is active whenever necessary. The only undesirable side effect of the software change is that the memory is active all the time. This increases average power dissipation but does not affect functionality.

The mistake was present in both the schematic representation and the layout. The chip design was easily fixed, and the software change will be reversed for devices produced in the future.

The third test program checks interchip communication. Complete images from the host computer are transferred to the processing element array. Chip operations are used to move image data north, south, east, or west. Resultant images are transferred from the array to the host computer. The image movement test program revealed a second logic error on the array board. A programmable logic device configuration was revised to fix the error.

The fourth test program checks processing element memory. To exercise all 128 bits of memory of every processing element, sets of sixteen eight-bit images are transferred to and from the processing element array. No functional problems were found.

4.3.2 Hold Time Characterization

The fourth test program also supports characterization of hold time, the maximum time that may elapse between refresh operations for each memory address without compromising data integrity.

The size and structure of the cell used to implement processing element memory is comparable to those of cells used in 1Mb DRAM chips. Most reported 1Mb DRAM chips exhibited an 8-ms hold time [31, page 159]. For commercial products, specified hold times must be sufficient for operation with an ambient temperature of 70°C.

Processing element memory was characterized at room temperature. As shown in Table 4.1, with a 15.7-ms refresh interval, twelve of fifteen chips exhibited fewer than seven failed cells. With a 7.86-ms refresh interval, none of the fifteen chips exhibited more than six failed cells. With a 1.97-ms refresh interval, eleven of the fifteen chips exhibited no failed cells.

While observed hold times for processing element memory are similar to the specified 1Mb DRAM hold times, the observed characteristics are worrisome. A study of DRAM leakage mechanisms included data for a planar cell similar to the one used to im-

Table 4.1**Hold Time Characterization Results**

In each row, each figure indicates the number of chips exhibiting the indicated number of failed DRAM cells. Fifteen chips were tested with a 60-ns clock cycle.

Refresh Interval	Failed Cells							
	0	1	2	3	4	5	6	≥ 7
15.4 μ s	15	0	0	0	0	0	0	0
123 μ s	14	1	0	0	0	0	0	0
246 μ s	12	3	0	0	0	0	0	0
492 μ s	11	3	1	0	0	0	0	0
983 μ s	11	3	1	0	0	0	0	0
1.97 ms	11	3	0	0	1	0	0	0
3.93 ms	4	7	2	1	1	0	0	0
7.86 ms	2	2	6	2	1	0	2	0
15.7 ms	0	1	3	3	3	0	2	3
31.5 ms	0	0	0	0	3	3	2	8

plement processing element memory [32]. Measured leakage current at 70°C was about two orders of magnitude greater than measured leakage current at room temperature. Thus, at the elevated temperatures used to test commercial products, the processing element memory may exhibit poor hold characteristics.

Leakage mechanisms may be classified using three categories: junction leakage, field leakage, and device leakage. Process specifications provide some insight into the importance of each category. Junction leakage includes minority carrier diffusion current and current produced by minority carrier generation associated with space-charge regions and surface states. Field leakage includes the current produced by bulk and surface generation in the transition region between gate oxide areas and field oxide areas. Field leakage may be quite high, especially when the polysilicon overlapping the gate oxide area and the transition region is held at a high potential [32]. Device leakage refers to subthreshold current through the transfer device.

Table 4.2 shows calculated leakage data. Maximum leakage currents were calculated directly from process specifications, cell layout dimensions, and the stored high potential. Maximum leakage rates were calculated by dividing the leakage currents by the storage node capacitance. The data indicate that junction leakage is not responsible for the observed cell failures. But both field leakage and device leakage may be significant for some of the longer experimental refresh intervals. Furthermore, device leakage alone may account for the observed room temperature hold time characteristics. If a stored 2.5-V potential falls at a rate of 200 V/s, then in less than 8 ms the stored potential will be below 1.25 V. Such a large drop in the stored potential, combined with unfavorable local variations in circuit characteristics, may certainly cause a cell failure.

Table 4.2
Calculated Cell Leakage Data

	Specification Units	Maximum Leakage Current	Maximum Leakage Rate
Junction Leakage			
area	fA/ $\mu\text{m}^2 \cdot \text{V}$	21 fA	0.5 V/s
perimeter	fA/ $\mu\text{m} \cdot \text{V}$	64 fA	1.4 V/s
Field Leakage	fA/ μm	1.1 pA	25 V/s
Device Leakage	pA/ μm	9.0 pA	200 V/s

Based on the hold time characterization results, the test and demonstration system is configured to execute a refresh sequence every 1.97 ms. Execution of the refresh sequence is triggered by a counter on the controller. The refresh sequence comprises 128 array instructions, one for each memory address. With a 60-ns clock cycle, the refresh sequence is executed in less than 1 μs . Thus, less than one-tenth of one percent of all clock cycles must be dedicated to refresh overhead.

4.3.3 Performance Characterization

Once initial work with the four basic test programs was completed, test programs based on typical image processing operations were developed. These programs transfer images from the host computer to the processing element array, perform an image processing operation on the images using the array, perform the same operation using a functional simulator on the host computer, transfer images from the array to the host computer and compare the results from the array with results from the simulator.

Of twenty-four chips, fifteen proved fully functional (one of the fifteen requires a 15.4- μs refresh interval). Nine are partially functional. As expected, the chips operate at speeds well over 10 MHz. A Tektronix HFS 9003 programmable stimulus system was substituted for the Hewlett-Packard pulse generators to determine the minimum required cycle time. The HFS 9003 provided direct software control of input clock signals. Automated test routines were employed to optimize clock timing. The minimum cycle time is 60 ns.¹ At this speed, typical power dissipation is 300 mW. Chip characteristics are summarized in Table 4.3.

¹Processing element memory and processing element logic cannot be independently tested. Therefore, it is not possible to experimentally determine the critical path that limits chip speed using the test and demonstration system.

Table 4.3
Chip Characteristics

Channel length	0.6 μm (drawn)
Polysilicon pitch	1.5 μm (without contacts)
First-level metal pitch	2.1 μm (contacted)
Second-level metal pitch	2.4 μm (contacted)
Third-level metal pitch	3.0 μm (contacted)
Processing elements	4096 (64 \times 64)
Memory	512Kb (twin cell DRAM)
Devices	2.7 M
Pads	144
Die size	9.7 \times 8.1 mm^2
Memory cell size	7.2 \times 7.2 μm^2
Power supplies	$V_{DD} = 2.5$ V (internal) $V_{HH} = 3.3$ V (interface) $V_{PP} = 3.3$ V (wordline)
Minimum cycle time	60 ns
Typical power dissipation	300 mW

4.4 System Performance

Three image processing operations are presented to demonstrate the performance of the system and to illustrate use of the programming framework. These operations reflect the computational demands of typical low-level image processing tasks. They are not presented as exemplary solutions to particular image processing problems. A video production demonstrates the system performing the tasks in real time [33].

4.4.1 Median Filtering

Median filtering may be applied to reduce image noise. The value of each output pixel is the median of all input values in a region centered at the output pixel. Median filtering eliminates spikes while maintaining sharp edges and preserving monotonic variations in pixel values.

Figures 4.4 (b) and (c) illustrate the effects of median filtering. For a 3×3 median filter, each output value is the median of the nine input values in a 3×3 pixel region. For a 5×5 median filter, each output value is the median of twenty-five input values.

Median filter operations are performed in a bit-serial manner starting with the most significant bits of all the input values and proceeding to the least significant bits. For a 5×5 median filtering operation, a twenty-five bit field, n , is used to hold, in each processing element, one bit of each of the input values in the corresponding 5×5 pixel

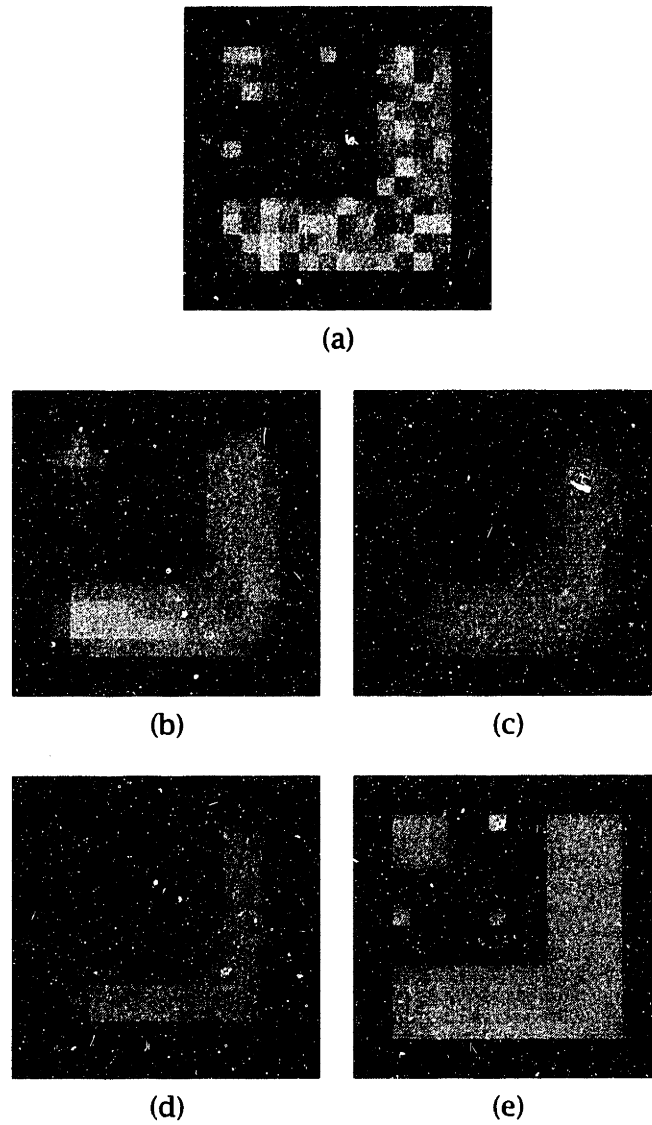


Figure 4.4: Median filtering, smoothing, and smoothing and segmentation. (a) Original image. (b) 3×3 median filtered image. (c) 5×5 median filtered image. (d) Smoothed image. (e) Smoothed and segmented image.

Table 4.4
Application Performance

	Execution Time	Array Utilization	Frames per Second
5 × 5 Median Filtering (8-bit image)	274 μ s	99 %	3653
Smoothing and Segmentation (8-bit image, 200 convolutions with threshold 20/256)	4.8 ms	97 %	207
Optical Flow (8-bit image, 16 × 16 support region 8-pixel maximum displacement)	30.1 ms	92 %	32

region. A fifty-bit field, t , is used to maintain, in each processing element, twenty-five two bit records. One bit of each record indicates whether a particular input value is known to be greater than the median value. The other bit indicates whether the input value is known to be less than the median value.

For each one-bit component of an input image, data are moved in a fixed pattern. For a 5 × 5 median filtering operation, twenty-four movement steps are performed. The movement pattern is designed to present data to each processing element for the corresponding 5 × 5 pixel region. Before the first step and after every step, image data are used to initialize one bit of field n .

After field n is initialized, values are altered according to the field t records. In each processing element, a 1 is stored in components of field n that correspond to input values known to be greater than the median value. A 0 is stored in components that correspond to input values known to be less than the median value. Once all the field n alterations are complete, a tally procedure is performed. The tally procedure computes, in each processing element, the sum of all twenty-five components of field n . The tally procedure, performed using a series of full-additions, requires a 13-bit field for temporary storage. If the tally result is greater than or equal to 13 the next output bit is 1. If the result less than or equal to 12 the next output bit is 0. The output result and field n values are used to update the field t records. If the output bit is 1, then all components of field n holding value 0 correspond to pixel values that are less than the median value. Conversely, if the output bit is 0, then all components of field n holding value 1 correspond to pixel values that are greater than the median value.

As shown in Table 4.4, operating with a 60-ns clock cycle, the processing element array performs a true 5 × 5 median filtering operation in less than 300 μ s. Near 100% array utilization is achieved by implementing the entire operation as a single sequence of array instructions.

4.4.2 Smoothing and Segmentation

Image acquisition noise may also be reduced by smoothing intensity variations. Repeatedly convolving an image with a 3×3 kernel,

$$\frac{1}{8} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

approximates a Gaussian smoothing operation. The number of convolutions applied determines the variance of the Gaussian filter.

As shown in Figure 4.4, simple smoothing suppresses higher spatial frequencies, reducing noise but also blurring edges. A smoothing and segmentation process reduces noise while preserving sharp edges. Before each convolution, the value of each pixel is compared with values of the pixel's four nearest neighbors. Where differences are greater than a segmentation threshold, the 3×3 kernel is locally modified to preserve the intensity variation. For example, if the value of a pixel differs from the value of the pixel's east neighbor by more than the threshold, but differs from the values of the pixel's other neighbors by less than the threshold, a modified 3×3 smoothing kernel,

$$\frac{1}{8} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 5 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

will be applied. Since the process of modifying and applying the 3×3 kernel must be performed for each pixel, the smoothing and segmentation task naturally maps onto pixel-parallel hardware.

Figure 4.5 shows a simplified smoothing and segmentation implementation, illustrating the use of the programming framework. File `hdpptype.h` gathers definitions and declarations for the processing element array architecture. The `#include` directive for `hdpptype.h` is the only architecture-dependent portion of the smoothing and segmentation code. The definition

```
hwSystem sys;
```

introduces an object (`sys`) representing the system hardware. Field definitions allocate processing element memory for pixel values (field `M`), north neighbor differences (field `ND`), and west neighbor differences (field `WD`). Field definitions specify field sizes and types. Field `M` holds 8-bit unsigned integers. Fields `ND` and `WD` hold 9-bit signed integers. Sequences produced by code generators are successively appended to a sequence object (`seq`) producing a composite sequence that performs one complete convolution step. The segmentation threshold is set at 20, and 200 convolutions are performed for each input image.

Table 4.4 presents performance data. Since each convolution step is performed using a single fixed sequence of array instructions, array utilization is very high. The processing element array performs 200 smoothing and segmentation convolutions in less than 5 ms.

```

#include <xapp.h>
#include <hdptype.h>

main()
{
    // Initialize system
    hwSystem sys;

    // Initialize PE memory management
    hfield hf(arch::array::FreeField);
    allocContext aC(hf);

    // Declare fields
    ufield M(aC.statically(), 8);           // pixel values
    sfield ND(aC.statically(), 9);         // north neighbor differences
    sfield WD(aC.statically(), 9);         // west neighbor differences

    // Build sequence
    sequence seq
        = nbrDifference(aC, M, North, ND); //  $ND \leftarrow M - M^{north}$ 
    seq += nbrDifference(aC, M, West, WD);  //  $WD \leftarrow M - M^{west}$ 

    seq += outside(aC, WD, -20, 20);       // if  $WD \notin \{-20, -19, \dots, 19, 20\}$ 
    seq += writeC(aC, WD, 0);              // then  $WD \leftarrow 0$ 

    seq += outside(aC, ND, -20, 20);       // if  $ND \notin \{-20, -19, \dots, 19, 20\}$ 
    seq += writeC(aC, ND, 0);              // then  $ND \leftarrow 0$ 

    seq += subtract(aC, (sfield) M, ND.at(3)); //  $M \leftarrow M - ND/8$ 
    seq += move(aC, ND, South);              //  $ND \leftarrow ND^{south}$ 
    seq += add(aC, (sfield) M, ND.at(3));     //  $M \leftarrow M + ND/8$ 

    seq += subtract(aC, (sfield) M, WD.at(3)); //  $M \leftarrow M - WD/8$ 
    seq += move(aC, WD, East);                //  $WD \leftarrow WD^{east}$ 
    seq += add(aC, (sfield) M, WD.at(3));     //  $M \leftarrow M + WD/8$ 

    // Load sequence
    sys.load(seq);

    for (;;) {
        // Acquire raw image

        // Process image, 200 iterations
        for (int i = 0; i < 200; i++) sys « seq;

        // Provide processed image
    }
}

```

Figure 4.5: Simplified smoothing and segmentation code.

4.4.3 Optical Flow

Given a time sequence of images, optical flow represents the apparent motion of image patterns. Optical flow may correspond to the motion of objects in front of a camera, or to the motion of a camera through an environment [34].

Little *et al.* developed an algorithm which determines optical flow using a series of image comparisons [35]. Given two successive images, $I_t(x, y)$ and $I_{t+\Delta t}(x, y)$, the algorithm produces a vector field $(d_x(x, y), d_y(x, y))$. At each pixel, the flow vector (d_x, d_y) is found by minimizing

$$\sum_{(x,y) \in R} C[I_t(x, y), I_{t+\Delta t}(x + d_x, y + d_y)]$$

where R is a local spatial region encompassing the pixel and C is a metric that indicates the amount of dissimilarity between image values.

The algorithm iterates over possible flow vectors. For each flow vector, two steps are performed. First, the dissimilarity metric C is evaluated at each pixel. Second, for each pixel the dissimilarity values are summed over the corresponding region R . At the end, at each pixel the flow vector which minimizes the sum of the dissimilarity values is chosen.

Table 4.4 presents performance data. Flow vectors are determined with the absolute value of the differences between pixel values serving as the dissimilarity metric. Each flow vector is determined based on sums of the absolute values for a 16×16 pixel region. Evaluating displacements as large as eight pixels in each direction (a total of $(8 + 1 + 8)^2 = 289$ flow vectors) the processing element array performs optical flow computation in about 30 ms.

4.5 Summary

A test and demonstration system employs four chips to form a 128×128 processing element array. This chapter described the system and presented results of test and characterization experiments performed with the system. Three items were investigated: the control path, the integrated processing element array, and the performance of the complete system. Control path characterization results show that the system efficiently executes typical sequences of array instructions. Array characterization results show that integrated processing element devices are fully functional, operate with a 60 ns cycle time, and typically dissipate only 300 mW. System performance results show that the system performs typical low-level image processing operations in less than 10 ms.

Chapter 5

Architectural Alternatives

This chapter compares the new integrated circuit architecture, described in the preceding chapters, with alternative architectures. The new architecture is identified as a pixel-parallel image processor. The comparisons serve two purposes. First, they help identify critical design features of the new architecture. Second, they highlight the accomplishments of this work.

Section 5.1 discusses a hypothetical processing element similar to the final design, but with simpler logic. Section 5.2 describes the methodology used to quantitatively compare the pixel-parallel image processor with other devices. Section 5.3 compares bit-serial and bit-parallel processing element designs. Sections 5.4, 5.5, and 5.6 compare implementations of low-level image processing tasks using the pixel-parallel image processor with implementations using a field programmable gate array, implementations using microprocessors, and implementations using a digital signal processor.

5.1 Simplified Processing Element

The most complex processing element circuit is the three-input Boolean function generator. As described in Chapter 2, the function generator requires eight four-transistor NAND stacks, a control signal for each stack, and a pair of complementary signals for each of the three inputs. Replacing the function generator with a simpler circuit might significantly reduce the total layout area per processing element.

A simplified processing element, shown in Figure 5.1, was originally considered. The simplified processing element uses a two-input Boolean function generator and has one less latch than the final design. Four control signals, f_{3-0} , select between the 16 two-input Boolean functions.

To evaluate the two processing element designs, several primitive arithmetic and data movement operations were considered. Efficient procedures were developed for each design. Then the characteristics of the two designs were compared. For arithmetic operations, large performance differences were encountered. The difference found for the sum operation is typical.

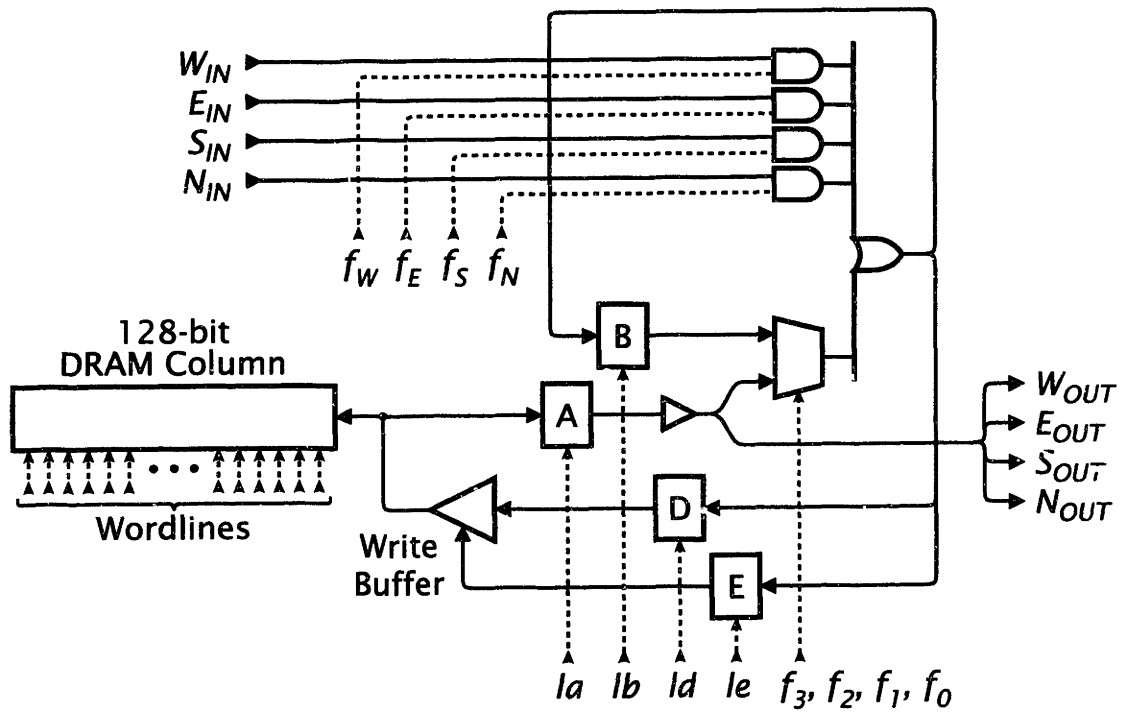


Figure 5.1: Hypothetical simplified processing element. Solid lines represent data signals. Dashed lines represent control signals.

Table 5.1
Sum Procedure, $S \leftarrow A + B$

	Logic Activity	Memory Activity
1	$E \leftarrow 1$	$A \leftarrow Mem[B_0]$
2	$B \leftarrow A$	$A \leftarrow Mem[A_0]$
3	$D \leftarrow A \oplus B$	$Mem[S_0] \leftarrow D$
4	$C \leftarrow A \wedge B$	$A \leftarrow Mem[B_1]$
5	$B \leftarrow A$	$A \leftarrow Mem[A_1]$
6	$D \leftarrow A \oplus B \oplus C$	$Mem[S_1] \leftarrow D$
7	$C \leftarrow (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$	$A \leftarrow Mem[B_2]$
8	$B \leftarrow A$	$A \leftarrow Mem[A_2]$
9	$D \leftarrow A \oplus B \oplus C$	$Mem[S_2] \leftarrow D$
10	$C \leftarrow (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$	$A \leftarrow Mem[B_3]$
11	$B \leftarrow A$	$A \leftarrow Mem[A_3]$
12	$D \leftarrow A \oplus B \oplus C$	$Mem[S_3] \leftarrow D$

Table 5.2
Sum Procedure, $S \leftarrow A + B$
for Hypothetical Simplified
Processing Element

	Logic Activity	Memory Activity
1	$E \leftarrow 1$	$A \leftarrow Mem[B_0]$
2	$B \leftarrow A$	$A \leftarrow Mem[A_0]$
3	$D \leftarrow A \oplus B$	$Mem[S_0] \leftarrow D$
4	$B \leftarrow A \wedge B$	$A \leftarrow Mem[B_1]$
5	$D \leftarrow A \vee B$	$Mem[U] \leftarrow D$
6	$D \leftarrow A \wedge B$	$Mem[V] \leftarrow D$
7	$B \leftarrow A \oplus B$	$A \leftarrow Mem[A_1]$
8	$D \leftarrow A \oplus B$	$Mem[S_1] \leftarrow D$
9	$B \leftarrow A$	$A \leftarrow Mem[U]$
10	$B \leftarrow A \wedge B$	$A \leftarrow Mem[V]$
11	$B \leftarrow A \vee B$	$A \leftarrow Mem[B_2]$
12	$D \leftarrow A \vee B$	$Mem[U] \leftarrow D$
13	$D \leftarrow A \wedge B$	$Mem[V] \leftarrow D$
14	$B \leftarrow A \oplus B$	$A \leftarrow Mem[A_2]$
15	$D \leftarrow A \oplus B$	$Mem[S_2] \leftarrow D$
16	$B \leftarrow A$	$A \leftarrow Mem[U]$
17	$B \leftarrow A \wedge B$	$A \leftarrow Mem[V]$
18	$B \leftarrow A \vee B$	$A \leftarrow Mem[B_3]$
19	$B \leftarrow A \oplus B$	$A \leftarrow Mem[A_3]$
20	$D \leftarrow A \oplus B$	$Mem[S_3] \leftarrow D$

Table 5.1 presents a sum procedure, $S \leftarrow A + B$, for the final design. Table 5.2 presents a corresponding procedure for the simplified processing element. The procedures read two four-bit values from memory and store a four-bit result in memory.

Having a three-input function generator, the final design requires only three logic operations per bit position: one to compute the carry value using values from the previous bit position, one to copy an addend value from latch A to latch B, and one to compute the sum value. The simplified design requires a much longer sum procedure. Each three-input logic function must be reduced to a series of two-input logic functions. Two one-bit fields, U and V , are needed to hold intermediate results.

To perform an n -bit sum, the simplified design requires $7n - 8$ cycles and memory operations. The final design requires only $3n$ cycles and memory operations. Because of such large differences in arithmetic performance, the simplified design was rejected.

5.2 Comparison Methodology

The performance of a system depends on the algorithms chosen to implement the desired tasks. An algorithm that works well on one system may be inappropriate for another system. Thus, when comparing systems, it is very important to choose the algorithms for each system independently. Furthermore, a designer of one system might not recognize efficient algorithms for competing systems. Therefore, experimental performance data for low-level image processing tasks executed on alternative systems were gathered from published documents. The data are compared with measured performance data for the same tasks executed on the pixel-parallel image processor.

Four figures of merit are used. Two of the four are silicon area per pixel and time per frame. Area per pixel values were derived from chip dimensions, the number of pixels handled per processor, and the number of processors per chip:

$$\text{Area per Pixel} = \frac{\text{Chip Area}}{\text{Pixels per Processor} \times \text{Processors per Chip}}$$

Area per pixel may also be understood as chip area divided by the number of pixels handled by a single chip. Time per frame values were taken directly from reported or measured data or derived from reported cycle counts and clock frequencies.

The number of pixels per processor may often be adjusted to achieve a desired area per pixel or a desired time per frame. Handling more pixels with each processor decreases area per pixel, but increases time per frame. Handling fewer pixels with each processor decreases time per frame, but increases area per pixel. A third figure of merit, the product of area per pixel and time per frame, provides a performance measure that is essentially independent of the number of pixels per processor. The inverse of the area-time product may be interpreted as processing speed per unit silicon area.

The fourth figure of merit is energy per pixel. Values were derived from power dissipation, time per frame, the number of pixels handled per processor, and the number of processors per chip:

$$\text{Energy per Pixel} = \frac{\text{Power Dissipation} \times \text{Time per Frame}}{\text{Pixels per Processor} \times \text{Processors per Chip}}$$

The product of power dissipation and time per frame is the energy used per chip. The product of pixels per processor and processors per chip is the number of pixels handled per chip. Thus, energy per pixel is simply energy per chip divided by the number of pixels per chip.

5.3 Bit-Parallel Processing Element

One of the most prominent features of the integrated processing element array is the use of one-bit-wide data paths and functional units. For primitive arithmetic and data movement operations, a processing element generally requires a few instructions for

each bit position. In contrast, a processing element design employing wider data paths might require only a few instructions to perform entire operations. Thus, the compact size of a single bit-serial processing element must be weighed against the superior processing power of a single bit-parallel processing element.

A simple analysis suggests that the performance of a comparable bit-parallel design might be similar to the performance of the bit-serial design of the pixel-parallel image processor. Consider forming an array of bit-parallel processing elements by substituting a single eight-bit-wide logic unit for groups of eight adjacent one-bit-wide logic units. A single eight-bit-wide arithmetic logic unit (ALU) would replace eight individual one-bit-wide function generators. In the bit-serial design, each processing element serves a single pixel. In the bit-parallel design, each processing element would serve a group of eight pixels. Consider addition, a typical primitive operation. To perform an eight-bit sum, the bit-serial design requires twenty-four memory operations: two read and one write operation per bit times eight bits. The bit-parallel design would also require twenty-four memory operations: two read and one write operation per pixel times eight pixels. If both designs have sufficiently powerful logic units, the performance of both will be limited by their equivalent memory demands.

With respect to integrated circuit implementation, the most significant inherent difference between bit-serial and bit-parallel architectures is the difference between the structures of one-bit-wide function generator circuits and bit-parallel ALU circuits. The DRAM memory array used for the bit-serial processing elements may also be used for bit-parallel processing elements. The one-bit-wide latches in the bit-serial processing elements may be logically grouped to form bit-parallel latches.

A bit-parallel ALU circuit appropriate for a processing element array must perform integer addition, subtraction, and comparison functions. Such a circuit may be created by connecting together one-bit ALUs, with each one-bit ALU circuit incorporating a full adder [36, pages 182-198]. Thus, the size of a simple full adder provides a basis for judging relative sizes of a bit-parallel ALU and the three-input Boolean function generator used in the bit-serial processing element.

A compact full adder circuit using pass transistors requires only twenty-eight transistors [37]. However, the connections between transistors in the full adder circuit are more complex than the connections in the Boolean function generator circuit. In the full adder, many connections must be formed using metal lines. In the Boolean function generator, only one internal node, the evaluation node, requires metal lines. All other internal nodes are *n*-diffusion regions shared by two adjacent transistors. Implemented in a CMOS process with 0.5 μm drawn features, the full adder occupies over 4200 μm^2 . The three-input Boolean function generator, implemented with 0.6 μm drawn features, requires less than 900 μm^2 . The difference in area between these two circuits provides a very strong indication that a bit-parallel ALU would require much more area than a corresponding number of three-input Boolean function generators.

An Integrated Memory Array Processor (IMAP) developed at NEC integrates 64 eight-bit processing elements [38]. The principle components of the IMAP processing element

are a $4\text{Kb} \times 8$ SRAM, an eight-bit ALU, an eight-bit shifter, and eight-bit registers. The $15.1 \times 15.6 \text{ mm}^2$ chip was fabricated in a $0.55\text{-}\mu\text{m}$ BiCMOS process with $5.8 \times 3.2 \mu\text{m}^2$ SRAM cells.

The eight-bit IMAP ALU occupies almost $100\,000 \mu\text{m}^2$. In the pixel-parallel image processor, eight three-input function generators occupy less than $7200 \mu\text{m}^2$. The chips were designed with similar minimum feature sizes. Thus, the IMAP design provides more evidence that the area required per bit to implement ALU circuits exceeds the area required to implement a one-bit-wide function generator.

The bit-parallel IMAP architecture incorporates a local addressing facility. Thus, IMAP processing elements can individually select memory locations. The local addressing capability makes the IMAP architecture suitable for a larger class of image processing tasks than the pixel-parallel image processor.

While IMAP processing elements are connected using a linear network, image processing operations may be performed efficiently by assigning a complete image column to each processing element. An eight-chip IMAP system processes 512×512 images. Table 5.3 presents performance data for a 3×3 median filtering operation. With each processing element handling 512 pixels, the IMAP chip uses less than half as much area per pixel as the pixel-parallel image processor. But the pixel-parallel image processor performs the median filtering operation over twenty times faster.

The area-time product data show that for the median filtering operation, the pixel-parallel image processor is almost ten times faster per unit silicon area than the IMAP chip. The energy per pixel values show that the pixel-parallel image processor requires less than one-tenth of the energy per pixel required by the IMAP chip. The difference between the area-time products reflect the size advantage of the one-bit-wide function generators over the bit-parallel IMAP ALU circuits. Part of the difference in the energy per pixel values may be attributed to a difference in internal operating voltages. The pixel-parallel image processor chips use a 2.5-V supply. The IMAP chips use a 3.3-V supply. An IMAP chip using a 2.5-V supply might require $43\% = 1 - (2.5^2/3.3^2)$ less energy per pixel than the reported IMAP chip.

5.4 Field-Programmable Gate Array

A field programmable gate array (FPGA) comprises a matrix of logic blocks with interconnections controlled by programmable switches. Using an FPGA, complex application-specific logic functions may be implemented on a single device. Unlike hardwired gate arrays, FPGAs can be reprogrammed to fix design flaws, improve performance, or perform an entirely different function. FPGAs offer a very useful compromise between the performance of application-specific custom integrated circuits and the flexibility of microprocessors and microcontrollers.

The Xilinx XC4000 series devices use switches formed by pass transistors with gates connected to SRAM cells [39]. Each logic block includes two four-input function generators and two flip-flops. A function generator may be configured as a 16×1 memory.

Table 5.3
Comparison of Image Processing Devices, 3×3 Median Filtering

	Area per pixel	×	Time per frame	Energy per pixel
Pixel-Parallel Image Processor 4096 One-bit-wide PEs One pixel per PE	$19\,222 \mu\text{m}^2$	×	$105 \mu\text{s}$ $2 \mu\text{m}^2 \cdot \text{s}$	7 nJ
IMAP [38] 64 Eight-bit-wide PEs One 512-pixel column per PE	$7\,189 \mu\text{m}^2$	×	$2\,421 \mu\text{s}$ $17 \mu\text{m}^2 \cdot \text{s}$	133 nJ
Xilinx XC4000 Series FPGA [39] 85 logic blocks, 66 MHz 4096 pixels, 1 cycle per pixel	—		$62 \mu\text{s}$	9 nJ
TI TMS320C80 DSP [40, 41] Four 32-bit integer processors, 50 MHz 512×512 image	$1\,305 \mu\text{m}^2$	×	12ms $16 \mu\text{m}^2 \cdot \text{s}$	211 nJ

IMAP power requirements depend on properties of executed tasks. The published IMAP description [38] does not report power data for the median filter operation. The IMAP energy per pixel figure was derived using reported power data for 3×3 convolution.

The Xilinx XC4000 Series figures are based on a pipelined median filter design using eight-bit comparators [42]. Each comparator takes two eight-bit inputs and produces two eight-bit outputs. One output is the larger input, the other output is the smaller input. The figures are also based on crude speed and power estimates derived from Xilinx product information [43].

The TMS320C80 time per frame figure is from a description of the MediaStation 5000 [44], a multi-media system based on the TMS320C80.

The TMS320C80 energy per pixel figure is derived from typical power dissipation reported in Texas Instruments product information [45].

Table 5.4
Comparison of Image Processing Devices, Sobel filtering

	Area per pixel	Time per frame	Energy per pixel
Pixel-Parallel Image Processor 4096 One-bit-wide PEs One pixel per PE	19 222 μm^2	72 μs	5 nJ
Xilinx XC4000 Series FPGA [39] 36 logic blocks, 66 MHz 4096 pixels, 9 cycles per pixel	—	559 μs	32 nJ

The Xilinx XC4000 Series figures are based on a distributed arithmetic filter design [48]. The figures are also based on crude speed and power estimates derived from Xilinx product information [43].

Currently, the largest available XC4000 FPGA provides just over two-thousand logic blocks [43, page 4-6].

Table 5.3 provides performance data for a 3×3 median filtering operation. Distributed arithmetic structures [46] may be applied to efficiently implement signal processing operations using FPGAs [47]. Tables 5.4 and 5.5 provide performance data for a 3×3 Sobel filtering operation and a 5×5 Laplacian convolution. Both tasks are implemented using distributed arithmetic. Not surprisingly, the data show that XC4000 FPGA circuits implemented using a modest number of logic blocks can perform the operations quite rapidly. While the data is based on processing 4096 pixels per FPGA circuit, the circuits may be replicated to accommodate images with more than 4096 pixels. (Chip area data for Xilinx FPGAs was not available.)

The processing power of small XC4000 FPGA circuits is quite impressive, but the circuits require more energy than the pixel-parallel image processor. For two of the three operations, the FPGA circuits require over twice as much energy per pixel. The high energy demands are a result of the large capacitances of internal FPGA nodes. To provide necessary interconnect flexibility, each flip-flop output is connected to several switches. The switches, open or closed, contribute to the capacitance of the output node. For the XC4000 series, using a 5-V power supply, a flip-flop driving only a neighboring flip-flop requires 0.1 nJ per transition [43, page 13-12]. This corresponds to an 8 pF output load. With such high internal node capacitances, even the compact application-specific FPGA circuits have high energy demands.

It is not clear whether FPGAs may be effectively applied to execute some of the more complex image processing tasks that are performed by the pixel-parallel image processor. All of the the filtering tasks for which FPGA circuits have been designed are amenable to implementation using fairly small, pipelined, application-specific data paths. Implementing optical flow computation and other more complex tasks would

Table 5.5
Comparison of Image Processing Devices, Laplacian Convolution

	Area per pixel	Time per frame	Energy per pixel
Pixel-Parallel Image Processor			
4096 One-bit-wide PEs One pixel per PE	19 222 μm^2	197 μs	16 nJ
Xilinx XC4000 Series FPGA [39]			
36 logic blocks, 66 MHz 4096 pixels, 9 cycles per pixel	—	683 μs	42 nJ

The Xilinx XC4000 Series figures are based on a distributed arithmetic filter design [49]. The figures are also based on crude speed and power estimates derived from Xilinx product information [43].

certainly require more complex FPGA circuits. With longer interconnections between logic blocks, more complex FPGA circuits generally require more power per internal node and operate at slower speeds than simpler circuits. Furthermore, the tools used to create FPGA circuits and the number of available interconnections between logic blocks may not be sufficient.

5.5 Microprocessor

Since modern pipelined microprocessor designs operate with very short cycle times, it is reasonable to consider applying them to real-time image processing tasks. A recent study measured the performance of a workstation using a 200 MHz DECchip 21064 microprocessor on low-level and high-level image processing tasks [50].

The DECchip 21064 microprocessor [51, 52] has 64-bit integer and floating point execution units. The $16.8 \times 13.9 \text{ mm}^2$ chip is fabricated in a $0.75 \mu\text{m}$ 3.3 V CMOS process. The power dissipation of the chip is 30 W, one hundred times greater than the typical power dissipation of the pixel-parallel image processor.

The study considered 5×5 convolution, a typical task for the pixel-parallel image processor. The measured performance figure is included in Table 5.6, along with calculated area per pixel, area-time product, and energy per pixel. While the speed of a DECchip 21064 microprocessor is impressive, it is not sufficient to support real time processing of large images with a conventional single-processor workstation. Furthermore, the area-time product data show that the pixel-parallel image processor is over ten times faster per unit area than the DECchip 21064 microprocessor. The energy per pixel data shows that the microprocessor requires almost five-hundred times more energy per pixel than the pixel-parallel image processor.

Table 5.6
Comparison of Image Processing Devices, 5×5 Convolution

	Area per pixel	×	Time per frame	Energy per pixel
Pixel-Parallel Image Processor 4096 One-bit-wide PEs One pixel per PE	19 222 μm^2	×	730 μs	55 nJ
			14 $\mu\text{m}^2 \cdot \text{s}$	
DECchip 21064 μP [51, 52] 64-bit integer & floating-point units 512 \times 512 image	891 μm^2	×	220 ms	25 177 nJ
			196 $\mu\text{m}^2 \cdot \text{s}$	
TI TMS320C80 DSP [40, 41] Four 32-bit integer processors, 50 MHz 512 \times 512 image	1 305 μm^2	×	40 ms	705 nJ
			52 $\mu\text{m}^2 \cdot \text{s}$	

The TMS320C80 time per frame figure is from a description of the MediaStation 5000 [44], a multi-media system based on the TMS320C80.

The TMS320C80 energy per pixel figure is derived from typical power dissipation reported in Texas Instruments product information [45].

The DECchip 21064 time per frame figure is from a study of the performance of microprocessor-based systems on image processing applications [50].

The huge difference in energy requirements is not at all surprising. In the DEC-chip 21064 design, like other modern microprocessor designs, deeply pipelined data paths, high speed cache memories, and fast multiple-port register files are employed to provide high performance while supporting the traditional single-sequence programming model. These features increase the energy required per computation. Also, when performing low-level image processing operations, microprocessors use energy decoding the same sequence of instructions for each pixel. Furthermore, the 64-bit data paths employed in the DECchip 21064 and other modern microprocessors are much wider than necessary for processing low-precision pixel data. Finally, microprocessors include support for a much more complex memory hierarchy than necessary for low-level image processing.

Some recent microprocessors use special graphics units to provide better performance on multimedia applications. One such microprocessor is the Sun UltraSPARC I [53, 54]. The chip incorporates a graphics unit including four 16-bit integer adders and four 8-bit \times 16-bit integer multipliers. A special set of instructions, the Visual Instruction Set (VIS) [55, 56], includes instructions that perform operations on four data streams in parallel. The 17.7×17.8 mm chip is fabricated in a $0.5 \mu\text{m}$ CMOS process with four metal layers. Operating with a 167 MHz clock, the chip dissipates 28 W.

Using the graphics unit, the UltraSPARC I can perform the low-precision integer operations typical of low-level image processing tasks more efficiently than it can using its general-purpose 64-bit integer units. Table 5.7 includes performance data for a separable 3×3 convolution operation executed on the pixel-parallel image processor, the UltraSPARC I 64-bit integer unit, and the UltraSPARC I graphics unit.

While the performance of the UltraSPARC I graphics unit is much better than the performance of the integer unit, it is not close to the performance of the pixel-parallel image processor. The separable 3×3 convolution is a very simple operation. The pixel-parallel image processor performs the operation in less than $200 \mu\text{s}$, yet the UltraSPARC I requires over 3 ms. Based on this result, the UltraSPARC I may not be capable of performing more complex image processing operations in real time. The area-time product for the pixel-parallel image processor is less than one-fifth of the UltraSPARC I value, even when the graphics unit is employed. And the pixel-parallel image processor requires less than one one-hundredth of the energy per pixel required by the UltraSPARC I.

5.6 Digital Signal Processor

The Texas Instruments TMS320C80 digital signal processor is a software-programmable device optimized for three groups of multimedia applications: data compression, graphics, and image processing [40, 41]. The TMS320C80 integrates a general-purpose master processor with integer and floating point units and four parallel processors optimized for integer operations. Each of the four parallel processors includes a 32-bit ALU and a 16-bit \times 16-bit multiplier. The ALU may be used to perform one 32-bit operation, two

16-bit operations, or four 8-bit operations. Thus, with the four parallel processors, the TMS320C80 can perform sixteen 8-bit operations simultaneously. The $18.1 \times 18.9 \text{ mm}^2$ chip is fabricated in a $0.5 \text{ }\mu\text{m}$ CMOS process with three metal layers. Operating at 50 MHz with a 3.3 V supply, typical power dissipation is 4.6 W.

Tables 5.3, 5.8, and 5.6 provide performance data for three low-level image processing tasks: 3×3 median filtering, 3×3 convolution, and 5×5 convolution, respectively. The TMS320C80 completes the two simpler tasks in less than 30 ms, but 5×5 convolution requires 40 ms. Adjusted to account for differences in area per pixel, the speed of the TMS320C80 is comparable to the speed of the pixel-parallel image processor. In addition, the TMS320C80, executing many operations in parallel, performs image processing tasks much more efficiently than a microprocessor. However, it requires over ten times more energy per pixel than the pixel-parallel image processor.

5.7 Summary

The architecture of the pixel-parallel image processor is much better suited to low-level image processing tasks than alternative architectures. Neither a less complex processing element, with a two-input function generator, nor a more complex processing element, with multi-bit data paths, would provide higher speed with the same area per pixel or the same speed with less area per pixel. Data for many low-level image processing tasks show that the pixel-parallel image processor requires as little as a sixth of the energy per pixel required by FPGA circuits, less than a tenth of the energy required by a digital signal processor and less than a hundredth of the energy required by two microprocessors. Furthermore, viewed together, area per pixel and time per frame data show that only FPGA circuits derive as much processing power per area as the pixel-parallel image processor.

Figure 5.2 qualitatively illustrates the operational characteristics and the relative flexibility of several architectures considered in this chapter. The area-time and energy characteristics of the pixel-parallel image processor are clearly superior to the those of the other architectures. General microprocessors are used in a very wide variety of applications, but for low-level image processing tasks they are very inefficient. Digital signal processors are better suited to low-level image processing tasks than microprocessors, but are not nearly as efficient as the pixel-parallel image processor. For very simple image processing tasks, field programmable gate arrays require somewhat more energy than the pixel-parallel image processor. Field programmable gate arrays are used in variety of applications, but it is not clear whether they can be applied to perform more complex image processing tasks.

Table 5.7
Comparison of Image Processing Devices, 3×3 Separable Convolution

	Area per pixel	×	Time per frame	Energy per pixel
Pixel-Parallel Image Processor 4096 one-bit-wide PEs One pixel per PE	$19\,222 \mu\text{m}^2$	×	$158 \mu\text{s}$	12 nJ
	$3 \mu\text{m}^2 \cdot \text{s}$			
UltraSPARC I μP [53, 54, 56] 64-bit integer unit, 167 MHz 256×256 image	$4807 \mu\text{m}^2$	×	$17\,126 \mu\text{s}$	7 317 nJ
	$82 \mu\text{m}^2 \cdot \text{s}$			
UltraSPARC I μP [53, 54, 56] 16-bit graphics unit, 167 MHz 256×256 image	$4807 \mu\text{m}^2$	×	$3\,253 \mu\text{s}$	1 390 nJ
	$16 \mu\text{m}^2 \cdot \text{s}$			

The UltraSPARC I energy per pixel figures are derived from reported cycle counts [56] and clock frequency [53].

Table 5.8
Comparison of Image Processing Devices, 3×3 Convolution

	Area per pixel	×	Time per frame	Energy per pixel
Pixel-Parallel Image Processor 4096 One-bit-wide PEs One pixel per PE	$19\,222 \mu\text{m}^2$	×	$253 \mu\text{s}$	19 nJ
	$5 \mu\text{m}^2 \cdot \text{s}$			
TI TMS320C80 DSP [40, 41] Four 32-bit integer processors, 50 MHz 512×512 image	$1\,305 \mu\text{m}^2$	×	19 ms	335 nJ
	$25 \mu\text{m}^2 \cdot \text{s}$			

The TMS320C80 time per frame figure is from a description of the MediaStation 5000 [44], a multimedia system based on the TMS320C80.

The TMS320C80 energy per pixel figure is derived from typical power dissipation reported in Texas Instruments product information [45].

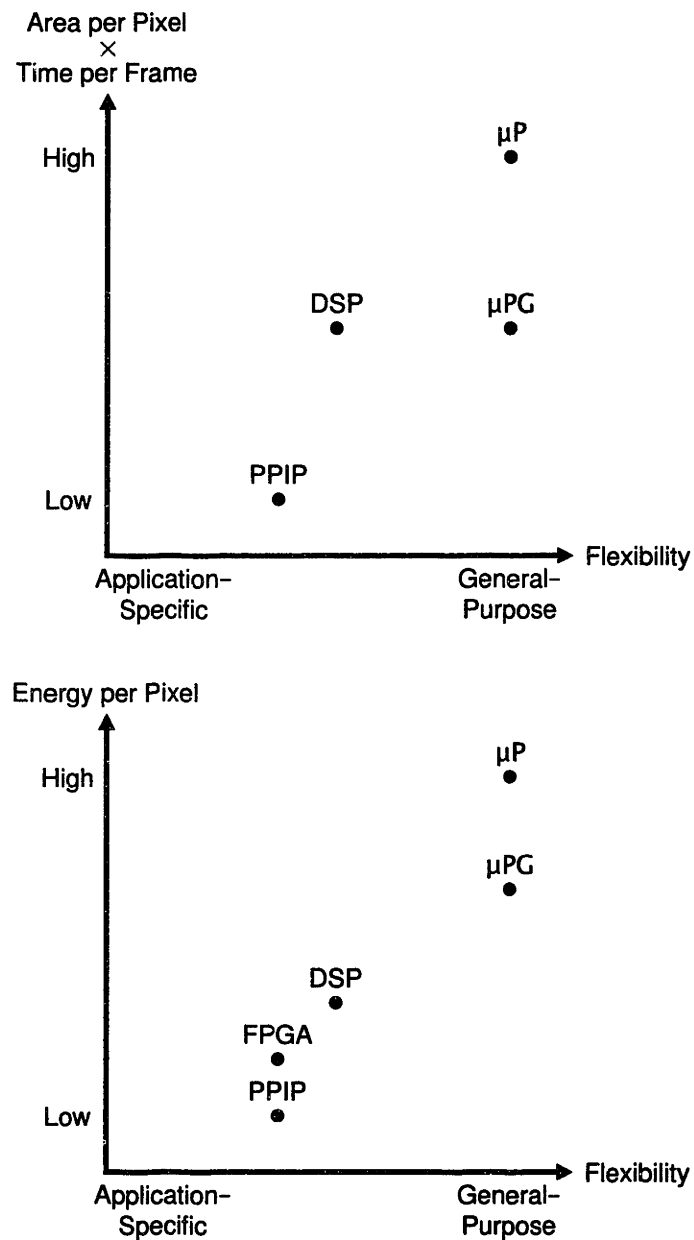


Figure 5.2: Characteristics of architectures applied to low-level image processing tasks: pixel-parallel image processor (PPIP), field-programmable gate array (FPGA), digital signal processor (DSP), microprocessor (μP), and microprocessor with graphics unit (μPG).

Chapter 6

Conclusion

This thesis has described the architecture and implementation of an integrated processing element array, and the application of the array to form a desktop pixel-parallel image processing system. This chapter provides a summary of the thesis and presents some ideas for future work.

6.1 Summary

Typical low-level image processing tasks are performed by applying a uniform set of operations for each pixel in each input image. Thus, they may be efficiently handled by an array of processing elements, one per pixel, sharing instructions issued by a single controller. The new integrated circuit architecture, employing one-bit-wide logic circuits pitch-matched to DRAM columns, provides the density necessary to produce large processing element arrays at low cost.

A processing element combines a 128-bit DRAM column and logic circuits. A prototype integrated circuit provides 4096 processing elements. With respect to both area and power dissipation, the processing element logic circuits are very efficient. They require less area and less power than the DRAM columns.

The processing elements are interconnected to create a 64×64 array. A multiplex circuit allows the interconnection network to be extended across chip boundaries using only one pad for every four perimeter processing elements. Thus, multiple chips may be used to form processing element arrays matching the size of large images. The interconnection network incorporates serial-access memories. The serial-access memories provide a means of transferring image data to and from the processing element array.

The area of the prototype device is less than 80 mm^2 . Operating with a 60 ns clock cycle, the device typically dissipates only 300 mW. Full functionality was proven using methodical test programs and test programs based on typical image processing operations.

A demonstration system employs four chips, forming a 128×128 array. Format converters, implemented using multidimensional access memories, transfer data be-

tween the processing element array and conventional bit-parallel components in real time. The control path, with simple hardware, sustains near-constant array activity. The complete system is fully functional and performs typical low level image processing tasks at speeds exceeding thirty frames per second.

Comparisons with alternative computing architectures highlight the performance and efficiency of the integrated processing element array. Field programmable gate arrays offer comparable speed, but require six times more energy per pixel. For large images, microprocessors, even with a special graphics unit, require much more time per frame than a suitably-sized processing element array. Furthermore, microprocessors require over a hundred times more energy per pixel. A digital signal processor provides processing power comparable to the processing element array, but requires ten times more energy per pixel.

6.2 Future Work

Work is underway to improve the demonstration system. A second fabrication run is in progress. Chips from the first run were packaged in a pin grid array. Chips from the second run will be packaged in a plastic quad flatpack. The flatpack is smaller than the pin grid array, and may be mounted on the surface of printed circuit boards without through-holes. A new array board with sixteen new chips will provide a 256×256 processing element array. One or two simple auxiliary boards, fitting inside the VMEbus chassis, will provide power supplies and clock signals to the new array board. Thus, the system will no longer require laboratory instruments.

To make the demonstration hardware more accessible, full support for personal computers using Intel microprocessors will be provided. Already, all host-independent portions of the system software have been ported to the Sun Solaris x86 and Linux operating environments. An adapter links the Peripheral Component Interconnect (PCI) backplane bus to the VMEbus chassis. To complete the work, a Solaris x86 or Linux device driver must be developed for the adapter. Alternatively, the system software may be reworked to support a Microsoft operating environment.

The decisions to use a VMEbus interface for the controller and format converter boards were pragmatic responses to contemporary considerations. Should a new I/O bus standard gain widespread acceptance, a more portable and less expensive system could be created using that standard. Two prospects are the Universal Serial Bus (USB) and IEEE 1394 (FireWire) standards. A system could also be created using the Small Computer System Interface (SCSI) standard.

The current system supports grayscale images. The system could be enhanced to support color images. The enhancement would require a new format converter implementation and extension of the programming framework. The programming framework could also be extended to handle two or more pixels with each processing element, trading performance to reduce the required number of chips.

The most significant remaining integrated circuit issue is high-volume production of processing element arrays. This issue would be best addressed in an industrial setting, where appropriate experimental work can be performed. Development of techniques for incorporating redundant processing elements would likely be beneficial.

Now that the integrated circuit and system designs are proven, the most challenging problem is the application of pixel-parallel image processing to real-world problems. It is difficult to convince designers to consider using hardware that is not widely available. It is also difficult to convince manufacturers to produce hardware for which there is no established market.

Fortunately, the dramatic performance advantages of pixel-parallel image processing and logic pitch-matched to dynamic memory are commanding attention. For example, researchers at the M.I.T. Artificial Intelligence Laboratory are working to apply the demonstration hardware to the design of vision systems. One project is a system for searching and indexing image and video databases. The integrated processing element array may be used to perform computationally intensive template matching operations. Another project is an adaptive automobile cruise control system. Images from three cameras mounted side-by-side are compared to determine the distance to nearby automobiles. The distance information is used to vary speed and avoid collision. The image comparisons may be performed in real time using an integrated processing element array.

References

- [1] Duncan G. Elliott, W. Martin Snelgrove, and Michael Stumm, "Computational RAM: A memory-SIMD hybrid and its application to DSP," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 30.6.1–30.6.4, May 1992.
- [2] Robert Heaton, Donald Blevins, and Edward Davis, "A bit-serial VLSI array processing chip for image processing," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, pp. 364–368, April 1990.
- [3] Maya Gokhale, Bill Holmes, and Ken Iobst, "Processing in memory: The Terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, April 1995.
- [4] Christine Tomovich, "MOSIS—a gateway to silicon," *IEEE Circuits and Devices Magazine*, vol. 4, no. 2, pp. 22–23, March 1988.
- [5] R. C. Foss and R. Harland, "Standards for dynamic MOS RAMs," *Electronic Design*, vol. 25, no. 17, pp. 66–70, August 1977.
- [6] Roy R. DeSimone, Nicholas M. Donofrio, Barry L. Flur, Robert H. Kruggel, and Howard H. Leung, "FET RAMs," in *IEEE International Solid-State Circuits Conference: Digest of Technical Papers*, pp. 154–155, 291, February 1979.
- [7] V. Leo Rideout, "One-device cells for dynamic random-access memories: A tutorial," *IEEE Transactions on Electron Devices*, vol. ED-26, no. 6, pp. 839–852, June 1979.
- [8] Howard L. Kalter, Charles H. Stapper, John E. Barth, Jr., John DiLorenzo, Charles E. Drake, John A. Fifield, Gordon A. Kelley, Jr., Scott C. Lewis, Willem B. van der Hoeven, and James A. Yankosky, "A 50-ns 16-Mb DRAM with a 10-ns data rate and on-chip ECC," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1118–1128, October 1990.
- [9] Timothy C. May and Murray H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. ED-26, no. 1, pp. 2–9, January 1979.

- [10] Loek Boonstra, Cees W. Lambrechtse, and Roelof H. W. Salters, "A 4096-b one-transistor per bit random-access memory with internal timing and low dissipation," *IEEE Journal of Solid-State Circuits*, vol. SC-8, no. 5, pp. 305-310, October 1973.
- [11] Clinton Kuo, Nori Kitagawa, Deene Ogden, and John Hewkin, "16-k RAM built with proven process may offer high start-up reliability," *Electronics*, vol. 49, no. 10, pp. 81-86, May 1976.
- [12] James A. Gasbarro and Mark A. Horowitz, "A single-chip, functional tester for VLSI circuits," in *IEEE International Solid-State Circuits Conference: Digest of Technical Papers*, pp. 84-85, 268, February 1990.
- [13] Don Speck, "The Mosaic fast 512K scalable CMOS dRAM," in *Advanced Research in VLSI: Proceedings of the 1991 University of California / Santa Cruz Conference* (Carlo H. Séquin, ed.), pp. 229-244, The MIT Press, March 1991.
- [14] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*. Reading, Massachusetts: Addison-Wesley, 1990.
- [15] Ronald J. C. Chwang, Mike Choi, Dan Creek, Seth Stern, Perry H. Pelley, III, Joseph D. Schutz, Paul A. Warkentin, Mark T. Bohr, and Ken Yu, "A 70 ns high density 64K CMOS dynamic RAM," *IEEE Journal of Solid-State Circuits*, vol. SC-18, no. 5, pp. 457-463, October 1983.
- [16] Hiroyuki Yamauchi, Toshiaki Yabu, Toshio Yamada, and Michihiro Inoue, "A circuit design to suppress asymmetrical characteristics in high-density DRAM sense amplifiers," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 1, pp. 36-41, February 1990.
- [17] Anantha P. Chandrakasan, Andrew Burstein, and Robert W. Brodersen, "A low-power chipset for a portable multimedia I/O terminal," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 12, pp. 1415-1428, December 1994.
- [18] John K. Ousterhout, "Corner stitching: A data-structuring technique for VLSI layout tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-3, no. 1, pp. 87-100, January 1984.
- [19] Frederick P. Herrmann and Charles G. Sodini, "A dynamic associative processor for machine vision applications," *IEEE Micro*, vol. 12, no. 3, pp. 31-41, June 1992.
- [20] Frederick P. Herrmann and Charles G. Sodini, "A 256-element associative parallel processor," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 4, pp. 365-370, April 1995.
- [21] Jeffrey C. Gealow, Frederick P. Herrmann, Lawrence T. Hsu, and Charles G. Sodini, "System design for pixel-parallel image processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp. 32-41, March 1996.

- [22] Lawrence T. Hsu, "A controller architecture for associative processors," Master's thesis, Massachusetts Institute of Technology, August 1993.
- [23] Daphne Yong-Hsu Shih, "A data path for a pixel-parallel image processing system," Master's thesis, Massachusetts Institute of Technology, October 1995.
- [24] Gary Hall, "Control and application of a pixel-parallel image processing system," Master's thesis, Massachusetts Institute of Technology, February 1997.
- [25] Kenneth E. Batchner, "The multidimensional access memory in STARAN," *IEEE Transactions on Computers*, vol. C-26, no. 2, pp. 174-177, February 1977.
- [26] W. Daniel Hillis, *The Connection Machine*. Cambridge, Massachusetts: The MIT Press, 1985.
- [27] Kenneth E. Batchner, "Array control unit," in *The Massively Parallel Processor* (Jerry L. Potter, ed.), pp. 170-190, Cambridge, Massachusetts: The MIT Press, 1985.
- [28] Claus M. Habiger and R. Miké Lea, "Hybrid-WSI: A massively parallel computing technology?," *Computer*, vol. 26, no. 4, pp. 50-61, April 1993.
- [29] Wayne M. Loucks, Martin Snelgrove, and Safwat G. Zaky, "A vector processor based on one-bit microprocessors," *IEEE Micro*, vol. 2, no. 1, pp. 53-62, February 1982.
- [30] Bjarne Stroustrup, *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley, second ed., 1991.
- [31] Jeffrey Carl Gealow, "Impact of processing technology on DRAM sense amplifier design," Master's thesis, Massachusetts Institute of Technology, June 1990.
- [32] Pallab K. Chatterjee, Geoffrey W. Taylor, Al F. Tasch, Jr., and Horng-Sen Fu, "Leakage studies in high-density dynamic MOS memory devices," *IEEE Journal of Solid-State Circuits*, vol. SC-14, no. 2, pp. 486-498, April 1979.
- [33] Jeffrey C. Gealow, Gary W. Hall, Daphne Y. Shih, and Charles G. Sodini, *Demonstration of a Pixel-Parallel Image Processing System*, January 1997. Video recording produced by the Massachusetts Institute of Technology Center for Advanced Educational Services.
- [34] Berthold K. P. Horn and Brian G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185-203, August 1981.
- [35] James J. Little, Heinrich H. Bülthoff, and Tomaso Poggio, "Parallel optical flow using local voting," in *Second International Conference on Computer Vision*, (Washington, D.C.), pp. 454-459, IEEE Computer Society, IEEE Computer Society Press, December 1988.

- [36] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, California: Morgan Kaufmann, 1994.
- [37] Kazuo Yano, Toshiaki Yamanaka, Takashi Nishida, Masayoshi Saito, Katsuhiko Shimohigashi, and Akihiro Shimizu, "A 3.8-ns CMOS 16×16 -b multiplier using complementary pass-transistor logic," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, pp. 388-395, April 1990.
- [38] Nobuyuki Yamashita, Tooru Kimura, Yoshihiro Fujita, Yoshiharu Aimoto, Takashi Manabe, Shin'ichiro Okazaki, Kazuyuki Nakamura, and Masakazu Yamashina, "A 3.84 GIPS Integrated Memory Array Processor with 64 processing elements and 2-Mb SRAM," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 11, pp. 1336-1343, November 1994.
- [39] Stephen Trimberger, "A reprogrammable gate array and applications," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1030-1041, July 1993.
- [40] Karl Gutttag, Robert J. Gove, and Jerry R. Van Aken, "A single-chip multiprocessor for multimedia: The MVP," *IEEE Computer Graphics and Applications*, vol. 12, no. 6, pp. 53-64, November 1992.
- [41] K. Balmer, N. Ing-Simmons, P. Moyse, I. Robertson, J. Keay, M. Hammes, E. Oakland, R. Simpson, G. Barr, and D. Roskell, "A single chip multimedia video processor," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 91-94, May 1994.
- [42] John L. Smith, "Implementing median filters in XC4000E FPGAs," *XCELL: The Quarterly Journal for Xilinx Programmable Logic Users*, no. 23, p. 16, fourth quarter 1996.
<http://www.xilinx.com/xcell/xcell23.htm>.
- [43] Xilinx, Inc., San Jose, California, *The Programmable Logic Data Book*, September 1996.
- [44] Woobin Lee, Yongmin Kim, Robert J. Grove, and Christopher J. Read, "MediaStation 5000: Integrating video and audio," *IEEE MultiMedia*, vol. 1, no. 2, pp. 50-61, Summer 1994.
- [45] Texas Instruments, Houston, Texas, *TMS320C80 Digital Signal Processor: Data Sheet*, March 1996. SPRS023A.
<http://www-s.ti.com/sc/psheets/sprs023a/sprs023a.pdf>.
- [46] Stanley A. White, "Applications of distributed arithmetic to digital signal processing: A tutorial review," *IEEE ASSP Magazine*, vol. 6, no. 3, pp. 4-19, July 1989.

- [47] Gregory Ray Goslin, "A guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance," in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic* (John Schewel, Peter M. Athanas, V. Michael Bove, Jr., and John Watson, eds.), vol. 2914 of *Proceedings*, pp. 321-331, SPIE-The International Society for Optical Engineering, November 1996.
- [48] Les Mintzer, "Digital filtering in FPGAs," in *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems & Computers* (Avtar Singh, ed.), pp. 1373-1377, IEEE Computer Society Press, October-November 1994.
- [49] Xilinx, "Distributed arithmetic Laplacian filter," *XCELL: The Quarterly Journal for Xilinx Programmable Logic Users*, no. 20, pp. 38-40, first quarter 1996.
<http://www.xilinx.com/xcell/xcell20.htm>.
- [50] Pierpaolo Baglietto, Massimo Maresca, Mauro Migliardi, and Nicola Zingirian, "Image processing on high-performance RISC systems," *Proceedings of the IEEE*, vol. 84, no. 7, pp. 917-930, July 1996.
- [51] Daniel W. Dobberpuhl, Richard T. Witek, Randy Allmon, Robert Anglin, David Bertucci, Sharon Britton, Linda Chao, Robert A. Conrad, Daniel E. Dever, Bruce Gieseke, Soha M. N. Hassoun, Gregory W. Heoppner, Kathryn Kuchler, Maureen Ladd, Burton M. Leary, Liam Madden, Edward J. McLellan, Derrick R. Meyer, James Montanaro, Donald A. Priore, Vidya Rajagopalan, Sridhar Samudrala, and Sribalan Santhanam, "A 200-MHz 64-b dual-issue CMOS microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, pp. 1555-1567, November 1992.
- [52] Edward McLellan, "The Alpha AXP architecture and 21064 processor," *IEEE Micro*, vol. 13, no. 3, pp. 36-47, June 1993.
- [53] Lavi A. Lev, Andy Charnas, Marc Tremblay, Alexander R. Dalal, Bruce A. Frederick, Chakra R. Srivatsa, David Greenhill, Dennis L. Wendell, Duy Dinh Pham, Eric Anderson, Hemraj K. Hingarh, Inayat Razzack, James M. Kaku, Ken Shin, Marc E. Levitt, Michael Allen, Philip A. Ferolito, Richard L. Bartolotti, Robert K. Yu, Ronald J. Melanson, Shailesh I. Shah, Sophie Nguyen, Sundari S. Mitra, Vinita Reddy, Vidyasagar Ganesan, and Willem J. de Lange, "A 64-b microprocessor with multimedia support," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1227-1238, November 1995.
- [54] Marc Tremblay and J. Michael O'Connor, "UltraSparc I: A four-issue processor supporting multimedia," *IEEE Micro*, vol. 16, no. 2, pp. 42-50, April 1996.
- [55] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner, "The Visual Instruction Set (VIS) in UltraSPARC," in *COMPCON '95: Digest of Papers*, pp. 462-469, IEEE Computer Society, March 1995.

- [56] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He, "VIS speeds new media processing," *IEEE Micro*, vol. 16, no. 4, pp. 10-20, August 1996.

Appendix A

Chip Data

A.1 Supplies and Signals

This section documents the prototype chip's external supplies and signals. The chip was tested under the conditions specified in Table A.3.

Supplies

Board-level bypass capacitors should be placed between `vdd` and `vss`, `vpp` and `vss`, and `vhh` and `vll`. Both `vss` and `vll` should be tied to ground.

<code>vss</code>	Low-voltage supply for internal circuits and <i>n</i> -channel substrate. Eight pads.
<code>vdd</code>	High-voltage supply for internal circuits. Eight pads.
<code>vpp</code>	High-voltage supply for wordline drivers and platelines. Four pads.
<code>vll</code>	Low-voltage supply for interface circuits. Ten pads.
<code>vhh</code>	High-voltage supply for interface circuits. Ten pads.

Clocks

<code>clk0 clk1</code>	Overlapping clock signals. The falling edges of <code>clk0</code> and <code>nck0</code> should be coincident. Instructions begin after the falling edge of <code>clk1</code> . Edges must be arranged as shown in Figure A.1.
------------------------	---

Instruction

All instruction signals, except *i3*, are stored by latches. The latches are opened after the rising edge of *clk1* and closed after the falling edge of *clk0*.

<i>i3</i>	Chip enable input.
<i>i0-i2</i>	Operation code, as described in Table A.1.
<i>a0-a6</i>	Memory address.
<i>f0-f7</i>	Boolean function, as described in Table A.2.
<i>fn fs fe fw</i>	Interconnection function.
<i>la lb lc</i>	Latch load inputs.
<i>ld le</i>	

Interchip Communication

<i>nck0 nck1</i>	Overlapping clock signals. The falling edges of <i>nck0</i> and <i>clk0</i> should be coincident. The edges of <i>nck0</i> and <i>nck1</i> should be spaced evenly.
<i>n0-15</i> <i>s0-15</i> <i>e0-15</i> <i>w0-15</i>	Bidirectional connections for the four edges of the processing element array. Each connection serves four processing elements on the perimeter of the array. Each connection should be tied to the corresponding connection of an adjacent chip or should be left open. 4 perimeter processing elements per pad × 16 pads per edge × 4 edges = 256 perimeter processing elements.

Serial Access Memory

<i>sck1 sck2</i>	Shift control signals.
<i>str sld</i>	Transfer and load control signals.
<i>si0-1</i>	Serial access memory inputs.
<i>so0-1</i>	Serial access memory outputs.

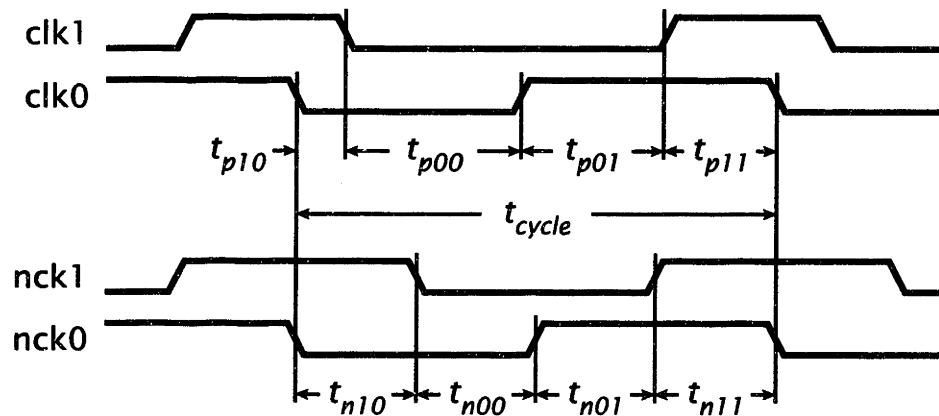


Figure A.1: Clock timing. The intervals t_{p10} , t_{p00} , t_{p01} , and t_{p11} control four different aspects of timing within the processing element array. They need not be equal. The intervals t_{n10} , t_{n00} , t_{n01} , and t_{n11} control interchip communication timing. They should all be one-fourth the cycle time, t_{cycle} .

Table A.1
Operation Codes

Name	i2	i1	i0	Interchip Communication	Memory Activity	
					la = 0	la = 1
Zero	0	0	0	none	none	read
Array	0	0	1	none	none	read
Write	0	1	0	none	write	undefined
Refresh	0	1	1	none	refresh	read
North	1	0	0	north \Rightarrow south	none	read
South	1	0	1	south \Rightarrow north	none	read
East	1	1	0	east \Rightarrow west	none	read
West	1	1	1	west \Rightarrow east	none	read

Table A.2
Function Generator Truth Table

Latch C	Latch B	Latch A	Result
0	0	0	f0
0	0	1	f1
0	1	0	f2
0	1	1	f3
1	0	0	f4
1	0	1	f5
1	1	0	f6
1	1	1	f7

Table A.3
Operating Conditions

vss	0.0 V	t_{p00}	22 ns	t_{n00}	15 ns
vdd	2.5 V	t_{p01}	18 ns	t_{n01}	15 ns
vpp	3.3 V	t_{p11}	14 ns	t_{n11}	15 ns
vll	0.0 V	t_{p10}	6 ns	t_{n10}	15 ns
vhh	3.3 V				

A.2 Pads

The prototype chip has 144 pads. Pad locations are shown in Figure A.2. Pad assignments are listed in Table A.4.

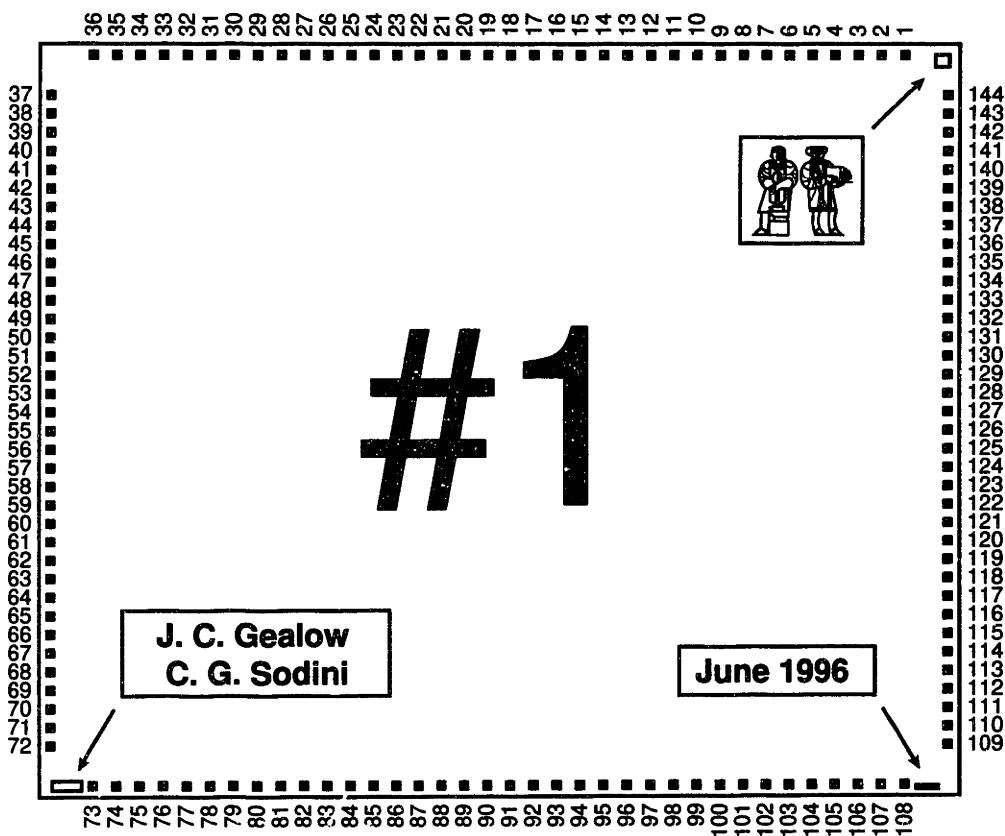


Figure A.2: Pad locations. The chip has large metal features in three corners. The upper right corner has an M.I.T. logo. The lower left corner has the names 'J. C. Gealow' and 'C. G. Sodini'. The lower right corner has the date 'June 1996'. The orientation of the marking '#1' (which is not a physical feature) corresponds to the orientation of the layout coordinate system.

Table A.4
Pad Assignments

1	vll	37	vhh	73	vll	109	vhh
2	n12	38	w1	74	s3	110	e14
3	n13	39	w0	75	s2	111	e15
4	n10	40	w3	76	s5	112	e12
5	vhh	41	vll	77	vhh	113	vll
6	n11	42	w2	78	s4	114	e13
7	n8	43	w5	79	s7	115	e10
8	n9	44	w4	80	s6	116	e11
9	vhh	45	vll	81	vhh	117	vll
10	fs	46	w7	82	a1	118	e8
11	fw	47	w6	83	a0	119	e9
12	f7	48	i0	84	a3	120	sck1
13	vll	49	vss	85	vll	121	vss
14	fe	50	so0	86	a2	122	sck2
15	fn	51	i1	87	a4	123	si1
16	f5	52	clk0	88	a5	124	sld
17	f6	53	clk1	89	a6	125	str
18	vss	54	vpp	90	vss	126	vpp
19	vdd	55	vpp	91	vdd	127	vpp
20	f4	56	i2	92	lb	128	si0
21	f2	57	so1	93	ld	129	nck0
22	f3	58	la	94	i3	130	nck1
23	f1	59	w8	95	lc	131	e7
24	vhh	60	vdd	96	vhh	132	vdd
25	f0	61	w9	97	le	133	e6
26	n7	62	w10	98	s8	134	e5
27	n6	63	w11	99	s9	135	e4
28	n4	64	w13	100	s11	136	e2
29	n5	65	w12	101	s10	137	e3
30	vll	66	vhh	102	vll	138	vhh
31	n3	67	w14	103	s12	139	e1
32	n2	68	w15	104	s13	140	e0
33	vss	69	vss	105	vss	141	vss
34	vdd	70	vdd	106	vdd	142	vdd
35	n1	71	s1	107	s14	143	n14
36	n0	72	s0	108	s15	144	n15

A.3 Packaging

The prototype chip is packaged in a Kyocera SD-560-8532 pin grid array. Figure A.3 shows the orientation of the die within the package. Figure A.4 shows the bond wires connecting the chip to the lead frame. Figure A.5 shows the pin assignments.

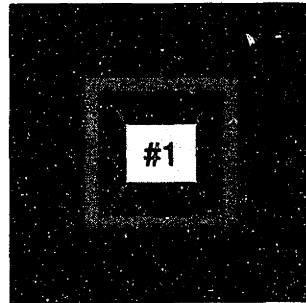


Figure A.3: Die orientation (top view). Kyocera SD-560-8532 package.

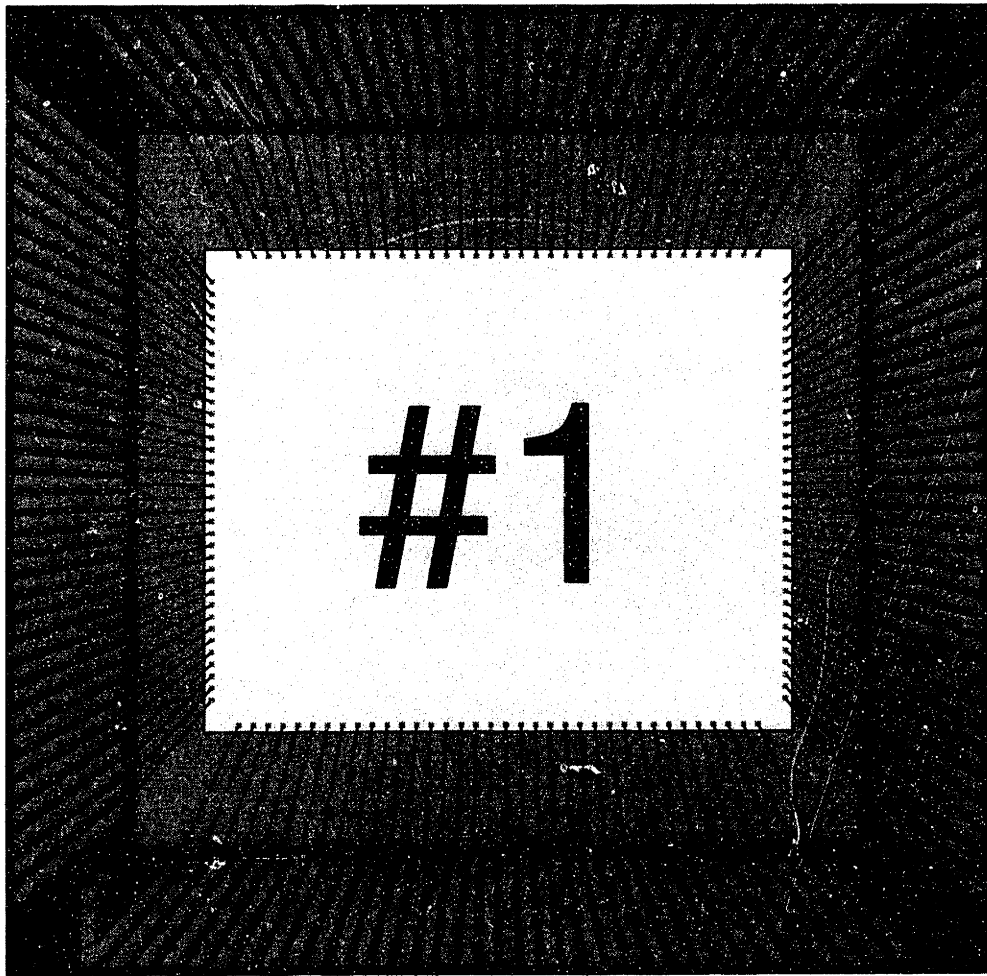


Figure A.4: Bonding diagram (top view). Kyocera SD-560-8532 package.

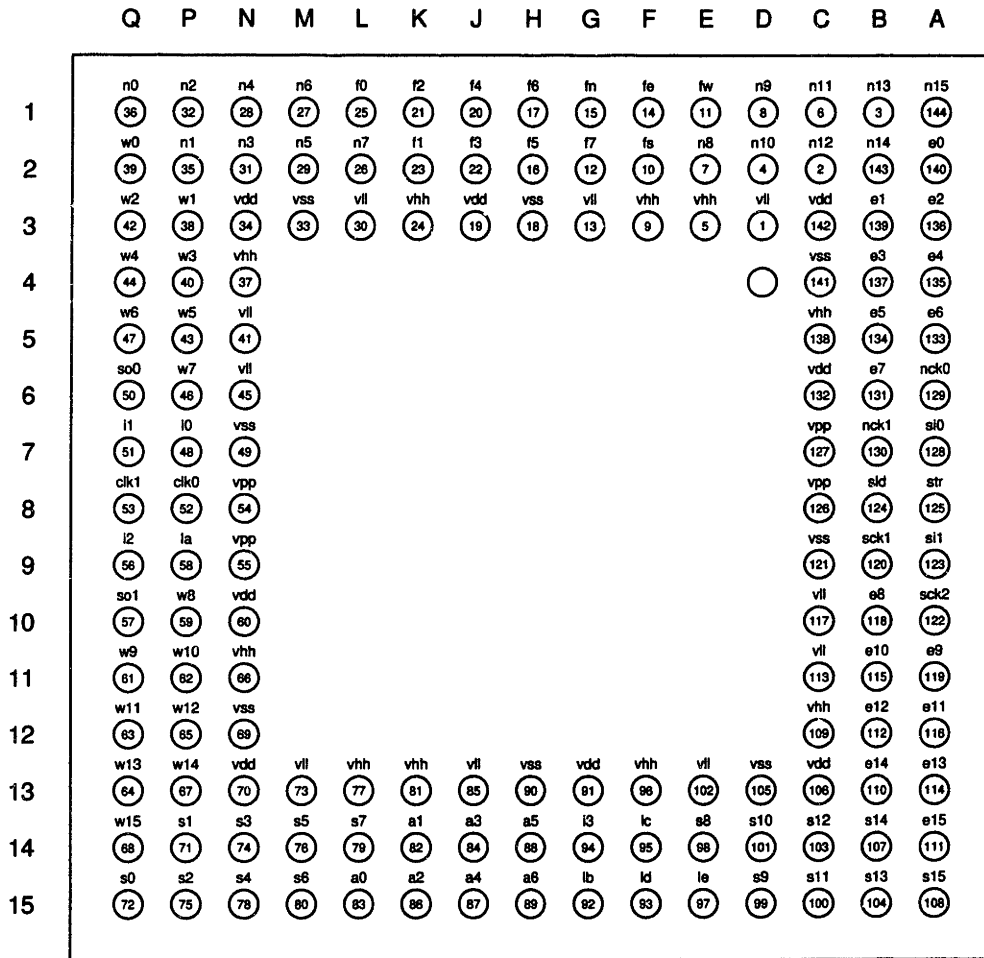


Figure A.5: Pin assignments (top view). Kyocera SD-560-8532 package. Pin D4 is not connected

Appendix B

Layout Design Rules

The rules used for layout design are based on the SCMOS_SUBM rules developed by MOSIS.

	<i>well (nwell, pwell)</i>	
1.1	well width	3.60 μm
1.2	well spacing, different potential	5.40 μm
1.3	well spacing, same potential	1.80 μm
	<i>active</i>	
2.1	active width	0.90 μm
2.2	active spacing	0.90 μm
2.3	diffusion to bulk spacing	1.80 μm
2.3	bulk enclosure of diffusion	1.80 μm
2.4	ohmic to bulk spacing	0.90 μm
2.4	bulk enclosure of ohmic	0.90 μm
	<i>poly</i>	
3.1	poly width	0.60 μm
3.2	poly spacing	0.90 μm
3.3	gate enclosure of active	0.60 μm
3.4	active enclosure of gate	0.90 μm
3.5	field poly to active spacing	0.30 μm

	<i>select (nselect, pselect)</i>	
4.1	select to channel spacing	0.90 μm
4.1	select enclosure of channel	0.90 μm
4.2	select to active spacing	0.60 μm
4.2	select enclosure of active	0.60 μm
4.3	select to active contact spacing	0.30 μm
4.3	select enclosure of active contact	0.30 μm
4.4	select width	0.60 μm
4.4	select spacing	0.60 μm
	<i>contact to poly</i>	
5B.1	contact size	0.60 μm \times 0.60 μm
5B.2	poly enclosure of contact	0.30 μm
5B.3	contact spacing on same poly	0.90 μm
5B.4	contact spacing on different poly	1.80 μm
5B.5	poly contact to poly spacing	1.35 μm
5B.6	poly contact to active spacing, one contact	0.75 μm
5B.7	poly contact to active spacing, many contacts	0.90 μm
	<i>contact to active</i>	
5B.1	contact size	0.60 μm \times 0.60 μm
6B.2	active enclosure of contact	0.45 μm
6B.3	contact spacing on same active	0.90 μm
6B.4	contact spacing on different active	2.10 μm
6B.5	active contact to active spacing	1.50 μm
6B.6	active contact to gate spacing	0.60 μm
6B.7	active contact to field poly spacing, one contact	0.60 μm
6B.8	active contact to field poly spacing, many contacts	0.90 μm
6B.9	active contact to poly contact spacing	1.20 μm
	<i>metal1</i>	
7.1	metal1 width	0.90 μm
7.2	metal1 spacing	0.90 μm
7.3	metal1 enclosure of poly contact	0.30 μm
7.4	metal1 enclosure of active contact	0.30 μm

	<i>via</i>	
8.1	via size	0.60 μm \times 0.60 μm
8.2	via spacing	0.90 μm
8.3	metal1 enclosure of via	0.30 μm
8.4	via to poly edge spacing	0.60 μm
8.4	via to active edge spacing	0.60 μm
8.5	via to contact spacing	0.60 μm
	<i>metal2</i>	
9.1	metal2 width	0.90 μm
9.2	metal2 spacing	1.20 μm
9.3	metal2 enclosure of via	0.30 μm
	<i>via2</i>	
14.1	via2 size	0.60 μm \times 0.60 μm
14.2	via2 spacing	0.90 μm
14.3	metal2 enclosure of via2	0.30 μm
14.4	via2 to via spacing	0.60 μm
	<i>metal3</i>	
15.1	metal3 width	1.80 μm
15.2	metal3 spacing	1.20 μm
15.3	metal3 enclosure of via2	0.60 μm
	<i>overglass</i>	
10.1	bonding pad width	100 μm
10.2	probe pad width	75 μm
10.3	pad enclosure of glass	6 μm
10.4	pad space to unrelated metal2	30 μm
10.5	pad space to unrelated metal1	15 μm
10.5	pad space to unrelated poly	15 μm
10.5	pad space to unrelated active	15 μm

Appendix C

Array Code

Table C.1
Neighbor Difference Code, $D \leftarrow M - M^{north}$

Operation Code	Memory Address	Boolean Function	Interconnect Function			Latch Load			Comments			
			f_w	f_E	f_N	le	ld	lc		lb	la	
1 Zero	M_0	1	0	0	0	0	1	0	0	0	1	$E \leftarrow 1, A \leftarrow Mem[M_0]$
2 North	—	—	—	—	—	—	0	0	0	0	0	
3 Zero	—	0	0	0	0	1	0	0	0	1	0	$B \leftarrow A^{north}$
4 Write	D_0	$A \oplus B$	0	0	0	0	0	1	0	0	0	$D \leftarrow diff., Mem[D_0] \leftarrow D$
5 Zero	M_1	$\bar{A} \wedge B$	0	0	0	0	0	1	0	1	0	$C \leftarrow borrow, A \leftarrow Mem[M_1]$
6 North	—	—	—	—	—	—	0	0	0	0	0	
7 Zero	—	0	0	0	0	1	0	0	0	1	0	$B \leftarrow A^{north}$
8 Write	D_1	$A \oplus B \oplus C$	0	0	0	0	0	1	0	0	0	$D \leftarrow diff., Mem[D_1] \leftarrow D$
9 Zero	M_2	$(\bar{A} \wedge B) \vee (\bar{A} \wedge C)$ $\vee (B \wedge C)$	0	0	0	0	0	1	0	1	0	$C \leftarrow borrow, A \leftarrow Mem[M_2]$
10 North	—	—	—	—	—	—	0	0	0	0	0	
11 Zero	—	0	0	0	0	1	0	0	0	1	0	$B \leftarrow A^{north}$
12 Write	D_2	$A \oplus B \oplus C$	0	0	0	0	0	1	0	0	0	$D \leftarrow diff., Mem[D_2] \leftarrow D$
13 Zero	M_3	$(\bar{A} \wedge B) \vee (\bar{A} \wedge C)$ $\vee (B \wedge C)$	0	0	0	0	0	1	0	1	0	$C \leftarrow borrow, A \leftarrow Mem[M_3]$
14 North	—	—	—	—	—	—	0	0	0	0	0	
15 Zero	—	0	0	0	0	1	0	0	0	1	0	$B \leftarrow A^{north}$
16 Write	D_3	$A \oplus B \oplus C$	0	0	0	0	0	1	0	0	0	$D \leftarrow diff., Mem[D_3] \leftarrow D$

Appendix D

Processing Element Timing Control

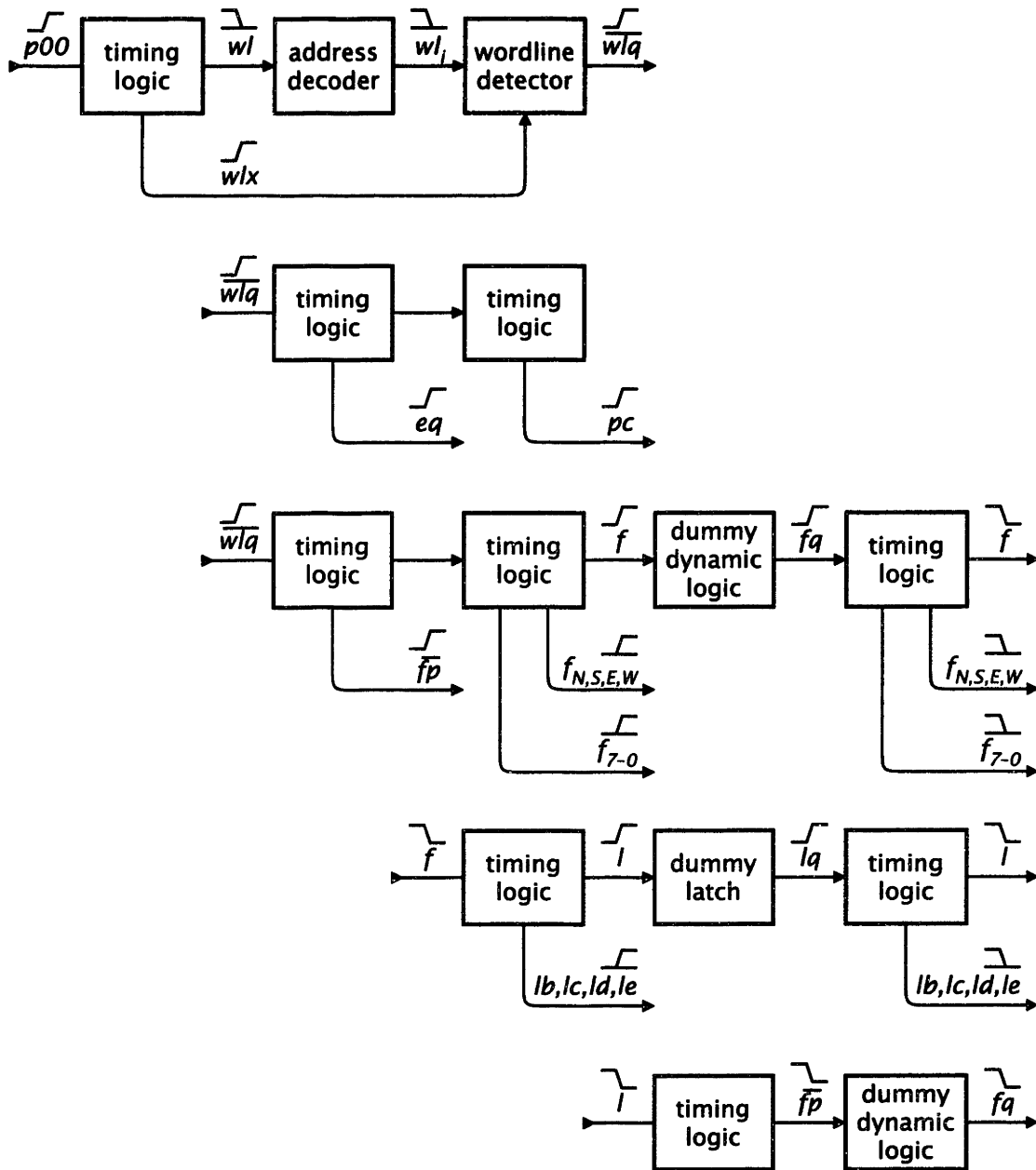


Figure D.1: Processing element timing control, part 1.

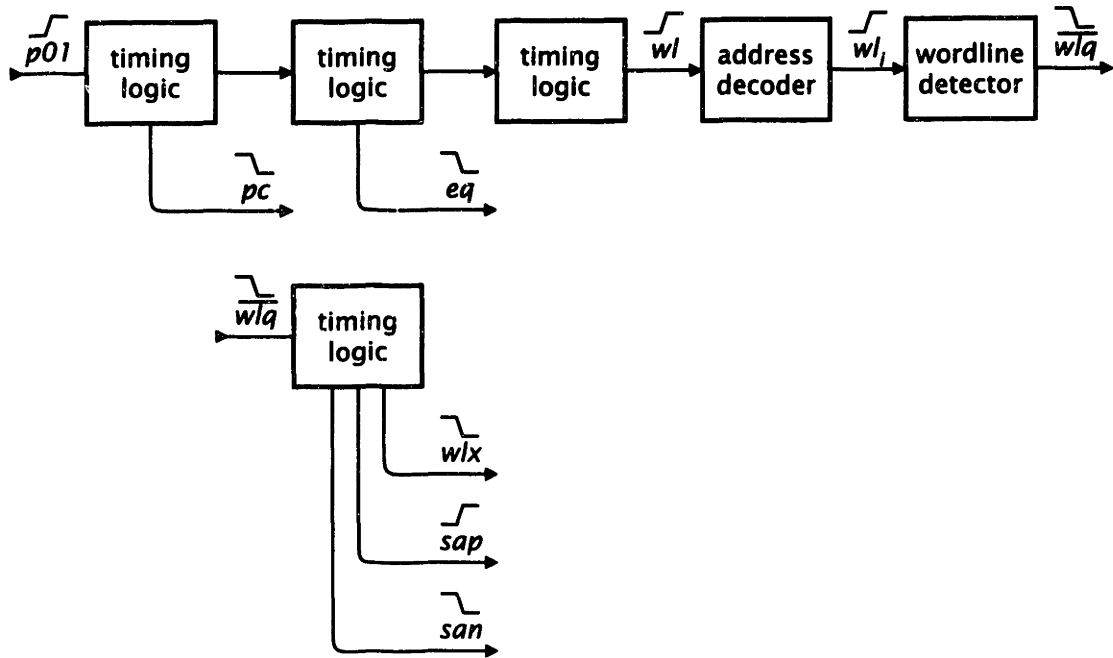


Figure D.2: Processing element timing control, part 2.

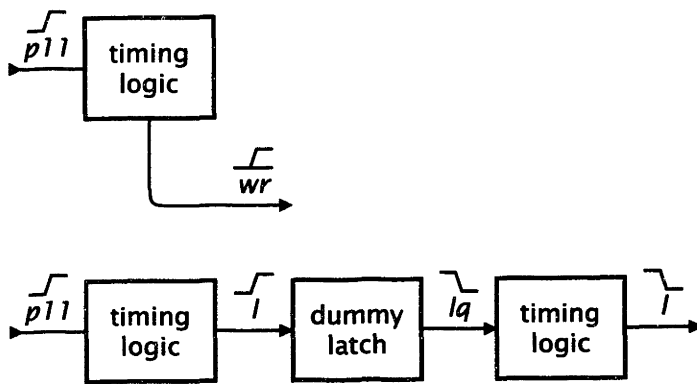


Figure D.3: Processing element timing control, part 3.

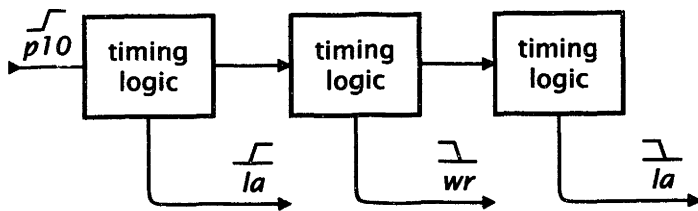


Figure D.4: Processing element timing control, part 4.

THESIS PROCESSING SLIP

FIXED FIELD: ill. _____ name _____

index _____ biblio _____

► COPIES: Archives Aero Dewey Eng Hum
Lindgren Music Rotch Science

TITLE VARIES: ► _____

NAME VARIES: ► _____

IMPRINT: (COPYRIGHT) _____

► COLLATION: 129 p

► ADD. DEGREE: _____ ► DEPT.: _____

SUPERVISORS: _____

NOTES:

cat'r:

date:

► DEPT: E.E.

page: ► <u>J162</u>

► YEAR: 1997 ► DEGREE: Sc.D.

► NAME: GEALOW, Jeffrey Carl