System Design for a Computational-RAM Logic-In-Memory Parallel-Processing Machine

by

Peter M. Nyasulu, B.Sc., M.Eng.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering,
Department of Electronics,
Faculty of Engineering,
Carleton University,
Ottawa, Ontario, Canada

May, 1999

0-612-42803-6

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of this thesis

"System Design for a Computational-RAM Logic-in-Memory Parallel-Processing Machine"

submitted by Peter M. Nyasulu (B.Sc., M.Eng.) in partial fulfillment of the requirements for the degree of Doctor of Philosophy

[illegible]

Chairman, Department of Electronics
Professor Jim Wight

[illegible]

Thesis Co-supervisor
Professor Ralph Mason

[illegible]

Thesis Co-supervisor
Professor Martin Snelgrove

[illegible]

External Examiner
Dr. Lluis Paris
MOSAID Technologies Incorporated

Carleton University
May, 1999

Abstract

Integrating several 1-bit processing elements at the sense amplifiers of a standard RAM improves the performance of massively-parallel applications because of the inherent parallelism and high data bandwidth inside the memory chip. However, implementing such a logic-in-memory system on a host computer poses several challenges because of the small bandwidth at the host system buses, and the different data formats used on the two systems. In this thesis, solutions to these system design issues, including control of the processing elements, interface to the host, data transposition, and application programming, are considered.

A minimal-hardware controller provides high utilization of processing elements while using a simple and general-purpose architecture. A buffer-based host interface unit enhances external data transfers, and minimizes the effect of the host on the performance of the logic-in-memory system. A parallel array-based corner-turning scheme reduces the time to convert data between bit-serial and bit-parallel formats. High-level programming tools, implemented with and using the standard C++ language, hide low-level architectural details of the system, allowing software developers and system analysts to concentrate on implementation details.

Two controller prototypes that interface to the PCI and ISA host buses, and one system prototype, with 64 processing elements and implemented as an ISA expansion card, demonstrate working models of this logic-in-memory system. Simulations using systems with a larger number of processing elements show that the logic-in-memory system yields significant performance speedup over uniprocessor systems when executing massively-parallel applications. Comparisons with other logic-in-memory systems and conventional supercomputers show comparable speed while using less and simpler hardware.

Acknowledgments

I am very grateful to the many people who have contributed to the successful completion of my doctoral studies.

I thank my thesis supervisors, Professor Martin Snelgrove and Professor Ralph Mason, for their guidance and support. I also thank my colleagues, Duncan Elliott, Thinh Le, Christian Cojocaru, Robert Mackenzie, and Philip Lauzon, for their cooperation throughout my research work.

I have also benefited greatly, both academically and financially, from working on the microelectronics bridge camps with Professor John Knight. I am therefore very grateful to him for giving me this opportunity. I also thank Professor Tadeusz Kwasniewski for his support and advice during my early years as a graduate student at Carleton.

I thank the sponsors of my scholarship, the Canadian Commonwealth Scholarship and Fellowship Plan, and acknowledge the help I received from my program managers at Canadian Bureau for International Education and International Council for Canadian Studies.

Studying so far away from home can be very stressful. For this, I am very grateful to the Malawian community in Ottawa and Montreal for the wonderful times we shared together.

I thank my parents and relatives in Malawi for their prayers and love. Most of all I thank my wife and best friend, Stella Nyakasambara, for her unfailing confidence in me, for her patience and love, and for all the help she has provided and all the sacrifices she has made.

Dedicated to my parents,

wadada wa Maynard Nyasulu na wamama wa Bella Nyamkandi

for everything they have done for me

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACU	Array Control Unit
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BiCMOS	Bipolar Complementary Metal Oxide Semiconductor
BPE	Baseline Processing Element
CAD	Computer-Aided Design
CAM	Content-Addressable Memory
CASM	CRAM Assembler
CD	Compact Disc
CLB	Configurable Logic Block
CMASM	CRAM Microcode Assembler
CMC	Canadian Microelectronics Corporation
CMOS	Complementary Metal Oxide Semiconductor
COP	Control Opcode
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRAM	Computational Random Access Memory
DRAM	Dynamic Random Access Memory
DRC	Design Rule Check
EDIF	Electronic Design Interchange Format
EISA	Extended Industry Standard Architecture
EPROM	Erasable Programmable Read-Only Memory
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GIPS	Giga Instructions Per Second
IMAP	Integrated Memory Array Processor

IC	Integrated Circuit
IO (I/O)	Input/Output
IOB	Input/Output Block
IQU	Instruction Queue Unit
IR	Instruction Register
IRAM	Intelligent Random Access Memory
ISA	Industry Standard Architecture
LMS	Least Means Squared
LSI	Large Scale Integration
LVS	Layout Versus Schematic
MAE	Mean Absolute Error
MCK	Memory Clock
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions Per Second
MPEG	Moving Picture Expert Group
MPP	Massively Parallel Processing/Processor
MSE	Mean Squared Error
MUX	Multiplexer
n-D	*n*-dimensional ($n = 1, 2, 3, \ldots$)
NNS	Nearest Neighbor Search
OPS	Operate Strobe
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PE	Processing Element
PIM	Processor In Memory
PIPRAM	Parallel Image Processing Random Access memory
PPR	Partition, Place and Route
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
Reg	Register

RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register Transfer Level
RW (R/W)	Read/Write
SDF	Standard Delay Format
SSD	Sum of Squared Differences
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
TMPA	Temporary or Mask Address
TTOP	Truth-Table Opcode
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VME	Versa Module Eurocard
VQ	Vector Quantization
VRAM	Video Random Access Memory
VSS	VHDL System Simulator
XNF	Xilinx Format
XPE	Extended Processing Element

PCI Bus Signals:

AD	Address/Data bus
CBE	Command and Byte Enable
DEVSEL	Device Select
FRAME	Cycle Frame
IDSEL	Initialization Device Select
IRDY	Initiator Ready
PAR	Parity
PERR	Parity Error
SERR	System Error
STOP	Stop Transaction
TRDY	Target Ready

ISA Bus Signals:

BALE	Bus Address Latch Enable
CHRDY	Channel Ready
IRQ*n*	Interrupt Request Line *n*
LA	Higher address bus (23:17)
M16	Memory 16-bit Chip Select
MRDC	Memory Read Control
MWTC	Memory Write Control
NOWS	No Wait State
SA	Lower address bus (19:0)
SBHE	System Byte High Enable
SD	Data bus

Chapter 1

Introduction

The idea of merging logic and memory on a single chip was conceived as a correction measure for the wide performance gap between the CPU (microprocessor) and its main memory (DRAM) [1]. Traditionally, microprocessors and DRAM are fabricated on totally different technology processes. Microprocessor fabrication lines are usually optimized to yield fast logic, whereas DRAM processes are designed to reduce leakage current and increase cell density. This has resulted in a processor-memory performance gap that increases by more than 50% every year [2]. While a number of solutions, such as sophisticated cache schemes and pipelined processors, have been used to correct this gap, these have not been totally effective and the processor-memory performance gap continues to be a major obstacle to improved computer system performance. In the continuing effort to improve system performance, some recent research work has concentrated on equipping standard memories with some processing power. The primary objective of these logic-in-memory or processing-in-memory systems is to utilize the high data bandwidth and inherent parallelism that is available inside the memory chip. This not only improves system performance, but also improves power consumption and in some cases reduces system cost.

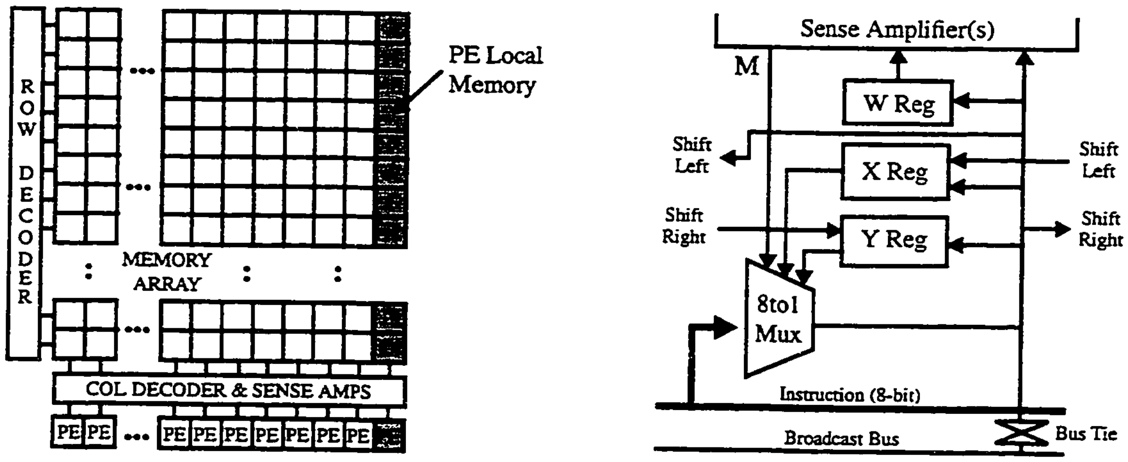
Computational RAM (CRAM) is one of the pioneering research projects in logic-in-memory systems. The following sections give a brief overview of CRAM, and outlines the scope, contributions and organization of this thesis.

1.1 Computational RAM

CRAM is a SIMD-memory hybrid in which very simple 1-bit processing elements (PEs) are integrated at the sense amplifiers of a standard RAM [3]. CRAM is designed to increase the speed of executing massively-parallel applications by utilizing the high bandwidth available at the sense amplifiers. Several PEs have access to the data at the sense amplifiers and operate on it without the need to read the data out of the RAM and transmit it over long buses to the processor. This improves performance and reduces power consumption. Figure 1.1 shows the architecture of CRAM and the 1-bit PE.

The computational logic of a PE consists of an 8-to-1 multiplexer and 3 registers (Y, X, M). The PE computes the result of an operation by using the 3 bits from the registers to select any one bit of the 8-bit instruction. Therefore, a PE instruction is simply the multiplexer truth table output for all 8 possible register combinations for that particular operation. For example, to have a PE result of '1', the instruction is set to 0xFF so that any combination of Y-X-M selects '1' as the output of the multiplexer. The W register is used for conditional write-back of the result to the local memory of the PE. The shift ports and the bus tie are used for inter-PE communication.

Since the PEs are only 1-bit wide, a single operation generally requires several instructions. For example, an *n*-bit addition requires (6*n* + 1) instructions. However, by making the PEs small, a number of them can be integrated in the pitch of one or a few sense amplifiers in order to increase the degree of computational parallelism.



(a) CRAM Architecture

(b) Processing Element

Figure 1.1 CRAM Architecture and Processing Element

1.2 Application to Parallel-Processing

Applications most suited for CRAM, like most massively parallel SIMD machines, are those that have fine grain parallelism and regular communication patterns. Such applications can be found in numerous fields including image processing, databases, video and image compression, digital signal processing, computer-aided design, graphics, and numerical analysis [4], [5], [6], [7]. Specific applications that have been studied for CRAM implementation include image convolution, FIR filters, data mining, fault simulation, and the satisfiability problem [4], as well as general image processing techniques, discrete cosine transform, run-length encoding, scalable and hierarchical vector quantization, and other MPEG-2 algorithms [5]. A few CRAM applications are described in this thesis. These include low-level image processing, basic database applications, vector quantization, and motion estimation.

Figure 1.2 illustrates the implementation of image inversion for a 256 x 256 8-bit image on a 64K-PE CRAM (image inversion is useful in the display of medical images and in producing negative prints of images). The image is spread out across the PEs, with one pixel per PE. All PEs perform the inversion operation in parallel. While each PE takes 24 instructions to invert its pixel, i.e. 1.2 μs for a 20 MHz CRAM, the fact that this is done in parallel for all the 65536 pixels still results in high speedup over a high-performance uniprocessor system. For example, a 400 MHz Pentium processor theoretically takes 7.5 ns to invert one pixel (two memory accesses and one computation). But since it has to

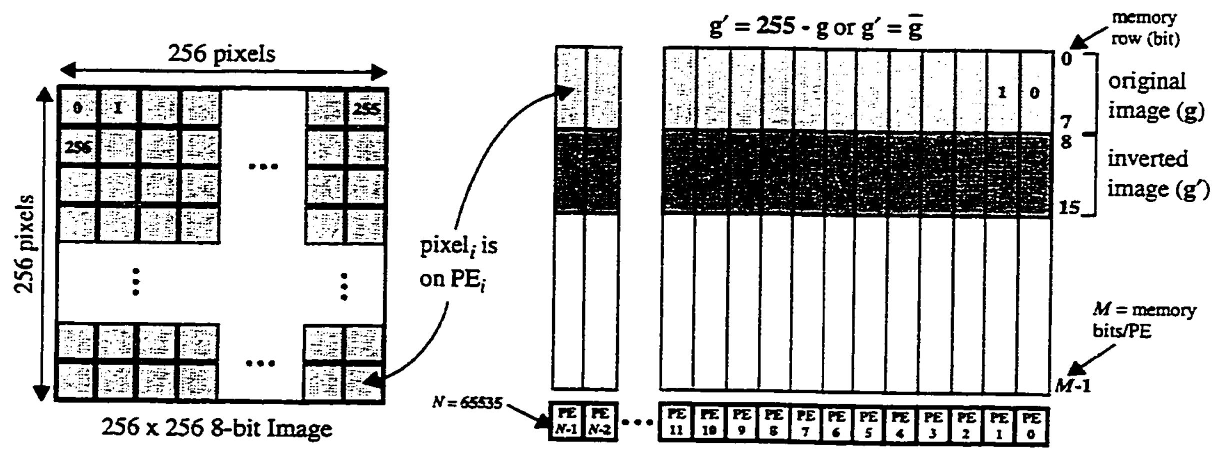


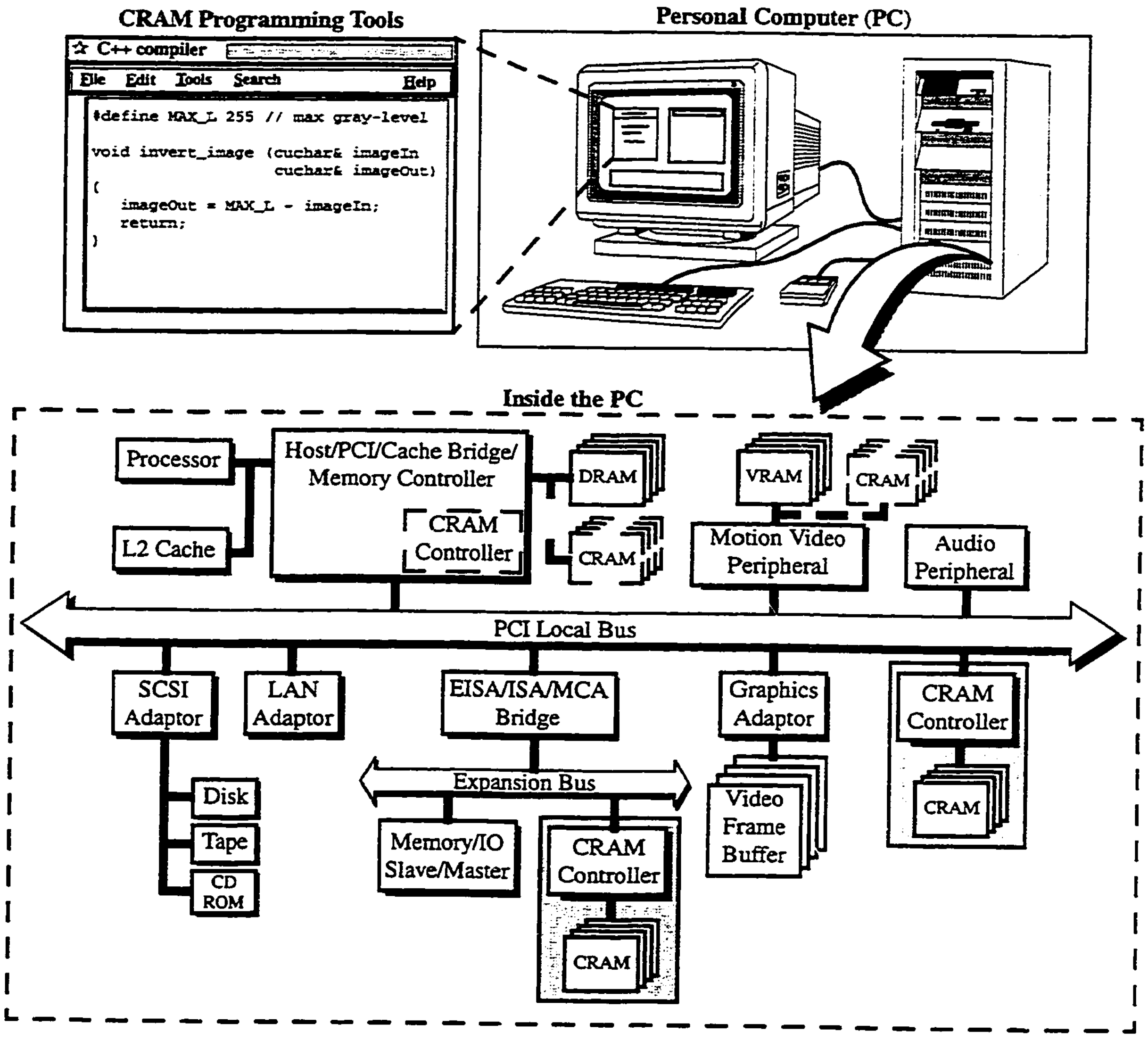
Figure 1.2 Implementation of Image Inversion on CRAM

process 64K pixels sequentially, its total execution time of 0.5 ms is more than 400 times slower than that of the CRAM system. Note that this is just a theoretical number used for illustration. As shown later in the thesis, practical speedups may be slightly lower or higher because of CRAM system design issues. Also, a uniprocessor memory access may not be executed in one CPU cycle because memory systems usually have slower cycle times.

1.3 System Design

As shown later in the thesis, most logic-in-memory systems are designed for specific applications and are implemented mostly on the VME bus or the Sbus. On the other hand, CRAM is designed to be a general-purpose parallel-processing system that can be used on a variety of platforms. One platform specifically targeted in our initial prototypes is the widely-used personal computer (PC) environment. Figure 1.3 shows possible implementations and use of CRAM systems in a typical PC-like environment. The dotted CRAM system is the case where CRAM either replaces or coexists with the standard RAM as the computer main memory or as video RAM. The implementations on the PCI local bus and the expansion buses (shaded boxes) are the ones developed in this work. The main use of the CRAM controller is to allow CRAM application programs to be run from the host computer. The controller acts as a PE array controller as well as an interface to the host bus.

Implementing a CRAM system on a host computer such as the PC poses several challenges. Considering that even a simple operation such as addition requires several instructions to be issued to the 1-bit PEs, the first challenge is to control the PEs and attain high PE utilization while using the small bandwidth of the host buses. Otherwise the PEs would be idle most of the time, thus reducing the overall advantage of CRAM over a uniprocessor system. Second, unlike most SIMD systems that are implemented using several PCBs and housed in large cabinets, a PC-based CRAM system has to be implemented on a single standard-size PC card. This means using as few external components as possible. The third challenge in using CRAM on a standard computer is due to the different data formats used on the two systems. CRAM is bit-serial, while most conventional computer systems are bit-parallel. Therefore, there is need for format conversion or data transposition when data is transferred between the two systems. This difference in data formats also means that programming tools on the host can not be directly used to write programs for CRAM. Therefore, new software tools have to be developed for CRAM, or existing tools on the host have to be modified or enhanced to support CRAM data types. Finally, there is need to minimize any extra CRAM hardware in order to reduce system costs as well as facilitate future integration of all CRAM hardware on a single chip.



```
#define MAX_L 255 // max gray-level

void invert_image (uchar& imageIn
                   uchar& imageOut)
{
   imageOut = MAX_L - imageIn;
   return;
}
```

Figure 1.3 CRAM in a Typical Computer System

1.4 Thesis Scope

The scope of this thesis encompasses the study of the system design challenges outlined in Section 1.3. The thesis argues and proves that even with the limitations imposed by the host computer, i.e. small bandwidth, bit-parallel data format, and constrained PCB size, it is still possible to build a minimal-hardware and general-purpose CRAM system that yields significant performance speedup over conventional uniprocessor systems when executing massively-parallel applications. The main system design issues covered in this work include the following:

- Design and implementation of a minimal-hardware and high-performance PE controller that also enhances the use of CRAM as a general-purpose parallel-processing system.
- Design and implementation of a CRAM-host interface that minimizes the effect of the host on the performance of a CRAM system. This allows the implementation of CRAM on a wide variety of platforms.
- Design and implementation of data transposition schemes that offer reasonable compromise between area/hardware and performance.
- Implementation of a CRAM system prototype to demonstrate a working model of the whole CRAM concept.
- Analysis of the general implementation and performance of CRAM applications.
- Analysis of the effect of different CRAM architectural features on the overall performance of a CRAM system.

A secondary objective of this thesis is to develop a set of high-level software tools that can be used for application programming, system simulation, architectural analysis, and other CRAM development work. Since the CRAM prototypes implemented so far are very small, almost all the analysis work reported in this thesis is based on simulations using the C++ CRAM system simulator developed in this work.

1.5 Thesis Contributions

This thesis contributes to the general system design, implementation, and analysis of a logic-in-memory parallel-processing machine. The major design contributions include:

- A novel constant broadcast unit that improves the performance of operations with constant values by more than 35%. It also simplifies the use of variable-size constants, and reduces the size of the control store, thus reducing the area of the CRAM controller.
- A FIFO-based instruction queue that improves the performance of short-sequence instructions by a factor as high as tenfold. It also allows the CRAM controller to approach an ideal PE controller (100% PE utilization) at a small number of microinstructions per instruction. This translates in increased performance for a wider range of applications.
- The use of read/write buffers for data transfers from the host to CRAM that simplifies synchronization, eases electrical and physical loading of the host bus, and may reduce the time of loading data onto CRAM by as much as 80%.
- A new parallel array-based data transposition approach that is more than a hundred times faster than host-based software transposition, and contributes less than 10% of the total I/O overhead for CRAM systems with more than 4K PEs.
- A simple but innovative approach of grouping microroutines that reduces the required size of the microprogram memory by more than 50%. This small size (less than 256 32-bit words) makes an on-chip control store feasible, even in standard ASIC technologies and FPGAs, and hence reduces the number of components on a CRAM PCB.
- A general-purpose architecture for the controller that enhances the use of CRAM as a general-purpose parallel-processing system.

The major implementation contributions include:

- A CRAM C++ library that is used to write CRAM application programs using the standard C++ language and standard C++ compilers.
- A CRAM C++ simulator that can be used by both hardware and software designers to analyze CRAM architectural features as well as the performance of applications.
- Implementation of two controller prototypes and an ISA-based 64-PE CRAM system prototype to demonstrate working models of the CRAM concept.

In terms of system analysis,

- this work demonstrates that using the design features described above minimizes the effect of the host computer on the performance of a CRAM system. This is important for CRAM as a general-purpose system because it means that CRAM can be implemented on a variety of platforms, including slow host systems such as ISA-based computers and embedded systems that use slow microcontrollers.
- It is also shown that even with a bandwidth-limited host, a CRAM system still yields reasonably high performance for a variety of massively-parallel applications because of the performance-enhancement features of the CRAM controller.
- Using practical applications, this work highlights architectural bottlenecks of CRAM, especially the inter-PE communication network.

Finally, the thesis proposes two new CRAM ideas: an on-chip CRAM controller and a MIMD-SIMD CRAM system, and a pipelined constant-sensitive CRAM PE.

1.6 Thesis Organization

This chapter has outlined the background, motivation, scope and contributions of this thesis. Chapter 2 outlines the reasons behind logic-in-memory systems and describes work related to CRAM. Chapter 3 summarizes the architectural details of CRAM and lists CRAM prototype chips implemented so far. Chapter 4 to Chapter 7 discuss the primary contributions of this thesis. Chapter 4 describes the architecture of the CRAM controller and its interface to the host processor. Chapter 5 discusses the design, implementation, and testing of CRAM prototypes. Chapter 6 describes the different CRAM system software tools that have been developed in this work. Applications and performance analysis of a CRAM system are described in Chapter 7. Chapter 8 summarizes major accomplishments and suggests ideas for future work.

Appendix A provides architectural and implementation details of the CRAM controller. Appendix B provides PCB layout and pinouts of the CRAM system prototypes. Details of CRAM software tools are described in Appendix C. Appendix D lists the C++ source code for all applications described in this thesis.

Chapter 2

Logic-In-Memory Systems

This chapter discusses the background of logic-in-memory systems, and gives examples of CRAM-related work. Section 2.1 discusses the rationale for logic-in-memory systems and outlines the major advantages of these systems. Section 2.2 presents some of the more advanced and well-known logic-in-memory systems, with emphasis on the architecture and size of the processing elements (PEs), as well as system design issues (such as control of the PEs and the interface to the host). Since most logic-in-memory systems are SIMD systems, a brief review of other common SIMD PE control strategies is also given in Section 2.3. The architecture of CRAM itself is presented in Chapter 3.

2.1 Introduction

The idea of logic-in-memory has been around for years. One of the earliest documented cases is the logic-in-memory computer proposed by Harold Stone [1]. Stone argued that since the cost of components was to become heavily dependent on the number of package pins and not on the chip gate count, it was logical to equip the memory with some processing power so that operations can be performed directly in memory and take advantage of the inherent parallelism. This enhanced logic-in-memory cache would then be used as a high-speed buffer between the CPU and the main memory. It can therefore be seen that much of the initial motivation for a logic-in-memory system was to bridge the speed gap between a CPU and its memory. This holds true even today.

The main reason for the differences in speed between the CPU (microprocessor) and its memory (DRAM) has been due to the fact that different fabrication processes are used for the two. Microprocessor fabrication processes are usually optimized for fast transistors and have many metal layers. This results in fast logic, accelerates communication, and simplifies power distribution. On the other hand, DRAM processes have more polysilicon layers in order to have small memory cells. They also have high threshold voltages and thicker oxides in order to reduce leakage current. This increases the density of the DRAM, and also reduces its refresh rate. Because of these differences in processes, microprocessor performance has been improving at a rate of 60% per year, while DRAM access time has been improving at less than 10% a year [2]. This widening performance gap between the processor and the memory is now the major obstacle to improved computer system performance. Using sophisticated multi-level caches has only partially solved the problem since these caches increase memory latency and also do little to increase memory bandwidth because they (SRAMs) are traditionally constrained by their narrow data bus widths. SRAMs are also very expensive when compared to DRAMs.

One solution to the processor-memory performance gap is to combine logic and memory on the same chip. The complexity of the logic that is merged into memory depends on the desired system performance and cost. On one end, a small number of complex full-featured processors or CPUs are integrated into memory. This allows the logic-in-memory system to perform all computations done by a standard microprocessor.

However, this type of system is expensive because of the size of the processors, and it is also difficult to implement a complex logic system in a DRAM process. The other type of logic-in-memory system, such as Computational RAM (CRAM) [3], integrates very simple single-bit processing elements (PEs) at the sense amplifiers of a standard RAM. The PEs add about 5-10% to the area of the RAM, and because of their simplicity, it is easier to implement them even in a DRAM process. Also, since the PEs are small, a big number of them can be integrated in a RAM chip, thus increasing the degree of computational parallelism. The following are advantages of logic-in-memory systems, with particular emphasis on CRAM-type systems:

- **High Memory bandwidth:** Typically, the data available at the RAM sense amplifiers is more than a 1000 times the width of the RAM external bus (which is typically 1, 8, 16, or 32 bits wide). Therefore, processing elements integrated at the sense amplifiers can utilize this high bandwidth to speed up the execution of parallel applications. Figure 2.1 shows the bandwidths available at different points in a computer system. This comparison [8] is based on a system with 256 MBytes of 16 Mb, 50ns DRAM chips, and a 100 MHz CPU with a 64-bit bus.

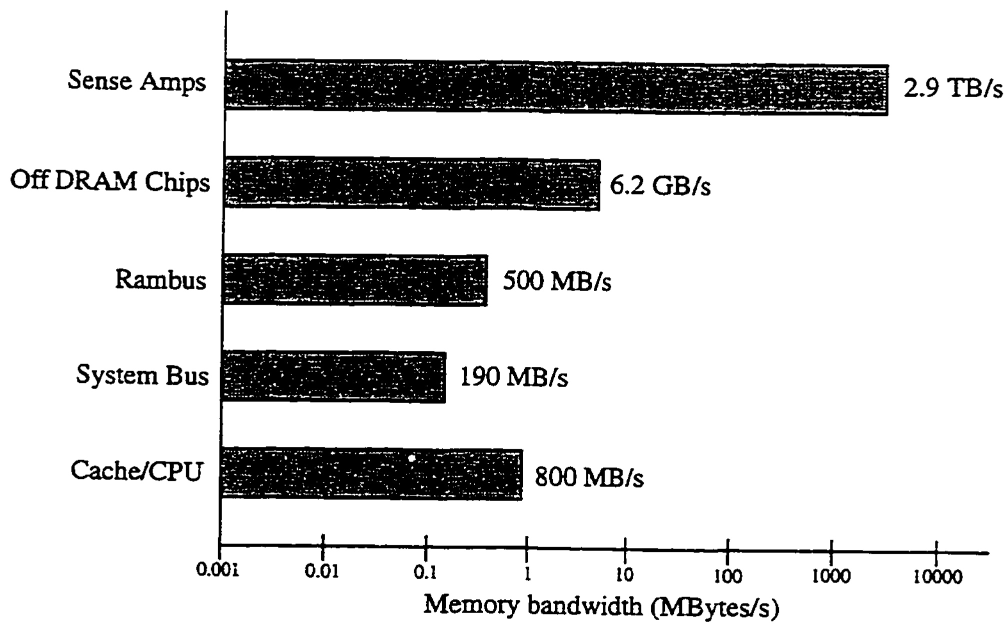


Figure 2.1 Memory Bandwidth in a Computer System

- **Lower Power Consumption:** Logic-in-memory systems improve power consumption. First, since most of the computations are done on the memory chip, there are fewer external memory accesses which would otherwise consume energy in driving the high-capacitance off-chip buses. Second, in a standard RAM chip, a single memory access drives several hundreds of bitlines even though a few tens of data bits are made available to the external RAM bus. This is an obvious wastage of energy. In a logic-in-memory system that integrates PEs at the RAM sense amplifiers, almost all the data bits driven onto the sense amplifiers are used in the computation, thus improving the efficiency of power consumption. For example, in a comparison done in [9], a 200 MHz Pentium accessing its cache transfers data over 10 cm of 32-bit buses. Assuming a 20 pF capacitance per data bit, and a 3.3 V power supply, this data transfer results in energy consumption of about 110 pJ per data bit. On the other hand, the PE's of an equivalent CRAM system with 16 x 1024b CRAM chips and a 50 ns cycle time, access their data over 5 mm metal lines with a total of about 1 pF per line. This results in power consumption of about 5.5 pJ.

- **Reduced Board Size:** A logic-in-memory system may result in fewer chips than a system implemented with discrete RAM and CPU chips. Also, for systems that require modest computation power but whose board area is precious, such as in portable electronic devices, a single chip with merged logic and memory is more attractive.

The CRAM project is one of the pioneering logic-in-memory research efforts. However, in the past five years, a number of universities and companies have pursued this idea. In this chapter, we present some of the more advanced and well-known logic-in-memory systems. This provides a basis of comparison with CRAM, and also gives an overview of the direction of research in such systems. Since most logic-in-memory systems are SIMD systems, a brief review of other common SIMD controllers is also given. The architecture of CRAM itself is presented in Chapter 3.

2.2 Logic-in-Memory Systems

2.2.1 Integrated Memory Array Processor (IMAP)

The integrated Memory Array Processor (IMAP) chip [10] integrates 64 processing elements (PE) with a 2-Mb SRAM. A 7000-transistor PE consists of an 8-bit ALU/Shifter, 14 8-bit registers, 4 1-bit registers, and a 4-bit exchange unit. Each PE has 4K x 8-bit of memory. IMAP was designed by NEC specifically for image processing, hence the 8-bit PEs. It was fabricated in a 0.55 μm BiCMOS double layer metal technology and contains 11 million transistors in a 15.1 x 15.6 mm^2 die area. The chip operates at 40 MHz and has a peak processing performance of 2.56 GIPS and a peak memory bandwidth of 1.28 GBytes/s. Figure 2.2 shows the configuration of the IMAP chip.

As an improvement to IMAP, NEC developed the Parallel Image-Processing RAM (PIP-RAM) [12]. This logic-in-memory image processor integrates 128 PE's and 16 Mb DRAM in a 64 Mb DRAM process technology. As in IMAP, the PE is 8-bits and consists mainly of an 8-bit ALU, a shifter, 24 general-purpose registers, and 5 special-purpose registers. Each PE has 128 Kb of DRAM. The PIP-RAM chip has a die size of 18.8 x 16.7 mm^2 and operates at 30 MHz. It has a peak processing performance of 7.68 GIPS and a peak memory bandwidth of 3.84 GB/s.

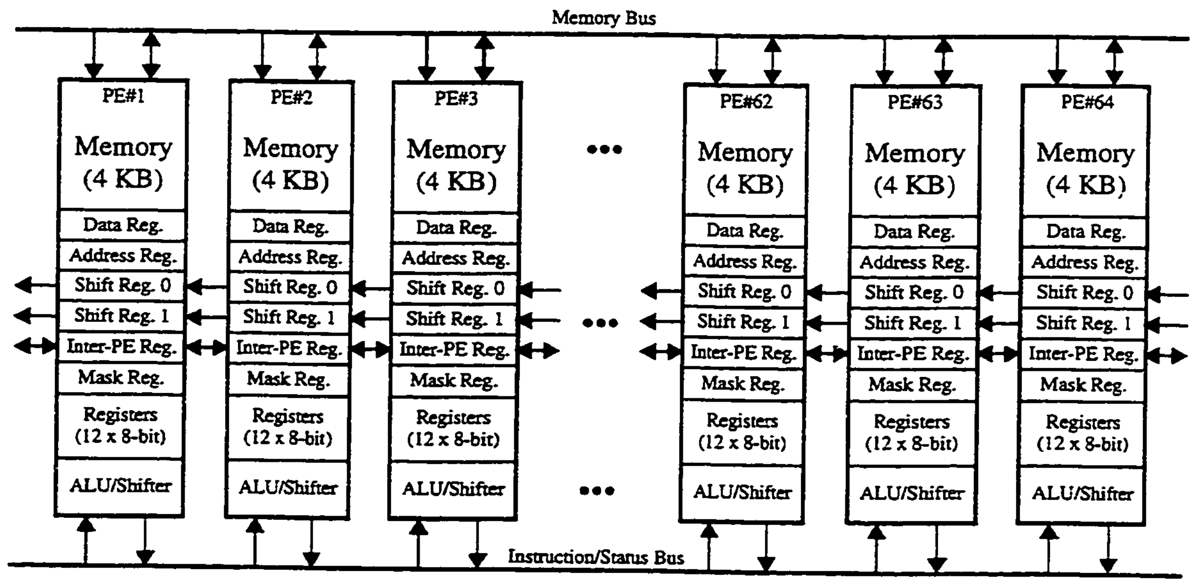


Figure 2.2 Integrated Memory Array Processor Configuration

A real-time vision system (RVS-2) [11] was designed using eight IMAP chips and a custom controller, and it interfaces to a host workstation using the VME bus. Figure 2.3 shows the block diagram of the RVS-2 system. It consists of an IMAP board, a video board, and a host workstation board. The IMAP board consists of eight IMAP LSI's, a controller LSI (RVSC), program memory, data memory, and VME interface.

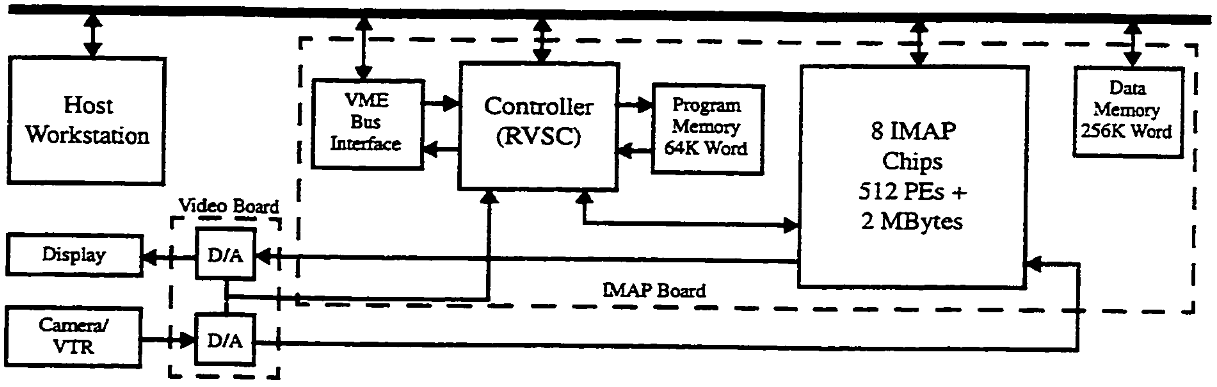


Figure 2.3 RVS-2 Configuration

Figure 2.4 shows the block diagram of the controller. The main functions of the controller are to sequence program execution, arbitrate memory accesses from the host, the PE array, and the controller itself, and to process global data. In order to meet the last requirement, the RVSC has a 16-bit processor, a special function to execute CAM-like functions, and selective access to data or status from an arbitrary PE.

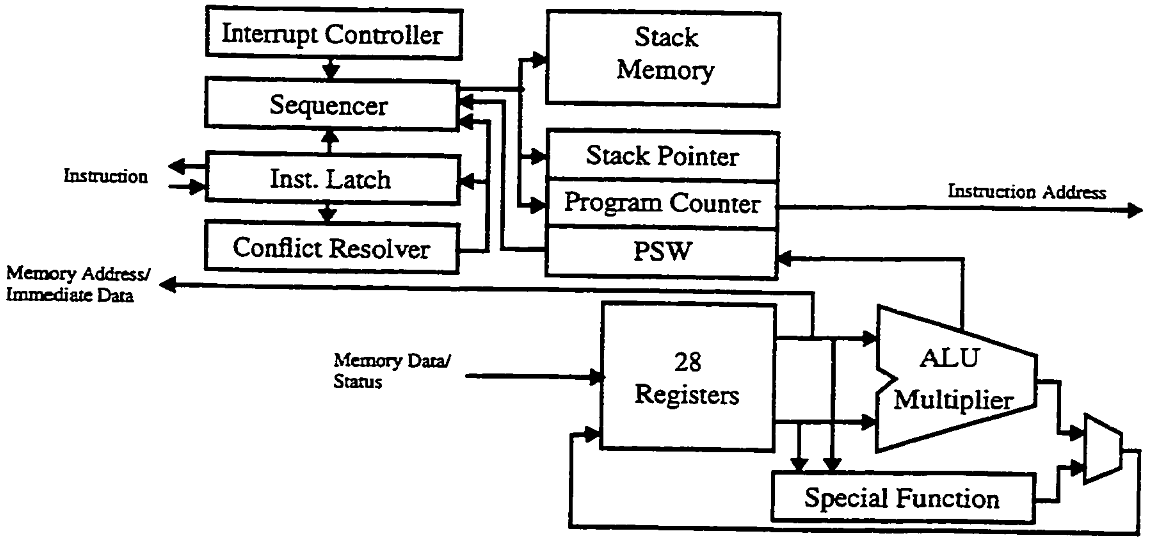


Figure 2.4 RVS-2 Controller (RVSC)

2.2.2 Terasys Processor In Memory (PIM) Array

The Terasys PIM system [13] was designed by researchers at the Supercomputing Research Center (SRC). It contains 2K x 64 bits of SRAM with 64 bit-serial processors. Processors are connected in a linear network and can communicate using a global-OR, partitioned OR and a parallel prefix network. A PIM array unit contains 64 PIM chips (4K processors) spread over two boards. A half dozen Terasys workstations were built at SRC. Each workstation consists of a Sun Sparc-2 workstation, an SBus interface card residing in the sparc cabinet, a Terasys interface board, and 8 PIM array units (32K processors). At an instruction issue rate of 100 ns, the system delivers 3.2 x 10^{11} peak bit operations per second. Figure 2.5 shows the organization of a 16K processor Terasys workstation. Applications on Terasys are programmed in a custom-designed language called dbc (data bit C), which is a superset of ANSI C.

The interface board is the controller for the PIM array and memory. A read and write operation to PIM memory are sent to the interface board which decodes whether it is a normal memory operation or a PIM array command. Each command is 12-bit wide, and indexes into a 4K lookup table. The lookup table contains 25-bit microcode instructions. These microinstructions are generated by a microcode assembler on a per-program basis, and are loaded into the controller board when a Terasys program is initiated. The interface board also provides PIM timers, selects one of two PIM commands based on the current value of the global OR signal, and registers 31 bits of global OR history.

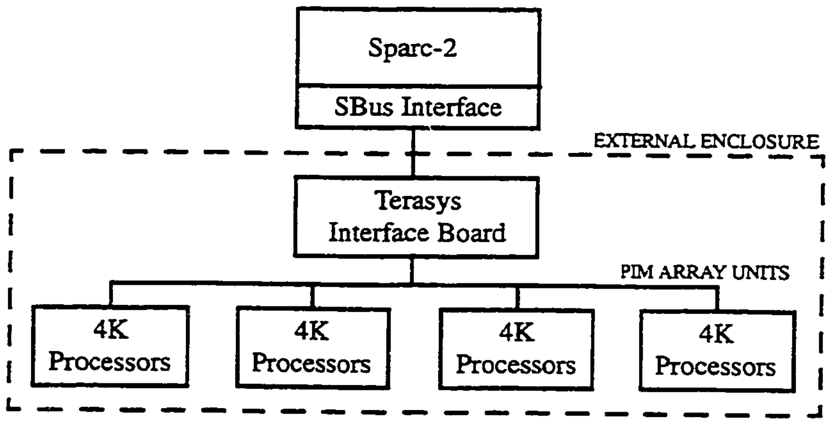


Figure 2.5 A 16K Processor Terasys Workstation

2.2.3 MIT Pixel-Parallel Image Processing System

The MIT image processing system [14] (Figure 2.6) integrates 64 x 64 bit-serial processing elements with 128 bits/PE of DRAM in a 0.6 μm HP CMOS14TB technology. The PE's use a 256 function generator and are connected to its four nearest (South, East, North, and West) neighbors. This system was designed purely for pixel-processing, and hence most of its architectural features (such as data format converters, array controller, and pixel/PE array configuration) were specifically tailored for 8-bit gray scale pixels. The interface to the host computer is through the VME bus. System performance, tested on a limited number of pixel-processing applications (Smoothing and Segmentation, Median Filtering, and Optical Flow) exceeds thirty frames per second.

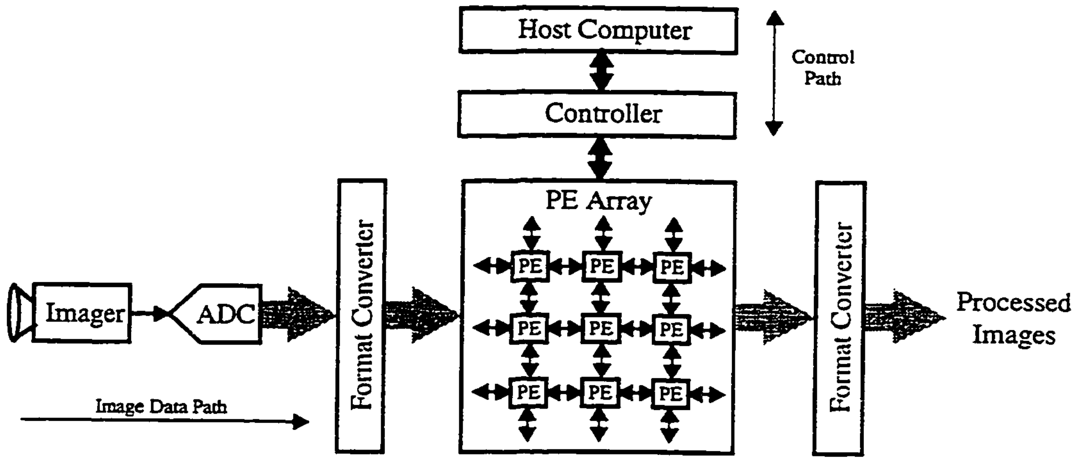


Figure 2.6 MIT Image Processing System

Figure 2.7 shows the controller architecture. Sequences of microinstructions are generated by the host and are stored in the control store. The sequences perform basic arithmetic, comparison, and data movement operations. 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 (100 ns).

The select register and the associated multiplexer are used for operations with scalar variables. Figure 2.8 shows how instruction selection is employed to add 4-bit scalar variable, *b*, to a parallel variable, *A*. Before processing begins, pairs of instruction sequences are stored in the controller. During program execution, the value of b is loaded into the select register, which is 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 pair.

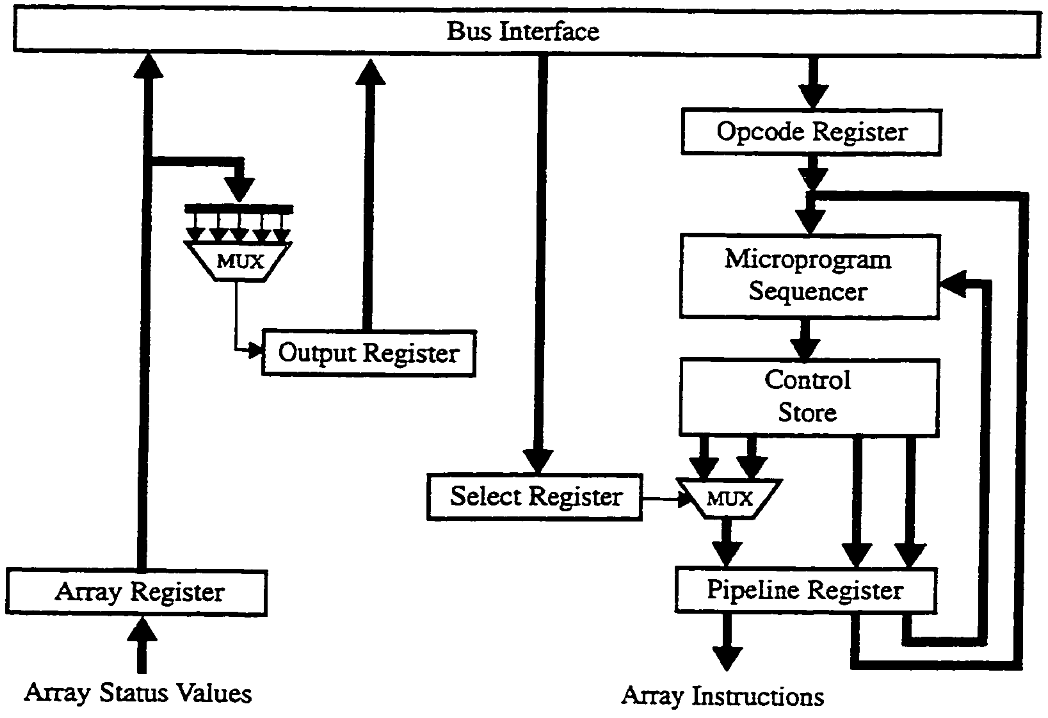


Figure 2.7 MIT Controller Architecture

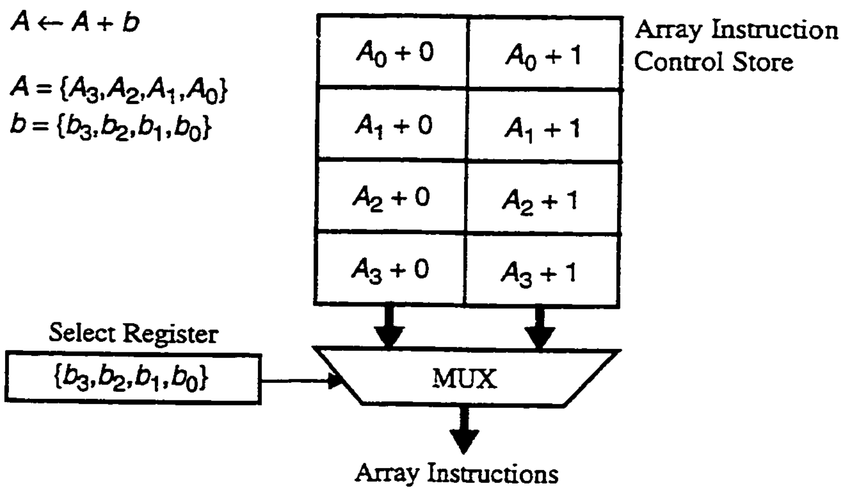


Figure 2.8 Instruction Selection

2.2.4 Intelligent RAM (IRAM)

IRAM [15] is another CRAM-related research effort that aims at merging processing and memory into a single chip to lower memory latency, increase memory bandwidth, and improve energy efficiency. This research is conducted by Patterson's group at the University of California at Berkeley. Though they have not yet built any prototype chips, they have proposed an IRAM vector processor shown in Figure 2.9. The processor includes sixteen 1024-bit-wide ports on the IRAM, thirty-two 64-element vector registers, pipelined vector units for floating-point add, multiply and divide, integer operations, load/store, and multiplication operations. It is projected that in a 0.18 μm DRAM process with a 600 mm^2 chip, a high performance IRAM-based vector accelerator would have eight add-multiply units running at 1000 MHz and sixteen 1-Kbit buses running at 50 MHz, resulting in system performance of 16 Gflops and 100 GB/s.

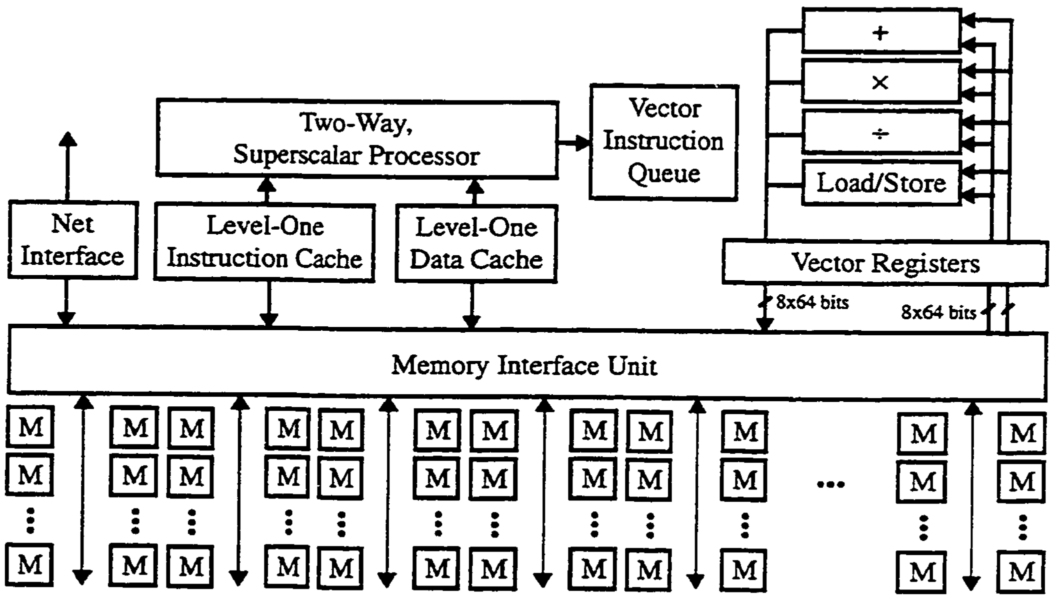


Figure 2.9 Organization of an IRAM Vector Processor

2.2.5 An Integrated Graphics Accelerator and Frame Buffer

This is a commercial chip [18] designed by Mosaid and Accelerix using some ideas from early CRAM research work [3]. The 33 GB/s, 13.4 Mb chip integrates parts of the graphics processor with DRAM. Other on-chip units include a PCI interface, VGA core and video input block, and custom-designed pixel output path (POP) logic. The 4K pixel-processing units (PPUs) are each pitch-matched to 4 DRAM columns. The chip is implemented in a 0.35 μm/0.5 μm blended DRAM and logic process. Because of a Non-Disclosure Agreement, the architecture of the chip will not be discussed.

2.2.6 A Memory-Based Parallel Processor for Vector Quantization

The Memory-Based Parallel Processor for Vector Quantization (FMPP-VQ64) [16], [17] is a memory-based parallel processor containing 64 PEs. It is designed to accelerate nearest neighbor search (NNS) in Vector Quantization (VQ). Almost all its features are specifically tuned and fixed for VQ processing. Each PE has sixteen 8-bit SRAM words that are used to store sixteen codebook vectors. The PE ALU can only compute the absolute distance between an input and a codebook vector. The ALU is 12-bits because the distance between an input vector and a codebook vector may only grow up to a 12-bit word. The FMPP-VQ64 was fabricated in a 0.7 μm CMOS process and has a die size of 55 mm^2. It operates at 25 MHz and dissipates 20 mW of power at 3.0 V. It can perform 53,000 nearest neighbor searches per second. Figure 2.10 shows the FMPP-V64 block diagram.

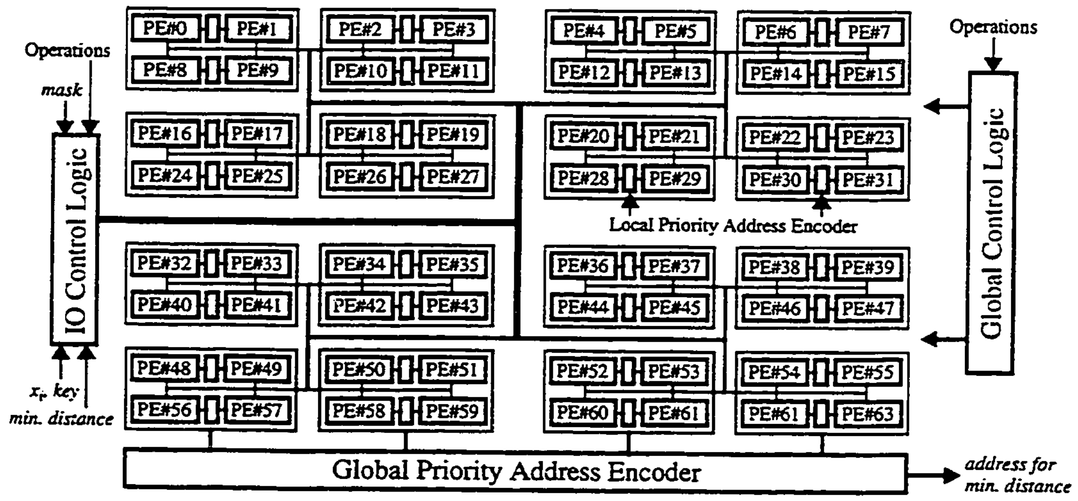

Operations
mask
IO Control Logic
x_i, key
min. distance
PE#0 PE#1 PE#2 PE#3 PE#4 PE#5 PE#6 PE#7
PE#8 PE#9 PE#10 PE#11 PE#12 PE#13 PE#14 PE#15
PE#16 PE#17 PE#18 PE#19 PE#20 PE#21 PE#22 PE#23
PE#24 PE#25 PE#26 PE#27 PE#28 PE#29 PE#30 PE#31
Local Priority Address Encoder
PE#32 PE#33 PE#34 PE#35 PE#36 PE#37 PE#38 PE#39
PE#40 PE#41 PE#42 PE#43 PE#44 PE#45 PE#46 PE#47
PE#48 PE#49 PE#50 PE#51 PE#52 PE#53 PE#54 PE#55
PE#56 PE#57 PE#58 PE#59 PE#60 PE#61 PE#61 PE#63
Global Priority Address Encoder
Operations
Global Control Logic
address for min. distance

Figure 2.10 Block Diagram of the FMPP-VQ64

2.3 Other Common SIMD Control Path Strategies

2.3.1 Hierarchical Controllers

Hierarchical controllers are used in SIMD machines to meet the high bandwidth of control required by the PE array. Examples of such machines include VASTOR [26], Associative String Processor (ASP) [27], and Massively Parallel Processor (MPP) [28]. Figure 2.11 show the controller hierarchy in VASTOR. The microcontroller executes sequences of microcode stored in an internal read-only memory. The starting address for a specific microcode sequence is loaded from the buffer memory. The buffer memory is divided into 16 separate task control blocks that are filled by the microprocessor. When ready, the microcontroller requests the address of the next control block by interrupting the microprocessor. The microprocessor further reduces the control bandwidth by translating high-level operations from the host computer into sequences of microcontroller tasks. It also handles storage management.

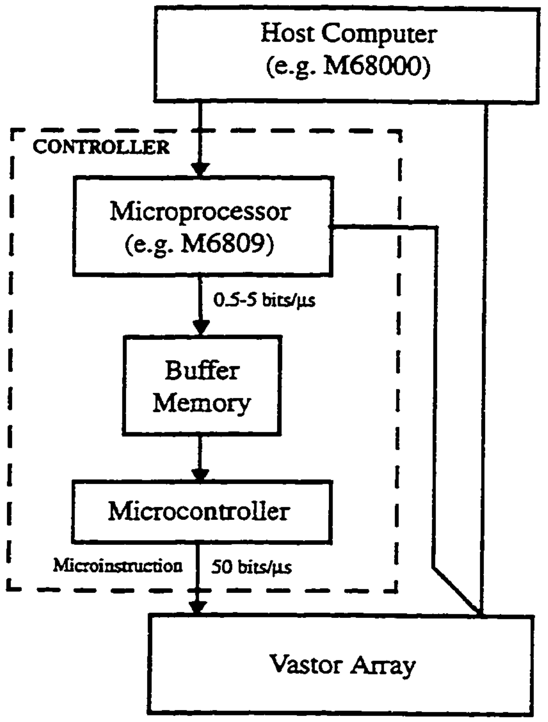


Figure 2.11 VASTOR Controller Hierarchy

2.3.2 Standard Microprogram Sequencers

Standard microsequencers, especially the Am2910 [30], have also been used as PE and memory controllers of SIMD machines. In this case, the microsequencer is the main functional block of a non-hierarchical control unit, with a few other components (such as an address processor) added to compliment its functions. Figure 2.12 shows the block diagram of the controller used in LUCAS Associate Array Processor [31]. An instruction is loaded from the master processor (in this case a Z80 microcomputer) into the instruction register (IR). This is then used by the Am2910 microsequencer to step through a 4K word microprogram memory. The address processor computes (increment, decrement, add constant, compare, etc.) the address to the bit-slice memory depending on the values in the parameter registers PR1-PR4.

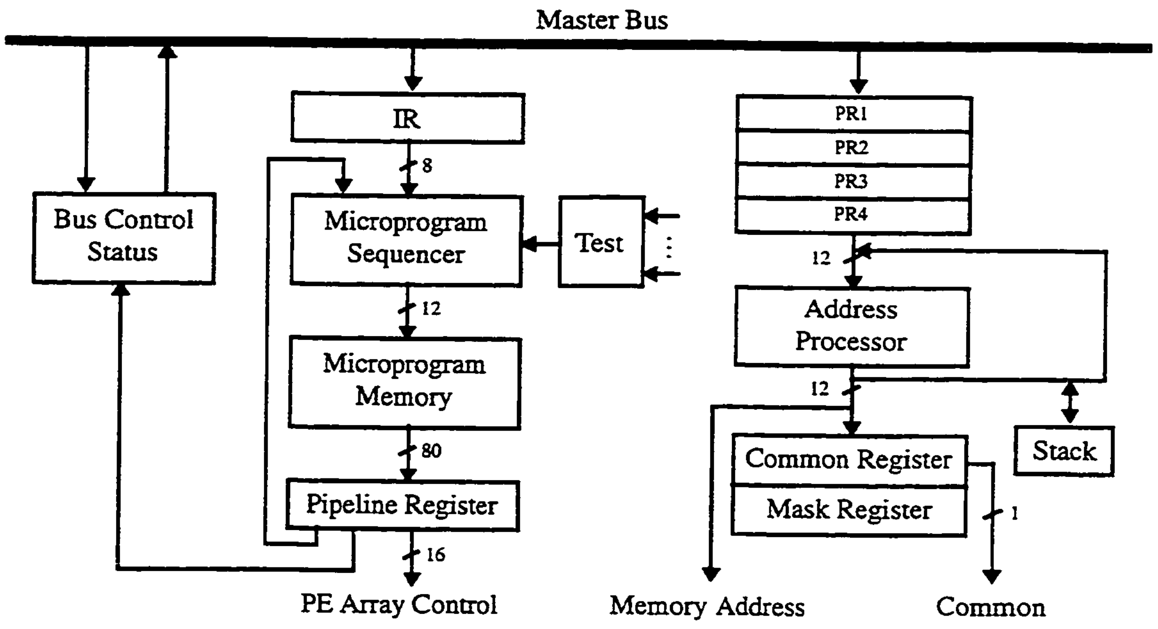


Figure 2.12 LUCAS Control Unit

2.3.3 High-Performance Processors

Some SIMD machines employ either standard or custom-built high-performance processors to control the PEs, as well as perform independent program execution. For example, the MasPar MP-1 array control unit (ACU) [32] is a 14 MIPS scalar processor with a RISC-like instruction set and a demand-paged instruction memory. The ACU fetches and decodes MP-1 instructions, computes addresses and scalar data values, issues control signals to the PE array, and monitors the status of the PE array. It is implemented with a microcoded engine and occupies one PCB.

2.4 Summary

In this chapter, the motivation behind logic-in-memory systems has been presented. A number of CRAM-related systems have been described to indicate the state of research in this area. Emphasis has been on system design issues, especially controller design strategies. This forms the basis of comparison for the controller and system design work presented in this thesis.

Most of the logic-in-memory systems described in this chapter are designed and implemented for specific applications, especially image processing and graphics. CRAM, on the other hand, is meant to be a general-purpose parallel-processing system targeted for a variety of applications and implemented on a number of standard platforms. The PC has been chosen as the major CRAM implementation platform because of its wide usage. This thesis describes the system design issues for a general-purpose logic-in-memory CRAM system. First, a brief overview of the architecture of CRAM is described in Chapter 3. Thereafter, the system design issues are presented. This includes the PE controller, the interface to the host computer, controller and system prototypes, programming and other software tools, applications, and performance analysis.

Chapter 3

Computational RAM

This chapter describes the architecture of Computational RAM (CRAM), and lists CRAM prototype chips implemented so far. Section 3.1 describes how logic is added to a standard RAM to form CRAM. It also gives details about the CRAM processing element (PE) array, including PE architecture, global and inter-PE communication networks, and how SIMD computations are performed on CRAM. Section 3.2 gives brief descriptions of prototype chips that have been implemented since the CRAM project began. One of the prototype chips, the C64p1k, has been described in more detail because it forms the basis of most of the system prototyping and performance analysis presented in this thesis.

3.1 Architecture

3.1.1 RAM with SIMD Processors

Computational RAM (CRAM) is a SIMD-memory hybrid architecture [19], [20], with single-bit processing elements (PEs) integrated at the sense amplifiers of a standard RAM. CRAM is designed to improve the speed of executing massively-parallel applications by utilizing the high bandwidth available at the sense amplifiers. Typically, the data width at the sense amplifiers is more than 1000 times the width of the RAM external bus (which is usually 1-, 8-, 16- or 32-bit wide). Several PE's can therefore have access to this data and operate on it without the need to read the data out of the RAM chip and transmit it over long high-capacitance buses to the processor. This improves performance and also reduces power consumption. Figure 3.1 shows the architecture of CRAM. The PE is laid out in the pitch of a few sense amplifiers (typically four or eight for DRAM, and one for SRAM).

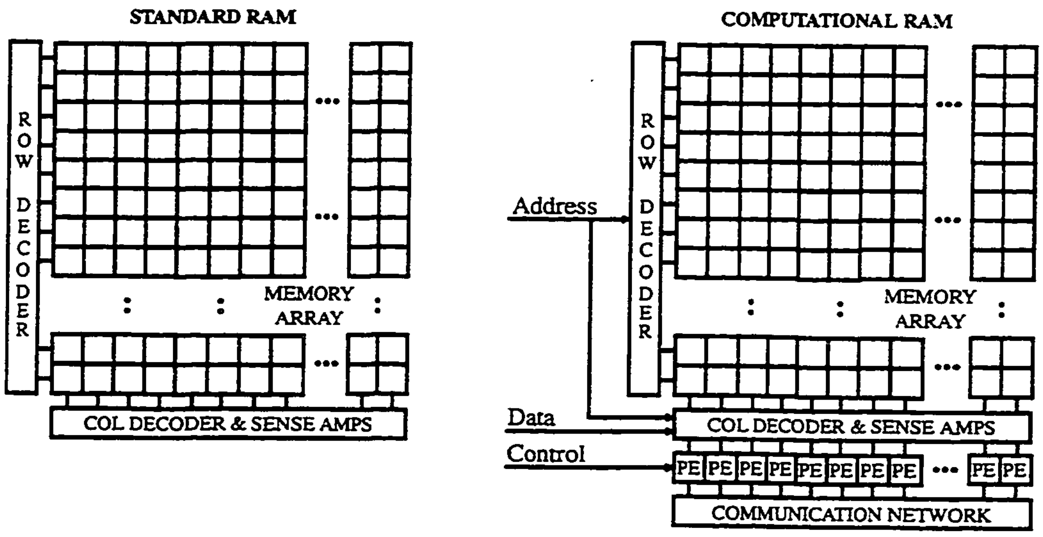


Figure 3.1 CRAM Architecture

3.1.2 Processing Element (PE)

A 1-bit PE architecture was chosen in order to achieve the highest performance per silicon area [21]. For applications with a high degree of parallelism, a trade-off can be made between higher processor complexity and a greater number of simple processors. In CRAM, two types of processing elements have been implemented.

- **Baseline PE (BPE):** The baseline PE shown in Figure 3.2 is the simpler of the two and consists basically of three registers (W, X, and Y) and an 8-to-1 multiplexer (mux). A fourth source of operands is the memory (M). The broadcast bus implements a global-OR (or bus-tie) of all the PEs outputs. Communication between adjacent PEs is through a shift-left/shift-right network. A PE instruction consists of an 8-bit multiplexer truth-table opcode (TTOP) and a 6-bit control opcode (COP). TTOP determines the actual operation performed by the PE depending on the contents of Y, X and M (the multiplexer select inputs). Three bits of COP (WY, WX, and WW) determine whether the output of the PE mux should be written to the PE registers Y, X and W registers respectively. The next two COP bits (SLX and SRY) control the PE shift operation by enabling the write of a PE mux output into the X or Y register of its left or right neighbor respectively. The sixth bit of COP enables the global-OR of the PE outputs. Register W controls the ability of a PE to write to its own memory. This is used in conditional code execution. TTOP and COP are multiplexed with data and address on the RAM data and address buses, respectively. Each PE operates on a single element of a vector, one bit at a time. The baseline PE is the architecture used in the bit-serial CRAM prototype chips C64p128, C64p1K, and C1Kp16K (Section 3.2). The BPE consists of 75 transistors in dynamic logic.

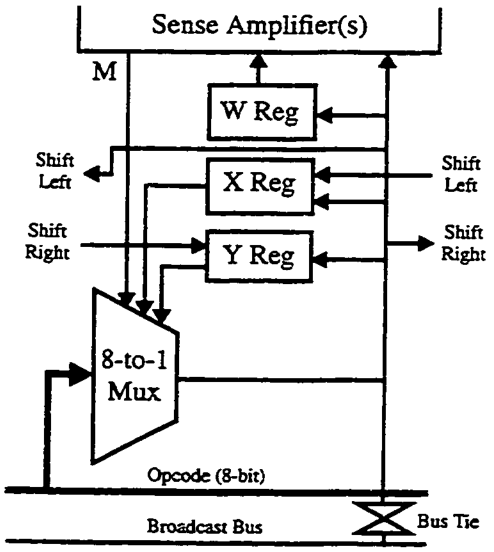


Figure 3.2 Baseline Processing Element

- **Extended Processing Element (XPE):** Shown in Figure 3.3, the extended PE is an enhancement of the baseline PE, with more registers to reduce the number of RAM cycles per computation. It also has a carry-chain to speed up addition, and has PE-chain segmentation features (such as the bus-tie segmentation register T, and the carry-in selection/segmentation registers S and B) to allow the grouping of adjacent PEs to work on multi-bit data. This data-parallel operation offers substantial improvement in the performance of the more complex arithmetic operations such as multiplication. The XPE can also be configured to function as a bit-serial baseline PE. This PE architecture has been used in the C512p512 CRAM chip and has 147 transistors.

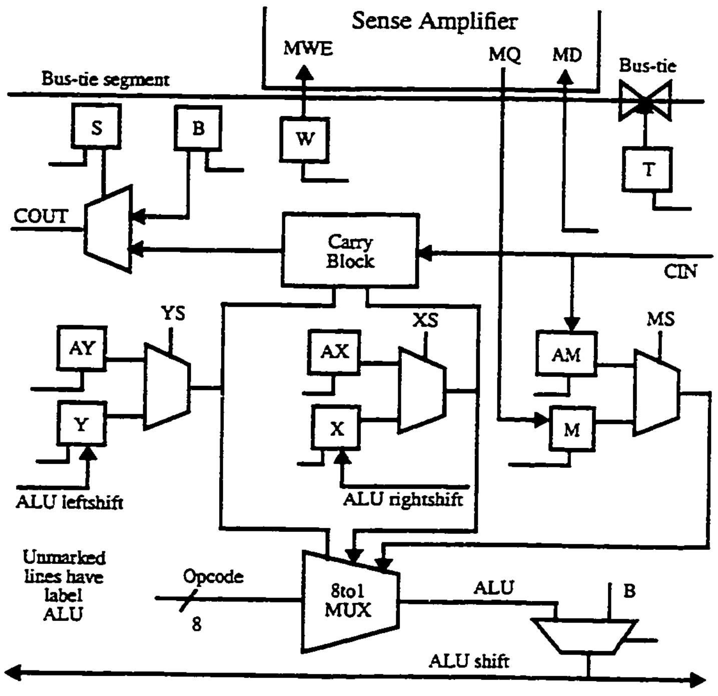
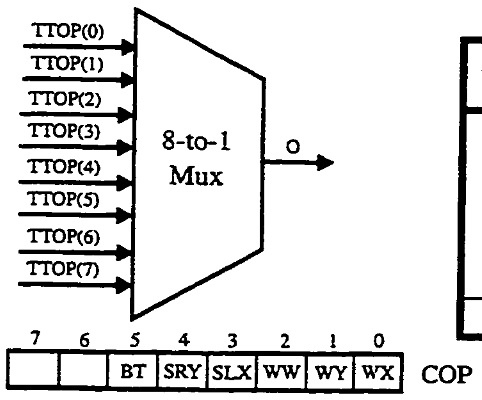


Figure 3.3 Extended Processing Element

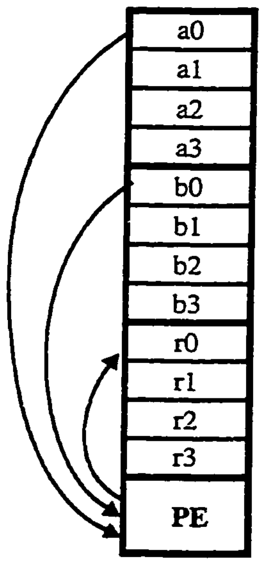
3.1.3 CRAM Computations

The PE ALU (multiplexer) computes any function of 3-bit variables Y, X, and M. TTOP defines the operation, with Y:X:M selecting which bit of TTOP should be the output of the ALU. Figure 3.4 shows the values that have to be set on TTOP in order for the multiplexer to compute the functions O=M, O=M, O=Y⊕X⊕M (addition) and O=YX+YM+XM (carry). The destination register of the operation is specified by setting the corresponding bit of COP to 1. To illustrate how each PE does the computation, consider 4-bit addition. Figure 3.5 shows how the data (operands) for the addition are arranged in each PE memory. The addition is then performed one bit at a time using the algorithm shown in the figure. Bit 0 (a_0, b_0, r_0) is the least significant bit. The looping and the setting of the opcodes are performed by a controller external to the CRAM chip.



Source Regs YXM	Selected TTOP Bit	O=M	O=$\overline{M}$	O=Y⊕X⊕M (Addition)	O=YX+YM+XM (Carry)
000	TTOP(0)	0	1	0	0
001	TTOP(1)	1	0	1	0
010	TTOP(2)	0	1	1	0
011	TTOP(3)	1	0	0	1
100	TTOP(4)	0	1	1	0
101	TTOP(5)	1	0	0	1
110	TTOP(6)	0	1	0	1
111	TTOP(7)	1	0	1	1
YXM	TTOP	AA	55	96	E8

Figure 3.4 Setting PE Opcodes (TTOP and COP)



r[3:0] = a[3:0] + b[3:0]

```
Y ⇐ 0;// clear the carry in Y (TTOP=00, COP=02)
FOR i=0 TO 3
   M ⇐ a[i];// read bit a_i from memory
   X ⇐ M;// load a_i into X (TTOP=AA, COP=01)
   M ⇐ b[i];// read bit b_i from memory
   Y⊕X⊕M// calculate the sum (TTOP=96, COP=00)
   r[i] ⇐ ALU output;// write ALU output to r_i
   Y ⇐ YX+YM+XM;// calculate carry into Y (TTOP=E8, COP=02)
END FOR
```

Figure 3.5 PE 4-bit Addition

3.2 Prototypes

This section describes the CRAM prototype chips that have been implemented since the CRAM project began. These are very brief descriptions. Detailed descriptions can be found in the referenced theses of the students who implemented them. The prototype chip C64p1K has been described in slightly more detail because it is the chip that has been used in the CRAM prototype system designed and implemented in this thesis.

3.2.1 A 64-PE, 128 bits/PE CRAM (C64p128)

This CRAM was designed and implemented by Duncan Elliott [4]. It is an 8 Kbit CRAM, with 64 baseline PEs and 128 SRAM memory bits per PE. Each PE fits in the pitch of one sense amplifier. The PEs occupy only 9% of the total area, with the 6-transistor SRAM cells dominating the chip area. The PE registers are designed as 5-transistor static CMOS latches. A PE cycle has three phases: precharge, ALU (mux) operation, and write (registers and memory). The C64p128 was implemented in a 1.2 μm CMOS process and has a read-operate-write cycle time of 114 ns. With the exception of an error in one of the 64 column decoders, the chips are functional.

3.2.2 A 64-PE, 1 Kbits/PE CRAM (C64p1K)

The C64p1K was implemented by Christian Cojocaru [22]. It is basically an enhancement of the C64p128, with more memory (1 Kbits) per PE and implemented in a faster BNR/NT's 0.8 μm BiCMOS technology. BNR synchronous SRAM modules were used as the PEs memory in order to speed up the implementation. These single-port SRAM modules use a standard 6-transistor cell and can operate up to 180 MHz. The C64p1K has an 8-bit data bus and a 13-bit address bus.

C64p1K memory accesses are clocked by the memory clock (MCK) signal. On the positive edge of MCK, the RW and RAM signals (which shows whether the access is read/write, or is external or between the PE and its memory) are sampled. The memory address, and the write data, are also latched. The positive edge then triggers a self-timing circuitry that generates all the timing signals necessary to complete the memory cycle.

The PE operate cycle is controlled by six signals, namely:

1. ALUPRE : Precharges the PE dynamic logic prior to evaluation. It is active LOW.
2. TTOPCK : The active HIGH evaluation clock.
3. COPCK : When HIGH, the results of the evaluation are written to the PE registers.
4. BTENCK : When HIGH, it enables the bus-tie (global-OR) of all the PE outputs.
5. TTOPLD : Its positive edge latches TTOP from the data bus into internal registers.
6. COPLD : Its positive edge latches COP from the address bus into internal registers.

These signals can either be generated externally by a CRAM controller, or they can be generated internally by the CRAM chip when the external control pin (XCTRL) is set to zero. These two timing schemes are described below.

- **External Timing:** In this approach, all the six PE timing signals must be generated by the CRAM controller. While this slightly reduces the area of the CRAM, it greatly increases the complexity of the controller, especially since a high clock rate must be used to generate the intermediate phases of the PE cycle. Figure 3.6 shows the timing of these signals. The signals ALUPRE, TTOPCK, and COPCK must be mutually non-overlapping. Also, BTENCK can not overlap ALUPRE. Unfortunately, the external timing scheme is not functional on the C64p1K chip due to a suspected layout error.

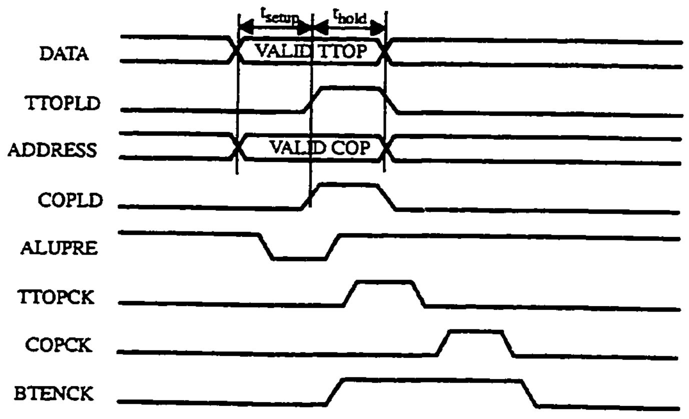
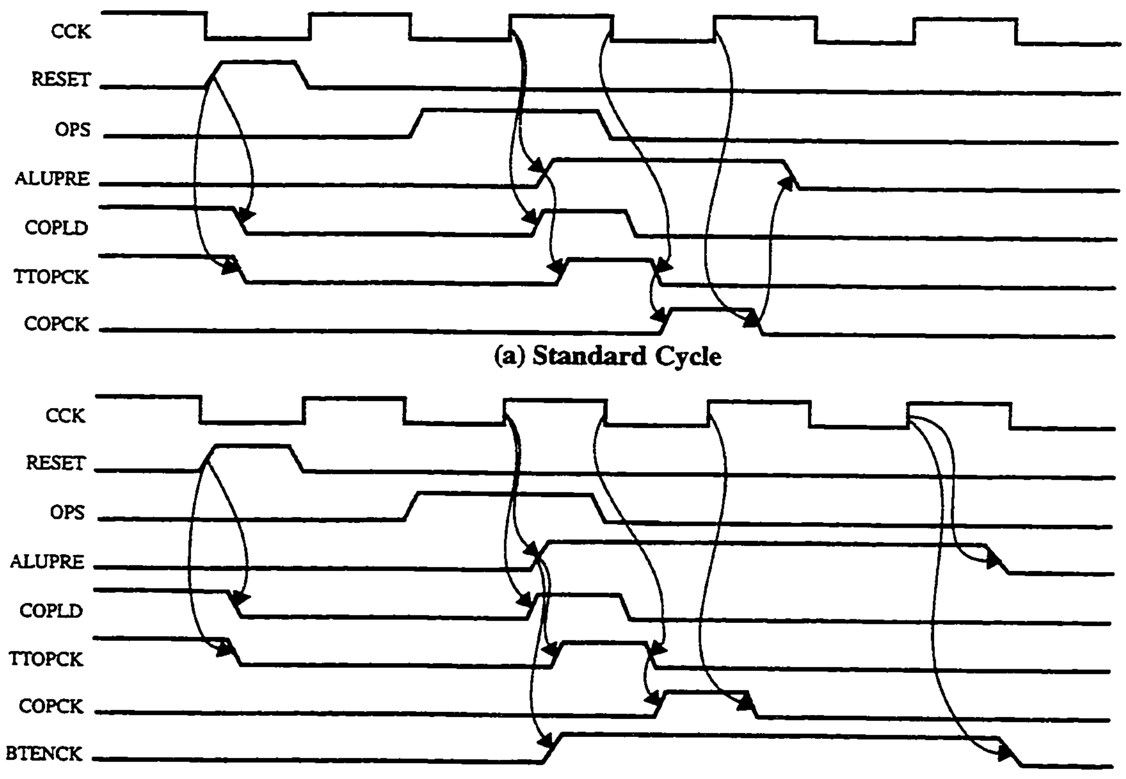


Figure 3.6 PE Cycle External Timing

- **Internal Timing:** For this scheme, the PE timing signals are generated on chip using a finite state machine (FSM) controlled by two external signals: OPS (Operate Strobe) and CCK (CRAM Clock). A PE cycle involving a bus-tie operation is one CCK period longer than a standard cycle to allow for worst case bus-tie propagation delays. Figure 3.7 shows the timing for the two PE cycles. OPS positive edge signals the beginning of a PE cycle. Thereafter, transitions in the timing signals are controlled by both edges of CCK, which was designed to be a free-running clock. RESET is only asserted at power-up to initialize the FSM. As attractive as it is for the design of the CRAM controller, the internal timing scheme implemented on the C64p1K chip has two bugs. The first one, not evident at the time the chip was designed, concerns the timing of operate-write cycles. Since ALUPRE was designed to be normally-active (instead of the more standard and more logical normally-inactive scheme), and since the second positive edge of CCK re-activates ALUPRE after being deactivated by CCK first positive edge, the results of a PE operation can not be correctly written back to memory since the PE output will already have been precharged again when the memory-write begins. This bug was discovered during the design of the controller. To correct this for an operate-write cycles, the controller no longer issues a free-running CCK. But rather



(a) Standard Cycle

(b) Bus-tie Cycle

Figure 3.7 PE Cycle Internal Timing

CCK is now a second control signal that is delayed (issued after MCK) if an internal memory cycle follows a PE-operate cycle. This correction has added to the complexity of the controller. The second bug, reported in the chip documentation [22], concerns the timing of the bus-tie operation. Even though the bus-tie is extended by one CCK period, the write-register (COPCK) signal is still asserted at exactly the same point as in a standard PE cycle. This means that the extra length of the bus-tie cycle is useless if the results of the bus-tie operation are to be written to a PE register (which is usually the case) instead of the PE memory. In this case, the extra CCK cycle simply complicates the timing sequencing, especially now that CCK is not free-running.

3.2.3 A 512-PE, 480 bits/PE CRAM (C512p480)

The C512p480 [22] is a bit-parallel/bit-serial CRAM based on the extended PE shown in Figure 3.3. Though initially designed to have 512 bits of memory per PE, it was implemented with 512 x 480 bits due to silicon area constraints. Like the C64p1K, the C512p480 was fabricated in the BNR/NT 0.8 μm BiCMOS technology. The C512p480 bit-parallel CRAM architecture simplifies system design and programming since the data doesn't have to be transposed from its bit-parallel format in the host computer. Also, the increased number of registers in the XPE reduces the number of memory accesses per computation. While organizing PEs in groups to work on multi-bit data in bit-parallel format speeds up the rate of computation for a single data item, it does not necessarily increase the overall performance since the degree of parallelism is reduced. Also, some algorithms, especially those requiring searches or comparisons on vector items, are difficult, if not slower, to implement on a bit-parallel architecture. Unfortunately, the C512p480 chips are not fully functional; only the RAM part is functional. It is suspected that there is a flaw in the PE timing circuitry or in the layout of the PEs themselves.

3.2.4 A 1024-PE, 16 Kbits/PE CRAM (C1Kp16K)

The design and implementation of the C1Kp16K CRAM chip is still in progress [9]. This is the first CRAM prototype that is based on DRAM. It integrates 1024 baseline PEs in a 16 Mb IBM DRAM [23]. Each PE in a DRAM octant is connected to 32 sense amplifiers and has 16 Kbits of local memory. A vacant test mode available on the original DRAM is used for CRAM operation. Therefore, no extra control pins are required for CRAM, thus maintaining JEDEC compatibility [24].

3.3 Summary

This chapter has described the architectural details of Computational RAM (CRAM). CRAM integrates 1-bit processing element (PEs) at the sense amplifiers of a standard RAM. These PEs work in parallel in SIMD style, and improve the performance of massively-parallel applications by utilizing the inherent parallelism and high data bandwidth inside the memory chip. The PEs are 1-bit to reduce their area so that a number of them can be implemented in a single memory chip. This increases the degree of computational parallelism and performance.

Prior to this work, the CRAM project had concentrated on the design and implementation of individual CRAM prototype chips described in Section 3.2. The next obvious step was therefore to explore system design issues, and that is the focus of this thesis. Specific issues studied include the design of the PE controller, interface to the host computer, CRAM system software tools, prototyping, and system performance analysis. The next chapter describes the design of the controller and the interface to the host computer.

Chapter 4

CRAM Controller

This chapter describes the architectural details of the CRAM controller and how they affect the performance of a CRAM system. The requirements and characteristics of the controller are laid out in Section 4.1. Section 4.2 describes the interface of the controller to the host bus. Four host buses (PCI, VME, EISA, ISA) are studied. Section 4.3 describes the instruction queue unit and how it affects the execution time of CRAM instructions issued from the host computer. After this, the execution unit, which comprises of a microprogram sequencer and an address unit, is described. The use of read/write buffers to improve the performance of data transfers and operate-immediate instructions is discussed in Section 4.5. The last two sections describe the mode of access and memory-map of all user-accessible units on the CRAM controller. All performance results reported in this chapter are based on simulations of CRAM systems hosted on a 133 MHz Pentium PC.

4.1 Introduction

The main use of the CRAM controller is to allow CRAM application programs to be run from a front-end host computer. The controller acts as a PE array controller as well as an interface to the host bus. There are two reasons why the PEs have to be controlled from a dedicated controller. First, in order to minimize their area, CRAM PEs do not have control structures. Instead, they are designed such that control, data, and address information is broadcast to them in an SIMD style from an external controller. Second, since the PEs are bit-serial, a single standard operation such as addition takes several PE instructions to execute. Rather than issue such low-level (micro) instructions from the host processor, only high-level (macro) instructions are passed from the host to the controller, which in turn generates the PE microinstructions. This frees the system bus from congestion with the high traffic microinstructions, and also increases CRAM system performance since the host cannot issue PE microinstructions fast enough to correspond to the rate at which the PEs execute them.

As described in Chapter 2, a number of approaches have been used to implement controllers for SIMD machines in general, and logic-in memory systems in particular. However, the architecture and implementation of the CRAM controller [25] has been based mainly on the following requirements:

- **Simple and Minimal Hardware:** Typically, SIMD machines use high performance microprocessors as their controllers. For example, VASTOR uses an M6809 microprocessor and a microcontroller [26]. The Massively Parallel Processor (MPP) and the MasPar MP-1 use custom processors as their controllers [28], [29]. Even some logic-in-memory systems, such as the IMAP [11], use full scale processors as controllers. In CRAM, the need for very minimal hardware arises for three reasons. The first one is to reduce system cost. Secondly, unlike all related systems, CRAM is designed to be a future replacement of standard RAM as computer main memory. For this to be a realizable idea, both the CRAM and its controller must add minimal hardware to the existing memory system (i.e. to RAM and RAM controller). The third reason for a simple and small controller is that one of the future efforts suggested in this thesis is a CRAM system in which both the CRAM and its controller are implemented on the same chip (see Chapter 8). Therefore, to maintain the small additional area of CRAM over standard RAM, the controller must use as few gates as possible. Also, since a combined RAM/logic system will generally mean logic being implemented in a slow RAM process [9], [18], a simple controller would yield higher speed.

- **High PE Utilization:** One of the underlying requirement of an SIMD PE controller is to make sure that the PEs are always kept busy. In other words, instructions to the PEs must be issued fast enough to match the rate at which the PEs execute them. To achieve this while maintaining a simple and minimal architecture, the CRAM controller has been equipped with performance-enhancement features such as a FIFO-based instruction queue unit, read/write buffers, and a new constant unit. Its sequencing features have also been trimmed to only those required to control the rather simple single-bit CRAM PEs.
- **General-Purpose Architecture:** In most logic-in-memory systems, the PE array and the controller are designed for specific applications. For example, the PE and the controller for IMAP[10], PIPRAM[12], and MIT-PP[14], are designed specifically for 8-bit pixel processing. FMPP-VQ[16] can only run Vector Quantization, while the logic-in-memory graphics accelerator and frame buffer [18] has a controller that is hard-wired for graphics operations only. On the other hand, the CRAM controller had to be designed to enhance the use of CRAM as a general-purpose parallel-processing system. For this reason, the controller supports operands of different word-lengths, and it has a microprogrammed architecture to allow different applications to be optimized in software (using the microcode development tools). Other architectural features, such as the organization of the data buffers and the isolation of the external bus interface unit from the controller core logic, allow the CRAM system to be easily implemented on different platforms (three system bus interface units have been designed).
- **Easy Programming:** No special programming language or instruction syntax is required for the controller. In fact, since the controller does not execute the standard processor instructions, but rather just sequences the PE microinstructions, the programmer need only worry about writing software for the variables in the CRAM mem-

ory. A CRAM C++ compiler (see Section 6.2) has been designed for this. The few instructions that need to be executed on the controller are limited to loading, incrementing and decrementing its registers. These are automatically handled by the CRAM C++ compiler, unless the programmer chooses to program in CRAM assembly code.

- **Constrained PCB Size:** Most SIMD machines occupy many PCBs housed in very large cabinets [13], [6], [73]. However, for a system to be implemented as an add-in card in a PC environment, it must be on a single standard-size board. Therefore, in order to fit the controller on the same small board as the CRAM chips it controls, it must use as few external chips as possible. Features of SIMD controllers that are commonly implemented with external chips include the microprogram control store, data-transposing hardware (sometimes referred to as data format converters), and program and data memory. The control store of the CRAM controller has been made small enough to be easily implemented on-chip. This has been achieved by storing microinstructions of basic operations only, and then using an innovative method of grouping microroutines that takes advantage of the unique architecture of the CRAM PEs and its multiplexed TTOP/Data bus to reduce the number of required microinstructions. Data-transposition is not done on the controller. Instead, for accesses involving a small number of elements, the conversion of data from the CRAM (vertical) format to the host format is done on the host. For large accesses, an array-based data-transposition, that takes advantage of the high degree of parallelism in the PE array, has been proposed (Section 6.3). No program or data memory is provided on the controller. All programs and data are store on the host computer.

Figure 4.1 shows the architecture of a CRAM controller that meets the above requirements. It is designed mainly as a PE microinstruction sequencing machine. All other sequential operations in an application program are executed on the host computer. The rest of the chapter describes the architectural features of the controller and how they affect the overall performance and design of a CRAM system.

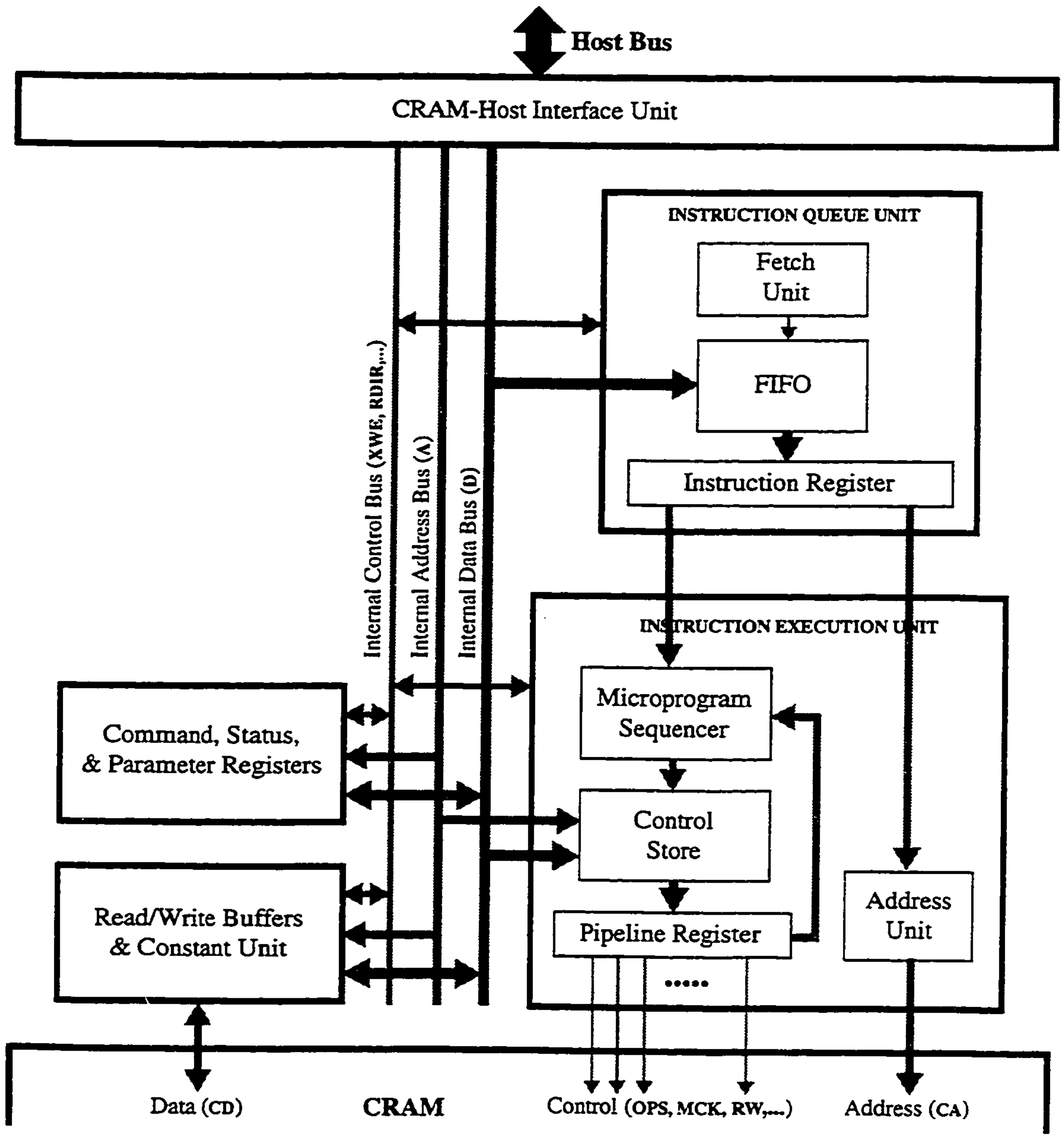


Figure 4.1 CRAM Controller Architecture

4.2 CRAM-Host Interface Unit

The CRAM-Host Interface unit connects the CRAM controller to the host processor. This allows the transfer of CRAM instructions and data from the host processor environment to the CRAM system, and the transfer of CRAM status and data to the host.

In this work, the major emphasis has been on the study of the effect of the CRAM-host interface on the performance, area, and design of a CRAM system, rather than on the implementation of a specific interface unit. For this reason, a number of host interfaces have been studied to allow a wide range of experimentation. For our main target environment, which is the Personal Computer (PC), we selected three buses: the PCI bus, the ISA bus, and the EISA bus. The PCI bus was chosen because of its high speed [34], and it is also becoming a standard in most PCs [35]. Its major disadvantage is that it has a very complicated bus protocol and requires significant configuration resources. In this work, we use the 33 MHz, 32-bit PCI target (slave) bus protocol compliant with the PCI Specification Revision 2.1 [36]. The ISA bus is the simplest of the PC expansion buses. While its data rate is low (16-bit at 8 MHz), unlike the PCI bus, its protocol [38] requires no configuration registers, it does not support burst cycles, and it has a very simple and slow timing scheme. The only configuration resources are jumpers for setting the address of the ISA card. The EISA bus [38], which is an extension of the ISA bus, offers a compromise between the ISA bus and the PCI bus. While the EISA bus runs at the same clock speed (8 MHz) as the ISA bus, the former supports burst transfers and has a 32-bit data bus allowing data transfers on the EISA to reach a maximum rate of 33M bytes/s [64]. Also, the EISA bus has a 32-bit address bus and has more advanced configuration resources than the ISA bus. This makes EISA devices easier to use in a plug-and-play system (automatic assignment of 32-bit address space) than ISA devices. It is not as widely used as the other two buses, however, the CRAM-EISA interface offers a speed improvement over the ISA bus while retaining the simplicity of the ISA bus protocol. Also, the EISA bus is still offered in many systems that have decided to drop the ISA bus as the secondary bus to the PCI bus. Finally, to allow CRAM to interface to systems other than the PC, a VME bus [39] interface has also been studied. VME is an asynchronous bus with data transfers of up to 40M bytes/s for 32-bit buses and 80M bytes/s for 64-bit buses.

As a proof-of-concept for the CRAM controller, and to enable a more realistic study of the effect on performance, area, and design effort, two interface units based on the PCI and ISA buses have been designed and implemented in two separate Xilinx FPGA controller prototypes. While the EISA and VME interfaces have not been implemented, their bus data transfer parameters have been used in the study of the effect of the CRAM-host interface on the performance of a CRAM system. The following sections describe the PCI and ISA interface units.

4.2.1 CRAM-ISA Interface Unit

Because of the relatively small amount of hardware and low design effort required to implement the ISA bus protocol, it was decided to use the ISA interface in the first controller prototype. This presented an ideal starting point for a Xilinx FPGA controller prototype because of the area and speed constraints in programmable devices. Figure 4.2 shows the block diagram of the CRAM-ISA interface unit. The address decoder compares the address on the LA bus with the value set by the CRAM address jumpers. If there is a HIT, the data transfer is executed by the ISA bus control logic. The CRAM interface unit, like the one in the CRAM-PCI interface unit (Figure 4.3), generates read/write control to the CRAM controller units and passes their status to the ISA bus control logic.

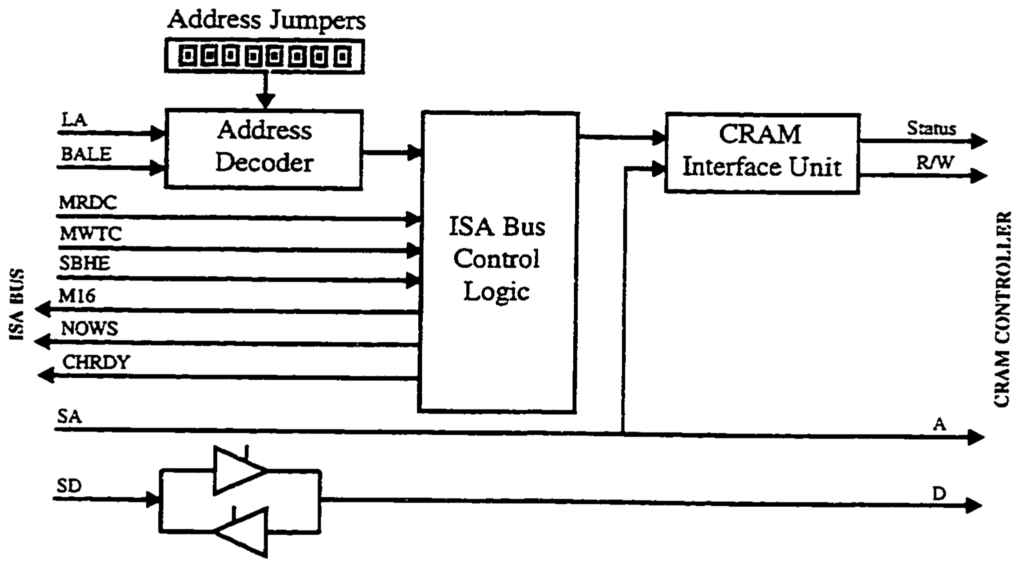


Figure 4.2 CRAM-ISA Interface Unit

4.2.2 CRAM-PCI Interface Unit

The PCI interface unit is shown in Figure 4.3. The address comparator and command decoder decode the PCI cycle and passes it to the PCI Target FSM/Control logic, where it is executed. The parity checker/generator generates even parity on the 32-bit Address/Data bus and the 4-bit Command/Byte Enable bus during PCI read cycles. This is implemented on the CRAM controller because it is mandatory for all PCI devices. Parity checking on the AD and CBE buses during write cycles is not mandatory, but it has also been implemented on the controller to guard against errors especially when loading microinstructions and macroinstructions from the host. This was also influenced by the minimal hardware required to implement parity checking. To minimize the area of the CRAM controller, only the PCI-required configuration registers have been implemented. These registers, together with other implementation details of the CRAM-PCI interface unit, are described in Appendix A.

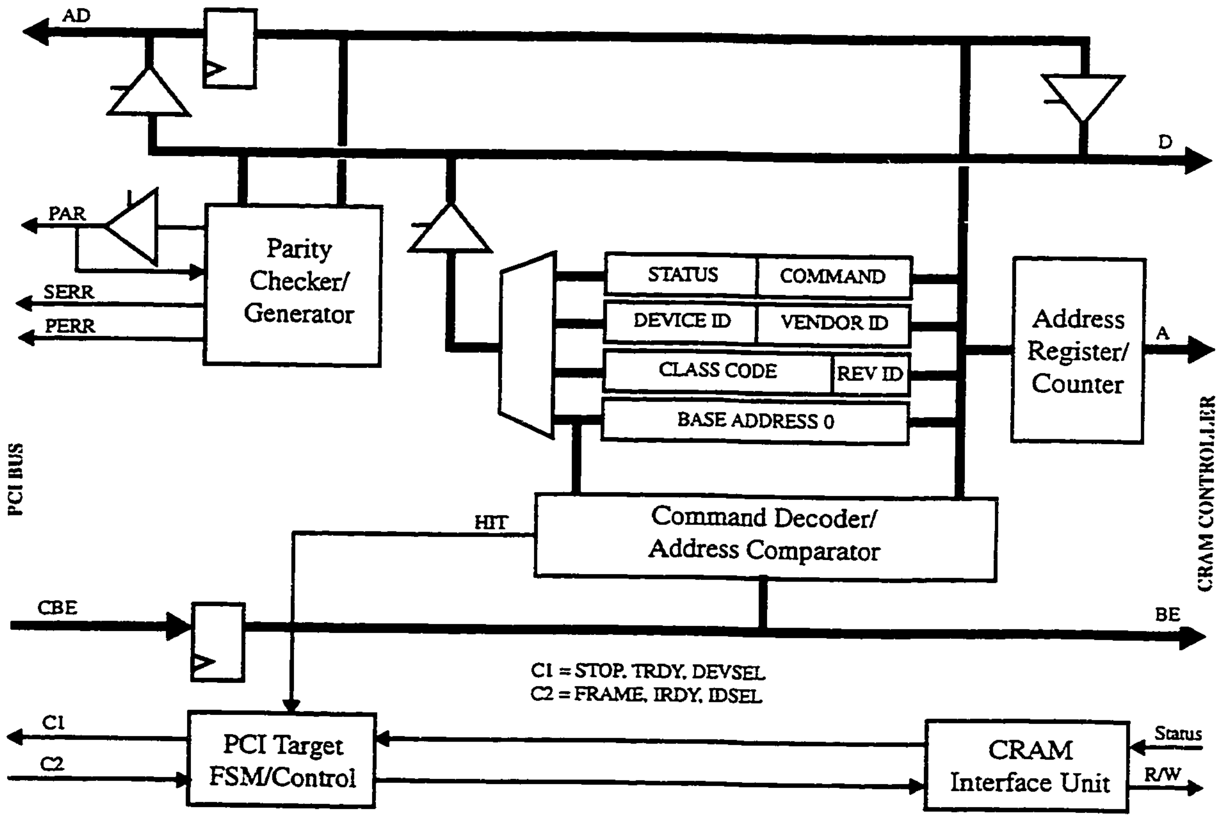


Figure 4.3 CRAM-PCI Interface Unit

4.3 Instruction Queue Unit

All CRAM instructions from the host processor are first loaded into the Instruction Queue Unit (IQU). Once in the IQU, instructions are executed by the CRAM controller completely independent of the host processor. The IQU is shown in Figure 4.4, and consists of an instruction FIFO, an instruction fetch unit, and the instruction register (IR). The FIFO removes the need to synchronize the external (host) bus transfers of CRAM instructions to the internal execution of these instructions. It also allows the host processor to transfer the instructions in advance without waiting for the previous instruction to finish executing, thus improving PE utilization. Otherwise, if instructions were loaded from the external bus directly into the instruction register, there would be big breaks in the instruction flow (several idle clock cycles between instructions) due to bus transaction delays. Also, for buses such as the PCI, EISA and VME, the instruction FIFO allows the host processor to transfer instructions at increased speed using burst cycles.

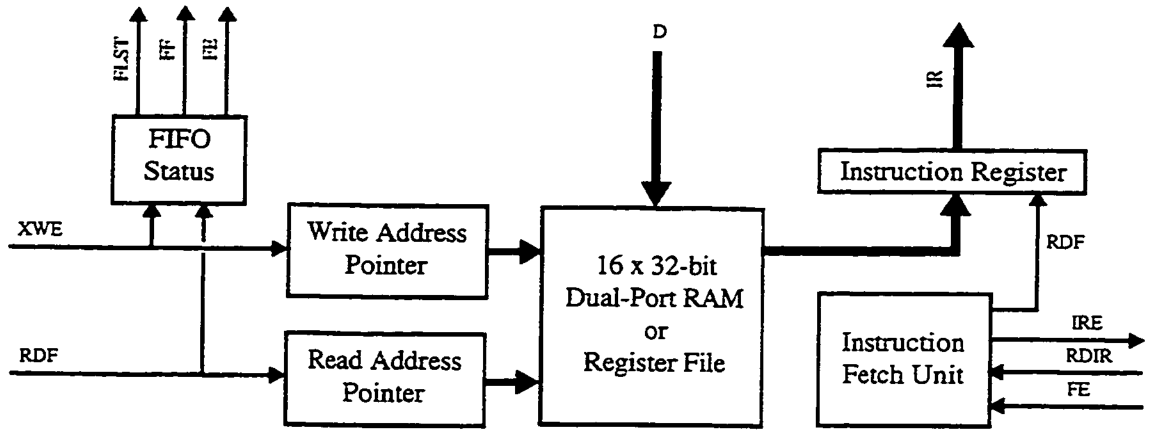


Figure 4.4 Instruction Queue Unit

4.3.1 Effect of IQU on Performance of PE Array Controller

The total time, T_{exe}, to execute an instruction on a CRAM-like three-layer instruction flow (Host Computer ⇒ Controller ⇒ PE Array) is given by

$$T_{exe} = T_{init} + T_{load} + T_{flow} + T_{pe} \tag{4.1}$$

where T_{init} is the host overhead of initializing the instruction and setting up its transfer to the CRAM controller, T_{load} is the time to transfer the instruction on the host bus, T_{flow} is the time for the instruction to flow through the instruction path on the controller, and T_{pe} is the time to execute the instruction on the PE array. If the instruction has n microinstructions, and the cycle times for the CRAM system and the host bus are T_c and T_{bus}, respectively, then

$$T_{pe} = nT_c \tag{4.2}$$

$$T_{flow} = 2T_c \tag{4.3}$$

$$T_{load} = 2T_{bus} \tag{4.4}$$

Note that the instruction path on the CRAM controller is a three-stage pipeline (FIFO $\Rightarrow$ IR $\Rightarrow$ Pipeline Register) and hence an instruction takes two clock cycles to flow from the FIFO to the pipeline register. If the components of Equation 4.1 were always executed sequentially, then the total execution time for a sequence of N instructions would be

$$T_{exe} = N[T_{init} + 2T_{bus} + (n+2)T_c] \tag{4.5}$$

However, the instruction queue unit allows that some of these components are executed in parallel with each other. This improves the performance of the CRAM controller when compared to controllers that only have IR in the instruction path, such as the MIT Pixel Processor [14] and the LUCAS Associate Array Processor [31]. IQU affects T_{exe} differently for the cases $(T_{pe} + T_{flow}) \leq (T_{init} + T_{load})$ and $(T_{pe} + T_{flow}) > (T_{init} + T_{load})$.

For the case where $(T_{pe} + T_{flow}) \leq (T_{init} + T_{load})$, the performance of the CRAM controller is better than that of an IR-only controller because of two reasons. First, for an IR-only controller, only one instruction can be transferred at a time (after IR is empty). Therefore to transfer N instructions, the host computer has to set up the transfer N times. This increases T_{init} when compared to the CRAM controller where a transfer of more than one instruction is possible. Also, for an IR-only controller, the total instruction load time, T_{load}, is always equal to $2NT_{bus}$. On the other hand, for host buses that support burst transfers, the transfer of instructions from the host to the CRAM controller may be executed in a shorter time, $(N + 1)T_{init} \leq T_{load} \leq 2NT_{bus}$, depending on how many instructions are transferred during each burst. The second reason for the improved performance is due to the way data transfers to an external bus are executed on the host computer. Once the host processor has set up a data transfer, the unit responsible for handling such transfers (an external bus interface unit, a load/store unit, or a dedicated I/O processor) can execute the data transfer in parallel with the execution of instructions in the other units of the processor. Since an instruction can be loaded onto the CRAM controller while the previous instruction is still executing, the initialization and setting up of the instruction on the host can be done in parallel (fully or partially) with either the execution of PE microinstructions (T_{pe}), the flow of an instruction in the controller (T_{flow}), or the transfer of an instruction on the host bus (T_{load}). This reduces T_{init} even further.

For the case where $(T_{pe} + T_{flow}) > (T_{init} + T_{load})$, a new instruction is loaded into the FIFO during the execution time of the previous instruction. This means that T_{init}, T_{load}, and T_{flow}, do not contribute to the total execution time of the instruction, and T_{exe} becomes equal to T_{pe}. This not only reduces the execution time to its minimum possible value, but also makes the execution time of CRAM instructions independent of the transfer characteristics of the host bus. This is very important for CRAM as a general-purpose system because it means that CRAM systems can be implemented for a variety of platforms, including slow host systems such as ISA-based computers and embedded systems that use slow microcontrollers.

Another parameter for measuring the performance of an SIMD controller is PE utilization. This is defined as the percentage of the total execution time for which the PEs are busy, i.e.

$$PE_{utilization} = \frac{T_{pe}}{T_{init} + T_{load} + T_{flow} + T_{pe}} \times 100 \qquad (4.6)$$

For all cases, a controller approaches an ideal controller (100% PE utilization) if $T_{pe} \gg (T_{init} + T_{load} + T_{flow})$. However, for the CRAM controller, this happens at a smaller value of T_{pe} because of the parallelism possible with IQU. In fact, at the point where $T_{pe} = (T_{init} + T_{load} + T_{flow})$, the utilization is 100% as opposed to 50%.

One way to characterize the performance of an array controller is to execute a sequence of instructions of varying number of microinstructions per instruction [14]. Figure 4.5 shows the simulation results based on a 50 ns CRAM system interfaced through the PCI bus to a 133 MHz Pentium PC. As noted before, not only is the performance

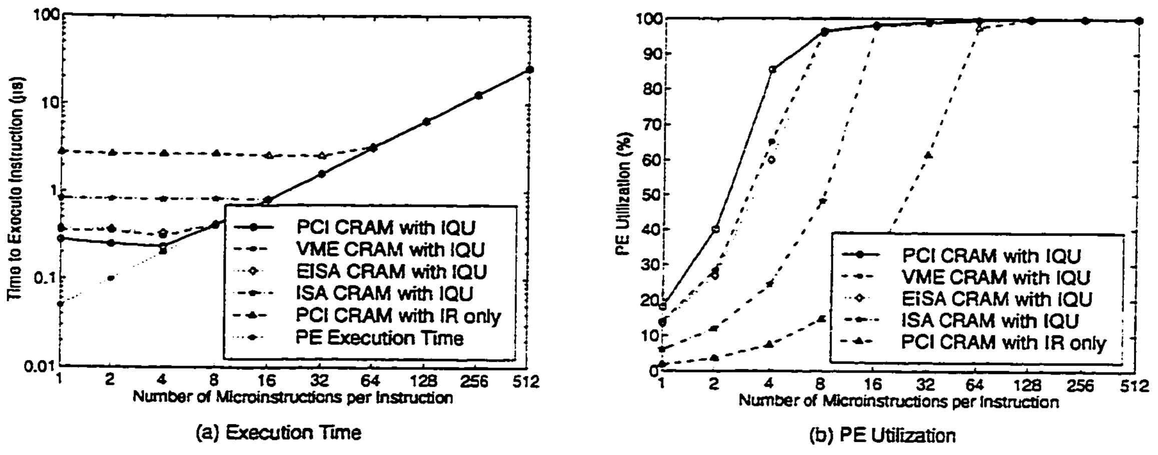


Figure 4.5 Effect of IQU on Controller Performance

better than for a controller with IR only, but it also becomes independent of the host bus at a smaller number of microinstructions per instruction. Another important thing to note from Figure 4.5 is the controller performance for short sequences, i.e. instructions with a small number of (less than 30) microinstructions. These instructions generally exhibit lower PE utilization because T_{pe} is much less than $(T_{init} + T_{load} + T_{flow})$. However, the IQU allows the performance of the CRAM controller to be as high as 10 times the performance of an IR-only controller. For application specific systems, especially pixel-processing, these instructions are dismissed as not typical because operands are generally 8-bit wide or bigger. But for a general-purpose system such as CRAM, short-sequence instructions are common, and are especially used in operand extension, updating of controller registers, and optimization of operations. They may also constitute the main operations in databases which have records with less than 8-bit precision. For example, a database of the ages of children at a nursery school, or the number of courses taken by a student per semester, is efficiently stored as 4-bit numbers on CRAM because these numbers are always less than 16, and 4-bit numbers use half the memory of, and execute twice as fast as 8-bit numbers. Therefore, short-sequence instructions may constitute a significant percentage of the total number of instructions if the application has either low-precision (or small-size) operands, or if the number of operations in the program is very small compared to register-initialization instructions. Table 4.1 gives examples of applications with a high percentage of short-sequence instructions. These applications are described in detail in Chapter 7. Notice that for 4-bit database records, almost all instructions in the search operations have less than 30 microinstructions. Therefore, the performance improvement for short-sequence instructions in the CRAM controller is of vital importance for a general-purpose CRAM system.

Application	Total Number of Instructions	Number of Instructions with less than 30 Microinstructions	Percentage of Instructions with less than 30 Microinstructions (%)
Low-level image processing:			
Brightness adjustment	9	8	89
Spatial average filtering	329	266	81
Edge enhancement	77	60	78
Conversion to binary image	5	5	100
Multiple-threshold segmentation	72	72	100
Database searches (32-bit records):			
Equal-to search	9	8	89
Maximum search	7	5	71
Greater-than search	9	8	89
Between-limits search	13	10	77
Database searches (4-bit records):			
Equal-to search	9	9	100
Maximum search	7	6	86
Greater-than search	9	9	100
Between-limits search	13	13	100

Table 4.1 Percentage of Short-Sequence Instructions

4.3.2 Instruction Fetch Unit and FIFO

The fetch unit reads an instruction from the instruction FIFO into the 32-bit instruction register when the current instruction finishes executing. It is comprised of two simple one-hot encoded finite state machines (FSMs). The 3-state fetch FSM is synchronized to the FIFO read/write and is therefore clocked by the external bus clock. The 2-state read FSM connects the IQU to the instruction execution unit and runs at the internal clock of the CRAM controller.

The instruction FIFO is a 16 x 32-bit dual-port RAM or register file. This acts as a buffer for the instructions from the host computer to the instruction execution unit. Instruction FIFO status include FIFO Full (FF), FIFO Empty (FE) and FIFO Last Word (FLST) flags. FLST is used for buses, such as the PCI, that signal the termination of a burst transfer one cycle before the last one.

The penalties of a small FIFO are twofold. First, a small FIFO is more likely to be completely empty before the next group of instructions is loaded into it, especially if long-sequence instructions are mixed with short sequences. With an empty FIFO, one or all of the components T_{init}, T_{load}, and T_{flow} of Equation 4.1 contribute to, and hence increase the total execution time. The second penalty of a small FIFO is the increased number of bus transfer retries that the host processor performs in order to transfer CRAM instructions into the FIFO. This number increases for a small FIFO because the FIFO is full more often. While the number of transfer retries does not directly affect the total execution time of an application, it may affect the performance of the host processor if it is operating in a multitasking environment. Apart from increasing the size of the instruction FIFO, the number of transfer retries may be reduced by using an interrupt, instead of polling, to signal the FIFO-full condition

To determine an optimum size of the FIFO, a number of simulations were carried out using practical applications and other theoretical combinations of instructions. An example of the results is shown in Figure 4.6. These were obtained by running Vector Quantization (VQ), with a 64-word codebook and 2 x 2 vectors, on a 50 ns PCI-based CRAM system. (see Section 7.5.1 and Appendix D.3 for the definition, implementation, and CRAM C++ source code for VQ). VQ is reported here as an example because it includes a variety of operations likely to be found in most applications. These operations include addition/subtraction, addition/subtraction with an immediate integer constant, operand extension, comparisons, conditional execution, updating of controller registers,

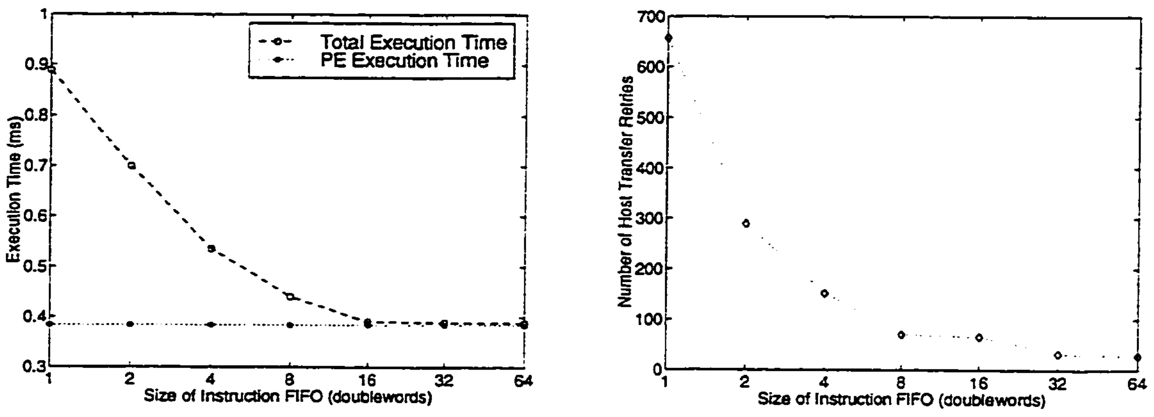


Figure 4.6 Effect of FIFO Size on Performance

and variable assignment. Based on these simulations, a FIFO size of 4 to 16 doublewords is adequate. It was also noted that increasing the size of the FIFO beyond 4 doublewords does not result in a proportional decrease in the execution time. However, a size of 16 doublewords was chosen for the Xilinx and ASIC implementation because of two reasons. First, for a Xilinx FPGA, a 16-word RAM occupies exactly the same number of CLBs as any other RAM with 15 words or less [40]. Secondly, the maximum number of continuous bus cycles allowed for the PCI bus is 16 [36]. Therefore a FIFO with more than 16 words does not have a significant transfer-time advantage, especially if the host processor has to re-arbitrate for the bus after completing the first 16-cycle burst transfer [36].

4.4 Instruction Execution Unit

The instruction execution unit (IXU) gets the instruction loaded in IR and outputs the lowest-level control, address and data signals to the CRAM. It is composed of a microprogram sequencer, an address unit, and parameter, status and command registers. The following sections describe the components of the execution unit and the formats of CRAM instructions.

4.4.1 Instruction Word

All CRAM instructions are 32-bit wide and have a uniform format (Figure 4.7). This RISC-like format reduces the overhead of instruction decoding and flow.

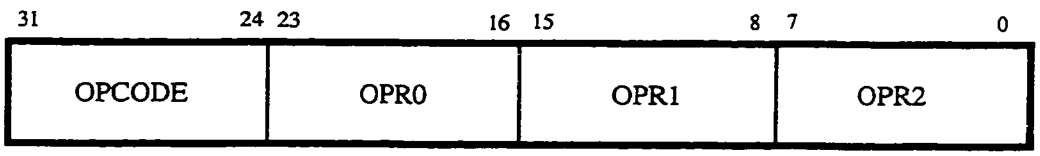


Figure 4.7 instruction Format

The 8-bit OPCODE represents the operation to be performed by the instruction, e.g. ADD. This field points to (is the address of) the beginning of the operation microinstructions in the control store. The 8-bit width means that the control store can be up to 256 words unless address mapping is used. While it might appear that this also limits the number of instructions to at most 256, this is not so because the control store is writable (Section 4.4.5) and hence any address can be re-mapped (even during run-time) to a new instruction by loading its microinstructions from the host computer. Since CRAM application programs will most likely be written in the CRAM C++ programming language (Section 6.2), this address-instruction re-mapping will rarely be necessary because the size of the control store (256 words) has been chosen to accommodate all the microroutines required to implement the language. Section 4.4.5 gives more details on the choice of the size of the control store.

There are three 8-bit operand fields in the instruction. These can either be addresses of operands in a CRAM operation, or they can be immediate values in instructions for loading CRAM controller registers. Immediate values for CRAM operations are not packed into the instruction. They use the write buffer and constant broadcast unit as outlined in Section 4.5.2. To address CRAM memory rows beyond 256, the operand addresses are extended using address extension registers (AX0-AX2). This is described under Address Unit in Section 4.4.6.

One major disadvantage of coding all instructions with a uniform format is that for instructions that require no or fewer operands, some bit fields carry no relevant information and hence waste bandwidth. For example, the instruction to reset a variable in

CRAM memory requires only one operand (the address of the operand). However, this instruction is coded as a 32-bit word, with 16 bits carrying no relevant information. One solution is to pack these unused bits with other information that may be used by this instruction (i.e. extending the functionality of the instruction) or other subsequent instructions. This, however, means extra decoding and a more complex execution unit. In our design, we have exploited the special architecture of the CRAM PEs and the multiplexed CRAM TTOP/Data bus by grouping microroutines that differ by COP and/or TTOP on only one microinstruction. Instead of loading all the microroutines of such instructions into the control store, only one representative routine is loaded and the differing COP and/or TTOP of the instruction is packed into the unused operand fields of the instruction. Two microinstruction bits indicate whether COP and/or TTOP is either coded into the microinstruction (internal COP/TTOP) or should be extracted from the instruction operands (external COP/TTOP). Using this approach has reduced the number of microinstructions required for implementing the CRAM Assembly Language (Section 6.4) from 462 to 197 (more than 50% reduction). This is the main reason why a control store of less than 256 words is feasible in our design. The only hardware overhead added by this are two 8-bit muxes to select between internal and external COP/TTOP. However, these muxes do not add to the critical path of the controller since they are after the pipeline register while the critical path is between IR and the pipeline register (Section 4.4.3). This microinstruction grouping is described in detail in Section 6.6.

4.4.2 Microinstruction Word

The 32-bit CRAM microinstruction word is shown in Figure 4.8. The following paragraphs describe the individual control bit fields. Details of the actual bit settings are described in Appendix A.1.

31 28	27 25	24	23 22	21	20 19	18 16	15 8	7 0
Next Address Instruction	Branch Condition Select	XTTOP	FUNC	RW	ASEL	CCI	Control Opcode (COP)/ Mask/Temp Address	Truth Table Opcode (TTOP) or Branch Address

Figure 4.8 Microinstruction Word

The 4-bit Next-Address Instruction field selects the next address of the control store (i.e. the next microinstruction). The Branch Condition selects one of the eight conditions to be used in a conditional jump or loop instruction. These conditions include the status of the loop counter, the status of the CRAM global-OR and PE shift outputs, and whether the byte read from CRAM using a special read-bank instruction contains all zeroes. The last condition is used for examining the results of searches and comparisons as described in Section 4.4.3. EXTTOP selects TTOP to be the value of either operand 2 (OPR2) of the instruction word or bits [7:0] of the current microinstruction. FUNC, together with RW, is used to indicate a no-operation, a PE operation, or an internal or external CRAM memory access. Note that any controller instruction (CCI) may still be embedded in the microinstruction that has FUNC set to CRAM no-operation. ASEL selects the value to be put on the CRAM address bus. This can either be one of the three address registers, or it can be COP or the address of a system mask or temporary (scratch) location (TMPA). CRAM controller instructions (CCI) are used for loading parameter registers and incrementing address registers. These instructions are described in Table A.4 and Table A.5.

Control bits [15:8] carry either COP or TMPA. There are two reasons why these are put on the same position in the control word. Firstly, they are always used exclusively and hence can occupy the same control word bits without requiring any hardware to multiplex them. This reduces the size of the control word. Secondly, since COP is multiplexed with addresses on the CRAM chip address bus, coding COP as an address means that the address bus multiplexer has to select between four addresses only (AR0, AR1, AR2 and TMPA) without the need for a 5-to-1 multiplexer to cater for the COP input. This reduces the size and critical path of the address unit (Section 4.4.6).

TTOP is coded on the eight least significant bits of the microinstruction. To limit the size of the microinstruction, these bits can also be occupied by the branch-to address during jump (JMP, JMPS, JSR) instructions. This means that these instructions cannot be used in a microinstruction that executes a PE operation (they can however be used with a CRAM read/write operation). This restriction does not affect the performance of the system because, as mentioned in Section 4.4.3, jump instructions are used very rarely in simple microroutines like the ones for CRAM. Also, since a PE operation is usually followed by a memory operation, it is more likely that the jump instruction will be after the memory instruction than after the PE operation. In this case, the jump can be put in the same microinstruction as the CRAM memory operation, thus avoiding the overhead of using an extra microinstruction just to implement the jump.

4.4.3 Microprogram Sequencer

The microgram sequencer is designed as a fairly standard architecture (Figure 4.9). The 8-bit opcode from the instruction register (IR[31:24]) points to the starting address of the microinstructions to be executed. The microprogram counter (uPC) provides the address of the next sequential microinstruction. The stack, the current address register (CAR) and the loop counter are used in subroutine calls, branches and looping.

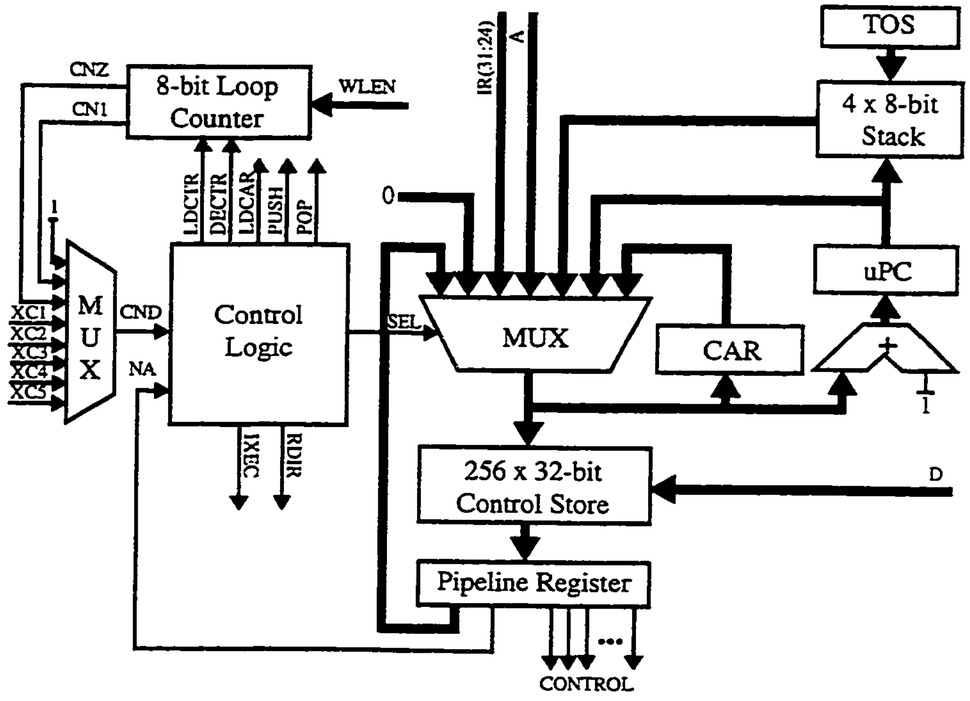


Figure 4.9 Microprogram Sequencer

The loop counter is loaded from the Word Length (WLEN) register (Section 4.6) at the beginning of every instruction. This represents the bit length of the operands in the current instruction. Even though the current CRAM C++ library consists of only CRAM integer objects (usually up to 32-bit values on many systems), the WLEN and loop counters are designed as 8-bits for future expansion (such as the inclusion of floating-point CRAM objects, which will be 64-bits or more). It must be pointed out that the CRAM controller and the CRAM C++ library do not restrict the size of CRAM integers to 32 bits. An application program can use integers of any size up to 256 bits. Also, the 8-bit loop counter allows operations that involve all the rows of the CRAM memory to execute faster. For example, clearing the memory of the C64p1k CRAM chip (1 Kbits/PE) requires only 4 MCLR (memory clear) instructions to be issued by the host processor compared to 32 instructions if the loop counter was 5-bit wide (for up to 32-bit integers).

The control logic generates signals for selecting the next address of the control store (the next microinstruction), loading and decrementing the loop counter, and reading the next instruction from the instruction register (RDIR). The conditions that can be used in looping, branches and subroutines are shown in Table A.2. Conditions derived from within the sequencer include CNZ and CN1. The external conditions (XC1-XC5) are mostly from the CRAM chip (GOR, SLO, and SRO). DZ is primarily designed for scanning out the result of CRAM search or comparison operations. In this case, the boolean result value is scanned (read) a byte at a time until a byte containing a 1 is read in. This byte is saved in the DTR register and can be used, together with the bank address register (CBA), by the host processor to calculate the index of the first PE that yielded a true (1) value during a search or comparison.

4.4.4 A Simplified Microprogram Sequencer

Because CRAM is a general-purpose parallel-processing system, the control store is not intended to include microroutines of application-specific algorithms such as FFT, pixel smoothing, etc. Otherwise the control store would be prohibitively large to be on-chip. Instead, only microroutines of basic operations such as addition, subtraction, maximum/minimum search, etc. are loaded into the control store. One noticeable feature of these basic microroutines is that they do not use most of the sequencing features shown in Figure 4.9. Their dominant feature is the *n*-looping through the microinstructions depending on the number of bits of the operands. Subroutines and sophisticated branching are not as necessary. Even simple branching is used rarely since the testing of the most commonly used CRAM status (Global-OR) is usually done on the CRAM chip and not on the controller (even though the status of the CRAM bus-tie is also passed to the controller). Conditions that are usually tested on the controller (such as DZ) usually occur as end-of-looping conditions and not as branch conditions. Therefore, a simplified microprogram sequencer, shown in Figure 4.10, was designed to reduce the area of the controller. In this design, both the subroutine and branching have been removed. The Loop from (LPR) register contains the address where looping is to start from. To avoid the overhead of including an extra instruction when the first microinstruction in a microroutine is the one to loop from, LPR is automatically loaded with the address of the first microinstruction when any instruction starts executing. It can also be loaded by the sequencer SETLP next address instruction.

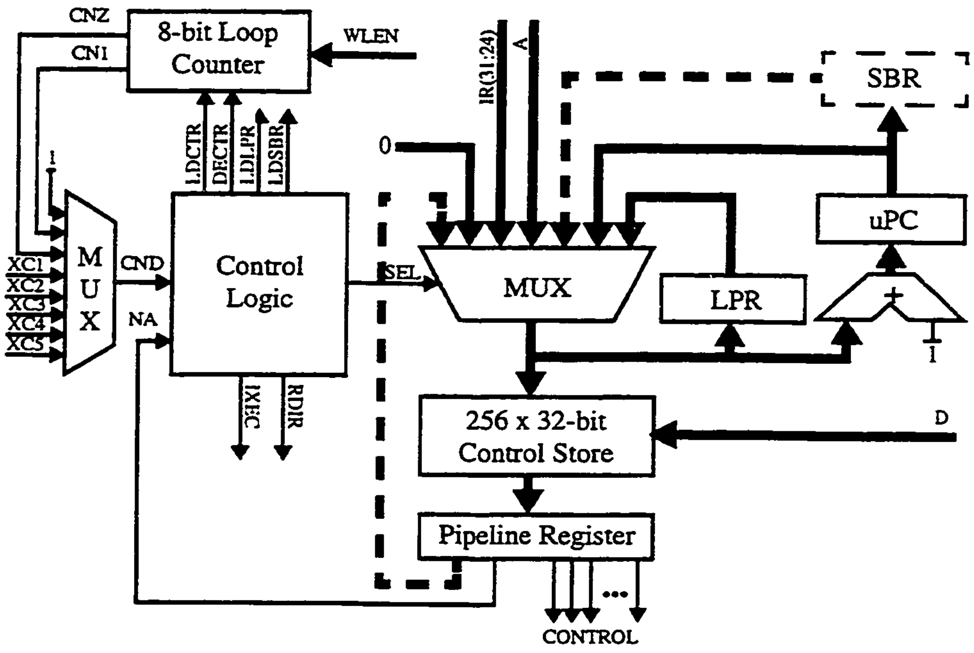


Figure 4.10 A Simplified Microprogram Sequencer

While the microroutines used to implement the CRAM C++ library do not use branching, it is our desire to retain the branching feature for future expansion. Also, the subroutine feature will not be entirely removed, but rather the multi-level subroutines will be replaced by simple single-level subroutine calls. This can be a very useful feature in implementing microroutines that are just the concatenation of the basic microroutines. If this feature is implemented, the stack is replaced by a subroutine return address (SBR) register. In order to be able to use the existing basic microroutines in their usual stand-alone mode (which will be their most common use, anyway) as well as in the concatenation mode, subroutines will not be implemented with the usual return-from-subroutine (RTS) instruction. Rather, the calling microinstruction will set a 1-bit subroutine mode (SMR) register. Whenever a basic microroutine finishes executing, if SMR is set, the next microinstruction address is selected to be the one in SBR, otherwise the address is selected as before. These two features (branching and single-level subroutine calls) are not implemented in the prototype controller (hence are shown dotted in the simplified sequencer of Figure 4.10) because of two reasons. First, excluding these features increases the speed and reduces the area of the controller. Second, and more important, these features are not used by any of the microroutines used to implement the first revision of the CRAM C++ library. However, a controller incorporating these two features, and one using the architecture shown in Figure 4.9, has been designed, synthesized, and fully simulated (including timing simulation). The synthesizable VHDL design files can be used in future prototypes.

4.4.5 Control Store

The control store is writable from the external bus, permitting dynamic refilling of microinstructions. This is important because, unlike general-purpose processors, CRAM processing elements perform operations, such as addition, one bit at a time. This means that a very large number of such operations are possible in CRAM, thus necessitating an arbitrarily large control store. By making the control store writable, it can be made small and refilled when an application requiring different microroutines than those in the control store is to be run. A motivation for making the control store small is to reduce the cost of the controller, and hence the whole CRAM system. A smaller control store also means that it can easily be implemented on-chip, thus increasing the speed of the controller as well as reducing the cost and ease of implementation of the CRAM system (fewer discrete components on the PCB). Also, since microinstruction reading represents the heaviest data movement in the controller, putting this on-chip reduces the controller power dissipation.

The size of the control store is of vital importance to the effective design and performance of the CRAM controller. While a small control store results in a smaller area, for a reasonably large application, it can also degrade the performance of the CRAM system since microinstructions have to be continuously refilled. On the other hand, a large control store completely removes the overhead of microinstruction refilling, but adds considerably to the area of the controller and may even reduce the speed of the controller, especially if it is to be implemented off-chip. As a compromise between performance and area, it was decided to make the control store big enough to store all the microinstructions required to implement the CRAM assembly code, i.e. all the primitive operations in the CRAM C++ library. With this control store size, an application that uses standard C++ operators (+, -, +=, >, &&, etc.) does not have to refill microinstructions. Dynamic refilling or pre-loading of microinstructions is necessary only if the programmer decides to implement some application-specific routines as microcode. As mentioned in Section 4.4.1, an innovative approach has been used to group similar microinstructions. This has halved the number of microinstructions to be stored on the control store. More important, this number (197) is below 256. It is easier and cost-effective to implement on-chip synchronous RAMs of less than 256 words in both an FPGA (the Xilinx Memgen program can automatically generate RAMs of up to 256 words) and an ASIC process (some technologies, such as the Nortel 0.8 μm BiCMOS and TSMC 0.35 μm CMOS, include 128-word and 256-word synchronous RAM macros in their standard cell libraries).

From the above argument, the size of the control store can be set to 197 words or more. However, a size of 256 was chosen because of two reasons. First, the depth of RAM macros in ASIC processes and some FPGAs run in steps of 2^n ($n = 3, 4, 5,...$). Common values are 16, 32, 64, 128 and 256. Secondly, the extra free space in the control store may be used for future microcode extension. It also allows pre-loading of application-specific microcode, thus reducing the penalty of refilling microinstructions during run-time. Figure 4.11 demonstrates the penalty of microinstruction refilling when the size of control store is set to less than the number of the required microinstructions (197). Again, this is based on a 64-word VQ and a 50 ns PCI CRAM system.

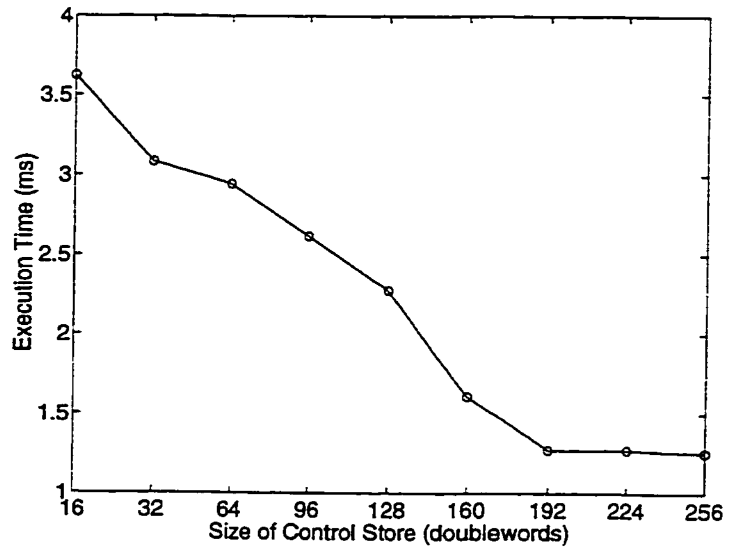


Figure 4.11 Effect of Control Store Size on Performance

In Figure 4.11, the main contributions to the increased execution time when the control store is small is the host processor execution time and the time to load microinstructions. The host processor execution time is increased because it now executes more conditional code, and also sets up more bus transfers. The time to load microinstructions is even more significant if CRAM is interfaced to a slow host bus such as the ISA bus. The other contributors to the increased execution time are the time to fill an empty instruction queue, and the time to load instructions into the FIFO. Before executing an instruction whose microinstructions are not in the control store, its microinstructions are first loaded into the controller. To make sure that active microinstructions are not overwritten, the controller status has to be read (through interrupts or polling) to make sure that all instructions in the IQU have finished executing before new microinstructions are loaded. Apart from increasing the host execution time, this also means that the instruction queue is always empty before a new instruction is loaded. This is the reason why the time to fill an empty instruction queue and the time to load instructions into the FIFO both go up as the control size decreases. It must be pointed out that as the size of the control store increases, more microroutines are loaded permanently into the control store depending on the frequency with which they are used in applications, and whether they belong to a group of instructions whose microroutines are already in the control store. For example, if the last microroutine in the control store is for addition, then the microroutine for subtraction will be the next one to be loaded should the size of the control store be increased. The same for microroutines for comparisons (greater than, equal to, etc.), or logical operations (AND, OR, XOR, etc.). This is the reason why the curve in Figure 4.11 has a stair-case shape in some regions. The flat portions, for control size of less than 192, represent increments of control store size for which there is no corresponding improvement in performance simply because microroutines loaded in that memory range are not used by that particular program.

4.4.6 Address Unit (ADU)

The address unit shown in Figure 4.12 consists of three 8-bit address registers (AR0-AR2), three corresponding address extension registers (AX0-AX2), an 8-bit bank address register (CBA), and an 8-bit incrementer/decrementer. The address registers AR0, AR1 and AR2 (ARn) are loaded from the instruction register at the beginning of every instruction (i.e. when an instruction is read from IR by the instruction execution unit). For memory access instructions, the values in ARn are concatenated with the values of AXn and CBA to form the address of the memory. AXn:ARn forms the row address of the PE local memory, which is the address used during internal CRAM memory accesses. The bank address is used during memory accesses between the CRAM and the controller to select a group of 8 PEs (for 8-bit RAM) whose memory is to be accessed. That is, for external memory accesses, the address is CBA:AXn:ARn. AXn and CBA are preloaded from the host processor using specific controller load instructions (LDAXn and LDCBA). This allows packing up to three operand addresses in the 32-bit instruction (Figure 4.7), while using addresses which are more than 8 bits wide. 256 continuous rows can be accessed in a program without the need to update AXn. The CRAM C++ compiler can also optimize the allocation of variables in memory so that AXn need as little updating as possible. The incrementer/decrementer computes the next value of the currently selected ARn register if there is an INCA or DECA instruction in the current microinstruction.

In the first prototype, not all bits of AXn and CBA are implemented. AXn is implemented as a 2-bit register because the maximum memory per PE for the C64p1k and

C512p512 CRAM prototype chips is 1Kb (i.e. 10-bit address). CBA is implemented as a 6-bit register because the C512p512 CRAM chip has 512 PEs.

Since the address registers are always loaded with IR[23:0] at the beginning of an instruction, they also act as temporary registers for latching operands of any instruction, especially the controller load-register instructions. In other words, OPR0, OPR1 and OPR2 are just aliases of AR0, AR1 and AR2, respectively.

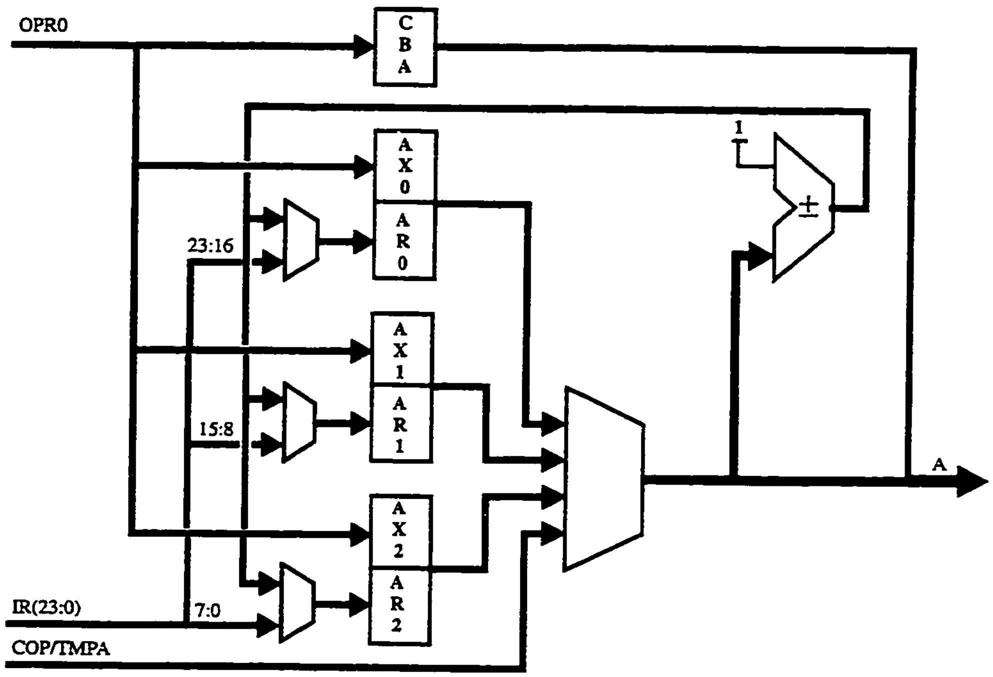


Figure 4.12 Address unit

4.5 Read/Write Buffers and Constant Broadcast Unit

Like CRAM instructions, data transfers from the host processor to the CRAM chip are buffered through the read/write buffers. In this way, data can be transferred quickly from the host processor to the controller using burst cycles, and then later transferred to CRAM. This feature also allows parallelism in that when the CRAM is executing instructions currently in the FIFO, the host can preload data into the write buffer, and then issue an instruction (which will be queued in the FIFO) to transfer this data from the buffer to CRAM. This reduces the time of initializing or reading CRAM variables.

The other reason for including data buffers in the controller architecture is that they simplify the timing of data transfer between the host and CRAM. If CRAM was connected directly to the external bus, there would have been a need to synchronize the data transfers with the flow of instructions in the instruction execution unit. Another problem of connecting CRAM directly to the host bus is the danger of violating the bus electrical characteristics as more CRAM chips are added to the system. For example, the PCI specification states that each device (embedded or an add-in card) may place only one load on each shared PCI signal line [36]. By buffering all data transfers through the controller, a number of CRAM chips may be connected to the controller, and the loading problem becomes local to the controller. This is less of a problem because the controller is not connected to as many devices as the system bus.

The fourth reason for using the read/write buffers is that the write buffer has been ingeniously combined with CRAM data bus multiplexing circuitry to implement a novel constant broadcast unit. This is described in Section 4.5.2. Figure 4.13 shows the block diagram of the read/write buffers and the constant broadcast unit. RIBA and WIBA are the counters for the internal address of the read and write buffers, respectively. These point to the byte that is to be used in the current data transfer between the CRAM and the controller. For transfers to/from the external bus, the external bus address (A) is provided from the CRAM-host interface unit.

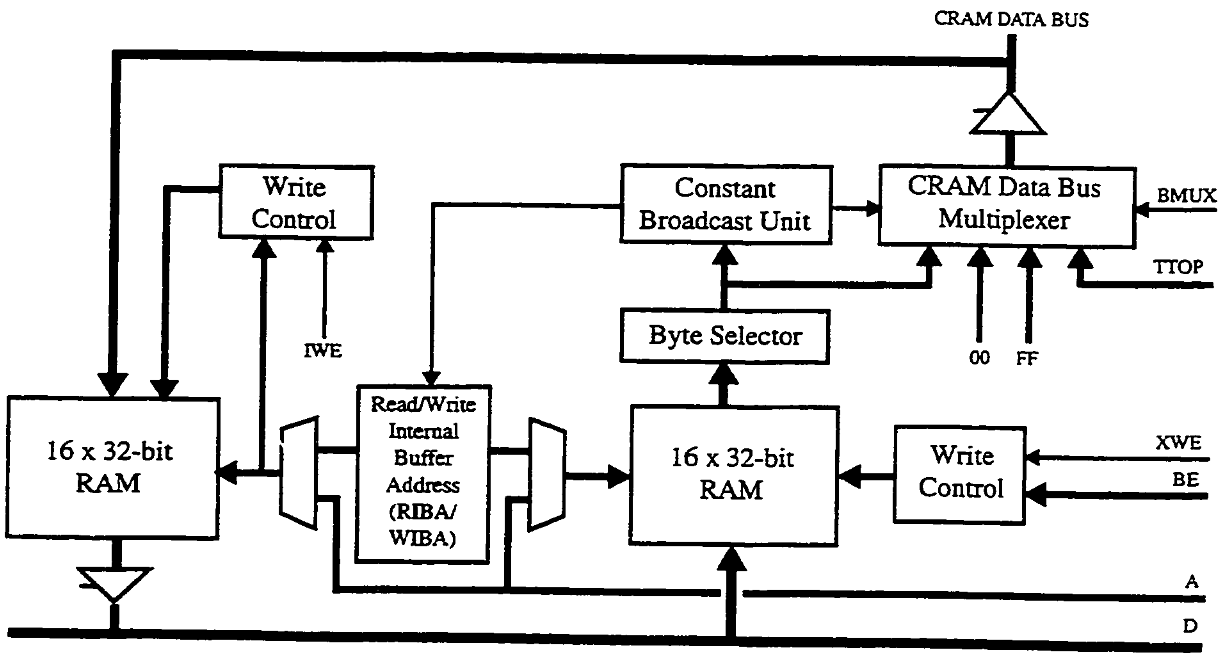


Figure 4.13 Read/Write Buffer and Constant Broadcast Unit

4.5.1 Effect of Read/Write Buffers on Variable Accesses

The read/write buffers are designed to match the sizes of the host and CRAM buses. To allow for scalability, the buffers are made of independent 8-bit memory blocks. A few such blocks are combined to match the width of the external bus so as to maximize data transferred in one cycle. Figure 4.14 shows how the buffers are organized for a CRAM system with an 8-bit CRAM data bus and a 32-bit PCI bus or a 16-bit ISA bus. Notice that because of this organization, the ISA buffers have fewer bytes when compared to the PCI. In other words, the size of the buffer is automatically scaled down to match the transfer capabilities of the host bus.

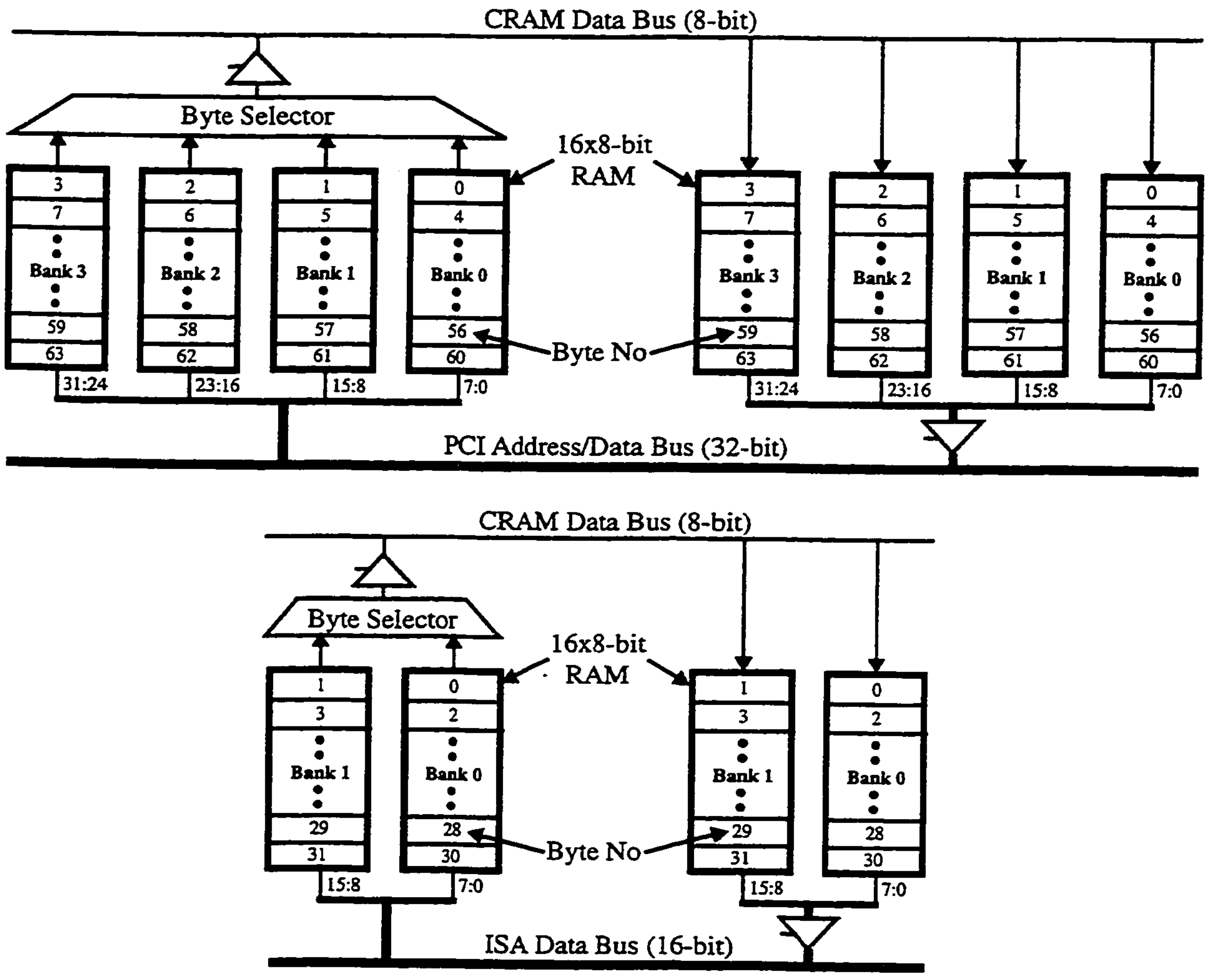


Figure 4.14 Data Buffers Organization

To transfer data for CRAM variables from the host, the data is first loaded into the write buffer. A `WRITE` instruction is then loaded into the instruction FIFO, after which it is executed, transferring the data from the buffer to CRAM. In order to reduce the total data transfer time, the loading of data and the `WRITE` instruction from the host to the controller must be done in parallel with the execution of the `WRITE` instruction. This is accomplished by dividing the buffer into two, and then executing the data transfers in chunks equal to half the size of the buffer. Assume a buffer size of B bytes, a CRAM system of cycle time T_c, and a host bus of cycle time T_{bus} (the following analysis is based on either an ISA bus or any 32-bit bus that supports burst transfers). Therefore, during each step, $B/2$ bytes are loaded into the buffer in time T_{xdata} given by

$$T_{xdata} = T_{host} + \begin{cases} \frac{B}{2}T_{bus}, & ISA \\ \left(1+\frac{B}{8}\right)T_{bus}, & otherwise \end{cases} \tag{4.7}$$

where T_{host} is the average time to initialize and set up a data transfer on the host. The time to load the `WRITE` instruction (4 bytes) from the host to the CRAM controller instruction FIFO is

$$T_{xins} = T_{host} + \begin{cases} 4T_{bus}, & ISA \\ 2T_{bus}, & otherwise \end{cases} \tag{4.8}$$

Therefore, the total time to load the $B/2$ data bytes and the `WRITE` instruction word from the host to the CRAM controller is

$$T_{xload} = 2T_{host} + \begin{cases} \left(4+\frac{B}{2}\right)T_{bus}, & ISA \\ \left(3+\frac{B}{8}\right)T_{bus}, & otherwise \end{cases} \tag{4.9}$$

For CRAM with an 8-bit data bus, the execution time for the `WRITE` instruction is

$$T_{exe} = \left(2+\frac{B}{2}\right)T_c \tag{4.10}$$

The $2T_c$ is for the instruction to flow through the instruction pipeline. If the loading from the host is done in parallel with the execution of the WRITE instruction, then the total time to transfer N bytes of the CRAM variables from the host is given by

$$T_{tx} = T_{lat} + \frac{2N}{B}\begin{cases} T_{exe}, & T_{exe} \geq T_{xload} \\ T_{xload}, & T_{exe} < T_{xload} \end{cases} \tag{4.11}$$

T_{lat} is the latency time for the parallelism, and is equal to the time to load the first $B/2$ data bytes, the WRITE instruction word, and three other instruction words for setting up parameter registers. Since no other operation on the CRAM chip can be executed in parallel with the transfer of data between the R/W buffers and CRAM, the minimum T_{tx} possible is when only T_{exe} contributes to the data transfer time, i.e. $T_{exe} \geq T_{xload}$. Using Equation 4.9 and Equation 4.10, the minimum size of the buffer at which this occurs can be found by using Equation 4.12.

$$B_{min} = \begin{cases} \dfrac{2(2T_{host} + 4T_{bus} - 2T_c)}{T_c - T_{bus}}, & ISA \\ \dfrac{8(2T_{host} + 3T_{bus} - 2T_c)}{4T_c - T_{bus}}, & otherwise \end{cases} \tag{4.12}$$

Since T_{host} depend on the type of computer used as the CRAM host, the value of B_{min} can be evaluated either by simulating the data transfers on a particular host, or by approximating the average value of T_{host} (using a few simulated values) and then using Equation 4.12. The former is more accurate. Table 4.2 shows B_{min} for different CRAM systems simulated on a 133 MHz Pentium PC. For the ISA bus, the condition $T_{exe} \geq T_{xload}$ is never true for any of the tabulated CRAM cycle times because of its low data transfer rate.

	$T_c = 100$ ns	$T_c = 50$ ns	$T_c = 25$ ns
PCI	16	32	80
VME	16	48	N/A
EISA	16	56	N/A
ISA	N/A	N/A	N/A

Table 4.2 Minimum Buffer Size for $T_{exe} \geq T_{xload}$ during Variable Initialization

Figure 4.15 shows the variation of the time to load 64K bytes of a 256 x 256 8-bit image from the host to a 50 ns CRAM system. It must be pointed out that for buffer sizes of 4 bytes or less (2 bytes or less for ISA), there is no advantage in using parallelism because the number of bytes during each transfer is less than the size of the external data bus. Therefore, the loading of data into the write buffer, and the execution of the WRITE instruction are done sequentially. Also note that even though the reduction in the total load time in Figure 4.15 is more than 90%, we quote it at 80% because a practical system would not implement a single byte buffer (or data register) for an external bus that can transfer four bytes in one cycle. Therefore, a practical minimum buffer size is actually four bytes for EISA, PCI, and VME, and two bytes for the 16-bit ISA bus. These are the sizes used as datum for performance evaluation.

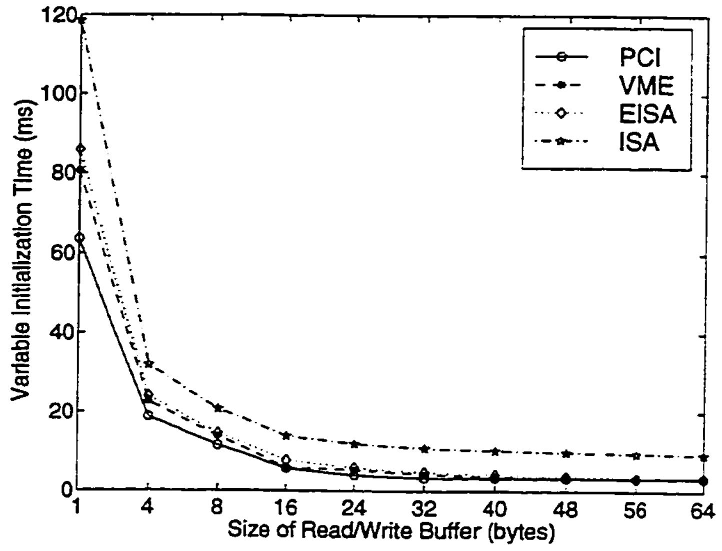


Figure 4.15 Effect of R/W Buffers on Variable Initialization

For buffer sizes of less than B_{min}, T_{xload} dominates the total load time. For this case, increasing the buffer size reduces the total load time because the host overhead is reduced (less looping, fewer data transfer initialization and setup, etc.), and for burst cycles, the actual transfer time on the external bus is also reduced. For buffer sizes of greater than B_{min}, the total load time is equal to T_{exe} (plus the latency), and hence becomes less dependent on the data transfer characteristics of the host bus. Again, this is important for

CRAM as a general-purpose system that can be used on different platforms.

In terms of choosing the size of the buffer, these simulations have shown that most of the reduction in the total load time occurs by increasing the buffer size to 32 bytes. Further increases results in correspondingly smaller performance improvements. However, because of the configuration of the buffers as explained earlier in this section (Figure 4.14), a size of 16 doublewords (64 bytes) for the 32-bit buses, and 16 words (32 bytes) for the 16-bit ISA bus, was chosen for the Xilinx FPGA implementation because RAMs with a depth of 16 or less occupy the same number of CLBs [40]. Also, for the 32-bit buses, this size is within the range of B_{min} for the cycle times of the prototype CRAM system (T_c = 100 ns), as well as that of a CRAM system likely to be implemented (T_c = 50 ns).

4.5.2 Constant Broadcast Unit

There are a number of ways to implement operations with scalar constants in bit-serial SIMD processors. The controller may produce the microroutines for each of the 2^n constants (for n-bit operands). But the amount of memory required to store all the 2^n microroutines might be prohibitively large unless n is very small. A more common method is to multiplex the microinstructions depending on the bit coming out of a select (constant) register [41]. This means that all microinstructions for constant operations must be in pairs, one for operation when the value of constant bit is '1', and the other for operation with '0'. If these pairs are laid side by side, the microinstruction word, and hence the control store size, is increased. If on the other hand, the microinstruction pairs are placed one after the other (which is the preferred approach), then the bit coming out of the select register is used as the LSB of the address to the control store. Apart from increasing the depth of the control store, this approach has several disadvantages. First, the use of the select register as one of the bits for the control store address adds extra multiplexing for the address. This may reduce the speed of the sequencer since control store address selection is usually part of the sequencer critical path. Second, code generation becomes more complicated because for operations with constants, microinstructions for operations with a '1' always have to be on an odd address, while those for '0' have to be on an even address. The third and, as far as CRAM is concerned, the most critical drawback of this approach is that the constants' size is limited by the size of the select register, unless special programming techniques are employed. This size limitation is acceptable for systems designed to process fixed-size operands such as in pixel-processing (usually 8-bit operands). It would however limit the versatility of CRAM, in which the ability to vary the bit-length of operands (even during run time) is one of its major strengths.

A novel constant broadcast unit shown in Figure 4.16 is used for CRAM. It takes advantage of the existing write buffer and also exploits the fact that for CRAM, the PE truth table opcode (TTOP) is multiplexed with data on the CRAM data bus. The constant bit (KBIT) is selected by the write buffer internal address (WIBA) register and the constant bit pointer (KPT). WIBA, which is an already implemented feature of the write buffer, selects the byte from the write buffer using the byte selector (Figure 4.13). KPT, which is a 3-bit binary up counter, then selects one bit (KBIT) from this byte. If the currently executing microinstruction is for an operation with a constant (i.e. LDK is asserted), then LDK and KBIT tell the data bus multiplexer to select either 0x00 (if KBIT is '0') or 0xFF (if KBIT is '1'). These two values are the opcodes (TTOP) to reset or set a CRAM PE register. The register to be reset/set is specified the same way as in ordinary PE microinstructions, and is reflected in the value of COP. At the end of executing an LDK microinstruction, KPT is incremented to point to the next bit of the constant. If KPT was 7 ("111") before incrementing, it wraps over to zero and instead WIBA is incremented. This

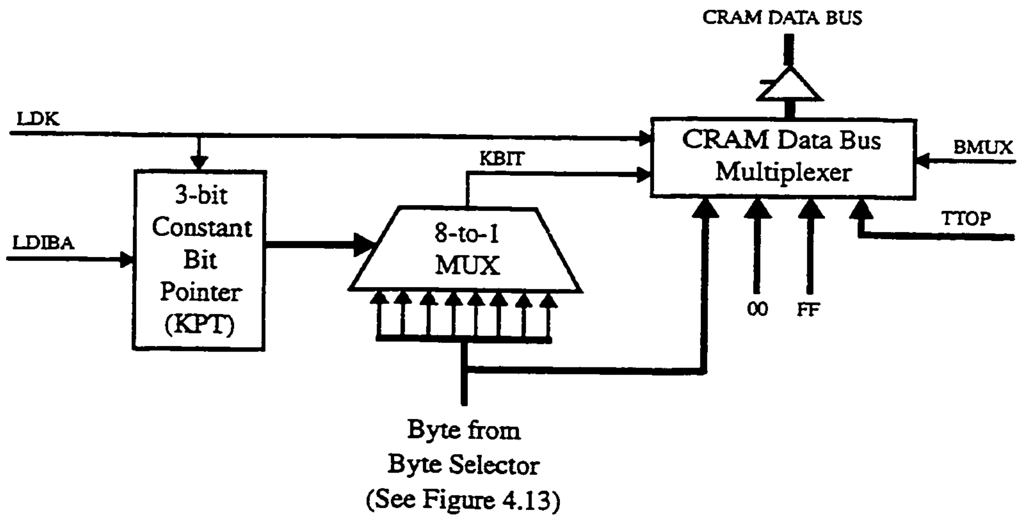

CRAM DATA BUS
LDK
KBIT
CRAM Data Bus Multiplexer
BMUX
3-bit Constant Bit Pointer (KPT)
LDIBA
8-to-1 MUX
TTOP
00
FF
Byte from Byte Selector (See Figure 4.13)

Figure 4.16 Constant Broadcast Unit

takes care of constants which are more than one byte in size or preloaded multiple constants. If constants are preloaded into the write buffer, you can point to any of these constants by loading WIBA with the corresponding buffer address. Loading WIBA also resets KPT to 0 in order to point to the first bit of the constant. If a microinstruction does not involve an operation with a constant, the data bus multiplexer will select either TTOP (microinstruction bits [7:0]) for PE operations or the data byte from the write buffer for memory writes from the controller to CRAM.

The extra hardware for the broadcast unit is the 3-bit counter (3 flip-flops) and an 8-to-1 multiplexer. The data bus multiplexer is also increased from a 2-to-1 to a 4-to-1 multiplexer. In a Xilinx FPGA implementation, the increased multiplexer does not add to the delay or area since a 2-to-1 and a 4-to-1 multiplexer both use a single CLB. In an ASIC implementation, the additional area is minimal, and the extra one-gate delay is of no effect since the critical path still remains in the sequencer.

Apart from the simplicity of this constant broadcast approach, there are other very important advantages. First, the constant size is now only limited by the size of the write buffer, which is large. Note that the size of the write buffer was made large not because we wanted to accommodate large constants, but rather because we wanted such a large buffer for efficient data transfers. The second advantage is that now a number of constants can be preloaded using the faster burst cycles, or constants can be loaded in parallel with the execution of the operate-immediate instruction or other instructions in the FIFO. This is similar to the use of the R/W buffers for data transfers as described in Section 4.5.1. The big difference is that operate-immediate instructions take a much longer time to use the data loaded in buffer. For example, an add-immediate (`ADDI`) instruction with a b-bit integer constant has $(5b + 1)$ microinstructions. Therefore, for 8-bit constants, an `ADDI` instruction takes 41 CRAM clock cycles to use one byte of the data in the buffer, whereas a `WRITE` instruction takes only 1 clock cycle to use the byte. Because of this, the value of B at which $T_{exe} \geq T_{xload}$ is much smaller for operate-immediate instructions. For 8-bit `ADDI`, the expression for B_{min} is given in Equation 4.13, and is simulated to be 8 bytes for all the system combinations tabulated in Table 4.2.

$$B_{min} = \begin{cases} \dfrac{4(T_{host} - T_c)}{41T_c - 5T_{bus}}, & ISA \\ \dfrac{16(T_{host} + T_{bus} - T_c)}{164T_c - 5T_{bus}}, & otherwise \end{cases} \quad (4.13)$$

Figure 4.17 shows the execution time of adding 256 8-bit integer constants to 8-bit CRAM variables for CRAM systems used in Figure 4.15. For buffer sizes greater than 8, the execution time of the ADDI instruction reduces by more than 35%. Also, this execution time becomes almost independent of the host bus, once again an important feature for the implementation of CRAM on different platforms.

The value of B_{min} for operate-immediate instructions is dependent on the type of instruction and the precision of the operands. Simpler operate-immediate instructions such as load-immediate (broadcast a constant value to all elements of a CRAM variable), or small-precision operands, result in fewer CRAM microinstructions. This increases B_{min}. However, since the fastest CRAM operate-immediate instruction (load-immediate) has $2b$ microinstructions (for b-bit operands), the execution time for operate-immediate instructions is always bigger than (typically more than 10 times) that of the WRITE instruction in Equation 4.10. Therefore, the buffer size set for efficient data transfers in Section 4.5.1 is automatically more than adequate for the constant broadcast unit.

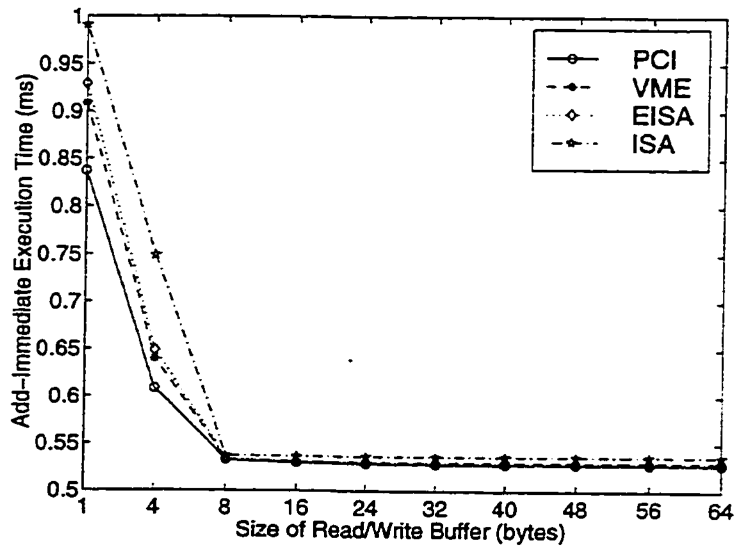


Figure 4.17 Effect of Buffer and Constant Unit on Operate-Immediate Instructions

4.6 User-Accessible Registers

There are two groups of registers that can be accessed by the programmer. The first group is memory-mapped. These include registers that are usually initialized at the beginning of program execution or at boot-up (such as command registers), and registers that are read to evaluate the status of the CRAM system. CRAM command registers specify the characteristics of the CRAM chip (such as type of PE), and the use of other controller resources such as the control store and interrupt lines. CRAM system status include global-OR and PE shift outputs, whether instructions are still executing or pending in the controller, and whether the controller issued an interrupt to the host computer. The second group of registers are those that usually change between instructions. These are used to set controller parameters such as address extension and word length. Most parameter registers are not memory-mapped and can only be initialized with specialized load-register instructions. Table 4.3 summarizes their characteristics. Details of all user-accessible CRAM controller registers can be found in Appendix A.2.

Figure 4.18 shows the memory map of the user-accessible registers. Registers outside the memory map can only be written using controller load-register instructions. This ensures that parameter registers are updated synchronously with the instructions that they are supposed to affect.

Register	Bits	R/W	Read/Write Method
Read Buffer Internal Address (RIBA)	6	W	LDIBA
Write Buffer Internal Address (WIBA)	6	W	LDIBA
Buffer Address Increment (BAI)	6	W	LDBAI
CRAM Bank Address (CBA)	6	R/W	LDCBA/Memory-mapped
Read-back Data (DTR)	8	R	Memory-mapped
Word Length (WLEN)	8	W	LDWLEN
Shift-Input selection Code (SIC)	2	W	LDSIC
Interrupt Number (INTNO)	1	W	SETINT
Address Extension Register 0 (AX0)	2	W	LDAX0
Address Extension Register 1 (AX1)	2	W	LDAX1
Address Extension Register 2 (AX2)	2	W	LDAX2

Table 4.3 CRAM Parameter Registers

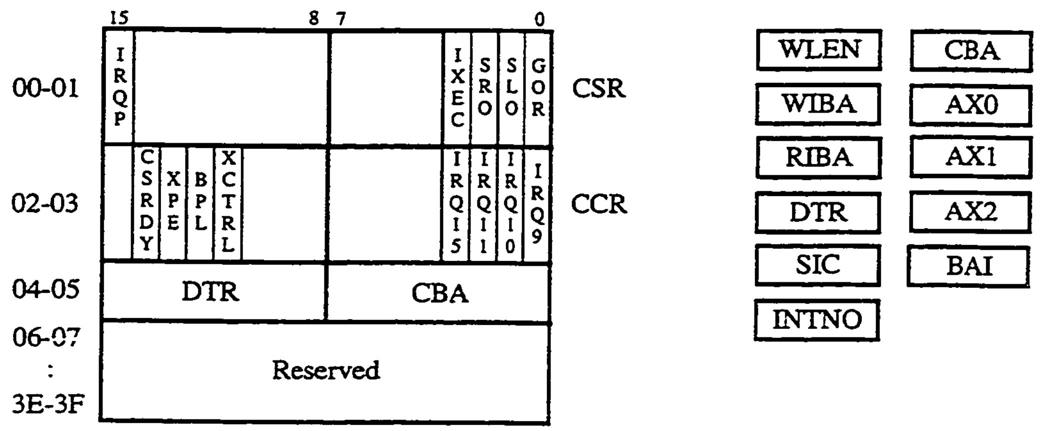


Figure 4.18 User-Accessible Registers

4.7 Memory Map

Apart from the parameter registers described in Section 4.6, all CRAM controller units that can be accessed by the programmer are memory-mapped. Since PCI configuration registers are qualified by the PCI device select (IDSEL) pin, they are not included in the CRAM controller memory map shown in Figure 4.19.

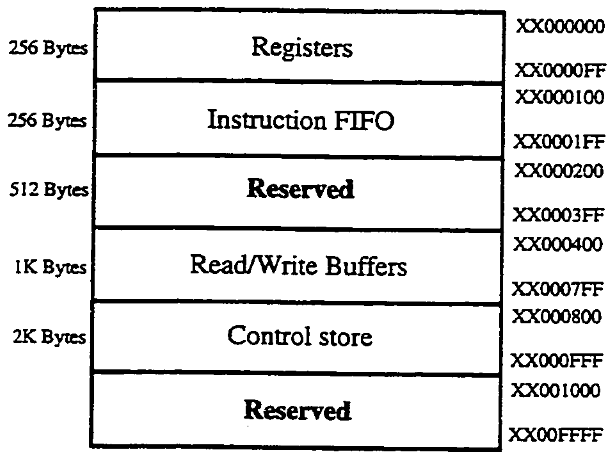


Figure 4.19 CRAM Controller Memory Map

The eight most significant bits of the address select the CRAM card. These are bits 31:24 for 32-bit buses (PCI, VME and EISA), and bits 23:16 for the 16-bit address of the ISA bus. The memory map has been allocated in such a way as to minimize the decoding of the controller units' addresses. For example, to decode the address of the control store, only bit [11] is examined, for the read/write buffers, address bits [11:10] are compared to the value "01", and for the controller registers and instruction FIFO, their addresses are decoded by comparing address bits [11:8] to "0000" and "0001" respectively.

4.8 Summary

In this chapter, the architecture of the CRAM controller has been described. The controller has been designed with a general-purpose architecture to enhance the use of CRAM as a general-purpose parallel-processing system. A microprogrammed controller allows the optimization of application-specific tasks in software. The architecture has also been kept as simple as possible, and its area minimized, in order to reduce system costs, allow easy implementation of a CRAM system on a single size-constrained PC card, and facilitate future integration of the controller and the PE array on the same chip. In this regard, a simplified microprogram sequencer is used, and an innovative approach has been used to group microinstructions, halving the size of the control store. Various features have been used to enhance the performance of a rather relatively simple controller. A FIFO-based instruction queue improves the performance of short-sequence instructions by a factor as high as 10 times. It also allows the controller to approach an ideal controller (100% PE utilization) at a smaller number of microinstructions per instruction. This translates into increased performance for a wider range of applications. The use of read/write buffers for data transfers from the host to CRAM chips simplifies synchronization, eases electrical and physical loading of the host bus, and may reduce the time of loading data onto CRAM by as much as 80%. A novel constant broadcast unit based on the buffers improves the performance of operate-immediate instructions by more than 35%. It also reduces the size of the control store, and simplifies the use of variable-size constants. Finally, we have demonstrated that these three performance-enhancement features also minimize the effect of the host bus on the performance of a CRAM system [42]. This is very important for CRAM as a general-purpose system because it means that CRAM systems can be implemented for a variety of platforms, including slow host systems such as ISA-based computers and embedded systems that use slow microcontrollers.

Chapter 5

CRAM System Implementation

This chapter describes the implementation of a CRAM system. Section 5.1 describes the design flow, CAD tools, and technologies used in the design, verification, and implementation of two CRAM controller prototypes. Section 5.2 gives the implementation details of the PCI and ISA CRAM controllers in a Xilinx FPGA and TSMC 0.35 μm CMOS technology. This section also describes the testing of the prototypes. Section 5.3 describes the implementation of a CRAM system prototype using the C64p1k CRAM chip and the Xilinx ISA CRAM controller.

5.1 Controller Logic Design and Verification

5.1.1 Design Flow

Although a full-custom design approach results in higher performance and smaller die size [48], design automation and flexible design strategies are usually used in the design of processors and other complex digital systems in order to reduce the design time and to provide a design model that is highly adaptable to different implementation technologies and applications [49], [50], [51]. Therefore, the design flow used in the design of the CRAM controller is that of a typical automated synthesis design. The detailed steps are however specific to Synopsys, Xilinx XACT Step, and Cadence, the three CAD tools used in this case for logic design and synthesis, FPGA placement and routing, and ASIC layout design, respectively. Figure 5.1 shows the details of the design flow. The first step, which is the determination of the architecture from the design specifications is described in Chapter 4. The other steps, plus the tools used, are described in the following sections.

5.1.2 VHDL Behavioral Description and Simulation

VHDL has the advantage as a specification and synthesis language in that it can describe hardware at various levels of abstraction, from the architectural (behavioral) level down to the structural level [52], [53]. It also provides a generic design entry platform in that the design may be created in VHDL and the target implementation technology chosen later [54]. In this case, the synthesis tool can be used to map the design to any specific target technology, thus forgoing the task of a complete logic redesign should the initial intended target technology become obsolete or unavailable. Lastly, VHDL synthesis tools allow designers to follow a true top-down design methodology and also allow a quick comparison of trade-offs in implementing the same design in different technologies or styles [55]. In general, high-level synthesis allows designers to describe a design in a purely behavioral form, devoid of implementation details, and then synthesize the design using CAD tools. This achieves higher productivity gains since the design process is moved to higher abstraction levels, where designers can specify, model, verify, synthesize, simulate and debug designs in less time [52]. It is for these reasons, plus the general advantages of automated synthesis, that VHDL synthesis tools were employed in the design of the CRAM controller. The choice of VHDL over Verilog as the description language for the design was justified by the author's level of knowledge and experience with VHDL when compared to Verilog, and by the fact that the Xilinx tools available at the time (XACT 5.2.1) only supported VHDL for post-place and route (timing) simulation.

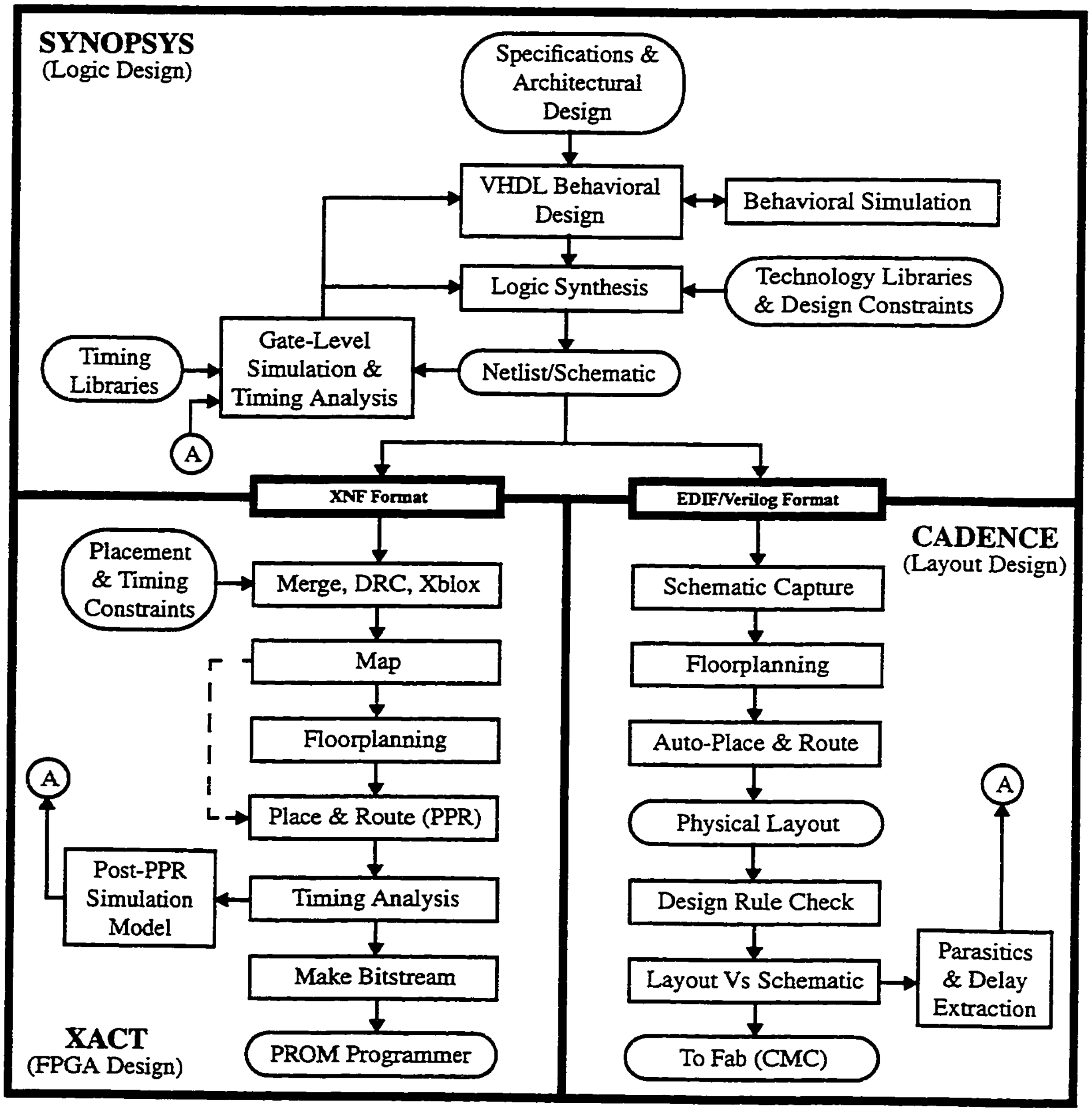


Figure 5.1 Design Flow

The VHDL behavioral description step involved a detailed specification of the controller through several iterative steps of partitioning, description and refinement. Variations in VHDL coding styles have significant effect on the logic synthesis optimization process. This is known as the problem of syntactic variance [52]. Generally, a structured, object-oriented style, that separates control and operations, is recommended for the writing of behavioral descriptions for synthesis of area-efficient hardware [56]. It is also important that the designer have very good knowledge of and experience with the VHDL language, as well as the translation, optimization and technology-mapping techniques used by a specific synthesis tool. In order to predict the results of the synthesis process, and hence effectively guide the tool in synthesizing more efficient and simpler hardware, one also needs to have a fairly good knowledge of conventional digital design techniques. All these factors have been extensively and effectively applied in this design.

After the behavior of each component has been described and simulated to verify its functionality, the components were connected up using structural VHDL, and then the whole controller was simulated to verify its functionality. Therefore problems relating to the functionality and interconnection of the controller functional blocks were detected and corrected at this stage. This structural design specification was introduced at this early stage, as opposed to using a flattened behavioral design, because the synthesis task becomes much easier and more efficient as the amount of structural detail increases [52].

5.1.3 Logic synthesis, Timing Analysis, and Gate-Level Simulation

High-level synthesis consists of converting the abstract VHDL behavioral description of the design into a register-transfer level (RTL) design implementing that behavior. Each component in the synthesized RTL netlist is then designed into the target implementation technology by means of logic synthesis and technology mapping tools. For this design, the target technologies include Xilinx XC4000E series FPGAs, Nortel 0.8 μm BiCMOS (BATMOS), HP 0.5 μm CMOS (CMOSIS5), and TSMC 0.35 μm CMOS.

After the synthesis, the Synopsys tools were used for static timing analysis and gate-level simulation. The timing analysis included investigating the design maximum/ minimum delay paths, critical paths, as well as setup and hold times. The netlist was then saved as a VHDL gate-level netlist and simulated to verify the functionality of the synthesized circuit. Unlike behavioral simulation, the gate-level simulation also includes gate delays and hence gives approximate timing information of the design. However, this timing information does not include routing and parasitics delays, which may be more dominant than gate delays for FPGAs and small feature size technologies such as the 0.35 µm. Routing delays are simulated in post layout (or post-PPR) simulation.

5.1.4 Xilinx FPGA Design Flow

Once the Synopsys synthesis and simulation stages are complete, the design netlist is written into the Xilinx format (XNF) and exported to the XACT tools. The first stage in the Xilinx flow is to prepare the design for the succeeding tools. This includes converting the Synopsys XNF netlist to the 'real' Xilinx XNF, combining the XNF files to form one flat (non-hierarchical) design file, specifying timing and placement constraints (especially necessary in the PCI CRAM controller for which timing is very tight), performing design rule check (DRC), and using Xilinx-optimized blocks of logic (XBLOX). After this, the design is mapped and floorplanned. While floorplanning is seldom used in the Xilinx design flow, it yields very good results especially if timing is critical or CLB resources are limited. Therefore, for the PCI CRAM controller, floorplanning was used to reduce routing delays of critical signals in order to meet the PCI 7 ns setup time in an XC4013E FPGA. The placement constraints from the floorplan are then used by the PPR tool to place and route the design. Static timing analysis is performed using Xdelay. In order to perform a post-place-and-route VHDL simulation, a post-layout XNF file with detailed timing information is generated. This file is then used to generate the timing model of the design as a VHDL architecture and a corresponding SDF file, which are then used to simulate the controller. Once the design functionality and timing are verified through the post-layout VHDL simulation, the configuration bitstream is generated and formatted into an MCS-86 (Intel) PROM format. Finally, the controller configuration PROM is programmed.

5.1.5 ASIC Layout Design Flow

For the ASIC technologies, the design is imported form Synopsys to Cadence through an EDIF or Verilog netlist. Once in Cadence, an auto place and route view is created from the schematic, and the design is placed and routed using Cell Ensemble. After that, Design Rule Check (DRC) is performed and the layout is compared to the schematic (LVS). For BATMOS and CMOSIS5, there is no support for post-layout timing verification. That is, routing and parasitics delays are not backannotated into the original design. However, CMC is currently working on the supporting of this feature in the TSMC 0.35 μm CMOS technology. Otherwise, for now, LVS is the only final check on the design layout. This is not a major problem for a digital system running at slow speeds, like the CRAM controller is.

5.1.6 CRAM System VHDL Simulation Model

Figure 5.2 shows the model that is used in all VHDL simulations involving the CRAM controller or the CRAM system as a whole. This consists of the VHDL models of the CRAM controller, the CRAM chips, the host processor, and the host system buses, as well as C++ tools to create text-file models of CRAM microinstructions and CRAM/host code to be executed by the system.

When using this model to simulate the CRAM controller, the controller behavioral model is used for the behavioral simulation in Section 5.1.2, the Synopsys-generated gate-level VHDL model is used for the simulation in Section 5.1.3, and the post-layout VHDL model and SDF file are used for the simulation in Section 5.1.4. This use of the same testbench for simulating the controller at different design stages is advantageous because it ensures that the design is subjected to the same test stimuli. This not only reduces the design time, but also removes 'false simulation' errors that result if the design is mistakenly subjected to slightly different test inputs after it has been synthesized.

All the other components in the CRAM system simulation model use simulation-oriented VHDL behavioral models (i.e. some of the VHDL constructs used are unsynthesizable). The use of components behavioral models in a testbench, as opposed to using a more standard and common testbench that simply assigns values to the inputs of the unit under test, allows the running of actual system machine code in testing the design functionality, thus saving time in generating the input test vectors and collecting the design test outputs [57], [58]. The VHDL simulation models of the CRAM chips (generic versions of C64p1k and C512p512) have been designed to exactly match the functionality of the actual designs. Even the bugs in the CRAM prototype chips have been incorporated in some of the models. This, when used with the synthesized model of the controller, makes the CRAM system VHDL model a very close representation of the CRAM system prototypes. Actually, the results of running machine code on this model yields the exact results and sequences as when the same code is run on the prototype system. This makes debugging of the prototype system and controller much easier and faster.

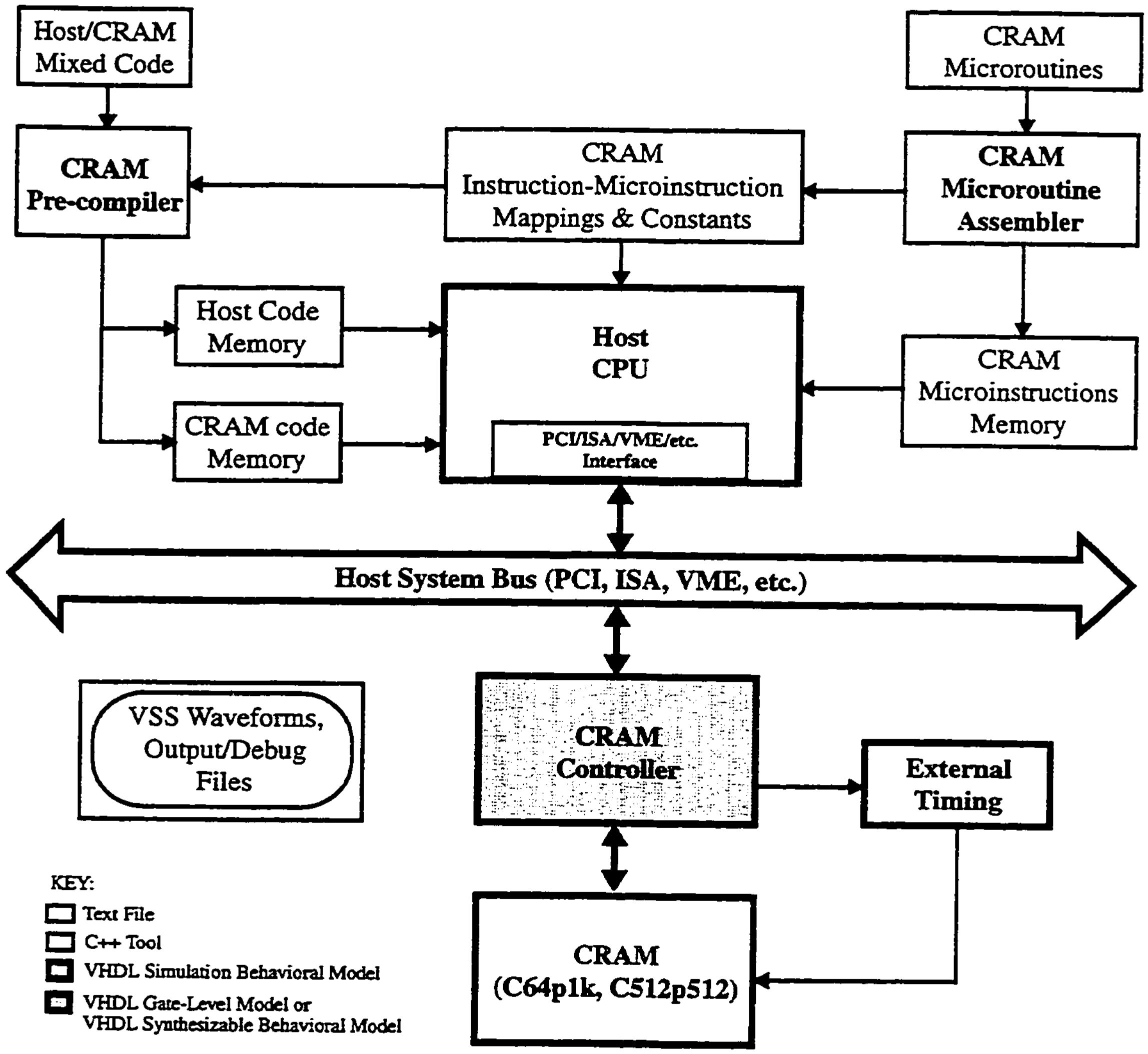

Host/CRAM Mixed Code
CRAM Microroutines
CRAM Pre-compiler
CRAM Instruction-Microinstruction Mappings & Constants
CRAM Microroutine Assembler
Host Code Memory
Host CPU
CRAM Microinstructions Memory
CRAM code Memory
PCI/ISA/VME/etc. Interface
Host System Bus (PCI, ISA, VME, etc.)
VSS Waveforms, Output/Debug Files
CRAM Controller
External Timing
KEY:
Text File
C++ Tool
VHDL Simulation Behavioral Model
VHDL Gate-Level Model or VHDL Synthesizable Behavioral Model
CRAM (C64p1k, C512p512)

Figure 5.2 CRAM System VHDL Simulation Model

The simulation output include a display of the CRAM memory (through the Synopsys VSS waveform viewer) and output files showing results of executing the instructions on the host side of the system bus. Other output files show timing and debug information.

This CRAM system VHDL simulation environment is intended primarily for simulation during the hardware (controller and CRAM chips) design flow. It is also used for extracting more accurate CRAM controller timing information. Small algorithms and applications may also be run on the system. But since VHDL simulation, like all digital simulations, is slow, more lengthy applications or architectural exploration should be carried out using the C++ CRAM simulator described in Section 6.7.

5.2 CRAM Controller Implementation and Testing

5.2.1 Xilinx Implementation

Programmable devices are well suited for prototyping because a design can easily, cheaply, and quickly be implemented in-house without the need for the otherwise expensive and time-consuming IC fabrication process. CPLDs and FPGAs are therefore the backbones of most prototype digital systems because such projects require tighter schedules, output low volume products, and require low non-recurring engineering (NRE) expenses [59]. The choice of Xilinx FPGAs over other programmable devices was influenced by availability of CAD tools and experience with the devices in the department.

The target FPGA for the controller is a 576-CLB XC4013E FPGA. The specific part used in both the ISA and PCI CRAM controller prototypes is a 240-pin, speed-grade -2 part housed in a flat plastic quad package (XC4013EPQ240-2). Table 5.1 shows the FPGA device utilization for the different controller designs. All these designs are based on the simplified architecture described in Section 4.4.3. The ISA design has a control store size of 192 words. The PCI CRAM controller with a control size of 192 words does not fit in an XC4013E part. Therefore a PCI controller prototype with a reduced control store size of 128 words was implemented in the FPGA.

FPGA Resource	No. Available	ISA		PCI†	
		No. Used	% Used	No. Used	% Used
Occupied CLBs	576	564	97	576	100
Bonded I/O Pins	192	91	47	83	43
F and G Function Generators	1152	969	84	986	85
H Function Generators	576	262	45	233	40
CLB Flip Flops	1152	234	20	308	26
IOB Input Flip Flops	192	0	0	0	0
IOB Output Flip Flops	192	0	0	0	0
3-State Buffers	1248	118	9	235	18
3-State Half Longlines	96	40	41	78	81
Edge Decode Inputs	288	0	0	0	0
Edge Decode Half Longlines	32	0	0	0	0
CLB Fast Carry Logic	576	30	5	38	6

†Control Store Size = 128 Words

Table 5.1 CRAM Controller FPGA Device Utilization

The implemented ISA CRAM controller runs at 14 MHz, while the PCI CRAM controller (with a control size of 128 words) runs at 10 MHz. Table 5.2 shows the percentage FPGA utilization (in terms of CLBs and tristate buffers) of each functional block of the three CRAM controllers. Again this is for designs based on the simplified architecture of Figure 4.10 with a 192-word control store. Tristate buffer numbers are tabulated separately because in a Xilinx FPGA these logic gates are not implemented in a CLB. Notice that in each case the control store occupies more than 39% of the total area, and that the PCI interface unit uses almost six times more resources than the ISA interface.

Functional Block	CLB'S				TRISTATE BUFFERS			
	No Used		% of Total		No Used		% of Total	
	ISA	PCI	ISA	PCI	ISA	PCI	ISA	PCI
CRAM-Host Interface	17	99	3	15	36	137	24	50
Instruction Queue Unit	58	57	10	9	0	0	0	0
Microprogram Sequencer	43	42	8	6	40	40	27	14
Control Store	256	256	46	39	0	0	0	0
Buffers & Constant Unit	73	95	13	14	16	32	11	12
Registers, Address Unit, and Other Logic	108	109	20	17	56	65	38	24
Total	**555**	**658**	**100**	**100**	**148**	**274**	**100**	**100**

Table 5.2 CRAM Controller Functional Blocks FPGA Utilization

5.2.2 ASIC Simulation

Since this is the first work on the design and implementation of a CRAM system, there has been, and will continue to be a lot of iterations in arriving at an optimum architecture for the CRAM controller as well as the CRAM system software tools. For this reason, the implementation of a CRAM controller as an ASIC was not within the scope of this thesis. Otherwise, the main focus has been to refine the architecture by using programmable devices to build and test prototypes, and use simulation tools (developed in this work) to explore different architectural features. However, because of the universality of the design entry method (the same VHDL can be used for implementing the controller in an ASIC technology), the design of the controller using CMC-supported ASIC technologies has also been performed. The main objective of this exercise was to get a more accurate picture of the controller characteristics, especially speed and area. The technologies used are the Nortel 0.8 μm BiCMOS technology (BATMOS), Hewlett Packard 0.5 μm CMOS technology (CMOSIS5), and TSMC 0.35 μm CMOS technology (CMOSp35).

Table 5.3 shows the area of the controller using CMOSp35. The area is given in 2-input NAND gate equivalents so that it can be used to approximate the area of the controller in other technologies. For both the PCI and ISA, the control store occupies a substantial percentage of the total area of the CRAM controller. Another important thing to note is the area of the buffers. As shown in Figure 4.14, buffers are designed using small RAM blocks. In an ASIC technology, the area per bit of an SRAM core increases as the size of the core decreases. For example, a 256 x 32 SRAM core has 256 times the

Functional Block	**PCI**		**ISA**	
	†wand2 equiv. gates	**% of Total Area**	**†wand2 equiv. gates**	**% of Total Area**
CRAM-Host Interface	1184.2	9.6	105.0	0.9
Instruction Queue Unit	880.5	7.2	996.3	8.2
Microprogram Sequencer	663.3	5.4	665.5	5.5
Control Store	3898.7	31.7	6782.8	56.2
Buffers & Constant Unit	4458.5	36.3	2349.4	19.5
Registers, Address Unit, etc.	1201.4	9.8	1174.6	9.7
Total	**12281.6**	**100**	**12073.6**	**100**

†wand2 is a 2-input NAND gate.

Table 5.3 Area of CRAM Controller in TSMC 0.35 μm Technology

memory capacity of an 8 x 4 core, but its area in CMOSp35 (3893.7 equivalent gates) is only 34 times bigger. This makes the area of the buffers much bigger, especially since the buffers have to be designed using 8 x 4 SRAM cores because of the absence of 16 x 8 or 8 x 8 cores. This is also the reason why the ISA instruction queue unit is bigger than that of the PCI CRAM controller (two 16 x 16 cores have a total area of 562.4 gates, where as one 16 x 32 core has an area of 449.7 gates), and why the ISA control unit, which uses four 256 x 8 SRAM cores, has a bigger area than the 256 x 32 SRAM core used for the PCI.

The speed of the controller in CMOSp35 is well above the projected maximum CRAM speed of 50 MHz (20 ns cycle time). The cycle time is limited by the 11 ns (90 MHz) clock period of the SRAM cores. The maximum propagation delay in the other paths (without optimization) is 6.6 ns.

5.2.3 CRAM Controller Prototype Testing

Figure 5.3 shows the configuration used for testing CRAM controller prototypes. It consists of a CRAM controller board, a CRAM PCB, a Hewlett Parckard Logic Analysis System, a power supply, and a clock generator. The controller board is comprised of a Xilinx 240-pin XC4013E FPGA, an AMD27C512 512-bit EPROM to hold the FPGA configuration bitstream, and a MACH210A CPLD that controls the downloading of the controller configuration from the EPROM onto the FPGA. The CPLD implements a

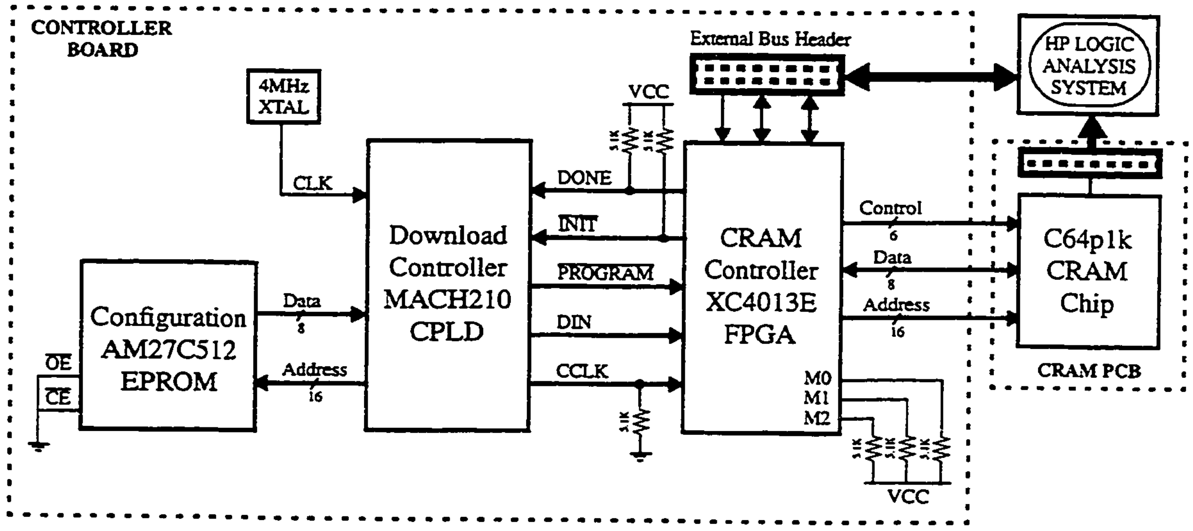


Figure 5.3 CRAM Controller Test Fixture

simple FSM that allows the use of standard byte-wide EPROMS instead of serial PROMS (which are more expensive and less widely available) during configuration. The CPLD logic was designed using the PALASM software. Details of Xilinx FPGA configuration can be found in the Xilinx devices data book [60]. The External Bus Header is a collection of test-point pins that are connected to the FPGA and represent the host bus signals. The number of these pins is big enough so that all the signals of the buses we are contemplating of using at some point (PCI, ISA, EISA, VME) can be accommodated. This makes the board a generic test fixture. All the main components of the controller board (FPGA, CPLD, and EPROM) are housed on sockets. The CRAM PCB consists of one socket for an 84-pin C64p1k CRAM chip and a few headers to allow tracing and driving of all the signals of the C64p1k. The first test fixture used a wire-wrapped controller board. But later, a CRAM system PCB comprising of the controller hardware and one socket for the C64p1k chip was fabricated. The layout of this latter board is shown in Appendix B. This is the board that is used in building the CRAM system prototype described in Section 5.3.

To test the controller, patterns of data are generated from the HP system to mimic specific host processor system bus cycles. This philosophy simplifies the testing because it mirrors exactly how the controller is used, connected, and tested in both the C++ and VHDL simulation models as well as the real CRAM system. Because of the absence of the C/C++ interface in this test fixture, CRAM instructions from the host processor (the HP pattern generator) are written in machine code. This machine code can be copied directly from the output of the CRAM Pre-compiler used in the VHDL simulation model (Figure 5.2), or from the debug output of the CRAM C++ Simulator. Both the host bus and CRAM bus signals are traced on the logic analyzer. Because of the simplicity of generating the test vectors and observing the test output, this test fixture has also been used to exhaustively test (memory and PE functionality) the C64p1k chips, something that was almost impossible to do with the chips tested in isolation.