Control and Application of a Pixel-Parallel Image Processing System

by

Gary Hall

B.S. Electrical Engineering
University of Florida, 1994

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

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February 1997

Signature of Author _____
Department of Electrical Engineering and Computer Science
December 4, 1996

Certified by _____
Prof. Charles G. Sodini
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by _____
Prof. Arthur C. Smith, Chairman
Department Committee on Graduate Students

Control and Application of a Pixel-Parallel Image Processing System

by

Gary Hall

Submitted to the
Department of Electrical Engineering and Computer Science

December 1997

in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis deals with the design and implementation of a control path for a Pixel-Parallel Image Processing system. This system uses high-density parallel processors (HDPPs) with a processor-per-pixel architecture to perform efficient manipulations of data. To do this, it employs SIMD (single instruction, multiple data) techniques well suited towards many image-processing algorithms. This system has been designed with low cost as a primary goal, such that it may be used in a consumer market for which typical parallel-processing systems are prohibitively expensive.

The control path must be capable of delivering instructions to the processor array at the highest speed the array is capable of supporting. It must also be able to interface with an inexpensive workstation as the host computer and human interface. These two requirements necessitated some form of instruction expansion from the host-controller interface to the controller-array interface, in order to achieve the bandwidth amplification called for.

Previous work has been done on a control path for a similar system, using a content-addressable parallel processor (CAPP) to build the processor array. This controller was designed for 10 MHz operation, which would suffice to support the CAPP array. However, new research has yielded the HDPP array, which uses more dense design and clock rates of up to 20 MHz. A new control path has been constructed to support the increased instruction rate requirements of this processor, as well as address several issues remaining with the previous control path design. In addition, further work has been done to generate useful image processing primitives, which have been designed for use in more complex image processing applications.

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

Acknowledgments

First, I'd like to thank Charlie Sodini, for the opportunity to work with him and his students, and for having good advice about the real world. The time I've been in the group has been a great combination of school and practical work experience, and I'm glad to have been a part of it.

I also owe a large debt to Jeff Gealow, who I worked closely with during the course of this research. While it's true that, on occasion, I did him some favors and answered a few questions, I'm quite sure I'm very far from catching up to the number of times he's helped me. I appreciate his patience in dealing both with tricky hardware questions and stupid UNIX questions, which occurred with roughly equal frequency.

Then there's the tons of fellow researchers who I have enjoyed spending time with over the last year or so. Joe Lutsky, Jen Lloyd, Steve Decker, and Jeff have provided endless hours of amusing conversation over Whoppers™ and fries. There have also been many evening dinners with them, Andy Karanicolas, and Dan Sobek, where I learned lots of interesting tidbits about industry, and discovered the joy of pan-fried ravs. To Mike Perrott, I owe debt for most of the exercise I managed to get at MIT, and earned a healthy respect for road bikes in a head wind. And to bring back a blast from the past, thanks to Daphne Shih, for showing me the ropes at MIT, and being the first good friend I had here.

Thanks to Iliana Fujimori, her brother Marco, MeeLan Lee, Howard Davis, and Tony Cate for getting me out of my dorm room every once in a while, and showing me most of what I saw of the social scene in Boston. I had fun every time we went out, and I'll miss that part of Boston a lot.

I'd also like to thank some friends at Ashdown, for going hiking, eating dinner, wasting time watching TV, and even studying once in a while. I can't list them all here, but thanks to Laura Johnson, Tom Lee, Pat Walton, Rebecca Xiong, Kathy Liu, and John Matz, who were the main culprits in forcing me out of my room to have a life.

I'd like to thank my bosses, mentors, and friends at Motorola for making this return to grad school possible. To Neil Eastman and Jerry Johnson I owe special thanks. I'll always appreciate their hard work and special attention that got me here, and I hope that in the future I can repay both of them in some way. I also thank Rae Zhang, Phil Teeling, Dave Melson, Hsi-Jen Chao, and Sandhya Chandarlapaty for guidance and advice when I worked with them. I also want to include Jason Robertson, Jerry Hsiao, Harry Chow, Tod Hufferd, Greg Coonley, and Mike Weatherwax, for being about the coolest group of people to hang out with after work that ever was, and keeping me going through grad school even though they knew they were having more fun.

Finally, there's Mom and Dad, Virginia and T. M. "Mac" Hall. It was while at grad school that I

had the epiphany that happens to everyone at some point, when I realized how smart they were, and how ignorant I was, in many facets of life. They got me started along the right path, helped me keep going when I was ready to quit, and gave me just the right amount of support along the way. If I can remember half of what they taught me, and apply that half as well as they did, I'll consider myself a success in all ways that matter.

Table of Contents

Chapter 4

Software Application Framework 55

Chapter 5

Testing and Results 69

Chapter 6

Conclusions 77

Appendix A

Test Points / Jumper Positions 81

Appendix B

Board Documentation 83

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Image processing in real time presents many difficulties for most general-purpose computers. The most serious of these is the well-known von Neumann bottleneck, which requires the processor to individually access each data element (usually a pixel in this context). Doing image processing with this type of machine would require an exceptionally fast general-purpose computer, as even a modest-sized image of 256 x 256 has 65,536 pixels which would all be processed separately.

Many image processing tasks today can be implemented to take advantage of *data parallelism,* which is a method of assigning one processor to each data element and performing all operations on the data in parallel [1]. This is known as single-instruction-multiple-data (SIMD) processing. This method has given rise to several forms of parallel processors, which have a much higher processor-to-memory ratio than general-purpose computers. The Connection Machine [1] is an example, with up to 64K processing elements (PEs), with each PE having up to 64K bits of memory (in the CM2). The fully configured CM2 requires four front-end host machines, as well as a *data vault,* to house the processor memory and provide data I/O.

However, this type of machine is too expensive for many common applications in industry. The expense of the processor itself is significant, and the overall system expense is such that it would be impossible to produce a reasonable consumer application containing this technology. This has given rise to new research on inexpensive systems that still take advantage of data parallelism, while having a simpler interface, more integration, and maintaining real-time performance.

Researchers at MIT have developed a system to meet this goal, using processors with a 16 x 16 internal PE array and content-addressable memory [2]. This system demonstrated efficient image processing with parallel-oriented tasks, but did not include a complete real-time image I/O mechanism. Further research is currently being conducted on a high-density parallel processor (HDPP) which contains 64 x 64 PEs on chip [2]. There is also a corresponding system design to allow image I/O at frame rate (30 frames/sec), as well as a simple programming interface. Figure 1-1 shows a block diagram of the system. A new architecture for the Control Path in this system is the subject of this thesis. Software applications demonstrating some capabilities of the system to perform useful image processing in real time are also shown.

1.2 Controlling a Real-Time Image Processing System

There were several challenges in creating a controller for this type of system. First, it had to be

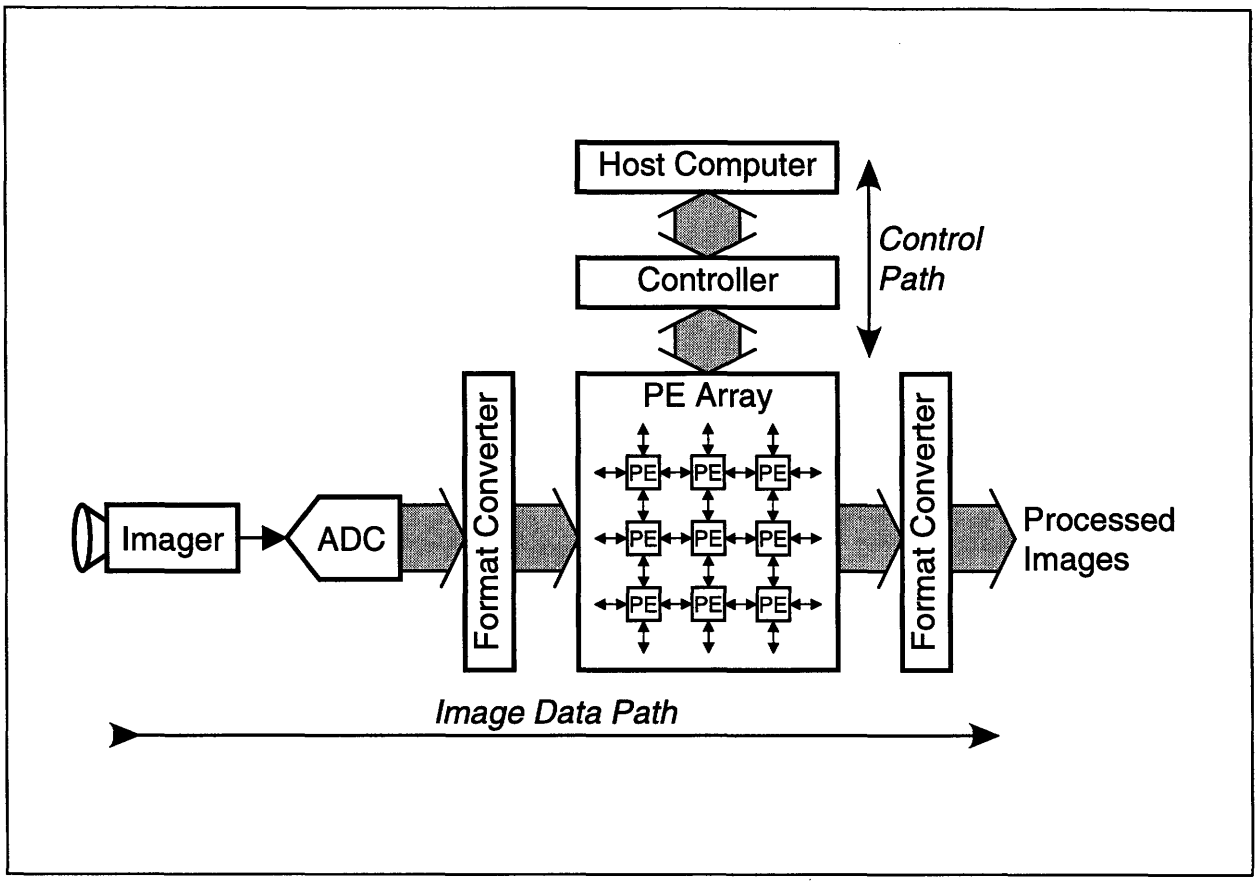

Figure 1-1: Block Diagram of Image Processing System

fast enough to place the bottleneck on the processors themselves, not on the control path. As most of the design time and expense on this type of project was devoted to these processors, one would prefer to get the most out of the processing power they present. Second, for a project of this scope, the controller had to be kept to a low expense. For a system to be commercially viable, it cannot be in the high price range of typical research systems for image processing in real time. Finally, it had to be small enough to be practical in the real world. A typically successful recipe in going from research to practice is smaller, faster, cheaper; though for this application, machines such as the CM2 and successors are so fast that a practical system can be much slower.

The primary function of this controller was to deliver sequences of array instructions to the processors fast enough to keep them busy as much as possible. In addition, certain other functions of the system fell under the controller's responsibility, such as any memory refreshes of the processors' internal DRAM, supervising and synchronizing image I/O between the array and the datapath hardware, and providing an interface between the system itself and the top-level application programmer.

One part of this research was to complete the next evolution of a controller for use in a pixel-parallel image processing system. The previous controller, designed by Hsu [3], was used in an image processing system employing the aforementioned CAPP array. The controller resides in the system built by Hermann and Gealow [2] around their image processing arrays. This system

includes both the hardware shown in Figure 1-1, as well as an extensive amount of software support. This software consists of simulators, to build and test application code on the system, and hardware interface code, to allow programmers to run image processing tasks on the system without a detailed knowledge of it's internal architecture.

There were three primary requirements to be met by this new controller, to be discussed in more detail in later sections:

- Decrease board size - the controller must fit completely inside a standard double-height VME chassis
- Increase system speed from 10 MHz to a target of 20 MHz - the new array processors will have a higher clock rate than their content-addressable counterparts
- Increase controller robustness - the previous controller exhibited intermittent problems at various clock frequencies, including the target of 10 MHz

Due to the large amount of software support for the previous controller's architectural model, it was advantageous to remain as similar as possible to that model, while still meeting the new requirements. This thesis documents the similarities and the differences between the two controller implementations.

In addition, a part of this research demonstrates the viability of this system, with some basic image-processing tasks. Several of these programs have already been written, to do tasks such as smoothing and segmentation, median filtering, and optical flow, on static images. This research extended these operations to real-time images, and produced new operations such as edge detection and primitive template matching.

1.3 Organization of Thesis

This chapter outlined the primary goals of this research, with some background on parallel processing methodologies. It introduced the concepts of SIMD processing and parallelism, and presented an overall system for parallel image processing.

Chapter 2 discusses the architectural model of the new controller design. It shows the evolution in controller design for this type of project, and discusses the similarities and differences in hardware implementation between this and the previous controller.

The decisions made in meeting the new requirements are discussed in Chapter 3. It includes an in-depth analysis of the critical paths of both the old and the new controller, outlining problems that were fixed with this design. It also presents some of the logic that went into the decision of what strategy to use for new logic integration.

Chapter 4 deals with the software application framework. It describes the interface provided to programmers creating applications on this system. It includes some tips on efficient use of resources in applications, and details some new applications written as part of this thesis.

The testing of the controller and results are described in Chapter 5. This chapter shows some of the pitfalls encountered with this design, some problems in board fabrication, and workarounds. It also includes the quantitative results in execution speed and a comparison to the previous controller.

The final chapter of this thesis is the conclusions from the work done, and a short description of where future work might lead. Following that are several appendices provided for reference on details such as board test points and organization of files used in this thesis (source code, board schematics, etc.). This includes pointers to documentation on that source code and details of the controller hardware.

Chapter 2

Controller Architecture

As mentioned in the introduction, one of the attributes of this controller is a design that is much simpler, and whose cost is much lower, than the sophisticated controllers used in expensive modern parallel-processing machines. This goal was accomplished by Hsu [3] in a previous thesis. This new design has other goals to be discussed later, but also sought to take advantage of earlier work done in this area.

To this end, the controller built for this system was designed to be software compatible with the previous design done by Hsu. The programming framework that supports this controller, and the system in general, simulates both bodies in software, and provides a layer of abstraction to the hardware. There is an obvious advantage to staying software compatible at the top-level controller model, so that application and high-level interface code may remain unchanged. Along these same lines, the closer the hardware implementation remained to the original, the less modification was needed to the low-level interface code in the framework. However, some changes were required to meet the new goals of the controller design. Therefore, there were some decisions to be made between this hardware/software trade-off. Some of these decisions will be discussed here, and some in the next chapter.

This chapter outlines the main architectural details put forth by Hsu in a previous image processing system, which are in common with the new controller. It also discusses changes made in this architecture, for the new controller design.

2.1 Previous Work

2.1.1 Controller Strategies

The controller model used in this design has evolved from work done on previous iterations of this research. A discussion of these previous implementations will help clarify the choices made behind the new controller model. This section summarizes a discussion by Hsu [3] on several controller strategies investigated before choosing a new model.

Figure 2-1 shows a simple control strategy used in the first system constructed by the Associative Processing Project (APP) (predecessor to this research). This system was constructed around a Database Accelerator chip (DBA), and the system was termed System I [4]. The control model used in this system was to have the host computer act as the controller as well, generating low-level instructions to be sent directly to the PE array. This model offers one obvious benefit, in that the "controller" is built almost entirely in software, with the only exception being the interface between the host's bus and the array. There were, however, two key limitations of this approach.

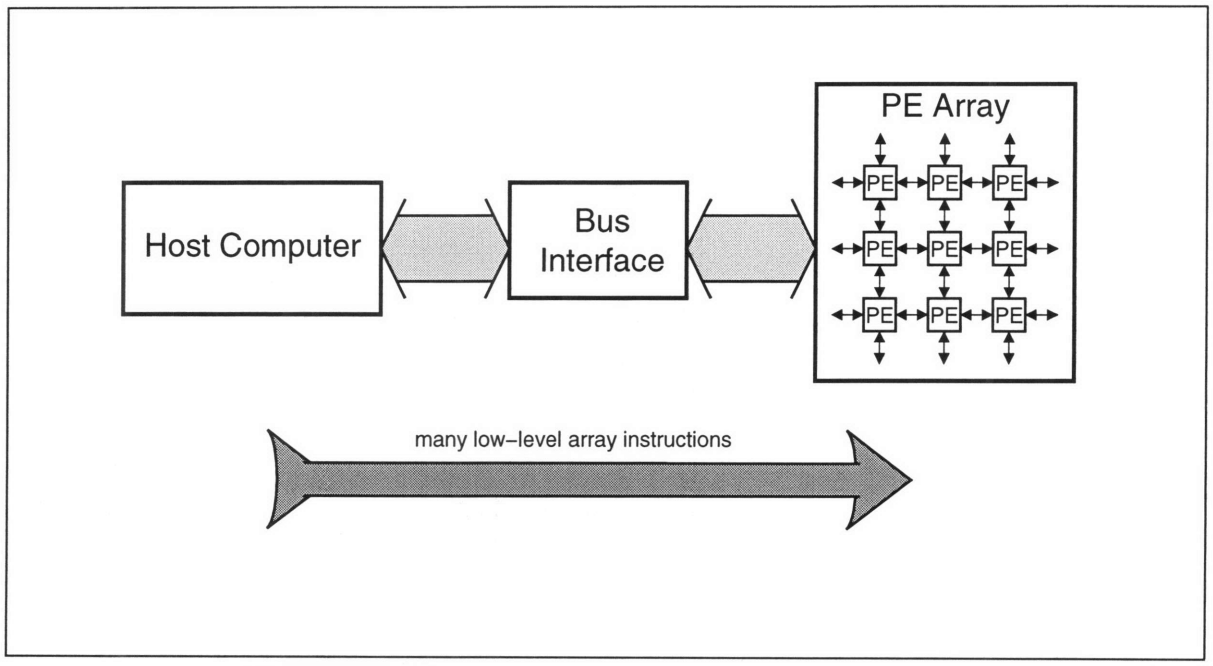

Figure 2-1: Direct Control From Host (System I DBA)

The first was that the instructions were generated at run time, which required the host computer to generate instructions as fast as the array could process them. The second limitation was that the system was slowed by the rate at which the instructions could be delivered to the array, through the bus interface. These problems combined to cause the system's performance to be degraded by two orders of magnitude below its potential [5].

The second approach was to use a more complex controller. This model, shown in Figure 2-2, has the host issuing *macroinstructions* to the controller. The controller then decodes these instruc-

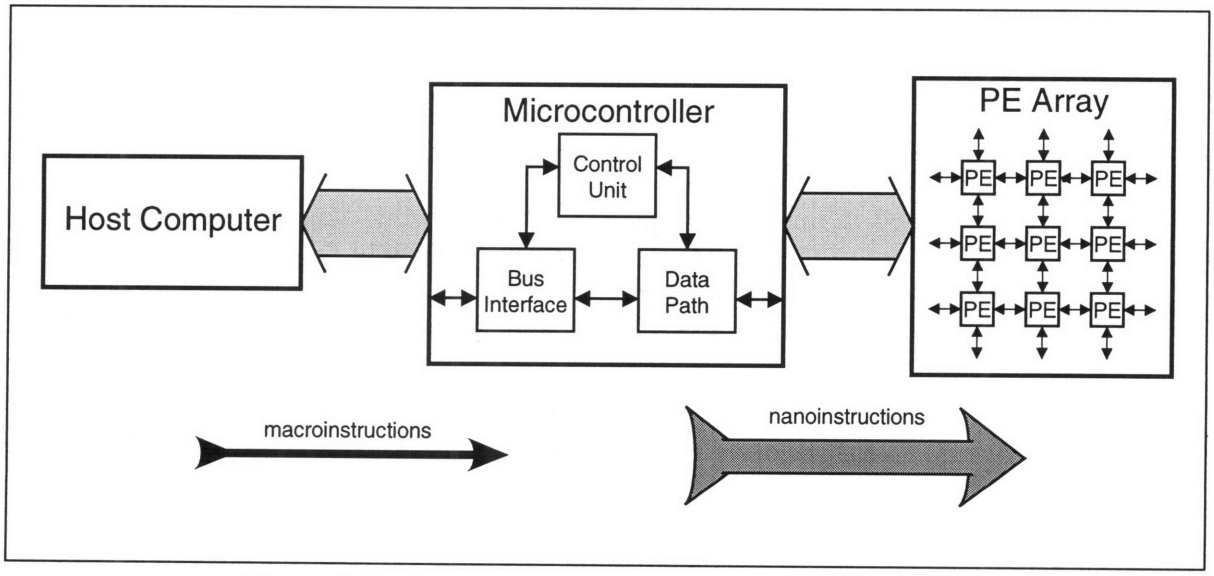

Figure 2-2: Complex Microcontroller (System II DBA)

tions and expands them into *nanoinstructions* which are sent to the PE array. In interpreting the microinstructions to produce the correct instructions for the PE array, the controller itself locally executes *microinstructions*. This approach eliminates both of the previous drawbacks, in that the host computer doesn't need to generate many instructions at run time, and the required bus bandwidth is reduced. This was the method used to build a controller for the second DBA system, System II [5]. System II's controller consisted of a microprogram control store and sequencer, and it's datapath for delivering instructions contained register files, a 32-bit arithmetic logic unit (ALU), a 32-trit[1] ternary logic unit (TLU), and control and interface logic. This type of architecture was also used in the Connection Machine [1].

However, this approach was not without its own shortcomings. In this model, the controller is a microprogrammed machine. It stores array instructions in a writable control store, allowing different array architectures to be used, with arbitrary instruction widths and formats. However, to be able to generate these instructions at a sufficient rate, the controller must be very fast and complex. System II's data path needed expensive and sophisticated components, the 32-bit ALU and 32-trit TLU, in order to generate the instructions fast enough to avoid the DBA chips becoming idle. This approach also greatly complicates application development for the system. In addition to writing code for the application itself, the microprograms on the controller must also be developed.

A new, simpler controller model was set forth by Gealow et. al. [2], which still draws upon the idea of amplifying the bus bandwidth using a form of instruction expansion. However, this new model focuses more on a simple, inexpensive implementation that can still meet the speed requirements of the processors being used in this research. One of the new ideas for this model was generating the sequences of array instructions at compile time, instead of run time. This reduces the need for a datapath in the controller, since the instructions are not generated on the fly. This new system architecture is shown in Figure 2-3. In this new architecture, the sequences of instructions (similar to procedure definitions) are stored in the controller. The host computer then simply calls these sequences at run-time, much like calling procedures. As noted by Gealow et. al., this type of system poses two problems for this system's designer:

- *Run-time scalar data.* Any sequences that depend on the use of data generated at run time (such as variables) would, in the simplest implementation, require the data to be compiled into the instruction stream. This would necessitate instruction generation on-the-fly.
- *Run-time flow control.* If the flow of program execution depends on results generated at run time, the instructions cannot be generated at compile time.

In a general sense, these problems would be very difficult to overcome. However, the fact that these processors are implemented in a bit-serial fashion gives some unique possibilities to circumvent the problem.

1. A *trit* is a ternary digit that can assume the logical values 0, 1, and don't care (X).

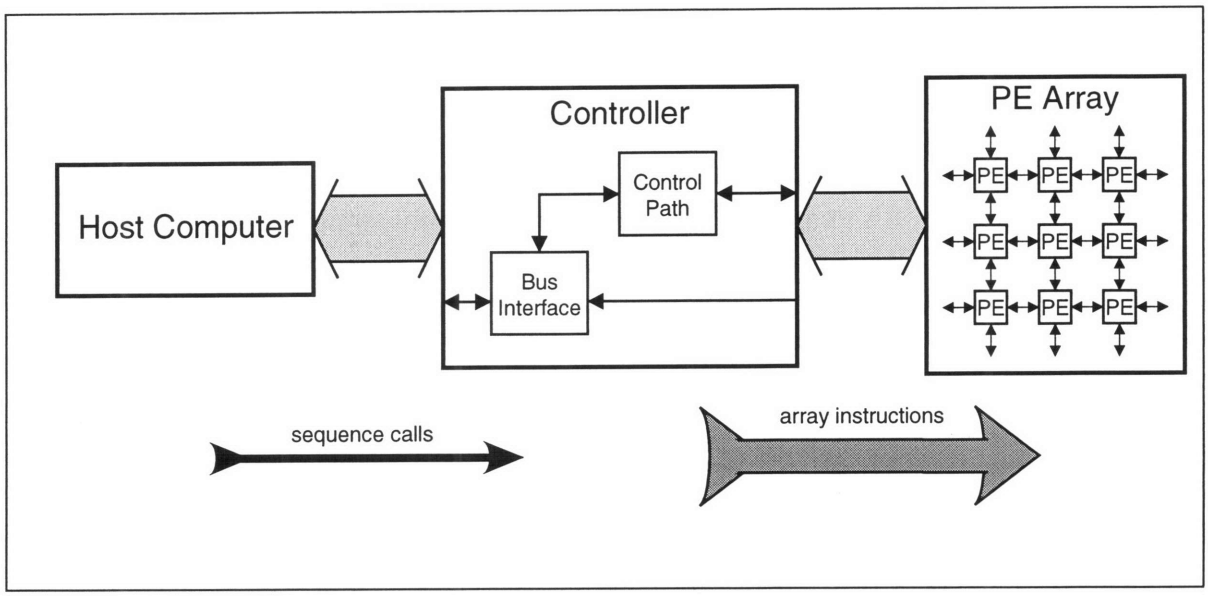

Figure 2-3: Current Controller Strategy

In the important case of bit-serial arithmetic, a solution to the first issue is *instruction selection.* Consider the case of adding a run-time scalar to an *n*-bit value stored in the array. One simple solution would be to generate a sequence of instructions for all 2^n possible scalar values, but clearly this is not acceptable for any significant value of n. However, most bit-serial algorithms can be broken down into pairs of instructions, one pair per bit. A simple case of this is shown in Figure 2-4, for a four-bit addition. The example shown is adding the binary value 1101. The sequence for adding 1 is stored in one portion of the control store, and another sequence for adding 0 is stored in another portion. At run-time, the scalar value is loaded into a shift register on the controller. This register then controls which instruction is sent to the array. The value is loaded in the shift register, and appropriate instructions are sent to the array. Practically speaking, the hardware can be further optimized, since a MUX of the width of the control store (32 bits for this controller) is impractical, but this diagram shows the model used.

The problem of run-time flow control is more complicated. In previous processors, which were associative, a mechanism called *responder feedback* was used to solve this problem [2]. Each processing element would provide a one-bit input to a response resolver. This resolver would perform a logical OR operation on its inputs, allowing Some/None operations to be done, which is a common feature of associative processors. One example of the usefulness of this mechanism to remove the need for run-time flow control is a maximum value computation. Without the response resolver, a program would have to traverse a binary tree of maximum values. For instance, it would first check for PEs with 1XXX. If none were found, it would check for 01XX; otherwise, it would check for 11XX. This would continue, and this program's flow is clearly dependent on the results of run-time generated data.

With the response resolver, this dependent flow is not necessary. First, all processing elements are active. A match is performed on 1XXX. Processing elements that match this value are left activated, while the rest are turned off. The response resolver is used to determine if *any* PEs

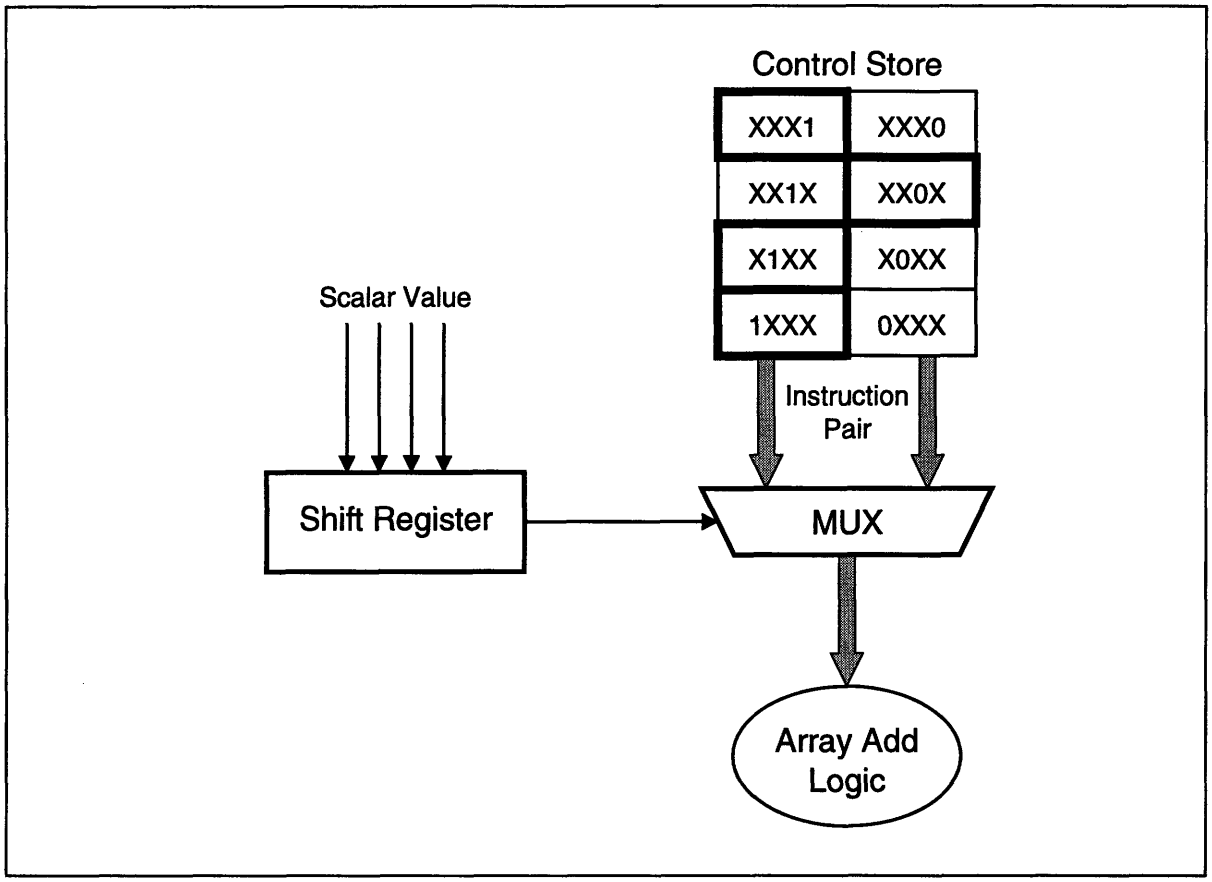

Figure 2-4: Instruction Selection

matched -- if not, then the previously active PEs remain active. The output of the response resolver is then shifted into a serial output register that will contain the maximum value. Successive matches are then performed on X1XX, XX1X, and XXX1. The PEs still active contain the maximum value, and the output register contains this value.

However, in the most recent version of the parallel processor being used with this controller, the response resolver is not present. Hardware exists to activate certain PEs based on various criteria, but there is no Some/None feature present to determine if any PEs are currently active. This means that run-time flow control for some algorithms is not possible, or not as efficient, as with architectures supporting the response resolver. However, as we will see in Section 4.4.2, for some important algorithms, efficient software solutions can be created to get around this problem.

2.1.2 Controller Model

The final model for the controller implementation is shown in Figure 2-5 [2]. This figure shows the various components that were part of the previous controller, and kept in this implementation for software compatibility. This model is that seen by the software framework, but in this controller, certain parts of the model are merely a layer of abstraction.

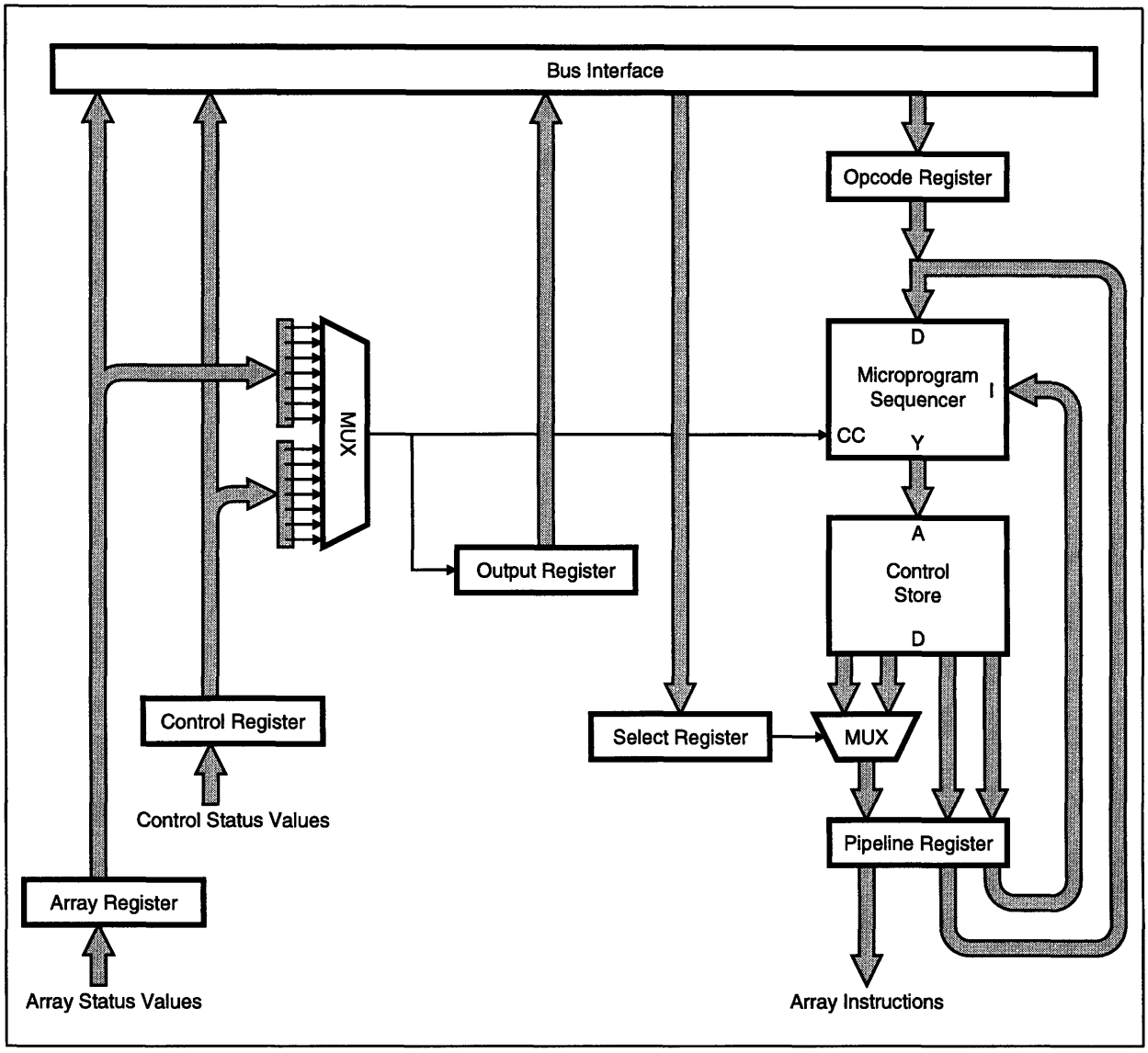

Figure 2-5: Controller Implementation Model

2.2 Hardware Implementation

2.2.1 VME Bus Interface

The details of the bus interface are largely unchanged from the previous implementation of this controller. Due to this fact, and the fact that bus interface is a fairly common and simple part of any system connected to a proprietary bus, the reader is referred to previous work for many of the details of this portion of the design [6], [3].

In this system, the host SPARCstation, along with the PT-SBS915 (the VME chassis used in this system) act as the VMEbus *master.* The controller board is the VMEbus *slave.* Because the controller has no need of external interrupts or block data transactions, the only transfers supported is read and write. Specifically, the bus interface is designed to support only 32-bit transfers, in

either user or supervisor mode.

For choice of address space, attention was paid to the other two VMEbus slaves designed as part of this research: the previous controller board, and the *data path* board [7] shown in Figure 1-1. Each of these boards occupy some of the available 32 bits of address space, ideally a few addresses each. The previous controller decoded eight bits of the VME address for comparison to a base address, occupying the addresses of FXXXFX00-FXXXFX1F. This design uses the same base address, to reduce the software changes to the VMEbus driver for the controller. The datapath board occupies address space XXXXAX00-XXXXAX1F (decoding only four bits for the base address).

Figure 2-6 shows the VMEbus interface implemented in all of the slave modules used in this research project to date. The PLX2000 chip interfaces directly with the VMEbus, signalling the

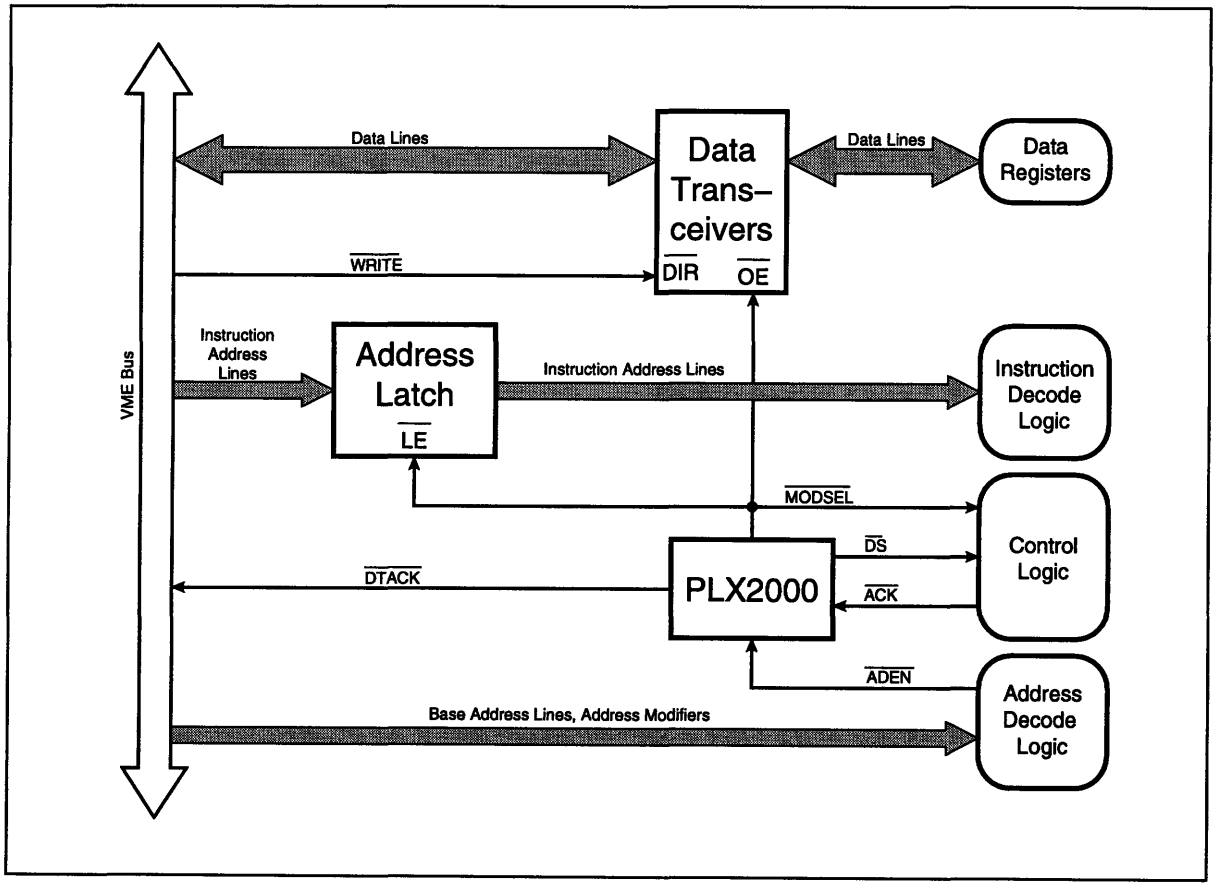

Figure 2-6: VMEbus Interface

start of a data transfer with $\overline{\text{MODSEL}}$ (module select) and its completion with $\overline{\text{DTACK}}$ (data transfer acknowledge). $\overline{\text{DS}}$ (data strobe) indicates valid data is present on the bus data lines. It uses $\overline{\text{ADEN}}$ (address enable), a signal asserted by the slave module when the address lines and address modifiers indicate a bus transfer for this module.

Instruction decoding logic, along with all of the other logic in this design, is integrated into Xilinx

EPLDs (erasable programmable logic devices). The reasoning behind this is discussed in depth in Section 3.1.2. Address decoding and the other interface signals to the PLX2000 were also done in these EPLDs.

2.2.2 Control Path

The control path is responsible for accepting control instructions from the host, and delivering array instructions to the array. Figure 2-7 shows the components of this path. The host computer

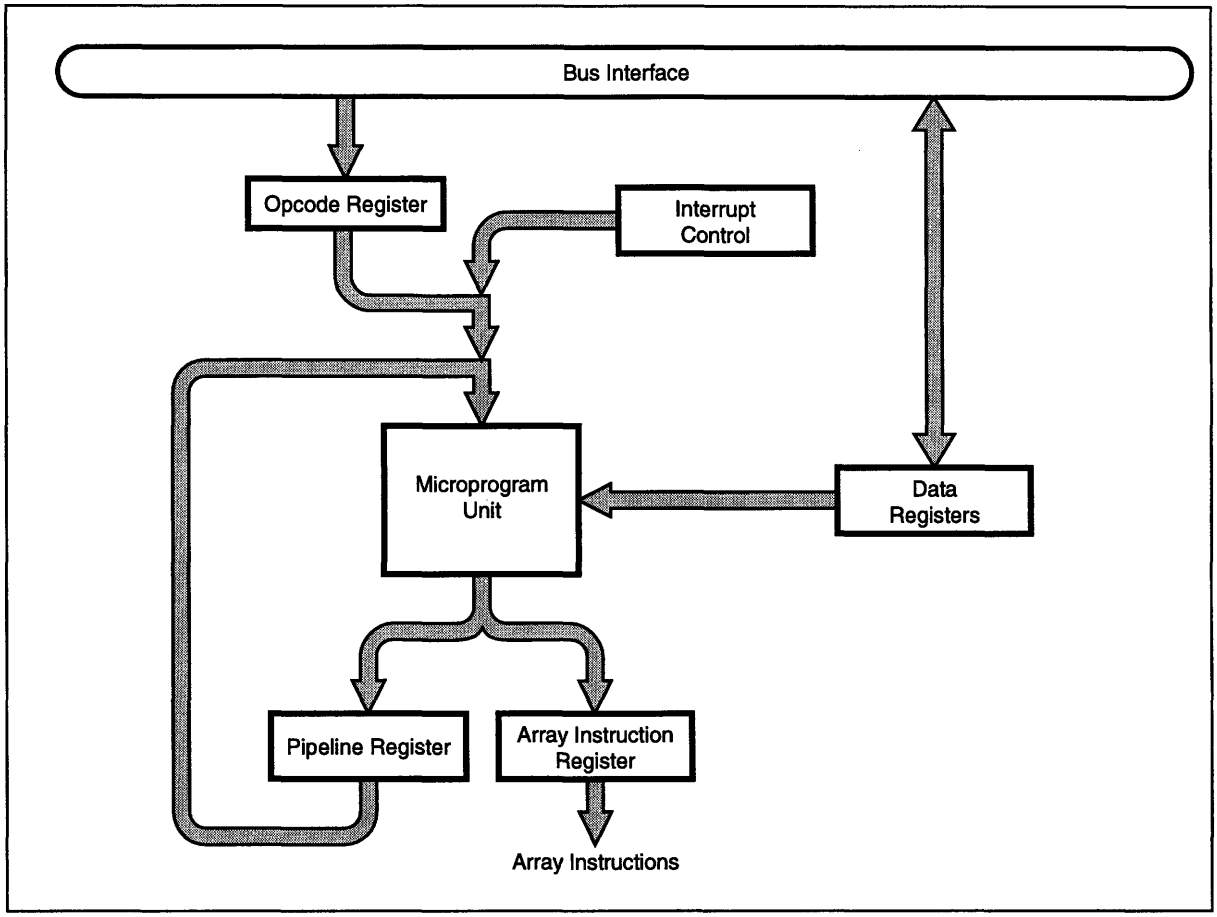

Figure 2-7: Control Path

writes the address of a sequence to be launched into the Opcode Register (OR). The Microprogram Unit is the most important part of this path, containing the Control Store (holding the instructions for the array and the controller), and the Microprogram Sequencer, which controls program flow on the controller. The data registers allow programming of the microprogram unit and interrupt control, and also serve as a method of returning controller status, array status, and other array output to the host computer.

The Interrupt Control mechanism here differs significantly from that of the previous controller. In that controller, interrupt control was a very complicated part of the architecture, supporting multiple priority interrupts, nested interrupts, and masked interrupts. In this implementation, it was decided to use the interrupts explicitly for refreshing the DRAM in the array processors. This was

due to the fact that the interrupt control was not used in the previous controller, never having been tested, and therefore memory refresh was the job of the programmer, who had to explicitly place a memory refresh sequence as part of the application.

2.2.3 Data Registers

As mentioned, one of the goals in implementing this controller was to stay software-compatible with the previous controller to as large an extent as possible. One way of achieving this goal was to make the data registers the same in both controllers, which makes the input/output interface of both controllers the same. For the sake of completeness, the description of the data registers is included as part of this thesis.

The Controller Status Register (CSR) is a read-only register containing various status bits from the controller. The host computer reads this register to determine what state the controller is in, and determine what operations it is ready to handle. Table 2-1 shows the make-up of the CSR.

reserved	*reserved*	$\overline{\text{REFRESH}}$	WMEMBUSY	DIBUSY	ALLBUSY	APBUSY	GO
bit 7	6	5	4	3	2	1	0

Table 2-1: Controller Status Register Bits

The reserved bits are currently tied to ground. The meaning of the bits are as follows:

$\overline{\text{REFRESH}}$ signifies that the controller is currently performing a memory refresh operation. For the most part, this bit is used by the controller itself, to control the interrupt logic and the memory programming logic. However, this bit is available for the host to read. This is the only bit in the CSR that is active low (that is, this bit is 0 if a refresh is currently being performed). Note that, due to the method for programming the control store (explained later), it is unsafe to attempt a memory write while a refresh is occurring. But in order to stay software compatible with the previous controller (which did not have this status bit), and to avoid race conditions where a refresh occurs between a CSR read and a memory write, this problem is resolved in hardware.

WMEMBUSY (write memory busy) shows the status of memory writes in the controller's Control Store. This bit is 1 if a memory write to the Controller Instruction Memory (CIM) or Array Instruction Memory (AIM) is currently pending or active. The host should read this bit before attempting another memory write -- this checking is done in the software framework, transparent to the application programmer.

DIBUSY (direct instruction busy) contains status pertaining to Direct Instruction operations on the controller. If the host has sent an array instruction directly to the array, bypassing the Control Store, this bit stays active until the instruction has been successfully delivered. The framework reads this bit before attempting another direct instruction, or a memory write.

ALLBUSY signals the host that the controller is currently executing a sequence. This bit is set

automatically upon the launch of a sequence, and cleared by the software framework at the completion of each sequence. This bit is read before attempting a memory write.

APBUSY (associative processor busy) is used when, during a sequence, the array is placing a value in the Output Shift Register (OSR). This bit is high until the value is valid to be read by the controller. The name is an artifact of the previous system, which used associative processors. This bit is set at the start of each sequence, and cleared by the software framework when the output is valid. It is also checked before attempting a memory write.

GO signals the host that a sequence is currently pending. This will occur if one sequence is running, and the host launches another sequence. This sequence will be saved in the Opcode Register until the previous sequence completes. The framework checks this bit before launching a sequence.

The Array Status Register (ASR), a read-only 8-bit register, contains status bits from the array, used by the controller in special situations. This mechanism provides for some limited run-time flow control with sequences created at compile time, limited in that the array can only provide certain status bits in the ASR. One use of this register is to forward status signals from the data path board to the controller, which controls the handshaking during image I/O.

The Select Shift Register (SSR) is the write-only 32-bit register used to perform the instruction selection mechanism shown in Figure 2-4 on page 23. The input operand is loaded in parallel into the register, and the LSB of this register is used to select an instruction from part of the AIM (either AIM0 or AIM1, depending on the bit value).

The Output Shift Register (OSR) is the read-only 32-bit register that allows the host to read various data outputs from the array. This shift register's input is tied to a 16:1 multiplexer, which also serves as the input to the sequencer's Condition Code (CC) bit (to control conditional branches). This register is generally written by array assembly code, transparent to the application programmer. The programmer would simply issue the sequence, and then read the OSR, and the framework would monitor the status and pend the read until the data is valid.

The Controller Control Register (CCR) functionality has changed slightly since the previous controller. Previously, it was a 24-bit register, 16 bits of which were used to load a refresh counter (this mechanism was never used on that controller). Six of the remaining bits were unused, and the last two bits were used to clear the Pipeline Register (PLR) and the Array Instruction Register (AIR), respectively. In this design, the clearing of both of these registers is completely controlled in hardware - they are cleared upon a Reset, and at appropriate times during programming of the controller. The CCR is now a 16-bit write-only register, whose contents are used to load a refresh counter. However, as will be discussed in Section 2.2.7, the behavior of this register and the counter have changed.

The Write Data Register (WDR) is a write-only 32-bit register, used to program the Array Instruction Memory (AIM) portion of the control store. The data to be written to memory is loaded into this register, and another operation provides the address, and does the memory write. This func-

tion is hidden from the application programmer, in the software framework.

The Control Data Register (CDR) performs the same function as the WDR, providing data to the Controller Instruction Memory (CIM), rather than the AIM. It is written simultaneously with the WDR, and is present only to avoid connecting the data outputs of the CIM and AIM.

The Opcode Register (OR) is an additional data register on the controller. To launch a sequence, the host sends the address in the CIM of the sequence, and signals the controller to branch to this address. The OR holds this destination address until the sequence has been successfully begun.

2.2.4 Microprogram Unit

The Microprogram Unit consists of the Microprogram sequencer [8], the CIM and AIM, and various control logic. Figure 2-8 shows the architecture of this system component. The Interrupt

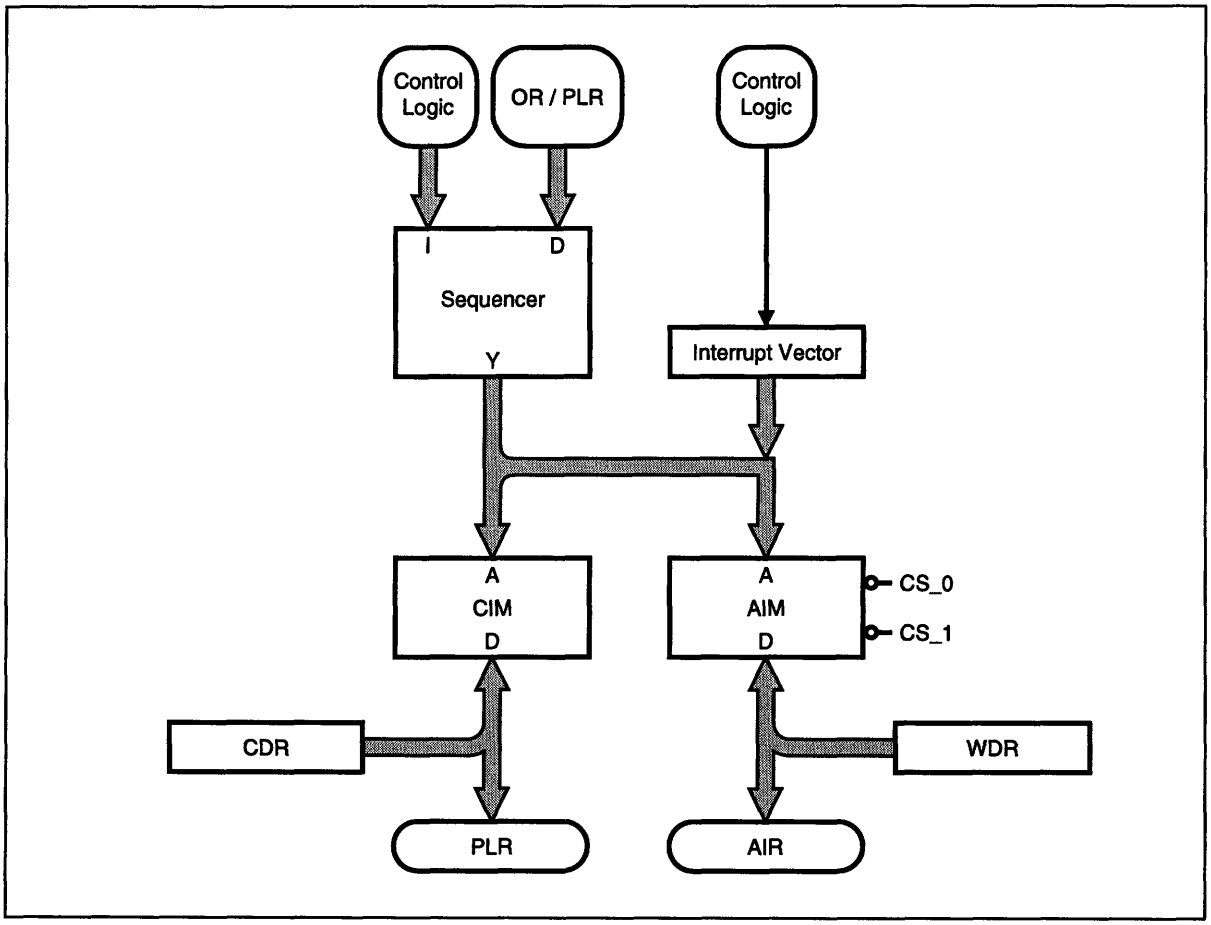

Figure 2-8: Microprogram Unit Architecture

Vector is a part of the refresh interrupt circuitry, and will be discussed later.

The Sequencer has a four-bit instruction input (I), a 16-bit branch address input (D), and a 16-bit address output (Y). In deciding which address to finally output to memory, the sequencer can choose from several sources. If there is no branching involved, there is an internal program

counter that is incremented and sent to the Y outputs. If the instruction is a Return of some sort, the sequencer uses the value on the top of its internal stack (presumably placed there by a gosub or loop instruction). If it is a branch instruction, the sequencer examines the Condition Code ($\overline{CC}$) input, as well as the Condition Code Enable ($\overline{CCEN}$) input. If these evaluate to a condition allowing the conditional branch (either both are TRUE or $\overline{CCEN}$ is FALSE), the sequencer uses the D inputs as the branch destination, and passes them through to the Y outputs. In this case, the branch destination is part of the PLR, which is enabled by the control logic. Finally, the sequencer can take an address from an external source (here, the OR), enables this source with a MAP output on the sequencer, and passes this value through from the D inputs to the Y outputs.

With these resources, there are sixteen control instructions provided by the sequencer. However, only eleven of these were useful in the construction of this system, and were provided interface functions in the software framework (Section 4.3.3). In the following descriptions, any "conditional" instruction means that the action is only taken if the $\overline{CC}$ and $\overline{CCEN}$ inputs dictate action:

Jump zero (JZ) - It resets the sequencer's stack pointer and causes the sequencer to jump to the very first location in microprogram memory. This is the location of the "idle loop," which then launches the next sequence. This instruction is added by the framework to the end of every sequence.

Jump map (JMAP) - Used to jump to an external "map" register. This gives the application designer added flexibility in branching, where the destination doesn't have to be in the pipeline path. In this system, the OR is used as the map register, and this instruction is used to launch new sequences.

Conditional branch to subroutine (CJS) - This instruction allows the controller to gosub to a subroutine, placing the return address on the stack. This is the instruction used when an interrupt is generated. The destination address is supplied from the Interrupt Vector register to the D inputs.

Conditional return (CRTN) - Used to return from a subroutine call.

Conditional jump to pipeline (CJP) - Same as CJS, except that there is no method of returning to the present address. This is the basic jump instruction.

Load counter and continue (LDCT) - Part of the Loop...endloop instruction set. This instruction loads the lone loop counter, and continues.

Repeat pipeline, counter != 0 (RPCT) - Used to continue looping N times, this instruction jumps back to the start of the loop until the counter is 0.

Push/load counter (PUSH) - Part of the Repeat...until instruction set. This instruction loads the lone loop counter, and pushes the address of the start of the loop onto the stack.

Test end of loop (LOOP) - Tests the condition input, and either jumps to the address on the top of the stack (continue looping), or pops the stack and continues. Used to terminate the repeat...until sequence.

Conditional jump to pipeline and pop (CJPP) - Used as a conditional break from the repeat...until sequence. If the condition is passed, pops the start of the loop from the stack (discarding it), and jumps to the address in the pipeline.

Continue (CONT) - Program flow continues to the next instruction.

2.2.5 Host-Controller Interface

There are several low-level interface functions between the host computer and the controller. All of these functions have higher-level wrappers provided by the software framework, but any communication between the host and the controller must go through the register reads/writes shown in Table 2-2. Note that in this controller, 12 functions are used, which is a reduction from 17 in the previous controller [3]. This reduction is due to the fact that the HDPP's instruction word is 27 bits wide, vs. 88 bits for the CAPP, and the reorganization of the Control Store that this fact allowed. To the system software, each of these operations appears as register read or write. The offsets shown are the memory-mapped offsets of the "registers" in the VMEbus interface.

Offset from Base Address[a]	Operation	Read / Write	Operand/ Output Size (bits)[b]	CSR bits to be checked
0	WRTORG	W	16	GO
1	WRTCIM	W	16	GO, ALLBUSY, DIBUSY, WMEMBUSY
2	WRTAIM0	W	16	GO, ALLBUSY, DIBUSY, WMEMBUSY
3	WRTAIM1	W	16	GO, ALLBUSY, DIBUSY, WMEMBUSY
4	GODI	W	32	GO, ALLBUSY, DIBUSY, WMEMBUSY
5	WRTSSR	W	32	GO, ALLBUSY
6	WRTWDR	W	32	
7	WRTCCR	W	16	
8	RDCSR	R	8	

Table 2-2: Low-Level Host-Controller Interface Operations

Offset from Base Address[a]	Operation	Read / Write	Operand/ Output Size (bits)[b]	CSR bits to be checked
9	RDASR	R	8	GO, APBUSY, DIBUSY
10	RDOSR	R	32	GO, APBUSY, DIBUSY
11	RESET	W	n/a	

Table 2-2: Low-Level Host-Controller Interface Operations

a. Because these transactions are 32-bit operations, where the minimum width possible on the VMEbus is 16 bits, address bits A_5 - A_2 are used to determine offset. A_1 is ignored.

b. All bus transactions are 32 bits; this value signifies the amount of this 32 bits that is used.

WRTORG - Write Origin. This operation takes the operand, which is the address in the Control Store of a sequence, and places it in the OR. It also sets the GO bit of the CSR, signalling the controller to launch this sequence.

WRTCIM - Write Controller Instruction Memory. The operand for this operation is the *address* of the memory location to be written. The data to be written is taken from the CDR, which should be placed there by a previous use of WRTWDR.

WRTAIM0 - Write Array Instruction Memory 0. Serves the same function as WRTCIM, but the data is taken from the WDR, and written into AIM0.

WRTAIM1 - Write Array Instruction Memory 1. Same as above, but writes to AIM1.

GODI - Go Direct Instruction. This operation takes the input and sends it directly to the array, without accessing the control store, for a single cycle. In the current implementation, the actual register used is the WDR, as the array instruction width was reduced to 27 bits. Assumes the WRTWDR has occurred previously.

WRTSSR - Write SSR. Writes the input data into the SSR, to be used to select instructions from the AIM.

WRTWDR - Write WDR. Writes a value into the WDR (and the CDR), to be used in a WRT(CIM/AIM0/AIM1) or GODI instruction.

WRTCCR - Write CCR. Uses the 16-bit input to load the top 16 bits of a 24-bit counter, used for the refresh interrupt.

RDCSR - Read CSR. Reads the 8-bit CSR from the controller.

RDASR - Read ASR. Reads the 8-bit ASR from the controller.

RDOSR - Read OSR. Returns the 32-bit OSR, whose values are the 32-bits shifted in from the CCMUX.

RESET - Resets the controller. Clears the PLR and AIR until a WRTORG or GODI are executed. Stops refresh interrupts from being generated until the same. Forces the sequencer to continually execute a JZ instruction. Clears any pending memory writes or sequence launches.

2.2.6 Control Store

Both the architecture and the programming of the new control store are significantly simplified over the 10 MHz controller. This is partly due to the fact that instructions for the CAPP were 88 bits wide, and therefore required a very wide instruction memory and surrounding architecture for reading and programming.

Figure 2-9 shows the old control store arrangement. Each memory module was 128K x 32 bits. The CIM used one module, and simply threw away 64K of memory space. Because the array instruction was 88 bits, a total of 96 bits were stored for each array instruction, 88 of which are used by the array, with the remaining eight being used as extra instruction bits for the controller. Rather than having 88 wires going from the controller to the array, the instructions were sent out in two 44-bit chunks. This was done by clocking the AIM reads twice as fast as the rest of the controller, sending half of the instruction each time. Address bit A_0 in the AIM was used to do the instruction selection, while the Left/Right chip selects were used to send either half of the 88 bits to the array. This arrangement added complexity and hardware to the 10 MHz controller's architecture. To program the AIM, two instructions had to be combined together, and divided into three 32-bit pieces to be loaded into the various modules. There were four host-controller interface functions (WRTAIML0, WRTAIMR0, WRTAIML1, WRTAIMR1) used to write to the memory.

The new design uses three 64K x 32 SRAM SIMMs. One module is for the CIM, and two are for the AIM, for the instruction selection. All address and data lines are wired identically, the only difference being the chip selects for the two AIM modules come directly from the SSR's output bit. This simplified the hardware present on the board, the logic controlling chip select signals, and the programming of the control store. There are now only two host-controller functions used to write the memory, and memory programming takes less cycles than in the previous controller.

The instruction format for the controller instructions was also significantly simplified. Figure 2-10 shows the CIM and AIM words for the old controller, and Figure 2-11 for the new controller. The software framework provided the controller instruction assembler, to translate from the high-level control instructions into the actual bits of the CIM word. To maintain software compatibility, the interface to the assembler was kept the same, and the assembler itself was changed to produce the new bits. For instance, an old instruction such as "next (shssrl shosrr)" now assembles to

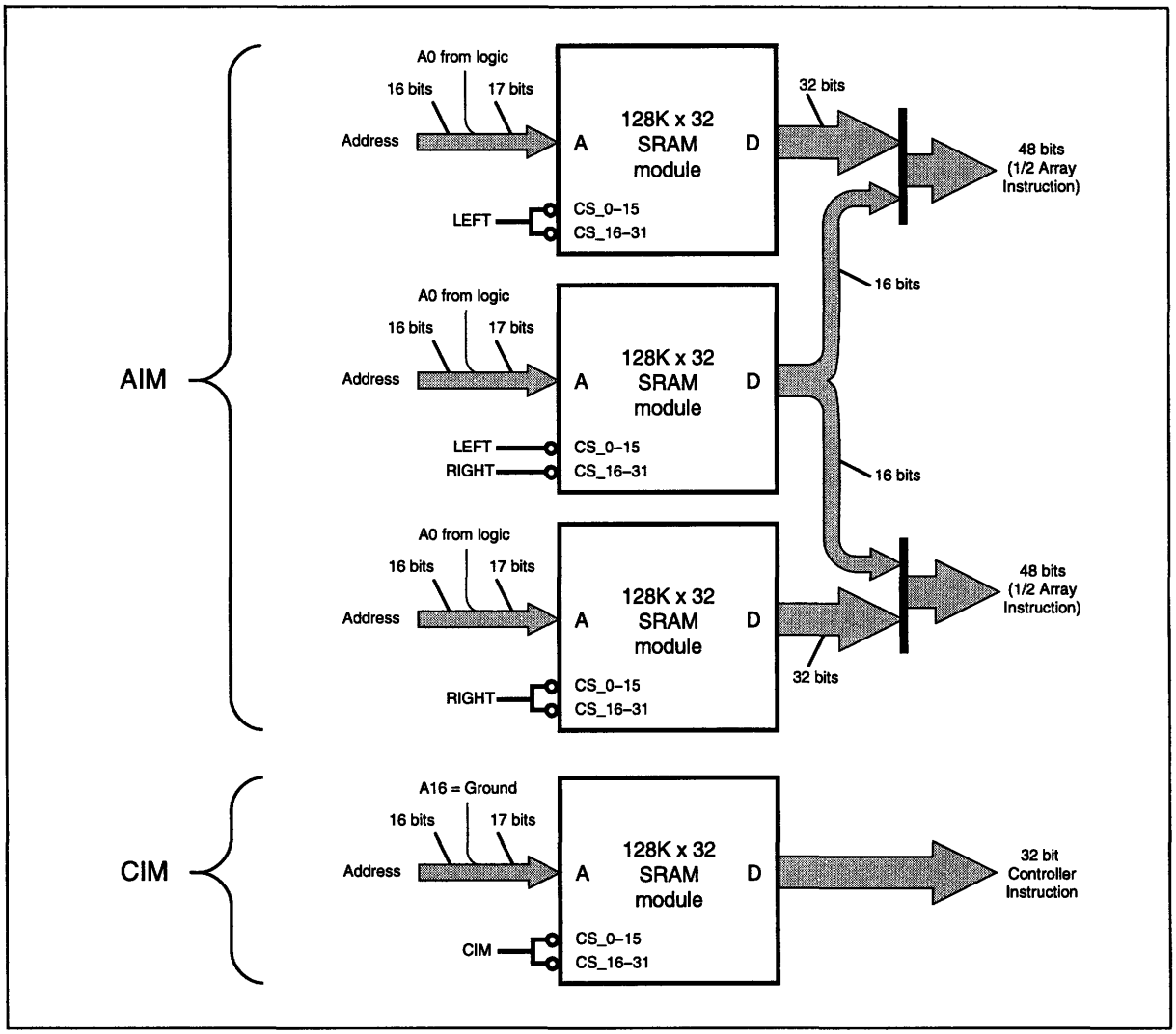

Figure 2-9: 10 MHz Controller Control Store

produce active bits (SSRCE, SHSSRL, OSRCE), rather than (SHSSRL, SHOSRR). There were many ways such as this to take advantage of the software interface in order to make the hardware more efficient without impacting the existing application programmer's interface. The reason for this particular change, in the SSR and OSR signals, lies in the way shift registers are implemented internally in the EPLDs.

Other changes are more obvious. A bit was eliminated by consolidating SETALLBUSY, CLRALLBUSY, SETAPBUSY, CLRAPBUSY into SETCLR, AP, ALL. This makes it impossible to, for instance, set ALLBUSY and clear APBUSY in the same cycle, but this capability is never required. Another change is that the GO bit is now cleared by hardware when a sequence is launched, removing the CLRGO bit. The interrupt logic is now entirely hardware controlled, removing the need for the IAK bit. Removing priority interrupts got rid of four INTCTRL bits. And simplifying the interrupt vector address generation removed the need for the 9-bit refresh vector address field. With these modifications, the control instruction is able to fit into the 32-bit CIM, without the need for the MODE or ASSERT bits changing the function of various bits

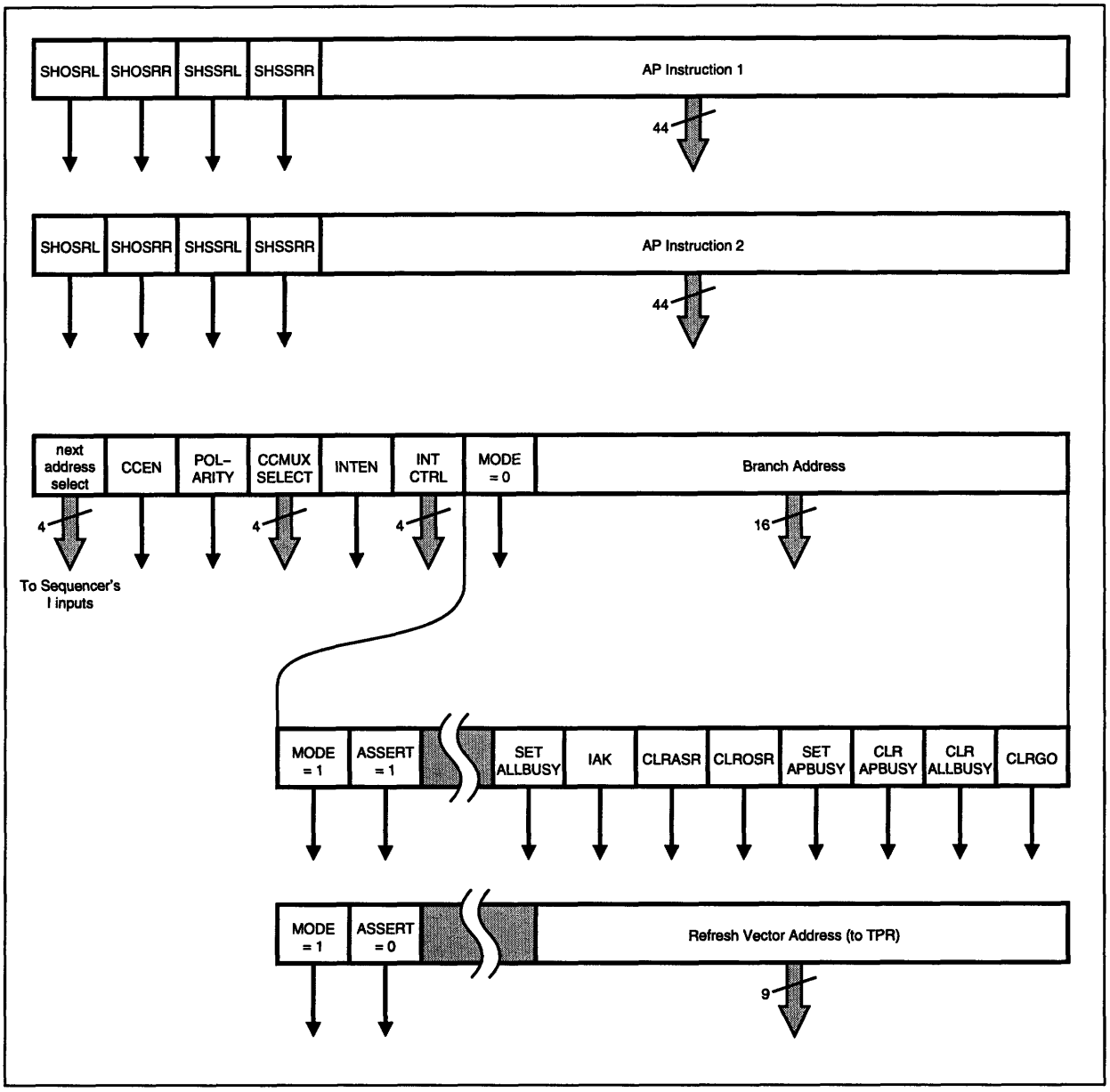

Figure 2-10: Old AIM, CIM Instruction Formats

(which significantly complicates instruction decoding). However, to accomplish this, the new controller uses five bits in the AIM for control instructions, where the old controller used four. This is acceptable, as the HDPP chips need only 27 bits for each instruction.

2.2.7 Memory Refresh Interrupt

As mentioned, the previous controller had hardware support for four levels of priority interrupts, which could be interrupted by higher-priority interrupts. This hardware was developed with no clear purpose in mind, and in the final system, was not used at all. In fact, the only real use for interrupts would have been to refresh the DRAM cells in the array, but even here, the interrupts were not tested or used -- instead, memory refresh sequences were embedded in the application explicitly by the programmer.

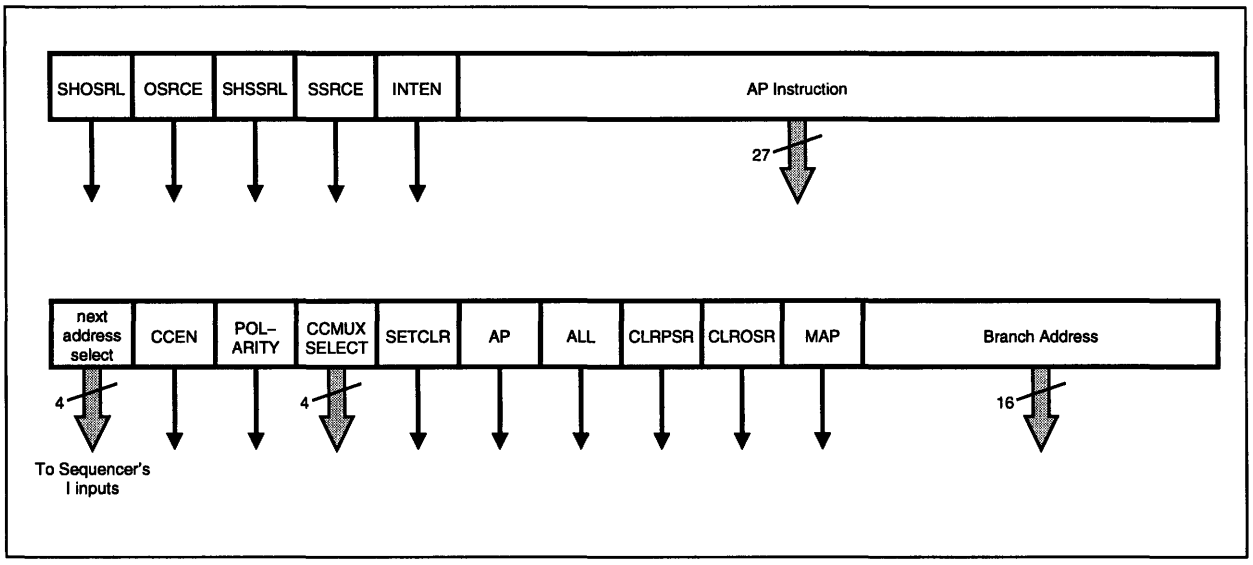

Figure 2-11: New AIM, CIM Instruction Formats

The final system has as a goal, as complete a separation between the hardware and the programmer as possible. One aspect of this is removing the need to know about the refresh requirements of the array being used. Therefore, a requirement for an interrupt for use in memory refresh was added to this controller, as well as built-in implementation of this interrupt.

This dictated the need for only a single interrupt, so priorities were eliminated. However, because the memory on the HDPP chips has not yet been characterized, there is a need for greater flexibility in the refresh interval. In addition, it is also desirable to have some method of specifying in software where the refresh sequence will be, as opposed to hard-coding the location.

A requirement for refresh intervals varying from 100 us to 500 ms was set, to cover all possible refresh intervals needed by the array. Because it was impossible to generate this much resolution in a 16-bit counter without adding several clock dividers, a new CCR architecture was developed. The 16-bit CCR is used to load the upper 16 bits of a 24-bit counter. The lower 8 bits are reset when starting the refresh count. This gives the capability of a count ranging from 256 to 16,776,960. Assuming a 20 MHz clock, this equates to refresh intervals from 12.8 us to .839 s. In addition, the framework now initializes this counter to set a refresh interval of 100 ms (assuming a 20 MHz clock), loading the 16-bit CCR with hexadecimal 1E85 (decimal 7813).

The generation of the interrupt vector was also changed. In the old controller, an additional register (TPR) was used to hold an interrupt address. There was a specifically formatted CIM instruction for programming this register, that didn't use the host-controller interface (see Figure 2-10). This register would then point to the memory location containing the interrupt service routine (ISR).

In the new controller, logic more similar to general-purpose computers is used. An address is hard-wired on the controller board, pointing to an *interrupt vector table*. In this case, the table has

only one entry, but the idea is the same. When an interrupt occurs, the register's contents are supplied to the memory, and the current address is placed on the stack. At this memory location resides the *interrupt vector*, which is simply a branch to wherever in memory the ISR resides. In this way, the table may be placed in a small portion of reserved memory, but the larger ISRs may be placed anywhere in memory, and moved around at run time. At the end of each ISR, then, is the return that pops the stack and continues program flow.

Chapter 3

New Controller Requirements

The previous controller had a number of issues that motivated this research. These issues were as follows:

Decreasing board size - the new system is targeted to fit into a VME chassis on a workbench. This is to enable this system to be portable and somewhat more sturdy. Because the previous controller was done using wire-wrap techniques and a large number of PLDs, there are ample opportunities to optimize the size of the board itself.

Increasing system speed - the previous controller was to deliver complete instructions to the array at a frequency of 10 MHz. This figure was chosen because it was estimated that this would be close to the maximum speed at which the array could operate. The new processor in this system is expected to have a higher clock frequency, and thus the controller must have a corresponding increase in speed. The new controller is capable of delivering instructions at 20 MHz.

Increasing controller robustness - a wire-wrap board, while suitable for fast prototyping and testing, is not a robust method for a portable system that is intended to operate for any long period of time. In addition to this shortcoming, the previous controller wouldn't function at a clock period of 100 ns or lower (the target), and fell into and out of stability at certain lower clock periods. The new design investigated this problem and implemented changes.

3.1 Decreasing Board Size

3.1.1 PCB Technology

As mentioned, one of the primary goals of this thesis was to achieve greater logic integration on the controller board, in order to have a system that will fit cleanly into a standard VME chassis. One way to achieve this goal was to make this controller a printed circuit board. This, along with using surface-mount ICs as much as possible, greatly reduced board area. This was due to the fact that not only were modern surface-mount ICs available with much smaller footprints than equivalent DIP packages, but the newer parts were also more integrated. For instance, register chips included twice as many registers as before.

Not all devices were available in surface-mount packages, however. Notably, the microprogram sequencer in a DIP package was retained on the new controller. This was due to the fact that other packages were not readily available, and too much work in the software application framework was built around the sequencer's architecture to change to a microprocessor-based controller.

3.1.2 Logic Integration

Another strategy was to integrate as much of the logic as possible into Xilinx programmable logic devices. The decision to use Xilinx was made primarily for economic reasons: previous research in this lab has been done with Xilinx products, so the software tools were already owned and installed.

Xilinx offered two suitable types of programmable logic devices: Field Programmable Gate Arrays (FPGAs), and Erasable Programmable Logic Devices (EPLDs). Each offers distinct advantages and disadvantages to different types of applications. EPLDs are suitable for complex controllers, high speed state machines, wide decoders, and PAL-integration. FPGAs have key applications as simple state machines, complex logic replacement, and board integration [9].

As the previous controller board consisted of 29 PALs (primarily 22V10s), EPLDs would make combining these devices a simpler task than with FPGAs, simply because the design software (at the time this research was begun) supported ABEL-style design files with EPLDs, but not with FPGAs. However, FPGAs allow much more integration of SSI logic, such as registers, and are capable of more complex logic internally. EPLDs have predictable timing paths and guarantee 100% routability and use of resources, while FPGAs have variable timing based on routing, and will not allow use of all internal resources due to routing limitations.

As the critical path analysis in Section 3.2.2 will show, only 15 ns of overhead was finally available in the 50 ns clock period, to resolve several 13-input 1-output registered equations. Xilinx FPGAs are made up of a dense array of Configurable Logic Blocks (CLBs) which allow any type of logic to be implemented using basic lookup tables. Figure 3-1 shows the structure of a Configurable Logic Block in a Xilinx FPGA. Each CLB receives eight inputs from the I/O blocks. These can each be used to generate two outputs based on four inputs each, or can be combined in the H logic block, to generate an output based on nine inputs. Xilinx specifies the delay through each CLB (via the H' path) as 7.0 ns (this is based on the fastest version of the 4000H family, which is high I/O). To implement the 13 input equation, a minimum of two CLBs would be necessary. Taking into account routing delays (which are significant and unpredictable in FPGAs) and I/O delays, the worst-case critical path using this FPGA would exceed 50 ns.

Figure 3-2 shows the basic design of a Xilinx X7300 family EPLD (more or less function blocks are present depending on which EPLD is chosen). Each Fast Function Block (FFB) receives a total of 24 inputs, from the Fast Input block and from the Universal Interconnect Matrix (UIM). Each FFB contains nine "macrocells", and each of these can implement five product terms with 24 inputs, with a total block delay of 7.0 ns. Therefore the critical path equation can be done with a single FFB, yielding the resultant critical path that is under 50 ns (refer to Section 3.2 for this calculation).

3.1.3 Logic Modification

In addition to increased logic integration, effort was taken to redesign any portions of logic that could be further reduced to save resources. One of the first ways of doing this was taking advantage of the new HDPP architecture, which only requires 27 bits per instruction (compared to the

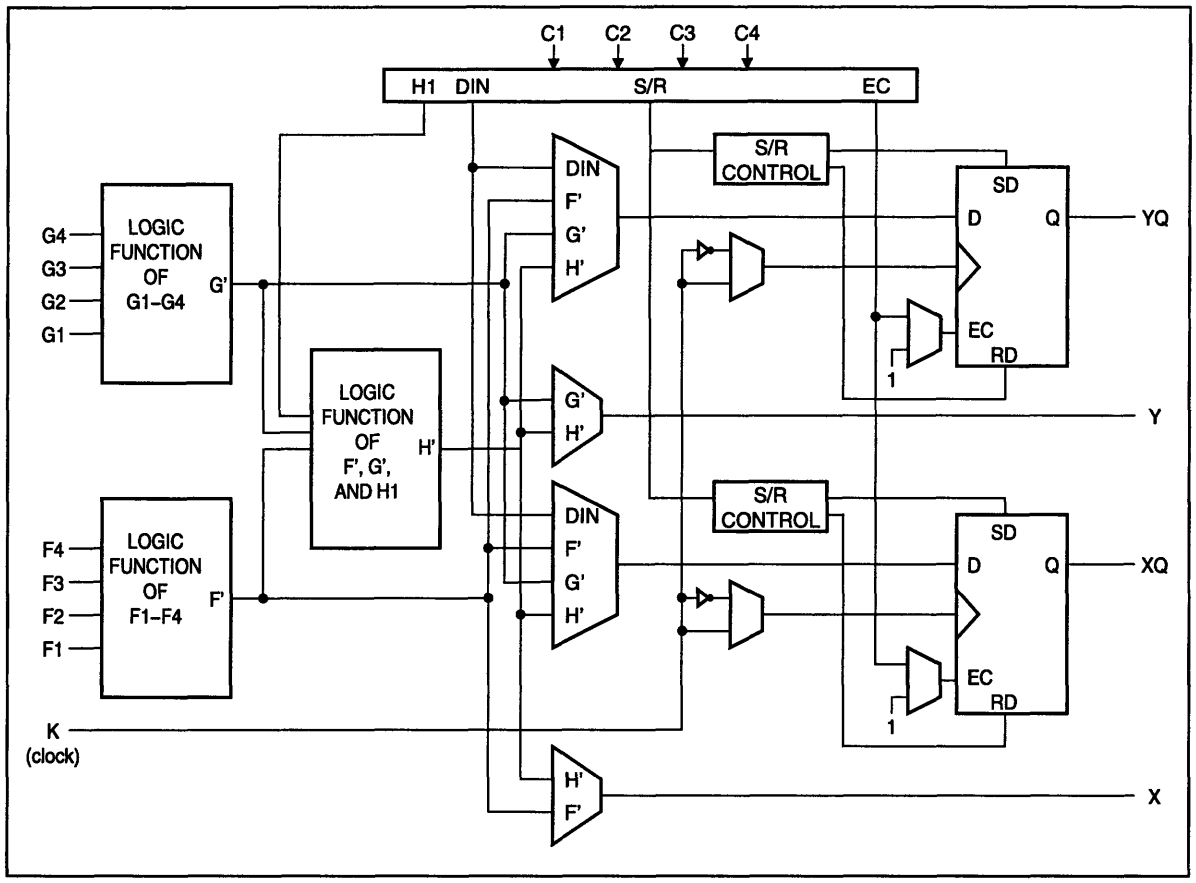

Figure 3-1: FPGA CLB Architecture

CAPP, which required 88 bits). This saves in several areas -- less registers were required to latch and read or write these instructions to the AIM or the array. In addition, the AIM could be constructed with two 64K x 32 SRAMs, rather than three, as was the case in the previous controller. This had other side effects of reduced memory-interface logic in the EPLDs, as well.

Another place where logic was saved was the Interrupt Control portion of the controller. The previous controller employed a sophisticated and complicated interrupt controller. It was capable of four levels of interrupts, the highest priority of which was used for memory refresh. It also supported nested interrupts, and maskable interrupts. However, in practice, this interrupt controller was never tested or used in the CAPPs image processing system. Instead, memory refresh sequences were explicitly added to application code by the programmer.

It was desired to completely hide the memory refresh from application writers in the final design, so this solution was not acceptable. In the interests of making the controller as simple as possible, though, it was decided not to have an interrupt capability comparable to that of a complete desktop computer. Instead, support for a single interrupt for use as memory refresh only, was added to the new design. The previous design used five PAL22V10s for the interrupt logic, and two more for the CCR refresh counter. By eliminating much of this logic, the counter is still present, but the five PALs were reduced to the equivalent of three (three function blocks in one EPLD).

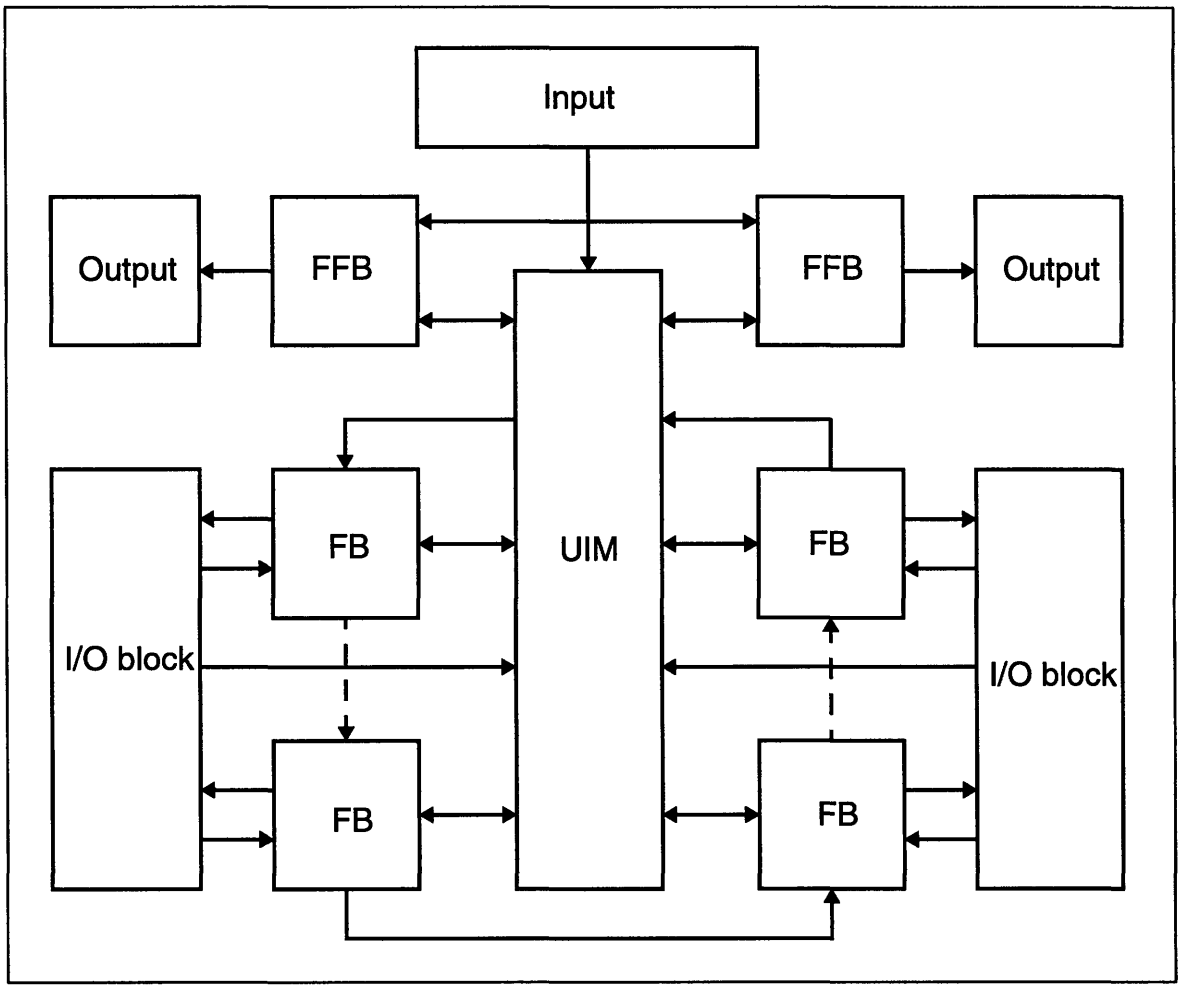

Figure 3-2: EPLD (XC7300) Device Block Diagram

3.1.4 Results of Integration/Modification

The final results of these efforts to reduce board size were successful. Table 3-1 shows a comparison of the final parts used on the old and the new controllers. Clearly, the largest savings were in

CAPP Controller		HDPP Controller	
Chips	**Type**	**Chips**	**Type**
4	Bus data latch	2	Bus transceiver
1	Bus address latch	0	(EPLD)
1	Bus interface IC	1	Bus interface IC

Table 3-1: Part Count Comparison

CAPP Controller		HDPP Controller	
Chips	**Type**	**Chips**	**Type**
2	Interrupt latches	0	(EPLD)
1	Clock receiver	1	Clock receiver
39	Register chips	10	Register chips + (EPLD)
29	PLDs	2	EPLDs
4	Memory modules	3	SIMMs

Table 3-1: Part Count Comparison

register ICs needed, and PLD integration/elimination. It should be noted that 50% of the reduction in register chips was due to the more narrow AIM instruction word, and the other 50% from logic redesign in the new controller.

Table 3-2 breaks down the results of redesigning and integrating logic in the EPLDs. Each function block corresponds closely to the logic capability of a 22V10 PLD. Integrating several PLDs into a single EPLD not only saved board space merely by grouping several parts into one, but also reduced the number of wires crossing the chip-to-chip boundary. This, in addition to the fact that signals in the EPLD's Universal Interconnect Matrix may be routed to any other Function Block, greatly reduced logic in some cases. Some of these benefits were realized even with an increase

Function	PLDs	EPLD FBs
Bus interface (read / write / enable)	1	1
Instruction decode / address latch	3	1
Memory read / write / enable	5	3
Sequencer control signals	2	2
Controller status register	2	2
Interrupt processing logic	4	3

Table 3-2: PLD Integration Results

Function	PLDs	EPLD FBs
32-bit P-in S-out shift register (SSR)	4.5	4
32-bit S-in P-out shift register (OSR)	4.5	4
Interrupt Counter	2	3
Interrupt generation logic	1	1

Table 3-2: PLD Integration Results

in complexity, such as the Interrupt Counter, which was 16 bits on the previous design, and 24 bits currently, to allow a wider range of refresh intervals.

3.2 Increase System Speed

Figure 2-5 on page 24 shows the entire controller architecture. Two key components that cannot be integrated into Xilinx devices are the microprogram sequencer and the control store memory. Because the delays of these two components are a large part of the 50 ns clock period, the critical path for the controller executing instructions includes these two devices. At this point, it is useful to analyze the critical path of the 10 MHz controller, and discuss the changes in the new design to meet a 50 ns specification.

3.2.1 10 MHz Controller Critical Path

In the previous controller design, Figure 3-3 shows the critical path for a 10 MHz design, as well as other paths that will become critical for a 20 MHz design. The path documented by Hsu [3] is the CCMUX path. The constraining equation for this path is the following:

$$t_{ckq\ PLR} + t_{CCmux} + t_{CC\text{-}Y,sequencer} + t_{mem\ rd} + t_{su\ PLR} < period \qquad (1)$$

$$10 + 20 + 16 + 30 + 4 = 80 \text{ ns}$$

where 80 ns is the worst-case result from the parts used on that controller.

However, there were two additional paths, each of which had a longer delay than the CCMUX path, and both of which affected design on the 20 MHz controller. The first was the path through the instruction selection logic, which depends on the longer I-Y delay of the sequencer:

$$t_{ckq\ PLR} + t_{Inst\ Logic} + t_{I\text{-}Y,sequencer} + t_{mem\ rd} + t_{su\ PLR} < period \qquad (2)$$

$$10 + 20 + 20 + 30 + 4 = 84 \text{ ns}$$

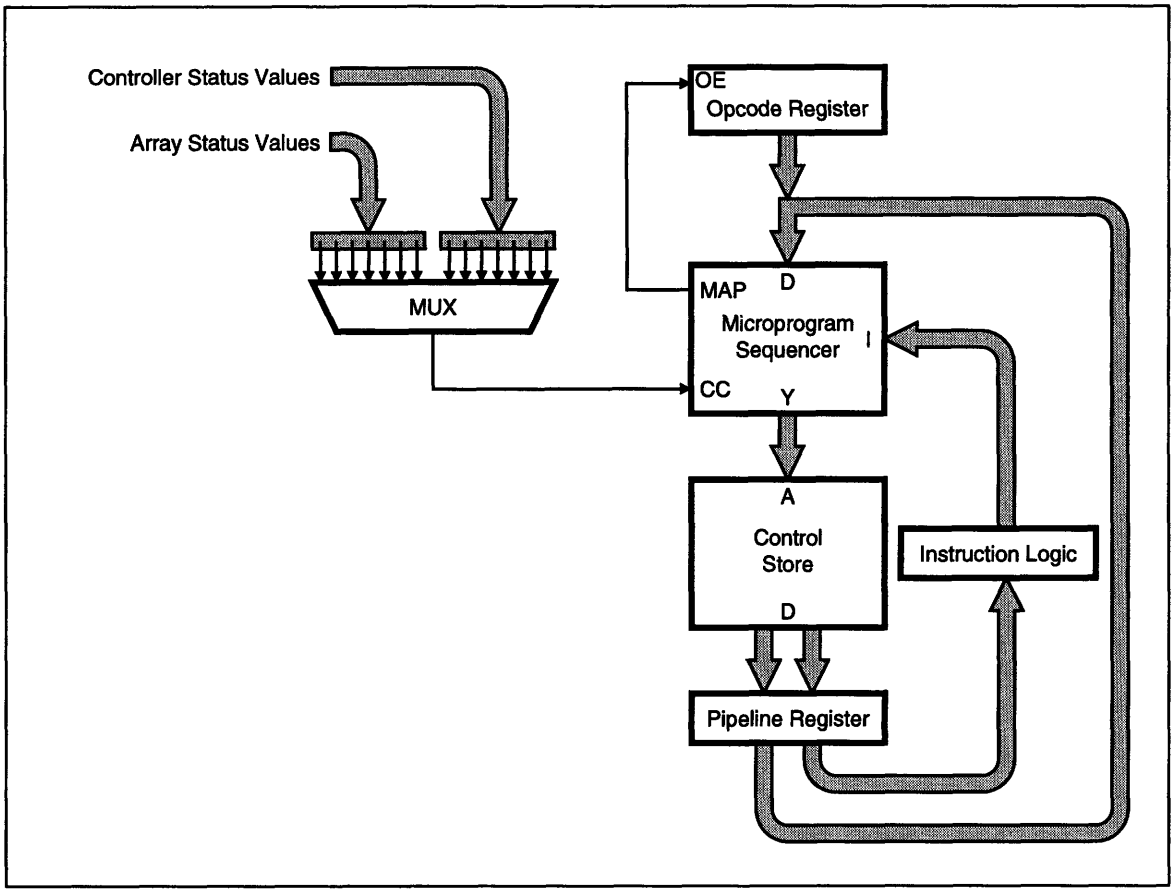

Figure 3-3: Critical Paths of 10 MHz Controller

While this time is longer than the documented critical path, it is still allowable in a 100 ns design.

The second undocumented path is the actual critical path in the 10 MHz controller. Each time a sequence is run, the Opcode Register's outputs are enabled, to override the Pipeline Register, and provide inputs for the Sequencer's jump. The Sequencer includes a MAP output, which was used in that design to enable the Opcode Register. Therefore, each time a new sequence is run, the resultant branch must take place in a single cycle, and the following critical path results:

$$t_{ckq\ PLR} + t_{Inst\ Logic} + t_{I\text{-}MAP,\ sequencer} + t_{OE\text{-}output,\ opcode\ reg} + t_{D\text{-}Y,\ sequencer} + t_{mem\ rd} + t_{su\ PLR} < period \qquad (3)$$

$$10 + 20 + 13 + 12 + 12 + 30 + 4 = 101 \text{ ns}$$

This path, using worst-case values, is greater than the 100 ns specified for this design. When this design was tested, it was found not to function at clock periods below 108 ns, supporting this analysis. The additional time needed could possibly be attributed to ringing and longer propagation times typical of wire-wrap boards.

3.2.2 Sequencer I-Y Correction

Referring back to equation (2), this was the simplest path to bring below 50 ns. First, the SIMMs

used in this design have a significantly faster access time -- 15 ns compared to 30 ns. In addition, the Instruction Logic was placed in a Fast Function Block of the EPLD. This gives the following times for equation (2):

$$5.5 + 7.5 + 20 + 15 + 4 = 52 \text{ ns}$$

While this is close to the target, the worst case is not below 50 ns as desired. To get there, a logic modification was made. At this point in the discussion, at the clock transition, the I bits from the CIM are clocked through the PLR (which is integrated into the EPLD as an input register). These bits then pass through logic to produce the sequencer's input Instruction bits (in the case of an interrupt, for example, the CIM's I bits are overridden). Following that are the sequencer delay, the next memory access, and the PLR's setup time.

The Instruction Logic is implemented in a large state machine. Based on whether we are in an interrupt state, a CIM-write or AIM-write state, reset state, or normal execution, the logic will output a special instruction, a JZERO instruction, or pass through the Instruction bits. The difficulty arises in that at the clock, the state bits change (5.5 ns), and these then get fed through the UIM and the FFBs logic block (7.5 ns). The solution (shown in fig) was to change the design, to place the register at the end of the clock cycle, instead of the beginning (register the next cycle's final Instruction bits at the end of a cycle, instead of calculating them at the start of the current cycle). This means that at the clock edge, *output* registers send the Instruction bits to the

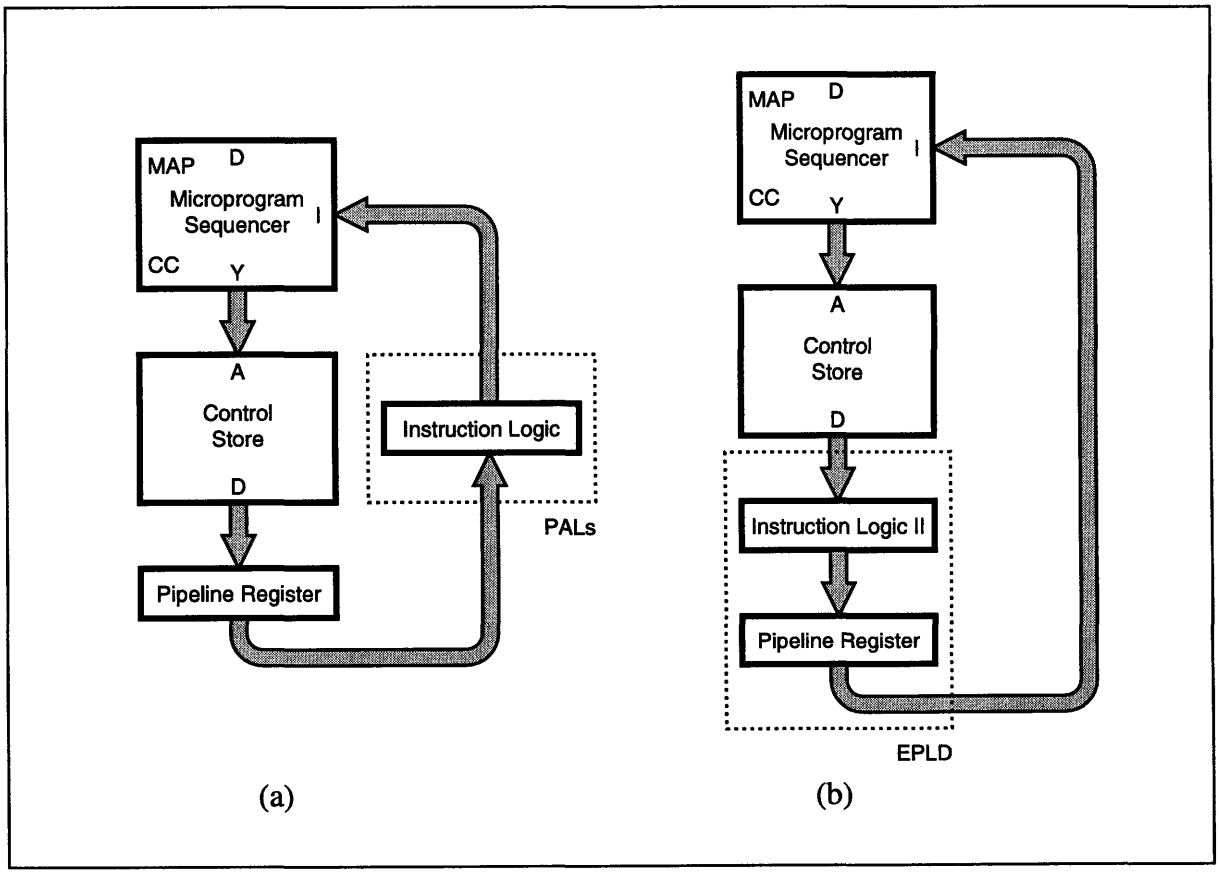

Figure 3-4: Critical Path Fix (a) 10 MHz path. (b) New 20 MHz path.

sequencer (5.5 ns). This value goes to the sequencer, and a resultant address is presented to memory. The next cycle's Instruction bits come into the EPLD as combinatorial inputs (2.5 ns), pass through the FFBs logic block (no UIM), and present the next cycle's Instruction bits to the output register (logic + setup time = 5.5 ns). This saves time by moving part of the PLR into the EPLD. The result is

$$t_{ckq\ PLR} + t_{I\text{-}Y,sequencer} + t_{mem\ rd} + t_{in,EPLD} + t_{FFB} + t_{su\ PLR} < period$$

$$5.5 + 20 + 15 + 2.5 + 1.5 + 4 = 48.5 \text{ ns}$$

which meets the 50 ns specification. The only caveat was to change the state machine, so that the desired Instruction bits were output one state earlier than before, due to the change in registering scheme.

It is important to note that this is the path that made FPGAs unfeasible for this design. There is only 15 ns available for the chosen logic device to evaluate the Instruction equations (13-inputs for each of four outputs). The CLB delay in the FPGA for a nine-input equation, as mentioned in Section 3.1.2, is 7.0 ns. This is neglecting I/O delays and setup times. The logic delay in the EPLD for a 24-input equation is 1.5 ns, assuming the equation will fit into the sum-of-product limitations of the device (this was true for this design). So to go from one FFB delay of 1.5 ns to the required two CLB delays would have added 12.5 ns to the critical path, making it very difficult to meet the specification.

3.2.3 Sequencer CCMUX Correction

Equation (1) showed the equation for the documented critical path of the previous design, along the CCMUX path. The problem arises because this 16:1 MUX also has a Polarity input, which doubles the number of product terms needed. This was implemented previously in two 10 ns PALs, for a total delay of 20 ns. Because of the large number of product terms needed (32), this equation used more than one macrocell[1] in the EPLD. Alternatively, it could be implemented as the 16:1 MUX, and then passed through a simple logic gate to handle polarity, but this second level of logic would still use another macrocell.

This extra macrocell causes extra delays through the UIM and feedback paths. To further complicate matters, one of the Fast Function Blocks was used by the previous section's solution, using up enough resources that this output will not fit entirely into an FFB.

To remove some delay, the PLR bits for this equation were also integrated into the EPLD, and the inputs and outputs of this local PLR are kept within the chip, greatly reducing the clock-Q delay (no I/O Pad to drive). The total delay for just the 16:1 MUX (21 inputs, one output) using FB delays (worst case) is 26.5 ns. This makes the entire path delay (referring to equation (1)):

1. A macrocell is the logic surrounding one I/O pin in the EPLD. Each FB is made up of nine macrocells.

$$2.5^{1} + 26.5 + 16 + 15 + 4 = 64 \text{ ns}$$

Efforts were made using built-in logic minimization to reduce the time needed to resolve the CCMUX equation, but the first implementation was the fastest achieved. This meant that another approach had to be used.

Because compensation in hardware would have caused a large increase in complexity, the end result was to add another level of pipelining to this path, and compensate in software. Because the next value of the CCMUX output does not depend upon the previous value, this added pipeline can be done with no increase in complexity for this output. The division was made at the calculation of the output. Therefore, two paths are created:

$$t_{ckq\ PLR} + t_{CCmux} + t_{su\ PLR2} < period$$

$$2.5 + 26.5 + 1.5 = 30.5 \text{ ns}$$

$$t_{ckq\ PLR2} + t_{CC\text{-}Y,sequencer} + t_{mem\ rd} + t_{su\ PLR} < period$$

$$5.5 + 16 + 15 + 4 = 40.5 \text{ ns}$$

The modification to software, at first glance, is rather simple. Because the hardware will now cause any condition codes (the output of the CCMUX) to be delayed by one additional cycle before appearing at the sequencer, the software must be modified to send the desired condition code to the controller one sequence earlier.

There are two simple approaches to accomplish this. One is simply to take the binary code produced by the controller instruction assemblers, and modify it by taking the MUXSEL and Polarity bits, and placing them in the previous instruction. This would be done at a low level in software, so the application programmer would never notice the difference. One of the difficulties that arise from this approach, however, is a conditional branch being the first instruction in a sequence. In this case, more complex software would have to be written. This does not seem like a large change, but because the assemblers are written in an object-oriented way, where an instruction assembles completely independent of other instructions or its position in the sequence, this change would require removing this elegant layer of abstraction, which is undesirable.

The other option is, for this particular case of conditional branches, is to have the assembler output more than one instruction -- one to be a NOP setting the test condition, and the other to be the actual branch instruction. This still isolates this instruction from others in the sequence, at the cost of an additional instruction in the output. This was the chosen method, both because it changed the overall software methodology less, and because conditional branches are not often used in sample code to date on this system, and even on general-purpose computer systems, branches only comprise approximately 12% of all instructions [10].

1. The clk-Q delay here is internal to the EPLD, so is shorter than the delay including an output pin.

However, as discussed in Section 2.2.7, the memory refresh interrupt further complicates this software solution. If an interrupt is triggered and serviced, the processing time causes the actual program to be interrupted two instructions after the request was generated. This can cause an undesirable result here, in that the two-instruction branch constitutes a *critical section* of code (one that cannot be interrupted). If an interrupt is allowed to occur between these instructions, the first instruction will set the condition code, and then the interrupt will be serviced. When the interrupt returns control to the program, the next instruction processed will be the conditional branch. However, the correct value of the condition code would have been lost in processing the interrupt.

The solution to this problem was to also disable interrupts between those two instructions. Because the Interrupt Disable bit takes effect two instructions later, a conditional branch now expands into the following instructions:

Source Code	Assembled Code	
Branch on Go to X	Next (Inten = 0)	← interruptible
	Next (CCMUX = Go)	← interruptible
	Cond. Jump to Pipeline (X)	← NOT interruptible

Table 3-3: Expansion of a Conditional Branch

In this table, the second and third rows compose the critical section of code. With the interrupt latency, this guarantees that while either of the two Next instructions might be interrupted[1], the conditional jump will not, thereby using the correct CCMUX value set by the second Next. This increases the inefficiency of conditional branches, but again, this trade-off was deemed appropriate given the relative frequency of conditional branches, and the high complexity of a hardware fix in a design targeted for simplicity and low cost. In addition, this solution requires no changes to existing top-level software, only the controller-specific assembler routines.

3.2.4 Opcode Register Output Enable Correction

The most serious problem to be corrected was the *actual* critical path in the controller design, which is one not documented in the previous controller. As equation (3) shows, there is an extremely long path, which passes through the sequencer *twice*, in the case of a JMAP instruction. This instruction is used every time the host computer tells the controller to launch a sequence, so a critical path here will certainly impact the system, no matter what the application.

While this problem is more serious than those previously mentioned, in that its result was to keep the previous controller from functioning at 10 MHz, its solution is simpler, though it consists of several parts. The first of these was to implement the instruction logic as a state machine, so that

1. "interrupted" here means that this instruction will not execute, and will be returned to after the interrupt

it's output times are only the CLK-Q time of the EPLD. While this implementation took up more resources than the previous implementation (due to state bits and their product terms), this made many of the product terms simpler and removed a level of logic required to evaluate these equations.

The second step was to redesign the path itself, removing the first pass through the sequencer. Instead of using the sequencer's MAP bit, the controller generates one of its own. At the same time the EPLD's internal PLR generates the output Instruction bits, it also outputs a MAP bit, equivalent to what the sequencer's would be, removing this path through the sequencer. The resultant critical path, standard outputs in the EPLD (not FAST outputs), is as follows:

$$t_{ckq\ PLR} + t_{OE\text{-}output,\ opcode\ reg} + t_{D\text{-}Y,\ sequencer} + t_{mem\ rd} + t_{su\ PLR} < period$$

$$7 + 23 + 12 + 15 + 4 = 61 \text{ ns}$$

Note that the specification for the Opcode Register cites the OE time to be from 12 ns, with a capacitive load of 50 pF, or 23 ns, for a capacitive load of 300 pF. With this design, the OR's outputs will be enabled at the same time as the PLR's outputs are disabled. This could initially provide a very high capacitive load to the OR, resulting in an unpredictable time for that component of the path. In a robust design, this dependence on a race must be eliminated.

The final part of this solution was one similar to that taken with the conditional branches -- expanding the JMAP instruction. If, as before, the state machine mentioned above is modified to output an active MAP bit one cycle earlier, the OR outputs will go active in the state before it's needed. If we can guarantee that the branch bits of the PLR are not needed during this cycle, operation will not be impaired (the MAP bit enables the OR, and disables the branch portion of the PLR). We are guaranteed that the OR's outputs will be valid by the end of the first cycle, removing this 23 ns component from the next cycle, where the JMAP actually occurs. This brings the total delay from this component of the design to 38 ns.

3.2.5 Controller Idle Loop

With these changes, the idle loop of the controller is changed drastically over the previous controller. This is the code that resides at location 0 in the Control Store, and is responsible for detecting a sequence run request by the host, and processing this request. The following table illustrates the *source* code (which is the same for both controllers), and the assembled code for both the 10 MHz and 20 MHz designs:

Source Code	Assembled Code (10 MHz)	Assembled Code (20 MHz)
Branch on ~Go to 0	Branch on ~Go to 0	Next (IntEn = 0)
		Next (CCMUX = Go, Pol = 0)
		Branch on ~Go to 0
JMAP	JMAP (clear GO)	Next (MAP = 1)
		JMAP (MAP = 1)

Table 3-4: Idle Loop Code Expansion

This modification, while necessary to meet the 20 MHz requirement, has also increased the size of the idle loop by 150 %. This is partially offset by the fact that the clock is expected to run faster in the new design, but it is nonetheless true that the new idle loop is less efficient than the old implementation. However, because this extra time will be swallowed by the savings gained in executing long sequences with a faster clock, this trade-off was acceptable. In addition, as will be shown later, due to more efficient logic in the bus interface portion of the design, the 20 MHz controller is in fact faster at launching a sequence than the 10 MHz design.

3.3 Increasing Controller Robustness

There were three main problems to be remedied in building a more robust controller. The first, as previously mentioned, was the fact that the previous controller was a wire-wrap design. This type of design does not last well under typical-use conditions. While useful for prototyping, wire-wrapped boards have a tendency to have broken connections, loose wires, and excessive noise. This problem was the easiest to address, simply by doing the new design with a printed circuit board.

The second issue was the fact that the old design didn't function at the target clock period of 100 ns. This problem was outlined in the previous section, and should be solved. The longest critical path delay, which is from equation (2) previously (the sequencer's I-Y path), is 48.5 ns. The solutions outlined in that section should enable this design to work at the specification of 20 MHz.

The final issue remaining was the controller's unstable behavior observed at clock rates *below* the successfully tested rate of nearly 10 MHz. The controller would fall in and out of operation as the clock rate was slowed. This was due to the nature of the design of this controller, and a property of VME Buses called *data rot*. Figure 3-5 shows the timing of a VME write cycle. In this architecture, after the VME *master* (in this case, the VME chassis) puts a valid address on the address lines, it asserts $\overline{AS}$ (Address Strobe). Following that, the two Data Strobe lines are asserted when the master has put valid data on the data lines. The *slave* (in our case, the controller board) latches the data, and then asserts $\overline{DTACK}$. At this point, once the master sees $\overline{DTACK}$ asserted, it is free

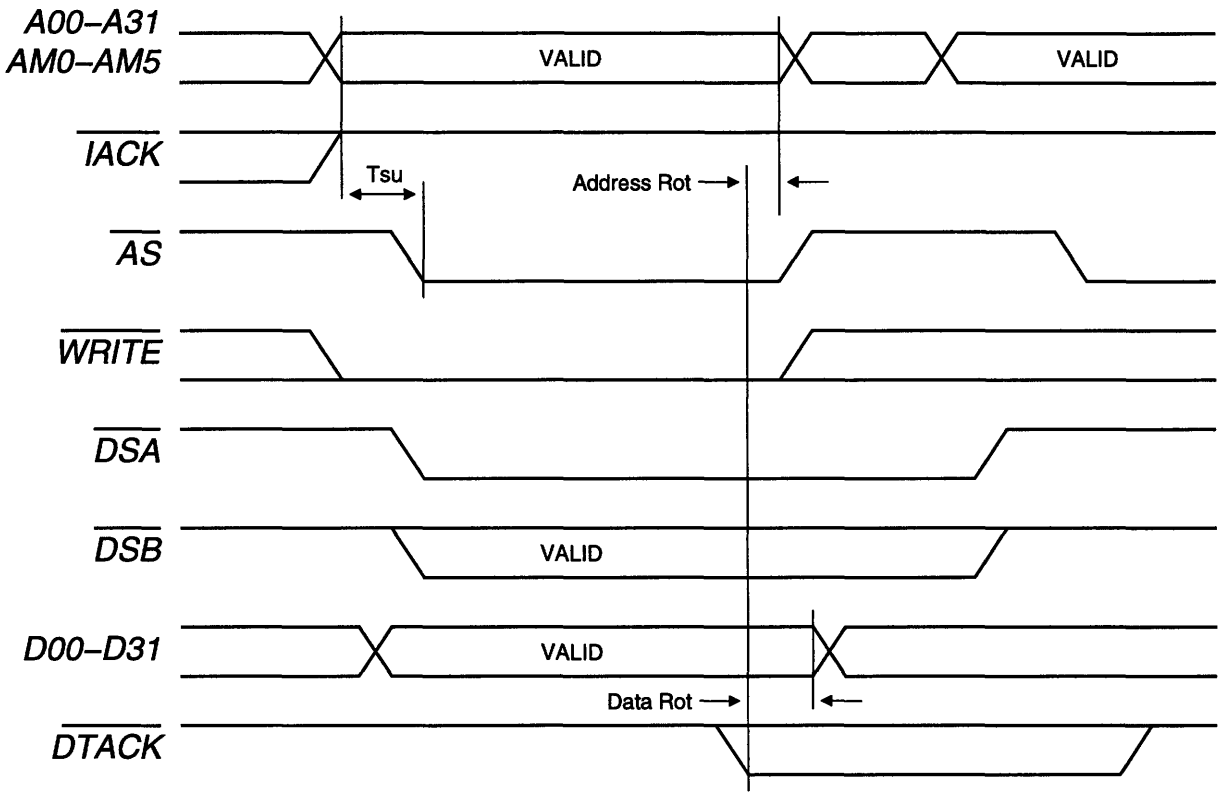

Figure 3-5: VME Write Cycle

to place new address and data values on the bus, to pipeline the next data transfer. Changing the data lines before $\overline{\text{DTACK}}$ is negated is referred to as data rot. Similarly, changing the address lines in this time period is called address rot.

The reason this presented a problem lies in the fact that the 10 MHz controller didn't use elements of state in the design regarding sequence launching. To launch a sequence, the host computer sends a WRTORG instruction to the controller. This results in a VME write, where the address of the sequence to be run is the data written. The controller then writes this data into the Opcode Register, and at a subsequent time, uses this data as an input to the sequencer as a branch address. The controller design had a problem in that, when the $\overline{\text{DS}}$ bits are asserted and the address is valid, a local board signal gets asserted for the duration of the bus cycle. During the time this signal is asserted, if the instruction is a WRTORG instruction, the Opcode Register continually clocks in data from the bus' data latch. This local signal would then be negated when it saw the $\overline{\text{DS}}$ bits get negated. Clearly, data rot will impact this design, as it is possible for invalid data to be latched into the Opcode Register.

However, as observed on the board, the clock frequency will determine whether this problem will cause a malfunction. For sufficiently slow clocks, the correct data will be latched, and before the next active clock edge, the local board signal will have gone false, preventing an incorrect value

from being written into the Opcode Register. For a faster clock, problems will occur as the board's clock edge hits before the $\overline{DS}$ lines have been negated. For a still faster clock, however, the correct value will be (repeatedly) latched in, and the controller will have time to launch the sequence before the bus cycle even completes, which will clear the GO bit signifying a pending sequence launch. This bit is not repeatedly set, so even as the Opcode Register has invalid values written to it after this sequence is launched, these values will never be used, and the controller will run the proper sequences.

The solution to this problem was simple, given that the new controller design heavily utilizes state machines. One of these state machines handles bus transactions, and another handles the WRTORG instruction. In the case of a WRTORG, the new controller will, at the appropriate time, latch a single value into the Opcode Register. However, instead of using the local board signal as a clock enable signal to the Opcode Register, it serves as the clock itself. Only one active edge will occur per WRTORG, then, and only one value latched into the register. The state machine includes a state that waits for the $\overline{DS}$ lines to be negated before returning to the idle state, waiting for new data transfers and new controller instructions. Therefore, address and data rot are no longer an issue. Similar techniques corrected this problem for accesses to other registers.

Chapter 4

Software Application Framework

In addition to the hardware comprising the image processing system for this research, a method for programmers to access this system had to be developed. The requirements for this system included removing the need for the programmer to understand the specific hardware involved; flexibility to run applications on either type of array (CAPP or HDPP); and the ability to run applications with a software system simulator, to test applications for functionality when the hardware system is unavailable, and to debug applications

This chapter will outline such a method, developed by Gealow and Herrmann [2]. This outline is given to give a basis for discussing new demonstration applications developed as a part of this thesis.

4.1 Framework Abstraction Levels

4.1.1 Application Programmer's Interface

The top level of abstraction is the programmer's interface with the system. This level must be flexible enough to not require changes should the hardware be changed. It must also be detailed enough to take advantage of the hardware design, without requiring the programmer to be familiar with that design.

Figure 4-1 shows a top-level view of the programming framework, updated to include the new controller architecture. The programmer specifies, in their application, which type of system (CAPP/HDPP) they are programming for. Alternatively, as applications can be executed on several combinations, the programmer can compile the program for any or all combinations. In any case, once this choice is made, these two components are combined into an object known as the *system.* While the framework is architecturally independent from the standpoint that applications will run on either array type, each array type is tied to a particular controller type, as shown.

In addition, the programmer must declare *fields* of data, which are blocks of memory allocated across the entire array. For example, an 8-bit image would consist of an 8-bit field, where each PE in the array will then have eight bits of its memory set aside for this field. As shown, these fields may be signed, unsigned, or ternary (an additional value for don't cares).

To access the system, the programmer then develops sequences of instructions to be run on the array, to do various image processing tasks. To aid in this task, *code generators* exist for many useful operations in the array, such as adding, multiplying, comparing, etc. As will be shown later, the outputs of these code generators are combined to form a sequence which operates on

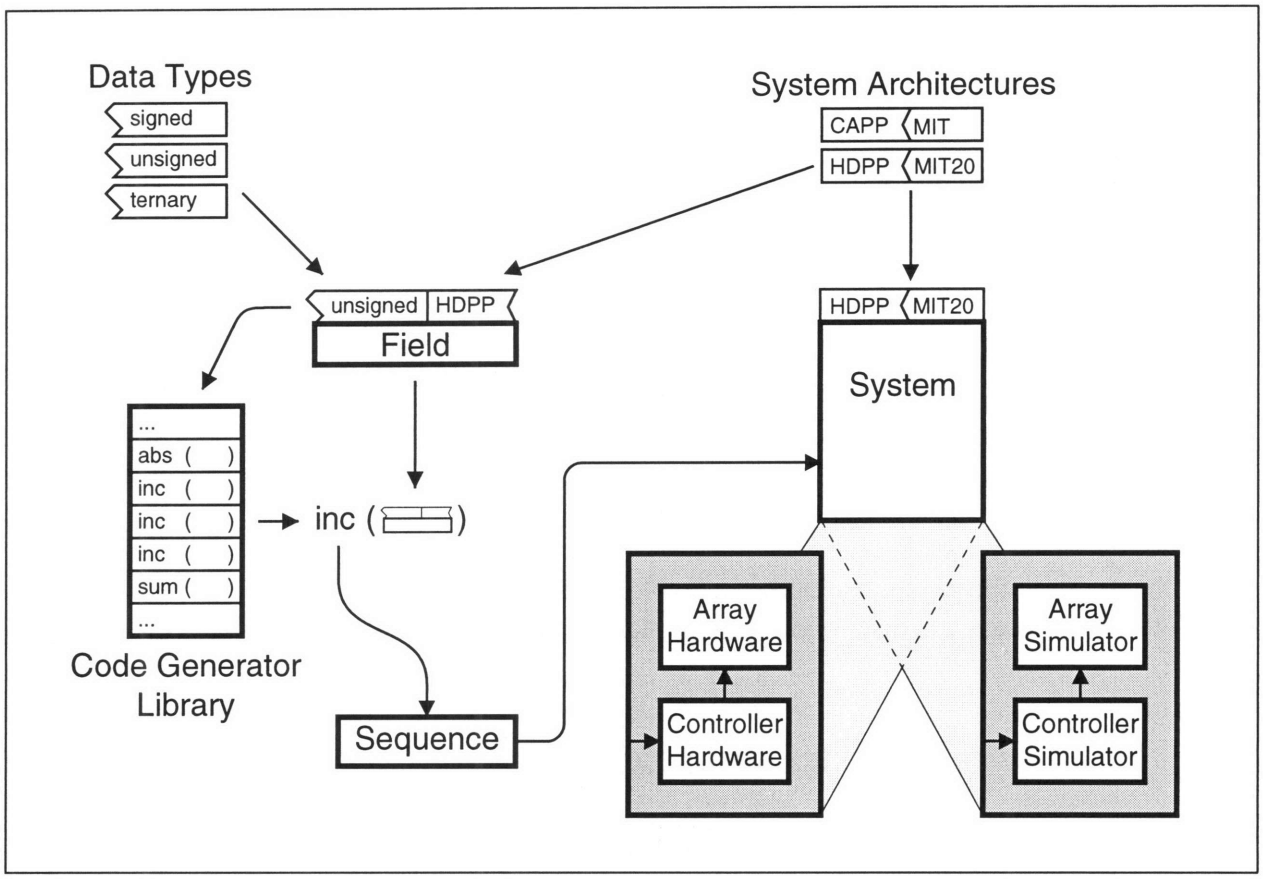

Figure 4-1: Programming Framework Overview

various fields. The result of these operations is itself stored in a field, which can be displayed or output in some other fashion. In this way, the programmer is kept isolated from such details as how each array architecture implements an "add" instruction, how adding of fields of different lengths is accomplished, etc.

As an example, in Figure 4-2, the programmer creates a sequence to add two fields, and store the result in the first field. They then load the sequence into the system hardware (or software, if it's a simulation). Finally, they execute the sequence a single time on the system. This figure shows how key instructions are expanded across the layers of abstraction in the framework. The following sections will discuss the rest of the figure, and how these tasks flow through the layers.

This portion of the programming framework was almost completely unchanged in working on this thesis. The only modification was the creation of the MIT20 class of controller, which is based on the same class as the MIT (10 MHz) controller. Both controller classes have the same interfaces at this level.

4.1.2 Hardware Abstraction Level

The next lower level of abstraction is tied more closely to the hardware. In the controller's case, functions exist to interface with the controller model shown in Figure 2-5. While still relying on a general model of the hardware, this level serves as a wrapper around the actual hardware interface

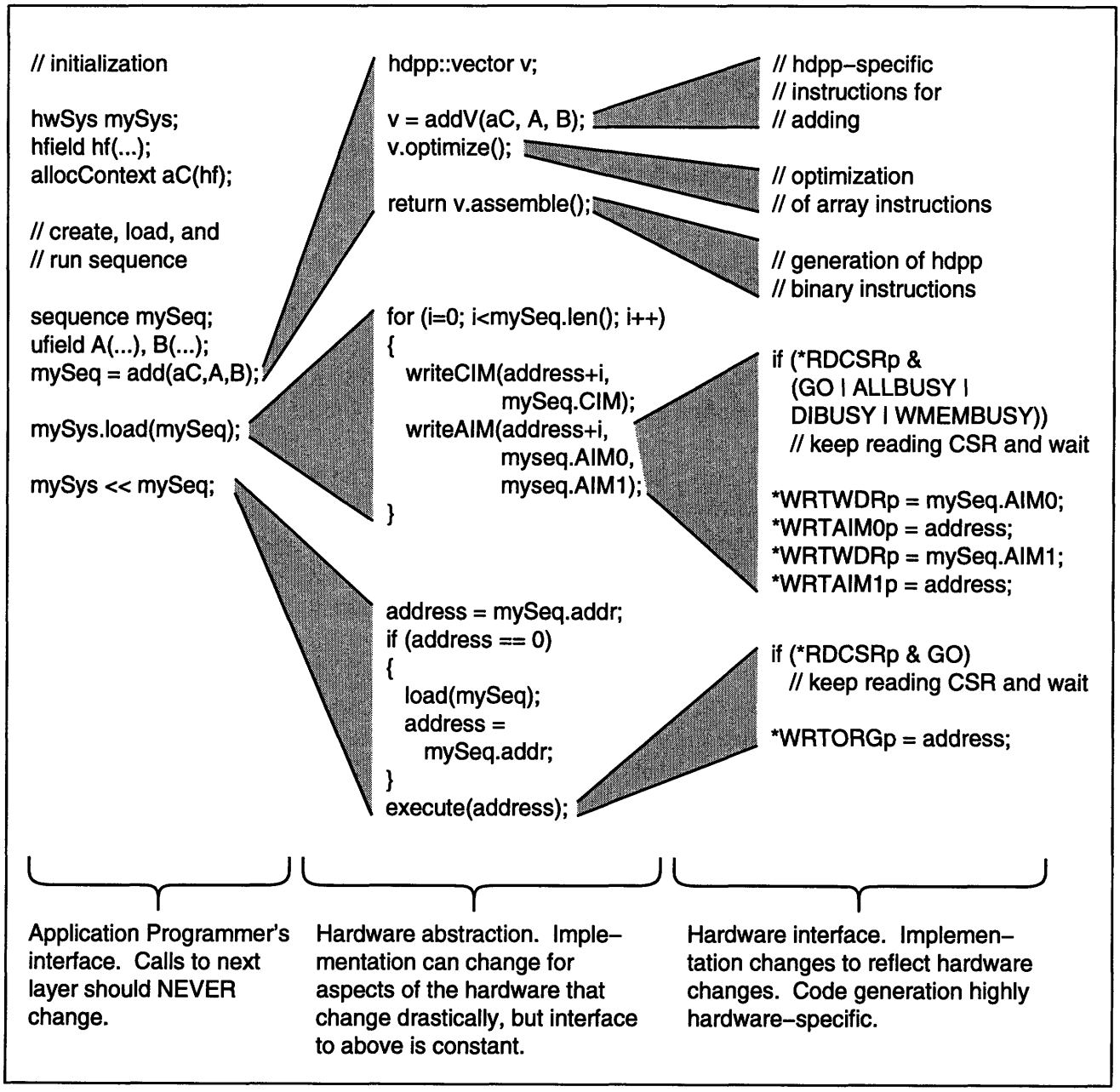

Figure 4-2: Programming Framework Abstraction Levels

functions outlined in Section 2.2.5.

This level gives a generic interface in the software between the *system* object, which encapsulates all of the hardware, and the *controller* object, which is the specific hardware implementation object. It is this layer that allows the underlying hardware implementation to change, without higher-level code being modified. By keeping to a generic interface, code both above and below this interface can be changed -- this is one of the benefits of object-oriented programming.

In Figure 4-2, one can see how an instruction like the *load* instruction interfaces with this level. It is expanded into calls to WriteCIM and WriteAIM, which, while these calls assume a certain

amount about the *model* of the controller's architecture, it is independent of the *implementation* of the model, at the hardware level. The designers of the framework merely had to stay consistent about the programming structures provided by each piece of the architecture.

In designing the new controller, very little was changed in this level of the framework. There were some additions for new architecture, for instance, a refresh sequence is now loaded into the control store as soon as the hardware is initialized with software.

4.1.3 Hardware Interface Level

This is the layer that is completely dependent upon hardware. Tasks at this level include assembling instructions from higher levels into actual binary data that is sent to the controller and array, interfacing with signals from the controller to program the control store, and using the host-controller interface functions to communicate with the controller.

The MIT20 controller class's implementation at this level was heavily changed from the MIT class, due to the changes in hardware design. For instance, as shown in the figure, a WriteCIM now translates to the register operations shown, whereas before, there were many additional register operations (WRTAIML0, WRTAIML1, WRTAIMR0, WRTAIMR1, etc.). This is where the change from 17 interface instructions to 11 impacts software design.

The binary assemblers that create the instructions to be sent to the CIM are also very different, for two reasons. First, the instruction format has changed, as mentioned. In addition, as Chapter 3 discussed, many control instructions (such as conditional branches) now assemble to more binary instructions for the controller than was true with the 10 MHz design. This necessitated many changes in the hardware interface code. However, because of the two levels on top of this level, much like network design, the implementation of this layer can change without impacting higher layers at all.

4.2 Software Simulators

The remaining requirement of the software framework was the capability to simulate sequence execution, the controller's state, and the array's state, in memory, both for debugging purposes and to allow software development without access to the hardware. As part of their research, Gealow and Herrmann also provided this functionality in the programming framework.

In doing this, they took advantage of many aspects of C++, in which the framework (and applications themselves) are implemented. There are two classes that represent the system - hwSystem and swSystem. The programmer need only create an object of one type or the other, and can then run any sequence on this system, ignoring whether it is in hardware or software.

The 20 MHz controller dictated very few changes in the simulation system. Because the simulation of the controller doesn't go all the way to the hardware level (it was more important to simulate the array hardware, and the controller functionality), most of the changes weren't seen in the software. A few changes were made to model the delay in the conditional branches (and therefore

the CCMUX output), but otherwise, the software simulator was largely untouched.

4.3 Application Writing

This section will discuss the programming requirements of an application designed for this system. It will first include an overview of the programming constructs provided by Gealow and Herrmann [2] to facilitate application development. Following that will be an example of how to use the system architecture efficiently, with a rudimentary discussion of parallel algorithms.

4.3.1 Initialization Overhead

When writing the application, there is surprisingly little overhead required, both to choose a system (array, controller, and hardware or software mode), and to initialize the system. In a basic application, the following is the required initialization:

```
#include <xapp.h>              // global system defines
#include <hdpptype.h>          // array defines[1]

hwSystem mySys;                // or swSystem

hfield hf(arch::array::FreeField());
allocContext(aC(hf));          // housekeeping
```

Apart from this overhead, all of the remaining code is up to the programmer, for defining fields, developing sequences, etc. Note that there is only one line (line 2 above) that is architecture-specific. The third line is as shown for an application which will run on hardware, or `swSystem` for a software system. The final two lines are housekeeping, to create an "allocation context," *aC,* which is passed into code generators to produce sequences.

4.3.2 Sequences and Code Generators

As mentioned, applications are typically made up of several "sequences" of instructions, which are stored in the control store. The host then calls these sequences repeatedly, like subroutines, to run the application itself. In keeping with this paradigm, a *sequence* class is part of the programming framework. It supports many member functions, including adding other sequences (and presequences, discussed later) and copying sequences.

Sequences, in turn, operate on classes called *fields.* Fields are objects that represent memory fields in the array. An eight-bit field consists of eight bits of memory in every PE in the array. In this way, operations on a field affect the entire array.

As mentioned, code generators exist to produce sequences on each array architecture, to do common tasks required for image processing applications. A compiled list of these sequences does

1. This is the only architecture-specific portion of code - either hdpptype.h or capptype.h - to tell the compiler which array libraries to include

not fall within the scope of this document, but they include sequences for performing common operations (add, subtract, multiply, accumulate, write, move), as well as conditionals for activating certain PEs in the array (inside, outside, mismatch). As an example of a sequence being constructed with these tools, the following demonstrates a simple edge detection application. While this is a very simple application, it demonstrates how effective the framework is at abstracting the program from the controller hardware, the array architecture, and other details:

```
// (perform initialization overhead here)

// construct fields to be used - 8 bits for the image
// field (unsigned), and 9 bits for fields used to hold
// a difference calculation
ufield m(aC.statically(), 8);       // image
sfield nd(aC.statically(), 9);      // north diff.
sfield wd(aC.statically(), 9);      // west diff.

sequence edge;

// imageio outputs current field to camera, and
// loads in a new image

edge += imageio(aC, m, m);          // m = 8-bit image
                                    // nd = m - North
edge += nbrDifference(aC, m, North, nd);
                                    // wd = m - West
edge += nbrDifference(aC, m, West, wd);

edge += write(aC, m, 0u);           // clear image

// activate only PEs that are outside a threshold
edge += outside(aC, wd, -thresh, thresh);
edge += writeC[1](aC, m, 255);      // write 'edge'

// do same for horizontal edges
edge += outside(aC, nd, -thresh, thresh);
edge += writeC(aC, m, 255);         // write 'edge'

// run sequence at frame rate
mySys.load(edge);
while(1)
   mySys << edge;
```

1. sequences with a 'C' appended affect only the PEs activated by the previous conditional statements. In this case, the 'outside' sequence.

4.3.3 Presequences and Control Instructions

The previous example showed how to use sequences to create an application. However, it did not include any of the previously mentioned control instructions (branches, etc.). To access this functionality of the system, it is necessary to create an application at the *presequence* level. A presequence is similar to a sequence, with the exception that the instructions have not yet been assembled to resolve branch addresses, and no optimization of the sequences have occurred.

In practice, developing a sequence by using presequence is very similar to the earlier example. The framework provides interface functions to wrap around the controller instructions mentioned in Section 2.2.4. The following table shows the functions provided that parallel code generation for the controller:

Presequence	Usage
next()	Continue to next instruction
branch(label)[a]	Branches to a label elsewhere in the sequence
gosub(label)[a]	Gosub to a label elsewhere in the sequence
ret()[a]	Return from gosub
loop(count)	Begins loop, loads loop counter
breakLoop()[a]	Breaks out of the loop
endloop()	Continues looping, going back to beginning
repeat()	Starts a repeat...until sequence
breakRepeat()[a]	Breaks out of a repeat...until sequence
until(polarity, condition)	Ends a repeat...until sequence
jzero()	Jumps to control store location zero. This instruction is automatically appended to every sequence.

Table 4-1: Controller Instruction Interface Functions

a. This instruction also has a conditional counterpart

These instructions exist as presequences because originally, these constructs were not meant for the use of application programmers. They instead are used as part of the array code generators. However, in practice, it has become evident that some programs might want to access them directly. If this is the case, the programmer must build a presequence, and assemble it to a sequence. An example using these instructions appears in Section 4.4.2.

4.4 New Applications

As part of this research, several new sequences were developed, adding to a small library of image-processing tools and demonstration sequences [2]. The algorithms, output, and execution times of these sequences are discussed in this section. Refer to (appendix) for the source code of these sequences, provided as a sample for other programmers.

4.4.1 Edge Detection

Using existing sequences developed by Gealow and Herrmann [2], a simple edge detection sequence can be done to demonstrate both the flexibility of this type of development system for rapid prototyping of a concept, as well as the extremely low processing time necessary for a useful application.

The algorithm for this sequence is very simple. The basic concept is taking the value of pixel A, and comparing it to its neighbor. If the difference is above a certain threshold, then this boundary is marked as being an edge. This is the basic sequence shown in Section 4.3.2. However, in noisy images, this produces garbled edges, and is not a finished product.

Using smoothing and segmentation, this algorithm can yield better results. Smoothing and segmenting serves to smooth out an image's pixels, but will not smooth across boundaries of some threshold. This tends to remove noise from an image, while preserving edges. Taking the output of this sequence and running the previous edge detection algorithm gives much better output. A simple modification to the algorithm can also add the ability to record only vertical or horizontal edges. This is done simply by comparing only to vertical or horizontal neighbors, in this example. A more sophisticated approach could be used to show the intensity of edges, but is beyond the scope of this example.

A sample, unprocessed input image is shown in Figure 4-3. The results of this edge detection sequence are also shown with various amounts of smoothing and segmentation, and detecting various edges. As can be seen, smoothing and segmenting can make a difference in removing noise from the resulting edges.

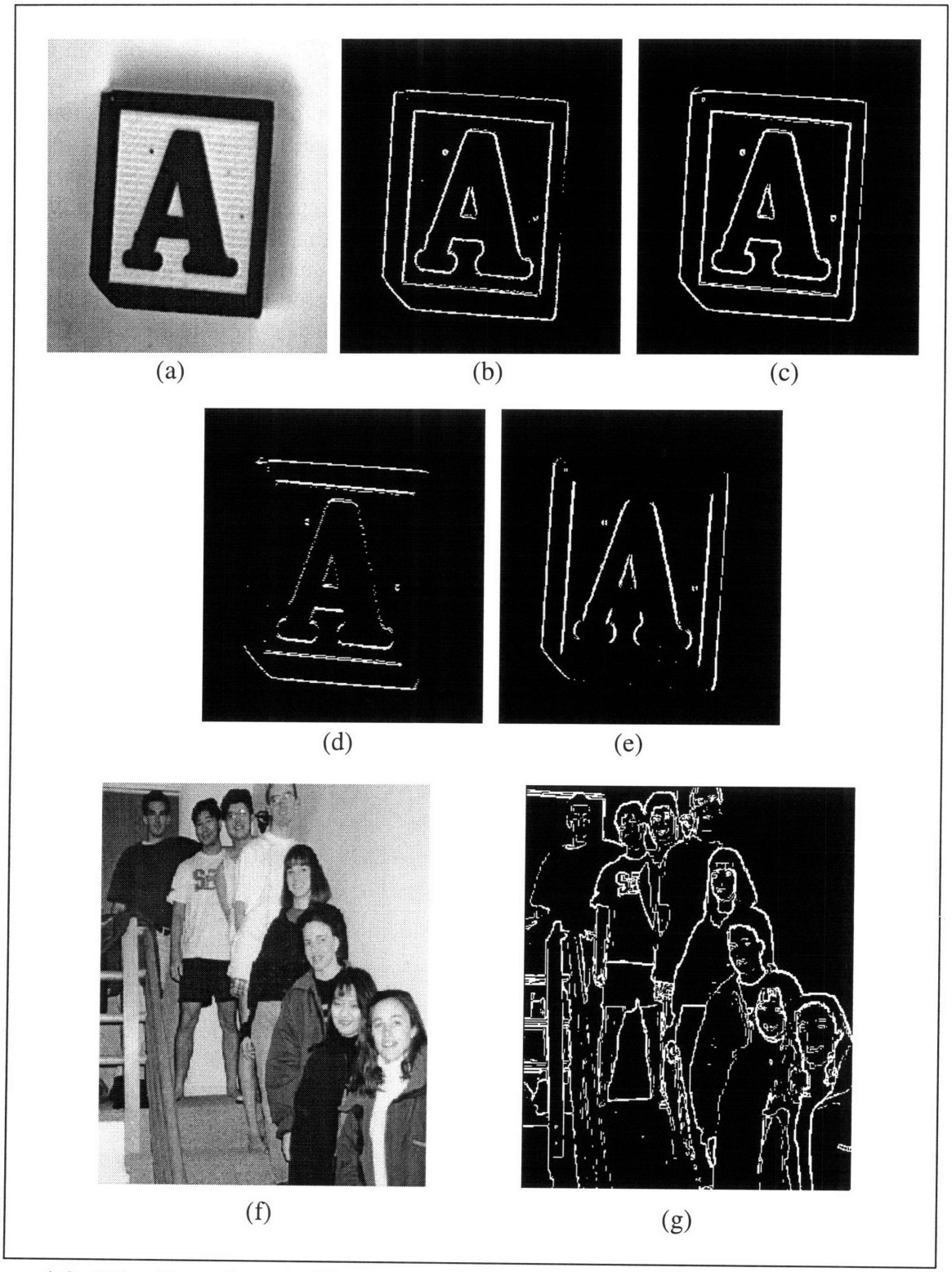

Figure 4-3: Edge Detection. (a) Unprocessed image. (b) Edge detection with no smoothing or segmentation. (c) Edge detection with 16 convolutions with segmentation threshold of 16. (d) Horizontal edges only. (e) Vertical edges only. (f) More complex original image. (g) 8 convolutions with threshold 24.

4.4.2 Template Matching

Another sample application well-suited to this parallel architecture is template matching. This is taking an input image, and searching it for some template image. The goal of this demonstration is to show how efficiently this architecture can be used to do complicated tasks in a short time.

The core of the algorithm designed during this research is simply to compare every region of array memory with an arbitrarily-sized template. The basic steps taken are shown in Figure 4-4. The comparison is to subtract each pixel in the region from the corresponding pixel in the template, and accumulate these differences. If the accumulated sum is below some threshold, it is considered a match. This algorithm will not attempt to correct for average brightness levels, template rotation, or template scaling. Building on this basic sequence, these features could be added as desired.

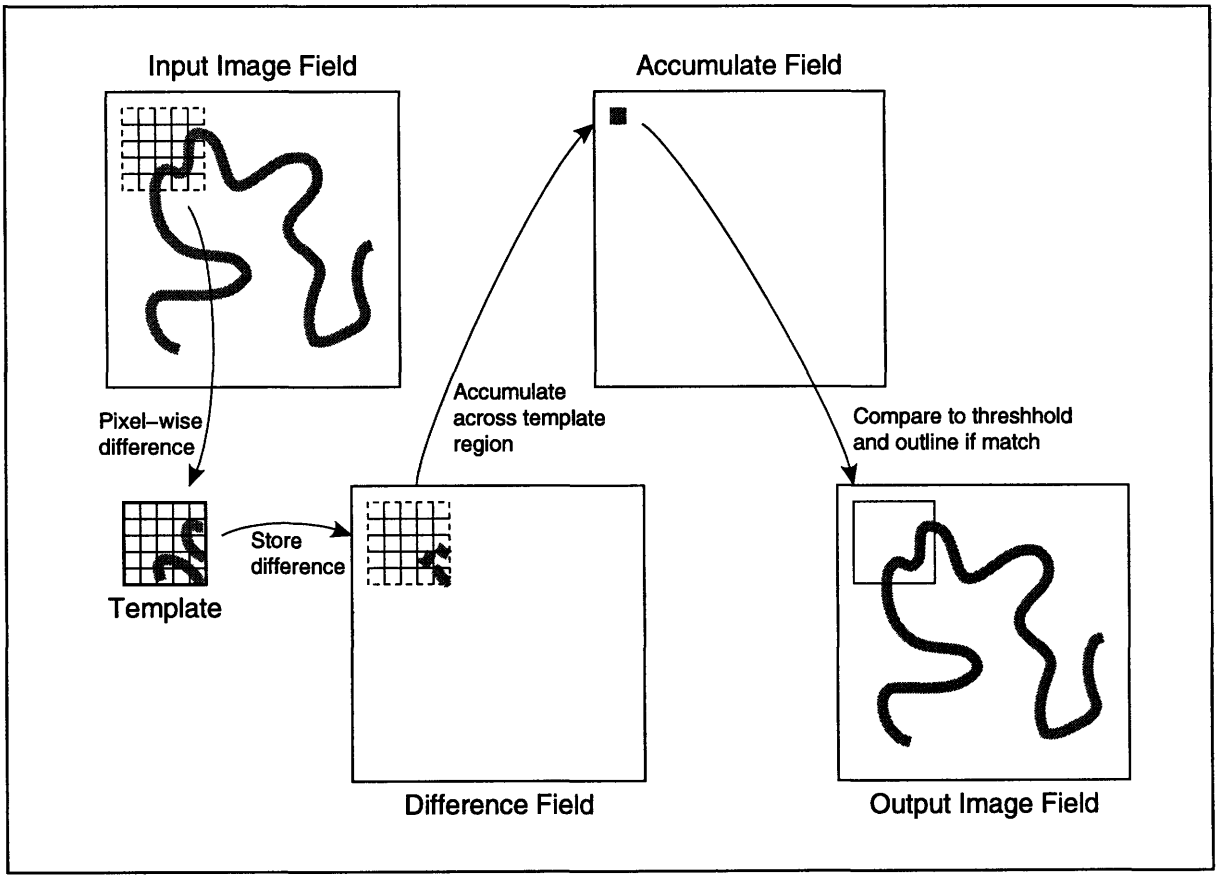

Figure 4-4: Template Matching Algorithm

Based on this idea, it is clear that the number of subtractions for each comparison will be N*M, for an N x M pixel template. This comparison will have to be done at every location in the array, or P*Q times, for a P x Q pixel array. For a sample case of a 256 x 256 array, with a 50 x 50 template, this is 163,840,000 subtractions (for comparing the template with the image), followed by 163,840,000 additions (for accumulating the differences). In a general-purpose computer doing sequential operations, this is an addition or subtraction operation every 90 ps, assuming a compar-

ison is done every 30 ms (frame rate). Clearly, this is an application that would benefit from, and in many architectures require, parallel processing.

The algorithm itself is not a new idea. However, a method of implementing it in a way to take advantage of the parallel architecture presented here required several novel approaches. Three different methods for generating this algorithm were explored, in an attempt to compensate for the lack of run-time flow control in this architecture. Together, they demonstrate methods of utilizing the parallel architecture, using the control instructions, and bypassing the need for run-time flow control. Each of the following three methods implements exactly the same task discussed above.

The first method was optimized for speed, designed to minimize the time that the sequence would take to execute. It would iterate through each pixel of the template, taking the difference between that pixel and the PE's pixel. It would then accumulate the difference with the previous sum, and continue on through the rest of the template. This was done in a parallel fashion, as shown in Figure 4-5, where the template is compared with EVERY pixel in the array at once. These differences are added to the accumulated differences at that location. Then the accumulations are shifted around in the array, and the process continues for each template pixel. In this way, the algorithm takes full advantage of parallelism, with no repeated operations.

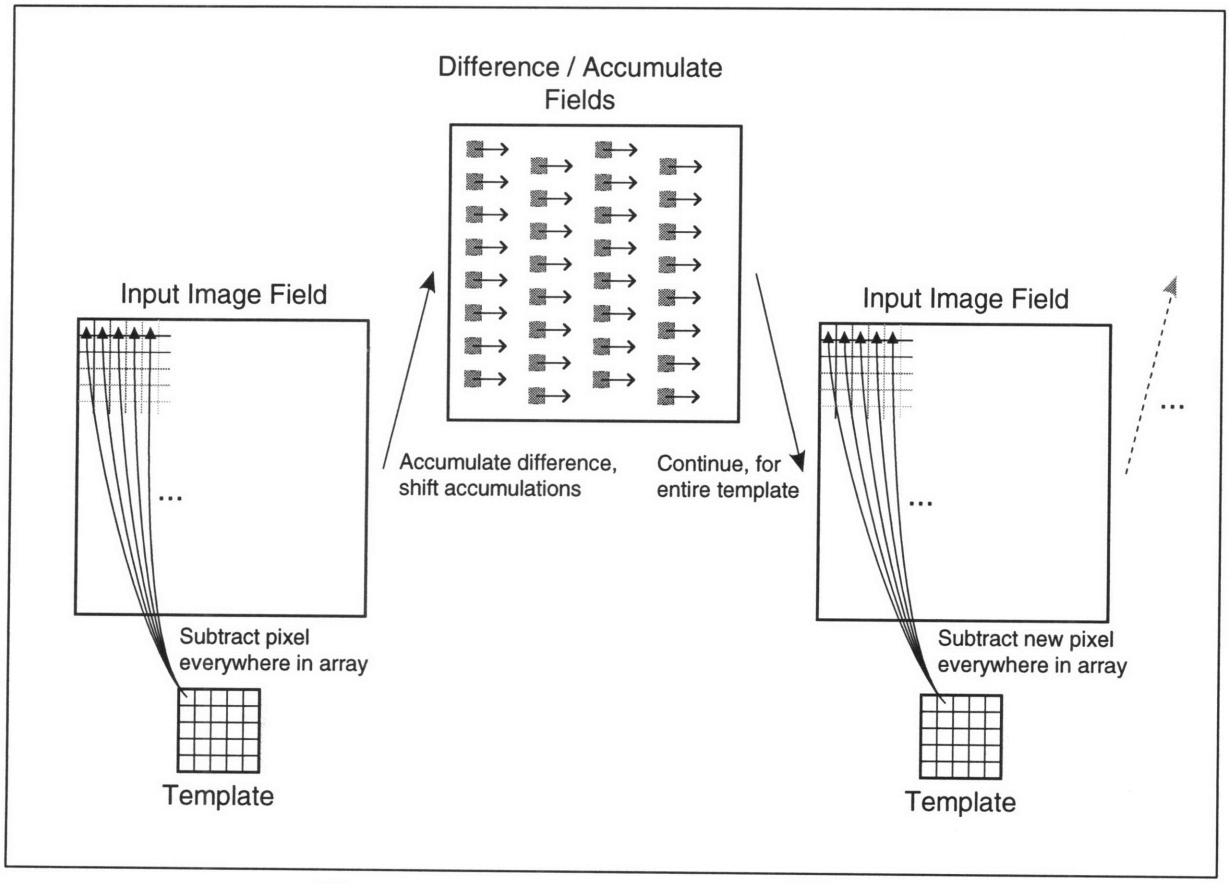

Figure 4-5: Parallelizing Template Matching

Because there is no fast way to supply run-time data to the array, the pixel value in the template would be written into the instruction sequence, so that all processing was done at compile time. This method would execute extremely fast, because there is only one sequence in memory (no waiting for launches), and there are no branches in its execution. It is also the most straightforward implementation. However, each pixel in the template resulted in a move sequence, a difference sequence, and an accumulate sequence. Inspecting the resultant instructions generated from this shows that each group of these sequences was roughly 160 array instructions. For a 100 x 100 template (the desired maximum size), this is 1,600,000 array instructions, far exceeding the storage capacity of the control store. This method was limited to (64,000 / 160) = 400 pixels, or very small templates, with no control store space remaining at all for other operations.

This necessitated a second method, more optimized for space. It was similar to the previous method, in that it traverses a spiral path around the template, calculating a difference at each pixel. However, the implementation here was to use a single sequence for all pixels, instead of a one-to-one basis. The previous method had a portion of the sequence for each template pixel, with the particular value of that pixel to be subtracted. If a generic sequence could be written, with the value to be subtracted supplied at run-time, the sequence size would be independent of template size. The cost is more sequence launches (one per pixel), which has an overhead cost.

This was done by writing a value into the SSR, once per pixel. A special write sequence then writes this value into a temporary field, and this field was subtracted from the image field. In this way, one sequence can be reused for all pixels in the template. This method was very space efficient, but for a 100 x 100 template, required 10,000 SSR writes and sequence launches. This method is too inefficient to be a useful operation at frame rate. However, for very large templates, this method will suffice for non-frame-rate comparisons.

The final implementation was a compromise between these two. First, 256 sequences were created, one for each possible pixel value in the template. Then, for each pixel in the template, a gosub to one of these 256 sequences was used, based on it's value. In reality, there were 4 * 256 sequences, since for each pixel, the template needed to be moved in one of four directions. In this way, a main sequence was built up of M x N gosubs (for an M x N template), where each of the 1024 sequences is a move, difference, and accumulate, as before. This results in a single sequence of gosubs, which is time efficient. However, this yields 1024 sequences of 160 instructions each, or over 163,000 instructions -- still too large. So the mini-sequences were again broken down, into a four sequences to move (one in each direction), 256 sequence to write a pixel into a field, and one sequence to subtract this field and accumulate the result. This results in 256 sequences of length 2, 4 sequences of length 60, one sequence of length 100 for the subtract/accumulate, and M x N gosubs (unconditional, so no expansion occurs). For a 100 x 100 template, this is 10,852 instructions. This is the final method used, and results in efficient operation where the sequence size is roughly proportional to the template size, and small enough to allow other processing to fit in the control store.

Figure 4-6 shows the results of template matching with a template of 20 x 20 pixels. This template, produced by cropping part of the original image, is shown in the figure. With a threshold of only 100, only the exact match is found. With a higher threshold, the second image shows multi-

ple matches beginning to appear.

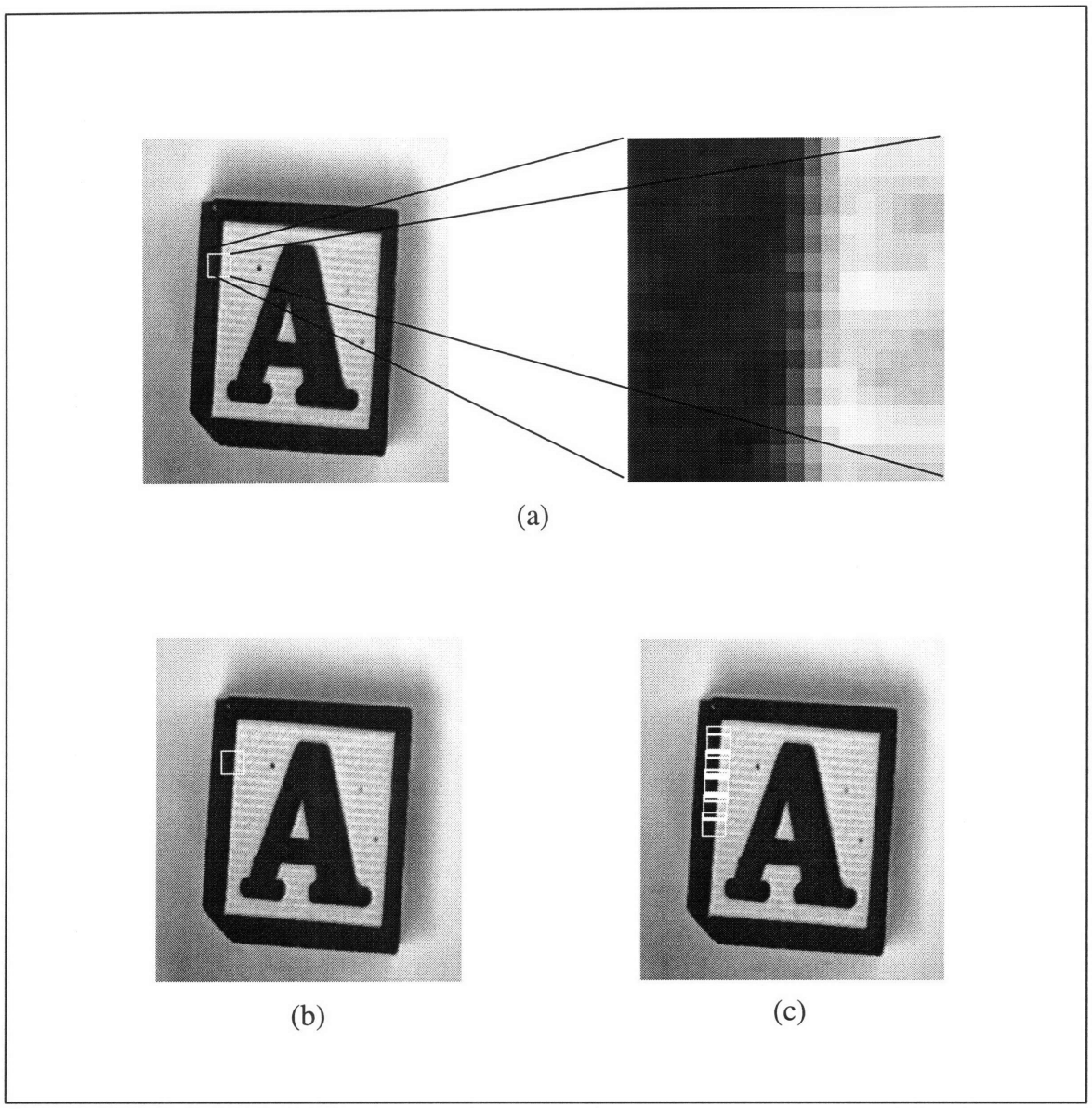

Figure 4-6: Template Matching. (a) Template being searched for.
(b) Match threshold of 100. (c) Match threshold of 1500.

As mentioned, this algorithm would need to have further enhancements to be a final product. Scaling and rotation of the template should be accounted for. Ambient light levels could be removed using some form of laplacian or edge detection. However, this method shows the power of the system to perform this type of task efficiently, and can be modified by other programmers to include these enhancements.

Because these methods clearly demonstrate several methods of writing image processing applications (sequences, presequences, flow control, parallelism), application code for each of these sequences are shown in Appendix D.

Chapter 5

Testing and Results

The logic for the EPLDs was done using Xilinx XACTStep software in combination with View-Logic's PowerView. This logic was verified using these software packages, as well, and timing data extracted. The board schematics were also done in PowerView. The layout and fabrication was send to Advanced, Inc. which produced a six-layer PCB and assembled it.

Programming of the UV-erasable EPLDs was done with a Xilinx HW-130 programmer, using data files from XACTStep. The controller was tested in the PT-SBS915 VME-to-SBus chassis, using the external clock input from an HP 8161A pulse generator, an HP1650A logic analyzer, and several digitizing oscilloscopes. The chassis provided power to the controller.

This chapter discusses the notable problems that occurred during testing, and the final results of testing the controller.

5.1 Board Production Problems

Most of the problems in board production stemmed from some poor choices made in getting the board fabricated and assembled. The company chosen decided to re-enter the schematics prior to doing the routing and fabrication of the design. Because of this, several errors in duplicating the schematics were made, and not all of these were found. Two that remained were signals from the VMEbus chassis not being connected to the EPLDs.

Once these problems were corrected, a serious flaw in the fabricated boards became apparent. Under the PLX2000 chip, the $\overline{\text{MODSEL}}$ signal had been shorted to VDD. Because of the previous error, this signal was never being asserted low. Once that problem was fixed, the signal went low, and a short to ground occurred, burning a trace visibly. This error was found to be a mistake of the fabrication company, and the remaining boards were fixed.

At this point, the most serious error was found. A symbol was reused incorrectly from a register chip in another design, resulting in pins being misnumbered. This error occurred on 16 pins of 10 chips, and was not an error that could be fixed. This resulted in a second fabrication of the controller, during which the previous errors were also remedied.

The last problem was found after this fabrication. It was found to be necessary to fit some extra logic in the Data Registers EPLD, to correct a problem with the interrupt controller. This caused two pins on the pinout of this chip to change. To minimize the impact, the previous pins were changed to No Connects, and a wire added between the No Connects (where the board's signal was routed to) and the EPLD pins, where the logic actually resided. For reference, the pins sol-

dered together were F2 to N3, and R13 to P4.

5.2 Clocked Testing

Initial testing was done at 10 MHz, to isolate logic problems from critical path problems. The first test in checking the timing requirements of the controller was to increase the clock frequency. Testing was done with a simple program written in previous research, that repeatedly launches sequences, reprograms the control store with longer sequences, and continues. This tests all aspects of the critical path (a conditional branch, and a JMAP).

The controller functions as expected at 20 MHz. Frequencies of 12, 15, and 18 MHz were also done. To test the critical path of the current controller, the clock rate was increased to 25 MHz. While the controller still functioned, the logic analyzer begins to show that certain outputs are becoming skewed. For instance, the I inputs to the sequencer are arriving only several nanoseconds before the clock edge. For this reason, the maximum *documented* safe speed for the controller will be 25 MHz, although it will function at higher speeds with decreasing stability.

The other aspect of clocked testing was testing lower clock rates, to insure that the data rot problem had been fixed. Testing was done in 1 MHz increments from 10 MHz to 1 MHz, with varied results. The controller functioned well at 3 MHz. Below this frequency, intermittent problems occurred, with increasing incidence as the clock rate slowed. These problems were characterized by the controller ending up in incorrect states while performing memory writes, which left certain CSR bits high, preventing other instructions from being sent. This problem got steadily worse, until at 150 kHz, the host interface failed completely, and was unable to send any instructions to the controller (core dumps occurred).

After analyzing these problems, two limitations were uncovered. The first problem was with the VME chassis interface itself. The VMEbus is an asynchronous bus, which is connected to the Sparc's SBus, a synchronous bus. The SBus being used for this research has a clock frequency of 20 MHz. Part of the SBus specification is a bus time-out error, that occurs after 256 clock cycles (12.8 us @ 20 MHz). This means that if the controller fails to acknowledge the SBus data transaction through the VMEbus, the host aborts the transfer and a core dump results. Table 5-1 shows the delays involved in this acknowledgment:

Event	Delay
SBus $\overline{AS} \Rightarrow$ VMEbus $\overline{DS}$ (PT-SBS915 delay)	220 ns
VMEbus $\overline{DS} \Rightarrow \overline{ADEN}$ (EPLD asynchronous delay)	18 ns
$\overline{ADEN} \Rightarrow \overline{MODSEL}$ (PLX2000 delay)	45 ns
$\overline{MODSEL} \Rightarrow$ PLX2000 $\overline{DS}$ (PLX2000 delay)	45 ns

Table 5-1: End-to-End Bus Transaction Delays

Event	Delay
PLX2000 $\overline{DS} \Rightarrow \overline{ACK}$ (EPLD synchronous delay - allow data latching)	2 clocks
$\overline{ACK} \Rightarrow \overline{DTACK}$ (PLX2000 delay)	45 ns
$\overline{DTACK} \Rightarrow$ SBus $\overline{ACK}$ (PT-SBS915 delay)	100 ns

Table 5-1: End-to-End Bus Transaction Delays

As shown, we must satisfy the following:

$$220\text{ ns} + 18\text{ ns} + 45\text{ ns} + 45\text{ ns} + 2\phi + 45\text{ ns} + 100\text{ ns} \leq 12.8\text{ us}$$

or

$$2\phi \leq 12.327\text{ us}$$

where ϕ is the clock period. This dictates that to avoid bus errors, the clock frequency must be at least 163 kHz, which is verified by testing.

The more serious problem was in the 1-2 MHz range. While this is not likely to affect the system performance (the array is nearly guaranteed to work at faster speeds), this problem must be explained. The problem arose because of the latching of an asynchronous signal prior to its being used in state machine logic in an EPLD (the $\overline{MODSEL}$ signal from the PLX2000). The following is the output from a single memory-write of the control store on the logic analyzer, with the controller running at 500 KHz:

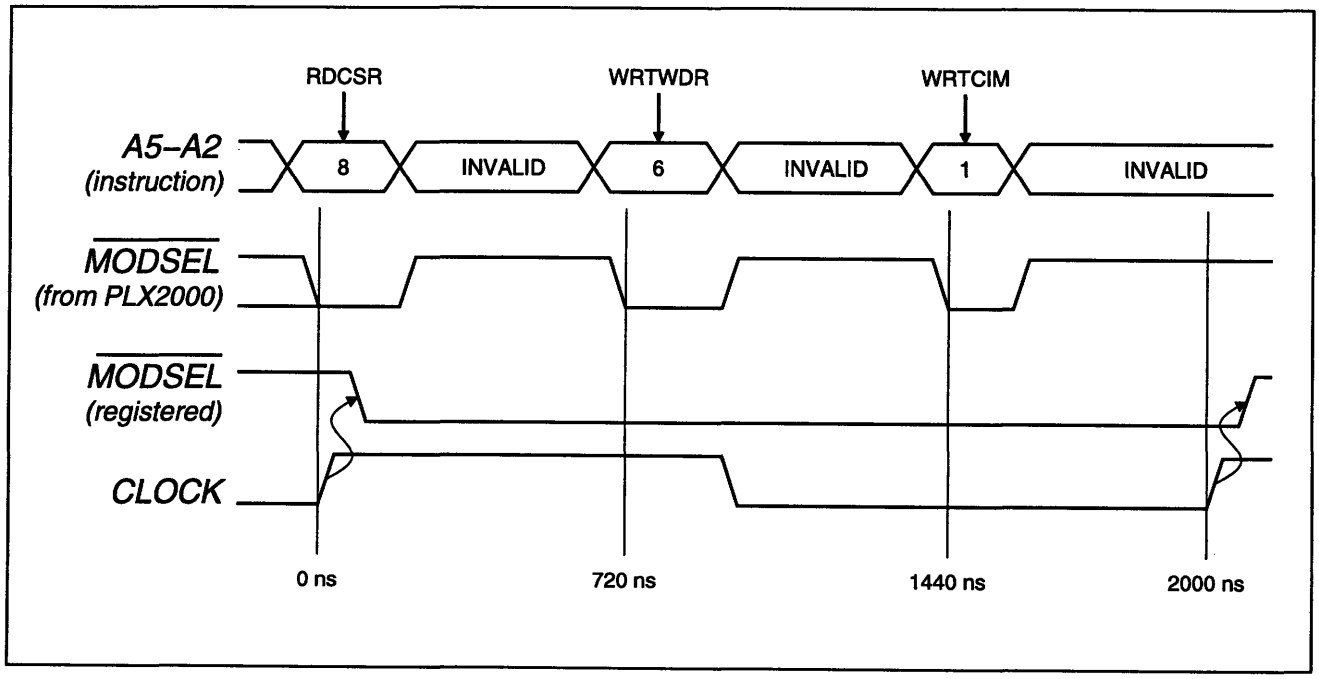

Figure 5-1: VME $\overline{MODSEL}$ Timing Problem

This error was not noticed immediately, because it was not thought to put a test pin on the registered value of $\overline{\text{MODSEL}}$. Once this trace was probed, the solution is evident. The minimum time observed between VMEbus cycles was 720 ns. This agrees with times recorded during design of the previous controller. The problems stemmed from the fact that the registered $\overline{\text{MODSEL}}$ was missing cycles altogether. This resulted in very unpredictable behavior on the controller, with a very high probability of ending up in a random state during memory writing. To remedy this, the clock must have a frequency at least twice that of the fastest signal in the system being registered. In this case, the maximum clock period is 360 ns, or 2.8 MHz. To stay well away from this failure point, because a minimum time between VME cycles is not documented, a minimum frequency for this controller will be specified as 4 MHz.

5.3 Interrupt / Memory Write Conflict

Two other serious problems were discovered during testing of the interrupts. At first glance, the circuit worked perfectly, with the refresh occurring right on time, and sequences running in between. However, upon debugging the slow clock problem in the previous section, a bug was noticed in the operation of the CSR bits controlling memory writes, where interrupts get involved.

The first problem arises when a memory write occurs during a refresh. There are two cases this can occur. The first is when an interrupt occurs during a sequence. In this case, ALLBUSY is already set by the controller, and the memory write will be pending until ALLBUSY is cleared, at the end of the sequence. The second case, when a refresh occurs during the idle loop, is the problem. In this case, the ALLBUSY flag is *not* set. It was decided early in the design not to have the interrupt set ALLBUSY, because then it was unclear how to tell whether to reset this signal at the end of the interrupt -- it should be done if the refresh occurred during the idle loop, but not if the interrupt came during a sequence. This causes a problem, because in this case, the memory write will not realize an interrupt is being serviced, and will proceed with the memory write. The effect of this logic is that the write will be done correctly, but the sequencer will end up in the idle loop, with the remainder of the refresh sequence never occurring. This would cause some portions of memory to not be refreshed.

The solution to this was implemented in software. It was decided to set ALLBUSY at the very beginning of the interrupt. The clearing of the bit, rather than being done at the end of the interrupt, is now added to instruction 0, in the idle loop. This fixed the first problem.

The second problem was more subtle. This occurs in the case where a memory write is requested during the idle loop. The CSR is read, and assuming no interrupt is going on, returns a successful value. At this point, a refresh could occur, and soon after that, the memory write will occur, since at the time the CSR was read, ALLBUSY was false. Part of this problem was dealt with in the state machine design of the EPLD, where the memory write logic will not carry out a memory write if the controller's local $\overline{\text{REFRESH}}$ signal is active (low). This was done to prevent this problem. However, the interrupt circuitry was designed in such a way that when the CCR counter reaches zero, an interrupt request is generated. When this request bit and the INTEN bit are both

true, $\overline{\text{REFRESH}}$ is asserted, and the refresh begins. The $\overline{\text{REFRESH}}$ signal remains true for at least one cycle (all that is required to begin the interrupt), and is unasserted when the INTEN bit is true after this point (this is how the IAK bit in the instruction word was eliminated). The problem comes about because the INTEN bit is true during the interrupt sequence itself, resetting $\overline{\text{REFRESH}}$, and allowing the memory write to occur, with the same result as the previous problem.

The solution was again in software, disabling interrupts during the jump vector and the interrupt sequence itself. This has the effect of keeping INTEN false, and $\overline{\text{REFRESH}}$ stays asserted during the entire refresh routine. This was verified to work correctly to solve this problem.

5.4 Performance Analysis

The final testing was to analyze the performance of the controller, to see how effective the clock rate increases were, taking into account the expansion of certain instructions and any other new overhead introduced. Figure 5-2 shows data taken from the workstation, outlining how long it takes to execute sequences of varying lengths on the controller. It also shows what length sequences should be, to keep the array busy as much as possible.

The first graph demonstrates two characteristics of each controller. The first is the latency in launching a new sequence, seen on the flat portion of the graph. A sequence launch requires two bus cycles - one for the RDCSR and one for the WRTORG. The old controller would acknowledge a bus cycle in as much as 250 ns [3], whereas MIT20 has a maximum time of 100 ns. This adds up to 300 ns differential, for two cycles. The graph shows a difference of approximately 460 ns. We believe this extra savings in time is due to newer versions of the compiler and operating system running the MIT20 versions of this diagnostic, which result in less overhead.

The second thing shown on the first graph is the direct effect of the faster clock. As sequence length increases to the point where sequences cannot be completed between bus cycles, it becomes clear that the 20 MHz controller is almost exactly twice as fast at executing these sequences. For example, at sequences of length 768, the 10 MHz controller took 76.042 us to execute, while the 20 MHz controller took 38.9375 us. The new design executes the same sequence in 51.2% of the time. Other data points show percentages as low as 50.2%, while these percentages do not vary directly with sequence length. This discrepancy is because the program gathering these numbers does not have time resolution to this depth, and approximates it by running the sequence many times, and averaging.

The second graph shows a measure of how long sequences must be to obtain a desirable array utilization percentage. As one would expect, with a clock twice as fast, the sequences need to longer to keep the array busy. With long sequences, it can be seen that doubling the sequence length on MIT20 yields roughly the same utilization as the originally sequence on MIT. For the new controller, sequences over 64 instructions long will execute efficiently.

Note that this comparison was done without enabling interrupts, to provide a clear comparison.

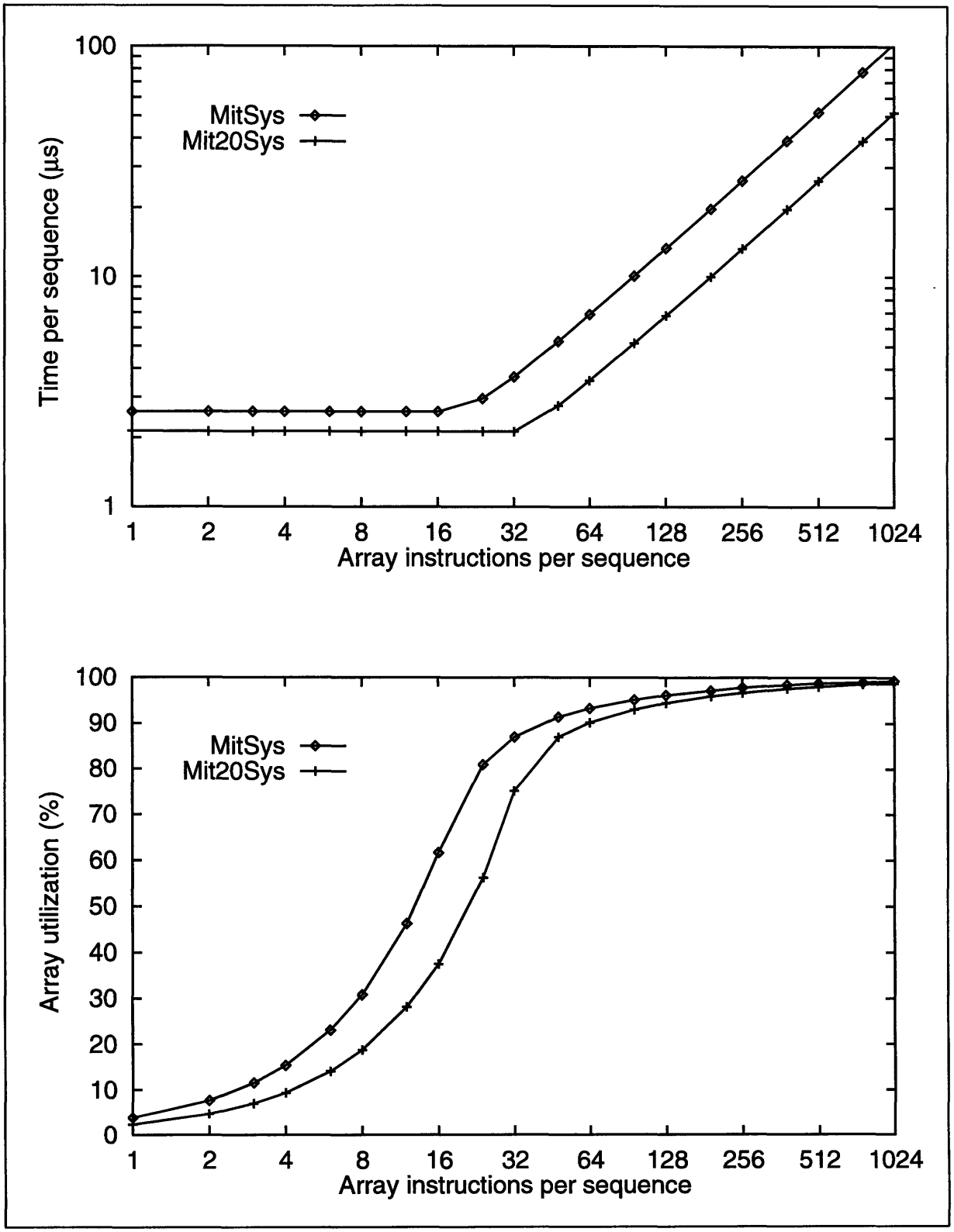

Figure 5-2: Controller Performance Comparison

However, it is obvious that interrupts add a static overhead cost, which is 129 clock cycles every

100 ms (the default interrupt interval), or .0065% at 20 MHz.

Table 5-2 shows some statistics reported by the software framework. The statistics on the MIT (10 MHz) controller are taken from previous work [2]. This shows that these operations can be done well within frame-rate on this system. These types of sequences would be useful as tools in generating still higher-level algorithms, using this output as their input. It also shows that, as expected, the execution times of the sequence are almost exactly cut in half. The times are slightly less than 50 %, due to the decreased latency of launching sequences shown by the previous performance graphs. There was no significant penalty seen by the extra inefficiencies introduced by the software-hardware trade-offs taken. The array utilization is slightly lower, as one would expect with sequences being executed faster than before, but utilization is nonetheless satisfactory.

Sequence	Execution time (MIT)	Array Utilization (MIT)	Execution time (MIT20)	Array Utilization (MIT20)
Smoothing and segmentation[a]	4.3 ms	99 %	1.56 ms	98.5 %
5 x 5 Median filtering[b]	495 us	99 %	201 us	99.2 %
Optical flow[c]	16.1 ms	91 %	7.36 ms	87.9 %
Edge detection[d]	n/a	n/a	207 us	98.9 %
Template matching[e]	n/a	n/a	3.25 ms	99.4 %

Table 5-2: Application Performance with HDPP Array

a. 7-bit image, 100 convolutions, threshold 16.
b. 7-bit image.
c. 8-bit image, 5 x 5 support region, 7 pixel maximum displacement.
d. 7-bit image, no segmentation, vertical and horizontal edges.
e. 7-bit image, 20 x 20 template, threshold of 100.

Chapter 6

Conclusions

6.1 Accomplishments

A new control path board was built, optimizing for a new array architecture. This controller is capable at executing instructions as fast or faster as the maximum speed the HDPP array architecture is expected to be able to receive them, at 20 MHz. The design also eliminated the stability problems exhibited by the previous 10 MHz design.

At the same time, the design was created keeping in mind software compatibility with earlier designs, minimizing the impact of changes to be made elsewhere. In several instances, examples of choosing trade-offs between software solutions and hardware solutions were shown, illustrating the strength of a system comprised of both, with flexible layers of abstraction to allow implementation changes without impacting the overall system.

Doing the board as a printed circuit board with surface-mount components allowed more integrated parts to be used, and provided a board with less noise and clocking problems associated with long traces. Signal termination techniques applying to PCBs were applied successfully for clean clock lines and other key signals throughout the design.

EPLDs showed many of the same advantages as FPGAs in allowing flexible system design with device integration. The choice to use EPLDs was warranted in the case of the control logic, where logic with many inputs was still possible in very short times with this type of logic. However, for the Data Registers EPLD, more integration might have been possible using an FPGA. In this design, the Data Registers EPLD contained relatively low-speed requirement devices, which were not part of any critical paths. An FPGA could probably have implemented this logic without impacting the 20 MHz specification, and allowing integration of the registers that remained on the controller board. However, this would have necessitated programming two different types of logic, and as the board had space for the external registers, using EPLDs was not an inappropriate choice.

As has been noted in previous PCB designs, this choice of board design does have a disadvantage as far as test nodes. With surface-mount components, it is difficult (even with test clips) to clip onto a chip for probing, and instead designers place test pins throughout the board. This is a rather inelegant solution, and can cause problems on a very dense board design. In addition, if there are errors in wiring, a signal may not lie on a visible layer of the board, making fixes very difficult if one is unlucky. However, due to the benefits of integration and noise reduction, this type of board is still preferable to a wire-wrap board.

Some additional sequences were developed to demonstrate the system's capability, including a primitive edge detection and template matching. These sequences exist for demonstration only, but successfully show the types of problems that could be solved with a more complete program.

6.2 Future Work

The hardware portion of this research is likely to be the final incarnation of the controller for this system. It is not planned to produce a faster array architecture, and if chips that are merely more dense with more PEs per chip are produced, this controller will still be able to interface with them.

Some work could be done in refining the software portion of the system revolving around the controller. It would be preferable to have a clean sequence interface to the controller instructions, rather than using presequences. In addition, with the removal of the response resolver, there is no all-purpose solution to the problem of program flow determined by run-time data, only specific solutions to a certain algorithm. There is no proposed solution to easily remedy this problem, and as of now, these problems do not prevent any proposed applications from being written. From a purist point of view, however, future work could address this situation.

Also, there is work to be done in the area of applications. For example, the template matching was very useful in demonstrating the method in which an algorithm can be made to take advantage of parallel processing. However, from a practical point of view, comparing raw images simply doesn't yield reliable matches for images of any interesting complexity, because things like ambient light level are not taken into account, nor are edges. A more interesting application could be developed using more sophisticated methods, perhaps combining the existing sequence with a Laplace transform and an edge detection sequence before the comparison. The edge detection shown in this thesis can also be greatly improved upon, using any of the many documented mathematical methods for edge detection -- however, this would not serve to demonstrate this system any better for this research, and was not done here.

References

[1] Lewis W. Tucker and George G. Robertson. "Architecture and applications of the Connection Machine." *Computer*, 21(8):26-38, August 1988.

[2] Jeffrey C. Gealow, Frederick P. Herrmann, Lawrence T. Hsu, and Charles G. Sodini. "System Design for Pixel-Parallel Image Processing." *IEEE Transactions on Very Large Scale Integration (VLSI) System*, 4(1):32-41, March 1996.

[3] Lawrence T. Hsu. "A Controller Architecture for Associative Processors." Master's thesis, Massachusetts Institute of Technology, August 1993.

[4] Peter J. Osler. A prototype content addressable memory system. Master's thesis, Massachusetts Institute of Technology, June 1987.

[5] Frederick P. Herrmann. A system architecture for associative memories and processors. Master's thesis, Massachusetts Institute of Technology, May 1989.

[6] Motorola. *The VMEBus Specification*, 1985. Rev. C.1.

[7] Daphne Y. Shih. "A Datapath for a Pixel-Parallel Image Processing System." Master's thesis, Massachusetts Institute of Technology, October 1995.

[8] Integrated Device Technology, Inc. IDT49C410 data sheet, 1994.

[9] Xilinx. *The Programmable Logic Data Book*, 1994.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1995.

[11] Howard W. Johnson and Martin Graham. *High-Speed Digital Design.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1993.

[12] Jon P. Wade and Charles G. Sodini. A ternary content addressable search engine. *IEEE Journal of Solid-State Circuits*, 24(4):1003-1013, August 1989.

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, second edition, 1991.

Appendix A

Test Points / Jumper Positions

In building the controller, many test points were inserted to allow key signals to be probed. Figure A-1 is a summary of the test points on the board specifically provided for this purpose. Also shown are the three jumpers, used to control the clock driver circuit, and its position explanations.

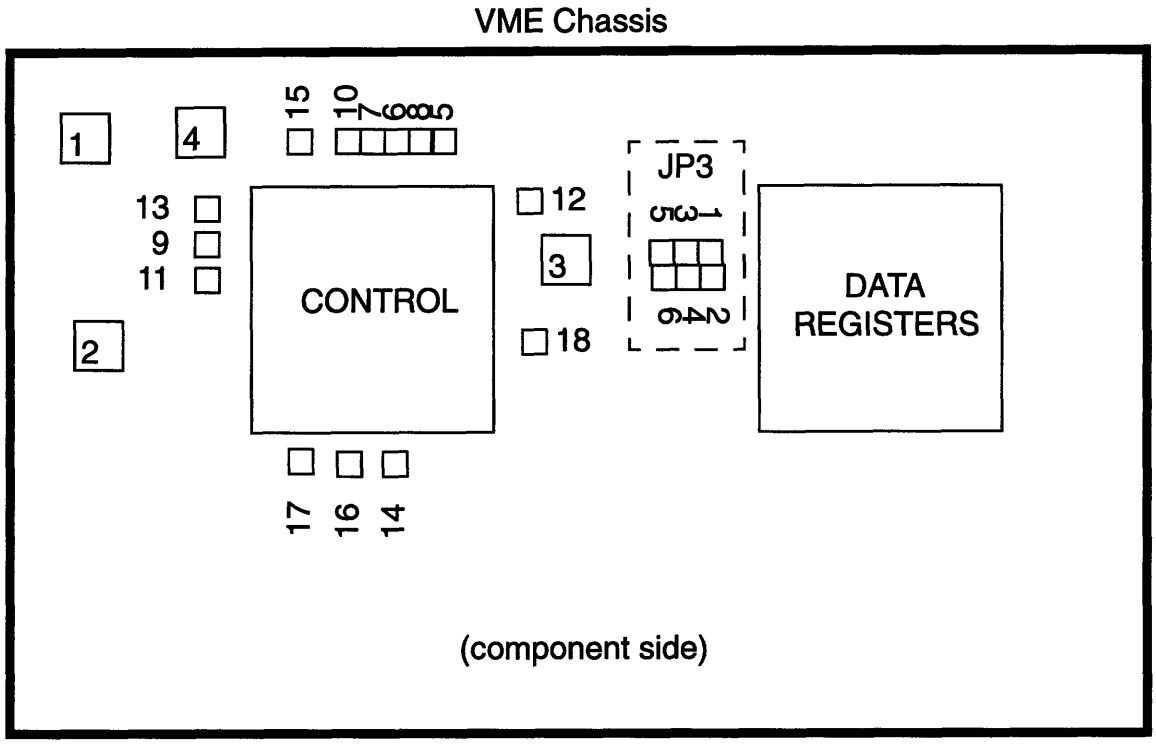

Figure A-1: Test Point Location

Jumper JP3 is used to control the clock circuit. It includes three jumpers, between pins 1-2, 3-4, and 5-6. These jumpers are used as follows:

Pins 1-2 : jumped = DISABLE PLL on clock driver (for clocks below 10 MHz)[1]
open = ENABLE PLL on clock driver (default)
Pins 3-4: jumped = select SYNC1 (external clock input) for clock driver input
open = select SYNC0 (PE array clock signal) for clock driver input

1. Note that if the PLL is disabled, the clock chip divides the input clock frequency by two.

Pin 5: connect to external clock input if 3-4 jumped. Connect the signal wire to pin 5, and the ground wire to pin 1 of this jumper (pin 1 = Ground).

Pin 6: No connection

Table A-1 lists the test point numbers and their associated signals. Note that test points 1-4 are oscilloscope probe jacks, provided for key timing signals (clocks, asynchronous signals, etc).

#	Name	#	Name	#	Name
1	$\overline{\text{ACK}}$	7	$\overline{\text{AIM0 CS}}$	13	$\overline{\text{OR OE}}$
2	$\overline{\text{MODSEL}}$	8	$\overline{\text{AIM1 CS}}$	14	SEQ I0
3	Q0 (clock output)	9	$\overline{\text{Mem Write}}$	15	SEQ I1
4	Refresh (unbuf)	10	WDR/CDR clock	16	SEQ I2
5	Reset	11	$\overline{\text{WDR/CDR OE}}$	17	SEQ I3
6	$\overline{\text{CIM CS}}$	12	OR clock	18	$\overline{\text{SEQ OE}}$

Table A-1: Test Point Signals

Appendix B

Board Documentation

B.1 Controller Board Schematics

Schematics for the controller board were done in ViewLogic's Powerview, using ViewDraw. This project is currently located on MTL workstations, in **/homes/ghall/XACT/pcboard**.

These schematics may be viewed or printed by entering PowerView, creating a project with the above path ('creating' here merely tells PowerView where to look -- this is only done the first time the files are accessed from a new location), invoking ViewDraw, and opening the main schematic (**board.1**).

B.2 EPLD Internal Schematics

The EPLD schematics are also done in Powerview. These schematics implement all logic internal to both EPLDs used in this research. These files are currently located in the **/homes/ghall/XACT/controller** project. Within this project are three top-level schematics: **control.1**, **dataregs.1**, and **datatest.1**. DataTest is the same as DataRegs, with the exception that it is modified to prevent the controller from ever having interrupts (for testing).

B.3 Pertinent File Information

B.3.1 ABEL Files

As part of the EPLD schematics, ABEL state machine designs have been implemented. These appear on the schematics as generic PAL parts. These include state machines for instruction decoding, memory operations, interrupt handling, etc. They are also found in **/homes/ghall/XACT/controller**. The following is the list of these files:

addrdec.abl - address decoding logic
ccmux.abl - CC-MUX logic
instrdec.abl - instruction decoding state machine
memctrl.abl - memory control and interrupt control state machine
memctrl2.abl - memory control logic
wrtorg.abl - logic to control the Opcode Register / GO bit / interrupt enable

B.3.2 Conversion of ABEL to PDS

Note that XACT (the EPLD schematic 'compiler') requires .PDS files. The following is a script

that, with the executables provided with XACT and PowerView, will convert .ABL to .PDS:

```
ahdl2blf $1
blifopt $1 -pla
pla2eqn $1.tt2 -language pds
echo
echo After TITLE, enter "chip (module name) generic"[1]
```

B.3.3 EPLD Simulation Files

Test vectors were not a part of the ABEL files in this project. Instead, PowerView provides a tool to simulate schematics of EPLD designs with a more rich interface and output timing diagrams, including exact timing specification. These simulations are invoked by entering ViewSim, and executing command the command scripts:

addrdec.cmd -	tests address decoding logic
busif.cmd -	tests bus interface logic (interfacing with PLX2000)
ccmux.cmd -	tests CCMUX functionality
ccr.cmd -	tests interrupt counter / reset counter logic
dataregs.cmd -	tests SSR / OSR functions
instrdec.cmd -	tests instruction decoding logic
intrpt.cmd -	tests interrupt generation logic (interface with sequencer)
memctrl.cmd -	tests logic used in writing the control store
rddata.cmd -	tests reading data from the controller
wrtorg.cmd -	tests logic used in launching a sequence

B.3.4 Other Important Files

Any other important information regarding programming the EPLDs, board documentation, etc., will reside in files reference by **/homes/ghall/XACT/README**.

1. The researcher had to MANUALLY insert the quoted line at the top of each .PDS line for XACT to function correctly.

Appendix C

Software Maintenance

The software framework is a very large software project. It is also very structured, and knowing the structure of this project will be necessary prior to any serious modifications. It is also important to understand how to correctly compile the framework if necessary. Currently, the most recent version of the framework is **xapp-3**. At the time of this thesis, it resides at **/homes/ghall/src/xapp-3**, on MTL machines.

C.1 Compilation

The framework is structured to take advantage of the *make* and *autoconf* utilities in UNIX. The following are brief descriptions of pertinent files used in compiling the framework:

xapp-3/configure.in - used by autoconf to tell the locations of all subdirectories that are to have makefiles produced, to create a configure script.

makefile.in - a file by this name should reside in every subdirectory referenced in configure.in with code to be compiled. It tells the configure script how to produce a makefile to compile this part of the framework.

xapp-3/configure-sh/configure* - several configure scripts, to produce makefiles for (for instance), debug or optimized release code, Solaris or SunOS operating systems, Sun's CC or Gnu's CC compiler, etc.

To produce the new framework binaries when changes are made (assuming the configure.in and makefile.in files are still correct), from the **xapp-3** directory:

- Run **autoconf** with no arguments - this will produce the configure scripts that will in turn create the makefiles. This ONLY needs to be done if the **configure.in** file was modified.
- Run **make distclean** to remove all binaries and makefiles from the framework directories. This is done to produce a clean build.
- Run an appropriate configure script to create the makefiles for the framework, based on the makefile.in files. For instance, to produce optimized code for Solaris machines using Sun's C++ compiler, execute **configure-sh/configure-sparc_CC_O_SCCS**. This will create all of the makefiles for the framework.
- Run **make**. This will compile the source code into the appropriate binaries.

C.2 XAPP Structure

This section will show the basic tree structure of the xapp-3 project in its current form. Also included are very brief descriptions of which files are where. This is to aid programmers who

need to be familiar with the details of the system's operation, to navigate through the software framework. Note that not all directories are listed - merely pertinent ones.

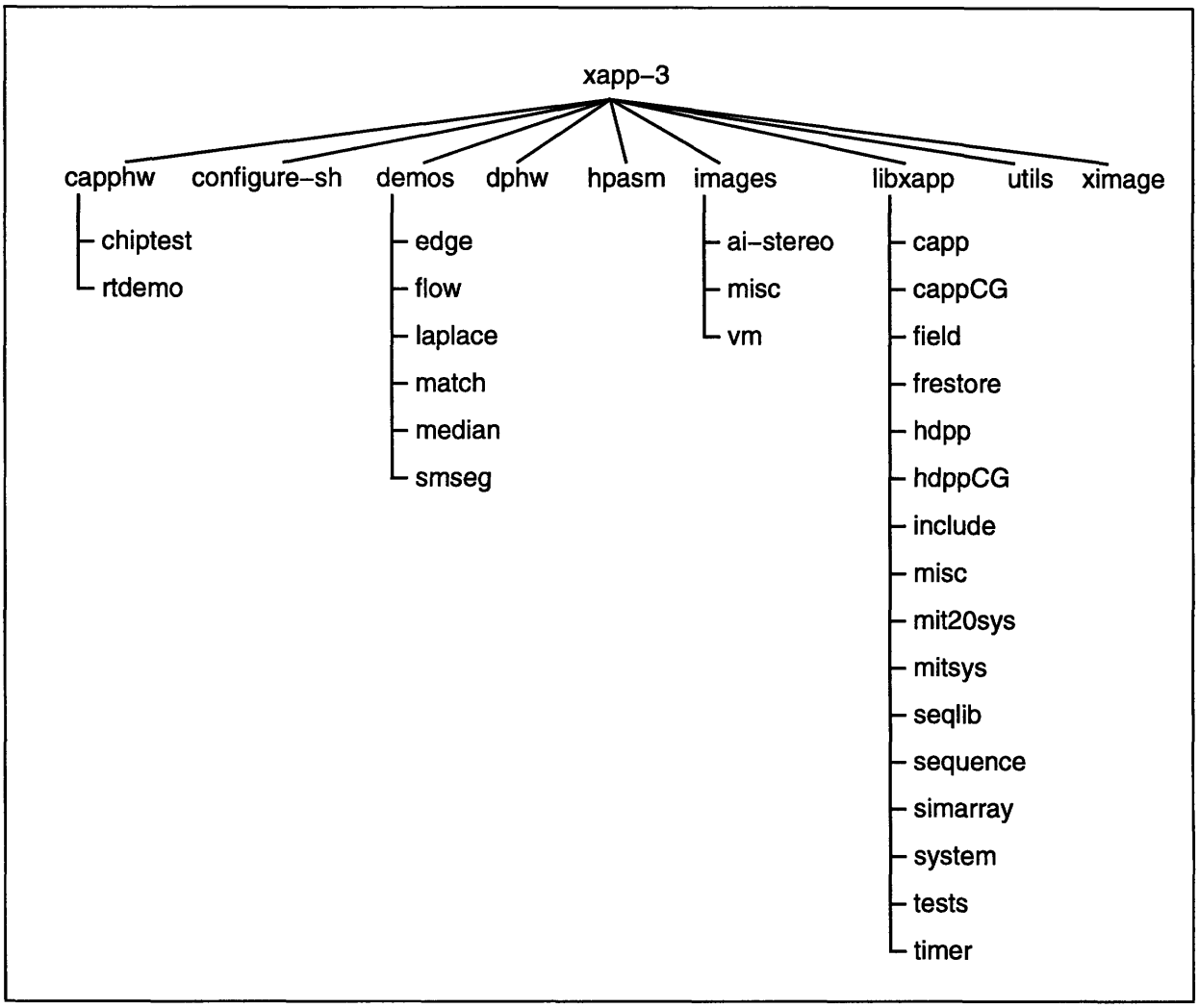

Figure C-1: XAPP-3 Directory Structure

The descriptions of these directories, as appropriate, follow:

xapp-3 - Main project directory, holds **configure.in** for the entire project.

capphw - Software specific to the CAPP array hardware.

chiptest - Software written to perform testing diagnostics on the CAPP array.

rtdemo - Software written to perform a real-time smooth & segment demonstration on the CAPP hardware.

configure-sh - Holds configure scripts to generate various format binaries.

demos - Holds software applications demonstrating all cited sequences, in software or hardware, on either type array. **This is the type of software written by application programmers.**

edge - Edge detection demonstration.

flow - Optical flow demonstration.

laplace - Laplace transform demonstration.

match - Template matching demonstration, for the third implementation only. The first two implementations' source code is here also, but not built as part of the project binaries.

median - Median flow demonstration. The makefile currently only produces executables for the HDPP array.

smseg - Smoothing and segmentation demonstration.

dphw - Source code for the datapath hardware's device drivers and interface functions.

images - Test images used by the project.

misc - ASCII PGM format files used in this research.

vm - ASCII PGM files used for optical flow demonstration.

libxapp - Holds the bulk of the xapp-3 implementation directories.

capp - Code to produce CAPP binary instructions and other CAPP-specific functions.

cappCG - Code generators that produce CAPP code for all generic sequences (add, write, etc.).

field - Code to manage *field* classes (sfield, ufield, etc.).

hdpp - Code to produce HDPP binary instructions and other HDPP-specific functions.

hdppCG - Code generators that produce HDPP code for all generic sequences (add, write, etc.).

include - Class declarations for all classes in the xapp project. **A good place to start to get a picture of which classes contain what.**

mit20sys - Implementation of 20 MHz controller hardware and software simulation. Also

contains code to generate binary controller instructions for the CIM.

mitsys - Implementation of 10 MHz controller hardware and software simulation. Also contains code to generate binary controller instructions for the CIM.

seqlib - Contains **sequence source code** for all sequences demonstrated to-date, included smooth & segment, median filter, optical flow, template matching, and edge detection. **This is the type of software written by application programmers.**

sequence - Source code to manage the *sequence* and *presequence* classes.

simarray - Code to simulate both the CAPP and HDPP arrays in software.

tests - Software to test many aspects of the system, including controller hardware and software, CAPP and HDPP arrays, code generators, etc.

ximage - Source code implementing graphical display of images used in some demonstrations.

Appendix D

Template Matching Application Code

This appendix illustrates each of the three investigated methods for generating a template matching sequence. They vary in speed of execution and maximum template size. Method 3 was the final method used in this research. It's space limitation is 100 x 100 pixels before the control store is full. A 20 x 20 template match requires roughly 3 ms, so much larger templates can also be used at frame rate, and very large templates at speeds near frame rate.

Note that only enough code to convey the sequence logic is shown here.

D.1 Method 1 - Maximum Speed

```
/*************************************************************************
 * FUNCTION : match
 *
 * INPUTS : aC - alloc context for the array
 *          ufield m - main image field
 *          ufield mark - bit to mark square
 *          ufield acc - accumulate field
 *          sfield diff - pixel diff. field
 *          unsigned int nMaxGrey - color used to mark a square
 *          unsigned int nThreshhold - threshhold match value
 *          nWidth - pattern width
 *          nHeight - pattern height
 *          pPatternVals - pointer to pattern pixels
 *
 * OUTPUTS : sequence match
 *
 * DESCRIPTION : creates a sequence to perform pattern matching on an
 *    image, within a threshhold.
 *
 * CAUTIONS : the result will be near the center of the region ONLY if
 *    the pattern is roughly square.  The more elongated the pattern,
 *    the farther off the center may be.
 *
 *************************************************************************/
sequence
    match(allocContext aC,
            ufield m, ufield mark,
            ufield acc, sfield diff, unsigned int nMaxGray,
            unsigned int nThreshhold,
            int nWidth, int nHeight, unsigned int* pPatternVals)
{
```

```
sequence s;

int nCols = nWidth;       // columns remaining
int nRows = nHeight;      // rows remaining
int nRow = 0;             // 0 necessary - see loop alg.
int nCol = -1;            // -1 necessary - see loop alg.
unsigned int nPatternPixel;

// zero out the accumulation field
s += write (aC, acc, 0u);

int x, y;

/*
 * this "spiral" is used to traverse the entire template, and is
 * a bit complex since the template need not be square
 */
while (1)
{
  // move east
  for (x = 0; x < nCols; x++)
  {
    nCol++;
    nPatternPixel = *(pPatternVals + nRow*nWidth + nCol);
    s += move (aC, acc, West);  // move FROM convention!!!
    s += difference (aC, (ufield)diff, m, nPatternPixel);
    s += abs (aC, diff);
    s += add (aC, acc, (ufield)diff);
  }

  if (--nRows == 0) break;

  // move south
  for (y = 0; y < nRows; y++)
  {
    nRow++;
    nPatternPixel = *(pPatternVals + nRow*nWidth + nCol);
    s += move (aC, acc, North);   // move FROM convention!!!
    s += difference (aC, (ufield)diff, m, nPatternPixel);
    s += abs (aC, diff);
    s += add (aC, acc, (ufield)diff);
  }

  if (--nCols == 0) break;

  // move west
  for (x = 0; x < nCols; x++)
  {
    nCol--;
    nPatternPixel = *(pPatternVals + nRow*nWidth + nCol);
    s += move (aC, acc, East);   // move FROM convention!!!
    s += difference (aC, (ufield)diff, m, nPatternPixel);
```

```
    s += abs (aC, diff);
    s += add (aC, acc, (ufield)diff);
  }

  if (--nRows == 0) break;

  // move north
  for (y = 0; y < nRows; y++)
  {
    nRow--;
    nPatternPixel = *(pPatternVals + nRow*nWidth + nCol);
    s += move (aC, acc, South);   // move FROM convention!!!
    s += difference (aC, (ufield)diff, m, nPatternPixel);
    s += abs (aC, diff);
    s += add (aC, acc, (ufield)diff);
  }

  if (--nCols == 0) break;
}

// draw a square around matches
s += write(aC, mark, 0u);
s += inside(aC, acc, 0, nThreshhold);
s += writeC(aC, mark, 1u);

// move the mark bit to the upper-left corner of template
while (nCol > 0)
{
  s += move(aC, mark, East);
  nCol--;
}
while (nRow > 0)
{
  s += move(aC, mark, South);
  nRow--;
}

/*
 * traverse the template square, drawing white outlines around
 * the regions that matched the template
 */

for (x = 0; x < nWidth-1; x++)
{
  s += match(aC, mark, 1u);
  s += writeC(aC, m, nMaxGray);
  s += move(aC, mark, West);
}
for (y = 0; y < nHeight-1; y++)
{
  s += match(aC, mark, 1u);
  s += writeC(aC, m, nMaxGray);
```

```
    s += move(aC, mark, North);
  }
  for (x = 0; x < nWidth-1; x++)
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, East);
  }
  for (y = 0; y < nHeight-1; y++)
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, South);
  }

  return s;
}
```

D.2 Matching 2 - Maximum Template Size

```
/*****************************************************************************
 * FUNCTION : match
 *
 * INPUTS : aC - alloc context for the array
 *          ufield m - main image field
 *          ufield mark - bit to mark square
 *          ufield acc - accumulate field
 *          unsigned int nMaxGrey - color used to mark a square
 *          unsigned int nThreshhold - threshhold match value
 *          nWidth - pattern width
 *          nHeight - pattern height
 *
 * OUTPUTS : sequence match
 *
 * DESCRIPTION : draws the square around matching patterns
 *
 * CAUTIONS : This sequence doesn't perform the match.  In this implementation,
thi
 *   is done within main()
 *
 *****************************************************************************/
sequence match(allocContext aC,
            ufield m, ufield mark,
            ufield acc, unsigned int nMaxGray,
            unsigned int nThreshhold,
            int nWidth, int nHeight)
{
  sequence s;

  int nCols = nWidth;       // columns remaining
  int nRows = nHeight;      // rows remaining
```

```
int nRow = 0;            // 0 necessary - see loop alg.
int nCol = -1;           // -1 necessary - see loop alg.

int x, y;

/*
 * this "spiral" is more complex than for flow, because the
 * pattern is not restricted to be square.
 */

// I'm just calculating the final point here...  see main() for algorithm
while (1)
{
  // move east
  for (x = 0; x < nCols; x++)
    nCol++;

  if (--nRows == 0) break;

  // move south
  for (y = 0; y < nRows; y++)
    nRow++;

  if (--nCols == 0) break;

  // move west
  for (x = 0; x < nCols; x++)
    nCol--;

  if (--nRows == 0) break;

  // move north
  for (y = 0; y < nRows; y++)
    nRow--;

  if (--nCols == 0) break;
}

// draw a square around matches
s += write(aC, mark, 0u);
s += inside(aC, acc, 0, nThreshhold);
s += writeC(aC, mark, 1u);
while (nCol > 0)
{
  s += move(aC, mark, East);  // move data west
  nCol--;
}
while (nRow > 0)
{
  s += move(aC, mark, South); // move data north
  nRow--;
}
```

```
  // put the square in now
  for (x = 0; x < nWidth-1; x++)
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, West);     // go east
  }
  for (y = 0; y < nHeight-1; y++)
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, North);    // draw south
  }
  for (x = 0; x < nWidth-1; x++)
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, East);     // draw west
  }
  for (y = 0; y < nHeight-1; y++)
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, South);    // draw north
  }

  return s;
}


/*****************************************************************************
 * FUNCTION : main
 *
 * INPUTS :
 *
 * OUTPUTS :
 *
 * DESCRIPTION :
 *
 * CAUTIONS :
 *
 *****************************************************************************/
int main()
{

  // perform sys initialization here

  /*
   * the acc field is given enough bits for a 100x100 pixel pattern,
   * assuming maxgrey of 255.   (100*100*255 = 0x26E8F0)
   */
```

```
ufield m(aC.statically(), mlen);              // main image
ufield p(aC.statically(), patLen);            // pattern pixel
sfield diff(aC.statically(), 9);              // hold pixel diff's
ufield acc(aC.statically(), 22);              // accumulate diffs
ufield mark(aC.statically(), 1);              // mark matches

sequence seqX;
seqX += write (aC, p);           // must have write val in SelShReg
seqX += difference (aC, (ufield)diff, m, p);
seqX += abs (aC, diff);
seqX += add (aC, acc, (ufield)diff);

// sequence names = move TO, the move() sequence is move FROM
sequence seqNorth;
seqNorth += move (aC, acc, South);
seqNorth += seqX;

sequence seqSouth;
seqSouth += move (aC, acc, North);
seqSouth += seqX;

sequence seqEast;
seqEast += move (aC, acc, West);
seqEast += seqX;

sequence seqWest;
seqWest += move (aC, acc, East);
seqWest += seqX;

sequence seqMatch = match(aC, m, mark, acc, maxGray, parm.threshold,
                          patWidth, patHeight);

sequence seqWrite = write (aC, acc, 0u);

// create array of pattern pixel values
unsigned int* pPattern = new unsigned int [patHeight*patWidth];
for (int a = 0; a < patHeight; a++)
{
  for (int b = 0; b < patWidth; b++)
  {
    PatternFile >> *(pPattern + a*patWidth + b);
  }
}

int nCols = patWidth;       // columns remaining
int nRows = patHeight;      // rows remaining
int nRow = 0;              // 0 necessary - see loop alg.
int nCol = -1;             // -1 necessary - see loop alg.
int x, y;
unsigned int nPatternPixel;
```

```
sys << seqWrite;

while (1)
{
  // move east
  for (x = 0; x < nCols; x++)
  {
    nCol++;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    sys << (unsigned long)nPatternPixel;  // write SSR
    sys << seqEast;
    cerr << ".";
  }

  if (--nRows == 0) break;
  cerr << endl;

  // move south
  for (y = 0; y < nRows; y++)
  {
    nRow++;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    sys << (unsigned long)nPatternPixel;  // write SSR
    sys << seqSouth;
    cerr << ".";
  }

  if (--nCols == 0) break;
  cerr << endl;

  // move west
  for (x = 0; x < nCols; x++)
  {
    nCol--;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    sys << (unsigned long)nPatternPixel;  // write SSR
    sys << seqWest;
    cerr << ".";
  }

  if (--nRows == 0) break;
  cerr << endl;

  // move north
  for (y = 0; y < nRows; y++)
  {
    nRow--;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    sys << (unsigned long)nPatternPixel;  // write SSR
    sys << seqNorth;
    cerr << ".";
  }
```

```
        cerr << endl;
        if (--nCols == 0) break;
      }

      // draw the squares around the matches
      sys << seqMatch;

  return EXIT_SUCCESS;
}
```

D.3 Method 3 - Compromise

```
/*************************************************************************
 * FUNCTION : match
 *
 * INPUTS : aC - alloc context for the array
 *          ufield m - main image field
 *          ufield p - field for pattern pixel
 *          sfield diff - hold pixel differences
 *          ufield acc - accumulates differences (big!)
 *          ufield mark - 1 bit field to mark matches
 *          ifstream* PatternFile - input PGM file of template
 *          int patWidth, int patHeight - pattern dimensions
 *          int maxGray - color to mark matches with
 *          int thresh - how close match must be
 *
 * OUTPUTS : sequence match
 *
 * DESCRIPTION : creates a sequence to perform template matching
 *
 * CAUTIONS :
 *
 *************************************************************************/
sequence match(allocContext aC, ufield m, ufield p, sfield diff,
               ufield acc, ufield mark, ifstream* PatternFile,
               int patWidth, int patHeight, int maxGray,
               int thresh)
{
  // initialize
  sequence seq = write (aC, acc, 0u);

  seq += calcmatch(aC, m, p, diff, acc, PatternFile,
                           patWidth, patHeight);

  seq += markmatch(aC, m, mark, acc, maxGray, thresh,
                            patWidth, patHeight);

  return seq;
}
```

```
/*****************************************************************************
 * FUNCTION : calcmatch
 *
 * INPUTS :
 *
 * OUTPUTS :  sequence calcmatch
 *
 * DESCRIPTION : Builds the list of gosubs and moves for the overall
 *   processing.  256 gusub'd presequences are used so that I don't
 *   need a new sequence in memory for each pixel in the template, just
 *   one for each possible template value.  There is also 4 sequences
 *   since each possible pixel can also be moved N,S,E,W after
 *   the computation.
 *
 * CAUTIONS :
 *
 *****************************************************************************/
sequence calcmatch(allocContext aC,
            ufield m, ufield p,
            sfield diff, ufield acc,
            ifstream* PatternFile,
            int patWidth, int patHeight)
{
  presequence ps;
  presequence psTemp;
  char cInst[50];                              // gosub instruction string

  // create array of pattern pixel values
  unsigned int* pPattern = new unsigned int [patHeight*patWidth];
  for (int a = 0; a < patHeight; a++)
  {
    for (int b = 0; b < patWidth; b++)
    {
      *PatternFile >> *(pPattern + a*patWidth + b);
    }
  }

  int nCols = patWidth;       // columns remaining
  int nRows = patHeight;      // rows remaining
  int nRow = 0;              // 0 necessary - see loop alg.
  int nCol = -1;             // -1 necessary - see loop alg.
  int x, y;
  unsigned int nPatternPixel;

  /*
   * for this algorithm, it's more efficient to have 2 gosubs per
   * pixel, than to have 1 gosub here and inserting the other
   * sequence.  Also better than inserting the 'move' into the
   * later 256 loop.  As a bonus, this is a lot more efficient for
   * smaller patterns.
   */
```

```
// for each pixel in the template, gosub to one of these preseqs
while (1)
{
  // move east
  for (x = 0; x < nCols; x++)
  {
    nCol++;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    ps += _CONTROLLER::inst::gosub((const char*)"East");
    sprintf(cInst, "%d", nPatternPixel);
    ps += _CONTROLLER::inst::gosub((const char*)cInst);
  }

  if (--nRows == 0) break;

  // move south
  for (y = 0; y < nRows; y++)
  {
    nRow++;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    ps += _CONTROLLER::inst::gosub((const char*)"South");
    sprintf(cInst, "%d", nPatternPixel);
    ps += _CONTROLLER::inst::gosub((const char*)cInst);
  }

  if (--nCols == 0) break;

  // move west
  for (x = 0; x < nCols; x++)
  {
    nCol--;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    ps += _CONTROLLER::inst::gosub((const char*)"West");
    sprintf(cInst, "%d", nPatternPixel);
    ps += _CONTROLLER::inst::gosub((const char*)cInst);
  }

  if (--nRows == 0) break;

  // move north
  for (y = 0; y < nRows; y++)
  {
    nRow--;
    nPatternPixel = *(pPattern + nRow*patWidth + nCol);
    ps += _CONTROLLER::inst::gosub((const char*)"North");
    sprintf(cInst, "%d", nPatternPixel);
    ps += _CONTROLLER::inst::gosub((const char*)cInst);
  }

  if (--nCols == 0) break;
}
```

```
  // delete array
  delete [] pPattern;

  // now the gosubs are done, so finish.   The actual destinations
  // come after this point.

  // jump to the end of this sequence, to fall on to either the
  // next sequence, if added, or the jzero at end.
  ps += _CONTROLLER::inst(" branch End () ");

  // this block is done EVERY pixel
  ps += "TempLabel";
  ps += differenceV(aC, (ufield)diff, m, p);
  ps += absV(aC, diff);
  ps += addV(aC, acc, (ufield)diff);
  ps += _CONTROLLER::inst::ret();        // return from gosub

  // subroutine for each dir. move - 1 of 4 per pixel
  ps += "East";                          // move TO
  ps += moveV(aC, acc, West);            // array notation - move FROM
  ps += _CONTROLLER::inst::ret();

  ps += "West";                          // move TO
  ps += moveV(aC, acc, East);            // array notation - move FROM
  ps += _CONTROLLER::inst::ret();

  ps += "South";                          // move TO
  ps += moveV(aC, acc, North);            // array notation - move FROM
  ps += _CONTROLLER::inst::ret();

  ps += "North";                          // move TO
  ps += moveV(aC, acc, South);            // array notation - move FROM
  ps += _CONTROLLER::inst::ret();

  // subroutine for each greyscale - 1 of 256 per pixel
  for (unsigned int nColor = 0; nColor < 256; nColor++)
  {
    sprintf(cInst, "%d", nColor);
    ps += (const char*)cInst;
    ps += writeV(aC, p, nColor);
    ps += _CONTROLLER::inst::branch("TempLabel");
  }

  // end of sequence
  ps += "End";

  // assemble this presequence and return it
  return ps.assemble();
}
```

```
/*************************************************************************
 * FUNCTION : markmatch
 *
 * INPUTS : aC - alloc context for the array
 *          ufield m - main image field
 *          ufield mark - bit to mark square
 *          ufield acc - accumulate field holding final values
 *          unsigned int nMaxGrey - used to mark a square
 *          unsigned int nThreshhold - threshhold match value
 *          nWidth - pattern width
 *          nHeight - pattern height
 *
 * OUTPUTS : sequence markmatch
 *
 * DESCRIPTION : draws the square around matching patterns
 *
 * CAUTIONS :
 *
 *************************************************************************/
sequence markmatch(allocContext aC,
            ufield m, ufield mark,
            ufield acc, unsigned int nMaxGray,
            unsigned int nThreshhold,
            int nWidth, int nHeight)
{
  sequence s;

  int nCols = nWidth;       // columns remaining
  int nRows = nHeight;      // rows remaining
  int nRow = 0;             // 0 necessary - see loop alg.
  int nCol = -1;            // -1 necessary - see loop alg.

  int x, y;

  /*
   * this "spiral" is more complex than for flow, because the
   * pattern is not restricted to be square.
   */

  // I'm just calculating the final point here...  see mainseq()
  // for algorithm
  while (1)
  {
    // move east
    for (x = 0; x < nCols; x++)
      nCol++;

    if (--nRows == 0) break;

    // move south
    for (y = 0; y < nRows; y++)
```

```
    nRow++;

  if (--nCols == 0) break;

  // move west
  for (x = 0; x < nCols; x++)
    nCol--;

  if (--nRows == 0) break;

  // move north
  for (y = 0; y < nRows; y++)
    nRow--;

  if (--nCols == 0) break;
}

// draw a square around matches
s += write(aC, mark, 0u);
s += inside(aC, acc, 0, nThreshhold);
s += writeC(aC, mark, 1u);
while (nCol > 0)
{
  s += move(aC, mark, East);  // move data west
  nCol--;
}
while (nRow > 0)
{
  s += move(aC, mark, South); // move data north
  nRow--;
}

// put the square in now
for (x = 0; x < nWidth-1; x++)
{
  s += match(aC, mark, 1u);
  s += writeC(aC, m, nMaxGray);
  s += move(aC, mark, West);    // go east
}
for (y = 0; y < nHeight-1; y++)
{
  s += match(aC, mark, 1u);
  s += writeC(aC, m, nMaxGray);
  s += move(aC, mark, North);   // draw south
}
for (x = 0; x < nWidth-1; x++)
{
  s += match(aC, mark, 1u);
  s += writeC(aC, m, nMaxGray);
  s += move(aC, mark, East);    // draw west
}
for (y = 0; y < nHeight-1; y++)
```

```
  {
    s += match(aC, mark, 1u);
    s += writeC(aC, m, nMaxGray);
    s += move(aC, mark, South);    // draw north
  }

  return s;
}
```