Splash-2 Shared-Memory Architecture for Supporting High Level Language Compilers

by

Bradley K. Fross

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. Peter M. Athanas, Chairman

Dr. A. Lynn Abbott

Dr. Nathaniel J. Davis, IV

June, 1995

Blacksburg, Virginia

Splash-2 Shared-Memory Architecture for Supporting High Level Language Compilers

by

Bradley K. Fross

Committee Chairman: Dr. Peter Athanas

Electrical Engineering

(ABSTRACT)

Modern computer technology has been evolving for nearly fifty years, and has seen many architectural innovations along the way. One of the latest technologies to come about is the reconfigurable processor-based *custom computing machine* (CCM). CCMs use *field programmable gate arrays* (FPGAs) as their processing cores, giving them the flexibility of software systems with performance comparable to that of dedicated custom hardware. Hardware description languages are currently used to program CCMs. However, research is being performed to investigate the use of high-level languages (HLLs), such as the C programming language, to create CCM programs. Many aspects of CCM architectures, such as local memory systems, are not conducive to HLL compiler usage. This thesis proposes and evaluates the use of a shared-memory architecture on a Splash-2 CCM to promote the development and usage of HLL compilers for CCM systems.

Acknowledgments

I would like to thank Dr. Peter Athanas for making my graduate career possible. His many contributions to my career at Virginia Tech include welcoming me into the program, helping me through my research, and most importantly, getting me out. His incessant dedication is apparent to all his students and peers, and I would like to be one of the many to thank him for all his help. I would also like to thank Dr. Abbott for being my co-advisor, and for helping me understand the sometimes complicated world of image processing. I am also grateful for Dr. Davis' willingness to serve on my defense committee, and for all his help with computer architectures and interconnection networks.

I want to thank the VTSplash group for all of their support, and for providing me with endless hours of companionship in the lab. Many thanks go to the Virginia Tech Crew Club for providing the best distraction anyone could hope for. I was able to avoid doing research while getting in shape and having a great time.

I would like to thank Heather for being there for me to lean on. Her caring and understanding are unsurpassed, and her interest in my work gave me a valuable new perspective I would not have had without her. I would like to say that I would not be here if it were not for the love, caring, and support of my family. I dedicate this thesis to my parents, Linda and Ralph Fross, for none of this would be possible without their unconditional love.

Table of Contents

List of Figures

Chapter 1

1. Introduction

This thesis describes the impetus behind the development of a shared-memory architecture for the Splash-2 custom computing machine. Recent hardware and software developments in computer technology are discussed to provide a foundation for the proposed architecture described herein. Finally, a proposed Splash-2 shared-memory architecture is presented, along with a detailed evaluation of its implementation.

1.1 Motivation

Advances in microelectronics have once again provided the means to change the face of computer architectures. A device called the *field programmable gate array*, or

FPGA, combines the flexibility of software with the high-performance characteristics of custom-built hardware [Xilinx94]. The FPGA is a very powerful processing element when used as the computational core of a new type of computer architecture called a *custom computing machine* (CCM) [Buell93].

The CCM architecture is arguably here to stay, as several second-generation systems such as the Splash-2 [Arnold92] and CHAMP-2 [Box95] have been developed. In addition to these seasoned veterans, many new CCMs have sprouted from the roots of this new technology. These new CCMs include the Hewlett Packard Teramac CCM [Amerson95], the Giga Operations G-800 [Giga94], and the Annapolis Micro Systems Wildfire [AMS95].

Programming environments for CCMs are still in their infancy. Many CCMs are programmed using hardware description languages, such as VHDL [VHDL92]. Programming CCMs using VHDL is analogous to programming a personal computer using assembly language. It requires great expertise to effectively program a CCM using VHDL. The development of high-level languages, including PRISM-2 [Athanas92] and dbC [Gohkale93], greatly improve the efficiency of programming CCMs. A rough approximation of the development time for programming CCMs using hardware description languages (HDLs) and high-level languages (HLLs) is shown in Figure 1.1.

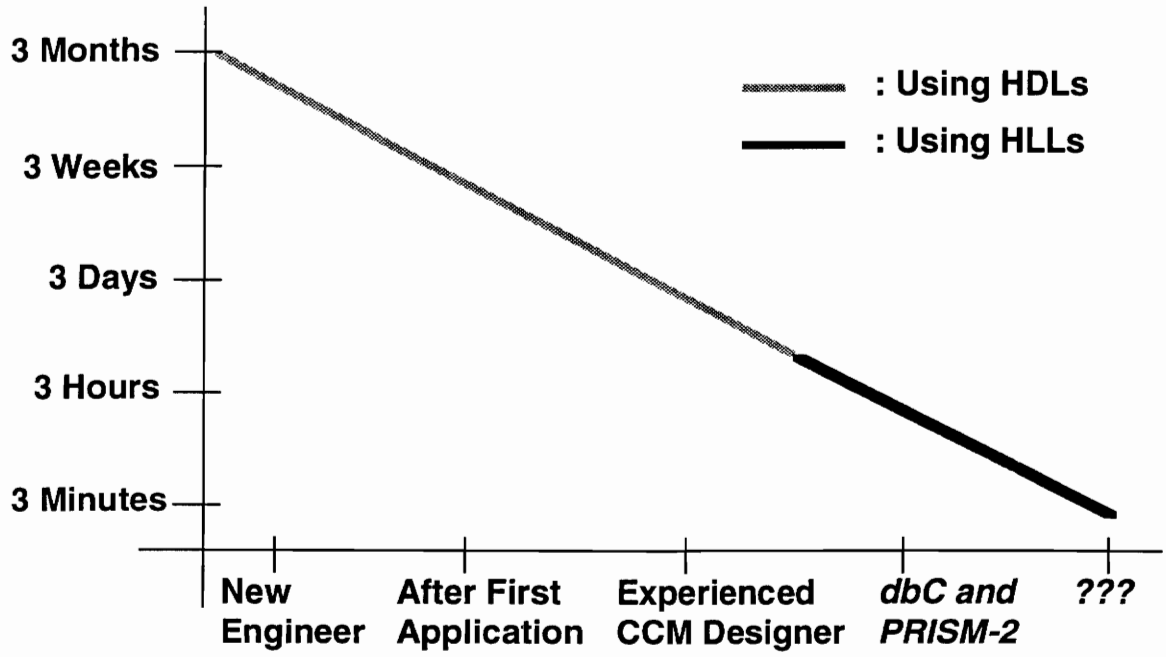


Figure 1.1 CCM program development time.

Program development time for a CCM depends on the characteristics of both the software and hardware components used in the design. This thesis describes several attributes of software and hardware systems that affect the efficiency of CCM programming environments, and methods to improve this efficiency.

1.2 Summary of Contributions

The contribututions made by the research covered in this thesis are shown below:

- Research into compiler theory and computer architectures is applied to the problem of making Splash-2 a more suitable environment for high-level language compilers.

- The Splash-2 system is evaluated for modification feasibilty, and a proposed hierarchical shared-memory system for Splash-2 is described.

- A VHDL simulation model of the Splash-2 shared-memory network is implemented, including full timing characteristics of the components that will be used to build it.

- The simulation model is tested by using a full set of test bench vectors and several typical parallel algorithms.

The next section describes the organization of these contributions within this thesis.

1.3 Thesis Organization

This thesis describes several methods of modifying computer architectures to improve their programming environments. A modification of the Splash-2 memory architecture and its effect on the Splash-2 programming environment are described and evaluated.

Chapter 2 presents a brief overview of some background information and terminology that is needed to have a full understanding of the task at hand. High-level language compilers, multiprocessor shared-memory architectures, interconnection

networks, and custom computing machines are introduced and described in detail. The relationship between these four areas and how they affect the programming environment is covered throughout the chapter.

Chapter 3 introduces a shared-memory architecture for improving the effectiveness of high-level language compilers designed for the Splash-2 custom computing machine. Several shared-memory architectures are examined, and a detailed VHDL model of the most suitable architecture is described and evaluated.

Chapter 4 presents an in-depth performance evaluation of the proposed Splash-2 shared-memory architecture. Results from benchmark tests and several applications are presented and analyzed.

Chapter 5 describes improvements that can be made to the shared-memory design. Any future work that needs to be done is covered in this chapter.

Chapter 6 concludes this thesis by summarizing the tasks performed by the author. The features and shortcomings of the Splash-2 shared-memory design are described.

Chapter 2

2. Background and Terminology

The purpose of this chapter is to describe the relationship between shared memory architectures, interconnection networks, and compilers for FPGA-based multiprocessor systems. This chapter covers the background and terminology of high-level language compilers, memory system organization, interconnection networks, and FPGA-based architectures.

The first section introduces techniques used by software compilers for creating programs on conventional processor-based computers. Many of these techniques are used by hardware compilers for reconfigurable processor-based systems. These hardware compilers are described in Section 2.2.3.

2.1 High Level Language Compilers

The advent of high-level languages such as FORTRAN or C has created the necessity for compilers that translate program behavior into executable form for modern computers [Hennessey90]. These high-level language (HLL) compilers perform much of this basic-level code generation and optimization that would otherwise have to be done by the programmer. HLL compilers must handle many different types of tasks in the transformation of algorithm behavior to efficient executable code. Typical tasks include the translation of the high-level language constructs of the program into a machine-dependent intermediate form (i.e. *assembly code* or *object code*) and the optimization of the intermediate code to produce more efficient executable code.

Compiler tasks such as resource scheduling and program partitioning depend on the pipelined or parallel nature of the target machine, respectively. For instance, a C-compiler for a RISC machine must translate not only the program to an intermediate form, but must also allocate and schedule the functional units (i.e. registers, ALUs, pipelines, memory) of the machine in an optimal manner. The next section further discusses these compiler tasks and how they are handled in a parallel multiprocessor system.

2.2 HLL Compilers for Parallel Multiprocessor Systems

High-level language compilers for parallel multiprocessor systems differ from compilers designed for a scalar uniprocessor system in several ways. A uniprocessor system has none of the overhead associated with interprocessor communication and

resource sharing characteristics of multiprocessor systems. A uniprocessor compiler does not have to partition the design among multiple processors. Resource allocation and functional unit scheduling is much more complex for multiprocessor compilers because of interprocessor synchronization and communication needs to be considered. Furthermore, a parallelizing compiler often must convert a sequential language such as C into parallel form suitable for partitioning into multiple segments that can operate on the multiple processors. The next section describes several of the general issues faced by a parallelizing compiler.

2.2.1 General Issues

A parallelizing compiler for a multiprocessor consists of the three major stages as described in [Hwang93]:

- **Flow analysis**: reveals the flow pattern in order to determine control and data dependencies in the source code. This phase is generally used to expose parallelism in a computer program.
- **Program Optimization**: manipulates the program in order to take advantage of specific hardware capabilities. This phase is used to maximize the speed of code execution.
- **Code Generation**: transforms the code into an intermediate form that includes parallel constructs such as message passing, synchronization and code/data

sharing, as needed. This phase is closely linked to the binding and partitioning of a program into parallel code form.

The flow analysis stage of a program compilation is generally machine independent, but relies heavily on the constructs of the high-level language itself. As the compiler begins the optimization and code generation stages, compilation depends more heavily on the architecture of the machine. This architectural dependency is especially apparent during the scheduling portion of the code generation stage due to the number of functional units available for use in different architectures. The next two sections discuss several architectural design philosophies and how they affect compiler development.

2.2.2 Conventional Processor Issues

A conventional processor-based platform presents a compiler with a finite number of non-changeable components which must be allocated according to a program's needs [Hennessey90]. Some of the main hardware components include registers, arithmetic logic units (ALUs), and memories. The compiler must map the program to these components in such a manner as to make efficient use of the resources while completing the program behavior in a timely fashion. The available resources of the target system might not meet the requirements of the program. One way to overcome this problem is to share commonly used resources between program tasks. If the behavior of the program calls for a single component to be used by multiple tasks, the compiler must *schedule* the tasks so that no conflicts, or *structural hazards* arise.

One of the tasks faced by a compiler is to schedule the use of these hardware components in order to minimize the occurrence of structural hazards. Hazards are defined as situations which prevent the next instruction in a stream from executing on the next clock cycle [Hennessey90]. A hazard in a pipelined architecture causes the pipeline to stall, which greatly decreases the performance of a program. By partitioning a program into smaller pieces, the occurrence of structural hazards can be avoided because of proper scheduling.

The computer architect's job is not disjoint from that of the computer programmer. It is well within the domain of the architect to provide hardware support to make the task of the compiler designer easier while maximizing the utilization of these resources. As a computer architect, one way to do this is to provide an abundant quantity of registers and ALUs so that scheduling becomes an easier task. Without imposing too many resource allocation constraints on the compiler, it is difficult to design a computer with the exact number and type of hardware resources needed by every program. The result of such a restriction inevitably ends up in a shortage or surplus of resources, thus decreasing the utility of these resources. This mismatch in hardware components and program requirements for such components is costly in both wasted resources and reduced performance. The next section introduces an alternative to the conventional processor design methodology.

2.2.3 Reconfigurable Processor Issues

The arrival of reconfigurable hardware provides an alternative solution to the scheduling and partitioning challenge by allowing the architecture to change according to a program's needs. The term "adaptable architecture" has been used to describe an architecture that literally changes its form according to the task at hand [Estrin50, Agarwala78]. The recent development of field programmable gate arrays (FPGAs) has brought on a new generation of adaptable architectures commonly referred to as *custom computing machines* (CCMs) [Buell92]. CCMs include systems such as the DEC-Paris Research Lab's PeRLe-1 [Touati93] and the Splash-2 attached processor built at the Center for Computing Sciences, in Bowie, Maryland (formerly known as the Supercomputing Research Center) [Arnold92]. CCMs will be covered further in Section 2.6.

The challenge of scheduling physical resources is not as restrictive in a FPGA-based processor as it was in the conventional processor case. For instance, if a program requires more registers but fewer ALUs than a conventional processor implementation has to offer, the scheduler is required to timeshare the available registers while leaving the unused ALUs idle. In this case, the reconfigurable processor version can reallocate the hardware dedicated to the idle ALUs to more register space, thus removing the need to schedule fixed functional units.

This reconfigurable aspect of FPGA-based processors presents compilers with the additional task of optimizing not only for speed, but also for area, due to the limited resources of an FPGA. Another difference between conventional compilers and compilers used for FPGA-based computers is the necessity for transforming program behavior into the physical configuration needed by FPGA-based processors. This transformation is known as the *synthesis* process [Synopsys94]. Several examples of such high-level language compilers for FPGA-based processors include PRISM-2 [Athanas92], dBC [Gokhale93], Transmogrifier-C [Galloway94], and the GigaOps compiler [Giga95].

Unfortunately, reconfigurable processors do have several drawbacks when compared to their conventional counterparts. A comparison of FPGA versus ASIC performance reveals that FPGAs are relatively slower. For instance, a very simple 8-bit loadable up/down binary counter implemented in a Xilinx XC3100-3 FPGA is at least 1.5 times slower than a dedicated ASIC counter of similar structure [Xilinx94,TI93]. The generalized structure of an FPGA leads to a lack of application specific logic structures and data paths, and therefore a slight reduction in performance.

The extensive internal resources of an FPGA that are dedicated to programmable signal routing and design configuration reduce the amount of area that can be used to implement logic functions. These same internal resources are also the reason that the usable gate count for an FPGA is significantly lower than an ASIC of the same die size. However, CCMs, or other FPGA-based systems, often have greater flexibility and comparative performance over a wider range of application than any one ASIC has. The

reconfigurable architecture of CCM FPGA-based processors can exploit more concurrency and better resource utilization than their ASIC counterparts. For these reasons, FPGAs are useful for constructing the custom processing elements of CCMs.

Even though FPGAs are very well suited for instantiating components such as registers and other types of data flow components, they are not capable of implementing significant amounts of memory. Typical applications need more than the maximum of 32 kbits of memory offered by large FPGAs such as the Xilinx XC4025 [Xilinx94]. Additional memory devices need to be attached to the FPGA to provide sufficient memory for the application to use. The memory devices can be organized in many different ways, depending on the applications needs and architecture of the CCM. The next two sections describe how the compiler can use memory organization to improve program performance.

2.3 HLL Compilers and Memory Organization

According to Hennessey and Patterson, compiler technology has made steady progress in utilizing ever larger register sets [Hennessey90]. Certain types of memory accesses such as local scalar variables, frequently used variables, and temporary variables can be replaced by faster register accesses. This increase in register set usage necessitates an increase in register capacity. This bodes well for reconfigurable processors that are capable of increasing the number of registers on a task-by-task basis. Unfortunately, certain types of memory accesses cannot be replaced by the allocation of more registers.

These types include unallocated references, global scalars, required stack references, and computed references. These memory accesses are not usually assigned to registers because of sparse occurrence, aliasing behavior, or referencing via a pointer or array index [Hennessey90].

Memory access in multiprocessor systems is further complicated when variables and other data are shared between processors. If the memory devices cannot be shared among processors, other interprocessor communication resources must be used to transfer this data. Several memory architectures in multiprocessors allow the memory to be shared among processors without using valuable processor-to-processor communication resources. The next section introduces several classes of memory architectures in addition to their advantages and disadvantages when used in multiprocessor systems.

2.4 Memory Model Classification

Multiprocessors can be organized many different ways. If each processor has a local memory and it can only communicate with its own local memory, the memory is said to be a *distributed unshared-memory*. On the other hand, if the memory is accessible to all the processors in the system, the memory is said to be *global* or *distributed shared-memory* or just *shared-memory*. The next few subsections describe the architectural and performance differences between unshared and shared-memory architectures, including differences between global and distributed shared-memory models.

2.4.1 Distributed Unshared-Memory Model

An example of a distributed unshared-memory is shown in Figure 2.1. Each of the processors in the system (depicted as P_1, P_2, through P_m) has a local memory (shown as LM_1, LM_2, through LM_n) to which it is directly connected. The interprocessor communication in such a system is separate from the memory architecture. All interprocessor communication must take place over the interconnection network, usually in the form of *message passing* [Hwang93]. This includes remote memory accesses, since the distributed memory is not shared among all of the processors. Remote memory accesses can use considerable amounts of processor and interconnect resources and/or time, especially if the accesses are frequent or occur between relatively distant processors in a system. This remote memory access overhead can be quite expensive in time and/or area, depending on the network topology of the interconnection network, but will involve at least two processors and one network link per access.

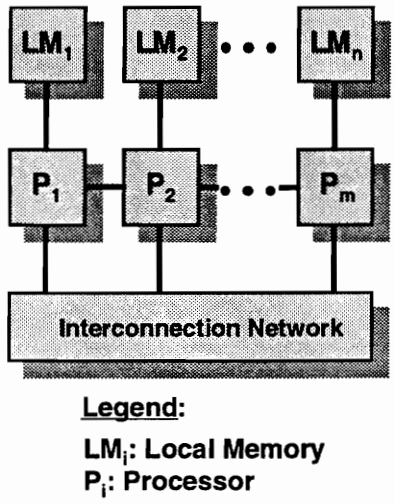
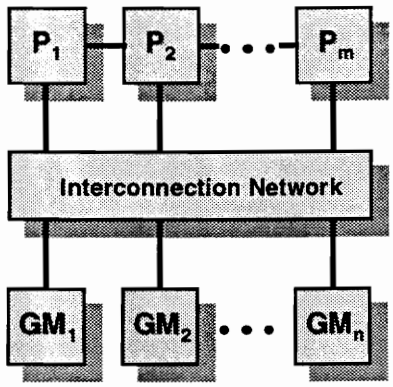


Figure 2.1 Distributed unshared-memory model.

2.4.2 UMA Shared-Memory Model

The *Uniform Memory Access* (UMA) shared-memory model shown in Figure 2.2 can be considered as the diametric counterpart to the distributed unshared-memory model discussed in the previous section. The memory is located in a common pool rather than being distributed among each of the processors. In addition, each processor can directly access the physical memory devices through an interconnection network in uniform (or constant) time rather than having to use interprocessor communication resources to perform expensive remote memory accesses.

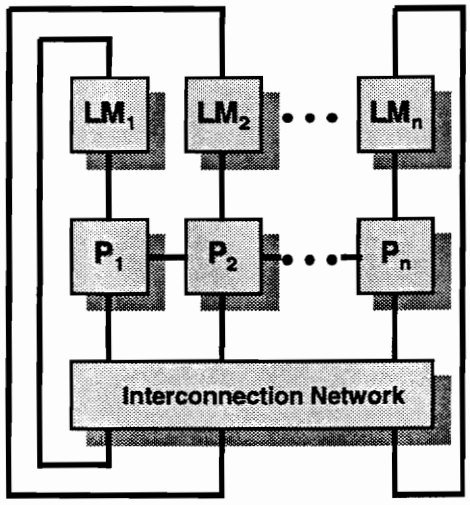
One advantage of the UMA model is that it is easily extended from a uniprocessor system design by modifying existing interconnection network hardware, as will be shown in Chapter 3. This UMA multiprocessor extension can be used for accelerating the execution of a single large program in time-critical applications or time-sharing programs by multiple users [Hwang93]. A disadvantage of the UMA model is the fact that there is no concept of memory locality. This means that the increased memory overhead is visible to all processors during all memory accesses resulting in reduced memory performance over its uniprocessor counterpart. An example of a memory model that does take memory locality into consideration is discussed in the following section.



Legend:

GM_i: Global Memory
P_i: Processor

Figure 2.2 UMA shared-memory model.



Legend:

LM_i: Local Memory
P_i: Processor

Figure 2.3 NUMA shared-memory model.

2.4.3 NUMA Shared-Memory Model

The Non-Uniform Memory Access (NUMA) shared-memory model is a form of *distributed shared-memory*. As shown in Figure 2.3, the NUMA model is similar to the distributed unshared-memory because of the local memory physically attached directly to each processor. The major difference between the distributed unshared-memory and NUMA models is that remote or non-local memory accesses take place through the interconnection network without the use of expensive interprocessor message passing.

The remote memory accesses take longer to complete than local memory accesses due to the indirect path that is followed during such accesses, hence the term non-uniform memory access. Even though there are physical hierarchy differences between the remote and local memories, the collection of the local memories forms a single, global address space accessible by all processors in the system. A variation of the NUMA shared-memory model is introduced in the next section.

2.4.4 Hierarchical Shared-Memory Model

A hierarchical memory model is shown in Figure 2.4. Each cluster is considered to be an UMA model since the distance between any processor and any cluster memory is uniform. However, this memory model as a whole is considered to be a NUMA model since the access time to a cluster memory is shorter than the access time to a global memory.

The CEDAR system at the University of Illinois is an example of this memory model [Kuck93]. Each cluster of the CEDAR model is a modified Alliant FX/80 multiprocessor. Global memory accesses from each cluster processor must pass through the cluster network switches and on to the global interconnection network via a global interface. The global interconnection network is a multistage shuffle-exchange network that connects each of the clusters to each of the global memory modules. The network is composed of crossbar switches and data buffers that allows a network bandwidth of 768 Mbytes/sec.

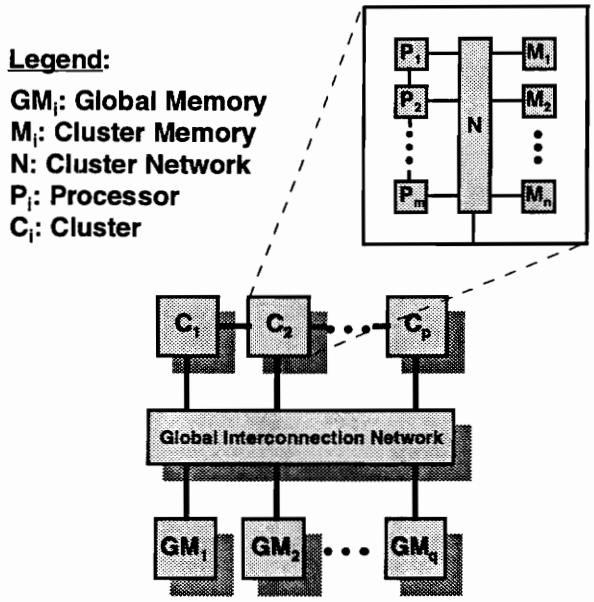


Figure 2.4 Hierarchical shared-memory model.

The next section discusses several different compositions of the interconnection networks present in the UMA, NUMA, and hierarchical memory models.

2.5 Interconnection Networks

The interconnection networks used in the shared-memory models described in the previous section are used to transfer data from any processor to any memory, or vice versa. The arbitrary processor-memory connection need not be *non-blocking* [Clos53], meaning a connection is always possible. However, in order for an interconnection network to be useful in a shared-memory model, a path from every processor to every memory must exist.

Interconnection networks can be placed into one of two main categories: static and dynamic networks. Static networks are interconnection networks where the connections between processor and memory devices, called *nodes*, are fixed. Examples of static interconnection networks include linear arrays, meshes, and hypercubes [Hwang93]. Contrary to static networks, the connections between nodes in dynamic networks can be changed by using intermediate nodes such as *arbiters* or *switches*.

Dynamic interconnection networks come in many varieties. Three types of dynamic networks commonly used in shared-memory interconnection networks include *bus systems*, *crossbar networks*, and *multistage interconnection networks* [Hwang93]. The next few sections will briefly describe and compare these types of interconnection networks in their use in shared-memory systems.

2.5.1 Bus Networks

A typical shared-memory bus network is depicted in Figure 2.5. The bus network is notable for its simplicity and low cost. However, a bus network can handle only one transaction per bus cycle, therefore the processors need to act as arbiters to decide which processor has use of the bus at any given time. Most uniprocessor computer systems have some sort of bus communication between the processor and memory systems. An example of a tightly coupled multiprocessor system which uses a bus for processor to memory communication is the Sequent Symmetry S1 [Hwang93].

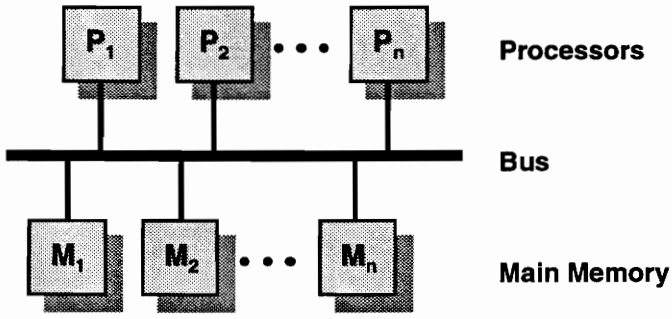


Figure 2.5 Bus-connected multiprocessor system.

Many different bus configurations and protocols have been implemented in recent years, including popular standards such as the VME and Futurebus+ protocols. Even with these innovative arbitrating techniques, bus systems still lack the necessary bandwidth and scalability factors to be competitive in a multiprocessor environment. Several characteristics of bus systems are shown in Table 2.1. The next section introduces the other extreme of interconnection devices called the crossbar network.

2.5.2 Crossbar Networks

In a multiprocessor memory system, full crossbar networks are used to provide fast, non-blocking connections between completely connected processors and memory. The crossbar network itself resembles a large grid with processors arranged along one side of the grid and the memory devices located along an adjacent side, as shown in Figure 2.6.

A connection is made between a processor and memory simply by connecting the horizontal processor data path with the vertical memory data path at their intersection. An example of a machine that uses a crossbar network for processor-memory interconnection is the C.mmp multiprocessor at Carnegie Mellon University [Wulf72].

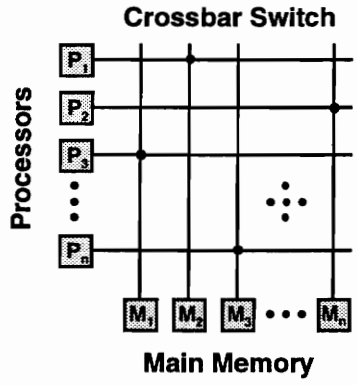


Figure 2.6 Multiprocessor crossbar network.

Crossbar networks solve many of the bandwidth and routability problems that were present in the bus systems, but are cost ineffective when used in large-scaled multiprocessor systems since the switching and wiring complexity grows exponentially

with the number of nodes connected. A summary of crossbar network characteristics is shown in Table 2.1.

2.5.3 Multistage Interconnection Networks

A *multistage interconnection network* (MIN) is a network that connects processors to memory through a series of alternating fixed connection and switch stages. The switches can be dynamically set to create a connection between arbitrary processors and memory devices. Figure 2.7 depicts a certain type of MIN called an 16x16 Omega network [Lawrie75]. The fixed connections between switch stages in the Omega network are called perfect shuffle interconnection patterns. Each switch in an Omega network can be set to one of four configurations, as shown in Figure 2.7.

Many different types of multistage interconnection networks exist. They can be categorized into two different classes: blocking or non-blocking. The Omega network is an example of a blocking MIN, since not all permutations of connections are possible at any given time. More examples of blocking multistage interconnection networks are the baseline network [Wu80], and the delta network [Patel81].

A non-blocking network is one that allows any permutation of connections at any time, such as the Clos [Clos53] and Benes [Benes62] networks. An example of a 16x16 Clos network is shown in Figure 2.8. Notice the redundant paths between any given processor and memory pair in the Clos network.

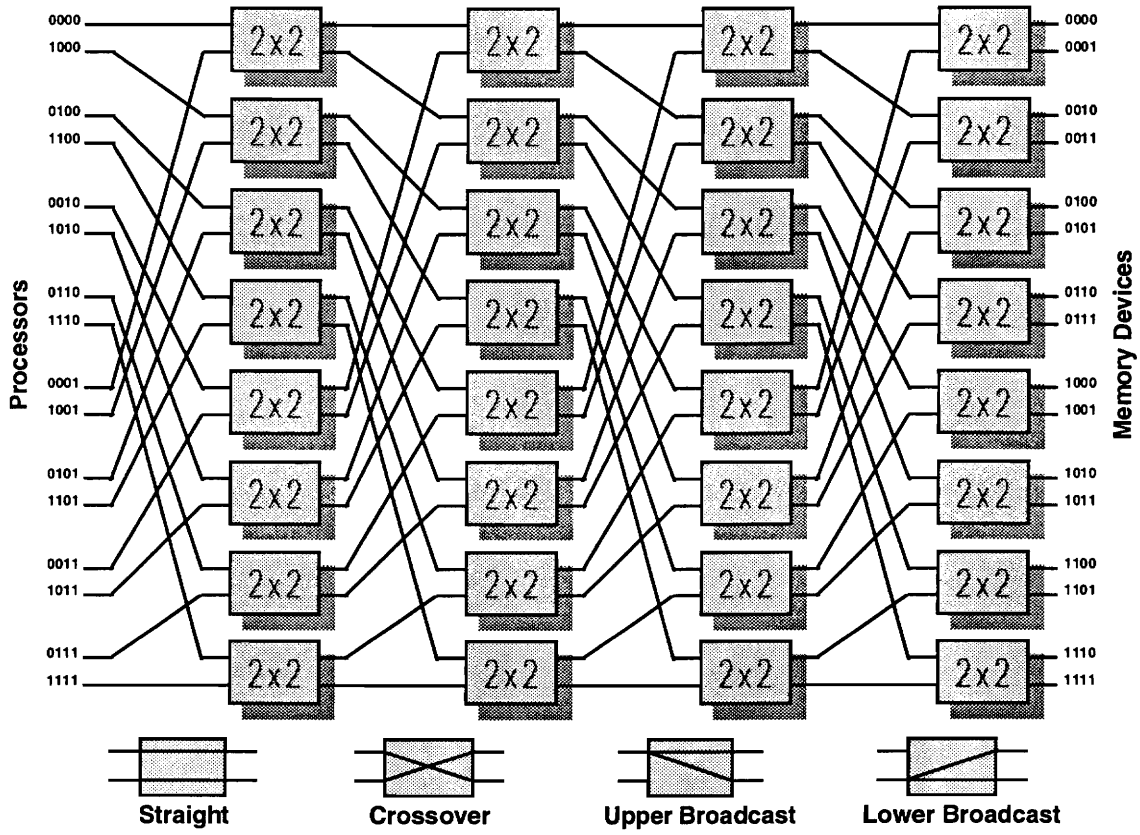


Figure 2.7 A 16x16 Omega network using 2x2 switches and a perfect shuffle as an interstage connection pattern, and the four possible switch settings.

The main difference between all of these examples of multistage interconnection networks is the number of stages, the size and capability of the switch used, and the interconnection pattern used between switch stages. For instance, there are considerable differences between the Omega network shown in Figure 2.7 and the Clos network shown in Figure 2.8, even though they both serve to provide connections between sixteen processors and sixteen memory devices.

Even though two MINs might have the same physical configuration, they can still behave considerably different due to their control structure. For instance, one MIN might have a centralized controller, while the other has a controller that is distributed among its switches. Interconnection network faults within the MIN would have more severe effects on the operation of the distributed controlled MIN due to corruption of data tags [Davis85].

Several multiprocessors systems use MINs to connect processors to memory, including the CEDAR multiprocessor at the University of Illinois [Konicek91]. Multistage interconnection networks offer many advantages over their bus and crossbar counterparts, including increased scalability, sufficient bandwidth and cost efficiency. These characteristics are summarized in Table 2.1.

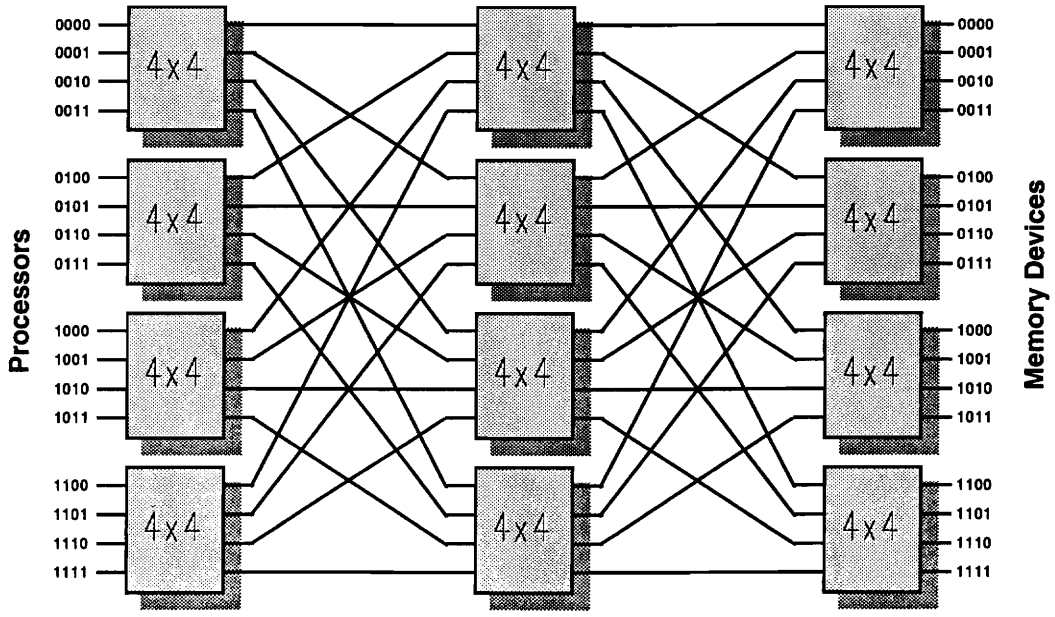


Figure 2.8 A 16x16 Clos network.

The discussion of memory systems and interconnection networks has so far been limited to multiprocessors that use conventional processors. The next section describes in detail a new class of computers that were introduced in Section 2.2.3, called *custom computing machines* (CCMs).

Table 2.1 Summary of Dynamic Network Characteristics [Hwang93].

Network Characteristics	*Bus System*	*Crossbar Switch*	*Multistage Network*
Minimum latency for unit data transfer	Constant	Constant	$O(\log_k n)$
Bandwidth per processor	$O(w/n)$ to $O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring complexity	$O(w)$	$O(n^2 w)$	$O(nw\log_k n)$
Switching Complexity	$O(n)$	$O(n^2)$	$O(n\log_k n)$
Connectivity and routing capability	Only one to one at a time	Some permutations and broadcast, all if non-blocking	All permutations, one at a time
Representative computers	Symmetry S1, Encore Multimax	Cray Y-MP/816, Fujitsu VPP500	BBN TC-2000, IBM RP3
Remarks	Assume n processors on the bus; bus width is w bits	Assume n x n crossbar with line width of w bits	n x n MIN using k x k switches with line width of w bits

2.6 FPGA-based Custom Computing Machines

The introduction of field-programmable gate arrays (FPGAs) in the 1980's ushered in a new paradigm of computing architectures. Originally designed for fast-prototyping of digital systems, FPGAs were later recognized as building blocks for custom computing machines because of their programmable hardware nature. The Splash-2 [Arnold92], Ganglion [Cox91], PAM [Bertin93], and CHAMP [Box94] architectures are several examples of FPGA-based custom computing architectures. An illustration of the Splash-2 system and its SPARC-2 host computer is shown in Figure 2.9. The next section gives an architectural overview of the Splash-2 system.

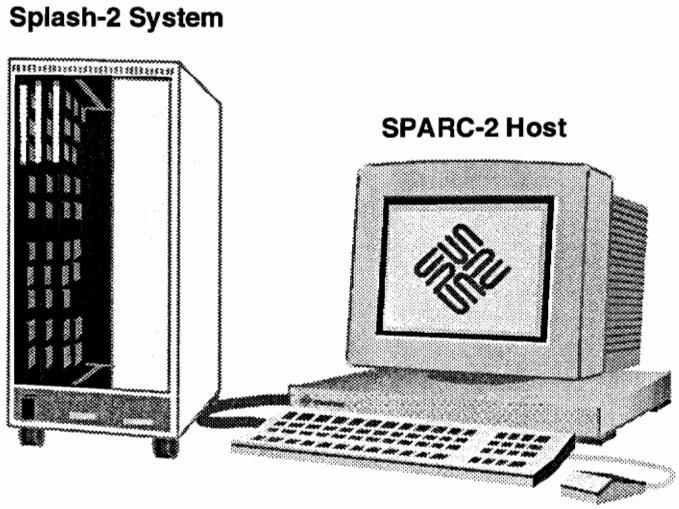


Figure 2.9 Splash-2 system and host computer.

2.6.1 Splash-2 Overview

A brief overview of the Splash-2 system is presented in this section. More details about Splash-2 can be found in the Splash-2 paper [Arnold92] and the Splash-2 hardware document [Burns94].

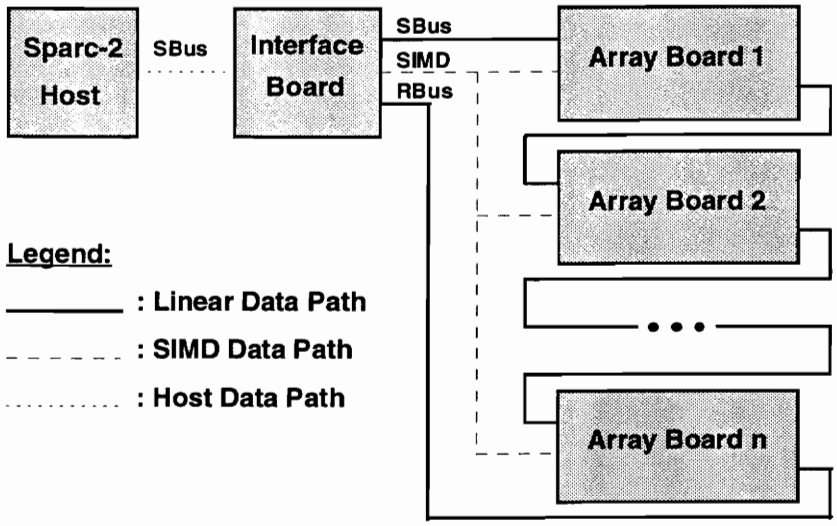


Figure 2.10 Splash-2 architecture.

The Splash-2 system is depicted in Figure 2.10. The system consists of one or more array boards connected to a Sun Microsystem's SPARC-2 host via an interface board and SBus adapter card (not shown in Figure 2.10). Each array board is made up of sixteen linearly connected processing elements (PEs), one SIMD control element, and a configurable crossbar network that connects the processing and control elements to one another, as shown in Figure 2.11. The control and processing elements consist of a Xilinx XC4010 FPGA as the processor core with an attached 256k by 16 bit local memory. The

linear and crossbar data paths are 36 bits wide, and can accommodate clock speeds of up to 25 Mhz.

The Splash-2 system is primarily programmed by using VHDL code that is synthesized into Xilinx Net Format (XNF) code. The XNF code is then partitioned, placed and routed into a Xilinx XC4010 formatted bit-stream. More details on how to program and execute programs on the Splash-2 system can be found in [Arnold92]. The next section describes the Splash-2 local memory architecture in more detail.

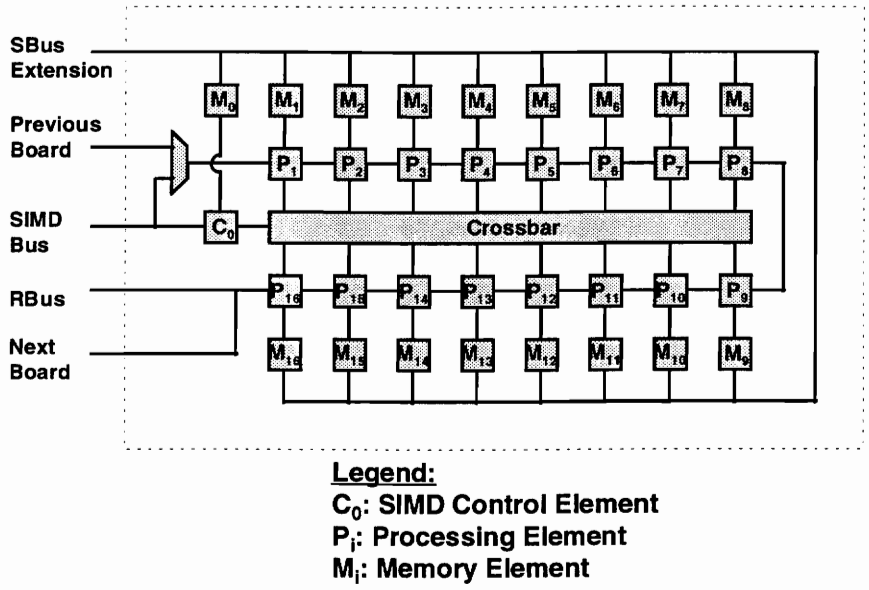


Figure 2.11 Splash-2 array board.

2.6.2 Splash-2 Local Memory Architecture

The Splash-2 memory architecture is based on a distributed unshared-memory model where each PE can directly access its own local memory. Any remote memory access must take place as a request to another PE via the linear or crossbar interconnect. Such remote memory accesses use valuable processor space and interconnection resources, thus making them very expensive and hence seldom used in most applications.

The physical components of the Splash-2 local memory architecture consist of a simple memory controller (not shown in Figure 2.11) and a 256k by 16 bit wide memory module. The memory controller arbitrates between host SBus and PE memory accesses and generates the proper timing signals for the memory module. The next section describes how the Splash-2 local memory architecture can be modified to create a shared-memory architecture.

2.6.3 Splash-2 Shared-Memory Architecture

The Splash-2 memory system can be modified to create a shared-memory system while retaining the original memory characteristics. These modifications are expected to increase the performance of many of the applications run on Splash-2 by allowing each PE to participate in a shared-memory model. In addition to performance increases, a shared-memory architecture will also allow Splash-2 to be programmed more easily by making it easier for HLL compilers to utilize the Splash-2 memory and interconnection resources more efficiently. A shared-memory architecture would also increase the effective memory

bandwidth of each processing element by up to a factor of eight. Simple alterations to the existing processing element I/O definitions and the addition of a memory interconnection network will enable the creation of a hierachical memory architecture on Splash-2. The next chapter describes this hierarchical shared-memory architecture and how it is to be implemented on Splash-2.

Chapter 3

3. Splash-2 Shared-Memory Design

Designing a Splash-2 shared-memory architecture is very much like constructing the tunnel beneath the English Channel. The channel tunnel, or *chunnel*, was built by both English and French construction crews, each starting from their respective side of the English Channel. Very careful planning and construction was required to make sure that the two ends of the tunnel met somewhere in the middle, otherwise the design would be considered a failure.

Although a shared-memory system for Splash-2 is not a multilateral undertaking, it does require careful planning an construction to ensure that the existing array processing elements and the new shared memories come together in such a way as to enhance the

system as a whole. The design must be started from two different ends, just as in the case of the chunnel. The existing Splash-2 system is to be used as the base system onto which the shared-memory system is to be added. A suitable shared-memory architecture must be chosen from those introduced in Chapter 2 that proves to be the best solution when implemented on Splash-2.

This chapter describes the design process undertaken to provide a shared-memory architecture on the Splash-2 system. The modifications for the original system and the new shared-memory architecture are described in depth, including design tradeoffs that are required to ensure that the final design is revolutionary yet feasible to build.

3.1 Modifications to Splash-2

The decision to modify the existing Splash-2 architecture (shown in Figure 3.1), as opposed to designing an entirely new CCM, is made for several reasons. The most important reason stems from the old "bird-in-hand" adage: Virginia Tech already has a perfectly functioning Splash-2 system. To build a new CCM would be like reinventing the wheel, since many of the features already offered on Splash-2 would have to be duplicated on a new CCM design. Features like the crossbar, Futurebus+ backplane connectivity, programming control logic, board-level design of processor interconnect, and many others already exist on Splash-2, and have been proven to work.

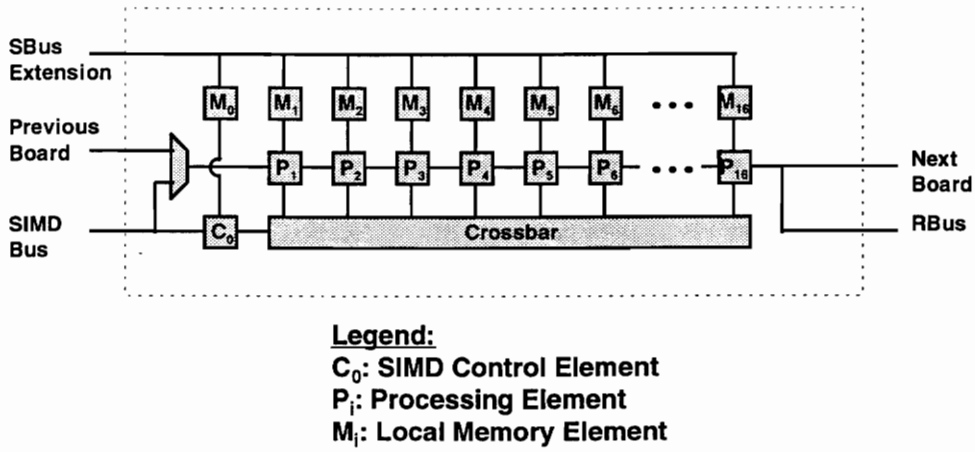


Figure 3.1 Splash-2 distributed non-shared memory architecture.

The only major feature that would be different on the new CCM would be the shared-memory system. The next section describes how this new memory system would have to interface to a Splash-2 array board.

3.1.1 Splash-2 Physical Interfacing Issues

One of the challenges in modifying an existing Splash-2 system is to find the right physical location to implement the shared-memory architecture, since it is not designed for this. The obvious choice is to add a memory daughter-card that would connect to the Splash-2 array board. Such an approach would have minimal impact on the existing processor array board.

A daughter-card approach requires the dimensions of the card be such that it fits inside the Futurebus+ cabinet that houses the Splash-2 array boards. The dimensions of a

Splash-2 array board are 18" x 18". The dimensions of the daughter-card must be slightly less than that of the array board and must not be deeper than 1/2 inch, so as to fit in-between the array boards in the cabinet (see the illustration in Figure 2.9 for an example of the Splash-2 cabinet). Another challenge is how to interface the daughter-card to the array board.

Each processing element has a 256k by 16 bit 20 ns SRAM and controller attached to it via sockets. Since the original memory devices and controllers would not be used in their existing configuration, these can be removed from the array board. If each memory device and its controller is removed, this leaves two possible connection points per processor that can be used to interface the shared-memory daughter-card to each PE on the array board. Each socket provides most of the necessary pin functionality required by a shared-memory network. Some changes do need to be made to provide a complete interface between the Splash-2 array board and shared-memory design. The next section describes some of these changes.

3.1.2 Memory Signal Modifications

The changes that need to made to the I/O configuration of a Splash-2 processing element are minimal, and do not change the functionality of the original array board. The shared-memory design does make use of existing memory signals. For instance, the address and data signals do not need to be modified. The new memory system is also

based on 256k by 16 bit SRAMs. In addition, the memory disable and read/write enable signals do not need to be changed.

Table 3.1 Splash-2 shared-memory signal allocation.

Existing Signal	Existing Signal Description	Shared-Memory Signal	Shared-Memory Signal Description
TDO	Unused Xilinx output signal[1]	XP_GMEM_REQ	Global memory request signal
INIT	Unused Xilinx I/O signal[2]	XP_GMEM_ACK	Global memory acknowledge signal
XP_LED	Processing element LED output signal.	XP_MEM_SEL0	Low-order memory select signal
RBTRIG	Processing element readback trigger signal[3]	XP_MEM_SEL1	High-order memory select signal

The additional PE I/O signals needed by a shared-memory system include additional address or chip select lines to be used to increase the effective address space of each PE. Shared-memory control signals are also needed, depending on the shared-memory architecture that is used. Since nearly all of the I/O capabilities are used by

[1] Test Data Output during boundary scan; User-programmable output after configuration is completed.

[2] Used during programming to clear configuration memory; User-programmable I/O after configuration is completed.

[3] Considered as unused I/O due to the lack of readback functionality with the Splash-2 Quick-and-Dirty Interface Board.

existing signals, some non-essential signals on the Splash-2 processing element need to be replaced by some of the new shared memory signals. These signals, their replacement shared-memory signals, and the signal descriptions are summarized in Table 3.1. The next section describes other small modifications that need to be made to Splash-2.

3.1.3 Other Modifications

Other modifications that need to be made include memory system clock generation and distribution, power distribution, and host access to shared-memory. The clock distribution problem can be solved by the usage of the extra Futurebus+ backplane signals, XTRAIN and XTRAOUT. These signals can be used to transmit and receive clocking signals generated either by the interface board or the shared-memory daughter-card. The power distribution capabilities of the Futurebus+ backplane would need to be studied in order to assess the need for an external power supply to be coupled to the existing one. Finally, the host would still use the memory disable signal to stop the clock and disable all processing elements from accessing the memory devices, but would access all memory devices via new data paths implemented on the daughter-card. The Splash-2 SBus adapter card is robust enough to handle the extra SBus cycles needed to perform modified host accesses [Kleinfelder94].

The modifications to the existing Splash-2 design depend heavily on which shared-memory architecture is implemented. The next section describes several possible architectures and how they could be implemented on Splash-2.

3.2 Survey of Shared-Memory Architectures

The shared-memory and interconnection network architectures introduced in Chapter 2 are again discussed and compared in this section. The premise for this discussion and comparison is that the architectures are to be used to modify the Splash-2 memory system. The network characteristics in Table 2.1 will be used to promote or refute the possibility of using the interconnection networks to implement the UMA, global NUMA, and hierarchical NUMA shared-memory architectures on Splash-2. The conclusion of this section assesses the characteristics of each architecture, and presents a feasible implementation of a Splash-2 shared-memory design.

3.2.1 UMA Design

The UMA shared-memory architecture introduced in Chapter 2 is one architecture that can be used for the Splash-2 shared-memory architecture. Figure 3.2 depicts a Splash-2 implementation of an UMA shared-memory design. Connecting all sixteen processing elements to all sixteen memories so that the memory access time is the same across all connections requires a complex interconnection network. Several of the interconnection systems described in Chapter 2 could be used to implement the UMA global interconnection network.

The bus network system is one interconnection network that could be used in an UMA design. In the Splash-2 case, all sixteen processors would share the bus in a single bus network, which would prove to be a costly bottleneck for many applications. The

controller for this network would be fairly simple, but could not offer much to improve the contention problems inherent in a single bus network. A multiple bus network might offer some improvement with the contention problem, but would require a much more complex controller. Although relatively inexpensive, a bus system is not considered a good network to use in this case because of its poor performance in handling network contention. The scalability of a bus network is also relatively low, thus ruling out the bus network for future multi-array board shared-memory designs.

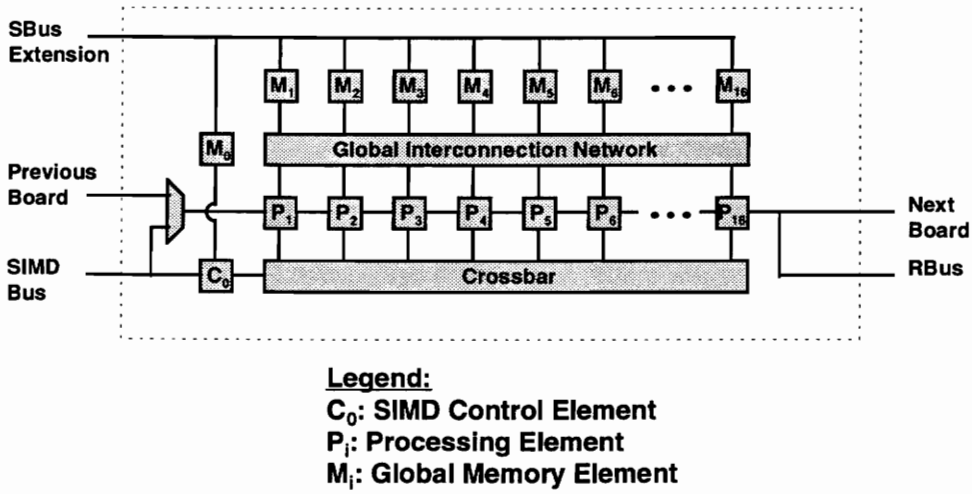


Figure 3.2 Splash-2 UMA shared-memory architecture.

A crossbar network is an alternative approach to the bus system that is nearly opposite in its characteristics. The crossbar network offers a solution that has no contention or bottlenecks between processors and memory. The downside of the crossbar

is the cost of implementation, mainly due to the sheer size of the crossbar needed for this design.

The dimensions of the crossbar are 16 x 16 x 39, since there are sixteen processing elements, sixteen memory devices, and thirty-nine memory signals (twenty effective address bits, sixteen data bits, and three memory control bits) needed for this design. The crossbar also needs to be bi-directional in the sixteen data bits, otherwise an additional sixteen signals need to be added to the pin count in order to implement the two data paths. Without counting the crossbar overhead signals such as latch and output enables, the crossbar would need as many as 13824 pins. It would take seventy-two 192-I/O Xilinx 4013 FPGAs to implement such a crossbar [Xilinx94], or forty-four 320-I/O ICube IQ320 Field Programmable Interconnect Devices (FPIDs) [ICube94] to create such a crossbar. The average cost of the Xilinx parts is around $500, which would put the cost of implementing the network alone at $36,000. The cost-efficiency of the crossbar network is quite poor in this case, thus ruling it out as a possibility.

The last interconnection network that is considered to implement the UMA architecture is the multistage interconnection network (MIN). The MIN design is physically realizable as a Clos network, as shown in Figure 2.8, but the control algorithm to configure the switches is too complex and requires a maximum $n*\log(n)$ clock cycles to complete [Benes62]. Another non-blocking MIN, called a self-routing Benes network needed only n cycles to configure the switches [Nassimi81]. Neither of these networks can be switched fast enough to provide effective data transfer, which would cause delays

during memory accesses. As many as sixteen clock cycles would be needed to determine the switch settings, which defeats the purpose of having a non-blocking MIN network. A simple bus network could get similar performance at considerably lower cost.

Another implementation is to implement a blocking MIN using four eight-by-eight crossbar network switches arranged in an Omega network that use complex contention arbitrators and buffers to avoid contention, as used in the CEDAR project [Kovicek93]. The eight-by-eight crossbar network is still too expensive and the controller needed to set up the switches and buffers is too complex for a Splash-2 design. The controller would have more compute power than a processing element has.

Based on evidence presented above, the UMA architecture is not a feasible solution to the shared-memory challenge. The next section describes another possible candidate shared-memory architecture known as the global NUMA shared-memory architecture.

3.2.2 Global NUMA Design

A global NUMA shared-memory architecture that requires all sixteen processing elements to be connected to a local memory and fifteen distributed shared memories can be ruled out on a single technicality. This implementation would require each PE to have an additional 39 unused I/O signals to be reserved for the additional memory signals needed by a second memory port. These extra I/O pins simply do not exist.

An alternative to this approach would be to multiplex the two signals onto the existing memory I/O signals present on each PE. This solution only serves to add another layer to the network needed to connect the PE to the fifteen memory devices. In addition to reduced performance, the complexity of the controller would be increased since it would have the additional task of arbitrating between local and external memory accesses.

The global NUMA shared-memory architecture can be ruled out as a viable solution for performance and complexity problems stated above. The remaining solution that was covered in Chapter 2 is the hierarchical NUMA shared-memory architecture.

3.2.3 Hierarchical NUMA Design

The hierarchical NUMA shared-memory design, as it would be implemented on Splash-2, is depicted in Figure 3.3. Each processing element in this implementation has eight memory devices available for access, as opposed to the sixteen memory devices in the UMA or global NUMA cases described above.

The clusters are composed of four processing elements, four memory devices, and a four-by-four cluster interconnection network. Each cluster is connected to a port of the global interconnection network. Since an array board has sixteen processing elements (PEs), and each cluster has four PEs each, the global interconnection network has four clusters attached to it. In order to preserve the symmetry between the cluster and global interconnection networks, only one connection per cluster to the global network is used.

This reduces the complexity of the global network, and also reduces design time and cost because the cluster and global networks are nearly identical.

The task of designing the cluster and global interconnection networks follows some of the same arguments as in the UMA architecture described in the previous section. The bus network still lacks the performance and scalability characteristics needed for such an implementation. The crossbar network, however, is easier to implement using this hierarchical NUMA design than was the case in the UMA architecture. The wiring complexity is reduced by a factor of sixteen per network, but the overall complexity is reduced by a factor of only 3.2, because one global and four cluster networks are needed to implement the design. The Xilinx and I-Cube devices mentioned earlier also have very complex controllers that would not allow these devices to run at clock frequencies of 10-20 Mhz.

The MIN strategy is the best approach for implementing the global and cluster networks in this hierarchical NUMA architecture. The reduced network dimensions allow the MIN switch stages to be comprised of simple two-by-two switches. The control algorithm complexity for a four-by-four non-blocking MIN network is still $\log(n)$, but here $n = 4$ instead of $n = 16$. With a memory network controller operating at 50 Mhz, the worst-case cluster access frequency with overhead included is slightly higher than 8 Mhz. The alternative to non-blocking MIN networks are the blocking networks such as Omega and Delta networks described in Chapter 2. The complexity of the blocking network controller is inherently less than that of a non-blocking network controller for the same

network dimensions, since switch settings for allowable permutations can be configured in unit time for some switches [Nassimi81]. The blocking network controller can be augmented to handle contention within the switches by buffering and pipelining of data [Lawrie75]. One such controller is described further in Section 3.3.

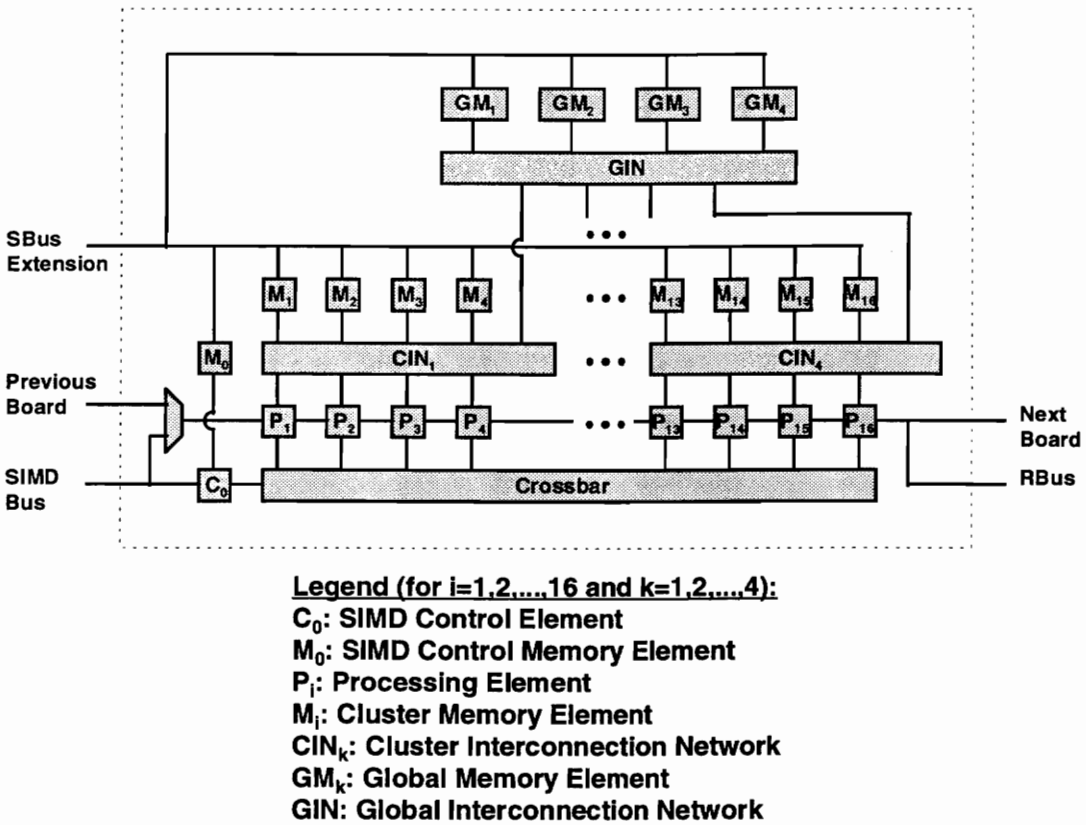


Figure 3.3 Splash-2 hierarchical NUMA shared-memory architecture.

Based on the criterion that the shared-memory network must have high performance and high scalability characteristics for relatively low-cost, the hierarchical

NUMA shared-memory architecture is the suitable choice for the Splash-2 shared-memory design. The next section describes the hierarchical NUMA implementation in more detail.

3.3 Hierarchical NUMA Shared-Memory Design

The components that make up the hierarchical NUMA architecture used for the Splash-2 shared-memory design are discussed in this section. The section begins with a top-level description of the implementation, then proceeds to describe the cluster and global networks in more detail.

3.3.1 Top-Level Design

The design in Figure 3.3 depicts a simplified top-level view of the actual implementation. The cluster and global networks are very similar in structure. Each network is responsible for proving connectivity between four memory devices and four processors or clusters. The main difference between the two networks stems from the fact that each cluster network has additional buffers and controllers to handle multiple accesses to the single global memory port present on each cluster network. Also, each cluster is responsible for working with the other clusters to generate the processor clock. In some circumstances, the global network controller can also take part in processor clock generation to ensure proper boundary synchronization.

The decision to use a 50 Mhz memory controller clock has a large impact on the precision of the design. This decision required the design to be modeled using very

precise timing constraints of the actual components chosen to be used. The design described below takes into consideration all the timing constraints found in [Xilinx94], [IDTMemory93], [IDTLogic92], [Altera94], and [AMD94]. The structural-level VHDL simulation of the Splash-2 shared-memory models a physical implementation as close as possible.

The operation of the shared-memory system from a processing element's point of view is shown in Table 3.3. A VHDL entity definition of the processing element is shown in Appendix A. Sections 3.2.2 and 3.2.3 describe the cluster and global memory designs, respectively.

Table 3.2 Shared-memory PE memory signal overview.

Signal Name	Signal Description	Operation
XP_GMEM_REQ	Global memory request signal	Initiates a global memory access when a '1'; denotes a cluster memory access otherwise.
XP_GMEM_ACK	Global memory acknowledge signal	A '1' denotes that a global memory access is in progress by the PE; The finish of the global access is signified by the deassertion of this signal.
XP_MEM_SEL(1..0)	Memory select signal	This two bit signal is used to extend the address space of the PE from one memory device to four.

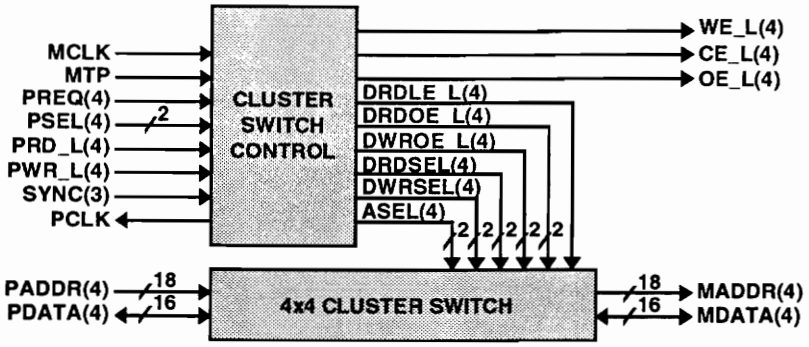
XP_MEM_RD_L	Memory read enable signal	The memory device OE_L and CE_L signals are derived from this signal.
XP_MEM_WR_L	Memory write enable signal	The memory device WE_L and CE_L signals are derived from this signal.
XP_MEM_ADDR(17..0)	Memory address signal	Used as the cluster and global memory address signal.
XP_MEM_DATA(15..0)	Memory data signal	Used as the cluster and global memory data signals.
XP_MEM_DISABLE	Memory disable signal	Used to disable all memory accesses so the host may access the memory.

3.3.2 Cluster Shared-Memory Network Design

The cluster shared-memory network portion of the design is comprised of two main parts: the cluster switch network and the global data registers/multiplexers. The cluster switch and controller are shown in Figure 3.4. The purpose of the cluster switch is to transport the memory signals from each of the processing elements to the appropriate memory device in the same cluster. A detailed layout of the 4 x 4 cluster switch network is shown in Figure 3.5.

The cluster switch network is called a modified Omega network because the switches do not have the same possible configurations as the switches in a true Omega network. The switches used in the cluster switch network do not have broadcast

capability. However, they are capable of establishing only one crossover or straight connection established at a time. The structural VHDL simulation model of the switch network uses the AMD Am29C983A as the core switch element. The VHDL model of the Am29C983A includes timing constraints found in [AMD94] to ensure accurate simulation results.



LEGEND

Key for diagram:

1. '_L' suffix denotes;
2. Number in parenthesis denotes number of sets of signals;
3. Number next to slashes on the signal denotes number of wires in signal;
4. Signal descriptions:

MCLK	: Memory controller clock
TP	: Timing pulse
PREQ	: Global memory request
PSEL	: Cluster memory select
PRD_L	: Read enable
PWR_L	: Write enable
SYNC	: PCLK sync pulse
PCLK	: Processor clock
PADDR	: Processor address
PDATA	: Processor data
WE_L	: Memory write enable
CE_L	: Memory chip enable
OE_L	: Memory output enable
DRDLE_L	: Data read latch enable
DRDOE_L	: Data read output enable
DWROE_L	: Data write output enable
DRDSEL	: Data read select
DWRSEL	: Data write select
ASEL	: Address select
MADDR	: Memory address
MDATA	: Memory data

Figure 3.4 Cluster switch and controller.

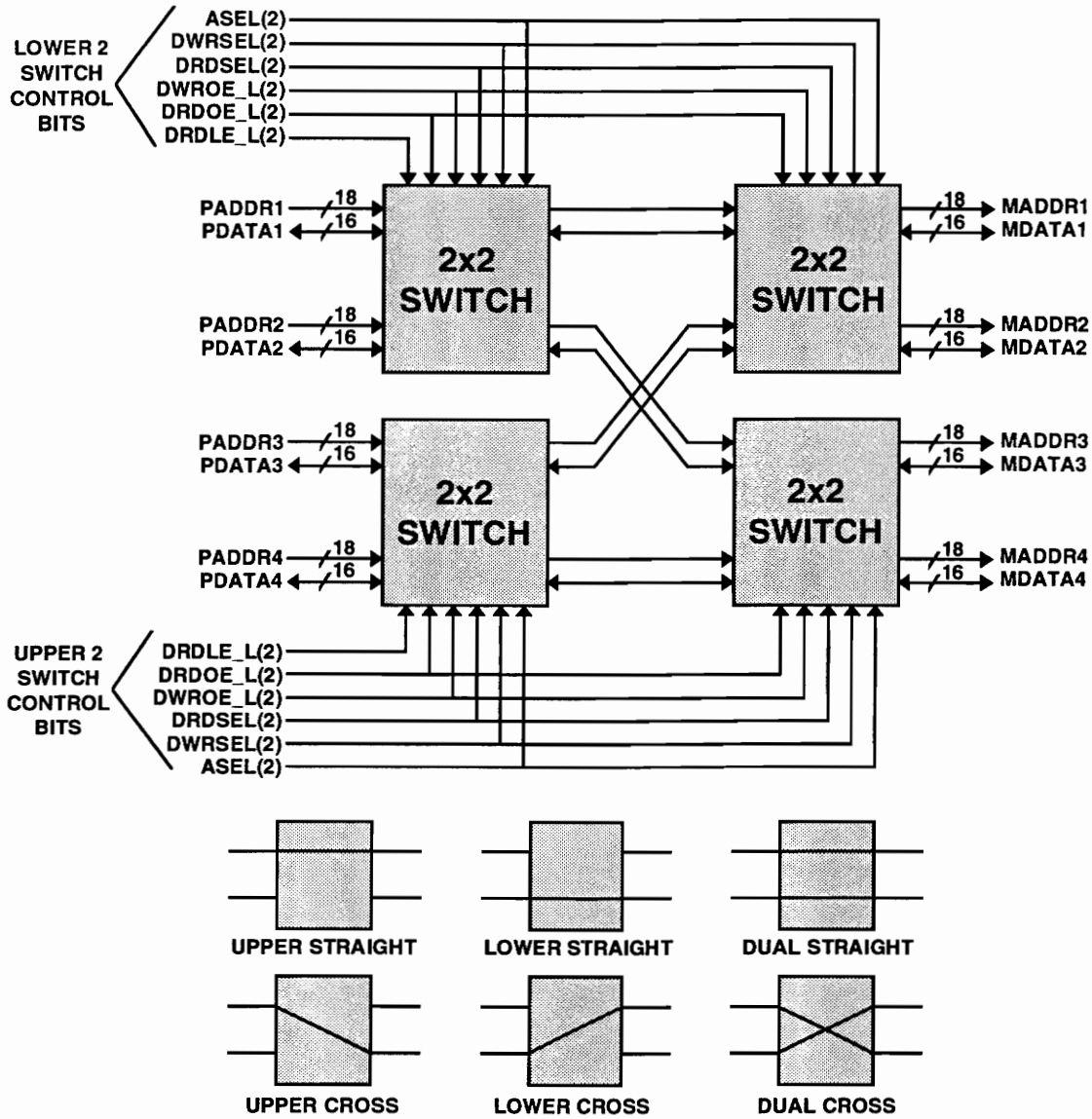


Figure 3.5 4 x 4 modified Omega network and six possible 2 x 2 switch configurations.

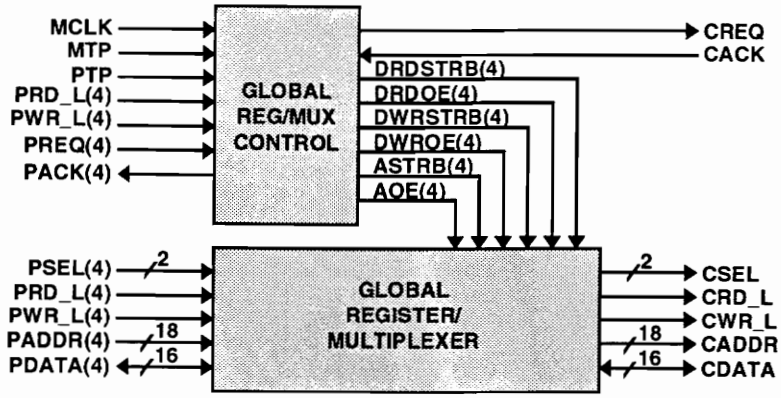
The controller for the cluster switch network is not only responsible for controlling the switch configurations, but also must arbitrate any switch contention brought on by

memory requests to the same switch port. The controller state machine is programmed to give priority according to the following rules:

- Memory write accesses have priority over memory read accesses, and
- Processing elements PE1, PE2, PE3 and PE4 within a cluster have priority P such that P(PE1) > P(PE2) > P(PE3) > P(PE4).

The network controller decodes memory requests according to these priority rules, and sets the switch configurations accordingly. Each network controller is also responsible for generating a processor clock. The cycle length of the processor clock varies depending on the type and quantity of pending memory accesses. This feature is used to increase the performance of the network during non-blocking switch configurations. The performance of the variable processor clock is discussed in greater detail in Chapter 4.

A detailed behavioral VHDL description of the network controller can be found in Appendix B. The simulation model of the network controller includes timing constraints of the Altera EPM70128 *electronically programmable logic device* (EPLD) found in [Altera94]. The VHDL code for the controller can then be translated directly into the EPLD format using Viewlogic VHDL synthesis tools.



LEGEND

Key for diagram:

1. '_L' suffix denotes;
2. Number in parenthesis denotes number of sets of signals;
3. Number next to slashes on the signal denotes number of wires in signal;
4. Signal descriptions:

MCLK	: Memory controller clock
MTP	: Memory timing pulse
PRD_L	: Processor memory read enable
PWR_L	: Processor memory write enable
PREQ	: Processor memory request
PACK	: Processor memory acknowledge
PSEL	: Processor memory select
PADDR	: Processor address
PDATA	: Processor data
CREQ	: Cluster memory request
CACK	: Cluster memory acknowledge
DRDSTRB	: Data read strobe
DRDOE	: Data read output enable
DWRSTRB	: Data write strobe
DWROE	: Data write output enable
ASTRB	: Address strobe
AOE	: Address output enable
CSEL	: Cluster memory select
CRD_L	: Cluster memory read enable
CWR_L	: Cluster memory write enable
CADDR	: Cluster address
CDATA	: Cluster data

Figure 3.6 Global register/multiplexer with controller.

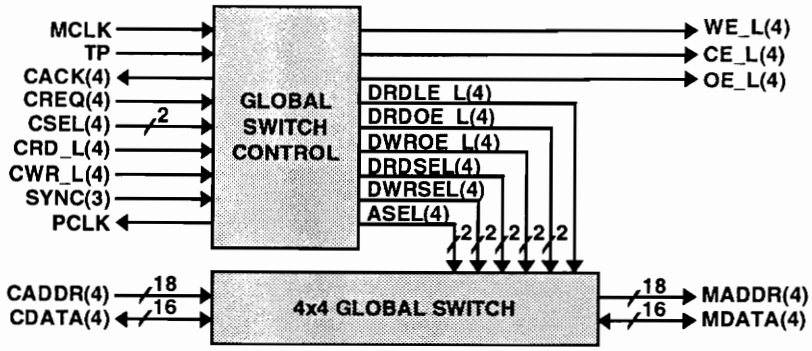
Another component of the Splash-2 shared-memory cluster is the global memory register/multiplexer, shown in Figure 3.6. Each cluster has four of these register/mux

components that is used to queue up global memory access requests according to the priority rules described above. The controller for the global register/mux component is designed to fit in an Altera EPM7064 EPLD. The controller VHDL model can be found in Appendix C and is designed using the EPM7064 timing constraints found in [Altera93]. The next section describes the global shared-memory network and the role it plays in the Splash-2 shared-memory design.

3.3.3 Global Shared-Memory Design

The global shared-memory design is shown in Figure 3.7. It is composed of a global switch network and controller that is very similar to the cluster switch network and controller shown in Figure 3.4. In fact, the global switch network itself is identical to the cluster switch network shown in Figure 3.5. As stated previously, the main difference between the two networks is that the global switch network controller must use handshaking signals since global memory accesses can possibly take more than one processor clock cycle to complete.

The global shared-memory network is designed to fit inside a single Am29C983A component. The behavioral VHDL-model for this design is found in Appendix D. The next section summarizes the features and implementation of the Splash-2 shared-memory design.



LEGEND

Key for diagram:

1. '_L' suffix denotes;
2. Number in parenthesis denotes number of sets of signals;
3. Number next to slashes on the signal denotes number of wires in signal;
4. Signal descriptions:

MCLK	: Memory controller clock
TP	: Timing pulse
CACK	: Cluster global memory acknowledge
CREQ	: Cluster global memory request
CSEL	: Cluster memory select
CRD_L	: Cluster memory read enable
CWR_L	: Cluster memory write enable
SYNC	: PCLK sync pulse
PCLK	: Processor clock
CADDR	: Cluster memory address
CDATA	: Cluster memory data
WE_L	: Memory write enable
CE_L	: Memory chip enable
OE_L	: Memory output enable
DRDLE_L	: Data read latch enable
DRDOE_L	: Data read output enable
DWROE_L	: Data write output enable
DRDSEL	: Data read select
DWRSEL	: Data write select
ASEL	: Address select
MADDR	: Memory address
MDATA	: Memory data

Figure 3.7 Global shared-memory network and controller.

3.4 Design Evaluation

This section serves to provide a brief evaluation of the Splash-2 shared-memory design prior to the more complete performance test results shown in Chapter 4. The rest of this chapter is used to describe any architectural tradeoffs made during the design phase and also to summarize all of the features of the Splash-2 shared-memory design.

3.4.1 Preliminary Area and Power Estimates

It is difficult to predict the area and power requirements of a design without actually building it, but preliminary estimates can be calculated in order to gauge the feasibility of such an undertaking. A summary of the preliminary area and power estimates is given below:

- The area of the components used to construct the design is approximately 73 square inches. This area is almost 4.5 times smaller than that of the area of one side of an 18SU Futurebus+ board.

- The average power supply current consumed by the components is approximately 33 amperes. The maximum power supply current is approximately 62 amperes[1]. This corresponds to a maximum power dissipation of approximately 300 Watts, which can be supplied by the Futurebus+ backplane.

[1] This average power supply consumption is difficult to determine accurately since much of it is based on maximum AC switching characteristics of the components. The maximum power supply current will probably never be reached.

These initial area and power estimates indicate that it is physically feasible to implement the design. The preliminary cost estimates of implementing the Splash-2 shared-memory discussed in the next section will reveal whether or not it is a cost-effective solution.

3.4.2 Preliminary Cost Evaluation

The cost evaluation of the Splash-2 shared memory network is a very subjective statistic. This evaluation is based on the component costs at the time this chapter was written. The 1995 first fiscal quarter cost of the components is approximately $7600, of which $5600 is the cost of the 12 ns IDT 256x by 4-bit SRAMs used as the memory devices. Since many vendors offer to donate their products to universities for testing and evaluation, the cost for memory devices could be reduced to nil. The second most expensive component used in the design was the AMD switch components, which accounted for nearly $1200 of the total component cost. A conservative cost estimate of a six-layer sub-18SU board that would be used in this design is approximately $6000. With miscellaneous costs added in, the approximate total cost per array board is $8000 to $14,000.

The cost-efficiency of the design is an even more subjective statistic than the absolute cost. The design tradeoffs that were made considerably increased the cost-efficiency of the design. For instance, the decision to use the hierarchical NUMA design reduced the complexity of the network switches. This decision essentially lowered the cost of the switches by approximately four to ten times, depending on which shared-

memory architecture is used as an alternative. The decision to use IDT SRAMs instead of the slightly faster Cypress SRAM devices reduced the cost of the memory devices nearly by a factor of two. Overall, the hierachical shared-memory design described in this chapter is a fairly cost-efficient implementation.

3.4.3 Feature Summary

A summary of many of the features of the Splash-2 shared-memory design are shown below:

- The address space of a processing element has increased by a factor of eight. Each PE now has 4 Mbytes of memory that it can access directly.
- The maximum bandwidth of each network switch is 400 Mbytes/second, which gives a total network bandwidth of 2 Gbytes/second. This corresponds to a maximum available memory bandwidth of 125 Mbytes/second per PE. This more than twice the bandwidth of the existing Splash-2 processing element.
- The new memory system supports shared-memory hierarchy, which helps to increase the performance of many parallel algorithms. In addition, it supports all existing applications designed for Splash-2.

The next chapter describes the performance of the Splash-2 shared-memory tests and applications in more detail.

Chapter 4

4. Performance Results

This chapter is composed of four sections that present performance results from both benchmark testing and several test applications. The test application systems are used to test some of the different types of memory accesses used in typical parallel algorithms. Some of these access types include deterministic memory accesses (memory address is known at compile-time), non-deterministic memory accesses (memory address is known at run-time), and global memory accesses. The first section presents results obtained from general memory access testing of both the cluster and global memory systems.

4.1 Benchmark Test Results

Fairly extensive benchmark testing is undertaken to provide performance specifications that can be used by Splash-2 programmers and compiler developers. The results in this section can be used as cost functions assigned to cluster and global memory accesses that can be used when designing an application. A programmer or compiler can use these cost functions to compute the access time for cluster accesses or to compute the latency of global accesses. The next few sections describe the performance of many different types of memory accesses possible on the Splash-2 shared-memory architecture.

4.1.1 Cluster Memory Performance

Cluster memory accesses can be performed by any of the four PEs assigned to a given cluster. The performance results from cluster memory testing are shown in Table 4.1. These test results are a fully representative set of delay values accrued while performing the different types of cluster memory accesses. The *PCLK* signal is clock used by the Splash-2 processing elements for synchronous execution of operations. The *MCLK* signal is the 50 Mhz clock used by the cluster and global memory controllers. The *PCLK* signal is generated so that all cluster memory accesses are completed in one *PCLK* cycle. This variable-length processor clock is designed to optimize best-case non-blocking memory accesses, while still allowing worst-case memory accesses to take place in one *PCLK* cycle. The test results in Table 4.1 are obtained while no global memory accesses are taking place.

Table 4.1 Cluster Memory Performance Results

P0	P1	P2	P3	PCLK Period (ns)	No. MCLK Cycles	Remarks
idle	idle	idle	idle	40	2	No memory accesses of any kind.
W(M0)	-	-	-	60	3	P0 writing M0; other P's not writing M0 or reading M*.
W(M0)	W(M0)	-	-	80	4	P0,1 writing M0; other P's not writing M0 or reading M*.
W(M0)	W(M0)	W(M0)	-	100	5	P0,1,2 writing M0; other P's not writing M0 or reading M*.
W(M0)	W(M0)	W(M0)	W(M0)	120	6	All P's writing M0.
W(M0)	W(M1)	W(M2)	W(M3)	80	4	Second level switch test. Pi reading Mi.
W(M0)	W(M2)	W(M1)	W(M3)	60	3	Local memory write test. All P's writing local memory.
R(M0)	-	-	-	60	3	P0 reading M0; other P's not reading M0 or writing M*.
R(M0)	R(M0)	-	-	80	4	P0,1 reading M0; other P's not reading M0 or writing M*.
R(M0)	R(M0)	R(M0)	-	100	5	P0,1,2 reading M0; other P's not reading M0 or writing M*.
R(M0)	R(M0)	R(M0)	R(M0)	120	6	All P's reading M0.
R(M0)	R(M1)	R(M2)	R(M3)	120	6	Second level switch test. Pi reading Mi.
R(M0)	R(M2)	R(M1)	R(M3)	60	3	Local memory read test. All

P0	P1	P2	P3	PCLK Period (ns)	No. MCLK Cycles	Remarks
						P's reading local memory.
W(M0)	R(M0)	-	-	80	4	Same memory W/R test.
W(M0)	W(M0)	R(M0)	-	100	5	Same memory W/W/R test
W(M0)	R(M0)	R(M0)	-	100	5	Same memory W/R/R test
W(M0)	R(M0)	R(M0)	R(M0)	120	6	Same memory W/R/R/R
W(M0)	W(M0)	R(M0)	R(M0)	120	6	Same memory W/W/R/R
W(M0)	W(M0)	W(M0)	R(M0)	120	6	Same memory W/W/W/R
W(M2)	R(M0)	R(M3)	W(M1)	80	4	Different memory W/R test.
				90	**4.5**	**Average results**

4.1.2 Global Memory Performance

Global memory accesses can be performed by any of the PE's in any of the clusters. The performance results from global memory testing are shown in Table 4.2. These test results are representative of the worst-case latencies encountered during different types of memory accesses. All 2^{32} combinations of global memory accesses are not tested, but the most commonly used configuration are shown. Most latency values can be obtained by combining and interpolating the results shown in Table 4.2. The test results shown in the table are carried out assuming that no cluster memory accesses are taking place, and that memory accesses can take multiple processor clock cycles to complete. However, the total access time required to complete all global memory

accesses is absolute, and is only based on the number of *MCLK* cycles required to complete the accesses.

Table 4.2 Global memory access test results (N = 1,2,3, and 4, where results are listed as (N = 1, N = 2, N = 3, N = 4)).

Test No.	Test Description	No. PCLK Cycles	Access Time (ns)	No. MCLK Cycles
1	No memory accesses	(1,1,1,1)	N/A	(2,2,2,2)
2	N cluster 0 processors writing GM0.	(4,6,10,10)	(160,240, 400,400)	(8,12,20,20)
3	N cluster 0 processors writing GM0, N cluster 1 processors writing GM0.	(4,7,9,12)	(160,280, 360,480)	(8,14,18,24)
4	N cluster 0 processors writing GM0, N cluster 1 processors writing GM0, N cluster 2 processors writing GM0.	(5,8,11,13)	(200,320, 440,520)	(10,16, 22,26)
5	N cluster 0 processors writing GM0, N cluster 1 processors writing GM0, N cluster 2 processors writing GM0, N cluster 3 processors writing GM0.	(5,9,12,16)	(200,360, 480,640)	(10,18, 24,32)
6	N cluster 0 processors reading GM0.	(4,6,8,10)	(160,240, 320,400)	(8,12,16,20)
7	N cluster 0 processors reading GM0, N cluster 1 processors reading GM0.	(6,7,9,12)	(240,280, 360,480)	(12,14, 18,24)
8	N cluster 0 processors reading GM0, N cluster 1 processors reading GM0, N cluster 2 processors reading GM0.	(6,9,12,14)	(240,360, 480,560)	(12,18, 24,28)
9	N cluster 0 processors reading GM0, N cluster 1 processors reading GM0, N cluster 2 processors reading GM0, N cluster 3 processors reading GM0.	(7,10, 12,16)	(280,400, 480,640)	(14,20, 24,32)
10	N cluster 0 processors R/W GM0, N cluster 1 processors R/W GM1, N cluster 2 processors R/W GM2,	(5,9,12,16)	(200,360, 480,640)	(10,18, 24,32)

Test No.	Test Description	No. PCLK Cycles	Access Time (ns)	No. MCLK Cycles
	N cluster 3 processors R/W GM3.			
11	*N* cluster 0 processors R/W GM0, *N* cluster 1 processors R/W GM2, *N* cluster 2 processors R/W GM1, *N* cluster 3 processors R/W GM3.	(4,6,8,10)	(160,240, 320,400)	(8,12,16,20)
	Average	(5,8,10,12)	(200,320, 400,480)	(10,16, 20,24)

4.1.3 Combined Cluster/Global Memory Performance

The performance of combined cluster/global memory performances can be obtained by combining the results found in Table 4.1, Table 4.2, and an understanding of the interaction between cluster and global memory systems. As can be seen in the tables, global accesses are inherently slower than similar cluster accesses, due to the longer data paths between processors and global memory. The memory systems are designed in order to optimize cluster memory accesses, which means that global accesses have increased complexity in order to reduce the complexity of the cluster memory network. This higher complexity causes a irregular cost increases in the results in Table 4.3 since global memory accesses depend on the behavior of the cluster memory. Despite the relative increase in global cost functions with respect to their cluster counterparts, global memory accesses are still very useful, as will be shown in Section 4.4.

The interpretation of the data found in Table 4.2 remains the same despite the addition of global accesses. The number of *MCLK* cycles required to complete global accesses while concurrent cluster accesses are taking place remains relatively the same as shown in Table 4.3. It can vary either way by the number of *MCLK* cycles required to complete pending cluster accesses. This *MCLK* differential is the cost of synchronizing the PCLK signal with the global memory acknowledge signal.

The *PCLK* performance for a given cluster/global memory access combination can be obtained from the two tables shown above by using the following method (assuming multiple processor clock cycle global memory accesses):

1. Obtain cluster memory access results directly from Table 4.1.
2. Obtain number of MCLK cycles required to complete global memory accesses directly from Table 3.2.
3. Calculate the number of PCLK cycles for global accesses is equal to X*Y/Z where X is number of MCLK cycles required to complete global accesses, Y is the MCLK period (in ns), and Z is the PCLK period (in ns).

The test results found in this section have direct effect on sample applications. The next several sections use the results shown above to evaluate the performance results for three diverse test applications.

4.2 Deterministic Memory Access Test Results

Deterministic memory accesses are defined as memory reads and writes that are performed when the addresses are known at compile-time. The application used to test deterministic memory accesses is the *scalar triple product* of three vectors. The scalar triple product applied to vectors $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ is defined as $\boldsymbol{a} \bullet \boldsymbol{b} \times \boldsymbol{c}$ or $\boldsymbol{a} \times \boldsymbol{b} \bullet \boldsymbol{c}$, where $\boldsymbol{a} \bullet \boldsymbol{b} \times \boldsymbol{c} = a_x(b_y c_z - b_z c_y) - a_y(b_x c_z - b_z c_x) + a_z(b_x c_y - b_y c_x)$. The scalar triple product can be used to calculate the volume enclosed by the three vectors $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$, or to determine vector coplanarity [Edwards85].

The performance of the parallel implementation of the scalar triple product on the Splash-2 shared-memory simulator (shown in Figure 4.1) is as follows:

- Number of processor clock cycles between results is four clock cycles.
- Average processor clock cycle frequency: 10 Mhz
- Speedup over sequential algorithm[1]: ~9.7

The algorithm shown at the top of Figure 4.2 works by having processors P_0-P_3 perform the vector cross product of b and c while P_3 performs the scalar product of a with the result from the vector cross product. This same implementation on an unshared-memory Splash-2 system would have consumed at least five processing elements. Since each

[1] This speedup factor is obtained by dividing the number of clock cycles required by the parallel algorithm described in Figure 4.1 by the number of clock cycles required by a single-processor sequential implementation of the same algorithm.

processor in this case needs at least one memory attached to it, and the scalar product calculation would have to be spread across two processors. Also, the compiler would have to spread the vectors across the memories in a non-regular fashion, which can be very complex and expensive.

Cycle	P_0	P_1	P_2	P_3
1	$V_0 \Leftarrow M_0(i+1)$	$V_0 \Leftarrow M_0(i+2)$	$V_0 \Leftarrow M_0(i)$	Idle
2	$V_1 \Leftarrow V_0 * M_1(i+2)$	$V_1 \Leftarrow V_0 * M_1(i)$	$V_1 \Leftarrow V_0 * M_1(i+1)$	Idle
3	$V_0 \Leftarrow M_0(i+2)$	$V_0 \Leftarrow M_0(i)$	$V_0 \Leftarrow M_0(i+1)$	Idle
4	$R \Leftarrow V_1 - V_0 * M_1(i+1)$	$R \Leftarrow V_1 - V_0 * M_1(i+2)$	$R \Leftarrow V_1 - V_0 * M_1(i)$	Idle
5	$i = i + 3$; $V_0 \Leftarrow M_0(i+1)$	$i = i + 3$; $R \Leftarrow L$; $V_0 \Leftarrow M_0(i+2)$	$i = i + 3$; $R \Leftarrow L$; $V_0 \Leftarrow M_0(i)$	$V_0 \Leftarrow L * M_2(i)$
6	$V_1 \Leftarrow V_0 * M_1(i+2)$	$V_1 \Leftarrow V_0 * M_1(i)$	$R \Leftarrow L$; $V_1 \Leftarrow V_0 * M_1(i+1)$	$V_1 \Leftarrow L * M_2(i+1) + V_0$
7	$V_0 \Leftarrow M_0(i+2)$	$V_0 \Leftarrow M_0(i)$	$V_0 \Leftarrow M_0(i+1)$	$V_2 \Leftarrow L * M_2(i+2) - V_1$
8	$R \Leftarrow V_1 - V_0 * M_1(i+1)$	$R \Leftarrow V_1 - V_0 * M_1(i+2)$	$R \Leftarrow V_1 - V_0 * M_1(i)$	$M_3(i) \Leftarrow V_2$
9	Goto Cycle 5	Goto Cycle 5	Goto Cycle 5	$i = i + 1$; Goto Cycle 5

Legend

P_i : i^{th} Processing element
M_i : i^{th} Memory element
V_i : i^{th} Register variable
R : Right data path
L : Left data path

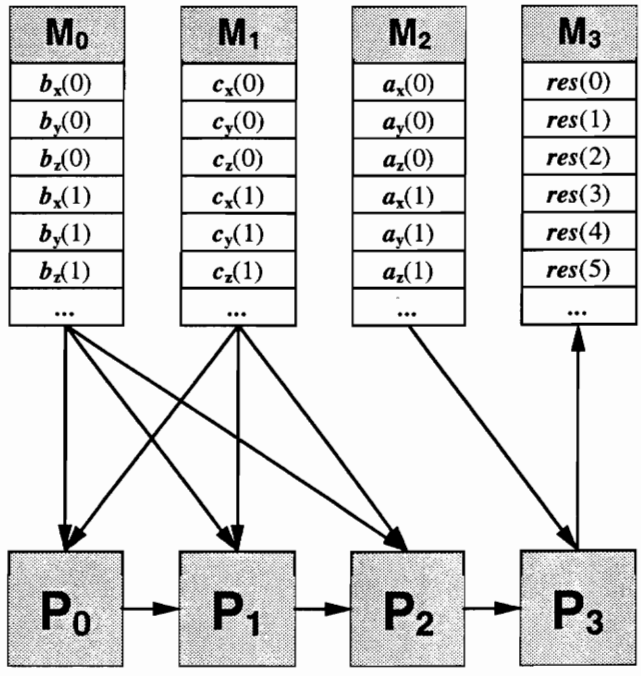


Figure 4.1 Scalar triple product algorithm (top) and memory access pattern (bottom) for a 4-processor shared-memory cluster (P_0-P_3).

This application illustrates several of the benefits gained by having shared-memory on Splash-2. These benefits include increased speedup, reduced computational elements, and reduced complexity. The next algorithm illustrates how shared-memory on Splash-2 can help increase the performance of non-deterministic memory access pattern algorithms.

4.3 Non-Deterministic Memory Access Test Results

Non-deterministic memory accesses are memory reads and writes that occur when the memory address is not known at compile-time, but is calculated during the execution of the program. One example of such an algorithm is the Hough transform. The straight-line Hough transform (SLHT) is an algorithm that is used to detect straight lines in an image [Hough62]. It works by taking pixel coordinates (x,y) in spatial coordinates and transforming them into Hough-space coordinates (θ,d) by using the equation $d = x*\cos\theta + y*\sin\theta$. This transformation from image-space coordinates to Hough-space is performed for each pixel in the image over many values of θ. The result in Hough-space is a collection of sinusoids whose intersections translate into straight lines in image-space. The more intersections that exist at a given point in Hough-space indicates strong evidence of a straight line in the image-space.

The Splash-2 shared-memory Hough transform algorithm is shown in Figure 4.2. Here each processor is responsible for accumulating the Hough-space of different sets of pixel coordinates. The pipeline is structured so that no memory conflicts arise. Since

memory writes have priority over reads, a processor is guaranteed to read valid data at any address location in the Hough-space.

The performance of the Hough transform algorithm is shown as follows:

- Number of processor clock cycles between results: 2.
- Maximum processor clock frequency: ~8.33 Mhz.
- Speedup over single-processor unshared-memory sequential algorithm for 512 pixels over 512-by-512 Hough-space[2]: ~14.

The purpose of this test is to show that values can be shared between processors without data invalidation. However, concurrent writes do follow the "last-in wins" convention. Other techniques, such as semaphores, need to be used if mutual exclusion methods of concurrent write arbitration are needed by the application. Global communication signals are available on the Splash-2 array board which can be used to create these semaphore signals.

[2] This speedup factor is obtained by dividing the number of clock cycles required by the parallel algorithm described in Figure 4.2 by the number of clock cycles required by a single-processor sequential implementation of the same algorithm.

Cycle	P_0	P_1	P_2	P_3
1	$V_0 \Leftarrow x_0 * \cos\theta$	Idle	Idle	Idle
2	$d \Leftarrow V_0 + y_0 * \sin\theta$	$V_0 \Leftarrow x_1 * \cos\theta$	Idle	Idle
3	$V_1 \Leftarrow M_0(d,\theta) + 1$	$d \Leftarrow V_0 + y_1 * \sin\theta$	$V_0 \Leftarrow x_2 * \cos\theta$	Idle
4	$M_1(d,\theta) \Leftarrow V_1$ $\theta = \theta + \delta\theta$	$V_1 \Leftarrow M_0(d,\theta) + 1$	$d \Leftarrow V_0 + y_2 * \sin\theta$	$V_0 \Leftarrow x_3 * \cos\theta$
5	$V_0 \Leftarrow x_0 * \cos\theta$	$M_1(d,\theta) \Leftarrow V_1$ $\theta = \theta + \delta\theta$	$V_1 \Leftarrow M_0(d,\theta) + 1$	$d \Leftarrow V_0 + y_3 * \sin\theta$
6	$d \Leftarrow V_0 + y_0 * \sin\theta$	$V_0 \Leftarrow x_1 * \cos\theta$	$M_1(d,\theta) \Leftarrow V_1$ $\theta = \theta + \delta\theta$	$V_1 \Leftarrow M_0(d,\theta) + 1$
7	$V_1 \Leftarrow M_0(d,\theta) + 1$	$d \Leftarrow V_0 + y_1 * \sin\theta$	$V_0 \Leftarrow x_2 * \cos\theta$	$M_1(d,\theta) \Leftarrow V_1$ $\theta = \theta + \delta\theta$
8	$M_1(d,\theta) \Leftarrow V_1$ $\theta = \theta + \delta\theta$	$V_1 \Leftarrow M_0(d,\theta) + 1$	$d \Leftarrow V_0 + y_2 * \sin\theta$	$V_0 \Leftarrow x_3 * \cos\theta$
9	Goto Cycle 5	Goto Cycle 5	Goto Cycle 5	Goto Cycle 5

Legend

P_i : i^{th} Processing element
M_i : i^{th} Memory element
V_j : j^{th} Register variable
x_i, y_i : i^{th} Image-space coordinate set
d, θ : Hough-space coordinate set

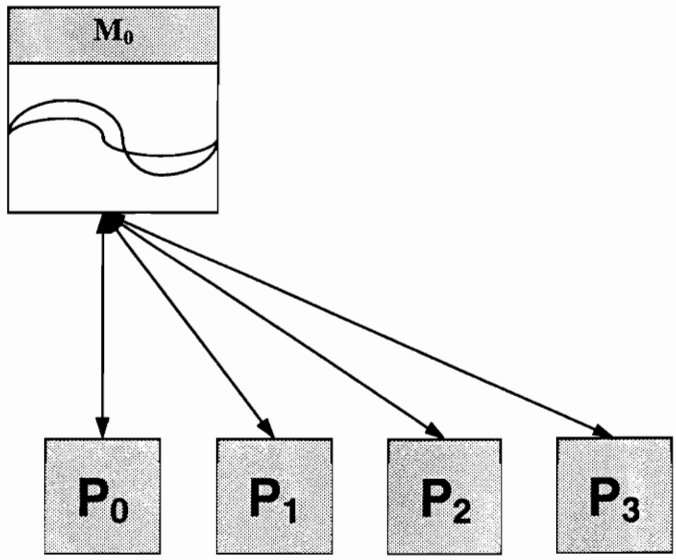


Figure 4.2 Hough transform algorithm (top) and memory access pattern (bottom) for a 4-processor shared-memory cluster (P_0-P_3).

4.4 Global Memory Test Results

The previous two test applications demonstrated how two cluster shared-memory applications perform on Splash-2. This section evaluates a global shared-memory test application. As stated earlier, the global memory performance may not seem to be better than that of a similar cluster memory implementation, but some hidden benefits of a global memory implementation often exist. The application used to demonstrate the capabilities of the global shared-memory is a matrix multiplication problem.

The test application is the transformation of 3-dimensional spatial coordinates from one coordinate frame to another. This operation is used very heavily in robotics and computer vision when the coordinates of an object with respect to one reference frame, i.e. a robot arm, need to be translated to another reference frame, i.e. the robot base [Craig89].

This application of matrix multiplication involves the multiplication of a four-by-four translation matrix **T** by the four-by-one position matrix **v**. The Splash-2 shared-memory access pattern is shown in Figure 4.3. The algorithm is fairly simple, and is described below:

- Step one is as follows for each processor $\mathbf{P}_{i,j}$ for 0 <= i,j <= 3:

    1. Read $T_{i,j}$ from GM_0

    2. Read b_j from GM_1

3. Calculate $d_{i,j}$.

- Step two is as follows for each processor $\mathbf{P}_{i,3}$ for 0 <= i <= 3. All other processors simply pass the data to the processor to its right.

    1. Read $d_{i,2}$ from processor to left.

    2. Calculate $v' = v' + d_{i,2}$. Repeat step 1 two more times then store v'.

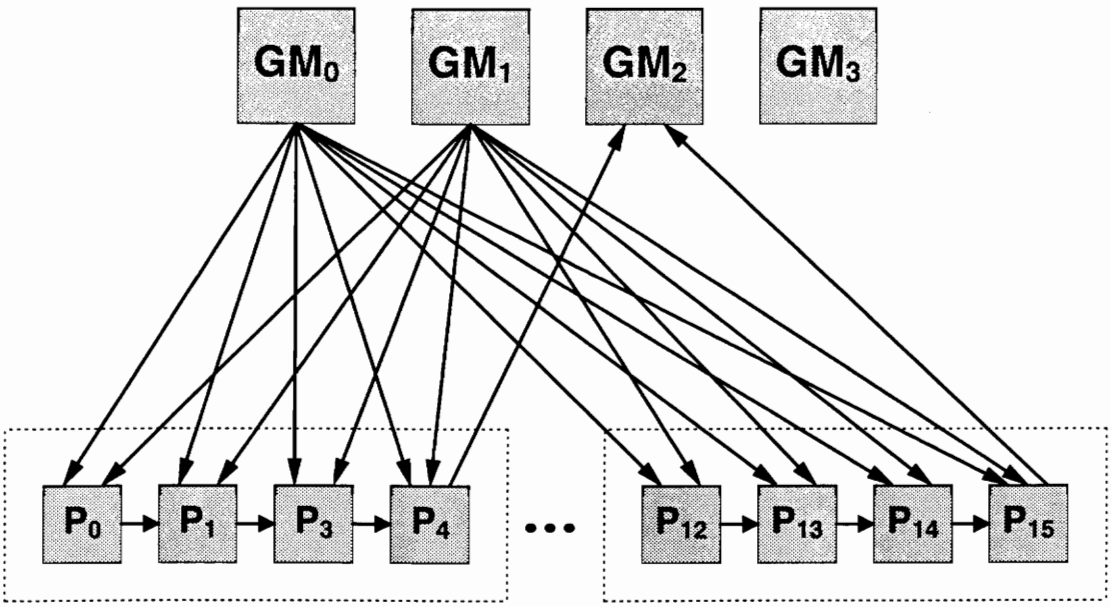


Figure 4.3 Matrix multiplication memory access pattern for a 16-processor shared-memory array board (P_0-P_{15}).

The performance results of this algorithm are as follows (using one PCLK cycle per global memory access):

- Number of processor clock cycles to complete one transformation: 6.

- Average processor clock frequency: ~15 Mhz
- Worst-case processor clock frequency: ~1.5 Mhz
- Speedup over sequential algorithm[3]: 10

It is also useful for allowing all of the processors to operate on the same data in order to increase parallelism. It also reduces the complexity of each processor and the usage of interprocessor communication, since data does not need to be transferred between processors and cluster memory data does not need to be updated and/or invalidated. The overhead involved with maintaining cluster memory coherency far outweighs the cost of using global shared-memory. This application shows that global shared-memory can be used to reduce the complexity of applications and can also reduce the amount of Splash-2 resources needed to implement a given application.

The next section briefly summarizes the performance results discussed in this chapter.

4.5 Summary of Performance Results

The performance results presented in this chapter provide application and compiler designers with the necessary information that can be used to make decisions between using cluster and global shared-memory. Cluster memory accesses are inherently faster

[3] This speedup factor is obtained by dividing the number of clock cycles required by the parallel algorithm described earlier by the number of clock cycles required by a single-processor sequential implementation of the same algorithm.

than their global memory counterparts, however, global memory accesses can help reduce the complexity of some applications.

The Splash-2 shared-memory system provides application programmers and compiler designers with many features that do not exist in the original Splash-2 architecture. These features include:

- Shared-memory communication that can be used to avoid costly usage of interprocessor communication to transfer data.
- Shared-memory helps reduce the number of processors needed to implement certain applications.
- Cluster shared-memory allows writes-after-reads.
- Multi-leveled shared-memory to provide several solutions to any one application's needs.

The Splash-2 shared-memory architecture offers many new features and improvements that can be used by Splash-2 designers. The next chapter introduces some more features that could be added in future shared-memory designs.

Chapter 5

5. Future Work

This chapter describes any future work that can be carried out now that the Splash-2 shared-memory design has been completed and tested.

5.1 Design Modifications

The design described in Chapters 3 and 4 is a first version release of the Splash-2 shared-memory design. Some modifications that could be made to improve the functionality and performance of the first version design are:

- Redesign the processor clock (PCLK) generation so that it is in a separate component. The logic used to generate the PCLK signal is duplicated in

each cluster and global switch controller. Having a separate PCLK generation unit would simplify the switch controllers and improve the performance of global shared-memory accesses.

- Expand the data paths between the clusters and global shared-memory. The performance of the global shared-memory would increase significantly if the global switch was expanded to an eight-by-eight configuration. The global switch and controller would be more complex, but different switch implementations could be simulated to check the feasibility of the new design.

- Optimize the cluster and switch controller logic for the Altera EPM70128 EPLD device. This will ensure proper functionality and timing once the system is constructed.

Other modifications should be made as needed in order to make construction of the Splash-2 shared-memory design more efficient. The next section describes the steps that need to be taken in order to fabricate this design.

5.2 Design Fabrication

A schematic layout of the Splash-2 shared-memory system needs to be designed in order to begin construction of the actual system. After schematic layout is finished, an experimental version of the system should be constructed and tested against the VHDL

simulation. Actual timing information should be back-annotated into the simulation for further verification of the model, since it will be used for application development. Once the experimental version of the design is functional, a printed circuit board layout can be designed. The boards can then be fabricated and fully tested.

5.3 Compiler and Application Development

Once a working system is finished, compiler and application development can continue, using the new Splash-2 shared-memory architecture. The applications described in Chapter 4 can be implemented and incorporated into larger programs. High-level language compilers can be developed for Splash-2 that will take advantage of the new memory system. The new features should enable designers to reach performance levels previously unattainable on the original Splash-2 system. The final chapter summarizes these features and concludes this thesis.

Chapter 6

6. Conclusion

The Splash-2 shared-memory architecture presented in this document is a solution to several of the challenges faced by compiler and application designers. It provides several new features on Splash-2 that enable designers to take better advantage of the inherent parallelism that exists in Splash-2, while keeping the generalized nature of the original Splash-2 system intact.

A summary of the features described in Chapters 3 and 4 is as follows:

- The address space of a processing element has increased by a factor of eight. Each PE now has 4 Mbytes of memory that it can access directly.

- The maximum bandwidth of each network switch is 400 Mbytes/second, which gives a total network bandwidth of 2 Gbytes/second. This corresponds to a maximum available memory bandwidth of 125 Mbytes/second per PE. This more than twice the bandwidth of the existing Splash-2 processing element.

- The new memory system supports shared-memory hierarchy, which helps to increase the performance of many parallel algorithms. In addition, it supports all existing applications designed for Splash-2.

- Shared-memory communication that can be used to avoid costly usage of interprocessor communication to transfer data.

- Shared-memory helps reduce the number of processors needed to implement certain applications.

- Cluster shared-memory allows writes-after-reads. This eliminates the extra clock cycle previously needed to switch memory access direction.

The Splash-2 shared-memory architecture is only a part of the solution to improve the efficiency of the custom computing machine's programming environment. However, as the work presented in this thesis demonstrates, it is definitely a step in the right direction.

Bibliography

[Agarwala78] A.K. Agarwala and T.G. Rauscher, "Dynamic Problem-Oriented Redefinition of Computer Architecture Via Microprogramming,", IEEE Transactions on Computers, Vol. C-27, No. 11, Nov. 1978, pp 1006-1014.

[AMD94] *Am29C983A Data Sheet.* Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA, 94088, 1994.

[Altera93] *MAX 7000 Data Book.* Altera Corp., San Jose, CA, 95134, 1993.

[Amerson95] R. Amerson, et al, "Teramac - Configurable Custom Computing," In *Preliminary Proc. IEEE Symposium on FPGAs for Custom Compting Machines*, April 1994.

[Arnold92] J. M. Arnold, D. A. Buell, and E. G. Davis. "Splash 2," In *Proc. 4th Annual ACM Symp. on Parallelism Algorithms and Architectures*, 1992, pp 316-322.

[Athanas93] P. M. Athanas, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *IEEE Computer*, 26(3), March 1993, pp 11-18.

[Benes62] V. E. Benes, "On Rearrangeable Three-Stage Connecting Networks," *The Bell System Technical Journal*, Vol. XLI, No. 5, Sept. 1962, pp 1481-1492.

[Bertin93] P. Bertin, D. Roncin, and J. Vuillemin. "Programmable Active Memories: A Performance Assessment," *Symposium on Integrated Systems*, MIT Press, 1993.

[Box94] B. Box, "Field Programmable Gate Array Based Reconfigurable Preprocessor," *IEEE Workshop on FPGAs for Custom Compting Machines*, April 1994,pp 40-48.

[Box95] B. Box, "Common Processor Element Packaging," In *Preliminary Proc. IEEE Symposium on FPGAs for Custom Compting Machines*, April 1994.

[Buell93] D. A. Buell and K. Pocek, editors, "Foreward", *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993.

[Buell95] D. A. Buell, J.M. Arnold, and W. J. Kleinfelder, editors. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1995.

[Burns94] D. Burns. *Splash2 Array Board Hardware Engineering Document*. Supercomputing Research Center, Feb. 1994.

[Chan93] P. K. Chan and M. D. F. Schlag, "Architectural Tradeoffs in Field-Programmable-Device-Based Computing Systems," *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993, pp 152-161.

[Clos53] C. Clos, "A Study of Non-Blocking Switching Networks," *The Bell System Technical Journal*. Volume XXXII, March, 1953, pp 406-424.

[Cox91] C. E. Cox and W. E. Blanz, "Ganglion - A Fast Hardware Implementation of a Connectionist Classifier," Proceedings of the 1991 IEEE Custom Integrated Circuits Conference, 1991, pp 6.5.1-6.5.4.

[Craig89] J. Craig. *Introduction to Robotics: Mechanics and Control, Second Edition.* Addison-Wesley Publishing Co., Inc., New York, 1989.

[Davis85] N. J. Davis IV, et al, "Fault Location Techniques for Distributed Control Interconnection Networks," *IEEE Transactions on Computers*, Vol C-34, No. 10, Oct. 1985, pp 902-910.

[Edwards85] C. H. Edwards, Jr. and D. E. Penney. *Calculus and Analytic Geometry*, Prentice Hall Inc., Englewood Cliffs, NJ, 07632, 1985.

[Estrin60] G. Estrin, "Organization of Computer Systems: The Fixed-Plus Variable Computer," *Proceedings of the Western Joint Computer Conference*, American Institute of Electrical Engineers, New York, 1960, pp 33-40.

[Galloway95] Galloway, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs", In *Prelim. Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, April 1995.

[Giga94] *G-800 Techinical Summary*, Giga Operations Corp. 2374 Eunice St., Berkeley, CA 94708, 1994.

[Gohkale93] M. Gohkale and J. Schlesinger, *dBC Reference Manual.* Technical Report SRC-TR-92-068 Rev. 2, CCS (SRC), 1993.

[Hennessey90] J. L. Hennessey and D.A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc. San Mateo, CA, 1990.

[Hough62] P.V.C. Hough. *Methods and means for recognizing complex patterns*, U.S. Patent 3.069.654, 1962.

[Hwang93] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., New York, NY, 1993.

[ICube94] *The FPID Family Data Sheet*. I-Cube2328-C Walsh Ave., Santa Clara, CA, 95051, Sept. 1994.

[IDTLogic92] *High Performance Logic Data Book*. Integrated Device Technology, Inc. 2975 Stender Way, Santa Clara, CA, 95054, 1992.

[IDTMemory93] *High Performance Static RAMs Data Book*. Integrated Device Technology, Inc. 2975 Stender Way, Santa Clara, CA, 95054, 1993.

[Kleinfelder94] W. Kleinfelder and D. Burns, *Splash-2 Interface Board Hardware Document*, SRC Technical Report, April 1994.

[Konicek91] J. Konicek, et al, "The Organization of the Cedar System," Proc. of the 1991 International Conference on Parallel Processing, 1991, pp 49-56.

[Kuck93] D. Kuck, et al, "The Cedar System and an Initial Performance Study," In Proc. 20th Annual International Symposium on Computer Architecture, IEEE Computer Society Press, 1993, pp 213-223.

[Lawrie75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, Vol. C-24, No. 12, Dec. 1975, pp 1145-1155.

[Nassimi81] D. Nassimi and S. Sahni, "A Self-Routing Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers*, Vol. C-30, No. 5, May 1981, pp 332-340.

[Patel81] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers*, Vol. C-30, No. 10, Oct. 1981, pp 771-780.

[Synopsys94] *Design Compiler User's Manual.* Synopsys, Inc., 700 East Middlefield Rd., Mountain View, CA, 94043, 1994.

[TI93] *Advanced CMOS Logic Data Book.* Texas Instruments, Austin, TX, 78759,1993.

[Touati93] P. Bertin and H. Touati. "PAM programming Environments: Practive and Experience," *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp 133-139.

[Wu80] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers*, Volume C-29, No. 8, Aug. 1980, pp 694-702.

[Wulf72] W. A. Wulf and C. G. Bell, "C.mmp - A Multi-Miniprocessor," *Proc. Fall Joint Computer Conference*, 1972, pp 765-777.

[VHDL92] *VHSIC Hardware Description Language Reference Manual.* IEEE Standard 1164, IEEE Computer Press, 1992.

[Xilinx94] *The Programmable Gate Array Data Book*, Xilinx, 2100 Logic Drive, San Jose, CA, 95124, 1994.

Appendix A

This appendix contains the new entity declaration of the Xilinx_Processing_Part component from the Splash-2 VHDL simulation model.

```
-------------------------------------------------------------------------------
--  Splash-2 Xilinx_Processing_Part Entity Declaration
-------------------------------------------------------------------------------
entity Xilinx_Processing_Part is
  Generic(
    BD_ID          : Integer := 0;                    --  Splash Board ID
    PE_ID          : Integer := 0;                    --  Processing Element ID
);
  Port (
    XP_Left        : inout DataPath;                  --  Left Data Bus
    XP_Right       : inout DataPath;                  --  Right Data Bus
    XP_Xbar        : inout DataPath;                  --  Crossbar Data Bus
    XP_Xbar_EN_L   : out Bit_Vector(4 downto 0);      --  Crossbar Enable (low-true)
    XP_Clk         : in Bit;                          --  Splash System Clock
    XP_Int         : out Bit;                         --  Interrupt Signal
    XP_GMem_REQ    : inout Rbit3;                     --  Global Memory Request
    XP_GMem_ACK    : inout RBit3;                     --  Global Memory Acknowledge
    XP_Mem_SEL     : inout RBit3_Vector(1 downto 0);  --  Memory Chip Select
    XP_Mem_A       : inout MemAddr;                   --  Memory Address Bus
    XP_Mem_D       : inout MemData;                   --  Memory Data Bus
    XP_Mem_RD_L    : inout Rbit3;                     --  Memory Read Signal (low-true)
    XP_Mem_WR_L    : inout RBit3;                     --  Memory Write Signal (low-true)
    XP_Mem_Disable : inout RBit3;                     --  Memory Disable Signal
    XP_Broadcast   : in Bit;                          --  Broadcast Signal
    XP_Reset       : in Bit;                          --  Reset Signal
    XP_HS0         : inout Rbit3;                     --  Handshake Signal Zero
    XP_HS1         : in Bit;                          --  Handshake Signal One
    XP_GOR_Result  : inout Rbit3;                     --  Global OR Result Signal
    XP_GOR_Valid   : inout Rbit3                      --  Global OR Valid Signal
);
end Xilinx_Processing_Part;
```

Appendix B

This appendix contains the behavioral VHDL code that describes the cluster switch controller component.

```
library IEEE,GMS2;
USE IEEE.std_logic_1164.all;
use GMS2.TYPES.all;

-------------------------------------------------------------------------------
--
--  CLUSTER_SWITCH_CTRL:
--
--      A memory controller switch control unit.
--
--      I/O: 74 used (96 total for the EPM70128)
--
-------------------------------------------------------------------------------
ENTITY CLUSTER_SWITCH_CTRL IS
  port (
    MCLK                : in Bit;                --
    TP                  : in RBit3;              --
    PCLK                : out Bit;               --
    SYNC_1              : in Bit;                --
    SYNC_2              : in Bit;                --
    SYNC_3              : in Bit;                --
    PE_REQ_A            : in RBit3 := 'Z';       --
    PE_REQ_B            : in RBit3 := 'Z';       --
    PE_REQ_C            : in RBit3 := 'Z';       --
    PE_REQ_D            : in RBit3 := 'Z';       --
    PE_SEL_A            : in RBit3_Vector(1 downto 0) := (others => 'Z');--
    PE_SEL_B            : in RBit3_Vector(1 downto 0) := (others => 'Z');--
    PE_SEL_C            : in RBit3_Vector(1 downto 0) := (others => 'Z');--
    PE_SEL_D            : in RBit3_Vector(1 downto 0) := (others => 'Z');--  15
    PE_RD_A_L           : in RBit3 := 'Z';       --
    PE_RD_B_L           : in RBit3 := 'Z';       --
    PE_RD_C_L           : in RBit3 := 'Z';       --
    PE_RD_D_L           : in RBit3 := 'Z';       --
    PE_WR_A_L           : in RBit3 := 'Z';       --
    PE_WR_B_L           : in RBit3 := 'Z';       --
    PE_WR_C_L           : in RBit3 := 'Z';       --
    PE_WR_D_L           : in RBit3 := 'Z';       --
    MEM_CE_A_L          : out RBit3 := 'Z';      --
    MEM_CE_B_L          : out RBit3 := 'Z';      --
    MEM_CE_C_L          : out RBit3 := 'Z';      --
    MEM_CE_D_L          : out RBit3 := 'Z';      --
    MEM_WE_A_L          : out RBit3 := 'Z';      --
    MEM_WE_B_L          : out RBit3 := 'Z';      --
    MEM_WE_C_L          : out RBit3 := 'Z';      --  30
    MEM_WE_D_L          : out RBit3 := 'Z';      --
    MEM_OE_A_L          : out RBit3 := 'Z';      --
    MEM_OE_B_L          : out RBit3 := 'Z';      --
    MEM_OE_C_L          : out RBit3 := 'Z';      --
    MEM_OE_D_L          : out RBit3 := 'Z';      --
    PE_ADDR_SEL_A       : out RBit3 := 'Z';      --
    PE_ADDR_SEL_B       : out RBit3 := 'Z';      --
    PE_ADDR_SEL_C       : out RBit3 := 'Z';      --
    PE_ADDR_SEL_D       : out RBit3 := 'Z';      --
    MEM_ADDR_SEL_A      : out RBit3 := 'Z';      --  40
    MEM_ADDR_SEL_B      : out RBit3 := 'Z';      --
    MEM_ADDR_SEL_C      : out RBit3 := 'Z';      --
    MEM_ADDR_SEL_D      : out RBit3 := 'Z';      --
    PE_DATA_RD_LE_A_L   : out RBit3 := 'Z';      --
    PE_DATA_RD_LE_B_L   : out RBit3 := 'Z';      --
    PE_DATA_RD_LE_C_L   : out RBit3 := 'Z';      --
    PE_DATA_RD_LE_D_L   : out RBit3 := 'Z';      --
```

```
    PE_DATA_RD_SEL_A     : out RBit3 := 'Z';      --
    PE_DATA_RD_SEL_B     : out RBit3 := 'Z';      --
    PE_DATA_RD_SEL_C     : out RBit3 := 'Z';      --  50
    PE_DATA_RD_SEL_D     : out RBit3 := 'Z';      --
    PE_DATA_WR_SEL_A     : out RBit3 := 'Z';      --
    PE_DATA_WR_SEL_B     : out RBit3 := 'Z';      --
    PE_DATA_WR_SEL_C     : out RBit3 := 'Z';      --
    PE_DATA_WR_SEL_D     : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_A    : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_B    : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_C    : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_D    : out RBit3 := 'Z';      --
    MEM_DATA_WR_SEL_A    : out RBit3 := 'Z';      --  60
    MEM_DATA_WR_SEL_B    : out RBit3 := 'Z';      --
    MEM_DATA_WR_SEL_C    : out RBit3 := 'Z';      --
    MEM_DATA_WR_SEL_D    : out RBit3 := 'Z';      --
    PE_DATA_RD_OE_A_L    : out RBit3 := 'Z';      --
    PE_DATA_RD_OE_B_L    : out RBit3 := 'Z';      --
    PE_DATA_RD_OE_C_L    : out RBit3 := 'Z';      --
    PE_DATA_RD_OE_D_L    : out RBit3 := 'Z';      --
    PE_DATA_WR_OE_A_L    : out RBit3 := 'Z';      --
    PE_DATA_WR_OE_B_L    : out RBit3 := 'Z';      --
    PE_DATA_WR_OE_C_L    : out RBit3 := 'Z';      --  70
    PE_DATA_WR_OE_D_L    : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_A_L   : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_B_L   : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_C_L   : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_D_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_A_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_B_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_C_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_D_L   : out RBit3 := 'Z'       --  79
  );
END CLUSTER_SWITCH_CTRL;

ARCHITECTURE Behavioral OF CLUSTER_SWITCH_CTRL IS

  CONSTANT tPD : TIME := 10 ns;          --   I/O propagation delay
  CONSTANT tSEXP : TIME := 5 ns;         --   Shared expander delay
  TYPE STATES IS (RESET,RESET_DELAY,SYNC_UP,DECODE,WRITE,READ,READ_DELAY);
  SIGNAL STATE : STATES;
  SIGNAL iPCLK : Bit := '0';
  SIGNAL A_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL B_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL C_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL D_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');

  SIGNAL B_SAME_AS_A : RBit3 := 'Z';
  SIGNAL C_SAME_AS_A : RBit3 := 'Z';
  SIGNAL C_SAME_AS_B : RBit3 := 'Z';
  SIGNAL D_SAME_AS_A : RBit3 := 'Z';
  SIGNAL D_SAME_AS_B : RBit3 := 'Z';
  SIGNAL D_SAME_AS_C : RBit3 := 'Z';

  SIGNAL A_WR : RBit3 := 'Z';
  SIGNAL B_WR : RBit3 := 'Z';
  SIGNAL C_WR : RBit3 := 'Z';
  SIGNAL D_WR : RBit3 := 'Z';

  SIGNAL WR_DONE : RBit3 := '0';
  SIGNAL A_WR_DONE : RBit3 := '0';
  SIGNAL B_WR_DONE : RBit3 := '0';
  SIGNAL C_WR_DONE : RBit3 := '0';
  SIGNAL D_WR_DONE : RBit3 := '0';

  SIGNAL A_RD : RBit3 := 'Z';
  SIGNAL B_RD : RBit3 := 'Z';
  SIGNAL C_RD : RBit3 := 'Z';
  SIGNAL D_RD : RBit3 := 'Z';

  SIGNAL RD_DONE : RBit3 := '0';
  SIGNAL A_RD_DONE : RBit3 := '0';
  SIGNAL B_RD_DONE : RBit3 := '0';
  SIGNAL C_RD_DONE : RBit3 := '0';
  SIGNAL D_RD_DONE : RBit3 := '0';

  SIGNAL B_RD_MUST_DELAY : RBit3 := 'Z';
```

```
  SIGNAL D_RD_MUST_DELAY : RBit3 := 'Z';
  SIGNAL B_RD_DELAYED : RBit3 := 'Z';
  SIGNAL D_RD_DELAYED : RBit3 := 'Z';

  SIGNAL iTP : RBit3 := '0';
  SIGNAL iA_WR_DONE : RBit3 := '0';
  SIGNAL iB_WR_DONE : RBit3 := '0';
  SIGNAL iC_WR_DONE : RBit3 := '0';
  SIGNAL iD_WR_DONE : RBit3 := '0';
  SIGNAL iA_RD_DONE : RBit3 := '0';
  SIGNAL iB_RD_DONE : RBit3 := '0';
  SIGNAL iC_RD_DONE : RBit3 := '0';
  SIGNAL iD_RD_DONE : RBit3 := '0';
  SIGNAL iMEM_CE_A_L : RBit3 := '1';
  SIGNAL iMEM_CE_B_L : RBit3 := '1';
  SIGNAL iMEM_CE_C_L : RBit3 := '1';
  SIGNAL iMEM_CE_D_L : RBit3 := '1';
  SIGNAL iMEM_WE_A_L : RBit3 := '1';
  SIGNAL iMEM_WE_B_L : RBit3 := '1';
  SIGNAL iMEM_WE_C_L : RBit3 := '1';
  SIGNAL iMEM_WE_D_L : RBit3 := '1';
  SIGNAL iMEM_OE_A_L : RBit3 := '1';
  SIGNAL iMEM_OE_B_L : RBit3 := '1';
  SIGNAL iMEM_OE_C_L : RBit3 := '1';
  SIGNAL iMEM_OE_D_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_LE_A_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_LE_B_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_LE_C_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_LE_D_L : RBit3 := '1';
  SIGNAL iPE_ADDR_SEL_A : RBit3 := '0';
  SIGNAL iPE_ADDR_SEL_B : RBit3 := '0';
  SIGNAL iPE_ADDR_SEL_C : RBit3 := '0';
  SIGNAL iPE_ADDR_SEL_D : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_A : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_B : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_C : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_D : RBit3 := '0';
  SIGNAL iPE_DATA_WR_SEL_A : RBit3 := '0';
  SIGNAL iPE_DATA_WR_SEL_B : RBit3 := '0';
  SIGNAL iPE_DATA_WR_SEL_C : RBit3 := '0';
  SIGNAL iPE_DATA_WR_SEL_D : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_A : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_B : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_C : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_D : RBit3 := '0';
  SIGNAL iPE_DATA_RD_SEL_A : RBit3 := '0';
  SIGNAL iPE_DATA_RD_SEL_B : RBit3 := '0';
  SIGNAL iPE_DATA_RD_SEL_C : RBit3 := '0';
  SIGNAL iPE_DATA_RD_SEL_D : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_A : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_B : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_C : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_D : RBit3 := '0';
  SIGNAL iPE_DATA_RD_OE_A_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_OE_B_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_OE_C_L : RBit3 := '1';
  SIGNAL iPE_DATA_RD_OE_D_L : RBit3 := '1';
  SIGNAL iPE_DATA_WR_OE_A_L : RBit3 := '1';
  SIGNAL iPE_DATA_WR_OE_B_L : RBit3 := '1';
  SIGNAL iPE_DATA_WR_OE_C_L : RBit3 := '1';
  SIGNAL iPE_DATA_WR_OE_D_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_A_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_B_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_C_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_D_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_A_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_B_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_C_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_D_L : RBit3 := '1';

BEGIN  --  Behavioral

  ------------------------------------------------------------------------------
  --  Assign PCLK
  ------------------------------------------------------------------------------
  PCLK <= iPCLK after tPD;
```

```
  ------------------------------------------------------------------------------
  --  Assign latch enable signals here.
  ------------------------------------------------------------------------------
  MEM_ADDR_LE_L        <= '1';
  MEM_DATA_WR_LE_A_L   <= '1';
  MEM_DATA_WR_LE_B_L   <= '1';
  MEM_DATA_WR_LE_C_L   <= '1';
  MEM_DATA_WR_LE_D_L   <= '1';

  ------------------------------------------------------------------------------
  --  Assign memory control signals here.
  ------------------------------------------------------------------------------
  MEM_CE_A_L <= iMEM_CE_A_L after tPD;
  MEM_CE_B_L <= iMEM_CE_B_L after tPD;
  MEM_CE_C_L <= iMEM_CE_C_L after tPD;
  MEM_CE_D_L <= iMEM_CE_D_L after tPD;
  MEM_WE_A_L <= (TP OR iMEM_WE_A_L) after tPD;
  MEM_WE_B_L <= (TP OR iMEM_WE_B_L) after tPD;
  MEM_WE_C_L <= (TP OR iMEM_WE_C_L) after tPD;
  MEM_WE_D_L <= (TP OR iMEM_WE_D_L) after tPD;
  MEM_OE_A_L <= iMEM_OE_A_L after tPD;
  MEM_OE_B_L <= iMEM_OE_B_L after tPD;
  MEM_OE_C_L <= iMEM_OE_C_L after tPD;
  MEM_OE_D_L <= iMEM_OE_D_L after tPD;

--  iTP <= TP after tPD;
--  MEM_CE_A_L <= iMEM_CE_A_L after 13 ns;
--  MEM_CE_B_L <= iMEM_CE_B_L after 13 ns;
--  MEM_CE_C_L <= iMEM_CE_C_L after 13 ns;
--  MEM_CE_D_L <= iMEM_CE_D_L after 13 ns;
--  MEM_WE_A_L <= (NOT (iTP AND (NOT iMEM_WE_A_L))) after 10 ns;
--  MEM_WE_B_L <= (NOT (iTP AND (NOT iMEM_WE_B_L))) after 10 ns;
--  MEM_WE_C_L <= (NOT (iTP AND (NOT iMEM_WE_C_L))) after 10 ns;
--  MEM_WE_D_L <= (NOT (iTP AND (NOT iMEM_WE_D_L))) after 10 ns;
--  MEM_OE_A_L <= iMEM_OE_A_L after 13 ns;
--  MEM_OE_B_L <= iMEM_OE_B_L after 13 ns;
--  MEM_OE_C_L <= iMEM_OE_C_L after 13 ns;
--  MEM_OE_D_L <= iMEM_OE_D_L after 13 ns;

  ------------------------------------------------------------------------------
  --  Assign switch control signals here
  ------------------------------------------------------------------------------
  PE_ADDR_SEL_A <= iPE_ADDR_SEL_A after tPD;
  PE_ADDR_SEL_B <= iPE_ADDR_SEL_B after tPD;
  PE_ADDR_SEL_C <= iPE_ADDR_SEL_C after tPD;
  PE_ADDR_SEL_D <= iPE_ADDR_SEL_D after tPD;
  MEM_ADDR_SEL_A <= iMEM_ADDR_SEL_A after tPD;
  MEM_ADDR_SEL_B <= iMEM_ADDR_SEL_B after tPD;
  MEM_ADDR_SEL_C <= iMEM_ADDR_SEL_C after tPD;
  MEM_ADDR_SEL_D <= iMEM_ADDR_SEL_D after tPD;
  PE_DATA_WR_SEL_A <= iPE_DATA_WR_SEL_A after tPD;
  PE_DATA_WR_SEL_B <= iPE_DATA_WR_SEL_B after tPD;
  PE_DATA_WR_SEL_C <= iPE_DATA_WR_SEL_C after tPD;
  PE_DATA_WR_SEL_D <= iPE_DATA_WR_SEL_D after tPD;
  MEM_DATA_WR_SEL_A <= iMEM_DATA_WR_SEL_A after tPD;
  MEM_DATA_WR_SEL_B <= iMEM_DATA_WR_SEL_B after tPD;
  MEM_DATA_WR_SEL_C <= iMEM_DATA_WR_SEL_C after tPD;
  MEM_DATA_WR_SEL_D <= iMEM_DATA_WR_SEL_D after tPD;
  PE_DATA_RD_SEL_A <= iPE_DATA_RD_SEL_A after tPD;
  PE_DATA_RD_SEL_B <= iPE_DATA_RD_SEL_B after tPD;
  PE_DATA_RD_SEL_C <= iPE_DATA_RD_SEL_C after tPD;
  PE_DATA_RD_SEL_D <= iPE_DATA_RD_SEL_D after tPD;
  MEM_DATA_RD_SEL_A <= iMEM_DATA_RD_SEL_A after tPD;
  MEM_DATA_RD_SEL_B <= iMEM_DATA_RD_SEL_B after tPD;
  MEM_DATA_RD_SEL_C <= iMEM_DATA_RD_SEL_C after tPD;
  MEM_DATA_RD_SEL_D <= iMEM_DATA_RD_SEL_D after tPD;
  PE_DATA_RD_OE_A_L <= NOT A_RD_DONE after tPD;
  PE_DATA_RD_OE_B_L <= NOT B_RD_DONE after tPD;
  PE_DATA_RD_OE_C_L <= NOT C_RD_DONE after tPD;
  PE_DATA_RD_OE_D_L <= NOT D_RD_DONE after tPD;
--  PE_DATA_RD_OE_A_L <= iPE_DATA_RD_OE_A_L after tPD;
--  PE_DATA_RD_OE_B_L <= iPE_DATA_RD_OE_B_L after tPD;
--  PE_DATA_RD_OE_C_L <= iPE_DATA_RD_OE_C_L after tPD;
--  PE_DATA_RD_OE_D_L <= iPE_DATA_RD_OE_D_L after tPD;
  PE_DATA_WR_OE_A_L <= iPE_DATA_WR_OE_A_L after tPD;
  PE_DATA_WR_OE_B_L <= iPE_DATA_WR_OE_B_L after tPD;
  PE_DATA_WR_OE_C_L <= iPE_DATA_WR_OE_C_L after tPD;
```

```
  PE_DATA_WR_OE_D_L <= iPE_DATA_WR_OE_D_L after tPD;
--  MEM_DATA_RD_OE_A_L <= iMEM_DATA_RD_OE_A_L after tPD;
--  MEM_DATA_RD_OE_B_L <= iMEM_DATA_RD_OE_B_L after tPD;
--  MEM_DATA_RD_OE_C_L <= iMEM_DATA_RD_OE_C_L after tPD;
--  MEM_DATA_RD_OE_D_L <= iMEM_DATA_RD_OE_D_L after tPD;
  MEM_DATA_WR_OE_A_L <= iMEM_DATA_WR_OE_A_L after tPD;
  MEM_DATA_WR_OE_B_L <= iMEM_DATA_WR_OE_B_L after tPD;
  MEM_DATA_WR_OE_C_L <= iMEM_DATA_WR_OE_C_L after tPD;
  MEM_DATA_WR_OE_D_L <= iMEM_DATA_WR_OE_D_L after tPD;

  ---------------------------------------------------------------------------
  --  Decode any switch destination contentions in this section
  ---------------------------------------------------------------------------
  A_DEST <= PE_REQ_A & PE_SEL_A;
  B_DEST <= PE_REQ_B & PE_SEL_B;
  C_DEST <= PE_REQ_C & PE_SEL_C;
  D_DEST <= PE_REQ_D & PE_SEL_D;

  --  B is considered the same as A if they both need to go to the same second
  --  stage switch (i.e. top two bits are the same) due to the switch structure
  B_SAME_AS_A <= '1' WHEN (B_DEST(2 downto 1) = A_DEST(2 downto 1)) ELSE '0';
  C_SAME_AS_A <= '1' WHEN (C_DEST = A_DEST) ELSE '0';
  C_SAME_AS_B <= '1' WHEN (C_DEST = B_DEST) ELSE '0';
  D_SAME_AS_A <= '1' WHEN (D_DEST = A_DEST) ELSE '0';
  D_SAME_AS_B <= '1' WHEN (D_DEST = B_DEST) ELSE '0';
  --  D is considered the same as C if they both need to go to the same second
  --  stage switch (i.e. top two bits are the same) due to the switch structure
  D_SAME_AS_C <= '1' WHEN (D_DEST(2 downto 1) = C_DEST(2 downto 1)) ELSE '0';

  ---------------------------------------------------------------------------
  --  Decode the internal write strobes in this section
  ---------------------------------------------------------------------------
  A_WR <= (NOT PE_WR_A_L) AND (NOT A_WR_DONE) AND (NOT PE_REQ_A);
  B_WR <= (NOT (B_SAME_AS_A AND (NOT PE_WR_A_L) AND (NOT A_WR_DONE)))
          AND (NOT PE_WR_B_L) AND (NOT B_WR_DONE) AND (NOT PE_REQ_B);
  C_WR <= (NOT ((C_SAME_AS_A AND (NOT PE_WR_A_L) AND (NOT A_WR_DONE))
          OR   (C_SAME_AS_B AND (NOT PE_WR_B_L) AND (NOT B_WR_DONE))))
          AND (NOT PE_WR_C_L) AND (NOT C_WR_DONE) AND (NOT PE_REQ_C);
  D_WR <= (NOT ((D_SAME_AS_A AND (NOT PE_WR_A_L) AND (NOT A_WR_DONE))
          OR   (D_SAME_AS_B AND (NOT PE_WR_B_L) AND (NOT B_WR_DONE))
          OR   (D_SAME_AS_C AND (NOT PE_WR_C_L) AND (NOT C_WR_DONE))))
          AND (NOT PE_WR_D_L) AND (NOT D_WR_DONE) AND (NOT PE_REQ_D);
  WR_DONE <= (A_WR_DONE OR PE_WR_A_L OR PE_REQ_A)
             AND (B_WR_DONE OR PE_WR_B_L OR PE_REQ_B)
             AND (C_WR_DONE OR PE_WR_C_L OR PE_REQ_C)
             AND (D_WR_DONE OR PE_WR_D_L OR PE_REQ_D);

  ---------------------------------------------------------------------------
  --  Decode the internal read strobes here
  ---------------------------------------------------------------------------
  A_RD <= (NOT PE_RD_A_L) AND (NOT A_RD_DONE) AND (NOT PE_REQ_A);
  B_RD <= (NOT (B_SAME_AS_A AND (NOT PE_RD_A_L) AND (NOT A_RD_DONE)))
          AND (NOT PE_RD_B_L) AND (NOT B_RD_DONE) AND (NOT PE_REQ_B);
  C_RD <= (NOT ((C_SAME_AS_A AND (NOT PE_RD_A_L) AND (NOT A_RD_DONE))
          OR   (C_SAME_AS_B AND (NOT PE_RD_B_L) AND (NOT B_RD_DONE))))
          AND (NOT PE_RD_C_L) AND (NOT C_RD_DONE) AND (NOT PE_REQ_C);
  D_RD <= (NOT ((D_SAME_AS_A AND (NOT PE_RD_A_L) AND (NOT A_RD_DONE))
          OR   (D_SAME_AS_B AND (NOT PE_RD_B_L) AND (NOT B_RD_DONE))
          OR   (D_SAME_AS_C AND (NOT PE_RD_C_L) AND (NOT C_RD_DONE))))
          AND (NOT PE_RD_D_L) AND (NOT D_RD_DONE) AND (NOT PE_REQ_D);
  RD_DONE <= (A_RD_DONE OR PE_RD_A_L OR PE_REQ_A)
             AND (B_RD_DONE OR PE_RD_B_L OR PE_REQ_B)
             AND (C_RD_DONE OR PE_RD_C_L OR PE_REQ_C)
             AND (D_RD_DONE OR PE_RD_D_L OR PE_REQ_D);

  B_RD_MUST_DELAY <= B_RD AND A_RD_DONE AND (PE_SEL_A(0) XOR PE_SEL_B(0))
                     AND (NOT B_RD_DELAYED);
  D_RD_MUST_DELAY <= D_RD AND C_RD_DONE AND (PE_SEL_C(0) XOR PE_SEL_D(0))
                     AND (NOT D_RD_DELAYED) AND (NOT B_RD_DELAYED);

  ---------------------------------------------------------------------------
  ---------------------------------------------------------------------------
  --  State Machine Process
  --
  --  Things To Do:
  --    - shave a clock cycle off after the last write/read is completed.
  --    - add control signals for the switch: might have to have a timing
```

```
--      pulse to generate memory control signals.
--            - figure out the timing for the memory CE, WE, and OE signals
--            - figure out the timing for the address latch enable.
--            - do the data latch enables
--            - do the data output enables
--
--    - once the write/read decode sections are working, move them back into
--      the state machine case statement.  once back in the individual state
--      cases, they can be optimized (i.e. removing the A_RD and A_WR
--      if-clauses from the READ and WRITE states, respectively) Also move
--      the DONE line resets back into the DECODE state.
--
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
STATE_MACHINE : PROCESS

  VARIABLE vA_WR_DONE : RBit3 := '0';
  VARIABLE vB_WR_DONE : RBit3 := '0';
  VARIABLE vC_WR_DONE : RBit3 := '0';
  VARIABLE vD_WR_DONE : RBit3 := '0';
  VARIABLE vA_RD_DONE : RBit3 := '0';
  VARIABLE vB_RD_DONE : RBit3 := '0';
  VARIABLE vC_RD_DONE : RBit3 := '0';
  VARIABLE vD_RD_DONE : RBit3 := '0';
  VARIABLE vMEM_CE_A_L : RBit3 := '1';
  VARIABLE vMEM_CE_B_L : RBit3 := '1';
  VARIABLE vMEM_CE_C_L : RBit3 := '1';
  VARIABLE vMEM_CE_D_L : RBit3 := '1';
  VARIABLE vMEM_WE_A_L : RBit3 := '1';
  VARIABLE vMEM_WE_B_L : RBit3 := '1';
  VARIABLE vMEM_WE_C_L : RBit3 := '1';
  VARIABLE vMEM_WE_D_L : RBit3 := '1';
  VARIABLE vMEM_OE_A_L : RBit3 := '1';
  VARIABLE vMEM_OE_B_L : RBit3 := '1';
  VARIABLE vMEM_OE_C_L : RBit3 := '1';
  VARIABLE vMEM_OE_D_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_A_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_B_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_C_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_D_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_A_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_B_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_C_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_D_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_A_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_B_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_C_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_D_L : RBit3 := '1';
  VARIABLE vPE_ADDR_SEL_A : RBit3 := '0';
  VARIABLE vPE_ADDR_SEL_B : RBit3 := '0';
  VARIABLE vPE_ADDR_SEL_C : RBit3 := '0';
  VARIABLE vPE_ADDR_SEL_D : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_A : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_B : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_C : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_D : RBit3 := '0';
  VARIABLE vPE_DATA_WR_SEL_A : RBit3 := '0';
  VARIABLE vPE_DATA_WR_SEL_B : RBit3 := '0';
  VARIABLE vPE_DATA_WR_SEL_C : RBit3 := '0';
  VARIABLE vPE_DATA_WR_SEL_D : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_A : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_B : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_C : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_D : RBit3 := '0';
  VARIABLE vPE_DATA_RD_SEL_A : RBit3 := '0';
  VARIABLE vPE_DATA_RD_SEL_B : RBit3 := '0';
  VARIABLE vPE_DATA_RD_SEL_C : RBit3 := '0';
  VARIABLE vPE_DATA_RD_SEL_D : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_A : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_B : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_C : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_D : RBit3 := '0';
  VARIABLE vPE_DATA_RD_OE_A_L : RBit3 := '1';
  VARIABLE vPE_DATA_RD_OE_B_L : RBit3 := '1';
  VARIABLE vPE_DATA_RD_OE_C_L : RBit3 := '1';
  VARIABLE vPE_DATA_RD_OE_D_L : RBit3 := '1';
  VARIABLE vPE_DATA_WR_OE_A_L : RBit3 := '1';
```

```
  VARIABLE vPE_DATA_WR_OE_B_L : RBit3 := '1';
  VARIABLE vPE_DATA_WR_OE_C_L : RBit3 := '1';
  VARIABLE vPE_DATA_WR_OE_D_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_A_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_B_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_C_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_D_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_A_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_B_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_C_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_D_L : RBit3 := '1';

BEGIN  --  PROCESS STATE_MACHINE
  WAIT UNTIL MCLK'EVENT AND MCLK = '1';

  --  Reset all of the DONE flags using variables, since they may be set
  --  later on in this state.  This is done outside of the state machine
  --  case statement because of the write/read decoding sections that are
  --  also outside of the state machine (these resets need to be before the
  --  decode sections)

  IF (STATE = DECODE) THEN
    vA_WR_DONE := '0';
    vB_WR_DONE := '0';
    vC_WR_DONE := '0';
    vD_WR_DONE := '0';
    vA_RD_DONE := '0';
    vB_RD_DONE := '0';
    vC_RD_DONE := '0';
    vD_RD_DONE := '0';
  END IF;

  vtmpMEM_CE_A_L := vMEM_CE_A_L;
  vtmpMEM_CE_B_L := vMEM_CE_B_L;
  vtmpMEM_CE_C_L := vMEM_CE_C_L;
  vtmpMEM_CE_D_L := vMEM_CE_D_L;
  vtmpMEM_WE_A_L := vMEM_WE_A_L;
  vtmpMEM_WE_B_L := vMEM_WE_B_L;
  vtmpMEM_WE_C_L := vMEM_WE_C_L;
  vtmpMEM_WE_D_L := vMEM_WE_D_L;
  vtmpMEM_OE_A_L := vMEM_OE_A_L;
  vtmpMEM_OE_B_L := vMEM_OE_B_L;
  vtmpMEM_OE_C_L := vMEM_OE_C_L;
  vtmpMEM_OE_D_L := vMEM_OE_D_L;

  --  Reset all of the memory and switch control variables
  vMEM_CE_A_L := '1';
  vMEM_CE_B_L := '1';
  vMEM_CE_C_L := '1';
  vMEM_CE_D_L := '1';
  vMEM_WE_A_L := '1';
  vMEM_WE_B_L := '1';
  vMEM_WE_C_L := '1';
  vMEM_WE_D_L := '1';
  vMEM_OE_A_L := '1';
  vMEM_OE_B_L := '1';
  vMEM_OE_C_L := '1';
  vMEM_OE_D_L := '1';
  vPE_DATA_RD_OE_A_L := '1';
  vPE_DATA_RD_OE_B_L := '1';
  vPE_DATA_RD_OE_C_L := '1';
  vPE_DATA_RD_OE_D_L := '1';
  vPE_DATA_WR_OE_A_L := '1';
  vPE_DATA_WR_OE_B_L := '1';
  vPE_DATA_WR_OE_C_L := '1';
  vPE_DATA_WR_OE_D_L := '1';
  vMEM_DATA_RD_OE_A_L := '1';
  vMEM_DATA_RD_OE_B_L := '1';
  vMEM_DATA_RD_OE_C_L := '1';
  vMEM_DATA_RD_OE_D_L := '1';
  vMEM_DATA_WR_OE_A_L := '1';
  vMEM_DATA_WR_OE_B_L := '1';
  vMEM_DATA_WR_OE_C_L := '1';
  vMEM_DATA_WR_OE_D_L := '1';

  --  WRITE decoding section:  This section contains the code that is used
  --  to generate the proper switch control signals for writes.  Since it is
```

```
--  used more than once in the state machine, it is easier to put it here
--  once rather than duplicate it many times in the state machine below.

IF ((STATE = DECODE) OR (STATE = WRITE)) THEN
  --  Decode the write flags and determine the source-select lines for
  --  the switch's address and data lines.
  IF (A_WR = '1') THEN
    IF (PE_SEL_A(1 downto 0) = "00") THEN
      vPE_ADDR_SEL_A := '0';
      vPE_DATA_WR_SEL_A := '0';
      vMEM_ADDR_SEL_A := '0';
      vMEM_DATA_WR_SEL_A := '0';
      vPE_DATA_WR_OE_A_L := '0';
      vMEM_DATA_WR_OE_A_L := '0';
      vMEM_CE_A_L := '0';
      vMEM_WE_A_L := '0';
    ELSIF (PE_SEL_A(1 downto 0) = "01") THEN
      vPE_ADDR_SEL_A := '0';
      vPE_DATA_WR_SEL_A := '0';
      vMEM_ADDR_SEL_B := '0';
      vMEM_DATA_WR_SEL_B := '0';
      vPE_DATA_WR_OE_A_L := '0';
      vMEM_DATA_WR_OE_B_L := '0';
      vMEM_CE_B_L := '0';
      vMEM_WE_B_L := '0';
    ELSIF (PE_SEL_A(1 downto 0) = "10") THEN
      vPE_ADDR_SEL_B := '0';
      vPE_DATA_WR_SEL_B := '0';
      vMEM_ADDR_SEL_C := '0';
      vMEM_DATA_WR_SEL_C := '0';
      vPE_DATA_WR_OE_B_L := '0';
      vMEM_DATA_WR_OE_C_L := '0';
      vMEM_CE_C_L := '0';
      vMEM_WE_C_L := '0';
    ELSIF (PE_SEL_A(1 downto 0) = "11") THEN
      vPE_ADDR_SEL_B := '0';
      vPE_DATA_WR_SEL_B := '0';
      vMEM_ADDR_SEL_D := '0';
      vMEM_DATA_WR_SEL_D := '0';
      vPE_DATA_WR_OE_B_L := '0';
      vMEM_DATA_WR_OE_D_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_WE_D_L := '0';
    END IF;
    vA_WR_DONE := '1';
  END IF;
  IF (B_WR = '1') THEN
    IF (PE_SEL_B(1 downto 0) = "00") THEN
      vPE_ADDR_SEL_A := '1';
      vPE_DATA_WR_SEL_A := '1';
      vMEM_ADDR_SEL_A := '0';
      vMEM_DATA_WR_SEL_A := '0';
      vPE_DATA_WR_OE_A_L := '0';
      vMEM_DATA_WR_OE_A_L := '0';
      vMEM_CE_A_L := '0';
      vMEM_WE_A_L := '0';
    ELSIF (PE_SEL_B(1 downto 0) = "01") THEN
      vPE_ADDR_SEL_A := '1';
      vPE_DATA_WR_SEL_A := '1';
      vMEM_ADDR_SEL_B := '0';
      vMEM_DATA_WR_SEL_B := '0';
      vPE_DATA_WR_OE_A_L := '0';
      vMEM_DATA_WR_OE_B_L := '0';
      vMEM_CE_B_L := '0';
      vMEM_WE_B_L := '0';
    ELSIF (PE_SEL_B(1 downto 0) = "10") THEN
      vPE_ADDR_SEL_B := '1';
      vPE_DATA_WR_SEL_B := '1';
      vMEM_ADDR_SEL_C := '0';
      vMEM_DATA_WR_SEL_C := '0';
      vPE_DATA_WR_OE_B_L := '0';
      vMEM_DATA_WR_OE_C_L := '0';
      vMEM_CE_C_L := '0';
      vMEM_WE_C_L := '0';
    ELSIF (PE_SEL_B(1 downto 0) = "11") THEN
      vPE_ADDR_SEL_B := '1';
      vPE_DATA_WR_SEL_B := '1';
```

```
    vMEM_ADDR_SEL_D := '0';
    vMEM_DATA_WR_SEL_D := '0';
    vPE_DATA_WR_OE_B_L := '0';
    vMEM_DATA_WR_OE_D_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_WE_D_L := '0';
  END IF;
  vB_WR_DONE := '1';
END IF;
IF (C_WR = '1') THEN
  IF (PE_SEL_C(1 downto 0) = "00") THEN
    vPE_ADDR_SEL_C := '0';
    vPE_DATA_WR_SEL_C := '0';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_WR_SEL_A := '1';
    vPE_DATA_WR_OE_C_L := '0';
    vMEM_DATA_WR_OE_A_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_WE_A_L := '0';
  ELSIF (PE_SEL_C(1 downto 0) = "01") THEN
    vPE_ADDR_SEL_C := '0';
    vPE_DATA_WR_SEL_C := '0';
    vMEM_ADDR_SEL_B := '1';
    vMEM_DATA_WR_SEL_B := '1';
    vPE_DATA_WR_OE_C_L := '0';
    vMEM_DATA_WR_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_WE_B_L := '0';
  ELSIF (PE_SEL_C(1 downto 0) = "10") THEN
    vPE_ADDR_SEL_D := '0';
    vPE_DATA_WR_SEL_D := '0';
    vMEM_ADDR_SEL_C := '1';
    vMEM_DATA_WR_SEL_C := '1';
    vPE_DATA_WR_OE_D_L := '0';
    vMEM_DATA_WR_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_WE_C_L := '0';
  ELSIF (PE_SEL_C(1 downto 0) = "11") THEN
    vPE_ADDR_SEL_D := '0';
    vPE_DATA_WR_SEL_D := '0';
    vMEM_ADDR_SEL_D := '1';
    vMEM_DATA_WR_SEL_D := '1';
    vPE_DATA_WR_OE_D_L := '0';
    vMEM_DATA_WR_OE_D_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_WE_D_L := '0';
  END IF;
  vC_WR_DONE := '1';
END IF;
IF (D_WR = '1') THEN
  IF (PE_SEL_D(1 downto 0) = "00") THEN
    vPE_ADDR_SEL_C := '1';
    vPE_DATA_WR_SEL_C := '1';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_WR_SEL_A := '1';
    vPE_DATA_WR_OE_C_L := '0';
    vMEM_DATA_WR_OE_A_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_WE_A_L := '0';
  ELSIF (PE_SEL_D(1 downto 0) = "01") THEN
    vPE_ADDR_SEL_C := '1';
    vPE_DATA_WR_SEL_C := '1';
    vMEM_ADDR_SEL_B := '1';
    vMEM_DATA_WR_SEL_B := '1';
    vPE_DATA_WR_OE_C_L := '0';
    vMEM_DATA_WR_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_WE_B_L := '0';
  ELSIF (PE_SEL_D(1 downto 0) = "10") THEN
    vPE_ADDR_SEL_D := '1';
    vPE_DATA_WR_SEL_D := '1';
    vMEM_ADDR_SEL_C := '1';
    vMEM_DATA_WR_SEL_C := '1';
    vPE_DATA_WR_OE_D_L := '0';
    vMEM_DATA_WR_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_WE_C_L := '0';
```

```
    ELSIF (PE_SEL_D(1 downto 0) = "11") THEN
      vPE_ADDR_SEL_D := '1';
      vPE_DATA_WR_SEL_D := '1';
      vMEM_ADDR_SEL_D := '1';
      vMEM_DATA_WR_SEL_D := '1';
      vPE_DATA_WR_OE_D_L := '0';
      vMEM_DATA_WR_OE_D_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_WE_D_L := '0';
    END IF;
    vD_WR_DONE := '1';
  END IF;
END IF;


--  READ decoding section:  This section contains the code that is used
--  to generate the proper switch control signals for writes.  Since it is
--  used more than once in the state machine, it is easier to put it here
--  once rather than duplicate it many times in the state machine below.

IF (((STATE = DECODE) AND (WR_DONE = '1'))
  OR ((STATE = WRITE)  AND (WR_DONE = '1'))
  OR  (STATE = READ)) THEN
  IF (A_RD = '1') THEN
    IF (PE_SEL_A(1 downto 0) = "00") THEN
      vPE_ADDR_SEL_A := '0';
      vPE_DATA_RD_SEL_A := '0';
      vMEM_ADDR_SEL_A := '0';
      vMEM_DATA_RD_SEL_A := '0';
      vPE_DATA_RD_OE_A_L := '0';
      vMEM_DATA_RD_OE_A_L := '0';
      vMEM_CE_A_L := '0';
      vMEM_OE_A_L := '0';
    ELSIF (PE_SEL_A(1 downto 0) = "01") THEN
      vPE_ADDR_SEL_A := '0';
      vPE_DATA_RD_SEL_A := '0';
      vMEM_ADDR_SEL_B := '0';
      vMEM_DATA_RD_SEL_A := '1';
      vPE_DATA_RD_OE_A_L := '0';
      vMEM_DATA_RD_OE_A_L := '0';
      vMEM_CE_B_L := '0';
      vMEM_OE_B_L := '0';
    ELSIF (PE_SEL_A(1 downto 0) = "10") THEN
      vPE_ADDR_SEL_B := '0';
      vPE_DATA_RD_SEL_A := '1';
      vMEM_ADDR_SEL_C := '0';
      vMEM_DATA_RD_SEL_C := '0';
      vPE_DATA_RD_OE_A_L := '0';
      vMEM_DATA_RD_OE_C_L := '0';
      vMEM_CE_C_L := '0';
      vMEM_OE_C_L := '0';
    ELSIF (PE_SEL_A(1 downto 0) = "11") THEN
      vPE_ADDR_SEL_B := '0';
      vPE_DATA_RD_SEL_A := '1';
      vMEM_ADDR_SEL_D := '0';
      vMEM_DATA_RD_SEL_C := '1';
      vPE_DATA_RD_OE_A_L := '0';
      vMEM_DATA_RD_OE_C_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_OE_D_L := '0';
    END IF;
    vA_RD_DONE := '1';
  END IF;
  IF (B_RD = '1') AND (B_RD_MUST_DELAY  = '0') THEN
    IF (PE_SEL_B(1 downto 0) = "00") THEN
      vPE_ADDR_SEL_A := '1';
      vPE_DATA_RD_SEL_B := '0';
      vMEM_ADDR_SEL_A := '0';
      vMEM_DATA_RD_SEL_A := '0';
      vPE_DATA_RD_OE_B_L := '0';
      vMEM_DATA_RD_OE_A_L := '0';
      vMEM_CE_A_L := '0';
      vMEM_OE_A_L := '0';
    ELSIF (PE_SEL_B(1 downto 0) = "01") THEN
      vPE_ADDR_SEL_A := '1';
      vPE_DATA_RD_SEL_B := '0';
      vMEM_ADDR_SEL_B := '0';
```

```
    vMEM_DATA_RD_SEL_A := '1';
    vPE_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_A_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_OE_B_L := '0';
  ELSIF (PE_SEL_B(1 downto 0) = "10") THEN
    vPE_ADDR_SEL_B := '1';
    vPE_DATA_RD_SEL_B := '1';
    vMEM_ADDR_SEL_C := '0';
    vMEM_DATA_RD_SEL_C := '0';
    vPE_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_OE_C_L := '0';
  ELSIF (PE_SEL_B(1 downto 0) = "11") THEN
    vPE_ADDR_SEL_B := '1';
    vPE_DATA_RD_SEL_B := '1';
    vMEM_ADDR_SEL_D := '0';
    vMEM_DATA_RD_SEL_C := '1';
    vPE_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_C_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_OE_D_L := '0';
  END IF;
  vB_RD_DONE := '1';
END IF;
IF (C_RD = '1') THEN
  IF (PE_SEL_C(1 downto 0) = "00") THEN
    vPE_ADDR_SEL_C := '0';
    vPE_DATA_RD_SEL_C := '0';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_RD_SEL_B := '0';
    vPE_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_OE_A_L := '0';
  ELSIF (PE_SEL_C(1 downto 0) = "01") THEN
    vPE_ADDR_SEL_C := '0';
    vPE_DATA_RD_SEL_C := '0';
    vMEM_ADDR_SEL_B := '1';
    vMEM_DATA_RD_SEL_B := '1';
    vPE_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_OE_B_L := '0';
  ELSIF (PE_SEL_C(1 downto 0) = "10") THEN
    vPE_ADDR_SEL_D := '0';
    vPE_DATA_RD_SEL_C := '1';
    vMEM_ADDR_SEL_C := '1';
    vMEM_DATA_RD_SEL_D := '0';
    vPE_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_D_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_OE_C_L := '0';
  ELSIF (PE_SEL_C(1 downto 0) = "11") THEN
    vPE_ADDR_SEL_D := '0';
    vPE_DATA_RD_SEL_C := '1';
    vMEM_ADDR_SEL_D := '1';
    vMEM_DATA_RD_SEL_D := '1';
    vPE_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_D_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_OE_D_L := '0';
  END IF;
  vC_RD_DONE := '1';
END IF;
IF (D_RD = '1') AND (D_RD_MUST_DELAY = '0') THEN
  IF (PE_SEL_D(1 downto 0) = "00") THEN
    vPE_ADDR_SEL_C := '1';
    vPE_DATA_RD_SEL_D := '0';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_RD_SEL_B := '0';
    vPE_DATA_RD_OE_D_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_OE_A_L := '0';
  ELSIF (PE_SEL_D(1 downto 0) = "01") THEN
```

```
      vPE_ADDR_SEL_C := '1';
      vPE_DATA_RD_SEL_D := '0';
      vMEM_ADDR_SEL_B := '1';
      vMEM_DATA_RD_SEL_B := '1';
      vPE_DATA_RD_OE_D_L := '0';
      vMEM_DATA_RD_OE_B_L := '0';
      vMEM_CE_B_L := '0';
      vMEM_OE_B_L := '0';
    ELSIF (PE_SEL_D(1 downto 0) = "10") THEN
      vPE_ADDR_SEL_D := '1';
      vPE_DATA_RD_SEL_D := '1';
      vMEM_ADDR_SEL_C := '1';
      vMEM_DATA_RD_SEL_D := '0';
      vPE_DATA_RD_OE_D_L := '0';
      vMEM_DATA_RD_OE_D_L := '0';
      vMEM_CE_C_L := '0';
      vMEM_OE_C_L := '0';
    ELSIF (PE_SEL_D(1 downto 0) = "11") THEN
      vPE_ADDR_SEL_D := '1';
      vPE_DATA_RD_SEL_D := '1';
      vMEM_ADDR_SEL_D := '1';
      vMEM_DATA_RD_SEL_D := '1';
      vPE_DATA_RD_OE_D_L := '0';
      vMEM_DATA_RD_OE_D_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_OE_D_L := '0';
    END IF;
    vD_RD_DONE := '1';
  END IF;
END IF;

-----------------------------------------------------------------------------
--  State Machine
-----------------------------------------------------------------------------
CASE STATE IS

  -----------------------------------------------------------------------------
  --
  --  RESET State
  --
  --  The following takes place in this state:
  --        - DONE flags are reset
  --        - iPCLK is asserted
  --
  --  The next state is:
  --        - SYNC_UP
  --
  -----------------------------------------------------------------------------
  WHEN RESET =>
    vA_WR_DONE := '0';
    vB_WR_DONE := '0';
    vC_WR_DONE := '0';
    vD_WR_DONE := '0';
    vA_RD_DONE := '0';
    vB_RD_DONE := '0';
    vC_RD_DONE := '0';
    vD_RD_DONE := '0';
    iPCLK <= '1';
    STATE <= RESET_DELAY;

  WHEN RESET_DELAY =>
    STATE <= DECODE;

  -----------------------------------------------------------------------------
  --
  --  SYNC_UP State
  --
  --  The following takes place in this state:
  --
  --  The next state is:
  --        - DECODE
  --
  -----------------------------------------------------------------------------
  WHEN SYNC_UP =>
    B_RD_DELAYED <= '0';
    D_RD_DELAYED <= '0';
    vA_WR_DONE := '0';
```

```
  vB_WR_DONE := '0';
  vC_WR_DONE := '0';
  vD_WR_DONE := '0';
  vA_RD_DONE := '0';
  vB_RD_DONE := '0';
  vC_RD_DONE := '0';
  vD_RD_DONE := '0';
  IF ((SYNC_1 = '0') AND (SYNC_2 = '0') AND (SYNC_3 = '0')) THEN
    iPCLK <= '1';
    STATE <= DECODE;
  ELSE
    STATE <= SYNC_UP;
  END IF;

-----------------------------------------------------------------------
--
--  DECODE State
--
--  The following takes place in this state:
--        - DONE flags are reset
--        - Pending reads/writes are scheduled
--        - iPCLK is toggled if no reads/writes are pending
--
--
--  The next state is:
--        - WRITE if writes are pending,
--        - READ if reads are pending,
--        - DECODE otherwise.
--
-----------------------------------------------------------------------
WHEN DECODE =>

  B_RD_DELAYED <= '0';
  D_RD_DELAYED <= '0';

  IF ((A_WR = '1') OR (B_WR = '1') OR (C_WR = '1') OR (D_WR = '1')) THEN
    STATE <= WRITE;
  ELSIF ((A_RD = '1') OR (B_RD = '1') OR (C_RD = '1') OR (D_RD = '1')) THEN
    STATE <= READ;
  ELSE
    iPCLK <= '0';
    STATE <= SYNC_UP;
  END IF;

-----------------------------------------------------------------------
--
--  WRITE State
--
--  The following takes place in this state:
--        - Pending writes are scheduled
--        - iPCLK is toggled if no reads/writes are pending
--
--  The next state is:
--        - WRITE if pending writes exist
--        - READ if pending reads exist
--        - DECODE otherwise
--
-----------------------------------------------------------------------
WHEN WRITE =>
  IF ((A_WR = '1') OR (B_WR = '1') OR (C_WR = '1') OR (D_WR = '1')) THEN
    STATE <= WRITE;
  ELSIF ((A_RD = '1') OR (B_RD = '1') OR (C_RD = '1') OR (D_RD = '1')) THEN
    STATE <= READ;
  ELSE
    iPCLK <= '0';
    STATE <= SYNC_UP;
  END IF;

-----------------------------------------------------------------------
--
--  READ State
--
--  The following takes place in this state:
--        - Pending reads are scheduled
--        - iPCLK is toggled if no reads are pending
--
--  The next state is:
```

```
  --        - READ if pending reads exist
  --        - DECODE otherwise
  --
  ---------------------------------------------------------------------------
  WHEN READ =>
    IF (C_RD = '1') THEN
      STATE <= READ;
    ELSIF ((B_RD = '1') AND (B_RD_MUST_DELAY = '1')) THEN
      B_RD_DELAYED <= '1';
      STATE <= READ_DELAY;
    ELSIF ((B_RD = '1') AND (B_RD_MUST_DELAY = '0')) THEN
      STATE <= READ;
    ELSIF ((D_RD = '1') AND (D_RD_MUST_DELAY = '1')) THEN
      D_RD_DELAYED <= '1';
      STATE <= READ_DELAY;
    ELSIF ((D_RD = '1') AND (D_RD_MUST_DELAY = '0')) THEN
      STATE <= READ;
    ELSE
      iPCLK <= '0';
      STATE <= SYNC_UP;
    END IF;

  ---------------------------------------------------------------------------
  --
  --  READ_DELAY State
  --
  --  The following takes place in this state:
  --
  --  The next state is:
  --
  ---------------------------------------------------------------------------
  WHEN READ_DELAY =>
    STATE <= READ;

  --  Just in case:
  WHEN others =>
    STATE <= RESET;

END CASE;                              --  STATE

---------------------------------------------------------------------------
--  Signal assignment section
---------------------------------------------------------------------------
A_WR_DONE <= vA_WR_DONE;
B_WR_DONE <= vB_WR_DONE;
C_WR_DONE <= vC_WR_DONE;
D_WR_DONE <= vD_WR_DONE;
A_RD_DONE <= vA_RD_DONE;
B_RD_DONE <= vB_RD_DONE;
C_RD_DONE <= vC_RD_DONE;
D_RD_DONE <= vD_RD_DONE;
iPE_DATA_RD_LE_A_L <= NOT A_RD_DONE;
iPE_DATA_RD_LE_B_L <= NOT B_RD_DONE;
iPE_DATA_RD_LE_C_L <= NOT C_RD_DONE;
iPE_DATA_RD_LE_D_L <= NOT D_RD_DONE;
PE_DATA_RD_LE_A_L <= iPE_DATA_RD_LE_A_L after tSEXP;
PE_DATA_RD_LE_B_L <= iPE_DATA_RD_LE_B_L after tSEXP;
PE_DATA_RD_LE_C_L <= iPE_DATA_RD_LE_C_L after tSEXP;
PE_DATA_RD_LE_D_L <= iPE_DATA_RD_LE_D_L after tSEXP;
iMEM_CE_A_L <= vMEM_CE_A_L after tSEXP;
iMEM_CE_B_L <= vMEM_CE_B_L after tSEXP;
iMEM_CE_C_L <= vMEM_CE_C_L after tSEXP;
iMEM_CE_D_L <= vMEM_CE_D_L after tSEXP;
iMEM_WE_A_L <= vMEM_WE_A_L after tSEXP;
iMEM_WE_B_L <= vMEM_WE_B_L after tSEXP;
iMEM_WE_C_L <= vMEM_WE_C_L after tSEXP;
iMEM_WE_D_L <= vMEM_WE_D_L after tSEXP;
iMEM_OE_A_L <= vMEM_OE_A_L after tSEXP;
iMEM_OE_B_L <= vMEM_OE_B_L after tSEXP;
iMEM_OE_C_L <= vMEM_OE_C_L after tSEXP;
iMEM_OE_D_L <= vMEM_OE_D_L after tSEXP;
iPE_ADDR_SEL_A <= vPE_ADDR_SEL_A;
iPE_ADDR_SEL_B <= vPE_ADDR_SEL_B;
iPE_ADDR_SEL_C <= vPE_ADDR_SEL_C;
iPE_ADDR_SEL_D <= vPE_ADDR_SEL_D;
iMEM_ADDR_SEL_A <= vMEM_ADDR_SEL_A;
iMEM_ADDR_SEL_B <= vMEM_ADDR_SEL_B;
```

```
    iMEM_ADDR_SEL_C <= vMEM_ADDR_SEL_C;
    iMEM_ADDR_SEL_D <= vMEM_ADDR_SEL_D;
    iPE_DATA_WR_SEL_A <= vPE_DATA_WR_SEL_A;
    iPE_DATA_WR_SEL_B <= vPE_DATA_WR_SEL_B;
    iPE_DATA_WR_SEL_C <= vPE_DATA_WR_SEL_C;
    iPE_DATA_WR_SEL_D <= vPE_DATA_WR_SEL_D;
    iMEM_DATA_WR_SEL_A <= vMEM_DATA_WR_SEL_A;
    iMEM_DATA_WR_SEL_B <= vMEM_DATA_WR_SEL_B;
    iMEM_DATA_WR_SEL_C <= vMEM_DATA_WR_SEL_C;
    iMEM_DATA_WR_SEL_D <= vMEM_DATA_WR_SEL_D;
    iPE_DATA_RD_SEL_A <= vPE_DATA_RD_SEL_A;
    iPE_DATA_RD_SEL_B <= vPE_DATA_RD_SEL_B;
    iPE_DATA_RD_SEL_C <= vPE_DATA_RD_SEL_C;
    iPE_DATA_RD_SEL_D <= vPE_DATA_RD_SEL_D;
    iMEM_DATA_RD_SEL_A <= vMEM_DATA_RD_SEL_A;
    iMEM_DATA_RD_SEL_B <= vMEM_DATA_RD_SEL_B;
    iMEM_DATA_RD_SEL_C <= vMEM_DATA_RD_SEL_C;
    iMEM_DATA_RD_SEL_D <= vMEM_DATA_RD_SEL_D;
    iPE_DATA_RD_OE_A_L <= NOT A_RD_DONE;
    iPE_DATA_RD_OE_B_L <= NOT B_RD_DONE;
    iPE_DATA_RD_OE_C_L <= NOT C_RD_DONE;
    iPE_DATA_RD_OE_D_L <= NOT D_RD_DONE;
    iPE_DATA_WR_OE_A_L <= vPE_DATA_WR_OE_A_L;
    iPE_DATA_WR_OE_B_L <= vPE_DATA_WR_OE_B_L;
    iPE_DATA_WR_OE_C_L <= vPE_DATA_WR_OE_C_L;
    iPE_DATA_WR_OE_D_L <= vPE_DATA_WR_OE_D_L;
    iMEM_DATA_RD_OE_A_L <= vMEM_DATA_RD_OE_A_L;
    iMEM_DATA_RD_OE_B_L <= vMEM_DATA_RD_OE_B_L;
    iMEM_DATA_RD_OE_C_L <= vMEM_DATA_RD_OE_C_L;
    iMEM_DATA_RD_OE_D_L <= vMEM_DATA_RD_OE_D_L;
    MEM_DATA_RD_OE_A_L <= iMEM_DATA_RD_OE_A_L after tPD;
    MEM_DATA_RD_OE_B_L <= iMEM_DATA_RD_OE_B_L after tPD;
    MEM_DATA_RD_OE_C_L <= iMEM_DATA_RD_OE_C_L after tPD;
    MEM_DATA_RD_OE_D_L <= iMEM_DATA_RD_OE_D_L after tPD;
    iMEM_DATA_WR_OE_A_L <= vMEM_DATA_WR_OE_A_L;
    iMEM_DATA_WR_OE_B_L <= vMEM_DATA_WR_OE_B_L;
    iMEM_DATA_WR_OE_C_L <= vMEM_DATA_WR_OE_C_L;
    iMEM_DATA_WR_OE_D_L <= vMEM_DATA_WR_OE_D_L;

  END PROCESS STATE_MACHINE;
END Behavioral;
```

Appendix C

This appendix contains the behavioral VHDL description of the global memory register/mux component of the shared-memory simulation model.

```
library IEEE,GMS2;
USE IEEE.std_logic_1164.all;
use GMS2.TYPES.all;

-------------------------------------------------------------------------------
--
--  GMREG_CTRL:
--
--      A global memory register controller.
--
--      I/O: 44 (64 total in an Altera EPM7064)
--
-------------------------------------------------------------------------------
ENTITY GMREG_CTRL IS
  port (
    MCLK                : in Bit;                 --
    MTP                 : in RBit3 := 'Z';        --
    PTP                 : in RBit3 := 'Z';        --
    CL_REQ              : out RBit3 := 'Z';       --
    CL_ACK              : in RBit3 := 'Z';        --
    PE_RD_A_L           : in RBit3 := 'Z';        --
    PE_RD_B_L           : in RBit3 := 'Z';        --
    PE_RD_C_L           : in RBit3 := 'Z';        --
    PE_RD_D_L           : in RBit3 := 'Z';        --
    PE_WR_A_L           : in RBit3 := 'Z';        --
    PE_WR_B_L           : in RBit3 := 'Z';        --  10
    PE_WR_C_L           : in RBit3 := 'Z';        --
    PE_WR_D_L           : in RBit3 := 'Z';        --
    PE_REQ_A            : in RBit3 := 'Z';        --
    PE_REQ_B            : in RBit3 := 'Z';        --
    PE_REQ_C            : in RBit3 := 'Z';        --
    PE_REQ_D            : in RBit3 := 'Z';        --
    PE_ACK_A            : out RBit3 := 'Z';       --
    PE_ACK_B            : out RBit3 := 'Z';       --
    PE_ACK_C            : out RBit3 := 'Z';       --
    PE_ACK_D            : out RBit3 := 'Z';       --  20
    ADDR_OE_A           : out RBit3 := 'Z';       --
    ADDR_OE_B           : out RBit3 := 'Z';       --
    ADDR_OE_C           : out RBit3 := 'Z';       --
    ADDR_OE_D           : out RBit3 := 'Z';       --
    DATA_RD_OE_A_L      : out RBit3 := 'Z';       --
    DATA_RD_OE_B_L      : out RBit3 := 'Z';       --
    DATA_RD_OE_C_L      : out RBit3 := 'Z';       --
    DATA_RD_OE_D_L      : out RBit3 := 'Z';       --
    DATA_WR_OE_A        : out RBit3 := 'Z';       --
    DATA_WR_OE_B        : out RBit3 := 'Z';       --  30
    DATA_WR_OE_C        : out RBit3 := 'Z';       --
    DATA_WR_OE_D        : out RBit3 := 'Z';       --
    ADDR_STRB_A         : out RBit3 := 'Z';       --
    ADDR_STRB_B         : out RBit3 := 'Z';       --
    ADDR_STRB_C         : out RBit3 := 'Z';       --
    ADDR_STRB_D         : out RBit3 := 'Z';       --
    DATA_RD_STRB_A      : out RBit3 := 'Z';       --
    DATA_RD_STRB_B      : out RBit3 := 'Z';       --
    DATA_RD_STRB_C      : out RBit3 := 'Z';       --
    DATA_RD_STRB_D      : out RBit3 := 'Z';       --  40
    DATA_WR_STRB_A      : out RBit3 := 'Z';       --
    DATA_WR_STRB_B      : out RBit3 := 'Z';       --
    DATA_WR_STRB_C      : out RBit3 := 'Z';       --
    DATA_WR_STRB_D      : out RBit3 := 'Z'        --  44
  );
```

```
END GMREG_CTRL;

ARCHITECTURE Behavioral OF GMREG_CTRL IS

  CONSTANT tPD : TIME := 8 ns;

  TYPE STATES IS (RESET,SYNC_UP,DECODE,A_RD,B_RD,C_RD,D_RD,A_WR,B_WR,C_WR,D_WR,
                  A_RD_DELAY,B_RD_DELAY,C_RD_DELAY,D_RD_DELAY);
  SIGNAL STATE : STATES;

  SIGNAL GM_BUSY_A : RBit3 := 'Z';
  SIGNAL GM_BUSY_B : RBit3 := 'Z';
  SIGNAL GM_BUSY_C : RBit3 := 'Z';
  SIGNAL GM_BUSY_D : RBit3 := 'Z';

  SIGNAL A_WR_REQ : RBit3 := 'Z';
  SIGNAL B_WR_REQ : RBit3 := 'Z';
  SIGNAL C_WR_REQ : RBit3 := 'Z';
  SIGNAL D_WR_REQ : RBit3 := 'Z';

  SIGNAL A_WR_DONE : RBit3 := '0';
  SIGNAL B_WR_DONE : RBit3 := '0';
  SIGNAL C_WR_DONE : RBit3 := '0';
  SIGNAL D_WR_DONE : RBit3 := '0';

  SIGNAL A_RD_REQ : RBit3 := 'Z';
  SIGNAL B_RD_REQ : RBit3 := 'Z';
  SIGNAL C_RD_REQ : RBit3 := 'Z';
  SIGNAL D_RD_REQ : RBit3 := 'Z';

  SIGNAL A_RD_DONE : RBit3 := '0';
  SIGNAL B_RD_DONE : RBit3 := '0';
  SIGNAL C_RD_DONE : RBit3 := '0';
  SIGNAL D_RD_DONE : RBit3 := '0';

  SIGNAL iDATA_RD_STRB_A : RBit3 := 'Z';
  SIGNAL iDATA_RD_STRB_B : RBit3 := 'Z';
  SIGNAL iDATA_RD_STRB_C : RBit3 := 'Z';
  SIGNAL iDATA_RD_STRB_D : RBit3 := 'Z';

BEGIN  --  Behavioral

  A_WR_REQ <= PE_REQ_A AND (NOT PE_WR_A_L) AND (NOT A_WR_DONE);
  B_WR_REQ <= (NOT A_WR_REQ) AND PE_REQ_B AND (NOT PE_WR_B_L) AND (NOT B_WR_DONE);
  C_WR_REQ <= (NOT A_WR_REQ) AND (NOT B_WR_REQ)
              AND PE_REQ_C AND (NOT PE_WR_C_L) AND (NOT C_WR_DONE);
  D_WR_REQ <= (NOT A_WR_REQ) AND (NOT B_WR_REQ) AND (NOT C_WR_REQ)
              AND PE_REQ_D AND (NOT PE_WR_D_L) AND (NOT D_WR_DONE);

  A_RD_REQ <= PE_REQ_A AND (NOT PE_RD_A_L) AND (NOT A_RD_DONE);
  B_RD_REQ <= (NOT A_RD_REQ) AND PE_REQ_B AND (NOT PE_RD_B_L) AND (NOT B_RD_DONE);
  C_RD_REQ <= (NOT A_RD_REQ) AND (NOT B_RD_REQ)
              AND PE_REQ_C AND (NOT PE_RD_C_L) AND (NOT C_RD_DONE);
  D_RD_REQ <= (NOT A_RD_REQ) AND (NOT B_RD_REQ) AND (NOT C_RD_REQ)
              AND PE_REQ_D AND (NOT PE_RD_D_L) AND (NOT D_RD_DONE);

  --  Fix this: must depend on who has bus at time of read.
  DATA_RD_OE_A_L <= PE_RD_A_L after tPD;
  DATA_RD_OE_B_L <= PE_RD_B_L after tPD;
  DATA_RD_OE_C_L <= PE_RD_C_L after tPD;
  DATA_RD_OE_D_L <= PE_RD_D_L after tPD;

  DATA_RD_STRB_A <= (NOT (iDATA_RD_STRB_A AND MTP)) after tPD;
  DATA_RD_STRB_B <= (NOT (iDATA_RD_STRB_B AND MTP)) after tPD;
  DATA_RD_STRB_C <= (NOT (iDATA_RD_STRB_C AND MTP)) after tPD;
  DATA_RD_STRB_D <= (NOT (iDATA_RD_STRB_D AND MTP)) after tPD;

  PE_ACK_A <= GM_BUSY_A after tPD;
  PE_ACK_B <= GM_BUSY_B after tPD;
  PE_ACK_C <= GM_BUSY_C after tPD;
  PE_ACK_D <= GM_BUSY_D after tPD;

  ---------------------------------------------------------------------------
  --  State Machine Process
  ---------------------------------------------------------------------------
  STATE_MACHINE : PROCESS
```

```
  VARIABLE vSTARTED_CYCLE : RBit3 := '0';
  VARIABLE vCL_REQ : RBit3 := 'Z';
  VARIABLE vPE_ACK_A : RBit3 := 'Z';
  VARIABLE vPE_ACK_B : RBit3 := 'Z';
  VARIABLE vPE_ACK_C : RBit3 := 'Z';
  VARIABLE vPE_ACK_D : RBit3 := 'Z';
  VARIABLE vADDR_OE_A : RBit3 := 'Z';
  VARIABLE vADDR_OE_B : RBit3 := 'Z';
  VARIABLE vADDR_OE_C : RBit3 := 'Z';
  VARIABLE vADDR_OE_D : RBit3 := 'Z';
  VARIABLE vDATA_RD_OE_A_L : RBit3 := 'Z';
  VARIABLE vDATA_RD_OE_B_L : RBit3 := 'Z';
  VARIABLE vDATA_RD_OE_C_L : RBit3 := 'Z';
  VARIABLE vDATA_RD_OE_D_L : RBit3 := 'Z';
  VARIABLE vDATA_WR_OE_A : RBit3 := 'Z';
  VARIABLE vDATA_WR_OE_B : RBit3 := 'Z';
  VARIABLE vDATA_WR_OE_C : RBit3 := 'Z';
  VARIABLE vDATA_WR_OE_D : RBit3 := 'Z';
  VARIABLE vADDR_STRB_A : RBit3 := 'Z';
  VARIABLE vADDR_STRB_B : RBit3 := 'Z';
  VARIABLE vADDR_STRB_C : RBit3 := 'Z';
  VARIABLE vADDR_STRB_D : RBit3 := 'Z';
  VARIABLE vDATA_RD_STRB_A : RBit3 := 'Z';
  VARIABLE vDATA_RD_STRB_B : RBit3 := 'Z';
  VARIABLE vDATA_RD_STRB_C : RBit3 := 'Z';
  VARIABLE vDATA_RD_STRB_D : RBit3 := 'Z';
  VARIABLE vDATA_WR_STRB_A : RBit3 := 'Z';
  VARIABLE vDATA_WR_STRB_B : RBit3 := 'Z';
  VARIABLE vDATA_WR_STRB_C : RBit3 := 'Z';
  VARIABLE vDATA_WR_STRB_D : RBit3 := 'Z';

BEGIN  --  PROCESS STATE_MACHINE
  WAIT UNTIL MCLK'EVENT AND MCLK = '1';

  ---------------------------------------------------------------------------
  --  State Machine
  ---------------------------------------------------------------------------
  CASE STATE IS

    ---------------------------------------------------------------------------
    --
    --  RESET State
    --
    --  The following takes place in this state:
    --        - DONE flags are reset
    --
    --  The next state is:
    --        - DECODE
    --
    ---------------------------------------------------------------------------
    WHEN RESET =>
      vCL_REQ := '0';
      vADDR_STRB_A := '0';
      vADDR_STRB_B := '0';
      vADDR_STRB_C := '0';
      vADDR_STRB_D := '0';
      vDATA_WR_STRB_A := '0';
      vDATA_WR_STRB_B := '0';
      vDATA_WR_STRB_C := '0';
      vDATA_WR_STRB_D := '0';
      vDATA_RD_STRB_A := '0';
      vDATA_RD_STRB_B := '0';
      vDATA_RD_STRB_C := '0';
      vDATA_RD_STRB_D := '0';
      A_WR_DONE <= '0';
      B_WR_DONE <= '0';
      C_WR_DONE <= '0';
      D_WR_DONE <= '0';
      A_RD_DONE <= '0';
      B_RD_DONE <= '0';
      C_RD_DONE <= '0';
      D_RD_DONE <= '0';
      GM_BUSY_A <= '0';
      GM_BUSY_B <= '0';
      GM_BUSY_C <= '0';
      GM_BUSY_D <= '0';
      vSTARTED_CYCLE := '0';
```

```
  State <= DECODE;

WHEN SYNC_UP =>
  IF (vSTARTED_CYCLE = '1') AND (PTP = '1') THEN
    STATE <= SYNC_UP;
  ELSIF ((vSTARTED_CYCLE = '1') AND (PTP = '0')) THEN
    vCL_REQ := '0';
    vADDR_STRB_A := '0';
    vADDR_STRB_B := '0';
    vADDR_STRB_C := '0';
    vADDR_STRB_D := '0';
    vDATA_WR_STRB_A := '0';
    vDATA_WR_STRB_B := '0';
    vDATA_WR_STRB_C := '0';
    vDATA_WR_STRB_D := '0';
    vDATA_RD_STRB_A := '0';
    vDATA_RD_STRB_B := '0';
    vDATA_RD_STRB_C := '0';
    vDATA_RD_STRB_D := '0';
    A_WR_DONE <= '0';
    B_WR_DONE <= '0';
    C_WR_DONE <= '0';
    D_WR_DONE <= '0';
    A_RD_DONE <= '0';
    B_RD_DONE <= '0';
    C_RD_DONE <= '0';
    D_RD_DONE <= '0';
    GM_BUSY_A <= '0';
    GM_BUSY_B <= '0';
    GM_BUSY_C <= '0';
    GM_BUSY_D <= '0';
    vSTARTED_CYCLE := '0';
    STATE <= RESET;
  ELSE
    vCL_REQ := '0';
    vADDR_STRB_A := '0';
    vADDR_STRB_B := '0';
    vADDR_STRB_C := '0';
    vADDR_STRB_D := '0';
    vDATA_WR_STRB_A := '0';
    vDATA_WR_STRB_B := '0';
    vDATA_WR_STRB_C := '0';
    vDATA_WR_STRB_D := '0';
    vDATA_RD_STRB_A := '0';
    vDATA_RD_STRB_B := '0';
    vDATA_RD_STRB_C := '0';
    vDATA_RD_STRB_D := '0';
    A_WR_DONE <= '0';
    B_WR_DONE <= '0';
    C_WR_DONE <= '0';
    D_WR_DONE <= '0';
    A_RD_DONE <= '0';
    B_RD_DONE <= '0';
    C_RD_DONE <= '0';
    D_RD_DONE <= '0';
    GM_BUSY_A <= '0';
    GM_BUSY_B <= '0';
    GM_BUSY_C <= '0';
    GM_BUSY_D <= '0';
    vSTARTED_CYCLE := '0';
    STATE <= DECODE;
  END IF;

-----------------------------------------------------------------------
--
--  DECODE State
--
--  The following takes place in this state:
--
--
--  The next state is:
--        - RESET otherwise.
--
-----------------------------------------------------------------------
WHEN DECODE =>

  vADDR_OE_A := '0';
```

```
vADDR_OE_B := '0';
vADDR_OE_C := '0';
vADDR_OE_D := '0';
vDATA_WR_OE_A := '0';
vDATA_WR_OE_B := '0';
vDATA_WR_OE_C := '0';
vDATA_WR_OE_D := '0';
vDATA_RD_STRB_A := '0';
vDATA_RD_STRB_B := '0';
vDATA_RD_STRB_C := '0';
vDATA_RD_STRB_D := '0';

IF ((PE_REQ_A = '1') AND (GM_BUSY_A = '0')) THEN
  GM_BUSY_A <= '1';
END IF;
IF ((PE_REQ_B = '1') AND (GM_BUSY_B = '0')) THEN
  GM_BUSY_B <= '1';
END IF;
IF ((PE_REQ_C = '1') AND (GM_BUSY_C = '0')) THEN
  GM_BUSY_C <= '1';
END IF;
IF ((PE_REQ_D = '1') AND (GM_BUSY_D = '0')) THEN
  GM_BUSY_D <= '1';
END IF;

IF (CL_ACK /= '1') THEN
  IF (A_WR_REQ = '1') AND (A_WR_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_A := '1';
    vADDR_OE_A := '1';
    vDATA_WR_STRB_A := '1';
    vDATA_WR_OE_A := '1';
    vCL_REQ := '1';
    STATE <= A_WR;
  ELSIF (B_WR_REQ = '1') AND (B_WR_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_B := '1';
    vADDR_OE_B := '1';
    vDATA_WR_STRB_B := '1';
    vDATA_WR_OE_B := '1';
    vCL_REQ := '1';
    STATE <= B_WR;
  ELSIF (C_WR_REQ = '1') AND (C_WR_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_C := '1';
    vADDR_OE_C := '1';
    vDATA_WR_STRB_C := '1';
    vDATA_WR_OE_C := '1';
    vCL_REQ := '1';
    STATE <= C_WR;
  ELSIF (D_WR_REQ = '1') AND (D_WR_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_D := '1';
    vADDR_OE_D := '1';
    vDATA_WR_STRB_D := '1';
    vDATA_WR_OE_D := '1';
    vCL_REQ := '1';
    STATE <= D_WR;
  ELSIF (A_RD_REQ = '1') AND (A_RD_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_A := '1';
    vADDR_OE_A := '1';
    vCL_REQ := '1';
    STATE <= A_RD;
  ELSIF (B_RD_REQ = '1') AND (B_RD_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_B := '1';
    vADDR_OE_B := '1';
    vCL_REQ := '1';
    STATE <= B_RD;
  ELSIF (C_RD_REQ = '1') AND (C_RD_DONE = '0') THEN
    vSTARTED_CYCLE := '1';
    vADDR_STRB_C := '1';
    vADDR_OE_C := '1';
    vCL_REQ := '1';
    STATE <= C_RD;
  ELSIF (D_RD_REQ = '1') AND (D_RD_DONE = '0') THEN
```

```
      vSTARTED_CYCLE := '1';
      vADDR_STRB_D := '1';
      vADDR_OE_D := '1';
      vCL_REQ := '1';
      STATE <= D_RD;
    ELSE
      STATE <= SYNC_UP;
    END IF;
  ELSE
    State <= DECODE;
  END IF;

-----------------------------------------------------------------------
--  A_WR State
-----------------------------------------------------------------------
WHEN A_WR =>
  IF (CL_ACK = '1') THEN
    vCL_REQ := '0';
    vADDR_OE_A := '0';
    vDATA_WR_OE_A := '0';
    A_WR_DONE <= '1';
    State <= DECODE;
  ELSE
    State <= A_WR;
  END IF;

-----------------------------------------------------------------------
--  B_WR State
-----------------------------------------------------------------------
WHEN B_WR =>
  IF (CL_ACK = '1') THEN
    vCL_REQ := '0';
    vADDR_OE_B := '0';
    vDATA_WR_OE_B := '0';
    B_WR_DONE <= '1';
    State <= DECODE;
  ELSE
    State <= B_WR;
  END IF;

-----------------------------------------------------------------------
--  C_WR State
-----------------------------------------------------------------------
WHEN C_WR =>
  IF (CL_ACK = '1') THEN
    vCL_REQ := '0';
    vADDR_OE_C := '0';
    vDATA_WR_OE_C := '0';
    C_WR_DONE <= '1';
    State <= DECODE;
  ELSE
    State <= C_WR;
  END IF;

-----------------------------------------------------------------------
--  D_WR State
-----------------------------------------------------------------------
WHEN D_WR =>
  IF (CL_ACK = '1') THEN
    vCL_REQ := '0';
    vADDR_OE_D := '0';
    vDATA_WR_OE_D := '0';
    D_WR_DONE <= '1';
    State <= DECODE;
  ELSE
    State <= D_WR;
  END IF;

-----------------------------------------------------------------------
--  A_RD State
-----------------------------------------------------------------------
WHEN A_RD =>
  IF (CL_ACK = '1') THEN
    vCL_REQ := '0';
    vADDR_OE_A := '0';
    vDATA_RD_STRB_A := '1';
    A_RD_DONE <= '1';
```

```
--          State <= A_RD_DELAY;
          State <= DECODE;
        ELSE
          State <= A_RD;
        END IF;

      ---------------------------------------------------------------------
      --  B_RD State
      ---------------------------------------------------------------------
      WHEN B_RD =>
        IF (CL_ACK = '1') THEN
          vCL_REQ := '0';
          vADDR_OE_B := '0';
          vDATA_RD_STRB_B := '1';
          B_RD_DONE <= '1';
--          State <= B_RD_DELAY;
          State <= DECODE;
        ELSE
          State <= B_RD;
        END IF;

      ---------------------------------------------------------------------
      --  C_RD State
      ---------------------------------------------------------------------
      WHEN C_RD =>
        IF (CL_ACK = '1') THEN
          vCL_REQ := '0';
          vADDR_OE_C := '0';
          vDATA_RD_STRB_C := '1';
          C_RD_DONE <= '1';
--          State <= C_RD_DELAY;
          State <= DECODE;
        ELSE
          State <= C_RD;
        END IF;

      ---------------------------------------------------------------------
      --  D_RD State
      ---------------------------------------------------------------------
      WHEN D_RD =>
        IF (CL_ACK = '1') THEN
          vCL_REQ := '0';
          vADDR_OE_D := '0';
          vDATA_RD_STRB_D := '1';
          D_RD_DONE <= '1';
--          State <= D_RD_DELAY;
          State <= DECODE;
        ELSE
          State <= D_RD;
        END IF;

--      ---------------------------------------------------------------------
--      --  A_RD_DELAY State
--      ---------------------------------------------------------------------
--      WHEN A_RD_DELAY =>
--        vDATA_RD_STRB_A := '0';
--        IF (PE_ACK_A = '1') THEN
--          State <= DECODE;
--        ELSE
--          State <= A_RD_DELAY;
--        END IF;
--
--      ---------------------------------------------------------------------
--      --  B_RD_DELAY State
--      ---------------------------------------------------------------------
--      WHEN B_RD_DELAY =>
--        vDATA_RD_STRB_B := '0';
--        IF (PE_ACK_B = '1') THEN
--          State <= DECODE;
--        ELSE
--          State <= B_RD_DELAY;
--        END IF;
--
--      ---------------------------------------------------------------------
--      --  C_RD_DELAY State
--      ---------------------------------------------------------------------
--      WHEN C_RD_DELAY =>
```

```
--        vDATA_RD_STRB_C := '0';
--        IF (PE_ACK_C = '1') THEN
--          State <= DECODE;
--        ELSE
--          State <= C_RD_DELAY;
--        END IF;
--
--      ---------------------------------------------------------------------
--      --  D_RD_DELAY State
--      ---------------------------------------------------------------------
--      WHEN D_RD_DELAY =>
--        vDATA_RD_STRB_D := '0';
--        IF (PE_ACK_D = '1') THEN
--          State <= DECODE;
--        ELSE
--          State <= D_RD_DELAY;
--        END IF;

      --  Just in case:
      WHEN others =>
        STATE <= RESET;

    END CASE;                             --  STATE

    CL_REQ <= vCL_REQ after tPD;
    ADDR_OE_A <= vADDR_OE_A after tPD;
    ADDR_OE_B <= vADDR_OE_B after tPD;
    ADDR_OE_C <= vADDR_OE_C after tPD;
    ADDR_OE_D <= vADDR_OE_D after tPD;
    DATA_WR_OE_A <= vDATA_WR_OE_A after tPD;
    DATA_WR_OE_B <= vDATA_WR_OE_B after tPD;
    DATA_WR_OE_C <= vDATA_WR_OE_C after tPD;
    DATA_WR_OE_D <= vDATA_WR_OE_D after tPD;
    ADDR_STRB_A <= vADDR_STRB_A after tPD;
    ADDR_STRB_B <= vADDR_STRB_B after tPD;
    ADDR_STRB_C <= vADDR_STRB_C after tPD;
    ADDR_STRB_D <= vADDR_STRB_D after tPD;
    iDATA_RD_STRB_A <= vDATA_RD_STRB_A;
    iDATA_RD_STRB_B <= vDATA_RD_STRB_B;
    iDATA_RD_STRB_C <= vDATA_RD_STRB_C;
    iDATA_RD_STRB_D <= vDATA_RD_STRB_D;
--    DATA_RD_STRB_A <= vDATA_RD_STRB_A after tPD;
--    DATA_RD_STRB_B <= vDATA_RD_STRB_B after tPD;
--    DATA_RD_STRB_C <= vDATA_RD_STRB_C after tPD;
--    DATA_RD_STRB_D <= vDATA_RD_STRB_D after tPD;
    DATA_WR_STRB_A <= vDATA_WR_STRB_A after tPD;
    DATA_WR_STRB_B <= vDATA_WR_STRB_B after tPD;
    DATA_WR_STRB_C <= vDATA_WR_STRB_C after tPD;
    DATA_WR_STRB_D <= vDATA_WR_STRB_D after tPD;

  END PROCESS STATE_MACHINE;
END Behavioral;
```

Appendix D

This appendix contains the behavioral VHDL code that describes the global switch controller component.

```
library IEEE,GMS2;
USE IEEE.std_logic_1164.all;
use GMS2.TYPES.all;

-------------------------------------------------------------------------------
--
--  GLOBAL_SWITCH_CTRL:
--
--      A memory controller switch control unit.
--
--      I/O: 74 used (96 total for the EPM70128)
--
-------------------------------------------------------------------------------
ENTITY GLOBAL_SWITCH_CTRL IS
  port (
    MCLK                : in Bit;                --
    TP                  : in RBit3;              --
    CL_REQ_A            : in RBit3 := 'Z';       --
    CL_REQ_B            : in RBit3 := 'Z';       --
    CL_REQ_C            : in RBit3 := 'Z';       --
    CL_REQ_D            : in RBit3 := 'Z';       --
    CL_ACK_A            : out RBit3 := 'Z';       --
    CL_ACK_B            : out RBit3 := 'Z';       --
    CL_ACK_C            : out RBit3 := 'Z';       --
    CL_ACK_D            : out RBit3 := 'Z';       --
    CL_SEL_A            : in RBit3_Vector(1 downto 0) := (others => 'Z');--
    CL_SEL_B            : in RBit3_Vector(1 downto 0) := (others => 'Z');--
    CL_SEL_C            : in RBit3_Vector(1 downto 0) := (others => 'Z');--
    CL_SEL_D            : in RBit3_Vector(1 downto 0) := (others => 'Z');--  15
    CL_RD_A_L           : in RBit3 := 'Z';       --
    CL_RD_B_L           : in RBit3 := 'Z';       --
    CL_RD_C_L           : in RBit3 := 'Z';       --
    CL_RD_D_L           : in RBit3 := 'Z';       --
    CL_WR_A_L           : in RBit3 := 'Z';       --
    CL_WR_B_L           : in RBit3 := 'Z';       --
    CL_WR_C_L           : in RBit3 := 'Z';       --
    CL_WR_D_L           : in RBit3 := 'Z';       --
    MEM_CE_A_L          : out RBit3 := 'Z';      --
    MEM_CE_B_L          : out RBit3 := 'Z';      --
    MEM_CE_C_L          : out RBit3 := 'Z';      --
    MEM_CE_D_L          : out RBit3 := 'Z';      --
    MEM_WE_A_L          : out RBit3 := 'Z';      --
    MEM_WE_B_L          : out RBit3 := 'Z';      --
    MEM_WE_C_L          : out RBit3 := 'Z';      --  30
    MEM_WE_D_L          : out RBit3 := 'Z';      --
    MEM_OE_A_L          : out RBit3 := 'Z';      --
    MEM_OE_B_L          : out RBit3 := 'Z';      --
    MEM_OE_C_L          : out RBit3 := 'Z';      --
    MEM_OE_D_L          : out RBit3 := 'Z';      --
    CL_ADDR_SEL_A       : out RBit3 := 'Z';      --
    CL_ADDR_SEL_B       : out RBit3 := 'Z';      --
    CL_ADDR_SEL_C       : out RBit3 := 'Z';      --
    CL_ADDR_SEL_D       : out RBit3 := 'Z';      --
    MEM_ADDR_SEL_A      : out RBit3 := 'Z';      --  40
    MEM_ADDR_SEL_B      : out RBit3 := 'Z';      --
    MEM_ADDR_SEL_C      : out RBit3 := 'Z';      --
    MEM_ADDR_SEL_D      : out RBit3 := 'Z';      --
    CL_DATA_RD_LE_A_L   : out RBit3 := 'Z';      --
    CL_DATA_RD_LE_B_L   : out RBit3 := 'Z';      --
    CL_DATA_RD_LE_C_L   : out RBit3 := 'Z';      --
    CL_DATA_RD_LE_D_L   : out RBit3 := 'Z';      --
```

```
    CL_DATA_RD_SEL_A     : out RBit3 := 'Z';      --
    CL_DATA_RD_SEL_B     : out RBit3 := 'Z';      --
    CL_DATA_RD_SEL_C     : out RBit3 := 'Z';      --  50
    CL_DATA_RD_SEL_D     : out RBit3 := 'Z';      --
    CL_DATA_WR_SEL_A     : out RBit3 := 'Z';      --
    CL_DATA_WR_SEL_B     : out RBit3 := 'Z';      --
    CL_DATA_WR_SEL_C     : out RBit3 := 'Z';      --
    CL_DATA_WR_SEL_D     : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_A    : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_B    : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_C    : out RBit3 := 'Z';      --
    MEM_DATA_RD_SEL_D    : out RBit3 := 'Z';      --
    MEM_DATA_WR_SEL_A    : out RBit3 := 'Z';      --  60
    MEM_DATA_WR_SEL_B    : out RBit3 := 'Z';      --
    MEM_DATA_WR_SEL_C    : out RBit3 := 'Z';      --
    MEM_DATA_WR_SEL_D    : out RBit3 := 'Z';      --
    CL_DATA_RD_OE_A_L    : out RBit3 := 'Z';      --
    CL_DATA_RD_OE_B_L    : out RBit3 := 'Z';      --
    CL_DATA_RD_OE_C_L    : out RBit3 := 'Z';      --
    CL_DATA_RD_OE_D_L    : out RBit3 := 'Z';      --
    CL_DATA_WR_OE_A_L    : out RBit3 := 'Z';      --
    CL_DATA_WR_OE_B_L    : out RBit3 := 'Z';      --
    CL_DATA_WR_OE_C_L    : out RBit3 := 'Z';      --  70
    CL_DATA_WR_OE_D_L    : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_A_L   : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_B_L   : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_C_L   : out RBit3 := 'Z';      --
    MEM_DATA_RD_OE_D_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_A_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_B_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_C_L   : out RBit3 := 'Z';      --
    MEM_DATA_WR_OE_D_L   : out RBit3 := 'Z'       --  79
  );
END GLOBAL_SWITCH_CTRL;

ARCHITECTURE Behavioral OF GLOBAL_SWITCH_CTRL IS

  CONSTANT tPD : TIME := 10 ns;          --   I/O propagation delay
  CONSTANT tSEXP : TIME := 5 ns;         --   Shared expander delay

  TYPE STATES IS (RESET,RESET_DELAY,DECODE,WRITE,READ,READ_DELAY);
  SIGNAL STATE : STATES;

  SIGNAL A_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL B_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL C_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL D_DEST : RBit3_Vector(2 downto 0) := (others => 'Z');
  SIGNAL B_SAME_AS_A : RBit3 := 'Z';
  SIGNAL C_SAME_AS_A : RBit3 := 'Z';
  SIGNAL C_SAME_AS_B : RBit3 := 'Z';
  SIGNAL D_SAME_AS_A : RBit3 := 'Z';
  SIGNAL D_SAME_AS_B : RBit3 := 'Z';
  SIGNAL D_SAME_AS_C : RBit3 := 'Z';

  SIGNAL A_WR : RBit3 := 'Z';
  SIGNAL B_WR : RBit3 := 'Z';
  SIGNAL C_WR : RBit3 := 'Z';
  SIGNAL D_WR : RBit3 := 'Z';
  SIGNAL WR_DONE : RBit3 := '0';
  SIGNAL A_WR_DONE : RBit3 := '0';
  SIGNAL B_WR_DONE : RBit3 := '0';
  SIGNAL C_WR_DONE : RBit3 := '0';
  SIGNAL D_WR_DONE : RBit3 := '0';
  SIGNAL A_RD : RBit3 := 'Z';
  SIGNAL B_RD : RBit3 := 'Z';
  SIGNAL C_RD : RBit3 := 'Z';
  SIGNAL D_RD : RBit3 := 'Z';
  SIGNAL RD_DONE : RBit3 := '0';
  SIGNAL A_RD_DONE : RBit3 := '0';
  SIGNAL B_RD_DONE : RBit3 := '0';
  SIGNAL C_RD_DONE : RBit3 := '0';
  SIGNAL D_RD_DONE : RBit3 := '0';

  SIGNAL B_RD_MUST_DELAY : RBit3 := 'Z';
  SIGNAL D_RD_MUST_DELAY : RBit3 := 'Z';
  SIGNAL B_RD_DELAYED : RBit3 := 'Z';
  SIGNAL D_RD_DELAYED : RBit3 := 'Z';
```

```
  SIGNAL iTP : RBit3 := '0';
  SIGNAL iA_WR_DONE : RBit3 := '0';
  SIGNAL iB_WR_DONE : RBit3 := '0';
  SIGNAL iC_WR_DONE : RBit3 := '0';
  SIGNAL iD_WR_DONE : RBit3 := '0';
  SIGNAL iA_RD_DONE : RBit3 := '0';
  SIGNAL iB_RD_DONE : RBit3 := '0';
  SIGNAL iC_RD_DONE : RBit3 := '0';
  SIGNAL iD_RD_DONE : RBit3 := '0';
  SIGNAL iMEM_CE_A_L : RBit3 := '1';
  SIGNAL iMEM_CE_B_L : RBit3 := '1';
  SIGNAL iMEM_CE_C_L : RBit3 := '1';
  SIGNAL iMEM_CE_D_L : RBit3 := '1';
  SIGNAL iMEM_WE_A_L : RBit3 := '1';
  SIGNAL iMEM_WE_B_L : RBit3 := '1';
  SIGNAL iMEM_WE_C_L : RBit3 := '1';
  SIGNAL iMEM_WE_D_L : RBit3 := '1';
  SIGNAL iMEM_OE_A_L : RBit3 := '1';
  SIGNAL iMEM_OE_B_L : RBit3 := '1';
  SIGNAL iMEM_OE_C_L : RBit3 := '1';
  SIGNAL iMEM_OE_D_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_LE_A_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_LE_B_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_LE_C_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_LE_D_L : RBit3 := '1';
  SIGNAL iCL_ADDR_SEL_A : RBit3 := '0';
  SIGNAL iCL_ADDR_SEL_B : RBit3 := '0';
  SIGNAL iCL_ADDR_SEL_C : RBit3 := '0';
  SIGNAL iCL_ADDR_SEL_D : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_A : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_B : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_C : RBit3 := '0';
  SIGNAL iMEM_ADDR_SEL_D : RBit3 := '0';
  SIGNAL iCL_DATA_WR_SEL_A : RBit3 := '0';
  SIGNAL iCL_DATA_WR_SEL_B : RBit3 := '0';
  SIGNAL iCL_DATA_WR_SEL_C : RBit3 := '0';
  SIGNAL iCL_DATA_WR_SEL_D : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_A : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_B : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_C : RBit3 := '0';
  SIGNAL iMEM_DATA_WR_SEL_D : RBit3 := '0';
  SIGNAL iCL_DATA_RD_SEL_A : RBit3 := '0';
  SIGNAL iCL_DATA_RD_SEL_B : RBit3 := '0';
  SIGNAL iCL_DATA_RD_SEL_C : RBit3 := '0';
  SIGNAL iCL_DATA_RD_SEL_D : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_A : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_B : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_C : RBit3 := '0';
  SIGNAL iMEM_DATA_RD_SEL_D : RBit3 := '0';
  SIGNAL iCL_DATA_RD_OE_A_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_OE_B_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_OE_C_L : RBit3 := '1';
  SIGNAL iCL_DATA_RD_OE_D_L : RBit3 := '1';
  SIGNAL iCL_DATA_WR_OE_A_L : RBit3 := '1';
  SIGNAL iCL_DATA_WR_OE_B_L : RBit3 := '1';
  SIGNAL iCL_DATA_WR_OE_C_L : RBit3 := '1';
  SIGNAL iCL_DATA_WR_OE_D_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_A_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_B_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_C_L : RBit3 := '1';
  SIGNAL iMEM_DATA_RD_OE_D_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_A_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_B_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_C_L : RBit3 := '1';
  SIGNAL iMEM_DATA_WR_OE_D_L : RBit3 := '1';

BEGIN  --  Behavioral

  ---------------------------------------------------------------------------
  --  Temporary assignment of cluster ack signals
  ---------------------------------------------------------------------------
  CL_ACK_A <= A_RD_DONE OR A_WR_DONE after tPD;
  CL_ACK_B <= B_RD_DONE OR B_WR_DONE after tPD;
  CL_ACK_C <= C_RD_DONE OR C_WR_DONE after tPD;
  CL_ACK_D <= D_RD_DONE OR D_WR_DONE after tPD;
```

```
-----------------------------------------------------------------------------
--  Assign latch enable signals here.  These will be removed eventually.
-----------------------------------------------------------------------------
MEM_ADDR_LE_L        <= '1';
MEM_DATA_WR_LE_A_L   <= '1';
MEM_DATA_WR_LE_B_L   <= '1';
MEM_DATA_WR_LE_C_L   <= '1';
MEM_DATA_WR_LE_D_L   <= '1';

-----------------------------------------------------------------------------
--  Assign memory control signals here.
-----------------------------------------------------------------------------
MEM_CE_A_L <= iMEM_CE_A_L after tPD;
MEM_CE_B_L <= iMEM_CE_B_L after tPD;
MEM_CE_C_L <= iMEM_CE_C_L after tPD;
MEM_CE_D_L <= iMEM_CE_D_L after tPD;
MEM_WE_A_L <= (TP OR iMEM_WE_A_L) after tPD;
MEM_WE_B_L <= (TP OR iMEM_WE_B_L) after tPD;
MEM_WE_C_L <= (TP OR iMEM_WE_C_L) after tPD;
MEM_WE_D_L <= (TP OR iMEM_WE_D_L) after tPD;
MEM_OE_A_L <= iMEM_OE_A_L after tPD;
MEM_OE_B_L <= iMEM_OE_B_L after tPD;
MEM_OE_C_L <= iMEM_OE_C_L after tPD;
MEM_OE_D_L <= iMEM_OE_D_L after tPD;

-----------------------------------------------------------------------------
--  Assign switch control signals here
-----------------------------------------------------------------------------
CL_ADDR_SEL_A <= iCL_ADDR_SEL_A after tPD;
CL_ADDR_SEL_B <= iCL_ADDR_SEL_B after tPD;
CL_ADDR_SEL_C <= iCL_ADDR_SEL_C after tPD;
CL_ADDR_SEL_D <= iCL_ADDR_SEL_D after tPD;
MEM_ADDR_SEL_A <= iMEM_ADDR_SEL_A after tPD;
MEM_ADDR_SEL_B <= iMEM_ADDR_SEL_B after tPD;
MEM_ADDR_SEL_C <= iMEM_ADDR_SEL_C after tPD;
MEM_ADDR_SEL_D <= iMEM_ADDR_SEL_D after tPD;
CL_DATA_WR_SEL_A <= iCL_DATA_WR_SEL_A after tPD;
CL_DATA_WR_SEL_B <= iCL_DATA_WR_SEL_B after tPD;
CL_DATA_WR_SEL_C <= iCL_DATA_WR_SEL_C after tPD;
CL_DATA_WR_SEL_D <= iCL_DATA_WR_SEL_D after tPD;
MEM_DATA_WR_SEL_A <= iMEM_DATA_WR_SEL_A after tPD;
MEM_DATA_WR_SEL_B <= iMEM_DATA_WR_SEL_B after tPD;
MEM_DATA_WR_SEL_C <= iMEM_DATA_WR_SEL_C after tPD;
MEM_DATA_WR_SEL_D <= iMEM_DATA_WR_SEL_D after tPD;
CL_DATA_RD_SEL_A <= iCL_DATA_RD_SEL_A after tPD;
CL_DATA_RD_SEL_B <= iCL_DATA_RD_SEL_B after tPD;
CL_DATA_RD_SEL_C <= iCL_DATA_RD_SEL_C after tPD;
CL_DATA_RD_SEL_D <= iCL_DATA_RD_SEL_D after tPD;
MEM_DATA_RD_SEL_A <= iMEM_DATA_RD_SEL_A after tPD;
MEM_DATA_RD_SEL_B <= iMEM_DATA_RD_SEL_B after tPD;
MEM_DATA_RD_SEL_C <= iMEM_DATA_RD_SEL_C after tPD;
MEM_DATA_RD_SEL_D <= iMEM_DATA_RD_SEL_D after tPD;
CL_DATA_RD_OE_A_L <= NOT A_RD_DONE after tPD;
CL_DATA_RD_OE_B_L <= NOT B_RD_DONE after tPD;
CL_DATA_RD_OE_C_L <= NOT C_RD_DONE after tPD;
CL_DATA_RD_OE_D_L <= NOT D_RD_DONE after tPD;
CL_DATA_WR_OE_A_L <= iCL_DATA_WR_OE_A_L after tPD;
CL_DATA_WR_OE_B_L <= iCL_DATA_WR_OE_B_L after tPD;
CL_DATA_WR_OE_C_L <= iCL_DATA_WR_OE_C_L after tPD;
CL_DATA_WR_OE_D_L <= iCL_DATA_WR_OE_D_L after tPD;
MEM_DATA_WR_OE_A_L <= iMEM_DATA_WR_OE_A_L after tPD;
MEM_DATA_WR_OE_B_L <= iMEM_DATA_WR_OE_B_L after tPD;
MEM_DATA_WR_OE_C_L <= iMEM_DATA_WR_OE_C_L after tPD;
MEM_DATA_WR_OE_D_L <= iMEM_DATA_WR_OE_D_L after tPD;

-----------------------------------------------------------------------------
--  Decode any switch destination contentions in this section
-----------------------------------------------------------------------------
A_DEST <= CL_REQ_A & CL_SEL_A;
B_DEST <= CL_REQ_B & CL_SEL_B;
C_DEST <= CL_REQ_C & CL_SEL_C;
D_DEST <= CL_REQ_D & CL_SEL_D;

--  B is considered the same as A if they both need to go to the same second
--  stage switch (i.e. top two bits are the same) due to the switch structure
B_SAME_AS_A <= '1' WHEN (B_DEST(2 downto 1) = A_DEST(2 downto 1)) ELSE '0';
C_SAME_AS_A <= '1' WHEN (C_DEST = A_DEST) ELSE '0';
```

```
C_SAME_AS_B <= '1' WHEN (C_DEST = B_DEST) ELSE '0';
D_SAME_AS_A <= '1' WHEN (D_DEST = A_DEST) ELSE '0';
D_SAME_AS_B <= '1' WHEN (D_DEST = B_DEST) ELSE '0';
--  D is considered the same as C if they both need to go to the same second
--  stage switch (i.e. top two bits are the same) due to the switch structure
D_SAME_AS_C <= '1' WHEN (D_DEST(2 downto 1) = C_DEST(2 downto 1)) ELSE '0';

-----------------------------------------------------------------------------
--  Decode the internal write strobes in this section
-----------------------------------------------------------------------------
A_WR <= (NOT CL_WR_A_L) AND (NOT A_WR_DONE) AND (CL_REQ_A);
B_WR <= (NOT (B_SAME_AS_A AND (NOT CL_WR_A_L) AND (NOT A_WR_DONE)))
         AND (NOT CL_WR_B_L) AND (NOT B_WR_DONE) AND (CL_REQ_B);
C_WR <= (NOT ((C_SAME_AS_A AND (NOT CL_WR_A_L) AND (NOT A_WR_DONE))
         OR   (C_SAME_AS_B AND (NOT CL_WR_B_L) AND (NOT B_WR_DONE))))
         AND (NOT CL_WR_C_L) AND (NOT C_WR_DONE) AND (CL_REQ_C);
D_WR <= (NOT ((D_SAME_AS_A AND (NOT CL_WR_A_L) AND (NOT A_WR_DONE))
         OR   (D_SAME_AS_B AND (NOT CL_WR_B_L) AND (NOT B_WR_DONE))
         OR   (D_SAME_AS_C AND (NOT CL_WR_C_L) AND (NOT C_WR_DONE))))
         AND (NOT CL_WR_D_L) AND (NOT D_WR_DONE) AND (CL_REQ_D);
WR_DONE <= (A_WR_DONE OR CL_WR_A_L OR (NOT CL_REQ_A))
           AND (B_WR_DONE OR CL_WR_B_L OR (NOT CL_REQ_B))
           AND (C_WR_DONE OR CL_WR_C_L OR (NOT CL_REQ_C))
           AND (D_WR_DONE OR CL_WR_D_L OR (NOT CL_REQ_D));

-----------------------------------------------------------------------------
--  Decode the internal read strobes here
-----------------------------------------------------------------------------
A_RD <= (NOT CL_RD_A_L) AND (NOT A_RD_DONE) AND (CL_REQ_A);
B_RD <= (NOT (B_SAME_AS_A AND (NOT CL_RD_A_L) AND (NOT A_RD_DONE)))
         AND (NOT CL_RD_B_L) AND (NOT B_RD_DONE) AND (CL_REQ_B);
C_RD <= (NOT ((C_SAME_AS_A AND (NOT CL_RD_A_L) AND (NOT A_RD_DONE))
         OR   (C_SAME_AS_B AND (NOT CL_RD_B_L) AND (NOT B_RD_DONE))))
         AND (NOT CL_RD_C_L) AND (NOT C_RD_DONE) AND (CL_REQ_C);
D_RD <= (NOT ((D_SAME_AS_A AND (NOT CL_RD_A_L) AND (NOT A_RD_DONE))
         OR   (D_SAME_AS_B AND (NOT CL_RD_B_L) AND (NOT B_RD_DONE))
         OR   (D_SAME_AS_C AND (NOT CL_RD_C_L) AND (NOT C_RD_DONE))))
         AND (NOT CL_RD_D_L) AND (NOT D_RD_DONE) AND (CL_REQ_D);
RD_DONE <= (A_RD_DONE OR CL_RD_A_L OR (NOT CL_REQ_A))
          AND (B_RD_DONE OR CL_RD_B_L OR (NOT CL_REQ_B))
          AND (C_RD_DONE OR CL_RD_C_L OR (NOT CL_REQ_C))
          AND (D_RD_DONE OR CL_RD_D_L OR (NOT CL_REQ_D));

B_RD_MUST_DELAY <= B_RD AND A_RD_DONE AND (CL_SEL_A(0) XOR CL_SEL_B(0))
                   AND (NOT B_RD_DELAYED);
D_RD_MUST_DELAY <= D_RD AND C_RD_DONE AND (CL_SEL_C(0) XOR CL_SEL_D(0))
                   AND (NOT D_RD_DELAYED) AND (NOT B_RD_DELAYED);

-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
--  State Machine Process
--
--  Things To Do:
--    - shave a clock cycle off after the last write/read is completed.
--    - add control signals for the switch: might have to have a timing
--      pulse to generate memory control signals.
--            - figure out the timing for the memory CE, WE, and OE signals
--            - figure out the timing for the address latch enable.
--            - do the data latch enables
--            - do the data output enables
--
--    - once the write/read decode sections are working, move them back into
--      the state machine case statement.  once back in the individual state
--      cases, they can be optimized (i.e. removing the A_RD and A_WR
--      if-clauses from the READ and WRITE states, respectively) Also move
--      the DONE line resets back into the DECODE state.
--
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
STATE_MACHINE : PROCESS

  VARIABLE vA_WR_DONE : RBit3 := '0';
  VARIABLE vB_WR_DONE : RBit3 := '0';
  VARIABLE vC_WR_DONE : RBit3 := '0';
  VARIABLE vD_WR_DONE : RBit3 := '0';
  VARIABLE vA_RD_DONE : RBit3 := '0';
  VARIABLE vB_RD_DONE : RBit3 := '0';
```

```
  VARIABLE vC_RD_DONE : RBit3 := '0';
  VARIABLE vD_RD_DONE : RBit3 := '0';
  VARIABLE vMEM_CE_A_L : RBit3 := '1';
  VARIABLE vMEM_CE_B_L : RBit3 := '1';
  VARIABLE vMEM_CE_C_L : RBit3 := '1';
  VARIABLE vMEM_CE_D_L : RBit3 := '1';
  VARIABLE vMEM_WE_A_L : RBit3 := '1';
  VARIABLE vMEM_WE_B_L : RBit3 := '1';
  VARIABLE vMEM_WE_C_L : RBit3 := '1';
  VARIABLE vMEM_WE_D_L : RBit3 := '1';
  VARIABLE vMEM_OE_A_L : RBit3 := '1';
  VARIABLE vMEM_OE_B_L : RBit3 := '1';
  VARIABLE vMEM_OE_C_L : RBit3 := '1';
  VARIABLE vMEM_OE_D_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_A_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_B_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_C_L : RBit3 := '1';
  VARIABLE vtmpMEM_CE_D_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_A_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_B_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_C_L : RBit3 := '1';
  VARIABLE vtmpMEM_WE_D_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_A_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_B_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_C_L : RBit3 := '1';
  VARIABLE vtmpMEM_OE_D_L : RBit3 := '1';
  VARIABLE vCL_ADDR_SEL_A : RBit3 := '0';
  VARIABLE vCL_ADDR_SEL_B : RBit3 := '0';
  VARIABLE vCL_ADDR_SEL_C : RBit3 := '0';
  VARIABLE vCL_ADDR_SEL_D : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_A : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_B : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_C : RBit3 := '0';
  VARIABLE vMEM_ADDR_SEL_D : RBit3 := '0';
  VARIABLE vCL_DATA_WR_SEL_A : RBit3 := '0';
  VARIABLE vCL_DATA_WR_SEL_B : RBit3 := '0';
  VARIABLE vCL_DATA_WR_SEL_C : RBit3 := '0';
  VARIABLE vCL_DATA_WR_SEL_D : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_A : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_B : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_C : RBit3 := '0';
  VARIABLE vMEM_DATA_WR_SEL_D : RBit3 := '0';
  VARIABLE vCL_DATA_RD_SEL_A : RBit3 := '0';
  VARIABLE vCL_DATA_RD_SEL_B : RBit3 := '0';
  VARIABLE vCL_DATA_RD_SEL_C : RBit3 := '0';
  VARIABLE vCL_DATA_RD_SEL_D : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_A : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_B : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_C : RBit3 := '0';
  VARIABLE vMEM_DATA_RD_SEL_D : RBit3 := '0';
  VARIABLE vCL_DATA_RD_OE_A_L : RBit3 := '1';
  VARIABLE vCL_DATA_RD_OE_B_L : RBit3 := '1';
  VARIABLE vCL_DATA_RD_OE_C_L : RBit3 := '1';
  VARIABLE vCL_DATA_RD_OE_D_L : RBit3 := '1';
  VARIABLE vCL_DATA_WR_OE_A_L : RBit3 := '1';
  VARIABLE vCL_DATA_WR_OE_B_L : RBit3 := '1';
  VARIABLE vCL_DATA_WR_OE_C_L : RBit3 := '1';
  VARIABLE vCL_DATA_WR_OE_D_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_A_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_B_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_C_L : RBit3 := '1';
  VARIABLE vMEM_DATA_RD_OE_D_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_A_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_B_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_C_L : RBit3 := '1';
  VARIABLE vMEM_DATA_WR_OE_D_L : RBit3 := '1';

BEGIN  --  PROCESS STATE_MACHINE
  WAIT UNTIL MCLK'EVENT AND MCLK = '1';

  --  Reset all of the DONE flags using variables, since they may be set
  --  later on in this state.  This is done outside of the state machine
  --  case statement because of the write/read decoding sections that are
  --  also outside of the state machine (these resets need to be before the
  --  decode sections)

  IF (STATE = DECODE) THEN
```

```
  vA_WR_DONE := '0';
  vB_WR_DONE := '0';
  vC_WR_DONE := '0';
  vD_WR_DONE := '0';
  vA_RD_DONE := '0';
  vB_RD_DONE := '0';
  vC_RD_DONE := '0';
  vD_RD_DONE := '0';
END IF;

vtmpMEM_CE_A_L := vMEM_CE_A_L;
vtmpMEM_CE_B_L := vMEM_CE_B_L;
vtmpMEM_CE_C_L := vMEM_CE_C_L;
vtmpMEM_CE_D_L := vMEM_CE_D_L;
vtmpMEM_WE_A_L := vMEM_WE_A_L;
vtmpMEM_WE_B_L := vMEM_WE_B_L;
vtmpMEM_WE_C_L := vMEM_WE_C_L;
vtmpMEM_WE_D_L := vMEM_WE_D_L;
vtmpMEM_OE_A_L := vMEM_OE_A_L;
vtmpMEM_OE_B_L := vMEM_OE_B_L;
vtmpMEM_OE_C_L := vMEM_OE_C_L;
vtmpMEM_OE_D_L := vMEM_OE_D_L;

--  Reset all of the memory and switch control variables
vMEM_CE_A_L := '1';
vMEM_CE_B_L := '1';
vMEM_CE_C_L := '1';
vMEM_CE_D_L := '1';
vMEM_WE_A_L := '1';
vMEM_WE_B_L := '1';
vMEM_WE_C_L := '1';
vMEM_WE_D_L := '1';
vMEM_OE_A_L := '1';
vMEM_OE_B_L := '1';
vMEM_OE_C_L := '1';
vMEM_OE_D_L := '1';
vCL_DATA_RD_OE_A_L := '1';
vCL_DATA_RD_OE_B_L := '1';
vCL_DATA_RD_OE_C_L := '1';
vCL_DATA_RD_OE_D_L := '1';
vCL_DATA_WR_OE_A_L := '1';
vCL_DATA_WR_OE_B_L := '1';
vCL_DATA_WR_OE_C_L := '1';
vCL_DATA_WR_OE_D_L := '1';
vMEM_DATA_RD_OE_A_L := '1';
vMEM_DATA_RD_OE_B_L := '1';
vMEM_DATA_RD_OE_C_L := '1';
vMEM_DATA_RD_OE_D_L := '1';
vMEM_DATA_WR_OE_A_L := '1';
vMEM_DATA_WR_OE_B_L := '1';
vMEM_DATA_WR_OE_C_L := '1';
vMEM_DATA_WR_OE_D_L := '1';

--  WRITE decoding section:  This section contains the code that is used
--  to generate the proper switch control signals for writes.  Since it is
--  used more than once in the state machine, it is easier to put it here
--  once rather than duplicate it many times in the state machine below.

IF ((STATE = DECODE) OR (STATE = WRITE)) THEN
  --  Decode the write flags and determine the source-select lines for
  --  the switch's address and data lines.
  IF (A_WR = '1') THEN
    IF (CL_SEL_A(1 downto 0) = "00") THEN
      vCL_ADDR_SEL_A := '0';
      vCL_DATA_WR_SEL_A := '0';
      vMEM_ADDR_SEL_A := '0';
      vMEM_DATA_WR_SEL_A := '0';
      vCL_DATA_WR_OE_A_L := '0';
      vMEM_DATA_WR_OE_A_L := '0';
      vMEM_CE_A_L := '0';
      vMEM_WE_A_L := '0';
    ELSIF (CL_SEL_A(1 downto 0) = "01") THEN
      vCL_ADDR_SEL_A := '0';
      vCL_DATA_WR_SEL_A := '0';
      vMEM_ADDR_SEL_B := '0';
      vMEM_DATA_WR_SEL_B := '0';
      vCL_DATA_WR_OE_A_L := '0';
```

```
    vMEM_DATA_WR_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_WE_B_L := '0';
  ELSIF (CL_SEL_A(1 downto 0) = "10") THEN
    vCL_ADDR_SEL_B := '0';
    vCL_DATA_WR_SEL_B := '0';
    vMEM_ADDR_SEL_C := '0';
    vMEM_DATA_WR_SEL_C := '0';
    vCL_DATA_WR_OE_B_L := '0';
    vMEM_DATA_WR_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_WE_C_L := '0';
  ELSIF (CL_SEL_A(1 downto 0) = "11") THEN
    vCL_ADDR_SEL_B := '0';
    vCL_DATA_WR_SEL_B := '0';
    vMEM_ADDR_SEL_D := '0';
    vMEM_DATA_WR_SEL_D := '0';
    vCL_DATA_WR_OE_B_L := '0';
    vMEM_DATA_WR_OE_D_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_WE_D_L := '0';
  END IF;
  vA_WR_DONE := '1';
END IF;
IF (B_WR = '1') THEN
  IF (CL_SEL_B(1 downto 0) = "00") THEN
    vCL_ADDR_SEL_A := '1';
    vCL_DATA_WR_SEL_A := '1';
    vMEM_ADDR_SEL_A := '0';
    vMEM_DATA_WR_SEL_A := '0';
    vCL_DATA_WR_OE_A_L := '0';
    vMEM_DATA_WR_OE_A_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_WE_A_L := '0';
  ELSIF (CL_SEL_B(1 downto 0) = "01") THEN
    vCL_ADDR_SEL_A := '1';
    vCL_DATA_WR_SEL_A := '1';
    vMEM_ADDR_SEL_B := '0';
    vMEM_DATA_WR_SEL_B := '0';
    vCL_DATA_WR_OE_A_L := '0';
    vMEM_DATA_WR_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_WE_B_L := '0';
  ELSIF (CL_SEL_B(1 downto 0) = "10") THEN
    vCL_ADDR_SEL_B := '1';
    vCL_DATA_WR_SEL_B := '1';
    vMEM_ADDR_SEL_C := '0';
    vMEM_DATA_WR_SEL_C := '0';
    vCL_DATA_WR_OE_B_L := '0';
    vMEM_DATA_WR_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_WE_C_L := '0';
  ELSIF (CL_SEL_B(1 downto 0) = "11") THEN
    vCL_ADDR_SEL_B := '1';
    vCL_DATA_WR_SEL_B := '1';
    vMEM_ADDR_SEL_D := '0';
    vMEM_DATA_WR_SEL_D := '0';
    vCL_DATA_WR_OE_B_L := '0';
    vMEM_DATA_WR_OE_D_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_WE_D_L := '0';
  END IF;
  vB_WR_DONE := '1';
END IF;
IF (C_WR = '1') THEN
  IF (CL_SEL_C(1 downto 0) = "00") THEN
    vCL_ADDR_SEL_C := '0';
    vCL_DATA_WR_SEL_C := '0';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_WR_SEL_A := '1';
    vCL_DATA_WR_OE_C_L := '0';
    vMEM_DATA_WR_OE_A_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_WE_A_L := '0';
  ELSIF (CL_SEL_C(1 downto 0) = "01") THEN
    vCL_ADDR_SEL_C := '0';
    vCL_DATA_WR_SEL_C := '0';
```

```
      vMEM_ADDR_SEL_B := '1';
      vMEM_DATA_WR_SEL_B := '1';
      vCL_DATA_WR_OE_C_L := '0';
      vMEM_DATA_WR_OE_B_L := '0';
      vMEM_CE_B_L := '0';
      vMEM_WE_B_L := '0';
    ELSIF (CL_SEL_C(1 downto 0) = "10") THEN
      vCL_ADDR_SEL_D := '0';
      vCL_DATA_WR_SEL_D := '0';
      vMEM_ADDR_SEL_C := '1';
      vMEM_DATA_WR_SEL_C := '1';
      vCL_DATA_WR_OE_D_L := '0';
      vMEM_DATA_WR_OE_C_L := '0';
      vMEM_CE_C_L := '0';
      vMEM_WE_C_L := '0';
    ELSIF (CL_SEL_C(1 downto 0) = "11") THEN
      vCL_ADDR_SEL_D := '0';
      vCL_DATA_WR_SEL_D := '0';
      vMEM_ADDR_SEL_D := '1';
      vMEM_DATA_WR_SEL_D := '1';
      vCL_DATA_WR_OE_D_L := '0';
      vMEM_DATA_WR_OE_D_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_WE_D_L := '0';
    END IF;
    vC_WR_DONE := '1';
  END IF;
  IF (D_WR = '1') THEN
    IF (CL_SEL_D(1 downto 0) = "00") THEN
      vCL_ADDR_SEL_C := '1';
      vCL_DATA_WR_SEL_C := '1';
      vMEM_ADDR_SEL_A := '1';
      vMEM_DATA_WR_SEL_A := '1';
      vCL_DATA_WR_OE_C_L := '0';
      vMEM_DATA_WR_OE_A_L := '0';
      vMEM_CE_A_L := '0';
      vMEM_WE_A_L := '0';
    ELSIF (CL_SEL_D(1 downto 0) = "01") THEN
      vCL_ADDR_SEL_C := '1';
      vCL_DATA_WR_SEL_C := '1';
      vMEM_ADDR_SEL_B := '1';
      vMEM_DATA_WR_SEL_B := '1';
      vCL_DATA_WR_OE_C_L := '0';
      vMEM_DATA_WR_OE_B_L := '0';
      vMEM_CE_B_L := '0';
      vMEM_WE_B_L := '0';
    ELSIF (CL_SEL_D(1 downto 0) = "10") THEN
      vCL_ADDR_SEL_D := '1';
      vCL_DATA_WR_SEL_D := '1';
      vMEM_ADDR_SEL_C := '1';
      vMEM_DATA_WR_SEL_C := '1';
      vCL_DATA_WR_OE_D_L := '0';
      vMEM_DATA_WR_OE_C_L := '0';
      vMEM_CE_C_L := '0';
      vMEM_WE_C_L := '0';
    ELSIF (CL_SEL_D(1 downto 0) = "11") THEN
      vCL_ADDR_SEL_D := '1';
      vCL_DATA_WR_SEL_D := '1';
      vMEM_ADDR_SEL_D := '1';
      vMEM_DATA_WR_SEL_D := '1';
      vCL_DATA_WR_OE_D_L := '0';
      vMEM_DATA_WR_OE_D_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_WE_D_L := '0';
    END IF;
    vD_WR_DONE := '1';
  END IF;
END IF;


--  READ decoding section:  This section contains the code that is used
--  to generate the proper switch control signals for writes.  Since it is
--  used more than once in the state machine, it is easier to put it here
--  once rather than duplicate it many times in the state machine below.

IF (((STATE = DECODE) AND (WR_DONE = '1'))
  OR ((STATE = WRITE)  AND (WR_DONE = '1'))
```

```
OR  (STATE = READ)) THEN
IF (A_RD = '1') THEN
  IF (CL_SEL_A(1 downto 0) = "00") THEN
    vCL_ADDR_SEL_A := '0';
    vCL_DATA_RD_SEL_A := '0';
    vMEM_ADDR_SEL_A := '0';
    vMEM_DATA_RD_SEL_A := '0';
    vCL_DATA_RD_OE_A_L := '0';
    vMEM_DATA_RD_OE_A_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_OE_A_L := '0';
  ELSIF (CL_SEL_A(1 downto 0) = "01") THEN
    vCL_ADDR_SEL_A := '0';
    vCL_DATA_RD_SEL_A := '0';
    vMEM_ADDR_SEL_B := '0';
    vMEM_DATA_RD_SEL_A := '1';
    vCL_DATA_RD_OE_A_L := '0';
    vMEM_DATA_RD_OE_A_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_OE_B_L := '0';
  ELSIF (CL_SEL_A(1 downto 0) = "10") THEN
    vCL_ADDR_SEL_B := '0';
    vCL_DATA_RD_SEL_A := '1';
    vMEM_ADDR_SEL_C := '0';
    vMEM_DATA_RD_SEL_C := '0';
    vCL_DATA_RD_OE_A_L := '0';
    vMEM_DATA_RD_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_OE_C_L := '0';
  ELSIF (CL_SEL_A(1 downto 0) = "11") THEN
    vCL_ADDR_SEL_B := '0';
    vCL_DATA_RD_SEL_A := '1';
    vMEM_ADDR_SEL_D := '0';
    vMEM_DATA_RD_SEL_C := '1';
    vCL_DATA_RD_OE_A_L := '0';
    vMEM_DATA_RD_OE_C_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_OE_D_L := '0';
  END IF;
  vA_RD_DONE := '1';
END IF;
IF (B_RD = '1') AND (B_RD_MUST_DELAY  = '0') THEN
  IF (CL_SEL_B(1 downto 0) = "00") THEN
    vCL_ADDR_SEL_A := '1';
    vCL_DATA_RD_SEL_B := '0';
    vMEM_ADDR_SEL_A := '0';
    vMEM_DATA_RD_SEL_A := '0';
    vCL_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_A_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_OE_A_L := '0';
  ELSIF (CL_SEL_B(1 downto 0) = "01") THEN
    vCL_ADDR_SEL_A := '1';
    vCL_DATA_RD_SEL_B := '0';
    vMEM_ADDR_SEL_B := '0';
    vMEM_DATA_RD_SEL_A := '1';
    vCL_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_A_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_OE_B_L := '0';
  ELSIF (CL_SEL_B(1 downto 0) = "10") THEN
    vCL_ADDR_SEL_B := '1';
    vCL_DATA_RD_SEL_B := '1';
    vMEM_ADDR_SEL_C := '0';
    vMEM_DATA_RD_SEL_C := '0';
    vCL_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_C_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_OE_C_L := '0';
  ELSIF (CL_SEL_B(1 downto 0) = "11") THEN
    vCL_ADDR_SEL_B := '1';
    vCL_DATA_RD_SEL_B := '1';
    vMEM_ADDR_SEL_D := '0';
    vMEM_DATA_RD_SEL_C := '1';
    vCL_DATA_RD_OE_B_L := '0';
    vMEM_DATA_RD_OE_C_L := '0';
    vMEM_CE_D_L := '0';
```

```
    vMEM_OE_D_L := '0';
  END IF;
  vB_RD_DONE := '1';
END IF;
IF (C_RD = '1') THEN
  IF (CL_SEL_C(1 downto 0) = "00") THEN
    vCL_ADDR_SEL_C := '0';
    vCL_DATA_RD_SEL_C := '0';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_RD_SEL_B := '0';
    vCL_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_OE_A_L := '0';
  ELSIF (CL_SEL_C(1 downto 0) = "01") THEN
    vCL_ADDR_SEL_C := '0';
    vCL_DATA_RD_SEL_C := '0';
    vMEM_ADDR_SEL_B := '1';
    vMEM_DATA_RD_SEL_B := '1';
    vCL_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_OE_B_L := '0';
  ELSIF (CL_SEL_C(1 downto 0) = "10") THEN
    vCL_ADDR_SEL_D := '0';
    vCL_DATA_RD_SEL_C := '1';
    vMEM_ADDR_SEL_C := '1';
    vMEM_DATA_RD_SEL_D := '0';
    vCL_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_D_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_OE_C_L := '0';
  ELSIF (CL_SEL_C(1 downto 0) = "11") THEN
    vCL_ADDR_SEL_D := '0';
    vCL_DATA_RD_SEL_C := '1';
    vMEM_ADDR_SEL_D := '1';
    vMEM_DATA_RD_SEL_D := '1';
    vCL_DATA_RD_OE_C_L := '0';
    vMEM_DATA_RD_OE_D_L := '0';
    vMEM_CE_D_L := '0';
    vMEM_OE_D_L := '0';
  END IF;
  vC_RD_DONE := '1';
END IF;
IF (D_RD = '1') AND (D_RD_MUST_DELAY = '0') THEN
  IF (CL_SEL_D(1 downto 0) = "00") THEN
    vCL_ADDR_SEL_C := '1';
    vCL_DATA_RD_SEL_D := '0';
    vMEM_ADDR_SEL_A := '1';
    vMEM_DATA_RD_SEL_B := '0';
    vCL_DATA_RD_OE_D_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_A_L := '0';
    vMEM_OE_A_L := '0';
  ELSIF (CL_SEL_D(1 downto 0) = "01") THEN
    vCL_ADDR_SEL_C := '1';
    vCL_DATA_RD_SEL_D := '0';
    vMEM_ADDR_SEL_B := '1';
    vMEM_DATA_RD_SEL_B := '1';
    vCL_DATA_RD_OE_D_L := '0';
    vMEM_DATA_RD_OE_B_L := '0';
    vMEM_CE_B_L := '0';
    vMEM_OE_B_L := '0';
  ELSIF (CL_SEL_D(1 downto 0) = "10") THEN
    vCL_ADDR_SEL_D := '1';
    vCL_DATA_RD_SEL_D := '1';
    vMEM_ADDR_SEL_C := '1';
    vMEM_DATA_RD_SEL_D := '0';
    vCL_DATA_RD_OE_D_L := '0';
    vMEM_DATA_RD_OE_D_L := '0';
    vMEM_CE_C_L := '0';
    vMEM_OE_C_L := '0';
  ELSIF (CL_SEL_D(1 downto 0) = "11") THEN
    vCL_ADDR_SEL_D := '1';
    vCL_DATA_RD_SEL_D := '1';
    vMEM_ADDR_SEL_D := '1';
    vMEM_DATA_RD_SEL_D := '1';
```

```
      vCL_DATA_RD_OE_D_L := '0';
      vMEM_DATA_RD_OE_D_L := '0';
      vMEM_CE_D_L := '0';
      vMEM_OE_D_L := '0';
    END IF;
    vD_RD_DONE := '1';
  END IF;
END IF;

-----------------------------------------------------------------------------
--  State Machine
-----------------------------------------------------------------------------
CASE STATE IS

  -----------------------------------------------------------------------------
  --
  --  RESET State
  --
  --  The following takes place in this state:
  --        - DONE flags are reset
  --
  --  The next state is:
  --        - SYNC_UP
  --
  -----------------------------------------------------------------------------
  WHEN RESET =>
    vA_WR_DONE := '0';
    vB_WR_DONE := '0';
    vC_WR_DONE := '0';
    vD_WR_DONE := '0';
    vA_RD_DONE := '0';
    vB_RD_DONE := '0';
    vC_RD_DONE := '0';
    vD_RD_DONE := '0';
    STATE <= RESET_DELAY;

  WHEN RESET_DELAY =>
    STATE <= DECODE;

  -----------------------------------------------------------------------------
  --
  --  DECODE State
  --
  --  The following takes place in this state:
  --        - DONE flags are reset
  --        - Pending reads/writes are scheduled
  --
  --
  --  The next state is:
  --        - WRITE if writes are pending,
  --        - READ if reads are pending,
  --        - DECODE otherwise.
  --
  -----------------------------------------------------------------------------
  WHEN DECODE =>

    B_RD_DELAYED <= '0';
    D_RD_DELAYED <= '0';

    IF ((A_WR = '1') OR (B_WR = '1') OR (C_WR = '1') OR (D_WR = '1')) THEN
      STATE <= WRITE;
    ELSIF ((A_RD = '1') OR (B_RD = '1') OR (C_RD = '1') OR (D_RD = '1')) THEN
      STATE <= READ;
    ELSE
      STATE <= DECODE;
    END IF;

  -----------------------------------------------------------------------------
  --
  --  WRITE State
  --
  --  The following takes place in this state:
  --        - Pending writes are scheduled
  --
  --  The next state is:
  --        - WRITE if pending writes exist
  --        - READ if pending reads exist
```

```
  --        - DECODE otherwise
  --
  ---------------------------------------------------------------------
  WHEN WRITE =>
    IF ((A_WR = '1') OR (B_WR = '1') OR (C_WR = '1') OR (D_WR = '1')) THEN
      STATE <= WRITE;
    ELSIF ((A_RD = '1') OR (B_RD = '1') OR (C_RD = '1') OR (D_RD = '1')) THEN
      STATE <= READ;
    ELSE
      STATE <= DECODE;
    END IF;

  ---------------------------------------------------------------------
  --
  --  READ State
  --
  --  The following takes place in this state:
  --        - Pending reads are scheduled
  --
  --  The next state is:
  --        - READ if pending reads exist
  --        - DECODE otherwise
  --
  ---------------------------------------------------------------------
  WHEN READ =>
    IF (C_RD = '1') THEN
      STATE <= READ;
    ELSIF ((B_RD = '1') AND (B_RD_MUST_DELAY = '1')) THEN
      B_RD_DELAYED <= '1';
      STATE <= READ_DELAY;
    ELSIF ((B_RD = '1') AND (B_RD_MUST_DELAY = '0')) THEN
      STATE <= READ;
    ELSIF ((D_RD = '1') AND (D_RD_MUST_DELAY = '1')) THEN
      D_RD_DELAYED <= '1';
      STATE <= READ_DELAY;
    ELSIF ((D_RD = '1') AND (D_RD_MUST_DELAY = '0')) THEN
      STATE <= READ;
    ELSE
      STATE <= DECODE;
    END IF;

  ---------------------------------------------------------------------
  --
  --  READ_DELAY State
  --
  --  The following takes place in this state:
  --
  --  The next state is:
  --
  ---------------------------------------------------------------------
  WHEN READ_DELAY =>
    STATE <= READ;

  --  Just in case:
  WHEN others =>
    STATE <= RESET;

END CASE;                              --  STATE

---------------------------------------------------------------------
--  Signal assignment section
---------------------------------------------------------------------
A_WR_DONE <= vA_WR_DONE;
B_WR_DONE <= vB_WR_DONE;
C_WR_DONE <= vC_WR_DONE;
D_WR_DONE <= vD_WR_DONE;
A_RD_DONE <= vA_RD_DONE;
B_RD_DONE <= vB_RD_DONE;
C_RD_DONE <= vC_RD_DONE;
D_RD_DONE <= vD_RD_DONE;
iCL_DATA_RD_LE_A_L <= NOT A_RD_DONE;
iCL_DATA_RD_LE_B_L <= NOT B_RD_DONE;
iCL_DATA_RD_LE_C_L <= NOT C_RD_DONE;
iCL_DATA_RD_LE_D_L <= NOT D_RD_DONE;
CL_DATA_RD_LE_A_L <= iCL_DATA_RD_LE_A_L after tSEXP;
CL_DATA_RD_LE_B_L <= iCL_DATA_RD_LE_B_L after tSEXP;
CL_DATA_RD_LE_C_L <= iCL_DATA_RD_LE_C_L after tSEXP;
```

```
    CL_DATA_RD_LE_D_L <= iCL_DATA_RD_LE_D_L after tSEXP;
    iMEM_CE_A_L <= vMEM_CE_A_L after tSEXP;
    iMEM_CE_B_L <= vMEM_CE_B_L after tSEXP;
    iMEM_CE_C_L <= vMEM_CE_C_L after tSEXP;
    iMEM_CE_D_L <= vMEM_CE_D_L after tSEXP;
    iMEM_WE_A_L <= vMEM_WE_A_L after tSEXP;
    iMEM_WE_B_L <= vMEM_WE_B_L after tSEXP;
    iMEM_WE_C_L <= vMEM_WE_C_L after tSEXP;
    iMEM_WE_D_L <= vMEM_WE_D_L after tSEXP;
    iMEM_OE_A_L <= vMEM_OE_A_L after tSEXP;
    iMEM_OE_B_L <= vMEM_OE_B_L after tSEXP;
    iMEM_OE_C_L <= vMEM_OE_C_L after tSEXP;
    iMEM_OE_D_L <= vMEM_OE_D_L after tSEXP;
    iCL_ADDR_SEL_A <= vCL_ADDR_SEL_A;
    iCL_ADDR_SEL_B <= vCL_ADDR_SEL_B;
    iCL_ADDR_SEL_C <= vCL_ADDR_SEL_C;
    iCL_ADDR_SEL_D <= vCL_ADDR_SEL_D;
    iMEM_ADDR_SEL_A <= vMEM_ADDR_SEL_A;
    iMEM_ADDR_SEL_B <= vMEM_ADDR_SEL_B;
    iMEM_ADDR_SEL_C <= vMEM_ADDR_SEL_C;
    iMEM_ADDR_SEL_D <= vMEM_ADDR_SEL_D;
    iCL_DATA_WR_SEL_A <= vCL_DATA_WR_SEL_A;
    iCL_DATA_WR_SEL_B <= vCL_DATA_WR_SEL_B;
    iCL_DATA_WR_SEL_C <= vCL_DATA_WR_SEL_C;
    iCL_DATA_WR_SEL_D <= vCL_DATA_WR_SEL_D;
    iMEM_DATA_WR_SEL_A <= vMEM_DATA_WR_SEL_A;
    iMEM_DATA_WR_SEL_B <= vMEM_DATA_WR_SEL_B;
    iMEM_DATA_WR_SEL_C <= vMEM_DATA_WR_SEL_C;
    iMEM_DATA_WR_SEL_D <= vMEM_DATA_WR_SEL_D;
    iCL_DATA_RD_SEL_A <= vCL_DATA_RD_SEL_A;
    iCL_DATA_RD_SEL_B <= vCL_DATA_RD_SEL_B;
    iCL_DATA_RD_SEL_C <= vCL_DATA_RD_SEL_C;
    iCL_DATA_RD_SEL_D <= vCL_DATA_RD_SEL_D;
    iMEM_DATA_RD_SEL_A <= vMEM_DATA_RD_SEL_A;
    iMEM_DATA_RD_SEL_B <= vMEM_DATA_RD_SEL_B;
    iMEM_DATA_RD_SEL_C <= vMEM_DATA_RD_SEL_C;
    iMEM_DATA_RD_SEL_D <= vMEM_DATA_RD_SEL_D;
    iCL_DATA_RD_OE_A_L <= NOT A_RD_DONE;
    iCL_DATA_RD_OE_B_L <= NOT B_RD_DONE;
    iCL_DATA_RD_OE_C_L <= NOT C_RD_DONE;
    iCL_DATA_RD_OE_D_L <= NOT D_RD_DONE;
    iCL_DATA_WR_OE_A_L <= vCL_DATA_WR_OE_A_L;
    iCL_DATA_WR_OE_B_L <= vCL_DATA_WR_OE_B_L;
    iCL_DATA_WR_OE_C_L <= vCL_DATA_WR_OE_C_L;
    iCL_DATA_WR_OE_D_L <= vCL_DATA_WR_OE_D_L;
    iMEM_DATA_RD_OE_A_L <= vMEM_DATA_RD_OE_A_L;
    iMEM_DATA_RD_OE_B_L <= vMEM_DATA_RD_OE_B_L;
    iMEM_DATA_RD_OE_C_L <= vMEM_DATA_RD_OE_C_L;
    iMEM_DATA_RD_OE_D_L <= vMEM_DATA_RD_OE_D_L;
    MEM_DATA_RD_OE_A_L <= iMEM_DATA_RD_OE_A_L after tPD;
    MEM_DATA_RD_OE_B_L <= iMEM_DATA_RD_OE_B_L after tPD;
    MEM_DATA_RD_OE_C_L <= iMEM_DATA_RD_OE_C_L after tPD;
    MEM_DATA_RD_OE_D_L <= iMEM_DATA_RD_OE_D_L after tPD;
    iMEM_DATA_WR_OE_A_L <= vMEM_DATA_WR_OE_A_L;
    iMEM_DATA_WR_OE_B_L <= vMEM_DATA_WR_OE_B_L;
    iMEM_DATA_WR_OE_C_L <= vMEM_DATA_WR_OE_C_L;
    iMEM_DATA_WR_OE_D_L <= vMEM_DATA_WR_OE_D_L;

  END PROCESS STATE_MACHINE;
END Behavioral;
```

Vita

Brad Fross was born in Painesville, Ohio, on the 8th of October in 1969. He went to Mentor High School, in Mentor, Ohio, for his secondary education. His interests in art led him to pursue an Industrial Design degree at The Ohio State University in the fall of 1988. His interests in computers began at Ohio State when he enrolled in the Electrical Engineering Department during his second year, and graduated *cum laude* with a B.S.E.E in March of 1993. He decided to continue his education at Virginia Tech by working towards a Master of Science degree in Electrical Engineering. He built onto his previous Splash-2 experience gained at the Center for Computational Sciences, in Bowie, MD by working on the VTSplash project for two years. He received his Master's in June, 1995.

His hobbies include rowing, sailing, biking, hockey, reading, and cooking spicy foods. He looks forward to spending his time building a wooden single-scull and rowing it up and down the Severn River at his new home in Annapolis, MD.