DESIGN AND IMPLEMENTATION OF AN INTEGRATED SNOOPING DATA CACHE[1]

Gaetano Borriello, Susan Eggers, Randy Katz, Harry McKinley,
Charles Perkins, Walter Scott, Robert Sheldon, Shaun Whalen, David Wood

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT: We describe the design and single chip implementation of a small data cache memory and associated controllers. The chip can be used as a building block of a multiprocessor system, positioned between the main memory bus and an individual processor. It implements an *ownership-based* cache consistency protocol. The chip has been designed to be interfaced to the MultiBus[2] system bus and the Motorola 68000 processor. In this paper, we present our cache consistency protocol and its evaluation, and the chip architecture, design decisions, and implementation details.

KEY WORDS AND PHRASES: Tightly Coupled Multiprocessor Building Block, Snooping Data Cache, Ownership Cache Consistency Protocol, Single Chip Implementation.

1. Introduction

Advances in integrated circuit technology make it possible to achieve even denser levels of integration. However, bigger chips do not necessarily mean faster systems. The limited information that can flow across a chip's pins per unit of time, the *pin bandwidth,* remains the fundamental bottleneck. A way for microprocessor architectures to take advantage of the increasing transistor density, while reducing pin bandwidth demands, is to place more memory on chip for instruction and data caches. We already see the next generation of commercial microprocessors dedicating some of their transistor resources to on-chip instruction memories (e.g., Motorola 68020).

Modern microprocessors achieve a level of performance that is competitive with many commercially available minicomputers. A very high performance system can be constructed at low cost by building it from multiple microprocessors. In this paper, we adopt an architectural model

[1] Borriello, Eggers, Katz, Scott, and Wood supported by the Defense Advanced Research Projects Agency (DoD) under contract N00034-K-0251. Perkins supported by a National Science Foundation Fellowship. Whalen supported by the Control Data Corporation and a California MICRO fellowship.

[2] MultiBus is a trademark of the Intel Corporation.

based on a small number of microprocessors (from four to twelve) connected to a shared memory through a system bus.

On-chip caches in a multiprocessor system require some mechanism for maintaining *consistency* across the caches, i.e., at no point in time should the entries in different caches for the same block of memory have different values. These mechanisms are called *cache consistency protocols*. The performance of a shared memory multimicroprocessor will be critically dependent on system bus utilization and memory system latency. Cache consistency protocols should be designed to minimize their affect on these aspects of system performance, while still keeping the caches consistent.

In this paper we describe the design and rapid prototype implementation of an *integrated snooping data cache*, a single chip system consisting of (1) a data cache memory, (2) a cache controller that interfaces with the processor, and (3) a "snooping controller" that monitors the system bus. The cache and snoop controllers together implement a cache consistency protocol.

In the next section, we describe the project's starting assumptions and intended goals. In section 3, the multiprocessor cache consistency problem is described. We present our cache consistency protocol, and compare it with other candidates for implementation. Section 4 describes the chip architecture and the interactions among its major components. The implementation details are given in section 5. In section 6, we review the design methodology employed and the implementation lessons learned. The current status of the implementation is given in section 7, and future directions are discussed in section 8.

2. Assumptions and Goals

The design and implementation of the integrated snooping data cache was undertaken by a group of students (the authors of this paper, nominally led by Professor Randy Katz) who participated in Berkeley's VLSI Implementation Seminar in the Spring of 1984. The course had been the basis of the RISC [PATT81, PATT83] and SOAR [UNGA84] implementation efforts in previous years.

Software development remains a difficult problem for multiprocessor systems. An inexpensive multiprocessor workstation would be a good starting point for software experimentation. To get programmers to use such a workstation, even its uniprocessor performance must be competitive with existing workstations. The workstation offers an additional incentive in that programmers can exploit the parallelism inherent in multiple processors should they wish to do so. To facilitate multiprocessor application development, some data sharing among processors should be supported by the hardware.

Thus, a basic assumption is that a multiprocessor system built from a small number of fast processors, connected together on a shared memory bus, would be an interesting research system. Such an architecture represents an evolutionary, rather than a revolutionary, path to multiprocessor systems. The single pathway to memory, in conjunction with the desire for high performance, dictated that caches be associated with each processor. We have assumed that write sharing occurs frequently enough that programmmers will not want to deal with it in software. The hardware will support write sharing.

The primary goal was to implement a single chip system that would contain a small data cache memory, cache controller, and snoop controller. The immediate objective was to build a rapid prototype of the cache chip in an effort to discover the implementation complexity of an integrated snooping cache. As such, we were more concerned with functional correctness and completeness than high performance. All design decisions were heavily weighted towards a quick implementation. To make the rapid prototype immediately useful, the Intel MultiBus was chosen as the system bus and the Motorola 68000 was selected as the target processor. This gave us the opportunity to build a multiprocessor system quickly, primarily with off-the-shelf parts, and to collect critical system design parameters for the future design of a single chip processor with on-chip caches.

3. Multiprocessor Cache Consistency

3.1. Consistency Issues and Protocols

A *Snooping Data Cache* is a data cache system whose control includes a bus monitor, which is called the *Snoop*. Besides servicing its own processor's requests, the cache monitors system bus transactions and may perform operations based on these external requests. The level of sophistication of the snoop control varies with the cache consistency protocol it implements.

Three kinds of consistency protocols have been proposed that use a snooping cache. A *write-through* strategy [AGRA77] writes all cache updates through to the memory system. Caches of other processors on the bus must monitor bus transactions, and invalidate (or copy to) any entries that match when the memory block is written through. A processor's performance is degraded whenever it performs a write, because of the additional latency of writing through to memory.

A second strategy is called *write-first* [GOOD83, THAC83]. On the first write to a cached entry, the update is written through to memory. This forces other caches to invalidate a matching entry, thus guaranteeing the writing processor that its holds the only cached copy. Subsequent writes can be performed in the cache. A processor read will be serviced either by the memory or by a cache, whichever has the most up-to-date version of the block. Snooping caches implementing this protocol are more complicated than those implementing write-through, because they must service external read requests in addition to performing invalidations. Note that write-first incurs an initial write to memory even when a memory block is not being shared. This represents an extra memory write if subsequent processor writes are directed to that block because of the copy-through to memory.

The third strategy, a version of which we have implemented, is called *ownership* (e.g., [FRAN84]). In this approach, a processor must "own" a block of memory before it is allowed to update it. Ownership is acquired through special read and write operations. By thus predeclaring an intention to update a portion of memory, the extra writes exhibited in the above protocols are

avoided, because the updating cache does not need to signal the invalidation to the other caches in the system. However, additional bus transactions may be incurred if the processor does not correctly predeclare its intentions. Reducing the number of necessary transactions improves system performance by reducing bus utilization, thus making the bus available to more processors, and avoiding memory system latency.

Some subtle differences exist among the proposed protocols, independent of the strategy employed. For example, how does memory know that it is not supposed to respond to a bus request? In some variations, memory is inhibited by the cache that will respond. In others, memory is smart enough to remember that it does not own the requested block. Another issue is whether globally requested blocks are returned to memory. For example, both [GOOD83] (write-first) and [FRAN84] (ownership) require that the cache return a block to main memory after it has been accessed by another processor. The Berkeley Protocol is based on ownership, owning caches inhibit memory, and owned blocks are kept in the cache.

3.2. The Berkeley Ownership Protocol: Concepts and Examples

Before describing the protocol, some cache terminology must be introduced. A *block* is a logical unit of memory consisting of one or a small number of words. It is identified by its address, and is the unit of transfer between main memory and the caches. A copy of a block's contents can simultaneously reside in main memory and/or in several of the cache memories.

A *cache entry* is a physical slot within cache memory that consists of a data portion, a tag, and a state. It is analogous to a page frame in a virtual memory system. The *data portion* holds the cached copy of a memory block. The *tag* is the portion of the block address that is used by cache's address mapping algorithm to determine whether the block is in the cache. Different blocks can be mapped to the same cache entry, and the tag is used to distinguish among these. The *state* encodes the state of the data portion of the cache entry, which is determined by the ownership protocol. The four possible state values are: **Invalid, UnOwned, Owned**

Exclusively, or **Owned NonExclusively.**[3] To understand the values, we must introduce the concept of ownership.

A copy of a memory block can reside in one of a cache's entries. To *own* the block bestows on the cache the right to update it by updating the entry's data. Ownership also makes the cache responsible for providing the data to other requesting caches and for updating main memory when the data is replaced in the cache. If the state is **Owned Exclusively**, then the owning cache holds the only cached copy of the block. Thus, local updates can occur without informing the other caches. Conversely, a state of **Owned NonExclusively** implies that other caches have copies and must be informed about updates to the block. Both states imply the obligation to fulfill the ownership responsibilities. The **UnOwned** state carries neither rights nor responsibilities for a cache. In this case, multiple caches may have copies and main memory is the implicit owner. A brief summary of the implications of each state are given in Table 3.1.

A state of an entry changes values in response to a system bus or processor operation that affects the entry's validity or exclusiveness. The system bus operations are the conventional bus *Read* and *Write*, augmented with special versions that help implement the protocol. A brief description of these operations is given in Table 3.2. A more complete description of each operation will be given in the next section.

Table 3.1 -- Summary of Cache Entry States

State	Description
Invalid	does not contain useful data.
UnOwned	contains a valid block, possibly shared among other caches; cannot be written locally without acquiring ownership first.
Owned Exclusively	the entry's block is unique, therefore data can be updated locally; its data must be given to any requesting cache and (eventually) flushed back to main memory.
Owned NonExclusively	the entry's block is owned, but it cannot be updated without informing the other caches.

[3] **States** are shown in boldface, while *Bus Operations* are shown in italics. In addition, new concepts are introduced in italics.

Table 3.2 – Bus Operation Summary	
Operation	**Description**
Read	a conventional read, gives a cache an UnOwned copy of the block; the data may be provided by a cache owner rather than by main memory.
Write	a conventional write, causes main memory to be updated and all cached copies to be invalidated; only issued by I/O devices and other bus users without caches.
Read-For-Ownership	like a normal read except that the requesting cache becomes the exclusive owner after the read completes.
Write-For-Invalidation	a quick version of the conventional write; need not entirely update main memory, but is guaranteed to invalidate any other cached copies; main memory will be updated correctly later when the (owned) block is flushed from its cache.
Write-Without-Invalidation	main memory is updated, but any cached copies are kept valid; used for flushing owned blocks to memory, but not necessary for correct operation since a normal Write could be used; having the extra operation avoids unnecessary invalidations.

We can now present some examples to show how the protocol behaves for different processor requests in various states. Each of the following four figures shows the states of different caches containing copies of the same block both before and after the operation. We adopt a three letter abbreviation to indicate the state of the entry. The abbreviations are: **INV** for **Invalid, UNO** for **UnOwned, EXC** for **Owned Exclusively,** and **NON** for **Owned NonExclusively.** All entries with an "A" next to their state contain the same data. A dotted line indicates that data was sent along the system bus. Each diagram will now be explained in turn.

Figure 3.1 indicates a processor read request being fulfilled by main memory. The Cache Controller of the requesting processor noticed that its entry was **Invalid,** so it signaled a *Read* on the system bus, which was handled by main memory in the usual fashion. The new entry is marked **UnOwned,** since it now contains a valid but read-only copy of the block. Notice that another cache already had a copy of the block. Since that cache did not own the block, it was not required to respond to the request.

Figure 3.2 depicts a processor read request in which the block is supplied by a cache and not by main memory. The Cache Controller of the requesting processor noticed that its block was

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

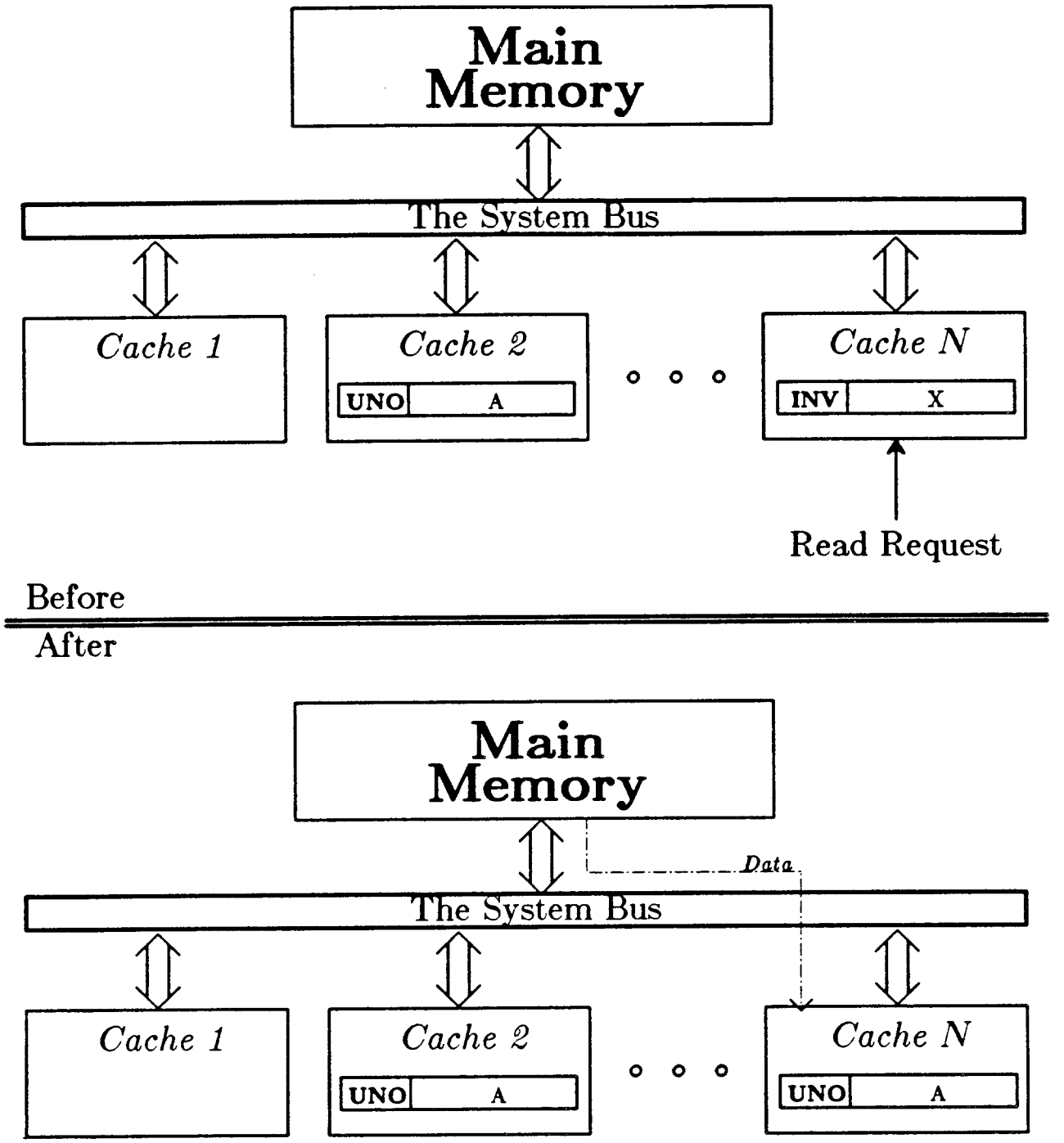


Figure 3.1 – Processor Read on Invalid Data

The Cache Controller in *Cache N* misses and sends out a *Read* request over the system bus to get the data. Main memory, the implicit owner here, replies to the request. The data ("A") travels along the system bus (as indicated by the broken line), and is placed into an entry in *Cache N*. This entry is then marked **UnOwned.** Note that although *Cache 2* had a copy of the data requested, it was not the owner and did not need to respond.

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

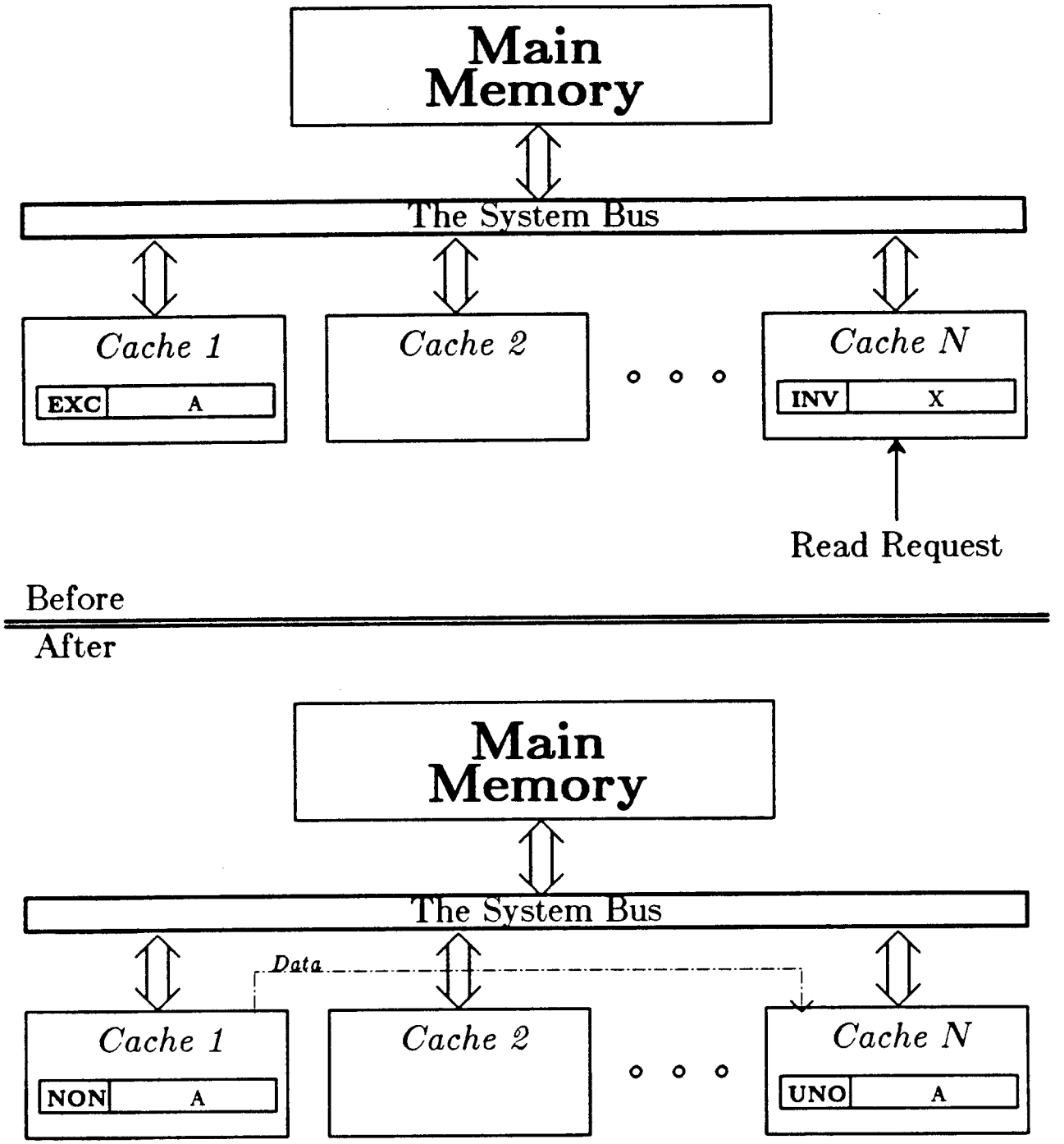


Figure 3.2 – Processor Read on Invalid/Exclusively-Owned Data

The Cache Controller in *Cache N* misses and sends out a *Read* request over the system bus to get the data. *Cache 1*, which owns the data, must respond to the request and supply the data. The data ("A") travels along the system bus (as indicated by the broken line), and is placed into an entry in *Cache N*. This entry is then marked **UnOwned.** Note that *Cache 1* must change its entry for "A" to *Owned NonExclusively*, since it no longer contains a unique copy of A.

Invalid and sent out a normal *Read* request on the system bus. The leftmost cache's Snoop

Controller detected that the read was for a block that it owned. It therefore inhibited main memory from responding, and sent the data over the system bus itself. In addition, its snoop changed the entry's state to **Owned NonExclusively,** because another cache had been given a copy of the block. This other cache marks its copy **UnOwned,** as before. If additional requests are made for the same block, the owning Snoop will continue to supply the data but will not need to change its local state again -- it can remain **Owned NonExclusively.**

Figure 3.3 shows a processor write being handled by the rightmost Cache Controller. Since that cache has a valid copy of the block, it does not need to perform a *Read-For-Ownership* to acquire the data. It can "steal" ownership by sending a *Write-For-Invalidation* over the bus and then updating its copy locally. When the Snoops in the other caches see this write, they will invalidate their local copies of the data. The rightmost cache can mark its newly updated entry as **Owned Exclusively,** since it is now guaranteed to contain unique data. If the entry's state had originally been **Owned NonExclusively** rather than **UnOwned**, the same procedure would have been followed.

The last example, figure 3.4, is again a processor write request, but here the requesting processor does not have a copy of the block in its cache. Therefore its Cache Controller must do a *Read-For-Ownership* to obtain the data before it can update it. As in the previous example, the rightmost Cache Controller changes its newly written entry to **Owned Exclusively** while the other caches invalidate their copies. Any further writes to this block can now be performed locally without any system bus traffic.

3.3. The Berkeley Ownership Protocol: Detailed Description

We will now present the protocol in more detail. The protocol will be described in terms of two independent finite state machines: the Cache Controller and the Snoop Controller. We will describe each in turn. The *Cache Controller* is primarily responsible for its own processor's use of the cache, interposing itself between the processor and memory. In addition, it assists in maintaining multiprocessor cache consistency: it must update the state of the cached block whenever it

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

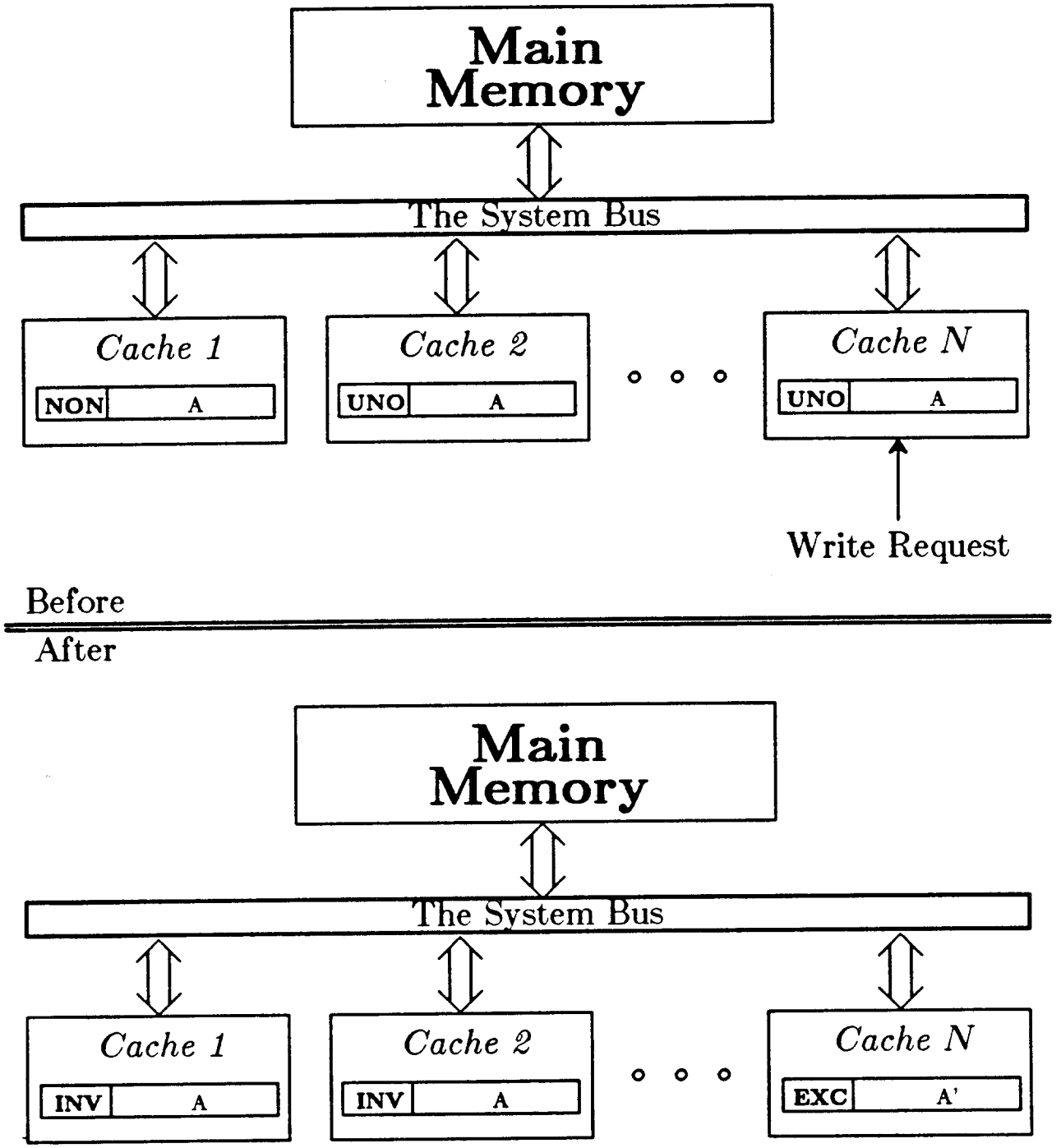


Figure 3.3 -- Processor Write on Valid UnOwned Data

The Cache Controller in *Cache N* has a current copy of the data ("A"), and so does not need to perform a read. It "steals" ownership with a *Write-For-Invalidation*, which causes both *Cache 1* and *Cache 2* to invalidate their local entries containing A. No data travels along the system bus. *Cache N* modifies its entry locally (to A'), and the entry is then marked **Owned Exclusively**, since it contains a unique copy of A'.

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

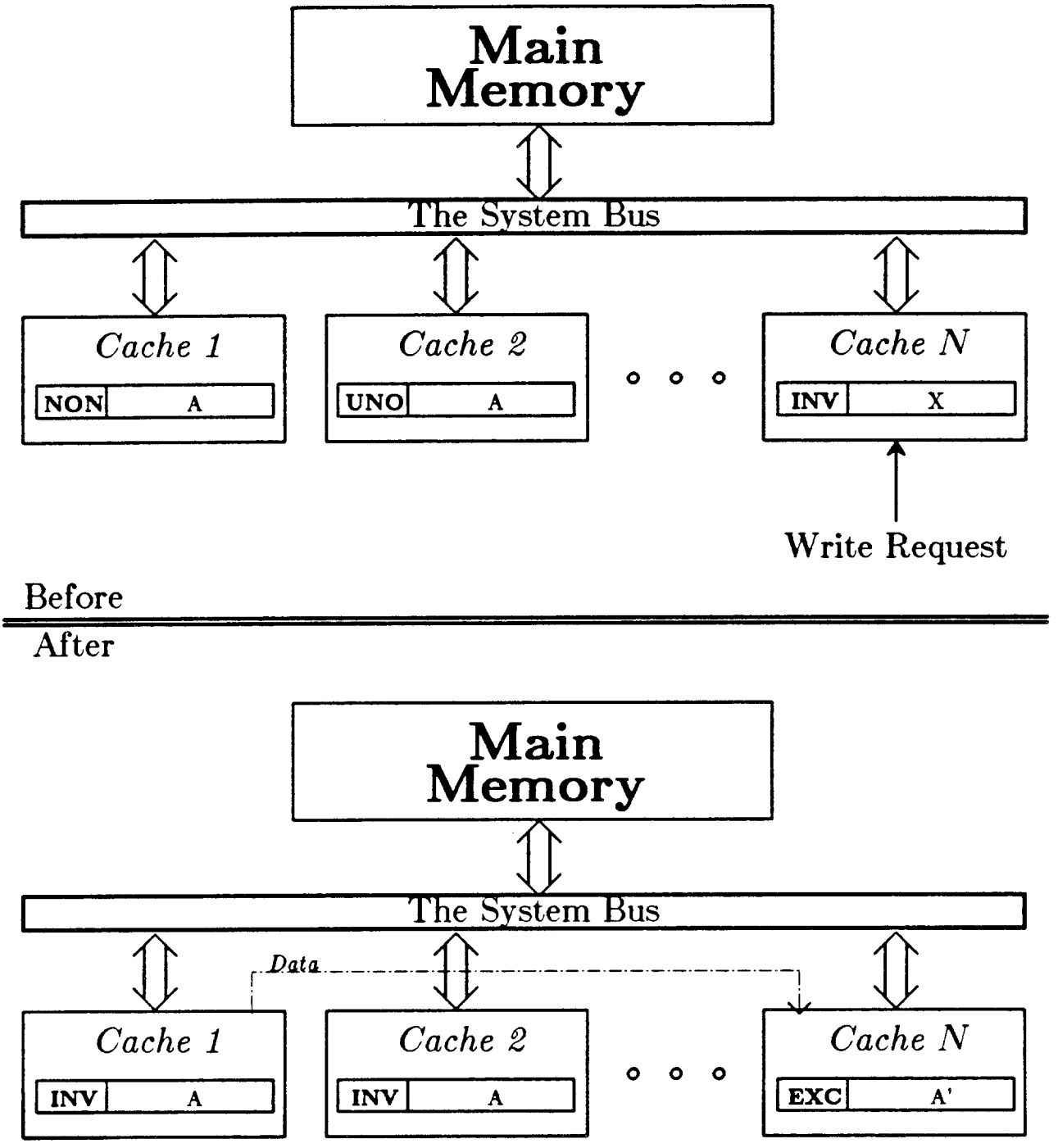


Figure 3.4 – Processor Write on Invalid Data

Cache N misses and needs to read the data ("A"). It must also modify this data, so it sends out a *Read-For-Ownership* request over the bus. *Cache 1*, the owner, responds (as shown by the broken line above), and sets its entry containing A to **Invalid,** because A is about to be updated elsewhere. *Cache 2*, noticing the ownership transfer, invalidates its entry for A. *Cache N* now updates the data locally (from A to A') and can then mark its entry **Owned Exclusively,** since it now contains a unique copy of A'.

obtains or relinquishes ownership, and it must synchronize its actions with the Snoop Controller whenever it enters a critical section, e.g., to change an entry's state bits. The Cache Controller is not responsible for responding to actions of other processors. This is the job of the Snoop.

3.3.1. The Cache Controller

The Cache Controller's behavior depends on: its processor's request (read or write), whether the data is in the cache (hit or miss), and the state of the cache entry (if a hit). When a processor read results in a cache hit (see figure 3.5), the appropriate word is provided to the processor. On a miss, the controller must first find an available cache entry, flushing data back to memory with a *Write-Without-Invalidation* if the replaced entry had been owned by the cache. It then issues a

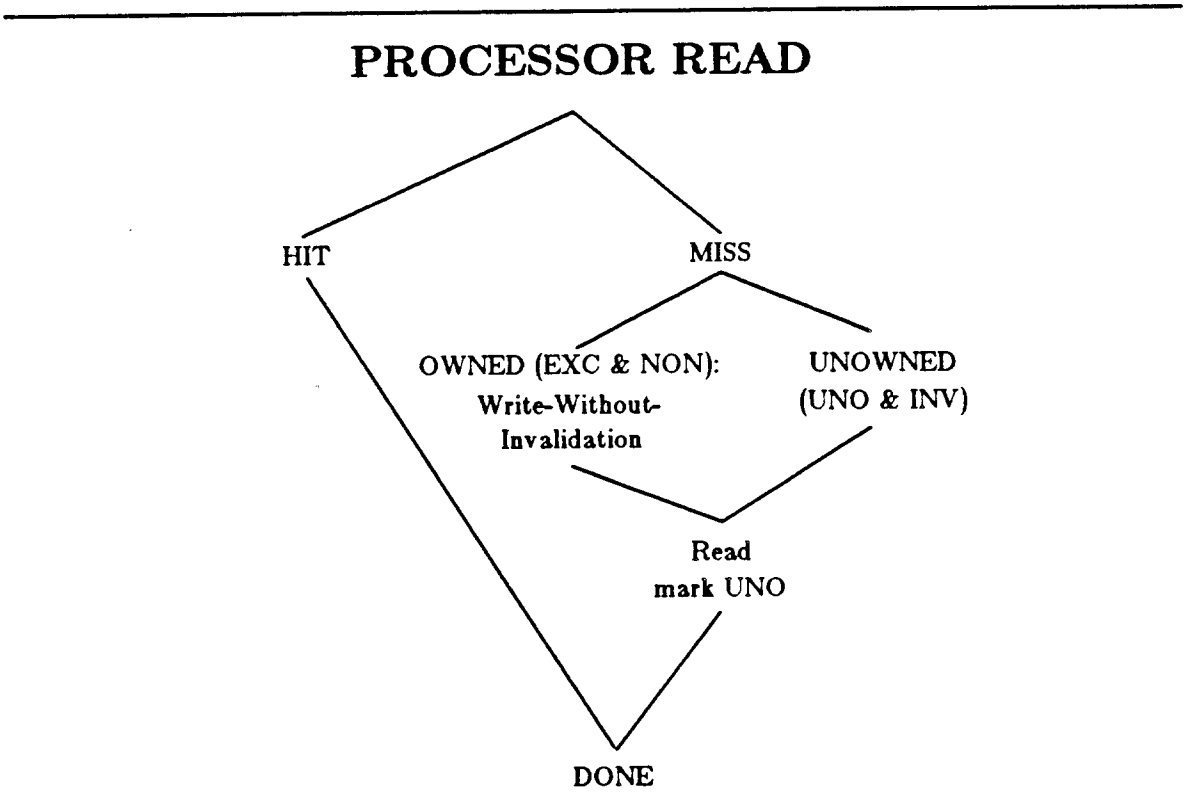


Figure 3.5 – The Cache Controller Handling A Processor Read Request

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

Read for the desired block and changes its state to **UnOwned.**

On a processor write to a block in the cache, a different procedure is followed, depending on the state of the block (see figure 3.6). If the entry hit is **Owned Exclusively,** then the processor can write to it without broadcasting on the bus. The Cache Controller must, however, obtain exclusive control of the cache from the Snoop before it does its writing. If the state of the hit block is **Owned NonExclusively** or **UnOwned,** then the Cache Controller must signal to the Snoops of other processors its intention to write before it modifies the block, so the other caches can invalidate their matching entries. An **Invalid** state indicates that the Snoop invalidated the

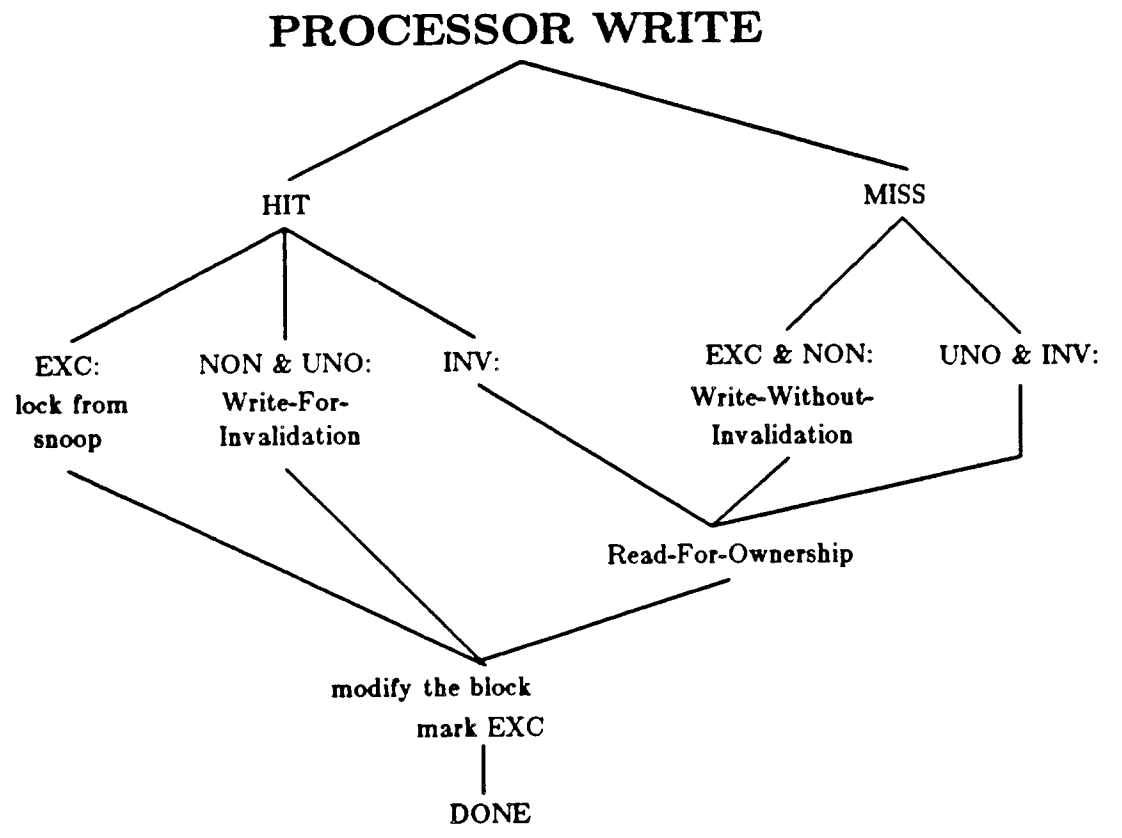


Figure 3.6 – The Cache Controller Handling A Processor Write Request

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

block in response to detecting a *Read-For-Ownership* or a *Write-For-Invalidation* from another processor, after the Cache Controller had recorded a hit. In this case the processor write is handled as though the Cache Controller had originally detected a cache miss. Once again, a block must be chosen for replacement on a miss. If the chosen block is owned, then it is written to memory using *Write-Without-Invalidation*. The requested block is then read with a *Read-For-Ownership* and is updated. Whether a hit or miss, and independent of initial state, the final state must be changed to **Owned Exclusively,** indicating that this is the only cached copy of the block, and that it is writable by this processor.

Some processor reads and writes do not cause bus activity. These occur when a processor reads a block already in its cache, or when it writes to a block it owns exclusively.

3.3.2. The Snoop Controller

The *Snoop Controller* monitors the bus for reads and writes from other processors. It accesses its cache only in response to some other processor's use of the blocks stored there, and only to invalidate a block written by another processor or to provide owned blocks to a requesting processor. In the latter case, it must also change the entry's state from **Owned Exclusively** to **Owned NonExclusively.** As in the case of the Cache Controller, the Snoop actions depend on the nature of the bus activity (read or write), whether it recorded a hit or miss on its cache, and the state of the block.

If the Snoop detects a read on the bus (see figure 3.7), it first determines whether the data being read is in its own cache. If not (a cache miss), no Snoop action is necessary. If the address tag comparison results in a hit, the Snoop's response depends on the type of read *(Read* or *Read-For-Ownership)* and the state of the block hit **(Owned Exclusively, Owned NonExclusively,** or **Unowned).**[4] If the block is owned, the Snoop must inhibit memory from responding to the bus read and instead provide the data to the requesting processor. For a block that is **Owned Exclusively**, the Snoop must first obtain sole use of the cache memory before responding, since

[4] A hit on a block marked Invalid is equivalent to a miss.

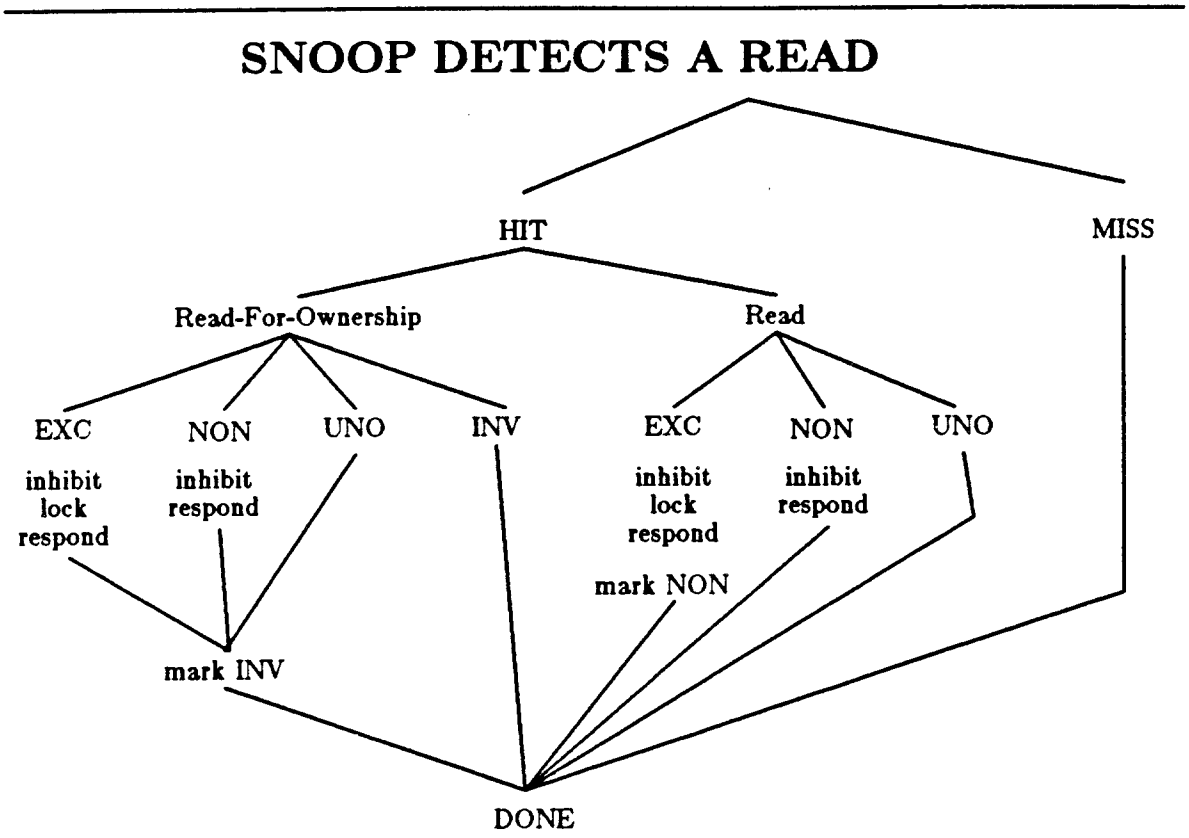


Figure 3.7 – The Snoop Controller Handling A System Bus Read Request

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

the Cache Controller may attempt to simultaneously update some entry. Under certain circumstances, the Snoop must change the state of the block. If the system bus request is a *Read-For-Ownership* that hits, then the Snoop must invalidate the state of its copy. A *Read* on the system bus that hits causes the Snoop to change that block's state to **Owned NonExclusively** if it was previously **Owned Exclusively.** For the other state and bus operation combinations, no action is necessary. The owning Snoop will respond to the bus request.

The Snoop actions for a bus write are simpler (see figure 3.8). On a cache hit, the Snoop only responds in the case of a *Write* or a *Write-For-Invalidation*. This means that some bus master is updating a copy of the block, and the Snoop must invalidate its own copy. If the write is a

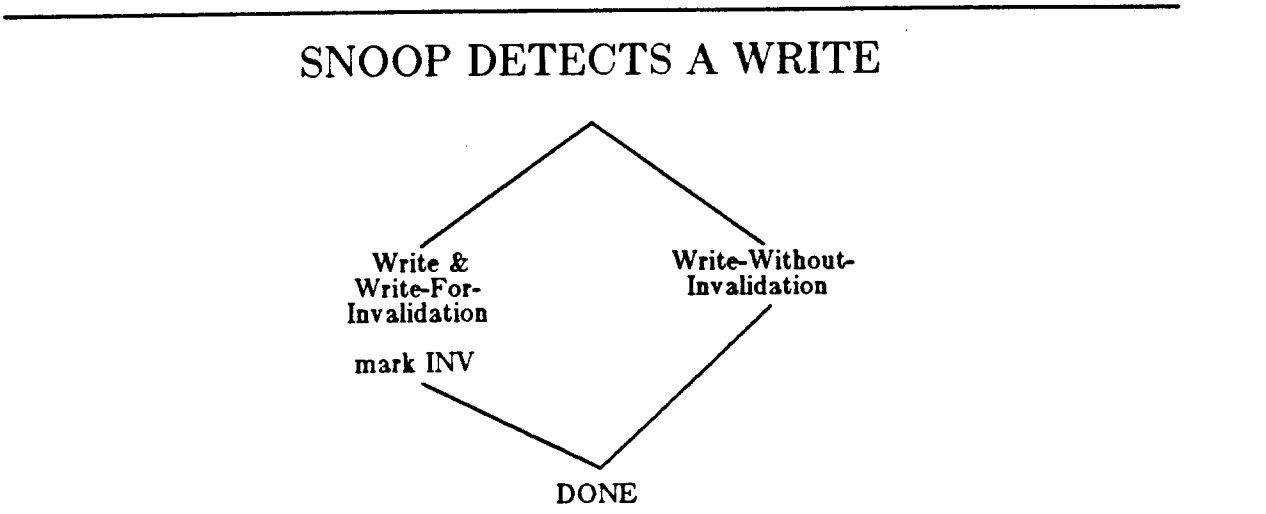


Figure 3.8 -- The Snoop Controller Handling A System Bus Write Request

INV: Invalid UNO: UnOwned EXC: Owned Exclusively NON: Owned NonExclusively

Write-Without-Invalidation, then another processor is flushing its cache, and the Snoop need do nothing. This completes the detailed description of the protocol.

3.4. Comparison and Evaluation

This section describes the trade-offs between several possible cache consistency protocols and motivates some of the choices we have made. We will present a qualitative comparison between Goodman's write-first protocol [GOOD83, RAVI83] and the Berkeley ownership protocol. Write-through [AGRA77] was immediately rejected because it will always generate more bus traffic than the other two approaches. The adequacy of a given protocol depends on the expected workload; therefore we examine the protocols under varying degrees of sharing: (1) no sharing, (2) read-only sharing, and (3) arbitrary read-write sharing.

Both the ownership protocol and Goodman's write-first protocol are based on *copy-back.* Dirty blocks are retained in the cache as long as possible, and main memory is updated when the block is forced out of the cache. [NORT82] has shown that multiple writes to the same block

occur frequently. Thus copy-back, which generates a single memory write, will create less memory traffic than write-through, which generates a memory write for each cache write.

Unless the processor can explicitly request *Read* or *Read-With-Ownership,* the cache must predict whether a block should be read with ownership because it will be updated later. The cost of guessing wrong, i.e., not acquiring ownership when a block is read for later update, is an extra read operation to acquire ownership at the time of the write. Table 3.3 shows that ownership requires less bbus transactions than write-first if the prediction is always correct, and more memory operations than write-first if the prediction is never correct. In practice, the actual prediction rate will fall somewhere between the two extremes.

Two possible static prediction algorithms are possible: always fetch with ownership, or always fetch (first) without ownership. Table 3.4 illustrates the behavior of these two strategies.

Table 3.3 -- Comparison of Ownership with Write-First

	Ownership (correct prediction)	Ownership (incorrect prediction)	Write-First
Read	Read (UnOwned)	Read-For-Ownership Write-Without-Invalidation	Read
Single Write	Read-For-Ownership Write-Without-Invalidation	Read (UnOwned) Write-With-Invalidation Write-Without-Invalidation	Read First write
Multiple Writes	Read-For-Ownership Write-Without-Invalidation	Read (UnOwned) Write-With-Invalidation Write-Without-Invalidation	Read First write Write on flush

Table 3.4 -- Comparison of Static Prediction Strategies

	Ownership (Read UnOwned)	Ownership (Read OwnedExclusively)	Write-First
Read	Read (UnOwned)	Read-For-Ownership Write-Without-Invalidation	Read
Single Write	Read (UnOwned) Write-With-Invalidation Write-Without-Invalidation	Read-For-Ownership Write-Without-Invalidation	Read First write
Multiple Writes	Read (UnOwned) Write-With-Invalidation Write-Without-Invalidation	Read-For-Ownership Write-Without-Invalidation	Read First write Write on flush

With no sharing, always fetching with ownership is roughly equivalent to write-first. The actual performance depends on the percentage of blocks that are read-only, written only once, and written more than once. Recall that owned blocks are written back to memory when they are replaced, whether or not they have been updated. The protocol does not recognize dirty blocks, since it assumes that blocks are acquired with ownership only when they are to be updated. Unfortunately, this generates unnecessary memory writes for read-only blocks that have been acquired with ownership. This is corrected by a simple extension of the protocol. The state is extended with a dirty bit, which must be transmitted with the block when ownership is transferred to another cache. Only owned blocks that are actually dirty are written back to memory. With this modification, ownership would always require fewer bus transactions than write-first.

When there is some sharing, always fetching with ownership is not a good strategy. *Read-for-Ownership* defeats read sharing by causing other cached copies of the requested block to be invalidated. Since the Multics results [MONT77] indicate that most sharing is read-sharing, this is a serious shortcoming.

The performance of "fetch without ownership" is also roughly equivalent to write-first. Again the actual performance depends on the reference pattern to the blocks in the cache, as well as the difference in cost between the *Write-For-Invalidation* and a conventional *Write*. This strategy and write-first both support read-sharing very efficiently, allowing multiple copies to proliferate throughout the caches. When the fetch without ownership strategy is employed, the ownership protocol degenerates to a slightly different version of write-first. An abbreviated write is used to signal the invalidation, rather than a full write, and it is always necessary to flush dirtied blocks.

It is obvious that the static prediction strategies are insufficient to fully utilize the power of the ownership protocol. The compiler, which has much more information about how variables will be used, could pass hints about block usage to the cache. While no analysis of such prediction

strategies has been made, we expect them to lie closer to 100% correct prediction than to 0%. In the following, we will assume nearly 100% correct predication.

As was seen in Table 3.3, ownership with 100% correct prediction is an optimal algorithm, making no unnecessary bus transfers. This holds in the case where no sharing exists, and also where there is read-only sharing. Let us now examine the read/write sharing case.

Suppose we have a block that is being continually read and updated, as might well be the case for a test-and-set lock. In the ownership protocol, this block is acquired with *Read-for-Ownership* and is modified in the cache. If the block is forced out of the cache and is flushed to memory, then the number of bus transactions is the same as if there is no sharing. However, requests for this block occur frequently, and it will likely pass to another cache before being forced out. Thus, updating the block only requires one bus transaction: to read it into the cache initially. In the write-first protocol, the block is read, modified in the cache, and then written to memory to invalidate other copies. Since two bus transactions are required to perform the update, write-first incurs twice the overhead of ownership for read/write sharing.

In summary, the performance of ownership is highly dependent on the cache's ability to correctly predict the usage of a block. The degree to which it is possible to make correct predictions is still an open research question. Assuming that correct predictions are possible, we have shown that ownership incurs less overhead for all cases, especially when sharing is frequent.

4. Chip Subsystems and Their Interactions

In this section, we discuss the function of each subsystem and how it interacts with the other subsystems of the chip.

4.1. Overview

The snooping data cache system consists of five distinct subsystems (see figure 4.1): a *Data Cache Memory*, a (processor) *Cache Controller*, a *Snoop Controller*, a *Processor Bus Interface*, and a *System Bus Interface*. Each is described briefly below, and in more detail in the following

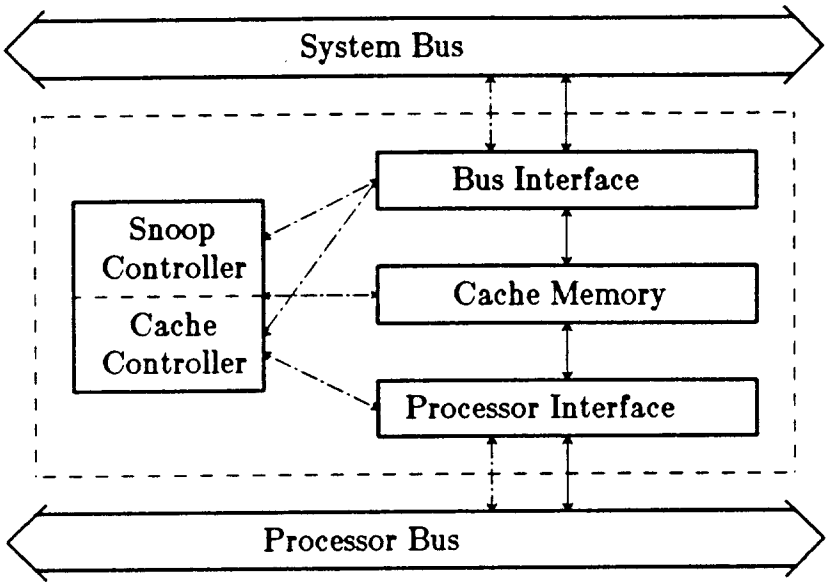


Figure 4.1 – A Block Diagram of the Data Cache Chip

The data cache chip is partitioned into the subsystems illustrated above. Data lines are solid arcs; control lines are broken. Data transfer between the cache memory and the processor is performed by the Processor Bus Interface subsystem. Data transfer between the cache memory and the system bus is performed by the System Bus Interface subsystem. The Cache Controller handles the actual processor reads and writes on the cache. The Snoop Controller watches the system bus and implements much of the cache consistency protocol.

subsections.

The *Cache Memory* is organized as a direct mapped array of 64-bit cache blocks with associated state and tag bits. This organization was chosen for its simplicity. Additional circuitry includes tag match logic to determine cache hits, and assembly registers to map 16-bit external data to and from 64-bit internal data blocks.

The *Cache Controller* accepts read or write requests from the processor through the Processor Bus Interface. It queries the cache memory to determine if the requested memory block is available locally. If not, it interacts with the System Bus Interface to read it from main memory into the cache. If the processor operation is a read, the Cache Controller delivers the selected word or byte to the Processor Bus Interface. If it is a write, the word or byte provided by the

Processor Bus Interface replaces the corresponding portion of the block in the data cache memory. The Cache Controller also assists in implementing the cache consistency protocol by modifying the state bits of cache entries in the appropriate way.

The *Snoop Controller* interfaces to the system bus, monitors bus traffic, and implements most of the details of the cache consistency protocol. An external read or write request triggers the snoop to initiate a lookup in the cache. If a match is found, state bits may be changed and/or the cache block may be written out to the system bus, according to the protocol specifics described earlier.

The *Processor Bus Interface* implements the handshake with the 68000 processor bus. It insulates the Cache Controller from the details of interfacing to the processor bus. Thus, it should be possible to change the processor bus without affecting the Cache Controller design.

The *System Bus Interface's* function is very much like that of the Processor Bus Interface: the details of communicating with the system bus are localized. Both the Cache Controller and the Snoop Controller must interface with the system bus.

4.3. The Cache Memory Subsystem

The chip's datapath is the cache memory subsystem (see figure 4.2). The cache itself is organized into sixteen entries, each entry containing a sixty-four bit data block (4 x 16-bit words), two state bits, and thirteen tag bits. The state bits can be modified independently of the data and tag portions of the cache. The datapath contains two sets of decode circuitry (Adec and Bdec). On a read, these can be driven independently, but on a write they are driven by the same inputs.

Data is read from and written to the cache from two 64-bit assembly registers (Aassembly and Bassembly). On a read, these registers deliver a 16-bit portion of the data to the system bus or an 8- or 16-bit portion to the processor bus. On a write, they assemble four 16-bit words from either the processor or the system bus into one 64-bit block and then transfer it to the cache. The

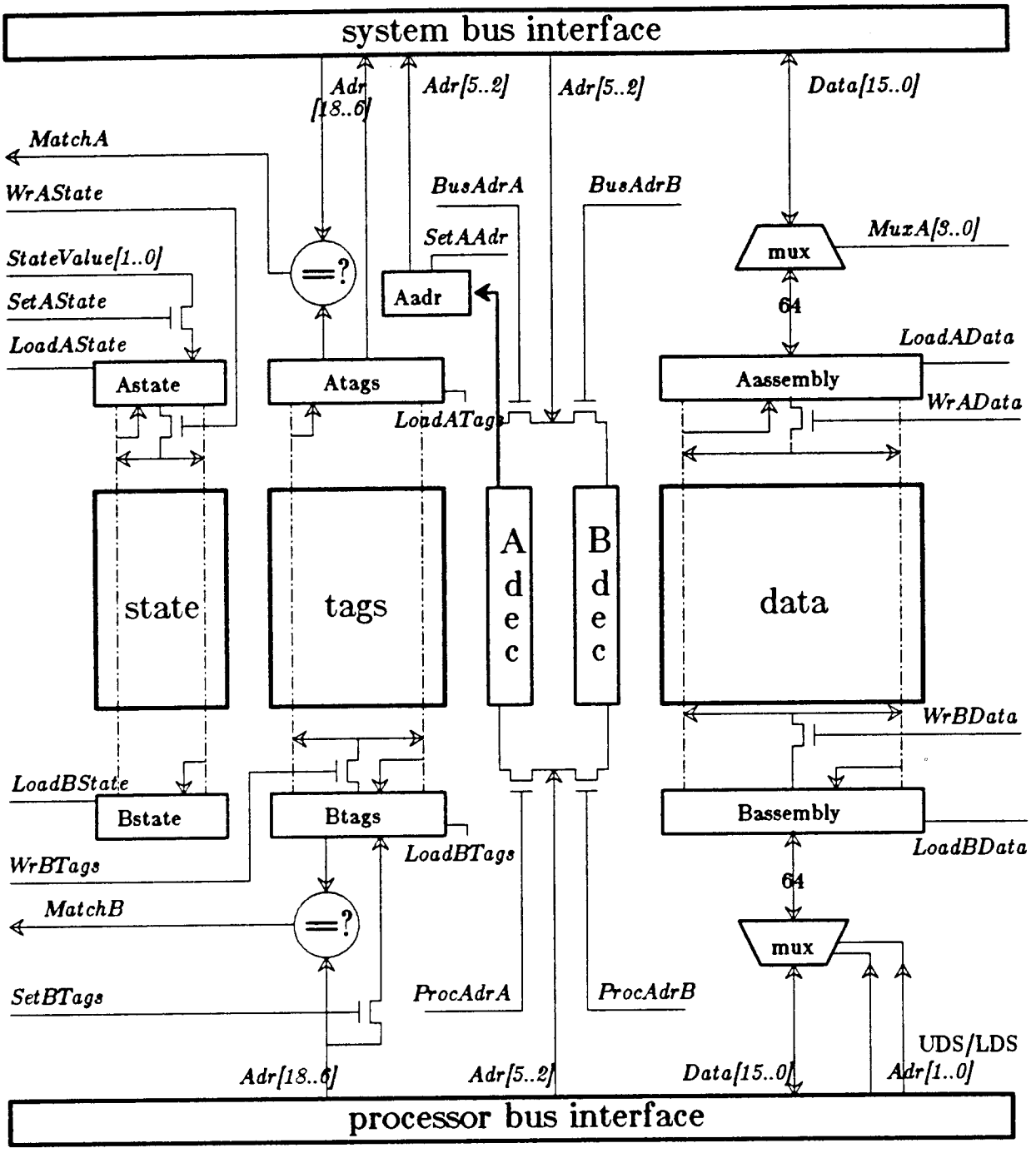


Figure 4.2 – The Datapath

This diagram illustrates the datapath assumed by the Snoop and Cache Controllers.

Processor Bus Interface is directly connected to the Bassembly register, while the System Bus

Interface is directly connected to the Aassembly register. This organization simplifies dataflow and pad assignment, but introduces some complexity in control routing. Although the Snoop Controller only accesses the system bus side of the datapath, the Cache Controller needs access to both.

The 68000 provides a word address, of which nineteen bits are decoded. The low order two bits (*Adr[2..1]*) select a word from within the four word memory block. These address lines control sixteen 4:1 multiplexors/demultiplexors on each side of the data portion of the cache memory. The next highest four bits (*Adr[6..3]*) select one of the sixteen entries of the cache, and are input to the decoders. The highest order thirteen bits (*Adr[19..7]*) are compared with the tag portion of the selected entry. If they match, and if the state is not **Invalid**, then the addressed word is *hit* in the cache and the appropriate match line is asserted (*MatchA* or *MatchB*); otherwise it is a *miss*.

The datapath is accessed via two mutually exclusive read and write cycles. Each cycle employs a standard two-phase clock. The cycles are distinguished by a special *WriteCycle* signal, asserted to initiate a write cycle. The clock phases operate in an identical fashion for both cycles. During the first phase, *Phi1*, the decoders and word lines are driven. During the second phase, *Phi2*, the bit-lines are driven either by the cache cells or the assembly registers. The datapath timing influences the timing of the entire chip. The simple two phase clock, derived from the processor's clock, is used throughout. Every subsystem obeys the standard convention of inputs sampled on *Phi1*, outputs valid on *Phi2*.

4.3. The Cache and Snoop Controllers

The signals used by the controllers to access the memory subsystem are also shown in figure 4.2. To begin a processor (system bus) read request, the cache (snoop) controller initiates address decode by asserting *ProcAdrB (BusAdrA)* on *Phi1*. The cache (snoop) controller reads into the B (A) registers by asserting the *LoadBData, LoadBTags,* and *LoadBState (LoadAData, LoadATags, LoadAState)* signals on *Phi2*. When writing the data and tag portions of the cache, the cache

controller asserts both decode signals (*ProcAAdr* and *ProcBAdr*), and writes with *WrBData* and *WrBTags*. Note that the snoop never updates these portions of the cache. However, either controller can change the state. This requires an extra phase to set (*SetAState*) the new state value (*StateValue*) into Astate.

4.4. Interlocks Between the Controllers

Because the Snoop and Cache Controllers operate independently, they contend for access to the cache memory. Possible sources of contention are classified as read-read, read-write, and write-write. The datapath has been implemented with a dual-ported read capability to eliminate read-read contention. Since the datapath is single-ported for write, only one controller can be active if it desires to write into the datapath. Read-write contention is eliminated by requiring the controllers to adhere to a separate read cycle/write cycle protocol for accessing the cache memory. Lastly, write-write contention is eliminated by an interlock mechanism described later in this subsection.

To illustrate how read-write contention could arise, consider the case where the processor issues a read request that hits in the cache, and simultaneously, the Snoop detects a *Write-For-Invalidation* for the same block. The Snoop needs to change the state to **Invalid,** while the Cache Controller needs to read from the cache. Because reads and writes occur on different cache cycles, the Snoop invalidates the entry only after the Cache Controller has read it. While a consistent view of memory is maintained by the separate cycles, higher level software, implementing application level locking, is responsible for insuring that one processor cannot read data being updated by another processor.

Write-write conflicts occur when both controllers need to update the cache at the same time. For example, the Snoop may need to invalidate an entry while the Cache Controller steals ownership by changing an entry's state from **UnOwned** to **Owned Exclusively.** Besides these *intra-cache* write conflicts, there are other *inter-cache* update anomalies, as figure 4.3 shows. Therefore, interlocks within and among caches are needed.

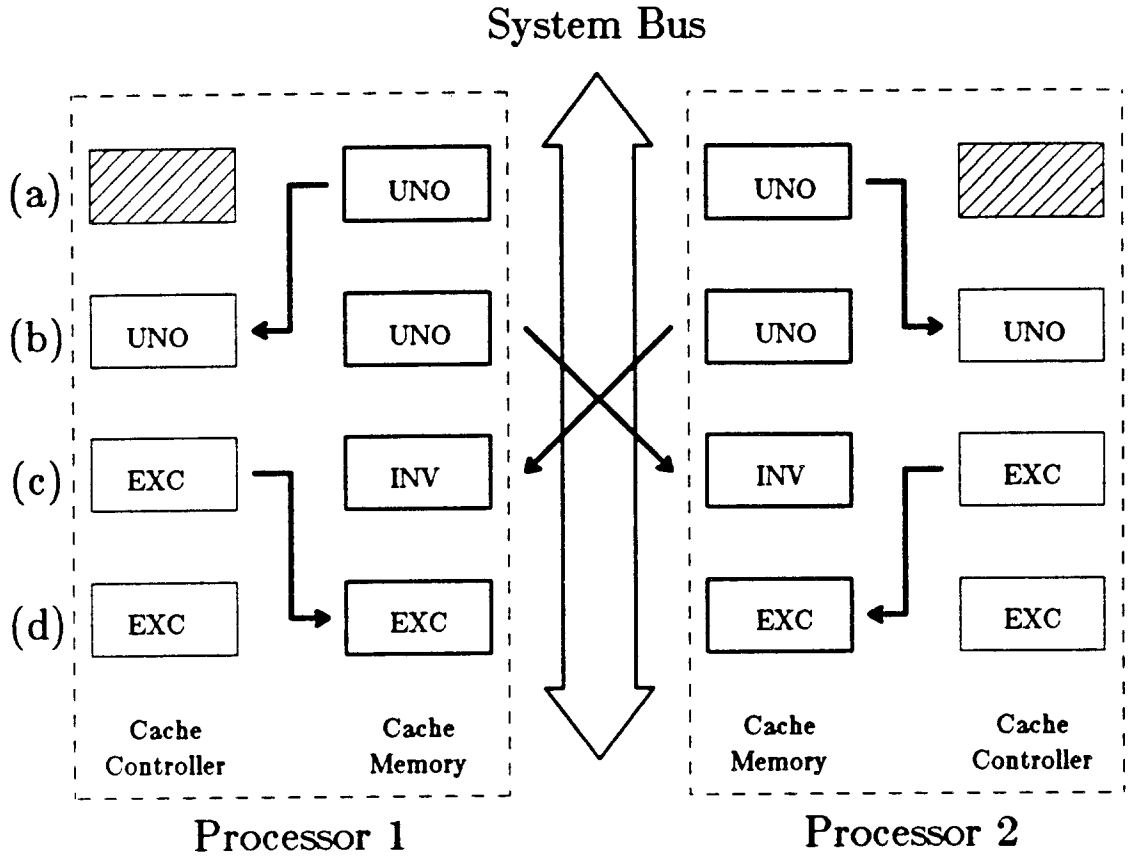


Figure 4.3 – Example of an Update Anomaly

In (a), processors 1 and 2 each have **UnOwned** copies of the same memory block to which they wish to write. Under the Berkeley protocol, they must obtain ownership by issuing a *Write-For-Invalidation* before the writes can be performed. In (b), the Cache Controllers simultaneously read and discover that the entries are **UnOwned.** In (c), they update the state to **Owned Exclusively,** and (in sequence) send *Write-For-Invalidation* requests to the system bus. The respective Snoops invalidate the entries in their caches. In (d), the controllers now update their caches, which they still believe to be **Owned Exclusively.** An inconsistent state has been reached.

Two interlock schemes are used to prevent write-write conflicts. In the first, the system bus is used as a semaphore. Before the Cache Controller can update the cache, it must acquire the system bus. Once it is in control of the bus, no other cache can generate requests on the bus, and therefore its Snoop is inactive. The bus is released only after the update is complete.

Using the system bus in this way has performance implications. Once a Cache Controller has obtained the bus, other Cache Controllers are prevented from using it until it has been

released. This is acceptable as long as the first Cache Controller either requires a data transfer to satisfy its processor write request or must signal a *Write-For-Invalidation* to other processors. Neither is true for updates to **Owned Exclusively** blocks, since such writes can be done locally. Figure 4.4 shows how a conflict can arise when a processor writes to an **Owned Exclusively** block, and there is an external read request for the same block.

An additional interlock mechanism is needed for writes to **Owned Exclusively** blocks (see figure 4.5). We expect write sharing to be infrequent: a processor is likely to write to its **Owned Exclusively** blocks more frequently than they will be read by other caches. The interlock has been designed to be asymmetric on purpose. The Cache Controller frequently wins ties because it

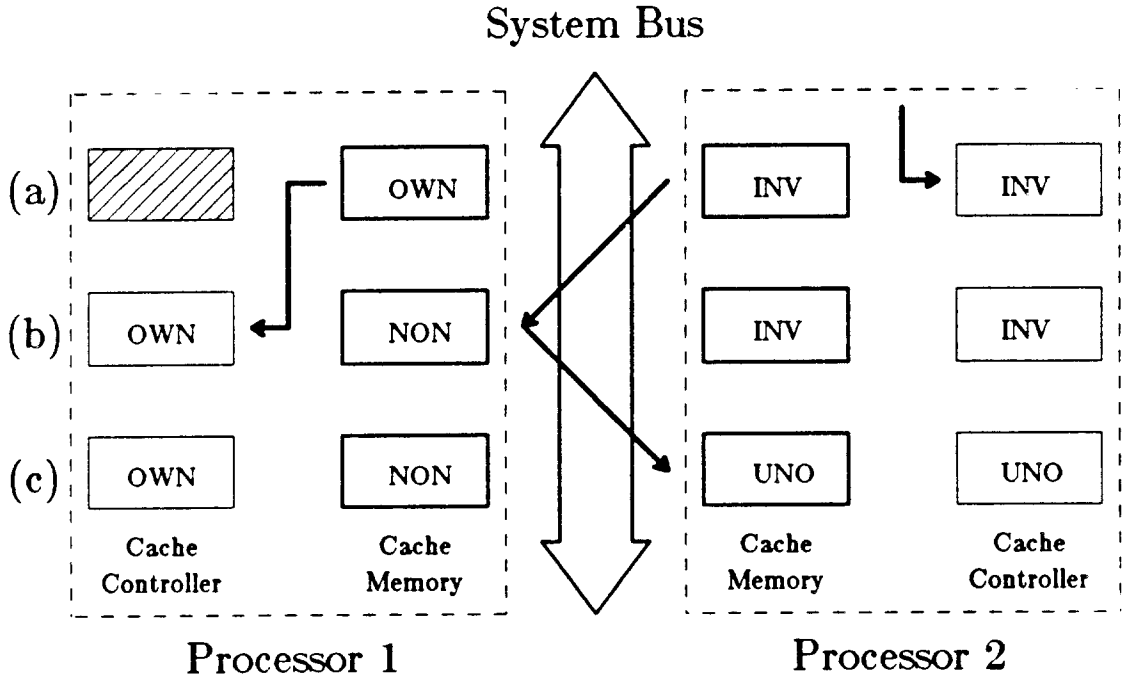


Figure 4.4 – Conflicts with Blocks that are Owned Exclusively

In (a), processor 2's Cache Controller tries to read a block in its cache and misses. Its Cache Controller then issues a *Read* for the block. In (b), processor 1 accesses the same block and discovers that it has a state of **Owned Exclusively**. It therefore believes it can update the entry. Simultaneously, its Snoop responds to processor 2's *Read* request, placing the block's data on the system bus and changing its entry to **Owned NonExclusively**. In (c), the cache controller for processor 2 changes the state of its copy to **UnOwned**. The Cache Controller of processor 1 updates its copy, causing an inconsistent state.

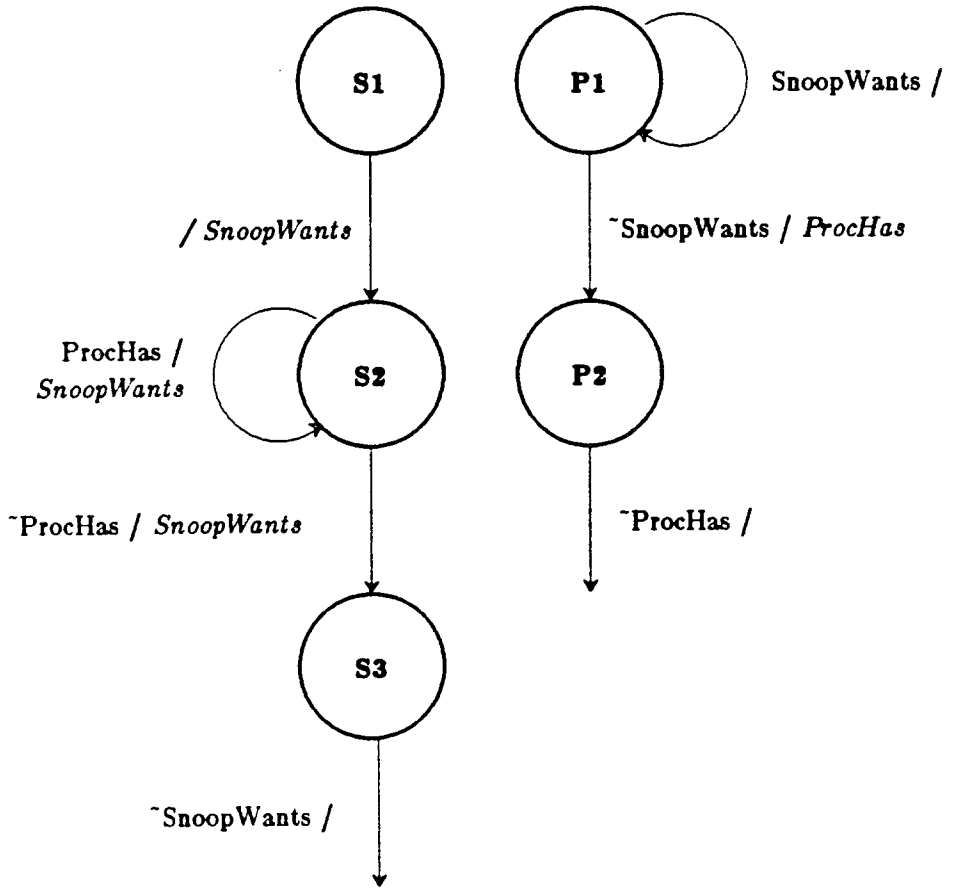


Figure 4.5 – Asymmetric Interlock Scheme

The state diagram above shows how the processor (P) and the snoop (S) should act to avoid entering their crticial sections at the same time. This interlock is only needed when an entry must be updated but it is **Owned Exclusively,** since otherwise the processor would have first waited to get the system bus, thus effectively locking out the snoop. The ProcHas line can be asserted by the processor right away, as long as SnoopWants is not already asserted, and the processor is safe while in state P2. The SnoopWants line can be asserted anytime, but the snoop must wait a cycle and check ProcHas before entering its safe state, S3. Note that the processor is purposely given the upper hand, since an ordering where the snoop wins first causes some wasted processor actions (such as rereading the data).

can obtain the lock in the same cycle it is needed, while the Snoop must request it a full cycle in advance. Thus, there is a one cycle penalty whenever another cache accesses an **Owned Exclusively** block. Once the state has been changed to **Owned NonExclusively** by an external access, the owning cache will no longer need to use the interlock.

4.5. The 68000 and The Processor Bus Interface

The 68000 bus signals can be organized into three functional groups: data, address and control (see figure 4.6). Word addresses are available on a 23-bit unidirectional bus, of which 19 bits are decoded by the cache chip. The effective address space is 1 MByte. Data is transferred between the 68000 and the cache chip via a 16-bit bidirectional data bus. When the processor asserts ~AS, indicating that there is a valid address on the address bus, a high signal on the R/~W line will signal a read from the cache onto the data bus. When R/~W line is low, the processor is writing to the cache. The upper and lower data strobes (~UDS, ~LDS) control the data on the data bus. Byte oriented instructions are signaled by asserting either ~UDS or ~LDS. The appropriate byte should be gated onto the data bus by the data cache or the processor, depending on whether a read or write was requested. Word addressing is supported by asserting both ~UDS and ~LDS.

The handshake between the processor and the cache chip follows a four cycle signaling scheme. After the processor has placed a valid address on the address lines, it asserts ~AS" low.

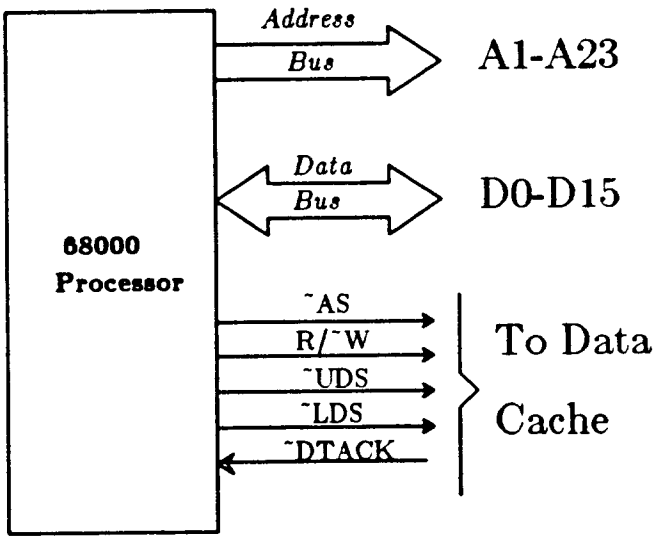


Figure 4.6 – 68000 Processor Bus Signals

If a read, the cache chip provides a word on the data lines, and when valid, asserts ~DTACK low. On a write, the cache chip updates the word or byte in cache, and then asserts ~DTACK. When the 68000 detects that ~DTACK is low, it removes its request. The cache chip resets the handshake by driving ~DTACK high.

The Processor Bus Interface lies between the processor bus and the cache memory, and is controlled by the Cache Controller. It should satisfy a processor read request that hits in the cache with as few waitstates as possible. In addition, it maps the 68000 bus signals into a smaller, simpler set of signals for the Cache Controller. Finally, it plays a major role in the implementation of the atomic *Read-And-Set* operation.

The Processor Bus Interface consists of a three state finite state machine, a buffered "transceiver" on the 68000 data bus, and some random logic. Figure 4.7 illustrates this module and its interface to the cache controller.

The default condition of the Cache Controller is to read a block whenever possible. Thus, if the B decoder is free (i.e., if *BusAdrB* and there not been raised by the Snoop), and there are no pending processor requests, then *AutoRead* is raised and a block is read. The Processor Bus Interface monitors the processor request lines, ~UDS, ~LDS, and ~AS. When one of these lines drops, a delay circuit is activated that detects when the cache has actually performed a read. The read takes place while *Phi2* and *AutoRead* are both high, and the detection of a hit is determined on the following *Phi1*. If the read hits, the data is latched into ProcDataOut during *Phi1*, SendDataOut is raised, and ~DTACK is lowered. SendDataOut and ~DTACK are asserted until the 68000 removes its request.

In the case of a miss, the *ProcReadReq* signal is raised to invoke the Cache Controller. The finite state machine goes into the Execute state, and waits until the Cache Controller has performed the necessary operations for a read. When the Cache Controller raises *ProcAck*, the finite state machine moves into the Complete state, the data is latched into ProcDataOut, *SendDataOut* is raised, and ~DTACK is lowered. When the 68000 removes the request, the finite state

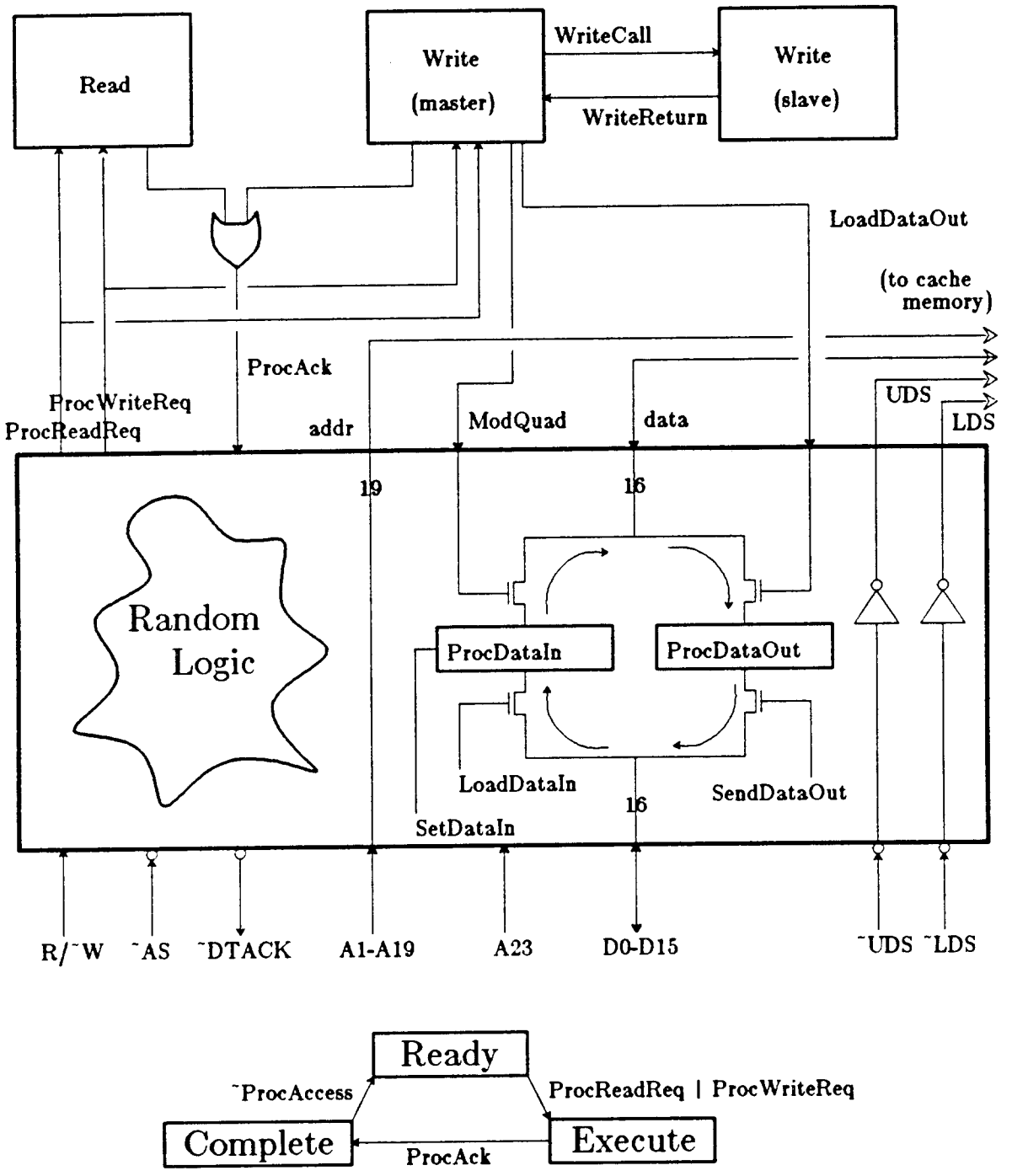


Figure 4.7 – Interaction Between the Processor Interface (bottom) and the Cache Controller (top)

machine moves into the Ready state again, and all the signals are unasserted. The third state is

necessary to prevent the false detection of another request.

Processor writes, whether they hit or miss, are handled by the Cache Controller. Upon receipt of a write-request, the Processor Bus Interface raises the *ProcWriteReq* signal, transferring control to the Cache Controller. The Cache Controller signals completion of the operation by raising the *ProcAck* signal. Notice that the interface to the Cache Controller is much simpler than the 68000 interface. This reduces the number of lines that must be routed, as well as making the Cache Controller PLAs smaller.

The Processor Bus Interface also supports an atomic bus transaction. Due to the design of the 68000, it is not possible to detect that the processor is performing a test-and-set instruction *before* the read request is made. This makes it difficult to implement the operation atomically. To simplify the problem, we have implemented an atomic *Read-And-Set* operation within the cache. The semantics of the operation are to read the contents of a word and set the word to all ones, atomically. The program can now examine the old contents of the word; if it was zero, then the process succeeded in obtaining the lock, otherwise it failed and must try again later. The program gives up the lock by writing zero back to the word.

The *Read-And-Set* operation is signaled by the 68000 by issuing a read operation on an address with the high order address bit high.[5] Since the MultiBus only supports a 20-bit byte address space, allocating the high order bit does not reduce the address space. The Processor Bus Interface detects the *Read-and-Set,* and passes control to the Cache Controller by raising both the *ProcReadReq* and *ProcWriteReq* signals. In addition, the Processor Bus Interface asserts the *Set-DataIn* signal, which forces the ProcDataIn buffer to be all ones. Under control of the Cache Controller, the word is read into ProcDataOut, and then set with the contents of ProcDataIn. The normal handshake is then completed with the 68000.

[5] Dedicated address bits could also have been used to distinguish among the different reads and writes of the cache consistency protocol.

4.6. MultiBus and System Bus Interface

The Intel MultiBus is an asynchronous bus with a 20-bit address bus and a 16-bit data bus. The bus signals can be grouped into five functional groups, as shown in figure 4.8: address lines that can address 1Mbyte; data lines for 16-bit word transfers; control lines that implement memory or I/O read and write operations, a memory inhibit line and an acknowledge line; bus exchange lines that include bus clock, priority resolution, bus request, and bus acquisition lines; and miscellaneous clock, interrupt, and initialization lines. Our cache consistency protocol must distinguish between two kinds of reads and two kinds of writes.[6] To have this capability, we have added an additional line to the control lines group of the MultiBus.

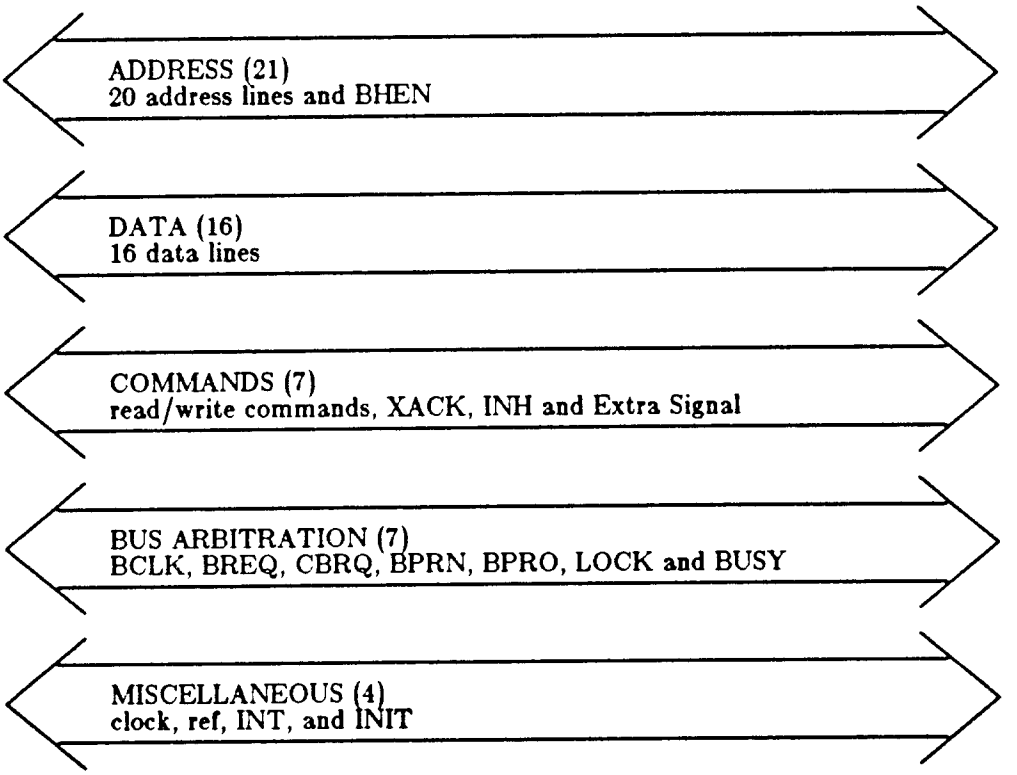


Figure 4.8 – MultiBus Summary

[6] *Write* and *Write-For-Invalidation* use the same system bus signaling conventions.

There are two types of devices on the MultiBus: masters and slaves. Masters can control the MultiBus and initiate data transactions. Slaves can only respond to the requests of the masters. Since the MultiBus is an asynchronous bus, the command and acknowledge lines employ a four-cycle handshake to effect a data transfer. However, the bus exchange and arbitration lines are all synchronous to the 10MHz bus clock to guarantee the proper priority resolution. There can only be one master on the bus at any one time, and only one transaction can be taking place at any given time. This last aspect of the MultiBus allows us to use it as a system wide semaphore.

When a master needs to acquire control of the bus, it asserts its bus request lines. If the bus is idle, it can immediately take control and proceed with its transactions. If not, it must first wait for the current master to relinquish control and for priority to be resolved among requesting masters. Once a master has control of the bus, no other device will be able to use it.

Both the Snoop Controller and the Cache Controller are interfaced through the System Bus Interface to the MultiBus. In keeping with our rapid prototype strategy, we employed a MultiBus Design Frame [BORR84] to accomplish these interfacing tasks. A MultiBus Design Frame is a collection of circuitry that interfaces directly to the MultiBus and provides a greatly simplified interface to the circuitry on the inside. The idiosyncratic details of MultiBus data transfer, set-up times, and bus exchange protocols are hidden from the circuitry. It only needs to deal with the finite-state-machine-like interface of the frame. The frame also concerns itself with synchronization issues between the system bus and the cache clock.

The System Bus Interface generates signals that are high for a single cycle. However, the Snoop requires that some signals stay high during the entire transaction. For example, the System Bus Interface generates a one cycle *SlaveRead* signal, whenever there is a memory read operation on the bus. The Snoop requires the signal to remain high until it has had a chance to examine it. Set/reset flip-flops are used to hold the signal for the Snoop. They have no effect on the System Bus Interface's functionality.

From the point of view of the Snoop Controller, the System Bus Interface performs three functions. First, it must signal the start of any external transaction on the system bus. Secondly, it must inhibit main memory from responding to a read request, should the Snoop determine that the requested block is owned by its cache. Finally, it must be able to supply the requested block, located in an assembly register, in response to an inhibited read request. From the point of view of the Cache Controller, the interface must be able to obtain and release control of the bus, and to read (write) a block of memory into (from) the assembly register. The signals used for communication between the controllers and the interface are shown in figure 4.9.

The Cache Controller is a MultiBus master. To accomplish a memory read or write the Cache Controller simply exerts a combination of the *MasterRead, MasterWrite* and *MasterSpecial* signals for a single cycle and then waits for a one cycle signal on *MasterAck* to signify the end of the transaction. A conventional *Read* is initiated by raising only the *MasterRead* signal. If *MasterSpecial* is also asserted, then a *Read-For-Ownership* operation is initiated, causing other caches to invalidate their copies of the block. To flush a block to main memory, the Cache Controller raises the *MasterWrite* and *MasterSpecial* signals (*Write-Without-Invalidation.*). If only the *MasterWrite* signal is raised, then other caches will invalidate their copies of the block (*Write-For-Invalidation*). Note that the cache never generates a *Write* operation; these are reserved for I/O devices and other bus masters without caches. The MultiBus address and data to be written need to be valid while the *MasterRead* or *MasterWrite* signals are high, and data being read will be valid as soon as *MasterAck* goes high. The design frame circuitry is responsible for obtaining the bus and performing the data transfer.

Since the Snoop is a MultiBus slave, it should not be activated if its local cache is the current bus master (indicated by *HaveBus* asserted). Therefore, a *SlaveRead* or *SlaveWrite* is only signaled to the Snoop if another cache has control of the bus (refer to figure 4.9). The address lines to the Snoop will be valid by the time the *SlaveRead* or *SlaveWrite* signals go high. The dashed line in the figure indicates a reasonable boundary between the System Bus Interface

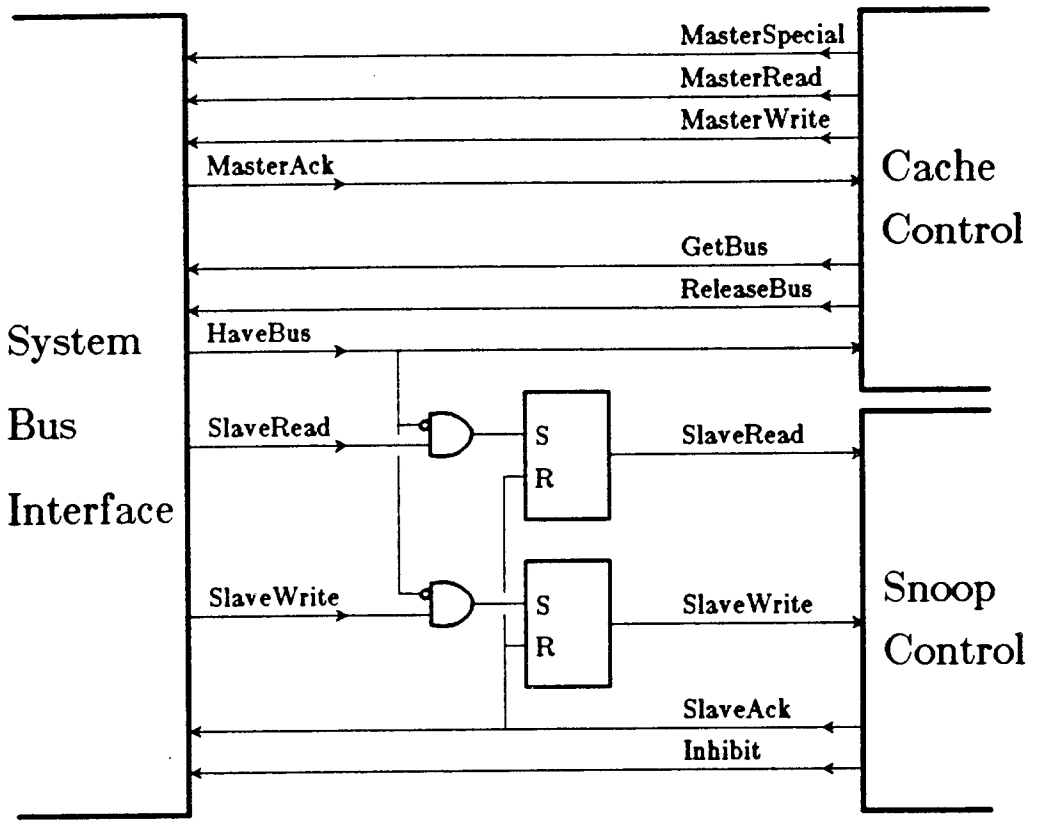


Figure 4.9 - Interaction between the Controllers and the System Bus Interface

This diagram illustrates the signals used for communication between the System Bus Interface and Cache and Snoop Controllers. The *Master* signals are used by the Cache Controller to initiate a read/write operation on the system bus (MultiBus). The *Slave* signals are used by the System Bus Interface to alert the Snoop to the presence of a read/write operation on the system bus. The System Bus Interface ignores the *SlaveAck* signal if *Inhibit* has not been raised. Every signal generated by these subsystems (with the exception of *HaveBus*) is high for a single cycle. The set/reset flip-flops are required since the Snoop may be recovering from an earlier operation when the next one begins.

and the Snoop.

The Snoop must generate *SlaveAck* for every bus transaction, even those that are to be ignored (e.g., *Write-Without-Invalidation*), to insure that the System Bus Interface continues to function in the proper manner. This is a requirement of the MultiBus Design Frame, which expects to be able to answer the command with an acknowledge. However, the Design Frame has been modified to repeat the acknowledgement on the MultiBus only if the *Inhibit* line is also high,

signifying that the Snoop was actually going to handle the bus transaction. The *Inhibit* signal, once raised by the Snoop, is latched by the System Bus Interface. It is lowered after the external read request has been satisfied.

Due to the difference between the MultiBus's 16-bit data bus and the cache's 64-bit blocks, the System Bus Interface must provide for the transfer of an entire block atomically. This block transfer is accomplished through the *GetBus* and *ReleaseBus* signals. When the Cache Controller needs to do a block transfer, it asserts *GetBus* along with the first *MasterRead* or *MasterWrite*. Once the System Bus Interface has obtained the bus, it responds with the *HaveBus* signal, which is asserted throughout the transfer. The Cache Controller then proceeds to perform the four transfers required in the normal fashion. When they are completed, the Cache Controller asserts *ReleaseBus* to relinquish control of the bus.

5. Implementation

In this section, we will describe the detailed implementation of the different functional modules of the design: the Datapath and the Snoop and Cache Controllers.

5.1. The Technology

We have chosen CMOS as our implementation technology. It is the preferred future technology because of its potential for high speed at low power. Future implementation services available to us will support a 2.0 micron N-Well process and a more aggressive 1.25 micron N-Well process, although only 3.0 micron P-Well technology is currently supported. While we are more experienced in NMOS design, the rapid prototype in CMOS gave us the opportunity to test our automated design tools on CMOS (especially the PLA generation tools and design rule checkers) and to gain design experience with the technology.

5.2. Datapath

While the organization of the cache, e.g., its set associativity, number of words per entry, number of entries in the cache, etc., can have a significant affect on performance, we have

assumed a simple organization. Throughout the project, emphasis has been placed on the design and implementation of the controllers and their interlocks, rather than an aggressive memory organization. A directed mapped organization was selected because it avoids the complexities and extra silicon area required by associative circuitry. The size of the cache was chosen to be 16 entries of 64 bits each, a small, but non-trivial amount of memory on-chip. This allows the memory array and the control circuitry to fit on a single chip. The entire memory array is 79 bits wide: 64 bits of data, 13 bits of tag address, and 2 bits of state.

The memory cell was designed to be small and fast. We considered two alternative designs: the CMOS analogs of the cells used in the RISC I and RISC II [SHER82] processors. The first cell (RISC I) contains two read ports and one write port. The cell is written by (1) breaking a feedback loop between the input and the output of the cell, and (2) driving the input. This cell is easy to write, but requires extra control lines running through the array to control the refresh transistor and the extra port. The second cell (RISC II) is also dual ported, but in this case, they function as both read and write ports. A cell write is performed by activating both ports *(WriteAData)* and driving the data and its inverse on the complementary bit lines (see figure 5.1). The second cell design was chosen because of its smaller size and fewer control signals. The operation of this six transistor cell was verified through extensive SPICE simulation. The completed cell was 55 lambda by 42 lambda. The time for the cell to be written from the time the word lines went high until the bit lines were active was 35 ns. The time for a read was 25 ns.

A two ported cell supports simultaneous read access to the cache array for the snoop and processor cache controllers. The alternative to a dual ported cache is two duplicate tag and state arrays, which is too expensive in area. The implications of the single ported write, and its effects on the the controllers, have already been described in section 4.4. Note there is only one situation where a dual ported write capability is useful, i.e., when the cache and the snoop controllers need to simultaneously update different state bits. In a future implementation, it might be worthwhile to design a dual ported write cell specifically for this section of the cache.

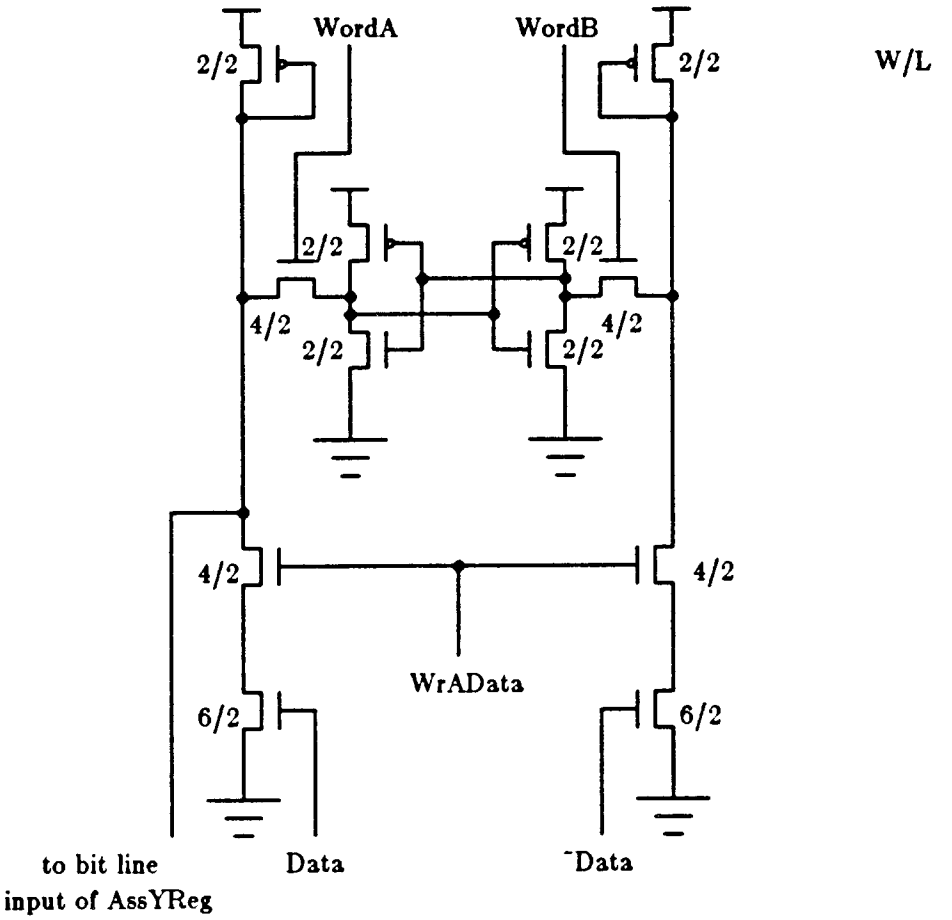


Figure 5.1 – Cache Cell
(Cell cifplot in Appendix 11)

We evaluated both precharged and static cell designs. The precharged memory cell has a precharge PMOS transistor at the top of each bit line. On a read, the bit lines are first precharged to a logic one, and then one line is selectively discharged. On a write, after the precharge phase, write transistors at the base of the bit lines drive the data and its complement on the two bit lines, and the cell is written by this push-pull action. The precharged pull-up has no real speed advantage over the non-precharged cell. Although read/write operations are faster, the advantage is lost because of the need for an extra precharge clock phase. Another disadvantage is the need to run the precharge clock lines through the array. However, the precharged cell requires less

power, since the pullup is off during the evaluation phase.

The non-precharged pull-up design is not purely static. It uses a pseudo-static bit line pullup which consists of a PMOS transistor that has its gate tied to its drain. This pull-up dissipates more power than the precharged pull-up, since it is on most of the time during cache operations. However, the power dissipation is not a significant disadvantage, since the remainder of the chip circuitry is static and dissipates virtually no power. In addition, the pseudo static pull-up eliminates the need for extra precharge clock cycles and control lines. For these reasons, we chose the pseudo-static design of figure 5.1.

Several NOR and a single NAND decoder designs were considered, ranging from full static to domino. The NAND based decoder (see figure 5.2) consists of a four input NAND gate and a PMOS pullup transistor with its gate tied to ground. It drives a CMOS buffer, which in turn drives the word line. Each design was laid out, simulated, and evaluated for area and speed. The NOR decoder designs were large and are not significantly faster than the NAND decoder. Hence, we chose the NAND decoder. SPICE simulations indicate that it can drive the word lines in 7 ns.

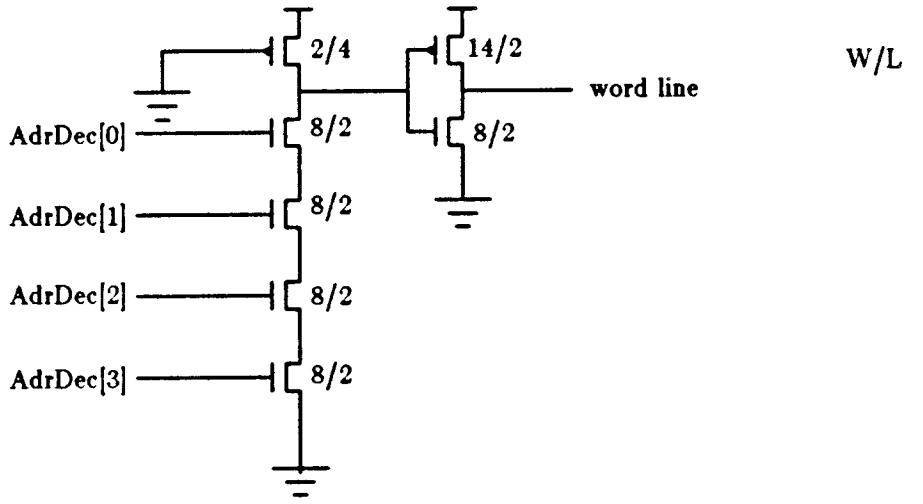


Figure 5.2 – Word Line Decoder
(Cell cifplot in Appendix 11)

Since the cache was 79 bits wide, the array was divided into two halves of 39 and 40 bits each. The decoders are placed in the center of the cache array. They are laid out so that each of four word lines is driven by one decoder. The decoders are nested so that two decoders are driven by the same address lines. Hence, only two address buses are necessary in the center of the cache. The two buses service either the snoop or cache controllers, or both in the case of a write into the cache.

The decoders were designed to drive the word lines. The small size of the arrays made precharged word lines unnecessary; SPICE simulations of the decoder designs indicated that these word lines could be driven in a reasonable amount of time. The decoder bus drivers were designed to allow an address to be driven from the processor side to the system side of the chip and vice versa. This is done by making the driver support a bidirectional bus. (see figure 5.3).

The assembly registers must be able to assemble four 16-bit words from either the processor or the system bus into one 64-bit block and then deliver it to the cache. These registers must also be able to read a 64-bit block from the cache and deliver a 16-bit portion of it to the system bus or an 8 or 16-bit portion of it to the processor bus. The assembly register cell was designed as a two inverter cell with CMOS pass gates to perform the various feedback functions (see figure 5.4). On a write into the assembly register from either processor or system bus, the data are gated to the correct cell by the appropriate mux (A or B). The bits are stored in an interleaved fashion to ease the routing associated with the mux. For example, bit 4 of words 0 through 3 are stored consecutively. Once the data are gated through the multiplexor, the *NoLoad Data* signal is unasserted. This breaks the feedback loop in the assembly register cell by turning off the feedback transistor. The inverters can then be written without contention for the input node. To read from the assembly register cell, the *NoLoad Data* is asserted. This closes the feedback loop in the cell and establishes a path to the bus multiplexor.

To write from the assembly register to the cache, the outputs of the two inverters in the assembly register cell are each connected to a pull down transistor whose drain is connected

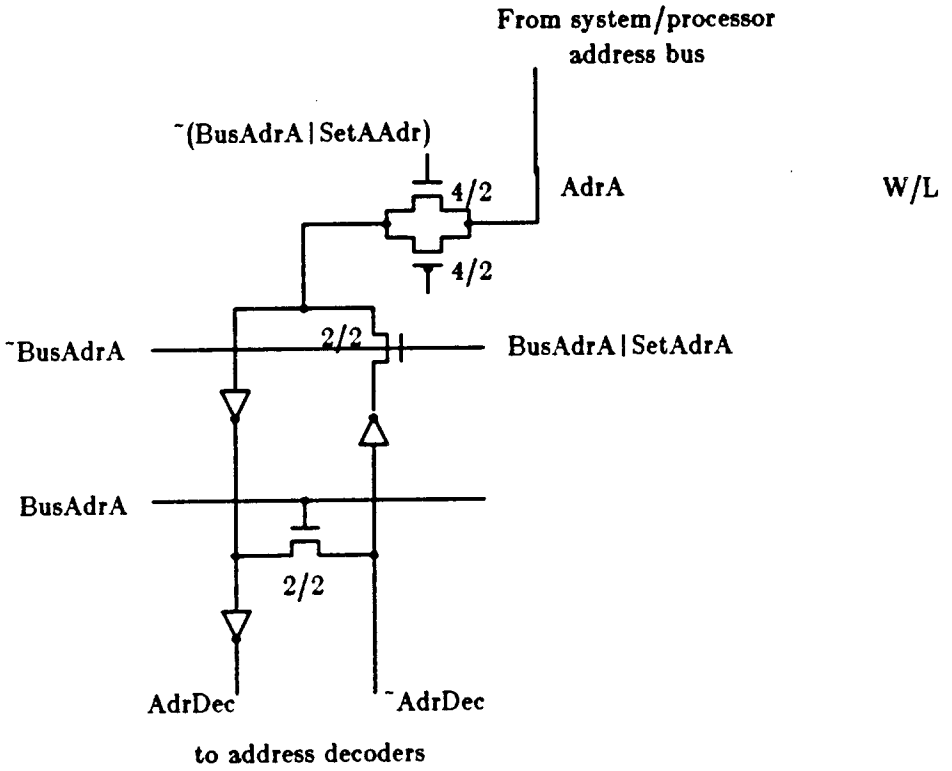


Figure 5.3 – Decoder Driver
(Cell cifplot in Appendix 11)

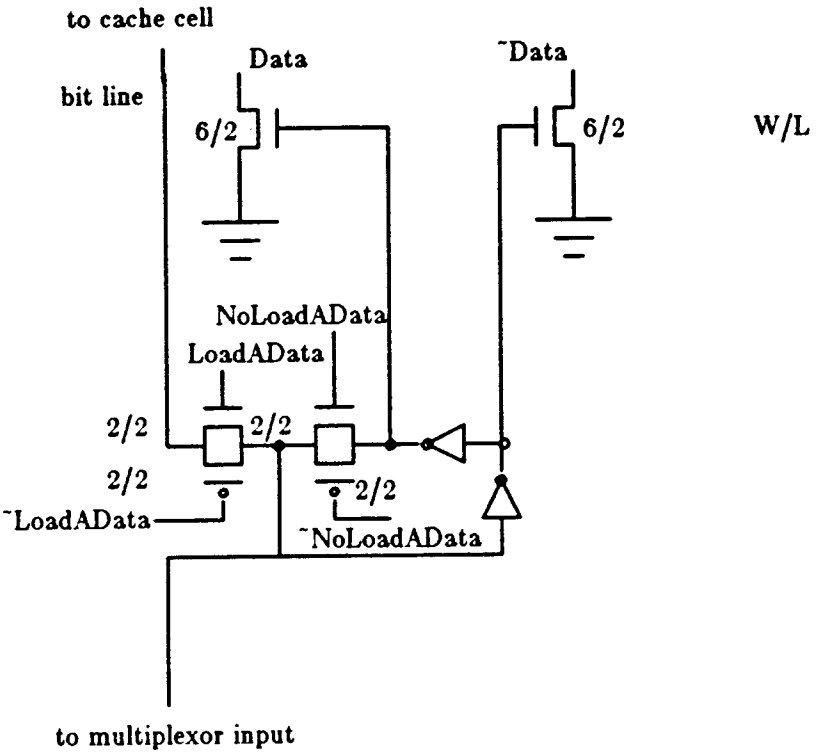
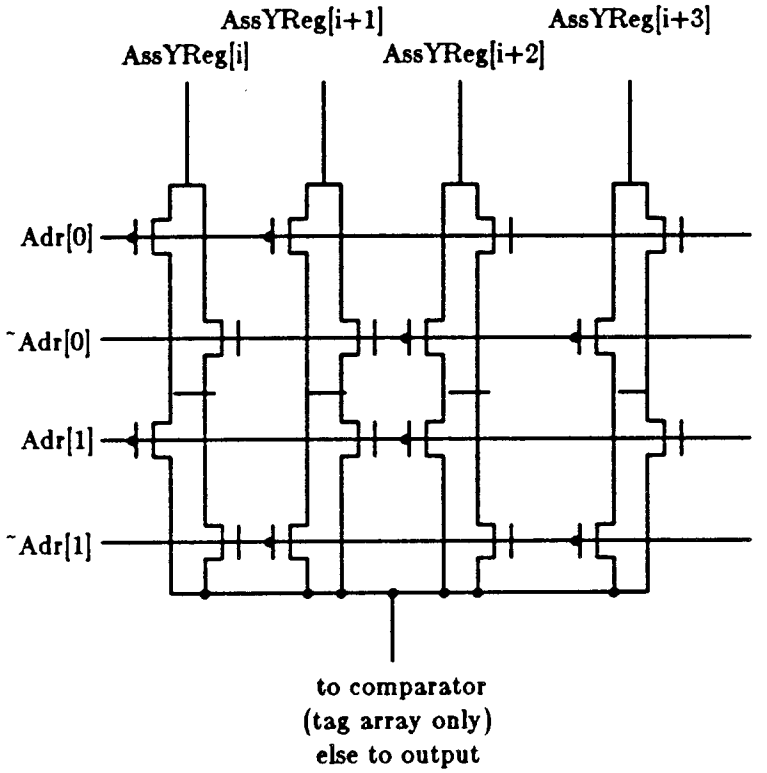


Figure 5.4 – Assembly Register Cell
(Cell cifplot in Appendix 11)

through a pass gate to the bit line. On a write, these pass gates are activated by a *WrFrom* signal, and the outputs of the assembly register inverters are allowed to put complementary data on each of the two bit lines. On a read from the cache, these *WrFrom* pass gates are inactive, disconnecting the write transistors from the bit lines. Instead, a *LoadData* control line is asserted which closes a pass gate connecting one of the bit lines to the input of the assembly register cell. The *NoLoad* signal is again unasserted to avoid driving the input node with two signals.

The bus multiplexors (see figure 5.5), Amux and Bmux, assist the function of the assembly registers by providing a path from the appropriate bus to the assembly registers. These multiplexors consist of two levels of CMOS pass gates that implicitly decode the select signals. The implicit decoding is accomplished because CMOS pass gates are driven by complementary signals.



(all devices W/L=2/2)

Figure 5.5 – Assembly Register Multiplexor
(Cell cifplot in Appendix 11)

The processor bus multiplexor has an extra level of multiplexors to do the ~UDS/~LDS selection required by the 68000. Hence, the processor side of the cache can deliver the high or low byte or the entire sixteen bits addressed.

Comparators match the tag data read from the cache array and the corresponding address bits from the system or processor buses (see figure 5.6). The comparator affects a match line based on the result of this comparison. The comparator design was constrained by the pitch of the memory cell. The pitch of the memory cell basically defined the pitch of all of the peripheral circuitry such as the assembly register, comparator, and superbuffers. The first design we considered was a distributed NOR gate. This NOR gate consisted of a line running through all of the com-

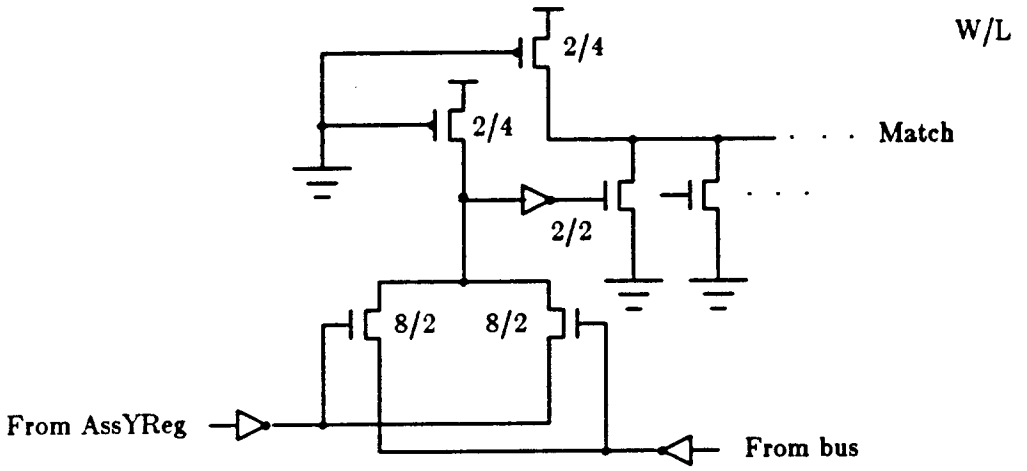


Figure 5.6 – Comparator and Match Logic
(Cell cifplot in Appendix 11)

parator cells, and was the output node of an and-or-invert tree driven by the logical equation (A AND ~B) OR (~A AND B). This type of comparator cell requires A, ~A, B, and ~B to be routed through the middle of the cell. Unfortunately, given the pitch constraints, this was not possible.

Hence, we selected a design which is based on the familiar crosscoupled inverters, where the input lines drive the gate of one pull down and the source of the other. In CMOS, there is no need for pull-up resistors on the input nodes of this circuit. However, since the output node must be driven through a pass gate, drivers on the inputs are necessary. The output of each of these comparator cells is inverted to provide the appropriate polarity, and then drive a pull down that is part of the distributed NOR gate which runs the width of the tag array. The *Match* line is the output node of this distributed NOR gate. It is pulled up by a pseudo-static PMOS device with its gate tied to ground.

5.3. Implementation of the Snoop and Cache Controllers

This section contains a description of the implementation of the Snoop and Cache controllers. This implementation proceeded through the following stages:

1) Detailed description of protocol

2) Analysis of protocol for critical sections

3) Initial datapath design

4) Initial description of finite-state machines

5) Final datapath design

6) Final descriptions of finite-state machines

7) Timing characteristics

8) Functional simulation

9) CMOS implementation

5.3.1. A Detailed Description of the Protocol

A detailed description of the protocol is given in Appendix 1. This description was used as a basis for the analysis of the protocol for critical sections, and for the preliminary descriptions of the datapath and the finite state machines (FSMs) implementing the Snoop and Cache Controllers. This analysis is necessary to insure the atomicity of the actions of the controllers in response to requests from the Multibus and the associated processor.

Looking ahead, the similarity between the Cache Controller's responses to write and *Read-And-Set* requests indicates that those tasks can be combined in that controller's FSM. This is no accident. The *Read-And-Set* primitive was designed after we realized that its implementation was practically free.

5.3.2. Analysis of the Protocol for Critical Sections

There are two resources for which the Snoop and Cache Controllers can contend: the two cache memory decoders, and the cache memory contents. Contention for these two resources were analyzed separately. A discussion of contention for these resources appears in Appendix 2.

Since a controller can write to the cache memory only when in a critical section, a simple scheme could be used to avoid contention for the memory decoders: a simple oscillator *(WriteCy-*

Cache is bus master?	Yes	No	No	No	No
Snoop in critical section?	-	-	Yes	No	No
Cache in critical section?	-	-	No	Yes	No
WriteCycle?	-	No	Yes	Yes	Yes
Cache can read	Yes	Yes	No	No	No
Cache can write	Yes	No	No	Yes	No
Snoop can read	No	Yes	No	No	No
Snoop can write	No	No	Yes	No	No

Table 5.1 – Resolving contention for the cache memory decoders

The Snoop and Cache Controllers could contend for the two memory decoders. Writing to the cache memory requires both decoders, reading only one. By requiring each controller to observe the *WriteCycle* signal, and by allowing writing to the cache memory only within critical sections, this contention is eliminated. The decision table above specifies those situations in which each controller can read or write the cache memory without contention.

cle) marks every other cycle as being suitable for writing the cache memory. The *ProcHas/SnoopWants* protocol (see Section 4) is used by the controllers to prevent more than one controller from being in a critical section. Table 5.1 shows when the Snoop and Cache Controllers may read and write the cache memory.

5.3.3. Initial Datapath Design

The initial design for the cache memory is illustrated in figure 5.7. The assembly registers on one side (the top of the diagram) are used exclusively by the Snoop Controller; those on the other are used by the Cache Controller. The System Bus Interface must be able to access both data assembly registers. A list of the signals and registers used to control this datapath is given in Appendix 3.

5.3.4. Initial Controller Specification

The initial specification for the Snoop and Cache Controllers are given in Appendix 4 and Appendix 5, respectively. These specifications are based on the initial datapath described above. Each controller is described as a simple Mealy machine. This choice was made for simplicity; no

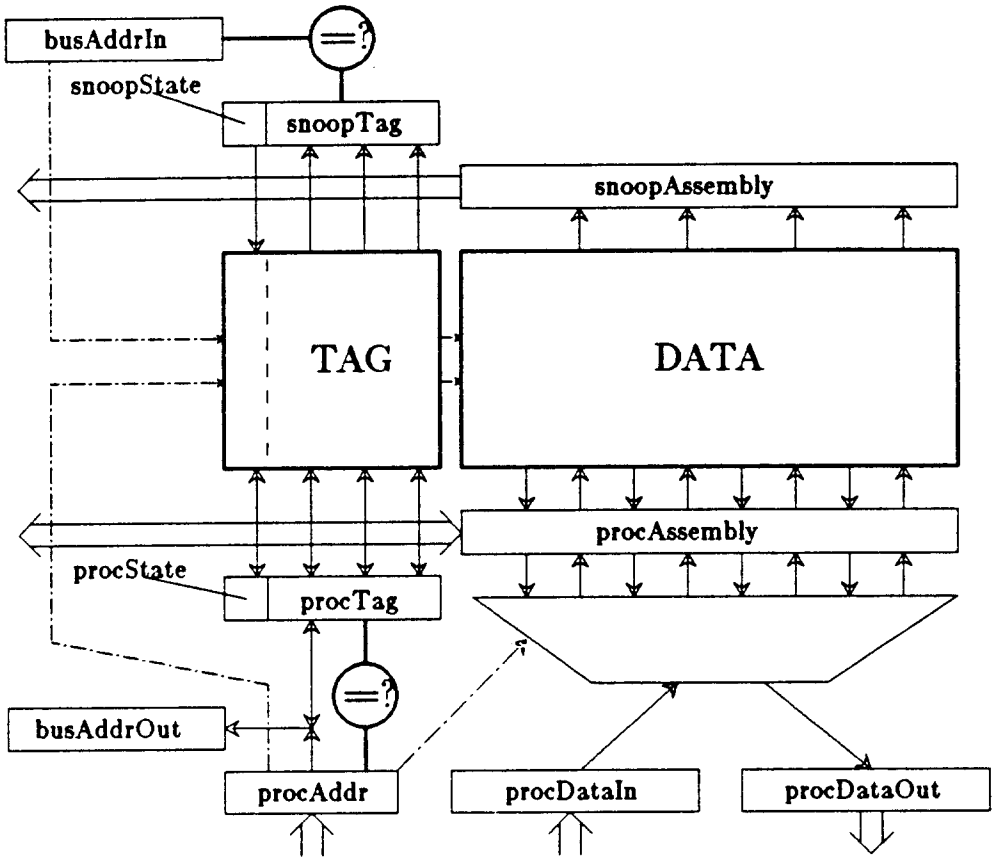


Figure 5.7 -- The initial datapath

This datapath was the basis of the initial descriptions of the Snoop and Cache Controllers. The System Bus Interface (to the left of this picture) includes the *busAddrIn* and *busAddrOut* registers. The Processor Bus Interface (below this picture) includes the *procAddr*, *procDataIn*, and *procDataOut* registers. Since the registers used by the Snoop and Cache Controllers are isolated, the System Bus Interface requires access to both assembly registers.

separate next-state circuitry would be required in the implementation of these FSMs.

There are three significant differences between the initial and final controller descriptions. First, the initial Cache Controller had no provision for a *Read-And-Set* operation. As noted above, the *Read-And-Set* synchronization primitive was designed after we noted how easy it would be to implement.

The other differences involve two different methods of resolving contention between the Snoop and Cache Controllers. Mutual exclusion within critical sections was initially obtained by

means of a hardware semaphore. Each controller has *Set* and *Clear* output signals which are used to obtain and release the semaphore. The semaphore produces two *OK* signals to indicate which controller currently owns it.

Contention over the two memory decoders is resolved by means of a hardware arbitrator. All tag/data read/write signals from the controllers are monitored by the arbitrator. The presence of any read signal from one controller causes the arbitrator to inhibit any write signal from the other controller. Similarly, if both controllers issue write signals, then the arbitrator inhibits one and allows the other to pass. When a controller issues a write signal, it must be prepared to wait for confirmation from the arbitrator.

5.3.5. Analysis of the Initial Datapath and Controller Designs

The fatal flaw of the initial datapath was its requirement that the System Bus Interface be able to access both assembly registers. The final datapath design (described section 5.2) avoids this problem by forcing the Cache Controller to interact with the System Bus Interface through the registers normally used by the Snoop Controller.

The initial approach to handling contention between the Snoop and Cache Controllers was considered to be too complex: nine signals and two black boxes were required. The final approach (using only the *ProcHas, SnoopWants,* and *WriteCycle* signals) involves no black boxes.

The size of the initial Cache Controller was cause for alarm. A PLA-based implementation of that controller's FSM would involve approximately 40 minterms, resulting in a 70ns delay (if implemented in NMOS).

5.3.6. Final Controller Design

The final controller design differs from the original. It reflects the final datapath design (section 5.2), and the Cache Controller is described in terms of four cooperating FSMs. Each FSM responds to a subset of the possible processor requests (*Read, Write,* or *Read-And-Set*) on a subset of the possible block states:

a) *Read* to an **UnOwned, Owned Exclusively,** or **Owned NonExclusively** block

b) *Read* to an **Invalid** block

c) *Write* or *Read-And-Set* to an **Owned Exclusively** block

d) *Write* or *Read-And-Set* to an **Invalid, UnOwned,** or **Owned NonExclusively,** block

The first of these FSMs is trivial (see the initial Cache Controller description) and so was incorporated into the Processor Bus Interface. In this way, a single controller (within the interface) could respond to processor read-requests to valid blocks within the cache memory. The problem of reading the block into the *BState, BTags,* and *BAssembly* registers was answered by causing the block to be read whenever possible while waiting for the next processor request. In this way, the Processor Bus Interface does not have to explicitly assert the control lines which read values into those registers.

The rest of these FSMs are termed the (b) Read, (c) Write1, and (d) Write2 FSMs. If the Processor Interface is unable to handle a processor read-request, then it passes control to the Read FSM. If a *Write* request or a *Read-And-Set* request is received, then control is passed to the Write1 FSM. If the block is not **Owned Exclusively,** then the Write1 FSM passes control to the Write2 FSM. When finished, the Write2 FSM signals the Write1 FSM, which then signals completion to the Processor Bus Interface. The structure of the Cache Controller is illustrated in figure 5.8.

Note that since the Read and Write1 FSMs both generate a *ProcAck* signal, the actual value for that signal is the OR of the values generated by the FSMs. Such treatment is common to all output signals issued by more than one FSM in the Cache Controller.

The signals used by the Snoop and Cache Controllers are given in Appendix 6. The final descriptions of the Snoop, Read, Write1, and Write2 FSMs are given in Appendices 7, 8, 9, and 10, respectively. By splitting the Cache controller into these FSMs, the largest FSM (i.e. that with the largest number of terms) is now the Snoop.

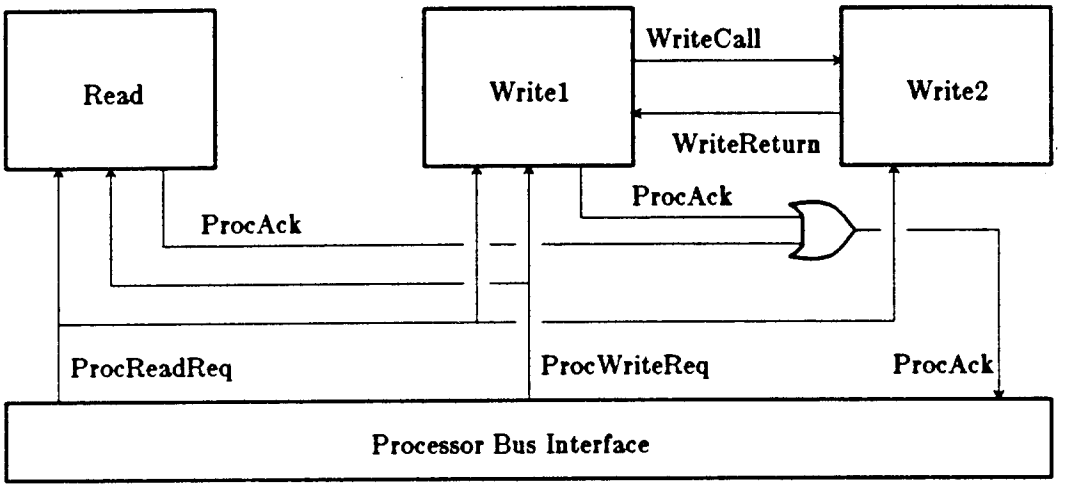


Figure 5.8 – The Structure of the (Processor) Cache Controller

The task of responding to requests from the processor is divided among four finite state machines (FSMs). Typically, only one of these machines is active. The Processor Bus Interface responds to read requests for words resident in the cache. The *Read* FSM responds to read requests that require a bus transaction. The *Write1* and *Write2* FSMs respond to *Write* and *Read-And-Set* requests. The signals shown in the diagram are those used to pass control from one of these machines to another.

5.3.7. Timing Considerations

Preliminary timings of the memory subsystem indicated that a minor cycle of at least 40 nanoseconds is required. The clocking of the processor is very close to what is required by the memory subsystem. Further, there are significant performance advantages if the cache controller can run synchronously with the processor: a processor request could then be detected with no wait states. The clock from a 10 megahertz M68000 can be translated into a two-phase clock suitable for use with the memory subsystem as shown in figure 5.9. This yields a 50 nanosecond minor cycle, with a 20 ns high time and a 30 ns low time. The memory decoders are driven while *Phi1* is high. The memory cells are read or written while *Phi2* is high.

The low-phase of the M68000 clock is associated with *Phi1* to minimize the time required to acknowledge a processor read-request to a (valid) line within the cache. In the M68000 read sequence, the *address strobe (AS)* is raised while the clock is high to initiate a read/write opera-

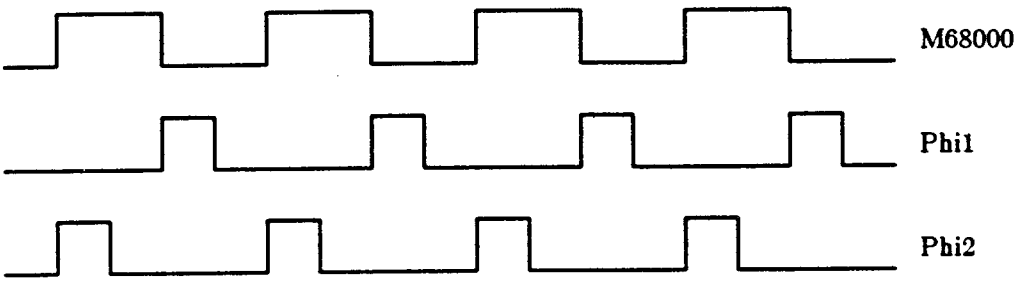


Figure 5.9 – Derivation of Two Phase Clock

The two phase clock used throughout the controller is derived from the M68000 clock as shown above. The association of the low phase of the processor's clock with *Phi1* is due to the *Address Strobe (AS)* signal. This signal is activated by the processor during a high phase *(Phi2)* to signal a read/write request. By generating the two phase clock in this way, the Processor Bus Interface can sample *AS* during *Phi1* to detect a read/write request without delay.

tion. When this signal is raised, the address lines can be assumed to be valid. Since the address cannot be considered to have reached the decoders before the M68000 enters a low-phase, and since the address must be valid at the start of *Phi1* (to drive the decoders), *Phi1* is associated with the low-phase of the 68000 clock.

All PLAs are clocked in the conventional way, with inputs sampled during *Phi1,* and outputs valid during *Phi2.* The decision is based on the observation that since the tag is read from the cache memory on *Phi2,* it not safe to propagate the result of the tag comparison to the inputs of the PLAs during the same phase.

5.3.8. Functional Simulation

Once the decision was made to use a simple two-phase clocking scheme, a simple simulator was produced which modeled the subsystems in the data cache chip and which allowed a simulation of a multiprocessor system including a number of these data caches.

The simulation of the FSMs within the controller was accomplished by translating the FSM specifications into C procedures and executing those procedures directly. In this way, we could be more certain of the correctness of the FSM specifications. The simulation system generated

random requests to memory, given a specification of the number of memory blocks, the amount of sharing, and the frequency of read and write requests to shared and private memory blocks. A simulation of a three-processor system executed for 50,000 simulated clock cycles without error.

5.3.9. CMOS Implementation

Due to the presence of a good PLA generator (tpla), the decision was made to use a PLA-based implementation for the controller FSMs. A translator was written to convert the FSM descriptions into truth-tables suitable for PLA generation. Both static and dynamic (precharged) CMOS PLAs were available. Preliminary timings of the static PLAs indicated that a minor cycle of 90 nanoseconds would be required. As a result, the dynamic CMOS PLAs were chosen.

The precharge clocks for the AND and OR planes are generated as follows. Since the inputs to the PLA (and to the AND plane) are not valid until the falling edge of *Phi1*, the AND plane is precharged while *Phi1* is high. Since the outputs from the PLA (and from the OR plane) must be valid by the falling edge of *Phi2*, the OR plane is evaluated while *Phi2* is high. This implies that both planes must be precharged while their inputs are being evaluated. To accomplish this, the precharge time for each plane is considerably longer than the evaluation time. The precharge time for the AND plane begins with the falling edge of *Phi2*, and the precharge time for the OR plane begins when *Phi1* goes high. The relationship between these clocks is illustrated in figure 5.10.

The CRYSTAL simulation parameters and results are given in Tables 5.2a and 5.2b. To achieve the desired 50 ns minor cycle time, the sum of the AND plane and OR plane delays must be less than 50 ns (the time between the falling edges of *Phi1* and *Phi2*). Both the Write2 PLA and the Snoop PLA fail to meet this objective, the latter by a wide margin. The controller implementation needs further refinement if the cache chip is to use the same clock as the processor.

In addition to a PLA, each FSM requires clocked input/output inverters. By transmitting control signals in an inverted state, the FSM output can be taken directly from the clocked output inverters. Finally, circuitry is associated with each FSM in order to implement *shared* output signals. Since these signals are inverted, this circuitry consists of a collection of static AND gates,

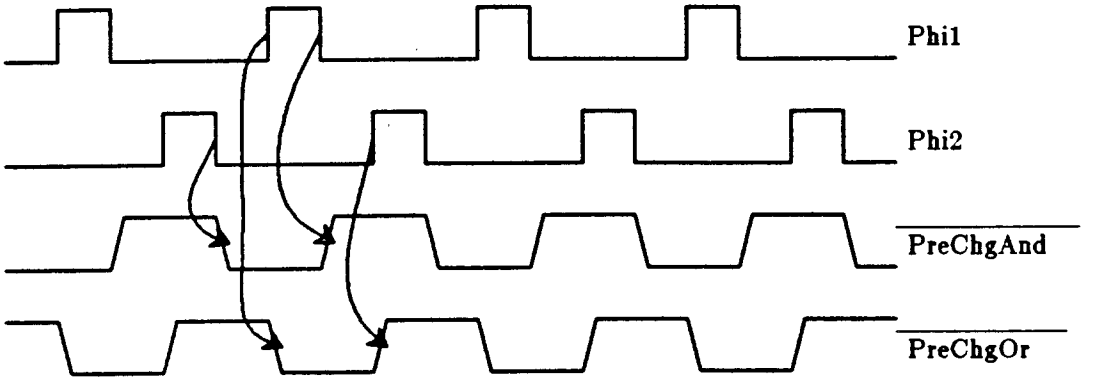


Figure 5.10 – Generating PLA precharge clocks based on Phi1 and Phi2

The relationship between the clocks which latch PLA inputs *(Phi1)* and outputs *(Phi2)* and those which precharge the AND and OR planes of each PLA is illustrated above. A plane within a PLA is being precharge while its associated clock is low.

Table 5.2a – CRYSTAL SIMULATION PARAMETERS	
areatocap diff	600
perimetertocap diff	400
areatocap poly	65
areatocap metal	52
areatocap poly/diff	637
capthreshold	0.1

Table 5.2b – CRYSTAL SIMULATION RESULTS		
PLA	And Plane Delay	Or Plane Delay
Write1Pla	29.99 nsec	19.68 nsec
Write2Pla	31.50 nsec	36.81 nsec
ReadPla	21.72 nsec	24.48 nsec
SnoopPla	61.55 nsec	31.54 nsec

one gate for each shared signal. The floorplan for a single FSM is illustrated in figure 5.11.

The floorplan for the controller (showing its relationship to the datapath) is illustrated in figure 5.12. The signals within the control bus which deal with the A-side of the cache or with the System Bus Interface terminate on the left, those which deal with the B-side or the Processor Bus Interface terminate on the right.

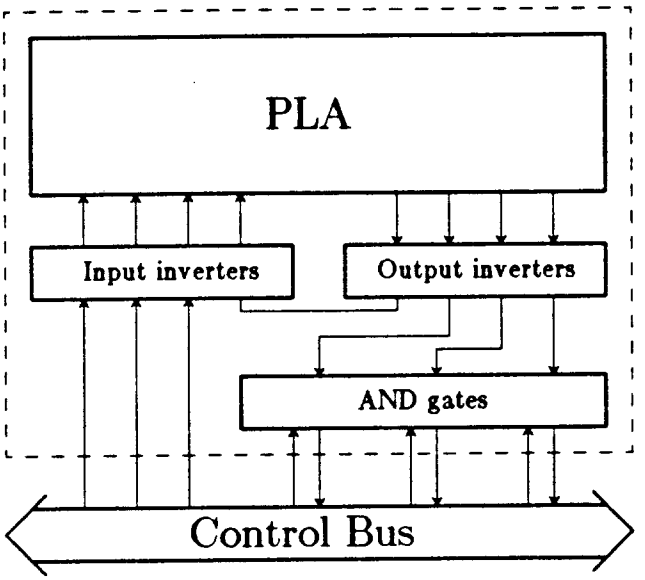


Figure 5.11 - The floorplan of a finite-state machine (FSM)

The FSMs implementing the Snoop and Cache controllers are based on dynamic PLAs. Clocked inverters latch the input and output signals. If this FSM generates a signal which already is present in the Control Bus (i.e. that signal is generated by another FSM), then the two values (from the PLA and from the Control Bus) are combined in an AND gate to form the updated value.

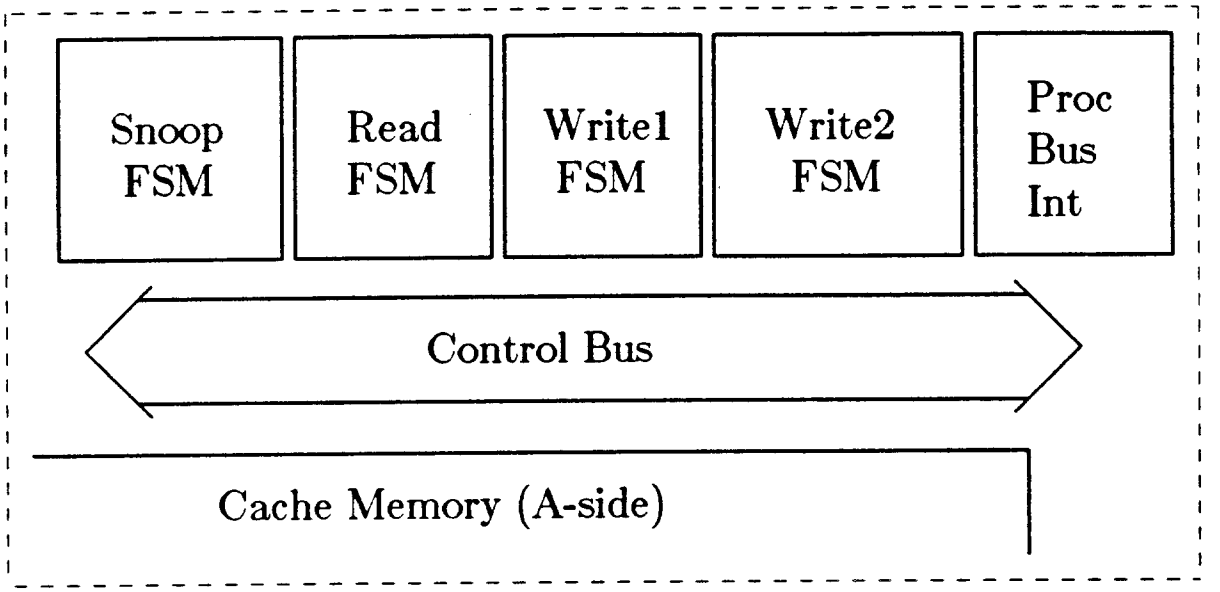


Figure 5.12 – The floorplan for the Snoop and Cache controllers

The FSMs implementing the Snoop and Cache controllers are laid out in a horizontal array in the upper-right corner of the chip. Signals involving the Processor Bus Interface (shown above) and the B-side of the Cache Memory are presented on the right edge of the Control Bus. Signals involving the System Bus Interface (not shown) and the A-side of the Cache Memory are presented on the left edge of the Control Bus. Signals heading toward the B-side must travel down the right edge of the Cache Memory.

Additional circuitry is required to provide the exact signals required by the datapath. This circuitry is illustrated in figure 5.13. Note that signals leave each FSM PLA when *Phi2* falls, and that they must arrive at the datapath before *Phi1* falls. Exceptions to this rule are the signals that drive the decoders: *BusAdrA*, *BusAdrB*, *ProcAdrA*, and *ProcAdrB*. These signals must arrive at the datapath by the rising edge of *Phi1* to drive the decoders while *Phi1* is high. These leads us to believe that the critical path for the control signals consists of the circuitry which produces *ProcAdrB*. This circuitry (including PLA output inverters, AND gates for shared signals, and interconnect) was timed at 26.5 nanoseconds using Crystal. This result indicates that there is sufficient time (30 nanoseconds) to deliver signals from the controller to the datapath, if the controller PLAs can be redesigned to operate within 50 ns between the falling *Phi1* and *Phi2* edges.

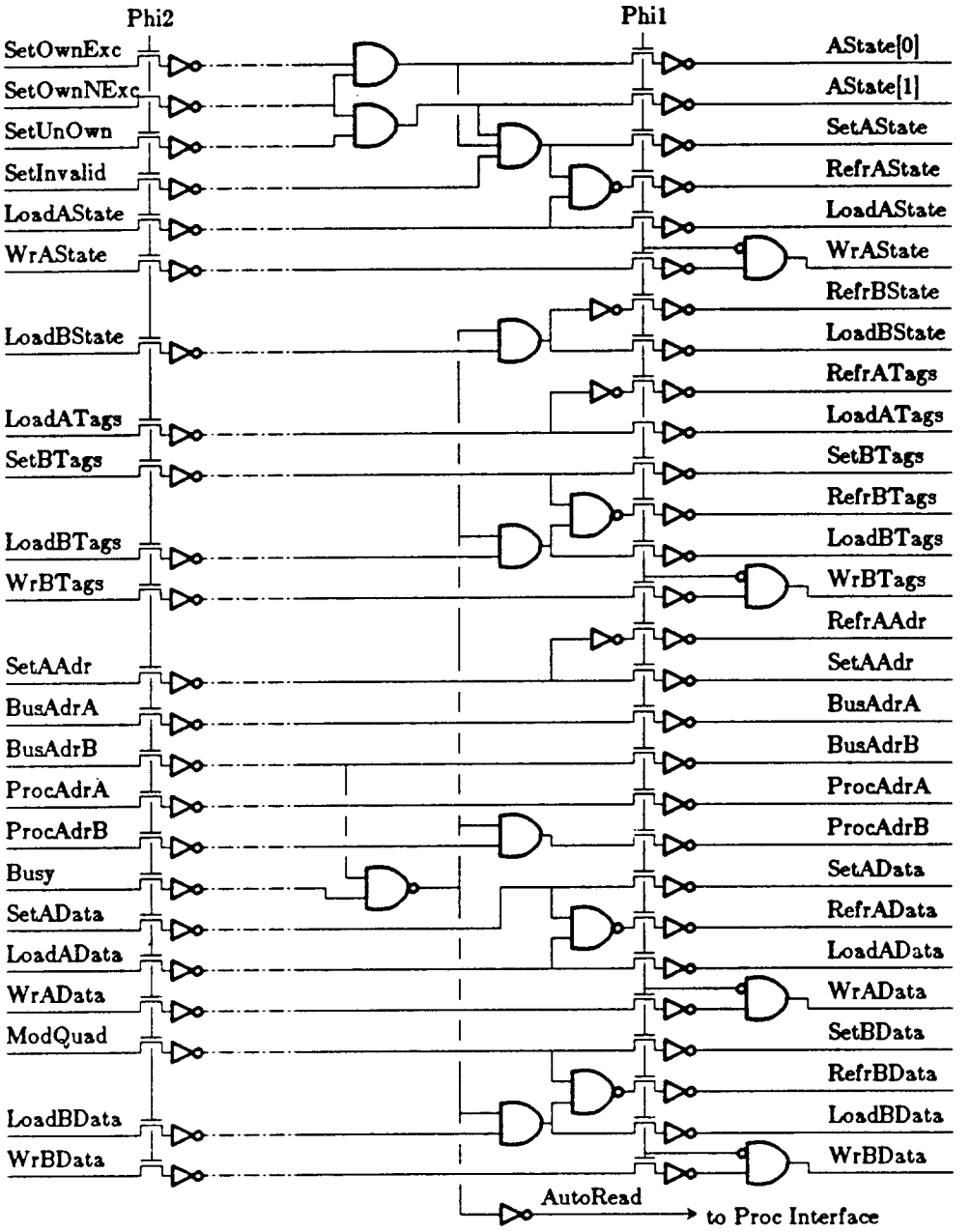


Figure 5.13 – Generation of signals required by the Cache Memory

Initial layout of the controller has been completed. The controller (including an area for the Processor Bus Interface) measures 2700 lambda wide by 800 lambda high. Figure 5.14 is a plot of the controller PLAs and the datapath.

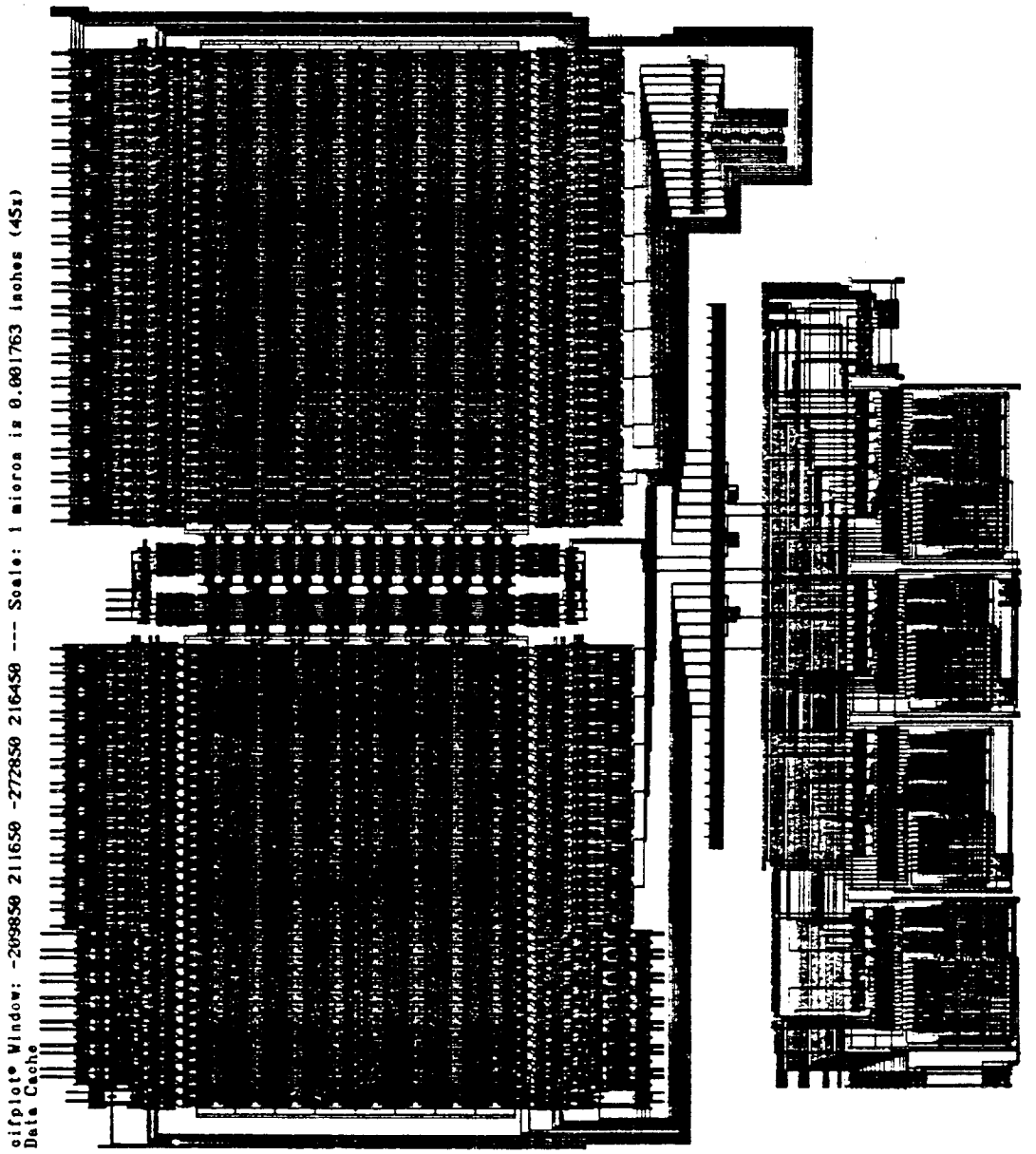


Figure 5.14 -- Controller and Datapath Cifplot

6. Discussion

6.1. Methodology

The designers' work divided up according to functional modules of the system: the Datapath, the Snoop Controller, the Cache Controller, the Processor Bus Interface, and the System

Bus Interface. In addition, several members of the team played special roles: design tool assistance, SPICE expert, and SLANG describers. Interestingly enough, none of the designers were very keen on doing layout, and in retrospect, more people should have been devoted to this task.

The block diagram view of the chip was determined very early, in fact, before the course began. The most pressing objective was to select or design a correct cache consistency protocol for implementation. While relatively easy to specify at the functional level, a detailed specification was needed to insure that the implementation would be correct and free of race conditions. This was accomplished by iteratively developing a SLANG functional description of the protocol throughout the semester. The controller design did not begin in earnest until this specification was complete.

The Datapath implementation proceeded in parallel with the SLANG development. It is unfortunate that we did not have a complete SLANG description before we started layout, but this was not possible because of the short time schedule. This caused a number of problems later (see section 7).

Unfortunately, SLANG is not as useful as we would have wished. To obtain the level of detail necessary to verify that the protocol implementation would be free of race conditions required a low-level and very complete specification. This required an effort almost equivalent to designing the control PLAs, generating and then simulating them. In fact, a separate language was designed and a translator was written from it to SLANG to avoid having to enter all the detail by hand. This new language provided some higher level ways of thinking about the operation of the controllers, and it could be translated into SLANG whenever the system was to be simulated. The effort was not wasted, as we also developed some tools for generating the PLAs and other control logic directly from this new language. Thus, the description of the controllers in this language was the central one, all others derived from it. These descriptions appear in the later Appendices.

6.2. Experiences/Lessons Learned

Undertaking the construction of a system of such complexity turned out to be more ambitious than we expected. We needed to design a cache consistency protocol, and convince ourselves that it was correct, before we could really begin designing the chip. Roughly half the effort was spent in this activity!

We believe that a snooping data cache is approximately four times as complex to design as an instruction cache. A simple data cache is about twice as complex as an instruction cache because of the need to support processor read and write operations, and the need to copy blocks back to memory when a cache entry is replaced. The complexity is doubled again when a snoop controller is also added to the system, because it needs to be synchronized with the processor cache controller during accesses to the cache memory.

While we are convinced that the ownership protocol is correct, we cannot yet argue that it is the best possible. However, we believe that no snooping cache protocol can be implemented with substantially less complexity. A dual-ported cache memory is still required, as are the interlocks between the snoop and the cache controllers. Several of the published protocols ignore some issues, such as support for atomic bus operations, which we have tried to address. The protocol we have implemented is at least functionally complete.

The project revealed a glaring deficiency in our battery of design tools: we have very weak support for functional simulation. No tools were available for quickly verifying the functional correctness of our design. Generating a SLANG [MAYO83] description by hand is difficult and tedious. Using the description for multilevel simulation with a tool like ESIM [MAYO83] requires it to be specified with such a fine level of detail that the goal of functional description is defeated.

7. Status of Implementation

At the time of this writing, the completed cache chip is not available. The Datapath, Cache Controller, and Snoop Controller have been completely laid out and verified for correctness. The

Processor Bus Interface and the System Bus Interface, although designed, have not been laid out.

A number of things went wrong on the way to completing the chip. The Datapath was laid out well before the final form of the control was determined. However, the Datapath and Controller designers were different groups that had never really decided on a common interface. Not all controller generated output signals are compatible with the control lines they drive in the Datapath. A considerable amount of random "glue" logic remains to be laid out to make these compatible.

Because the final floorplan placement of the controllers was not determined until late in the design, the datapath was implemented with a poor control signal interconnection topology. For example, some control wires need to be routed around the sides of the datapath because they cannot be reached from the top where the control PLAs have been placed. This significantly increases an already large layout area, and adds additional RC delay between the controllers and the datapath control points. Future iterations on the implementation will have to pay more careful attention to the interconnect topology, overall chip floorplan, and interface between control and datapath.

Based on a CRYSTAL performance evaluation of the implemented pieces of the design, we have discovered that we cannot meet our goal of a 100 ns clock cycle. The critical path through the controller should have been not more than 20 ns, but it is actually about 40 ns. A completely new controller implementation is necessary to obtain the needed performance.

Evaluation of the datapath is currently underway, and we hope to have some real numbers before this report is officially published.

8. Conclusions and Future Directions

We have described a multiprocessor cache consistency protocol, its evaluation, and its design and implementation by a single chip snooping data cache. While snooping caches have been built before, to our knowledge this is the first attempt to construct a version that integrates the

memory and the controllers on a single chip.

The implementation effort was directed towards completing a rapid prototype in a single fifteen week semester, a rather ambitious task. While the design is complete, the chip layout is still incomplete. This indicates some of the difficulties in undertaking a large design project that cannot be finished in a single semester.

We had hoped to be able to use the chip, in conjunction with the 68020 and the MultiBus, to perform experiments on the utility of snooping caches and our particular consistency protocol. While the protocol specification and the completed chip design are notable successes, our inability to complete the implementation is something of a disappointment. The attempt to complete the prototype was worthwhile, nonetheless, as it reenforced many of the guiding principles of VLSI design.

A future objective is to undertake the design and implementation of a single chip multiprocessor building block that incorporates on-chip instruction and snooping data caches as well as the processor. The result would be a very high performance system at very low cost. The lessons learned by this rapid prototype implementation will be used to guide the implementation of this next project.

9. References

[AGRA77] Agrawal, O. P., A. V. Pohm, "Cache Memory Systems for Multiprocessor Architectures," Proc. AFIPS National Computer Conference, Dallas, TX, (June 1977).

[FRAN84] Frank, S., "Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics,* (Jan 12, 1984).

[GOOD83] Goodman, J., "Using Cache Memories to Reduce Processor-Memory Traffic," 10th Annual Symposium on Computer Architecture, Trondheim, Norway, (June 1983).

[MAYO83] Mayo, R. N., J. K. Ousterhout, W. S. Scott, eds., "1983 VLSI Tools: Selected Works by the Original Artists," Report No. UCB/CSD 83/115, (March 1983).

[MONT77] Montgomery, W. A., "Measurements of Sharing in Multics," Proceedings of the 6th SOSP, Operating Systems Review, V 11, N1 (November 1977).

[NORT82] Norton, R. L., J. L. Abraham, "Using Write Back Cache to Improve Performance of Multiuser Multiprocessors," Intl. Conf. on Parallel Processing, (1982).

[PATT81] Patterson, D. A., C. Sequin, "RISC 1: A Reduced Instruction Set VLSI Computer," 8th Annual Symposium on Computer Architecture, Minneapolis, MN, (June 1981).

[PATT83] D. A. Patterson, et. al., "Architecture of a VLSI Instruction Cache," 10th Annual Symposium on Computer Architecture, Trondheim, Norway, (June 1983).

[RAVI83] Ravishankar, C. V., J. Goodman, "Cache Implementation For Multiple Processors," IEEE Spring Compcon Conference, San Francisco, (Feb 1983).

[SHER81] Sherburne, R. W., et. al., "Datapath Design for RISC," MIT Conference on Adv. Research in VLSI, Cambridge, (January 1982).

[SMIT82] Smith, A. J., "Cache Memories," *ACM Computing Surveys*, V 14, N 3, (September 1982).

[THAC82] Thacker, C. P., Presentation at UC Berkeley Systems Seminar, (1982).

[UNGA84] Ungar, D., et. al., "Architecture of SOAR: Smalltalk on a RISC," 11th Annual Symposium on Computer Architecture, Ann Arbor, MI, (June 1984).

10. Appendix 1 - Detailed Description of the Snoop and Cache Controllers

The Snoop Controller:

If the Snoop Controller overhears a *Read* for a block owned by the cache, then the main memory is inhibited, and the block's state becomes **Owned NonExclusively.**

Invalid or UnOwned:
    Ignored.
Owned Exclusively:
    Raise the Multibus INHIBIT signal and supply the desired word.
    Change the block's state to Owned NonExclusively.
Owned NonExclusively:
    Raise the Multibus INHIBIT signal and supply the desired word.

If the Snoop Controller overhears a *Read-For-Ownership* for a block owned by the cache, then the main memory in inhibited, and the block's state becomes **Invalid.**

Invalid:
    Ignored.
UnOwned:
    Change the block's state to Invalid.
Owned Exclusively or Owned NonExclusively:
    Raise the Multibus INHIBIT signal and supply the desired word.
    Change the block's state to Invalid.

If the Snoop Controller overhears a *Write-For-Invalidation,* then the block state becomes **Invalid.**

Invalid:
    Ignored.
UnOwned, Owned Exclusively, or Owned NonExclusively:
    Change the block's state to Invalid.

The Snoop Controller ignores all *Write-Without-Invalidation* operations.

The Cache Controller:

In response to a processor *Read* request, the Cache Controller can immediately supply a word from any valid block in the cache memory. Otherwise, an **UnOwned** copy of the block must be read through the Multibus.

Invalid:
    Obtain control of the Multibus.
    Write (without invalidation) any bumped block to main memory.
    Read (publicly) the requested block.
    Mark the block's state as UnOwned.
    Release control of the Multibus.
    Supply the requested word to the processor.
UnOwned, Owned Exclusively, or Owned NonExclusively:
    Supply the requested word to the processor.

In response to a processor *Write* request, the Cache controller can immediately alter a word from any exclusively owned block in the cache memory. Otherwise, an exclusively owned copy of the block must first be obtained. To prevent another cache from stealing ownership of the block, the Cache Controller releases control of the Multibus after storing the modified block in the cache memory.

Invalid:

Obtain control of the Multibus.
Write (without invalidation) any bumped block to main memory.
Read (for ownership) the requested block.
Mark the block's state as Owned Exclusively.
Write the modified block into the cache memory.
Release control of the Multibus.

UnOwned or Owned NonExclusively:

Obtain control of the Multibus.
Write (without invalidation) any bumped block to main memory.
Write (with invalidation) the requested block.
Mark the block's state as Owned Exclusively.
Write the modified block into the cache memory.
Release control of the Multibus.

Owned Exclusively:

Write the modified block into the cache memory.

In response to a processor *Read-And-Set* request, the Cache Controller behaves much as for a write-request. The only difference is that the original contents of the word is saved and returned to the processor before the block is altered.

Invalid:

Obtain control of the Multibus.
Write (without invalidation) any bumped block to main memory.
Read (for ownership) the requested block.
Mark the block's state as Owned Exclusively.
Supply the requested word to the processor.
Write the modified block into the cache memory.
Release control of the Multibus.

UnOwned or Owned NonExclusively:

Obtain control of the Multibus.
Write (without invalidation) any bumped block to main memory.
Write (with invalidation) the requested block.
Mark the block's state as Owned Exclusively.
Supply the requested word to the processor.
Write the modified block into the cache memory.
Release control of the Multibus.

Owned Exclusively:

Supply the requested word to the processor.
Write the modified block into the cache memory.

11. Appendix 2 - Analysis of the Cache Consistency Protocol Implementation

This analysis determines the "critical sections" in the behavior of the Snoop and Cache Controllers. We claim that if the two controllers are never simultaneously in a critical section, then their responses to external requests can be considered to be atomic, thereby preserving the correctness of the cache-consistency protocol.

Contention for the Cache Memory:

We argue that in no case will the Cache Controller interfere with the Snoop while servicing a processor *Read* request. Suppose that the block being read is **Invalid.** In this case, the Cache Controller must first obtain control of the Multibus. Once it is obtained, the Snoop is disabled, precluding any snooping activity.

Otherwise, the Snoop is active and could either be reading the block or altering the tag field. If the Snoop is altering the block's tag to **Invalid,** then the processor's request can still be handled. The processor will either see the **Invalid** state, indicating that the read happened after the write, or the state of the block before it was invalidated, indicating that the write happened before the read. To avoid this non-deterministic result, a program must be structured with the appropriate synchronization primitives to insure proper mutual exclusion.

In the event of a processor *Write* request to an **Invalid, UnOwned,** or **Owned NonExclusively** block, control of the Multibus is required before the *Write* request can be serviced. If the Cache Controller does nothing before obtaining control of the Multibus, no contention exists, since the Snoop will be inactive while the Cache Controller is reading/modifying the cache memory. Note that the Cache controller must reexamine the tag after control of the Multibus is obtained.

Furthermore, conflict cannot occur between a processor *Write* request and a *Write-For-Invalidation* to an **OwnedExclusively** block. Since a *Write-For-Invalidation* can only be issued by a cache with a valid copy of the block, no other cache could initiate this bus command. And if the processor and an I/O controller are trying to simultaneously write into a block (i.e. and I/O buffer), then mutual exclusion is again being violated at the program level.

So for the purposes of this discussion, we only need consider two possible conflicts on an **OwnedExclusively** block: between a processor write-request and bus read-request (with or without invalidation). Since the snooping behaviors for these two external requests are similar, their treatment can be combined.

Let's look at this snooping behavior more closely:

* Read the block
* Raise the Multibus INHIBIT signal
* Place the appropriate word on the Multibus
* Store the new tag (Owned NonExclusively or Invalid)
* Lower INHIBIT

Now, let's look at the activity resulting from a processor write-request:

* Read the block
* Form the modified block in the assembly register
* Rewrite the block

The critical section for the Snoop includes reading the block from the cache memory and storing the new state. The critical section for the Cache Controller includes reading the tag (to insure that the state of the block still has an **Owned Exclusively** state) and writing the modified block. The danger is that the Snoop might supply the old block and that the modified block is written into the cache with an erroneous state: **Owned NonExclusively** or **Invalid.**

The revised snooping behavior:

* Read the state (Owned Exclusively), tag, and data
* Raise the Multibus INHIBIT signal
* Obtain mutual exclusion
* Re-read the data
* Place the appropriate word on the Multibus and raise ACK
* Store the new tag (Owned NonExclusively or Invalid)
* Release mutual exclusion
* Lower INHIBIT

The revised behavior of the Cache Controller:

* Read the state (Owned Exclusively), tag, and data
* Obtain mutual exclusion
* Re-read the state
* If the state is still Owned Exclusively, then
  - Form the modified block in the assembly register
  - Write the block into the cache memory
  - Release mutual exclusion
* If the state is not Owned Exclusively, then
  - Release mutual exclusion
  - Handle the write-request as for any Invalid / Owned NonExclusively block

The behavior of a *Read-And-Set* request from the processor has the same effect as a *Write* request, as far as contention is concerned. So, the Cache Controller has a critical section while servicing a *Read-And-Set* request to a **Owned Exclusively** block. This critical section (as for a write-request) includes reading the tag and writing the modified block.

Contention for the Cache Memory Decoders

Contention for the two cache memory decoders occurs only when one of the two controllers is trying to write to the cache memory. Note that the Cache controller only writes into the cache when it has control of the Multibus (implying an inactive Snoop) or when it is in a critical section (above). So, by forcing the Snoop to enter a critical section before writing a new state into the cache memory, we are certain that the two controllers can never simultaneously attempt to write.

12. Appendix 3 - Registers and Signals in the Initial Controller Descriptions

Registers:.

busAddrIn -
the address of the block being read or written by the current bus master.

busAddrOut -
the address of a block to be read/written on the bus.

procAddr -
the address of the word which the processor wishes to read/write.

procAssembly -
the Cache controller's data assembly register.

procDataIn -
the data supplied by the processor in a write-request.

procDataOut -
the data to be supplied to the processor in response to a read-request.

procState, procTag -
the registers containing a block's state and tag, respectively.

snoopAssembly -
the Snoop's data assembly register.

snoopState, snoopTag -
the registers containing a block's state and tag.

Signals used in the controller descriptions:

ackRead - Cache OUTPUT
The servicing of a processor read-request is complete. The requested word has been loaded into the *procDataOut* register.

ackWrite - Cache OUTPUT
The servicing of a processor write-request is complete.

answerDone - Snoop INPUT
The system bus interface has finished supplying the block in the snoopAssembly register in response to a bus read-request.

answerRead - Snoop OUTPUT
Cause the system bus interface to respond to the current read-request on the Multibus, using the block currently in the assembly register.

busMaster - Cache/Snoop INPUT
This cache currently has control of the Multibus. This is the only signal shared by the Cache and Snoop controllers.

busRead - Snoop INPUT
A read-request is being issued on the Multibus; the address of the requested block is in the *busAddrIn* register.

busSpecial - Snoop INPUT
The current Multibus read/write operation is special: *Read-For-Ownership* or *Write-Without-Invalidation*.

busWrite - Snoop INPUT
A write-request is being issued on the Multibus; the address of the requested block is in the *busAddrIn* register.

ioDone - Cache INPUT
The System Bus Interface has completed a block read/write operation.

loadAddrFromProc - Cache OUTPUT
The *busAddrOut* register is to be loaded with an address derived from the current contents of the *procAddr* register.

loadAddrFromTag - Cache OUTPUT
The *busAddrOut* register is to be loaded with an address derived from the current contents of the *procTag* register.

loadDataOut - Cache OUTPUT
Using the address in the *procAddr* register, load the *procDataOut* register with a word taken from the *procAssembly* register.

loadTagFromProc - Cache OUTPUT
The *procTag* register is to be loaded with the tag-value in the *procAddr* register.

modQuad - Cache OUTPUT
The *procAssembly* register is modified, based on the contents of the *procDataIn* register and on the address in the *procAddr* register.

procClear - Cache OUTPUT
The controller is leaving a critical section and clears the semaphore.

procDataAck - Cache INPUT
When high, indicates that a *procDataWrite* completed successfully.

procDataRead - Cache OUTPUT
The block (indexed by the address in the *procAddr* register) is read into the *procAssembly* register.

procDataWrite - Cache OUTPUT
A block is written from the *procAssembly* register into the cache data memory. *procDataAck* is raised if the write is successful.

procHit - Cache INPUT
The tag-value in the *procTag* register matches that in the *procAddr* register, and the value of the *procState* register is other than **Invalid.**

procOK - Cache INPUT
The controller has obtained control of the semaphore. See *procSet.*

procOwned - Cache INPUT
The value of the *procState* register is **Owned Exclusively** or **Owned NonExclusively.**

procOwnedExc - Cache INPUT
The value of the *procState* register is **Owned Exclusively.**

procOwnedNonExc - Cache INPUT
The value of the *procState* register is **Owned NonExclusively.**

procReadReq - Cache INPUT
The processor is issuing a read-request. The controller is to latch the data into the *procDataOut* register and raise *ackRead.*

procSet - Cache OUTPUT
The controller is about to enter a critical section and is trying to set the semaphore. *procOK* will be raised if it succeeds.

procTagAck - Cache INPUT
When high, indicates that a *procTagWrite* completed successfully.

procTagRead - Cache OUTPUT
The tag and status of the block specified by the *procAddr* register are read into the *procTag* and *procState* registers.

procTagWrite - Cache OUTPUT
A block's tag and status are written from the *procTag* and *procState* registers. *procTagAck* is raised if the write completed successfully.

procWriteReq - Cache INPUT
The processor is issuing a write request. The data to be written has been latched in the *procDataIn* register. The controller is to raise *ackWrite* when the operation is completed.

readOwnership - Cache OUTPUT
The block currently in the *procAssembly* register is to be read (for ownership), using the address in the *busAddrOut* register.

readPublic - Cache OUTPUT
The block currently in the *procAssembly* register is to be read (publicly), using the address in the *busAddrOut* register.

releaseBus - Cache OUTPUT
The control of the Multibus is to be released by the System Bus Interface.

requestBus - Cache OUTPUT
The Cache controller wishes to obtain control of the bus. The *busMaster* signal will be raised high when control is obtained. The Cache controller will raise *releaseBus* to release control of the bus.

setInvalid - Snoop OUTPUT
Set the value of the *snoopState* register to **Invalid.**

setOwnedExc - Cache OUTPUT
Set the *procState* register to **Owned Exclusively.**

setOwnedNonExc - Snoop OUTPUT
Set the value of the *snoopState* register to **Owned NonExclusively.**

setUnOwned - Cache OUTPUT
Set the *procState* register to **UnOwned.**

snoopClear - Snoop OUTPUT
The snoop is leaving its critical section and clears the semaphore.

snoopDataRead - Snoop OUTPUT
The data portion of the block indexed by the *busAddrIn* register is read into the *snoopAssembly* register.

snoopInvalid - Snoop INPUT
The entry in the tag-memory indexed by the contents of the *busAddrIn* register is invalid. Either the current value of the *snoopState* register is **Invalid** or the tag-value in the *snoopTag* register doesn't match that in the *busAddrIn* register.

snoopOK - Snoop INPUT
The snoop can enter its critical section (see *snoopSet*).

snoopOwnedExc - Snoop INPUT
The value of the *snoopState* register is **Owned Exclusively** and the tag-value in the *snoopTag* register matches that in the

snoopOwnedNonExc - Snoop INPUT
The value of the *snoopState* register is **Owned NonExclusively** and the tag-value in the

snoopTag register matches that in the *busAddrIn* register.

snoopSet - Snoop OUTPUT
: The snoop wants to enter its critical section is trying to set the semaphore. *snoopOK* will be raised if the Snoop can proceed.

snoopTagAck - Snoop INPUT
: The tag-memory entry was written (see *snoopTagWrite*).

snoopTagRead - Snoop OUTPUT
: The tag-memory is to be read into the *snoopTag* and *snoopState* registers. The line-index is taken from the address in the register.

snoopTagWrite - Snoop OUTPUT
: Write the contents of the *snoopState* and *snoopTag* registers to the tag-memory, raising *snoopTagAck* if the write is successful.

snoopUnOwned - Snoop INPUT
: The value of the *snoopState* register is **UnOwned** and the tag-value in the *snoopTag* register matches that in the *busAddrIn* register.

writeWithoutInv - Cache OUTPUT
: The block currently in the *procAssembly* register is to be written (without invalidation) to memory, using the address in the *busAddrOut* register.

writeWithInv - Cache OUTPUT
: The block currently in the *procAssembly* register is to be written (with invalidation) to memory, using the address in the *busAddrOut* register.

13. Appendix 4 - Initial Specification of the Snoop Controller

```
Home:                                   -- The "busRead" and "busWrite"
                                        -- lines indicate a new request on
                                        -- the bus.  As these lines go high,
                                        -- the address is latched into the
                                        -- "busAddrIn" register.

    if busMaster | ~busRead & ~busWrite | busWrite & busSpecial
            then Home
    if busRead & busSpecial & ~busMaster
            then OwnRead( snoopTagRead, snoopDataRead )
    if busRead & ~busSpecial & ~busMaster
            then PubRead( snoopTagRead, snoopDataRead )
    if busWrite & ~busSpecial & ~busMaster
            then WriteInvalid( snoopTagRead )

OwnRead:                                -- Respond to a "Read-For-Ownership"
                                        -- request on the Multibus.
                                        -- If the block is "Invalid," then
                                        -- ignore this request.
    if snoopInvalid
            then Home
    if snoopUnOwned
            then WriteTag( setInvalid, snoopTagWrite )
    if snoopOwnedExc
            then ReadExc( inhibit, snoopSet, setInvalid )
    if snoopOwnedNonExc
            then OwnReadPub( inhibit, answerRead, setInvalid )

WriteTag:                               -- Wait until the modified tag is
                                        -- written in the cache, then go "Home"
    if snoopTagAck
            then Home
            else WriteTag( snoopTagWrite )

AllDone:                                -- Wait until the bus transaction is
                                        -- completed, then go "Home"
    if answerDone
            then Home
            else AllDone

OwnReadPub:                             -- Handle a "Read-For-Ownership" request
                                        -- to a Owned NonExclusively block.
                                        -- The block's state is made Invalid.
    if snoopTagAck
       then AllDone
       else OwnReadPub( snoopTagWrite )

ReadExc:                                -- Handle a read request to
                                        -- an exclusively owned block.
                                        -- The snoop reads the block only after
                                        -- obtaining ownership of the semaphore.
    if snoopOK
```

```
            then  ReadExc2( snoopDataRead )
            else  ReadExc( snoopSet )
```

Appendix 4 - The initial specification of the Snoop controller (cont.)

```
ReadExc2:
    ReadExc3( answerRead, snoopTagWrite )

ReadExc3:
    if snoopTagAck
            then  AllDone( snoopClear )
            else  ReadExc3( snoopTagWrite )

PubRead:                                -- Handle a Read request
    if snoopInvalid | snoopUnOwned
            then  Home
    if snoopOwnedExc
            then  ReadExc( inhibit, snoopSet, setOwnedNonExc )
    if snoopOwnedNonExc
            then  ReadNotExc( inhibit, snoopDataRead )

ReadNotExc:
    AllDone( answerRead )

WriteInvalid:                   -- Handle a Write-For-Invalidation
                                -- request
    if snoopInvalid
            then  Home
            else  WriteTag( setInvalid )
```

14. Appendix 5 - Initial Specification of the Processor Cache Controller

```
Home:                                   -- Wait for a read/write request from
                                        -- the processor.
    if procReadReq
        then Read( procTagRead, procDataRead )
    if procWriteReq
        then Write( procTagRead, procDataRead )
    if ~procReadReq & ~procWriteReq
        then Home

Read:                                   -- Handle a "read" request from the
                                        -- processor. If we have a cache hit,
                                        -- then we can service the request
                                        -- immediately. Otherwise, we have to
                                        -- read the block from the bus.
    if procHit
        then ReadDone( loadDataOut )
        else Read2( requestBus )

ReadDone:
    Home( ackRead )

Read1:                                  -- Wait to become master of the bus
                                        -- then re-read the block, since its
                                        -- state may have changed during the
                                        -- wait.
    if busMaster
        then Read2( procTagRead, procDataRead )
        else Read1( requestBus )

Read2:                                  -- If we're bumping an owned block, then
                                        -- we first have to write it to memory.
                                        -- Any UnOwned block can be overwritten.
    if procOwned
        then FlushRead( loadAddrFromTag, writeWithoutInv )
        else Read3( loadAddrFromProc, readPublic,
                    setUnOwned, loadTagFromProc )

FlushRead:                              -- Write a bumped block to memory
    if ioDone
        then Read3( loadAddrFromProc, readPublic,
                    setUnOwned, loadTagFromProc )
        else FlushRead

Read3:                                  -- While waiting for the block to be
                                        -- read, update the tag memory.
    if procTagAck
        then Read4
        else Read3( procTagWrite )

Read4:                                  -- Wait until the block requested by
                                        -- the processor has been read, then
                                        -- store the block in the cache.
```

```
    if ioDone
        then Read5( loadDataOut, procDataWrite )
        else Read4

Read5:                              -- Wait until the block has been
                                    -- written into the cache
    if procDataAck
        then Home( ackRead, releaseBus )
        else Read5( procDataWrite )

Write:                              -- Handle a "write" request from the
                                    -- processor
    if procHit & procOwnPrivate
        then WritePriv( procSet )
        else Write1( requestBus )

Write1:                             -- Wait until control of the bus is
                                    -- obtained, then re-read the block.
    if busMaster
        then Write2( procTagRead, procDataRead )
        else Write1( requestBus )

Write2:                             -- We first have to obtain exclusive
                                    -- ownership of the block.  For an
                                    -- Owned NonExclusively block, we write
                                    -- it with invalidation.  Anything else
                                    -- is read for ownership.
                                    -- And don't forget to write any bumped
                                    -- block that we own!
    if procHit & procOwned
        then Write3( loadAddrFromProc, writeWithInv,
                     loadTagFromProc, setOwnedExc )
    if ~procHit & procOwned
        then WriteFlush( loadAddrFromTag, writeWithoutInv )
    if ~procOwned
        then Write3( loadAddrFromProc, readOwnership,
                     loadTagFromProc, setOwnedExc )

Write3:                             -- Wait for the tag-write and the
                                    -- block-read to complete
    if procTagAck
        then Write4
        else Write3( procTagWrite )

Write4:                             -- When the block has been read,
                                    -- modify it (in the assembly register)
    if ioDone
        then Write5( modQuad, ackWrite )
        else Write4

Write5:
    if procDataAck
        then Home( releaseBus )
```

```
            else  Write5( procDataWrite )

WritePriv:                          – Modify an Owned Exclusively block.
                                    – The block is re-read after mutual
                                    – exclusion is obtained.
    if  procOK
            then  WritePriv2( procTagRead, procDataRead )
            else  WritePriv( procSet )

WritePriv2:                         – If the block is still Owned Exclusively,
                                    – then we can update it in place.
                                    – Otherwise, it's a write to a
                                    – Owned NonExclusively or Invalid block.
    if  procHit & procOwnedExc
            then  WritePriv3( modQuad, ackWrite )
    if  procHit & ~procOwnedExc
            then  Write3( loadAddrFromProc, writeWithInv,
                          loadTagFromProc, setOwnedExc )
    if  ~procHit
            then  Write3( loadAddrFromProc, readOwnership,
                          loadTagFromProc, setOwnedExc )

WritePriv3:
    if  procDataAck
            then  Home( procClear )
            else  WritePriv3( procDataWrite )

FlushWrite:                         – Write a bumped block as a result
                                    – of trying to handle a processor
                                    – write request.
    if  ioDone
            then  Write3( loadAddrFromProc, readOwnership,
                          loadTagFromProc, setOwnedExc )
            else  FlushWrite
```

15. Appendix 6 - Signals Used in the Final Controller Descriptions

Memory control signals:

BusAdrA	The A-decoder is driven by the bus-address
BusAdrB	The B-decoder is driven by the bus-address
LoadAData	"AAssembly" is loaded from the A-lines
LoadAState	"AState" is loaded from the A-lines
LoadATags	"ATags" is loaded from the A-lines
LoadBData	"BAssembly" is loaded from the B-lines
LoadBState	"BState" is loaded from the B-lines
LoadBTags	"BTags" is loaded from the B-lines
ProcAdrA	The A-decoder is driven by the processor-address
ProcAdrB	The B-decoder is driven by the processor-address
WrAData	"AAssembly" is written into the data memory
WrAState	"AState" is written into the state memory
WrBData	"BAssembly" is written into the data memory
WrBTags	"BTags" is written into the tag memory

Register control signals:

ModQuad	Multiplex processor-supplied data into "BAssembly"
SetAAdr	Load the "AAdr" register from the A-decoder lines
SetBTags	Load the "BTags" register from the processor-address
SetOwnedExc	Set the "AState" register to OwnedExclusively
SetInvalid	Set the "AState" register to Invalid
SetOwnedNonExc	Set the "AState" register to Owned NonExclusively
SetUnOwned	Set the "AState" register to UnOwned

System bus interface signals:

GetBus	This cache wants to control the bus
HaveBus	This cache controls the bus?
Inhibit	The Multibus INHIBIT line is to be raised
MasterAck	The Multibus read/write request has completed?
MasterRead	Initiate a read operation on the Multibus
MasterWrite	Initiate a write operation on the Multibus
MasterSpecial	Read-For-Ownership or Write-Without-Invalidation
ReleaseBus	Control of the bus is to be relinquished
SlaveAck	The block in "Aassembly" is to be placed on the bus
SlaveRead	There is a read request pending on the bus?
SlaveSpecial	The read/write request is "special"?
SlaveWrite	There is a write request pending on the bus?

Processor interface signals:

LoadDataOut	Load the "ProcDataOut" register from "BAssembly"
ProcAck	The read/write operation is complete
ProcReadReq	The processor has issued a read request?
ProcWriteReq	The processor has issued a write request? Both "ProcReadReq" and "ProcWriteReq" are high if a Read-And-Set operation was issued.

State signals:

ProcHit	The "BTags" match and the "Bstate" is valid?
ProcOwned	The "BState" is Owned Exclusively or Owned NonExclusively?
ProcPublic	The "BState" is UnOwned or Owned NonExclusively?
SnoopHit	The "ATags" match and the "Astate" is valid?

SnoopOwned	The "AState" is Owned Exclusively or Owned NonExclusively?
SnoopPublic	The "AState" is UnOwned or Owned NonExclusively?

Synchronization:

Busy	The cache controller is handling a request
ProcHas	The processor cache-controller is in a critical section
SnoopWants	The snoop is in (or wants to be in) a critical section
WriteCall	The slave write-controller (Write2) is to begin
WriteCycle	High during an allowed write-cycle
WriteReturn	The slave write-controller (Write2) is done

16. Appendix 7 - Final Specification of the Snoop Controller

```
– The SNOOP controller...
–       This controller responds to "private" requests by invalidating any
–       copy of the quad being read or written. If a "read" request to any
–       quad owned by this cache occurs, then the snoop raises the INHIBIT
–       line and supplies the quad. The "SlaveAck" and "Inhibit"
–       signals are used to communicate with the system bus interface,
–       which performs the 4-cycle handshake with the current bus master.
–       The "Inhibit" signal is raised if the Snoop intends to supply the
–       quad in service of the current read-request; the "SlaveAck" signal
–       is raised when the quad is in the "AAssembly" register. The
–       Snoop raises the "SlaveAck" signal for every bus request, indicating
–       that the "SlaveRead" and "SlaveWrite" signals are to be lowered
–       until the next bus request.
–       It is assumed that neither "SlaveRead" nor "SlaveWrite" will be
–       raised while this cache has control of the system bus.

Snoop:

input                                   – Input signals
        ProcHas             = 0..1;
        SlaveSpecial    = 0..1;
        SnoopHit        = 0..1;
        SnoopOwned      = 0..1;
        SnoopPublic     = 0..1;
        WriteCycle      = 0..1;
        SlaveRead       = 0..1;
        SlaveWrite      = 0..1;

output                                  – Output signals
        BusAdrA         = 0..1 := 0;
        BusAdrB         = 0..1 := 0;
        Inhibit         = 0..1 := 0;
        LoadAData       = 0..1 := 0;
        LoadAState      = 0..1 := 0;
        LoadATags       = 0..1 := 0;
        SetInvalid      = 0..1 := 0;
        SetOwnedNonExc      = 0..1 := 0;
        SlaveAck        = 0..1 := 0;
        SnoopWants      = 0..1 := 0;
        WrAState        = 0..1 := 0;

begin

   Home:                                – The "SlaveRead" and "SlaveWrite"
                                        – lines indicate a new request on
                                        – the bus. As these lines go high,
                                        – the address lines from the bus
                                        – interface become valid.
        if WriteCycle
                then Home;
        if ~SlaveRead & ~SlaveWrite & ~WriteCycle
                then Home;
```

```
    if ~SlaveRead & SlaveWrite & SlaveSpecial & ~WriteCycle
            then Home( SlaveAck );

    if ~SlaveWrite & SlaveRead & SlaveSpecial & ~WriteCycle
            then ReadOwn( BusAdrA, LoadAState, LoadATags, LoadAData );

    if ~SlaveWrite & SlaveRead & ~SlaveSpecial & ~WriteCycle
            then ReadPub( BusAdrA, LoadAState, LoadATags, LoadAData );

    if ~SlaveRead & SlaveWrite & ~SlaveSpecial & ~WriteCycle
            then WriteInv( BusAdrA, LoadAState, LoadATags );


ReadOwn:                        -- Respond to a "read for ownership"
                                -- request on the Multibus.
                                -- Wait for the comparison result.
                                -- ASSERT: WriteCycle
    ReadOwn2;

ReadOwn2:                       -- If the block is "Invalid", then
                                -- ignore this request.
                                -- If "Public", then invalidate it.
                                -- If owned, then invalidate the
                                -- block, and inhibit main memory
                                -- from answering the request.
                                -- ASSERT: ~WriteCycle
    if ~SnoopHit
            then Home( SlaveAck );
    if SnoopHit & ~SnoopOwned
            then WriteState( SetInvalid, SnoopWants, SlaveAck );
    if SnoopHit & SnoopOwned & ~SnoopPublic
            then ReadPriv( Inhibit, SnoopWants, SetInvalid );
    if SnoopHit & SnoopOwned & SnoopPublic
            then WriteState( SetInvalid, SnoopWants, SlaveAck, Inhibit );

WriteState:                     -- Write the block's new state from
                                -- the "Astate", then go "Home".
                                -- ASSERT: SnoopWants
    if ProcHas
            then WriteState( SnoopWants );
    if ~ProcHas & ~WriteCycle
            then WriteState( SnoopWants );
    if ~ProcHas & WriteCycle
            then Home( SnoopWants, BusAdrA, BusAdrB, WrAState );

ReadPriv:                       -- On a read request to an exclusively
                                -- owned block, mutual exclusion has
                                -- to be obtained before the data can
                                -- be read.
                                -- ASSERT: SnoopWants
    if ProcHas
            then ReadPriv( SnoopWants );
    if ~ProcHas & ~WriteCycle
```

```
            then ReadPriv( SnoopWants );
        if ~ProcHas & WriteCycle
            then Home( SnoopWants, SlaveAck,
                    BusAdrA, BusAdrB, WrAState, LoadAData );

    ReadPub:                        -- Handle a "public read" request.
                                    -- Wait for "SnoopHit".
                                    -- ASSERT: WriteCycle
        ReadPub2;

    ReadPub2:                       -- Handle this request almost like
                                    -- a "read for ownership".
                                    -- ASSERT: ~WriteCycle
        if ~SnoopHit
            then Home( SlaveAck );
        if SnoopHit & ~SnoopOwned
            then Home( SlaveAck );
        if SnoopHit & SnoopOwned & ~SnoopPublic
            then ReadPriv( Inhibit, SnoopWants, SetOwnedNonExc );
        if SnoopHit & SnoopOwned & SnoopPublic
            then Home( Inhibit, SlaveAck );


    WriteInv:                       -- Handle a "write with invalidation"
                                    -- request by invalidating the quad.
                                    -- Wait for "SnoopHit".
        WriteInv2;

    WriteInv2:
        if SnoopHit
            then WriteState( SlaveAck, SnoopWants, SetInvalid );
        if ~SnoopHit
            then Home( SlaveAck );

where
    Home = 0;
    ReadOwn2 = 8;
    ReadPub2 = 9;
    WriteInv2 = 10;
```

17. Appendix 8 - Specification of the Cache Controller Read FSM

```
-- The CACHE controller: handling a "missed" read
--      The "ReadReq" signal is raised (and the "WriteReq" signal is kept low)
--      if a processor read request could not be handled "automatically"
--      (i.e. if there was a "miss") by the Processor Bus Interface.
--      The "ProcAck" signal is raised when the quad has been loaded into
--      the "ProcDataOut" register.

Read:

input
        HaveBus         = 0..1;
        MasterAck       = 0..1;
        ProcOwned       = 0..1;
        ProcReadReq     = 0..1;
        ProcWriteReq    = 0..1;

output
        Busy            = 0..1 := 1;
        GetBus          = 0..1 := 0;
        LoadAData       = 0..1 := 0;
        LoadATags       = 0..1 := 0;
        LoadBData       = 0..1 := 0;
        LoadBState      = 0..1 := 0;
        LoadBTags       = 0..1 := 0;
        LoadDataOut     = 0..1 := 0;
        MasterRead      = 0..1 := 0;
        MasterSpecial   = 0..1 := 0;
        MasterWrite     = 0..1 := 0;
        ProcAck         = 0..1 := 0;
        ProcAdrA        = 0..1 := 0;
        ProcAdrB        = 0..1 := 0;
        ReleaseBus      = 0..1 := 0;
        SetAAdr         = 0..1 := 0;
        SetBTags        = 0..1 := 0;
        SetUnOwned      = 0..1 := 0;
        WrAData         = 0..1 := 0;
        WrAState        = 0..1 := 0;
        WrBTags         = 0..1 := 0;

begin

    Home:                                   -- Wait for a read request that missed.
                                            -- Nothing is assumed about the state
                                            -- of the B-side registers.
        if ProcReadReq & ~ProcWriteReq
                then Read1( GetBus );
        if ~ProcReadReq
                then Home( ~Busy );
        if ProcWriteReq
                then Home( ~Busy );

    Read1:                                  -- Wait to become master of the bus.
```

```
                                    -- Once master, we can do anything
                                    -- since the Snoop is hibernating.
    if ~HaveBus
        then Read1( GetBus );
    if HaveBus
        then Read2( ProcAdrB, LoadBState, LoadBTags );

Read2:                              -- Wait for the quad to be read.
    Read3;

Read3:                              -- If we're bumping an owned quad, then
                                    -- we first have to write it to memory.
                                    -- Any UnOwned block can be overwritten.
    if ProcOwned
        then FlushRead( ProcAdrA, LoadATags, LoadAData, SetAAdr,
                        MasterWrite, MasterSpecial );
    if ~ProcOwned
        then Read4( SetBTags );

Read4:                              -- Transfer the address to the A-side,
                                    -- and initiate the public read.
    Read5( ProcAdrA, ProcAdrB, WrBTags, LoadATags, SetAAdr, SetUnOwned,
        MasterRead );

Read5:                              -- Wait until the quad requested by
                                    -- the processor has been read, then
                                    -- store the quad in the cache and in
                                    -- the "Bassembly" register.
    if MasterAck
        then Read6( ProcAdrA, ProcAdrB, WrAState, WrAData,
                    LoadBData, ReleaseBus );
    if ~MasterAck
        then Read5;

Read6:
    Read7( LoadDataOut, ProcAck );

Read7:                              -- Wait for "ProcReadReq" to drop.
    Home( ~Busy );

FlushRead:                          -- Write a bumped quad to memory
    if MasterAck
        then Read5( ProcAdrA, ProcAdrB, WrBTags, LoadATags,
                    SetUnOwned, MasterRead );
    if ~MasterAck
        then FlushRead( SetBTags );

where
  Home = 0;
```

18. Appendix 9 - Specification of the Cache Controller Write1 FSM

```
-- The CACHE controller: responding to a processor write request
--      This finite-state machine controls the handling of a processor
--      write request, signaled when "ProcWriteReq" goes high.  The
--      request is satisfied when the quad has been modified and the
--      "ProcAck" signal is raised.
--      The task of responding to write requests has been split between
--      two controllers: a master and a slave.  This controller is the
--      master, and signals the slave to begin by raising "WriteCall".
--      The slave signals completion by raising "WriteReturn".
--      The slave is not invoked if an Owned Exclusively block is involved.

Write1:

input
        ProcHit         = 0..1;
        ProcOwned       = 0..1;
        ProcPublic      = 0..1;
        ProcReadReq     = 0..1;
        ProcWriteReq    = 0..1;
        SnoopWants      = 0..1;
        WriteCycle      = 0..1;
        WriteReturn     = 0..1;

output
        Busy            = 0..1 := 1;
        LoadBData       = 0..1 := 0;
        LoadBState      = 0..1 := 0;
        LoadBTags       = 0..1 := 0;
        LoadDataOut     = 0..1 := 0;
        ModQuad         = 0..1 := 0;
        ProcAck         = 0..1 := 0;
        ProcAdrA        = 0..1 := 0;
        ProcAdrB        = 0..1 := 0;
        ProcHas         = 0..1 := 0;
        SetDataIn       = 0..1 := 0;
        WrBData         = 0..1 := 0;
        WriteCall       = 0..1 := 0;

begin
   Home:                                -- Wait for a processor write request.
        if ProcWriteReq & ~WriteCycle
                then Write( ProcAdrB, LoadBState, LoadBTags, LoadBData );
        if ~ProcWriteReq
                then Home( ~Busy );
        if WriteCycle
                then Home( ~Busy );

   Write:                               -- Wait for the comparison result.
        Write1;

   WriteX:                              -- Wait, but with mutual exclusion(?)
        Write1( ProcHas );
```

```
Write1:                                 -- Handle a "write" request from the
                                        -- processor.
                                        -- We need exclusive (private) ownership
                                        -- if we don't already have it.
                                        -- ASSERT: ~WriteCycle
    if ProcHit & ProcOwned & ~ProcPublic & SnoopWants
        then WriteX( ProcHas, ProcAdrB,
                     LoadBState, LoadBTags, LoadBData );
    if ProcHit & ProcOwned & ~ProcPublic & ~SnoopWants & ~ProcReadReq
        then WritePriv( ProcHas, ModQuad, ProcAck );
    if ProcHit & ProcOwned & ~ProcPublic & ~SnoopWants & ProcReadReq
        then ReadSet( LoadDataOut );
    if ~ProcHit
        then Write2( WriteCall );
    if ~ProcOwned
        then Write2( WriteCall );
    if ProcPublic
        then Write2( WriteCall );

Write2:                                 -- Wait 'til the other FSM is done.
    if WriteReturn
        then Home( ProcAck );
    if ~WriteReturn
        then Write2;

WritePriv:                              -- Finish the job of writing the
                                        -- exclusively owned block.
                                        -- The block had already been modified.
                                        -- Mutual exclusion has been obtained.
                                        -- ASSERT: WriteCycle
    Home( ProcHas, ProcAdrA, ProcAdrB, WrBData );

ReadSet:                                -- Finish the job of reading and
                                        -- modifying an OwnedExc block.
                                        -- ASSERT: WriteCycle
    ReadSet2( ProcHas, ModQuad );

ReadSet2:
    WritePriv( ProcHas, ProcAck );

where
  Home = 0;
```

19. Appendix 10 - The Specification of the Cache Controller Write2 FSM

```
- The CACHE controller: responding to a processor write request (continued)
-       This finite-state machine controls the handling of a processor
-       write request to other than an exclusively owned block.
-       This controller is a slave to the other "write" controller and
-       is activated when the "WriteCall" signal is raised.
-       This controller signals completion by raising the "WriteReturn" signal.

Write2:

input
        HaveBus         = 0..1;
        MasterAck       = 0..1;
        ProcHit         = 0..1;
        ProcOwned       = 0..1;
        ProcReadReq     = 0..1;
        WriteCall       = 0..1;

output
        GetBus          = 0..1 := 0;
        LoadAData       = 0..1 := 0;
        LoadATags       = 0..1 := 0;
        LoadBData       = 0..1 := 0;
        LoadBState      = 0..1 := 0;
        LoadBTags       = 0..1 := 0;
        LoadDataOut     = 0..1 := 0;
        MasterRead      = 0..1 := 0;
        MasterSpecial   = 0..1 := 0;
        MasterWrite     = 0..1 := 0;
        ModQuad         = 0..1 := 0;
        ProcAdrA        = 0..1 := 0;
        ProcAdrB        = 0..1 := 0;
        ReleaseBus      = 0..1 := 0;
        SetAAdr         = 0..1 := 0;
        SetBTags        = 0..1 := 0;
        SetOwnedExc     = 0..1 := 0;
        WrAData         = 0..1 := 0;
        WrAState        = 0..1 := 0;
        WrBData         = 0..1 := 0;
        WrBTags         = 0..1 := 0;
        WriteReturn     = 0..1 := 0;

begin
   Home:                                -- Wait for a signal from the other
                                        -- "write" controller.
        if ~WriteCall
                then Home;
        if WriteCall
                then Write2( GetBus );

   Write2:                              -- Wait 'til we're the bus master,
                                        -- then read the quad (again).
        if HaveBus
```

```
            then  Write3( ProcAdrB, LoadBState, LoadBTags );
        if ~HaveBus
            then  Write2( GetBus );

Write3:                                 -- Wait for the read...
    Write4;

Write4:                                 -- We first have to obtain exclusive
                                        -- ownership of the quad.  If there's
                                        -- a "hit", then the block is UnOwned
                                        -- or OwnedNonExc and we can
                                        -- write-with-invalidation.
                                        -- Otherwise, read-for-ownership.
                                        -- And don't forget to write any bumped
                                        -- quad that we own!
        if ProcHit
            then  Write6( ProcAdrA, ProcAdrB, LoadATags, SetAAdr, LoadAData,
                          WrBTags, MasterWrite, SetOwnedExc );
        if ~ProcHit & ProcOwned
            then  FlushWrite( ProcAdrA, LoadATags, LoadAData, SetAAdr,
                              MasterWrite, MasterSpecial );
        if ~ProcHit & ~ProcOwned
            then  Write5( SetBTags );

Write5:                                 -- For a read-private, we have to
                                        -- transfer the tag through the
                                        -- tag memory to the A-side.
        Write6( ProcAdrA, ProcAdrB, LoadATags, SetAAdr, WrBTags,
            MasterRead, MasterSpecial, SetOwnedExc );

Write6:                                 -- When the quad has been read,
                                        -- send it to the B-side.
        if MasterAck
            then  Write7( ProcAdrA, ProcAdrB, WrAData, WrAState,
                          LoadBData );
        if ~MasterAck
            then  Write6;

Write7:                                 -- Modify the quad in "Bassembly"...
                                        -- (for "ReadAndSet", save the word's
                                        -- original contents before modifying)
        if ~ProcReadReq
            then  Write8( ModQuad );
        if ProcReadReq
            then  ReadSet( LoadDataOut );

ReadSet:
     Write8( ModQuad );

Write8:                                 -- Replace the quad and go home.
    Home( ProcAdrA, ProcAdrB, WrBData, ReleaseBus, WriteReturn );

FlushWrite:                             -- Write a bumped quad as a result
```

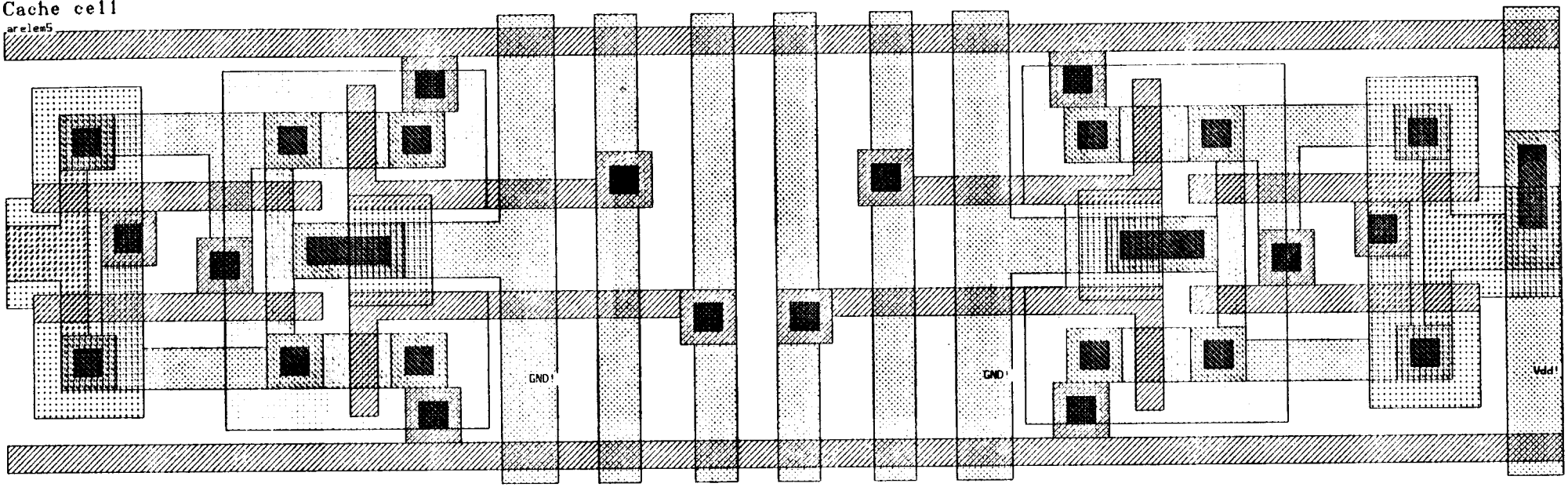
```
                                        -- of trying to handle a processor
                                        -- write request.
        if MasterAck
                then Write6( ProcAdrA, ProcAdrB, LoadATags, SetAAdr,
                             WrBTags, MasterRead, MasterSpecial,
                             SetOwnedExc );
        if ~MasterAck
                then FlushWrite( SetBTags );

where
   Home = 0;
```

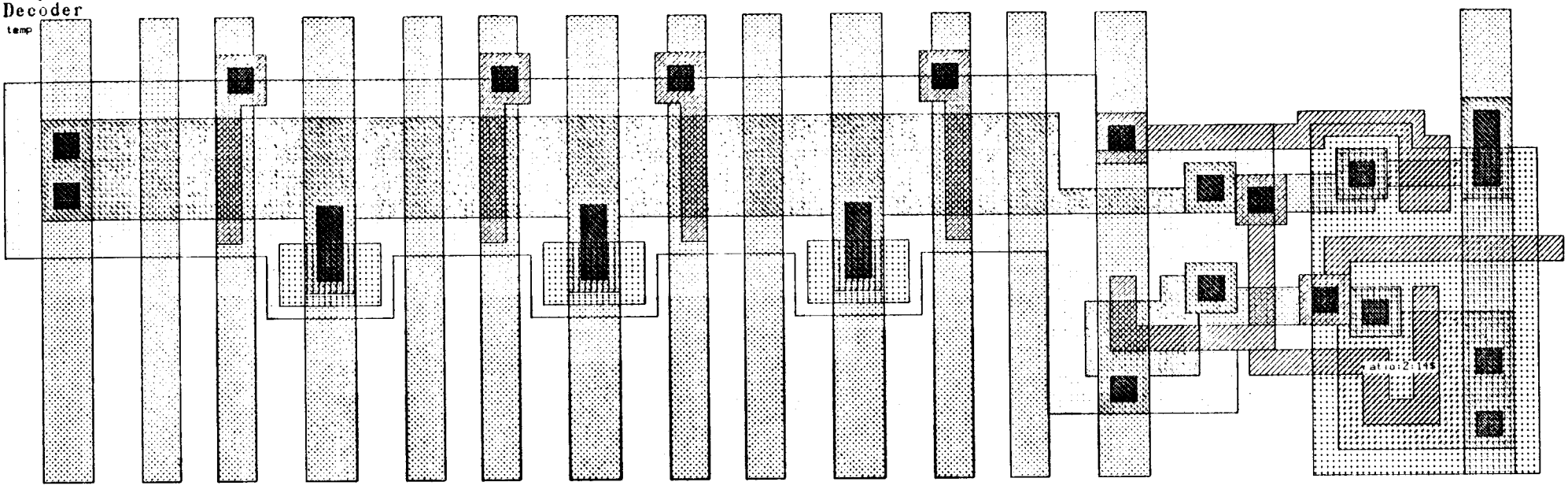
20. Appendix 11 - Clfplots of Important Cells

* Cache Cell
* Decoder
* Decoder Driver
* Assembly Register
* Multiplexor
* Comparator

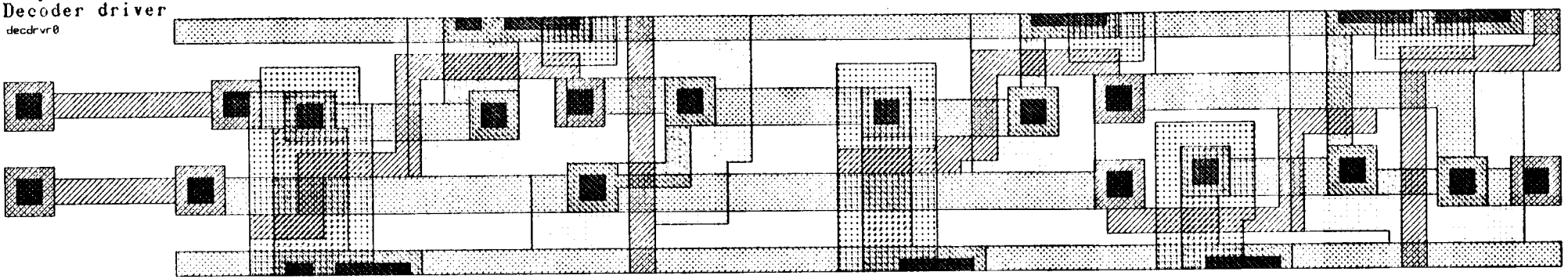
cifplot* Window: -600 4500 -750 16200 --- Scale: 1 micron is 0.0622124 inches (1580x)
Cache cell



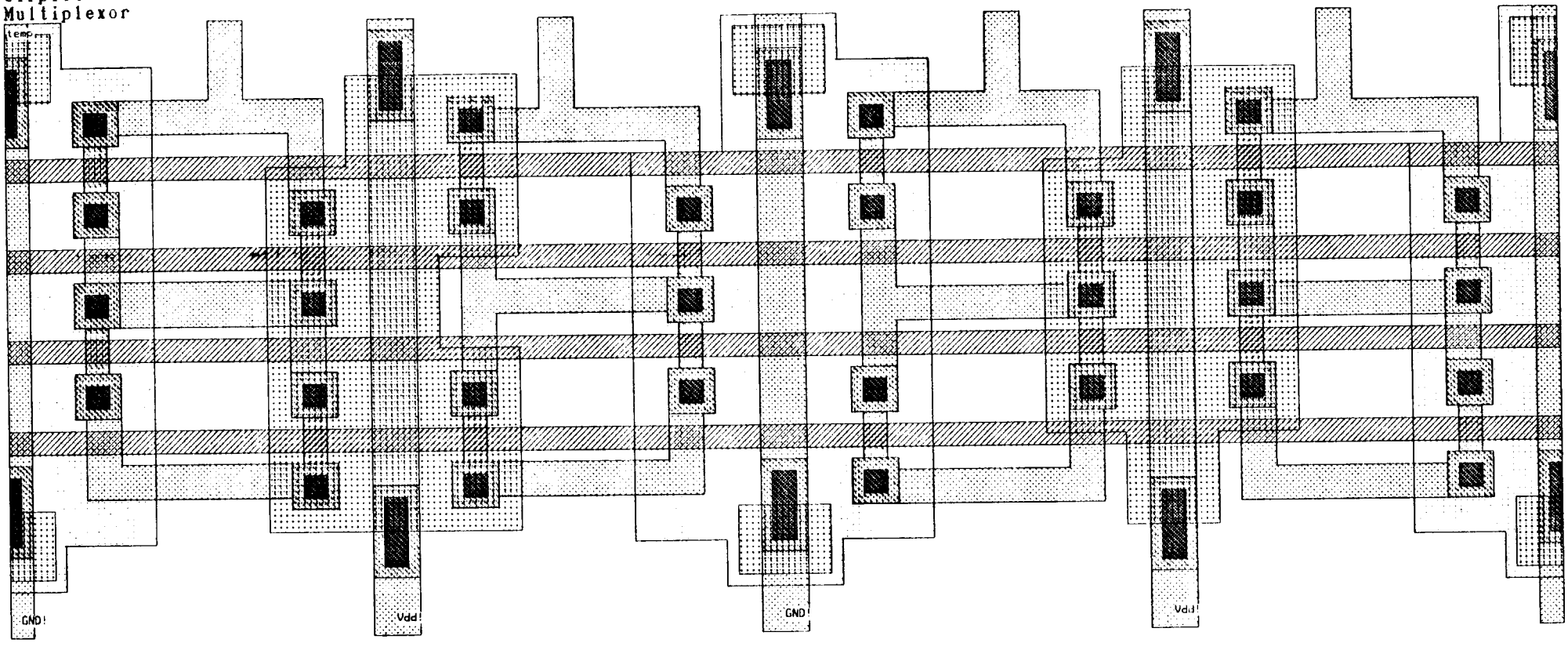
cifplot* Window: -600 4950 -600 18000 --- Scale: 1 micron is 0.0566936 inches (1440x)
Decoder



cifplot* Window: -48150 -45000 141300 160350 --- Scale: 1 micron is 0.0553543 inches (1406x)
Decoder driver



cifplot* Window: -600 7500 -600 19800 --- Scale: 1 micron is 0.0516912 inches (1313x)
Multiplexor



cifplot* Window: -2850 2250 -14400 -3600 --- Scale: 1 micron is 0.0976389 inches (2480x)
Comparator