6.823 Computer System Architecture
Multiprocessor Systems
Problem Set #6

Spring 2002

Students are encouraged to collaborate in groups of up to 3 people. A group needs to hand in only one copy of the solution to a problem set. To facilitate grading, each problem must be stapled separately and your latest group number must appear on each problem (If your group has changed please indicate this). Homework will not be accepted once solutions are handed out. *It may be necessary to make certain assumptions in order to do the following problems. Be sure to explicitly state your assumptions in your write-ups.*

Problem 1: Synchronization

Ben Bitdiddle is developing a DLX-based Symmetric Multi-Processor system (SMP). To implement synchronization primitives, Ben has added the following instructions to the DLX ISA.

EXCH	R2,A(R1)	Atomically exchanges the contents of R2 with the contents at M[<R1>+A]. Temp ← M[<R1>+A]; M[<R1>+A] ← R2; R2 ← Temp
LL	R2,A(R1)	Load Linked. Loads R2 with the contents at M[<R1>+A] and reserves the memory address <R1>+A by storing it in a special link register (R_{link}, not R31). R_{link} ← <R1>+A; R2 ← M[<R1>+A];

SC	A(R1),R2	Store Conditional. Checks if the reservation of the memory address is valid in the link register. If so, the contents of R2 is written to M[<R1>+A] and R2 is set to 1; otherwise no memory store is performed and 0 is written into R2.
		
		if <R_{link}> = <R1>+A then
		    Cancel other processors' reservation on <R1>+A;
		    M[<R1>+A] ← <R2>; R2 ← 1;
		else
		    R2 ← 0;

Problem 1.A **EXCH with LL/SC**

Alyssa P. Hacker tells Ben that he does not need the `EXCH` instruction because it can be implemented using Load Linked (LL) and Store Conditional (SC). Using `LL` and `SC`, write a code sequence equivalent to:

```
EXCH R2,0(R1)
```

Problem 1.B **`SC` vs. `SW`**

Is there any additional cost to the `SC` instruction compared to a standard `SW` instruction on an SMP with an invalidate cache coherence protocol? Please explain considering bus usage and operations that have to be done.

Problem 1.C **Even better with `LL/SC`**

Now consider the following code sequence, implementing a lock (at the beginning of the sequence, `R1` contains the memory address of the mutex):

```
          ADDI R2,R0,#1
try:      EXCH R2,0(R1)
          BNEZ R2,try
```

If you insert the code you wrote from Part B for `EXCH R2,0(R1)` you should still get a working mutex. However, this code can be improved to perform better on an SMP. Give a modification to the new lock code (using `LL` and `SC`), and a scenario in which your modification makes a difference. What advantage is shown here for using `LL/SC` instead of an atomic exchange?

Problem 1.D **Context switching**

Bob Cratchet gives Ben a set of SMP programs to run on the new system. When running just one of Bob's programs, everything works fine. However, when running two or more of Bob's programs together, the machine gives erroneous results. Alyssa informs Ben that she believes `LL/SC` along with the link register does not provide correct synchronization across context switches.

Describe a problematic situation illustrating Alyssa's point. Specifically, think of a situation where a processor is running several tasks that reserve a common memory location. Note that a processor has only one link register (R_{link}).

What fix or fixes can Ben make to provide correct synchronization across context switches?

Problem 2: Snoopy Cache Coherent Shared Memory

In this problem, we will examine an invalidation coherence protocol for write-back caches similar to those employed by the SUN Mbus. As in most invalidation protocols, only a single cache may *own* a modified copy of a cache line at any one time. However, in addition to allowing multiple shared copies of clean data, multiple shared copies of modified data may also exist. (Here, modified data refers to data different from memory. When multiple shared copies of modified data exist, one of the caches *owns* the current copy of the data instead of the memory.) All shared copies are invalidated any time a new modified (write) copy is created.

The MBus transactions with which we are concerned are:

- Coherent Read (**CR**): issued by a cache on a read miss to load a cache line.
- Coherent Read and Invalidate (**CRI**): issued by a cache on a write-allocate after a write miss.
- Coherent Invalidate (**CI**): issued by a cache on a write hit to a block that is in one of the shared states.
- Block Write (**WR**): issued by a cache on the write-back of a cache block.
- Coherent Write and Invalidate (**CWI**): issued by an I/O processor (DMA) on a block write (a full block at a time).

In addition to these primary bus transactions, there is:

- Cache to Cache Intervention (**CCI**): used by a cache to satisfy other caches' read transactions when appropriate. A **CCI** intervenes and overrides the answers normally supplied by memory. Data should be supplied using **CCI** whenever possible for faster response relative to the memory. However, only the cache that ***owns*** the data can respond by **CCI.**

The five possible states of a data block are:

- Invalid (**I**): Block is not present in the cache.
- Clean exclusive (**CE**): The cached data is consistent with memory, and no other cache has it.
- Owned exclusive (**OE**): The cached data is different from memory, and no other cache has it. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.
- Clean shared (**CS**): The data has not been modified by the corresponding CPU since cached. Multiple **CS** copies and at most one **OS** copy of the same data could exist.
- Owned shared (**OS**): The data is different from memory. Other **CS** copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the **OE** state.)

The following questions are to help you check your understanding of the coherence protocol. (You don't need to hand in the answers to these short questions.)

- Explain the differences between **CR, CI,** and **CRI** in terms of their purpose, usage, and the actions that must be taken by memory and by the different caches involved.
- Explain why **WR** is not snooped on the bus.
- Explain the I/O coherence problem that **CWI** helps avoid.

Problem 2.A Where in the Memory System is the Current Value

In the tables on the following page, column 1 indicates the initial state of a certain address X in a cache. Column 2 indicates whether address X is currently cached in any other cache. (The "cached" information is known to the cache controller only immediately following a bus transaction. Thus, the action taken by the cache controller must be independent of this signal, but state transition could depend on this knowledge.) Column 3 enumerates all the available operations on address X, either issued by the CPU (read, write), snooped on the bus (**CR, CRI, CI**. etc), or initiated by the cache itself (replacement). Some state-operation combinations are impossible; you should mark them as such. (See the first table for examples). In columns 6, 7, and 8 (corresponding to this cache, other caches and memory, respectively), **check all possible locations where up-to-date copies of this data block could exist after the operation in column 3 has taken place.** The first table has been completed for you. Make sure the answers in this table make sense to you.

Problem 2.B MBus Cache Block State Transition Table

In this problem, **we ask you to fill out the state transitions in Column 4 and 5.** In column 5, fill in the resulting state after the operation in column 3 has taken place. In column 4, list the necessary MBus transactions that are issued by the cache as part of the transition. Remember, the protocol should be optimized such that data is supplied using **CCI** *whenever possible*, and only the cache that ***owns*** a line should issue **CCI**.

initial state	cached	ops	actions by this cache	final state	this cache	other caches	mem
Invalid	no	none	none	**I**			√
		CPU read	**CR**	**CE**	√		√
		CPU write	**CRI**	**OE**	√		
		replace	none	*Impossible*			
		CR	none	**I**		√	√
		CRI	none	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
Invalid	yes	none	same as above	**I**		√	√
		CPU read		**CS**	√	√	√
		CPU write		**OE**	√		
		replace		*Impossible*			
		CR		**I**		√	√
		CRI		**I**		√	
		CI		**I**		√	
		WR		**I**		√	√
		CWI		**I**			√

initial state	cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**			
		CPU read					
		CPU write					
		replace					
		CR		**CS**			
		CRI					
		CI					
		WR					
		CWI					

initial state	cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedExclusive	no	none	none	**OE**			
		CPU read					
		CPU write					
		replace					
		CR		**OS**			
		CRI					
		CI					
		WR					
		CWI					

initial state	cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanShared	no	none	none	**CS**			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					
cleanShared	yes	none	same as above				
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					

initial state	cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedShared	no	none	none	**OS**			
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					
ownedShared	yes	none	same as above				
		CPU read					
		CPU write					
		replace					
		CR					
		CRI					
		CI					
		WR					
		CWI					

Problem 3: Directory-based cache coherence protocol

In this problem we consider a CCDSM (cache-coherent distributed shared memory) system. The system consists of a number of sites connected by an interconnection network. As shown in Figure 1, each *site* has a processor, an L1 cache, a shared memory, and a protocol processing component (PP). The PP implements global cache coherence using a directory-based cache coherence protocol. For each cache line, we maintain a cache state to specify the current coherence state of the cache line. For each memory block, we maintain a directory entry to record the sites that are currently caching that block. For every global address, there is a *home* site where the physical memory and directory entry is maintained. Assume that the home site can be determined by the global address using its most significant bits.

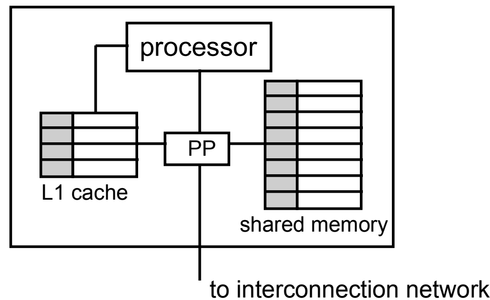

Figure 1: Site Configuration

A simple full-map directory structure is used. Each directory entry keeps a complete record of the sites that are sharing the memory block. The most common implementation keeps a bit-vector in each directory entry. The bit-vector has one bit for each site, indicating if a valid copy of the memory block is cached at that site. A dirty bit is also needed to indicate if the block has been modified. Unlike bus-based snoopy protocols, the directory-based protocol does not rely on broadcast to invalidate stale copies. Instead, because the locations of shared copies are known, cache coherence can be achieved by sending point-to-point protocol messages to only the sites that have cached the accessed memory block. The elimination of broadcast overcomes the major limitation on scaling cache coherent systems to parallel machines with a large number of processors.

The PPs are responsible for servicing memory access instructions, processing protocol messages, and maintaining cache line states and home directory states. When the processor issues a memory access instruction, the PP checks the addressed cache line's state. If the cache state shows that the instruction cannot be completed locally, the PP suspends the instruction, and sends a protocol request message to the corresponding home site. When this request arrives at the home site, the PP at the home site checks the home directory state, and sends a protocol reply message back to the requesting site to supply the requested data and/or exclusive ownership (in order to do this, the home site may need to obtain, from a remote site, the most up-to-date data and/or exclusive ownership if the memory block has been modified; or invalidate all the shared copies if the memory block is shared and the protocol message received is a store-request). At the requesting site, when the PP receives the protocol reply message, it resumes the suspended memory access instruction.

We make the following assumptions about the interconnection network:

- Message passing is reliable, and free from deadlock, livelock and starvation. In other words, the transfer latency of any protocol message is finite.
- Message passing is FIFO. That is, protocol messages with the same source and destination sites are always received in the same order as that in which they were issued.

Memory instructions: The basic memory access instructions are load and store. A load instruction reads the most up-to-date value of a given location, while a store instruction writes a specific value to a given location. We also need to consider cache replacement operations. A replace operation invalidates a cache line and, if the cache line has been modified, writes the modified data back to memory.

Both load and store are processor-issued instructions. Replace, on the other hand, is normally caused by a load/store instruction when a read/write miss leads to an associative conflict in the cache. In this situation, the load/store instruction cannot be processed before the replace (which is a "side-effect" of the load/store instruction) operation is completed. A cache line in a transient state cannot be replaced.

Cache states: For each cache line, there are 4 possible states:

- C-invalid: The accessed data is not resident in the cache.
- C-shared: The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified: The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
- C-transient: The accessed data is in a *transient* state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home directory states: For each memory block, there are 4 possible states:

- H-uncached: The memory block is not cached by any site. Memory has the most up-to-date data.
- H-shared[*S*]: The memory block is shared by the sites specified in *S* (*S* is a set of sites). The data in memory is also valid.
- H-modified[*m*]: The memory block is exclusively cached at site *m*, and has been modified at that site. Memory does not have the most up-to-date data.
- H-transient: The memory block is in a transient state (for example, the home site has just sent a protocol request to the modified site in order to obtain the most up-to-date data, but has not received the corresponding protocol reply). A counter, *count*, is needed when H-transient represents a transient state in which the home site is waiting for the acknowledgments to the invalidation requests it has issued.

Protocol messages: There are 12 different protocol messages, which are summarized in the following table (their meaning will become clear later). A protocol message includes the message type, the accessed memory address and, if necessary, the requested or written-back data. A protocol message usually comes in a *request* and *reply* pair. However, there are two

exceptions: write-back and retry. Write-back writes a modified cache line back to the main memory. This is a one-way message that does not need a reply (compared with protocol requests, this saves one reply message). Retry is a NAK (negative acknowledgment) message which indicates that something abnormal has happened, and some request cannot be processed and should be retried later. This is possible since the parallel system runs in a distributed way so that operations (e.g. sending a protocol message from one site to another) cannot be treated as atomic operations.

No.	Message Type	Includes data?
1	load-request	no
2	store-request	no
3	shared-copy-request	no
4	exclusive-copy-request	no
5	invalidate-request	no
6	load-reply	yes
7	store-reply	yes
8	shared-copy-reply	yes
9	exclusive-copy-reply	yes
10	invalidate-reply	no
11	write-back	yes
12	retry	no

The behavior of the PP can be defined by two finite state machines: one for cache line states, the other for home directory states. In this problem, we consider a very simple invalidation-based cache coherence protocol that implements the *sequential consistency* memory model. A brief (but neither formal nor complete) description is given below to help you understand the protocol.

Cache state transitions:

When the processor issues a load instruction,

- If the cache state is C-shared or C-modified, the PP supplies the processor the data from the cache. The cache state is not changed.
- If the cache state is C-invalid, the PP suspends the load instruction, and sends a load-request to the accessed memory's home site to request the data. The cache state is changed to C-transient. Later when the corresponding load-reply arrives, the PP places the data in the cache, changes the cache state to C-shared, and resumes the suspended load instruction.

When the processor issues a store instruction,

- If the cache state is C-modified, the PP allows the processor to write to the cache. The cache state is not changed.
- If the cache state is C-invalid or C-shared, the PP suspends the store instruction, and sends a store-request to the accessed memory's home site to request the data and exclusive ownership. The cache state is changed to C-transient. Later when the

corresponding store-reply arrives, the PP places the data in the cache, changes the cache state to C-modified, and resumes the suspended store instruction.

When a replace operation happens,

- If the cache state is C-shared, the PP simply changes the cache state to C-invalid.
- If the cache state is C-modified, the PP sends a write-back message to the home site to write the modified data to memory, and changes the cache state to C-invalid.

Home directory state transitions:

When a load-request from site k arrives at the home site,

- If the home directory state is H-uncached, the PP sends a load-reply to site k to supply the requested data. The directory state is changed to H-shared[S], where $S = \{k\}$.
- If the home directory state is H-shared[S], the PP sends a load-reply to site k to supply the requested data. The directory state is changed to H-shared[S'], where $S' = S \cup \{k\}$.
- If the home directory state is H-modified[m], the PP sends a shared-copy-request to site m in order to obtain the most up-to-date data. The directory state is changed to H-transient. Later when the corresponding shared-copy-reply arrives at the home site, the PP updates memory, sends a load-reply to site k to supply the requested data, and then changes the directory state to H-shared[S], where $S = \{m, k\}$.

When a store-request from site k arrives at the home site,

- If the home directory state is H-uncached, the PP sends a store-reply to site k to supply the requested data and exclusive ownership. The directory state is changed to H-modified[k].
- If the home directory state is H-shared[S], the PP sends an invalidate-request to each of the sites specified in S. The directory state is then changed to H-transient, with a dedicated counter initialized to the number of invalidate-requests that have been issued. Later when all the invalidations have been acknowledged, the PP sends a store-reply to site k, and changes the directory state to H-modified[k].
- If the home directory state is H-modified[m], the PP sends a exclusive-copy-request to site m in order to obtain the most up-to-date data and exclusive ownership. The directory state is then changed to H-transient. Later when the corresponding exclusive-copy-reply is received, the PP sends a store-reply to site k, and changes the directory state to H-modified[k].

When a write-back from site m arrives at the home site,

- If the home directory state is H-modified[m] or H-transient, the PP updates the memory with the write-back data, and changes the directory state to H-uncached.

*Note: The cache state transitions described above are for each physical address at the granularity of the cache line size.

Problem 3.A — Cache State Transitions

Table 1 shows the cache state transitions of the protocol (note that some tricky transitions are intentionally ignored here). The "current state" is the current cache line state. The "event

received" is the event that the PP receives, which can be a load/store instruction, a replace operation, or a protocol message issued from the home site. The "next state" is the next cache line state after the PP processes the received event. The "action" is what the PP must do when processing the received event. This usually includes generating some new protocol message, placing some data in the cache, and so on.

Complete Table 1.

Problem 3.B **Directory State Transitions**

Table 2 shows the home directory state transitions of the protocol (note that some tricky transitions are intentionally ignored here). The "current state" is the current home directory state. The "message received" is the protocol message that the PP receives. The "next state" is the next directory state after the PP processes the received message. The "action" is what the PP must do when processing the received event. This usually includes generating some new protocol message(s), updating memory with the most up-to-date data, and so on. We use *k* to represent the site that issued the received message. For H-transient state, we use *j* to represent the site that issued the *original* protocol request (load-request/store-request).

Complete Table 2.

Problem 3.C **Protocol Understanding**

Consider the situation in which the home site sends an exclusive-copy-request to a site. This can only happen when the home directory shows that the modified copy resides at that site. The home site intends to obtain the most up-to-date data and exclusive ownership, and then supply them to another site that has issued a store-request. In Table 1, the last row (line 19) specifies the PP behavior when the current cache state is C-transient (not C-modified) and an exclusive-copy-request is received.

Give a simple scenario that causes this situation. You should explain your answer clearly.

Problem 3.D **Non-FIFO Network**

FIFO message passing is a necessary assumption for the correctness of the protocol. Assume now that the network is non-FIFO. Give a simple scenario that shows how the protocol fails.

No.	Current State	Event Received	Next State	Action
1	C-invalid	load	C-transient	load-request -> home
2	C-invalid	store		
3	C-invalid	invalidate-request		
4	C-invalid	shared-copy-request		
5	C-invalid	exclusive-copy-request		
6	C-shared	load		processor reads cache
7	C-shared	store		
8	C-shared	replace		nothing
9	C-shared	invalidate-request		
10	C-modified	load		
11	C-modified	store		
12	C-modified	replace		
13	C-modified	shared-copy-request		
14	C-modified	exclusive-copy-request		
15	C-transient	load-reply		data -> cache, processor reads cache
16	C-transient	store-reply		
17	C-transient	invalidate-request		
18	C-transient	shared-copy-request		
19	C-transient	exclusive-copy-request		

Table 1: Cache State Transitions

No.	Current State	Message Received	Next State	Action
1	H-uncached	load-request	H-shared[{k}]	load-reply -> k
2	H-uncached	store-request	H-modified[k]	
3	H-shared[S]	load-request	H-shared[$S \cup \{k\}$]	
4	H-shared[S]	store-request		
5	H-modified[m]	load-request		
6	H-modified[m]	store-request		
7	H-modified[m]	write-back		data -> memory
8	H-transient	load-request		
9	H-transient	store-request		
10	H-transient	write-back		
11	H-transient[count > 1]	invalidate-reply	H-transient[--count]	nothing
12	H-transient[count = 1]	invalidate-reply		
13	H-transient	shared-copy-reply		
14	H-transient	exclusive-copy-reply		

Table 2: Home Directory State Transitions

Problem 4: Relaxed Memory Models

Lem E. Tweakit has started a software company called Compilers"R"Us® (CRU) whose product is a compiler for shared memory multi-processor machines. CRU would like to present a sequentially consistent memory model to programmers, but many of the machines supported by the CRU compiler use relaxed memory models. Lem has hired Ben Bitdiddle and Alyssa P. Hacker to solve this problem.

Ben thinks that they should simply write a compiler pass that inserts the equivalent of a MIPS `SYNC` instruction after every memory operation. The `SYNC` instruction guarantees that all loads and stores initiated before the `SYNC` will be seen before any load or store initiated after it. Alyssa disagrees; she thinks that even though Ben's scheme will produce code that operates correctly, the performance would be abysmal, since memory barriers are very expensive operations. She claims that they can do better by addressing each relaxed memory model separately, and using finer-grain memory barrier instructions.

In this problem, you will help Ben and Alyssa convert code written under a sequential consistency assumption so that it operates correctly on machines that support relaxed memory models. You have at your disposal the following four memory barrier instructions:

- $\texttt{Membar}_{rr}$ guarantees that all read operations initiated before the $\texttt{Membar}_{rr}$ will be seen before any read operation initiated after it.
- $\texttt{Membar}_{rw}$ guarantees that all read operations initiated before the $\texttt{Membar}_{rw}$ will be seen before any write operation initiated after it.
- $\texttt{Membar}_{wr}$ guarantees that all write operations initiated before the $\texttt{Membar}_{wr}$ will be seen before any read operation initiated after it.
- $\texttt{Membar}_{ww}$ guarantees that all write operations initiated before the $\texttt{Membar}_{ww}$ will be seen before any write operation initiated after it.

As memory barrier instructions are expensive, you should use one only when necessary. Each of the relaxed memory models you are asked to consider is summarized below, but you should refer to Lecture 19 or Chapter 8.6 of the text for details.

To help you along, here is Alyssa's explanation of how the compiler should compile code for a machine using total store ordering (TSO), so that sequential consistency is preserved (in TSO, a read may complete before a write that is earlier in program order if the read and write are to different addresses):

> The compiler needs to insert a $\texttt{Membar}_{wr}$ after a write if the next memory instruction that occurs after the write may be a read.

Problem 4.A **Partial Store Ordering**

In partial store ordering (PSO), a read or a write may complete before a write that is earlier in program order if they are to different addresses. Explain how the compiler should transform code for a machine using PSO so that sequential consistency is preserved.

Problem 4.B **Weak Ordering**

In weak ordering (WO), a read or a write must complete before any synchronization operation following it in program order, and a synchronization operation must complete before any reads or writes following it. Explain how the compiler should transform code for a machine using WO so that sequential consistency is preserved. You may assume that the compiler is able to identify synchronization operations.

Problem 4.C **Release Consistency**

Release consistency (RC) distinguishes between *acquire* and *release* synchronization operations. An *acquire* must complete before any reads or writes following it in program order, while a read or a write before a *release* must complete before the *release*. Explain how the compiler should transform code for a machine using RC so that sequential consistency is preserved. You may assume that the compiler is able to identify *acquire* and *release* operations. *Acquire* and *release* are special operations that cannot be reordered with respect to memory barriers.