Problem M4.3: Sequential Consistency

For this problem we will be using the following sequences of instructions. These are small programs, each executed on a different processor, each with its own cache and register set. In the following **R** is a register and **X** is a memory location. Each instruction has been named (e.g., B3) to make it easy to write answers.

Assume data in location X is initially 0.

Processor A	Processor B	Processor C
A1: ST X, 1	B1: R := LD X	C1: ST X, 6
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

For each of the questions below, please circle the answer and provide a short explanation assuming the program is executing under the SC model. **No points will be given for just circling an answer!**

Problem M4.3.A

Can X hold value of 4 after all three threads have completed? Please explain briefly.

Yes / No

Problem M4.3.B

Can X hold value of 5 after all three threads have completed?

Yes / No

Problem M4.3.C

Can X hold value of 6 after all three threads have completed?

Yes / No

Problem M4.3.D

For this particular program, can a processor that reorders instructions but follows local dependencies produce an answer that cannot be produced under the SC model?

Yes / No

Problem M4.4: Synchronization Primitives

One of the common instruction sequences used for synchronizing several processors are the LOAD RESERVE/STORE CONDITIONAL pair (from now on referred to as LdR/StC pair). The LdR instruction reads a value from the specified address and sets a local reservation for the address. The StC attempts to write to the specified address provided the local reservation for the address is still held. If the reservation has been cleared the StC fails and informs the CPU.

Problem M4.4.A

Describe under what events the local reservation for an address is cleared.

Problem M4.4.B

Is it possible to implement LdR/StC pair in such a way that the memory bus is not affected, i.e., unaware of the addition of these new instructions? Explain

Problem M4.4.C

Give two reasons why the LdR/StC pair of instructions is preferable over atomic read-test-modify instructions such as the TEST&SET instruction.

Problem M4.4.D

LdR/StC pair of instructions were conceived in the context of snoopy busses. Do these instructions make sense in our directory-based system in Handout #11? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

Problem M4.5: Implementing Directories

Ben Bitdiddle is implementing a directory-based cache coherence invalidate protocol for a 64-processor system. He first builds a smaller prototype with only 4 processors to test out the cache coherence protocol described in Handout #11. To implement the list of sharers, **S**, kept by **home**, he maintains a bit vector per cache block to keep track of all the sharers. The bit vector has one bit corresponding to each processor in the system. The bit is set to one if the processor is caching a shared copy of the block, and zero if the processor does not have a copy of the block. For example, if Processors 0 and 3 are caching a shared copy of some data, the corresponding bit vector would be `1001.`

Problem M4.5.A

The bit vector worked well for the 4-processor prototype, but when building the actual 64-processor system, Ben discovered that he did not have enough hardware resources. Assume each **cache block is 32 bytes**. What is the overhead of maintaining the sharing bit vector for a 4-processor system, as a **fraction of data storage bits**? What is the overhead for a 64-processor system, as a **fraction of data storage bits**?

Overhead for a 4-processor system: _____

Overhead for a 64-processor system: _____

Problem M4.5.B

Since Ben does not have the resources to keep track of all potential sharers in the 64-processor system, he decides to limit **S** to keep track of only 1 processor using its 6-bit ID as shown in Figure M4.5-A (**single-sharer scheme**). When there is a load [`C2P_Req(a) S`] request for a shared cache block, Ben invalidates the existing sharer to make room for the new sharer (home sends a invalidate request `[P2C_Req(a) I]` to the existing sharer, the existing sharer sends an invalidate response `[C2P_Rep(a) I]` to home, home replaces the exiting sharer's ID with the new sharer's ID and sends the load response `[P2C_Rep(a) I S]` to the new sharer).

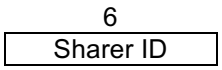


Figure M4.5-A

Consider a 64-processor system. To determine the efficiency of the bit-vector scheme and single-sharer scheme, **fill in the number of invalidate-requests** that are generated by the protocols for each step in the following two sequences of events. Assume cache block **B** is uncached initially for both sequences.

Sequence 1	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**		
Processor #0 reads **B**		

Sequence 2	**bit-vector scheme # of invalidate-requests**	**single-sharer scheme # of invalidate-requests**
Processor #0 reads **B**	0	0
Processor #1 reads **B**		
Processor #2 writes **B**		

Problem M4.5.C

Ben thinks that he can improve his original scheme by adding an extra "**global bit**" to **S** as shown in Figure M4.5-B (**global-bit scheme**). The global bit is set when there is more than 1 processor sharing the data, and zero otherwise.

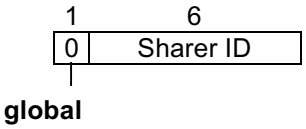


Figure M4.5-B

When the global bit is set, home stops keeping track of a specific sharer and assumes that all processors are potential sharers.

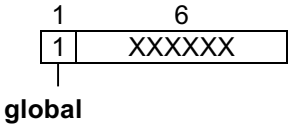


Figure M4.5-C

Consider a 64-processor system. To determine the efficiency of the global-bit scheme, **fill in the number of invalidate-requests** that are generated for each step in the following two sequences of events. Assume cache block **B** is uncached initially for both sequences.

Sequence 1	**global-bit scheme # of invalidate-requests**
Processor #0 reads **B**	0
Processor #1 reads **B**	
Processor #0 reads **B**	

Sequence 2	**global-bit scheme # of invalidate-requests**
Processor #0 reads **B**	0
Processor #1 reads **B**	
Processor #2 writes **B**	

Problem M4.6: Tracing the Directory-based Protocol [? Hours]

For the problem we will be using the following sequences of instructions. These are small programs, each executed on a different processor, each with its own cache and register set. In the following **R** is a register and **X** is a memory location. Each instruction has been named (e.g., B3) to make it easy to write answers.

Assume data in location X is initially 0.

Processor A	Processor B	Processor C
A1: ST X, 1	B1: R := LD X	C1: ST X, 6
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

These questions relate to the directory-based protocol in Handout #11 (as well as Lecture 23). Unless specified otherwise, assume all caches are initially empty and *no voluntary responses are sent (i.e. responses are sent only on receiving a request).*

Problem M4.6.A

Suppose we execute Program A, followed by Program B, followed by Program C and all caches are initially empty. Write down the sequence of messages that will be generated. We have omitted ADD instructions because they cannot generate any messages. EO indicates the global execution order.

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1	1	<M,A,Req,x,M> <A,M,Rep,x,I,M,0>	B1	4		C1	8	
A2	2		B3	5		C2	9	
A4	3		B4	6		C4	10	
			B6	7				

How many messages are generated? _____

Problem M4.6.B

Is there an execution sequence that will generate even fewer messages? Fill in the EO columns to indicate the global execution order. Also, fill in the messages.

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1			B1			C1		
A2			B3			C2		
A4			B4			C4		
			B6					

How many messages are generated? _____

Problem M4.6.C

Can the number of messages in Problem M4.6.B be decreased *by using voluntary responses*? Explain.

Problem M4.6.D

What is the execution sequence that generates the most messages *without any voluntary responses*? Fill in the global execution order (EO) and the messages generated. Partial credit will be given for identifying a bad, but not necessarily the worst sequence.

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1			B1			C1		
A2			B3			C2		
A4			B4			C4		
			B6					

How many messages are generated? _____

Problem M4.7: Snoopy Cache Coherent Shared Memory

In this problem, we investigate the operation of the snoopy cache coherence protocol in Handout #12.

The following questions are to help you check your understanding of the coherence protocol.

- 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 M4.7.A — Where in the Memory System is the Current Value

In Table M4.7-1, M4.7-2, and M4.7-3, 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** and ignore column 4 and 5 for now. Table M4.7-1 has been completed for you. Make sure the answers in this table make sense to you.

Problem M4.7.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**.

Problem M4.7.C **Adding atomic memory operations to MBus**

We have discussed the importance of atomic memory operations for processor synchronization. In this problem you will be looking at adding support for an atomic fetch-and-increment to the MBus protocol.

Imagine a dual processor machine with CPUs A and B. Explain the difficulty of CPU A performing fetch-and-increment(x) when the most recent copy of x is cleanExclusive in CPU B's cache. You may wish to illustrate the problem with a short sequence of events at processor A and B.

Fill in the rest of the table below as before, indicating state, next state, where the block in question may reside, and the CPU A and MBus transactions that would need to occur atomically to implement a fetch-and-increment on processor A.

State	other cached	ops	actions by this cache	next state	this cache	other caches	mem
Invalid	yes	read					
		write					

initial state	other 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	other 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					

Table M4.7-1

initial state	other 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	other 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					

Table M4.7-2

initial state	other 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					

Table M4.7-3

Problem M4.8: Snoopy Cache Coherent Shared Memory

This problem improves the snoopy cache coherence protocol presented in Handout #12. As a **review** of that protocol:

> When multiple shared copies of a *modified* data block exist, one of the caches *owns* the current copy of the data block instead of the memory (the owner has the data block in the OS state). When another cache tries to retrieve the data block from memory, the owner uses *cache to cache intervention* (CCI) to supply the data block. CCI provides a faster response relative to memory and reduces the memory bandwidth demands. However, when multiple shared copies of a *clean* data block exist, there is no owner and CCI is *not* used when another cache tries to retrieve the data block from memory.

To enable the use of CCI when multiple shared copies of a *clean* data block exist, we introduce a new cache data block state: *Clean owned shared* (COS). This state can only be entered from the clean exclusive (CE) state. The state transition from CE to COS is summarized as follows:

initial state	other cached	ops	actions by this cache	final state
cleanExclusive (CE)	no	**CR**	**CCI**	**COS**

There is no change in cache bus transactions but a slight modification of cache data block states. Here is a summary of the possible cache data block states (**differences from problem set highlighted in bold**):

- 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. **This cache is responsible for supplying this data instead of memory when other caches request copies of this data.**
- 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.)
- **Clean owned shared (COS): The cached data is consistent with 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 CE state.)**

Problem M4.8.A

Fill out the state transition table for the new COS state:

initial state	other cached	ops	actions by this cache	final state
COS	yes	none	none	**COS**
		CPU read		
		CPU write		
		replace		
		CR		
		CRI		
		CI		
		WR		
		CWI		

Problem M4.8.B

The COS protocol is not ideal. Complete the following table to show an example sequence of events in which multiple shared copies of a clean data block (*block B*) exist, but CCI is *not* used when another cache (*cache 4*) tries to retrieve the data block from memory.

cache transaction	source for data	state for data block B			
		cache 1	cache 2	cache 3	cache 4
0. *initial state*	**—**	**I**	**I**	**I**	**I**
1. cache 1 reads data block B	**memory**	**CE**	**I**	**I**	**I**
2. cache 2 reads data block B	**CCI**	**COS**	**CS**	**I**	**I**
3. cache 3 reads data block B	**CCI**	**COS**	**CS**	**CS**	**I**
4.					
5.					

Problem M4.8.C

As an alternative protocol, we could eliminate the CE state entirely, and transition directly from I to COS when the CPU does a read and the data block is not in any other cache. This modified protocol would provide the same CCI benefits as the original COS protocol, but its performance would be worse. **Explain the advantage of having the CE state.** You should not need more than one sentence.

Problem M4.9: Snoopy Caches

This part explores multi-level caches in the context of the bus-based snoopy protocol discussed in Lecture 22 (2011). Real systems usually have at least two levels of cache, smaller, faster L1 cache near the CPU, and the larger but slower L2. The two caches are usually inclusive, that is, any address in L1 is required to be present in L2. L2 is able to answer every snooper inquiry immediately but usually operates at 1/2 to 1/4th the speed of CPU-L1 interface. For performance reasons it is important that snooper steals as little bandwidth as possible from L1, and does not increase the latency of L2 responses.

Problem M4.9.A

Consider a situation when the L2 cache has a cache line marked Sh, and an ExReq comes on the bus for this cache line. The snooper asks both L1 and L2 caches to invalidate their copies but responds OK to the request, even before the invalidations are complete. Suppose the CPU ends up reading this value in L1 before it is truly discarded. What must the cache and snooper system do to ensure that sequential consistency is not violated here?

Hint: Consider how much processing can be performed safely on the following sequences after an invalidation request for x has been received

Ld x; Ld y; Ld x

Ld x; St y; Ld x

Problem M4.9.B

Consider a situation when L2 has a cache line marked Ex and a ShReq comes on the bus for this cache line. What should the snooper do in this case, and why?

Problem M4.9.C

When an ExReq message is seen by the snooper and there is a Wb message in the C2M queue waiting to be sent, the snooper replies ***retry***. If the cache line is about to be modified by another processor, why is it important to first write back the already modified cache line? Does your answer change if cache lines are restricted to be one word? Explain.

Problem M4.10: Relaxed Memory Models

Consider a system which uses Weak Ordering, meaning that a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies.

Our processor has four fine-grained memory barrier instructions:

- **MEMBAR$_{RR}$** guarantees that all read operations initiated before the MEMBAR$_{RR}$ will be seen before any read operation initiated after it.
- **MEMBAR$_{RW}$** guarantees that all read operations initiated before the MEMBAR$_{RW}$ will be seen before any write operation initiated after it.
- **MEMBAR$_{WR}$** guarantees that all write operations initiated before the MEMBAR$_{WR}$ will be seen before any read operation initiated after it.
- **MEMBAR$_{WW}$** guarantees that all write operations initiated before the MEMBAR$_{WW}$ will be seen before any write operation initiated after it.

We will study the interaction between two processes on different processors on such a system:

P1	P2
`P1.1: LW R2, 0(R8)`	`P2.1: LW R4, 0(R9)`
`P1.2: SW R2, 0(R9)`	`P2.2: SW R5, 0(R8)`
`P1.3: LW R3, 0(R8)`	`P2.3: SW R4, 0(R8)`

We begin with following values in registers and memory (same for both processes):

register/memory	Contents
R2	0
R3	0
R4	0
R5	8
R8	0x01234567
R9	0x89abcdef
M[R8]	6
M[R9]	7

After both processes have executed, is it possible to have the following machine state? Please circle the correct answer. If you circle **Yes**, please provide sequence of instructions that lead to the desired result (one sequence is sufficient if several exist). If you circle **No**, please explain which ordering constraint prevents the result.

Problem M4.10.A

Memory	contents
M[R8]	7
M[R9]	6

Yes **No**

Problem M4.10.B

memory	Contents
M[R8]	6
M[R9]	7

Yes **No**

Problem M4.10.C

Is it possible for M[R8] to hold 0?

Yes **No**

Now consider the same program, but with two **MEMBAR** instructions.

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	$MEMBAR_{RW}$
$MEMBAR_{WR}$	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

We want to compare execution of the two programs on our system.

Problem M4.10.D

If both M[R8] and M[R9] contain 6, is it possible for R3 to hold 8?

Without **MEMBAR** instructions? **Yes** **No**

With **MEMBAR** instructions? **Yes** **No**

Problem M4.10.E

If both M[R8] and M[R9] contain 7, is it possible for R3 to hold 6?

Without **MEMBAR** instructions? **Yes** **No**

With **MEMBAR** instructions? **Yes** **No**

Problem M4.10.F

Is it possible for both M[R8] and M[R9] to hold 8?

Without **`MEMBAR`** instructions? **Yes** **No**

With **`MEMBAR`** instructions? **Yes** **No**

Problem 4.11: Memory Models

Consider a system which uses **Sequential Consistency (SC)**. There are three processes, **P1**, **P2** and **P3**, on different processors on such a system (the values of R_A, R_B, R_C were all zeros before the execution):

P1	P2	P3
P1.1: ST (A), 1 P1.2: LD R_C, (C)	P2.1: ST (B), 1 P2.2: LD R_A, (A)	P3.1: ST (C), 1 P3.2: LD R_B, (B)

Problem 4.11.A

After all processes have executed, it is possible for the system to have multiple machine states. For example, $\{R_A, R_B, R_C\} = \{1,1,1\}$ is possible if the execution sequence of instructions is P1.1→P2.1→P3.1→P1.2→P2.2→P3.2. Also, $\{R_A, R_B, R_C\} = \{1,1,0\}$ is possible if the sequence is P1.1 →P1.2 →P2.1 →P3.1 →P2.2 →P3.2.

For each state of $\{R_A, R_B, R_C\}$ below, specify the execution sequence of instructions that results in the corresponding state. If the state is **NOT** possible with SC, just put X.

{0,0,0} :

{0,1,0} :

{1,0,0} :

{0,0,1} :

Problem 4.11.B

Now consider a system which uses **Weak Ordering(WO)**, meaning that a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies.

Does WO allow the machine state(s) that is not possible with SC? If yes, provide an execution sequence that will generate the machine states(s).

Problem 4.11.C

The WO system in Problem 4.11.B provides four fine-grained memory barrier instructions. Below is the description of these instructions.

- **MEMBAR_{RR}** guarantees that all read operations initiated before the MEMBAR_{RR} will be seen before any read operation initiated after it.
- **MEMBAR_{RW}** guarantees that all read operations initiated before the MEMBAR_{RW} will be seen before any write operation initiated after it.
- **MEMBAR_{WR}** guarantees that all write operations initiated before the MEMBAR_{WR} will be seen before any read operation initiated after it.
- **MEMBAR_{WW}** guarantees that all write operations initiated before the MEMBAR_{WW} will be seen before any write operation initiated after it.

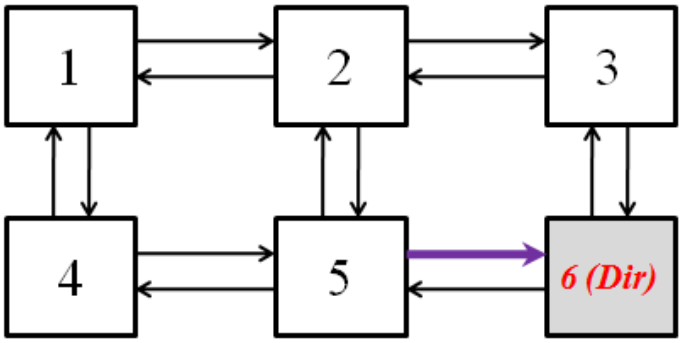
Using the minimum number of memory barrier instructions, rewrite **P1**, **P2** and **P3** so the machine state(s) that is not possible with SC by the original programs is also not possible with WO by your programs.

P1	**P2**	**P3**
P1.1: ST (A), 1	P2.1: ST (B), 1	P3.1: ST (C), 1
P1.2: LD R_C, (C)	P2.2: LD R_A, (A)	P3.2: LD R_B, (B)

Problem M4.12: Directory-based Protocol

Problem 4.12.A

The following questions deal with the directory-based protocol discussed in class. Assume XY routing, and message passing is FIFO. (**XY routing algorithm** first routes packets horizontally, towards their X coordinates, and then vertically towards their Y coordinates.) Protocol messages with the same source and destination sites are always received in the same order as that in which they were sent. **For this question, assume that the cache coherence protocol is free from deadlock, livelock and starvation**.



Assume the node 6 serves as the home directory, where the states for memory blocks are stored. Assume all caches are initially empty and no responses are sent voluntarily (i.e. every response is caused by a request)

Processor 1	Processor 4	Processor 5
I1.1: ST X, 10		I4.1: LD R1, X
	I5.1: ST X, 20	

Suppose the global execution order is as follows:

I4.1 => I5.1 => I1.1

Assume that the next instruction will start its execution only when the previous instruction has completed. For each instruction, list all protocol messages that are sent over the link 5 -> 6 (the purple link in the above figure).

I4.1:

I5.1:

I1.1:

Problem 4.12.B

For the directory protocol, we assume the message passing to be FIFO, meaning protocol messages with the same source and destination are always received in the same order as that in which they were sent. Now suppose messages can be delivered out-of-order for the same source and destination pairs. Describe one scenario that the cache coherence protocol will break due to this out-of-order delivery.

Problem 4.12.C

Under the 6823 directory-based protocol, a cache will receive a writeback request from the directory <M2C_Req, a, S> for address "a" when it is in state M and another cache wants a shared copy. Is it possible for a cache in the **S** state to receive <M2C_Req, a, S> ? Describe how this scenario can occur using the messages passed between the cache and the memory, and the state transitions.