Last updated:
2/10/2006 4:35 PM

Problem M4.1: 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 M4.1.A

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

Yes / No

Problem M4.1.B

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

Yes / No

Problem M4.1.C

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

Yes / No

Problem M4.1.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 M4.2: 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 M4.2.A

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

Problem M4.2.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 M4.2.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 M4.2.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 #12? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

Problem M4.3: Directory-based Cache Coherence Invalidate Protocols

In this problem we consider a cache-coherence protocol presented in Handout #12.

Problem M4.3.A — Protocol Understanding

Consider the situation in which memory sends a FlushReq message to a processor. This can only happen when the memory directory shows that the exclusive copy resides at that site. The memory processor intends to obtain the most up-to-date data and exclusive ownership, and then supply it to another site that has issued a ExReq. Table H12-1 row 21 specifies the PP behavior when the current cache state is C-pending (not C-exclusive) and a FlushReq is received.

Give a simple scenario that causes this situation.

Problem M4.3.B — 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.

Problem M4.3.C — Replace

In the current scheme, when a cache wants to voluntarily invalidate a shared cache line, the PP informs the memory of this operation. Describe a simple scenario where there would be an error, if the line was "silently dropped." Can you provide a simple fix for this problem in the protocol? Give such a fix if there is one, or explain why it wouldn't be a simple fix.

Problem M4.4: 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 #12. 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 M4.4.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 M4.4.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 M4.4-A (**single-sharer scheme**). When there is a load (`ShReq`) requests to a shared cache block, Ben invalidates the existing sharer to make room for the new sharer (home sends an `InvReq` to the existing sharer, the existing sharer sends an `InvRep` to home, home replaces the exiting sharer's ID with the new sharer's ID and sends a `ShRep` to the new sharer).

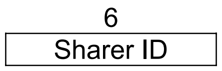

Figure M4.4-A

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 (R(dir) & dir= ε)) 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 M4.4.C

Ben thinks that he can improve his original scheme by adding an extra "**global bit**" to **S** as shown in Figure M4.4-B (**global-bit scheme**). The global bit is set when there is more than 1 processor sharing the data, and zero otherwise.

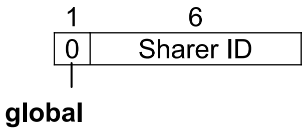

Figure M4.4-B

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

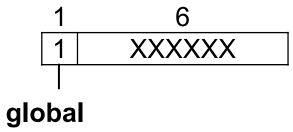

Figure M4.4-C

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 (i.e., R(dir) & (dir = ε)) 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 M4.4.D

Ben decides to modify the protocol from Handout #12 for his global-bit scheme (Problem 4.4.C) for a **64-processor system**. Your job is to complete the following table (Table M4.4-1) for him.

Use the same assumptions for the interconnection network, cache states, home directory states, and protocol messages as Handout #12. However, *R(dir)* (if *dir*≠ ε) now means that there is only one processor sharing the cache data (global bit is unset), and *R(all)* means the global bit is set.

Use *k* to represent the site that issued the received message. For *Tr(dir)* and *Tw(id)* states, use *j* to represent the site that issued the original protocol request (ShReq/ExReq).

No.	Current State	Message Received	Next State	Action
1	R(dir) & (dir = ε)	ShReq	R({k})	ShRep->k
2	R(dir) & (dir = ε)	ExReq	W(k)	ExRep->k
3	R(dir) & (dir ≠ ε)	ShReq		
4	R(all)	ShReq		
5	R(dir) & (dir ≠ ε)	ExReq		
6	R(all)	ExReq		
7	W(id)	ShReq	Tw(id)	WbReq->id
8	Tr(dir) & (id ∈ dir)	InvRep	Tr(dir - {id})	nothing
9	Tr(dir) & (dir = {k})	InvRep	W(*j*)	ExRep->j
10	Tw(id)	FlushRep		

Table M4.4-1: Partial List of Home Directory State Transitions

Problem M4.5: Tracing the Directory-based Protocol

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 #12 (as well as Lecture 18). Unless specified otherwise, assume all caches are initially empty and *no voluntary rules are used.*

Problem M4.5.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	ExReq; ExRep	B1	4		C1	8	
A2	2		B3	5		C2	9	
A4	3		B4	6		C4	10	
			B6	7				

How many messages are generated? ____________________

Problem M4.5.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 M4.5.C

Can the number of messages in Problem M4.5.B be decreased *by using voluntary rules*? Explain.

Problem M4.5.D

What is the execution sequence that generates the most messages? 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 M4.5.E

Assuming the protocol with voluntary rules, provide a sequence of events that would create a situation, where one of the processors' cache is in **Pending** state, and receives a **FlushReq** message.

Problem M4.5.F

Does reducing the number of cache coherence messages generated necessarily improve the time to execute a program? Explain.

Problem M4.6: Directory-base Cache Coherence Update Protocols

In Handout #12, we examined a cache-coherent distributed shared memory system. Ben wants to convert the directory-based invalidate cache coherence protocol from the handout into an update protocol. He proposes the following scheme.

Caches are write-through, not write allocate. When a processor wants to write to a memory location, it sends a WriteReq to the memory, along with the data word that it wants written. The memory processor updates the memory, and sends an UpdateReq with the new data to each of the sites caching the block, unless that site is the processor performing the store, in which case it sends a WriteRep containing the new data.

If the processor performing the store is caching the block being written, it must wait for the reply from the home site to arrive before storing the new value into its cache. If the processor performing the store is not caching the block being written, it can proceed after issuing the WriteReq.

Ben wants his protocol to perform well, and so he also proposes to implement silent drops. When a cache line needs to be evicted, it is silently evicted and the memory processor is not notified of this event.

Note that WriteReq and UpdateReq contain data at the word-granularity, and not at the block-granularity. Also note that in the proposed scheme, memory will always have the most up-to-date data and the state C-exclusive is no longer used.

As in the lecture, the interconnection network guarantees that message-passing is reliable, and free from deadlock, livelock, and starvation. Also as in the lecture, message-passing is FIFO.

Each home site keeps a FIFO queue of incoming requests, and processes these in the order received.

Problem M4.6.A Sequential Consistency

Alyssa claims that Ben's protocol does not preserve sequential consistency because it allows two processors to observe stores in different orders. Describe a scenario in which this problem can occur.

Problem M4.6.B State Transitions

Noting that many commercial systems do not guarantee sequential consistency, Ben decides to implement his protocol anyway. Fill in the following state transition tables (Table M4.6-1 and Table M4.6-2) for the proposed scheme. (Note: the tables do not contain all the transitions for the protocol).

No.	Current State	Event Received	Next State	Action
1	C-nothing	Load	C-transient	ShReq(id, Home, a)
2	C-nothing	Store		
3	C-nothing	UpdateReq		
4	C-shared	Load	C-shared	processor reads cache
5	C-shared	Store		
6	C-shared	UpdateReq		
7	C-shared	(Silent drop)		Nothing
8	C-transient	ShRep		data → cache, processor reads cache
9	C-transient	WriteRep		
10	C-transient	UpdateReq		

Table M4.6-1: Cache State Transitions

No.	Current State	Message Received	Next State	Action
1	R(dir) & id $\notin$ dir	ShReq	R(dir + {id})	ShRep(Home, id, a)
2	R(dir) & id $\notin$ dir	WriteReq		
3	R(dir) & id $\in$ dir	ShReq		ShRep(Home, id, a)
4	R(dir) & id $\in$ dir	WriteReq		

Table M4.6-2: Home Directory State Transitions

Problem M4.6.C UpdateReq

After running a system with this protocol for a long time, Ben finds that the network is flooded with UpdateReqs. Alyssa says this is a bug in his protocol. What is the problem and how can you fix it?

Problem M4.6.D FIFO Assumption

As in M4.3, FIFO message passing is a necessary assumption for the correctness of the protocol. If the network were non-FIFO, it becomes possible for a processor to never see the result of another processor's store. Describe a scenario in which this problem can occur.

Problem M4.7: Snoopy Cache Coherent Shared Memory

In this problem, we investigate the operation of the snoopy cache coherence protocol in Handout #13.

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 #13. 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 19 (2005). 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: Sequential Consistency, Synchronization, and Relaxed Memory Models

Cy D. Fect wants to run the following code sequences on processors P1 and P2, which are part of a two-processor MIPS64 machine. The sequences operate on memory values located at addresses `A`, `B`, `C`, `D`, `E`, `F`, and `G`, which are all sequentially located in memory (e.g. `B` is 8 bytes after `A`). Initially, `M[A]`, `M[B]`, `M[C]`, `M[D]`, and `M[E]` are all 0. `M[F]` is 1, and `M[G]` is 2. For each processor, `R1` contains the address `A` (all of the addresses are located in a shared region of memory). Also, remember that for a MIPS processor, `R0` is hardwired to 0. In the below sequences, a semicolon is used to indicate the start of a comment.

```
P1                              P2
ADDI R2,R0,#1  ;R2=1            ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2            SD   R2,8(R1)  ;B=R2
LD   R3,40(R1) ;R3=F            LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3            SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B            LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4            SD   R4,32(R1) ;E=R4
```

Problem M4.10.A — Sequential Consistency

If Cy's code runs on a system with a sequentially consistent memory model, what are the possible results of execution? List all possible results in terms of values of `M[C]`, `M[D]`, and `M[E]` (since the values in the other locations will be the same across all possible execution paths).

Problem M4.10.B — Generalized Synchronization

Assume now that Cy's code is run on a system that does not guarantee sequential consistency, but that memory dependencies are not violated for the accesses made by any individual processor. The system has a `MEMBAR` memory barrier instruction that guarantees the effects of all memory instructions executed before the `MEMBAR` will be made globally visible before any memory instruction after the `MEMBAR` is executed.

Add `MEMBAR` instructions to Cy's code sequences to give the same results as if the system were sequentially consistent. Use the minimum number of `MEMBAR` instructions.

Problem M4.10.C — Total Store Ordering

Now consider a machine that uses finer-grain memory barrier instructions. The following instructions are available:

- `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.

There is no generalized `MEMBAR` instruction as in Part B of this problem.

In total store ordering (TSO), a read may complete before a write that is earlier in program order if the read and write are to different addresses and there are no data dependencies. For a machine using TSO, insert the minimum number of memory barrier instructions into the code sequences for P1 and P2 so that sequential consistency is preserved.

Problem M4.10.D — 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 and there are no data dependencies. For a machine using PSO, insert the minimum number of memory barrier instructions from Part C into the code sequences for P1 and P2 so that sequential consistency is preserved.

Problem M4.10.E — Weak Ordering

In weak ordering (WO), 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. For a machine using WO, insert the minimum number of memory barrier instructions from Part C into the code sequences for P1 and P2 so that sequential consistency is preserved.

Problem M4.10.F 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*. However, reads and writes before an *acquire* may complete after the *acquire*, and reads and writes after a *release* may complete before the release. Consider the following modified versions of the original code sequences.

```
P1                                    P2
ADDI R2,R0,#1  ;R2=1                  ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                  SD   R2,8(R1)  ;B=R2
L1:ACQUIRE R2,56(R1);SWAP(R2,H)       L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                            BNEZ R2,L1
LD   R3,40(R1) ;R3=F                  LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                  SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0                 RELEASE R0,56(R1);H=0
LD   R4,8(R1)  ;R4=B                  LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4                  SD   R4,32(R1) ;E=R4
```

In the above sequences, the *acquire* and *release* operations modify memory location `H`, which is located sequentially after `G`. The *acquire* operation performs a read and a write, while the *release* operation performs a write. For a machine using RC, insert the minimum number of memory barrier instructions from Part C into the above code sequences for P1 and P2 so that sequential consistency is preserved.

Problem M4.11: 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, same as in **Problem M4.10**. Below is the description of these instructions, copied from the Problem Set.

- **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 M4.11.A

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

Yes **No**

Problem M4.11.B

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

Yes **No**

Problem M4.11.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 M4.11.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 M4.11.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 M4.11.F

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

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

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