

# Problem Set 6 Solutions

CS152 Spring 2009

## Problem P6.1: Sequential Consistency

### Problem P6.1.A

---

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

Yes /  No

A1-A4, B1-B4, C1-C4, B5, B6

### Problem P6.1.B

---

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

Yes /  No

All results must be even!

### Problem P6.1.C

---

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

Yes /  No

All of C, All of A, All of B

### Problem P6.1.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

All stores/loads must be done in order because they're to the same address, so no new results are possible.

## **Problem P6.2: Synchronization Primitives**

The mechanism here is as follows: LdR requests READ access to the address, StC requests WRITE access to the address. Many students suggested that LdR can request WRITE access to the address right away, which could lead to live lock.

### **Problem P6.2.A**

---

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

If another processor requests Write access to the same cache line.

### **Problem P6.2.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

Yes. WbReq and ShReq are sent normally. The “reservation” is local (probably in the snooper or in the cache, though that might take too much resources – there are very few reservations needed at the same time for any processor).

### **Problem P6.2.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.

1. Bus doesn't need to be aware of them.
2. Everything is local.
3. No ping-pong.
4. No extra hardware (tied to 1)

### **Problem P6.2.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 #7? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

No – our bus invalidates before transitioning from Sh to Ex. In general, maybe.

## **Problem P6.3: Directory-based Cache Coherence Invalidate Protocols**

In this problem we consider a cache-coherence protocol presented in Handout #6.

### **Problem P6.3.A**

### **Protocol Understanding**

Consider the following scenario:

Assume initially the home directory state is  $W(m)$ , indicating that the block is modified at site  $m$ .

1. The home site receives a  $ExReq$  from a site (not  $m$ ). The home site sends a  $FlushReq$  to site  $m$ . The  $ExReq$  is suspended and buffered at the home site. The home directory site becomes  $Tw(m)$ .
2. Before the  $FlushReq$  arrives, the modified cache line is replaced at site  $m$  (due to a cache line conflict). Site  $m$  sends a  $FlushRep$  to the home site. The cache line state becomes  $C$ -nothing. The processor at site  $m$  issues a  $ExReq$  to the home site. The  $ExReq$  is suspended at site  $m$ . The cache line state becomes  $C$ -pending.
3. The  $FlushReq$  arrives at site  $m$ .

### **Problem P6.3.B**

### **Non-FIFO Network**

Assume initially that a particular block is not cached at any site. Consider the following scenario:

1. The home site receives a  $ShReq$  from site  $A$ .
2. The home site sends a  $ShRep$  to site  $A$  and changes the state to  $R(\{A\})$ .
3. The home site receives a  $ExReq$  from site  $B$ .
4. The home site sends a  $InvReq$  to site  $A$  and changes the state to  $Tr(\{A\})$ .
5. The  $InvReq$  arrives at site  $A$  before the  $ShRep$ .
6. Site  $A$  dequeues the  $InvReq$ , it DOESN'T send a  $InvRep$  to the home site (because Site  $A$  is in  $C$ -pending state).
7. The  $ShRep$  arrives at site  $A$  and  $A$  changes state to  $C$ -shared.
8. The home site is still in  $Tr(\{A\})$ .
9. Site  $B$  is still waiting for  $ExRep$ .

There is a deadlock in this situation.

### **Problem P6.3.C**

### **Replace**

When a cache  $A$  decides to invalidate a shared cache line and the directory is not informed, the home directory will continue to have  $A$  in its member list  $R(dir)$ . Say cache  $C$  (not a current sharer) now wants to write to this cache line, an  $ExReq$  will be sent to the directory. The directory then sends an  $InvReq$  to cache  $A$  as well as any other nodes in the member list and the directory changes to the  $Tr(dir)$  state. It waits for  $InvRep$  from all these nodes before an  $ExRep$  to  $C$  can be sent. However, it will not receive a reply from  $A$ .

## Problem P6.4: Directory-base Cache Coherence Update Protocols

### Problem P6.4.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.

Consider blocks X and Y, whose home sites are A and B, respectively:

1. A receives a WriteReq for X, and B receives a WriteReq for Y
2. C and D are both caching X and Y. So both A and B send UpdateReq to C and D.
3. C receives first the UpdateReq for X, then the UpdateReq for Y
4. D receives first the UpdateReq for Y, then the UpdateReq for X

C and D can receive the UpdateReq in different orders because they are arriving from different home sites, and FIFO message passing only provides guarantees for messages with the same source and destination.

### Problem P6.4.B

### State Transitions

No.	Current State	Event Received	Next State	Dequeue Message?	Action
1	C-nothing	Load	C-transient	No	ShReq(id, Home, a)
2	C-nothing	Store	<b>C-transient</b>	No	<b>WriteReq(id, Home, a)</b>
3	C-nothing	UpdateReq	<b>C-nothing</b>	Yes	None
4	C-shared	Load	C-shared	Yes	reads cache
5	C-shared	Store	<b>C-transient</b>	No	<b>WriteReq(id, Home, a)</b>
6	C-shared	UpdateReq	<b>C-shared</b>	Yes	<b>data-&gt; cache</b>
7	C-shared	(Silent Drop)	<b>C-nothing</b>	N/A	None
8	C-transient	ShRep	<b>C-shared</b>	Yes	data → cache
9	C-transient	WriteRep	<b>C-shared</b>	Yes	<b>data -&gt; cache, Store retires</b>
10	C-transient	UpdateReq	<b>C-transient</b>	Yes	<b>data -&gt; cache</b>

Table P6.4-1: Cache State Transitions

No.	Current State	Message Received	Next State	Dequeue Message?	Action
1	$R(\text{dir}) \ \& \ id \notin \text{dir}$	ShReq	$R(\text{dir} + \{id\})$	Yes	ShRep(Home, id, a)
2	$R(\text{dir}) \ \& \ id \notin \text{dir}$	WriteReq	$R(\text{dir} + \{id\})$	Yes	data -> memory forall $i \in \text{dir}$ if( $i==id$ ) WriteRep-> id else UpdateReq-> i
3	$R(\text{dir}) \ \& \ id \in \text{dir}$	ShReq	$R(\text{dir})$	Yes	ShRep(Home, id, a)
4	$R(\text{dir}) \ \& \ id \in \text{dir}$	WriteReq	$R(\text{dir})$	Yes	data -> memory forall $i \in \text{dir}$ if( $i==id$ ) WriteRep-> id else UpdateReq-> i

Table P6.4-2: Home Directory State Transitions

### **Problem P6.4.C**

### **UpdateReq**

Because caches do not notify the home site when a line gets replaced, the set S for a memory block will only increase. Eventually, every site that has ever loaded a particular block will be in the set S for that block, resulting in numerous UpdateReq on the network, even though many of the recipients of the update have already replaced that cache line. The solution is to notify the home site when a cache line is replaced, and have the home site remove a site from S when such a notification is received.

### **Problem P6.4.D**

### **FIFO Assumption**

Consider a site A:

1. Site A sends ShReq for block X to X's home site.
2. X's home site receives ShReq and issues a ShRep.
3. Site B sends a WriteReq for block X to X's home site.
4. X's home receives WriteReq, and issues an UpdateReq to A
5. The ShReq and UpdateReq are re-ordered in the network
6. The UpdateReq arrives at A. Since A is waiting for ShRep, it is in C-transient. It updates its cache with the UpdateReq data, but remains in C-transient.
7. The ShRep arrives at A. The cache writes the stale data into its cache, and reads the result.

Although A received the UpdateReq, it will never see the result of B's store unless the cache line gets replaced, and is re-loaded.

## Problem P6.5: Snoopy Cache Coherent Shared Memory

### Problem P6.5.A

### Where in the Memory System is the Current Value

See Table P6.5-1, P6.5-2 and P6.5-3.

### Problem P6.5.B

### MBus Cache Block State Transition Table

See Table P6.5-1, P6.5-2 and P6.5-3.

### Problem P6.5.C

### Adding atomic memory operations to MBus

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.

The problem is that CPU B can read the value in location x while CPU A is performing the fetch-and-increment operation—which violates the idea of fetch-and-increment being atomic. For example, consider the following sequence of events and corresponding state transitions and operations:

Event	CPU A	CPU B
1	Read(x); I->CS; send CR	
2		Snoop CR; CE->CS
3		Read(x)
4	Write(x); CS->OE; send CI	
5		Snoop CI; CS->I

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
<b>Invalid</b>	yes	read	<b>CR</b>	<b>CS</b>	✓	✓	✓
<b>cleanShared</b>	yes	write	<b>CI</b>	<b>OE</b>	✓		

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

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem	
<b>cleanExclusive</b>	no	none	none	<b>CE</b>	√		√	
		CPU read	none	<b>CE</b>	√		√	
		CPU write	none	<b>OE</b>	√			
		replace	none	<b>I</b>			√	
		<b>CR</b>	none or CCI <sup>1</sup>	<b>CS</b>	√	√	√	
		<b>CRI</b>	none or CCI <sup>1</sup>	<b>I</b>		√		
		<b>CI</b>	none	<i>Impossible</i>				
		<b>WR</b>	none	<i>Impossible</i>				
		<b>CWI</b>	none	<b>I</b>				√

Table P6.5-1

<sup>1</sup> Some Sun MBus implementations perform CCI from the cleanExclusive state, while others do not. We accept both answers.

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem	
<b>ownedExclusive</b>	no	none	none	<b>OE</b>	√			
		CPU read	none	<b>OE</b>	√			
		CPU write	none	<b>OE</b>	√			
		replace	WR	<b>I</b>			√	
		<b>CR</b>	CCI	<b>OS</b>	√	√		
		<b>CRI</b>	CCI	<b>I</b>		√		
		<b>CI</b>	none	<i>Impossible</i>				
		<b>WR</b>	none	<i>Impossible</i>				
		<b>CWI</b>	none	<b>I</b>			√	

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem	
<b>cleanShared</b>	no	none	none	<b>CS</b>	√		√	
		CPU read	none	<b>CS</b>	√		√	
		CPU write	CI	<b>OE</b>	√			
		replace	none	<b>I</b>			√	
		<b>CR</b>	none <sup>2</sup>	<b>CS</b>	√	√	√	
		<b>CRI</b>	none	<b>I</b>		√		
		<b>CI</b>	none	<i>Impossible</i>				
		<b>WR</b>	none	<i>Impossible</i>				
		<b>CWI</b>	none	<b>I</b>			√	
<b>cleanShared</b>	yes	none	same as above	<b>CS</b>	√	√	√	
		CPU read		<b>CS</b>	√	√	√	
		CPU write		<b>OE</b>	√			
		replace		<b>I</b>		√	√	
		<b>CR</b>		<b>CS</b>	√	√	√	
		<b>CRI</b>		<b>I</b>		√		
		<b>CI</b>		<b>I</b>		√		
		<b>WR</b>		<b>CS</b>	√	√	√	
		<b>CWI</b>		<b>I</b>			√	

**Table P6.5-2**

<sup>2</sup> Some Sun MBus implementations perform CCI from the cleanShared state. However, in these implementations, requests are not broadcast on a bus, but are handled by a central system controller. The system controller arbitrates which cache with a cleanShared copy provides the data. Unless an explanation is provided, CCI is not a valid response from this state.

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem	
<b>ownedShared</b>	no	none	none	<b>OS</b>	√			
		CPU read	none	<b>OS</b>	√			
		CPU write	CI	<b>OE</b>	√			
		replace	WR	<b>I</b>			√	
		<b>CR</b>	CCI	<b>OS</b>	√	√		
		<b>CRI</b>	CCI	<b>I</b>		√		
		<b>CI</b>	none	<i>Impossible</i>				
		<b>WR</b>	none	<i>Impossible</i>				
		<b>CWI</b>	none	<b>I</b>			√	
<b>ownedShared</b>	yes	none	same as above	<b>OS</b>	√	√		
		CPU read		<b>OS</b>	√	√		
		CPU write		<b>OE</b>	√			
		replace		<b>I</b>		√	√	
		<b>CR</b>		<b>OS</b>	√	√		
		<b>CRI</b>		<b>I</b>		√		
		<b>CI</b>		<b>I</b>		√		
		<b>WR</b>		<i>Impossible</i>				
		<b>CWI</b>		<b>I</b>			√	

**Table P6.5-3**