Problem M11.1: 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 M11.1.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 M11.1.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. Writeback [P2C_Req(a) S] and [C2P_Req(a) S] 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 M11.1.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 M11.1.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 the handout? 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 S to M. In general, maybe.

Problem M11.2: Implementing Directories

Problem M11.2.A

Overhead for a 4-processor system: 4 bits / 32 bytes = 4 / (32 * 8) = 1/64

Overhead for a 64-processor system: 64 bits / 32 bytes = 64 / (32 * 8) = 1/4

Problem M11.2.B

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

For the bit-vector scheme: No invalidate-requests are sent.

For the single-sharer scheme:
1 invalidate-request is sent to P0 when P1 reads B.
1 invalidate-request is sent to P1 when P0 reads B the second time.

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

For the bit-vector scheme:
1 invalidate-request is sent to each shared processor (P0 and P1) when P2 writes B.
-> 2 invalidate-requests are sent.

For the single-sharer scheme:
1 invalidate-request is sent to P0 when P1 reads B.
1 invalidate-request is sent to the only sharer (P1) when P2 writes B.

Problem M11.2.C

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

For the global-bit scheme: No invalidate-requests are sent.

Sequence 2	global-bit scheme # of invalidate-requests
Processor #0 reads **B**	0
Processor #1 reads **B**	**0**
Processor #2 writes **B**	**64**

For the global-bit scheme:
1 invalidate-request is sent to each of the 64 processors because the global bit is set when P2 writes B. -> 64 invalidate-requests are sent.

Note: If the protocol is optimized, no invalidate-request would be sent to P2 and the number of invalidate-requests would be 63 instead of 64.

Problem M11.3: Tracing the Directory-based Protocol

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	

Problem M11.3.A

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	**<M,B,Req,x,S> <A,M,Req,x,S> <M,A,Rep,x,M,S,2> <B,M,Rep,x,I,S,2>**	C1	8	**<M,C,Req,x,M> <B,M,Req,x,I> <M,B,Rep,x,M,I,6> <C,M,Rep,x,I,M,6>**
A2	2		B3	5	**<M,B,Req,x,M> <A,M,Req,x,I> <M,A,Rep,x,S,I,-> <B,M,Rep,x,S,M,->**	C2	9	
A4	3		B4	6		C4	10	
			B6	7				

How many messages are generated? **14**

Problem M11.3.B

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

How many messages are generated? **12**

Problem M11.3.C

Can the number of messages in Problem M11.3.B be decreased *by using voluntary responses*? Explain.

Yes – all the requests can be eliminated using voluntary rules. Total number of messages would be 6 instead of 12.

Problem M11.3.D

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	**2**	**<M,B,Req,x,S> <A,M,Req,x,S> <M,A,Rep,x,M,S,1> <B,M,Rep,x,I,S,1>**	C1	**3**	**<M,C,Req,x,M> <A,M,Req,x,I> <B,M,Req,x,I> <M,A,Rep,x,S,I> <M,B,Rep,x,S,I> <C,M,Rep,x,I,M,1>**
A2	**4**	**<M,A,Req,x,S> <C,M,Req,x,S> <M,C,Rep,x,M,S,6> <A,M,Rep,x,S,6>**	B3	**5**	**<M,B,Req,x,M> <A,M,Req,x,I> <C,M,Req,x,I> <M,A,Rep,x,S,I> <M,C,Rep,x,S,I> <B,M,Rep,x,I,M,6>**	C2	**6**	**<M,C,Req,x,S> <B,M,Req,x,S> <M,B,Rep,x,M,S,2> <C,M,Rep,x,I,S,2>**
A4	**7**	**<M,A,Req,x,M> <B,M,Req,x,I> <C,M,Req,x,I> <M,B,Rep,x,S,I> <M,C,Rep,x,S,I> <A,M,Rep,x,I,M,2>**	B4	**8**	**<M,B,Req,x,S> <A,M,Req,x,S> <M,A,Rep,x,M,S,12> <B,M,Rep,x,S,12>**	C4	**9**	**<M,C,Req,x,M> <A,M,Req,x,I> <B,M,Req,x,I> <M,A,Rep,x,S,I> <M,B,Rep,x,S,I> <C,M,Rep,x,I,M,12>**
			B6	**10**	**<M,B,Req,x,M> <C,M,Req,x,I> <M,C,Rep,x,M,I,4> <B,M,Rep,x,I,M,4>**			

How many messages are generated? **46**

Problem M11.4: Snoopy Cache Coherent Shared Memory

Problem M11.4.A Where in the Memory System is the Current Value

See Table M11.4-1, M11.4-2 and M11.4-3.

Problem M11.4.B MBus Cache Block State Transition Table

See Table M11.4-1, M11.4-2 and M11.4-3.

Problem M11.4.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
Invalid	yes	read	**CR**	**CS**	√	√	√
cleanShared	yes	write	**CI**	**OE**	√		

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	none	CE	√		√
		CPU write	none	OE	√		
		replace	none	I			√
		CR	none or CCI[1]	**CS**	√	√	√
		CRI	none or CCI[1]	I		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	I			√

Table M11.4-1

[1] 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
ownedExclusive	no	none	none	**OE**	√		
		CPU read	none	**OE**	√		
		CPU write	none	**OE**	√		
		replace	WR	**I**			√
		CR	CCI	**OS**	√	√	
		CRI	CCI	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanShared	no	none	none	**CS**	√		√
		CPU read	none	**CS**	√		√
		CPU write	CI	**OE**	√		
		replace	none	**I**			√
		CR	none[2]	**CS**	√	√	√
		CRI	none	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
cleanShared	yes	none	same as above	**CS**	√	√	√
		CPU read		**CS**	√	√	√
		CPU write		**OE**	√		
		replace		**I**		√	√
		CR		**CS**	√	√	√
		CRI		**I**		√	
		CI		**I**		√	
		WR		**CS**	√	√	√
		CWI		**I**			√

Table M11.4-2

[2] 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
ownedShared	no	none	none	**OS**	√		
		CPU read	none	**OS**	√		
		CPU write	CI	**OE**	√		
		replace	WR	**I**			√
		CR	CCI	**OS**	√	√	
		CRI	CCI	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
ownedShared	yes	none	same as above	**OS**	√	√	
		CPU read		**OS**	√	√	
		CPU write		**OE**	√		
		replace		**I**		√	√
		CR		**OS**	√	√	
		CRI		**I**		√	
		CI		**I**		√	
		WR		*Impossible*			
		CWI		**I**			√

Table M11.4-3

Problem M11.5: Snoopy Cache Coherent Shared Memory

Problem M11.5.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	**none**	**COS**
		CPU write	**CI**	**OE**
		replace	**none**	**I**
		CR	**CCI**	**COS**
		CRI	**CCI**	**I**
		CI	**none**	**I**
		WR	**Impossible**	
		Or:	**none**	**COS**
		CWI	**none**	**I**

Note that WR is not necessary during replace because the line is clean.
Also, an incoming WR operations is Impossible because other caches can only have the block in the CS state, but (none, COS) was also accepted as a correct answer.

Problem M11.5.B

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. cache 1 replaces block B	**-**	**I**	**CS**	**CS**	**I**
5.cache 4 reads data block B	**memory**	**I**	**CS**	**CS**	**CS**

Problem M11.5.C

When the CPU does a write, it can change a cache block from CE to OE with no bus operation, but to transition from COS to OE it must first broadcast a CI on the bus to invalidate any shared (CS) copies of the block.

Problem M11.6: Snoopy Caches

Problem M11.6.A

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

The snooper can allow the CPU to continue executing normally, but cannot allow any new messages from the outside to enter the caches until AFTER the caches cleared their content.

Problem M11.6.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?

Here the snooper MUST respond RETRY and get the cache to write back the value.

Problem M11.6.C

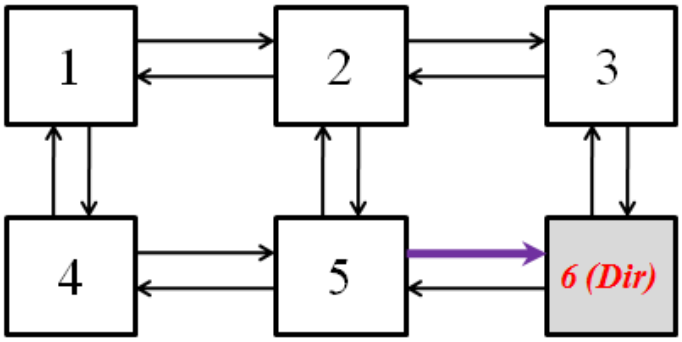
When an ExReq message is seen by the snooper and there is a Wb message in the C2M queue waiting to be sent, the snooper replies ***retry***. If the cache line is about to be modified by another processor, why is it important to first write back the already modified cache line? Does your answer change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

Because otherwise the Wb can happen out of order with some other memory operation and SC could be broken.

Problem M11.7: Directory-based Protocol

Problem M11.7.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
1.1: ST X, 10	4.1: LD R1, X	5.1: ST X, 20

Suppose the global execution order is as follows:

4.1 => 5.1 => 1.1

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).

4.1: <6,4,C2M_Req,X,S> (4.1),

5.1: <6,5,C2M_Req,X,M>, <6,4,C2M_Rep,X,S,I> (5.1),

1.1: <6,5,C2M_Rep,X,M,I,20> (1.1)

Problem M11.7.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.

1. Core 1: <M,1,C2M_Req,a,S> => <1,M,M2C_Rep,a,I,S,data> (not yet reached)
2. Core 2: <M,2,C2M_Req,a,M> => <1,M,M2C_Req,a,I>

If <1,M,M2C_Req,a,I> arrives earlier than <1,M,M2C_Rep,a,I,S,data>, it will be ignored, and the core will not send any reply to home which is waiting. => Deadlock.

Problem M11.7.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.

Cache 1 in M, does voluntary writeback <M,1,M2C_Rep,a,M,S,data> and goes to S state. Now Cache 2 in I state does a <M,2,C2M_Req,a,S>. If the Mem hasn't received Cache 1's response yet, it will send a <1,M,P2C_Req,a,S> to Cache 1 which is in S.

Problem M11.8: Synchronicity (Spring 2014 Quiz 4, Part B)

You are writing a queue to be used in a multi-producer/single-consumer application. (Producer threads write messages that are read by one consumer.) We assume here a queue with infinite space. The basic code is shown below.

`TST rs, Imm(rt)` is the test-and-set instruction, which *atomically* loads the value at `Imm(rt)` into `rs`, and if the value is zero, updates the memory location at `Imm(rt)` to 1. This atomic instruction is useful for implementing locks: a value of 1 at the memory location indicates that someone holds the lock, and a value of 0 means the lock is free.

Producer pushes a message onto queue: (memory operations in bold)

```
void push(int** tail_ptr, int* tail_write_lock, int message) {
      while (lock_try(tail_write_lock) == false);
      **tail_ptr = message;
      *tail_ptr++;
      lock_release(tail_write_lock);
}

# R1 – contains address of data to enqueue
# R2 – contains the address of the tail pointer of queue
# R3 – address of tail pointer write lock
P1 SpinLock:TST R4, 0(R3)           # try to acquire tail write lock
P2          BNEZ R4, R4, SpinLock
P3          LD R4, 0(R2)            # get tail pointer
P4          ST R1, 0(R4)            # write message to tail
P5          ADD R4, R4, 4           # update tail pointer
P6          ST R4, 0(R2)
P7          ST R0, 0(R3)            # release lock
```

Consumer pops a message off queue: (memory operations in bold)

```
int pop(int** head_ptr, int** tail_ptr) {
      while (*head_ptr == *tail_ptr);
      int message = **head_ptr;
      *head_ptr++;
      return message;
}

# R1 – will receive address contained in message
# R2 – contains the address of the head pointer of queue
# R3 – contains the address of the tail pointer of the queue
C1 Retry:   LD R4, 0(R2)            # get head pointer
C2          LD R5, 0(R3)            # get tail pointer
C3          SUB R5, R4, R5          # is there a message?
C4          BNEZ R5, Pop
C5          JMP Retry
C6 Pop:     LD R1, 0(R4)            # read message from queue
C7          ADD R4, R4, 4           # update head pointer
C8          ST R4, 0(R2)
```

Problem M11.8.A

You are trying to port this code to an architecture that does not have the `TST` instruction (but, happily, the rest of the ISA is unchanged). Instead the new architecture has load-reserve/store-conditional instructions. Implement `TST rs, 0(rt)` using load-reserve/store-conditional:

```
LR rs, Imm(rt):
     rs ← Memory[(rt) + Imm]
     Track address (rt) + Imm

SC rs, Imm(rt):
     If (rt) + Imm modified:
          rs ← 0                          # Fail
     Else:
          Memory[(rt) + Imm] = (rs)# Succeed
          rs ← 1
```

```
TST rs, 0(rt):
     LR rs, 0(rt)   # test: is 0(rt) 1?
     BNEZ rs, skip
     ADD rs, rs, 1  # set: try to store 1
     SC rs, 0(rt)
     NOR rs, rs, rs # invert result to match TST
skip: NOP
```

Problem M11.8.B

This new architecture is also *not* sequentially consistent. Give an example of memory orderings between the producer and consumer that would result in incorrect behavior. *Explain your answer fully or you will not receive credit.*

Your answer should look something like:
`P1, P3, P4, C1, C2, P6, P7, C1, C2, C6, C8`
(Except that this is a sequentially consistent ordering, so it is not a correct answer.)

If the tail write is visible to the consumer before the message write, then we have a problem. Thus any sequence that contains the subsequence:

P6 C6 P4

Will read an invalid message. There are many other invalid sequences.

Problem M11.8.C

Show where memory fences should be added to the producer and consumer code to ensure correctness with a weak consistency model. Explain your answer fully.

```
P1 SpinLock:TST R4, 0(R3)          # try to acquire tail write lock
            FENCE_WR # don't read tail ptr before getting lock

P2          BNEZ R4, R4, SpinLock

P3          LD R4, 0(R2)           # get tail pointer

P4          ST R1, 0(R4)           # write message to tail
            FENCE_WW # don't update tail before writing message

P5          ADD R4, R4, 4          # update tail pointer

P6          ST R4, 0(R2)
            FENCE_WW # don't release lock before updating tail

P7          ST R0, 0(R3)           # release lock




C1 Retry:   LD R4, 0(R2)           # get head pointer

C2          LD R5, 0(R3)           # get tail pointer

C3          SUB R5, R4, R5         # is there a message?

C4          BNEZ R5, Pop

C5          JMP Retry
            FENCE_RR # don't read message before tail is updated

C6 Pop:     LD R1, 0(R4)           # read message from queue

C7          ADD R4, R4, 4          # update head pointer

C8          ST R4, 0(R2)
```

Problem M11.8.D

Let's next consider performance with a single producer thread and consumer thread. The following happens repeatedly:

1. The producer executes all instructions to push a message on the queue.
2. The consumer executes all instructions to pop a message off the queue.

Assume data, head, and tail pointers all lie in different, non-conflicting cache blocks.

First, after a few messages have been sent through the queue, will the consumer ever miss reading the head pointer? Will the producer ever miss reading the tail write lock, or fail to acquire the tail write lock? Explain in one or two sentences.

No, the head pointer belongs exclusively to the consumer. Likewise with a single producer, the tail write lock belongs exclusively to the producer. The consumer and producer will ping-pong on the tail pointer, however, since each uses it.

Problem M11.8.E

We'll now focus on the tail pointer only. Assuming a MSI invalidate coherence protocol, show the state of the tail pointer in the producer and consumer cache after each operation in the sequence below. Show any data or permissions transfers, e.g. "Memory→C" or "C invalidates P".

Operation	Producer tail pointer state	Consumer tail pointer state	Transfers
	I	I	
P1 TST try lock			
P3 LD tail_ptr	S		P ← Memory
P4 ST message			
P6 ST new_tail	M		
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr	S	S	C ← P
C6 LD message			
C7 ST new_head			
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail	M	I	P invalidates C
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr	S	S	C ← P; Memory ← P
C6 LD message			
C7 ST new_head			

How many state transitions occur per message in the steady state? 2 (second half of table)

Problem M11.8.F

Stay focused on the tail pointer only. Assume an update coherence protocol where the state of each line is either valid (V) or invalid (I). Show the state of the tail pointer in the producer and consumer cache after each operation in the sequence below in the steady state. Show any data or permissions transfers, e.g. "Memory→C" or "C invalidates P".

Operation	Producer tail pointer state	Consumer tail pointer state	Transfers
	I	I	
P1 TST try lock			
P3 LD tail_ptr	V		P ← Memory
P4 ST message			
P6 ST new_tail	V		
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr	V	V	C ← P
C6 LD message			
C7 ST new_head			
P1 TST try lock			
P3 LD tail_ptr	V	V	
P4 ST message			
P6 ST new_tail	V	V	C ← P
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr	V	V	
C6 LD message			
C7 ST new_head			

How many state transitions occur per message in the steady state? Zero, but one data transfer (P6). Memory may also be updated at P6, depending on the protocol (if not, V→I must writeback).

Problem M11.8.G

Your new architecture supports "remote access" for cached lines. This lets you assign a "home cache" for lines so that all memory operations will be sent *over the network* to operate remotely on the line *without allocating it in the requesting cache*.

For example, if line 0x100 is homed to processor A, and processor B writes 0x100, then *processor A's cache will be updated* and processor B's will be unchanged.

Assume the tail pointer is mapped to the producer's cache, and the cache uses an MSI invalidate protocol (similar to Question 5). Once again, show the state of the tail pointer for the sequence of operations in the steady state and data/permission transfers:

Operation	Producer tail pointer state	Consumer tail pointer state	Transfers
	I	I	
P1 TST try lock			
P3 LD tail_ptr	S		P ← Memory
P4 ST message			
P6 ST new_tail	M		
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			C *processor* ← P
C6 LD message			
C7 ST new_head			
P1 TST try lock			
P3 LD tail_ptr			
P4 ST message			
P6 ST new_tail			
P7 ST release lock			
C1 LD head_ptr			
C2 LD tail_ptr			C *processor* ← P
C6 LD message			
C7 ST new_head			

How many state transitions occur per message in the steady state? Zero, but one data transfer. The difference is that in this case the transfer is on demand—which may or may not be an improvement, depending on consumer behavior.

Problem M11.9: Cache Coherence (Spring 2015 Quiz 3, Part B)

Ben Bitdiddle is designing a snoopy-based, write-invalidate MSI protocol for write-back caches. Under the standard MSI protocol, when a cache observes a Bus Read Exclusive message (BusRdX), it has to invalidate its own copy of the cache block. Ben instead proposes an optimization, called delayed invalidation, to potentially reduce the number of read misses. The optimization works as follows:

Delayed invalidation: When a cache observes a Bus Read Exclusive message (BusRdX) and it has a copy of the block in the Shared (S) state, the cache delays the invalidation of the block until before a cache miss happens. In other words, the cache will treat any subsequent requests from its own processor as if the BusRdX had not happened, until one of those requests causes a miss. At that point, all pending invalidations are performed before processing the miss.

Problem M11.9.A

Suppose processors P1 and P2 are have private, snoopy caches. Both caches are initially empty. Consider the following sequence of accesses:

```
I0   P2: read   A
I1   P1: write  A
I2   P2: read   A
I3   P1: write  A
I4   P2: read   A
I5   P2: read   B
I6   P2: read   A
```

Assume blocks A and B do not conflict in the cache. Compare Ben's delayed invalidation optimization with the standard MSI protocol by filling the states (on the next page) for each cache block after each operation is done and calculate the number of misses in both cases.

Assume we use the standard MSI protocol. Fill in the following table.

Standard MSI Protocol				
	Processor P1's Cache		Processor P2's Cache	
Initial State	A: I	B: I	A: I	B: I
After P2 reads A	A: I	B: I	A: S	B: I
After P1 writes A	A: M	B: I	A: I	B: I
After P2 reads A	A: S	B: I	A: S	B: I
After P1 writes A	A: M	B: I	A: I	B: I
After P2 reads A	A: S	B: I	A: S	B: I
After P2 reads B	A: S	B: I	A: S	B: S
After P2 reads A	A: S	B: I	A: S	B: S

How many misses occur in the two caches? 2 write misses + 4 read misses = 6 misses

Assume we adopt Ben's delayed invalidation optimization. Fill in the following table. If there is a delayed invalidation, write it in the invalidation queue (the "Inv Queue" column). For example, "Inv L" means there is a delayed invalidation on block L.

MSI Protocol with Delayed Invalidation						
	Processor P1's Cache			Processor P2's Cache		
	MSI state		Inv Queue	MSI state		Inv Queue
Initial State	A: I	B: I		A: I	B: I	
After P2 reads A	A: I	B: I		A: S	B: I	
After P1 writes A	A: M	B: I		A: S	B: I	Inv A
After P2 reads A	A: M	B: I		A: S	B: I	Inv A
After P1 writes A	A: M	B: I		A: S	B: I	Inv A
After P2 reads A	A: M	B: I		A: S	B: I	Inv A
After P2 reads B	A: M	B: I		A: I	B: S	
After P2 reads A	A: S	B: I		A: S	B: S	

How many misses occur in the two caches? 1 write miss + 3 read misses = 4 misses

Problem M11.9.B

Does Ben's delayed invalidation optimization violate cache coherence rules? Please explain your answer in one or two sentences.

No. There are two coherence rules:

(1) Write propagation: Writes eventually become visible to all processors.

➔ Yes. With delayed invalidation, writes from other processors become visible when a local miss, either a read miss (I->S) or a write miss (I->M or S->M), occurs.

(2) Write serialization: Writes to the same location are serialized, and all processors see them in the same order.

➔ Yes. With delayed invalidation, all processors still see the same global ordering of writes.

Problem M11.9.C

Suppose the original system guarantees sequential consistency. Does adding the delayed invalidation optimization break sequential consistency? Please explain your answer in one or two sentences. If your answer is yes, please provide a sequence of load/store operations that violates sequential consistency.

No. The system is sequential consistent if the following conditions are met:

(1) The result of any execution is the same as if the operations of all the processors were executed in some sequential order. In other words, all processors agree on a global ordering of reads and writes.

➔ Yes. With delayed invalidation, the reads that happen before the invalidation is processed can be seen as reads happening before the write that causes BusRdX. Those reads hit in the cache and are not visible to other processors.
For example, in Question 1, all processors agree on a logical ordering:
I0 -> I2 -> I4 -> I1 -> I3 -> I5 -> I6.

(2) The operations of each individual processor appear in program order.

➔ Yes. Delayed invalidation only tries to re-order reads from other processors' writes.

Problem M11.9.D

Ben only applies delayed invalidation on cache blocks that are in the S state. When a cache observes a Bus Read Exclusive message (BusRdX) and the associated cache block is in the Modified (M) state, it sends out the data in response to a BusRdX message and changes the cache state to Invalid (I).

Is it possible to delay invalidation when the cache block is in the Modified (M) state? If it is not, please explain why. If it is possible, please describe how to make delayed invalidations work when the block is in the M state. In other words, please describe the actions the cache needs to take when the cache observes a BusRdX message, how to handle subsequent read and write accesses if the invalidation is delayed, and when the invalidation needs to be processed.

When observing a BusRdX message, change the cache state from M to S and send the data value to the bus. The invalidation needs to be processed before processing any subsequent read or write miss.

Problem M11.10: Cache Coherence (Spring 2015 Quiz 3, Part C)

Problem M11.10.A

Ben designs an architecture that does not have the atomic compare-and-swap (`CAS`) instruction but has load-reserve (`LR`) and store-conditional (`SC`) instructions.

Help Ben implement a Boolean compare-and-swap instruction `BCAS old, new, Imm(base)` using load-reserve and store-conditional instructions:

```
LR rs, Imm(rt):
     <flag, addr> ← <1, rt + Imm>
     rs ← Memory[rt + Imm]

SC rs, Imm(rt):
     If <flag, addr> == <1, rt + Imm>:
          Memory[rt + Imm] ← rs
          rs ← 1                    # Succeed
     Else:
          rs ← 0                    # Fail
```

BCAS is a simplified CAS instruction that only deals with values 0 and 1. You can use temporary registers (`tmp1, tmp2, tmp3…`) and any algorithmic, logical, memory, and branch instructions in the MIPS instruction set.

```
BCAS old, new, Imm(base):
          LR   tmp1, Imm(base)   # load M[Imm+base] into tmp1
          BNE  tmp1, old, fail   # if tmp1 != old, go to fail
          MOV  tmp2, new         # copy new to tmp2
          SC   tmp2, Imm(base)   # try to store tmp2
          BNEZ tmp2, skip        # check if SC succeeds
          NOR  tmp1, tmp1, tmp1  # invert the value of tmp1
                                   (since M[Imm+base] is changed)
  fail:   MOV  old,  tmp1        # copy tmp1 to old
  skip:   NOP
```

Problem M11.10.B

Suppose the hardware where the shared-memory queue from Handout #15 is executed has a weak consistency model that relaxes all the orderings of reads and writes. Give an example of memory orderings between the producer and consumer that would result in incorrect behavior. *Please fully explain your answer to get full credit.*

Your memory ordering example should look something like:
`P1, C2, P2, C4, P4, C5, C7, C9, C10`

If the tail write is visible to the consumer before the message write, then we have a problem. Thus any sequence that contains the subsequence:

`P4, C7, P2`

will read an invalid message.

Problem M11.10.C

Please add the minimum number of memory fences ($FENCE_{WR}$, $FENCE_{RW}$, $FENCE_{WW}$, or $FENCE_{RR}$) to the producer and consumer codes to ensure correctness with a weak consistency model. Please explain your answer fully.

Code for producer to enqueue a message:

```
P1:  LD   R3, 0(R2)   # get tail pointer

P2:  ST   R1, 0(R3)   # write message to tail

P3:  ADD R3, R3, 4    # update tail pointer
     FENCE_WW # don't update tail before writing message
P4:  ST   R3, 0(R2)
```

Code for consumer to dequeue a message:

```
C1: SpinLock: MOV  R6, R0          # set R6 to 0

C2:           CAS  R6, R5, 0(R4) # try to acquire lock

C3:           BNEZ R6, SpinLock
     FENCE_WR # don't read head pointer before getting lock
C4:           LD   R7, 0(R2)      # get head pointer

C5: Retry:    LD   R8, 0(R3)      # get tail pointer

C6:           BEQ  R7, R8, Retry # is there a message?
     FENCE_RR # don't read message before tail is updated
C7:           LD   R1, 0(R7)      # read message from queue

C8:           ADD  R7, R7, 4      # update head pointer

C9:           ST   R7, 0(R2)
     FENCE_WW # don't release lock before updating head
C10:          ST   R0, 0(R4)      # release lock
```