## Problem M11.1: 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 M11.1.A

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

### 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

### 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.

### 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 Handout #13? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

## Problem M11.2: 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 #13. 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 M11.2.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 M11.2.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 M11.2-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).

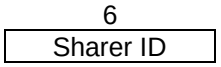



**Figure M11.2-A**

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 M11.2.C**

Ben thinks that he can improve his original scheme by adding an extra "**global bit**" to **S** as shown in Figure M11.2-B (**global-bit scheme**). The global bit is set when there is more than 1 processor sharing the data, and zero otherwise.

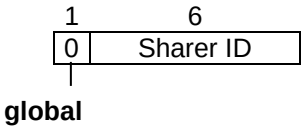



**Figure M11.2-B**

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

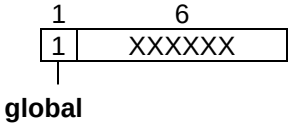



**Figure M11.2-C**

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 M11.3: 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 #13 (as well as Lecture 15). 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 M11.3.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><br><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 M11.3.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 M11.3.C**

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

**Problem M11.3.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 M11.4: Snoopy Cache Coherent Shared Memory

In this problem, we investigate the operation of the snoopy cache coherence protocol in Handout #14.

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

In Table M11.4-1, M11.4-2, and M11.4-3, 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 M11.4-1 has been completed for you. Make sure the answers in this table make sense to you.

## Problem M11.4.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 M11.4.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 M11.4-1**

| 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 M11.4-2**

| 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 M11.4-3**

## Problem M11.5: Snoopy Cache Coherent Shared Memory

This problem improves the snoopy cache coherence protocol presented in Handout #14. 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 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 | | |
| | | CPU write | | |
| | | replace | | |
| | | **CR** | | |
| | | **CRI** | | |
| | | **CI** | | |
| | | **WR** | | |
| | | **CWI** | | |

**Problem M11.5.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 M11.5.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 M11.6: Snoopy Caches

This part explores multi-level caches in the context of the bus-based snoopy protocol discussed in Lecture 14 (2017). 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 1/4$^{th}$ 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 M11.6.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 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?

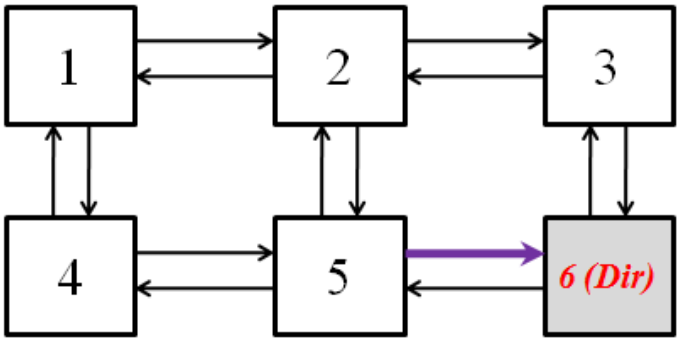
### Problem M11.6.C

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.

# 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 |
|---|---|---|
| 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 M11.7.B**

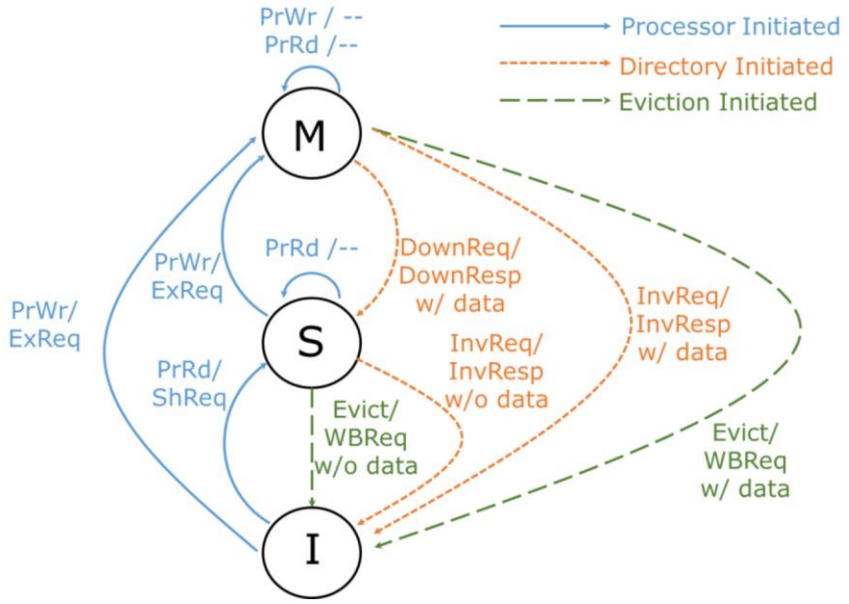
For the directory protocol, we assume the message passing to be FIFO, meaning protocol messages with the same source and destination are always received in the same order as that in which they were sent. Now suppose messages can be delivered out-of-order for the same source and destination pairs. Describe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

**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.

## Problem M11.8: Cache Coherence (Spring 2020 Quiz 2, Part B)

Ben Bitdiddle wants to study design tradeoffs in a directory-based MSI coherence protocol. Ben starts by considering the directory-based MSI protocol presented in lecture. The protocol is described in the quiz handout and summarized in this cache-side state transition diagram:




In this protocol, evictions of a cache line in the S state require sending a WBReq (without data) to notify the directory, so the directory can remove the cache from the sharer set.

Ben thinks that he can reduce the number of messages sent on the network during cache evictions. Ben wants to **silently drop** cache lines when evicting cache lines in the S state, sending no message on the network. This means that a cache line can move from S to I without informing the directory.

### Problem M11.8.A

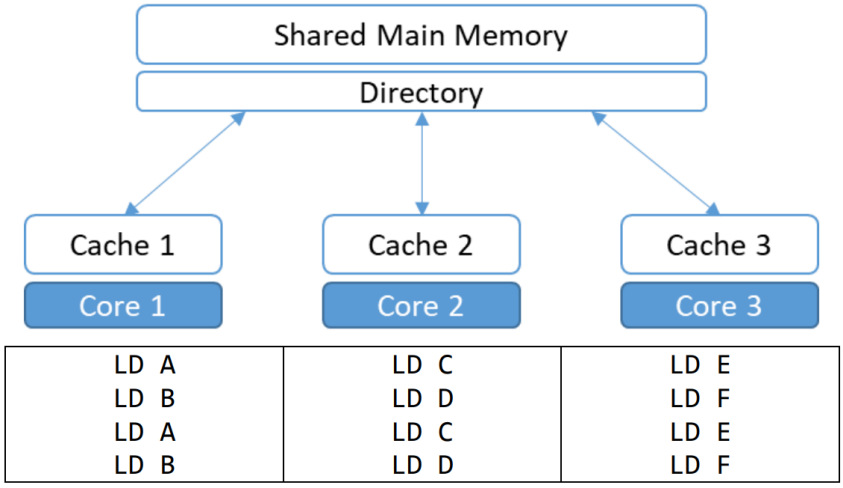
Consider a machine with two cores, where each core has a private cache that uses Ben's proposal for silent drops. Suppose a cache line A is in S state and it is in Core 0's cache. Core 0's cache evicts line A, silently dropping it. The directory still has Core 0 in the sharer set for cache line A.

(a) Assume that after the silent drop by Core 0's cache, Core 0 performs a read of the evicted cache line A. To reobtain the cache line, Core 0's cache sends a ShReq to the directory. Assume there have been no writes to cache line A. What network message, if any, should the directory send to respond to the ShReq to make the protocol work?

(b) Assume that after the silent drop by Core 0's cache, Core 1 performs a write to cache line A. Core 1's cache sends an ExReq to the directory. Since Core 0 is in the sharer set, the directory sends an InvReq to Core 0. What network message, if any, should Core 0's cache send when receiving the InvReq while the requested cache line is in the I state?

**Problem M11.8.B**

Consider the three-core system below. Each core has a private cache *that can only hold a single cache line*, and the caches start out empty. Each core runs a thread that performs four reads, alternating between reading two addresses. Each thread accesses different addresses on different cache lines. (Core 1's thread reads addresses A and B, Core 2's thread reads C and D, etc.) Due to evictions, all 12 accesses will be cache misses. *Assume the directory has unlimited capacity.*

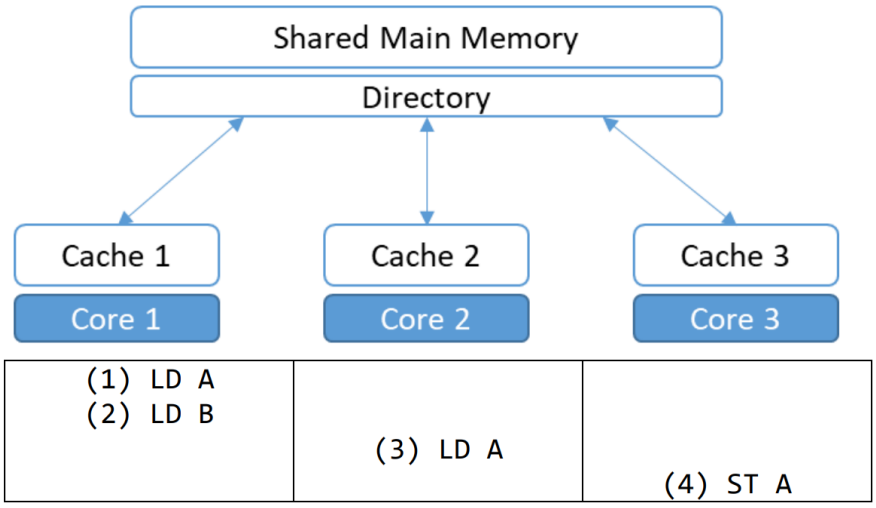



(a) How many writeback requests are sent in the original MSI protocol from lecture?

(c) How many writeback requests are sent with Ben's proposal for silent drops?

**Problem M11.8.C**

Consider a different workload where the threads access shared data, as shown below. The number in parenthesis indicates the global order of the accesses (i.e. Core 1's LD A happens before LD B, which happens before Core 2's LD A, etc.). Each access completes before the next one begins. Again, assume all caches start empty and each cache can only hold a single line at a time.




(a) How many WBReq and InvReq messages are sent in the original MSI protocol from lecture? Count an invalidation of multiple caches as multiple requests. Do not count response messages.

(b) How many writeback requests and invalidation requests are sent with Ben's proposal for silent drops?

**Problem M11.8.D**

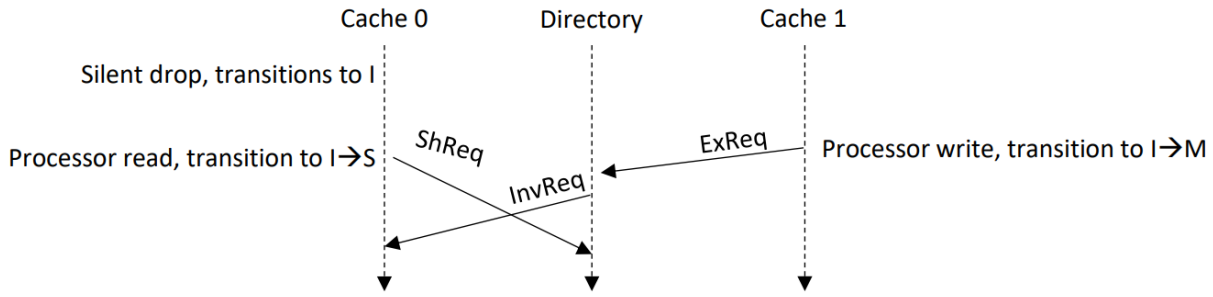
So far, we assumed each coherence transaction completes before the next transaction begins. Alyssa P. Hacker points out that Ben's silent drops make it harder to solve races when there are concurrent coherence requests. To see this, we will consider two scenarios. In each scenario, Core 0 silently drops a line from its private cache that it later needs to read, while Core 1 attempts to write to the **same cache line**. In each scenario, Core 0 receives an InvReq, and you must pick **one** of the three following answers:

A: Acknowledge the InvReq by sending an InvResp, remaining in the I→S transient state to wait for a later ShResp.

B: Buffer or NACK the InvReq, waiting for a ShResp to first serve its read before performing an invalidation.

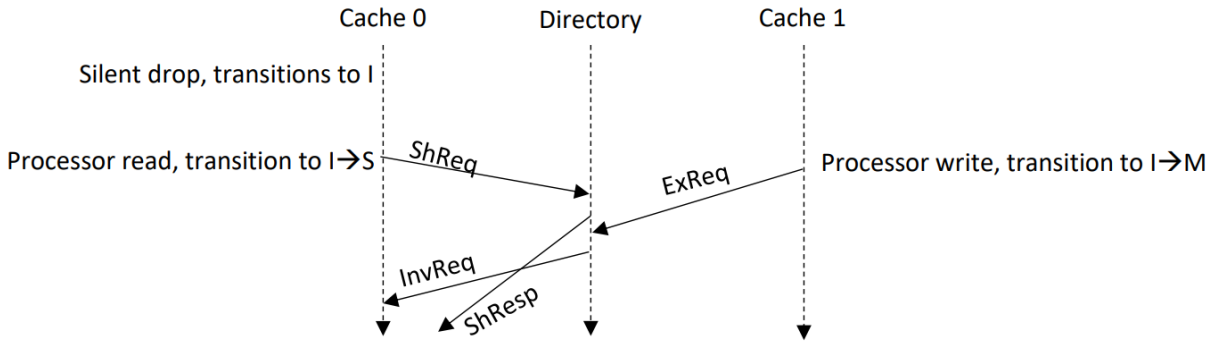
C: Performing either of A or B will result in correct behavior.

(a) In this scenario, the directory receives Cache 1's ExReq before Cache 0's ShReq. While Cache 0 is waiting for a ShResp, it receives a InvReq from the directory.




To maintain coherence, what action should Cache 0 take in response to the InvReq while in the I→S transient state?

(b) In this scenario, the directory receives Cache 0's ShReq before Cache 1's ExReq. The directory sends a ShResp to Core 0 followed by a InvReq. However, the ShResp is traveling slowly in the network, and Cache 0 receives the InvReq before the ShResp.




To maintain coherence, what action should Cache 0 take in response to the InvReq while in the I→S transient state?

## 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: | B: | A: | B: |
| After P2 reads A | A: | B: | A: | B: |
| After P1 writes A | A: | B: | A: | B: |
| After P2 reads A | A: | B: | A: | B: |
| After P2 reads B | A: | B: | A: | B: |
| After P2 reads A | A: | B: | A: | B: |

How many misses occur in the two caches?

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: | B: | | A: | B: | |
| After P2 reads A | A: | B: | | A: | B: | |
| After P1 writes A | A: | B: | | A: | B: | |
| After P2 reads A | A: | B: | | A: | B: | |
| After P2 reads B | A: | B: | | A: | B: | |
| After P2 reads A | A: | B: | | A: | B: | |

How many misses occur in the two caches?

**Problem M11.9.B**

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

**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.

**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.

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

Please use Handout #15 to answer the questions in this part.

### 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.

**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`

**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

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

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?

C7:           LD   R1, 0(R7)     # read message from queue

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

C9:           ST   R7, 0(R2)

C10:          ST   R0, 0(R4)     # release lock
```