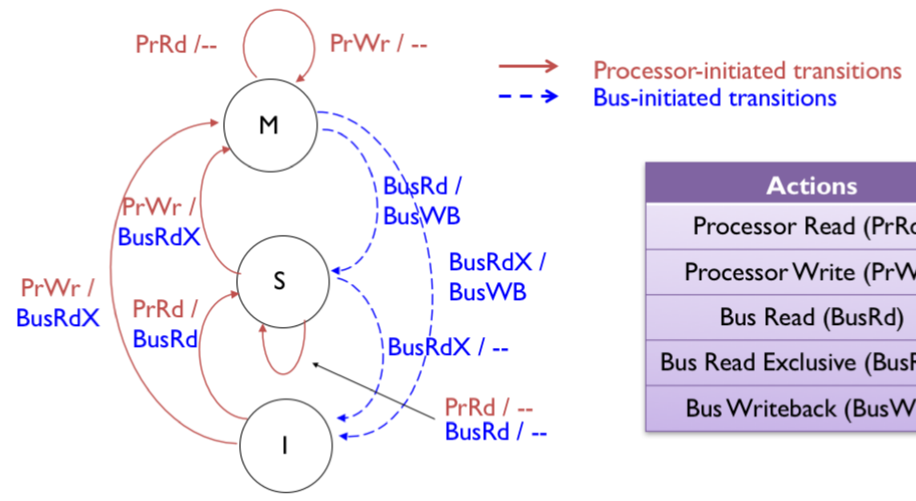
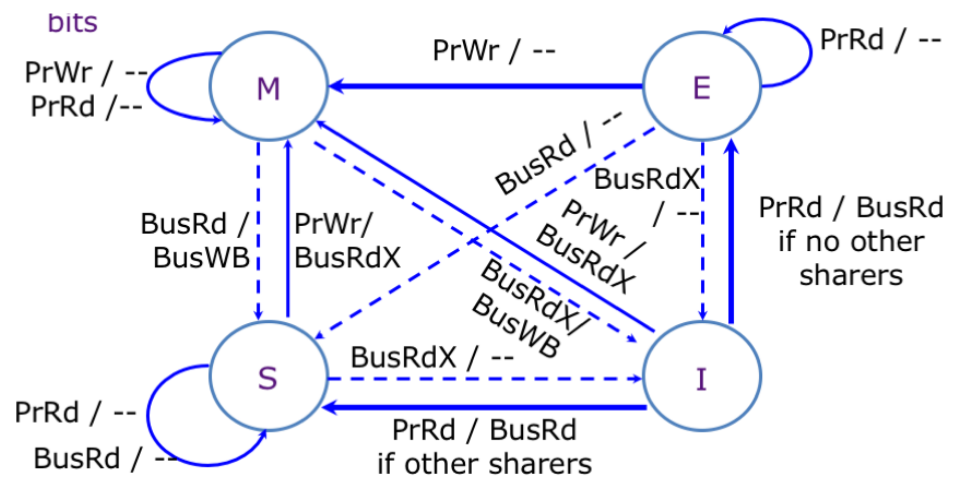
6.004 Worksheet
L25 – Cache Coherence

MSI State Transition Diagram



Actions
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Read Exclusive (BusRdX)
Bus Writeback (BusWB)

MESI State Transition Diagram



Problem 1.

A multicore processor has multiple threads each running on a different core. The threads share memory (but has its own stack).

We introduce a new instruction `swap` that supports an atomic read-modify-write operation on a memory location, where the processor's cache coherence protocol guarantees that no other `swap` operation can access the same memory location at the same time.

```
swap rs1, literal, rd
   PC <= PC + 4
   EA <= Reg[rs1] + literal
   TMP     <= Mem[EA]      // atomic with following line
   Mem[EA] <= Reg[rd]      // atomic with preceding line
   Reg[rd] <= TMP
```

The `swap` instruction can implement locks as a binary semaphore.

```
      li x1, 0           // WAIT operation on lock
loop: swap x0, lock, x1
      beqz x1, loop

      … critical section …

      li x1, 1           // SIGNAL operation on lock
      sw x1, lock(x0)
```

Using the code segments shown above, write code for the more general `Signal(s)` and `Wait(s)` operations on a semaphore, where `s:` is a location in the shared memory. `Signal(s)` and `Wait(s)` need to work even if different cores are calling `Signal` or `Wait` at the same time.

Code for Wait(s):

```
again: li x1, 0  // lock mutex
loop:  swap x0, lock, x1
       beqz x1, loop

       lw x1, s(x0)
       bnez x1, gotit
       li x1, 1     // unlock mutex
       sw x1, lock(x0)
       beqz x0, again // repeat

gotit: addi, x1, x1, -1
       sw x1, s(x0)
       li x1, 1  // unlock mutex
       sw x1, lock(x0)
```

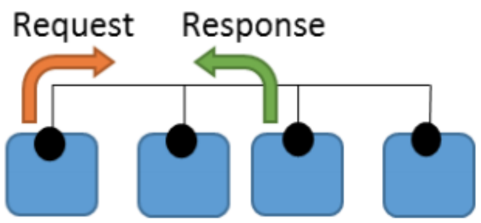
Code for Signal(s):

```
       li x1, 0  // lock mutex
loop:  swap x0, lock, x1
       beqz x1, loop

       lw x1, s(x0)
       addi x1, x1, 1
       sw x1, s(x0)

       li x1, 1  // unlock mutex
       sw x1, lock(x0)
```

Problem 2.

Snoopy coherence protocols rely on broadcast communication to detect sharing and updates. These are conventionally implemented using bus networks that allow for one message to be sent at a time to all nodes on the network.

(A) Ben Bitdiddle is implementing a bus-based snoopy coherence protocol. One fifth of instructions access memory, and one quarter of these miss in the core's local cache (either because the line is invalid or doesn't have necessary permissions). Assuming each memory operation consists of a request and acknowledgement, the network traffic per core is therefore:

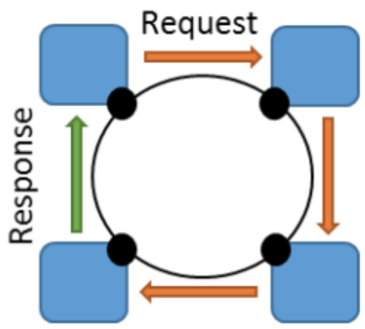


$$\frac{1}{5} \cdot \frac{1}{4} \cdot 2 = \frac{1}{10}\ messages/instruction$$

Assume all bus messages take a single cycle. Considering only the message carrying capacity of the shared bus, how many cores can the bus support?

The bus has an aggregate throughput of 1 message per cycle. A memory operation requires 2 messages on 1/20 of instructions, or 1/10 messages per cycle. The number of cores this system can support is 1 = N/10 so N = 10.

(B) Ben needs to build a larger system than the bus network will allow, so he changes the system to use a unidirectional ring network. In this design, the core issuing the memory operation sends the request around the ring, and each node along the way either forwards the request or replaces it with its response. Assuming a single-cycle per hop in the network, at how many cores will this design saturate?



The ring with N cores has an aggregate throughput of N messages per cycle.

Each memory operation requires one circuit around the ring, or N messages. Each core produces one request every 20 messages, so the number of messages generated per core is N/20.

Thus, the number of cores is N = N/(N/20) = 20.

Problem 3.

Ben Bitdiddle is designing a snoopy-based, write-invalidate MSI protocol for write-back caches. 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
```

(A) Assume blocks A and B do not conflict in the cache. Using **MSI**, fill in the table showing the required transactions and the cache line states for A and B *after* each access.

Access	*Shared bus transaction*	*Processor P1's cache*		*Processor P2's cache*	
Initial state		A: I	B: I	A: I	B: I
After P2 reads A	P2: BusRd(A)	A: I	B: I	A: S	B: I
After P1 writes A	P1: BusRdX(A)	A: M	B: I	A: I	B: I
After P2 reads A	P2: BusRd(A) → P1: BusWB(A,val)	A: S	B: I	A: S	B: I
After P1 writes A	P1: BusRdX(A)	A: M	B: I	A: I	B: I
After P2 reads A	P2: BusRd(A) → P1: BusWB(A,val)	A: S	B: I	A: S	B: I
After P2 reads B	P2: BusRd(B)	A: S	B: I	A: S	B: S
After P2 reads A	none	A: S	B: I	A: S	B: S

(B) If Ben switches to a **MESI protocol**, which bus transactions and cache states would be different?

In the second row, P2's cache state for A would be "E".
In the last two rows, P2's cache would for B would be "E".

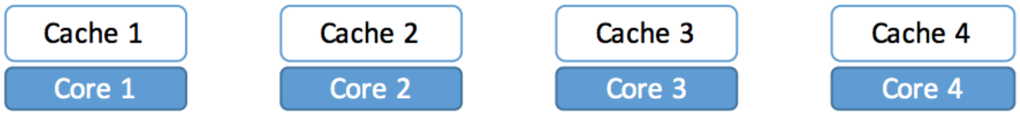
(C) Briefly describe a scenario where the additional E state in the MESI protocol would eliminate a shared bus transaction.

If there was an additional access "P2: write B", no BusRdX(B) transaction would be required since at the time P2 has exclusive access to B.

Problem 4.

We want to study the tradeoffs between the standard directory-based MSI and MESI coherence protocols. The state transition diagrams for the two protocols are shown on the front page of this handout.

(A) Consider the four-core system below. Each core has a private cache, and caches are kept coherent with a directory protocol. Each core runs a thread that issues a load followed by a store to a single address, as shown below. Each thread accesses a different address (core 1's thread accesses A, core 2's thread accesses B, etc.). These thread-private addresses are on different cache lines. The number in parenthesis indicates the global order of the accesses (i.e. LD A happens before ST A, which happens before LD B, etc.). Each access completes before the next one begins.

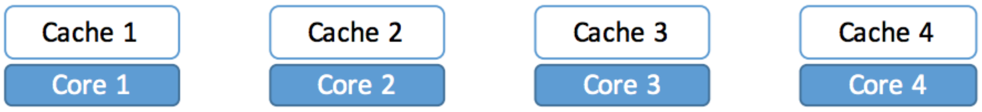


`(1) LD A`	`(3) LD B`	`(5) LD C`	`(7) LD D`
`(2) ST A`	`(4) ST B`	`(6) ST C`	`(8) ST D`

For this sequence of 8 accesses, provide in the table below the total number of each type of bus requests for the MSI and MESI protocols. *Assume all caches are initially empty*, i.e., the initial states for each cache line is "I".

Protocol	*# of BusRd*	*# of BusRdX*	*# of BusWB*
MSI	4	4	0
MESI	4	0	0

(B) Consider a different program where each thread reads globally shared data:



`(1) LD A`	`(2) LD A`	`(3) LD A`	`(4) LD A`

For this sequence of 4 accesses, fill in the table below for the MSI and MESI protocols. Ignore coherence responses. Assume all caches are initially empty.

Protocol	*# of BusRd*	*# of BusRdX*	*# of BusWB*
MSI	4	0	0
MESI	4	0	0