6.004 Worksheet
L21 – Cache Coherence

Implementing Cache Coherence

- Coherence protocols must enforce two rules:
  - **Write propagation**: Writes eventually become visible to all processors
  - **Write serialization**: Writes to the same location are serialized (all processors see them in the same order)
- How to ensure write propagation?
  - **Write-invalidate protocols**: Invalidate all other cached copies before performing the write
  - **Write-update protocols**: Update all other cached copies after performing the write
- How to ensure write serialization?
  - **Snooping-based protocols**: All caches observe each other’s actions through a shared bus
  - **Directory-based protocols**: A coherence directory tracks contents of private caches and serializes requests

Snooping-Based Coherence

- Bus provides serialization point
  - Broadcast, totally **ordered**
  - Each cache controller “snoops” all bus transactions
  - Controller updates state of cache in response to processor and snoop events and generates bus transactions
- Snoopy protocol (FSM)
  - State-transition diagram
  - Actions

ANSWERS
Problem 1.

Consider a multicore processor with multiple threads each running on a different core. The threads access a shared memory that holds their code and data. Note that each thread has been given a different region of the shared memory to hold its stack.

In lecture we introduced 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.

\[
\text{SWAP}(Ra, \text{literal}, Rc)
\]

\[
\begin{align*}
\text{PC} & \leftarrow \text{PC} + 4 \\
\text{EA} & \leftarrow \text{Reg}[Ra] + \text{literal} \\
\text{TMP} & \leftarrow \text{Mem}[EA] \quad \text{// atomic with following line} \\
\text{Mem}[EA] & \leftarrow \text{Reg}[Rc] \quad \text{// atomic with preceding line} \\
\text{Reg}[Rc] & \leftarrow \text{TMP}
\end{align*}
\]

The SWAP instruction can implement mutual exclusion as using the memory location \text{lock} as a binary semaphore.

\[
\text{CMOVE}(0, R0) \quad \text{// WAIT operation on lock}
\]

\[
\text{loop: SWAP(R31, lock, R0)}
\]

\[
\text{BEQ(R0, loop)}
\]

\[
\ldots \text{critical section} \ldots
\]

\[
\text{CMOVE}(1, R0) \quad \text{// SIGNAL operation on lock}
\]

\[
\text{ST(R0, lock, R31)}
\]

Using the SWAP instruction to implement mutual exclusion as shown above, write code sequences for the more general Signal(s) and Wait(s) operations on an integer semaphore \(s\), where \(s\) is location in the shared memory. The goal is to ensure correct operation of Signal(s) and Wait(s) even if different cores are running the Signal or Wait code at the same time.

<table>
<thead>
<tr>
<th>Code for Wait(s):</th>
<th>Code for Signal(s):</th>
</tr>
</thead>
</table>
| \[
\text{again: CMOVE}(0, R0) \quad \text{// lock mutex}
\]
| \[
\text{loop: SWAP(R31, lock, R0)}
\]
| \[
\text{BEQ(R0, loop)}
\] |
| \[
\text{LD}(s, R0)
\] |
| \[
\text{BNE}(R0, \text{gotit})
\] |
| \[
\text{CMOVE}(1, R0) \quad \text{// unlock mutex}
\] |
| \[
\text{ST}(R0, \text{lock}, R31)
\] |
| \[
\text{BR(again)}
\] |
| \[
\text{CMOVE}(0, R0) \quad \text{// lock mutex}
\]
| \[
\text{loop: SWAP(R31, lock, R0)}
\]
| \[
\text{BEQ(R0, loop)}
\] |
| \[
\text{LD}(s, R0)
\] |
| \[
\text{ADDC(R0, 1, R0)}
\] |
| \[
\text{ST}(R0, s)
\] |
| \[
\text{CMOVE}(1, R0) \quad \text{// unlock mutex}
\] |
| \[
\text{ST}(R0, \text{lock}, R31)
\] |
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:

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

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:

<table>
<thead>
<tr>
<th>Access</th>
<th>Shared bus transaction</th>
<th>Processor P1’s cache</th>
<th>Processor P2’s cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td></td>
<td>A: I B: I</td>
<td>A: I B: I</td>
</tr>
<tr>
<td>After P1 writes A</td>
<td>P1: BusRdX(A)</td>
<td>A: M B: I</td>
<td>A: I B: I</td>
</tr>
<tr>
<td>After P2 reads A</td>
<td>P2: BusRd(A) →</td>
<td>A: S B: I</td>
<td>A: S B: I</td>
</tr>
<tr>
<td>After P2 reads A</td>
<td>P2: BusRd(A) →</td>
<td>A: S B: I</td>
<td>A: S B: I</td>
</tr>
<tr>
<td>After P2 reads B</td>
<td>P2: BusRd(B)</td>
<td>A: S B: I</td>
<td>A: S B: S</td>
</tr>
<tr>
<td>After P2 reads A</td>
<td>none</td>
<td>A: S B: I</td>
<td>A: S B: S</td>
</tr>
</tbody>
</table>

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

(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.

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”.

<table>
<thead>
<tr>
<th>Protocol</th>
<th># of BusRd</th>
<th># of BusRdX</th>
<th># of BusWB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>MESI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(B) Consider a different program where each thread reads globally shared data, as shown below.

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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th># of BusRd</th>
<th># of BusRdX</th>
<th># of BusWB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MESI</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**Problem 5. Cache Coherence**

Consider a multicore system where processors P1 and P2 each have a private, snooping-based, write-back cache that uses the MESI coherence protocol. Suppose the processors make a sequence of accesses to the shared variable A in the order listed below, i.e., P1 completes four accesses, followed by two accesses by P2:

P1: LD A
P1: ST A
P1: LD A
P1: ST A
P2: LD A
P2: ST A

Please fill in the table below showing the shared-bus operations (if any) required to complete each access (one of BusRd, BusRdX, or BusWB). If there is no shared-bus transaction, leave the entry blank. Also give the state of the cache line for location A in each of P1’s and P2’s cache after the access is complete (one of M, E, S, or I)

<table>
<thead>
<tr>
<th>Access</th>
<th>Shared bus transaction(s)</th>
<th>P1’s cache state for A</th>
<th>P2’s cache state for A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td></td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>After P1: LD A</td>
<td>P1: BusRd(A)</td>
<td>E</td>
<td>I</td>
</tr>
<tr>
<td>After P1: ST A</td>
<td></td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>After P1: LD A</td>
<td></td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>After P1: ST A</td>
<td></td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>After P2: LD A</td>
<td>P2: BusRd(A), P1: BusWB(A, ...)</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>After P2: ST A</td>
<td>P2: BusRdX(A)</td>
<td>I</td>
<td>M</td>
</tr>
</tbody>
</table>

Here’s the state transition diagram for the MESI cache coherence protocol: