MOESI Cache Coherency

With the MOESI concurrency protocol implemented, accesses to cache accesses appear *serializable*. This means that the result of the parallel cache accesses appear the same as if there were done in serial from one processor in some ordering.

<table>
<thead>
<tr>
<th>State</th>
<th>Cache up to date?</th>
<th>Memory up to date?</th>
<th>Others have a copy?</th>
<th>Can respond to other’s reads?</th>
<th>Can write without changing state?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes, Required</td>
<td>Yes</td>
</tr>
<tr>
<td>Owned</td>
<td>Yes</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Yes, Optional</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Yes</td>
<td>No</td>
<td>Maybe</td>
<td>Yes, Optional</td>
<td>No</td>
</tr>
<tr>
<td>Shared</td>
<td>Yes</td>
<td>Maybe</td>
<td>Maybe</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Invalid</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

1. Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache with one cache block and a two cache block memory. Assume the MOESI protocol is used, with write- back caches, write-allocate, and invalidation of other caches on write (instead of updating the value in the other caches).

<table>
<thead>
<tr>
<th>Time</th>
<th>After Operation</th>
<th>P1 cache state</th>
<th>P2 cache state</th>
<th>Memory @ 0 up to date?</th>
<th>Memory @ 1 up to date?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1: read block 1</td>
<td>Exclusive (1)</td>
<td>Invalid</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>1</td>
<td>P2: read block 1</td>
<td>Owned (1)</td>
<td>Shared (1)</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>2</td>
<td>P1: write block 1</td>
<td>Modified (1)</td>
<td>Invalid</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>3</td>
<td>P2: write block 1</td>
<td>Invalid</td>
<td>Modified (1)</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>4</td>
<td>P1: read block 0</td>
<td>Exclusive (0)</td>
<td>Modified (1)</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>5</td>
<td>P2: read block 0</td>
<td>Owned (0)</td>
<td>Shared (0)</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>6</td>
<td>P1: write block 0</td>
<td>Modified (0)</td>
<td>Invalid</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>7</td>
<td>P2: read block 0</td>
<td>Owned (0)</td>
<td>Shared (0)</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>8</td>
<td>P2: write block 0</td>
<td>Invalid</td>
<td>Modified (0)</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>9</td>
<td>P1: read block 0</td>
<td>Shared (0)</td>
<td>Owned (0)</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

Concurrency

2. Consider the following function:

```c
void transferFunds(struct account *from, struct account *to, long cents)
{
    from->cents -= cents;
    to->cents += cents;
}
```

a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: if the problem isn’t obvious, translate the function into MIPS first)
Each thread needs to read the “current” value, perform an add/sub, and store a value for from->cents and to->cents. Two threads could read the same “current” value and the later store essentially erases the other transaction at either line.

b. How could you fix or avoid these races? Can you do this without hardware support?

Wrap transferFunds in a critical section, or divide up the accounts array and for loop in a way that you can have separate threads work on different accounts

Midterm Questions:

3. Summer ’12, MT1, Q1f
   In our 32-bit single-precision floating point representation, we decide to convert one significand bit to an exponent bit. How many **denormalized numbers** do we have relative to before? (Circle one)
   
   - More
   - Fewer
   - Half as many because lost a significand bit

Rounded to the nearest power of 2, how many denorm numbers are there in our new format? (Answer in IEC format)

22 significand bits + sign bit but not counting ±0, so exactly $2^{23} - 2$ denoms

___8 Mebi #s___
4. Fall ’14, Final, M2a-d

Assume we are working in a 32-bit virtual and physical address space, byte-address memory. We have two caches: cache A is a direct-mapped cache, while cache B is fully associative with LRU replacement policy. Both are 4 KiB caches with 256 B blocks and write-back policy. **Show all work!**

a) For cache B, calculate the number of bits used for the Tag, Index, and Offset: T:____ I:____ O:____

Consider the following code:

```c
uint32_t H[32768]; // 32768 = 2^15. H is block-aligned.

for (uint32_t i = 0; i < 32768; i += 2048) H[i] += 1;
for (uint32_t i = 1; i < 32768; i += 2048) H[i] += 2;
```

**Read is a comp miss, write a hit within the loop 50%**

b) If the code were run on cache A, what would the hit rate be? _________________

**For 1st loop miss on read, hit on write. For 2nd loop hits 75%**

c) If the code were run on cache B, what would the hit rate be? _________________

d) Consider several modifications, each to the original cache A. How much will the modifications change the hit-rate and why?

i. Same cache size, same block size, 2-way associativity
   **Associativity too low, capacity misses first entries replaced before 2nd loop -- no change**

ii. Double the cache size, same block size
   **Still too many replacements first entries replaced before 2nd loop—no change**

iii. Same cache size, block size is reduced to 8B
   **Still capacity misses first entries replaced before 2nd loop -- No change**