MOESI Protocol

- Invalid
  - Read Miss Exclusive
    - Shared
      - Read Hit
    - Write Miss (WB/Write Alloc)
      - Modified
        - Write Hit
          - Read Hit Write Hit
        - Probe Read Hit
          - Read Hit
            - Owned
              - Probe Write Hit
                - Invalid
                  - Read Hit
                    - Exclusive
                      - Write Hit
                        - Read Hit
                          - Owned
                            - Probe Read Hit
                              - Read Hit
                                - Shared
                                  - Probe Write Hit
                                    - Invalid
                                      - Read Hit
                                        - Exclusive
                                          - Probe Write Hit
                                            - Modified
                                              - Read Hit Write Hit
                                              - Probe Read Hit
                                                - Probe Write Hit
                                                  - Shared
                                                    - Probe Read Hit
                                                      - Probe Write Hit
                                                        - Owned
                                                          - Probe Read Hit
                                                            - Modified
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 write without changing state?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Owned</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Shared</td>
<td>Yes</td>
<td>Maybe</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Invalid</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P1: write block 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P2: write block 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P1: read block 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P2: read block 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>P1: write block 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P2: read block 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>P2: write block 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>P1: read block 0</td>
<td></td>
<td></td>
<td></td>
<td></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)

   b. How could you fix or avoid these races? Can you do this without hardware support?
Thread Level Parallelism
Assume n is very large for every part!

```c
#pragma omp parallelism
{  /* code here */
}
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  /* code here */
}
```

1. For the following snippets of code below, circle one of the following to indicate what issue, if any, the code will experience. Then provide a short justification.

Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume arr is an int array with length n.

a) // Set element i of arr to i
   #pragma omp parallel
   { for (int i = 0; i < n; i++)
       arr[i] = i; }

   Sometimes incorrect Always incorrect Slower than serial Faster than serial

b) // Set arr to be an array of Fibonacci numbers.
   arr[0] = 0;
   arr[1] = 1;
   #pragma omp parallel for
   for (int i = 2; i < n; i++)
       arr[i] = arr[i-1] + arr[i-2];

   Sometimes incorrect Always incorrect Slower than serial Faster than serial

c) // Set all elements in arr to 0;
   int i;
   #pragma omp parallel for
   for (i = 0; i < n; i++)
       arr[i] = 0;

   Sometimes incorrect Always incorrect Slower than serial Faster than serial