MOESI Cache Coherence

<table>
<thead>
<tr>
<th>State</th>
<th>Cache up to date?</th>
<th>Memory up to date?</th>
<th>Others have 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>YES</td>
<td>NO</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>

With the MOESI concurrency protocol implemented, accesses to cache accesses appear \textit{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 \textit{some} ordering.

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>Owner(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>

2. Consider if we run the following two loops in parallel (as two threads on two processors).

```java
for(int i = 0; i < N; i+=2) array[i] += 1;
for(int j = 1; j < N; j+=2) array[j] += 2;
```

Would we expect more, less, or the same number of cache misses than if we were to run this serially (assume each processor has its own cache and all data is invalid to start with)? \textit{Possibly more since both are modifying the same cache blocks causing invalidation of each others blocks.}
Concurrency

1. 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 called simultaneously from two (or more) threads on the same account? (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.

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

      Could fix by adding a lock to each struct account. Without hardware support, can still have data race when reading lock = 0 and have multiple threads that think they have the lock. Adding hardware support to implement atomic read/write memory operations fixes this problem.

2. A reader-writer lock is a lock which can either be obtained exclusively by one thread (a “write lock”) or shared by an arbitrary number of threads (who share a “read lock”). Consider implementing a reader-writer lock by choosing the following values for the lock (which will be one MIPS word):

   - 0: unlocked
   - positive number: read-locked; lock value is number of readers
   - -1: write-locked

Write MIPS assembly implementations of write_lock, write_unlock, read_lock, and read_unlock. When the lock cannot be obtained, have your functions loop until it becomes free.

```c
write_lock:     ll $t0, 0($s0)
bne $t0, $0, write_lock
addi $t0, $0, -1
sc $t0, 0($s0)
beq $t0, $0, write_lock

write_unlock:   sw $0, 0($s0)
```

```c
read_lock:      ll $t0, 0($s0)
slt $t1, $t0, $0
bne $t1, $0, read_lock
addi $t0, $t0, 1
sc $t0, 0($s0)
beq $t0, $0, read_lock
```

```c
write_unlock:   ll $t0, 0($s0)
addi $t0, $t0, -1
sc $t0, 0($s0)
beq $t0, $0, read_unlock
```

```c
read_unlock:    ll $t0, 0($s0)
addi $t0, $t0, 1
sc $t0, 0($s0)
beq $t0, $0, read_unlock
```
Combinational Logic

Convert to Boolean (use truth table):
NAND: \( \overline{A} \overline{B} + \overline{A}B + AB \)
XOR: \( \overline{A}B + AB \)
XNOR: \( \overline{A}B + AB \)

How many different two-input logic gates are possible? How many n-input logic gates?
Hint: Think about the truth table.
A truth table with \( n \) inputs has \( 2^n \) rows. Each logic gate has a 0 or 1 in each of these rows. Imagining a function as a \( 2^n \)-bit number, we count \( 2^{2^n} \) total functions, or 16 in the case of \( n = 2 \).

Create an inverter (NOT gate) using only NAND gates. Hint: look at the truth tables for each.

Boolean Logic Tricks

Minimize the following Boolean expressions:

Standard: \((A + B)(A + \overline{B})C\)
\((AA + AB + A\overline{B} + B\overline{B})C = (A + A(\overline{B} + B))C = AC\)

Grouping & Extra terms: \( \overline{ABC} + \overline{A}B\overline{C} + AB\overline{C} + A\overline{B}C + ABC + A\overline{B}C\)
\( \overline{AC}(B + B) + AC(B + B) + AC(B + \overline{B}) = \overline{AC} + AC + AC = \overline{AC} + AC + AC = A + \overline{C} \)
\( = (\overline{A} + A)\overline{C} + A(\overline{C} + C) = A + \overline{C} \) by DeMorgan’s.

DeMorgan’s: \( \overline{A(B\overline{C} + BC)} \)
\( \overline{A} + (\overline{B\overline{C}} + BC) = \overline{A} + (\overline{B\overline{C}})BC = \overline{A} + (B + C)(\overline{B} + \overline{C}) = \overline{A} + B\overline{C} + \overline{B}C \)
Extra practice on your own:

One common Boolean operator is the implication operator (A->B, A implies B, or “if A, then B”). One common function that high-level languages use is the add function. Both can be represented with gates. For these two operations, create truth tables for the input/output values, create a minimized Boolean expression, and lastly create the logic gates to represent them.

<table>
<thead>
<tr>
<th>Implication</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(\sim A)(\sim B) + (\sim A)B + AB
(\sim A)(\sim B) + B
\sim A + B

(\sim X0)(\sim X1)X2+(\sim X0)X1(\sim X2)+[X0](\sim X1)(\sim X2)+X0X1X2=Y0
X0(\sim X1)(\sim X2) + X0(\sim X1)X2 + X0X1(\sim X2) + X0X1X2=Y1