Problem 1 (5.3):

a) Private Reads: \(0.7 \times 0.03 \times 3\) bus cycles = 0.063 bus cycles
Private Writes: \(0.2 \times 0.03 \times 5\) bus cycles = 0.03 bus cycles
Shared Reads: \(0.08 \times 0.05 \times 3\) bus cycles = 0.012 bus cycles
Shared Writes: \(0.02 \times 0.05 \times 5\) bus cycles = 0.005 bus cycles
Sum = 0.063 + 0.03 + 0.012 + 0.005 = 0.11

\(0.5 \times 0.11 = 0.055\) bus cycles \quad //\text{Because only 50\% are loads \& stores}\n
Instructions: \(0.015 \times 3\) bus cycles = 0.045 bus cycles

Total = 0.045 bus cycles + 0.055 bus cycles = 0.1 bus cycles

Because each processor will hold the bus for 10\% of each bus cycles, 10
processors could be put on this bus to saturate the bus.

Explanation: Read misses take two bus cycles, both for instructions and
data, one cycle for putting the address on the bus and two cycles for the data
to return, 128 bits takes two cycles on a 64 bit wide bus. Writes misses take
5 cycles to complete for a write-through cache. Two cycles are to write
the new value back to memory (only two because the address and first half of the
data can be put on the bus at the same time for writes) and three cycles to
read the new line into the cache because it is write-allocate.

b) Private Reads: \(0.7 \times 0.03 \times (3\) cycles + \(0.3\) \times 2\ cycles) = 0.0756 bus cycles
Private Writes: \(0.2 \times 0.03 \times (3\) cycles + \(0.3\) \times 2\ cycles) = 0.0216 bus cycles
Shared Reads: \(0.08 \times 0.05 \times (3\) cycles + \(0.3\) \times 2\ cycles) = 0.0144 bus cycles
Shared Writes: \(0.02 \times 0.05 \times (3\) cycles + \(0.3\) \times 2\ cycles) = 0.0036 bus cycles
Sum = 0.0756 + 0.0216 + 0.0144 + 0.0036 = 0.1152

\(0.5 \times 0.1152 = 0.0576\) bus cycles \quad //\text{Because only 50\% are loads \& stores}\n
Instructions: \(0.015 \times (3\) cycles) = 0.054 bus cycles

Total = 0.0576 bus cycles + 0.054 bus cycles = 0.1026 bus cycles

Because each processor will hold the bus for 10.26\% of each bus cycle, only
9 processors could be put on the bus before it will saturate.

Explanation: Read and write misses take the same amount of bus cycles for
a write-back cache. Each case acts like a read miss in part a, one cycle for
the address and two for the data. The extra .6 bus cycles is required because
30\% of the time the miss will cause a dirty line to be evicted from the cache
and be written back to memory. This write takes two bus cycles.
Problem 2 (5.4):

For MESI:
- Stream 1: $90 + 1 + 1 + 1 + 90 + 60 + 1 + 1 + 90 + 60 + 1 + 1 = 397$
- Stream 2: $90 + 90 + 90 + 60 + 60 + 60 + 90 + 90 + 1 + 60 + 90 = 781$
- Stream 3: $90 + 90 + 90 + 1 + 60 + 1 + 1 + 1 + 90 + 90 = 514$

For Dragon:
- Stream 1: $90 + 1 + 1 + 1 + 90 + 90 + 1 + 90 + 90 + 90 + 1 + 90 = 635$
- Stream 2: $90 + 90 + 90 + 90 + 90 + 90 + 1 + 1 + 1 + 90 + 90 = 723$
- Stream 3: $90 + 90 + 90 + 1 + 90 + 90 + 90 + 90 + 90 + 90 + 90 = 811$

Explanation:
- Stream 1: The reason that the MESI protocol performs well for Stream 1 is that once a processor gets a modified copy of the line it performs a series of operations on it without any invalidations from other processors. The MESI protocol allows this to occur without any bus transactions. Stream 1 performs much worse for the Dragon protocol because even though each cache has the line for an extended period of time, every time that it writes to the line it must put the line out on the bus for every other processor to update. The Dragon protocol would have performed much better had there been many more reads than writes to the line after each processor gained ownership of the line.
- Stream 2: The Dragon protocol outperformed the MESI protocol for Stream 2 because of the series of reads that were performed at the end of Stream 2 after each processor had a shared copy of the line. This could be done with no bus transactions. The MESI protocol performed poorly for Stream 2 because it could not take advantage of this series of reads because only one processor had a valid copy of the line at the beginning of the reads because these copies had been invalidated by the previous writes. This meant that each processor had to acquire a copy of the line from the bus before it could be read.
- Stream 3: The MESI protocol performed much better for Stream 3 because when processor 1 performed its series of writes no bus transactions were required. The Dragon protocol required the data to be written to the bus on each of these writes. If this had been a series of reads, the Dragon protocol would have performed much better.
Problem 3:

Proc 1
sw r1, r2, #27
lw r3, r2, #27
sw r5, r2, #27
add r4, r3, r3

Proc 2
lw r3, r2, #27
add r4, r3, r3

In this situation, the intended program order of instructions is flowing down the page. At the end of these instructions the values of r3 and r4 should be the same in each processor. The clause “store that produced the value to complete” means that before processor 1 can complete its first load and move on to the store and add instructions, it must stall until the value stored to the memory location by the first store becomes globally visible (observable by the load of processor 2). This ensures that the load for each processor will get the same value into r3, and the values of r3 and r4 will be the same after this code snippet, as intended. If processor 1 did not wait until the first store became globally visible to execute the second store to the same memory location, processor 2’s load might return the value of the second store and cause the values of r3 and r4 to be different after the code snippet (incorrect execution).

Problem 4 (5.12):

a) 286
Explanation: Initially all 16 processors will have to get a shared copy of the line to test it (16 bus transactions). One will obtain the lock and invalidate all other copies (1 bus transaction). Then each of the remaining 15 processors will have to read the line again to test for the release of the lock (15 bus transactions). The processor that possesses the lock will release, causing an invalidate (1 bus transaction). This process will repeat until all processors have acquired the lock. The pattern looks like this:

\[ 16 + 1 + 15 + 1 + 15 + 1 + 14 + 1 + 14 + 1 + \ldots \]

There will be no invalidates when the last processor has the lock because it will have an exclusive copy.

b) 1650 cycles until the first processor releases the lock (including bus transaction to invalidate all copies of the line when it releases the lock).
16 + 1 + 15 + 1 = 33 bus transactions

14300 cycles until the 16th processor acquires the lock (including the bus transaction when the 16th processor acquires the line). There is no invalidation bus transaction for the last processor to release the lock because it has an exclusive copy of the line.
**Problem 5 (6.5):**

**a)** The sequence LW0, LW4, LW0, LW8, LW0, LW12, LW0, LW16 would cause these two caches to violate inclusion. The notation above represents loads of the numbered 8-byte cache lines. Since the L1 cache contains 2 sets and the L2 cache contains 4 sets, the referenced lines will map to the same set in both the L1 and L2. Here are the final contents of both caches. The violation of inclusions occurs because the reference to LW0 hits every time in the L1, causing it to never be replaced. It is never referenced again in the L2, causing it to be the LRU line.

<table>
<thead>
<tr>
<th>L1</th>
<th>0</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2</th>
<th>16</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**b)** The sequence LW0, LW4, LW1, LW8 would cause these two caches to violate inclusion. Now, since the block sizes are different in the two levels of cache, the same lines don't always map to the same set in each cache. This makes the violation of inclusion much easier. So, when LW0 is executed, both lines 0 and 1 are loaded into the same block in the L2, while only line 0 is loaded into the L1. Lines 0 and 1 map to different sets in the L1, just as they would in part A. When LW1 is executed, it hits in the L2, causing the L1 to be updated and the block containing the lines 4 and 5 to be the LRU in the L2. The execution of LW8 then replaces the block containing lines 4 and 5 in the L2, but line 4 remains in the L1.

<table>
<thead>
<tr>
<th>L1</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2</th>
<th>0</th>
<th>1</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 6 (6.6):

a) Does not naturally provide inclusion. If the L1 is direct mapped, the block size of the L1 and L2 are the same, and the L1 and L2 share the same line size then inclusion is maintained naturally. In this case the L1 is direct mapped, both caches have 32-byte lines, and each L1 has 256 sets while the L2 has 32768 sets. However, because the L1 is a split cache, inclusion is not maintained naturally. It is possible for a line from each of the L1 caches to map to the same place in the L2. If replacement occurs, the L1 and L2 copies of either the instruction of data will be replaced, but the other L1 will still have the line. Inclusion can be violated in this way.

b) Does not naturally provide inclusion. If LRU replacement is used, a situation similar to that of problem 5, part b could occur, where because the L2’s lines are twice as large as the lines in the L1, references to both halves of the line will affect its replacement policy and possibly cause different lines to be replaced in the L1 and L2 in the event of a cold miss.

Problem 7 (6.19):

The state diagram is drawn on the following page. For clarity in my drawing, I omitted the intermediate states that are the result of bus arbitration. These omitted transient states are states that would exist in any level of the cache hierarchy. An example of this is the transient state that exists on an I -&gt; M transition that is only caused by the bus delay to do the read and invalidates. Also, I assumed inclusion and a write-back cache structure.

Because the cache hierarchy is write-back, it is possible for the L1 to have the line in a modified state and not yet updated that state in the L2. The L2 has to then have a new state for this case, I call it modified stale (MS). When the line is to be written for the first time, the line enters the MS state in the L2 and the M state in the L1. The L2 can only get to the M state if the line is evicted from the L1 and therefore written back to the L2. The L2 then has the most recent copy of the data and is responsible for it until the next write to that line when the L1 obtains the M copy and the L2 goes back to the MS state.

The transient state that I added accounts for the decision of whether to transition from the I state to either the E state or to the S state. This decision requires a bus transaction to determine if the line being loaded is shared or not.