Chapter 5 sample problems.

1. Use figure 5.37 for this problem. Given the entries as shown in the figure, show the result of each of the following read/write operations. Use a notation like shown in problem 5.1. For instance, if core C0 writes 0010 to AC08, the result would be: C0.1 (M, AC08, 0010), core 0's line 1 will change state to modified and the address, AC08, will receive the datum 0010. If a line must be discarded and has been modified, then include a write to memory which would appear as M: address ← value. Each of the operations below, a-e, are independent of each other. That is, each operation should occur on the state of the caches and memory as shown in figure 5.37.
   a. C0 reads AC18
   b. C0 writes 0016 to AC10
   c. C0 reads AC28
   d. C0 reads AC08
   e. C0 reads AC30

   Answer:
   a. AC18 is currently invalid in core 0's cache, so it must obtain the datum memory, which sends back 0018. Core 0 updates its cache: C0.3 (S, AC18, 0018)
   b. AC10 is currently modified in core 0, so a new update causes no state change, and since it was modified at some time of the past, any other core that has cached AC10 has noted it as invalid, so no invalidation message is required. C0.2 (M, AC10, 0016)
   c. AC28 is not present in core 0's cache, so it must be retrieved from memory. But memory has an invalid (stale) copy because core 1 has modified it. In core 1, AC28 is indicated as M (modified). Therefore, core 1 sends its copy to core 0 while also updating memory and changing its state from M to S (shared)

      C0.1 (S, AC28, 0068)
      C1.1 (S, AC28, 0068)
      M: AC28 ← 0068
   d. As AC08 is already in core 0's cache and shared, all that happens is 0008 is returned from the cache to the core. No changes are made to its cache.
   e. AC30 is not in core 0's cache, nor in any other cache. So memory must return it to core 0. In core 0, AC30 will be placed in line 2. But line 2 has been modified, so before storing AC30, core 0's cache must return AC10 to memory. Upon receiving AC30, it will be marked as shared because the datum has not been modified and is valid.

      M: AC10 ← 0030
      C0.2 (S, AC30, 0030)

2. For each row in figure 5.5, explain what that combination of request/source/state is and provide a brief example.

   Answer:

   Read hit from processor, shared or modified, normal hit: processor is successful at reading a datum in its own cache and since that datum is valid, no other action is required.

   Read miss from processor because datum is invalid: the datum is present in the processor's cache but invalid and so must be retrieved from whoever has the version cached. The processor sends a request on the shared bus and the cache which has the valid (modified) version responds. This processor updates its cache to have a shared copy, and the other processor modifies its cache to indicate the value is now shared rather than modified. If these are a write-back caches, the datum is also sent to memory (if write-through, the datum would already have been returned to memory).

Read miss from processor, shared: this datum had been in the cache at some point but replaced, therefore the processor sends a request onto the bus and memory responds.

Read miss, processor, modified: this one is stated a little inaccurately. There is a read miss and the item being brought in conflicts with a modified item. In this case, the item must be sent back to memory first, and then the new item brought it. The new item is listed as shared since it has not been modified.

Write hit, processor, modified: the datum is found in local cache and modified. Since it was previously modified, no other action is needed (no invalidation message is required because the last modification would have resulted in that). However, if this is a write-through cache, then the datum is sent to memory at this time.

Write hit, processor, shared: the datum is found in cache and modified, the state of the cache is updated to modified and an invalidation message is sent out on the bus so that any other cached copies are marked as invalid.

Write miss, processor, invalid: datum is not in the cache, so the datum is cached and marked as modified, and sent out on to the bus for other caches to be marked as invalid and memory to update its value.

Write miss, processor, shared: the datum is not found in the cache but in doing a write, the current shared item in the cache must be discarded, the new datum is stored in the cache and marked as modified, and an invalidate message is sent out across the bus to all other caches.

Write miss, processor, modified: the datum is not found in the cache but in doing a write, the current modified item in the cache must be discarded, before being discarded though, since it was modified, it must be sent to memory, then the modified item is stored in cache as modified and an invalidate message is sent out across the bus.

Read miss, bus, shared: a read miss has been received from another processor, the item is stored locally as shared and returned to the requester.

Read miss, bus, modified: a read miss has been received from another processor, the item is stored locally as modified, place datum (entire block) on bus to be sent to any requesting cache and memory, modify local state to shared.

Invalidate, bus, shared: a locally cached item that is currently marked as shared has been modified by another cache, mark the local copy as invalid.

Write miss, bus, shared: a locally cached item that is currently marked as shared has been modified by another cache, mark the local copy as invalid.

Write miss, bus, modified: a locally cached item that is currently marked as modified has been modified by another cache, send modified copy to memory and then mark local copy as invalid.

3. MESI adds exclusive on top of modified. MOESI adds owner on top of exclusive. How does exclusive differ from shared? Provide an example to illustrate how the two differ. How does owner differ from exclusive? Provide an example to illustrate how the two differ.

   Answer: The E state is used when a copy is known to exist only in one cache. When that cache has exclusive ownership, it can modify the datum without having to send an invalidate signal as no other cache can have a copy. If another cache requests a copy, this cache must update its state from E to S (shared). The O state is used when a datum is cached in one cache and shared by other caches such that if the cache with the O state modifies the datum, it must share the new datum with the other caches but does not have to write the datum back to memory. In effect, the cache is taking ownership of the datum and must then respond to all other caches rather than allowing memory to respond.

4. All processors need to have snoopy caches at the same level. For instance, if L2 is shared, the L1 caches would be the snoopy caches, and if L3 is shared, it would be the L2 caches that are snoopy caches. Which level would be best for the snoopy caches, L1, L2, or L3? Explain.

   Answer: I am going to make the following assumption: each processor has its own L1 cache on the chip and the L2 cache is on the chip while the L3 cache is not on the chip. If this is the case, we do not want the L3 to serve as a snoopy cache. We usually will have a single L3 cache anyway, and the idea behind snoopy caches is that they collectively are at the interface between the processors and the shared memory. So L3 is out. This leaves L1 or L2. If we have a single, shared L2, then the L1 caches would be our snoopy caches. I would argue for a multi-core processor that this makes the most sense. But in a true MIMD computer (multiple processors), each processor would have its own L1 and L2 caches. This would then require the l2 caches be the snoopy caches. A good reason to move the snoopy caches to the L2 level is that the L1 caches are always busy responding to processor requests. We don’t want to burden them further by having them snoop on a shared bus for messages coming from other caches. This would take access time away from processor requests.

5. Define the following terms.
   a. cache coherence
   b. coherence misses
   c. true sharing misses
   d. false sharing

   Answer:
   a. Requiring that a processor receive the most up-to-date datum no matter where that datum is currently cached in the system. That is, if a processor’s local cache is storing a datum that has been modified elsewhere, then that entry in the local cache must be marked as invalid so that the processor, when it attempts to access that datum, must look for the modified version.
   b. A miss that arises because a locally cached block is now invalid.
   c. Building on the definition of part b, the invalidation occurred because the datum had been shared and another processor had modified the datum.
   d. Here, the datum being referenced has itself not been modified but because the entire block was denoted as invalid, the processor believes the datum is invalid. This will be a miss if it arises, but it can be avoided if, instead of having an invalid (dirty) bit for an entire block, there are individual invalid bits for every datum in the block.

6. When it comes to cache coherence, we only have to worry about data, not instructions. Why? Which has the bigger impact on performance, instruction cache misses, true sharing misses or false sharing misses? Use figure 5.11-5.13 in section 5.3 to answer this question. When testing an OS workload on a multiprocessor computer, kernel miss rate is far higher than user miss rate (see figure 5.15) in spite of user code being executed far more often than kernel code (according to the text, the amount of user code instructions executed is 8 times that of kernel code). Why would this be the case?

   Answer: Instructions are never modified and therefore any cached instruction can be shared among all caches without concern that it will be modified elsewhere and thus invalidated. False sharing misses are always in the minority no matter the size of cache, processor size or block count. True sharing misses are roughly constant no matter the cache size, decrease slightly as block sizes increase and increase greatly as processor count increases. Instruction misses are roughly the same for processor count and block size but decrease greatly as cache size increases. So the answer to this question is really based on cache size and processor count. So let's make the assumption that we will have at least 4 processors (since quad core processors are so common) and that we will have a cache of at least 4 MB, which is common for L3 caches today. With these assumptions, true sharing misses have a greater impact than instruction misses. User processes are more predictable than OS code. User code often consists of loops and is written in small chunks. Further, a lot of user processes are smaller in size than kernel code and user processes tend to run "localized" code whereas any part of the OS might run at any time so keeping relevant items in the cache is far easier for user code.