Chapter 11

1. Using the syntax in Figure 11-2, show how to use the load-linked/store conditional primitives to synthesize a compare-and-swap operation.

```
/* r1 contains compare value, r2 contains swap value */
cmpswap: ll r0, A
         cmp r0,r1
         bne fail
         stc r2, A
         bfail cmpswap
fail:    ...
```

2. Using the syntax in Figure 11-2, show how to use the load-linked/store conditional primitives to acquire a lock variable before entering a critical section.

```
/* lock is at A, r1 contains '1' */
spin: ll r0, A
      cmpi r0,0
      bne spin
      stc r1, A
      bfail spin
```

3. A processor such as the PowerPC G3, widely deployed in Apple Macintosh systems, is primarily intended for use in uniprocessor systems, and hence has a very simple MEI cache coherence protocol. Identify and discuss one reason why even a uniprocessor design should support cache coherence. Is the MEI protocol of the G3 adequate for this purpose? Why or why not?

   High-performance I/O subsystems use DMA to access memory. Without hardware cache coherence, the OS must manually flush all I/O buffers to/from caches before/after I/O operations where those buffers are read or written by DMA agents. Simple hardware MEI coherence is adequate to handle this case (there is no need for a shared state), since the reads and writes issued by the DMA agent are snooped by the processor's MEI protocol handler, and dirty blocks are flushed on DMA reads and writes while shared blocks are invalidated on DMA writes

4. Apple marketed a G3-based dual processor system that was mostly used for running asymmetric workloads. In other words, the second processor was only used to execute parts of specific applications, such as Adobe Photoshop, rather than being used in a symmetric manner by the operating system to execute any ready thread or process. Assuming a multiprocessor-capable operating system (which the MacOS, at the time, was not), explain why symmetric use of a G3-based dual processor system might result in very poor performance. Propose a software solution implemented by the operating system that would mitigate this problem, and explain why it would help.

The G3 only supports MEI cache coherence. Without a shared state, read-only blocks can only exist in a single processor's cache at the same time. This would have a dramatic effect on instruction cache performance, since instructions are generally read-only, and in a time-shared system, multiple processors often share instruction working set. A software solution to this problem would be to replicate program text in different places of the address space for each processor. This would use more physical memory, and would complicate process migration (since every time a process migrated, it's virtual->real address mappings for program text would have to change).

5. Given the MESI protocol described in Figure , create a similar specification (state table and diagram) for the much simpler MEI protocol. Comment on how much easier it would be to implement this protocol.

MEI cache coherence protocol for Problem 5.

	Event and Local Coherence Controller Responses and Actions (s' refers to next state)			
Current State s	**Local Read (LR)**	**Local Write (LW)**	**Local Eviction (EV)**	**Bus Fetch (BF)**
Invalid (I)	Issue bus fetch s' = E	Issue bus fetch s' = M	s' = I	Do nothing
Exclusive (E)	Do nothing	s' = M	s' = I	s' = I

Modified (M)	Do nothing	Do nothing	Write data back;
			s' = I
				Respond dirty;
				Write data back;
				s' = I

Lack of a shared state eliminates quite a bit of complexity, particularly since bus reads no longer need to observe the shared line to determine whether the block should go into E or S (it always goes into E). Bus reads become equivalent to bus writes, in effect. Also, no upgrades are needed.

6. Many modern systems use a MOESI cache coherence protocol, where the semantics of the additional O state are that the line is shared-dirty: i.e., multiple copies may exist, but the other copies are in S state, and the cache that has the line in O state is responsible for writing the line back if it is evicted. Modify the table and state diagram shown in Figure to include the O state.

MOESI cache coherence protocol for Problem 6.

Current State s	Event and Local Coherence Controller Responses and Actions (s' refers to next state)					
	Local Read (LR)	**Local Write (LW)**	**Local Eviction (EV)**	**Bus Read (BR)**	**Bus Write (BW)**	**Bus Upgrade (BU)**
Invalid (I)	Issue bus read if no sharers then s' = E else s' = S	Issue bus write s' = M	s' = I	Do nothing	Do nothing	Do nothing
Shared (S)	Do nothing	Issue bus upgrade s' = M	s' = I	Respond shared	s' = I	s' = I
Exclusive (E)	Do nothing	s' = M	s' = I	Respond shared s' = S	s' = I	Error
Owned (O)	Do nothing	Issue bus upgrade s' = M	Write data back; s' = I	Respond dirty/ shared; Supply data to requestor;	Respond dirty; Supply data to requestor; s' = I	s' = I
Modified (M)	Do nothing	Do nothing	Write data back; s' = I	Respond dirty/ shared; Supply data to requestor; s' = O	Respond dirty; Supply data to requestor; s' = I	Error

7. Explain what benefit accrues from the addition of O state to the MESI protocol.

There are two benefits: migratory blocks (blocks written by multiple processors over time) need not be written back at every transfer, reducing memory bandwidth and potentially request latency. Also, “shared interventions” now become easy to implement, since the O state owner of a block can easily supply requested data (without O, multiple caches could have the block in S state, and separate arbitration is needed to decide who will supply the data; or, oftentimes, the memory controller will supply it. This can have longer latency and higher power consumption).

8. Real coherence controllers include numerous transient states in addition to the ones shown in Figure to support split-transaction buses. For example, when a processor issues a bus read for an invalid line (I), the line is placed in a IS transient state until the processor has received a valid data response that then causes the line to transition into shared state (S). Given a split-transaction bus that separates each bus command (bus read, bus write, and bus upgrade) into a request and response, augment the state table and state transition diagram of Figure to incorporate all necessary transient states and bus responses. For simplicity, assume that any bus command for a line in a transient state gets a negative acknowledge (NAK) response that forces it to be retried after some delay.

   Note: for writebacks, we assume that once the data shows up on the bus as a BD command, the processor issuing the writeback also sees the BD and can then transition to I or S. Similarly, we assume that any subsequent bus read/write will then be satisfied by memory (this is sometimes called the writeback race).

   MESI cache coherence protocol for Problem 8.

	Event and Local Coherence Controller Responses and Actions (s' refers to next state)						
Current State s	**Local Read (LR)**	**Local Write (LW)**	**Local Eviction (EV)**	**Bus Read (BR)**	**Bus Write (BW)**	**Bus Upgrade (BU)**	**Bus Data (BD)**
Invalid (I)	Issue bus read if no sharers then s' = IE else s' = IS	Issue bus write s' = IM	s' = I	Do nothing	Do nothing	Do nothing	Error
ItoS (IS)	Stall	Stall	Stall	NAK	NAK	NAK	s’ = S
ItoE (IE)	Stall	Stall	Stall	NAK	NAK	NAK	s’ = E
ItoM (IM)	Stall	Stall	Stall	NAK	NAK	NAK	s’ = M
Shared (S)	Do nothing	Issue bus upgrade s' = M	s' = I	Respond shared	s' = I	s' = I	Error
Exclusive (E)	Do nothing	s' = M	s' = I	Respond shared s' = S	s' = I	Error	Error

Modified (M)	Do nothing	Do nothing	Write data back;
			s' = I
				Respond dirty;
				Write data back;
				s' = MS
					Respond dirty;
					Write data back;
					s' = MI
						Error	Error

MtoI (MI)	Do nothing	Stall	Stall	NAK	NAK	NAK	s’ = I
MtoS (MS)	Do nothing	Stall	Stall	NAK	NAK	NAK	s’ = S

9. Given Problem 8, further augment Figure to eliminate at least three NAK responses by adding necessary additional transient states. Comment on the complexity of the resulting coherence protocol.

   MESI cache coherence protocol for Problem 9.

Current State s	Event and Local Coherence Controller Responses and Actions (s' refers to next state)						
	Local Read (LR)	**Local Write (LW)**	**Local Eviction (EV)**	**Bus Read (BR)**	**Bus Write (BW)**	**Bus Upgrade (BU)**	**Bus Data (BD)**
Invalid (I)	Issue bus read if no sharers then s' = IE else s' = IS	Issue bus write s' = IM	s' = I	Do nothing	Do nothing	Do nothing	Error
ItoS (IS)	Stall	Stall	Stall	Respond shared	s’ = ISI	s’ = ISI	s’ = S
ItoE (IE)	Stall	Stall	Stall	Respond shared s’ = IS	s’ = ISI	s’ = ISI	s’ = E
ItoStoI (ISI)	Stall	Stall	Stall	Do nothing	Do nothing	Do nothing	Satisfy LR s’ = I
ItoM (IM)	Stall	Stall	Stall	NAK	NAK	NAK	s’ = M
Shared (S)	Do nothing	Issue bus upgrade s' = M	s' = I	Respond shared	s' = I	s' = I	Error
Exclusive (E)	Do nothing	s' = M	s' = I	Respond shared s' = S	s' = I	Error	Error

Modified (M)	Do nothing	Do nothing	Write data back;
			s' = I
				Respond dirty;
				Write data back;
				s' = MS
					Respond dirty;
					Write data back;
					s' = MI
						Error	Error

MtoI (MI)	Do nothing	Stall	Stall	NAK	NAK	NAK	s’ = I
MtoS (MS)	Do nothing	Stall	Stall	NAK	NAK	NAK	s’ = S

Note: this solution assumes a true shared bus, so other processors will simultaneously see all bus messages, including Bus Data (BD).We could also try to eliminate the NAKs for IM/MI/MS with similar addition of transient states like ISI, but those are not shown below. The ISI state is tricky, since it must satisfy a single local read before reverting to I. Similarly, states like IMI or IMS would have to satisfy a single local write before either writing back (IMI) or supplying data to the next requestor (IMS).

10. Assuming a processor frequency of 1 GHz, a target CPI of 2, a per-instruction level-2 cache miss rate of 1% per instruction, a snoop-based cache coherent system with 32 processors, and 8-byte address messages (including command and snoop addresses), compute the inbound and outbound snoop bandwidth required at each processor node.

    Outbound snoop rate = .01 miss/inst x 1 inst/2 cyc x 1 cyc/ns x 8 bytes/miss = .04b/ns = 40 million bytes per second

    Inbound snoop rate = 31 x 40 = 1240 million bytes per second = 1182 MB/sec.

11. Given the assumptions of Problem 10, assume you are planning an enhanced system with 64 processors. The current level-two cache design has a single-ported tag array with a lookup latency of 3 ns. Will the 64-processor system have adequate snoop bandwidth? If not, describe an alternative design that will.

    A single port with latency 3ns provides 333 million lookups per second peak bandwidth. The rate in Problem 10 is 1280 million bytes per second @ 8 bytes per request, resulting in 160 million lookups per second. Doubling this for a 64-processor system results in 320 million lookups per second. Hence, on average, there is adequate snoop bandwidth. However, this does not account for the number of local lookups, which depends on the (unspecified) L1 miss rate. Further, since misses can be clustered and peak bandwidth demand can be much higher, providing higher snoop bandwidth would be a good idea. One alternative is to provide multiple ports into the tag array, either through true multiporting or address interleaving (banking) the tag array.

12. Using the equation in Section 11.3.5, compute the average memory latency for a three-level hierarchy where hits in the level-one cache take one cycle, hits in the level-two cache take 12 cycles, hits in the level-three cache take 50 cycles, and misses to memory take 250 cycles. Assume a level one miss rate of 5% misses per program reference, a level two miss rate of 2% per program reference, a level three miss rate of 0.5% per program reference.

$$lat_{avg} = \sum_{i=1}^{n} ref_i \times lat_i$$

    Hence, avg. lat = .95 x 1 + (.05-.02) x 12 + (.02-.005) x 50 + .005 x 250 = 3.31 cyc/ref

13. Given the assumptions of Problem 12, compute the average memory latency for a system with no level three cache, and only 200 cycle latency to memory (since the L3 lookup is no longer performed before initiating the fetch from memory). Which system performs better? What is the breakeven miss rate per program reference for the two systems (i.e. the L3 miss rate at which both systems provide the same performance)?

avg. lat = .95 * 1 + (.05-.02) * 12 + .02 * 200 = 5.31 cyc/ref; clearly the L3 performs better.

To break even, solve for: .02 * 200 = (.02 - y) * 50 + y * 250 => y = 3/200 = .015

Hence, even if the L3 tripled its miss rate from .5% to 1.5%, the L3-based system would be just as good. If the L3 miss rate exceeds 1.5%, the L2-based system will be better. Note that a 1.5% L3 miss rate is a local L3 miss rate of 1.5%/2.0% or 75%!!! I.e. a very ineffective L3 that misses 75% of the time it is accessed is still just as good as no L3 with a faster memory.

14. Assume a processor similar to the Hewlett-Packard PA-8500, with only a single level of data cache. Assume the cache is virtually-indexed but physically tagged, is 4-way associative with 128B lines, and is 512 KB in size. In order to snoop coherence messages from the bus, a reverse-address translation table (RAT) is used to store physical-to-virtual address mappings stored in the cache. Assuming a fully-associative RAT and 4KB pages, how many entries must it contain so that it can map the entire data cache?

    There are 4K blocks in the L2 (512KB/128B = 4K); each could have a unique physical page number in the tag. Hence, a total of 4K entries are needed in the RAT to guarantee that all cache blocks can be mapped. Of course, a 4K-entry fully-associative RAT is quite expensive and possibly too slow since it needs to be accessed for each snoop.

15. Given the assumptions of Problem 14, describe a reasonable set-associative organization for the RAT that is still able to map the entire data cache.

    There are 12 untranslated bits in each address, and only 7 of these are used for block offset. Hence, 5 bits are left to use as part of the index into the L2. Hence, the RAT could also use these 5 bits to index into the RAT, giving 2^5 = 32 sets. Hence, you would have 4k/32 = 128-way set associative RAT, which is much more feasible.

16. Given the assumptions of Problem 14, explain the implications of a RAT that is not able to map all possible entries in the data cache. Describe the sequence of events that must occur whenever a RAT entry is displaced due to replacement.

    The problem is due to inclusion (the RAT must be inclusive of all of the L2)--whenever a RAT entry is displaced, all of the L2 blocks that it pointed to must also be evicted. Hence, the L2 must be searched for all matching physical tags, and those blocks need to be evicted or written back (if dirty).

Problem 17 through Problem 19

In a two-level cache hierarchy, it is often convenient to maintain inclusion between the primary cache and the secondary cache. A common mechanism for tracking inclusion is for the L2 cache to maintain presence bits for each L2 directory entry that indicate the line is also present in the L1 cache. Given the following assumptions, answer the following questions:

- Presence bit mechanism for maintaining inclusion
- 4K virtual memory page size
- Physically indexed, physically tagged 2MB 8-way set associative cache with 64B lines

17. Given a 32KB 8-way set associative L1 data cache with 32-byte lines, outline the steps that the L2 controller must follow whenever it removes a cache line from the L2 cache. Be specific, explain each step, and make sure the L2 controller has the information it needs to complete each step.

    Since the L1 only needs 12bits for index+offset (5 bits of offset, 7 bits to pick one of 128 sets), it can be indexed with untranslated bits and tagged with physical addresses. Hence, whenever the L2 evicts a block, it must check for up to two presence bits (since L1 lines are half the size of L2 lines), then query the L1 for each set presence bit, and invalidate any matching line. No reverse-address-translation is needed, since the L1 is physically-addressed.

18. Given a virtually-indexed physically-tagged 16KB direct-mapped L1 data cache with 32-byte lines, how does the L2 controller's job change?

    Here, 14 bits are needed to index into the L1, and two of those bits are virtual address bits. Hence, the L2 has to check 2^2 = 4 different locations in the L1 for every presence bit set in the L2 block being evicted. Hence, in the worst case the L2 has to check and evict 2 x 4 = 8 blocks from the L1 for every L2 eviction.

19. Given a virtually-indexed, virtually-tagged 16KB direct-mapped L1 data cache with 32-byte lines, are presence bits still a reasonable solution, or is there a better one? Why or why not?

    It is not very reasonable, each of the 8 entries (see solution to Prob. 18) must now have its tag read out, sent through the TLB to translate to a physical address, and then checked against the L2 tag. A much better solution is to place pointers in the L2 that point to matching entries in the L1. See [Wang, Baer, Levy 1989] for details.

20. Figure 11-8 on page 9-602 explains read-set tracking as used in high-performance implementations of sequentially consistent multiprocessors. As shown, a potential ordering violation is detected by snooping the load queue, and refetching a marked load when it attempts to commit. Explain why the processor should not refetch right away, as soon as the violation is detected, instead of waiting for the load to commit.

    The refetch is expensive, and should be delayed until it is known that the load won't replay for some other reason or get squashed due to a branch misprediction. Waiting till commit is a sufficient condition for knowing that the load must replay.

21. Given the mechanism referenced in Problem 20, false sharing (where a remote processor writes the lower half of a cache line, but the local processor reads the upper half) can cause additional pipeline refetches. Propose a hardware scheme that would eliminate such refetches. Quantify the hardware cost of such a scheme.

    One could increase the snoop granularity by adding 1 or more bits to the snoop to indicate which half, quartile, etc. of the block was written. However, the writing processor would then have to send additional snoops out of it wanted to write additional sections of the block. An alternative scheme would be to replay a load that is hit by a remote snoop, and check to see if the new value matches the original value. If the values match, no refetch is needed; if they don't a refetch is triggered. A scheme that builds on this is described in [Cain & Lipasti, ISCA 2004].

22. Given Problem 21, describe a software approach that would derive the same benefit.

    Most performance-sensitive commercial applications (e.g. relational databases) are heavily optimized to avoid any kind of false sharing. This usually means that shared data objects are padded to make sure that two objects can't coexist in the same cache line.

23. A chip multiprocessor (CMP) implementation enables interesting combinations of on-chip and off-chip coherence protocols. Discuss all combinations of the following coherence protocols and implementation approaches and their relative advantages and disadvantages. On-chip, consider update and invalidate protocols, implemented with snooping and directories. Off-chip, consider invalidate protocols, implemented with snooping and directories. Which combinations make sense? What are the tradeoffs?

    On-chip, update protocols may be feasible, since there is abundant bandwidth. However, consistency model restrictions may complicate this, since most consistency models do not allow "some" processors to see the effects of a write before "all" processors see it (this is known as write atomicity). If the update makes the write visible to on-chip processors, but not to off-chip processors, the consistency model could be violated. Invalidate protocols don't suffer from this problem. Snooping on-chip will require substantial L1 snoop bandwidth if there are more than a few on-chip cores. Hence, keeping a "mini-directory" at the L2 that keeps track of which processor's have copies of each block can be more scalable. Since the L2 is still fairly close, the latency penalty of the directory indirection won't be too severe.

    Off-chip, the tradeoffs are similar to conventional SMPs: directory protocols scale to larger numbers of nodes, but suffer from longer latencies for communication misses. Since CMPs incorporate multiple processors on chip, it is likely that scalability to a large number of nodes will not be required. Hence, a snooping protocol would likely be the right choice.

    A reasonable combination might be a directory-like protocol on chip, with a snooping protocol off-chip. This avoids multiporting of L1 tags (required for on-chip snooping) yet provides low latency for off-chip communication misses.

24. Assume that you are building a fine-grained multithreaded processor similar to the Tera MTA that masks memory latency with a large number of concurrently active threads. Assuming your processor supports 100 concurrently active threads to mask a memory latency of 100 1-ns processor cycles. Further assume that you are using conventional DRAM chips to implement your memory subsystem. Assume the DRAM chips you are using have a 30 ns command occupancy, i.e. each command (read or write) occupies the DRAM chip interface for 30 ns. Compute the minimum number of independent DRAM chip interfaces your memory controller must provide to prevent your processor from stalling by turning around a DRAM request for every processor cycle.

    The memory controller must be able to satisfy one request per ns. If each DRAM interface requires 30ns per request, you would need at least 30 DRAM interfaces to sustain that bandwidth, given a perfect interleaving of requests. Ouch.

25. Assume what is described in Problem 24. Further, assume your DRAM chips support page mode, where sequential accesses of 8 B each can be made in only 10 ns. That is, the first access requires 30 ns, but subsequent accesses to the same 512 B page can be satisfied in 10 ns. The scientific workloads your processor executes tend to perform unit stride accesses to large arrays. Given this memory reference behavior, how many independent DRAM chip interfaces do you need now to prevent your processor from stalling?

    Assuming each access is 8B, one needs 30ns for the first access to a page and then 10ns for each of the next (512/8 - 1 =) 63 accesses. Hence, the average access time is 660/64 = 10.3ns. It follows that 11 DRAM interfaces should be sufficient to provide adequate bandwidth for unit-stride access streams.

26. Existing coarse-grained multithreaded processors such as the IBM Northstar and Pulsar processors only provide in-order execution in the core. Explain why or why not coarse-grained multithreading would be effective with a processor that supports out-of-order execution.

    Coarse-grained multithreading is not a good match for OOO processors, since it would require draining the OOO core of one thread on a cache miss and refilling it with instructions from the other thread. In modern deeply-pipelined processors, draining and refilling the core can take many cycles; this cost would eat into and perhaps subsume the benefit obtained from coarse-grained multithreading.

27. Existing simultaneous multithreaded processors such as the Intel Pentium 4 also support out-of-order execution of instructions. Explain why or why not simultaneous multithreading would be effective with an in-order processor.

    SMT could be made to work with in-order processors, but the dispatch logic would be quite complicated, since it would need to merge instructions from multiple fetch streams. Also, the processor would likely end up behaving very much like a coarse-grained mt processor, since any stall condition (e.g. cache miss) would stall one thread and let the other thread take over the entire dispatch width. Hence, the only benefit of SMT over CGMT would be in cases where both threads are able to run. This is a fairly marginal benefit given the complexity added to the dispatch and fetch stages.

28. An IMT design with distributed processing elements (e.g. Multiscalar or TLS) must perform some type of load balancing to ensure that each processing element is doing roughly the same amount of work. Discuss hardware- and software-based load balancing schemes and comment on which might be most appropriate for both Multiscalar and TLS.

    Essay solution not provided.

29. An implicit multithreaded processor such as the proposed DMT design must insert instructions into the reorder buffer out of program order. This implies a complex free-list management scheme for tracking the available entries in the reorder buffer. The physical register file that is used in existing out-of-order processors also requires a similar free-list management scheme. Comment on how DMT ROB management differs, if at all, from free-list management for the physical register file. Describe such a scheme in detail, using a diagram and pseudocode that implements the management algorithm.

Detailed solution not provided. The main differences are that searching the ROB to find instructions to commit becomes quite complicated, since the commit logic can no longer simply examine adjacent entries. One solution is to use a "compacting" ROB that keeps the oldest instructions at the physical "end" of the ROB. This type of scheme is described in [Bell & Lipasti, Deconstructing Commit, ISPASS 2004].

30. The DEE proposal appears to rely on fairly uniform branch prediction rates for its limited eager execution to be effective. Describe what happens if branch mispredictions are clustered in a non-uniform distribution (i.e., a mispredicted branch is likely to be followed by one or more other mispredictions). What happens to the effectiveness of this approach? Use an example to show whether or not DEE will still be effective.

    The effectiveness of DEE degrades in this case, since clusters of mispredictions imply that there are also long sequences of correctly predicted branches. Whenever the latter is true, the DEE approach is wastefully executing alternate paths. Also, in a cluster of mispredicted branches, even DEE is unlikely to choose the correct path as one of its alternate paths.

31. A recent study shows that the TLS architecture benefits significantly from silent stores. Silent stores are store instructions that write a value to memory that is already stored at that location. Create a detailed sample execution that shows how detecting and eliminating a silent store can substantially improve performance in a TLS system.

    Detailed solution not provided. Assume a future thread that issues a speculative load to a location that is later written by an earlier thread. In a conventional TLS scheme, the future thread would get squashed and replayed. However, if the write is silent, and a mechanism exists for detecting that silence, no squash/replay is necessary.

32. Pre-execution of conditional branches in a redundant runahead thread allows speedy resolution of mispredicted branches, as long as branch instances from both threads are properly synchronized. Propose a detailed design that will keep the runahead thread and the main thread synchronized for this purpose. Identify the design challenges and quantify the cost of such a hardware unit.

    Detailed solution not provided. One option to pursue is a branch-history queue between the runahead thread and the trailing thread. This has to contain a bit for each conditional branch, as shifted in by the runahead thread. Of course, this does not handle computed branches. To handle those, there could be a fetch address queue connecting the two threads. Of course, whenever the trailing thread detects a problem, this queue has to be flushed and the runahead thread has to restart.