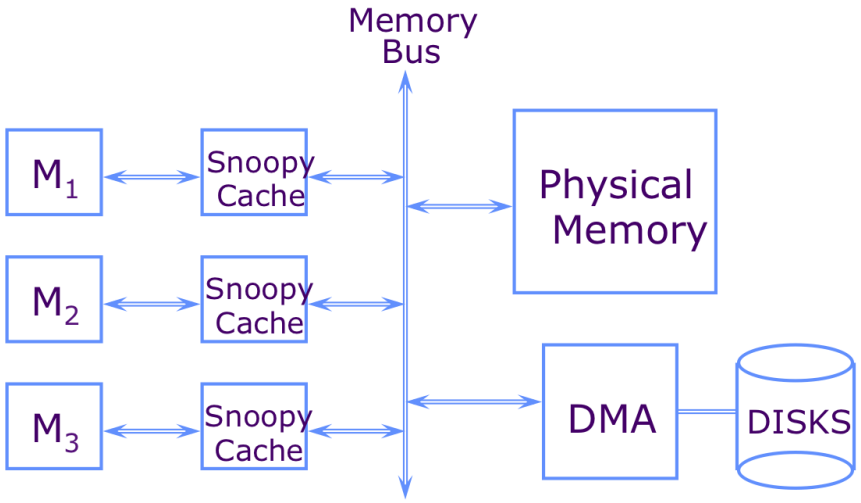
Name _____

Question 2: Snoopy Cache Coherence [32 points]

In class we discussed MSI and MESI cache coherence protocols on a bus-based processor. We will assume 3 cores in a processor. Each core has one snoopy write-back cache and is connected to the bus. There is also a memory controller and a DMA engine connected to an array of hard disk drives. The bus has a latency of one cycle.



For this question, we will consider an extended version of MESI, the simplified MOESI protocol. The MOESI protocol has five states for each cache line as follows (states are explained from the viewpoint of a single cache):

- Invalid (I): Cache line is not valid (does not exist).
- Shared (S): A shared cache line may exist in other caches. A line in shared state can be read, but not written to without causing a transition to a new state. The value in the cache is the same as the value in main memory.
- Exclusive (E): The value is the same as the value in main memory. This line is not cached in any other cache (other caches have it as invalid).
- Modified (M): The value in the cache line has been modified. Therefore, it is different than the value in main memory. This line is not in any other cache.
- Owned (O): Cache line is dirty (has been modified). Thus, it is different than the value in main memory. In this state, the cache line exists in other caches. Only one cache can have the line in the owned state, and that contains the most up to date data. Other caches may have the line in the shared state. If the cache line in the owned state is updated (written), the cache must *broadcast* the new value to all other caches.

As a default, if a line is written, the MOESI protocol jumps to the modified (M) state if that line does not exist in other caches. Otherwise, it jumps to the owned (O) state.

The state transition diagram for MESI is provided in the appendix.

Name _____

Q2.A Transitions to the Owned State [7 points]

To make use of the new state, we need to define transitions to and from it. For each of the possible transitions, define if they are allowed. If they are, explain what is the trigger (i.e., a local cache update, remote read, etc), the condition (e.g., other sharers exist), and if there is an action (e.g., writeback). Do not answer for (O) -> (M) in this question.

(I) -> (O): Trigger: local cache writes the cache line. Condition: Other sharers exist. Action: Broadcast up to date value

(O) -> (I): Not allowed. If another sharer writes, it will go to owned and the local cache would transition to shared.

(S) -> (O): Trigger: local cache writes. Condition: Other sharers exist (which is most likely). Action: broadcast up to date value.

(O) -> (S): Trigger: another cache writes. No condition or action.

(E) -> (O): Not allowed. While in E no other cache has a copy so just transition to M instead.

(O) -> (E): Not allowed.

(M) -> (O): Trigger: Another cache reads. Action: Broadcast modified value.

(O) -> (M): Do not answer this.

Name _____

Q2.B Why MOESI [3 points]

Why would we choose MOESI over MESI? Do many loads favor MESI or MOESI? Do many stores many MESI or MOESI?

MOESI has the advantage that one cache can write a line and still allow other caches to be in the shared state. With MESI, each write would invalidate the line in all other caches that would cause future cache misses. Many loads favor MOESI. Many stores favor MESI the owned state causes a broadcast for each store.

Q2.C Owned to Modified [5 points]

Lets focus on the transition from owned (O) to modified (M) in MOESI. If a cache writes a cache line that is in the owned state (O), should the transition be taken or should the cache line remain owned? If the transition from owned (O) to modified (M) is taken, an invalidate is sent to all other caches. To answer, use the example of (a) code with a small number of writes (stores) and many reads and (b) code with a large number of writes and few reads. The same code runs on all cores. Is your answer based on correctness, or is it a matter of improving performance such as by increasing the cache's hit rate or decreasing the number of messages?

Correctness is not violated in either case. The key is that while in owned (O) state, each write to the cache triggers a broadcast to all other caches. The benefit is that other caches can be in shared. In case (a) with few writes, we prefer to stay in owned because the expensive operation (writes) is infrequent, and we have many reads so we prefer to have them stay in shared. In case (b), we prefer to transition to modified (M).

Name _____

Q2.D Sequential Consistency [3 points]

Does cache coherency guarantee sequential consistency? Explain your answer.

It does not. A system with an out-of-order cache and coherency still does not guarantee sequential consistency.

Q2.E Snoopy Bus Avoids Transients [3 points]

The finite state machines for MOESI and MESI have no transient states with the snoopy bus, but they do with directories. What is the property of the bus that lets us avoid transient states?

The bus has to provide atomicity. This means that once a message enters, it reaches all other caches and any replies to that message enter the bus and complete, before any other request can enter the bus.

Name _____

Q2.F Coherency Misses [11 points]

Consider the following code running in **two** cores. For this question we will use the original MESI protocol instead of MOESI. The state diagram for MESI can be found in the appendix.

```
(1) LW  x1, 0(x5)
(2) LW  x2, 0(x6)
(3) SW  x3, 0(x6)
(4) SW  x2, 0(x5)
(5) LW  x1, 0(x6)
```

Do not optimize or re-order the code. Assume the processor guarantees sequential consistency. Also, the addresses in x5 and x6 map to different cache lines.

Assume the following execution sequence:
A.1, A.2, B.1, B.2, A.3, B.3, B.4, A.4, A.5, B.5 (A and B are the two cores)

Fill in the table below with the states of the cache lines at every step. Also, count the number of communication events. A communication event is a message sent from the cache as part of the transition, and is part of certain transitions. For example, shared to owned causes communication to broadcast the new value. Assume invalid to exclusive does not cause communication (i.e., we assume the cache knows there is no other sharer), but invalid to shared does only if another cache has the line as exclusive. The initial state of both cache lines in both caches is invalid.

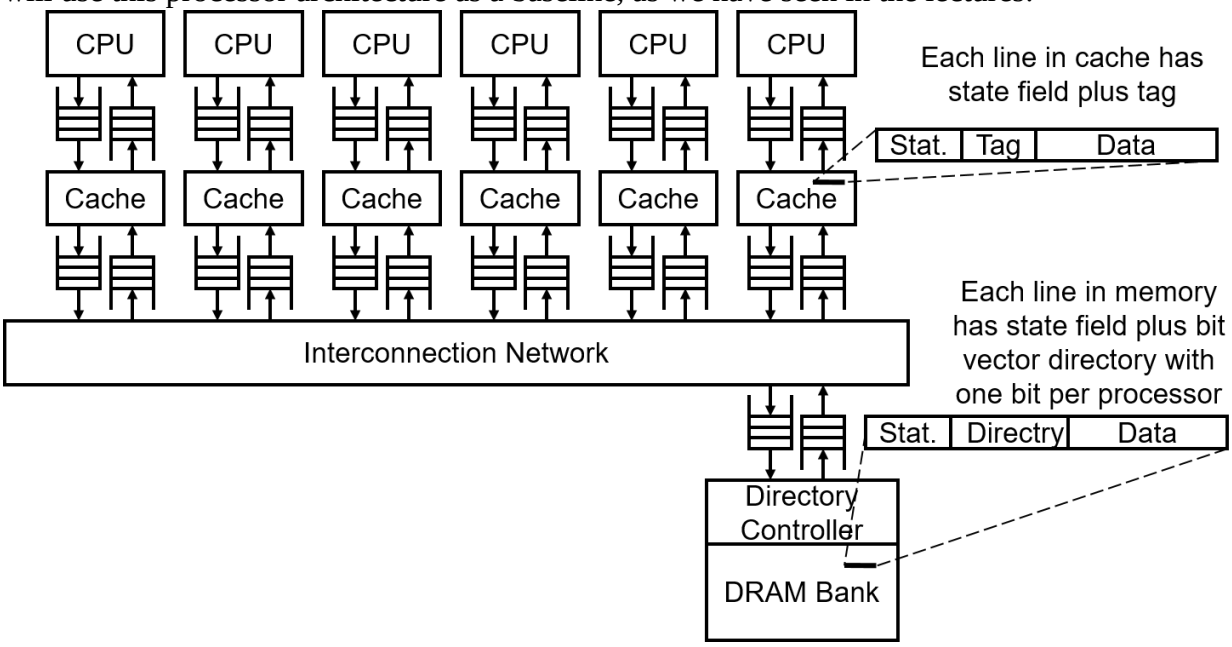
Core: Instruction	State x5 cache line in core A	State x6 cache line in core A	State x5 cache line in core B	State x6 cache line in core B
A: LW x1, 0(x5)	E	I	I	I
A: LW x2, 0(x6)	E	E	I	I
B: LW x1, 0(x5)	S	E	S	I
B: LW x2, 0(x6)	S	S	S	S
A: SW x3, 0(x6)	S	M	S	I
B: SW x3, 0(x6)	S	I	S	M
B: SW x2, 0(x5)	I	I	M	M
A: SW x2, 0(x5)	M	I	I	M
A: LW x1, 0(x6)	M	S	I	S
B: LW x1, 0(x6)	M	S	I	S

7 communication events.

Name _____

Question 3: Directory Protocols [21 points]

Even directory-based coherence is not easy to scale to large core counts. For this question, we will use this processor architecture as a baseline, as we have seen in the lectures:



For simplicity, we assume that there is only one directory. In the above figure, we show 6 cores. However, for this problem we will discuss how things change as we scale up to 2048 cores. Assume the MSI protocol where cache lines can be in 6 states including transients in the caches, and 13 states in the directory.

Q3.A Size of Entries [4 points]

For each cache line, the directory maintains state and a bit vector to record which of the cores have that cache line. For example with 6 cores, a bit vector of "100011" means that cores 1, 5, and 6 have that cache line.
If a cache line has 64 bytes of data, how many bits of the coherence protocol in the directory, and how many bits for coherency in a cache? Answer first for 6 cores and then for 2048 cores. Ignore the tag.

With 6 cores we have $\log_2(6)$ bits in every cache line for every 64 bytes of data. So $\log_2(6)$ bits. The answer is the same for 2048 cores.
For the directory the answer changes. For every 64 bytes of data, we have $\log_2(13) + 6$ bits for 6 cores, and $\log_2(13) + 2048$ bits for 2048 cores.

Name _____

Q3.B Alternatives to Bit Vectors [7 points]

From your answer above, it should be apparent that the bit vector each directory maintains in each cache line quickly becomes a problem as we scale the number of cores. One alternative is to replace the bit vector with a linked list of core IDs. In that case, if cores 2 and 10 share a cache line, the directory contains the numbers "2" and "10" for that cache line. With 2048 cores, each of those numbers needs 11 bits. Assume no other overhead for the linked list.

First determine how many core IDs we can store if we limit the space for the list to 1024 bits. Then, propose a mechanism to deal with the case where we have more cores than your answer sharing the line. For example, if you answer that we can keep up to 10 sharers, how would we deal with 12 sharers? In your answer, state how and when a cache line can revert back to "normal" operation, when the number of sharers drop.

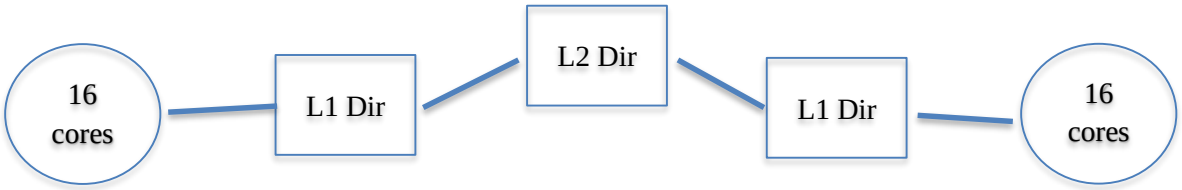
Do not worry about performance in your solution, just correctness. Also, do not add extra directories. Maintain one directory for the entire processor.

We can store 1024 / 11 rounded down = 93 core IDs. If we get more than 93 sharers, we would have to force the directory to broadcast any messages to sharers of that line to all cores, whether they share it or not. That means we no longer have to keep track of what cores share this line, but the expense is that now we have to broadcast until that line is evicted from the directory or a core modifies the line (which forces all other sharers to invalidate so we know there is only one core that has the line at that point).

Name _____

Q3.C More Directories [7 points]

Based on the possible performance losses from solutions that still keep one directory, now we would like to add a *hierarchy* of directories. Hierarchical directories work much like a hierarchy of caches. We will assume only two levels of directories. In the first level, we have one directory every 16 cores (128 directories for 2048 cores). In the second level, we still have a single directory. A core's cache sends a message to its first-level directory. If that cache line is only shared within the group of processors that this first-level directory serves, the request is satisfied and an answer is returned. Otherwise, the request is further propagated to the second-level directory, which then forwards messages to any first-level directories that serve cores which have that cache line. The second-level directory does not send messages to cores directly, and treats groups as a single domain. I.e., is a core in a group shares a line, the second-level directory does not need to know which caches(s) is that group the line is in.



What is the advantage of this design that helps scalability?
Then, at a high level, describe the sequence of events and messages if core A has a line in the shared state and wants to write to it. Core B is in the same 16-core group and has the same line in the shared state. Also, core C has it in shared, but is in a different 16-core group.

The advantage is that first-level directories only maintain sharer state (e.g., the bit vector) for 16 cores. The second-level directory only maintains a bit vector for 128 first-level directories.

In that scenario, core A sends a message to its first-level directory. That directory then sends a message to invalidate core B's copy, and also propagates the message to the second-level directory. That then sends it to the first-level directory that serves core C, which forwards it to core C. The acknowledgment from core C is sent via the same directories to core A's first-level directory which also waits for the acknowledgment from core B. Then, core A's first-level directory sends the final acknowledgment to core A which then transitions that line to the modified state.

Name _____

Q3.D More Levels [3 points]

Name one advantage and one disadvantage of increasing the levels of our directory hierarchy to three from two. Does having a hierarchy of directories affect where we want to place cores that tend to share data (e.g., producer – consumer)?

One advantage: less state required per directory. This makes directories smaller and thus can be clocked faster. Disadvantage: more messages between directories. Having groups motivates placing cores that share data in the same group to reduce communication outside the group.

Name _____

Question 4: Potpourri [16 points]

Q4.A Measure of Performance [5 points]

Suppose we write a parallel program with a critical section. We provide mutual exclusion with a store conditional, as the example below for the consumer, as we have seen in the lectures:

```
try:    Load-reserve R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head = R_head + 1
        Store-conditional (head), R_head
        if (status==fail) goto try
        process(R)
```

Is CPI (clocks per instruction) an accurate measure of performance? Can you propose a more accurate measure of performance for parallel applications?

The problem with CPI in this example (same as with locks) is that a core can be executing instructions but not be making forward progress. In the above example, trying again after failing the store conditional increases the instruction count, but is not helping the code make progress. A more accurate measurement will be completion time. That is, the time is takes for the slowest thread of the same application to complete.

Name _____

Q4.B Sequential Consistency with Out-of-Order Cores [3 points]

Can an out-of-order execute and out-of-order commit core be used in a sequential consistent system? How about an out-of-order execute but in-order commit?

No to the first and yes to the second. The key is what order loads and stores are made visible outside of the core and to the cache. Out-of-order commit cores violate sequential consistency in the order loads and stores arrive to the cache.

Q4.C Misses with Coherence [4 points]

In a uniprocessor system, we previously discussed that keeping the number of lines constant in a cache and increasing the cache line size tends to decrease the number of cache misses. In a cache coherent system executing a parallel program with locks and critical sections, is increasing the cache line size always a win?

No. Cache coherence applies in a cache line granularity. Especially with locks, small 4 or 8 byte values will frequently be accessed by different threads. If those values are different but map to the same cache line, they will cause false sharing coherence misses, caused by the coherence protocol invalidating entire lines.

Name _____

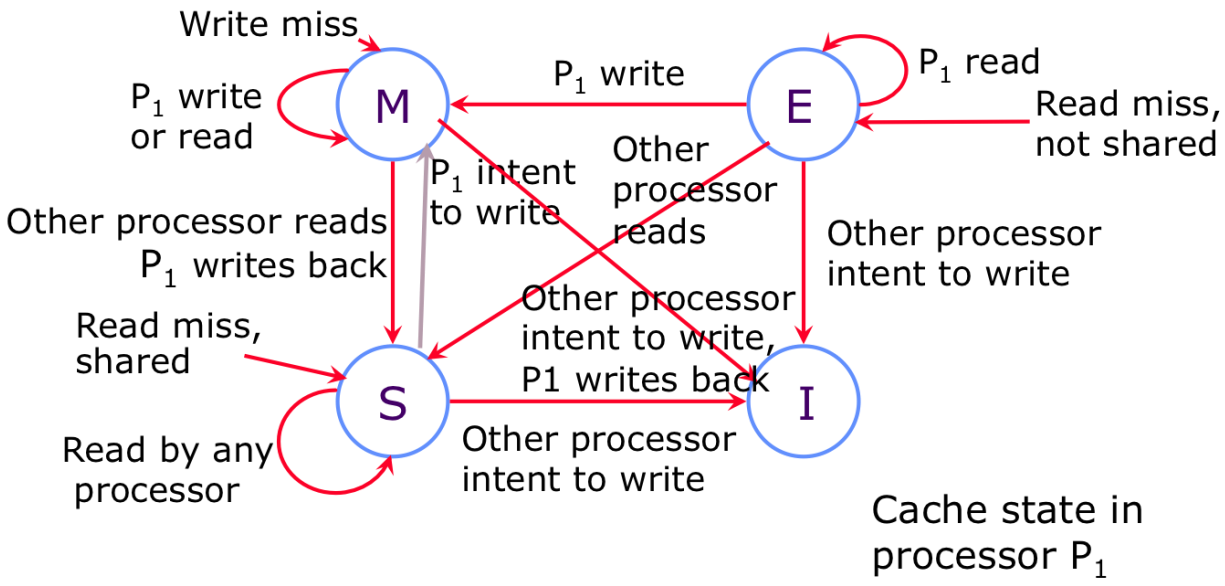
Q4.D Multiple Synchronization Primitives [4 points]

In class and in problem 1 of this quiz, we mentioned multiple synchronization primitives such as test&set, fetch&add, compare&swap, etc. Why do modern ISAs provide these many primitives even though only one is enough to guarantee correctness, such as by implementing locks?

Multiple reasons. ISAs need to provide different alternatives because the underlying hardware or architectural decisions can make one of these primitives more efficient. Also, the same is true for characteristics of program behavior such as the number of writes and contention. In addition, different primitives can be used for different needs. For example, non-blocking functions need non-blocking promitives (such as compare&swap) for example.

Appendix

The state machine for the unmodified MESI protocol:



YOU MAY DETACH THIS PAGE