I affirm that the work I am submitting is my own. I have not copied or used any portion of another student's work. I have not used external sources without proper acknowledgment. If I have used AI tools, I will acknowledge the version used and explain how I used it.

Mir Mahathir Mohammad
November 24, 2025

Collaborators: Your Collaborators, including TAs
Sources: Your Sources

Question 1

a

The multicore processor enforces the *Total Store Order (TSO)* memory consistency model.

From the definition of preserved program order:

- If a is a read and b is a read or a write, and a precedes b in program order, then a must precede b in the global memory order.
  ⇒ All Load→Load and Load→Store orders are preserved.
- If a is a write and b is a write, and a precedes b in program order, then a must precede b in the global memory order.
  ⇒ All Store→Store orders are preserved.

The only program-order pair that is not required to be preserved is:

$$\text{Store} \rightarrow \text{Load}$$

i.e., a later load can be reordered before an earlier store to a different address. This matches TSO, whose characteristic relaxation is allowing Store→Load reordering, while still preserving Load→Load, Load→Store, and Store→Store.

It is therefore:

- Weaker than Sequential Consistency (which preserves all program-order pairs, including Store→Load), and
- Stronger than PSO (which would also allow Store→Store reordering).

The load value axiom also matches TSO semantics:

- A load returns the value of the latest store to that address that either:
  1. precedes the load in the global memory order, or
  2. precedes the load in program order.

This corresponds to:

- *Store buffer forwarding*: a load can see an earlier store from the same thread even if that store is not yet globally visible (condition based on program order).
- *Total order of stores*: in the absence of such a local store, the load sees the most recent globally ordered store (condition based on global order), which is provided by the directory-based coherence protocol.

An out-of-order core with a FIFO store buffer can thus:

- Reorder Store→Load to different addresses, but
- Enforce all preserved orders (Load→Load, Load→Store, Store→Store) at commit/retirement.

Hence, the described model and its natural implementation on an out-of-order, directory-coherent multicore correspond to the **Total Store Order (TSO)** memory consistency model.

b

The memory model is Total Store Order (TSO). Two litmus tests illustrating its properties are:

1. **Store Buffering (shows allowed `st` → `ld` reordering)**

   Initial state:
   $$x = 0, \quad y = 0$$

   Threads:

   ```
   Thread 0:  st x = 1;
              ld r0 = y;
   Thread 1:  st y = 1;
              ld r1 = x;
   ```

   Question: Is outcome $r0 = 0, r1 = 0$ allowed?

   In each thread, the pair `st` → `ld` (write → read) is not in preserved program order, so the global memory order may reorder the load before the previous store to a different address.

   A legal global memory order is:

   `ld r0 = y`, `ld r1 = x`, `st x = 1`, `st y = 1`

   By the load value axiom:

   - For `ld r0 = y`: there is no earlier store to y in Thread 0, and in the above global order no store to y precedes this load, so it reads 0.
   - For `ld r1 = x`: there is no earlier store to x in Thread 1, and in the above global order no store to x precedes this load, so it reads 0.

   Thus the outcome $r0 = 0, r1 = 0$ is allowed, demonstrating that the model permits `st` → `ld` reordering to different addresses, as in TSO.

2. **Message Passing (shows preserved `st` → `st`)**

   Initial state:
   $$data = 0, \quad flag = 0$$

   Threads:

   ```
   Thread 0 (sender):    st data = 1;
                         st flag = 1;
   Thread 1 (receiver):  ld r0 = flag;
                         ld r1 = data;
   ```

   Question: Is outcome $r0 = 1, r1 = 0$ allowed?

   In Thread 0 the pair `st data = 1` → `st flag = 1` is a write → write, which is in preserved program order. Therefore any global memory order must have:

   `st data = 1` < `st flag = 1`

If Thread 1 observes $r0 = 1$, then by the load value axiom, `st flag = 1` must precede `ld r0 = flag` in global memory order.

Because `st data = 1` must precede `st flag = 1` in global memory order, it also precedes `ld r0 = flag`, and hence precedes (or can precede) `ld r1 = data`. For `ld r1 = data` to return 0, it would have to ignore this earlier store to the same address, which is forbidden by the load value axiom.

Thus the outcome $r0 = 1, r1 = 0$ is forbidden. Whenever Thread 1 sees `flag = 1`, it must also be able to see `data = 1`. This shows that `st` $\rightarrow$ `st` order is preserved, as required by TSO.

c

To implement the given memory consistency model efficiently on an out-of-order multicore with directory-based coherence, use a standard OoO core with a reorder buffer (ROB), a load/store queue, a FIFO store buffer, and a coherent cache hierarchy.

Core structures

- Reorder buffer (ROB) with in-order retirement.
- Load queue (LQ) and store queue (SQ) for in-flight memory operations.
- Per-core FIFO store buffer (SB) between the core and the L1 cache.
- Coherent L1 cache; directory-based coherence between cores.

Store implementation

- On execution (address and data ready):
  - Allocate an entry in the store queue and in the store buffer with (addr, data, size) and a validity/commit bit.
  - Do not send coherence requests yet; the store is speculative.
- On retirement:
  - Stores retire from the ROB in program order.
  - Mark the corresponding store buffer entry as committed (non-speculative).
- Draining the store buffer (making stores globally visible):
  - The store buffer is strictly FIFO in program order of stores.
  - Only the committed store at the head of the SB may be drained:
    - If the line is not in Modified/Exclusive state, send a GetM/upgrade request to the directory.
    - After invalidations/acknowledgements are received from sharers, write the data into the local L1 in Modified state.
  - This point (update of L1 in Modified state) defines when the store becomes globally visible and its position in the global memory order.

- – FIFO draining of the SB guarantees that write→write program order is preserved in the global memory order.

Load implementation

- Issuing:
  - A load may execute out of order when its address and register operands are ready.
  - It does not have to wait for older stores to *different* addresses (allowing store→load reordering for different addresses).
- Store buffer forwarding (same-core ordering and load-value axiom):
  - On issuing a load to address A:
    - Search the store buffer for older (in program order) stores to A.
    - If one or more matches exist, select the youngest such store and forward its value to the load.
  - This enforces that a load sees the latest earlier same-thread store to that address, even if that store is not yet globally visible, satisfying the "stores preceding in program order" part of the load value axiom.
- If there is no older same-address store in the SB:
  - Access the coherent L1 cache:
    - On hit, read the value.
    - On miss, send a GetS request to the directory to obtain a shared copy.
  - Directory coherence ensures that the value returned is from the latest globally visible store to that address, satisfying the "stores preceding in global memory order" part of the load value axiom.
- Preserving read→read and read→write order:
  - Loads and following memory operations commit in program order in the ROB.
  - Define the load's position in the global memory order at its retirement point.
  - Because retirement is in program order, the global memory order respects all read→read and read→write program order constraints.
- Speculation and recovery:
  - Loads may execute before the addresses of some older stores are known, or before coherence transactions complete.
  - When an older store's address becomes known, compare it against younger loads in the load queue; if a younger load accessed the same address earlier, squash and re-execute from that point.
  - On receiving invalidations or updates for a line read by a speculative load, mark that load for replay or squash and restart from it.
  - This ensures that only loads that satisfy the load value axiom are allowed to retire.

Coherence protocol (directory-based)

- For loads (GetS):
  - Directory tracks sharers and provides data from the latest Modified/Exclusive owner or from memory.
  - Enforces a single coherent order of stores per address as seen by all cores.
- For stores (GetM/upgrades):
  - Head-of-SB committed stores send GetM/upgrade requests.
  - Directory sends invalidations to sharers and waits for acknowledgements.
  - After that, the core writes the data and the store becomes globally visible.

Resulting ordering properties

- write→write program order is preserved by in-order retirement of stores and FIFO store buffer draining.
- read→read and read→write program order is preserved by in-order retirement in the ROB and by defining the load's global position at retirement.
- write→read to different addresses may be reordered because loads can bypass older stores to other addresses via early execution and store buffer forwarding.
- The load value axiom is satisfied by combining:
  - store buffer forwarding for earlier same-thread stores (program order),
  - coherent cache/directory order for globally visible stores (global memory order).

Question 2

Bus-based update coherence protocol

States (per cache line)

- **I** (Invalid): line not present or not valid in this cache.
- **E** (Exclusive-clean): this cache has the only copy, clean with respect to memory; local reads and writes do not cause bus traffic.
- **M** (Modified): this cache has the only copy, dirty with respect to memory; local reads and writes do not cause bus traffic.
- **Sc** (Shared-clean): line may be present in multiple caches, clean with respect to memory; all caches can read without bus traffic.
- **Sm** (Shared-modified / owner): line may be present in multiple caches, memory is stale; this cache is the owner and supplies data to others, and sends updates on writes.

Private read/write variables tend to stay in E or M (no bus traffic on hits). Shared read-only variables stay in Sc. Shared read/write variables use Sm as owner with Sc copies and bus updates on writes.

Events

Local processor events:

- **PrRd**: processor read.
- **PrWr**: processor write.

Bus transactions initiated by this cache:

- **BusRd**: read miss issued on the bus.
- **BusUpd**: update (write) broadcast to other caches on a shared line.

Bus transactions snooped by this cache:

- **BusRd**: another cache issues a read miss.
- **BusUpd**: another cache updates a shared line.

We assume each state transition is atomic and ignore evictions.

State transitions

From I (Invalid).

- PrRd miss:
  - Issue BusRd.
  - If no sharers: $I \rightarrow E$ on PrRd/BusRd (no sharers).
  - If sharers: $I \rightarrow Sc$ on PrRd/BusRd (sharers).

- PrWr miss:
  - Issue BusRd (read-for-ownership).
  - If no sharers: $I \to M$ on PrWr/BusRd (no sharers).
  - If sharers: $I \to Sm$ on PrWr/BusRd (sharers).

From E (Exclusive-clean).

- PrRd hit: $E \to E$.
- PrWr hit: $E \to M$.
- Snooped BusRd from another cache: $E \to Sc$ (requester goes to Sc).

From M (Modified).

- PrRd or PrWr hit: $M \to M$.
- Snooped BusRd from another cache: $M \to Sm$ (requester goes to Sc, this cache becomes owner in Sm).

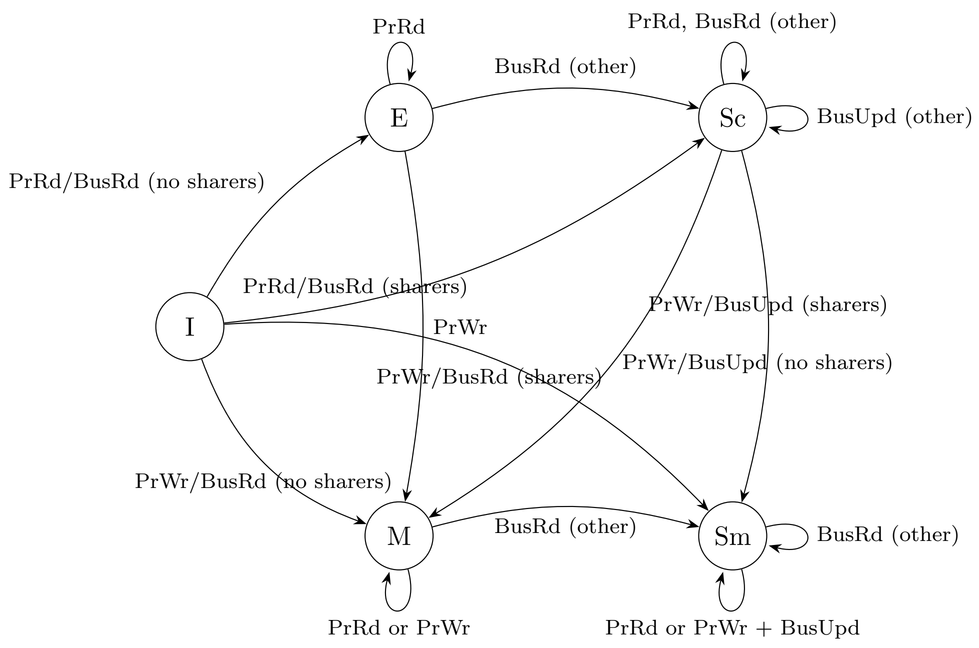
From Sc (Shared-clean).

- PrRd hit: $Sc \to Sc$.
- PrWr hit:
  - Cache writes and sends BusUpd.
  - If other sharers exist: $Sc \to Sm$ on PrWr/BusUpd (sharers).
  - If effectively no other sharers: $Sc \to M$ on PrWr/BusUpd (no other sharers).
- Snooped BusUpd from another cache: $Sc \to Sc$ (data updated, state unchanged).
- Snooped BusRd from another cache: $Sc \to Sc$.

From Sm (Shared-modified / owner).

- PrRd hit: $Sm \to Sm$.
- PrWr hit: owner writes and sends BusUpd, $Sm \to Sm$.
- Snooped BusRd from another cache: owner supplies data, $Sm \to Sm$ (requester goes to Sc).
- Snooped BusUpd from another cache does not normally occur (only one owner); can be treated as no effect.

State-transition diagram (TikZ)



Question 3

We reinterpret the 10-bit sharing vector as a group vector instead of a per-core vector.

1. Partition the 16 cores into 10 disjoint groups, each group mapped to one bit of the sharing vector. For example:

$$\begin{aligned}
&\text{Bit } 0 \to \{0,1\},\\
&\text{Bit } 1 \to \{2,3\},\\
&\text{Bit } 2 \to \{4,5\},\\
&\text{Bit } 3 \to \{6,7\},\\
&\text{Bit } 4 \to \{8,9\},\\
&\text{Bit } 5 \to \{10,11\},\\
&\text{Bit } 6 \to \{12\},\\
&\text{Bit } 7 \to \{13\},\\
&\text{Bit } 8 \to \{14\},\\
&\text{Bit } 9 \to \{15\}.
\end{aligned}$$

   Each core belongs to exactly one group.

2. Change the semantics of the directory bits:
   - Original (10-core) system: bit $i = 1 \Rightarrow$ core i has (or may have) a copy.
   - New (16-core) system: bit $j = 1 \Rightarrow$ at least one core in group j may have a copy.

   This may introduce false positives (some cores in a group are invalidated even if they do not cache the line), but never false negatives, so correctness is preserved.

3. Coherence actions:
   - **Read miss from core k:** Let $G(k)$ be the group of core k.

(a) If the line is Uncached: supply data from memory and set the bit corresponding to $G(k)$.

(b) If the line is Shared: set the bit of $G(k)$ (if not already set) and supply data (from memory or a sharer).

(c) If the line is Modified/Exclusive in some group $G(m)$: recall/forward data from the owner in $G(m)$ and update the state (e.g., to Shared, with bits for $G(m)$ and $G(k)$ set, or transfer ownership).

- **Write miss or upgrade by core k:**

  (a) The directory inspects the 10-bit vector; for each bit set (except that of $G(k)$), it sends invalidations to *all* cores in that group.

  (b) Cores that actually hold the line invalidate it and send acknowledgments.

  (c) After receiving all acknowledgments, the directory leaves only the bit for $G(k)$ set and marks the line as Modified/Exclusive for that group.

- **Evictions:** When a core evicts a shared line, the directory can clear the corresponding group bit once no core in that group holds the line anymore (or conservatively leave it set, which only increases unnecessary invalidations but does not affect correctness).

4. This scheme maintains the single-writer, multiple-reader invariant and ensures that all sharers see a consistent value. The only overhead is extra invalidation traffic due to group-level false positives, but the directory entry size remains 10 bits.