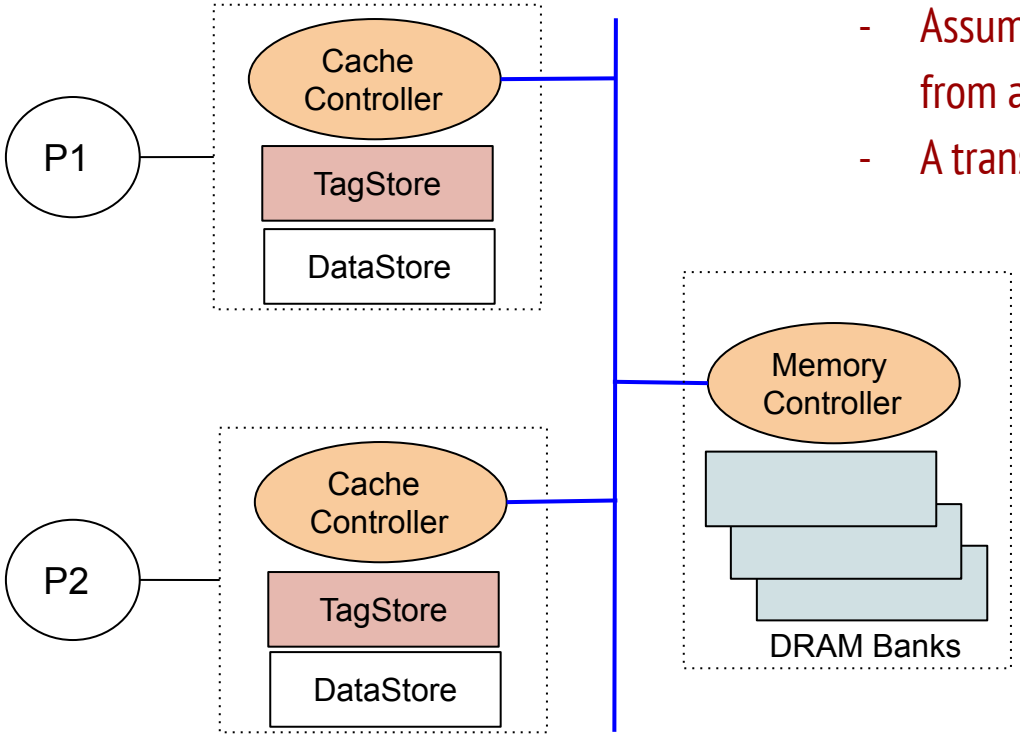
CS622: Advanced Computer Architecture

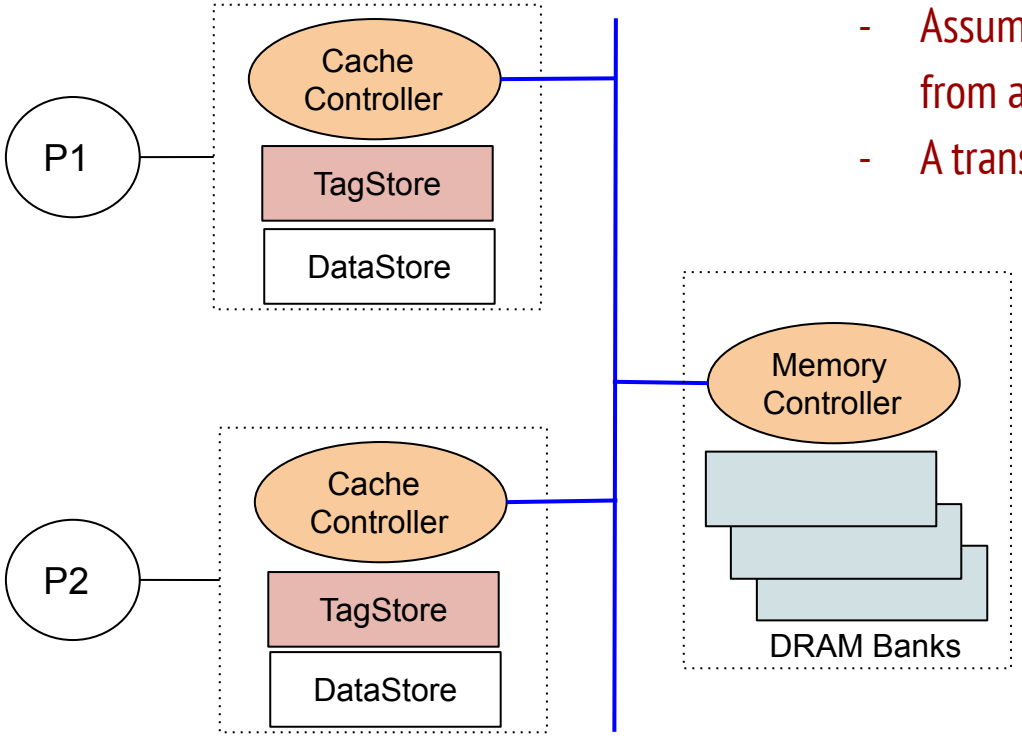
Design of Snoop-based Coherence

Single Level Caches with Atomic Bus



- Assumptions: Single level cache, one outstanding write from any processor at a given time and bus is atomic
- A transaction using atomic bus
  - Request to acquire the bus
  - Wait for bus grant by the bus controller
  - Place address and/or command
  - Wait for ack from the device
  - Transfer data

Single Level Caches with Atomic Bus

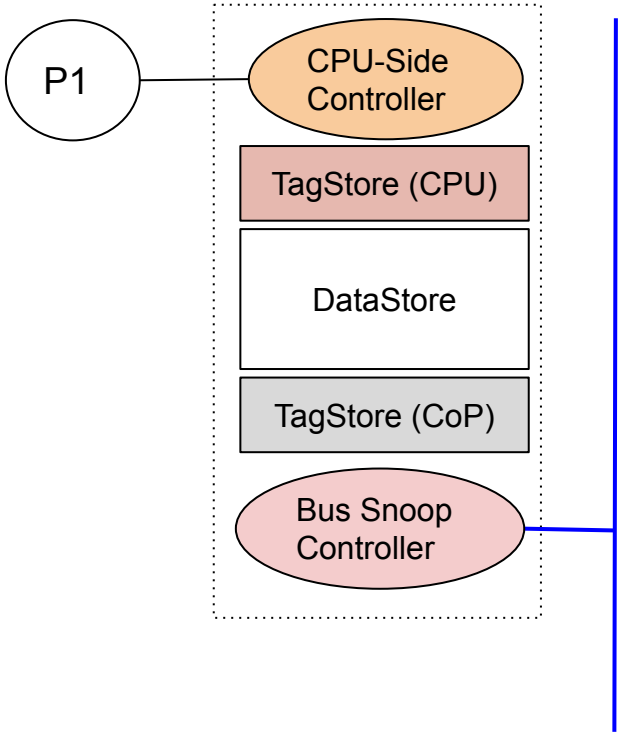


- Assumptions: Single level cache, one outstanding write from any processor at a given time and bus is atomic
- A transaction using atomic bus
  - Request to acquire the bus
  - Wait for bus grant by the bus controller
  - Place address and/or command
  - Wait for ack from the device
  - Transfer data

CPU side requests can stall because of contention, mostly due to tag store access from both sides

- Without coherence, the cache controllers require to monitor (and respond) only to the CPU side bus
- Cache coherence requires monitoring the mem-side bus and access tag store (to match addresses)

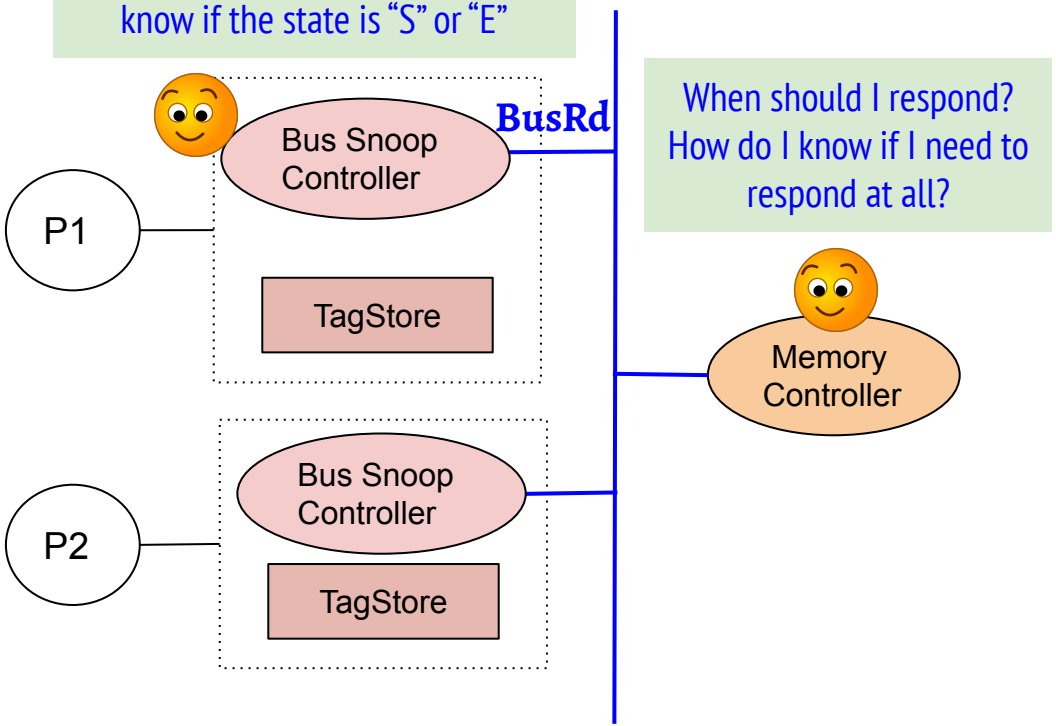
Duplicated Tag Stores



- Two tag stores can be read in a concurrent manner. What about modifications to tag entries?
- As a simple design choice, the content of the two tag sets should be synchronized.
- During sync operation, access should be blocked
- Possible optimizations
  - Update the CPU side tags only when the cache block is modified
  - Subtle cases: invalidations from the bus-side (due to BusUpgr/ReadX), E $\longrightarrow$ M transition in MESI

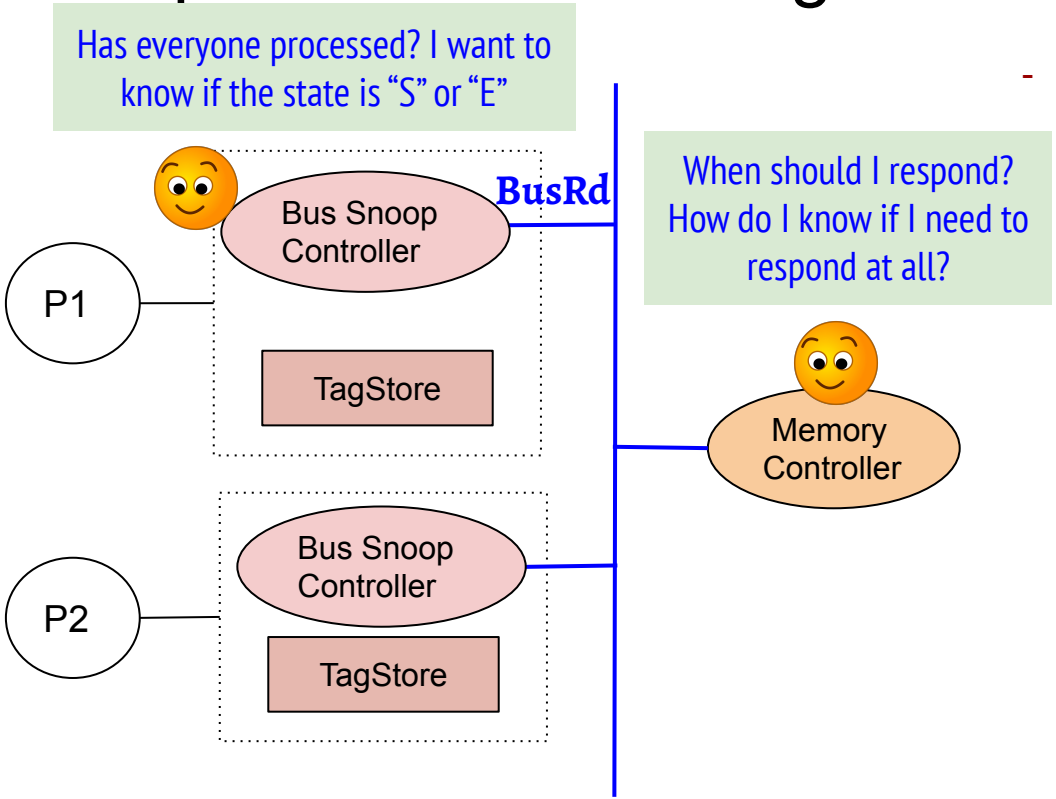
Snoop Result and Timing

Has everyone processed? I want to know if the state is "S" or "E"



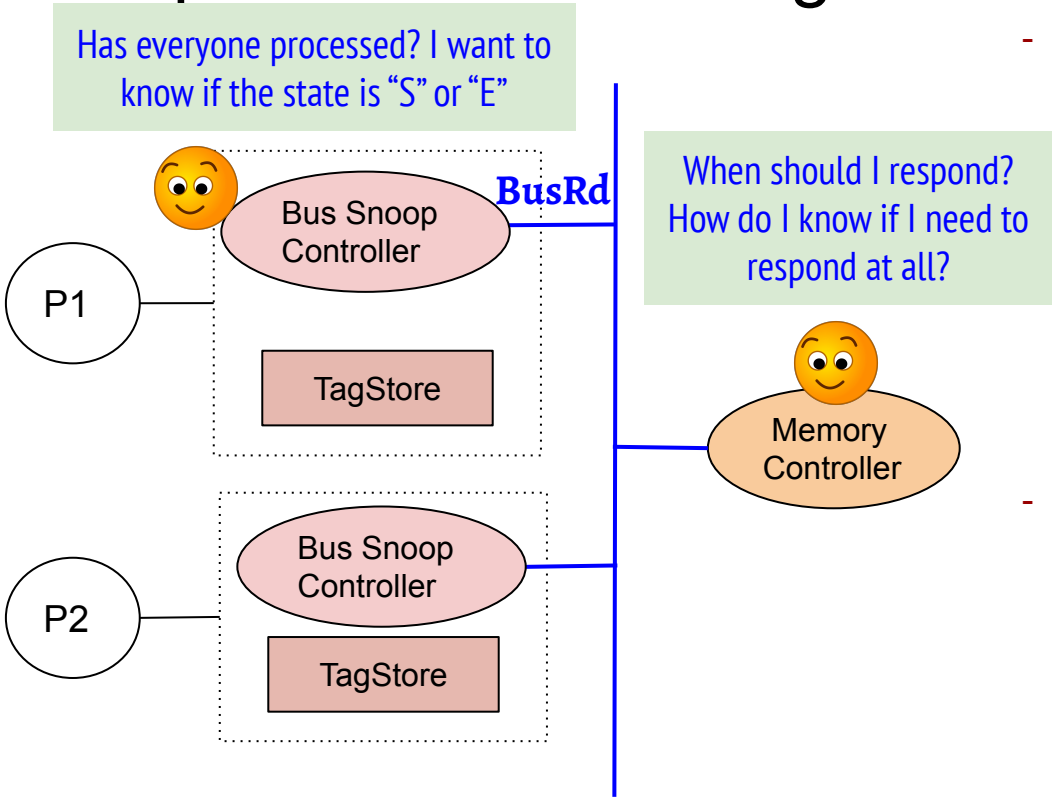
When should I respond? How do I know if I need to respond at all?

Snoop Result and Timing



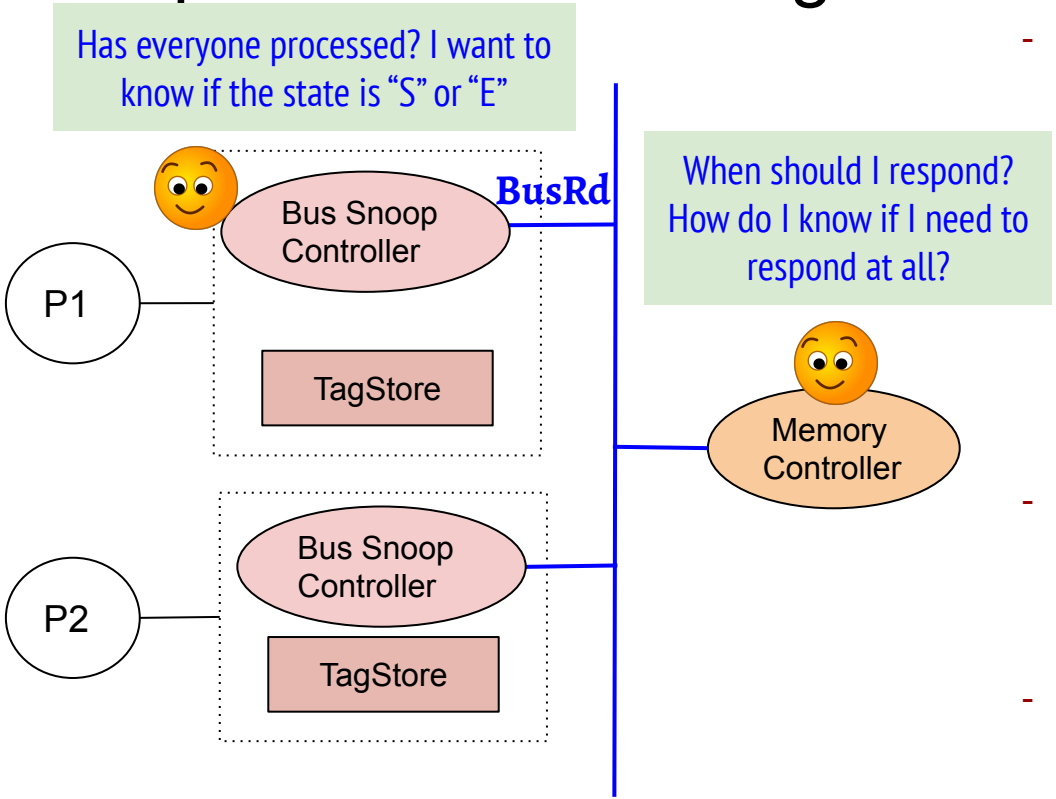
- One possible solution is to collect the responses from the snoopers and formulate a three-bit response
- **bit-0**: The bit is initially 1 and becomes 0 only when all processors have responded → resolves the issue of timing (avoids worst-case wait)

Snoop Result and Timing



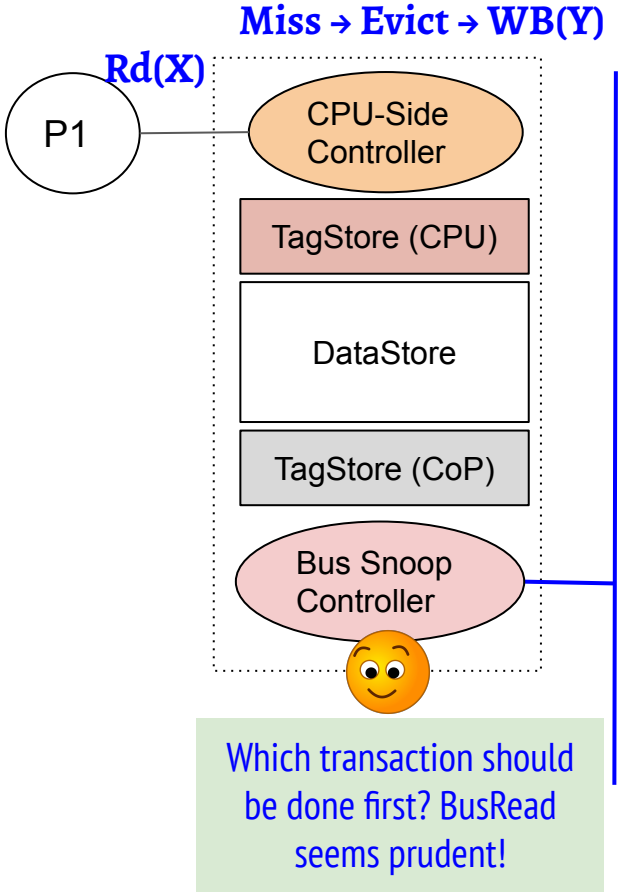
- One possible solution is to collect the responses from the snoopers and formulate a three-bit response
  - **bit-0**: The bit is initially 1 and becomes 0 only when all processors have responded → resolves the issue of timing (avoids worst-case wait)
- **bit-1**: The bit is set if *any* processor has the block in its cache → resolves "E" or "S" for the requesting processor

Snoop Result and Timing



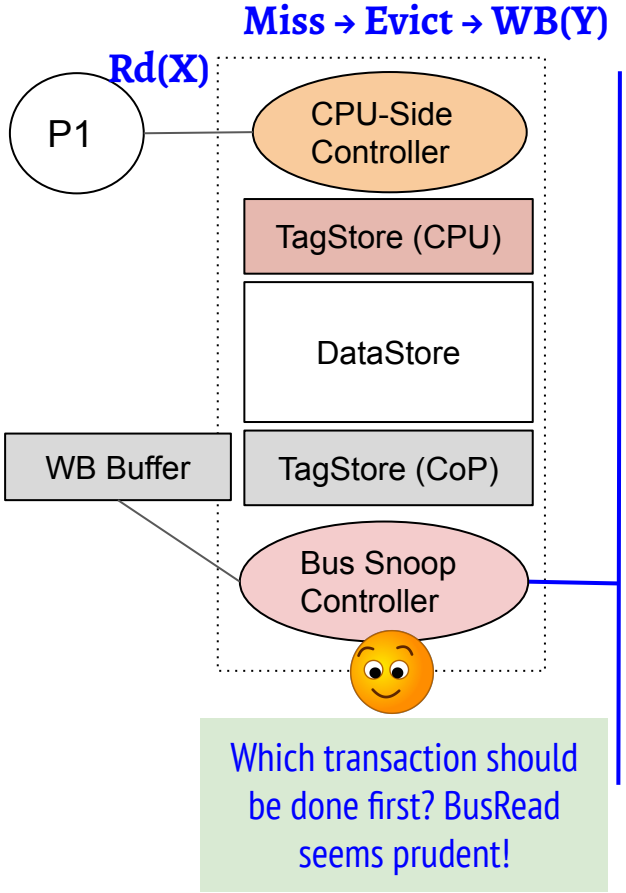
- One possible solution is to collect the responses from the snoopers and formulate a three-bit response
  - **bit-0**: The bit is initially 1 and becomes 0 only when all processors have responded ⟶ resolves the issue of timing (avoids worst-case wait)
- **bit-1**: The bit is set if *any* processor has the block in its cache ⟶ resolves "E" or "S" for the requesting processor
- **bit-2**: The bit is set if *any* processor has the block in modified state ⟶ Determines the data provider i.e., memory controller or one of the processors

Write-back handling



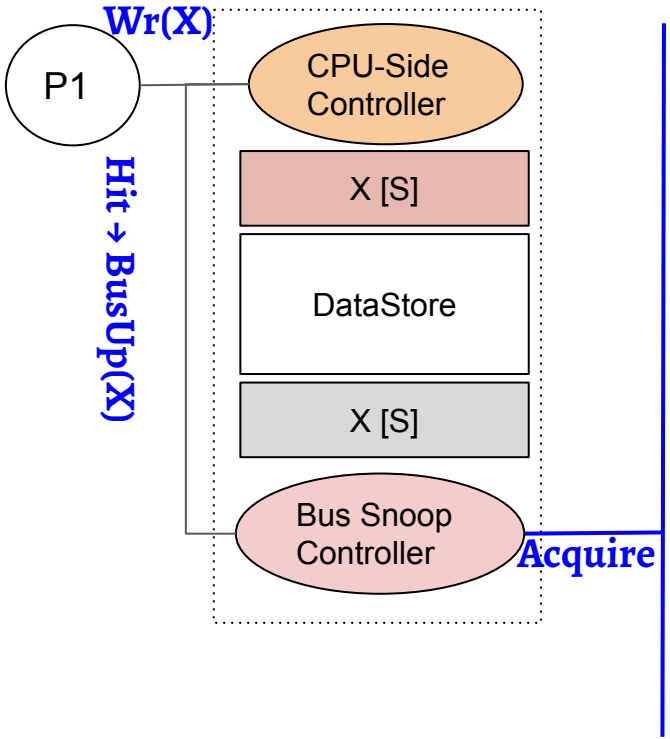
- If *BusRead* transaction is prioritized, the write-back should be stored into a buffer (write-back buffer)
- The write-back operation is initiated after the transaction for *BusRead* is completed
  - What if another processor sends a coherence message for **Y** before write-back is completed?

Write-back handling



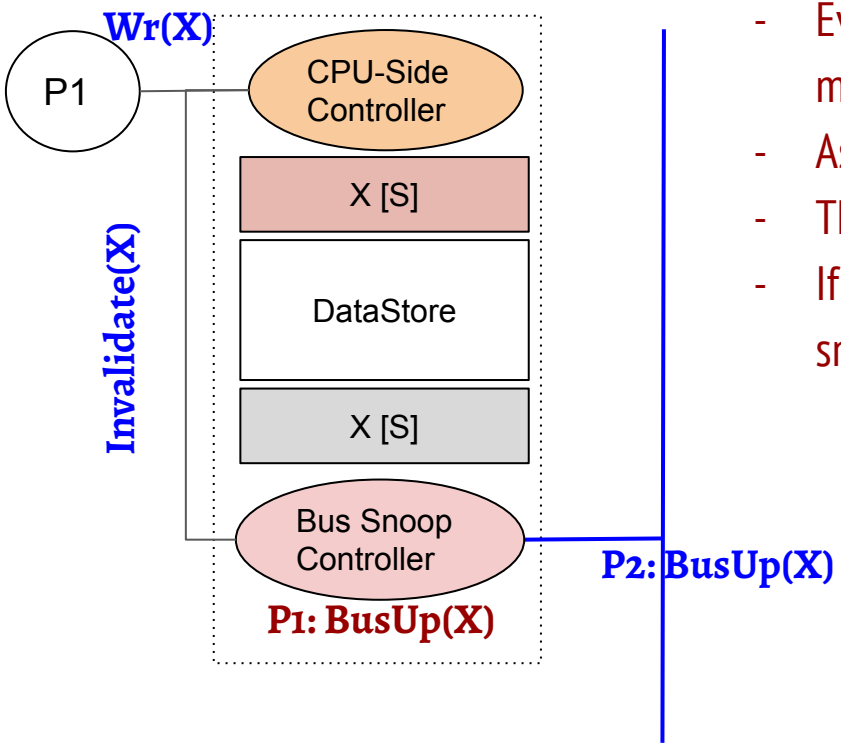
- If *BusRead* transaction is prioritized, the write-back should be stored into a buffer (write-back buffer)
- The write-back operation is initiated after the transaction for *BusRead* is completed
  - What if another processor sends a coherence message for **Y** before write-back is completed? The snoop controller should also match the address with the WB buffer and provide the data (flush) and cancel write-back

Non-atomic Operations



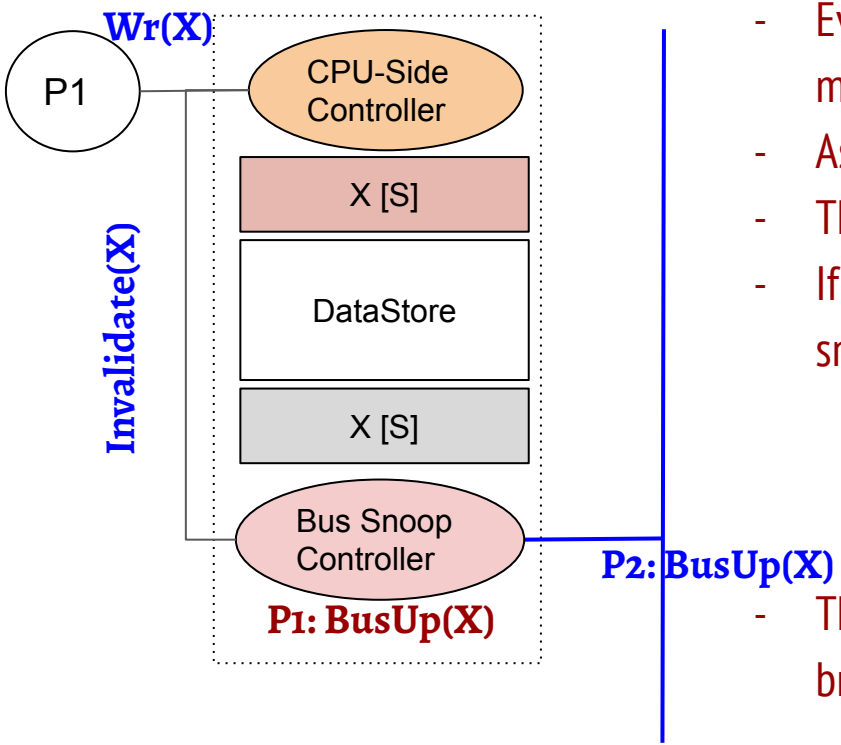
- Even though the bus is atomic, the operations in a processor may not be atomic leading to erroneous protocol actions
- Assume that X is in “S” state in P1 and another processor P2
- The bus-side controller tries to acquire the bus

Non-atomic Operations



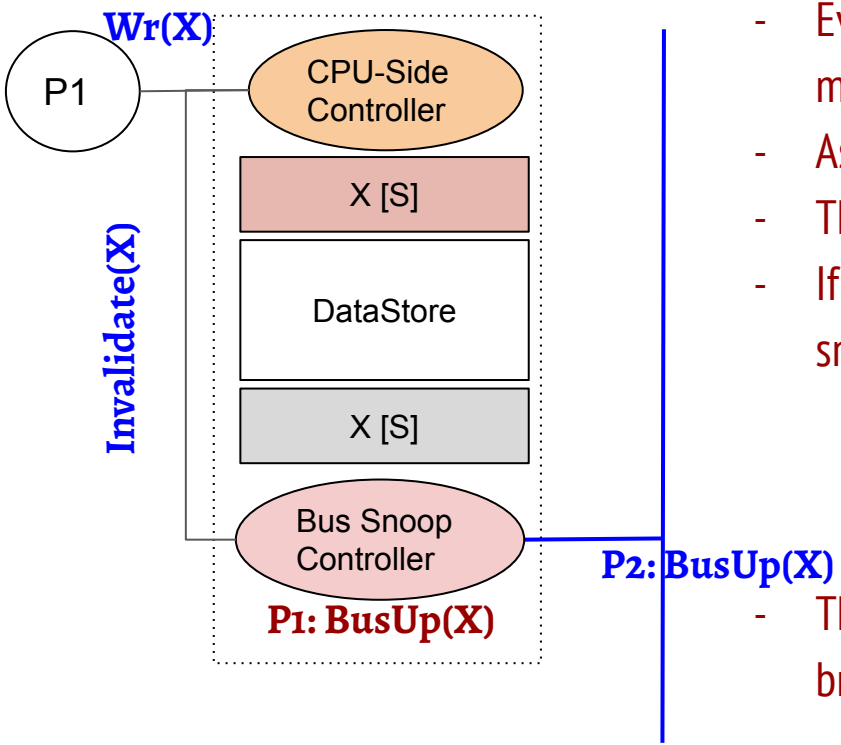
- Even though the bus is atomic, the operations in a processor may not be atomic leading to erroneous protocol actions
- Assume that X is in "S" state in P1 and another processor P2
- The bus-side controller tries to acquire the bus
- If P2 wins the race and broadcasts the *BusUp* on the bus, P1 snoop controller
  - (a) Invalidate the block and acknowledge OR
  - (b) Keep on trying to acquire the bus

Non-atomic Operations



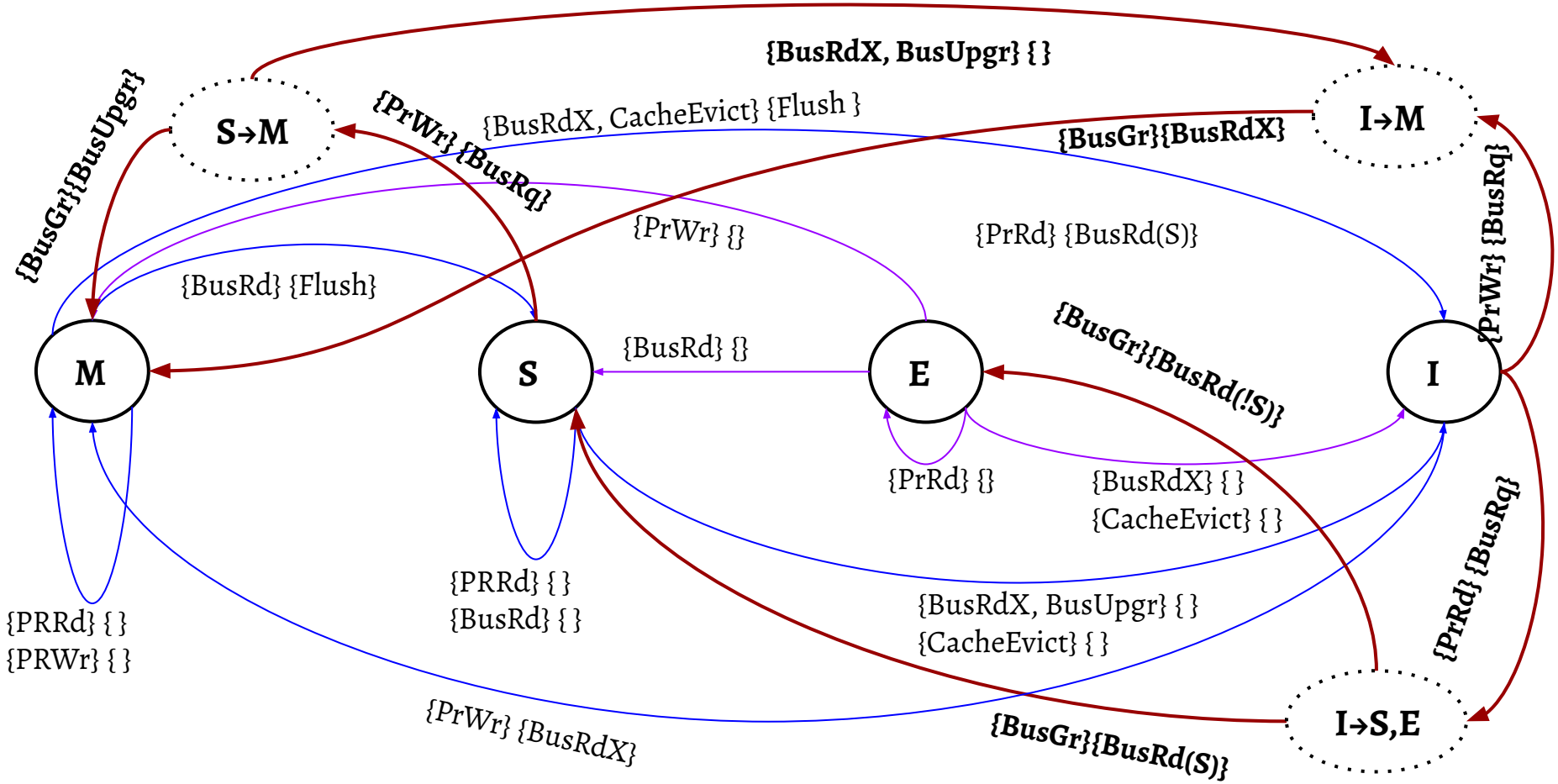
- Even though the bus is atomic, the operations in a processor may not be atomic leading to erroneous protocol actions
- Assume that X is in "S" state in P1 and another processor P2
- The bus-side controller tries to acquire the bus
- If P2 wins the race and broadcasts the *BusUp* on the bus, P1 snoop controller
  - (a) Invalidate the block and acknowledge OR
  - (b) Keep on trying to acquire the bus
  - P1 has to respond otherwise leads to a deadlock
- The *BusUp* of P1 is still to be completed. If P1 controller broadcasts BusUp, it will be violation of the protocol, why?

Non-atomic Operations

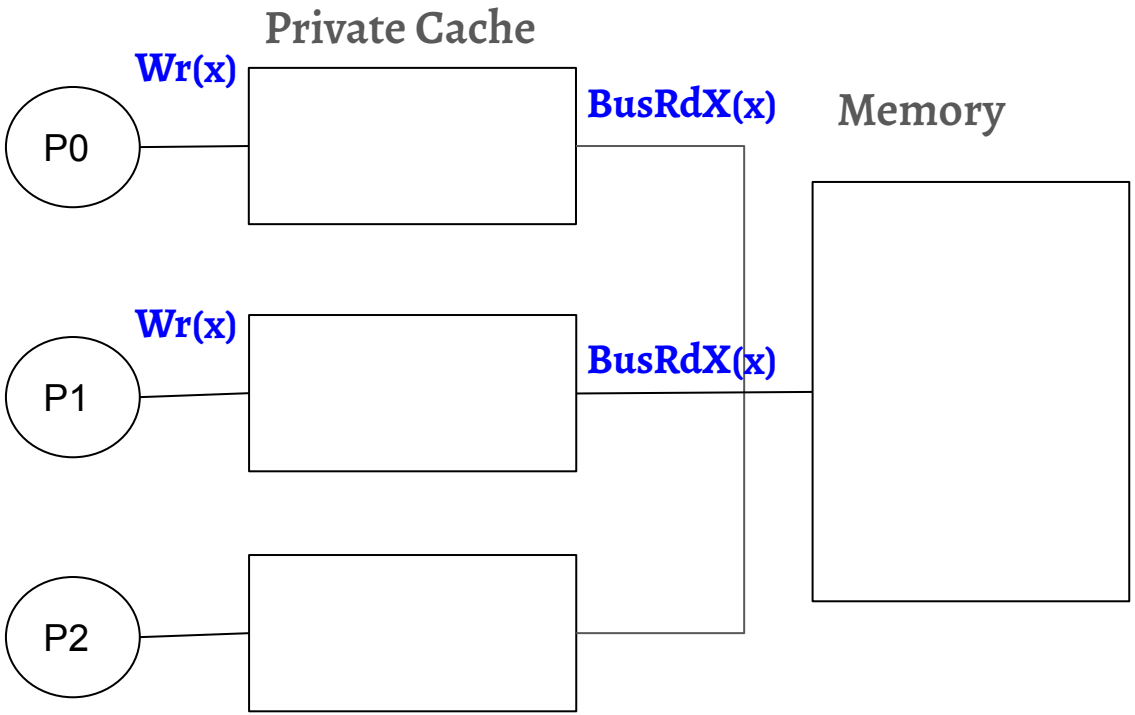


- Even though the bus is atomic, the operations in a processor may not be atomic leading to erroneous protocol actions
- Assume that X is in "S" state in P1 and another processor P2
- The bus-side controller tries to acquire the bus
- If P2 wins the race and broadcasts the *BusUp* on the bus, P1 snoop controller
  - (a) Invalidate the block and acknowledge OR
  - (b) Keep on trying to acquire the bus
  - P1 has to respond otherwise leads to a deadlock
- The *BusUp* of P1 is still to be completed. If P1 controller broadcasts BusUp, it will be violation of the protocol, why?
  - P1 should send *ReadX* (as per the MESI protocol)
- One way to tackle these non-atomic operations is to introduce transient states in the protocol

MESI with transient states

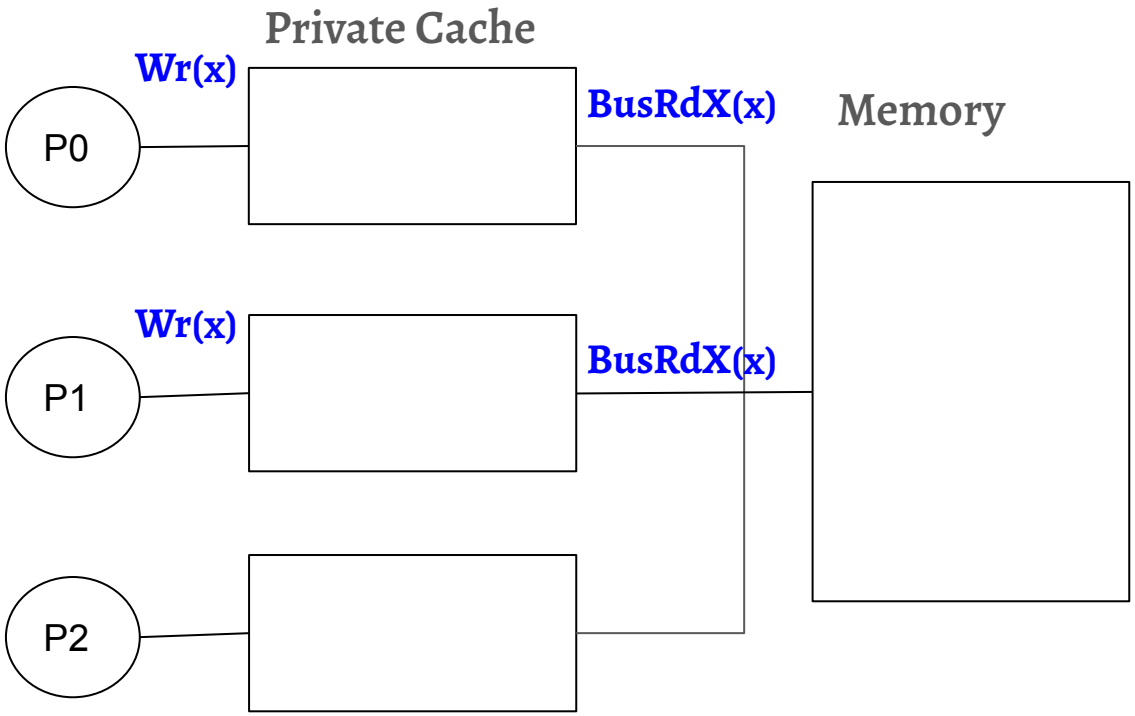


Livelock and Starvation



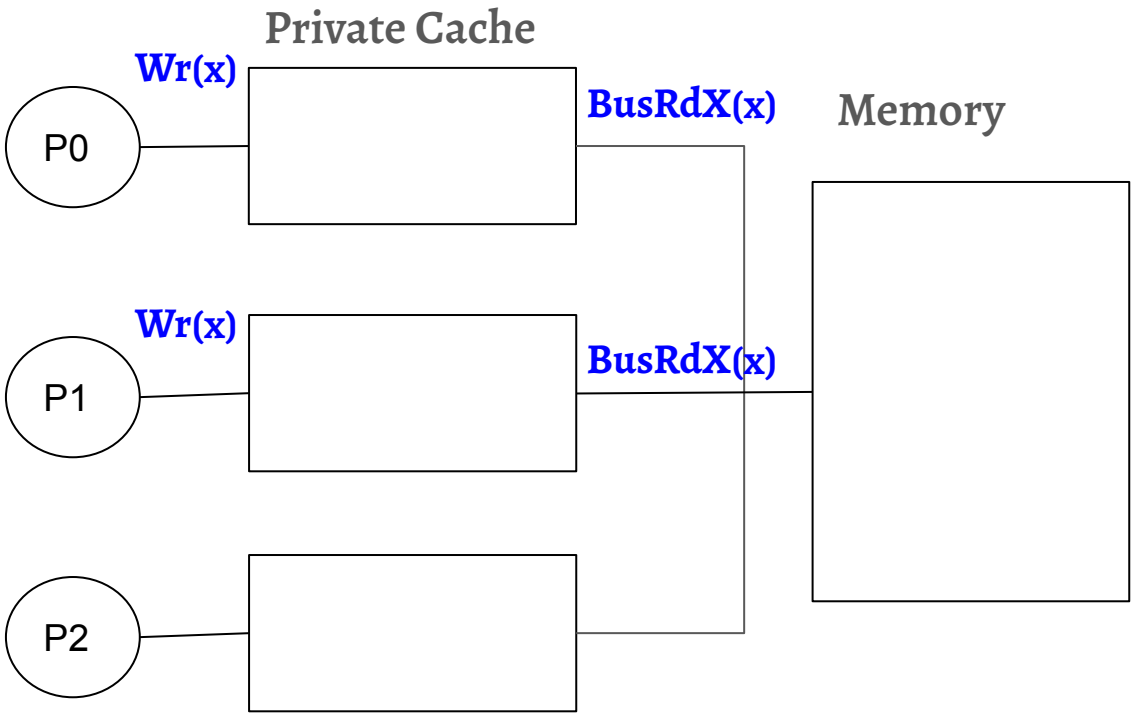
- Assume that P0 acquires the bus, gets the content into its memory in 'M' state
- Before the write operation is complete in P0, P1's *BusRdX* is received at P0 invalidating the block (after flush)
- If the above steps continue, a livelock situation arises
- Solution?

Livelock and Starvation



- Assume that P0 acquires the bus, gets the content into its memory in 'M' state
- Before the write operation is complete in P0, P1's *BusRdX* is received at P0 invalidating the block (after flush)
- If the above steps continue, a livelock situation arises
- Solution? Bus controller should ensure cache write before responding

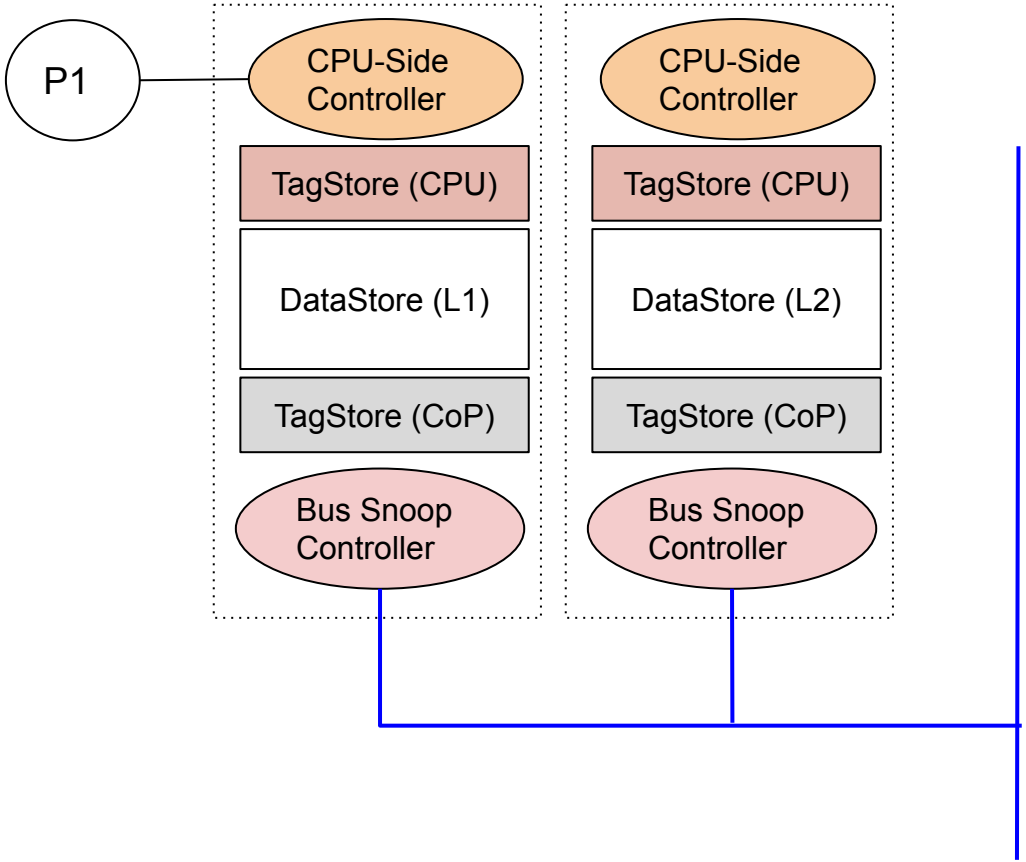
Livelock and Starvation



The bus arbitration logic should maintain fairness – using ordering to implement FCFS (need extra state)

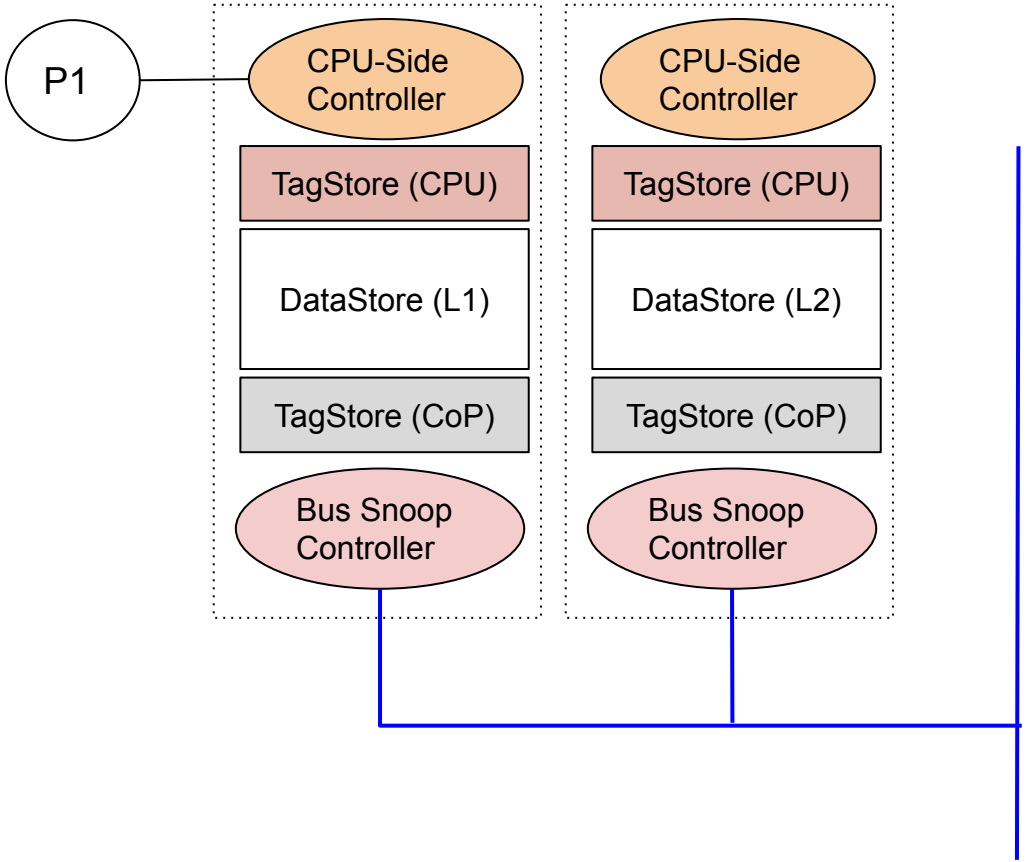
- Assume that P0 acquires the bus, gets the content into its memory in 'M' state
- Before the write operation is complete in P0, P1's *BusRdX* is received at P0 invalidating the block (after flush)
- If the above steps continue, a livelock situation arises
- Solution? Bus controller should ensure cache write before responding

Multi-level Caches with Atomic Bus



- Independent snooping at all private levels could be a possible solution. Disadvantages?

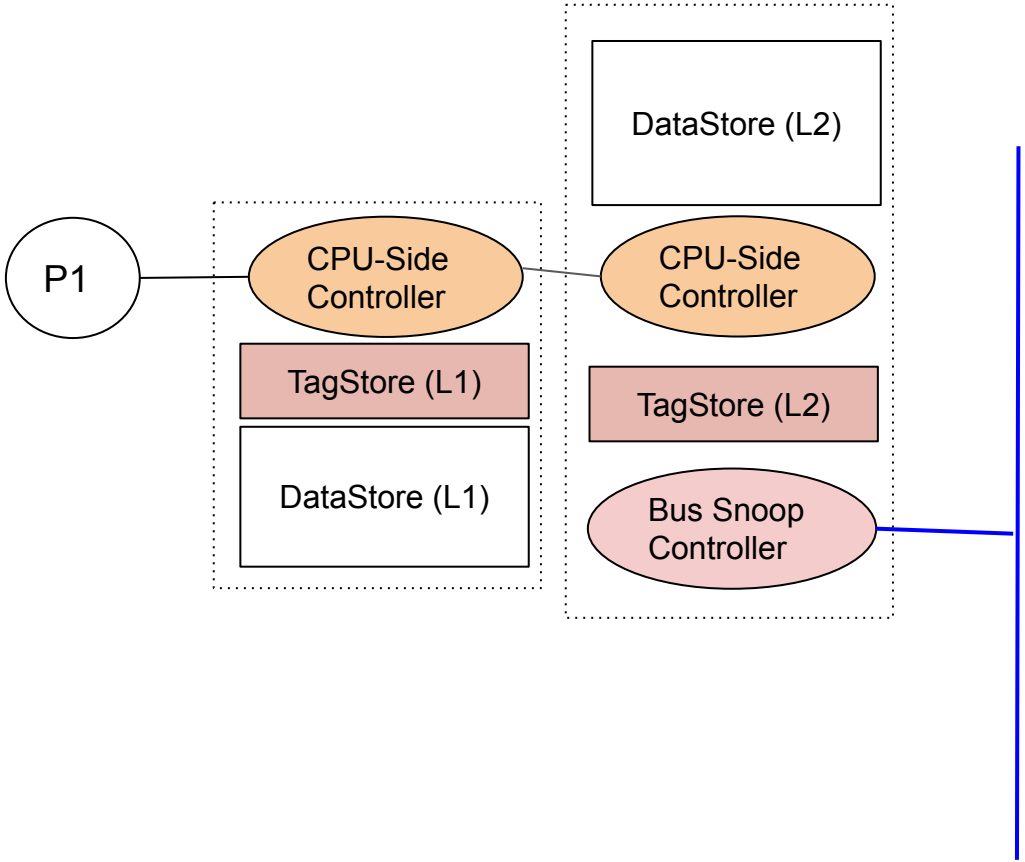
Multi-level Caches with Atomic Bus



- Independent snooping at all private levels could be a possible solution. Disadvantages?
  - Increased hardware cost (logic and storage)
  - Duplicated processing as L2 are (mostly) inclusive
- What if Snoop logic is implemented in L2 (outermost) level?

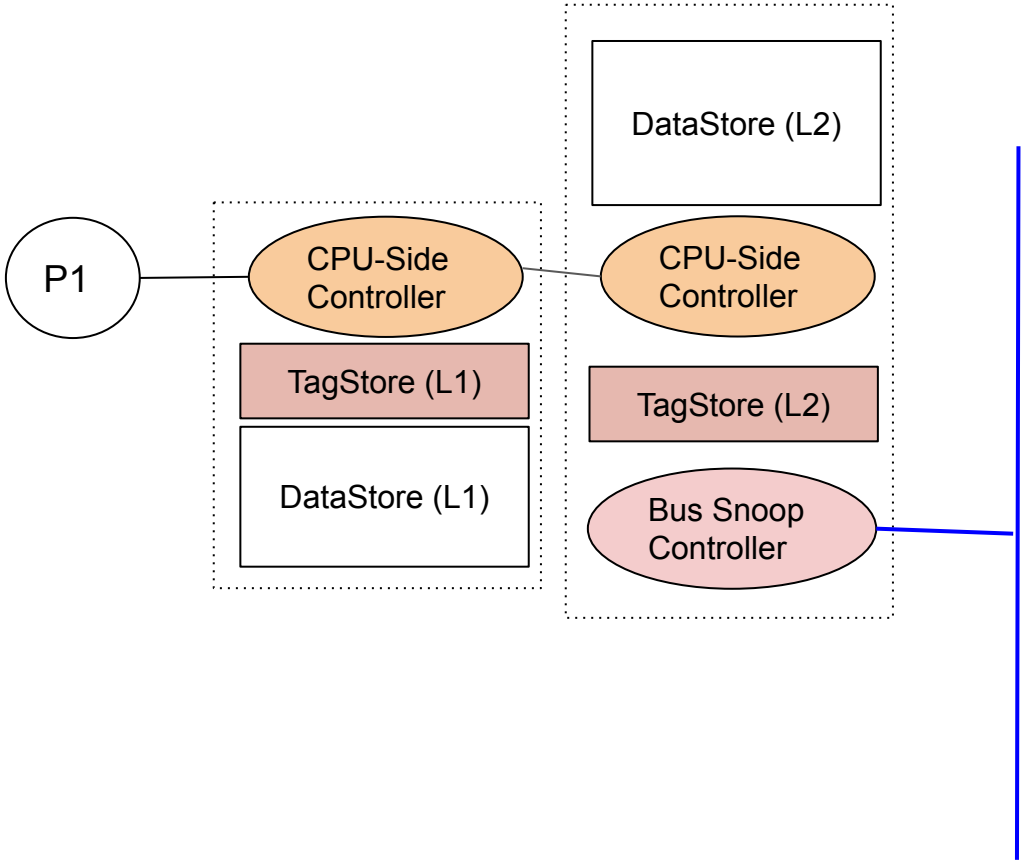
  - L2 access is not as latency sensitive as L1 (dup-tags may be avoided)
  - Require additional constraints for correctness

Two-level snoopy caches without tag duplication



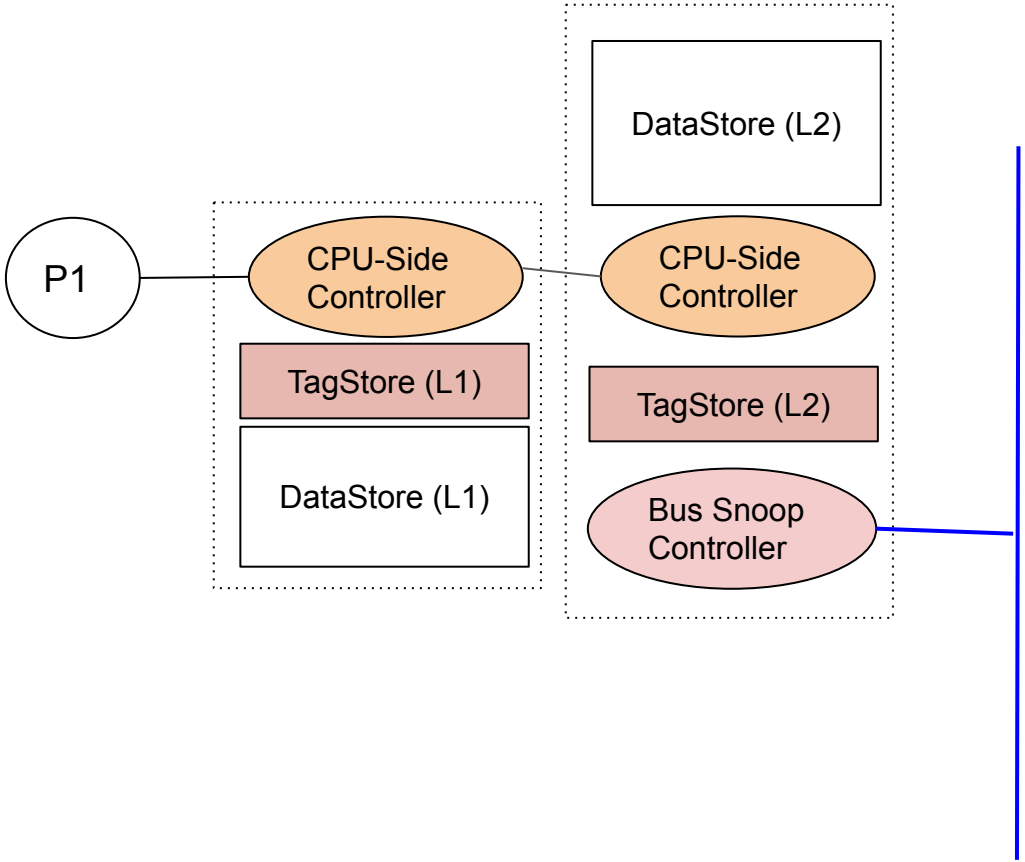
- L2 tags are accessed by both the cache and snoop controllers
- How correctness is ensured?

Two-level snoopy caches without tag duplication



- L2 tags are accessed by both the cache and snoop controllers
- How correctness is ensured?
- Inclusion property
  - If the block is in L1, it must be present in L2
  - If a block is “M” in L1, it must be marked “M” in L2

Two-level snoopy caches without tag duplication



- L2 tags are accessed by both the cache and snoop controllers
- How correctness is ensured?
- Inclusion property
  - (P1) If the block is in L1, it must be present in L2
  - (P2) If a block is “M” in L1, it must be marked “M” in L2
- How to maintain inclusive caches (i.e., P1)? What is the cost of ensuring P2?

Enforcing inclusion

- Eviction from L2 sends an invalidation to L1 (Checking the WB buffer may be needed)
- Processor access to L1 may change the state of a cache block (clean⟶dirty) or evict a block ({clean, dirty} ⟶ Invalid).
  - Making L1 write-through simplifies coherence at the cost of increased BW
  - For WB caches, the state of the cache block in L2 can be marked “modified but stale” to allow write hits at L1 to proceed without any communication

Enforcing inclusion

- Eviction from L2 sends an invalidation to L1 (Checking the WB buffer may be needed)
- Processor access to L1 may change the state of a cache block (clean⟶dirty) or evict a block ({clean, dirty} ⟶ Invalid).
  - Making L1 write-through simplifies coherence at the cost of increased BW
  - For WB caches, the state of the cache block in L2 can be marked “modified but stale” to allow write hits at L1 to proceed without any communication
  - Actions that modify the state from E⟶M or S⟶M must be performed through the coherence controller
- Bus transactions at L2 need to be forwarded to L1
  - With inclusion, #of invalidations due to coherence (BusUp/BusRdX) can be reduced
  - In non-inclusive settings, L2 can maintain extra bits to record if block is present in L1