# Lab 5: Coherence

Mehrad Khalesi 998694202 Hooman Haji 999615000

## Pre-lab Questions:

**1. Why are transient states necessary?**

- In real hardware, state transitions along with their associated actions do not happen instantaneously and atomically due to interconnection network delays. To resolve this issue, transient states which are intermediate states are used to facilitate the transitions between two stable states. The block stays in a transient state until all the associated actions of the transition are completed and then can safely transition to a stable state.

**2. Why does a coherence protocol use stalls?**

- Blocks in transient states are stored outside the cache, in some temporary buffers commonly known as Miss Handling Registers (MSHRs). A cache-entry is also reserved at this point. Each MSHR entry holds the data block, along with its current transient state information. some requests to blocks in transient states will stall until data will be ready or all acks will be received. Once the block transitions to a stable state, the data block and stable state information is copied to the cache entry, and any previously stalled accesses can safely proceed in order.

**3. What is deadlock and how can we avoid it?**

- Deadlock is the phenomenon where no forward progress is made in a system because of two inter-dependent events. For example, if an event A can only occur after event B completes, and event B can only complete after event A occurs, then none of these two events will ever take place. This circular resource dependencies between A and B indefinitely deadlocks the system.
- A solution to avoid deadlock in coherence protocols is to use separate networks for each class of message. (request message, response message and forward message)

**4. What is the functionality of Put-Ack (i.e., WB-Ack) messages? They are used as a response to which message types?**

- Put_Ack is issued to acknowledge a write back request (Replacement). It is used as a response to PUTS variations :(PUTS_NotLast, PUTS_Last) and PUTM variations: (PUTM_DataOwner, PUTM_Data_From_NonOwner)

**5. What determines who is the sender of a data reply to the requesting core? Which are the possible options?**

- Directory controller knows who should be the sender of a data reply to the requesting core. There are three possible options
  - if a block is invalid/uncached state (I or U)in the directory, the directory controller needs to retrieve the data block from memory and then send it to the requesting core.
  - If a block is shared state (S)in the directory, the directory can reply with the data

    - If a block is modified state (M) in the directory, then directory's copy is out-of-date, the directory  controller forwards the request to the owner and the block's owner provides the data reply.

**6. What is the difference between a Put-Ack and an Inv-Ack message?**

- Put_Ack is issued by directory controller to acknowledge write back requests (Replacement). Inv_Ack is issued by cache controllers to acknowledge they have invalidated their copies of the block in state S.

## Section 5 Questions:

**1. How does the FSM know it has received the last Inv Ack reply for a TBE entry?**

- The directory responds with data and AckCount equal to number of sharers to the requesting core, plus it forwards Invalidations to each core in the sharer list. Cache controllers that receive Invalidation messages invalidate their shared copies and send Inv-Acks to the requestor. The requester each time receive Inv-Ack updates the AckCount which is equal to number of Inv-Acks that is still waiting for. When the requestor receives the last Inv-Ack, it transitions to state M.

**2. How is it possible to receive a PUTM request from a NonOwner core? Please provide a set of events that will lead to this scenario.**

- Here is a scenario. Assume there are 2 cores and one core (C1) is the owner of the block A in state M and the directory has block A as M and the owner as C1. Let's say C2 wants to read from block A and at the same time C1 a replacement was invoked for block A as well. C2 notifies directory controller that it wants to read from block A. Since directory has C1 as the owner, the directory controller removes C1 as the owner (since the next state will be S) and forwards the Fwd_GetS to C1 since it has the most up to date data for block A. At this stage, C1 is in MI_A because of the concurrent replacement. Once it receives Fwd_GetS, it send data to C2 and C2 state for block A transitions to state S. Now for some reason the PUTM requested by C1 is delayed. Now the directory has block A as in S and the delayed PUTM arrives from C1 which is no longer the owner of the block. In fact, the block has no owner in S. So directory controller receives a PUTM request from a NonOwner core.

**3. Why do we need to differentiate between a PUTS and a PUTS-Last request?**

- Let's say we have block A in state S with a list of sharers in the directory. Now assume replacement invoked for block A in the cores having the block in state S. As long as directory controller receives PUTS for block A, the actions that need to be performed is to remove the requesting core from the sharers list and send a Put_Ack to the requesting core but the state won't be changed and remains in S. However, if directory controllers receives a PUT_LAST request, it does the same two actions but it transitions to state I since there is no more core having this block in S state. Therefore, we need to differentiate between PUTS and PUTS_Last request.

**4. How is it possible to receive a PUTS Last request for a block in modified state in the directory? Please provide a set of events that will make this possible.**

- Here is a scenario. Assume there are 2 core and one core C1 has block A in S state and the directory has block A in S state and C1 as the only sharer in the sharers list. Let's say C1 wants to replace block A sends a PutS and at the same time C2 want to writes to block A  and send a GetM request to the directory. Assume C2's request reaches the directory first. So directory contoller performs the following actions. It sends the data and AckCount (number of sharers here is only 1) to C2, forwards Invalidation message to C1 and transitions from S to M. Now for some reason the PUTS requested by C1 is delayed and reaches the directory. Block A in the directory receives a PUTS_Last request since C1 was the only and the last block core having block A in S state. Therefore, block A receives a PUTS_Last request while it is in M state in the directory.

**5. Why is it not possible to get an Invalidation request for a cache block in a Modified state? Please explain.**

- MSI coherence protocol enforces single writer, multiple reader (SWMR) invariant. Therefore, at any point in time for any block A,
    - either all caches have it in I state
    - or some caches have it S state and others in I state
    - or only one cache have it in M state and others in I state

  If a cache wants to write into block A and directory has the block in S, the directory controller sends an Invalidation request to the cores in block A's sharers list (where all of them have block A in S state not M).

  If a cache wants to write into block A and directory has the block in M, the directory controller forwards GetM  request to the only owner having the block in state M and the owner will provide the requesting core with the data and transitions to I state since there has to be only one writer.

  Therefore, it is not possible to get an Invalidation request for a cache block in M state.

**6. Why is it not possible for the cache controller to get a Fwd-GetS request for a block in SI_A state? Please explain.**

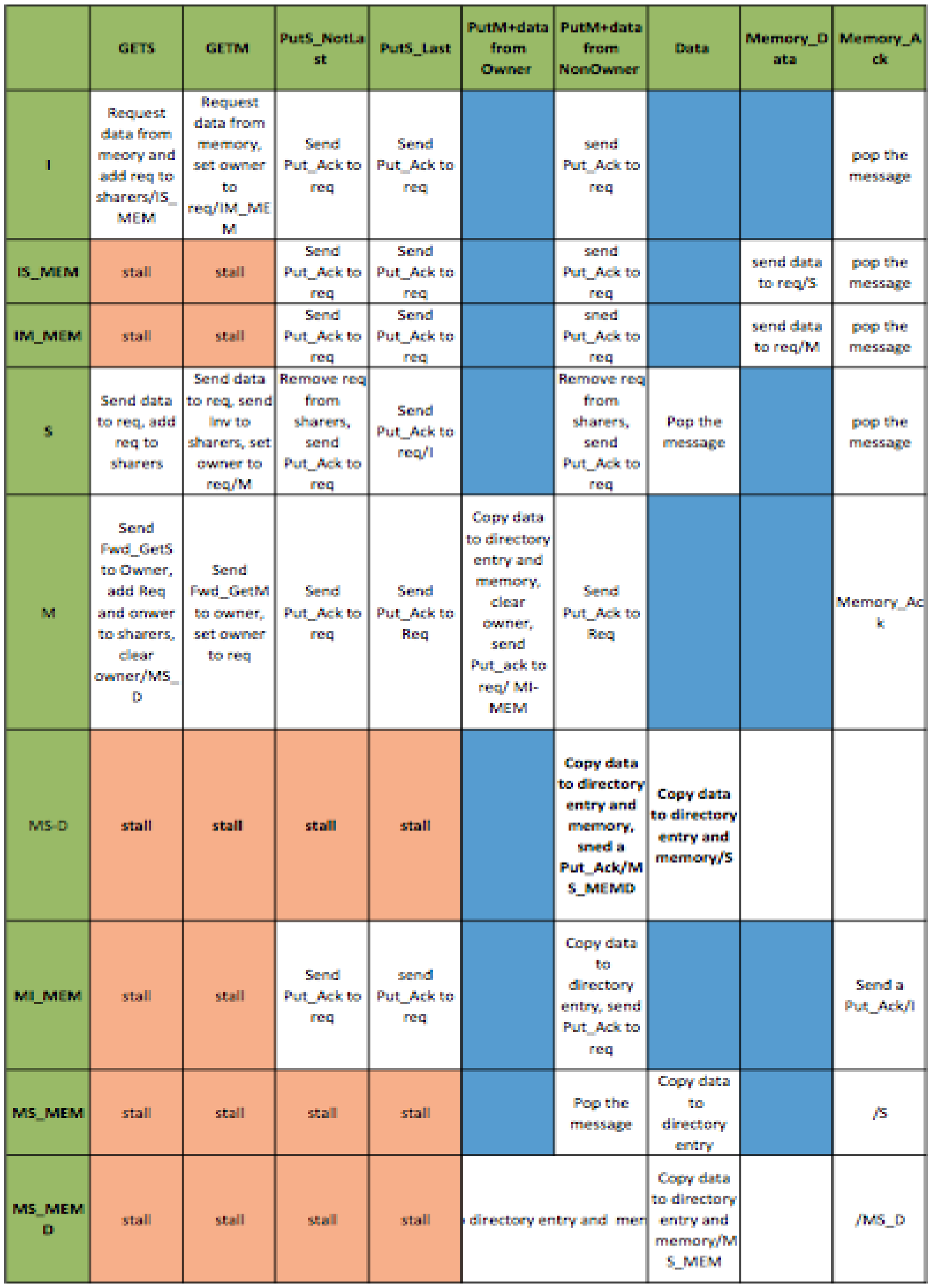
- When a directory controller receives a GetS request it sends Fwd_GetS request if and only if the block is in state M. In fact , the request will be send to the owner (the cache which has the block in M state). As a result, it is impossible for the cache controller to get a Fwd_GetS request for a block in SI_A state.

**7. Was your verification testing exhaustive? How can you ensure that the random tester has exercised all possible transitions (i.e., all fState, Eventg pairs)?**

- Our verification testing was not exhaustive as it is very difficult to ensure that all transitions and states did actually happen . However, we were able to pass core configurations with 16 cores, the cache sizes are small and load count around of around million.

## Protocol Modifications with respect to Tables 8.1 and 8.2

MSI directory protocol for cache controller the states are the same as Table 8.1 in the the Primer on Memory Coherence and Consistency book. However, 5 new transient states have been added for implementing the protocol for directory controller which is presented in Table 1 below.



| | GETS | GETM | PutS_NotLast | PutS_Last | PutM+data from Owner | PutM+data from NonOwner | Data | Memory_Data | Memory_Ack |
|---|---|---|---|---|---|---|---|---|---|
| I | Request data from meory and add req to sharers/IS_MEM | Request data from memory, set owner to req/IM_MEM | Send Put_Ack to req | Send Put_Ack to req | 🟦 | send Put_Ack to req | 🟦 | 🟦 | pop the message |
| IS_MEM | stall | stall | Send Put_Ack to req | Send Put_Ack to req | 🟦 | send Put_Ack to req | 🟦 | send data to req/S | pop the message |
| IM_MEM | stall | stall | Send Put_Ack to req | Send Put_Ack to req | 🟦 | sned Put_Ack to req | 🟦 | send data to req/M | pop the message |
| S | Send data to req, add req to sharers | Send data to req, send Inv to sharers, set owner to req/M | Remove req from sharers, send Put_Ack to req | Send Put_Ack to req/I | 🟦 | Remove req from sharers, send Put_Ack to req | Pop the message | 🟦 | pop the message |
| M | Send Fwd_GetS to Owner, add Req and onwer to sharers, clear owner/MS_D | Send Fwd_GetM to owner, set owner to req | Send Put_Ack to req | Send Put_Ack to Req | Copy data to directory entry and memory, clear owner, send Put_ack to req/ MI-MEM | Send Put_Ack to Req | 🟦 | 🟦 | Memory_Ack |
| MS-D | **stall** | **stall** | **stall** | **stall** | 🟦 | **Copy data to directory entry and memory, sned a Put_Ack/MS_MEMD** | **Copy data to directory entry and memory/S** | | |
| MI_MEM | stall | stall | Send Put_Ack to req | send Put_Ack to req | 🟦 | Copy data to directory entry, send Put_Ack to req | 🟦 | 🟦 | Send a Put_Ack/I |
| MS_MEM | stall | stall | stall | stall | 🟦 | Pop the message | Copy data to directory entry | 🟦 | /S |
| MS_MEMD | stall | stall | stall | stall | directory entry and mem | | Copy data to directory entry and memory/MS_MEM | | /MS_D |

### Work Completed by Team Members

Mehrad Khalesi: writing code and testing, writing the report

Houman Haji: writing code and testing, writing the report