DECaxp Emulator Cache Coherency

Design Specification

Author: Jonathan D. Belanger  
Creation Date: April 7, 2018  
Modify Date: April 7, 2018

Table of Contents

[1 Overview 3](#_Toc510889907)

[2 Design Specifics 3](#_Toc510889908)

[2.1 Cache Block State 3](#_Toc510889909)

[2.2 Cache Block State Transitions 4](#_Toc510889910)

[2.3 CSRs Affecting Cache Coherency 4](#_Toc510889911)

[2.4 Commands sent from AXP CPU 6](#_Toc510889912)

[2.5 Commands sent to AXP CPU 7](#_Toc510889913)

[3 Uniprocessor Cache Coherency 8](#_Toc510889914)

Table and Figures

[Table 2‑1 AXP CPU Supported Cache States 3](#_Toc510889915)

[Table 2‑2 Cache Block State Transitions 4](#_Toc510889916)

[Table 2‑3 Cache Coherency CSRs 4](#_Toc510889917)

[Table 2‑4 AXP CPU to System Commands 6](#_Toc510889918)

[Table 2‑5 Probe Request Data Movement Commands 7](#_Toc510889919)

[Table 2‑6 Probe Request Next Cache State Commands 7](#_Toc510889920)

# Overview

Cache coherency is a necessity for Symmetric Multi-Processor (SMP) Systems. This is so that a memory location that is also in more than one cache, all caches need to agree. Otherwise, one processor accessing the same physical memory location could be utilizing different values. There are three basic cache coherency styles. They are:

1. Where all caches are maintained with the same value, whether read from or written to.
2. Where all caches maintain the same for reading, but the one that wants to be written to will invalidate the other caches values
3. Where all caches may be out of sync with one another (called Non-conforming).

This last option we are not going to utilize within our implementation. In the EV68CB/EV68DC Hardware Reference Manual (AXP HRM) in Section 4.5.1 Cache Coherency Basics, it states that this processor provides hardware mechanisms to support several cache coherency protocols. The protocols can be separated into two classes: write invalidate cache coherency protocol and flush cache coherency protocol.

The following tasks must be performed to maintain cache coherency:

* Istream data from memory spaces may be cached in the Icache and Bcache. Icache coherency is not maintained by hardware – it must be maintained by software using the CALL\_PAL IMB instruction.
* The AXP CPU maintains the Dcache as a subset of the Bcache. The Dcache is set-associative but is kept a subset of the larger externally implemented direct-mapped Bcache.
* System logic must help the AXP CPU to keep the Bcache coherent with main memory and other caches in the system.
* The AXP CPU requires the system to allow only one change to a block at a time. This means that if the AXP CPU gains the bus to read or write a block, no other node on the bus should be allowed to access that block until the data has been moved.

# Design Specifics

In this section we will document the various specifics of the AXP CPU that will be used throughout the remainder of this design specification. This includes information maintained within the caches and Control and Status Registers (CSRs).

## Cache Block State

The following states are possible for the caches within the AXP CPU:

Table ‑ AXP CPU Supported Cache States

| **State Name** | **Description** |
| --- | --- |
| Invalid | This AXP CPU does not have a copy of the block. |
| Clean | This AXP CPU holds a read-only copy of the block, and no other agent in the system holds a copy. Upon eviction, the block is not written to memory. |
| Clean/Shared | This AXP CPU holds a read-only copy of the block, and at least one other agent is the system may hold a copy of the block. Upon eviction, the block is not written to memory. |
| Dirty | This AXP CPU holds a read-write copy of the block, and no other agent in the system holds a copy. Upon eviction, the block must be written to memory. |
| Dirty/Shared | This AXP CPU holds a read-only copy of the dirty block, which may be shared with another agent. Upon eviction, the block must be written to memory. |

## Cache Block State Transitions

Cache block state transitions are reflected by the AXP CPU generated commands to the system. Cache block state transitions can also be caused by system-generated commands to the AXP CPU, via probes. Probes control the next state for the cache block. The next state can be based on the current state of the cache block. Table lists the next state for the cache block.

Table ‑ Cache Block State Transitions

| **Next State** | **Action Based on Probe Hit** |
| --- | --- |
| No change | Do not update the current state. Useful for DMA transitions that sample data but do not want to update tag state. |
| Clean | Independent of the current state, update the next state to Clean. |
| Clean/Shared | Independent of current state, update the next state to Clean/Shared. This transaction is useful for systems that update memory on probe hits. |
| T1:  Clean ⇒ Clean/Shared  Dirty ⇒ Dirty/Shared | Based on the dirty bit, update the next state to Clean/Shared or Dirty/Shared. This transaction is useful for systems that do not update memory on probe hits. |
| T3:  Clean ⇒ Clean/Shared  Dirty ⇒ Invalid  Dirty/Shared ⇒ Clean/Shared | If the cache block is Clean or Dirty, update the next state to Clean/Shared. If the cache block is Dirty, update the next state to Invalid. This transaction is useful for systems that use the Dirty/Shared state as an exclusive state. |

## CSRs Affecting Cache Coherency

The following CSRs in the AXP CPU affect how cache coherency is performed based on their settings. Table list the CSRs, their possible values and what those values represent.

Table ‑ Cache Coherency CSRs

| **CSR** | **Description** |
| --- | --- |
| BC\_CLEAN\_VICTIM | Enable CleanVictimBlk commands to the system interface. |
| BC\_RDVICTIM | Enable RdBlkVic, RdBlkNodVic, and InvalToDirtyVic commands to the system interface. |
| ENABLE\_EVICT | Enable issue Evict command for all ECB instructions. If this field is set, then the BC\_CLEAN\_VICTIM must also be set. |
| ENABLE\_STC\_COMMAND | Enable STx\_C instructions. Systems that require an explicit indication of ChangeToDirty status changes initiated by STx\_C instructions can assert Cbox CSR ENABLE\_STC\_COMMAND [0]. When this register field = 000, CleanToDirty and SharedToDirty commands are used. The distinction between a ChangeToDirty command generated by a STx\_C instruction and one generated by a STx instruction is important to systems that want to service ChangeToDirty commands with dirty data from a source processor. In this case, the distinction between a locked exclusive instruction and a normal instruction is critical to avoid livelock for a LDx\_L/STx\_C sequence.  **NOTE**: The AXP HRM sometimes has this as STC\_ENABLE. |
| INVAL\_TO\_DIRTY\_ENABLE | Enable WH64 functionality.   | **INVAL\_TO\_DIRTY\_ENABLE**  **[1:0]** | **Cbox Action** | | --- | --- | | x0 | WH64 instructions are converted to RdModx commands at the interface. Beyond this point, no other agent sees the WH64 instruction. This mode is useful for AXP CPUs that do not want to support InvalToDirty transactions. | | 01 | WH64 instructions are enabled, but they are acknowledged within the AXP CPU. | | 11 | WH64 instructions are enabled and generate InvalToDirty transactions off chip. | |
| PRB\_TAG\_ONLY | Enable probe-tag only mode. The AXP CPU expects to hit in cache on a probe response, so it always fetches a cache block from the Bcache on system probes. This can become a performance problem for systems that do not monitor the Bcache tags, so the EV68CB/EV68DC provides Cbox CSR PRB\_TAG\_ONLY[0], which only accesses Bcache tags for system probes. For a Bcache hit, the AXP CPU retries the probe reference to get the associated data. In this mode, the AXP CPU has a cache-hit counter that maintains some history of past cache hits in order to fetch the data with the tag in the cases where streamed transactions are being performed to the host processor. |
| RDVIC\_ACK\_INHIBIT | Enable inhibition of incrementing acknowledge counter for RdBlkVic, RdBlkNodVic, and InvalToDirtyVic commands. |
| SET\_DIRTY\_ENABLE | SetDirty Acknowledge.   | **SET\_DIRTY\_ENABLE**  **[2:0]** | **Cbox Action** | | --- | --- | | 000 | Everything acknowledged internally (uniprocessor). | | 001 | Only clean blocks generate external. acknowledge (CleanToDirty commands only). | | 010 | Only clean/shared blocks generate external acknowledge (SharedToDirty command only) | | 011 | Clean and clean/shared blocks generate external acknowledge | | 100 | Only dirty/shared blocks generate external acknowledge (SharedToDirty commands only) | | 101 | Only dirty/shared and clean blocks generate external acknowledge. | | 110 | Only dirty/shared and clean/shared blocks generate external acknowledge. | | 111 | All transactions generate external acknowledge. | |
| SYSBUS\_MB\_ENABLE | Enable MB commands off chip. See AXP RTM Section 2.12.2, Memory Barrier (MB/WMB/TB Fill Flow). |

## Commands sent from AXP CPU

There are quite a few commands that can be sent by the AXP CPU to the system to request some off chip resource (memory, storage, or cache coherency information). lists all the commands that can be sent to the System from the AXP CPU.

Table ‑ AXP CPU to System Commands

| **Command** | **Function** |
| --- | --- |
| NOP | The AXP CPU drives this command on idle cycles during a reset. Once the first NZOP is generated, this command is no longer generated. |
| ProbeResponse | Returns the probe status and ID number of the VDB entry holding the requested cache block. |
| NZNOP | This nonzero NOP helps to parse the command packet. |
| VDBFlushRequest | VDB flush request. The AXP CPU sending this command to the system when an internally generated transaction Bcache index matches a Bcache victim or probe in the VDB. The system should flush all VDB entries associated with all outstanding probe and WrVictimBlk transactions that where queued up prior to this request. |
| MB | Indicates that an MB instruction was issued. |
| ReadBlk | Memory read request. Usually as the result of an LDx instruction. |
| ReadBlkMod | Memory read request with modify intent. Usually as the result of a STx instruction. |
| ReadBlkI | Memory read request for the Instruction Stream (Istream). This is internally generated when the AXP CPU attempt to parse the next instruction for execution and misses in the Icache. |
| FetchBlk | Noncached memory read request. |
| ReadBlkSpec | Speculative memory read request. |
| ReadBlkModSpec | Speculative memory read request with modify intent. |
| ReadBlkSpecI | Speculative memory read request for Istream. |
| FetchBlkSpec | Speculative noncached memory read request. |
| ReadBlkVic | Memory read request with a victim. |
| ReadBlkModVic | Memory read request with modify intent, with a victim. |
| ReadBLKVicI | Memory read request for Istream with victim. |
| WrVictimBlk | Write-back of dirty block. Sent when a dirty cache block is evicted. |
| CleanVictimBlk | Supply address of a clean victim. Sent when a clean cache block is evicted. |
| Evict | Invalidate evicted block at the given Bcache index. |
| ReadBytes | I/O read request. Mask indicates which bytes of the quadword are valid. |
| ReadLWs | I/O read request. Mask indicates which longwords of 32-byte block are valid. |
| ReadQWs | I/O read request. Mask indicates which quadwords of the 64-byte block are valid. |
| WrBytes | I/O write request. Mask indicates which bytes of the quadword are valid. |
| WrLWs | I/O write request. Mask indicates which longwords of 32-byte block are valid. |
| WrQWs | I/O write request. Mask indicates which quadwords of the 64-byte block are valid. |
| CleanToDirty | Sets a cache block to a Dirty state, but only if it is currently Clean. This is used when duplicate tags have been enabled. |
| SharedToDirty | Sets a cache block to a Dirty state, but only if it is currently in a Shared state. This is used for multiprocessor systems. |
| STCChangeToDirty | Sets a cache block to a Dirty state that was previously Clean or Shared for an STx\_C instruction. |
| InvalToDirtyVic | Invalid to Dirty state with a victim. |
| InvalToDirty | WH64 acts like a ReadBlkMod without the fill cycles. |

## Commands sent to AXP CPU

The following command are send from the System to the CPU for processing in the Cbox. The Cbox utilizes the Bcache and it’s Duplicate Tag (DTAG) array to respond back to the system. The commands sent by the system are broken up into two components. The first component is for a data movement request (see ). The second component is for a next cache state request.

Table ‑ Probe Request Data Movement Commands

|  |  |
| --- | --- |
| **Data Movement Commands** | **Data Movement Function** |
| NOP | No operation. |
| ReadHit | Read if hit. Return the data back to the system if block is valid. No other state matters. |
| ReadDirty | Read if dirty. Return the data back to the system if the block is valid and dirty. For both Dirty and Dirty/Shared cache blocks. |
| ReadAlways | Read anyway. Return the data at the probe index back to the system. State of the block is irrelevant. |

Table ‑ Probe Request Next Cache State Commands

| **Next Cache State Commands** | **Next Cache State** |
| --- | --- |
| NOP | No state changed. |
| Clean | State changed to Clean. |
| Clean/Shared | State changed to Clean/Shared. |
| Transition 3[[1]](#footnote-1) | Clean ⇒ Clean/Shared  Dirty ⇒ Invalid  Dirty/Shared ⇒ Clean/Shared |
| Dirty/Shared |  |
| Invalid | Not used. |
| Transition 1[[2]](#footnote-2) | Clean ⇒ Clean/Shared  Dirty ⇒ Dirty/Shared |
| Reserved | Not used. |

# Uniprocessor Cache Coherency

For uniprocessors, cache coherency is the most straightforward. There is no need to utilize the Clean/Shared and Dirty/Shared states, as there is only a single cache and no coherency requirements.

Figure ‑ Uniprocessor Cache State Transitions

4

1

2

5

3

6

1. When a Rdx is performed, memory is read and stored into the cache in a Clean state (read-only).
2. When a RdModx is performed, memory is read and stored into the cache in a Dirty state (read-write).
3. When a previously Rdx block is changed to a read-write block.
4. When a previously Rdx block is evicted from the cache.
5. When a previously RdModx or read-write block is evicted and written out to memory.
6. This transition is not necessary in a uniprocessor system.

1. Transition 3 is useful in non-duplicate tag systems that want to give writeable status to the reader and do not know if the block is clean or dirty. [↑](#footnote-ref-1)
2. Transition 1 is useful in non-duplicating tag systems that do not update memory on ReadBlk hits to a dirty block in another processor. [↑](#footnote-ref-2)