DECaxp 21264 Emulator Cache Coherency

Design Specification

Author: Jonathan D. Belanger  
Creation Date: April 7, 2018  
Modify Date: April 21, 2018

**April 18**

The information in this publication is subject to change without notice.

JONATHAN D. BELANGER SHALL NOT BE LIABLE FOR TECHNICAL OR EDITORIAL ERRORS OR OMISSIONS CONTAINED HEREIN, NOR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL. THIS INFORMATION IS PROVIDED "AS IS" AND JONATHAN D. BELANGER DISCLAIMS ANY WARRANTIES, EXPRESS, IMPLIED OR STATUTORY AND EXPRESSLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, GOOD TITLE AND AGAINST INFRINGEMENT.

This publication contains information protected by copyright. No part of this publication may be photocopied or reproduced in any form without prior written consent from Jonathan D. Belanger.

©21 April 2018 Jonathan D. Belanger. All rights reserved. Printed in U.S.A.

COMPAQ, DIGITAL, DIGITAL UNIX, OpenVMS, VAX, VMS, and the  logo registered in United States Patent and Trademark Office.

GRAFOIL is a registered trademark of Union Carbide Corporation.  
IEEE is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.  
Windows NT is a trademark of Microsoft Corporation.

Other product names mentioned herein may be trademarks and/or registered trademarks of their respective companies.

Table of Contents

[1 Preface 5](#_Toc511488740)

[1.1 Audience 5](#_Toc511488741)

[1.2 Terminology and Conventions 5](#_Toc511488742)

[2 Overview 10](#_Toc511488743)

[3 Design Constraints 11](#_Toc511488744)

[3.1 Cache Block State 11](#_Toc511488745)

[3.2 Cache Block State Transitions 11](#_Toc511488746)

[3.3 CSRs Affecting Cache Coherency 12](#_Toc511488747)

[3.4 Commands sent from AXP CPU 13](#_Toc511488748)

[3.5 Commands sent to AXP CPU 14](#_Toc511488749)

[4 Uniprocessor Cache Coherency 16](#_Toc511488750)

[4.1 Instructions, Cache States, and System Messages 17](#_Toc511488751)

[4.2 CSR and IPR Settings to Support a Uniprocessor Configuration 17](#_Toc511488752)

[5 Multiprocessor Cache Coherency 19](#_Toc511488753)

[5.1 Modified-Shared-Invalid (MSI) Cache Coherency Protocol 19](#_Toc511488754)

[5.1.1 Instructions, Cache States, and System Messages for MSI Cache Coherency 21](#_Toc511488755)

[5.1.2 CSR and IPR Settings to Support a Multiprocessor Configuration for MSI 22](#_Toc511488756)

[5.2 MESI Cache Coherency Protocol 24](#_Toc511488757)

[5.3 MOSI Cache Coherency Protocol 24](#_Toc511488758)

[5.4 MOESI Cache Coherency Protocol 24](#_Toc511488759)

[5.5 MERSI Cache Coherency Protocol 24](#_Toc511488760)

[5.6 MESIF Cache Coherency Protocol 24](#_Toc511488761)

[5.7 Write-Once Cache Coherency Protocol 24](#_Toc511488762)

[5.8 Synapse Cache Coherency Protocol 24](#_Toc511488763)

[5.9 Berkeley Cache Coherency Protocol 24](#_Toc511488764)

[5.10 Firefly Cache Coherency Protocol 24](#_Toc511488765)

[5.11 Dragon Cache Coherency Protocol 24](#_Toc511488766)

Tables

[Table 3‑1 AXP CPU Supported Cache States 11](#_Toc511488619)

[Table 3‑2 Cache Block State Transitions 11](#_Toc511488620)

[Table 3‑3 Cache Coherency CSRs 12](#_Toc511488621)

[Table 3‑4 AXP CPU to System Commands 13](#_Toc511488622)

[Table 3‑5 Probe Request Data Movement Commands 14](#_Toc511488623)

[Table 3‑6 Probe Request Next Cache State Commands 15](#_Toc511488624)

[Table 3‑7 System Data Control (SysDc) Commands 15](#_Toc511488625)

[Table 4‑1 IPR and CSR Uniprocessor Settings 17](#_Toc511488626)

[Table 5‑1 MSI to AXP CPU Cache State Mapping 19](#_Toc511488627)

[Table 5‑2 IPR and CSR Uniprocessor Settings 23](#_Toc511488628)

Figures

[Figure 4‑1 Uniprocessor Cache State Transitions 16](#_Toc511488658)

[Figure 4‑2 Processing Flows 17](#_Toc511488659)

[Figure 5‑1 Cache Coherency for Multiprocessors 19](#_Toc511488660)

[Figure 5‑2 MSI State Diagram 20](#_Toc511488661)

[Figure 5‑3 Processing Flows 21](#_Toc511488662)

# Preface

## Audience

This document is for the designers and programmers who plan to code or update the DECaxp 21264 Emulator source code.

## Terminology and Conventions

This section defines the abbreviations, terminology, and other conventions used throughout this document.

**Abbreviations**

* Binary Multiples

The abbreviations K, M, and G (kilo, mega, and giga) represent binary multiples and have the following values.

For example:

* Sign extension

SEXT(*x*) means *x* is sign-extended to the required size.

* Register Access  
    
  The abbreviations used to indicate the type of access to register fields and bits have the following definitions:

| **Abbreviation** | **Meaning** |
| --- | --- |
| IGN | Ignore. Bits and fields specified are ignored on writes. |
| MBZ | Must Be Zero. Software must never place a nonzero value in bits specified as MBZ. A nonzero read produces an Illegal Operation exception. Also, MBZ fields are reserved for future use. |
| RAZ | Read As Zero. Bits and fields return a zero when read. |
| RC | Read Clears. Bits and fields are cleared when read. Unless otherwise specified, such bits cannot be written. |
| RES | Reserved. Bits and fields are reserved and should not be used. However, zeros can be written to reserved fields that cannot be masked. |
| RO | Read Only. The value may be read by software. It is written by the emulation code. Software write operations are ignored. |
| RO, *n* | Read Only, and takes the value *n* at power-on reset. The value may be read by software. It is written by the emulation code. Software write operations are ignored. |
| RW | Read Write. Bits and fields can be read and written. |
| RW, *n* | Read Write and takes the value *n* at power-on reset. Bits and fields can be read and written. |
| W1C |  |
| W1S |  |
| WO | Write Only. Bits and fields can be written but not read. |
| WO, *n* | Write Only, and takes the value *n* at power-on reset. Bits and fields can be written but not read. |

**Addresses**

Unless otherwise noted, all addresses and offsets are hexadecimal.

**Aligned and Unaligned**

The terms aligned and naturally aligned are interchangeable and refer to data objects that are powers of two in size. An aligned datum of size 2n is stored in memory at a byte address that is a multiple of 2n; that is, one that has n low-order zeros. For example, an aligned 64-byte stack frame has a memory address that is a multiple of 64.

A datum of size 2n is unaligned if it is stored in a byte address that is not a multiple of 2n.

**Bit Notation**

Multiple-bit fields can include contiguous and noncontiguous bits contained in square brackets ([]). Multiple contiguous bits are indicated by a pair of numbers separated by a colon [:]. For example, [9:7,5,2:0] specifies bits 9,8,7,5,2,1, and 0. Similarly, single bits are frequently indicated with square brackets. For example, [27] specifies bit 27. See also Field Notation.

**Data Units**

The following data unit terminology is used throughout this manual.

| **Term** | **Words** | **Bytes** | **Bits** | **Other** |
| --- | --- | --- | --- | --- |
| Byte | ½ | 1 | 8 | --- |
| Word | 1 | 2 | 16 | --- |
| Longword | 2 | 4 | 32 | Dword |
| Quadword | 4 | 8 | 64 | 2 longwords |

**Do Not Care (X)**

A capital X represents any valid value.

**External**

Unless otherwise stated, external means not contained in the chip.

**Field Notation**

The names of single-bit and multiple-bit fields can be used rather than the actual bit numbers (see Bit Notation). When the field name is used, it is contained in square brackets ([]). For example, **RegisterName[LowByte]** specifies **RegisterName[7:0]**.

**Note**

Notes emphasize particularly important information.

**Numbering**

All numbers are decimal or hexadecimal unless otherwise indicated. The prefix 0x indicates a hexadecimal number. For example, 19 is decimal, but 0x19 and 0x19a are hexadecimal (also see Addresses). Otherwise, the base is indicated by a subscript; for example, 1002 is a binary number.

**Ranges and Extents**

*Ranges* are specified by a pair of numbers separated by two periods (..) and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

*Extents* are specified by a pair of numbers in square brackets ([]) separated by a colon (:) and are inclusive. Bit fields are often specified as extents. For example, bits [7:3] specifies bits 7, 6, 5, 4, and 3.

**Register Figures**

The gray areas in register figures indicate reserved or unused bits and fields.

Bit ranges that are coupled with the field name specify the bits of the named field that are included in the register. The bit range may, but need not necessarily, correspond to the bit Extent in the register. See the explanation above Table 5–1 for more information.

**Signal Names**

The following examples describe signal-name conventions used in this document.

**AlphaSignal[n:n]** Boldface, mixed-case type denotes signal names that are assigned internal and external to the EV68CB/EV68DC (that is, the signal traverses a chip interface pad).

**AlphaSignal\_x[n:n]** When a signal has high and low assertion states, a lowercase italic x represents the assertion states. For example, **SignalName\_*x*[3:0]** represents **SignalName\_H[3:0]** and **SignalName\_L[3:0]**.

**UNDEFINED**

Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing to stopping system operation.

UNDEFINED operations may halt the processor or cause it to lose information. However, UNDEFINED operations must not cause the processor to hang, that is, reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions.

**UNPREDICTABLE**

UNPREDICTABLE results or occurrences do not disrupt the basic operation of the processor;

it continues to execute instructions in its normal manner. Further:

* Results or occurrences specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.
* An UNPREDICTABLE result may acquire an arbitrary value subject to a few constraints. Such a result may be an arbitrary function of the input operands or of any state information that is accessible to the process in its current access mode. UNPREDICTABLE results may be unchanged from their previous values.  
    
  Operations that produce UNPREDICTABLE results may also produce exceptions.
* An occurrence specified as UNPREDICTABLE may happen or not based on an arbitrary choice function. The choice function is subject to the same constraints as are UNPREDICTABLE results and, in particular, must not constitute a security hole.  
    
  Specifically, UNPREDICTABLE results must not depend upon, or be a function of, the contents of memory locations or registers that are inaccessible to the current process in the current access mode.  
    
  Also, operations that may produce UNPREDICTABLE results must not:
* Write or modify the contents of memory locations or registers to which the current process in the current access mode does not have access, or
* Halt or hang the system or any of its components.

For example, a security hole would exist if some UNPREDICTABLE result depended on the value of a register in another process, on the contents of processor temporary registers left behind by some previously running process, or on a sequence of actions of different processes.

**X**

Do not care. A capital X represents any valid value.

# 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 Constraints

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 its 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 | State changed to Dirty/Shared |
| Invalid | State changed to Invalid |
| Transition 1[[2]](#footnote-2) | Clean ⇒ Clean/Shared  Dirty ⇒ Dirty/Shared |
| Reserved | Not used. |

Table ‑ System Data Control (SysDc) Commands

| **SysDc Command** | **Description** |
| --- | --- |
| NOP | NOP, SysData is ignored by the AXP CPU. |
| ReadDataError | Data is returned for read commands. The system sends SysData, I/O, or memory NXM |
| ChangeToDirtySuccess | No data. SysData is ignored by the AXP CPU. This command is also used for the InvalToDirty response |
| ChangeToDirtyFail | No data. SysData is ignored by the AXP CPU. This command is also used for the Evict response. |
| MBDone | Memory barrier operation completed. |
| ReleaseBuffer | Command to alert the AXP CPU that the RVB, RPB, and ID field are valid. |
| ReadData | Data returned for read commands. The system returns SysData. The cache status is set to Clean. The system uses the lower 2 bits of the command to control wrap order. |
| ReadDataDirty | Data is returned for Rdx and RdModx commands. The cache status is set to Dirty. The system uses the lower 2 bits of the command to control wrap order. |
| ReadDataShared | Data returned for read commands. The system returns SysData. The cache status is set to Clean/Shared. The system uses the lower 2 bits of the command to control wrap order. |
| ReadDataShareDirty | Data is returned for RdBlk commands. The cache status is set to Shared/Dirty. The system uses the lower 2 bits of the command to control wrap order. |
| WriteData | Data is sent for AXP CPU write commands or system probes. The AXP CPU sends data to the System. The AXP CPU uses the lower 2 bits of the command to control the wrote order. |

# 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.

## Instructions, Cache States, and System Messages

This section describes each of the instructions that generate messages to and gets responses from the System. The processing for this functionality, is controlled by CSRs and IPRs. Please see Section 4.2 for more information about these settings.

Figure ‑ Processing Flows

| **Step** | **Instruction** | **Cache State** | **Interface** | **System** | **Memory Access** |
| --- | --- | --- | --- | --- | --- |
| L1.1 | LDx ⇒ | Invalid ⇒ | ReadBlk ⇒ | Read Memory ⇒ |  |
| L1.2 | ⇐ LDx | ⇐ Clean | ⇐ ReadData | ⇐ Return Data | ⇐ Return Data |
| L2.1 | LDx ⇒ | Clean/Hit ⇓ |  |  |  |
| L2.2 | ⇐ LDx | ⇐ Clean |  |  |  |
| L3.1 | LDx ⇒ | Clean/Miss ⇓ |  |  |  |
| L3.2 |  | Evict ⇒ | ReadBlk ⇒ | Read Memory ⇒ |  |
| L3.3 | ⇐ LDx | ⇐ Clean | ⇐ ReadData | ⇐ Return Data | ⇐ Return Data |
| L4.1 | LDx ⇒ | Dirty/Hit ⇓ |  |  |  |
| L4.2 | ⇐ LDx | ⇐ Dirty |  |  |  |
| L5.1 | LDx ⇒ | Dirty/Miss ⇓ |  |  |  |
| L5.2 |  | Evict ⇒ | WrVictimBlk ⇒ | Write Memory ⇒ |  |
| L5.3 |  |  | ⇐ WriteData | ⇐ Write Memory | ⇐ Write Memory |
| L5.4 |  |  | ReadBlk ⇒ | Read Memory ⇒ |  |
| L5.5 | ⇐ LDx | ⇐ Clean | ⇐ ReadData | ⇐ Return Data | ⇐ Return Data |
| S1.1 | STx ⇒ | Invalid ⇒ | ReadBlkMod ⇒ | Read Memory ⇒ |  |
| S1.2 | ⇐ STx | ⇐ Dirty | ⇐ ReadDataDirty | ⇐ Return Data | ⇐ Return Data |
| S2.1 | STx ⇒ | Clean/Hit ⇓ |  |  |  |
| S2.2 |  | CleanToDirty ⇓ |  |  |  |
| S2.3 | ⇐ STx | ⇐ Dirty |  |  |  |
| S3.1 | STx ⇒ | Clean/Miss ⇓ |  |  |  |
| S3.2 |  | Evict ⇒ | ReadBlkMod ⇒ | Read Memory ⇒ |  |
| S3.3 | ⇐ STx | ⇐ Dirty | ⇐ ReadDataDirty | ⇐ Return Data | ⇐ Return Data |
| S4.1 | STx ⇒ | Dirty/Hit ⇓ |  |  |  |
| S4.2 | ⇐ STx | ⇐ Dirty |  |  |  |
| S5.1 | STx ⇒ | Dirty/Miss ⇓ |  |  |  |
| S5.2 |  | Evict ⇒ | WrVictimBlk ⇒ | Write Memory ⇒ |  |
| S5.3 |  |  | ⇐ WriteData | ⇐ Write Memory | ⇐ Write Memory |
| S5.4 |  |  | ReadBlkMod ⇒ | Read Memory ⇒ |  |
| S5.5 | ⇐ STx | ⇐ Dirty | ⇐ ReadData | ⇐ Return Data | ⇐ Return Data |
| M1.1 | MB ⇒ |  |  |  |  |
| M1.2 | ⇐ MB |  |  |  |  |

## CSR and IPR Settings to Support a Uniprocessor Configuration

Table 4‑1 lists the Internal Processor Register (IPR) and Control and Status Register (CSR) settings to support uniprocessing. These settings are no necessarily required. If the multiprocessor settings are used, there will be unnecessary communication between the AXP CPU and System. Since the System knows how many processors are present, when there is just one, sending Cache Status and other requests that do not actually request data movement, are not necessary.

Table ‑ IPR and CSR Uniprocessor Settings

| **IPR or CSR** | **Setting** | **Description** |
| --- | --- | --- |
| I\_CTL[TB\_MB\_MEM] | 0 | Deasserting this field in the Ibox IPR will disable inserting an MB instruction within the TB fill flow. |
| SYSBUS\_MB\_ENABLE | 0 | Deasserting this CSR will internally acknowledge MB commands/ instructions. |
| SET\_DIRTY\_ENABLE | 000 | Deasserting this CSR will internally acknowledge SetDirty, SharedToDirty or CleanToDirty, requests (called a SetModify). |
| ENABLE\_STC\_COMMAND | 0 | Deasserting this CSR will internally acknowledge a SetDirty request for an STx\_C instruction. Deasserting means that rather than sending a STCChangeToDirty command, a SharedToDirty or CleanToDirty command would be send. Deasserting the SET\_DIRTY\_ENABLE CSR, as well, will internally acknowledge these. |
| INVAL\_TO\_DIRTY | x0 | Deasserting this CSR will internally acknowledge InvalToDirty requests. |
| ENABLE\_EVICT | 0 | Deasserting this CSR will cause the AXP CPU to not send a command to the System to indicate that an evict is being performed. |
| BC\_CLEAN\_VICTIM | 0 | This must also be deasserted when the ENABLE\_EVICT is also deasserted. |

# Multiprocessor Cache Coherency

For multiprocessors, cache coherency is the most difficult. In addition to the states in the uniprocessor case, we also introduce a Clean/Shared and a Dirty/Shared Cache State. Of these, the Clean/Shared is the simpler of the two. Depending upon how the IPRs and CSRs are set, it is possible to support more than one cache coherency protocol. The cache coherency protocols are defined in the next sections. The following figure shows the components involved in cache coherency

Figure ‑ Cache Coherency for Multiprocessors

S  
y  
s  
t  
e  
m

Memory

CPU

Cache

CPU

Cache

Coherency

## Dragon Cache Coherency Protocol

The Dragon Protocol is an update-based cache coherence protocol used in multiprocessor systems.

### Dragon Protocol Overview

Write propagation is performed by directly updating all the cached values across multiple processors. Update based protocols such as the Dragon protocol perform efficiently when a write to a cache block is followed by several reads made by other processors, since the updated cache block is readily available across caches associated with all the processors.

### Dragon Protocol Cache Block States

In this protocol, each block contained within a cache can have one of five states:

|  |  |
| --- | --- |
| * **Invalid**: | This is the initial state of any cache block. This is like the MSI, MESI, MOESI, MERSI and MESIF Invalidating protocols. It is unlike these protocols, in that transitions into this state is only performed when explicitly requested or the cache is flushed. |
| * **Clean**: | This is an exclusive read-only state. This state is entered under the following conditions:   1. A read on a block not in the cache (local miss) and no other processor contains a copy. 2. A read by the current processor on a block already in this state will remain in this state. |
| * **Dirty**: | This is an exclusive read-write state. This state is entered under the following conditions:   1. A write on a block not in the cache (local miss) and no other processor contains a copy. 2. A read or write by the current processor on a block already in this state will remain in this state.   Evicting a block in this state or transitioning it to another will cause it to be written to memory. |
| * **Shared/Clean**: | This is a non-exclusive read-only state. This state is entered under the following conditions:   1. A read on a block not in the cache (local miss), and one or more other processors contain a copy in any state. 2. Another processor requests a read or write on a block that is in a Clean, Dirty or Shared/Dirty state in this processor. |
| * **Shared/Dirty**: | This is a semi-exclusive read-write state. This state is entered under the following conditions:   1. A write on a block that is in the current processor (local hit) that is also is a Shared/Clean or Shared/Dirty state in other processors. 2. A read or write by the current processor on a block already in this state will remain in this state.   Evicting a block in this state or transitioning it to another will cause it to be written to memory.  Updating the contents of this block will update the contents in other processors |

In the AXP CPU cache implementation there is a 1-to-1 mapping of the above states to cache states.

### Dragon Protocol State Transitions

Figure 5‑5 diagrams the various states along with the possible state transitions. Unlike other documents showing the Dragon State Transition Diagram, I have introduced the Invalid state. This is the initial state and the state a cache block goes into when it is flushed. The numbers in the numbered list below the figure correspond to the numbers in the dashed blocks in the figure. This specification documents the characteristics of what happens for these state transmissions to occur.

Also, it is not clear if the local processor maintains the Shared/Dirty state, even after it has updated the block and let the other processors know of the update. Since memory is only written to upon eviction, this implementation will maintain the Shared/Dirty state either until it is evicted from the cache or another processor requests to write to the block. The real pain will be when more than one processor is writing to different locations within the same block. The following serialized steps will need to be performed:

1. Two or more processors are sharing a cache block, and they are both in Shared/Clean state.
2. Both processors are executing a STx instruction to different locations within the same block.
3. Both processors send a request to be able to write to the block.
4. The system processes them one at a time.
5. One processor’s request is processed first, and it is granted the Shared/Dirty state. It also locks it block, so that it cannot be evicted or changed.
6. The second processor’s request is now processed and sent to the first processor.
7. Because of the lock, the local processor does not respond to the request until updating the other cache with the new value (at the time of instruction retirement).
8. The response to the second processor’s request goes out with the data in the previous step and causes the local processor to transition the local cache block to Shared/Clean. The second processor transitions its cache block to Shared/Dirty.

Now, it we have a poorly designed loop, the above steps could be repeated many times causing the state of the cache block to thrash to and from the Shared/Clean from and to the Shared/Dirty states. Just something to think about.

The above also holds true for the Shared/Clean state. If a block is correctly in the Shared/Clean state because more than one processor has the same block for read-only, and then the other processors evict the block, the local processor does not necessarily know that it has the only copy of the block. A block in the Shared/Clean state does not go back to the Clean state in the Dragon Cache Coherency protocol.

Figure ‑ Dragon State Diagram

Initialize/Flush

1

2

3

4

5

6

7

8

9

10

11

12

13

14

1. Transition from **Dirty to Shared/Dirty** and **Shared/Dirty to Dirty**:
   1. **Dirty 🡪 Shared/Dirty**:
      1. Another processor indicates that it wants to read the block.
      2. The local processor has it in a **Dirty** state and the value in memory may be stale.
      3. The local processor provides the value to the other processor and transitions its local state to **Dirty/Shared**.
      4. The other processor will receive the data and set its cache block state to Shared/Clean. The value in memory may or may not be stale.
      5. The local processor has not executed an instruction for this transition to occur.
      6. Upon eviction, the local processor updates memory.
   2. **Dirty/Shared 🡪 Dirty**:
      1. The local processor is executing a STx[\_C] instruction.
      2. At retirement it updates the value in its cache.
      3. Because the cache block state is **Share/Dirty**, the processor sends the update so that the other processors have the updated value.
      4. The return from the system indicates that the block is no longer shared with any other processor and transitions the state to **Dirty**.
      5. Upon eviction, the local processor updates memory.
2. Transition from **Shared/Clean to Shared/Dirty** and **Shared/Dirty to Shared/Clean**:
   1. **Shared/Clean 🡪 Shared/Dirty**:
      1. The local processor is executing a STx[\_C] instruction.
      2. The local processor sends a request to the other processors indicating the it wants to put the cache block into a writeable (**Dirty** or **Shared/Dirty**) state.
      3. If another processor had the block in a **Shared/Dirty** state, the other processor updated memory and sent the value to the local processor. This other processor also put its copy of the cache block into a **Shared/Clean** state.
      4. Whether the other processors had the block in a **Shared/Dirty** or **Shared/Clean** state, the local processor transitions the local cache block to the **Shared/Dirty** state.
      5. At retirement it updates the value in its cache.
      6. Because the cache block state is bow **Share/Dirty**, the processor sends the update so that the other processors have the updated value.
      7. Upon eviction, the local processor updates memory.
   2. **Shared/Dirty 🡪 Shared/Clean**:
      1. Another processor indicates that it wants to write to the block.
      2. Since the local processor has it in a **Shared/Dirty** state, it updates memory and transitions the block to the **Shared/Clean** state.
      3. The local processor has not executed an instruction for this transition to occur.
      4. Upon eviction, the local processor **does not** update memory.
3. Transition from **Shared/Clean 🡪 Dirty**:
   1. The local processor is executing a STx[\_C] instruction.
   2. The local processor sends a request to the other processors indicating that it want to put the cache block into a writeable (**Dirty** or **Shared/Dirty**) state.
   3. No other processor has the block and the system responds with a Dirty indication (not a Shared/Dirty indication).
   4. The local processor transitions the block to the **Dirty** state.
   5. At retirement it updates the value in its cache.
   6. Upon eviction, the local processor updates memory.
4. Transition from **Clean 🡪 Shared/Clean**:
   1. Another processor indicates that it wants to read the block.
   2. Since the local processor has it in a **Clean** state, it provides it for the other processor and transitions the state to **Shared/Clean**.
   3. The local processor has not executed an instruction for this transition to occur.
   4. Upon eviction, the local processor **does not** update memory.
5. Transition from **Clean 🡪 Dirty**:
   1. The local processor is executing a STx[\_C] instruction.
   2. The local processor, since it has exclusive ownership, transitions the state from **Clean** to **Dirty**.
   3. At retirement it updates the value in its cache.
   4. Upon eviction, the local processor updates memory.
6. Transition from **Invalid 🡪 Clean**:
   1. The local processor is executing a LDx[\_L] and misses in the cache.
   2. The local processor sends a request to system to read the block from memory.
   3. The system did not find the block in any other processor, so gets the block from memory and returns it to the local processor.
   4. The local processor sets the cache blocks state to **Clean**.
   5. Upon eviction, the local processor **does not** update memory.
7. Transition from **Invalid 🡪 Dirty**:
   1. The local processor is executing a STx[\_C] and misses in the cache.
   2. The local processor sends a request to system to read-to-modify the block from memory.
   3. The system did not find the block in any other processor, so gets the block from memory and returns it to the local processor.
   4. The local processor sets the cache blocks state to **Dirty**.
   5. Upon eviction, the local processor updates memory.
8. Transition from **Invalid 🡪 Shared/Clean**:
   1. The local processor is executing a LDx[\_L] and misses in the cache.
   2. The local processor sends a request to system to read the block from memory.
   3. The system did find the block in another processor, so returns the block from the other processor to the local processor.
   4. The local processor sets the cache blocks state to **Shared/Clean**.
   5. Upon eviction, the local processor **does not** update memory.
9. Transition from **Invalid 🡪 Shared/Dirty**:
   1. The local processor is executing a STx[\_C] and misses in the cache.
   2. The local processor sends a request to system to read-to-modify the block from memory.
   3. The system did find the block in another processor, so returns the block from the other processor to the local processor.
   4. The local processor sets the cache blocks state to **Shared/Dirty**.
   5. Upon eviction, the local processor updates memory.
10. Staying in the **Clean** state:
    1. The local processor is executing a LDx[\_L] instruction and hits in the cache.
    2. Upon eviction, the local processor **does not** update memory.
11. Staying in the **Dirty** state:
    1. The local processor is executing a LDx[\_L] or STx[\_C] instruction and hits in the cache.
    2. Upon eviction, the local processor updates memory.
12. Staying in the **Shared/Dirty** state:
    1. The local processor is executing a LDx[\_L] or STx[\_C] instruction and hits in the cache.
    2. Upon eviction, the local processor updates memory.
13. Staying in the **Shared/Clean** state:
    1. The local processor is executing a LDx[\_L] instruction and hits in the cache.
    2. Upon eviction, the local processor **does not** update memory.
14. Initializing or transitioning into the **Invalid** state:
    1. When the processor is initialized, all cache entries are marked in the **Invalid** state.
    2. When a cache block, a set of cache blocks, or all cache blocks are flushed, the affected cache blocks are set to the **Invalid** state.

### Dragon Protocol Support in the Alpha AXP 21264 and DECchip 21272/21274

Sections 3.4 and 3.5 detail the messages that pass between the AXP CPU and the System. These messages are utilized to support the Dragon Cache Coherency Protocol. This section will document the various instructions, local and remote cache states, and the messages uses to support the protocol.

#### Alpha AXP Instructions Initiating Dragon Protocol Messages

Most of the Alpha AXP instructions do not actually cause a message to be sent to the system. Additionally, instructions in another processor (remote) may cause a cache state transition in this processor (local) with messages received by the local processor from the system on behalf of one or more remote processors.

We will not concern ourselves with the instructions that do not cause a message to be sent to the system.

The following Alpha AXP instructions may cause a cache state transition. A state transition does not need to occur, depending upon the current state of the cache block. If there is a state transition, this transition may be local, remote, or both.

Table ‑ Instructions 🡪 States 🡪 Caches

| **Mnemonic** | **Instructions** | **Description** | **State Transitions** | **Local, Remote, Both** |
| --- | --- | --- | --- | --- |
| LDx | LDBU  LDL  LDL\_L  LDQ  LDQ\_L  LDWU  LDF  LDG  LDS  LDT  FETCH  PREFETCH  PREFETCH\_EN | These instructions request memory data to be loaded into a register. | Invalid 🡪 Clean  Invalid 🡪 Shared/Clean  Clean 🡪 Clean  Clean 🡪 Shared/Clean  Shared/Clean 🡪 Shared/Clean  Dirty 🡪 Dirty  Shared/Dirty 🡪 Shared/Dirty | Local  Both  Local  Both  Local  Local  Local |
| STx | STB  STL  STL\_C  STQ  STQ\_C  STQ\_U  STW  STF  STG  STS  STT  FETCH\_M  PREFETCH\_M  PREFETCH\_MEN | These instructions update the local, and potentially remote, cache when they are retired. They do not update memory but may update the remote cache. Memory will be updated when the cache block is evicted. | Invalid 🡪 Dirty  Invalid 🡪 Shared/Dirty  Clean 🡪 Dirty  Shared/Clean 🡪 Shared/Dirty  Shared/Clean 🡪 Dirty | Local  Both[[3]](#footnote-3)  Local  Both[[4]](#footnote-4)  Local |
| ECB | ECB | This instruction indicates that a cache block should be evicted. | Clean 🡪 Invalid  Dirty 🡪 Invalid  Shared/Clean 🡪 Invalid  Shared/Dirty 🡪 Invalid | Local  Local  Local  Local[[5]](#footnote-5) |
| MB | MB | This instruction causes all outstanding reads and writes prior to this instruction to complete before this instruction completes. | None directly | N/A |
| WMB | WMB | This instruction causes all outstanding writes prior to this instruction to complete before this instruction completes. | None directly | N/A |
| WH64 | WH64  WH64EN | These instructions cause the contents of the cache block to be potentially written, if dirty, out to memory. | Clean 🡪 Invalid  Dirty 🡪 Invalid  Shared/Clean 🡪 Invalid  Shared/Dirty 🡪 Invalid | Local  Local  Local  Local12 |

##### Load (LDx) Instruction versus Dragon Cache Coherency Protocol

The AXP CPU will determine what processing with the system needs to happen based on the state of the cache. Remember that the cache is indexed and tagged, so an index will have multiple addresses associated with it. So, an LDx instruction could refer to an index that is in any of the possible cache states. This section will document each of the potential states and what, if any, messages need to be exchanged with the system and the system with both the local and remote processors.

###### LDx to a Cache Block in the Invalid State

Here are the steps that occur and by which component to support an LDx instruction in a multiprocessor configuration.

| **Local Processor** | **System** | **Remote Processor(s)** |
| --- | --- | --- |
| 1. LDx execution begins 2. The Ibox queues up the load to the Mbox. 3. The Mbox attempts to locate the cache block in the Dcache specified by the LDx. 4. The Mbox gets a miss because the block at the index is Invalid. 5. The Mbox queues up a request to the MAF for the Cbox to process. 6. The Cbox attempts to locate the cache block in the Bcache. 7. The Cbox gets a miss because the block at the index is Invalid. 8. The Cbox sends **ReadBlk** to the system with the physical address for the base of the block. The ReadBlk may go out with additional information requested by the system. |  |  |
|  | 1. The system receives the **ReadBlk** and begins processing. 2. The system sends, in-turn, a **Probe** request to each of the remote processors. The probe request will indicate that a hit should return the data and transition the state to its shared counterpart. |  |
|  |  | 1. A remote processor receives the **Probe** request from the system. 2. The remote processor looks in its Bcache for the block. 3. If the Bcache does not have the block, a cache miss is noted. On the next request to the System, this miss will be returned. A request will be sent with either the M1 or M2 flag set on a miss. 4. If the Bcache does have the block, it will change the state and return the data in a **ProbeResponse** and indicating the new state of the block (Shared/Clean or Shared/Dirty). |
|  | 1. If the system receives a request with either M1 or M2 set:    1. It will send **Probes** to the other remote processors.    2. If all remote processors returned a miss, then the System will read memory for the data. 2. If the system receives a **ProbeResponse:**    1. it will send a **Probe** to the other remote processors but not have them return any data. The state may change. 3. The system will respond with a **SysDc** command indicating the state to which the cache should be initialized. |  |
| 1. The Cbox receives the **ReadDatax** from the system and sends the data and cache state to the Mbox to put into the Dcache. 2. The Mbox puts the block into the Dcache (and Bcache) and sets the state accordingly. 3. The Mbox resumes processing the LDx Instruction and let’s the Ibox know that the instruction is ready to be retired. 4. The Ibox will retired the LDx instruction in issue order and read the data out of the cache and into the target register. |  |  |

### Design Considerations

There are a few things to consider as part of the design.

1. The system is responsible for initializing itself, the devices, and each of the CPUs.
2. Once everything in the system has been initialized, the system does not normally generate any messages on its own (Sleep and Shutdown are the two exceptions).
3. An AXP CPU will send a command to the system for processing.
4. Each AXP CPU will grab the Cchip mutex, queue up its message for processing, signal the Cchip condition variable and then unlock the mutex.
5. Entries in the Cchip queue will be processed from the queue head and queued up for processing at the queue tail.
6. The Cchip will lock its mutex, check its queue for empty or for an entry being processed in the dispatcher.
7. If the queue and dispatcher are empty, the Cchip will wait on its condition variable to be signaled (which will unlock the mutex while waiting and relock it after being signaled).
8. If the queue is empty and the dispatched has something being processed, it will continue to process the dispatched entry until it is completed.
9. The entry in the dispatcher may require one or more requests to be sent to one or more AXP CPUs. The CPUs will respond in kind by placing their response into the queue.
10. When the entry in the dispatcher is being processed, an there can be as many outstanding requests for the entry as there are CPUs.
11. When a CPU queue up an entry as the result of an outstanding dispatcher request, the Cchip will search for a queue entry that is a response to the request and process that portion of the response (there may be more things in the response than just a response).
12. Once all the outstanding dispatcher requests have been processed, the system will response to the dispatch request itself.

### Data Structures to Support Dragon Protocol

The following data structures need to be defined to support the Dragon Protocol.

#### CPU to System Request and ProbeResponse

struct CPU2System

{

u8 command;

union

{

/\*

\* When command != ProbeResponse

\*/

struct

{

u64 pa;

u8 mask;

u8 id;

bool m1;

bool m2;

bool ch;

bool rv;

};

/\*

\* When command == ProbeResponse

\*/

struct

{

u8 status;

u8 vdb;

u8 maf;

bool dm;

bool vs;

bool ms;

};

};

u64 sysData[8]; /\* used in data movement \*/

u32 cpuID;

};

The status field can have one of the following values:

b’00’ HitClean  
b’01’ HitSharedClean  
b’10’ HitDirty  
b’11’ HitSharedDirty

### System to CPU Probe and SysDc

struct System2CPU

{

bool format; /\* if true, dm, nextState, and pa are valid \*/

u8 dm;

u8 nextState;

u64 pa;

u8 sysDc;

u8 id;

bool rvb;

bool rpb;

bool a;

u64 sysData[8]; /\* used in data movement \*/

};

The dm field can have one of the following values:

b'00’ No-op.  
b’01’ Read if hit, return data to system if block is valid (any state, except invalid).  
b’10’ Read if dirty, return data to the system if block is dirty (dirty or shared/dirty).  
b’11’ Read any way, return data to system without regard to block state.

The nextState field can have one of the following values:

b'000’ No-op.  
b’001’ Clean.  
b’010’ Shared/Clean.  
b’011’ [Clean|Shared/Dirty] 🡪 Clean/Shared; Dirty 🡪 Invalid.  
b’100’ Shared/Dirty.  
b’101’ Invalid  
b’110’ [Clean|Dirty] 🡪 Shared/<same>

### Cchip Dispatcher

struct SystemDispatcher

{

bool probesSent;

bool

struct CPU2System \*rq;

struct CPU2System \*rsp[4]; /\* rsp[rq->cpuID] == NULL; \*/

};

### Cchip Dispatcher Processing

When a request gets placed in the dispatcher the following processing is performed. This is incredibly simplified.

if (probesSent == false)

{

for (ii = 0; ii < maxCPUs; ii++)

{

probeSent[ii] = false;

if ((ii != rq->cpuID) && (cpu[ii].present == true))

{

send Probe request to CPU ii;

probeSent[ii] = true;

}

rsp[ii] = NULL;

}

}

else

{

entry = rq->flink;

done = false;

while ((entry != rq) && (done == false))

{

if (entry->command == ProbeResponse)

{

previous = entry->blink;

remque(entry, rsp[entry->cpuID]);

entry = previous;

}

else if ((entry->m1 == true) || (entry->m2 == true))

rsp[entry->cpuID] = entry;

entry = entry->flink;

done = true;

for (ii = 0; ((ii < maxCPUs) && (done == true)); ii++)

if ((probeSent[ii] == true) && (rsp[ii] == NULL)

done = false;

}

if (done == true)

{

miss = true;

hit = 0;

for (ii = 0; ii < maxCPUs; ii++)

{

if ((probeSent[ii] == true) && (rsp[ii]->command == ProbeResponse))

{

if (miss == true)

{

miss = false;

hit = ii;

}

else

mark rsp[ii] available for another

}

}

if (miss == true)

get block from memory;

else

{

get block from rsp[hit];

mark rsp[hit] available;

}

Send probe/SysDc back to rq->cpuID

}

}

# AXP CPU Cache Redesign

The initial implementation of the Icache, Dcache, and Bcache, had each cache separate from one another. See Figure 6‑1 about how the caches can be diagramed.

Figure ‑ Cache Subset Hierarchy

System

Main Memory

In reality just the Icache is separate from the Bcache and Dcache, but the Dcache is a subset of the Bcache. Another way to consider the Bcache/Dcache relationship is that the Bcache is a superset of the Dcache.

Therefore, one of the optimizations that I’d like to redesign into the CPU caches, is that the Bcache holds all the information about each cache block in the CPU and the Dcache is just an index into the Bcache. All state information is maintained in one and only one place, the Bcache. This will allow for the Cbox to more efficiently respond to system commands about the caches and we have minimized data copying within the CPU. This section of the design will document how this will be implemented to better support Cache Coherency.

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)
3. The remote cache will be transitioned to a Shared/Clean state. [↑](#footnote-ref-3)
4. The remote cache will be transitioned to a Shared/Clean state. [↑](#footnote-ref-4)
5. The remote cache will not be transitioned, since other processors could also contain the cache block. One potential optimization would be that if there were just two processors, then we’d know that there were no other caches. The system could take this into consideration. [↑](#footnote-ref-5)