CCP Micro-Architecture Specification

Version 0.91, September 26, 2023

History:

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version # | Changed by | Comments |
| 08/13/2016 | 0.1 | Parimal Gaikwad | Initial version |
| 9/13/2016 | 0.2 | Parimal Gaikwad | Feedback from review incorporated |
| 9/23/2016 | 0.3 | Parimal Gaikwad | Added Security attribute  Added Strobes for Rdrsp and EvictRsp Ports  Separated Fill state and fill data interface  Updated Table 9 of datapipe operation |
| 10/11/2016 | 0.4 | Parimal Gaikwad | Added missing parameters  Added debug interface  Updated table 9 of datapipe operation |
| 8/15/2017 | 0.5 | Boon Chuan | Added missing signal description to match behavior in Concerto 2.x release. Added Tag pipeline and Data pipeline descriptions. |
| 8/31/2017 | 0.6 | Boon Chuan | Added Scratchpad and Atomics. Added new configuration parameters. |
| 9/22/2017 | 0.7 | Boon Chuan | Removed Atomics. Added Memory Sleep interface and description. |
| 11/30/2017 | 0.8 | Boon Chuan | Updated Memory Sleep interface and description. Added Appendix B (CCP Memory Storage Structure) and Appendix C (Cacheline Replacement and Insertion). |
| 9/14/2018 | 0.81 | Boon Chuan | Various clarifications and corrections. cache\_nack\_no\_allocate\_p2 is MIDDLE in Table 2. useDinBuffer description in Table 1. enPartialFill description in Table 1. Add notes to Table 5 for fill data interface. Add “CCP’s various NACK behaviors” to Table 8. Add Appendix 10 to capture various JIRA issues / RTL bugs / functional bugs. |
| 5/13/2018 | 0.82 | Boon Chuan | NRU details added as per AR-468. |
| 1/21/2020 | 0.83 | Boon Chuan | Update Appendix 10 to capture additional JIRA issues/RTL bugs. |
| 11/7/2022 | 0.84 | Boon Chuan | Ncore 3.4: Add new output port signal cache\_current\_nru\_vec\_p2 as per CONC-10667, CONC-10678. |
| 12/8/2022 | 0.85 | Xiaojue Zeng and Rong Xu | Added 2-cycle Data Ram support and configurable Tag Pipe timing stages |
| 1/1/2023 | 0.86 | Kjeld Svendsen | Updated the references |
| 4/13/2023 | 0.87 | Boon Chuan | Add PLRU (pseudo least recently used) to Appendix C, and update parameter description for RepPolicy and nWays accordingly. Also clarifies the NRU description and example.  Update description for the parameters PriSubDiagAddrBits and SecSubRows. |
| 5/10/2023 | 0.90 | Boon Chuan | Updated as per Functional Safety Inspection review spreadsheet. |
| 9/11/2023 | 0.91 | Boon Chuan | Updated PLRU block diagram. PLRU also marks Hit Way as Most Recently Used by updating the PLRU state bits using a table. Clarify the valid\_ways to the PLRU module. Add cache\_set\_index\_p2 signal. |

Contents

[CCP Micro-Architecture Specification 1](#_Toc145661269)

[Reference Documents 6](#_Toc145661270)

[1.0 Introduction 7](#_Toc145661271)

[2.0 Feature List 7](#_Toc145661272)

[3.0 Configurations 7](#_Toc145661273)

[3.1 CPP Assumptions 7](#_Toc145661274)

[3.2 CCP Parameters 7](#_Toc145661275)

[4.0 Interfaces 12](#_Toc145661276)

[4.1 Command Interfaces 12](#_Toc145661277)

[4.2 Data In Interfaces 15](#_Toc145661278)

[4.2.1 Write Interface 15](#_Toc145661279)

[4.2.2 Fill Interface 15](#_Toc145661280)

[4.2.3 cacheData Out Interface 17](#_Toc145661281)

[4.2.4 Debug and CSR Interface 17](#_Toc145661282)

[4.2.5 Scratchpad command interface 19](#_Toc145661283)

[4.2.6 Scratchpad data in interface 20](#_Toc145661284)

[4.2.7 Scratchpad data out interface 20](#_Toc145661285)

[4.2.8 Memory Sleep interface 20](#_Toc145661286)

[4.3 CCP Read Hit Operation 21](#_Toc145661287)

[4.4 CCP Read Hit Operation with correctable tag error 23](#_Toc145661288)

[4.5 CCP Read Miss Allocate with Evict Operation 24](#_Toc145661289)

[4.6 CCP Write Hit Operation 25](#_Toc145661290)

[4.7 CCP Write Miss with Evict Operation 26](#_Toc145661291)

[4.8 CPP Tag pipeline resource usage 27](#_Toc145661292)

[4.9 CPP Tag pipeline replay(nack) conditions 28](#_Toc145661293)

[4.10 Tag pipe op-queue scheduler 29](#_Toc145661294)

[4.11 Fill Op-Queue Scheduler 31](#_Toc145661295)

[4.12 Data Bank Selection 31](#_Toc145661296)

[4.13 CCP data flow 32](#_Toc145661297)

[4.14 CCP datapipe scheduler 32](#_Toc145661298)

[4.14.1 CCP datapipe pipe control – single cycle sram access 33](#_Toc145661299)

[4.14.2 CCP datapipe pipe control – 2-cycle sram 34](#_Toc145661300)

[4.14.3 Pack & unpack for 2 beats per data ram bank 36](#_Toc145661301)

[4.14.4 Sram clock for Data Ram 37](#_Toc145661302)

[4.14.5 Datapipe supported operations 39](#_Toc145661303)

[4.14.6 Control Op Queue data structure 39](#_Toc145661304)

[4.14.7 Fill Op Queue data structure 40](#_Toc145661305)

[4.15 Datapipe Operation pipeline 41](#_Toc145661306)

[4.16 Evict and RdRsp Ports 44](#_Toc145661307)

[4.17 Memory Sleep 44](#_Toc145661308)

[4.18 Debug and Error Insertion Capabilities 44](#_Toc145661309)

[4.19 Performance Counters and Status Capabilities 44](#_Toc145661310)

[4.20 Engineering Registers 44](#_Toc145661311)

[4.21 Control and Status Registers 44](#_Toc145661312)

[5.0 Verification Corner Cases 45](#_Toc145661313)

[6.0 Performance Requirements 45](#_Toc145661314)

[7.0 Appendix A – Scratchpad 46](#_Toc145661315)

[7.1 Scratchpad microarchitecture diagram 46](#_Toc145661316)

[8.0 Appendix B – CCP Memory Storage Structure 47](#_Toc145661317)

[9.0 Appendix C – Cacheline Replacement and Insertion 51](#_Toc145661318)

[10.0 Appendix D – Issues 55](#_Toc145661319)

[Table 1 CCP Parameters 11](#_Toc134631222)

[Table 2 CCP command interface signals 14](#_Toc134631223)

[Table 3 CCP write interface signals 15](#_Toc134631224)

[Table 4 Fill Data Interface signals 16](#_Toc134631225)

[Table 5 Fill Control Interface signals 16](#_Toc134631226)

[Table 6 CCP fill interface signals 16](#_Toc134631227)

[Table 7 CCP evict interface signals 17](#_Toc134631228)

[Table 8 CCP read response interface signals 17](#_Toc134631229)

[Table 9 Error Logging, Initialization and Maintenance Interface signals 18](#_Toc134631230)

[Table 10 Scratchpad Command Interface signals 19](#_Toc134631231)

[Table 11 Scratchpad Data In Interface signals 20](#_Toc134631232)

[Table 12 Scratchpad Data Out Interface signals 20](#_Toc134631233)

[Table 13 Memory Sleep Interface signals 21](#_Toc134631234)

[Table 14 CCP NACK Behaviors 28](#_Toc134631235)

[Table 15 CCP Command Operations 30](#_Toc134631236)

[Table 16 CCP Tagpipe Pipeline 31](#_Toc134631237)

[Table 17 CCP Datapipe Time Slot Reservation Table for 1 cycle SRAM 33](#_Toc134631238)

[Table 18 CCP Datapipe Time Slot Reservation Table for 2 cycle SRAM 35](#_Toc134631239)

[Table 19 Datapipe Opcodes 39](#_Toc134631240)

[Table 20 Datapipe Op-queue Data Structure 39](#_Toc134631241)

[Table 21 Datapipe Bank Per-Data-Beat Micro Operation 40](#_Toc134631242)

[Table 22 Datapipe Bank Per-Data-Beat Control signals 40](#_Toc134631243)

[Table 23 Fill Op Queue Data Structure 40](#_Toc134631244)

[Figure 1 CCP top level block diagram 7](#_Toc134631245)

[Figure 2 CCP Command Interface Timing 12](#_Toc134631246)

[Figure 3 CCP read hit pipeline 22](#_Toc134631247)

[Figure 4 CCP read hit pipeline with correctable tag error 23](#_Toc134631248)

[Figure 5 CCP read miss with Evict pipeline 24](#_Toc134631249)

[Figure 6 CCP write Hit pipeline 25](#_Toc134631250)

[Figure 7 CCP Write Miss with Evict pipeline 26](#_Toc134631251)

[Figure 8 Tagpipe 27](#_Toc134631252)

[Figure 9 Tagpipe with configurable pipeline stages P0+ adnd P1- 27](#_Toc134631253)

[Figure 10 CCP Tagpipe Operation Priority 30](#_Toc134631254)

[Figure 11 CCP cache\_nack\_ce\_p2 Timing Diagram 31](#_Toc134631255)

[Figure 12 CCP Data Flow 32](#_Toc134631256)

[Figure 13 Datapipe 32](#_Toc134631257)

[Figure 14 Datapipe with single cycle SRAM access 33](#_Toc134631258)

[Figure 15 Datapipe with 2 cycle SRAM access 34](#_Toc134631259)

[Figure 16 Datapipe pack and unpack for 2 beats per Data RAM entry 36](#_Toc134631260)

[Figure 17 Data RAM running at half frequency 37](#_Toc134631261)

[Figure 18 Timing Diagram for Data RAM at half frequency 37](#_Toc134631262)

[Figure 19 Data RAM at half fequency with 25% clock duty cycle 38](#_Toc134631263)

[Figure 20 Data RAM at half frequency with early enable 38](#_Toc134631264)

[Figure 21 Data array Read Operation with correctable errors 41](#_Toc134631265)

[Figure 22 Data array Evict Write Operation. 42](#_Toc134631266)

[Figure 23 Scratchpad Data Array Read Operation (useLowLatency=1) 43](#_Toc134631267)

[Figure 24 Scratchpad Data Array Write Operation 43](#_Toc134631268)

[Figure 25 Scratchpad microarchitecture diagram 46](#_Toc134631269)

[Figure 26 Pseudo LRU block diagram 52](#_Toc134631270)

# Reference Documents

|  |  |
| --- | --- |
| **Specification Name** | **Link** |
| Ncore 3 System Architecture | [​Folder icon Architecture\_docs](https://arterisip.atlassian.net/wiki/spaces/ENGR/pages/16160456/Arch+Ncore+3.4+Docs?preview=/16160456/212762657/Ncore34_systarch_draft_v0.51.pdf) |
| Concerto Messaging Protocol Specification | [​Folder icon Architecture\_docs](https://arterisip.atlassian.net/wiki/spaces/ENGR/pages/16160456/Arch+Ncore+3.4+Docs?preview=/16160456/217055344/ConcertoCProtocolArch_NoCB_96_clean%20copy.pdf) |
| Ncore 3 Parameters specification | [​Folder icon Architecture\_docs](https://arterisip.atlassian.net/wiki/spaces/ENGR/pages/16160456/Arch+Ncore+3.4+Docs?preview=/16160456/190808075/Ncore%203.4%20Architecture%20Parameter%20Specification.pdf) |

# Introduction

The scope of this document is to outline the micro architectural details of CCP. CCP is a configurable IP block which supports various cache operations. It can be configured to support different caching models.



Figure 1 CCP top level block diagram

# Feature List

CCP supports following features

* Supports various cache configurations with sets ways.
* Support different cache models
* Cache read hit latency of 3 to 5 cycles @ 500MHz to 1.2GHz in TSMC FF corner 32nm@1.2V,125C.
* Support various replacement policies: random, NRU.
* Supports configurable number of data banks and tag banks
* Support for configurable datapath din and dout buffers
* Scratchpad memory configurable by power-of-2 way, with read latency of 1 to 2 cycles @ 500MHz to 1.2GHz
* CCP can be build-time configured as Cache only, Scratchpad only, or Cache plus Scratchpad
* Support for memory sleep.

# Configurations

## CPP Assumptions

Tag state value 0 is assumed as invalid.

Tag and data arrays are single ported arrays.

Replacement Policy(RP) array can be single port or dual port.

Memories do not have byte enables.

## CCP Parameters

Different CCP parameters are listed in Table 1.

| **Parameter** | **Legal Values** | **Defaults** | **Description** |
| --- | --- | --- | --- |
| useCache | 0,1 | 1 | To enable/instantiate CCP cache logic and interfaces. |
| useScratchpad | 0,1 | 0 | To enable/instantiate CCP scratchpad logic and interfaces. |
| useLowLatency | 0,1 | 0 | To save 2 cycle latency by changing the OpQueue in Datapipe to bypassable. and the ECC self-correcting-FIFO in the DataOuput module to bypassable.  Default setting is non-bypassable. |
| useMemorySleep | 0,1 | 0 | To enable/instantiate CCP memory sleep logic and interfaces. |
| enPartialFill | 0,1 | 0 | To enable Partial Cacheline Fill where the number of data beats can be less than a full cacheline, and the byte enables might not be all 1s. |
| nSets | Refer to Concerto System Parameter Spec | 1024 | Number of sets in the cache. |
| UseTagRamInputFlop | 0,1 | 0 | To enable tag ram input flops for slow SRAM timing fix. |
| UseTagRamOutputFlop | 0,1 | 0 | To enable tag ram output flops for slow SRAM timing fix. |
| nSramAccess | 1, 2 | 1 | To configure the data ram access cycle, 1 to select single cycle SRAM access, 2 to use two cycle SRAM access |
| useSramInputFlop | 0,1 | 0 | To enable data ram input flops for slow SRAM timing fix. Must be enabled if nBeatsPerBank > 1 |
| nWays | Refer to Concerto System Parameter Spec | 16 | Number of ways in the cache.  When RepPolicy=”PLRU”, then legal nWays value is 2,4,6,8,12,16,20,24,28,32. |
| wData | 64,128,256,512 | 128 | Width of data interface to the cache in Bits. |
| nDataBeatsPerBank | 1,2,4,8 | 1 | Number of beats of data stored in data array bank. Please not that all data bank operations are done on a beat wide basis. Entire cacheline resides in the same bank. Above parameter decides each operation in data array is done on how many beats together.e.g. If nDataBeat = 2, DataBank accesses are performed for 2 beats together. So, we could have wider memories. |
| nTagBanks | 1,2,4 | 1 | Number of Tag Banks |
| nDataBanks | 1,2,4 | 4 | Number of data banks |
| TagErrInfo | Refer to Concerto System Parameter Spec for details | NONE | NONE, PARITY and ECC Variants |
| DataErrInfo | Refer to Concerto System Parameter Spec for details | NONE | NONE, PARITY and ECC Variants |
| nClkDiv | 1,2,4 | 1 | This indicates clock frequency at which data arrays are to be run at. |
| PriSubDiagAddrBits[] | Refer to Concerto System Parameter Spec for details | Bits selecting Index | Primary submatrix diagonal address bits. This element represents a vector. Each element of the vector is an integer that represents a primary address bit, and each element must have a unique value.  NOTE: set\_select is a hardware library module that uses PriSubDiagAddrBits[] and SecSubRows[]. This set\_select module selects certain bits in an input address, as specified by a primary bit array parameter PriSubDiagAddrBits[], and exclusive-OR each selected bit with every address bit as specified by a secondary bit array parameter SecSubRows[].  Assume PriSubDiagAddrBits[] to be A[] with 4 elements.  Assume SecSubRows[] to be B[] with 4 elements, each element is an array of 8 integers.  The resulting Hash function output C[] of 4 elements would be  C[0]=A[3]^{^B[3]}  C[1]=A[2]^{^B[2]}  C[2]=A[1]^{^B[1]}  C[3]=A[0]^{^B[0]}  NOTE: PriSubDiagAddrBits is an array of integer.  SecSubRows is an array of array of integer. |
| SecSubRows[]{} | Refer to Concerto System Parameter Spec for details |  | Secondary submatrix row bundles. This element represents a vector whose length equals PriSubDiagAddrBits[]. Each element of the vector is a bundle containing a vector of integers that represent the secondary address bits in each row.  NOTE: SecSubRows is an array of array of integer. |
| TagBankSelectBits[] |  | Bits selecting Tag Bank | Bits selecting Tag Bank |
| PortPriSubDiagAddrBits[] | Refer to Concerto System Parameter Spec for details | Bits selecting CCP Port | For port select |
| PortSecSubRows | Refer to Concerto System Parameter Spec for details |  | For port select |
| num\_port | Refer to Concerto System Parameter Spec for details | 1 | Number of CCP instances interleaved in a group. |
| DataBankSelectBits[] |  | Bits selecting data bank array | Bits selecting data bank array |
| wAddr |  | Byte address width |  |
| nRPPorts | 1,2 | Number of ports to RP bank | Number of ports to RP bank |
| RepPolicy |  | NRU | Replacement Policy: PLRU, NRU, Random. |
| useDinBuffer | 0,1 | 0 | If this buffering is enabled, then logic expects data coming on write ~~and fill buffer~~ to be buffered inside data bank (for timing and performance reason) |
| useDoutBuffer | 0,1 | 0 | If this buffering is enabled, then data coming out on evict and readrsp port is buffered after reading from data arrays |

Table 1 CCP Parameters

NOTE:

When CCP is configured to Low Latency Mode:

* cache read hit latency is 3 cycles (data appears near the end of P3 cycle after ECC error detection. Note: P0 is the tag lookup cycle. If ECC correctible error is detected, then the corrected data appears in P4 cycle after ECC error correction).
* scratchpad memory read latency is 1 cycle (data appears near the end of P1 cycle after ECC error detection. Note: P0 is the scratchpad command cycle. If ECC correctible error is detected, then the corrected data appears in P2 cycle after ECC error correction).

By default, CCP is not configured in Low Latency Mode:

* cache read hit latency is 5 cycles (data appears in the beginning of P5 cycle. Note: P0 is the tag lookup cycle. If ECC correctible error is detected, then the corrected data appears in the beginning of P6 cycle).
* scratchpad memory read latency is 2 cycles (data appears in the beginning of P2 cycle. Note: P0 is the scratchpad command cycle. If ECC correctible error is detected, then the corrected data appears in the beginning of P3 cycle).

When useDinBuffer is set to 0, i.e. the write buffer is not instantiated, a valid-ready handshake on CCP write port means the write data is being written to the CCP Data Array, and the valid-ready handshake on the last data beat on CCP write port pops the write request from the CCP Op Queue. Unlike Fill data port that provides “fill\_done”, CCP write port doesn’t provide “write\_done” as CCP write port is synchronous to the CCP command port.

# Interfaces

The CPP interfaces can be divided in to two main categories. Control interface and Data interfaces.

## Command Interfaces

CCP operations and status is communicated through this interface. This is a pipelined interface and each request traverses through three cycles. Interface signals with respect to different cycles are shown in Figure 2. The signals are described in Table 2.



Figure 2 CCP Command Interface Timing

| Signal | IO | Width | Cycle | Timing | Description |
| --- | --- | --- | --- | --- | --- |
| ctrl\_op\_valid\_p0[] | In | Cfg | P0 | Middle | Valid input pulse. (per bank signal) |
| ctrl\_op\_address\_p0[] | In | Cfg | P0 | Middle | Complete operation byte address. |
| ctrl\_op\_security\_p0 | In | Cfg | P0 | Middle | Security attribute for the address |
| cache\_op\_ready\_p0[] | Out | Cfg | P0 | Early | Cache is ready to accept inputs. (per bank signal) |
| ctrl\_op\_allocate\_p2 | In | 1 | P2 | Early | Allocate attribute of the transaction |
| ctrl\_op\_read\_data\_p2 | In | 1 | P2 | Late | When set data is read from cache for the current transaction. |
| ctrl\_op\_write\_data\_p2 | In | 1 | P2 | Late | When set data is written into cache for the current transaction. |
| ctrl\_op\_port\_sel\_p2 | In | 1 | P2 | Early | Set one to route output data to evict port. Set zero to route output data to read response port. |
| ctrl\_op\_bypass\_p2 | In | 1 | P2 | Early | Set one to bypass data on write port to evict port or read response port selectable by ctrl\_op\_port\_sel\_p2. |
| ctrl\_op\_rp\_update\_p2 | In | 1 | P2 | Late | Set to update the replacement policy |
| ctrl\_op\_tag\_state\_update\_p2 | In | 1 | P2 | Late | Set to update the state and tag array |
| ctrl\_op\_state\_p2[] | In | Cfg | P2 | Late | Set the new value of state, valid only when ctrl\_op\_tag\_state\_update\_p2 is set.  A value of zero indicates the cacheline state is invalid.  A value of non-zero indicates the cacheline state is valid. |
| ctrl\_op\_burst\_len\_p2[] | In | Cfg | P2 | Early | Burst length in beats |
| ctrl\_op\_setway\_debug\_p2 | In | 1 | P2 | Early | Identifies the operation as a set way operation for maintenance operation (address is a dummy address which matched the Index from CSR).  NOTE: CCP requires the external logic to wait till the CCP pipeline stages P0, P1, P2 are empty before issuing a maintenance request. |
| ctrl\_op\_ways\_busy\_vec\_p2[] | In | Cfg | P2 | Early | All fill pending ways.  Ways busy vector for the current transactions index.  External control logic looks at the address during P1 cycle and compares all the pending addresses with the transaction address. For all the pending transaction which match index with current transaction, external logic is tracking way allocated to them (indicated through cache\_alloc\_way\_vec\_p2  while allocating). External logic combines this way allocation information to generate ctrl\_op\_ways\_busy\_vec\_p2 signal. Also note that information sent by external logic is a cycle stale. (done in P1). If transaction ahead of it belonged to the same index and ended up allocating, we need to add way which was allocated to ctrl\_op\_ways\_busy\_vec\_p2 vector. This is done by external logic.  Corner case to be taken care by external logic: when there is an allocation in P2 cycle while ways pending calculation is being done in P1 cycle. A correction is needed in this case to make the ways pending P2 cycle accurate.  ways\_pending = ways\_pending generated in P1 cycle + corrected ways pending.  Corrected ways pending is if there was an allocation in P2 in the previous cycle for the same index, then that way should be part of ways pending. |
| ctrl\_op\_ways\_stale\_vec\_p2[] | In | Cfg | P2 | Early | This is a subset of all fill pending ways. It does not include ways where only the state is being updated and the tag will remain the same i.e. upgrade cases.  Corner case to be taken care by external logic: when there is an eviction in P2 cycle while ways stale calculation is being done in P1 cycle. A correction is needed in this case to make the ways stale P2 cycle accurate.  ways\_stale = ways\_stale generated in P1 cycle + corrected ways stale.  Corrected ways stale is if there was an eviction in P2 in the previous cycle for the same index, then that way which was evicted should be part of ways stale. |
| cache\_valid\_p2 | Out | 1 | P2 | Early | CCP cache interface output is valid |
| cache\_current\_state\_p2[] | Out | Cfg | P2 | Middle | Current state of the line in cache.  It’s a cache hit if current state of the cacheline is valid (i.e. the value of the state is non-zero) and cache\_valid\_p2 is set. |
| cache\_current\_nru\_vec\_p2[] | Out | Cfg | P2 | Middle | Current NRU bit vector, each bit per way. This port only exists when the replacement policy is not RANDOM and the number of ways is more than 1. |
| cache\_alloc\_way\_vec\_p2[] | Out | Cfg | P2 | Late | Allocated way (one hot bit vector) |
| cache\_hit\_way\_vec\_p2[] | Out | Cfg | P2 | Late | Tag hit (i.e. tag matches) way (bit vector) |
| cache\_set\_index\_p2[] | Out | Cfg | P2 | Early | The index for the set for the tag array. This signal is useful for verification. |
| cache\_evict\_valid\_p2 | Out | 1 | P2 | Middle | Set if eviction is needed for the current operation to complete.  NOTE: ignore this signal if it’s a WriteHitUpgrade i.e. this cacheline is reused by the external logic. |
| cache\_evict\_address\_p2[] | Out | Cfg | P2 | Late | Complete cache line address of the line being evicted. In the case of set way operation cache line address of the line being invalidated. Valid if cache\_evict\_valid\_p2 is set |
| cache\_evict\_security\_p2 | Out | Cfg | P2 | Late | Security attribute for the evict address |
| cache\_evict\_state\_p2[] | Out | Cfg | P2 | Late | State of the line being evicted. Valid if cache\_evict\_valid\_p2 is set |
| cache\_nack\_uce\_p2 | Out | 1 | P2 | Middle | ECC error indication, set for uncorrectable error from Tag or RP array is detected. |
| cache\_nack\_p2 | Out | 1 | P2 | Middle | If set the ctrl logic should replay the transaction. (this is set if CCP is not able to make forward progress due to data flow control) |
| cache\_nack\_ce\_p2 | Out | 1 | P2 | Middle | Operations in P0 & P1 are flushed, operation in P2 stalls for 2 cycle and then makes forward progress. This is a pulse.  If this signal is asserted external logic should ignore all other signals. |
| cache\_nack\_no\_allocate\_p2 | Out | 1 | P2 | Middle | Set with cahce\_nack\_p2 signal if a way could not be allocated (evicted) to complete the operation. |

Table 2 CCP command interface signals

Notes

(1)

Don’t assert ctrl\_op\_read\_data\_p2, ctrl\_op\_write\_data\_p2, ctrl\_op\_bypass\_p2 if the corresponding ctrl\_op\_address\_p2 (and ctrl\_op\_address\_p1 also) is a tag hit with way and index address matches any outstanding pending fill way and index address (unless the system design prevents this from happening), otherwise the request\_muxarb in CCP Datapipe Bank controller could deadlock. To avoid this deadlock, the existing CCP implementation can be modified such that the fill request to the request\_muxarb is only suppressed for a new fill request, and never for a current fill request that is being serviced (i.e. in the middle of a fill transfer) by the request\_muxarb.

## Data In Interfaces

This interface is further categorized into two sub interfaces. Write interface and Fill interface each of these CCP interfaces can be configured to have multiple ports.

For Atomics, the organization for the atomic data on the CCP Write port and CCP Fill port is shown in the Appendix section of this document.

### Write Interface

The write interface is used for incoming write transactions. Signals associated with this port are shown in Table 3. Note that this is a valid and ready flow controlled interface.

| Signal | IO | Width | | Description |
| --- | --- | --- | --- | --- |
| ctrl\_wr\_valid | In | 1 | Input valid signal | |
| ctrl\_wr\_data[] | In | Cfg | Input data (beat wide) with poison bit in the most significant bit | |
| ctrl\_wr\_byte\_en[] | In | Cfg | Byte enables | |
| ctrl\_wr\_beat\_num[] | In | Cfg | Input data write beat number | |
| ctrl\_wr\_last | In | 1 | Indicates last beat | |
| cache\_wr\_ready | Out | 1 | Set when ready to accept a beat of a cache line worth data | |

Table 3 CCP write interface signals

### Fill Interface

This is an asynchronous interface with respect to the pipeline. The fill interface is dedicated for fills, signals associated with this interface port are shown in Table 5. Note that this is a valid and ready flow controlled interface. Signals cache\_fill\_done0 and cache\_fill\_done\_id0 in red are sideband signals which are not part of the main valid ready flow control.

Fill interface is separated into fill data interface and fill control interface. Fill control interface communicates tag updates to tag array.

Whenever state update interface has some data to be written, it asserts *cache\_fill\_valid,* tag control to insert a memory write, de-asserts ready for the bank to which this fill is to be done. Tag control logic assert *cache\_fill\_ready* when above tag memory update is set up. External logic can assume that state update is done when it sees assertion of *cache\_fill\_ready.* Hence, no fill\_id is passed with control interface.

On fill data interface ctrl\_fill\_data\_valid is asserted when it has data to be written. Based on the address it goes to a specific bank of the data array. Data array bank scheduler sees these fill data and whenever fill data is updated, it asserts the fill done for the specific ID. Note that the fill data interface is extended to support byte enables for partial fill.

| Signal | IO | Width | Description |
| --- | --- | --- | --- |
| ctrl\_fill\_data\_valid | In | 1 | Input valid signal |
| ctrl\_fill\_data[] | In | Cfg | Input data (beat wide), with poison bit in the most significant bit.  NOTE: fill is always a full cacheline, therefore the number of data beats in fill is always the number of data beats in a full cacheline. |
| ctrl\_fill\_data\_id[] | In | Cfg | Transaction table ID to track the fill. |
| ctrl\_fill\_data\_address[] | In | Cfg | Cache line address |
| ctrl\_fill\_data\_way\_num[] | In | Cfg | Fill way number. This signal exists if the parameter nWays is greater than 1. |
| ctrl\_fill\_data\_beat\_num[] | In | Cfg | Input data fill beat number. This is the LS bits of the CCP Data Array address. Each entry of the CCP Data Array is the data beat of a cacheline. |
| ctrl\_fill\_data\_last | In | Cfg | Input data fill beat is last. This signal exists when enPartialFill=1. |
| ctrl\_fill\_data\_byte\_en[] | In | Cfg | Byte enables. This signal exists when enPartialFill=1. |
| cache\_fill\_data\_ready | Out | 1 | Set when ready to accept a beat of a cache line worth data |

Table Fill Data Interface signals

| Signal | IO | Width | Description |
| --- | --- | --- | --- |
| ctrl\_fill\_valid | In | 1 | Input valid signal |
| ctrl\_fill\_address[] | In | Cfg | Cache line address |
| ctrl\_fill\_security | In | Cfg | Fill Security attribute |
| ctrl\_fill\_way\_num[] | In | Cfg | Fill way number. This signal exists if the parameter nWays is greater than 1.  NOTE: Fill also sets the NRU bit to 1 for the Fill way. Fill is simply a write to the Tag SRAM for the Fill Way. |
| ctrl\_fill\_state[] | In | Cfg | Install state |
| cache\_fill\_ready | Out | 1 | Set when ready to accept a ctrl\_fill\_valid request |

Table Fill Control Interface signals

| Signal | IO | Width | Description |
| --- | --- | --- | --- |
| cache\_fill\_done | Out | 1 | Fill done indication pulse for fill associated with id on cache\_fill\_done\_id.  NOTE: cache\_fill\_done is asserted when the last data beat of the fill cacheline is written into CCP via the ctrl\_fill\_data interface. |
| cache\_fill\_done\_id[] | Out | Cfg | Transaction table ID for tracking the fill. valid when cache\_fill\_done is set |

Table 6 CCP fill interface signals

NOTE:

Fill Data Interface is a packet interface. The last signal is asserted to indicate the last beat of data transfer. The CCP request arbiter re-arbs on last data beat transfer. External logic does not have to assert "last" with every data beat in the partial fill case. The fill done signal will be asserted eventually for every last data beat transfer.

### cacheData Out Interface

This interface is further categorized into two sub interfaces. Read response interface and evict interface, each of these CCP interfaces can be configured to have multiple ports. The signals for evict interface port are shown in Table 6. Note that this is a valid and ready flow controlled interface. The signals for read response interface port are shown in Table 7. Note that this is a valid and ready flow controlled interface.

| **Signal** | **IO** | **Width** | **Description** |
| --- | --- | --- | --- |
| **cache\_evict\_valid** | Out | 1 | Cache evict valid |
| **cache\_evict\_data[]** | Out | Cfg | Cache data, beat wide, with poison bit in the most significant bit |
| **cache\_evict\_byteen[]** | Out | Cfg | Cache data byte enables |
| **cache\_evict\_last** | Out | 1 | Indicates last beat |
| **cache\_evict\_cancel** | Out | 1 | Cache evict cancel asserted if correctable ECC error its detected |
| **cache\_evict\_ready** | In | 1 | Cache evict ready signal (asserted by external logic when it is ready to accept data) |

Table 7 CCP evict interface signals

| **Signal** | **IO** | **Width** | **Description** |
| --- | --- | --- | --- |
| **cache\_rdrsp\_valid** | Out | 1 | Cache read response valid |
| **cache\_rdrsp\_data[]** | Out | Cfg | Cache data, beat wide, with poison bit in the most significant bit |
| **cache\_rdrsp\_byteen[]** | Out | Cfg | Cache data byte enables |
| **cache\_rdrsp\_last** | Out | 1 | Indicates last beat |
| **cache\_rdrsp\_cancel** | Out | 1 | Cache read response cancel asserted if correctable ECC error its detected |
| **cache\_rdrsp\_ready** | In | 1 | Cache read response ready signal (asserted by external logic when it is ready to accept data) |

Table 8 CCP read response interface signals

### Debug and CSR Interface

| **Signal** | **IO** | **Width** | **Description** |
| --- | --- | --- | --- |
| **CorrErrDetectEn** | In | 1 | Correctable Error Detect Enable |
| **UnCorrErrDetectEn** | In | 1 | Uncorrectable Error Detect Enable |
| **correctible\_error\_valid** | Out | 1 | Refer CSAS |
| **correctible\_error\_type[]** | Out | 4 | Refer CSAS |
| **correctible\_error\_info[]** | Out | 8 | Refer CSAS |
| **correctible\_error\_entry[]** | Out | 20 | Refer CSAS |
| **correctible\_error\_way[]** | Out | 6 | Refer CSAS |
| **correctible\_error\_word[]** | Out | 6 | Refer CSAS |
| **correctible\_error\_double\_error** | Out | 1 | Refer CSAS |
| **correctible\_error\_addr\_hi[]** | Out | 12 | Refer CSAS |
| **uncorrectible\_error\_valid** | Out | 1 | Refer CSAS |
| **uncorrectible\_error\_type[]** | Out | 4 | Refer CSAS |
| **uncorrectible\_error\_info[]** | Out | 8 | Refer CSAS |
| **uncorrectible\_error\_entry[]** | Out | 20 | Refer CSAS |
| **uncorrectible\_error\_way[]** | Out | 6 | Refer CSAS |
| **uncorrectible\_error\_word[]** | Out | 6 | Refer CSAS |
| **uncorrectible\_error\_double\_error** | Out | 1 | Refer CSAS |
| **uncorrectible\_error\_addr\_hi[]** | Out | 12 | Refer CSAS |
| **reinit** | In | 1 | Initialize tag memory. It is a pulse. |
| **init\_done** | Out | 1 | Indicates tag memory init is done |
| **instance\_id[]** | In | Cfg | This signal exists when the parameter num\_ports is greater than 1. |
| **maint\_req\_opcode[]** | In | 4 | Refer CSAS |
| **maint\_req\_array\_sel** | In | 1 | Refer CSAS |
| **maint\_req\_data[]** | In | 32 | Refer CSAS |
| **maint\_req\_way[]** | In | 6 | Refer CSAS |
| **maint\_req\_entry[]** | In | 20 | Refer CSAS |
| **maint\_req\_word[]** | In | 6 | Refer CSAS |
| **maint\_active** | Out | 1 | Refer CSAS |
| **maint\_read\_data[]** | Out | 32 | Refer CSAS |
| **maint\_read\_data\_en** | Out | 1 | Refer CSAS |

Table Error Logging, Initialization and Maintenance Interface signals

CCP does not have any registers in it. All configuration status and performance signals run through this interface.

The maintenance operations supported by Tagpipe are:

- MAINT\_READ\_INDEX\_WAY

- MAINT\_WRITE\_INDEX\_WAY

- MAINT\_RECALL\_INDEX\_WAY

The maintenance operations supported by Datapipe are:

- MAINT\_READ\_INDEX\_WAY

- MAINT\_WRITE\_INDEX\_WAY

CCUCMLR0 Maintenance Location Register comprises

[19:0]=MntEntry=Set of the cacheline,

[25:20]=MntWay=Way of the cacheline,

[31:26]=MntWord=32-bit Word of the cacheline

NOTE: the MS bits of MntWord is used to specify the beat address for the Data Array. Each entry in the Data Array is a data beat. See Chapter 8 Appendix B for the CCP Data Array organization and also CCP Tag Array.

maint\_req\_word[\=wMntAddr-1=\:\=wMntAddr-wBeats=] => This is beat address in CCP Datapipe Bank.

where wMntAddr = log2ceil( Math.pow(2,u.getParam('wCacheLineOffset')) / 2 );

For 128-bit data bus, 64-byte cacheline: wBeats=2, wMntAddr=log2ceil(2^6 / 2)=log2ceil(32)=5.  
So maint\_req\_word[4:3] indicates beat address, maint\_req\_word[2:0] indicates the word within the beat.

For 256-bit data bus, 64-byte cacheline: wBeats=1, wMntAddr=log2ceil(2^6 / 2)=log2ceil(32)=5.  
So maint\_req\_word[4:4] indicates beat address, maint\_req\_word[3:0] indicates the word within the beat.

CCUCELR00 Correctable Error Location Register comprises

[19:0]=ErrEntry=Set of the cacheline

[25:20]=ErrWay=Way of the cacheline

[31:26]=ErrWord=Data Beat of the cacheline

### Scratchpad command interface

Scratchpad interface is designed to be an independent parallel interface. Scratchpad has its own command interface, data in interface and data out interface.

Scratchpad command interface bypasses the CCP Tagpipe module and interfaces directly to the CCP Datapipe module for cacheline read and write operations to the Data Array. Scratchpad command is translated, routed, and written into one of the CCP Datapipe Bank modules’ Scratchpad Control Queue. See Appendix for Scratchpad microarchitecture diagram.

| Signal | IO | Width | Cycle | Timing | Description |
| --- | --- | --- | --- | --- | --- |
| scratch\_op\_valid[] | In | Cfg | P2 | Middle | Valid input pulse (per data bank signal, must be onehot0). |
| scratch\_op\_way\_num[]  scratch\_op\_index\_addr[]  scratch\_op\_beat\_num[] | In | Cfg | P2 | Middle | The address that is fed directly to the selected bank of data memory is in this format [way, index, beatID]. |
| scratch\_op\_read\_data | In | 1 | P2 | Middle | When set data is read from cache for the current transaction. The read data is always sent to scratchpad read response port. |
| scratch\_op\_write\_data | In | 1 | P2 | Middle | When set data is written into cache for the current transaction |
| scratch\_op\_burst\_len | In | Cfg | P2 | Middle | Burst length in data beats. |
| scratch\_op\_ready[] | Out | Cfg | P2 | Early | Cache is ready to accept inputs. (per bank signal) |

Table Scratchpad Command Interface signals

The Data Array is organized as banks, and each bank stores an entire cacheline in data beats (each entry in a bank is a data beat). The address format used to access an entry in a bank is [way, index, beat], with way in the most significant bits.

### Scratchpad data in interface

| Signal | IO | Width | | Description |
| --- | --- | --- | --- | --- |
| scratch\_wr\_valid | In | 1 | Input valid signal | |
| scratch\_wr\_data[] | In | Cfg | Input data (beat wide) with poison bit in the most significant bit | |
| scratch\_wr\_byte\_en[] | In | Cfg | byte enables | |
| scratch\_wr\_beat\_num[] | In | Cfg | Input data write beat number | |
| scratch\_wr\_last | In | 1 | Indicates last beat | |
| scratch\_wr\_ready | Out | 1 | Set when ready to accept a beat of a cache line worth data | |

Table Scratchpad Data In Interface signals

### Scratchpad data out interface

| **Signal** | **IO** | **Width** | **Description** |
| --- | --- | --- | --- |
| **scratch\_rdrsp\_valid** | Out | 1 | Cache read response valid |
| **scratch\_rdrsp\_data[]** | Out | Cfg | Cache data, beat wide, with poison bit in the most significant bit |
| **scratch\_rdrsp\_byteen[]** | Out | Cfg | Cache data byte enables |
| **scratch\_rdrsp\_last** | Out | 1 | Indicates last beat |
| **scratch\_rdrsp\_cancel** | Out | 1 | Cache read response cancel asserted if correctable ECC error its detected |
| **scratch\_rdrsp\_ready** | In | 1 | Cache read response ready signal (asserted by external logic when it is ready to accept data) |

Table Scratchpad Data Out Interface signals

### Memory Sleep interface

| Signal | IO | Width | | Description |
| --- | --- | --- | --- | --- |
| tag\_bank\_sleep | In | Cfg | Per Tag Bank signal.  Sleep request from external logic to put the CCP tag memory bank to sleep. The external logic monitors the Busy status from CCP for the tag memory bank. If Busy status is low and Sleep request is high, then the external logic can assert the LS pin to the RAM for the tag memory bank to put it to sleep. Note that CCP doesn’t contain any RAM. The RAM is located outside the CCP. | |
| tag\_bank\_busy | Out | Cfg | Per Bank signal.  Busy status for the tag memory bank from the CCP Tagpipe controller. | |
| tag\_bank\_wakeup | Out | Cfg | Per Bank signal.  Wakeup request tor the tag memory bank from the CCP Tagpipe controller. | |
| data\_bank\_sleep | In | Cfg | Per Data Bank signal.  Sleep request from external logic to put the CCP data memory bank to sleep. The external logic monitors the Busy status from CCP for the data memory bank. If Busy status is low and Sleep request is high, then the external logic can assert the LS pin to the RAM for the data memory bank to put it to sleep. Note that CCP doesn’t contain any RAM. The RAM is located outside the CCP. | |
| data\_bank\_busy | Out | Cfg | Per Bank signal.  Busy status for the data memory bank from the CCP Datapipe controller. | |
| data\_bank\_wakeup | Out | Cfg | Per Bank signal.  Wakeup request for the data memory bank from the CCP Tagpipe controller. | |

Table Memory Sleep Interface signals

## CCP Read Hit Operation

For Configuration with useLowLatency=1, useDinBuffer=0, useDoutBuffer=0

CCP read hit pipeline is shown in the pipeline diagram below, assuming the op-queue in datapipe is configured to be bypassable, so that the data array is accessed for read during P2 (otherwise the data array is accessed for read during P3). Tag/State/RP array read is set up during P0. Tag/State/RP array output appear during P1.

If multiple Banks are present, data output is muxed between different banks and registered to be used in P2. During cycle P2 Tag address compare is done. If hit is detected, then read op-que is scheduled for banks on which read is to be performed. If this op-queue is empty, then data array read can be set up in P2 itself. Data array output appears in P3 and is written into a ECC self-correcting FIFO. The FIFO output is available during cycle P4.

CCP itself doesn’t detect a read hit. It passes a read hit status to the external logic. External logic asserts **ctl\_op\_read\_data\_p2** signal if external logic wants to perform read of the data array for the matched index and way.



Figure 3 CCP read hit pipeline

## CCP Read Hit Operation with correctable tag error

For Configuration with useLowLatency=1, useDinBuffer=0, useDoutBuffer=0

During cycle P2 Tag address compare is done. ECC check is also performed in parallel to the tag compare. If correctible error is detected then, CCP asserts **cache\_nack\_ce\_p2**. This indicates to the external logic that correctible error is detected and it should flush its pipeline for next 2 cycles. CCP does the error detection in cycle P2. In next cycle it does the correction and writes back corrected data to the tag. In next cycle it performs compare operation again with the corrected data. If hit is detected, then read op-que is scheduled for banks on which read is to be performed.



Figure 4 CCP read hit pipeline with correctable tag error

## CCP Read Miss Allocate with Evict Operation

For Configuration with useLowLatency=1, useDinBuffer=0, useDoutBuffer=0

CCP read miss allocate with evict pipeline is shown in the pipeline diagram below, assuming the op-queue in datapipe is configured to be bypassable, so that the data array is accessed for read during P2 (otherwise the data array is accessed for read during P3). Tag/State/RP array read is set up during P0. Tag/State/RP array output appear during P1.

If multiple Banks are present, data output is muxed between different banks and registered to be used in P2. During cycle P2 Tag address compare is done. If **ctl\_op\_allocate\_p2** is set and no tag matches the address then miss is detected by the CCP logic. If no ways are invalid, CCP select a way to evict based on replacement policy. It asserts **cache\_evict\_valid\_p2** signal. It also schedules read for the bank on which read is to be performed. If this op-que is empty, then data array read can be set up in P2 cycle itself. Data array output appears in P3 and is written into a ECC self-correcting FIFO. The FIFO output is available during cycle P4.

When external logic is ready with the data to service the miss, it puts that data and ending state on fill port. Once this fill gets scheduled, tag is updated and fill data is written subsequently to data arrays.



Figure 5 CCP read miss with Evict pipeline

## CCP Write Hit Operation

For Configuration with useLowLatency=1, useDinBuffer=0, useDoutBuffer=0

CCP write hit pipeline is shown in the pipeline diagram below, assuming the op-queue in datapipe is configured to be bypassable, so that the data array is accessed for read during P2 (otherwise the data array is accessed for read during P3). Tag/State/RP array read is set up during P0. Tag/State/RP array output appear during P1.

If multiple Banks are present, data output is muxed between different banks and registered to be used in P2. During cycle P2 Tag address compare is done. If hit is detected, then write op is scheduled for banks on which write is to be performed. If this op-que is empty, then data array write can be set up in P2 itself. Data array output appears during P3. Based on strobes associated with the operation either Write or Read Modify Write is performed.

CCP itself doesn’t detect a write hit. It passes a read hit status to the external logic. External logic asserts **ctl\_op\_write\_data\_p2** signal if external logic wants to perform write of the data array for the matched index and way.



Figure 6 CCP write Hit pipeline

## CCP Write Miss with Evict Operation

For Configuration with useLowLatency=1, useDinBuffer=0, useDoutBuffer=0



Figure 7 CCP Write Miss with Evict pipeline

## CPP Tag pipeline resource usage

CCP tag array read access setup is always done in p0 cycle. CPP may decide to write tag array conditionally to a buffer based on assertion of **ctrl\_op\_tag\_state\_update\_p2,** this update is visible to subsequent transactions. CCP de-asserts **cache\_op\_ready\_p0** for the bank to which conditional write is performed. (tag memory is updated in this cycle).

Whenever tag array is updated, it is first temporarily written into a conditional buffer and **cache\_op\_ready\_p0** is de-asserted to perform the write in the next cycle.

Diagram below shows the micro-architecture of tag pipe:



Figure Tagpipe

Diagram below shows the tagpipe micro-architecture with two configurable pipeline stages P0+ and P1- controlled by parameter “UseTagRamInputFlop” and “UseTagRamOutputFlop”. P0+ stage is addressing to\_sram timing path. P1- stage is addressing from\_sram timine path assisted by clock skewing.



Figure Tagpipe with configurable pipeline stages P0+ adnd P1-

## CPP Tag pipeline replay(nack) conditions

Tagpipe asserts **cache\_nack\_p2** to a transaction if it cannot assign datapipe resources for the forward progress. Resources which can cause this condition to occur are

1. Op-queue full

Tagpipe also asserts **cache\_nack\_no\_allocate\_p2** to indicate when all ways are pending for any allocating transaction. External logic will take decision if it want to reissue a new transaction or make it a no allocate.

Following table lists how various nack conditions should be treated by external logic.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cache\_nack\_uce\_p2 | Cache\_nack\_ce\_p2 | Cache\_nack\_p2 | Cache\_nack\_no\_allocate | Comments |
| 1 | X | X | X | If uncorrectable error is set and no correctable error, Ignore everything else. Given transaction should return error |
| 0 | 1 | X | X | Correction will take place. Flush p1,p0 |
| 0 | 0 | 1 | X | Resource constraints. Retry |
| 0 | 0 | 0 | 1 | All ways pending, could not allocate |
| 0 | 0 | 0 | 0 | Transaction processed successfully |

Table CCP NACK Behaviors

When this expression is true:

cache\_nack\_ce\_p2 | cache\_nack\_uce\_p2 | cache\_nack\_p2 | (cache\_nack\_no\_allocate\_p2 & ctrl\_op\_allocate\_p2)

(1)

the following signals are ignored by CCP:

u.port('input',  'ctrl\_op\_allocate\_p2',         1);

u.port('input',  'ctrl\_op\_read\_data\_p2',        1);

u.port('input',  'ctrl\_op\_write\_data\_p2',       1);

u.port('input',  'ctrl\_op\_port\_sel\_p2',         1);

u.port('input',  'ctrl\_op\_bypass\_p2',           1);

u.port('input',  'ctrl\_op\_rp\_update\_p2',        1);

u.port('input',  'ctrl\_op\_tag\_state\_update\_p2', 1);

u.port('input',  'ctrl\_op\_burst\_len\_p2',        wBeats);

u.port('input',  'ctrl\_op\_ways\_busy\_vec\_p2',    nWays);

u.port('input',  'ctrl\_op\_ways\_stale\_vec\_p2',   nWays);

(2)

This signal is ignored by CCP only when the maintenance operation is targeting Data Array  (maint\_req\_array\_sel == 1’b1)

This signal is NOT ignored by CCP when the maintenance operation is targeting Tag Array (maint\_req\_array\_sel == 1’b0). I think this is a CCP RTL bug because the tag array is written if the maintenance operation opcode, which is a sideband signal to CCP, indicates a DEBUG WRITE.

u.port('input',  'ctrl\_op\_setway\_debug\_p2',     1);

(3)

These two signals are currently not ignored by CCP, and this is actually a CCP RTL bug because they cause side effects to CCP internal state machines. The work-around is that the external logic suppresses these two signals.

u.port('input',  'ctrl\_op\_rp\_update\_p2',        1);

u.port('input',  'ctrl\_op\_tag\_state\_update\_p2', 1);

## Tag pipe op-queue scheduler

CCP in P2 cycle decides to take any action based on a hit or miss would like to read data array or write to data array. This decision is made by tag pipe logic and written into an op-queue which decides ordering of data array accesses. Number of data banks is one of the configuration parameters for CCP. Each Bank has its own op-que and associated data pipe controller.

Every tag pipe operation determines which banks it needs to access and pushes operations it wants to perform into op-queues corresponding to banks.

Op-queue can be bypassed and data pipe controller can start doing its operation while opcode is being assigned (if op-que is empty)

Tag pipe has to schedule for transactions coming from control pipe or fills coming from fill port of Din buffer. Whenever fill data is available, tag pipe scheduler needs to schedule fill data and update the tag when fill data is scheduled.

Table below gives all possible combinations control signals coming from external logic which controls what is to be done with data pipe for that transaction. Table also lists opcode written to the datapipe op-queue for performing that operation.

Whenever **ctl\_op\_allocate\_p2** is asserted with these signals, it may trigger an eviction if no invalid ways are available. This is indicated to data pipe by setting **Evict** Field in the op-queue.

While doing this scheduling in P2 cycle, read response port ordering or evict response port ordering fifo is written if operation involves sending output data to one of these ports. Data written to this fifo is the bank to which the operations are issued. This fifo control evict port and read response port assignment to banks. This ensures that evict responses and read responses follow the order which is decided in cycle P2.

For Pure Bypass operations, Datapipe scheduler is bypassed and only ports are scheduled.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cache\_evict\_valid\_p2 | ctrl\_op\_read\_data\_p2 | ctrl\_op\_write\_data\_p2 | ctrl\_op\_port\_sel\_p2 | ctrl\_op\_bypass\_p2 | Description | Datapipe  Op-Queue | Evict | Port  Sched | DataPipeSched |
| Evict | Read data array | Write data array | Port select | Bypass |  |  |  |  |  |
| X | 0 | 0 | 0 | 0 | 0. No operation scheduled | - | No | No | No |
| X | 0 | 0 | 0 | 1 | 1. Bypass data on write port to read response port | - | No | Rbypass | No |
| X | 0 | 0 | 1 | 0 | 2. Illegal | - | - | - | - |
| X | 0 | 0 | 1 | 1 | 3. Bypass data on write port to evict port | NOP | No | EBypass | No |
| X | 0 | 1 | 0 | 0 | 4. Write the data on write port to the data array | WR | No | No | Yes |
| X | 0 | 1 | 0 | 1 | 5. write data to array and provide it on read response port | WT | No | R | Yes |
| X | 0 | 1 | 1 | 0 | 6. Write the data on write port to the data array. | WR | No | No | Yes |
| X | 0 | 1 | 1 | 1 | 7. write data to array and provide it on evict response port | WT | No | E | Yes |
| 0 | 1 | 0 | 0 | 0 | 8. Read data array and send it on read response port | RD | No | R | Yes |
| 1 | 1 | 0 | X | 0 | 9. Read data array and send it on evict port for eviction | NOP | Yes | EEvict | Yes |
| 0 | 1 | 0 | X | 1 | 10. Illegal | - | - | - | - |
| 1 | 1 | 0 | 0 | 1 | 11. Read Data Array for Eviction, Bypass data on write port to rdrsp port (data is read only for eviction) | NOP | Yes | RBypass ,EEvict | Yes |
| 0 | 1 | 0 | 1 | 0 | 12. Read data array and send it on evict port | RD | No | E | Yes |
| 1 | 1 | 0 | 1 | 1 | 13.Read Data Array for Eviction, Bypass data on write port to evict port (data is read only for eviction) Note: not used by DMI | NOP | Yes | EBypass,  EEvict | Yes |
| 0 | 1 | 1 | 0 | 0 | 18. Read data array and send it to RdRsp port | WR | No | R | Yes |
| 0 | 1 | 1 | 1 | 0 | 19. Read data array and send it to Evict port | WR | No | E | Yes |
| 1 | 1 | 1 | X | 0 | 15. Read Data Array for Eviction, Write the data from write port to the Data Array. | WR | Yes | EEvict | Yes |
| 1 | 1 | 1 | 0 | 1 | 16. Illegal | - | - | - | - |
| 1 | 1 | 1 | 1 | 1 | 17. Illegal | - | - | - | - |

Table CCP Command Operations

CCP Tagpipe processes lookup request in P0 cycle and update requests (state,nru,fill) in P2 cycle.

CCP Tagpipe reads Tag Memory in P0 cycle, and writes Tag Memory in P3 cycle.

State update and NRU update can be merged together by CCP Tagpipe if they happen in the same cycle.

State/NRU update always has higher priority than Fill update. But Fill update won't be starved for long

because of the nature of the R--W pipeline and the Lookup request backpressure logic; eventually

a time slot will naturally be freed up for Fill update. Fill has higher priority than Lookup.

A close-up of a document

Description automatically generated with low confidence

Figure CCP Tagpipe Operation Priority

If NRU is configured to have its own NRU memory, then NRU is not stored in TAG\_MEM.

In case state\_update/nru\_update collides with fill\_update, then state\_update/nru\_update

wins, and fill\_update is backpressured.

Note that state\_update has higher priority than nru\_update.

The nru info in state\_update and nru\_update is the same (has the same value).

Tagpipe Pipeline

|  |  |  |  |
| --- | --- | --- | --- |
| P0 | P1 | P2 | P3 |
| TAG\_MEM\_RD  ctrl\_op\_valid\_p0  mem\_read  i.e. access mem | TAG\_MEM\_RD\_DATA  mem\_read\_en  i.e. mem data avail | TAG\_MEM\_RD\_OUT  corrected  registered mem data  random\_arb\_waysel  bypass/forwarding  tag compare | TAG\_MEM\_WR  state\_way\_wren,  index\_p3,  state\_update\_bank\_busy |

Table CCP Tagpipe Pipeline

The timing diagram below shows the CCP cache\_nack\_ce\_p2 timing behavior. Note that CCP p2\_valid holds for 2 additional cycles (i.e. pipeline stalls for 2 cycles) when cache\_nack\_ce\_p2 is asserted.

A picture containing text, diagram, screenshot, line

Description automatically generated

Figure CCP cache\_nack\_ce\_p2 Timing Diagram

## Fill Op-Queue Scheduler

A fill data operation is scheduled to the datapipe by writing the fill data request to the fill op-queue. Whenever a fill op-queue is written, we need to check if any read or Evict operation for the same address is pending in the control op-queue. If it is pending, then fill should not be performed until this operation completes.

A fill control operation is scheduled to the tagpipe via a tag update state machine that arbitrates between the fill control request as well as the tag update request.

## Data Bank Selection

Each beat in the cache is uniquely addressed by

|  |  |  |
| --- | --- | --- |
| WayNum | IndexBits | BeatID |

Data bank to which operation is dispatched is selected based on configuration function which selects bits from [WayNum,IndexBits]

## CCP data flow



Figure 12 CCP Data Flow

Above figure explains how data flow through CCP works. Datapipe executes operations en-queued in the op-queue. While doing writes it consumes data on fill interface and write interface using valid ready handshake. If Operation results requires data array read access, it performs read and result is put on the port requested by the external logic. Eviction may be triggered depending on state of the internal logic and they are routed to evict port. The diagram below shows the Datapipe module.

## CCP datapipe scheduler



Figure Datapipe

Above figure explains high level view of Data Pipe Scheduler. Datapipe scheduler performs operations en-queued. It is responsible for sequencing all data array accesses, Error Detection, Correction.

### CCP datapipe pipe control – single cycle sram access

Diagram below describes the u-arch of module ccp\_datapipe\_pipe\_control when data ram is single cycle access.



Figure Datapipe with single cycle SRAM access

The CCP datapipe with 1 cycle sram runs as per the time slot reservation table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Data Memory | Output Port | Self Correcting Buffer | Pipeline Time Slot |
| P0 P1 P2 | P0 P1 P2 | P0 P1 P2 | P0 P1 P2 |
| READ | R | Y | WE | Y Y Y |
| WRITE | W |  |  | Y |
| WRITE\_THROUGH | W | Y |  | Y |
| READ\_MODIFY\_WRITE | R W |  | WE | Y Y Y |
| READ\_MODIFY\_WRITETHROUGH | R W | Y | WE | Y Y Y |
| MAINT\_READ | R |  |  | Y Y Y |
| MAINT\_WRITE | R W |  |  | Y Y Y |

Table CCP Datapipe Time Slot Reservation Table for 1 cycle SRAM

The CCP datapipe with **correctable errors** runs as time slot below:

READ:

|  |  |  |  |
| --- | --- | --- | --- |
| P0 | P1 | P2 | P2\_CORR\_WR |
| R | WE |  | corr W |

READ\_MODIFY\_WRITE:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P0 | P1 | P2 | P2\_CORR\_WR | P2\_READ\_MERGE |
| R | WE |  | corr W | merge W |

READ\_MODIFYWRITETHROUGH:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P0 | P1 | P2 | P2\_CORR\_WR | P2\_READ\_MERGE |
| R | WE |  | corr W | merge W |

The CCP datapipe with **uncorrectable** read modify write runs as time slot below:

READ\_MODIFY\_WRITE:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P0 | P1 | P2 | P2\_CORR\_WR | P2\_READ\_MERGE |
| R | WE |  | corr W | merge W + poison |

READ\_MODIFYWRITETHROUGH:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P0 | P1 | P2 | P2\_CORR\_WR | P2\_READ\_MERGE |
| R | WE |  | corr W | merge W + poison |

Error detection is in P2 stage. When read modify write/writethrough correctable error occurs, merge data write follows correctable data write. When read modify write/writethrough uncorrectable error occurs, dummy correction write occurs in P2\_d stage (P2\_CORR\_WR stage as above) followed by merge data + poison write in P2\_dd stage(P2\_READ\_MERGE stage as above).

Merge data + poison write used to happen in p2 stage and introduce timing violations because merge data selection logic depends on uncorrectable error detection in p2 stage.

Notes:

Acceptance into pipeline P0 cycle occurs on P0 valid ready handshake. Once accepted, make sure P0 can make forward progress to P1 and P2 (if P1 and P2 are needed for the opcode execution), or to P1 only (if only P1 is needed).

### CCP datapipe pipe control – 2-cycle sram

Diagram below describes the u-arch of module ccp\_datapipe\_pipe\_control when data ram is 2-cycle access. Configurable to-sram stage flop for timing purpose is highlighted in the diagram. When 2-cycle ram is used, to-sram stage flop must be turned on. When 1 cycle ram is used, to-sram stage flop is optional.



Figure Datapipe with 2 cycle SRAM access

The CCP datapipe with 2 cycle sram runs as per the time slot reservation table below:

P0/P1/P2/P3 stage flop toggles every 2 cycles.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Data Memory | Output Port | Self-Correcting Buffer | Pipeline Time Slot |
| P0 P1 P2 P3 | P0 P1 P2 P3 | P0 P1 P2 P3 | P0 P1 P2 P3 |
| READ | R | Y | WE | Y Y Y |
| WRITE | W |  |  | Y |
| WRITE\_THROUGH | W | Y |  | Y |
| READ\_MODIFY\_WRITE | R W |  | WE | Y Y Y Y |
| READ\_MODIFY\_WRITETHROUGH | R W | Y | WE | Y Y Y Y |
| MAINT\_READ | R |  |  | Y Y Y |
| MAINT\_WRITE | R W |  |  | Y Y Y Y |

Table CCP Datapipe Time Slot Reservation Table for 2 cycle SRAM

The CCP datapipe scheduler with **correctable errors** runs as per the time slot below:

READ:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P0 | P0\_d | P1 | P1\_d | P2 | P2\_d | P3 | P3\_d |
| R | R |  | WE | Err detect | Error enc | corr W | corr W |

READ\_MODIFY\_WRITE:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P0 | P0\_d | P1 | P1\_d | P2 | P2\_d | P3 | P3\_d | P4 | P4\_d |
| R | R |  | WE | Err detect | Error enc | corr W | corr W | merge W | merge W |

READ\_MODIFYWRITETHROUGH:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P0 | P0\_d | P1 | P1\_d | P2 | P2\_d | P3 | P3\_d | P4 | P4\_d |
| R | R |  | WE | Err detect | Error enc | corr W | corr W | merge W | merge W |

The CCP datapipe with **uncorrectable** read modify write runs as time slot below:

READ\_MODIFY\_WRITE:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P0 | P0\_d | P1 | P1\_d | P2 | P2\_d | P3 | P3\_d |
| R | R |  | WE | Err detect |  | Poison+merge | Poison+merge |

READ\_MODIFYWRITETHROUGH:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P0 | P0\_d | P1 | P1\_d | P2 | P2\_d | P3 | P3\_d |
| R | R |  | WE | Err detect |  | Poison+merge | Poison+merge |

When read modify write/writethrough uncorrectable error occurs, poison bit and merge data is encoded in P2 stage, and the actual write occurs in P3 stage.

### Pack & unpack for 2 beats per data ram bank

Diagram below describes the u-arch of pack & unpack logics for nBeatsPerBank == 2, which means 2 data beats are stored in the same data ram bank entry. In ideal case, the packing logic collects 2 beats of write data, or combines 2 beats of read operations, and packed as one data ram entry access before pushing it into pipe ctrl. The unpacking logic receives 2 beats data every other cycle and outputs 1 beat per cycle to the selected port. So, the average throughput is not degraded when data ram is half clocked. Whereas some facts are now different than the case with default 1 beat per bank setting:

* Operation valid is 2-bit wide to represent upper and lower half of the packed operation respectively, valid bits are passsing through the pipeline and be used for data unpacking.
* In non-ideal case, only 1 beat of operation is sent to pipe ctrl due to the last of transaction, head or tail of wrapping transaction, or write through read residue is an odd number of beats data.
* Triggering size of Read modify write is now 2 data beats (1 data ram entry) size. If the packed operation is accessing less than 2 full beats of data, RMW will be performed.
* Data poison is combined for the 2 packed beats, as only one poison bit for a data ram entry exists.
* The LSB of address is not used when accessing the data ram.

A picture containing graphical user interface

Description automatically generated

Figure Datapipe pack and unpack for 2 beats per Data RAM entry

### Sram clock for Data Ram

Diagram below describes the implementation of clock to sram that runs half the control block frequency. The idea is to gate off the sram clock when the pipe is inactive and only enable sram clock when there is sram access.

Diagram, schematic

Description automatically generated

Figure Data RAM running at half frequency

Option1: To generate half frequency clock for the SRAM, we control the enable to be asserted at most every other cycle, to produce a 25% duty cycle clock (clk\_cg). Then apply a second latch to extend the logic high level, therefore get the standard 50% duty cycle and half input frequency clock (clk\_o). However, the second latch’s clk-to-q delay will introduce a skew to clk\_o, so appropriate delay is necessary to be added in SRAM input pins,Otherwise clk\_o may not catch the input request correctly.

Diagram

Description automatically generated

Figure Timing Diagram for Data RAM at half frequency

Option2: If the SRAM accept a 25% duty cycle clock, the second latch in the above figure can be removed. In this case the delay is not necessary for SRAM input pins. But the big assumption here is whether the slow SRAM can access a fast-toggling clock.

Diagram

Description automatically generated

Figure Data RAM at half fequency with 25% clock duty cycle

Option3: To avoid option1’s extra effort to add delay in backend flow, another option is feasible to assert the enable 1 fast cycle earlier, so skewed clk\_o will still have enough time to catch input information. Drawback of this option is the setup time budget is halved, now only 1 fast cycle time is available to meet setup time instead of 2 in the first 2 options.

Diagram

Description automatically generated

Figure Data RAM at half frequency with early enable

### Datapipe supported operations

NOTE: These opcodes are issued by Tagpipe. Operation for Atomics is Write(WR). The atomic opcodes are supplied via a sideband signal from the Tagpipe to the Datapipe, and are passed along to the per-bank Datapipe Scheduler. Atomics is considered as Write(WR) partial at the request level, and is decomposed to Read-Modify-Write at the data beat level.

|  |  |
| --- | --- |
| Operation | Description |
| Read(RD) | read of the data array |
| Write(WR) | When all strobes are set, issues a write to array. If beat-wise partial, read modify write is done |
| WriteThrough(WT) | It behaves like a write for data array. Also sends data on evict port. |
| NOP(NOP) | Do nothing |
| MaintenanceRead | Used for data array maintenance read operation |
| MaintenanceWrite | Used for data array maintenance write operation |

Table Datapipe Opcodes

### Control Op Queue data structure

|  |  |
| --- | --- |
| Field | Description |
| Operation | One of the operation defined above |
| Evict | If eviction needs to be perform before operation |
| Port | 1. ReadRsp 2. Evict |
| Bank Start Address | Address from which operation starts |
| Operation burst length | How many operations in wrap mode |
| Is Invalidate | If operation is invalidation, then correction is not required. |

Table Datapipe Op-queue Data Structure

The per-bank Datapipe Scheduler translates the Operation from the Control Op Queue into the following per-data-beat micro-operation:

|  |  |
| --- | --- |
| Micro-Operation (per data beat) | Description |
| ctrl\_read | To support Operation.Read.  To support Evict that needs to be performed before Operation (i.e. when evict\_active=1).  To support Operation.WriteThrough when write\_through\_read\_residue=1 (see Notes below). |
| ctrl\_write | To support Operation.Write when the data beat’s byte enable is all ones AND evict\_active=0. |
| ctrl\_read\_modify\_write | To support Operation.Write when the data beat’s byte enables is NOT all ones, AND evict\_active=0. |
| ctrl\_write\_through | To support Operation.WriteThrough when the data beat’s byte enable is all ones AND evict\_active=0 AND write\_through\_read\_residue=0. |
| ctrl\_read\_modify\_writethrough | To support Operation.WriteThrough when the data beat’s byte enable is NOT all ones, AND evict\_active=0 AND write\_through\_read\_residue=0. |
| ctrl\_maint\_read | To support Operation.MaintenanceRead. |
| ctrl\_maint\_write | To support Operation.MaintenanceWrite. |

Table Datapipe Bank Per-Data-Beat Micro Operation

The per-bank Datapipe Scheduler also generates the following per-data-beat control signals:

|  |  |
| --- | --- |
| Control signal (per data beat) | Description |
| ctrl\_port | If evict\_active=1, then Evict port is selected.  If evict\_active=0, then the value comes from Port. |
| ctrl\_corr\_needed | If evict\_active=1, then the value is 0 i.e. no correction needed.  If evict\_active=0, then the value is 1 i.e. correction needed. |
| ctrl\_last | Indicates the last data beat.  For evict\_active, it’s the last data beat in the evicted cacheline.  For write through, it’s the last data beat in the evicted cacheline, otherwise it’s the last data beat in burst request as specified by the burst length.  For NOP, it’s always true. |

Table Datapipe Bank Per-Data-Beat Control signals

Notes:

For Operation.WriteThrough, if the burst length is less than nBeats i.e. number of data beats in a cacheline, then it is executed as Write Through (to burst length) followed by Write Through Read Residue (to nBeats). Write Through writes to the Data Array and also sends write data to Evict/RdRsp Port, whereas Write Through Read Residue reads from Data Array and also sends read data to Evict Port/RdRsp Port.

### Fill Op Queue data structure

|  |  |
| --- | --- |
| Field | Description |
| Bank Start Address | Address from which operation starts |

Table Fill Op Queue Data Structure

## Datapipe Operation pipeline



Figure 21 Data array Read Operation with correctable errors



Figure 22 Data array Evict Write Operation.



Figure Scratchpad Data Array Read Operation (useLowLatency=1)



Figure Scratchpad Data Array Write Operation

## Evict and RdRsp Ports

CCP implements only one Evict port and one RdRsp port.

## Memory Sleep

CCP provides a “busy” status bit for each tag bank memory and each data bank memory. CCP asserts the “busy” bit when the pipeline and the input queues and the state machines associated with the bank memory is busy. The external logic provides a “sleep” request bit for each tag bank memory and each data bank memory. When the “busy” status bit is de-asserted and the “sleep” request bit is asserted, that means the external logic can assert the light sleep pin “LS” to the bank memory; and it also means CCP stops accepting any external request coming in by de-asserting the associated “ready” output signal. CCP provides a “wakeup” request bit for each tag bank memory and each data bank memory. CCP asserts the “wakeup” request bit when there is an external request waiting for accessing the memory. When the “wakeup” request bit is asserted and the “LS” pin is asserted, that means the external logic can de-assert the “LS” pin and then in the following clock cycle de-assert the “sleep” request bit.

## Debug and Error Insertion Capabilities

None.

## Performance Counters and Status Capabilities

None.

## Engineering Registers

None.

## Control and Status Registers

CCP does not implement any CSR registers.

# Verification Corner Cases

Datapipe

-index\_block

-bank\_used\_for\_write

-read\_bypass\_possible

-err\_detected

-error\_correction\_write

-Pipeline stall

-datapipe\_table\_cond[13]: Read Data Array for Eviction and Bypass data on write port to evict port (data is read only for eviction).

Tagpipe

-Fill update collides with State update/NRU update

-Bypass memory data out by forwarding previous P2 nru,state,tag in the pipeline to current P2 nru,state,tag WHEN P2 bank and index addresses match previous P2

-Bypass Fill update’s tag and state to State update

-Pipeline flush due to cache\_nack\_ce\_p2

-rp\_update\_all\_ways, all\_ways\_inuse, all\_ways\_busy

# Performance Requirements

Frequency 🡪 1.2GHz

Technology 🡪 32FF+

Cell Type 🡪100% RVT

Corner 🡪 p72V125C

Clock gating 🡪 98%

Area 🡪 Not specified

# Appendix A – Scratchpad

## Scratchpad microarchitecture diagram

The existing CCP Datapipe microarchitecture is extended to support Scratchpad. Note that Scratchpad interfaces directly to the CCP Datapipe, and completely bypasses the CCP Tagpipe.

Tagpipe

OpQ IF

CCP

Write IF

CCP

Fill IF

Scratchpad

Ctrl IF

Scratchpad

Write IF

Control

queue

Write

queue

Fill

queue

Control

queue

Write

queue

MUX

ARB

Datapipe

Bank

Pipeline

Controller

Evict

FIFO

RdRsp

FIFO

nBanks

CCP Datapipe Bank

CCP Dout

CCP

Evict

IF

CCP

RdRsp

IF

Write

control

FIFO

Write

bypass

FIFO

Bank

Select

Bank

Select

MUX

ARB

Write

control

FIFO

Scratchpad

RdRsp

IF

Scratchpad

Dout

Figure Scratchpad microarchitecture diagram

# Appendix B – CCP Memory Storage Structure

**Data Bank Addressing**

A cacheline is stored in a Data Bank.

A cacheline can have 1 beat (for 512-bit data width), 2 beats (for 256-bit data width), 4 beats (for 128-bit data width), or 8 beats (for 64-bit data width).

Each entry in the Data Bank stores a beat of a cacheline, with a poison bit, plus ECC/parity bits computed for the beat with the poison bit.

The organization of the Data Bank entry is:

|  |  |  |
| --- | --- | --- |
| poison bit  (width=1) | data beat of a cacheline  (width=wDataBeat=512 or 256 or 128 or 64) | optional ECC/parity bits  (width=wErrorBits) |

Each entry in the Data Bank is uniquely addressed by Data Bank Index that comprises:

|  |  |  |
| --- | --- | --- |
| WayNum  (width = log2ceil(nWays)) | IndexBits  (width = PriSubDiagAddrBits.length - DataBankSelBits.length) | BeatNum  (width=wBeat) |

wBeat = 0 (for wDataBeat=512), 1 (for wDataBeat=256), 2 (for wDataBeat=128), 3 (for wDataBeat=64)

Data Bank Index is calculated from the cacheline byte address bits, based on the parameters "PriSubDiagAddrBits", "SecSubRows", "wAddr", "DataBankSelBits".

Data Bank Select is generated from picking the bits from the Data Bank Index.

width for Data Bank Select = u.getParam('DataBankSelBits').length

**Tag Bank Addressing**

The Tag Bank is an N-way Set-Associative structure.

Each entry in the Tag Bank stores multiple ways of cacheline state with optional NRU bit, plus optional ECC/parity bits computed for the cacheline state with optional NRU bit.

NRU bit is only stored in Tag Bank when RepPolicy=NRU and nRPPorts=1.

The organization of the Tag Bank entry is:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| optional security bit of  Way N-1  (width=1) | cacheline tag of  Way N-1  (width=  wTagEntry) | cacheline state of  Way N-1  (width=2) | optional NRU bit  of Way N-1  (width=1) | optional ECC/parity bits  of Way N-1  (width=  wErrorBits) | ...... | optional security bit of  Way 0  (width=1) | cacheline tag of  Way 0  (width=  wTagEntry) | cacheline state of  Way 0  (width=2) | optional NRU bit  of Way 0  (width=1) | optional ECC/parity bits of  Way 0  (width=  wErrorBits) |

wTagEntry = wAddr - wCachelineOffset - PriSubDiagAddrBits.length - PortPriSubDiagAddrBits.length + log2ceil( Math.pow(2, PortPriSubDiagAddrBits.length) / num\_ports )

Each entry in the Tag Bank is uniquely addressed by Tag Bank Index that comprises:

|  |
| --- |
| IndexBits  (width = u.getParam('PriSubDiagAddrBits').length - u.getParam('TagBankSelBits').length) |

Tag Bank Index is calculated from the cacheline byte address bits, based on the parameters "PriSubDiagAddrBits", "SecSubRows", "wAddr", "TagBankSelBits".

Tag Bank Select is generated from picking the bits from Tag Bank Index.

width for Tag Bank Select = u.getParam('TagBankSelBits')

**RP Bank Addressing**

NRU bit is stored in RP Bank when RepPolicy=NRU and nRPPorts=2. RP Bank is dual port, one read, one write.

RP Bank addressing is the same as Tag Bank addressing.

**ECC Encoding Scheme**

The ECC Encoding scheme, represented by the parameter "fnErrDetectCorrect" of type string, can be one of the following:

NONE, PARITYENTRY, PARITY16BITS, PARITY8BITS, SECDED, SECDED64BITS, SECDED128BITS.

The ECC encoding schemes SECDED64BITS and SECDED128BITS support the optional parameter blockWidths which is an array of widths, where the sum of widths is the data width. Effectively the data is subdivided into multiple pieces, each piece has its ECC bits. See <https://jira.arteris.com/browse/AR-470>

**ECC Encoder**

The ECC Encoder has one input and one output:

u.input("data\_in", wData);

u.output("data\_out\_with\_edc", wData + wErrorBits);

| **Error Encoding Scheme**  **(fnErrDetectCorrect)** | **Error Bits Width (wErrorBits),**  **ECC encoder output (data\_out\_with\_edc)** |
| --- | --- |
| NONE | //!! **wErrorBits = 0;**  **data\_out\_with\_edc = data\_in**; |
| PARITYENTRY | //!! **wErrorBits** = 1;  parity\_bit = [data\_in].reduceXor; **data\_out\_with\_edc = [data\_in, parity\_bit].concat;** |
| PARITY16BITS,  PARITY8BITS | //!! if (fnErrDetectCorrect === "PARITY16BITS") { //!! resolution = 16; //!! } else if (fnErrDetectCorrect === "PARITY8BITS") { //!! resolution = 8; //!! }  //!! **wErrorBits = Math.ceil(wData/resolution);**  //!! for (var i=0; i < wErrorBits; i++) { //!! var high\_bit = ''; //!! if (i == wErrorBits-1) { //!! high\_bit += wData-1; //!! } else { //!! high\_bit += (i \* resolution) + (resolution-1); //!! } //!! var low\_bit = (i\*resolution); parity\_bit[$i$] = [data\_in[$high\_bit$, $low\_bit$]].reduceXor; //!! } **data\_out\_with\_edc = [** **data\_in,** **//!! for (var i=wErrorBits-1; i >= 0; i--) {** **parity\_bit[$i$],** **//!! }** **].concat;** |
| SECDED,  SECDED64BITS,  SECDED128BITS | //!! var numInst; //!! var wErrorAndParityBits = []; //!! var wDataAndErrorBits = []; //!! var wErrorBits = 0;  // calculate numInst and wInstData[] //================================== //!! if (fnErrDetectCorrect === "SECDED") { //!! numInst = 1; //!! wInstData[0] = wData; //!! } else if (blockWidths) { // i.e. SECDEC64BITS/SECDED128BITS with blockWidths[] //!! numInst = blockWidths.length; //!! wInstData = blockWidths; //!! } else { // i.e. SECDED64BITS/SECDED128BITS //!! var resolution; //!! if (fnErrDetectCorrect === "SECDED64BITS") { //!! resolution = 64; //!! } else if (fnErrDetectCorrect === "SECDED128BITS") { //!! resolution = 128; //!! } //!! numInst = Math.ceil(wData / resolution); //!! wInstData = []; //!! for (var inst = 0; inst < numInst; inst++) { //!! if ((resolution \* (inst + 1)) > wData) { //!! wInstData[inst] = wData % resolution; //!! } else { //!! wInstData[inst] = resolution; //!! } //!! } //!! }  // calculate wDataAndErrorBits[] and wErrorBits // ============================================ //!! for (var inst = 0; inst < numInst; inst++) { //!! if (wInstData[inst] === 1) { //!! wErrorAndParityBits[inst] += 3; //!! } else if (wInstData[inst] === 2) { //!! wErrorAndParityBits[inst] += 4; //!! } else { //!! **wErrorAndParityBits[inst] += Math.ceil(Math.log2(wInstData[inst] + Math.ceil(Math.log2(wInstData[inst])) + 1)) + 1;** //!! } //!! wDataAndErrorBits[inst] = wInstData[inst] + wErrorAndParityBits[inst]; //!! **wErrorBits += wErrorAndParityBits[inst];** //!! }  // secded calculation starts here // ============================== //!! var low\_bit; //!! var high\_bit = - 1; //!! for (var inst = 0; inst < numInst; inst++) { //!! low\_bit = high\_bit + 1; //!! high\_bit = low\_bit + wInstData[inst] - 1; data\_in$inst$ = data\_in["high\_bit", "low\_bit"]; //!! }  // set the width // ============= err\_bit$inst$ = '0'.d("wErrorBits[inst]"); data\_out\_with\_edc\_int$inst$ = '0'.b().repeat("wDataAndErrorBits[inst]");  // place data bits // =============== //!! var low\_bit = 0; //!! var high\_bit = -1; //!! var low\_bit\_out; //!! var high\_bit\_out; //!! for (var i = 1; i < wErrorBits[inst]; i++) { //!! low\_bit = high\_bit + 1; //!! high\_bit = low\_bit + (Math.pow(2, i) - 2); //!! high\_bit\_out = Math.pow(2, i + 1) - 1; //!! low\_bit\_out = Math.pow(2, i) + 1; //!! if (high\_bit > wInstData[inst] - 1) high\_bit = wInstData[inst] - 1; //!! if (high\_bit\_out > wDataAndErrorBits[inst] - 1) high\_bit\_out = wDataAndErrorBits[inst] - 1; data\_out\_with\_edc\_int$inst$[$high\_bit\_out$, $low\_bit\_out$] = data\_in$inst$[$high\_bit$, $low\_bit$]; //!! }  // calculate parity bits // ===================== //!! for (var i = 0; i < wErrorBits[inst]; i++) { err\_bit$inst$[$i$] = [ //!! for (var j = 1; j < wDataAndErrorBits[inst]; j++) { //!! if ((j & Math.pow(2, i)) == Math.pow(2, i)) { data\_out\_with\_edc\_int$inst$[$j$], //!! } //!! } ].reduceXor; //!! }  // calculate line parity // ===================== parity\_bit$inst$ = [data\_out\_with\_edc\_int$inst$, err\_bit$inst$].reduceXor;  // final output // ============ **data\_out\_with\_edc = [** **//!! for (var inst = numInst - 1; inst >= 0; inst--) {** **data\_in$inst$,** **//!! }** **//!! for (var inst = numInst - 1; inst >= 0; inst--) {** **err\_bit$inst$,** **//!! }** **//!! for (var inst = numInst - 1; inst >= 0; inst--) {** **parity\_bit$inst$,** **//!! }** **].concat;** |

# Appendix C – Cacheline Replacement and Insertion

CCP supports three replacement policies, RANDOM, NRU and PLRU, for cacheline replacement (i.e. way selection for eviction) and cacheline insertion (i.e. way selection for insertion).

For Policy=RANDOM:

if all ways are in use, then randomly select a valid way that's not busy,

else just select an invalid way.

For Policy=NRU:

if all ways NRU bits are set, then randomly select a valid way that's not busy,

else if all ways are in use, then randomly select a way that has current NRU bit clear that's not busy,

else just select an invalid way.

CCP can be configured to support a form of not-recently-used (NRU) replacement policy. Each set is represented by an *n*-bit NRU vector, where *n* equals the number of ways in the cache. On a read hit or write hit, when the external logic asserts ctrl\_op\_rp\_update\_p2, the corresponding bit in the NRU vector is set. For read miss allocate or write miss allocate, external logic can also choose to assert ctrl\_op\_rp\_update\_p2 to set the corresponding bit in the NRU vector. If all bits in the vector are set after the bit corresponding to the recently accessed way is set, the bits in the NRU vector corresponding to the not recently accessed ways are cleared.

To select a replacement way for cacheline allocation, an invalid way is selected preferentially. If no invalid way is available, a way is selected from the cleared bits in the NRU vector using an arbiter whose priority is set by a counter that increments from 0 to *n*-1 (or an *n*-bit one-hot shift register that shifts) every cycle. If no cleared bits are available, the replacement way is selected from the way indicated by the counter.

While a way is selected for allocation and the protocol transaction has not completed, the way is removed from the selection process. Once the protocol transaction completes and the entry has been filled, the corresponding NRU vector bit is set.

Note: A fill bypasses the cache replacement policy module and unconditionally sets the NRU bit that corresponds to the fill way to 1. After a fill, all bits in the NRU vector are allowed to be set. The bits will be cleared by the next hit to a cacheline in the set.

Random policy can be considered as a special case of above mentioned policy when all NRU bits are set.

CCP implements a simple linearly incrementing counter that’s used as the parking point for round-robin arbitration to emulate “randomly select a way.” The round-robin arbitration starts in the reverse direction, starting from the one behind the parking point.

CCP Way Replacement module that generates the NRU bits doesn't know (and doesn't need to know) if it's hit or fill.

The CCP NRU description as captured in the CCP uarch spec, Appendix C, was written by viewing the CCP and its external control logic as a whole.

The NRU bits generated by the CCP Way Replacement module is this logic expression:  
assign policy\_info\_out = (all\_ways\_nru\_setting | all\_ways\_nru\_set) ? (matchvec | alloc\_vec) : (policy\_info\_in | matchvec | alloc\_vec);

where

all\_ways\_nru\_set = &{ways\_busy | (policy\_info\_in & ways\_valid)};

all\_ways\_nru\_setting = &{ways\_busy | (policy\_info\_in & ways\_valid) | matchvec | alloc\_vec};

matchvec indicates hit (i.e. valid & non-stale tag matching vector in P2 cycle, one bit per way)  
alloc\_vec indicates miss allocate (way allocation vector in P2 cycle, one bit per way)  
policy\_info\_in indicates the NRU bits read from memory storage.

Let's say there are 3 ways. In the beginning, NRU bits are all clear. Assumes external logic only asserts “ctrl\_rp\_update\_p2” for read hit or write hit.  
Initial condition => NRU = 3'b00  
Read Miss Allocate to Way 0 => NRU = 3'001 (Fill sets Way 0 bit)  
Read Miss Allocate to Way 1 => NRU = 3'011 (Fill sets Way 1 bit)  
Read Hit to Way 0 => NRU = 3'011 (Hit sets Way 0 bit)  
Read Miss Allocate to Way 2 => NRU = 3'b111 (Fill sets Way 2 bit)

Read Hit to Way 2 => NRU = 3’b100

For Policy=PLRU (Pseudo Least Recently Used)

Each set is represented by an (*n*-1) bit PLRU vector, where *n* equals the number of ways in the cache. On a miss allocate for allocating request, an invalid way that is not busy is selected preferentially, otherwise a valid way that is not busy is selected; also, the PLRU state bit vector is updated accordingly. On a hit, the hit way is marked as Most Recently Used and the PLRU state bit vector is updated accordingly.

A dual port (1 read port, 1 write port) SRAM per Tag Bank is implemented.

In P0 cycle, the PLRU SRAM is read.

In P1 cycle, the PLRU SRAM has the read data available and is captured in flops.

In P2 cycle, the captured PLRU value is fed to the PLRU module that computes the next state and the victim way. The result is captured in flops.

In P3 cycle, the result that is captured in flops is written to the PLRU SRAM.

Forwarding logic is implemented to ensure the PLRU computation always uses the latest PLRU value.

The Pseudo-LRU module implements a binary decision tree to select a least-recently-used way for replacement for an N-way set associative cache. Each node in the binary decision tree carries a one-bit value. The value determines if the left branch is to be taken, or the right branch is to be taken. A binary decision tree can be trimmed by removing the invalid leaf nodes by overriding the values in the associated nodes that lead to the invalid leaf nodes.

The Pseudo-LRU module implements two functions, one for cache miss allocate, another for cache hit (i.e. line access). The first function is to find the victim way that is Least Recently Used for **way allocation** and generate the next PLRU state bits. The second function is to mark the way being accessed (i.e. hit way) as Most Recently Used and generate the next PLRU state bits.

A diagram of a computer program

Description automatically generated

Figure Pseudo LRU block diagram

The override block creates the effective state by overriding the associated binary decision nodes in the current state based on the invalid ways (invalid ways are basically the inverse of valid ways). Effectively the binary decision tree is trimmed so that invalid ways are removed; what’s left are valid ways.

The first LUT block uses the override state to look up a table to find the victim way.

A MUX selects the victim way or the hit way. The output of the MUX goes to the second LUT block.

The second LUT block finds the next PLRU state bits.

The final victim way is forced to zeros when the valid ways are all zeros, or hit way is not all zeros.

The truth table for the pseudo-LRU implementation - per number of ways – is listed below.

**'x' means don't care.**

**'\_' means unchanged.**

An example for a 4-way set associative Pseudo LRU is shown as below:

**// state[0]**

**// / \**

**// 0 1**

**// / \**

**// state[1] state[2]**

**// / \ / \**

**// 0 1 0 1**

**// | | | |**

**// way way way way**

**// 0 1 2 3**

**//**

**// lut**

**// -------------------- -------------------**

**// state[2:0] | victim ref to | next state**

**// -----------+-------- -------+-----------**

**// x0 0 | way 0 way 0 | \_1 1**

**// x1 0 | way 1 way 1 | \_0 1**

**// 0x 1 | way 2 way 2 | 1\_ 0**

**// 1x 1 | way 3 way 3 | 0\_ 0**

**//**

**// override**

**// ------------------------------------------------**

**// state[0] = valid\_ways[1:0] == 00 ? 1 :**

**// valid\_ways[3:2] == 00 ? 0 : \_;**

**// ------------------------------------------------**

**// state[1] = valid\_ways[1:0] == 00 ? \_ :**

**// valid\_ways[0] == 0 ? 1 :**

**// valid\_ways[1] == 0 ? 0 : \_;**

**// ------------------------------------------------**

**// state[2] = valid\_ways[3:2] == 00 ? \_ :**

**// valid\_ways[2] == 0 ? 1 :**

**// valid\_ways[3] == 0 ? 0 : \_;**

**// ------------------------------------------------**

The initial state is 3’b000.

If all ways are valid, then the input state is unmodified, i.e. the effective state is the same as input state.

The effective state is used to look up the getVictim table as shown above.

If state == 3’bx00, then way 0 is selected as victim way, and next state will be 3’b011 pointing to way 2.

If state == 3’bx10, then way 1 is selected as victim way.

If state == 3’b0x1, then way 2 is selected as victim way.

If state == 3’b1x1, then way 3 is selected as victim way.

The next time around way 0 is invalid (say way 0 is busy).

A successive 4 operations will produce the following sequence: way 0, way 2, way 1, way 3.

NOTES:

If way 0 is invalid, then way 0 is removed from the tree, and the binary branching point for node 1 is forced to 1, to steer the decision to the right branch.

|  |
| --- |
| **// state[0]**  **// / \**  **// 0 1**  **// / \**  **// state[1] state[2]**  **// forced |**  **// to 1 |**  **// \ / \**  **// 1 0 1**  **// | | |**  **// way way way**  **// 1 2 3** |

Given the input state = 3’b011, the effective state would then be 3b0**1**1. Then way 2 will be picked as victim way.

If way 1 is invalid, then way 1 is removed from the tree, and the binary branching point for node 1 is forced to 0, to steer the decision to the left branch.

|  |
| --- |
| **// state[0]**  **// / \**  **// 0 1**  **// / \**  **// state[1] state[2]**  **// forced |**  **// to 0 |**  **// / / \**  **// 0 0 1**  **// } | |**  **// way way way**  **// 0 2 3** |

# Appendix D – Issues

So far we have found 6 CCP RTL functional bugs (not performance bugs) after NCore 2.2 shipped on Oct 4, 2017.

These RTL bugs apply to both the ACHL version of CCP (the original CCP) and the TACHL version of CCP (the current CCP, ported over from ACHL version of CCP, plus additional features. See P/S).

(1)

12 Dec 2017

<https://jira.arteris.com/browse/CONC-3812>

<https://jira.arteris.com/browse/CONC-3560>

Fix CONC-3812, CONC-3560 related to CCP NRU bit causing unexpected way selected for eviction

Both configurations: separate NRU memory (dual port), or NRU-bit in tag memory (single-port).

RTL fix:

<http://gitlab.arteris.com/hardware/hw-ccp/commit/6049da7dc501c178ee6132f281366b47bbaf6494>

NOTE: This RTL fix is in Ncore 2.5.

(2)

07 Mar 2017

<https://jira.arteris.com/browse/AR-86>

CCP WriteMissAllocatePartial with Eviction doesn’t work (ctrl\_op\_write\_data\_p2=1, ctrl\_op\_read\_data\_p2=1, ctrl\_op\_bypass=0, cache\_evict\_valid\_p2=1).

RTL fix:

<http://gitlab.arteris.com/hardware/hw-ccp/commit/404709071f75b00e42f4df855fde4605df4eea04>

NOTE: This bug doesn’t affect Ncore 2.X NCB and DMI.

(3)

24 May 2018

<https://jira.arteris.com/browse/AR-291>

Basically there's a Maintenance Request accepted into CCP Pipeline P0 cycle followed immediately by a AXI Request accepted into CCP Pipeline P0 cycle.

CCP detects a correctable error for the Maintenance Request in P2 cycle, but CCP doesn't assert cache\_nack\_ce\_p2.

RTL fix:

<http://gitlab.arteris.com/hardware/hw-ccp/commit/e1ed20c2ff3c0880263015b3801a24b45dcc7e34>

NOTE: This bug doesn’t affect Ncore 2.X NCB and DMI.

(4)

07 Jun 2018

<https://jira.arteris.com/browse/AR-352>

Error reporting on evict data error. RTL is doing error correction for eviction, but not logging the error for eviction.

RTL fix:

<http://gitlab.arteris.com/hardware/hw-ccp/commit/ed93616768387d08799cfe669ec752feda57932d>

(5)

09 Aug 2018

<https://jira.arteris.com/browse/CONC-4253>

CCP Write Bypass won’t work when Write Allocate sees a NACK No Allocate

RTL fix: pending.

NOTE: This bug doesn’t affect Ncore 2.X NCB and DMI.

(6)

23 Aug 2018

<https://jira.arteris.com/browse/CONC-4280>

CCP writes tag to wrong place when nack\_p2 and rp\_update\_p2.

2 consecutive read hits, causing "ctrl\_op\_rp\_update\_p2" to 2 different indices. The 2nd hit gets a "cache\_nack\_p2". The configuration must be NRU stored in Tag memory, nWays > 1.

RTL fix: pending.

(7)

When this expression is true:

cache\_nack\_ce\_p2 | cache\_nack\_uce\_p2 | cache\_nack\_p2 | (cache\_nack\_no\_allocate\_p2 & ctrl\_op\_allocate\_p2)

ctrl\_op\_setway\_debug\_p2 is currently NOT ignored by CCP when the maintenance operation is targeting Tag Array (maint\_req\_array\_sel == 1’b0 && maint\_req\_opcode == 4'b1110 )

“ctrl\_op\_rp\_update\_p2” and “ctrl\_op\_tag\_state\_update\_p2” are currently not ignored by CCP, and this is actually a CCP RTL bug because they cause side effects to CCP internal state machines. The work-around is that the external logic suppresses these two signals.

(8) This is a performance bug.

<https://jira.arteris.com/browse/CONC-3850>

CCP data bank request muxarb, when arbitrating a read/write request and a fill request, accepts the read/write request into pipeline without checking output port availability, thus blocking the fill request.

NOTE: This issue causes the following problems to the external logic in Ncore unit that drives the CCP:

<https://jira.arteris.com/browse/CONC-3258>

NCBU causing system to hang

<https://jira.arteris.com/browse/CONC-5774>

NCB is responding Reads in interleaved fashion below the cacheline boundary

(9)

4 Dec 2019

<https://jira.arteris.com/browse/AR-501>

Debug read or write for cache maintenance operation is longer than necessary.

When Codacache generates cache maintenance debug read or write, the command on the target memory is repeated nBeat times instead of 1 time only.

(10)

17 Jan 2020

<https://jira.arteris.com/browse/CONC-5974>

mntop\_active goes low before Maintenance Debug Write Data is captured into the CCP Datapipe pipeline