

# High-Performance, Multi-Requester, Out-of-Order, L1 Dcache (HPDcache)

---

Author:	César Fuguet
Release date:	February, 2023

---

This document version is 1.0.0-draft

Copyright © 2023

*Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA)*

Contributors to all versions of this specification document:  
Cesar Fuguet, Adrian Evans, Tanuj-Kumar Khandelwal, Nicolas Perbost.

---

# Contents

<b>1</b>	<b>Feature Specification</b>	<b>13</b>
1.1	List of features	15
1.2	Synthesis-time (static) Configuration Parameters	16
<b>2</b>	<b>Interfaces and Communication Protocols</b>	<b>19</b>
2.1	Global Signals	20
2.2	Requesters' Request/Response Interface	21
2.2.1	Signal Descriptions	21
2.3	Memory Request/Response Interfaces	22
2.3.1	Signal Descriptions	22
2.4	Interfaces' requirements	25
2.4.1	Valid/ready handshake process	25
2.4.2	Address, data and byte enable alignment	27
2.5	Requesters interface attributes	29
2.5.1	Type of operation	29
2.5.2	Source identifier	29
2.5.3	Transaction identifier	29
2.5.4	Cacheability	30
2.5.5	Need response	30
2.5.6	Error response	30
2.6	Memory interface attributes	31
2.6.1	Type of operation	31
<b>3</b>	<b>Architecture</b>	<b>35</b>
3.1	Cache Controller	36
3.1.1	On-Hold Requests	37
3.1.2	Memory Consistency Rules (MCRs)	38
3.2	Miss Handler	38
3.2.1	Multiple-entry Miss Status Holding Register (MSHR)	39
3.3	Uncacheable Handler	40
3.4	Cache Maintenance Operation (CMO) Handler	40
3.5	Cache Directory and Data	40
3.5.1	RAM Organization	40
3.5.2	Example cache data/directory RAM organization	41
3.6	Replay Table (RTAB)	42
3.6.1	RTAB integration in the cache	45

3.6.2	Policy for taking new requests in the data cache . . . . .	45
3.6.3	Possible improvements for the RTAB integration. . . . .	46
3.7	<b>Write-buffer</b> . . . . .	46
3.7.1	Memory Write Consistency Model . . . . .	46
3.7.2	Functional Description . . . . .	47
3.8	<b>Cache-coherency</b> . . . . .	47
4	<b>Configuration-and-Status Registers (CSRs)</b> . . . . .	49
4.1	<b>Dedicated CSR address space</b> . . . . .	50
4.2	<b>Configuration registers</b> . . . . .	51
4.3	<b>Performance counters.</b> . . . . .	52
4.4	<b>Event signals.</b> . . . . .	53
5	<b>Cache Management Operations (CMOs)</b> . . . . .	55
5.1	<b>Memory write fence</b> . . . . .	57
5.2	<b>Invalidate a cacheline by its physical address</b> . . . . .	58
5.3	<b>Invalidate a group of cachelines by their a set and way</b> . . . . .	59
5.4	<b>Invalidate the entire cache</b> . . . . .	60
5.5	<b>Prefetch a cacheline given its physical address</b> . . . . .	61
6	<b>Atomic Memory Operations (AMOs)</b> . . . . .	63
6.1	<b>Background</b> . . . . .	64
6.2	<b>Supported Atomic Memory Operations (AMOs)</b> . . . . .	64
6.3	<b>Implementation</b> . . . . .	64
6.4	<b>AMO ordering</b> . . . . .	65
6.5	<b>LR/SC support.</b> . . . . .	65
6.5.1	LR/SC reservation set . . . . .	65
6.5.2	Store-Conditional (SC) failure response code . . . . .	66
7	<b>Hardware Memory Prefetcher</b> . . . . .	67
7.1	<b>Triggering</b> . . . . .	68
7.2	<b>Activation/Deactivation Policies</b> . . . . .	69
7.3	<b>CSRs</b> . . . . .	69
7.4	<b>Prefetch Request Algorithm</b> . . . . .	71
7.5	<b>Prefetch Abort.</b> . . . . .	71
A	<b>Appendices</b> . . . . .	73
A.1	<b>RAM macros</b> . . . . .	74
A.2	<b>Implementations</b> . . . . .	74
A.2.1	EPI Accelerator and RHEA Chip . . . . .	74

---

## List of Tables

1.1	<a href="#">HPDcache synthesis-time parameters</a>	16
1.2	<a href="#">HPDcache synthesis-time physical parameters</a>	16
1.3	<a href="#">HPDcache internal parameters</a>	17
2.1	<a href="#">Global signals</a>	20
2.2	<a href="#">Request channel signals</a>	21
2.3	<a href="#">Response channel signals</a>	21
2.4	<a href="#">Memory miss read request channel signals</a>	22
2.5	<a href="#">Memory miss read response channel signals</a>	22
2.6	<a href="#">Memory write-buffer write request channel signals</a>	23
2.7	<a href="#">Memory write-buffer write data request channel signals</a>	23
2.8	<a href="#">Memory write-buffer write response channel signals</a>	23
2.9	<a href="#">Memory read uncached request channel signals</a>	24
2.10	<a href="#">Memory read uncached response channel signals</a>	24
2.11	<a href="#">Memory write uncached request channel signals</a>	25
2.12	<a href="#">Memory write data uncached request channel signals</a>	25
2.13	<a href="#">Memory write uncached response channel signals</a>	25
2.14	<a href="#">Request operation types</a>	29
2.15	<a href="#">Memory request operation types</a>	31
2.16	<a href="#">Memory request atomic operation types</a>	31
2.17	<a href="#">Supported operation types by request interfaces to the memory</a>	32
5.1	<a href="#">CMO operation types</a>	56
A.1	<a href="#">Summary of RAM macros in the HPDcache</a>	74



---

## List of Figures

1.1	High-Level View of the HPDcache Sub-System . . . . .	14
2.1	VALID/READY scenarios. . . . .	26
2.2	Address, Data and Byte Enable Alignment in Requests . . . . .	28
3.1	HPDcache core . . . . .	36
3.2	Data Cache Micro-Architecture . . . . .	41
4.1	Dedicated CSR address space. . . . .	50
4.2	Configuration registers in the High-Performance, Multi-Requester, Multi-Issue, Out-of-Order, L1 Dcache (HPDcache) . . . . .	51
4.3	Performance counters in the High-Performance, Multi-Requester, Multi-Issue, Out-of-Order, L1 Dcache (HPDcache) . . . . .	53
4.4	Event signals in the High-Performance, Multi-Requester, Multi-Issue, Out-of-Order, L1 Dcache (HPDcache) . . . . .	53
7.1	Request issuing algorithm of prefetch engines . . . . .	72





---

## Table of Acronyms

**HPDcache** High-Performance, Multi-Requester, Multi-Issue, Out-of-Order, L1 Dcache

**DMA** Direct Memory Access

**AMBA** Advanced Microcontroller Bus Architecture

**NoC** Network-on-Chip

**PoS** Point-of-Serialization

**AMO** Atomic Memory Operation

**CSR** Configuration-and-Status Register

**CMO** Cache Maintenance Operation

**MSHR** Miss Status Holding Register

**RTAB** Replay Table

**MCR** Memory Consistency Rule

**RVWMO** RISC-V Weak Memory Ordering

**WBUF** Write Buffer

**ASIC** Application Specific Integrated Circuit

**FPGA** Field-Programmable Gate Array

**SRAM** Static Random-Access Memory

**RTL** Register-Transfer Level

**LR** Load-Reserved

**SC** Store-Conditional

**OS** Operating System



---

# Preface

The document contains the version 1.0.0-draft of the HPDcache.

## **Preface to document version 1.0.0-draft**

The changes in this version of the document include:

- Initial version of the L1 data cache (HPDcache) specification.



# Chapter 1

---

## Feature Specification

### Contents

---

1.1	List of features. . . . .	15
1.2	Synthesis-time (static) Configuration Parameters . . . . .	16

---

This High-Performance, Multi-Requester, Multi-Issue, Out-of-Order, L1 Dcache (**HPDcache**) is the responsible for serving data accesses issued by a RISC-V core, tightly-coupled accelerators and hardware memory prefetchers. All these "clients" are called requesters.

The **HPDcache** implements a hardware pipeline capable of serving one request per cycle. An arbiter in the requesters' interface of the **HPDcache** guarantees the correct behavior when there are multiple requesters. This is illustrated in [figure 1.1](#).

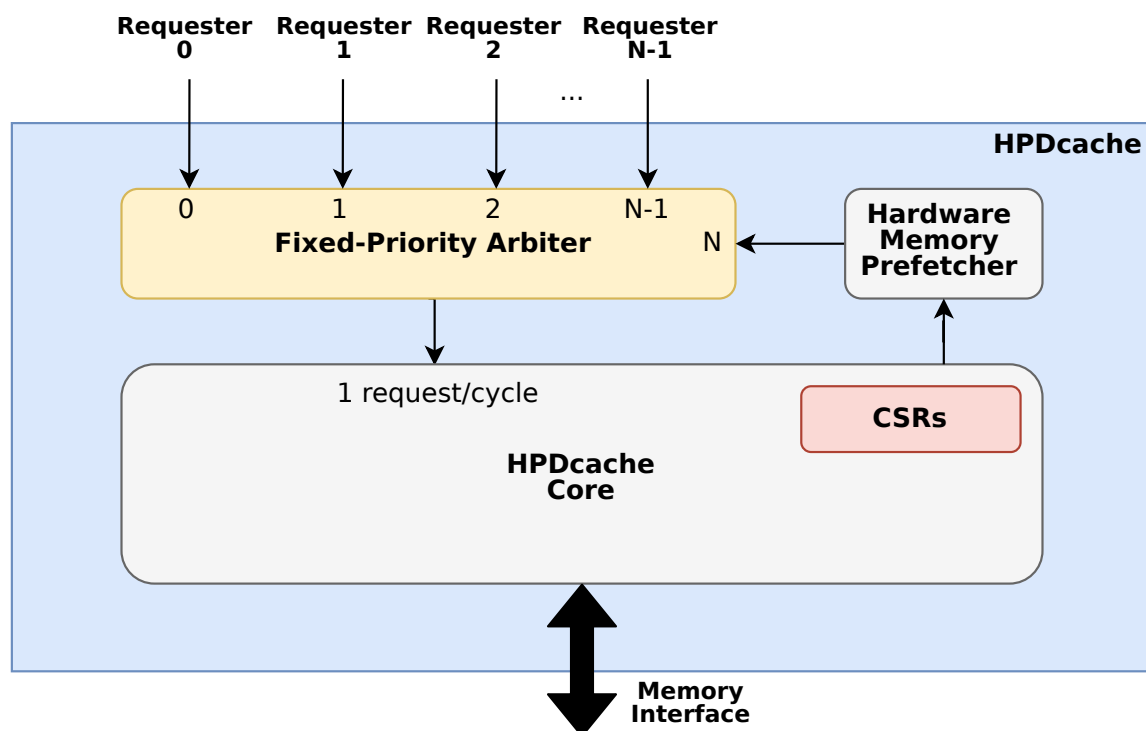


Figure 1.1: High-Level View of the HPDcache Sub-System

## 1.1 List of features

- Support for multiple outstanding requests per requester.
- Support for multiple outstanding read misses and writes to memory.
- Processes one request per cycle.
- Any given requester can access 1 to 32 bytes of a cacheline per cycle.
- Reduced energy consumption by limiting the number of RAMs consulted per request.
- Fixed priority arbiter between requesters: the requester port with the lowest index has the highest priority.
- Non-allocate, write-through policy.
- Hardware write-buffer to mask the latency of write acknowledgements from the memory system.
- Internal, configurable, hardware, memory-prefetcher that supports up to 4 simultaneous prefetching streams.
- Compliance with RISC-V Weak Memory Ordering ([RVWMO](#)).
  - For address-overlapping transactions, the cache guarantees that these are committed in the order in which they are consumed from the requesters.
  - For non-address-overlapping transactions, the cache may execute them in an out-of-order fashion to improve performance.
- Support for [CMOs](#): cache invalidation operations, and memory fences for multi-core synchronisation. Cache invalidation operations support the ones defined in the RISC-V CMO Standard.
- Memory-mapped [CSRs](#) for runtime configuration of the cache, status and performance monitoring.
- Ready-Valid, 8 channels (4 request/4 response), interface to the memory. This interface, cache memory interface (CMI), can be easily adapted to mainstream Network-on-Chip ([NoC](#)) interfaces like [AMBA AXI](#) [[1](#)].
- An adapter for interfacing with AXI5 is provided.

## 1.2 Synthesis-time (static) Configuration Parameters

The [HPDcache](#) has several static configuration parameters. These parameters must be defined at compilation/synthesis.

[Table 1.1](#) summarizes the list of parameters that can be set when integrating the [HPDcache](#). In [appendix A.2](#), we describe different systems where the [HPDcache](#) was integrated, and we list the parameters used in those implementations.

Table 1.1: HPDcache synthesis-time parameters

CONF_DCACHE_PA_WIDTH	Physical address width (in bits)
CONF_DCACHE_WORD_WIDTH	Width (in bits) of a data word
CONF_DCACHE_REQ_WORDS	Number of words in the data channels from/to requesters
CONF_DCACHE_REQ_TRANS_ID_WIDTH	Width (in bits) of the transaction ID from requesters
CONF_DCACHE_REQ_SRC_ID_WIDTH	Width (in bits) of the source ID from requesters
CONF_DCACHE_SETS	Number of sets
CONF_DCACHE_WAYS	Number of ways (associativity)
CONF_DCACHE_CL_WORDS	Number of words in a cacheline
CONF_DCACHE_WBUF_DIR_ENTRIES	Number of entries in the directory of the write buffer
CONF_DCACHE_WBUF_DATA_ENTRIES	Number of entries in the data part of the write buffer
CONF_DCACHE_WBUF_WORDS	Number of data words per entry in the write buffer
CONF_DCACHE_WBUF_TIMECNT_WIDTH	Width (in bits) of the time counter in write buffer entries
CONF_DCACHE_RTAB_ENTRIES	Number of entries in the replay table
CONF_DCACHE_MSHR_SETS	Number of sets in the Miss Status Holding Register ( <a href="#">MSHR</a> )
CONF_DCACHE_MSHR_WAYS	Number of ways (associativity) in the <a href="#">MSHR</a>
CONF_DCACHE_MEM_WORDS	Number of words in the data channels from/to the memory interface
CONF_DCACHE_MEM_ID_WIDTH	Width (in bits) of the transaction ID from the memory interface

Some parameters are not directly related with functionality ([table 1.2](#)). Instead, they allow adapting the [HPDcache](#) to physical constraints in the target technology node. Generally, these control the mapping to SRAM macros.. Depending on the technology, some dimensions are a more efficient than others (in terms of area, power, and performance). These also need to be provided by the user at synthesis-time.

Table 1.2: HPDcache synthesis-time physical parameters

CONF_DCACHE_MSHR_WAYS_PER_RAM_WORD	Number of ways in the same MSHR SRAM word
CONF_DCACHE_MSHR_SETS_PER_RAM	Number of sets per RAM macro in the MSHR array of the cache
CONF_DCACHE_DATA_WAYS_PER_RAM_WORD	Number of ways in the same CACHE data SRAM word
CONF_DCACHE_DATA_SETS_PER_RAM	Number of sets per RAM macro in the DATA array of the cache
CONF_DCACHE_ACCESS_WORDS	Number of words of a given SET that can be accessed simultaneously from the CACHE data array

Several internal configuration values are computed from the above ones. [Table 1.3](#) has a non-complete list of these internal configuration values that may be mentioned in the remainder of this document.



Table 1.3: HPDcache internal parameters

DCACHE_CL_WIDTH	Width (in bits) of a cacheline $\text{CONF\_DCACHE\_CL\_WORDS} \times \text{CONF\_DCACHE\_WORD\_WIDTH}$
DCACHE_NLINE_WIDTH	Width (in bits) of the CACHELINE index part of the address $\text{CONF\_DCACHE\_PA\_WIDTH} - \log_2(\text{DCACHE\_CL\_WIDTH}/8)$
DCACHE_SET_WIDTH	Width (in bits) of the SET part of the address $\log_2(\text{CONF\_DCACHE\_SETS})$
DCACHE_TAG_WIDTH	Width (in bits) of the TAG part of the address $\text{DCACHE\_NLINE\_WIDTH} - \text{DCACHE\_SET\_WIDTH}$
DCACHE_WBUF_WIDTH	Width (in bits) of an entry in the write-buffer $\text{CONF\_DCACHE\_WBUF\_WORDS} \times \text{CONF\_DCACHE\_WORD\_WIDTH}$



# Chapter 2

---

## Interfaces and Communication Protocols

### Contents

---

2.1	<a href="#">Global Signals</a>	20
2.2	<a href="#">Requesters' Request/Response Interface</a>	21
2.2.1	<a href="#">Signal Descriptions</a>	21
2.3	<a href="#">Memory Request/Response Interfaces</a>	22
2.3.1	<a href="#">Signal Descriptions</a>	22
2.4	<a href="#">Interfaces' requirements</a>	25
2.4.1	<a href="#">Valid/ready handshake process</a>	25
2.4.2	<a href="#">Address, data and byte enable alignment</a>	27
2.5	<a href="#">Requesters interface attributes</a>	29
2.5.1	<a href="#">Type of operation</a>	29
2.5.2	<a href="#">Source identifier</a>	29
2.5.3	<a href="#">Transaction identifier</a>	29
2.5.4	<a href="#">Cacheability</a>	30
2.5.5	<a href="#">Need response</a>	30
2.5.6	<a href="#">Error response</a>	30
2.6	<a href="#">Memory interface attributes</a>	31
2.6.1	<a href="#">Type of operation</a>	31

---

## 2.1 Global Signals

Table 2.1: Global signals

Signal	Source	Description
CLK_I	Clock source	Global clock signal. The <a href="#">HPDcache</a> is synchronous to the rising-edge of the clock.
RST_NI	Reset source	Global reset signal. Asynchronous, active LOW, reset signal.
WBUF_FLUSH_I	System	Force the write-buffer to send all pending writes. Active HIGH, one-cycle, pulse signal. Synchronous to CLK_I.
WBUF_EMPTY_O	System	Indicates if the write-buffer is empty (there is no pending write transactions). When this signal is set to 1, the write-buffer is empty.

## 2.2 Requesters' Request/Response Interface

This section describes the interfaces between the requesters and the [HPDcache](#).

All these interfaces are synchronous to the rising edge of the global clock CLK\_I ([section 2.1](#))

### 2.2.1 Signal Descriptions

Table 2.2: Request channel signals

Signal	Source	Description
DCACHE_REQ_VALID	Requester	Indicates that the channel is signaling a valid request. See <a href="#">section 2.4.1</a> .
DCACHE_REQ_READY	Cache	Indicates that the cache is ready to accept a request. See <a href="#">section 2.4.1</a> .
DCACHE_REQ_ADDR	Requester	Target physical address of the request. The address shall be aligned to the DCACHE_REQ_SIZE field. See <a href="#">section 2.4.2</a> .
DCACHE_REQ_OP	Requester	Indicates the type of operation to be performed. See <a href="#">section 2.5.1</a> .
DCACHE_REQ_WDATA	Requester	Write data (little-endian). It shall be naturally aligned to the address. See <a href="#">section 2.4.2</a> .
DCACHE_REQ_BE	Requester	Byte-enable for write data (little-endian). It shall be naturally aligned to the address. See <a href="#">section 2.4.2</a> .
DCACHE_REQ_SIZE	Requester	Indicate the size of the access. The size is encoded as the power-of-two of the number of bytes (e.g. 0 is $2^0 = 1$ , 5 is $2^5 = 32$ ).
DCACHE_REQ_UNCACHEABLE	Requester	Indicates whether the access needs to be cached (unset) or not (set). Uncacheable accesses are directly forwarded to the memory. See <a href="#">section 2.5.4</a> .
DCACHE_REQ_SID	Requester	The identification tag for the requester. It shall be identical to the index of the request port binded to that requester. See <a href="#">section 2.5.2</a> .
DCACHE_REQ_TID	Requester	The identification tag for the request. A requester can issue multiple requests. The corresponding response from the cache will return this TID. See <a href="#">section 2.5.3</a> .
DCACHE_REQ_NEED_RSP	Requester	The identification tag for the request. Indicates whether the request needs a response from the cache. When unset, the cache will not issue a response for the corresponding request. See <a href="#">section 2.5.5</a> .

Table 2.3: Response channel signals

Signal	Source	Description
DCACHE_RSP_VALID	Cache	Indicates that the channel is signaling a valid response. See <a href="#">section 2.4.1</a> .
DCACHE_RSP_RDATA	Cache	Response read data. It shall be naturally aligned to the request address. See <a href="#">section 2.4.2</a> .
DCACHE_RSP_SID	Cache	The identification tag for the requester. It corresponds to the SID transferred with the request. See <a href="#">section 2.5.2</a> .
DCACHE_RSP_TID	Cache	The identification tag for the request. It corresponds to the TID transferred with the request. See <a href="#">section 2.5.3</a> .
DCACHE_RSP_ERROR	Cache	Indicates whether there was an error condition while processing the request. See <a href="#">section 2.5.6</a> .

## 2.3 Memory Request/Response Interfaces

This section describes the interfaces between the [HPDcache](#) and the [NoC](#)/memory.

All these interfaces are synchronous to the rising edge of the global clock CLK\_I ([section 2.1](#))

### 2.3.1 Signal Descriptions

Table 2.4: Memory miss read request channel signals

Signal	Source	Description
MEM_REQ_MISS_READ_VALID	Cache	Indicates that the channel is signaling a valid request.
MEM_REQ_MISS_READ_READY	NoC	Indicates that the <a href="#">NoC</a> is ready to accept a request.
MEM_REQ_MISS_READ_ADDR	Cache	Target physical address of the request. The address shall be aligned to the MEM_REQ_MISS_READ_SIZE field. See <a href="#">section 2.4.2</a> .
MEM_REQ_MISS_READ_LEN	Cache	Indicates the number of transfers in a burst minus one. <b>In this interface, for this version, this number is always 0 (one transfer). However, bigger values may be used in the future. Thus, it should be decoded.</b>
MEM_REQ_MISS_READ_SIZE	Cache	Indicate the size of the access. The size is encoded as the power-of-two of the number of bytes. <b>In the current design implementation, the size value is equal to <math>\log_2(\text{DCACHE\_CL\_WIDTH}/8)</math>. However, smaller values may be used in the future. Thus, it should be decoded.</b>
MEM_REQ_MISS_READ_ID	Cache	The identification tag for the request
MEM_REQ_MISS_READ_COMMAND	Cache	Indicates the type of operation to be performed. <b>This interface only issues READ operations.</b>
MEM_REQ_MISS_READ_ATOMIC	Cache	In case of atomic operations, it indicates its type. <b>In this interface, this signal is not used, thus its value shall be ignored.</b>
MEM_REQ_MISS_READ_CACHEABLE	Cache	This is a hint for the cache hierarchy in the system. It indicates if the request can be allocated by the cache hierarchy. That is, data can be prefetched from memory or can be reused for multiple read transactions. <b>This bit is always set in this interface.</b>

Table 2.5: Memory miss read response channel signals

Signal	Source	Description
MEM_RESP_MISS_READ_VALID	NoC	Indicates that the channel is signaling a valid response.
MEM_RESP_MISS_READ_READY	Cache	Indicates that the cache is ready to accept a response.
MEM_RESP_MISS_READ_ERROR	NoC	Indicates whether there was an error condition while processing the request.
MEM_RESP_MISS_READ_ID	NoC	The identification tag for the request. It corresponds to the ID transferred with the request. See <a href="#">section 2.4.2</a> .
MEM_RESP_MISS_READ_DATA	NoC	Response read data. It shall be naturally aligned to the request address. See <a href="#">section 2.4.2</a> .
MEM_RESP_MISS_READ_LAST	NoC	Indicates the last transfer in a read response burst.

Table 2.6: Memory write-buffer write request channel signals

Signal	Source	Description
MEM_REQ_WBUF_WRITE_VALID	Cache	Indicates that the channel is signaling a valid request.
MEM_REQ_WBUF_WRITE_READY	NoC	Indicates that the cache is ready to accept a response.
MEM_REQ_WBUF_WRITE_ADDR	Cache	Target physical address of the request. The address shall be aligned to the MEM_REQ_WBUF_WRITE_SIZE field. See <a href="#">section 2.4.2</a> .
MEM_REQ_WBUF_WRITE_LEN	Cache	Indicates the number of transfers in a burst minus one. <b>In this interface, this number is always 0 (one transfer). However, bigger values may be used in the future. Thus, it should be decoded.</b>
MEM_REQ_WBUF_WRITE_SIZE	Cache	Indicate the size of the access. The size is encoded as the power-of-two of the number of bytes. <b>In this interface, the size shall be less or equal to <math>\log_2(\text{CONF\_DCACHE\_WBUF\_WORDS})</math>.</b>
MEM_REQ_WBUF_WRITE_ID	Cache	The identification tag for the request.
MEM_REQ_WBUF_WRITE_COMMAND	Cache	Indicates the type of operation to be performed. <b>In this interface, this signal is always a WRITE operation.</b>
MEM_REQ_WBUF_WRITE_ATOMIC	Cache	In case of atomic operations, it indicates its type. <b>In this interface, this signal is not used, thus its value shall be ignored.</b>
MEM_REQ_WBUF_WRITE_CACHEABLE	Cache	This is a hint for the cache hierarchy in the system. It indicates if the write is bufferable by the cache hierarchy. This means that the write must be visible in a timely manner at the final destination. However, write responses can be obtained from an intermediate point. <b>This bit is always set in this interface.</b>

Table 2.7: Memory write-buffer write data request channel signals

Signal	Source	Description
MEM_REQ_WBUF_WRITE_DATA_VALID	Cache	Indicates that the channel is transferring a valid data.
MEM_REQ_WBUF_WRITE_DATA_READY	NoC	Indicates that the target is ready to accept the data.
MEM_REQ_WBUF_WRITE_DATA_WDATA	Cache	Request write data. It shall be naturally aligned to the request address. See <a href="#">section 2.4.2</a> .
MEM_REQ_WBUF_WRITE_DATA_BE	Cache	Request write byte-enable. It shall be naturally aligned to the request address. See <a href="#">section 2.4.2</a> .
MEM_REQ_WBUF_WRITE_DATA_LAST	Cache	Indicates the last transfer in a write request burst.

Table 2.8: Memory write-buffer write response channel signals

Signal	Source	Description
MEM_RESP_WBUF_WRITE_VALID	NoC	Indicates that the channel is transferring a valid write acknowledgement.
MEM_RESP_WBUF_WRITE_READY	Cache	Indicates that the cache is ready to accept the acknowledgement.
MEM_RESP_WBUF_WRITE_IS_ATOMIC	NoC	Indicates whether the atomic operation was successfully processed (atomically). <b>The value in this signal is ignored in this interface.</b>
MEM_RESP_WBUF_WRITE_ERROR	NoC	Indicates whether there was an error condition while processing the request.
MEM_RESP_WBUF_WRITE_ID	NoC	The identification tag for the request. It corresponds to the ID transferred with the request.

Table 2.9: Memory read uncached request channel signals

Signal	Source	Description
MEM_REQ_UC_READ_VALID	Cache	Indicates that the channel is signaling a valid request.
MEM_REQ_UC_READ_READY	NoC	Indicates that the NoC is ready to accept a request.
MEM_REQ_UC_READ_ADDR	Cache	Target physical address of the request. The address shall be aligned to the MEM_REQ_MISS_READ_SIZE field. See <a href="#">section 2.4.2</a> .
MEM_REQ_UC_READ_LEN	Cache	Indicates the number of transfers in a burst minus one. <b>In this interface, this number is always 0 (one transfer).</b>
MEM_REQ_UC_READ_SIZE	Cache	Indicate the size of the access. The size is encoded as the power-of-two of the number of bytes.
MEM_REQ_UC_READ_ID	Cache	The identification tag for the request
MEM_REQ_UC_READ_COMMAND	Cache	Indicates the type of operation to be performed. <b>In this interface, this signal is always a READ operation.</b>
MEM_REQ_UC_READ_ATOMIC	Cache	In case of atomic operations, it indicates its type. <b>In this interface, this signal is not used, thus its value shall be ignored.</b>
MEM_REQ_UC_READ_CACHEABLE	Cache	This is a hint for the cache hierarchy in the system. It indicates if the request can be allocated by the cache hierarchy. That is, data can be prefetched from memory or can be reused for multiple read transactions. <b>This bit is always unset in this interface. Thus data shall come from the final destination.</b>

Table 2.10: Memory read uncached response channel signals

Signal	Source	Description
		Signals are identical that for the miss response channel signals.



Table 2.11: Memory write uncached request channel signals

Signal	Source	Description
MEM_REQ_UC_WRITE_VALID	Cache	Indicates that the channel is signaling a valid request.
MEM_REQ_UC_WRITE_READY	NoC	Indicates that the cache is ready to accept a response.
MEM_REQ_UC_WRITE_ADDR	Cache	Target physical address of the request. The address shall be aligned to the MEM_REQ_UC_WRITE_SIZE field. See <a href="#">section 2.4.2</a>
MEM_REQ_UC_WRITE_LEN	Cache	Indicates the number of transfers in a burst minus one. <b>In the current HPDcache implementation, this number is always 0 (one transfer).</b>
MEM_REQ_UC_WRITE_SIZE	Cache	Indicate the size of the access. The size is encoded as the power-of-two of the number of bytes.
MEM_REQ_UC_WRITE_ID	Cache	The identification tag for the request.
MEM_REQ_UC_WRITE_COMMAND	Cache	Indicates the type of operation to be performed. <b>In this interface, this signal is either a WRITE or an ATOMIC operation.</b>
MEM_REQ_UC_WRITE_ATOMIC	Cache	In case of atomic operations, it indicates its type.
MEM_REQ_UC_WRITE_CACHEABLE	Cache	This is a hint for the cache hierarchy in the system. It indicates if the write is bufferable by the cache hierarchy. This means that the write must be visible in a timely manner at the final destination. However, write responses can be obtained from an intermediate point. <b>This bit is always unset in this interface (thus transactions are non-bufferable, and the response shall come from the final destination).</b>

Table 2.12: Memory write data uncached request channel signals

Signal	Source	Description
		Signals are identical to those for the write data request channel signals.

Table 2.13: Memory write uncached response channel signals

Signal	Source	Description
MEM_RESP_UC_WRITE_VALID	NoC	Indicates that the channel is transferring a valid write acknowledgement.
MEM_RESP_UC_WRITE_READY	Cache	Indicates that the cache is ready to accept the acknowledgement.
MEM_RESP_UC_WRITE_IS_ATOMIC	NoC	Indicates whether the atomic operation was successfully processed (atomically).
MEM_RESP_UC_WRITE_ERROR	NoC	Indicates whether there was an error condition while processing the request.
MEM_RESP_UC_WRITE_ID	NoC	The identification tag for the request. It corresponds to the ID transferred with the request.

## 2.4 Interfaces' requirements

This section describes the basic protocol transaction requirements for the different interfaces in the [HPDcache](#).

### 2.4.1 Valid/ready handshake process

All interfaces in the [HPDcache](#) use a **VALID/READY** handshake process to transfer a payload between a source and a destination. The payload contains the address, data and control information.

As a reminder, the interfaces in the [HPDcache](#) are the following:

- Requesters' request interface ([table 2.2](#));
- Requesters' response interface ([table 2.3](#));

- Memory miss read request interface (table 2.4);
- Memory miss read response interface (table 2.5);
- Memory write-buffer write request interface (table 2.6);
- Memory write-buffer write data request interface (table 2.7);
- Memory write-buffer write response interface (table 2.8);
- Memory uncached read request interface (table 2.9);
- Memory uncached read response interface (table 2.10);
- Memory uncached write request interface (table 2.11);
- Memory uncached write data request interface (table 2.12);
- Memory uncached write response interface (table 2.13);

The source sets to 1 the **VALID** signal to indicate when the payload is available. The destination sets to 1 the **READY** signal to indicate that it can accept that payload. Transfer occurs only when both the **VALID** and **READY** signals are set to 1 on the next rising edge of the clock.

A source is not permitted to wait until **READY** is set to 1 before setting **VALID** to 1.

A destination may or not wait for **VALID** to set the **READY** to 1 (figure 2.1 (a) & (c)). In other words, a destination may set **READY** to 1 before an actual transfer is available (figure 2.1 (a)).

When **VALID** is set to 1, the source must keep it that way until the handshake occurs. This is, at the next rising edge when both **VALID** and **READY** (from the destination) are set to 1. In other words, a source cannot retire a pending **VALID** transfer (figure 2.1 (b)).

After an effective transfer (**VALID** and **READY** set to 1), the source may keep **VALID** set to 1 in the next cycle to signal a new transfer (with a new payload). In the same manner, the destination may keep **READY** set to 1 if it can accept a new transfer. This allows back-to-back transfers, with no idle cycles, between a source and a destination (figure 2.1 (d)).

All interfaces are synchronous to the rising edge of the same global clock (table 2.1).

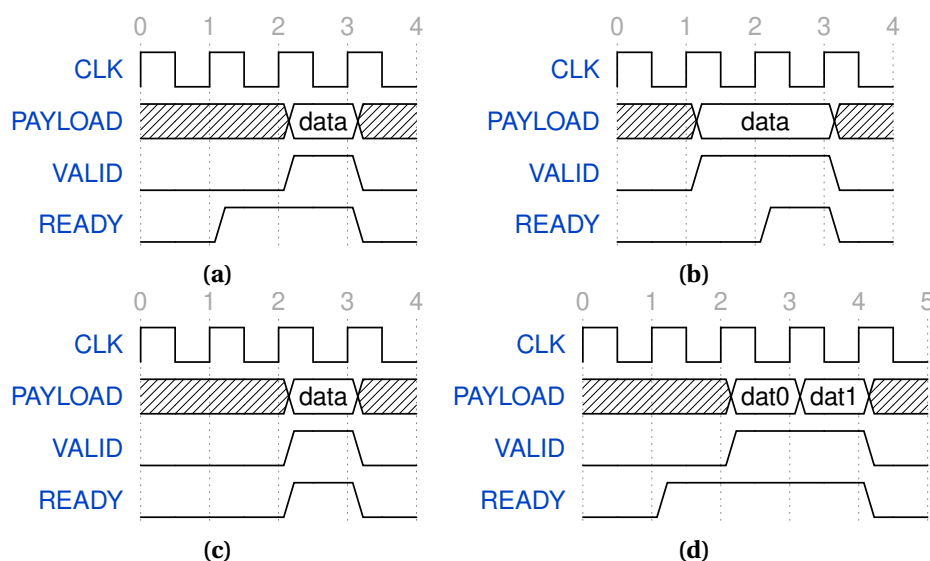


Figure 2.1: VALID/READY scenarios

### Requesters' reponse interface

In the case of the requesters' response interfaces, there is a particularity. For these interfaces, it is assumed that the **READY** signal is always set to 1. That is why the **READY** signal is not actually implemented on those interfaces. In other words, the requester must unconditionally accept the response, when it arrives.

## 2.4.2 Address, data and byte enable alignment

### Address alignment

In all request interfaces (Requesters' request interface, Memory miss read request interface, Memory write-buffer write request interface, Memory uncached read request interface, Memory uncached write request interface), the address transfered (**ADDR**) shall be byte-aligned to the value of the corresponding **SIZE** signal in that interface.

Some examples are illustrated in [figure 2.2](#). In the first case, the **SIZE** value is 2 (which corresponds to  $2^2 = 4$  bytes). Thus, the address must be a multiple of 4; In the second case, **SIZE** value is 3. Thus, the address must be a multiple of 8. Finally, in the third case, **SIZE** value is 0. Thus, there is no constraint on the address alignment.

### Data alignment

The data must be naturally aligned to the address (**ADDR**) and the maximum valid bytes of the transfer must be equal to  $2^{\text{SIZE}}$ . This means that the first valid byte in the **DATA** signal must be at the indicated offset of the address. Here, the offset corresponds to the least significant bits of the address, that allow to indicate a byte within the **DATA** word. For example, if the **DATA** signal is 128 bits wide (16 bytes), then the offset corresponds to the first 4 bits of the **ADDR** signal.

Some examples are illustrated in [figure 2.2](#). As illustrated, within the data word, only bytes in the range from the indicated offset in the address, to that offset plus  $2^{\text{SIZE}}$  can contain valid data. Other bytes must be ignored by the destination.

Additionally, within the range described above, the **BE** signal indicates which bytes within that range are actually valid. Bytes in the **WDATA** signal where the **BE** signals are set to 0, must be ignored by the destination.

### Byte Enable (BE) alignment

The **BE** signal must be naturally aligned to the address (**ADDR**) and the number of bits set in this signal must be less or equal to  $2^{\text{SIZE}}$ . This means that the first valid bit in the **BE** signal must be at the indicated offset of the address. The offset is the same as the one explained above in the "Data alignment" paragraph.

Some examples are illustrated in [figure 2.2](#). As illustrated, within the **BE** word, only bits in the range from the indicated offset in the address, to that offset plus  $2^{\text{SIZE}}$  can be set. Other bits outside that range must be set to 0.



Figure 2.2: Address, Data and Byte Enable Alignment in Requests

## 2.5 Requesters interface attributes

### 2.5.1 Type of operation

A requester indicates the required operation on the 4-bit, DCACHE\_REQ\_OP signal. The supported operation are detailed in [table 2.14](#).

Table 2.14: Request operation types

Mnemonic	Encoding	Type
DCACHE_REQ_LOAD	0b0000	Read operation
DCACHE_REQ_STORE	0b0001	Write operation
DCACHE_REQ_AMO_LR	0b0100	Atomic Load-reserved operation
DCACHE_REQ_AMO_SC	0b0101	Atomic Store-conditional operation
DCACHE_REQ_AMO_SWAP	0b0110	Atomic SWAP operation
DCACHE_REQ_AMO_ADD	0b0111	Atomic integer ADD operation
DCACHE_REQ_AMO_AND	0b1000	Atomic bitwise AND operation
DCACHE_REQ_AMO_OR	0b1001	Atomic bitwise OR operation
DCACHE_REQ_AMO_XOR	0b1010	Atomic bitwise XOR operation
DCACHE_REQ_AMO_MAX	0b1011	Atomic integer signed MAX operation
DCACHE_REQ_AMO_MAXU	0b1100	Atomic integer unsigned MAX operation
DCACHE_REQ_AMO_MIN	0b1101	Atomic integer signed MIN operation
DCACHE_REQ_AMO_MINU	0b1110	Atomic integer unsigned MIN operation
DCACHE_REQ_CMO	0b1111	Cache Maintenance Operation ( <a href="#">CMO</a> )

Load and store operations are normal read and write operations from/to the specified address.

Atomic operations are the ones specified in the Atomic (A) extension of the *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*[\[2\]](#). More details on how this cache implements them are found in [chapter 6](#).

Cache Maintenance Operations ([CMOs](#)) are explained in [chapter 5](#)

### 2.5.2 Source identifier

Each request identifies its source through the DCACHE\_REQ\_SID signal. The DCACHE\_REQ\_SID signal shall be decoded when the DCACHE\_REQ\_VALID signal is set to 1.

The width of this signal is CONF\_DCACHE\_REQ\_SRC\_ID\_WIDTH ([table 1.1](#)) bits.

The [HPDCache](#) reflects the value of the **SID** of the request into the corresponding **SID** of the response.

Each port must have an unique ID that corresponds to its number. Each port is numbered from 0 to N – 1. Port number N is dedicated to the hardware memory prefetcher. This number shall be constant for a given port (requester).

The [HPDCache](#) uses this information to route responses to the correct requester.

### 2.5.3 Transaction identifier

Each request identifies transactions through the DCACHE\_REQ\_TID signal. The DCACHE\_REQ\_TID signal shall be decoded when the DCACHE\_REQ\_VALID signal is set to 1.

The width of this signal is CONF\_DCACHE\_REQ\_TRANS\_ID\_WIDTH bits ([table 1.1](#)).

This signal can contain any value from 0 to  $2^{\text{CONF\_DCACHE\_REQ\_TRANS\_ID\_WIDTH}} - 1$ .

The **HPDcache** forwards the value of the **TID** of the request into the **TID** of the corresponding response.

A requester can issue multiple transactions without waiting for earlier transactions to complete. Because the **HPDcache** can respond to these transactions in a different order than that of requests, the requester can use the **TID** to match the responses with respect to requests.

The ID of transactions is not necessarily unique. A requester may reuse a given transaction ID for different transactions. That is, even when some of these transactions are not yet completed. In this case, when the requester starts multiple transactions with the same **TID**, the requester cannot match responses and requests. As explained above, this is because the cache can respond out-of-order with respect to requests.

### 2.5.4 Cacheability

This cache considers that the memory space is segmented. A segment corresponds to an address range: a base address and an end address. Some segments are cacheable and others not. The **HPDcache** needs to know which segments are cacheable to determine if for a given read request, it needs to replicate read data into the cache.

The request interface implements an uncacheable bit (**DCACHE\_REQ\_UNCACHEABLE**). When this bit is set, the access is considered uncacheable. The **DCACHE\_REQ\_UNCACHEABLE** signal shall be decoded when the **DCACHE\_REQ\_VALID** signal is set to 1.

#### Important

For a given address, the uncacheable attribute must be consistent between accesses. The granularity is the cacheline. In the event that the same address is accessed with different values in the uncacheable attribute, the behavior of the cache for that address is unpredictable.

### 2.5.5 Need response

For any given request, a requester can set to 0 the bit **DCACHE\_REQ\_NEED\_RSP** to indicate that it does not wish a response for that request. The **DCACHE\_REQ\_NEED\_RSP** signal shall be decoded when the **DCACHE\_REQ\_VALID** signal is set to 1.

When **DCACHE\_REQ\_NEED\_RSP** is set to 0, the **HPDcache** processes the request but it does not send an acknowledge to the corresponding requester when the transaction is completed.

### 2.5.6 Error response

The response interface contains a single-bit **DCACHE\_RSP\_ERROR** signal. This signal is set to 1 by the **HPDcache** when some error condition occurred during the processing of the corresponding request. The **DCACHE\_RSP\_ERROR** signal shall be decoded when the **DCACHE\_RSP\_VALID** signal is set to 1.

When the **DCACHE\_RSP\_ERROR** signal is set to 1 in the response, the effect of the corresponding request is undetermined. In the case of **LOAD** or **AMOs** operations (see [section 2.5.1](#)), the **RDATA** signal in the response does not contain any valid data.

## 2.6 Memory interface attributes

### 2.6.1 Type of operation

Table 2.15: Memory request operation types

Mnemonic	Encoding	Type
DCACHE_MEM_LOAD	0b00	Read operation
DCACHE_MEM_STORE	0b01	Write operation
DCACHE_MEM_ATOMIC	0b10	Atomic operation

Load and store operations are normal read and write operations from/to the specified address.

In case of an atomic operation request (DCACHE\_MEM\_ATOMIC), the specific operation is specified in the MEM\_REQ\_ATOMIC signal.

#### Atomic operations on the memory interface

The supported atomic operations are listed in [table 2.16](#). These are transmitted in the MEM\_REQ\_ATOMIC signal. Note that these operations are compatible with those in AXI.

Table 2.16: Memory request atomic operation types

Mnemonic	Encoding	Type
DCACHE_MEM_ATOMIC_ADD	0b0000	Atomic fetch-and-add operation
DCACHE_MEM_ATOMIC_CLR	0b0001	Atomic fetch-and-clear operation
DCACHE_MEM_ATOMIC_SET	0b0010	Atomic fetch-and-set operation
DCACHE_MEM_ATOMIC_EOR	0b0011	Atomic fetch-and-exclusive-or operation
DCACHE_MEM_ATOMIC_SMAX	0b0100	Atomic fetch-and-maximum (signed) operation
DCACHE_MEM_ATOMIC_SMIN	0b0101	Atomic fetch-and-minimum (signed) operation
DCACHE_MEM_ATOMIC_UMAX	0b0110	Atomic fetch-and-maximum (unsigned) operation
DCACHE_MEM_ATOMIC_UMIN	0b0111	Atomic fetch-and-minimum (unsigned) operation
DCACHE_MEM_ATOMIC_SWAP	0b1000	Atomic swap operation
DCACHE_MEM_ATOMIC_LDEX	0b1100	Load-exclusive operation
DCACHE_MEM_ATOMIC_STEX	0b1101	Store-exclusive operation

#### Operations used per interface

As a reminder, the [HPDcache](#) implements multiple (four) request interfaces to the memory:

- Memory miss read request interface ([table 2.4](#));
- Memory write-buffer (wbuf) write request interface ([table 2.6](#));
- Memory uncached read request interface ([table 2.9](#));
- Memory uncached write request interface ([table 2.11](#));

[Table 2.17](#) indicates the type of operations that each of these four request interfaces can issue.

Table 2.17: Supported operation types by request interfaces to the memory

Type	Interfaces
MEM_REQ_LOAD	- Memory miss read request; - Memory uncached read request.
MEM_REQ_STORE	- Memory write-buffer write request; - Memory uncached write request.
MEM_REQ_ATOMIC	- Memory uncached write request.

### Responses for read-modify-write atomic operations on the memory interface

The requests listed below behave as a read-modify-write operations:

- DCACHE\_MEM\_ATOMIC\_ADD
- DCACHE\_MEM\_ATOMIC\_CLR
- DCACHE\_MEM\_ATOMIC\_SET
- DCACHE\_MEM\_ATOMIC\_EOR
- DCACHE\_MEM\_ATOMIC\_SMAX
- DCACHE\_MEM\_ATOMIC\_SMIN
- DCACHE\_MEM\_ATOMIC\_UMAX
- DCACHE\_MEM\_ATOMIC\_UMIN
- DCACHE\_MEM\_ATOMIC\_SWAP

These requests are forwarded to the memory through the uncached write request interface ([table 2.11](#)). A particularity of these requests is that they generate two responses from the memory:

- Old data value from memory is returned through the memory uncached read response interface ([table 2.10](#)).
- Write acknowledgement is returned through the memory uncached write response interface ([table 2.13](#)).

Both responses may arrive in any given order to the initiating [HPDcache](#).

Regarding errors, if any response has its ERROR signal set to 1 (MEM\_RESP\_UC\_\*\_ERROR), the [HPDcache](#) considers that the operation was not completed. It waits for both responses and it forwards an error response (DCACHE\_RSP\_ERROR is set to 1) to the corresponding requester on the [HPDcache](#) requesters' side.

### Responses for exclusive load and store operations on the memory interface

Exclusive load and store operations are issued as normal load and store operations on the memory uncached read request interface ([table 2.9](#)) and memory uncached write request interface ([table 2.11](#)), respectively.

Specific operation types are however used on these exclusive requests: DCACHE\_MEM\_ATOMIC\_LDEX for loads; and DCACHE\_MEM\_ATOMIC\_STEX for stores.

These requests behave similarly to normal load and store to the memory but provide some additional properties described in [chapter 6](#).

In the case of the DCACHE\_MEM\_ATOMIC\_STEX request, the write acknowledgement contains an additional information in the MEM\_RESP\_UC\_WRITE\_IS\_ATOMIC. If this signal is set to 1, the exclusive store was "atomic", hence the data was actually written in memory. Otherwise, if this signal is set to 0, the exclusive store was "non-atomic". Hence the write operation was aborted.

The [HPDcache](#) uses exclusive stores in case of Store-Conditional (SC) operations from requesters. Depending on the MEM\_RESP\_UC\_WRITE\_IS\_ATOMIC value, the [HPDcache](#) responds to the requester



according to the rules explained in [section 6.5.2](#). A "non-atomic" response is considered a "SC failure", and a "atomic" response is considered a "SC success".



# Chapter 3

---

## Architecture

### Contents

---

<b>3.1</b>	<b>Cache Controller</b>	<b>36</b>
3.1.1	On-Hold Requests	37
3.1.2	Memory Consistency Rules (MCRs)	38
<b>3.2</b>	<b>Miss Handler</b>	<b>38</b>
3.2.1	Multiple-entry Miss Status Holding Register (MSHR)	39
<b>3.3</b>	<b>Uncacheable Handler</b>	<b>40</b>
<b>3.4</b>	<b>Cache Maintenance Operation (CMO) Handler</b>	<b>40</b>
<b>3.5</b>	<b>Cache Directory and Data</b>	<b>40</b>
3.5.1	RAM Organization	40
3.5.2	Example cache data/directory RAM organization	41
<b>3.6</b>	<b>Replay Table (RTAB)</b>	<b>42</b>
3.6.1	RTAB integration in the cache	45
3.6.2	Policy for taking new requests in the data cache	45
3.6.3	Possible improvements for the RTAB integration	46
<b>3.7</b>	<b>Write-buffer</b>	<b>46</b>
3.7.1	Memory Write Consistency Model	46
3.7.2	Functional Description	47
<b>3.8</b>	<b>Cache-coherency</b>	<b>47</b>

---

Figure 3.1 depicts a global view of the HPDcache. On the upper part of the cache there is the interface from/to requesters. On the bottom part there is the interface from/to the memory.

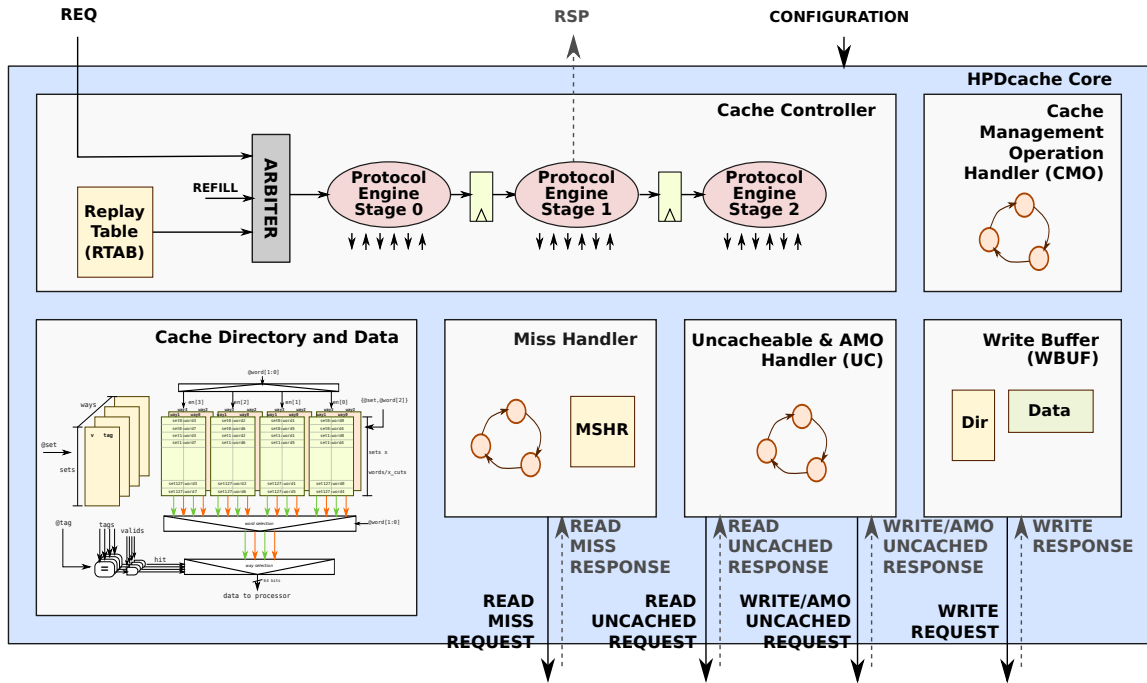


Figure 3.1: HPDcache core

### 3.1 Cache Controller

The cache controller is responsible for decoding and issuing the requests to the appropriate handler. The cache controller implements a 3-stage pipeline. This pipeline is capable of accepting one request per cycle. However, there are some scenarios where the pipeline, may either stall, or put a request on hold in a side buffer called Replay Table (RTAB).

The first stage (stage 0) of the pipeline arbitrates between requests from the miss handler (refill), RTAB, and requesters; the second stage (stage 1) responds to loads (in case of hit) and to stores; the third stage (stage 2) is only used by loads in case of miss. In this last stage, the cache allocates a new entry in the MSHR.

A request on stage 0 can either be consumed on that cycle (forwarded to the stage 1 or to the RTAB), or wait, when the pipeline is stalled. A request on stage 1 or stage 2 always advances. In stage 1 the request is either acknowledged (load hit or write acknowledgement), forwarded to stage 2 (load miss), or put into the RTAB.

#### Pipeline stalls in stage 0

Stalls in stage 0 are necessary in some specific scenarios, that are listed below. When there is a stall in stage 0, a new request from a requester cannot be accepted, this is, the corresponding READY signal is kept low (see section 2.4.1). Requests in the other stages (1 and 2) are processed normally (even in case of a stall in stage 0).

**Event 1:** The RTAB is full;

**Event 2:** A CMO invalidation or fence operation is being processed by the corresponding handler;

**Event 3:** An uncacheable or atomic operation is being processed by the corresponding handler;

**Event 4:** There is a load miss in stage 1;

**Event 5:** There is a store in stage 1 and the request in stage 0 is a load (structural hazard on access to the internal cache data memory);

The number of clock cycles of the stall in stage 0 depends on the type of event:

- **Events 4 & 5:** the number of clock cycles is always one.
- **Events 1, 2 & 3:** the number of clock cycles is variable:
  - **Event 1:** it depends on when an entry of the [RTAB](#) is freed.
  - **Events 2 & 3:** it depends on the latency of the corresponding operation.

### 3.1.1 On-Hold Requests

In some scenarios, a request that has been accepted in the pipeline can be later put on-hold by the cache controller. When a request is put on-hold, it is re-executed when all the blocking conditions have been removed. The blocking conditions putting a request on-hold are the following:

**Case 1: Cacheable LOAD or PREFETCH, and there is a hit on a pending miss (hit on the [MSHR](#))**

When there is a read miss on an address (cacheline granularity) for which there is a pending read miss, then the more recent one needs to wait for the previous one to be served. This allows the latest one to read the data from the cache after the refill operation completes.

**Case 2: Cacheable LOAD or PREFETCH, there is a miss on the cache, and there is a hit (cacheline granularity) on an opened, closed or sent entry of the Write Buffer ([WBUF](#))**

When there is a read miss on an address, the cache controller needs to read from the memory the missing cacheline. As the [NoC](#) implements different physical channels for read and write requests, there is a race condition between the read miss and a pending write operation. If the read miss arrives first to the memory, it would read the old data (which violates data consistency rules [section 3.1.2](#)). This blocking condition causes that the LOAD or PREFETCH will have a delay penalty of up to two transaction delays: one for the write to complete, then one for the read.

**Case 3: Cacheable STORE, there is a miss on the cache, and there is a hit on a pending miss (hit on the [MSHR](#))**

When writing, as the [NoC](#) implements different physical channels for read and write requests, there is a race condition between the STORE and the pending read miss. If the STORE arrives first to the memory, the earlier read miss would read the new data (which violates data consistency rules in [section 3.1.2](#)).

**Case 4: Cacheable STORE, and there is a hit on a closed entry of the [WBUF](#), or the [WBUF](#) is full**

Writes on the same address need to be sent in order (to respect data consistency rules). When there is a closed entry in the [WBUF](#), this means that it is waiting to be sent to the memory. While it is not sent, the cache cannot open a new entry in the [WBUF](#) for the same address, because they may be sent in an arbitrary order.

**Case 5: Cacheable LOAD/PREFETCH/STORE, and there is a hit on an entry of the [RTAB](#)**

Accesses to the same address (in cacheline granularity) MUST be processed in order (to respect data consistency rules). In case of a hit with a valid entry in the [RTAB](#), the new request is written into the corresponding list of the [RTAB](#).

### Case 6: Cacheable LOAD or PREFETCH, there is a miss on the cache, and the **MSHR** has no available slots

When there is a read miss on an address, the cache controller needs to allocate a new entry in the **MSHR**. The **MSHR** is a set-associative memory. If there is no available WAY to store the new read miss request, then this request needs to wait for an entry in the **MSHR** with the corresponding SET to be freed. This is when a refill operation is completed for a cacheline with the same **MSHR** SET index.

### Case 7: Cacheable LOAD or PREFETCH, there is a miss on the cache, and the miss handler FSM cannot send the read miss request

When there is a read miss on an address, the cache controller needs to read from memory the missing cacheline. The read miss request is sent by the miss handler FSM, but if there is congestion in the **NoC**, this read request cannot be issued. To avoid blocking the pipeline and creating a deadlock, the request is put on-hold.

All these conditions, except for case 5, are checked on the second stage (stage 1) of the pipeline. Case 5 is checked in the first stage (stage 0) of the pipeline. If one of the conditions is met, the request is put into the **RTAB**. It is kept on-hold until its blocking condition is solved. At that moment, the request can be replayed from the **RTAB** on the pipeline from stage 0.

The **RTAB** can store multiple requests (on-hold requests). The idea is to improve the throughput of the cache by reducing the number of cases where there is a head of line blocking at the client interface.

When a request cannot be processed right away, because it depends on the completion of a previous one, the request is stored in the replay table. This allows new requests to arrive to the data cache and to be potentially executed (in an out-of-order fashion). To prevent a deadlock, if the **RTAB** is full, the **HPDcache** does not accept new requests.

The ready requests in the **RTAB** have higher priority than new requests. These requests are executed as soon as possible, that is, when their dependencies are resolved.

To execute a request from the **RTAB**, the cache controller complies to the rules defined in [section 3.1.2](#).

## 3.1.2 Memory Consistency Rules (**MCRs**)

When multiple requests are put on-hold in the **RTAB**, the cache controller may issue them (once they are ready) in a different order than the order in which they arrived (program order). However, the cache controller needs to respect certain rules, here called Memory Consistency Rules, to allow the requesters to have a predictable behavior.

The set of rules followed by the cache controller are those defined by the **RVWMO** memory consistency model [2]. A brief statement summarizing these rules is the following: **if one memory access (read or write), A, precedes another memory access (read or write), B, and they access overlapping addresses, then they MUST be executed in program order (A then B)**. It can be deduced from this statement, that non-overlapping accesses can be executed in any order.

Of course, the cache controller also needs to respect the progress axiom: **"no memory operation may be preceded by an infinite number of memory operations"**. That is, all memory operations need to be processed at some point in time, thus cannot wait indefinitely.

## 3.2 Miss Handler

This block is in charge of handling read miss requests to the memory. It has three parts:

1. The first part is in charge of forwarding read miss requests to the memory;

2. The second part is in charge of tracking the status of in-flight read misses;
3. The third part is in charge of writing into the cache the response data from the memory, and update the cache directory accordingly.

### 3.2.1 Multiple-entry Miss Status Holding Register (MSHR)

The second part (tracking) of the miss handler contains an essential component of the [HPDcache](#): the set-associative multi-entry [MSHR](#). Each entry of this component contains the status for each in-flight read miss request to the memory. Therefore, the number of entries in the [MSHR](#) defines the maximum number of in-flight read miss requests.

The number of entries in the [MSHR](#) depends on two configuration values: `CONF_DCACHE_MSHR_WAYS` and `CONF_DCACHE_MSHR_SETS`. The number of entries is computed as:

$$\text{DCACHE\_MSHR\_SETS} \times \text{CONF\_DCACHE\_MSHR\_WAYS}$$

As for any set-associative array:

---

When `CONF_DCACHE_MSHR_SETS = 1` and `CONF_DCACHE_MSHR_WAYS > 1`  
 → The [MSHR](#) behaves as a fully-associative access array.

---

When `CONF_DCACHE_MSHR_SETS > 1` and `CONF_DCACHE_MSHR_WAYS = 1`  
 → The [MSHR](#) behaves as a direct access array.

---

When `CONF_DCACHE_MSHR_SETS > 1` and `CONF_DCACHE_MSHR_WAYS > 1`  
 → The [MSHR](#) behaves as a set-associative access array

---

A high number of entries in the [MSHR](#) allows to overlap multiple accesses to the memory, and hides its latency. Of course, the more entries there are, the more area the [MSHR](#) consumes. Therefore, the system architect must choose [MSHR](#) parameters depending on a combination of memory latency, memory throughput, required area and performance, and the capability of requesters to issue multiple read transactions.

#### Important

Regarding the last condition, regardless whether the requesters can issue multiple read requests, the hardware memory prefetcher exploits having multiple in-flight read miss requests.

An entry in the [MSHR](#) contains the following information:

Bits	T	R	S	W	1
Description	MSHR Tag	Request ID	Source ID	Word Index	Need Response
Field	Width				
MSHR tag (T)	T = DCACHE_NLINE_WIDTH – log <sub>2</sub> (CONF_DCACHE_MSHR_SETS)				
Request ID (R)	R = CONF_DCACHE_REQ_TRANS_ID_WIDTH				
Source ID (S)	S = CONF_DCACHE_REQ_SRC_ID_WIDTH				
Word Index (W)	W = log <sub>2</sub> (CONF_DCACHE_CL_WORDS)				

### MSHR implementation

In order to limit the area cost of the [MSHR](#), it can be implemented using SRAM macros. The depth of the macros is `CONF_DCACHE_MSHR_SETS_PER_RAM`. Multiple ways, for the same set, can be put side-by-side in the same SRAM word (`CONF_DCACHE_MSHR_WAYS_PER_RAM_WORD`), therefore the width is a multiple of `DCACHE_MSHR_ENTRY = T + R + S + W + 1` bits. The total number of SRAM macros is:

$$(\text{CONF\_DCACHE\_MSHR\_WAYS} / \text{CONF\_DCACHE\_MSHR\_WAYS\_PER\_RAM\_WORD}) \times \lceil \text{CONF\_DCACHE\_MSHR\_SETS} / \text{CONF\_DCACHE\_MSHR\_SETS\_PER\_RAM} \rceil$$

SRAM macros shall be selected depending on the required number of entries, and the target technology node. Additional information about [MSHR](#) SRAM macros can be found in [appendix A.1](#). When the number of entries is low (e.g. sets times ways are less than 16), it is generally better to implement the [MSHR](#) using flip-flops.

This makes [MSHR](#) fully-associative and thus removes associativity conflicts.

## 3.3 Uncacheable Handler

This block is responsible for handling uncacheable (see [section 2.5.4](#)) load and store requests, as well as atomic requests (regardless of whether they are cacheable or not). For more information about atomic requests see [chapter 6](#).

All requests handled by this block produce a request to the memory. This request to the memory is issued through the memory uncached interfaces. Uncacheable read requests are forwarded to the memory through the memory read uncached interface ([table 2.9](#)); and uncacheable write requests or atomic requests are forwarded through the memory write uncached interface ([table 2.11](#)).

## 3.4 Cache Maintenance Operation (CMO) Handler

This block is responsible for handling [CMOs](#). [CMOs](#) are special requests from requesters that address the cache itself, and not the memory nor a peripheral. These operations allow to either invalidate designated cachelines in the cache, or produce explicit memory read and write fences.

The complete list of supported [CMOs](#) is detailed in [chapter 5](#).

## 3.5 Cache Directory and Data

### 3.5.1 RAM Organization

The [HPDcache](#) cache uses RAM macros for the directory and data parts of the cache. These RAM macros are synchronous, read/write, single-port RAMs. Additional information about RAM macros in the cache can be found in [appendix A.1](#).

The organization of the RAMs, for the directory and the data, targets the following:

1. **High memory bandwidth to/from the requesters**

To improve performance, the organization allows to read one data word (1, 2, 4, 8, 16 or 32 bytes) per cycle, with a latency of one cycle.



## 2. Low energy-consumption

To limit the energy-consumption, the RAMs are organized in a way that the cache enables only a limited number of RAM macros. This number depends on the number of requested bytes, and it also depends on the target technology. Depending on the target technology, the RAM macros have different trade-offs between width, depth and timing (performance).

## 3. Small RAM footprint

To limit the footprint of RAMs, the selected organization implements a small number of RAMs macros. The macros are selected in a way that they are as deep and as wide as possible. The selected ratios (depth x width) depend on the target technology as explained above.

### 3.5.2 Example cache data/directory RAM organization

Figure 3.2 illustrates an example organization of the RAMs. The illustrated organization allows to implement 32 KB of data cache (128 sets, 4 ways, and 64 bytes lines). This example organization has a refilling latency of two cycles because the cache needs to write two different entries on a given memory cut.

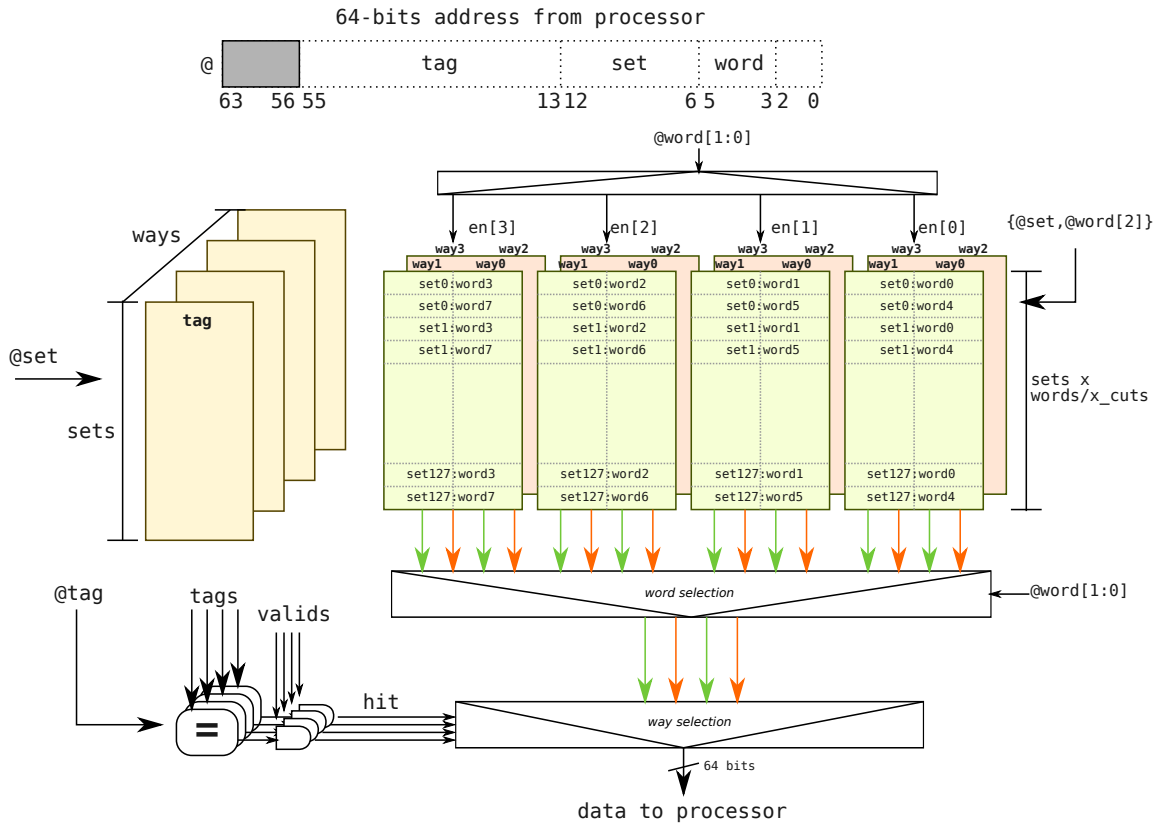


Figure 3.2: Data Cache Micro-Architecture

The example RAM organization in figure 3.2 allows to access from 1 to 32 bytes of a given cacheline per cycle.

The energy consumption is dependent on the length of the access. Accesses from 1 to 8 bytes need to read two memory cuts (one containing ways 0 and 1, and the other containing ways 2 and 3); accesses from 8 to 16 bytes need to read 4 memory cuts; and so on. For reading 24 to 32 bytes, the cache needs to access all the cuts at the same time (8 cuts).

### 3.6 Replay Table (RTAB)

The [RTAB](#) is implemented as an array of linked lists. It is a fully-associative multi-entry buffer, where each valid entry, belongs to a linked list. It is implemented in flip-flops. The linked lists contain a list of requests that target the same cacheline. There can be multiple linked lists, but each shall target a different cacheline. The head of each linked list contains the oldest request while the tail contains the newest request. The requests are processed from the head to the tail in order to respect the Memory Consistency Rules (MCRs) explained in [section 3.1.2](#).

Regarding the pop operation (extracting a ready request from the replay table), it is possible that once the request is replayed, some of the resources it needs are again busy. Therefore, the request needs to be put on-hold again. In this case, the request needs to keep its position as head of the linked list. This is to preserve the program order. For this reason, the pop operation is implemented as a two-step operation: pop then commit, or pop then rollback. The commit operation allows to actually remove the request, while the rollback allows to undo the pop.

An entry of the [RTAB](#) has the following structure (LL means Linked List):

Request (≈200 bits)	LL tail (1 bit)	LL head (1 bit)	LL next (2-3 bits)	Deps (5 bits)	Valid (1 bit)
------------------------	--------------------	--------------------	-----------------------	------------------	------------------

- Request: contains the on-hold request from the core (data + meta-data).
- LL tail: indicates if the entry is the tail of a linked list.
- LL head: indicates if the entry is the head of a linked list.
- LL next: designates the next (older) request in the linked list.
- Deps bits: indicates the kind of dependency that keeps the request on-hold.
- Valid: indicates if the entry contains valid information (if unset the entry is free).

The following table briefly describes the possible dependencies between memory requests. For each kind of dependency, there is a corresponding bit in the "deps bits" field of [RTAB](#) entries.

Dependency	Description
MSHR_hit	There is an outstanding miss request on the target address
MSHR_full	The MSHR is full
MISS_handler_busy	The MISS HANDLER is busy and cannot send a new miss request
WBUF_hit	There is a match with a open, closed, or sent entry in the write buffer
WBUF_not_ready	There is a match with a closed entry in the write buffer or the write-buffer is full

#### [RTAB](#) operations

The [RTAB](#) implements the following operations:

Operation	Description
<code>rtab_alloc()</code>	Allocate a new linked list
<code>rtab_alloc_and_link()</code>	Allocate a new entry and link it to an existing linked list
<code>rtab_pop_try()</code>	Get a ready request from one of the linked list (without actually removing it from the list)
<code>rtab_pop_commit()</code>	Actually remove a popped request from the list
<code>rtab_pop_rollback()</code>	Rollback a previously popped request (with a possible update of its dependencies)
<code>rtab_find_ready()</code>	Find a ready request among the heads of valid linked lists
<code>rtab_find_empty()</code>	Find an empty request
<code>rtab_empty()</code>	Is the RTAB empty ?
<code>rtab_full()</code>	Is the RTAB full ?
<code>update_deps()</code>	Update the dependency bits of valid requests

```

int rtab_alloc(req_t r, deps_t d)
{
    int index = rtab_find_empty_entry();
    rtab[index] = {
        valid      : 1,
        deps       : d,
        ll_head    : 1,
        ll_tail    : 1,
        ll_next    : 0,
        request    : r
    };
    return index;
}

```

```

int rtab_alloc_and_link(req_t r, int n)
{
    int index = rtab_find_empty_entry();

    // replace the tail of the linked list
    rtab[n].ll_tail = 0;

    // add the new request as the tail of the linked list
    rtab[index] = {
        valid      : 1,
        deps       : 0,
        ll_head    : 0,
        ll_tail    : 1,
        ll_next    : n,
        request    : r
    };

    return index;
}

```

```

req_t rtab_pop_try(int &index)
{
    index = rtab_find_ready_entry();

    // Temporarily unset the head bit. This is to prevent the
    // request to be rescheduled.
    rtab[index].ll_head = 0;

    return rtab[index].request;
}

void rtab_pop_commit(int index)
{
    // Change the head of the popped linked list
    // (look for a valid entry with the next field
    // pointing to the popped entry)
    for (int i = 0; i < RTAB_NENTRIES; i++) {
        if (rtab[i].valid && (i != index) && (rtab[i].next == index) {
            rtab[i].ll_head = 1;
        }
    }

    rtab[index].valid = 0;
}

void rtab_pop_rollback(int index, bitvector deps)
{
    rtab[index].ll_head = 1;
    rtab[index].deps     = deps;
}

int rtab_find_ready_entry(int last)
{
    // choose a ready entry using a round-robin policy
    int i = (last + 1) % RTAB_NENTRIES;
    for (;;) {
        // ready entry found
        if (rtab[i].valid && rtab[i].ll_head && (rtab[i].deps == 0))
            return i;

        // there is no ready entry
        if (i == last)
            return -1;

        i = (i + 1) % RTAB_NENTRIES;
    }
}

int rtab_find_empty_entry()
{
    for (int i = 0; i < RTAB_NENTRIES; i++)
        if (!rtab[i].valid)

```

```

    return i;

    return -1;
}

bool rtab_is_full()
{
    return (rtab_find_empty_entry() == -1);
}

int rtab_is_empty()
{
    for (int i = 0; i < RTAB_NENTRIES; i++)
        if (rtab[i].valid)
            return 0;

    return 1;
}

```

### 3.6.1 [RTAB](#) integration in the cache

The data cache has a 3-stages pipeline. The [RTAB](#) will be used in stages 0 and 1 (st0 and st1). The following table summarizes the actions performed on the [RTAB](#):

New Request	Match @ in <a href="#">RTAB</a>	Match @ in <a href="#">MSHR</a>	Match @ in <a href="#">WBUF</a>	Cache Miss AND <a href="#">MSHR</a> is full	Cache Miss AND Handler is not ready	<a href="#">WBUF</a> is full
LOAD	alloc_and_link (st0)	alloc_new (st1)	alloc_new (st1)	alloc_new (st1)	alloc_new (st1)	$\phi$
STORE	alloc_and_link (st0)	alloc_new (st1)	alloc_new (st1) (if wbuf_entry is closed)	$\phi$	$\phi$	alloc_new (st1)

### 3.6.2 Policy for taking new requests in the data cache

With the [RTAB](#), the cache has three possible sources of requests:

1. Requesters (new requests);
2. the [RTAB](#) (on-hold requests);
3. the miss handler (refill requests).

The policy to choose the request is as follows:

```

rtab_req = rtab_find_ready_entry();
if (rtab_is_full()) {
    new_req = rtab_req;
} else {
    new_req = (rtab_req != -1) ? rtab_req : core_req;
}

accepted_req = round_robin(new_req, refill_req);

```

To summarize: [RTAB](#) ready requests have higher priority than core requests (this is to flush the pipeline as fast as possible). However, if the [RTAB](#) is full, the cache does not accept core requests because if they need to be put on-hold that could cause a deadlock. Then, between the refill requests, the [RTAB](#) or the core requests, the data cache applies a round-robin policy.

### 3.6.3 Possible improvements for the [RTAB](#) integration

- Avoid introducing a NOP after an entry is replayed (popped). This is currently done to simplify the resolution of a concurrent `alloc_and_link` and `pop_commit` where the request being allocated depends on the one being popped.

## 3.7 Write-buffer

This cache implements a write-through policy. In this policy, the write accesses from requesters are systematically transferred to the memory, regardless of whether the write access hits or misses in the [HPDcache](#).

To decouple the acknowledgement from the memory to the [HPDcache](#), and the acknowledgement from the [HPDcache](#) to the requester, this [HPDcache](#) implements a write-buffer. The goal is to increase the performance: the requester does not wait the acknowledgement from the memory, which may suffer from a very high latency. Additionally, to improve the bandwidth utilization of data channels in the [NoC](#), the write-buffer implements coalescing of write data.

The write-buffer implements two different parts: directory and data. The directory enables tracking of active writes. The data buffers are used to coalesce writes from the requester. Entries in the data buffers are usually wider (`CONF_DCACHE_WBUF_WORDS`) than the data interface of requesters. This is to enable the coalescing of multiple writes onto contiguous addresses.

A given entry in the directory of the write-buffer may be in four different states:

<b>FREE</b>	The entry is available.
<b>OPEN</b>	The entry is currently used by a previous write access. The entry accepts new write accesses (in the same address range) for coalescing.
<b>CLOSED</b>	The entry does not accept any new writes, and is waiting to be sent to the memory.
<b>SENT</b>	The entry was forwarded to the memory, and is waiting for the acknowledgement.

### 3.7.1 Memory Write Consistency Model

The [HPDcache](#) complies with the [RVWMO](#) memory consistency model. Regarding writes, in this consistency model, there are two important properties:

1. The order in which write accesses on different addresses are forwarded to memory MAY differ from the order they arrived from the requester (program order);
2. Writes onto the same address, MUST be visible in order. If there is a data written by a write A on address @x followed by an another write B on the same address, the data of A cannot be visible after the processing of B.

The second property allows write coalescing if the hardware ensures that the last write persists.

The write-buffer exploits the first property. Multiple "in-flight" writes are supported due to the multiple directory and data entries. These writes can be forwarded to the memory in an order different than the program order.

To comply with the second property, the write-buffer does not accept a write when there is an address conflict with a **CLOSED**, or **SENT** entry. In that case, the write is put on-hold following the policy described in [section 3.1.1](#). The system may choose to relax the constraint of putting a write on-hold in case of an address conflict with a **SENT** entry. This can be relaxed when the NoC guaranties in-order delivery. The runtime configuration bit `cfg_wbuf.S` (see [table 4.2](#)) shall be deasserted to relax this dependency.

### 3.7.2 Functional Description

When an entry of the write-buffer directory is in the **OPEN** or **CLOSED** states, there is an allocated data buffer, and it contains data that has not yet been sent to the memory. When an entry of the write-buffer directory is in the **SENT** state, the corresponding data was transferred to the memory, thus the corresponding data buffer was freed. A given entry in the write-buffer directory goes from **FREE** to **OPEN** state when a new write is accepted, and cannot be coalesced with another **OPEN** entry (e.g. not in the same address range).

A directory entry passes from **OPEN** to **CLOSED** after a given number of clock cycles. This number of clock cycles depends on different runtime configurable values. Each directory entry contains a life-time counter. This counter starts at 0 when a new write is accepted (**FREE**->**OPEN**), and incremented each cycle while in **OPEN**. When the counter reaches `cfg_wbuf.threshold` (see [table 4.2](#)), the write-buffer directory entry goes to **CLOSED**. Another runtime configurable bit, `cfg_wbuf.R` (see [table 4.2](#)), defines the behavior of an entry when a new write is coalesced into an **OPEN** entry. If this last configuration bit is set, the life-time counter is reset to 0 when a new write is coalesced. Otherwise, the counter keeps its value.

The life-time of a given write-buffer directory entry is longer than the life-time of a data entry. A given directory entry is freed (**SENT**->**FREE**) when the write acknowledgement is received from the memory. The number of cycles to get an acknowledgement from the memory may be significant and it is system-dependent. Thus, to improve utilization of data buffers, the number of entries in the directory is generally greater than the number of data buffers. However, there is a trade-off between area and performance because the area cost of data buffers is the most critical cost in the write-buffer. The synthesis-time parameters `CONF_DCACHE_WBUF_DIR_ENTRIES` and `CONF_DCACHE_WBUF_DATA_ENTRIES` define the number of entries in the write-buffer directory and write-buffer data, respectively.

#### Memory Fences

In multi-core systems, or more generally, in systems with multiple DMA-capable devices, when synchronization is needed, it is necessary to implement memory fences from the software. In the case of RISC-V, there is specific instructions for this (i.e. fence).

Fence instructions shall be forwarded to the cache to ensure ordering of writes. The fence will force the write-buffer to send all pending writes before accepting new ones. This cache implements two ways of signalling a fence: sending a specific **CMO** instruction from the core (described later on [chapter 5](#)), or by asserting `wbuf_flush_i` pin (during one cycle).

## 3.8 Cache-coherency

The current version of the cache does not implement any hardware cache-coherency protocol.

In multi-core systems integrating this cache, cache-coherency needs to be enforced by the software. To this end, this cache provides cache invalidation instructions among the supported **CMOs**. These are described in [chapter 5](#). These can be used to solve the cache-obsolence problem.

As the cache implements a write-through policy, there is no memory-obsolence problem. This is because all writes are forwarded to the memory.





# Chapter 4

---

## Configuration-and-Status Registers (CSRs)

### Contents

---

4.1	Dedicated CSR address space . . . . .	50
4.2	Configuration registers . . . . .	51
4.3	Performance counters. . . . .	52
4.4	Event signals. . . . .	53

---

## 4.1 Dedicated CSR address space

### Important

This CSR address space is not yet implemented in version 1.0.0 of the RTL. In this version, runtime configuration values are passed through external ports of the HPDcache. Performance counters are not implemented either.

The [HPDcache](#) defines a dedicated memory address space for configuring and checking the internal status. This memory space is shared among all the requesters connected to the same [HPDcache](#). However, this space is private to those requesters in a system-wide point of view. This is, this dedicated [CSR](#) address space is not visible to other requesters integrated in the system.

The dedicated [CSR](#) address space is aligned to 4 Kibytes and has this same size. Current version of the [HPDcache](#) uses a very small subset of this address space, but the aligning to 4 Kibytes, allows easier mapping in the virtual address space by the Operating System (OS). The smallest virtual/physical page size defined in the *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*[\[3\]](#) is 4 Kibytes. This is the reason of this choice. [Figure 4.1](#) displays the layout of the dedicated [CSR](#) address space of the [HPDcache](#).

The CFG\_BASE address is specified through an input port of the [HPDcache](#). The name of this input pin is `cfg_base_i`. It is a multi-bit signal. The number of bits is `CONF_DCACHE_PA_WIDTH`.

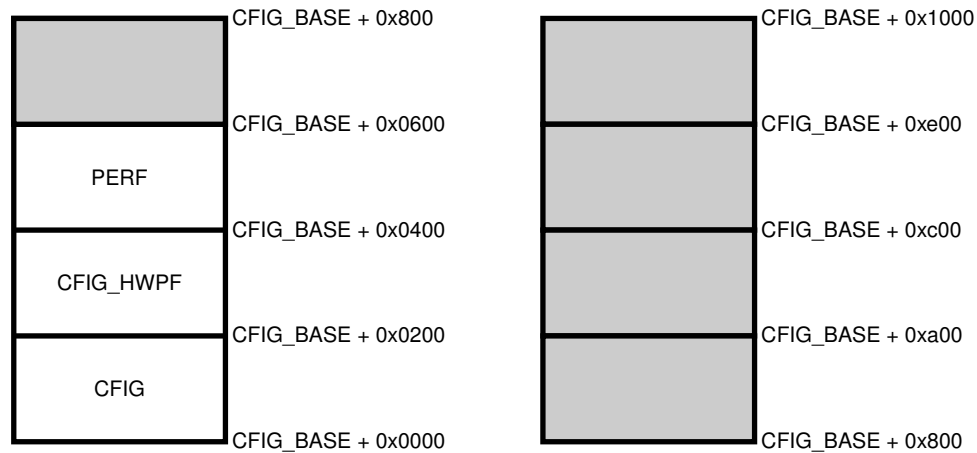


Figure 4.1: Dedicated CSR address space

## 4.2 Configuration registers

Table 4.2 lists the configuration registers implemented in the [HPDcache](#).

These are mapped on the CFG memory address segment in [figure 4.1](#).

Figure 4.2: Configuration registers in the [HPDcache](#)

CFG Segment		
Register	Description	Base address
cfg_info	64-bits register with cache information	< CFG_BASE > + 0x00
cfg_ctrl	64-bits register for configuring the cache controller	< CFG_BASE > + 0x08
cfg_wbuf	64-bits register for configuring the write-buffer	< CFG_BASE > + 0x10
CFG_HWP Segment		
Register	Description	Base address
cfg_hwpf_status	64-bits register with the status of the hardware prefetcher	< CFG_BASE > + 0x200
for (i = 0; i < 4; i++) {		
cfg_hwpf_base_engine[i]	64-bits base cline register of the engine i of the hardware prefetcher	< CFG_BASE > + 0x200 + (i + 1) × 0x20 + 0x0
cfg_hwpf_param_engine[i]	64-bits parameters register of the engine i of the hardware prefetcher	< CFG_BASE > + 0x200 + (i + 1) × 0x20 + 0x8
cfg_hwpf_throttle_engine[i]	64-bits throttle register of the engine i of the hardware prefetcher	< CFG_BASE > + 0x200 + (i + 1) × 0x20 + 0x10
}		

**cfg\_info** - < CFG\_BASE > + 0x00

63		48		23		20 19		16 15		8 7	
ID					HwPf	LnSz	Ways		Sets		
Field	Description	Mode	Reset value		Comment						
Sets	Number of sets	RO	CONF_DCACHE_SETS		Indicates the number of sets implemented.						
Ways	Number of ways	RO	CONF_DCACHE_WAYS		Indicates the number of ways implemented.						
LnSz	Number of bytes per cacheline (power of 2)	RO	log <sub>2</sub> (DCACHE_CL_WIDTH/8)		It contains the log <sub>2</sub> of the size in bytes of cachelines.						
HwPf	Number of engines in the hardware prefetcher	RO	4		Indicates the number of simultaneous streams supported by the hardware prefetcher						
ID	Version ID	RO	0xCEA0		Version ID of the <a href="#">HPDcache</a> .						

**cfg\_ctrl** - < CFG\_BASE > + 0x08

63	57 56				0
	A	R			E
Field	Description	Mode	Reset value	Comment	
E	Cache Enable	RW	0b0	When set to 0, all memory accesses are considered non-cacheable.	
R	Single-entry RTAB (fallback mode)	RW	0b0	This is a fallback mode. When set to 1, the cache controller only uses one entry in the <a href="#">RTAB</a> .	
A	Forbid AMO mode ( <a href="#">section 6.3</a> )	RW	0b0	When set to 1, the cache controller responds with an error to AMO requests targeting cacheable addresses.	

**cfg\_wbuf** - < CFG\_BASE > + 0x10

63			15	8	1	0
				Threshold		S R
Field	Description	Mode	Reset value	Comment		
R	Reset time-counter on write	RW	0b1	When set to 1, writes restart the time-counter in the corresponding write-buffer entry.		
S	Sequential Write-After-Write	RW	0b0	When set to 1, the write-buffer holds-back writes requests that matches the target address of an on-the-fly write.		
Threshold	Number of keep-alive cycles of entries in the write-buffer	RW	0x04	The maximum accepted value is CONF_DCACHE_WBUF_TIMECNT_MAX. When set to 0, a write immediately closes the corresponding entry.		

**cfg\_hwpf\_\***

These registers are related to the hardware prefetcher. They are mapped on the CFG\_HWPF memory address segment.

Details on hardware prefetcher configuration registers are in [section 7.3](#).

## 4.3 Performance counters

The [HPDcache](#) provides a set of performance counters. These counters provide important information that can be used by software developers, at [OS](#) level or user application level, to, for example, debug performance issues.

[Table 4.3](#) lists the performance counters provided by the [HPDcache](#). These are mapped on the PERF memory address segment in [figure 4.1](#).

Figure 4.3: Performance counters in the HPDcache

Counter	Description	Base address
perf_write_req	64-bits counter for processed write requests	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x00$
perf_read_req	64-bits counter for processed read requests	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x08$
perf_prefetch_req	64-bits counter for processed prefetch requests	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x10$
perf_uncached_req	64-bits counter for processed uncached requests	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x18$
perf_cmo_req	64-bits counter for processed CMO requests	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x20$
perf_accepted_req	64-bits counter for accepted requests	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x28$
perf_cache_write_miss	64-bits counter for write cache misses	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x30$
perf_cache_read_miss	64-bits counter for read cache misses	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x38$
perf_on_hold_req	64-bits counter for requests put on-hold	$\langle \text{PERF\_BASE} \rangle^{\alpha} + 0x40$

$$\alpha : \langle \text{PERF\_BASE} \rangle = \langle \text{CFG\_BASE} \rangle + 0x400$$

## 4.4 Event signals

In addition to the performance registers explained in [section 4.3](#), the HPDcache provides a set of one-shot signals that indicate when a given event is detected. As one-shot signals, they are set to 1 for one cycle each time the corresponding event is detected. If the same event is detected N cycles in a row, the corresponding event signal will remain set to 1 for N cycles. [Table 4.4](#) lists these event signals.

These event signals are output-only. They can be either left unconnected, if they are not used, or connected with the remainder of the system. The system can use those signals, for example, for counting those events externally or for triggering some specific actions.

Figure 4.4: Event signals in the HPDcache

Signal	Event description
evt_cache_write_miss_o	Cache miss on write operation
evt_cache_read_miss_o	Cache miss on read operation
evt_uncached_req_o	The cache processed an uncached request
evt_cmo_req_o	The cache processed a CMO request
evt_write_req_o	The cache processed a write request
evt_read_req_o	The cache processed a read request
evt_prefetch_req_o	The cache processed a prefetch request
evt_on_hold_req_o	The cache put on-hold a request



# Chapter 5

---

## Cache Management Operations (CMOs)

### Contents

---

5.1	Memory write fence . . . . .	57
5.2	Invalidate a cacheline by its physical address . . . . .	58
5.3	Invalidate a group of cachelines by their a set and way . . . . .	59
5.4	Invalidate the entire cache . . . . .	60
5.5	Prefetch a cacheline given its physical address . . . . .	61

---

The [HPDcache](#) is able of performing the following Cache Maintenance Operations:

- memory write fence;
- invalidate a cacheline given a physical address;
- invalidate one or more cachelines in a given set given the set and one or more ways;
- invalidate all the cachelines;
- prefetch the cacheline indicated by its physical address.

Any of the clients of the DCACHE can trigger one of this operation anytime by using specific opcodes in their request.

Table 5.1: CMO operation types

Mnemonic	Encoding	Type
DCACHE_CMO_FENCE	0b000	Memory write fence.
DCACHE_CMO_INVAL_NLINE	0b010	Invalidate a given cacheline.
DCACHE_CMO_INVAL_SET_WAY	0b011	Invalidate one or more ways of in a given set of the cache.
DCACHE_CMO_INVAL_ALL	0b100	Invalidate the entire cache.
DCACHE_CMO_PREFETCH	0b101	Prefetch a given cacheline.

The DCACHE\_REQ\_OP must be set to DCACHE\_REQ\_CMO (see [table 2.14](#)). The CMO subtype ([table 5.1](#)) is transferred into the DCACHE\_REQ\_SIZE signal of the request.

The following sections describe in detail each of the [CMO](#) operations, and how the requests shall be encoded to trigger each of them.



## 5.1 Memory write fence

To make sure that the [HPDcache](#) accepts new requests only when all previous writes are sent and acknowledged from the memory, a requester can issue a fence operation.

To do this, the requester shall build the request as follows:

Signal	Value
DCACHE_REQ_ADDR	*
DCACHE_REQ_OP	DCACHE_REQ_CMO
DCACHE_REQ_WDATA	*
DCACHE_REQ_BE	*
DCACHE_REQ_SIZE	DCACHE_CMO_FENCE
DCACHE_REQ_UNCACHEABLE	*
DCACHE_REQ_SID	Corresponding source ID of the requester
DCACHE_REQ_TID	Transaction identifier from the requester
DCACHE_REQ_NEED_RSP	*

\* means don't care

As for any regular request, the request shall follow the **VALID/READY** handshake protocol described in [section 2.4.1](#).

This operation has the following effects:

- All open entries in the write buffer (write requests waiting to be sent to the memory) are immediately closed;
- No new requests from any requester are acknowledged until all pending write requests in the cache have been acknowledged on the [NoC](#) interface.

## 5.2 Invalidate a cacheline by its physical address

To invalidate a cacheline by its physical address, the requester shall build the request as follows:

Signal	Value
DCACHE_REQ_ADDR	Physical address to invalidate in the cache.
DCACHE_REQ_OP	DCACHE_REQ_CMO
DCACHE_REQ_WDATA	*
DCACHE_REQ_BE	*
DCACHE_REQ_SIZE	DCACHE_CMO_INVAL_NLINE
DCACHE_REQ_UNCACHEABLE	*
DCACHE_REQ_SID	Corresponding source ID of the requester
DCACHE_REQ_TID	Transaction identifier from the requester
DCACHE_REQ_NEED_RSP	*

\* means don't care

As for any regular request, the request shall follow the **VALID/READY** handshake protocol described in [section 2.4.1](#).

For the sake of design simplification, this operation works as a memory read fence. That is, before handling the operation, the **HPDcache** waits for all pending read misses to complete. Future versions of the HPDcache could wait only for a pending read miss on the same address that is being invalidated.

If the given physical address is not cached, the operation does nothing. However it still works as a memory read fence.

Regarding the latency of this operation, it depends on the time to serve all pending read misses. Only one cycle is needed to invalidate the corresponding cacheline.

### 5.3 Invalidate a group of cachelines by their a set and way

To invalidate a group of cachelines, the requester shall build the request as follows:

Signal	Value
DCACHE_REQ_ADDR	Index of the set to invalidate.
DCACHE_REQ_OP	DCACHE_REQ_CMO
DCACHE_REQ_WDATA	Bit-vector with target ways to invalidate. The number of bits decoded depends on the number of ways implemented (CONF_DCACHE_WAYS). The least significant bit corresponds to way 0, the second to way 1, etc.
DCACHE_REQ_BE	*
DCACHE_REQ_SIZE	DCACHE_CMO_INVAL_SET_WAY
DCACHE_REQ_UNCACHEABLE	*
DCACHE_REQ_SID	Corresponding source ID of the requester
DCACHE_REQ_TID	Transaction identifier from the requester
DCACHE_REQ_NEED_RSP	*

\* means don't care

As for any regular request, the request shall follow the **VALID/READY** handshake protocol described in [section 2.4.1](#).

For the sake of design simplification, this operation works as a memory read fence. That is, before handling the operation, the **HPDcache** waits for all pending read misses to complete. Future versions of the HPDcache could wait only for a pending read misses on the same set that is being invalidated.

If the given set and ways contains no valid cachelines, the operation does nothing. However it still works as a memory read fence.

Regarding the latency of this operation, it depends on the time to serve all pending reads. Only one cycle is needed to invalidate the given set and ways because the ways are invalidated simultaneously.

## 5.4 Invalidate the entire cache

With this operation, all the cachelines in the [HPDcache](#) are invalidated.

To perform a complete invalidation of the [HPDcache](#), the requester shall build the request as follows:

Signal	Value
DCACHE_REQ_ADDR	*
DCACHE_REQ_OP	DCACHE_REQ_CMO
DCACHE_REQ_WDATA	*
DCACHE_REQ_BE	*
DCACHE_REQ_SIZE	DCACHE_CMO_INVAL_ALL
DCACHE_REQ_UNCACHEABLE	*
DCACHE_REQ_SID	Corresponding source ID of the requester
DCACHE_REQ_TID	Transaction identifier from the requester
DCACHE_REQ_NEED_RSP	*

\* means don't care

As for any regular request, the request shall follow the **VALID/READY** handshake protocol described in [section 2.4.1](#).

This operation works as a memory read fence. This is, before handling the operation, the [HPDcache](#) waits for all pending read misses to complete.

Regarding the latency of this operation, it has two aggregated components:

- The time to serve all pending reads.
- One cycle per set implemented in the [HPDcache](#) (all ways of a given set are invalidated in simultaneously).

## 5.5 Prefetch a cacheline given its physical address

With this operation, the cacheline corresponding to the indicated physical address is (pre-)fetched into the [HPDcache](#)

To perform a prefetch, the requester shall build the request as follows:

Signal	Value
DCACHE_REQ_ADDR	*
DCACHE_REQ_OP	DCACHE_REQ_CMO
DCACHE_REQ_WDATA	*
DCACHE_REQ_BE	*
DCACHE_REQ_SIZE	DCACHE_CMO_PREFETCH
DCACHE_REQ_UNCACHEABLE	*
DCACHE_REQ_SID	Corresponding source ID of the requester
DCACHE_REQ_TID	Transaction identifier from the requester
DCACHE_REQ_NEED_RSP	Indicates if the requester needs an acknowledgement when the prefetch of the cacheline is completed.
* means don't care	

As for any regular request, the request shall follow the **VALID/READY** handshake protocol described in [section 2.4.1](#).

If the requested cacheline is already in the cache, at the moment the request is processed, this request has no effect. If the requested cacheline is not present in the cache, the cacheline is fetched from the memory and replicated into the cache.

When the prefetch transaction is completed, and the DCACHE\_REQ\_NEED\_RSP signal was set to 1, an acknowledgement is sent to the corresponding requester.



# Chapter 6

---

## Atomic Memory Operations (AMOs)

### Contents

---

6.1	<a href="#">Background</a>	64
6.2	<a href="#">Supported Atomic Memory Operations (AMOs)</a>	64
6.3	<a href="#">Implementation</a>	64
6.4	<a href="#">AMO ordering</a>	65
6.5	<a href="#">LR/SC support</a>	65
6.5.1	<a href="#">LR/SC reservation set</a>	65
6.5.2	<a href="#">SC failure response code</a>	66

---

## 6.1 Background

The Atomic Memory Operations (AMOs) are special load/store accesses that implements a read-modify-write semantic. A single instruction is able to read a data from the memory, perform an arithmetical/-logical operation on that data, and store the result. All this is performed as a single operation (no other operation can come in between the read-modify-write operations).

These operations are meant for synchronization in multi-core environments. To enable this synchronization, AMOs need to be performed on the Point-of-Serialization (PoS), point where all accesses from the different cores converge. This is usually a shared cache memory (when multiple levels of cache are implemented) or the external RAM controllers. Thus, the HPDcache needs to forward these operations to the PoS through the NoC interface.

## 6.2 Supported AMOs

On the interface from requesters, the supported AMOs are the ones listed in table 2.14. The supported AMOs are the ones defined in the atomic (A) extension of the RISC-V ISA specification: *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*[2].

## 6.3 Implementation

This cache does not implement a hardware cache-coherency protocol. Therefore, the software needs to solve the cache obsolescence problem to ensure it reads the last value of the shared data. There are two common ways of doing this:

1. Statically, by placing all shared data into uncacheable segments (never replicated in the HPDcache);
2. Dynamically, by explicitly invalidating local copies of shared data from the HPDcache.

The cache obsolescence problem applies to AMOs. As these operations are used for implementing synchronization mechanism, the manipulated data is by nature shared and need to be coherent between the different caches. The HPDcache implements two different modes for handling AMOs:

### Replicated AMO mode

Forward the AMO to the PoS, and wait for the response with the old data. If the data of the target address is replicated in the HPDcache, the HPDcache computes the new value locally, and updates the target word in the corresponding cacheline. With this solution the modified word with the AMO will be up to date (coherent) with respect to the value in memory. This solution needs explicit treatment from the software. In particular, it requires that synchronization variables are always written (and possibly read) using AMOs. For reading a shared variable, the software has two possibilities: (1) send an AMO that does not modify the memory (e.g. AMOOR with bit-mask equal to zero); (2) invalidate the local cacheline prior to issuing the load instruction.

### Forbid AMO mode

This is a much more strict policy. AMOs can only be performed on uncacheable memory addresses. If requests do not follow this rule, an exception is signalled from the cache to the corresponding requester.

The HPDcache supports both modes, but only one can be active at any given time. The `cfg_error_on_cacheable_amo` configuration bit selects the mode of operation. When this bit is set to 0, the **Replicated AMO mode** is active. When this bit is set to 1, the **Forbid AMO mode** is active.

The HPDcache handle AMOs as non-allocating operations, regardless of the AMO mode described above. This is, AMOs never fetch a replica of the target cacheline from the memory to the cache. If the target



cacheline IS NOT replicated in the cache, the **AMO** modifies the memory. If the target cacheline IS replicated in the cache, the **AMO** modifies both the memory and the cache.

## 6.4 AMO ordering

As specified in the RISC-V ISA specification [2], the base RISC-V ISA has a relaxed memory model. To provide additional ordering constraints, **AMOs** (including **LR/SC**) specify two bits, *aq* and *rl*, for *acquire* and *release* semantics.

The **HPDcache** always ignores *aq* and *rl* bits. It considers that they are always set. Hence, **HPDcache** handles **AMOs** always as sequentially consistent memory operations. The **HPDcache** waits for all pending read and write operations to complete before serving the **AMO** request.

This behavior implies that when the **HPDcache** forwards an **AMO** to the **NoC**, it will be the only pending request from the **HPDcache**. In addition, no new requests from the requesters are served until the **AMO** is completed.

## 6.5 LR/SC support

Load-Reserved (**LR**) and Store-Conditional (**SC**) are part of the Atomic (A) extension of the RISC-V ISA specification [2]. These instructions allow *"complex atomic operations on a single memory word or double-word"*.

The **HPDcache** fully supports all the instructions of the A extension of the RISC-V ISA, including **LR** and **SC** operations.

In the specification of these instructions in the RISC-V ISA document, some details are dependent to the implementation. Namely, the size of the reservation set and the return code of a **SC** failure.

### 6.5.1 LR/SC reservation set

When a requester executes a **LR** operation, it "reserves" a set of bytes in memory. This set contains at least the bytes solicited in the request but may contain more. RISC-V ISA defines two sizes for **LR** operations: 4 bytes or 8 bytes. **The **HPDcache** reserves 8-bytes (double-word) containing the addressed memory location regardless of whether the **LR** size is 4 or 8 bytes.** The start address of the reservation set is a 8-bytes aligned address.

When the **LR** size is 8 bytes, the address is also aligned to 8 bytes (section 2.4.2). In this case, the reservation set matches exactly the address interval defined in the request. When the **LR** size is 4 bytes, there are two possibilities: (1) the target address is not aligned to 8 bytes. The start address of the reservation set contains additional 4 bytes before the target address ; (2) the target address is aligned to 8 bytes. The reservation set starts at the target address but contains additional 4 bytes after the requested ones.

In summary, in case of **LR** operation, the reservation set address range is computed as follows:

$$\text{reservation\_set} = \begin{cases} \lfloor \text{DCACHE\_REQ\_ADDR}/8 \rfloor \times 8 & (\text{start\_address}) \\ (\lfloor \text{DCACHE\_REQ\_ADDR}/8 \rfloor \times 8) + 8 & (\text{end\_address}) \end{cases}$$

**When a requester executes a **SC** operation, the **HPDcache** forwards the operation to the memory ONLY IF the bytes addressed by the **SC** are part of an active reservation set.** If the **SC** accesses a smaller number of bytes than those in the active reservation set but within that reservation set, the **SC** is still forwarded to the memory.

The [HPDcache](#) keeps an unique active reservation set. If multiple requesters perform [LR](#) operations, the unique active reservation set is the one specified by the last [LR](#) operation.

After a [SC](#) operation, the unique active reservation set, if any, is invalidated. This is regardless whether the [SC](#) operation succeeds or not.

### 6.5.2 [SC](#) failure response code

The RISC-V ISA [2] specifies that when a [SC](#) operation succeeds, the core shall write zero into the destination register of the operation. Otherwise, in case of [SC](#) failure, the core shall write a non-zero value into the destination register.

The [HPDcache](#) returns the status of an [SC](#) operation into the DCACHE\_RSP\_RDATA ([table 2.3](#)) signal of the response interface to requesters. The following table specifies the values returned by the [HPDcache](#) into the DCACHE\_RSP\_RDATA signal in case of [SC](#) operation.

Case	Return value (status)
SC success	0x0000_0000
SC failure	0x0000_0001

Depending on the specified size (DCACHE\_REQ\_SIZE) in the request ([table 2.3](#)), the returned value is extended with zeros on the most significant bits. This is, if the SC request size is 8 bytes, and the SC is a failure, then the returned value is 0x0000\_0000\_0000\_0001.

In addition, if the width of the DCACHE\_RSP\_RDATA signal is wider than the size of the SC request, the return value is replicated CONF\_DCACHE\_REQ\_WORDS ([table 1.1](#)) times.

# Chapter 7

---

## Hardware Memory Prefetcher

### Contents

---

7.1	Triggering . . . . .	68
7.2	Activation/Deactivation Policies . . . . .	69
7.3	CSRs . . . . .	69
7.4	Prefetch Request Algorithm . . . . .	71
7.5	Prefetch Abort. . . . .	71

---

In order to predict future data accesses and reduce the data cache miss rate, the cache implements a programmable hardware mechanism allowing to prefetch cachelines before they are actually requested.

The [HPDcache](#) implements a prefetcher that contains multiple prefetch engines. Each prefetch engine works independently, and simultaneously. A round-robin arbiter at the output of the prefetcher allows to select one prefetch request from one of the engines per cycle. This arbiter guarantees the correct behavior when multiple prefetch engines are active.

Each engine, if activated, fetches a stream of cachelines. A stream is defined as a sequence of prefetch requests. An engine reads one or multiple blocks of a given number of cachelines. The first block starts at a given base cacheline. Between blocks, one can configure a given address offset (also known as the stride).

The four parameters (base cacheline, number of cachelines in a block, number of blocks, and the stride) of each stream (one per engine) are configured through dedicated [CSRs](#).

## 7.1 Triggering

A given prefetcher engine starts operating when the following conditions are met:

1. Each engine implements an enable bit in its dedicated [CSRs](#). This enable bit shall be set to 1 to allow the triggering of a given prefetcher engine.
2. Each enabled engine (condition 1 is met), snoops on the requests ports from the requesters. If there is a match between the issued address and the configured base cacheline of the engine, the engine starts the prefetching.

Once an engine starts its operation, it does not snoop anymore the request ports. At that moment, it issues a sequence of prefetch operations starting from `base_cline`, until the cacheline in the equation below. When the last cacheline is reached, the behavior of the engine is described in the following section.

$$\text{end\_cline} = \text{base\_cline} + (\text{Nblocks} + 1) \times (\text{Stride} + 1)$$

A prefetch operation behaves as a read in the cache, but no data is expected in response by the prefetcher. This means, that prefetch operations do not need to enable the data array of the cache (thus reducing the energy consumption for this operation). Prefetch operations only access cache directory memories to check if the requested cacheline is cached or if it needs to be fetched from the memory.

### Programming note

As explained in this section, a requester needs to issue a load transaction within the base cacheline of an engine to start its operation.

When the requester is a programmable processor core, an additional feature that could be implemented in the core is a software prefetch instruction. This instruction would allow the software to prefetch a given cacheline, without stalling the core while waiting the response from the cache. Such instruction could also be used to start an enabled prefetcher engine.

In RISC-V cores, one possibility to implement this software prefetch instruction could be to use the following:

```
lw x0, offset(rs1)
```

As the `x0` register is always equal to zero, the data is dropped. Therefore, an efficient implementation of this instruction in the core consists on forwarding the load to the L1 data cache but do not wait for the response.

## 7.2 Activation/Deactivation Policies

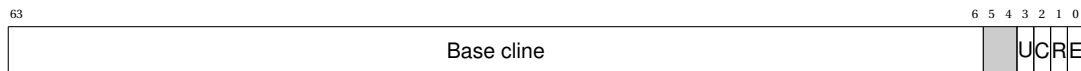
The prefetcher engines implement different automatic activation/deactivation policies:

Policies	
<b>Disarm when finished</b>	When the engine completes the configured stream, it is automatically disabled.
<b>Rearm when finished</b>	When the prefetcher completes the configured stream, it does not disable. However, it stops and waits to be triggered again. At that point, the base cacheline <b>CSR</b> of the engine saves the last accessed cacheline plus the stride. This is, it saves the next address to prefetch. The <b>CSRs</b> for the number of blocks, cachelines per block and stride keep their originally configured values.
<b>Rearm and Cycle when finished</b>	In this policy, the prefetch engine behaves as in the "Rearm when finished" policy, but the base cacheline <b>CSR</b> is reset to the originally configured value.

## 7.3 CSRs

Each prefetcher engine has three dedicated **CSRs**.

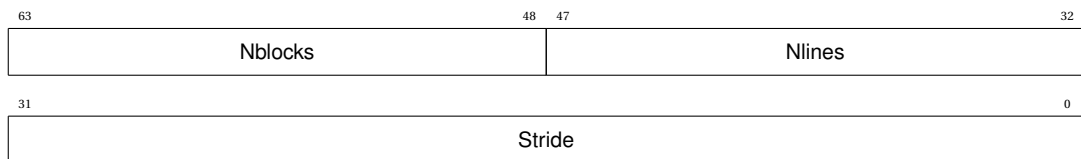
- Base cline (base cacheline) - `cfg_hwpf_base_engine` (see [table 4.2](#))



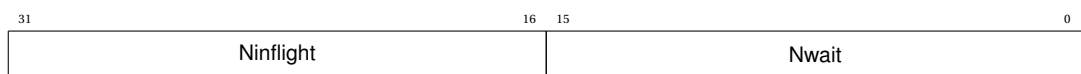
**E:** Enable bit  
**R:** Rearm bit  
**C:** Cycle bit  
**U:** Upstream bit

Mode	R	C
Disarm when finished	0	X
Rearm when finished	1	0
Cycle and rearm when finished	1	1

- Parameters - `cfg_hwpf_param_engine` (see [table 4.2](#))



- Throttle - `cfg_hwpf_throttle_engine` (see [table 4.2](#))



### Stride parameter

It is an unsigned, one-based (value + 1), 32-bits wide value. The stride is in number of cachelines. This means that the stride is always a multiple of (DCACHE\_CL\_WIDTH/8) bytes.

**Nblocks parameter**

It is an unsigned, one-based (value + 1), 16-bits wide value. This value corresponds to the number of blocks to prefetch. The 16-bit value allows the prefetcher to prefetch up to 65536 blocks (of at least one cacheline). This parameter is clearly over-dimensioned with respect to the usual capacity of the [HPDcache](#) (e.g. 512, 64-byte, cachelines with a 32KB capacity).

**Nlines parameter**

It is an unsigned, one-based, 16-bits wide value. It indicates the number of cachelines within blocks. As the number of bits is 16, the maximum number of cachelines in a given block is 65536.

**Nwait parameter**

It is an unsigned, one-based, 16-bits wide value. It defines the number of cycles (plus 1) between two requests of the prefetcher engine. The zero value indicates that the engine can issue a request every cycle.

**Ninflight parameter**

It is an unsigned, zero-based, 16-bits wide value. It defines the maximum number of in-flight (sent but not yet acknowledged) transactions from the prefetcher engine. This parameter allows to throttle the memory bandwidth solicited by the prefetcher engine. The zero value indicates that the number of in-flight transactions is unlimited.

**U bit**

When this upstream bit is set, prefetch operations targets the next level in the memory hierarchy. In this case, upstream prefetch operations do not allocate cachelines in the L1 data-cache. These are forwarded to the next memory level that can then prefetch the requested address. THIS BIT IS NOT CURRENTLY IMPLEMENTED AND IGNORED.

**C bit**

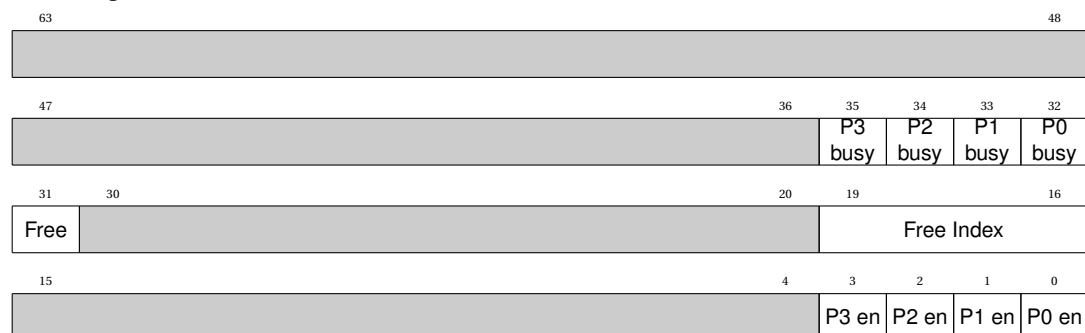
This bit is only considered when the R bit (rearm) is also set. When this cycle bit is set, after the prefetcher engine completes the prefetch stream, it resets the base cacheline to the originally configured one (see [section 7.2](#)).

**R bit**

When this rearm bit is set, after the prefetcher engine completes the prefetch stream, it "rearms" itself (remains enabled), and snoops for core requests (see [section 7.2](#)). The address it snoops after it finishes depends on the Cycles (C) bit. If the C bit is set, the behavior is described here above. If the C bit is unset, after the prefetcher engine finishes, the snoop address is set to end\_cline.

There is also a global status register for the prefetcher to monitor the status of the different prefetcher engines:

- Status register

**P0-P3 enable bits**

Indicate if the corresponding prefetcher is enabled.

**Free index bits**

Indicate the index (decimal) of the first available prefetcher from 0 to 3. The software can use this information to easily compute the address offset of the configuration registers of the target prefetcher engine.

**Free bit**

It is set when the *Free Index* is valid, this is, when there is effectively a free prefetcher.

**P0-P3 busy bits**

Indicate if the corresponding prefetcher is busy (it is enabled and active).

## 7.4 Prefetch Request Algorithm

[Algorithm 7.1](#) shows how a prefetch engine calculates the address to prefetch and the operation of the throttling mechanisms.

## 7.5 Prefetch Abort

It is possible for the user to abort an active prefetch sequence from an engine. To do that, the user can reset to 0 the enable(E) bit in the base\_cline [CSR](#) register of the corresponding prefetcher engine.

Such action, makes the corresponding target engine to stop its current sequence of prefetch requests. If there were inflight not-yet-acknowledge requests from that engine, it will wait for the corresponding acknowledgements. During this time, the prefetcher engine is not usable, and its corresponding busy bit in the Status [CSR](#) is kept set to 1. While the busy bit is set to 1, any write in [CSR](#) registers of that prefetcher engine has no effect on the engine behavior. However, the modified [CSR](#) registers will keep the written values.

When all acknowledgements are received, the corresponding prefetcher engine has its busy bit set to 0. All other [CSR](#) of the prefetcher engine keep their configured values. At this point, the prefetcher engine is usable and can be reconfigured normally.

```

1  const bit [63:0] LINES_PER_BLOCK = Nlines + 1;
2  const bit [63:0] BLOCK_INCREMENT = Stride + 1;
3  bit [63:0] block_nline;
4
5  // Iterates over the blocks of cachelines
6  block_nline = Base_cline;
7  for (nb = 0; nb < (Nblocks + 1); nb++)
8      // Iterates over the cachelines within each block.
9      for (nl = 0; nl < LINES_PER_BLOCK; nl++) {
10         // Skip the first cacheline of the first block as it is already requested by the
11         // request triggering the prefetcher
12         if ((nl == 0) && (nb == 0)) continue;
13
14         if (Ninflight > 0) {
15             // Wait while the number of in-flight prefetch requests is equal to the configured
16             // threshold (Ninflight). This is a throttling mechanism.
17             // The inflight counter is decremented by another process each time the prefetcher
18             // receives an acknowledgement for an inflight request.
19             while (inflight >= Ninflight) {
20                 wait (1); // cycles
21             }
22         }
23
24         // Send the prefetch operation for the calculated cacheline.
25         // Cachelines are contiguous within a block.
26         prefetch_address((block_nline + nl)*64);
27
28         // The inflight counter is incremented each time the prefetcher sends a prefetch request.
29         inflight++;
30
31         // Wait a given number of cycles between two prefetch requests.
32         // This is a throttling mechanism
33         wait (Nwait + 1); // cycles
34     }
35
36     // The first cacheline of a block is offset (defined by the Stride)
37     // with respect to the previous block.
38     block_nline += BLOCK_INCREMENT;
39 }
40
41 // If the cycle bit is not set, update the Base_cline with the
42 // address that would follow the last accessed one.
43 if (!cfg.base_cline.c) {
44     Base_cline = block_nline;
45 }

```

Figure 7.1: Request issuing algorithm of prefetch engines



# Appendix A

---

## Appendices

### Contents

---

A.1	<a href="#">RAM macros</a>	74
A.2	<a href="#">Implementations</a>	74
A.2.1	<a href="#">EPI Accelerator and RHEA Chip</a>	74

---

## A.1 RAM macros

This cache uses memory arrays in multiple subcomponents. When targeting [ASIC/FPGA](#) implementations integrating this cache, memory arrays shall be implemented using technology-specific [SRAM](#) macros. In the case of [FPGA](#) implementations, this is less critical because synthesis tools for [FPGA](#) usually select automatically embedded RAMs when possible.

[Table A.1](#) summarises the instances of RAM macros implemented in the [HPDcache](#). This table has:

1. The path in the RTL model where that memory array is found;
2. a reference to the section that gives details about their dimensions, the number of instances and their content;
3. the number of read/write ports;
4. the read/write latency.

Table A.1: Summary of RAM macros in the [HPDcache](#)

MSHR	
<b>RTL Instance</b>	<hpdcache_instance>.hpdcache_miss_handler_i.hpdcache_mshr_i.mshr_sram
<b>Details</b>	<a href="#">Section 3.2.1</a>
<b>Latency</b>	1 clock cycle (RW)
<b>Ports</b>	1RW
Cache Directory	
<b>RTL Instance</b>	<hpdcache_instance>.hpdcache_ctrl_i.hpdcache_memctrl_i.hpdcache_memarray_i.dir_sram[i]
<b>Details</b>	<a href="#">Section 3.5.1</a>
<b>Latency</b>	1 clock cycle (RW)
<b>Ports</b>	1RW
Cache Data	
<b>RTL Instance</b>	<hpdcache_instance>.hpdcache_ctrl_i.hpdcache_memctrl_i.hpdcache_memarray_i.data_sram[i]
<b>Details</b>	<a href="#">Section 3.5.1</a>
<b>Latency</b>	1 clock cycle (RW)
<b>Ports</b>	1RW

## A.2 Implementations

### A.2.1 EPI Accelerator and RHEA Chip

In the context of the European Processor Initiative (EPI) project, in the accelerator stream [\[4\]](#), this cache is used as the L1 data cache for the VRP [\[5\]](#) accelerator that is integrated in both, the EPI accelerator (EPAC1.5) test-chip and the RHEA chip.

The parameters of the cache are the following on those implementations:

---

<b>Capacity</b>	32 KBytes
<b>Sets</b>	128
<b>Ways</b>	4
<b>Line Size</b>	64 bytes
<b>Physical Address Width</b>	49 bits
<b>Write Policy</b>	Write-Through
<b>Maximum Access width (Requester-side)</b>	32 bytes per cycle



---

## Bibliography

- [1] Arm Limited, “AMBA AXI and ACE Protocol Specification,” en, 110 Fulbourn Road, Cambridge, England, Specification Document, 2020, p. 440.
- [2] A. Waterman and K. Asanovic. “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA.” (2019), [Online]. Available: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>.
- [3] A. Waterman, K. Asanovic, and J. Hauser. “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.” (2021), [Online]. Available: <https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf>.
- [4] “European Processor Initiative (EPI) Accelerator Stream.” (2022), [Online]. Available: <https://www.european-processor-initiative.eu/accelerator/>.
- [5] Y. Durand, E. Guthmuller, C. Fuguet, J. Fereyre, A. Bocco, and R. Alidori, “Accelerating variants of the conjugate gradient with the variable precision processor,” in *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022, pp. 51–57. DOI: [10.1109/ARITH54963.2022.00017](https://doi.org/10.1109/ARITH54963.2022.00017).