# Zicmobase Extension, Version 0.2

## Table of Contents


1. Introduction . . . 1
   1.1. Goals . . . 1
   1.2. Non-Goals . . . 2
2. Overview . . . 2
   2.1. Memory and Caches in RISC-V . . . 2
   2.2. System Topology . . . 4
   2.3. FIXME: Memory Ordering . . . 8
   2.4. FIXME: Discovery . . . 9
   2.5. Traps . . . 9
3. Formats . . . 12
   3.1. Instructions . . . 12
   3.2. CSRs . . . 12
4. Instructions . . . 17
   4.1. Cache Block Management Instructions . . . 17
   4.2. Cache Block Write Instruction . . . 18
   4.3. Cache Block Hint Instructions . . . 19
   4.4. FIXME: Cache Block Discovery Instruction . . . 20
5. Opcode Map . . . 20


# 1. Introduction

The Zicmobase extension adds a base set of *cache management operation* (or *CMO*) instructions and CSRs to the RISC-V architecture. The base extension includes several classes of instructions that operate on cache blocks:

- A set of cache block management instructions, i.e. `CBO.INVAL`, `CBO.CLEAN`, and `CBO.FLUSH`
- A cache block write instruction, i.e. `CBO.ZERO`
- A set of cache block hint instructions, i.e. `PREFETCH.R`, `PREFETCH.W`, `PREFETCH.I`, and `DEMOTE`
- A cache block size discovery instruction, i.e. `CBO.SIZE` (TBD)

The execution of these instructions in various privilege modes is controlled by a set of CSRs. In addition, this specification introduces architectural abstractions for caches and system topologies to support portable software across various system designs.

## 1.1. Goals

- Define instructions that operate on an architectural abstraction of caches and system topology

- Specify the interactions of CMOs with the RVWMO memory ordering model
- Support various software use-cases such as software-managed cache coherence and non-coherent I/O
- Allow execution of CMOs in lower privilege modes, including U-mode

## 1.2. Non-Goals

The following goals are intentionally deferred to a later specification:

- Cache management operations on microarchitectural characteristics, e.g. set/way operations or whole cache operations
- Specifying the behavior of cache management operations on different cache block sizes
- Performance has *not* been a primary goal for this version of the specification (though the specification is mindful not to hinder performance)

# 2. Overview

In general, a CMO instruction initiates an operation on a set of caches based on instruction type, CSR state, physical memory attributes, and other architectural state. A CMO instruction specifies an effective address, which may be translated by various translation mechanisms into a physical address. A hart may then perform a subsequent memory access with that physical address in order to operate on various caches throughout a system.

## 2.1. Memory and Caches in RISC-V

As specified in the RISC-V base and privileged architectures, memory is organized as an array of bytes, with a given physical address identifying a particular memory location. Each memory location is further associated with a main memory region, an I/O device region, or a vacant region. In this extension, a *memory agent* services memory accesses to a set of memory locations that comprise a memory region (of any type), while a *system agent* performs memory accesses to memory locations and may be a hart or an I/O device (or *device* for short).

> *A particular I/O device may be a memory agent, a system agent, or both.*

Caches in RISC-V reduce average memory latency by buffering copies of data from frequently accessed memory locations close to system agents. Specifically, a cache buffers copies of *cache blocks*, each of which consists of a contiguous, non-overlapping, naturally aligned power-of-two (NAPOT) set of bytes and corresponds to a similarly NAPOT set of memory locations and associated physical addresses. A system may make multiple copies of a given cache block, and these copies may be transferred to and from various caches and memory. The underlying memory locations for a cache block form the *backing storage* for the block, and the organization of a cache and the size of a cache block are *implementation-defined*.

RISC-V defines three types of caches: A *private cache* is accessed by a single system agent, while a

*shared cache* is accessed by more than one system agent. In addition, a *memory cache* is localized to memory agent and is effectively "invisible" to the system agents that access a given memory location. (In other words, a memory cache may serve as a proxy for the actual physical memory location.) The base set of cache management operations operates on private and shared caches only and is *not* guaranteed to operate on memory caches.

***FIXME:*** types of cache blocks?

### 2.1.1. Cache Block States

Relative to a given cache, a cache block may be in one of three states:

- *Invalid* — the cache block is *not* present in the cache
- *Unmodified* — the cache block is present in the cache and *this* copy has *not* been written by a system agent
- *Modified* — the cache block is present in the cache and *this* copy has been written by a system agent

All caches differentiate between invalid and unmodified cache blocks, and a cache may or may not track modified cache blocks depending on *implementation-defined* write policies, e.g. write-back or write-through, respectively. A cache, therefore, may implement only two states (invalid and unmodified) or all three states (invalid, unmodified, and modified). Additionally, a cache block in either the unmodified or modified state is considered to be *valid*.

In addition, based on *implementation-defined* system-wide cache policies and mechanisms, an unmodified cache block may *not* be consistent with memory, and a modified copy may be present in another cache at the same time. Furthermore, the state of the cache block may also be communicated when the block is transferred to or from the cache. These policies and mechanisms are typically defined by a cache coherence protocol (see the System Topology section) and are beyond the scope of this specification.

A transition from invalid to valid occurs when a cache allocates a cache block, resulting in a transfer of the block to the cache from another cache or memory (performing a *memory read* in the latter case). To the extent allowed by the base and privileged architectures, a cache may allocate a cache block for any physical address at any time for any reason. The cache block is allocated in either the unmodified or modified state depending on whether the cache tracks modified cache blocks and whether an unmodified or modified cache block is transferred to the cache.

A transition from valid to invalid occurs when a cache deallocates the cache block, potentially resulting in a transfer of the block from the cache to another cache or memory (performing a *memory write* in the latter case). A cache may deallocate a cache block at any time for any reason. If the cache block is unmodified, the cache may transfer the cache block to another cache but *must not* transfer the cache block to memory to avoid overwriting memory with potentially stale data. (The behavior of CMO instructions becomes UNSPECIFIED if a cache transfers an unmodified cache block to memory.) If the cache block is modified, the cache *must* transfer the modified block to another cache or memory.

Finally, if the cache tracks modified cache blocks, a transition from unmodified to modified occurs whenever a system agent executes an instruction, e.g. a store instruction, or performs an operation,

e.g. a page table entry write, that writes to a cache block that has been transferred to the cache. To cause the transition, the instruction or operation only needs to be classified as a data write that *may* change the data values in the cache block; the data write is *not* required to change the data values. If the cache does *not* track modified cache blocks, the data write *must* be propagated either to another cache that does track modified cache blocks or to memory. In such a cache, the data write *must* update the data values of the cache block, or the cache block *must* transition to invalid.

### 2.1.2. CMO Effects on Caches

CMO instructions may also affect the state of a cache block. A cache block management instruction performs one of the following operations, which may change the state of a cache block and may result in a transfer of data from the cache:

- An *invalidate* operation *must* change the state of a valid cache block to invalid; otherwise, no state change occurs. The operation may transfer the cache block if its state was valid before the operation; however, the transfer of the cache block is *not* required.
- A *clean* operation *must* change the state of a modified cache block to unmodified, although the operation may change the state of a valid cache block to invalid; otherwise, no state change occurs. The operation *must* transfer the cache block if its state was modified before the operation and may transfer the cache block if its state was unmodified before the operation.
- A *flush* operation *must* change the state of a valid cache block to invalid; otherwise, no state change occurs. The operation *must* transfer the cache block if its state was modified before the operation and may transfer the cache block if its state was unmodified before the operation.

A cache block write instruction effectively performs a series of byte atomic data write operations, similar to a series of store byte instructions. An implementation may or may not update the entire cache block atomically.

Finally, a cache block hint instruction may perform an *implementation-defined* operation or no operation, the latter of which does not affect the state of the cache.

## 2.2. System Topology

A memory access from a CMO instruction proceeds along *memory access path* (or *path* for short) from a given system agent toward a given memory location. A path is determined primarily by the following characteristics:

- The physical address of the memory access
- The memory attributes associated with the memory access

The physical address identifies the memory location being accessed and is a function of the effective address specified by a CMO instruction and any enabled translation mechanisms. In addition, the memory attributes for a memory access may be specified by either architectural or *implementation-defined* mechanisms. Other factors, such as type of operation, may also influence the path.

> *The memory attributes that typically affect a path are related to cacheability and coherence; however, other memory attributes may affect a path.*
>
> *From the same system agent, paths for memory accesses with the same memory attributes to different memory locations may be different. Likewise, paths for memory accesses with different memory attributes to the same memory location may be different.*

Paths from different system agents to the same memory location converge at a *point of convergence* (or *PoC*), and from a given PoC, the paths that have converged do not diverge. In addition, the memory accesses on those paths are ordered, and remain ordered, with respect to each other from a PoC until the memory accesses can be completed. A PoC is *not* required to order memory accesses to different memory locations. Once an order has been established, those memory accesses are considered to be *access ordered* and cannot be reordered within the system.

> *This ordering definition is necessary to implement cache coherence protocols and forms the basis for the memory ordering model below. Effectively, a PoC establishes a coherence order for a given memory location with respect to a given set of agents.*

For every memory location in a system, the *point of convergence of memory*, or *PoC-memory*, is the PoC where all paths for a given memory location converge, independent of all other characteristics that define a path. At the PoC-memory, all accesses to a memory location have been access ordered, and the CMO instructions defined in this extension are guaranteed to operate on a path up to the PoC-memory.

***FIXME:*** Define other standard PoCs?

> *This extension does not prohibit system agents from bypassing the PoC-memory to access a memory location, nor does the extension prohibit memory caches beyond the PoC-memory. However, in such a system, software cannot expect the currently defined cache operations to have the desired effects with respect to those system agents or caches.*
>
> *Additional system topology beyond the PoC-memory may be specified in future extensions. For example, additional points of convergence may be defined to manage memory caches, or various points of persistence may be defined to support different classes of storage.*

A system may define additional custom PoCs before the PoC-memory, and when such a PoC is specified in a CMO instruction, the instruction *must* operate on a given path up to the custom PoC and may operate on the path up to the PoC-memory. A CMO instruction is *not* required, however, to operate on the path beyond a custom PoC.

> *The above definition allows an implementation to perform all operations to custom PoCs before the PoC-memory as if such operations were performed to the PoC-memory.*

While traversing a given path, a memory access from a CMO instruction operates on the caches up

to the specified PoC. Between a system agent and the first PoC on the path, the memory access operates on private caches, and between subsequent PoCs, the memory access operates on shared caches. There is no requirement, however, for any caches to be present either between a system agent and the first PoC or between subsequent PoCs. Caches on the path are accessed *directly* by the memory access. Additional caches on the paths that converge at a given PoC may be accessed *indirectly* depending on the memory attributes associated with a memory access and any *implementation-defined* cache coherence mechanisms.

Systems may implement hardware cache coherence mechanisms to ensure that the copies in a set of caches remain *coherent* with respect to each other, i.e. the copies in the set of caches appear to have the same data values, regardless of which cache in the set is accessed. The set of caches on which hardware can maintain this property corresponds to a *hardware coherence domain* (or *domain* for short), which may consist of any number of caches, including an individual cache. Only a subset of the caches in a domain may be accessed depending on the memory attributes of a memory access and the cache coherence protocol.

> *A hardware cache coherence protocol may add additional cache states and may cause additional cache block state transitions. The effects of a hardware cache coherence protocol on cache block states are beyond the scope of this specification.*

If two caches are in different domains, the copies in those caches are *non-coherent* with respect to each other. In addition, two copies in different caches within the same domain are also non-coherent with respect to each other if the memory attributes of a memory access do not require both caches to be accessed. Non-coherent copies may appear to have different data values, or the copies may appear to have the same data values. Software may enforce coherence on non-coherent copies using CMO instructions.

> *The term* coherent *implies a guarantee of coherence, while the term* non-coherent *implies only the lack of such a guarantee, not a guarantee of non-coherence.*

Below are some properties/implications of the above definitions:

- Paths form a tree with the system agents as leaves and the PoC-memory as the root; intermediate PoCs are nodes on the tree, while caches lie on the edges
  - For example, a private L1 and L2 cache lie on the edge between a system agent and the first PoC
- PoCs establish a hierarchy
  - At each PoC, the set of agents whose memory accesses are ordered is the union of the sets defined by the previous PoCs
- Memory accesses on a path obey uniprocessor semantics
- Caches on the path from a domain PoC to the next PoC are effectively part of the domain
- Caches between PoCs are effectively part of the same domain
  - The access order of caches between PoCs is implementation-defined (?)
- PoCs and domains
- PoCs are accessed serially (?)

## Ignore Me

Random stuff being worked on:

*PoCs-domain*

memory accesses due to loads and stores can be ordered anywhere in a domain, but the ultimate point of ordering for the domain is the PoC-domain. memory accesses due to CMOs operate to a PoC-domain and guarantee the whole domain has been operated on

Domains correspond to particular PoCs. A *PoC-domain* (working title) is defined to be the PoC at which all paths that pass through any cache in the domain before the PoC converge. Systems may have any number of PoCs-domain visible to software via memory attributes or as custom PoCs. In addition, a domain may be asymmetric with respect to the paths that converge at a PoC-domain. Specifically, a memory access on one path may indirectly access the caches on a second path, while a memory access on the second path may *not* indirectly access the caches on the first.

SW view: POCs define sets of agents that communicate coherently without SW help

***FIXME:*** Domains can be mapped outward; within the frontier, outside the frontier

Is there such a thing as a PoC-hart that represents uniprocessor ordering? That makes the tree definition a bit more easy since a leaf becomes a PoC (but nothing is really converging, unless you consider loads and stores as separate things converging)

Structural definition: relative to system agent. L1 is the "first" cache accessed, etc.

### 2.2.1. FIXME: PMA Behaviors

FIXME: Coherence and cacheability attributes…

Ignore cacheability to enable changes in attribute

Non-coherent implies that caches may not be accessed indirectly.

## 2.3. FIXME: Memory Ordering

### 2.3.1. Preserved Program Order

The preserved program order (abbreviated *PPO* below) rules are defined by the RVWMO memory ordering model. How the operations resulting from CMO instructions fit into these rules is described below.

For cache block management instructions, the resulting invalidate, clean, and flush operations behave as stores in the PPO rules subject to one additional overlapping address rule. Specifically, if *a* precedes *b* in program order, then *a* will precede *b* in the global memory order if:

- *a* is an invalidate, clean, or flush, *b* is a load, and *a* and *b* access overlapping memory addresses

> *The above rule ensures that a subsequent load operation in program order never appears in the global memory order before a preceding invalidate, clean, or flush operation to an overlapping address.*

For cache block write instructions, the resulting write operations simply behave as stores in the PPO rules.

As cache block hint instructions do not modify architectural memory state, the resulting operations are *not* ordered by the PPO rules.

### 2.3.2. Load Values

In addition, an invalidate operation changes the set of values that may be returned by a load. In particular, a third condition is added to the Load Value Axiom:

3. If an invalidate precedes *i* in program order and operates on a byte, and no store to that byte appears in program order or in the global memory order between the invalidate and *i*, the load value is *implementation-defined*

What does global memory order mean for software managed coherence:

- Can describe global to mean "global" for all agents and domains (single universe)
- Can describe global to mean "global" for some agents and domains (a multiverse)

The above definition is written using a multiverse definition for global memory order. A single universe definition would constrain the result to orders that could be produced by other agents. Maybe…?

### 2.3.3. Ordering Events

Ordering event for CMO: access ordered at explicit PoC

Ordering event for load: access ordered in the implicit domain

Ordering event for store: access ordered in the implicit domain

PPO specifies the order of the ordering events

## 2.4. FIXME: Discovery

FIXME: HW vs. SW

For now, fixed size across all harts and devices that share a domain

## 2.5. Traps

### 2.5.1. Illegal Instruction and Virtual Instruction Exceptions

Cache block management instructions and cache block write instructions may take an illegal instruction exception depending on the *current privilege mode* and the state of the CMO control registers described in the CSRs section. The current privilege mode refers to the privilege mode of the hart at the time the instruction is executed.

Cache block hint instructions do *not* take illegal instruction exceptions.

Additionally, CMO instructions do *not* take virtual instruction exceptions.

### 2.5.2. Page Fault and Guest-Page Fault Exceptions

During address translation, CMO instructions may take a page fault depending on the type of instruction, the *effective privilege mode* (as determined by the `MPRV`, `MPV`, and `MPP` bits in `mstatus`) of the resulting access, and the permissions granted by the page table entry (PTE). If two-stage address translation is enabled, CMO instructions may also take a guest-page fault.

Cache block management instructions require a valid translation (`V=1`) and either read (`R=1`) or execute (`X=1`) permission and, if applicable, user access (`U=1`) in the effective privilege mode. If these

conditions are *not* met, the instruction takes a store/AMO page fault exception. In addition, `CBO.INVAL` instructions may take a store/AMO page fault exception depending on the state of the CMO control registers described in the CSRs section and whether the access has been granted write permission by the PTE.

Cache block write instructions require a valid translation (`V=1`) and write (`W=1`) permission and, if applicable, user access (`U=1`) in the effective privilege mode. If these conditions are *not* met, the instruction takes a store/AMO page fault exception.

If G-stage address translation is enabled, the above instructions take a store/AMO guest-page fault if the G-stage PTE does *not* allow the access.

Cache block hint instructions require a valid translation (`V=1`) and either read (`R=1`) or execute (`X=1`) permission and, if applicable, user access (`U=1`) in the effective privilege mode. If these conditions are *not* met, however, the instruction does *not* take a page fault or guest-page fault exception and retires without accessing memory.

FIXME: PREFETCH.W interacts with LR/SC; doesn't require W=1

**Effect of other `xstatus` bits**

The `mstatus.MXR` bit (also `sstatus.MXR`) and the `vsstatus.MXR` bit do *not* affect the execution of CMO instructions.

The `mstatus.SUM` bit (also `sstatus.SUM`) and the `vsstatus.SUM` bit do *not* affect the execution of CMO instructions beyond modifying permissions for S/HS-mode and VS-mode accesses as specified by the privileged architecture.

### 2.5.3. Access Fault Exception

A CMO instruction may take an access fault exception, as detailed in the privileged architecture specification, that interrupts the address translation process. Assuming the address translation process completes with a valid translation, a CMO instruction may also take an access fault exception depending on the type of instruction, the effective privilege mode of the resulting access, and the permissions granted by the physical memory protection (PMP) unit and the physical memory attributes (PMAs).

> *For now, we assume two things about PMAs:*
>
> 1. *PMAs are the same for all physical addresses in a cache block*
> 2. *Memory that can be cached cannot be write-only*

Read (`R`), write (`W`), and execute (`X`) permissions are granted by the PMP and the PMAs. Although the PMP may grant different permissions to different physical addresses in a cache block, the PMAs for a cache block *must* be the same for *all* physical addresses in the cache block and read permission *must* be granted if write permission has been granted. If these PMA constraints are *not* met, the behavior of CMO instruction is UNSPECIFIED.

For the purposes of access fault determination, *joint permission* is granted for a given physical address when the same access type is allowed by both the PMP and the PMAs for that physical address. For example, joint read permission implies that both the PMP and PMAs allow a read access. In addition, for a given cache block, *partial joint write permission* implies that joint write permission has been granted to only *some* of the physical addresses in the cache block, while *full joint write permission* implies that joint write permission has been granted to *all* physical addresses in the cache block.

Cache block management instructions require either joint read or joint execute permission for *all* accessed physical addresses. If this condition is *not* met, the instruction takes a store/AMO access fault exception. In addition, `CBO.INVAL` instructions may take a store/AMO access fault exception depending on the state of the CMO control registers described in the CSRs section and whether the access has been granted partial joint write permission by the PMP and PMAs.

Cache block write instructions require full joint write permission. If this condition is *not* met, the instruction takes a store/AMO access fault exception.

Cache block hint instructions require either joint read or joint execute permission for *all* accessed physical addresses. If this condition is *not* met, however, the instruction does *not* take an access fault exception and retires without accessing memory.

### 2.5.4. Address Misaligned Exception

CMO instructions do *not* generate address misaligned exceptions.

### 2.5.5. Breakpoint Exception

CMO instructions may generate breakpoint exceptions (or may cause other debug actions) subject to the general trigger module behaviors specified in the debug architecture. When `type=2` (i.e. `mcontrol`), the behavior of a trigger for load and store address matches is UNSPECIFIED for CMO instructions. When `type=6` (i.e. `mcontrol6`), the behavior of a trigger for load and store address matches is based on the following classification of a CMO instruction:

- A cache block management instruction is both a load and a store
- A cache block write instruction is a store
- It is *implementation-defined* whether a cache block hint instruction is both a load and a store or neither a load nor a store

Load and store data matches for all CMO instructions are UNSPECIFIED.

> *An implementation may convert cache block hint instructions into NOPs prior to executing the instruction. Load and store matches are not applicable in such an implementation.*
>
> *For load and store address matches on a CMO effective address, software should program the trigger to match on NAPOT ranges, i.e.* `mcontrol6.match=1`*, and should program the NAPOT range to equal the cache block size.*

# 3. Formats

## 3.1. Instructions

For Zicbom and Zicboz:

```
inst[6:0]   - 0b0001111 (MISC-MEM)
inst[11:7]  - 0b00000 (rd - reserved)
inst[14:12] - 0b010 (new funct3 encoding)
inst[19:15] - rs1 (effective address)
inst[24:20] - 0b00000 (rs2 - reserved)
inst[31:25] - 0bxxxyyyy (funct7), where
   xxx:  0b000 (reserved for future modifiers)
   yyyy: 0b0000 - CBO.INVAL
         0b0001 - CBO.CLEAN
         0b0010 - CBO.FLUSH
         0b0100 - CBO.ZERO
         all others reserved
```

## 3.2. CSRs

***FIXME***: How is this extension disabled?

Four CSRs control execution of CMO instructions:

- `mcmocontrol`
- `scmocontrol`
- `hcmocontrol`
- `vscmocontrol`

> *The `scmocontrol` and `vscmocontrol` registers are both required to *distinguish CMO execution behavior when the effective privilege mode is U-mode *or VU-mode, respectively. These registers are only present if the H-extension *is implemented and enabled.*
>
> We need a separate vscmocontrol register to differentiate between the effective VU-mode behaviors and the effective U-mode behaviors in the scmocontrol when MPRV=1. So even though the hypervisor could swap out scmocontrol before returning to either VU/VS or U, M could arbitrarily perform effective VU or U accesses without letting the hypervisor know.

Each `xcmocontrol` register has the following generic format:

*Table 1. Generic Format for xcmocontrol CSRs*

| Bits | Name | Description |
|---|---|---|
| [0] | CBME | Cache Block Management instruction Enable<br><br>Determines the behavior of a cache block management instruction (i.e. CBO.INVAL, CBO.CLEAN, or CBO.FLUSH) when the instruction is executed in *privilege_mode*.<br><br>• 0: The instruction takes an illegal instruction exception<br>• 1: The instruction is executed |
| [1] | CBWE | Cache Block Write instruction Enable<br><br>Determines the behavior of a cache block write instruction (i.e. CBO.ZERO) when the instruction is executed in *privilege_mode*.<br><br>• 0: The instruction takes an illegal instruction exception<br>• 1: The instruction is executed |
| [7:2] | *Rsvd* | *Reserved* |
| [8] | INVW0I | CBO.INVAL access without write permission performs an Invalidate operation<br><br>Determines the operation performed by a CBO.INVAL instruction when the resulting access *has not been* granted write permission in the effective privilege mode (*Wx*=W0) and when the instruction does *not* raise an exception:<br><br>• 0: The instruction performs a flush operation<br>• 1: The instruction performs an invalidate operation |
| [9] | INVW0E | CBO.INVAL access without write permission Enable<br><br>Determines the behavior of a CBO.INVAL instruction when a *protection_mechanism* is enabled and the resulting access *has not been* granted write permission in the effective privilege mode (*Wx*=W0):<br><br>• 0: The instruction takes an exception (page fault, guest-page fault, or access fault depending on the CSR)<br>• 1: The instruction performs an operation based on INVW0I |

| Bits | Name | Description |
|---|---|---|
| [10] | INVW1I | CBO.INVAL access with write permission performs an Invalidate operation<br><br>Determines the operation performed by a CBO.INVAL instruction when the resulting access *has been* granted write permission in the effective privilege mode (*Wx*=W1) and when the instruction does *not* raise an exception:<br><br>• 0: The instruction performs a flush operation<br>• 1: The instruction performs an invalidate operation |
| [11] | INVW1E | CBO.INVAL access with write permission Enable<br><br>Determines the behavior of a CBO.INVAL instruction when a *protection_mechanism* is enabled and the resulting access *has been* granted write permission in the effective privilege mode (*Wx*=W1):<br><br>• 0: The instruction takes an exception (page fault, guest-page fault, or access fault depending on the CSR)<br>• 1: The instruction performs an operation based on INVW1I |
| [x:12] | *Rsvd* | *Reserved* |

Each xcmocontrol register is WARL, where CSR reads return the behaviors supported by the implementation.

The following subsections detail how the xcmocontrol CSRs govern the execution of CMO instructions.

#### Determining Illegal Instruction Exceptions

The descriptions for the CBME and CBWE bits in the xcmocontrol registers include a *privilege_mode* parameter that corresponds to the privilege modes controlled by a given CSR. Each CSR defines this parameter as follows:

- For mcmocontrol, *privilege_mode* corresponds to S/HS-mode, U-mode, VS-mode, and VU-mode
- For scmocontrol, *privilege_mode* corresponds to U-mode
- For hcmocontrol, *privilege_mode* corresponds to VS-mode and VU-mode
- For vscmocontrol, *privilege_mode* corresponds to VU-mode

Depending on the *current privilege mode*, a cache block management instruction takes an illegal instruction exception based on the CBME bits:

- M-mode:
  FALSE (cache block management instructions never take illegal instruction exceptions)
- S/HS-mode:

`!mcmocontrol.CBME`

- U-mode:
  `!mcmocontrol.CBME || !scmocontrol.CBME`
- VS-mode:
  `!mcmocontrol.CBME || !hcmocontrol.CBME`
- VU-mode:
  `!mcmocontrol.CBME || !hcmocontrol.CBME || !vscmocontrol.CBME`

Depending on the *current privilege mode*, a cache block write instruction takes an illegal instruction exception based on the `CBWE` bits:

- M-mode:
  `FALSE` (cache block write instructions never take illegal instruction exceptions)
- S/HS-mode:
  `!mcmocontrol.CBWE`
- U-mode:
  `!mcmocontrol.CBWE || !scmocontrol.CBWE`
- VS-mode:
  `!mcmocontrol.CBWE || !hcmocontrol.CBWE`
- VU-mode:
  `!mcmocontrol.CBWE || !hcmocontrol.CBWE || !vscmocontrol.CBWE`

Otherwise, the above instructions are executed in the *current privilege mode*.

#### Determining Page Fault, Guest-Page Fault, and Access Fault Exceptions

The descriptions for the `INVWxE` and `INVWxI` bits in the `xcmocontrol` registers include a *protection_mechanism* parameter that corresponds to the protection mechanism that determines write permission for an access and a *Wx* parameter that represents whether write permission has been granted (`W1`) or not (`W0`). Each CSR defines these as follows:

- For `mcmocontrol`, *protection_mechanism* corresponds to the PMP and PMAs and *Wx* corresponds to whether partial joint write permission has been granted by the PMP and PMAs
- For `scmocontrol`, *protection_mechanism* corresponds to the `satp` page table and *Wx* corresponds to whether write permission has been granted by the leaf PTE `W` bit
- For `hcmocontrol`, *protection_mechanism* corresponds to the `hgatp` page table and *Wx* corresponds to whether write permission has been granted by the leaf PTE `W` bit
- For `vscmocontrol`, *protection_mechanism* corresponds to the `vsatp` page table and *Wx* corresponds to whether write permission has been granted by the leaf PTE `W` bit

For each CSR, the resulting `INVWxE` value is determined by the designated *protection_mechanism*, which selects the `INVW0E` bit if *Wx*=`W0` or the `INVW1E` bit if *Wx*=`W1`. Depending on the *effective privilege mode*, a `CBO.INVAL` instruction takes the following types of traps based on the `INVWxE` values:

- M-mode:

    - *N/A* (`CBO.INVAL` never faults due to the CMO control registers)
- S/HS-mode:
    - Access fault:
      `!(mcmocontrol.INVWxE)`
- U-mode:
    - Page fault:
      `!(scmocontrol.INVWxE || satp.MODE==Bare)`
    - Access fault:
      `(scmocontrol.INVWxE || satp.MODE==Bare) &&`
      `!(mcmocontrol.INVWxE)`
- VS-mode:
    - Guest-page fault:
      `!(hcmocontrol.INVWxE || hgatp.MODE==Bare)`
    - Access fault:
      `(hcmocontrol.INVWxE || hgatp.MODE==Bare) &&`
      `!(mcmocontrol.INVWxE)`
- VU-mode:
    - Page fault:
      `!(vscmocontrol.INVWxE || vsatp.MODE==Bare)`
    - Guest-page fault:
      `(vscmocontrol.INVWxE || vsatp.MODE==Bare) &&`
      `!(hcmocontrol.INVWxE || hgatp.MODE==Bare)`
    - Access fault:
      `(vscmocontrol.INVWxE || vsatp.MODE==Bare) &&`
      `(hcmocontrol.INVWxE || hgatp.MODE==Bare) &&`
      `!(mcmocontrol.INVWxE)`

> *The above exception priorities reflect the architected exception priorities in the privileged architecture specification.*

For each CSR, the resulting `INVWxI` value is determined by the designated *protection_mechanism*, which selects the `INVW0I` bit if *Wx*=`W0` or the `INVW1I` bit if *Wx*=`W1`, if that protection mechanism is enabled. If the protection mechanism is disabled, the `INVWxI` value is the logical AND of the `INVW0I` bit and the `INVW1I` bit, i.e. both bits *must* be set to perform an invalidate operation. Assuming that no exception arises and depending on the *effective privilege mode*, a `CBO.INVAL` instruction performs the following operations based on the `INVWxI` values:

- M-mode:
    - Flush:
      `FALSE` (`CBO.INVAL` never performs a flush operation)
    - Invalidate:

`TRUE` (`CBO.INVAL` always performs an invalidate operation)

- S-mode:
  - Flush:
    `!(mcmocontrol.INVWxI)`
  - Invalidate:
    `(mcmocontrol.INVWxI)`
- U-mode:
  - Flush:
    `!(scmocontrol.INVWxI && mcmocontrol.INVWxI)`
  - Invalidate:
    `(scmocontrol.INVWxI && mcmocontrol.INVWxI)`
- VS-mode:
  - Flush:
    `!(hcmocontrol.INVWxI && mcmocontrol.INVWxI)`
  - Invalidate:
    `(hcmocontrol.INVWxI && mcmocontrol.INVWxI)`
- VU-mode:
  - Flush:
    `!(vscmocontrol.INVWxI && hcmocontrol.INVWxI && mcmocontrol.INVWxI)`
  - Invalidate:
    `(vscmocontrol.INVWxI && hcmocontrol.INVWxI && mcmocontrol.INVWxI)`

> *Until a modified cache block has updated memory, a `CBO.INVAL` instruction may expose stale data values in memory if the CSRs are programmed to perform an invalidate operation. This behavior may result in a security hole if lower privileged level software performs an invalidate operation and accesses sensitive information in memory. To avoid such holes, higher privileged level software must perform either a clean or flush operation on the cache block before permitting lower privileged level software to perform an invalidate operation on the block.*
>
> *Alternatively, higher privileged level software may program the CSRs so that `CBO.INVAL` either traps or performs a flush operation in a lower privileged level. The W0 and W1 bits allow higher privileged software finer-grained control of the behavior of `CBO.INVAL` in lower privilege levels based on whether write permission has been granted to that level by a particular protection mechanism.*

# 4. Instructions

## 4.1. Cache Block Management Instructions

Cache block management instructions operate on the cache blocks containing the effective address specified in *rs1*. These instructions also specify a *PoC* that, along with the coherence PMA,

determines the set of caches on which the operation is performed. In particular, the set of caches consists of one of the following:

- If the coherence PMA indicates that hardware enforces coherence on the physical address, all the caches accessed by the hart directly and indirectly in the coherence domains on the path from the hart to the *PoC*
- If the coherence PMA indicates that hardware does *not* enforce coherence on the physical address, only the caches accessed by the hart directly on the path from the hart to the *PoC*

### 4.1.1. `CBO.INVAL`

A `CBO.INVAL` instruction performs an *invalidate* operation or a *flush* operation, depending on the state of the CMO CSRs, on the set of caches determined by the *PoC* and the coherence PMA.

### 4.1.2. `CBO.CLEAN`

A `CBO.CLEAN` instruction performs a *clean* operation on the set of caches determined by the *PoC* and the coherence PMA.

### 4.1.3. `CBO.FLUSH`

A `CBO.FLUSH` instruction performs a *flush* operation on the set of caches determined by the *PoC* and the coherence PMA.

## 4.2. Cache Block Write Instruction

Cache block write instructions operate on the cache blocks containing the effective address specified in *rs1*. These instructions also specify a *level*, which is a hint to the hardware to allocate the cache block in a designated cache. *level* is specified as follows:

- `default` — an *implementation-defined* level, which may be a function of physical addresses, dynamic allocation policies, or any other characteristic
- `L1` — the first cache logically accessed by a hart on the path to memory
- `L2` — the second cache logically accessed by a hart on the path to memory
- `L3` — the third cache logically accessed by a hart on the path to memory

An implementation may ignore *level* and assume *level* is `default` for all cache block write instructions.

> *To a certain degree, level is implementation-defined for all systems; however, `L1`, `L2`, and `L3` are intended to communicate their common, informal meaning.*

### 4.2.1. `CBO.ZERO`

A `CBO.ZERO` instruction performs a series of byte writes whose data value equals zero to all the bytes in a cache block. An implementation may write any number of bytes in the cache block atomically.

The instruction may allocate, but is *not* guaranteed to allocate, the cache block in the cache specified by *level*.

## 4.3. Cache Block Hint Instructions

Cache block hint instructions operate on the cache blocks containing the effective address specified in *rs1*. These instructions also specify a *level*, which is a hint to the hardware to allocate the cache block in a designated cache. *level* is specified as follows:

- `default` — an *implementation-defined* level, which may be a function of physical addresses, dynamic allocation policies, or any other characteristic
- `L1` — the first cache logically accessed by a hart on the path to memory
- `L2` — the second cache logically accessed by a hart on the path to memory
- `L3` — the third cache logically accessed by a hart on the path to memory

An implementation may ignore *level* and assume *level* is `default` for all cache block hint instructions.

> *To a certain degree, level is implementation-defined for all systems; however, `L1`, `L2`, and `L3` are intended to communicate their common, informal meaning.*

### 4.3.1. PREFETCH.R

A `PREFETCH.R` instruction indicates to the cache at the specified *level* that a subsequent read operation is likely to be performed on the cache block at the specified effective address in the near future.

An implementation typically allocates the cache block in the cache at the specified *level* in a state that allows read access; however, the instruction is *not* guaranteed to allocate the cache block in that cache.

### 4.3.2. PREFETCH.W

A `PREFETCH.W` instruction indicates to the cache at the specified *level* that a subsequent write operation is likely to be performed on the cache block at the specified effective address in the near future.

An implementation typically allocates the cache block in the cache at the specified *level* in a state that allows write access; however, the instruction is *not* guaranteed to allocate the cache block in that cache.

A PREFETCH.W instruction may interfere with the eventual success guarantee of store-conditional instructions.

### 4.3.3. PREFETCH.I

A PREFETCH.I instruction indicates to the cache at the specified *level* that a subsequent instruction fetch operation is likely to be performed on the cache block at the specified effective address in the near future.

An implementation typically allocates the cache block in the cache at the specified *level* in a state that allows instruction fetch access; however, the instruction is *not* guaranteed to allocate the cache block in that cache.

Instruction fetch operations may access caches different from those accessed by read and write operations. It is *implementation-defined* whether the cache at the specified *level* in a PREFETCH.I instruction is the same cache at the specified *level* in a PREFETCH.R or PREFETCH.W instruction.

### 4.3.4. DEMOTE

A DEMOTE instruction indicates to the cache at the specified *level* that the cache block at the specified effective address is no longer required to be cached.

> *Typically, a DEMOTE instruction operates on the replacement algorithm information for a cache block rather than the cache block itself.*

## 4.4. FIXME: Cache Block Discovery Instruction

### 4.4.1. FIXME: CBO.SIZE

# 5. Opcode Map

---

**FIXME BEYOND HERE**

---

## Ignore Me

PoC Wording:

Paths converge at a *point of convergence*, or *PoC*, which designates the point at which a set of memory accesses to the same memory location is logically ordered. Once ordered by a PoC, a memory access in the set cannot be reordered with respect to other memory accesses in the set, and the set of memory accesses shares the remainder of the path to the memory location.

The fundamental PoC for a given physical address is the *point of convergence for memory*, or *PoC-memory*, which is the PoC where all paths for that physical address converge, independent of all other characteristics that define a path.

Memory Ordering Wording:

*The existing* `FENCE` *instruction is sufficient to order the cache management operations in the base extension. In future extensions, an additional fence instruction may be required to determine when certain cache management operations are complete.*

CSR stuff:

| [9:8] | `INVW0` | CBO.INVAL access without write permission (`Wx=0`).

Determines the behavior of a CBO.INVAL instruction when the corresponding access does *not* have write permission in the effective privilege mode:

- `0`: Execution results in an exception (page fault, guest-page fault, or access fault) [trap]
- `1`: *Reserved* (aliases to `0`)
- `2`: Execution performs a flush operation [flush]
- `3`: Execution performs an invalidate operation [invalidate]

| [11:10] | `INVW1` | CBO.INVAL access with write permission (`Wx=1`).

Determines the behavior of a CBO.INVAL instruction when the corresponding access has write permission in the effective privilege mode:

- `0`: Execution results in an exception (page fault, guest-page fault, or access fault depending on the CSR) [trap]
- `1`: *Reserved* (aliases to `0`)
- `2`: Execution performs a flush operation [flush]
- `3`: Execution performs an invalidate operation [invalidate]

| | |