[] | *risc-v_logo.png*

Random Document Title

Version 0.01, 06/2021: Pre-release version

Table of Contents

Preamble

This is the preamble, which can contain contributor information along with the Creative Commons Attribution statement below.

For this book example, I am here mentioning that I have copied information directly from several Web sites that are hosted by asciidoctor.org and devoted to providing Asciidoc/Asciidoctor documentation. Graphics used are either explicitly available for free, are property of RISC-V International, or were created using Wavedrom.

Chapter 1. Introduction (Ignore)

Cache management operations (*CMOs*) consist of operations that perform

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.

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.

Chapter 2. Background (Only Read This)

START READING HERE

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. An *observer* of a memory location is a hart or an I/O device that can load from or store to that memory location; a given observer may *not* be able to access all memory locations in the system.

> *Loads and stores are operations performed by an observer, while reads and writes are operations performed on a memory location. These operations may be decoupled by caches, described below. For example, a load may be serviced by a cache without performing a read of memory, or a write may be serviced by a cache by first performing a read of memory.*

When performing loads and stores, an observer may access any number of *caches* that may provide the requested data instead of the underlying memory locations. A cache buffers copies of data organized into *cache blocks*, which consist of naturally aligned power-of-two (*NAPOT*) sets of contiguous bytes. Each cache block is tagged with a physical address that identifies the corresponding memory locations. The organization of each cache and the size of a cache block are both *implementation-defined*.

A cache may be *private*, in which case the cache is accessed by a single observer, or *shared*, in which case cache is accessed by more than one observer. Multiple cache blocks for the same memory locations may be present in the system simultaneously, introducing multiple copies of data, and a system may implement mechanisms to keep the data in some or all of those copies coherent.

For a given memory location, a *set of coherent observers* consists of the set of observers for which all of the following are true:

- Stores from all observers in the set appear to be serialized with respect to each other
- Stores from all observers in the set eventually appear to all other observers in the set
- A load from an observer in the set returns the data values from a store from an observer in the set (or the initial data values in memory)

Coherent observers *must* access a given memory location with the same physical address and the same memory attributes, particularly coherence and cacheability.

> *Effectively, the loads and stores performed by the set of coherent observers are subject to one of the memory order models defined by the ISA.*

An observer who is a member of a set of coherent observers is said to be *coherent* with respect to the other observers in the set. On the other hand, an observer who is *not* a member is said to be *non-coherent* with respect to the observers in the set.

For a given cache block, the caches accessed by the coherent observers are kept coherent by an *implementation_defined* mechanism. Such a *coherent cache* may read the cache block at any time from another coherent cache or from the underlying memory locations. Similarly, a coherent cache may write the cache block at any time to another coherent cache. In addition, a coherent cache may write the cache block at any time to the underlying memory locations, provided that a coherent observer performed a store to the cache block since the previous such write. In this case, in the absence of an invalidate operation performed by a coherent observer, at least one coherent cache *must* write the cache block to the underlying memory locations; otherwise, no coherent cache may write the cache block to the underlying memory locations.

> *The above restrictions ensure that a "clean" copy cannot be written back into memory.*

Cache block management operations enable software running on a set of coherent observers to communicate with a set of non-coherent observers:

- An *invalidate operation* makes stores from a set of non-coherent observers appear to the set of coherent observers by removing all copies of a cache block from the coherent caches
- A *clean operation* makes stores from the set of coherent observers appear to a set of non-coherent observers by writing a copy of a cache block to the underlying memory locations (or to a cache shared by both sets), provided a coherent observer stored to the cache block since the previous such write
- A *flush operation* atomically performs a clean operation followed by an invalidate operation

FIXME: There is some question whether the two sub-operations in a flush operation must be atomic.

Cache block zero operations perform a series of store byte operations where the data are zero. An implementation may or may not update the entire cache block atomically.

Cache block prefetch operations are performance hints to the coherent caches to guide the placement of cache blocks. As hints, these operations may or may not cause cache blocks to be transferred to a particular cache.

2.1. Specifying Caches

Replacement for PoC:

- Point of
- Shared Access Point (SAP)
- Common Access Point (CAP)
- point of common access (PCA)
- Point of Shared Access

- Joint,

The set of coherent observers is a function of the physical address and the memory attributes of the access. As a result, the whatever point/level is determined by the same characteristics.

For example, the set of coherent observers for a non-coherent attribute is only the executing hart.

CMOs ignore cacheability so PCA must be explicit vs. implicit above.

STOP READING HERE

Chapter 3. Don’t read beyond here

3.1. System Topology (Stale)

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 (?)

3.1.1. 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.

3.2. FIXME: Memory Ordering

3.2.1. 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.

3.2.2. 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…?

Chapter 4. Traps

4.1. 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 Section 5.2 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.

4.2. 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 Section 5.2 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

4.2.1. 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.

4.3. 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 Section 5.2 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.

4.4. Address Misaligned Exception

CMO instructions do *not* generate address misaligned exceptions.

4.5. 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.*

Chapter 5. Formats

5.1. 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
```

5.2. 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 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*. • 0: The instruction takes an illegal instruction exception • 1: The instruction is executed
[1]	CBWE	Cache Block Write instruction Enable Determines the behavior of a cache block write instruction (i.e. CBO.ZERO) when the instruction is executed in *privilege_mode*. • 0: The instruction takes an illegal instruction exception • 1: The instruction is executed
[7:2]	*Rsvd*	*Reserved*
[8]	INVW0I	CBO.INVAL access without write permission performs an Invalidate operation 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: • 0: The instruction performs a flush operation • 1: The instruction performs an invalidate operation
[9]	INVW0E	CBO.INVAL access without write permission Enable 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): • 0: The instruction takes an exception (page fault, guest-page fault, or access fault depending on the CSR) • 1: The instruction performs an operation based on INVW0I

Bits	Name	Description
[10]	INVW1I	CBO.INVAL access with write permission performs an Invalidate operation 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: • 0: The instruction performs a flush operation • 1: The instruction performs an invalidate operation
[11]	INVW1E	CBO.INVAL access with write permission Enable 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): • 0: The instruction takes an exception (page fault, guest-page fault, or access fault depending on the CSR) • 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.

5.2.1. Determining Traps

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*.

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.*

Chapter 6. Instructions

6.1. 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*

6.1.1. `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.

6.1.2. `CBO.CLEAN`

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

6.1.3. `CBO.FLUSH`

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

6.2. 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.

6.2.1. `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*.

6.3. 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.

6.3.1. `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.

6.3.2. `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.

6.3.3. `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.