RISC-V Base Cache Management Operation ISA Extensions

Version 0.5-92cebdd, 2021-08-11: Development (subject to change) [Arch Review Candidate 2]

Table of Contents

Colophon

Acknowledgments

Contributors to this specification (in alphabetical order) include:
John Ingalls, David Kruckemyer, and Philipp Tomsich.

We express our gratitude to everyone that contributed to, reviewed, or improved this specification through their comments and questions.

Pseudocode for instruction semantics

The semantics of each instruction in the Instructions chapter is expressed in a SAIL-like syntax.

Chapter 1. Introduction

Cache management operation (or *CMO*) instructions perform operations on cached copies of data in the memory hierarchy. These instructions are grouped by operation into the following instruction classes:

- A *management* instruction manipulates cached copies of data with respect to sets of observers
- A *zero* instruction zeros out a range of memory locations, potentially allocating cached copies of data in one or more caches
- A *prefetch* instruction indicates to hardware that data at a given memory location may be accessed in the near future, potentially allocating cached copies of data in one or more caches

This document introduces a base set of CMO ISA extensions that operate specifically on cache blocks. Each of the above classes of instructions represents an extension in this specification:

- The *Zicbom* extension defines a set of cache block management instructions: `CBO.INVAL`, `CBO.CLEAN`, and `CBO.FLUSH`
- The *Zicboz* extension defines a cache block zero instruction: `CBO.ZERO`
- The *Zicbop* extension defines a set of cache block prefetch instructions: `PREFETCH.R`, `PREFETCH.W`, and `PREFETCH.I`

The execution behavior of the above instructions is also modified by CSR state added by this specification.

The remainder of this document provides general background information on CMO instructions and describes each of the above ISA extensions.

> *The term CMO encompasses all operations on caches or resources related to caches. The term CBO represents a subset of CMOs that operate only on cache blocks. The first CMO extensions only define CBOs.*

Chapter 2. Background

This chapter provides information common to all CMO extensions.

2.1. Memory and Caches

A *memory location* is a physical resource in a system uniquely identified by a *physical address*. The *observers* of a given memory location consist of the RISC-V harts or I/O devices that can access that memory location. A given observer may not be able to access all memory locations in a system, and two different harts or devices may or may not be able to observe the same set of memory locations.

In this specification, a *load operation* (or *store operation*) is performed by an observer to consume (or modify) the data at a given memory location. For a RISC-V hart, a load or store operation may be performed as a result of an explicit or implicit memory access. Additionally, a *read operation* (or *write operation*) is an operation performed on the memory location to fetch (or update) the data at the physical resource.

> *Load and store operations are decoupled from read and write operations by caches, described below. For example, a load operation may be satisfied by a cache without performing a read operation in memory, or a store operation may be satisfied by a cache that first performs a read operation.*

A *cache* is a structure that buffers copies of data to reduce average memory latency. Any number of caches may be interspersed between an observer and a memory location, and load and store operations from an observer may be satisfied by a cache instead of the memory location.

Caches organize copies of data into *cache blocks*, each of which represents a contiguous, naturally aligned power-of-two (or *NAPOT*) range of memory locations. A cache block is identified by a physical address corresponding to the underlying memory locations, and a *cache block operation* (or *CBO*) operates on one or more cache blocks.

> *The Zicbom, Zicboz, and Zicbop extensions define operations on a single cache block only.*

Like an operation on a memory location, a read operation may be performed on a cache to fetch a copy of a cache block, and a write operation may be performed on a cache to update a copy of a cache block. In effect, read and write operations transfer copies of cache blocks among caches and memory.

The capacity and organization of a cache and the size of a cache block are both *implementation-defined*, and the execution environment provides software a means to discover information about the caches and cache blocks in a system. For the initial base set of CBOs, the size of a cache block shall be uniform throughout the system.

> *In future CMO extensions, the requirement for a uniform cache block size may be relaxed.*

2.2. Coherent Observers

For a given memory location, a *set of coherent observers* consists of the set of observers for which all of the following hold without software intervention:

- Store operations from all observers in the set appear to be serialized with respect to each other

- Store operations from all observers in the set eventually appear to all other observers in the set
- A load operation from an observer in the set returns data from a store operation from an observer in the set (or from the initial data in memory)

The coherent observers within such a set shall access a given memory location with the same physical address and the same physical memory attributes; however, if the coherence PMA for a given observer indicates a given memory location is not coherent, that observer shall not be a member of a set of coherent observers with any other observer for that memory location.

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.

Caches introduce multiple copies of a given cache block, and the copies accessed by the coherent observers are kept coherent by an *implementation-defined* mechanism. A *coherent cache* may allocate a copy of the cache block at any time, obtaining a copy from another coherent cache or from the underlying memory locations by performing a read operation. Similarly, a coherent cache may deallocate a copy of the cache block at any time, transferring a copy to another coherent cache by performing a write operation. Additionally, a coherent cache may transfer a copy of the cache block to the underlying memory locations at any time by performing a write operation, provided that a coherent observer performed a store operation to the cache block since the previous write operation to the memory locations. In the absence of an invalidate operation performed by a coherent observer (see Cache Block Management Instructions), at least one coherent cache shall write the cache block to the underlying memory locations if a coherent observer performed a store operation to the cache block; otherwise, no coherent cache may perform a write operation of the cache block to the underlying memory locations.

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

2.3. Memory Ordering

2.3.1. Preserved Program Order

The preserved program order (abbreviated *PPO*) 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 in program order never appears in the global memory order before a preceding invalidate, clean, or flush operation to an overlapping address.*

Additionally, invalidate, clean, and flush operations are classified as W or O (depending on the physical memory attributes for the corresponding physical addresses) for the purposes of predecessor and successor sets in `FENCE` instructions.

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

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

2.3.2. Load Values

An invalidate operation may change the set of values that can be returned by a load. In particular, an additional condition is added to the Load Value Axiom:

- If an invalidate operation i precedes a load r and operates on a byte x returned by r, and no store to x appears between i and r in program order or in the global memory order, then r returns any of the following values for x:
  1. If no clean or flush operations on x precede i in the global memory order, either the initial value of x or the value of any store to x that precedes i
  2. If no store to x precedes a clean or flush operation on x in the global memory order and if the clean or flush operation on x precedes i in the global memory order, either the initial value of x or the value of any store to x that precedes i
  3. If a store to x precedes a clean or flush operation on x in the global memory order and if the clean or flush operation on x precedes i in the global memory order, either the value of the latest store to x that precedes the latest clean or flush operation on x or the value of any store to x that both precedes i and succeeds the latest clean or flush operation on x that precedes i
  4. The value of any store to x by a non-coherent observer regardless of the above conditions

> *The first three bullets describe the possible load values at different points in the global memory order relative to clean or flush operations. The final bullet implies that the load value may be produced by a non-coherent observer at any time.*

2.4. Instruction Execution and Traps

Similar to load and store instructions, CMO instructions are memory access instructions that compute an effective address. The effective address is ultimately translated into a physical address based on the privilege mode and enabled translation mechanisms.

Execution of certain CMO instructions may result in traps due to CSR state, described in the Control and Status Register State section, or due to the various memory translation and protection mechanisms. The trapping behavior of CMO instructions is described in the following sections.

2.4.1. Illegal Instruction and Virtual Instruction Exceptions

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

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

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

2.4.2. 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.

A cache block management instruction requires read (R=1), write (W=1), or execute (X=1) permission (given a legal PTE encoding for the XWR bits, as specified by the privileged architecture) and, if applicable, user access (U=1) in the effective privilege mode; otherwise, the instruction takes a store page fault exception.

A cache block zero instruction requires write (W=1) permission (given a legal PTE encoding for the XWR bits, as specified by the privileged architecture) and, if applicable, user access (U=1) in the effective privilege mode; otherwise, the instruction takes a store page fault exception.

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

A cache block prefetch instruction requires read (R=1), write (W=1), or execute (X=1) permission (given a legal PTE encoding for the XWR bits, as specified by the privileged architecture) and, if applicable, user access (U=1) in the effective privilege mode. In addition, an implementation may require any of the following to perform a memory access:

- PREFETCH.R may require read (R=1) permission
- PREFETCH.W may require write (W=1) permission
- PREFETCH.I may require execute (X=1) permission

If the required permission is *not* granted, however, the instruction does *not* take a page fault or guest-page fault exception and retires without performing a memory access.

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.

2.4.3. 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 shall be the same for *all* physical addresses in the cache block and read permission shall be granted if write

permission has been granted. If these PMA constraints are *not* met, the behavior of a CMO instruction is UNSPECIFIED.

For the purposes of access fault determination, the following terms are defined for a given physical address:

- *joint read permission* is granted when both the PMP and PMAs allow read access to the physical address
- *joint write permission* is granted when both the PMP and PMAs allow write access to the physical address
- *joint execute permission* is granted when both the PMP and PMAs allow execute access to the physical address

A cache block management instruction requires joint read, joint write, or joint execute permission (given legal PMA and PMP encodings for the `XWR` bits, as specified by the privileged architecture) for each physical address in a cache block; otherwise, the instruction takes a store access fault exception.

A cache block zero instruction requires joint write permission (given legal PMA and PMP encodings for the `XWR` bits, as specified by the privileged architecture) for each physical address in a cache block; otherwise, the instruction takes a store access fault exception.

A cache block prefetch instruction requires joint read, joint write, or joint execute permission (given legal PMA and PMP encodings for the `XWR` bits, as specified by the privileged architecture) for each physical address in a cache block. In addition, an implementation may require any of the following to perform a memory access:

- `PREFETCH.R` may require joint read permission
- `PREFETCH.W` may require joint write permission
- `PREFETCH.I` may require joint execute permission

If the required permission is *not* granted, however, the instruction does *not* take an access fault exception and retires without performing a memory access.

2.4.4. Address Misaligned Exception

CMO instructions do *not* generate address misaligned exceptions.

2.4.5. Breakpoint Exception and Debug Mode Entry

Unless otherwise defined by the debug architecture specification, the behavior of trigger modules with respect to CMO instructions is UNSPECIFIED.

For the Zicbom, Zicboz, and Zicbop extensions, this specification recommends the following common trigger module behaviors:

- Type 6 address match triggers, i.e. `tdata1.type=6` and `mcontrol6.select=0`, should be supported
- Type 2 address/data match triggers, i.e. `tdata1.type=2`, should be unsupported
- The size of a memory access equals the size of the cache block accessed, and the compare values follow from the addresses of the NAPOT memory region corresponding to the cache block containing the effective address
- Unless an encoding for a cache block is added to the `mcontrol6.size` field, an address trigger should only match a memory access from a CBO instruction if `mcontrol6.size=0`

If the Zicbom extension is implemented, this specification recommends the following additional trigger module behaviors:

- Implementing address match triggers should be optional
- Type 6 data match triggers, i.e. `tdata1.type=6` and `mcontrol6.select=1`, should be unsupported
- Memory accesses are considered to be stores, i.e. an address trigger matches only if `mcontrol6.store=1`

If the Zicboz extension is implemented, this specification recommends the following additional trigger module behaviors:

- Implementing address match triggers should be mandatory
- Type 6 data match triggers, i.e. `tdata1.type=6` and `mcontrol6.select=1`, should be supported, and implementing these triggers should be optional
- Memory accesses are considered to be stores, i.e. an address trigger matches only if `mcontrol6.store=1`

If the Zicbop extension is implemented, this specification recommends the following additional trigger module behaviors:

- Implementing address match triggers should be optional
- Type 6 data match triggers, i.e. `tdata1.type=6` and `mcontrol6.select=1`, should be unsupported
- Memory accesses may be considered to be loads or stores depending on the implementation, i.e. whether an address trigger matches on these instructions when `mcontrol6.load=1` or `mcontrol6.store=1` is *implementation-defined*

This specification also recommends that the behavior of trigger modules with respect to the Zicboz extension should be defined in version 1.0 of the debug architecture specification. The behavior of trigger modules with respect to the Zicbom and Zicbop extensions is expected to be defined in future extensions.

2.5. Effects on Constrained LR/SC Loops

Executing any cache block management instruction (`CBO.INVAL`, `CBO.CLEAN`, or `CBO.FLUSH`) or a cache block zero instruction (`CBO.ZERO`) may cause a reservation on another hart to be lost. As a result, executing one of these instructions constitutes an additional event (similar to executing an unconditional store or an AMO instruction) that satisfies the eventuality guarantee of constrained LR/SC loops defined the A extension.

Executing any cache block prefetch instruction (`PREFETCH.I`, `PREFETCH.R`, or `PREFETCH.W`) does not impact the eventuality guarantee of constrained LR/SC loops defined by the A extension; however, these instructions may cause the periodic cancellation of a reservation on another hart.

2.6. Discovery

TBD

- general cache structure/organization?
- relationship among harts?
- cache block size for management, zero, prefetch
- CBIE support at each privilege level == Control and Status Register State

The CMO extensions rely on state in envcfg CSRs that will be defined in a future extension. If this CSR extension is not ratified, the CMO extension will define its own CSRs.

Three CSRs control the execution of CMO instructions:

- `menvcfg`
- `senvcfg`
- `henvcfg`

The `senvcfg` register is used by all supervisor modes, including VS-mode. A hypervisor is responsible for saving and restoring `senvcfg` on guest context switches. The `henvcfg` register is only present if the H-extension is implemented and enabled.

Each `xenvcfg` register (where `x` is `m`, `s`, or `h`) has the following generic format:

Table 1. Generic Format for xenvcfg CSRs

Bits	Name	Description
[5:4]	`CBIE`	Cache Block Invalidate instruction Enable Enables the execution of the cache block invalidate instruction, `CBO.INVAL`, in a lower privilege mode: • `00`: The instruction takes an illegal instruction exception • `01`: The instruction is executed and performs a flush operation • `10`: *Reserved* (implementations are expected, but not required, to treat this value as `00`; however, software must not rely on this behavior) • `11`: The instruction is executed and performs an invalidate operation

Bits	Name	Description
[6]	CBCFE	Cache Block Clean and Flush instruction Enable Enables the execution of the cache block clean instruction, CBO.CLEAN, and the cache block flush instruction, CBO.FLUSH, in a lower privilege mode: • 0: The instruction takes an illegal instruction exception • 1: The instruction is executed
[7]	CBZE	Cache Block Zero instruction Enable Enables the execution of the cache block zero instruction, CBO.ZERO, in a lower privilege mode: • 0: The instruction takes an illegal instruction exception • 1: The instruction is executed

The xenvcfg registers control CMO instruction execution based on the *current privilege mode* and the state of the appropriate CSRs, as detailed below.

A cache block invalidate instruction executes or takes an illegal instruction exception based on the state of the xenvcfg.CBIE fields:

```
// this pseudocode assumes the expected implementation of the reserved encoding
if (((curr_priv_mode == S/HS) && menvcfg.CBIE[0]) ||
    ((curr_priv_mode == U)    && menvcfg.CBIE[0] && senvcfg.CBIE[0]) ||
    ((curr_priv_mode == VS)   && menvcfg.CBIE[0] && henvcfg.CBIE[0]) ||
    ((curr_priv_mode == VU)   &&
     menvcfg.CBIE[0] && senvcfg.CBIE[0] && henvcfg.CBIE[0]))
{
  if (((curr_priv_mode == S/HS) && menvcfg.CBIE[1]) ||
      ((curr_priv_mode == U)    && menvcfg.CBIE[1] && senvcfg.CBIE[1]) ||
      ((curr_priv_mode == VS)   && menvcfg.CBIE[1] && henvcfg.CBIE[1]) ||
      ((curr_priv_mode == VU)   &&
       menvcfg.CBIE[1] && senvcfg.CBIE[1] && henvcfg.CBIE[1]))
  {
    <execute CBO.INVAL and perform an invalidate operation>
  } else {
    <execute CBO.INVAL and perform a flush operation
  }
}
else
{
  <illegal instruction trap>
}
```

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.

A cache block clean or flush instruction executes or takes an illegal instruction exception based on the state of the xenvcfg.CBCFE bits:

```
if (((curr_priv_mode == S/HS) && menvcfg.CBCFE) ||
    ((curr_priv_mode == U)    && menvcfg.CBCFE && senvcfg.CBCFE) ||
    ((curr_priv_mode == VS)   && menvcfg.CBCFE && henvcfg.CBCFE) ||
    ((curr_priv_mode == VU)   &&
     menvcfg.CBCFE && senvcfg.CBCFE && henvcfg.CBCFE))
{
  <execute CBO.CLEAN or CBO.FLUSH>
}
else
{
  <illegal instruction trap>
}
```

Finally, a cache block zero instruction executes or takes an illegal instruction exception based on the state of the xenvcfg.CBZE bits:

```
if (((curr_priv_mode == S/HS) && menvcfg.CBZE) ||
    ((curr_priv_mode == U)    && menvcfg.CBZE && senvcfg.CBZE) ||
    ((curr_priv_mode == VS)   && menvcfg.CBZE && henvcfg.CBZE) ||
    ((curr_priv_mode == VU)   &&
     menvcfg.CBZE && senvcfg.CBZE && henvcfg.CBZE))
{
  <execute CBO.ZERO>
}
else
{
  <illegal instruction trap>
}
```

Each xenvcfg register is WLRL, where the legal values are defined by the execution environment discovery mechanism.

This specification suggests, but does not require, that implementations capture only legal values in the CSRs.

Chapter 3. Extensions

CMO instructions are defined in the following extensions:

- Cache Block Management Instructions
- Cache Block Zero Instructions
- Cache Block Prefetch Instructions

Cache block management instructions and cache block zero instructions operate on the cache block containing the effective address equal to the base address specified in *rs1*. Cache block prefetch instructions operate on the cache block containing the effective address equal to the sum of the base address specified in *rs1* and a sign-extended offset encoded in the *imm* field, where offset[4:0] shall equal 0b00000. The effective address is translated into a corresponding physical address by the translation mechanisms appropriate in the effective privilege level.

3.1. Cache Block Management Instructions

Cache block management instructions enable software running on a set of coherent observers to communicate with a set of non-coherent observers by performing one of the following operations:

- An *invalidate operation* makes store operations from a set of non-coherent observers appear to the set of coherent observers at a point common to both sets by removing all copies of a cache block from the coherent caches up to that point
- A *clean operation* makes store operations from the set of coherent observers appear to a set of non-coherent observers at a point common to both sets by performing a write operation of a copy of a cache block to that point, provided a coherent observer performed a store operation to the cache block since the previous such write operation
- A *flush operation* atomically performs a clean operation followed by an invalidate operation

In the Zicbom extension, the instructions operate to a point common to *all* observers in the system. In other words, an invalidate operation ensures that store operations from all non-coherent observers appear to observers in the set of coherent observers, and a clean operation ensures that store operations from coherent observers appear to all non-coherent observers.

> *The Zicbom extension does not prohibit observers that fall outside of the above architectural definition; however, software cannot rely on the defined cache operations to have the desired effects with respect to those observers.*
>
> *Future extensions may define different sets of observers for the purposes of performance optimization.*

The following instructions comprise the Zicbom extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	cbo.clean *base*	Cache Block Clean
✓	✓	cbo.flush *base*	Cache Block Flush
✓	✓	cbo.inval *base*	Cache Block Invalidate

3.2. Cache Block Zero Instructions

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

The following instructions comprise the Zicboz extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	cbo.zero *base*	Cache Block Zero

3.3. Cache Block Prefetch Instructions

Cache block prefetch instructions are hints to the hardware to indicate that software intends to perform a particular type of memory access in the near future. The types of memory accesses are instruction fetch, data read (i.e. load), and data write (i.e. store).

An implementation is not required to perform any memory accesses in response to a cache block prefetch instruction.

The following instructions comprise the Zicbop extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	prefetch.i *base*, *offset*	Cache Block Prefetch for Instruction Fetch
✓	✓	prefetch.r *base*, *offset*	Cache Block Prefetch for Data Read
✓	✓	prefetch.w *base*, *offset*	Cache Block Prefetch for Data Write

Chapter 4. Instructions

4.1. cbo.clean

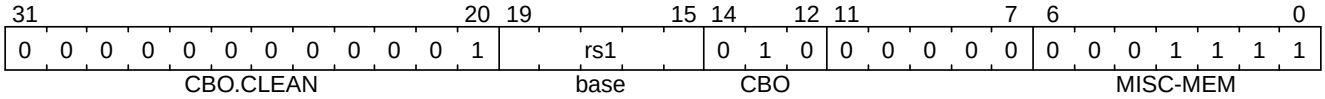
Synopsis

Perform a clean operation on the cache block containing the effective address

Mnemonic

cbo.clean *base*

Encoding



Description

A **cbo.clean** instruction performs a clean operation on the set of coherent caches accessed by the observer executing the instruction.

Operation

```
TODO
```

4.2. cbo.inval

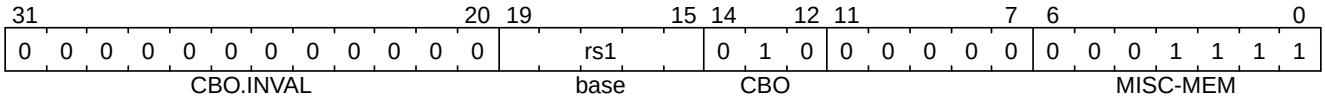
Synopsis

Perform an invalidate operation on the cache block containing the effective address

Mnemonic

cbo.inval *base*

Encoding



Description

A **cbo.inval** instruction performs an invalidate operation on the set of coherent caches accessed by the observer executing the instruction. The instruction may perform a flush operation, however, depending on CSR programming.

Operation

```
TODO
```

4.3. cbo.flush

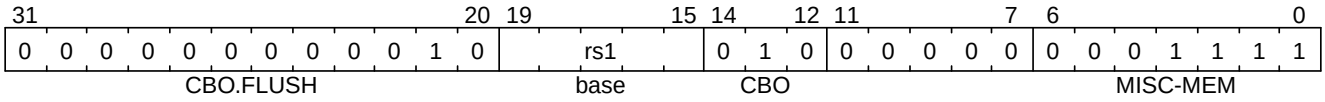
Synopsis

Perform a flush operation on the cache block containing the effective address

Mnemonic

cbo.flush *base*

Encoding



Description

A **cbo.flush** instruction performs a flush operation on the set of coherent caches accessed by the observer executing the instruction.

Operation

```
TODO
```

4.4. cbo.zero

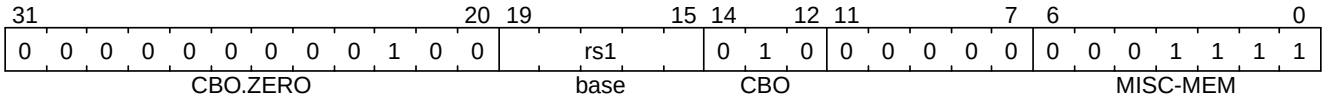
Synopsis

Write zeros to the cache block containing the effective address

Mnemonic

cbo.zero *base*

Encoding



Description

A **cbo.zero** instruction performs a series of byte-sized store operations where the data equal zero. An implementation may or may not update the entire cache block atomically and may or may not allocate a copy of the cache block in one of the coherent caches accessed by the observer executing the instruction.

Operation

```
TODO
```

4.5. prefetch.i

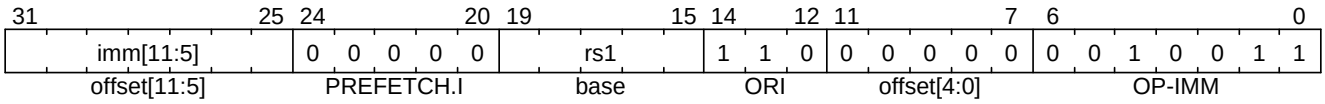
Synopsis

Provide a hint to hardware that the cache block containing the effective address is likely to be accessed by an instruction fetch in the near future

Mnemonic

prefetch.i *base*, *offset*

Encoding



Description

A **prefetch.i** instruction indicates to hardware that the cache block containing the effective address is likely to be accessed by an instruction fetch in the near future. An implementation may opt to cache a copy of the cache block in a cache accessed by an instruction fetch in order to improve memory access latency, but this behavior is *not* required.

Operation

```
TODO
```

4.6. prefetch.r

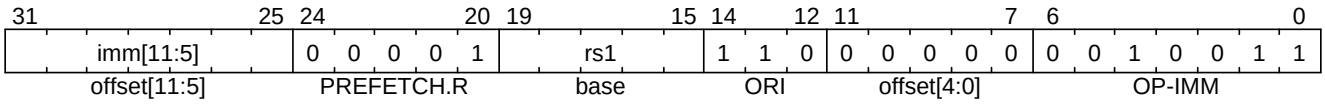
Synopsis

Provide a hint to hardware that the cache block containing the effective address is likely to be accessed by a data read in the near future

Mnemonic

prefetch.r *base*, *offset*

Encoding



Description

A **prefetch.r** instruction indicates to hardware that the cache block containing the effective address is likely to be accessed by a data read (i.e. load) in the near future. An implementation may opt to cache a copy of the cache block in a cache accessed by a data read in order to improve memory access latency, but this behavior is *not* required.

Operation

```
TODO
```

4.7. prefetch.w

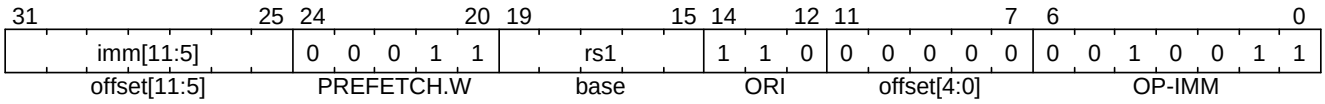
Synopsis

Provide a hint to hardware that the cache block containing the effective address is likely to be accessed by a data write in the near future

Mnemonic

prefetch.w *base*, *offset*

Encoding



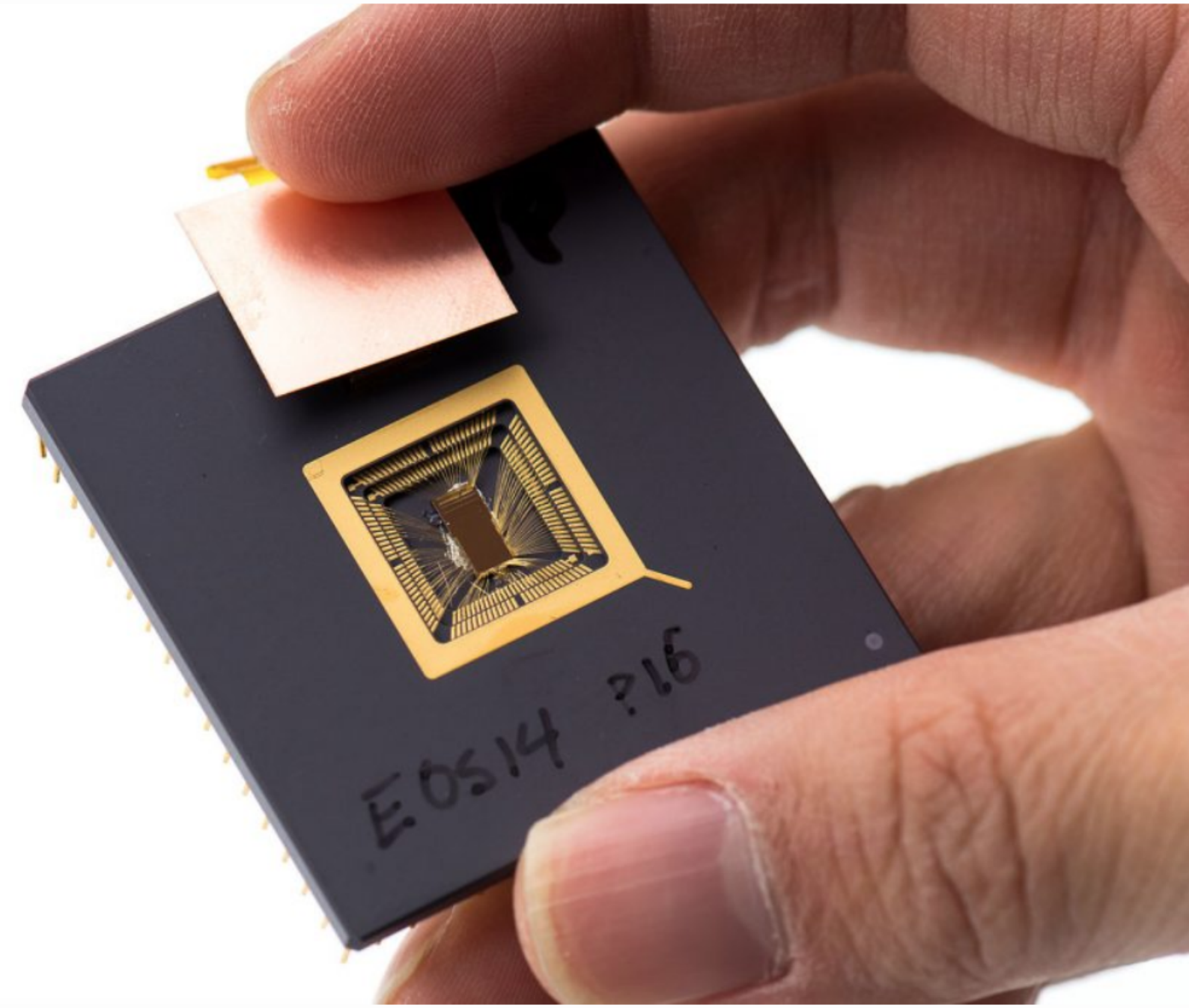
Description

A **prefetch.w** instruction indicates to hardware that the cache block containing the effective address is likely to be accessed by a data write (i.e. store) in the near future. An implementation may opt to cache a copy of the cache block in a cache accessed by a data write in order to improve memory access latency, but this behavior is *not* required.

Operation

```
TODO
```

Appendix A: Software guide


EOS14 ?16