RISC-V Base Cache Management Operation ISA Extensions

Version 0.6-43a42c6, 2021-09-15: Development (subject to change)

Table of Contents

Colophon

Acknowledgments

Contributors to this specification (in alphabetical order) include:
Allen Baum, Paul Donahue, Greg Favor, Andy Glew, John Ingalls, David Kruckemyer, Josh Scheid, Philipp Tomsich, Paul Walmsley, and Derek Williams

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 copies of data in the memory hierarchy. In general, CMO instructions operate on cached copies of data, but in some cases, a CMO instruction may operate on memory locations directly. Furthermore, CMO instructions are grouped by operation into the following classes:

- A *management* instruction manipulates cached copies of data with respect to a set of agents that can access the data
- 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 or the memory locations corresponding to a cache block; these are known as *cache-block operation* (or *CBO*) instructions. 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*. An *agent* is a logic block, such as a RISC-V hart, accelerator, I/O device, etc., that can access a given memory location.

> *A given agent may not be able to access all memory locations in a system, and two different agents may or may not be able to access the same set of memory locations.*

A *load operation* (or *store operation*) is performed by an agent to consume (or modify) the data at a given memory location. Load and store operations are performed as a result of explicit memory accesses to that memory location. Additionally, a *read transfer* from memory fetches the data at the memory location, while a *write transfer* to memory updates the data at the memory location.

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

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

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. The capacity and organization of a cache and the size of a cache block are both *implementation-specific*, and the execution environment provides software a means to discover information about the caches and cache blocks in a system. In the initial set of CMO extensions, 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.*

Implementation techniques such as speculative execution or hardware prefetching may cause a given cache to allocate or deallocate a copy of a cache block at any time, provided the corresponding physical addresses are accessible according to the supported access type PMA and are cacheable according to the cacheability PMA. Allocating a copy of a cache block results in a read transfer from another cache or from memory, while deallocating a copy of a cache block may result in a write transfer to another cache or to memory depending on whether the data in the copy were modified by a store operation. Additional details are discussed in Coherent Agents and Caches.

2.2. Cache-Block Operations

A CBO instruction causes one or more operations to be performed on the cache blocks identified by the instruction. In general, a CBO instruction may identify one or more cache blocks; however, in the initial set of CMO extensions, CBO instructions identify a single cache block only.

A cache-block management instruction performs one of the following operations, relative to the copy of a given cache block allocated in a given cache:

- An *invalidate operation* deallocates the copy of the cache block
- A *clean operation* performs a write transfer to another cache or to memory if the data in the copy of the cache block have been modified by a store operation
- A *flush operation* atomically performs a clean operation followed by an invalidate operation

Additional details, including the actual operation performed by a given cache-block management instruction, are described in Cache-Block Management Instructions.

A cache-block zero instruction performs a set of store operations that write zeros to the set of bytes corresponding to a cache block. An implementation may or may not update the entire set of bytes atomically with a single store operation. Additional details are described in Cache-Block Zero Instructions.

A cache-block prefetch instruction is a HINT to the hardware that software expects to perform a particular type of memory access in the near future. Additional details are described in Cache-Block Prefetch Instructions.

2.3. Coherent Agents and Caches

For a given memory location, a *set of coherent agents* consists of the agents for which all of the following hold:

- Store operations from all agents in the set appear to be serialized with respect to each other
- Store operations from all agents in the set eventually appear to all other agents in the set
- A load operation from an agent in the set returns data from a store operation from an agent in the set (or from the initial data in memory)

The coherent agents 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 agent indicates a given memory location is not coherent, that agent shall not be a member of a set of coherent agents with any other agent for that memory location and shall be the sole member of a set of coherent agents consisting of itself.

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

Caches introduce the possibility that multiple copies of a given cache block may be present in a system at the same time. An *implementation-specific* mechanism keeps these copies coherent with respect to the load and store operations from the agents in the set of coherent agents. Additionally, if a coherent agent in the set executes a CBO instruction that specifies the cache block, the resulting operation shall apply to any and all of the copies in the caches that can be accessed by the load and store operations from the coherent agents.

> *An operation from a CBO instruction is defined to operate only on the copies of a cache block that are cached in the caches accessible by the explicit memory accesses performed by the set of coherent agents. This includes copies of a cache block in caches that are accessed only indirectly by load and store operations, e.g. coherent instruction caches.*

The set of caches subject to the above mechanism form a *set of coherent caches*, and each coherent cache has the following behaviors, assuming all operations are performed by the agents in a set of coherent agents:

- A coherent cache is permitted to allocate and deallocate copies of a cache block and perform read and write

transfers as described in Memory and Caches

- A coherent cache is permitted to perform a write transfer to memory provided that a store operation has modified the data in the cache block since the most recent invalidate, clean, or flush operation on the cache block
- At least one coherent cache is responsible for performing a write transfer to memory once a store operation has modified the data in the cache block until the next invalidate, clean, or flush operation on the cache block, after which no coherent cache is responsible (or permitted) to perform a write transfer to memory until the next store operation has modified the data in the cache block
- A coherent cache is required to perform a write transfer to memory if a store operation has modified the data in the cache block since the most recent invalidate, clean, or flush operation on the cache block and if the next clean or flush operation requires a write transfer to memory

> *The above restrictions ensure that a "clean" copy of a cache block, fetched by a read transfer from memory and unmodified by a store operation, cannot later overwrite the copy of the cache block in memory updated by a write transfer to memory from a non-coherent agent.*

A non-coherent agent may initiate a cache-block operation that operates on the set of coherent caches accessed by a set of coherent agents. The mechanism to perform such an operation is *implementation-specific*.

2.4. Memory Ordering

2.4.1. 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, for cache-block prefetch instructions, the resulting operations are *not* ordered by the PPO rules.

2.4.2. 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 agent 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 agent at any time.

2.5. 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 the 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 address translation and protection mechanisms. The trapping behavior of CMO instructions is described in the following sections.

2.5.1. Illegal Instruction and Virtual Instruction Exceptions

Cache-block management instructions and cache-block zero instructions may raise illegal instruction exceptions or virtual instruction exceptions depending on the current privilege mode and the state of the CMO control registers described in the Control and Status Register State section.

Cache-block prefetch instructions raise neither illegal instruction exceptions nor virtual instruction exceptions.

2.5.2. Page Fault and Guest-Page Fault Exceptions

A cache-block management instruction is permitted to access the specified cache block if the translation table permits instruction fetches or load instructions to access the cache block in the same privilege mode; otherwise, the instruction raises a store page fault exception or a store guest-page fault exception.

A cache-block zero instruction is permitted to access the specified cache block if the translation table permits store instructions to access the cache block in the same privilege mode; otherwise, the instruction raises a store page fault exception or a store guest-page fault exception.

A cache-block prefetch instruction is permitted to access the specified cache block if the translation table permits instruction fetches or load instructions to access the cache block in the same privilege mode; otherwise, the instruction is not permitted to access any caches or memory and does not raise any exceptions.

As with load and store instructions, the privilege mode of a CMO instruction is modified by the MPRV, MPV, and MPP bits in mstatus and the SUM bit in either mstatus/sstatus or vsstatus. The MXR bit in either mstatus/sstatus or vsstatus does not affect the privilege mode of a CMO instruction.

This specification expects that implementations will process cache-block management instructions like store/AMO instructions, so store/AMO exceptions are appropriate for these instructions, regardless of the permissions required.

Note that the permission to execute a store instruction implies the permission to execute a load instruction, so store instructions are omitted from the list of permissions required to execute cache-block management and prefetch instructions.

2.5.3. Access Fault Exceptions

The CMO extensions impose the following constraints on the physical addresses in a given cache block:

- The PMAs shall be the same for *all* physical addresses in the cache block, and if write permission is granted by the supported access type PMAs, read permission shall also be granted
- The PMP access control bits shall be the same for *all* physical addresses in the cache block, and if write permission is granted by the PMP access control bits, read permission shall also be granted

If the above constraints are not met, the behavior of a CBO instruction is UNSPECIFIED.

This specification assumes that the above constraints will typically be met for main memory regions and may be met for certain I/O regions.

The Zicboz extension introduces an additional supported access type PMA for cache-block zero instructions. Main memory regions are required to support accesses by cache-block zero instructions; however, I/O regions may specify whether accesses by cache-block zero instructions are supported.

A cache-block management instruction is permitted to access the specified cache block if the supported access type PMAs and the PMP access control bits permit instruction fetches or load instructions to access the cache block in the same privilege mode; otherwise, the instruction raises a store access fault exception.

A cache-block zero instruction is permitted to access the specified cache block if the supported access type PMAs include cache-block zero instructions and if the supported access type PMAs and the PMP access control bits permit store instructions to access the cache block in the same privilege mode; otherwise, the instruction raises a store access fault exception.

A cache-block prefetch instruction is permitted to access the specified cache block if the supported access type PMAs and the PMP access control bits permit instruction fetches or load instructions to access the cache block in the same privilege mode; otherwise, the instruction is not permitted to access any caches or memory and does not raise any exceptions.

2.5.4. Address Misaligned Exceptions

CMO instructions do *not* generate address misaligned exceptions.

2.5.5. Breakpoint Exceptions 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:

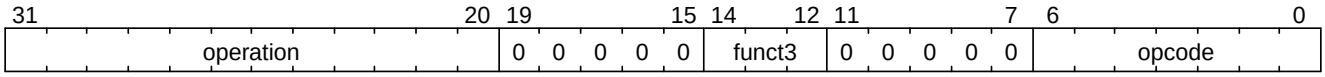
- Implementing address match triggers should be optional
- Type 6 data match triggers, i.e. `tdata1.type=6` and `mcontrol6.select=1`, should be unsupported
- Memory accesses may be considered to be loads or stores depending on the implementation, i.e. whether an address trigger matches on these instructions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

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.

2.5.6. Hypervisor Extension

For the purposes of writing the `mtinst` or `htinst` register on a trap, the following standard transformation is

defined for cache-block management instructions and cache-block zero instructions:



The `operation` field corresponds to the 12 most significant bits of the trapping instruction.

> *As described in the hypervisor extension, a zero may be written into `mtinst` or `htinst` instead of the standard transformation defined above.*

2.6. Effects on Constrained LR/SC Loops

Executing a cache-block management instruction or a cache-block zero instruction may cause a reservation on another hart to be lost. As a result, the following is added to the list of events that satisfy the eventuality guarantee of constrained LR/SC loops, as defined in the A extension:

- Some other hart executes a cache-block management instruction or a cache-block zero instruction to the reservation set of the LR instruction in *H*'s constrained LR/SC loop.

> *Executing a cache-block prefetch instruction does not impact the eventuality guarantee of any hart's constrained LR/SC loop, as defined in the A extension.*

2.7. Software Discovery

The initial set of CMO extensions requires the following information to be discovered by software:

- The size of the cache block for management and prefetch instructions
- The size of the cache block for zero instructions
- CBIE support at each privilege level

Other general cache characteristics may also be specified in the discovery mechanism.

Chapter 3. Control and Status Register State

The CMO extensions rely on state in envcfg CSRs that will be defined in a future update to the privileged architecture. If this CSR update 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 raises an illegal instruction or virtual instruction exception • `01`: The instruction is executed and performs a flush operation • `10`: *Reserved* • `11`: The instruction is executed and performs an invalidate operation
[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 raises an illegal instruction or virtual 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 raises an illegal instruction or virtual instruction exception • `1`: The instruction is executed

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

A CBO.INVAL instruction executes or raises either an illegal instruction exception or a virtual instruction exception based on the state of the xenvcfg.CBIE fields:

```
// illegal instruction exceptions
if (((priv_mode != M) && (menvcfg.CBIE == 00)) ||
    ((priv_mode == U) && (senvcfg.CBIE == 00)))
{
  <raise illegal instruction exception>
}
// virtual instruction exceptions
else if (((priv_mode == VS) && (henvcfg.CBIE == 00)) ||
         ((priv_mode == VU) && ((henvcfg.CBIE == 00) || (senvcfg.CBIE == 00))))
{
  <raise virtual instruction exception>
}
// execute instruction
else
{
  if (((priv_mode != M) && (menvcfg.CBIE == 01)) ||
      ((priv_mode == U) && (senvcfg.CBIE == 01)) ||
      ((priv_mode == VS) && (henvcfg.CBIE == 01)) ||
      ((priv_mode == VU) && ((henvcfg.CBIE == 01) || (senvcfg.CBIE == 01))))
  {
    <execute CBO.INVAL and perform flush operation>
  }
  else
  {
    <execute CBO.INVAL and perform invalidate operation>
  }
}
```

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 CBO.CLEAN or CBO.FLUSH instruction executes or raises an illegal instruction or virtual instruction exception based on the state of the xenvcfg.CBCFE bits:

```
// illegal instruction exceptions
if (((priv_mode != M) && !menvcfg.CBCFE) ||
    ((priv_mode == U) && !senvcfg.CBCFE))
{
  <raise illegal instruction exception>
}
// virtual instruction exceptions
else if (((priv_mode == VS) && !henvcfg.CBCFE) ||
         ((priv_mode == VU) && !(henvcfg.CBCFE && senvcfg.CBCFE)))
{
  <raise virtual instruction exception>
}
// execute instruction
else
{
  <execute CBO.CLEAN or CBO.FLUSH>
}
```

Finally, a CBO.ZERO instruction executes or raises an illegal instruction or virtual instruction exception based on the state of the xenvcfg.CBZE bits:

```
// illegal instruction exceptions
if (((priv_mode != M) && !menvcfg.CBZE) ||
    ((priv_mode == U) && !senvcfg.CBZE))
{
  <raise illegal instruction exception>
}
// virtual instruction exceptions
else if (((priv_mode == VS) && !henvcfg.CBZE) ||
         ((priv_mode == VU) && !(henvcfg.CBZE && senvcfg.CBZE)))
{
  <raise virtual instruction exception>
}
// execute instruction
else
{
  <execute CBO.ZERO>
}
```

Each xenvcfg register is WARL; however, software should determine the legal values from the execution environment discovery mechanism.

Chapter 4. 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 operate on cache blocks containing the effective address equal to the base address specified in *rs1*.

Cache-block zero instructions operate on cache blocks, or the memory locations corresponding to cache blocks, containing the effective address equal to the base address specified in *rs1*.

Cache-block prefetch instructions operate on cache blocks 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.

In all cases, the effective address is translated into a corresponding physical address by the appropriate translation mechanisms.

4.1. Cache-Block Management Instructions

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

- An invalidate operation makes data from store operations performed by a set of non-coherent agents visible to the set of coherent agents at a point common to both sets by deallocating all copies of a cache block from the set of coherent caches up to that point
- A clean operation makes data from store operations performed by the set of coherent agents visible to a set of non-coherent agents at a point common to both sets by performing a write transfer of a copy of a cache block to that point provided a coherent agent performed a store operation that modified the data in the cache block since the previous invalidate, clean, or flush operation on the cache block
- 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* agents in the system. In other words, an invalidate operation ensures that store operations from all non-coherent agents visible to agents in the set of coherent agents, and a clean operation ensures that store operations from coherent agents visible to all non-coherent agents.

> *The Zicbom extension does not prohibit agents 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 agents.*
>
> *Future extensions may define different sets of agents 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

4.2. Cache-Block Zero Instructions

Cache-block zero instructions store zeros to the set of bytes corresponding to a cache block. An implementation may update the bytes in any order and with any granularity and atomicity, including individual bytes.

> *Cache-block zero instructions store zeros independent of whether data from the underlying memory locations are cacheable. In addition, this specification does not constrain how the bytes are written.*

The following instructions comprise the Zicboz extension:

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

4.3. 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).

The following instructions comprise the Zicbop extension:

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

Chapter 5. Instructions

5.1. cbo.clean

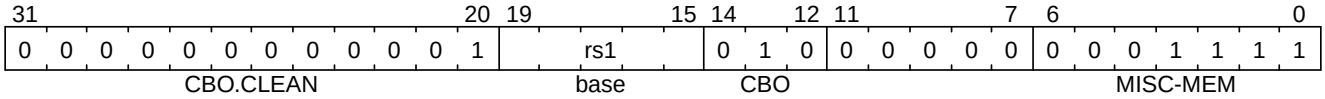
Synopsis

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 agent executing the instruction.

Operation

```
TODO
```

5.2. cbo.flush

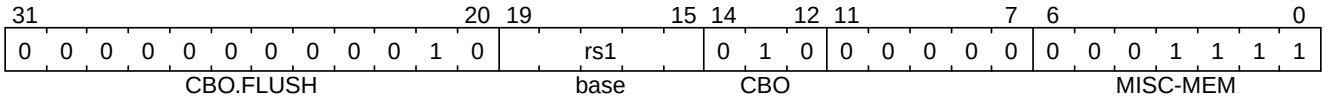
Synopsis

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 agent executing the instruction.

Operation

```
TODO
```

5.3. cbo.inval

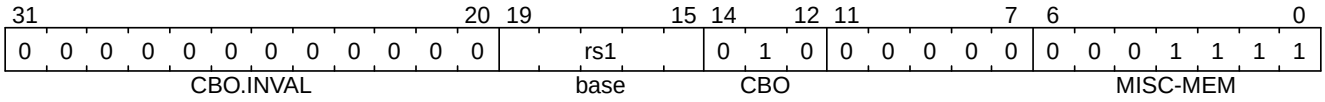
Synopsis

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 agent executing the instruction. The instruction may perform a flush operation, however, depending on CSR programming.

Operation

```
TODO
```

5.4. cbo.zero

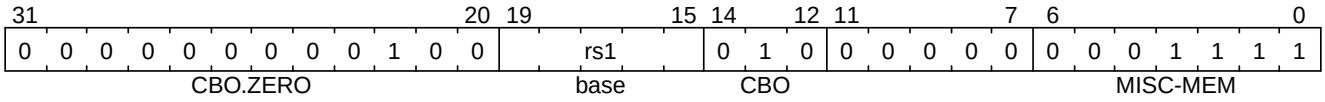
Synopsis

Store zeros to the set of the bytes corresponding to the cache block containing the effective address

Mnemonic

cbo.zero *base*

Encoding



Description

A **cbo.zero** instruction performs stores of zeros to the set of bytes corresponding to a cache block. An implementation may or may not update the entire set of bytes atomically.

Operation

```
TODO
```

5.5. prefetch.i

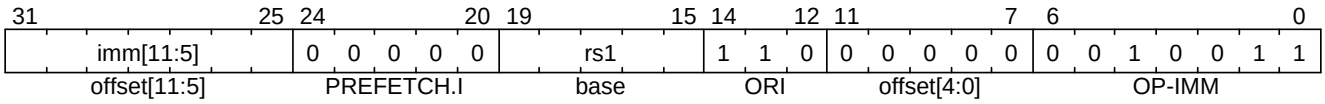
Synopsis

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 *offset*(*base*)

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
```

5.6. prefetch.r

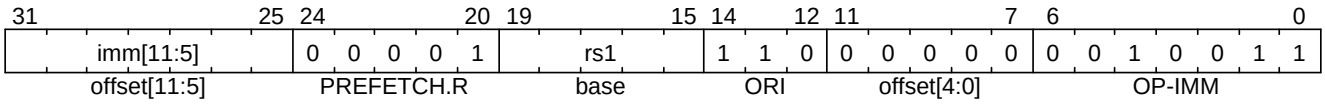
Synopsis

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 *offset*(*base*)

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
```

5.7. prefetch.w

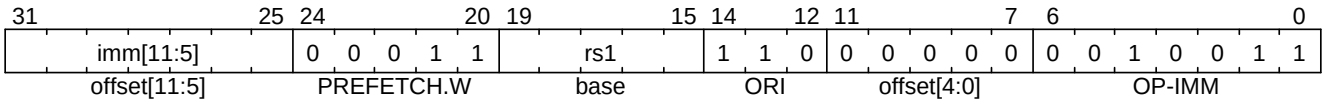
Synopsis

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 *offset*(*base*)

Encoding



Description

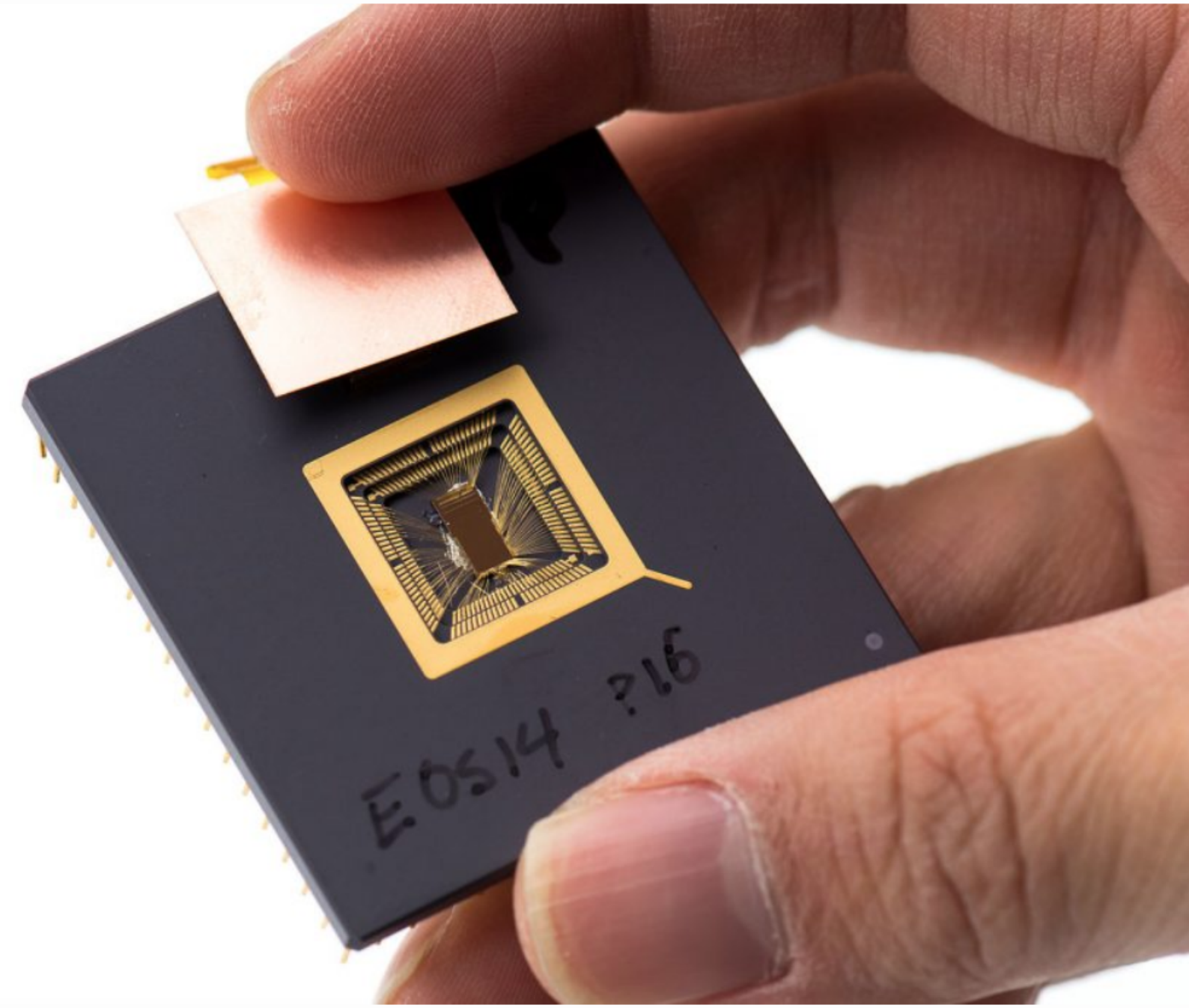
A **prefetch.w** instruction indicates to hardware that the cache block containing the effective address is likely to be accessed by a data write (i.e. store) in the near future.

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