Ri5 CMOs proposal: Cache Management Operations

2020-06-11 11:21:15 -0700 - When this draft was generated. Not necessarily modified. See Techpubs Information section below for more details.

About this document

This document is a proposal for cache management operations for RISC-V.

•

Table of Contents

1. CMO instruction formats and CMO operation types

There are 3 *formats* of CMO instructions:

- [Fixed Block Size Prefetches (PREFETCH.*)] operating on 64B naturally aligned regions of memory
- [Variable Address Range CMOs (CMO.VAR)] operating on arbitrary address ranges
- [Microarchitecture Structure Range CMOs (CMO.UR)] supporting whole cache operations operating on "cache entry numbers" or "indexes" which generalize and abstract cache set + way

There are *many* types of CMO operations, which are formed by the combination of

- which caches the operation applies to (and/or other parts of the memory system)
- what operation is actually performed (e.g. invalidate, flush dirty data)
- other aspects, such as invalidating related prefetchers and predictors

The CMO types are represented in the assembly syntax as the .<cmo_specifier> field. They are encoded in the instruction encoding as described below.

1.1. PREFETCH.* fixed size block prefetches

Briefly:

PREFETCH.64B.R: imm12.rs1:5.110.rd=00000.0010011, e.g. ORI with RD=x0

PREFETCH.64B.WL imm12.rs1:5.110.rd=00000.0110011, i.e. ANDI with RD=x0

See section [Fixed size block prefetches] for discussion.

1.2. CMO.VAR and CMO.UR instruction formats

The range CMOs - including CMO.VAR (variable address range CMOs) and CMO.UR (microarchitecture structure index range CMOs) - are encoded as folows:

MISC-MEM major opcode, with funct3=100 and 101.

`CMO.* Funct7:7.rs2:5.rs1:5.10x.rd:5.010111`

1.2.1. CMO funct7 field encodes .<cmo_specifier>

This 7 bit field encodes the type of the CMO:

- which caches the operation applies to (and/or other parts of the memory system)
- what operation is actually performed (e.g. invalidate, flush dirty data)
- other aspects, such as invalidating related prefetchers and predictors

The CMO types are represented in the assembly syntax as the .<cmo_specifier> field.

The CMO.VAR (address range) and CMO.UR (microarchecture index range) instructions interpret the funct7 .cmo_specifier field in the same way, if applicable. Some funct7 encodings are invalid for either CMO.VAR and/or CMO.UR.

1.2.2. CMO register fields in instruction encodings

These encodings have 3 register fields.

CMO.VAR are encoded with register fields. RS1 contains the start address. RS2 contains the stop address. RD is written with the updated start address. i.e. RD=RS1 is required, so that this range instruction take exceptions when partially complete. It is not validg for any of CMO.VAR's register operands to contain X0, the zero register.

`CMO.VAR: Funct7:7.rs2:5.rs1:5.10x.rd:5.010111` with RD=RS1. All of RD,RS1, and RS2 are != X0 (00000).

CMO.UR is encoded with register numbers RD=RS1, and RS2=X0, the zero register. RD=RS1 is once again required to permit partial progress and restartability after exceptions. RS1 contains the start index. Software initializes the instruction with RS1 set to -1 (all 1s). Instruction execution (HW or SW emulation) writes RD with the updated index. The enclosing software loop terminates when RD is -1 after the instruction. This allows CMO.UR to be a true NOP. It is not valid for RD and RS1, to be different, or to be X0, the zero register.

`CMO.UR: Funct7:7.00000:5.rs1:5.10x.rd:5.010111` with RD=RS1. All of RD,RS1, and RS2 are != X0 (00000).

These encodings with other values of the register operands are not valid encodings for CMO instructions, where "not valid" means that they are either undefined instructions, or are used to encode other instructions using the register fields.

In particular:

- RD = X0 is not valid for CMOs
- RS1 = X0 is not valid for CMOs
- RD != RS1 is not valid for CMOs
- while RS2=X0 distinguishes CMO.VAR from CMO.UR

Other instructions in this proposal using registing register number dependent instruction encodings:

TIMING_FLUSH: the encoding `TIMING_FLUSH.<timing_state>: Funct7:7.rs2:5.00000.100.00000.010111`, i.e. the CMO instruction, and Funct3=100, RD and RS1 = x0 is used to encode a TIMING_FLUSH instruction, with the sets of predictors and prefetchers bulk invalidated specified by Funct7 and RS2.

COMPLETION_FENCE: the encoding `COMPLETION_FENCE.<cmo_specifier>: Funct7:7.rs2:5.00000.101.00000.010111`, i.e. the CMO instruction, and Funct3=101, RD and RS1 = x0 is used to encode a COMPLETION_FENCE instruction, e.g. for persistence to battery backed up RAM or NVRAM as specified by the .<cmo_specifier> field.

Rationale: Register number dependent instruction encodings

CMO.VAR requires three register fields, given the requirement that it it update its start index.

CMO.UR only requires 2 register fields. It is proposed to use the CMO.VAR encoding with RS2=X0, not just to save instruction encodings, but to save the administrative hassle of obtaining a new set of 2-register instruction encodings. This is not an important consideration, just convenient. If decoding RS2=X0 poses difficulties, we can just allocate a new 2-register field instruyction encoding for CMO.VAR.

NOTE It may be useful to use RS1=X0 and RS2=start index for more flavours of CMO.UR. However, for the most part CMO.UR and CMO.VAR have the same .<cmo_specifiers>

This proposal describes CMO.VAR and CMO.UR as independent instructions, including for assembly syntax. Another instructions whose decode is based on register numbers, SFENCE.VMA is described as a singole instruction mnemonic, and it is always necessary to say things like SFENCE.VMA with RS1=x0 and RS2=x0 to order all reads and writes anywhere in the page table, versus RS1=x0 and RS2!=x0 ordering reads and writes only for the address space specified by RS2, and so on. These approaches are equivalent, except that it is fekt that separate mnemonics for CMO,VAR and CMO.UR increase understandability.

TBD: rename CMO.VAR. CMO.AR.* ?*

NOTE CMO.VAR.* were named *variable* address range to distinguish them from the *fixed* address range or block size instructions in an earlier version of this propsal. The *fixed* block size instructions have now been removed. CMO.VAR could be renamed CMO.AR

1.3. TIMING_FLUSH: Flushes of Microarchiture State that Affects Timing Channels

Requirement: it must be possible to flush all microarchitecture state that influences timing, such as predictors, prefetchers, LRU bits, etc. This is required to mitigate timing channels for security - i.e. to mitigate security flaws such as Spectre. It is occasionallty also desired to improved reprodeability of benchmarks and tests.

Some of these structures, such as LRU bits and some large branch predictors, are associated with memory addresses, and are flushed (zeroized) by the CMO.* range instructions when the appropriate bit in the .<cmo_specifier> funct7 is set, aka the "security" bit

Some of these mechanisms are associated with caches, etc., outside the CPU, but not necessarily with addresses or microarchitecture indexes. E.g. they may be bulk invalidated or O(1) flushable. These are flushed (zeroized) by the CMO.* range instructions when the appropriate bit in the .<cmo_specifier> funct7 is set, aka the "security" bit

Some of these mechanisms are not naturally associated with any caches are other data structures managed by the CMO.* instructions. Often these data structures can be bulk invalidated or O(1) flushable; or at least they are not so difficult to flush as to require instructions that support partial completion in the way that the CMO.VAR and CMO.UR instructions do.

An instruction is proposed to perform this last group of micfroarchitecture timing state flushes. It is encoded as follows, using the same MISC-MEM opcide as the CMO.VAR and CMO.UR range instructions, distinguished by register operands:

`TIMING_FLUSH.<timing_state>: Funct7:7.rs2:5.00000.100.00000.010111`

i.e. the CMO instruction encoding with RD and RS1 being x0.

The Funct7 field and RS2 value indicate what microarchitecture timing related state is to be flushed.

Such microarchitecture data structures are inherently implementation dependent.

The only standard values are Funct7 = 0000000 and RS2=x0, specifically the register number zero. These zero values indicate "flush all microarchitecture timing related state that can possibly flushed" \- or at least such state that is not separately flushed by one of the CMO.* instructions, by address or (most frequently) microarchitecture index.

Other values are implementation defined.

E.g. Funct7 and they contents of RS2 may be encoded as a bitmask, with a 0 in Funct7.bit0 indicated "flush all branch predictors", and so on.

1.4. COMPLETION_FENCE: ensure persistence when power is removed from CPU, or entire system including DRAM

Requirement: while many synchronization and ordering operations may be optimized away by microarchitecture so long as equivalent behaviour is obtained during normal operation, operations involving powering down the CPU (leaving state in battery backed up DRAM) or even tolerating powering down thec entire system (including battery backed up DRAM, leaving state in NVRAM) require that the operation actually be completed.

The instruction encoding

`COMPLETION_FENCE.<cmo_specifier>: Funct7:7.rs2:5.00000.101.00000.010111`

is provided for this purpose.

1.4.1. COMPLETION_FENCE..<cmo_specifier>.<which_cache>

The .<cmo_specifier> in Funct7 indicates to which level of the memory hierarchy completion is required. The level is encoded as in the CMO.* instructions.

The interpretation of the .<cmo_specifier>.<which_cache> values is the same for COMPLETION_FENCE as it is for CMO.* instructions. They are discussed here in detail, because COMPLETION_FENCE motivates some levels that may be surprsing for other C<O instructions

Which levels are implemented for COMPLETION_FENCE is implementation dependent. It is expected that, if a CMO ism provided to flush all state inside a level, then that level will be supported by COMPLETION_FENCE.

Implementations may provide completion semantics to any, some. or all levels of the memory hierarchy

Of particular importance are the .<cmo_specifier>.<which_cache> values that correspond to

- Battery backed up DRAM
  - e.g. to remove power from CPU
- First commit to non-volatile storage
  - persistence across power and battery failures
- Full commit to non-volatile storage
  - commit to redundant copies
  - survives failutes of one (or more) non-volatile storage devices.

Other completion/persistence levels are possible, for example

- persistence to non-battery-backed DRAM
  - permitting hot-plug while power is maintained

    - may be the same as completion to battery backed-up DRAM
- completion to points where non-cache-coherent memory accesses can be accessed comnsistenly
    - e.g. DRAM, if non-coherent I/O is only performed there
    - e.g. an L4 cache, if non-coherent I/O can inject into this level of the cache, but not further
    - e.g. a cache level shared by multiple CPUs that do not maintain full cache coherence to other caches
        - noting that it is possible for CPU non-coherence and I/O non-coherence to be resolved at different levels.

1.4.2. COMPLETION_FENCE ignores other parts of .<cmo_specifier>

COMPLETION_FENCE only takes heed of .<cmo_specifier>.<which_cache> field.

The specification of which operation is actually performed by a CMO instruction is ignored for COMPLETION_FENCE.

1.4.3. Which pending operations does COMPLETION_FENCE wait for?

COMPLETION_FENCE waits for completion of all pending operations in the from domain specified by .<cmo_specifier>, to the level specified by the to-domain of .<cmo_specifier>.

As discussed in .<cmo_specifier>, this may be limited to operations produced locally, e.g. by the current CPU, or it may extend to other CPUs in a cohedrence domain, especially if there may arise migration of data between peer caches without updating outer hierarchy levels.

1.5. Issues for COMPLETION_FENCE

NOTE

TBD: Issues for COMPLETION_FENCE

Q: Should COMPLETION_FENCE apply to specific operation types - e.g. writebacks, but not invalidates? A: as proposed, COMPLETION_FENCE applies to all operations initiated by CMO instructions, e.g. FLUSHes that write modified data to outer levels, and INVALIDATEs that remove data that may be rendered stale by non-coherent actions buy other devices. COMPLETION_FENCE does not apply to stores that are not affected by a CMO.* instruction.

Q: Should COMPLETION_FENCE apply to specific memory addresses? A: not proposed. If this is to be done, it will an address range oriented instruction encoding, with RS1 and RD, just like CMO.VAR - essentially a new .<cmo_specifier>.<cmo_operation>

Q: Should it be necessary to apply a COMPLETION_FENCE after any CMO? I.e. is it permitted to implement CMOs in a non-blockimg or asynchronous manner, and require COMPLETION_FENCE to ensure completion_fence even just for ordering semantics?

Q: Should COMPLETION_FENCE be preemptable? A: yes, probably, since may be very long latency. But there is no address or index range that can be monotonically completed to guarantee forward progress.

Q: Perhaps COMPLETION_FENCE should return a value, so that it can be wrapped in a loop? Q: but then do context switches need to save/rstore a progress indicator? A: not pursuing at this time. Would need to permit non-zero RD, with zero RS1 - an encoding which is available, but not currently permitted for COMPLETION_FENCE.

A: strawman: COMPLETION_FENCE is blocking. OS may need to emulate. Otherwise, restarts from scratch, which may make forward progress difficult if othrr harts can initiate CMOs while yhe first is preempted.

Q: shoould the delegation mechanism comprehend COMPLETION_FENCE? A: yes, probably. Probably needs to be treated like an extra .<cmo_operation>.<cmo_operation> value, for purposes of allocating corresponding fields in the CMO delegation CSRs.

•

2. Fixed Block Size PREFETCHes

Proposed name: PREFETCH.64B.R

- encoding: ORI with RD=R0, i.e. M[rs1+offset12]
  - imm12.rs1:5.110.rd=00000.0010011
- affects cache line containing virtual address M[rs1+offset12]
- see ***Mnemonics and Names*** for a discussion of proposed mnemonics and names

Proposed name: PREFETCH.64B.W [^mnemonics]

- encoding: ANDI with RD=R0, i.e. M[rs1+offset12]
  - imm12.rs1:5.110.rd=00000.0110011
- affects cache line containing virtual address M[rs1+offset12]
- see ***Mnemonics and Names*** for a discussion of proposed mnemonics and names

2.1. OBSOLETE: Fixed Block Size Clean and Flush CMOs

NOTE	*Obsolete* Earlier drafts of this propiosal contained fixed block size CMOs, e.g. cache flushes. Like the PREFETCHes, but without the full addressing mode to save instruction encoding space. These have been removed from the proposal, subsumed by the prefetch flavors of the variable address range CMO.VAR instructions.

2.2. DETAILS

- Page Fault: NOT taken for PREFETCH
  - The intent is that loops may access data right up to a page boundary beyond which they are not allowed, and may contain prefetches that are an arbitrary stride past the current ordinary memory access. Therefore, such address range prefetches should be ignored.
    - ⇒ Not useful for initiating virtual memory swaps from disk, copy-on-write, and prefetches in some "Two Level Memory" systems, e.g. with NVRAM, etc., which may involve OS page table management in a deferred manner. (TBD: link to paper (CW))
- Page Fault: NOT taken for CMO.CLEAN/FLUSH
  - again, the intent is that the CMOs defined on this page may be treated as NOPs or hints by an implementation. I.e. they are for performance only.
  - Note that this implies that these CMOs /may/ not be suitable for cache flushing related to software consistency or persistence.
    - Some OSs treat the hardware page tables as a cache for a larger data structure that translates virtual to physical memory address translation
    - This means that physical addresses in the cache may be present even the translations from their virtual address those physical addresses are no longer present in the page tables. In such a situation a true guaranteed flush might require taking page faults.
    - Obviously this is OS specific. Software with knowledge of the OS behavior may use these instructions for guaranteed flushes. However, it is not possible for the instruction set architecture to make this guarantee.
- Debug exceptions, e.g. data address breakpoints: YES taken.

Note that page table protections are sometimes used as part of a debugging strategy. Therefore, ignoring page table faults is inconsistent with permitting debug exceptions

- ECC and other machine check exceptions: taken?
  - In the interest of finding bugs earlier.
  - Although this is somewhat incompatible with allowing these prefetches and CMOs to become NOPs

NOTE

Rationale: Addressing Modes

Want full addressing mode for fixed block size prefetches, `Reg+Offset`, so that compiler can just add a prefetch stride to the offset, doesn't neeed to allocate extra registers for the prefedtch address

NOTE

Rationale:Fixed minimum block size - NOT cache line size

These instructions are associated with a fixed block size - actually a minimum fixed block size. NOT necessarily the microarchitecture specific cache line size.

Currently the fixed block size is only defined to be 64 bytes. Instruction encodings are reserves for other block sizes, e.g. 256 bytes. However, there is unlikely to be room to support all possible cache line sizes in these instructions.

The fixed block size of these instructions is NOT necessarily a cache line size. The intention is to hide the microarchitecture cache line size, which may even be different on different cache levels in the same machine, while allowing reasonably good performance across machines with different cache line sizes.

The fixed minimum block size (FSZ) is essentially a contract that tells software that it does not need to prefetch more often than that size. Implementations are permitted to "round up" FSZ: e.g. on a machine with 256 byte cache lines, each PREFETCH.64B.[RW] and CMO.64B.{CLEAN,FLUSH} may apply to an entire 256 byte cache line. Conversely, on a machine with 32 byte cache lines, it is recommended that implementations of these instructions to address A apply similar operations to cache lines containing address A and A+32. "It is recommended" because it is permissible for all of these operations defined on this page to be ignored, treated as NOPs or hints.

The intent of the fixed minimum block size is to set an upper bound on prefetch instruction overhead. E.g. if standing an array of 32 byte items `LOOP A[i] ENDLOOP`, one might prefetch at every iteration of the loop `LOOP A[i]; prefetch A[i+delta] ENDLOOP`. However, prefetch instruction overhead often outweighs the memory latency benefit of prefetch instructions. If one knows that the cache line size is 256 bytes, i.e. once every 256/4=64 iterations of the loop, one might unroll the loop 64 times `LOOP A[i+0]; ⋯ A[i+63]; prefetch A[i+63+delta] ENDLOOP`, thereby reducing the prefetch instruction overhead to 1/64. But if the cache line size is 64 bytes you only need to enroll 64/4=16 times: `LOOP A[i+0]; ⋯ A[i+15]; prefetch A[i+15+delta] ENDLOOP`. The prefetches are relatively more important, but the overhead of unrolling code to exactly match the line size is greatly reduced.

The fixed minimum block size is an indication that the user does not need to place prefetches any closer together to get the benefit of prefetching all of a contiguous memory region.

•

3. Variable Address Range CMOs

Traditional CMOs are performed a cache line at a time, in a loop. This exposes the cache line size, and inhibits performance for some implementations.

Some use cases require or prefer CMOs that apply to a set of memory addresses, typically a

contiguous range. Furthermore, address ranges permit optimizations that perform better on some implementations than looping a cache line at a time.

This proposal defines the instruction in such a way that allows [_possible_implementations_ranging_from_cache_line_at_a_time_to_full_address_range], with a loop such as that below

In pseudocode:

```
x11 := lwb
x12 := upb (= lwb + size_in_bytes)
LOOP
   CMO.VAR.<> x11,x11,x12
UNTIL x1 ==x12
```

In assembly code:

```
    x11 := lwb
    x12 := upb
L:  CMO.VAR.<> x11,x11,x12
    bne x11,x12,L
```

See below, [_possible_implementations_ranging_from_cache_line_at_a_time_to_full_address_range], for more details.

3.1. CMO.VAR: Variable Address Range CMOs

Proposed name: CMO.VAR.<cmo-specifier>

Encoding: R-format

- R-format: 3 registers: RD, RS1, RS2
  - Register numbers in RD and RS1 are required to be the same
    - If the register numbers in RD and RS1 are not the same an illegal instruction exception is raised (unless such encodings have been reused for other instructions in the future).
    - The term RD/RS1 will refer to this register number
- numeric encoding: TBD
  - 2 funct7 encodings ⇒ 256 possible <cmo-specifiers>

Assembly Syntax:

- CMO.VAR.<cmo-specifier> rd,rs1,rs2

But, since register numbers in RD and RS1 are required to be the same, assemblers may choose to provide the two register operand version

- CMO.VAR.<cmo-specifier> rd_and_rs1,rs2

Operands:

- Input:
  - memory address range:
    - RS1 (RD/RS1) contains `lwb`, the lower bound, the address at which the CMO will start
    - RS2 contains `upb`, the upper bound of the range
  - type of operation and caches involved
    - .<***cmo-specifier***>: i.e. specified by the encoding of the particular CMO.VAR instruction
- Output
  - RD (RSD/RS1) contains `stop_address`, the memory address at which the CMO operation stopped
    - if RD = RS2: `upb`, the operation was completed
  - if RD = RS1: `lwb`, the operation stopped immediately, e.g. an exception such as a page fault or a data address breakpoint at lwb
  - if `lwb` < RD < `upb`, the operation has been partially completed
    - e.g. at an exception

3.1.1. Range specification

The CMO is applied to the range [RS1,RS2), i.e. to all memory addresses A such that RS1 <= A < RS2. Not that the upper bound `upb` is exclusive, one past the end of the region. This allows the calculation `upb=lwb+size_in_bytes`.

Pedantically, the range is all memory addresses `A` such that `0 ⇐ A < upb-lwb`. This permits wrapping around the address space. To specify a range that reaches the maximum possible (unsigned) address, specify `upb=0`.

3.1.2. Return value RD

This instruction family is ***restartable after partial completion***. E.g. on an exception such as a page fault or debug address breakpoint the output register RD is set to the data address of the exception, and since the instruction is ***source/dest***, with the register numbers in RD and RS1 required to be the same, returning from the exception to the CMO.VAR instruction will pick up execution where it left off.

Similarly, implementations may only process part of the range specified by [RS1,RS2), e.g. only the 1st cache line, setting RD to an address *within* the next cache line, typically the start, Software using this instruction is required to wrap it in a loop to process the entire range.

See [_loop_to_support_cacheline_at_a_time_implementations].

3.1.3. CMO Operation Type and Caches Involved - .<cmo-specifier>

The .<cmo-specifier> is derived from the instruction encoding. This proposal asks for a total of 256, two funct7 R-format encoding groups.

The .<cmo-specifier> specifies both the caches involved in the CMO - more precisely, the parts of the cache hierarchy involved - as well as the actual cache management operation.

The cache management operations specified include

- CLEAN (write back dirty data, leaving clean data in cache)
- FLUSH (writeback dirty data, leaving invalid data in cache) and other operations, as well as the caches involved. See ***CMO (Cache Management Operation) Types***. (TBD: I expect that one or more of the .<cmo-specifier> will be something like a number identifying a group of CSRs loaded with an extended CMO type specification.)

In assembly code certain CMO specifiers will be hardlined, and others may be indicated by the group number:

- CMO.VAR.CLEAN
- CMO.VAR.FLUSH
- CMO.VAR.0
- CMO.VAR.1

TBD: full list of CMOs .<cmo-specifiers> is in a spreadsheet. TBD: include here.

3.2. DETAILS

3.2.1. Range Definition [RS1:lwb,RS2:upb)

The CMO is applied to the range [RS1,RS2), i.e. to all memory addresses A such that RS1 <= A < RS2. Not that the upper bound `upb` is exclusive, one past the end of the region. This allows the calculation `upb=lwb+size_in_bytes`.

Pedantically, the range is all memory addresses `A` such that `0 ⇐ A < upb-lwb`. This permits wrapping around the address space. To specify a range that reaches the maximum possible address, spercify `upb=0`.

CMOs (Cache Maintenance Operations) operate on ***NAPOT*** memory blocks such as cache lines, e.g. 64B, that are implementation specific, and which may be different for different caches in the system.

CMO.VAR is defined to always apply to at least such memory block, even if RS1 >= RS2.

The range's upper and lower bounds, RS1:`lwb` and `upb` are *not* required to be aligned to the relevant block size. Therefore RS1:`lwb` is an address *within* the first memory block to which the operation will apply. Similarly, `upb`, the highest address in the range specified by the user, may lie within such a memory block, so the operation may include and apply beyond `upb` to the next block boundary.

As described in ***Advisory vs Mandatory CMOs***:

- Some CMOs are optional or advisory: they may or may not be performed,
  - Such advisory CMOs may be performed beyond the range [`lwb`,`upb`)
- However, some CMOs are mandatory, and may affect the values observed by ***timing independent code***.
  - if `upb` lies in a memory block that does not overlap any of the blocks in [`lwb`,`upb`) then the implementation must guarantee that the mandatory or destructive CML has not been applied to the memory block starting at address `upb`.

Security timing channel related CMOs are mandatory but do not affect the values observed by ***timing independent code***. TBD: are such CMOs required not to apply beyond the ***address range rounded to block granularity***? POR: it is permitted for any non-value changing operations to apply beyond the range.

NOTE	There is much disagreement with respect to terminology, whether operations that directly affect values (such as ***DISCARD cache line***) are to be considered CMOs at all, or whether they might be specified by the CMO instructions such as CMO.VAR. For the purposes of this discussion we will assume that they could be specified by these instructions.

3.2.2. Possible implementations of CMO.VAR ranging from cache line at a time to full address range

The CMO.VAR instruction family permits implementations that include

1. operating a cache line at a time
2. trapping and emulating (e.g. in M-mode)
3. HW state machines that can operate on the full range
   - albeit stopping at the first page fault or exception.

First: Cache line at a time implementations are typical of many other ISAs, RISC and otherwise.

Second: On some implementations the actual cache management interface is non-standard, e.g. containing sequences of CSRs or MMIO accesses to control external caches. Such implementations may trap the CMO instruction, and emulate it using the idiosyncratic mechanisms. Such trap and emulation would have a high-performance cost if performed a cache line at a time. Hence, the address range semantics, permitting the trap ciost to b e amortized.

Third: While hardware state machines have some advantages, it is not acceptable to block interrupts for a long time while cache flushes are applied to every cache line in address range. Furthermore, address range CMOs may be subject to address related exceptions such as page-faults and debug breakpoints.

3.2.2.1. CMO.VAR Loop to support cacheline at a time implementations

The CMO.VAR instruction is intended to be used in a software loop such as that below:

In pseudocode:

```
x11 := lwb
x12 := upb (= lwb + size_in_bytes)
LOOP
   CMO.VAR.<> x11,x11,x12
UNTIL x1 ==x12
```

In assembly code:

```
    x11 := lwb
    x12 := upb
L:  CMO.VAR.<> x11,x11,x12
    bne x11,x12,L
```

Note that the closing comparison BNE is exact. The CMO.VAR instruction is required to return the exact upper bound when it terminates

NOTE	*Rationale: Exact next start address returned in RD* Returning the exact upper bound rather than an address in a cache block containing or just past the upper bound, allows the exact comparison BNE in the reference loop, and hence permits the exclusive range to apply right up to last address, and to wrap, at the cost of a more complicated address computation.

3.2.2.2. Variable Address Range CMO loop construct

The software loop around the CMO range instructions is required only to support cache line at a time implementations. If this proposal only wanted to support hardware state machines or trap and emulate, the software loop would not be needed.

Similarly, the upper bound operand RS2:upb?, is only required to support address range aware implementations, such as trap and emulate or hardware state machines. Cache line at a time implementations may ignore the RS2 operand. Therefore, the operation is always applied to at least one memory address.

To guarantee that the loop wrapped around the CMO range instructions makes forward progress in the absence of an exception the value output to RD must always be greater than the value input from RS1, recalling that register numbers RD and RS1 are required to be the same. (On an exception output RD may be unchanged from input RS1.)

Typically, the output value RD will be the start address of the next cache block.

To guarantee that the loop terminates, on the final iteration the output value RD must be equal to RS2.

In other words ~ IF rs1 && rs2 are in the same cache line perform CMO for cache line containing rs1 IF not at beginning of cache line rd := rs2 ELSE perform CMO for cache line containing rs1 rd :=

(rs1 + CL_SIZE) & ((1<<CL_SIZE)-1) ~

Although some CMOs may be optional or advisory, that refers to their effect upon memory or cache. The range oriented CMOs like CMO.VAR cannot simply be made into NOPs, because the loops above would never terminate. The cache management operation may be dropped or ignored, but RD must always be set to guarantee that the loop will make eventually terminate,

3.2.3. Exceptions

- Illegal Instruction Exceptions: taken, if the CMO.VAR.<cmo-specifier> is not supported.
- Permission Exception: for CMO not permitted
  - Certain CMO (Cache Management Operations) may be permitted to a high privilege level such as M-mode, but may be forbidden to lower privilege lebels such as S-mode or U-mode.
  - TBD: exactly how this is reported. Probably like a read/write permission exception. Possibly requiring a new exception because identifier
- Page Faults: taken
- Other memory permissions exceptions (e.g. PMP violations): taken
- Debug exceptions, e.g. data address breakpoints: taken.
- ECC and other machine checks: taken or logged
  - see below

3.2.4. ECC and other machine check exceptions during CMOs

NOTE	the term "machine check" refers to an error reporting mechanism for errors such as ECC or lockstep execution mismatches. TBD: determine and use the appropriate RISC-V terminology for "machine checks".

Machine checks may be reported as exceptions or recorded in logging registers or counters without producing exceptions.

In general, machine checks should be reported if enabled and if an error is detected that might produce loss of data. This consideration applies to CMOs: e.g. if a CMO tries to flush a dirty cache line that contains an uncorrectable error, a machine check should be reported. However, an uncorrectable error in a clean cache line may be ignorable since it is about to be invalidated and will never be used in the future.

Similarly, a DISCARD cache line CMO may invalidate dirty cache line data without writing it back. In which case, even an uncorrectable error might be ignored, or might be reported without causing an exception.

Such machine check behavior is implementation dependent.

3.2.5. Permissions for CMOs

3.2.5.1. CMO.VAR: Memory address based permissions for CMOs

The CMO.VAR.<cmo-specifier> instructions affect one or more memory addresses, and therefore are subject to memory access permissions.

Most CMO (Cache Management Ops) require only read permission:

- CLEAN (write out dirty data, leaving clean data in cache)
- FLUSH (Write out dirty data, invalidate all lines)

Even though "clean" and "flush" may seem to be like write operations, and the dirty data can only have occurred as result of write operations, the dirty cache lines may have been written by a previous mode that shares memory with the current mode that has only read access.

The overall principal is, if software could have accomplished the same operation e.g. flushing dirty data or evicting lines, using ordinary loads and stores, then only read permissions are required.

If the operation is performed read permissions are required to all bytes in the range.

(If an optional or advisory operation is not performed, no read permissions checks or exceptions are required.)

Some CMOs affect values, and therefore require at least write permission:

- ZALLOC (Allocate Zero Filled Cache Line without RFO)
  - e.g. IBM POWER DCBZ

3.2.5.2. Permissions by CMO type

Some CMOs not only affect value, but might also affect the cache protocol and/or expose data from other privileged domains. If implemented, these require privileges beyond those specified for memory addresses. Such operations include:

- CLALLOC (Allocate Cache Line with neither RFO nor zero fill)
  - e.g. IBM POWER DCBA
- DISCARD cache line
  - discard dirty data without writing back

Similarly, while it might be possible for an ordinary user to arrange to flush a line out of a particular level of the cache hierarchy, doing so with ordinary loads and stores might be a very slow process, whereas doing so with a CMO instruction would be much more efficient, possibly leading to DOS (Denial of Service) attacks. Therefore, even CMOs that might otherwise require only read permission may be "modulated" by privileged software.

See section Privilege for CMOs which applies to both address range CMO.VAR.<cmo-specifier> and microarchitecture entry range CMO.UR.<cmo-specifier> CMOs, as well as to ***Fixed Block Size CMOs*** and prefetches.

•

4. Microarchitecture Structure Range CMOs

Some situations require cache management operations that are NOT associated with a single address or an address range.

E.g. if an entire cache needs to be invalidated, it is inefficient to iterate over every possible address that might be in the cache.

Some traditional RISC ISAs instructions that invalidate by (set,way). Problems with this include: exposing microarchitecture details to code that might otherwise be portable, inability to take advantage of hardware optimizations like bulk invalidates and state machines, etc.

This proposal defines instructions in such a way that allows [possible_implementations_ranging_from_cache_line_at_a_time_to_full_address_range], with a loop such as that below:

In pseudocode:

```
x11 := -1
LOOP
   CMO.UR.<> x11,x11
UNTIL X11 < 0
```

In assembly code:

```
    ADDI x11,x0,-1
L:  CMO.UR.<> x11,x11
    BGEZ x11,L
```

4.1. SUMMARY: Microarchitecture Structure Range CMOs

Proposed name: CMO.UR.<cmo-specifier>

Encoding: R-format

- 2 registers: RD, RS1
  - R format actually has three registers: unused register RS2 is required to be zero
  - Register numbers in RD and RS1 are required to be the same
    - Why?: restartability
    - If the register numbers in RD and RS1 are not the same an illegal instruction exception is raised (unless such encodings have been reused for other instructions in the future).

        - The term RD/RS1 will refer to this register number

Assembly Syntax:

- CMO.UR.<cmo-specifier> rd,rs1,x0

But, since register numbers in RD and RS1 are required to be the same, and RS2 is required X0, assemblers are encouraged to provide the single register operand version

- CMO.UR.<cmo-specifier> rd_and_rs1

Operands:

- Input:
    - memory address range:
        - RS1 (RD/RS1) contains `start_entry` or `index`, the ***microarchitecture entry number*** for the specified cache at which the CMO will start
            - RS1 = zero is the first entry
    - type of operation and caches to which it is applied
        - .<***cmo-specifier***>: i.e. specified by the encoding of the particular CMO.UR instruction
- Output
    - RD (RSD/RS1) contains `stop_entry`, the microarchitecture entry number at which the CMO operation stopped
        - if RD is negative the operation has completed
            - IF RD=-1 (all 1s unsigned, ~0) the operation completed successfully
            - Other negative values of RD are reserved

This instruction family is ***restartable after partial completion***. E.g. on and exception such as a ***machine check error*** or a debug address breakpoint the output register RD is to the microarchitecture entry number where the exception was incurred. Since the instruction is ***source/dest***, with the register numbers in RD and RS1 required to be the same, returning from the exception to the CMO.UR instruction will pick up execution where it left off.

Similarly, implementations may only process part of the range specified by microarchitecture entry numbers [0,num_entries), e.g. only the 1st cache line, setting RD/RS1 to an address *within* the next cache line. Software using this instruction is required to wrap it in a loop to process the entire range.

The .<cmo-specifier> derived from the instruction encoding (not a general-purpose register operand) specifies operations such as

- CLEAN (write back dirty data, leaving clean data in cache)
- FLUSH (writeback dirty data, leaving invalid data in cache) and other operations, as well as the caches involved. See ***CMO (Cache Management Operation) Types***. (TBD: I expect that one or more of the .<cmo-specifier> will be something like a number identifying a group of CSRs loaded with an extended CMO type specification.)

In assembly code certain CMO specifiers will be hardlined, and others may be indicated by the group number:

- CMO.UR.CLEAN
- CMO.UR.FLUSH
- CMO.UR.0
- CMO.UR.1

Loops to support cacheline at a time implementations - CMO.UR

In pseudocode:

```
x11 := -1
LOOP
   CMO.UR.<> x11,x11
UNTIL X11 < 0
```

In assembly code:

```
    ADDI x11,x0,-1
L:  CMO.UR.<> x11,x11
    BGEZ x11,L
```

4.2. DETAILS

4.2.1. Microarchitecture Entry Range - countdown

When used in a loop such as `X11:= ~0; LOOP CMO.UR X1; UNTIL X11 < 0` the cache management operation is applied to a range of microarchitecture entry numbers for the cache specified by the .<cmo-type> field of the CMO.UR instruction.

For the purposes of this instruction, a cache has entries numbered from 0 extending to some maximum entry number.

The CMO.UR instructions avoid the need for the user to know or discover the number of lines in the cache, by starting the iteration at the possible number (all set bits, i.e. -1 or ~0) and modifying the index until <0. The implementation is therefore required to "skip" index numbers that are not valid.

Typical cache entries have have (set,way) coordinates.
The microarchitecture entry number may be a simple transformation such as `e = set*nways+way` or `e = way*nsets+set`, and the iteration may simply decrement by one for every cache line affected until the maximum number of entries is reached.

Pseudocode for a simple CMO.UR instruction implementation might look like:

```
CMO.UR rd,rs1  // where register numbers rd and rs1 are required to be the same
    index := rs1 & (CACHE_ENTRIES - 1)
    perform CMO for cache entry #index
    index := index-1
```

Other recurrences for the index are possible. All that is required is that the sequence begin with -1 and terminate with -1, with all intermediate values positive and no repetitions, so that the reference CMO.UR loop loop is guaranteed to make forward progress and eventually terminate, after visiting all of the entries in the cache, It is not required that the index be monotonically decreasing.

Indeed "twisting" the index sequence might be used to hide microarchitecture details and mitigate information leaks. (The twisting might even be PC dependent.)

The sequence of indexes may contain values that do not map to actual cache lines, so long as those invalid mappings do not cause exceptions. (E.g. ***Way Locking and CMO.UR*** or ***Multiple Caches and CMO.UR***.) Such implementations should not, however, "waste too much time" on invalid entries.

Users should not assume or rely on a simple mapping of CMO.UR indexes to (set,way). E.g. users should not assume that they can invalidatean entire of a 4-way set associative cache by stepping the index by -4 in a non-standard loop structure.

See CMO.UR indexes should not be created out of thin air.

4.2.2. *Advisory vs Mandatory CMOs*

As described in ***Advisory vs Mandatory CMOs***:

- Some CMOs are optional or advisory: they may or may not be performed,
  - Such advisory CMOs may be performed beyond the range of microarchitecture entry numbers specified
- However, some CMOs are mandatory, and may affect the values observed by ***timing independent code***.
  - Such architectural CMOs are guaranteed not to be performed beyond the range of microarchitecture entry numbers specified (?? TBD: is this possible, if cache line size is very ??)

Security timing channel related CMOs are mandatory but do not affect the values observed by ***timing independent code***. POR: it is permitted for any non-value changing operations to apply beyond the range.

NOTE	There is much disagreement with respect to terminology, whether operations that directly affect values (such as ***DISCARD cache line***) are to be considered CMOs at all, or whether they might be specified by the CMO instructions such as CMO.UR. For the purposes of this discussion we will assume that they could be specified by these instructions.

4.2.3. Possible implementations ranging from cache line at a time to whole cache

The CMO.UR instruction family permits implementations that include

1. operating a cache line at a time
2. trapping and emulating (e.g. in M-mode)
3. HW state machines that can operate on the full range
    - albeit stopping at the first page fault or exception.

First: Cache line at a time implementations using (set,way) are typical of many other ISAs, RISC and otherwise.

Second: On some implementations the actual cache management interface is non-standard, e.g. containing sequences of CSRs or MMIO accesses to control external caches. Such implementations may trap the CMO instruction, and emulate it using the idiosyncratic mechanisms. Such trap and emulation would have high performance cost if performed a cache line at a time. Hence, the address range semantics.

Third: While hardware state machines have some advantages, it is not acceptable to block interrupts for a long time while cache flushes are applied to every cache line in address range. Furthermore, address range CMOs may be subject to address related exceptions such as page-faults and debug breakpoints. The definition of this instruction permits state machine implementations that are ***restartable after partial completion***.

4.2.4. *Actual CMO Operations*

4.2.4.1. Discussion:

The software loop around the CMO range instructions is required only to support cache line at a time implementations. If this proposal only wanted to support hardware state machines or trap and emulate, the software loop would not be needed.

Although some CMOs may be optional or advisory, that refers to their effect upon memory or cache. The range oriented CMOs like CMO.UR cannot simply be made into NOPs, because the loops above would never terminate. The cache management operation may be dropped or ignored, But RD must be set in such a way that the sequence beginning with zeros will eventually touch all cache lines necessary and terminate with -1. (TBD: link the text above.)

4.2.5. CMO.UR: Exceptions

- Illegal Instruction Exceptions: taken, if the CMO.UR.<cmo-specifier> is not supported.
- Permission Exception: for CMO not permitted
    - Certain CMO (Cache Management Operations) may be permitted to a high privilege level such as M-mode, but may be forbidden to lower privilege lebels such as S-mode or U-mode.
    - TBD: exactly how this is reported. Probably like a read/write permission exception. Possibly requiring a new exception because identifier

- Page Faults:
  - most cache hierarchies cannot receive page-faults on CMO.UR instructions, since the virtual the physical address translation has been performed before the data has been placed in the cache
  - however, there do exist microarchitectures (not necessarily RISC-V microarchitectures as of the time of writing) whose caches use virtual addresses, and which perform the virtual the physical address translation on eviction from the cache
    - such implementations *might* receive page-faults, e.g. evicting dirty data for which there is no longer a valid virtual to physical translation in TLB or page table
    - although we recommend that system SW on such systems arrange so that dirty data is flushed before translations are invalidated
- Other memory permissions exceptions (e.g. PMP violations): taken
- Debug exceptions, e.g. data address breakpoints: taken.
- ECC and other machine checks: taken

4.2.6. ECC and other machine check exceptions during CMOs

NOTE	the term "machine check" refers to an error reporting mechanism for errors such as ECC or lockstep execution mismatches. TBD: determine and use the appropriate RISC-V terminology for "machine checks".

Machine checks may be reported as exceptions or recorded in logging registers or counters without producing exceptions.

In general, machine checks should be reported if enabled and if an error is detected that might produce loss of data. This consideration applies to CMOs: e.g. if a CMO tries to flush a dirty cache line that contains an uncorrectable error, a machine check should be reported. However, an uncorrectable error in a clean cache line may be ignorable since it is about to be invalidated and will never be used in the future.

Similarly, a DISCARD cache line CMO may invalidate dirty cache line data without writing it back. In which case, even an uncorrectable error might be ignored, or might be reported without causing an exception.

Such machine check behavior is implementation dependent.

4.2.7. Permissions for CMOs

4.2.7.1. Memory address based permissions for CMOs

Most CMO.UR.<> implementations do not need to use address based permissions. CMO.UR for the most part are controlled by ***Permissions by CMO type***.

Special cases for memory address based permissions for CMO.UR include:

E.g. virtual address translation permissions

- do not apply to most implementations
- might apply to implementations that perform page table lookup when evicting dirty data from the cache.
    - are not required to invalidate cache lines in such implementations

E.g. PMP based permissions

- TBD: what should be done if CMO.UR is evicting a dirty line a memory region whose PMP indicates not writable in the current mode?
    - this may be implementation specific
    - most implementations will allow this
        - assuming that privileged SW will have flushed the cache before entering the less privilege mode in order to prevent any problems that might arise (e.g. physical DRAM bank switching)

4.2.7.2. *Permissions by CMO type*

See section ***Permissions by CMO type*** which applies to both address range CMO.UR.<cmo-specifier> and microarchitecture entry range CMO.VAR.<cmo-specifier> CMOs, as well as to ***Fixed Block Size CMOs***.

4.2.8. Multiple Caches and CMO.UR

Cache management operations may affect multiple caches in a system. E.g. flushing data from a shared L2 may invalidate data in multiple processors' L1 I and D-caches, in addition to writing back dirty data from the L2, while traversing and invalidating an L3 before eventually being sent to memory. However, often the invalidation of multiple peer caches, the L1 I and D caches, is accomplished by cache inclusion mechanisms such as backwards and validate.

However, sometimes it is necessary to flush multiple caches without relying on hardware coherence cache inclusion. This could be achieved by mapping several different caches's (set,way) or other physical location into the same microarchitecture entry number space. However, this is by no means required

4.3. *CMO UR index*

4.3.1. Traditional microarchitecture cache invalidation loops

Many ISAs invalidate a cache in time proportional to the number of entries within the cache using a code construct that looks like the following:

```
FOR S OVER ALL sets in cache C
   FOR W OVER ALL ways in cache C
        INVALIDATE (cache C, set S, way W)
```

Note that not all microarchitecture data structures have the associative (set,way) structure. We

might generalize the above as

```
FOR E OVER ALL entries in hardware data structure HDS
    INVALIDATE (hardware data structure HDS, entry E)
```

If multiple hardware data structures need to be flushed or invalidated one might do something like the following

```
 FOR H OVER ALL hardware data structures that we wish to invalidate
   FOR E OVER ALL entries in hardware data structure HDS
      INVALIDATE (hardware data structure H, entry E)
```

Without loss of generality we will assume that if a hardware data structure has an O(1) bulk invalidate, that it is handle as above, e.g. that the "entry" for the purposes of invalidation will be the entire hardware data structure. Similarly, some hardware data structures might invalidate for entries, e.g. all of the lines in a cache set, at once.

Portable code might be able to determine what hardware data structures it needs to invalidate by inspecting a ***system description such as CPUID or config string***. However, it may be necessary to invalidate the hardware data structures e.g. caches in a particular order. E.g. on a system with no cache coherence, not even hierarchical, it may be necessary to flush dirty data first from the L1 to the L2, then from the L2 to the L3, … ultimately to memory.

4.3.2. CMO.UR on non-strictly inclusive cache levels may not be able to guarantee completion flushes or invalidation

It is expected that typical implementations will iterate over a single cache level. Strict inclusion with backwards invalidation may provide the effect of invalidating all inner levels of of memory hierarchy.

However, it is very common for cache hierarchies NOT to be strictly inclusive. Examples include:

- Strictly exclusive caches
- Intel P6’s "accidentally inclusive" L2$
  - fills allocated in both L1$ and L2$, but L2$ may evict without backwards invalidatimng L1$.
  - Snoops probe both.
- ARM’s pseudo-inclusive and pseudo-exclusive caches.

CMO.UR operations for non-strictly inclusive cache levels may not be able to guarantee that a cache level has been completely flushed or invalidated. E.g. a line may be in the inner exclusive cache when the outer is scanned, and vice versa.

Implementations may provide mechanisms to permit complete invalidation and flushes. E.g. performing the CMO in a no-fill cache mode.\ However, such special cache modes are NOT included in this proposal.

This consideration applues to CMO.UR operations.

CMO.VAR implementations are, however, required to guarantee that all addresses in the range specified have been affected by the CMO.

4.3.3. CMO.UR implementations may iterate over multiple cache levels

It is expected that typical implementations of CMO.UR will iterate over a single cache level. Strict inclusion with backwards invalidation may provide the effect of invalidating all inner levels of of memory hierarchy. Whether such strict inclusion exists, or whether it is implemented by an actual cache layer, or by mechanisms such as inclusive snopp filter without data, are implementation dependent.

Furthermore, implementations may iterate over multiple caches and cache levels, by mapping several such caches into the same inded space. However, this must be done withing the constraints of the abstract cache model in .<cmo_specifier>.<wwhich_cache>

4.4. CMO.UR indexes should not be created out of thin air

Software invoking CMO.UR should not create arbitrary CMO UR indexes "out of thin air".

The index values should only be as obtained from the ***CMO.UR loop construct***, except for the starting value, -1.

reg_for_cmo_index := 1<<(XLEN-1) - 1 // maximum positive signed integer LOOP CMO.UR RD:reg_for_cmo_index, RS1:reg_for_cmo_handle UNTIL reg_for_cmo_index <= 0

Invoking CMO.UR with input register (RD) index values that were not as obtained from the sequence above is undefined.

- Obviously, if invoked from user code there must be no security flaw. Similarly, if executed by a guest OS on top of a hypervisor.
- It is permissible for an implementation to ignore CMO UR index values that are incompatible with the ***CMO descriptor***

If the software executing the ***CMO loop construct*** performs its own skipping of CMO UR indexes, the effect is undefined (although obviously required to remain secure). In particular, it cannot be guaranteed that any or all of the work required to be done by the ***CMO.UR loop construct*** will have been completed.

NOTE	the loop construct can be interrupted and restarted from scratch. There is no requirement that the loop construct be completed.

A thread might migrate from one CPU to another while the CMO loop construct is in progress. If this is done it is the responsibility of the system performing the migration to ensure that the desired semantics are obtained. For example, the code that is being migrated might be restricted to only apply to cache levels common to all processors migrated across. Or similarly the runtime performing the migration might be required to ensure that all necessary caches are consistent. *_(see issue)	ISSUE: process migration argues for whole cache invalidation operations and against the partial progress loop construct_*.

ISSUE: should it be legal for software to save the indexes from a first traversal of this loop and replay them later?

- Certainly not if the operation as specified by the ***CMO descriptor*** is different from that for which the indexes were obtained.
- I would like to make it illegal overall, but I can't CNP practical way to do this.

NOTE

Obsolete: Error reporting using final value of RS1.

Earlier versions of this proposal returned final values of RS1 other than -1 to indicate errors. This is no longer proposed.

•

5. CMO operation types: .<cmo_specifier>

TBD: include spreadsheet of encodings?

In addition to the different CMO instruction formats such as CMO.VAR and CMO.UR discussed above there are *many* types of CMO operations. The CMO types are represented in the assembly syntax as the .<cmo_specifier> field. They are encoded in the instruction encoding in the Funct7 field of the instruction encoding, in conjunction with the lowest numbered bit of Funct3, bit 11 of the instruction encoding.

These instruction types are formed by the the combination of

- which caches the operation applies to (and/or other parts of the memory system) - .<cmo_specifier>.<which_cache>
- what operation is actually performed (e.g. invalidate, flush dirty data) - .<cmo_specifier>.<cmo_operation>
- other aspects, such as invalidating related prefetchers and predictors .<cmo_specifier>.<cmo_other>

The subcomponents .<which_cache>, .<cmo_operation> and .<cmo_other> are NOT orthogonal

bitfields of the .<cmo_specifier> bitset formed by Funct7 and Funct3.0/11. Nevertheless, it is convenient to use the .<cmo_specifier>.<property> notation, to describe these subcomponent properties that are computed from irregular encodings.

CSR bitfields would be less tightly encoded than instruction bitfields

<cmo_specifier> might be specified quite simply in a CSR with 64 bits as follows;

- standard or implementation dependent: 1 bit
- CMO operation: 5 bits
  - e.g. FLUSH, CLEAN, DISCARD, PREFETCH.W/R, ...
  - with room for innovation
- From domain: 5 bits
- To domain: 5 bits
- Security: 1 bit - flush predictors and otheer timi8ng channel related state
- Mandatory/Advisory: 1 bit - HW is permitted to ignore, or not

NOTE This encoding occupies 18 bits, much more than the 128-256 reasonable to place in an instruction encoding.

Such a specification has encodings reserved for future instruction extensions.

The biggest consumker of bits, however, are the from-domains and to-domains.

E.g. for third party remote cache operations: hart1 performing a CMO that prefetches data from hardt2's L4 cache and moves it to hart's L2 cache. Even 5 bits is conservative, allowing only 32 distinct caches.

E.g. for prefetch instructions that fetch into level N, bit do not prefetch past level M, since the interconnect past that level is saturated.

However, since this proposal places the .<cmo_specifier> in the instruction encoding, the CMO types must be restricted and more tightly encoded.

5.1. Actual CMO operations .<cmo_specifier>.<cmo_operation>

5.1.1. Actual CMO operations- flushes and prefetches, etc.

This proposal includes the following actual CMO operations. Short names are listed here - more cvomplete deascriptions in a section below.

- Traditional CMOs: CLEAN, FLUSH, INVALIDATE-I$, DISCARD
- Less Common: INVALIDATE-CLEAN, SET-LRU, LOCK-LINE.

Space should be reserved for more operations, included SAFER_DISCARD_1 and SAFER_DISCARD_2, that remedy the security deficiences of the DISCARD operation (the well known PowerPC DCBA)

while preserving much of the performance advantage.

In addition to these CBOs that perform various forms of flushes and invalidates, this proposal includes operations that are often not called CMOs.

- Prefetches: PREFETCH-R, PREFETCH-EW, PREFETCH-X - using the variable address range approach.
- Destructive: ZALLOC - allocate a zero-filled-cache line.

Some have requested locking versions: ZALLOC-and-LOCK, and FETCH-R/W/X-and-LOCK.

COUNT: 13 encodings: 4 bits.

5.1.2. Security / Timing Channel Bit

Requirement: in addition to flushing caches, it is also required, for timing channel mitigation such as in Spectre, to flush microarchitecture mechanisms that can provide timing channekls, such as LRU bits, predictors and prefetchers. Some of these are associated with cache entries - hence the security/timing channel "bit". Not actually a bit - applied only to 2 CMOs.

The security property is applied to the CMO.UR variants that leave no data behind: FLUSH and INVALIDATE.

This increases the ***COUNT*** to 15 encodings: 4 bits.

5.1.3. Detailed description of CMO operations

Unfortunately, there is no widespread agreement as to what CMO names should be. It is therefore necessary to define their behavior more completely according to cache states.

Without loss of generality we will mention only tywo cache states, Clean and Dirty, relevant to writeback caches. Writethrough and instruction caches contain only clean data, so may map to more than one operation that handles dirty data.

Traditi0nal CMOs

- CLEAN
  - Dirty-→WB-→Clean
  - Clean-→Clean
- FLUSH
  - Dirty-→WB-→Invalid
  - Clean-→Invalid
  - Alternate names
  - Intel calls this WBINVD
  - Special considerations: security/timing channel variant for CMO.UR
- DISCARD

    - Dirty-→no WB-→Invalid
    - Clean-→Invalid
    - Alternate names
    - Intel calls this INVD
    - Special considerations:
        - security/timing channel variant for CMO.UR
        - security hole
            - there are several safedr variants of DISCARD, reserving space for bit not actually part of this proposal
- DISCARD-CLEAN
    - Dirty-→unaffected
    - Clean-→Invalid
    - Special considerations:
        - can be used in some incoherehnt I/O use cases
        - remedies the security problems of DISCARD - safe for user mode
- SET-LRU
    - CMO.VAR only
    - most useful sp[ecial case of the class of replacement algorithm manipulation CMOs

Operations not typically considered CMOs:

- PREFETCH-R
- PREFETCH-W
    - prefetches in exclusive clean or dirty state - ready for writes with least possible expense
- PREFETCH-X
    - prefetch code, to execute
    - like PREFETCH-R, except targetting I$ level(s)

Destructive

- ZALLOC
    - allocate cache line with reading - zero filling
    - PowerPC DCBZ
- ALLOC
    - allocate cache line with reading - using whatever was there before
    - security hole - but still sometimes used
    - PowerPC DCBA

Locking variants of the above * FETCH-R-and-LOCK * FETCH-W-and-LOCK * FETCH-X-and-LOCK * ZALLOC-and-LOCK * ALLOC-and-LOCK

Count: 15 operations - 4 bits

5.2. CMO memory hierarchy domains and levels .<cmo_specifier>.<which_cache>

The .<which_cache> property specified the domains and levels involved in CMO operations.

"Domains" refers to CMOs that flush data from not just one cache, but from severral layers of cache. Sometimes by flushing an outer inclusive layer. Sometimes by traversing multiple levels.

Actual implementations may have many idiosyncratic caches and other parts of the memory hierarchy.

There should bne a standard RISC-V way to flush such non-standard implementation specific cache levels, but that is not part of this proposal.

Instead this proposal defines a small(?) number of abstract cache layers. Implementation cache layers will be mapped onto these layers.

These pseudo-abstract layers are

Cache levels and domains

- POC(I,D)
  - the Point of Consistency for Instructions and Data, for the common case of inconsistent instruction and data caches
  - ARM calls this the Point of Unification
- The POC(ID) defibnes two domains that may need to be flushed
  - I-→POC(ID) - the path from processor through I$ to the Point of ID consistency
  - D-→POC(ID) - the path from processor through D$ to the Point of ID consistency
- POC(D*), domain P*-→POC(D*)
  - the path from any or all of a set of processors to the common level for all processors in that set.
  - ARM calls this the Point of Inner Comsistency
  - assumed cache coherent in this domain
  - used for performance optimizations, not correctness
- POC(Unc), domain P*-→POC(Unc)
  - the path from any or all of a set of non-cache-coherent processors to a common point
  - SW managed consistency works if this domain is flushed to POC(Unc)
- POC(Uio), domain P*-→POC(Uio)

    - the path from any or all of a set of non-cache-coherent processors to a point in common with non-coherent I/O
    - SW managed consistency for I/O devices works if this domain is flushed to POC(Uio)

Frequently, POC(Unc), POC(Uio) are identical. Frequently, POC(Unc), POC(Uio) are DRAM. But not always, therefore distinguished.

Memory, Volatile and Non-Volatile

- M, domain P*-→M
    - memory, eg DRAM
    - not necessarily battery backed up
- BM, domain P*-→BM
    - memory that survives power removal from system parts such as harts
    - frequently the same as main memory, bit not always. May be a subset.
- NV1, domain P*-→MN/BM-→NV
    - memory that survives even when batteries fail
        - i.e,. last years, not days
- NVR, domain P*-→MN/BM-→NV-→NVR
    - a;;, last, or redundant/reliable level of nonvolatile memory
    - memory that tolerates failures of other NV1 components
        - i.e,. last years, not days

Count: 9 - 4 bits

Unfortunately, would like local/global flavors of the above. ⇒ 5 bits ⇒ exceeds 256 emncodimngs.

So need to compress more.

TBD…

5.3. CMO type spreadsheet

A spreadsheet CMOs.xlsx presents the desired CMO types in a format more convenient than this asciidoc text.

This spreadsheet is available on GitHub at https://github.com/AndyGlew/Ri5-stuff/blob/master/CMOs.xlsx

TBD: ensure that the latest version of the spreadsheet has been uploaded. As of <2020-06-11 Thursday, June 11, WW24> the version online is dated April 30th.

-

6. Considerations common to CMO instruction formats

6.1. *Source/dest* to support *exception transparency*

This instruction family is ***restartable after partial completion***. E.g. on an exception such as a page fault or debug address breakpoint the output register RD is set to the data address of the exception, and since the instruction is ***source/dest***, with the register numbers in RD and RS1 required to be the same, returning from the exception to the CMO.UR instruction will pick up execution where it left off.

NOTE

Rationale: source/dest by requiring RD=RS1

This proposal has chosen to implement ***source/dest*** by requiring separate register fields RD and RS1 to contain the same value. An alternative was to make register field RD both an input and an output, allowing RS1 and RS2 to be used for other inputs. Separate RD=RS1 source/dest is more natural for a RISC instruction decoder, and detecting RD=RS1 has already been performed for other RISC-V instructions, e.g. in the V extension. However separate RD=RS1 "wastes" instruction encodings by making RD!=RS1 illegal, and leaves no register free in the CMO.VAR instruction format for any 3rd operand such as the CMO type, hence requiring .<cmo-specifier> in the instruction encoding.

TBD: see ***who cares about RD=RS1 source/dest?***

6.2. Privilege for CMOs

6.2.1. SUMMARY: Privilege for CMOs and Prefetches

Each of the prefetches and CMOs, including the fixed block size prefetches PREFETCH.64B.* and CMOs CMO.64B.**, and the address range CMOs CMO.VAR.** and cache index CMOs CMO.UR.* are mapped to a number 0..Ncmo-1, where Ncmo is the Number of CMO instruction encodings.

(Note: the encodings do not necessarily have a contiguous field that corresponds to these values.)

Several CSRs CMO-Privilege contains Ncmo 2-bit fields where bitfield CMO_Privilege.2b[J] indicates the privilege required to perform the corresponding CMO operation J. More than one CSR is required, since there are more than 64/2=32 different CMO flavors, each of which has a separate 2-bit delegation field. E.g. if tyeere are 256 CMOs ⇒ 512 2-bit fields, 8 64-bit delegation CSRs.

TBD: see elsewhere for exactly how many CMOs are provided.

The 2-bit fields are encoded as follows:

- 00 ⇒ disabled.
- 01 ⇒ traps to M mode

    - TBD: exception info (cause, address)
- 10 ⇒ reserved
- 11 ⇒ can execute in any mode, including user mode

6.2.1.1. Disabling CMOs - almost but not quite a NOP

The disabled behavior for CMO.VAR is as follows:

CMO_Privilege.2[J] ⇒ CMO.#J

- the instruction does not actually perform any cache maintenance operation.
- but it returns a value such that the ***canonical range CMO loop*** exits
- CMO.VAR rd:next_addr, rs1=rd:start_addr, rs2:stop_addr
- sets RD to stop_addr
- CMO.UR rd:next_entry, rs1:start_entry
- sets RD to -1

I.e. CMO.VAR may *disable*, i.e. not perform, any cache management operation, but must still write a value to RD guaranteeing that the surrounding software loop wuill terminate.

CMO.UR, on the other hand, may actually be a NOP, since its start and ultimate stop indexes are both -1.

NOTE

TBD:

Q: how should we arrange to permit the OS to perform certain CMOs, while allowing some CMOs to be executed by user mode, and otyhers to be implemented by hypervisor or M-mode? Should we provided separate CMO-Privilege CSR sets for each privilege level? (As is being proposed for cerrtain other RISC-V extensions, such as Pointer Tagging.)

A: not currently proposed. M-mode can emulate delegation if this is necessary. At least the cost of bouncing through M-mode will be amortized because of the range nature of the CMOs.

6.2.1.2. Context Switch

System software such as an operating system may provide the same values of the CMO_Privilege CSRs to all processes. If so, no context switch of the CMO-Privilege CSRs is required.

However, if the OS provides different values to different processes, then the CMO-Privilege CSRs must be context switched.

6.2.1.3. Unimplemented and Cross Wired CMOs

Although there may be as many as 256 CMOs architected, an implementation need not build all of them.

An implementation may leave some CMOs unimplemented. If so, then the CMO-Privilege field for those CMOs is hardwired to 00, corresponding to CMO disabled. Such fields are WARL, any write value permitted, but only the disabled value 0 is read. This can be used to discover which CMOs are implemented. (Not which cache levels are implemented - that must be discovered using some other mechanism, not defined here.)

NOTE

TBD: CMO Discovery

Q: should it be possible for user code to discover which CMOs are implemented? That would require the CMO-Privilege CSRs to be readable but not writeable from user mode. But that would be a security hole, so would require a delegation field to control which privilege levels can read (and write?) the CMO-Privilege CSRs.

Certain CMOs may be equivalent.

For example, certain of the "abstract" domains and levels of the memory hierarchy expressed by the .<cmo_specifier>.<which_cache> field may be identical in some implementatuions but not others. E.g. the point of coherence for instructions and data, POC(I,D) (called the Point Of Unification in ARM terminology), may also be the Point of (Inner) Consistency for all harts. E.g. there all DRAM may be battery backed up. E.g. there may be a single level of consistency for both non-coherent I/O and processors - typically this is also DRAM, although it may be a cache level if I/O injection is supported.

Similarly, certain abstract CMOs may be equivalent. E.g. ona system with writethrough caches, CLEAN and FLUSH may be treated equivalently.

The CMO delegation fields for equivalent CMOs *must* be cross wired, so that writes in one position appear in all equivalent positions.

NOTE

TBD

Q: should there be a discovery mechanism for equivalent CMOs? A: strictly speaking not needed, since can determine by a test pattern. However, that can be expensive.

NOTE

Rationale: Privilege for CMOs and Prefetches

Requirement: in some CPU implementations all or some CMOs **must** be trapped to M-mode and emulated. E.g. caches that require MMIOs or CSR actions to flush, which are not directly connected to

Requirement: in some platform configurations some CMOs may **optionally** be trapped to M-mode and emulated. E.g. ***CMOs involving idiosyncratic external caches and devices***, devices that use MMIOs or CSRs to perform CMOs, and which are not (yet?) directly connected to whatever

Requirement: it is highly desirable to be able to perform CMOs in user mode. E.g. for performance. But also for security, persistence, since everywhere the ***Principle of Least Privilege*** should apply: e.g. the cache management may be performed by a privileged user process, i.e. a process that is part of the operating system but which is running at reduced privilege. In such a system the operating system or hypervisor may choose to context switch the CSR_Privilege CSR, or bitfields therein.

Requirement: even though it is highly desirable to be able to perform CMOs in user mode, in some situations allowing arbitrary user mode code to perform CMOs is a security vulnerability. vulnerability possibilities include: information leaks, denial of service, and facilitating RowHammer attacks.

Requirement: many CMOs should be permitted to user code, e.g. flush dirty data, since they do nothing that user code cannot itself do using ordinary load and store instructions. Such CMOs are typically advisory or performance related. note that doing this using ordinary load and store instructions might require detailed microarchitecture knowledge, or might be unreliable in the presence of speculation that can affect things like LRU bits.

Requirement: some CMOs should **not** be permitted to user code. E.g. discard or forget dirty data without writing it back. This is a security vulnerability in most situations. (But not all - although the situations in which it is not a security vulnerability are quite rare, e.g. certain varieties of supercomputers, although possibly also privileged software, parts of the OS, running in user mode.)

Requirement: some CMOs may usefully be disabled.

- Typically performance related CMOs, such as flushing to a shared cache level, or prefetching using the range CMOs Software is notorious for thinking that it knows the best thing to do,
- Also possibly software based on assumptions that do not apply to the current system
- e.g. system software may be written so that it can work with incoherent MMIO but may be running on a system that has coherent MMIO
- e.g. persistence software written so that it can work with limited nonvolatile storage running on a system where all memory is nonvolatile

Requirement: Sometimes there needs to be a mapping between the CMO that a user wants and the CMOs that hardware provides, where the mapping is not known to CPU hardware, not known to user code, but depends on the operating system and/or runtime, and might dynamically depend on the operating system and/or runtime.

- e.g. For performance related CMOs, the user may only know that she wants to flush whatever caches are smaller than a particular size like 32K. The user does not know which caches those are on a particular system.
- e.g. in software coherence all dirty data written by the sending process P_producer may need to be flushed to a shared cache level so that it can be read by the consuming process P_consumer
- consider if the sending process P_producer is part of a HW coherent cache consistency domain, but the receiving process P_consumer is part of a different such domain
- if the hardware cache consistency domain permits cache-to-cache migration of dirty data, then all caches in that dirty domain be flushed.

> - however, if the hardware cache consistency domain does NOT permit cache-to-cache migration, then
> - if the system software performs thread or process migration between CPUs that do not share caches
> - without cache flushes ⇒ THEN this SW dirty domain must be flushed
> - but if the system software performs cache flushes on thread migration, ⇒ THEN only the local processor cache need be flushed.
> - if the system software does not perform thread or process migration, t hen only the local processor cache be flushed. Other processor caches in the HW clean consistency domain do not need to be flushed.
>
> Optionally trapping such CMOs allows the system or runtime software to choose the most appropriate hardware CMO for the users' need.
>
> **I.e. the mapping is done by SW in the trap handler**

•

Appendix A: Techpubs Information

A.1. Conventions specific to this document.

Bold italic ***links*** indicate text that should be links to pages in the original wiki. The tools used to generate this document HTML and PDF from asciidoc and markdown do not handle these links (yet).

A.2. Techpubs Information

This source document: Ri5-CMOs-proposal.asciidoc

- docdatetime: 2020-06-11 07:59:10 -0700 - last modified date and time
  - unfortunately, this is only for the topmost file, NOT across all of the included files
- localdatetime: 2020-06-11 11:21:15 -0700 - when generated

Revisions - manually maintained, frequently obsolete:

- revdate: June 6, 2020
- revnumber: 0.3
- revremark: Fixed Block Prefetches + CMOs, + Privilege for CMOs, + CMO.VAR and CMO.UR

More techpubs information, including history thrashing as to how and where to build and store, on wiki page techpub / file:techpubs.asciidoc (TBD: fix so that works both checked out as file: links and on GitHub wiki).

You may be reading this in any of several different places:

- On GitHub
  - This document's source files (mostly asciidoc) on its Github repository wiki:
    - https://github.com/AndyGlew/Ri5-stuff/wiki
    - top of document: https://github.com/AndyGlew/Ri5-stuff/wiki/Ri5-CMOs-proposal
      - this is a .asciidoc file, rendered by GitHub's wiki
      - asciidoc includes link to other parts of the document
      - the wiki contains other pages, not part of the document, some of which provide more background
  - Generated HTML and PDF files on GitHub:
    - https://github.com/AndyGlew/Ri5-stuff/
      - HTML: https://github.com/AndyGlew/Ri5-stuff/blob/master/Ri5-CMOs-proposal.html
        - displays raw, does not render
      - PDF: https://github.com/AndyGlew/Ri5-stuff/blob/master/Ri5-CMOs-proposal.pdf
        - displays - in GitHub's ugly way
    - https://github.com/AndyGlew/Ri5-stuff/wiki
      - PDF: https://github.com/AndyGlew/Ri5-stuff/wiki/Ri5-CMOs-proposal.pdf
        - downloads, does not display
      - HTML: https://github.com/AndyGlew/Ri5-stuff/wiki/Ri5-CMOs-proposal.html
        - displays raw, does not render
- On your local system, where you may have cloned the GitHub parent and wiki repositories, and where you may have built the document:
  - local where built:
    - won't work from web
    - file:Ri5-CMOs-proposal.html
    - file:Ri5-CMOs-proposal.pdf
    - file:Ri5-CMOs-proposal.asciidoc

When and where converted (i.e. when asciidoctor was run, to generate this file):

- docfile: /cygdrive/c/Users/glew/Documents/GitHub/Ri5-stuff/Ri5-stuff.wiki/Ri5-CMOs-proposal.asciidoc - full path
- localdatetime: 2020-06-11 11:21:15 -0700 - when generated
- outfile: /cygdrive/c/Users/glew/Documents/GitHub/Ri5-stuff/Ri5-stuff.wiki/Ri5-CMOs-proposal.pdf - full path of the output file
- TBD: what system (PC, Linux system) was asciidoctor run on?

Note: paths local to system document generated on are mostly meaningless to others, but have

already been helpful finding source for orphaned drafts generated as PDF and HTML.