Ri5 CMOs proposal: Cache Management Operations

2020-06-02 13:58:19 -0700 - When this draft was generated. Not necessarily modified. See Techpubs Information section below for more details.

1. About this document

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

.

Table of Contents

2. CMO instruction formats and CMO operation types

There are 3 formats of CMO instructions: * Fixed Block Size Prefetches and CMOs operating on 64B naturally aligned regiions of memory * Variable Address Range CMOs operating on arbitrary address ranges * Microarchitecture Structure Range CMOs 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)

-

3. CMO instruction formats

4. Fixed Block Size Prefetches and CMOs

4.1. SUMMARY: Fixed Block Size Prefetches and CMOs

4.1.1. 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

4.1.2. Fixed Block Size Clean and Flush CMOs

Proposed name: CMO.64B.CLEAN.toL2

- more descriptive name: D1-Clean-to-L2 64B
- OR format with RD=R0, RS2=R0

    - funct7=0000000.rs2=00000.rs1:5.110.rd=00000.0110011
- affects cache line containing virtual address M[rs1]
- "clean"
    - write dirty back,
    - keep clean copy of all lines in cache, both originally dirty and clean
- see ***Mnemonics and Names*** for a discussion of proposed mnemonics and names

Proposed name: CMO.64B.FLUSH.toL2

- more descriptive name: D1-Flush-to-L2 64B
- encoding: AND format with RD=R0, RS2=R0
    - funct7=0000000.rs2=00000.rs1:5.111.rd=00000.0110011
- affects cache line containing virtual address M[rs1]
- "flush"
    - write dirty back,
    - invalidate all lines in cache, both originally dirty and clean
- see ***Mnemonics and Names*** for a discussion of proposed mnemonics and names

The more descriptive names "D1-Clean-to-L2" and "D1-Flush-to-L2" are more descriptive of the implementation & intent on a typical system at the time this is being written. The proposed names such as CMO.64B.FLUSH.toL2 are more generic, and may apply when the cache hierarchy is different. (Obviously "toL2" is microarchitecture specific, and should be replaced by something like "SHARED-LEVEL".) See ***Mnemonics and Names*** for a discussion of proposed mnemonics and names.

The intent is that dirty data be flushed to some cache level common or shared between all or most processors of interest. E.g. if all processors share the L2, flush their L1s to the L2. If all processors share and L3, then flush their L1s and L2s to the L3. And so on. Obviously, exactly what level flushes done to depends on the cache hierarchy and platform.

(More precise control is found in the variable address range CMOs. We do not want to spend all of the increasingly scarce instruction encodings to encode all hypothetically desirable prefetches and CMOs in the instruction format that touches Mem[reg+imm12]. Some other instructions use register operands to allow more prefetch and CMO types.)

4.2. DETAILS

4.2.1. The Usual 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

4.2.2. Rationale

4.2.2.1. Addressing Modes

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

CMO clean/flushes with full Offset addressing mode would be nice to have, but consumes encoding space.

4.2.2.2. Fixed minimum block size - NOT cache line size

These instructions are associated with a fixed block size - actually a minimum fixed block size. NOT 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 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.

5. 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.

5.1. SUMMARY: 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

5.1.1. 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`.

5.1.2. 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].

5.1.3. COMMON: 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.

5.2. DETAILS

5.2.1. 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.

5.2.2. COMMON+CMO.VAR: Possible implementations 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.

5.2.2.1. COMMON+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

RATIONALE: 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.

5.2.2.2. RATIONALE: 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 $_{\sim}$~ 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) $_{\sim}$~

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,

5.2.3. COMMON+some specific: 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

5.2.4. 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.

5.2.5. COMMON: Permissions for CMOs

5.2.5.1. 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

5.2.5.2. COMMON: 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.

6. 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
```

6.1. 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
```

6.2. DETAILS

6.2.1. 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].

6.2.2. CMO-COMMON: *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.

6.2.3. CMO-COMMON: 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***.

6.2.4. CMO-COMMON: CMO-OPERATIONS: *Actual CMO Operations*

6.2.4.1. CMO-COMMON: 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.)

6.2.5. 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

6.2.6. CMO-COMMON: 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.

6.2.7. Permissions for CMOs

6.2.7.1. 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)

6.2.7.2. CMO-COMMON + CMO.UR: *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***.

6.2.8. 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

6.3. OBSOLETE: REMOVE: *CMO descriptor* - what is affected

OR, AT LEAST NOT PART OF MAIN PROPOSSAL

See ***CMO descriptor*** for an explanation of how the CMO instructions, and this instruction CMO.UR in particular, specify which caches and branch predictors and other microarchitecture state should be managed.

Suffice it to say that a single CMO.UR instruction is able to perform invalidations and/or cache flushes of multiple caches in the same CMO.UR loop construct. This is accomplished by the ***CMO UR descriptor operand***

6.4. *CMO UR index*

6.4.1. 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.

6.4.2. OBSOLETE - DISCUSSION - CMO.UR abstracts and unifies microarchitecture cache invalidation loops

CMO.UR and the ***CMO UR index*** abstract such microarchitecture dependencies as follows:

There is no need to loop over various caches and other hardware data structures. The ***CMO.UR loop construct*** loops over all hardware data structures and all entries in those hardware data structures as needed.

```
reg_for_cmo_index := 0   // maximum positive signed integer
LOOP
   CMO.UR RD:reg_for_cmo_index, RS1:reg_for_cmo_handle
UNTIL reg_for_cmo_index < 0
```

For any particular CMO (as specified by a ***CMO descriptor***) all of the various hardware data structures, caches are combined into the same ***CMO UR index*** space, which is a subset of the positive integers of length XLEN.

TBD: move this example into a separate page.

For example:

CMO UR index	Description
1<<(XLEN-1)-1	maximum positive integer, RD input value at start of CMO loop construct

CMO UR index	Description
	unused - CMO.UR skips over
L2$	16-way 64M L2 D$ / 64B lines => 2^20 = 1M entries
1FFFFF	L2$ entry #1M-1, i.e. set #16K-1, way #15
…	… other L2$ entries …
1.0000	L2$ entry #0, i.e. set #0, way #0
	unused - CMO.UR skips over
L1 D$	8-way 32K L1 D$ / 64B lines => 2^9 = 512 entries
21FF	D$ entry #511, i.e. set #63, way #7
…	… other D$ entries …
2001	D$ entry #1, i.e. set #0, way #1
2000	D$ entry #0, i.e. set #0, way #0
	unused - CMO.UR skips over
L1 I$	4-way 16K L1 I$ / 64B lines => 2^8 = 256 entries 1000H
10FF	I$ entry #255, i.e. set #63, way #3
…	… other I$ entries …
1001	I$ entry #1, i.e. set #0, way #1
1000	I$ entry #0, i.e. set #0, way #0
	unused - CMO.UR skips over
0	final value if CMO.UR completes successfully
<0	reserved for error reporting

I.e. each hardware data structure is allocated a range of indexes in the ***CMO UR index*** space. The ***CMO.UR loop construct*** loops over the indexes from largest possible to smallest. Therefore, the order of the flushes or invalidations is implied by the index range allocation.

```
reg_for_cmo_index := 0
LOOP
   CMO.UR RD:reg_for_cmo_index, RS1:reg_for_cmo_handle
UNTIL reg_for_cmo_index < 0
```

The mapping of hardware data structures and their entries into the ***CMO UR index*** space is microarchitecture and implementation specific. Portable software should not rely on any such mapping.

The invalidation or flushes desired by any particular cache management operation, as specified by its ***CMO descriptor***, will probably not involve touching all possible ***CMO UR index***. Therefore, the

CMO UR index traversal for any particular ***CMO descriptor*** may skip - e.g. in the above example, if one desires only to invalidate the instruction cache, one would skip the ranges for the L2 in the L1 data cache. An alternate implementation would be to create a different mappings of CMO UR indexes to hardware data structure entries for different CMOs.

At the very least, CMO.UR will skip from the starting value 1<<(XLEN-1)-1 to the largest ***CMO UR index*** value defined for the requested operation. It is also expected to skip CMO UR index values when the hardware data structures are not allocated contiguously within the CMO UR index space.

6.5. KEEP: 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.

6.6. KEEP, BUT MOVE OUT OF PROPOSAL: Errors during CMO.UR

CMO.UR’s output register RD is set to -1 on successful completion of the operation.

Negative values for CMO.UR’s output register RD are reserved to indicate possible errors. At this time no such errors are defined. Any errors encountered while performing CMO.UR (e.g. on correctable errors in a cache tag) should be reported using normal ***RISC-V hardware error reporting*** mechanisms (e.g. ***machine check*** exceptions).

Rationale: it is probably best for CMO.UR to use normal ***RISC-V hardware error reporting*** mechanisms, such as immediate and/or deferred machine check exceptions, and/or recording error status in CSRs. However, it is possible that some CMOs may be used in situations where normal error reporting is either not available or is inconvenient. It is straightforward to reserve negative values to indicate possible errors when the instruction is created. It would not be possible to retroactively change the definition of the CMO.UR instruction to do such error reporting if there were no such reserved values.

NOTE	unfortunately, such error reporting can not be performed for the memory address range based CMOs, CMO.UR, since their RD return value can assume all permissible values as it is an excellent address.

NOTE	if we were using a counter based mechanism for CMO.VAR, we **could** use negative return values for errors.

-

7. CMO operation types

TBD: list of CMO operation tuypes

like CLEAN, FLUSH, DISCARD

anstract cache levels

and instruction encodings

TBD: include spreadsheet of encodings?

-

8. Considerations common to CMO instruction formats

8.1. CMO-COMMON: *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.

RATIONALE: 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?***

8.2. Privilege for CMOs

8.2.1. 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.)

CSR CMO-Privilege contains Ncmo 2-bit fields where bitfield CMO_Privilege.2b[J] indicates the privilege required to perform the corresponding CMO operation J.

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

The disabled behavior 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

8.2.2. 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 <i>dynamically</i> 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*** surroundindicate 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)g.

A.2. Techpubs Information

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.

This source document: Ri5-CMOs-proposal.asciidoc

- docdatetime: 2020-06-02 13:22:20 -0700 - last modified date and time
- docfile: /cygdrive/c/Users/glew/Documents/GitHub/Ri5-stuff/Ri5-stuff.wiki/Ri5-CMOs-proposal.asciidoc - full path

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

- localdatetime: 2020-06-02 13:58:19 -0700
- 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?

Revisions - manually maintained, frequently obsolete:

- revdate: + Privilege for CMOs, + CMO.VAR and CMO.UR
- revnumber: 0.3: Fixed Block Prefetches + CMOs
- revremark: {revremark}

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

Generated files:

- TBD: need to create conditional links that direct to different places based on viewing environment
    - until then …
- local where built:
    - won’t work from web
    - file:Ri5-CMOs-proposal.html
    - file:Ri5-CMOs-proposal.pdf
- GitHub:
    - note: inconsistent unless viewed on web

- 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, not HTML