Ri5 CMOs proposal: Cache Management Operations

Table of Contents

Top level last modified: 2020-05-13 21:54:15 -0700 - top level (TBD: included subdocuments). See Techpubs Information section below for more details.

About this document

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

1. Fixed Block Size Prefetches and CMOs

1.1. SUMMARY: Fixed Block Size Prefetches and CMOs

1.1.1. Fixed Block Size Prefetches

Proposed name: PREFETCH.64B.R

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

1.1.2. 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
- 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
- AND format with RD=R0, RS2=R0
- 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.)

1.2. DETAILS

1.2.1. Fixed minimum block size - NOT cache line size

These instructions are defined to be associated with a 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.

1.2.2. 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

1.2.3. Options, Options, All around! - JUST SAY NO

Above we have mentioned three arbitrary decisions: * take/don’t take page faults * take/don’t take debug exceptions * take/don’t take ECC or other machine check error exceptions

While the policies we suggest seem reasonable, cases can be made for taking each of the other alternatives.

At this time we do NOT suggest making global CSR bits for these policies. I.e., not global CSR bits for these policies that apply to all prefetches and CMOs. But policies that apply to particular prefetches and CMOs.

1.2.4. Alternatives

Other block sizes, "non-temporal hints", etc., fetch into L2 but not L1, etc. But 4 is a good start, assuming that the encodings are scarce.

CMO clean/flushes with full addressing mode. Nice to have, but consumes opcode space.

2. Privilege for CMOs

Actual proposal/draft asciidoc draft-Privilege-for-CMOs forked from discussion Privilege-for-CMOs.

2.1. PROPOSAL:

Each CMO.VR.* and CMO.UR.* is 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
- 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.VR 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

2.2. RATIONALE:

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

WHINING & HISTORY:

TBD: remove

- I had originally planned to define CSR operands for the CMO instructions, both to provide the privilege modulation (trapping, disabling) and mapping functionalitiess of the requirements listed above.
- key reviewers reject this possibility, and/or suggest providing it only later if the need is proven
- however, thesse key reviiewers CANNOT deny the requirements of enabling or disabling CMOs listed above
- therefore, providing this compact privilege mechanism.
- I am actually just as happy not to defiine the CSR operand to coontain an encoding of CMO operations desired, since I can easily imagine that in some circumstances more than one CSR will be required. E.g. a CSR that might contain a way mask. Therefore, this " permission vector" approach allows the actual CSR is to be defined later, while enabling [[privilege modulation]] today.

Appendix A: Techpubs Information

Note: paths local to system document generated on are mostly meaningless to others, but have already been helpful finding source for orphaned PDF and HTML.

This source document: Ri5-CMOs-proposal.asciidoc

- docdatetime: 2020-05-13 21:54:15 -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: 2020-05-13 22:00:05 -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: 2020-05-06
- revnumber: 0.0
- revremark: Setting up techpubs stuff - no content yet

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).