Instruction/Data (I/D) consistency proposal for RISC-V (aka Son of FENCE.I)

Derek Williams (IBM)
Grigorios Magklis (Esperanto)
Martin Maas (Google)
Paul Loewenstein (WDC Retired)
December 13th, 2020 – Version 0.9

DRAFT COPY ONLY – NOT COMPLETE - v0.9.1

First a disclaimer

- For at least one of the authors (Derek Williams), this represents personal opinions only and should not be construed as any sort of official position endorsed by IBM.
- Further, no information in this presentation is considered IBM Confidential, nor Western Digital Confidential, nor Google Confidential.
- This is an incomplete draft copy. This notice and the footers will be updated when this presentation reaches a full 1.0 version.

Acknowledgments.

- This presentation is not, in most ways, new or novel. It borrows many concepts and techniques that have existed in other architectures for many, many years. We didn't invent most of this.

- Additionally, we are grateful to several people (listed below and missing a few that wished to be anonymous) who commented on several early drafts in ways that corrected errors and helped clarified the exposition.

- Names (in particular order): Andrew Waterman (SiFive), Foivos Zakkak, Christos-efthymios Kotselidis (University of Manchester), John Ingalls (SiFive), Dan Lustig (Nvidia), Brian Grayson (SiFive), Ben Simner, Christopher Pulte, Peter Sewell (U. Cambridge), Andy Glew (SiFive), Paul Mackerras (IBM), Andy Glew (SiFive).

Change Log

- 12/13/2020 - Version 0.9
  - First public release.
- 12/14/2020 - Version 0.9.1
  - Fix overzealous edit of FENCE.ID section from 12/13/2020

Part I : Introduction.

Introduction

- The RISC-V ISA intends to support implementations with incoherent I-caches and/or post I-cache buffering and execution pipes containing fetched instructions.

- In order to support these microarchitectural features, a set of ISA instructions and mechanisms is needed to provide a means for software to bring the instruction fetching and execution in line with updates made to the instructions in memory.

- This is a “clean-sheet-of-paper” proposal that replaces FENCE.I and will eventually be its own extension that is independent of the zifence.i extension (and of the J extension).

- No attempt whatsoever is made to make this proposal “compatible” (whatever that would mean) with the existing FENCE.I (we don’t think that is even possible if one wants to handle incoherent I/D caches in any reasonable way).

Solution characteristics/requirements

- The instructions for the new mechanisms proposed here are user-level unprivileged instructions. This is necessary to support user-level threading libraries.

- Consisting of user level instructions, any sequence of instructions for managing I/D consistency have to tolerate being interrupted after any instruction and moved to a new HART.

- The mechanisms proposed here are intended to be OS/Hypervisor agnostic. In other words, everything one would need to know about I/D consistency handling can be found in the ISA description. There can be no hidden dependency/requirement on OS or Hypervisor level software (save that the OS/Hypervisor must use the ISA mechanisms correctly).*

  * A lofty goal not quite realized.. See the description of xRET later that will require an IMPORT.I to be placed in front of it in many cases.

Adjudicating this vs “cache control” instructions

- The new instructions (and clarifications) here should largely be adjudicated separately from any new “cache control” instructions for several reasons:
  - Only one of the two instructions proposed here (**EXPORT.I**) has any cache control implications at all.
  - This discussion should not be coupled any more than necessary to the larger, likely more contentious, and more lengthy discussion around the totality of the different cache control instructions.
  - These instructions must be unprivileged. The cache control instructions (or some of them) may wind up being privileged.
- There are some issues that are common between EXPORT.I and the other cache control instructions. These should be settled the same way for both. So, there is some dependency/interaction here between the conversations.

Common issues: EXPORT.I and cache ctrl instr.

- These are the known common issues for EXPORT.I and cache control ops. The options in red are what is assumed here for EXPORT.I (in some cases just for simplicity).

Issue	Choices	Notes:
How the region of memory affected is specified? **DECIDED**	• Single cache block: RS1 specifies memory location within a cache block.	The CMO TG has decided to initially go with SINGLE cache block style CMO instructions in the initial phase. The EXPORT.I (really the two instructions in part V, will be block based).
Are these considered as a load or a store for processing?	• Considered as a store • Considered as a load	This choice should not change. Cache control operations are like “stores” and should be run down the “store” pipes.
Protection faults?	• No permissions required. • User permission required. • Read permission required • Read/Write permission required • Execute permission required (with or without R/W).	Permissions for other cache control ops will vary, but EXPORT.I isn’t reading, writing, or executing, so why have a protection fault? Except for checking “user” access. **Clearly, a translation must exist or a page fault occurs even if the protection violatons are ignored.**
Intra-thread ordering with or without fences?	• Order based on treating operation as a store and by address collisions with loads or stores. • Require a fence to order operation relative to loads and stores within thread.	This one is open to change. If single block is used, it may make the most sense to do addr collisions. If ranges are decided and detecting intra-thread hazards is harder due to that, it may be prudent to require fences to accomplish this ordering.

A bit of a simplification: EXPORT.I

- EXPORT.I (informally) pushes any modified cache data for the block far enough down the cache hierarchy for the incoherent I-cache mechanism to pick it up the updates and then invalidates all I-cache copies of the block.

- Originally, EXPORT.I was intended to be one instruction for good reasons.

- Unfortunately, various even better reasons to split EXPORT.I into two distinct instructions were discovered later. This is explained and justified at length in part V below.

- I did not rewrite the full presentation to reflect this change. Until part V, this presentation sticks to the convenient fiction/shorthand that EXPORT.I is a single instruction. The full sequence of instructions that replace “EXPORT.I” and how to substitute those for an “EXPORT.I” is addressed in part V below.

Minutiae

- The names given here for the instructions (EXPORT.I/IMPORT.I/FENCE.ID) are simply placeholders. We don’t care what they get called (including those names), just what the instructions do.

- We care even less about which instruction opcodes the new instructions get assigned to. Wiser minds than ours can decide that.

- As stated above, now that the CMO TG has settled on block CMOs, the EXPORT.I instruction refers to a single cache block of a known size. We also assume the I-cache and D-cache line sizes are the same (we remove this restriction in Part V).

Part II : EXPORT.I, FENCE.ID, and IMPORT.I.

Steps involved in I/D consistency: producer

- To make instruction fetches consistent with data updates in a system that may have incoherent I-caches, the “producing” thread must:

1. Write the new code image to memory using data stores.
2. After step 1, force the new image to be pushed to the point where the I-fetch mechanisms for all HARTs will see the new code (for some systems, that’s main memory, in others, it’s some level of the cache hierarchy).
3. After step 2, invalidate all instruction caching for the new block of memory systemwide.

- We call steps 2 and 3 “exporting” the instructions. Exporting makes the instructions visible to all I-cache fetching and removes any stale I-cache copies. Exporting does nothing to in-flight stale instructions in post I-cache instruction buffers/execution pipes.

Steps involved in I/D consistency: consumer

- To make instruction fetch consistent with data updates in any system (whether the I-cache is coherent or not), the “consuming” thread must:

    1. Flush any architecturally uncommitted instructions from any post I-cache buffers and execution pipes so that those instructions will be re-fetched before they are executed.

- We call this step “importing” the instructions. Note that the consumer actions apply only to the consuming HART’s post I-cache buffers/execution pipes and does not affect I-caches in either the consuming HART or any other HART.

Producer thread instruction: EXPORT.I

- (In more detail) EXPORT.I will:

1. **Step 1**: Cause any data updates visible to the HART executing the EXPORT.I in the cache block specified by the EXPORT.I to be made visible to any subsequent instruction cache fetches by any HART in the system (i.e. copy the data to the Point of Unification or PoU -- where data updates and instructions fetches meet in the system).

2. **Step 2** : AFTER step 1, Invalidate any instruction caching (if present) for the referenced memory region. By “invalidate” we mean EXPORT.I will ensure that for the cache block containing the given byte address, all instruction cache copies for that block are invalidated and cannot be reloaded with an image of memory any older than the image pushed to the PoU in step 1. The various I-caches need not be invalidated all at once, but all the I-cache invalidations must be complete before the EXPORT.I instruction can be considered “done” by FENCE.ID (see below).

Producer thread instruction: FENCE.ID

- FENCE.ID (called the "ID"(*) fence with apologies to Sigmund Freud) will:

    1. Ensure the effects of any older EXPORT.I's on the HART executing the FENCE.ID have fully propagated throughout the system before any younger IMPORT.I (described on later slide) or STORE instructions can begin execution on the HART executing the FENCE.ID.

- Note this is a completion fence (something RISC-V doesn't have until now), not an ordering FENCE. The older EXPORT.I's must be completely "baked" and finished throughout the system before the younger stores or IMPORT.I's can even start.

- It may also be necessary for FENCE.ID to stall subsequent loads. This is under consideration (TBD).

(*) Sigmund Freud's model of the psyche: The "id" acts according to the "pleasure principle" -- the psychic force that motivates the tendency to seek immediate gratification of any impulse.

Sidebar: EXPORT.I need not invalidate D-caches

- As a matter of microarchitecture, EXPORT.I may, but is certainly not required (or even encouraged) to, invalidate (i.e., change the cache state to “invalid”) any D-cache lines that are pushed down the cache hierarchy to ensure that the data storage updates are visible at the Point of Unification (PoU).

- EXPORT.I is not even required to “clean” the cache states (e.g. convert a “modified” line to “exclusive” once a copy of the line is written to main memory). All that is truly required is that the data is updated in a way such that future I-cache fetches will observe the new values and the cache states and data are left in a state consistent with the cache coherence protocol.

Sidebar: EXPORT.I I-cache invalidation details

- The I-cache invalidation portion of EXPORT.I must not invalidate (simply set the cache state to invalid without writing back data) any cache line that can service both data and instruction reads (unified I/D cache line). Simply invalidating the line would potentially destroy modified data.

- As described above, the I-cache invalidations for an EXPORT.I need not happen at all I-caches at once. This means, for example, one HART could see a new instruction after it felt the effect of the I-cache invalidation of the EXPORT.I but before the subsequent FENCE.ID has fully propagated the EXPORT.I, set a flag a second HART reads, and then have the second HART attempt to execute the same new instruction and instead see an old instruction at the location. Put another way, the instruction kills in EXPORT.I are NOT "multi-copy-atomic". This is deliberate. The one to all nature of the I-cache kills inside an EXPORT.I makes them difficult to make atomic and there's no good reason to do bother. The FENCE.ID is what ensures the EXPORT.I's i-cache kills have finished everywhere.

Sidebar: EXPORT.I is not the “flush” for security

- For security reasons (Spectre/Meltdown attacks and other such things), there is a desire amongst the security folks to have various instructions or mechanisms to “flush” or “clean” (whatever those words will wind up meaning) side microarchitectural state. Things like branch buffers, I-caches, LRU mechanisms, and so on.

- EXPORT.I does not and should not ever provide these functions as a matter of architecture. The extension that eventually houses EXPORT.I is much more “base” architecture than any extension that will house these security functions. The choice to implement them should be able to be made separately. Keep EXPORT.I simple and handling its specific task. Also, security users cannot count on EXPORT.I invalidating I-caches. If the I-caches are coherent, the EXPORT.I I-cache invalidating function can be and often is NOPed in the implementation.

- We do not advocate for or against any future security mechanisms, we simply advocate against mixing those into EXPORT.I in any way.

Consumer thread instruction: IMPORT.I

- IMPORT.I will (this instruction is similar to POWER's isync/ARM's isb):

  1. Step 1:First wait until all prior instructions have "completed" (*).

  2. Step 2:, Flush any instructions younger than the IMPORT.I from post I-cache buffering and execution pipes on the HART executing the IMPORT.I so that these instructions are re-fetched before they are executed.

  (*)Note: defining "completed" precisely is quite subtle, but it at least means any prior branches have resolved (this is critical and will be covered in more detail later) and that all prior instructions, if applicable, have gotten past the point of taking a trap. It does not mean that a prior load has bound its eventual return value, much less returned it.

  The IMPORT.I stalling in step 1 for prior instructions completing is critical to making some examples below work properly.

Simple Intra-HART example (one instruction):


```
HART0                          // no other HART is executing this new code.

ST new_instruction             // Create new instruction.

// older ST ordered to EXPORT.I by intra-thread address collision, not a barrier.

EXPORT.I new_instruction       // Make new instruction visible to any HART I-cache fetch,
                               // including this HART
FENCE.ID                       // Be sure the EXPORT is really truly done.

IMPORT.I                       // Flush any post I-cache buffers on this HART.
                               // FENCE.ID ensures post I-cache flush cannot occur
                               // until the local I-cache has been cleared.

BR new_instruction             // This HART can execute new instructions now.
```

Intra-HART example: larger code image.

HART0

```
ST new_instruction_1          // Create new instructions, Takes "n" stores to create the
ST new_instruction_2          // new code image (may be multiple cache lines).
        …
ST new_instruction_n          // "n" stores to fill code image out.

// Stores ordered by address collisions to the EXPORT.I instructions.

EXPORT.I new_instruction_1    // need "m" exports (m usually smaller than n) to push the
        …                     // modified cache blocks to the point where all I-fetches
EXPORT.I new_instruction_m    // into the I-cache will see them

FENCE.ID                      // Be sure prior EXPORT.I's are done.

IMPORT.I                      // Only need one import.i (has no address and is for this HART)
BR new_instruction            // This HART can execute the new instructions.
```

Cross HART example, data flag signal, HART1

HART0

ST new_instruction (1) // create new inst.
EXPORT.I new_instruction (2) // export it
FENCE.ID (3) // be sure export is done to all HARTs

ST flag,1 (4) // come and get it…

IMPORT.I (5) // flush post I-cache pipe on HART0
BR new_instruction (6) // go fetch it on this HART(*)

HART1

loop: LD flag (7) // wait for flag. The
CMP flag,1 // IMPORT.I can't clear
BNE loop (8) // until this BR resolves

IMPORT.I (9) // flush pipe.
BR new_instruction // go run it.

For HART1 executing the new code, IMPORT.I (9) will stall until the prior BNE loop (8) is resolved. BNE loop (8) cannot resolve until the LD flag (7) receives the value from HART0. The ST flag,1 (4) cannot provide the value to HART1 until FENCE.ID (3) is cleared. FENCE.ID (3) won't clear until EXPORT.I new_instruction (2) has made the new instruction visible to HART1 and cleared the I-cache for HART1. Once IMPORT.I (9) can stop stalling, the post I-cache pipe is flushed on HART1 and HART1 is guaranteed to re-fetch the new instructions written by ST new_instruction (1).

IMPORT.I stalling for prior branches is critical.

- The IMPORT.I instruction stalling for prior branch instructions is critical.
- This example would not work without that stalling effect on HART1.
- Generally, IMPORT.I stalling for a prior branch allows a LD/CMP/BNE on the consuming HART to properly synchronize (stall in this case) a subsequent IMPORT.I's post I-cache flush to occur after a flag update from another HART has reached the consumer HART.
- Due to the FENCE.ID on the producer HART, the flag update is not visible to the receiving HART until the FENCE.ID ensures that the producer HART EXPORT.I has finished on all HARTs.
- We'll see another example later where the stalling action of IMPORT.I for a prior unresolved branch is also required to make the example work.

Executing new code on HART0 in prior example

HART0

ST new_instruction (1) // create new inst.
EXPORT.I new_instruction (2) // export it
FENCE.ID (3) // be sure export is done to all HARTs
ST flag,1 (4) // come and get it…
IMPORT.I (5) // flush post I-cache pipe on HART0
BR new_instruction (6) // go fetch it on this HART

For HART0, the IMPORT.I (5) cannot flush the pipe until the FENCE.ID (3) has ensured the prior EXPORT.I new_instruction (2) has made the new instruction image stored by ST new_instruction (1) available to all mechanisms that fetch into the I-cache (including HART0's) and has then cleared all I-cache copies of new_instruction (including HART0's).

At that point, IMPORT.I (5) ceases stalling, flushes the post I-cache pipe on HART0, and HART0 then fetches new instructions after BR new_instruction (6). IMPORT.I (5) is held up here by FENCE.ID (3), not a branch as it was on HART1. While both ST flag, 1 (4) and IMPORT.I (5) must occur after FENCE.ID (3), there is no required ordering between them.

Send EXPORT.I down the store pipe?

- We'll assume here that EXPORT.I will be processed like a store and flows down the store pipes/machinery. Like stores, EXPORT.I is a "fire-and-forget" operation that the core issues and gets no response back on, so it is natural to route EXPORT.I's through the store pipes.

- This is a more natural fit for ordering with fences and allows EXPORT.I's to be pipelined which is useful for modifying large blocks of new code.

S/W thread migration across HARTs

- The I/D consistency mechanisms and instructions are intended for user level software use and user level software threads can be migrated from one HART to another at any time by the OS or Hypervisor (HYP). As such, these mechanisms/instructions must tolerate these migrations.

- To illustrate the issues, we'll consider the simple intra-thread code modification sequence and allow a migration after any instruction.

- The interesting cases are when the migration occurs after the stores to create new instruction(s), but before the EXPORT.I and when the migration occurs after the IMPORT.I and before branching to the new instructions.

Simple Intra-HART example again:

HART0

```
ST new_instruction          // Create new instruction(s).
EXPORT.I new_instruction    // Make visible to I-cache fetches on all HARTs.
FENCE.ID                    // Make sure EXPORT.I is done.
IMPORT.I                    // Flush any post I-cache buffers on this HART and re-fetch.
BR new_instruction          // go get the new instructions.
```

Starting with a migration occurring after the IMPORT.I, we'll migrate this thread after each instruction to illustrate the issues around migration.

Migration after IMPORT.I and before the BR new

HART0

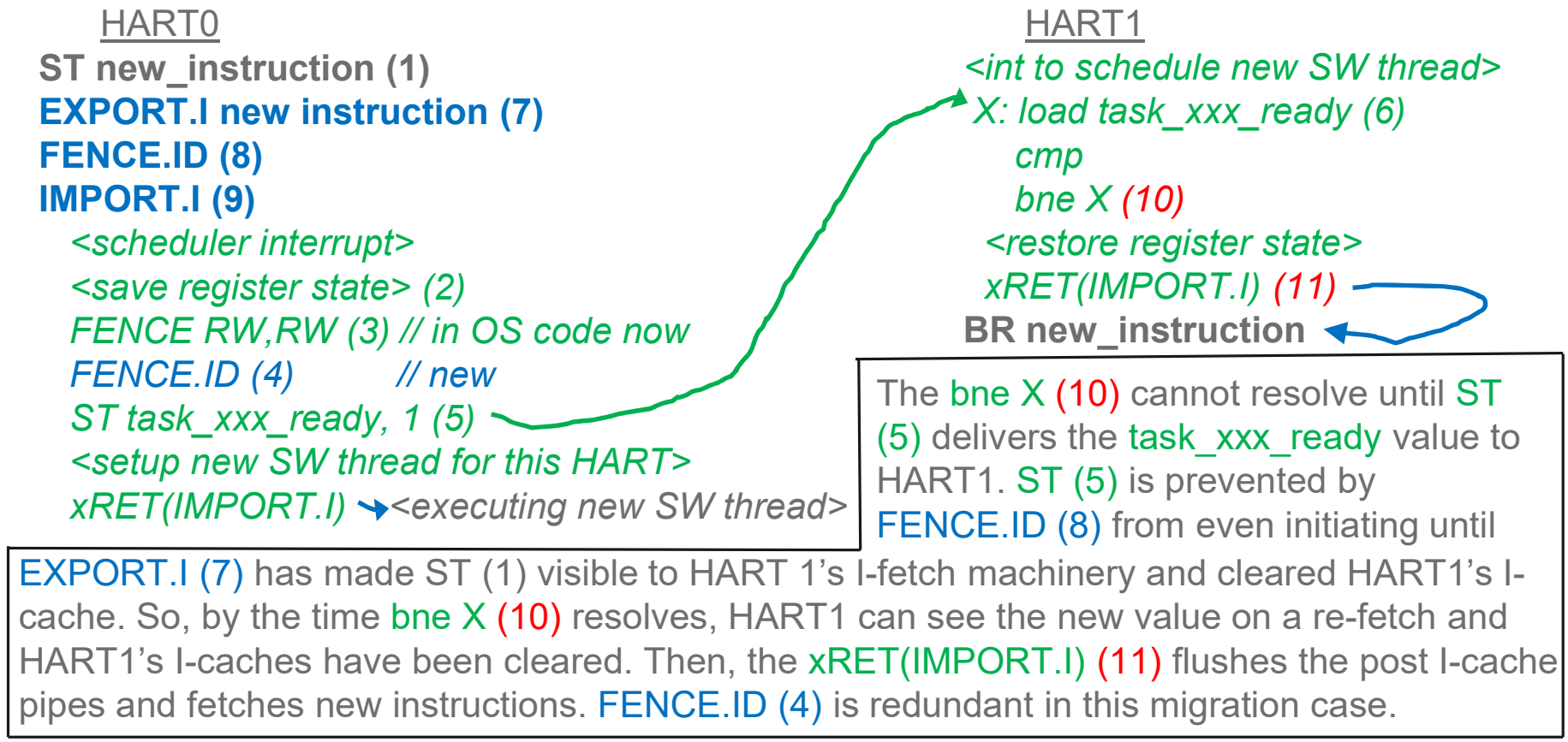
ST new_instruction (1)
EXPORT.I new instruction (7)
FENCE.ID (8)
IMPORT.I (9)
<scheduler interrupt>
<save register state> (2)
FENCE RW,RW (3) // in OS code now
FENCE.ID (4) // new
ST task_xxx_ready, 1 (5)
<setup new SW thread for this HART>
xRET(IMPORT.I) → *<executing new SW thread>*

HART1

<int to schedule new SW thread>
X: load task_xxx_ready (6)
cmp
bne X (10)
<restore register state>
xRET(IMPORT.I) (11)
BR new_instruction

In this case, the sequence through IMPORT.I (9) is completed on HART0. Unfortunately, this does not clear out the post I-cache pipe on HART1. To clear the post I-cache pipe on HART1, the xRET (11) instruction needs to include the net effect of an IMPORT.I to make up for IMPORT.I (9) that was missed or an IMPORT.I needs to be manually inserted by the OS/HYP programmer immediately preceding the xRET (11). The xRET (11) or the manually inserted IMPORT.I is stalled by the bne X (10) as explained on the next slide.

Migration after IMPORT.I continued



xRET semantics.

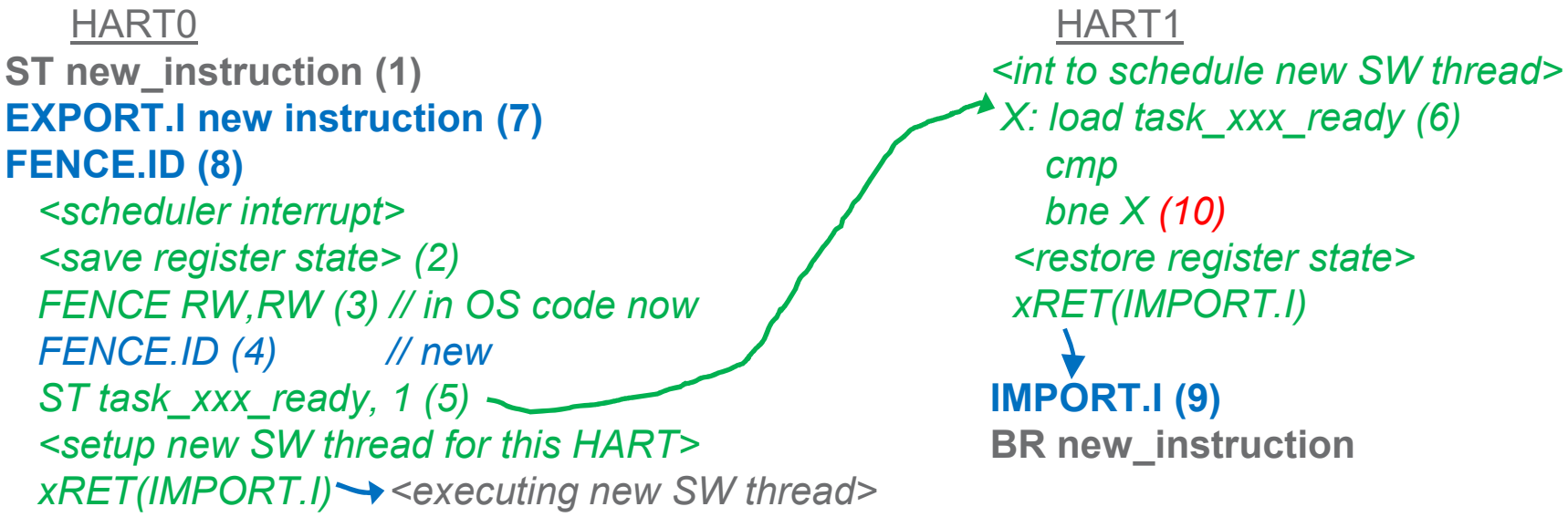
- On any xRET that may be starting a new software thread on the HART, the net effect of an IMPORT.I must be included in the xRET return path (either implicitly in the xRET, or as an explicit additional IMPORT.I instruction) to ensure that no stale instructions in the post I-cache buffers are executed. This is a subtle point that can easily be lost.

- Some architectures (Power/ARM) include this effect implicitly in the xRET instruction as part of those instructions being “context synchronizing”. Providing the IMPORT.I effect on all xRETs can be more overhead than is desired. The other option is to require the IMPORT.I to be explicitly coded ahead of xRETs returning to a new software context.

- ARM (V8.5) provides a mechanism to control whether exception entry and exception exit are context synchronization events.

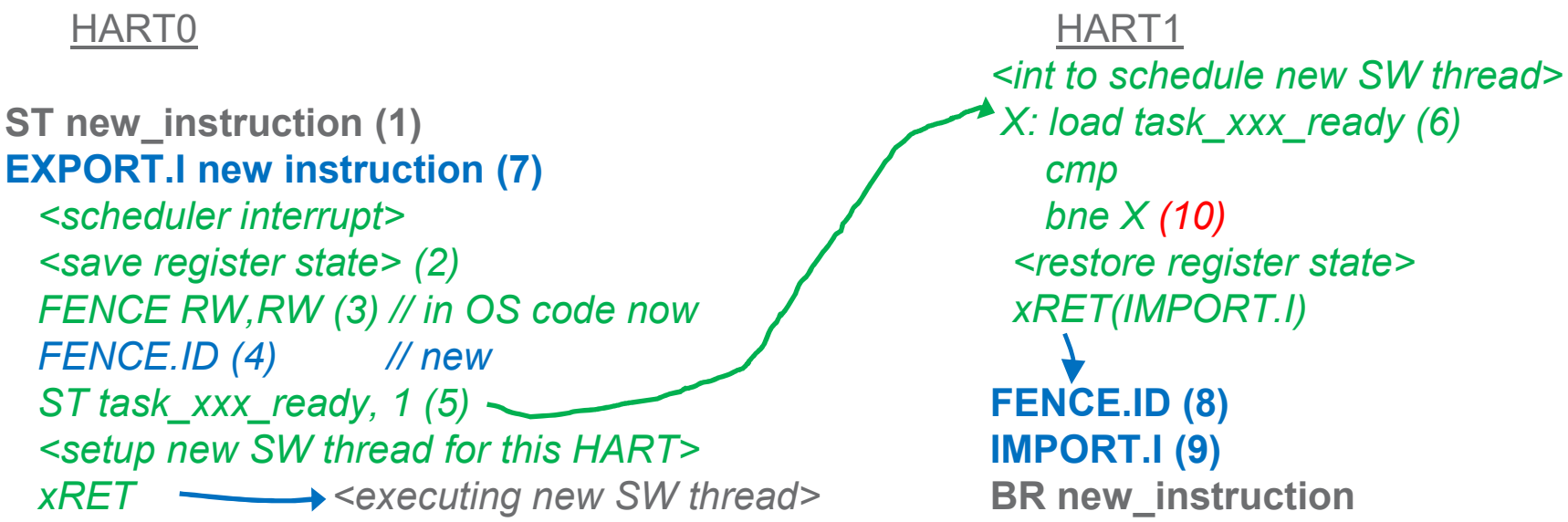
- RISC-V will need to decide whether xRETs inherently include the IMPORT.I semantics or OS/threading library programmers are required to add the explicit IMPORT.I in front of the appropriate xRETs (i.e. the xRETs returning to a new software thread).

Migration after FENCE.ID before IMPORT.I



In this case, the entire sequence except the IMPORT.I (9) occurs on HART0. The implicit (or explicitly coded) IMPORT.I at the xRET is stalled by bne X (10) by the same mechanisms explained in the previous case. Those mechanisms ensure that ST (1) and EXPORT.I (7) have propagated to HART1 before the branch resolves and the xRET IMPORT.I clears the post I-cache pipes. IMPORT.I (9) is redundant in this case as is FENCE.ID (4).

Migration after EXPORT.I before FENCE.ID



In this case EXPORT.I (7) is executed on HART0 and the OS added FENCE.ID (4) does the work that FENCE.ID (8) would have done. Again, the IMPORT.I effect at the xRET on HART1 takes the place of IMPORT.I (9) and is held up until ST (1) and EXPORT.I (7) have propagated to HART1 and then the pipe is cleared. FENCE.ID (8) and IMPORT.I (9) are redundant in this case. Also, FENCE.ID (8) won't stall at all because there are no EXPORT.I's on HART1.

Migration after store, but before EXPORT.I

HART0

ST new_instruction (1)

<scheduler interrupt>
<save register state> (2)
FENCE RW,RW (3) // in OS code now
FENCE.ID (4) // new
ST task_xxx_ready, 1 (5)
<setup new SW thread for this HART>
xRET <excuting new SW thread>

HART1

<int to schedule new SW thread>
X: load task_xxx_ready (6)
cmp
bne X (10)
<restore register state>
xRET(IMPORT.I)

EXPORT.I new_instruction (7)
FENCE.ID (8)
IMPORT.I (9)
BR new_instruction

ST (1) is made visible to HART1 by FENCE (3) and ST (5) being read by HART1. After that, EXPORT.I (7) on HART1 executes and makes ST (1) visible to all HART I-fetchers and clears the I-caches. FENCE.ID (8) stalls IMPORT.I (9) until EXPORT.I (7) is done. After that IMPORT.I (9) flushes the pipes on HART1 and the BR gets the new instructions. FENCE.ID (4) is redundant here and doesn't stall (no EXPORT.I on HART0).

Summary of migration cases

HART0

ST new_instruction // Create new instructions.

// Migration here fixed by FENCE RW,RW in OS pushing ST new_instruction to all threads.

EXPORT.I new_instruction // Make visible to I-fetches on all HARTs.

// Migration here fixed by FENCE.ID in OS finishing EXPORT.I before thread can migrate
// and IMPORT.I semantics at xRET in OS/HYP thread migration code.

FENCE.ID // Make sure EXPORT.I is done.

// Migration here fixed by IMPORT.I semantics at xRET in OS/HYP thread migration code.

IMPORT.I // Flush any post I-cache buffers on this HART and re-fetch.

// Migration here fixed by IMPORT.I semantics at xRET in OS/HYP thread migration code.

BR new_instruction // go get the new instructions.

The bne X (10) is critical

- The thread migration cases above, except for the migration after the initial stores to create the new instructions, depend critically on the bne X (10) instruction in the HYP/OS thread migration code not resolving too early.
- In particular, that branch needs to be conditioned on some load instruction that was sourced by a store done by the thread creating the new code image. That store must occur after the FENCE.ID on the creating thread, either directly as a store from that thread or as a store ordered after that store through some number of barriers and flag writes through one or more threads.
- We don't believe there is any way for an operating system to migrate a software thread from one thread to another without there being some load and branch conditioned on such a store.

The $2 US Bill challenge.

- We've checked with multiple OS/HYP coders about this and none of them have a way to build thread migration code that doesn't have a store, load, and branch instruction sequence like this in the path. We solicit any examples people have of a way this could happen.

- A crisp, unused $2 US bill and untold bragging rights are yours if you're the first person who can produce such an example. Contact Derek Williams if you think have such an example (derekwilliams.austin@gmail.com). The contest judge's decision is final ☺.

- No cheating… the interrupt that starts the process of brining the thread onto HART1 must be a random timer interrupt that doesn't know anything about the state of threads being put to sleep (i.e. you can't just assume that there is some IPI that triggers the new thread that "just knows" the task is there without having to look at data structures in memory).

Migration cases for cross HART example, HART0

HART0

ST new_instruction (1) // create new inst.
EXPORT.I new_instruction (2) // export it
FENCE.ID (3) // be sure export is done to all HARTs

ST flag,1 (4) // come and get it…

IMPORT.I (5) // flush post I-cache pipe on HART0
BR new_instruction (6) // go fetch it on this HART

HART1

loop: LD flag (7) // wait for flag. The
CMP flag,1 // IMPORT.I can't clear
BNE loop (8) // until this BR resolves

IMPORT.I (9) // flush pipe.
BR new_instruction // go run it.

Migration of HART0 works for the same reasons as the simple intra-thread example. A migration after ST (1) will be corrected by OS/HYP FENCE RW,RW (3) pushing ST (1) to all HARTs for the EXPORT.I (2) to pick up on the next HART. For a migration after EXPORT.I (2), OS/HYP FENCE.ID (4) will complete EXPORT.I (2) before the HART migrates. A migration after FENCE.ID (3) occurs after the instructions are fully exported to all HARTs. At that point, it does not matter which HART executes ST (4) to signal HART1. For HART0 executing the new code at BR (6), whatever HART we land on has been cleared out by the FENCE.ID(3)/EXPORT.I(2) or the OS barriers/exports before IMPORT.I(5) executes.

Migration cases for cross HART example, HART1

HART0

ST new_instruction (1) // create new inst.
EXPORT.I new_instruction (2) // export it
FENCE.ID (3) // be sure export is done to all HARTs

ST flag,1 (4) // come and get it…

IMPORT.I (5) // flush post I-cache pipe on HART0
BR new_instruction (6) // go fetch it on this HART

HART1

loop: LD flag (7) // wait for flag. The
CMP flag,1 // IMPORT.I can't clear
BNE loop (8) // until this BR resolves

IMPORT.I (9) // flush pipe.
BR new_instruction (10) // go run it.

Migrations of HART1 that occur before LD (7) reads a '1' for 'flag' just "work out" and loop back to LD (7) to re-read 'flag'. There is nothing on HART1 in the LD/CMP/BR loop that is affected by a migration (aside from which HART executes the LD (7)). In the special case where HART1 migrates onto HART0 after ST (4), the FENCE RW,RW (3) in the OS/HYP will have finished ST (4) before the code for HART1 begins executing on HART0. Once a value of '1' is read for 'flag' by LD (7), subsequent HART1 migrations before IMPORT.I (9) are fixed by IMPORT.I (9) and migrations after IMPORT.I (9) and right after BR (10) are corrected by the IMPORT.I semantics at the OS/HYP xRET that establishes HART1 on a new HART.

Bit of trivia: no EXPORT.I in the OS code

- The OS/HYP software thread migration code above only has fences (FENCE RW,RW (3) and a FENCE.ID (4)) and an IMPORT.I (if not implicitly included in the xRET) added to it. In other words, only operations that do not take an address as a parameter.
- It's important that no EXPORT.I's are needed in the OS migration path. EXPORT.I is tied to working on a given cache block. In other words, to use EXPORT.I one needs to know which addresses are being modified.
- Unfortunately, an OS will never be able to generally know which code in memory is eing modified by a user level program. User level thread switching library's semantics aren't know to the OS, for example.
- It's a good sanity check of the proposed architecture that it doesn't require the OS to know this information which, in general, it can never know.

EXPORT.I can get expensive.

- The EXPORT.I function can get expensive to implement as you scale to larger and larger numbers of cores (in a retrying bus system, it can get hideously expensive).
- This is generally because EXPORT.I is a global operation that must affect all HARTs in the system. Starts to hurt with larger numbers of HARTs.
- For this reason, many systems, as they scale to larger numbers of HARTs, resort to coherent I/D caches. The coherence protocol can more efficiently track where the line is and invalidate it on the instruction-creating stores vs. a broadside EXPORT.I to every HART. IMPORT.I is still required however.
- In systems that go for coherent I/D caches (to whatever degree) it is possible to NOP various portions of EXPORT.I's semantics as the hardware takes over the responsibility for maintaining I/D consistency.

Part III : Some JIT sequences.

JIT code modification sequences.

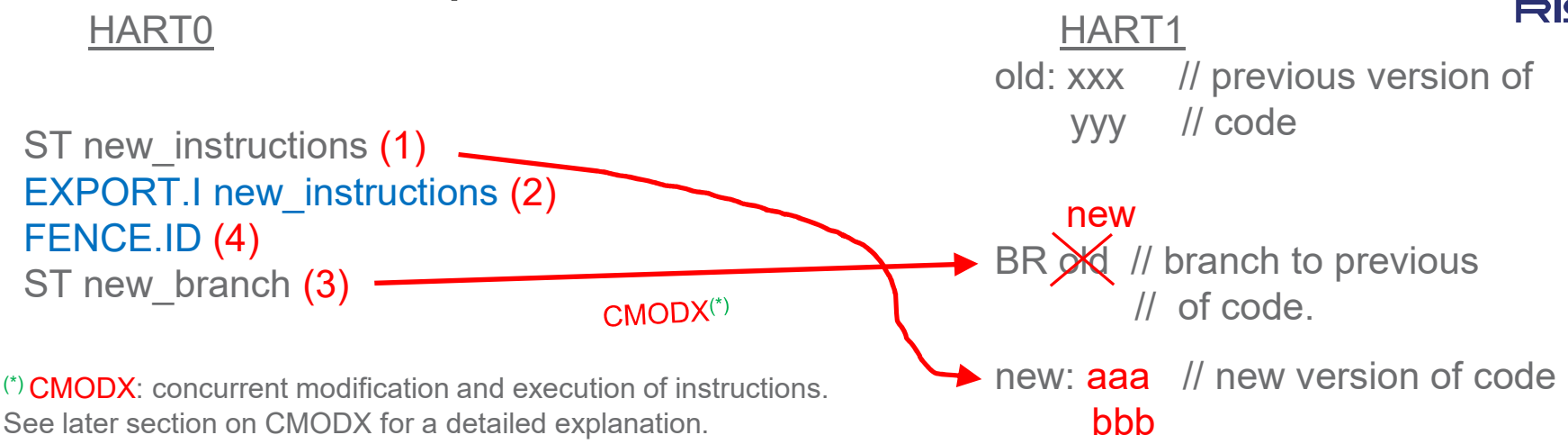
- We now consider some JIT code modification sequences.
- These are different from what we've seen. They examples may not be fully synchronized on the either the producing or consuming HARTs.
- The first example comes from Ben Simner* of Peter Sewell's group in Cambridge (or at least that's where we learned of it – called MP.FF+cachesync+fpo). WebAssembly requires this to work. Other JITs do this as well. The technique consists of a thread writing a new version of a buffer of code and then overwriting the branch to the old version of the code with a branch to the new version of the code.
- ARM has already corrected for this case in their architecture (Section B.2.3.5, V8.5).

* I would strongly recommend talking to Ben and Peter's group about their instruction modeling work (see https://www.cl.cam.ac.uk/~pes20/iflat/) and applying it to RISC-V as the I/D mechanisms for RISC-V are settled.

We’re skipping over a detail here: CMODX

- For the first JIT sequence we examine, one HART does an unsynchronized store to an instruction that might be executing concurrently. This sort of thing is typically called Concurrent Modification of Instructions (or CMODX).
- The architecture will eventually need to give a precise definition of what possible values and I-fetch may retrieve in the presence of racing data stores that modify that instruction.
- For now, we’ll skip over the full precise definition of the complete range of legal possible instructions fetched and simply specify what instruction was obtained in the examples.
- Eventually, a precise definition does need to be provided.

First JIT example: the “branch modifier”



(*) CMODX: concurrent modification and execution of instructions. See later section on CMODX for a detailed explanation.

Stores (1) create a new version of a block of code. EXPORT.I (2) exports the instructions to all HARTs. HART0 then modifies the branch to “old” to point at “new” (3). Since there is no IMPORT.I on HART1, it can see either the branch to “old” or to “new” (and alternate). If HART1 sees the branch to “new”, but stale values for the instructions at “new”, this breaks. This can occur if HART1 has stale values in post I-cache buffers after seeing “BR new” (the EXPORT.I (2)/FENCE.ID (4) ensure that any fetches from the I-cache will see new code at “new:” before HART1 sees the branch to “new”, but they don’t clear the post I-cache pipes).

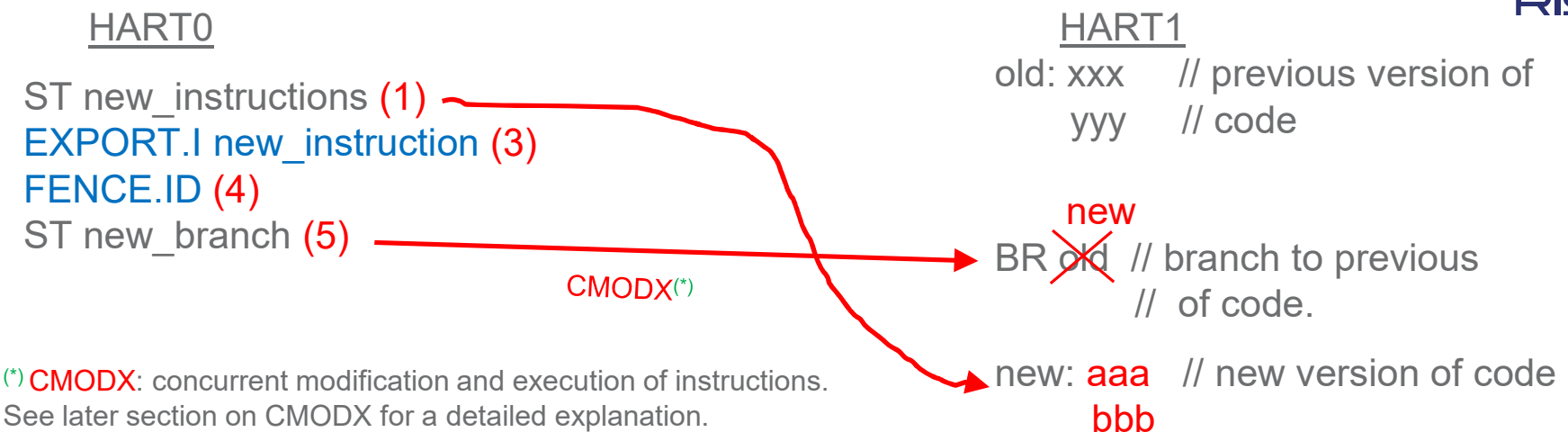
The “branch modifier” example: fixed

- The branch modifier example breaks when the executing HART (HART1 here) fetches instructions from the I-cache Out-of-Order (OoO). We’ll now call this an “instruction pull” from the I-cache to distinguish it from an I-fetch into the I-cache (OoO I-Fetches from memory into the I-cache are not a problem). An instruction pull is OoO if some instruction that was pulled from the I-cache earlier in time (the stale image at “new:” in this case) was sent down the execution pipes after an instruction that was pulled from the I-cache later in time was sent down the execution pipes (the branch to “new:” in this case).

- The easy solution is to simply outlaw OoO instruction pulls from the I-cache as a matter of architecture. If it hurts… don’t do that, the problem goes away ☺. This fixes other issues as well and generally simplifies life for your verification team. **From here out, we assume all I-cache pulls are In-Order.**

- This does NOT mean that microarchitectures cannot continue to pull OoO from the I-cache under the hood; it simply means that programs (like this particular JIT idiom) can’t ever be allowed to tell that OoO I-cache pulls are occurring.

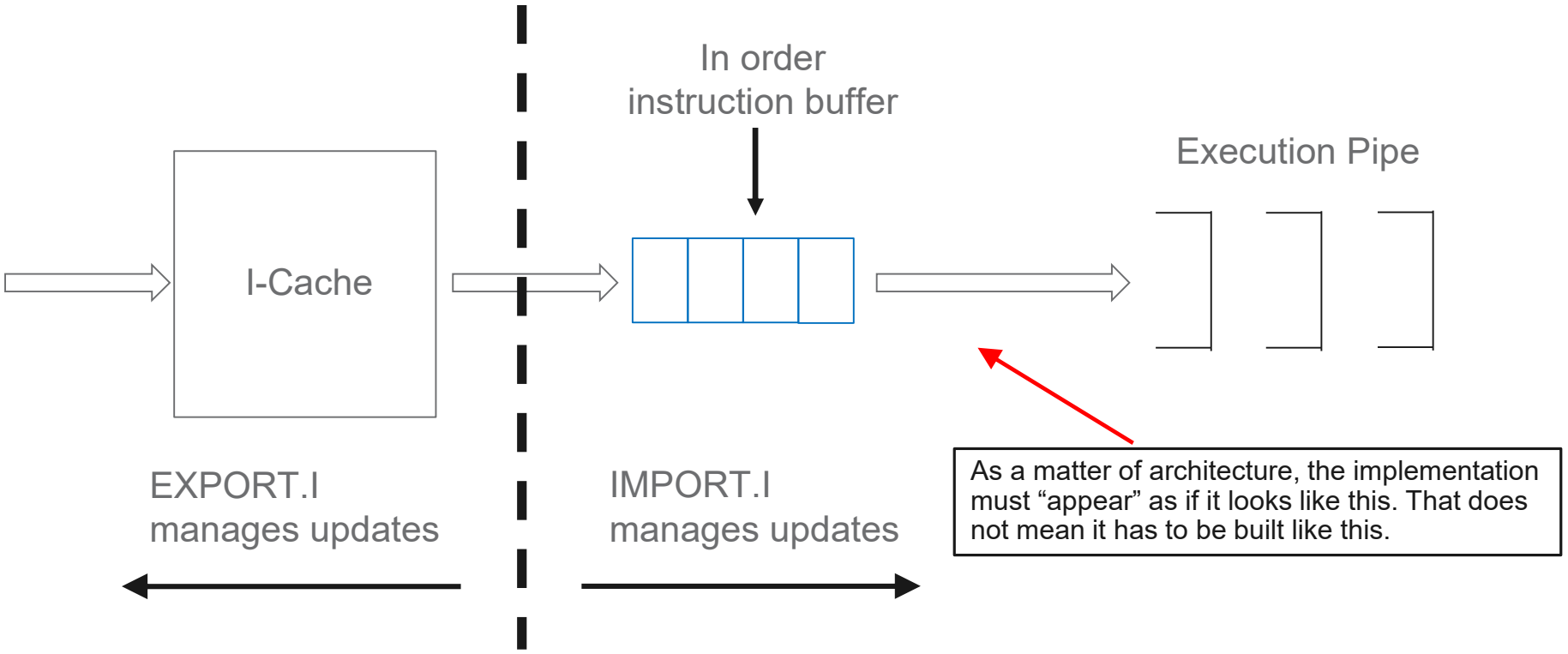
Making OoO appear in-order.



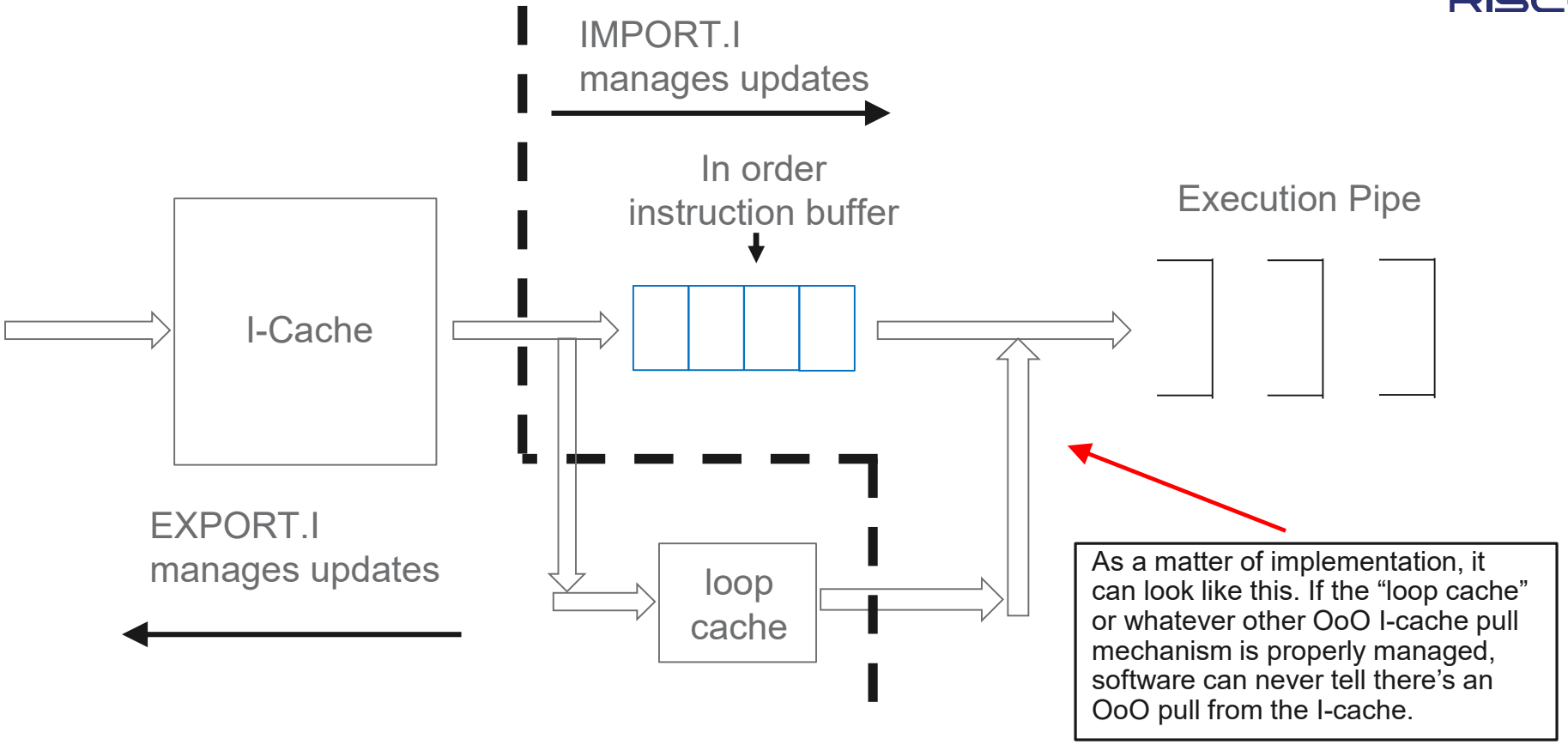
(*) CMODX: concurrent modification and execution of instructions. See later section on CMODX for a detailed explanation.

Because of EXPORT.I (3) and FENCE.ID (4), we know that the new_instructions stored by store (1) are visible to HART1's I-fetcher before HART1's I-fetcher will see the new branch from store (5). As such, if we simply invalidated any post I-cache OoO mechanisms in HART1 when we see EXPORT.I (3) on HART1, we can make the OoO pulls appear in-order (note: if HART1 didn't have an I-cache to invalidate, the EXPORT.I would still need to be presented to HART1 to flush the OoO mechanisms if present). Bottom line: you invalidate any post I-cache buffers that act OoO on the EXPORT.I's I-cache invalidation.

Simple actual in-order fetching machine:

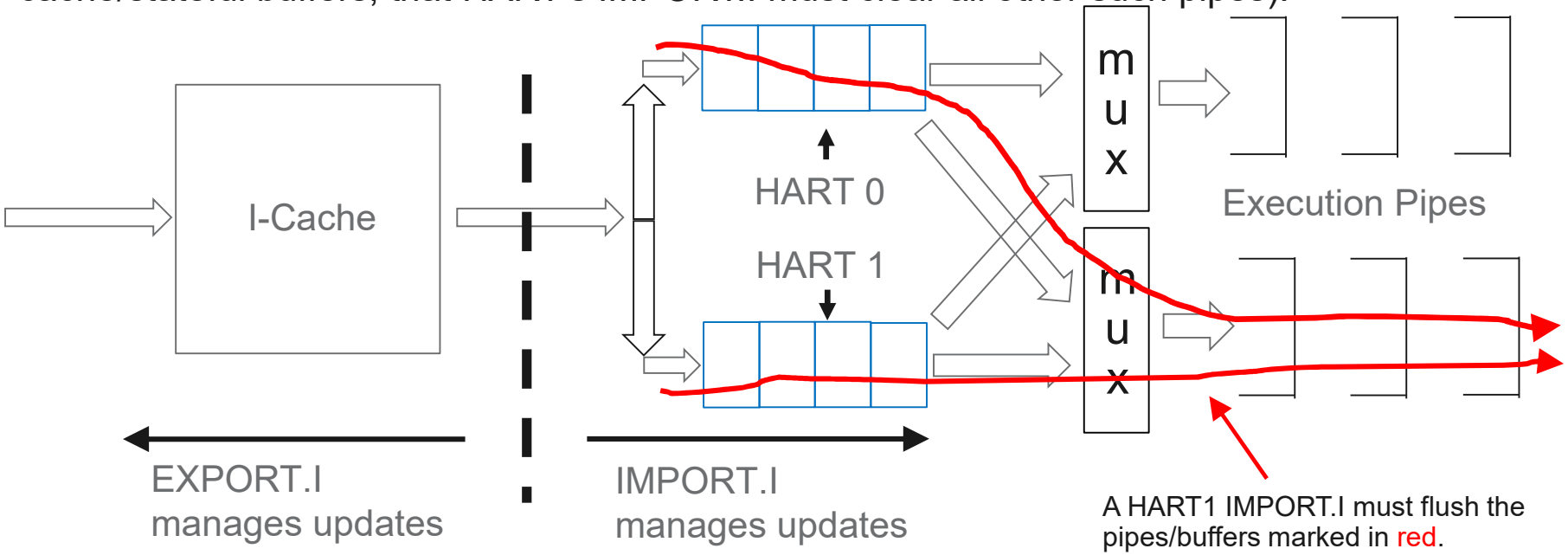


OoO machine that appears in-order:

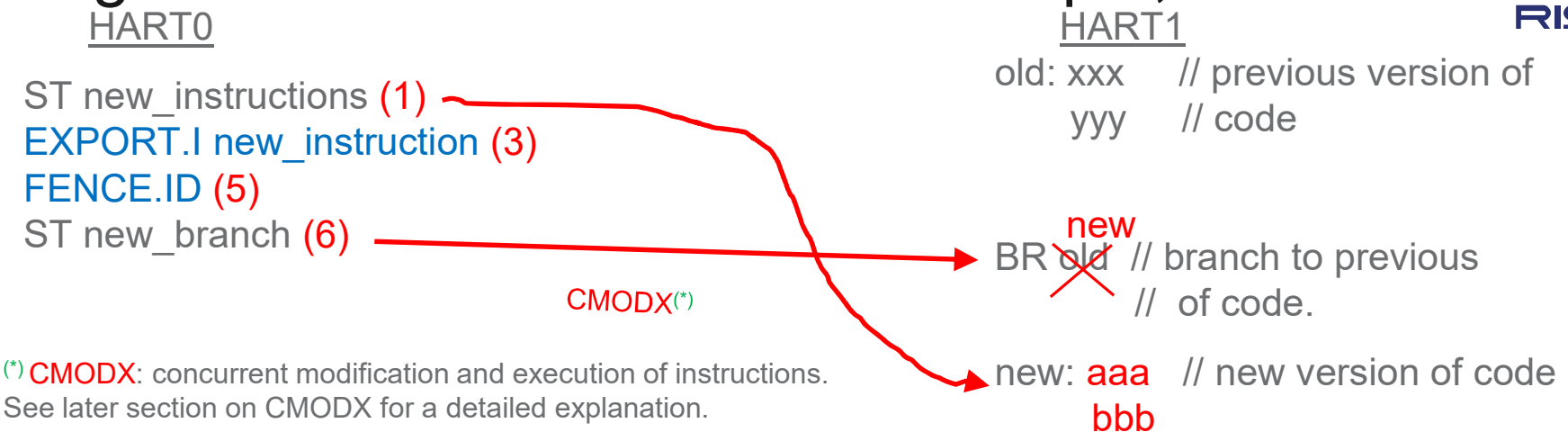


Simultaneous-multi-thread example (not OoO pull).

In this example, an IMPORT.I on HART1 would also have to flush any of the HART0 pipe that HART1 may pull from and vice-versa. The IMPORT.I effects are in terms of instructions the HART may pull, wherever they may come from after the I-Cache or post I-cache stateful buffers like loop caches (i.e. if a HART looks at another HARTs pipe post I-cache/stateful buffers, that HART's IMPORT.I must clear all other such pipes).



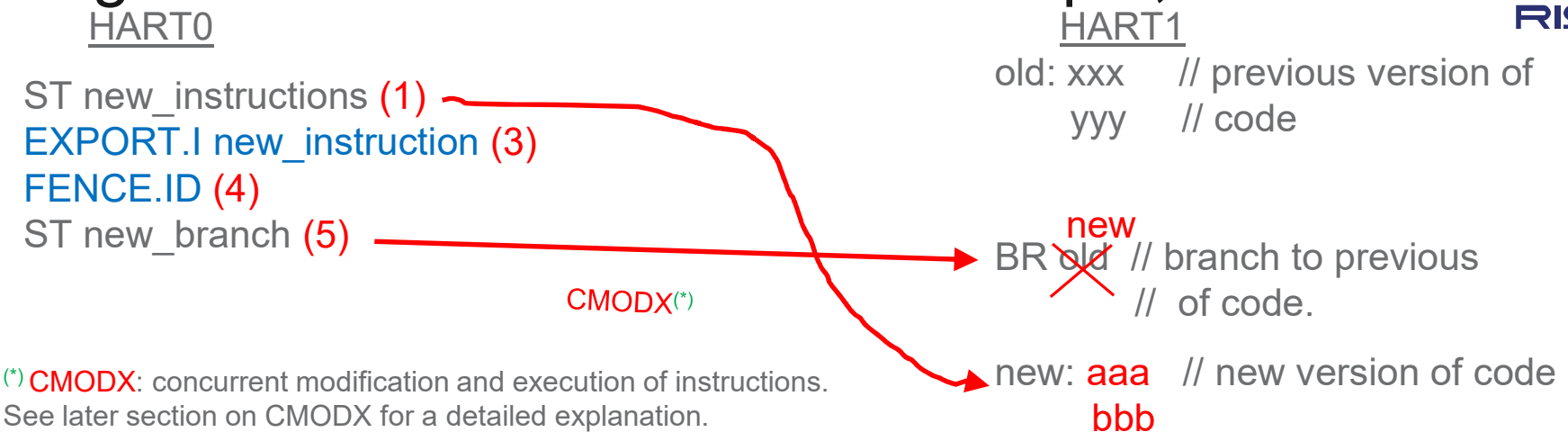
Migration for branch modifier example, HART0



(*) CMODX: concurrent modification and execution of instructions. See later section on CMODX for a detailed explanation.

Migration of HART0 works for the same reasons as the simple intra-thread example. A migration after ST (1) will be corrected by OS/HYP FENCE RW,RW (3) pushing ST (1) to all HARTs for the EXPORT.I (3) to pick up on the next HART. For a migration after EXPORT.I (3), OS/HYP FENCE.ID (4) will complete EXPORT.I (3) before the HART migrates, rendering FENCE.ID (5) redundant in that case. A migration after FENCE.ID (5) occurs after the instructions are fully exported to all HARTs. At that point, it does not matter which HART executes ST (6) to change the branch that HART1 picks up.

Migration for branch modifier example, HART1



(*) CMODX: concurrent modification and execution of instructions.
See later section on CMODX for a detailed explanation.

If HART1 migrates before the BR instruction, no harm, no foul… HART1 is just some other arbitrary HART for this example. The case that can matter is if the migration occurs after BR new (if it's BR old, the old image has already been brought up to date in some prior iteration). In the BR new case, because of EXPORT.I (3) and FENCE.ID (4) the new_instructions stored by store (1) are visible to all I-fetchers on all HARTs and all HART I-caches have been cleared after the code at new was created. Therefore, the I-cache fetch on whatever HART HART1 migrates to will see the new code at "new" (thanks to in-order I-cache pulls and execution) when HART1 migrates to the new HART and resumes execution.

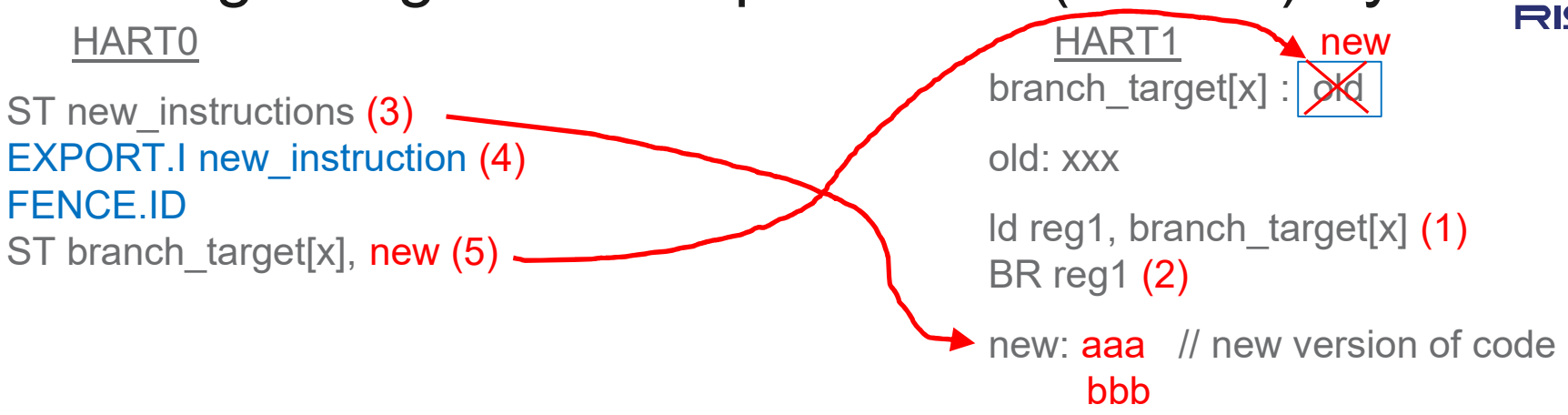
Second JIT example: “data signaling”:

- The “branch modifier” example signaled through an update to code (the branch being overwritten).

- Alternatively, JITs can signal through updates to a data table containing the entrypoint address of the routine to be run. When a new version is created, the value in the table is modified with a data store.

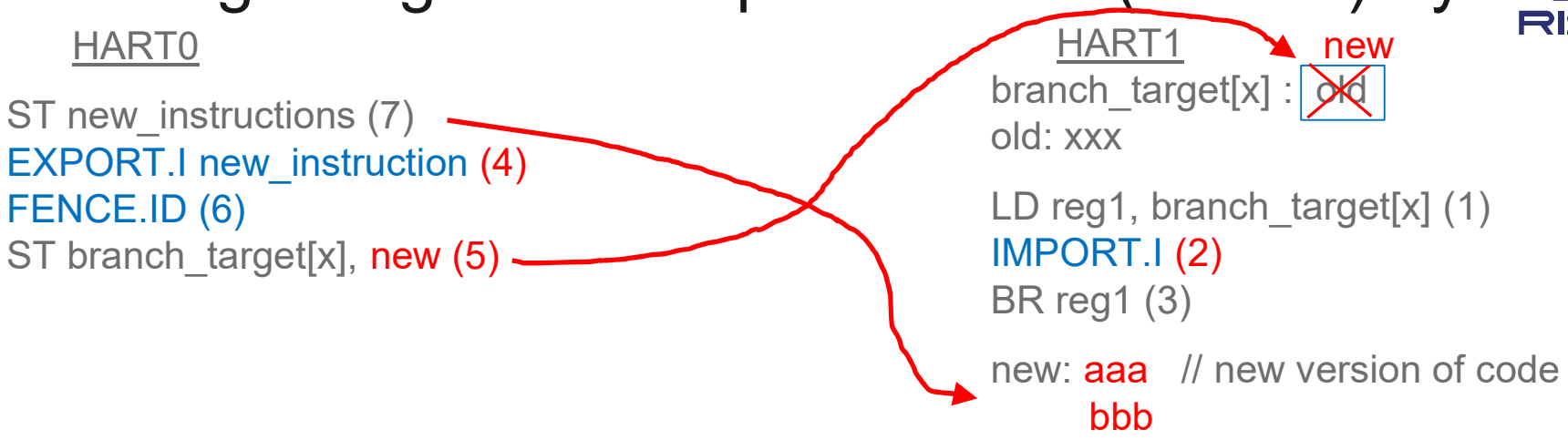
- Unfortunately, in-order I-cache pulls won’t fix this case (the update is a data store and not an update of a fetched instruction). Other techniques need to be applied.

Data signaling JIT example: a first (broken) try



Even with in-order pulls from the I-cache (which are needed to fix the WebAssembly example and we assume here), HART1 could fetch LD (1), speculatively guess the right value of reg1, and fetch the stale image of the code at “new:” before it is updated. Then the following occur: the update to “new:” from stores (3), the EXPORT.I (4) , and the update to branch_target[x] (5). The guessed value at (1) then resolves and HART1 falls through and executes the previously pulled stale image at “new:”. This breaks even if instructions are pulled in-order because the signaling is through a data write instead of updating an instruction that is fetched. All pulls from the I-cache are in-order here.

Data signaling JIT example: Second (broken) try

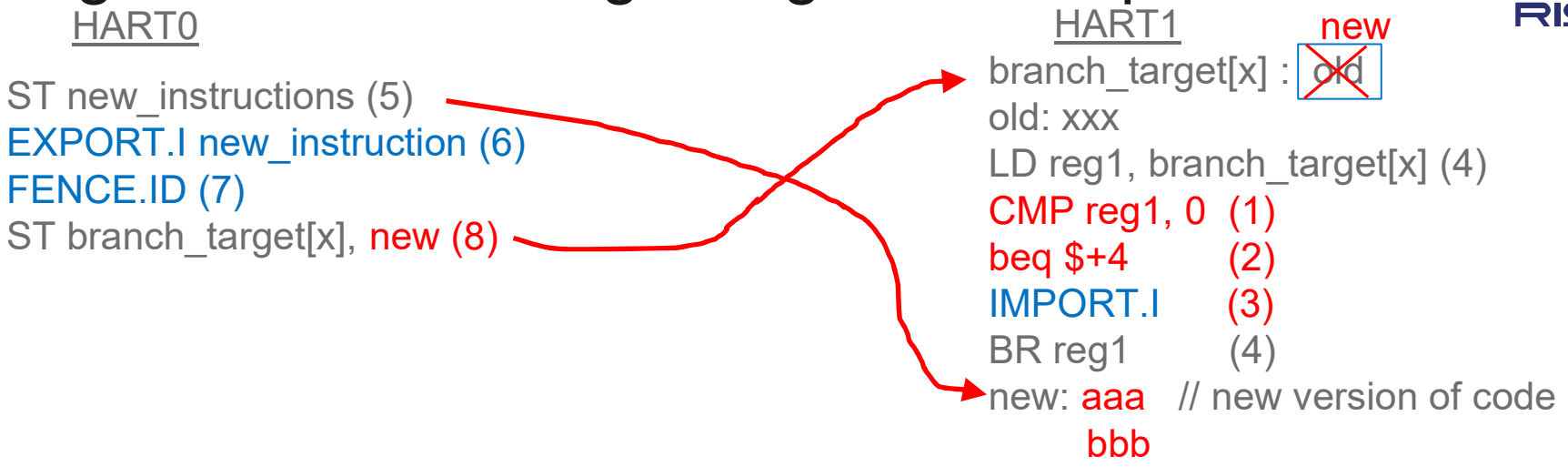


It would seem that putting IMPORT.I (2) between the LD (1) and BR (3) would fix the problem here. By the time LD (1) reads the new value, EXPORT (4) must have completed relative to HART1 and IMPORT.I (2) will clear the post I-cache pipe and BR reg1 (3) will get new instructions? No. Unfortunately, IMPORT.I (2) only needs to wait for LD (1) to "complete" (translate and not take an exception). This can happen before LD (1) even attempts to read the value. At that point IMPORT.I (2) flushes the pipe and HART1 fetches stale instructions. Then ST (7), EXPORT.I (4), FENCE.ID (6) finish, LD (1) reads the new value, and BR (3) jumps to the already fetched stale instructions. IMPORT.I (2) is not stalled and flushes "early".

Data signaling JIT example: Third (working) try.

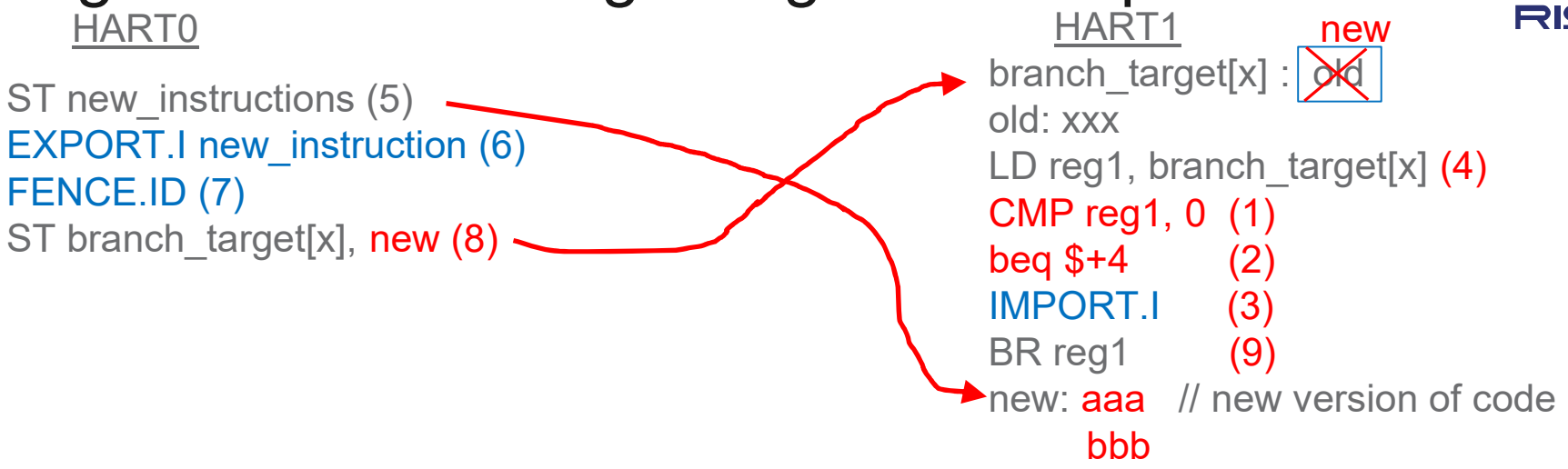
A bit of a hack can be used to fix the "problem" of IMPORT.I (3) not being stalled properly. If a compare and dummy branch to the next instruction ((1) and (2)) that are conditioned on LD (4) are inserted in the path before IMPORT.I (3), the IMPORT.I (3) will be stalled appropriately. This solution is vaguely unsatisfying, but it does work. There's a better solution beyond this that can be used for a data signaled JIT that we will discuss later: "bait-and-switch".

Migration for data signaling JIT example: HART0



Migration of HART0 works for the same reasons as the simple intra-thread example. A migration after ST (5) will be corrected by OS/HYP FENCE RW,RW (3) pushing ST (5) to all HARTs for the EXPORT.I (6) to pick up on the next HART. For a migration after EXPORT.I (6), OS/HYP FENCE.ID (4) will complete EXPORT.I (6) on HART0 before HART0 is migrated. A migration after FENCE.ID (7) occurs after the instructions are fully exported to all HARTs. At that point, it does not matter which HART executes ST (8) to change the branch_target[] array value. HART1 will eventually pick up the new entrypoint.

Migration for data signaling JIT example: HART1



HART1 migrations where LD (4) reads “old” are branching to the already established version of the code for all HARTs at “old:”. As such, migrations after LD (4)/CMP (1)/BEQ (2)/IMPORT.I (3)/BR (9) are harmless since we ultimately arrive at the already fully established code at “old:” no matter where the migration goes. If LD (4) reads “new”, the new instructions are fully exported to all thread due to FENCE.ID (7) and migrations after LD (4)/ CMP (1)/BEQ (2)/IMPORT.I (3)/BR (9) are corrected by the IMPORT.I semantics at whatever xRET establishes HART1 on a new HART. IMPORT.I (3) becomes redundant in these cases.

Would outlawing value speculation fix it?

- In the explanation above about how the data signaled case worked, it was assumed that the core correctly “guessed” the right value for the load and ran off early and pulled a stale image from the I-cache (all in order, of course).
- Could you outlaw value speculation to fix this (i.e. prevent guesses of the data value)?
- No, not really.
- Even if the data value can’t be guessed, nothing stops the fetching machinery from getting lucky (unlucky?) and “just deciding” to fetch whatever address the data write will ultimately have even if it pays no attention to the fact there is a load there.
- Aside from addressability (valid translations), nothing in the architecture constrains where the fetcher can go (and, aside from in-order I-cache pulls, not much should).

IMPORT.I always in the path

- The data signaling technique listed above is simple to understand and requires nothing more than a JITing thread writing a new block of instructions whenever needed, exporting the block, and then overwriting the entrypoint location data value. Unfortunately, that technique requires an IMPORT.I be executed on every entry to the generated code block -- even on a HART long after that HART has been "caught up" with the new code image. This can degrade performance.

- One way around the performance degradation is to optimize IMPORT.I in the implementation by keeping track, on a HART, of whether any microarchitectural event that requires the IMPORT.I to flush has occurred since the last IMPORT.I (at least one such event is an I-cache invalidate from an EXPORT.I) and NOP the IMPORT.I flush if nothing has occurred since the last IMPORT.I. This is subtle to build, but not impossible. It improves IMPORT.I performance generally.

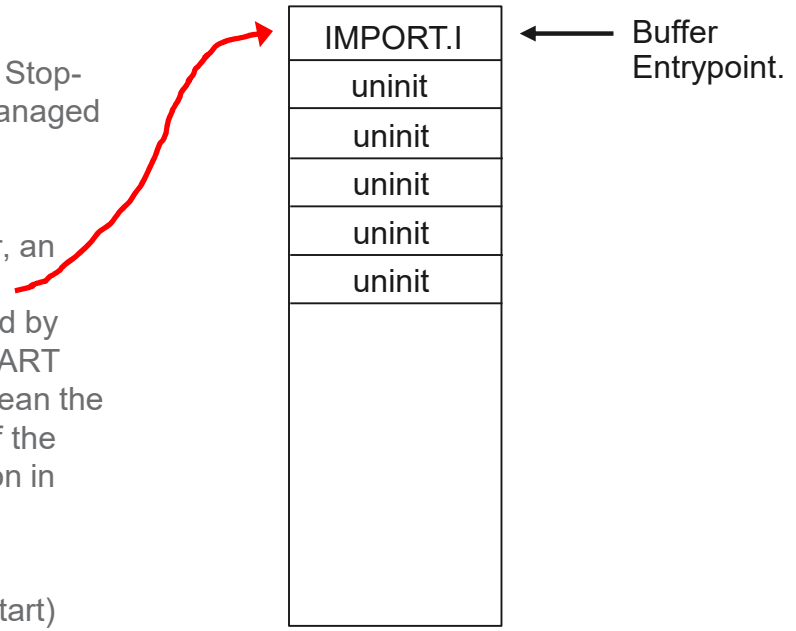
- We'll now discuss another way to implement data signaling that depends on additional guarantees from the JIT to remove the IMPORT.I overhead when possible. We call this the "bait-and-switch".

Caveat Emptor

- We believe the “bait-and-switch” technique we’re about to describe for JITs does work.
- However, “bait-and-switch” is new (to us at least) and it is subtle, so there is a possibility of oversights or errors.
- We welcome a brutal and thorough evaluation of this proposed scheme.

Setting up instruction buffers: the “bait”

- This technique for a JIT is based on instruction buffers that are pre-initialized at the “beginning of time”.
- This is accomplished using a “Stop-the-World” event in the JIT. A Stop-the-World event pauses the applications on all HARTs that are managed by the JIT. This allows the runtime to set up the buffers.
- Once in the Stop-the-World event, at the entrypoint of each buffer, an IMPORT.I instruction is written. After the buffers are updated, an EXPORT.I for each memory location that was updated is executed by the writing HART, and the JIT executes an IMPORT.I on **every** HART controlled by the JIT. The IMPORT.I’s the JIT directly executes clean the pipes on all relevant HARTs so they now see the correct image of the buffer. The Stop-the-World is then ended. **Only** the first instruction in the buffer must be initialized and exported.
- After initialization, the buffers (with IMPORT.I instructions at the start) are fully visible to I-cache fetches by all relevant HARTs and there are no stale copies in the post I-cache buffers. This is the “bait”.



Using a buffer once it's initialized: the "switch"

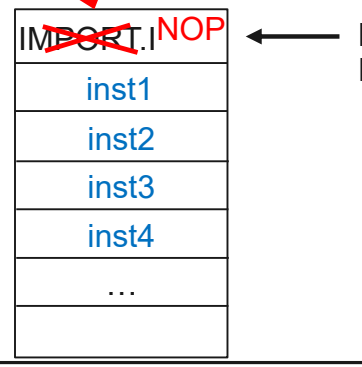
- When a buffer is to be used, the following sequence is used to rewrite the buffer and the branch to the buffer:

HART 0
ST new_instructions
EXPORT.I new_instructions
FENCE.ID(*) // Get new instructions exported before anything else.
ST branch_target[x], new
ST NOP // replace entrypoint IMPORT.I with a NOP.
EXPORT.I NOP // export it.

CMODX

HART1
branch_target[x] : ~~old~~ new

ld reg1, branch_target[x]
br reg1



- HART0 first writes the new instructions into the buffer without altering the entrypoint IMPORT.I. The new instructions are then exported and a FENCE.ID ensures that is done throughout the system before the ST branch_target[x], new even initiates, or the store and export of the NOP initiate. Once the new instructions are fully exported, the update of branch_target[x] allows the load on HART1 to resolve and execution to go to the entrypoint of the buffer. Once there, HART1 will encounter a NOP or an IMPORT.I as described on the next slide. The ST and EXPORT.I for the NOP can occur in any order relative to the store to branch_target[x], and eventually replace the IMPORT.I at the buffer entrypoint with a NOP. The ST NOP and EXPORT.I or just the EXPORT.I could be removed and this would still function(*).

(*)See later section on FENCE.ID as a completion barrier for an explanation of why this barrier cannot be a FENCE W,W.
(*)If NOP never overwritten, IMPORT.I is just executed every entry. Still works.

Two cases occur when finally branching to "new"

- **Case 1:** The branch to reg1 on HART1 encounters an IMPORT.I at the entrypoint. At this point, the IMPORT.I must stall the HART until all prior instructions have completed. In particular, the branch to reg1 must resolve. For the branch to resolve, the store to update to the branch table to "new" must have become visible to HART1. Before that can occur, the new instruction EXPORT.I on HART0 must have exported the new instructions to HART1. The IMPORT.I will then clear the post I-cache pipes and re-fetch the subsequent instructions and see the new code image (this case **does not** depend on in-order I-cache pulls in order to work -- the IMPORT.I on HART1 flushes the post I-cache pipes).

- **Case 2:** The branch to reg1 on HART1 encounters a NOP at the entrypoint. At this point, the EXPORT.I from HART1 has already been done (the ST for the NOP can't initiate until FENCE.ID clears). Because the I-cache pulls are in-order and the new instructions were written/exported before the NOP was, the I-cache pulls for the instructions after the NOP must see the new instructions (this case **does** depend on in-order I-cache pulls in order to work -- the post I-cache pipe is not cleared by the NOP).

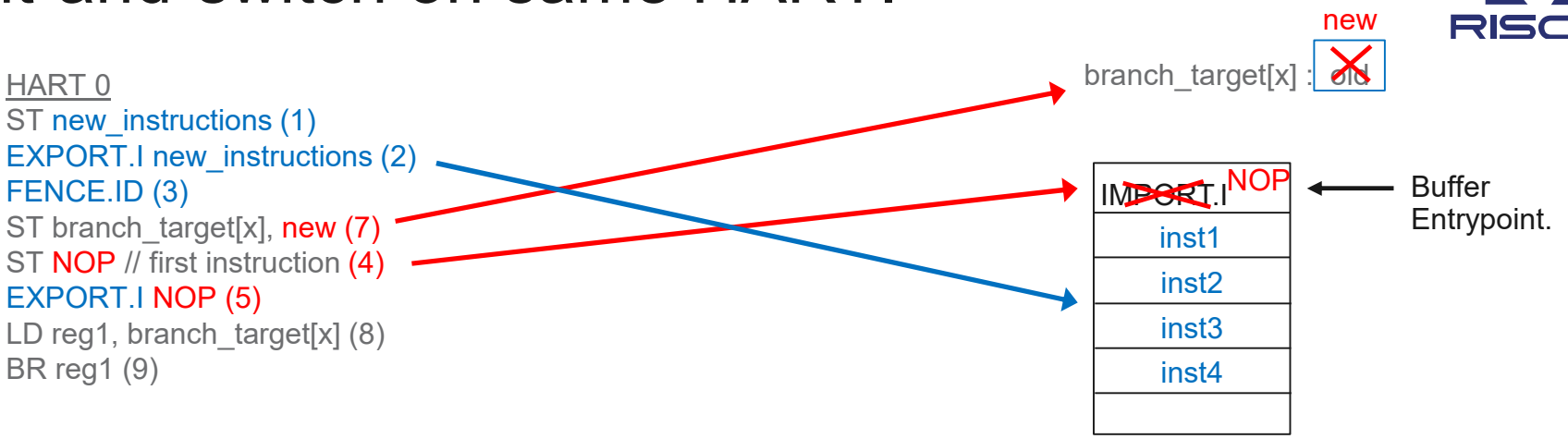
Careful when speculatively executing IMPORT.I

- To re-iterate, when executing an IMPORT.I speculatively (e.g. prior branches have not resolved), care must be taken.

- A speculatively executed IMPORT.I must stall the pipe until all prior branches are completed (aka resolved) before it can flush the post I-cache buffers. The bait-and-switch example breaks if the IMPORT.I flushes the post I-cache buffers before the branch to "new" is resolved (if done before the branch is resolved, there's no guarantee that the new instructions or the NOP have been written into the buffer and exported).

- Also, as suggested earlier, if the IMPORT.I is optimized to track microarchitectural events and not flush the post I-cache buffers if nothing has happened since the last IMPORT.I that requires it, the IMPORT.I must still stall the pipe until the prior branches are resolved. Also, the HART cannot decide to not flush the post I-cache pipe until that stall is done. The flush of the post I-cache pipe can be skipped in some cases, but the stall until prior branches are resolved can never be skipped. Also, all microarchitectural events up to the point the prior branches are resolved must be considered when deciding whether to skip the flush.

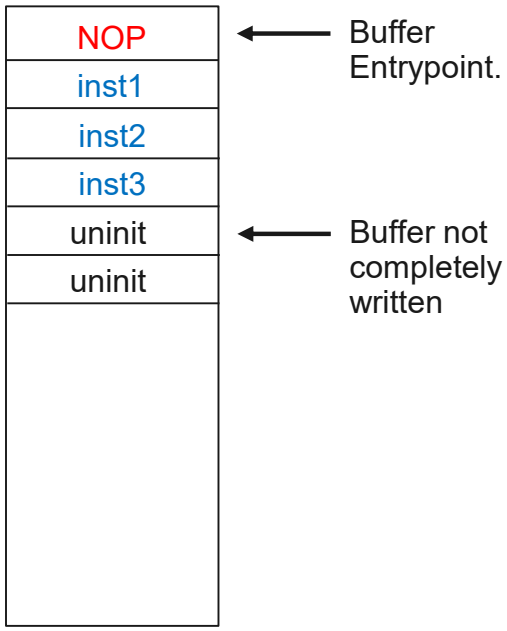
Bait-and-switch on same HART.



- This illustrates what happens when we do a bait-and-switch, but the HART executing the code is the same HART that is modifying it. The prior example had a different HART executing the new buffer. As before, the new_instructions are written first and exported (ST (1) and EXPORT.I (2)) to all HARTs including HART0 (due to FENCE.ID (3)) before the branch_target[x] is written to "new" (ST (7)). At soon as ST (7) executes (and before it propagates to other HARTs), LD (8) and BR (9) can resolve. If the fetcher on HART0 has run ahead and seen an IMPORT.I at the entrypoint, the IMPORT.I has been stalled until BR (9) resolves and the pipe is then flushed and the new_instructions are fetched and executed. If the fetcher ran ahead and encountered a NOP at the entrypoint, the new_instructions must be visible to the fetcher and are pulled in-order after the pull of the NOP. Again, the ST (4) and EXPORT.I (5) are not ordered with respect to ST (7) and eventually replace the IMPORT.I at the entrypoint with a NOP. ST (4)/EXPORT.I (5) or just EXPORT.I (5) could be removed and the sequence would continue to work (worst case you just see an IMPORT.I that's redundant at the entry).

What breaks if IMPORT.I “bait” not replaced last?

- It’s a little tedious to write all the instructions in the buffer with the exception of the entrypoint, export those, and then go back and replace the entrypoint instruction and export.
- If instead, the NOP is written before the remainder of the instruction buffer is written, it would be possible for the HART executing the buffer to pull from the I-cache a partially complete image of the buffer with a NOP as the first instruction, some of the instructions, and some uninitialized garbage as shown here.
- For “bait-and-switch” to work, when the I-cache pull receives a NOP for the entrypoint instruction, it must be the case that the rest of the new instructions in the buffer are visible to the subsequent I-cache pulls (which are in order). The way that is ensured is to write the new instructions first and export them and then write the NOP at the entrypoint and export it.

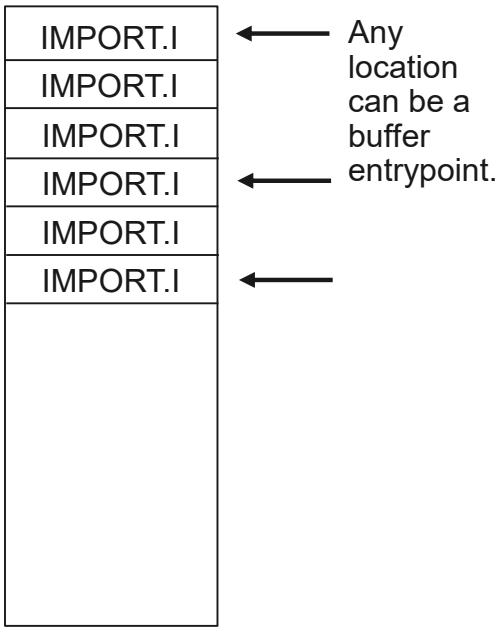


Why overwrite the entrypoint IMPORT.I with NOP?

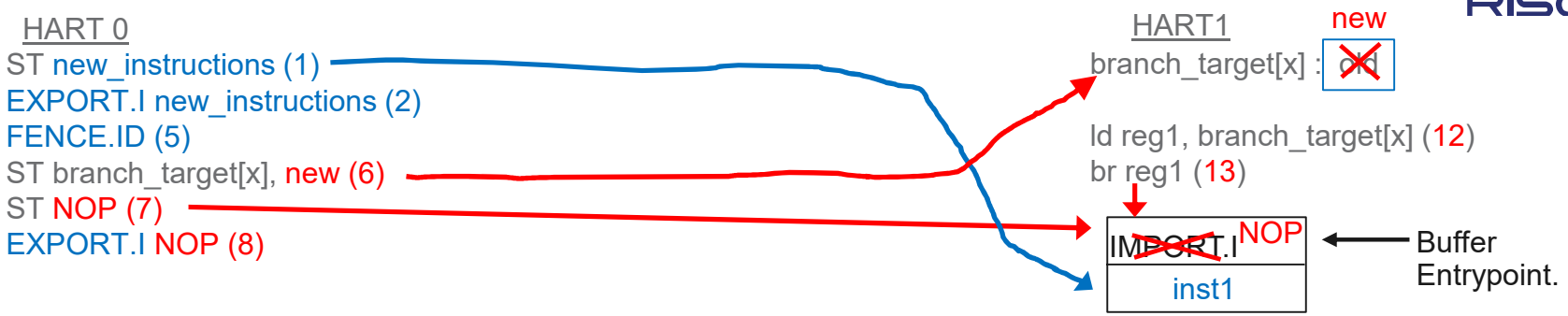
- Bait-and-switch writes a NOP over the entrypoint as shown above.
- Why “waste” the instruction with a NOP when you could replace it with the first instruction of the real code “inst1” instead?
- Because you can’t replace the IMPORT.I with a real instruction.
- If the entrypoint IMPORT.I is replaced with “inst1” and HART1 fetches “inst1” when it gets to the entrypoint, everything is fine (case 2 above). However if HART1 reaches the entrypoint “too fast” and fetches the IMPORT.I instead, HART1 will stall until the prior branch resolves, flush the pipe, and execute the next instruction -- “inst2”. That would skip over “inst1”. Boom! For this reason, the entrypoint instruction must be replaced with something that has no functional effect on the subsequent code. A NOP is the easiest answer and likely the fastest executing one as well ☺.

Multiple late bound entry points.

- If the entrypoints in the buffer are known, only those entrypoints need be initialized with IMPORT.I's when the JIT is setting up the buffers in the Stop-the-World event. This is the example we showed above (with just one entrypoint).
- If the entrypoints aren't known, an alternative technique is to fill an entire range of memory with IMPORT.I instructions. This slows down the buffer creation in the Stop-the-World
- The instruction sequence above to modify the buffers is the same, only it will be overwriting the new instructions onto IMPORT.I's instead of uninitialized garbage values.
- In fact, differing routines that share a cacheline can be JITed concurrently without impacting one another (obviously the routines can't directly overlap, only share a cacheline).

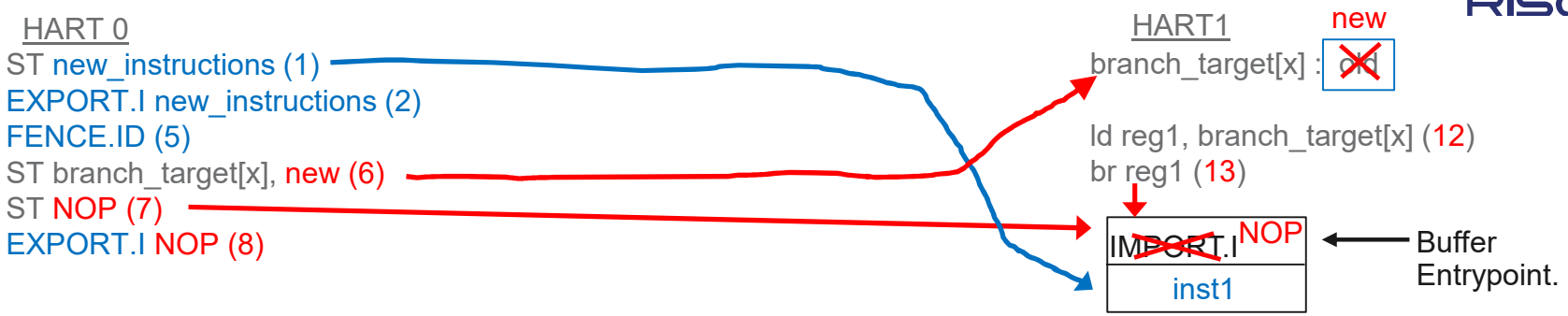


Bait and Switch HART migration, HART0



A migration of HART0 after ST (1) is corrected by OS/HYP FENCE RW,RW (3) pushing ST (1) to all HARTs for the EXPORT.I (2) to pick up on the next HART. For a migration after EXPORT.I (2), OS/HYP FENCE.ID (4) will complete EXPORT.I (2) on HART0 before HART0 is migrated and render FENCE.ID (5) redundant. A migration after FENCE.ID (5) occurs after the instructions are fully exported to all HARTs. At that point, it does not matter which HART executes ST (6) to change the branch_target[] array value. HART1 will eventually pick up the new entrypoint. Finally, a migration after ST (7) will be corrected by OS/HYP FENCE RW,RW (3) pushing ST (7) to all HARTs for the EXPORT.I (8) to find on the next HART.

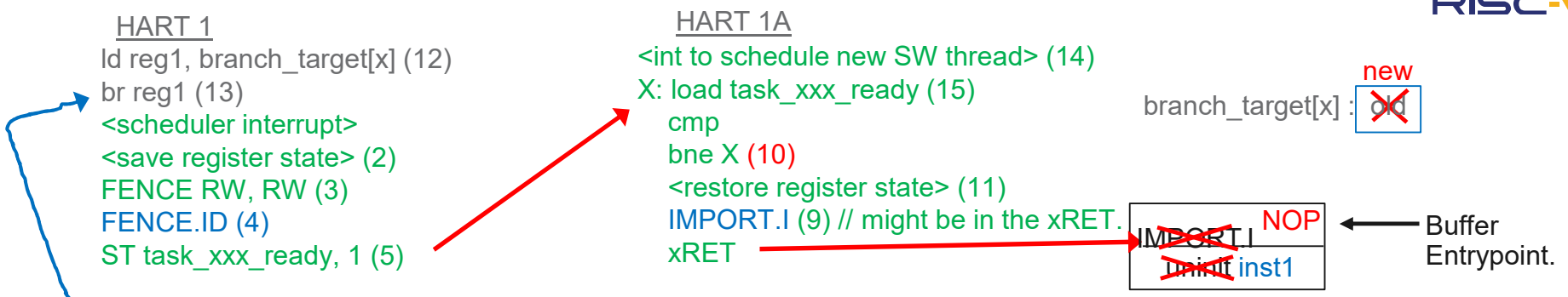
Bait and Switch HART migration, HART1



- For migrations on HART1, we only need consider the case where LD (12) receives a value of “new”. If LD (12) receives a value of “old” that means the HART1 is merely branching to some “older” buffer that is being or has already been set up by some other “HART0” executing the process shown on HART0 here.

- The same reasoning shown on the preceding slide applies to that other “HART0” setting up the buffer at “old”. A buffer cannot be actually, non-speculatively executed without that buffer having already been set up by some “HART0” (i.e. a “HART0” that has executed FENCE.ID (5) after ST (1) and EXPORT.I (2) ensuring the new image is fully exported).

Bait and Switch HART migration, HART1, cont'd



- HART 1 could be interrupted after br (13), to be migrated to HART 1A. HART1A could be interrupted and then "guess" and pull the instructions for the OS/HYP thread migration code, the buffer entrypoint IMPORT.I, and uninitialized instructions past the entrypoint into the post I-cache buffers on HART 1A before HART 0 has updated the buffer and made the value "new" available to HART 1A (yes, this is very hard to do, but not impossible or illegal). If this occurs, bne X (10) will stall the pipe until LD (15) is satisfied. LD (15) will not be satisfied until ST (5) propagates to HART 1A which cannot happen until LD (12) has been satisfied. LD (12) cannot be satisfied until ST (6) on HART0 (not shown) has propagated to HART 1. ST (6) cannot initiate until FENCE.ID (5) on HART 0 has ensured that ST (1) and EXPORT.I (2) on HART 0 have fully exported the new instructions to all HARTs including HART 1A. Therefore, IMPORT.I (9) will flush the HART 1A post I-cache pipe after the new instructions are visible to an I-cache pull on HART 1A and the new instructions will be pulled from the I-cache. The buffer entrypoint may still contain an IMPORT.I (ST (7)/EXPORT.I (8) on HART0 that writes NOP over the buffer entrypoint can occur later in a lazy fashion), but this IMPORT.I is redundant in this migration case. A similar chain of reasoning applies if HART 1A is migrated after LD (12) and before BR (13). If HART1A encounters a NOP at the entrypoint, the next instructions must be the new instructions due to in-order I-cache pulls and the NOP being written after the new instructions were fully exported to all HARTs. Note: IMPORT.I (9) might be part of the xRET instead.

Recycling a buffer in a JIT.

- Instruction buffers that have been replaced eventually need to be reclaimed. A buffer becomes eligible to be reclaimed when two things have occurred:
  - The branch instruction to the new buffer or the buffer address in the entrypoint data table is overwritten.
  - The JIT can ensure that there is no possible path that will actually execute that leads to the old buffer (the garbage collector ensures this by means not discussed here).
- For the bait-and-switch case, once the JIT enters a Stop-the-World event and determines both of these things have occurred, the JIT can simply overwrite the buffer entrypoint with an IMPORT.I or fills the buffer with IMPORT.I's as desired and then EXPORT.I's the buffer or entrypoints. The JIT then executes an IMPORT.I on all HARTs to ensure that the newly reclaimed buffer is fully visible to all HARTs and ready to be re-used (this is the same process as when the buffer was first initialized). For the branch-modifier case, the buffer doesn't need to be re-initialized, it's simply overwritten and exported.

Before ‘bait-and-switch’ there was ‘all-zeros’

- Before bait-and-switch was developed, a technique called “all-zeros” was also considered. Essentially, instead of using an IMPORT.I instruction, in “all-zeros”, the entire buffer was filled with words with 0’s which is an illegal instruction.

- When an illegal instruction is executed, the illegal instruction handler is invoked. The illegal instruction handler executes a normal data read to determine if the location appears as a ‘0’ for a data read. If so, the illegal instruction handler takes the fault.

- Otherwise, the illegal instruction handler executes an IMPORT.I (or the handler xRET does this, if it includes an implicit IMPORT.I) to clear the local HART pipe and returns to the “illegal instruction” address to re-try the instruction (which the I-fetch hopefully sees differently now). Because this executes the instruction again, the word at the entry point can be replaced with an actual instruction instead of a NOP.

- This technique requires the illegal instruction handler in the OS to be hacked to work. As such, we prefer “bait-and-switch” and won’t discuss “all-zeros” further here, though if you can convince the OS coder into altering the handler, this should also work.

No new ISA primitives required for JIT cases

- All of the JIT techniques discussed above (branch modifier, data signaled, bait-and-switch, all zeros) can be built on top of the base set of ISA instructions and techniques needed for the simple single-thread I/D consistency case.

- No additional mechanisms or instructions were required to support the various JIT techniques.

- While not definitive, of course, this is a good sanity check that the basic architecture is robust and complete.

Part IV : Must FENCE.ID be a completion fence?

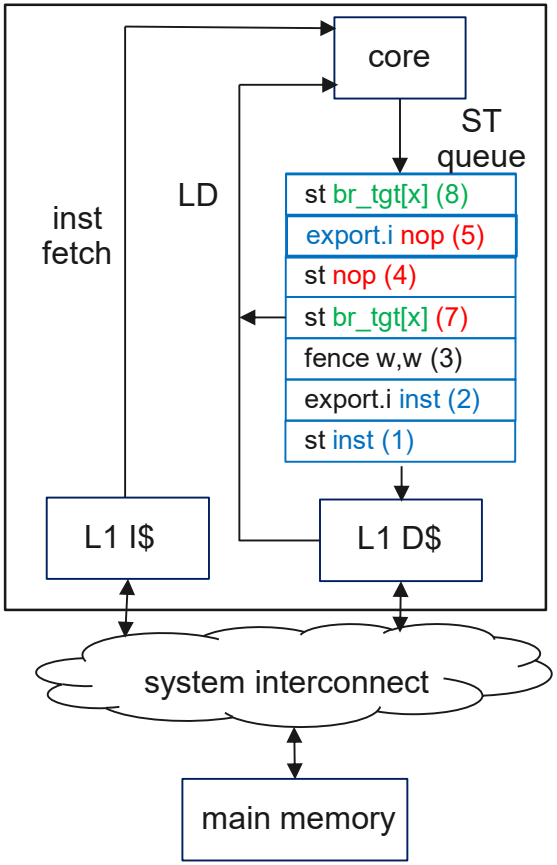
Does FENCE.ID have to be a completion fence?

- Short answer: Yes, yes it does.
- Slightly longer answer: There's a certain amount of stalling that has to happen after the EXPORT.I's in order to make I/D consistency work. If you don't use a completion barrier after the EXPORT.I and before the 'signaling event' (be it a store to update a branch, store to alter entrypoint value in table, etc.) this stalling will have to go somewhere. There are (poor) ways to hide the stalls in the EXPORT.I or in FENCE W,W, but it's just cleaner architecturally (and microarchitecturally) to put the delay where it belongs: in a clearly defined completion barrier. If nothing else, it's MUCH easier to reason about and probably easier to implement correctly as well.
- Done with some care, the completion fence won't cost you much more in the way of stalls or delays that any of the other techniques for hiding the stalls.
- This is easier to demonstrate by looking at some sample microarchitectures.

Microarchitecture of FENCE W,W; EXPORT.I

- If we were to use just an ordering fence in place of FENCE.ID, FENCE W,W would be the logical choice. So, we first set some assumptions about FENCE W,W's implementation.
- FENCE W,W is an ordering-only fence for writes. It says nothing about stores being fully propagated through the system, just that the stores on either side of the FENCE W,W will propagate to the other HARTs in the order implied by the barrier. As such, the usual implementation for FENCE W,W is to place a "fence" operation in the store pipe that follows the same path as stores and prevents reordering across the fence. There is no stalling of subsequent instructions and no "ack" back from the fence.
- We argue that if FENCE W,W is anything more than that, you're doing it wrong!
- Given that assumption, we'll look at how some of the prior sequences would behave with FENCE W,W instead of FENCE.ID.
- For performance, the implementations also need to be able to pipeline EXPORT.I's.

Bait-and-Switch on same core with FENCE W,W



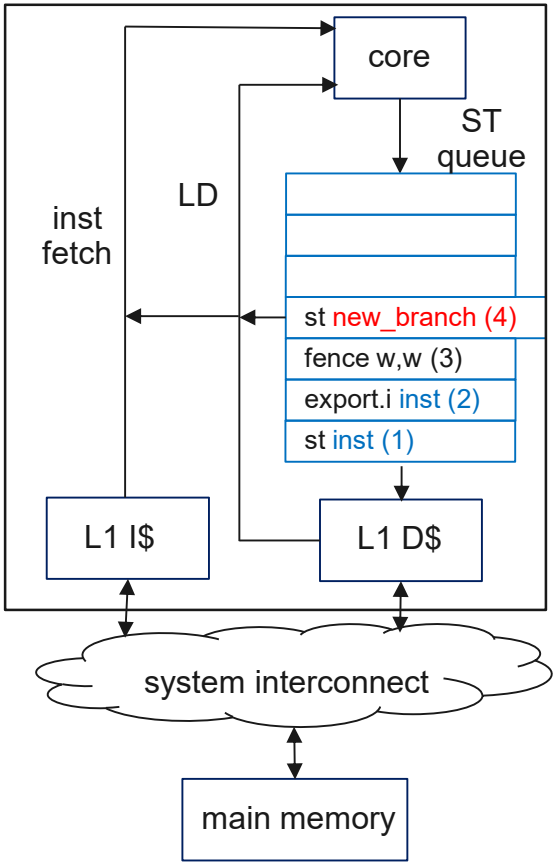
HART 0
ST new_instructions (1)
EXPORT.I new_instructions (2)
FENCE W,W (3)
ST branch_target[x], new (7)
ST NOP (4)
EXPORT.I NOP (5)
LD reg1, branch_target[x] (8)
BR reg1 (9)

- This shows FENCE.ID (3) replaced by FENCE W,W (3) and what can happen.
- Execution has been stopped at the LD reg1, branch_target[x] (8) instruction with the pre-L1 store queue loaded with all the stores, FENCEs, and EXPORT.I's, but none of these have had any effect yet.
- At this point, LD (8) is satisfied from the pre-L1 STQ and BR (9) will then jump off to the wrong instruction image because the EXPORT.I (2) and the ST (1), stuck in the pre-L1 store queue, haven't yet updated memory or the caches.
- Some mechanism must keep LD (8) from seeing ST (7) before the ST (1) and EXPORT.I (2) have had their effects fully propagated. The completion semantics for FENCE.ID when it replaces FENCE W,W (3) prevent LD (8) from seeing ST (7) too early. FENCE W,W does not.

Fixes (poor ones) for not having FENCE.ID here

- For this example, a first way to stop LD (8) from seeing ST (7) too early would be to hide the “stall” inside the EXPORT.I (2). In other words, no store younger than an EXPORT.I would be allowed to begin execution until the EXPORT.I had completed its updates throughout the system.
- That’s a bad plan… you can no longer pipeline EXPORT.I’s now.
- A second way would be to make FENCE W,W (3) perform stalling and ordering functions. If a FENCE W,W has any unfinished older EXPORT.I’s, it will hold up initiation of any younger stores. This gums up the implementation and definition of FENCE W,W. To add insult to injury, that’s just a stall! In fact, the same stall that a completion FENCE.ID would be doing anyways!
- It is simpler, cleaner, and clearer to just admit to the stalling behavior that’s inherently necessary and put it where it belongs: in a dedicated completion barrier -- FENCE.ID.

Branch Modifier example with FENCE W,W



HART 0
ST new_instructions (1)
EXPORT.I new_instructions (2)
FENCE W,W (3)
ST new_branch (4)

- This shows FENCE.ID (3) replaced by FENCE W,W (3) and what can happen. HART1 just executes "BR new_branch" that's been overwritten by ST (4).

- The execution is frozen at the point that HART0 has put all the stores, FENCEs, and EXPORT.I's in the store queue. We further assume that "core" is multi-threaded and that the "core" can selectively bypass i-fetches from the store queue. Yes, this would be illegal for a normal load instruction since it violates multi-copy-atomicity. But, I-fetches are NOT coherent or multi-copy atomic, so it's not illegal ☺. Aren't I-fetches fun?

- HART1 can fetch the old instructions out of the L1 I$ and forward the "br new_branch" out of the L1 store queue. BOOM!

- Some mechanism must keep the I-fetch for HART1 from seeing the "br new_branch" from ST (4) until the I-fetching mechanisms will see the new instructions. FENCE.ID's completion stall normally fixes this.

Fixes (poor ones) for not having FENCE.ID here

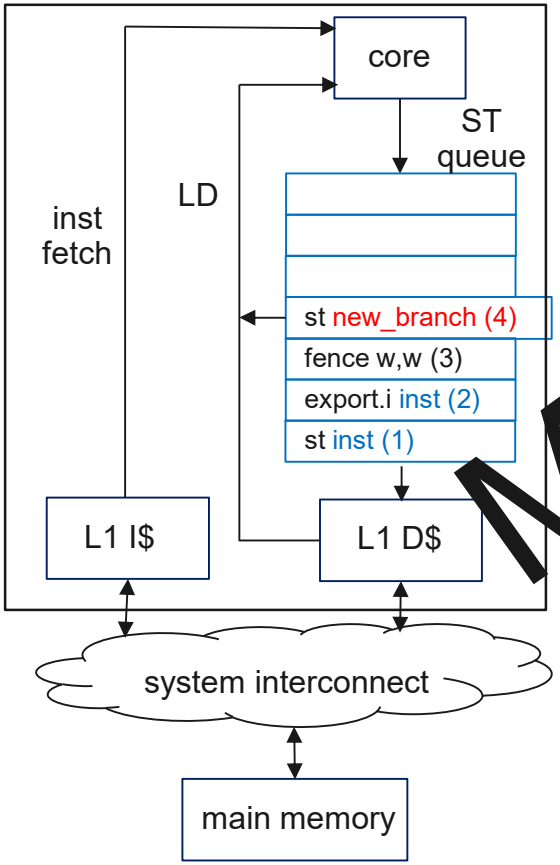
- First off, we must confess the example above is a bit contrived. Fetch the old instructions out of the I-cache while ignoring the store queue and then go pay attention to the store queue to forward the br new from it? Really? Seems stupid to build that.

- It is stupid to build that? Probably…… but it's NOT illegal to build that. Nothing in the architecture today stops it (and nothing ever should).

- So, to fix this, one could add ugly architecture language (that's really microarchitecture language) that says instruction fetches can't read "early" (i.e. before the store is visible to the data side read for a given HART). That partially plugs this one hole. It would also prevent who knows exactly what other sorts of system structures or pipes. See the next page for another place it breaks … and there are more. Just how many of these "patches" are we going to need for more complex structures?

- The simplest answer is to architect a stalling barrier at the FENCE.ID. With that, there's a clear line in the sand where the EXPORT.I's are truly done before the "signaling" store turns a HART loose to go get the "new" code.

Branch Modifier example without FENCE.ID, cont'd



HART 0
ST new_instructions (1)
EXPORT.I new_instructions (2)
FENCE W,W (3)
ST new_branch (4)

- The prior example of how this would break showed the contrived, but not illegal, forwarding I-fetches out of the D-side pre-L1 store queue.
- As a less contrived example of how this could break, assume "core" is single threaded. The execution is shown here stopped after HART0 has placed all the operations in the store queue. Once "st inst" is visible to all HARTs, "fence w,w" can release and stops holding up "st new_branch". The "fence w,w" has no ordering effect on "export.i inst".
- The "st new_branch" bypasses around the "export.i inst" and updates the branch in the L1 D-cache. HART0's I-cache fetches "new_branch" from the L1 D$ (the I-cache does coherent reads, but ignores snoops to invalidate) and executes it. HART0 then branches to the new location, but finds a stale copy of the unmodifed instructions in the L1 I-cache and executes the old instructions. Boom! Even if the L1 I-cache were to only read from main memory, the L1 D-cache could just evict the "new branch" line before the L1 I-cache fetched it.

MAY BE WRONG!

The moral of these examples…

- These examples are just the first ones that come to mind. Doubtless there are many others. Though they will all follow a general theme: the signaling event gets ahead of the visibility of the new code at some HART. The only generic, microarchitecturally agnostic way to solve this problem is to ensure the EXPORT.I's are fully finished everywhere before the signaling event can happen: in other words, a completion fence.

- If one tries to leave it as an "ordering" barrier, the Holy Wars about exactly what "ordering" means for operations like EXPORT.I that have global effects will rage for years. If one tries to disallow specific microarchitectures in the architecture instead, one spends years playing microarchitectural "whack-a-mole" trying to find and patch all the holes. You'll also likely overreach and disallow something that may have been useful.

- A completion fence, however, is simple to understand and just works. The performance isn't significantly worse -- the stall must go somewhere. An architected completion fence doesn't prohibit aggressive implementations -- it just prohibits them from being so aggressive that they're broken. We also question whether there's any useful performance to be gained here by being aggressive. KISS - "Keep It Simple Stupid".

Completion fences more generally

- The subject of so-called completion fences comes up in other contexts.

- For example, a fence needing to ensure that a prior cache flush instruction has pushed the modified line all the way to DRAM (not just to the caches and memory controller) so that an incoherent DMA agent that only reads from the DRAM directly will see the updates.

- In the process of architecting FENCE.ID for I/D consistency, we need to be careful to ensure that the notion of “completion” for FENCE.ID is sensible with whatever other notions of “completion fence” are applied to other fences.

- This does NOT mean we suggest lumping FENCE.ID in with any other FENCE instruction -- just that the notion of “completion” should be as consistent as possible.

Part V : EXPORT.I as two or three instructions?

Originally EXPORT.I was intended to be one instr.

- Originally, EXPORT.I was intended to be one instruction. The justification was that the two parts of EXPORT.I, pushing the data writes to the PoU (called CLEAN.ID) and invalidating all the I-caches (called INVAL.ID), are mostly useless done in isolation.
- The only real way to tell if a CLEAN.ID has done anything is to do an I-fetch and an I-pull on the line after you know the I-cache line has been invalidated. In the absence of such an I-fetch, the CLEAN.ID could be NOPed and there'd be no way to tell. A cacheable read certainly can't tell if the data came from the PoU.
- Similarly, the only way to tell if an INVAL.ID has really been done is to have new data at the PoU before the invalidation and then re-fetch and re-pull it. Without new data at the PoU, a hardware prefetcher could pull the line right back into the I-cache and there'd be no way to know the INVAL.I had ben done.
- Because the CLEAN.ID and the INVAL.ID really need to go together to have any meaningful effect in most cases (see I-cache management below), normally it's best not to code them separately. We initially chose to have one instruction do both.

First reason to split EXPORT.I: line sizes

- The first reason to split EXPORT.I into two instructions is cache line sizes.
- In particular, the Instruction and Data cache line sizes may be different.
- The interconnect protocols will likely only have commands matching the line sizes for caches in question, so an EXPORT.I would have to generate differing numbers of commands for the Instruction and Data commands.
- Impossible to do? No.
- A needless complication for the microarchitecture? Yes.

Second reason to split EXPORT.I: different pipes

- The second reason to split EXPORT.I into discrete instructions is that in some implementations the INVAL.I and the CLEAN.ID may want to or need to flow down separate pipes (e.g. INVAL.I flowing through a non-cacheable unit instead of flowing through the cacheable D-side store pipes. This might be due to keeping the INVAL.I's out of the D-side cacheable machinery due to differing coherence interconnect protocols for INVAL.I's and CLEAN.IDs).

- Having a single instruction cover both functions means that the microarchitecture would have to split the instruction into multiple operations across these pipes (and possibly different numbers of operations if the I and D side line sizes also differed) and having to maintain order between those pipes for these operations.

- Again, it can be implemented, but it's complexity to no good purpose.

Third reason to split EXPORT.I: completion points

- A third reason to split EXPORT.I into discrete instructions is that the point in the microarchitecture where the INVAL.I and the CLEAN.ID finish their effects may vary wildly.

- For example, a CLEAN.ID in an implementation with coherent I-fetches (I-caches are still incoherent because they ignore kills but do fetch coherently) may finish very quickly at an L1 cache that can hold modified data. An INVAL.I will have to broadcast to and clear the I-caches.

- By having separate instructions, the microarchitecture can avoid complexities due to different completion points and can handle each instruction/operation in its own right.

Fourth reason to split EXPORT.I: manage I-caches

- Some applications/circumstances may want a way to manage the I-cache footprint by invalidating lines that the software knows aren't needed in the I-cache anymore.

- Thus, the INVAL.I does have a use in isolation. It would be best not to have to pay the overhead of the CLEAN.ID every time we merely wanted to knock a block out of the I-cache for capacity reasons.

Ordering the two parts of EXPORT.I

- If a single EXPORT.I instruction is used and the microarchitecture requires different operations to represent the push to the PoU and the I-cache invalidation, these operations have to be kept in order.

- In many implementations, keeping these different operations in order will require additional comparators and/or machinery. Again, it can be built, but it's needless complexity that can be avoided.

- A better answer is to use an explicit completion fence (FENCE.ID) between the CLEAN.ID's and the INVAL.I's as a matter of architecture. This is the "big-A" architectural choice that puts the least pressure on the microarchitecture.

Replacement code sequence

- The new proposal, therefore, is to replace this:

```
EXPORT.I
```

- with this:

```
CLEAN.ID        // push data to PoU
FENCE.ID        // be sure that's really done. This is a new use of FENCE.ID
INVAL.I         // go clear the I-caches now that they will fetch from PoU
```

- With the changes to the FENCE.ID semantics described below.

Replacement code sequence, cont'd

- Sequences of multiple EXPORT.I's do NOT need to be replaced with instructions that have a barrier for every CLEAN.ID. Only one barrier is needed to separate the CLEAN.ID's and the INVAL.I's. For example:

This:	Is replaced by this:	NOT this:
EXPORT.I A	CLEAN.ID A	CLEAN.ID A
EXPORT.I B	CLEAN.ID B	**FENCE.ID**
EXPORT.I C	CLEAN.ID C	INVAL.I A
	FENCE.ID	CLEAN.ID B
	INVAL.I A	**FENCE.ID**
	INVAL.I B	INVAL.I B
	INVAL.I C	CLEAN.ID C
		FENCE.ID
		INVAL.I C

New FENCE.ID semantics

- Previously FENCE.ID required that all older EXPORT.I's were complete throughout the system before any younger IMPORT.I begins execution (and flushes the instruction pipe) or any younger ST instruction begins execution.

- Now, FENCE.ID will first need to ensure all older CLEAN.ID's are fully propagated throughout the system before allowing any younger INVAL.I's to being invalidating I-caches. Second, FENCE.ID will need to ensure that all older INVAL.I's are fully propagated throughout the system before allowing any younger IMPORT.I's to begin executing (and flush the instruction pipe) or allowing any younger STs to begin execution (possibly younger loads as well if that's deemed to be necessary later).

New FENCE.ID semantics, cont'd

- The completion fence semantics between older CLEAN.ID's and younger INVAL.I's is simply being used to order older CLEAN.ID's and younger INVAL.I's with as little microarchitectural pressure as possible. The microarchitecture can simply process the older CLEAN.ID's and wait until they are complete everywhere. This provides the ordering desired without having to couple the CLEAN.ID and INVAL.ID in the pipe(s).

- In implementations where ordering between older CLEAN.ID's and younger INVAL.I's is simple to implement, the stalling FENCE.ID semantics can sometimes be elided in the implementation. However, implementations that do not have this ordering naturally can use a simple stalling implementation for the CLEAN.ID to INVAL.ID barrier effect.

- The stalling semantics from older INVAL.I's to younger IMPORT.I's or ST's are much harder to remove and in many cases cannot be removed. Those stalling semantics ensure that the INVAL.ID (and by implication the CLEAN.ID) have been fully propagated to all HARTs before any HART executes the new instruction.

Why no FENCE between ST inst and CLEAN.ID ?

- Why is there no FENCE (be it a FENCE W,W or a FENCE.ID) between the ST's that create the new instructions and the CLEAN.ID's that push those updates to the PoU?
- Simple. In the case of the ST's and the CLEAN.ID's, it is much, much more reasonable to assume these operations are both going down the D-side store pipelines and machinery and can easily be ordered by existing comparators in those pipes.
- As such, keeping these operations in order just by using intra-thread address collisions is the most reasonable ordering choice.
- Unlike CLEAN.ID and INVAL.I that could have significantly different piping, plumbing and completion points, ST's and CLEAN.ID's are almost always going to have the exact same pipes and semantics, so requiring a barrier here is needless overkill. If the pipes/semantics are different, the implementation will have to maintain order (this is rare and not a wise choice anyways).

Part VI : CMODX.

CMODX in one slide (must be expanded later).

- In the WebAssembly example, a store updates a Branch instruction while other HARTs may be executing the Branch instruction. This is an example of a concurrent modification of instructions (CMODX), a technique in which a HART fetches and executes an instruction that has not yet been made fully consistent between the data and instruction fetching mechanism.

- A set of rules for what the fetching mechanisms can see in these cases needs to be added for RISC-V.

- The short, intuitive, and a bit incorrect version of what Power says is that such a fetch can get any value from the set of values written to the instruction between the points that the instruction has been made fully consistent (i.e. if you update a branch to BR A that was made fully consistent to BR B, and then to BR C before a final update to BR D which is made fully consistent, in that interval between A and D being made fully consistent, the I-fetch can get A or B or C and can even get C and then go back to B, or A).

Part VII : Microarchitectural Examples

I/D consistency in sample microarchitectures

- This section is TBD.
- One day this section hopes to include examples of various microarchitectures and how these concepts play out in those microarchitectures. For example, a cache hierarchy where the I-caches and D-caches are split all the way to a last level bus before memory. And in that sort of machine what happens if the I-fetch is coherent at the last level bus vs. always going to main memory. L1/L2 are both writeback at each core and the L1 has split I/D, but the L2 is unified. Consider with both coherent I caches in the L1 and with I-caches in the L1 that are not coherent and are shot down by INVAL.I's. And so on. and So forth. L1 could be writeback and L2 unified. Consider per L1 store queues. Give examples with PoC and PoU in various spots (include incoherent fetch at last-level-bus and how PoU lower than PoC).
- This section is TBD.

Part VIII : Summary.

Summary: I/D consistency

- This section is TBD.
- One day this section hopes to be a brief summary of the list of mechanisms called for in this proposal.
- Roughly: Give one pager semantics for: INVAL.I, CLEAN.ID, FENCE.ID, IMPORT.I. Explain the need for IMPORT.I effects on xRETs or requiring IMPORT.I ahead of xRETs that return to a new software thread of execution.
- I also plan to provide these slides in a standalone separate deck for people who want a “cheat sheet”/”reference card”.
- This section is TBD.

- Backup Material

“Context Synchronizing” – Power ISA definition

1.5.1 Context Synchronization

An instruction or event is *context synchronizing* if it satisfies the requirements listed below. Such instructions and events are collectively called *context synchronizing operations*. The context synchronizing operations are the ***isync*** instruction, the *System Linkage* instructions, the ***mtmsr***[***d***] instructions with L=0, and most interrupts (see Section 6.4).

1. The operation causes instruction dispatching (the issuance of instructions by the instruction fetching mechanism to any instruction execution mechanism) to be halted.

2. The operation is not initiated or, in the case of ***isync***, does not complete, until all instructions that precede the operation have completed to a point at which they have reported all exceptions they will cause.

3. The operation ensures that the instructions that precede the operation will complete execution in the context (privilege, relocation, storage protection, etc.) in which they were initiated, except that the operation has no effect on the context in which the associated Reference and Change bit updates are performed.

4. If the operation directly causes an interrupt (e.g., ***sc*** directly causes a System Call interrupt) or is an interrupt, the operation is not initiated until no exception exists having higher priority than the exception associated with the interrupt (see Section 6.9).

5. The operation ensures that the instructions that follow the operation will be fetched and executed in the context established by the operation. (This requirement dictates that any prefetched instructions be discarded and that any effects and side effects of executing them out-of-order also be discarded, except as described in Section 5.5, “Performing Operations Out-of-Order”.)

Power ISA Ver 3.0B
Book III
Section 1.5.1
Pg. 925

https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

“CMODX” – Power ISA definition

SOMEDAY:

TBD:

ADD CMODX text here.

TBD:

And here….

Power ISA Ver 3.0B
Book III
Section 1.5.1
Pg. 925

https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

- The End.