RISC-V Memory Consistency Model Specification (DRAFT)

The RISC-V Memory Model Task Group
Chair: Daniel Lustig
dlustig@nvidia.com

December 1, 2017

Contents

Draft Status

This document is a draft of the RISC-V memory consistency model specification, plus some accompanying explanatory material. The details, formatting, and commentary are all still subject to change before official ratification takes place. The formalizations are still a work in progress.

If you see any typos, errors, or omissions, or have feedback for how to improve the document, please reach out to the RISC-V memory model task group, to isa-dev, and/or to chair Dan Lustig (dlustig@nvidia.com).

Over the coming weeks, we will aim to merge the memory model specification itself into the User-Level ISA specification, and the rest of the explanatory material will be compiled into a separate document or chapter (details TBD). Official ratification of the User-Level ISA (including the memory model) will take place sometime after that merge is completed.

Chapter 1

The RISC-V Memory Model

This document defines the RISC-V memory consistency model. A memory consistency model is a set of rules that specifies which values can be legally returned by loads of memory. The RISC-V ISA by default uses a memory model called "RVWMO" (RISC-V Weak Memory Ordering). RVWMO is designed to provide lots of flexibility for architects to build high-performance scalable designs while simultaneously providing a relatively sane memory model to programmers.

The RISC-V ISA also provides the optional Ztso extension which imposes the stronger RVTSO (RISC-V Total Store Ordering) memory model for hardware that chooses to provide it. RVTSO provides a simpler programming model but is more restrictive on the implementations that can be legally built.

Under both models, code running on a single hart will appear to execute in order from the perspective of other memory operations in the same hart, but memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order. Therefore, under both models, multithreaded code may require explicit synchronization to guarantee ordering between memory operations from different harts. The base RISC-V ISA provides a `fence` instruction for this purpose, while the optional "A" atomics extension defines load-reserved/store-conditional and atomic read-modify-write operations.

The RVWMO and RVTSO models are formalized as presented, but the interaction of the memory model with I/O, instruction fetches, page table walks, and `sfence.vma` *is not formalized. We may formalize some or all of them in a future revision. The "V" vector, transactional memory, and "J" JIT extensions will need to be incorporated into a future revision as well.*

Memory models which permit memory accesses of different sizes are an active and open area of research, and the specifics of how the RISC-V memory model deals with mixed-size memory accesses is subject to change. Nevertheless, we provide an educated guess at sane rules for such situations as a guideline for architects and implementers.

1.1 Definition of the RVWMO Memory Model

The RVWMO memory model is defined in terms of the *global memory order*, a total ordering of the memory accesses produced by all harts. In general, a multithreaded program will have many different possible executions, and each execution will have its its own corresponding global memory order.

The global memory order is defined in terms of the primitive load(s) and/or store(s) generated by each memory instruction. It is then subject to the constraints defined in the rest of this chapter. Any execution which satisfies all of the memory model constraints is a legal execution (as far as the memory model is concerned).

1.1.1 Memory Model Primitives

The RVWMO memory model is specified in terms of the memory accesses generated by each instruction. The *program order* over memory accesses reflects the order in which the instructions that generate each load and store are originally laid out in that hart's dynamic instruction stream; i.e., the order in which a simple in-order processor would execute the instructions of that hart. Table 1.1 summarizes the memory accesses generated by each type of memory instruction.

RISC-V Instruction	Memory Accesses
`l{b|h|w|d}`	load*
`s{b|h|w|d}`	store*
`lr`	load
`lr.aq`	load-acquire-RCpc
`lr.aqrl`	load-acquire-RCsc
`lr.rl`	(deprecated)
`sc`	store
`sc.rl`	store-release-RCpc
`sc.aqrl`	store-release-RCsc
`sc.aq`	(deprecated)
`amo<op>`	load; <*op*>; store
`amo<op>.aq`	load-acquire-RCpc; <*op*>; store
`amo<op>.rl`	load; <*op*>; store-release-RCpc
`amo<op>.aqrl`	load-SC; <*op*>; store-SC

*: possibly multiple if misaligned

Table 1.1: Mapping of instructions into memory accesses, for the purposes of describing the memory model. "`.sc`" is a synonym for "`.aqrl`".

Every aligned load or store instruction gives rise to exactly one memory access that executes in a single-copy atomic fashion: it can never be observed in a partially-incomplete state. Every misaligned load or store may be decomposed into a set of component loads or stores at any granularity. The memory accesses generated by misaligned loads and stores are not ordered with respect to each other in program order, but they are ordered with respect to the memory accesses generated by preceding and subsequent instructions in program order.

The legal decomposition of unaligned memory operations down to even byte granularity facilitates emulation on implementations that do not natively support unaligned accesses. Such implementations might, for example, simply iterate over the bytes of a misaligned access one by one.

AMOs give rise to exactly two memory accesses: one load and one store. These accesses are said to be *paired*. Every `lr` instruction gives rise to exactly one load. Every `sc` instruction gives rise to either zero stores or one store, depending on whether the store conditional succeeds. An `lr` instruction is said to be paired with the first `sc` instruction that follows it in program order, unless the `sc` is not successful or there is another `lr` instruction in between. The memory accesses generated by paired `lr` and (successful) `sc` instructions are also said to be paired. Both `lr` and `sc` instructions may be unpaired if they do not meet the conditions above. The complete list of conditions determining the success or failure of store conditional instructions is defined in the "A" extension.

Loads and stores generated by atomics may carry ordering annotations. Loads may carry "acquire-RCpc" or "acquire-RCsc" annotations. The term "load-acquire" without further annotation refers to both collectively. Stores may carry "release-RCpc" or "release-RCsc" annotations, and once again "store-release" without further annotation refers to both together. In the memory model literature, the term "RCpc" stands for release consistency with processor-consistent synchronization operations, and the term "RCsc" stands for release consistency with sequentially-consistent synchronization operations. Finally, AMOs with both `.aq` and `.rl` set are sequentially-consistent in an even stronger sense: they do not allow any reordering in either direction. The precise semantics of these annotations as they apply to RVWMO are described by the memory model rules below.

Although the ISA does not currently contain `l{b|h|w|d}.aq[rl]` *or* `s{b|h|w|d}.[aq]rl` *instructions, we may add them as assembler pseudoinstructions to facilitate forwards compatibility with their potential future addition into the ISA. These pseudoinstructions will generally assemble per the fence-based mappings of Section 2.6 until if and when the instructions are added to the ISA. The RVWMO memory model is also designed to easily accommodate the possible future inclusion of such instructions.*

1.1.2 Dependencies

The definition of the RVWMO memory model depends in part on the notion of a syntactic dependency. A register r read by an instruction b has a syntactic dependency on an instruction a if a precedes b in program order, r is not `x0`, and either of the following hold:

1. r is written by a and read by b, and no other instruction between a and b in program order modifies r
2. There is some other instruction i such that a register read by i has a dependency on a, and a register read by b has a dependency on i

Specific types of dependencies are defined as follows. First, for two instructions a and b, b has a syntactic address dependency on a if a register used to calculate the address accessed by b has a syntactic dependency on a. Second, b has a syntactic data dependency on a if b is a store and a register used to calculate the data being written by b has a syntactic dependency on a. Third, b has a syntactic control dependency on a if there exists a branch m between a and b in program order such that a register checked as part of the condition of m has a syntactic dependency on a. Finally, b has a syntactic success dependency on a if a is a store-conditional and a register read by b has a register dependency on the store conditional success register written by a.

1.1.3 Preserved Program Order

The global memory order for any given execution of a program respects some but not necessarily all of each hart's program order. The subset of program order which must be respected by the global memory order for all executions is known as *preserved program order.*

The complete definition of preserved program order is as follows: memory access a precedes memory access b in preserved program order (and hence also in the global memory order) if a precedes b in program order, a and b are both accesses to normal memory (i.e., not to I/O regions), and any of the following hold:

- Basic same-address orderings:

  1. a and b are accesses to overlapping memory addresses, and b is a store

- Explicit synchronization

  2. a and b are separated in program order by a fence, a is in the predecessor set of the fence, and b is in the successor set of the fence
  3. a is a load-acquire
  4. b is a store-release
  5. a is a store-release-RCsc and b is a load-acquire-RCsc
  6. a is a store-SC
  7. b is a load-SC

- Dependencies

  8. a is a load, and b has a syntactic address dependency on a
  9. a is a load, b is a store, and b has a syntactic data dependency on a
  10. a is a load, b is a store, and b has a syntactic control dependency on a
  11. a is a load, and there exists some m such that m has an address or data dependency on a and b has a success dependency on m

- Same-address load-load ordering

  12. a and b are loads to overlapping memory addresses, there is no store to overlapping memory location(s) between a and b in program order, and a and b return values from different stores

- Pipeline dependency artifacts

  13. a and b are loads, and there exists some store m between a and b in program order such that m has an address or data dependency on a, m accesses a memory address that overlaps the address accessed by b, and there is no other store to an overlapping memory location(s) between m and b
  14. b is a store, and there exists some instruction m between a and b in program order such that m has an address dependency on a

15. a and b are loads, b has a syntactic control dependency on a, and there exists a `fence.i` between the branch used to form the control dependency and b in program order
16. a is a load, there exists an instruction m which has a syntactic address dependency on a, and there exists a `fence.i` between m and b in program order

1.1.4 Memory Model Axioms

An execution of a RISC-V program obeys the RVWMO memory consistency model only if it obeys the *load value axiom* and the *atomicity axiom*.

Load Value Axiom: Each byte of each load returns the corresponding byte written by the whichever of the following two stores comes later in the global memory order:

1. the latest store to the same address and preceding the load in the global memory order
2. the latest store to the same address and preceding the load in program order

Atomicity Axiom: If r and w are a paired load and store, and if s is any store from which r returns a value, then there can be no store from another hart to an overlapping memory location which follows both r and s and which precedes w in the global memory order.

1.2 Definition of the RVTSO Memory Model

RISC-V cores which implement Ztso impose RVTSO onto all memory accesses. RVTSO behaves just like RVWMO but with the following modifications:

- All `l`{`b|h|w|d|r`} instructions behave as if `.aq` is set
- All `s`{`b|h|w|d|c`} instructions behave as if `.rl` is set
- All AMO instructions behave as if `.aq` and `.rl` are both set

These rules render PPO rules 1 and 8–16 redundant. They also make redundant any non-I/O fences that do not have both `.pw` and `.sr` set. Finally, they also imply that all AMO instructions are fully-fenced; nothing will be reordered past an AMO.

The definitions of RVTSO and Ztso are new and have not yet been reviewed as carefully as the RVWMO model. For example, it is not yet properly specified how LR/SC will behave under RVTSO.

Chapter 2

RVWMO Explanatory Material

This section provides more explanation for the RVWMO memory model, using more informal language and concrete examples. These are intended to clarify the meaning and intent of the axioms and preserved program order rules.

2.1 Why RVWMO?

Memory consistency models fall along a loose spectrum from weak to strong. Weak memory models (e.g., ARMv7, Power, Alpha) allow more hardware implementation flexibility and deliver arguably better performance, performance per watt, power, scalability, and hardware verification overheads than strong models, at the expense of a more complex programming model. Strong models (e.g., sequential consistency, TSO) provide simpler programming models, but at the cost of imposing more restrictions on the kinds of hardware optimizations that can be performed in the pipeline and in the memory system, with some cost to power and area overheads, and with some added hardware verification burden.

For the base ISA, RISC-V has chosen the RVWMO memory model, which is a variant of release consistency. This places it in between the two extremes of the memory model spectrum. It is not as weak as the Power memory model, and this buys back some programming model simplicity without giving up very much in terms of performance. RVWMO is also not as restrictive as RVTSO, and hence it remains weak enough to ensure that implementations can be performant and scalable without incurring huge hardware complexity overheads. RVWMO is similar to the ARMv8 memory model in this regard.

As such, the RVWMO memory model enables architects to build simple implementations, aggressive implementations, implementations embedded deeply inside a much larger system and subject to complex memory system interactions, or any number of other possibilities, all while simultaneously being strong enough to support programming language memory models at high performance.

The risk of a weak memory model lies in the complexity of the programming model. Buggy code which “just worked” on stronger implementations may well break on more aggressive implementations due to the bugs simply not manifesting on the stronger-than-necessary implementations. For these situations, though, the root cause is the bug in the original software, not the memory model itself. The risk of finding short-term bugs in code ported from other architectures is outweighed by the long-term benefits that the weak memory model delivers more generally.

To mitigate this risk, some hardware implementations may choose to stick with RVTSO, and that is perfectly acceptable and perfectly compatible with the RVWMO memory model. The cost that the weak memory model imposes on such implementations is the incremental overhead of fetching instructions (e.g., `fence r,rw` and `fence rw,w`) which become no-ops on that implementation. (These fences must remain present in the code to ensure compatibility with other more weakly-ordered RISC-V implementations.)

Most software is also fully compatible with weak memory models. C/C++, Java, and Linux, to name some of the most notable and more formally analyzed examples, are all entirely compatible with weak non-atomic memory models, as all are designed to run not just on x86 but also on ARM, Power, and many other architectures. It is true that some code, e.g., code ported from x86, does sometimes (correctly or incorrectly) assume a stronger model such as TSO. For such code, the RVWMO memory model provides a means for restoring TSO to sections of code through fences and atomics with `.aq` and `.rl` bits in the "A" extension, until such code can be ported to RVWMO over time.

Designers who wish to provide drop-in compatibility with x86 code can also implement the Ztso extension which enforces RVTSO. Code written for RVWMO is automatically and inherently compatible with RVTSO, but code written assuming RVTSO is not guaranteed to run correctly on RVWMO implementations. In fact, RVWMO implementations will (and should) simply refuse to run TSO-only binaries. Each implementation must therefore choose whether to prioritize compatibility with RVTSO code (e.g., to facilitate porting from x86) or whether to instead prioritize compatibility with other RISC-V cores implementing RVWMO.

2.2 Litmus Tests

The explanations in this chapter make use of *litmus tests*, or small programs designed to test or highlight one particular aspect of a memory model. Figure 2.1 shows an example of a litmus test with two harts. For this figure (and for all figures that follow in this chapter), we assume that `s0`–`s2` are pre-set to the same value in all harts. As a convention, we will assume that `s0` holds the address labeled `x`, `s1` holds `y`, and `s2` holds `z`, where `x`, `y`, and `z` are different memory addresses. This figure shows the same program twice: on the left in RISC-V assembly, and again on the right in graphical form.

Litmus tests are used to understand the implications of the memory model in specific concrete situations. For example, in the litmus test of Figure 2.1, the final value of `a0` in the first hart can be either 2, 4, or 5, depending on the dynamic interleaving of the instruction stream from each hart at runtime. However, in this example, the final value of `a0` in Hart 0 will never be 1 or 3: the value 1 will no longer be visible at the time the load executes, and the value 3 will not yet be visible by the time the load executes.

We analyze this test and many others below.

```
            Hart 0          ||         Hart 1
            ⋮               ||         ⋮
            li t1, 1        ||         li t4, 4
      (a)   sw t1,0(s0)     ||   (e)   sw t4,0(s0)
            ⋮               ||         ⋮
            li t2, 2        ||
      (b)   sw t2,0(s0)     ||
            ⋮               ||         ⋮
      (c)   lw a0,0(s0)     ||
            ⋮               ||         ⋮
            li t3, 3        ||         li t5, 5
      (d)   sw t3,0(s0)     ||   (f)   sw t5,0(s0)
            ⋮               ||         ⋮
```

(picture coming soon)

Figure 2.1: A sample litmus test

2.3 Explaining the RVWMO Rules

In this section, we provide explanation and examples for all of the RVWMO rules and axioms.

2.3.1 Preserved Program Order and Global Memory Order

Preserved program order represents the set of intra-hart orderings that the hart's pipeline must ensure are maintained as the instructions execute, even in the presence of hardware optimizations that might otherwise reorder those operations. Events from the same hart which are not ordered by preserved program order, on the other hand, may appear reordered from the perspective of other harts and/or observers.

Informally, the global memory order represents the order in which loads and stores perform. The formal memory model literature has moved away from specifications built around the concept of performing, but the idea is still useful for building up informal intuition. A load is said to have performed when its return value is determined. A store is said to have performed not when it has executed inside the pipeline, but rather only when its value has been propagated to globally visible memory. In this sense, the global memory order also represents the contribution of the coherence protocol and/or the rest of the memory system to interleave the (possibly reordered) memory accesses being issued by each hart into a single total order agreed upon by all harts.

The order in which loads perform does not always directly correspond to the relative age of the values those two loads return. In particular, a load b may perform before another load a to the same address (i.e., b may execute before a, and b may appear before a in the global memory order), but a may nevertheless return an older value than b. This discrepancy captures the reordering effects of store buffers placed between the core and memory: a younger load may read from a value in the store buffer, while an older load which appears before that store in program order may ignore that younger store and read an older value from memory instead. To account for this, at the time each load performs, the value it returns is determined by the load value axiom, not just strictly by determining the most recent store to the same address in the global memory order, as described below.

2.3.2 Store Buffering (Load Value Axiom)

> Load value axiom: Each byte of each load returns the corresponding byte written by the whichever of the following two stores comes later in the global memory order:
>
> 1. the latest store to the same address and preceding the load in the global memory order
> 2. the latest store to the same address and preceding the load in program order

Preserved program order is *not* required to respect the ordering of a store followed by a load to an overlapping address. This complexity arises due to the ubiquity of store buffers in nearly all implementations. Informally, the load may perform (return a value) by forwarding from the store while the store is still in the store buffer, and hence before the store itself performs (writes back to globally visible memory). Any other hart will therefore observe the load as performing before the store.

	`Hart 0`		`Hart 1`
	`li t1, 1`		`li t1, 1`
`(a)`	`sw t1,0(s0)`	`(e)`	`sw t1,0(s1)`
`(b)`	`lw a0,0(s0)`	`(f)`	`lw a2,0(s1)`
`(c)`	`fence r,r`	`(g)`	`fence r,r`
`(d)`	`lw a1,0(s1)`	`(h)`	`lw a3,0(s0)`

(picture coming soon)

Figure 2.2: A store buffer forwarding litmus test

Consider the litmus test of Figure 2.2. When running this program on an implementation with store buffers, it is possible to arrive at the final outcome `a0=1, a1=0, a2=1, a3=0` as follows:

- (a) executes and enters the first hart's private store buffer
- (b) executes and forwards its return value 1 from (a) in the store buffer
- (c) executes since all previous loads (i.e., (b)) have completed
- (d) executes and reads the value 0 from memory
- (e) executes and enters the second hart's private store buffer
- (f) executes and forwards its return value 1 from (d) in the store buffer
- (g) executes since all previous loads (i.e., (f)) have completed
- (h) executes and reads the value 0 from memory
- (a) drains from the first hart's store buffer to memory

- (e) drains from the second hart's store buffer to memory

Therefore, the memory model must be able to account for this behavior.

To put it another way, suppose the definition of preserved program order did include the following hypothetical rule: memory access *a* precedes memory access *b* in preserved program order (and hence also in the global memory order) if *a* precedes *b* in program order and *a* and *b* are accesses to the same memory location, *a* is a write, and *b* is a read. Call this "Rule X". Then we get the following:

- (a) precedes (b): by rule X
- (b) precedes (d): by rule 2
- (d) precedes (e): by the load value axiom. Otherwise, if (d) preceded (c), then (d) would be required to return the value 1. (This is a perfectly legal execution; it's just not the one in question)
- (e) precedes (f): by rule X
- (f) precedes (h): by rule 2
- (h) precedes (a): by the load value axiom, as above.

The global memory order must be a total order and cannot be cyclic, because a cycle would imply that every event in the cycle happens before itself, which is impossible. Therefore, the execution proposed above would be forbidden, and hence the addition of rule X would break the memory model.

Nevertheless, even if (b) precedes (a) and/or (f) precedes (e) in the global memory order, the only sensible possibility in this example is for (b) to return the value written by (a), and likewise for (f) and (e). This combination of circumstances is what leads to the second option in the definition of the load value axiom. Even though (b) precedes (a) in the global memory order, (a) will still be visible to (b) by virtue of sitting in the store buffer at the time (b) executes. Therefore, even if (b) precedes (a) in the global memory order, (b) should return the value written by (a) because (a) precedes (b) in program order. Likewise for (e) and (f).

2.3.3 Same-Address Orderings, Part 1 (Rule 1)

> Rule 1: *a* and *b* are accesses to overlapping memory addresses, and *b* is a store

Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation turns out to be invalid, so such behavior is simply disallowed by the model.

Same-address load-load orderings are far more subtle; see Chapter 2.3.7.

2.3.4 Fences (Rule 2)

> Rule 2: *a* and *b* are separated in program order by a fence, *a* is in the predecessor set of the fence, and *b* is in the successor set of the fence

By default, the `fence` instruction ensures that all memory accesses from instructions preceding the fence in program order (the "predecessor set") appear later in the global memory order than memory accesses from instructions appearing after the fence in program order (the "successor set"). However, fences can optionally further restrict the predecessor set and/or the successor set to a smaller set of memory accesses in order to provide some speedup. Specifically, fences have `.pr`, `.pw`, `.sr`, and `.sw` bits which restrict the predecessor and/or successor sets. The predecessor set includes loads (resp. stores) if and only if `.pr` (resp. `.pw`) is set. Similarly, the successor set includes loads (resp. stores) if and only if `.sr` (resp. `.sw`) is set.

The full RISC-V opcode encoding currently has nine non-trivial combinations of the four bits `pr`, `pw`, `sr`, and `sw`, plus one extra encoding which is expected to be added to facilitate mapping of "acquire+release" or TSO semantics. The remaining seven combinations have empty predecessor and/or successor sets and hence are no-ops. Of the ten non-trivial options, only six are commonly used in practice:

- `fence rw,rw`
- `fence.tso` (i.e., a combined `fence r,rw` + `fence rw,w`)
- `fence rw,w`
- `fence r,rw`
- `fence r,r`
- `fence w,w`

We strongly recommend that programmers stick to these six, as these are the best understood. `fence` instructions using any other combination of `.pr`, `.pw`, `.sr`, and `.sw` are reserved.

Finally, we note that since RISC-V uses a multi-copy atomic memory model, programmers can reason about fences and the `.aq` and `.rl` bits in a thread-local manner. There is no complex notion of "fence cumulativity" as found in memory models which are not multi-copy atomic.

2.3.5 Acquire/Release Ordering (Rules 3–7)

> Rule 3: *a* is a load-acquire
> Rule 4: *b* is a store-release
> Rule 5: *a* is a store-release-RCsc and *b* is a load-acquire-RCsc
> Rule 6: *a* is a store-SC
> Rule 7: *b* is a load-SC

An *acquire* operation is used at the start of a critical section. The general requirement for acquire semantics is that all loads and stores inside the critical section are up to date with respect to the synchronization variable being used to protect it. In other words, an acquire operation requires load-to-load/store ordering. Acquire ordering can be enforced in one of two ways: setting `.aq`, which enforces ordering with respect to just the synchronization variable itself, or with a `FENCE r,rw`, which enforces ordering with respect to all previous loads.

```
    sd           x1, (a1)     # Random unrelated store
    ld           x2, (a2)     # Random unrelated load
    li           t0, 1        # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez         t0, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
    sd           x3, (a3)     # Random unrelated store
    ld           x4, (a4)     # Random unrelated load
```

Figure 2.3: A spinlock with atomics

Consider Figure 2.3: Because this example uses `.aq`, the loads and stores in the critical section are guaranteed to appear in the global memory order after the `amoswap` used to acquire the lock. However, assuming `a0`, `a1`, and `a2` point to different memory locations, the loads and stores in the critical section may or may not appear after the "random unrelated load" at the beginning of the example in the global memory order.

```
    sd           x1, (a1)     # Random unrelated store
    ld           x2, (a2)     # Random unrelated load
    li           t0, 1        # Initialize swap value.
again:
    amoswap.w    t0, t0, (a0) # Attempt to acquire lock.
    fence        r, rw        # Enforce "acquire" memory ordering
    bnez         t0, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    fence        rw, w        # Enforce "release" memory ordering
    amoswap.w    x0, x0, (a0) # Release lock by storing 0.
    sd           x3, (a3)     # Random unrelated store
    ld           x4, (a4)     # Random unrelated load
```

Figure 2.4: A spinlock with fences

Now, consider the alternative in Figure 2.4. In this case, even though the `amoswap` does not enforce ordering with an `.aq` bit, the fence nevertheless enforces that the acquire `amoswap` appears earlier in the global memory order than all loads and stores in the critical section. Note, however, that in this case, the fence also enforces additional orderings: it also requires that the "random unrelated load" at the start of the program appears also appears earlier in the global memory order than the loads and stores of the critical section. (This particular fence does not, however, enforce any ordering with respect to the "random unrelated store" at the start of the snippet.) In this way, fence-enforced orderings are slightly coarser than orderings enforced by `.aq`.

Release orderings work exactly the same as acquire orderings, just in the opposite direction. Release semantics require all loads and stores in the critical section to appear before the lock-releasing store (here, an `amoswap`) in the global memory order. This can be enforced using the `.rl` bit or with a `fence rw,w` operations. Likewise, the ordering between the loads and stores in the critical section and the "random unrelated store" at the end of the code snippet is enforced only by the `fence rw,w` in the second example, not by the `.rl` in the first example.

By default, store-release-to-load-acquire ordering is not enforced. This facilitates the porting of code written under the TSO and/or RCpc memory models; see Chapter 2.6 for details. To enforce store-release-to-load-acquire ordering, use store-release-RCsc and load-acquire-RCsc operations, so that PPO rule 5 applies. The use of only store-release-RCsc and load-acquire-RCsc operations implies sequential consistency, as the combination of PPO rules 3–5 implies that all RCsc accesses will respect program order.

AMOs with both `.aq` and `.rl` set are fully-ordered operations. Treating the load part and the store part as independent RCsc operations is not in and of itself sufficient to enforce full fencing behavior, but this subtle weak behavior is counterintuitive and not much of an advantage architecturally, especially with `lr` and `sc` also available. For this reason, AMOs annotated with `.aqrl` are strengthened to being fully-ordered under RVWMO.

2.3.6 Dependencies (Rules 8–11)

Rule 8: a is a load, and b has a syntactic address dependency on a
Rule 9: a is a load, b is a store, and b has a syntactic data dependency on a
Rule 10: a is a load, b is a store, and b has a syntactic control dependency on a
Rule 11: a is a load, and there exists some m such that m has an address or data dependency on a and b has a success dependency on m

Dependencies from a load to a later memory operation in the same hart are respected by the RVWMO memory model. The Alpha memory model was notable for choosing *not* to enforce the ordering of such dependencies, but most modern hardware and software memory models consider allowing dependent instructions to be reordered too confusing and counterintuitive. Furthermore, modern code sometimes intentionally uses such dependencies as a particularly lightweight ordering enforcement mechanism.

Like other modern memory models, the RVWMO memory model uses syntactic rather than semantic dependencies. In other words, this definition depends on the identities of the registers being accessed by different instructions, not the actual contents of those registers. This means that an address, control, or data dependency must be enforced even if the calculation could seemingly be "optimized away". This choice ensures that RVWMO remains compatible with programmers that use these false syntactic dependencies intentionally to form a lightweight type of ordering mechanism.

For example, there is a syntactic address dependency from the first instruction to the last instruction in the Figure 2.5, even though `a1` XOR `a1` is zero and hence has no effect on the address accessed by the second load.

```
ld  a1,0(s0)
xor a2,a1,a1
add s1,s1,a2
ld  a5,0(s1)
```

Figure 2.5: A syntactic address dependency

The benefit of using dependencies as a lightweight synchronization mechanism is that the ordering enforcement requirement is limited only to the specific two instructions in question. Other non-dependent instructions may be freely-reordered by aggressive implementations. One alternative would be to use a load-acquire, but this would enforce ordering for the first load with respect to *all* subsequent instructions. Another would be to use a `fence r,r`, but this would include all previous and all subsequent loads, making this option each more expensive.

Control dependencies behave differently from address and data dependencies in the sense that a control dependency always extends to all instructions following the original target in program order. Consider Figure 2.6: the instruction at `next` will always execute, but it nevertheless still has control dependency from the first instruction.

```
      lw  x1,0(x2)
      bne x1,x0,NEXT
      sw  x3,0(x4)
next: sw  x5,0(x6)
```

Figure 2.6: A syntactic control dependency

```
      lw  x1,0(x2)
      bne x1,x0,NEXT
next: sw  x3,0(x4)
```

Figure 2.7: Another syntactic control dependency

Likewise, consider Figure 2.7. Even though both branch outcomes have the same target, there is still a control dependency from the first instruction in this snippet to the last. This definition of control dependency is subtly stronger than what might be seen in other contexts (e.g., C++), but it conforms with standard definitions of control dependencies in the literature.

	Hart 0		Hart 1
(a)	`ld a0,0(s0)`	(e)	`ld a3,0(s2)`
(b)	`lr a1,0(s1)`	(f)	`sd a3,0(s0)`
(c)	`sc a2,a0,0(s1)`		
(d)	`sd a2,0(s2)`		

(picture coming soon)

Figure 2.8: A variant of the LB litmus test

Finally, we highlight a unique new rule regarding the success registers written by store-conditional instructions. In certain cases, without PPO rule 11, a store conditional could in theory be made to store its own success output value as its data, in a manner reminiscent of so-called out-of-thin-air behavior. This is shown in Figure 2.8. Suppose a hypothetical implementation could occasionally make some early guarantee that a store-conditional operation will succeed. In this case, (c) could return 0 to `a2` early (before actually executing), allowing the sequence (d), (e), (f), (a), and then (b) to execute, and then (c) might execute (successfully) only at that point. This would imply that (c) writes its own success value to `0(s1)`!

To rule out this bizarre behavior, PPO rule 11 says that store-conditional instructions may not return success or failure into the destination register until both the address and data for the instruction have been resolved. In the example above, this would enforce an ordering from (a) to (d), and this would in turn form a cycle that rules out the strange proposed execution.

2.3.7 Same-Address Load-Load Ordering (Rule 12)

> Rule 12: *a* and *b* are loads to overlapping memory addresses, there is no store to overlapping memory location(s) between *a* and *b* in program order, and *a* and *b* return values from different stores

In contrast to same-address orderings ending in a store, same-address load-load ordering requirements are very subtle.

The basic requirement is that a younger load must not return a value which is older than a value returned by an older load in the same hart to the same address. This is often known as "CoRR" (Coherence for Read-Read pairs), or as part of a broader "coherence" or "sequential consistency per location" requirement. Some architectures in the past have relaxed same-address load-load ordering, but in hindsight this is generally considered to complicate the programming model too much, and so RVWMO requires CoRR ordering to be enforced. However, because the global memory order corresponds to the order in which loads perform rather than the ordering of the values being returned, capturing CoRR requirements in terms of the global memory order requires a bit of indirection.

```
        Hart 0      ||         Hart 1
--------------------||--------------------
     li t1, 1       ||      li  t2, 2
(a)  sw t1,0(s0)    || (d)  lw  a0,0(s1)
(b)  fence w, w     || (e)  sw  t2,0(s1)
(c)  sw t1,0(s1)    || (f)  lw  a1,0(s1)
                    || (g)  xor t3,a1,a1
                    || (h)  add s0,s0,t3
                    || (i)  lw  a2,0(s0)
```

(picture coming soon)

Figure 2.9: Litmus test MP+FENCE+fri-rfi-addr

Consider the litmus test of Figure 2.9, which is one particular instance of the more general "fri-rfi" pattern. The term "fri-rfi" refers to the sequence (d),(e),(f): (d) "from-reads" (i.e., reads from an earlier write than) (e) which is the same hart, and (f) reads from (e) which is in the same hart.

From a microarchitectural perspective, outcome `a0=1, a1=2, a2=0` is legal (as are various other less subtle outcomes). Intuitively, the following would produce the outcome in question:

- (a), (b), (c) execute
- (d) stalls (for whatever reason; perhaps it's stalled waiting for some other preceding instruction)
- (e) executes and enters the store buffer
- (f) forwards from (e) in the store buffer
- (g), (h), and (i) execute
- (d) unstalls and executes
- (e) drains from the store buffer to memory

This corresponds to a global memory order of (e),(f),(i),(a),(c),(d). Note that even though (f) performs before (d), the value returned by (f) is newer than the value returned by (d). Therefore, this execution is legal and does not violate the CoRR requirements even though (f) appears before (d) in global memory order.

Likewise, if two back-to-back loads return the values written by the same store, then they may also appear out-of-order in the global memory order without violating CoRR. Note that this is not the same as saying that the two loads return the same value, since two different stores may write the same value. Consider the litmus test of Figure 2.10:

	Hart 0		Hart 1
	`li t1, 1`	(d)	`lw a0,0(s1)`
(a)	`sw t1,0(s0)`	(e)	`xor t2,a0,a0`
(b)	`fence w, w`	(f)	`add s2,s2,t2`
(c)	`sw t1,0(s1)`	(g)	`lw a1,0(s2)`
		(h)	`lw a2,0(s2)`
		(i)	`xor t3,a2,a2`
		(j)	`add s0,s0,t3`
		(k)	`lw a3,0(s0)`

(picture coming soon)

Figure 2.10: Litmus test RSW

The outcome `a0=1,a1=a2,a3=0` can be observed by allowing (g) and (h) to be reordered. This might be done speculatively, and the speculation can be justified by the microarchitecture (e.g., by snooping for cache invalidations and finding none) because replaying (h) after (g) would return the value written by the same store anyway. Hence assuming `a1=a2`, (g) and (h) can be reordered. The global memory order corresponding to this execution would be (h),(k),(a),(c),(d),(g).

Executions of the above test in which `a1` does not equal `a2` do in fact require that (g) appears before (h) in the global memory order. Allowing (h) to appear before (g) in the global memory order would in fact result in a violation of CoRR, because then (h) would return an older value than that returned by (g). Therefore, PPO rule 12 forbids this CoRR violation from occurring. As such, PPO rule 12 strikes a careful balance between enforcing CoRR in all cases while simultaneously being weak enough to permit "RSW" and "fri-rfi" patterns that commonly appear in real microarchitectures.

2.3.8 Atomics and LR/SCs (Atomicity Axiom)

> Atomicity axiom: If r and w are a paired load and store, and if s is any store from which r returns a value, then there can be no store from another hart to an overlapping memory location which follows both r and s and which precedes w in the global memory order.

The RISC-V architecture decouples the notion of atomicity from the notion of ordering. Unlike architectures such as TSO, RISC-V atomics under RVWMO do not impose any ordering requirements by default. Ordering semantics are only guaranteed by the PPO rules that otherwise apply. This relaxed nature allows implementations to be aggressive about forwarding values even before a paired store has been committed to memory.

Roughly speaking, the atomicity rule states that there can be no store from another hart during the time the reservation is held. For AMOs, the reservation is held as the AMO is being performed. For successful `lr`/`sc` pairs, the reservation is held between the time the `lr` is performed and the time the `sc` is performed. In most cases, the atomicity rule states that there can be no store from another hart between the load and its paired store in global memory order.

There is one exception, however: if the paired load returns its value from a store s still in the store buffer (which some implementations may permit), then the reservation may not need to be acquired until s is ready to leave the store buffer, and this may occur after the paired load has already performed. Therefore, in this case, the requirement is only that no other store from another hart to an overlapping address can appear between time that s performs and the time that the paired store performs. Consider the example of Figure 2.11:

	Hart 0			Hart 1	
	`li`	`t1, 2`		`li`	`t3, 2`
	`li`	`t2, 1`		`li`	`t4, 1`
(a)	`sd`	`t1,0(s0)`	(d)	`sd`	`t3,0(s1)`
(b)	`amoor.aq`	`a0,t2,0(s0)`	(e)	`amoswap.rl`	`x0,t4,0(s0)`
(c)	`sd`	`t2,0(s1)`			

(picture coming soon)

Figure 2.11: A litmus test where the reservation for `0(s0)` may not be acquired until after the load of (b) has already completed

The outcome `0(s0)=3, 0(s1)=2` is legal, with the global memory order of (b0),(c),(d),(e),(a),(b1), where (b0) and (b1) represent the load and store parts, respectively, of (b). The atomic operation (b) does not need to grab the reservation until (a) is ready to leave the store buffer. Therefore, although (e) is a store to the same address from another hart, and even though (e) lies between (b0) and (b1) in global memory order, this execution does not violate the atomicity axiom because (e) comes after (a) in global memory order.

```
(a) lr t0, 0(a0)
(b) sd t1, 0(a0)
(c) sc t2, 0(a0)
```

Figure 2.12: Store-conditional (c) may succeed on some implementations

The atomicity rule does not forbid loads from being interleaved between the paired operations in program order or in the global memory order, nor does it forbid stores from the same hart from appearing between the paired operations in either program order or in the global memory order. For example, the sequence in Figure 2.12 is legal, and the `sc` may (but is not guaranteed to) succeed. By preserved program order rule 1, the program order of the three operations must be maintained in the global memory order. This does not violate the atomicity axiom, because the intervening non-conditional store is from the same hart as the paired load-reserved and store-conditional instructions.

```
           Hart 0                          Hart 1
    li t0, 1                   ||
(a) amoor.aq a0,t0,0(s0)       || (c) amoadd.aq a1,x0,0(s1)
(b) sd       a0,0(s1)          || (d) ld        a2,0(s0)
```

(picture coming soon)

Figure 2.13: The `.aq` applies only to the load part of (a), and hence it does not order the store part of (a) before (b)

Likewise, in the test of Figure 2.13, the following global memory order could result in the outcome `a1=1, a2=0`: (a0), (b), (c), (d), (a1).

Overall, the atomicity rule ensures that non-synchronization atomic operations (e.g., incrementing a counter) can be made as efficient as possible in high-performance implementations, while simultaneously ensuring that the atomicity conditions necessary for achieving consensus are maintained.

2.3.9 Pipeline Dependency Artifacts (Rules 13–16)

> Rule 13: a and b are loads, and there exists some store m between a and b in program order such that m has an address or data dependency on a, m accesses a memory address that overlaps the address accessed by b, and there is no other store to an overlapping memory location(s) between m and b
> Rule 14: b is a store, and there exists some instruction m between a and b in program order such that m has an address dependency on a
> Rule 15: a and b are loads, b has a syntactic control dependency on a, and there exists a `fence.i` between the branch used to form the control dependency and b in program order
> Rule 16: a is a load, there exists an instruction m which has a syntactic address dependency on a, and there exists a `fence.i` between m and b in program order

These four "compound dependency" rules reflect behaviors of almost all real processor pipeline implementations, and they are added into the model explicitly to simplify the definition of the formal operational memory model and to improve compatibility with known patterns on other architectures.

```
(a)   lw a0, 0(s0)
(b)   sw a0, 0(s1)
(c)   lw a1, 0(s1)
```

(picture coming soon)

Figure 2.14: Because of the data dependency from (a) to (b), (a) is also ordered before (c)

Rule 13 states that a load forward from a store until the address and data for that store are known.

Consider Figure 2.14: (c) cannot be executed until the data for (b) has been resolved, because (c) must return the value written by (b) (or by something even later in the global memory order). Therefore, (c) will never execute before (a) has executed.

```
     li t1, 1
(a)  lw a0, 0(s0)
(b)  sw a0, 0(s1)
     sw t1, 0(s1)
(c)  lw a1, 0(s1)
```

(picture coming soon)

Figure 2.15: Because of the extra store between (b) and (c), (a) is no longer necessarily ordered before (c)

If there were another store to the same address in between (b) and (c), as in Figure 2.15, then (c) would no longer dependent on the data of (b) being resolved, and hence the dependency of (c) on (a), which produces the data for (b), would be broken.

One subtle related note is that `amoswap` does not contain a data dependency from its load to its store. Nor does every `sc` have a data dependency on its paired `lr`. Therefore, Rule 13 does not enforce an ordering from paired loads of this category to subsequent loads to overlapping addresses.

Rule 14 makes a similar observation to the previous rule: a store cannot be performed at memory until all previous loads which might access the same address have themselves been performed. Such a load must appear to execute before the store, but it cannot do so if the store were to overwrite the value in memory before the load had a chance to read the old value.

```
     li t1, 1
(a)  lw a0, 0(s0)
(b)  lw a1, 0(a0)
(c)  sw t1, 0(s1)
```

(picture coming soon)

Figure 2.16: Because of the address dependency from (a) to (b), (a) is also ordered before (c)

Consider Figure 2.16: (c) cannot be executed until the address for (b) is resolved, because it may turn out that the addresses match; i.e., that `a0=s1`. Therefore, (c) cannot be sent to memory before (a) has executed and confirmed whether the addresses to indeed overlap.

```
      (a)  ld a0, 0(s0)
           xor a1,a0,a0
           bne a1, critical
critical:  fence.i
      (c)  ld a1, 0(s1)
```

(picture coming soon)

Figure 2.17: Because of the control dependency from (a) to (c), (a) is also ordered before (c)

Rule 15 reflects the idiom of Figure 2.17 for a lightweight acquire fence: In this code snippet, (c) cannot execute until the `fence.i` is cleared. The `fence.i` cannot clear until the branch has executed and drained. The branch cannot execute until it receives the value from (a) through the `xor`. Therefore, (a) must be ordered before (c) in the global memory order.

Rule 16 and Figure 2.18 present a similar situation: Once again, (c) cannot execute until the `fence.i` is cleared. The `fence.i` cannot clear until both (a) and (b) have at least issued (even if

```
(a)  ld a0, 0(s0)
(b)  ld a1, 0(a0)
     fence.i
(c)  ld a1, 0(s1)
```

(picture coming soon)

Figure 2.18: Because of the address dependency from (a) to (b) and the `fence.i` between (b) and (c), (a) is also ordered before (c)

they have not yet returned a value). Finally, (b) cannot issue until it receives its address from (a). Therefore, (a) must be ordered before (c).

2.4 FENCE.I, SFENCE.VMA, and I/O Fences

In this section, we provide an informal description of how the `fence.i`, `sfence.vma`, and I/O fences interact with the memory model.

Instruction fetches and address translation operations (where applicable) follow the RISC-V memory model as well as the rules below.

- `fence.i`: Conceptually, `fence.i` ensures that no instructions following the `fence.i` are issued until all instructions prior to the `fence.i` have executed (but not necessarily performed globally). This implies that the fetch of each instruction following the `fence.i` in program order appears later in the global memory order than all stores prior to the `fence.i` in program order. That in turn means that instruction caches which hardware does not keep coherent with normal memory must be flushed when a `fence.i` instruction is executed. (`fence.i` is also used form the patterns of Chapter 2.3.9.)

- `sfence.vma`: Conceptually, the instruction fetch and address translation operations of each instruction following the `sfence.vma` in program order appears later in the global memory order than all stores prior to the `sfence.vma` in program order. This implies that stale entries in the local hart's TLBs must be invalidated.

- Conceptually, updates to the page table made by a hardware page table walker form a paired atomic read-modify-write operation subject to the rules of the atomicity axiom

2.4.1 Coherence and Cacheability

The RISC-V ISA defines Physical Memory Attributes (PMAs) which specify, among other things, whether portions of the address space are coherent and/or cacheable. See the privileged spec for the complete details. Here, we simply discuss how the various details in each PMA relate to the memory model:

- Main memory vs. I/O, and I/O memory ordering PMAs: the memory model as defined applies to main memory regions. I/O ordering is discussed below.

- Supported access types and atomicity PMAs: the memory model is simply applied on top of whatever primitives each region supports.
- Coherence and cacheability PMAs: neither the coherence nor the cacheability PMAs affect the memory model. The RISC-V privileged specification suggests that hardware-incoherent regions of main memory are discouraged, but the memory model is compatible with hardware coherence, software coherence, implicit coherence due to read-only memory, implicit coherence due to only one agent having access, or otherwise. Likewise, non-cacheable regions may have more restrictive behavior than cacheable regions, but the set of allowed behaviors does not change regardless.
- Idempotency PMAs: Idempotency PMAs are used to specify memory regions for which loads and/or stores may have side effects, and this in turn is used by the microarchitecture to determine, e.g., whether prefetches are legal. This distinction does not affect the memory model.

2.4.2 I/O Ordering

For I/O, the load value axiom and atomicity axiom in general do not apply, as both reads and writes might have device-specific side effects. The preserved program order rules do not generally apply to I/O either. Instead, we informally say that memory access a is ordered before memory access b if a precedes b in program order and one or more of the following holds:

1. a and b are accesses to overlapping addresses in an I/O region
2. a and b are accesses to the same strongly-ordered I/O region
3. a and b are accesses to I/O regions, and the channel associated with the I/O region accessed by either a or b is channel 1
4. a and b are accesses to I/O regions associated with the same channel (except for channel 0)
5. a and b are separated in program order by a FENCE, a is in the predecessor set of the FENCE, and b is in the successor set of the FENCE. The predecessor and successor sets include the sets described by all eight FENCE bits `.pr`, `.pw`, `.pi`, `.po`, `.sr`, `.sw`, `.si`, and `.so`.
6. a and b are accesses to I/O regions, and a has `.aq` set
7. a and b are accesses to I/O regions, and b has `.rl` set
8. a and b are accesses to I/O regions, and a and b both have `.aq` and `.rl` set
9. a and b are accesses to I/O regions, and a is an AMO that has `.aq` and `.rl` set
10. a and b are accesses to I/O regions, and b is an AMO that has `.aq` and `.rl` set

As described above, accesses to I/O memory require stronger synchronization that what is enforced by the RVWMO PPO rules. For such cases, `FENCE` operations with `.pi`, `.po`, `.si`, and/or `.so` are needed. For example, to enforce ordering between a write to normal memory and an MMIO write to a device register, a `FENCE w,o` or stronger is needed. Even `.aq` and `.rl` do not enforce ordering between normal memory accesses and accesses to I/O memory. When a fence is in fact used, implementations must assume that the device may attempt to access memory immediately after receiving the MMIO signal, and subsequent memory accesses from that device to memory must observe the effects of all accesses ordered prior to that MMIO operation.

```
sd t0, 0(a0)
fence w,o
sd a0, 0(a1)
```

Figure 2.19: Ordering memory and I/O accesses

In other words, in Figure 2.19, suppose `0(a0)` is in normal memory and `0(a1)` is the address of a device register in I/O memory. If the device accesses `0(a0)` upon receiving the MMIO write, then that load must conceptually appear after the first store to `0(a0)` according to the rules of the RVWMO memory model. In some implementations, the only way to ensure this will be to require that the first store does in fact complete before the MMIO write is issued. Other implementations may find ways to be more aggressive, while others still may not need to do anything different at all for I/O and normal memory accesses. Nevertheless, the RVWMO memory model does not distinguish between these options; it simply provides an implementation-agnostic mechanism to specify the orderings that must be enforced.

Many architectures include separate notions of "ordering" and "completion" fences, especially as it relates to I/O (as opposed to normal memory). Ordering fences simply ensure that memory operations stay in order, while completion fences ensure that predecessor accesses have all completed before any successors are made visible. RISC-V does not explicitly distinguish between ordering and completion fences. Instead, this distinction is simply inferred from different uses of the FENCE bits.

For implementations that conform to the RISC-V Unix Platform Specification, I/O devices, DMA operations, etc. are required to access memory coherently and via strongly-ordered I/O channels. Therefore, accesses to normal memory regions that are shared with I/O devices can also use the standard synchronization mechanisms. Implementations which do not conform to the Unix Platform Specification and/or in which devices do not access memory coherently will need to use platform-specific mechanisms (such as cache flushes) to enforce coherency.

I/O regions in the address space should be considered non-cacheable regions in the PMAs for those regions. Such regions can be considered coherent by the PMA if they are not cached by any agent.

The ordering guarantees in this section may not apply beyond a platform-specific boundary between the RISC-V cores and the device. In particular, I/O accesses sent across an external bus (e.g., PCIe) may be reordered before they reach their ultimate destination. Ordering must be enforced in such situations according to the platform-specific rules of those external devices and buses.

2.5 Code Examples

2.5.1 Compare and Swap

An example using `lr/sc` to implement a compare-and-swap function is shown in Figure 2.20. If inlined, compare-and-swap functionality need only take three instructions.

```
    # a0 holds address of memory location
    # a1 holds expected value
    # a2 holds desired value
    # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)        # Load original value.
    bne t0, a1, fail     # Doesn't match, so fail.
    sc.w a0, a2, (a0)    # Try to update.
    jr ra                # Return.
fail:
    li a0, 1             # Set return to failure.
    jr ra                # Return.
```

Figure 2.20: Sample code for compare-and-swap function using `lr/sc`.

2.5.2 Spinlocks

An example code sequence for a critical section guarded by a test-and-set spinlock is shown in Figure 2.21. Note the first AMO is marked `.aq` to order the lock acquisition before the critical section, and the second AMO is marked `.rl` to order the critical section before the lock relinquishment.

```
    li           t0, 1        # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez         t0, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Figure 2.21: Sample code for mutual exclusion. `a0` contains the address of the lock.

2.6 Code Porting Guidelines

Normal x86 loads and stores are all inherently acquire and release operations: TSO enforces all load-load, load-store, and store-store ordering by default. All TSO loads must be mapped onto `l{b|h|w|d}; fence r,rw`, and all TSO stores must either be mapped onto `amoswap.rl x0` or onto `fence rw,w; s{b|h|w|d}`. Alternatively, TSO loads and stores can be mapped onto `l{b|h|w|d}.aq` and `s{b|h|w|d}.rl` assembler pseudoinstructions to facilitate forwards compatibility in case such instructions are added to the ISA one day. However, in the meantime, the assembler will generate the same fence-based and/or `amoswap`-based versions for these pseudoinstructions. x86 atomics using the LOCK prefix are all sequentially consistent and when ported naively to RISC-V must be marked as `.aqrl`.

A Power `sync/hwsync` fence, an ARM `dmb` fence, and an x86 `mfence` are all equivalent to a RISC-V `fence rw,rw`. Power `isync` and ARM `isb` map to RISC-V `fence.i`. A Power `lwsync` map onto `fence.tso`, or onto `fence rw,rw` when `fence.tso` is not available. ARM `dmb ld` and `dmb st` fences map to RISC-V `fence r,rw` and `fence w,w`, respectively.

A direct mapping of ARMv8 atomics that maps unordered instructions to unordered instructions, RCpc instructions to RCpc instructions, and RCsc instructions to RCsc instructions is likely to work in the majority of cases. Mapping even unordered load-reserved instructions onto `lr.aq` (particularly for LR/SC pairs without internal data dependencies) is an even safer bet, as this ensures C/C++ release sequences will be respected. However, due to a subtle mismatch between the two models, strict theoretical compatibility with the ARMv8 memory model requires that a naive mapping translate all ARMv8 store conditional and load-acquire operations map onto RISC-V RCsc operations. Any atomics which are naively ported into RCsc operations may revert back to the straightforward mapping if the programmer can verify that the code is not relying on an ordering from the store-conditional to the load-acquire (as this is not common).

The Linux fences `smp_mb()`, `smp_wmb()`, `smp_rmb()` map onto `fence rw,rw`, `fence w,w`, and `fence r,r`, respectively. The fence `smp_read_barrier_depends()` map to a no-op due to preserved program order rules 8–10. The Linux fences `dma_rmb()` and `dma_wmb()` map onto `fence r,r` and `fence w,w`, respectively, since the RISC-V Unix Platform requires coherent DMA. The Linux fences `rmb()`, `wmb()`, and `mb()` map onto `fence ri,ri`, `fence wo,wo`, and `fence rwio,rwio`, respectively.

C/C++ Construct	RVWMO Mapping
Non-atomic load	`l{b|h|w|d}`
`atomic_load(memory_order_relaxed)`	`l{b|h|w|d}`
`atomic_load(memory_order_acquire)`	`l{b|h|w|d}; fence r,rw`
`atomic_load(memory_order_seq_cst)`	`fence rw,rw; l{b|h|w|d}; fence r,rw`
Non-atomic store	`s{b|h|w|d}`
`atomic_store(memory_order_relaxed)`	`s{b|h|w|d}`
`atomic_store(memory_order_release)`	`fence rw,w; s{b|h|w|d}`
`atomic_store(memory_order_seq_cst)`	`fence rw,rw; s{b|h|w|d}`
`atomic_thread_fence(memory_order_acquire)`	`fence r,rw`
`atomic_thread_fence(memory_order_release)`	`fence rw,w`
`atomic_thread_fence(memory_order_acq_rel)`	`fence.tso`
`atomic_thread_fence(memory_order_seq_cst)`	`fence rw,rw`

Table 2.1: Mappings from C/C++ primitives to RISC-V primitives.

The C11/C++11 `memory_order_*` primitives should be mapped as shown in Table 2.1. The `memory_order_acquire` orderings in particular must use fences rather than atomics to ensure that release sequences behave correctly even in the presence of `amoswap`. The `memory_order_release` mappings may use `.rl` as an alternative.

Ordering Annotation	Fence-based Equivalent
`l{b|h|w|d|r}.aq`	`l{b|h|w|d|r}; fence r,rw`
`l{b|h|w|d|r}.aqrl`	`fence rw,rw; l{b|h|w|d|r}; fence r,rw`
`s{b|h|w|d|c}.rl`	`fence rw,w; s{b|h|w|d|c}`
`s{b|h|w|d|c}.aqrl`	`fence rw,w; s{b|h|w|d|c}`
`amo<op>.aq`	`amo<op>; fence r,rw`
`amo<op>.rl`	`fence rw,w; amo<op>`
`amo<op>.aqrl`	`fence rw,rw; amo<op>; fence rw,rw`

Table 2.2: Mappings from `.aq` and/or `.rl` to fence-based equivalents. An alternative mapping places a `fence rw,rw` after the existing `s{b|h|w|d|c}` mapping rather than at the front of the `l{b|h|w|d|r}` mapping.

It is also safe to translate any `.aq`, `.rl`, or `.aqrl` annotation into the fence-based snippets of Table 2.2. These can also be used as a legal implementation of `l{b|h|w|d}` or `s{b|h|w|d}` pseudoinstructions for as long as those instructions are not added to the ISA.

2.7 Implementation Guidelines

The RVWMO and RVTSO memory models by no means preclude microarchitectures from employing sophisticated speculation techniques or other forms of optimization in order to deliver higher performance. The models also do not impose any requirement to use any one particular cache hierarchy, nor even to use a cache coherence protocol at all. Instead, these models only specify the behaviors that can be exposed to software. Microarchitectures are free to use any pipeline design, any coherent or non-coherent cache hierarchy, any on-chip interconnect, etc., as long as the design satisfy the memory model rules. That said, to help people understand the actual implementations of the memory model, in this section we provide some guidelines below on how architects and programmers should interpret the models' rules.

Both RVWMO and RVTSO are multi-copy atomic (or "other-multi-copy-atomic"): any store value which is visible to a hart other than the one that originally issued it must also be conceptually visible to all other harts in the system. In other words, harts may forward from their own previous stores before those stores have become globally visible to all harts, but no other early intra-hart forwarding is permitted. Multi-copy atomicity may be enforced in a number of ways. It might hold inherently due to the physical design of the caches and store buffers, it may be enforced via a single-writer/multiple-reader cache coherence protocol, or it might hold due to some other mechanism.

Although multi-copy atomicity does impose some restrictions on the microarchitecture, it is one of the key properties keeping the memory model from becoming extremely complicated. For example, a hart may not legally forward a value from a neighbor hart's private store buffer, unless those two harts are the only two in the system. Nor may a cache coherence protocol forward a value from one hart to another until the coherence protocol has invalidated all older copies from other caches. Of course, microarchitectures may (and high-performance implementations likely will) violate these rules under the covers through speculation or other optimizations, as long as any non-compliant behaviors are not exposed to the programmer.

As a rough guideline for interpreting the PPO rules in RVWMO, we expect the following from the software perspective:

- programmers will use PPO rules 1–7 regularly and actively.
- expert programmers will use PPO rules 8–10 to speed up critical paths of important data structures.
- even expert programmers will rarely if ever use PPO rules 11–16 directly. These are included to facilitate common microarchitectural optimizations (rule 12) and the operational formal modeling approach (rules 11 and 13–16) described in Chapter 3.3. They also facilitate the process of porting code from other architectures which have similar rules.

We also expect the following from the hardware perspective:

- PPO rules 1–4 and 6–7 reflect well-understood rules that should pose few surprises to architects.
- PPO rule 5 may not be immediately obvious to architects, but is somewhat standard nevertheless
- The load value axiom, the atomicity axiom, and PPO rules 8–10 and 13–16 reflect rules that most hardware implementations will enforce naturally, unless they contain extreme optimizations. Of course, implementations should make sure to double check these rules nevertheless. Hardware must also ensure that syntactic dependencies are not "optimized away".
- PPO rule 11 is not obvious, but it is necessary to avoid certain out-of-thin-air-like behavior that appears with store-conditional success values
- PPO rule 12 reflects a natural and common hardware optimization, but one that is very subtle and hence is worth double checking carefully.

Architectures are free to implement any of the memory model rules as conservatively as they choose. For example, a hardware implementation may choose to do any or all of the following:

- interpret all fences as if they were `fence rw,rw` (or `fence iorw,iorw`, if I/O is involved), regardless of the bits actually set
- implement all fences with `.pw` and `.sr` as if they were `fence rw,rw` (or `fence iorw,iorw`, if I/O is involved), as "`w,r`" is the most expensive of the four possible normal memory orderings anyway
- ignore any addresses passed to a fence instruction and simply implement the fence for all addresses
- implement an instruction with `.aq` set as being preceded immediately by `fence r,rw`

- implement an instruction with `.rl` set as being succeeded immediately by `fence rw,w`
- enforcing all same-address load-load ordering, even in the presence of patterns such as "fri-rfi" and "RSW"
- forbid any forwarding of a value from a store in the store buffer to a subsequent AMO or `lr` to the same address
- forbid any forwarding of a value from an AMO or `sc` in the store buffer to a subsequent load to the same address
- implement TSO on all memory accesses, and ignore any normal memory fences that do not include "`w,r`" ordering
- implement all atomics to be RCsc; i.e., always enforce all store-release-to-load-acquire ordering

Architectures which implement RVTSO can safely do the following:

- Ignore all `.aq` and `.rl` bits, since these are implicitly always set under RVTSO. (`.aqrl` cannot be ignored, however, due to PPO rules 5–7.)
- Ignore all fences which do not have both `.pw` and `.sr` (unless the fence also orders I/O)
- Ignore PPO rules 1 and 8–16, since these are redundant with other PPO rules under RVTSO assumptions

Other general notes:

- Silent stores (i.e., stores which write the same value that already exists at a memory location) do not have any special behavior from a memory model point of view. Microarchitectures that attempt to implement silent stores must take care to ensure that the memory model is still obeyed, particularly in cases such as RSW (Chapter 2.3.7) which tend to be incompatible with silent stores.
- Writes may be merged (i.e., two consecutive writes to the same address may be merged) or subsumed (i.e., the earlier of two back-to-back writes to the same address may be elided) as long as the resulting behavior does not otherwise violate the memory model semantics.

The question of write subsumption can be understood from the following example:

	Hart 0		Hart 1
	`li t1, 3`		`li t3, 2`
	`li t2, 1`		
(a)	`sw t1,0(s0)`	(d)	`lw a0,0(s1)`
(b)	`fence w, w`	(e)	`sw a0,0(s0)`
(c)	`sw t2,0(s1)`	(f)	`lw t3,0(s0)`

(picture coming soon)

Figure 2.22: Write subsumption litmus test

As written, (a) must follow (f) in the global memory order:

- (a) follows (c) in the global memory order because of rule 2
- (c) follows (d) in the global memory order because of the Load Value axiom
- (d) follows (e) in the global memory order because of rule 7
- (e) follows (f) in the global memory order because of rule 1

A very aggressive microarchitecture might erroneously decide to discard (e), as (f) supersedes it, and this may in turn lead the microarchitecture to break the now-eliminated dependency between (d) and (f) (and hence also between (a) and (f)). This would violate the memory model rules, and hence it is forbidden. Write subsumption may in other cases be legal, if for example there were no data dependency between (d) and (e).

2.8 Summary of New/Modified ISA Features

At a high level, PPO rules 5, 11, and 13–16 are all new rules that did not exist in the original ISA spec. Rule 12 and the specifics of the atomicity axiom were addressed but not stated in detail.

Other new or modified ISA details:

- There is an RCpc (`.aq` and `.rl`) vs. RCsc (`.aqrl`) distinction
- Load-release and store-acquire are deprecated
- `lr`/`sc` behavior was clarified

2.8.1 Possible Future Extensions

We expect that any or all of the following possible future extensions would be compatible with the RVWMO memory model:

- 'V' vector ISA extensions
- A transactional memory subset of the 'T' ISA extension
- 'J' JIT extension
- Native encodings for `l{b|h|w|d}.aq`/`s{b|h|w|d}.rl`
- Fences limited to certain addresses
- Cache writeback/flush/invalidate/etc. hints, but these should be considered hints, not functional requirements. Any cache management operations which are required for basic correctness should be described as (possibly address range-limited) fences to comply with the RISC-V philosophy (see also `fence.i` and `sfence.vma`). For example, a functional cache writeback instruction might instead be written as "`fence rw[addr],w[addr]`".

2.9 Litmus Tests

These litmus tests represent some of the better-known litmus tests in the field, plus some tests that are randomly-generated, plus some tests that are generated to be particularly relevant to the RVWMO memory model.

All will be made available for download once they are generated.

We expect that these tests will one day serve as part of a compliance test suite, and we expect that many architects will use them for verification purposes as well.

COMING SOON!

Chapter 3

Formal Memory Model Specifications

To facilitate formal analysis of RVWMO, we present a set of formalizations in this chapter. Any discrepancies are unintended; the expectation is that the models will describe exactly the same sets of legal behaviors, pending some memory model changes that have not quite been added to all of the formalizations yet.

As such, these formalizations should be considered snapshots from some point in time during the development process rather than finalized specifications.

At this point, no individual formalization is considered authoritative, but we may designate one as such in collaboration with the ISA specification and/or formalization task groups.

3.1 Formal Axiomatic Specification in Alloy

We present two formal specifications of the RVWMO memory model in Alloy (`http://alloy.mit.edu`).

The first corresponds directly to the natural language model earlier in this chapter.

```
////////////////////////////////////////////////////////////////////////////////
// =RISC-V RVWMO axioms=

// Preserved Program Order
fun ppo : Event->Event {
  // same-address ordering
  po_loc :> Store

  // explicit synchronization
  + ppo_fence
  + Load.aq <: ^po
  + ^po :> Store.rl
  + Store.aq.rl <: ^po :> Load.aq.rl
  + ^po :> Load.sc
  + Store.sc <: ^po

  // dependencies
  + addr
  + data
  + ctrl :> Store
  + (addr+data).successdep

  // CoRR
  + rdw & po_loc_no_intervening_write

  // pipeline dependency artifacts
  + (addr+data).rfi
  + addr.^po :> Store
  + ctrl.(FenceI <: ^po)
  + addr.^po.(FenceI <: ^po)
}

// the global memory order respects preserved program order
fact { ppo in gmo }
```

Figure 3.1: The RVWMO memory model formalized in Alloy (1/4: PPO)

```
// Load value axiom
fun candidates[r: Load] : set Store {
  (r.~gmo & Store & same_addr[r]) // writes preceding r in gmo
  + (r.^~po & Store & same_addr[r]) // writes preceding r in po
}

fun latest_among[s: set Event] : Event { s - s.~gmo }

pred LoadValue {
  all w: Store | all r: Load |
    w->r in rf <=> w = latest_among[candidates[r]]
}

fun after_reserve_of[r: Load] : Event { latest_among[r + r.~rf].gmo }

pred Atomicity {
  all r: Store.~rmw |                 // starting from the read r of an atomic,
    no x: Store & same_addr[r + r.rmw] | // there is no write x to the same addr
      x not in same_hart[r]           // from a different hart, such that
      and x in after_reserve_of[r]    // x follows (the write r reads from) in gmo
      and r.rmw in x.gmo              // and r follows x in gmo
}

pred RISCV_mm { LoadValue and Atomicity }
```

Figure 3.2: The RVWMO memory model formalized in Alloy (2/4: Axioms)

```
////////////////////////////////////////////////////////////////////////////////
// Basic model of memory

sig Hart {  // hardware thread
  start : one Event
}
sig Address {}
abstract sig Event {
  po: lone Event // program order
}

abstract sig MemoryEvent extends Event {
  address: one Address,
  aq: lone MemoryEvent, // opcode bit
  rl: lone MemoryEvent, // opcode bit
  sc: lone MemoryEvent, // for AMOs with .aq and .rl, to distinguish from lr/sc
  gmo: set MemoryEvent   // global memory order
}
sig Load extends MemoryEvent {
  addr: set Event,
  ctrl: set Event,
  data: set Store,
  successdep: set Event,
  rmw: lone Store
}
sig Store extends MemoryEvent {
  rf: set Load
}
sig Fence extends Event {
  pr: lone Fence, // opcode bit
  pw: lone Fence, // opcode bit
  sr: lone Fence, // opcode bit
  sw: lone Fence  // opcode bit
}
sig FenceI extends Event {}

// FENCE PPO
fun FencePRSR : Fence { Fence.(pr & sr) }
fun FencePRSW : Fence { Fence.(pr & sw) }
fun FencePWSR : Fence { Fence.(pw & sr) }
fun FencePWSW : Fence { Fence.(pw & sw) }

fun ppo_fence : MemoryEvent->MemoryEvent {
    (Load  <: ^po :> FencePRSR).(^po :> Load)
  + (Load  <: ^po :> FencePRSW).(^po :> Store)
  + (Store <: ^po :> FencePWSR).(^po :> Load)
  + (Store <: ^po :> FencePWSW).(^po :> Store)
}
```

Figure 3.3: The RVWMO memory model formalized in Alloy (3/4: model of memory)

```
// auxiliary definitions
fun po_loc_no_intervening_write : MemoryEvent->MemoryEvent {
  po_loc - ((po_loc :> Store).po_loc)
}

fun RFInit : Load { Load - Store.rf }
fun rsw : Load->Load { ~rf.rf + (RFInit <: address.~address :> RFInit) }
fun rdw : Load->Load { (Load <: po_loc :> Load) - rsw }

fun po_loc : Event->Event { ^po & address.~address }
fun same_hart[e: Event] : set Event { e + e.^~po + e.^po }
fun same_addr[e: Event] : set Event { e.address.~address }

// basic facts about well-formed execution candidates
fact { acyclic[po] }
fact { all e: Event | one e.*~po.~start }  // each event is in exactly one hart
fact { rf.~rf in iden } // each read returns the value of only one write
fact { total[gmo, MemoryEvent] } // gmo is a total order over all MemoryEvents

//rf
fact { rf in address.~address }
fun rfi : Store->Load { Store <: po_loc_no_intervening_write :> Load }

//dep
fact { addr + ctrl + data in ^po }
fact { successdep in (Write.~rmw) <: ^po }
fact { ctrl.*po in ctrl }
fact { rmw in ^po }

////////////////////////////////////////////////////////////////////////////////
// =Opcode encoding restrictions=

// opcode bits are either set (encoded, e.g., as f.pr in iden) or unset
// (f.pr not in iden).  The bits cannot be used for anything else
fact { pr + pw + sr + sw + aq + rl + sc in iden }
fact { sc in aq + rl }
fact { Load.sc.rmw in Store.sc and Store.sc.~rmw in Load.sc }

// Fences must have either pr or pw set, and either sr or sw set
fact { Fence in Fence.(pr + pw) & Fence.(sr + sw) }

// there is no write-acquire, but there is write-strong-acquire
fact { Store & Acquire in Release }
fact { Load & Release in Acquire }

////////////////////////////////////////////////////////////////////////////////
// =Alloy shortcuts=
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
  all disj e, e': bag | e->e' in rel + ~rel
  acyclic[rel]
}
```

Figure 3.4: The RVWMO memory model formalized in Alloy (4/4: Auxiliaries)

The second is an equivalent formulation which is slightly more complex but which is more computationally efficient. We expect that analysis tools will be built off of this second formulation. Also included are empirical checks that the two models match.

This formulation, however, does not apply when mixed-size accesses are used, nor when `lr`/`sc` to different addresses are used.

```
// coherence order: a total order on the writes to each address
fun co : Write->Write { Write <: ((address.~address) & gmo) :> Write }
// from-read: from a read to the coherence successors of the rf-source of the write
fun fr : Read->Write { ~rf.co + ((Read - Write.rf) <: address.~address :> Write) }

// e = external; i.e., from a different hart
fun rfe : Store->Load  { rf - iden - ^po - ^~po }
fun coe : Store->Store { co - iden - ^po - ^~po }
fun fre :  Load->Store { fr - iden - ^po - ^~po }

pred sc_per_location  { acyclic[rf + co + fr + po_loc] }
pred atomicity { no rmw & fre.coe }
pred causality { acyclic[rfe + co + fr + ppo] }

// equality checks
run RISCV_mm_com_sanity { RISCV_mm_com } for 3
check RISCV_mm_gmo_com { RISCV_mm => RISCV_mm_com } for 6
check RISCV_mm_com_gmo {
  rmw in address.~address => // the rf/co/fr model assumes rmw in same addr
  RISCV_mm_com =>
  rfe + co + fr in gmo =>    // pick a gmo which matches rfe+co+fr
  RISCV_mm
} for 6
```

Figure 3.5: An alternative, more computationally efficient but less complete axiomatic definition

3.2 Formal Axiomatic Specification in Herd

See also: `http://moscova.inria.fr/~maranget/cats7/riscv`

(This herd model is not yet updated to account for rules 6–7 and 11, and rule `r4` has been tentatively removed from RVWMO. Updates to come...)

```
(*************)
(* Utilities *)
(*************)

let fence.r.r = [R];fencerel(Fence.r.r);[R]
let fence.r.w = [R];fencerel(Fence.r.w);[W]
let fence.r.rw = [R];fencerel(Fence.r.rw);[M]
let fence.w.r = [W];fencerel(Fence.w.r);[R]
let fence.w.w = [W];fencerel(Fence.w.w);[W]
let fence.w.rw = [W];fencerel(Fence.w.rw);[M]
let fence.rw.r = [M];fencerel(Fence.rw.r);[R]
let fence.rw.w = [M];fencerel(Fence.rw.w);[W]
let fence.rw.rw = [M];fencerel(Fence.rw.rw);[M]

let fence =
  fence.r.r | fence.r.w | fence.r.rw |
  fence.w.r | fence.w.w | fence.w.rw |
  fence.rw.r | fence.rw.w | fence.rw.rw


let po-loc-no-w = po-loc \ (po-loc;[W];po-loc)
let rsw = rf^-1;rf

let LD-ACQ = R & (Acq|AcqRel)
and ST-REL = W & (Rel|AcqRel)

(*************)
(* ppo rules *)
(*************)

let r1 = [M];po-loc;[W]
and r2 = fence
and r3 = [LD-ACQ];po;[M]
and r4 = [R];po-loc;[LD-ACQ]
and r5 = [M];po;[ST-REL]
and r6 = [W & AcqRel];po;[R & AcqRel]
and r7 = [R];addr;[M]
and r8 = [R];data;[W]
and r9 = [R];ctrl;[W]
and r10 = ([R];po-loc-no-w;[R]) \ rsw
and r11 = [R];(addr|data);[W];po-loc-no-w;[R]
and r12 = [R];addr;[M];po;[W]
and r13 = [R];ctrl;[Fence.i];po;[R]
and r14 = [R];addr;[M];po;[Fence.i];po;[M]

let ppo =
 r1
| r2
| r3
| r4
| r5
| r6
| r7
| r8
| r9
| r10
| r11
| r12
| r13
| r14
```

Figure 3.6: `riscv-defs.cat`, part of a herd version of the RVWMO memory model (1/3)

```
Total

(* Notice that herd has defined its own rf relation *)

(* Define ppo *)
include "riscv-defs.cat"

(********************************)
(* Generate global memory order *)
(********************************)

let gmo0 = (* precursor: ie build gmo as an total order that include gmo0 *)
  loc & (W\FW) * FW | # Final write before any write to the same location
  ppo |               # ppo compatible
  rfe                 # first half of

(* Walk over all linear extensions of gmo0 *)
with  gmo from linearisations(M\IW,gmo0)

(* Add initial writes upfront -- convenient for computing rfGMO *)
let gmo = gmo | loc & IW * (M\IW)

(**********)
(* Axioms *)
(**********)

(* Compute rf according to the load value axiom, aka rfGMO *)
let WR = loc & ([W];(gmo|po);[R])
let rfGMO = WR \ (loc&([W];gmo);WR)

(* Check equality of herd rf and of rfGMO *)
empty (rf\rfGMO)|(rfGMO\rf) as RfCons

(* Atomic axion *)
let infloc = (gmo & loc)^-1
let inflocext = infloc & ext

let winside  = (infloc;rmw;inflocext) & (infloc;rf;rmw;inflocext) & [W]
empty winside as Atomic
```

Figure 3.7: `riscv.cat`, a herd version of the RVWMO memory model (2/3)

```
Partial

(***************)
(* Definitions *)
(***************)

(* Define ppo *)
include "riscv-defs.cat"

(* Compute coherence relation *)
include "cos-opt.cat"

(**********)
(* Axioms *)
(**********)

(* Sc per location *)
acyclic co|rf|fr|po-loc as Coherence

(* Main model axiom *)
acyclic co|rfe|fr|ppo as Model

(* Atomicity axiom *)
empty rmw & (fre;coe) as Atomic
```

Figure 3.8: `riscv.cat`, part of an alternative herd presentation of the RVWMO memory model (2/3)

3.3 Formal Operational Specification

An interactive operational model is described in the pages that follow, and can be explored via a web app at the following URL: `http://www.cl.cam.ac.uk/~sf502/RISCV/rmem`.

(This operational model is not yet updated to account for rules 6–7 and 11, and the model is slightly too conservative in how it implements load-acquire operations compared to the current natural language specification of RVWMO. Updates to come...).

The “Flat” RISC-V Operational Model

Christopher Pulte, Shaked Flur, Susmit Sarkar and Peter Sewell

October 2, 2017

1 Operational Model

An interactive version of the model, together with a library of litmus tests, is provided online: `http://www.cl.cam.ac.uk/~pes20/rmem`

To help reading this section the text is colour-coded as follows:

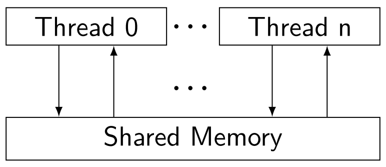
- [.aq/.rl] Acquire/Release semantics
- [atomic] Atomics (AMOs and `LR`/`SC`) semantics

In the following text, *load* means any instruction that performs a memory read (including AMOs), and *store* means any instruction that performs a memory write (including AMOs). *load-acquire* means any load that has *.aq* set (including load-acquire-RCsc), and *store-release* means any store that has *.rl* set (including store-release-RCsc).

1.1 Overview of the Model

The operational model is expressed as a state machine, with states that are an abstract representation of hardware machine states.

Model states A model state consists of a shared memory and a tuple of thread model states:



The shared memory state records the most recent write to each location. To handle `LR`/`SC` and AMOs, the memory is extended with a map (the atomics map) from read requests to sets of write slices, that associates a read request of an atomic load with the write slices it read from (excluding writes that have been forwarded to the read and have not reached memory yet).

Each thread model state (§1.4) consists principally of a tree of instruction instances, some of which have been finished, and some of which have not. Non-finished instruction instances can be subject to restart, e.g. if they depend on an out-of-order or speculative read that turns out to be unsound. The load part of AMOs can be marked as finished before the entire instruction is finished; when such instruction is restarted it is rolled back to the state it was in when the load part was marked as finished. Conditional branch instructions and `JALR` may have multiple successors in the instructions tree. When such instruction is finished, any un-taken alternative paths are discarded.

Each instruction instance state (§1.3) includes an execution state of the instruction's ISA pseudocode, which one can think of as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information, detailed below, about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc.

Model transitions For any state, the model defines the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Each transition arises from a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its thread state and the shared memory state. The transitions are introduced below and defined in §1.5, with a precondition and a construction of the post-transition model state for each. The transitions labelled ◦ can always be taken eagerly, as soon as they are enabled, without excluding other behaviour; the • cannot.

Transitions for all instructions:

- • Fetch instruction: This transition represents a fetch and decode of a new instruction instance, as a program-order successor of a previously fetched instruction instance, or at the initial fetch address for a thread.
- ◦ Register read: This is a read of a register value from the most recent program-order predecessor instruction instance that writes to that register.
- ◦ Register write
- ◦ Pseudocode internal step: This covers pseudocode internal computation, function calls, etc.
- ◦ Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted or discarded, and all memory effects have taken place. For a conditional branches and `JALR`, any non-taken po-successor branches are discarded.

Load instructions:

- ◦ Initiate memory reads of load instruction: At this point the memory footprint of the load is provisionally known and its individual reads can start being satisfied.
- • Satisfy memory read by forwarding from writes: This partially or entirely satisfies a single read by forwarding from its po-previous writes.
- • Satisfy memory read from memory: This entirely satisfies the outstanding slices of a single read, from memory.
- ◦ Complete load instruction (when all its reads are entirely satisfied): At this point all the reads of the load have been entirely satisfied and the instruction pseudocode can continue execution. A load instruction can be subject to being restarted until the Finish instruction transition or Finish load part of AMO instruction. In some cases it is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all the instructions po-before the load instruction are finished. The Restart condition over-approximates the set of instructions that might be restarted.

Store instructions:

- ◦ Initiate memory writes of store instruction, with their footprints: At this point the memory footprint of the store is provisionally known.
- ◦ Instantiate memory write values of store instruction: At this point the writes have their values and po-subsequent reads can be satisfied by forwarding from them.
- ◦ Commit store instruction: At this point the store is guaranteed to happen (it cannot be restarted or discarded), and the writes can start being propagated to memory.
- • Propagate memory write: This propagates a single write to memory.
- ◦ Complete store instruction (when its writes are all propagated): At this point all writes have been propagated to memory, and the instruction pseudocode can continue execution.

`SC` instructions:

- • Guarantee the success of `SC`: This guarantees the success of the `SC`.

- Make a sc fail: This makes the sc fail.

AMO instructions:

- Finish load part of AMO instruction: At this point the load part of an AMO instruction is done, the load part cannot be restarted or discarded, and all memory read effects have taken place. If the AMO instruction is restarted after the transition is taken, the instruction rolls back to its state right after the transition.

Fence instructions:

- Commit fence

1.2 Intra-instruction Pseudocode Execution

The intra-instruction semantics for each instruction instance is expressed as a state machine, essentially running the instruction pseudocode. Given a pseudocode execution state, it computes the next state, as one of the following:

READ_MEM(*read_kind*, *address*, *size*, *read_continuation*)	Load request
ATOMIC_RES(*res_continuation*)	sc result
WRITE_EA(*write_kind*, *address*, *size*, *next_state*)	Store effective address
WRITE_MEMV(*memory_value*, *write_continuation*)	Store value
FENCE(*fence_kind*, *next_state*)	Fence
READ_REG(*reg_name*, *read_continuation*)	Register read request
WRITE_REG(*reg_name*, *register_value*, *next_state*)	Register write
INTERNAL(*next_state*)	Pseudocode internal step
DONE	End of pseudocode

Here memory values are lists of bytes, addresses are 64-bit numbers, read and write kinds identify whether they are regular, atomic, acquire, acquire-RCsc, release and/or release-RCsc operations, register names identify a register and slice thereof (start and end bit indices), and the continuations describe how the instruction instance will continue for each value that might be provided by the surrounding memory model. Stores are split into two steps, WRITE_EA and WRITE_MEMV. We ensure these are paired in the pseudocode, but there may be other steps between them: it is observable that the WRITE_EA can occur before the value to be written is determined, because the potential memory footprint of the instruction becomes provisionally known then.

The pseudocode of each instruction performs at most one store, load, or fence, except for AMOs that preform exactly one load and one store. Those memory accesses are then split apart into the architecturally atomic units by the thread semantics (see Initiate memory reads of load instruction and Initiate memory writes of store instruction, with their footprints below).

Informally, each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit, or from the thread's initial register state if there is no such. That instance may not have executed its register write yet, in which case the register read should block. Hence, it is essential to know the register write footprint of each instruction instance, which we calculate when the instruction instance is created (see the action of Fetch instruction below). We ensure in the pseudocode that each instruction does at most one register write to each register bit, and also that it does not try to read a register value it just wrote.

Data-flow dependencies in the model emerge from the fact that register read has to wait for the appropriate register write to be executed (as described above).

1.3 Instruction Instance States

Each instruction instance i has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, rmw (i.e. AMO), or fence instruction, each with the associated kind; or a conditional branch; or a 'simple' instruction.
- *regs_in*, the set of input *reg_name*s, as statically determined;
- *regs_out*, the output *reg_name*s, as statically determined;
- *pseudocode_state* (or sometimes just 'state' for short), one of
  - Plain *next_state*, ready to make a pseudocode transition;
  - Pending_mem_reads *read_cont*, performing the memory read(s) of a load; or
  - Pending_mem_writes *write_cont*, performing the memory write(s) of a store;
  - Pending_exception *exception*, performing an exception;
- *reg_reads*, the accumulated register reads, including their sources and values, of this instance's execution so far;
- *reg_writes*, the accumulated register writes, including dependency information to identify the register reads and memory reads (by this instruction) that might have affected each;
- *mem_reads*, a set of memory read requests. Each request includes a memory footprint (an address and size) and, if the request has already been satisfied, the set of write slices (each consisting of a write and a set of its byte indices) that satisfied it.
- *mem_writes*, a set of memory write requests. Each request includes a memory footprint and, when available, the memory value to be written. In addition, each write has a flag that indicates whether the write has been propagated (passed to the memory) or not.
- [atomic] *successful_atomic*, for SC, indicates whether it was previously guaranteed to succeed or made to fail; for AMOs this will always be set to true.
- information recording whether the instance is committed, finished, etc.

Read requests include their read kind and their memory footprint (their address and size), the as-yet-unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information about the write(s) that they were satisfied from. Write requests include their write kind, their memory footprint, and their value. When we refer to a write or read request without mentioning the kind of request we mean the request can be of any kind. A load instruction which has initiated (so its read request list *mem_reads* is not empty) and for which all its read requests are satisfied (i.e. there are no unsatisfied slices) is said to be *entirely satisfied*. A LR is called *successful* if it is paired with a SC that has been guaranteed to succeed (as opposed to not paired, or the SC has not been guaranteed to succeed, or the SC has made to fail). If a successful LR has a read request that is mapped, in the atomics map, to a write slice *ws*, we say the LR has an outstanding lock on *ws*.

1.4 Thread States

The model state of a single hardware thread includes:

- *thread_id*, a unique identifier of the thread;
- *register_data*, the name, bit width, and start bit index for each register;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial fetch address for this thread;
- *instruction_tree*, a tree of the instruction instances that have been fetched (and not discarded), in program order.

1.5 Model Transitions

Fetch instruction A possible program-order successor of instruction instance i can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the thread's *instruction_tree* are from *loc*; and
2. *loc* is a possible next fetch address for i:
   (a) for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);
   (b) for an instruction that has performed a write to the program counter register (`PC`), the value that was written;
   (c) for a conditional branch, either the successor address or the branch target address; or
   (d) for `JALR`, when the target address is not yet determined, any address (this is approximated in our tool implementation, necessarily);

Action: construct a freshly initialized instruction instance i' for the instruction in the program memory at *loc*, with state PLAIN *next_state*, including the static information available from the ISA model such as its *instruction_kind*, *regs_in*, and *regs_out*, and add i' to the thread's *instruction_tree* as a successor of i. If the instruction fails to decode, set the state of i' to PENDING_EXCEPTION *exception* with *exception* that describes the decoding error.

This involves only the thread, not the storage subsystem, as we assume a fixed program rather than modelling fetches with memory reads; we do not model self-modifying code.

Initiate memory reads of load instruction An instruction instance i with next pseudocode state READ_MEM*(read_kind, address, size, read_cont)* can initiate the corresponding memory reads if:

1. all po-previous `fence` instructions with *.sr* set are finished;
2. all po-previous `fence.i` instructions are finished;
3. [.aq/.rl] if i is a load-acquire-RCsc, all po-previous store-releases-RCsc are finished; and
4. [.aq/.rl] all non-finished po-previous load-acquire instructions are entirely satisfied.

Action:

1. Construct the appropriate read requests *rrs*:
   - if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;
   - otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *address*...*address+size-1*.
2. set *i.mem_reads* to *rrs*; and
3. update the state of i to PENDING_MEM_READS *read_cont*.

Satisfy memory read by forwarding from writes For a load instruction instance i in state PENDING_MEM_READS *read_cont*, and a read request, r in *i.mem_reads* that has unsatisfied slices, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before i.

Let *wss* be the maximal set of unpropagated write slices from store instruction instances po-before i (if i is a load-acquire, exclude `SC` and AMO writes), that overlap with the unsatisfied slices of r, and which are not superseded by intervening stores that are either propagated or read from by this thread. That last condition requires, for each write slice *ws* in *wss* from instruction i':

- that there is no store instruction po-between i and i' with a write overlapping *ws*, and

- that there is no load instruction po-between i and i' that was satisfied from an overlapping write slice from a different thread.

Action:

1. update r to indicate that it was satisfied by *wss*; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a po-successor of i, and every read request r' of i' that was satisfied from *wss*$'$, if there exists a write slice *ws*$'$ in *wss*$'$, and an overlapping write slice from a different write in *wss*, and *ws*$'$ is not from an instruction that is a po-successor of i, restart i' and its data-flow dependents (including po-successors of restarted load-acquire instructions).

A consequence of the above is that store-release-RCsc writes cannot be forwarded to load-acquires-RCsc: a load-acquire-RCsc instruction cannot be in state PENDING_MEM_READS *read_cont* before all the po-previous store-release-RCsc instructions are finished, and *wss* does not include writes from finished stores (as those must be propagated writes).

Satisfy memory read from memory For a load instruction instance i in state PENDING_MEM_READS *read_cont*, and a read request r in *i.mem_reads*, that has unsatisfied slices, the read request can be satisfied from memory under the condition that if i is an AMO or a successful LR then no other AMO or successful LR from a different thread has an outstanding lock on the writes r is trying to read from. Action: let *wss* be the write slices from memory covering the unsatisfied slices of r, and apply the action of Satisfy memory read by forwarding from writes. In addition, if i is an AMO or a successful LR, union *wss* with the set of write slices r is mapped to in the atomics map.

Note that Satisfy memory read by forwarding from writes might leave some slices of the read request unsatisfied. Satisfy memory read from memory, on the other hand, will always satisfy all the unsatisfied slices of the read request.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state PENDING_MEM_READS *read_cont* can be completed (not to be confused with finished) if all the read requests *i.mem_reads* are entirely satisfied (i.e., there are no unsatisfied slices). Action: update the state of i to PLAIN *(read_cont (memory_value))*, where *memory_value* is assembled from all the write slices that satisfied *i.mem_reads*.

Finish load part of AMO instruction An AMO instruction instance i that has completed the load part, i.e., Complete load instruction (when all its reads are entirely satisfied) has been taken, can be marked as such if the conditions of Finish instruction are satisfied, except for the fully determined data condition for the src register (rs2). Action: mark the load part of i as finished. If later on i has to be restarted, reset its state to the current state.

Guarantee the success of SC A SC instruction instance i with next pseudocode state ATOMIC_RES*(res_cont)* can be guaranteed to succeed if:

1. i has not been made to fail (as recorded in *i.successful_atomic*);
2. i is paired with a LR i'; and
3. if i' has already been satisfied (not necessarily entirely), let *wss* be the set of propagated write slices i' has read from, then, no slice in *wss* has been overwritten (in memory) by a write from a different thread, and no other AMO or successful LR from a different thread has an outstanding lock on a write slice from *wss*.

Action:

1. record in *i.successful_atomic* that i will be successful;

2. if i' has already been satisfied, union *wss* with the set of write slices the read request of i' is mapped to in the atomics map, where *wss* is as above; and
3. update the state of i to PLAIN *(res_cont (true))*.

Make a SC fail A SC instruction instance i with next pseudocode state ATOMIC_RES*(res_cont)* can be made to fail if the SC has not been guaranteed to succeed (as recorded in *i.successful_atomic*) Action:

1. record in *i.successful_atomic* that the SC was made to fail; and
2. update the state of i to PLAIN *(res_cont (false))*.

Note the promise-success transition is enabled before the SC commits, and we do not require it to have a fully-determined address or to be non-restartable. As a result, a SC that has already promised its success might be restarted. Since other instructions may rely on its promise, the restart will not affect the value of *i.successful_atomic*. Instead, when the SC is restarted it will take the same promise/failure transition as before its restart — based on the value of *i.successful_atomic*.

Initiate memory writes of store instruction, with their footprints An instruction instance i with next pseudocode state WRITE_EA*(write_kind, address, size, next_state')* can announce its pending write footprint. Action:

1. construct the appropriate write requests:
   - if *address* is aligned to *size* then *ws* is a single write request of *size* bytes to *address*;
   - otherwise *ws* is a set of *size* write requests, each of one byte size, to the addresses *address*...*address+size-1*.
2. set *i.mem_writes* to *ws*; and
3. update the state of i to PLAIN *next_state'*.

Note that at this point the write requests do not yet have their values, but other, non-overlapping po-following writes, can already propagate.

Instantiate memory write values of store instruction An instruction instance i with next pseudocode state WRITE_MEMV*(memory_value, write_cont)* can initiate the corresponding memory writes. Action:

1. split *memory_value* between the write requests *i.mem_writes*; and
2. update the state of i to PENDING_MEM_WRITES *write_cont*.

Commit store instruction For an uncommitted store instruction i in state PENDING_MEM_WRITES *write_cont*, i can commit if:

1. i has fully determined data (i.e., the register reads cannot change, see §1.6);
2. all po-previous conditional branch instructions are finished;
3. all po-previous `fence` instructions with *.sw* set are finished;
4. all po-previous `fence.i` instructions are finished;
5. [.aq/.rl] all po-previous load-acquire instructions are finished;
6. all po-previous store instructions, except for SC that failed, have initiated and so have non-empty *mem_writes*;
7. [.aq/.rl] if i is a store-release, all po-previous memory access instructions are finished;
8. all po-previous memory access instructions have a fully determined memory footprint; and
9. all po-previous load instructions have initiated and so have non-empty *mem_reads*.

Action: record i as committed.

Propagate memory write For an instruction i in state PENDING_MEM_WRITES *write_cont*, and an unpropagated write, w in *i.mem_writes*, the write can be propagated if:

1. all memory writes of po-previous store instructions that overlap w have already propagated
2. all read requests of po-previous load instructions that overlap with w have already been satisfied, and (the load instruction) is non-restartable (see §1.6);
3. all read requests satisfied by forwarding w are entirely satisfied; and
4. [atomic] no AMO or successful LR from a different thread has an outstanding lock on a write slice that overlaps with w.

Action:

1. update the memory with w;
2. record w as propagated;
3. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' that was satisfied from wss', if there exists a write slice ws' in wss' that overlaps with w and is not from w, and ws' is not from a po-successor of i, restart i' and its data-flow dependents; and
4. [atomic] for every AMO and successful LR that has read from w (by forwarding), add the slices of w this load read from to the set of write slices the read request of the load is mapped to in the exclusives map.

Complete store instruction (when its writes are all propagated) A store instruction i in state PENDING_MEM_WRITES *write_cont*, for which all the memory writes in *i.mem_writes* have been propagated, can be completed. Action: update the state of i to PLAIN*(write_cont(true))*.

Commit fence A fence instruction i in state PLAIN *next_state* where *next_state* is FENCE*(fence_kind, next_state')* can be committed if:

1. all po-previous conditional branch instructions are finished; **[TODO: this looks stronger than intended, but actually it has no observable effect for most fences; the exception is "fence w,r", e.g., MP+fence.w.w+ctrlfence.w.r is allowed by Daniel's model but forbidden as a consequence of this condition. Fixing this means changing the invariant that finished instructions are never discarded, to finished load/store instructions are never discarded. Is RISC-V going to include "fence w,r"?]**
2. if i is a `fence` instructions with *.pr* set, all po-previous load instructions are finished;
3. if i is a `fence` instructions with *.pw* set, all po-previous store instructions are finished; and
4. if i is a `fence.i` instruction, all po-previous memory access instructions have fully determined memory footprints.

Action: update the state of i to PLAIN *next_state'*.

Register read An instruction instance i with next pseudocode state READ_REG*(reg_name, read_cont)* can do a register read if every instruction instance that it needs to read from has already performed the expected register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *register_value* be the assembled value from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *register_value*; and

2. update the state of i to PLAIN *(read_cont(register_value))*.

Register write An instruction instance i with next pseudocode state WRITE_REG*(reg_name, register_value, next_state′)* can do the register write. Action:

1. add *reg_name* to *i.reg_writes* with *write_deps* and *register_value*; and
2. update the state of i to PLAIN *next_state′*.

where *write_deps* is the set of all *read_sources* from *i.reg_reads* and a flag that is set to true if i is a load instruction that has already been entirely satisfied.

Pseudocode internal step An instruction instance i with next pseudocode state INTERNAL*(next_state′)* can do that pseudocode-internal step. Action: Update the state of i to PLAIN *next_state′*.

Finish instruction A non-finished instruction i with next pseudocode state DONE can be finished if:

1. if i is a load instruction:
   (a) [.aq/.rl] all po-previous load-acquire instructions are finished; and
   (b) it is guaranteed that the values read by the read requests of i will not cause coherence violations, i.e., for any po-previous instruction instance i', let *cfp* be the combined footprint of propagated writes from store instructions po-between i and i' and fixed writes that were forwarded to i from store instructions po-between i and i' including i', and let *cfp′* be the complement of *cfp* in the memory footprint of i. If *cfp′* is not empty:
      i. i' has a fully determined memory footprint;
      ii. i' has no unpropagated memory write that overlaps with *cfp′*; and
      iii. If i' is a load with a memory footprint that overlaps with *cfp′*, then all the read requests of i' that overlap with *cfp′* are satisfied and i' can not be restarted (see §1.6).

   Here a memory write is called fixed if it is the write of a store instruction that has fully determined data.
2. i has fully determined data; and
3. all po-previous conditional branches are finished.

Action:

1. if i is a branch instruction, discard any untaken path of execution, i.e., remove any (non-finished) instructions that are not reachable by the branch taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

1.6 Auxiliary Definitions

Fully determined Informally, an instruction is said to have *fully determined data* if the load instructions feeding its input registers are finished. Similarly, it is said to have *fully determined memory footprint* if the load instructions feeding its memory location register are finished. Formally, we first define the notion of *fully determined register write*: a register write w, of instruction i, with the associated *write_deps* from *i.reg_writes* is said to be *fully determined* if one of the following conditions hold:

1. i is finished (just the load part for AMOs); or
2. the load flag in *write_deps* is *false* and every register write in *write_deps* is fully determined.

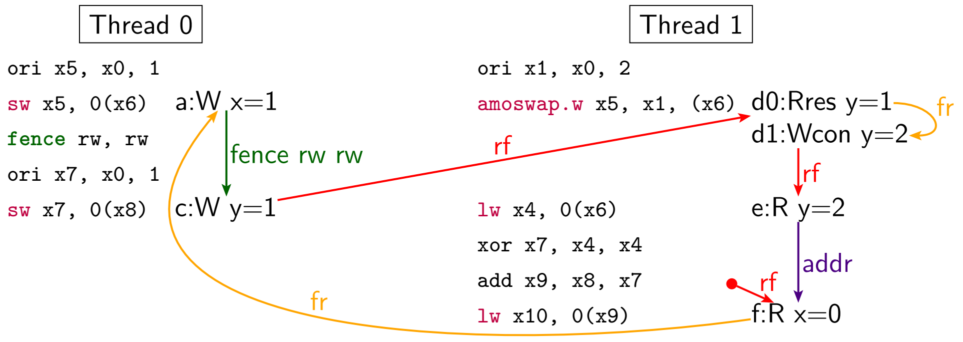
Now, an instruction i is said to have a *fully determined data* if all the register writes of *read_sources* in *i.reg_reads* are fully determined; and, i is said to have a *fully determined memory footprint* if all the register writes of *read_sources* in *i.reg_reads* that are associated with registers that feed into i's memory access footprint are fully determined.

Restart condition To determine if instruction i might be restarted we use the following condition: i is a non-finished instruction and at least one of the following holds,

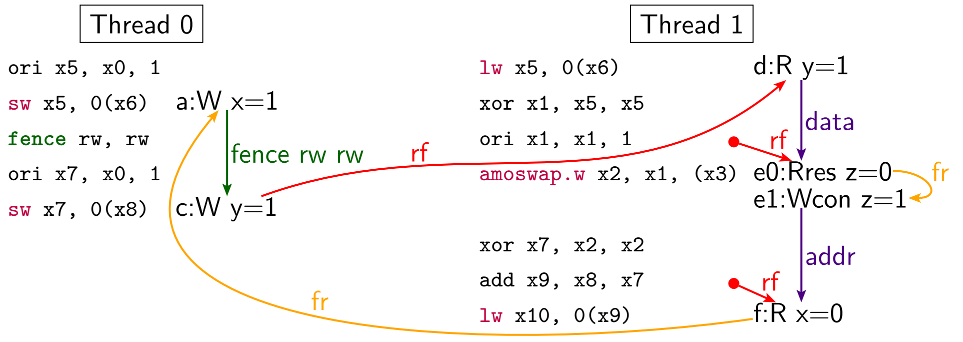
1. there exists a store instruction s and an unpropagated write w of s such that applying the action of the Propagate memory write transition to w will result in the restart of i;
2. there exists a non-finished load instruction l and a read request rr of l such that applying the action of the Satisfy memory read from memory transition to rr will result in the restart of i (even if rr is already satisfied); or
3. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents, or i' is a load-acquire.

1.7 AMOSWAP

The pseudocode execution of each instruction is serial (though not atomic). For most instructions we were able to find a maximally liberal order of the pseudocode that allows all intended behaviours. `amoswap` is an exception, as the order in which the load part and the store part are executed is observable. If we serialise the load part first, the litmus tests MP+fence.rw.rw+amoswap-rfi-addr:



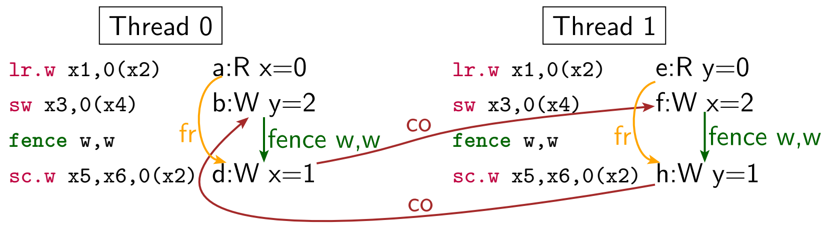
is forbidden (allowed by the axiomatic model). If we serialise the store part first, the litmus test MP+fence.rw.rw+data-amoswap-addr:



is forbidden (allowed by the axiomatic model). Currently, in the *rmem* tool, the load is serialised before the store.

1.8 Remarks about LR/SC and AMO instructions

The RISC-V architecture intends that the dest register (rd) of `SC` does not introduce dependencies, to allow (e.g.) hardware optimisations that dynamically replace `LR`/`SC` pairs by atomic read-modify-write operations that can execute in the memory subsystem and therefore be guaranteed to succeed. In the axiomatic definition this is expressed by all address/data/control dependencies stemming from load instructions (hence, not SC). In the operational model, matching this weakness has proved to be difficult: it means the operational model must be able to promise the success or failure of a SC even before any of its registers reads/writes have been performed, and in particular, before the SC's address and data are available. This can lead to deadlocks in the operational model. To illustrate this consider, for example, the following litmus test and a state where both a and e are satisfied and finished, and where b and f are not propagated.



In this state, d can promise its success, locking memory location x, and immediately after, h can promise its success, locking location y. The resulting state has a deadlock:

- For d to propagate, the preceding `fence w,w` has to be committed and hence b propagated. But b cannot propagate since y is locked.
- For h to propagate, the preceding `fence w,w` has to be committed and hence f propagated. But f cannot propagate since x is locked.

Similar situations arise from cases where there are other barriers or release/acquire instructions in-between the LR/SC, or if the SC has additional dependencies that the LR does not have (this can also lead to a deadlock when using AMOs instead of LR/SC). These are cases that are not really intended to be supported by the