RISC-V Memory Consistency Model Tutorial

Dan Lustig
May 7, 2018
ABOUT ME

• Senior Research Scientist at NVIDIA in Santa Clara, CA
• PhD from Princeton in 2015
• Chair of RISC-V Memory Consistency Model task group
• Co-responsible for NVIDIA GPU memory consistency model
OUTLINE

• Setting the Stage
• Litmus Tests
• RISC-V Weak Memory Ordering ("RVWMO")
• Extensions: "Zam" and "Ztso"
• Documentation and Tools
• Conclude
WHAT IS A MEMORY CONSISTENCY MODEL?

Specifies the values that can be returned by loads
WHY DO WE NEED A MEMORY MODEL?
WHY DO WE NEED A MEMORY MODEL?

...to give everyone a headache?
WHY DO WE NEED A MEMORY MODEL?

For the same reason we need any other technical specification:

It is (one specific part of) the contract between the software and the implementation about the set of legal behaviors.
WHY DO WE NEED A MEMORY MODEL?

For the same reason we need any other technical specification:

It is (one specific part of) the contract between the software and the implementation about the set of legal behaviors

The other parts of the contract are defined by the rest of the ISA specification (including the ISA Formal Specification; see that TG’s tutorial later today)
A WIDE RANGE OF MEMORY MODELS

Note: diagram obviously not to scale, just a rough picture 😊
A WIDE RANGE OF MEMORY MODELS

Note: diagram obviously not to scale, just a rough picture 😊

Low Performance

Sequential Consistency

Total Store Ordering (TSO)

RISC-V (RVWMO)

IBM Power

NVIDIA GPUs

There is a big cliff here called “multi-copy-atomicity”

Hard for Implementers

Hard for Programmers
SOME CASES ARE EASY (RELATIVELY)...

Initial condition on both harts: $s_0 = \text{address } x$; $s_1 = \text{address } y$.
Initial conditions in memory: all locations initialized to 0.

<table>
<thead>
<tr>
<th>Hart 0</th>
<th>Hart 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>li t1, 1</td>
<td>loop:</td>
</tr>
<tr>
<td>sw t1, 0(s0)</td>
<td>lw a0, 0(s1)</td>
</tr>
<tr>
<td>fence w,w</td>
<td>beqz a0, loop</td>
</tr>
<tr>
<td>sw t1, 0(s1)</td>
<td>fence r,r</td>
</tr>
<tr>
<td></td>
<td>lw a1, 0(s0)</td>
</tr>
</tbody>
</table>

Final output: what are the possible final values of $a_0$ and $a_1$ on hart 1?
**SOME CASES ARE EASY (RELATIVELY)…**

Initial condition on both harts: $s_0 == \text{address } x; s_1 == \text{address } y$.
Initial conditions in memory: all locations initialized to 0

<table>
<thead>
<tr>
<th>Hart 0</th>
<th>Hart 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>li t1, 1</td>
<td>loop:</td>
</tr>
<tr>
<td>sw t1, 0(s0)</td>
<td>lw a0, 0(s1)</td>
</tr>
<tr>
<td>fence w,w</td>
<td>beqz a0, loop</td>
</tr>
<tr>
<td>sw t1, 0(s1)</td>
<td>fence r,r</td>
</tr>
<tr>
<td></td>
<td>lw a1, 0(s0)</td>
</tr>
</tbody>
</table>

Final output: what are the possible final values of $a_0$ and $a_1$ on hart 1?

Only possible outcome is $a_0 == a_1 == 1$
SOME CASES ARE HARD...

• Should this outcome be permitted or forbidden? We’re not even sure ourselves...

<table>
<thead>
<tr>
<th>Hart 0</th>
<th>Hart 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>li t1, 1</td>
<td>li t1, 1</td>
</tr>
<tr>
<td>(a) lw a0,0(s0)</td>
<td>(d) sw t1,4(s1)</td>
</tr>
<tr>
<td>(b) fence rw,rw</td>
<td>(e) ld a1,0(s1)</td>
</tr>
<tr>
<td>(c) sw t1,0(s1)</td>
<td>(f) lw a2,4(s1)</td>
</tr>
<tr>
<td></td>
<td>xor a3,a2,a2</td>
</tr>
<tr>
<td></td>
<td>add s0,s0,a3</td>
</tr>
<tr>
<td></td>
<td>(g) sw a2,0(s0)</td>
</tr>
</tbody>
</table>

Outcome: a0=1, a1=0x100000001, a1=1

Figure A.22: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)
ARCHITECTURE VS. MICROARCHITECTURE

An implementation can do anything it wants under the covers, as long as the load return values satisfy RVWMO i.e., implementations can speculate past a lot of these rules, as long as they make sure to, e.g., squash and replay whenever the violation might actually become observable
OPERATIONAL VS. AXIOMATIC

In modern practice, at ISA level, two common modeling approaches:

Axiomatic: define a set of criteria (“axioms”) to be satisfied
• Executions permitted unless they fail one or more axioms

Operational: define a golden abstract machine model
• Executions forbidden unless producible when executing this model

Ideally: figure out how to meet in the middle (can be difficult!)
• lots of gray area, obscure code, etc.
SEQUENTIAL CONSISTENCY [LAMPORT ‘79]

**Axiomatic**

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order

3. Loads return the value written by the latest store to the same address in the total order

**Operational**

1. Harts take turn executing instructions. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Loads return the value written by the most recent preceding store to the same address
SEQUENTIAL CONSISTENCY [LAMPORT ‘79]

**Axiomatic**

1. There is a total order on all memory operations. The order is non-deterministic.
2. That total order respects program order
3. Loads return the value written by the latest store to the same address in the total order

**Operational**

1. Harts take turn executing instructions. The order is non-deterministic.
2. Each hart executes its own instructions in order
3. Loads return the value written by the most recent preceding store to the same address

**Preserved Program Order (PPO)**

**Load Value Axiom**

**Global memory order**
GLOBAL MEMORY ORDER

A total order over all memory operations in a program

A memory operation “performs” (enters the global memory order) when:

• a load determines its return value
• a store becomes globally visible
# SEQUENTIAL CONSISTENCY

**Axiomatic**

1. There is a total order on all memory operations. The order is non-deterministic.
2. That total order respects program order
3. Loads return the value written by the latest store to the same address in the total order

**Operational**

1. Harts take turn executing instructions. The order is non-deterministic.
2. Each hart executes its own instructions in order
3. Loads return the value written by the most recent preceding store to the same address
TOTAL STORE ORDERING (SPARC, X86, RVTSO)

**Axiomatic**

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order, except Store→Load ordering.

3. Loads return the value written by the latest store to the same address in program or memory order (whichever is later).

**Operational**

1. Harts take turn executing steps. The order is non-deterministic.

2. Each hart executes its own instructions in order.

3. Stores execute in two steps: 1) enter store buffer, 2) drain to memory.

4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address.
ADDING A STORE BUFFER

- If a load bypasses a store in the (FIFO) store buffer, then the load appears before the store in global memory order.
  - The load determines its return value before the store becomes globally visible.
  - The performance win is too important...the model needs to be changed to account for this!

[Figure 1: x86-TSO block diagram]

[Sewell et al., CACM ‘10]
TOTAL STORE ORDERING (SPARC, X86, RVTSO)

Axiomatic

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order, except Store→Load ordering

3. Loads return the value written by the latest store to the same address in program or memory order (whichever is later)

Operational

1. Harts take turn executing steps. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Stores execute in two steps: 1) enter store buffer, 2) drain to memory

4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address
TOTAL STORE ORDERING (SPARC, X86, RVTSO)

Axiomatic

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order, except Store→Load ordering.

3. Loads return the value written by the latest store to the same address in program or memory order (whichever is later).

Operational

1. Global memory order:

2. Preserved Program Order (PPO):

3. Stores execute in two steps: 1) enter store buffer, 2) drain to memory.

4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address.
RISC-V WEAK MEMORY ORDERING (RVWMO)

Axiomatic

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects thirteen specific patterns (next slide)

3. Loads return the value written by the latest store to the same address in program or memory order (whichever is later)

Operational

1. Harts take turn executing steps. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Multiple steps for each instruction (see spec Appendix B)

4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address
RISC-V WEAK MEMORY ORDERING (RVWMO)

**Axiomatic**

1. There is a total order on all memory operations. The order is non-deterministic.
2. That total order respects **thirteen specific patterns** (next slide)
3. Loads return the value written by the latest store to the same address in **program or memory order** (whichever is later)

**Operational**

1. Harts take turn executing steps. The order is non-deterministic.
2. Each hart executes its own instructions in order
3. Multiple steps for each instruction (see spec Appendix B)
4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address

**Global memory order**

**Preserved Program Order (PPO)**

**Load Value Axiom**
RVWMO PPO RULES IN A NUTSHELL

- **Preserved Program Order**: if \( A \) appears before \( B \) in program order, and \( A \) and \( B \) match one of the patterns below, then \( A \) appears before \( B \) in global memory order.

\[
\text{A} \quad \text{Overlap} \quad \text{Store} \quad \text{except “rsw” and “fri;rfi”}
\]

\[
\text{Load} \quad \text{Overlap} \quad \text{AMO/SC}
\]

\[
\text{Fence} \quad \text{with pr/pw/sr/sw set appropriately}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load} \quad \text{except “rsw” and “fri;rfi”}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{AMO/SC}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{SC}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{SC}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]

\[
\text{A} \quad \text{Fence} \quad \text{Overlap} \quad \text{LR}
\]

\[
\text{B} \quad \text{Overlap} \quad \text{Load}
\]
PPO RULE 1

If A and B access the same address (or have any overlapping footprint), then A must appear before B in global memory order:

• A load A must determine its value before B becomes globally visible
• A store A must become globally visible before B becomes globally visible
A load B cannot determine its return value by forwarding from an Atomic Memory Operation or Store-Conditional operation that has not yet become globally visible.
PPO RULE 3

A load B cannot determine its return value by forwarding from an Atomic Memory Operation or Store-Conditional operation that has not yet become globally visible.

(Recall: this defines the architectural rules. Implementations can do whatever they want, as long as all outcomes are legal.)
PPO RULE 4

fence \([r][w][i][o], [r][w][i][o]\)

Orders operations in the *predecessor set* before operations in the *successor set*

PR: previous reads. SR: subsequent reads

PW: previous writes. SW: subsequent writes

PI: previous I/O reads. SI: subsequent I/O reads

PO: previous I/O writes. SO: subsequent I/O writes

with pr/pw/sr/sw set appropriately
PPO RULES 5-7

AMOs and LR/SC have optional *acquire* and *release* annotations for release consistency

- All operations following an *acquire* in program order also following it in global memory order
- All operations preceding a *release* in program order also precede it in global memory order
- A *release* that precedes an *acquire* in program order also precedes it in global memory order
  - i.e., the RCsc variant of release consistency
PPO RULES 9-11

If B has a syntactic *address, control, or data* dependency on A, then A precedes B in global memory order

- Except control dependencies where B is a store
- Address dependency: the result of A is used to determine the address accessed by B
- Control dependency: the result of A feeds a branch that determines whether B is executed at all
- Data dependency: the result of A is used to determine the value written by store B

Note: ordering maintained regardless of actual values!
PPO RULES 12-13

1. B follows M in program order, and M has an address dependency on A

2. B returns a value from an earlier store M in the same hart, and M has an address or data dependency on A

Most processors will maintain these naturally, yet most programmers won’t ever use them anyway

We made them explicit rules so that the operational and axiomatic models all agree

• And also for Linux, which has similar rules too
PPO RULE 8

A load-reserve operation determines its value before the paired store-conditional becomes globally visible

(Mostly redundant with rules 1 and 11, except in rare cases of mismatched addresses and no data dependency)
PPO RULE 2

Same-address load-load ordering is also maintained, with two exceptions:

1. Both return values come from the same store
   - A form of architecturally-visible speculation
   - Common in many implementations

2. B forwards from a store M between A and B in program order
   - B can determine its value from the store buffer while A is still fetching an older value from memory
ATOMICITY OF AMO AND LR/SC

AMOs grab an old value in memory, perform an arithmetic operation (except for swap), and write the new value to memory, all in one single atomic operation

• One node in the global memory order

LR grabs a reservation. SC performs a store if the reservation is still valid, and then releases the reservation.

• A reservation can be killed for any reason. A reservation must be killed if there is a store to the reserved address range from any other hart.

• Certain constrained LR/SC sequences guaranteed to eventually succeed (see spec)
PROGRESS AXIOM

No operation can be preceded in the global memory order by an infinite sequence of operations from other harts

• Very intentionally the weakest forward progress guarantee that is needed to make the memory model work

• Does not imply any stronger notion of fairness!
...AND THAT’S IT!
MEMORY MODEL ISA EXTENSIONS

• “Zam” extends “A” by permitting misaligned AMOs
  • “A” without “Zam” now forbids misaligned AMOs or LR/SC pairs

• “Ztso” strengthens the baseline memory model to TSO
  • TSO-only code is not backwards-compatible with RVWMO
ONGOING/FUTURE WORK

• Mixed-size, partially-overlapping memory accesses
• Formalize instruction fetches and FENCE.I TLB flushes and SFENCE.VMA, etc.
• Integration with other extensions (V, J, N, T, ...)
• Integration with the ISA formalization task group’s effort
• Cache flush/writeback/etc. operations
• (The task group logistics for all this are still TBD)
DOCUMENTATION & TOOLS

- Appendix A: two dozen pages explaining the details in plain English

- Appendix B: Two axiomatic models and one operational model, with associated tools (Alloy, herd, rmem)

- More than 7000 litmus tests online
  - (also to be used to test compliance)

Figure A.16: Because of the address dependency from (d) to (e), (d) also precedes (f) (outcome forbidden)
MEMORY MODEL RATIFICATION TIMELINE

• Released for public review on 5/2/18
• Foundation requires at least 45 days for public review. This will end no earlier than 6/16/18.
• If you have comments or feedback:
  • send to isa-dev
  • send as a PR or issue on riscv-isa-manual GitHub repo
  • send to me directly
TOTAL STORE ORDERING (SPARC, X86, RVTSO)

Axiomatic

\[
ppo := \text{(program order)} - W \rightarrow R
\]

\[
\text{acyclic}(ppo \cup \text{rfe} \cup \text{co} \cup \text{fr} \cup \text{fence})
\]

\[
\text{acyclic}(\text{po}_\text{loc} \cup \text{rf} \cup \text{co} \cup \text{fr})
\]

Operational

\[\text{Operational}\]

\[\text{Operational}\]

\[\text{Operational}\]

Fig. 8. The message passing pattern \(mp\) with lightweight fence and \(ppo\) (forbidden)

[Alglave et al., TOPLAS ‘09]

[Sewell et al., CACM ‘10]
RVWMO

Axiomatic (App. B.2)

ppo := (13 rules, on next slide)

acyclic(ppo U rfe U co U fr)

acyclic(po_loc U rf U co U fr)

Operational (App. B.3)

B.3 An Operational Memory Model

This is an alternative presentation of the RVWMO memory model in operational style. It aims to admit exactly the same extensional behaviour as the axiomatic presentation: for any given program, admitting an execution if and only if the axiomatic presentation allows it.

The axiomatic presentation is defined as a predicate on complete candidate executions. In contrast, this operational presentation has an abstract microarchitectural flavour: it is expressed as a state machine, with states that are an abstract representation of hardware machine states, and with explicit out-of-order and speculative execution (but abstracting from non-implementation-specific microarchitectural details such as register renaming, store buffers, cache hierarchies, cache protocols, etc.). As such, it can provide useful intuition. It can also construct executions incrementally, making it possible to interactively and randomly explore the behaviour of larger examples, while the axiomatic model requires complete candidate executions over which the axioms can be checked.

The operational presentation covers mixed-size execution, with potentially overlapping memory accesses of different power-of-two byte sizes. Misaligned accesses are broken up into single-byte accesses.

An interactive version of the model, together with a library of litmus tests, is provided online: http://www.cs.ox.ac.uk/~pes021/lmm. This is integrated with a fragment of the RISC-V ISA semantics (RV64 and RV32) expressed explicitly in SML (https://github.com/eme-project/rvml).

Below is an informal introduction of the model states and transitions. The description of the formal model starts in the next subsection.

Terminology: In contrast to the axiomatic presentation, here every memory operation is either a load or a store. Hence, A3MOs give rise to two distinct memory operations: a load and a store. When used in conjunction with “instruction”, the terms “load” and “store” refer to instructions that give rise to such memory operations. As such, both include A3MO instructions. The term “acquire” refers to memory operations which are protected by a lock.

Model states

Hart 0 ... Hart n

Shared Memory

Figure A.13: A variant of the LB litmus test (outcome forbidden)
MULTI-COPY ATOMICITY

A load may only return a value from:

• An earlier store from the same hart ("hardware thread")
• A store that is globally visible

In other words, a store may not “peek” into a neighbor hart’s private store buffer
WHO FEELS THE PAIN?

- **C/C++ MM**: Canonical Mapping
- **Java MM**: Canonical Mapping
- **Linux MM**: Canonical Mapping
- **Synchronization Libraries**: Hand Mapping

---

RISC-V ISA Memory Consistency Model

- **Misconception**: end users will have to deal with the memory model
- **Reality**: end users rarely interact with the ISA memory model directly
- **Burden falls instead on library/compiler writers and microarchitects**
MEMORY MODEL TASK GROUP PROGRESS

• May 2017 Workshop: Formed the task group
  (...debate...)

• November 2017 Workshop: Settled on the basics
  • RVWMO baseline, and optional RVTSO extension
    (...refinement...)

• May 2018 Workshop: released for ratification!
  • Public review period runs May 2 through June 16
RISC-V MEMORY MODEL SPECIFICATION

• Chapter 6: RISC-V Weak Memory Ordering ("RVWMO")

• Chapter 20: “Zam” Std. Extension for Misaligned AMOs

• Chapter 21: “Ztso” Std. Extension for Total Store Ordering

• Appendix A: Explanatory Material and Litmus Tests

• Appendix B: Formal Memory Model Specifications
RVWMO RULES IN A NUTSHELL

• **Load Value Axiom:** each byte of each load $i$ returns the value written to that byte by the store that is the latest in global memory order among the following stores:
  1. Stores that write that byte and that precede $i$ in the global memory order
  2. Stores that write that byte and that precede $i$ in program order

• **Atomicity Axiom:** no store from another hart can appear in the global memory order between a paired LR and successful SC
  - *(this axiom simplified here for clarity...see spec for complete definition)*

• **Progress Axiom:** no memory operation may be preceded in the global memory order by an infinite sequence of other memory operations