# RISC-V Memory Consistency Model Status Update

Dan Lustig and the Memory Model TG

Nov. 28, 2017

## WHAT WERE OUR GOALS?

- Define the RISC-V <u>memory consistency model</u>
  - *Specifies the values that can be returned by loads*

- Support a wide range of HW implementations

- Support Linux, C/C++, and lots of other critical SW

# ~~DECADES~~ MONTHS OF DEBATE

**Strong Models**
(e.g., x86-TSO)

- Stricter ordering rules
- Simpler for programmers and for architects

**Weak Models**
(e.g., ARM, IBM Power)

- More relaxed ordering rules
- Better perf/power/area
- More microarchitectural freedom

Linux/C/C++/Java/… work either way!  That's not a deciding factor

# FINDING A COMPROMISE

- First step: narrow choice to two specific models
  - RVTSO: as used by SPARC, x86 *(strong)*
  - RVWMO: similar to ARMv8 *(weak)*
    - "<u>R</u>ISC-<u>V</u> <u>W</u>eak <u>M</u>emory <u>O</u>rdering"

- Both are "multi-copy atomic" = both are simpler than the IBM Power and ARMv7 memory models

## THE RISC-V MEMORY MODEL PLAN

- **Base RISC-V memory model: RVWMO**
  - HW can still be as conservative as it wants to be
  - Portable SW must nevertheless assume RVWMO

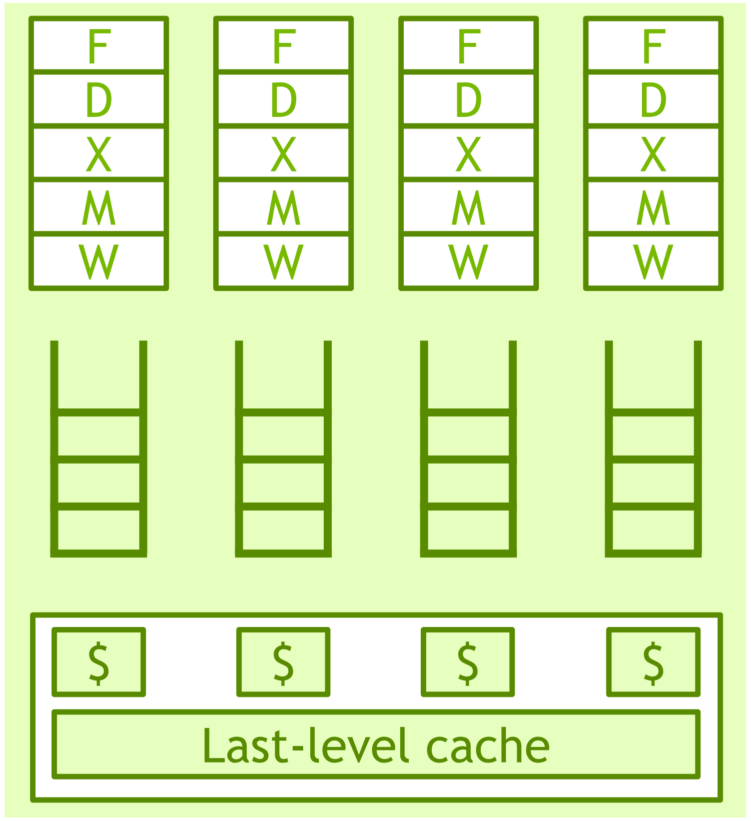
- **New ISA extension “Ztso”**: HW which implements RVTSO can (optionally) choose to expose this to SW
  - E.g., a RISC-V core can implement “IMAFD+Ztso”

# ARCHITECTURAL INTUITION (FOR BOTH)

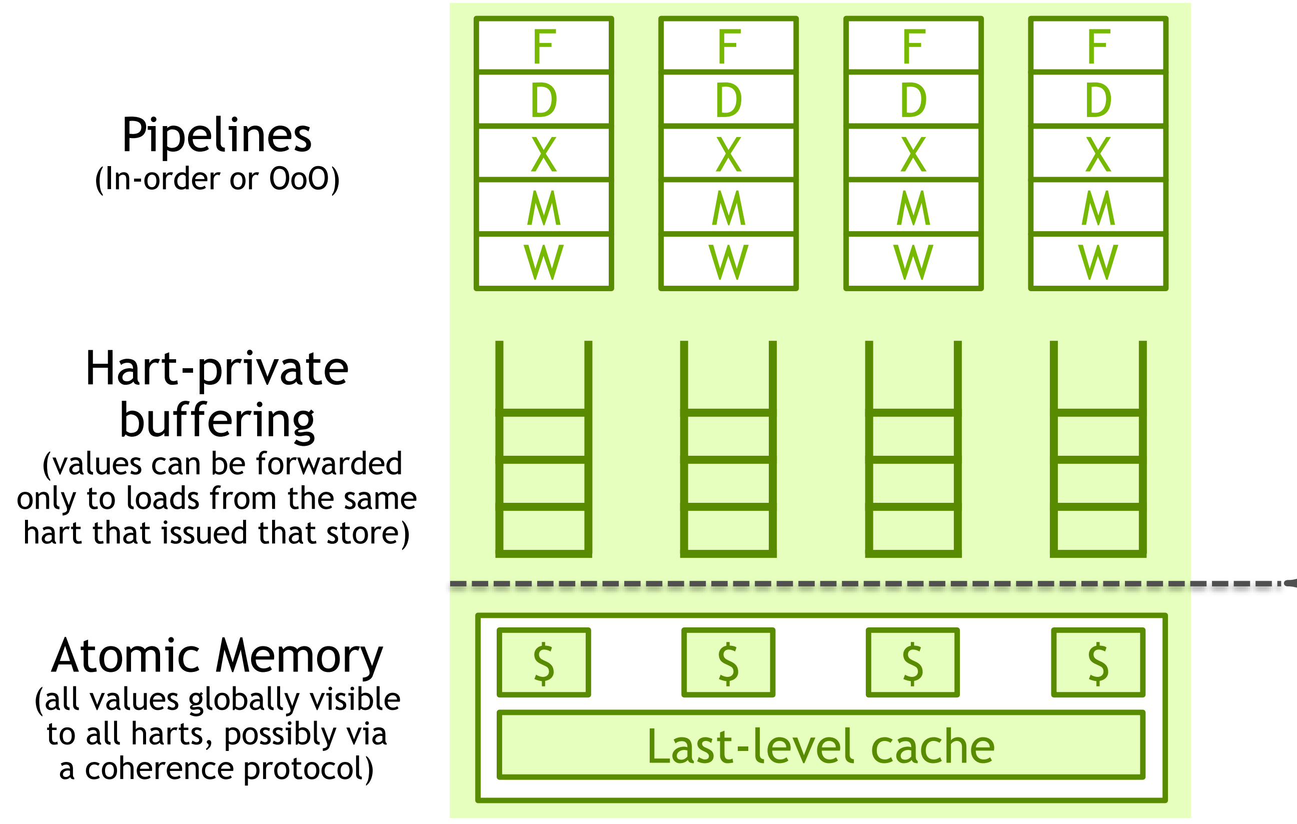
Pipelines
(In-order or OoO)

Hart-private buffering
(values can be forwarded only to loads from the same hart that issued that store)

Atomic Memory
(all values globally visible to all harts, possibly via a coherence protocol)



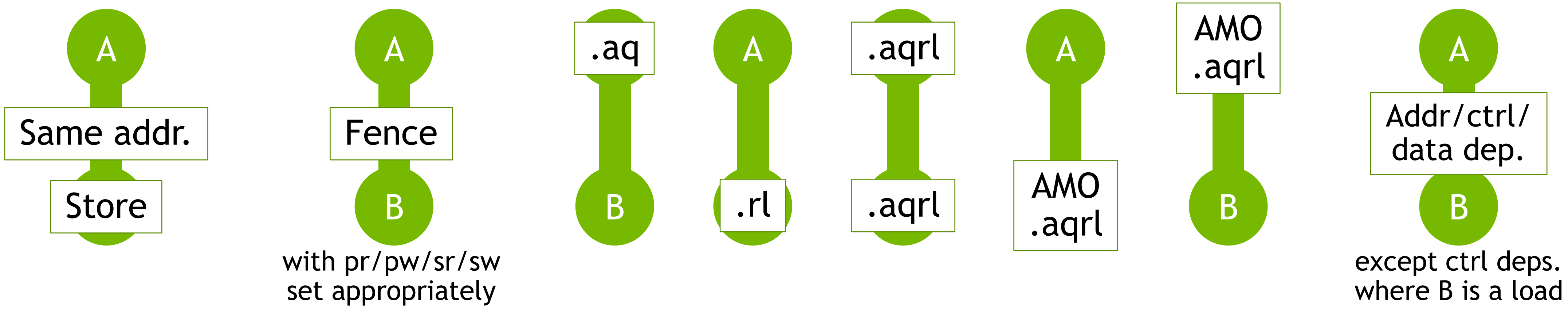
# ARCHITECTURAL INTUITION (FOR BOTH)


Pipelines
(In-order or OoO)
F
D
X
M
W
Hart-private buffering
(values can be forwarded only to loads from the same hart that issued that store)
Atomic Memory
(all values globally visible to all harts, possibly via a coherence protocol)
$
Last-level cache

- **RVWMO** and **RVTSO** differ in the degree of memory access reordering they permit at the point of global visibility
- **RVTSO**: only store→load reordering can be observed
- **RVWMO**: unless otherwise synchronized (e.g., via .aq, .rl, and/or fences), most memory accesses can be reordered freely

# RVWMO RULES IN A NUTSHELL

- A guaranteed to happen before B only if:



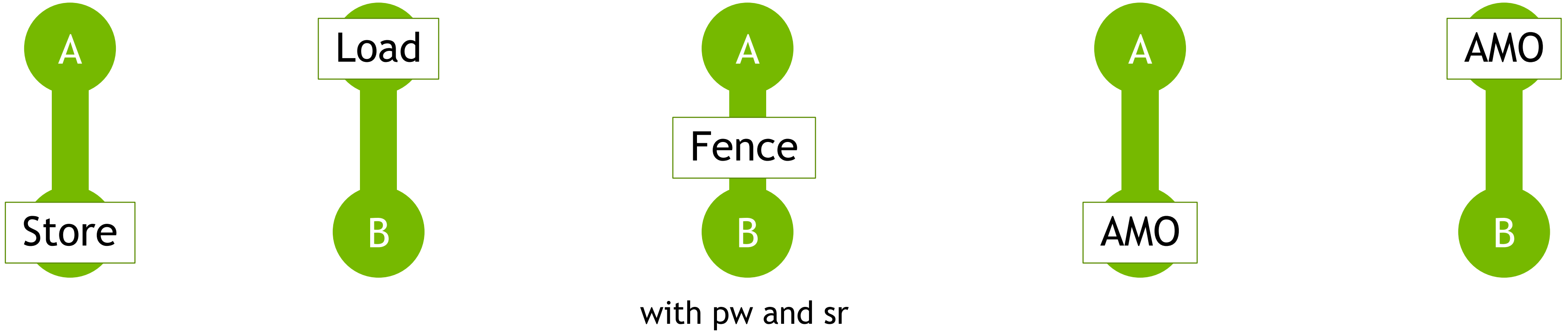
- Each load returns value from most recent store to same address
- No store from another hart can interrupt an AMO or LR/SC
- Available offline: Alloy/herd/operational model, lots of docs

(some last details still being finalized...)

# RVTSO RULES IN A NUTSHELL

Strictly stronger than RVWMO

- A guaranteed to happen before B only if:



- Each load returns value from most recent store to same address
- No store from another hart can interrupt an AMO or LR/SC
- To be made available: Alloy/herd/operational model, docs

# THE RISC-V MEMORY MODEL: SOFTWARE

- Portable SW must assume RVWMO, but it will run on both RVWMO HW and Ztso HW
  - Linux, gcc, bintools, … will target RVWMO by default
- RVTSO-only SW can be written, but it will only run on HW implementing Ztso
  - Object files will use different magic number

| | Base HW (RVWMO) | HW using Ztso ISA ext. |
|---|---|---|
| Standard SW (RVWMO) | OK | OK |
| RVTSO-only SW | **WILL NOT RUN** | OK |

## EVERYONE GETS WHAT THEY WANT

- **If you don’t want to think about memory models:** just use the standard OSes and toolchains
- **If you care about PPA or flexibility:** use RVWMO
- **If you have lots of legacy x86 code:** use HW implementing Ztso, so that any SW will work
- **If you believe TSO is the future:** use HW with Ztso, and emit code with the “TSO-only” magic number

## FRAGMENTATION

- Risk: SW will fragment into “RVWMO version” and “RVTSO version”, double the maintainer burden
- Solution: discourage the 2x model, and encourage RVWMO SW, since it’s portable across all HW
  - Current plan for Linux, gcc, binutils, etc.
  - Redundant fences simply become no-ops
- However: can’t stop people from customizing; RISC-V’s openness is a feature, not a bug

## OTHER ISA CHANGES

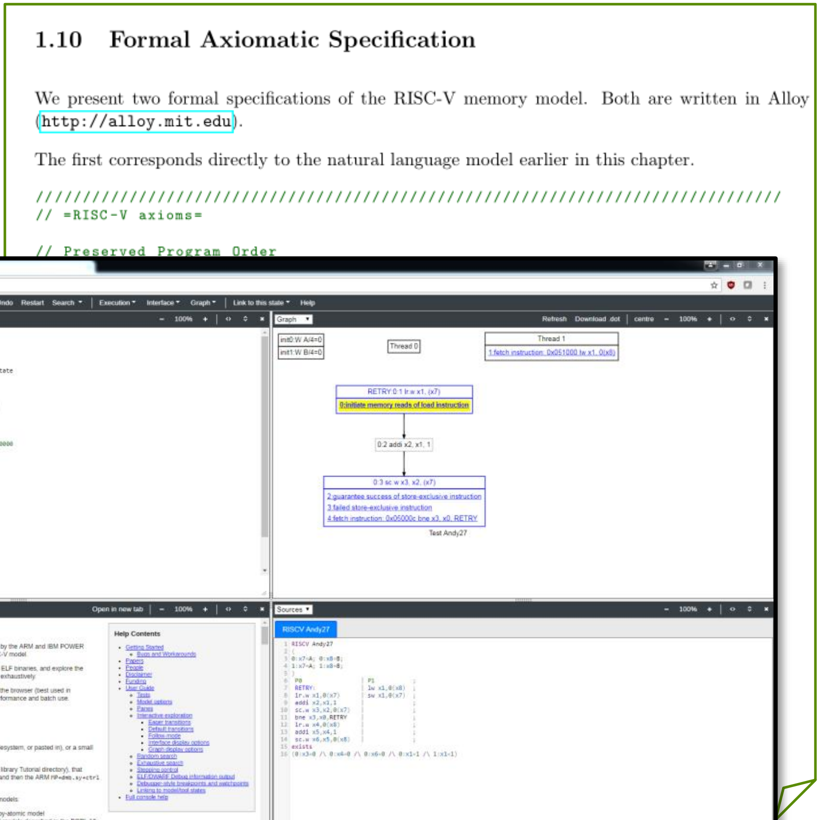
- ld.rl and sd.aq are deprecated
- ld.aqrl and sd.aqrl means RCsc, not fully fenced
- Clarified other subtleties of atomicity and LR/SCs
- Possibly in a future extension:
  - Fences may take an address restriction parameter
  - Opcodes for “l{b|h|w|d}.aq” and “s{b|h|w|d}.rl”

## I/O FENCES, FENCE.I, SFENCE.VM

- Informal descriptions given in memory model spec
- No change to I/O channel ordering behavior
- Some clarification to FENCE .pi/.po/.si/.so
- More detail available offline
- Future: V/T/J compatibility as well

# DOCUMENTATION & TOOLS

- Definitions of RVWMO/RVTSO and Ztso
- A dozen pages explaining the details in plain English
- Two axiomatic models (Alloy + herd)
- One formal operational model + app
- Lots of litmus tests
    - (also to be used to test compliance)
- …and the memory model TG as a resource to people who have questions
- <u>**Come find me or email me!**</u>



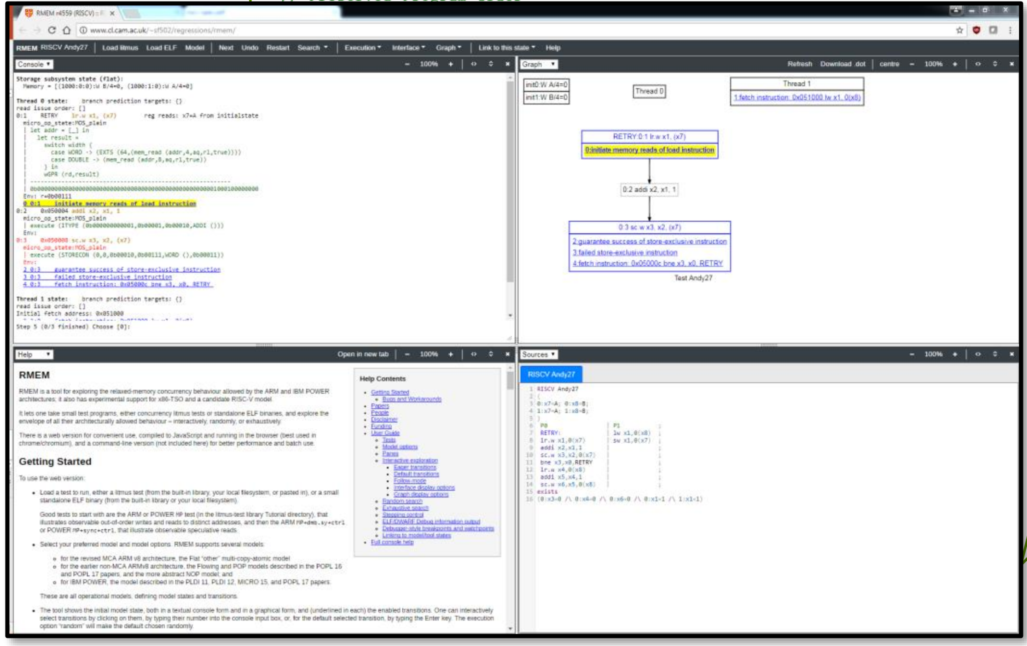
1.10 Formal Axiomatic Specification

We present two formal specifications of the RISC-V memory model. Both are written in Alloy (http://alloy.mit.edu).

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

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

// Preserved Program Order
```



```
Storage subsystem state (flat):
  Memory = [(1000:0:0):W B/4=0, (1000:1:0):W A/4=0]

Thread 0 state:    branch prediction targets: {}
read issue order: []
0:1    RETRY    lr.w x1, (x7)        reg reads: x7=A from initialstate
  micro_op_state:MOS_plain
  | let addr = [_] in
  |   let result =
  |     switch width {
  |       case WORD -> (EXTS (64,(mem_read (addr,4,aq,rl,true))))
  |       case DOUBLE -> (mem_read (addr,8,aq,rl,true))
  |     } in
  |     wGPR (rd,result)
  | ------------------------------------------------------------
  | 0b0000000000000000000000000000000000000000000001000100000000
  Env: r=0b00111
  0 0:1    initiate memory reads of load instruction
0:2    0x050004 addi x2, x1, 1
  micro_op_state:MOS_plain
  | execute (ITYPE (0b000000000001,0b00001,0b00010,ADDI ()))
  Env:
0:3    0x050008 sc.w x3, x2, (x7)
  micro_op_state:MOS_plain
  | execute (STORECON (0,0,0b00010,0b00111,WORD (),0b00011))
  Env:
  2 0:3    guarantee success of store-exclusive instruction
  3 0:3    failed store-exclusive instruction
  4 0:3    fetch instruction: 0x05000c bne x3, x0, RETRY

Thread 1 state:    branch prediction targets: {}
read issue order: []
Initial fetch address: 0x051000

Step 5 (0/3 finished) Choose [0]:
```

**RMEM**

RMEM is a tool for exploring the relaxed-memory concurrency behaviour allowed by the ARM and IBM POWER architectures; it also has experimental support for x86-TSO and a candidate RISC-V model.

It lets one take small test programs, either concurrency litmus tests or standalone ELF binaries, and explore the envelope of all their architecturally allowed behaviour – interactively, randomly, or exhaustively.

There is a web version for convenient use, compiled to JavaScript and running in the browser (best used in chrome/chromium), and a command-line version (not included here) for better performance and batch use.

**Getting Started**

To use the web version:

- Load a test to run, either a litmus test (from the built-in library, your local filesystem, or pasted in), or a small standalone ELF binary (from the built-in library or your local filesystem).

  Good tests to start with are the ARM or POWER MP test (in the litmus-test library Tutorial directory), that illustrates observable out-of-order writes and reads to distinct addresses, and then the ARM `MP+dmb.sy+ctrl` or POWER `MP+sync+ctrl`, that illustrate observable speculative reads.
- Select your preferred model and model options. RMEM supports several models:
  - for the revised MCA ARM v8 architecture, the Flat "other" multi-copy-atomic model
  - for the earlier non-MCA ARMv8 architecture, the Flowing and POP models described in the POPL 16 and POPL 17 papers, and the more abstract NOP model; and
  - for IBM POWER, the model described in the PLDI 11, PLDI 12, MICRO 15, and POPL 17 papers.

  These are all operational models, defining model states and transitions.
- The tool shows the initial model state, both in a textual console form and in a graphical form, and (underlined in each) the enabled transitions. One can interactively select transitions by clicking on them, by typing their number into the console input box, or, for the default selected transition, by typing the Enter key. The execution option "random" will make the default chosen randomly.

**Help Contents**

- Getting Started
  - Bugs and Workarounds
- Papers
- People
- Disclaimer
- Funding
- User Guide
  - Tests
  - Model options
  - Panes
  - Interactive exploration
    - Eager transitions
    - Default transitions
    - Follow mode
    - Interface display options
    - Graph display options
  - Random search
  - Exhaustive search
  - Stepping control
  - ELF/DWARF Debug information output
  - Debugger-style breakpoints and watchpoints
  - Linking to model/tool states
- Full console help

```
RISCV Andy27
{
0:x7=A; 0:x8=B;
1:x7=A; 1:x8=B;
}
 P0                   | P1          ;
 RETRY:               | lw x1,0(x8) ;
 lr.w x1,0(x7)        | sw x1,0(x7) ;
 addi x2,x1,1         |             ;
 sc.w x3,x2,0(x7)     |             ;
 bne x3,x0,RETRY      |             ;
 lr.w x4,0(x8)        |             ;
 addi x5,x4,1         |             ;
 sc.w x6,x5,0(x8)     |             ;
exists
(0:x3=0 /\ 0:x4=0 /\ 0:x6=0 /\ 0:x1=1 /\ 1:x1=1)
```

## RISC-V MEMORY MODEL ROADMAP

- We’ll discuss this further Thursday at the members-only meeting
  - Or come find me in the meantime
- We’ll aim to release the complete documentation publicly (on isa-dev) soon after that
- If all goes well, we’ll work with the Technical Committee to work towards ratification