Formal Specification of RISC-V Systems Instructions

Arvind

Andy Wright, Sizhuo Zhang, Thomas Bourgeat, Murali Vijayaraghavan

Computer Science and Artificial Intelligence Lab. MIT

RISC-V Workshop, MIT, Cambridge, MA

July 12, 2016

Chips with Proofs

joint project with *Adam Chlipala*

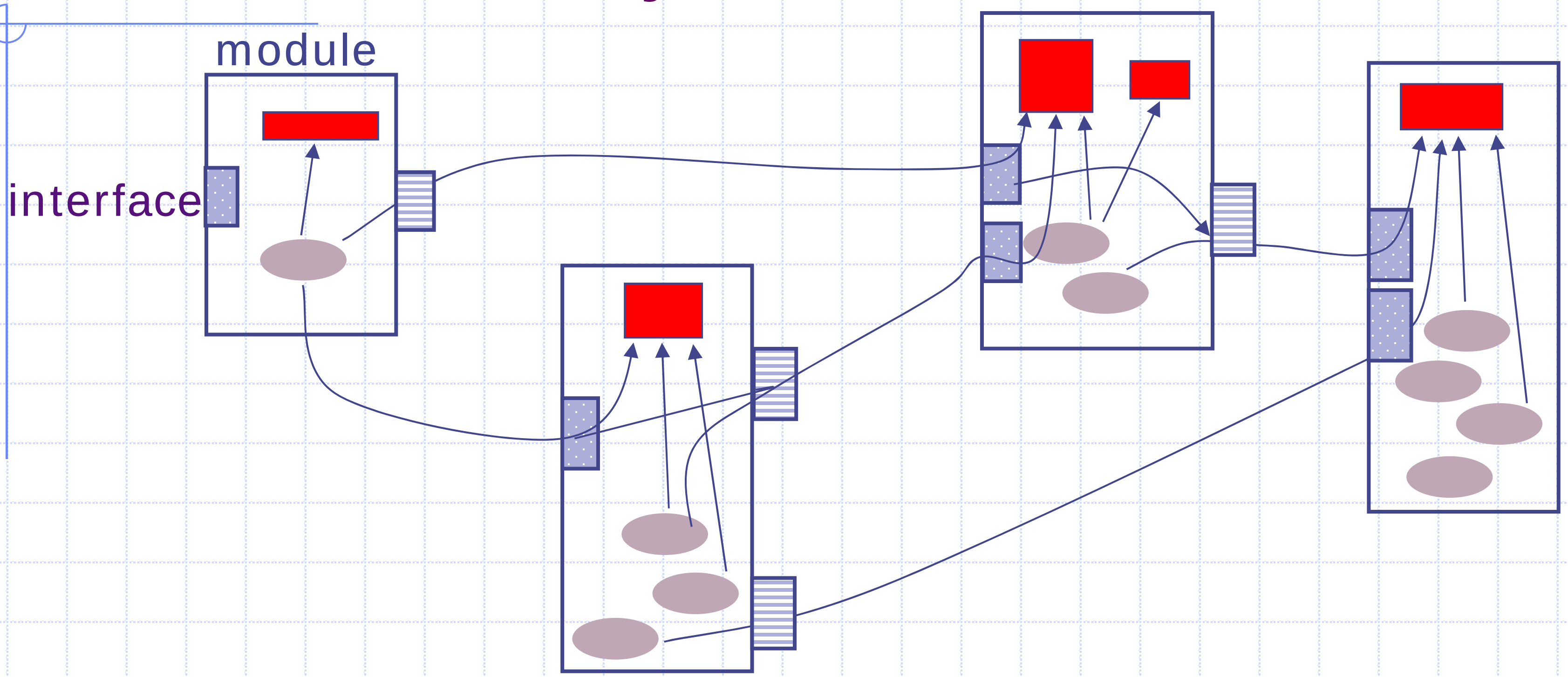
RISC V ISA specifications

- RISC V multicore chips that satisfy RISC V ISA specifications
  - Must boot Linux
- “No compromise” implementations – both in-order and out-of-order processors
  - Our current implementations boot Linux
- Both the design and the proofs must be modular and amenable to modular refinement
  - Need for modular specifications, e.g., specification of a processor without being connected to memory
- Mostly concerned about microarchitecture and memory system correctness
  - Taking the correctness of arithmetic for granted

Specifications

- Strongly prefer operational semantics
- Specify semantics in terms of simple and abstract machines
- Express implementations in the same style and prove following types of theorems
  - $[\![P_S]\!] + [\![M_S]\!] \sqsubseteq [\![P_S \oplus M_S]\!]$
  - $[\![P_{I'}]\!] \sqsubseteq [\![P_I]\!] \implies [\![P_{I'}]\!] + [\![M_I]\!] \sqsubseteq [\![P_I]\!] + [\![M_I]\!]$
- Express specs and designs in Bluespec, a language based on *guarded atomic actions*
  - Bluespec compiler generates RTL in Verilog; thus, all specs are executable. However, exploration of non-determinism in a systematic way is an issue
- Kami is a framework in Coq for doing proofs about Bluespec programs

Bluespec: System as a set *concurrent objects*

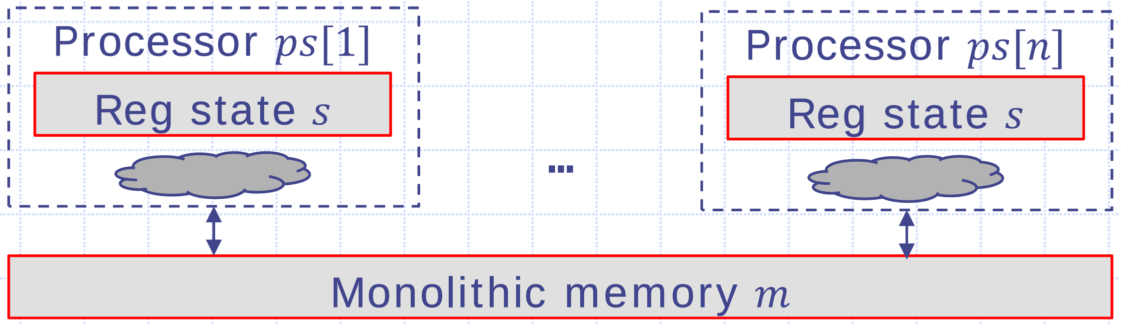


- Only way to observe or manipulate the state 🟥 of an object is through its interface methods: def ▨ use ▤
- Latency-insensitive interfaces
- The state within a module and across modules is controlled using *guarded atomic actions* ⬬

ISA specification

- Specification should *avoid* using concepts like:
  - partially executed instructions,
  - “a store has been performed wrt …”
- Nondeterminism in semantics is essential to capture multiprocessor behavior, missing fences etc.
- Unspecified behavior should be avoided at all cost
  - “we specify behaviors for Data Race Free Programs”
  - Such constrains may be acceptable for high-level software if they can be verified statically
- We will use the *Instantaneous Instruction Execution (I^2E)* framework

Instantaneous Instruction Execution (I^2E) Framework



- An instruction executes instantaneously; processor state is always up-to-date
- Monolithic memory processes loads and stores instantaneously
- Data moves between processors and memory asynchronously according to some background rules
    - Memory-Model specific buffers between ps[i] and m

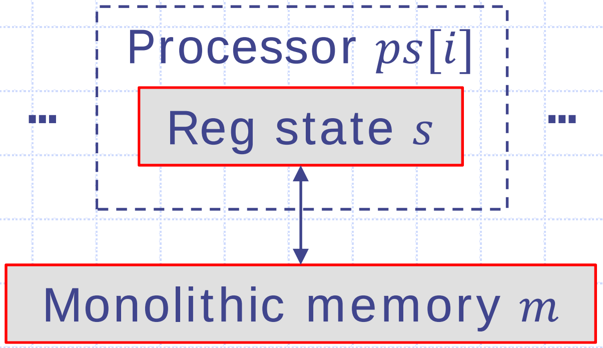
Basic problem – concurrency & nondeterminism:
Multiple paths to the same memory

Methods to manipulate the register state s (includes pc)

- decode(): fetches the next instruction and returns the decoded and partially executed version of it. Results of decode:
  - *Non-memory instruction* <Nm, dst, v>: Write computation result v into destination register dst
  - *Load* <Ld, a, dst>: read memory address a and update dst
  - *Store* <St, a, v>: write value v to memory address a
  - Other miscellaneous instructions: atomic read-modify-write, fences, …
- execute(ins, ldRes): updates register state s based on decoded instruction ins. A Ld requires a second argument ldRes, which should be the value supplied by the memory system

SC in I^{2}E

- Ld and St directly access the monolithic memory



SC-Nm rule

$$\frac{\langle Nm, dst, v\rangle = ps[i].decode()}{ps[i].execute(\langle Nm, dst, v\rangle, -)}$$

SC-Ld rule

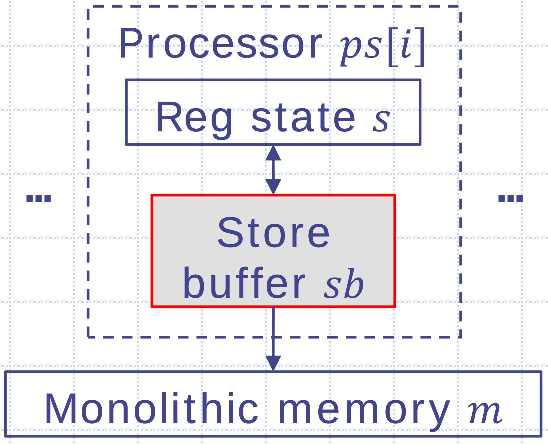
$$\frac{\langle Ld, a, dst\rangle = ps[i].decode()}{ps[i].execute(\langle Ld, a, dst\rangle, m[a])}$$

SC-St rule

$$\frac{\langle St, a, v\rangle = ps[i].decode()}{ps[i].execute(\langle St, a, v\rangle, -) \quad m[a] \Leftarrow v}$$

TSO in I^{2}E

- TSO: Includes store buffer sb
  - St goes into sb in execution
  - Ld first searches sb for data forwarding
  - St is moved from sb to m in the background
  - A “Commit” fence to flush sb



TSO-Nm rule

$$\frac{\langle Nm, dst, v\rangle = ps[i].decode()}{ps[i].execute(\langle Nm, dst, v\rangle, -)}$$

TSO-Background rule

$$\frac{when(\neg ps[i].sb.empty())}{\langle a, v\rangle \leftarrow ps[i].sb.deq() \qquad m[a] \Leftarrow v}$$

TSO-St rule

$$\frac{\langle St, a, v\rangle = ps[i].decode()}{ps[i].execute(\langle St, a, v\rangle, -) \qquad ps[i].sb.enq(a, v)}$$

TSO-Com rule

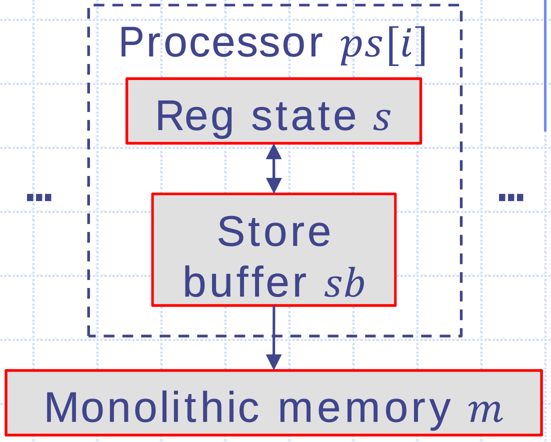
$$\frac{\langle Commit\rangle = ps[i].decode() \quad when(ps[i].sb.empty())}{ps[i].execute(\langle Commit\rangle, -)}$$

TSO-Ld rule

$$\frac{\langle Ld, a, dst\rangle = ps[i].decode() \quad v = if\ a \in ps[i].sb\ then\ ps[i].sb.youngest(a)\ else\ m[a]}{ps[i].execute(\langle Ld, a, dst\rangle, v)}$$

TSO in I^{2}E

- Non-determinism of the background rule with processor rules allows us to capture the permitted instruction-reorderings in TSO

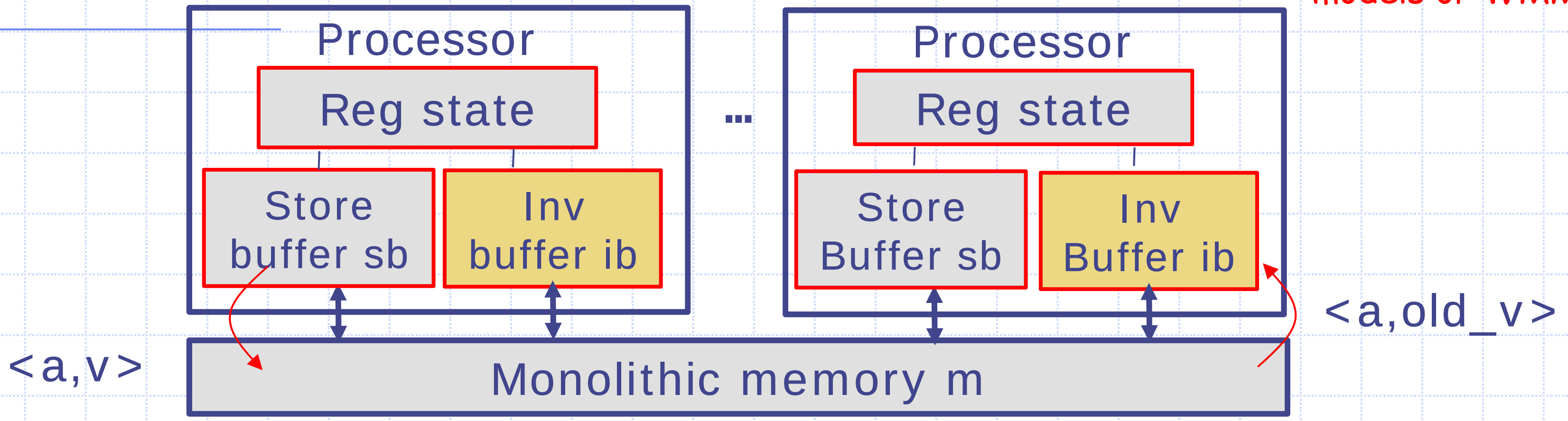


- PSO can be modeled similarly just by changing the background rule

$$\frac{a = ps[i].sb.anyAddr() \quad when(a \neq \varepsilon)}{v \leftarrow ps[i].sb.rmOldest(a) \quad m[a] \Leftarrow v}$$

WMM- A possible memory model for RISC-V

This talk is not about memory models or WMM



- Introduce *Invalidation Buffers* (ib), a conceptual device to make stale values visible
  - Whenever <*a*,*v*> from sb is moved to the memory, the old value for *a* in memory is inserted into ib of all other processors, and all values for *a* are purged from the local ib
  - A load may read values from ib or m if the address is not found in sb; staler values than the one read are purged from ib
  - A Reconcile fence clears the invalidation buffer

Memory issues within a uniprocessor

- Self modifying code and Instruction cache
  - I-cache is not coherent with respect to stores

- Page table access and TLB
  - Hardware page-table walks; may do background writes in the page table
  - Software page-table updates
  - TLB is not coherent with respect to stores

Basic problem:

Multiple paths to the same memory

Self modifying code

- Instruction cache is not coherent with the memory or store buffer
  - Simplifies hardware implementations
  - Justified on the grounds that self-modifying code is rare
  - User is aware of code self-modification
- Typical solution
  - User flushes the I-Cache

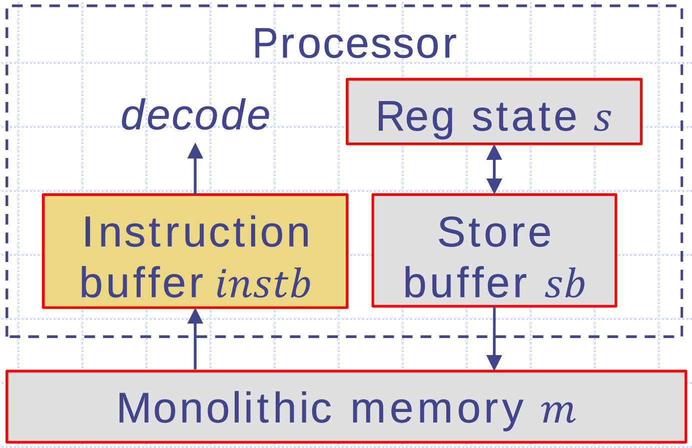
RISC-V solution is a FENCE.I instruction to signal the required synchronization between I-Cache and the memory system; most implementations flush store buffer and I-Cache and re-fetch the next instruction

Meaning of FENCE.I

- We assume an instruction buffer and store buffer in the abstract machine; the accesses on these two paths are not coherent
- FENCE.I instruction does the following actions atomically
  - Commit sb to m
  - Reconcile instb

FENCE.I rule

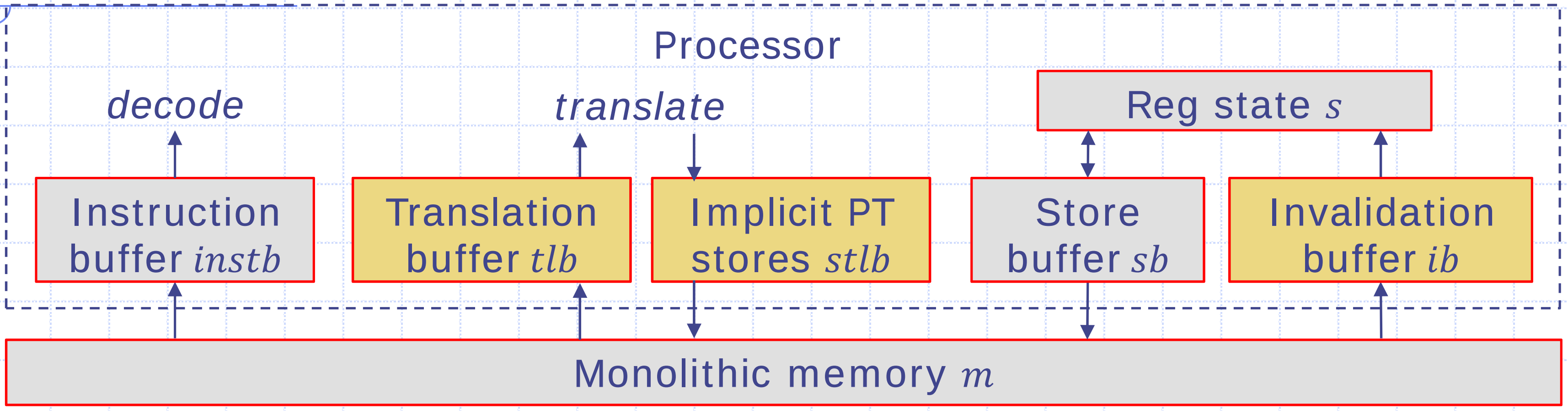
$$\frac{\begin{array}{c}\langle FENCE.I\rangle = ps.decode() \\ when(ps.sb.empty())\end{array}}{\begin{array}{c}ps.execute(\langle FENCE.I\rangle, -) \\ ps.instb.clear()\end{array}}$$



Page Table (PT) Management

- RISC-V specifies that PT walks should be done in hardware
  - Hardware PT walks are in charge of setting dirty and referenced bits in PT entries (*implicit writes*)
  - Additional PT management is done in software using normal load and store instructions in kernel mode
  - No guarantee of coherency in these two paths
- RISC-V solution
  - SFENCE.VM instruction to perform synchronization between software updates and hardware reads of PT
- The solution is incomplete
  - New instruction is required to synchronize implicit hardware updates with software reads

Virtual memory instructions



- SFENCE.TS.DL (new instruction)
  - Atomically commits stlb to m, and reconciles ib

$$\frac{\langle SFENCE.TS.DL\rangle = ps.decode() \quad when(ps.stlb.empty())}{ps.execute(\langle SFENCE.TS.DL\rangle, -) \quad ps.ib.clear()}$$

- SFENCE.DS.TL (new name for SFENCE.VM)
  - Atomically commits sb to m, and reconciles tlb

$$\frac{\langle SFENCE.DS.TL\rangle = ps.decode() \quad when(ps.sb.empty())}{ps.execute(\langle SFENCE.DS.TL\rangle, -) \quad ps.tlb.clear()}$$

More on fence instructions

- Our abstract machine has joint TLB for instructions and data. However this does not restrict implementations
- For performance, we may want to delete specific TLB entries as opposed to flushing the whole TLB
  - Can be easily incorporated
- Fence instruction effects are local; for multicores, higher level software protocol is needed to get other processors to take the appropriate actions
- It may be preferable to have the following instructions
  - instb.Reconcile
  - stlb.Commit (half of SFENCE.TS.DL)
  - tlb.Reconcile page_number (half of SFENCE.DS.TL)
  - ib.Reconcile
  - sb.Commit

} Needed for multicores anyway

Work in progress

- This talk represents our current thinking – not the final proposal
- Memory model for RISC-V
  - SC?, TSO?, weaker model than SC or TSO
  - One or more models (RISC-V-SC, RISC-V-WMM,...)
- Not sure of several issues, e.g.,
  - Should physical addresses be visible inside a processor?
  - Should memory system sees only physical addresses?
  - Should the abstract machine assume a special datapath between the local instruction buffer and store buffer?

We would like community participation in settling these issues

thanks