Demystifying Memory Models Across the Computing Stack

Yatin A. Manerkar, Caroline Trippel, Margaret Martonosi

Princeton University

ISCA 2019

While you wait:

1) Make sure you've got VirtualBox downloaded to your laptop:

https://www.virtualbox.org/wiki/Downloads

2) Make sure you have Tutorial VM downloaded (or use one of the USB drives):

http://check.cs.princeton.edu/tutorial_vm/Check_Tools_VM_2019.ova

VM Password: mcmsarefun

http://check.cs.princeton.edu/tutorial.html

Goals

- Reestablish the basics: Why Memory Consistency Models matter… more than ever!
- Give you concrete tools and techniques for broader MCM research
- Foster a broader community conversant and active in MCM issues
- Show connections outwards to other topics: Security, Distributed Systems, etc.
- Get you thinking about future research possibilities in this area

Our Approach Today

- Start from basic knowledge of Memory Consistency Models
  - Instruction at level of first-year graduate student
  - Will give background info.
  - If it's too basic or too fast, say so.
- Variety is the spice of life… Intersperse:
  - Theory
  - Techniques
  - Tool specifics
  - Demos

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **“Answer is: 42”**?

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **“Answer is: 42”**? **Yes**, eg: ❶ ❷ ❸ ❹

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **“Answer is: 42”**? **Yes**, eg: ❶ ❷ ❸ ❹

How about just **“42”**?

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **“Answer is: 42”**? **Yes**, eg: ❶ ❷ ❸ ❹

How about just **“42”**? **Yes**, eg: ❶ ❸ ❹ ❷

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **“Answer is: 42”**? **Yes**, eg: ❶ ❷ ❸ ❹

How about just **“42”**? **Yes**, eg: ❶ ❸ ❹ ❷

Could it print **nothing**?

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **“Answer is: 42”**? **Yes**, eg: ❶ ❷ ❸ ❹

How about just **“42”**? **Yes**, eg: ❶ ❸ ❹ ❷

Could it print **nothing**? **Yes**, eg: ❸ ❹ ❶ ❷

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

Can it print **"Answer is: 42"**? **Yes**, eg: ❶ ❷ ❸ ❹

How about just **"42"**? **Yes**, eg: ❶ ❸ ❹ ❷

Could it print **nothing**? **Yes**, eg: ❸ ❹ ❶ ❷

These executions obey **Sequential Consistency (SC)** [Lamport79], which requires that the results of the overall program correspond to some in-order interleaving of the statements from each individual thread.

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

How about **“Answer is:”**?

❷ ❶ ❸ ❹

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

How about **“Answer is:”**? **It depends!** ❷ ❶ ❸ ❹

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

How about **“Answer is:”**?

It depends!

❷ ❶ ❸ ❹

NO!

What does this program print?

Thread 0	Thread 1
❶ `x = 1;`	❸ `if (y == 1)` `  print("Answer is:");`
❷ `y = 1;`	❹ `if (x == 1)` `  print("42");`

How about **"Answer is:"**?

It depends!

❷ ❶ ❸ ❹

NO!

YES!

arm

What does this program print?

Thread 0	Thread 1
1 x = 1;	3 if (y == 1)

Why would we reorder memory operations?

How to specify what's allowed and forbidden?

How do check that implementations match spec?

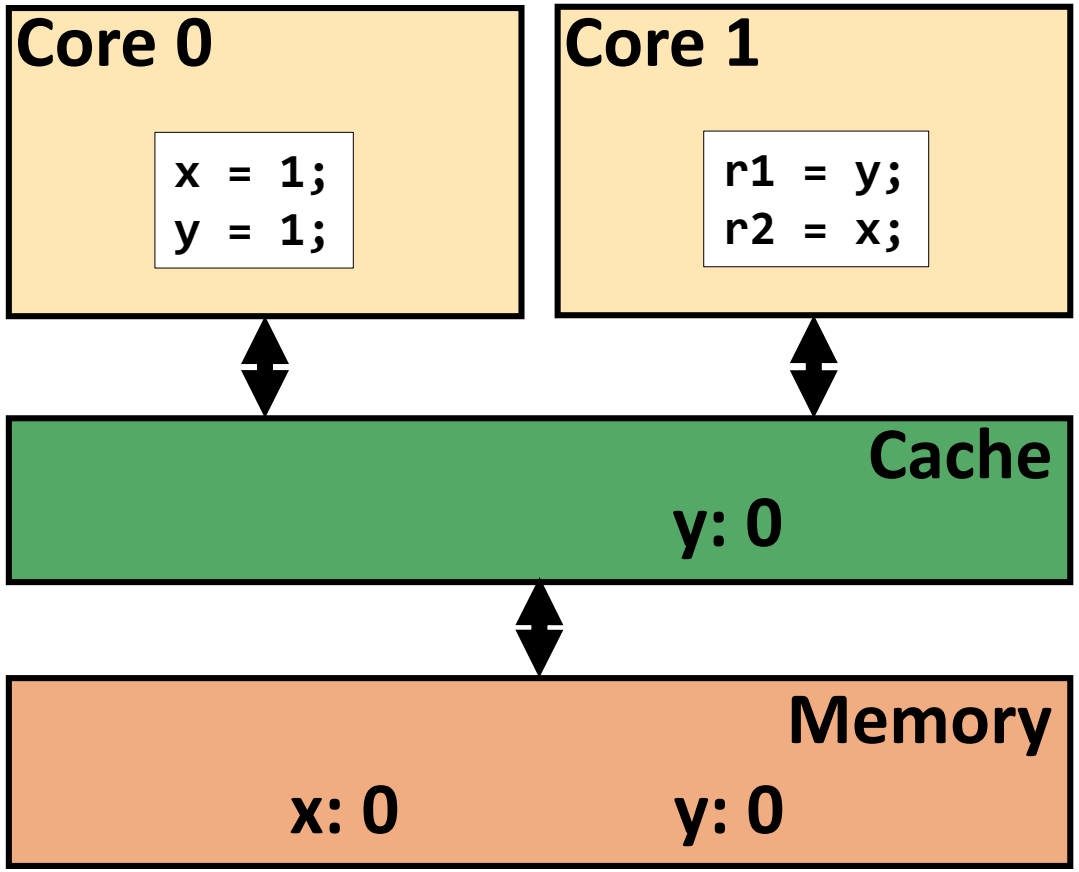
We'll cover the answers today!

Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
`x = 1;` `y = 1;`	`r1 = y;` `r2 = x;`
`Can r1=1 and r2=0?`	

Answer: Performance!

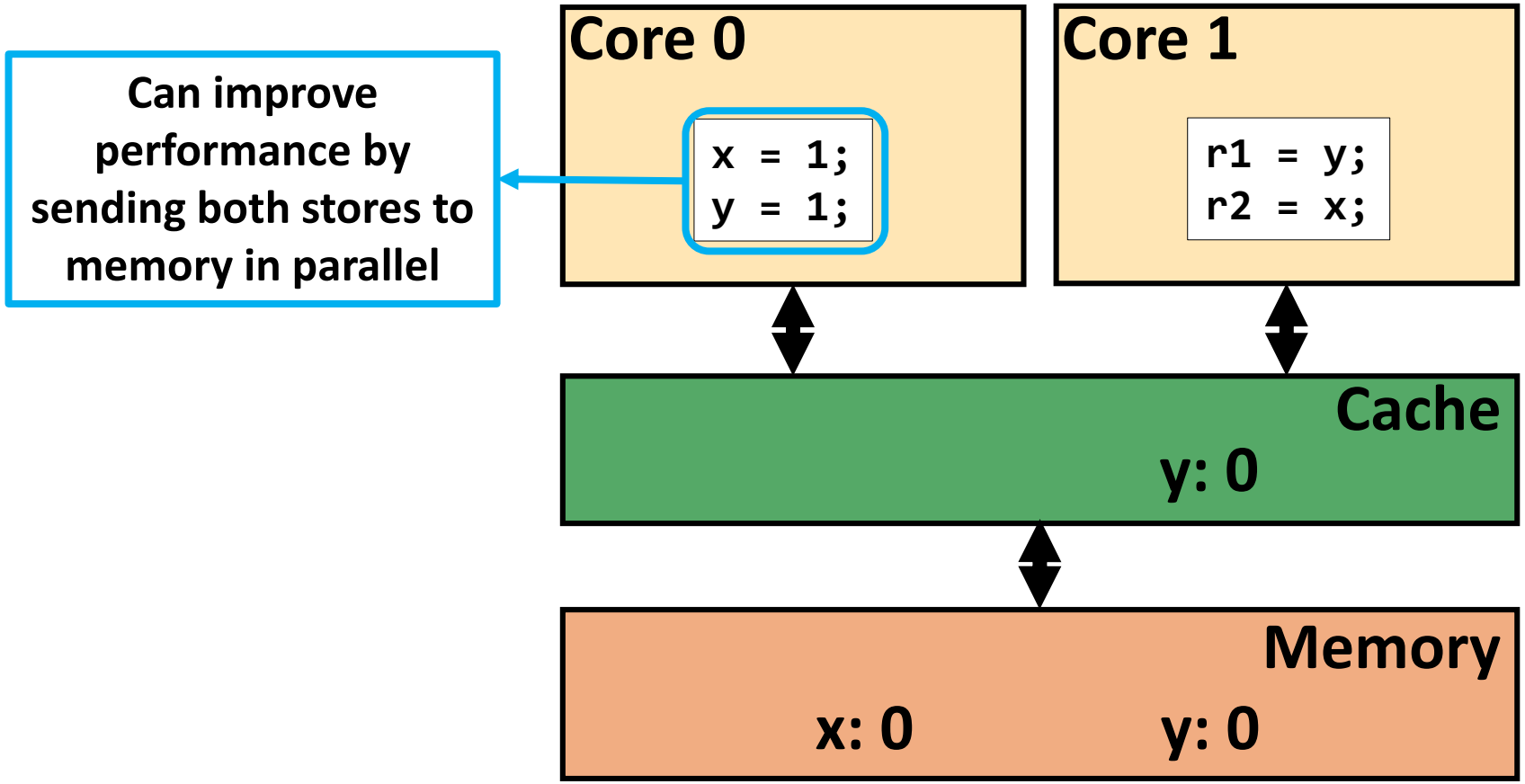


Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
`x = 1;` `y = 1;`	`r1 = y;` `r2 = x;`
`Can r1=1 and r2=0?`	

Answer: Performance!

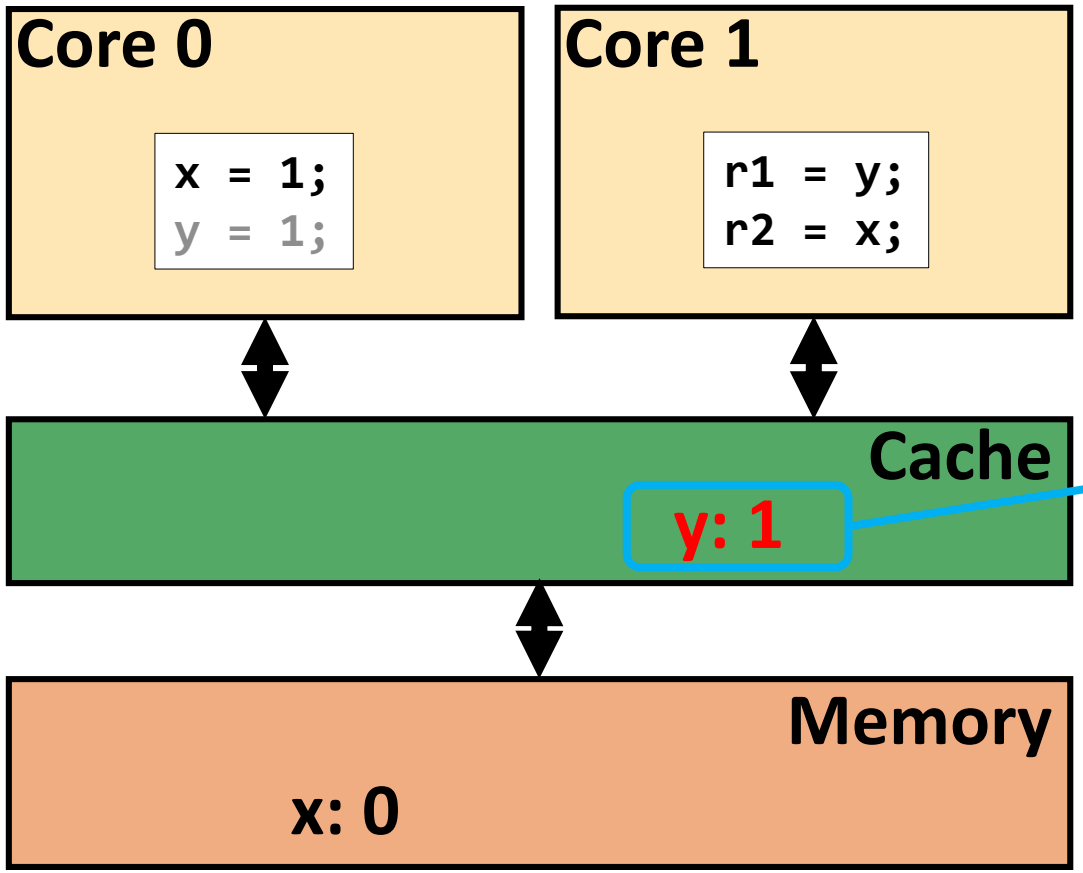


Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
x = 1; y = 1;	r1 = y; r2 = x;
Can r1=1 and r2=0?	

Answer: Performance!



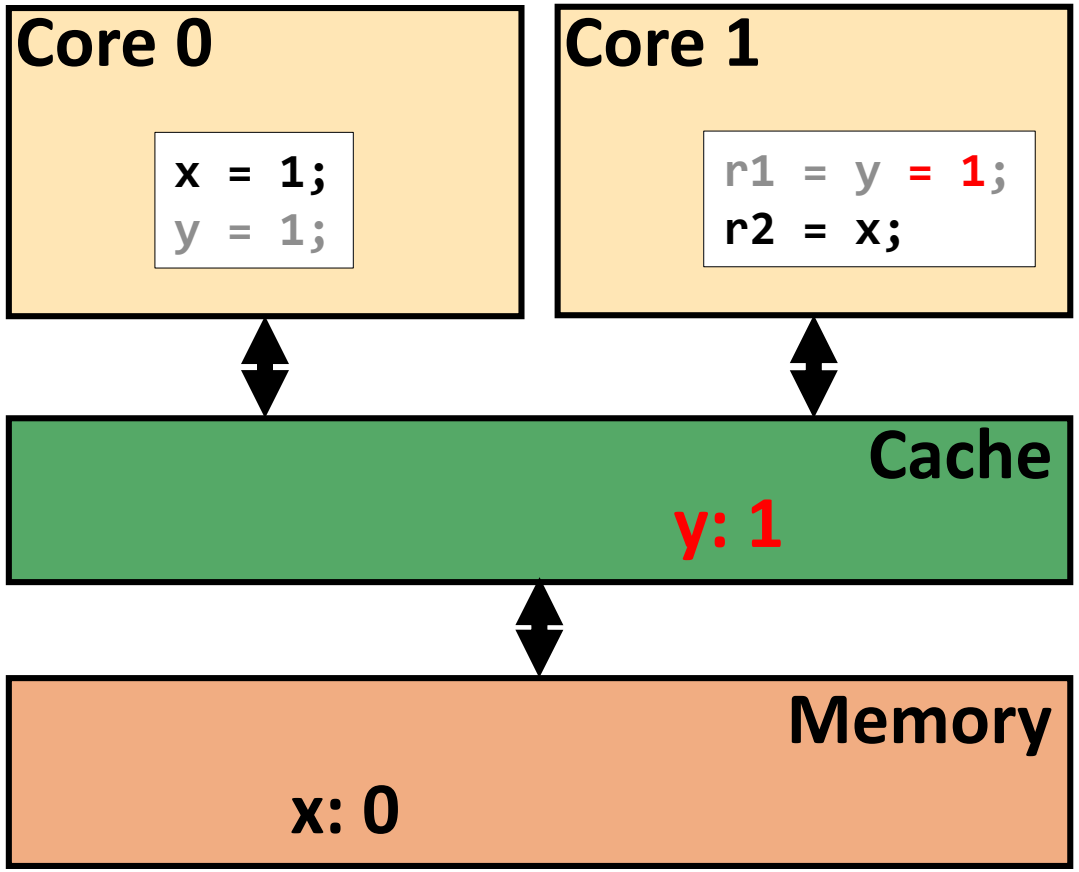
Store to y finishes quickly in cache

Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
`x = 1;` `y = 1;`	`r1 = y;` `r2 = x;`
`Can r1=1 and r2=0?`	

Answer: Performance!

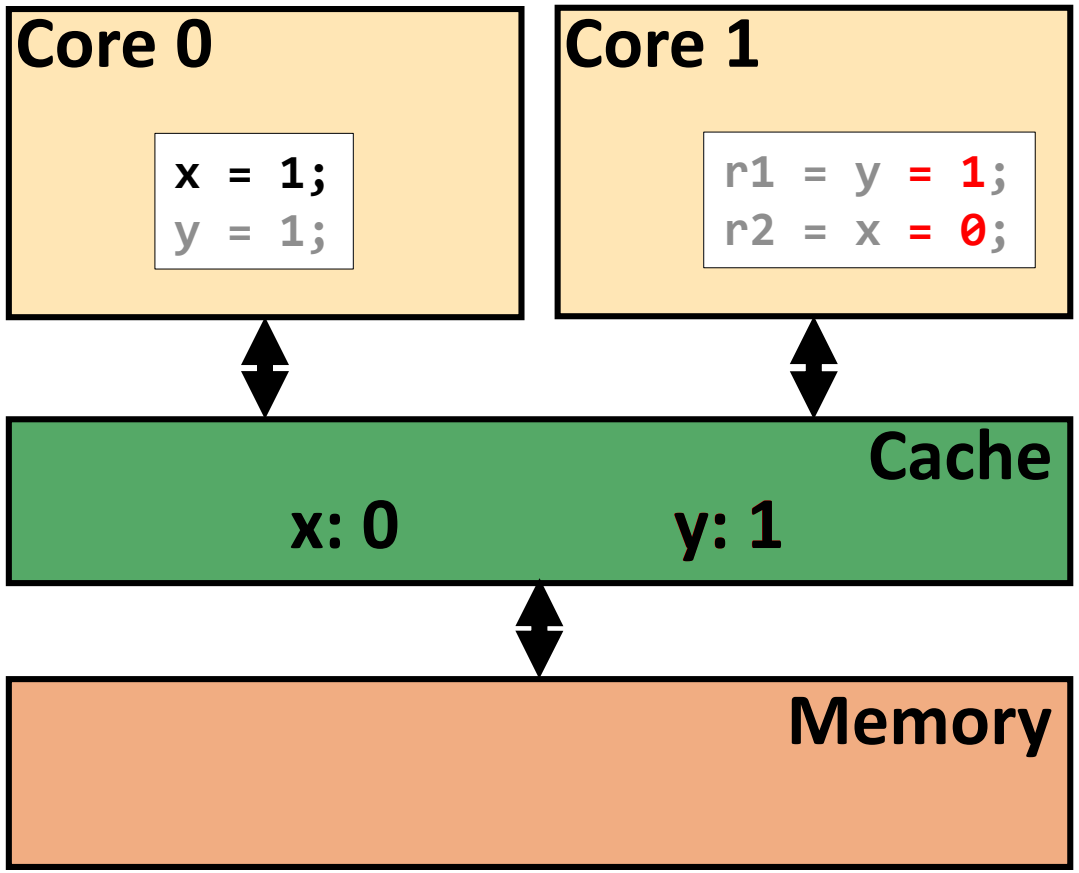


Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
x = 1; y = 1;	r1 = y; r2 = x;
Can r1=1 and r2=0?	

Answer: Performance!

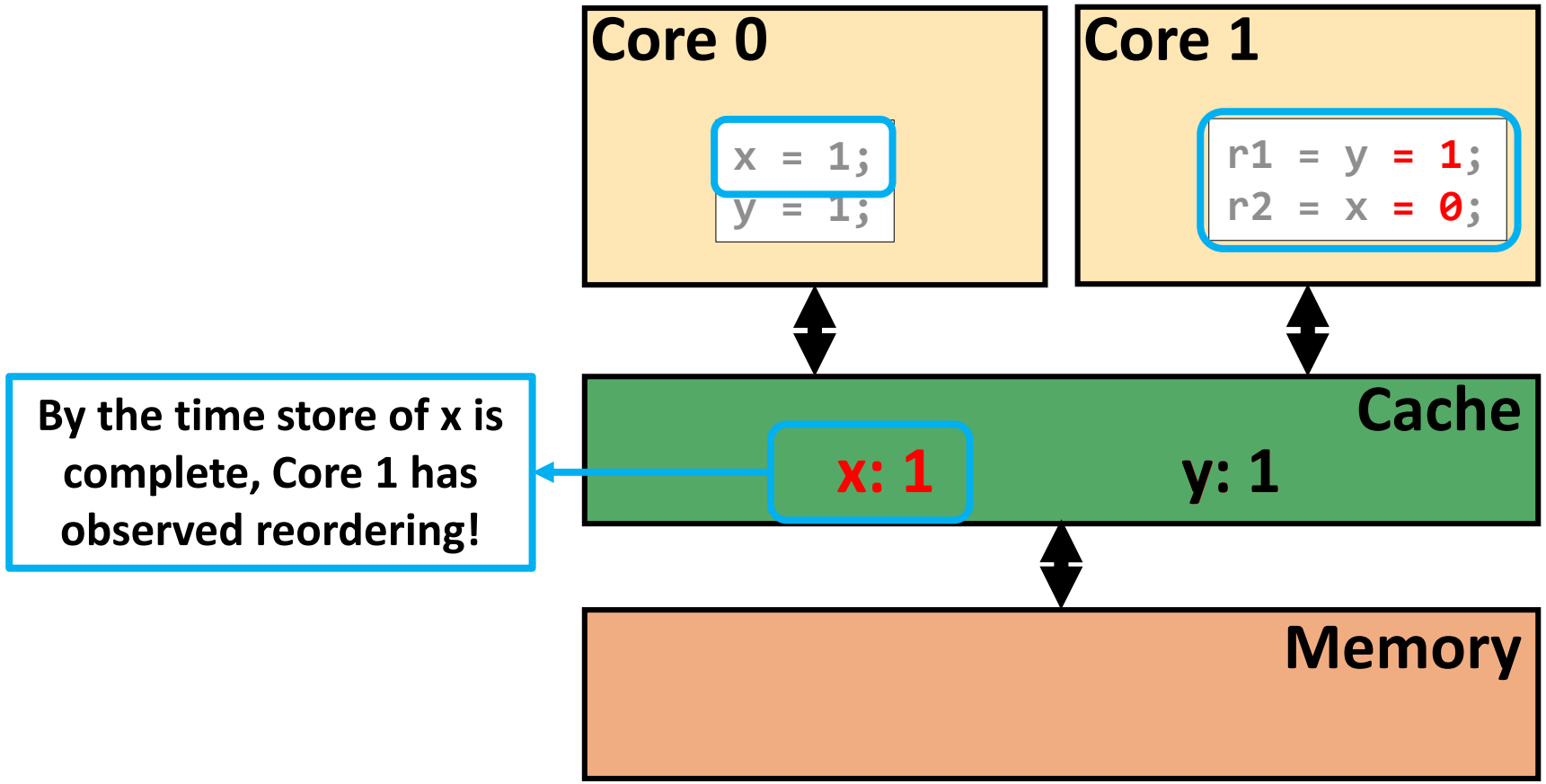


Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
`x = 1;` `y = 1;`	`r1 = y;` `r2 = x;`
`Can r1=1 and r2=0?`	

Answer: Performance!



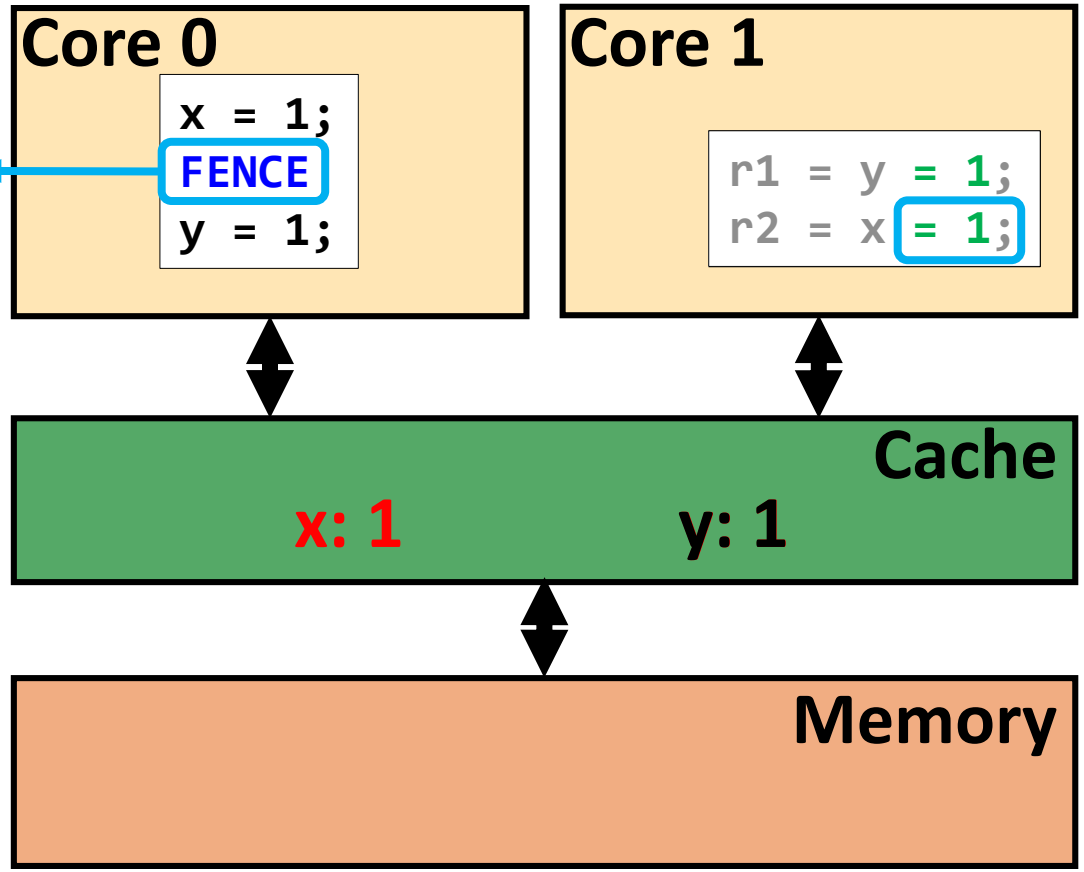
Why reorder memory operations?

Message Passing (mp)

Core 0	Core 1
`x = 1;` `y = 1;`	`r1 = y;` `r2 = x;`
`Can r1=1 and r2=0?`	

Answer: Performance!

Fence/synchronization instructions can enforce order between memory operations where needed



Compilers Reorder Memory Operations Too!

- Compiler optimizations can also result in weak memory behaviours
  - Example below: assume CPU performs instrs in order and 1 at a time

Thread 0	Thread 1
❶ x = 1; ❷ y = 1; ❸ x = 2;	❹ r1 = y; ❺ r2 = x;
Can r1 = 1 and r2 = 0?	

Compilers Reorder Memory Operations Too!

- Compiler optimizations can also result in weak memory behaviours
  - Example below: assume CPU performs instrs in order and 1 at a time

Compiler may coalesce these 2 stores (since no same-thread reads of x in between)

Thread 0	Thread 1
1 x = 1; 2 y = 1; 3 x = 2;	4 r1 = y; 5 r2 = x;
Can r1 = 1 and r2 = 0?	

Compilers Reorder Memory Operations Too!

- Compiler optimizations can also result in weak memory behaviours
  - Example below: assume CPU performs instrs in order and 1 at a time

Thread 0	Thread 1
❷ y = 1; ❸ x = 2;	❹ r1 = y; ❺ r2 = x;
Can r1 = 1 and r2 = 0?	

Compilers Reorder Memory Operations Too!

- Compiler optimizations can also result in weak memory behaviours
  - Example below: assume CPU performs instrs in order and 1 at a time

Thread 0	Thread 1
❷ y = 1; ❸ x = 2;	❹ r1 = y; ❺ r2 = x;
Can r1 = 1 and r2 = 0?	

Now ❷❹❺❸ gives **r1 = 1 and r2 = 0**!

Memory Consistency Models (MCMs)

- ISA instructions represent hardware operations (add, sub, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Compiler

Microarchitecture[1]

[1]Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

Memory Consistency Models (MCMs)

- ISA instructions represent hardware operations (add, sub, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Which compiler optimizations can I use?

Compiler

Microarchitecture[1]

[1]Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

Memory Consistency Models (MCMs)

- ISA instructions represent hardware operations (add, sub, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Which compiler optimizations can I use?

Compiler

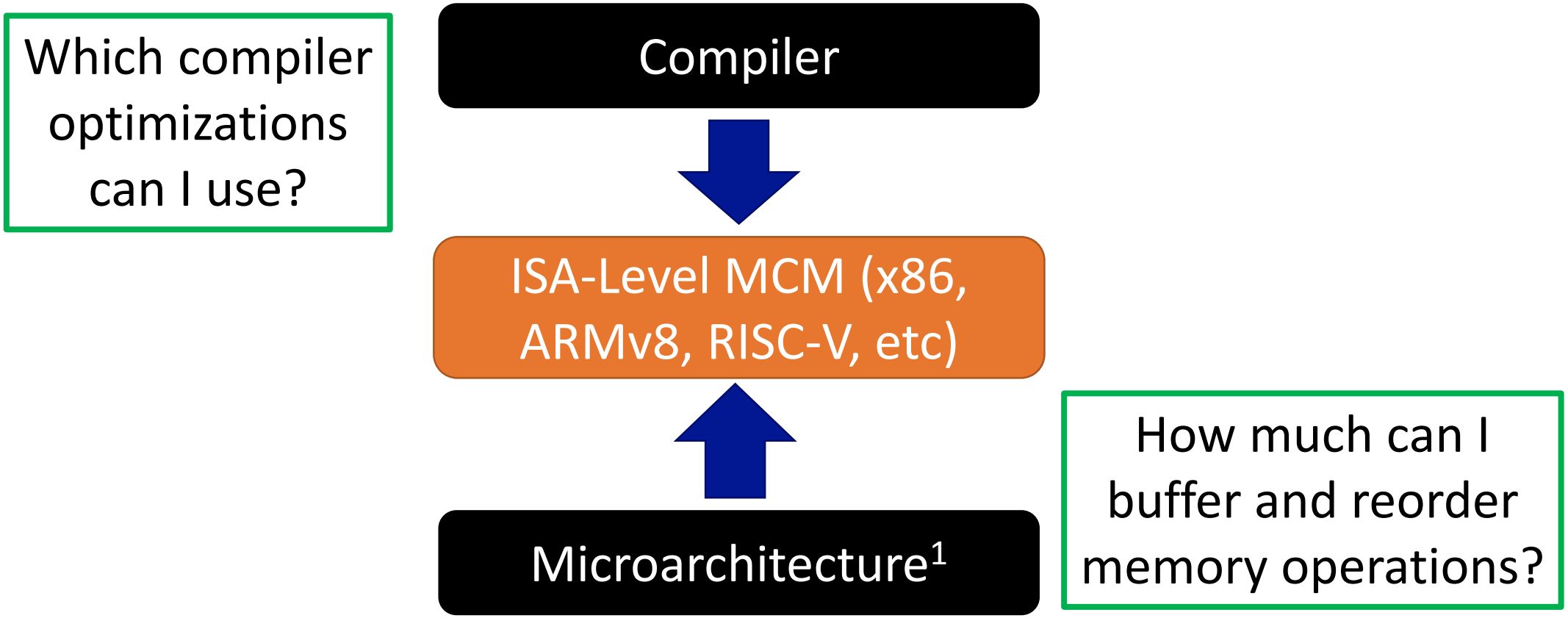
Microarchitecture[1]

How much can I buffer and reorder memory operations?

[1]Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

Memory Consistency Models (MCMs)

- ISA instructions represent hardware operations (add, sub, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops



[1]Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

Memory Consistency Models (MCMs)

- ISA instructions represent hardware operations (add, sub, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

In a nutshell: MCMs specify what value will be returned when your program does a load!

ARMv8, RISC-V, etc)

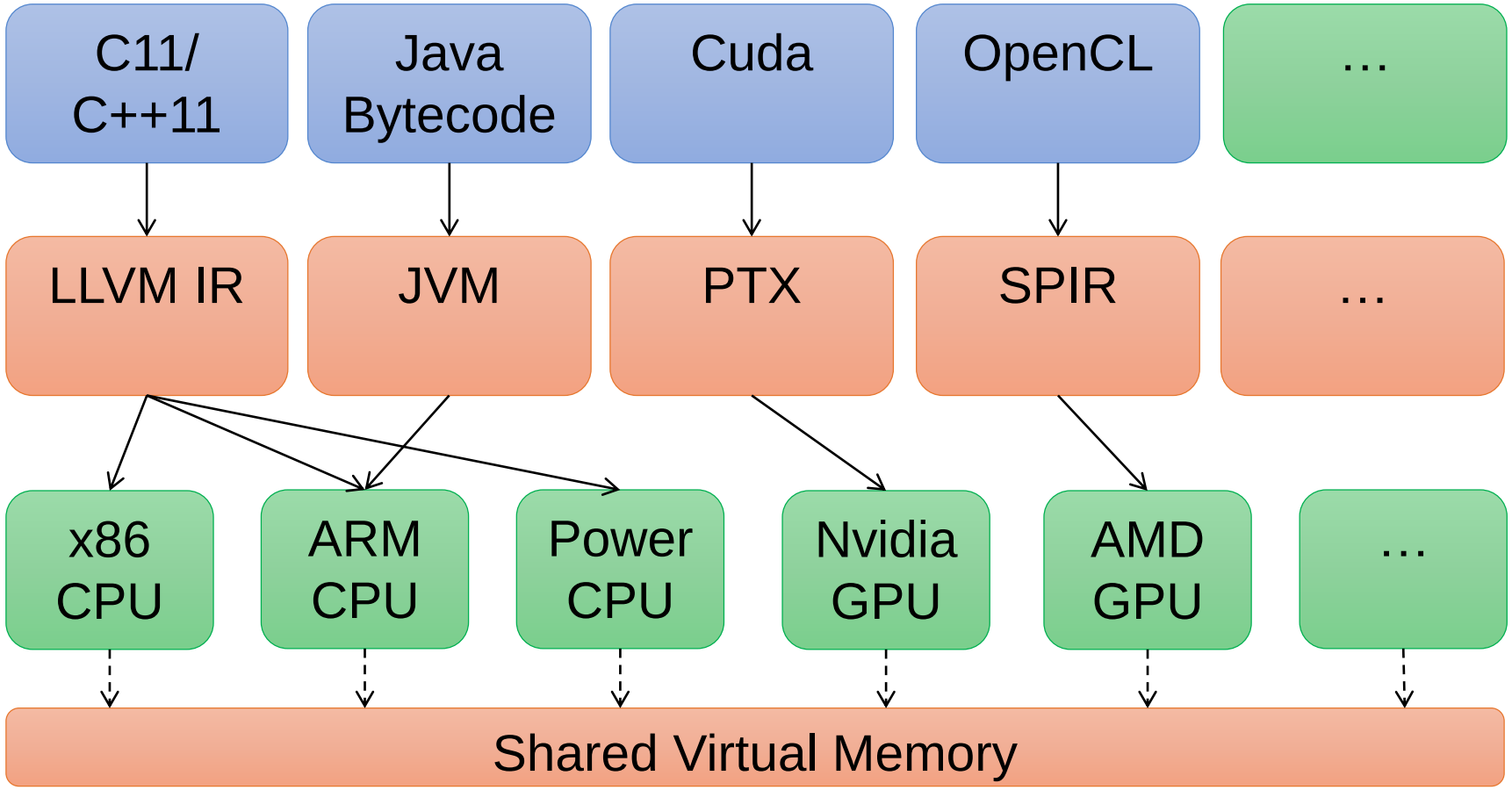
Microarchitecture[1]

How much can I buffer and reorder memory operations?

[1]Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

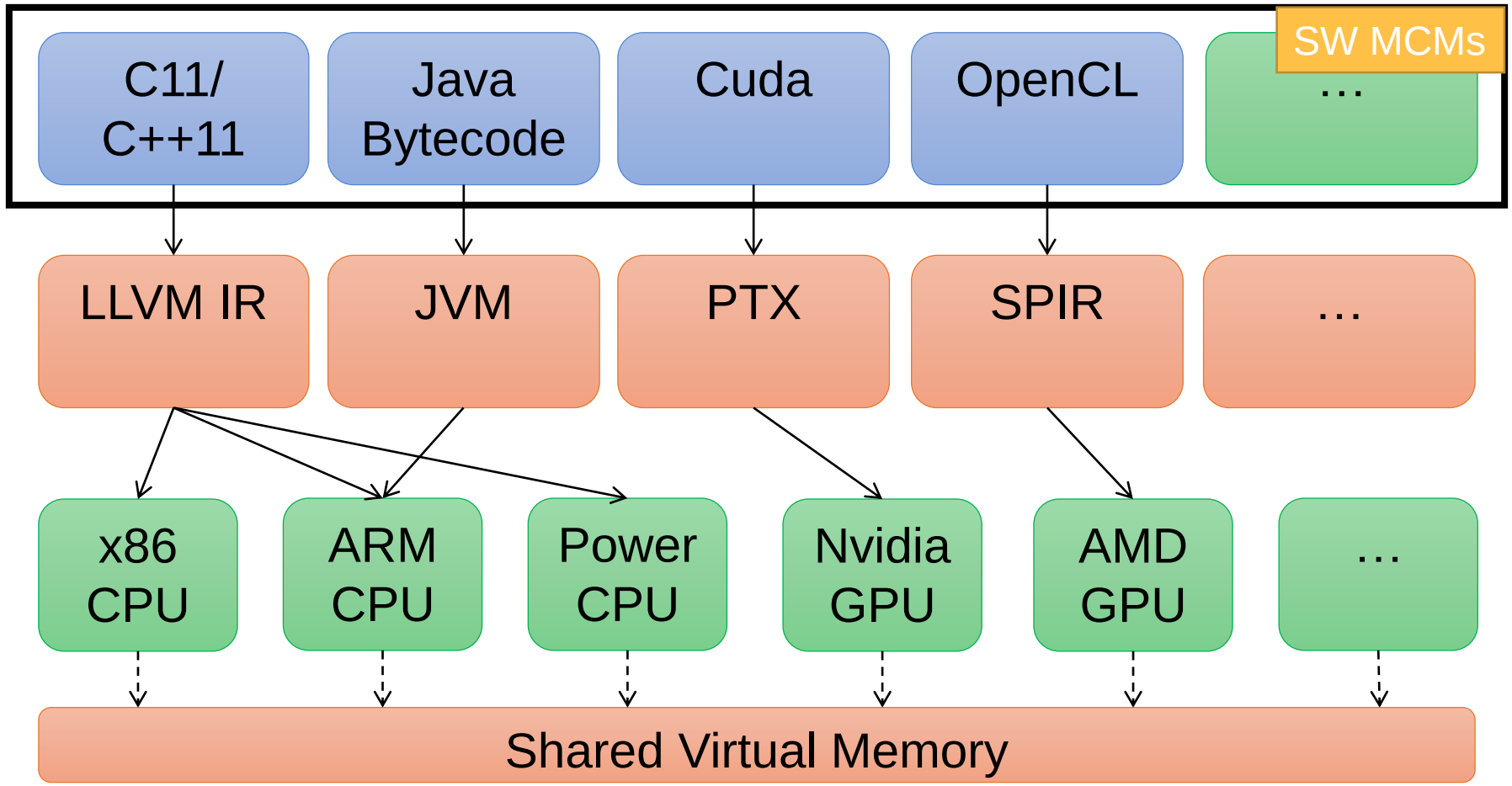
Memory Consistency Models (MCMs)

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



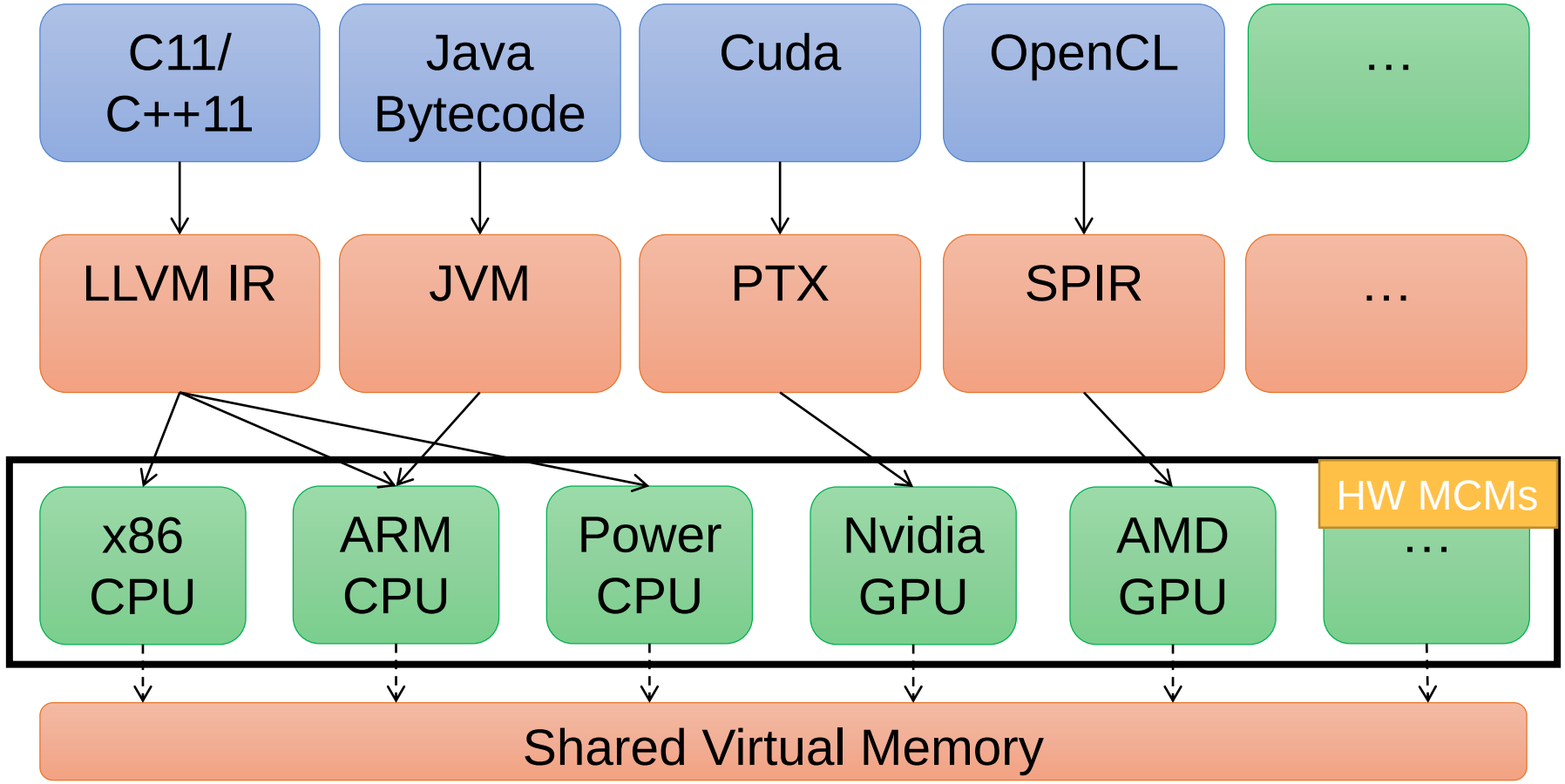
Memory Consistency Models (MCMs)

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



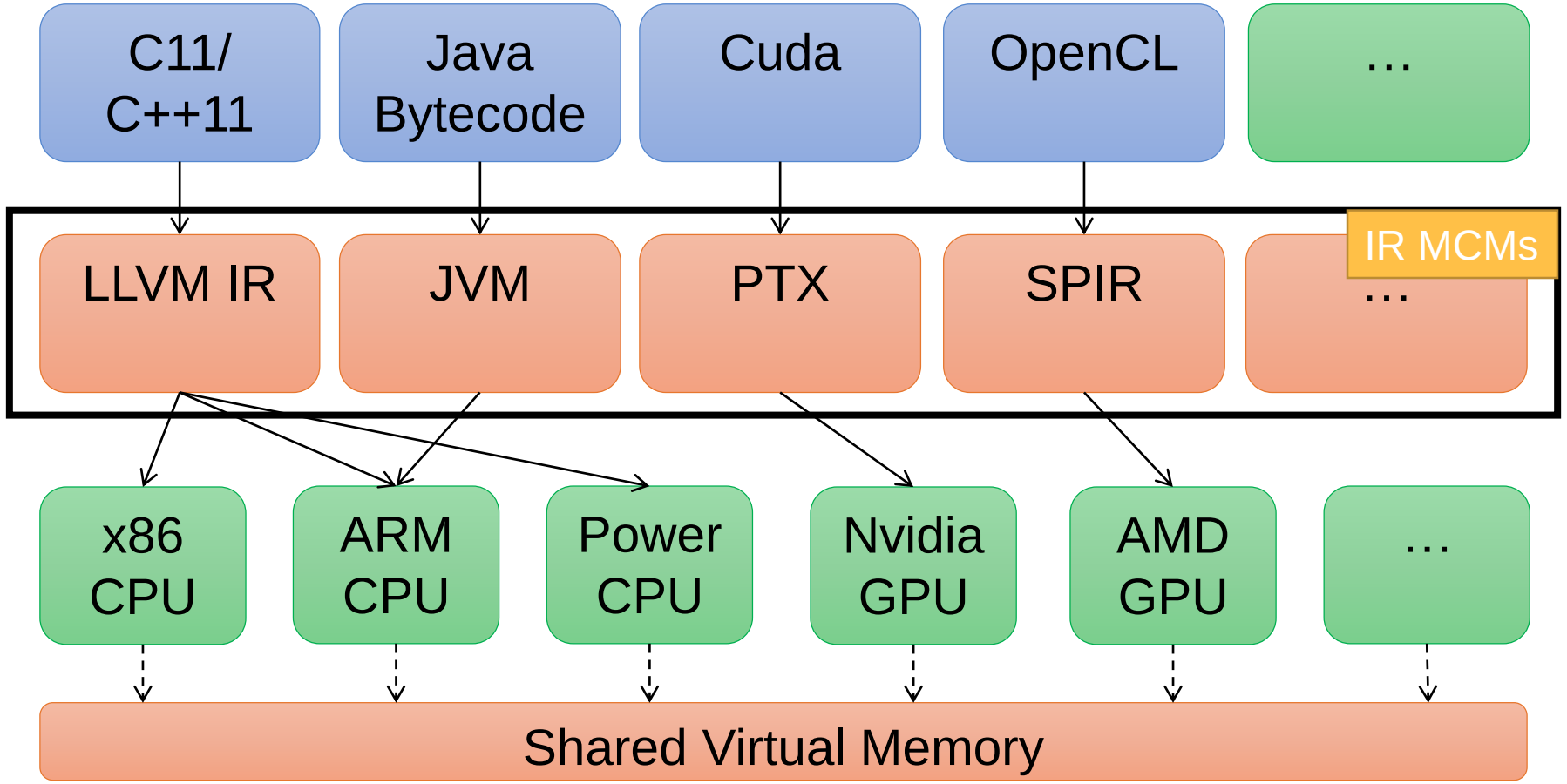
Memory Consistency Models (MCMs)

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



Memory Consistency Models (MCMs)

Memory Consistency Models (MCMs)
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



How are MCMs specified?

- Natural language?
  - E.g. Sequential Consistency [Lamport 1979]

"The result of any execution is the same as if the **operations of all the processors were executed in some sequential order**, and the **operations of each individual processor appear** in this sequence **in the order specified by its program**."

- What about more complicated models?

How are MCMs specified?

- Excerpt from the ARMv8 manual (memory model section):

Architecturally well-formed

An architecturally well-formed execution must satisfy both of the following requirements:

Internal visibility requirement

For a read or a write RW_1 that appears in program order before a read or a write RW_2 to the same Location, the internal visibility requirement requires that exactly one of the following statements is true:

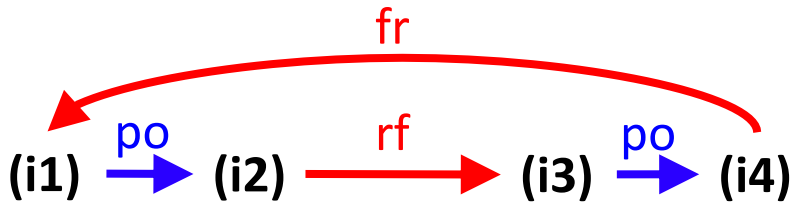
- RW_2 is a write W_2 that is Coherence-after RW_1.
- RW_1 is a write W_1 and RW_2 is a read R_2 such that either:
  - R_2 Reads-from W_1.
  - R_2 Reads-from another write that is Coherence-after W_1.
- RW_1 and RW_2 are both reads R_1 and R_2 such that R_1 Reads-from a write W_3 and either:
  - R_2 Reads-from W_3.
  - R_2 Reads-from another write that is Coherence-after W_3.

Note

If a Memory effect M_1 from an Observer appears in program order before a Memory effect M_2 from the same Observer, then M_1 will be seen to occur before M_2 by that Observer.

MCM Specifications Using Relations

- ISA-level MCMs defined using relational patterns [Shasha and Snir TOPLAS 1988]
- ISA-level executions are graphs
  - **nodes:** instructions, **edges:** ISA-level relations
- Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$



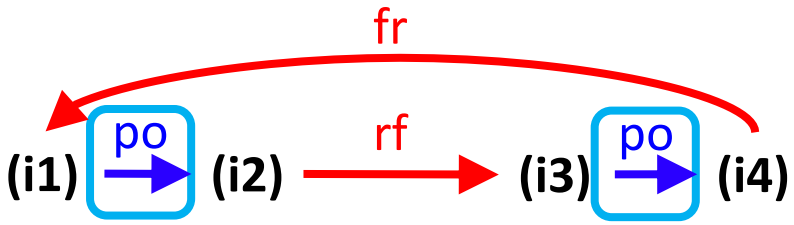
Message passing (mp) litmus test

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Legend:
po = Program order
co = Coherence order
rf = Reads-from
fr = From-reads

MCM Specifications Using Relations

- ISA-level MCMs defined using relational patterns [Shasha and Snir TOPLAS 1988]
- ISA-level executions are graphs
  - **nodes:** instructions, **edges:** ISA-level relations
- Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$



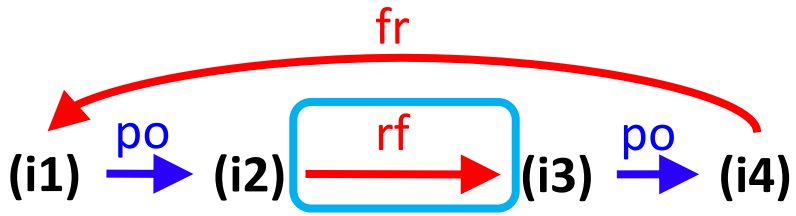
Message passing (mp) litmus test

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Legend:
po = Program order
co = Coherence order
rf = Reads-from
fr = From-reads

MCM Specifications Using Relations

- ISA-level MCMs defined using relational patterns [Shasha and Snir TOPLAS 1988]
- ISA-level executions are graphs
  - **nodes:** instructions, **edges:** ISA-level relations
- Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$



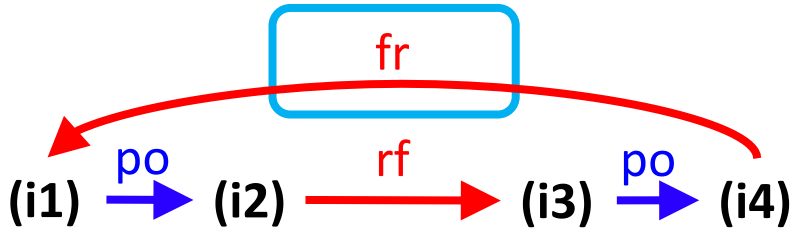
Message passing (mp) litmus test

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Legend:
po = Program order
co = Coherence order
rf = Reads-from
fr = From-reads

MCM Specifications Using Relations

- ISA-level MCMs defined using relational patterns [Shasha and Snir TOPLAS 1988]
- ISA-level executions are graphs
  - **nodes:** instructions, **edges:** ISA-level relations
- Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$



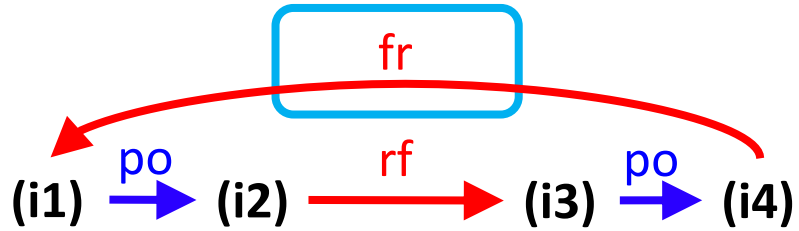
Message passing (mp) litmus test

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Legend:
po = Program order
co = Coherence order
rf = Reads-from
fr = From-reads

MCM Specifications Using Relations

- ISA-level MCMs defined using relational patterns [Shasha and Snir TOPLAS 1988]
- ISA-level executions are graphs
  - **nodes:** instructions, **edges:** ISA-level relations
- Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$



Message passing (mp) litmus test

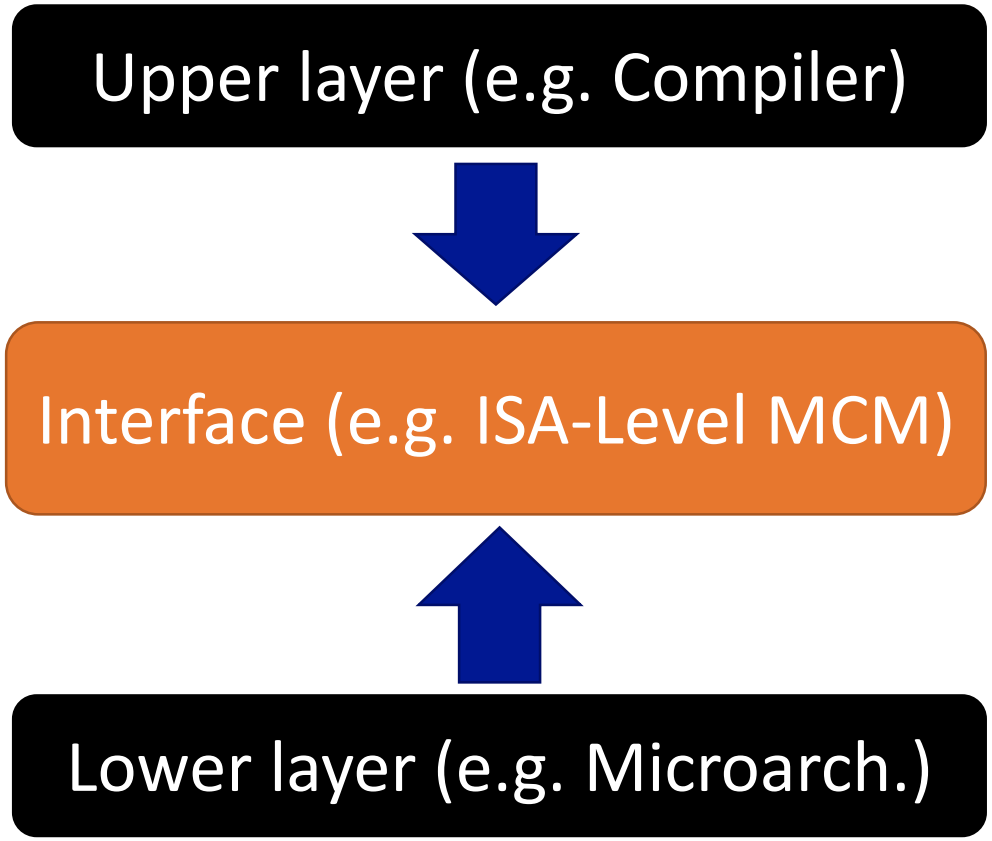
Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Legend:
po = Program order
co = Coherence order
rf = Reads-from
fr = From-reads

- Formal specifications of ISA + HLL MCMs
  - x86 [Owens et al. TPHOLS2009], ARM [Pulte et al. POPL2018], C11 [Batty et al. POPL 2011], ...
- Automated formal tools e.g. **herd** [Alglave et al. TOPLAS 2014]
  - Can formally analyse small test programs against these models

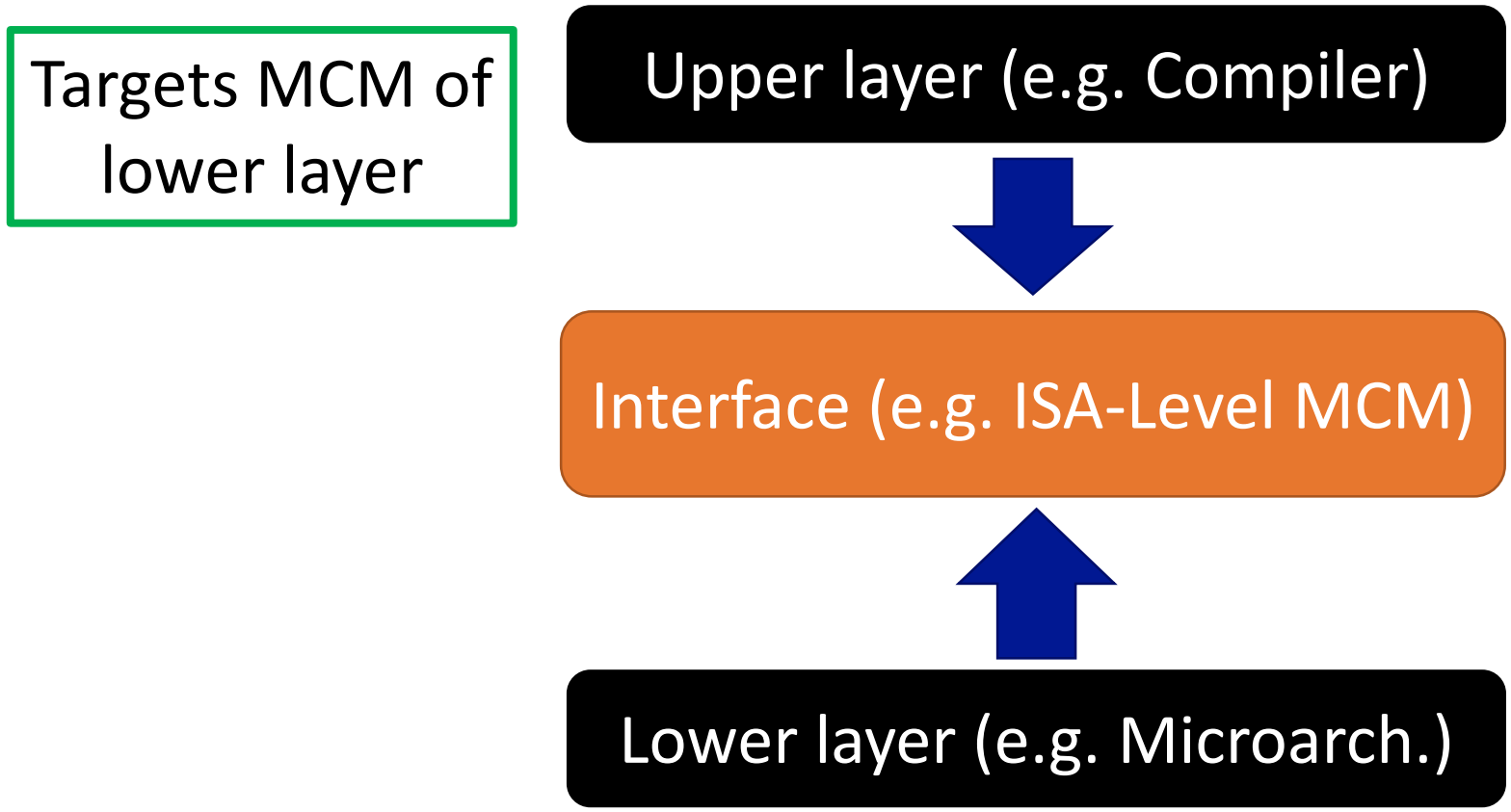
The Need for MCM Verification

- MCM specified at an interface between layers of the stack
- Upper layers target the MCM; lower layers must maintain it!



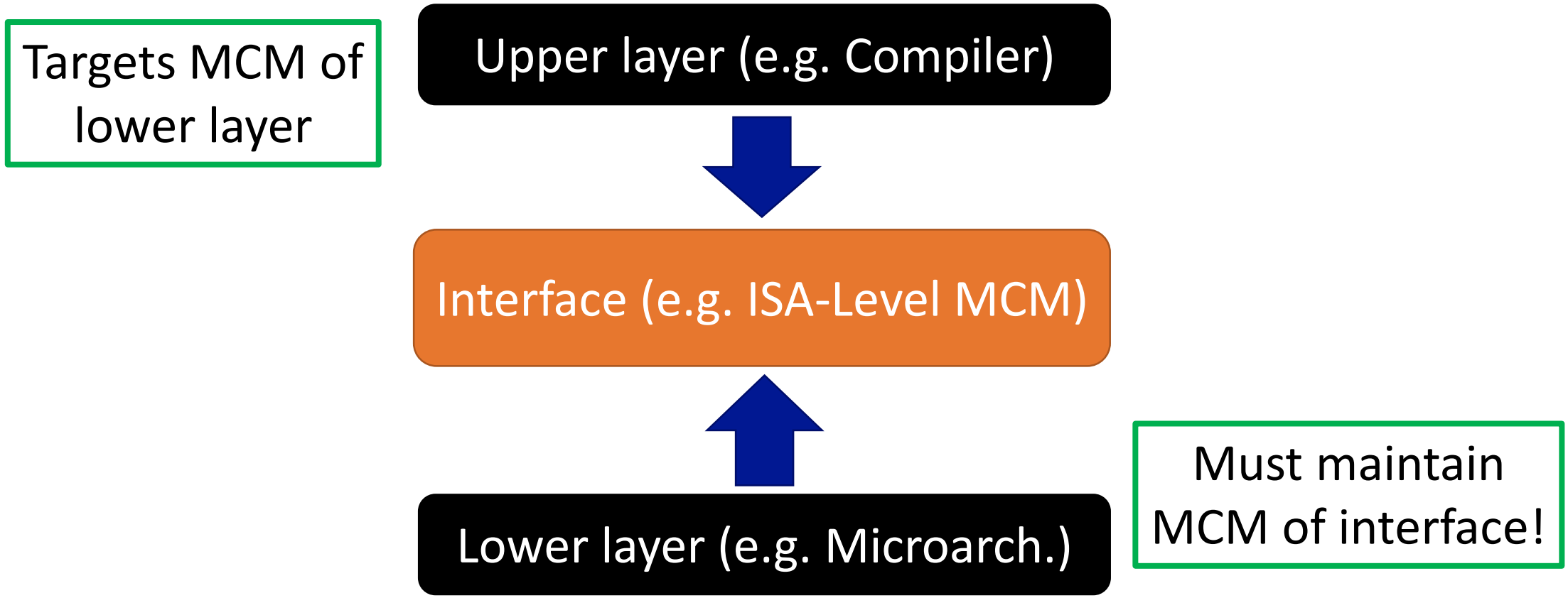
The Need for MCM Verification

- MCM specified at an interface between layers of the stack
- Upper layers target the MCM; lower layers must maintain it!



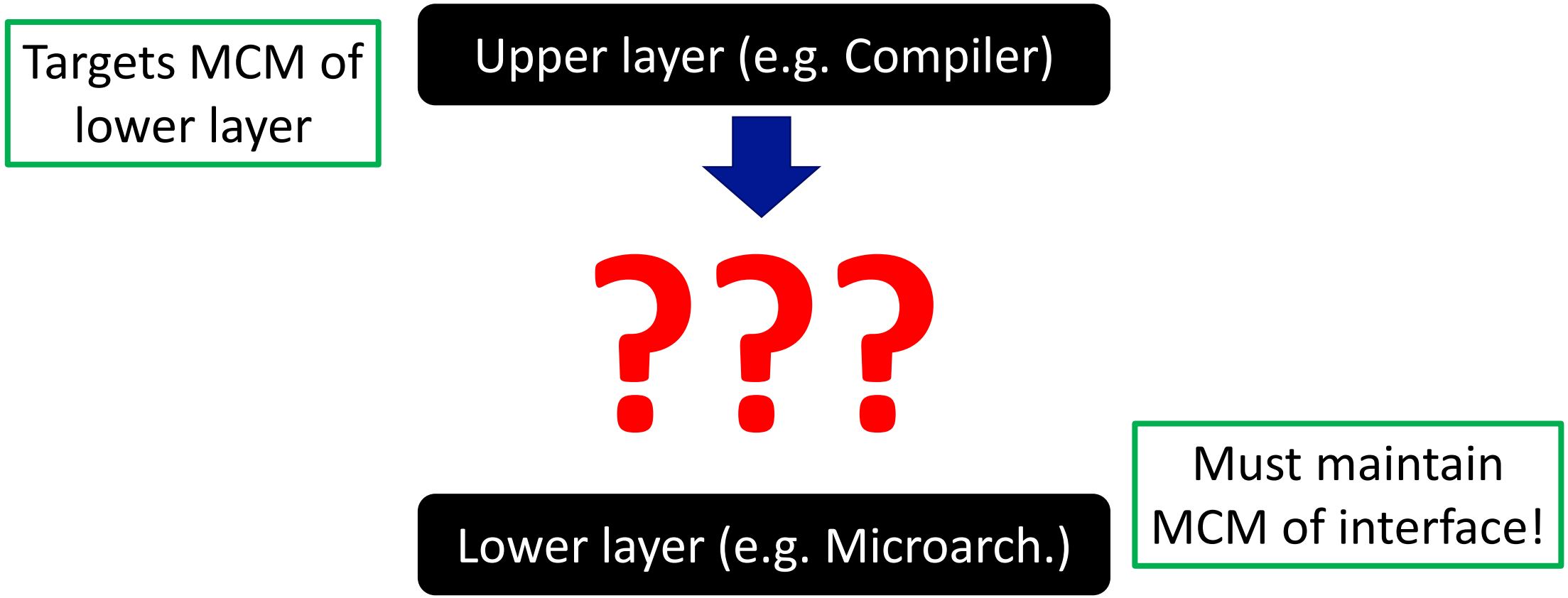
The Need for MCM Verification

- MCM specified at an interface between layers of the stack
- Upper layers target the MCM; lower layers must maintain it!

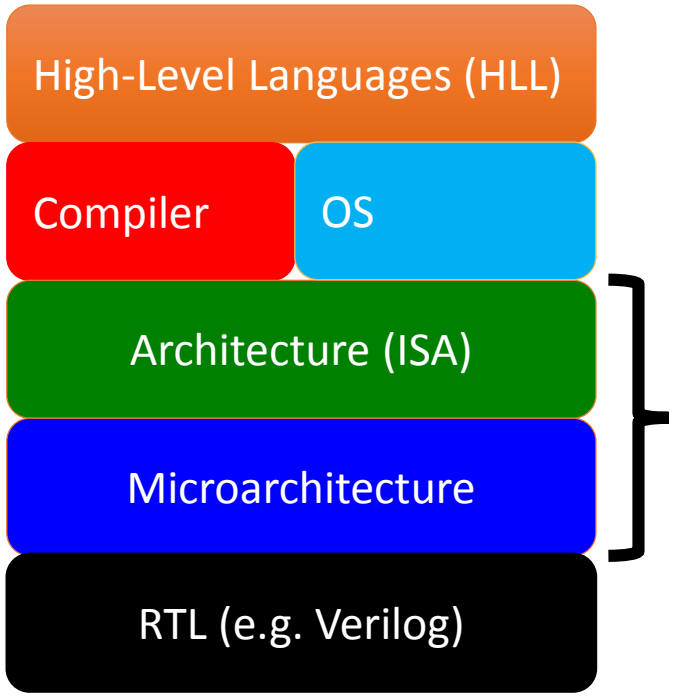


The Need for MCM Verification

- MCM specified at an interface between layers of the stack
- Upper layers target the MCM; lower layers must maintain it!



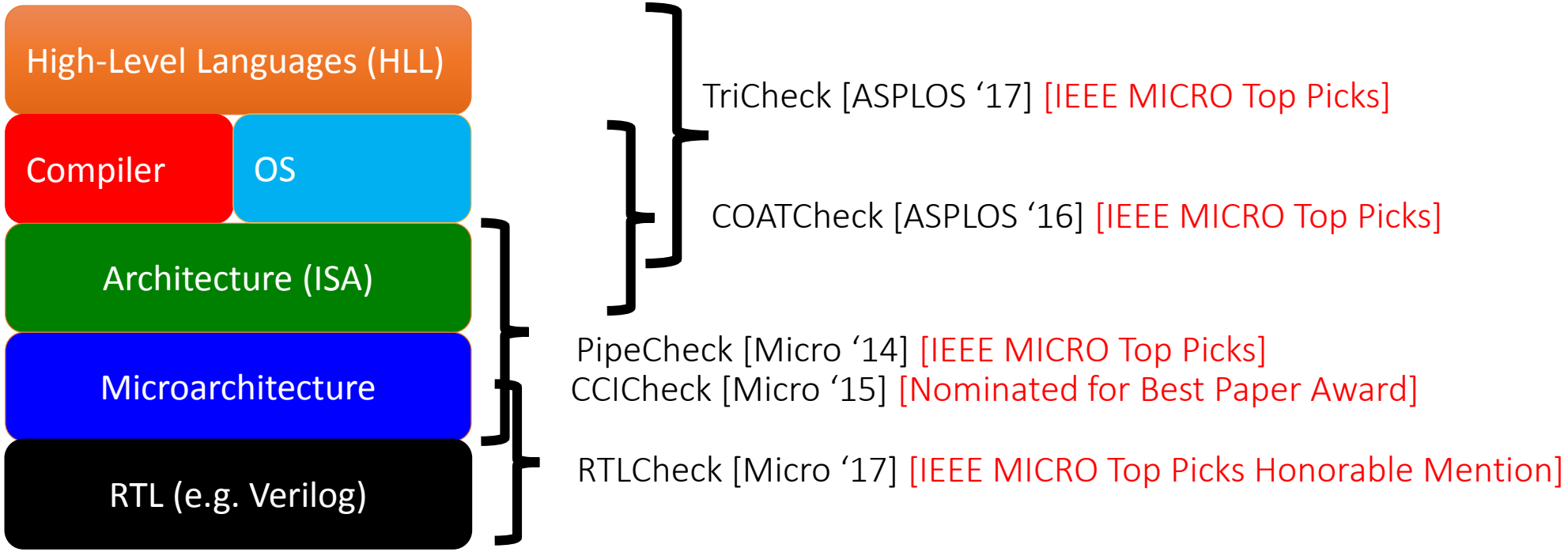
The Check Suite: Tools For Verifying Memory Orderings and their Security Implications



PipeCheck [Micro '14] [IEEE MICRO Top Picks]

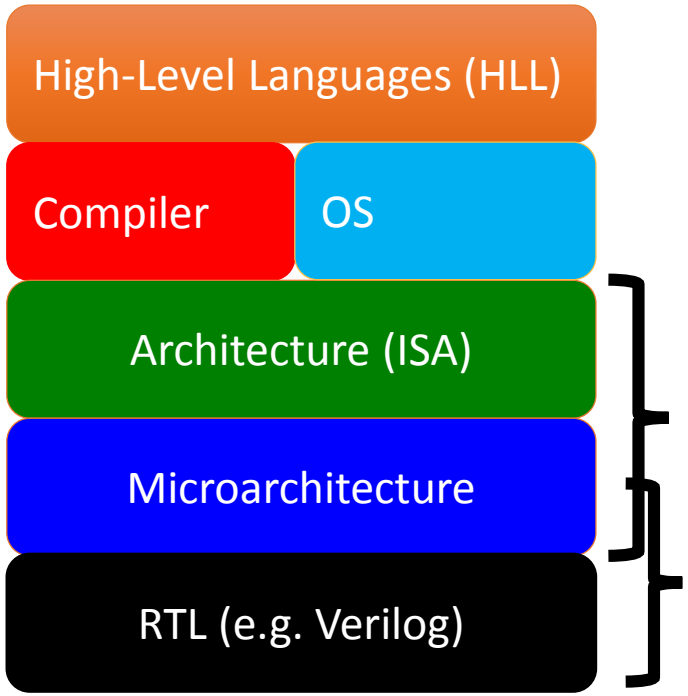
For more info: check.cs.Princeton.edu

The Check Suite: Tools For Verifying Memory Orderings and their Security Implications



For more info: check.cs.Princeton.edu

The Check Suite: Tools For Verifying Memory Orderings and their Security Implications

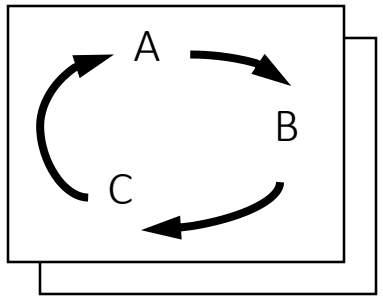


TriCheck [ASPLOS '17] [IEEE MICRO Top Picks]

COATCheck [ASPLOS '16] [IEEE MICRO Top Picks]

PipeCheck [Micro '14] [IEEE MICRO Top Picks]
CCICheck [Micro '15] [Nominated for Best Paper Award]

RTLCheck [Micro '17] [IEEE MICRO Top Picks Honorable Mention]

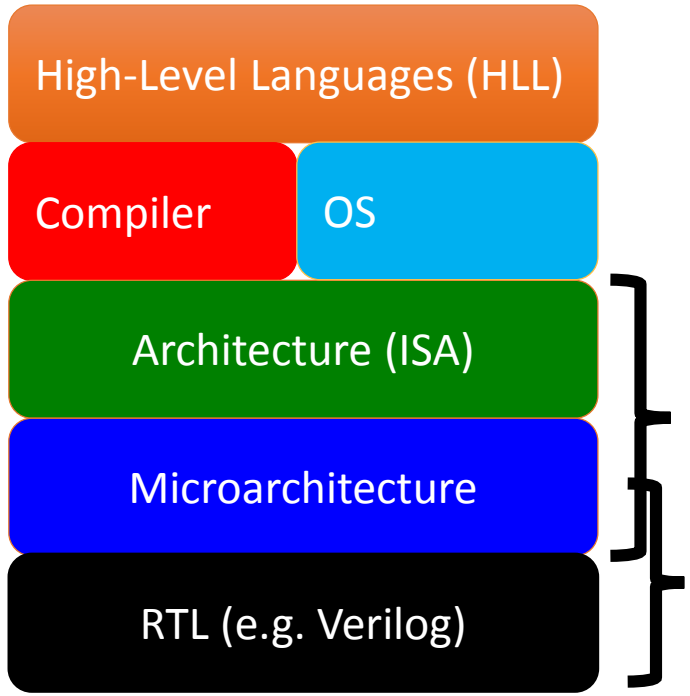


Our Approach

- Axiomatic specifications -> Happens-before graphs
- Check Happens-Before Graphs via Efficient SMT solvers
  - Cyclic => A->B->C->A… Can't happen
  - Acyclic => Scenario is observable

For more info: check.cs.Princeton.edu

The Check Suite: Tools For Verifying Memory Orderings and their Security Implications



CheckMate [Micro '18] [IEEE Micro Top Picks]

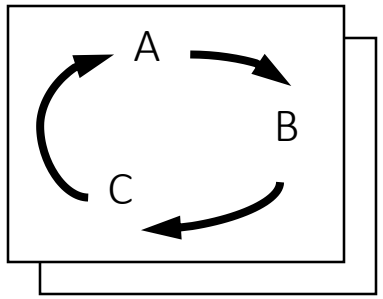
PipeProof [Micro '18] [Best Paper Nominee. IEEE Micro Top Picks Honorable Mention]

TriCheck [ASPLOS '17] [IEEE MICRO Top Picks]

COATCheck [ASPLOS '16] [IEEE MICRO Top Picks]

PipeCheck [Micro '14] [IEEE MICRO Top Picks]
CCICheck [Micro '15] [Nominated for Best Paper Award]

RTLCheck [Micro '17] [IEEE MICRO Top Picks Honorable Mention]

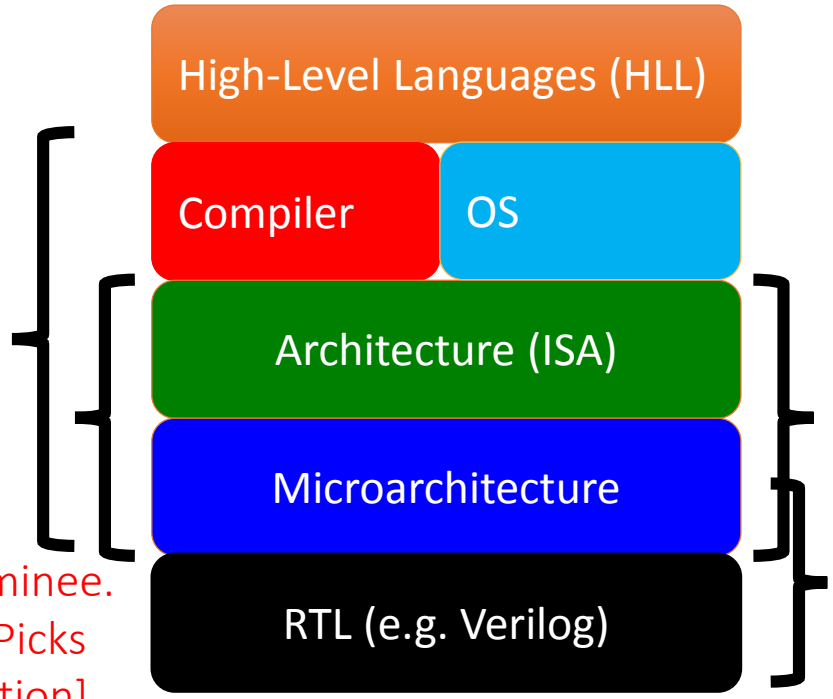


Our Approach

- Axiomatic specifications -> Happens-before graphs
- Check Happens-Before Graphs via Efficient SMT solvers
  - Cyclic => A->B->C->A… Can't happen
  - Acyclic => Scenario is observable

For more info: check.cs.Princeton.edu

The Check Suite: Tools For Verifying Memory Orderings and their Security Implications



CheckMate [Micro '18] [IEEE Micro Top Picks]

PipeProof [Micro '18] [Best Paper Nominee. IEEE Micro Top Picks Honorable Mention]

TriCheck [ASPLOS '17] [IEEE MICRO Top Picks]

COATCheck [ASPLOS '16] [IEEE MICRO Top Picks]

PipeCheck [Micro '14] [IEEE MICRO Top Picks]

CCICheck [Micro '15] [Nominated for Best Paper Award]

RTLCheck [Micro '17] [IEEE MICRO Top Picks Honorable Mention]

Our Approach

- Axiomatic specifications -> Happens-before graphs
- Check Happens-Before Graphs via Efficient SMT solvers
  - Cyclic => A->B->C->A… Can't happen
  - Acyclic => Scenario is observable

So far, tools have found bugs in:

- Widely-used Research simulator
- Cache coherence paper
- IBM XL C++ compiler (fixed in v13.1.5)
- In-design commercial processors
- RISC-V ISA specification
- Compiler mapping proofs
- C++ 11 mem model
- SpectrePrime, MeltdownPrime

For more

In a nutshell, our tool philosophy…

- Automate specification, verification, and translation related to MCMs
- Comprehensive exploration of ordering possibilities
- Key Techniques: Happens-before Graphs and SMT solvers
- Initially: Litmus-test driven (small test programs, 4-8 instrs)
- Now: PipeProof demonstrates *complete* (i.e. all-program) analysis

Outline

- Overview, Motivation, and MCM Background (15 minutes) (mm)
- PipeCheck: Verifying Microarchitectural Implementations against ISA Specs (45 minutes)
  - Includes hands-on of using uSpec DSL for specifying axioms (30 minutes) (ym)
- PipeProof: Beyond Litmus Tests (45 minutes) (ym)
  - Includes hands-on of proving simple microarch. across all programs (25 minutes)
- Coffee Break. 11-11:20
- Up and Down the Stack
  - RTLCheck (15 minutes) (ym)
  - TriCheck (10 minutes) (ct)
- Looking forward: Other uses of tools and techniques
  - CheckMate for security (25 minutes) (ct)
- Conclusions and Bigger Picture (10 minutes)