

# 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](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
<b>1</b> <code>x = 1;</code>	<b>3</b> <code>if (y == 1)</code> <code>print("Answer is:");</code>
<b>2</b> <code>y = 1;</code>	<b>4</b> <code>if (x == 1)</code> <code>print("42");</code>



# What does this program print?

Thread 0	Thread 1
<b>1</b> <code>x = 1;</code>	<b>3</b> <code>if (y == 1)</code> <code>print("Answer is:");</code>
<b>2</b> <code>y = 1;</code>	<b>4</b> <code>if (x == 1)</code> <code>print("42");</code>

Can it print “Answer is: 42”?



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

Can it print “Answer is: 42”? **Yes**, eg: ① ② ③ ④



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

Can it print “Answer is: 42”? **Yes**, eg: ① ② ③ ④

How about just “42”?



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

Can it print “Answer is: 42”? **Yes**, eg: ① ② ③ ④

How about just “42”? **Yes**, eg: ① ③ ④ ②





# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

Can it print “Answer is: 42”? **Yes**, eg: ① ② ③ ④

How about just “42”? **Yes**, eg: ① ③ ④ ②

Could it print **nothing**?



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

- Can it print “Answer is: 42”? **Yes**, eg: ① ② ③ ④
- How about just “42”? **Yes**, eg: ① ③ ④ ②
- Could it print nothing? **Yes**, eg: ③ ④ ① ②



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

- Can it print “Answer is: 42”? **Yes**, eg: ① ② ③ ④
- How about just “42”? **Yes**, eg: ① ③ ④ ②
- Could it print **nothing**? **Yes**, eg: ③ ④ ① ②

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
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

How about “Answer is:”?

② ① ③ ④



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

How about “Answer is:”?

**It depends!**

② ① ③ ④



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

How about “Answer is:”?

**It depends!**

② ① ③ ④



**NO!**



# What does this program print?

Thread 0	Thread 1
① <code>x = 1;</code>	③ <code>if (y == 1)</code> <code>print("Answer is:");</code>
② <code>y = 1;</code>	④ <code>if (x == 1)</code> <code>print("42");</code>

How about “Answer is:”?

It depends!

② ① ③ ④



**NO!**

**YES!**



**arm**



# What does this program print?

Thread 0

1 `y = 1;`

Thread 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?

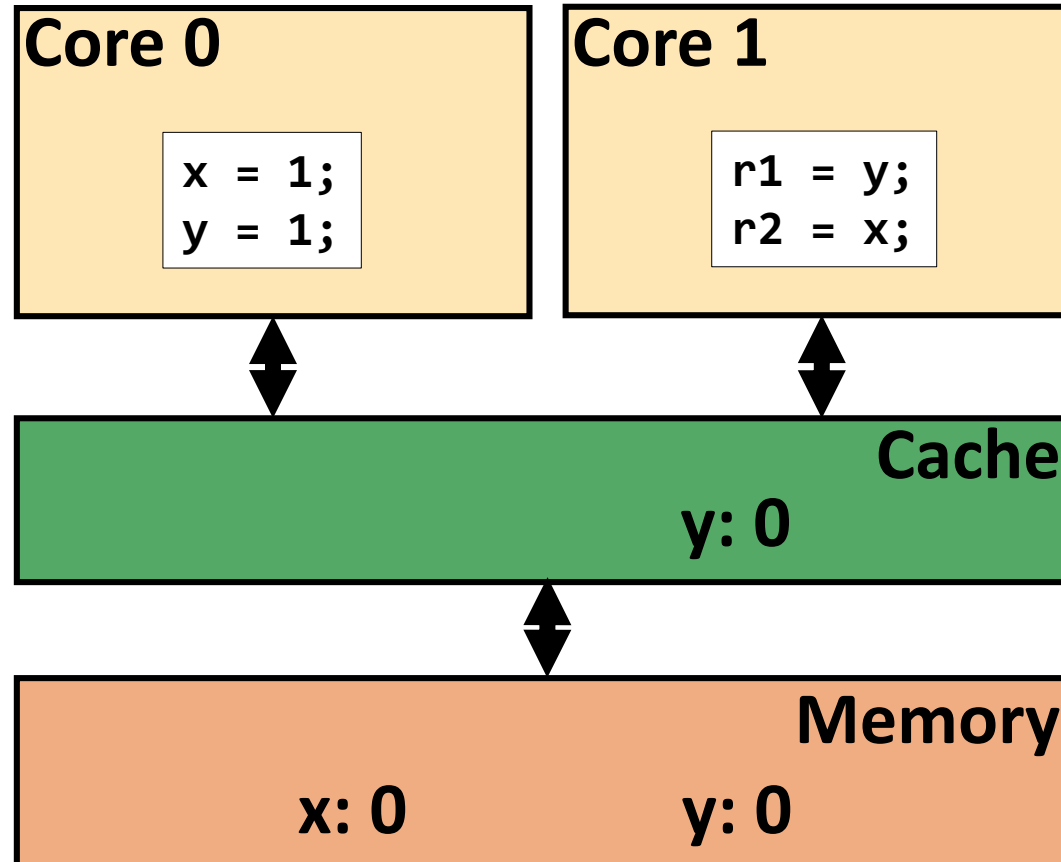
**We'll cover the answers today!**





# Why reorder memory operations?

**Answer: Performance!**



Message Passing (mp)

Core 0	Core 1
<code>x = 1;</code> <code>y = 1;</code>	<code>r1 = y;</code> <code>r2 = x;</code>
Can <code>r1=1</code> and <code>r2=0</code> ?	



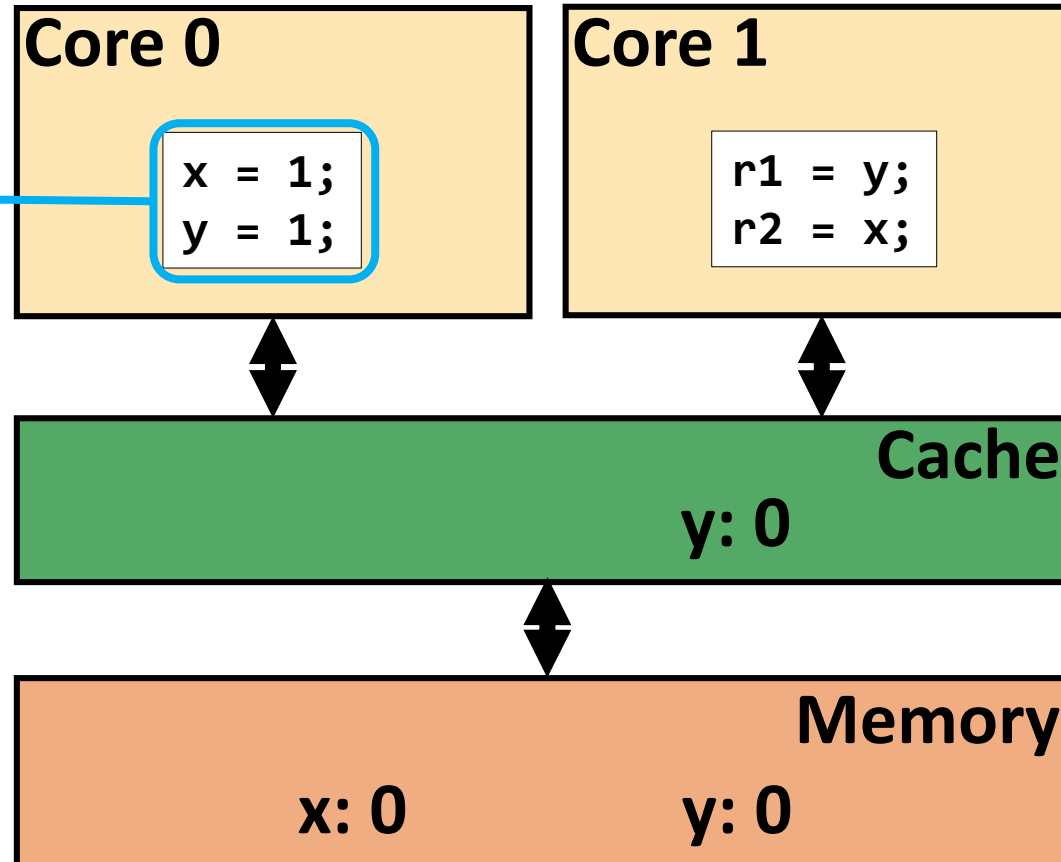
# Why reorder memory operations?

**Answer: Performance!**

Message Passing (mp)

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

Can improve performance by sending both stores to memory in parallel

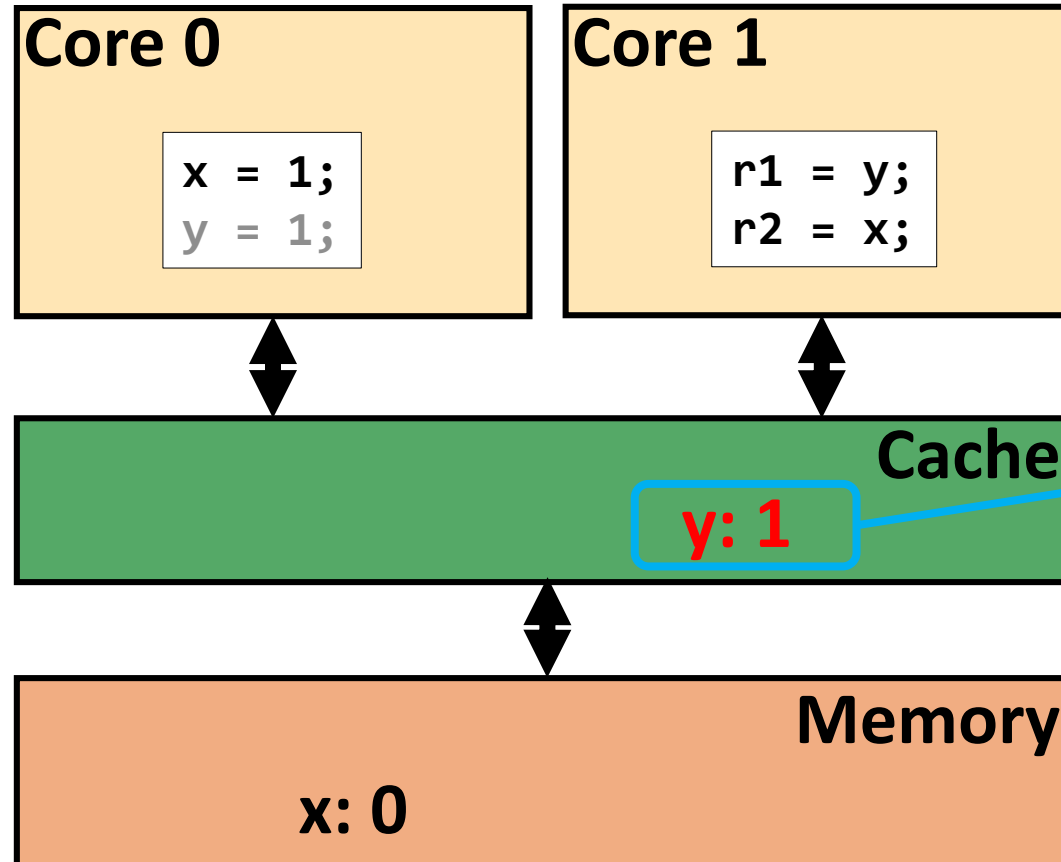


# Why reorder memory operations?

**Answer: Performance!**

Message Passing (mp)

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

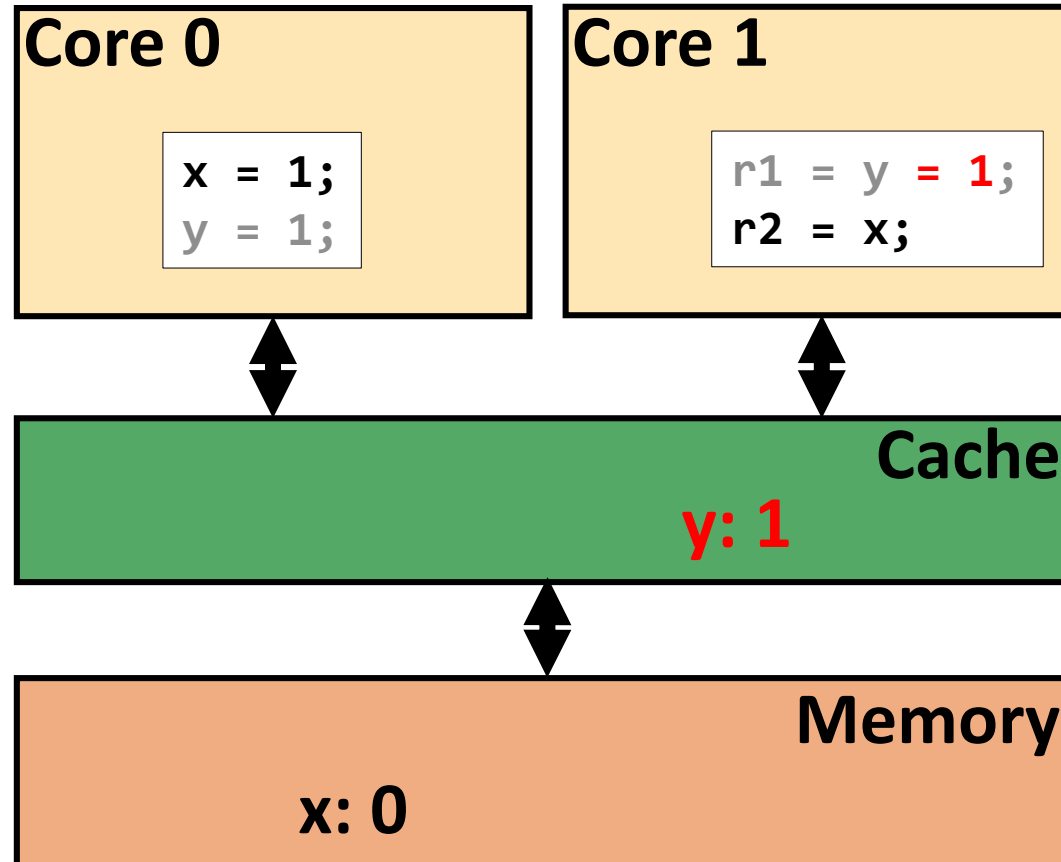


Store to y finishes quickly in cache



# Why reorder memory operations?

**Answer: Performance!**



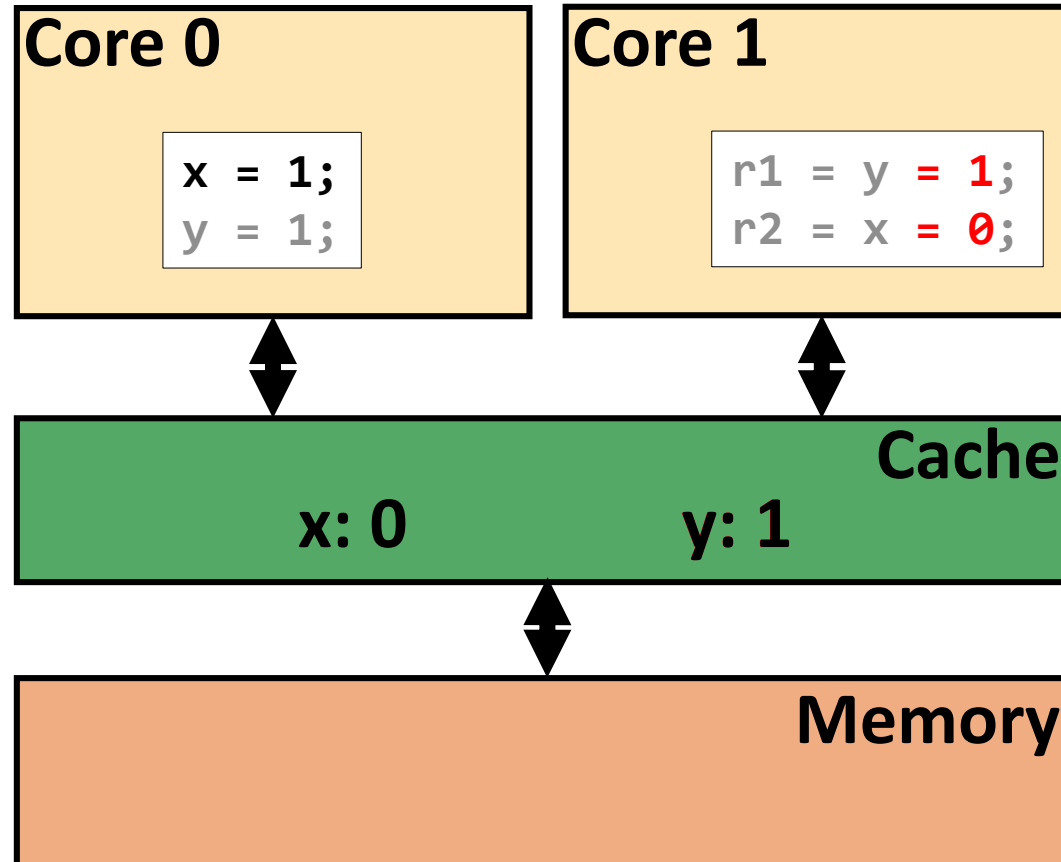
Message Passing (mp)

Core 0	Core 1
<code>x = 1;</code>	<code>r1 = y;</code>
<code>y = 1;</code>	<code>r2 = x;</code>
Can <code>r1=1</code> and <code>r2=0</code> ?	



# Why reorder memory operations?

**Answer: Performance!**



Message Passing (mp)

Core 0	Core 1
<code>x = 1;</code>	<code>r1 = y;</code>
<code>y = 1;</code>	<code>r2 = x;</code>
Can <code>r1=1</code> and <code>r2=0</code> ?	

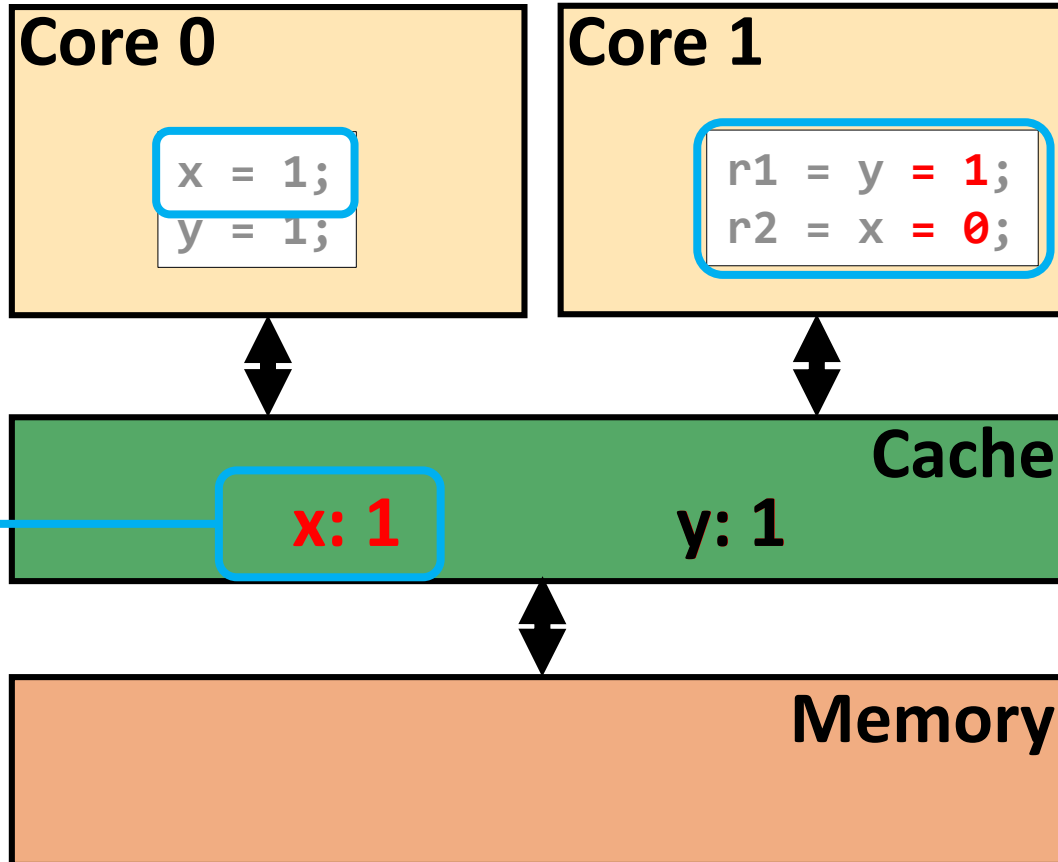


# Why reorder memory operations?

**Answer: Performance!**

Message Passing (mp)

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



By the time store of x is complete, Core 1 has observed reordering!



# Why reorder memory operations?

**Answer: Performance!**

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

**Core 0**

```
x = 1;  
FENCE  
y = 1;
```

**Core 1**

```
r1 = y = 1;  
r2 = x = 1;
```

**Cache**

**x: 1**

**y: 1**

**Memory**

Message Passing (mp)

Core 0	Core 1
x = 1; y = 1;	r1 = y; 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
① $x = 1;$	④ $r1 = y;$
② $y = 1;$	⑤ $r2 = x;$
③ $x = 2;$	
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 $x = 1;$	④ $r1 = y;$
② $y = 1;$	⑤ $r2 = x;$
③ $x = 2;$	
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
<b>2</b> $y = 1;$ <b>3</b> $x = 2;$	<b>4</b> $r1 = y;$ <b>5</b> $r2 = x;$
<b>Can <math>r1 = 1</math> and <math>r2 = 0</math>?</b>	



# 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
<b>2</b> $y = 1;$ <b>3</b> $x = 2;$	<b>4</b> $r1 = y;$ <b>5</b> $r2 = x;$
Can $r1 = 1$ and $r2 = 0$ ?	

Now **2** **4** **5** **3** 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<sup>1</sup>

<sup>1</sup>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<sup>1</sup>

<sup>1</sup>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<sup>1</sup>

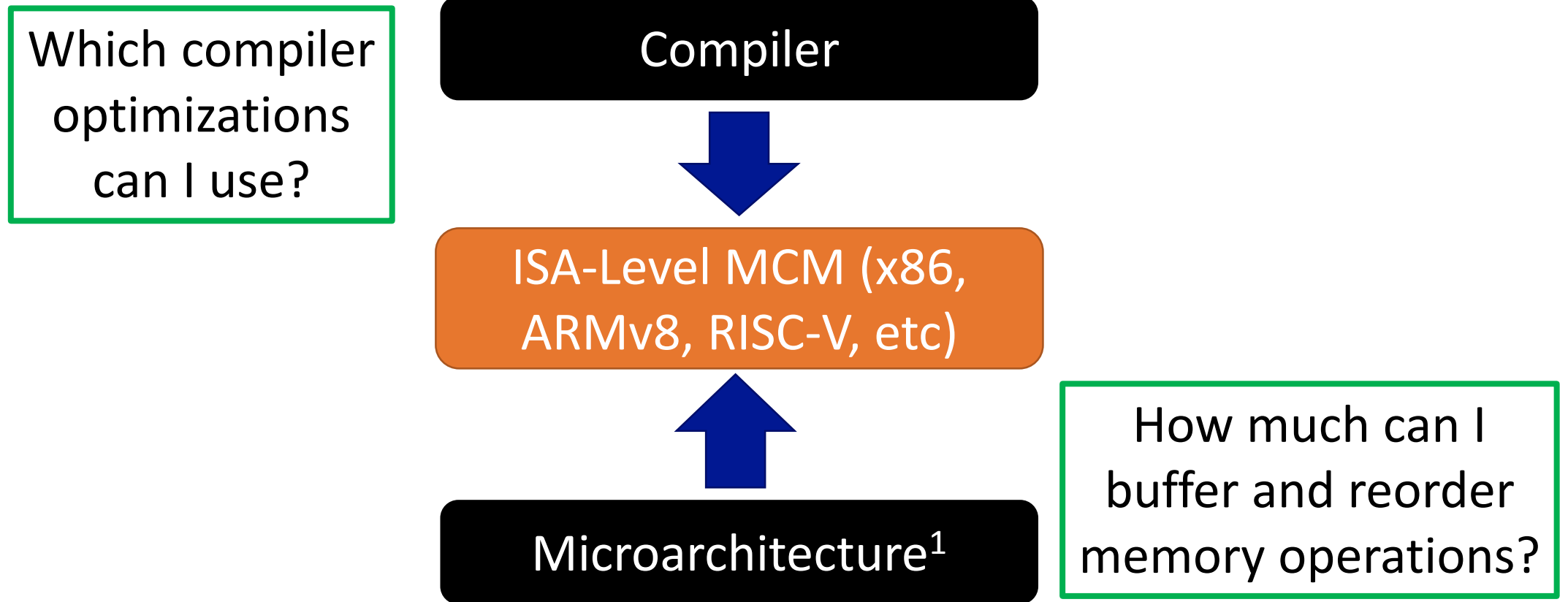
How much can I buffer and reorder memory operations?

<sup>1</sup>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



<sup>1</sup>Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

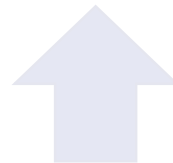


# Memory Consistency Models (MCMs)

- 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<sup>1</sup>

How much can I buffer and reorder memory operations?

<sup>1</sup>Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

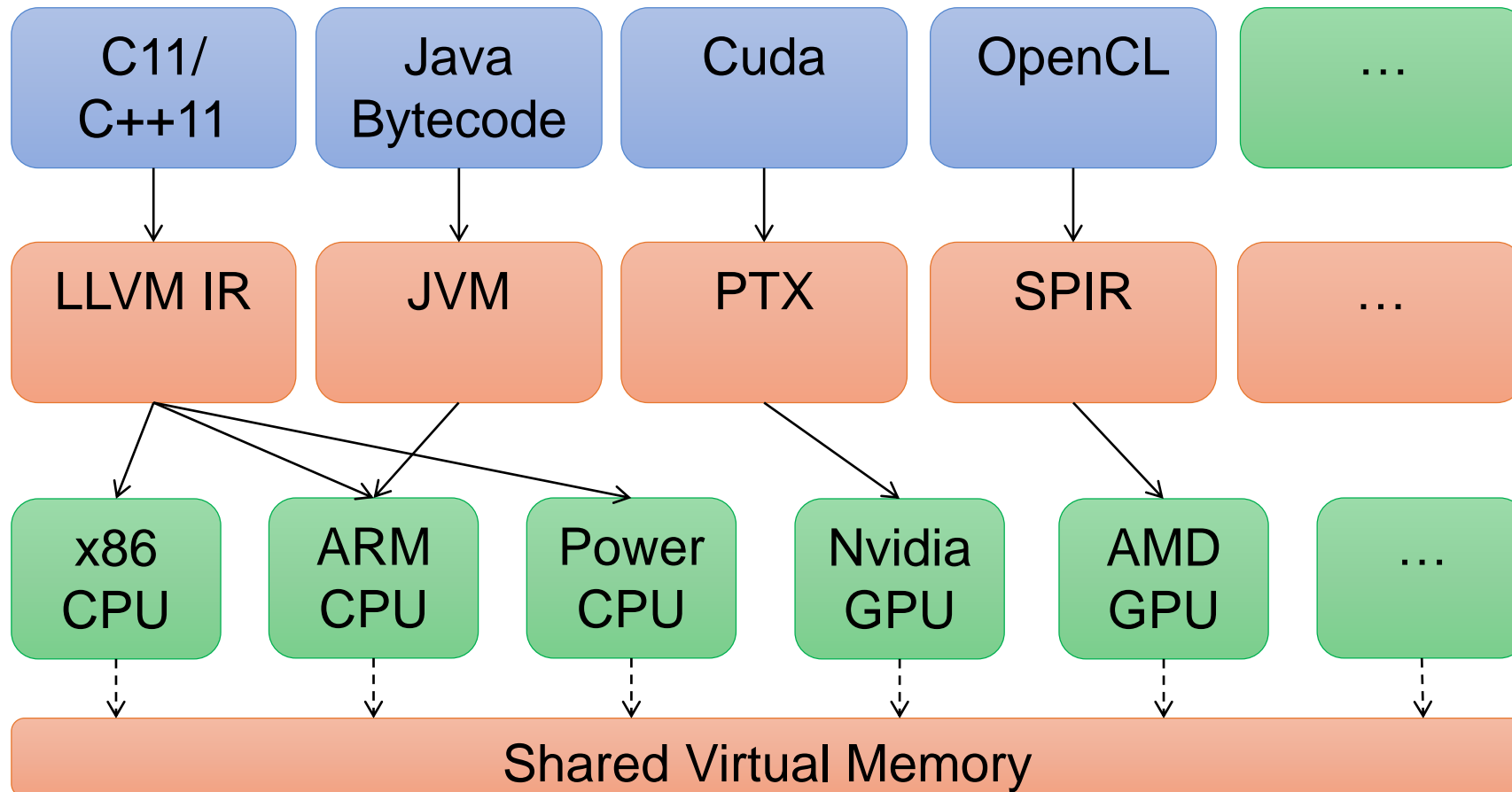




# Memory Consistency Models (MCMs)

## Memory Consistency Models (MCMs)

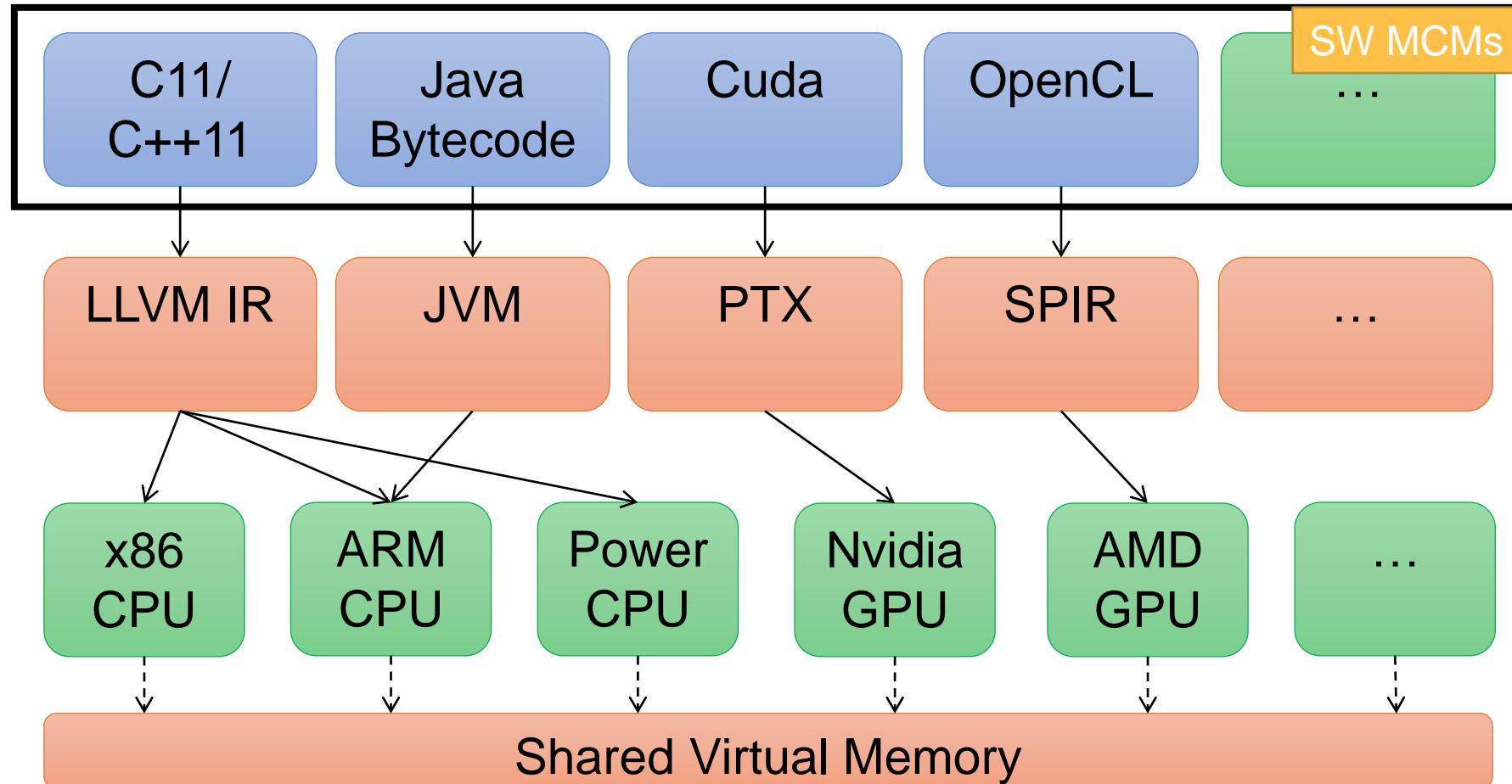
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



# Memory Consistency Models (MCMs)

## Memory Consistency Models (MCMs)

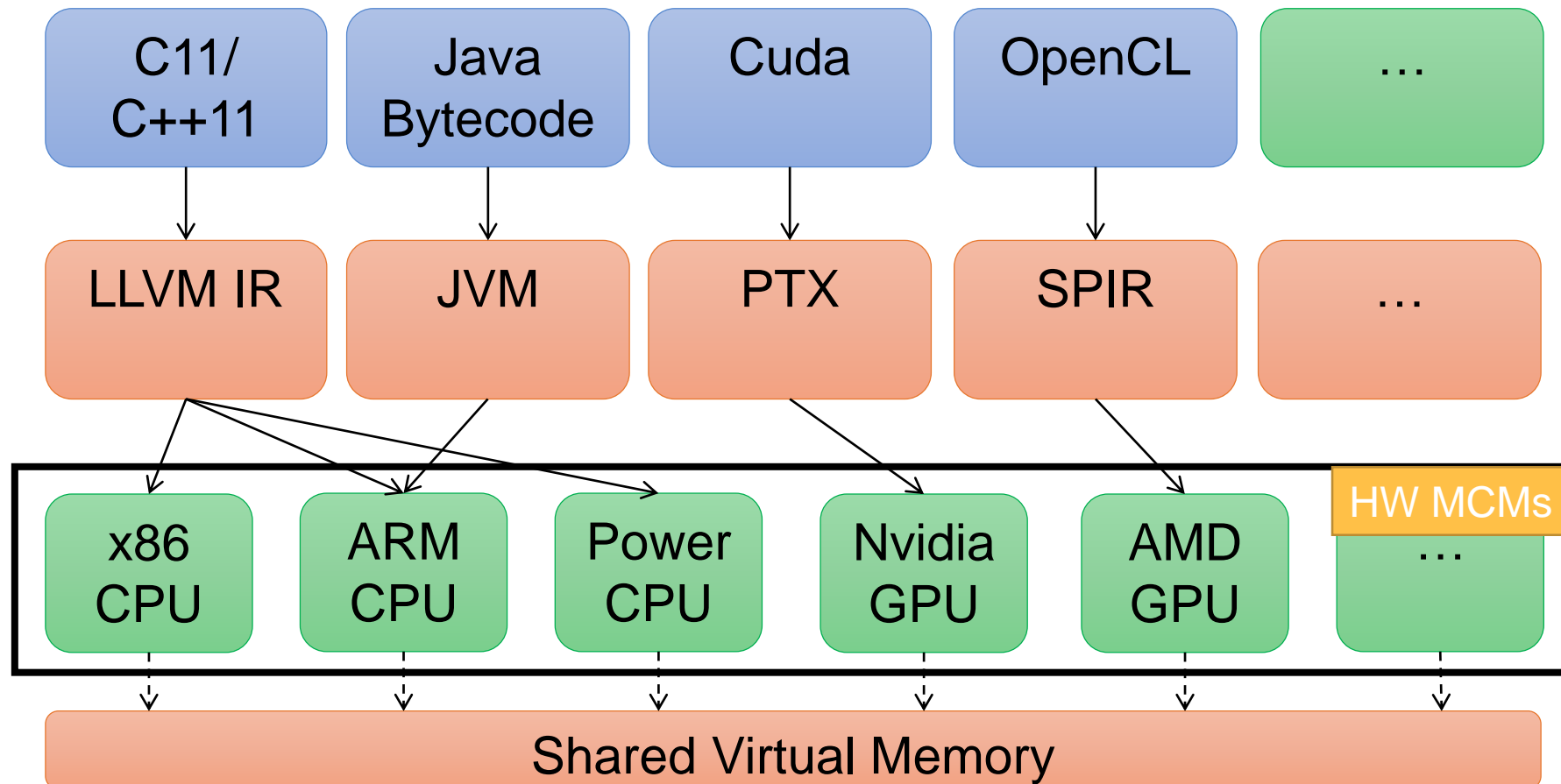
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



# Memory Consistency Models (MCMs)

## Memory Consistency Models (MCMs)

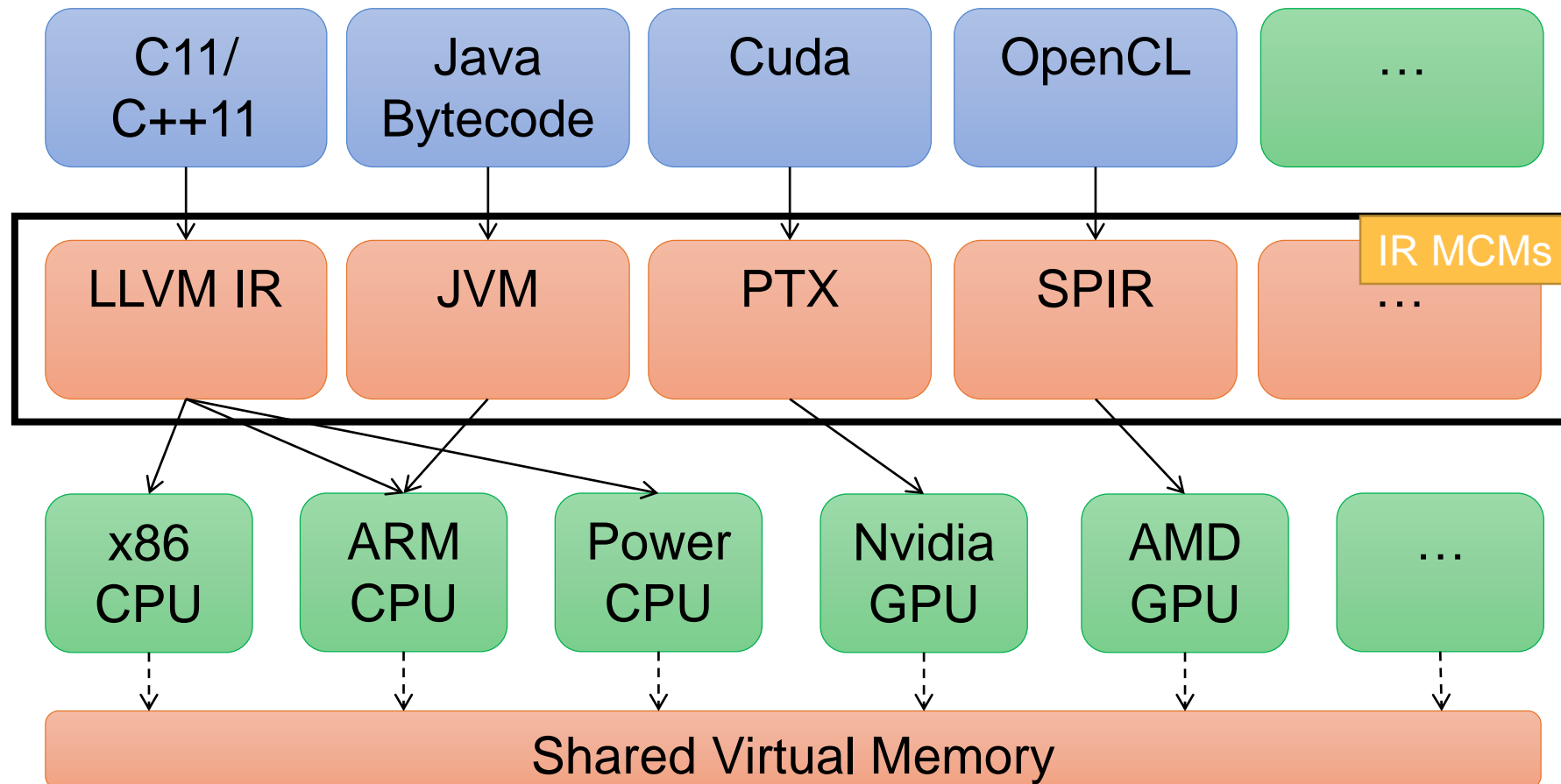
Specify rules and guarantees about the ordering and visibility of accesses to shared memory [Sorin et al., 2011].



# 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**  $W_3$ .

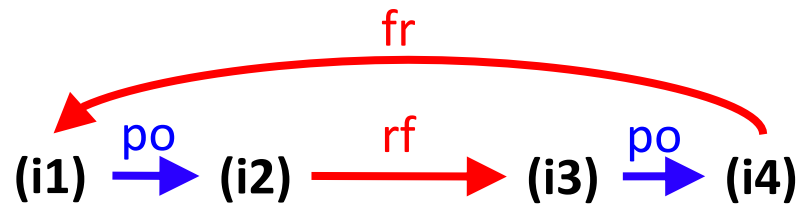
### Note

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;
<b>SC Forbids: r1 = 1, r2 = 0</b>	

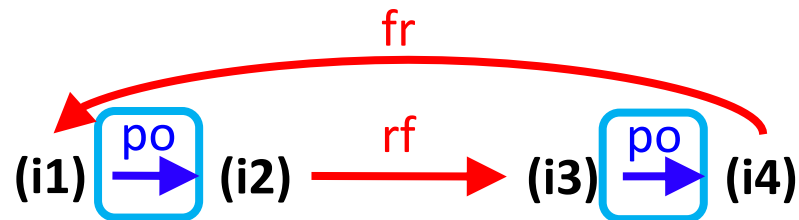
### Legend:

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;
<b>SC Forbids: r1 = 1, r2 = 0</b>	

### Legend:

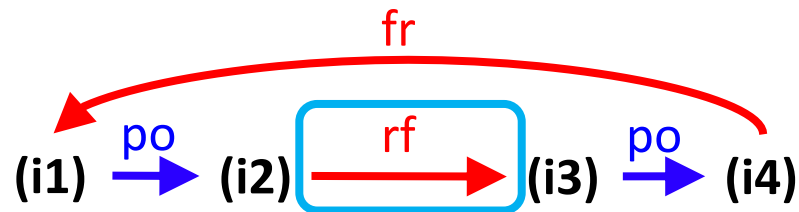
po = Program order  
co = Coherence order  
rf = Reads-from  
fr = From-reads





# 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;
<b>SC Forbids: r1 = 1, r2 = 0</b>	

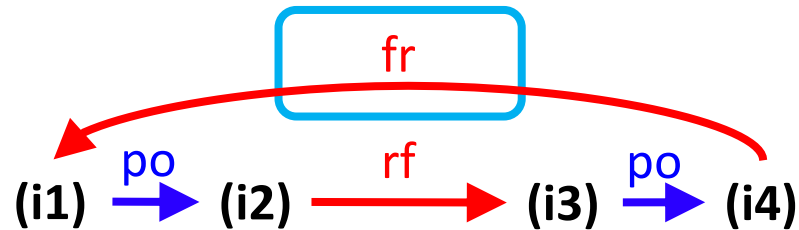
### Legend:

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;	(i3) r1 = y;
(i2) y = 1;	(i4) r2 = x;
<b>SC Forbids: r1 = 1, r2 = 0</b>	

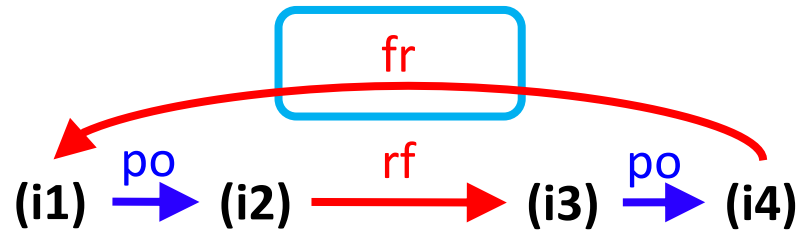
### Legend:

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  $acyclic(po \cup co \cup rf \cup fr)$



- 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

## Message passing (mp) litmus test

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

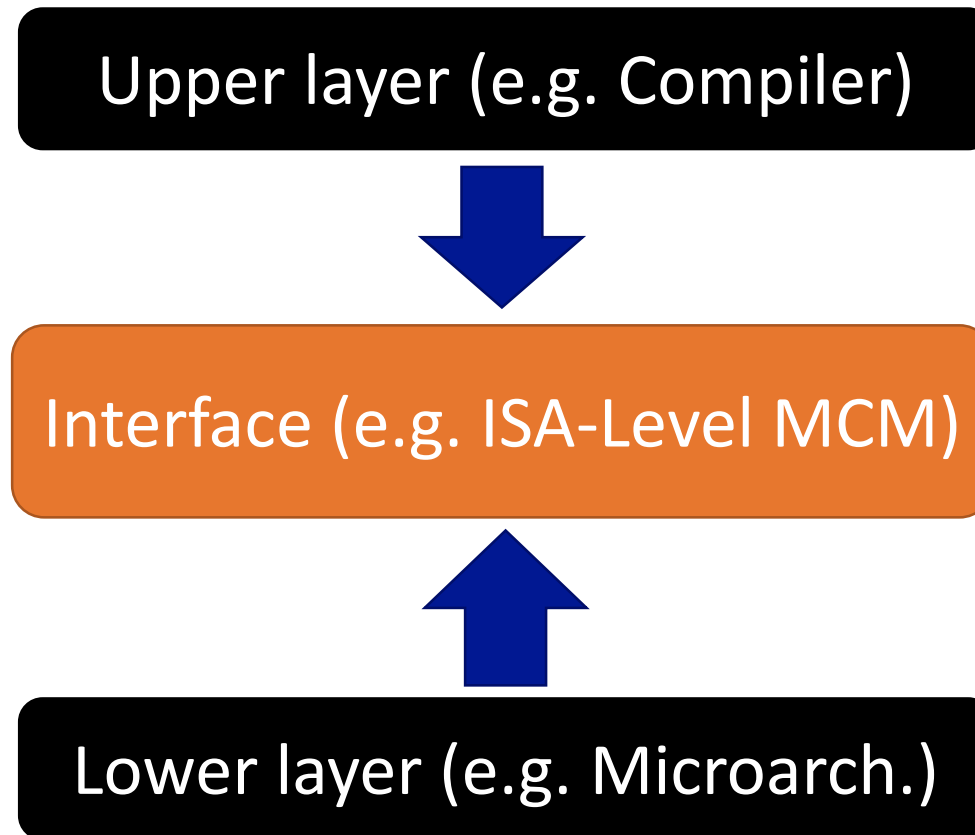
### Legend:

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



# 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!

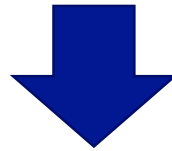


# The Need for MCM Verification

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

Targets MCM of  
lower layer

Upper layer (e.g. Compiler)



Interface (e.g. ISA-Level MCM)

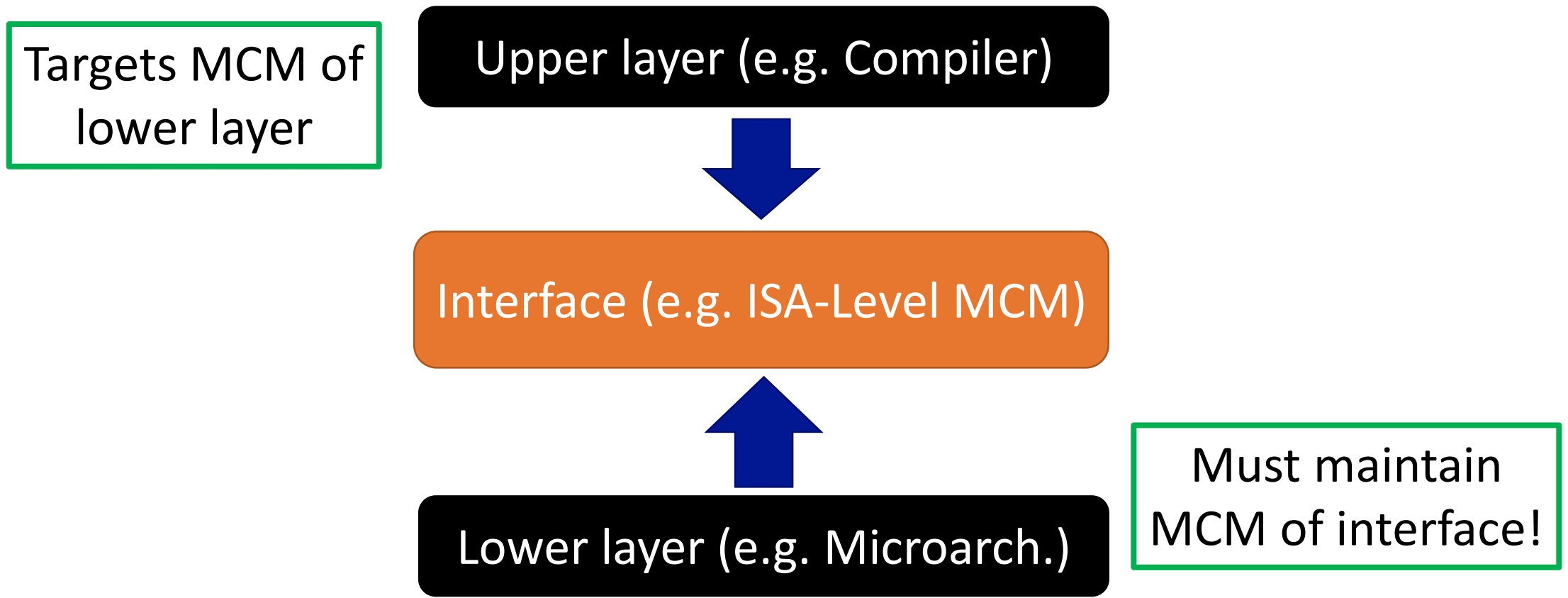


Lower layer (e.g. Microarch.)



# 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!

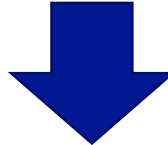


# The Need for MCM Verification

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

Targets MCM of lower layer

Upper layer (e.g. Compiler)

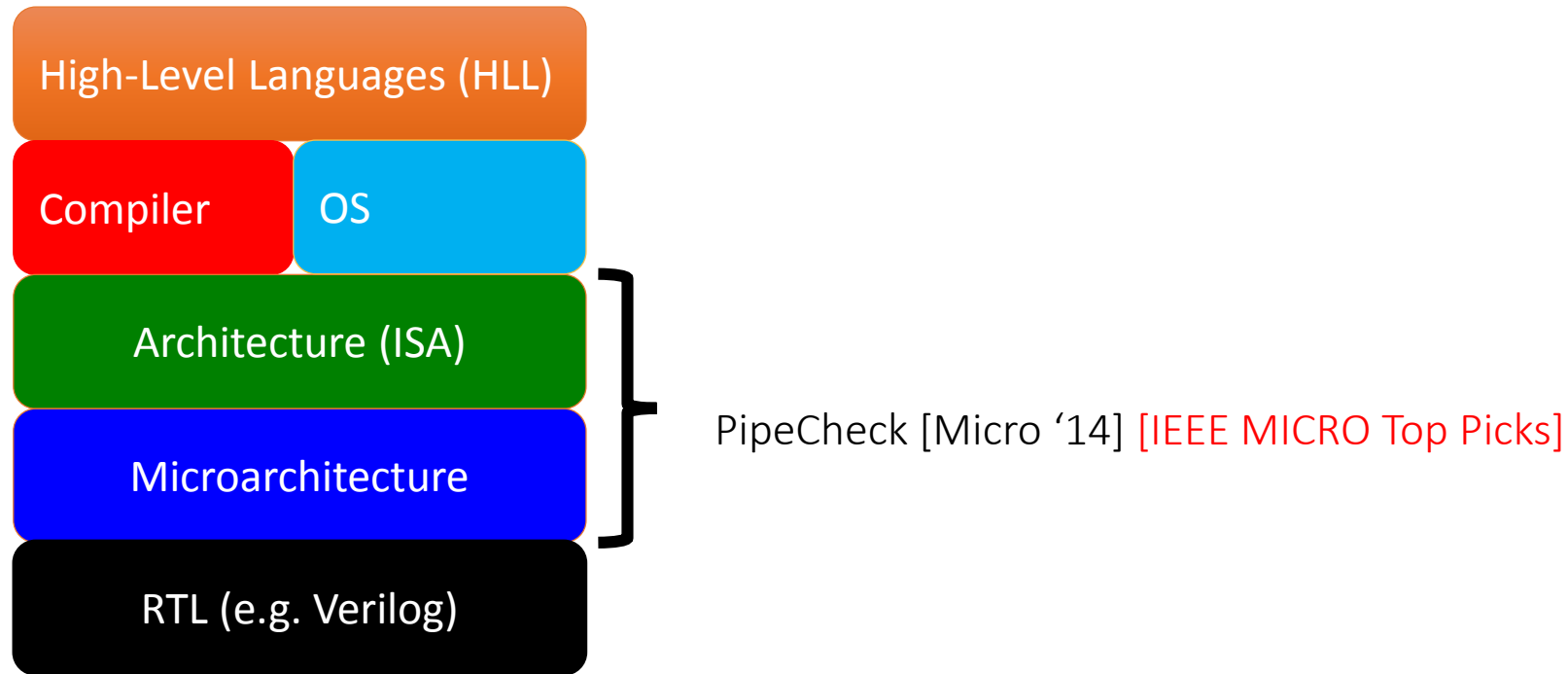


Lower layer (e.g. Microarch.)

Must maintain MCM of interface!

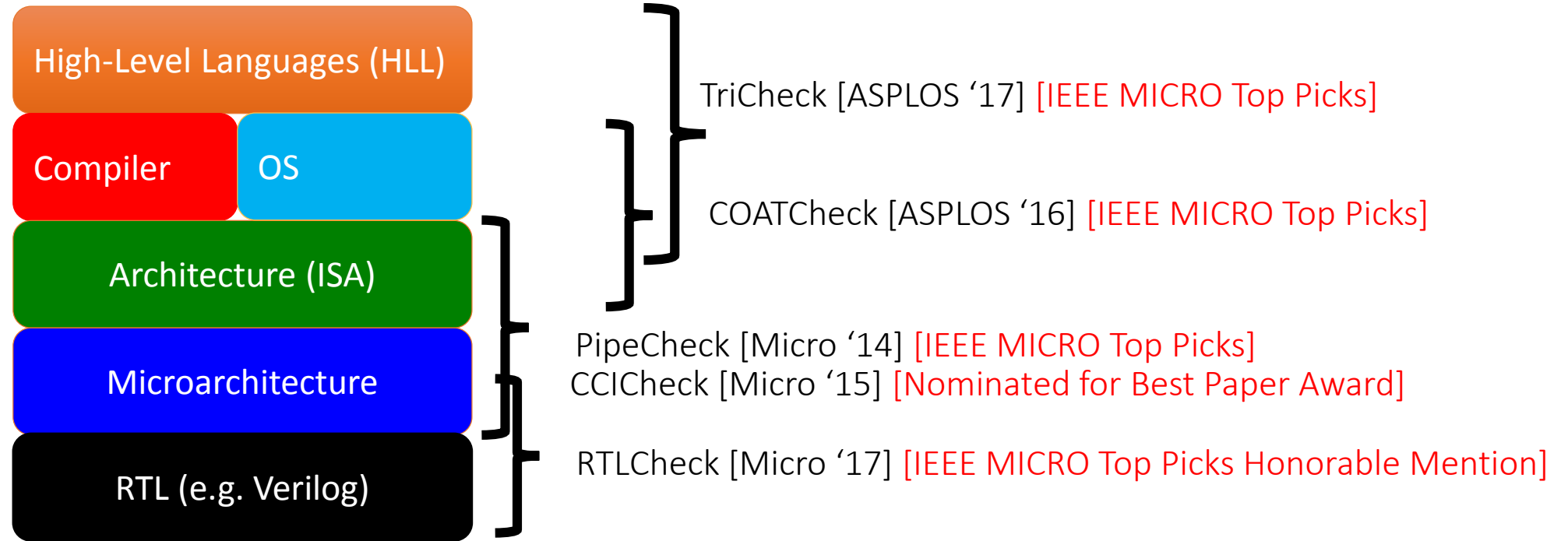


# The Check Suite: Tools For Verifying Memory Orderings and their Security Implications

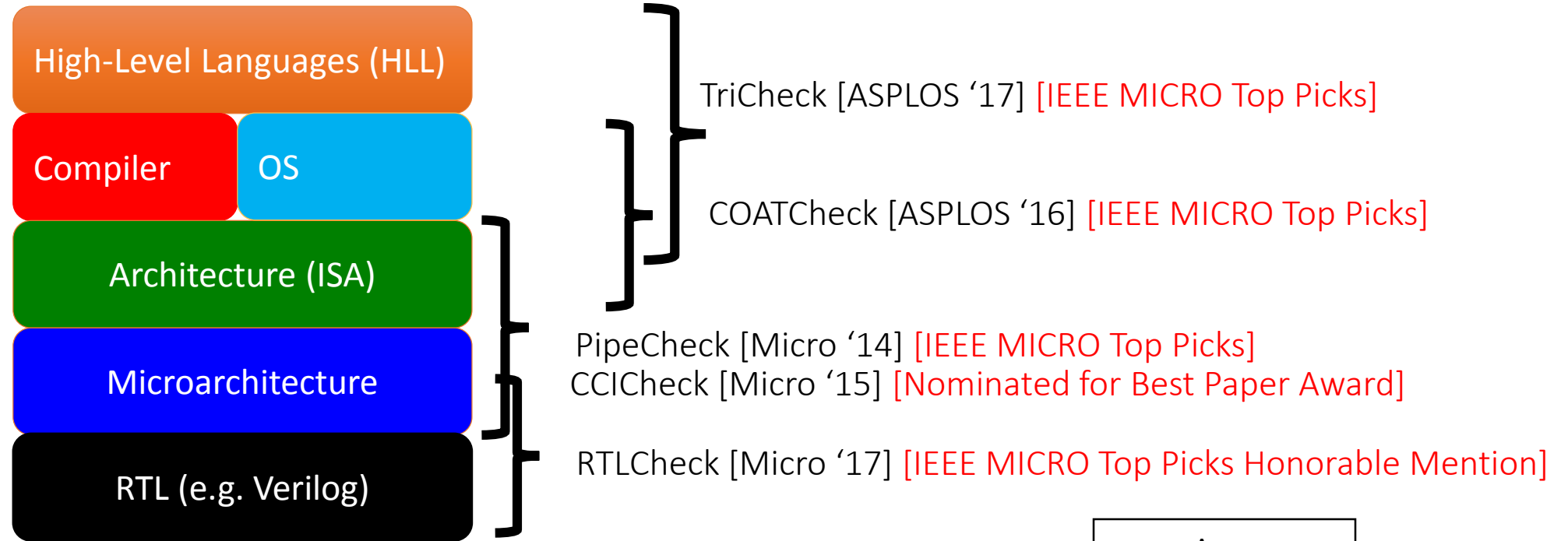




# The Check Suite: Tools For Verifying Memory Orderings and their Security Implications

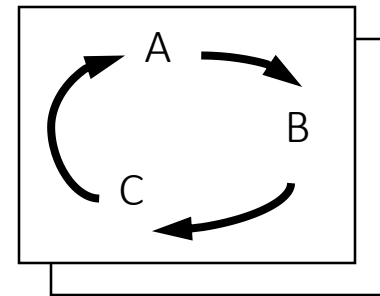


# The Check Suite: Tools For Verifying Memory Orderings and their Security Implications

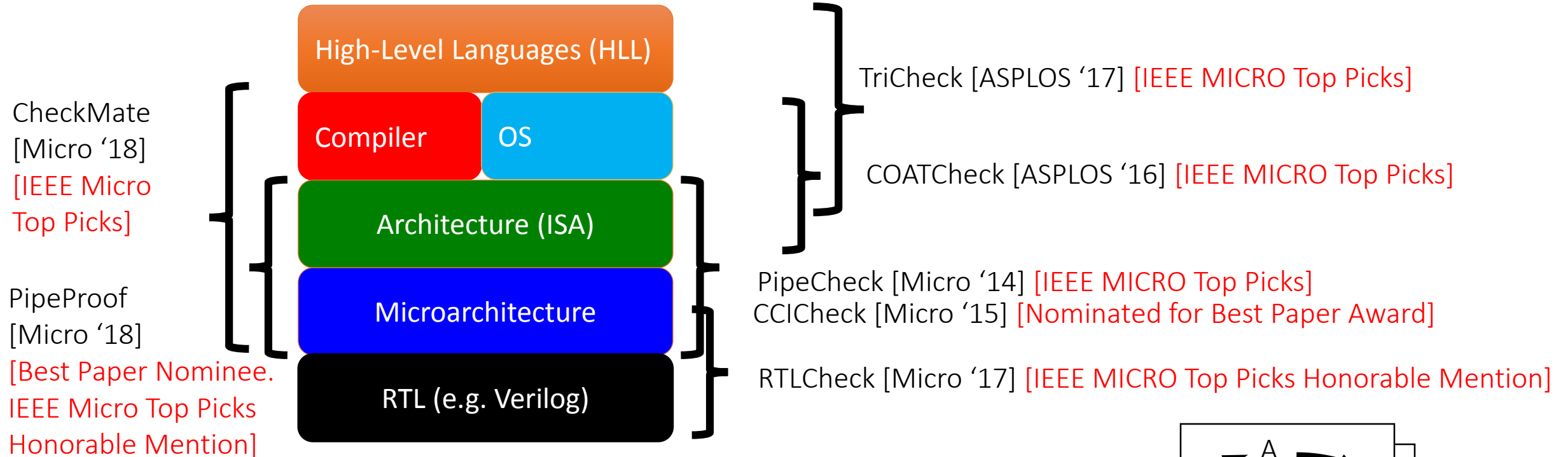


## Our Approach

- 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**

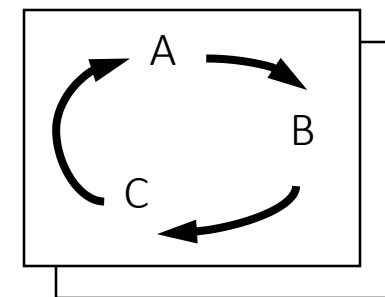


# The Check Suite: Tools For Verifying Memory Orderings and their Security Implications

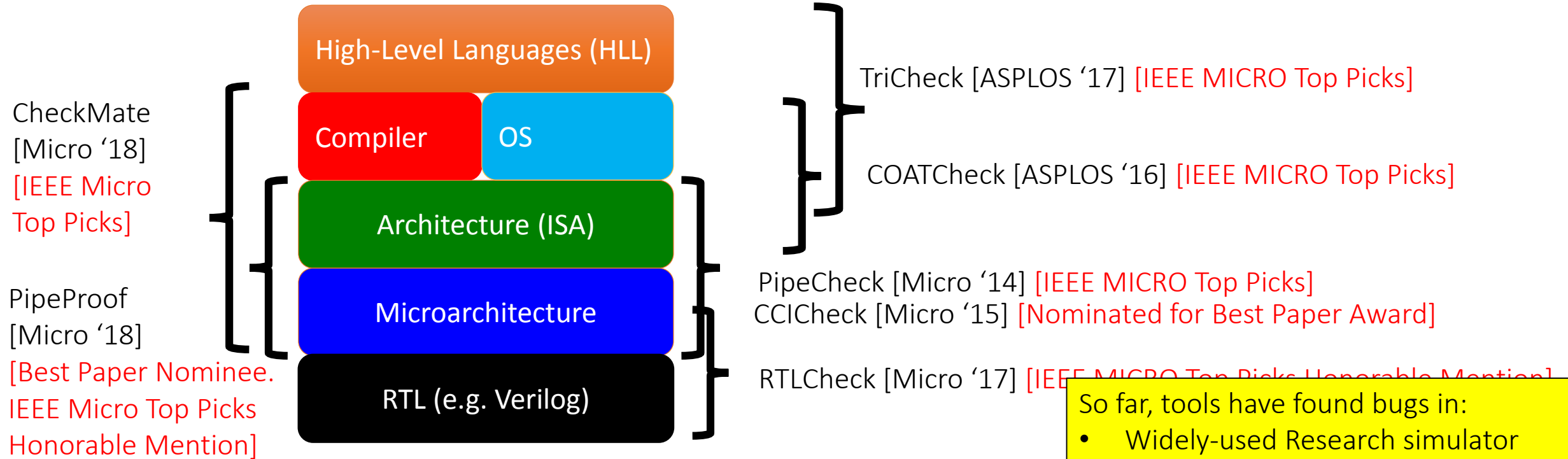


## Our Approach

- 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**



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



## 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

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

# 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)

