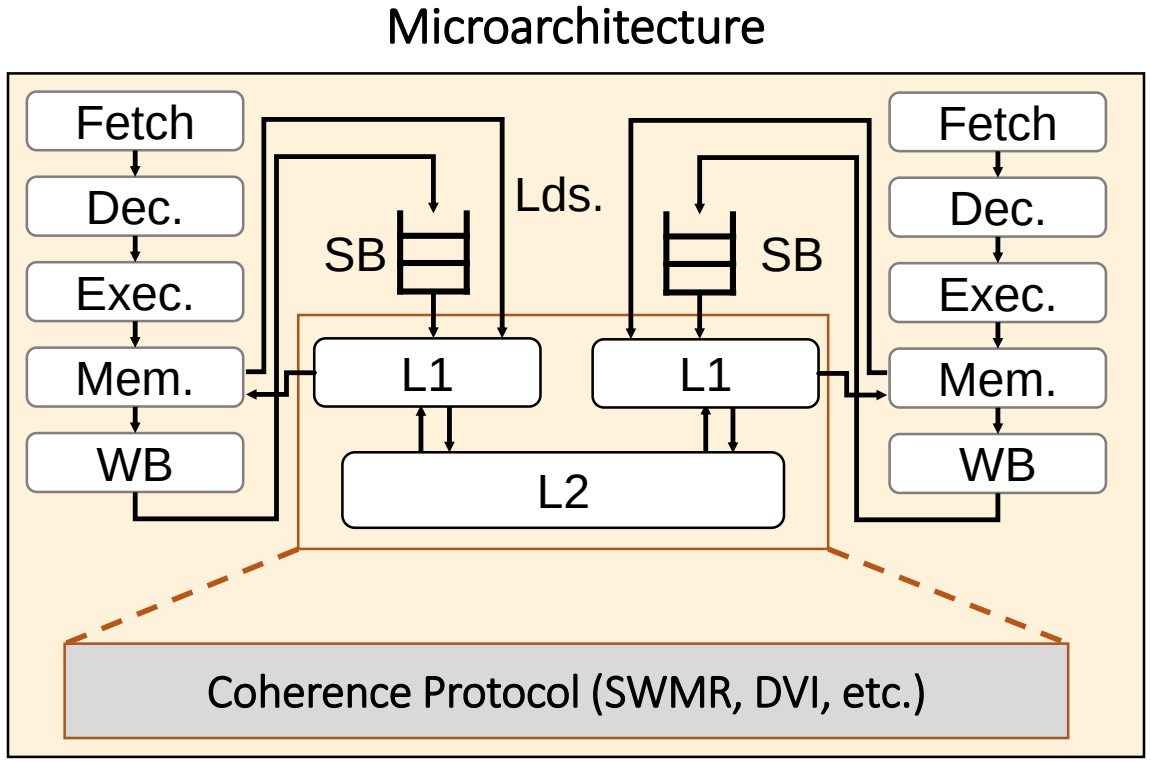
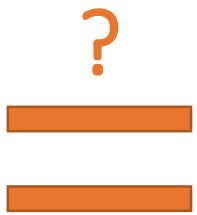
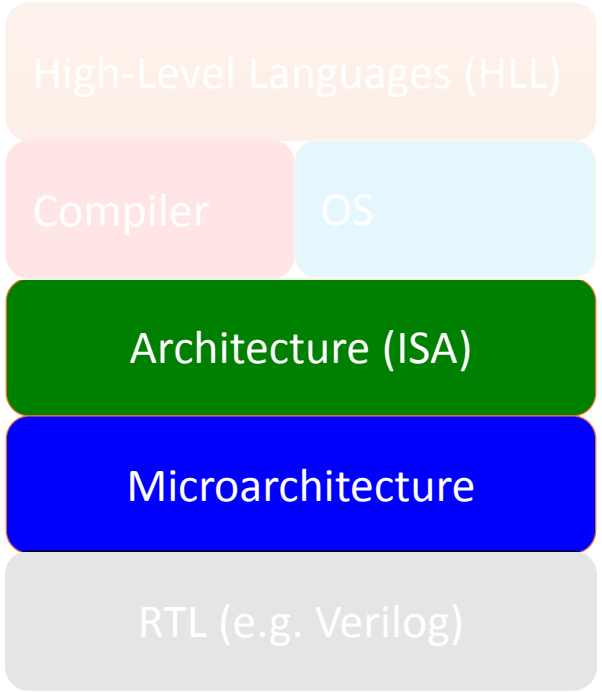
Up and Down the Stack!

What we did before the break…

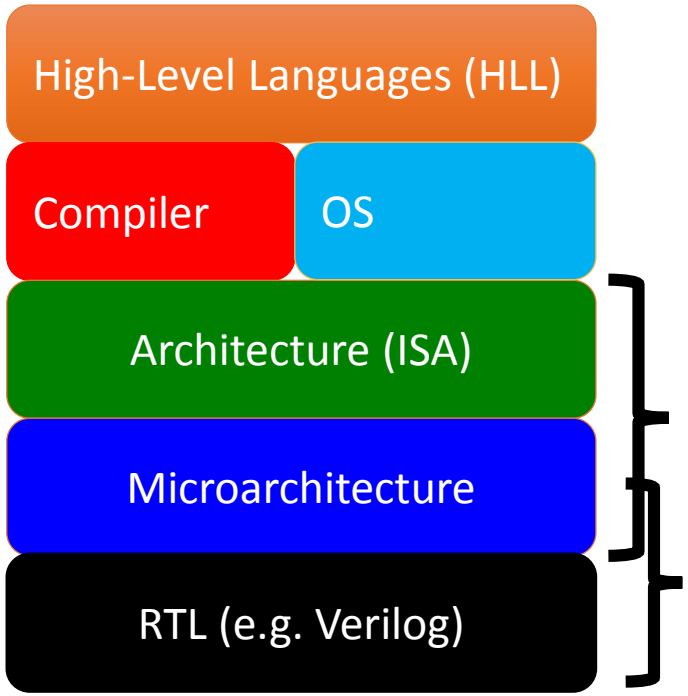




SC/TSO/RISC-V MCM?



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]

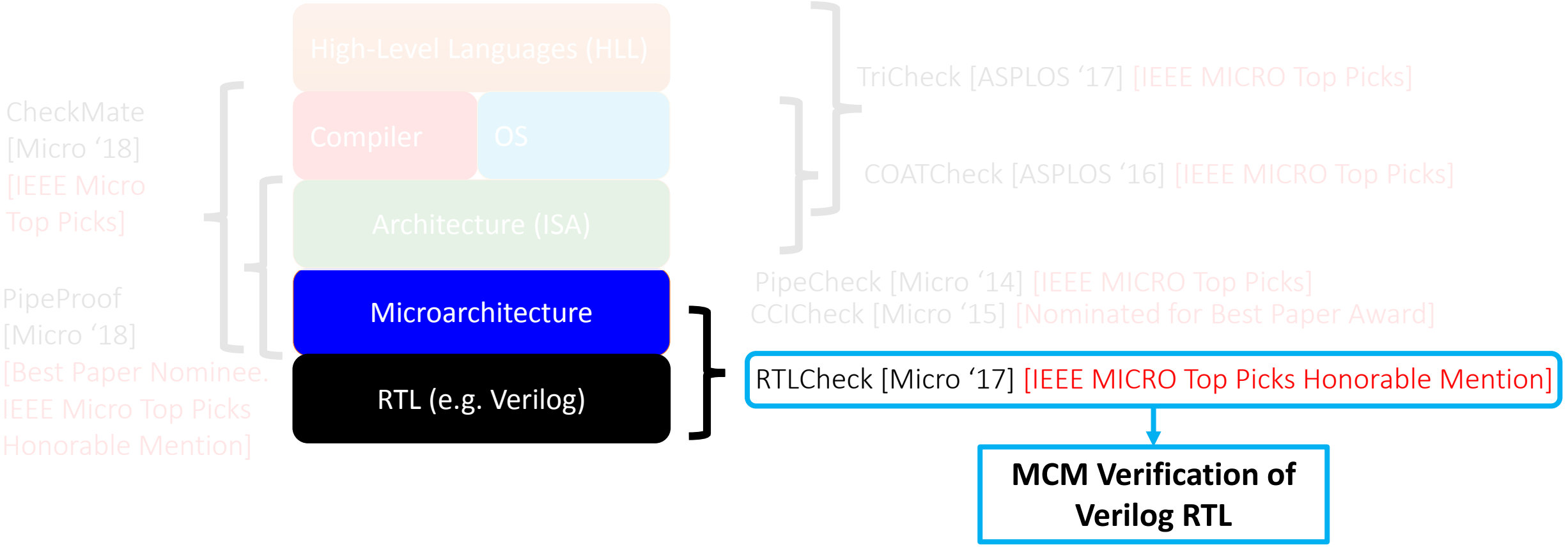
PipeCheck [Micro '14] [IEEE MICRO Top Picks]

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

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

For more info: check.cs.Princeton.edu

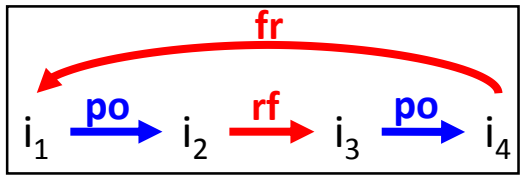
The Check Suite: Tools For Verifying Memory Orderings and their Security Implications



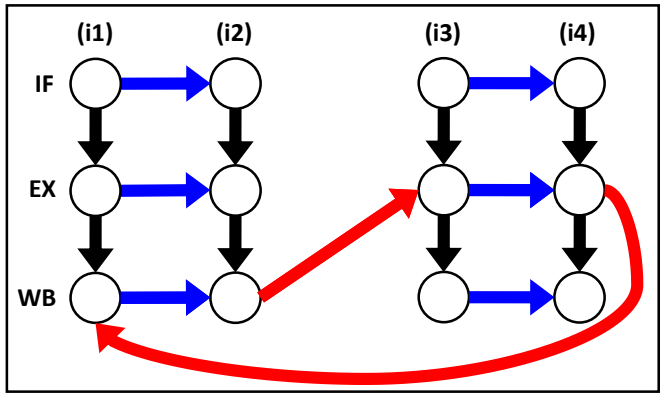
For more info: check.cs.Princeton.edu

What if I want to verify RTL (Verilog)?

ISA-Level MCM



Microarchitectural Orderings




```
acyclic (po U co U rf U fr)
```

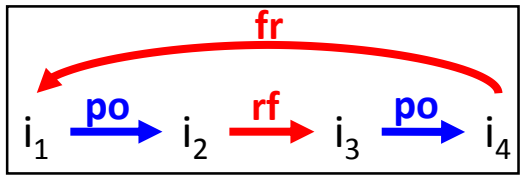
=

```
Axiom "PO_Fetch":
forall microop "i1", "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
      AddEdge ((i1, IF), (i2, IF)).
...
```

Verified with PipeProof

What if I want to verify RTL (Verilog)?

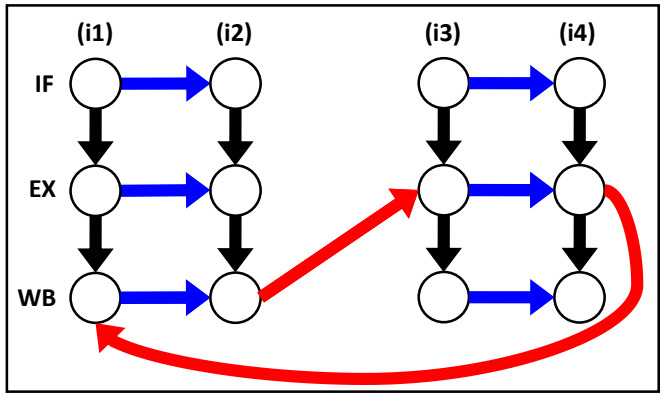
ISA-Level MCM




```
acyclic (po U co U rf U fr)
```

=

Microarchitectural Orderings

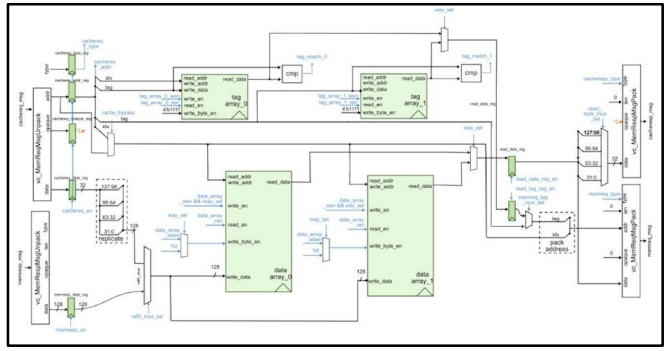



```
Axiom "PO_Fetch":
forall microop "i1", "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
      AddEdge ((i1, IF), (i2, IF)).
...
```

Verified with PipeProof

?=

RTL implementation (Verilog)

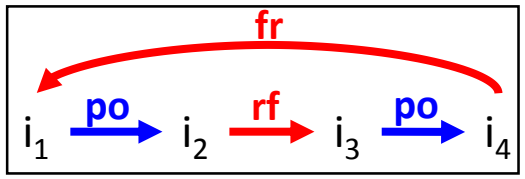


?

[RTL Image: Christopher Batten]

What if I want to verify RTL (Verilog)?

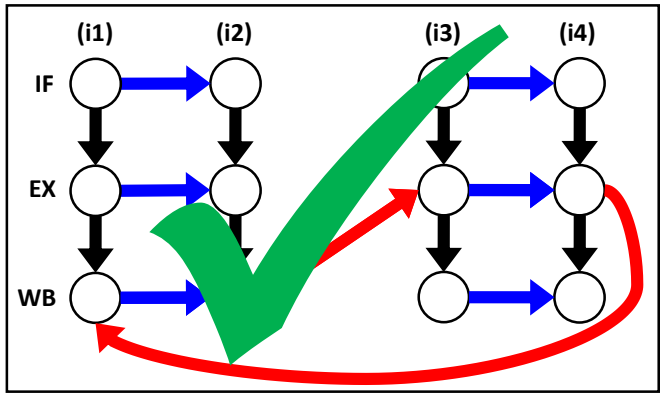
ISA-Level MCM




```
acyclic (po U co U rf U fr)
```

=

Microarchitectural Orderings

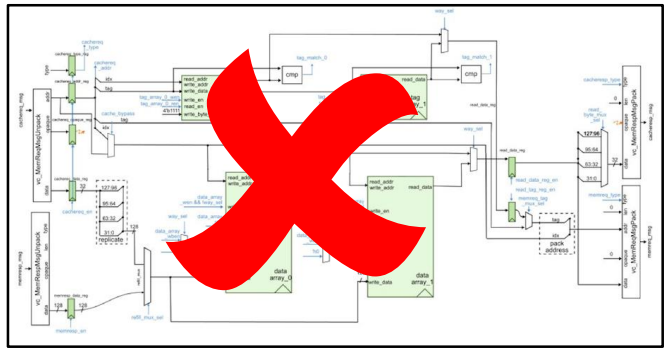



```
Axiom "PO_Fetch":
forall microop "i1", "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
      AddEdge ((i1, IF), (i2, IF)).
...
```

Verified with PipeProof

?=

RTL implementation (Verilog)



?

[RTL Image: Christopher Batten]

RTL Verification is Maturing…

- …but usually ignores memory consistency!

RTL Verification is Maturing…

- …but usually ignores memory consistency!

ISA-Formal [Reid et al. CAV 2016]
-Instr. Operational Semantics

No MCM verification!

RTL Verification is Maturing…

- …but usually ignores memory consistency!

ISA-Formal [Reid et al. CAV 2016]
-Instr. Operational Semantics

No MCM verification!

DOGReL [Stewart et al. DIFTS 2014]
-Memory subsystem transactions

No multicore MCM verification!

RTL Verification is Maturing…

- …but usually ignores memory consistency!

ISA-Formal [Reid et al. CAV 2016]
-Instr. Operational Semantics

No MCM verification!

DOGReL [Stewart et al. DIFTS 2014]
-Memory subsystem transactions

No multicore MCM verification!

Kami
[Vijayaraghavan et al. CAV 2015] [Choi et al. ICFP 2017]
-MCM correctness for all programs, but…

Needs Bluespec design and manual proofs!

RTL Verification is Maturing…

- …but usually ignores memory consistency!

Lack of automated memory consistency verification at RTL!

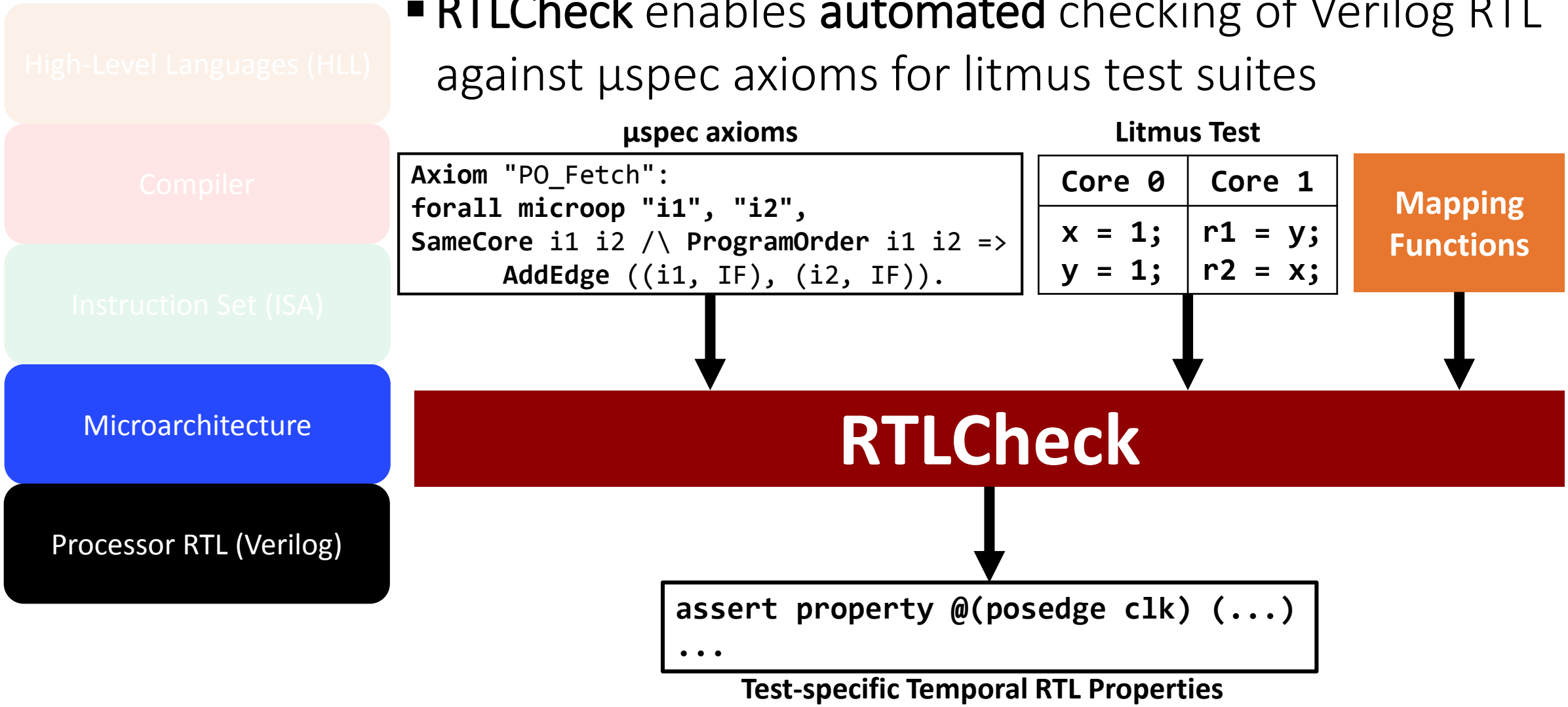
[Vijayaraghavan et al. CAV 2015] [Choi et al. ICFP 2017]

-MCM correctness for all programs, but…

Needs Bluespec design and manual proofs!

RTLCheck: Checking RTL Consistency Orderings

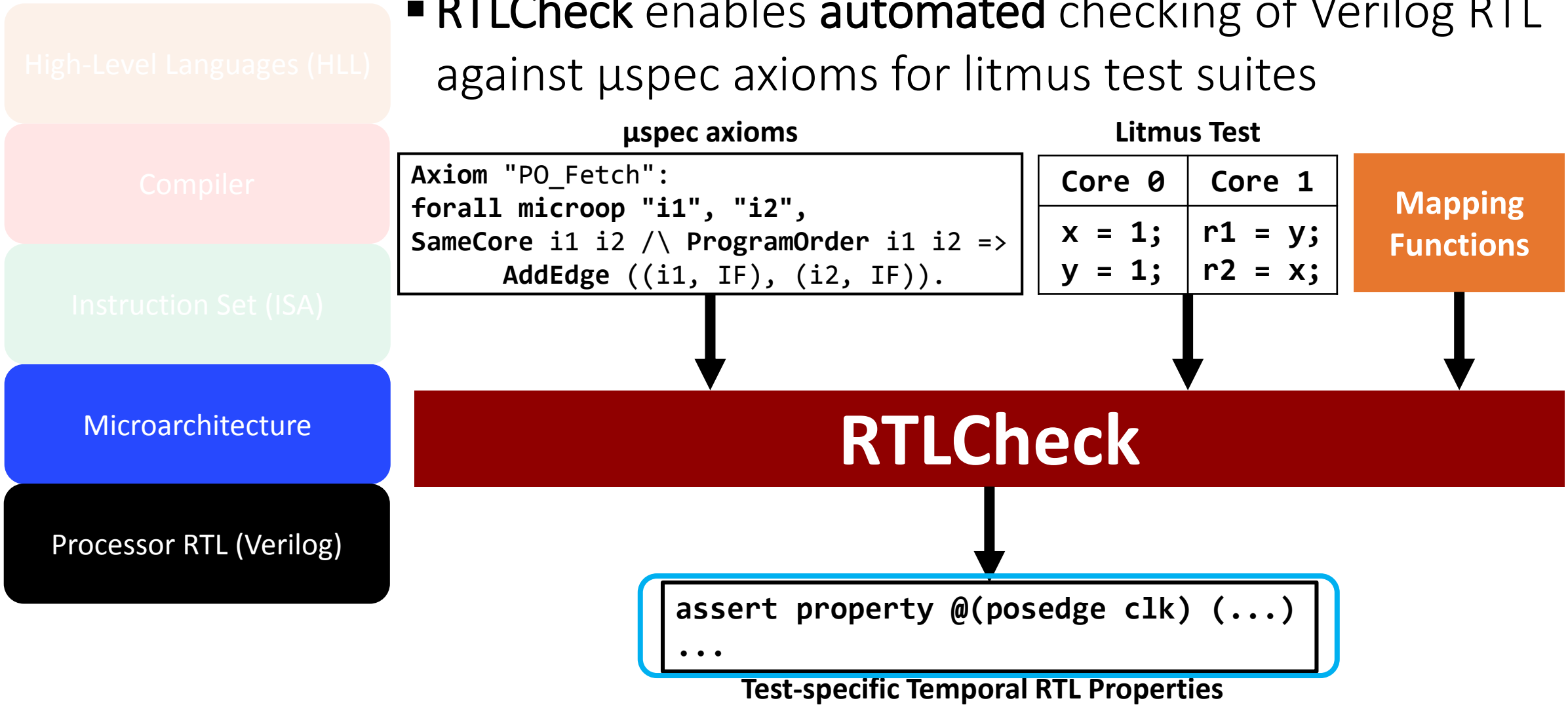
- **RTLCheck** enables **automated** checking of Verilog RTL against μspec axioms for litmus test suites



[**Yatin A. Manerkar**, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the Memory Consistency of RTL Designs. The 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), October 2017.]

RTLCheck: Checking RTL Consistency Orderings

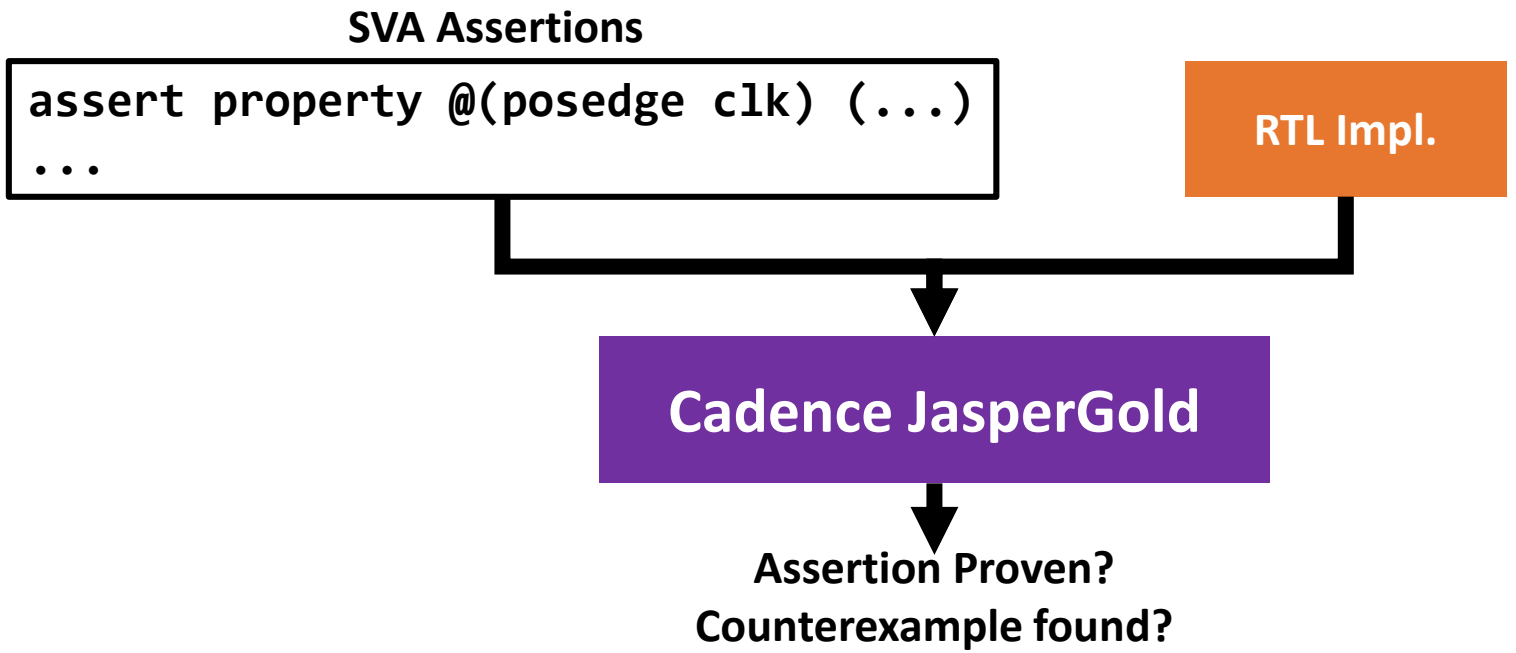
- **RTLCheck** enables **automated** checking of Verilog RTL against μspec axioms for litmus test suites



[**Yatin A. Manerkar**, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the Memory Consistency of RTL Designs. The 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), October 2017.]

SystemVerilog Assertions (SVA)

- SVA: Industry standard for RTL verification, e.g.: ARM [Reid et al. CAV 2016]
  - Based on Linear Temporal Logic (LTL) with regular operators
- Commercial tools (e.g. JasperGold) can formally verify SVA assertions
- **Translating μspec to SVA => RTL MCM verification using industry flows**
- **But it's not that simple!**



Meaning can be Lost in Translation!

小心地滑

Meaning can be Lost in Translation!

小心地滑

(Caution: Slippery Floor)

Meaning can be Lost in Translation!

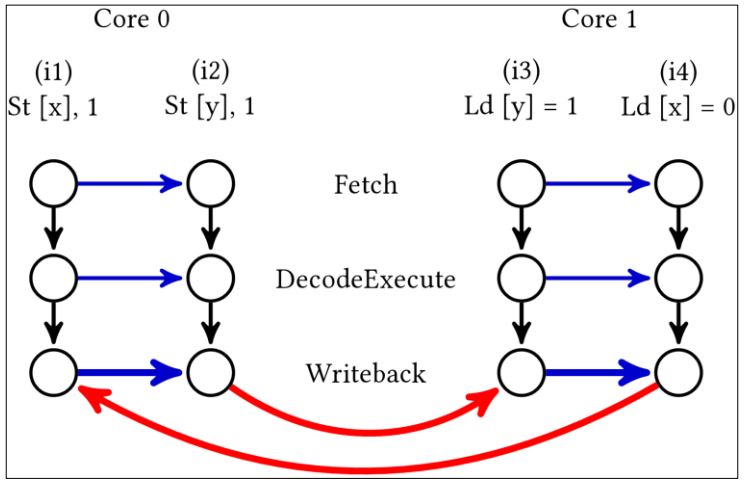
小心地滑

(Caution: Slippery Floor)

[Image: Barbara Younger]
[Inspiration: Tae Jun Ham]

The μspec/SVA Mismatch

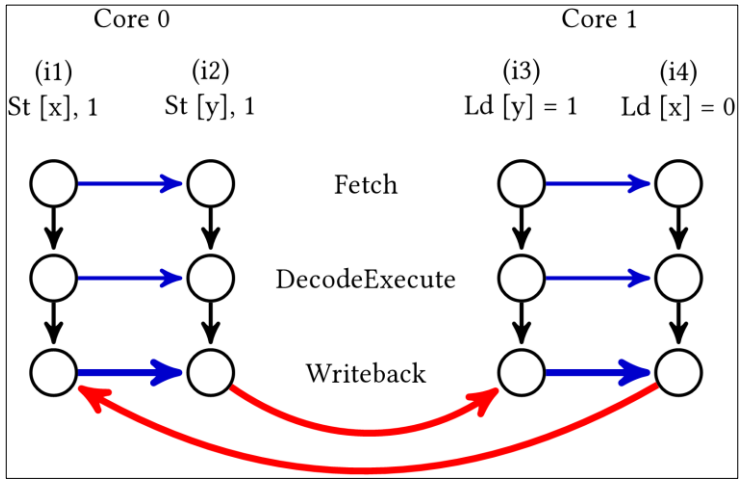
Axiomatic
Microarch.
Verification



Execution examined as a single unit (graph)

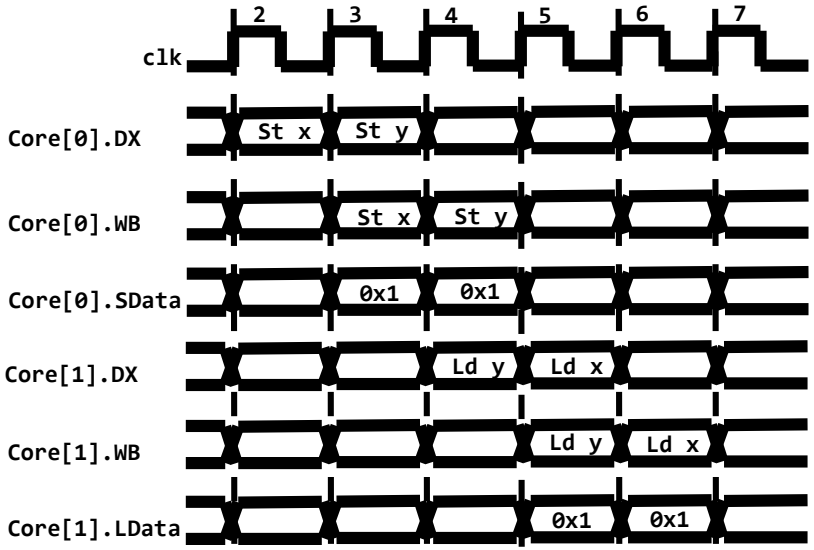
The μspec/SVA Mismatch

Axiomatic
Microarch. Verification



Execution examined as a single unit (graph)

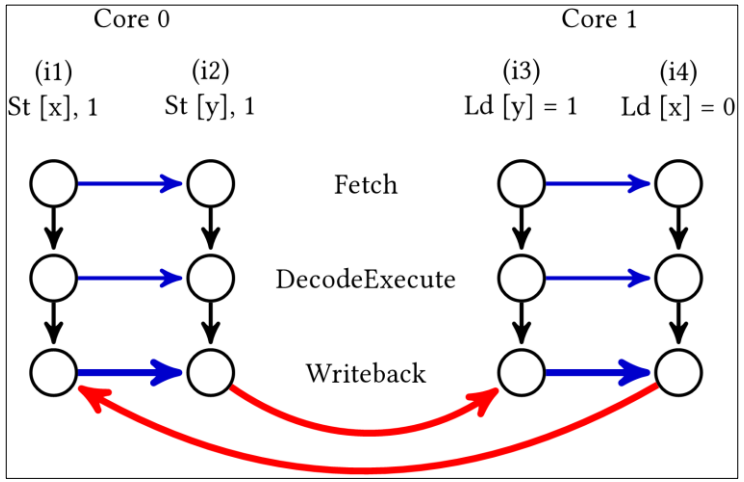
Temporal
RTL Verification (SVA, etc)



Execution examined cycle by cycle

The μspec/SVA Mismatch

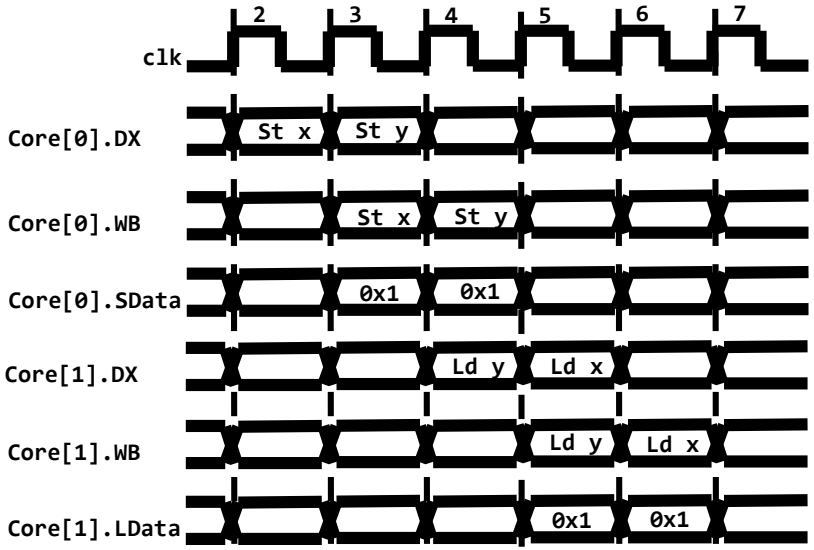
Axiomatic
Microarch. Verification



Execution examined as a single unit (graph)

μspec/SVA Mismatch!

Temporal
RTL Verification (SVA, etc)



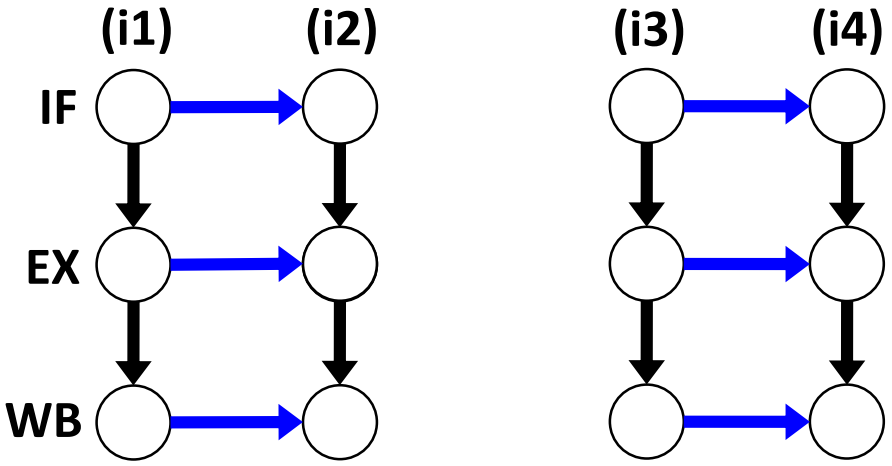
Execution examined cycle by cycle

The μspec/SVA Mismatch

- Tricky to translate μspec to SVA while maintaining μspec semantics
- SVA Verifiers (JasperGold) don't implement full SVA spec!
  - Causes further complications
- **Example: Outcome Filtering**
  - Filtering litmus test executions to those that have particular values for loads

Outcome Filtering with Execution as a Single Unit

- In this case, outcome filtering is **easy and efficient**
- Always know what the load values are
  - Can draw (red) edges based on these values

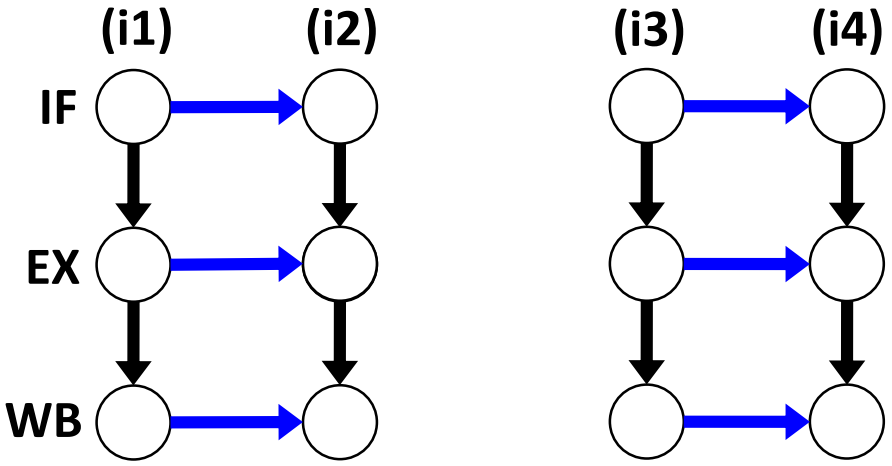


mp litmus test

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;

Outcome Filtering with Execution as a Single Unit

- In this case, outcome filtering is **easy and efficient**
- Always know what the load values are
  - Can draw (red) edges based on these values

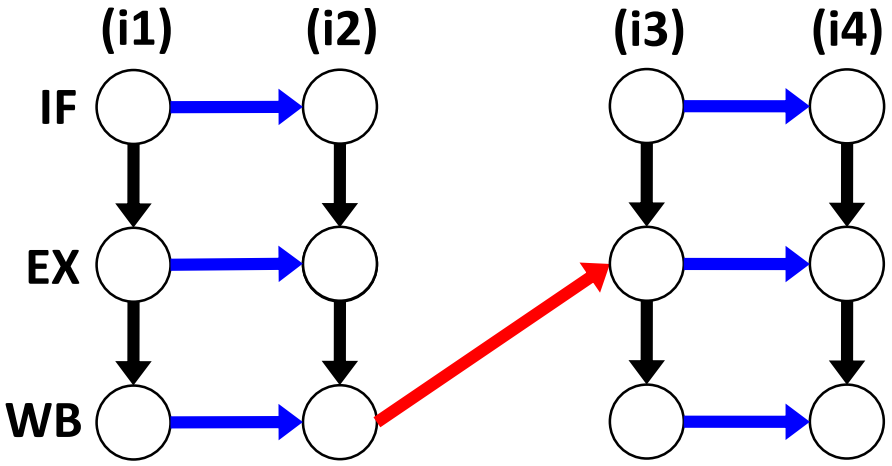


mp litmus test

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

Outcome Filtering with Execution as a Single Unit

- In this case, outcome filtering is **easy and efficient**
- Always know what the load values are
  - Can draw (red) edges based on these values

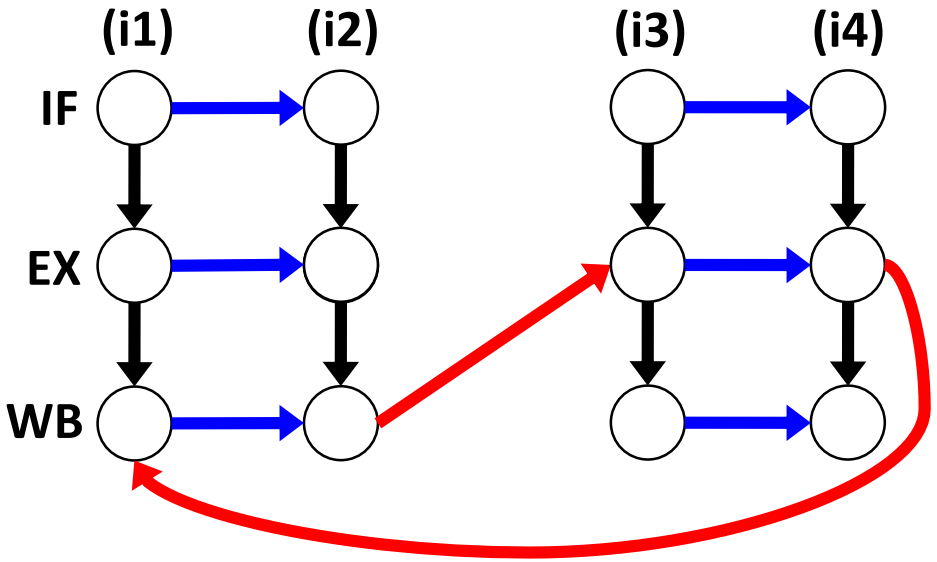


mp litmus test

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

Outcome Filtering with Execution as a Single Unit

- In this case, outcome filtering is **easy and efficient**
- Always know what the load values are
  - Can draw (red) edges based on these values



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	

Outcome Filtering when Executing Cycle by Cycle

- Don’t know load values until the end of the execution!
- **Must look into future** to ensure we’re checking the right executions

mp litmus test

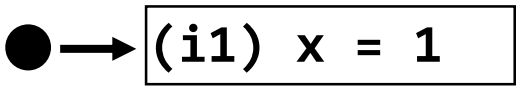
Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
SC Forbids: r1 = 1, r2 = 0	

Outcome Filtering when Executing Cycle by Cycle

- Don’t know load values until the end of the execution!
- **Must look into future** to ensure we’re checking the right executions

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	



Step 1

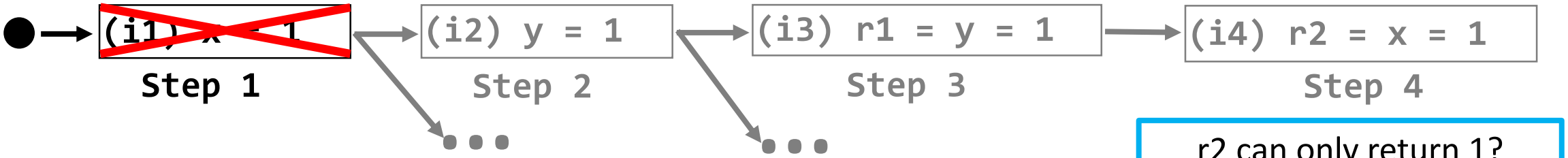
Does this path correspond to r1=1, r2=0?
Need to look into future!

Outcome Filtering when Executing Cycle by Cycle

- Don’t know load values until the end of the execution!
- **Must look into future** to ensure we’re checking the right executions

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	



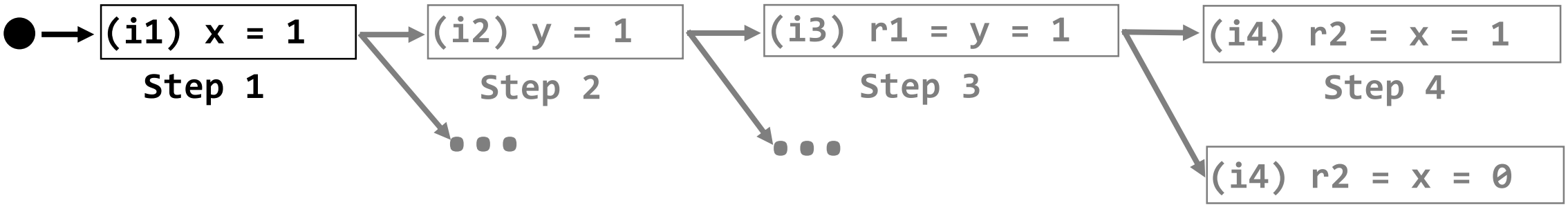
r2 can only return 1?
Don’t look at this path!

Outcome Filtering when Executing Cycle by Cycle

- Don't know load values until the end of the execution!
- **Must look into future** to ensure we're checking the right executions

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	



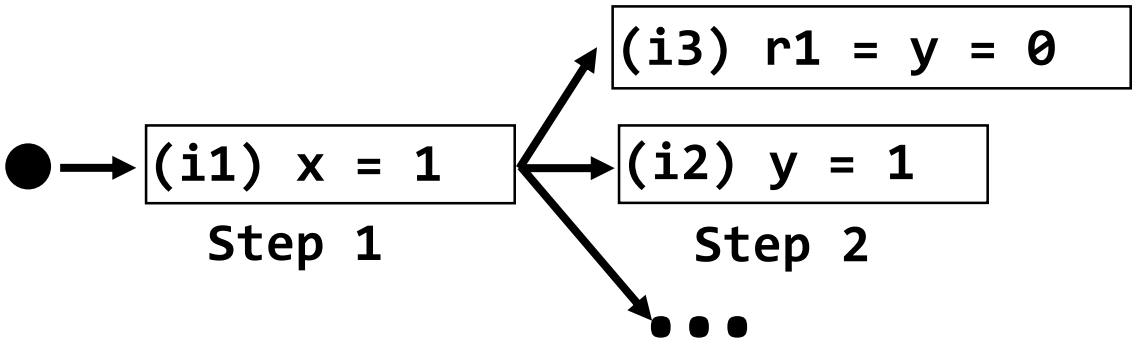
r2 can return 0?
Carry on to step 2.

Outcome Filtering when Executing Cycle by Cycle

- Don’t know load values until the end of the execution!
- **Must look into future** to ensure we’re checking the right executions

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	



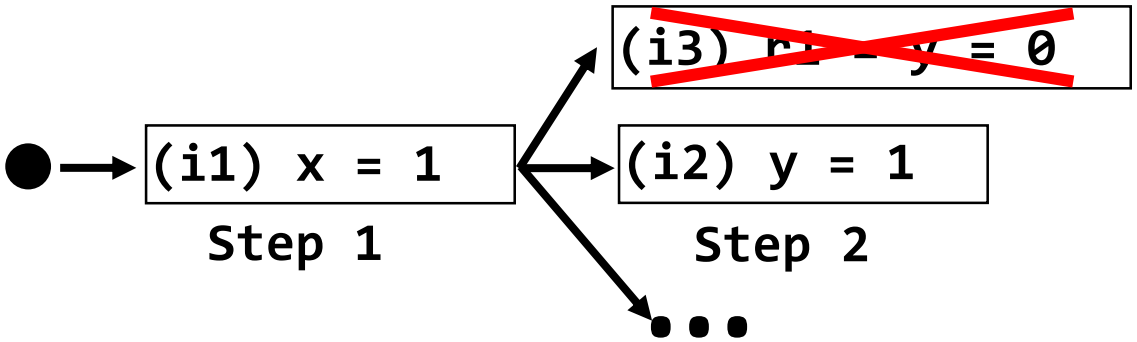
Do these paths correspond to r1=1,r2=0?
Look into future again!

Outcome Filtering when Executing Cycle by Cycle

- Don’t know load values until the end of the execution!
- **Must look into future** to ensure we’re checking the right executions

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	



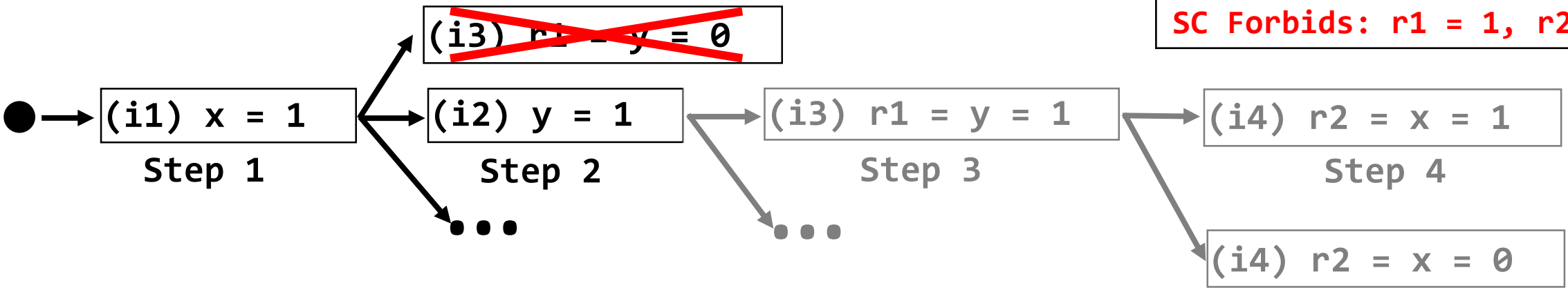
Do these paths correspond to r1=1,r2=0?
Look into future again!

Outcome Filtering when Executing Cycle by Cycle

- Don’t know load values until the end of the execution!
- **Must look into future** to ensure we’re checking the right executions

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	

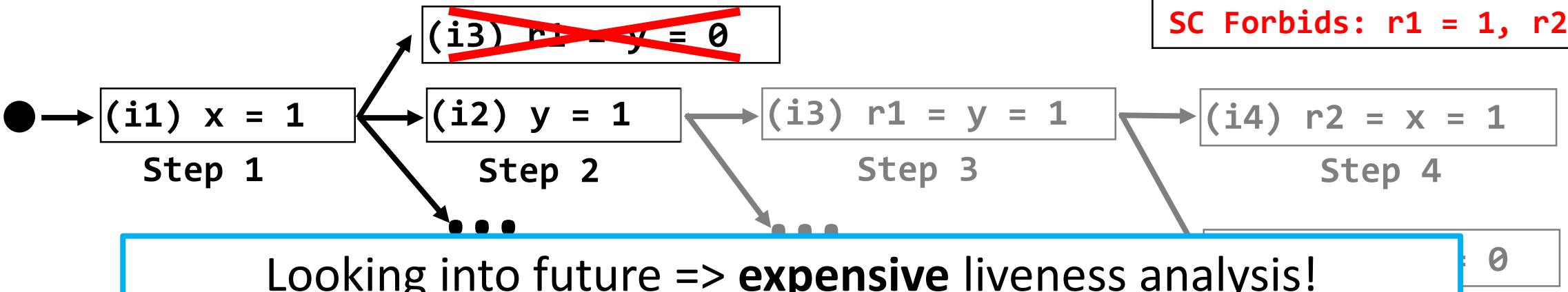


Outcome Filtering when Executing Cycle by Cycle

- Don't know load values until the end of the execution!
- **Must look into future** to ensure we're checking the right executions

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	



Looking into future => **expensive** liveness analysis!
SVA Verifiers **approximate** by only checking upto current step!
Makes **outcome filtering impossible*** to do!

* = as far as we know

Solution: Load Value Constraints

mp

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

- Don't filter based on outcome
  - Translate **all** possible outcomes
- Tag each case with appropriate **load value constraints**
  - reflect the data constraints required for edge(s)
- Ongoing work: Precisely formalise the μspec/SVA mismatch
  - How much is fundamental? How much is due to SVA verifier approximation?

```
Axiom "Read_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite
```

```
Property to check:
mapNode(Ld x → St x, Ld x == 0) or mapNode(St x → Ld x, Ld x == 1);
```

Note: Axioms and properties abstracted for brevity

Solution: Load Value Constraints

mp

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

- Don't filter based on outcome
  - Translate **all** possible outcomes
- Tag each case with appropriate **load value constraints**
  - reflect the data constraints required for edge(s)
- Ongoing work: Precisely formalise the μspec/SVA mismatch
  - How much is fundamental? How much is due to SVA verifier approximation?

```
Axiom "Read_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite
```

```
Property to check:
mapNode(Ld x → St x, Ld x == 0) or mapNode(St x → Ld x, Ld x == 1);
```

Note: Axioms and properties abstracted for brevity

Solution: Load Value Constraints

mp

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

- Don't filter based on outcome
  - Translate **all** possible outcomes
- Tag each case with appropriate **load value constraints**
  - reflect the data constraints required for edge(s)
- Ongoing work: Precisely formalise the μspec/SVA mismatch
  - How much is fundamental? How much is due to SVA verifier approximation?

```
Axiom "Read_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite
```

```
Property to check:
mapNode(Ld x → St x, Ld x == 0) or mapNode(St x → Ld x, Ld x == 1);
```

Note: Axioms and properties abstracted for brevity

Solution: Load Value Constraints

mp

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

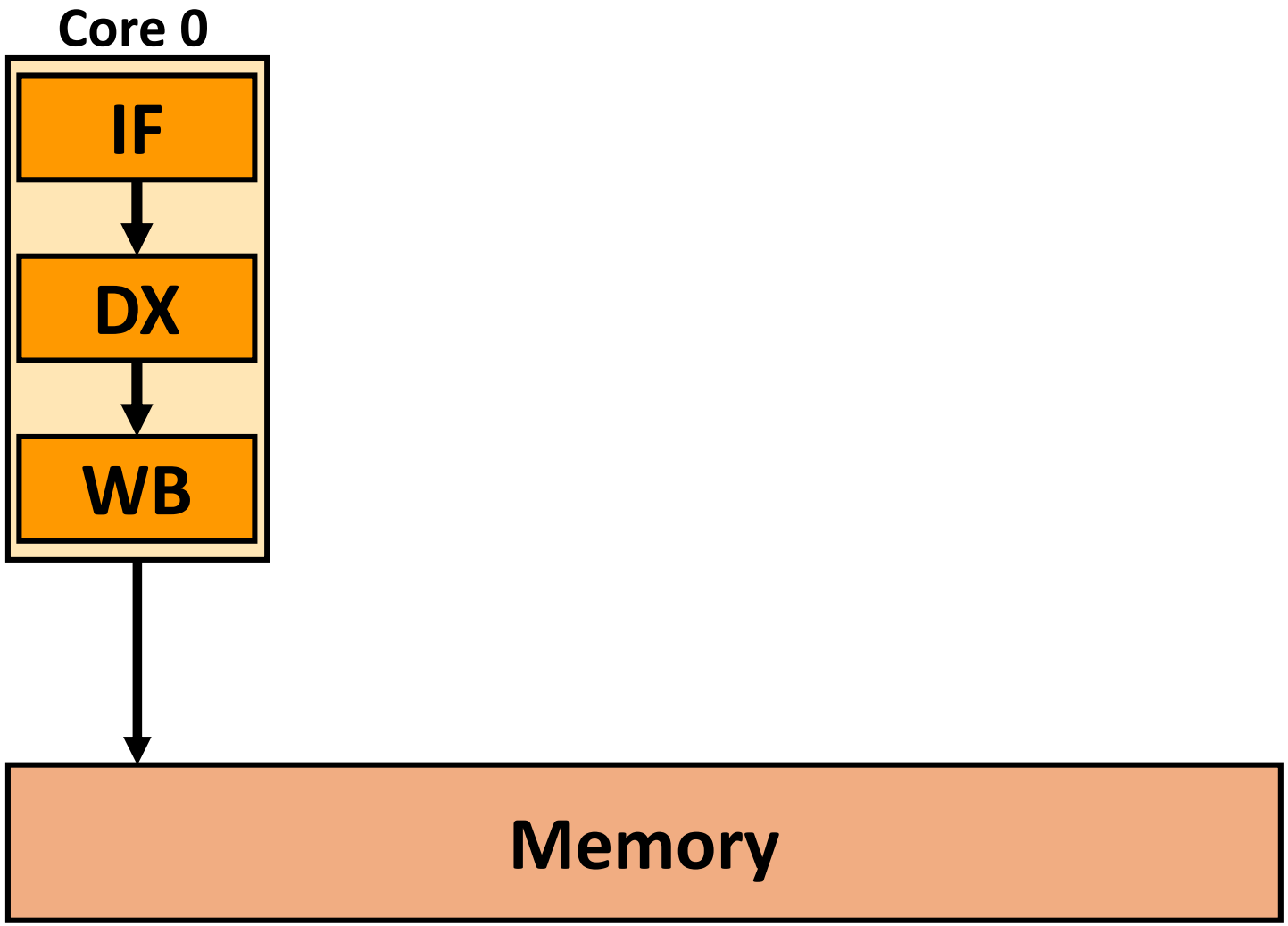
- Don't filter based on outcome
  - Translate **all** possible outcomes
- Tag each case with appropriate **load value constraints**
  - reflect the data constraints required for edge(s)
- Ongoing work: Precisely formalise the μspec/SVA mismatch
  - How much is fundamental? How much is due to SVA verifier approximation?

```
Axiom "Read_Values":
Every load either reads BeforeAllWrites OR reads FromLatestWrite
```

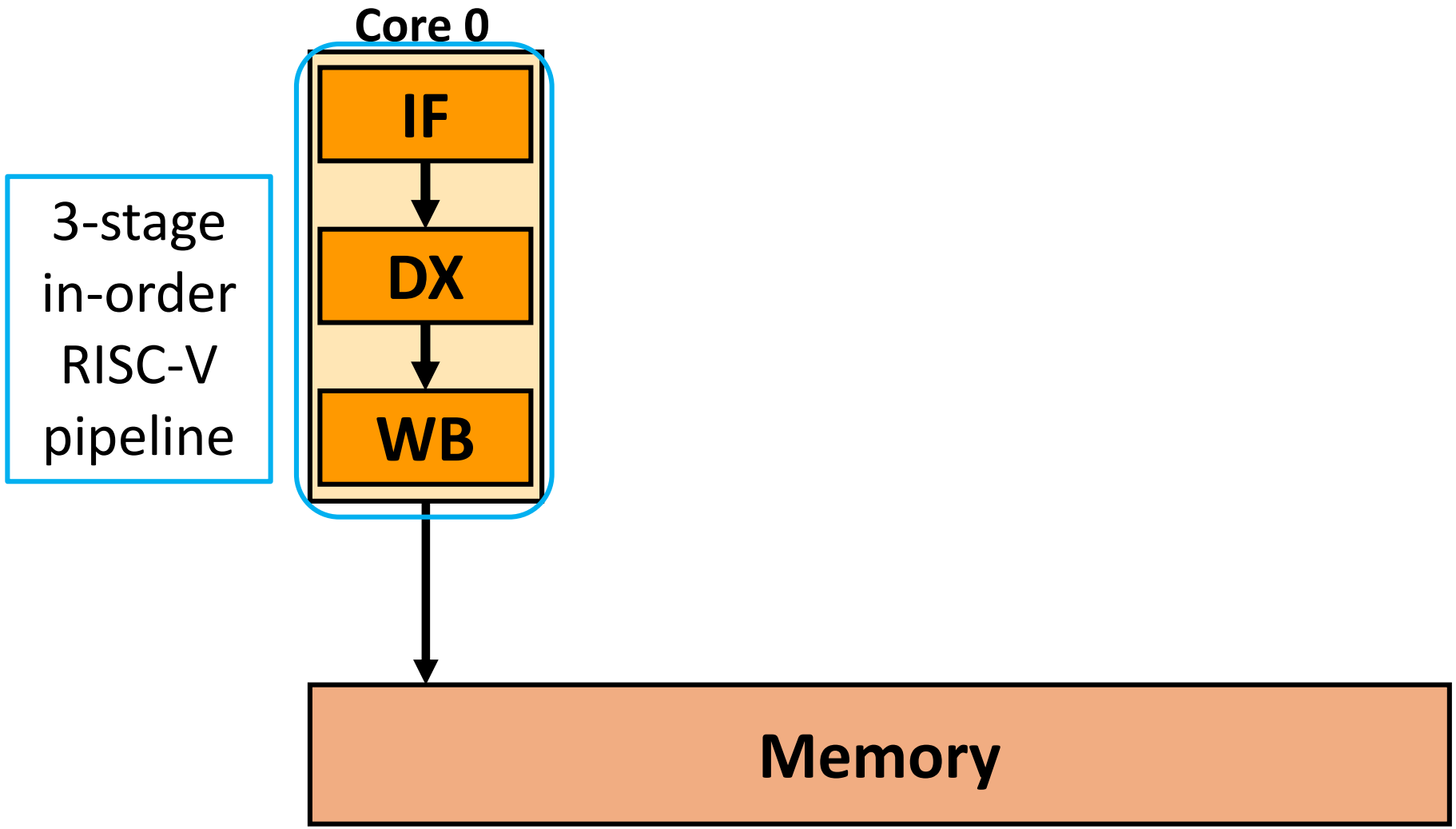
```
Property to check:
mapNode(Ld x → St x, Ld x == 0) or mapNode(St x → Ld x, Ld x == 1);
```

Note: Axioms and properties abstracted for brevity

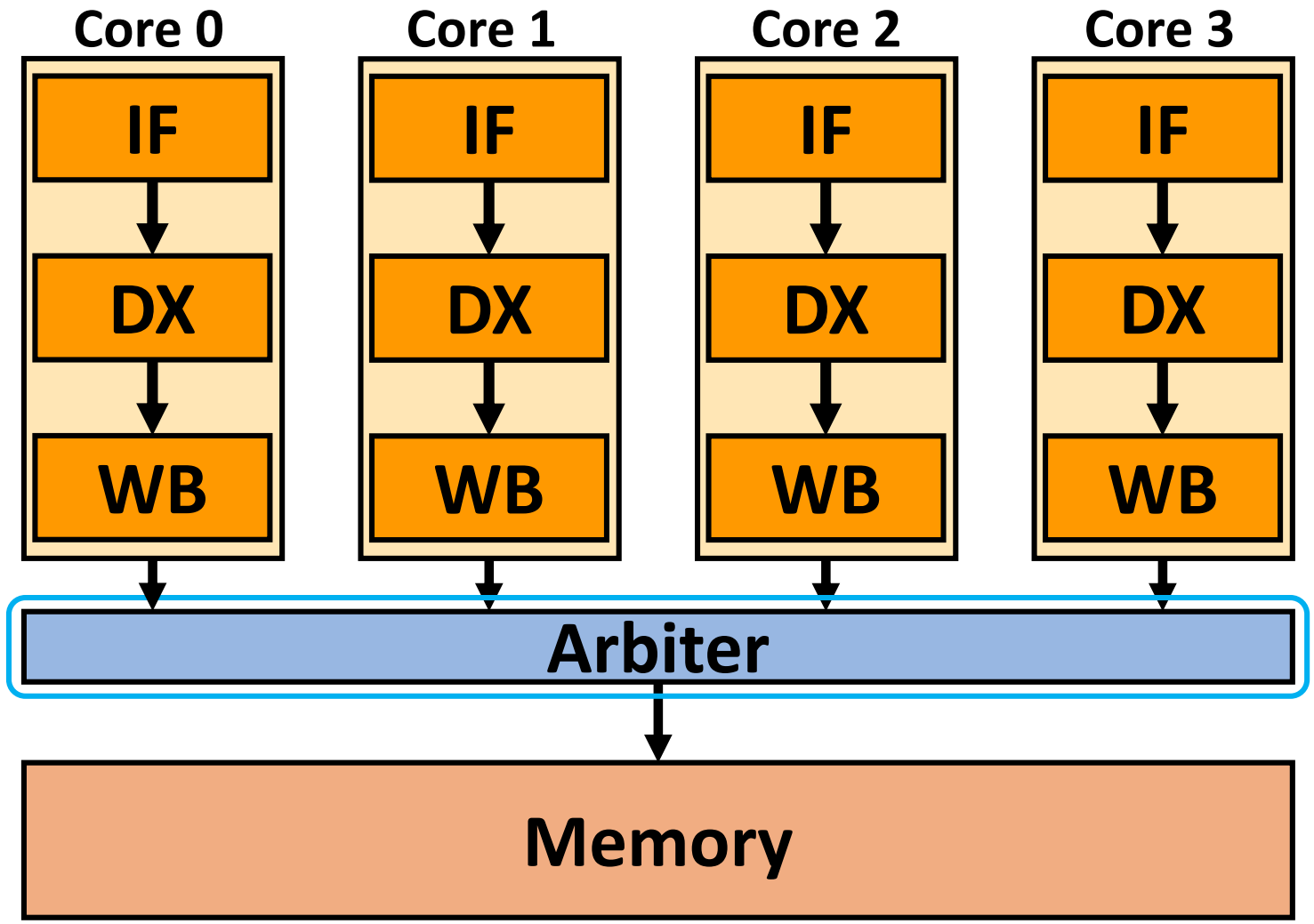
Multi-V-scale: a Multicore Case Study



Multi-V-scale: a Multicore Case Study



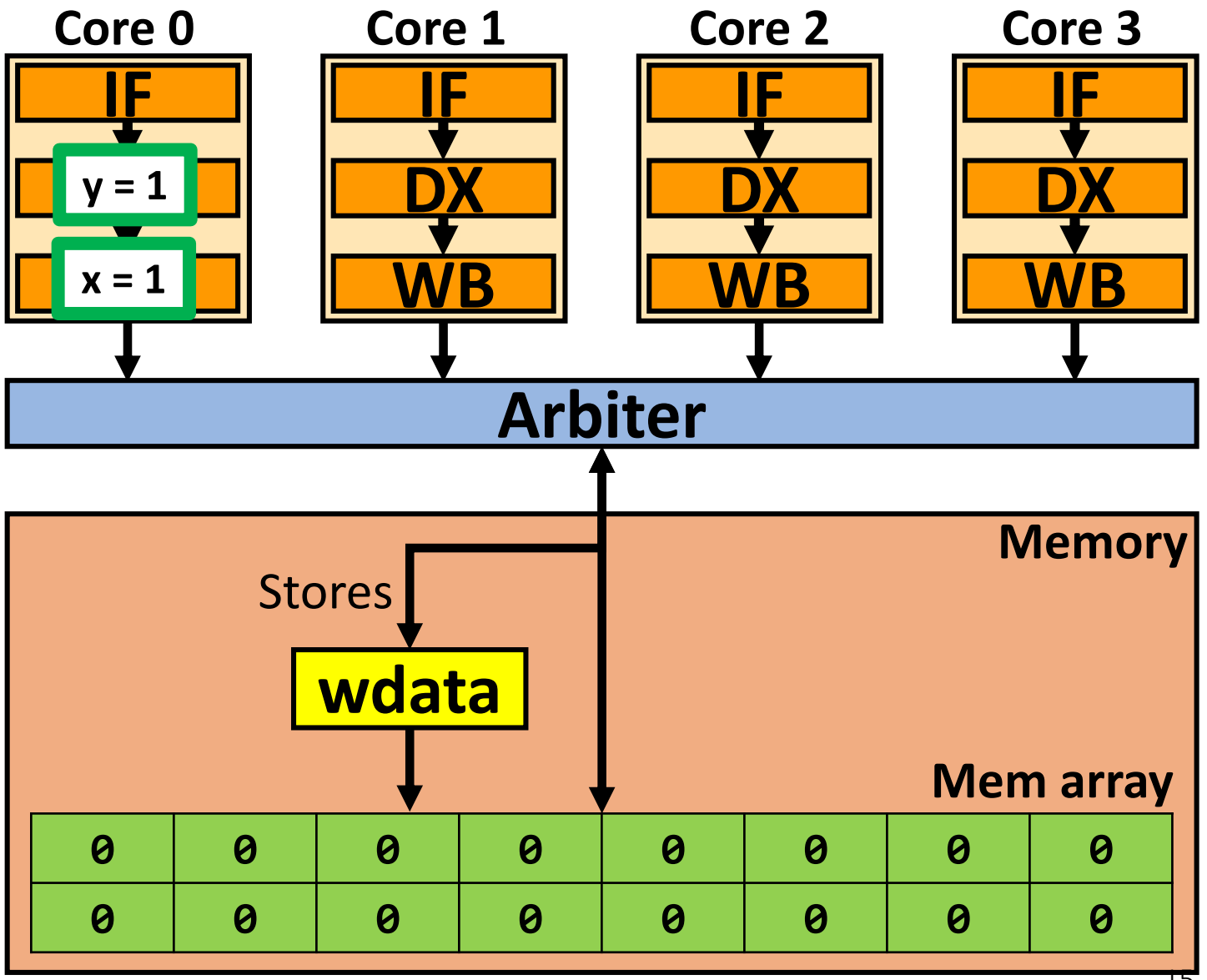
Multi-V-scale: a Multicore Case Study



Arbiter enforces that only **one core** can access memory at any time

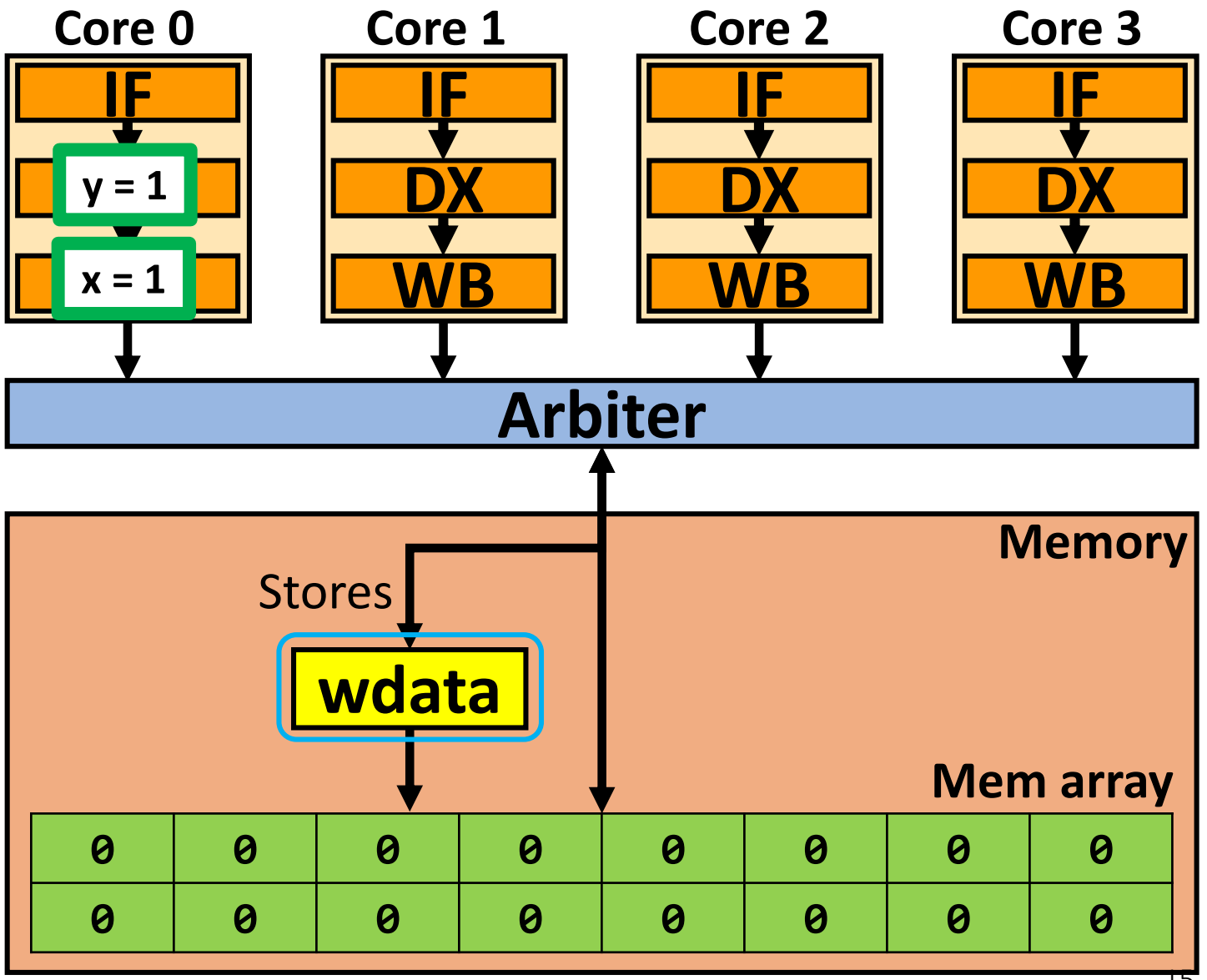
Bug Discovered in V-scale Mem. Implementation

- **When two stores are sent to memory in successive cycles, first of two stores is dropped by memory!**
- Bug would occur even in single-core V-scale
- Fixed bug by eliminating intermediate `wdata` reg



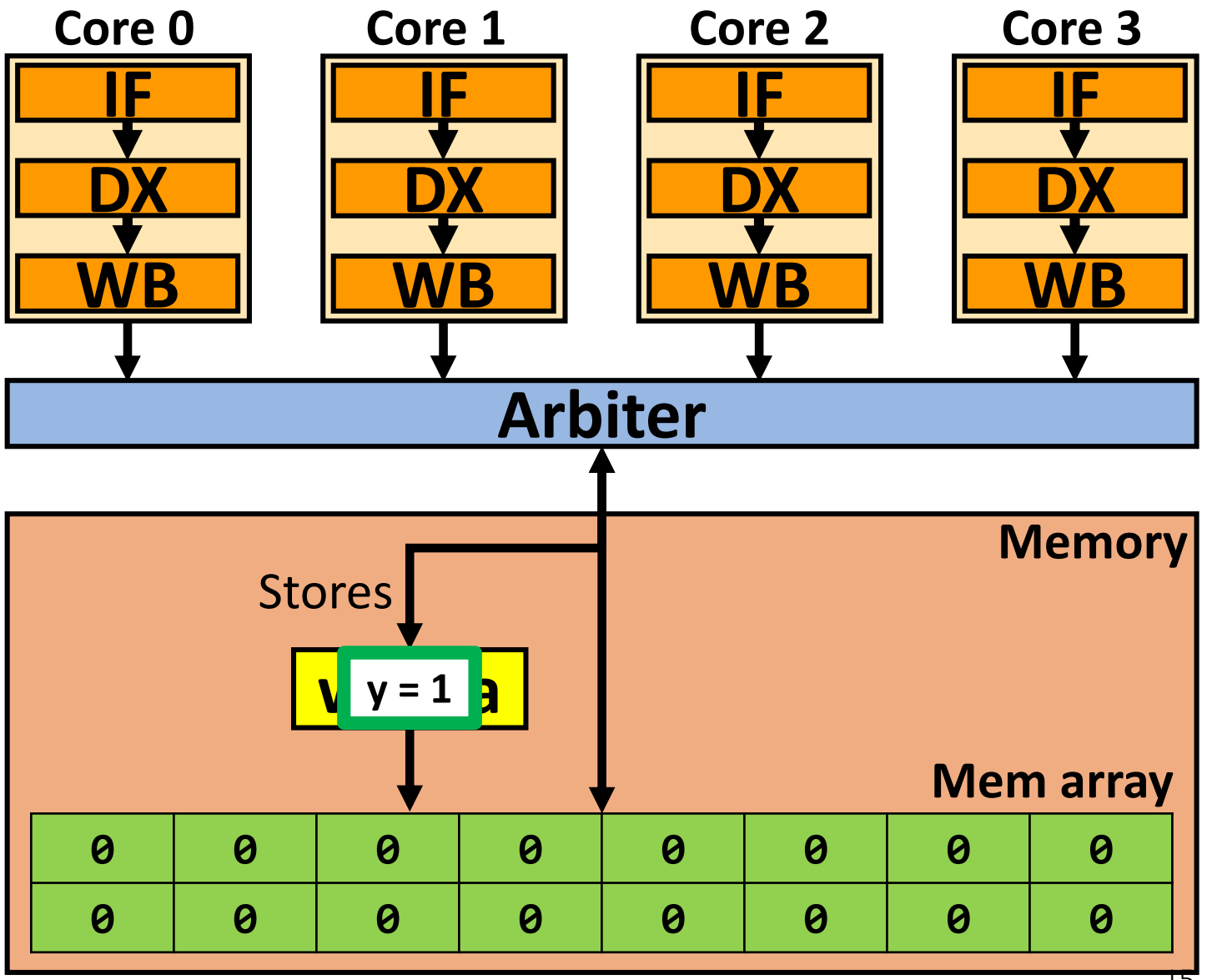
Bug Discovered in V-scale Mem. Implementation

- **When two stores are sent to memory in successive cycles, first of two stores is dropped by memory!**
- Bug would occur even in single-core V-scale
- Fixed bug by eliminating intermediate `wdata` reg



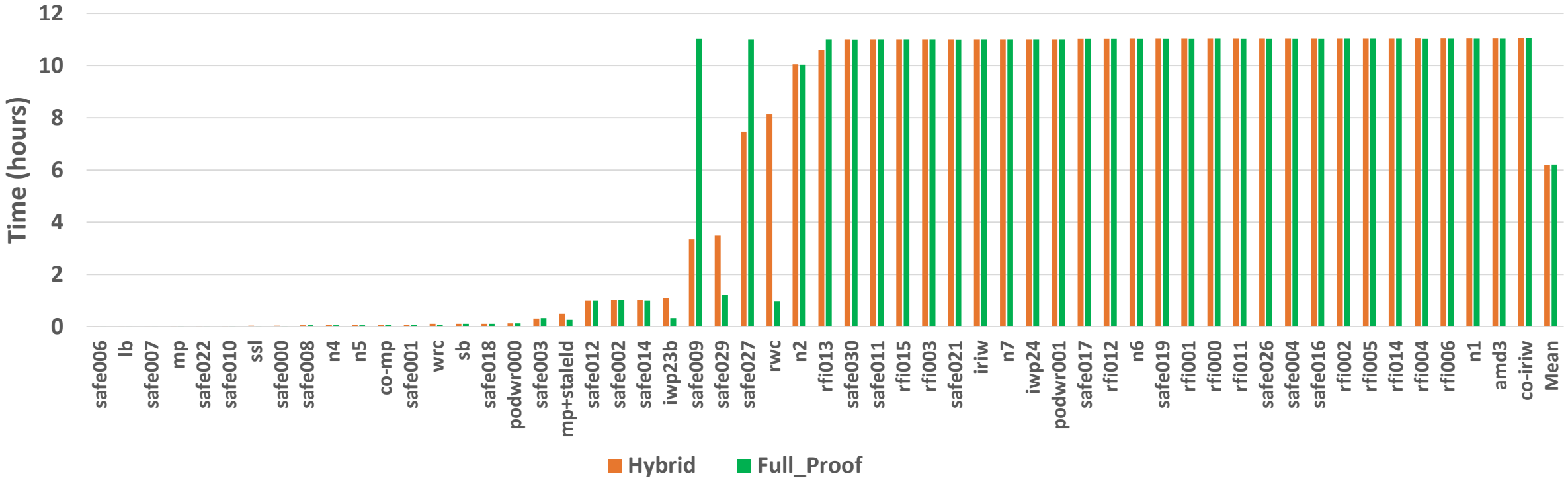
Bug Discovered in V-scale Mem. Implementation

- **When two stores are sent to memory in successive cycles, first of two stores is dropped by memory!**
- Bug would occur even in single-core V-scale
- Fixed bug by eliminating intermediate `wdata` reg



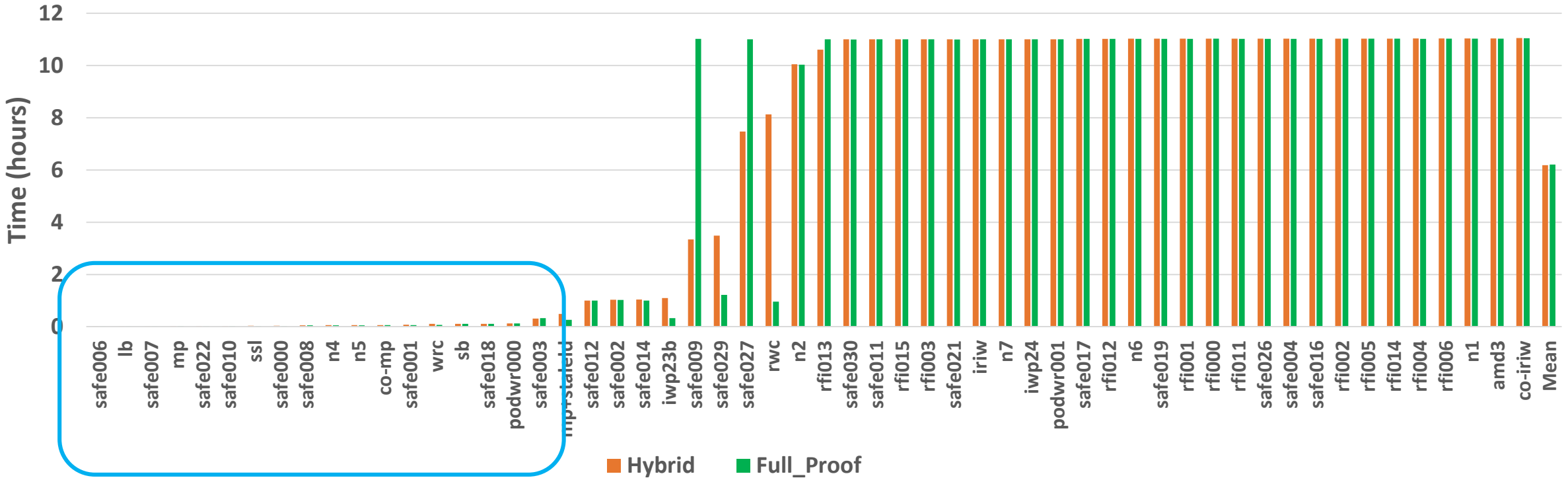
Results: Time to Prove Properties

- Two configurations (`Hybrid` and `Full_Proof`), avg. runtime 6.2 hrs



Results: Time to Prove Properties

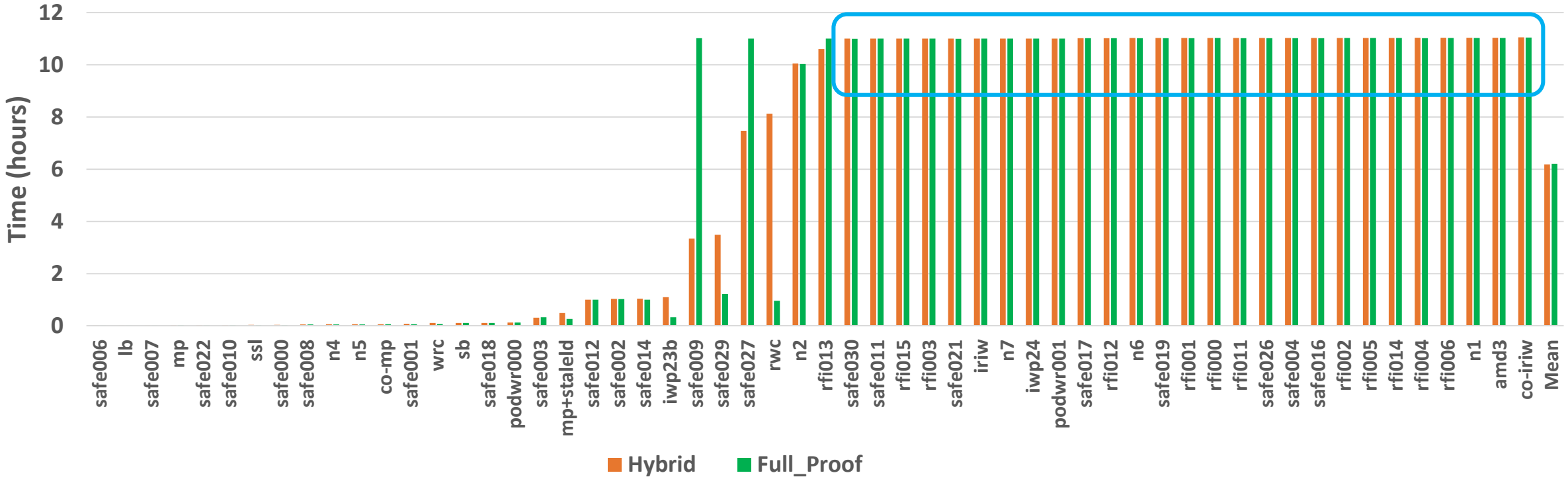
- Two configurations (`Hybrid` and `Full_Proof`), avg. runtime 6.2 hrs



Complete quickly when JasperGold detects that **litmus test outcome can never occur**

Results: Time to Prove Properties

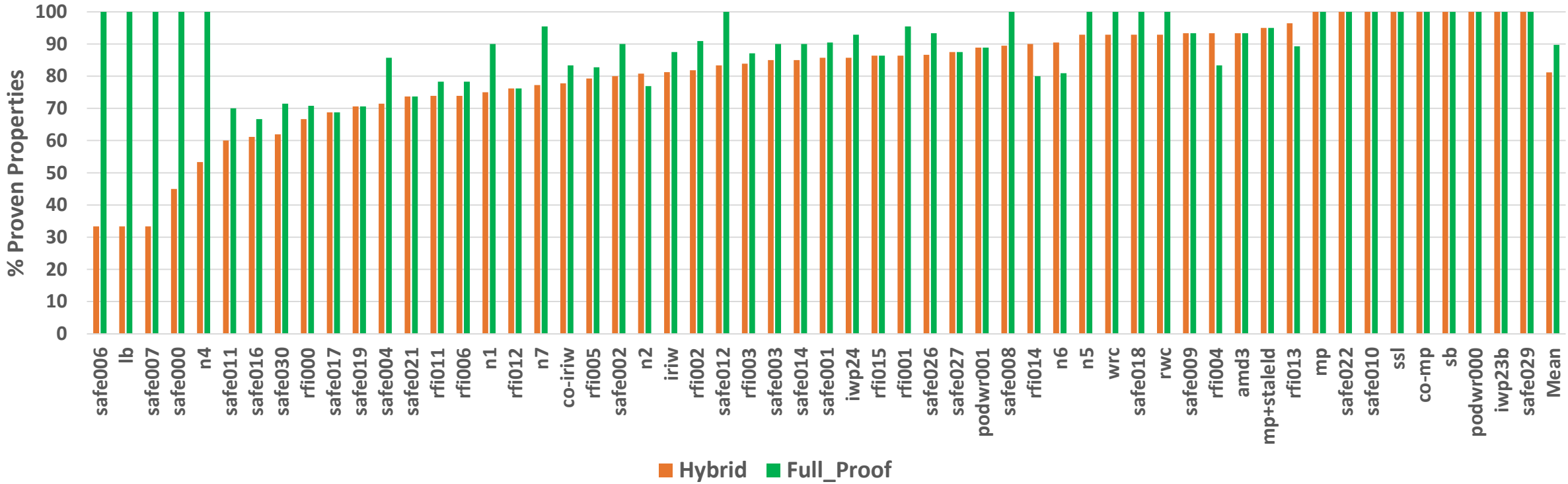
- Two configurations (`Hybrid` and `Full_Proof`), avg. runtime 6.2 hrs



Max runtime 11 hours (if some properties unproven)

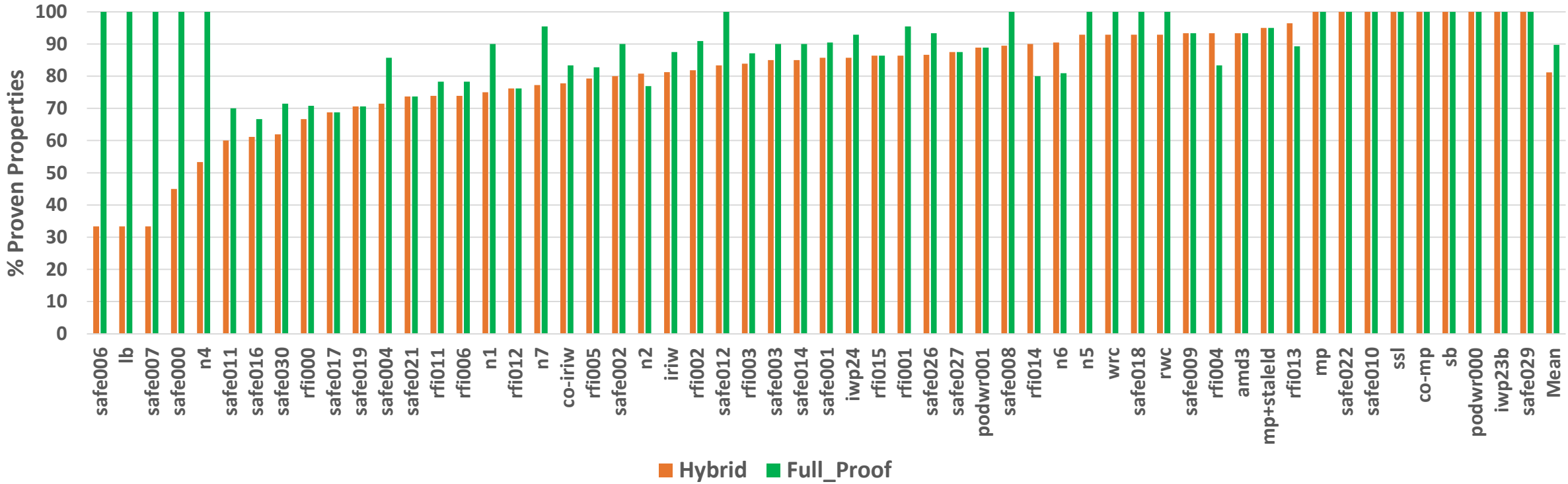
Results: Percentage of Proven Properties

- `Full_Proof` config generally better (90%/test) than `Hybrid` (81%/test)



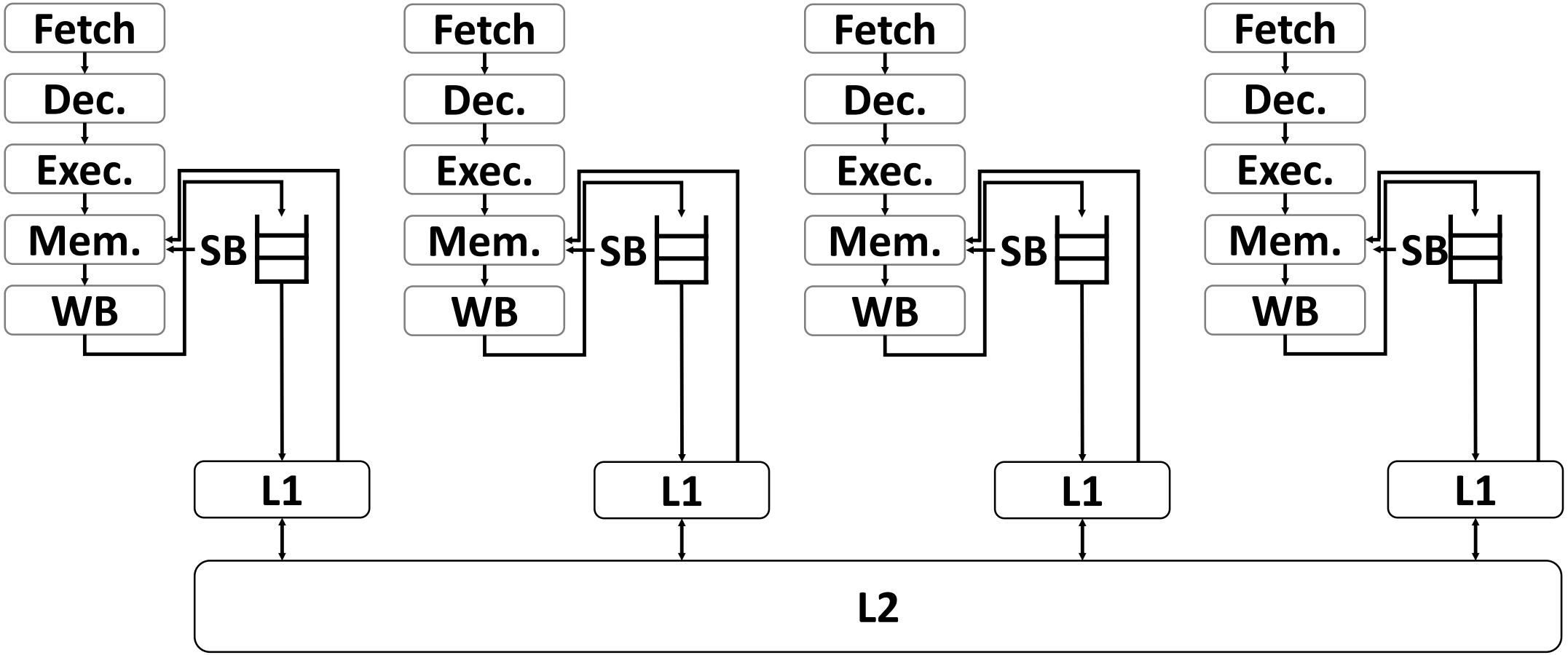
Results: Percentage of Proven Properties

- `Full_Proof` config generally better (90%/test) than `Hybrid` (81%/test)



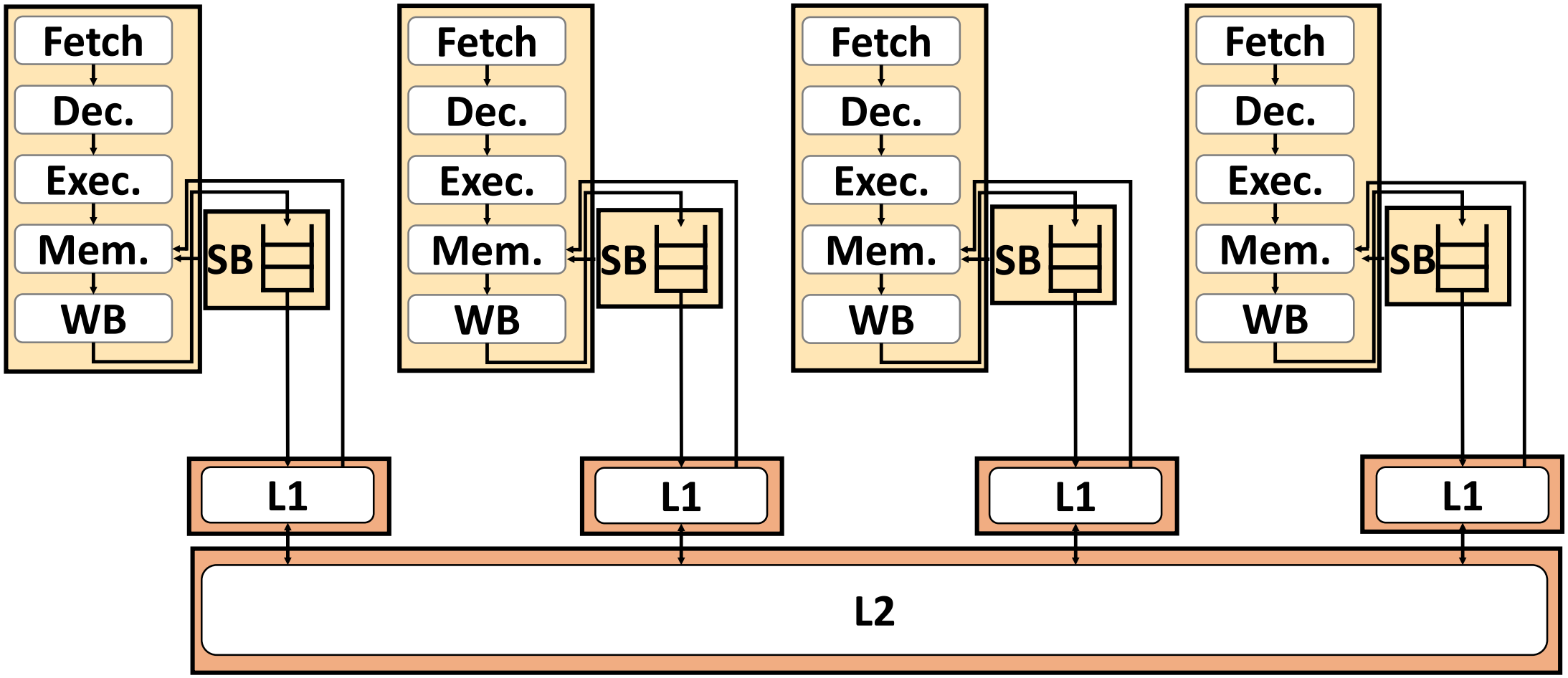
What about larger designs?

Modular MCM Verification: Scalable Analysis



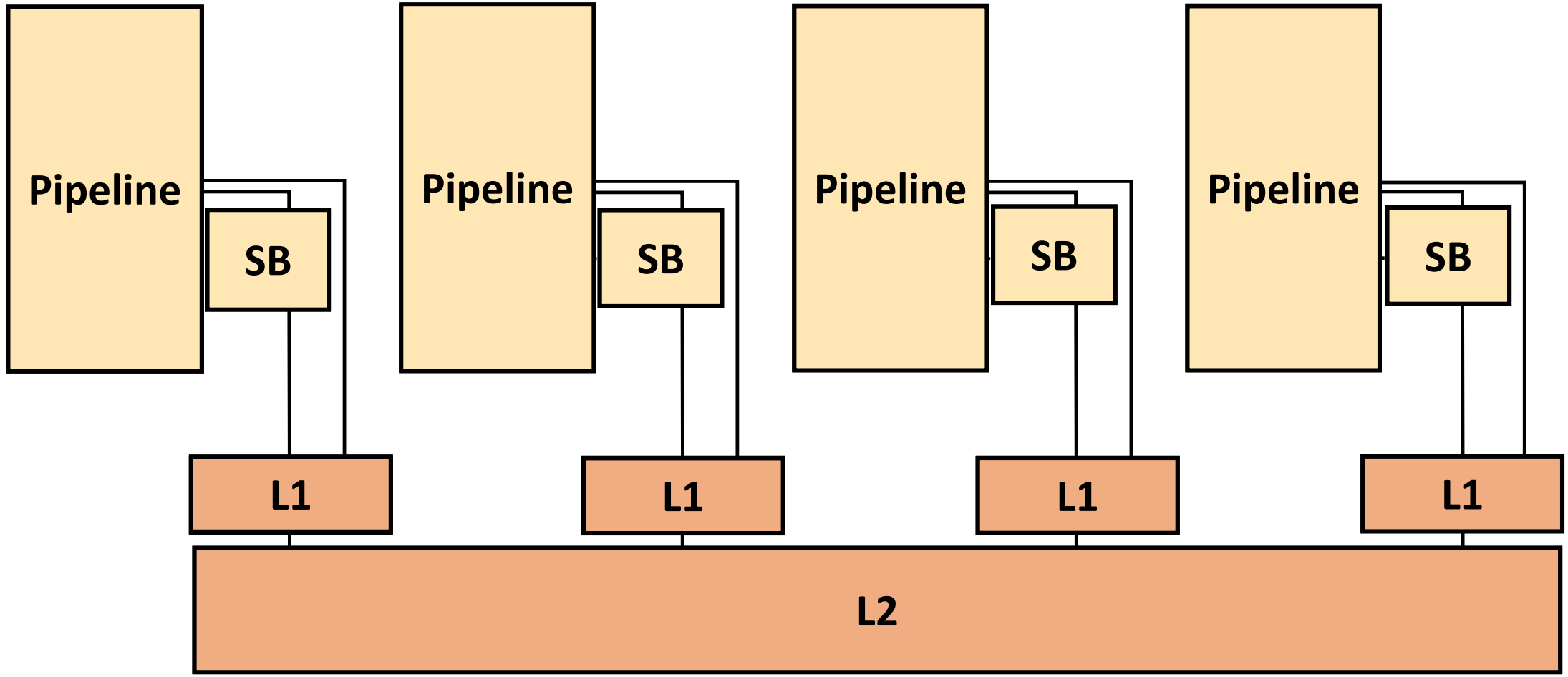
- Verify modules, compose together hierarchically
  - Great for early-stage verification!
- Improved **scalability** and handling of **heterogeneity**

Modular MCM Verification: Scalable Analysis



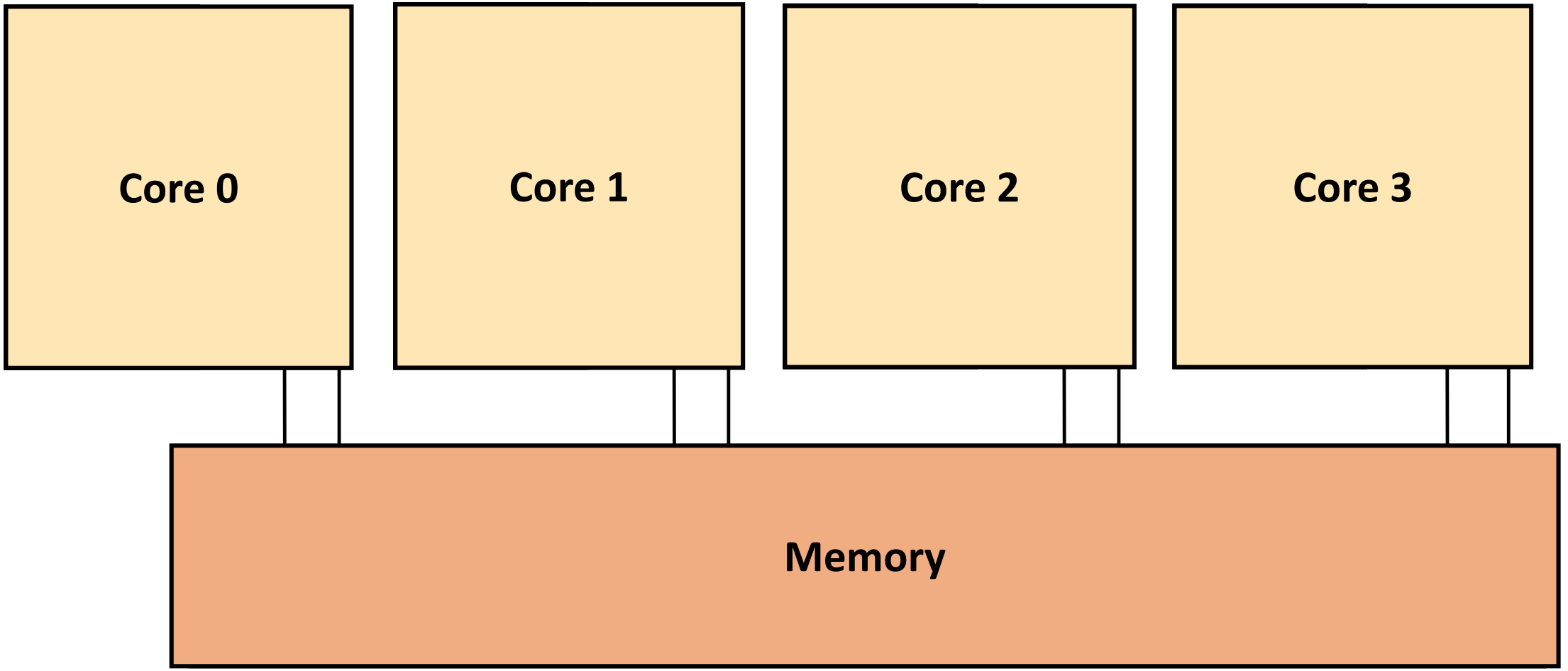
- Verify modules, compose together hierarchically
  - Great for early-stage verification!
- Improved **scalability** and handling of **heterogeneity**

Modular MCM Verification: Scalable Analysis



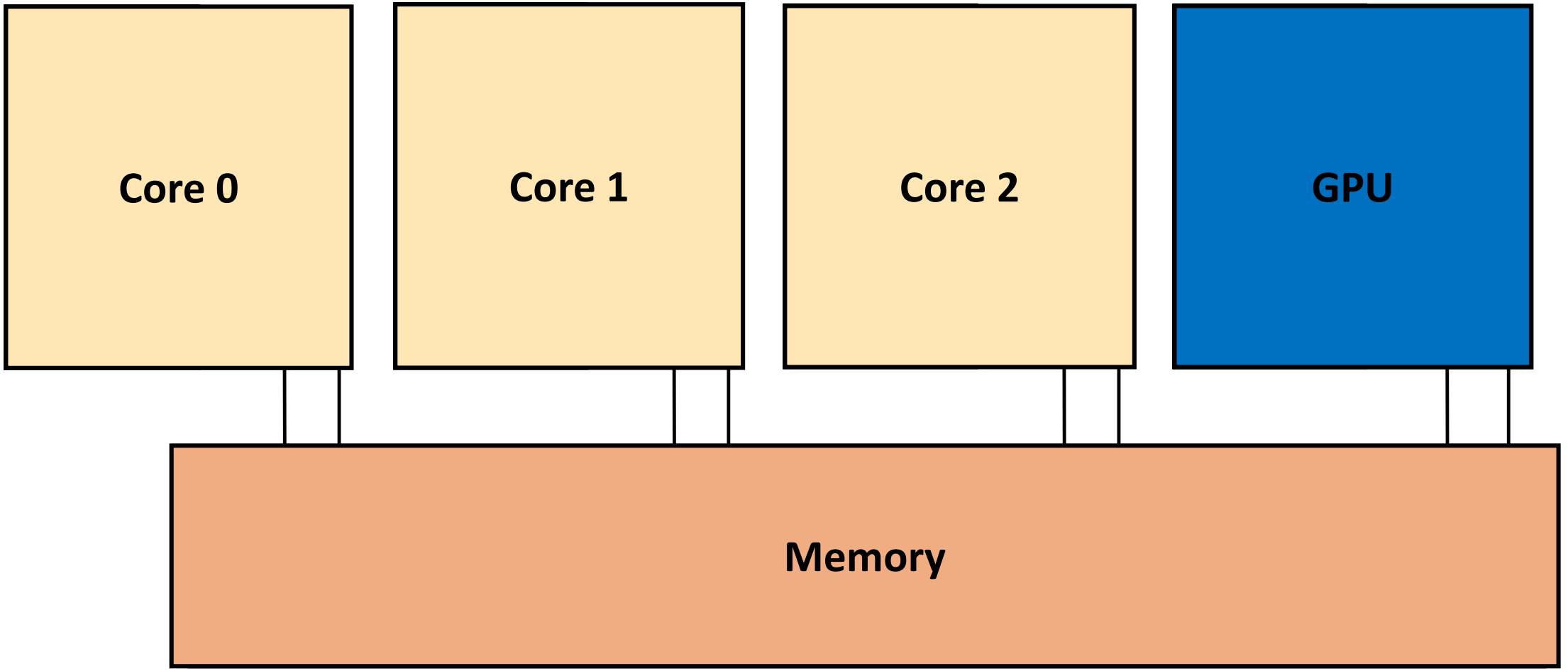
- Verify modules, compose together hierarchically
  - Great for early-stage verification!
- Improved **scalability** and handling of **heterogeneity**

Modular MCM Verification: Scalable Analysis



- Verify modules, compose together hierarchically
  - Great for early-stage verification!
- Improved **scalability** and handling of **heterogeneity**

Modular MCM Verification: Scalable Analysis



- Verify modules, compose together hierarchically
  - Great for early-stage verification!
- Improved **scalability** and handling of **heterogeneity**

RTLCheck Takeaways

- **First automated RTL MCM verification** for litmus test suites
  - Engineers can check MCM properties of their RTL themselves
  - Compatible with existing industry flows and tools
- Novel algorithms to translate **μspec** axioms to temporal **SVA** properties
- **Discovered bug** in memory implementation of RISC-V V-scale processor
- Open-source and available at https://github.com/ymanerka/rtlcheck
- Ongoing Work: Modular MCM Verification for Scalable Analysis
- Accolades:
  - "Honorable Mention" from 2017 *Top Picks of Comp. Arch. Conferences*