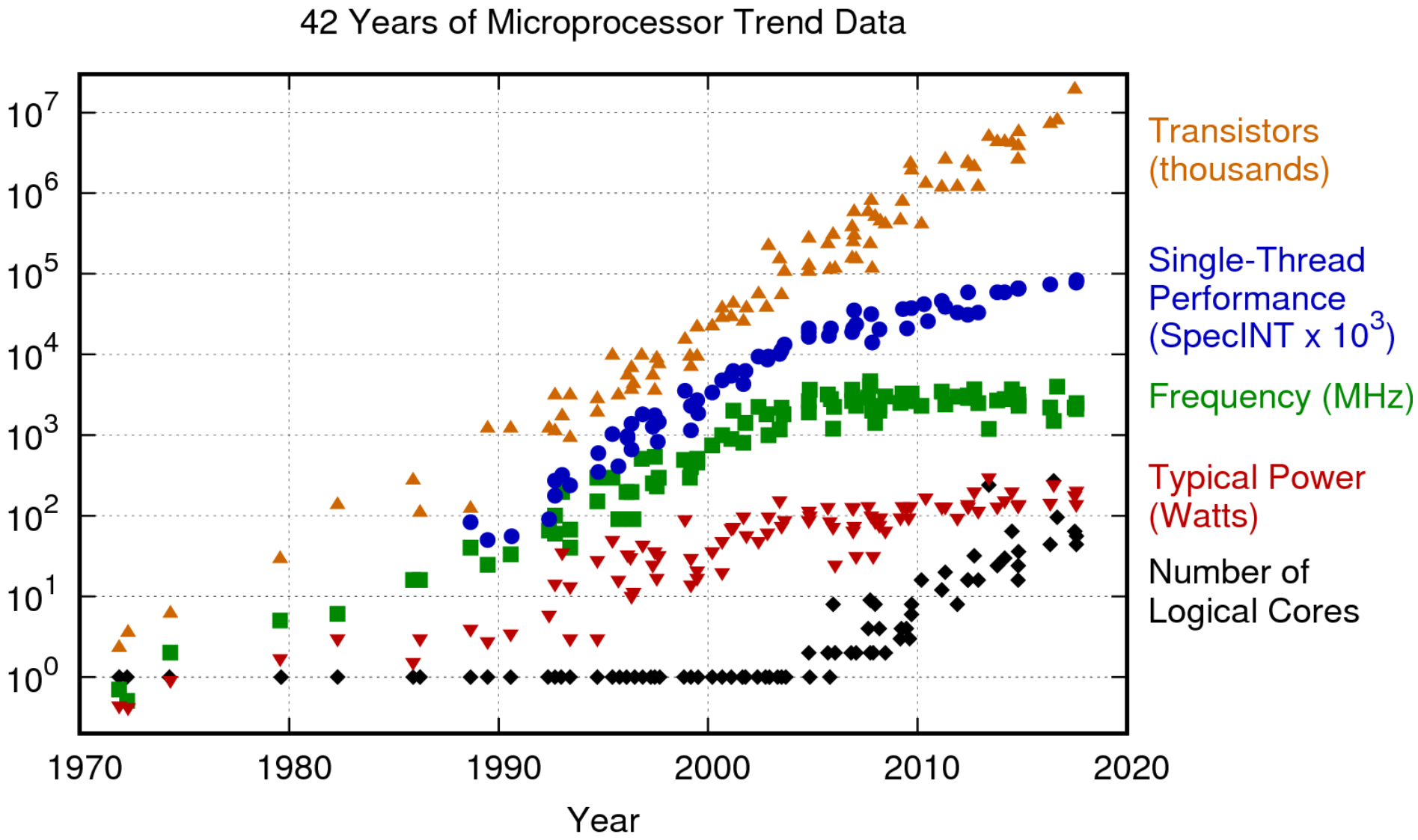
Automated Formal Memory Consistency Verification of Hardware

Yatin A. Manerkar

Princeton University

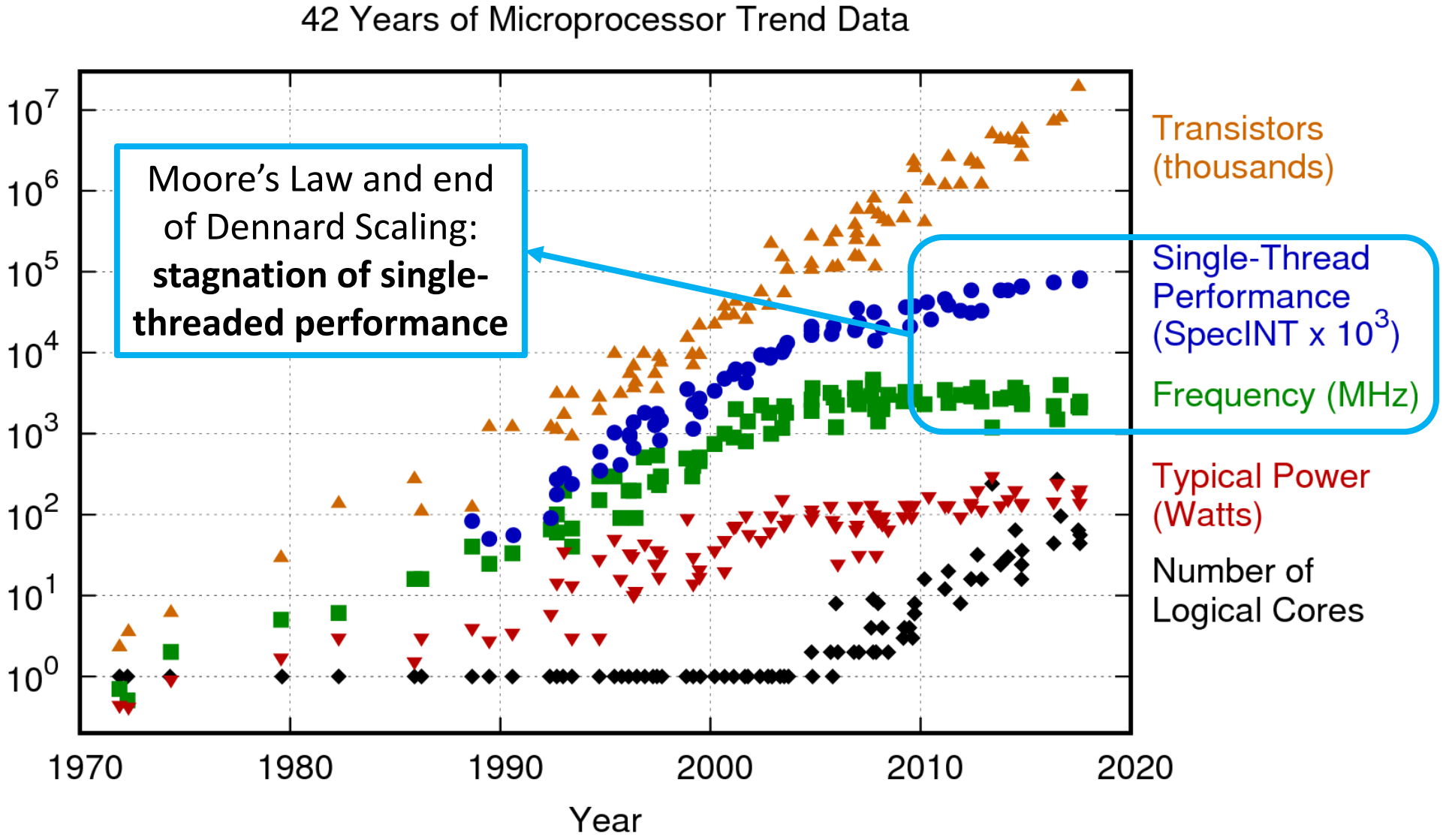
June 23rd, 2019

The Rise of Parallelism…



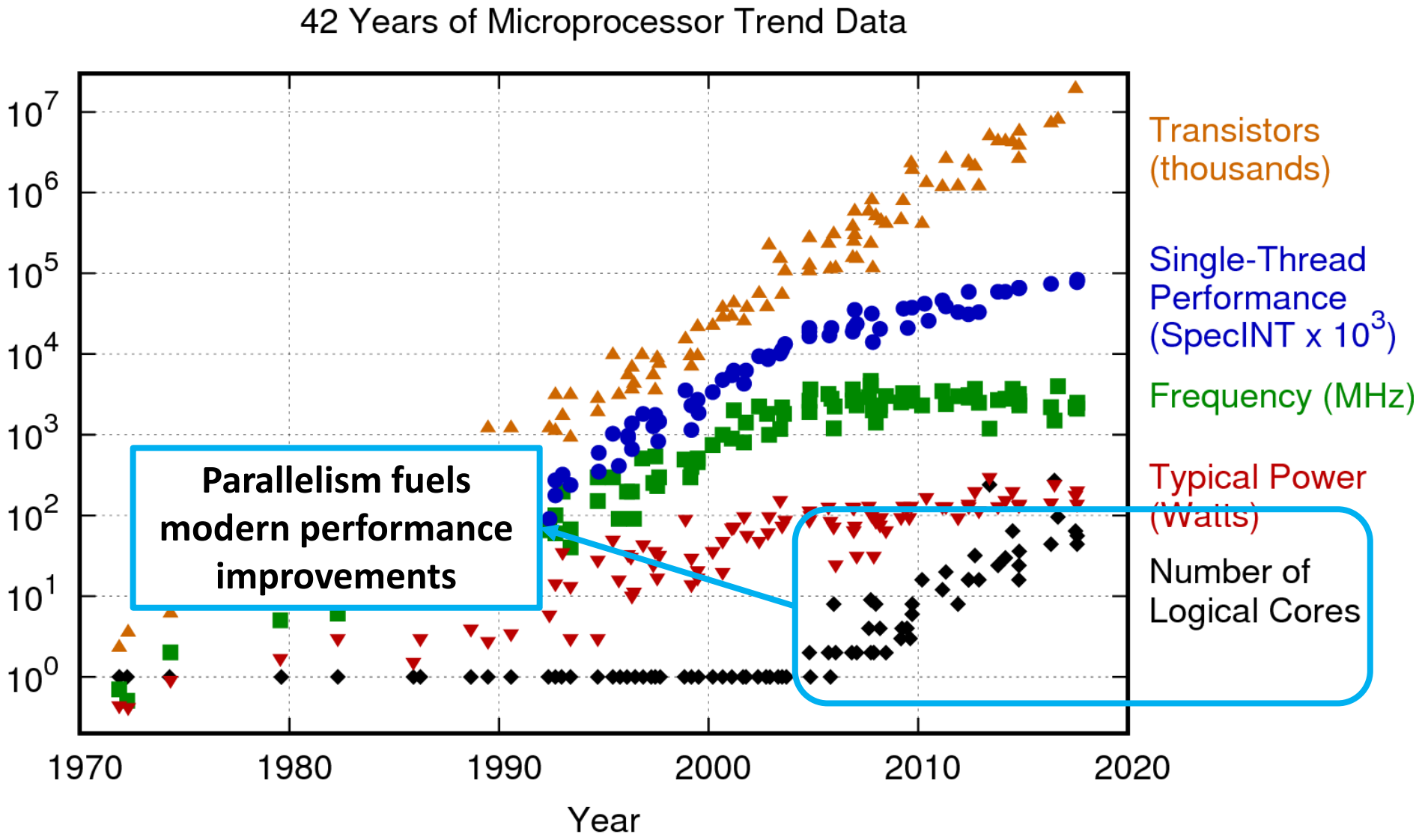
[Image: K. Rupp, M. Horowitz et al.]

The Rise of Parallelism…



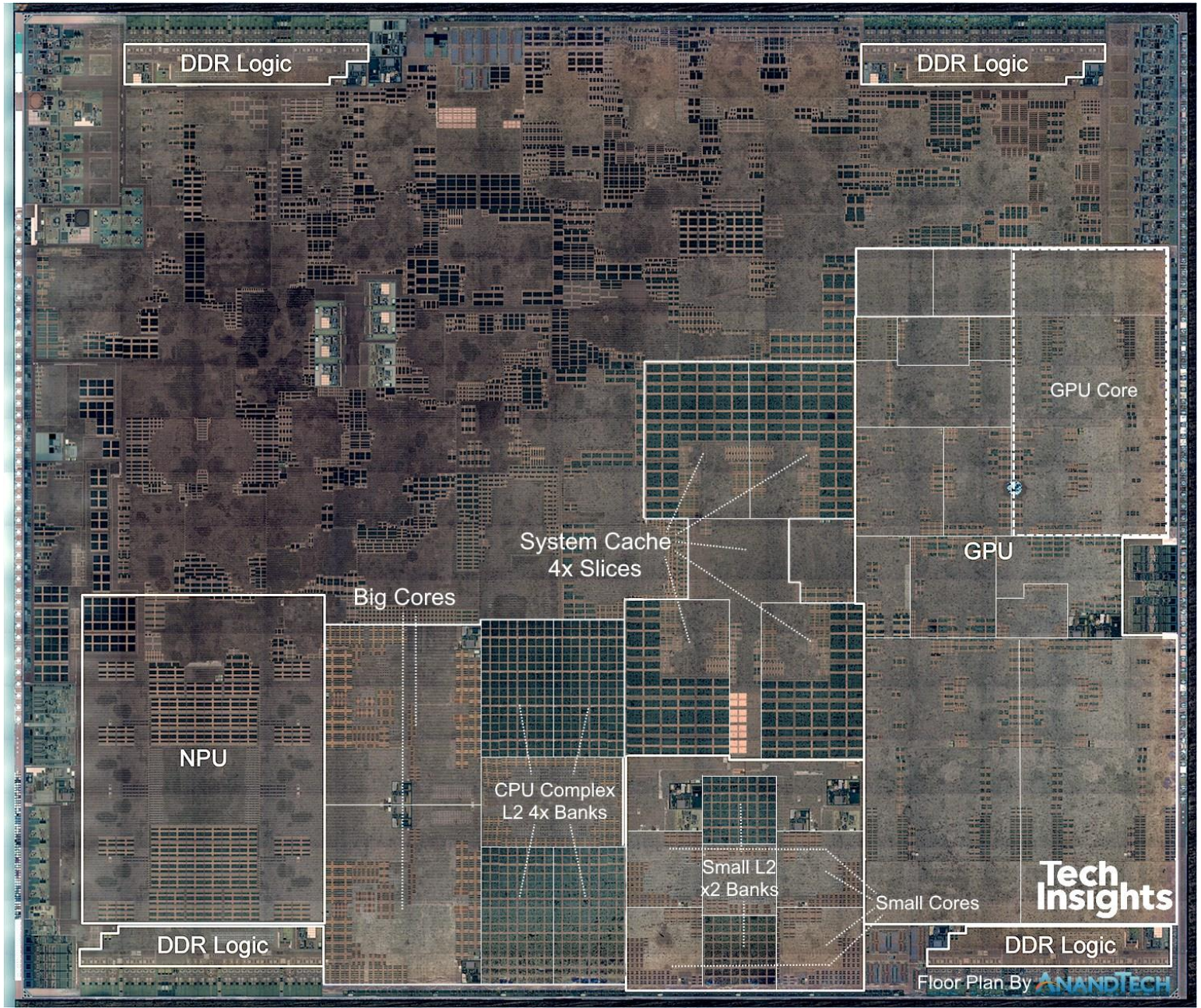
[Image: K. Rupp, M. Horowitz et al.]

The Rise of Parallelism…

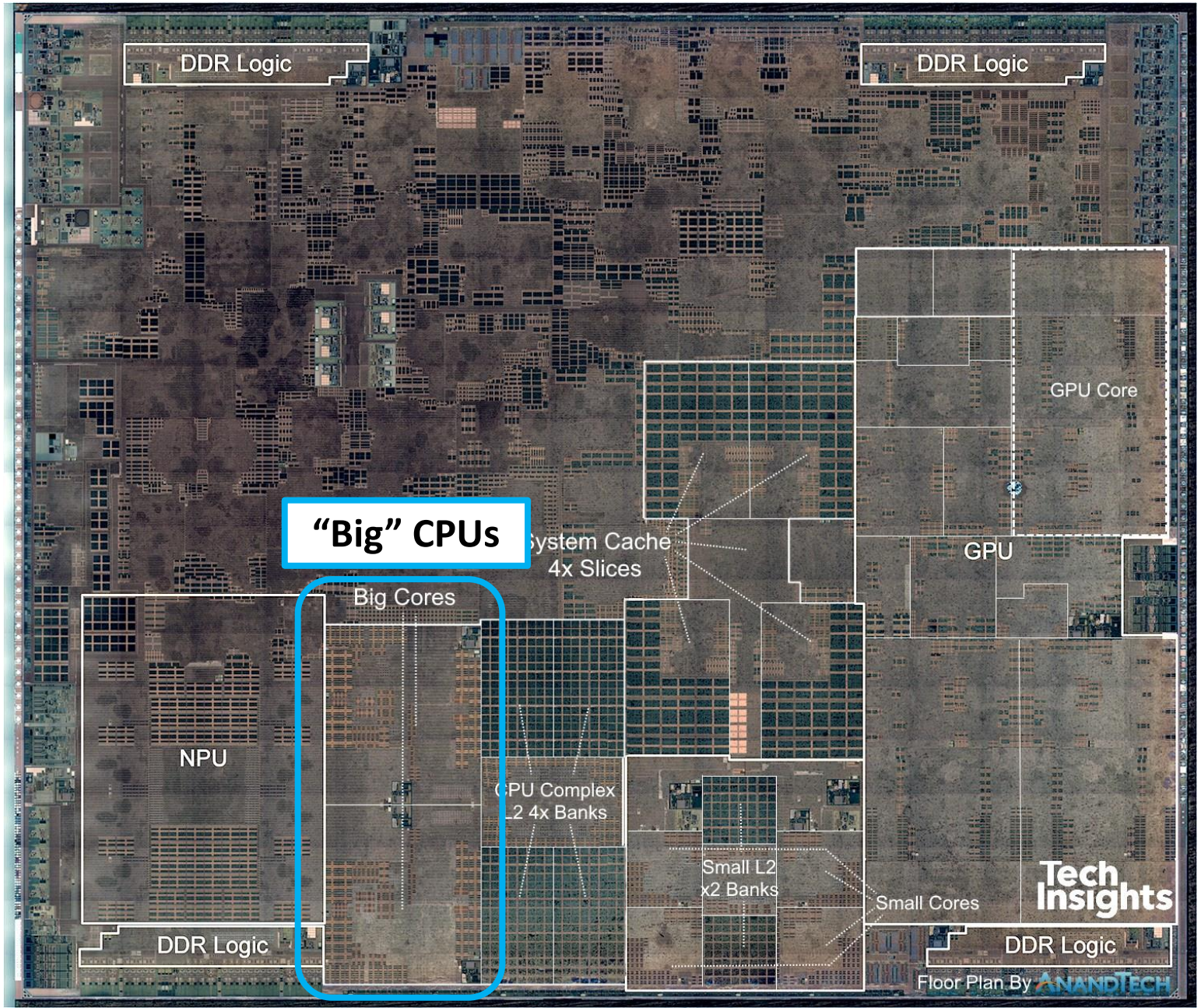


[Image: K. Rupp, M. Horowitz et al.]

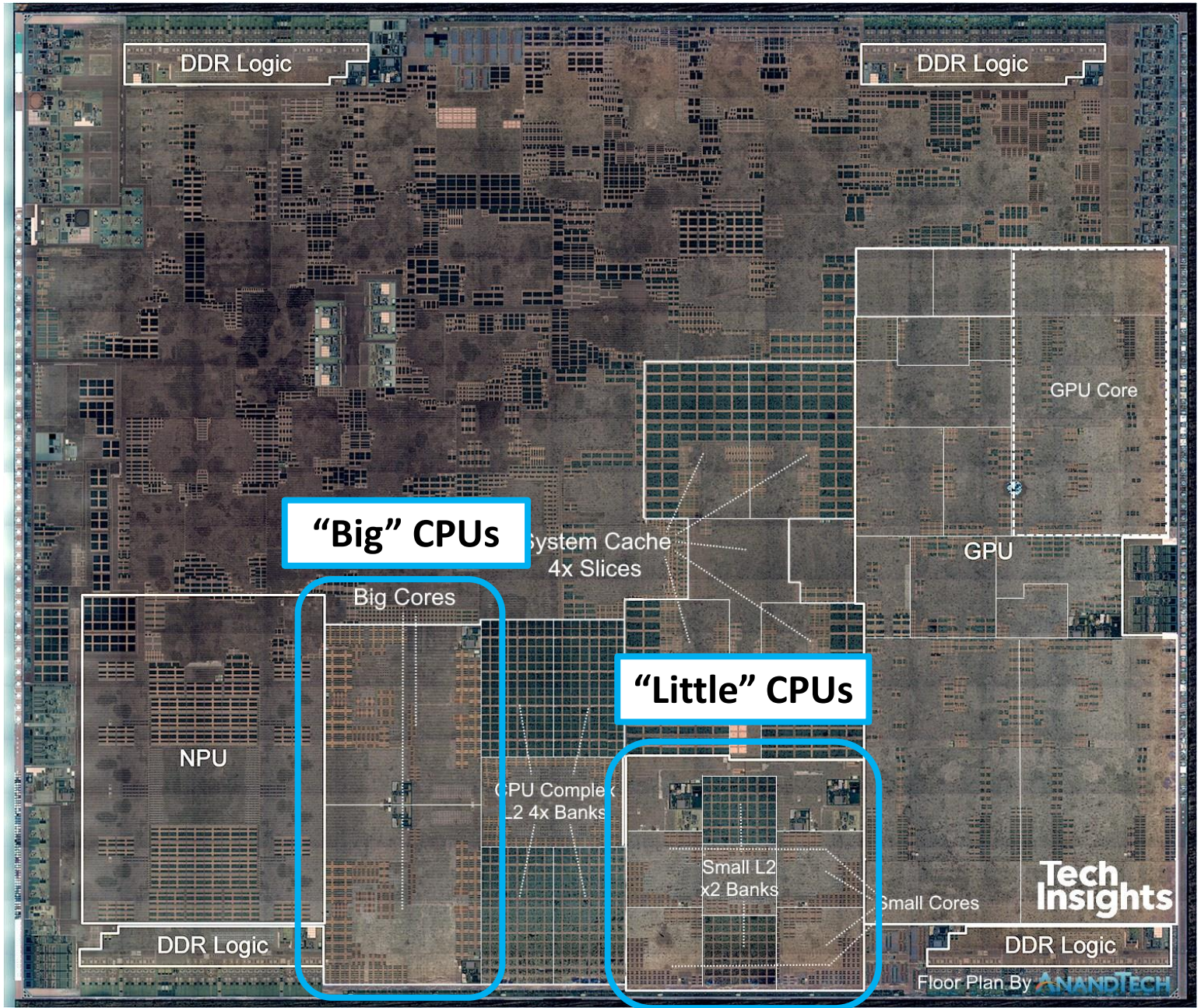
…and Heterogeneity (Example: Apple A12)



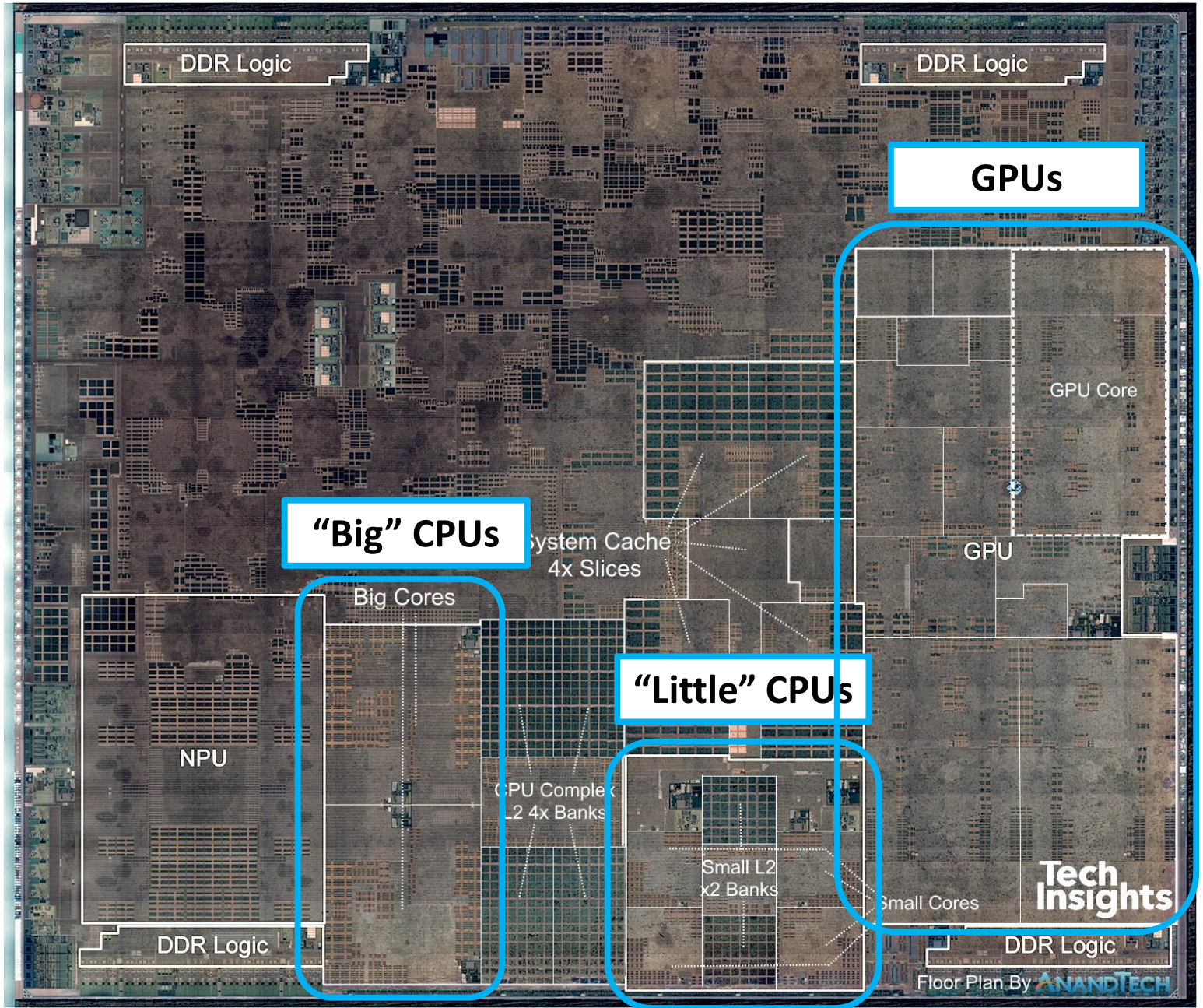
…and Heterogeneity (Example: Apple A12)



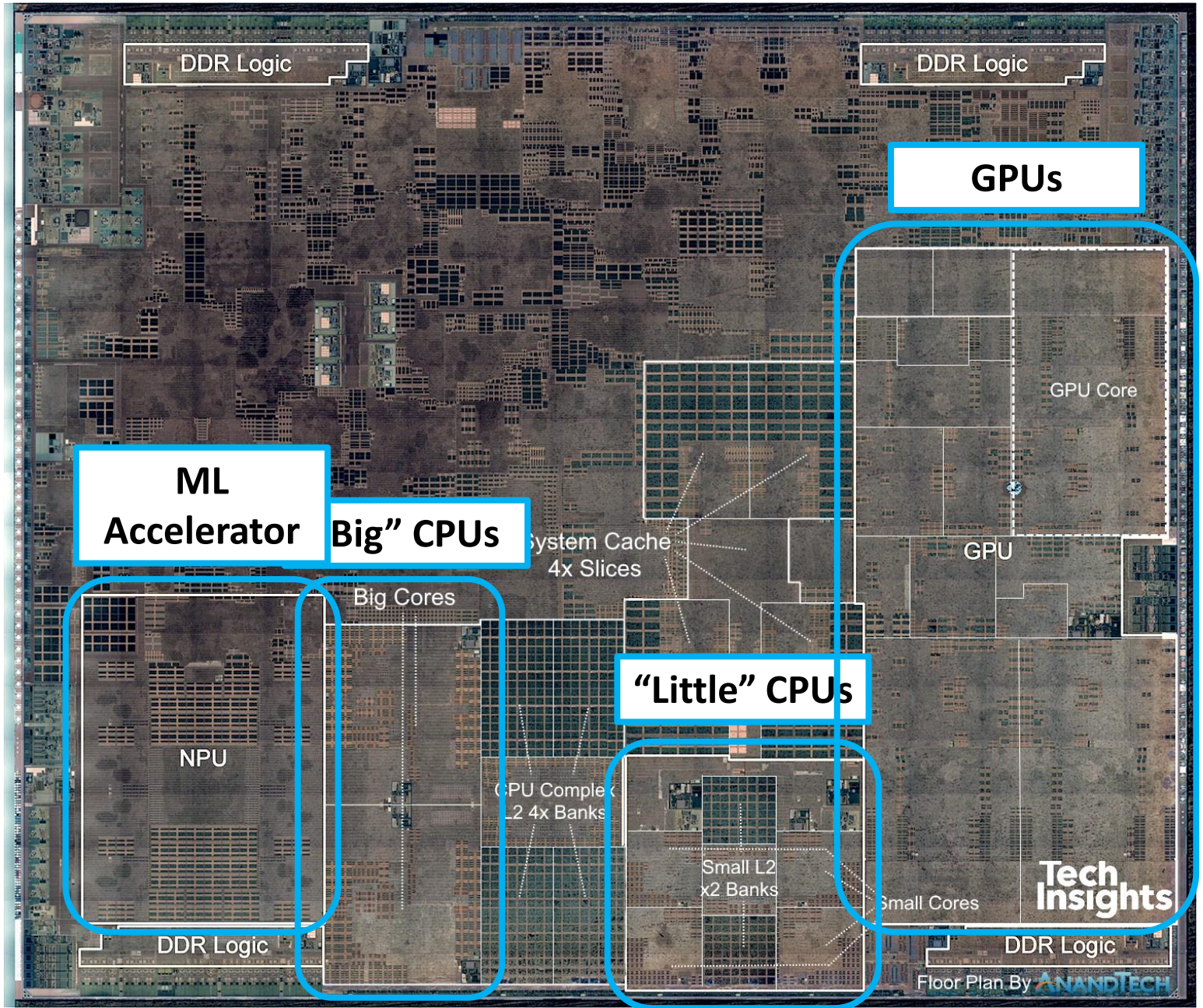
…and Heterogeneity (Example: Apple A12)



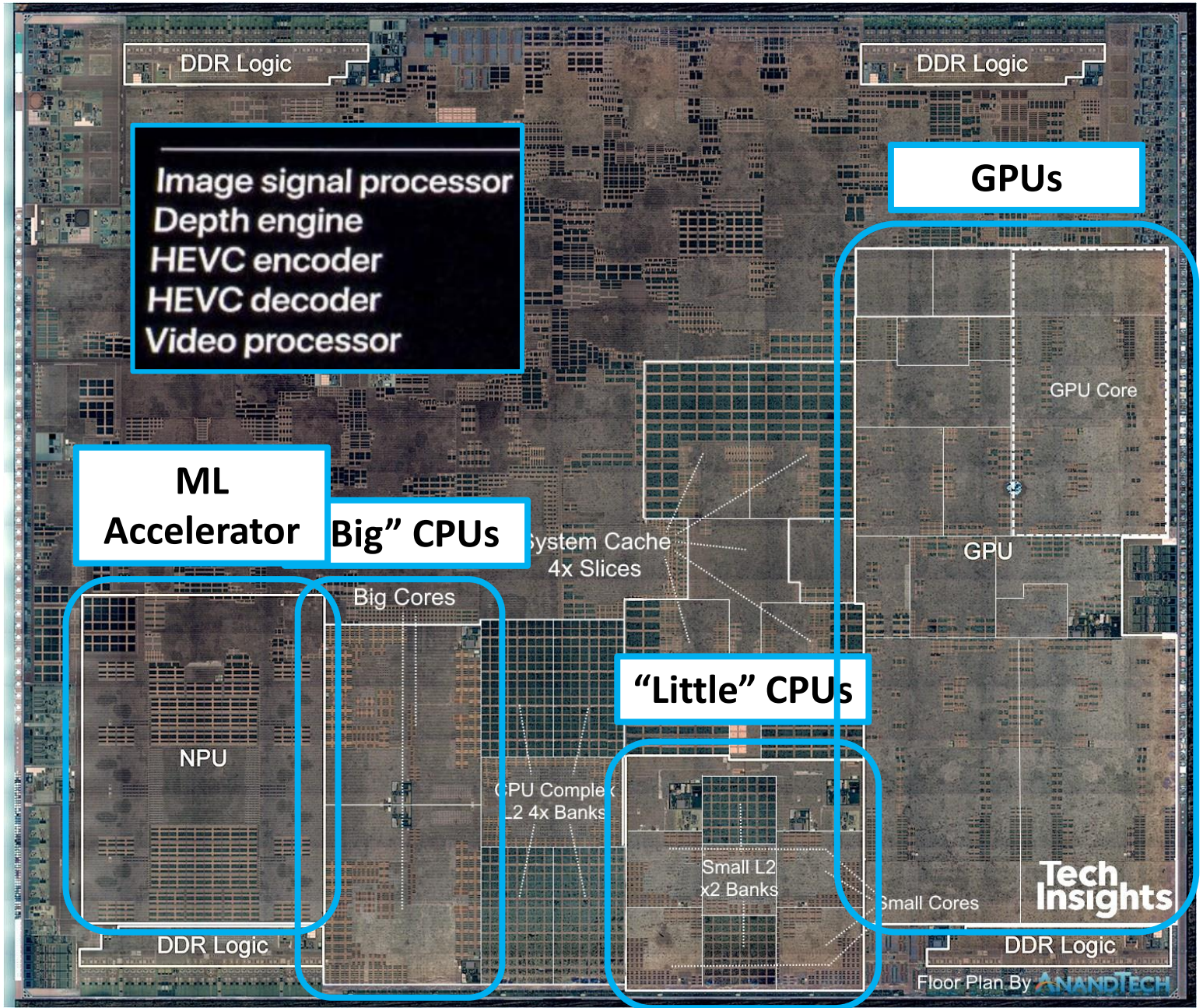
…and Heterogeneity (Example: Apple A12)



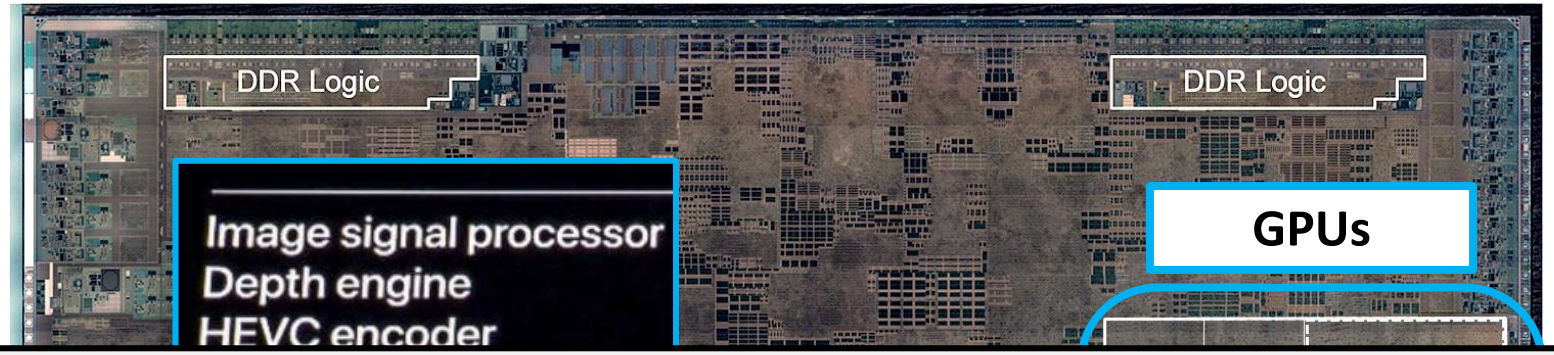
…and Heterogeneity (Example: Apple A12)



…and Heterogeneity (Example: Apple A12)

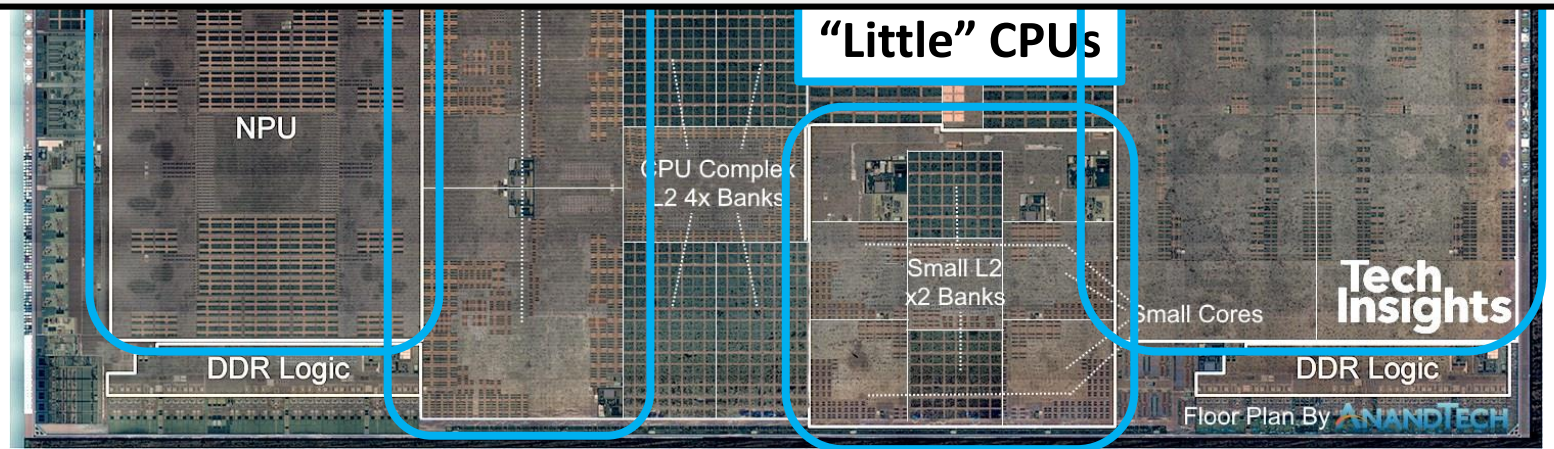


…and Heterogeneity (Example: Apple A12)



Parallel processors are hard to get right!

How can we formally verify parallel hardware?



Building a Formally Verified Processor

Formal Methods Expert

- Build proven-correct processor (e.g. Kami) or…

Building a Formally Verified Processor

Formal Methods Expert

REMS

- Build proven-correct processor (e.g. Kami) or…
- …construct formal model of implementation and verify that (REMS)

Building a Formally Verified Processor

Formal Methods Expert

REMS

- Build proven-correct processor (e.g. Kami) or…
- …construct formal model of implementation and verify that (REMS)
- **Formal methods expert carries most of the verification burden**

Building a Formally Verified Processor

Formal Methods Expert

REMS

- Build proven-correct processor (e.g. Kami) or…
- …construct formal model of implementation and verify that (REMS)
- **Formal methods expert carries most of the verification burden**

Computer Architect

OpenPiton

- Experts on building processors
- Generally not much formal methods expertise
- **Can they share more of the verification burden?**

Building a Formally Verified Processor

Formal Methods Expert

Computer Architect

My work: Automated tools that enable engineers to formally verify their systems by themselves!

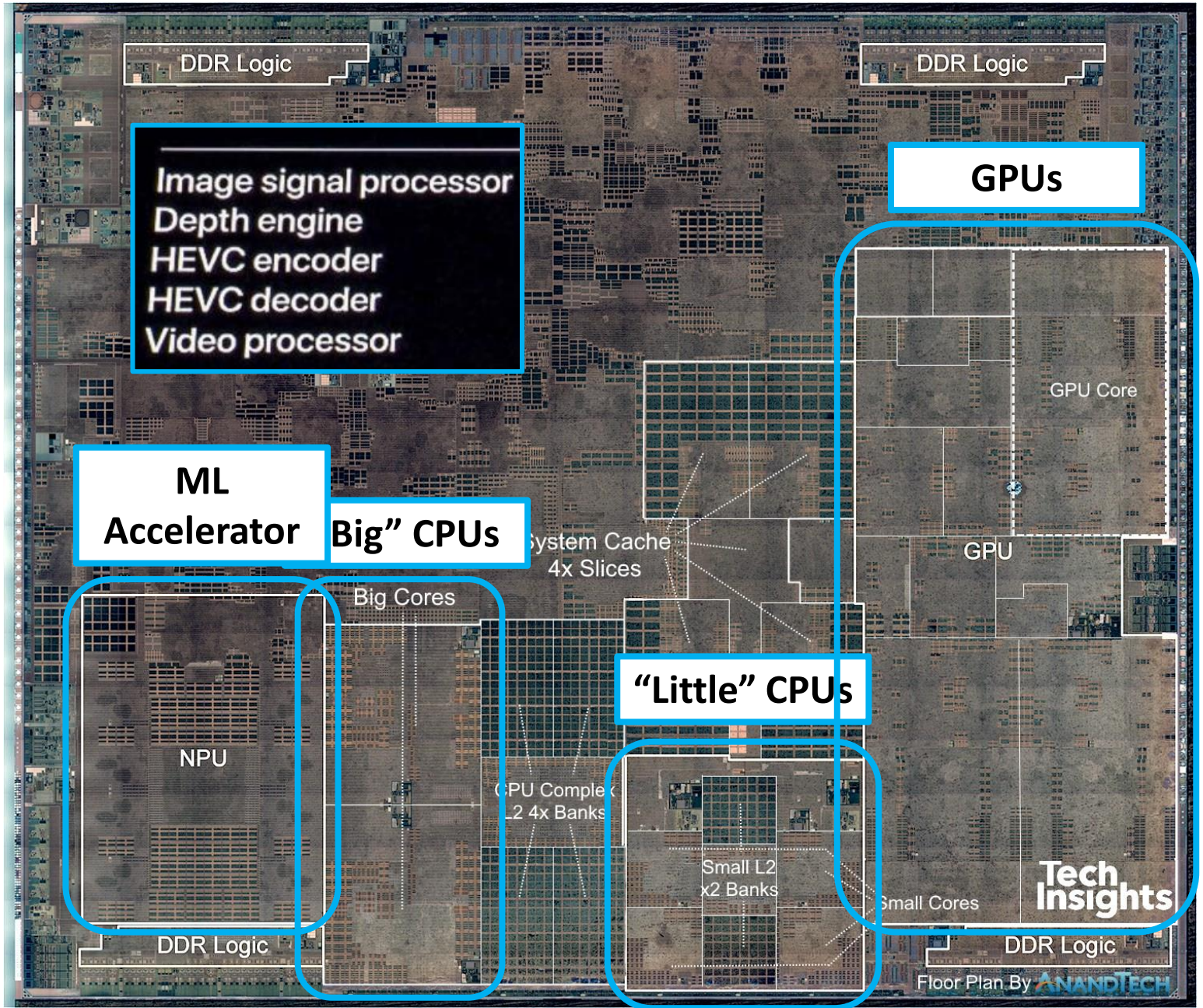
Case Study: Memory Consistency Verification

- Formal methods expert carries most of the verification burden
- Can they share more of the verification burden?

Talk Outline

- Overview
- Memory Consistency Background
- **PipeProof:** All-Program Microarchitectural MCM Verification
- **RTLCheck:** MCM Verification of Verilog RTL
- Expanding to other domains
- Conclusion

Processors Communicate via Shared Memory



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”**? **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”**? **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**? **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)` `  print("Answer is:");`

Most processors today implement "weak memory models" that relax orderings required by SC!

NO!

YES!

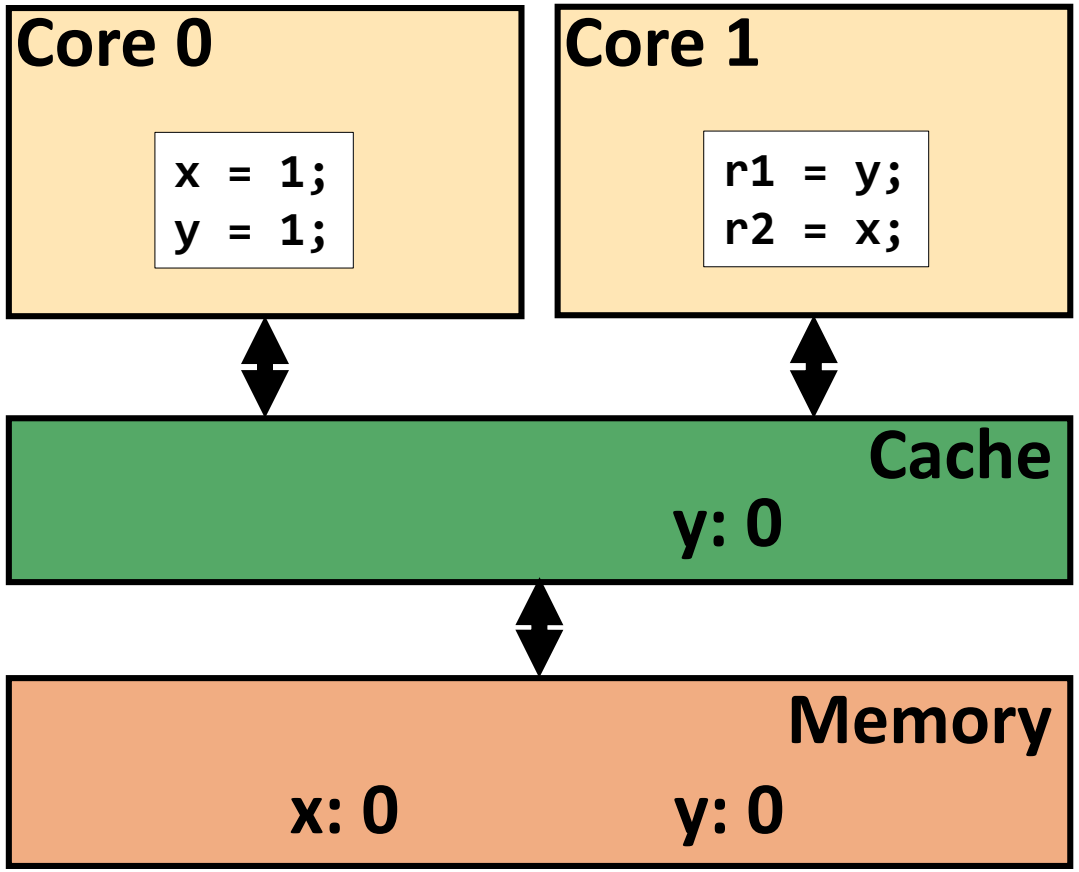
arm

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!

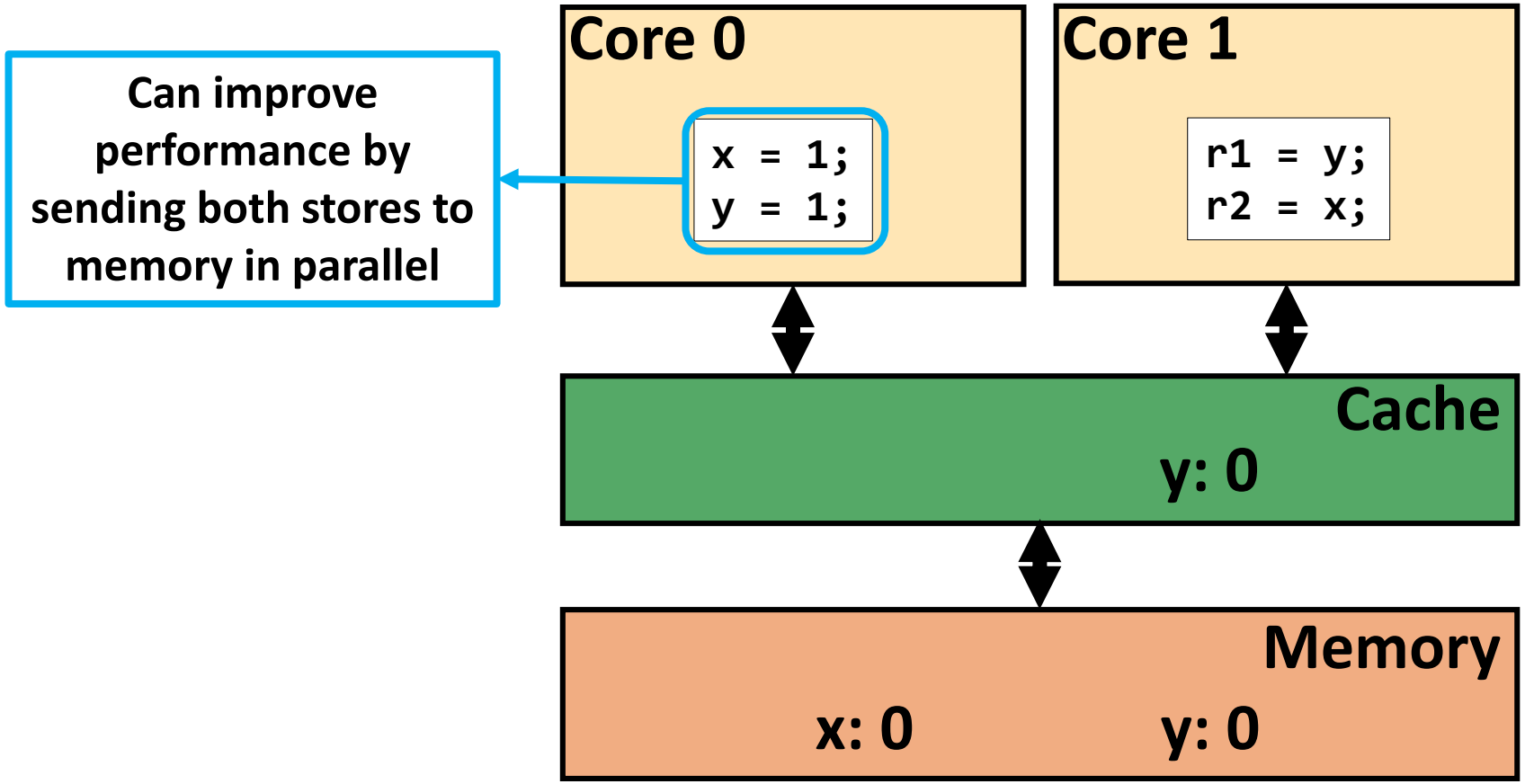


Why reorder memory operations?

Message Passing (mp)

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

Answer: Performance!

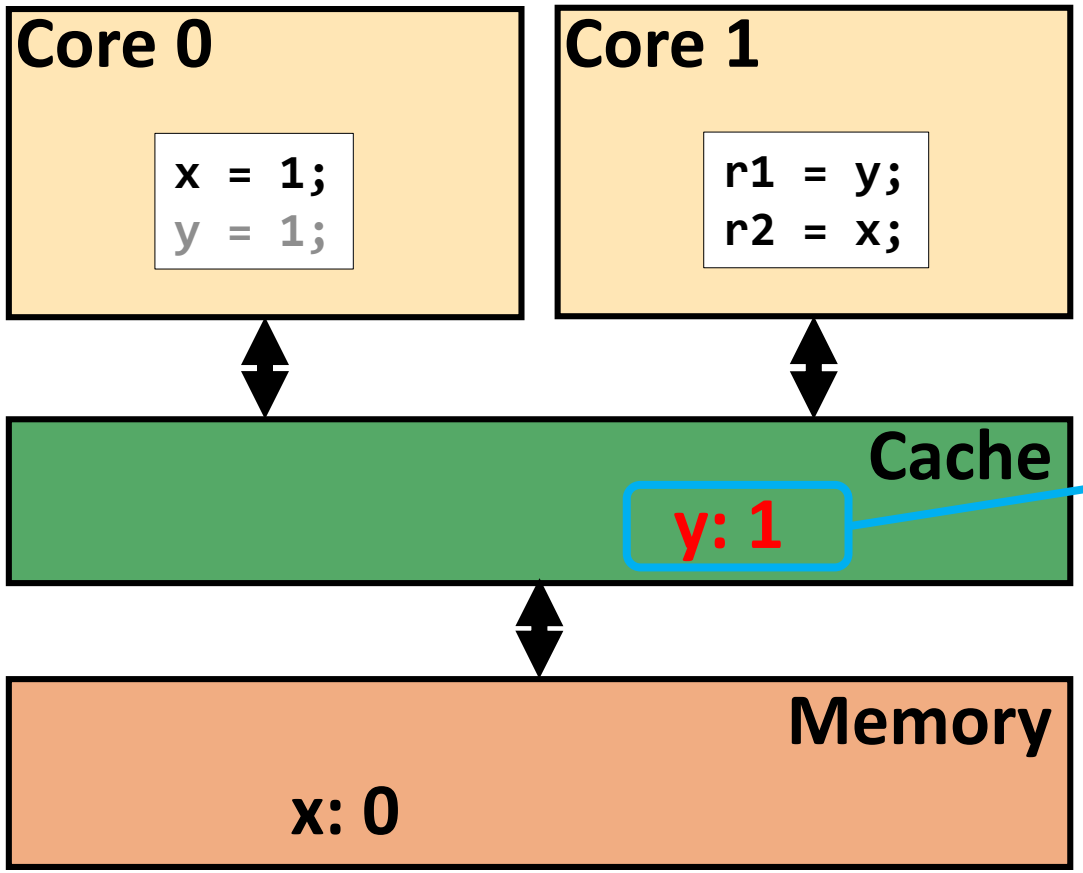


Why reorder memory operations?

Message Passing (mp)

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

Answer: Performance!



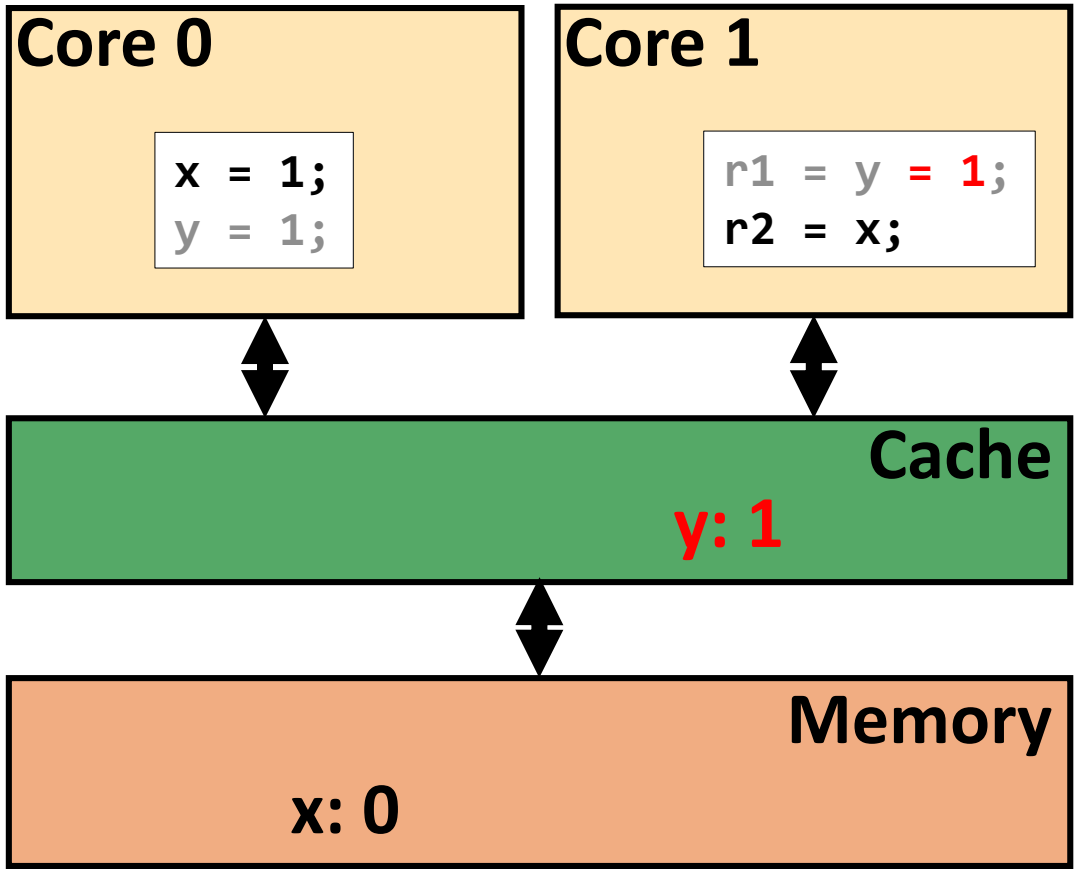
Store to y finishes quickly in cache

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!

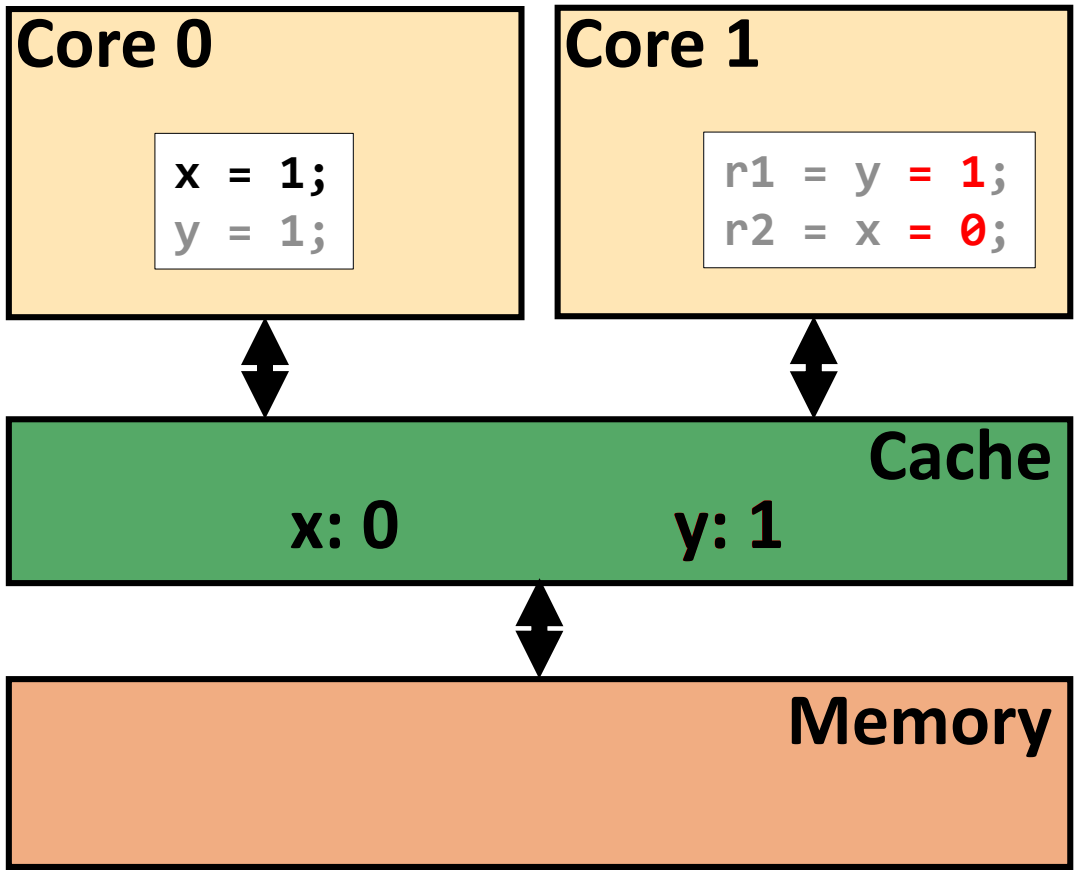


Why reorder memory operations?

Message Passing (mp)

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

Answer: Performance!

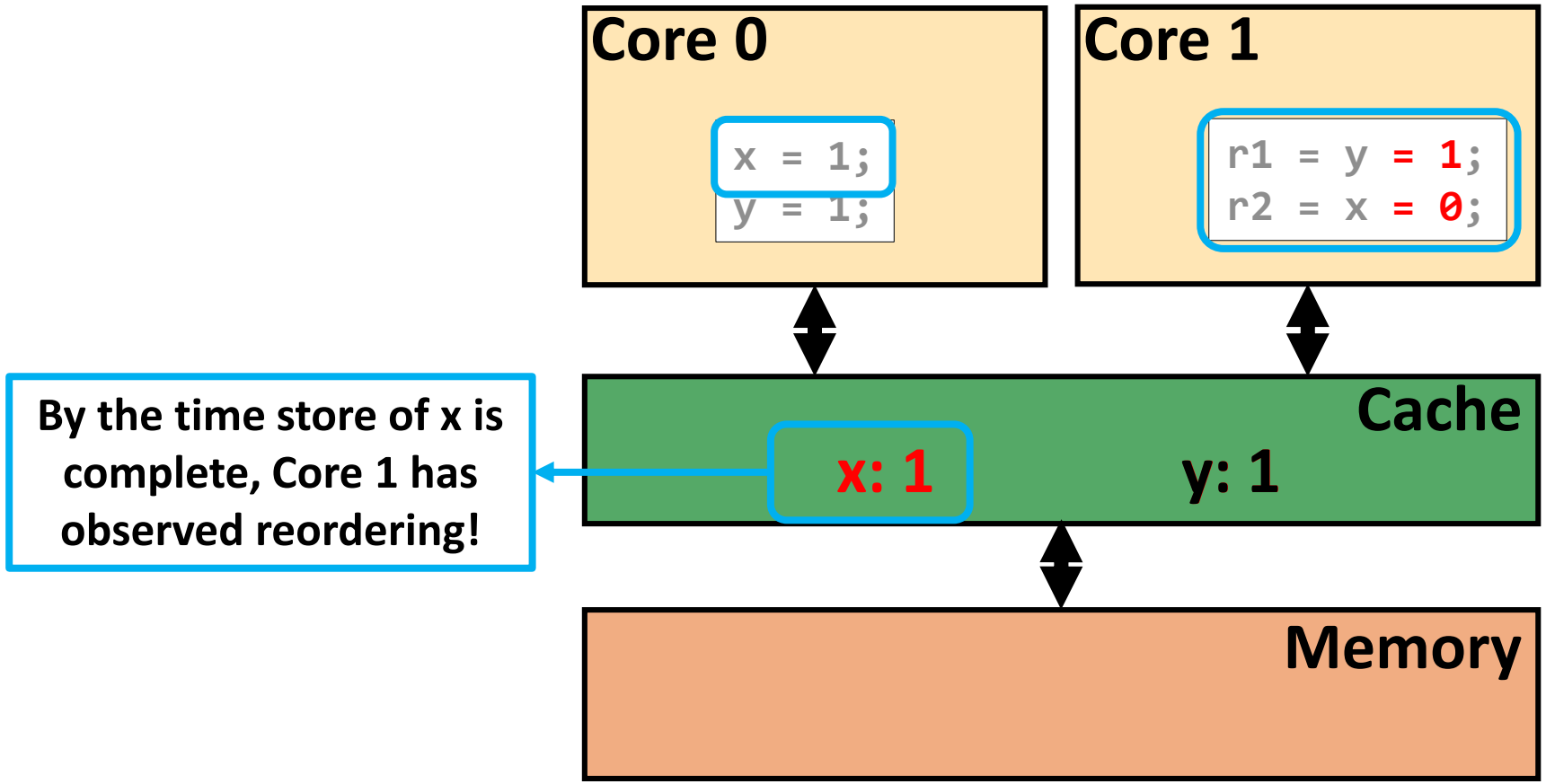


Why reorder memory operations?

Message Passing (mp)

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

Answer: Performance!



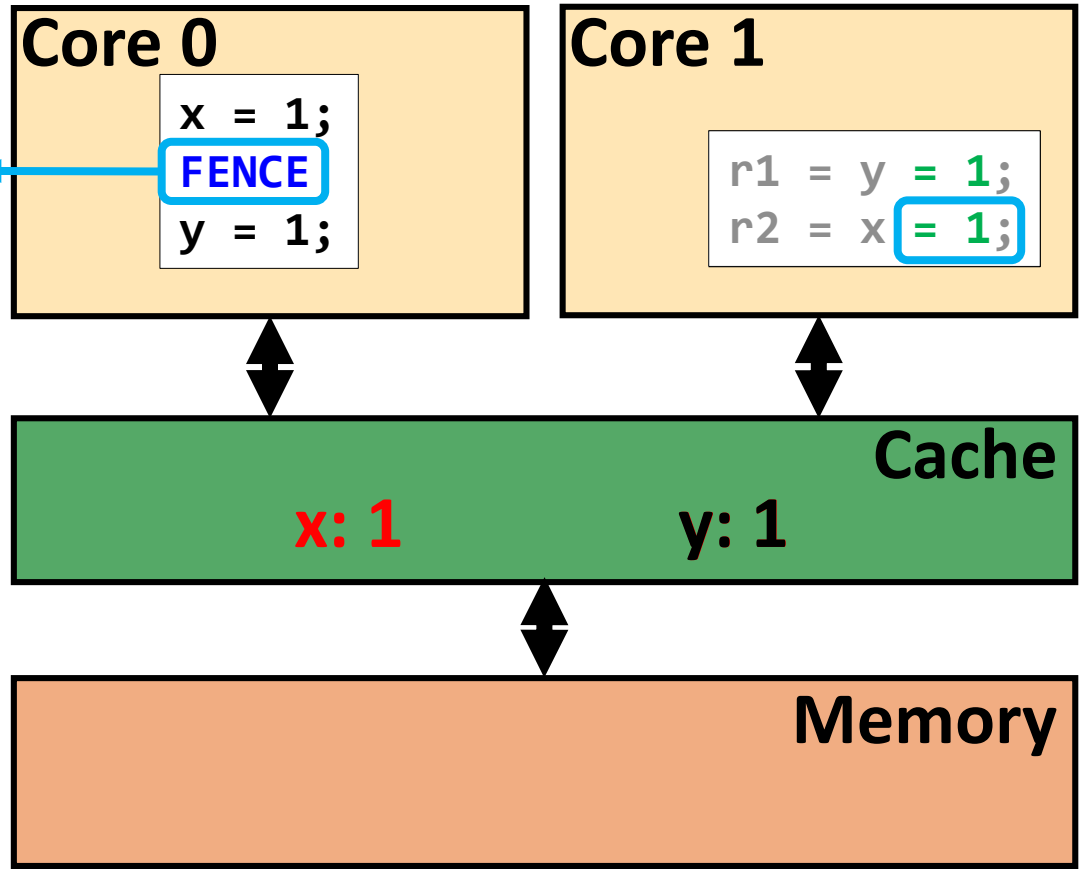
Why reorder memory operations?

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



Memory Consistency Models (MCMs)

- Instruction sets (ISAs) represent hardware operations (add, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Compiler

Hardware

Memory Consistency Models (MCMs)

- Instruction sets (ISAs) represent hardware operations (add, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Where do I need to add fences?

Compiler

Hardware

Memory Consistency Models (MCMs)

- Instruction sets (ISAs) represent hardware operations (add, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Where do I need to add fences?

Compiler

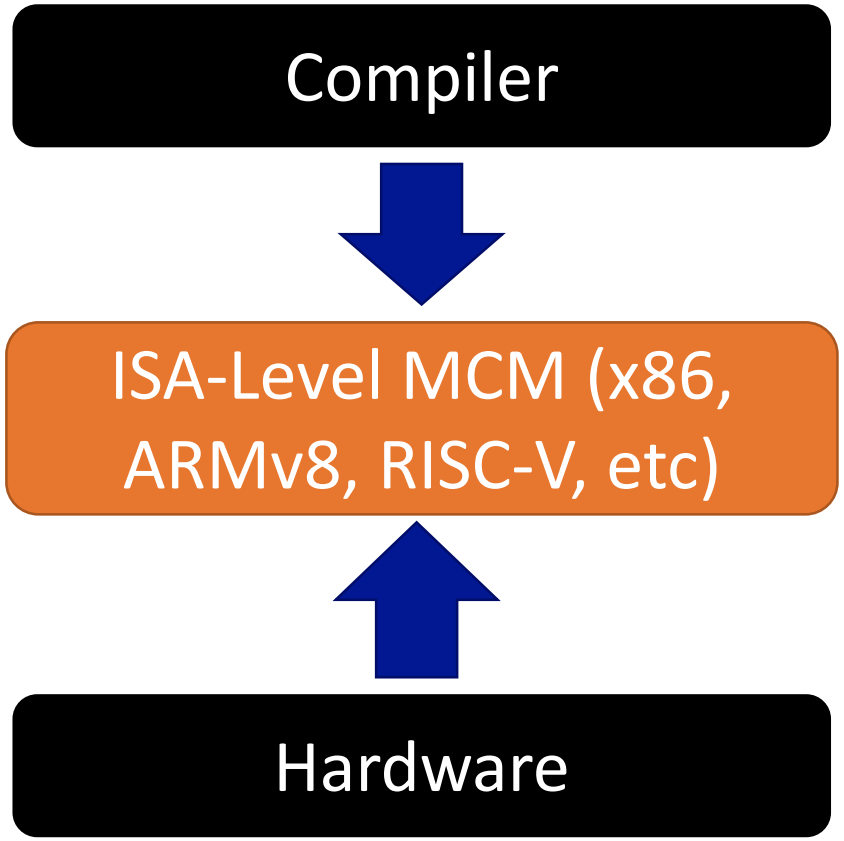
Hardware

How much can I buffer and reorder memory operations?

Memory Consistency Models (MCMs)

- Instruction sets (ISAs) represent hardware operations (add, ld, st, …)
- MCMs similarly represent the orderings among hardware memory ops

Where do I need to add fences?



How much can I buffer and reorder memory operations?

Memory Consistency Models (MCMs)

- Instruction sets (ISAs) represent hardware operations (add, 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!

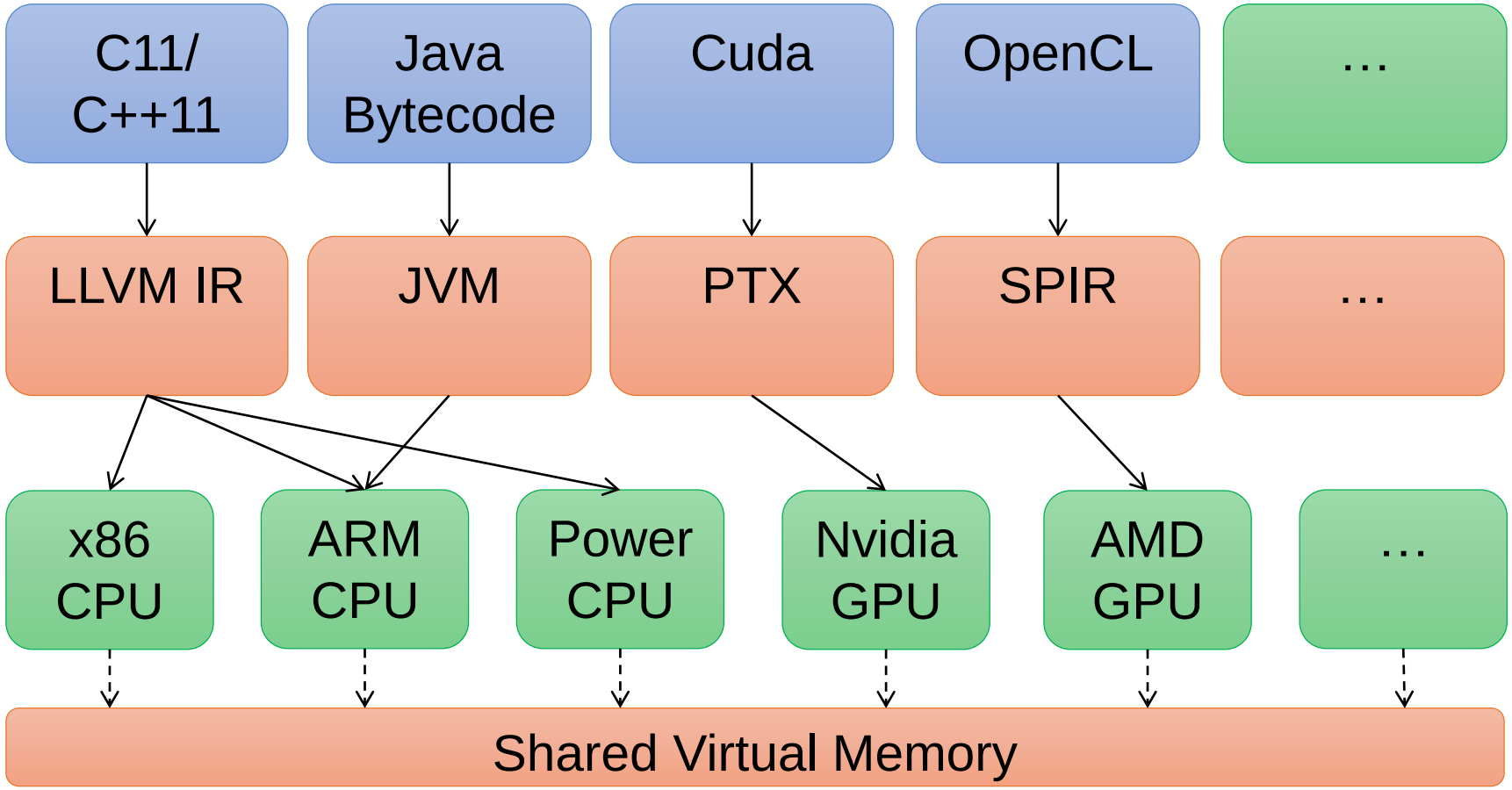
ARMv8, RISC-V, etc)

Hardware

How much can I buffer and reorder memory operations?

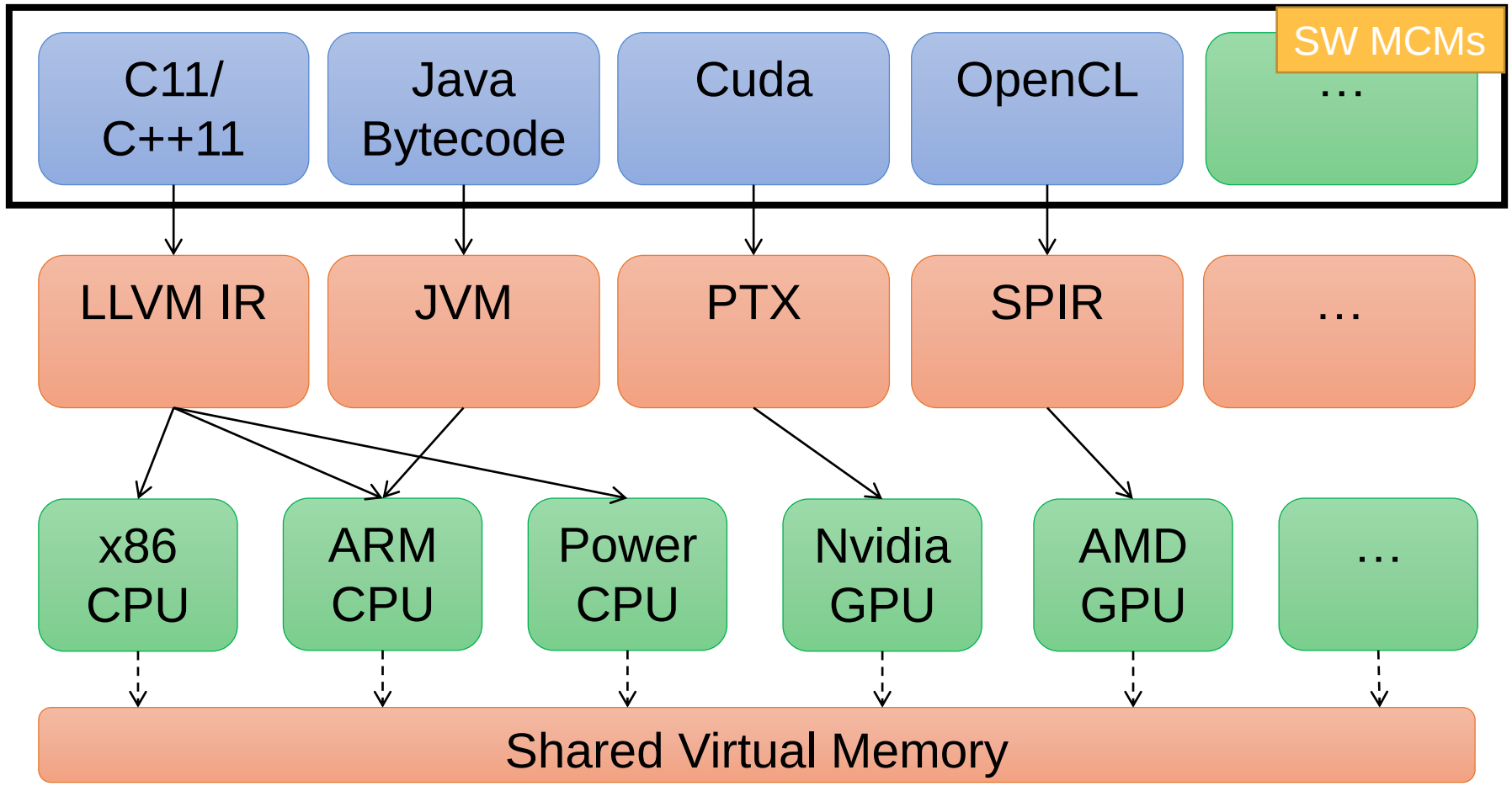
Memory Consistency Models (MCMs)

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



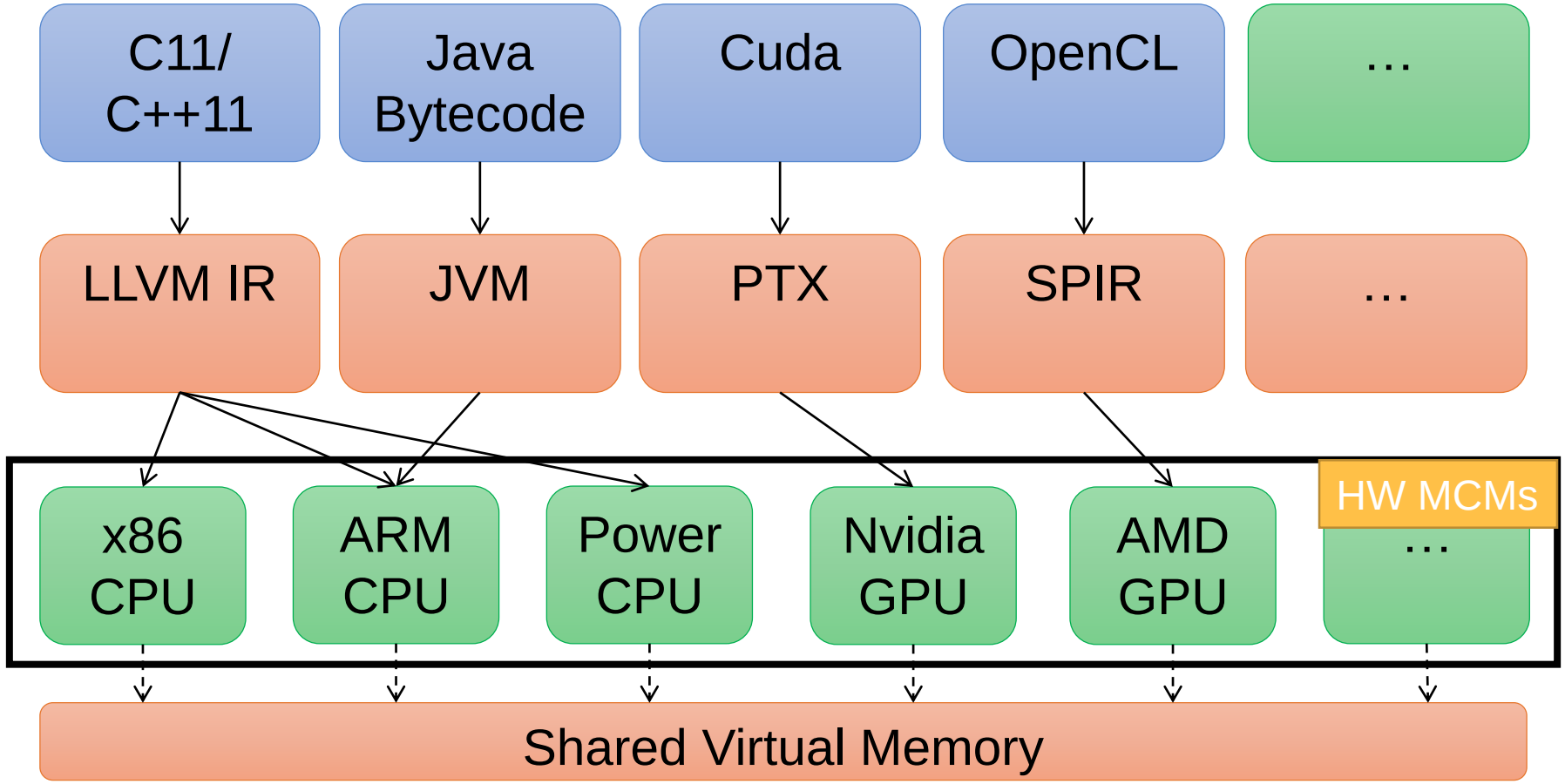
Memory Consistency Models (MCMs)

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



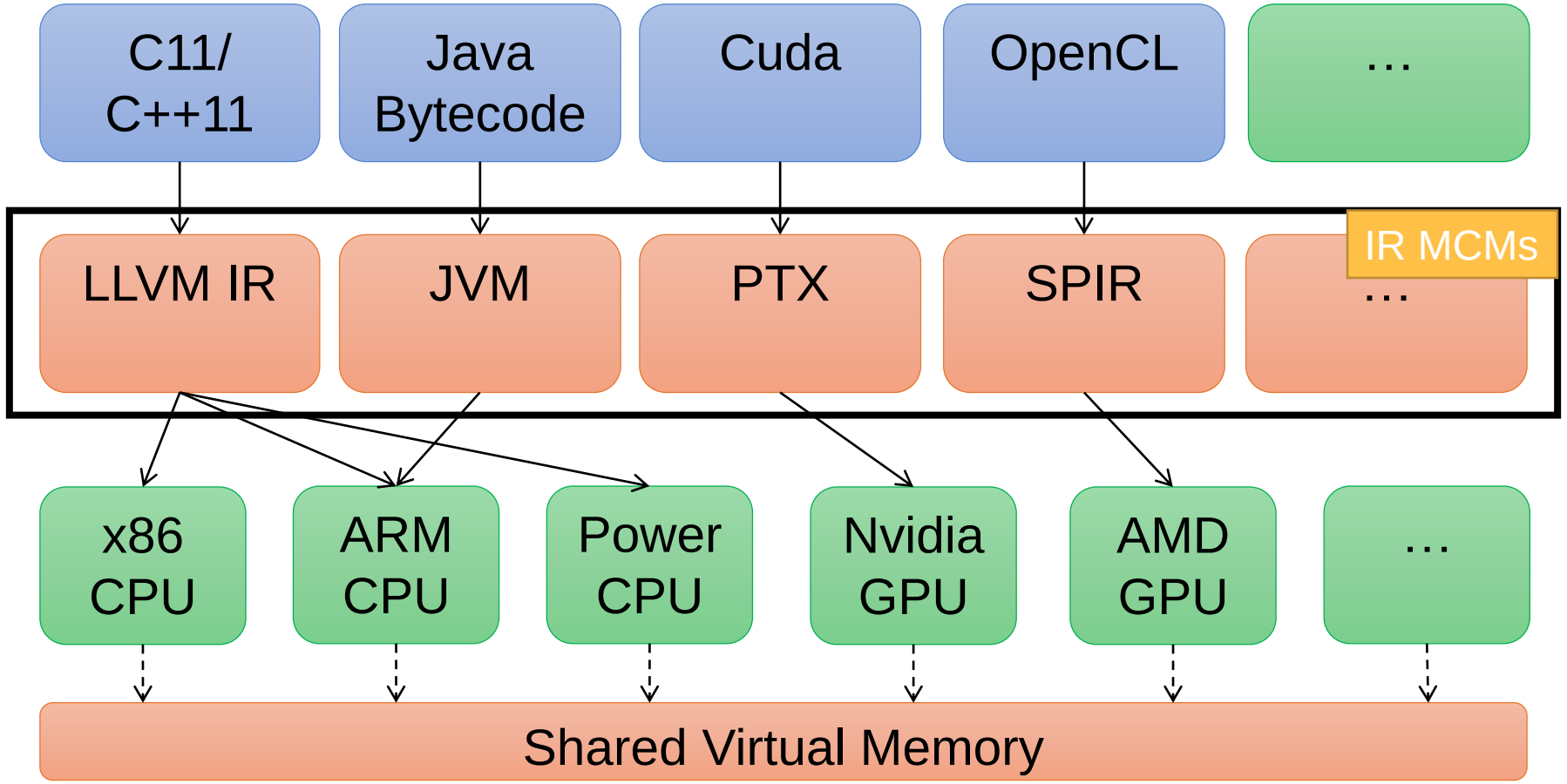
Memory Consistency Models (MCMs)

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



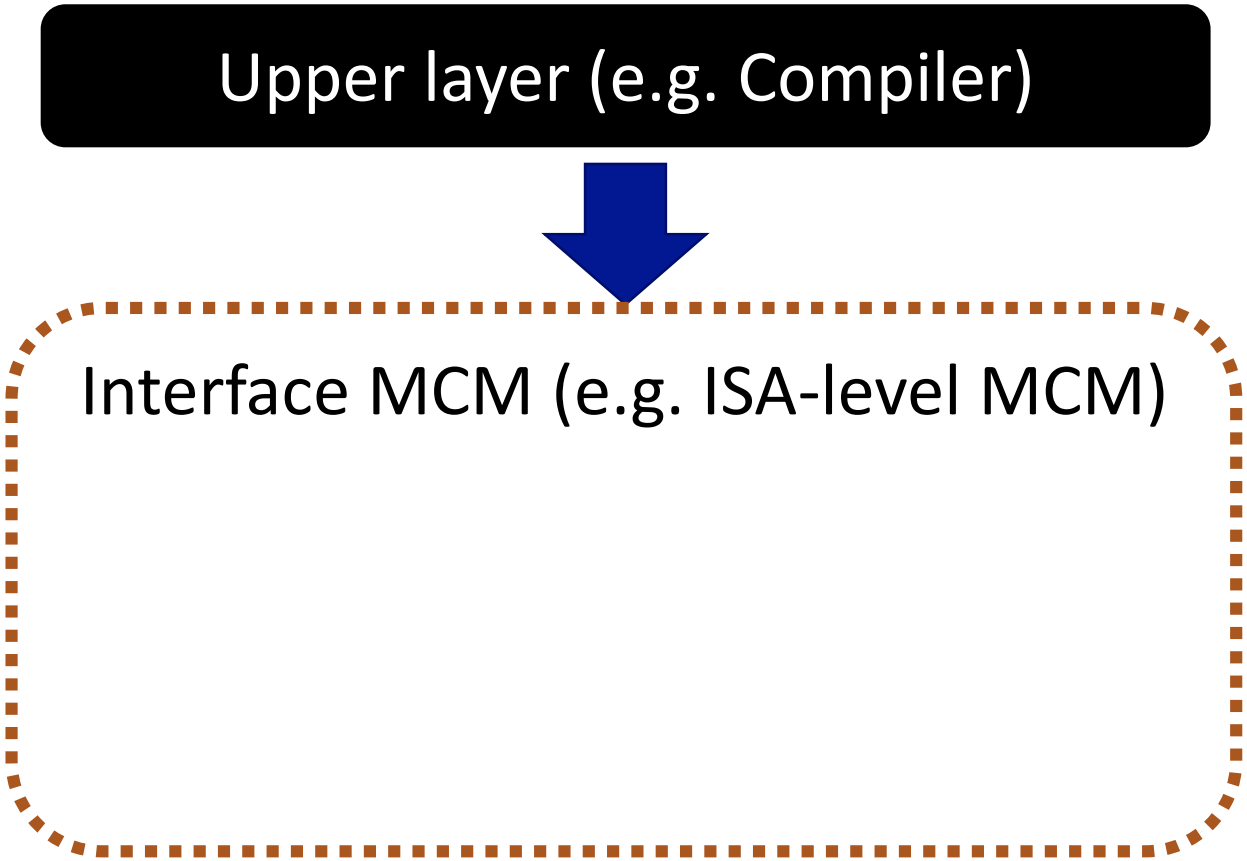
Memory Consistency Models (MCMs)

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



The Need for MCM Verification

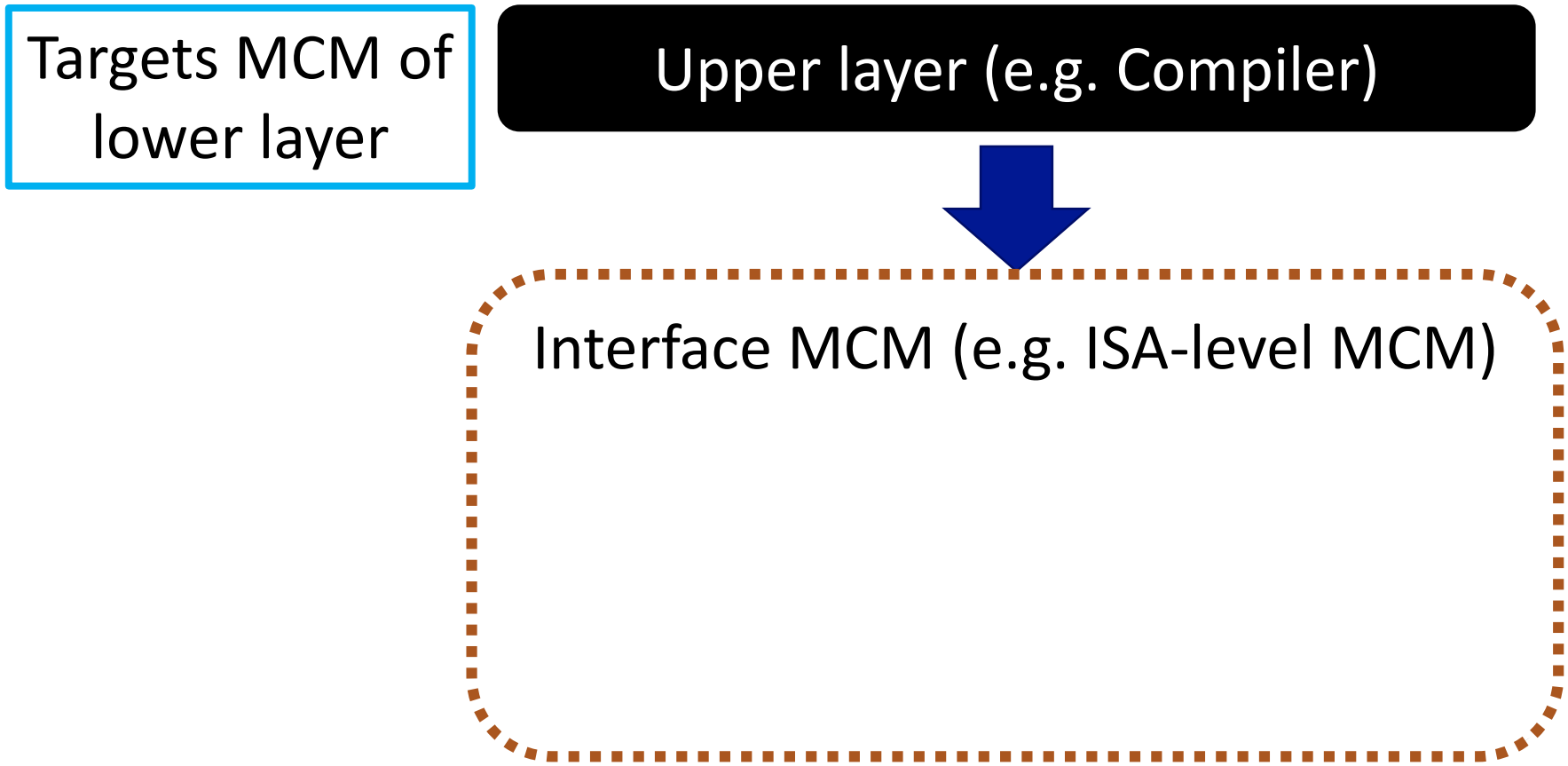
- MCMs are specified at interfaces between layers of the stack
  - Upper layers target MCM; **lower layers must maintain it for all programs!**



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

The Need for MCM Verification

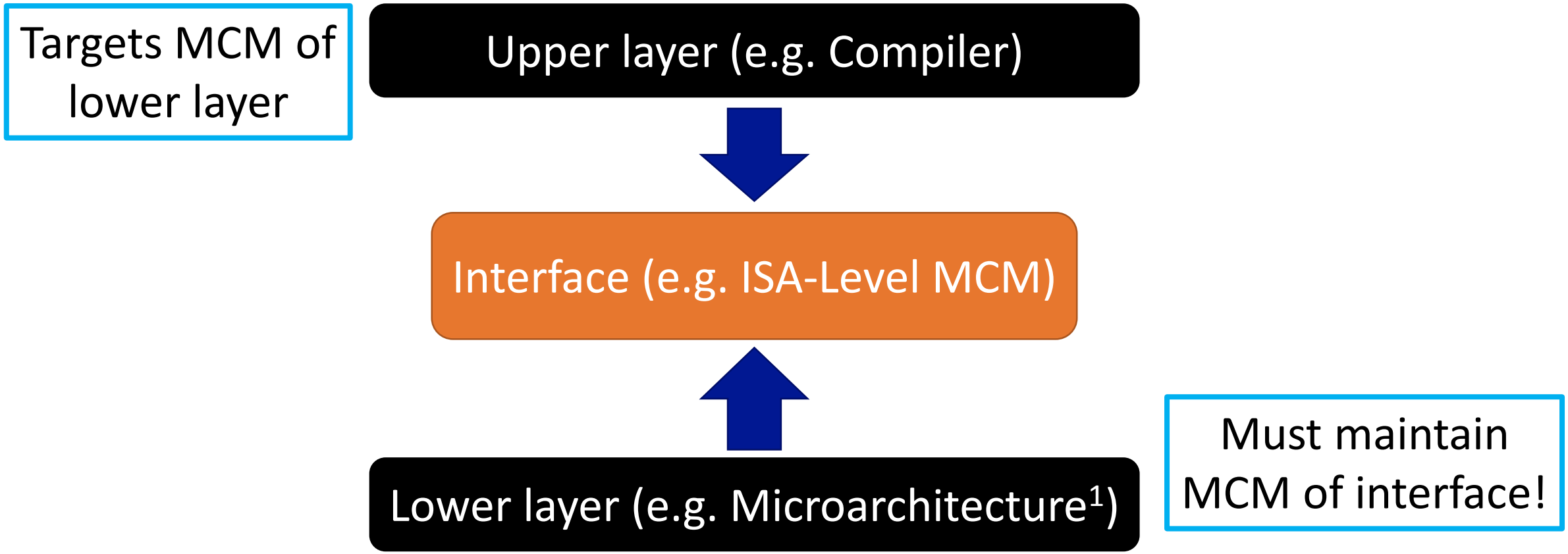
- MCMs are specified at interfaces between layers of the stack
  - Upper layers target MCM; **lower layers must maintain it for all programs!**



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

The Need for MCM Verification

- MCMs are specified at interfaces between layers of the stack
  - Upper layers target MCM; **lower layers must maintain it for all programs!**



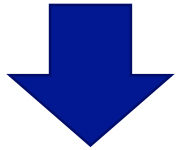
[1]Microarchitecture is a component-level (e.g. caches, pipeline stages, store buffers) model of the hardware.

The Need for MCM Verification

- MCMs are specified at interfaces between layers of the stack
  - Upper layers target MCM; **lower layers must maintain it for all programs!**

Targets MCM of lower layer

Upper layer (e.g. Compiler)



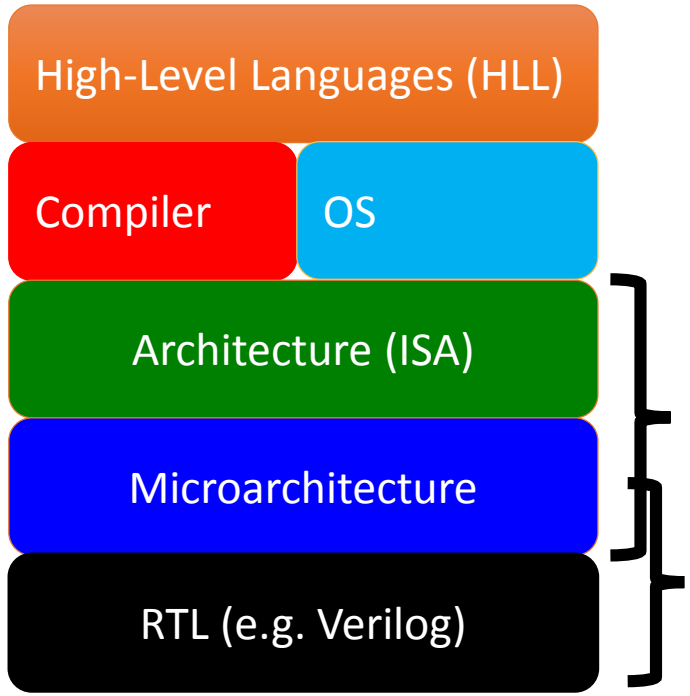
???

Lower layer (e.g. Microarchitecture[1])

Must maintain MCM of interface!

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

The Check Suite: Automated 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]

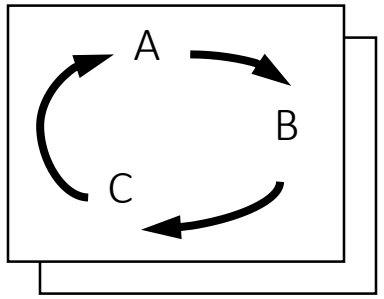
TriCheck [ASPLOS '17] [IEEE MICRO Top Picks]

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

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

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

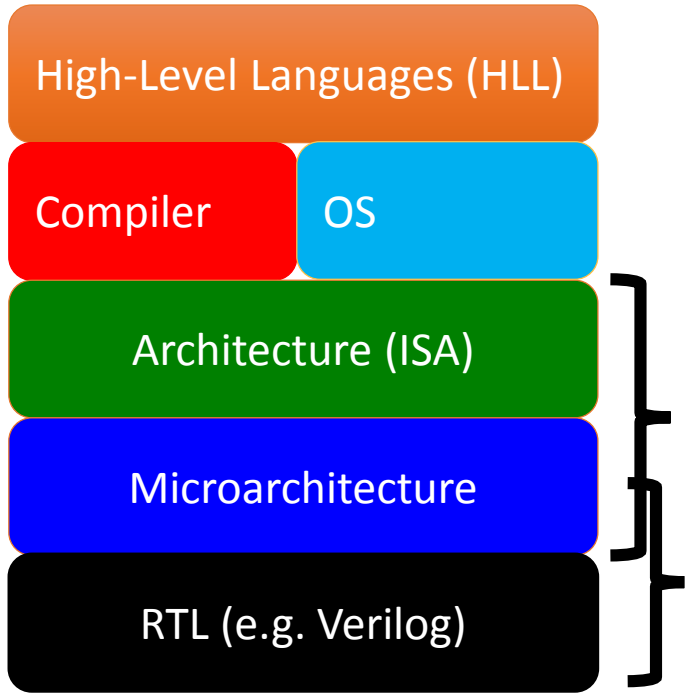
RTLCheck [Micro '17] [IEEE MICRO Top Picks Honorable Mention]



- **Axiomatic specifications** -> **Happens-before graphs**
  - Cyclic => Impossible, Acyclic => Possible
- Model Checking space of graphs using **SMT solvers**
- Most tools **written in Gallina** => can be proven correct

http://check.cs.princeton.edu

The Check Suite: Automated 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]

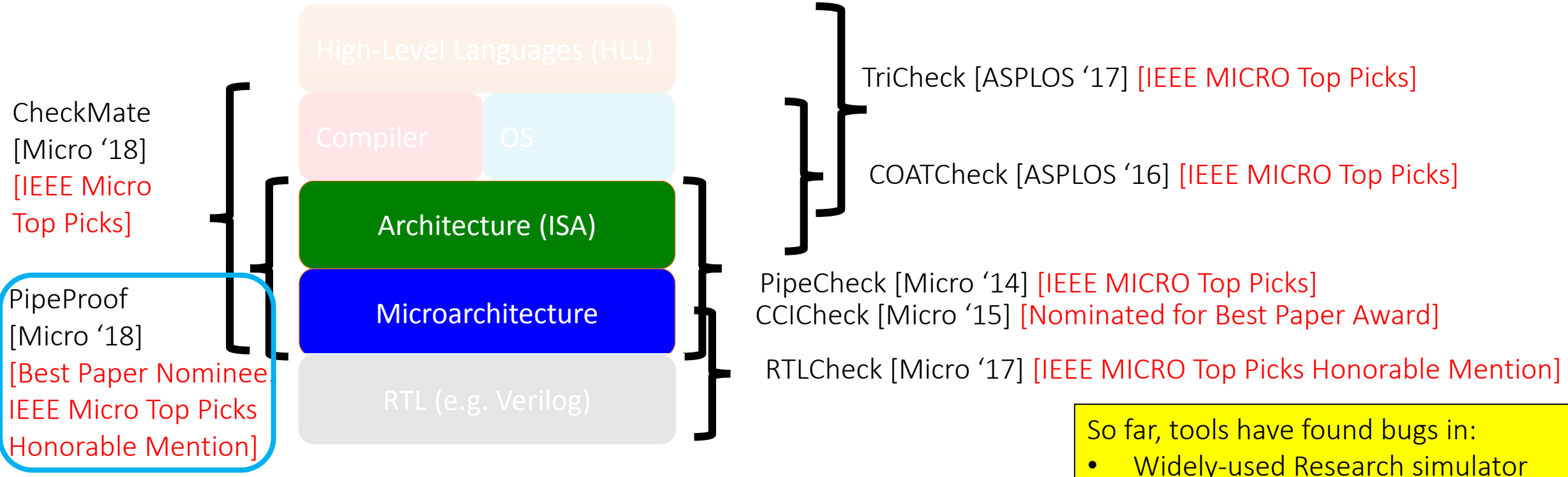
RTLCheck [Micro '17] [IEEE MICRO Top Picks Honorable Mention]

- **Axiomatic specifications** -> **Happens-before graphs**
  - Cyclic => Impossible, Acyclic => Possible
- Model Checking space of graphs using **SMT solvers**
- Most tools **written in Gallina** => can be proven correct

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
- Open-source RTL (Verilog)
- C++ 11 mem model
- SpectrePrime, MeltdownPrime

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

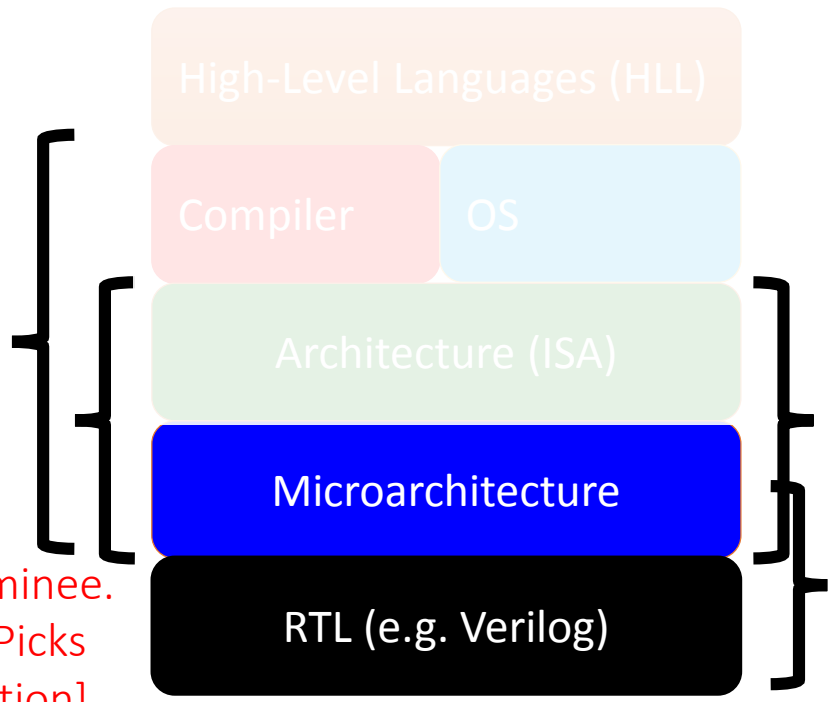


- **Axiomatic specifications** -> **Happens-before graphs**
  - Cyclic => Impossible, Acyclic => Possible
- Model Checking space of graphs using **SMT solvers**
- Most tools **written in Gallina** => can be proven correct

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
- Open-source RTL (Verilog)
- C++ 11 mem model
- SpectrePrime, MeltdownPrime

The Check Suite: Automated 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]

- **Axiomatic specifications** -> **Happens-before graphs**
  - Cyclic => Impossible, Acyclic => Possible
- Model Checking space of graphs using **SMT solvers**
- Most tools **written in Gallina** => can be proven correct

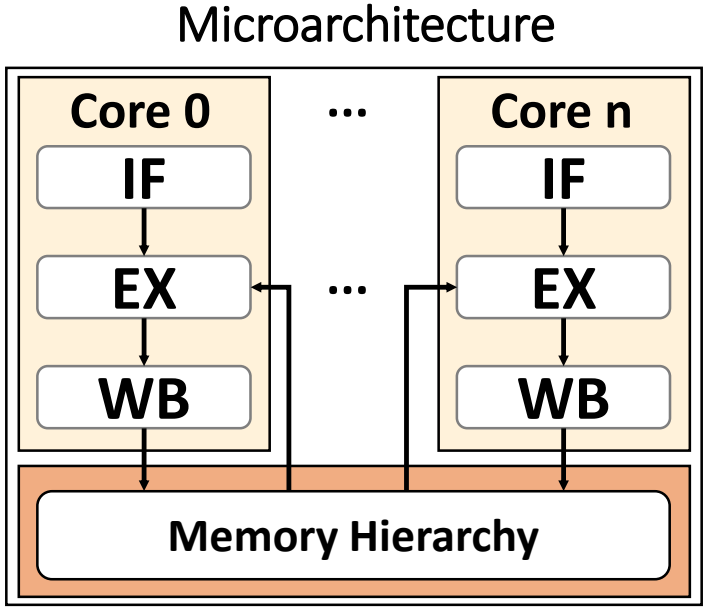
So far, tools have found bugs in:

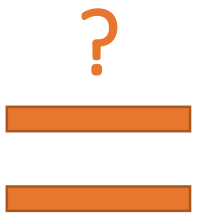
- Widely-used Research simulator
- Cache coherence paper
- IBM XL C++ compiler (fixed in v13.1.5)
- In-design commercial processors
- RISC-V ISA specification
- Open-source RTL (Verilog)
- C++ 11 mem model
- SpectrePrime, MeltdownPrime

Talk Outline

- Overview and Motivation
- Memory Consistency Background
- **PipeProof:** All-Program Microarchitectural MCM Verification
- **RTLCheck:** MCM Verification of Verilog RTL
- Expanding to other domains
- Conclusion

Microarchitectural MCM Verification





SC/TSO/RISC-V MCM?

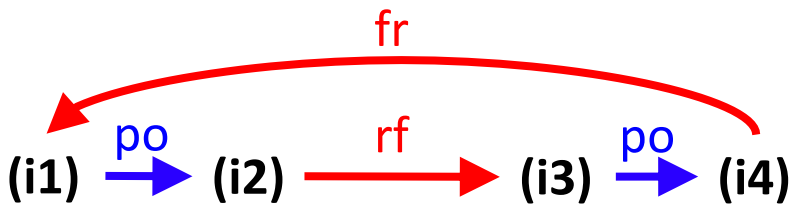
- **PipeProof proves that a microarchitecture respects its ISA MCM**
  - **For all possible programs!**
- How do we formally specify
  - ISA-level MCMs?
  - Microarchitectural orderings?

ISA-Level MCM Specifications

- MCMs often defined using relational patterns
  - [Shasha and Snir TOPLAS 1988] [Alglave et al. TOPLAS 2014]
- 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

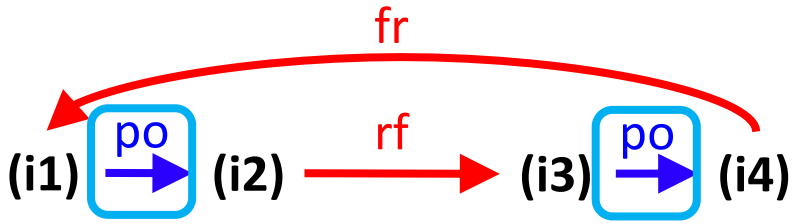
- Formal specifications of ISA + HLL MCMs in recent years
  - x86 [Owens et al. TPHOLS2009], ARM [Pulte et al. POPL2018], C11 [Batty et al. POPL 2011], ...
- Automated formal tools e.g. **herd** [Alglave et al. TOPLAS 2014]
  - Can formally analyse small test programs against these models

ISA-Level MCM Specifications

- MCMs often defined using relational patterns
  - [Shasha and Snir TOPLAS 1988] [Alglave et al. TOPLAS 2014]
- 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

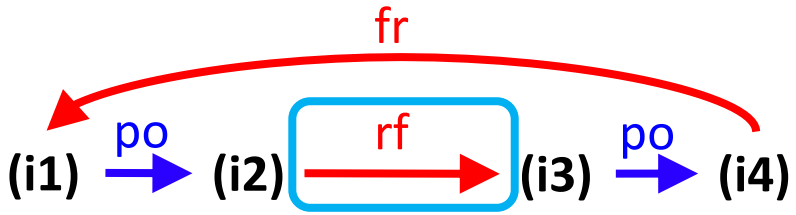
- Formal specifications of ISA + HLL MCMs in recent years
  - x86 [Owens et al. TPHOLS2009], ARM [Pulte et al. POPL2018], C11 [Batty et al. POPL 2011], ...
- Automated formal tools e.g. **herd** [Alglave et al. TOPLAS 2014]
  - Can formally analyse small test programs against these models

ISA-Level MCM Specifications

- MCMs often defined using relational patterns
  - [Shasha and Snir TOPLAS 1988] [Alglave et al. TOPLAS 2014]
- 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

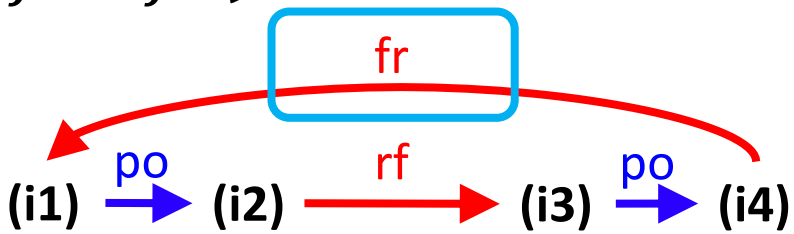
- Formal specifications of ISA + HLL MCMs in recent years
  - x86 [Owens et al. TPHOLS2009], ARM [Pulte et al. POPL2018], C11 [Batty et al. POPL 2011], ...
- Automated formal tools e.g. **herd** [Alglave et al. TOPLAS 2014]
  - Can formally analyse small test programs against these models

ISA-Level MCM Specifications

- MCMs often defined using relational patterns
  - [Shasha and Snir TOPLAS 1988] [Alglave et al. TOPLAS 2014]
- 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 in recent years
  - 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

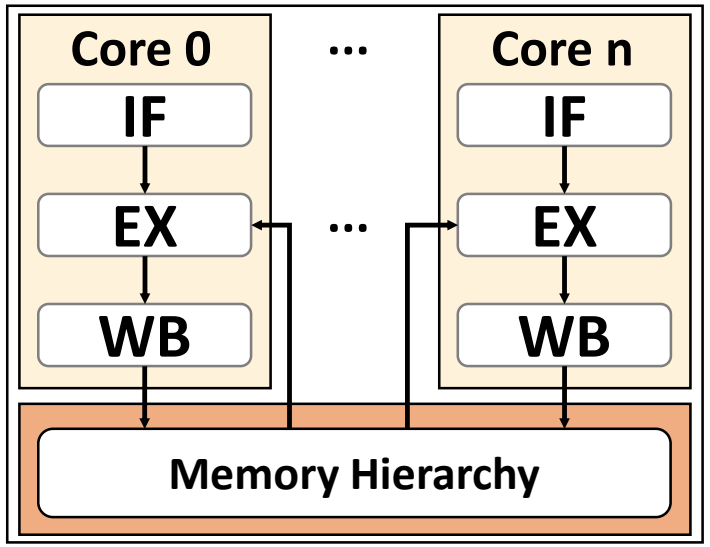
Microarchitectural Happens-Before (μhb) Graphs

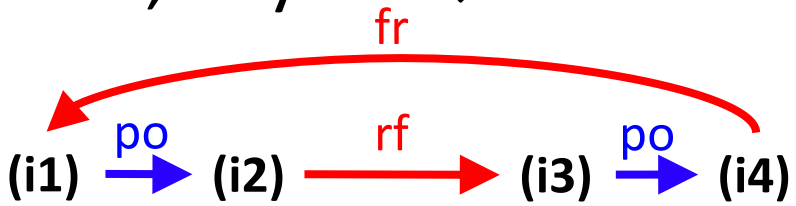
- Developed by PipeCheck [Lustig et al. MICRO 2014]
- Microarchitecture performs instrs. in stages
- Microarchitectural executions are μhb graphs
  - **Nodes:** instr. sub-events, **edges:** happens-before relationships
- Cyclic μhb graph → **unobservable**, Acyclic → **observable**

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	

simpleSC microarchitecture





Legend:
IF = Fetch
EX = Execute
WB = Writeback

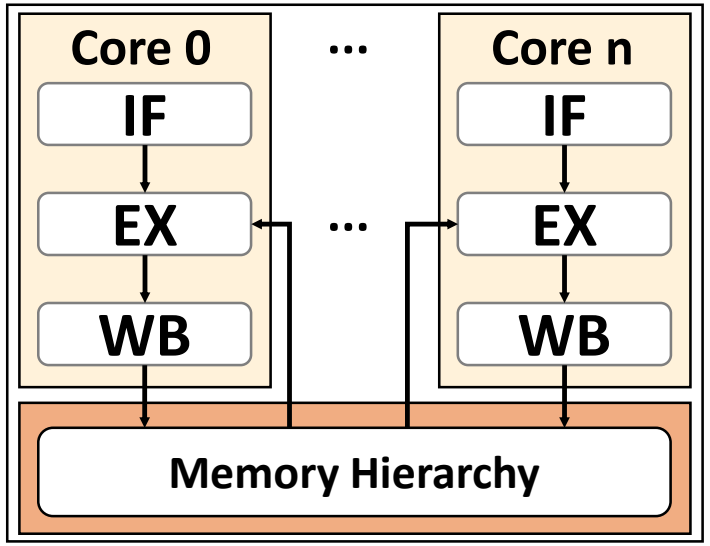
Microarchitectural Happens-Before (μhb) Graphs

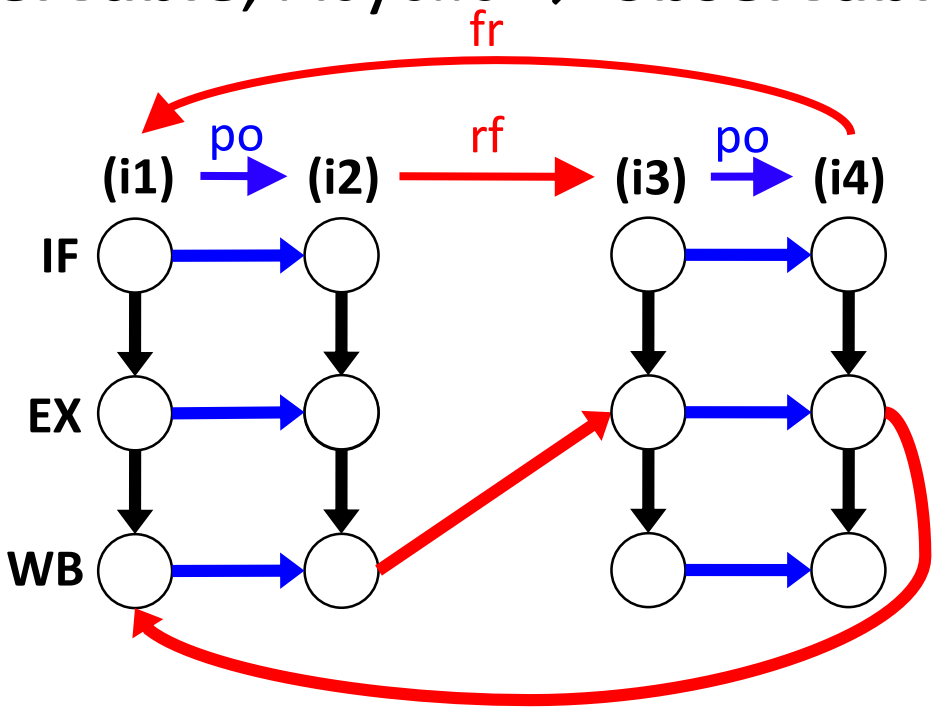
- Developed by PipeCheck [Lustig et al. MICRO 2014]
- Microarchitecture performs instrs. in stages
- Microarchitectural executions are μhb graphs
  - **Nodes:** instr. sub-events, **edges:** happens-before relationships
- Cyclic μhb graph → **unobservable**, Acyclic → **observable**

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	

simpleSC microarchitecture





Legend:
IF = Fetch
EX = Execute
WB = Writeback

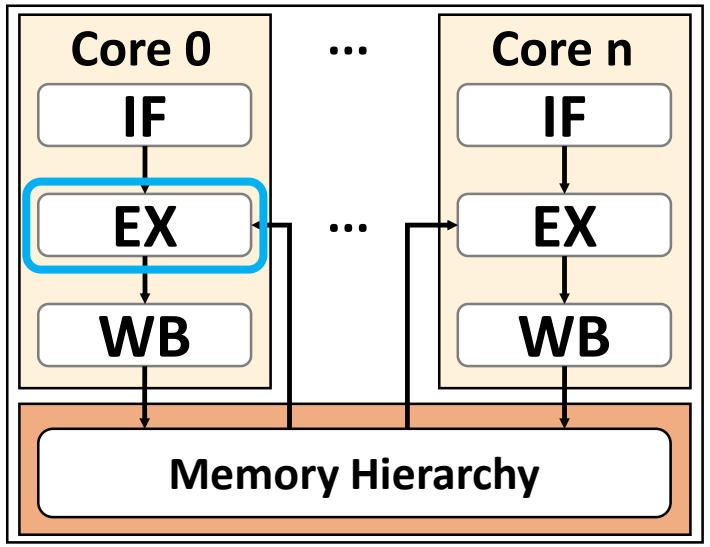
Microarchitectural Happens-Before (μhb) Graphs

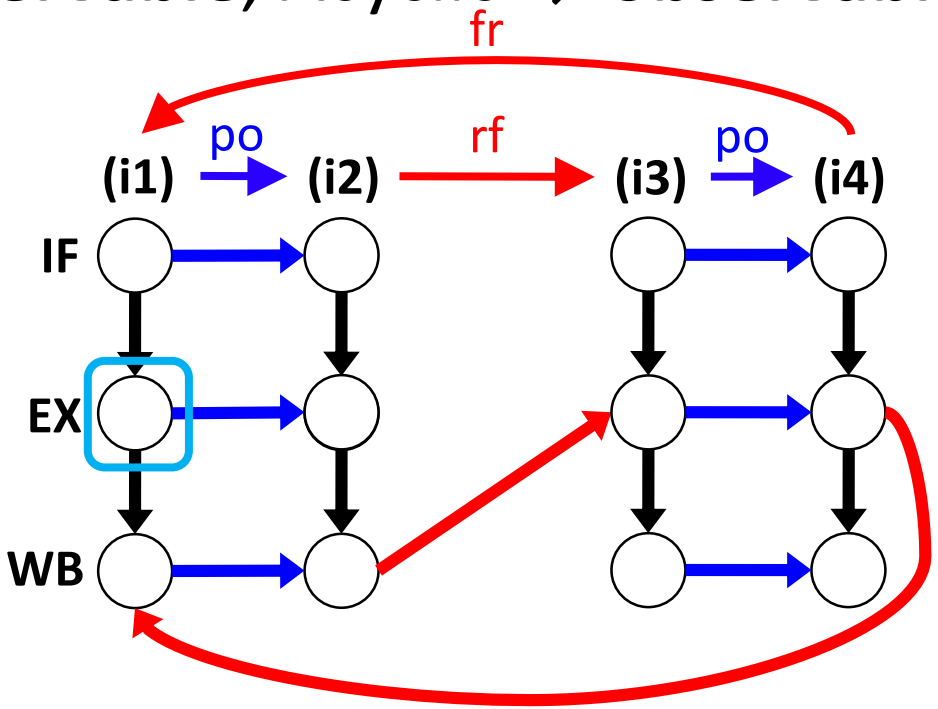
- Developed by PipeCheck [Lustig et al. MICRO 2014]
- Microarchitecture performs instrs. in stages
- Microarchitectural executions are μhb graphs
  - **Nodes:** instr. sub-events, **edges:** happens-before relationships
- Cyclic μhb graph → **unobservable**, Acyclic → **observable**

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	

simpleSC microarchitecture





Legend:
IF = Fetch
EX = Execute
WB = Writeback

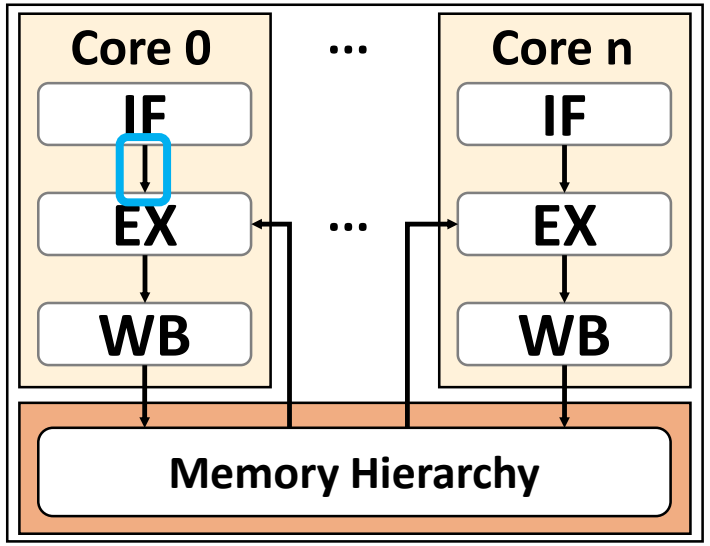
Microarchitectural Happens-Before (μhb) Graphs

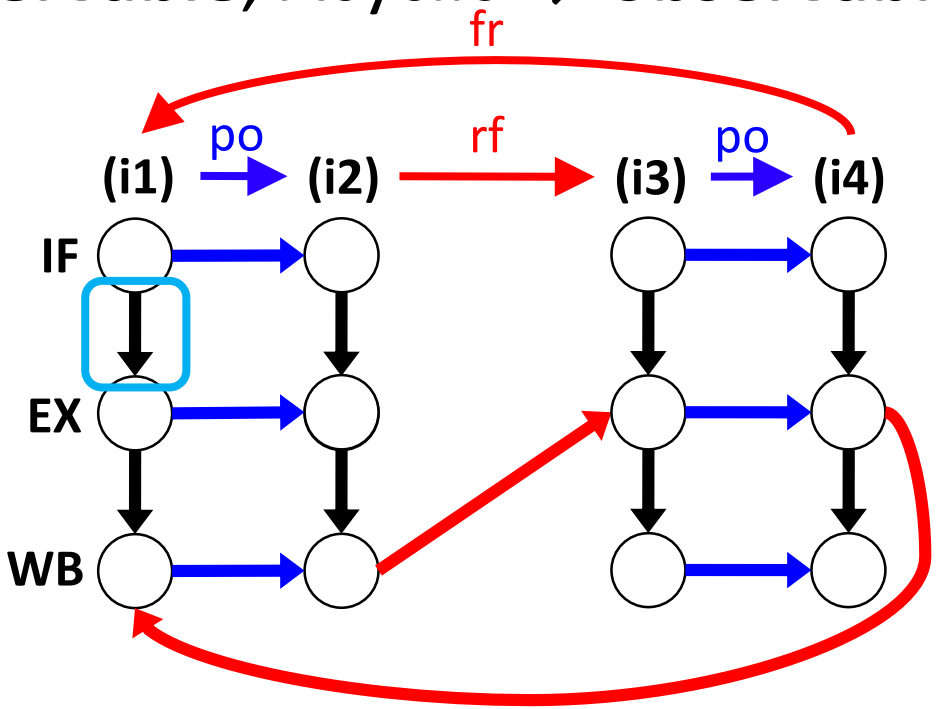
- Developed by PipeCheck [Lustig et al. MICRO 2014]
- Microarchitecture performs instrs. in stages
- Microarchitectural executions are μhb graphs
  - **Nodes:** instr. sub-events, **edges:** happens-before relationships
- Cyclic μhb graph → **unobservable**, Acyclic → **observable**

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	

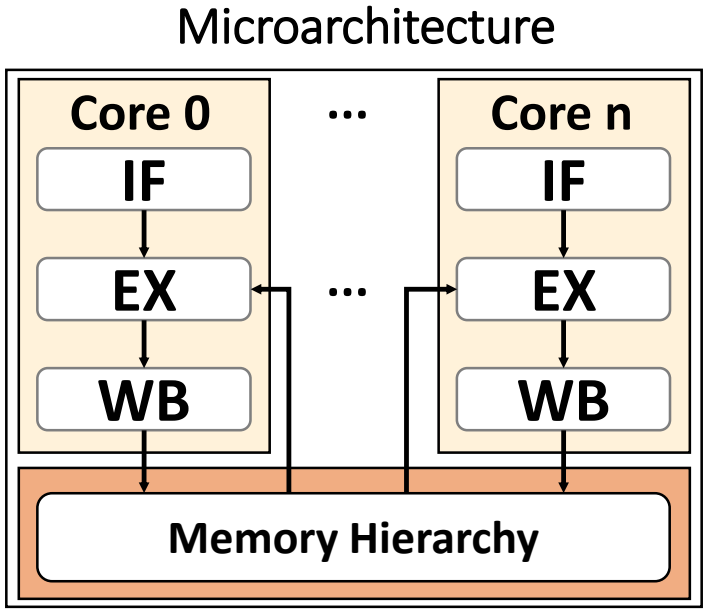
simpleSC microarchitecture





Legend:
IF = Fetch
EX = Execute
WB = Writeback

Microarchitectural MCM Verification



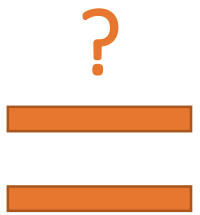
?

SC/TSO/RISC-V MCM?

Microarchitectural MCM Verification

Microarchitecture Specification in *μSpec* DSL

```
Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
  AddEdge ((i1, Fetch), (i2, Fetch), "PO").

Axiom "Execute_stage_is_in_order":
forall microops "i1",
...
```



SC/TSO/RISC-V MCM?

- μSpec DSL [Lustig et al. ASPLOS 2016] is similar to first-order logic (FOL)
  - `forall`, `exists`, AND (/\), OR (\/), NOT (~), implication (=>)
  - Has built-in predicates which take memory operations as input
    - e.g. `ProgramOrder i j` where `i` and `j` are loads/stores
  - Predicates can reference nodes and edges (μhb edges closed under transitivity)
    - e.g. `EdgeExists ((i1, Fetch), (i2, Fetch))`

PipeProof: Automated All-Program MCM Verif.

High-Level Languages (HLL)

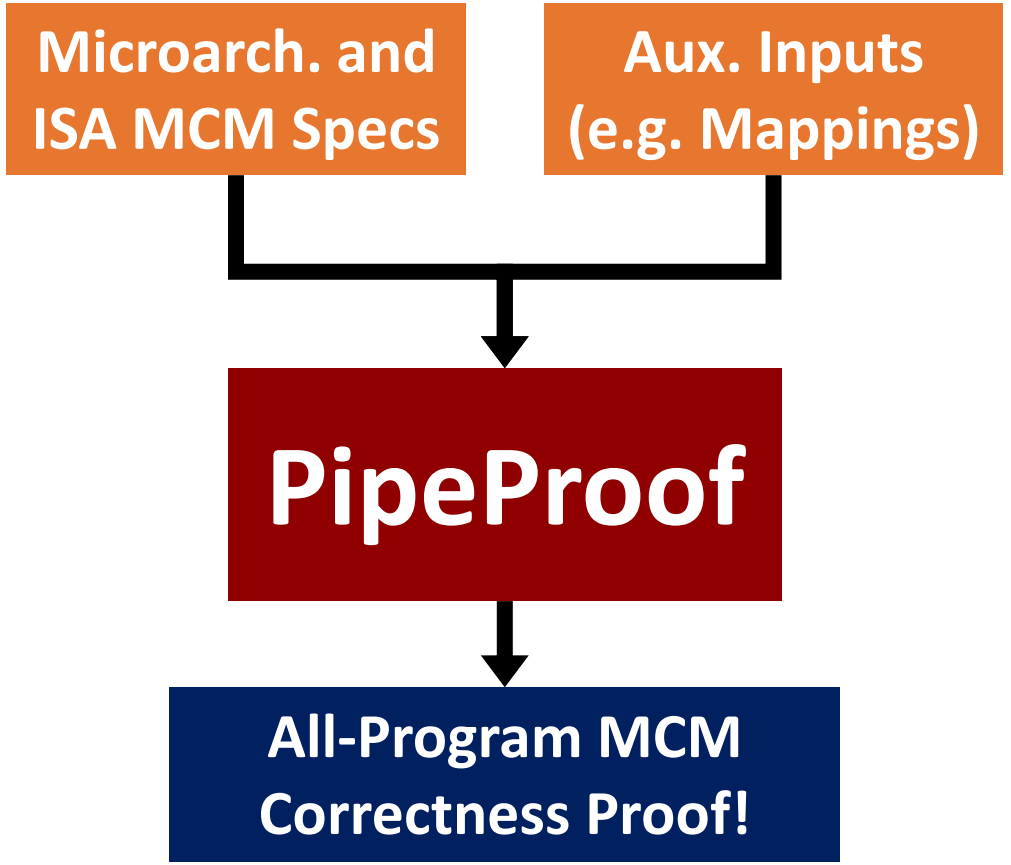
Compiler

Instruction Set (ISA)

Microarchitecture

Processor RTL (Verilog)

- **PipeProof** verifies that a microarchitecture correctly respects its ISA MCM across all possible programs
  - Early-stage **design-time verification** (i.e. before RTL)



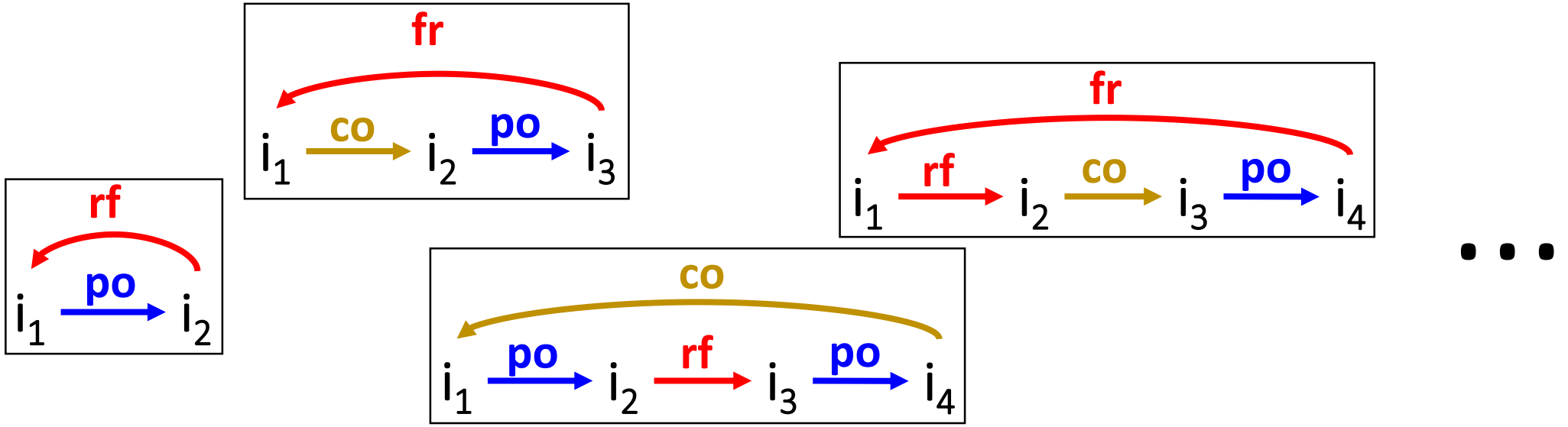
[**Yatin A. Manerkar**, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications. The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), October 2018.]

Verifying Across All Possible Programs

- Are all forbidden programs microarchitecturally unobservable?
  - If so, then microarchitecture is correct
- **Infinite** number of forbidden programs
  - E.g.: For SC, must check all possibilities of $cyclic(po \cup co \cup rf \cup fr)$
- Prove using **abstractions and induction**
  - Based on Counterexample-guided abstraction refinement [Clarke et al. CAV 2000]

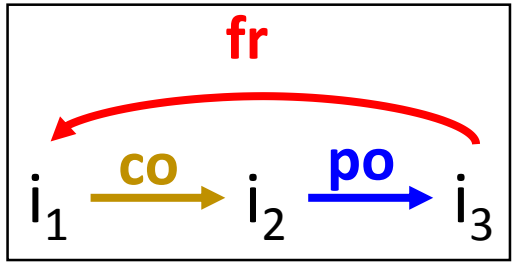
Verifying Across All Possible Programs

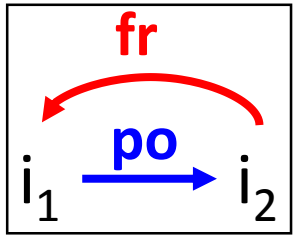
- Are all forbidden programs microarchitecturally unobservable?
  - If so, then microarchitecture is correct
- **Infinite** number of forbidden programs
  - E.g.: For SC, must check all possibilities of $cyclic(po \cup co \cup rf \cup fr)$
- Prove using **abstractions and induction**
  - Based on Counterexample-guided abstraction refinement [Clarke et al. CAV 2000]

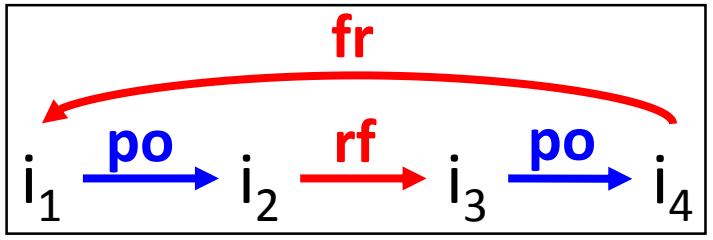


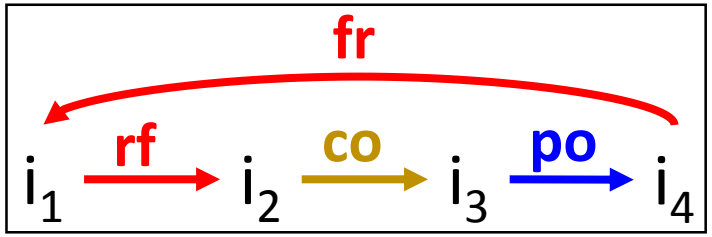
The Transitive Chain (TC) Abstraction

All non-unary cycles containing **fr**
(**Infinite set**)





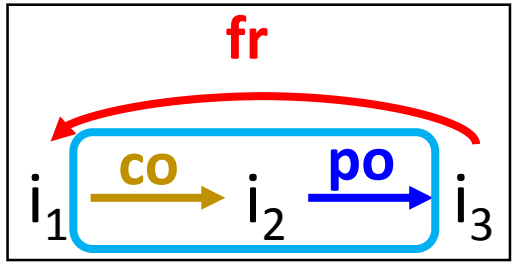


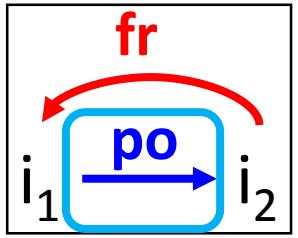


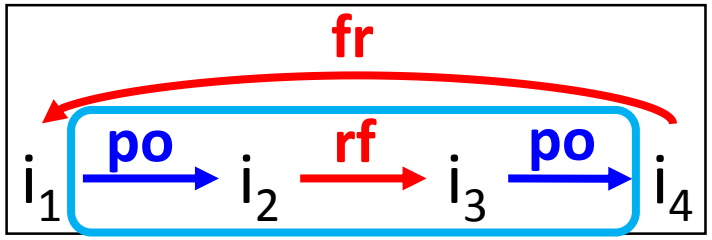
. . .

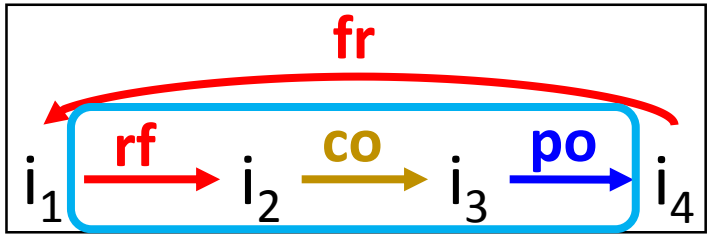
The Transitive Chain (TC) Abstraction

All non-unary cycles containing **fr**
(**Infinite set**)







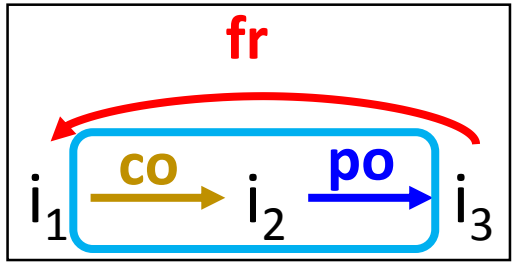


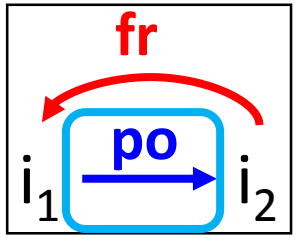
. . .

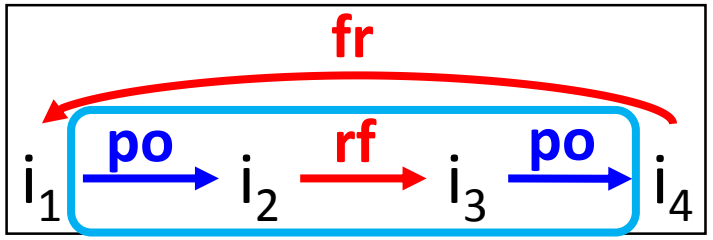
Cycle = Transitive Chain (sequence)
+ Loopback edge (fr)

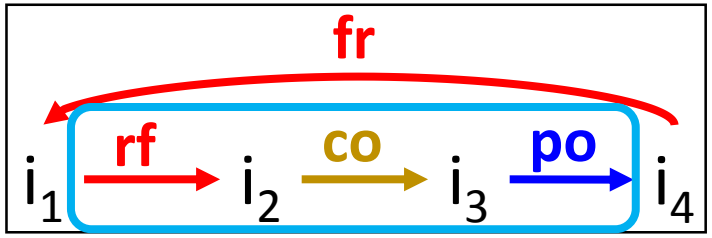
The Transitive Chain (TC) Abstraction

All non-unary cycles containing **fr**
(**Infinite set**)





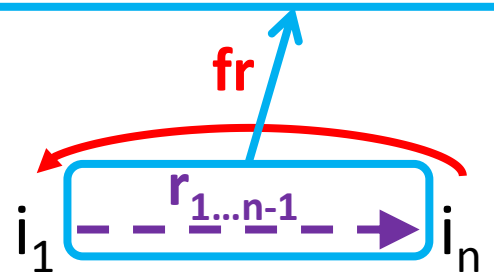




. . .

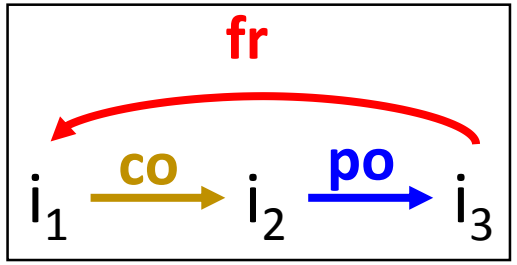
Cycle = Transitive Chain (sequence)
+ Loopback edge (fr)

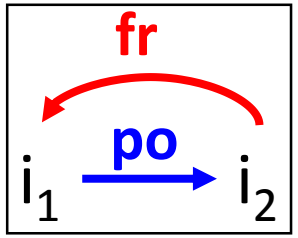
Transitive chain (sequence)
of ISA-level edges

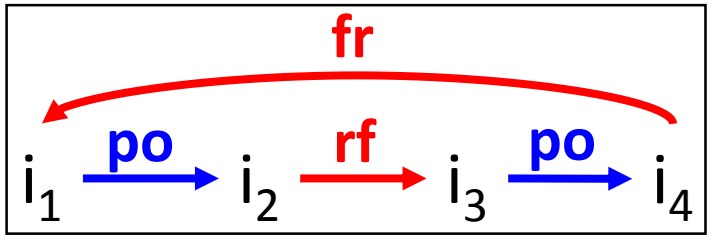


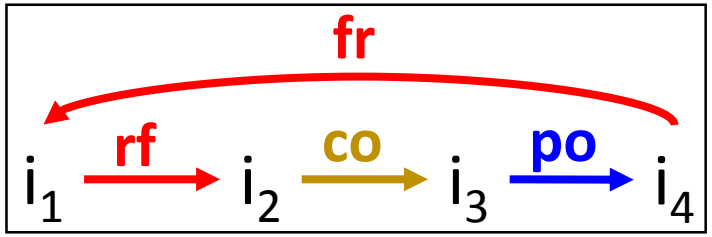
The Transitive Chain (TC) Abstraction

All non-unary cycles containing **fr**
(**Infinite set**)

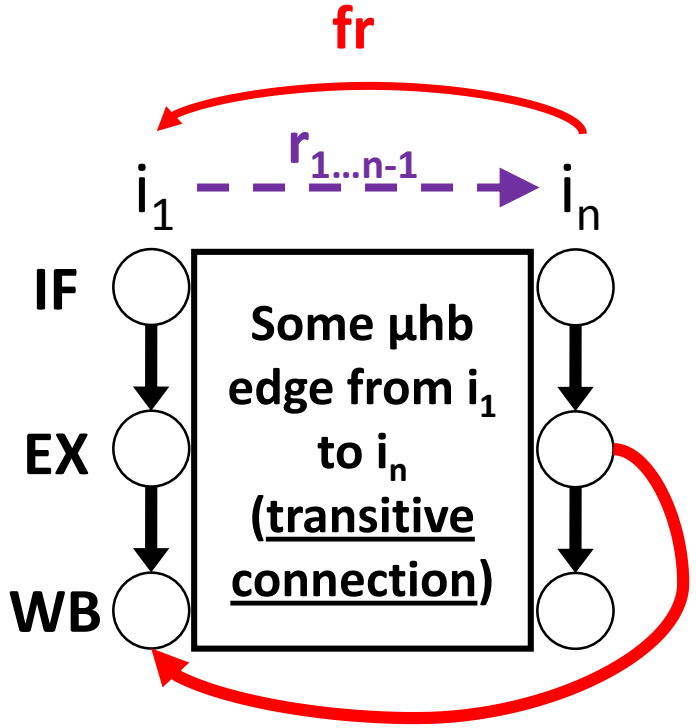








. . .

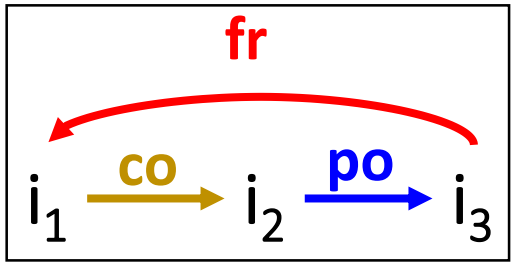


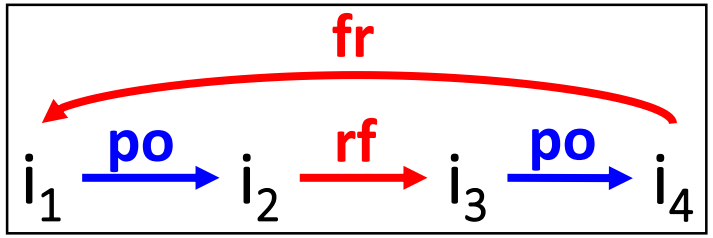
Cycle = Transitive Chain (sequence)
+ Loopback edge (fr)

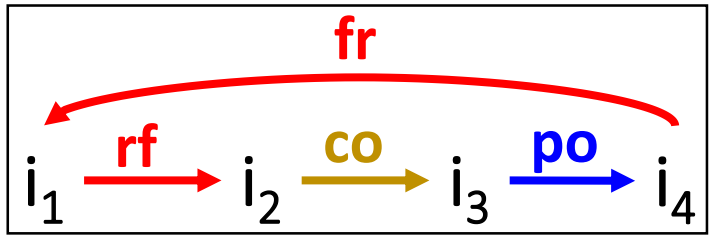
ISA-level **transitive chain** =>
Microarch. level **transitive connection**

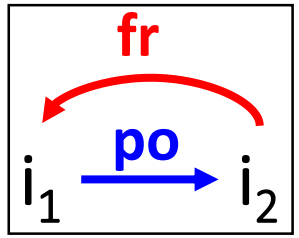
The Transitive Chain (TC) Abstraction

Infinite!





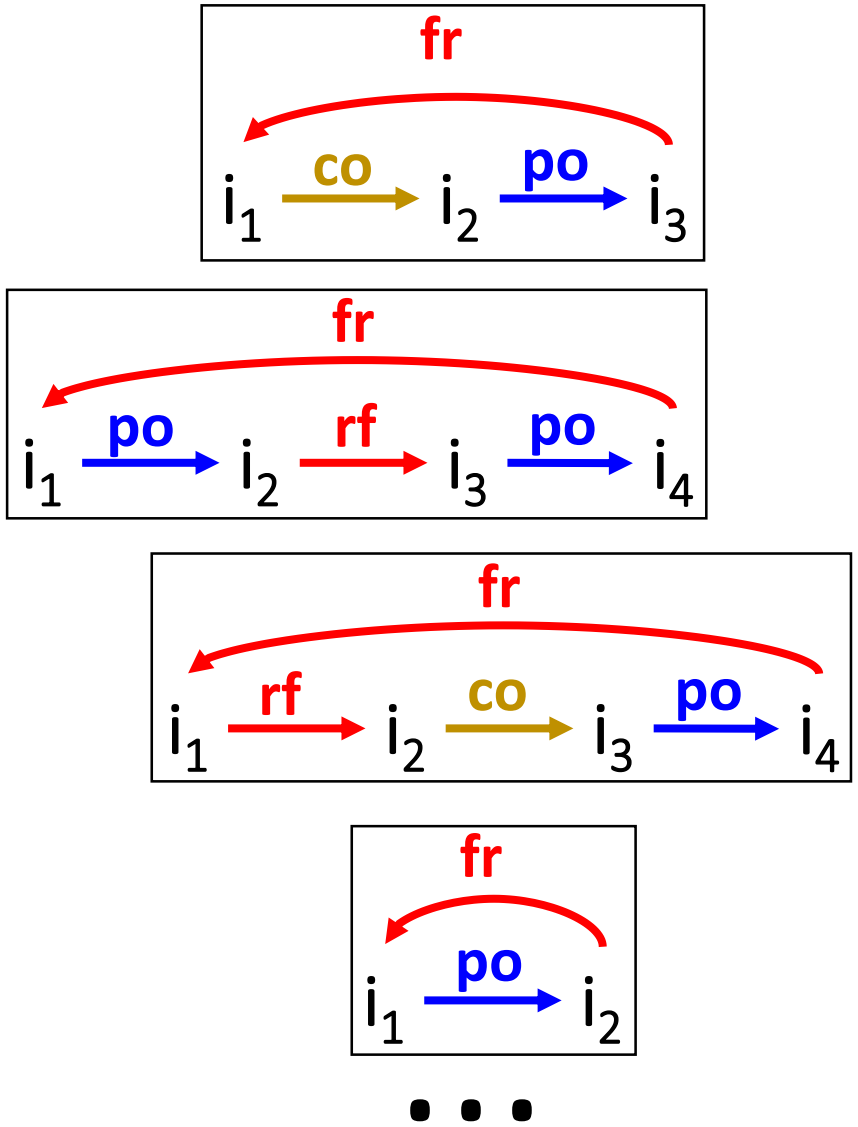




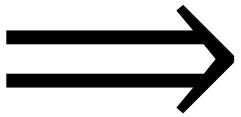
. . .

The Transitive Chain (TC) Abstraction

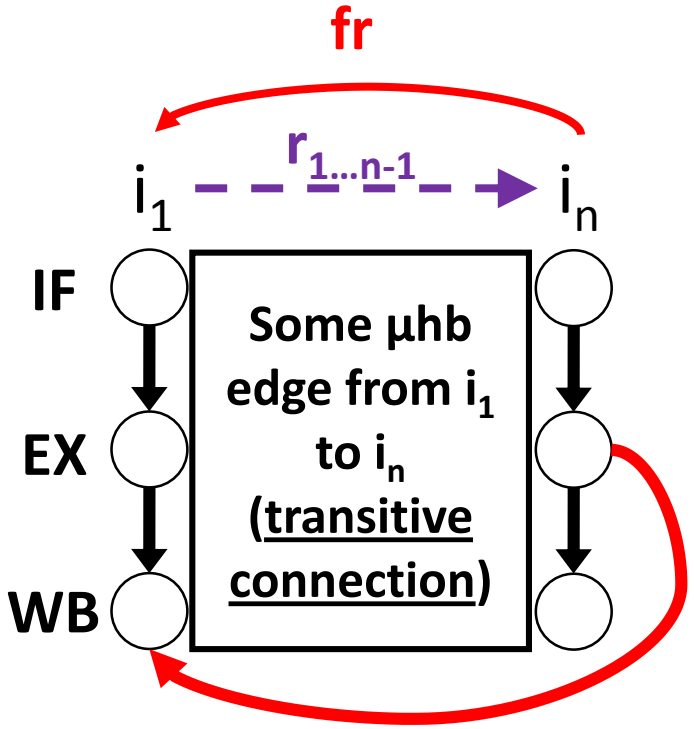
Infinite!



Using TC Abstraction



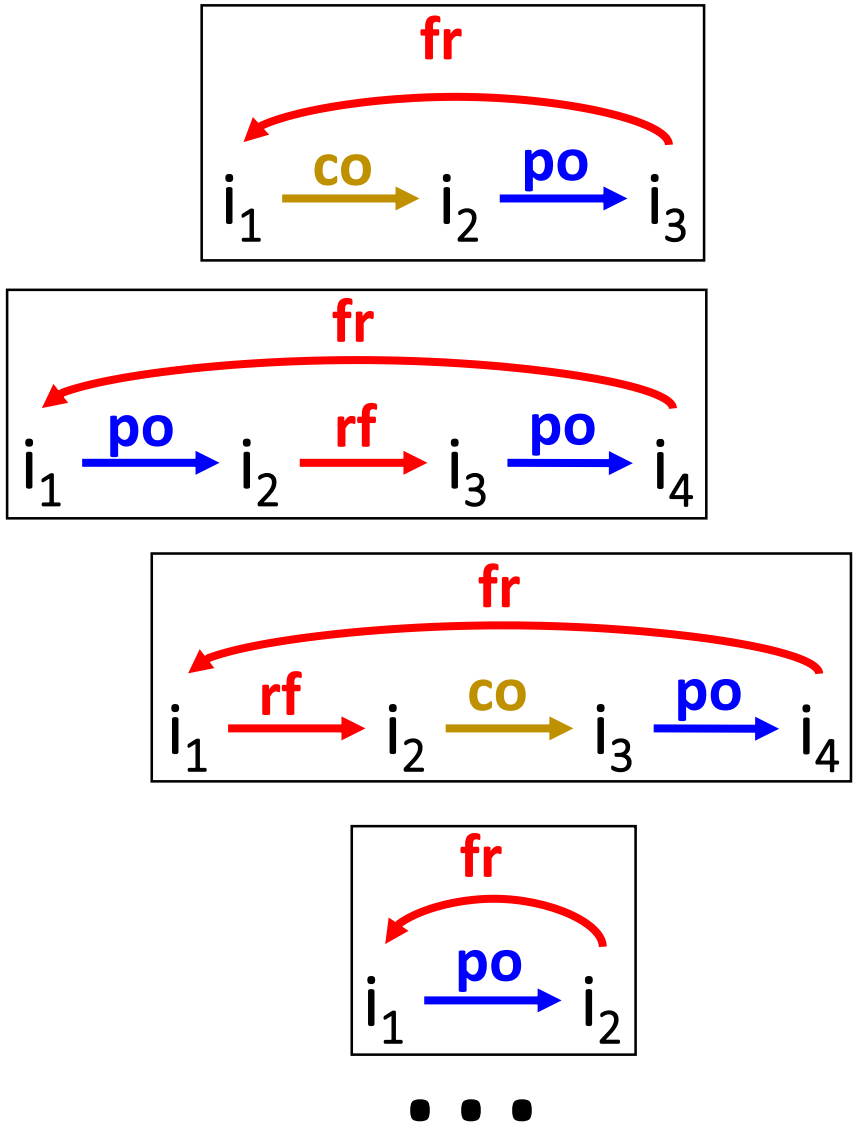
Finite!



3 x 3 = 9 possible transitive connections from i_1 to i_n

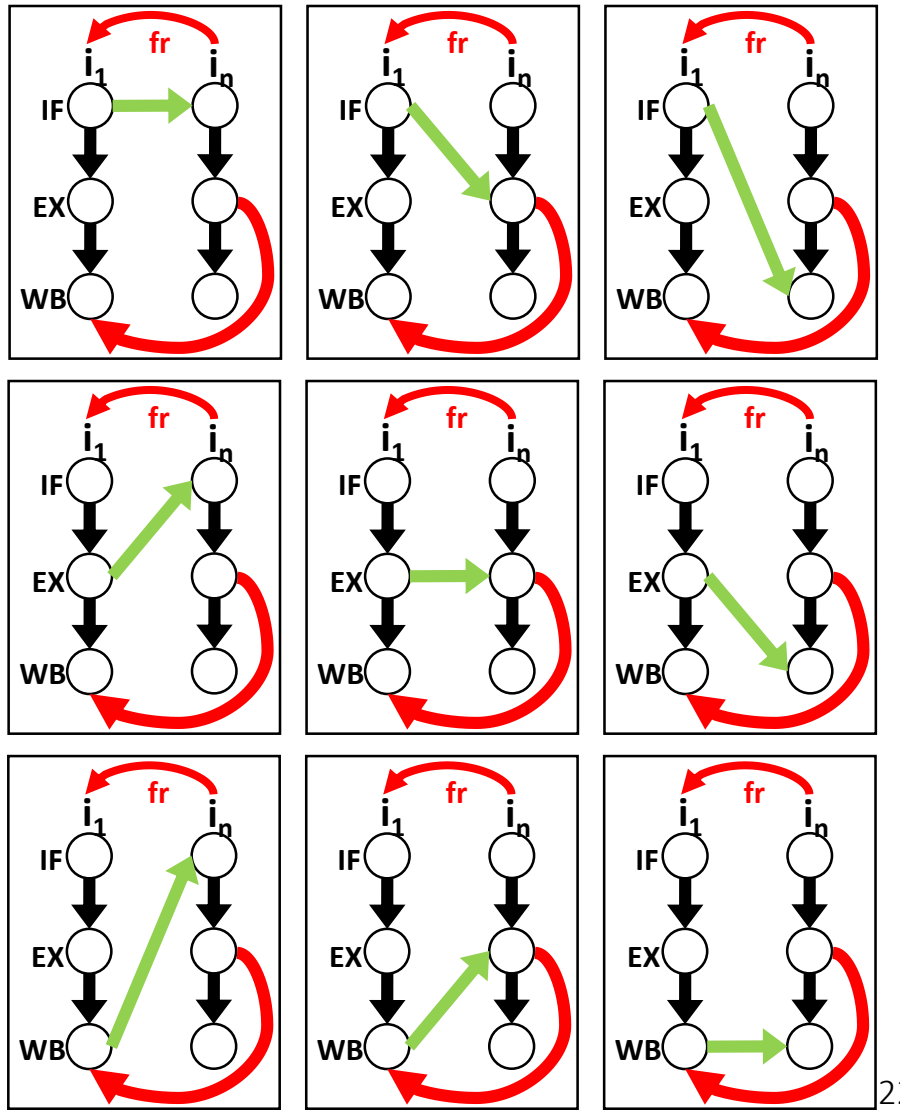
The Transitive Chain (TC) Abstraction

Infinite!



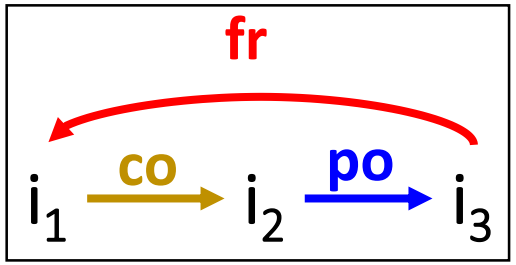
Using TC Abstraction

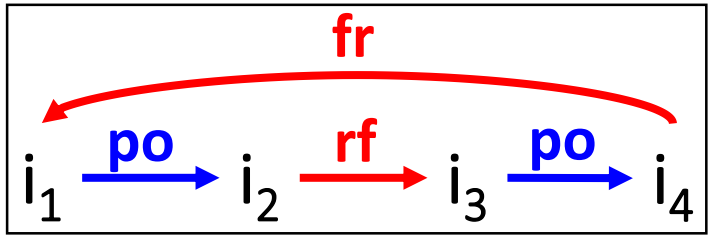
Finite!

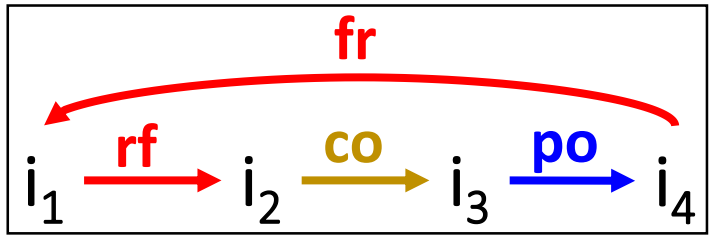


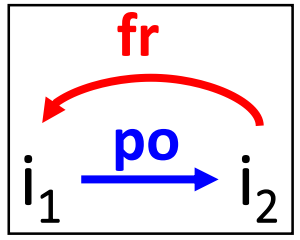
The Transitive Chain (TC) Abstraction

Infinite!









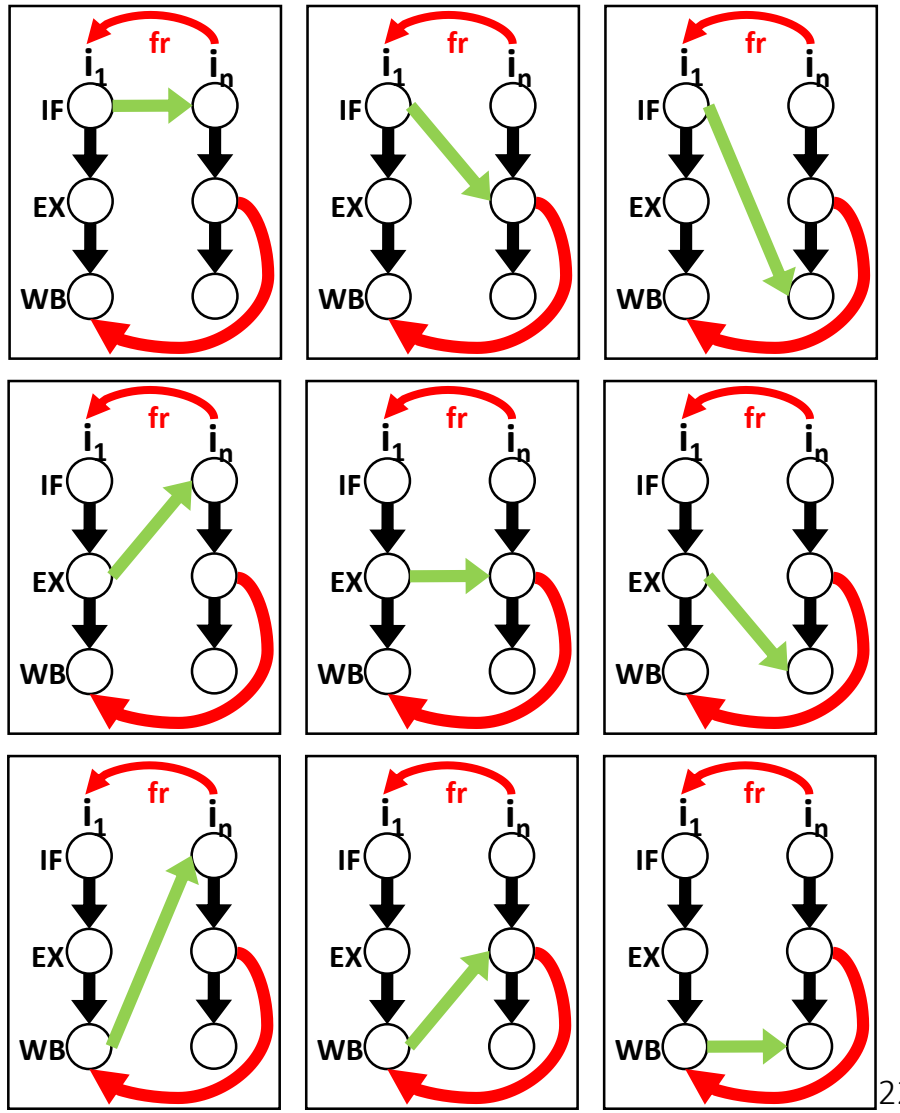
. . .

Abstraction soundness automatically verified as a supporting proof!

Using TC Abstraction

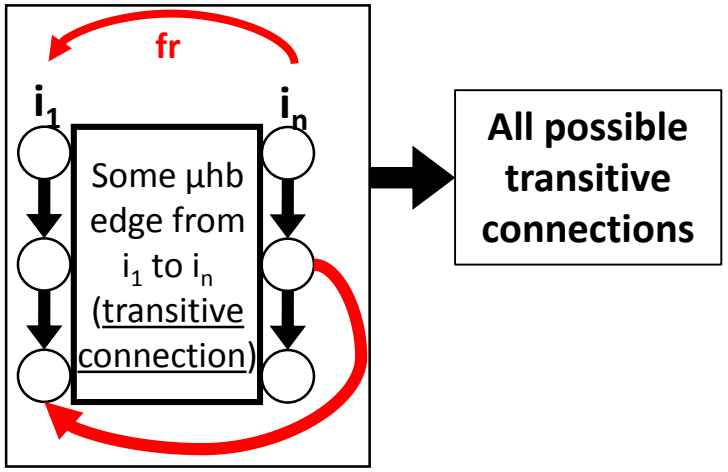
⟹

Finite!



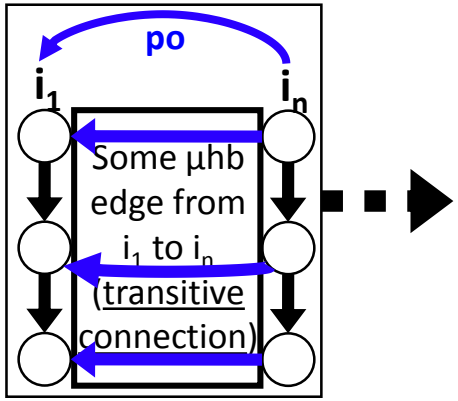
Microarchitectural Correctness Proof

Cycles containing *fr*



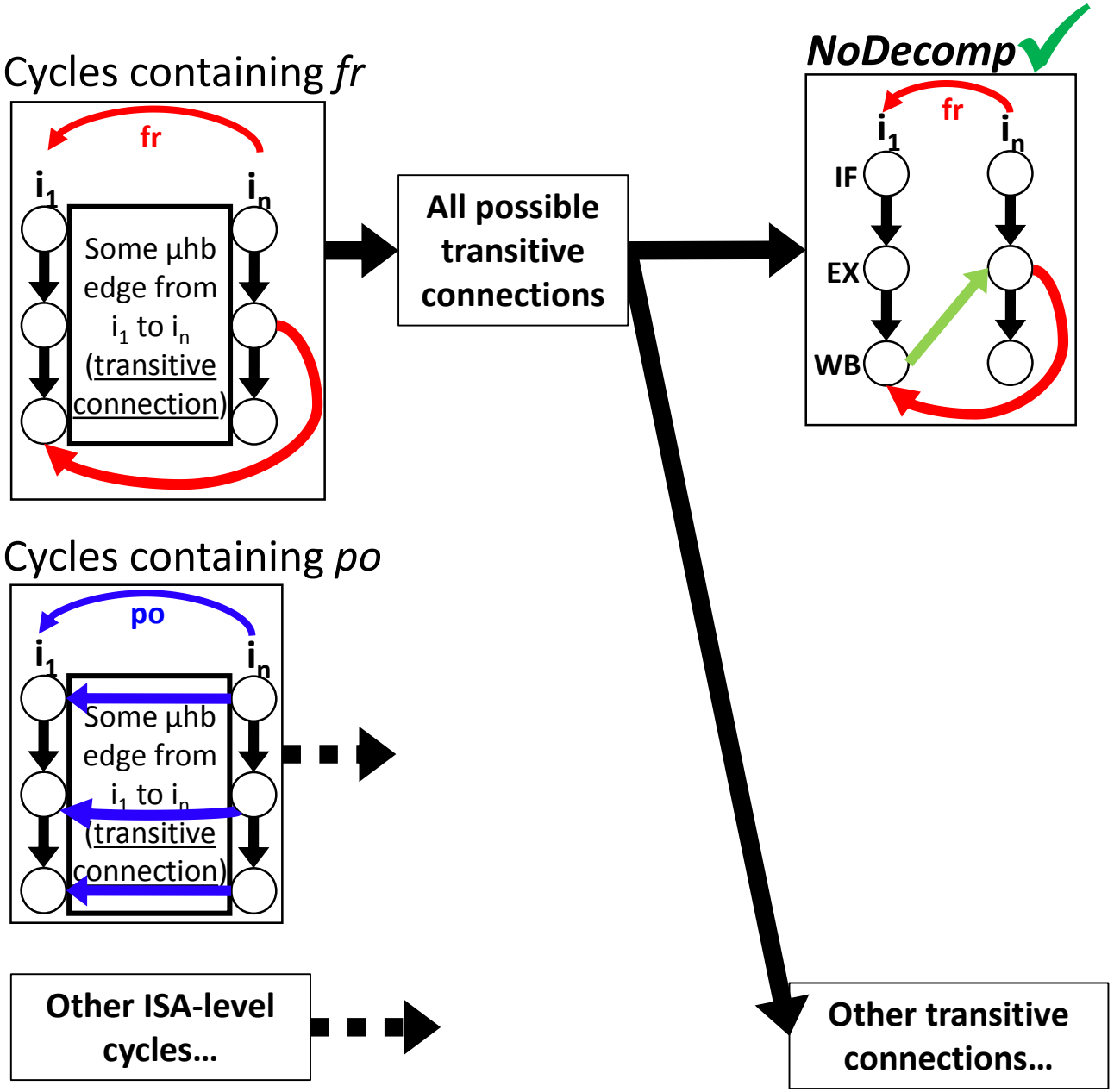
All possible transitive connections

Cycles containing *po*

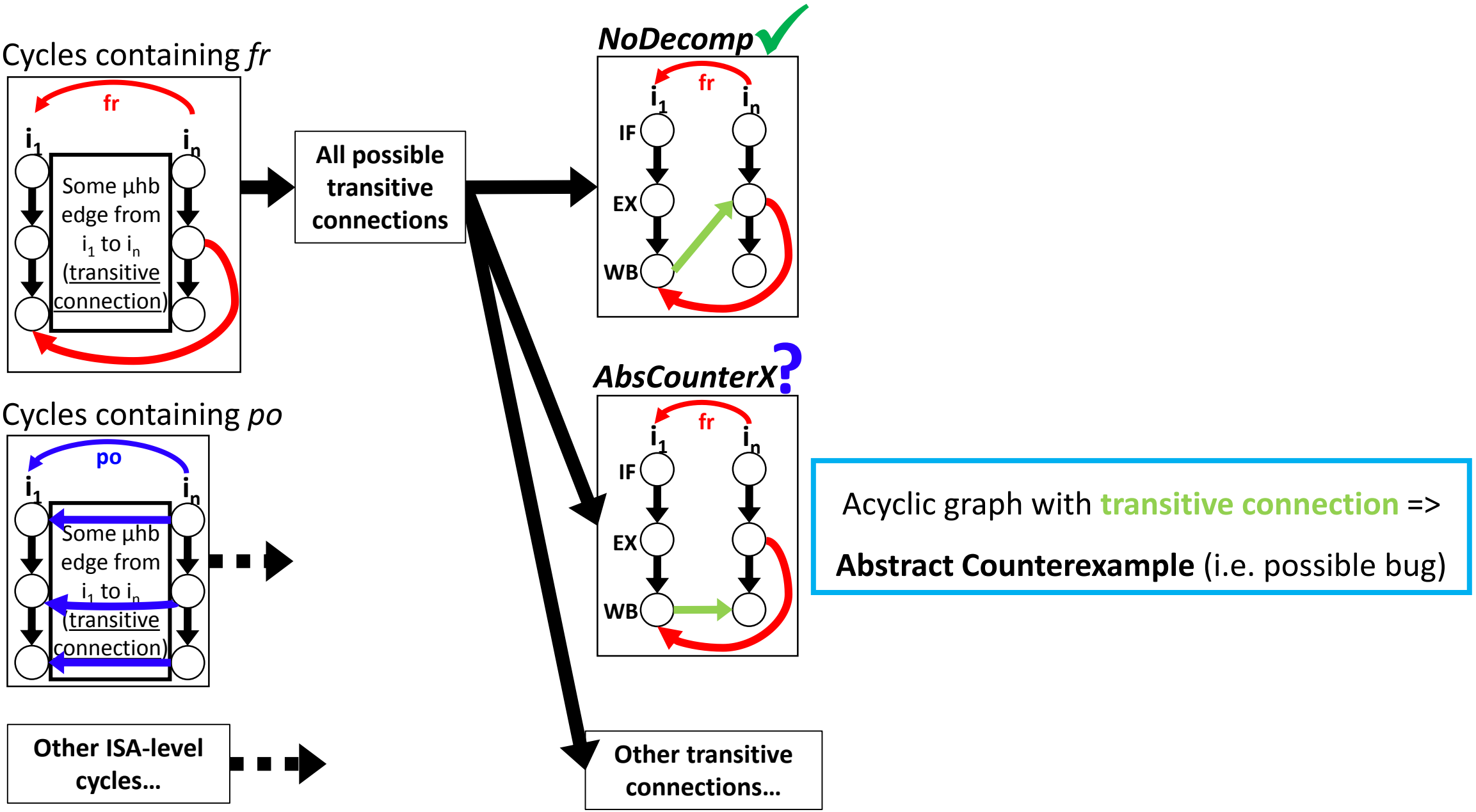


Other ISA-level cycles…

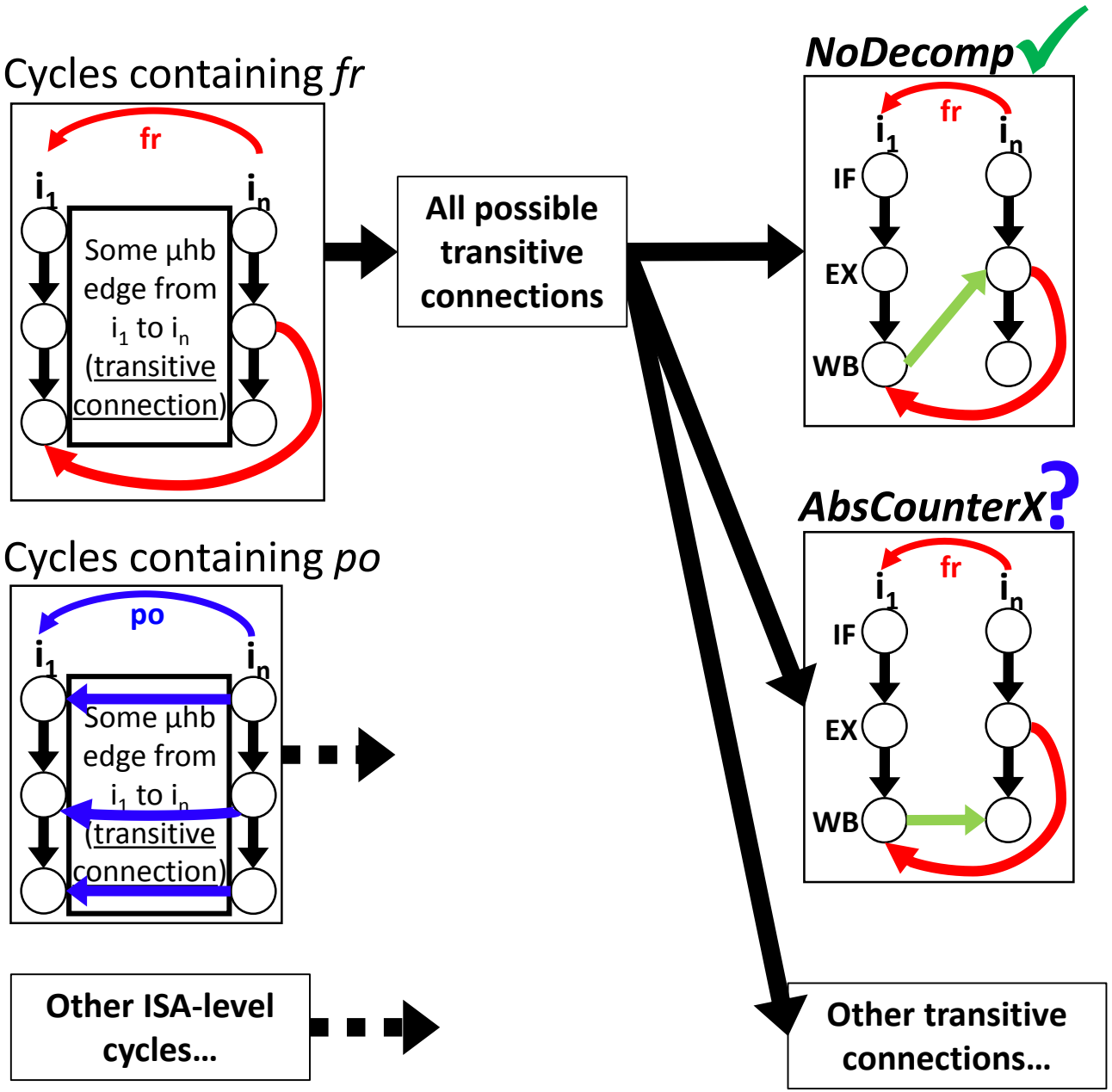
Microarchitectural Correctness Proof



Microarchitectural Correctness Proof



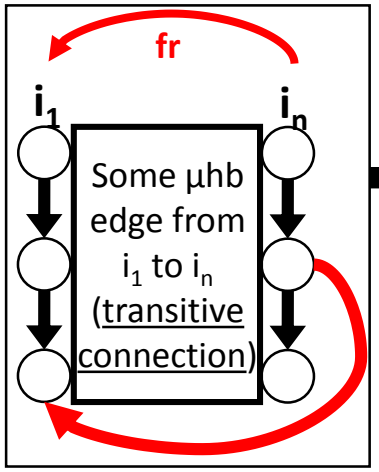
Microarchitectural Correctness Proof



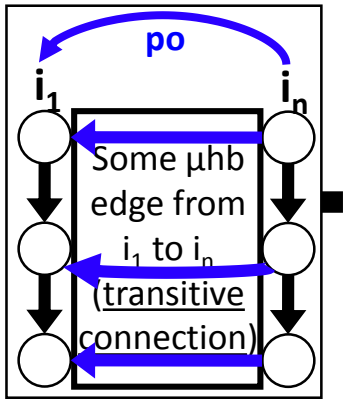
Transitive connection (green edge) may represent one or multiple ISA-level edges

Microarchitectural Correctness Proof

Cycles containing *fr*



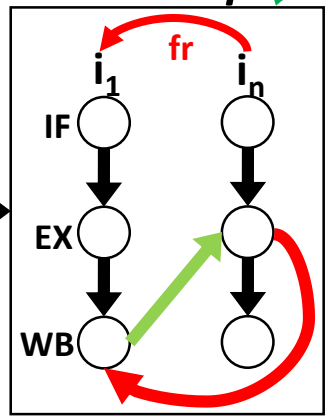
Cycles containing *po*



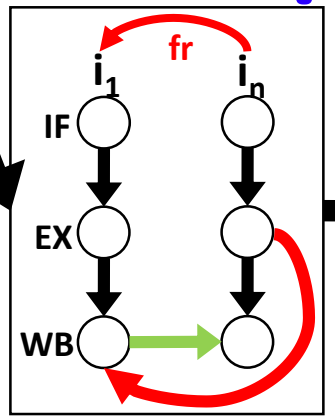
Other ISA-level cycles…

All possible transitive connections

NoDecomp ✓



AbsCounterX ?



Other transitive connections…

Try to Concretize (Replace transitive connection with one ISA-level edge)

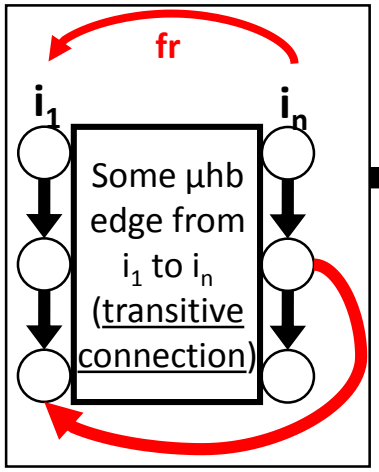
Observable

Microarch Buggy, Return Counterexample

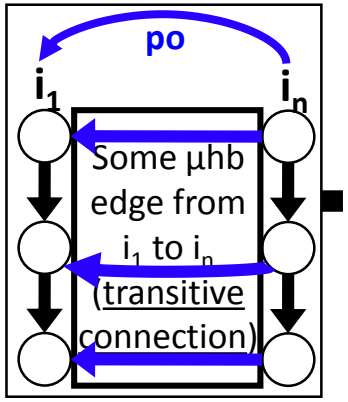
Transitive connection (green edge) may represent one or multiple ISA-level edges

Microarchitectural Correctness Proof

Cycles containing *fr*



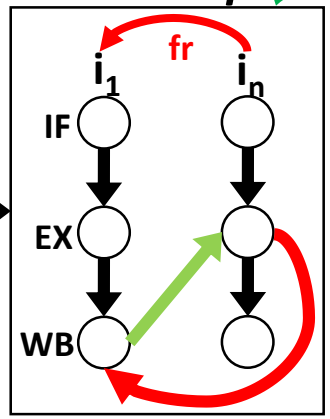
Cycles containing *po*



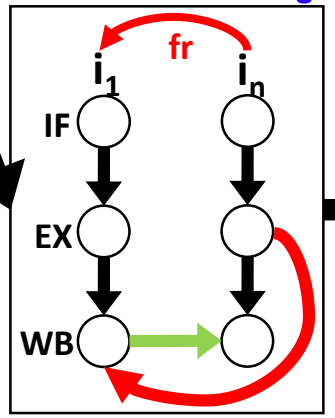
Other ISA-level cycles…

All possible transitive connections

NoDecomp ✓



AbsCounterX ?



Other transitive connections…

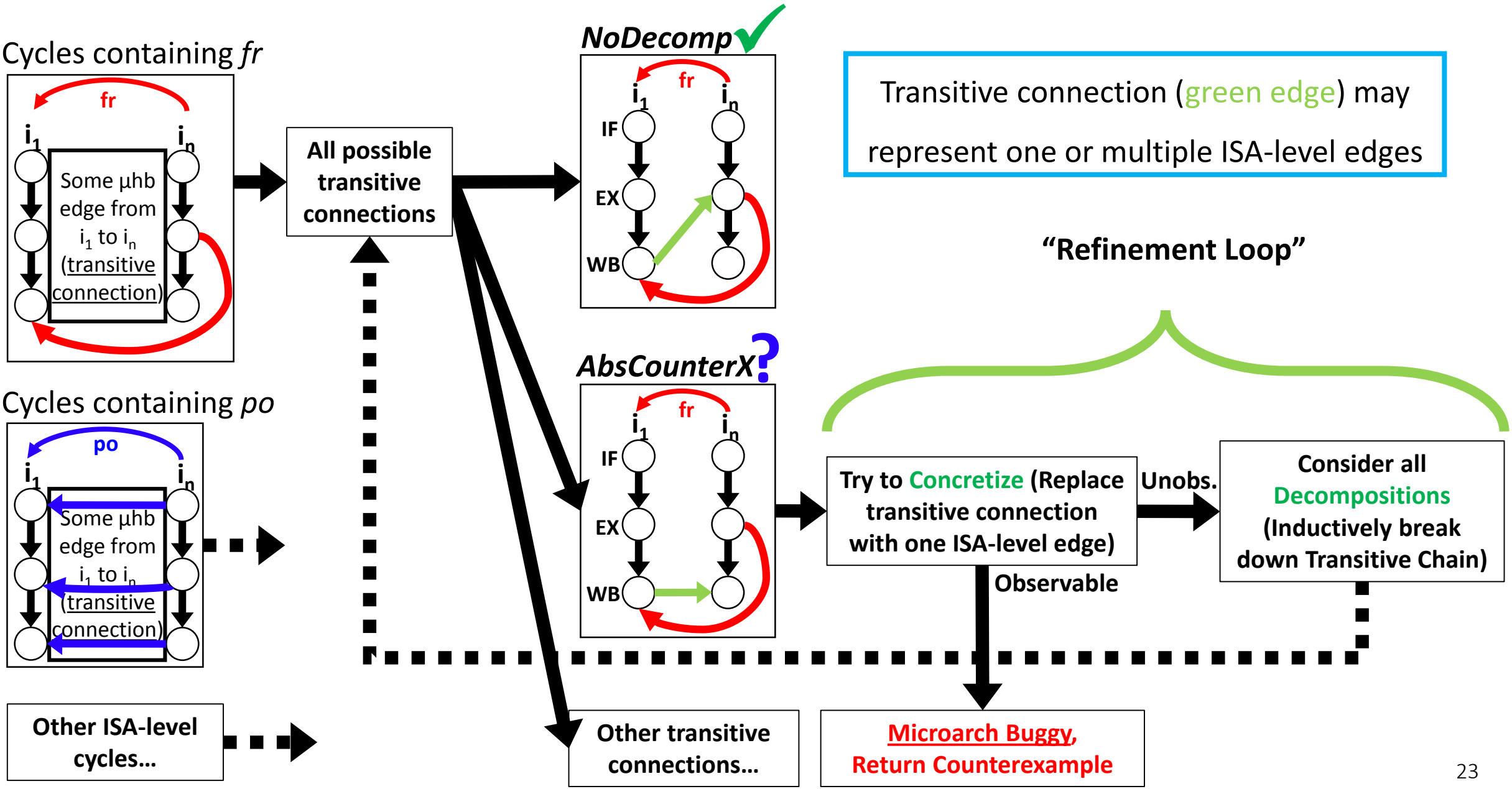
Transitive connection (green edge) may represent one or multiple ISA-level edges

Try to Concretize (Replace transitive connection with one ISA-level edge)

Unobs. → **Consider all Decompositions (Inductively break down Transitive Chain)**

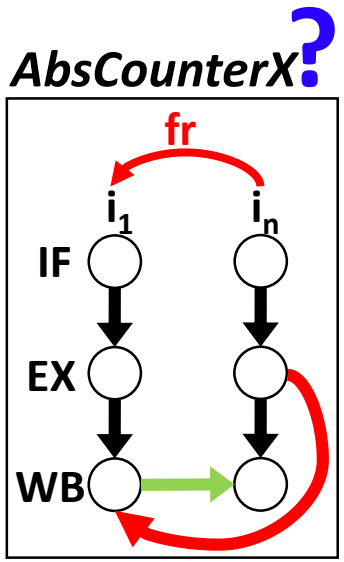
Observable → **Microarch Buggy, Return Counterexample**

Microarchitectural Correctness Proof



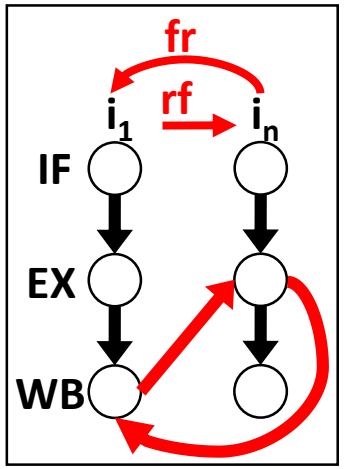
Refinement Loop: Concretization

- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)



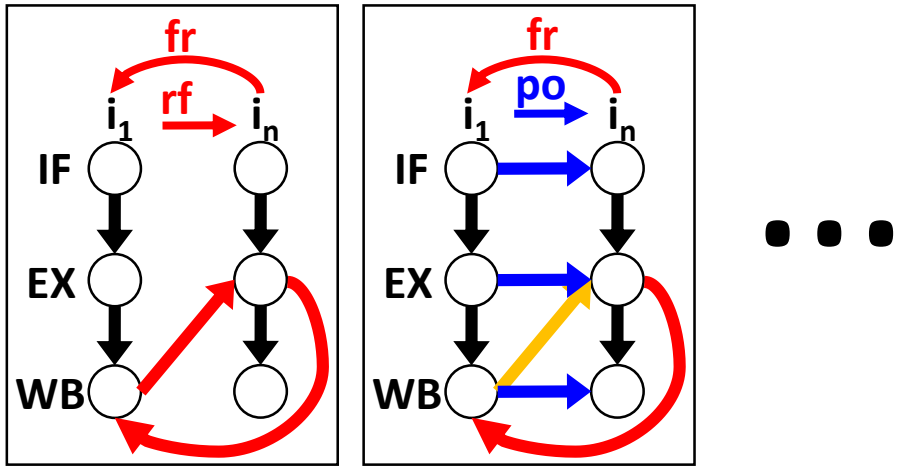
Refinement Loop: Concretization

- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)



Refinement Loop: Concretization

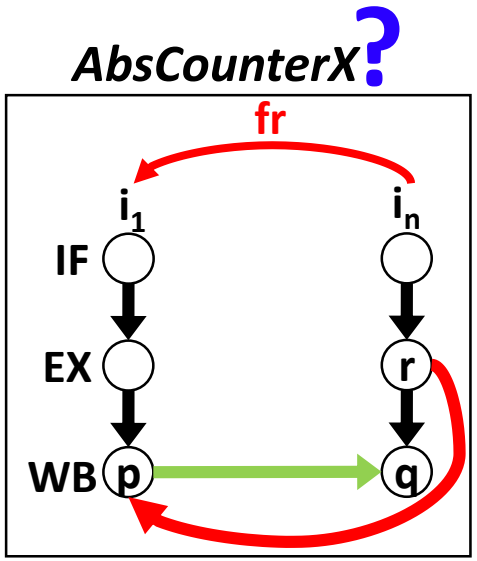
- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)



Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

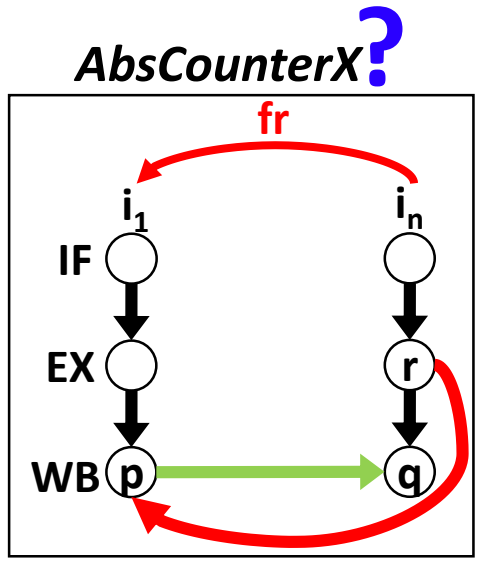
```
factorial(n)   =  factorial(n-1)  *          n
```



Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

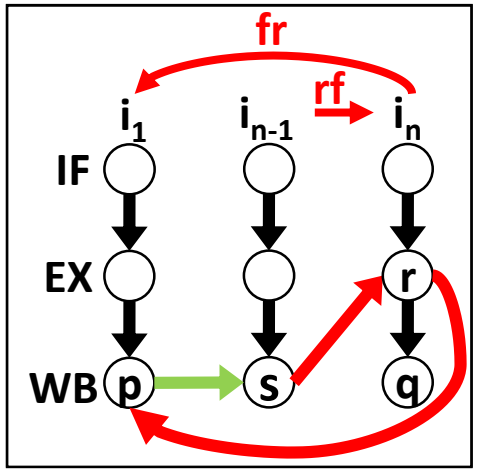
```
factorial(n)    =  factorial(n-1)  *          n
     ¦                    ¦                  ¦
Chain of length n   =  Chain of length n-1  +  “Peeled-off” edge
```



Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

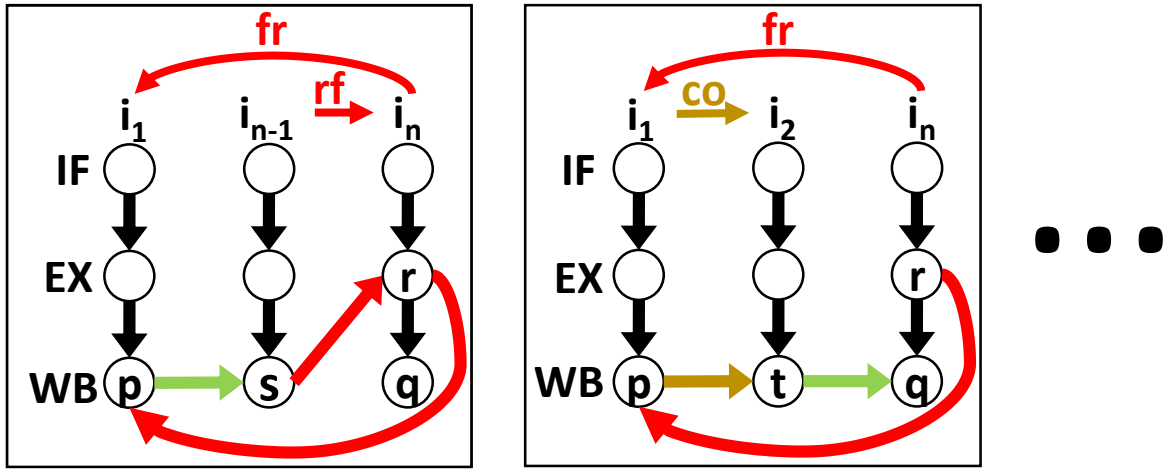
```
factorial(n)    =  factorial(n-1)  *          n
     ¦                    ¦                  ¦
Chain of length n   =  Chain of length n-1  +  “Peeled-off” edge
```



Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

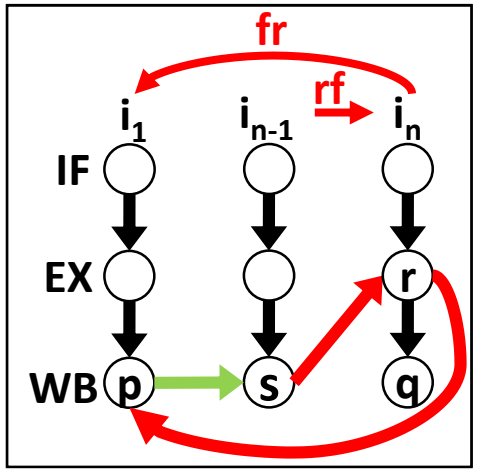
```
factorial(n)    =  factorial(n-1)  *          n
     ¦                    ¦                  ¦
Chain of length n   =  Chain of length n-1  +  “Peeled-off” edge
```

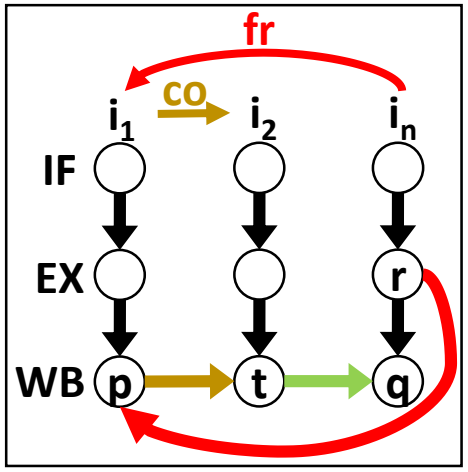


Refinement Loop: Decomposition

- Inductively break down transitive chain
  - Additional constraints may be enough to make execution unobservable

```
factorial(n)    =  factorial(n-1)  *          n
     ¦                    ¦                  ¦
Chain of length n  =  Chain of length n-1  +  "Peeled-off" edge
```





If decomposition is abstract counterexample, **repeat concretization and decomposition!**

Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO[1]) μarches
  - 3-stage in-order pipelines
- TSO verification made feasible by optimizations
  - Explicitly checking all decompositions => **case explosion**
  - **Covering Sets Optimization** (eliminate redundant transitive connections)
  - **Memoization** (eliminate previously checked ISA-level cycles)

	simpleSC	**simpleSC (w/ Covering Sets + Memoization)**
Total Time	**225.9 sec**	**19.1 sec**

	simpleTSO	**simpleTSO (w/ Covering Sets + Memoization)**
Total Time	**Timeout**	**2449.7 sec (≈ 41 mins)**

[1]TSO (Total Store Order) is the MCM of Intel x86 processors. It relaxes Store->Load ordering.

PipeProof Takeaways

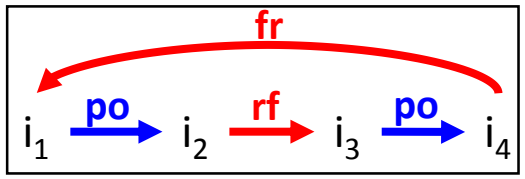
- First Ever Automated **All-Program** Microarchitectural MCM Verification
  - Designers get both **completeness** and **automation** of verification
  - Engineers can verify microarchitectures themselves, before RTL is written!
- Based on techniques from formal methods (CEGAR) [Clarke et al. CAV 2000]
- **Transitive Chain (TC) Abstraction** models infinite set of executions
- Accolades:
  - Nominated for Best Paper at MICRO 2018
  - "Honorable Mention" in 2018 IEEE Micro *Top Picks of Comp. Arch. Conferences*

Talk Outline

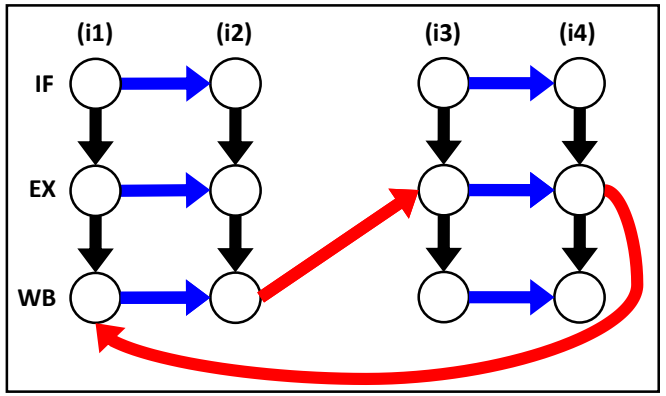
- Overview and Motivation
- Memory Consistency Background
- **PipeProof:** All-Program Microarchitectural MCM Verification
- **RTLCheck:** MCM Verification of Verilog RTL
- Expanding to other domains
- Conclusion

What if I want to verify RTL (Verilog)?

ISA-Level MCM



Microarchitectural Orderings




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

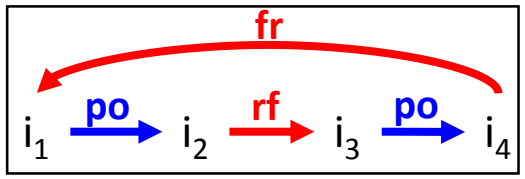
=

```
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)?

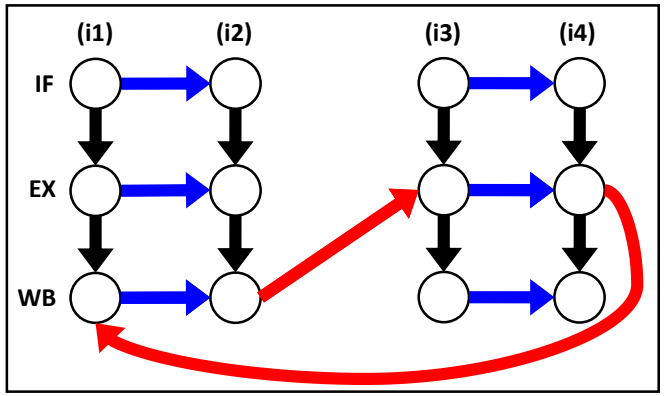
ISA-Level MCM




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

=

Microarchitectural Orderings

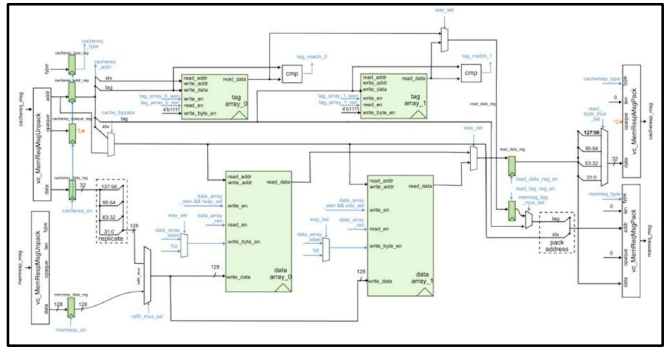



```
Axiom "PO_Fetch":
forall microop "i1", "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
      AddEdge ((i1, IF), (i2, IF)).
...
```

Verified with PipeProof

?=

RTL implementation (Verilog)

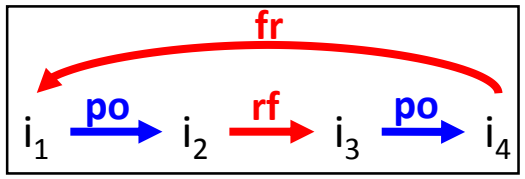


?

[RTL Image: Christopher Batten]

What if I want to verify RTL (Verilog)?

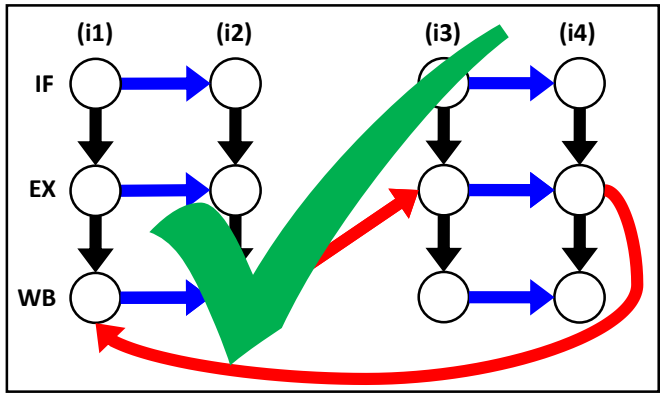
ISA-Level MCM




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

=

Microarchitectural Orderings

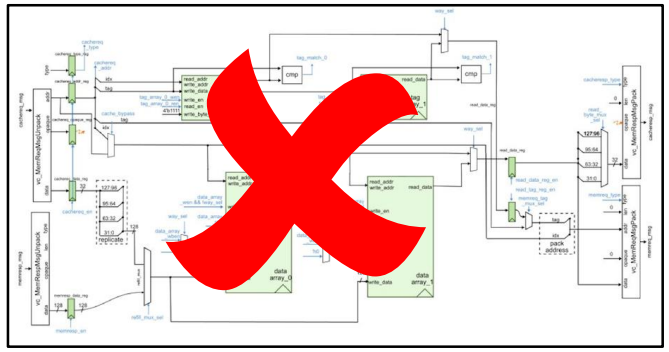



```
Axiom "PO_Fetch":
forall microop "i1", "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
      AddEdge ((i1, IF), (i2, IF)).
...
```

Verified with PipeProof

?=

RTL implementation (Verilog)

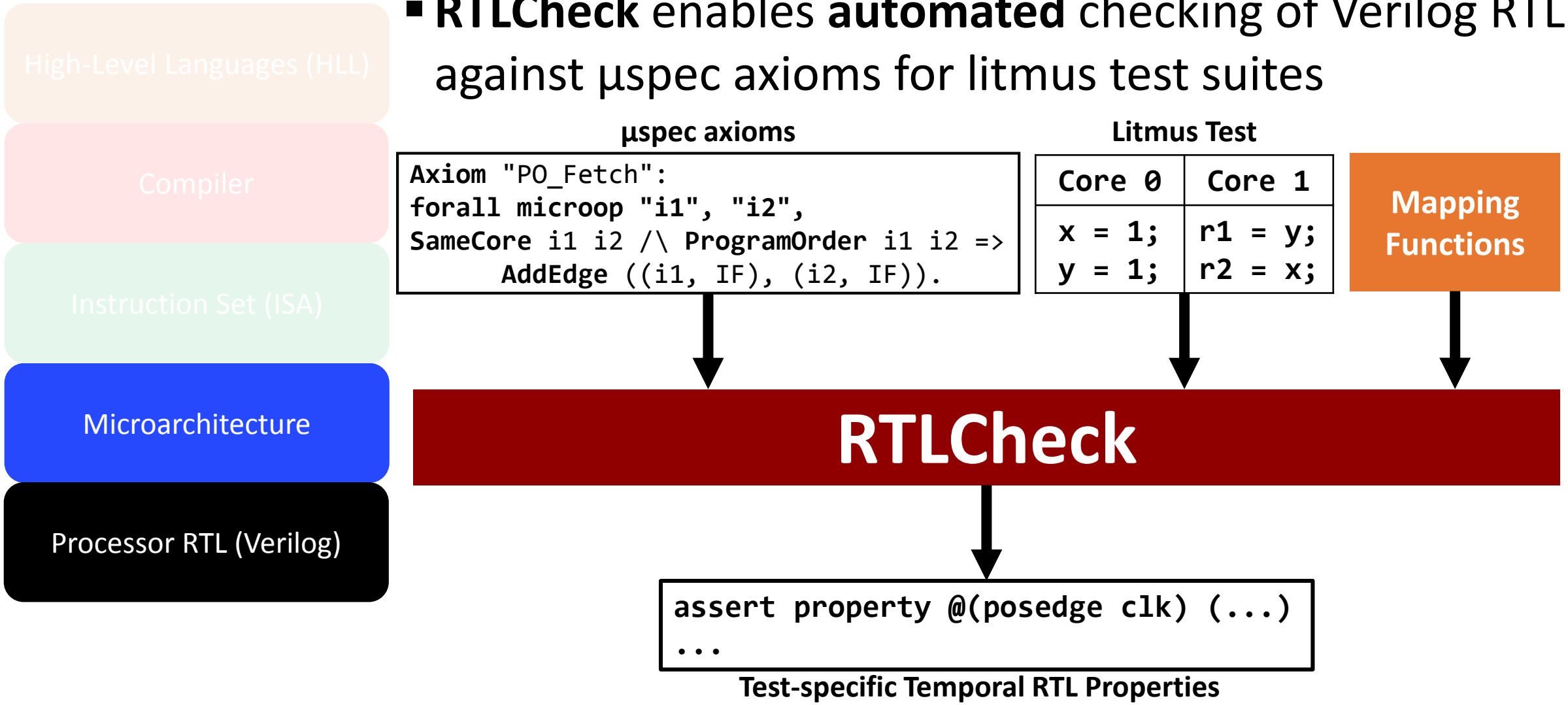


?

[RTL Image: Christopher Batten]

RTLCheck: Checking RTL Consistency Orderings

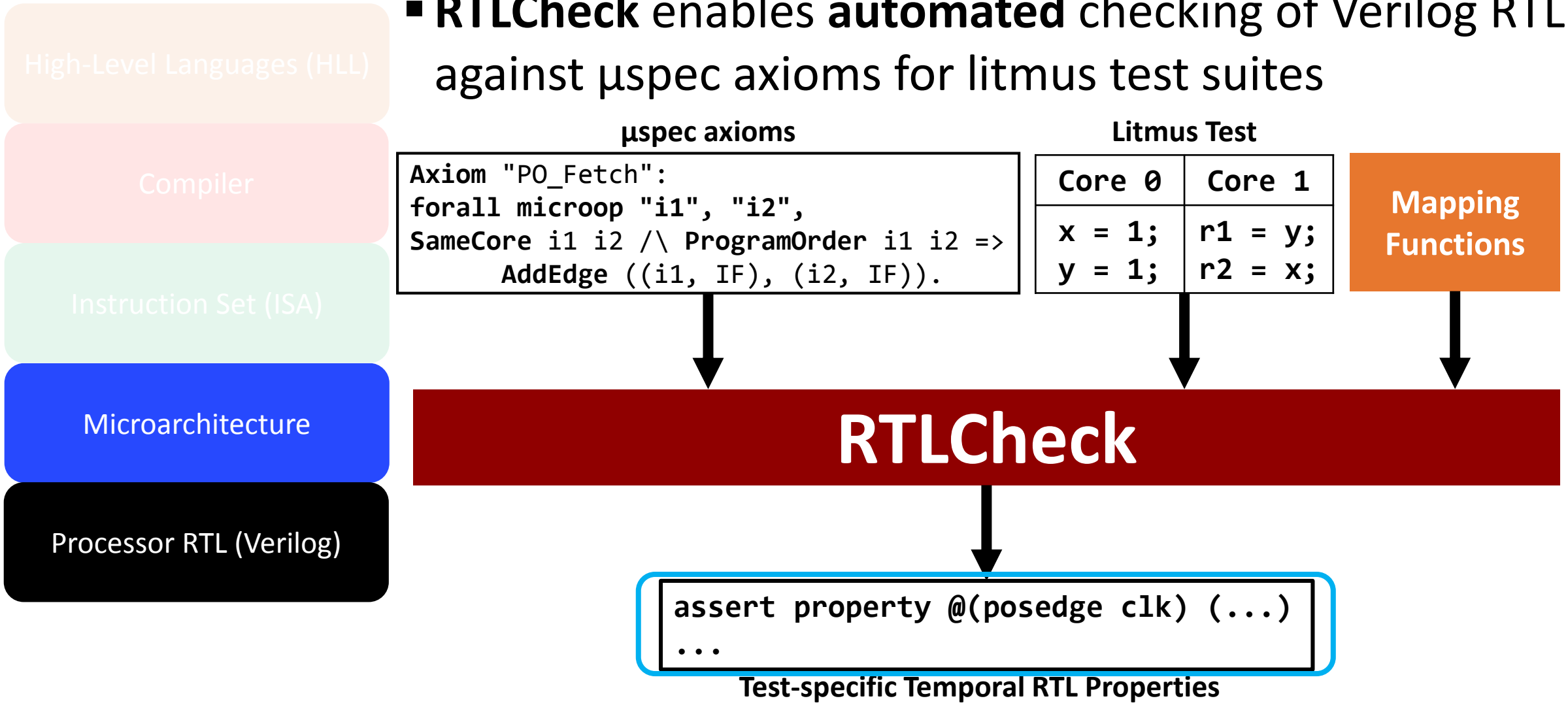
- **RTLCheck** enables **automated** checking of Verilog RTL against μspec axioms for litmus test suites



[**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

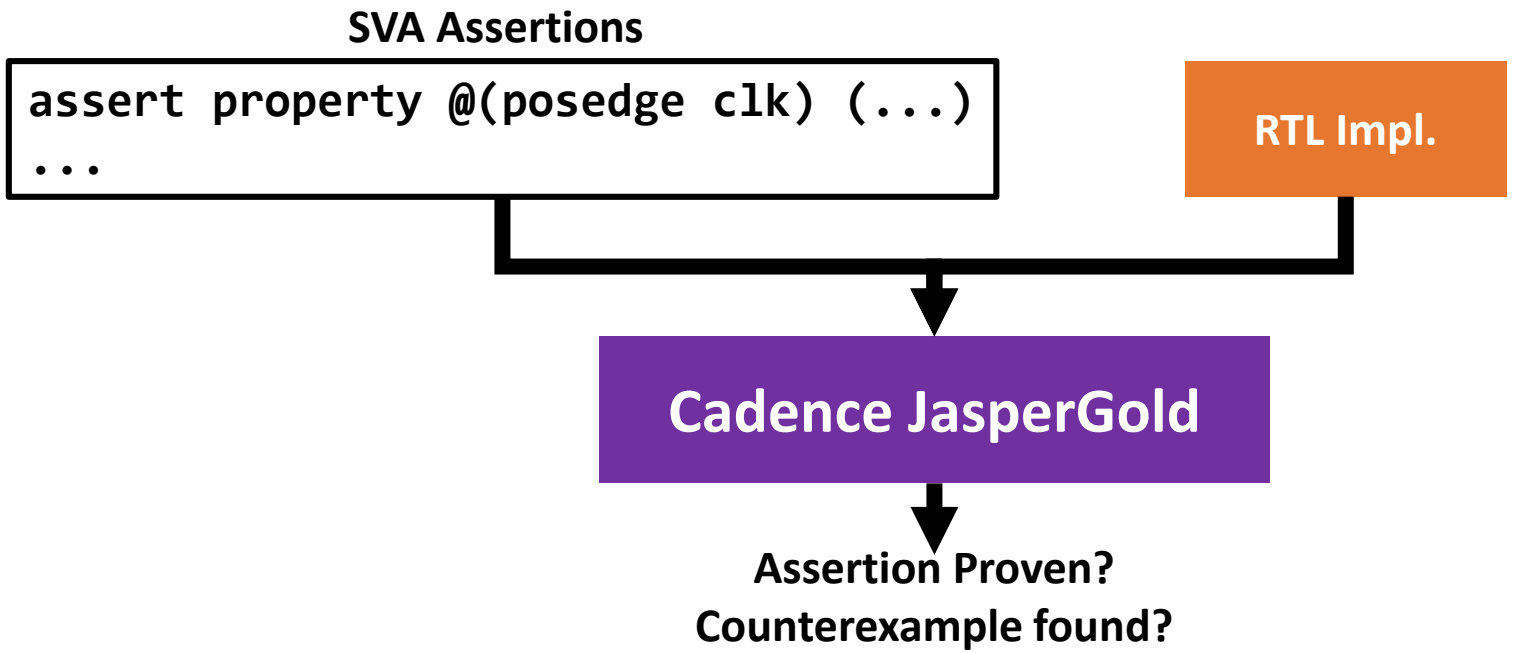
- **RTLCheck** enables **automated** checking of Verilog RTL against μspec axioms for litmus test suites



[**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

(Caution: Slippery Floor)

Meaning can be Lost in Translation!

小心地滑

(Caution: Slippery Floor)

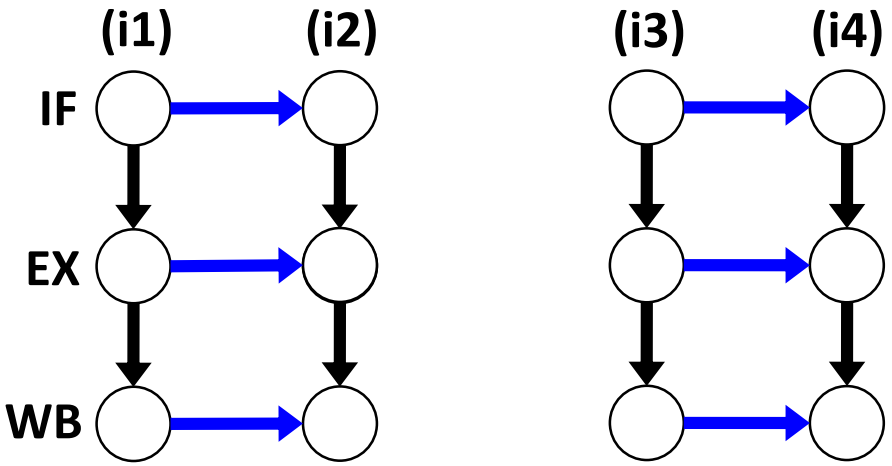
[Image: Barbara Younger]
[Inspiration: Tae Jun Ham]

The μspec/SVA Mismatch

- 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**
- Know load values, so can draw (red) edges based on these values
  - Example: `i4 reads 0 => i4 must read mem before write i1`

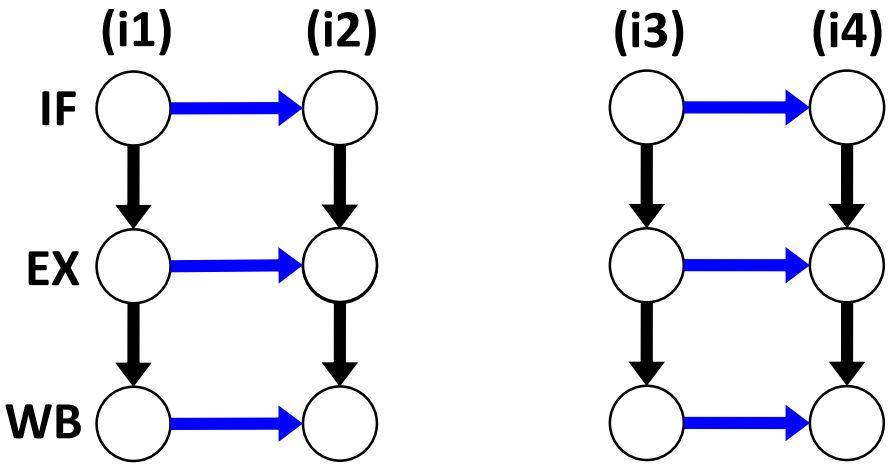


mp litmus test

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**
- Know load values, so can draw (red) edges based on these values
  - Example: `i4 reads 0 => i4 must read mem before write i1`

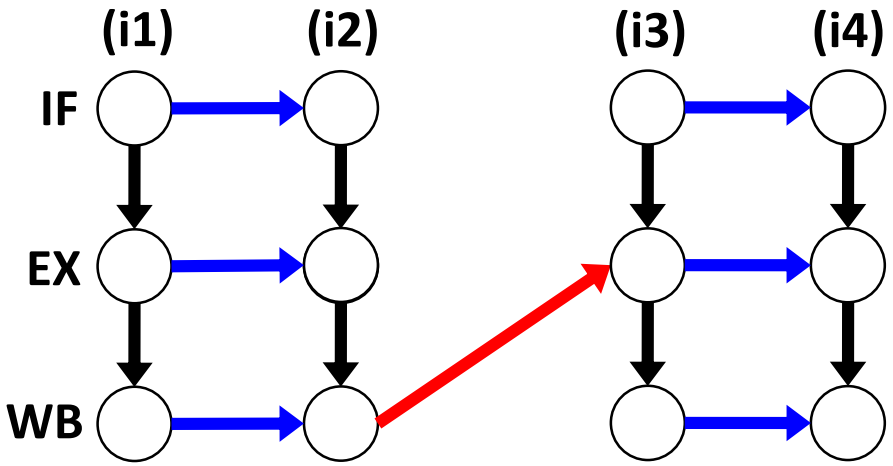


mp litmus test

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**
- Know load values, so can draw (red) edges based on these values
  - Example: `i4 reads 0 => i4 must read mem before write i1`

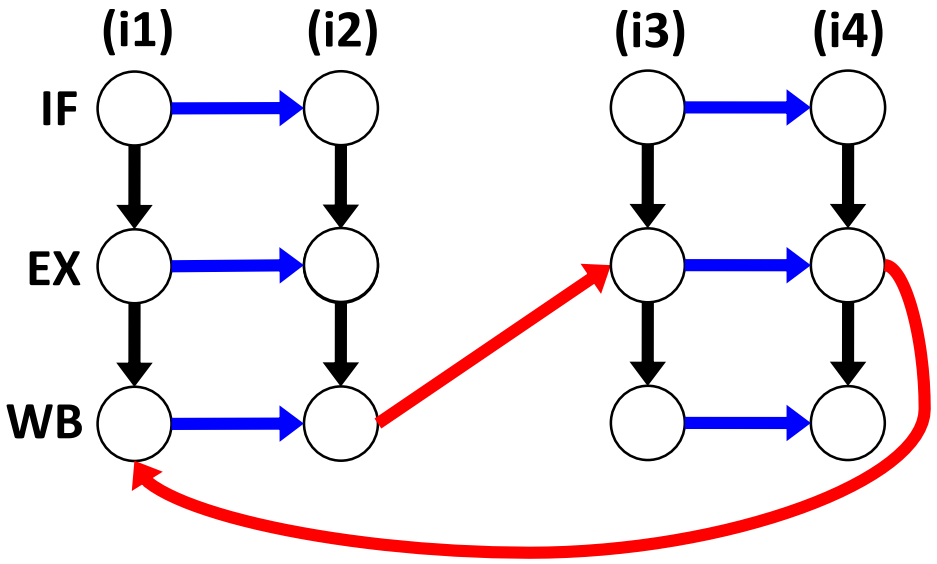


mp litmus test

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**
- Know load values, so can draw (red) edges based on these values
  - Example: `i4 reads 0 => i4 must read mem before write i1`



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 Temporal Logic

```
assume property (a); // e.g. Load i4 returns 0
assert property (b); // e.g. i4 reads mem before write i1

//The above is equivalent to...
assert property ((always a) implies (always b));
```

- In temporal logic syntax (G = always, F = eventually), this becomes:

```
G a -> G b = (~(G a)) \/ G b = (F ~a) \/ G b
```

- Assumptions introduce liveness: expensive to check! [Cerny et al. 2010]
- SVA verifiers **approximate**: only check assumptions until current state
  - This results in a property which is easier to check…
  - …but makes outcome filtering impossible with such verifiers!
- **RTLCheck Solution:** Generate properties that handle all test outcomes

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
  - Ongoing work: Formalise mismatch between μspec and SVA
- **Discovered bug** in memory implementation of RISC-V V-scale processor
- Accolades:
  - "Honorable Mention" in 2017 IEEE Micro *Top Picks of Comp. Arch. Conferences*

Talk Outline

- Overview and Motivation
- Background on MCM Specification and Verification
- **PipeProof:** All-Program Microarchitectural MCM Verification
- **RTLCheck:** MCM Verification of Verilog RTL
- Expanding to other domains
- Conclusion

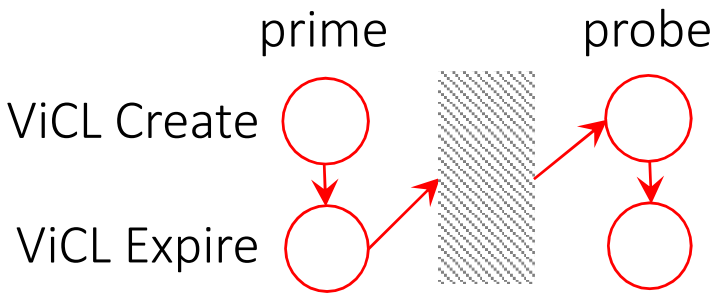
Security Analysis with CheckMate [Trippel et al. MICRO 2018]

- Work by another member of our research group (Caroline Trippel)
- Her key insight: **μhb graphs can be used for reasoning about security!**

Microarchitecture + OS Specification in Alloy

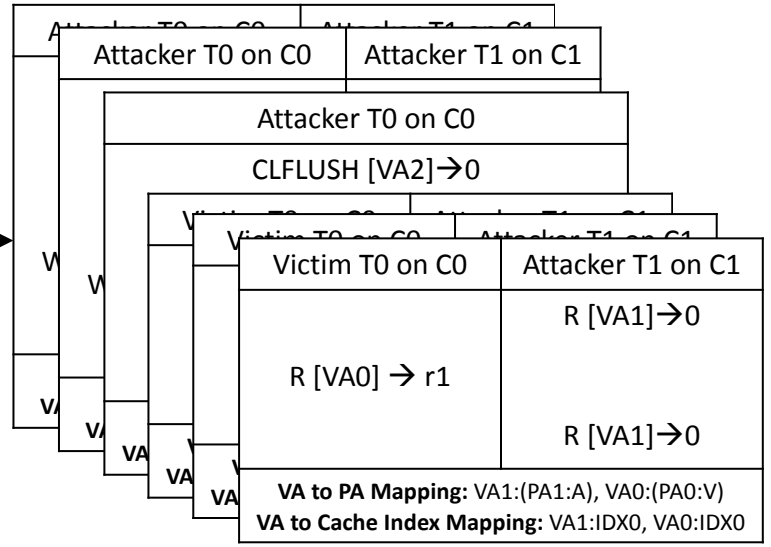
```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Exploit Pattern Specification



CheckMate Hardware Exploit Prog. Synthesis

Exploits synthesized from μhb analysis



[CheckMate: Automated Exploit Program Generation for Hardware Security Verification. Caroline Trippel, Daniel Lustig, and Margaret Martonosi. In Proceedings of the 51st International Symposium on Microarchitecture (MICRO), October 2018.]

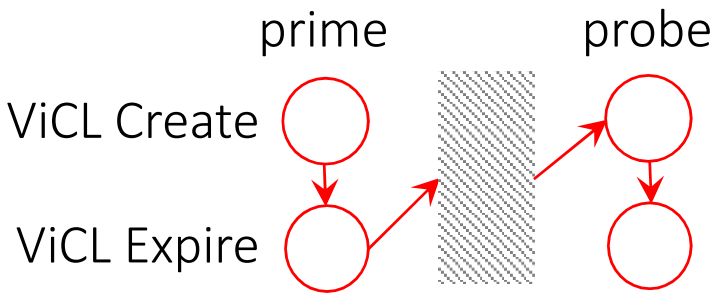
Security Analysis with CheckMate [Trippel et al. MICRO 2018]

- Work by another member of our research group (Caroline Trippel)
- Her key insight: **μhb graphs can be used for reasoning about security!**

Microarchitecture + OS Specification in Alloy

```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

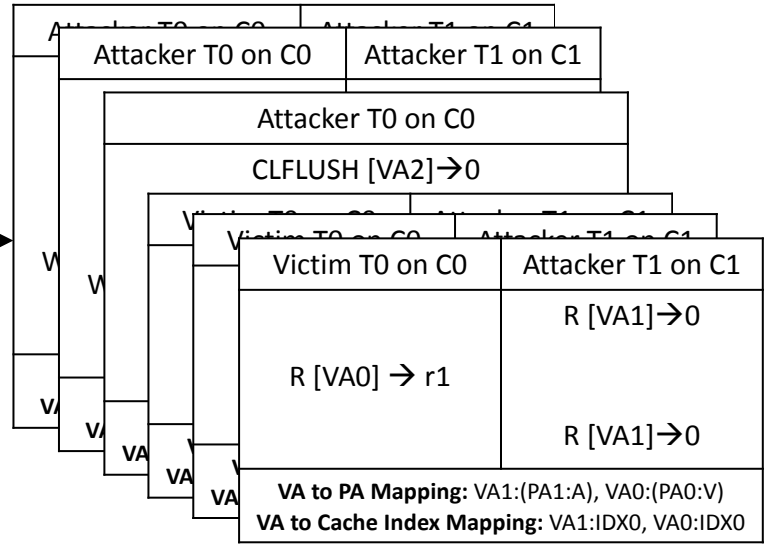
Exploit Pattern Specification



CheckMate Hardware Exploit Prog. Synthesis

Includes new exploits! (SpectrePrime, MeltdownPrime)

Exploits synthesized from μhb analysis



[CheckMate: Automated Exploit Program Generation for Hardware Security Verification. Caroline Trippel, Daniel Lustig, and Margaret Martonosi. In Proceedings of the 51st International Symposium on Microarchitecture (MICRO), October 2018.]

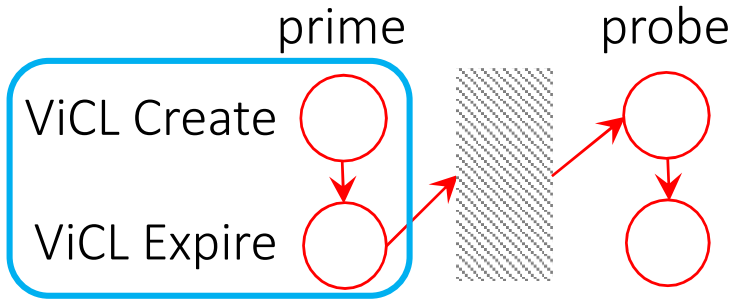
Security Analysis with CheckMate [Trippel et al. MICRO 2018]

- Work by another member of our research group (Caroline Trippel)
- Her key insight: **μhb graphs can be used for reasoning about security!**

Microarchitecture + OS Specification in Alloy

```
fact Program_Order_Fetch {
  all disj e0, e1 : Event |
  ProgramOrder[e0, e1] =>
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter]
}

fact In_Order_Decode {
  all disj e0, e1 : Event |
  EdgeExists[e0, Fetch, e1, Fetch, uhb_inter] =>
  EdgeExists[e0, Decode, e1, Decode, uhb_inter]
}
```

Exploit Pattern Specification

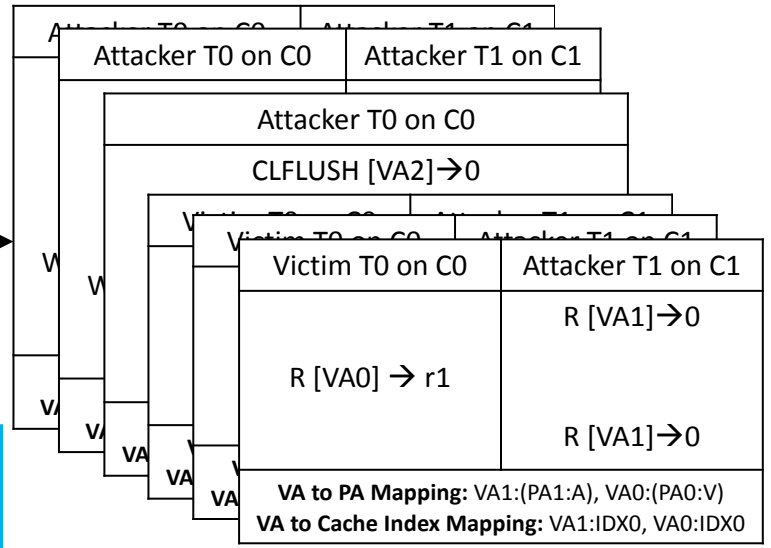


CheckMate Hardware Exploit Prog. Synthesis

ViCL abstraction [Manerkar et al. MICRO 2015] used to model cache behaviour

Includes new exploits! (SpectrePrime, MeltdownPrime)

Exploits synthesized from μhb analysis



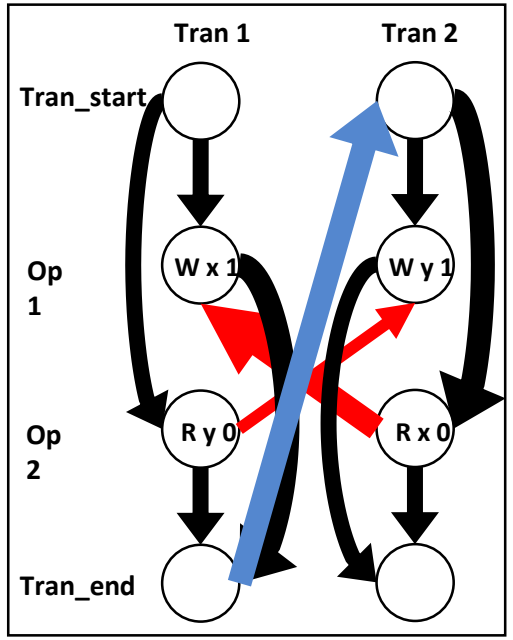
[CheckMate: Automated Exploit Program Generation for Hardware Security Verification. Caroline Trippel, Daniel Lustig, and Margaret Martonosi. In Proceedings of the 51st International Symposium on Microarchitecture (MICRO), October 2018.]

Ongoing Work: Verifying Distributed Systems

- Joint work with Themis Melissaris
- Distributed systems have some similarities to shared-memory systems
  - Distributed protocols (e.g. Paxos) similar to **cache coherence protocols**
  - Replicated data store consistency models similar to **MCMs**

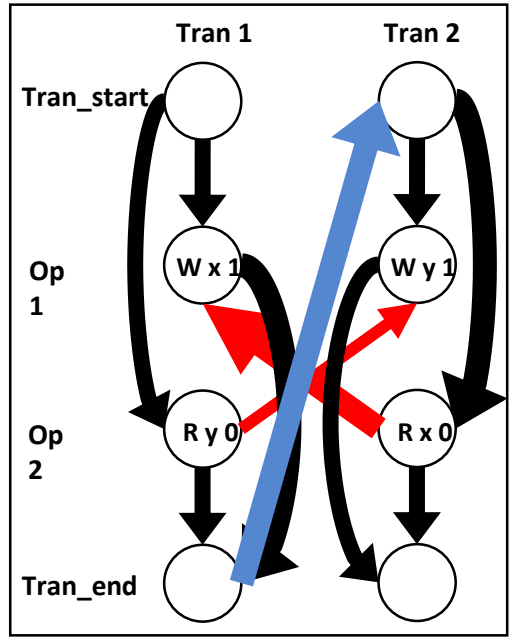
Ongoing Work: Verifying Distributed Systems

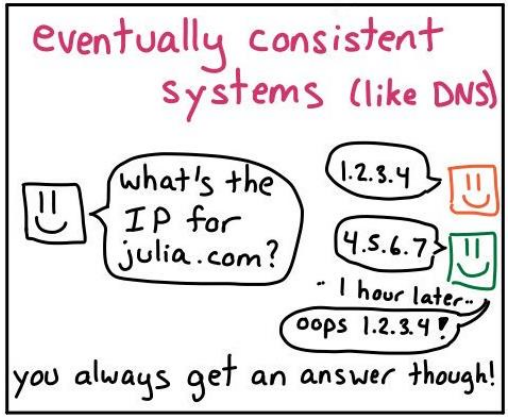
- Joint work with Themis Melissaris
- Distributed systems have some similarities to shared-memory systems
  - Distributed protocols (e.g. Paxos) similar to **cache coherence protocols**
  - Replicated data store consistency models similar to **MCMs**



Ongoing Work: Verifying Distributed Systems

- Joint work with Themis Melissaris
- Distributed systems have some similarities to shared-memory systems
  - Distributed protocols (e.g. Paxos) similar to **cache coherence protocols**
  - Replicated data store consistency models similar to **MCMs**
- Also have features with no shared-memory analogue!
  - Correctness in the presence of node failures
  - Eventual consistency [Vogels CACM 2009]





[Cartoon by Julia Evans]

Talk Outline

- Overview and Motivation
- Background on MCM Specification and Verification
- **PipeProof:** All-Program Microarchitectural MCM Verification
- **RTLCheck:** MCM Verification of Verilog RTL
- Expanding to other domains
- Conclusion

Conclusions

- **Complexity of computing hardware is increasing**
  - Ubiquitous parallelism and increased heterogeneity
- **Automated formal verification helps engineers handle this complexity**
  - Give engineers the ability to formally verify their systems themselves
  - **PipeProof**: Automated All-Program Microarchitectural MCM Verification
  - **RTLCheck**: Per-Program MCM Verification of RTL Designs
- **Techniques for MCM analysis applicable to other domains**
  - e.g. Security [Trippel et al. MICRO 2018] and distributed systems

Collaborators

Margaret Martonosi

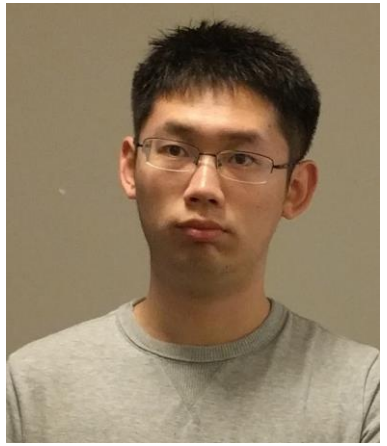
Daniel Lustig
(NVIDIA)

Aarti Gupta

Michael Pelluaer
(NVIDIA)

Caroline Trippel

Sharad Malik



Hongce Zhang

Automated Formal Memory Consistency Verification of Hardware

Yatin A. Manerkar

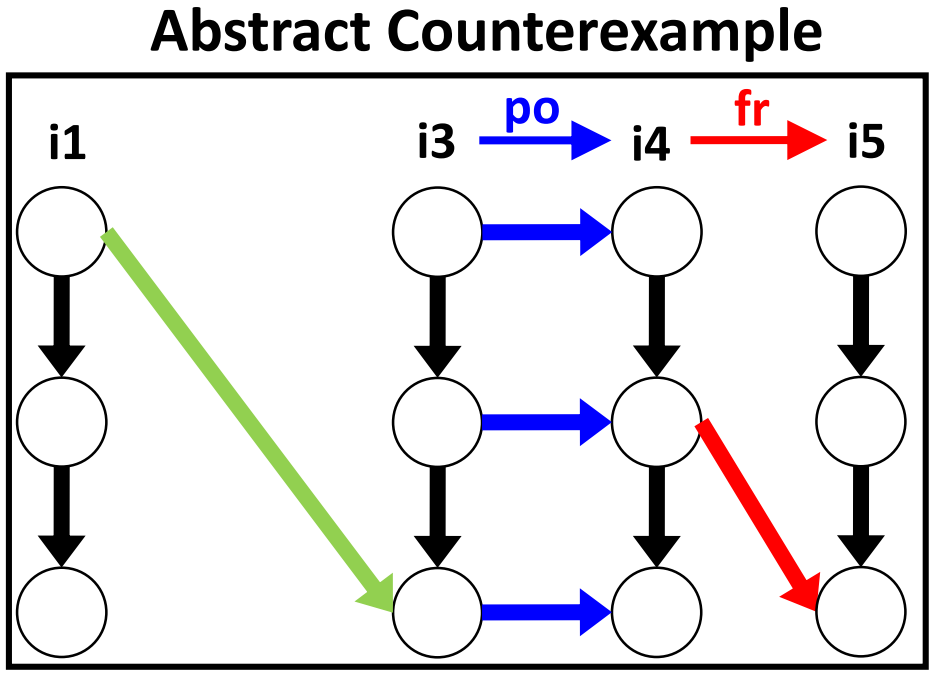
Princeton University

June 23rd, 2019

Backup Slides

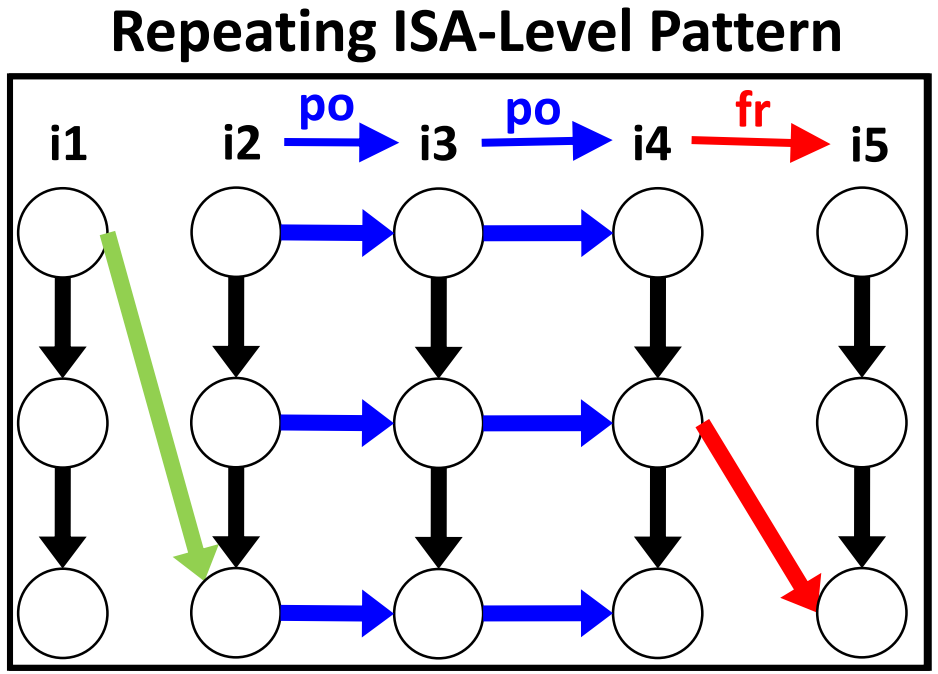
Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- **Inductively proven by PipeProof before their use in proof algorithms**
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)



Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- **Inductively proven by PipeProof before their use in proof algorithms**
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)

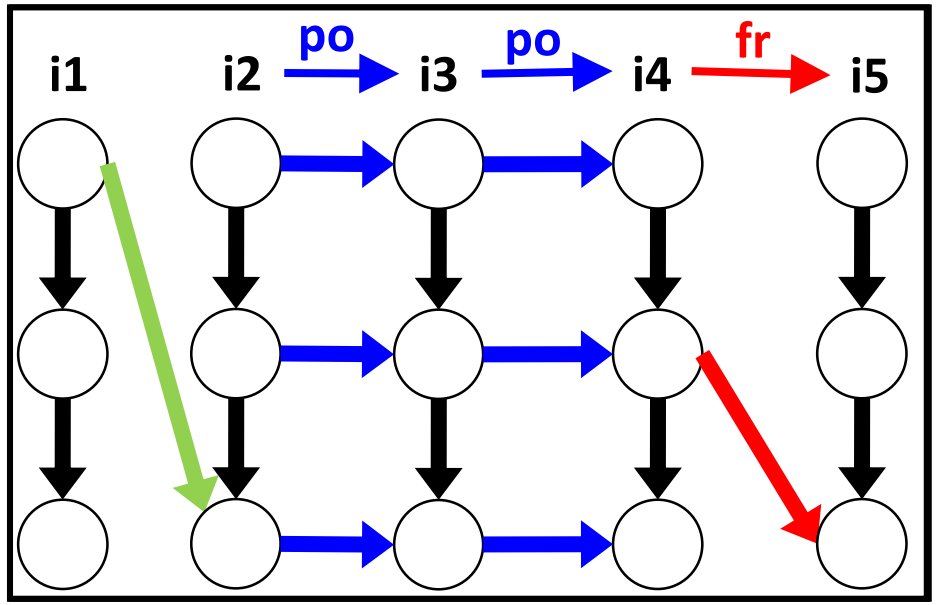


Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- **Inductively proven by PipeProof before their use in proof algorithms**
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)

Can continue decomposing in this way forever!

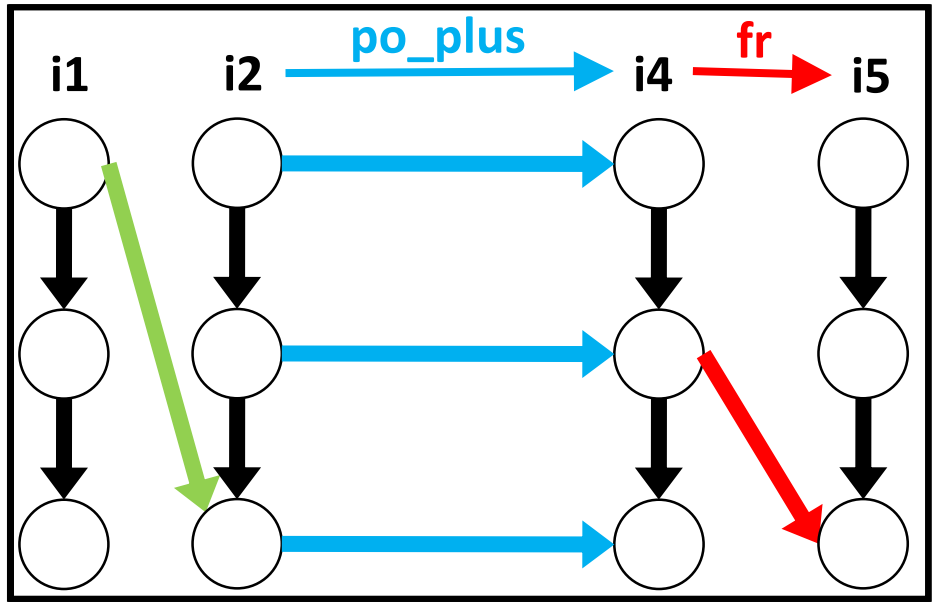
Repeating ISA-Level Pattern



Chain Invariants

- Abstractly represent repeated ISA-level patterns
- Sometimes needed for refinement loop to terminate
- **Inductively proven by PipeProof before their use in proof algorithms**
- **Example:** checking for edge from i1 to i5 (TC abstraction support proof)

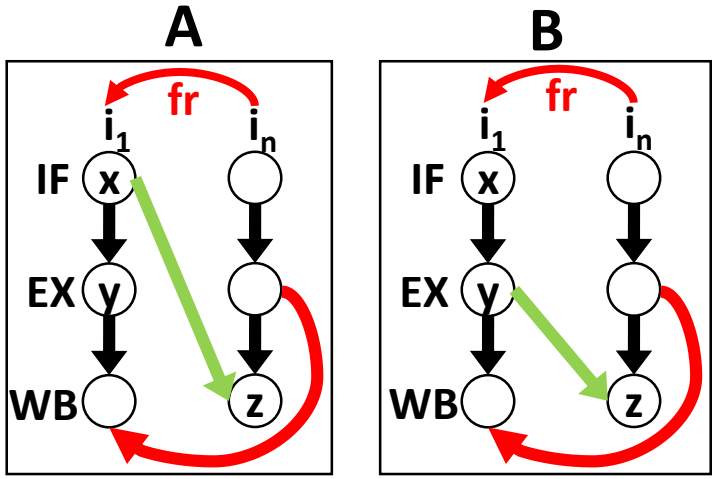
Chain Invariant Applied



-**po_plus** = arbitrary number of repetitions of **po**
-Next edge peeled off will be something other than **po**

Covering Sets Optimization

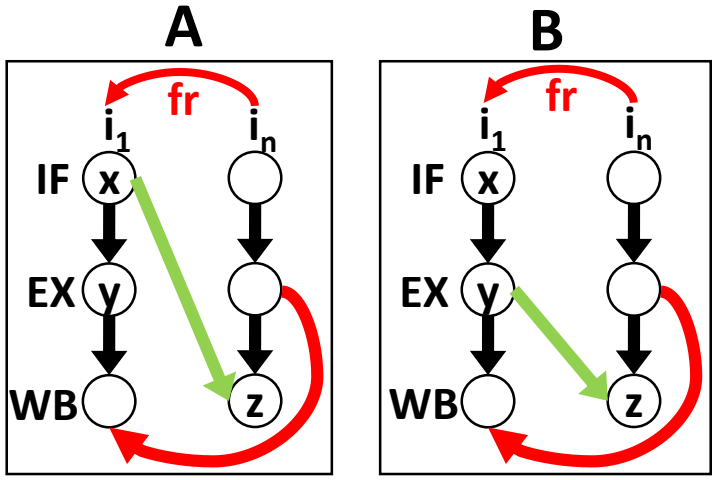
- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections



Covering Sets Optimization

- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

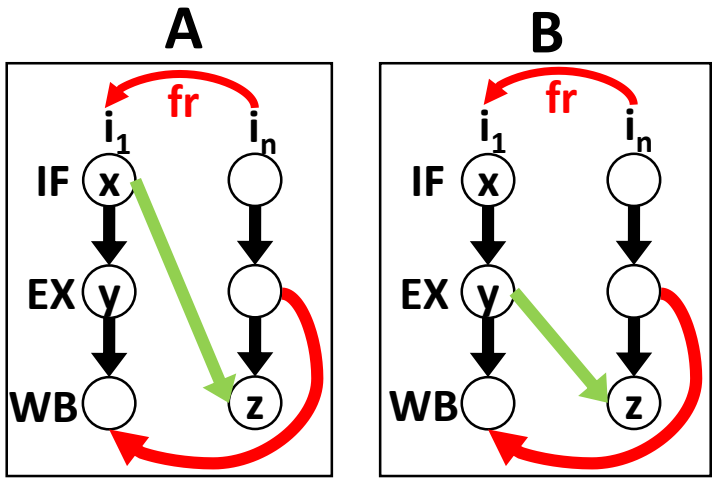
Graph A has an edge from x→z (tran conn.)



Covering Sets Optimization

- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

Graph A has an edge from x→z (tran conn.)

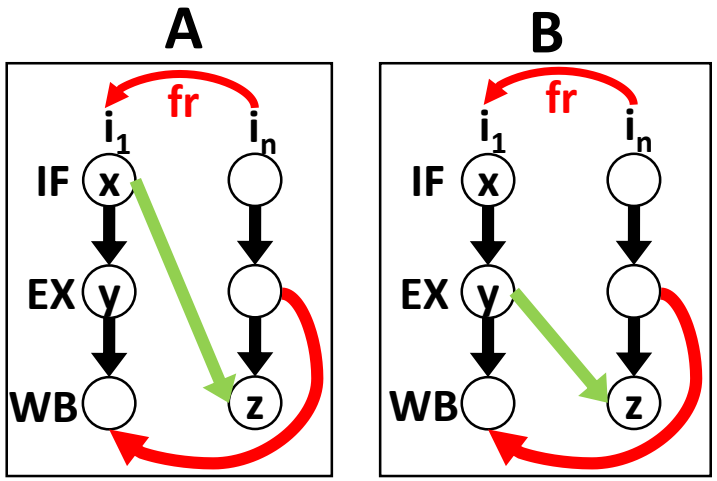


Graph B has edges from y→z (tran conn.) and x→z (by transitivity)

Covering Sets Optimization

- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

Graph A has an edge from x→z (tran conn.)



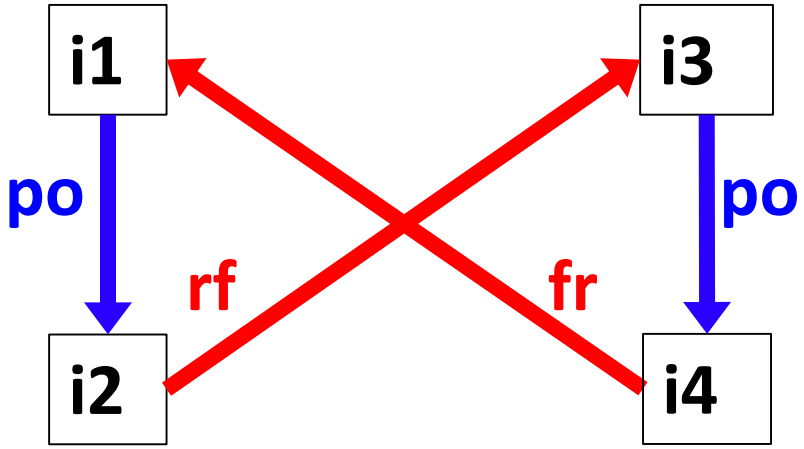
Graph B has edges from y→z (tran conn.) and x→z (by transitivity)

Correctness of A => Correctness of B (since B contains A's tran conn.)

Checking B explicitly is redundant!

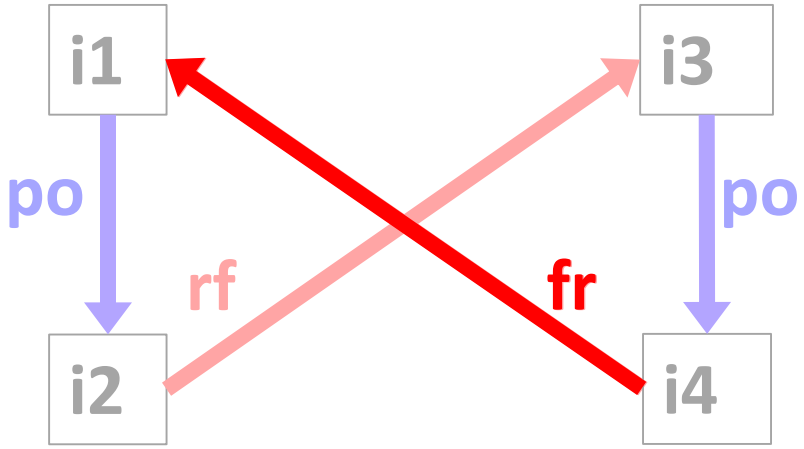
Memoization Optimization

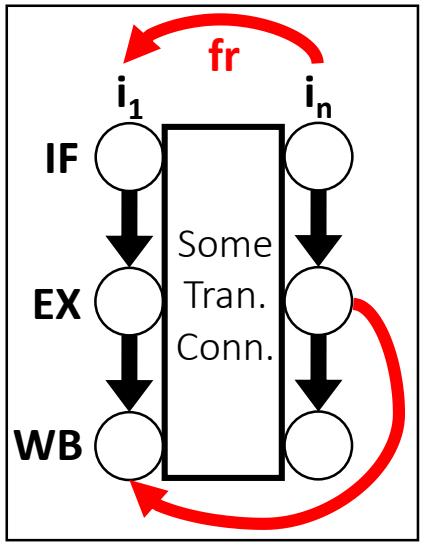
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified



Memoization Optimization

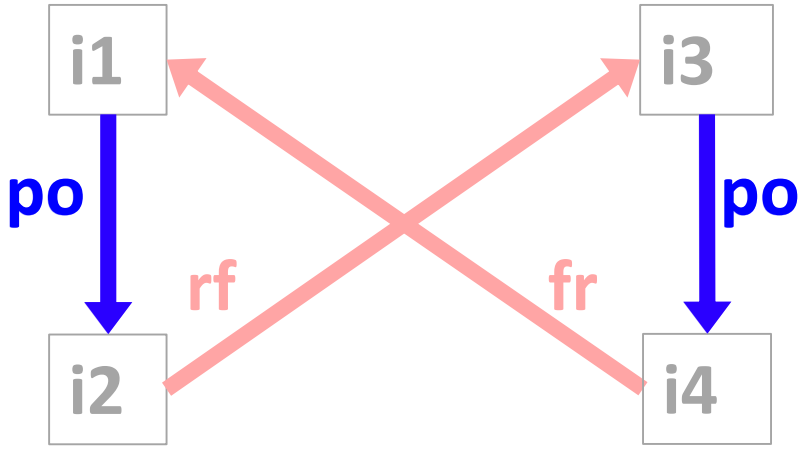
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

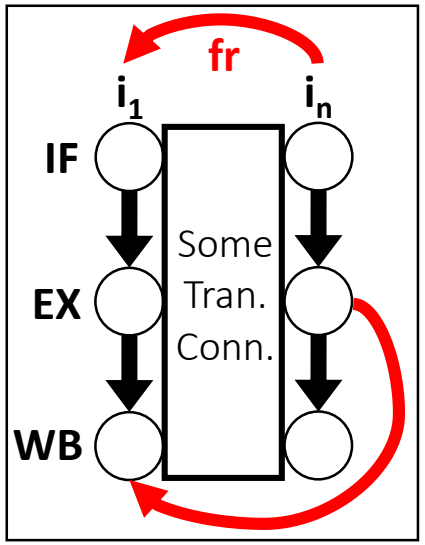


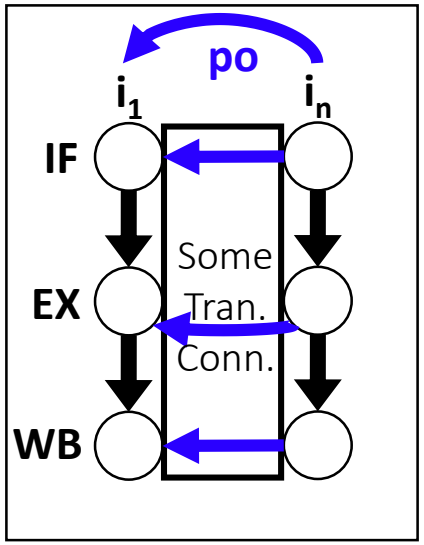


Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

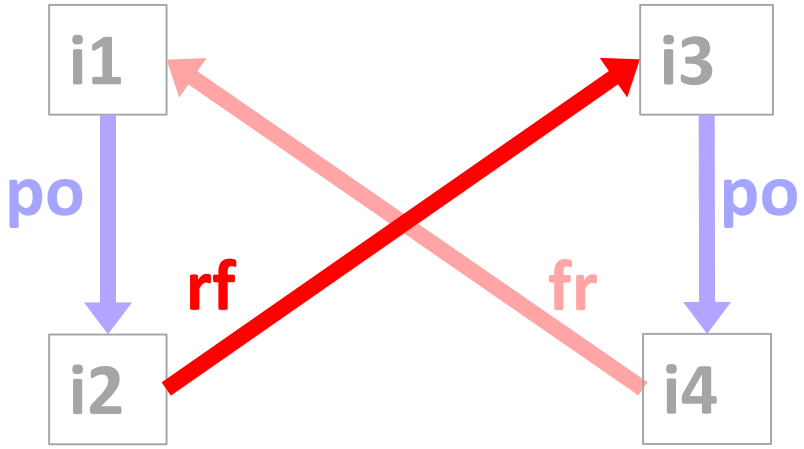




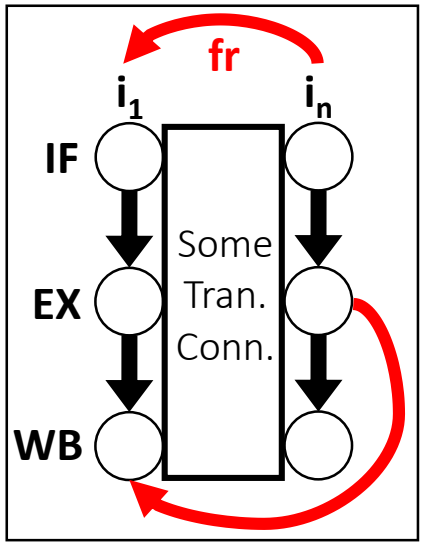


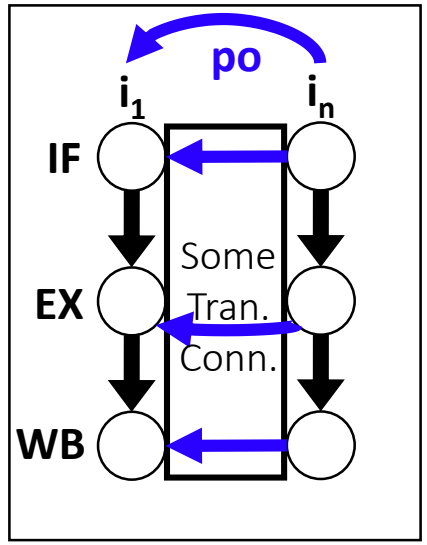
Memoization Optimization

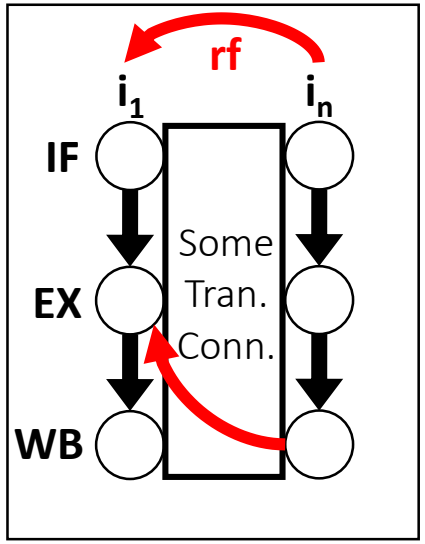
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified



Same cycle is checked 3 times!

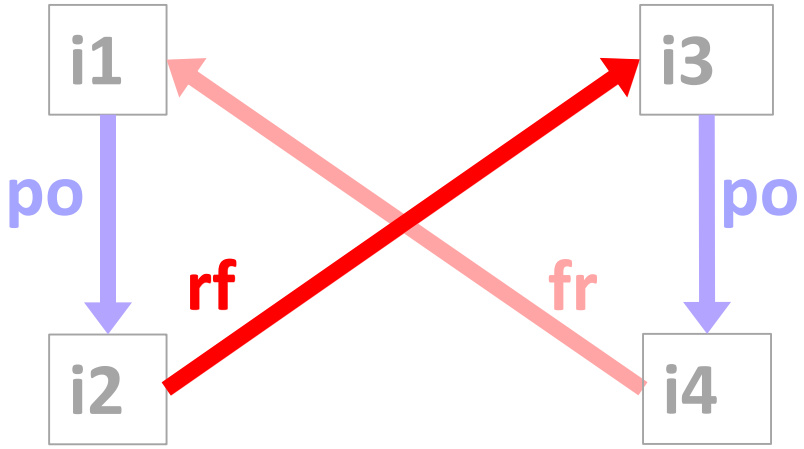


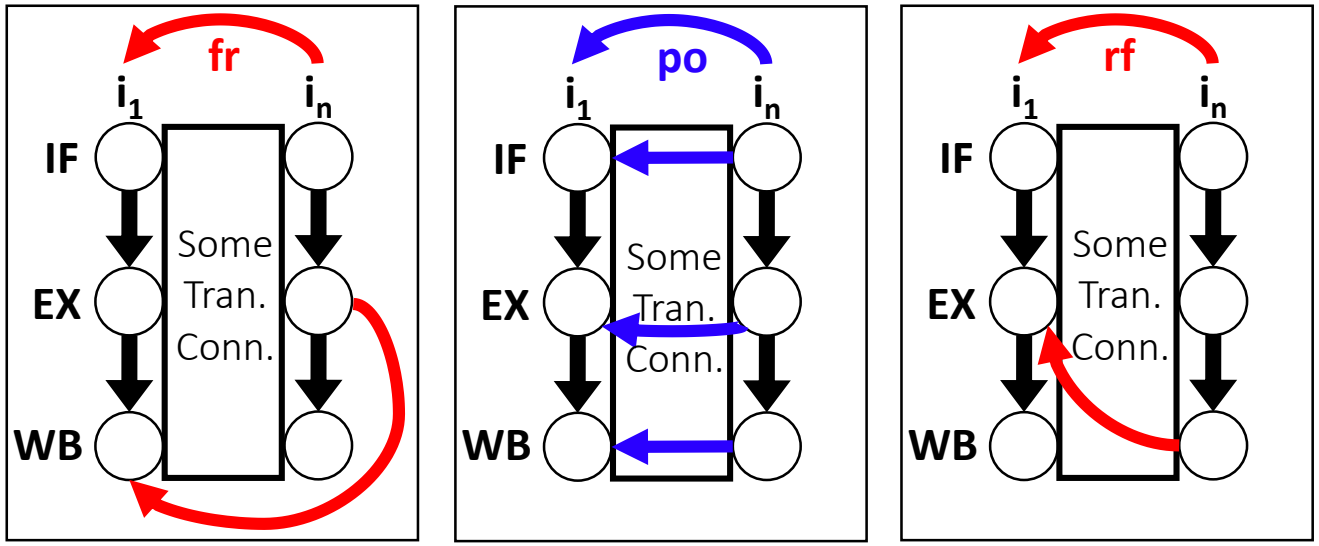




Memoization Optimization

- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified



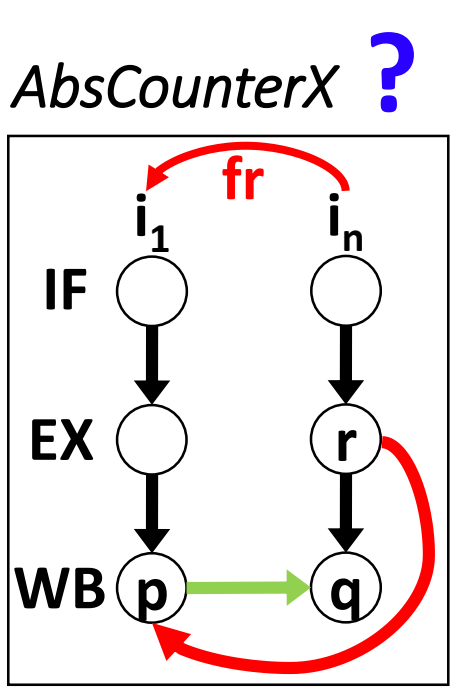


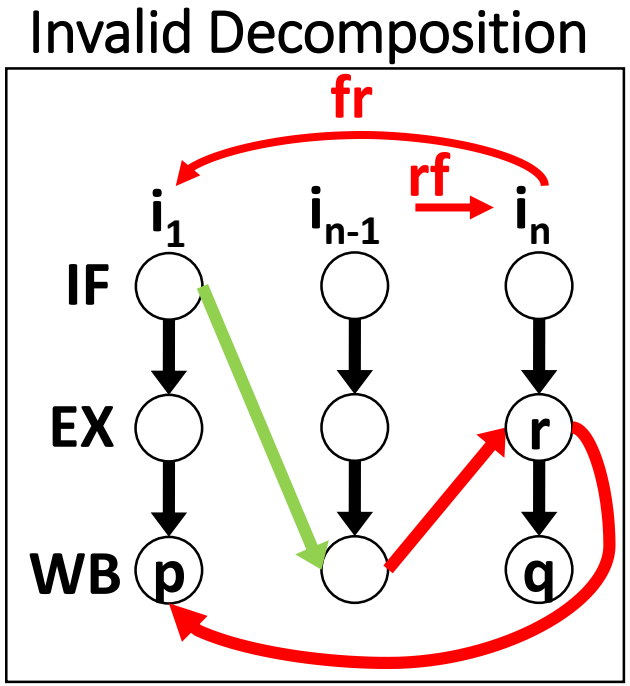
Same cycle is checked 3 times!

<u>Procedure:</u> If all ISA-level cycles containing edge r_i have been checked, do not peel off r_i edges when checking subsequent cycles

Filtering Invalid Decompositions

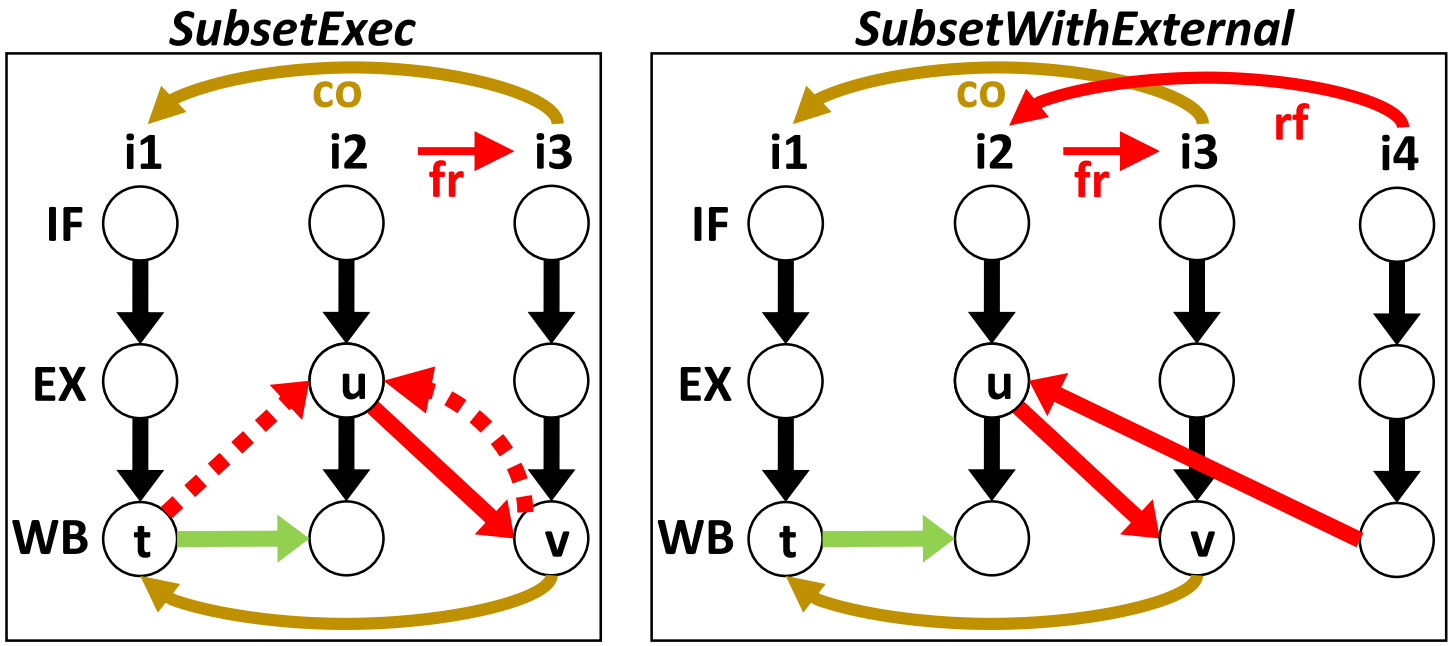
- When decomposing a transitive connection, the decomposition should guarantee the transitive connections of its parent abstract cexes.
- Decompositions that do not do this are invalid and filtered out



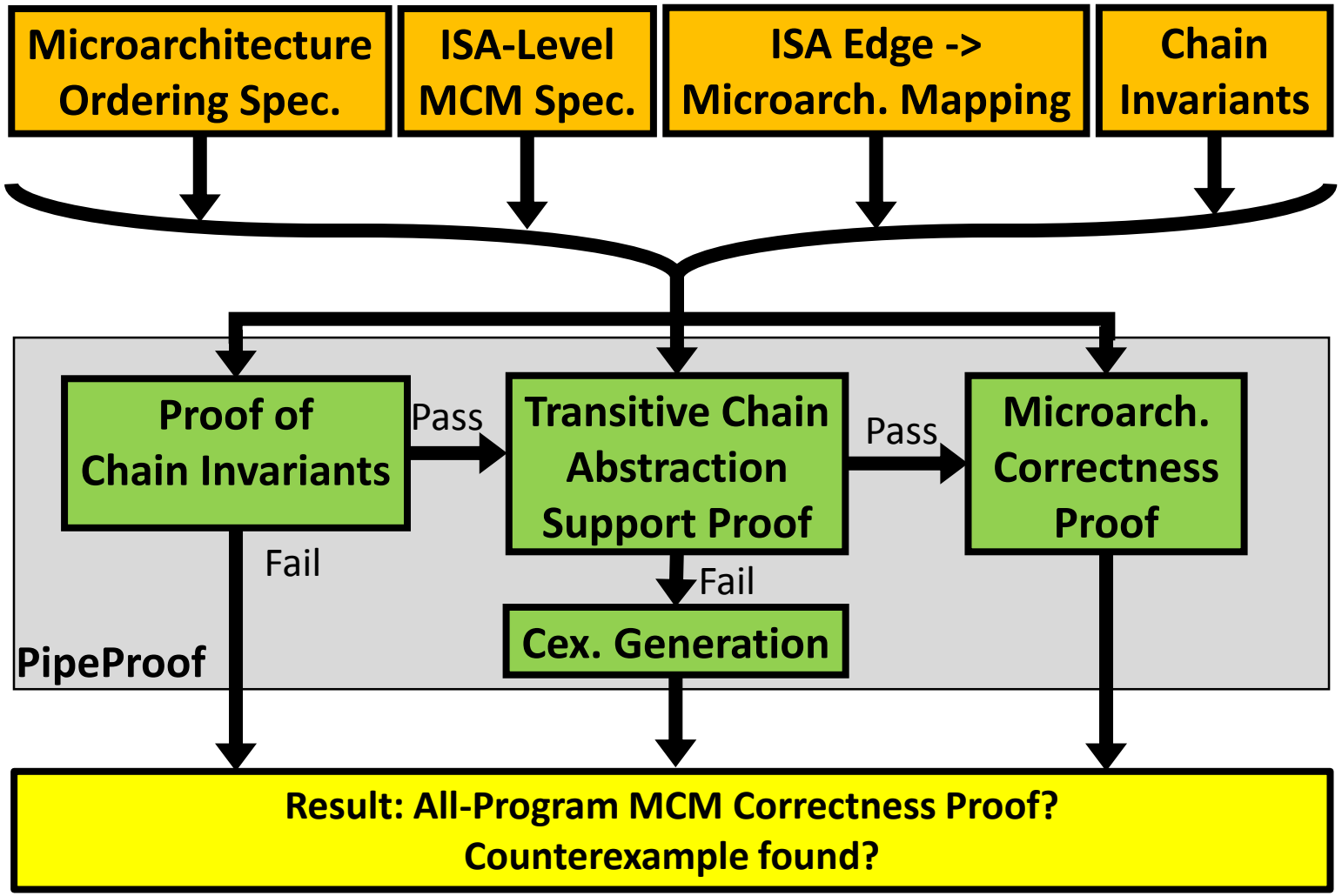


The Adequate Model Over-Approximation

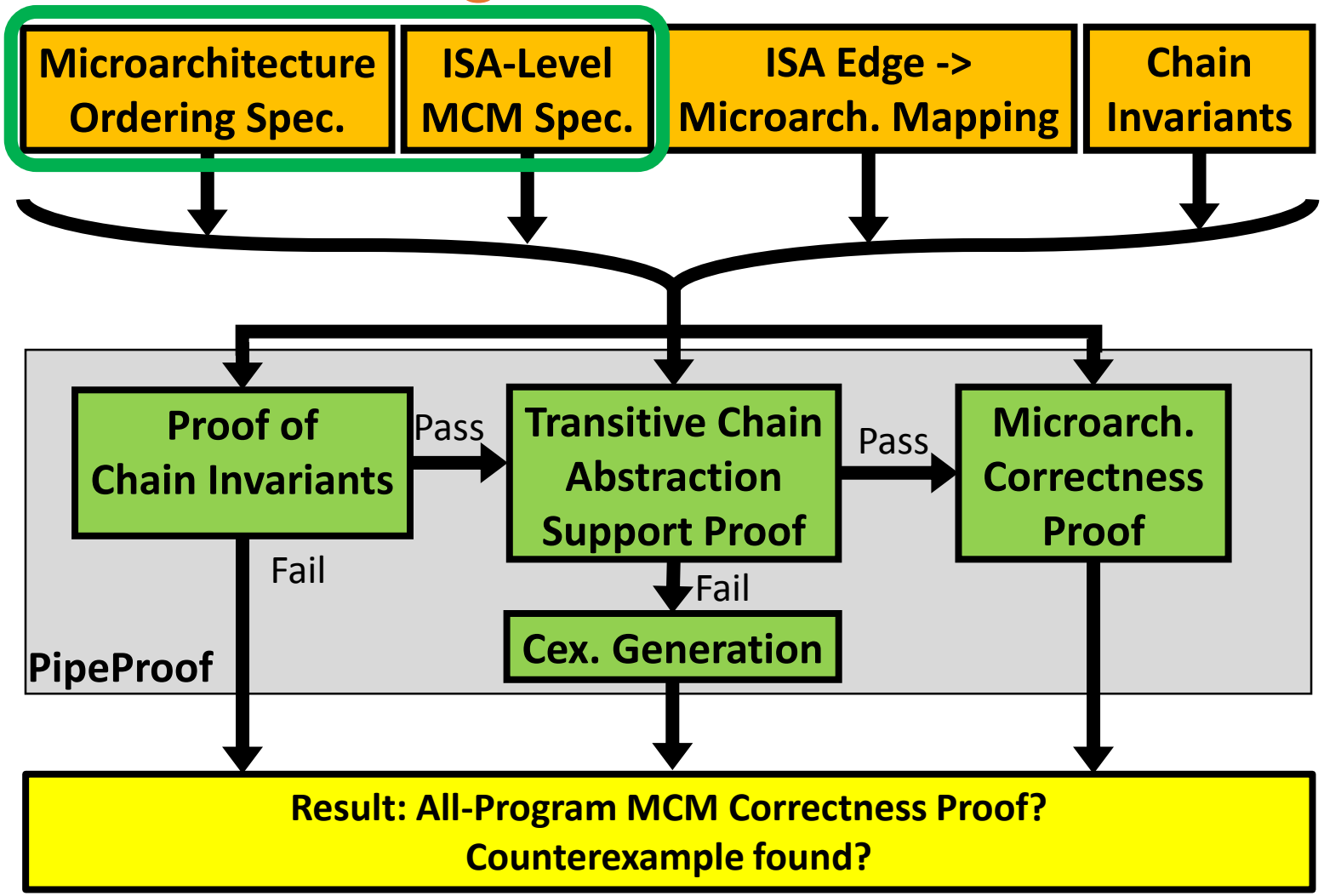
- Addition of an instruction can make unobservable execution observable!
- Need to work with over-approximation of microarchitectural constraints
- PipeProof sets all `exists` clauses to true as its over-approximation



PipeProof Block Diagram

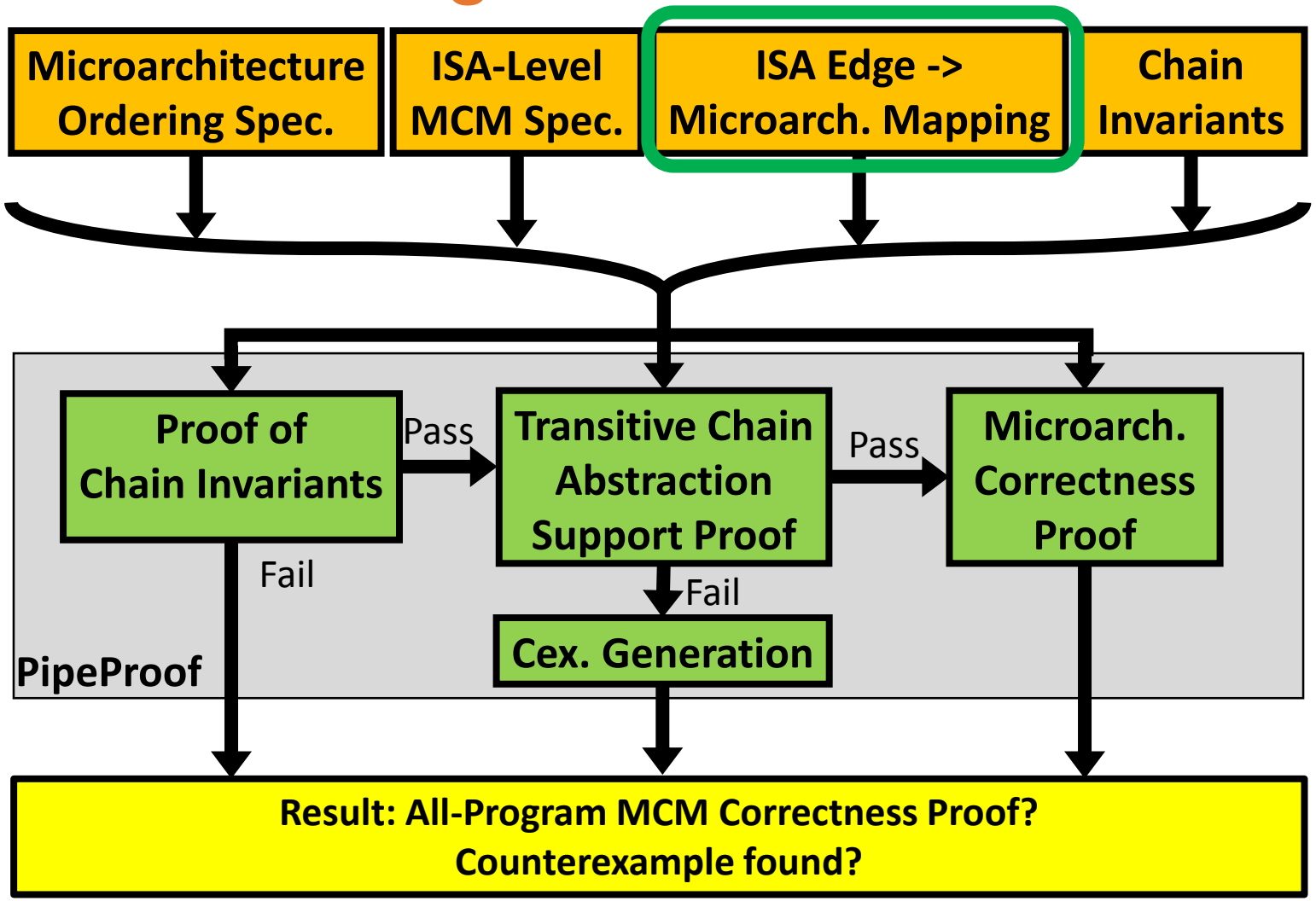


PipeProof Block Diagram

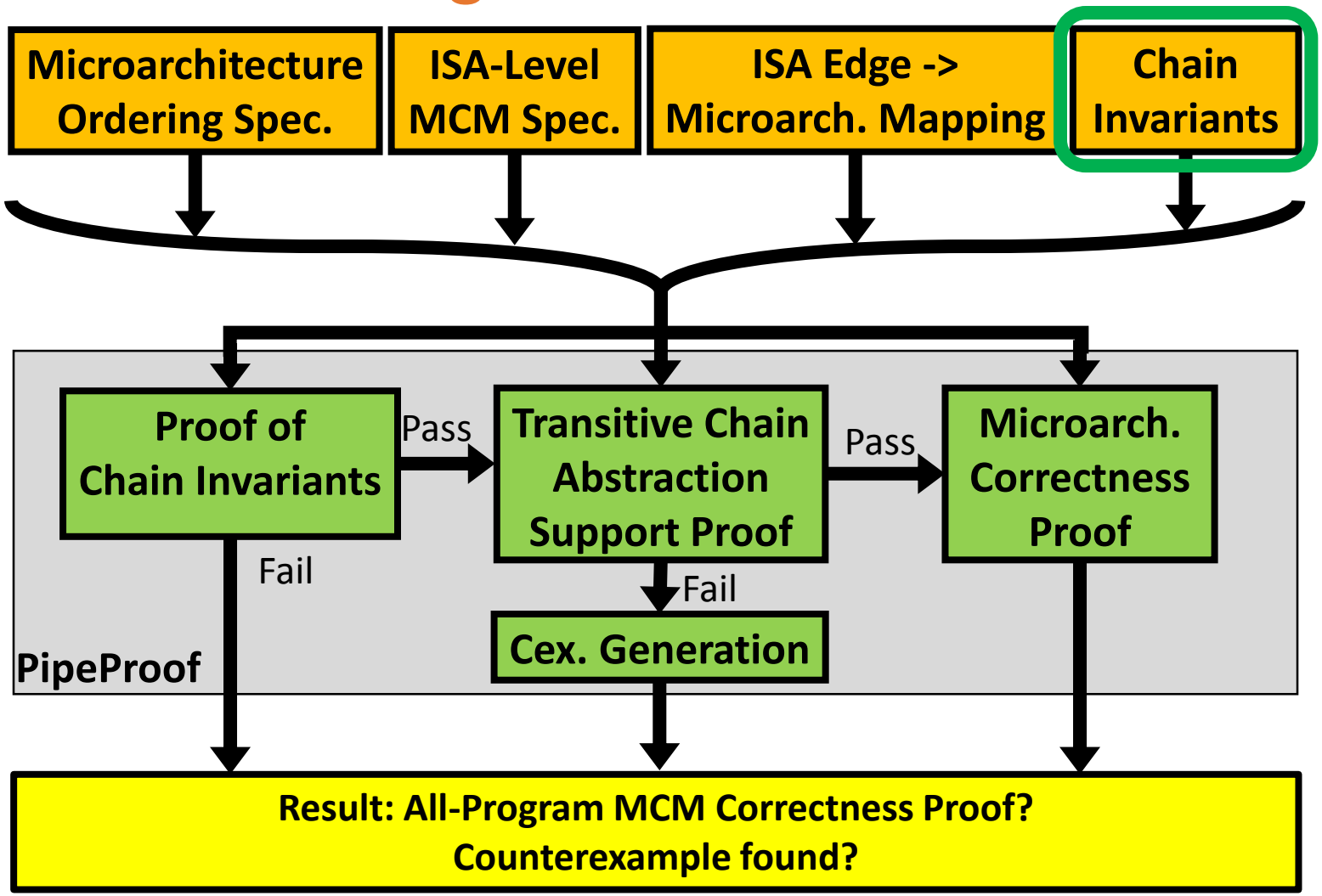


PipeProof Block Diagram

Links ISA-level and µarch executions

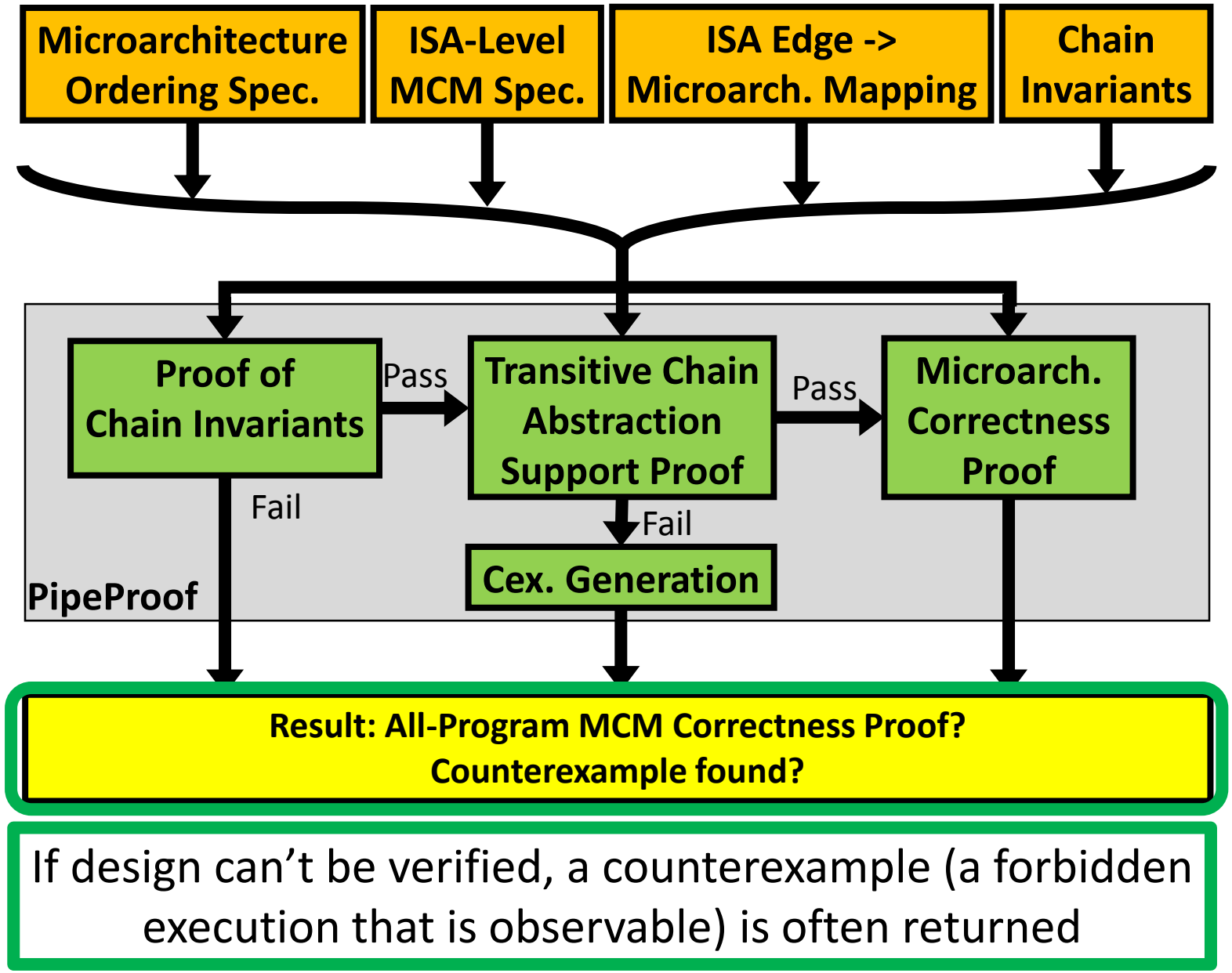


PipeProof Block Diagram



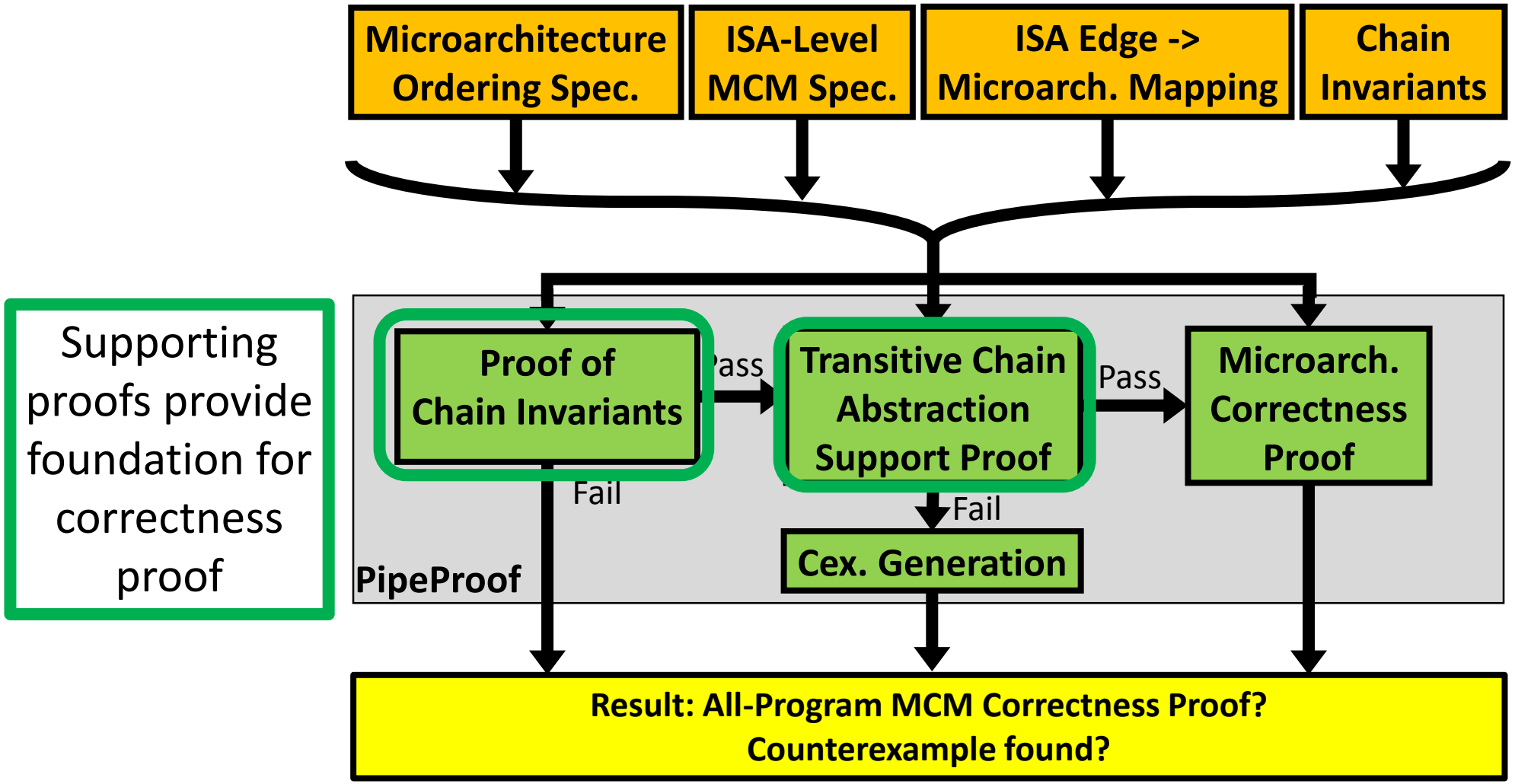
Represent repeated ISA-level patterns

PipeProof Block Diagram



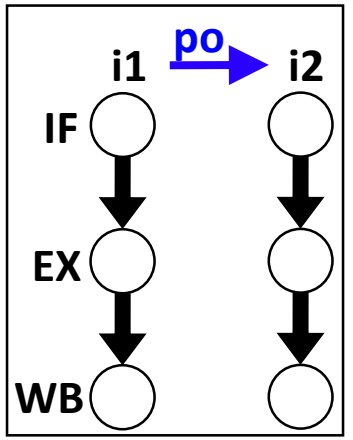
If design can't be verified, a counterexample (a forbidden execution that is observable) is often returned

PipeProof Block Diagram



Mapping ISA-Level Edges to Microarchitecture

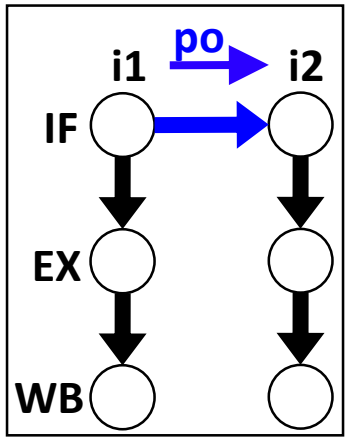
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of po edges




```
Axiom "Mapping_po":
forall microop "i",
forall microop "j",
(HasDependency po i j =>
    AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue")).
```

Mapping ISA-Level Edges to Microarchitecture

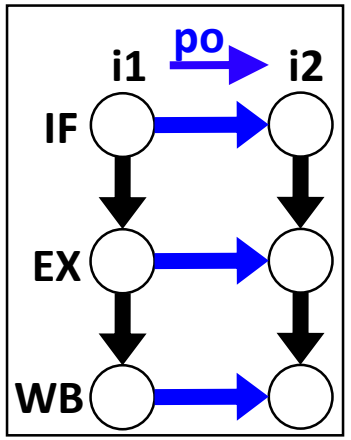
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of po edges




```
Axiom "Mapping_po":
forall microop "i",
forall microop "j",
(HasDependency po i j =>
    AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue")).
```

Mapping ISA-Level Edges to Microarchitecture

- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of po edges



Blue edges between EX and WB stages added by other FIFO axioms (refer to μspec file)

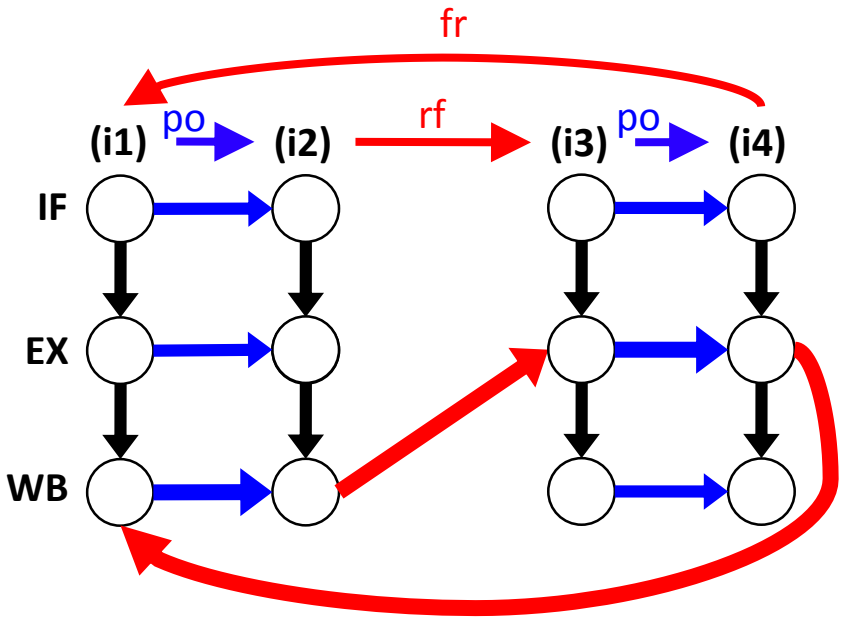
```
Axiom "Mapping_po":
forall microop "i",
forall microop "j",
(HasDependency po i j =>
    AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue")).
```

Can “litmus tests” provide complete coverage?

- Open question as to whether a set of litmus tests is **complete**

mp Litmus Test

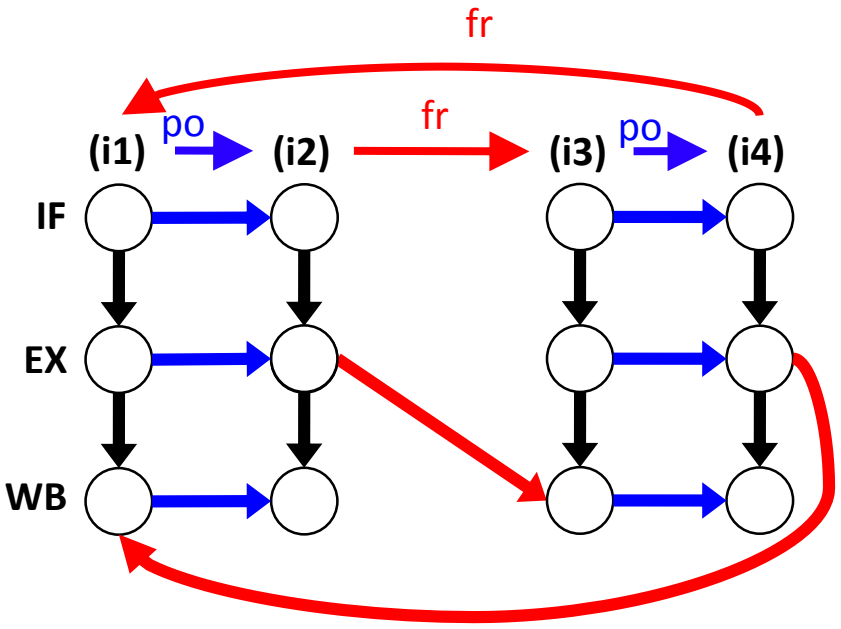
Core 0	Core 1
x = 1; y = 1;	r1 = y; r2 = x;
Forbid: r1 = 1, r2 = 0	



Cyclic => Still unobservable

sb Litmus Test

Core 0	Core 1
x = 1; r1 = y;	y = 1; r2 = x;
Forbid: r1 = 0, r2 = 0	



Acyclic => BUG!

Can “litmus tests” provide complete coverage?

- Open question as to whether a set of litmus tests is **complete**

mp Litmus Test

Core 0	Core 1
x = 1;	r1 = y;

sb Litmus Test

Core 0	Core 1
x = 1;	y = 1;

Different tests catch different bugs!

<u>To catch all bugs, must verify across all programs!</u>

Cyclic => Still unobservable

Acyclic => BUG!

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	

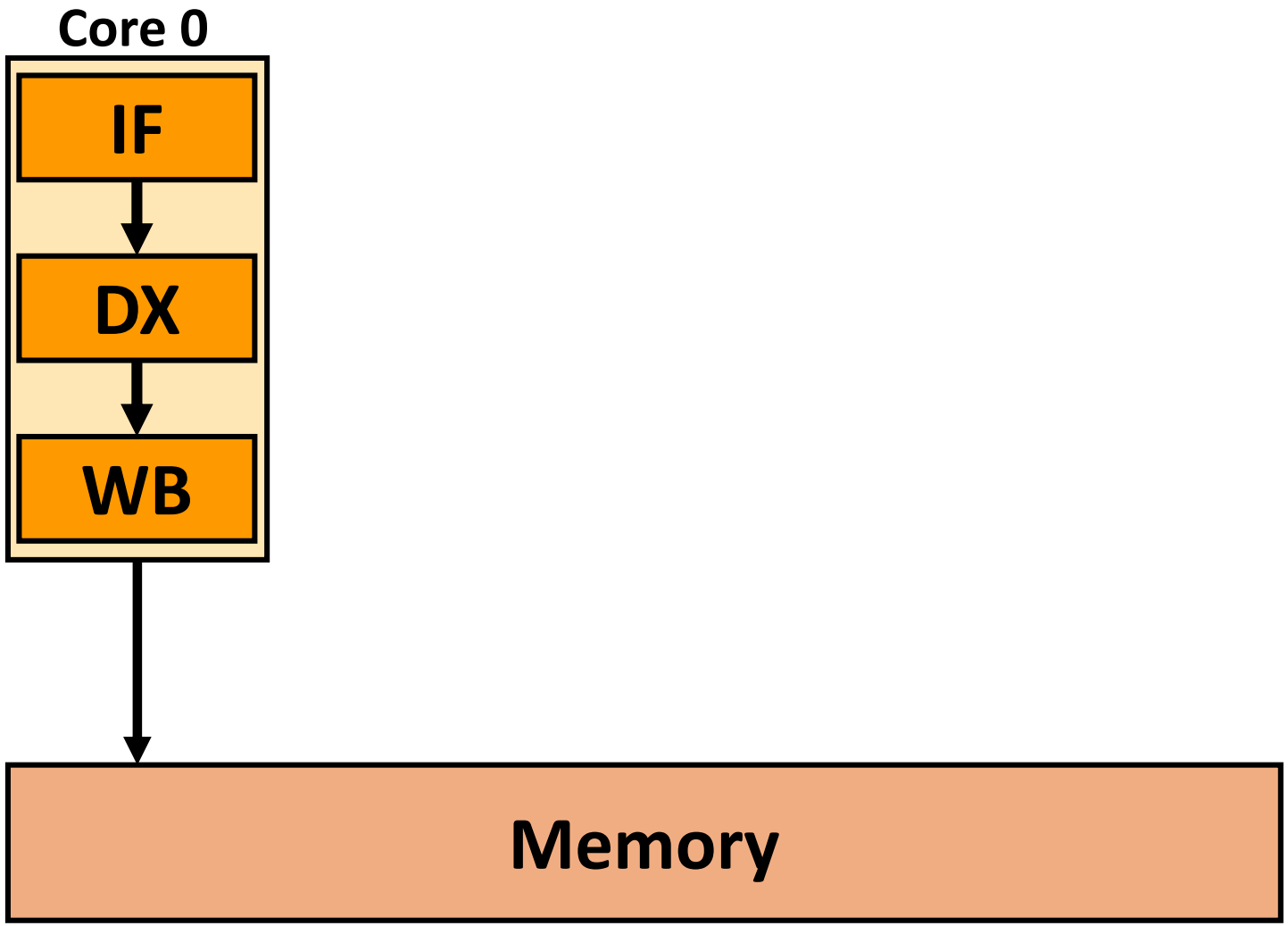
- Don't filter based on outcome
  - Translate **all** possible outcomes
- Tag each case with appropriate **load value constraints**
  - reflect the data constraints required for edge(s)
- Ongoing work: Precisely formalise the μspec/SVA mismatch
  - How much is fundamental? How much is due to SVA verifier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

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

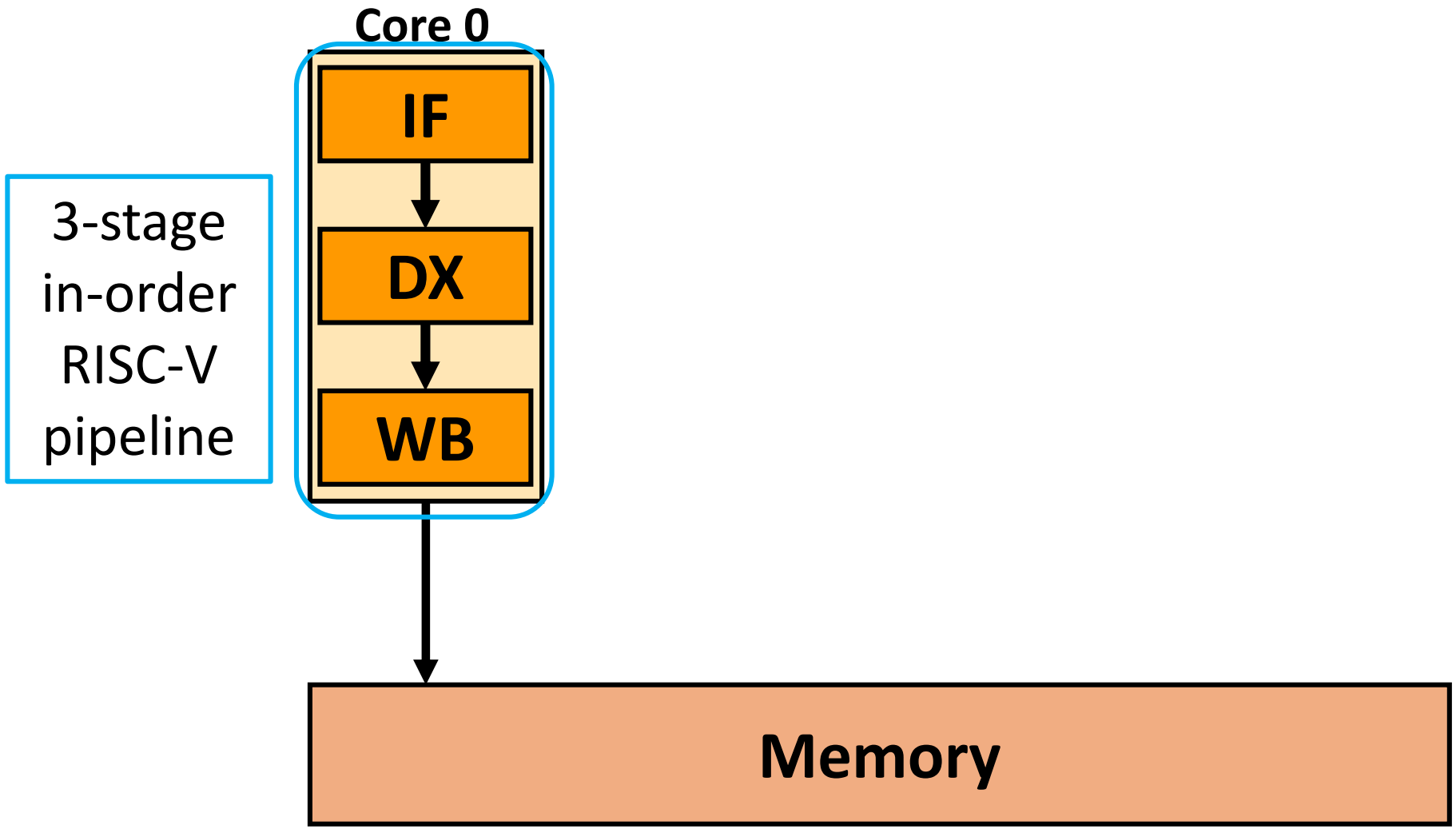
```
Property to check:
mapNode(Ld x → St x, Ld x == 0) or mapNode(St x → Ld x, Ld x == 1);
```

Note: Axioms and properties abstracted for brevity

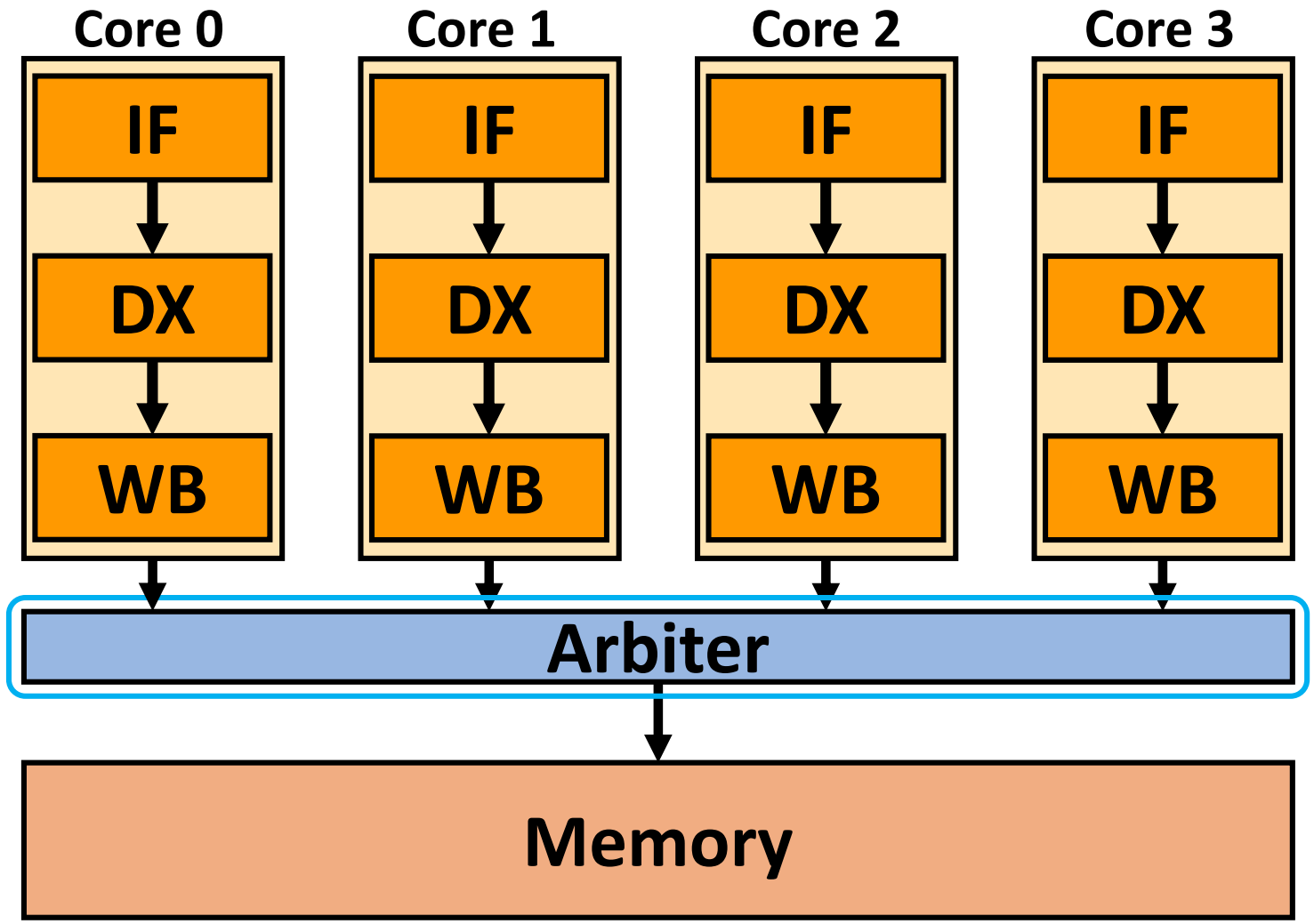
Multi-V-scale: a Multicore Case Study



Multi-V-scale: a Multicore Case Study



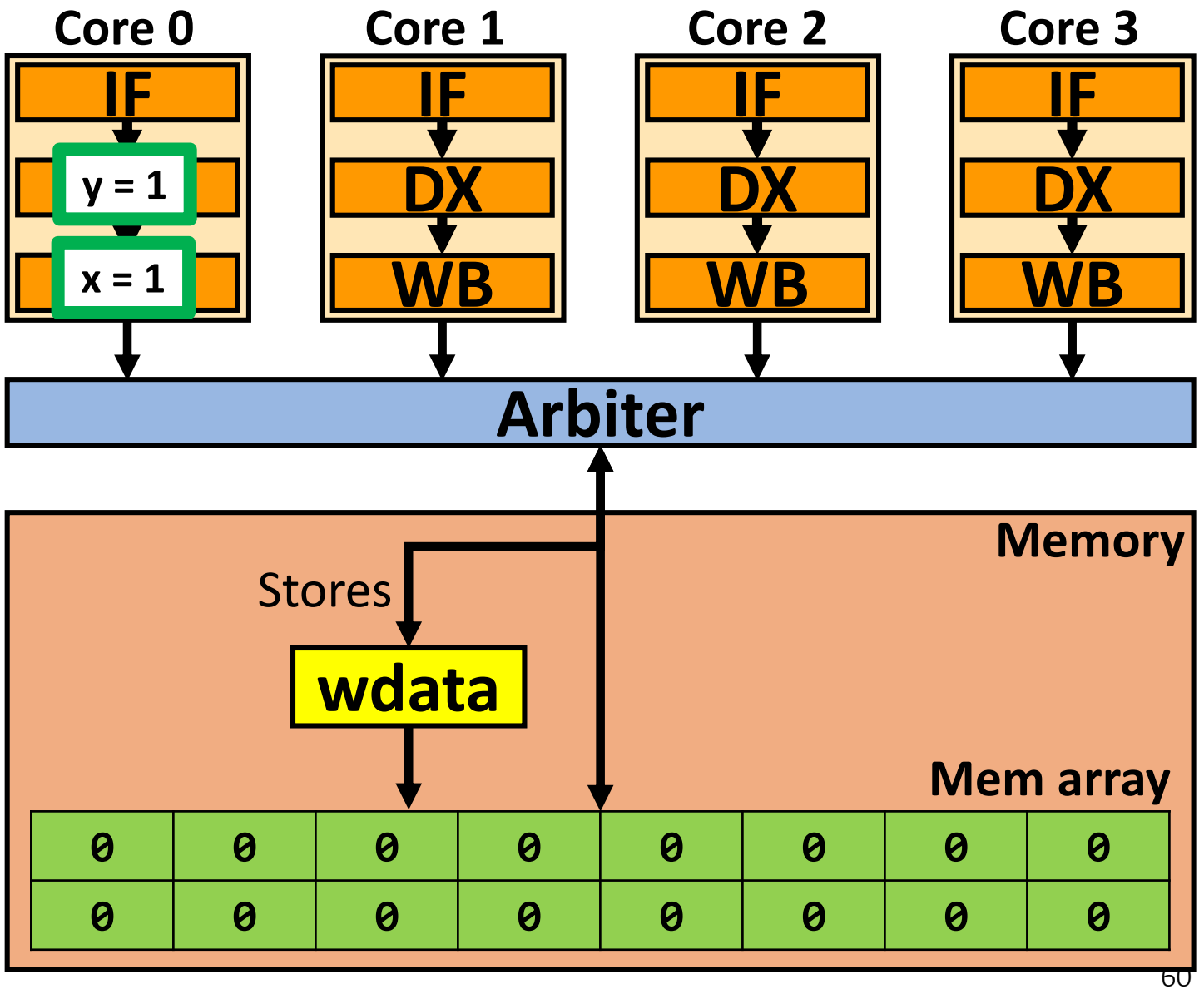
Multi-V-scale: a Multicore Case Study



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

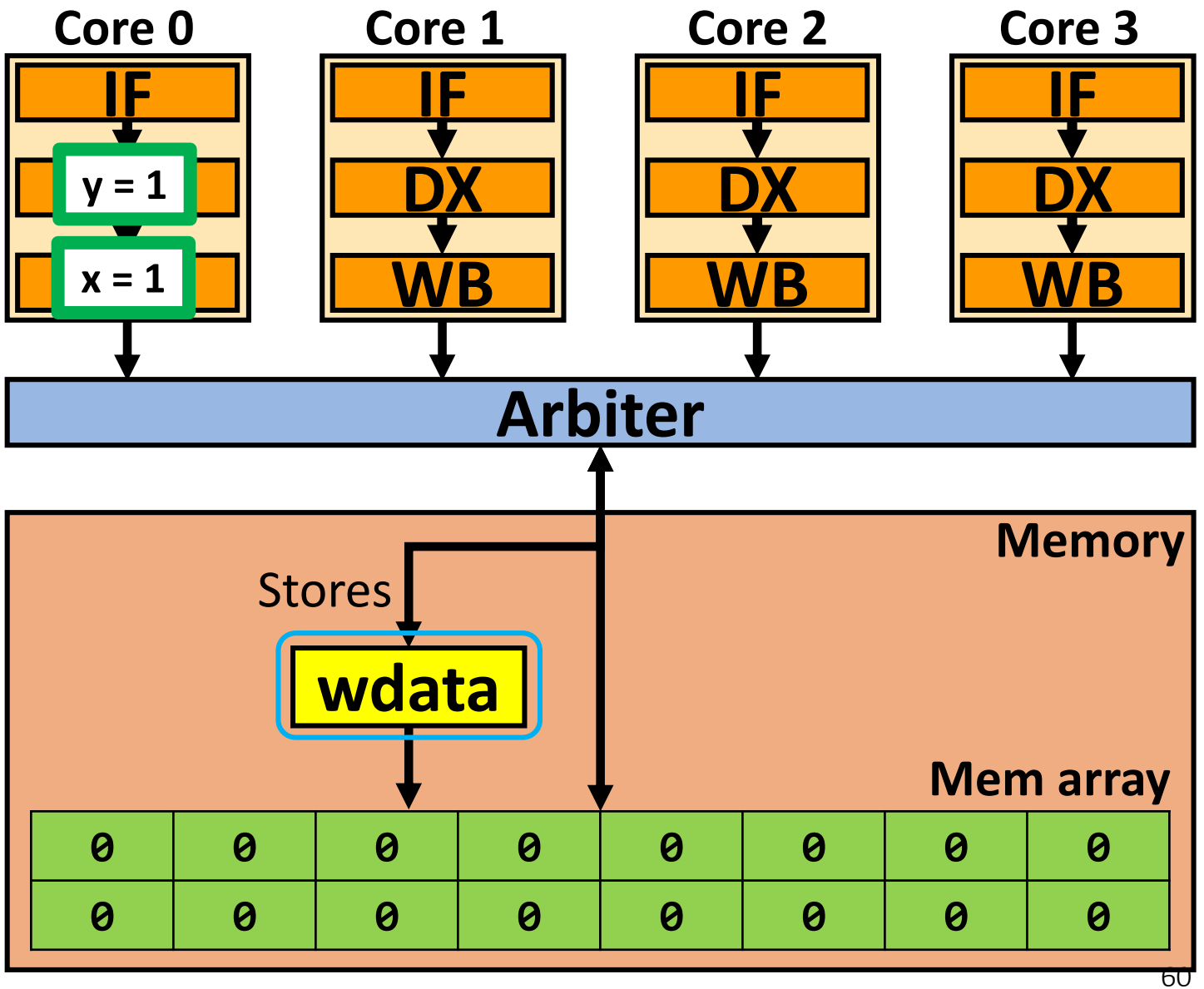
Bug Discovered in V-scale Mem. Implementation

- **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