Synthesizing High Level Models from RTL for Efficient Verification of Memory Model Implementations

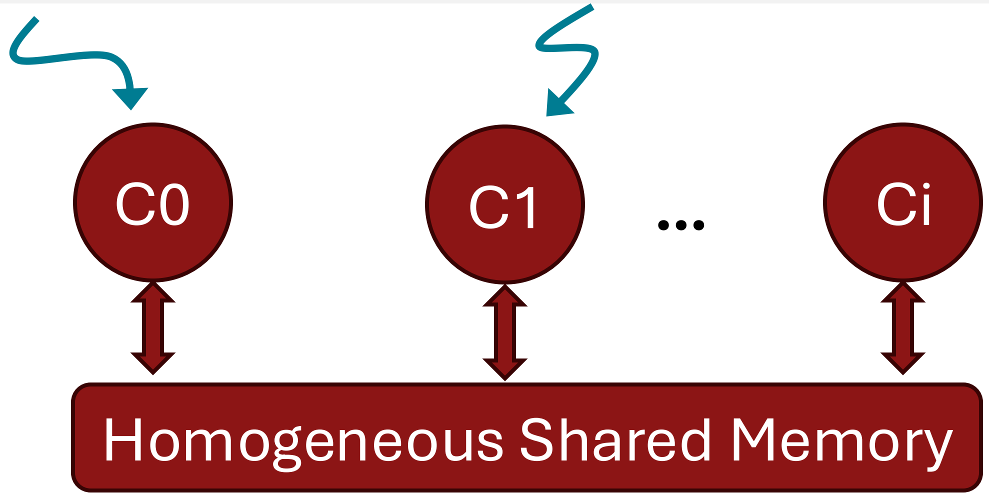
Yao Hsiao, Caroline Trippel

Jan 10, 2025

Stanford University

Memory Consistency Model (MCM) defines the ordering and visibility of shared memory accesses on a multiprocessor

```
      data = 0, flag = 0
       C0                 C1
⓪ ST [data] 1   ② LD [flag] 1
① ST [flag] 1   ③ LD [data] 0
```



Can this execution order of ①②③⓪ happen ?

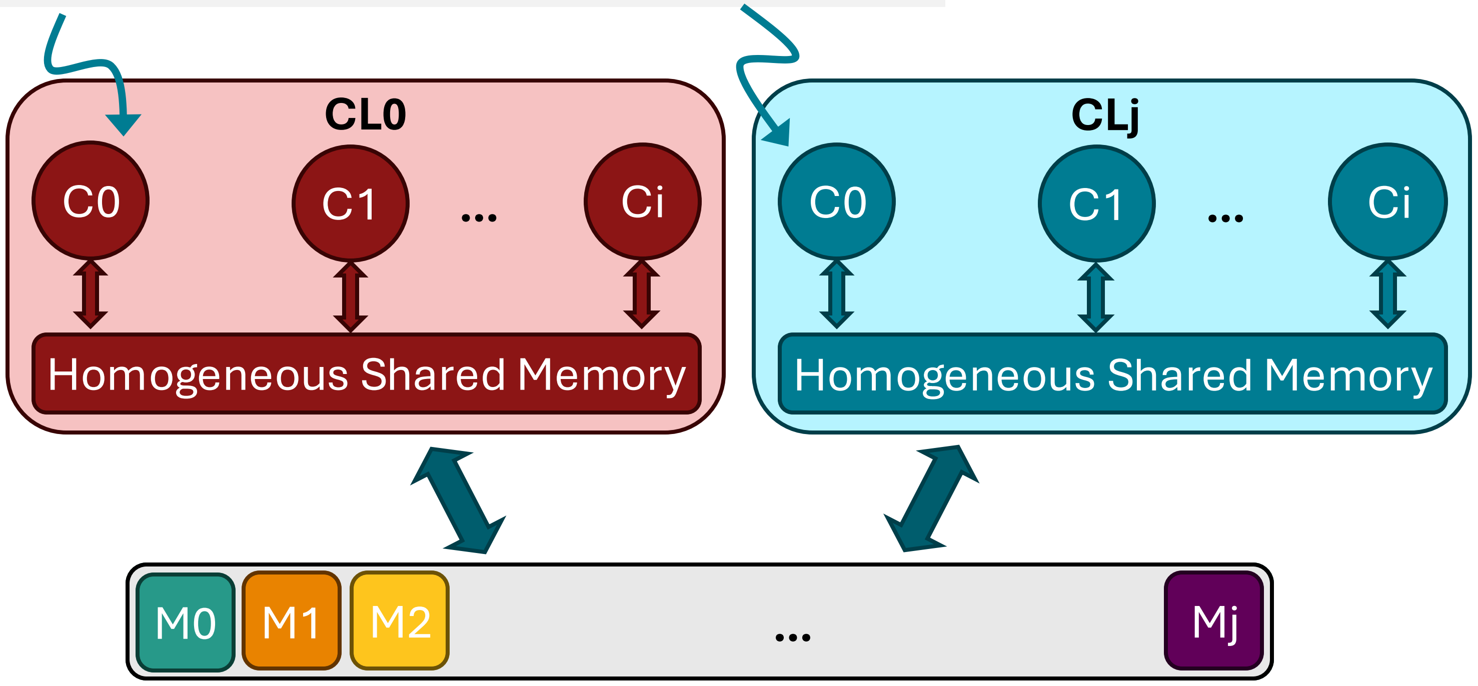
✗ Intel x86 Sequential Consistency (SC)

✓ arm NVIDIA RISC-V IBM

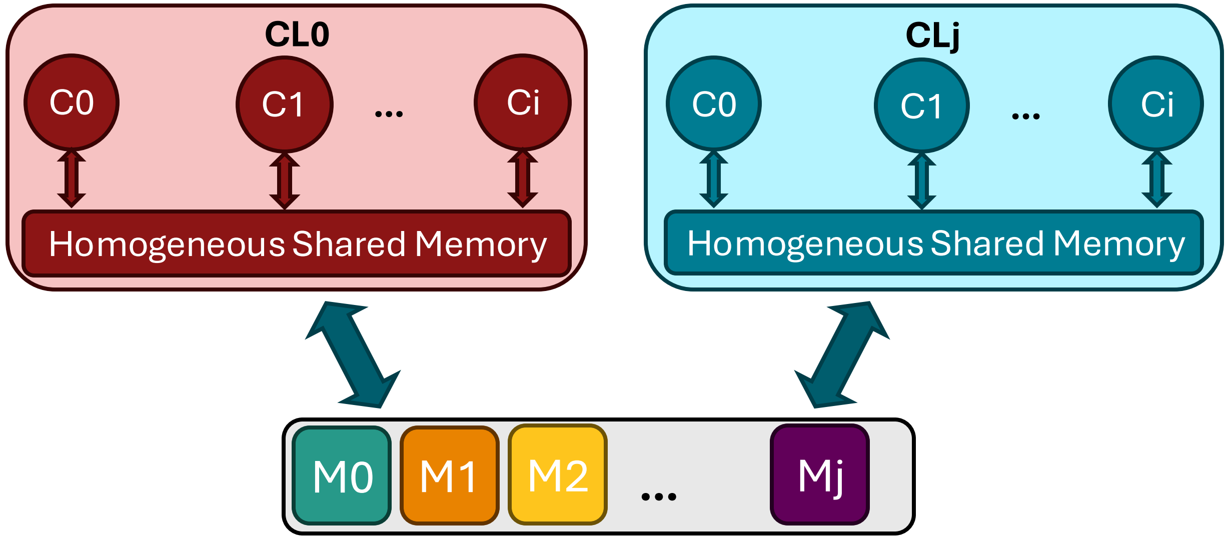
MCMs determine **legal outcomes** of a parallel program on a machine

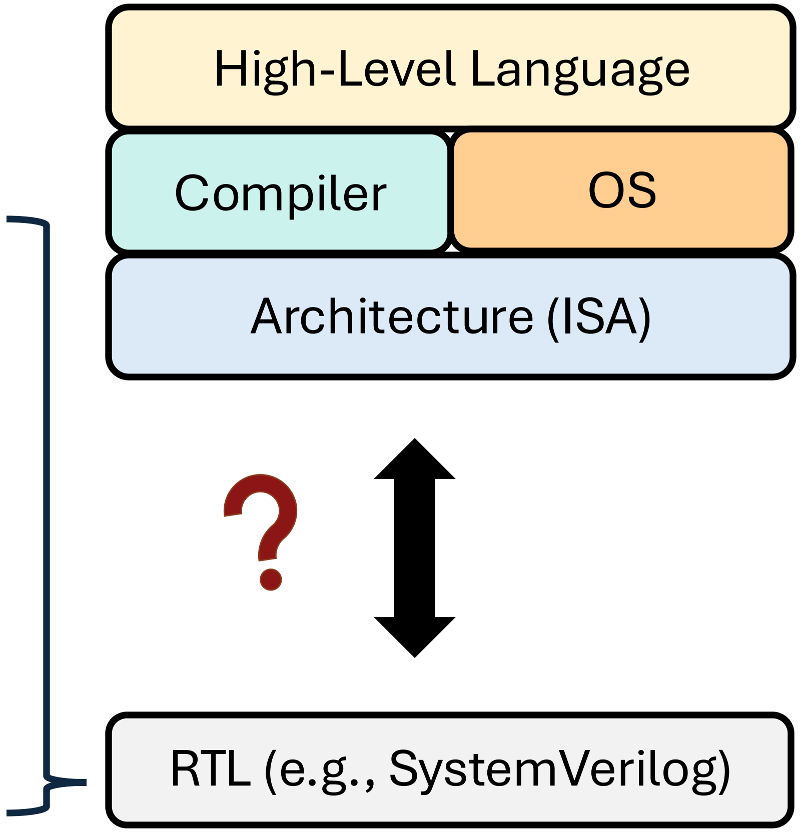
Memory Consistency Model (MCM) defines the ordering and visibility of shared memory accesses on a multiprocessor

```
        data = 0, flag = 0
        C0                 C1
 ⓪ ST [data] 1    ② LD [flag] 1
 ① ST [flag] 1    ③ LD [data] 0
```

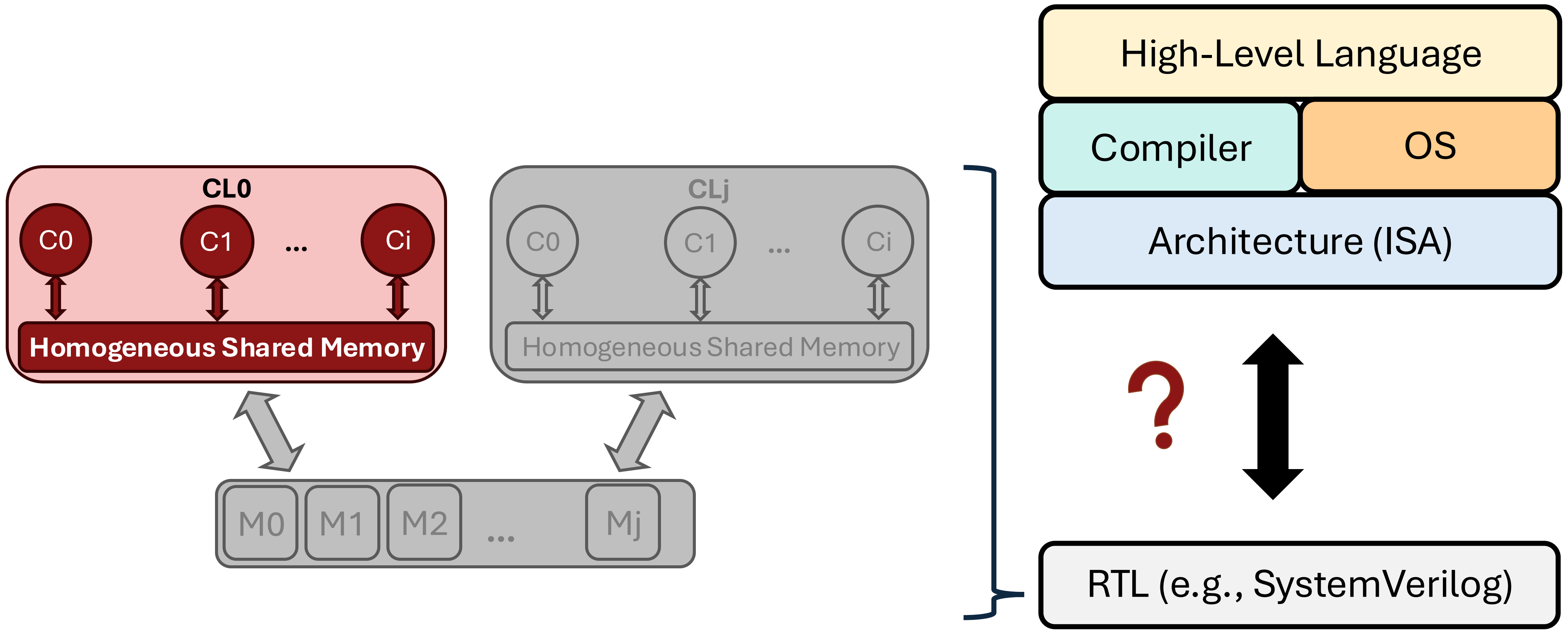


Challenge: How do we ensure that microarchitecture correctly implements its ISA MCM?

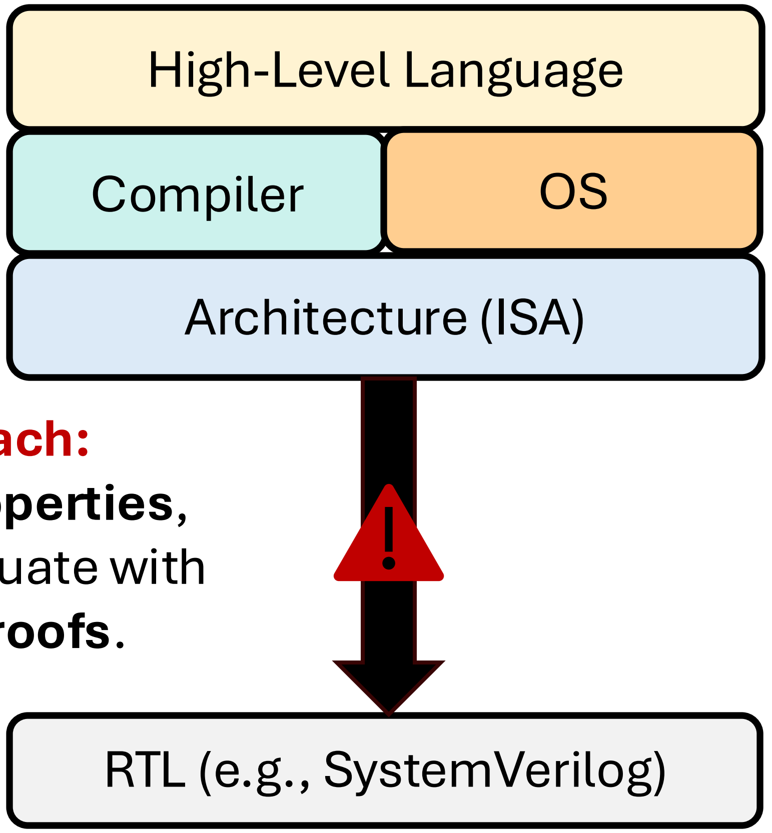




Challenge: How do we ensure that microarchitecture correctly implements its ISA MCM?

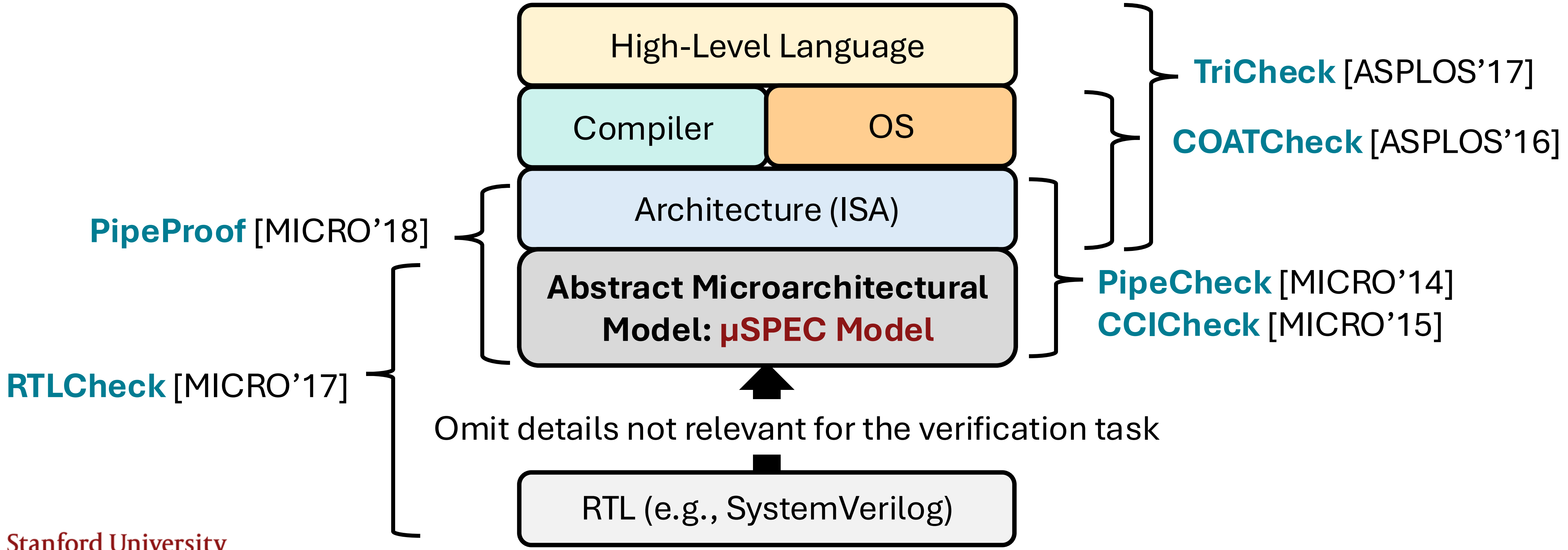


Challenge: How do we ensure that microarchitecture correctly implements its ISA MCM?



State-of-the-art top-down approach:
Manually encode **formal MCM properties**, map down to RTL signals, and evaluate with model checkers to get **bounded proofs**.

The *Check Tools* Automate Formal Verification of Hardware Memory Model Implementations By Analyzing Abstract Model of Hardware



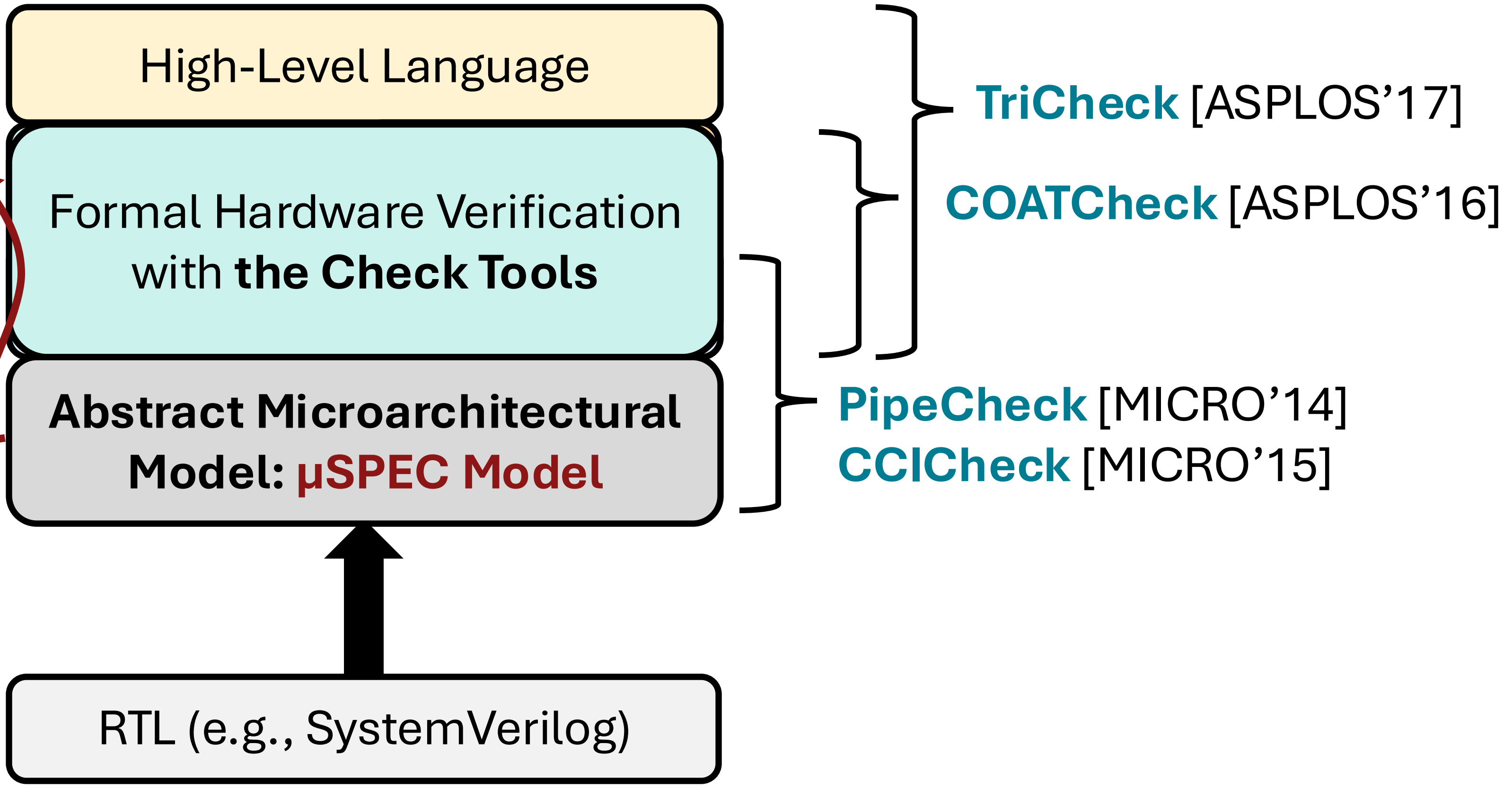
Axiomatic Microarchitectural Models Enable Formal Analysis

[Lustig+, ASPLOS'16]

```
Axiom Ld_exe_path:
forall microops i,
IsAnyRead i ⇒
AddEdges [((i, IF), (i, DX));
((i, DX), (i, RdMm));
((i, RdMm), (i, L1$));
((i, L1$), (i, WB));] \/
AddEdges [((i, IF), (i, DX));
((i, DX), (i, WB));]
// Other axioms...
```

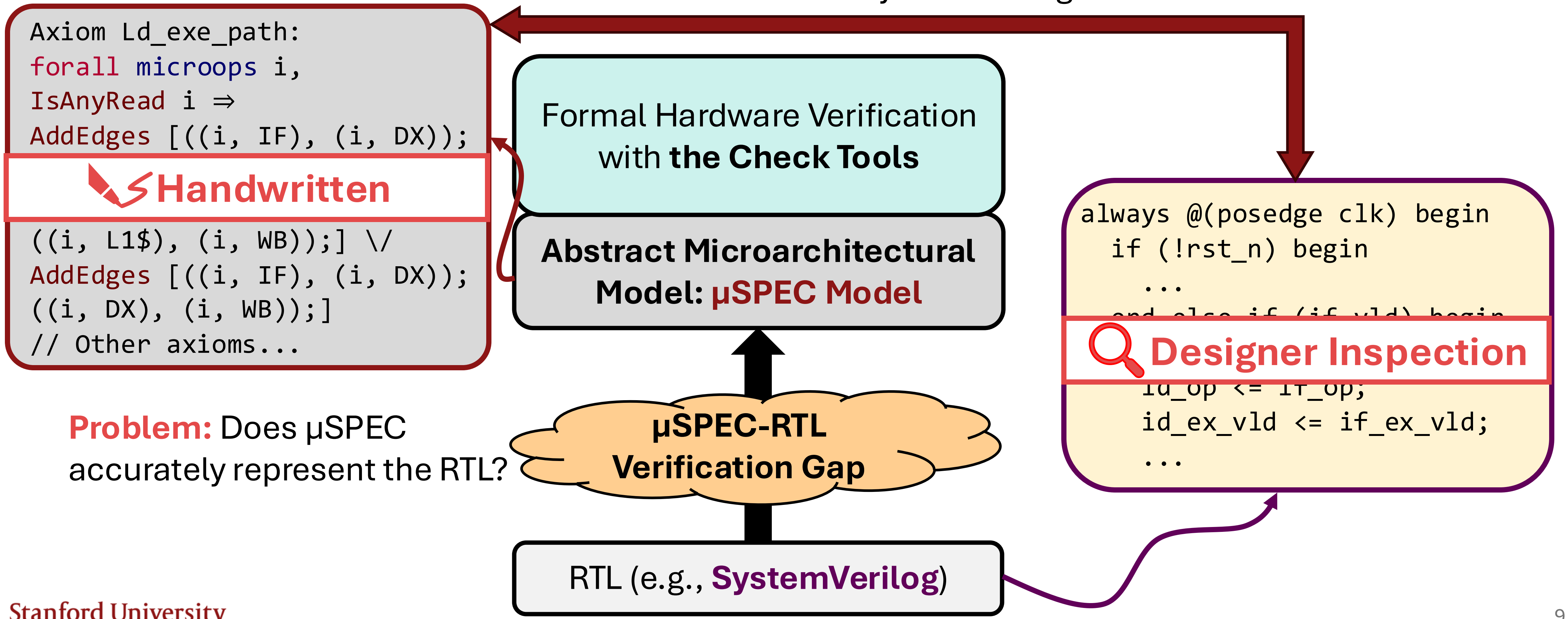
First order logic **axiomatic model** of a microarchitecture = a set of invariants upheld by the microarchitecture:

- Omits **combinational** logic details
- Retains **state updates** and **ordering** details


High-Level Language
Formal Hardware Verification with **the Check Tools**
Abstract Microarchitectural Model: µSPEC Model
RTL (e.g., SystemVerilog)
TriCheck [ASPLOS'17]
COATCheck [ASPLOS'16]
PipeCheck [MICRO'14]
CCICheck [MICRO'15]

Verification Challenge: How to Verify that a µSPEC Accurately Represents a SystemVerilog Microarchitecture

µSPEC looks **quite different** from SystemVerilog!



Handwritten
Formal Hardware Verification with **the Check Tools**
Abstract Microarchitectural Model: µSPEC Model
µSPEC-RTL Verification Gap
RTL (e.g., **SystemVerilog**)
Designer Inspection

```
Axiom Ld_exe_path:
forall microops i,
IsAnyRead i ⇒
AddEdges [((i, IF), (i, DX));
((i, L1$), (i, WB));] \/
AddEdges [((i, IF), (i, DX));
((i, DX), (i, WB));]
// Other axioms...
```

```
always @(posedge clk) begin
  if (!rst_n) begin
    ...
  end else if (if_vld) begin
    id_op <= if_op;
    id_ex_vld <= if_ex_vld;
    ...
```

Problem: Does µSPEC accurately represent the RTL?

Roadmap Toward Automatic Synthesis of Verified µSPEC

- **Background:** The Microarchitecture-µSPEC Model Verification Challenge
- **RTL2µSPEC:** Synthesizing µSPEC model from Simple Processor RTL Designs
- **RTL2MµPATH:** Synthesizing ("Uncovering") All µPATHs per Instruction from Advanced SystemVerilog Processors
- **Next Steps:** Support synthesis of µSPEC axioms for coherence protocol and complex data dependencies in complex processors

Toward automatic synthesis of µSPEC model for complex multiprocessors

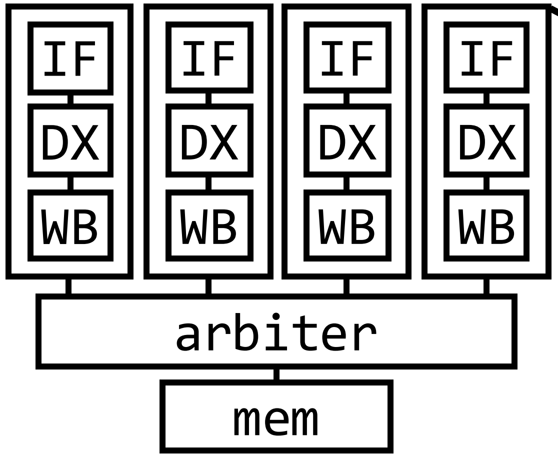
Roadmap Toward Automatic Synthesis of Verified µSPEC

- **Background**: The Microarchitecture-µSPEC Model Verification Challenge
- **RTL2µSPEC**: Synthesizing µSPEC model from Simple Processor RTL Designs
- **RTL2MµPATH**: Synthesizing ("Uncovering") All µPATHs per Instruction from Advanced SystemVerilog Processors
- Next Steps: Support synthesis of µSPEC axioms for coherence protocol and complex data dependencies in complex processors

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

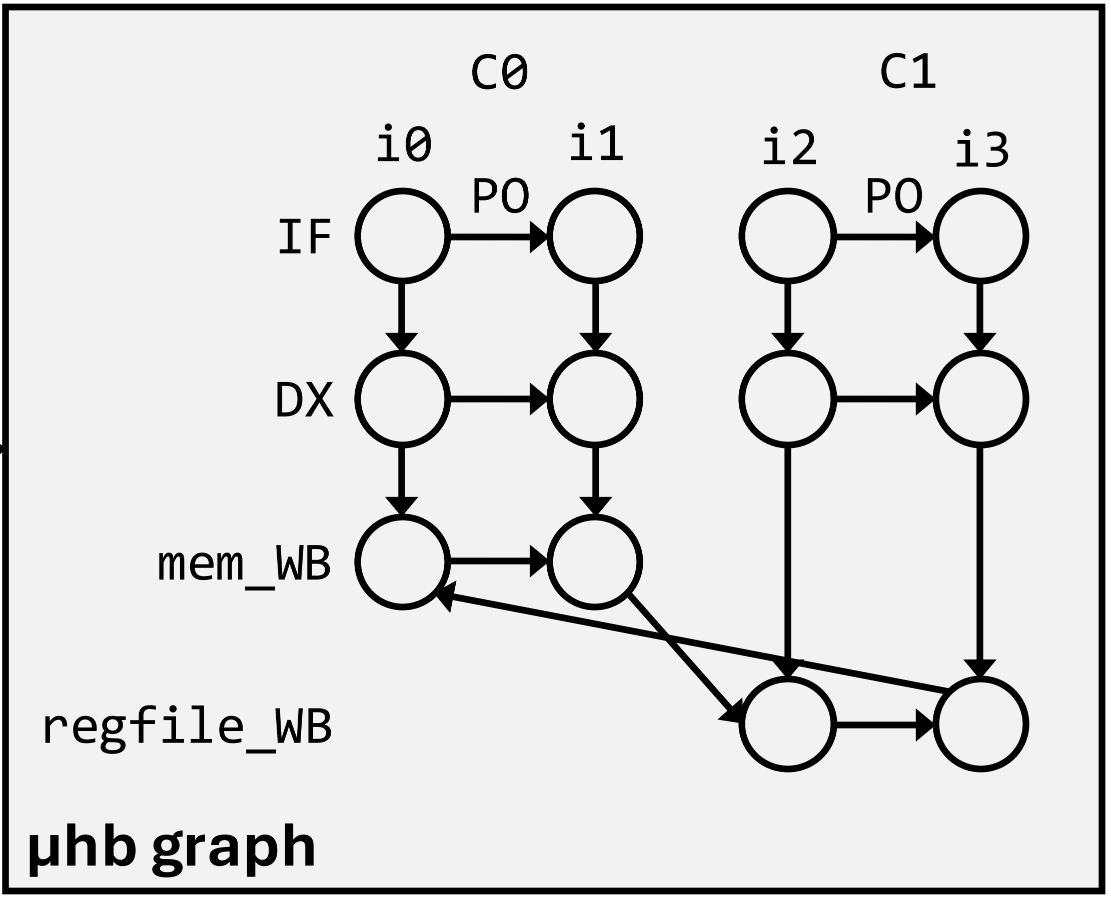
Forbidden on SC

```
          C0                    C1
(i0) ST [data] 1   (i2) LD [flag] 1
(i1) ST [flag] 1   (i3) LD [data] 0
```



RISC-V multi-V-scale

μSPEC Model



Implement Sequential Consistency (SC)?

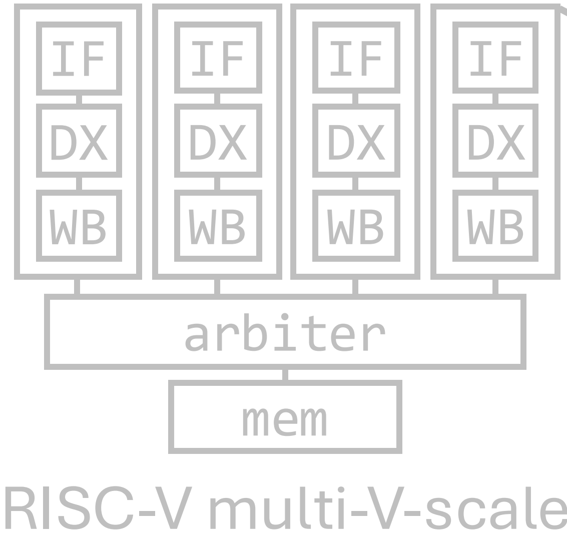
Check Tools: searches all ways program executes on hardware

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

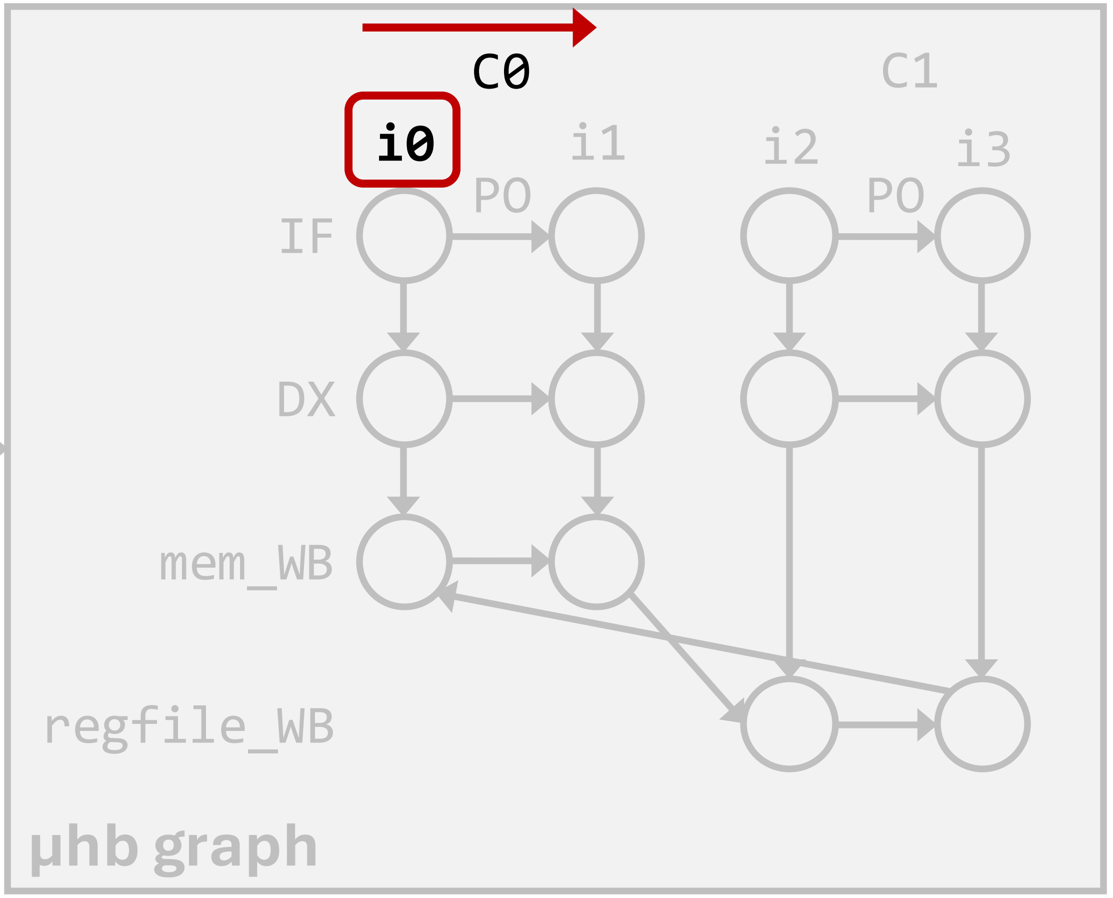
Forbidden on SC

Program Order (PO)

C0	C1
(i0) ST [data] 1	(i2) LD [flag] 1
(i1) ST [flag] 1	(i3) LD [data] 0



μSPEC Model

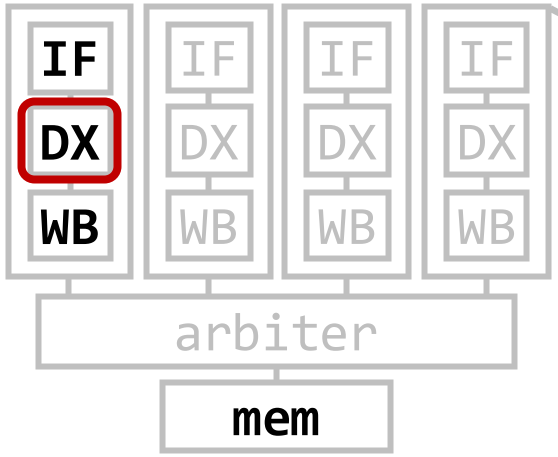


Implement Sequential Consistency (SC)?

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

Forbidden on SC

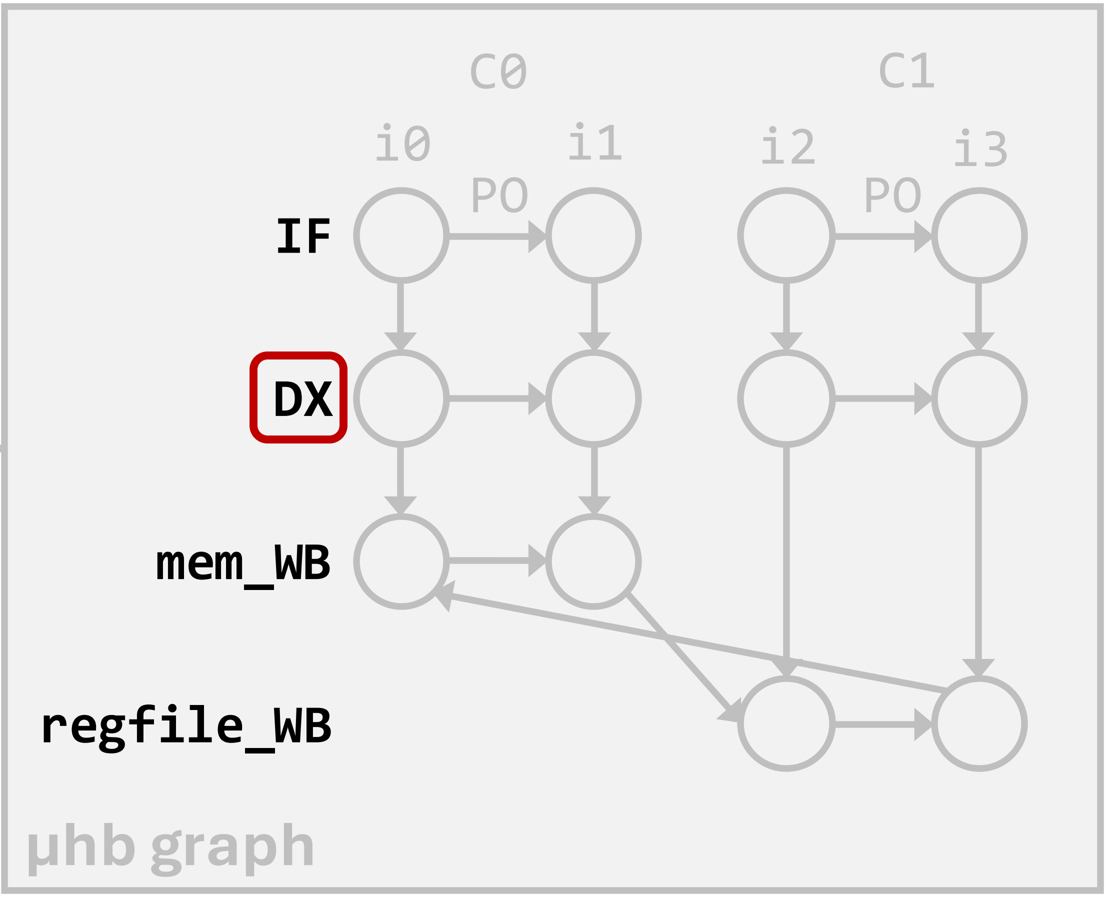
C0	C1
(i0) ST [data] 1	(i2) LD [flag] 1
(i1) ST [flag] 1	(i3) LD [data] 0



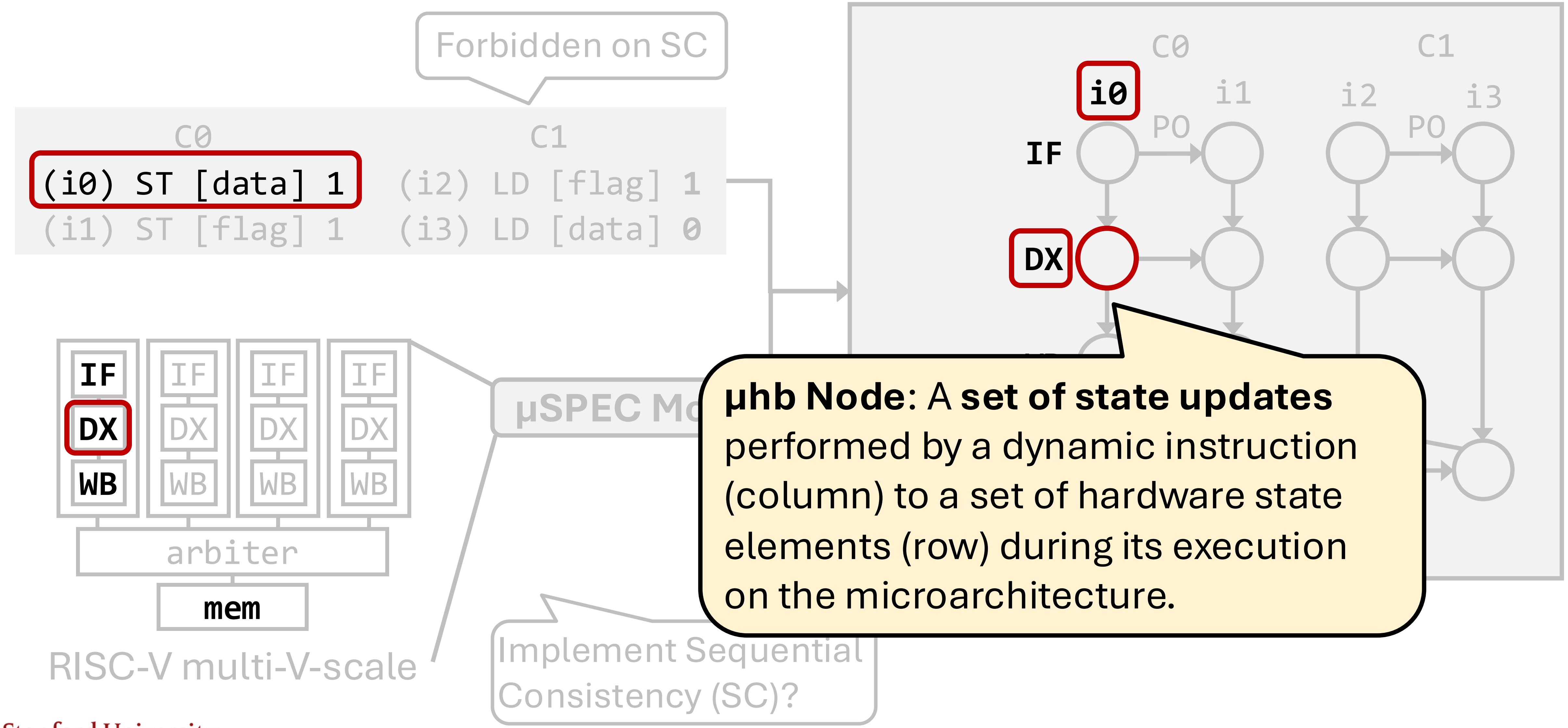
RISC-V multi-V-scale

μSPEC Model

Implement Sequential Consistency (SC)?

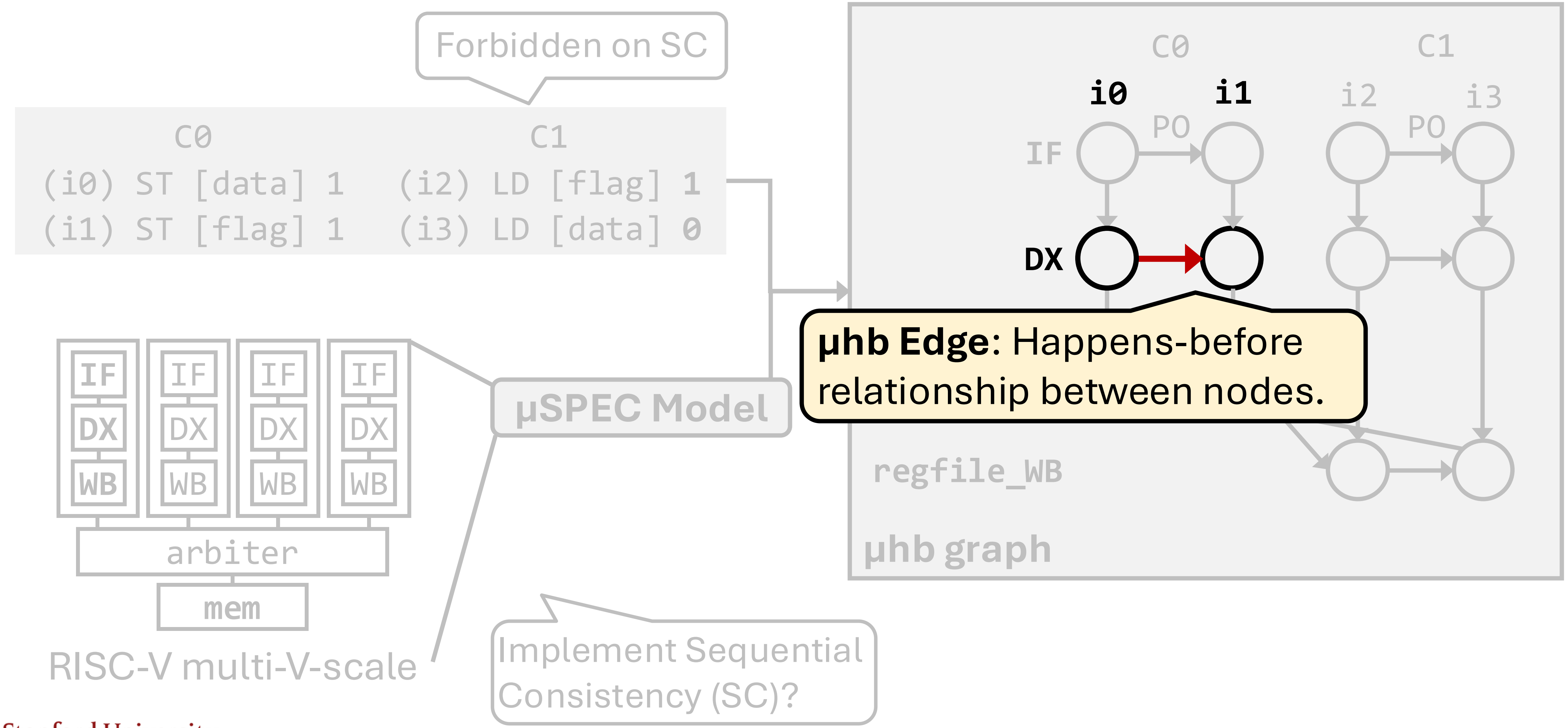


Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]



μhb Node: A **set of state updates** performed by a dynamic instruction (column) to a set of hardware state elements (row) during its execution on the microarchitecture.

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO’14]



```
C0                    C1
(i0) ST [data] 1   (i2) LD [flag] 1
(i1) ST [flag] 1   (i3) LD [data] 0
```

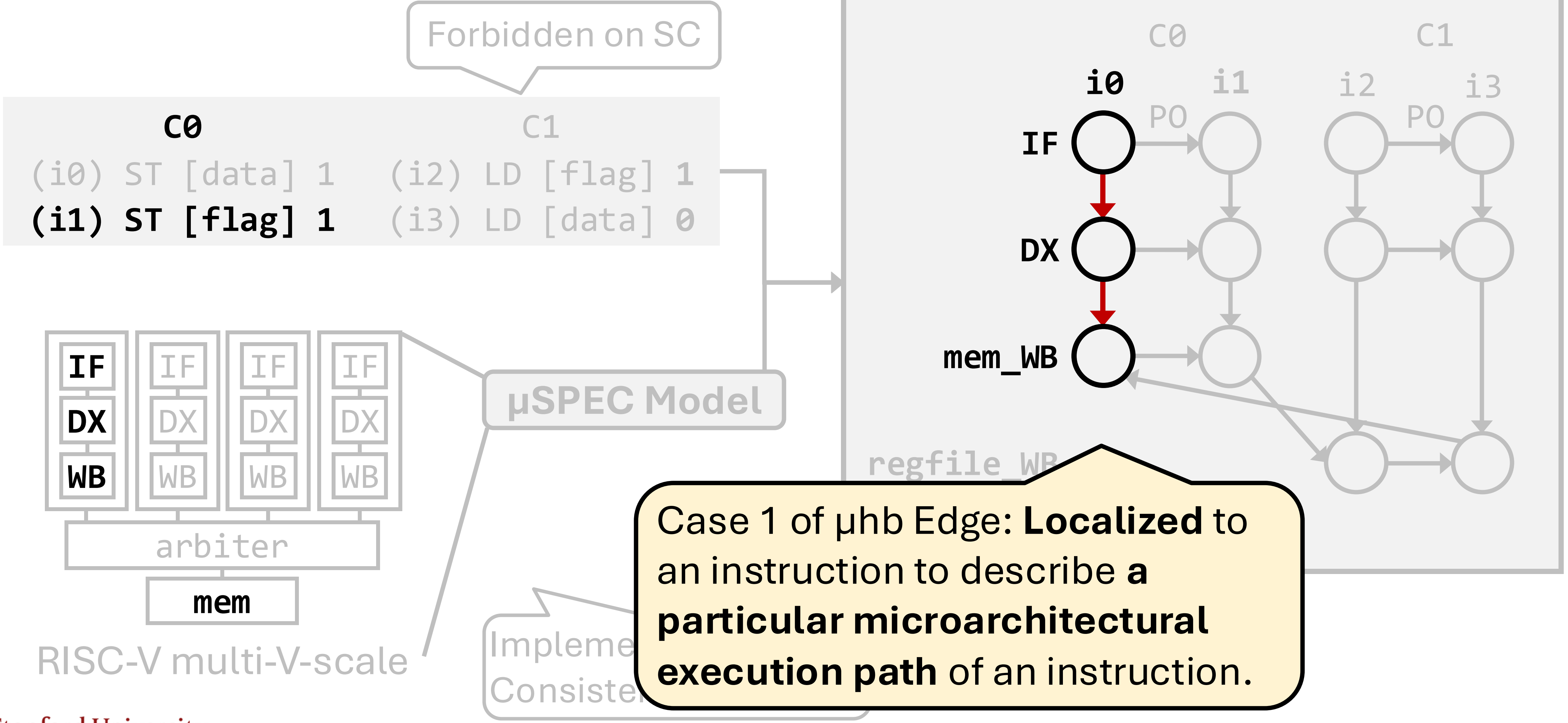
μhb Edge: Happens-before relationship between nodes.

Stanford University

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

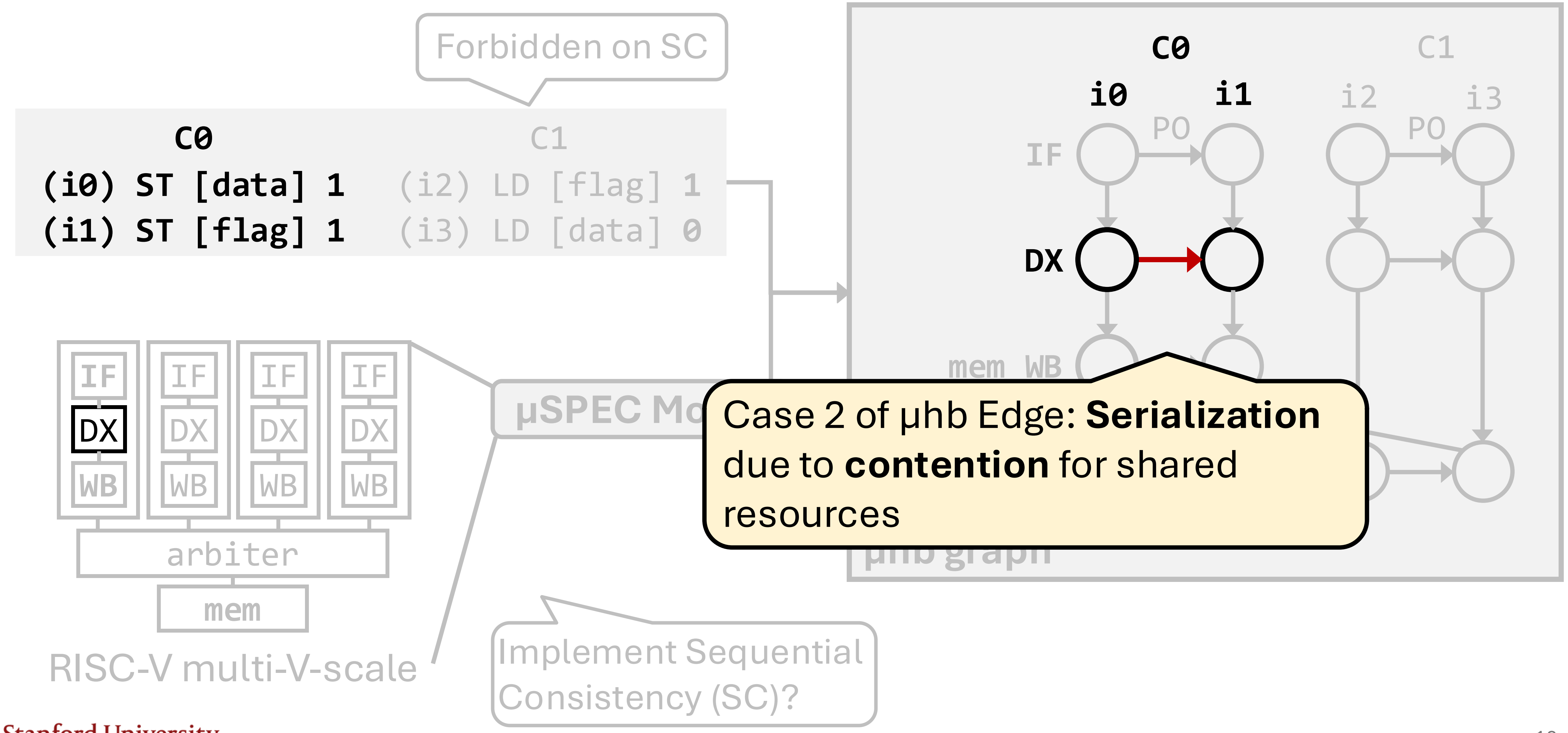
Forbidden on SC

C0	C1
(i0) ST [data] 1	(i2) LD [flag] 1
(i1) ST [flag] 1	(i3) LD [data] 0



Case 1 of μhb Edge: **Localized** to an instruction to describe **a particular microarchitectural execution path** of an instruction.

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]



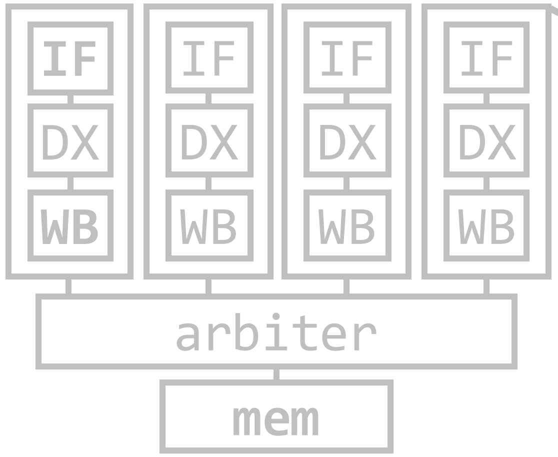
Case 2 of μhb Edge: **Serialization** due to **contention** for shared resources

Stanford University

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

Forbidden on SC

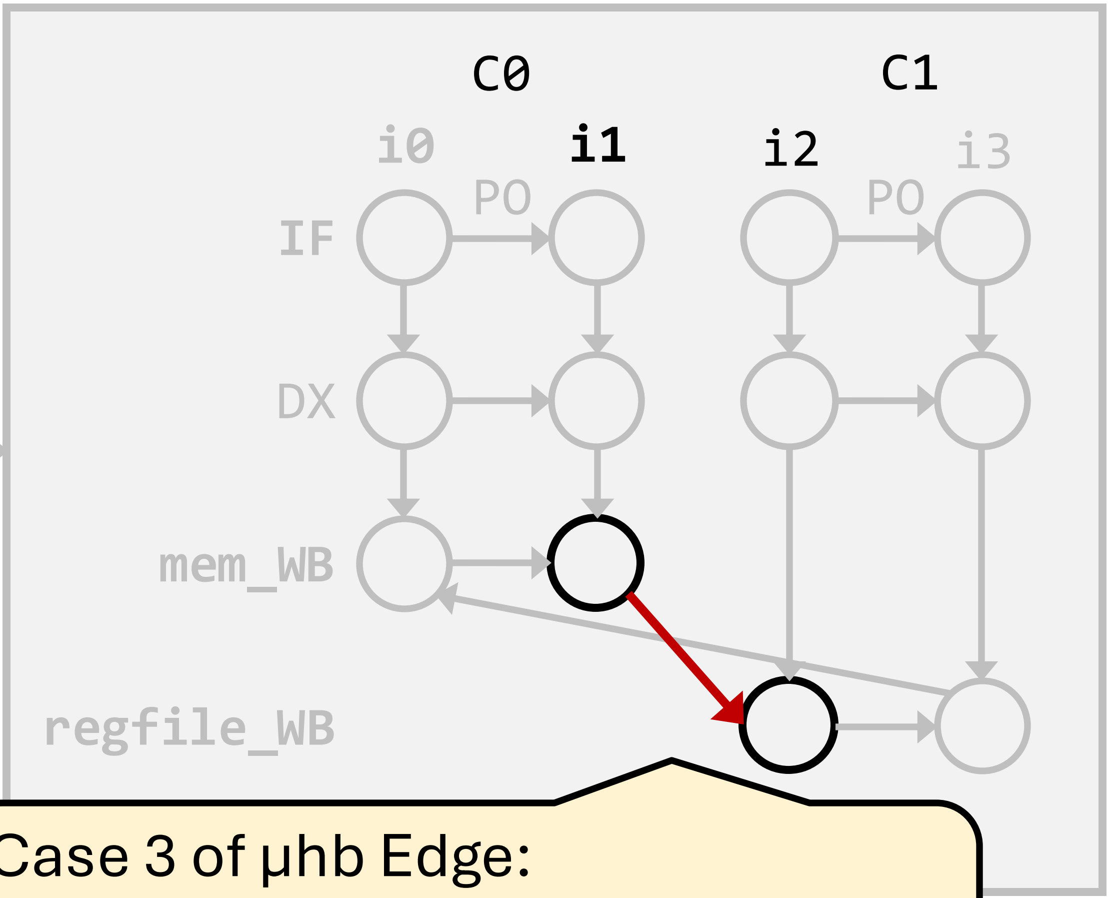
```
C0                        C1
(i0) ST [data] 1    (i2) LD [flag] 1
(i1) ST [flag] 1    (i3) LD [data] 0
```



RISC-V multi-V-scale

μSPEC Model

Implement Sequent
Consistency (SC)?

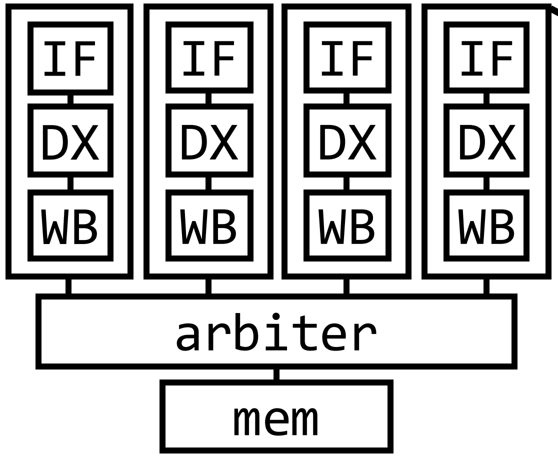


Case 3 of μhb Edge:
Communication via **shared resources**

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

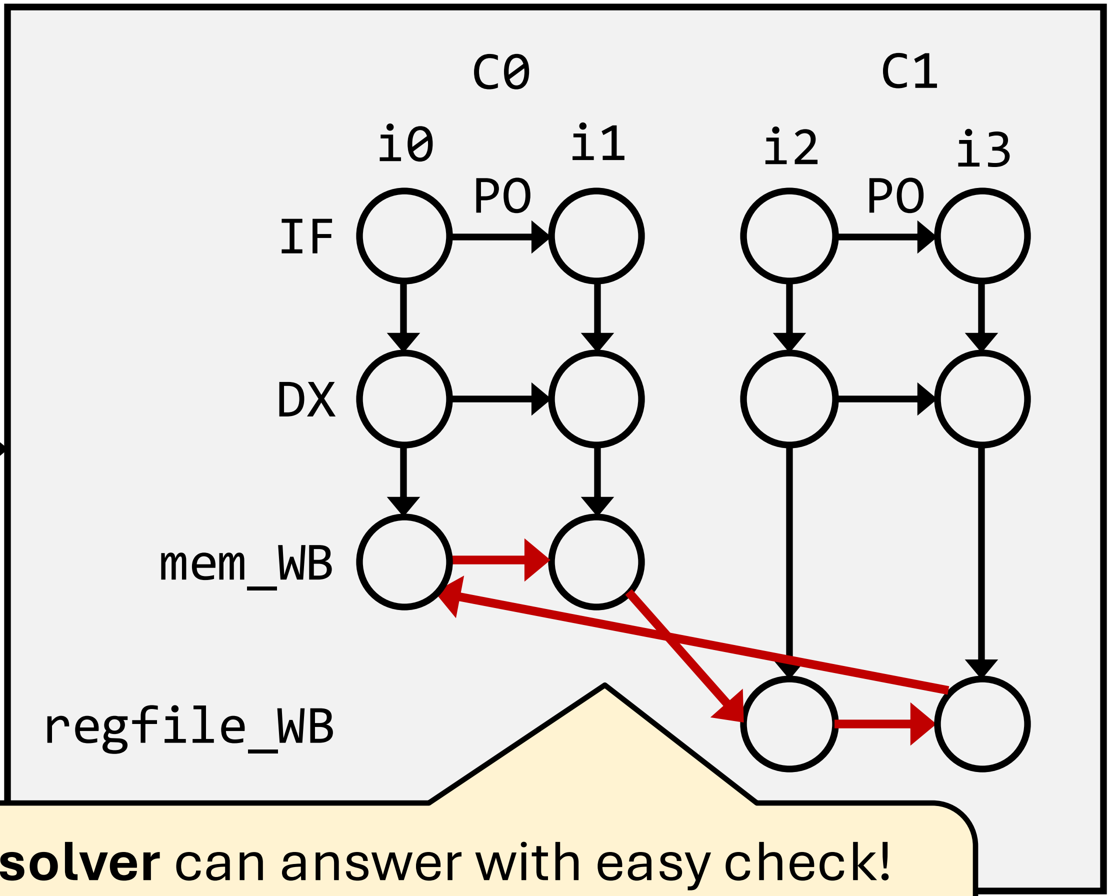
Is illegal outcome possible?

```
        C0                    C1
(i0) ST [data] 1   (i2) LD [flag] 1
(i1) ST [flag] 1   (i3) LD [data] 0
```



RISC-V multi-V-scale

μSPEC Model

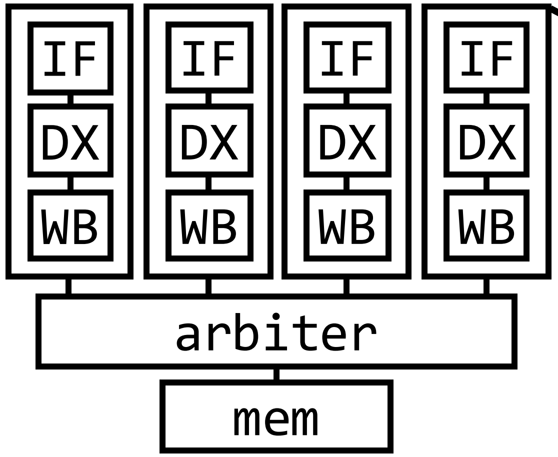


SAT or SMT solver can answer with easy check!

Microarchitectural Happens-Before (μHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO’14]

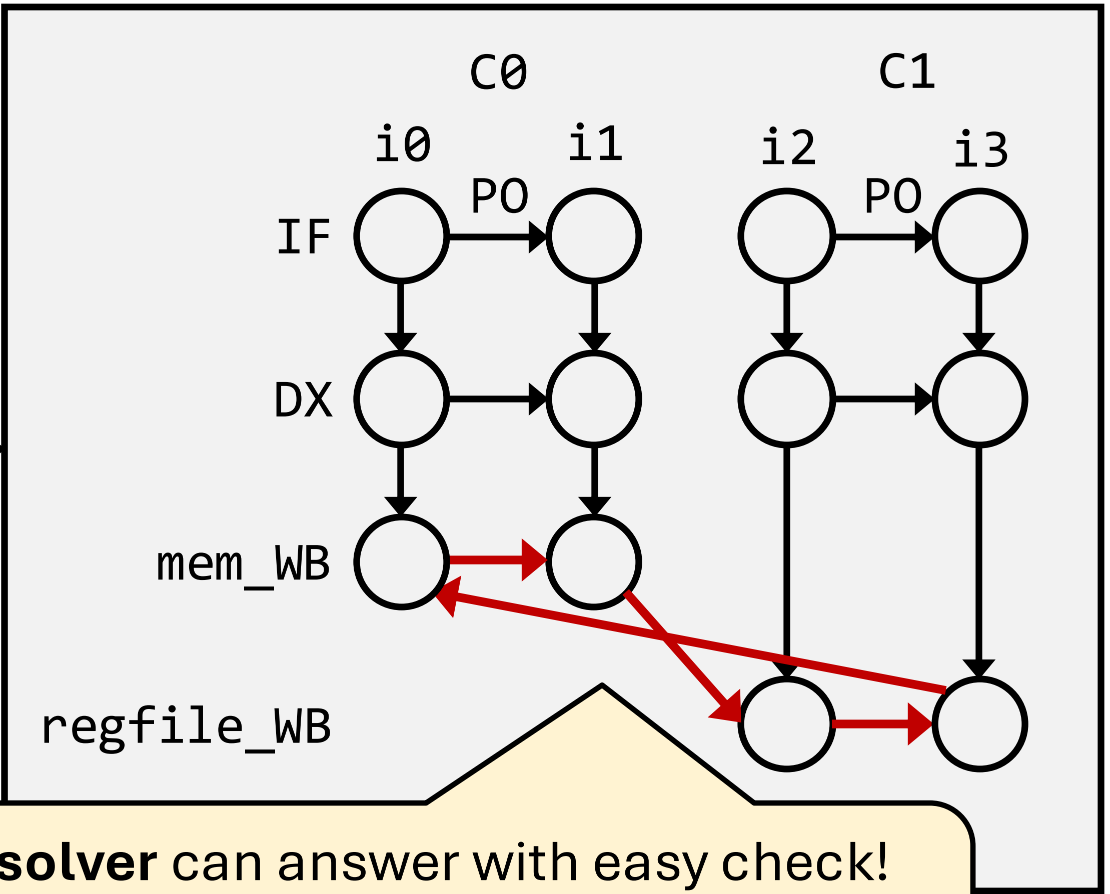
Is illegal outcome possible?

```
      C0                   C1
(i0) ST [data] 1   (i2) LD [flag] 1
(i1) ST [flag] 1   (i3) LD [data] 0
```



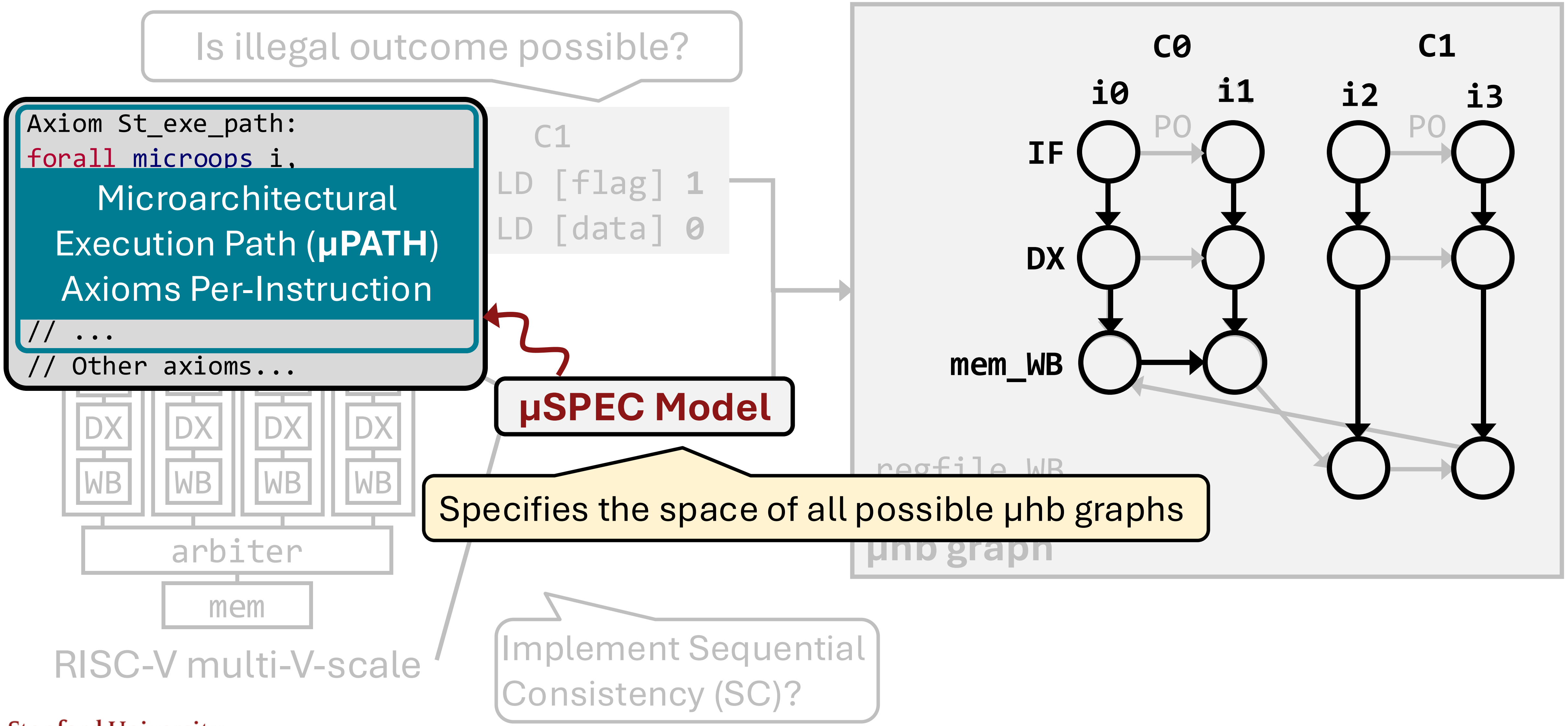
RISC-V multi-V-scale

μSPEC Model



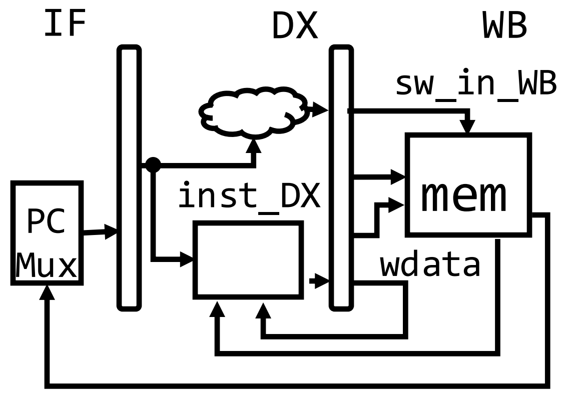
SAT or SMT solver can answer with easy check!
- **Cyclic**: Not observable
- **Acyclic:** Observable

Microarchitectural Happens-Before (µHB) Analysis Reasons About Observability of Hardware-level Program Execution [Lustig+, MICRO'14]

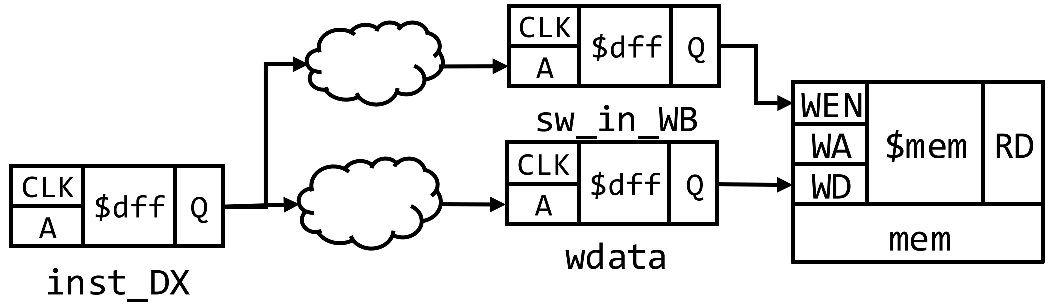


Is illegal outcome possible?
Axiom St_exe_path:
forall microops i,
Microarchitectural Execution Path (µPATH) Axioms Per-Instruction
// ...
// Other axioms...
C1
LD [flag] 1
LD [data] 0
DX
WB
arbiter
mem
RISC-V multi-V-scale
µSPEC Model
Specifies the space of all possible µhb graphs
C0
C1
i0
i1
i2
i3
PO
IF
DX
mem_WB
regfile_WB
µhb graph
Implement Sequential Consistency (SC)?

RTL2μSPEC: Synthesizing μSPEC from Processor Design



SystemVerilog design



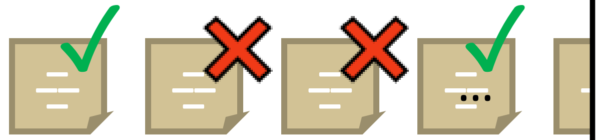
#1 Netlist a.k.a. control- and data-flow graph

```
P0: assume (first |-> ( (`PCR_0 != pc0 [*0:$]) ##1
        (`PCR_0 == pc0 [*1:$]) ##1 (`PCR_0 != pc0) ));
P1: assume (first |-> s_eventually(`PCR_<stage(s)> == pc0));
P2: assume (`PCR_0 == pc0 |-> `IFR == i0);
P3: assume (opcode(i0) == op);
A0: assert (`PCR_<stage(s)> == pc0 |-> s == $past(s));
```

#3 SVA Embedding w/ Templates

assert (property)

#4 JasperGold



#5 Comprehensive set of μSPEC axioms

Open-source RISC-V multi-V-scale case study [Hsiao+, MICRO'21]:

- **6.84 mins serial proof time** w/ 120 SVA properties evaluated
- **> 780x performance improvement** over prior work [Manerakr+, MICRO'17]

Roadmap Toward Automatic Synthesis of Verified µSPEC

- **Background**: The Microarchitecture-µSPEC Model Verification Challenge
- **RTL2µSPEC**: Synthesizing µSPEC model from Simple Processor RTL Designs
- **RTL2MµPATH**: Synthesizing ("Uncovering") All µPATHs per Instruction from Advanced SystemVerilog Processors
- Next Steps: Support synthesis of µSPEC axioms for coherence protocol and complex data dependencies in complex processors

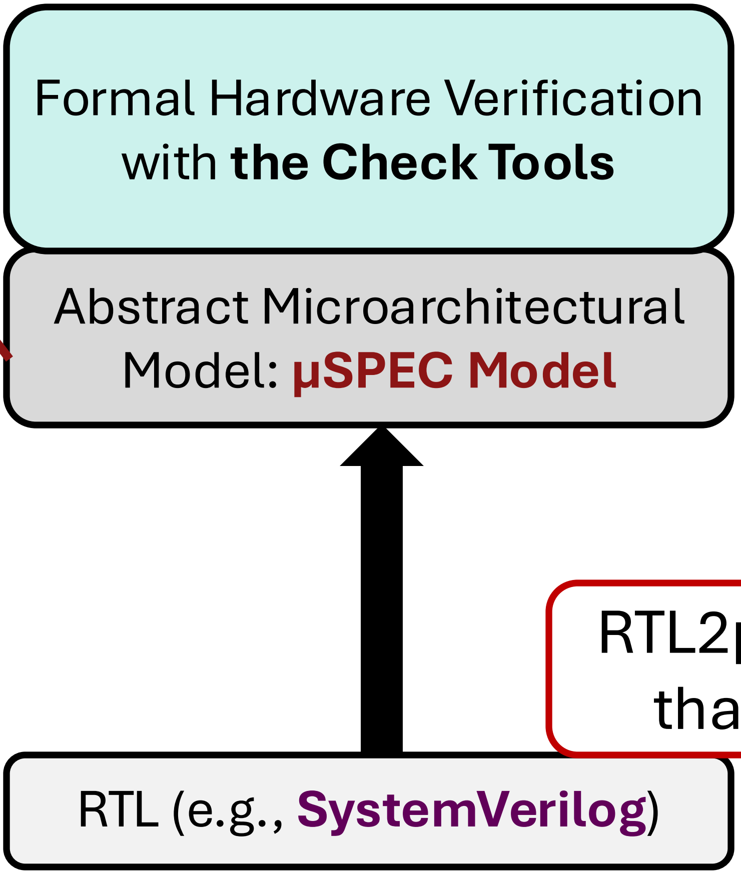
Single Microarchitectural Execution Path (µPATH) Assumption

```
Axiom St_exe_path:
forall microops i,
```

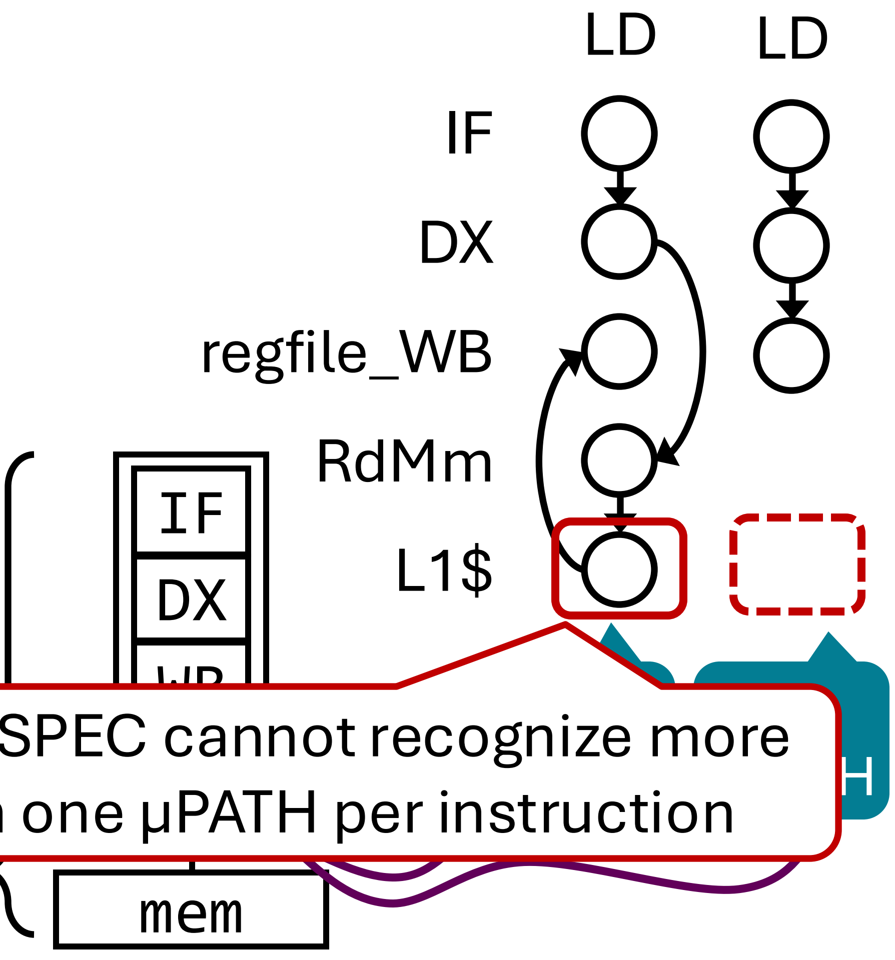
Microarchitectural Execution Path (µPATH) Axioms Per-Instruction

```
// ...
// Other axioms...
```

µPATH: Models a **specific execution** of a **specific dynamic instruction** on a **specific microarchitecture** as a directed µHB graph [Lustig+, MICRO'14].

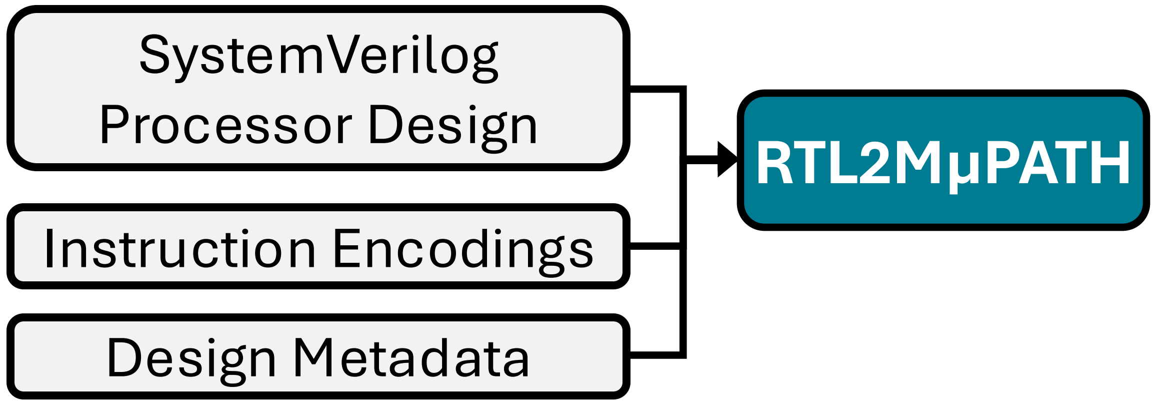

Formal Hardware Verification with **the Check Tools**
Abstract Microarchitectural Model: **µSPEC Model**
RTL (e.g., **SystemVerilog**)

All LD µPATHs:



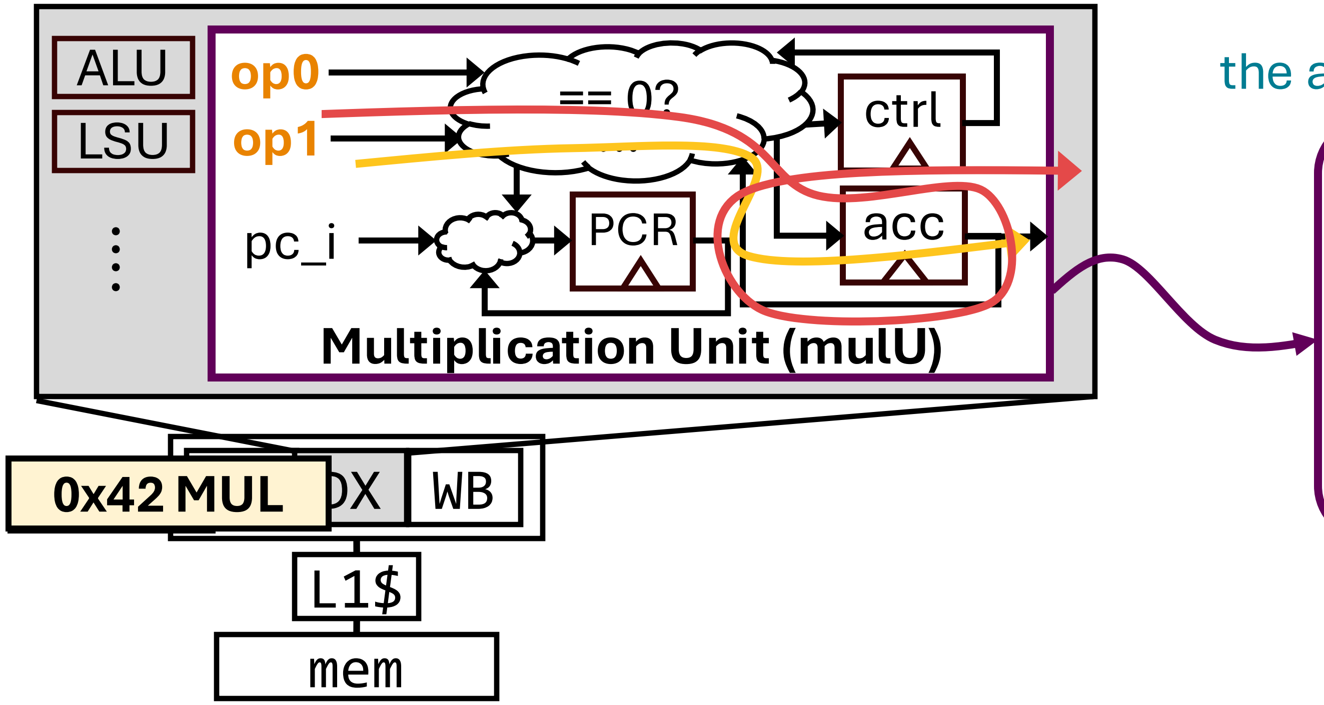
RTL2µSPEC cannot recognize more than one µPATH per instruction

Overview of RTL2MμPATH: Multi-μPATH Synthesis from RTL



Stanford University

Overview of RTL2MμPATH: Multi-μPATH Synthesis from RTL

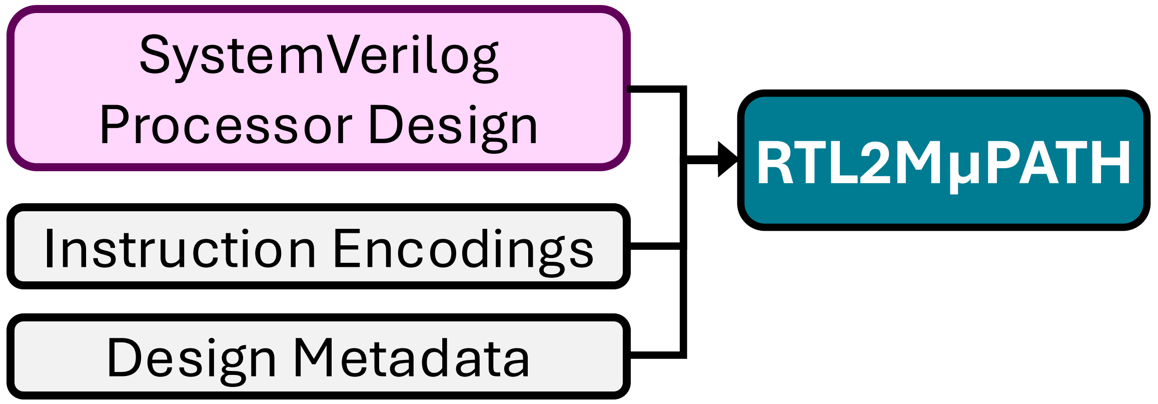


visits mulU by putting its PC value the

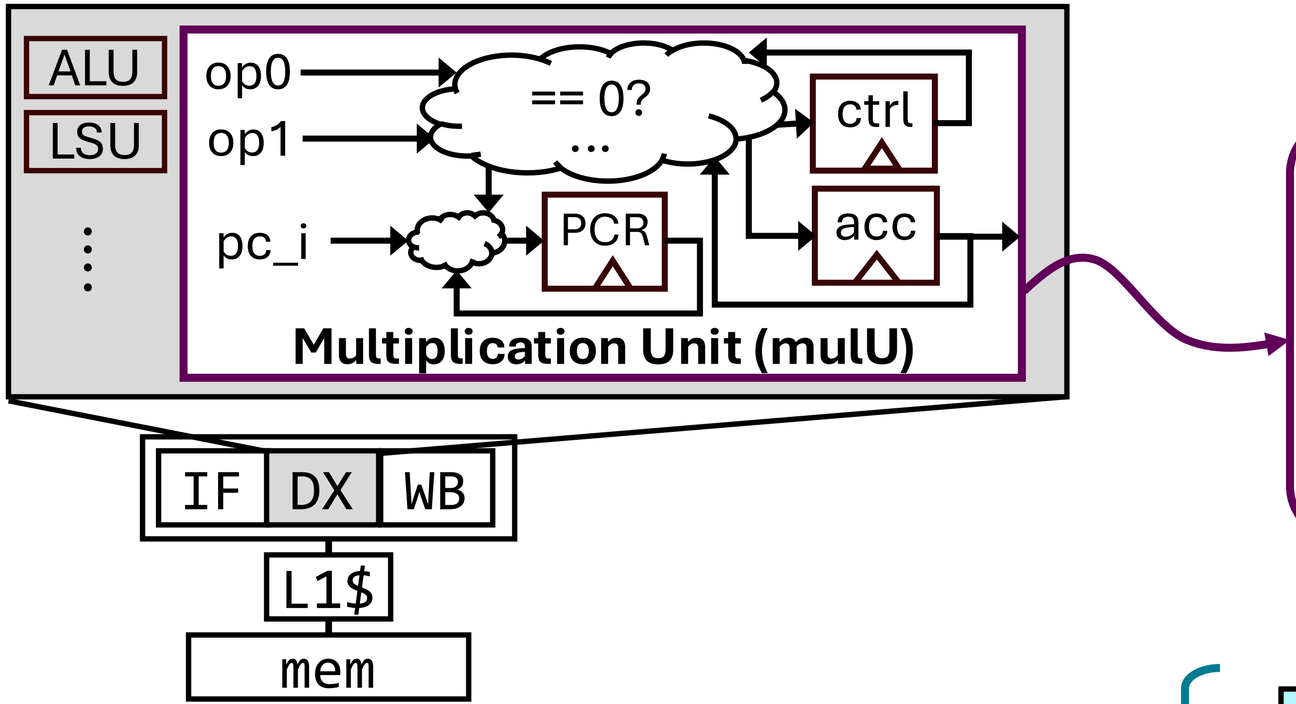
the active i mulU's MUL's own (N) operands

```
MUL_mulU(MUL^N i0):
 if (i0.op0 == 0 ||
     i0.op1 == 0):
   return fast_path  # 1 cyc
 return slow_path    # 3 cyc
```

Zero-Skip Optimization

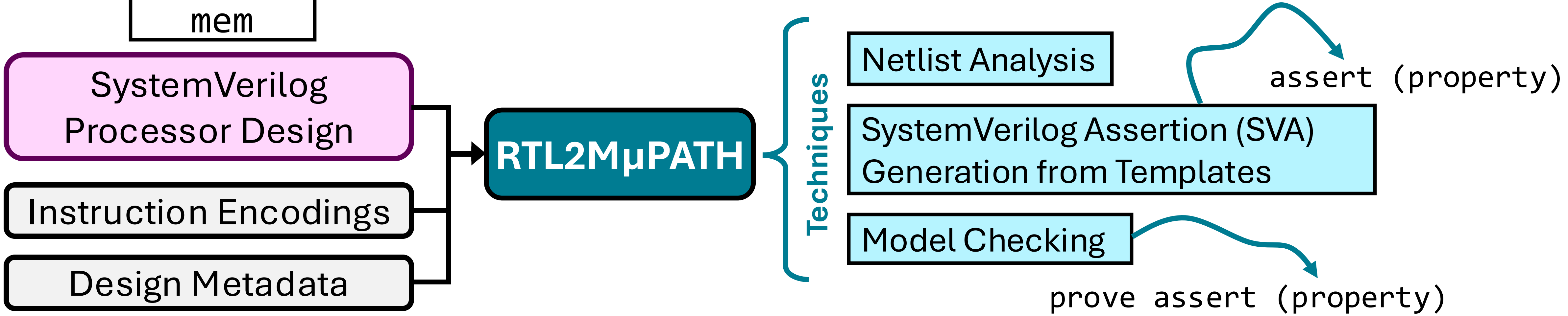


Overview of RTL2MμPATH: Multi-μPATH Synthesis from RTL



```
MUL_mulU(MUL^N i0):
 if (i0.op0 == 0 ||
     i0.op1 == 0):
   return fast_path  # 1 cyc
return slow_path    # 3 cyc
```

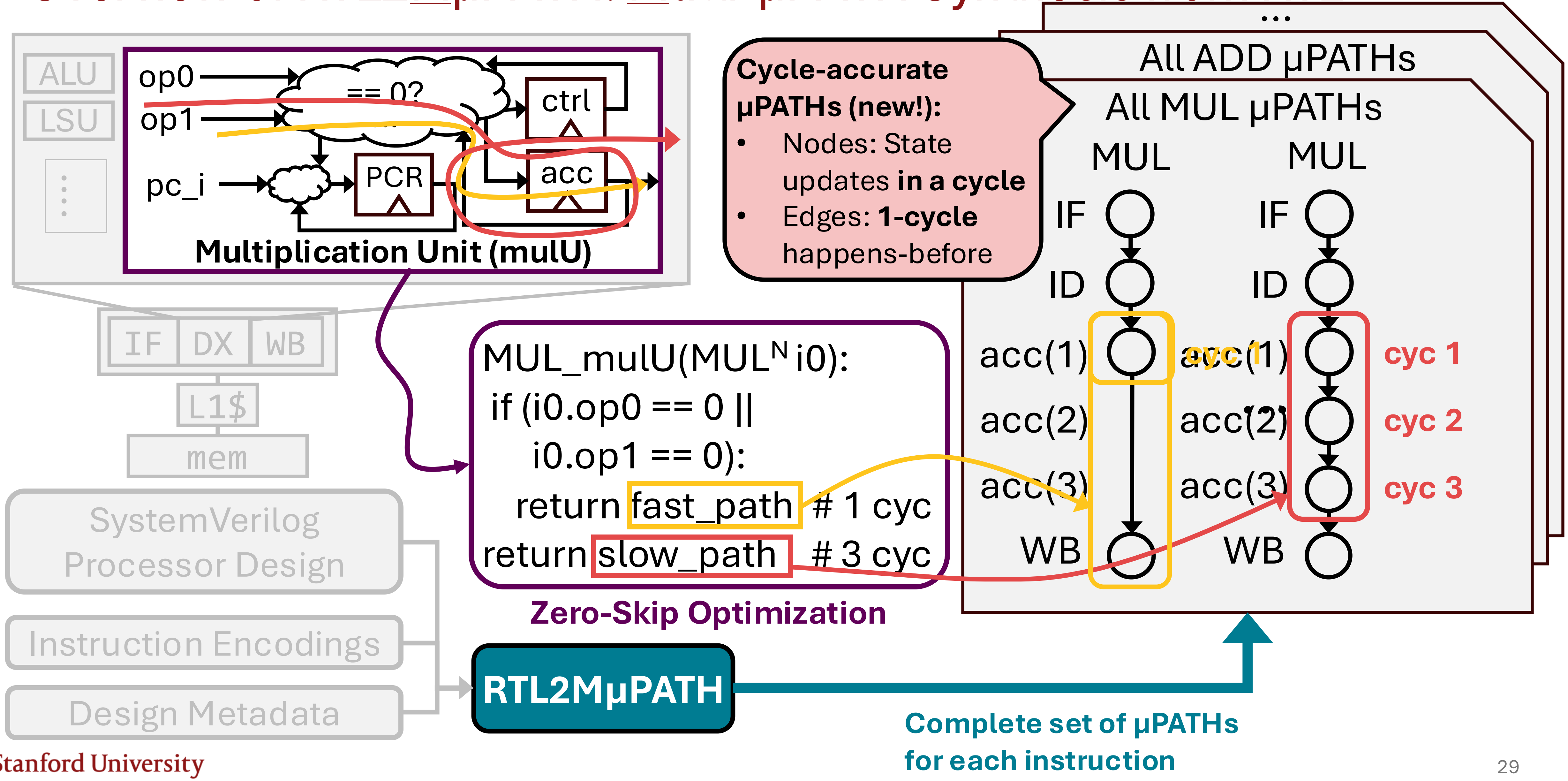
Zero-Skip Optimization



```
assert (property)
```

```
prove assert (property)
```

Overview of RTL2MμPATH: Multi-μPATH Synthesis from RTL



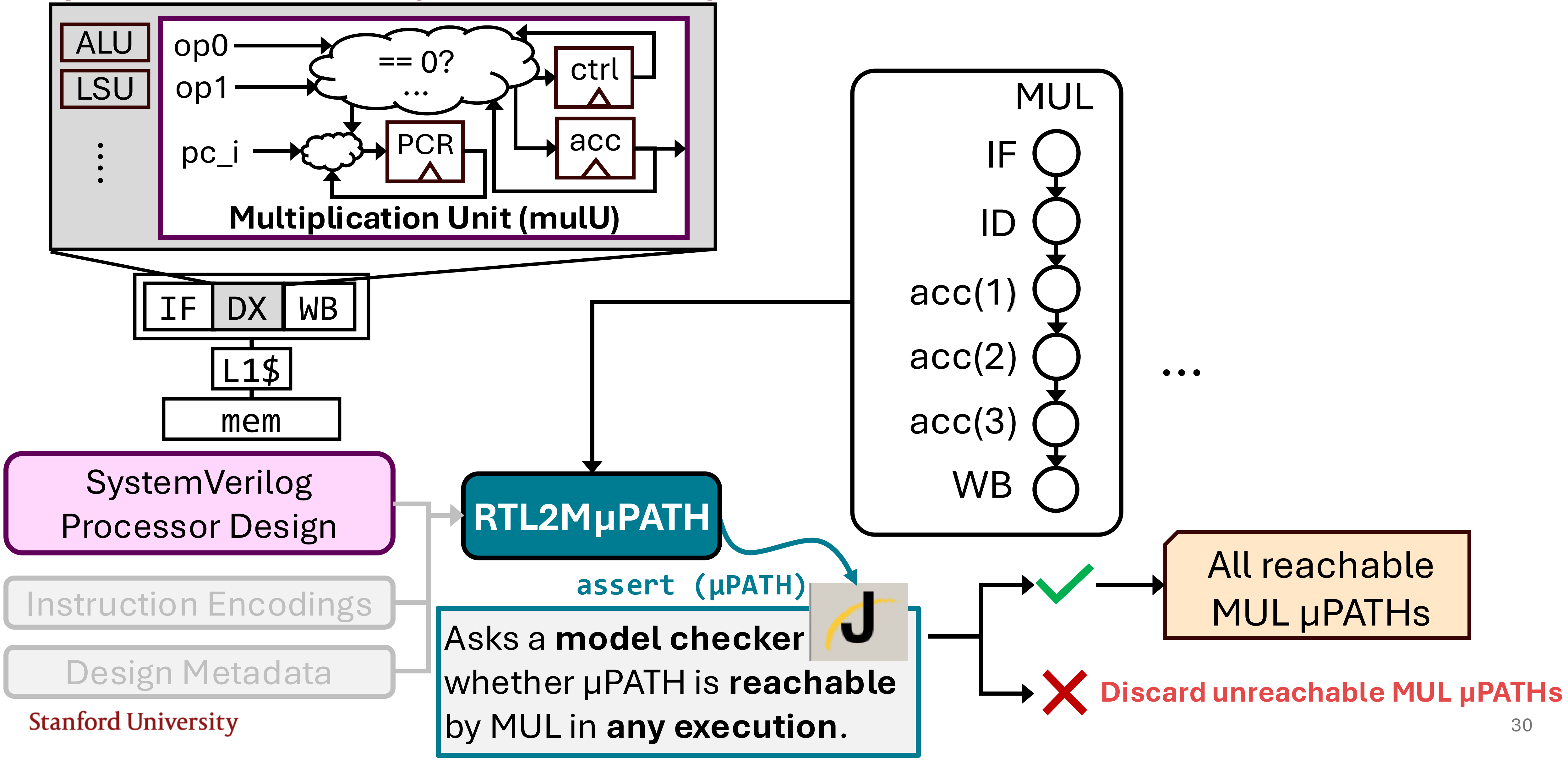
Cycle-accurate μPATHs (new!):

- Nodes: State updates **in a cycle**
- Edges: **1-cycle** happens-before

```
MUL_mulU(MUL^N i0):
 if (i0.op0 == 0 ||
     i0.op1 == 0):
   return fast_path  # 1 cyc
return slow_path     # 3 cyc
```

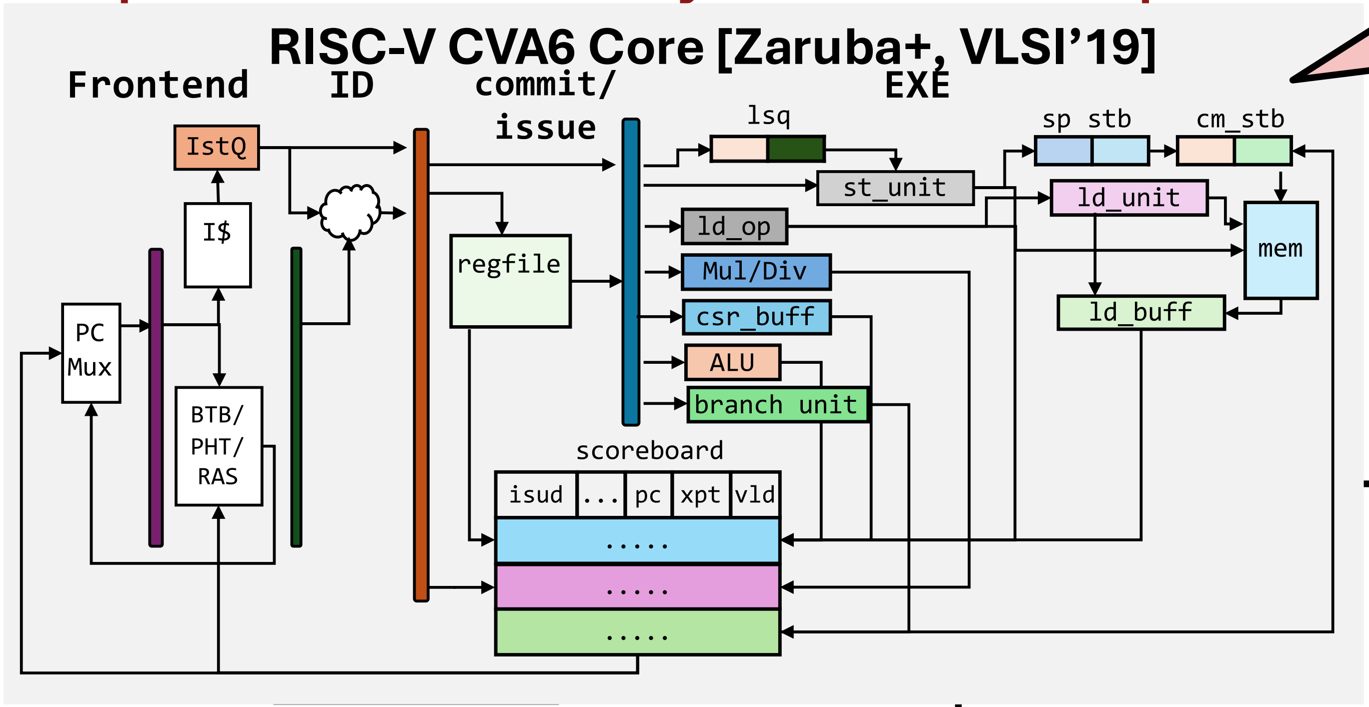
Stanford University

Conceptualizing Nodes in a μPATH: A Key Challenge to Automated μPATH Discovery with RTL2MμPATH

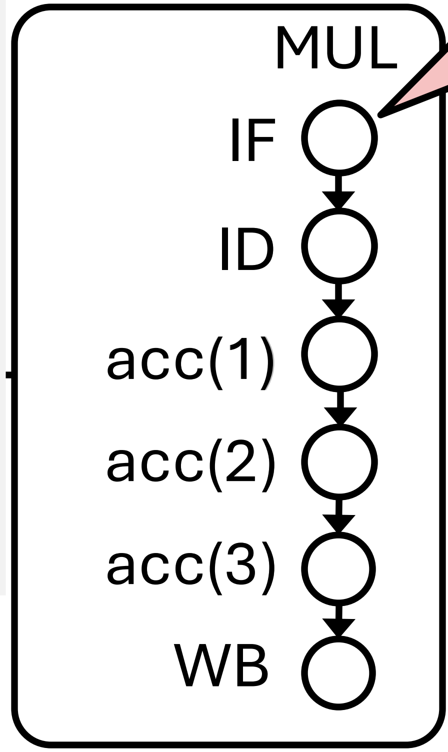


Asks a **model checker** whether μPATH is **reachable** by MUL in **any execution**.

Conceptualizing Nodes in a μPATH: A Key Challenge to Automated μPATH Discovery with RTL2MμPATH



Many instructions (colors) **in-flight at the same time!**



How to recognize a node? Requires **detecting** & **attributing** state update to specific instructions

...

SystemVerilog Processor Design

Instruction Encodings

Design Metadata

RTL2MμPATH

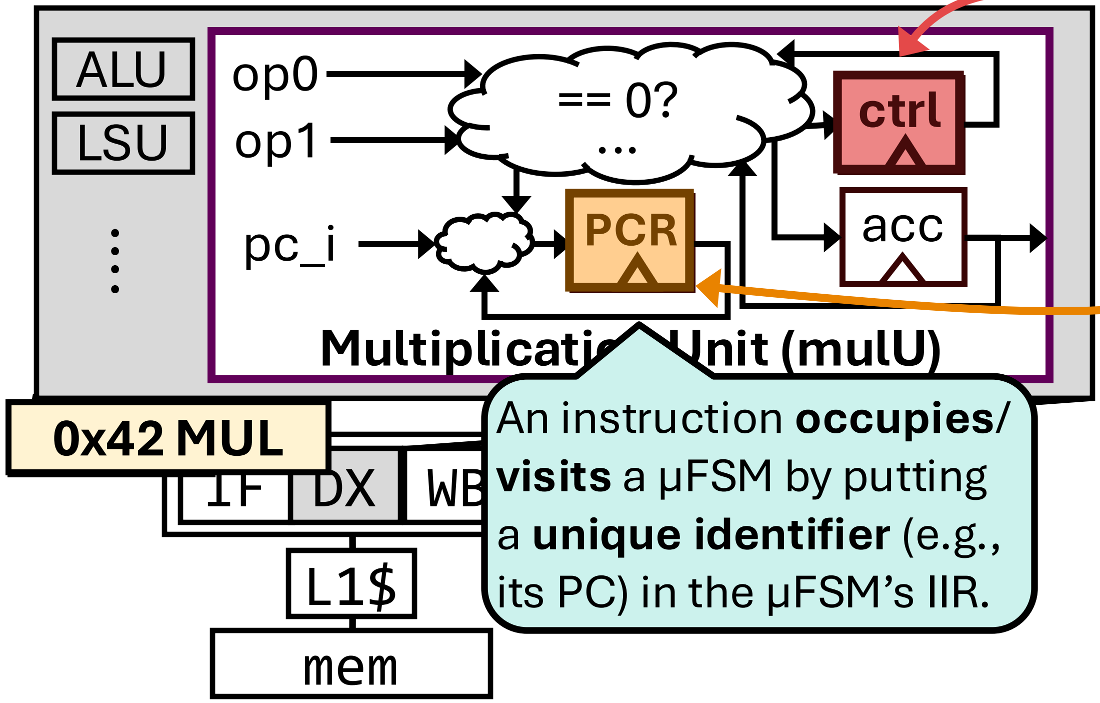
assert (μPATH)

Asks a **model checker** whether μPATH is **reachable** by MUL in **any execution.**

All reachable MUL μPATHs

Discard unreachable MUL μPATHs

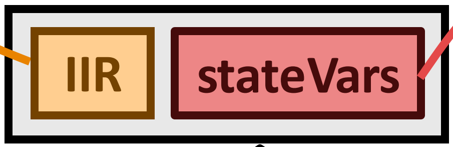
Our Solution: Expressing Nodes in μPATHs using Micro-op Finite State Machines (μFSMs)



An instruction **occupies/visits** a μFSM by putting a **unique identifier** (e.g., its PC) in the μFSM's IIR.

SystemVerilog Processor Design

Instruction-identifying register (IIR), e.g., register holding PC value.

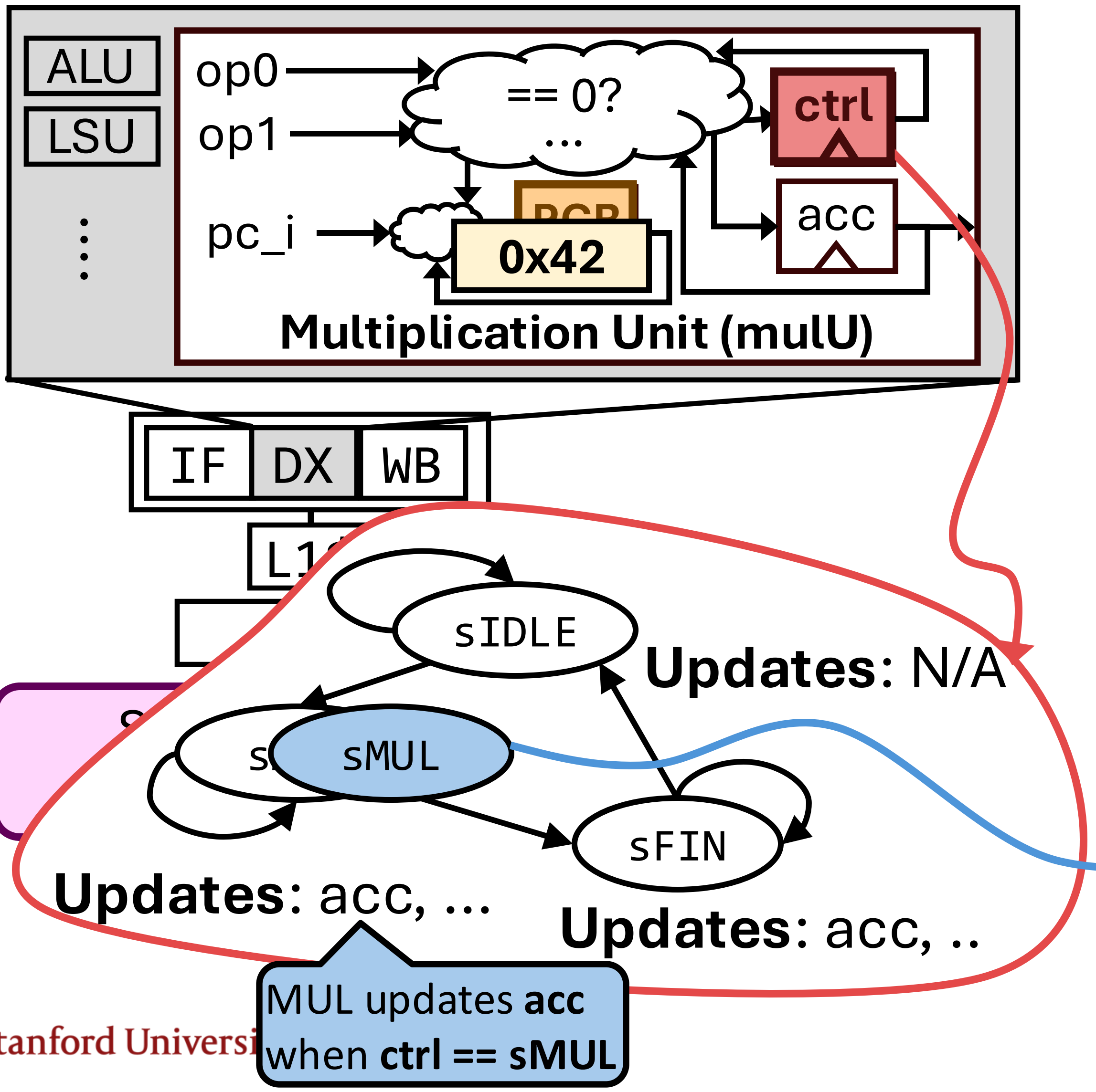


State variables, encoding a concrete FSM state

Micro-op Finite State Machine (μFSM):

- **<IIR, stateVars>** tuple
- Orchestrate instruction execution from fetch until possibly after commit
- Control instruction state updates per-cycle

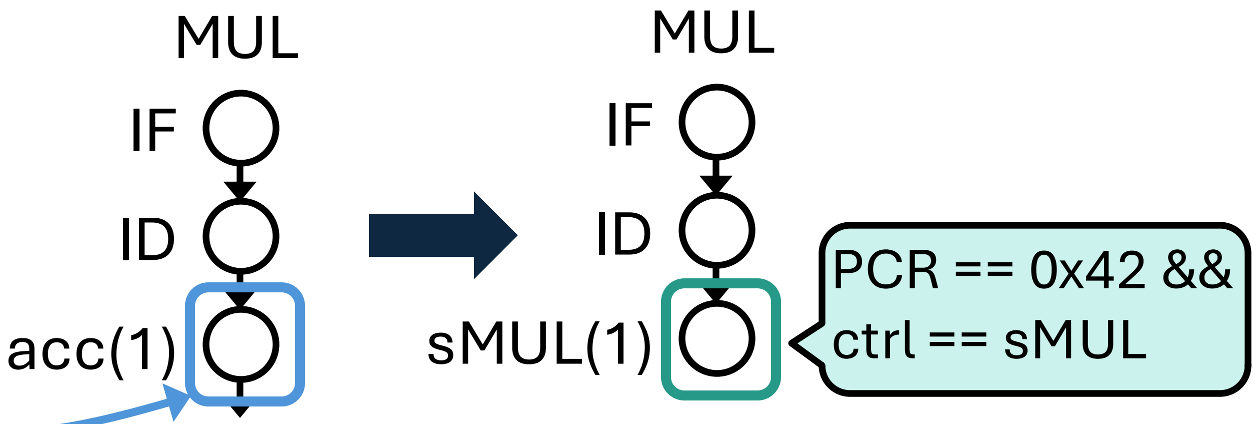
Our Solution: Expressing Nodes in µPATHs using Micro-op Finite State Machines (µFSMs)



MUL updates **acc** when **ctrl == sMUL**

IIR | stateVars

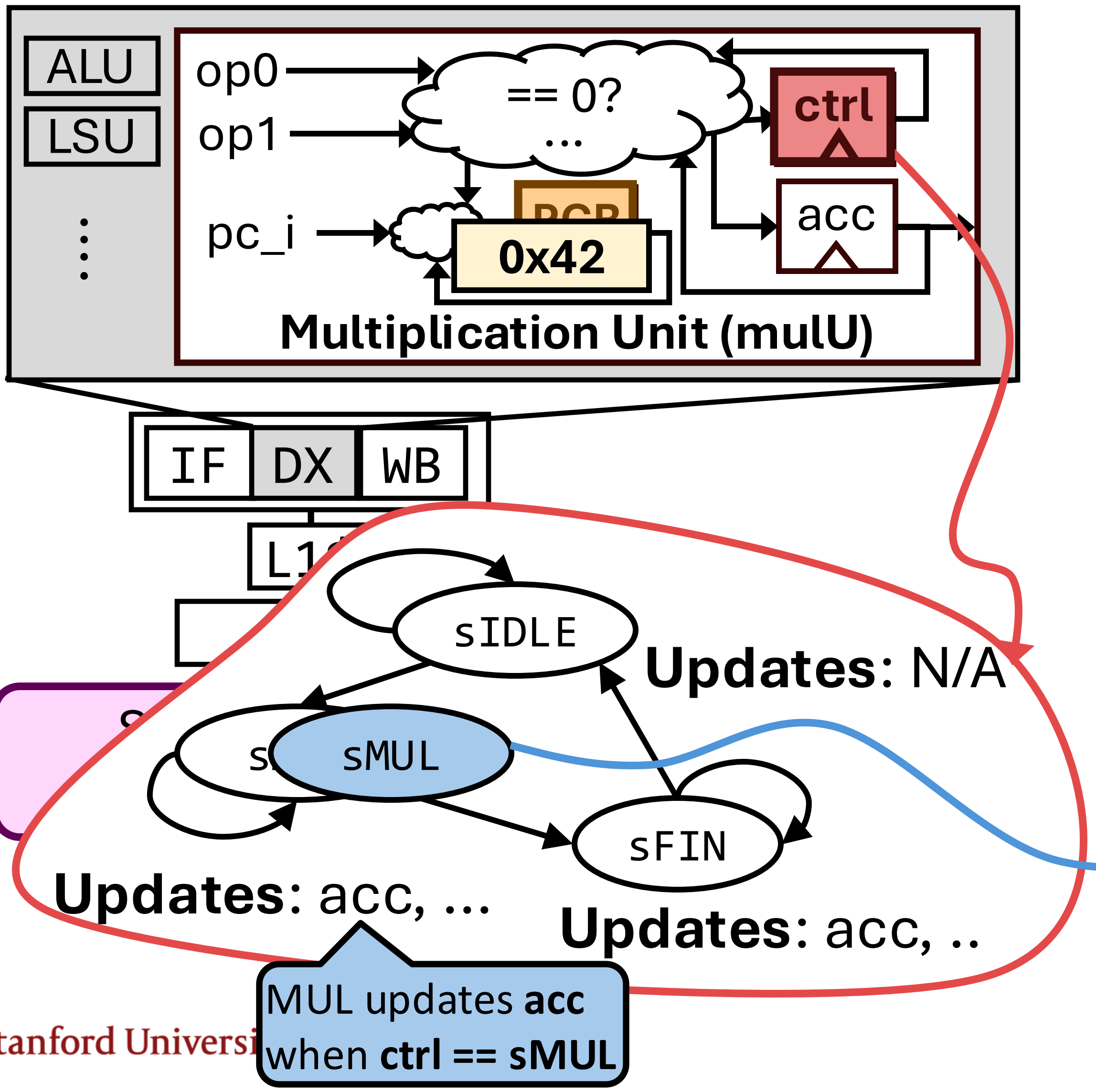
Micro-op Finite State Machine (µFSM):
- **<IIR, stateVars>** tuple
- Orchestrate instruction execution from fetch until possibly after commit
- Control instruction state updates per-cycle



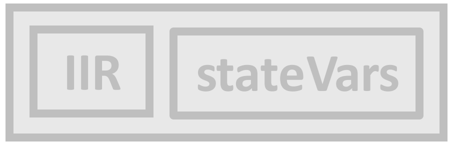
PCR == 0x42 && ctrl == sMUL

A µHB node ↔ A µFSM in a **non-idle state** and **occupied** by an instruction

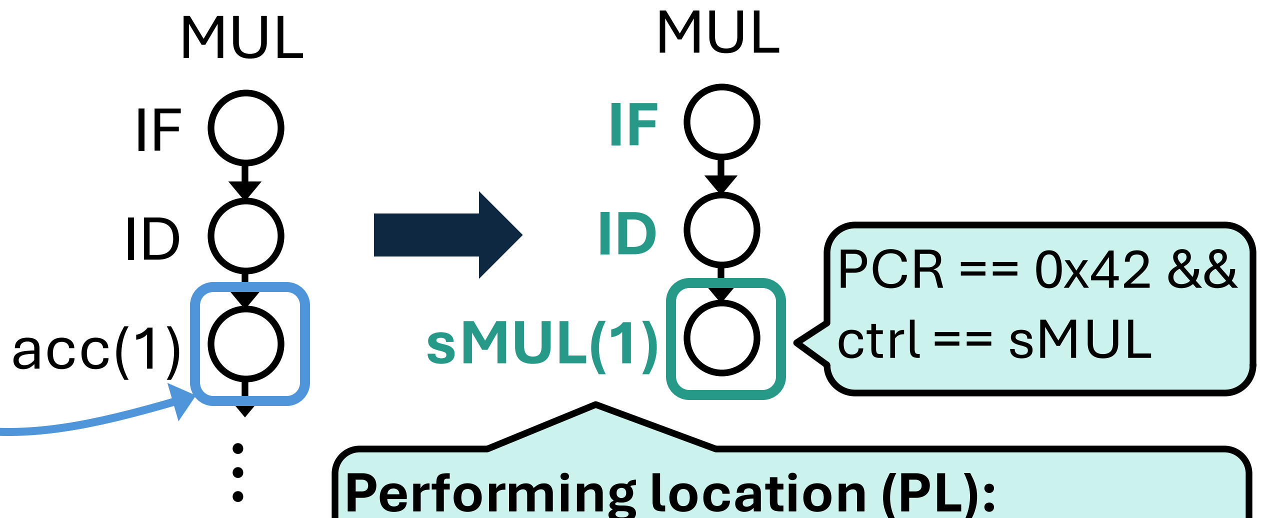
Our Solution: Expressing Nodes in μPATHs using Micro-op Finite State Machines (μFSMs)



MUL updates **acc** when **ctrl == sMUL**



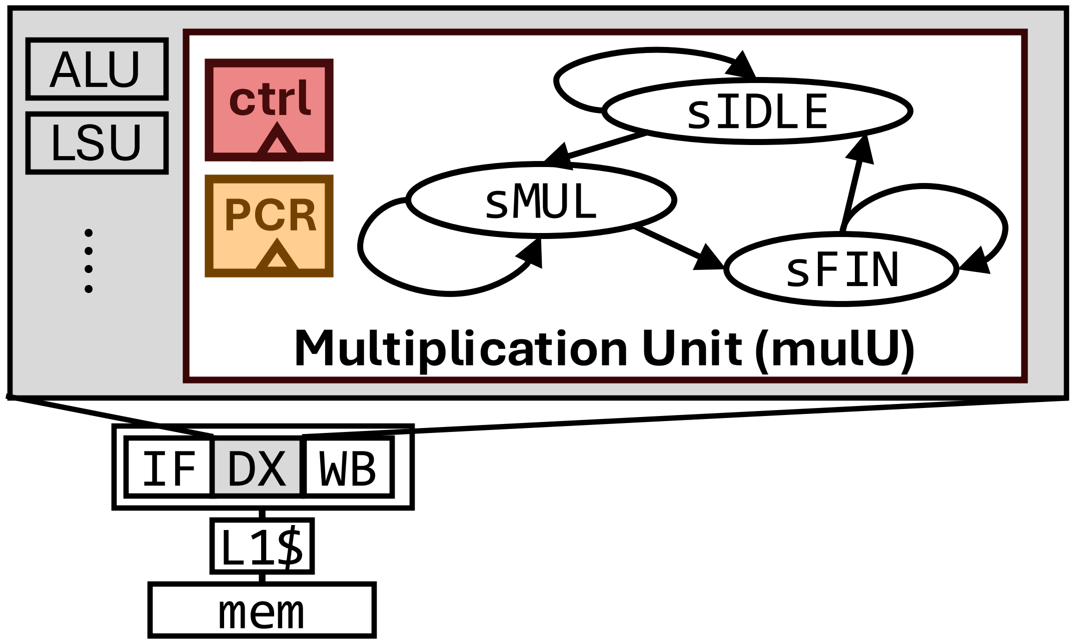
Micro-op Finite State Machine (μFSM):
- **<IIR, stateVars>** tuple
- Orchestrate instruction execution from fetch until possibly after commit
- Control instruction state updates per-cycle

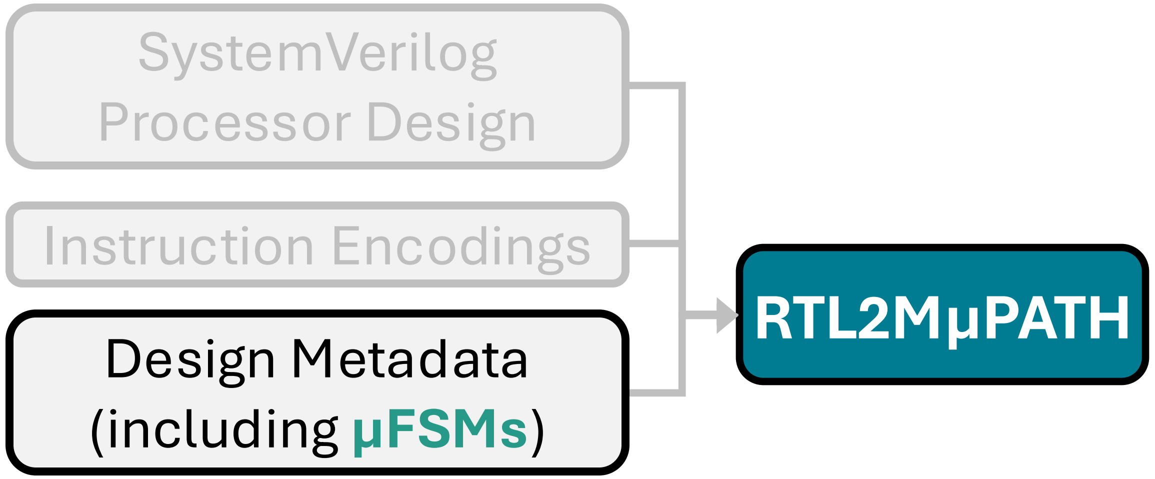


PCR == 0x42 && ctrl == sMUL

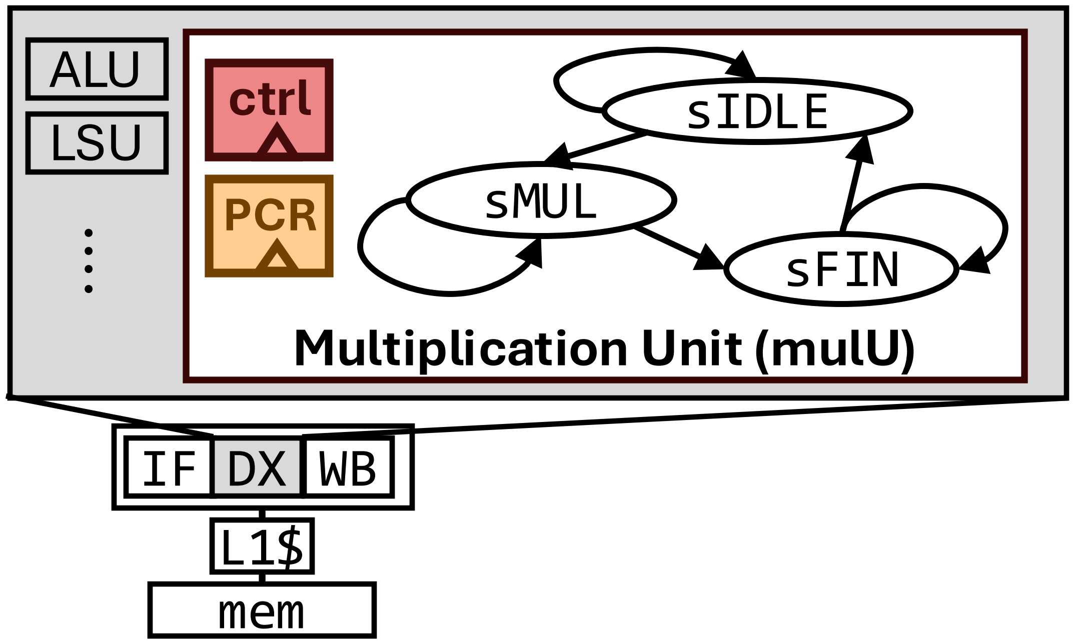
Performing location (PL): Concrete non-idle state of a μFSM

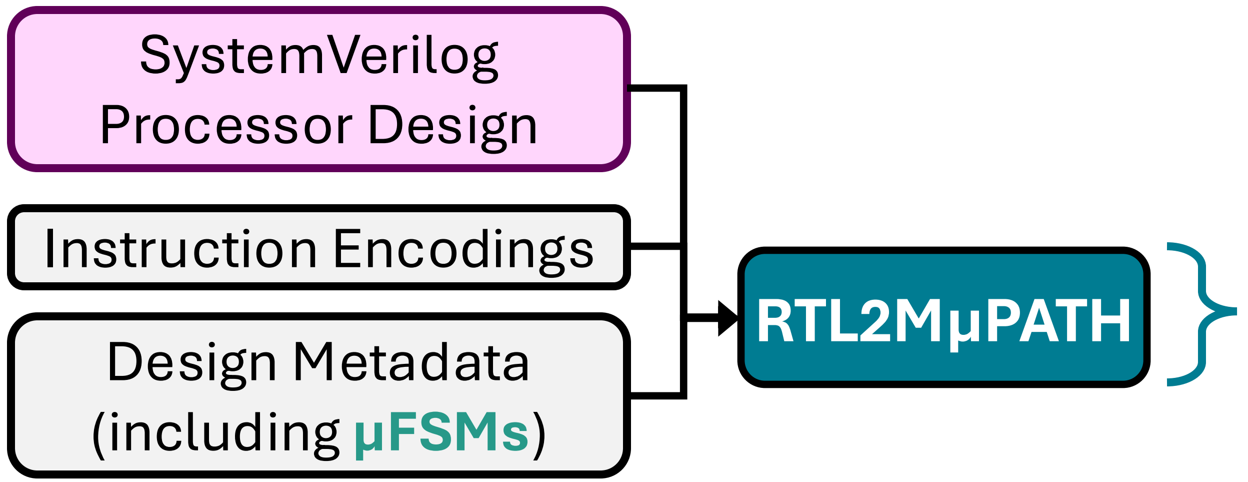
RTL2MμPATH: Synthesizing μPATHs from Processor Design




SystemVerilog Processor Design
Instruction Encodings
Design Metadata (including μFSMs)
RTL2MμPATH

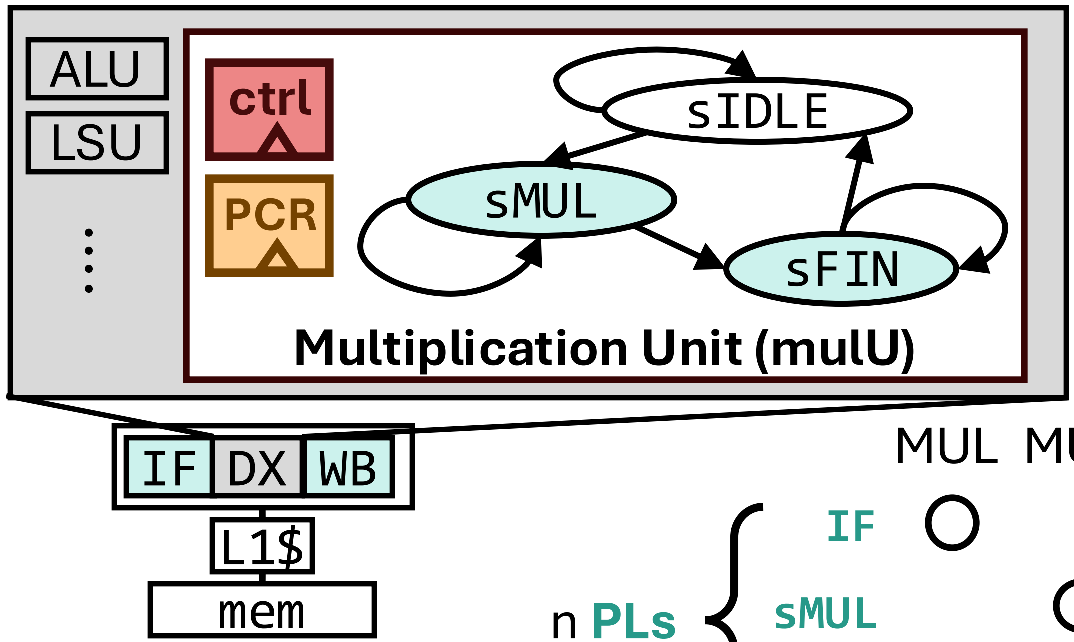
RTL2MµPATH: Synthesizing µPATHs from Processor Design

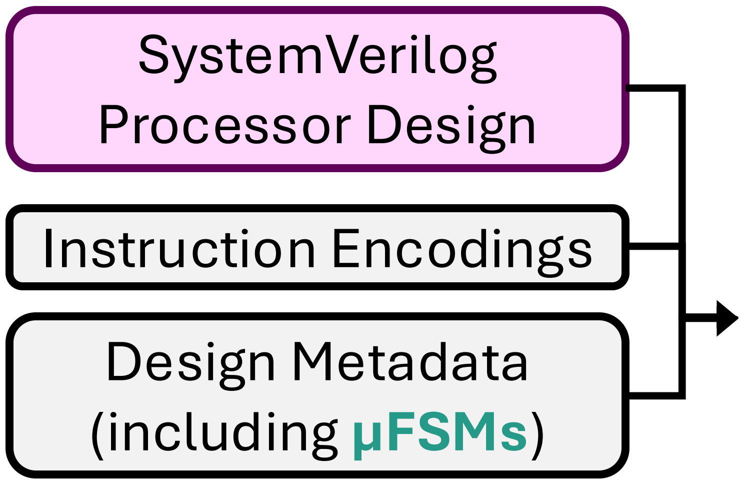



SystemVerilog Processor Design
Instruction Encodings
Design Metadata (including µFSMs)
RTL2MµPATH

Step 1: Synthesizing **sets of nodes** that can form reachable µPATHs

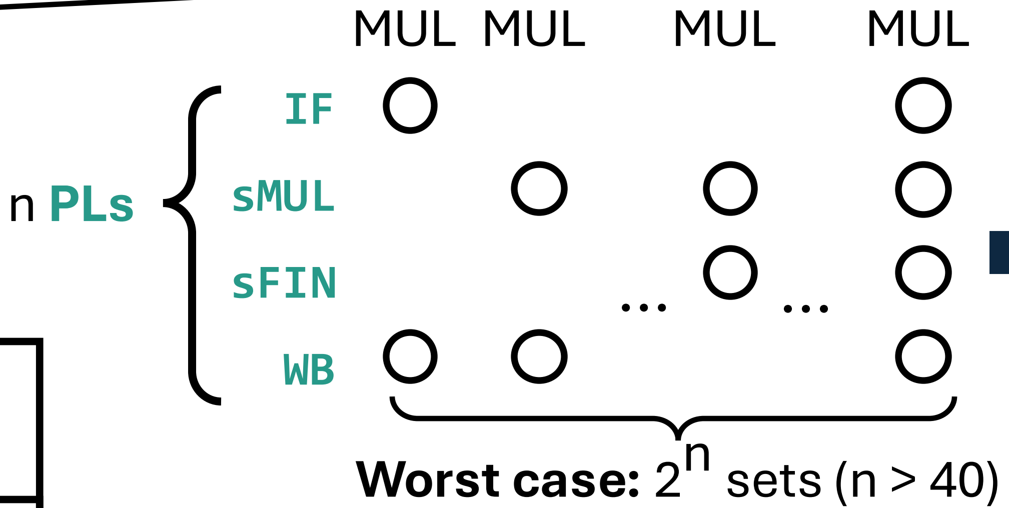
RTL2MμPATH: Synthesizing μPATHs from Processor Design





SystemVerilog Processor Design
Instruction Encodings
Design Metadata (including μFSMs)

Step 1: Synthesizing **sets of nodes** that can form reachable μPATHs



Worst case: 2^n sets (n > 40)

Step 1A: Enumerate all possible **PLs** (concrete **μFSM states**) with netlist analysis

Ask a model checker...

```
Can any instruction
visit <PL>?
```

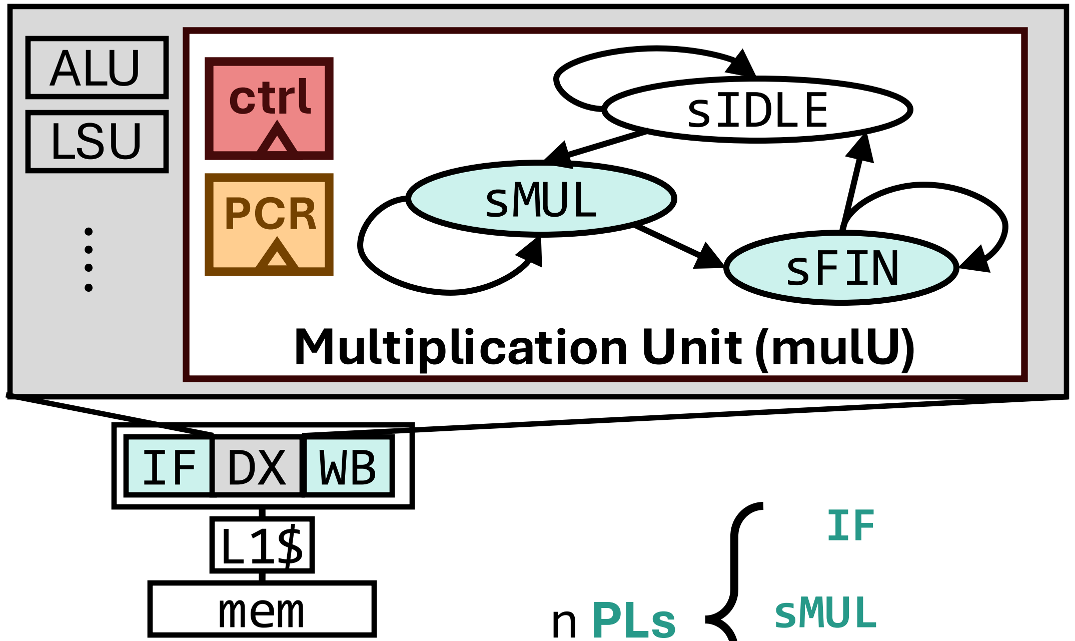
```
Can <inst> visit <PL>?
```

```
Can <inst> visit <PL1>
without visiting <PL2>?
```

```
Can <inst> visit both
<PL1> and <PL2>?
```

Step 1B: SVA property-driven μPATH pruning

RTL2MμPATH: Synthesizing μPATHs from Processor Design



SystemVerilog Processor Design

Instruction Encodings

Design Metadata (including **μFSMs**)

n **PLs** { IF, sMUL, sFIN, WB }

Step 1: Synthesizing **sets of nodes** that can form reachable μPATHs

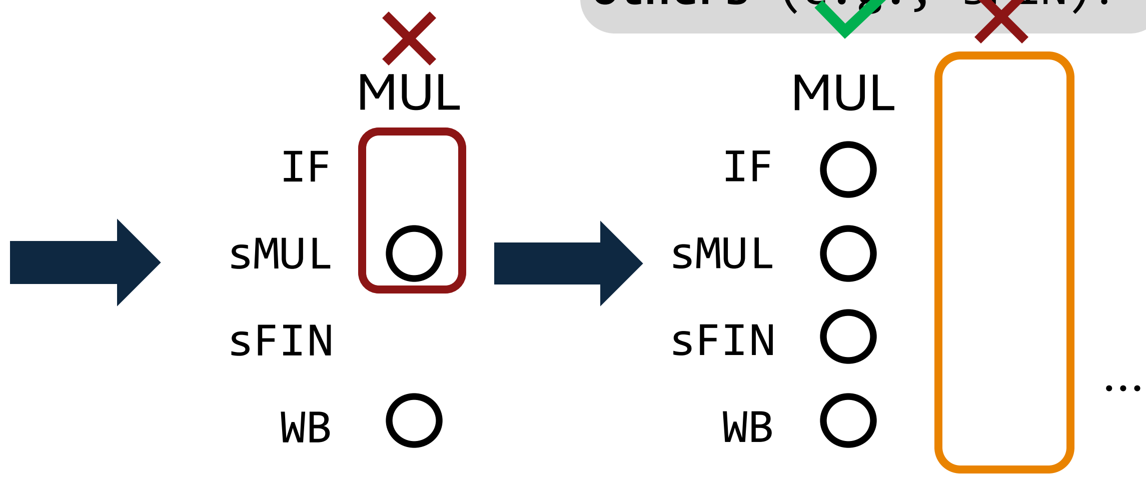
Step 2: Synthesizing **full μPATHs** by adding edges to reachable node sets

Ask a model chec

Can <MUL> visit <IF

sMUL, WB> **but no others** (e.g., sFIN)?

Details in the paper [Hsiao, MICRO’24]!



Candidate sets of nodes (**10s to ~1k**)

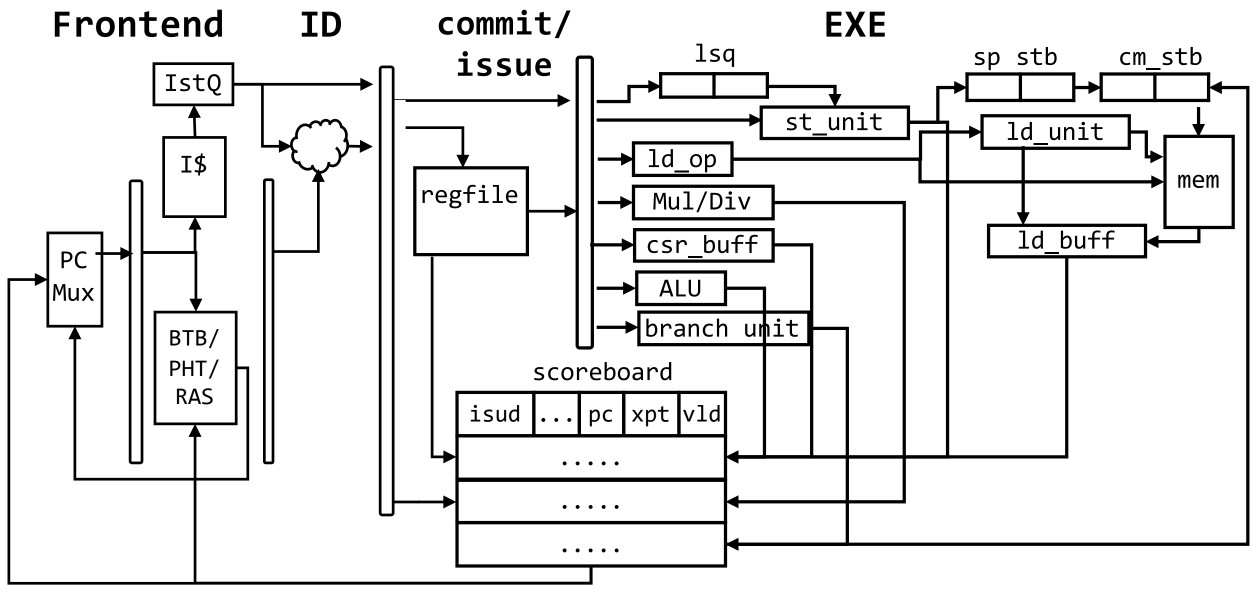
Step 1A: Enumerate all possible **PLs** (concrete **μFSM states**) with netlist analysis

Step 1B: SVA property-driven μPATH pruning

Step 1C: Embed node sets as SVA properties to deduce reachability

CVA6 Core and Cache Case Study [Hsiao+, MICRO'24]

- RISC-V CVA6
  - 64-bit, 6-stage, single-issue core
  - **Speculation** and limited **out-of-order write-back** with diverse functional units (ALU, LSU, Mul/Div, CSR buffer)
- 72 instructions in **RV64I base ISA + M extension** (RV64IM)
- Synthesize per-instruction μPATH axioms from **Core** and **Data Cache** respectively



CVA6 Core [Zaruba+, VLSI'19]

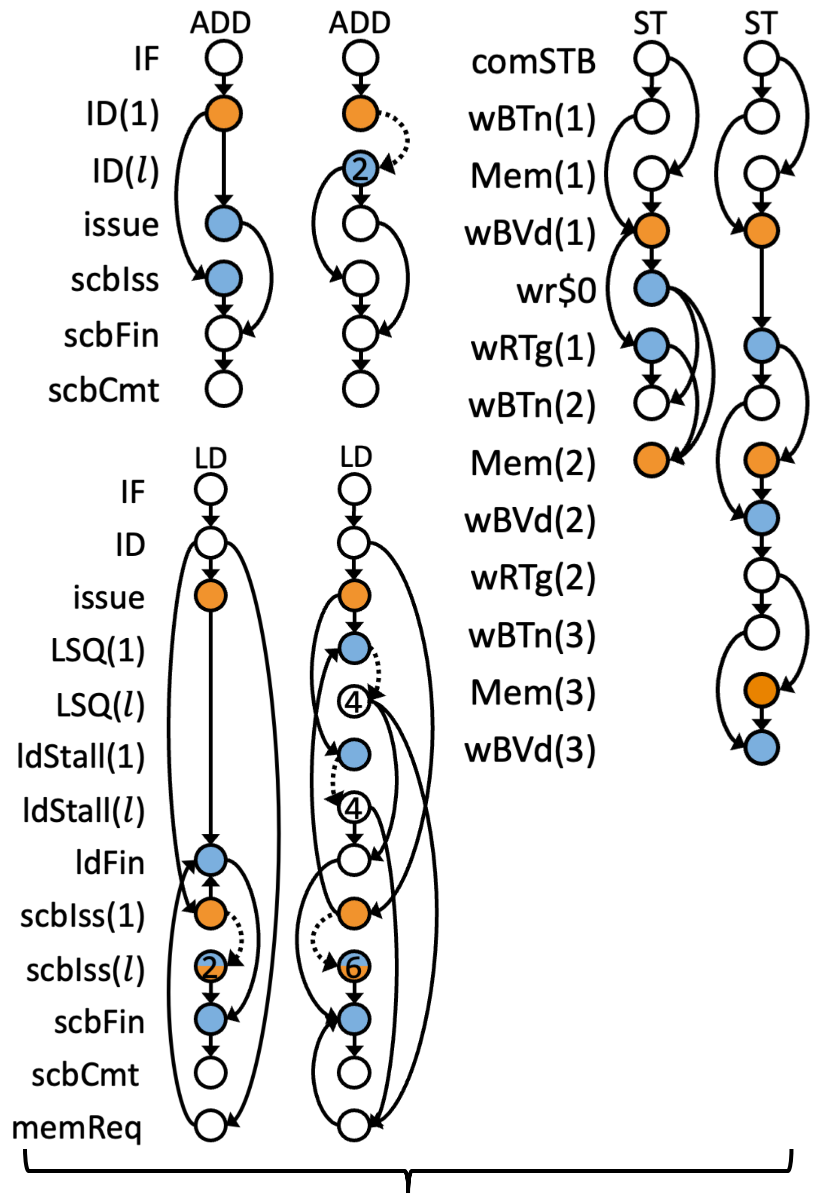
CVA6 Core: Results

- Complexity statistics: 8,577 LoC (SystemVerilog); after elaboration: 22,138 wires, 19,575 standard cells, 482 registers (11,985 D flip-flop bits), 3 memories.
- 124,459 properties
- Average ~4 min per property
- ~16% undetermined under timeout of 30 minutes

CVA6 Cache: Preliminary Results

- Complexity statistics: 2,279 LoC (SystemVerilog); 4-way, 128B (scaled down from 32 KB), write-through, coalescing write-buffer
- 4,178 properties
- **Average < 3 sec per property**
- **All completed**

Benefits of modularization



https://github.com/yaohsiaopid/SynthLC

Stanford University

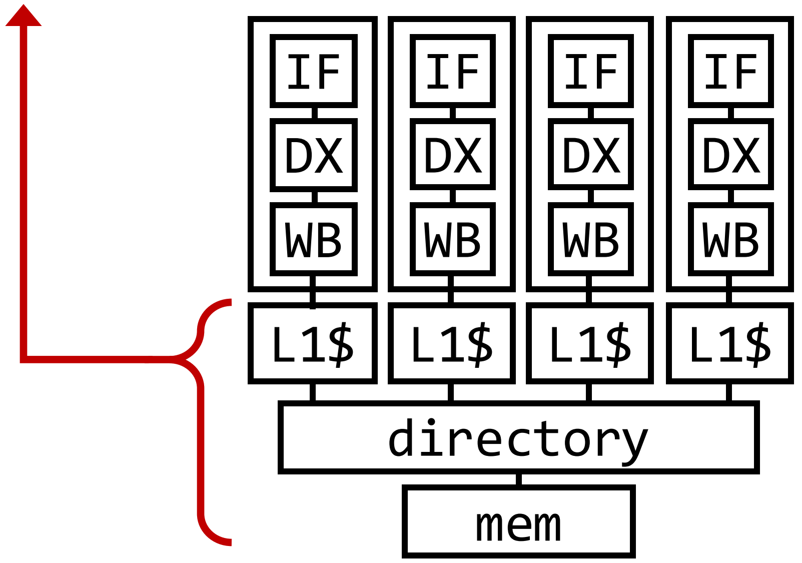
Roadmap Toward Automatic Synthesis of Verified µSPEC

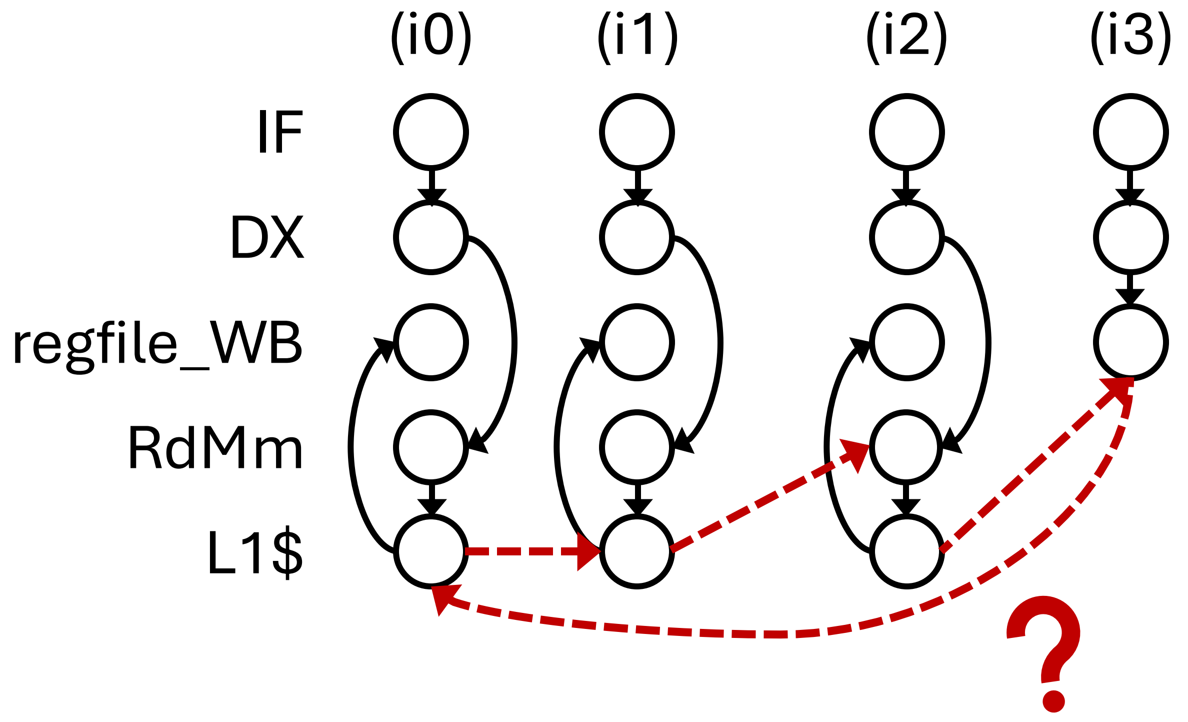
- **Background**: The Microarchitecture-µSPEC Model Verification Challenge
- **RTL2µSPEC**: Synthesizing µSPEC model from Simple Processor RTL Designs
- **RTL2MµPATH**: Synthesizing ("Uncovering") All µPATHs per Instruction from Advanced SystemVerilog Processors
- **Next Steps**: Support synthesis of µSPEC axioms for **coherence protocol** and **complex data dependencies** in complex processors

Challenge #1: Synthesizing coherence protocol related axioms

```
        C0                  C1
(i0) ST [data] 1  (i2) LD [data] 2
(i1) ST [data] 2  (i3) LD [data] 1
```

Cache coherence protocols ensure that multiple cached copies of an address are kept up-to-date

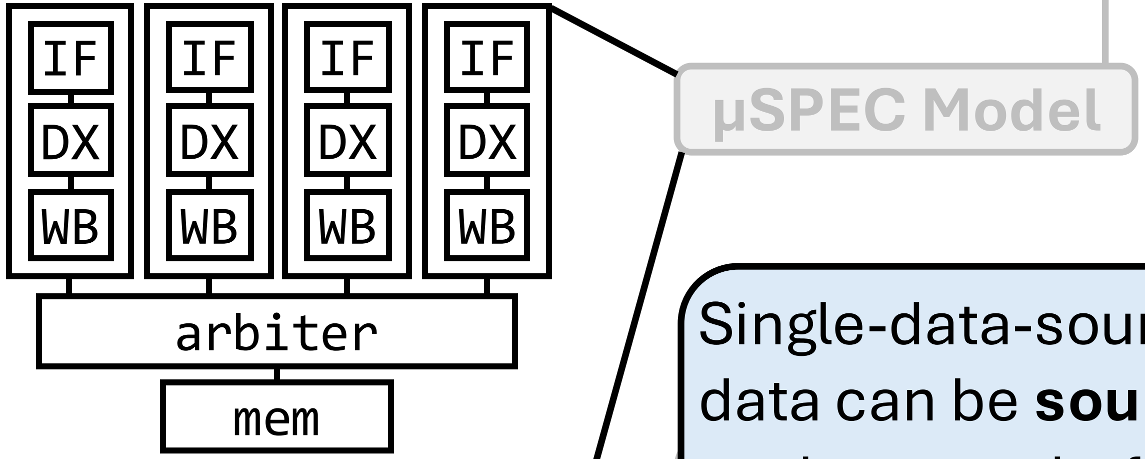




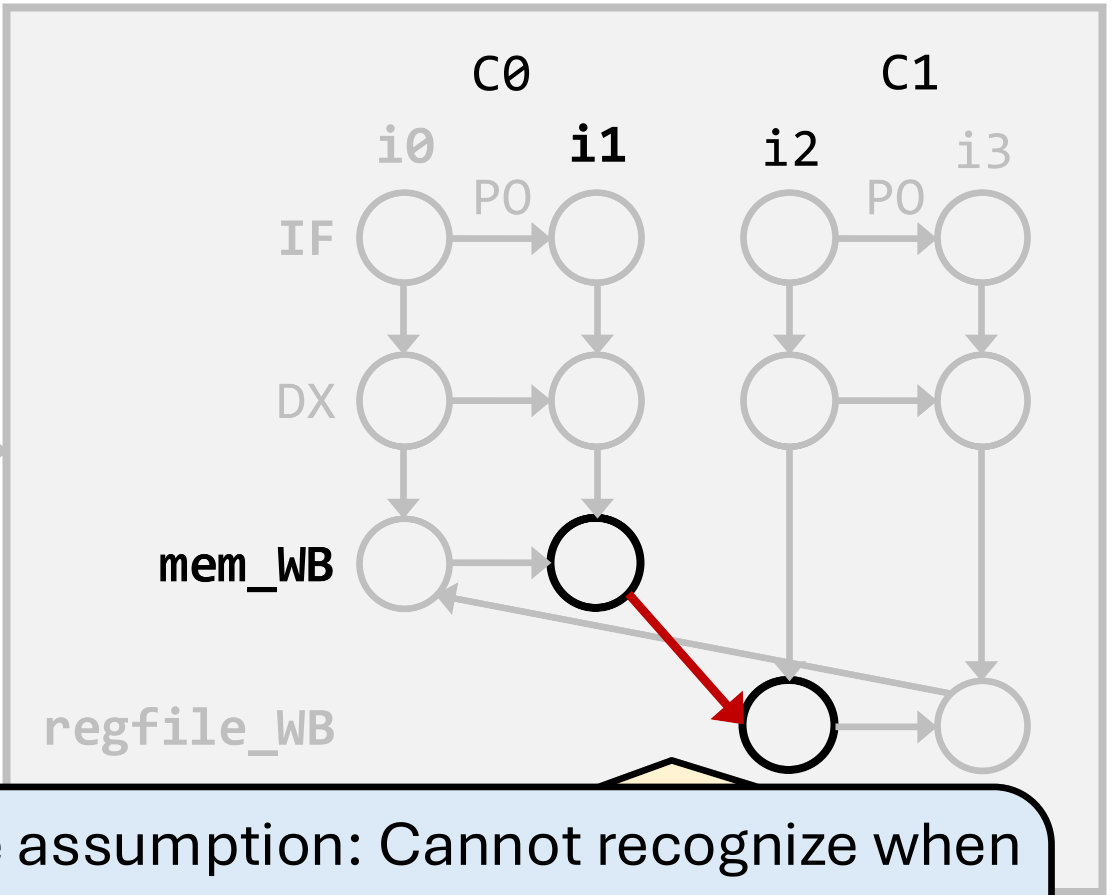
Challenge #2: axioms synthesis regarding complex dataflow dependencies

Forbidden on SC

C0	C1
(i0) ST [data] 1	(i2) LD [flag] 1
(i1) ST [flag] 1	(i3) LD [data] 0



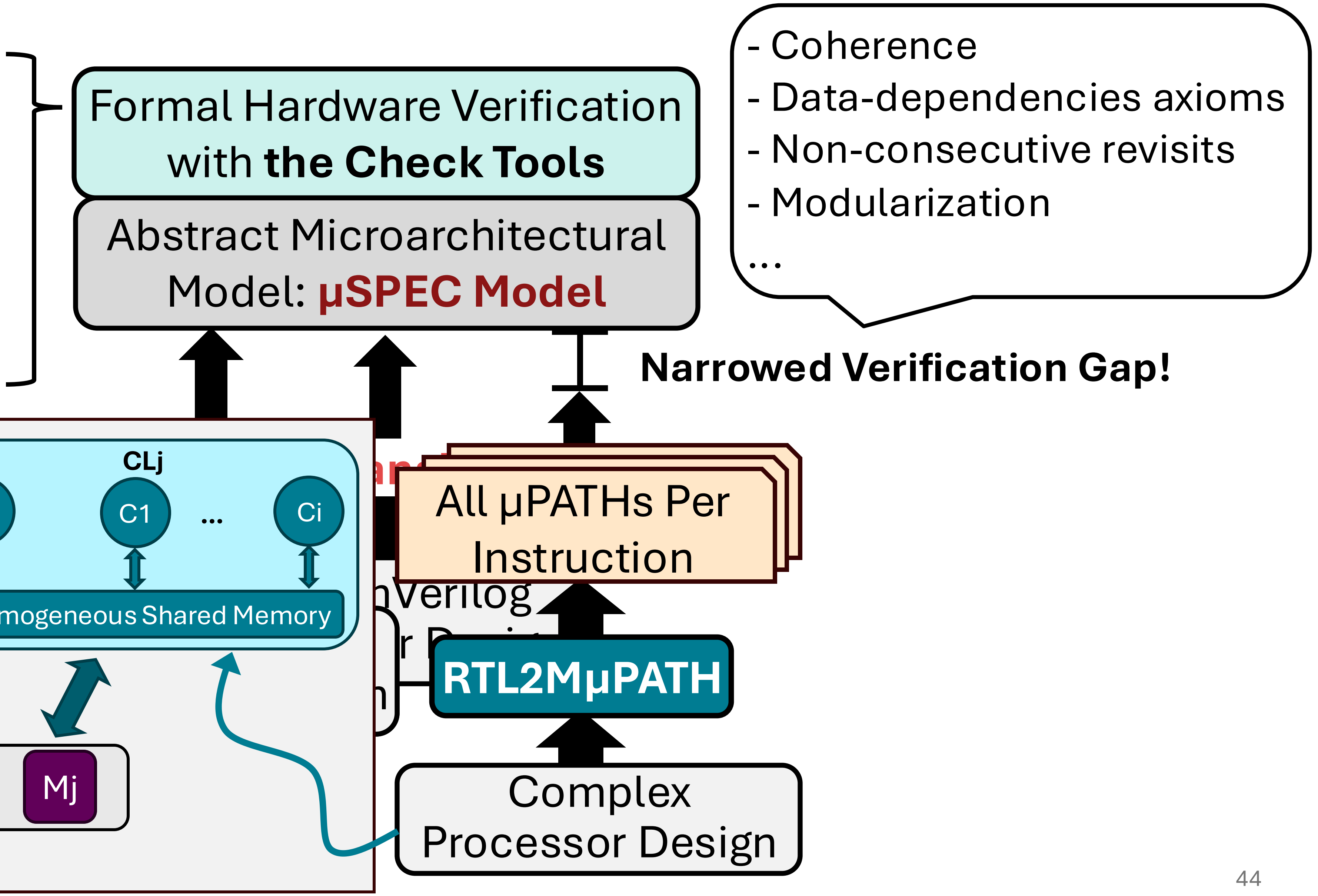
RISC-V multi-V-scale



Single-data-source assumption: Cannot recognize when data can be **sourced from more than one place** (e.g, cache, store buffer, or more)→ Incompleteness in µSPEC axioms synthesis

Takeaways

PipeCheck [MICRO'14]
CCICheck [MICRO'15]
COATCheck [ASPLOS'16]
TriCheck [ASPLOS'17]
RTLCheck [MICRO'17]
PipeProof [MICRO'18]
CheckMate [MICRO'18]



Formal Hardware Verification with **the Check Tools**
Abstract Microarchitectural Model: **μSPEC Model**
- Coherence
- Data-dependencies axioms
- Non-consecutive revisits
- Modularization
...
Narrowed Verification Gap!
All μPATHs Per Instruction
RTL2MμPATH
Complex Processor Design
CL0
C0
C1
...
Ci
Homogeneous Shared Memory
CLj
C0
C1
...
Ci
Homogeneous Shared Memory
M0
M1
M2
...
Mj

Thank you!