PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications

Yatin A. Manerkar, Daniel Lustig*,

Margaret Martonosi, and Aarti Gupta

Princeton University

*NVIDIA

MICRO-51

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

Memory Consistency Models (MCMs)

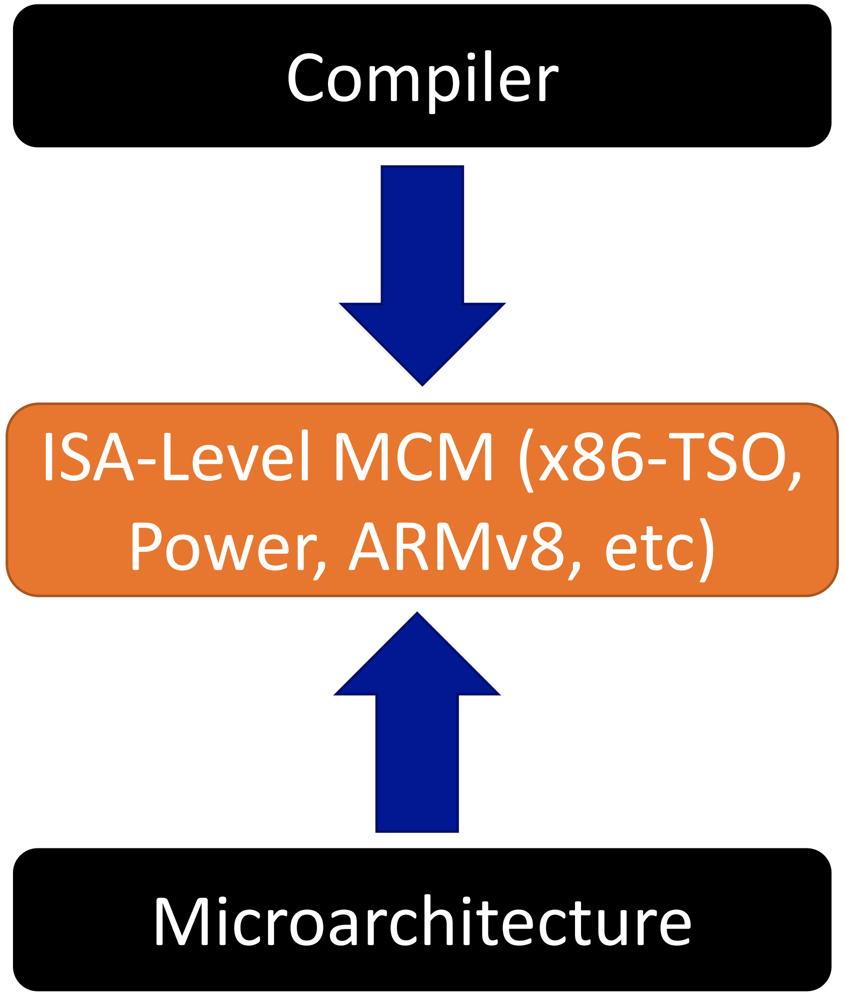
- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for **all possible programs**

Compiler

Microarchitecture

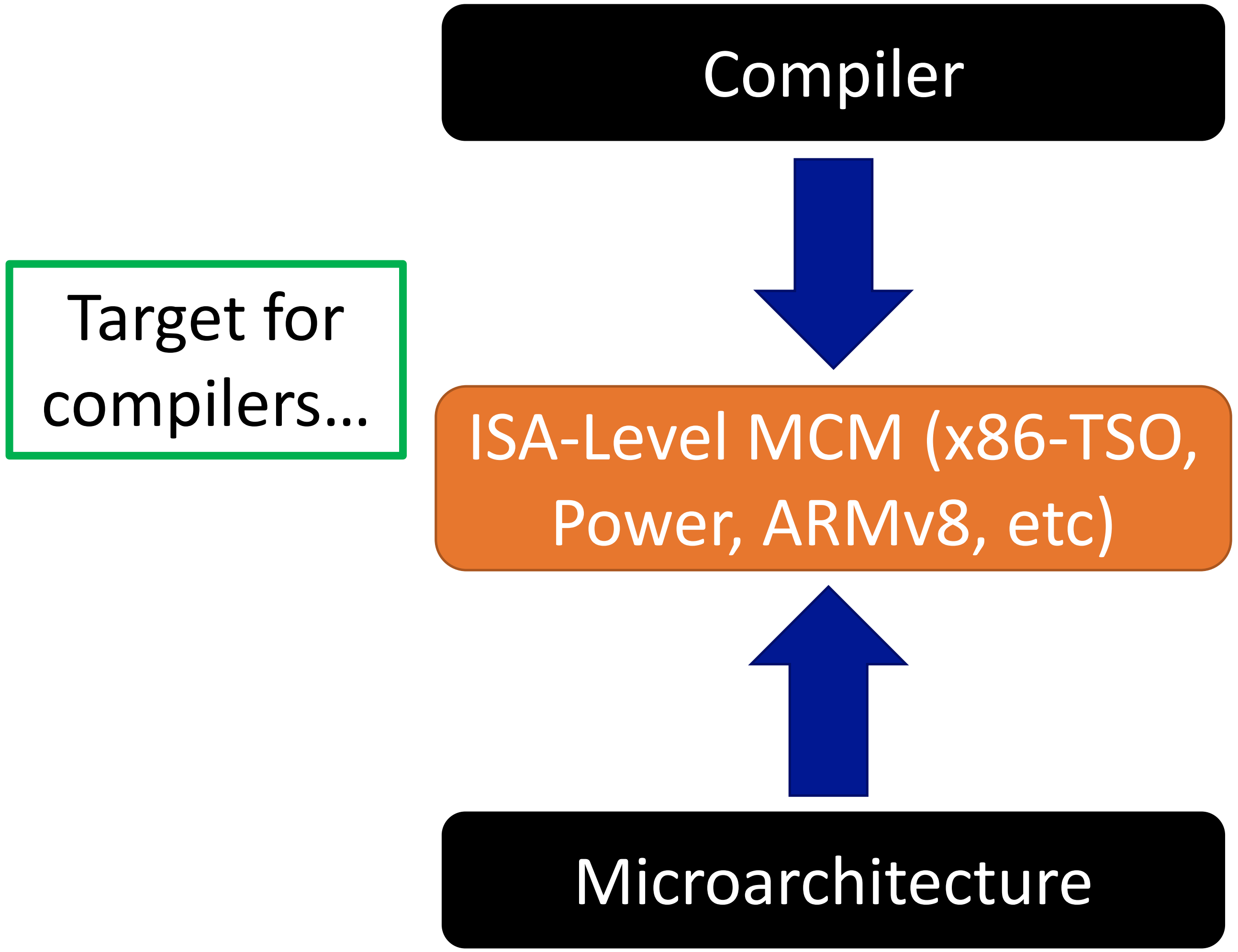
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for <u>**all possible programs**</u>



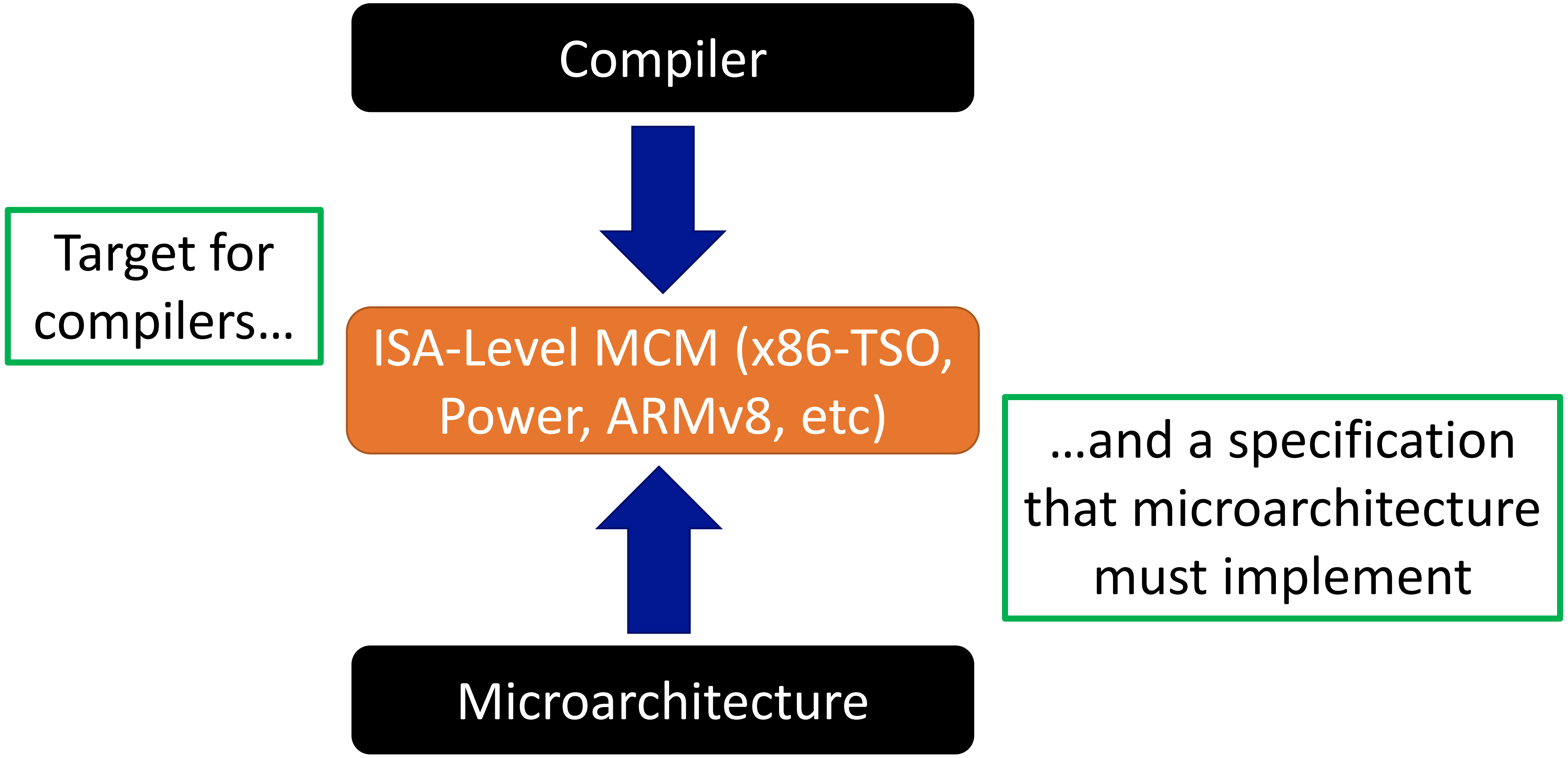
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for **all possible programs**



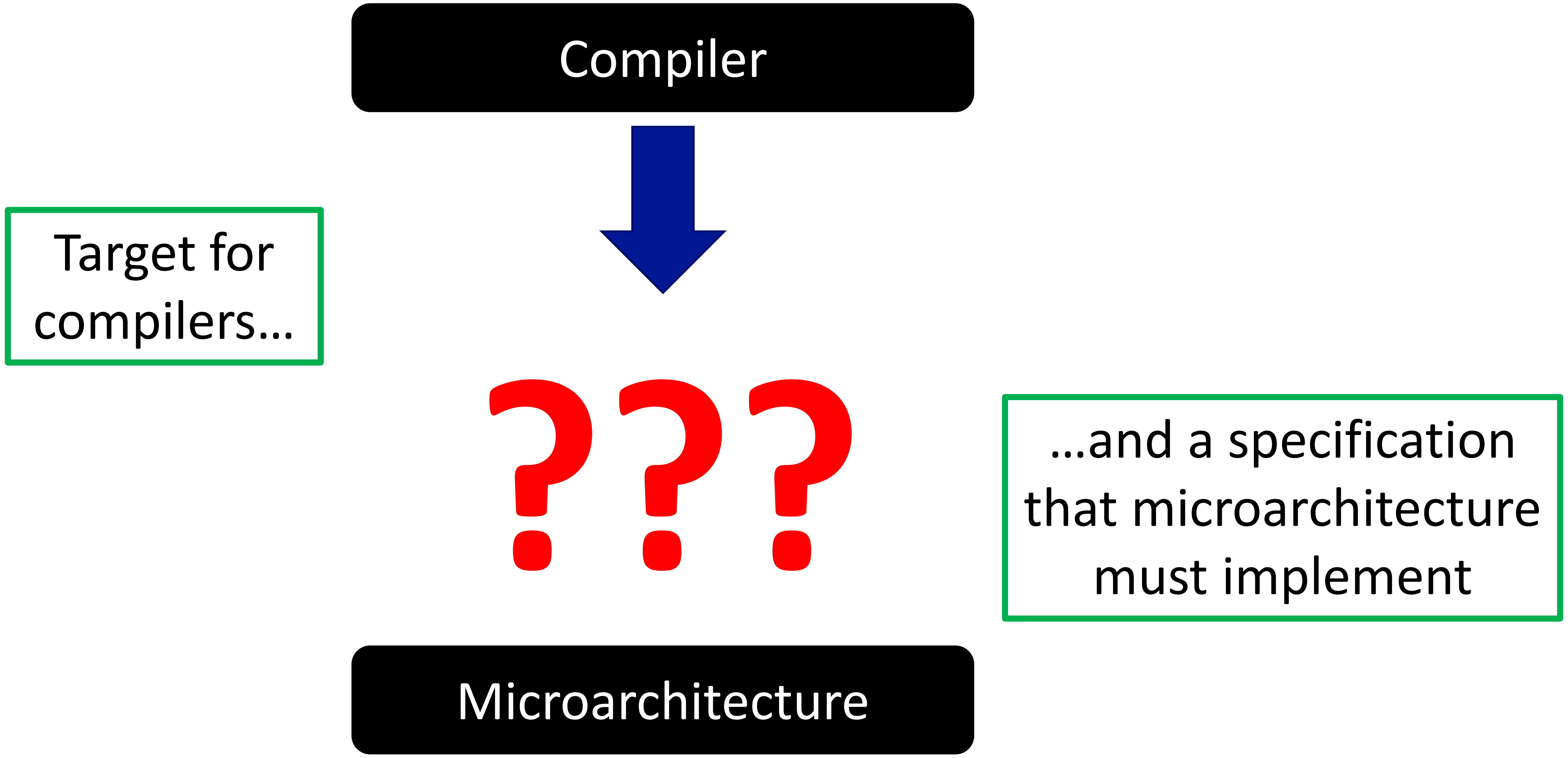
Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for **<u>all possible programs</u>**


Compiler
Target for compilers…
ISA-Level MCM (x86-TSO, Power, ARMv8, etc)
…and a specification that microarchitecture must implement
Microarchitecture

Memory Consistency Models (MCMs)

- Specify rules governing values returned by loads in parallel programs
- MCM must be correctly implemented for **<u>all possible programs</u>**

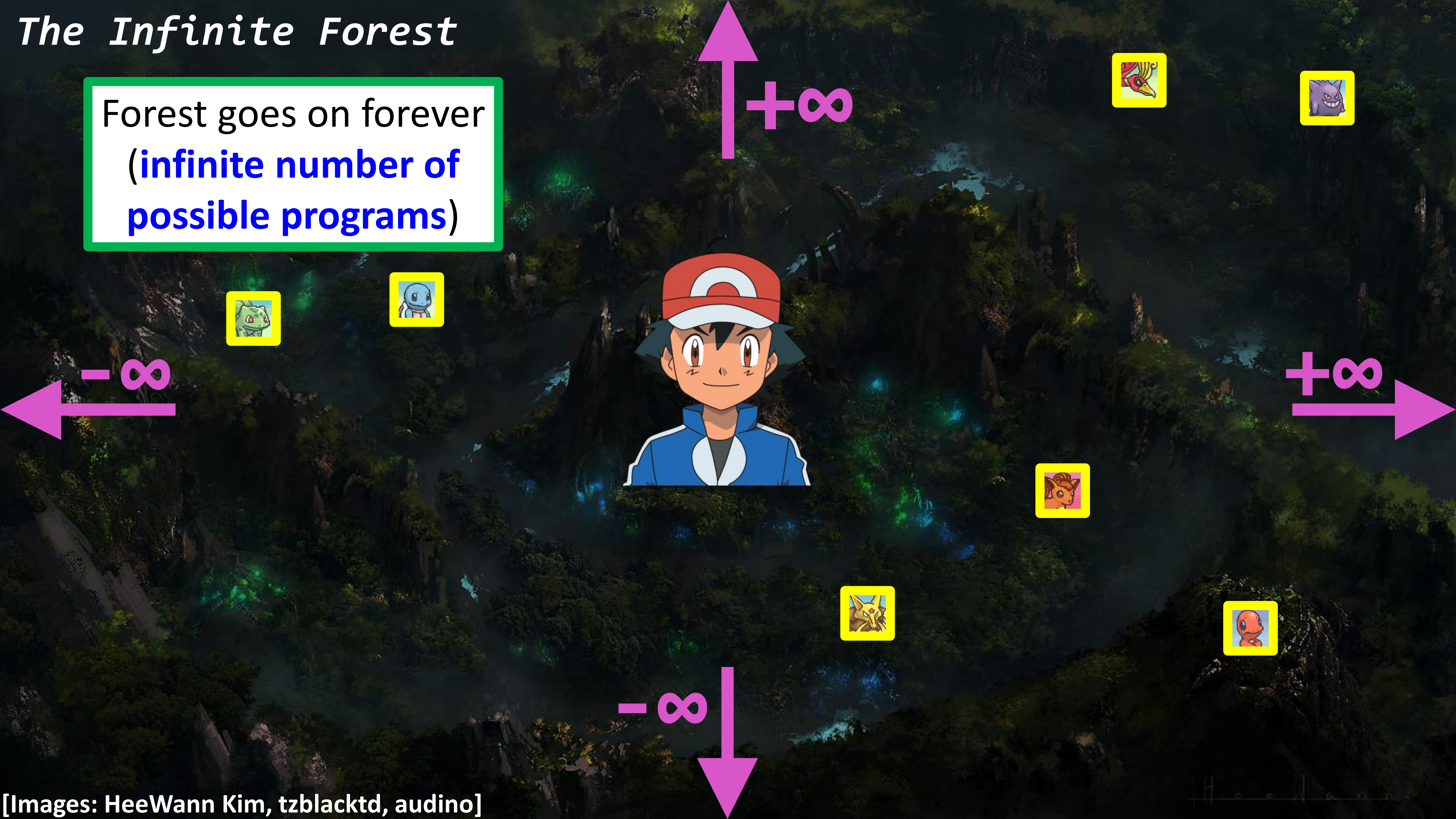


The Infinite Forest

[Images: HeeWann Kim, tzblacktd, audino]

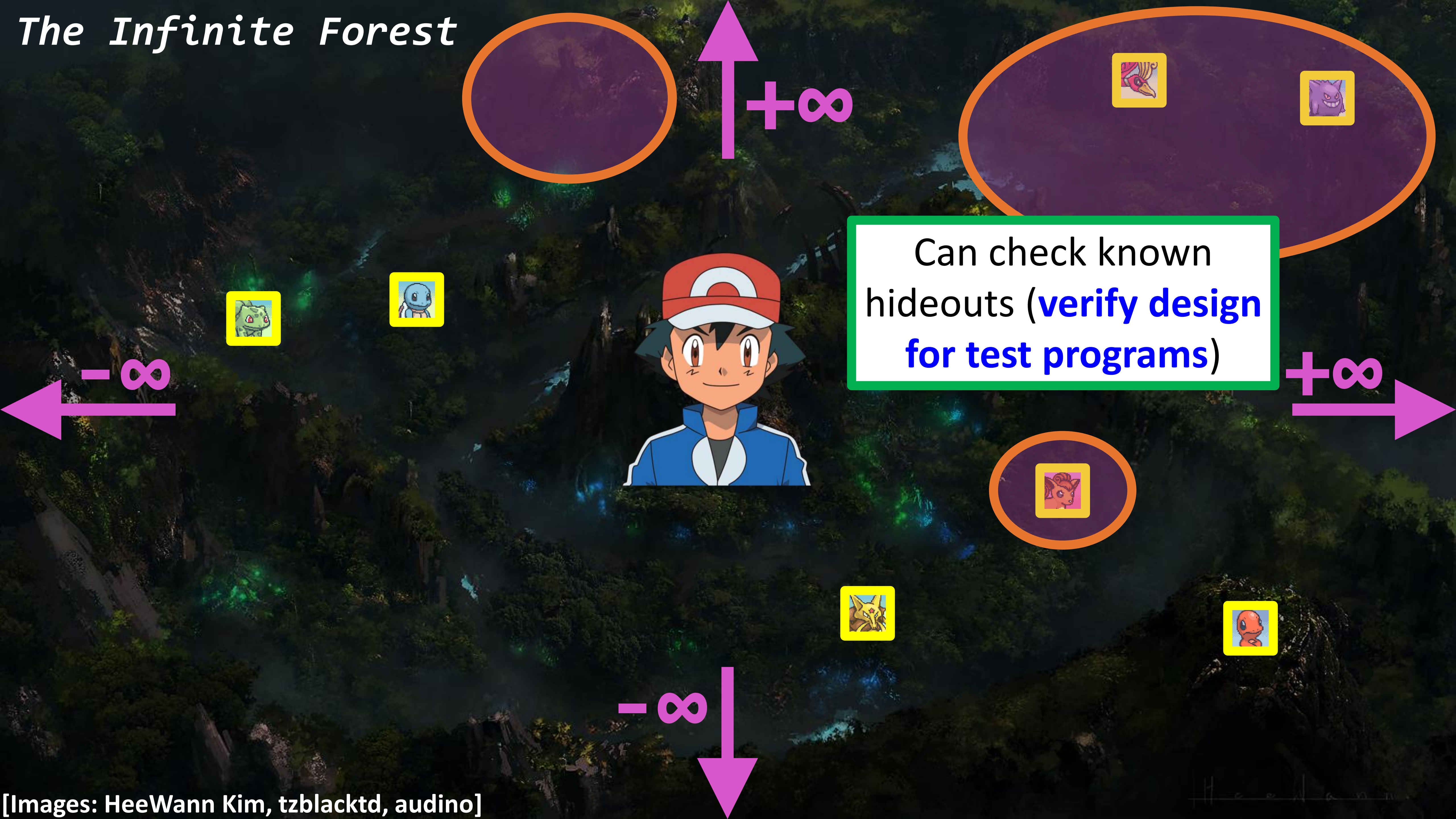
The Infinite Forest

Forest goes on forever (**infinite number of possible programs**)



[Images: HeeWann Kim, tzblacktd, audino]

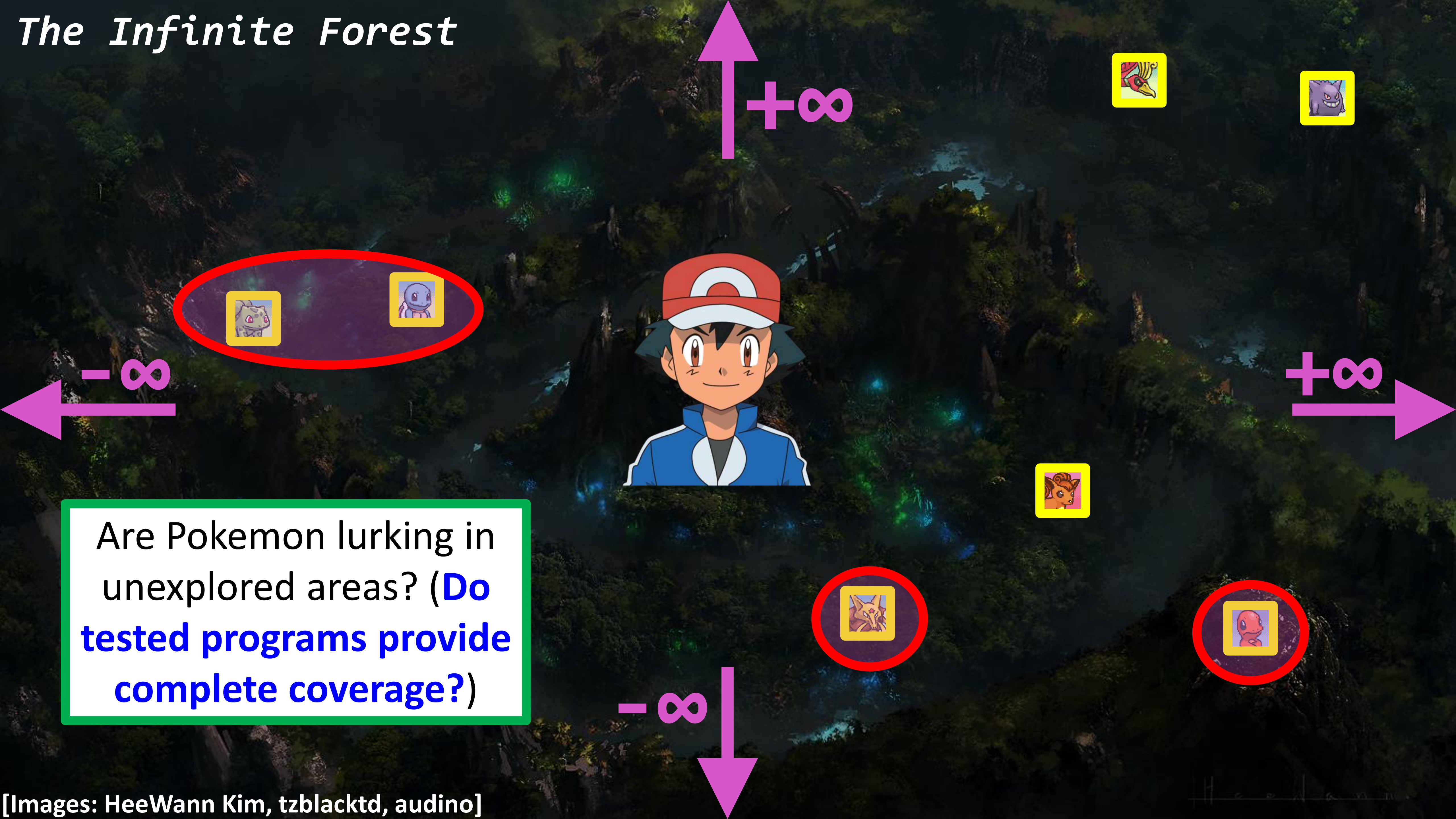
The Infinite Forest



Can check known hideouts (**verify design for test programs**)

[Images: HeeWann Kim, tzblacktd, audino]

The Infinite Forest



Are Pokemon lurking in unexplored areas? (**Do tested programs provide complete coverage?**)

[Images: HeeWann Kim, tzblacktd, audino]

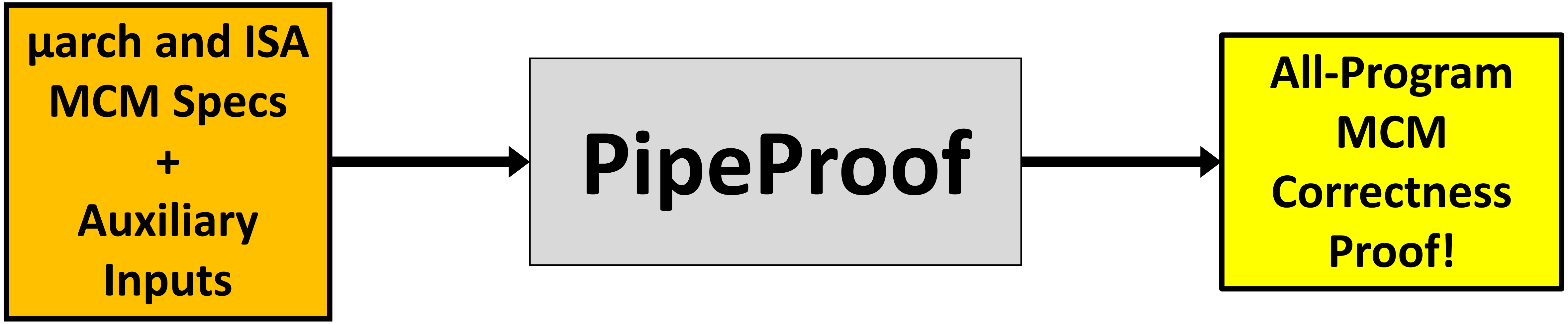
The Infinite Forest

Have we caught all the Pokemon?
(**Are there any MCM bugs left in the design?**)

[Images: HeeWann Kim, tzblacktd, audino]

PipeProof Overview

- **First automated all-program microarchitectural MCM verification!**
  - Covers all possible addresses, values, numbers of cores
- Proof methodology based on automatic abstraction refinement
- Early-stage: Can be conducted before RTL is written!


µarch and ISA MCM Specs + Auxiliary Inputs
PipeProof
All-Program MCM Correctness Proof!

Outline

- Background
  - ISA-level MCM specs
  - Microarchitectural ordering specs
- Microarchitectural Correctness Proof
  - Transitive Chain (TC) Abstraction
- Overall PipeProof Operation
  - TC Abstraction Support Proof
  - Chain Invariants
- Results

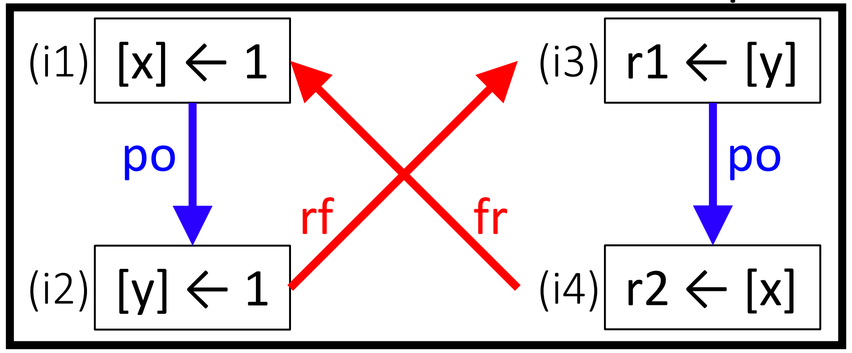
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes:** instructions, **edges:** ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - Eg: SC is $acyclic(po \cup co \cup rf \cup fr)$

Message passing (mp) litmus test

Core 0	Core 1
(i1) $[x] \leftarrow 1$ (i2) $[y] \leftarrow 1$	(i3) $r1 \leftarrow [y]$ (i4) $r2 \leftarrow [x]$
Under SC: Forbid r1=1, r2=0	

An ISA-level execution of mp



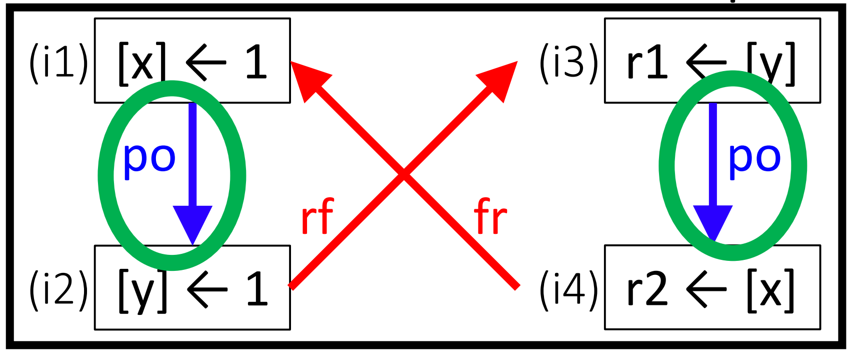
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes:** instructions, **edges:** ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - 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]
Under SC: Forbid r1=1, r2=0	

An ISA-level execution of mp



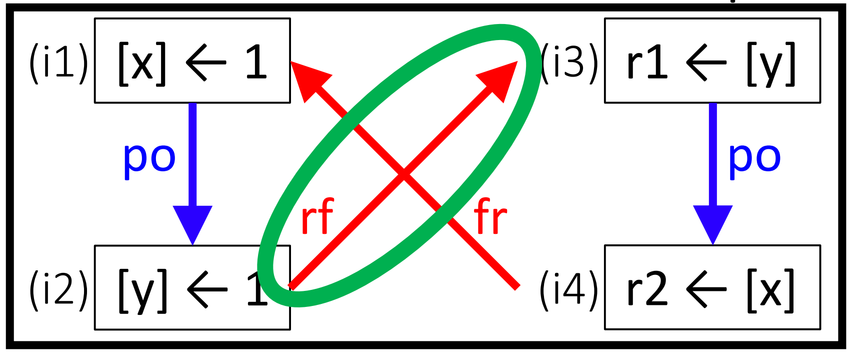
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes:** instructions, **edges:** ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - 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]
Under SC: Forbid r1=1, r2=0	

An ISA-level execution of mp



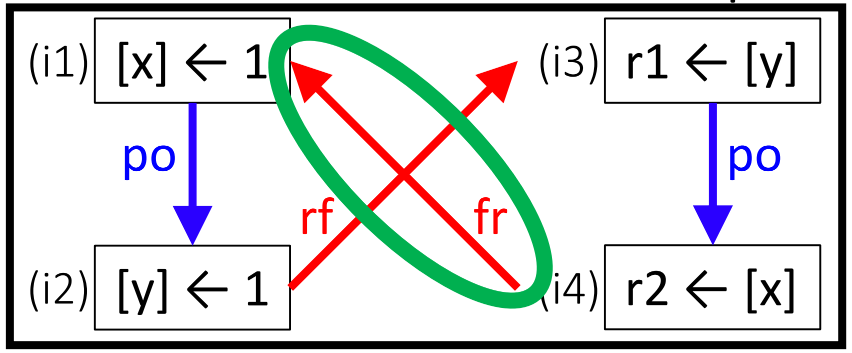
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes:** instructions, **edges:** ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - 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]
Under SC: Forbid r1=1, r2=0	

An ISA-level execution of mp



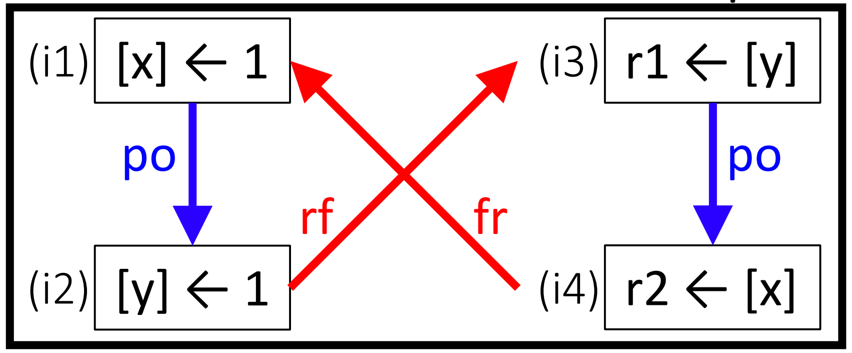
ISA-Level MCM Specifications

- Defined in terms of relational patterns [Alglave et al. TOPLAS 2014]
- ISA-level executions are graphs
  - **Nodes:** instructions, **edges:** ISA-level relations between instrs
- Correctness based on acyclicity, irreflexivity, etc of relational patterns
  - 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]
Under SC: Forbid r1=1, r2=0	

An ISA-level execution of mp



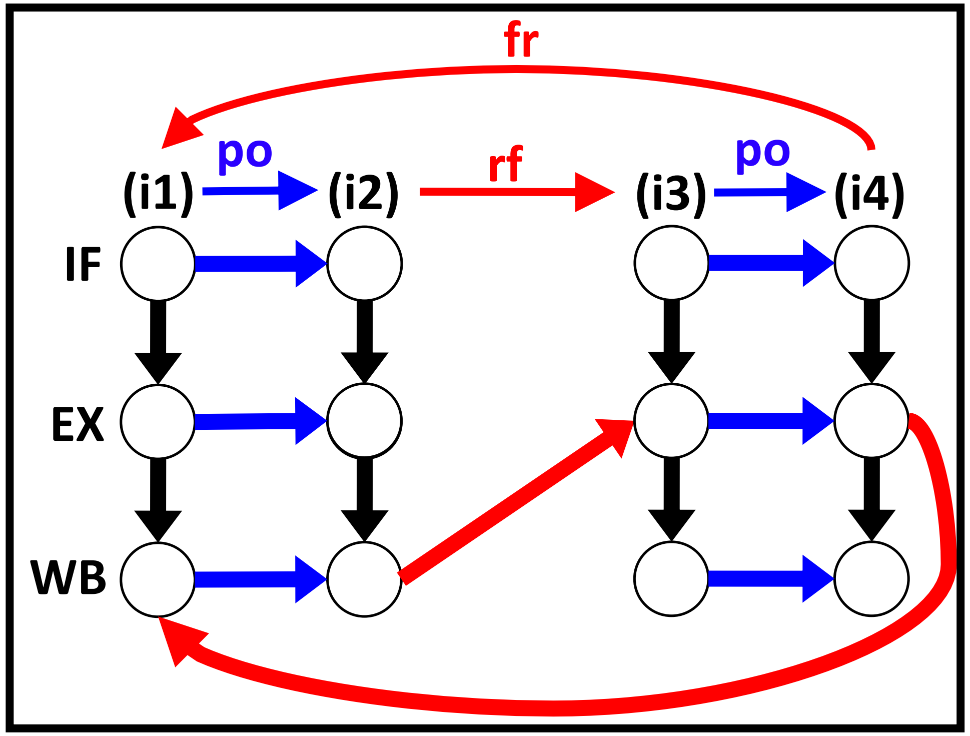
Microarchitectural Ordering Specifications

- Set of axioms in µspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as **µhb graphs**
  - **Nodes:** instr. sub-events, **edges:** happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

Message passing (mp) litmus test

Core 0	**Core 1**
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

A µhb graph of mp on simpleSC



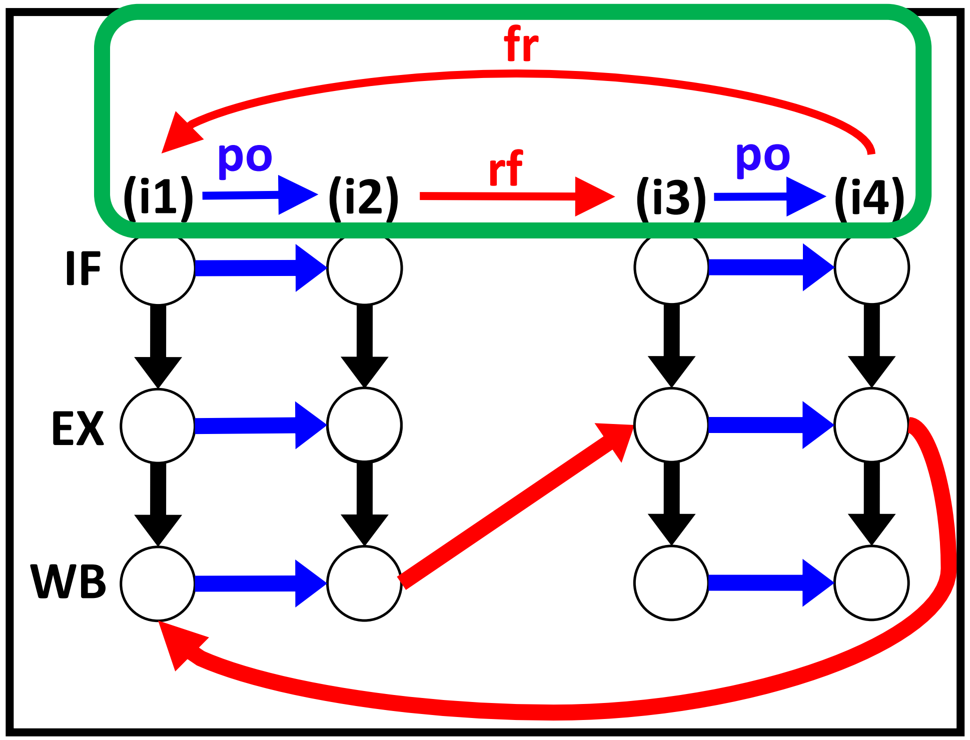
Microarchitectural Ordering Specifications

- Set of axioms in μspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as **μhb graphs**
  - **Nodes:** instr. sub-events, **edges:** happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

Message passing (mp) litmus test

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

A μhb graph of mp on simpleSC



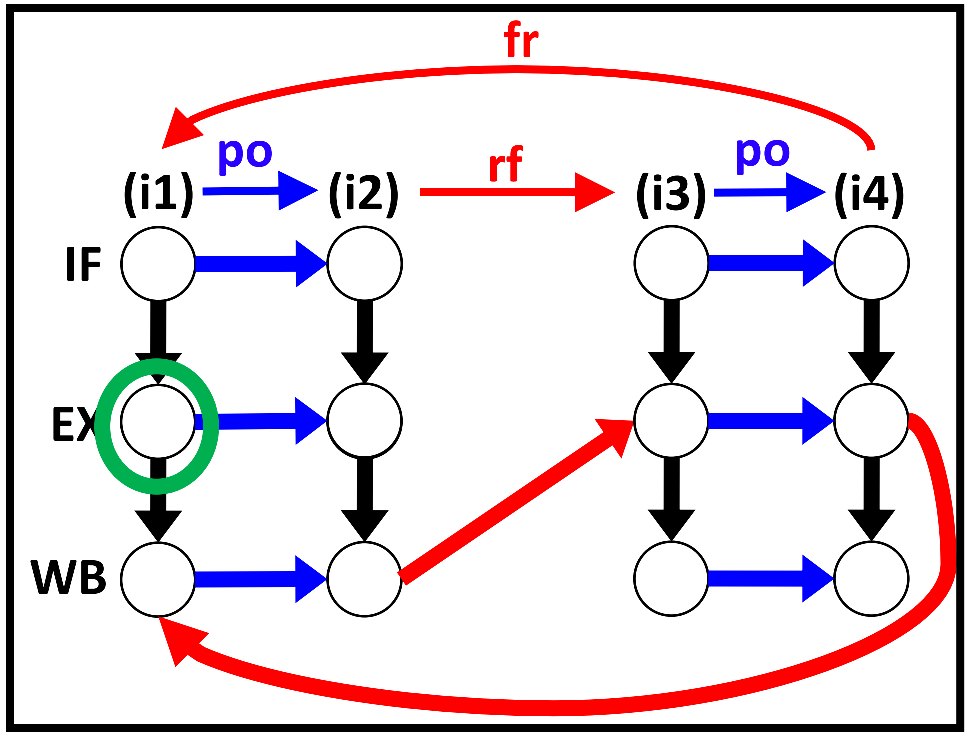
Microarchitectural Ordering Specifications

- Set of axioms in µspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as **µhb graphs**
  - **Nodes:** instr. sub-events, **edges:** happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

Message passing (mp) litmus test

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

A µhb graph of mp on simpleSC



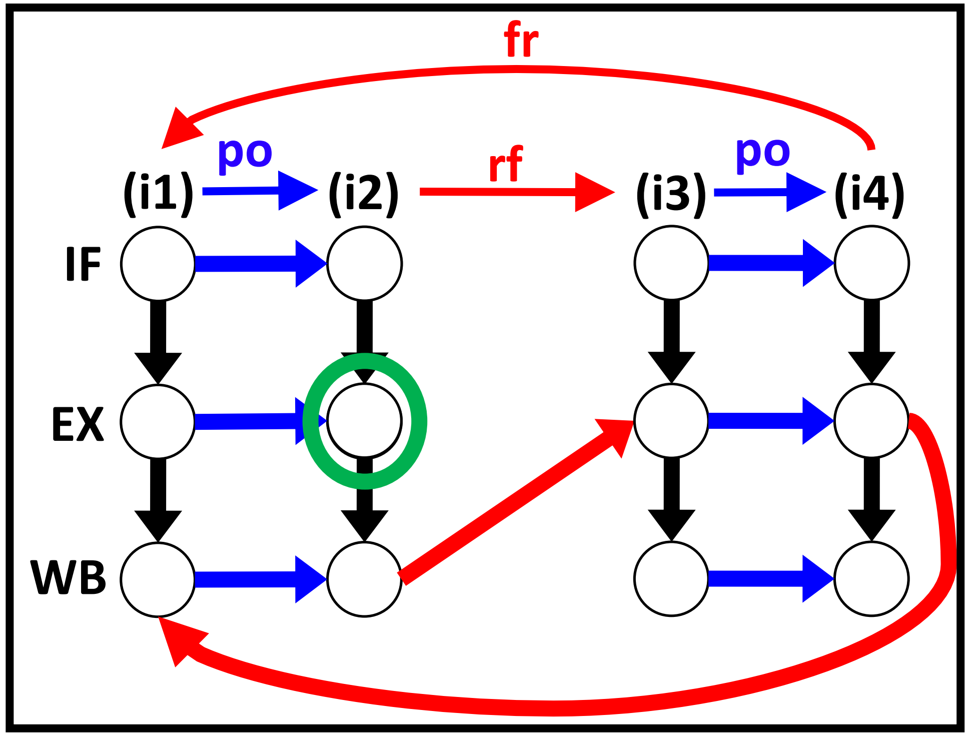
Microarchitectural Ordering Specifications

- Set of axioms in µspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as **µhb graphs**
  - **Nodes:** instr. sub-events, **edges:** happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

Message passing (mp) litmus test

Core 0	**Core 1**
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

A µhb graph of mp on simpleSC



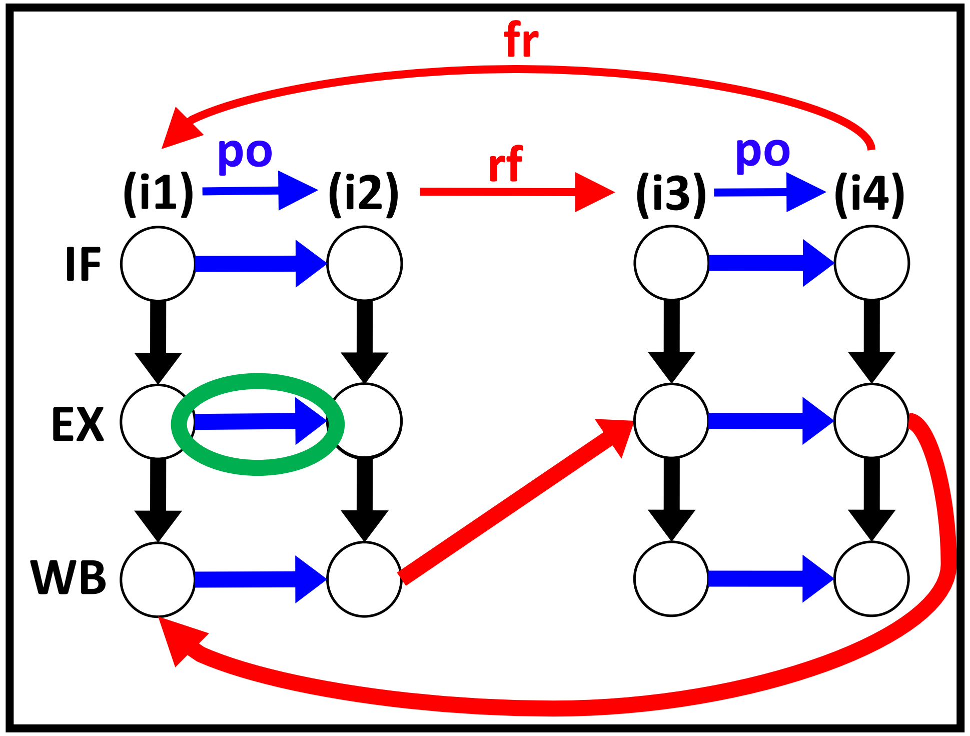
Microarchitectural Ordering Specifications

- Set of axioms in µspec DSL [Lustig et al. ASPLOS 2016]
- Used to generate microarchitectural executions as **µhb graphs**
  - **Nodes:** instr. sub-events, **edges:** happens-before relations between instrs
- Observability based on cyclicity of graphs
  - Cyclic graph → Unobservable
  - Acyclic graph → Observable

Message passing (mp) litmus test

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

A µhb graph of mp on simpleSC



Our Prior Work: Litmus Test-Based MCM Verification

Microarchitecture in μspec DSL

```
Axiom “Decode_is_FIFO":
... EdgeExists ((i1, Decode), (i2, Decode))
      => AddEdge ((i1, Execute), (i2, Execute)).
...
Axiom "PO_Fetch":
... SameCore i1 i2 /\ ProgramOrder i1 i2 =>
      AddEdge ((i1, Fetch), (i2, Fetch)).
```

+

Litmus Test

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	

Our Prior Work: Litmus Test-Based MCM Verification

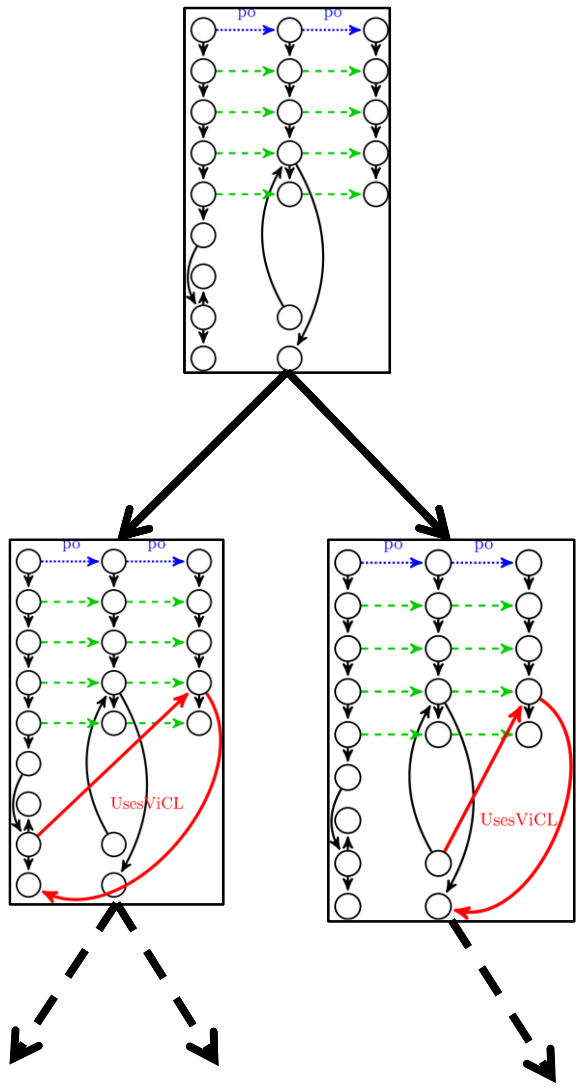
Microarchitecture in μspec DSL

```
Axiom “Decode_is_FIFO":
... EdgeExists ((i1, Decode), (i2, Decode))
     => AddEdge ((i1, Execute), (i2, Execute)).
...
Axiom "PO_Fetch":
... SameCore i1 i2 /\ ProgramOrder i1 i2 =>
     AddEdge ((i1, Fetch), (i2, Fetch)).
```

+

Litmus Test

Core 0	Core 1
(i1) [x] ← 1 (i2) [y] ← 1	(i3) r1 ← [y] (i4) r2 ← [x]
Under SC: Forbid r1=1, r2=0	



Microarchitectural happens-before (μhb) graphs

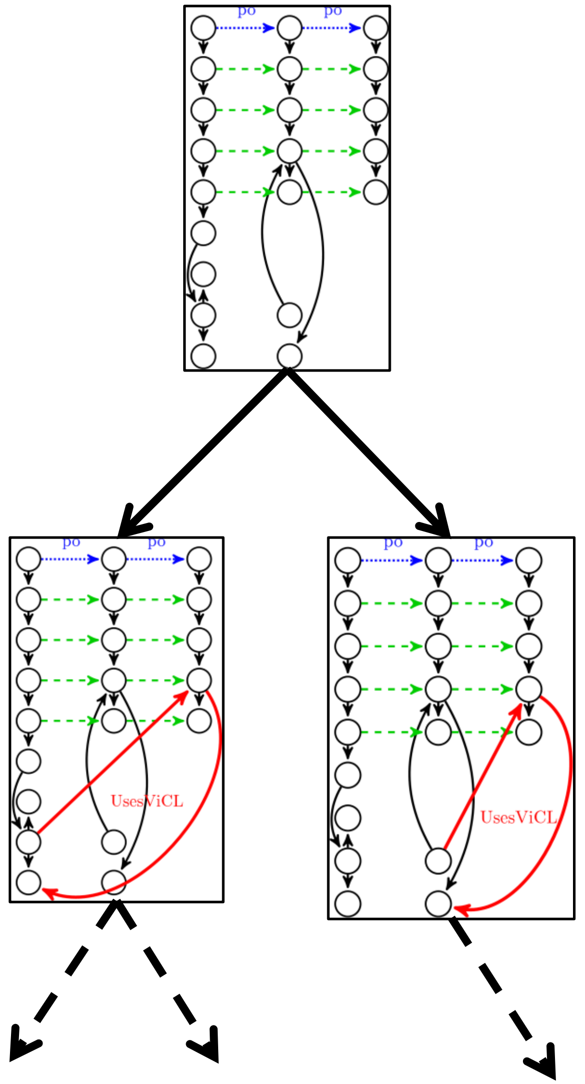
Our Prior Work: Litmus Test-Based MCM Verification

Microarchitecture in μspec DSL

```
Axiom “Decode_is_FIFO":
... EdgeExists ((i1, Decode), (i2, Decode))
       => AddEdge ((i1, Execute), (i2, Execute)).
...
Axiom "PO_Fetch":
... SameCore i1 i2 /\ ProgramOrder i1 i2 =>
       AddEdge ((i1, Fetch), (i2, Fetch)).
```

ISA-Level Outcome	Observable (≥ 1 Graph Acyclic)	Not Observable (All Graphs Cyclic)
Allowed	OK	OK (stricter than necessary)
Forbidden	Consistency violation!	OK

Under SC: Forbid r1=1, r2=0



Microarchitectural happens-before (μhb) graphs

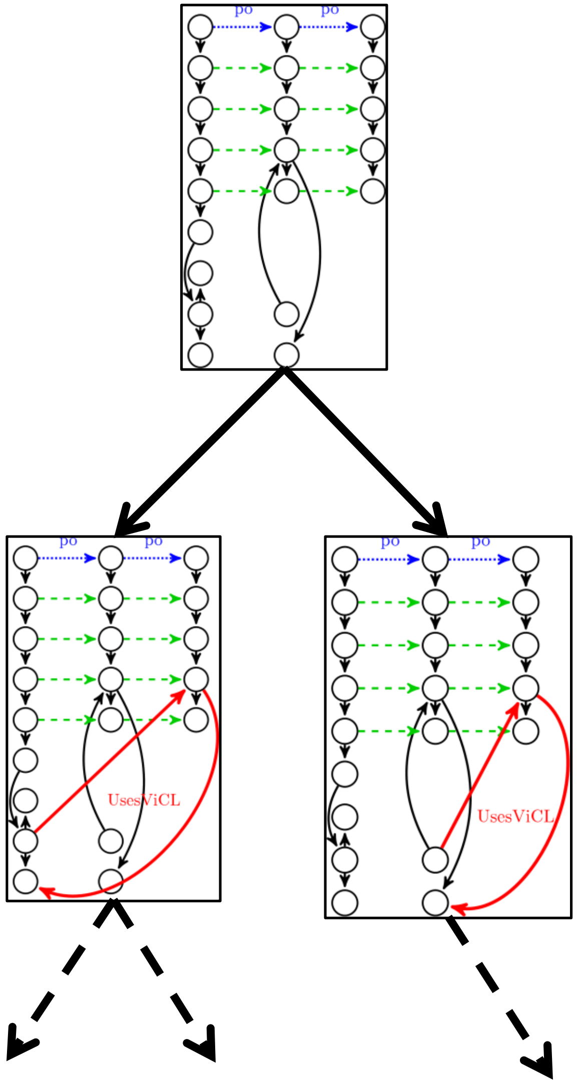
Our Prior Work: Litmus Test-Based MCM Verification

Microarchitecture in μspec DSL

```
Axiom “Decode_is_FIFO":
... EdgeExists ((i1, Decode), (i2, Decode))
     => AddEdge ((i1, Execute), (i2, Execute)).
...
Axiom "PO_Fetch":
... SameCore i1 i2 /\ ProgramOrder i1 i2 =>
     AddEdge ((i1, Fetch), (i2, Fetch)).
```

ISA-Level Outcome	Observable (≥ 1 Graph Acyclic)	Not Observable (All Graphs Cyclic)
Allowed	OK	OK (stricter than necessary)
Forbidden	Consistency violation!	OK

Under SC: Forbid r1=1, r2=0



Microarchitectural happens-before (μhb) graphs

[Lustig et al. MICRO-47, …]

Our Prior Work: Litmus Test-Based MCM Verification

Microarchitecture in μspec DSL

```
Axiom "Decode_is_FIFO":
... EdgeExists ((i1, Decode), (i2, Decode))
      => AddEdge ((i1, Execute), (i2, Execute)).
```

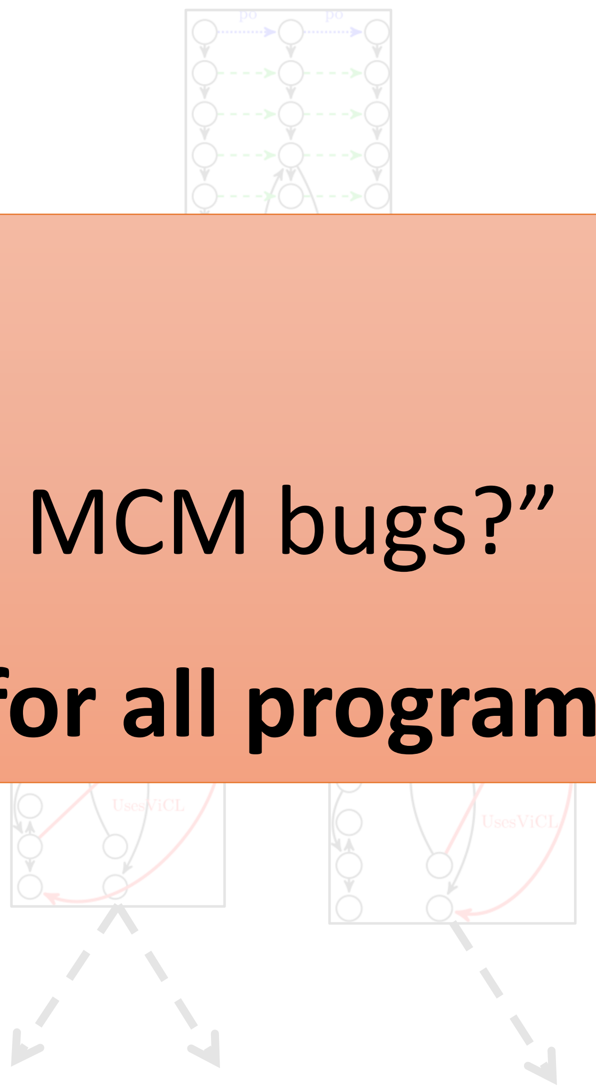
Perennial Question:

"Do your litmus tests cover all possible MCM bugs?"

How to automatically prove correctness for all programs?

Allowed	OK	than necessary)
Forbidden	Consistency violation!	OK

Under SC: Forbid r1=1, r2=0



Microarchitectural happens-before (μhb) graphs

[Lustig et al. MICRO-47, ...]

The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

$$\begin{tikzcd} i_1 \arrow[r, dashed, "r_{1\ldots n-1}"] & i_n \arrow[l, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "co"] & i_2 \arrow[r, "po"] & i_3 \arrow[ll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[l, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[r, "rf"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

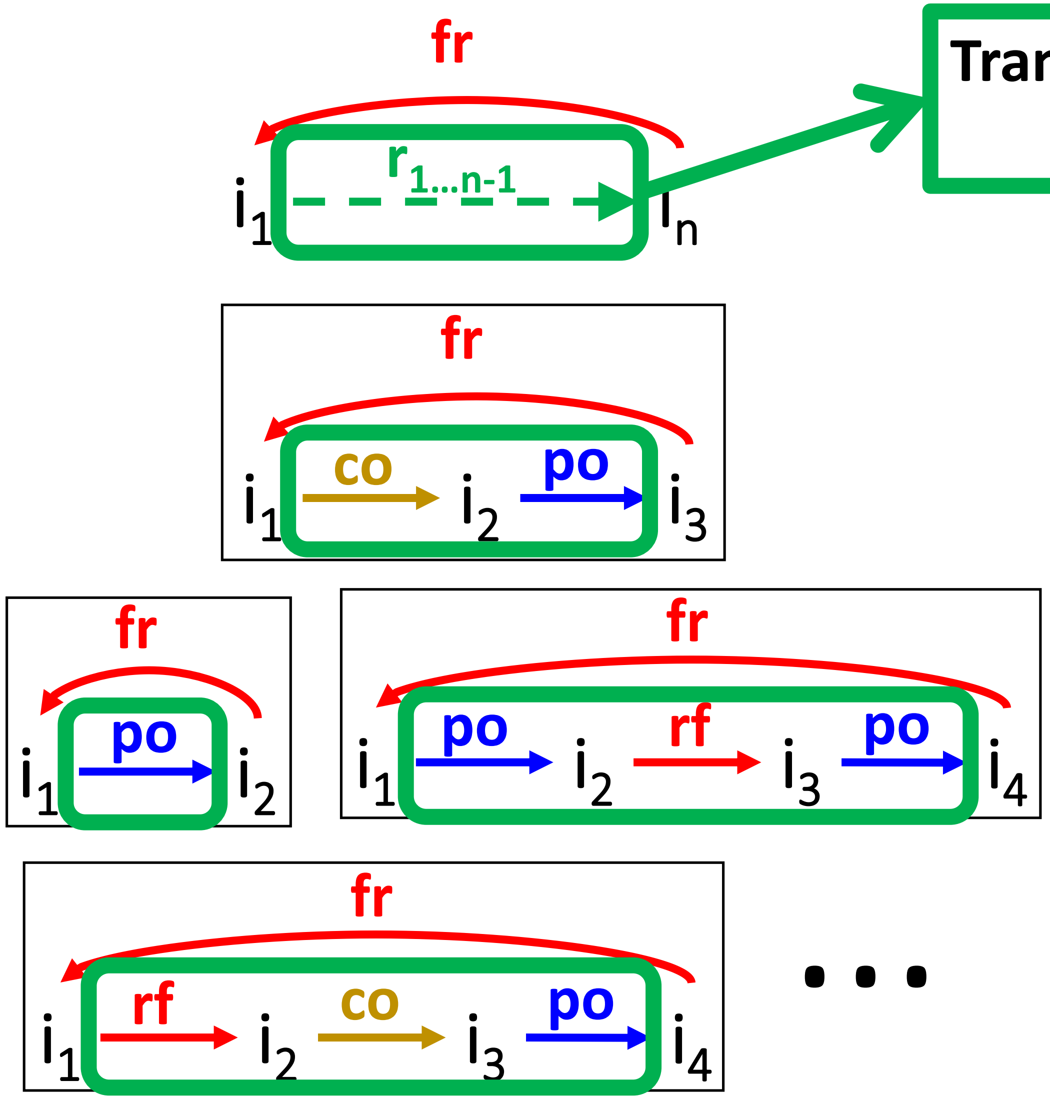
$$\begin{tikzcd} i_1 \arrow[r, "rf"] & i_2 \arrow[r, "co"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

...

The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

Transitive chain (sequence) of ISA-level edges



The Transitive Chain (TC) Abstraction

All non-unary cycles containing fr

$$\begin{tikzcd} i_1 \arrow[r, dashed, "r_{1\ldots n-1}"] & i_n \arrow[l, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "co"] & i_2 \arrow[r, "po"] & i_3 \arrow[ll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[l, bend right, "fr"'] \end{tikzcd}$$

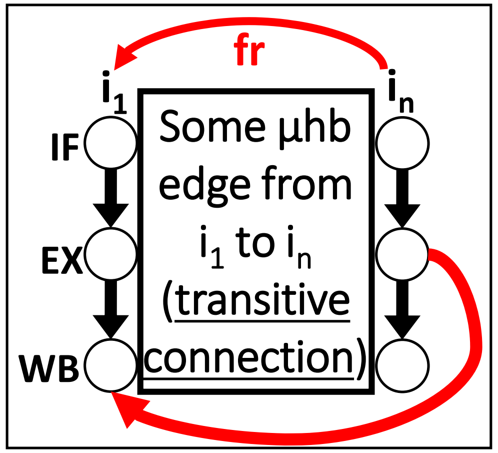
$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[r, "rf"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

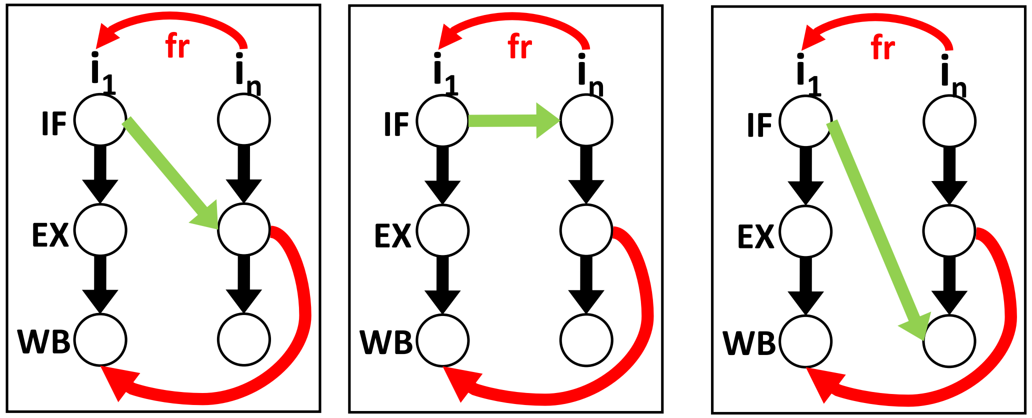
$$\begin{tikzcd} i_1 \arrow[r, "rf"] & i_2 \arrow[r, "co"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

. . .

Using TC Abstraction

$\Longrightarrow$

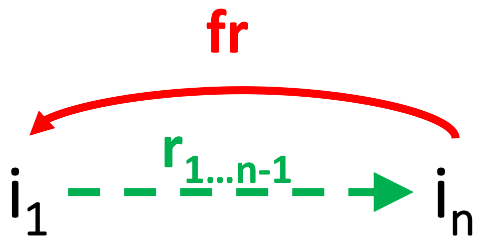


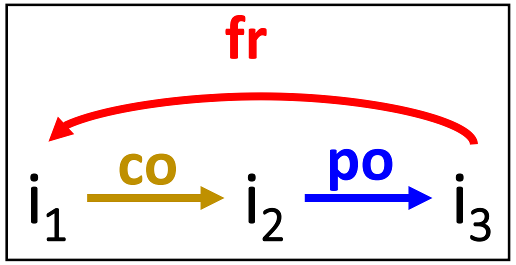


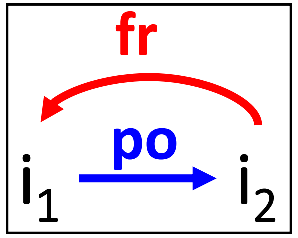
. . .

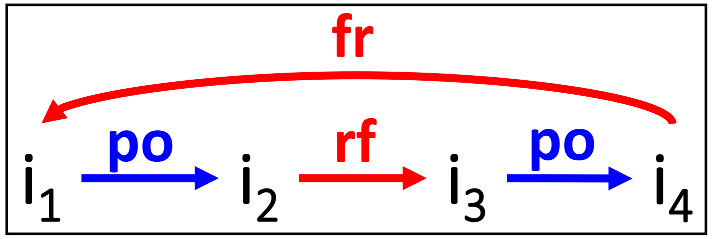
The Transitive Chain (TC) Abstraction

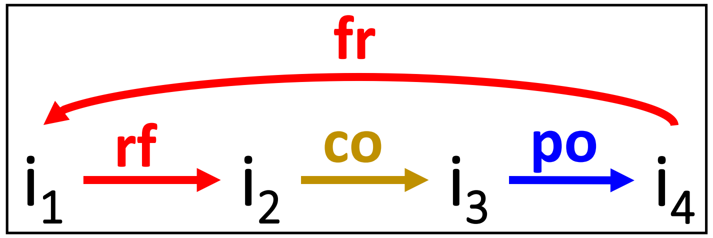
All non-unary cycles containing fr







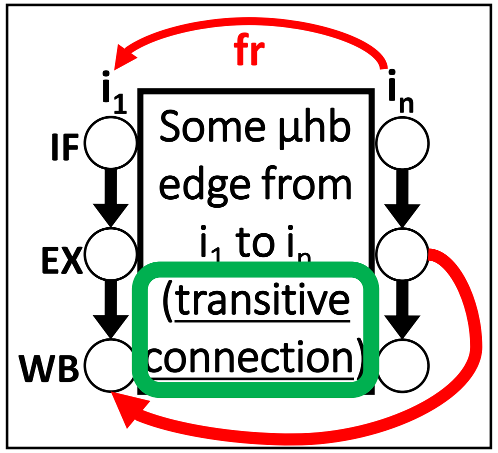


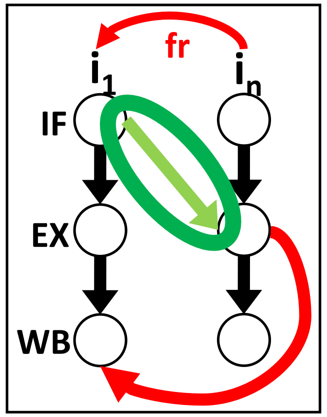


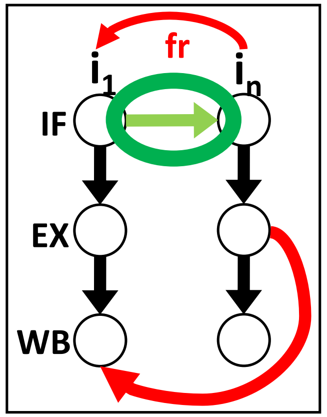
…

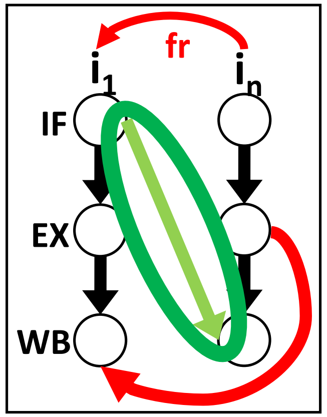
Using TC Abstraction

$\Longrightarrow$









…

The Transitive Chain (TC) Abstraction

Infinite

$$\begin{tikzcd} i_1 \arrow[r, "co"] & i_2 \arrow[r, "po"] & i_3 \arrow[ll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[r, "rf"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "rf"] & i_2 \arrow[r, "co"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[l, bend right, "fr"'] \end{tikzcd}$$

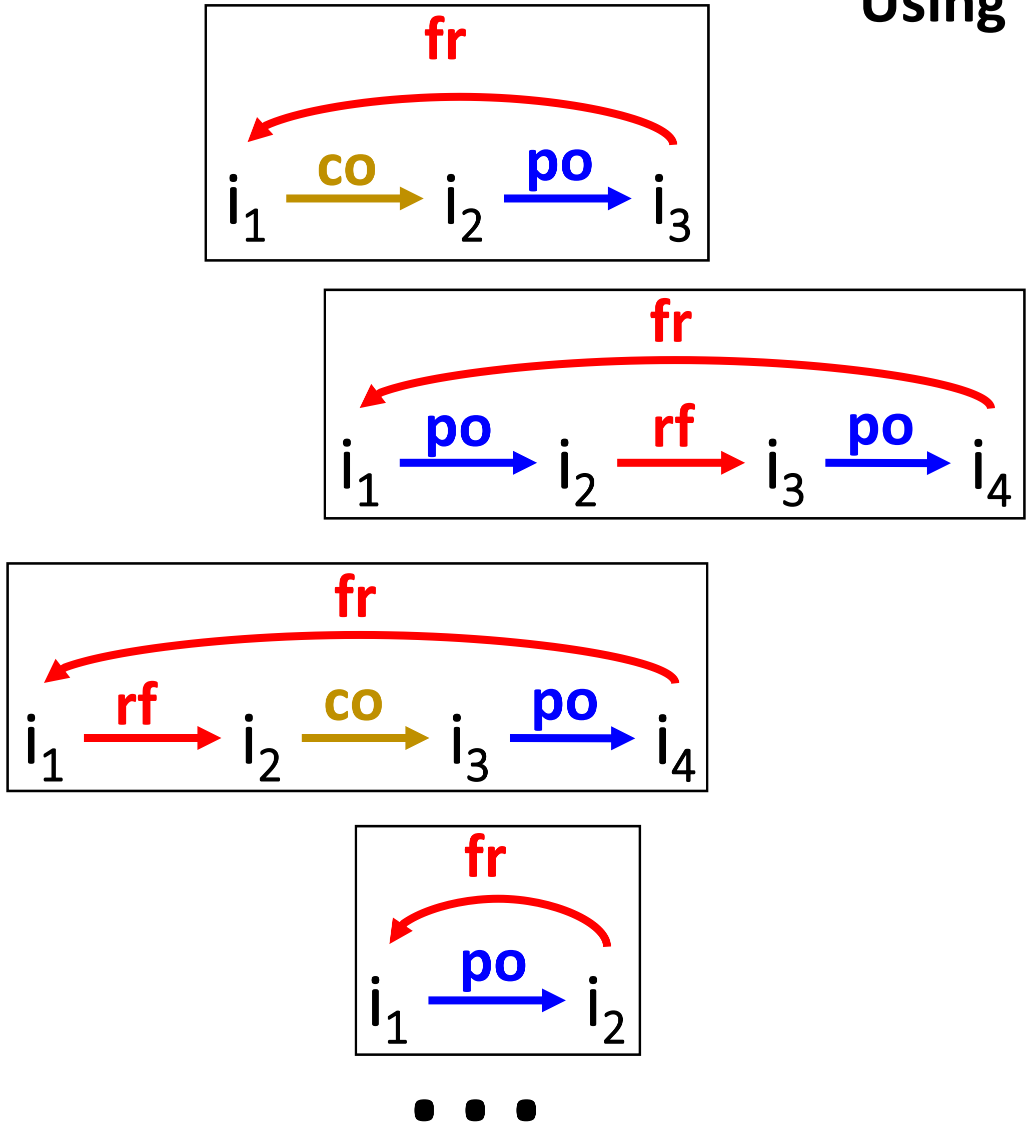
...

$\Longrightarrow$

Using TC Abstraction

The Transitive Chain (TC) Abstraction

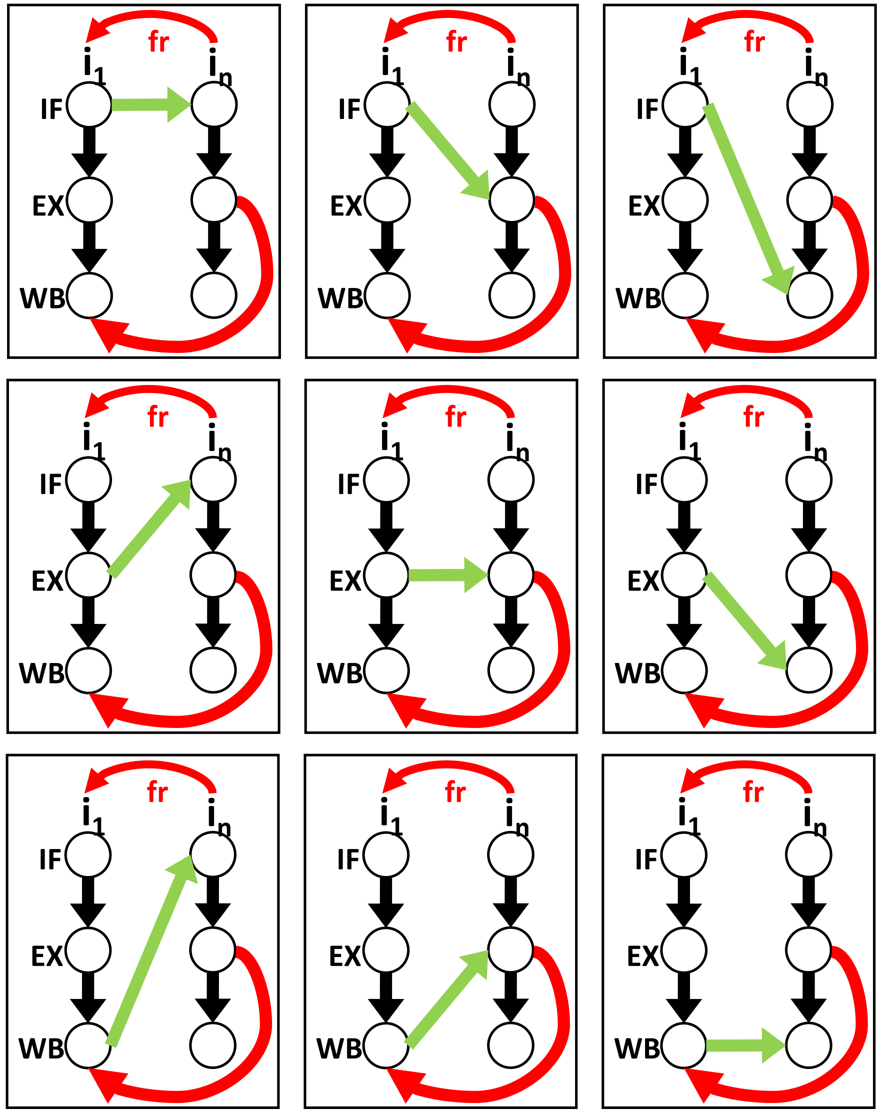
Infinite



$\Longrightarrow$

Using TC Abstraction

Finite!



The Transitive Chain (TC) Abstraction

Infinite

$$\begin{tikzcd} i_1 \arrow[r, "co"] & i_2 \arrow[r, "po"] & i_3 \arrow[ll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[r, "rf"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

$$\begin{tikzcd} i_1 \arrow[r, "rf"] & i_2 \arrow[r, "co"] & i_3 \arrow[r, "po"] & i_4 \arrow[lll, bend right, "fr"'] \end{tikzcd}$$

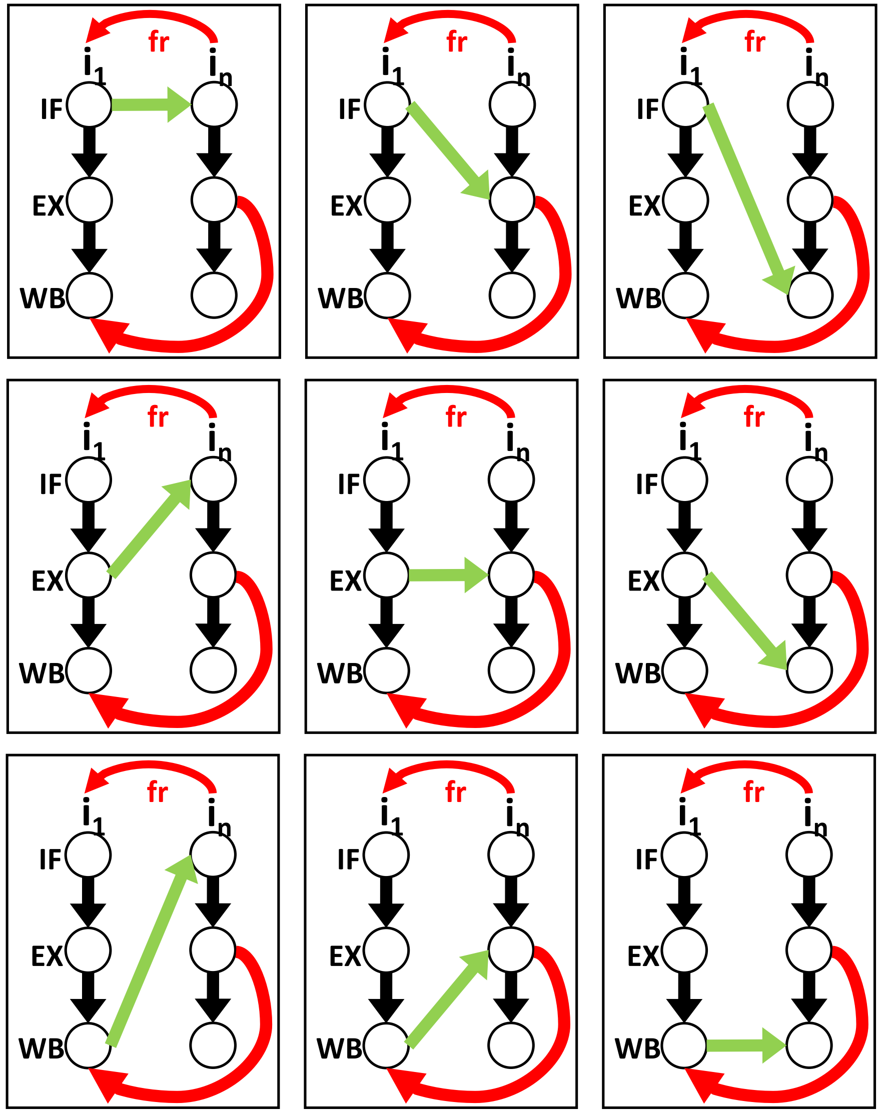
$$\begin{tikzcd} i_1 \arrow[r, "po"] & i_2 \arrow[l, bend right, "fr"'] \end{tikzcd}$$

...

$\Longrightarrow$

Using TC Abstraction

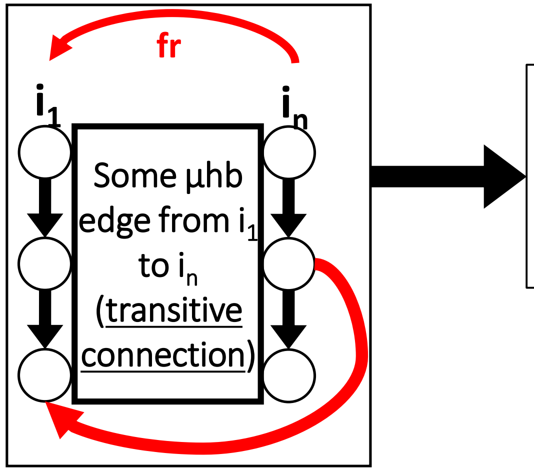
Finite!



Soundness verified as a supporting proof!

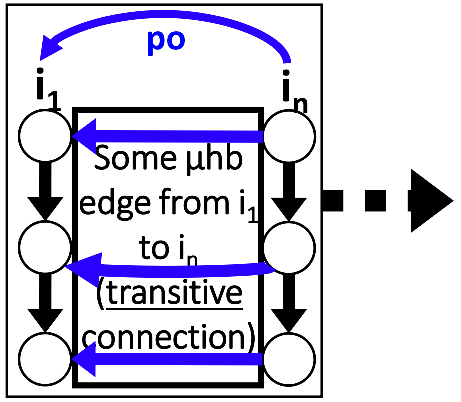
Microarchitectural Correctness Proof

Cycles containing *fr*



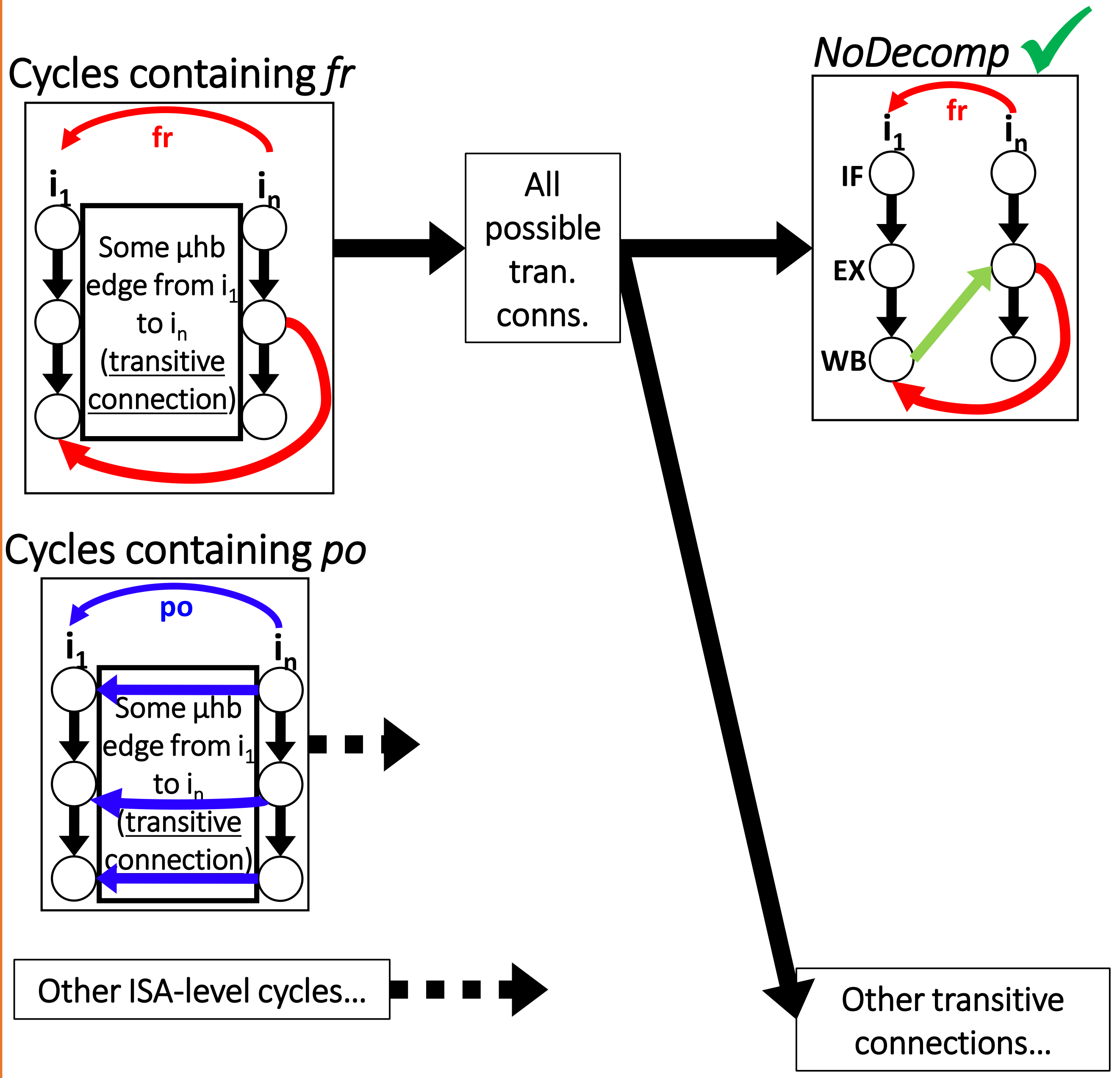
All possible tran. conns.

Cycles containing *po*

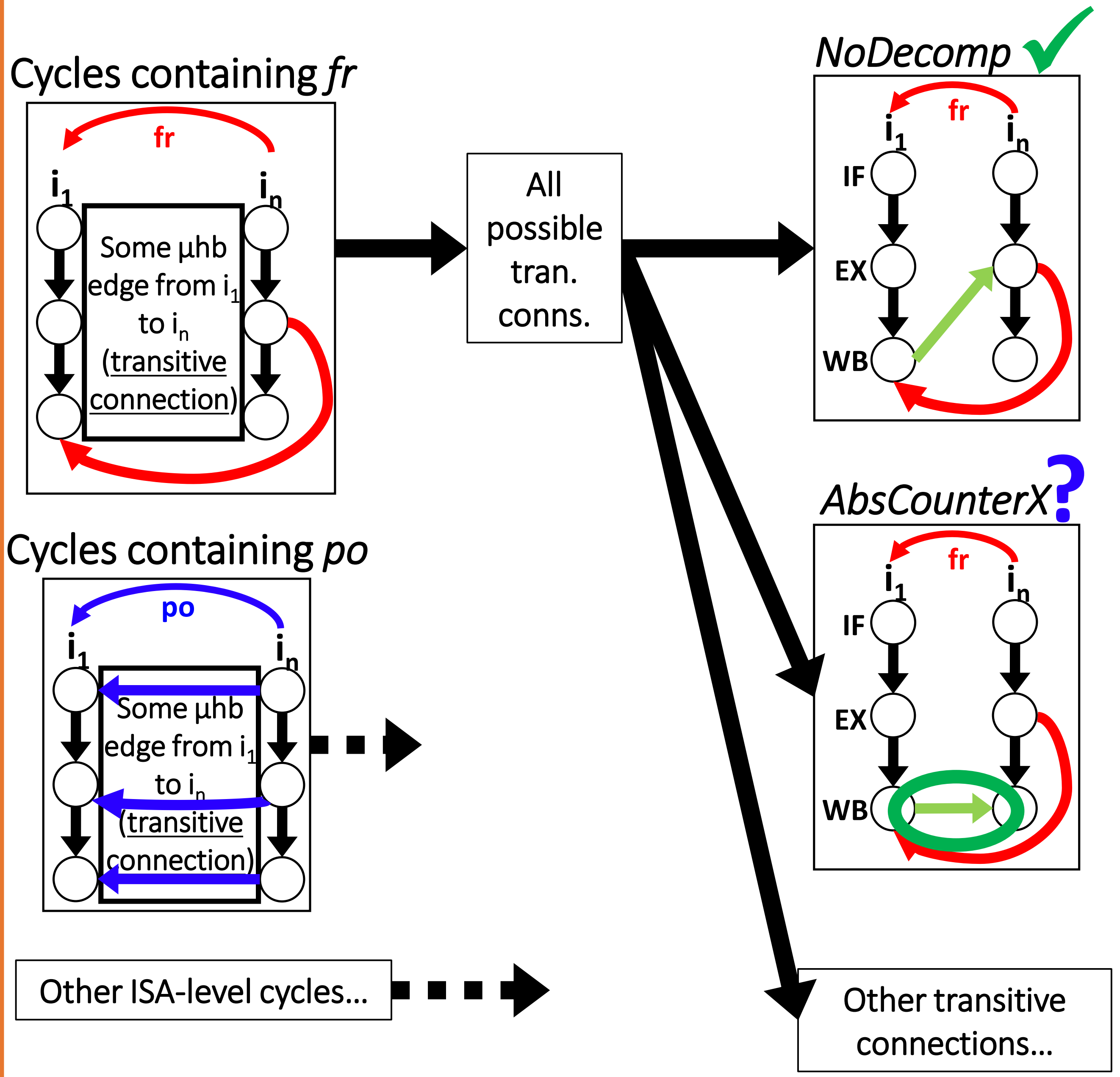


Other ISA-level cycles…

Microarchitectural Correctness Proof

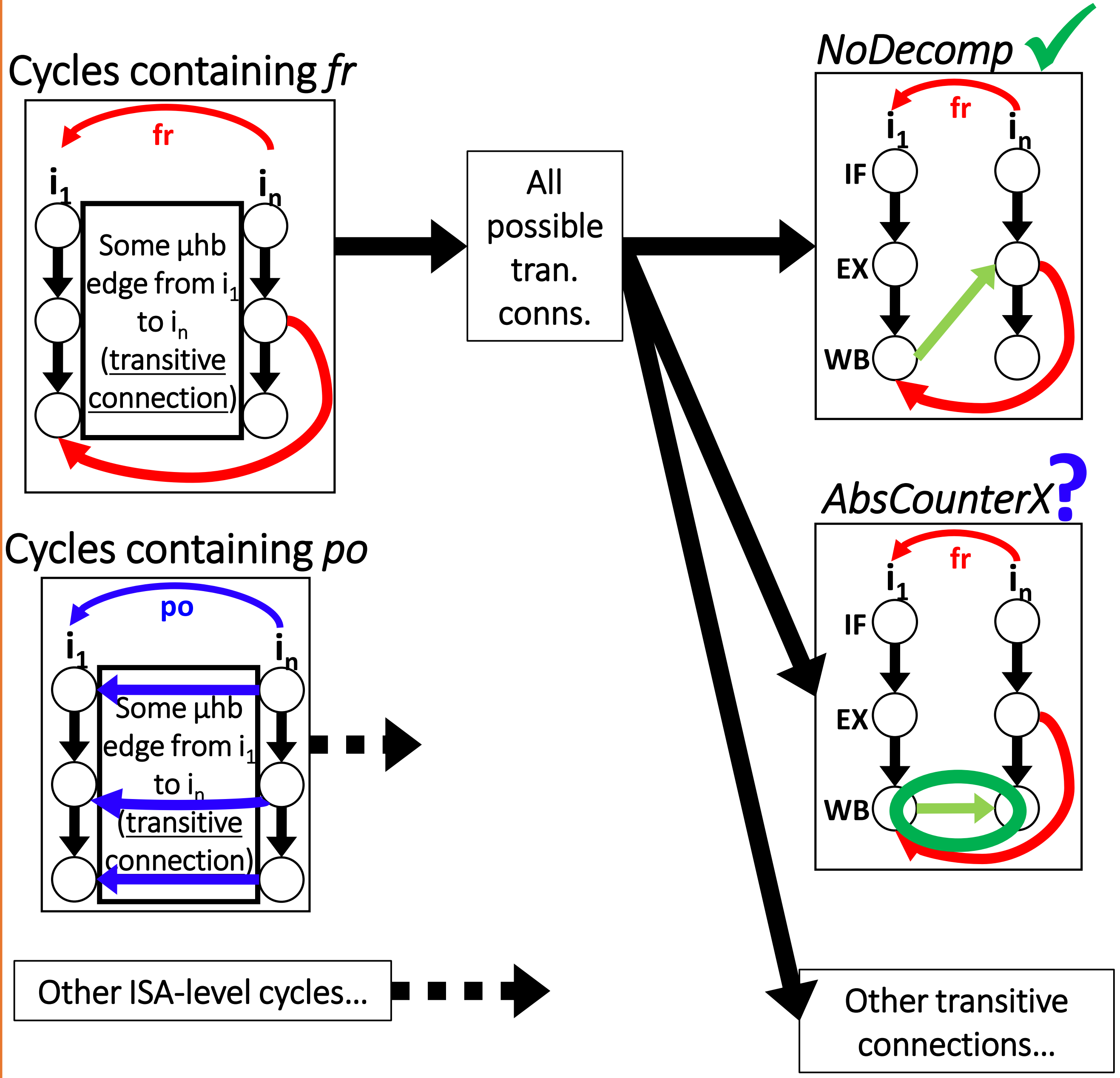

Cycles containing *fr*
fr
i1
in
Some µhb edge from i1 to in (transitive connection)
All possible tran. conns.
NoDecomp ✓
IF
EX
WB
Cycles containing *po*
po
Other ISA-level cycles…
Other transitive connections…

Microarchitectural Correctness Proof



Cycles containing *fr*
fr
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Cycles containing *po*
po
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Other ISA-level cycles...
All possible tran. conns.
NoDecomp ✓
fr
i_1
i_n
IF
EX
WB
AbsCounterX ?
fr
i_1
i_n
IF
EX
WB
Other transitive connections...

Microarchitectural Correctness Proof


Cycles containing *fr*
fr
i1
in
Some μhb edge from i1 to in (transitive connection)
Cycles containing *po*
po
i1
in
Some μhb edge from i1 to in (transitive connection)
Other ISA-level cycles...
All possible tran. conns.
NoDecomp
fr
i1
in
IF
EX
WB
AbsCounterX
fr
i1
in
IF
EX
WB
Other transitive connections...

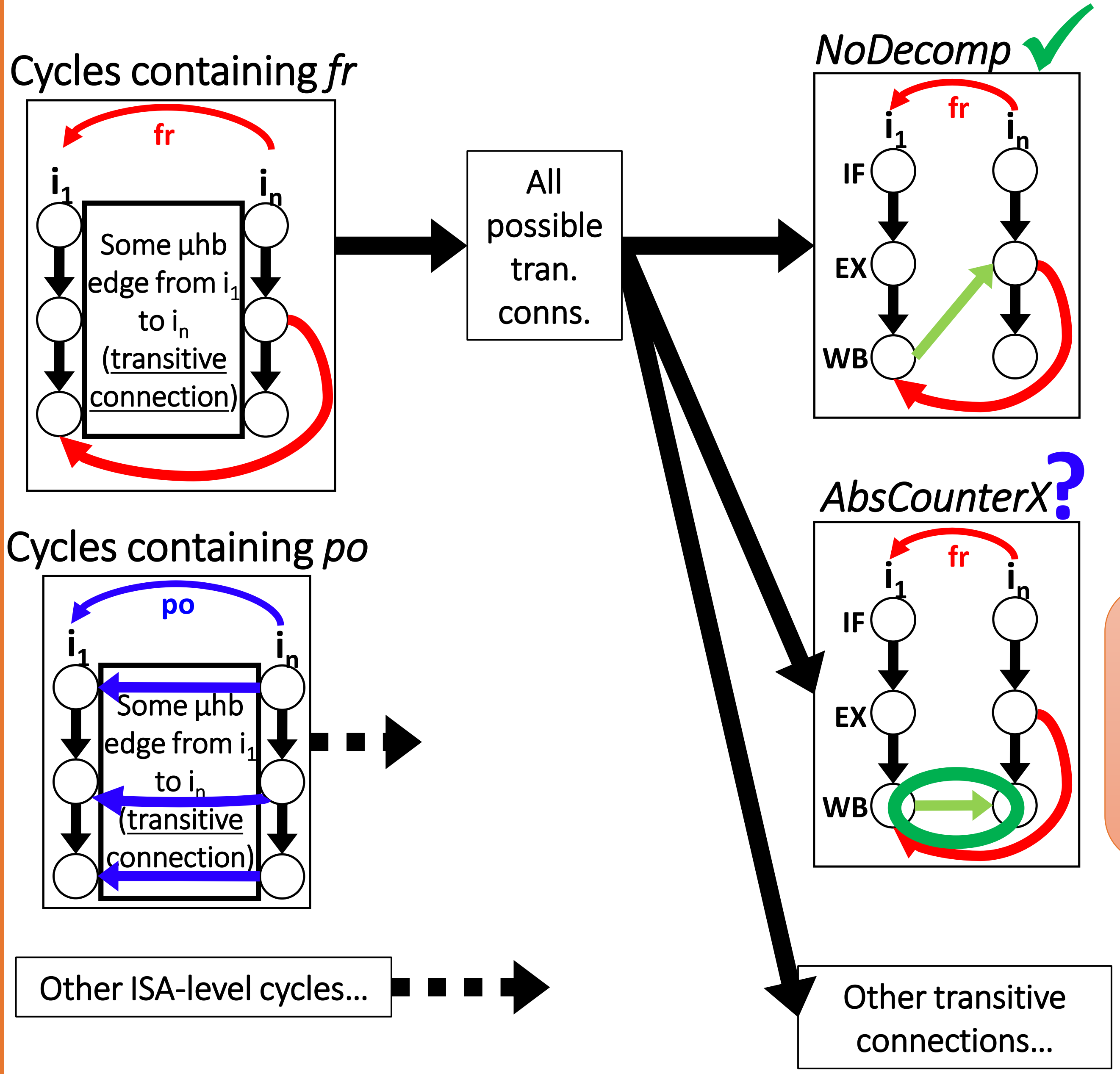
Acyclic graph with

transitive connection =>

Abstract Counterexample

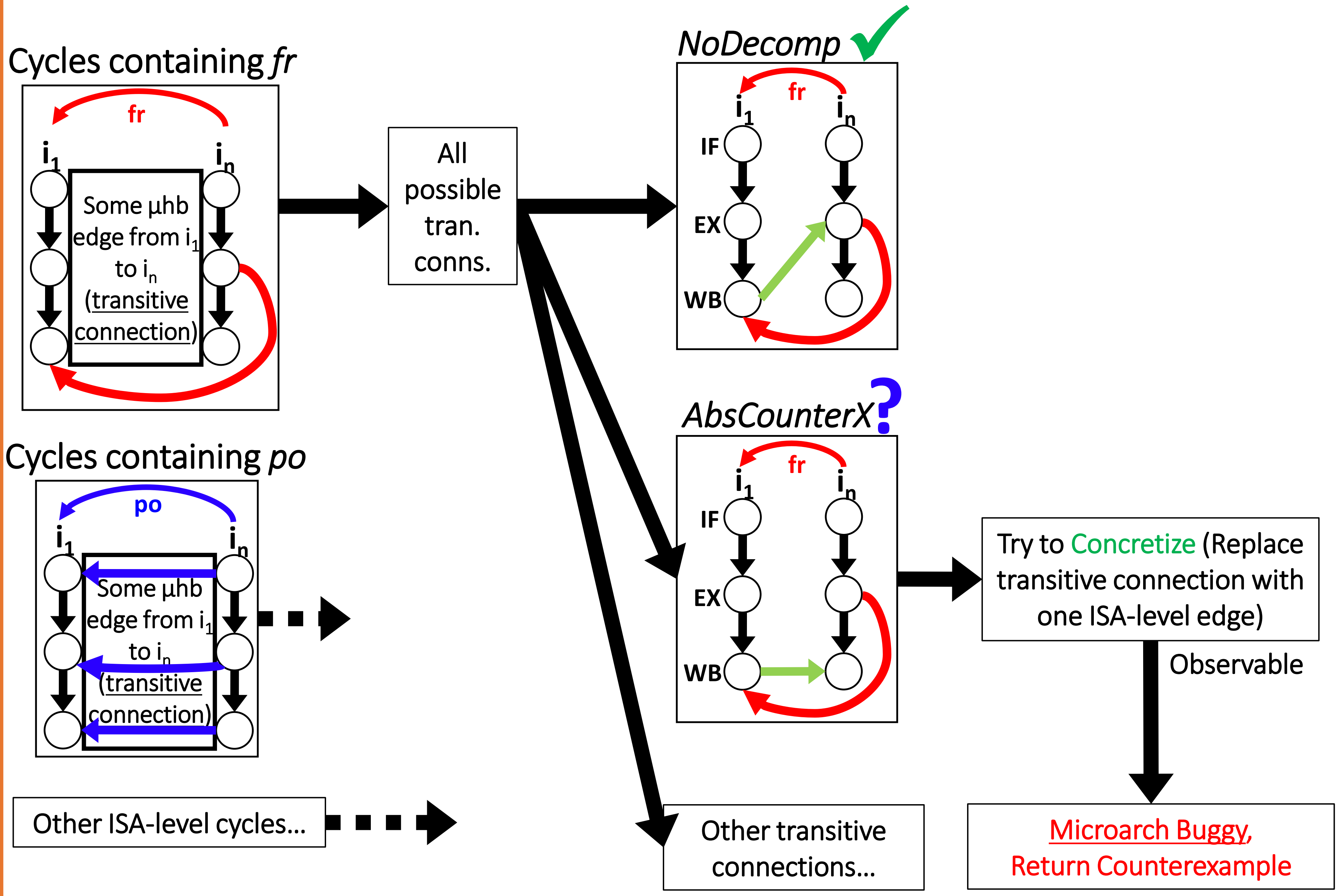
(i.e. possible bug)

Microarchitectural Correctness Proof


Cycles containing *fr*
fr
i1
in
Some μhb edge from i1 to in (transitive connection)
Cycles containing *po*
po
i1
in
Some μhb edge from i1 to in (transitive connection)
Other ISA-level cycles…
All possible tran. conns.
NoDecomp
fr
i1
in
IF
EX
WB
AbsCounterX
fr
i1
in
IF
EX
WB
Other transitive connections…

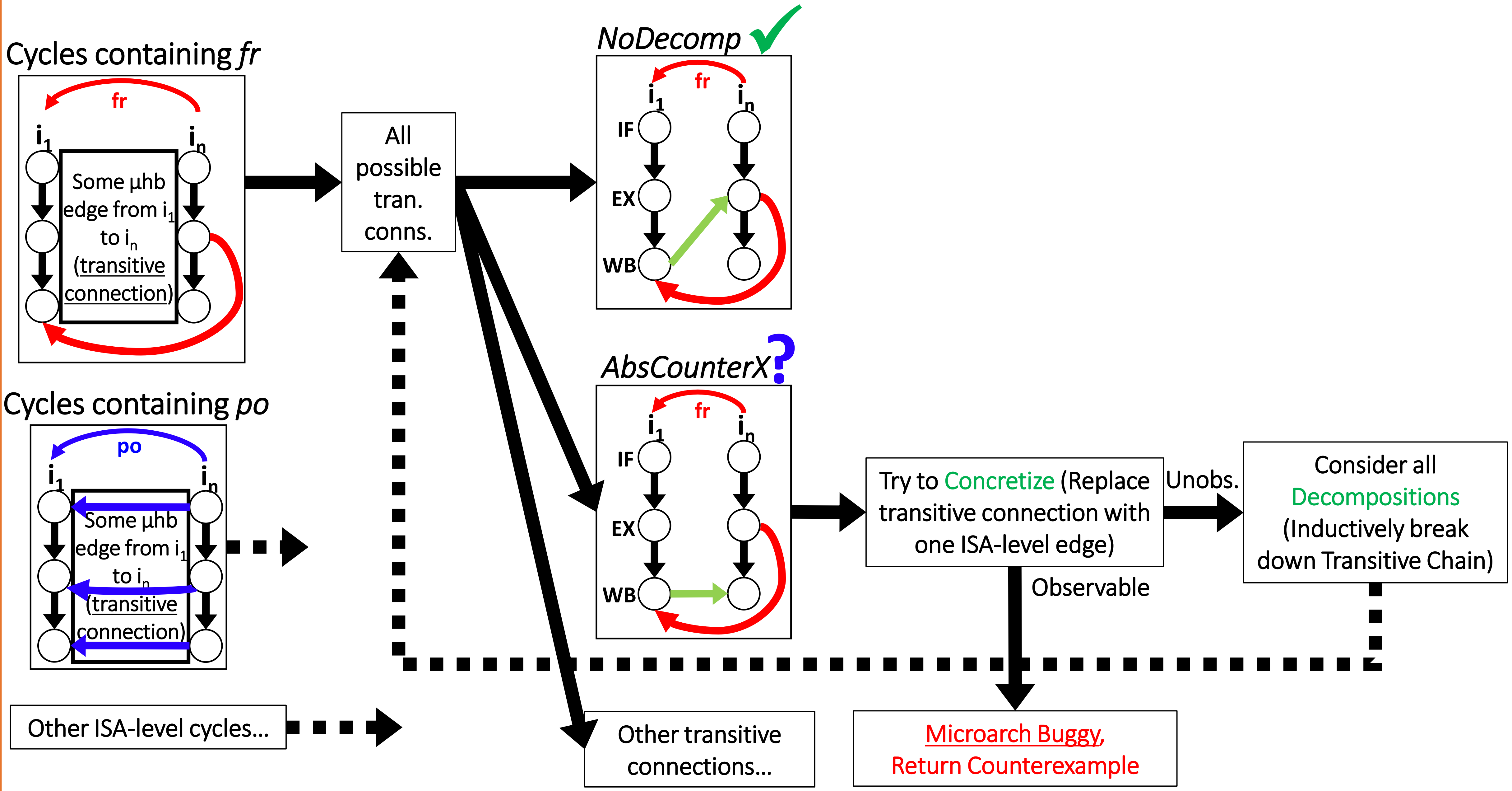
Transitive connection may represent one or multiple ISA-level edges

Microarchitectural Correctness Proof

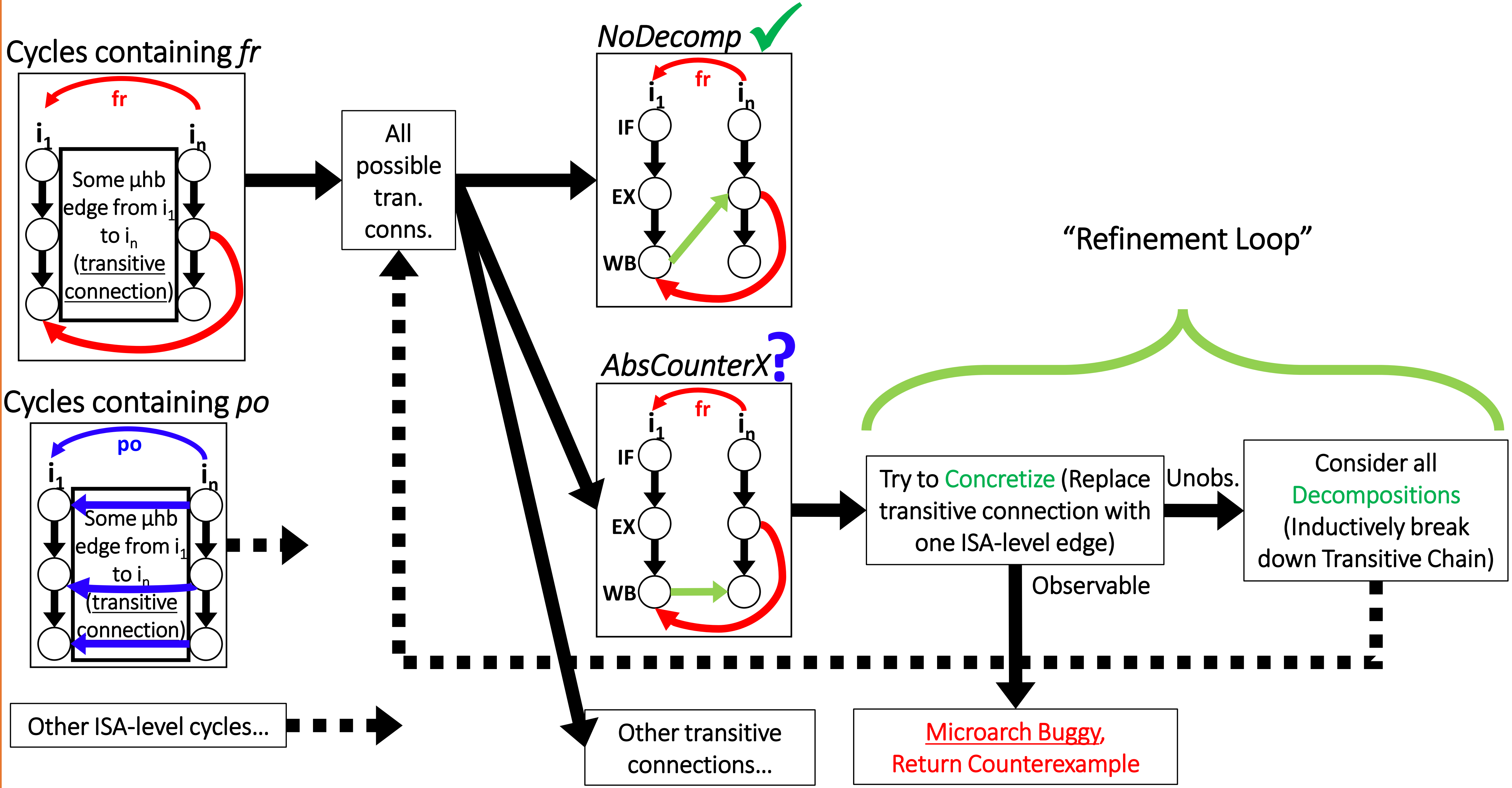


Cycles containing *fr*
fr
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Cycles containing *po*
po
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Other ISA-level cycles...
All possible tran. conns.
NoDecomp ✓
fr
i_1
i_n
IF
EX
WB
AbsCounterX ?
fr
i_1
i_n
IF
EX
WB
Other transitive connections...
Try to Concretize (Replace transitive connection with one ISA-level edge)
Observable
Microarch Buggy, Return Counterexample

Microarchitectural Correctness Proof

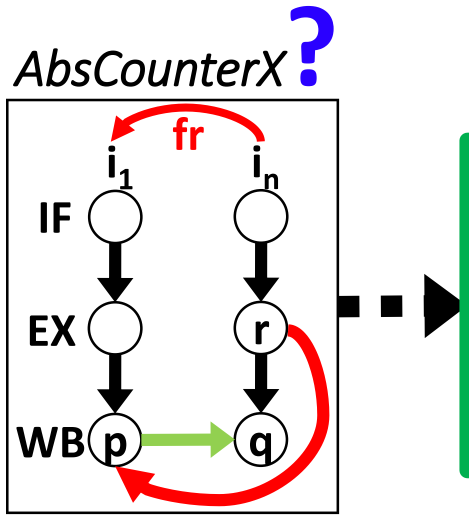

Cycles containing *fr*
fr
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Cycles containing *po*
po
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Other ISA-level cycles…
All possible tran. conns.
NoDecomp
fr
i_1
i_n
IF
EX
WB
AbsCounterX
fr
i_1
i_n
IF
EX
WB
Try to Concretize (Replace transitive connection with one ISA-level edge)
Unobs.
Consider all Decompositions (Inductively break down Transitive Chain)
Observable
Other transitive connections…
Microarch Buggy, Return Counterexample

Microarchitectural Correctness Proof


Cycles containing *fr*
fr
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Cycles containing *po*
po
i_1
i_n
Some μhb edge from i_1 to i_n (transitive connection)
Other ISA-level cycles…
All possible tran. conns.
NoDecomp ✓
fr
i_1
i_n
IF
EX
WB
AbsCounterX?
fr
i_1
i_n
IF
EX
WB
Other transitive connections…
"Refinement Loop"
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

Concretization

- All concretizations must be unobservable
- Observable concretizations are counterexamples

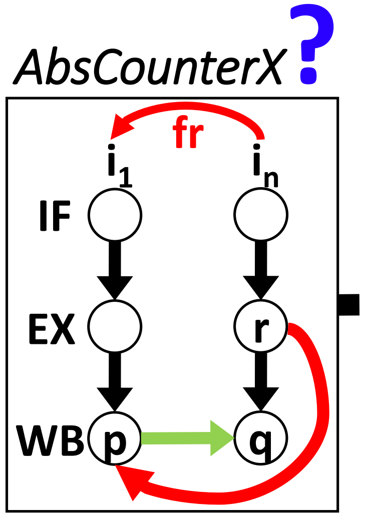


Concretization:

Replace transitive connection with single ISA-level edge

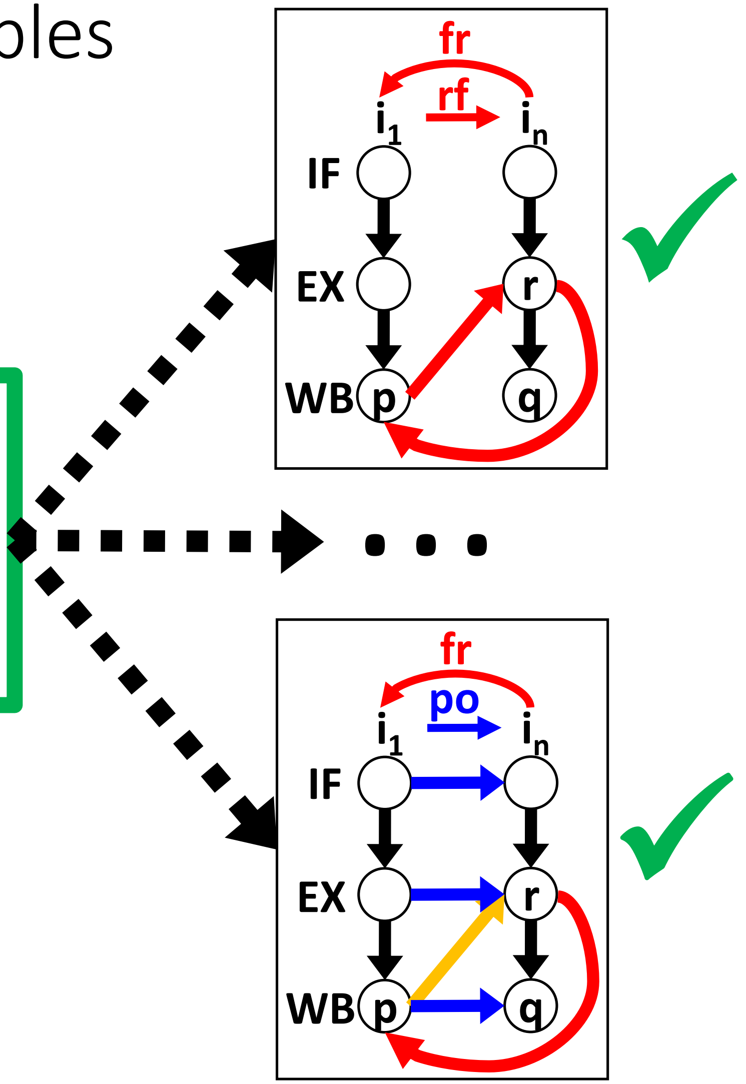
Concretization

- All concretizations must be unobservable
- Observable concretizations are counterexamples



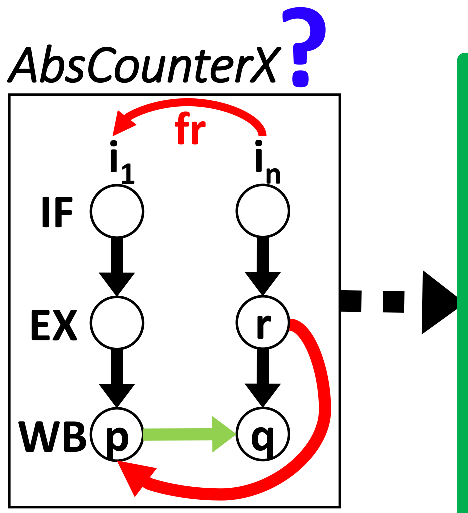
Concretization:

Replace transitive connection with single ISA-level edge



Decomposition

- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable



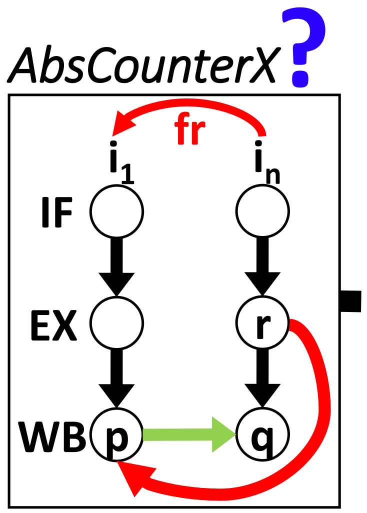
Decomposition:

Inductively break down transitive chain

(Chain of length n == Chain of length n-1 + single “peeled-off” edge)

Decomposition

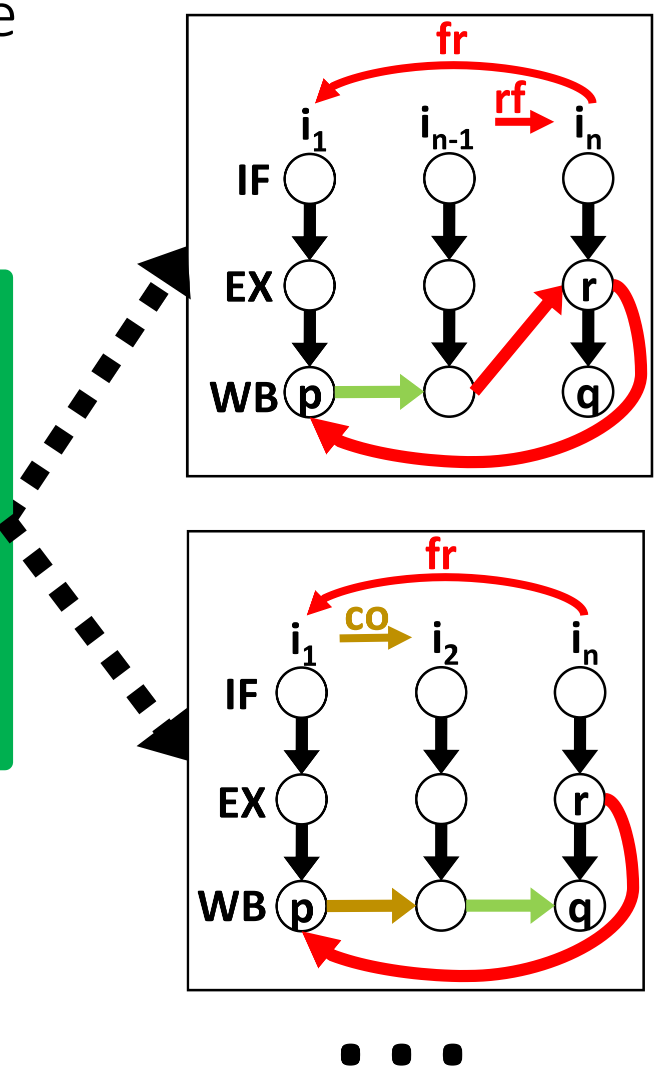
- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable



Decomposition:

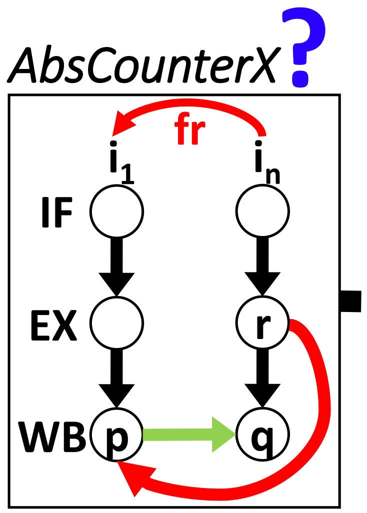
Inductively break down transitive chain

(Chain of length n == Chain of length n-1 + single "peeled-off" edge)



Decomposition

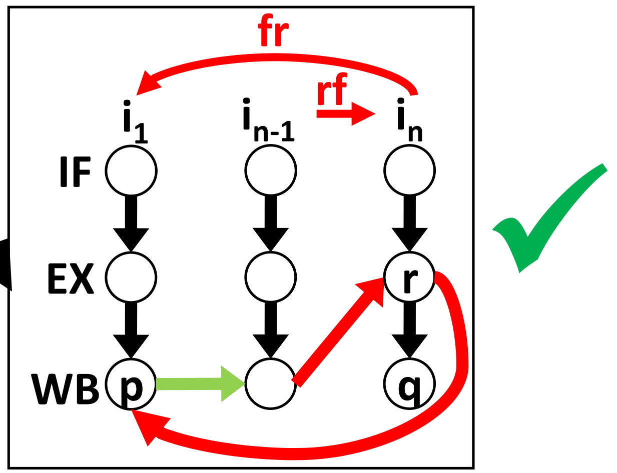
- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

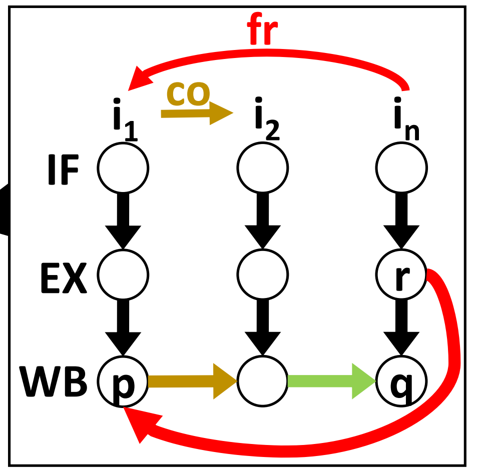


Decomposition:

Inductively break down transitive chain

(Chain of length n == Chain of length n-1 + single "peeled-off" edge)

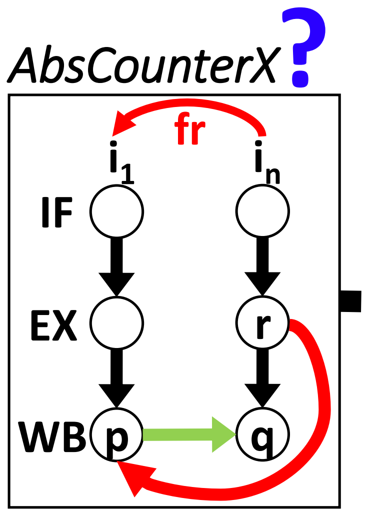




...

Decomposition

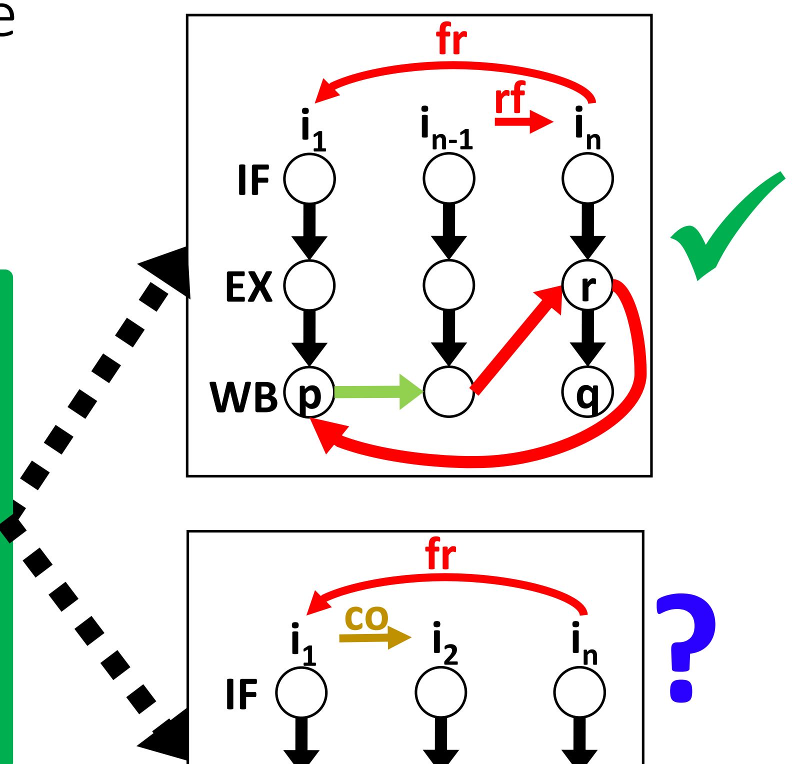
- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable

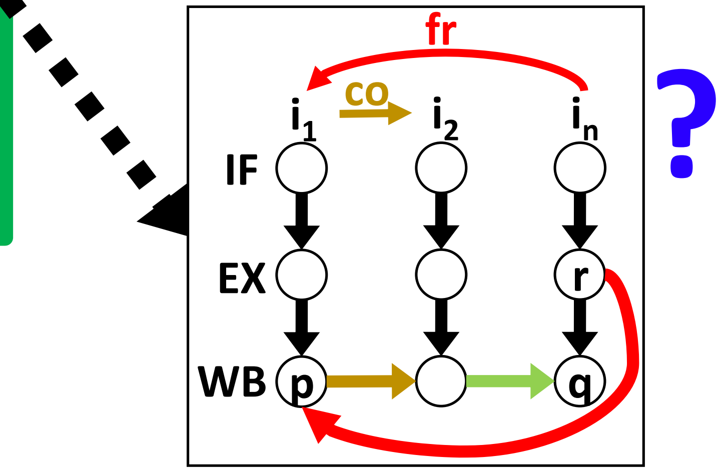


Decomposition:

Inductively break down transitive chain

(Chain of length n == Chain of length n-1 + single "peeled-off" edge)

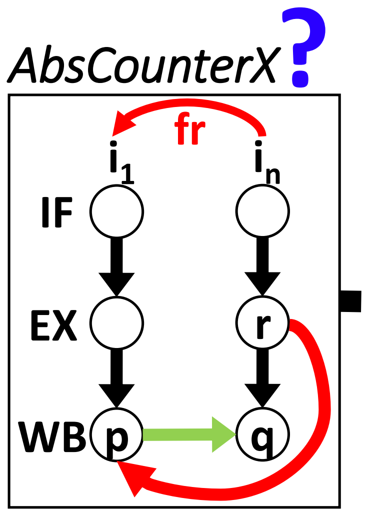




. . .

Decomposition

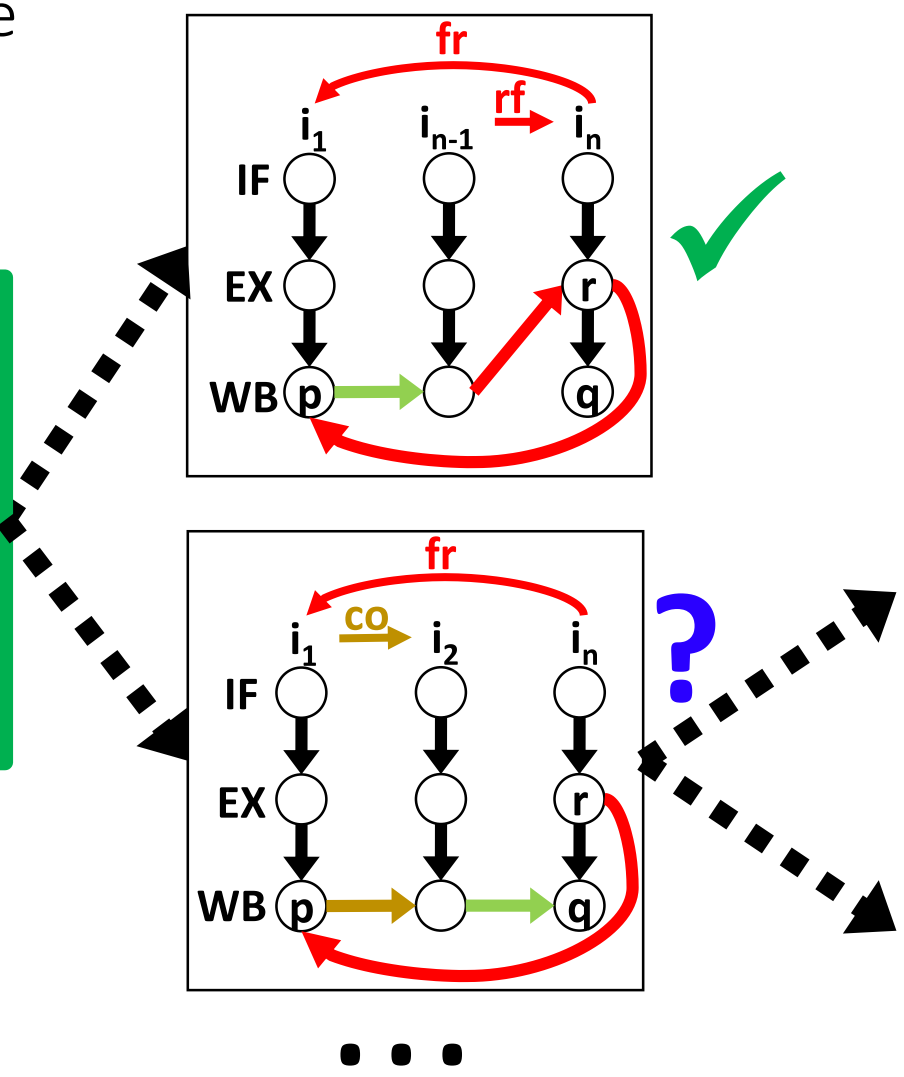
- Additional instruction and ISA-level edge modelled => extra constraints
  - May be enough to make execution unobservable



Decomposition:

Inductively break down transitive chain

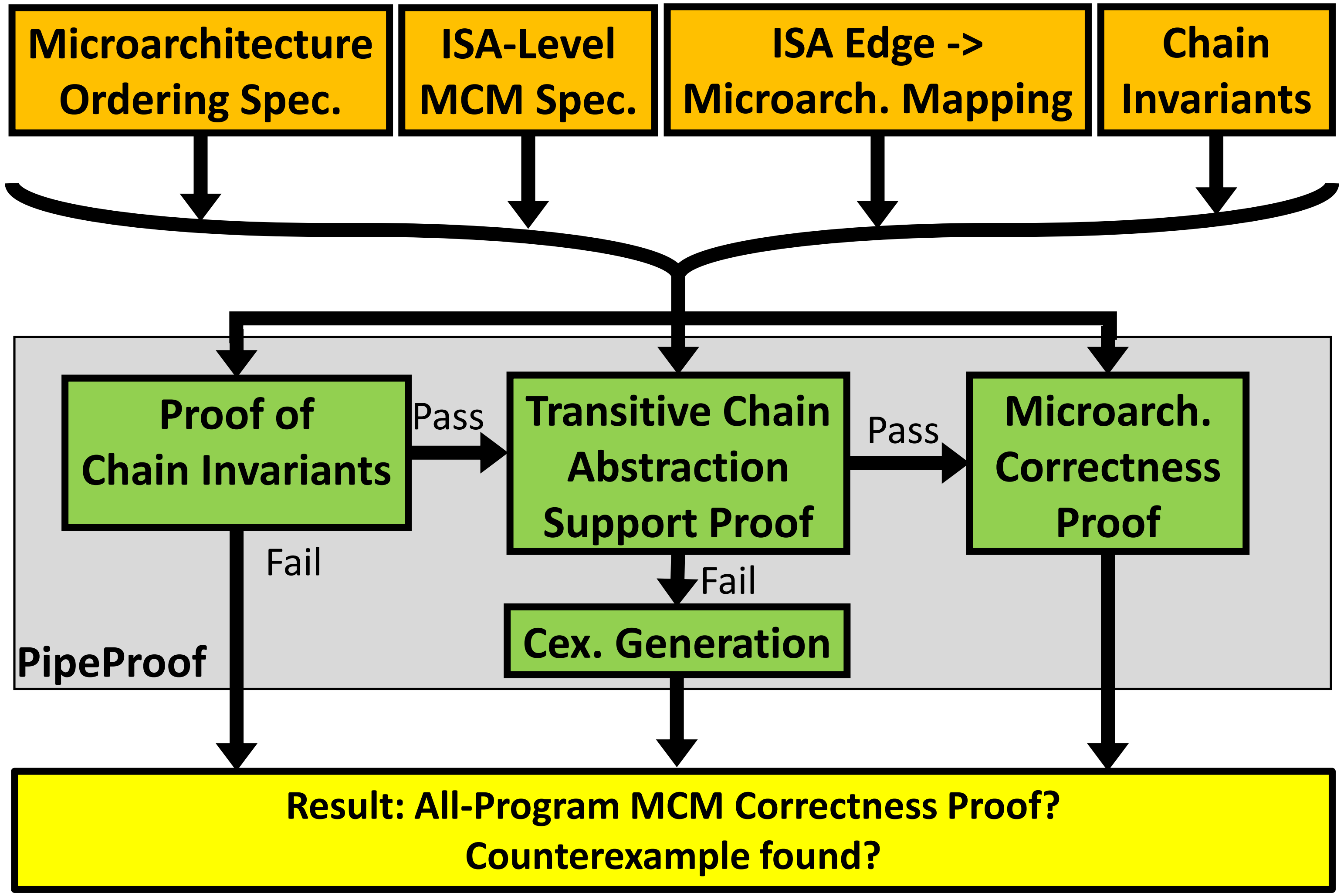
(Chain of length n == Chain of length n-1 + single "peeled-off" edge)



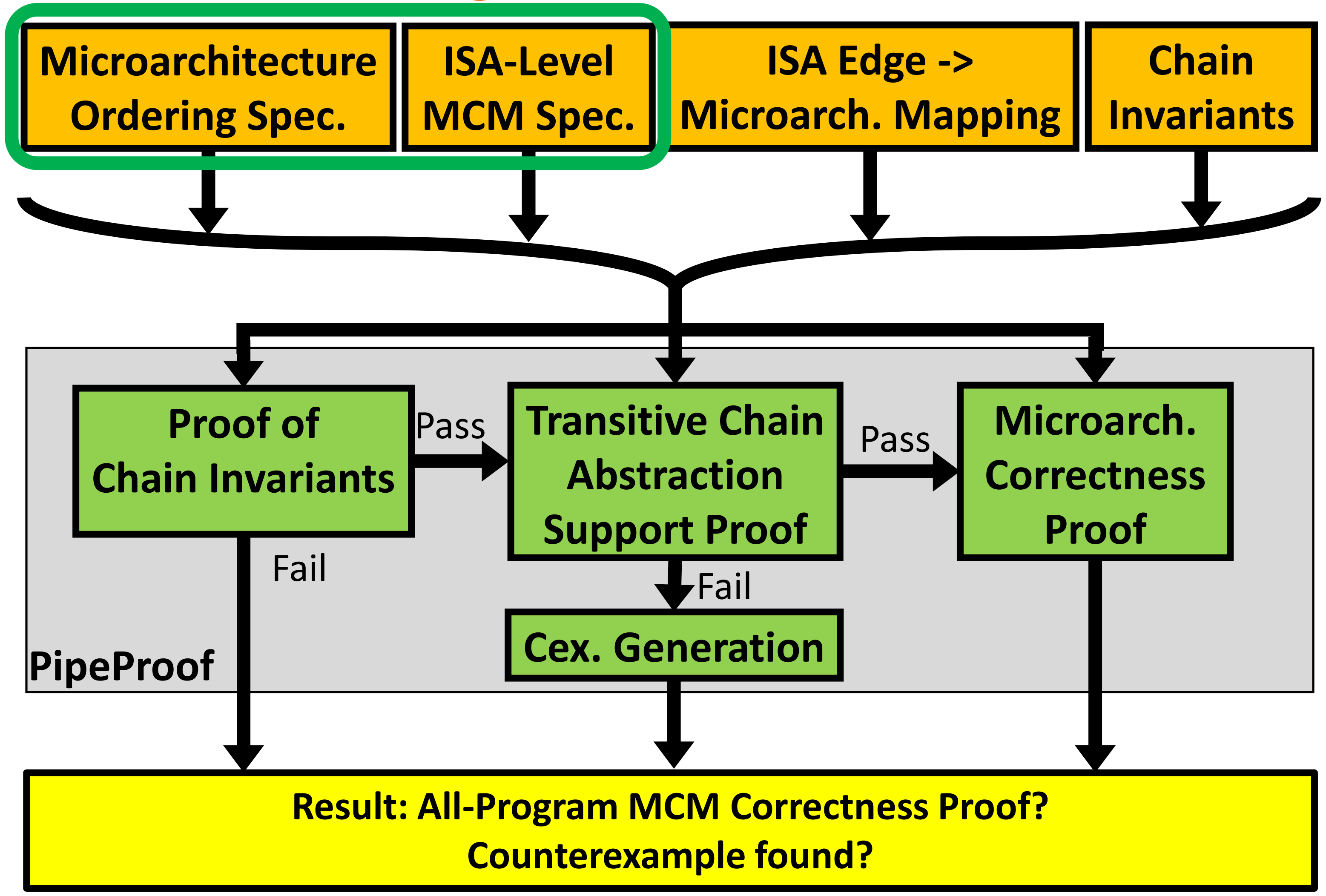
Outline

- Background
  - ISA-level MCM specs
  - Microarchitectural ordering specs
- Microarchitectural Correctness Proof
  - Transitive Chain (TC) Abstraction
- Overall PipeProof Operation
  - TC Abstraction Support Proof
  - Chain Invariants
- Results

PipeProof Block Diagram

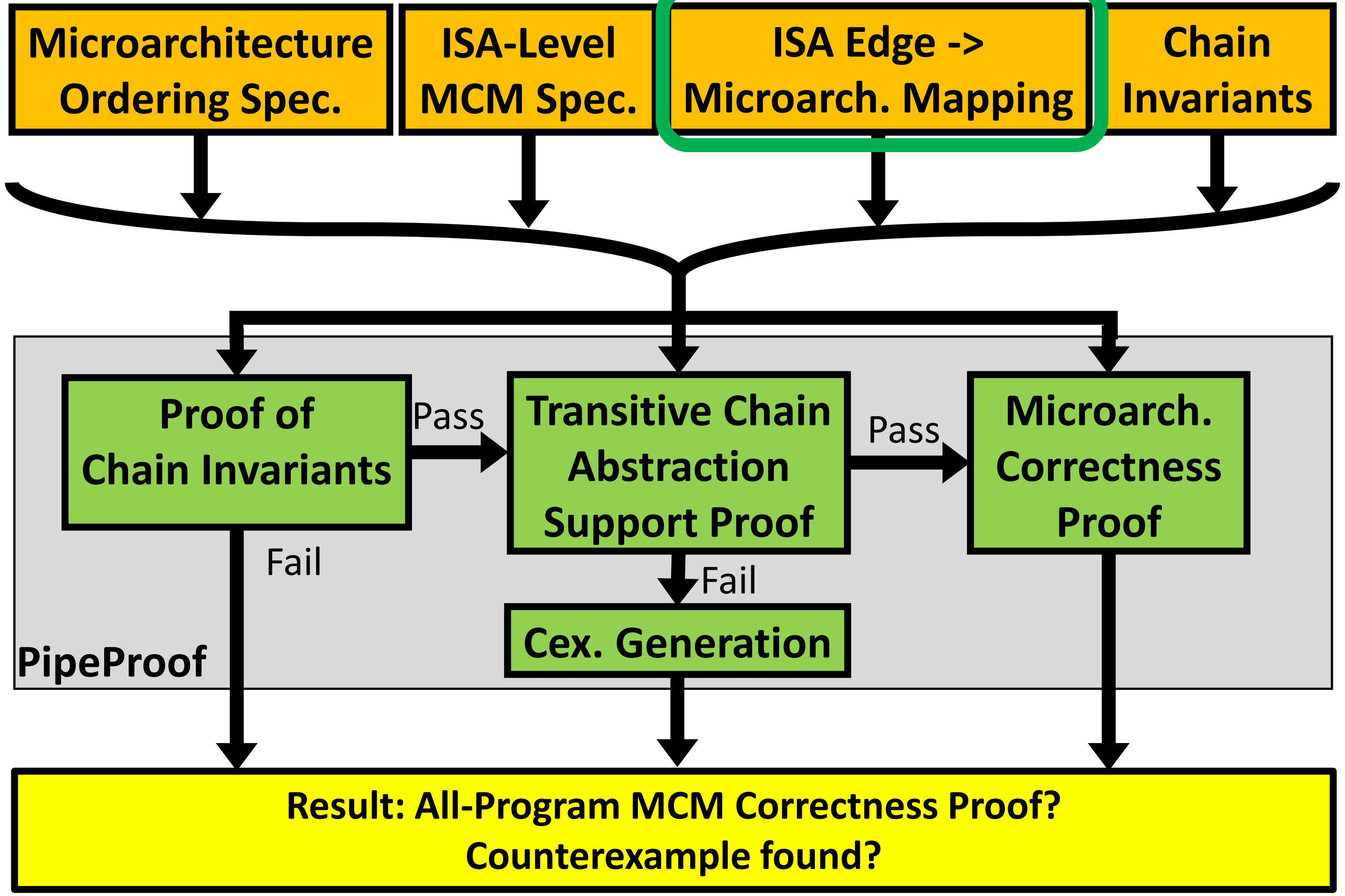

Microarchitecture Ordering Spec.
ISA-Level MCM Spec.
ISA Edge -> Microarch. Mapping
Chain Invariants
Proof of Chain Invariants
Pass
Transitive Chain Abstraction Support Proof
Pass
Microarch. Correctness Proof
Fail
Fail
Cex. Generation
PipeProof
Result: All-Program MCM Correctness Proof? Counterexample found?

PipeProof Block Diagram


Microarchitecture Ordering Spec.
ISA-Level MCM Spec.
ISA Edge -> Microarch. Mapping
Chain Invariants
Proof of Chain Invariants
Pass
Transitive Chain Abstraction Support Proof
Pass
Microarch. Correctness Proof
Fail
Fail
Cex. Generation
PipeProof
Result: All-Program MCM Correctness Proof? Counterexample found?

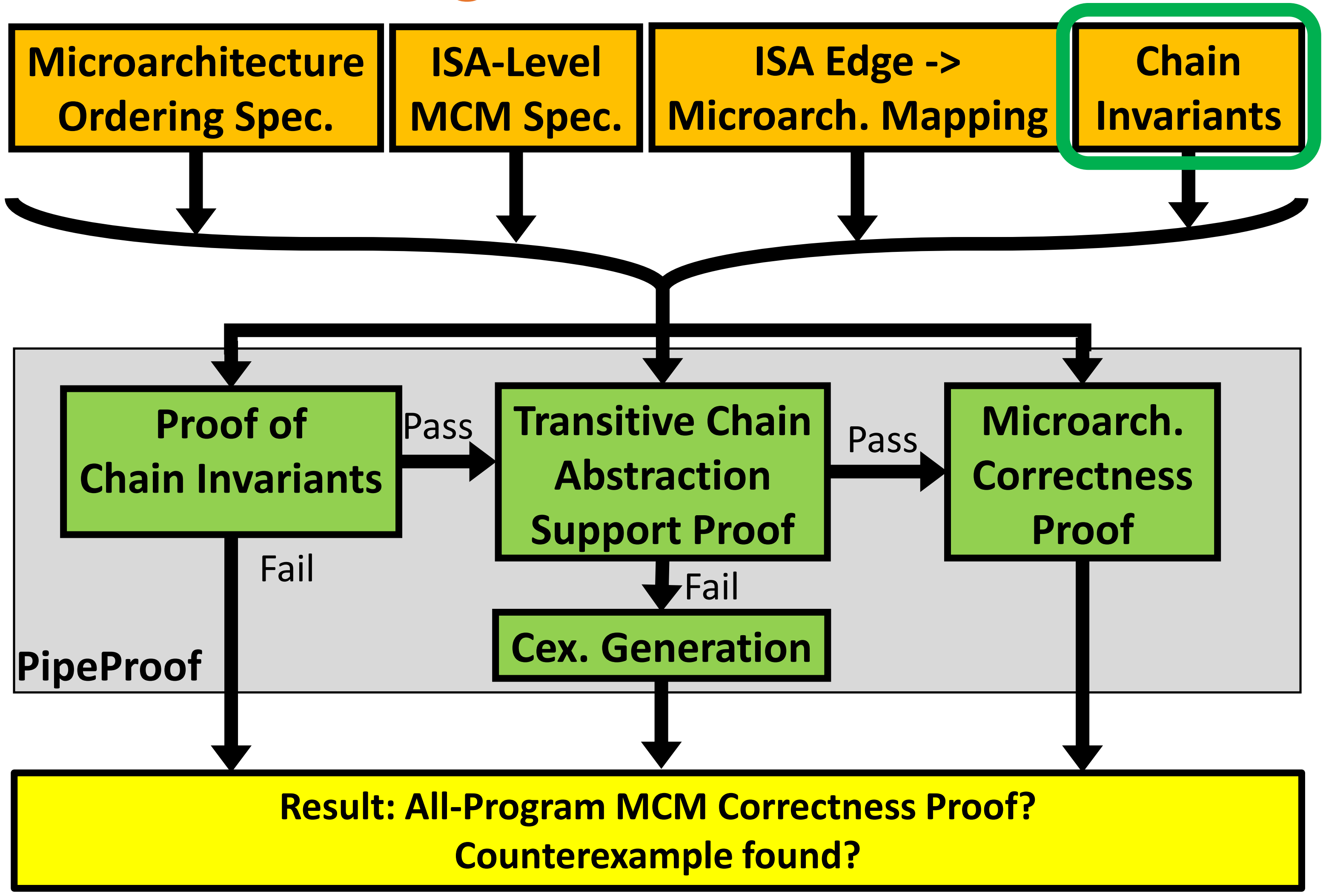
PipeProof Block Diagram

Links ISA-level and µarch executions



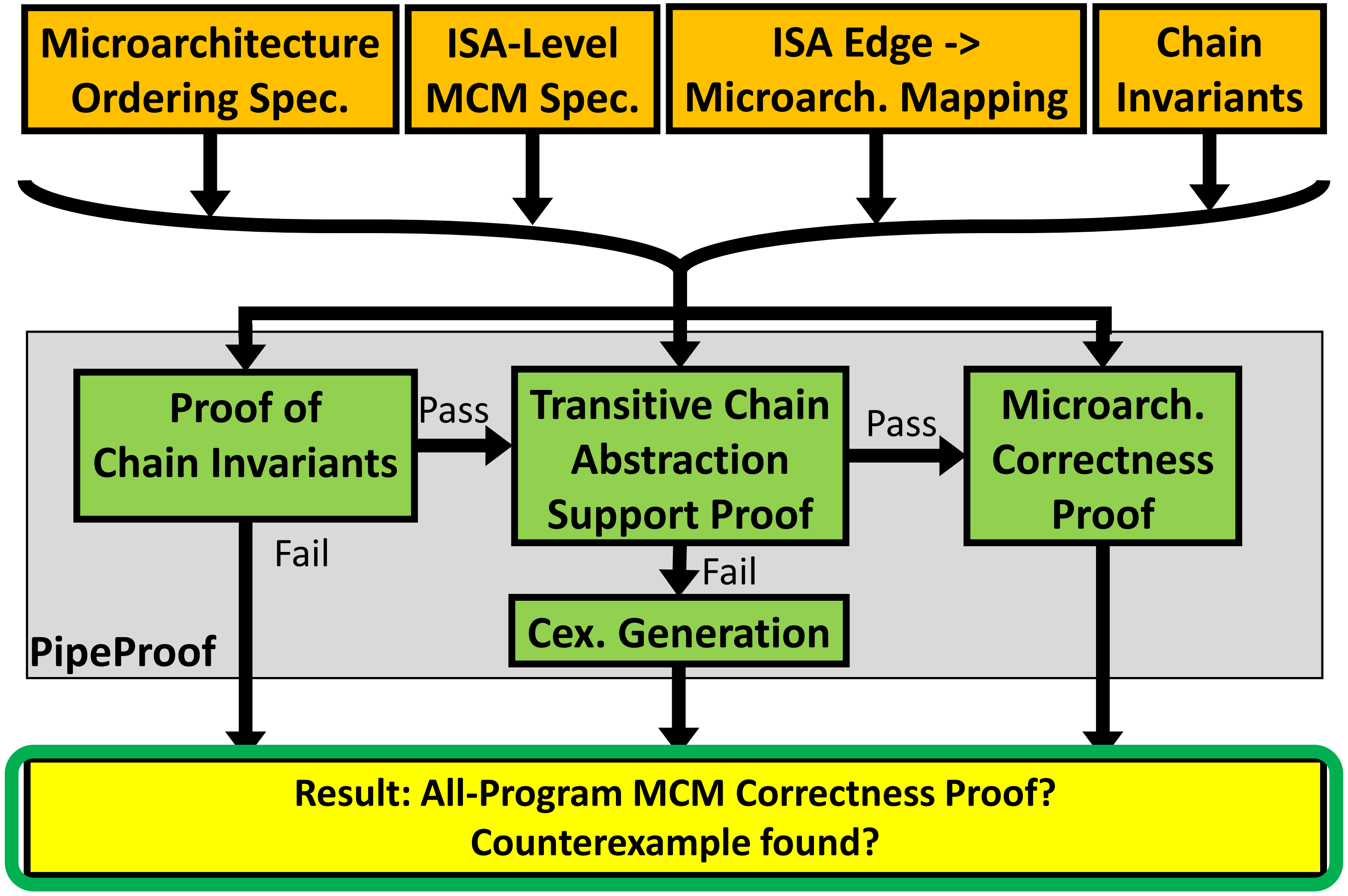
Microarchitecture Ordering Spec.
ISA-Level MCM Spec.
ISA Edge -> Microarch. Mapping
Chain Invariants
Proof of Chain Invariants
Pass
Transitive Chain Abstraction Support Proof
Pass
Microarch. Correctness Proof
Fail
Fail
Cex. Generation
PipeProof
Result: All-Program MCM Correctness Proof? Counterexample found?

PipeProof Block Diagram


Microarchitecture Ordering Spec.
ISA-Level MCM Spec.
ISA Edge -> Microarch. Mapping
Chain Invariants
Proof of Chain Invariants
Pass
Transitive Chain Abstraction Support Proof
Pass
Microarch. Correctness Proof
Fail
Fail
Cex. Generation
PipeProof
Result: All-Program MCM Correctness Proof? Counterexample found?

Represent repeated ISA-level patterns

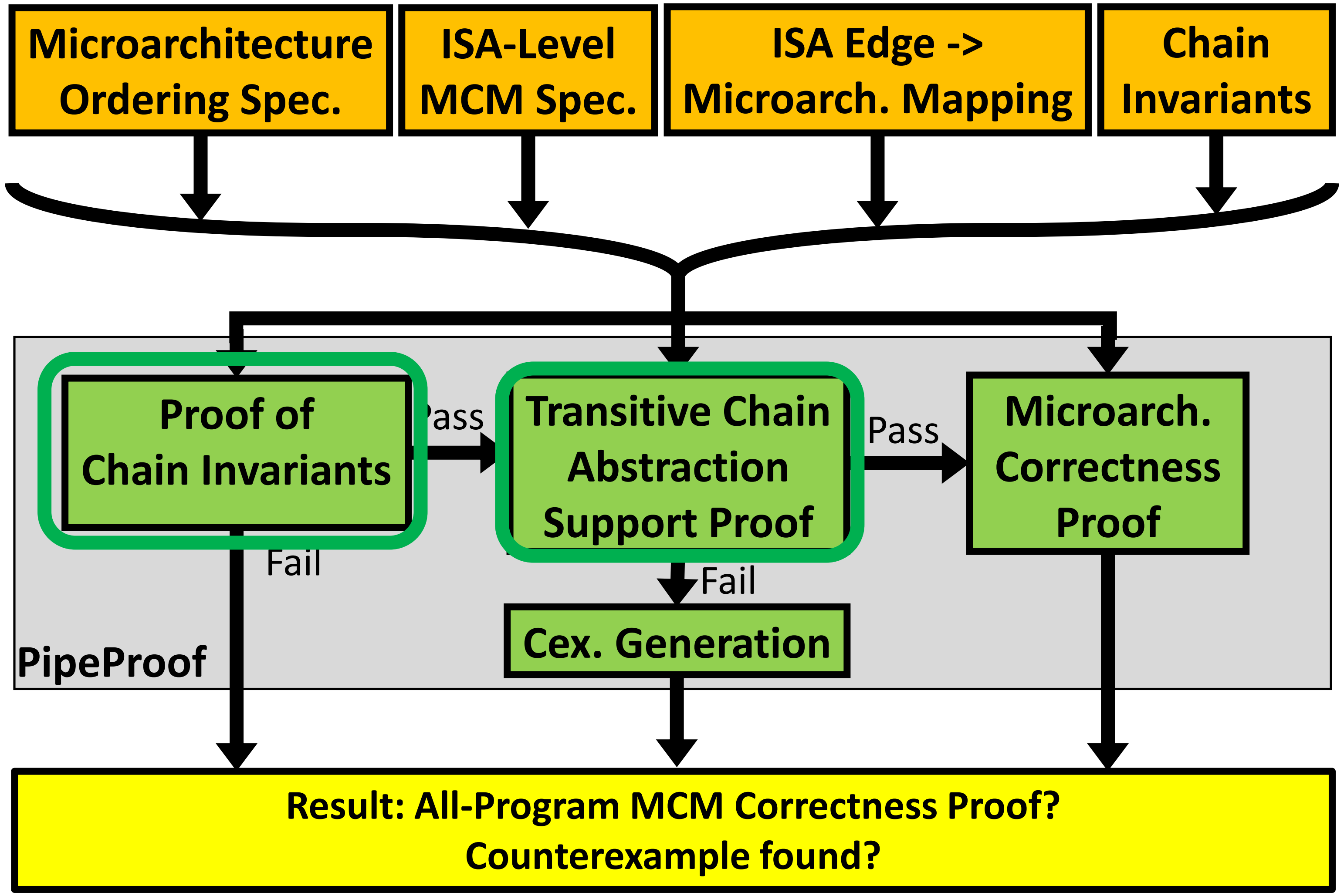
PipeProof Block Diagram



Microarchitecture Ordering Spec.
ISA-Level MCM Spec.
ISA Edge -> Microarch. Mapping
Chain Invariants
Proof of Chain Invariants
Pass
Transitive Chain Abstraction Support Proof
Pass
Microarch. Correctness Proof
Fail
Fail
Cex. Generation
PipeProof
Result: All-Program MCM Correctness Proof? Counterexample found?

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

PipeProof Block Diagram

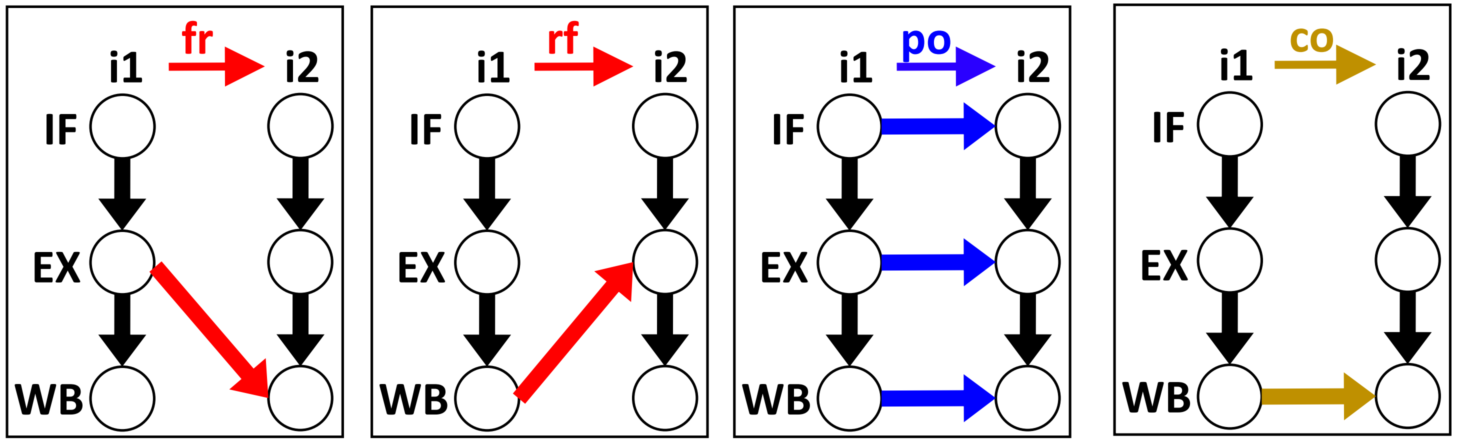


Microarchitecture Ordering Spec.
ISA-Level MCM Spec.
ISA Edge -> Microarch. Mapping
Chain Invariants
Proof of Chain Invariants
Pass
Fail
Transitive Chain Abstraction Support Proof
Pass
Fail
Microarch. Correctness Proof
Cex. Generation
PipeProof
Result: All-Program MCM Correctness Proof? Counterexample found?

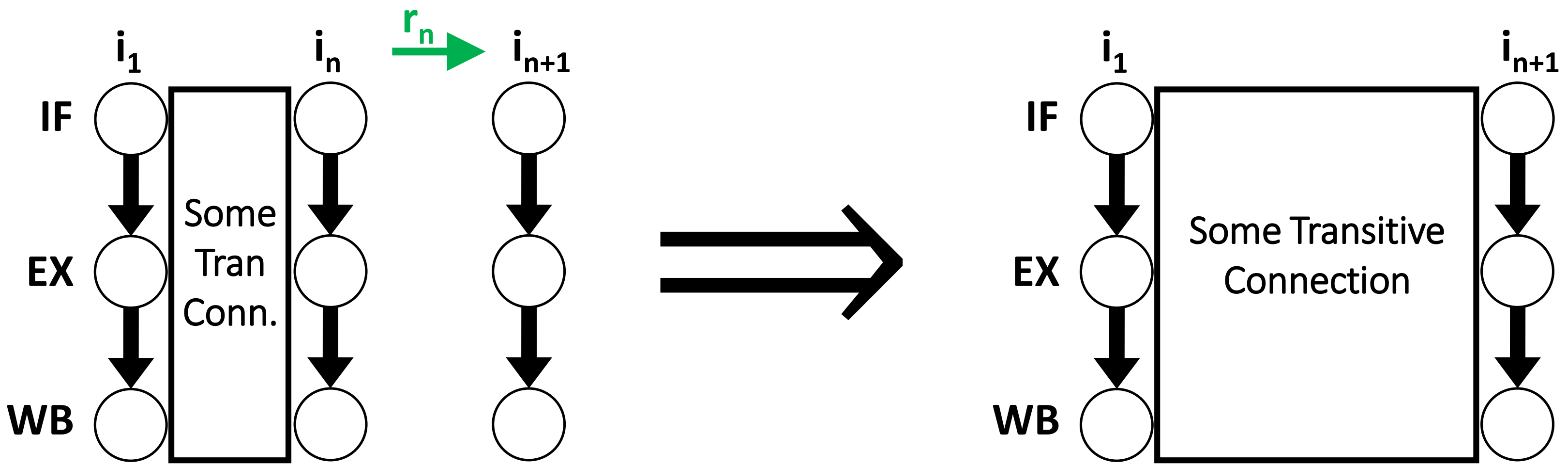
Supporting proofs provide foundation for correctness proof

Transitive Chain (TC) Abstraction Support Proof

- Ensure that ISA-level pattern and µarch. support TC Abstraction
- Base case: Do initial ISA-level edges guarantee connection?



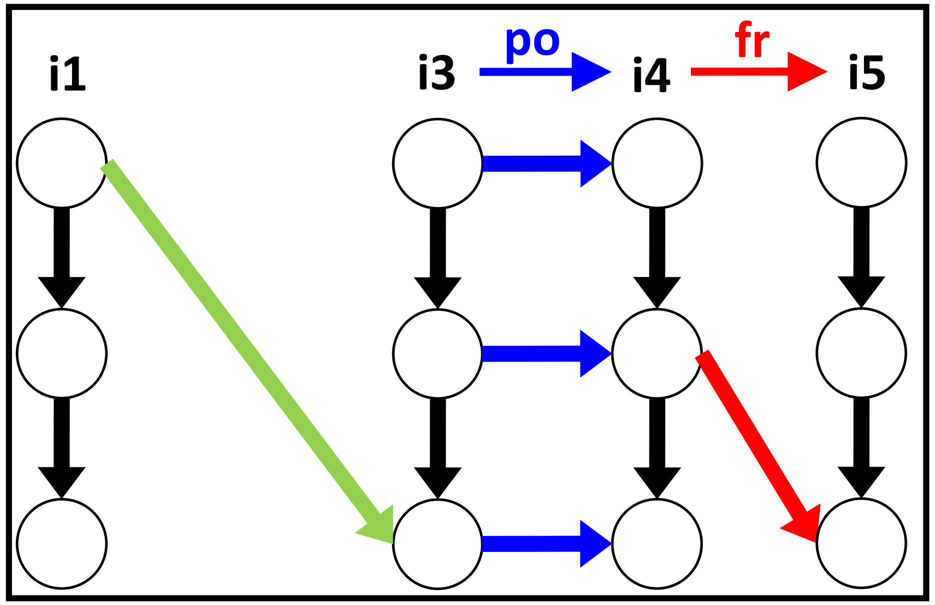
- Inductive case: Extend transitive chain => extend transitive connection?



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
- <u>Example:</u> checking for edge from i1 to i5 (TC abstraction support proof)

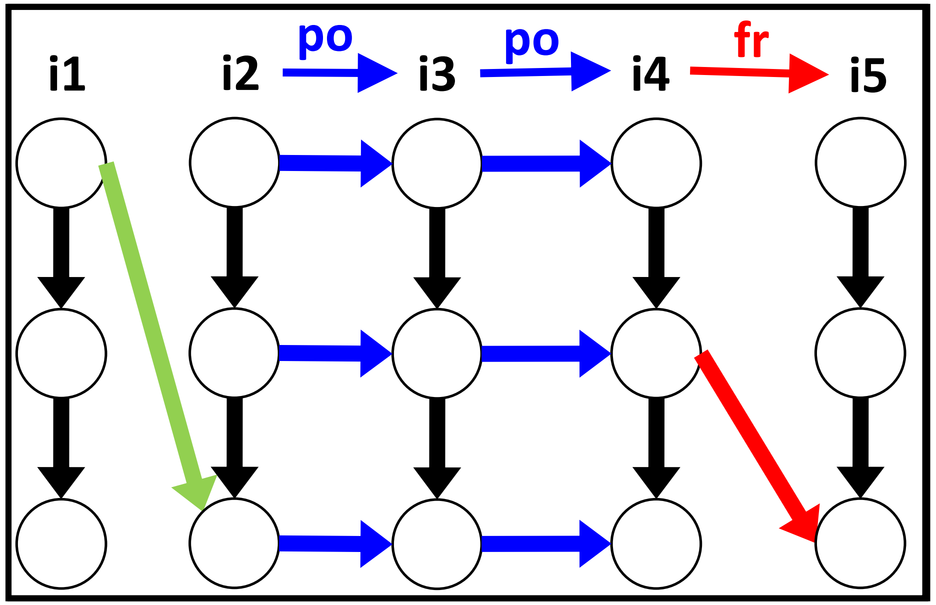
Abstract Counterexample



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
- <u>**Example:**</u> checking for edge from i1 to i5 (TC abstraction support proof)

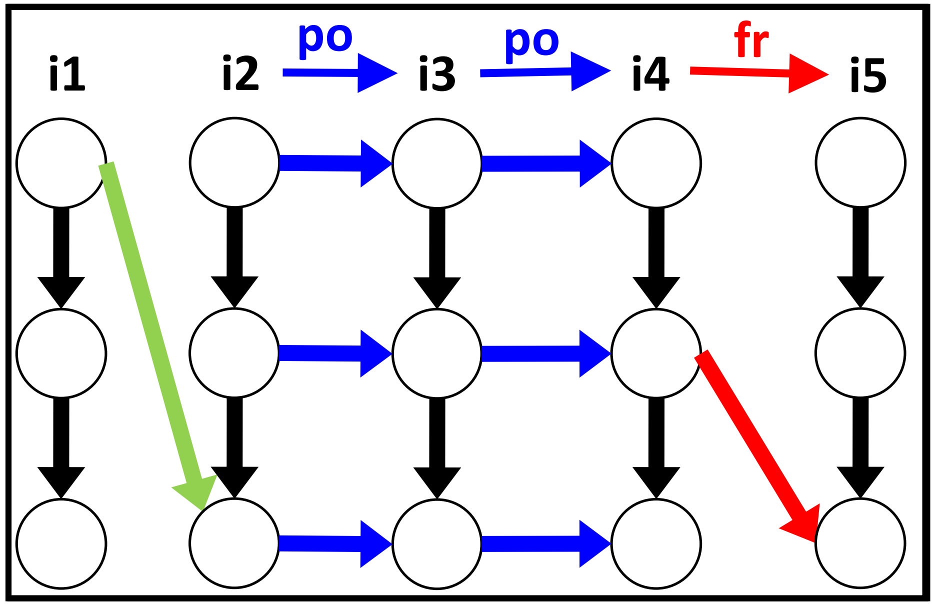
Repeating ISA-Level Pattern



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
- <u>**Example:**</u> checking for edge from i1 to i5 (TC abstraction support proof)

Repeating ISA-Level Pattern

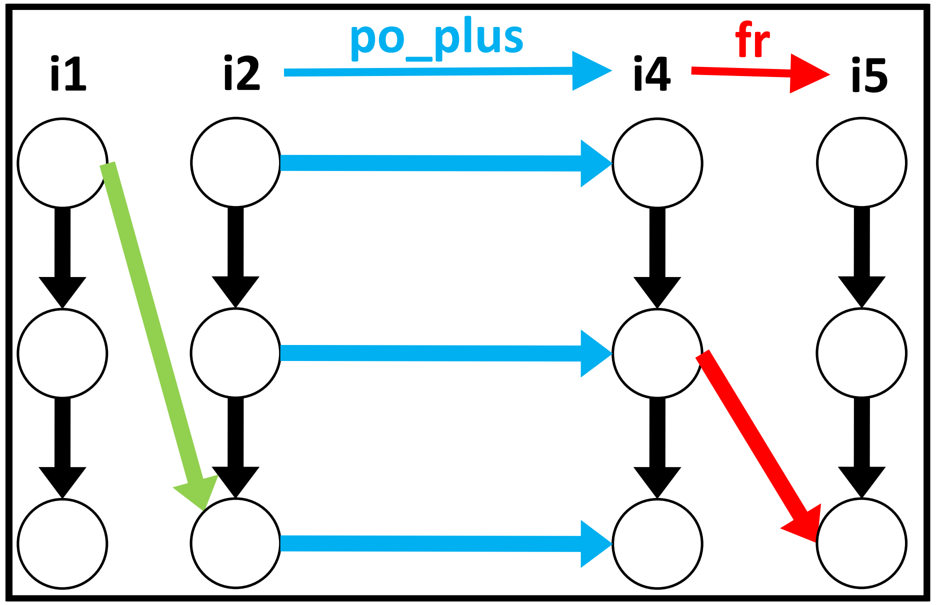


Can continue decomposing in this way forever!

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**

In the paper…

- Optimizations
  - Covering Sets: Eliminate redundant transitive connections
  - Memoization: Eliminate redundant ISA-level cycles
- Inductive ISA edge generation
- Adequate Model Over-Approximation
  - Needed to ensure soundness of PipeProof's abstraction-based approach
- …and more!

Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) µarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

	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)**

Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) μarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

	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)**

Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) μarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

	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)**

Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) μarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

	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)**

Results

- Ran PipeProof on simpleSC (SC) and simpleTSO (TSO) µarches
  - 3-stage in-order pipelines
- Proved correctness of both microarchitectures for all programs
  - With optimizations, runtimes < 1 hour!

	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)**

Conclusions

- **PipeProof:** Automated ***All-Program*** Microarchitectural MCM Verification
  - Designers no longer need to choose between completeness and automation
- **Transitive Chain Abstraction** allows inductive modelling and verification of the infinite set of all possible executions
  - Abstraction is automatically refined as necessary to prove correctness
- Verified simple microarchitectures implementing SC and TSO in < 1 hour!

Code available at
https://github.com/ymanerka/pipeproof

[Image: Napish]

PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications

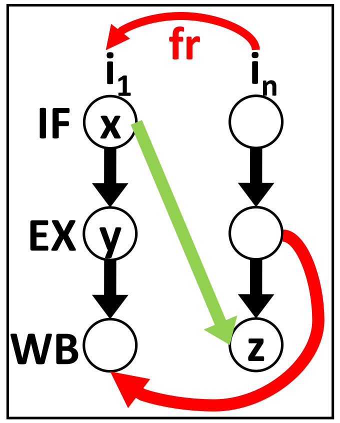
Yatin A. Manerkar, Daniel Lustig*,
Margaret Martonosi, and Aarti Gupta

Code available at
https://github.com/ymanerka/pipeproof

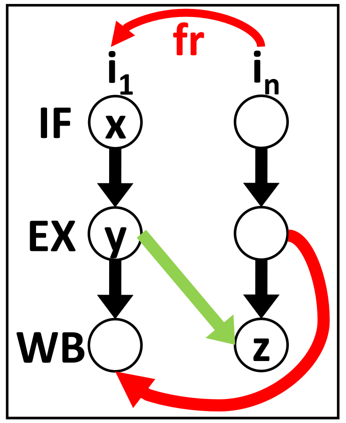
Covering Sets Optimization

- 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

A



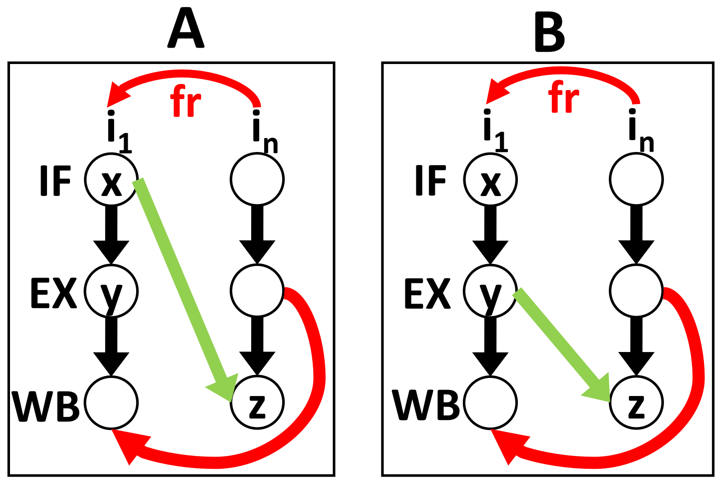
B



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

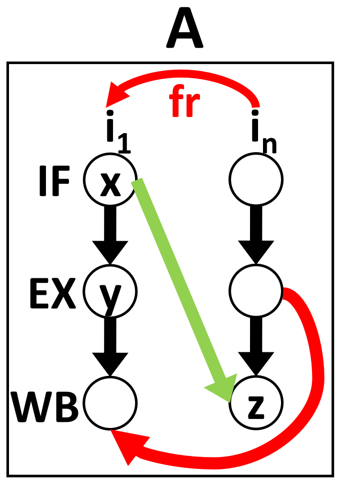
Graph A has an edge from x→z (tran conn.)

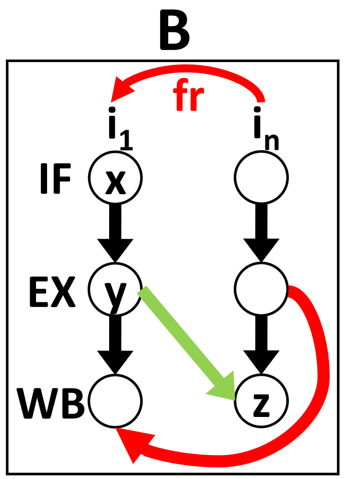


Covering Sets Optimization

- 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.)





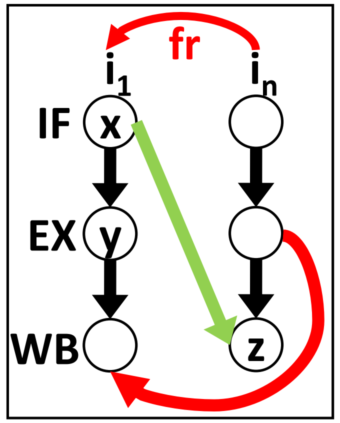
Graph B has edges from y→z (tran conn.) and x→z (by transitivity)

Covering Sets Optimization

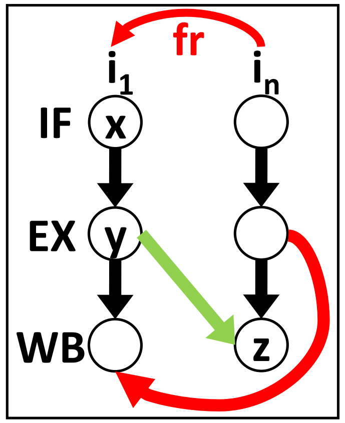
- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections

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

A



B

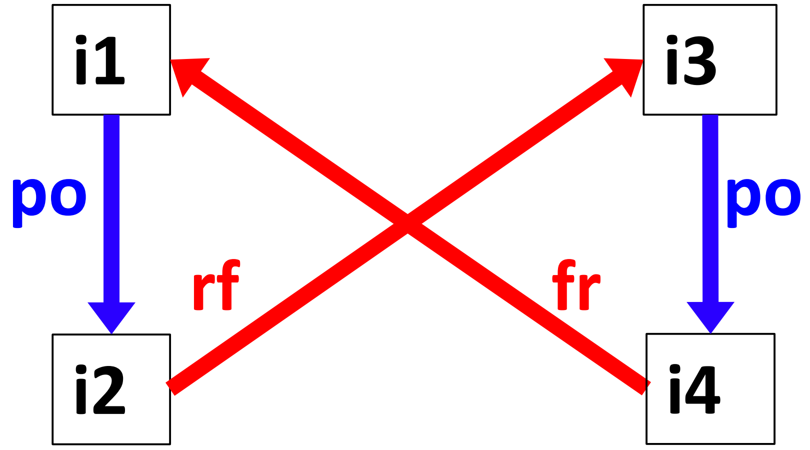


Graph B has edges from y→z (tran conn.) and x→z (by transitivity)

Correctness of A => Correctness of B (since B contains A's tran conn.)
Checking B explicitly is redundant!

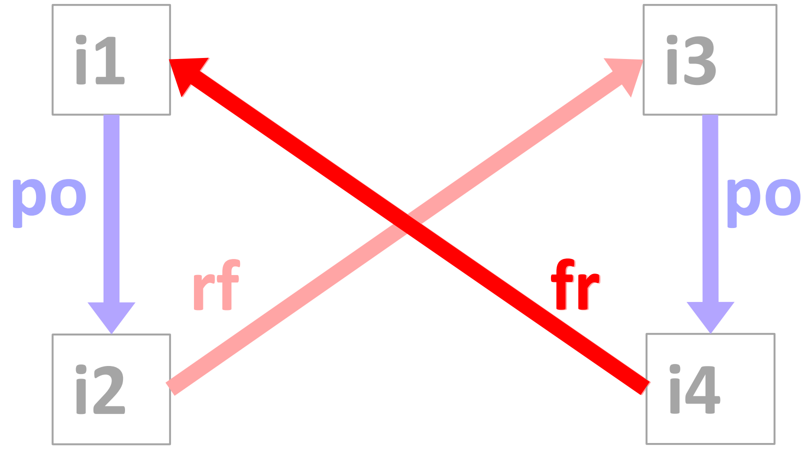
Memoization Optimization

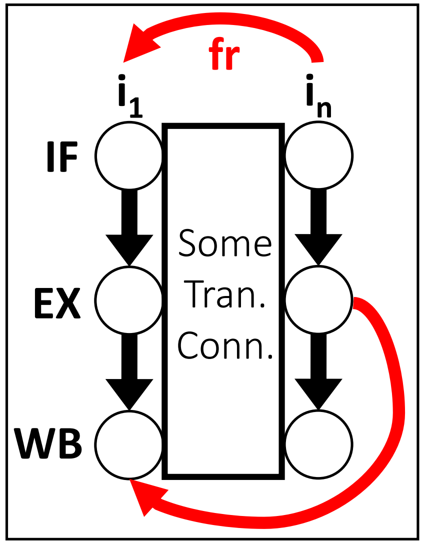
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified



Memoization Optimization

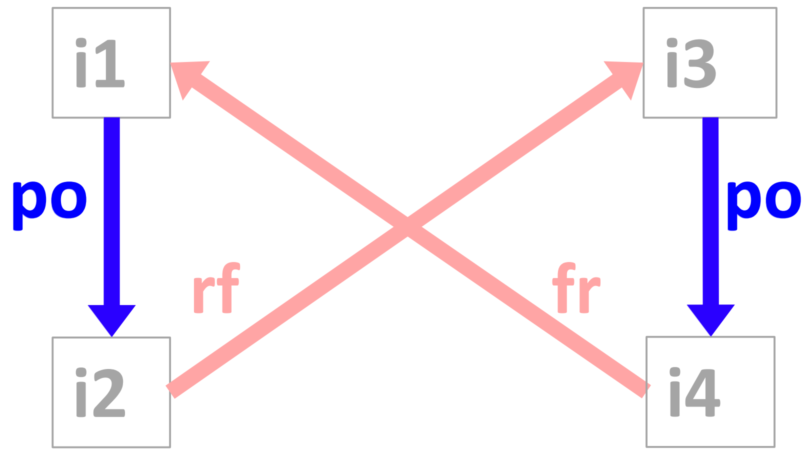
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

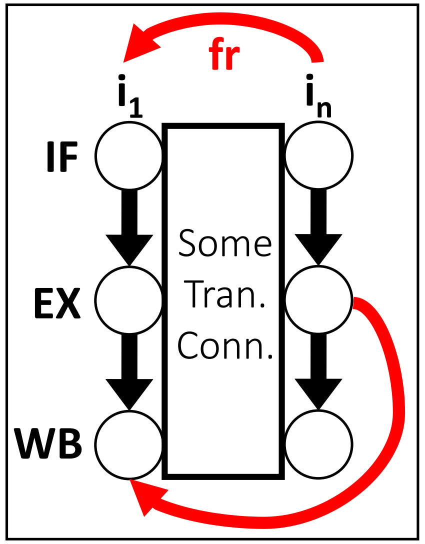


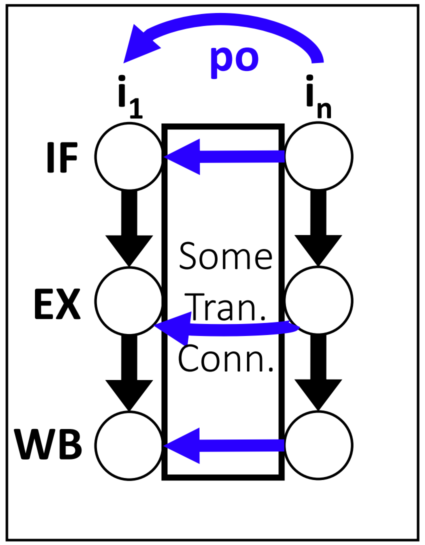


Memoization Optimization

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

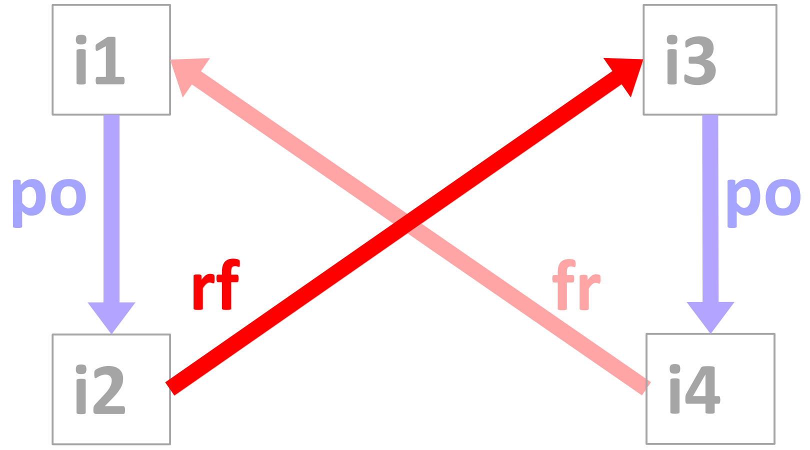




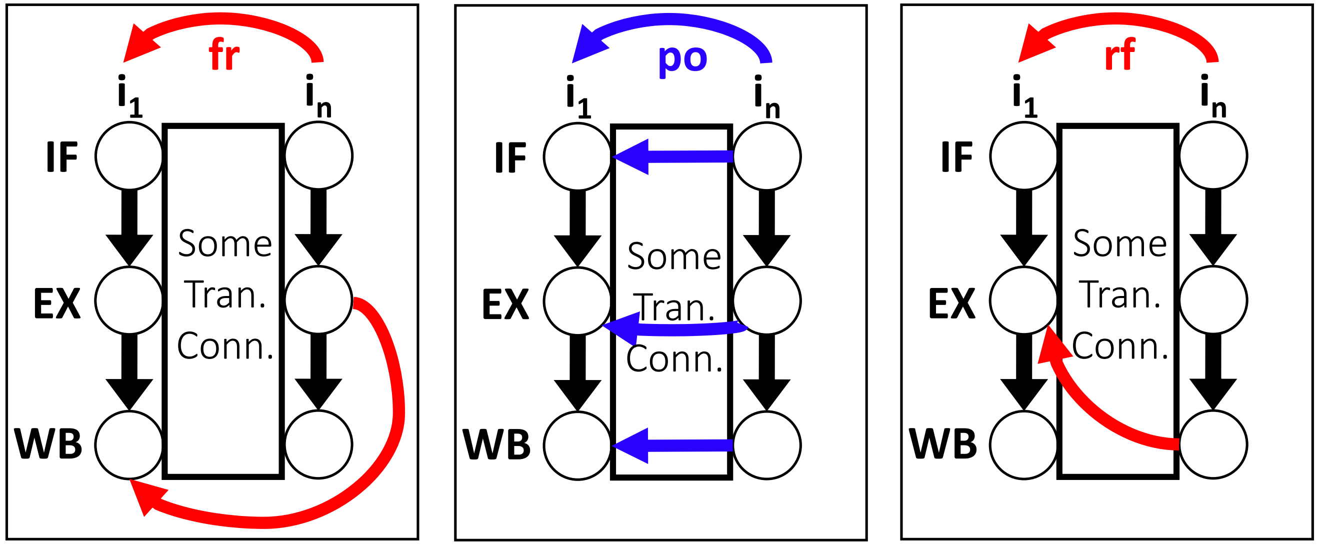


Memoization Optimization

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

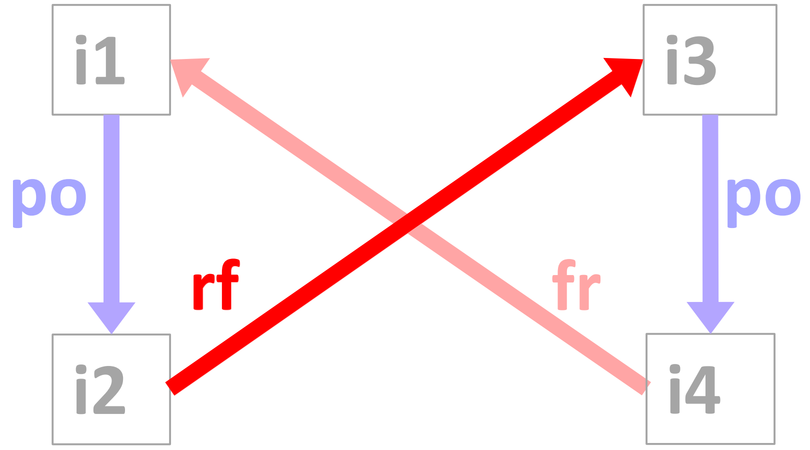


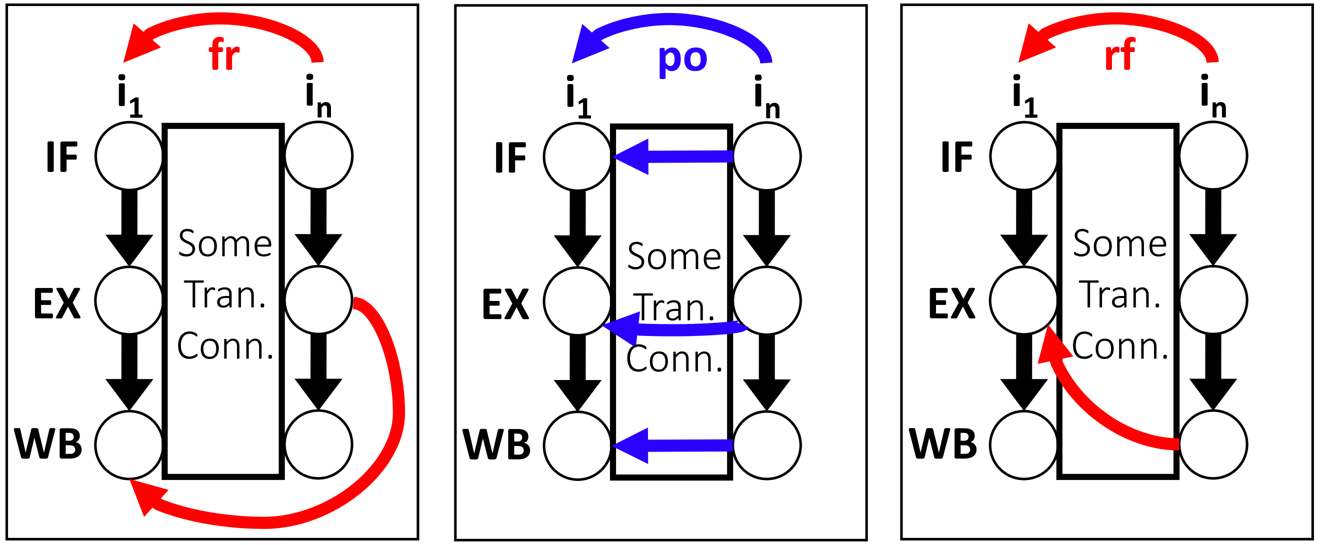
Same cycle is checked 3 times!



Memoization Optimization

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



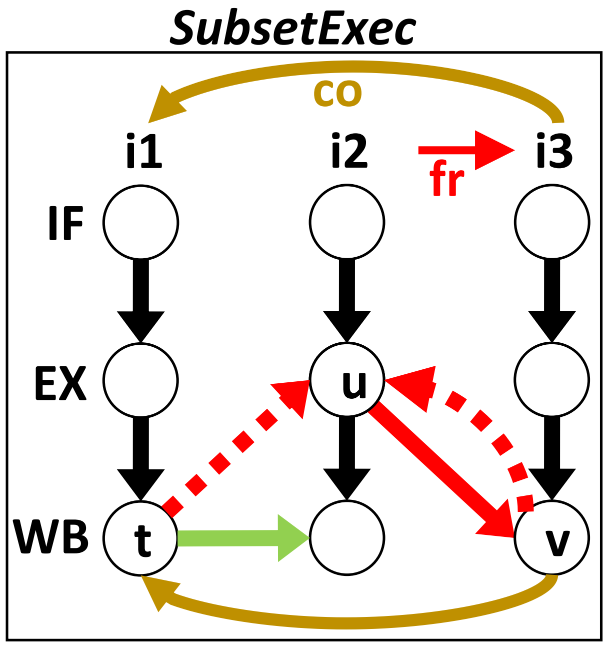


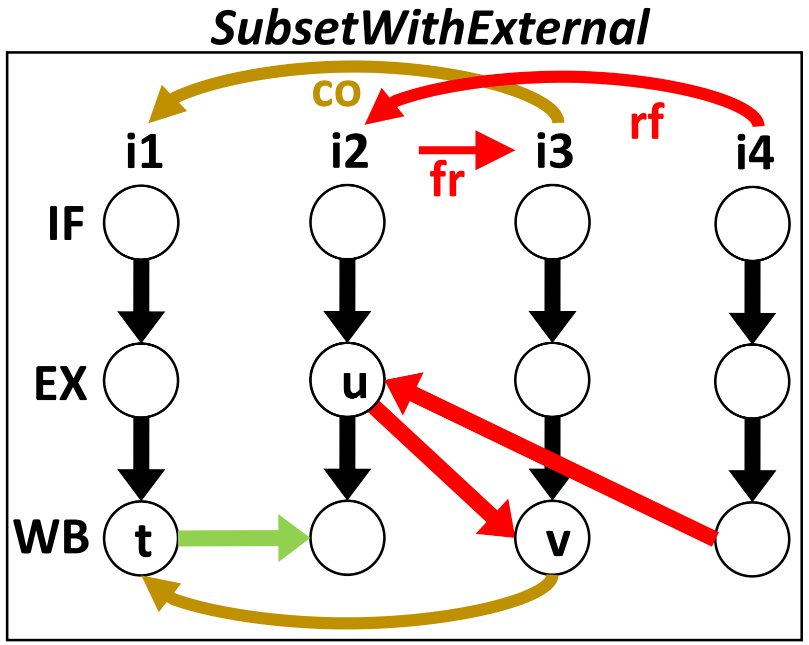
Same cycle is checked 3 times!

<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

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


SubsetExec
co
i1
i2
fr
i3
IF
EX
u
WB
t
v


SubsetWithExternal
co
rf
i1
i2
fr
i3
i4
IF
EX
u
WB
t
v