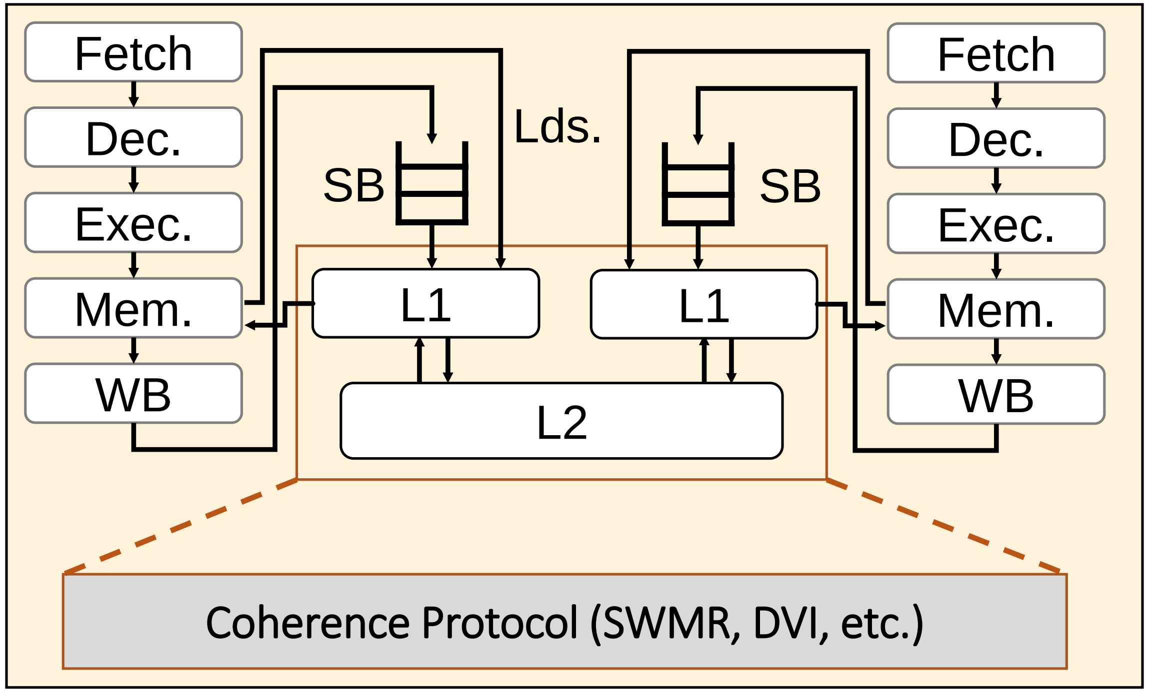
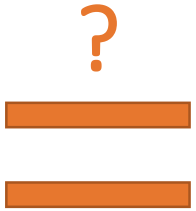
PipeProof (including hands-on):

Verifying simpleSC across all programs

Does hardware correctly implement ISA MCM?

Microarchitecture



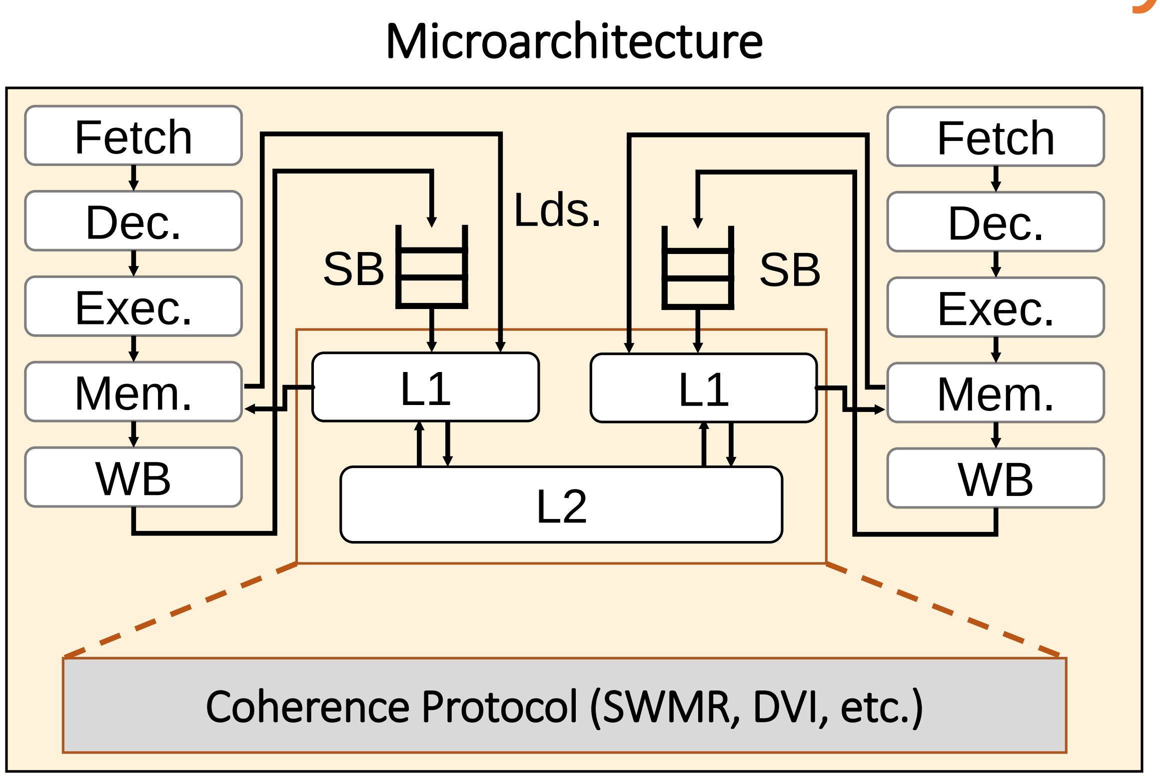


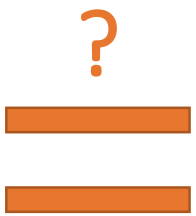
SC/TSO/RISC-V MCM?
(for the litmus test)

Litmus Test

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

Does hardware correctly implement ISA MCM?





SC/TSO/RISC-V MCM?

PipeCheck vs PipeProof

- PipeCheck:

- PipeProof:

Why do we need PipeProof?

- Test-based verification only checks that tested programs run correctly!
- **Open question:** Does a suite of litmus tests cover all µarch bugs?
- **Example:** Remove `EnforceWritePPO` axiom from simpleSC
  - `/home/check/pipecheck_tutorial/uarches/SC_fillable.uarch`
  - Some orderings between same-core stores and loads removed, **violating SC**
  - Will bug be detected? **Depends what tests you run!**

```
Axiom "EnforceWritePPO":
  forall microop "w",
  forall microop "i",
  (IsAnyWrite w /\ SameCore w i
     /\ EdgeExists((w, Fetch), (i, Fetch), "")) =>
          AddEdge ((w, Writeback), (i, Execute)).
```

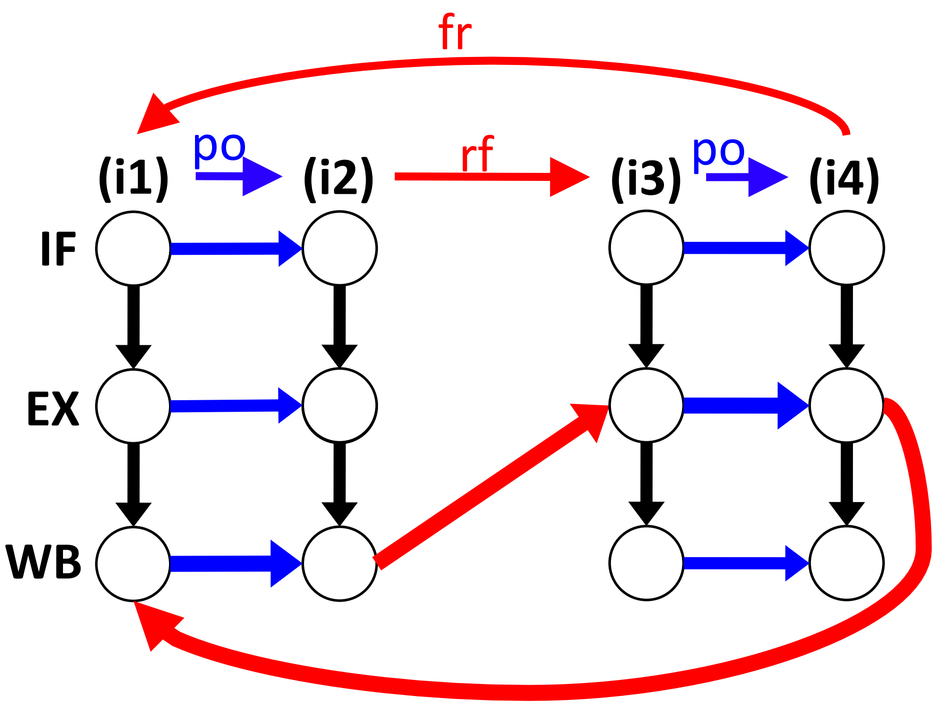
SimpleSC without EnforceWritePPO

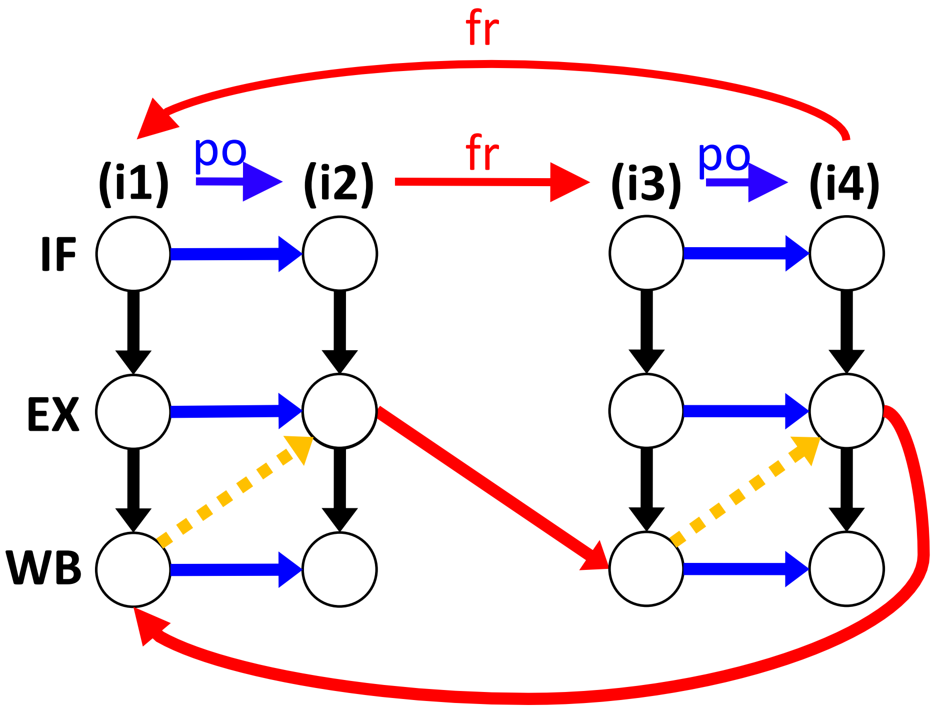
mp Litmus Test

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

sb Litmus Test

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





Cyclic => Still unobservable

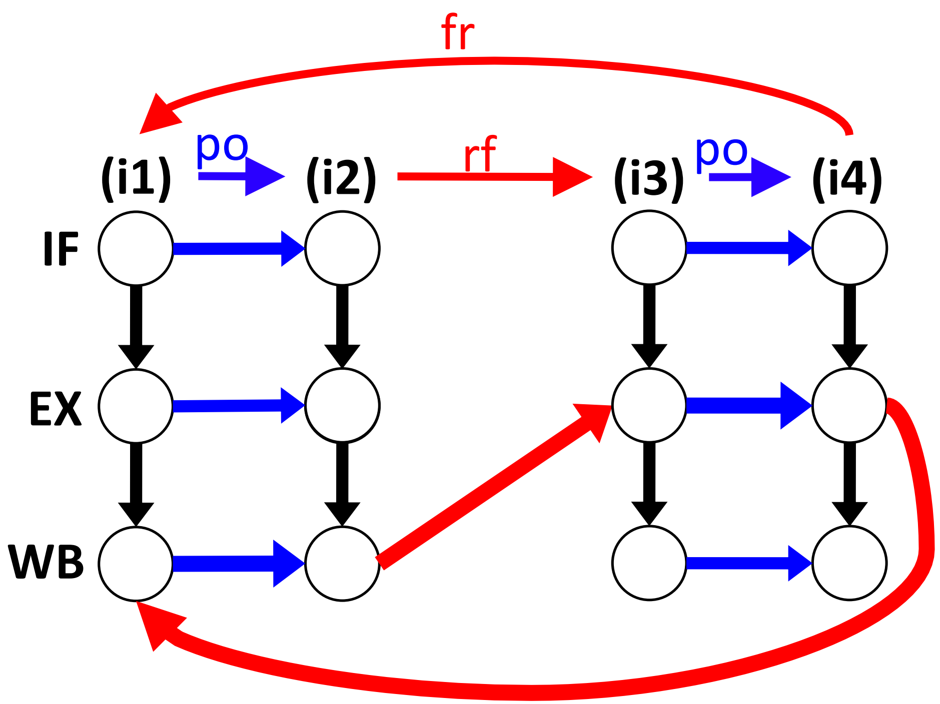
SimpleSC without `EnforceWritePPO`

mp Litmus Test

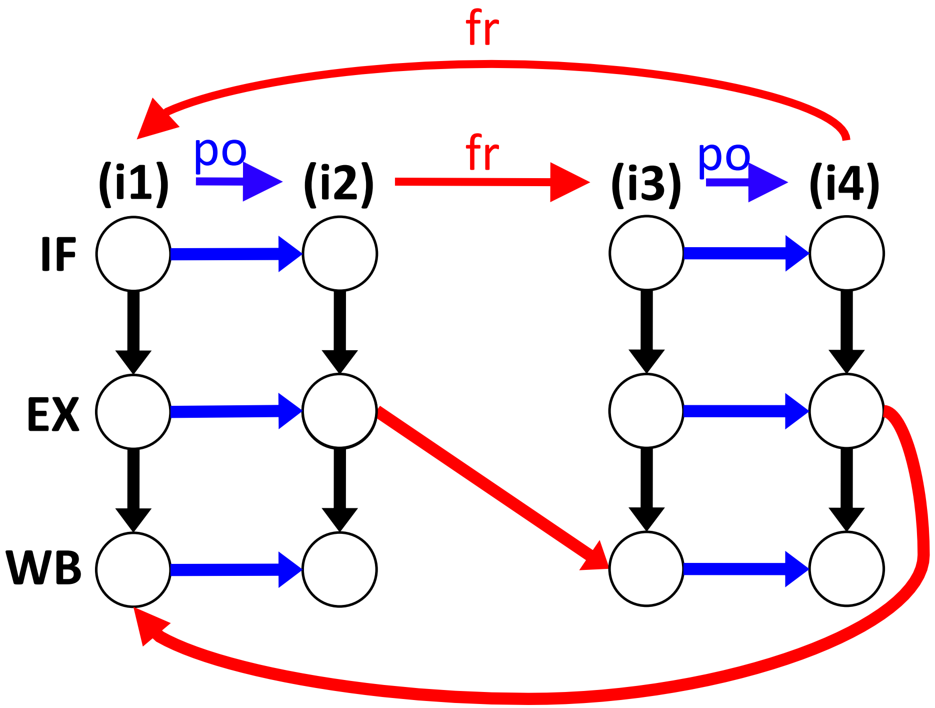
Core 0	Core 1
`x = 1;` `y = 1;`	`r1 = y;` `r2 = x;`
`Forbid: r1 = 1, r2 = 0`	

sb Litmus Test

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



Cyclic => Still unobservable



Acyclic => BUG!

SimpleSC without EnforceWritePPO

mp Litmus Test

Core 0	Core 1
x = 1; y = 1;	r1 = y; r2 = x;

sb Litmus Test

Core 0	Core 1
x = 1; r1 = y;	y = 1; r2 = x;

Different tests catch different bugs!

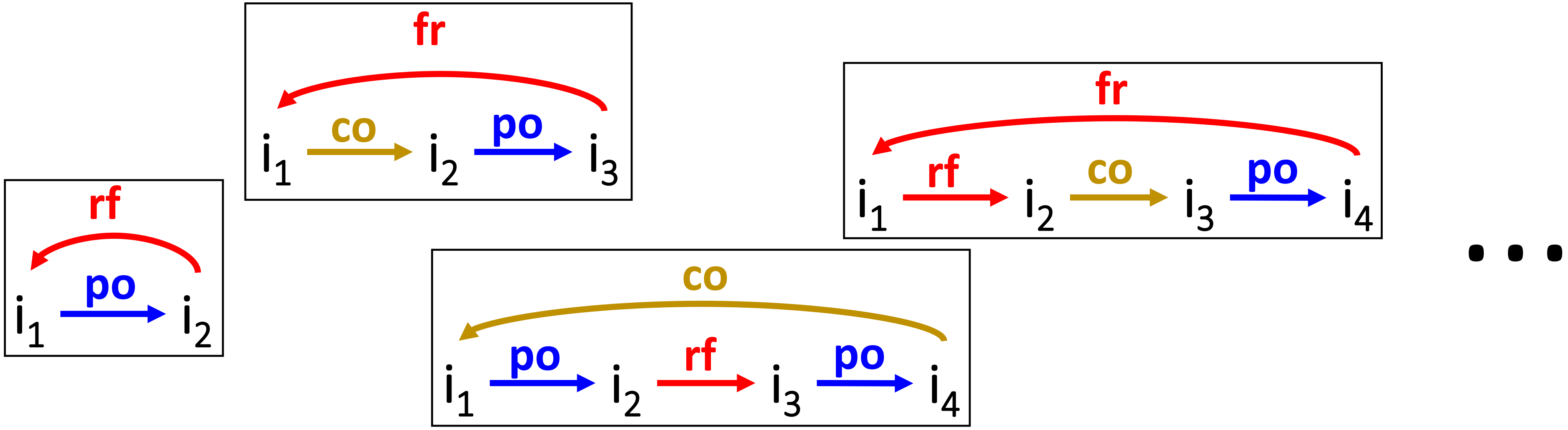
<u>To catch all bugs, must verify across all programs!</u>

Cyclic => Still unobservable

Acyclic => BUG!

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 $cyclic(po \cup co \cup rf \cup fr)$
- How are these ISA-level patterns related to litmus tests?



Symbolic Analysis: Generalise to ISA-Level Cycles

- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

mp

Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [y] ← 1	i3: r1 = Load [y] i4: r2 = Load [x]
SC Forbids: r1=1, r2=0	

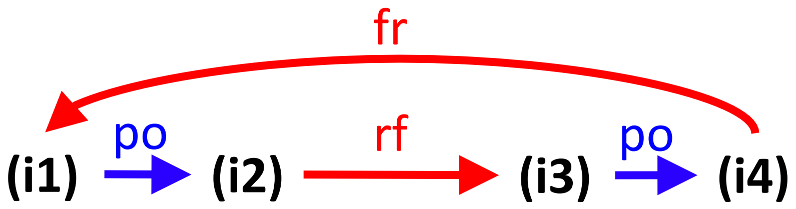
Symbolic Analysis: Generalise to ISA-Level Cycles

- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

mp

Thread 0	Thread 1
i1: Store [x] ← 1	i3: r1 = Load [y]
po ↓	po ↓
i2: Store [y] ← 1	i4: r2 = Load [x]
SC Forbids: r1=1, r2=0	

(rf: i2 → i3; fr: i4 → i1)



Symbolic Analysis: Generalise to ISA-Level Cycles

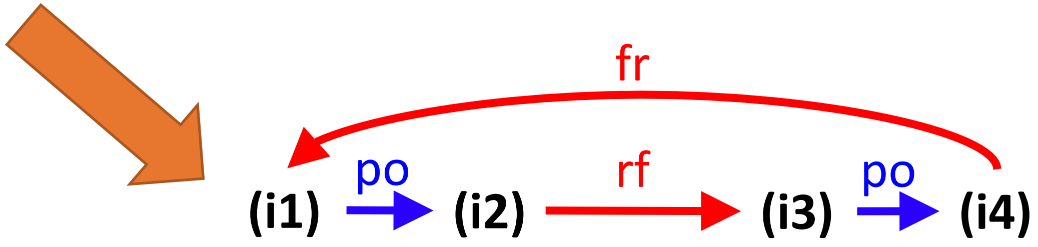
- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

mp

Thread 0	Thread 1
i1: Store [x] ← 1	i3: r1 = Load [y]
i2: Store [y] ← 1	i4: r2 = Load [x]
SC Forbids: r1=1, r2=0	

mp+changedVals

Thread 0	Thread 1
i1: Store [x] ← 1	i3: r1 = Load [z]
i2: Store [z] ← 2	i4: r2 = Load [x]
SC Forbids: r1=2, r2=0	



Symbolic Analysis: Generalise to ISA-Level Cycles

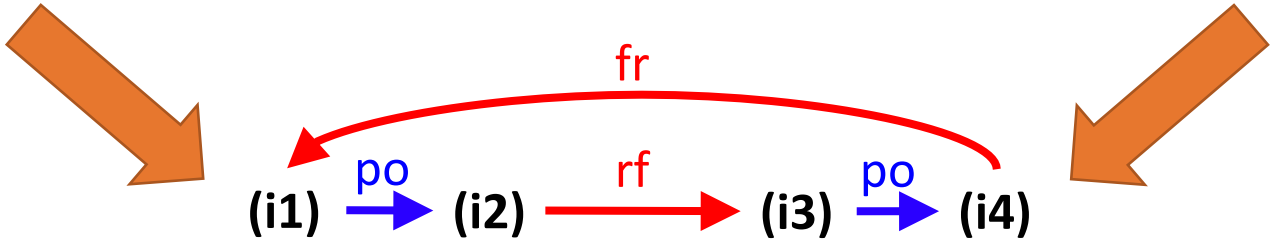
- Each forbidden litmus test is an **instance** of an ISA-level cycle
- PipeProof verifies the ISA-level cycles rather than litmus tests
  - Instructions in the ISA-level cycle are **symbolic** (no concrete addresses/values)
  - Verification of ISA-level cycle checks it for all possible addresses/values!

mp

Thread 0	Thread 1
i1: Store [x] ← 1	i3: r1 = Load [y]
i2: Store [y] ← 1	i4: r2 = Load [x]
SC Forbids: r1=1, r2=0	

mp+changedVals

Thread 0	Thread 1
i1: Store [x] ← 1	i3: r1 = Load [z]
i2: Store [z] ← 2	i4: r2 = Load [x]
SC Forbids: r1=2, r2=0	

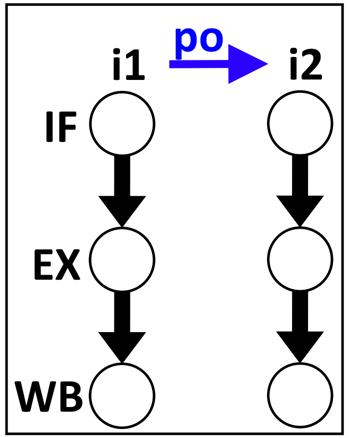


PipeProof: What's Needed

1. Link ISA-level MCM to microarchitectural specification
   - **ISA Edge Mapping**
2. Add universal constraints that symbolic analysis must respect
   - **Theory Lemmas**
3. A finite representation of all forbidden ISA-level cycles
   - **Transitive Chain (TC) Abstraction**
4. Automated refinement checking of the finite representation
   - **Microarchitectural Correctness Proof**
   - **Chain invariants** (for termination)

Mapping ISA-Level Edges to Microarchitecture

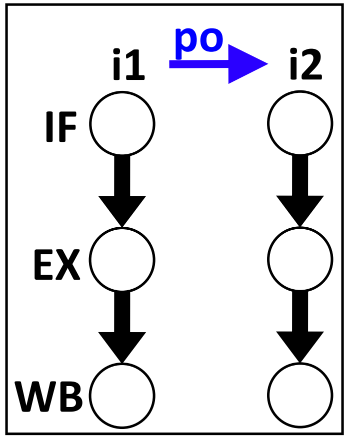
- Open **`/home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch`**
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of po edges




```
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

- Open **`/home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch`**
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of po edges

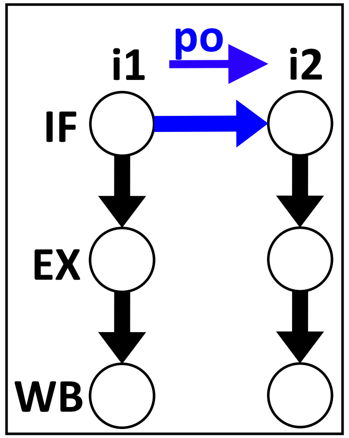



```
Axiom "Mapping_po":
forall microop "i",
forall microop "j",
(HasDependency po i j) =>
     AddEdge ((i, Fetch), (j, Fetch), "po_arch", "blue")).
```

Check whether a po edge from i to j exists

Mapping ISA-Level Edges to Microarchitecture

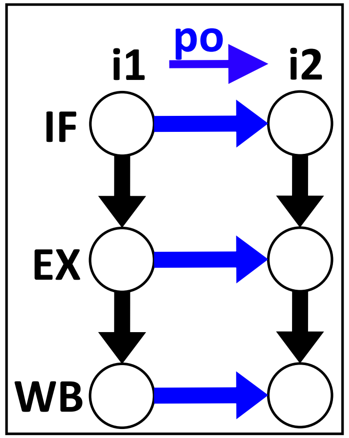
- Open **`/home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch`**
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of *po* edges




```
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

- Open **`/home/check/pipeproof_tutorial/uarches/simpleSC_fill.uarch`**
- Translate each edge in ISA-level cycle to microarchitectural constraints
- Do so with user-provided **Mapping Axioms**
- Example: Mapping of *po* edges

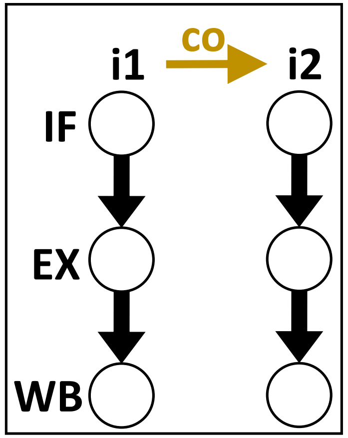


Blue edges between EX and WB stages added by other FIFO axioms (refer to µspec file)

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

Mapping Axioms Hands-on

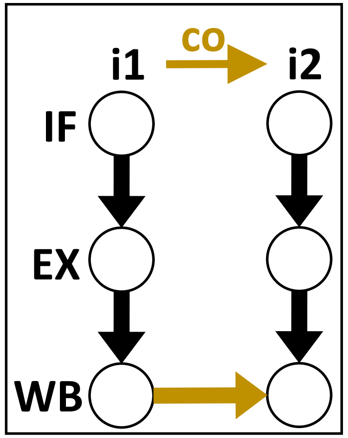
- How about mapping *co* (coherence order) edges?
- Hint:
  - *po* edge mapping was similar to `PO_Fetch` axiom
  - *co* edge mapping is based on `WriteSerialization` axiom




```
Axiom "Mapping_co":
forall microop "i",
forall microop "j",
(HasDependency co i j => SamePhysicalAddress i j /\
     AddEdge ((i, __________), (j, __________), "co_arch")).
```

Mapping Axioms Hands-on

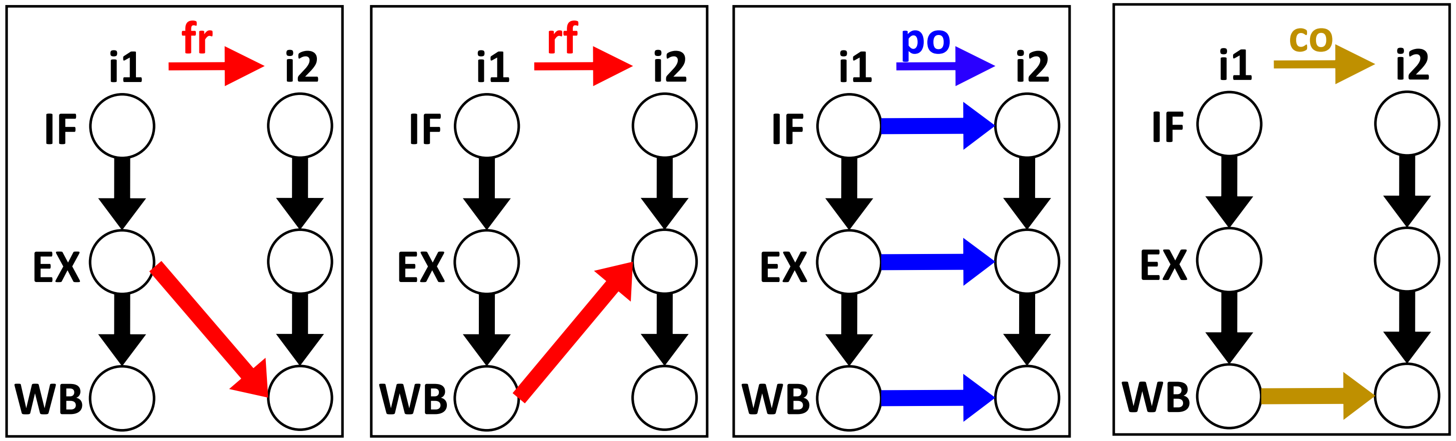
- How about mapping *co* (coherence order) edges?
- Hint:
  - *po* edge mapping was similar to `PO_Fetch` axiom
  - *co* edge mapping is based on `WriteSerialization` axiom




```
Axiom "Mapping_co":
forall microop "i",
forall microop "j",
(HasDependency co i j => SamePhysicalAddress i j /\
     AddEdge ((i, Writeback), (j, Writeback), "co_arch")).
```

ISA Edge Mappings for SimpleSC

- Refer to `simpleSC_fill.uarch` to see mapping axioms for rf, fr



PipeProof: What's Needed

1. Link ISA-level MCM to microarchitectural specification
   - **ISA Edge Mapping**
2. Add universal constraints that symbolic analysis must respect
   - **Theory Lemmas**
3. A finite representation of all forbidden ISA-level cycles
   - **Transitive Chain (TC) Abstraction**
4. Automated refinement checking of the finite representation
   - **Microarchitectural Correctness Proof**
   - **Chain invariants** (for termination)

Symbolic Analysis Requires Theory Lemmas

- Symbolic analysis: predicates are just variables that can be true or false
  - "Theory Lemmas" necessary to enforce "universal" laws on predicates
- **Example:** Is an instruction guaranteed to be a read or write?

```
i: r1 = Load [x]
```

Concrete: Look at instruction -> **IsAnyRead i is true**

Symbolic Analysis Requires Theory Lemmas

- Symbolic analysis: predicates are just variables that can be true or false
  - "Theory Lemmas" necessary to enforce "universal" laws on predicates
- **Example:** Is an instruction guaranteed to be a read or write?

i

Concrete: Look at instruction -> **IsAnyRead i is true**

Symbolic: We now know nothing about the instruction!

Both **IsAnyRead i** and **IsAnyWrite i** could be false! (even though this can't happen in reality)

Symbolic Analysis Requires Theory Lemmas

- Symbolic analysis: predicates are just variables that can be true or false
  - "Theory Lemmas" necessary to enforce "universal" laws on predicates
- **Example:** Is an instruction guaranteed to be a read or write?

```
i
```

Concrete: Look at instruction -> **IsAnyRead i is true**

Symbolic: We now know nothing about the instruction!

Both **IsAnyRead i** and **IsAnyWrite i** could be false! (even though this can't happen in reality)

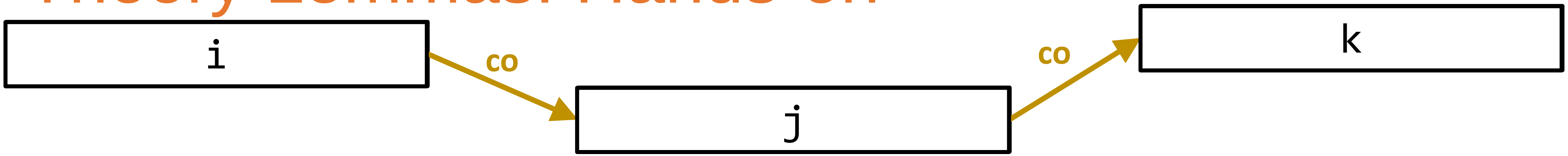
Need Additional Theory Lemma to enforce that op is either a read or write!

```
Axiom “Theory_Lemmas":
forall microop "i",
...
IsAnyRead i \/ IsAnyWrite i).
```

Theory Lemmas: Hands-on

Concrete: Directly compare instructions i and k -> **SamePhysicalAddress i k is true**

Theory Lemmas: Hands-on

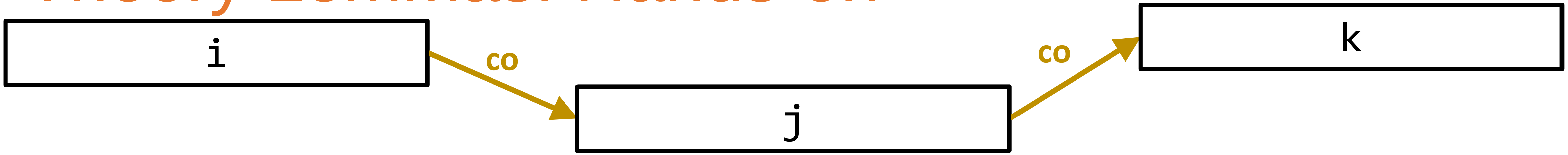


Concrete: Directly compare instructions i and k -> **SamePhysicalAddress i k is true**

Symbolic: co edge mapping gives **SamePhysicalAddress i j** and **SamePhysicalAddress j k**

But **SamePhysicalAddress i k could be false!** (even though this can never happen in reality)

Theory Lemmas: Hands-on



Concrete: Directly compare instructions i and k -> **SamePhysicalAddress i k is true**

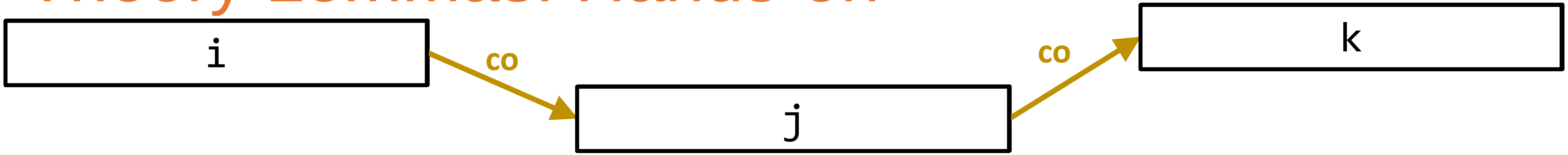
Symbolic: co edge mapping gives **SamePhysicalAddress i j** and **SamePhysicalAddress j k**

But **SamePhysicalAddress i k could be false!** (even though this can never happen in reality)

Need Additional Theory Lemma for Transitivity of SamePhysicalAddress!

```
Axiom “Theory_Lemmas":
forall microop "i",
...
forall microop "j",
...
forall microop “k”,
(SamePhysicalAddress _ _ /\ SamePhysicalAddress _ _ =>
SamePhysicalAddress _ _)...
```

Theory Lemmas: Hands-on



Concrete: Directly compare instructions i and k -> **SamePhysicalAddress i k is true**

Symbolic: co edge mapping gives **SamePhysicalAddress i j** and **SamePhysicalAddress j k**

But **SamePhysicalAddress i k could be false!** (even though this can never happen in reality)

Need Additional Theory Lemma for Transitivity of SamePhysicalAddress!

```
Axiom “Theory_Lemmas":
forall microop "i",
...
forall microop "j",
...
forall microop “k”,
(SamePhysicalAddress i j /\ SamePhysicalAddress j k =>
SamePhysicalAddress i k)...
```

PipeProof: What's Needed

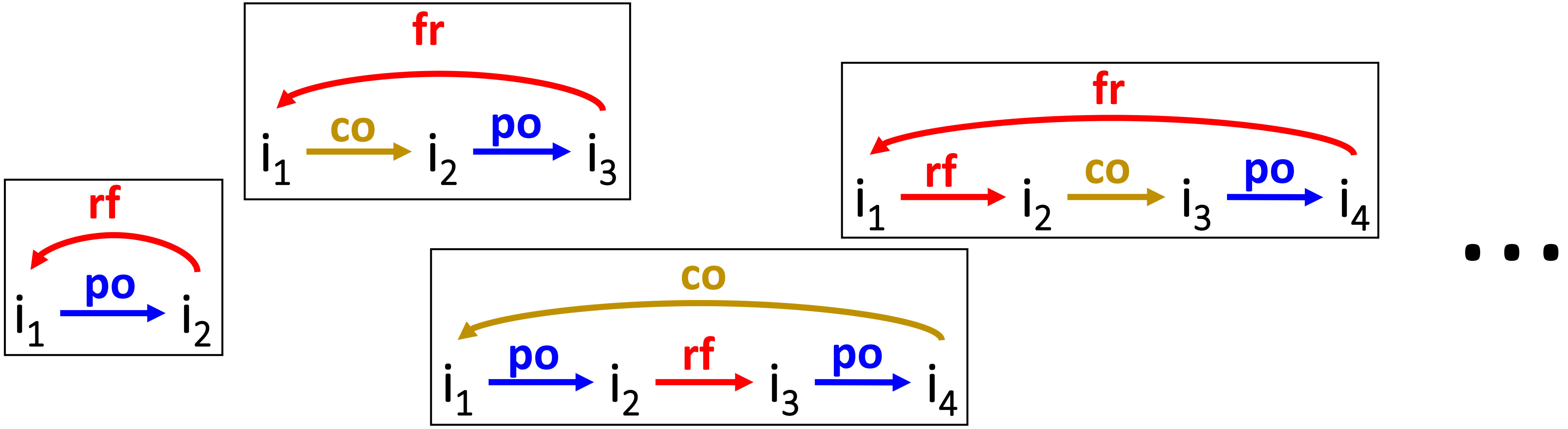
1. Link ISA-level MCM to microarchitectural specification
   - **ISA Edge Mapping**
2. Add universal constraints that symbolic analysis must respect
   - **Theory Lemmas**
3. A finite representation of all forbidden ISA-level cycles
   - **Transitive Chain (TC) Abstraction**
4. Automated refinement checking of the finite representation
   - **Microarchitectural Correctness Proof**
   - **Chain invariants** (for termination)

Verifying Across All Possible Programs

- **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]

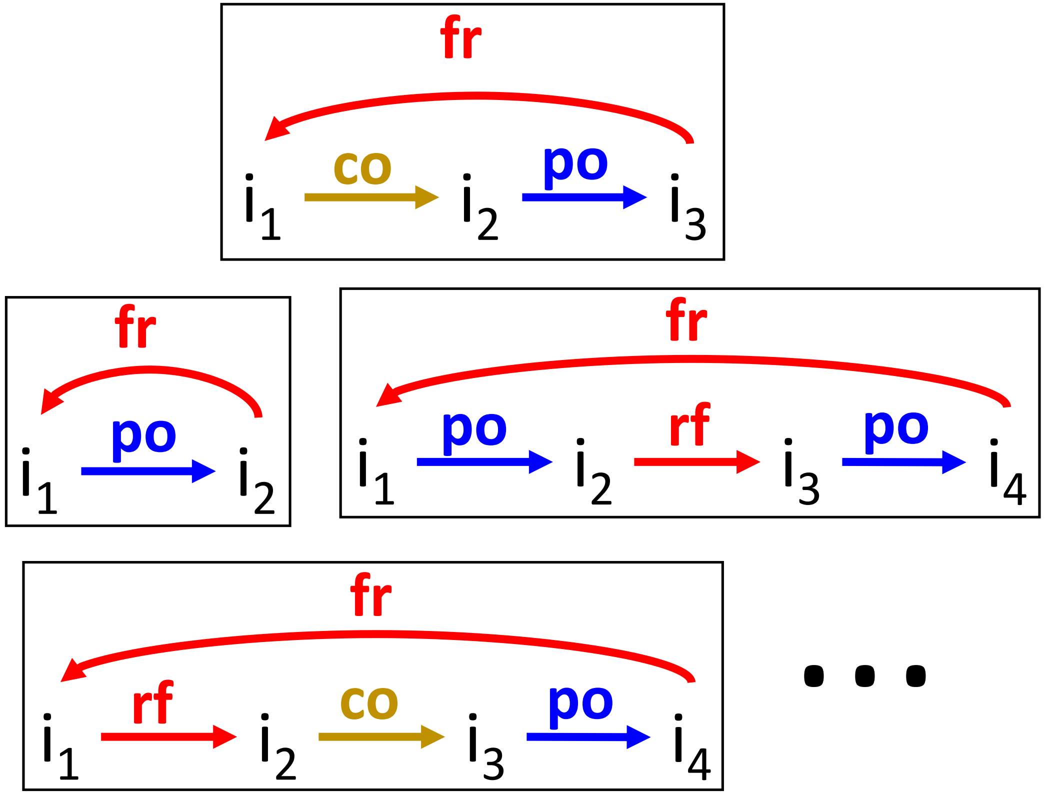
Verifying Across All Possible Programs

- **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]



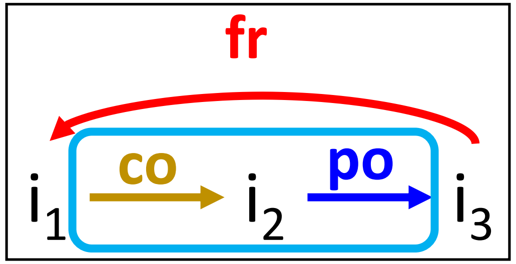
The Transitive Chain (TC) Abstraction

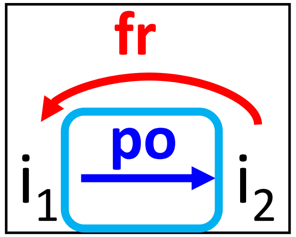
All non-unary cycles containing **fr**
(**Infinite set**)

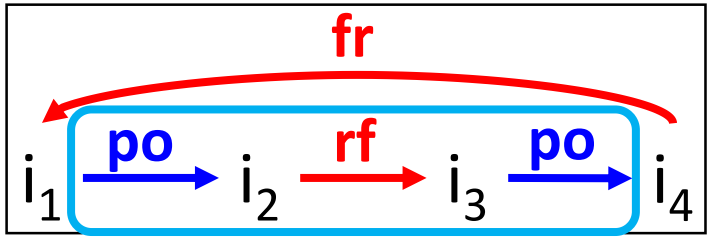


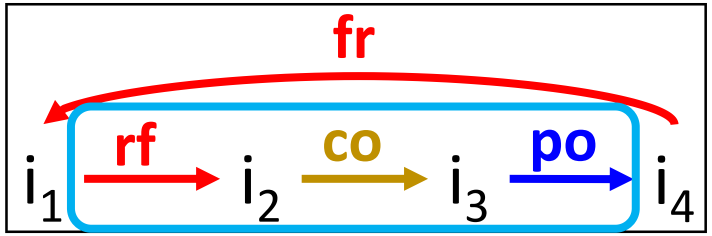
The Transitive Chain (TC) Abstraction

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







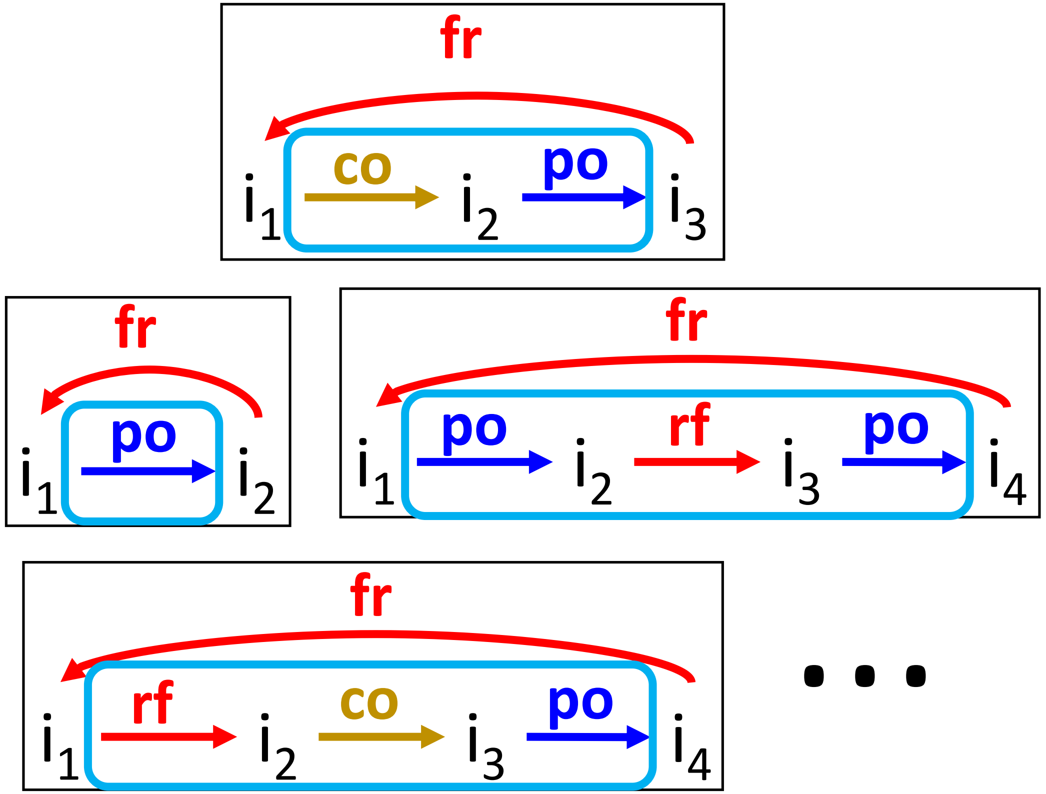


...

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

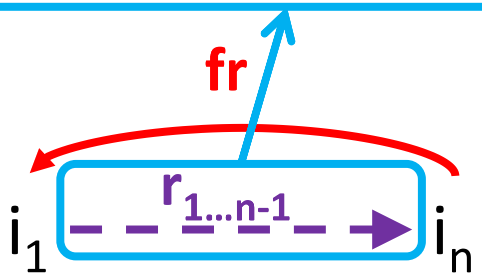
The Transitive Chain (TC) Abstraction

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



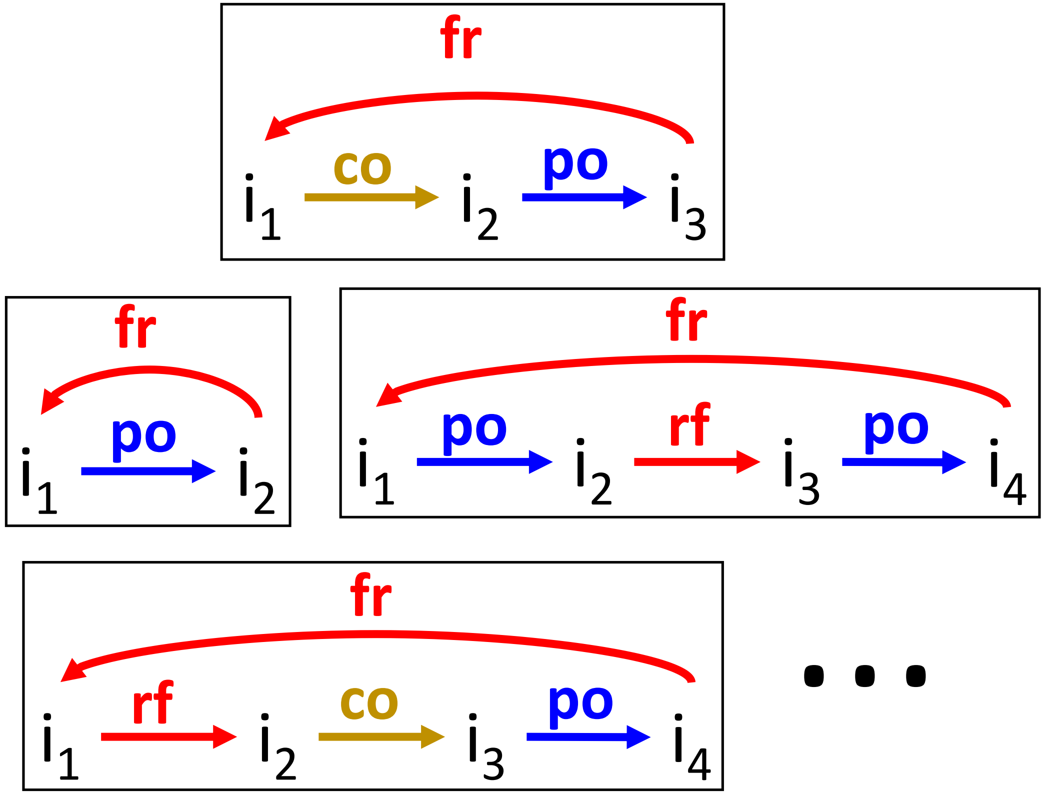
Cycle = Transitive Chain (sequence)
+ Loopback edge (fr)

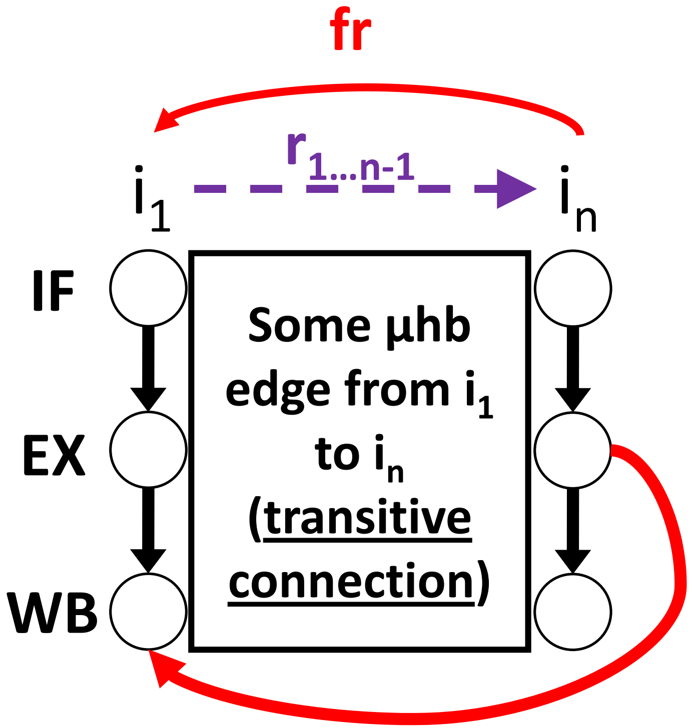
Transitive chain (sequence)
of ISA-level edges



The Transitive Chain (TC) Abstraction

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



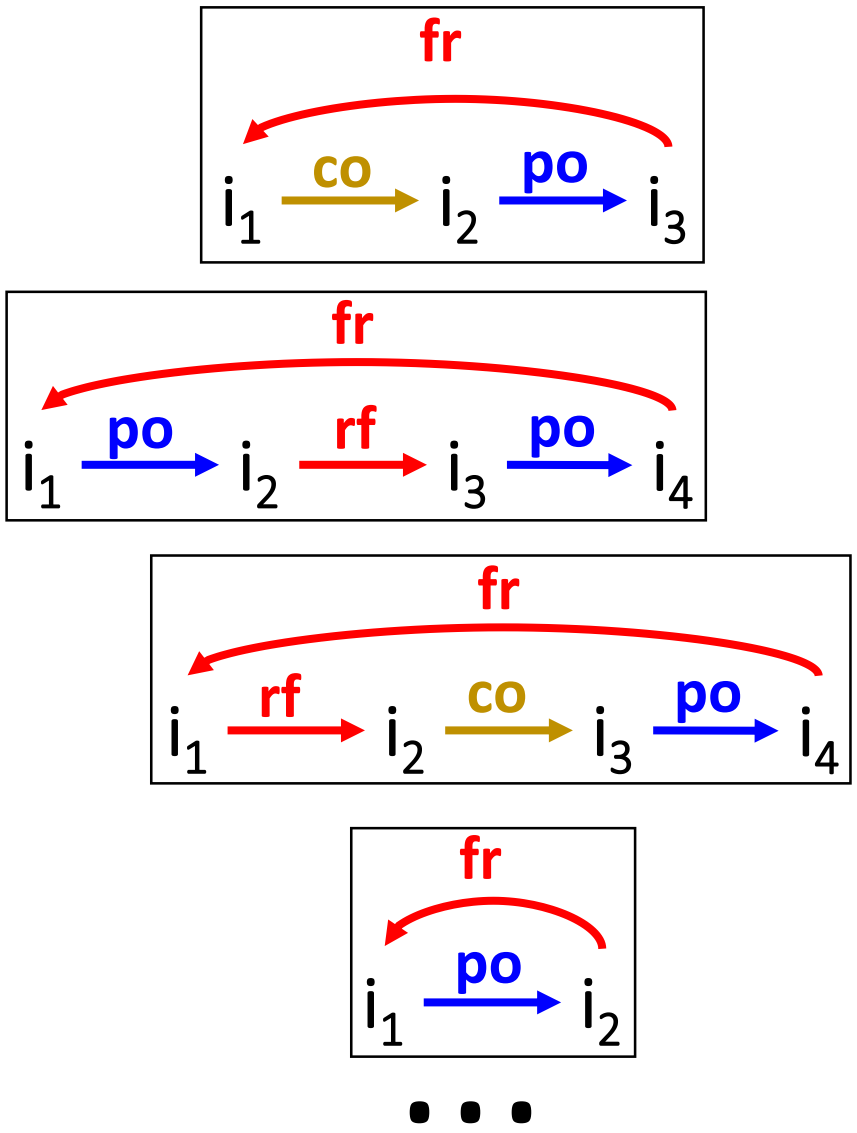


Cycle = Transitive Chain (sequence)
+ Loopback edge (fr)

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

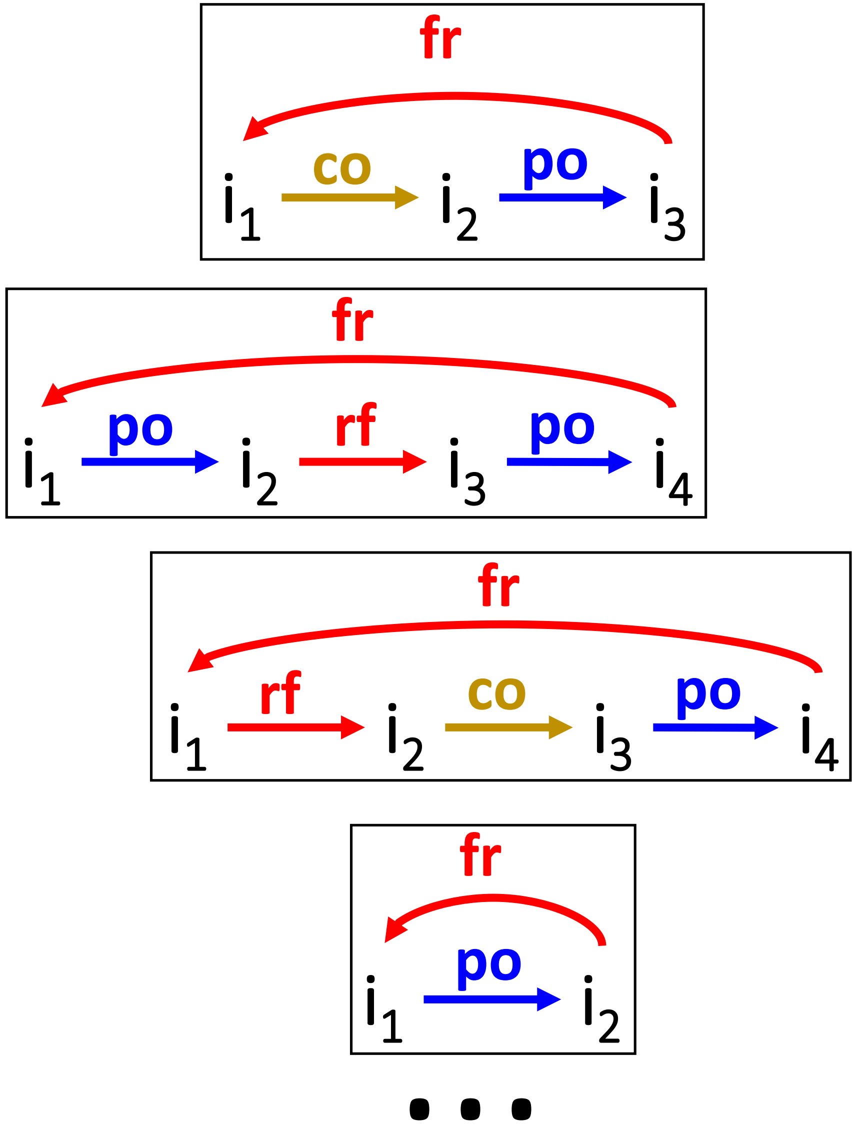
The Transitive Chain (TC) Abstraction

Infinite!



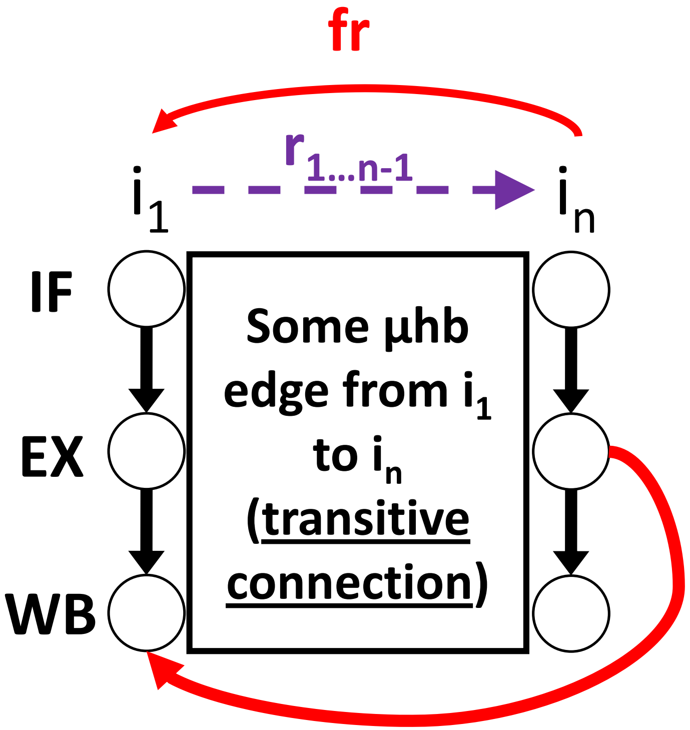
The Transitive Chain (TC) Abstraction

Infinite!



Using TC Abstraction ⟹

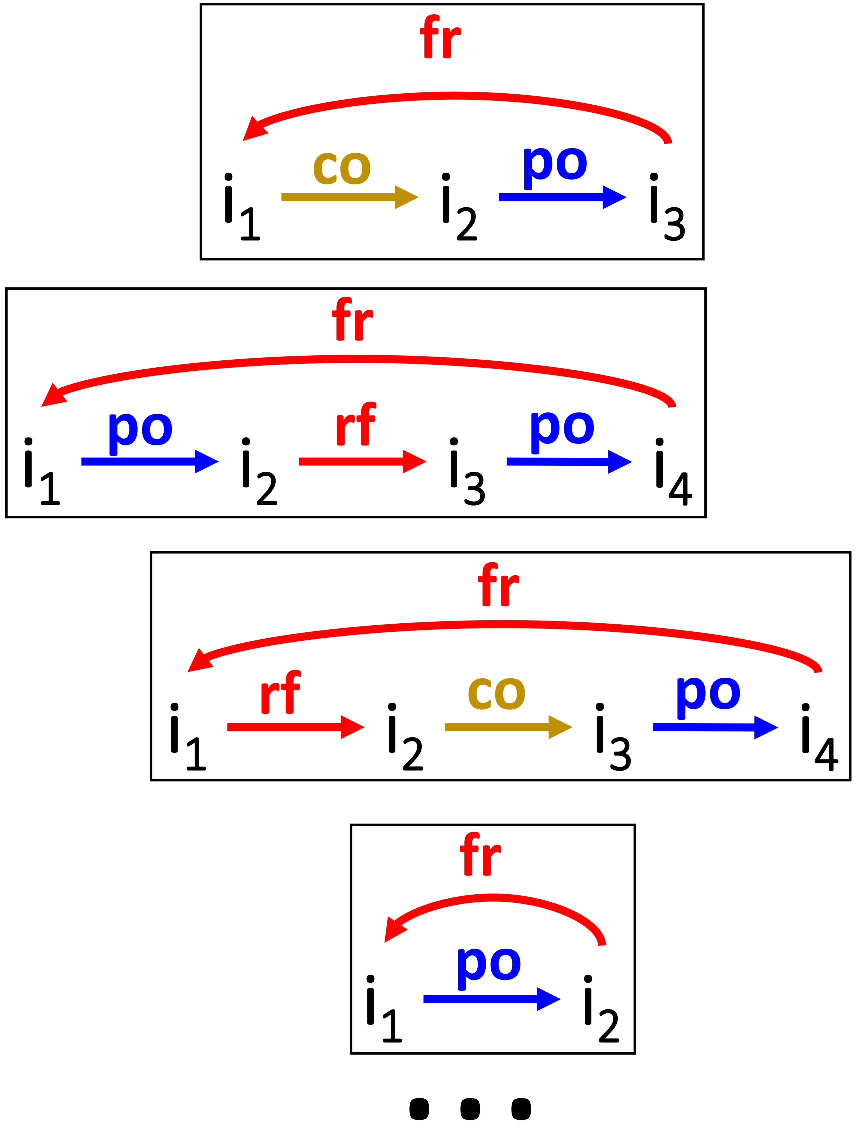
Finite!



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

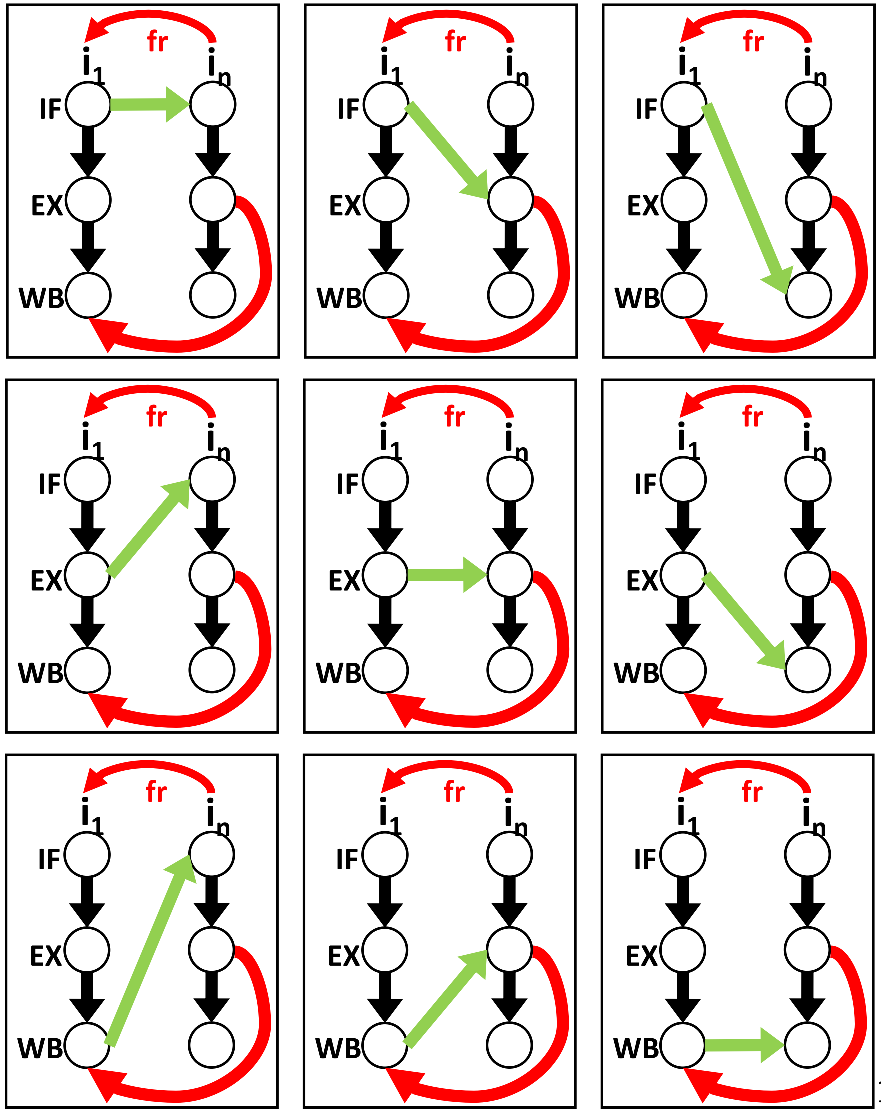
The Transitive Chain (TC) Abstraction

Infinite!



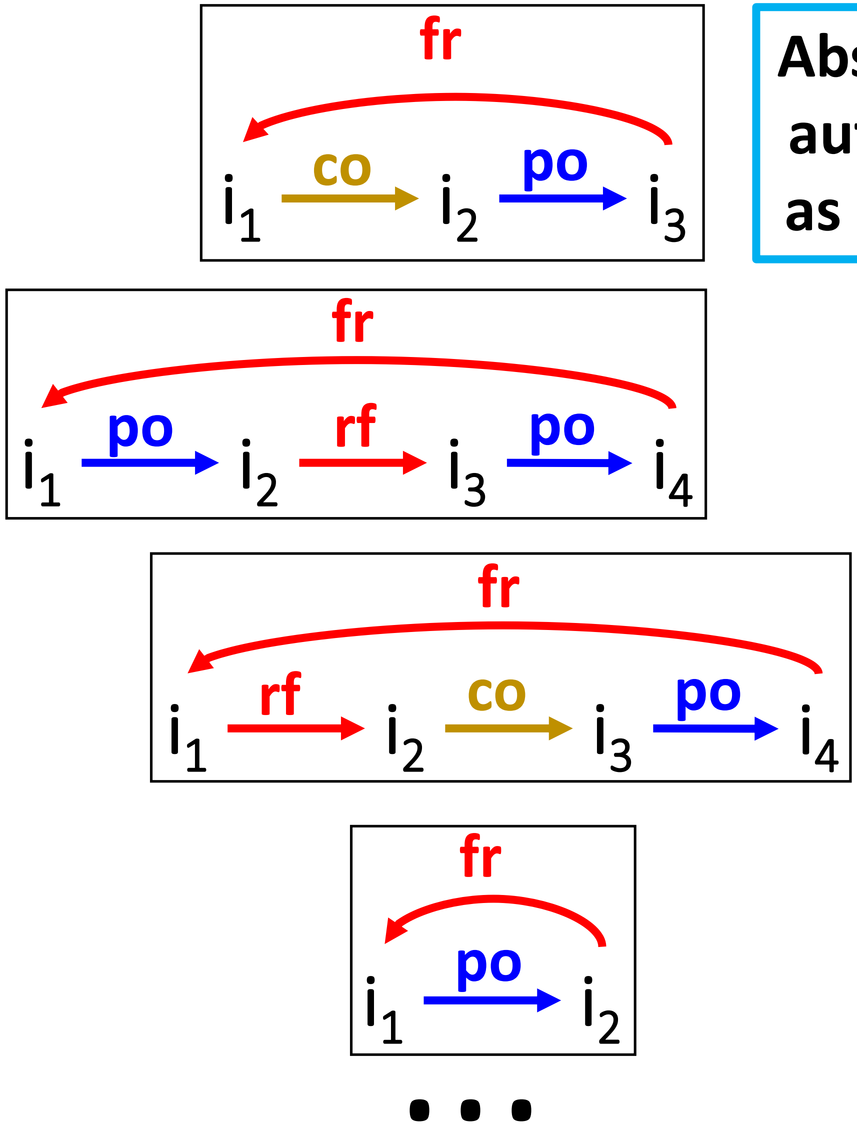
Using TC Abstraction ⟹

Finite!



The Transitive Chain (TC) Abstraction

Infinite!

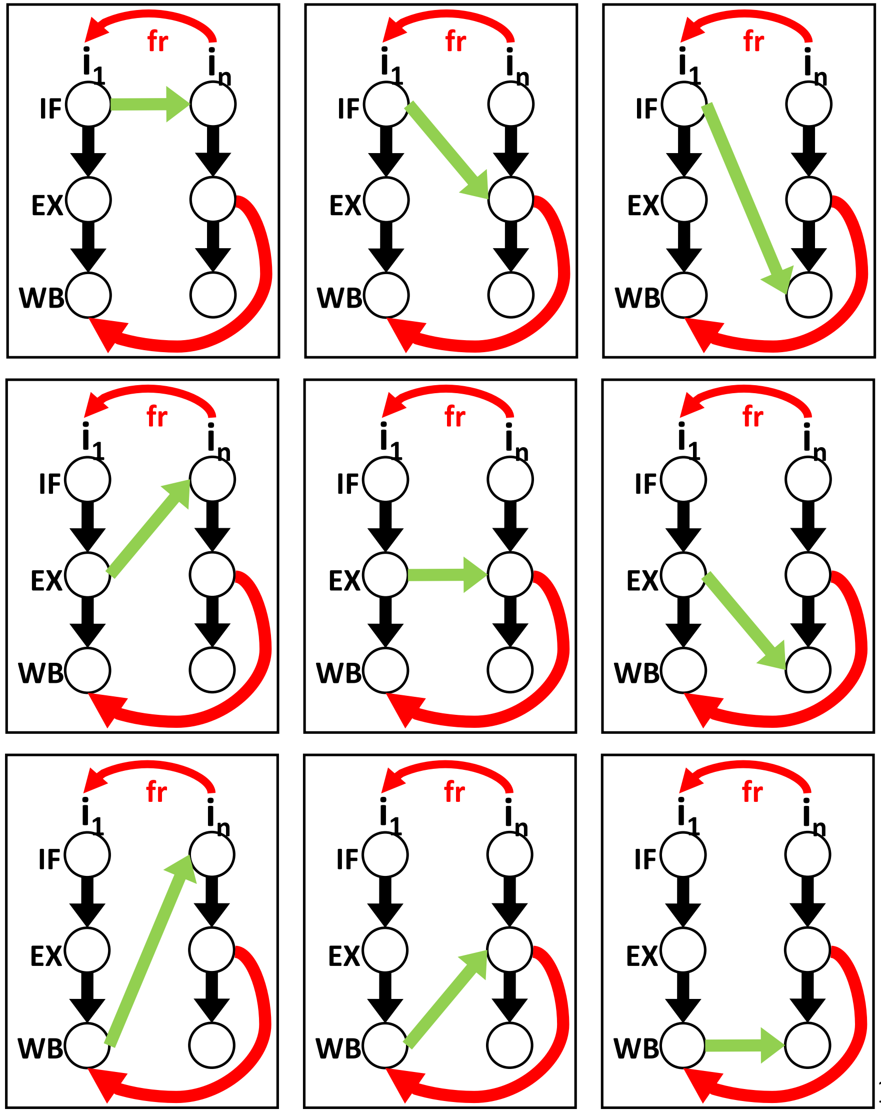


Abstraction soundness automatically verified as a supporting proof!

Using TC Abstraction

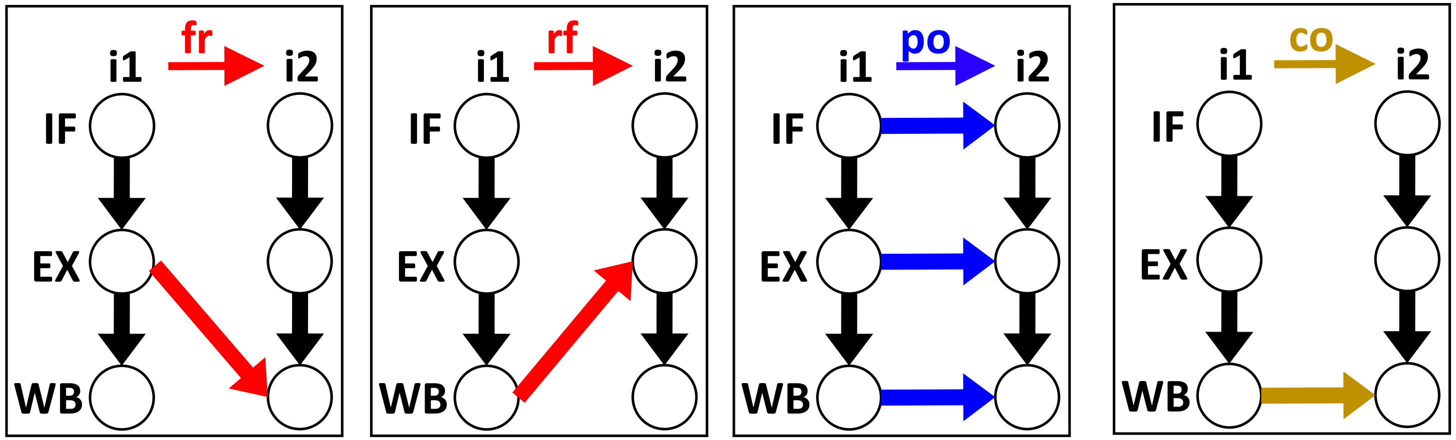
⟹

Finite!

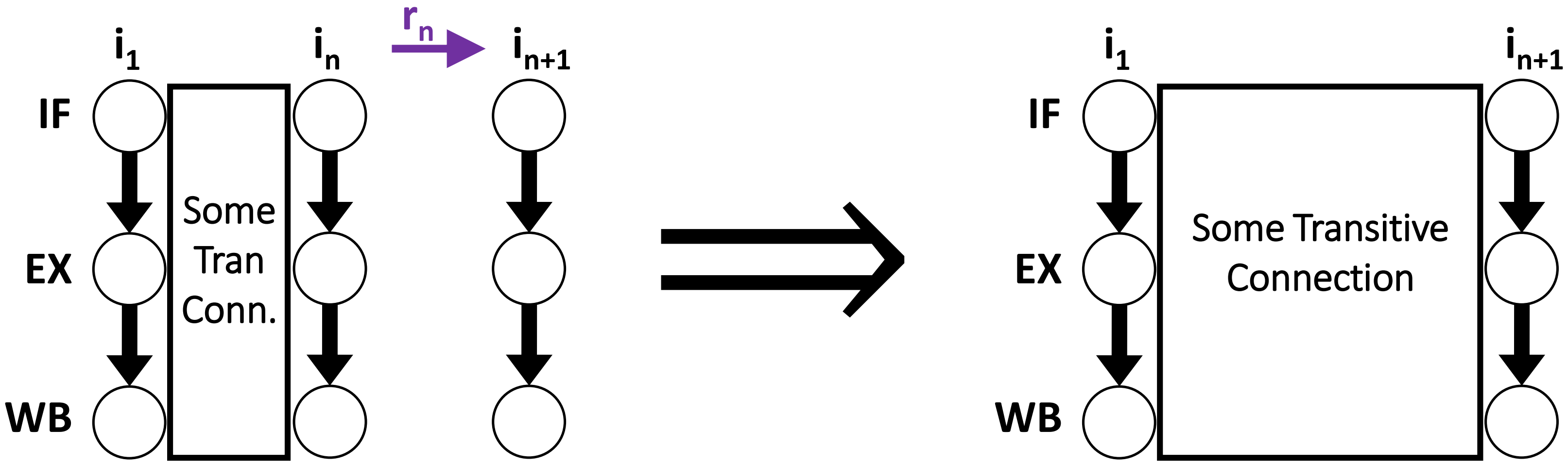


Transitive Chain (TC) Abstraction Support Proof

- 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?

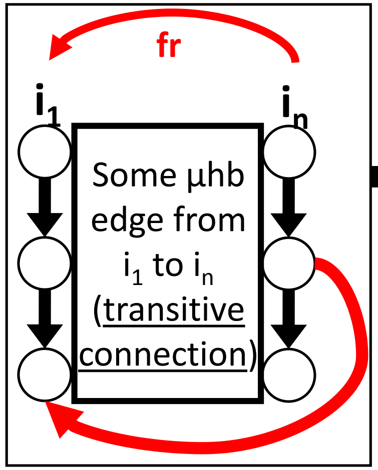


PipeProof: What's Needed

1. Link ISA-level MCM to microarchitectural specification
   - **ISA Edge Mapping**
2. Add universal constraints that symbolic analysis must respect
   - **Theory Lemmas**
3. A finite representation of all forbidden ISA-level cycles
   - **Transitive Chain (TC) Abstraction**
4. Automated refinement checking of the finite representation
   - **Microarchitectural Correctness Proof**
   - **Chain invariants** (for termination)

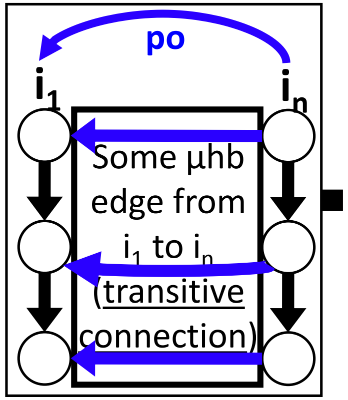
Microarchitectural Correctness Proof

Cycles containing *fr*



All possible transitive connections

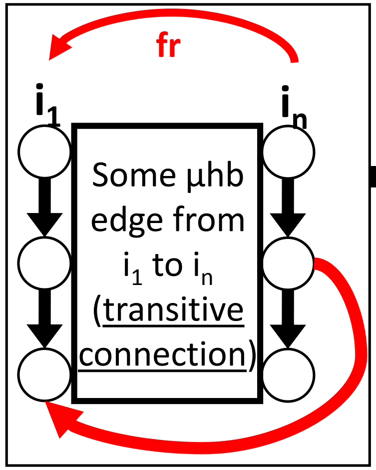
Cycles containing *po*



Other ISA-level cycles…

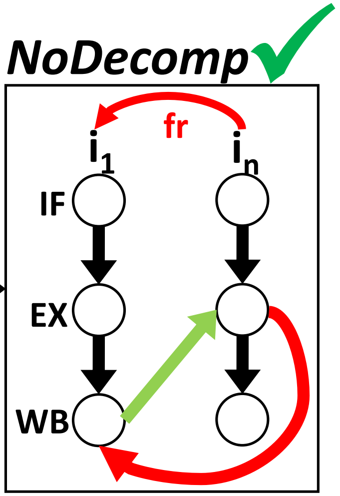
Microarchitectural Correctness Proof

Cycles containing *fr*

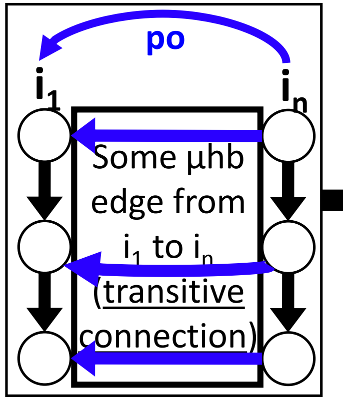


All possible transitive connections

NoDecomp



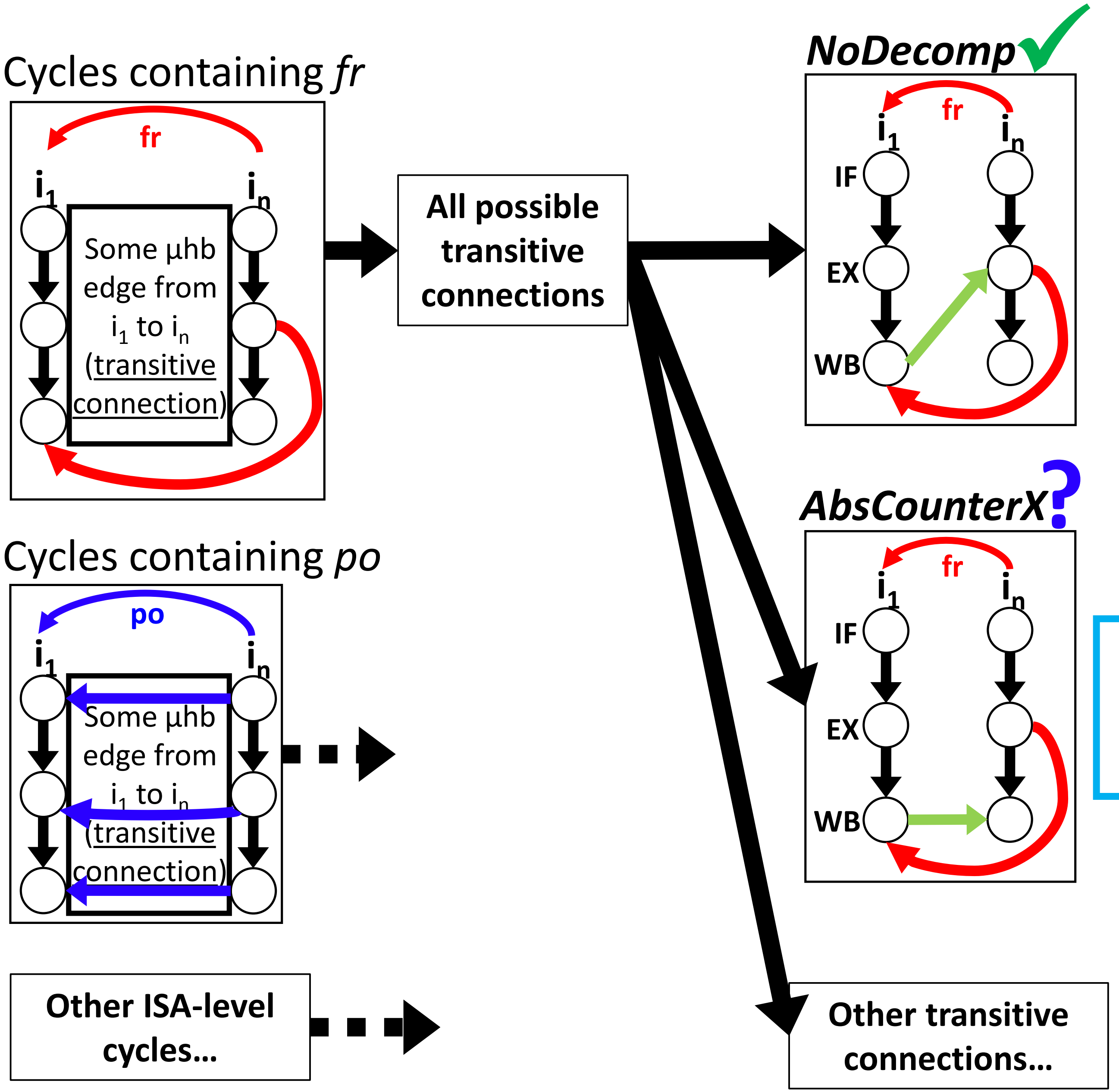
Cycles containing *po*



Other ISA-level cycles…

Other transitive connections…

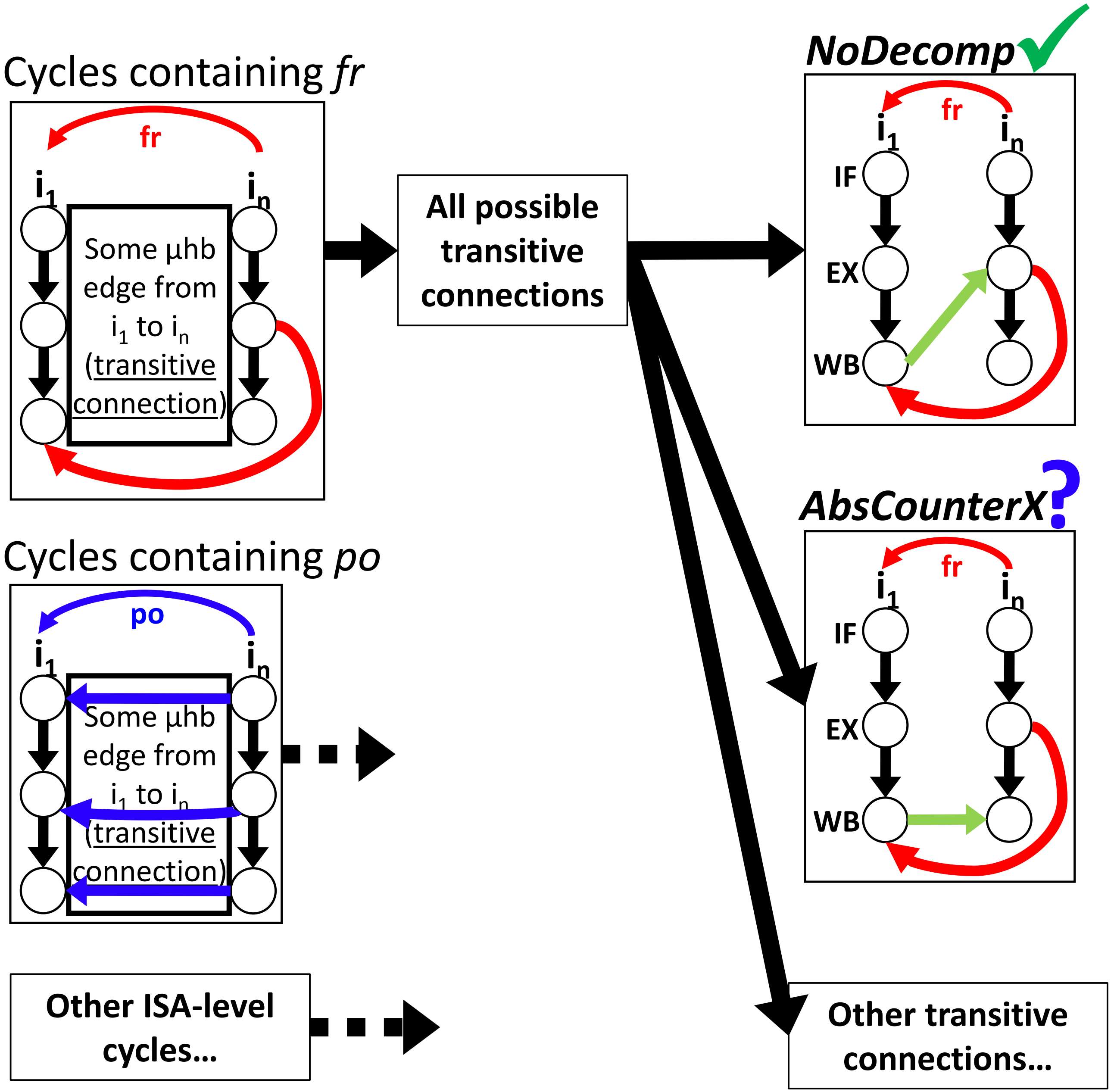
Microarchitectural Correctness Proof



Acyclic graph with transitive connection =>

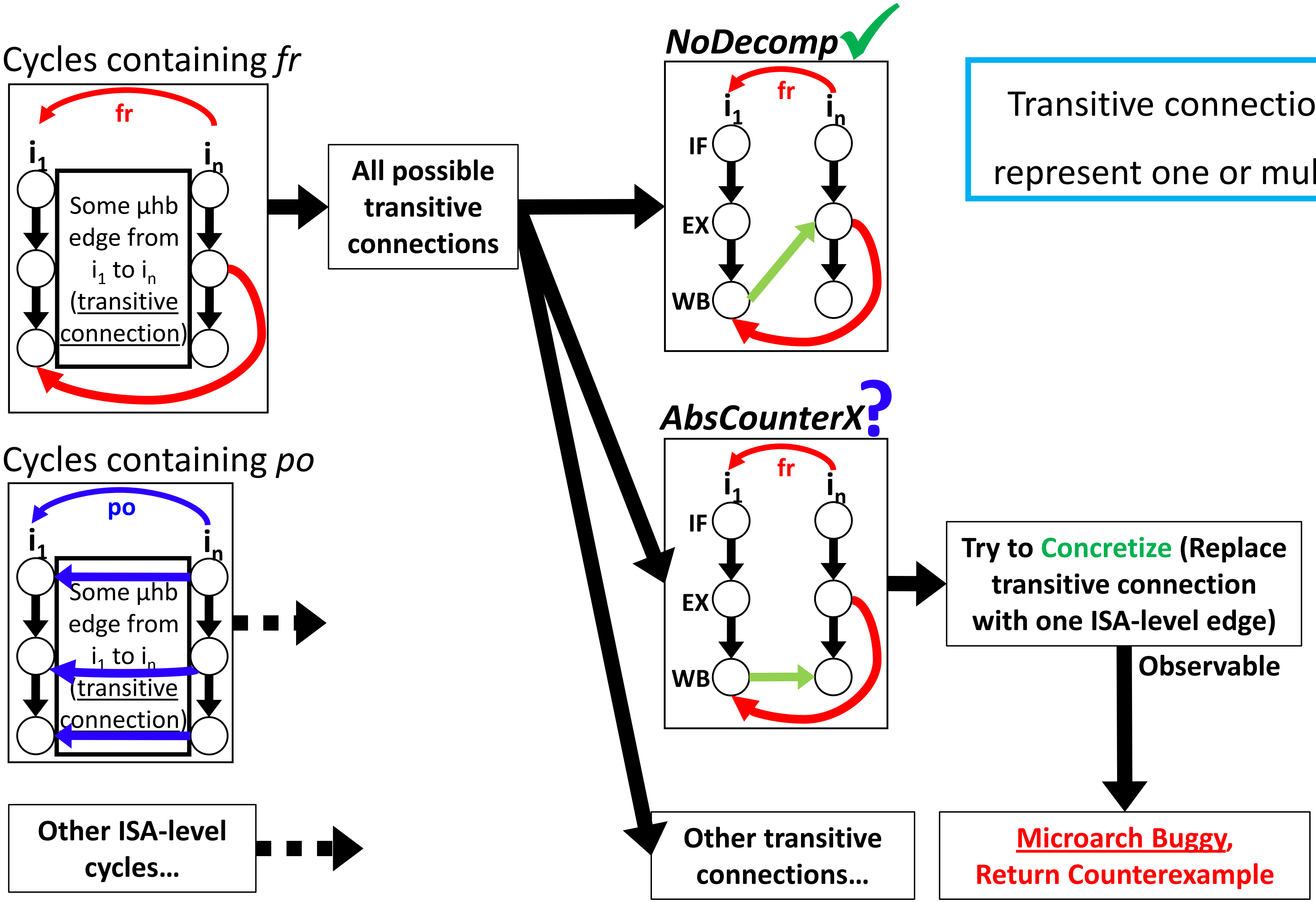
Abstract Counterexample (i.e. possible bug)

Microarchitectural Correctness Proof



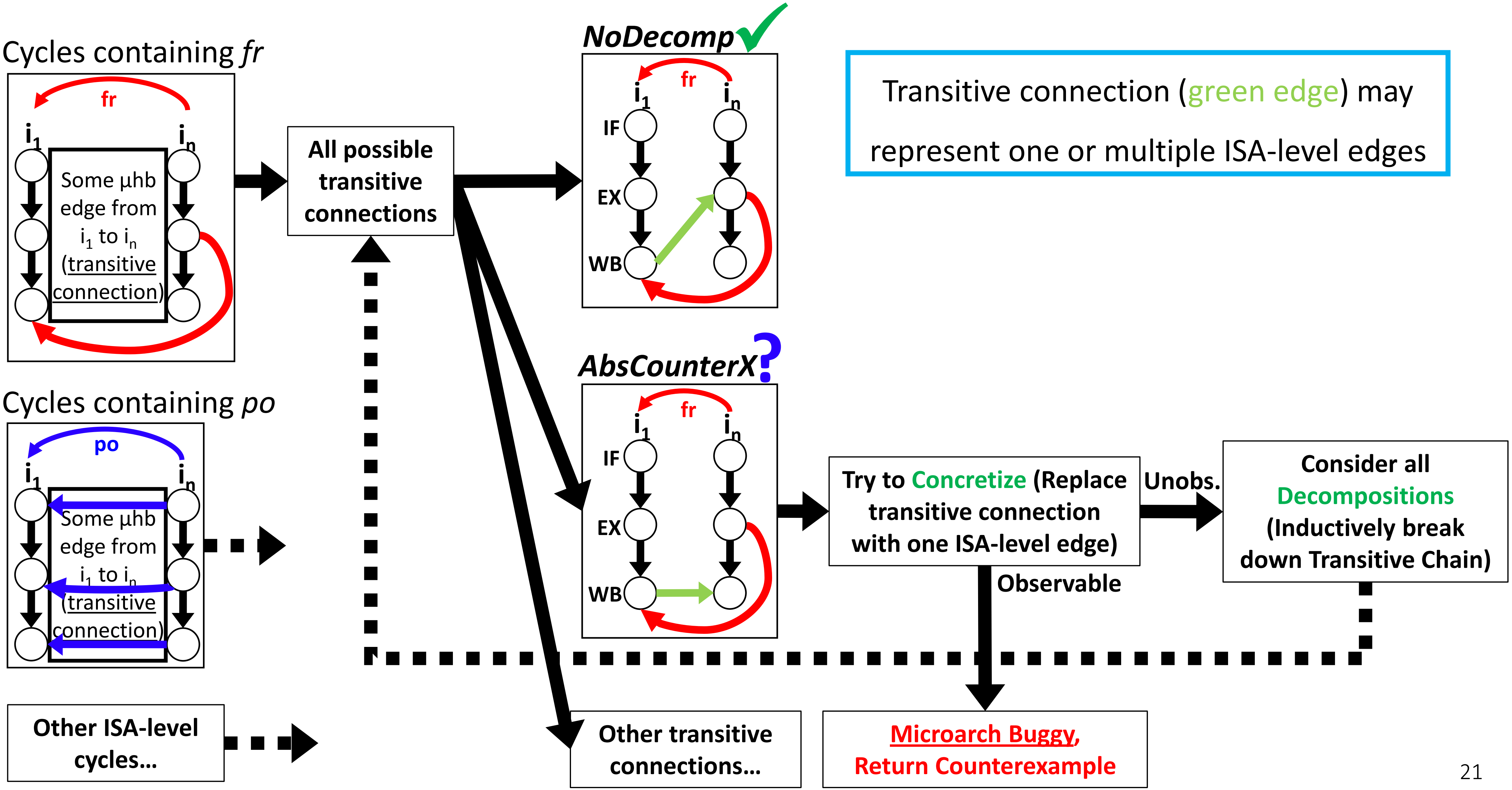
Transitive connection (green edge) may represent one or multiple ISA-level edges

Microarchitectural Correctness Proof



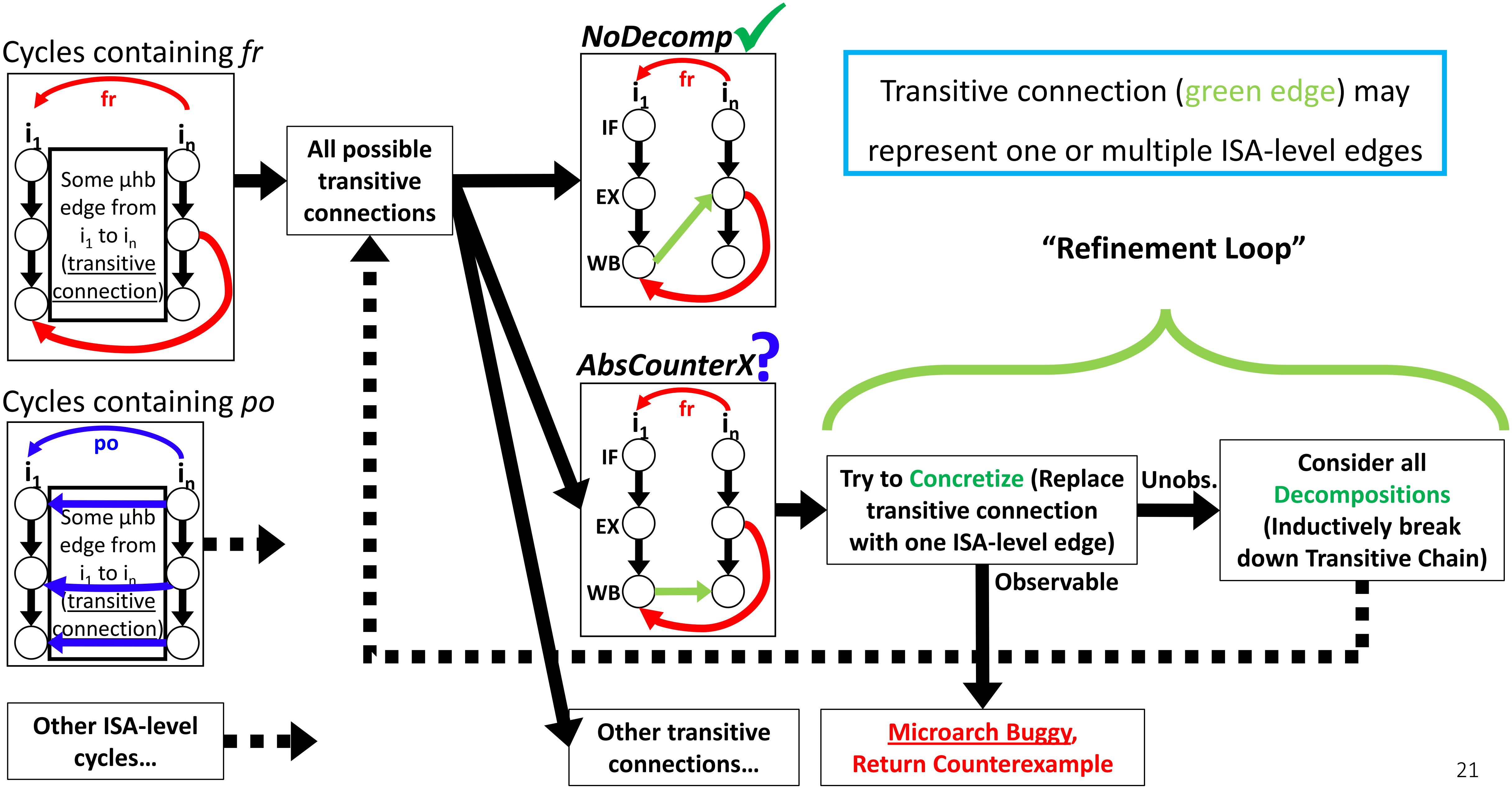
Transitive connection (green edge) may represent one or multiple ISA-level edges

Microarchitectural Correctness Proof



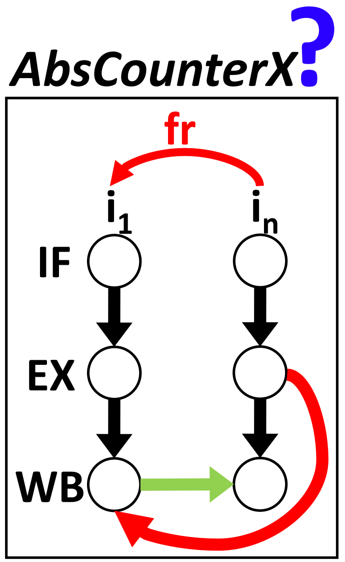
Transitive connection (green edge) may represent one or multiple ISA-level edges

Microarchitectural Correctness Proof



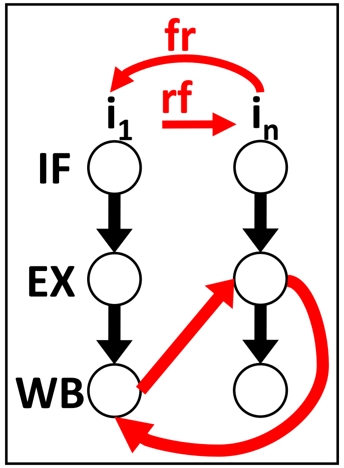
Refinement Loop: Concretization

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



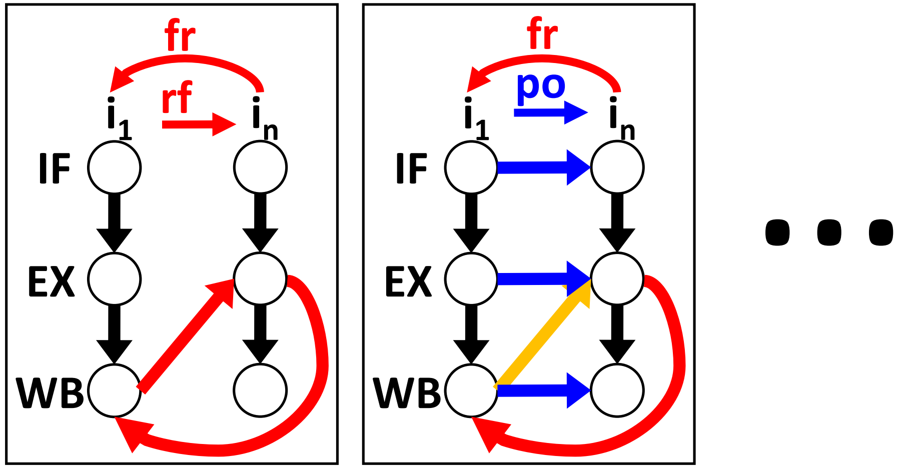
Refinement Loop: Concretization

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



Refinement Loop: Concretization

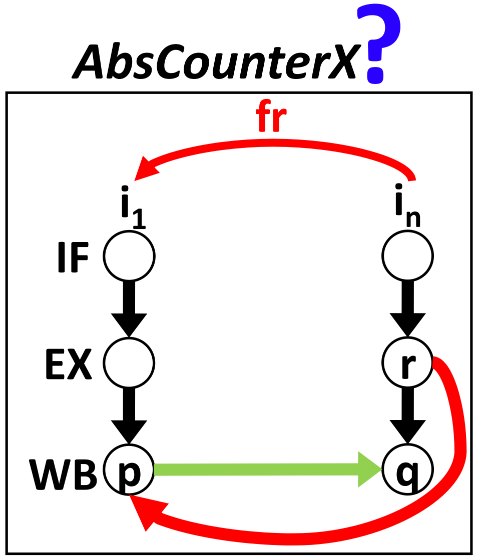
- Replaces transitive connection with a single ISA-level edge
  - All concretizations must be unobservable
  - Observable concretizations are counterexamples (bugs)



Refinement Loop: Decomposition

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

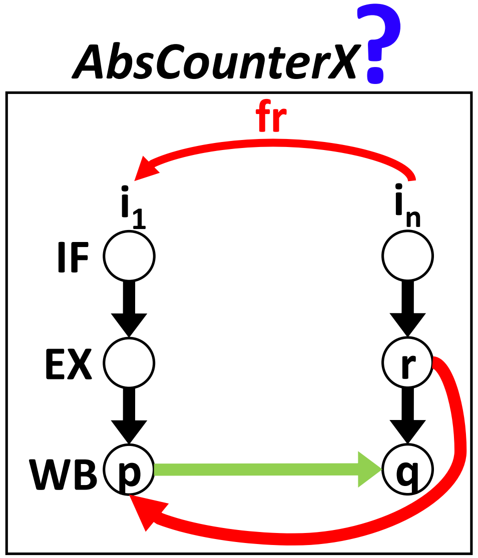
```
factorial(n)   =  factorial(n-1)  *         n
```



Refinement Loop: Decomposition

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

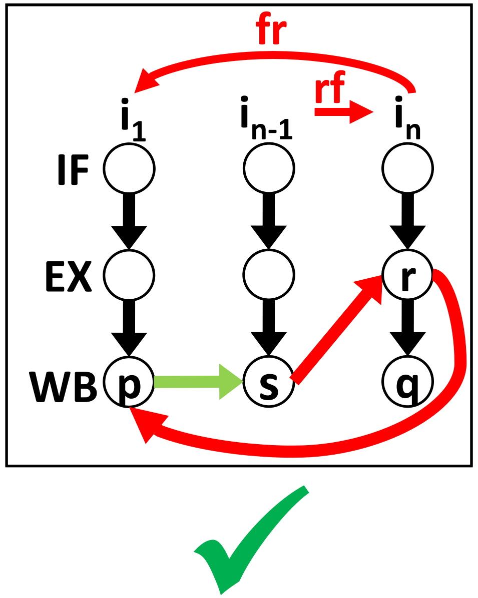
factorial(n)	=	factorial(n-1)	*	n
¦		¦		¦
Chain of length n	=	Chain of length n-1	+	"Peeled-off" edge



Refinement Loop: Decomposition

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

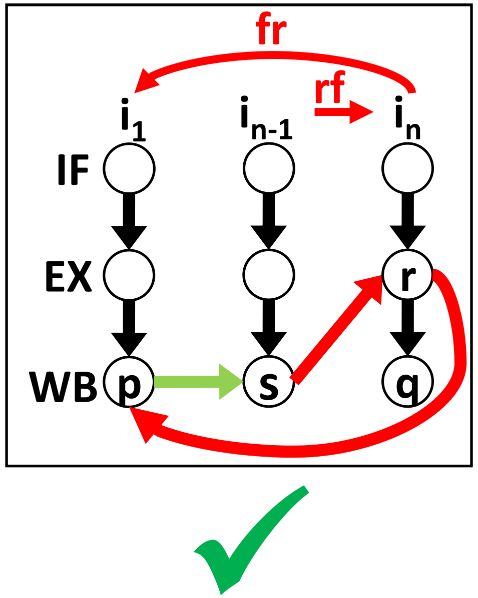
factorial(n)	=	factorial(n-1)	*	n
¦		¦		¦
Chain of length n	=	Chain of length n-1	+	"Peeled-off" edge

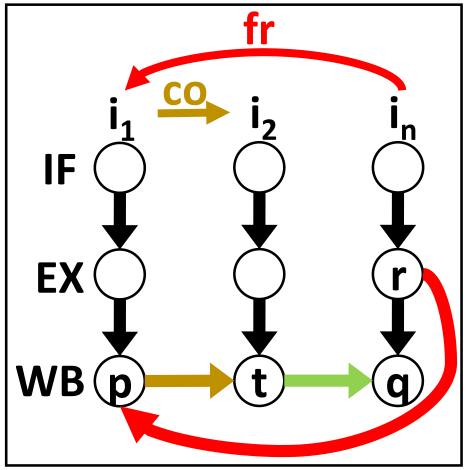


Refinement Loop: Decomposition

- 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



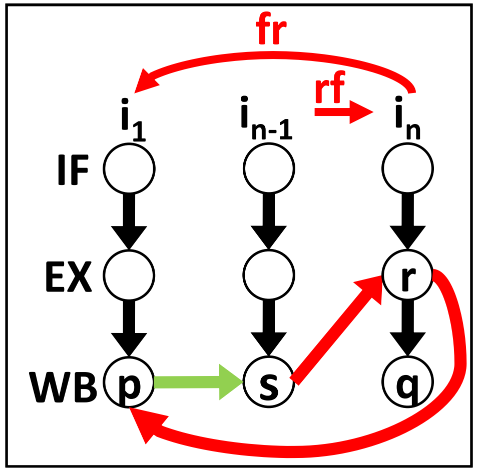


...

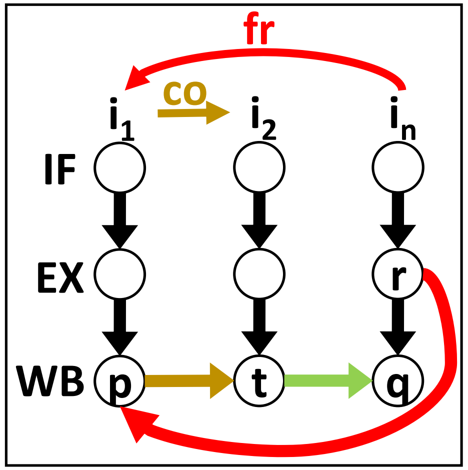
Refinement Loop: Decomposition

- 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!**

Hands-on: Let's Run PipeProof!

```
# Assuming you are in ~/pipeproof_tutorial/uarches/
$ prove_uarch -m simpleSC_fill.uarch -i SC -n
```

- What happens?

Hands-on: Let’s Run PipeProof!

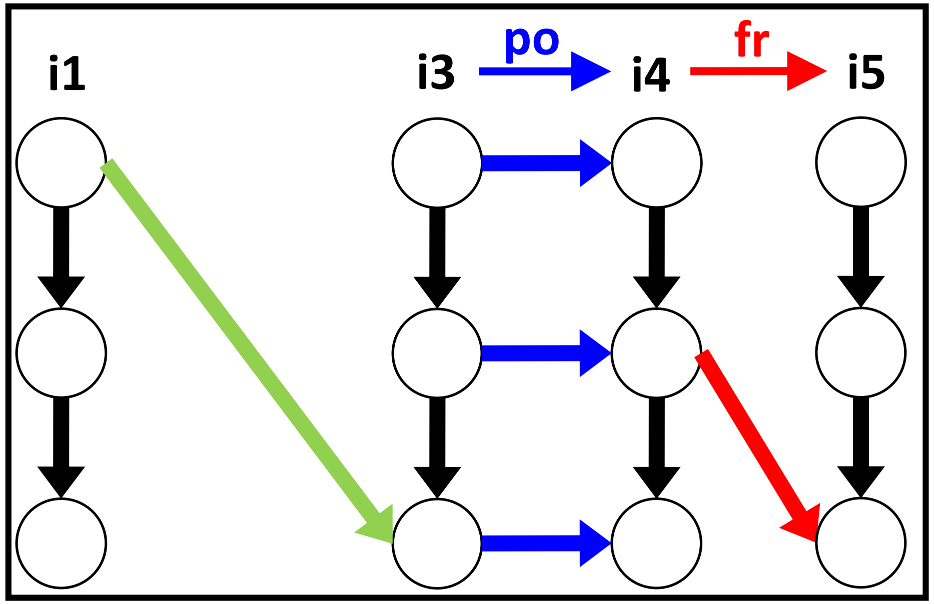
- PipeProof does not terminate; why?

```
...
// Checking Path:  (1/1, fr;)
// Checking Path:  (1/1, fr;) (1/1, po;fr;)
// Checking Path:  (1/1, fr;) (1/1, po;fr;) (1/1, po;po;fr;)
// Checking Path:  (1/1, fr;) (1/1, po;fr;) (1/1, po;po;fr;) (1/1,
po;po;po;fr;)
...
```

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)

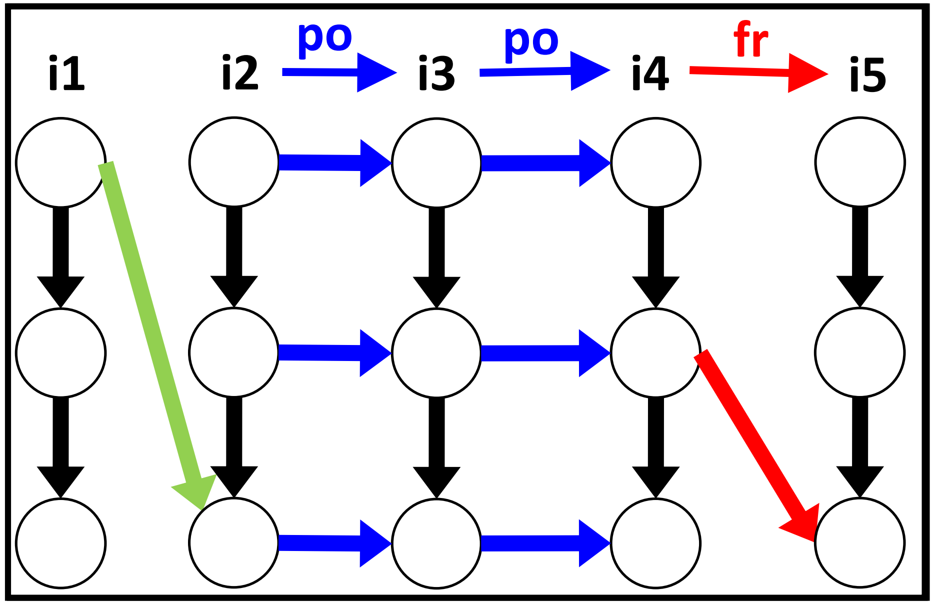
Abstract Counterexample



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)

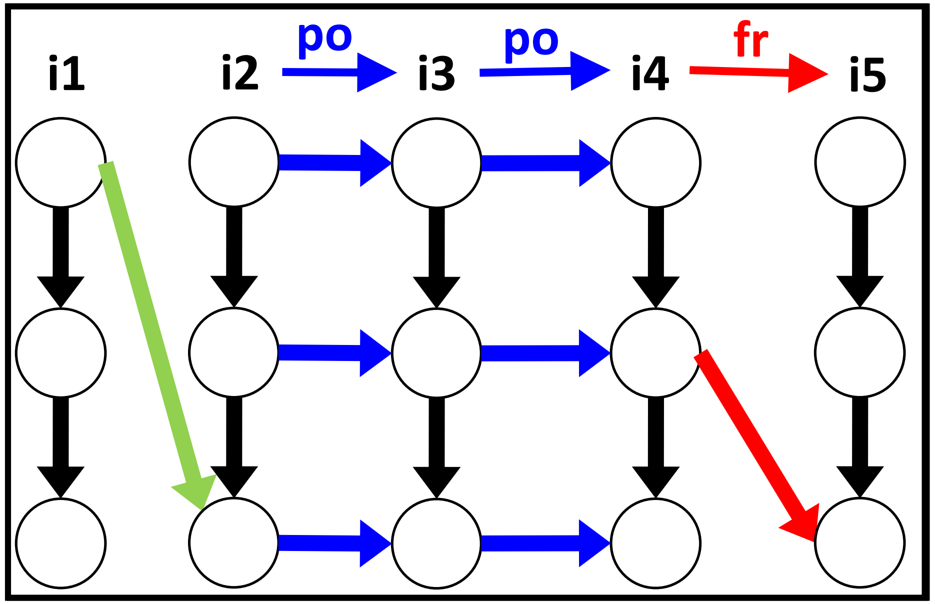
Repeating ISA-Level Pattern



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

Repeating ISA-Level Pattern

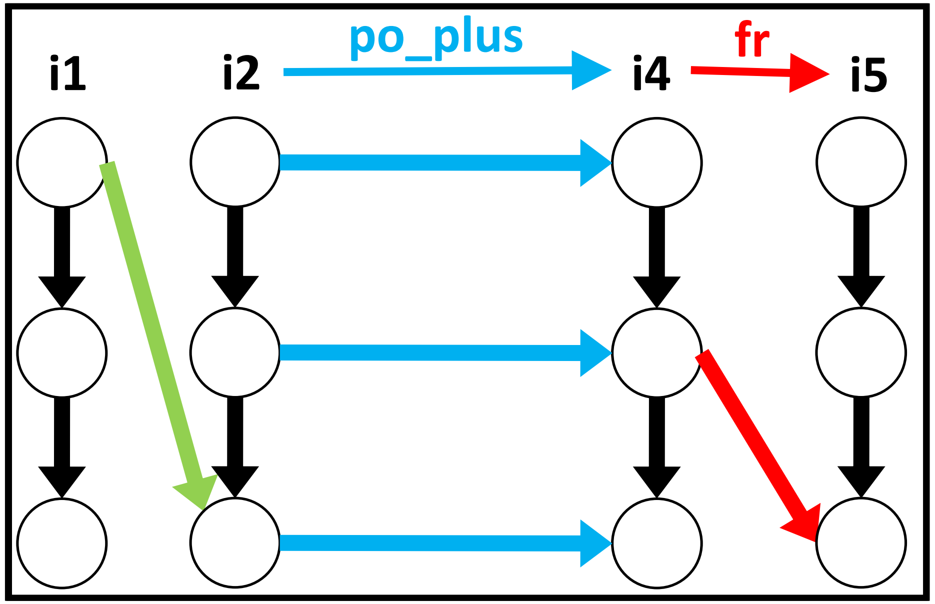


Can continue decomposing in this way forever!

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**

Adding the Chain Invariant for po+

- Uncomment the invariant at the end of `simpleSC_fill.uarch`:

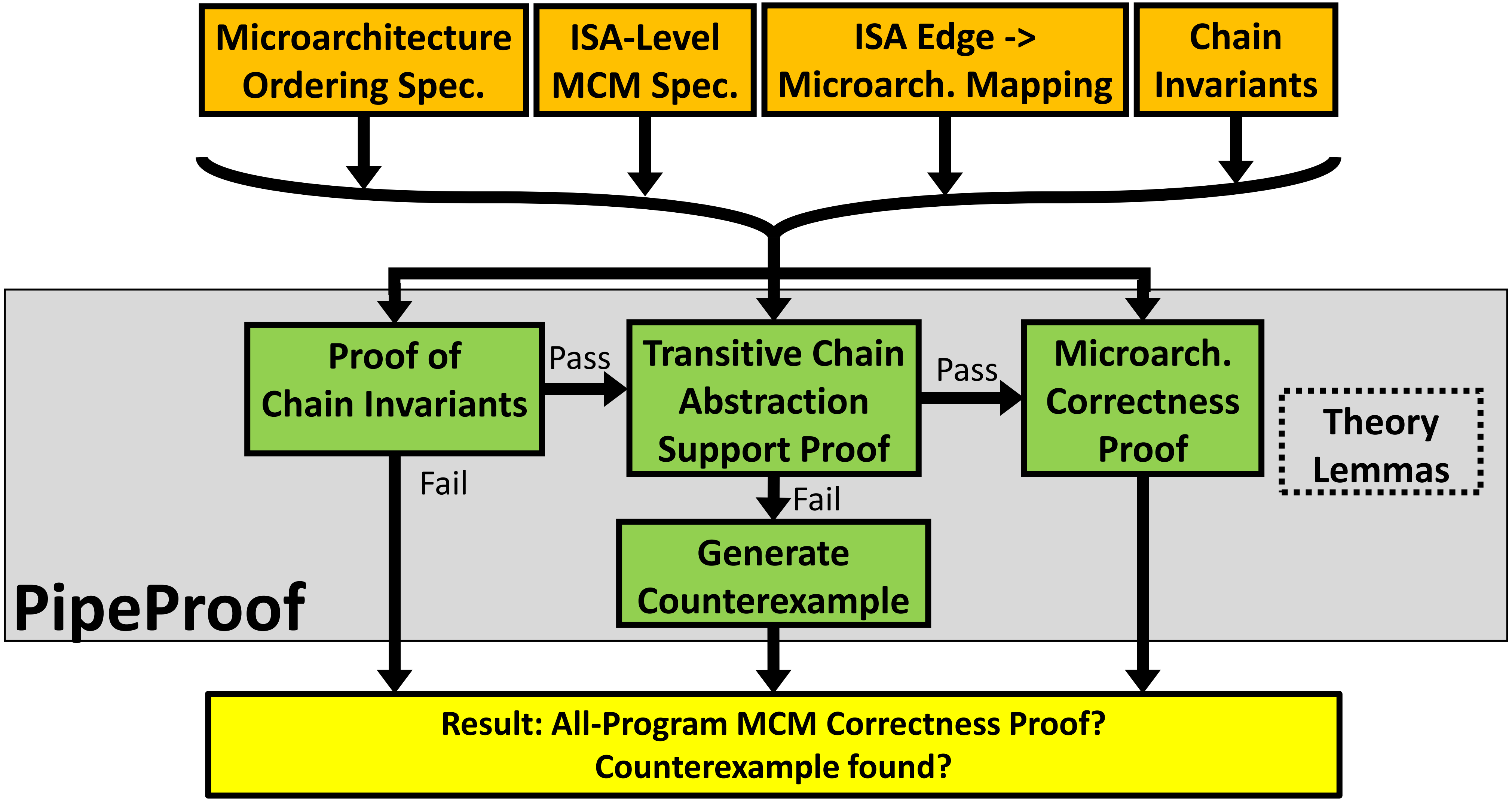
```
Axiom "Invariant_poplus":
forall microop "i",
forall microop "j",
HasDependency po_plus i j =>
  (AddEdge ((i, Fetch), (j, Fetch), "") /\ SameCore i j).
```

- Now re-run PipeProof:

```
# Assuming you are in ~/pipeproof_tutorial/uarches/
$ prove_uarch -m simpleSC_fill.uarch -i SC
```

- Should be proven in about a minute on the VM

PipeProof Block Diagram



PipeProof Does the Difficult Stuff for You!

- Users simply provide axioms, mappings, theory lemmas, and invariants
- PipeProof takes care of:
  - Proving TC Abstraction soundness
  - Proving any chain invariants
  - Refining abstraction (concretization and decomposition)
  - Inductively generating ISA-level cycles and covering all possibilities
- **Architects can use PipeProof; not just for formal methods experts!**

PipeProof: TSO Case Study

- Provided in VM as `solutions/simpleTSO.uarch`
  - Can try on your own time
  - Requires additional ISA-level relations, theory lemmas, and chain invariants
  - Will take at least 41 minutes to verify

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.

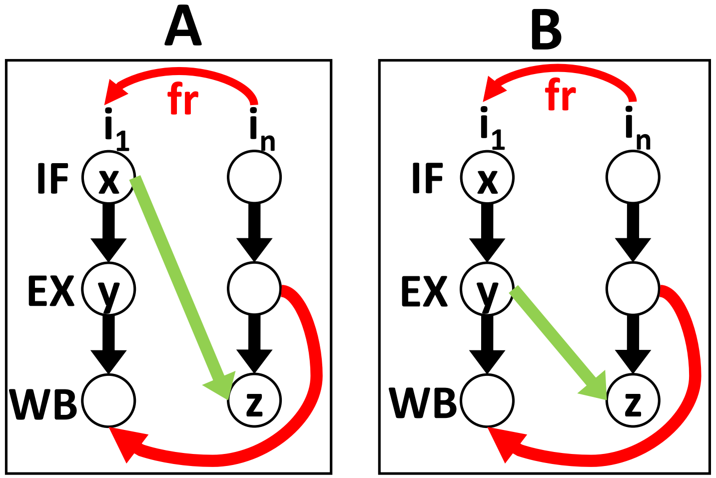
PipeProof Takeaways

- Automated All-Program Microarchitectural MCM Verification
  - Designers no longer need to choose between completeness and automation
  - 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
- Open-source: https://github.com/ymanerka/pipeproof
- Accolades:
  - Nominated for Best Paper at MICRO 2018
  - "Hon. Mention" from 2018 IEEE Micro Top Picks of Comp. Arch. Conferences

Backup Slides

Covering Sets Optimization

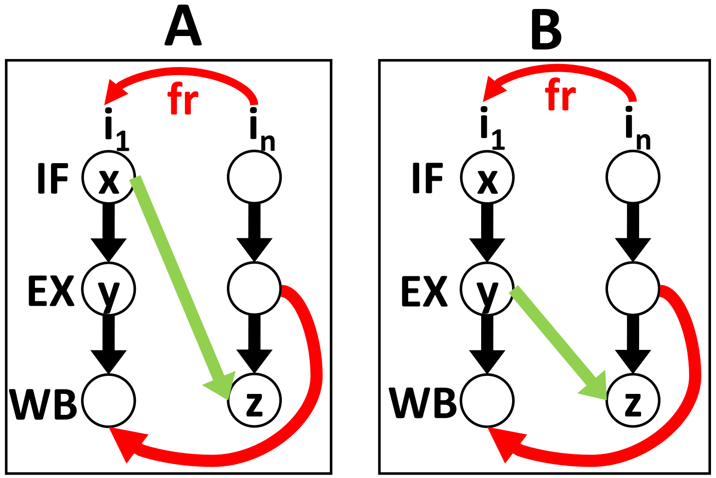
- Must verify across all possible transitive connections
- Each decomposition creates a new set of transitive connections
  - Can quickly lead to a case explosion
- The Covering Sets Optimization eliminates redundant transitive connections



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

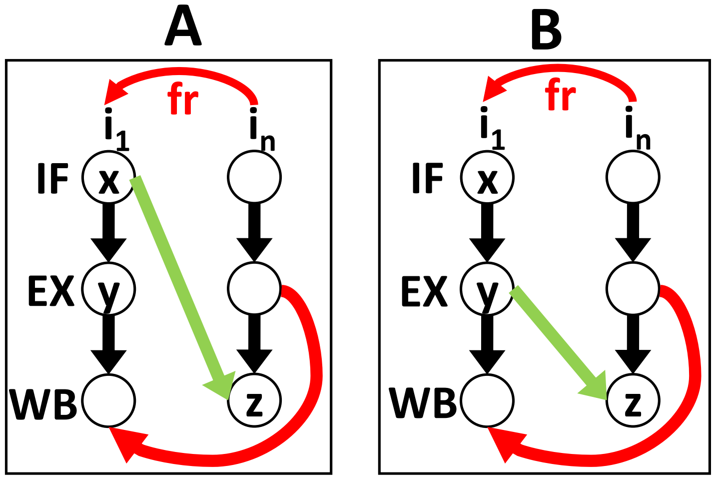
Graph A has an edge from x→z (tran conn.)



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.)

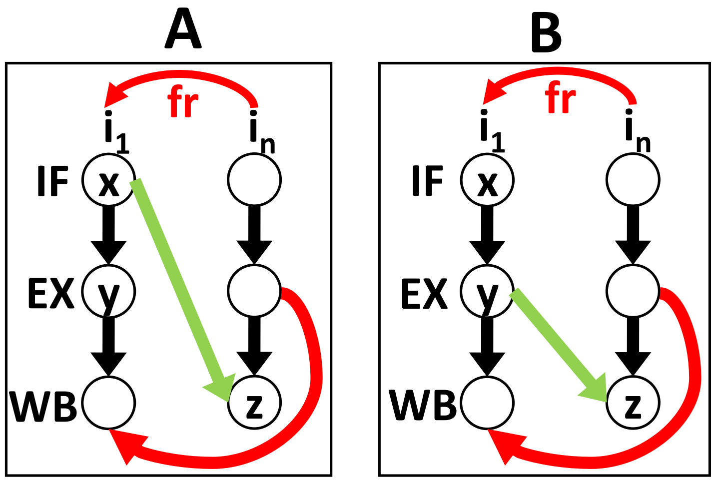


Graph B has edges from y→z (tran conn.) and x→z (by transitivity)

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.)

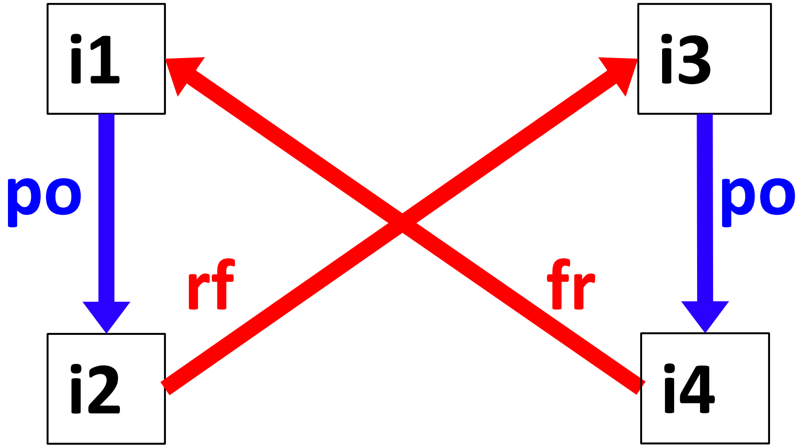


Graph B has edges from y→z (tran conn.) and x→z (by transitivity)

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

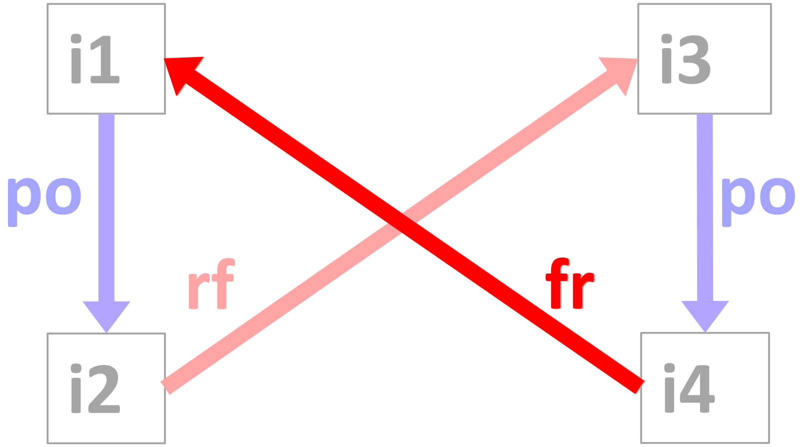
Memoization Optimization

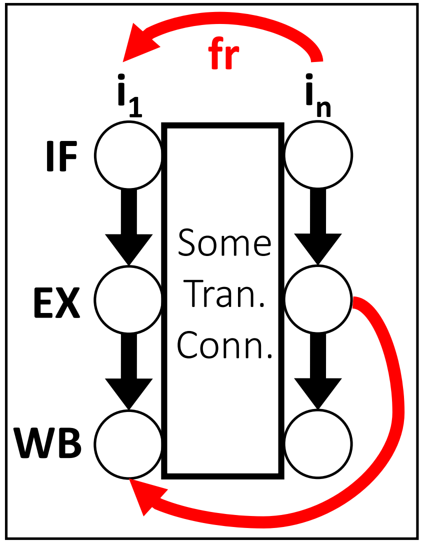
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified



Memoization Optimization

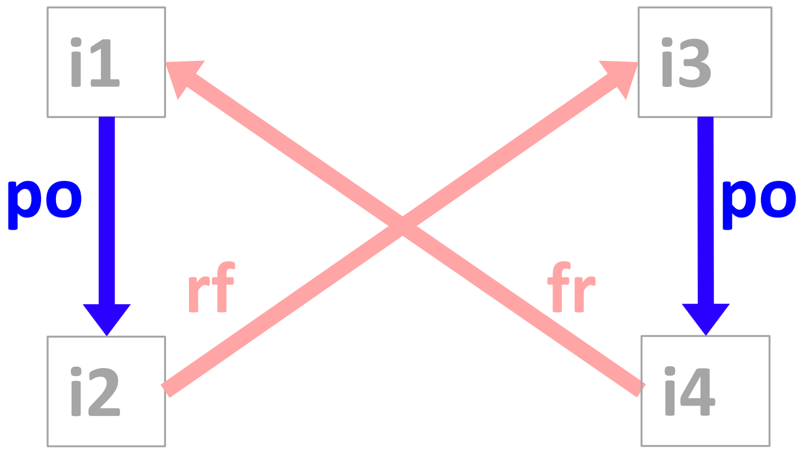
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified

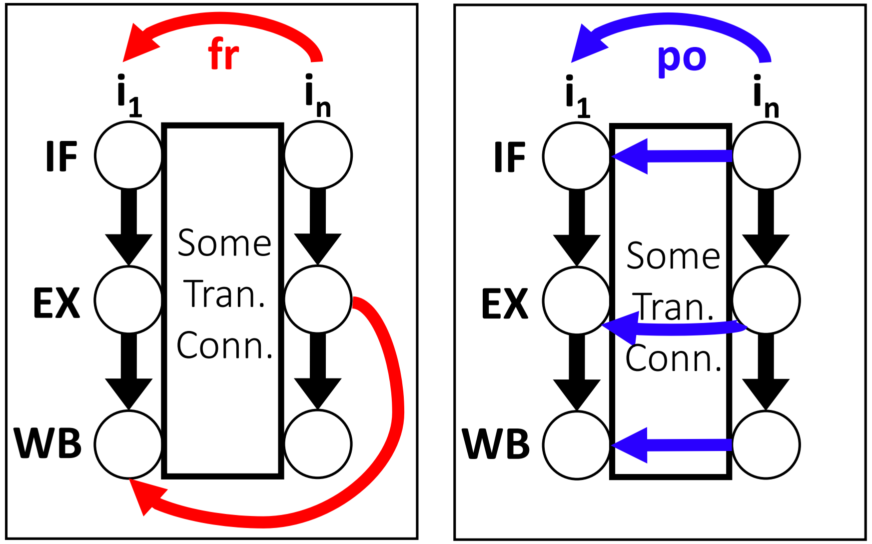




Memoization Optimization

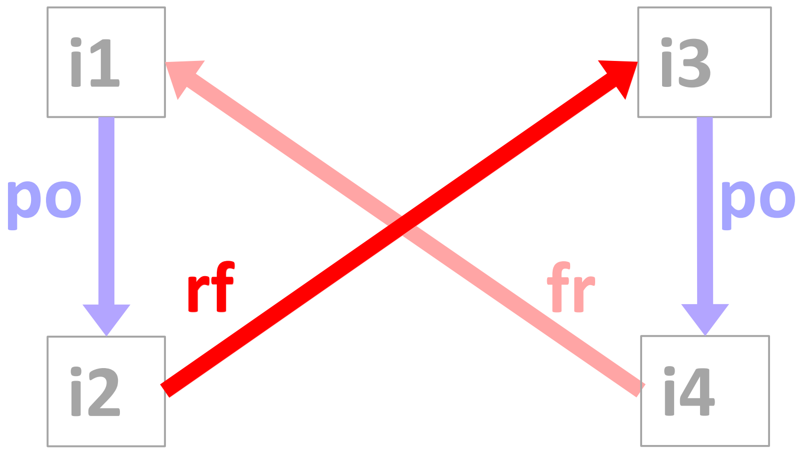
- Base PipeProof algorithm examines some cycles multiple times
- Memoization eliminates redundant checks of cycles that have already been verified



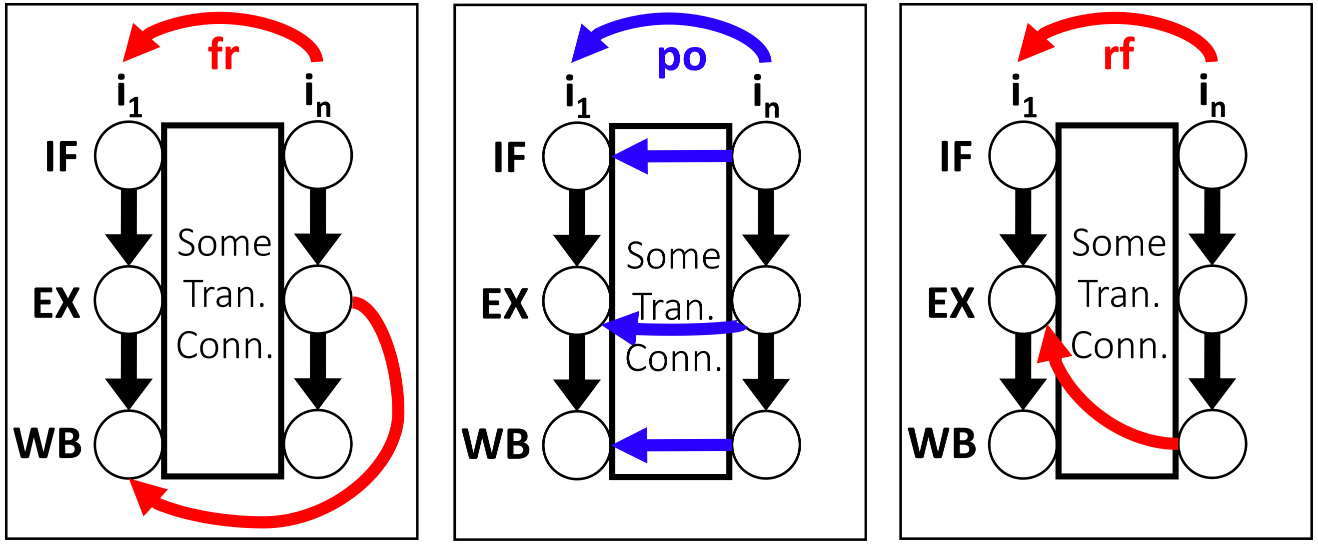


Memoization Optimization

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

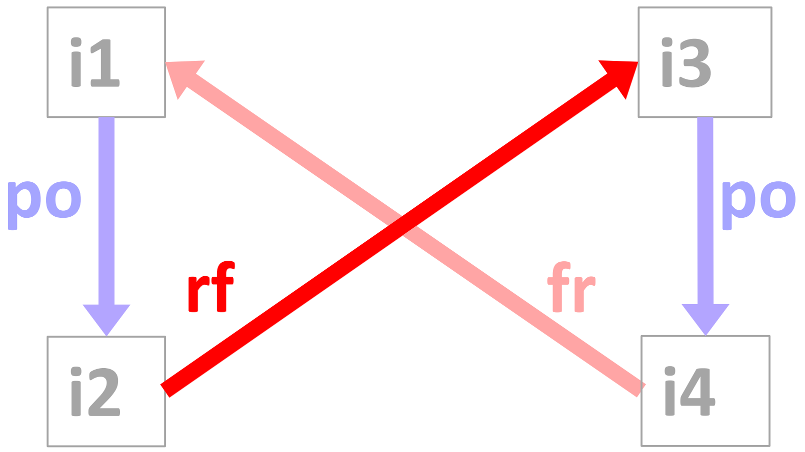


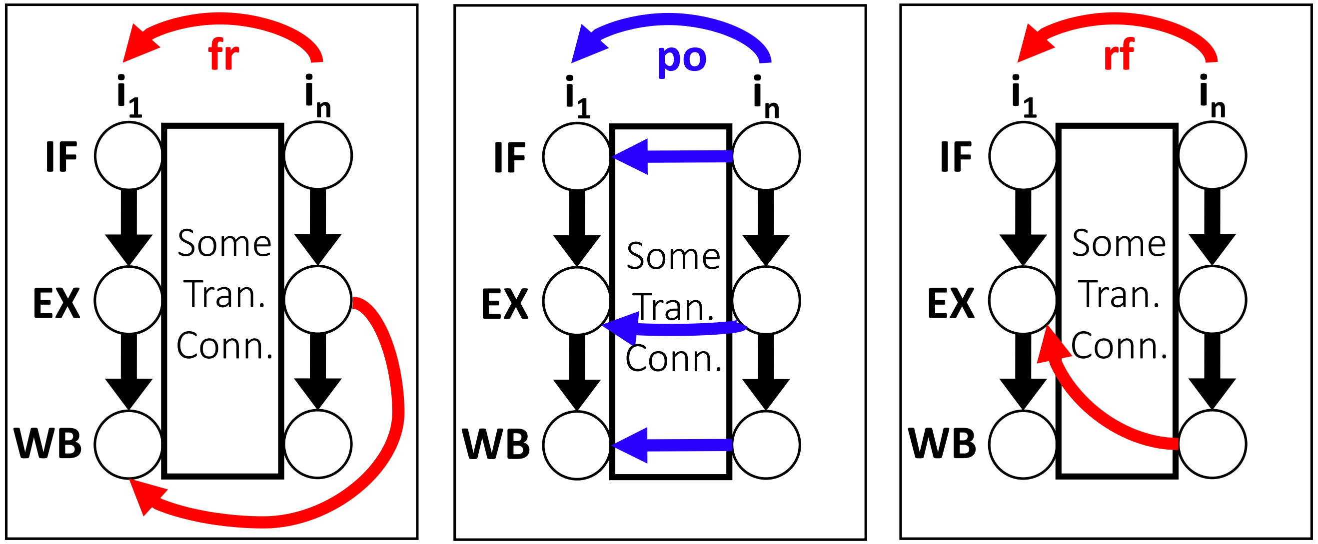
Same cycle is checked 3 times!



Memoization Optimization

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



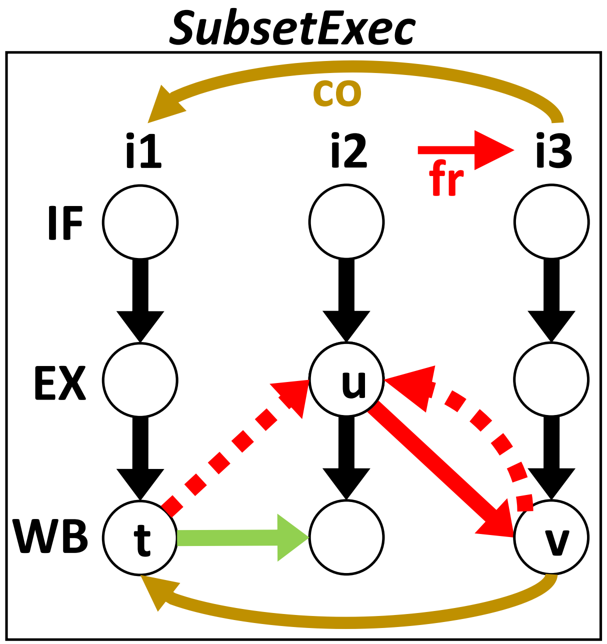


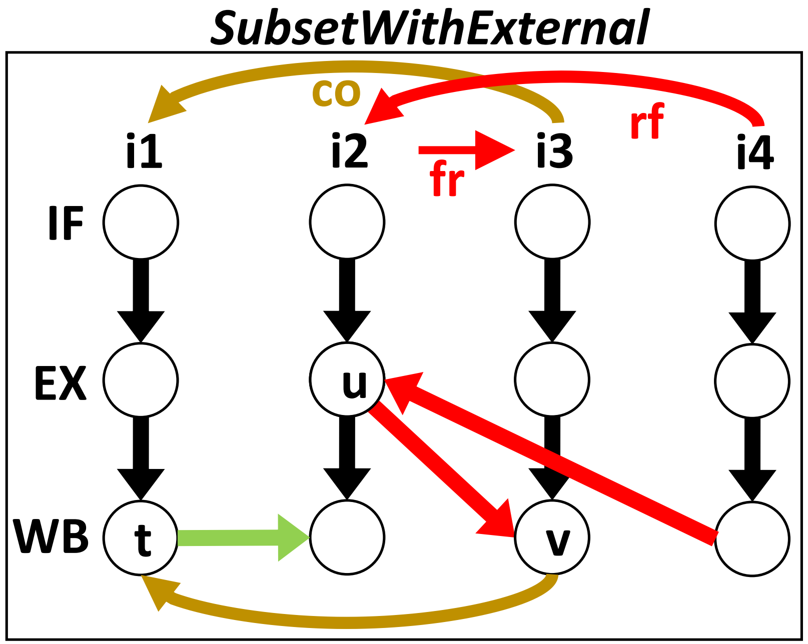
Same cycle is checked 3 times!

<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

The Adequate Model Over-Approximation

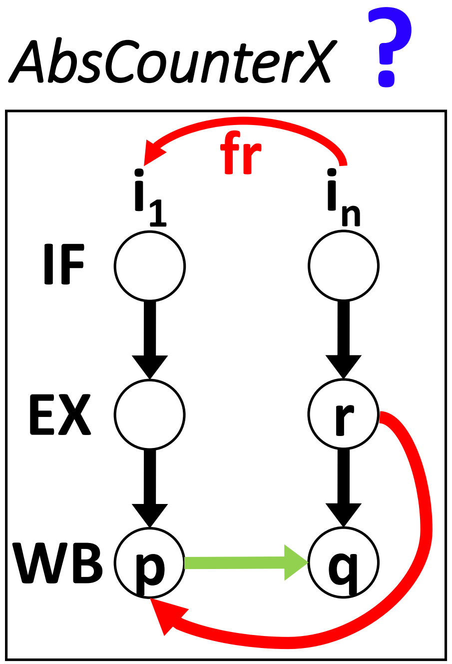
- Addition of an instruction can make unobservable execution observable!
- Need to work with over-approximation of microarchitectural constraints
- PipeProof sets all **`exists`** clauses to true as its over-approximation

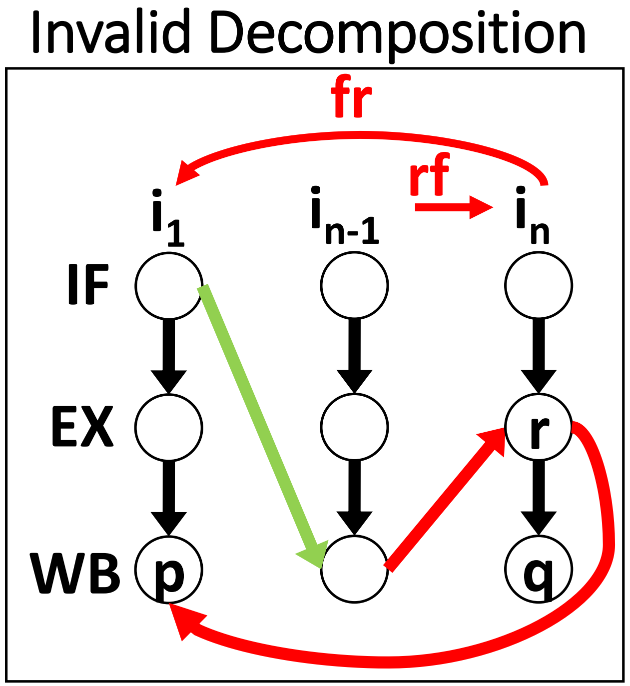




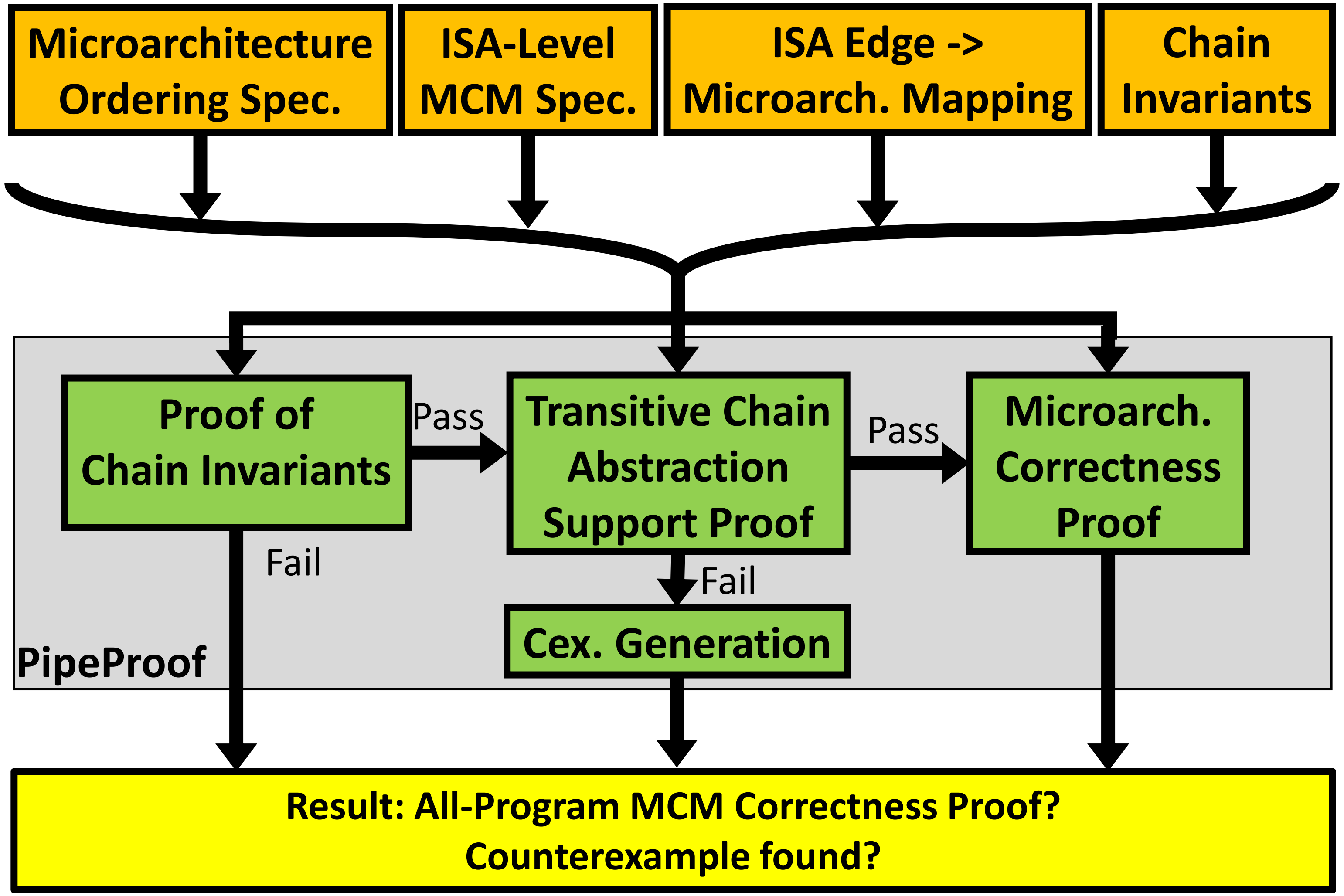
Filtering Invalid Decompositions

- 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

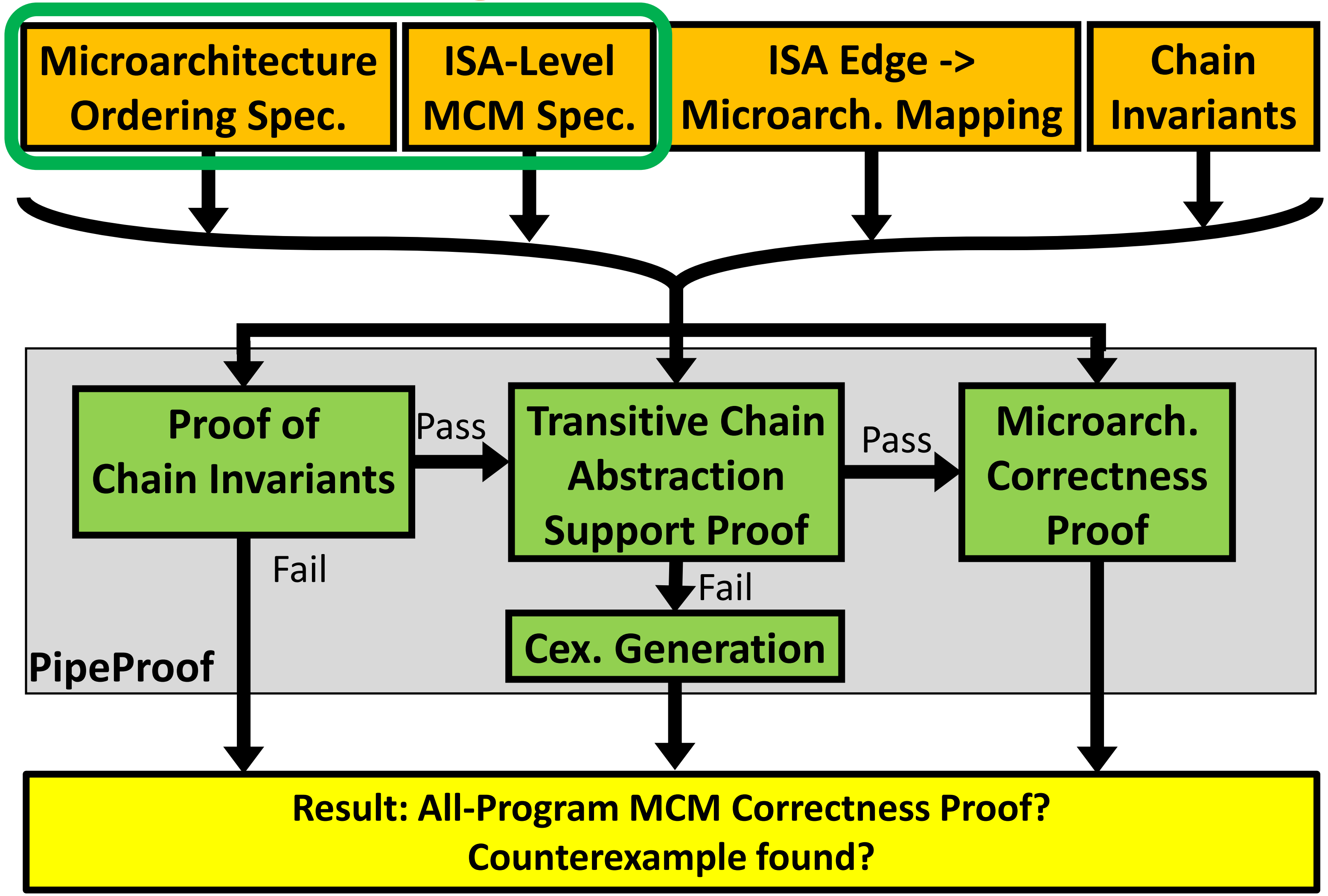




PipeProof Block Diagram

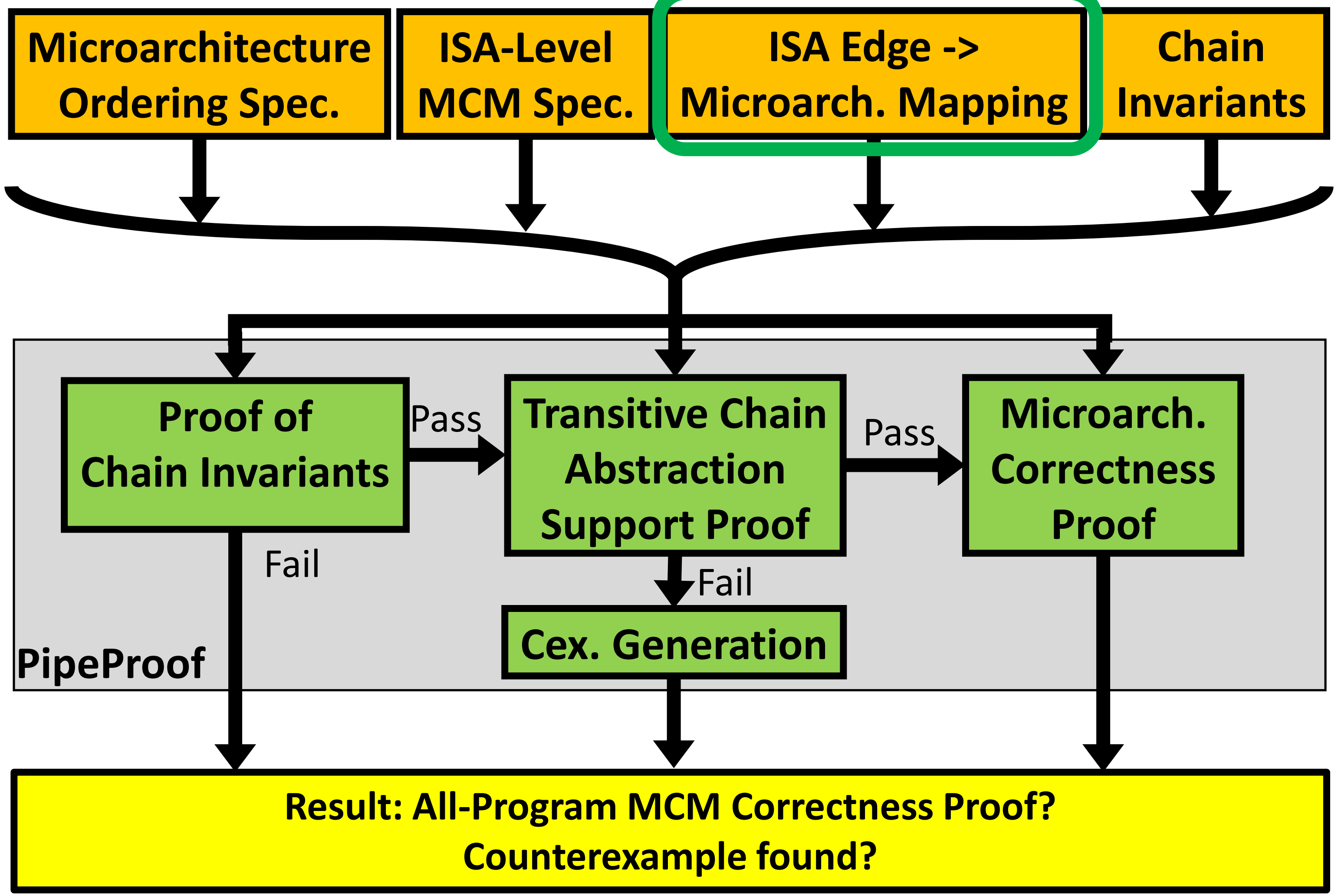


PipeProof Block Diagram

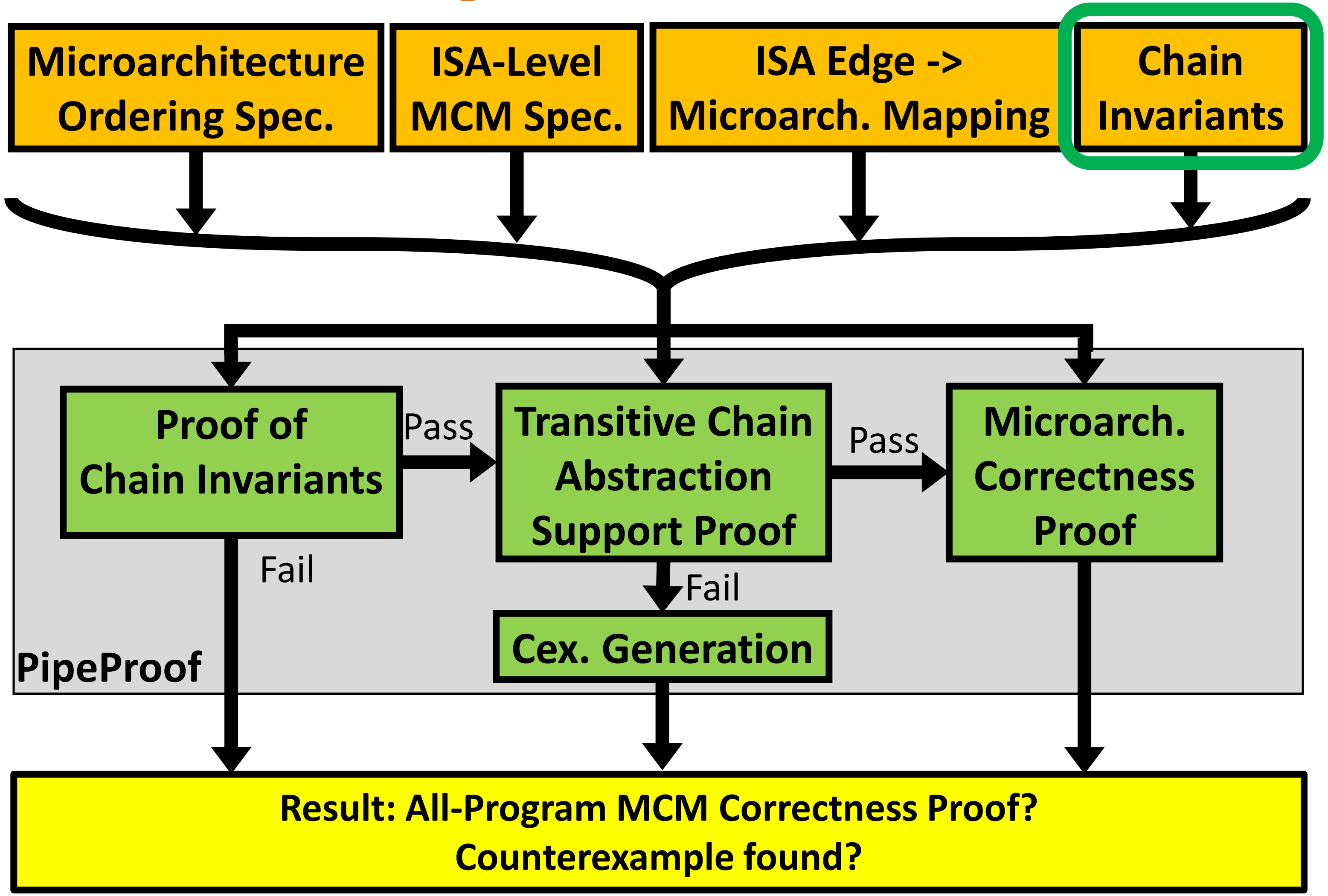


PipeProof Block Diagram

Links ISA-level and µarch executions

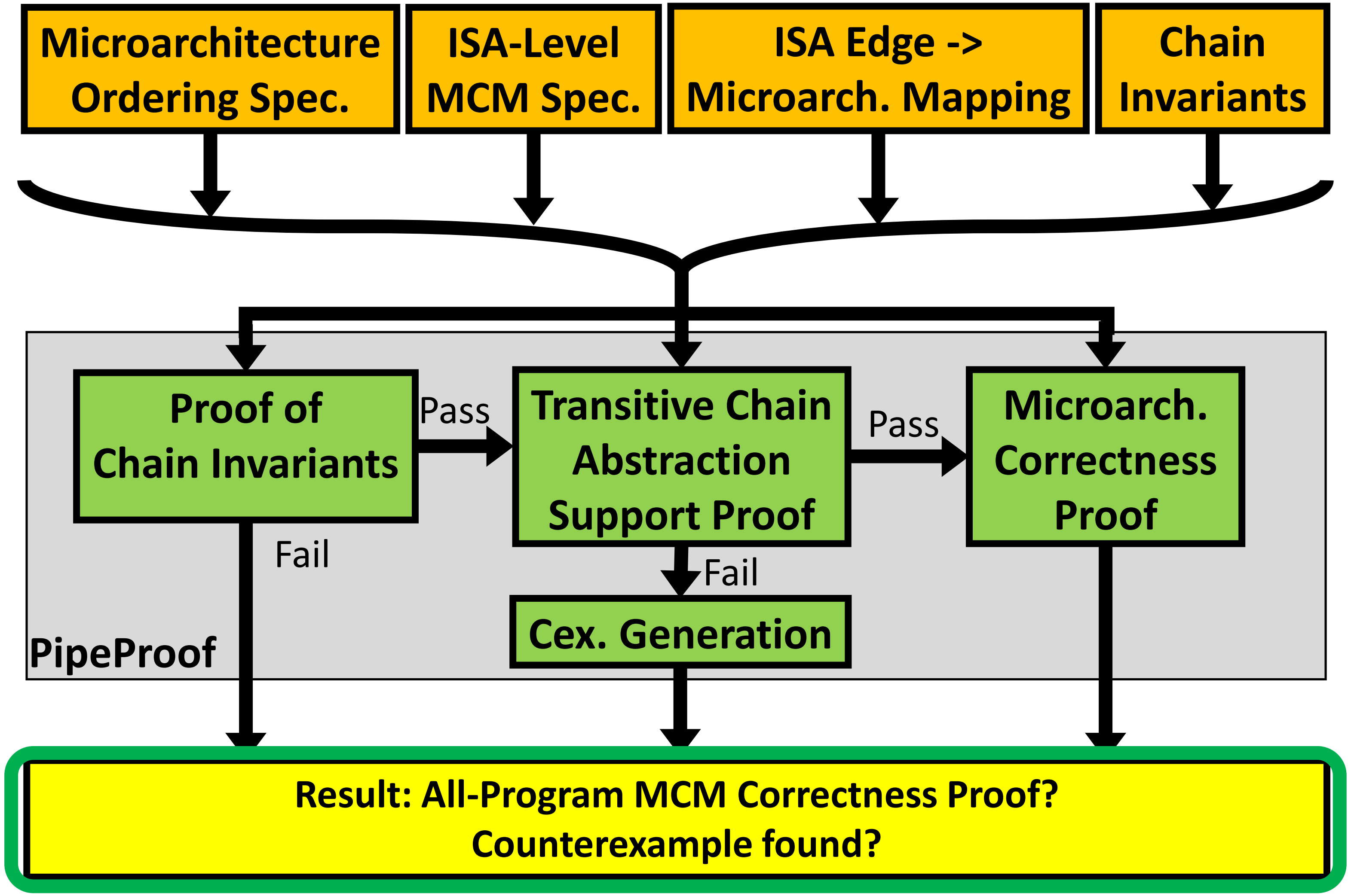


PipeProof Block Diagram



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

Supporting proofs provide foundation for correctness proof