Why Memory Consistency Models Matter... And tools for analyzing and verifying them

Caroline Trippel & Yatin A. Manerkar

Princeton University

UPMARC 2018

While you wait:

1) Make sure you've got VirtualBox downloaded to your laptop:

https://www.virtualbox.org/wiki/Downloads

2) Make sure you have the most recent version of the Tutorial VM downloaded:

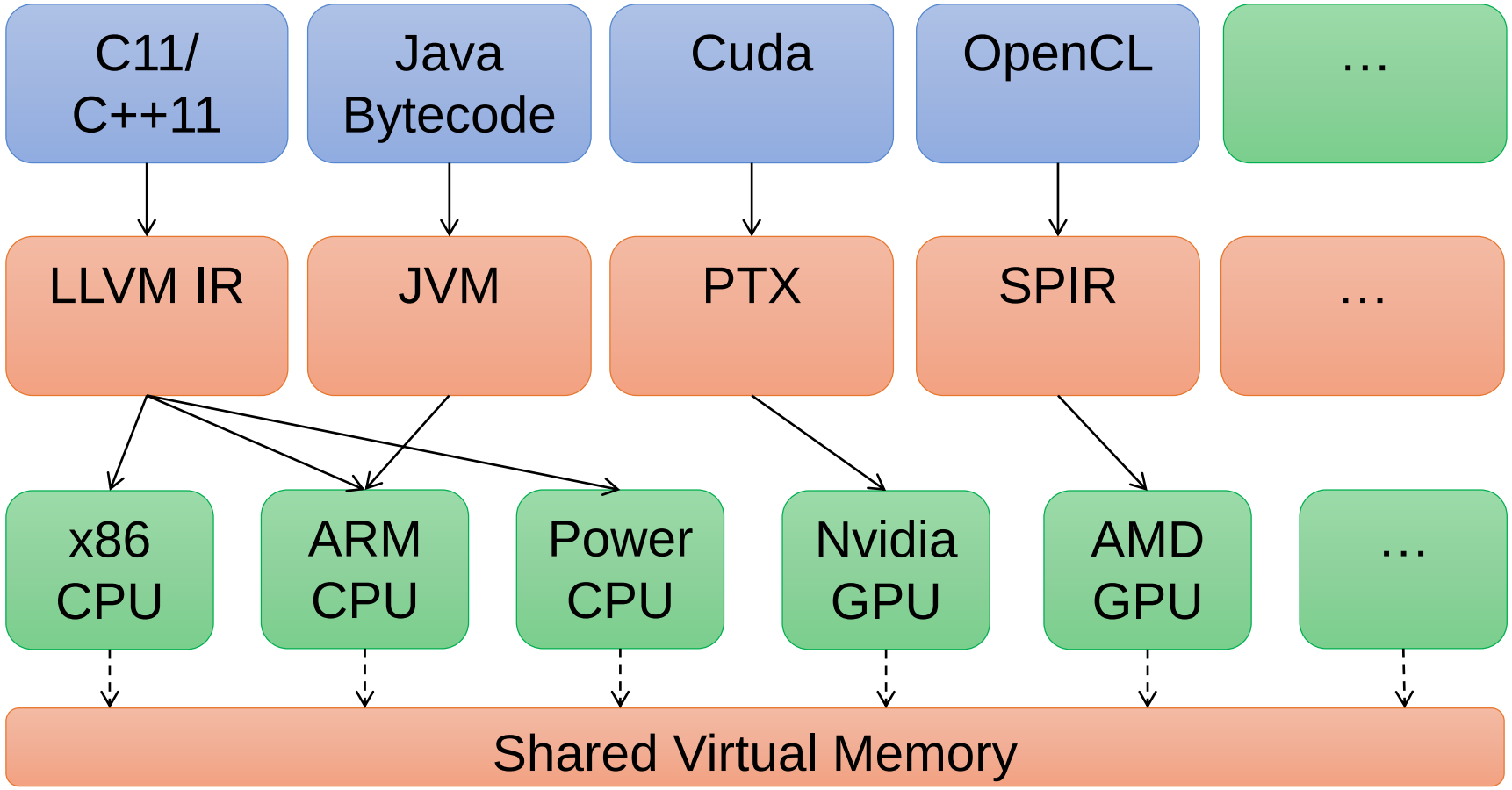
http://check.cs.princeton.edu/tutorial_vm/Check_Tools_VM.ova

VM Password: mcmsarefun

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

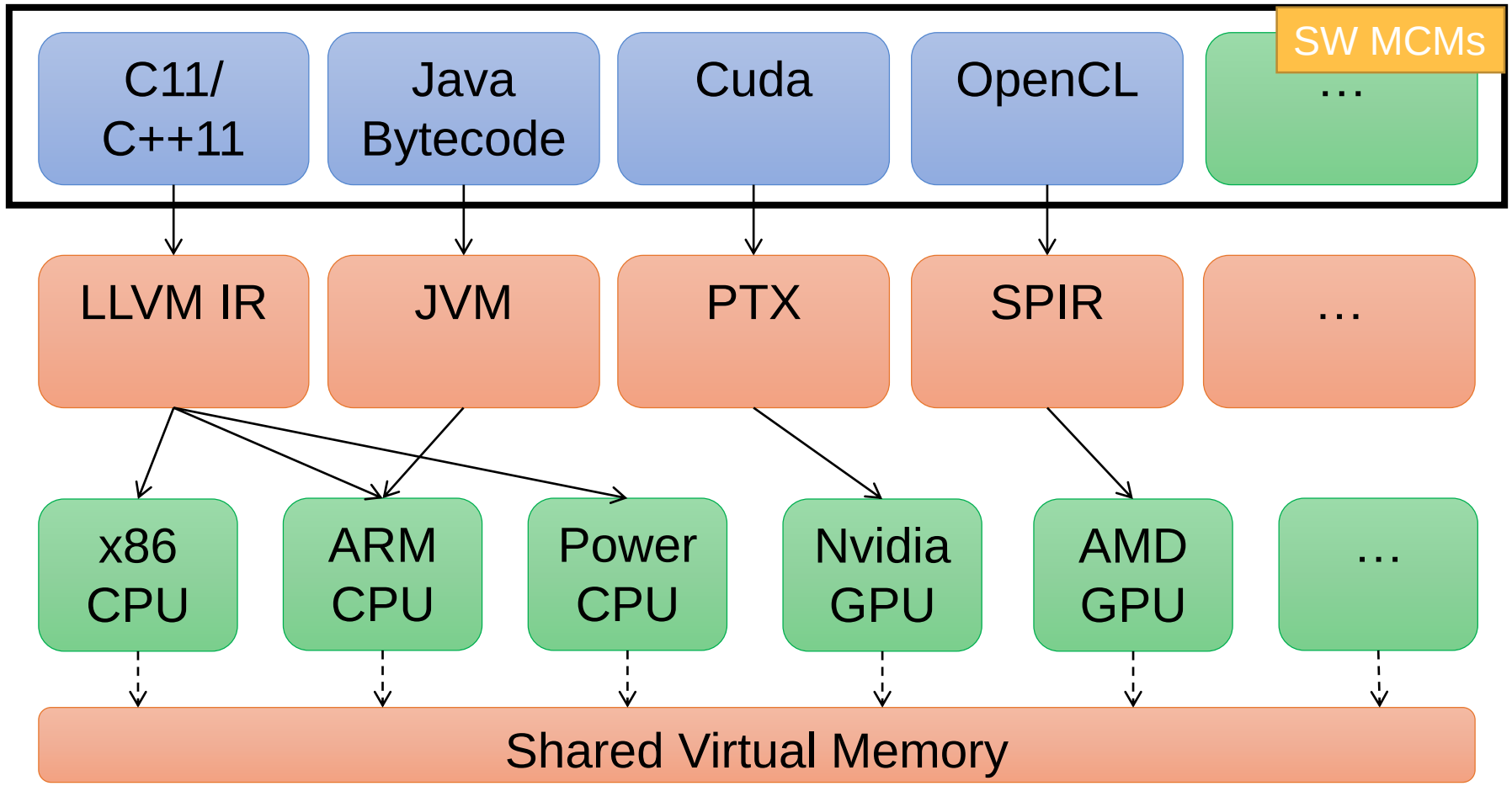
Memory Consistency Models

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



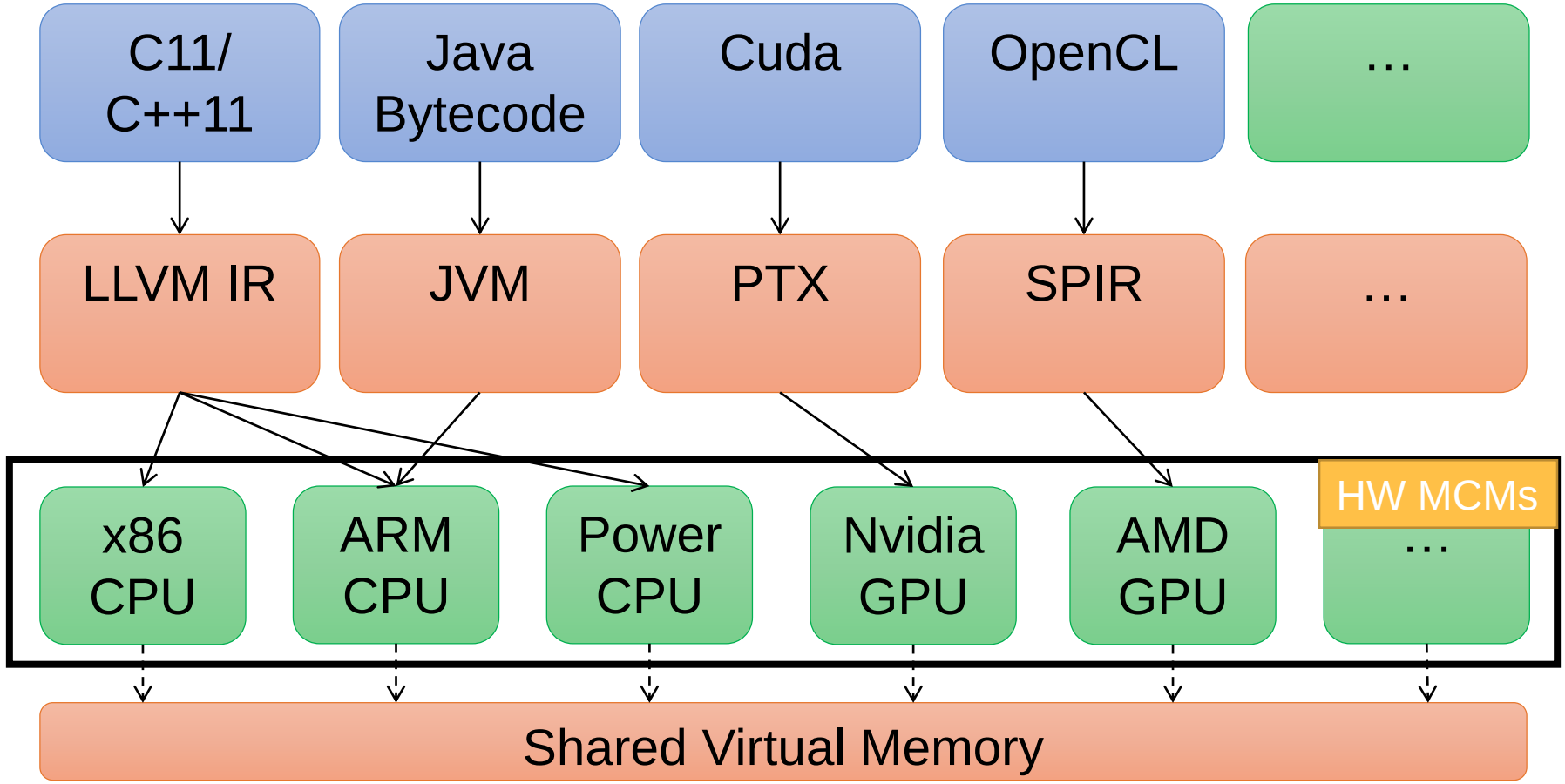
Memory Consistency Models

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



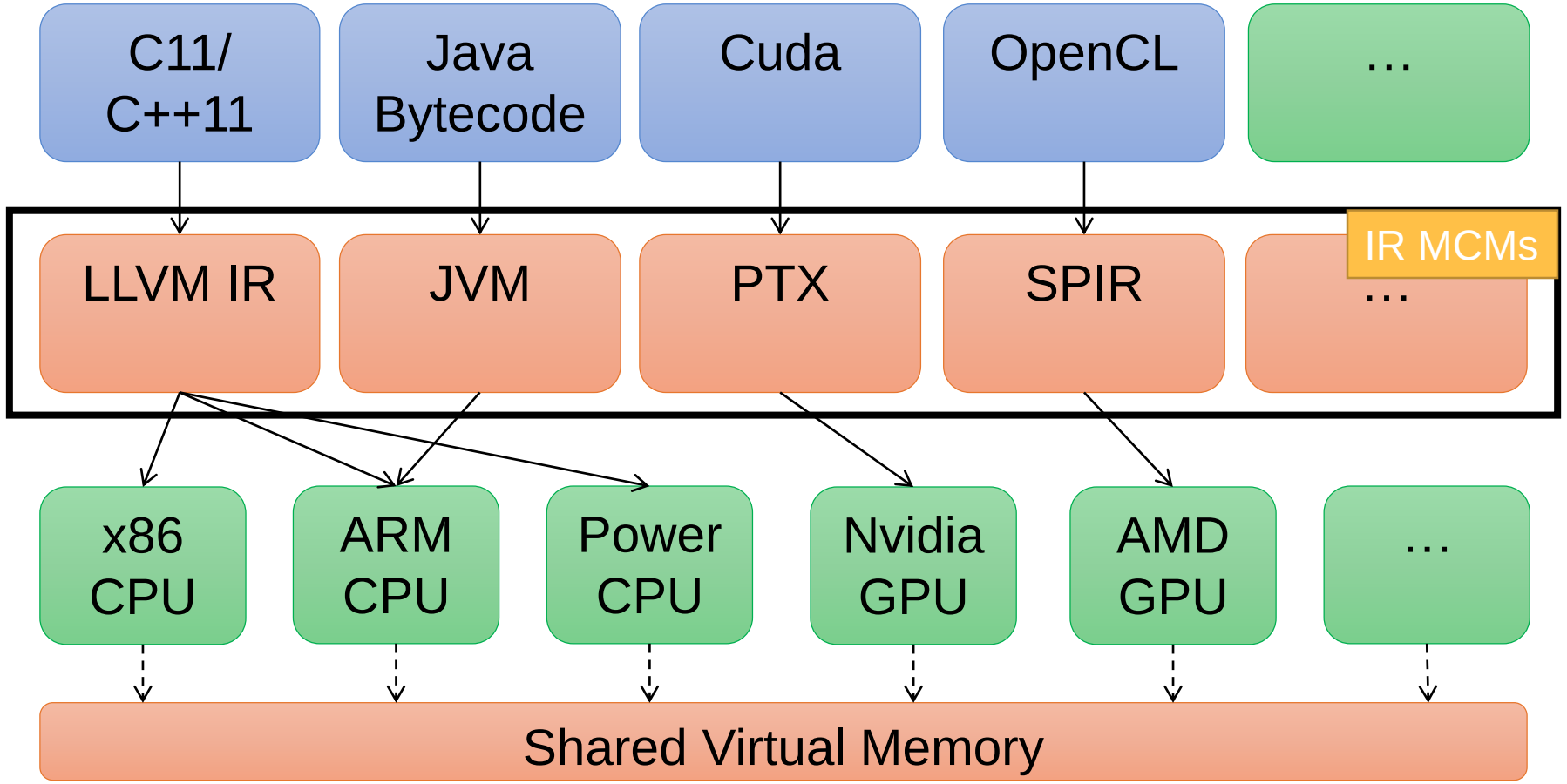
Memory Consistency Models

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

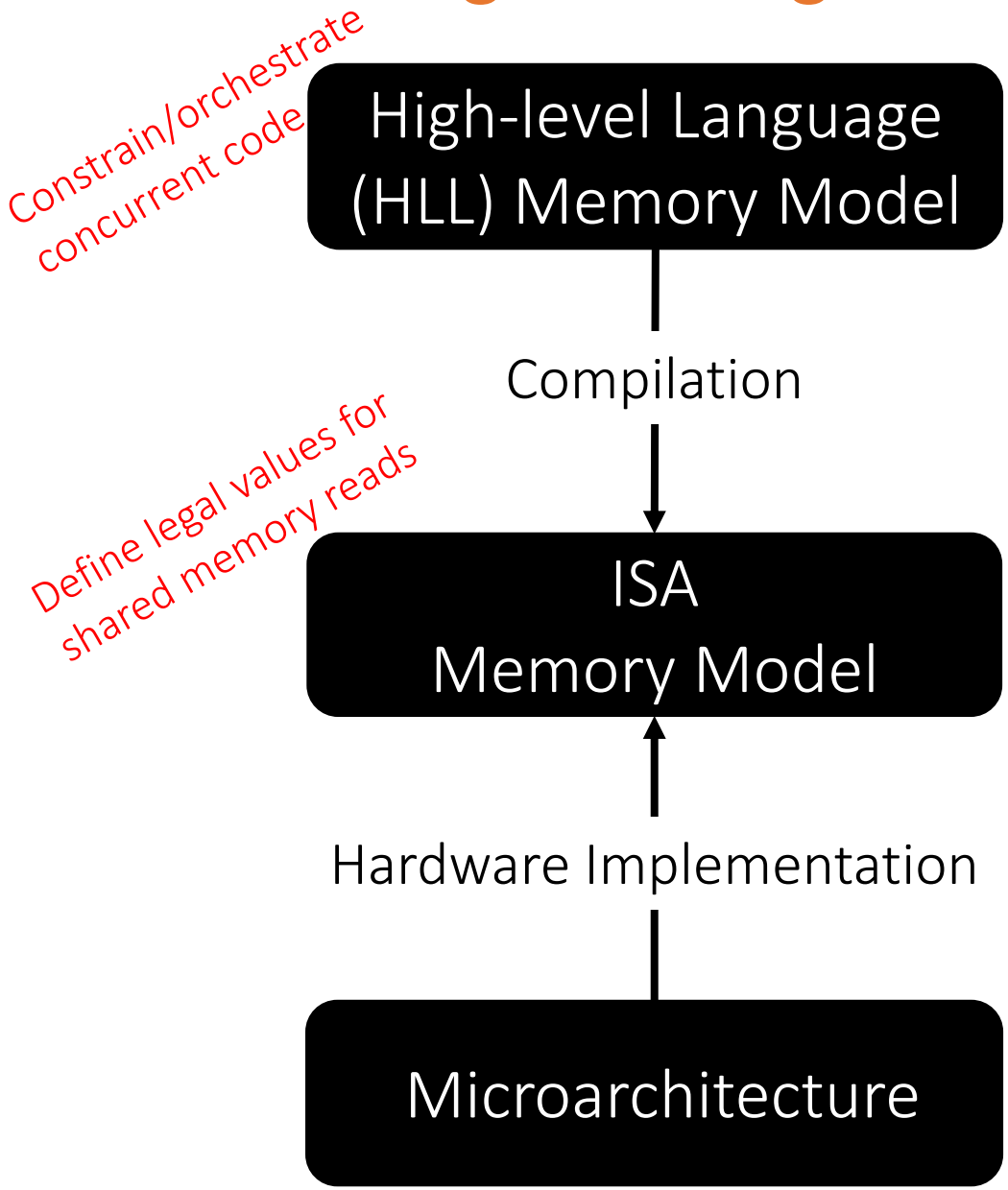


Memory Consistency Models

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



What can go wrong?



- A bug in any layer can cause a "correct" program to produce incorrect outcomes
  - Ill-specified HLL memory model
  - Incorrect HLL→ISA compilation
  - Inadequate ISA specification
  - Incorrect hardware implementation
- Benefits to verifying this stack as a whole

Goals

- Ultimately want to write correct and efficient concurrent programs
- Concurrent programs are compiled and eventually run on hardware
  - Hardware reorders instructions and state updates for performance
  - Shared memory for inter-thread communication
- Memory Consistency Models (MCMs): govern inter-thread communication in the presence of shared memory
  - Specified at the various layers of the hardware-software stack
  - Require precise specifications, translations between layers
- MCM bugs anywhere in hardware-software stack can cause a "correct" high-level language program can produce incorrect results

Our Approach Today

- Basic overview of MCMs
- Our suite of tools for MCM verification
- Hands-on verification examples
  - **PipeCheck:** Verification of a HW design w.r.t. an ISA MCM specification
  - **TriCheck:** Full-stack (HLL→Compiler→ISA→HW) MCM verification
- Provide you with a general modeling/verification approach that can be applied to other problem areas

Outline

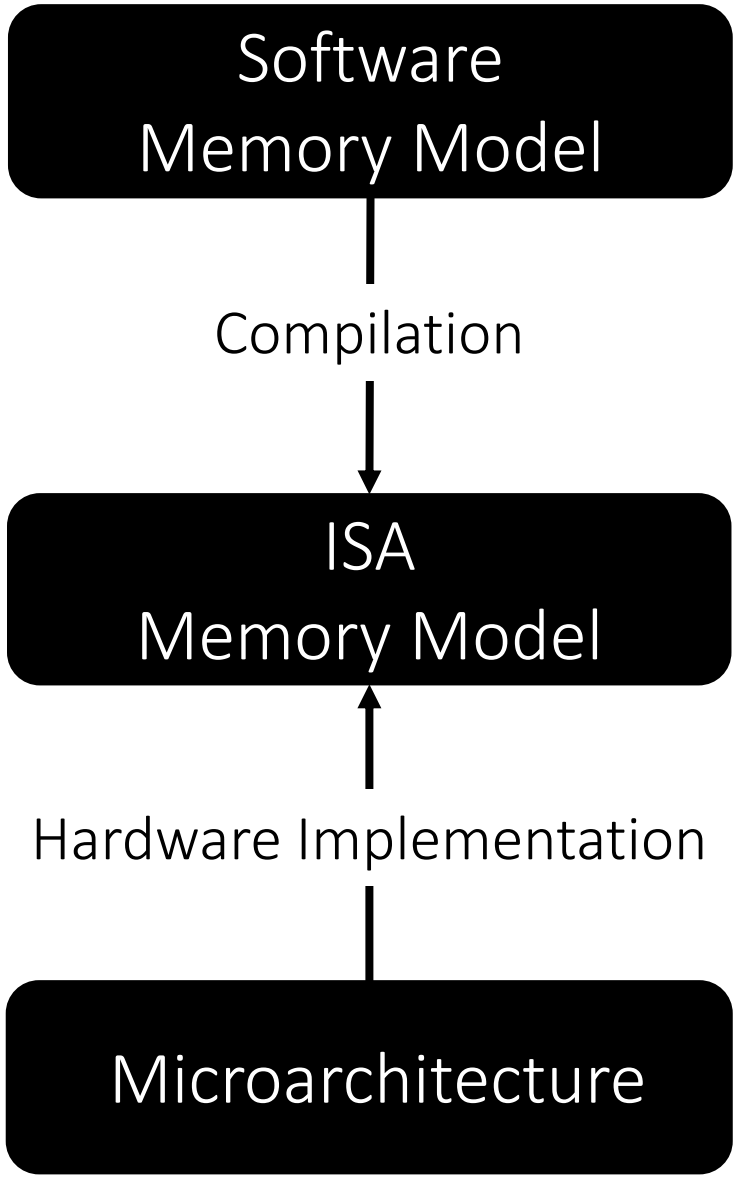
- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, …

Motivating Example: ARM Read-after-Read Hazard

- ARM ISA spec ambiguous regarding same-address Ld→Ld ordering:
  - ARM compilers did not insert fences
  - Some ARM implementations relax same-address Ld→Ld ordering
- C/C++ variables with atomic type require same-addr. Ld→Ld ordering
- ARM issued errata1:
  - Rewrite compilers to insert fences with performance penalties
- ARM had ordering instructions in ISA to guarantee correctness

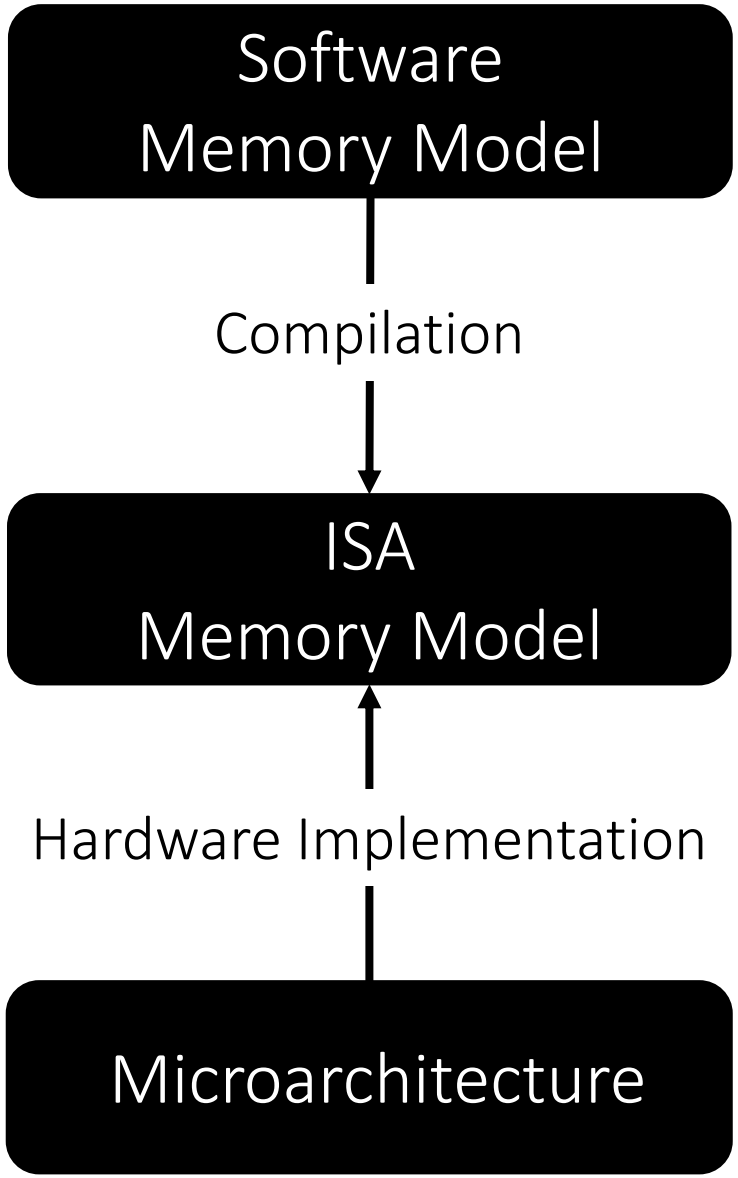
[1]ARM. Cortex-A9 MPCore, programmer advice notice, read-after-read hazards. ARM Reference 761319., 2011. http://infocenter.arm.com/help/topic/com.arm.doc. uan0004a/UAN0004A_a9_read_read.pdf.

ARM Read-Read Hazard



ARM Cortex-A9

ARM Read-Read Hazard

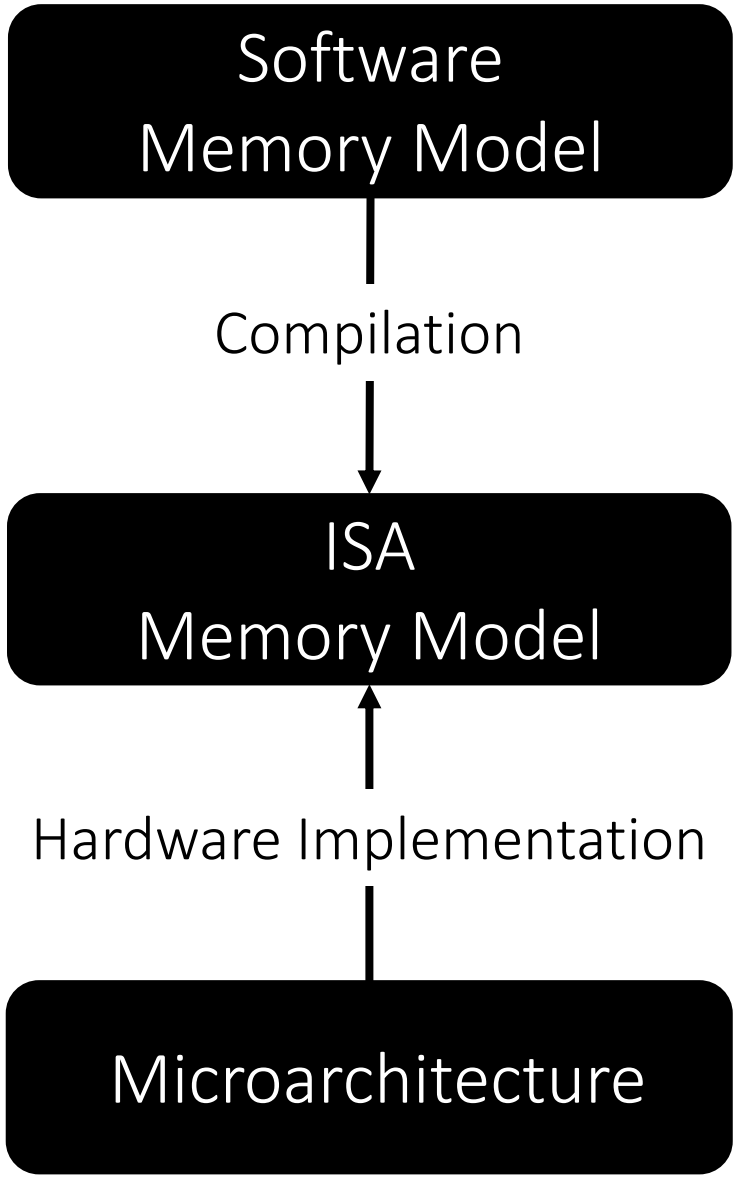


Which HLL(s) to support?

C11/C++11	ARMv7
st(rlx)	STR
ld(rlx)	LDR
ld(acq)	LDR; DMB
...	...

ARM Cortex-A9

ARM Read-Read Hazard



How does this affect real programs?

Loading data through a pointer

Initial conditions: data=0, atomic *ptr=&data

Forbidden by C11: r1=2, r2=1

T0	T1
st(data,1,rlx)	st(data,2,rlx)
r1=ld(*ptr,rlx)	
r2=ld(data,rlx)	

C11/C++11	ARMv7
st(rlx)	STR
ld(rlx)	LDR
ld(acq)	LDR; DMB
...	...

ARM Cortex-A9

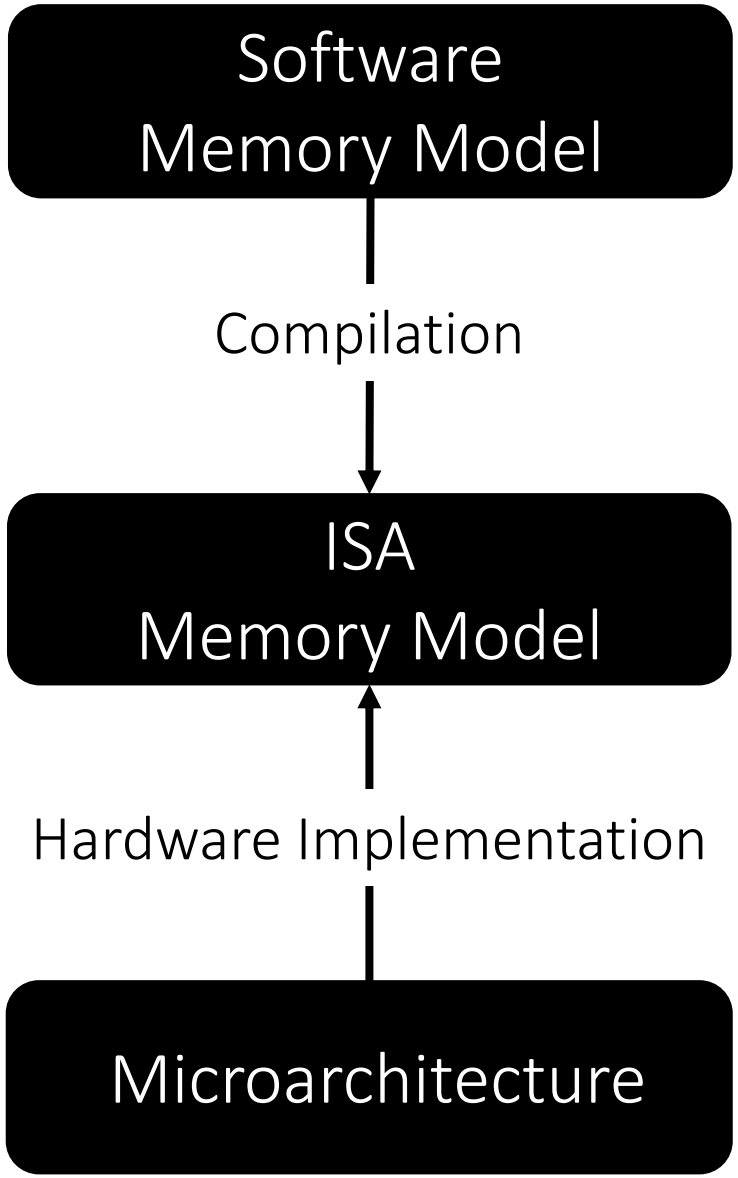
ARM Read-Read Hazard

Initial conditions: data=0, atomic *ptr=&data

Forbidden by C11: r1=2, r2=1

T0	T1
st(data,1,rlx)	st(data,2,rlx)
r1=ld(*ptr,rlx)	
r2=ld(data,rlx)	

Loading data through a pointer



C11/C++11	ARMv7
st(rlx)	STR
ld(rlx)	LDR
ld(acq)	LDR; DMB
...	...

Naïve compilation from C11 to ARMv7

C0	C1
ST [data]←1	ST [data]←2
LD [ptr]→r0	
LD [r0]→r1	
LD [data]→r2	

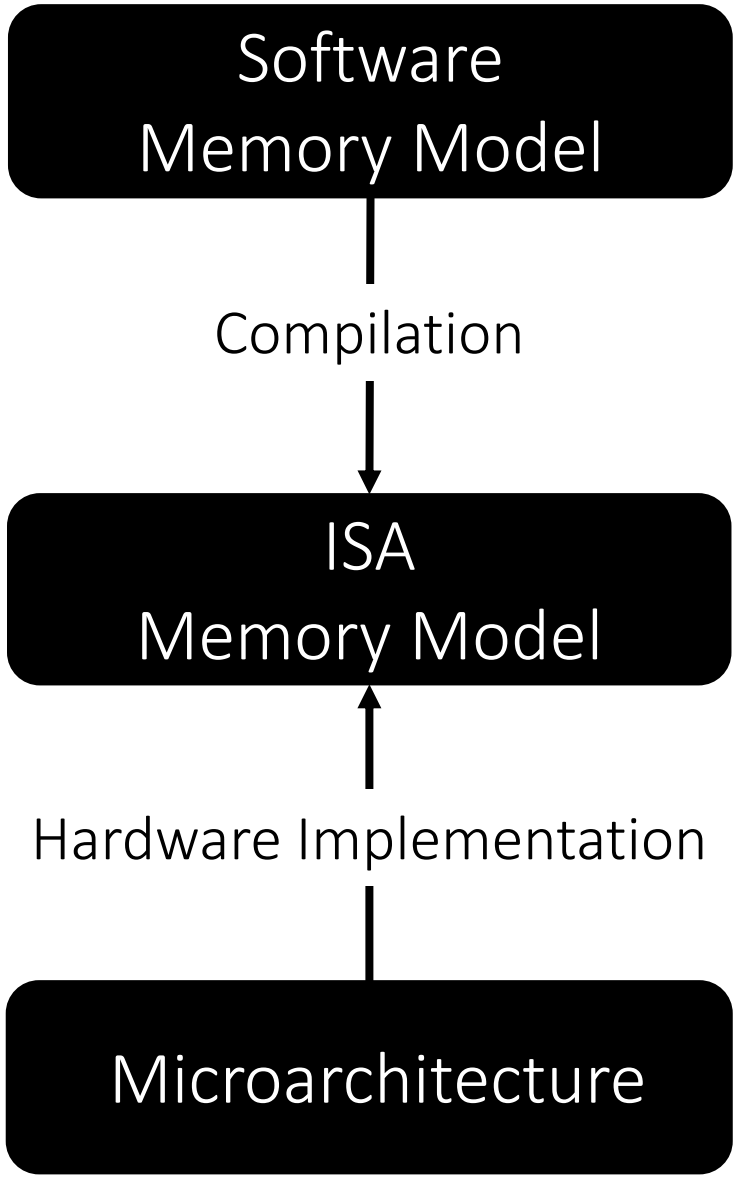
ARM Cortex-A9

ARM Read-Read Hazard

Initial conditions: data=0, atomic *ptr=&data

Forbidden by C11: r1=2, r2=1

T0	T1
st(data,1,rlx)	st(data,2,rlx)
r1=ld(*ptr,rlx)	
r2=ld(data,rlx)	



C11/C++11	ARMv7
st(rlx)	STR
ld(rlx)	LDR
ld(acq)	LDR; DMB
...	...

C0	C1
ST [data]←1	ST [data]←2
LD [ptr]→r0	
LD [r0]→r1	
LD [data]→r2	

Two loads of the same address

Forbidden outcome observable on Cortex-A9

ARM Cortex-A9

ARM Read-after-Read Hazard Demo Google Nexus 6 (Snapdragon 805)

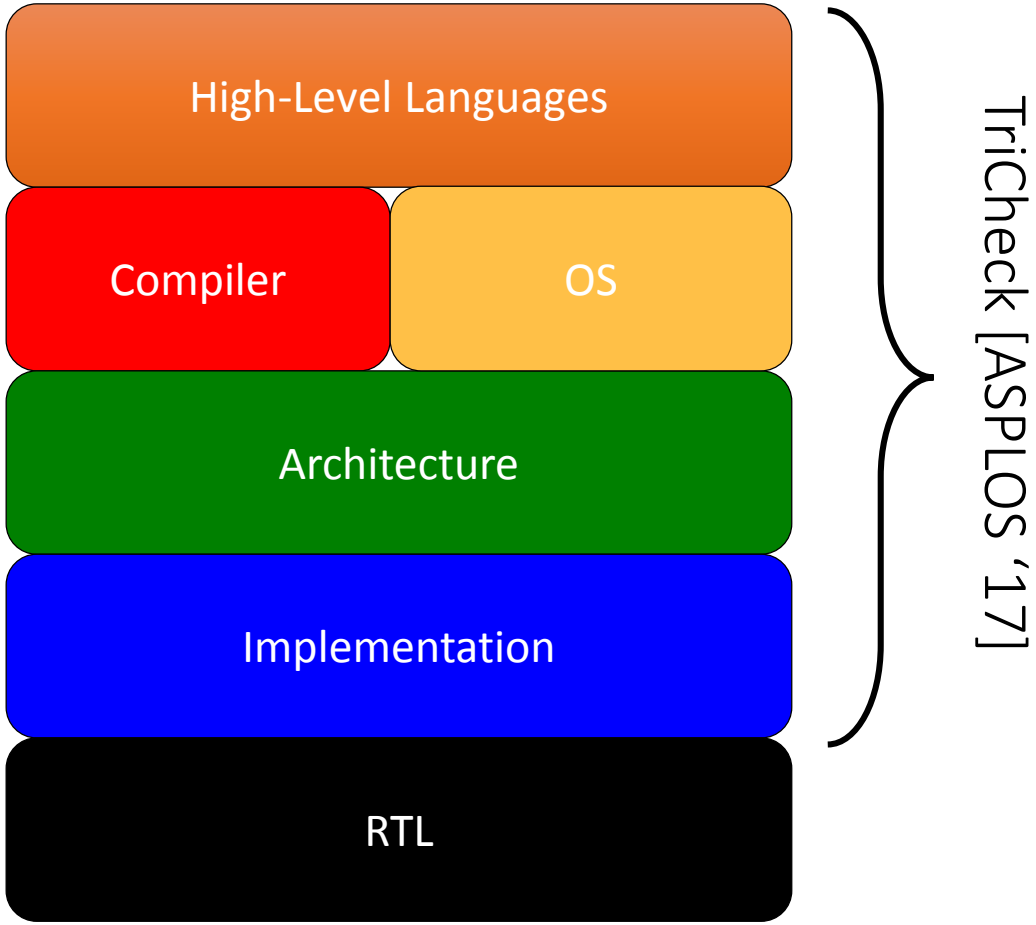
```
std::atomic<int> z = {0};
std::atomic<int> *y = {&z};

void thread0()
{
    z.store(1, std::memory_order_relaxed);
    int r0 = y->load(std::memory_order_relaxed);
    int r1 = z.load(std::memory_order_relaxed);
    if(r0 != r1)
        z.store(3, std::memory_order_relaxed);
}

void thread1()
{
    z.store(2, std::memory_order_relaxed);
}
```

http://check.cs.princeton.edu/tutorial_extras/SnapVideo.mov

Outline

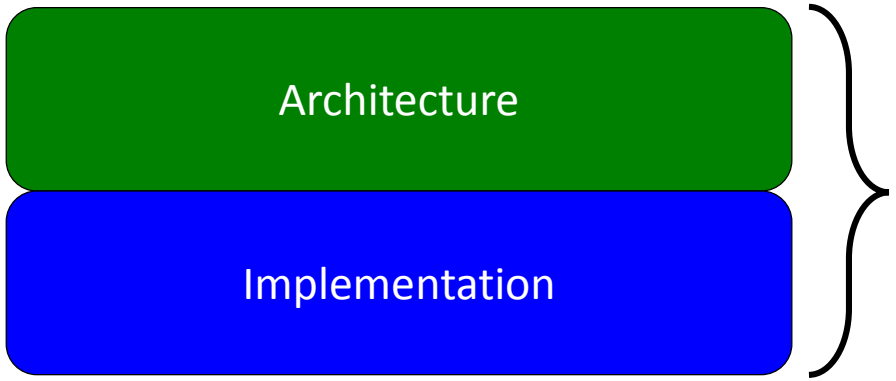
- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, …

Benefits of Full-Stack Verification

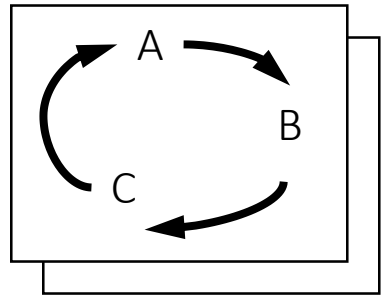


- Categorization & quantification of bugs in the HW-SW stack
  - Effects of ISA MCM issues on the correctness of HLL programs
  - Effects of desirable HW optimizations on ISA-HLL compatibility
- We have found real bugs:
  - RISC-V MCM draft spec
  - Compiler mappings from C11 to Power and ARMv7, leading to discovery of C11 MCM bug

Full-Stack is the Result of a Whole Line of Work

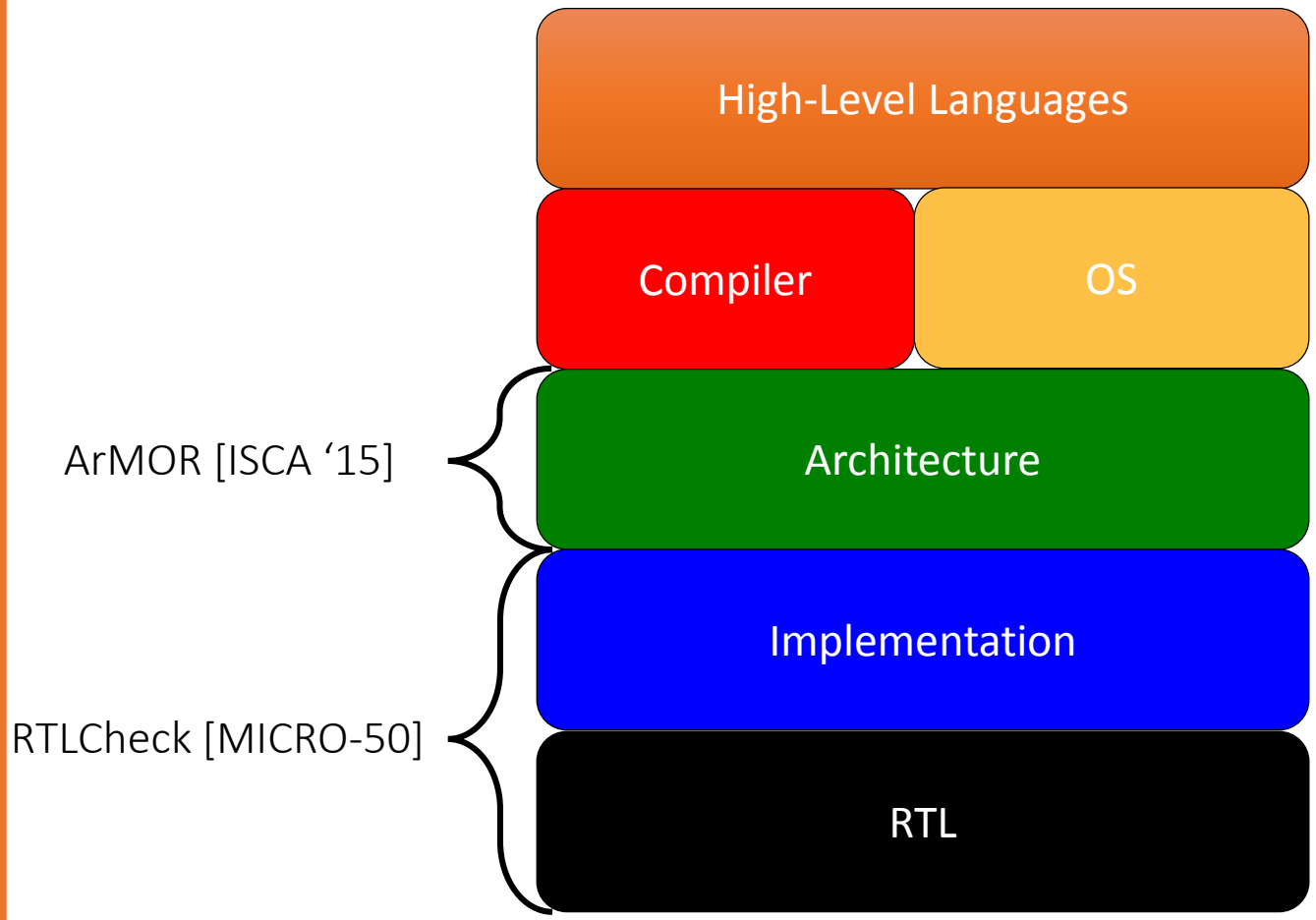


PipeCheck [Micro-47]



- Formal specifications -> Happens-before graphs
- Litmus tests for events of interest
- Check Happens-Before Graphs via Efficient SMT solvers
  - Cyclic => A->B->C->A… Can't happen
  - Acyclic => Scenario is observable
- Fast enough for broad testing

Full Stack is the Result of a Whole Line of Work



ArMOR [ISCA '15]

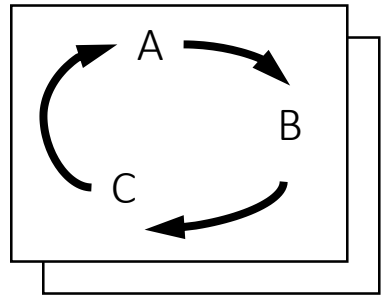
RTLCheck [MICRO-50]

TriCheck [ASPLOS '17]

COATCheck [ASPLOS '16]

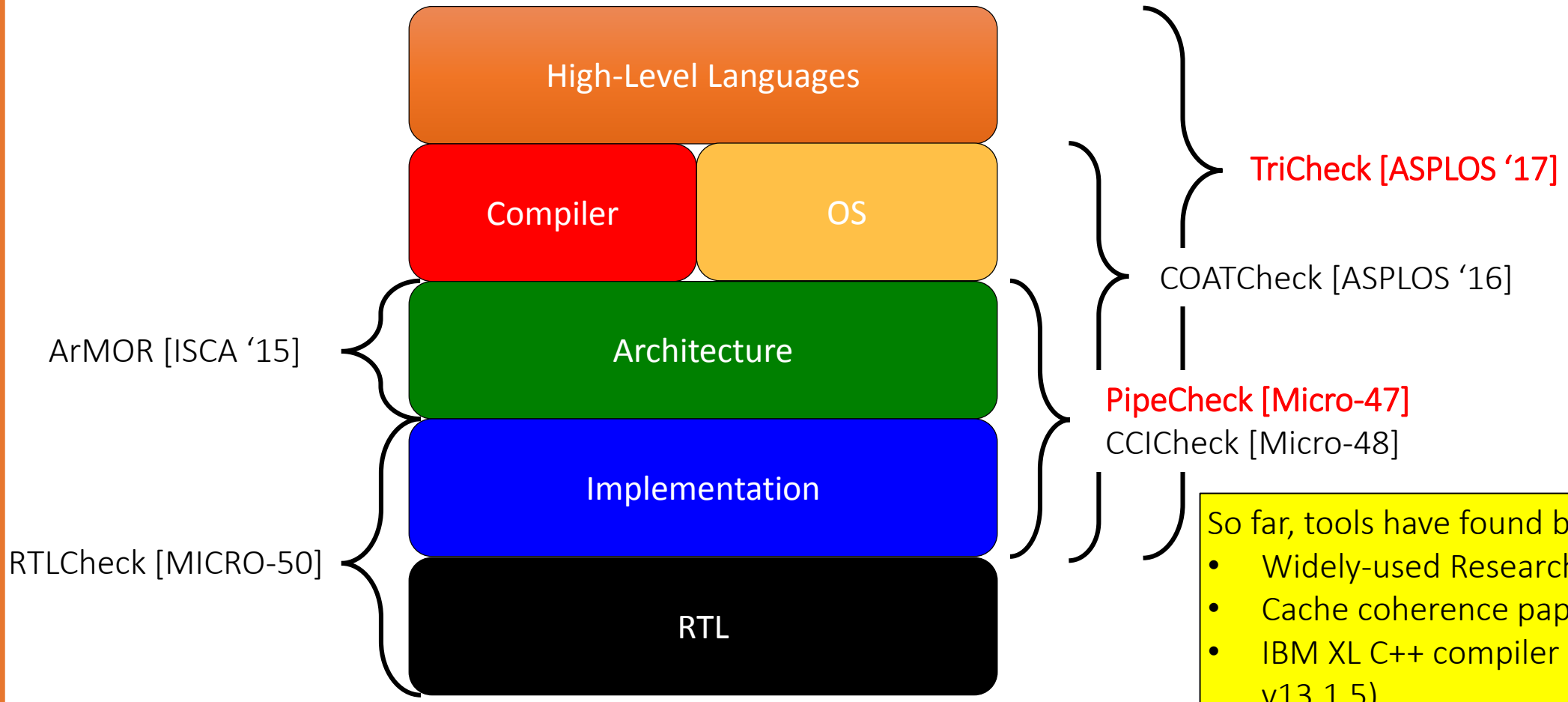
PipeCheck [Micro-47]

CCICheck [Micro-48]



- Formal specifications -> Happens-before graphs
- Litmus tests for events of interest
- Check Happens-Before Graphs via Efficient SMT solvers
  - Cyclic => A->B->C->A… Can't happen
  - Acyclic => Scenario is observable
- Fast enough for broad testing

Full Stack is the Result of a Whole Line of Work



So far, tools have found bugs in:

- Widely-used Research simulator
- Cache coherence paper
- IBM XL C++ compiler (fixed in v13.1.5)
- In-design commercial processors
- RISC-V ISA specification
- Compiler mapping proofs
- C++ 11 mem model

Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, …

Sequential Consistency (SC)

- Defined by [Lamport 1979], execution is the same as if:
  - **(R1)** Memory ops of each processor appear in program order
  - **(R2)** Memory ops of all processors were executed in some global sequential order

Program

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x

Legal Executions

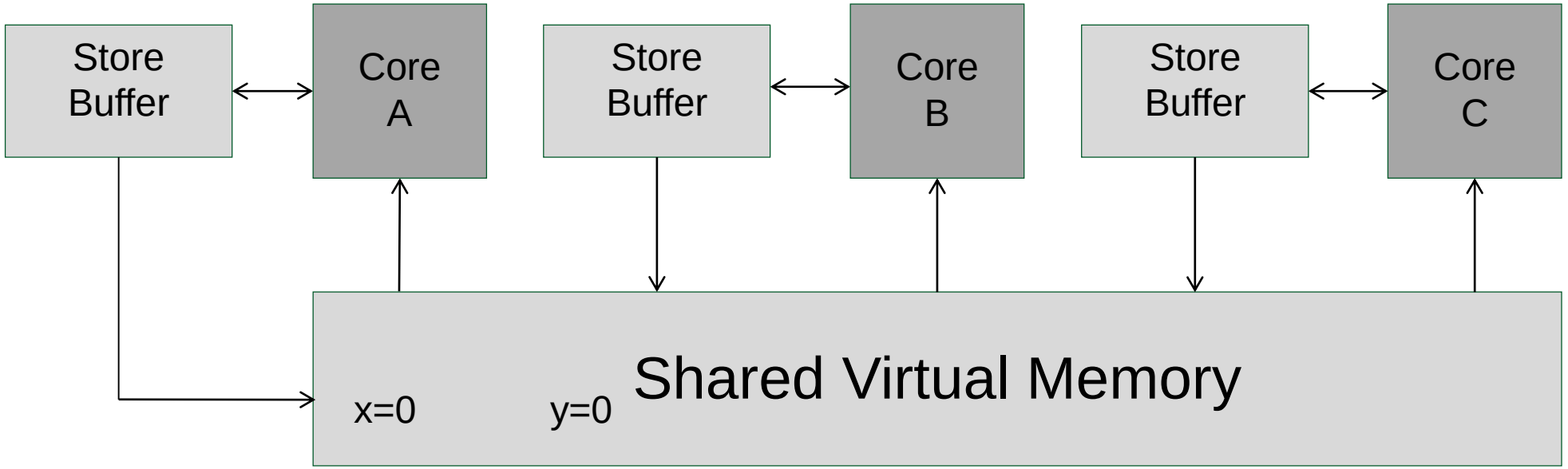
x=1	x=1	x=1	y=1	y=1	y=1
r1=y	y=1	y=1	r2=x	x=1	x=1
y=1	r1=y	r2=x	x=1	r2=x	r1=y
r2=x	r2=x	r1=y	r1=y	r1=y	r2=x

Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x

Second Inst. (i1)

First inst. (i0)	TSO PPO	Ld	St
	Ld	✓	✓
	St		✓



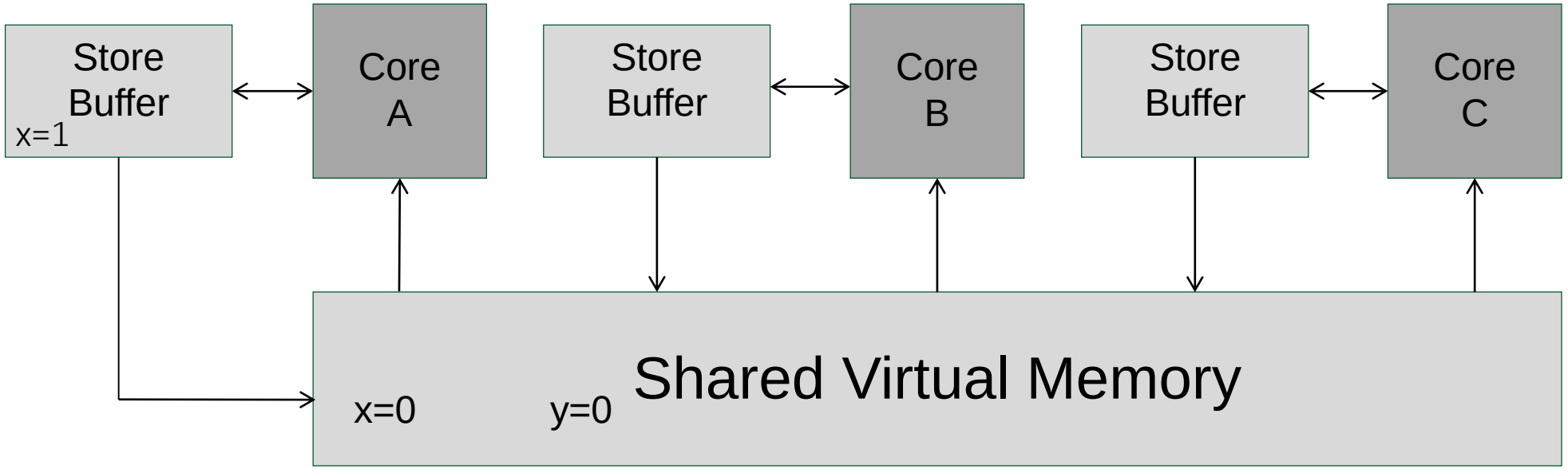
Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x

Second Inst. (i1)

First inst. (i0)

TSO PPO	Ld	St
Ld	✓	✓
St		✓



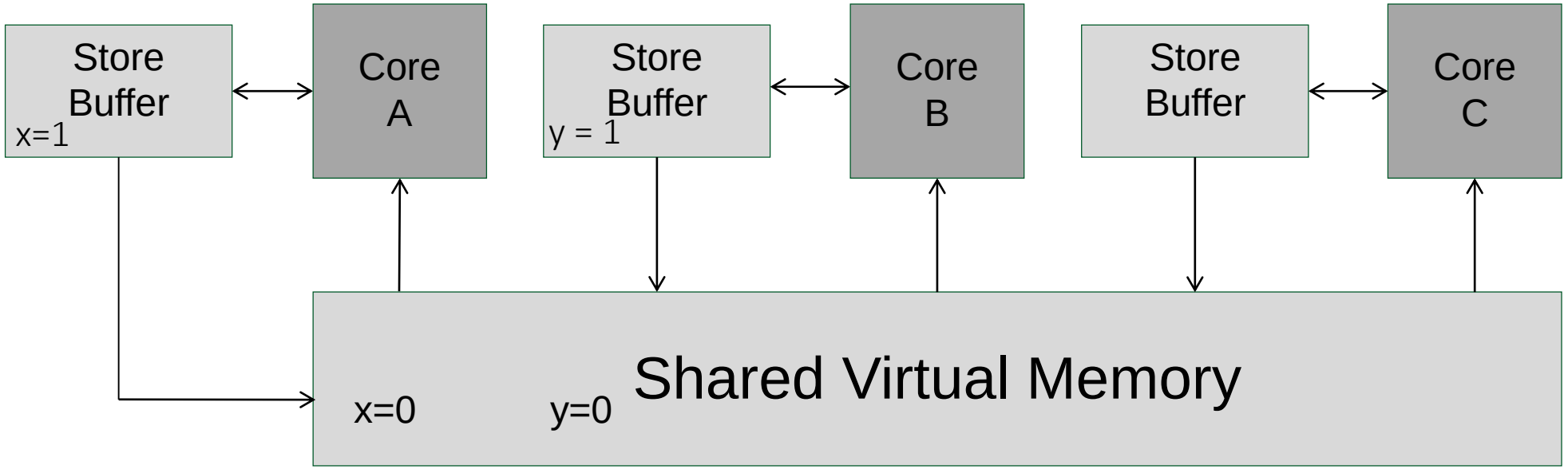
Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x

Second Inst. (i1)

First inst. (i0)

TSO PPO	Ld	St
Ld	✓	✓
St		✓



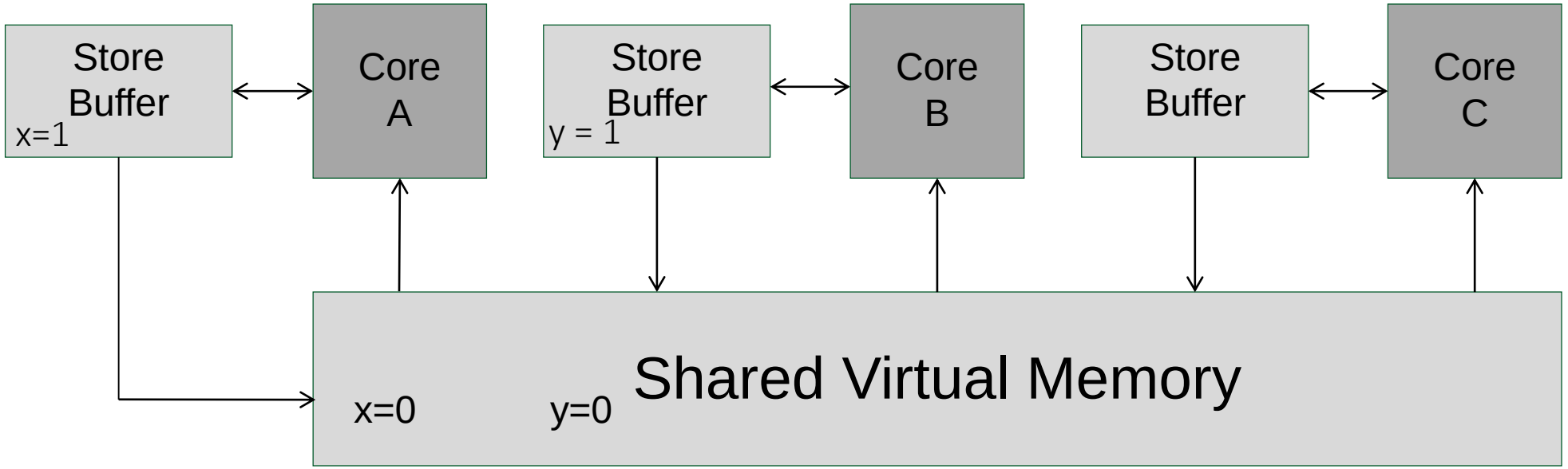
Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x

r1=0

Second Inst. (i1)

First inst. (i0)	TSO PPO	Ld	St
	Ld	✓	✓
	St		✓

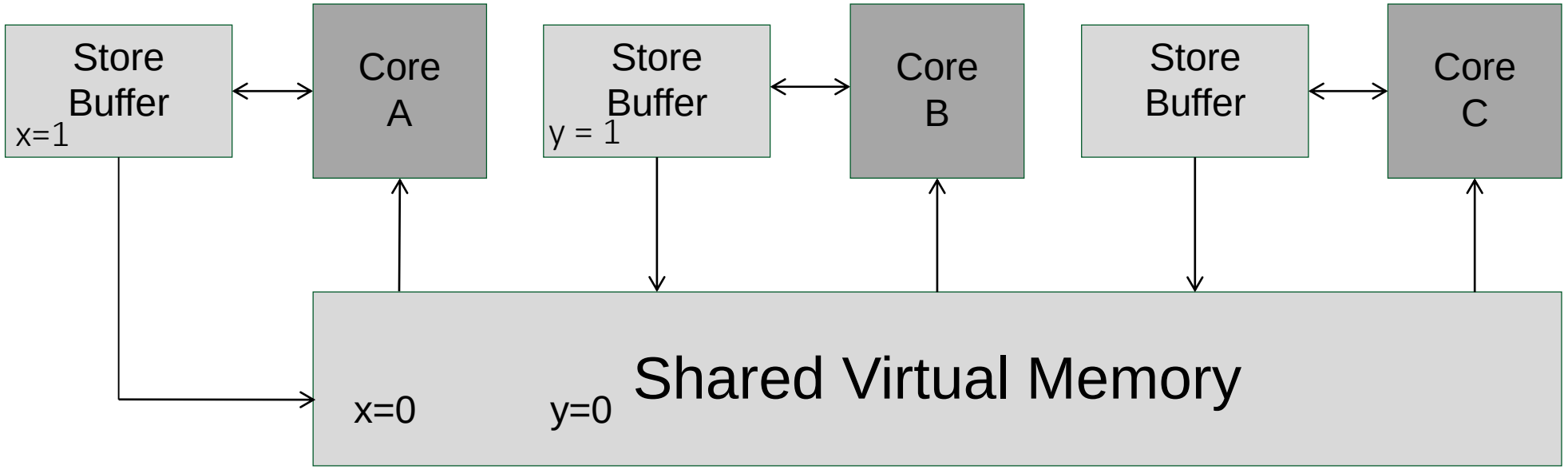


Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x
r1=0	r2=0

Second Inst. (i1)

First inst. (i0)	TSO PPO	Ld	St
	Ld	✓	✓
	St		✓



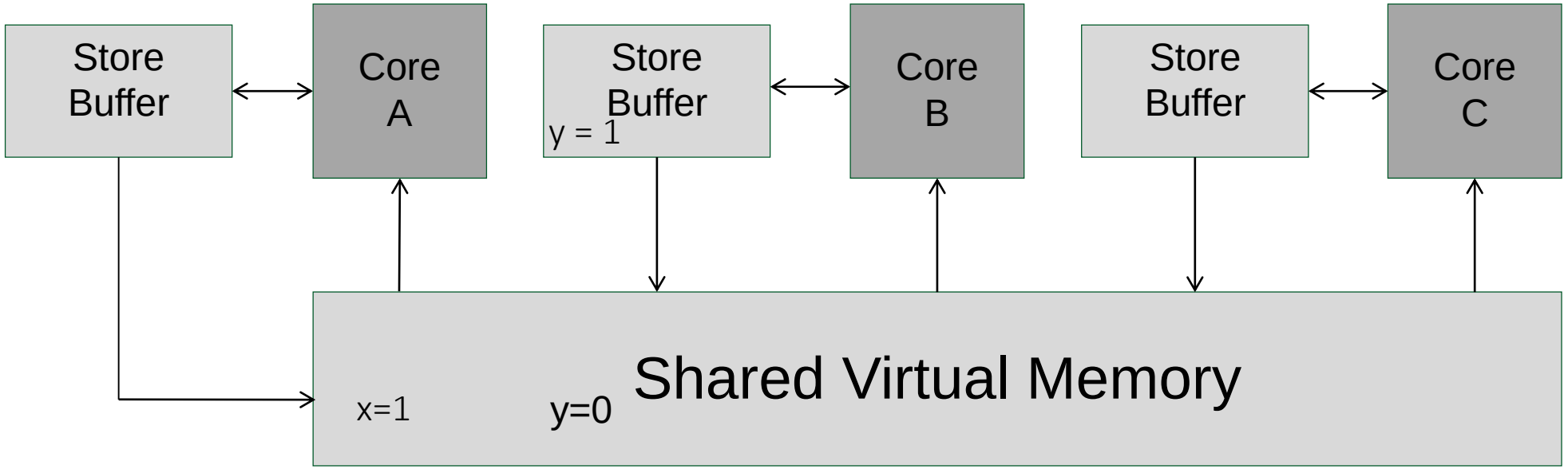
Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x
r1=0	r2=0

Second Inst. (i1)

First inst. (i0)

TSO PPO	Ld	St
Ld	✓	✓
St		✓

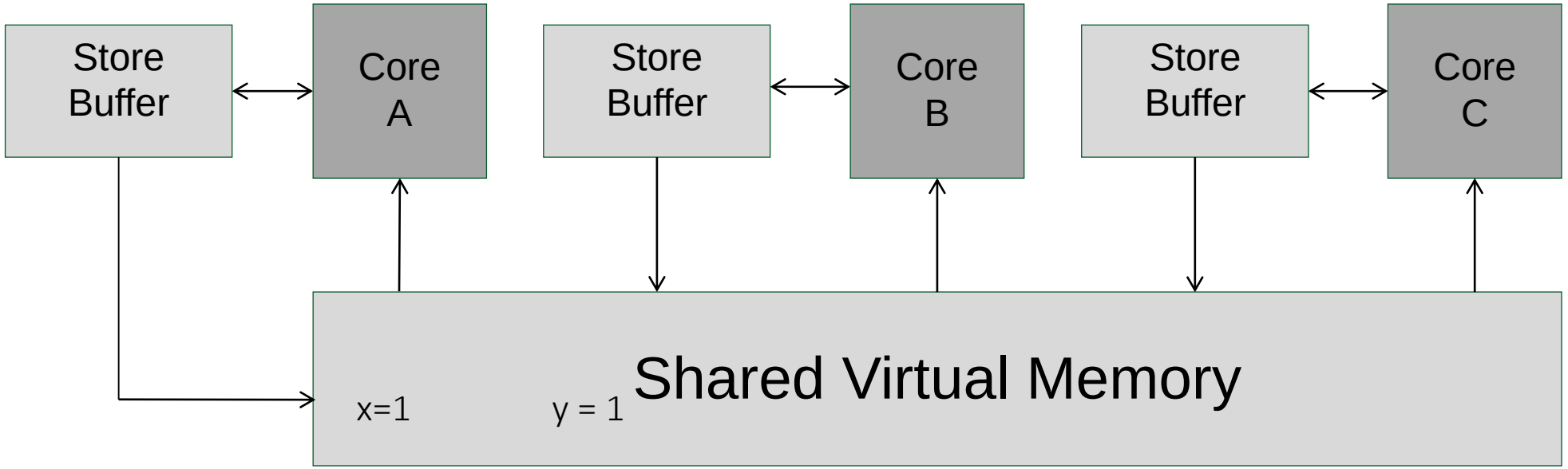


Total Store Order (TSO)

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x
r1=0	r2=0

Second Inst. (i1)

First inst. (i0)	TSO PPO	Ld	St
	Ld	✓	✓
	St		✓



Total Store Order

Program

Thread 0	Thread 1
x=1	y=1
r1=y	r2=x

Legal Executions

x=1 r1=y y=1 r2=x	x=1 y=1 r1=y r2=x	x=1 y=1 r2=x r1=y	y=1 r2=x x=1 r1=y	y=1 x=1 r2=x r1=y	y=1 x=1 r1=y r2=x
r1=y x=1 y=1 r2=x	r1=y y=1 x=1 r2=x	r1=y y=1 r2=x x=1	y=1 r2=x r1=y x=1	y=1 r1=y r2=x x=1	y=1 r1=y x=1 r2=x
x=1 r1=y r2=x y=1	x=1 r2=x r1=y y=1	x=1 r2=x y=1 r1=y	r2=x y=1 x=1 r1=y	r2=x x=1 y=1 r1=y	r2=x x=1 r1=y y=1
r1=y x=1 r2=x y=1	r1=y r2=x x=1 y=1	r1=y r2=x y=1 x=1	r2=x y=1 r1=y x=1	r2=x r1=y y=1 x=1	r2=x r1=y x=1 y=1

Memory Consistency Models: Critical ISA & System Component

8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (**Note** the memory-ordering principles for single-processor and multiple-processor systems are written from the perspective of software executing on the processor, where the term "processor" refers to a logical processor. For example, a physical processor supporting multiple cores and/or HyperThreading Technology is treated as a multi-processor systems.):

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes, with the following exceptions:
  - writes executed with the CLFLUSH instruction;
  - streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD); and
  - string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.

...

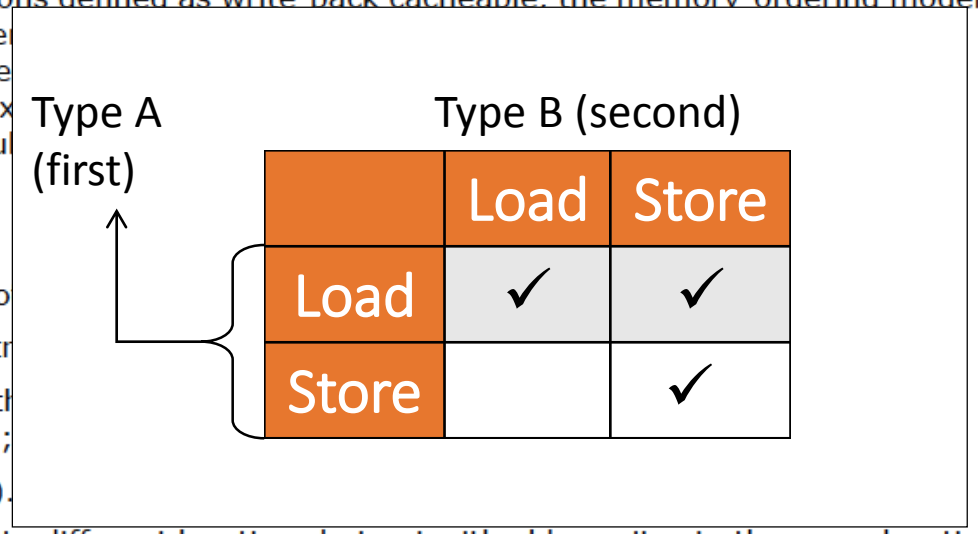
Memory Consistency Models: Critical ISA & System Component

8.2.2 Memory Ordering in P6 and More Recent Processor Families

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and P6 family processors also use a processor-ordered memory-ordering model that can be further defined as "write ordered with store-buffer forwarding." This model can be characterized as follows.

In a single-processor system for memory regions defined as write-back cacheable, the memory-ordering model respects the following principles (**Note** the me
processor systems are written from the perspe
"processor" refers to a logical processor. For ex
HyperThreading Technology is treated as a mu

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with o
  - writes executed with the CLFLUSH instr
  - streaming stores (writes) executed wit
    MOVNTDQ, MOVNTPS, and MOVNTPD);
  - string operations (see Section 8.2.4.1).
- Reads may be reordered with older writes to different locations but not with older writes to the same location.



...

Memory Consistency Models: Critical ISA & System Component

8.2.2 Memory Ordering in P6 and More Recent Processor Fa

The Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, and
memory-ordering model that can be further defined as "
can be characterized as follows.

In a single-processor system for memor
respects the following principles (
processor systems are writ
"processor" refers t
HyperThreadi

- R

- Re

with older writes to the same location.

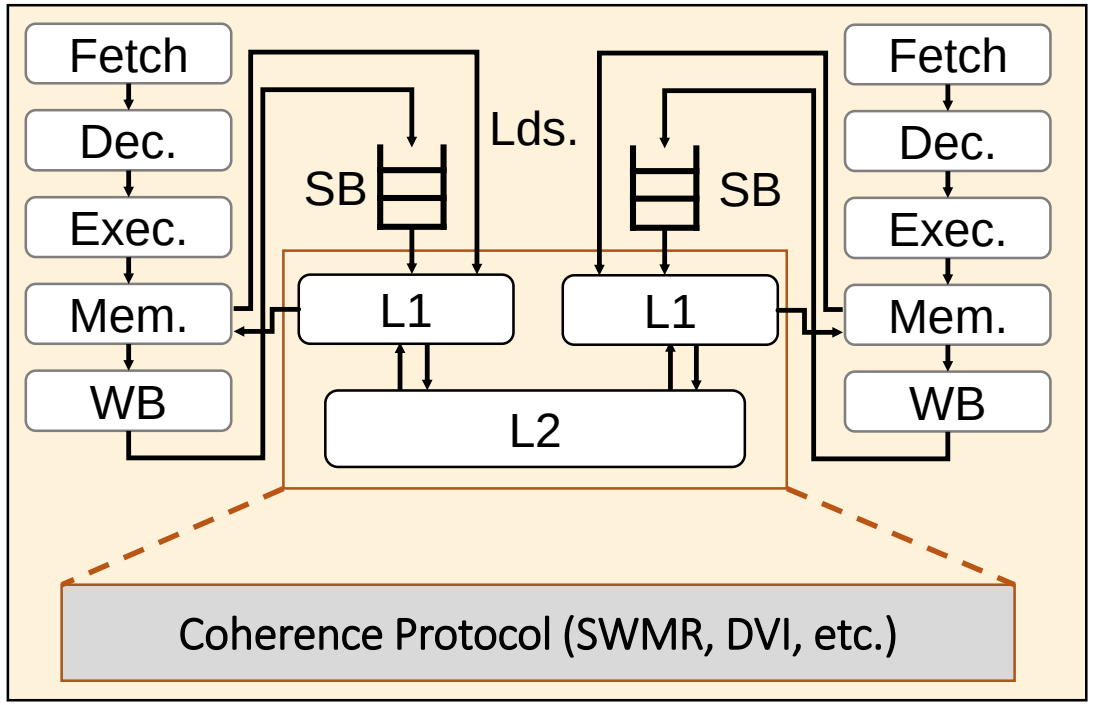
...

In a nutshell: MCMs are part of the ISA

- They are the spec of what value will be returned when your program does a load
- Rest of the software stack expects them to be implemented correctly

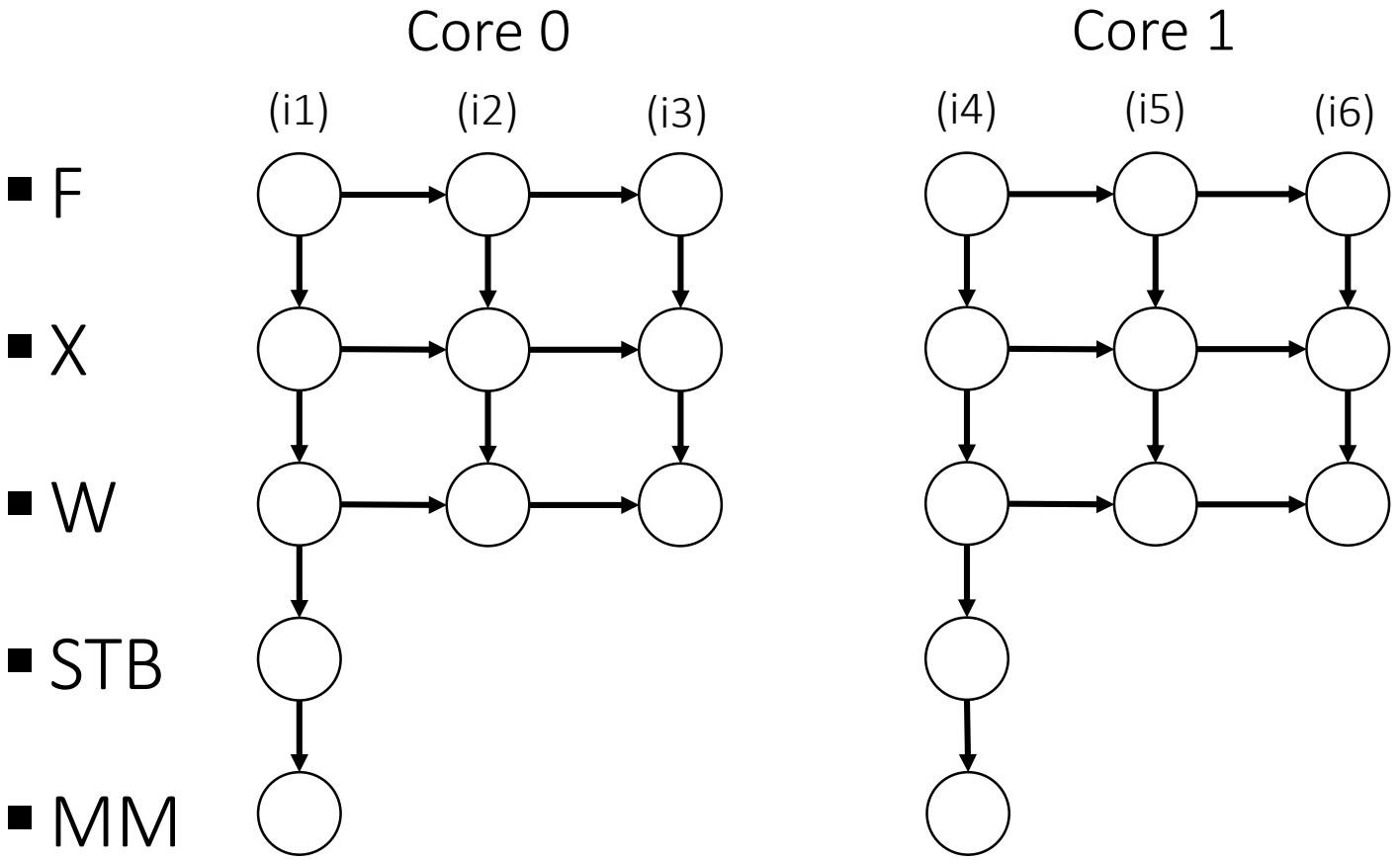
Microarchitectural Consistency Verification

- Microarch. enforces ISA-level MCM through many small orderings
  - In-order fetch/commit
  - FIFO store buffers
  - Coherence protocol
  - …
- Difficult to ensure that these orderings ***always*** enforce the required orderings



- Designs may also be complicated by optimizations (**speculative load reordering, early fence retirement, OoO execution**), or novel organization (**heterogeneity**)

Our approach: "microarchitecturally happens-before" (μhb) graphs



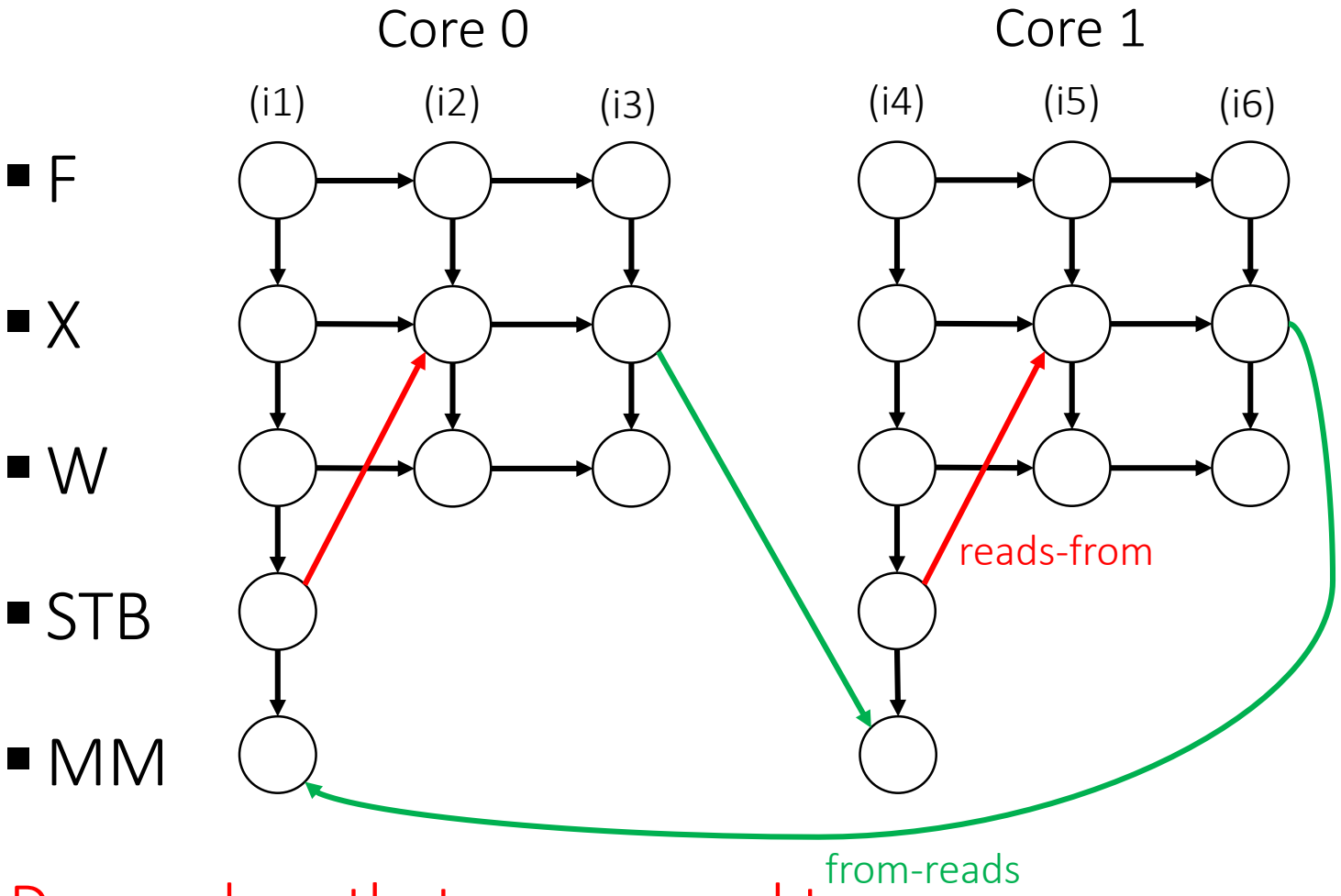
Initially: [x]=[y]=0	
Core 0	Core 1
(i1) st [x] ← 1 (i2) ld [x] → r1 (i3) ld [y] → r2	(i4) st [y] ← 1 (i5) ld [y] → r3 (i6) ld [x] → r4
Program outcome of interest: r1=1, r2=0, r3=1, and r4=0	

Key Idea: Model executions of programs on HW as **μhb graphs**

- **Nodes:** Microarchitectural events in an execution
- **Edges:** Happens-before relationships between nodes

1. Draw edges that correspond to outcome-independent orderings

Our approach: "microarchitecturally happens-before" (μhb) graphs



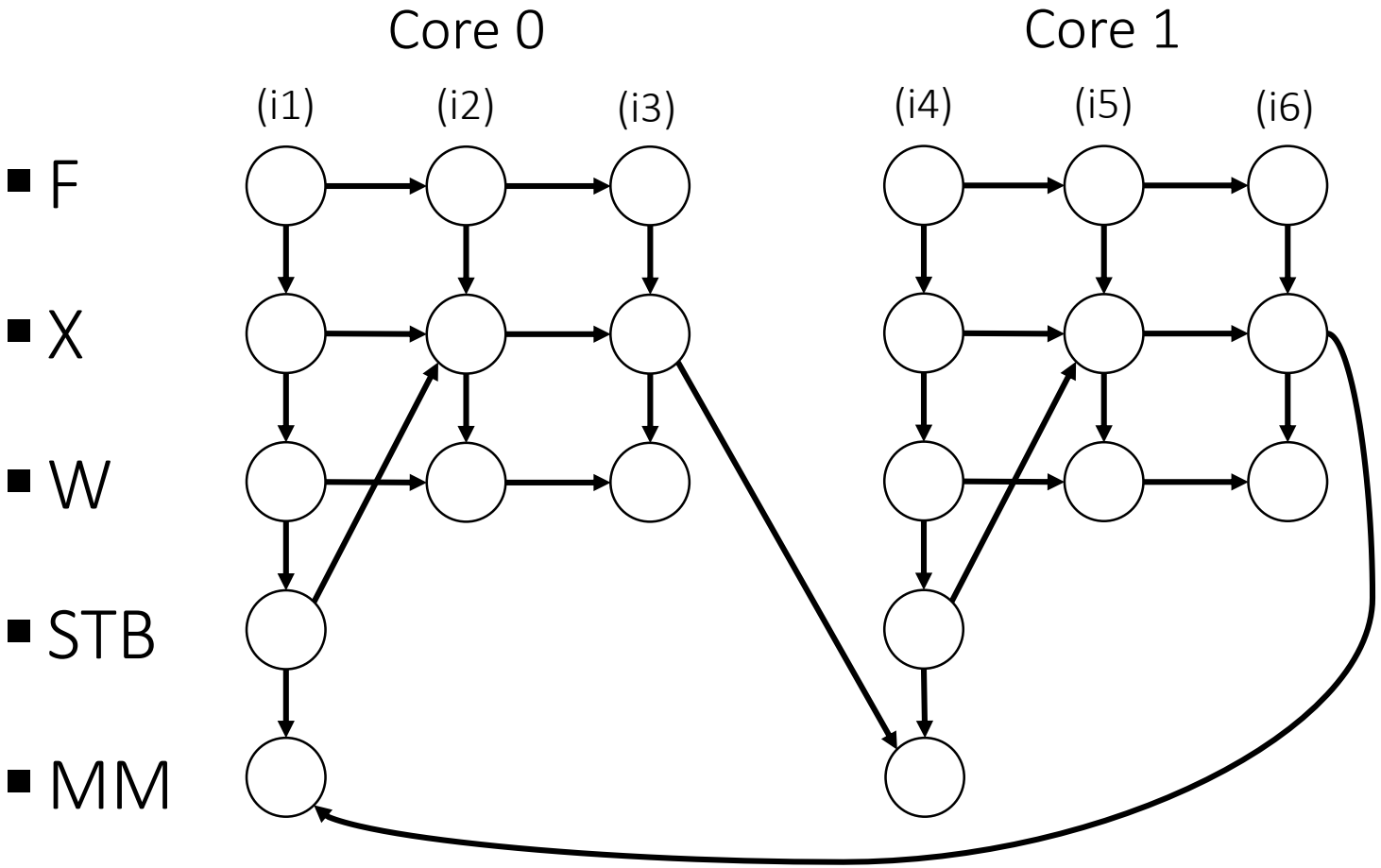
Initially: [x]=[y]=0	
Core 0	Core 1
(i1) st [x] ← 1 (i2) ld [x] → r1 (i3) ld [y] → r2	(i4) st [y] ← 1 (i5) ld [y] → r3 (i6) ld [x] → r4
Program outcome of interest: r1=1, r2=0, r3=1, and r4=0	

Key Idea: Model executions of programs on HW as **μhb graphs**

- **Nodes:** Microarchitectural events in an execution
- **Edges:** Happens-before relationships between nodes

2. Draw edges that correspond to outcome-dependent orderings

Our approach: "microarchitecturally happens-before" (μhb) graphs



Initially: [x]=[y]=0	
Core 0	Core 1
(i1) st [x] ← 1 (i2) ld [x] → r1 (i3) ld [y] → r2	(i4) st [y] ← 1 (i5) ld [y] → r3 (i6) ld [x] → r4
Program outcome of interest: r1=1, r2=0, r3=1, and r4=0	

Key Idea: Model executions of programs on HW as **μhb graphs**

- **Nodes:** Microarchitectural events in an execution
- **Edges:** Happens-before relationships between nodes

No cycle in graph, so program outcome is observable!

Litmus test verification

- Litmus tests – small parallel programs
  - Used to highlight memory model differences/features
  - Typically there is one non-SC outcome of interest
- Different litmus tests associated with different ISA models
  - ISA memory model often characterized by their Permitted and Forbidden non-SC litmus test outcomes
  - TSO litmus test suite, Power litmus test suite, ARM litmus test suite
- Why litmus test verification?
  - Higher performance when evaluating complex designs
  - Enables us to have a fast, iterative design process
  - Focus on verification cases most likely to exhibit bugs

Litmus test verification (for TSO)

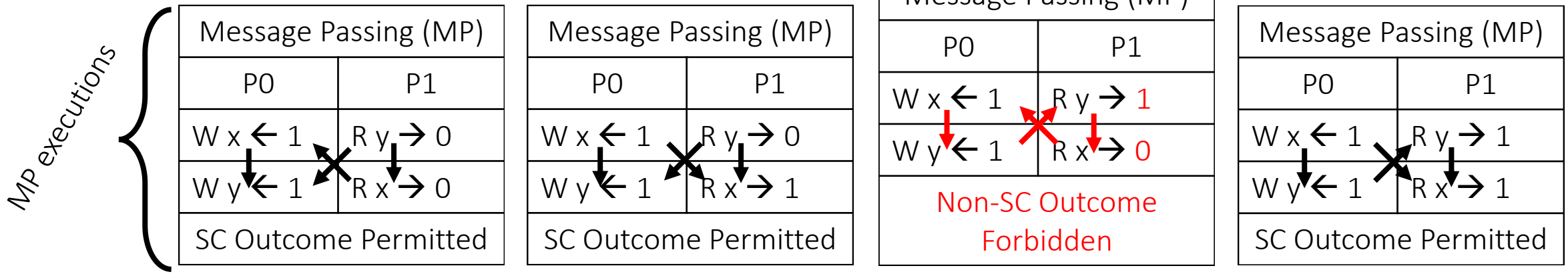
Many litmus tests have been developed over the years; they have names e.g., MP, SB

Initial conditions are all 0 unless otherwise stated

This tutorial: we use a sprinkling of established tests

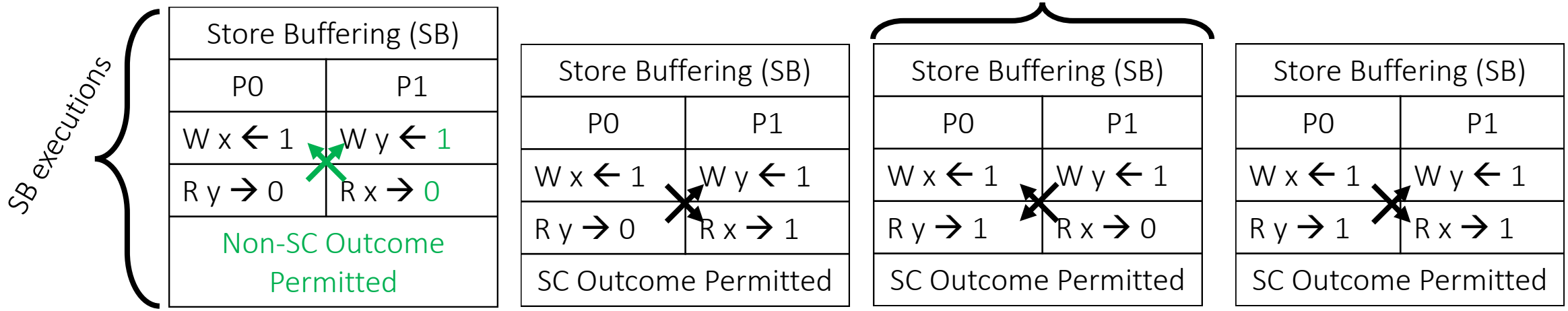
MP executions

MP litmus test



SB executions

SB litmus test



Compare ISA Executions with Hardware Executions

- At the ISA level a litmus test outcome can be:
  - Permitted
  - Forbidden
- **Our approach:** At the hardware level a litmus test outcome can be:
  - Observable
  - Unobservable

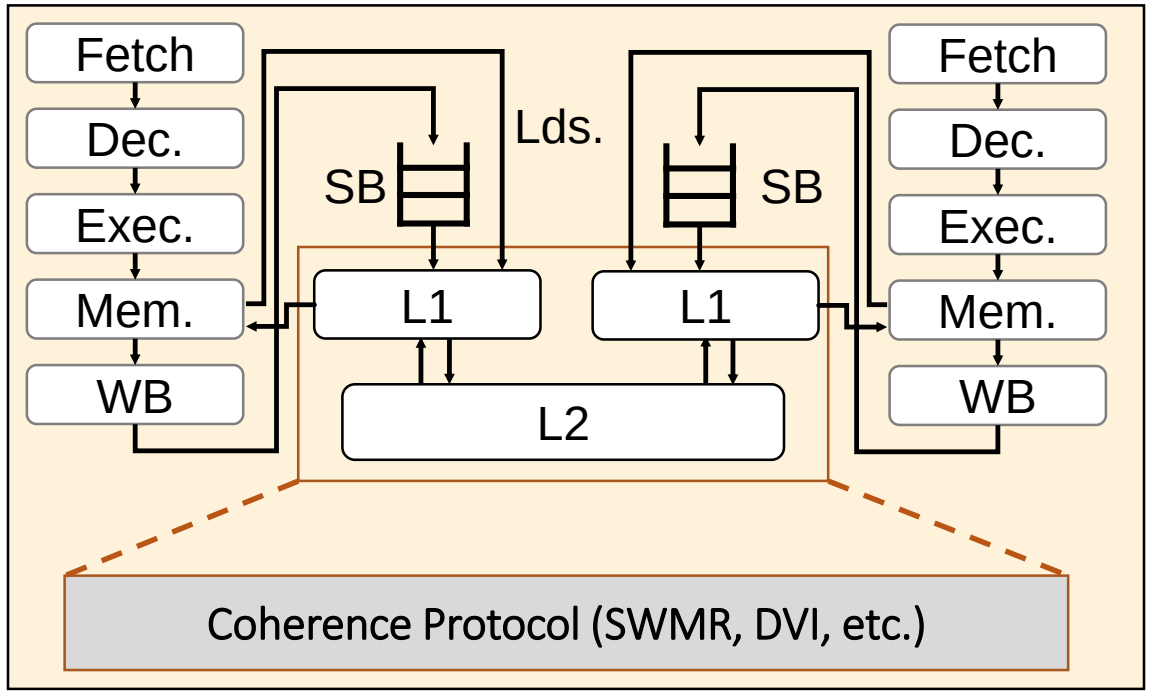
What ISA level analysis tells us

What our hardware-level analysis tells us

	Observable	Unobservable
Permitted	OK	OK (Stricter than necessary)
Forbidden	BUG	OK

Does hardware correctly implement memory model?

Microarchitecture



Instruction level analysis

Our analysis

	Observable	Unobservable
Permitted	OK	OK
Forbidden	BUG	OK

?

Permitted AND Observable
OR
Permitted AND Unobservable
OR
Forbidden AND Unobservable

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	

Check Inputs: Microarchitecture Spec + Litmus Tests

Microarchitecture Specification in *μspec* DSL

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

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
 EdgeExists ((i1, Fetch), (i2, Fetch)) =>
   AddEdge ((i1, Execute), (i2, Execute), "PPO").
```

Refer to Quick Start Guide for more information on the μSpec DSL and how to write axioms.

+

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	

?

Permitted AND Observable
OR
Permitted AND Unobservable
OR
Forbidden AND Unobservable

Check Inputs: Microarchitecture Spec + Litmus Tests

Microarchitecture Specification in *μSpec* DSL

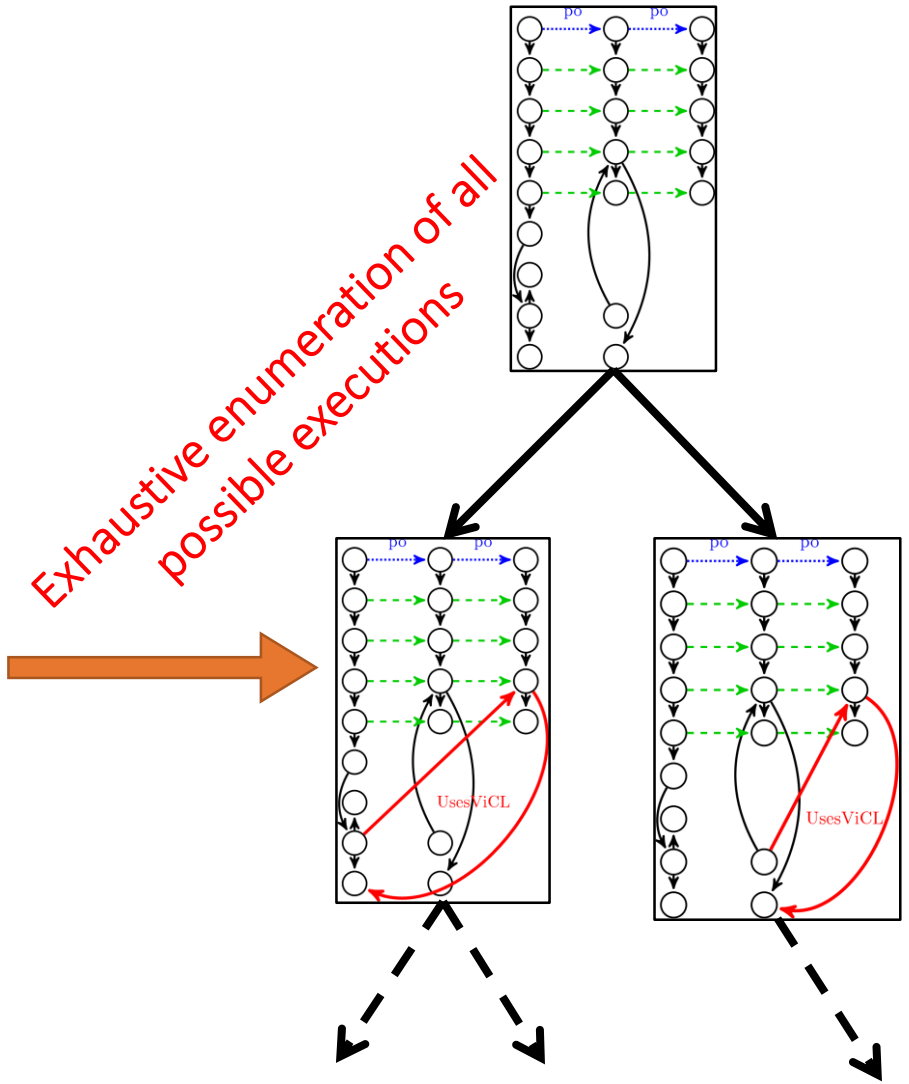
```
Axiom "PO_Fetch":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ ProgramOrder i1 i2 =>
  AddEdge ((i1, Fetch), (i2, Fetch), "PO").

Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
 EdgeExists ((i1, Fetch), (i2, Fetch)) =>
   AddEdge ((i1, Execute), (i2, Execute), "PPO").
```

+

Litmus Test

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

Exhaustive enumeration of all possible executions



Microarchitectural happens-before (μhb) graphs

In a nutshell, our tool philosophy…

- Automate specification, verification, and translation related to MCMs
- Comprehensive exploration of ordering possibilities
- Key Techniques: Happens-before Graphs and SMT solvers
- Bounded, litmus-test driven verificatiom
  - We have other related techniques for whole-program, more comprehensive design analysis
  - And we have templates to assist in the automatic generation of “families” of litmus tests
- Verificaiton conducted on and axiomatic model of hardware
  - We have tools for verifying this model is representative of real RTL

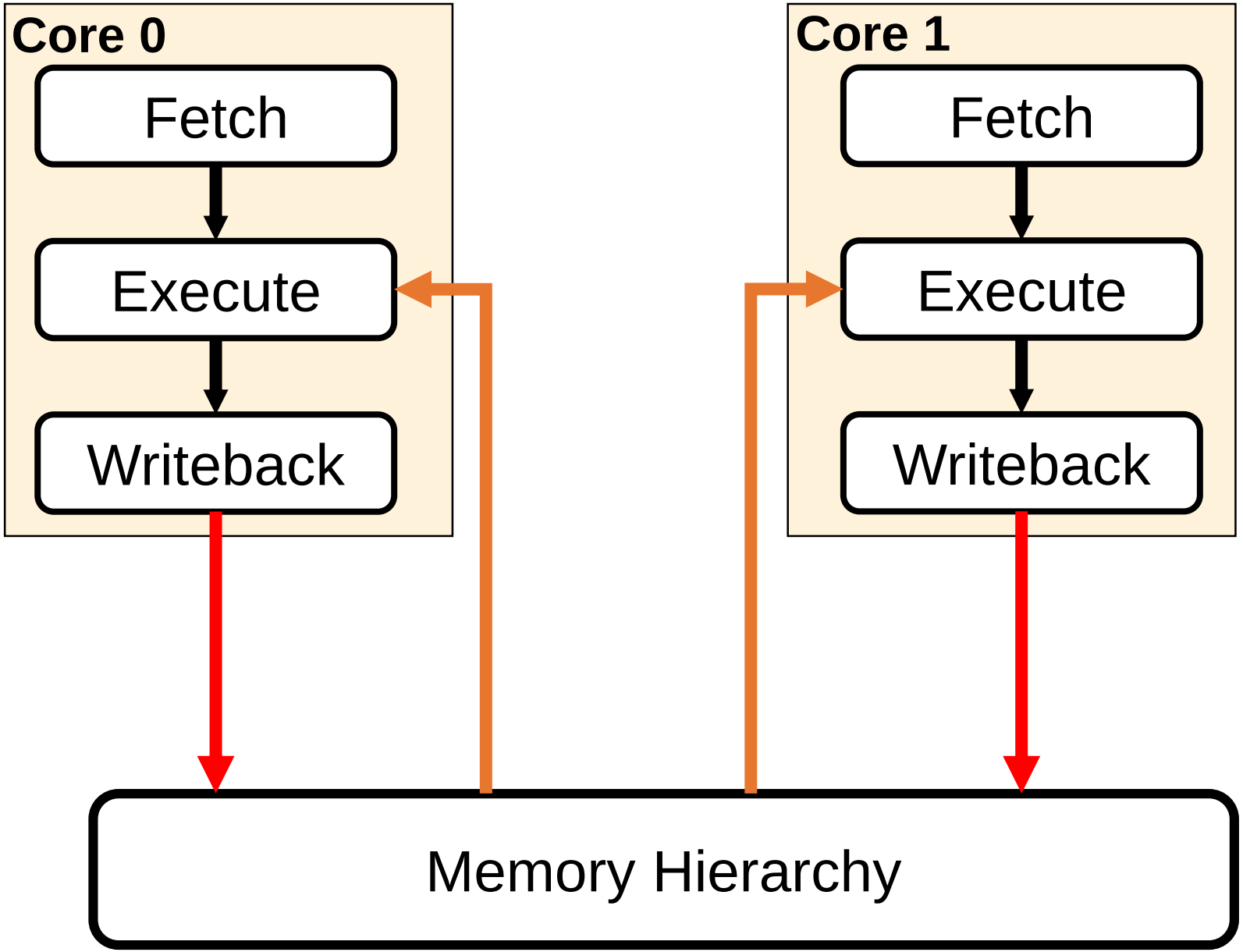
Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, …

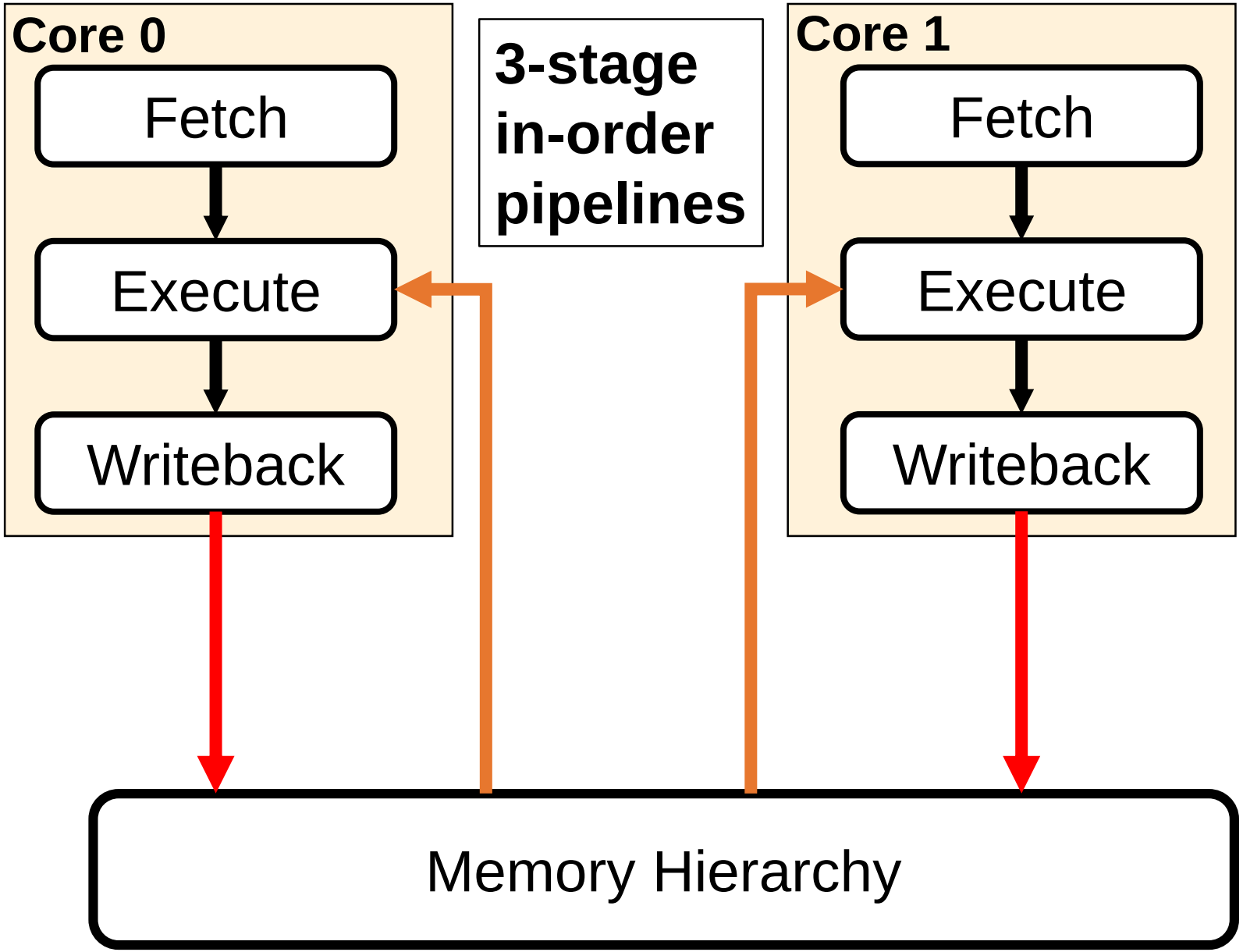
Overview

- Modelling simple microarchitectures (μarches) in μSpec
  - Give you a taste of what μSpec can model and reason about
- Begin by modelling a SC μarch
  - Partially completed μarch in VM, you will fill in remainder
- Post-coffee break, will look at expanding this SC μarch to TSO
  - Store buffers
  - Reading own write early (time permitting)
  - Fences (time permitting)

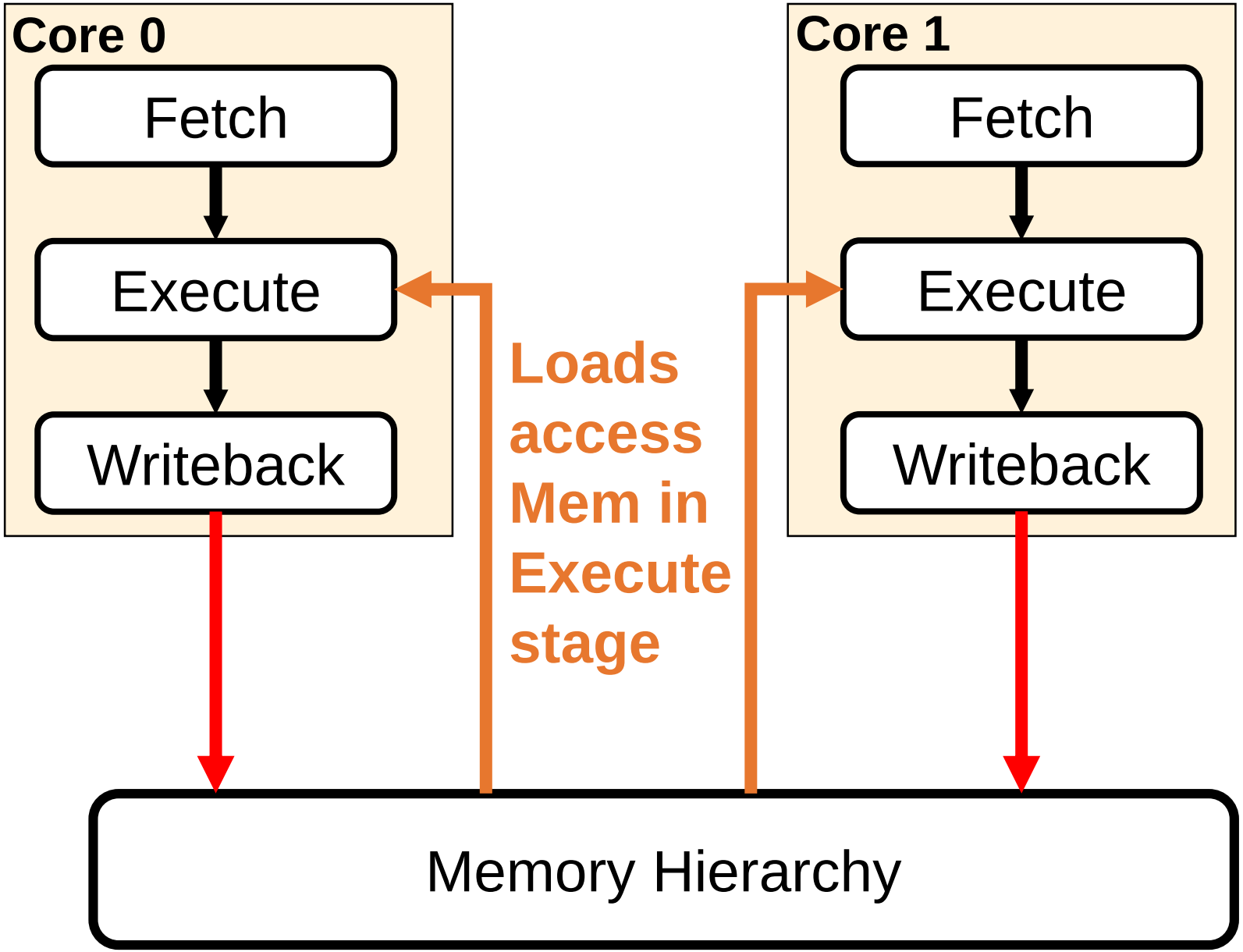
Specifying a Simple (SC) Microarch. in μSpec



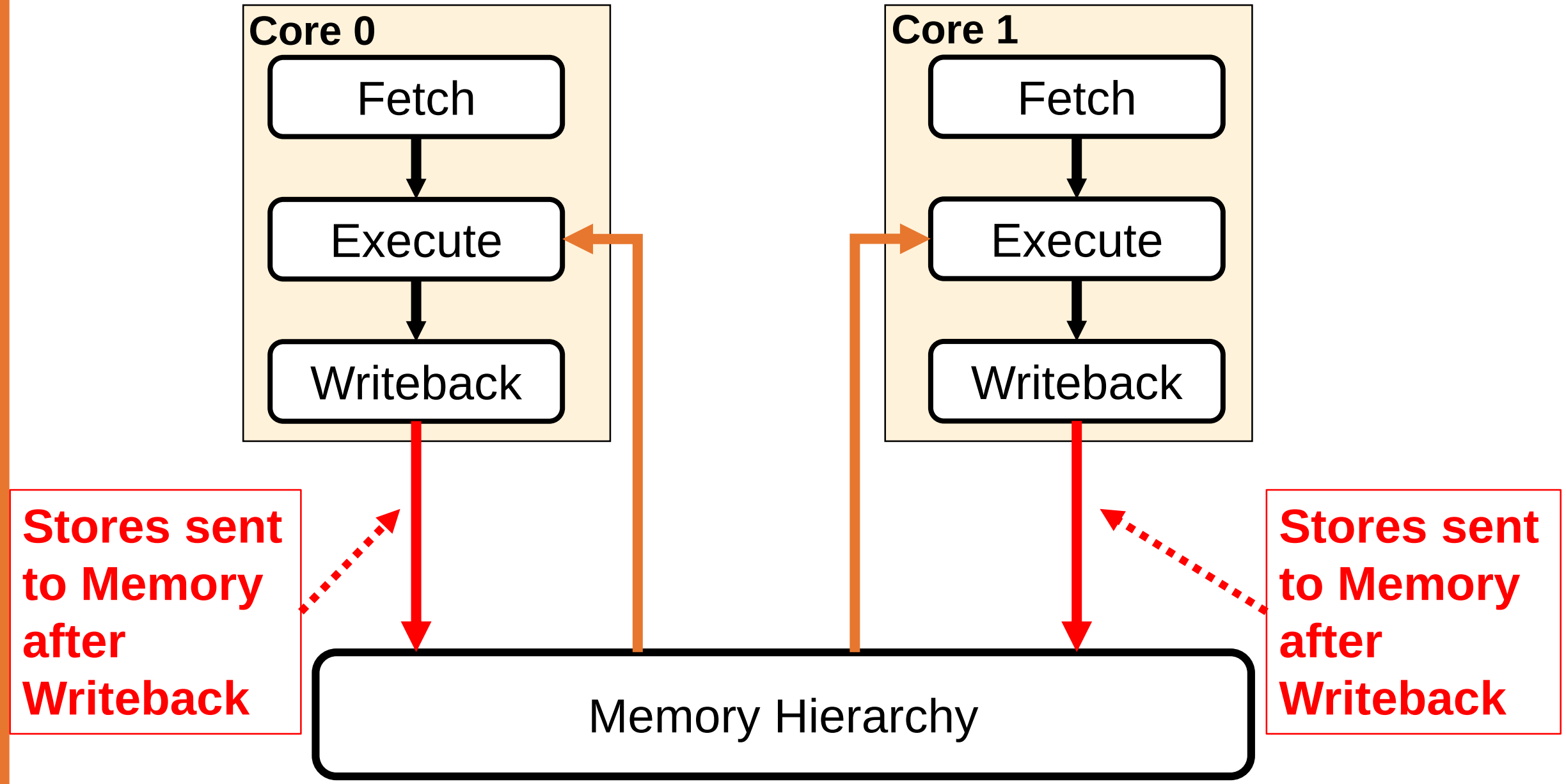
Specifying a Simple (SC) Microarch. in μSpec



Specifying a Simple (SC) Microarch. in μSpec



Specifying a Simple (SC) Microarch. in μSpec

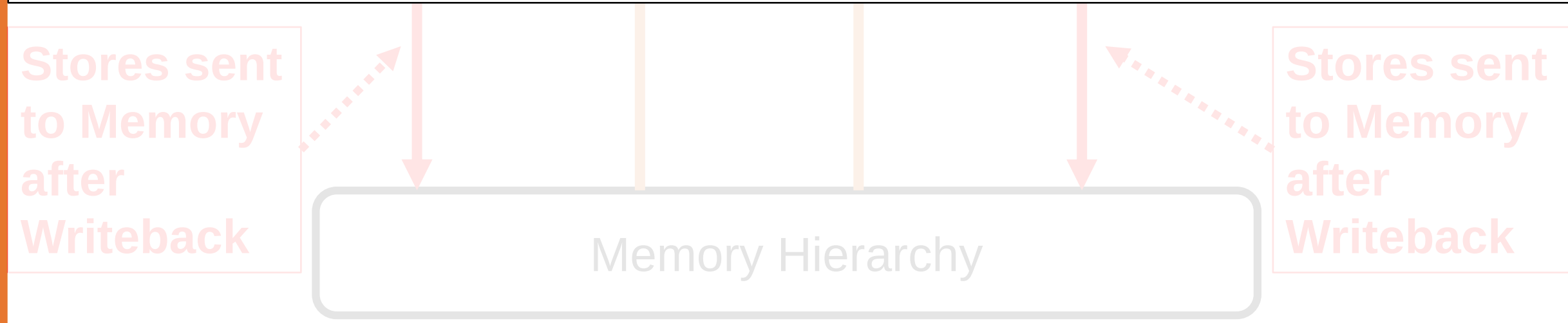


Specifying a Simple (SC) Microarch. in μSpec

1. **Start VirtualBox VM**

2. **Open a Terminal**

3. **Partially completed SC uarch in**
`/home/check/pipecheck_tutorial/uarches/SC_fillable.uarch`
`(i.e. ~/pipecheck_tutorial/uarches/SC_fillable.uarch)`



μSpec: A DSL for Specifying Microarchitectures

- Language has capabilities similar to first-order logic
  - forall, exists, AND (/\), OR (\/), NOT (~), implication (=>)
- Microarchitecture spec is a set of **axioms**
  - Each axiom enforces a ***partial ordering*** on execution events…
  - …which correspond to individual smaller microarchitectural orderings!
  - Eg: Some axioms for pipeline stages, others for coherence…
- Job of PipeCheck is to ensure that axioms correctly work ***together*** to uphold the requirements of the ISA-level MCM
- Axiom writing is an iterative process

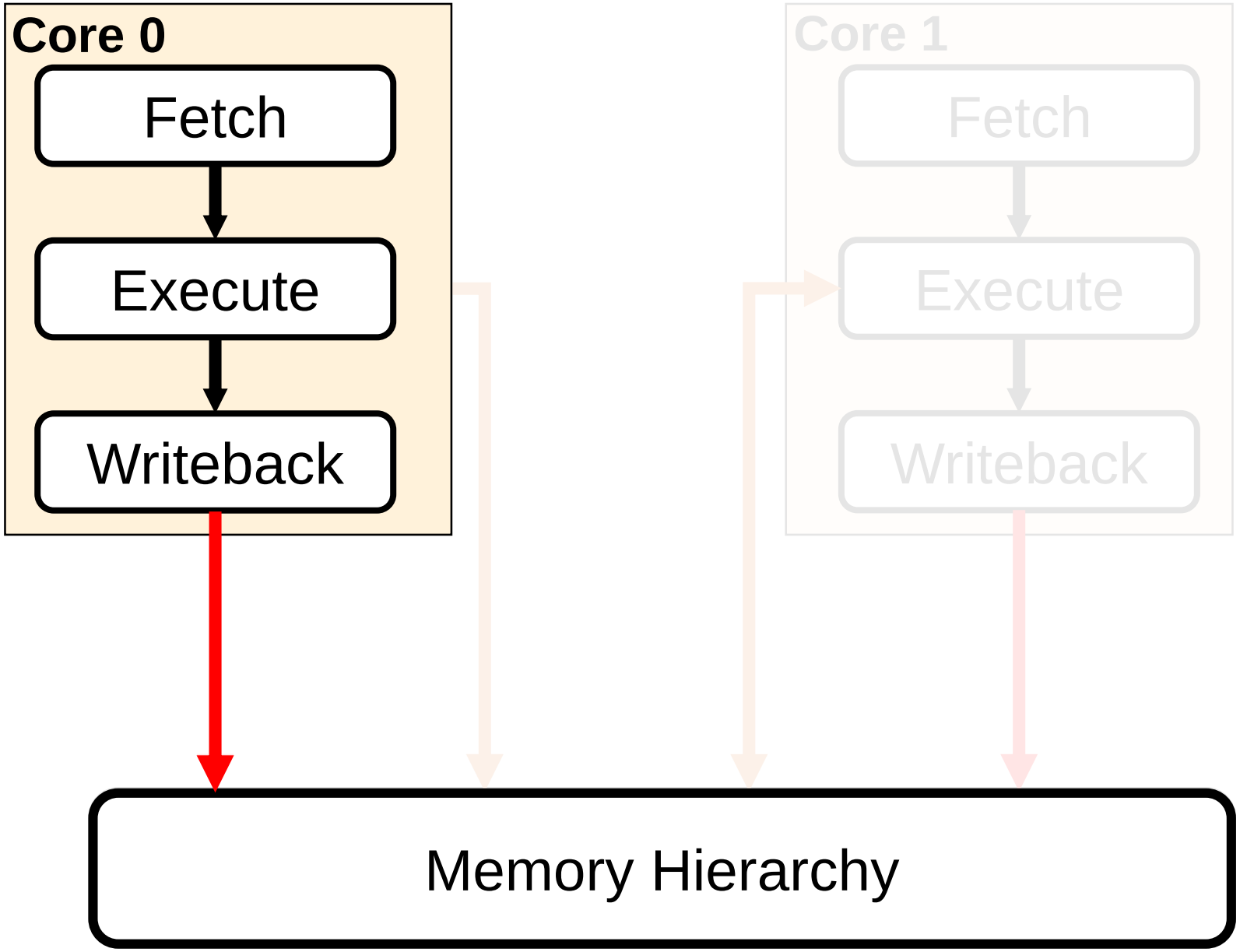
Specifying μSpec Nodes

- A node represents a particular event in a particular instruction's execution
- Eg: `(i, Fetch)` represents the fetch stage of instruction `i`
- Sometimes the core of a node needs to be explicitly specified
- Eg: `(i, (0, MemoryHierarchy))` represents `i` reaching the memory hierarchy, which is nominally on core 0
  - Reflects that there's only one memory hierarchy, not one per core

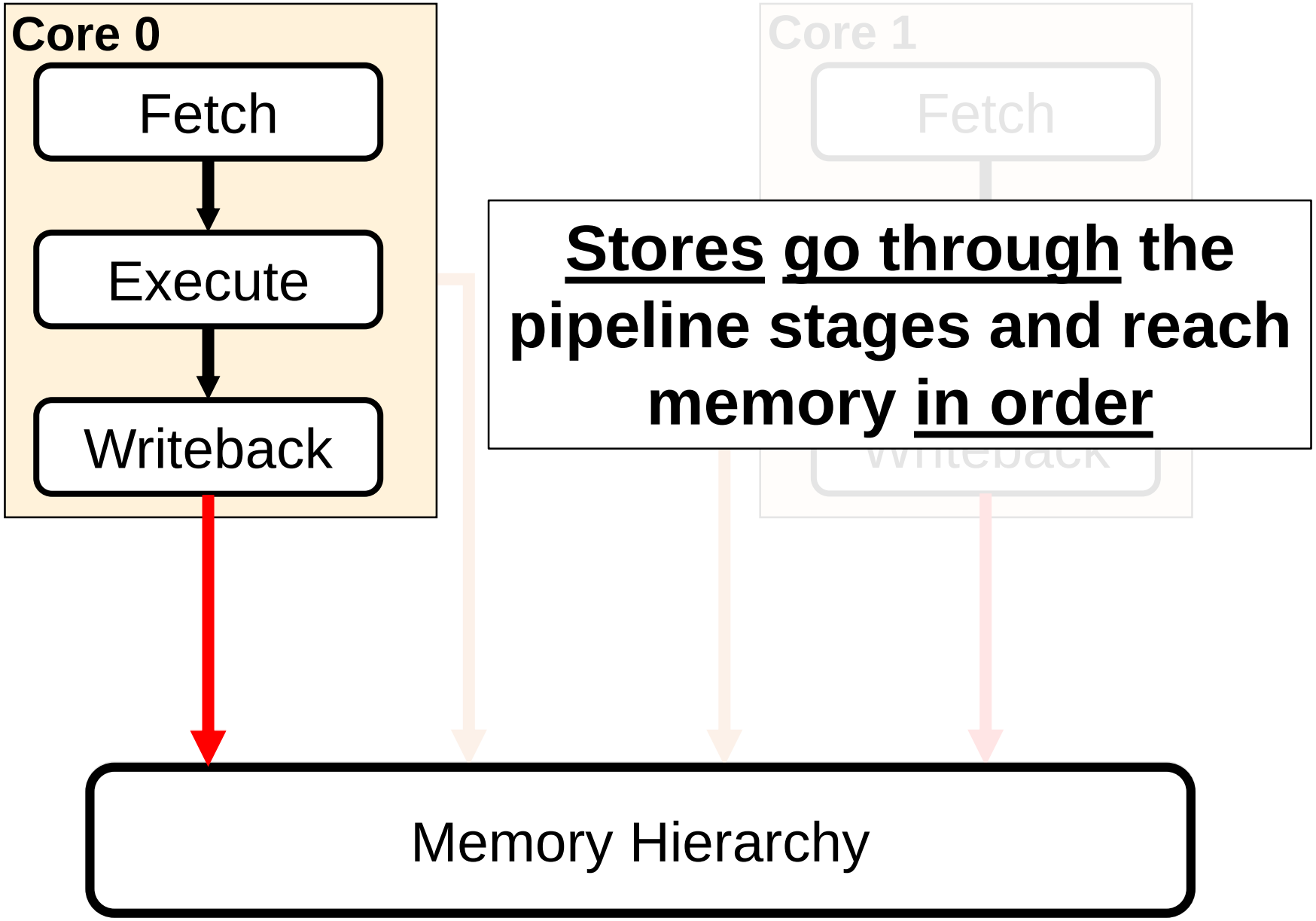
Writing μSpec Axioms

- A microarchitecture spec has three components:
  - Stage identifier definitions
  - Macro definitions (optional) for axiom decomposition and reuse
  - Axiom definitions
- Axioms are comprised of FOL operators and built-in ***predicates***
- Some predicates deal with nodes and edges
  - `EdgeExists ((i1, Fetch), (i2, Fetch))`
  - `NodeExists ((i1, Execute))`
- Other predicates represent architecture-level properties
  - `SameCore <instr1> <instr2>`
  - `SamePhysicalAddress, SameData, IsAnyRead, ProgramOrder,…`

Finding Axioms



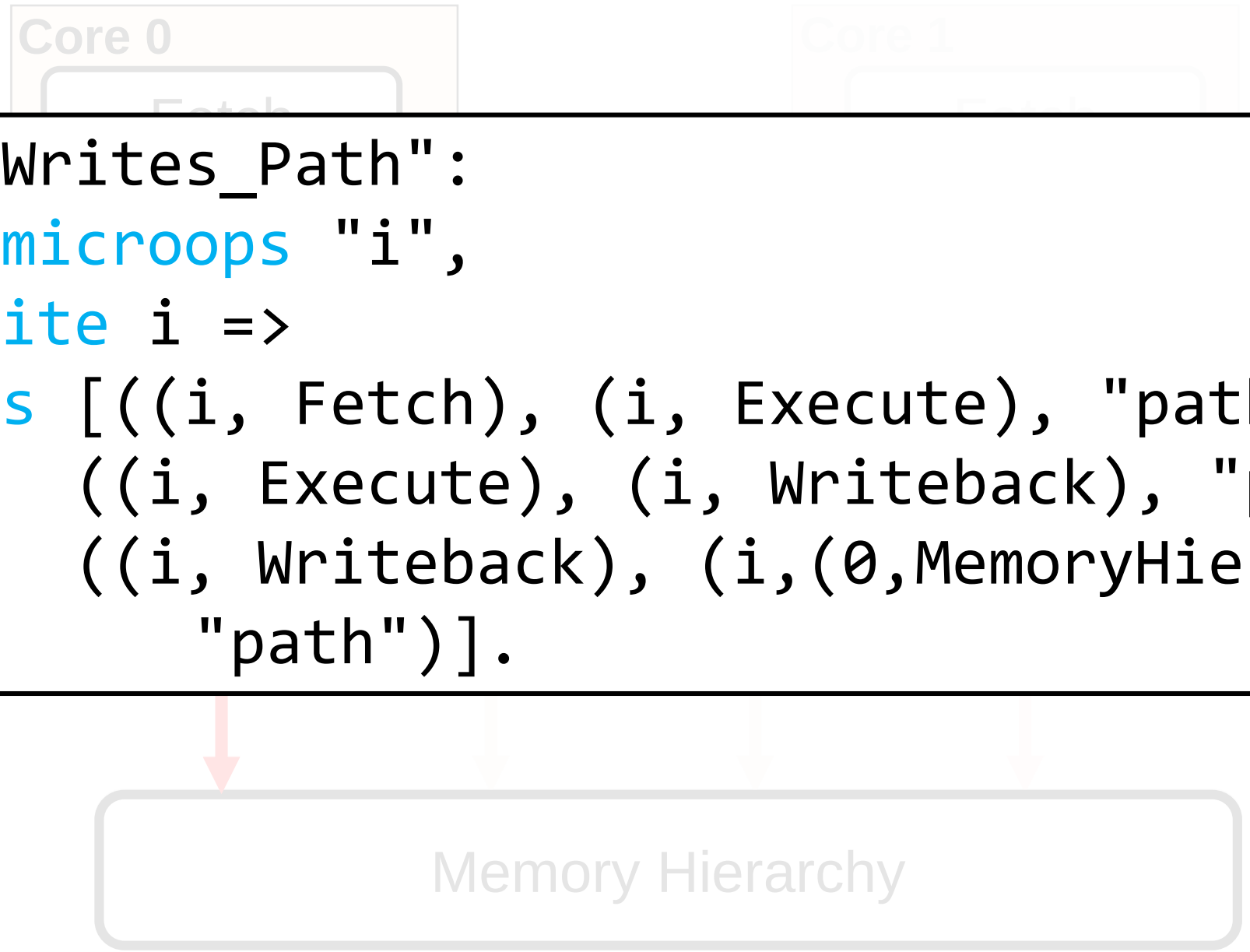
Finding Axioms



The Writes_Path Axiom

SC_fillable.uarch, line 18

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path");
          ((i, Writeback), (i,(0,MemoryHierarchy)),
             "path")].
```

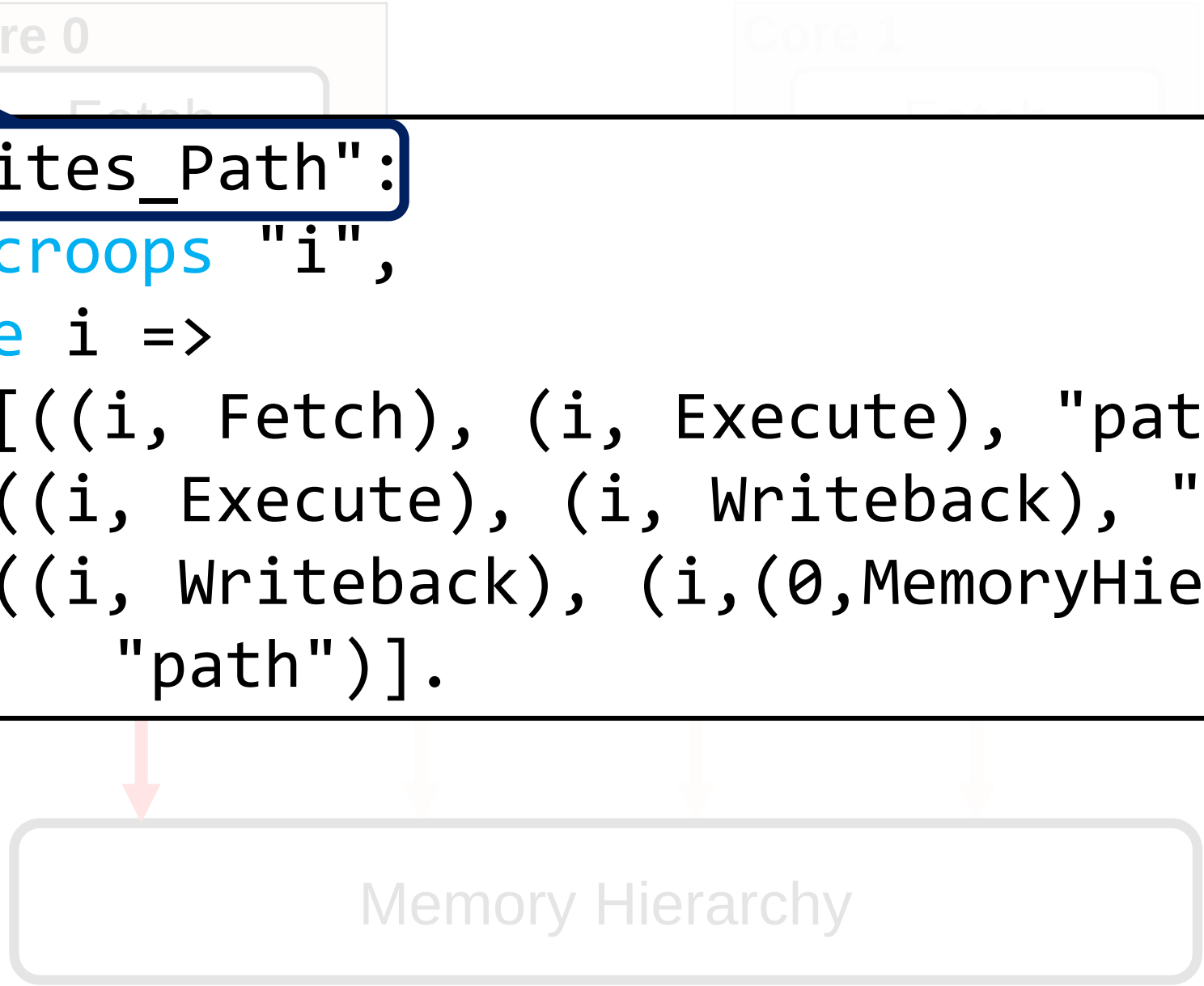


The Writes_Path Axiom

SC_fillable.uarch, line 18

Axiom name

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path");
          ((i, Writeback), (i,(0,MemoryHierarchy)),
             "path")].
```

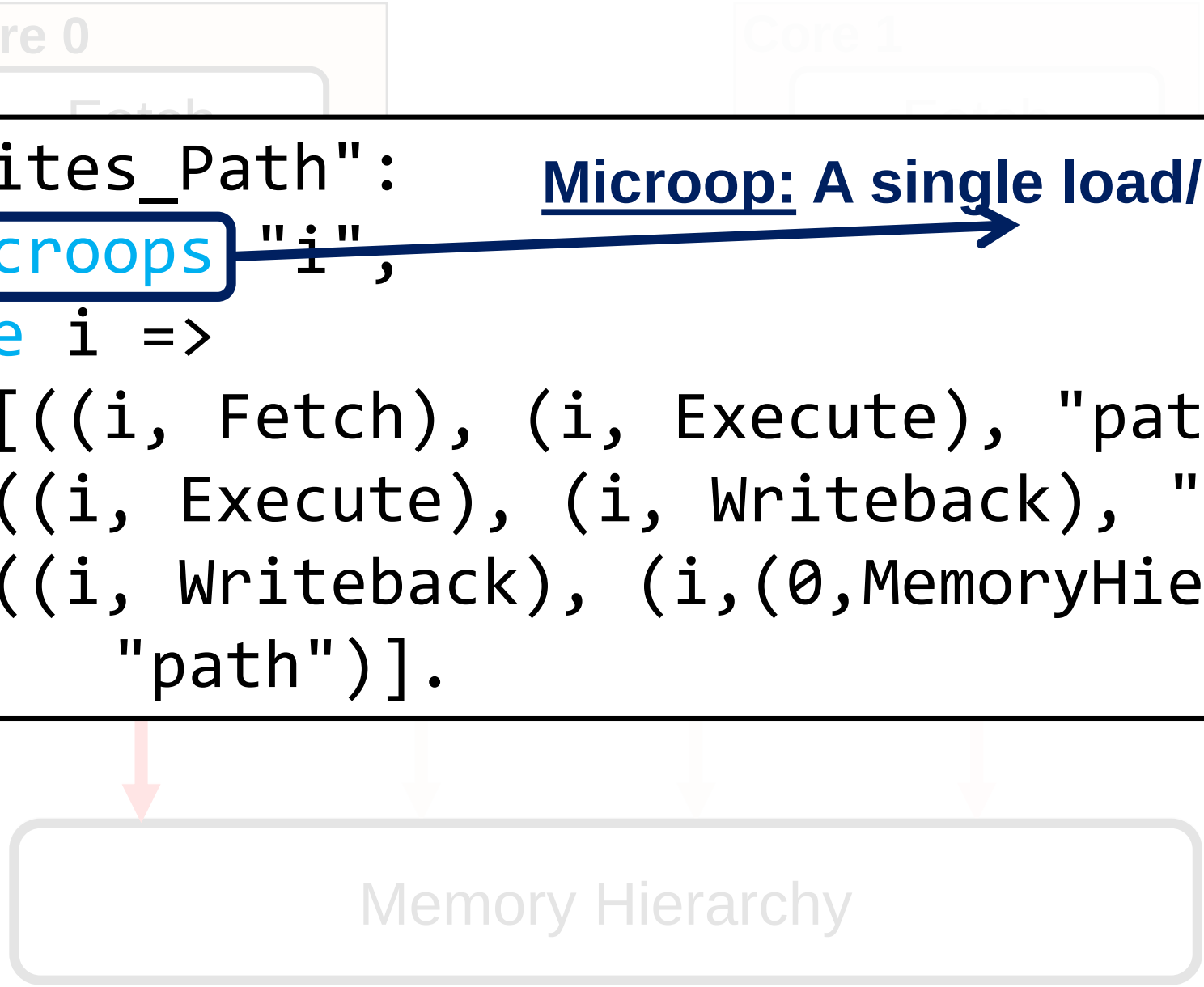


The Writes_Path Axiom

SC_fillable.uarch, line 18

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path");
          ((i, Writeback), (i,(0,MemoryHierarchy)),
             "path")].
```

Microop: A single load/store op.



The Writes_Path Axiom

SC_fillable.uarch, line 18

For all writes (IsAnyWrite predicate)

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path");
          ((i, Writeback), (i,(0,MemoryHierarchy)),
             "path")].
```

The Writes_Path Axiom

SC_fillable.uarch, line 18

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path");
          ((i, Writeback), (i,(0,MemoryHierarchy)),
              "path")].
```

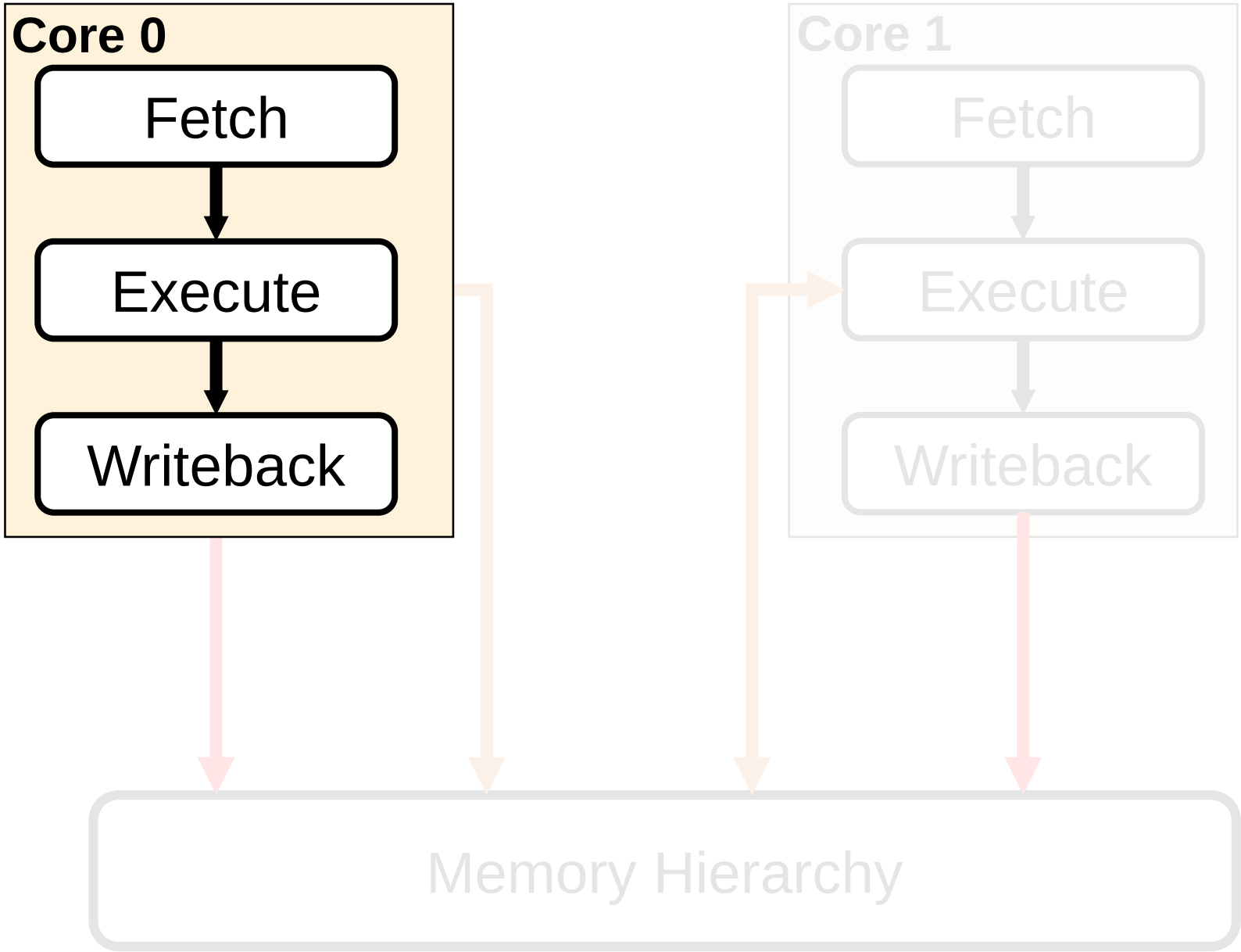
Add edges from:

-Fetch to Execute

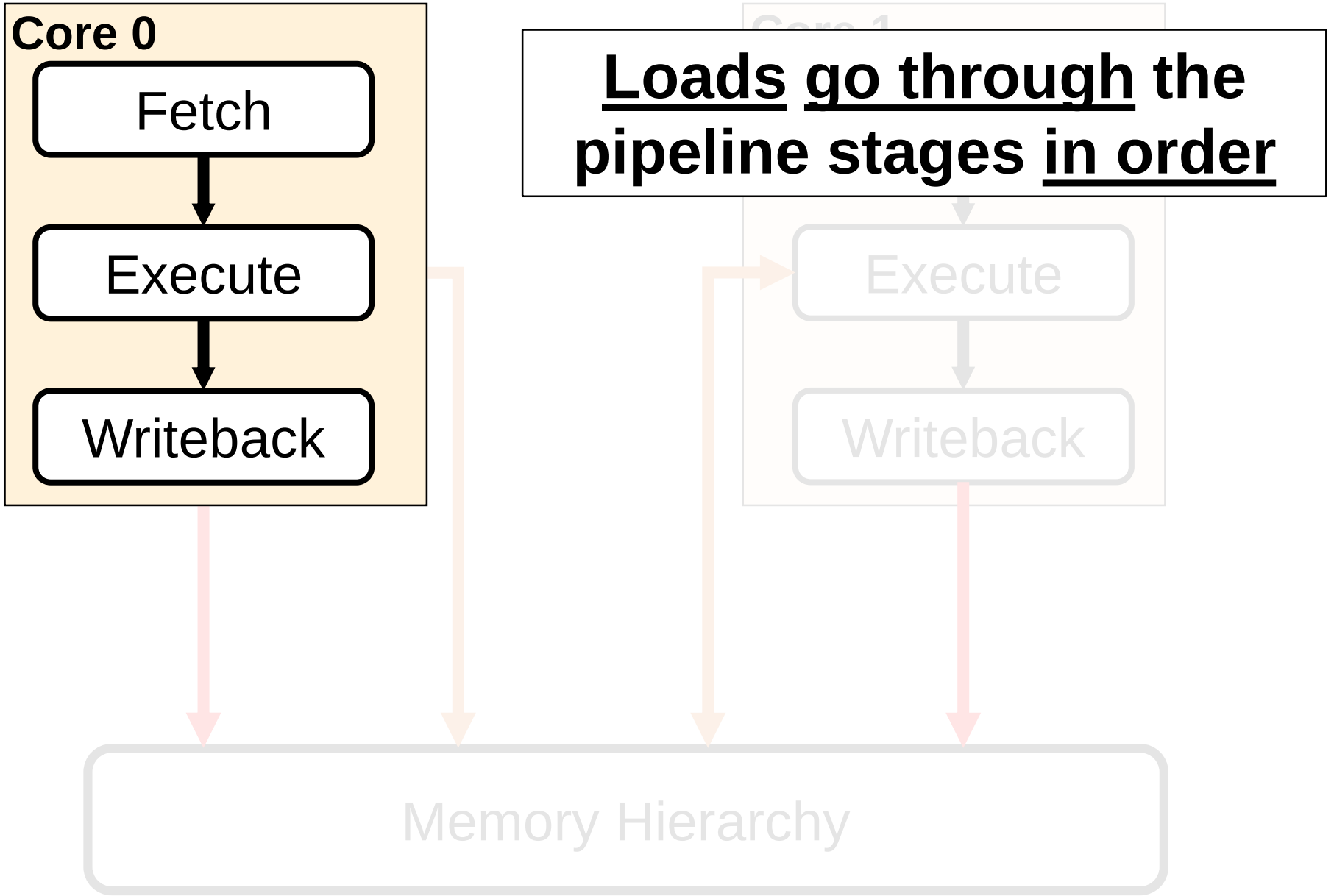
-Execute to Writeback

-Writeback to MemoryHierarchy

Finding Axioms

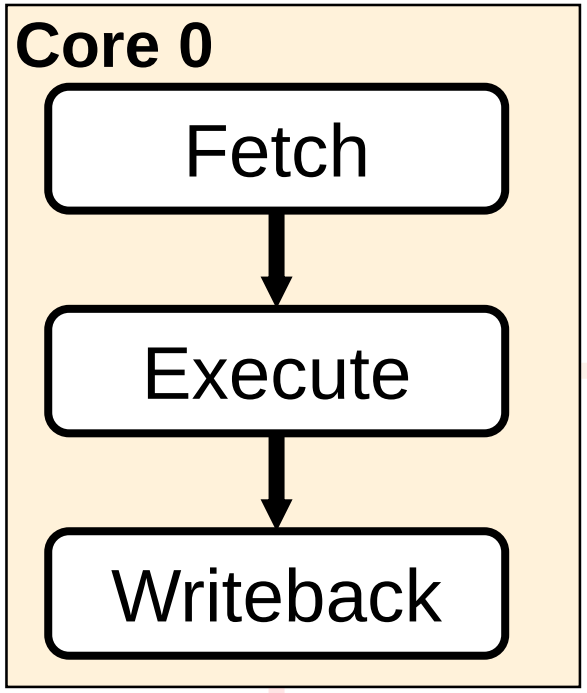


Finding Axioms



Loads go through the pipeline stages in order

Finding Axioms



<u>Loads</u> <u>go through</u> the pipeline stages <u>in order</u>

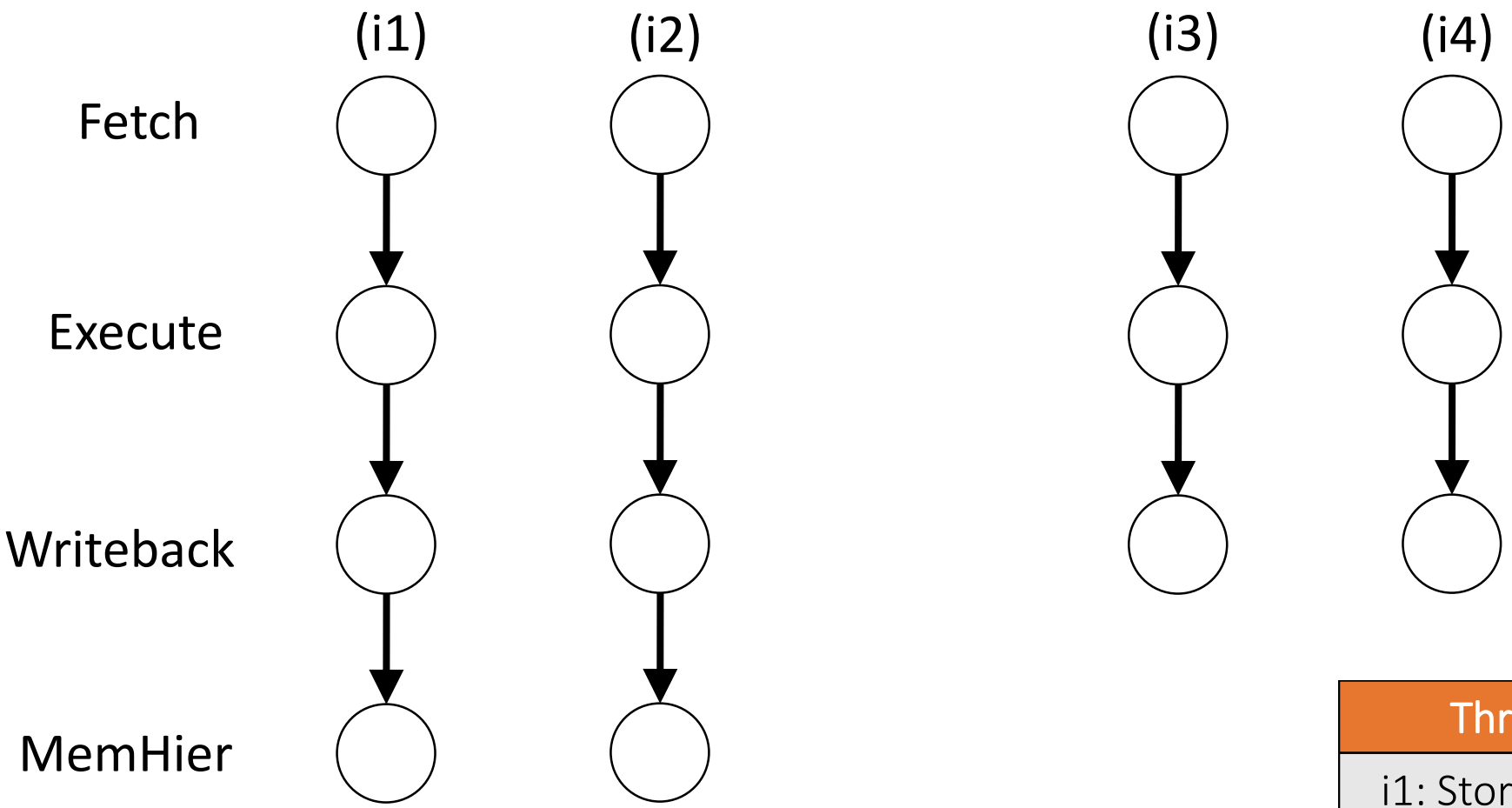
<u>Reads_Path</u> Axiom is very similar to **<u>Writes_Path</u>** axiom (no WB→MemHier edge)

`SC_fillable.uarch, line 26`

Memory Hierarchy

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

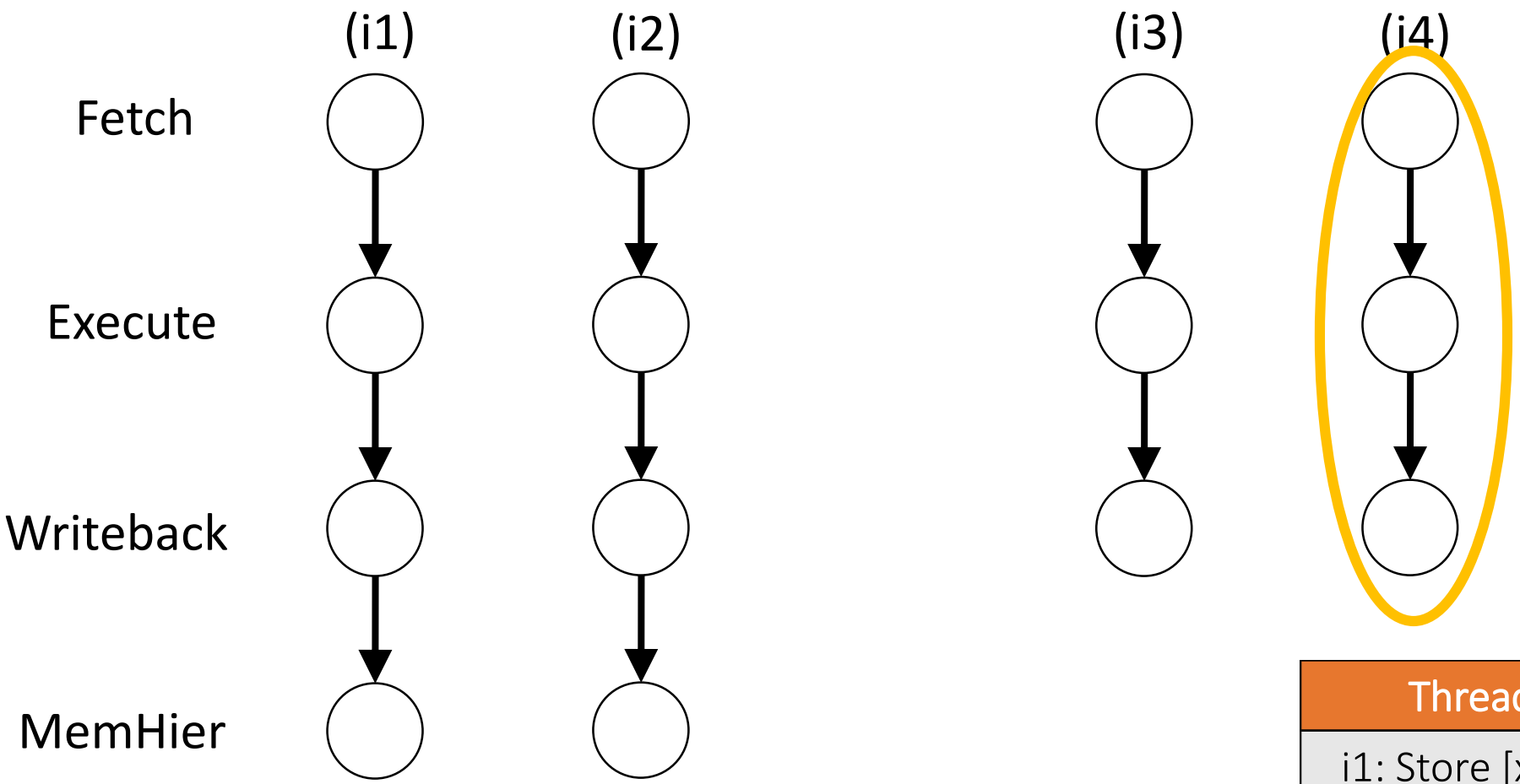


Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline



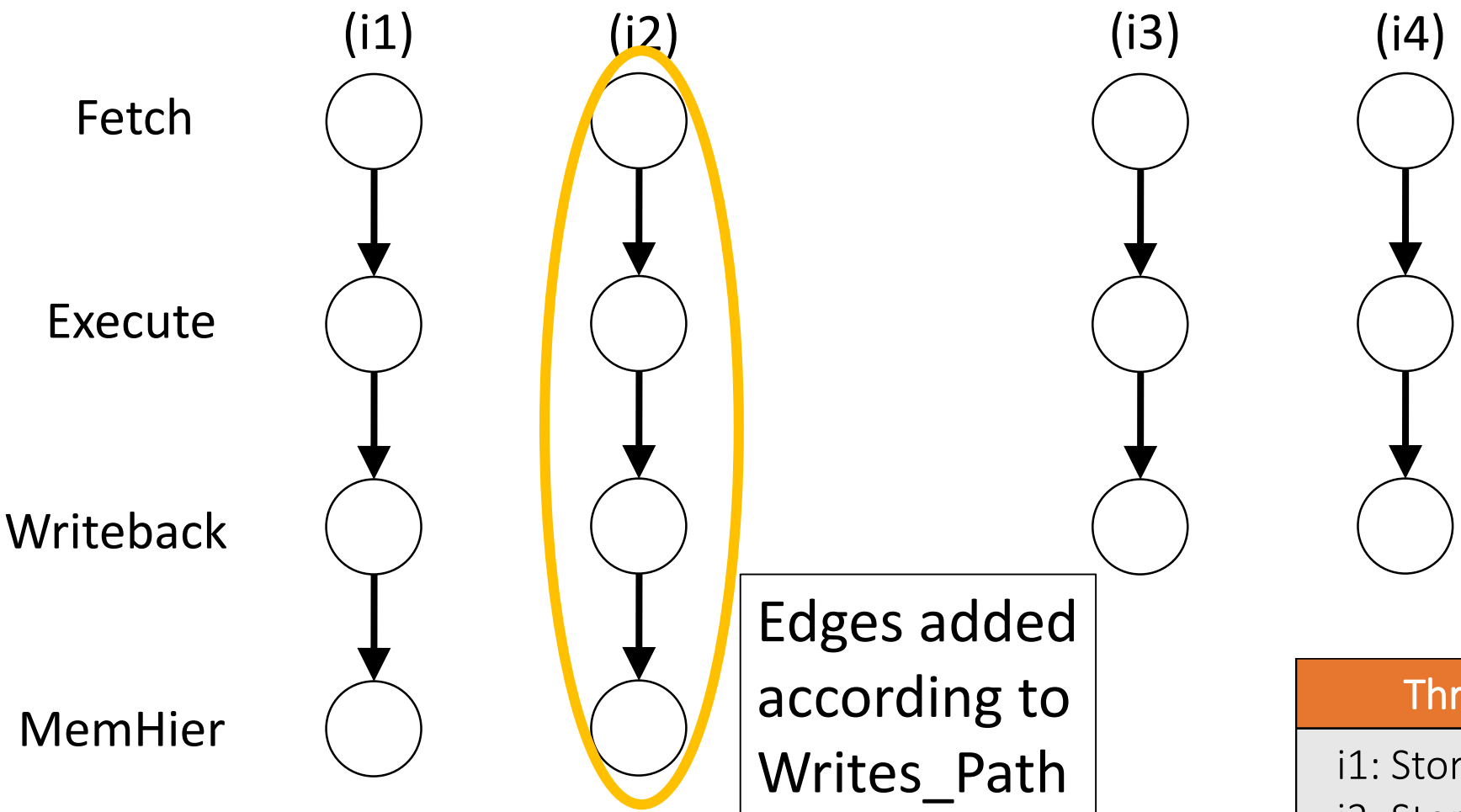
Edges added according to Reads_Path axiom

Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

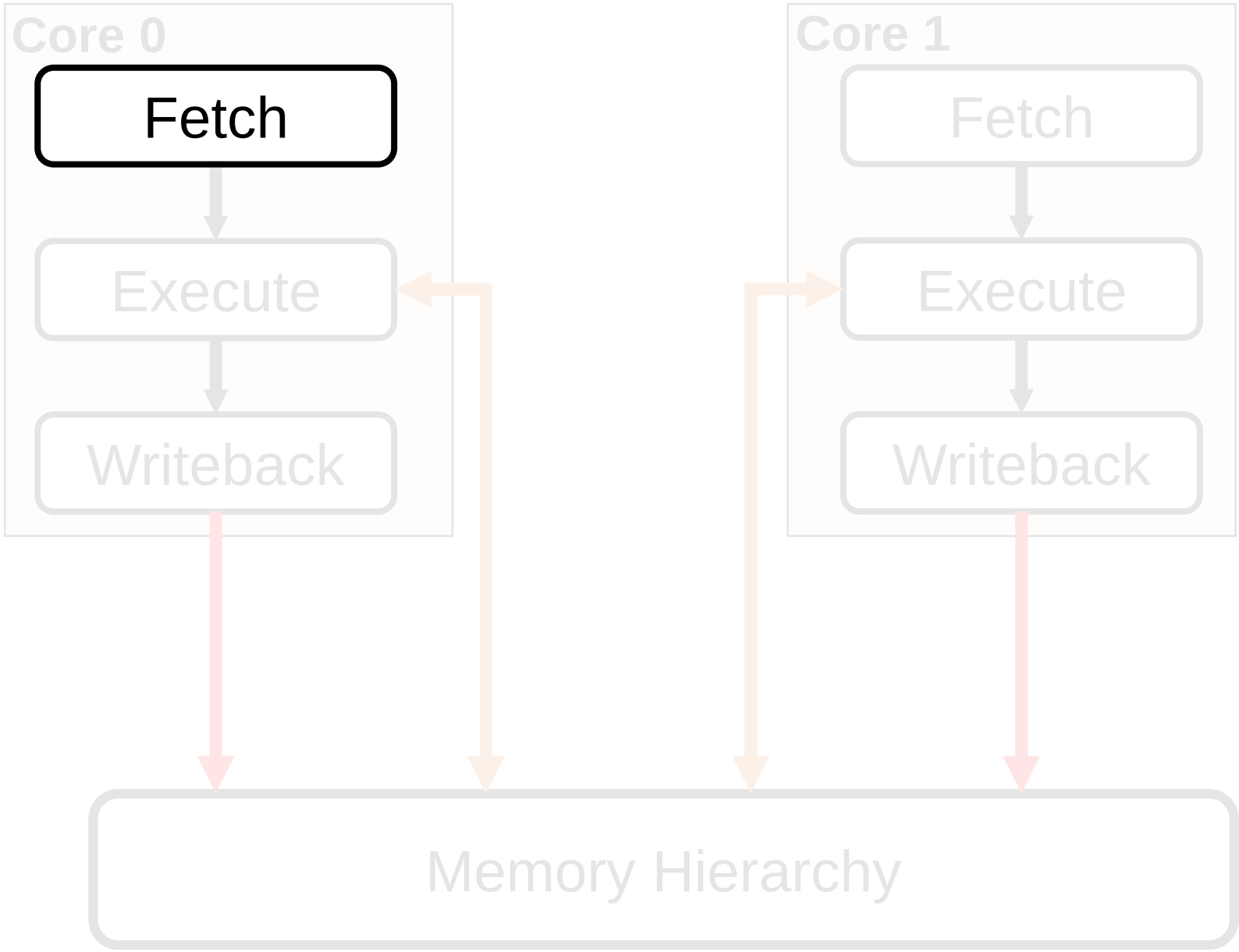


Edges added according to Writes_Path axiom

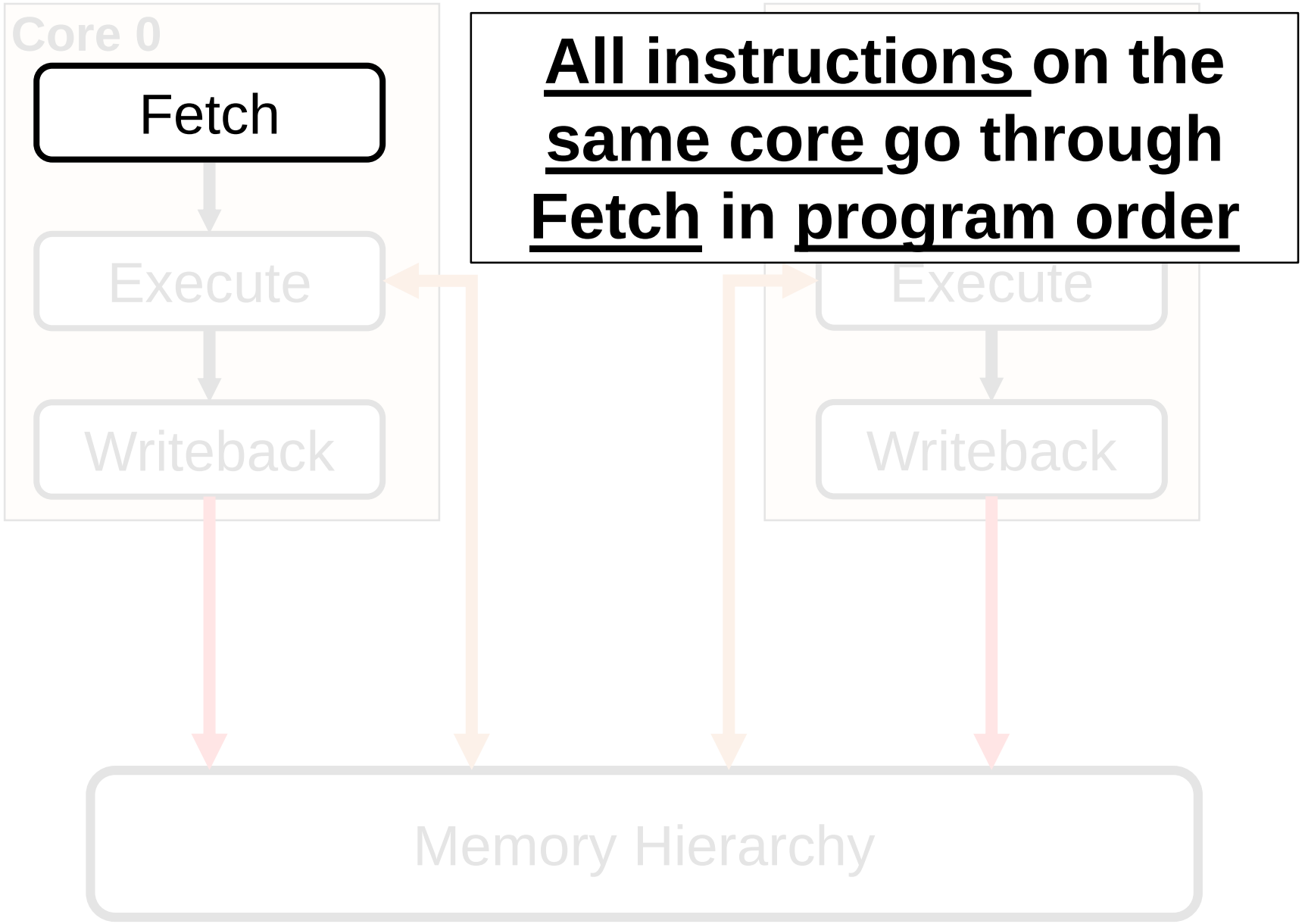
Initially, Mem[x] = 0

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

Finding Axioms



Finding Axioms



All instructions on the same core go through Fetch in program order

The PO_Fetch Axiom

SC_fillable.uarch, line 32

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

The PO_Fetch Axiom

SC_fillable.uarch, line 32

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

Predicates check that instrs are on the same core and in program order

The PO_Fetch Axiom

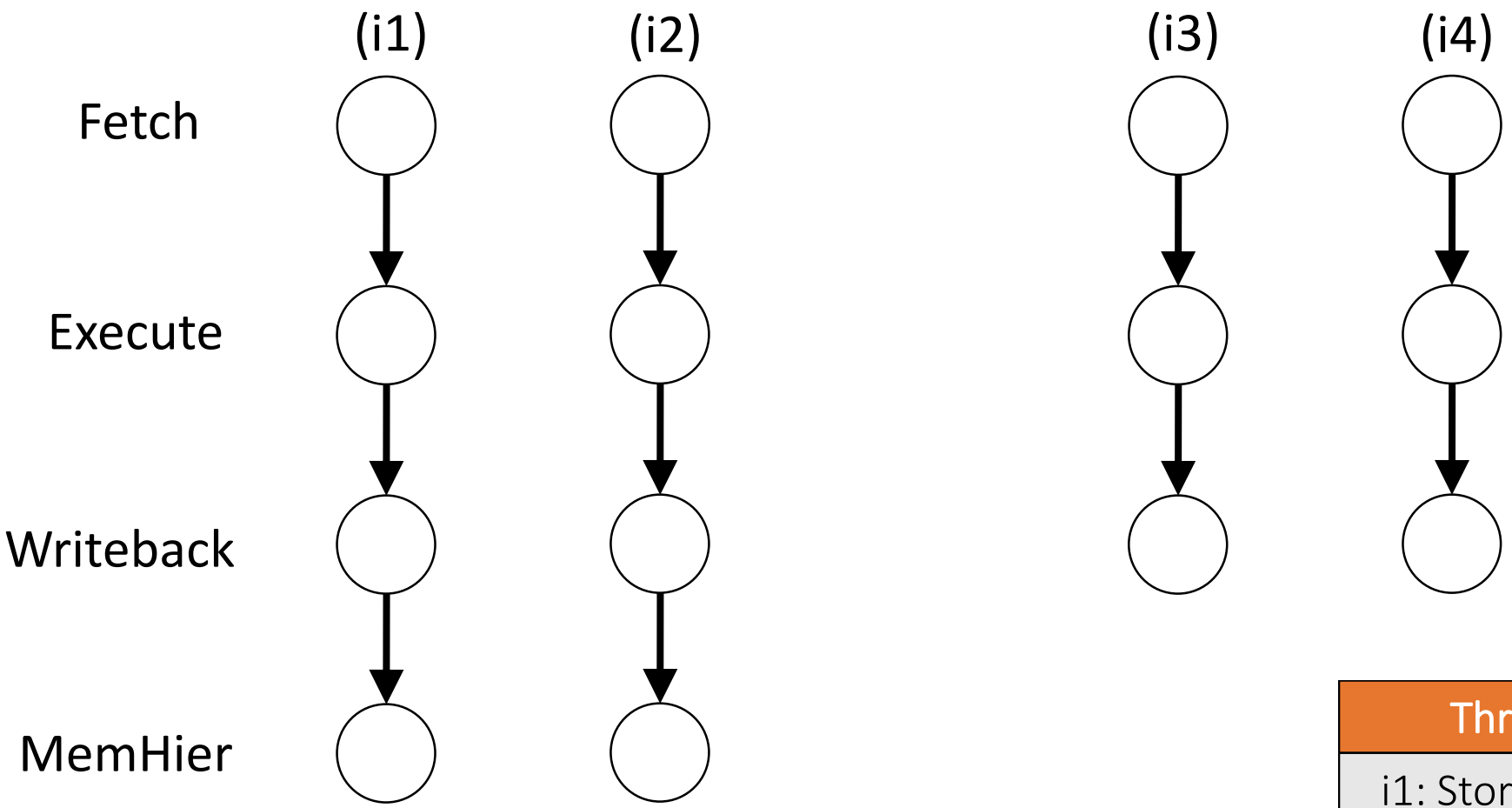
SC_fillable.uarch, line 32

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

Add edge from Fetch stage of earlier instruction i1 to Fetch stage of later instruction i2

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

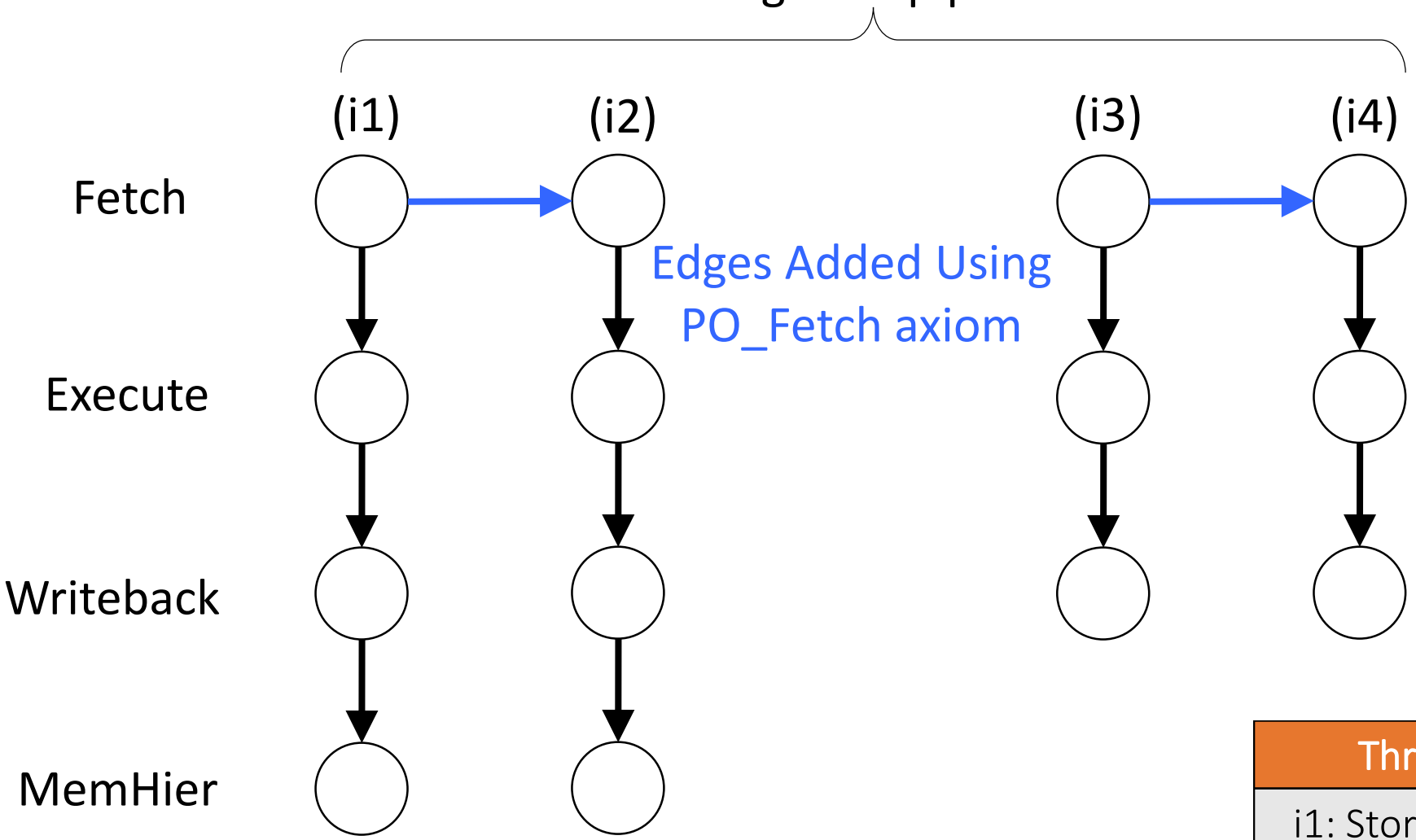


Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

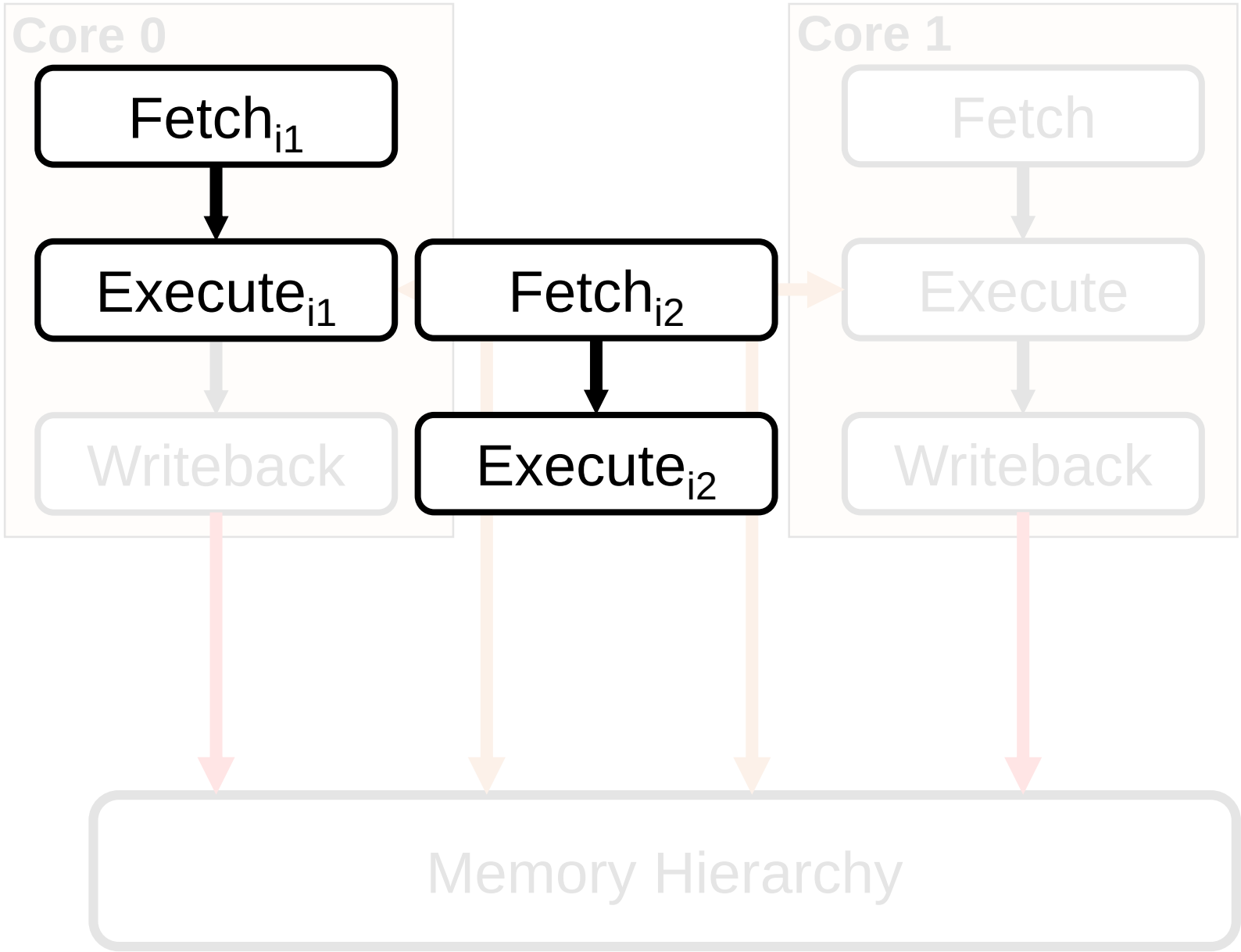
Each column represents an instruction flowing through the pipeline



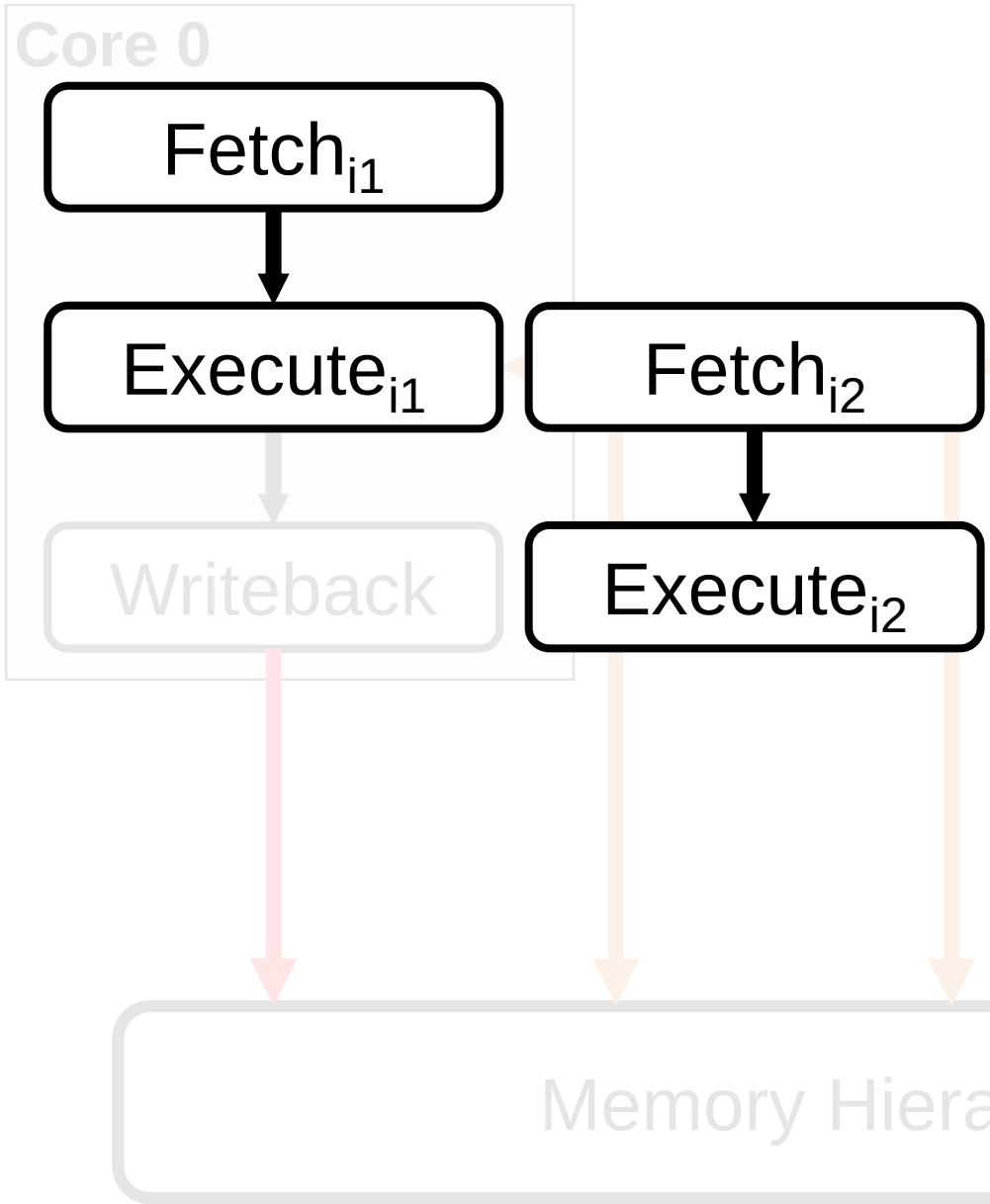
Initially, Mem[x] = 0

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

Finding Axioms



Finding Axioms



If <u>two instructions</u> on the <u>same core</u> go through <u>Fetch in order</u>, they will go through <u>Execute</u> in the <u>*same* order

The Execute_Stage_Is_In_order Axiom

SC_fillable.uarch, line 38

```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").
```

The Execute_Stage_Is_In_order Axiom

SC_fillable.uarch, line 38

```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").
```

If instructions i1 and i2 on same core go through Fetch in order…

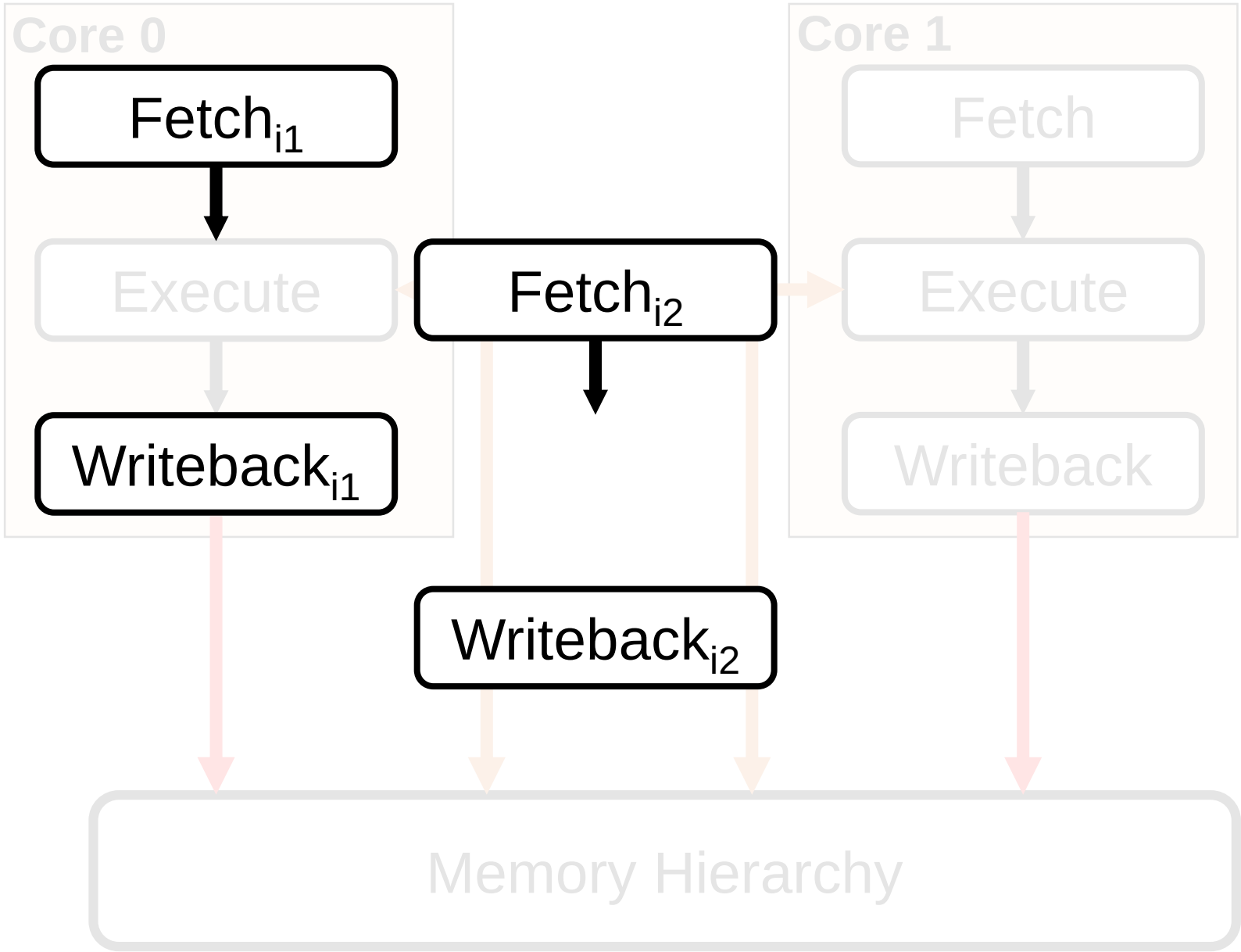
The Execute_Stage_Is_In_order Axiom

SC_fillable.uarch, line 38

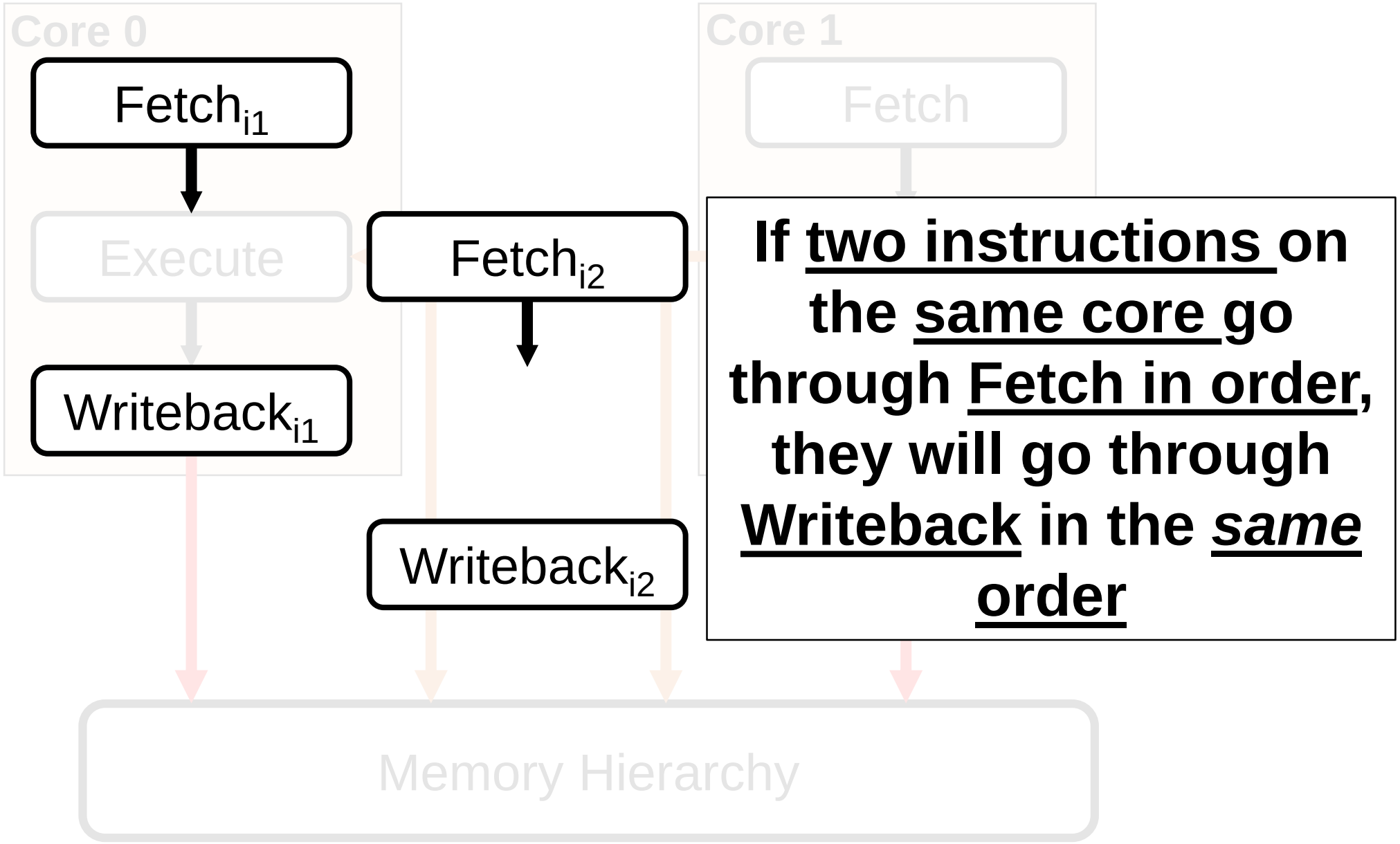
```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").
```

…then they go through Execute in the same order.

Finding Axioms



Finding Axioms



The Writeback_Stage_Is_In_Order Axiom

SC_fillable.uarch, line 55

If two instructions on the same core go through Fetch in order, they will go through Writeback in the *same* order

```
Axiom "Writeback_stage_is_in_order":
forall microops "i1",
forall microops "i2",
_________ i1 i2 /\
EdgeExists ((i1, _____), (i2, _____), "") =>
AddEdge ((i1, __________), (i2, __________), "PPO").
```

The Writeback_Stage_Is_In_Order Axiom

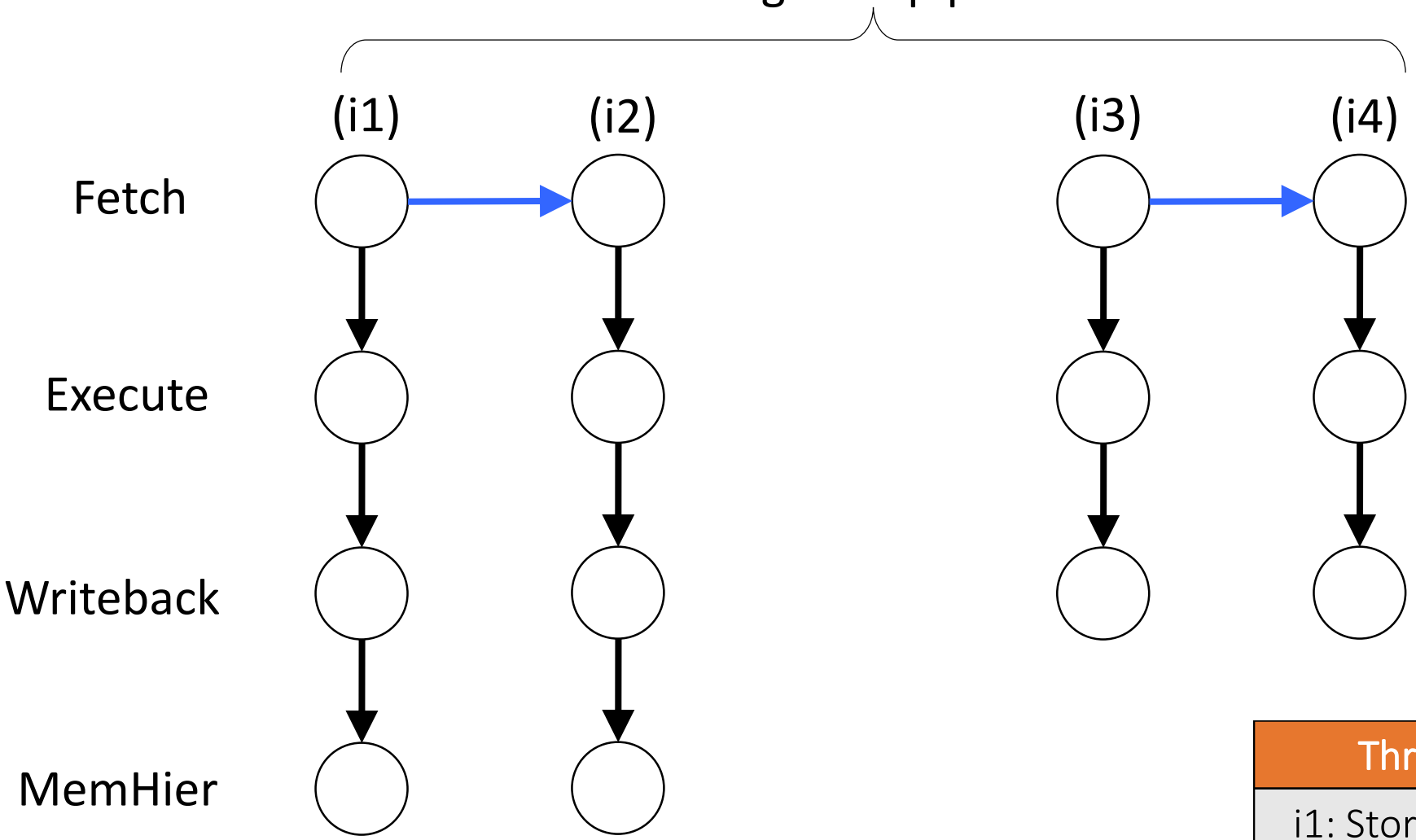
SC_fillable.uarch, line 55

If two instructions on the same core go through Fetch in order, they will go through Writeback in the *same* order

```
Axiom "Writeback_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Writeback), (i2, Writeback), "PPO").
```

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

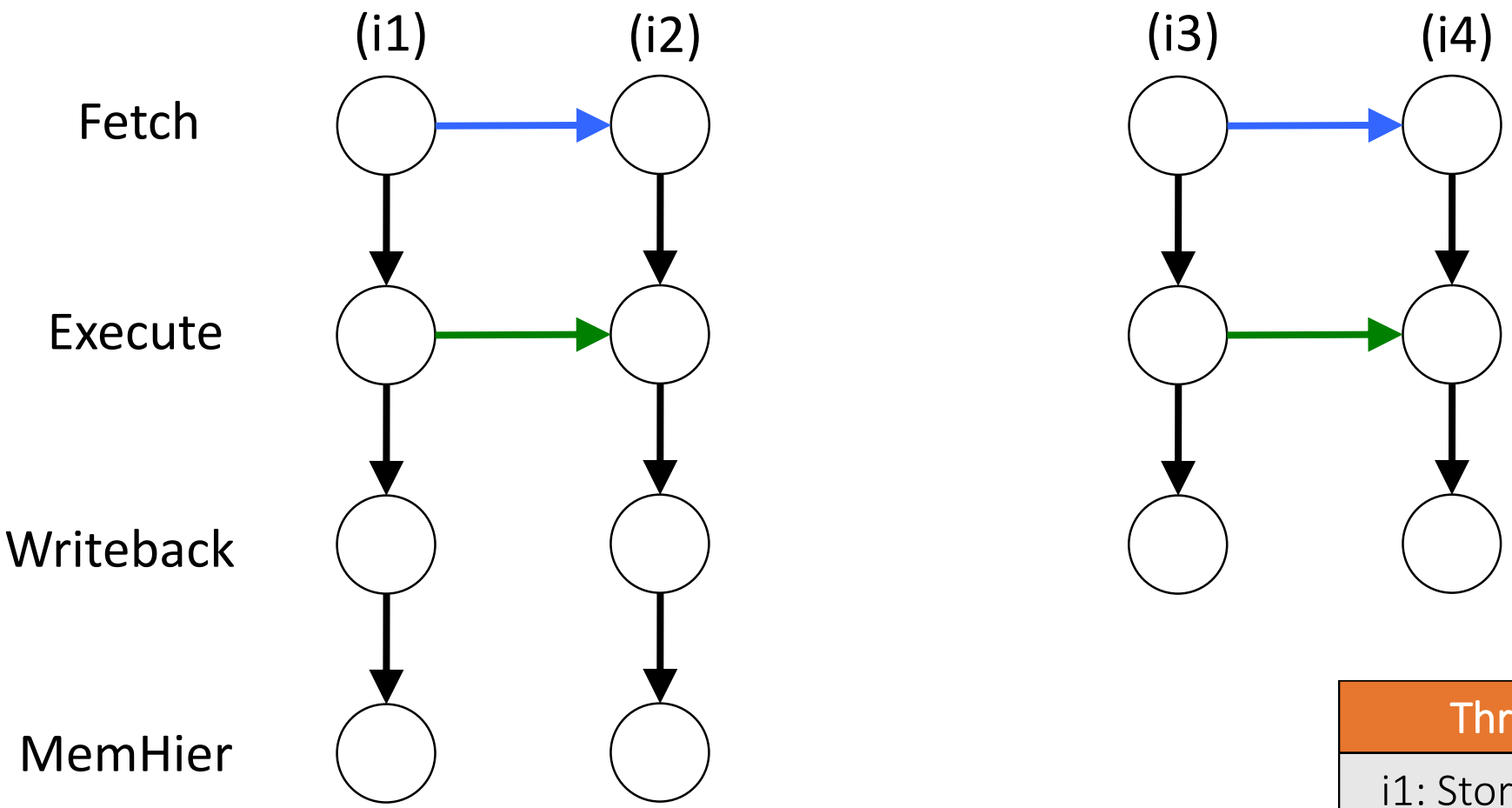


Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

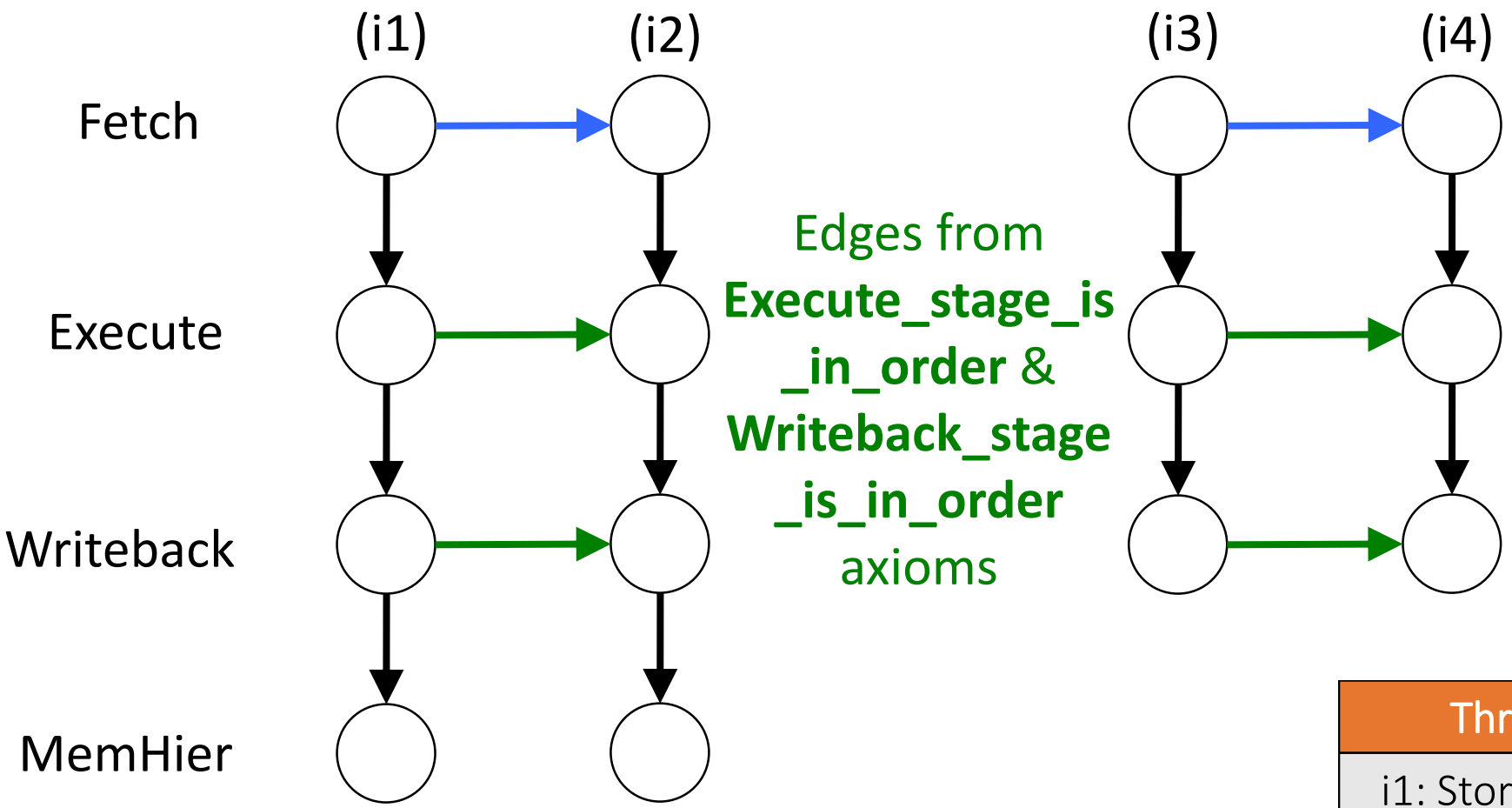


Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

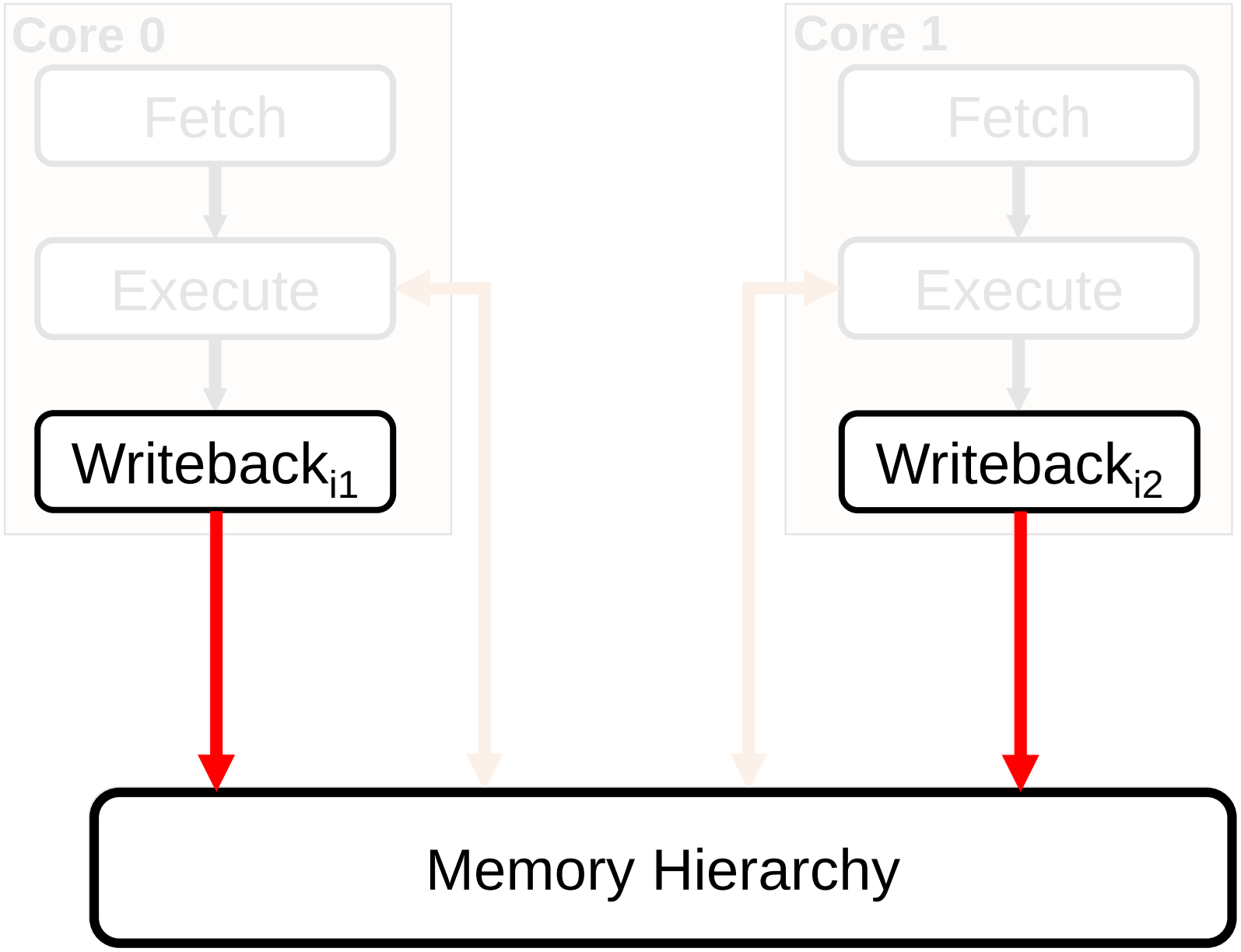


Edges from **Execute_stage_is_in_order** & **Writeback_stage_is_in_order** axioms

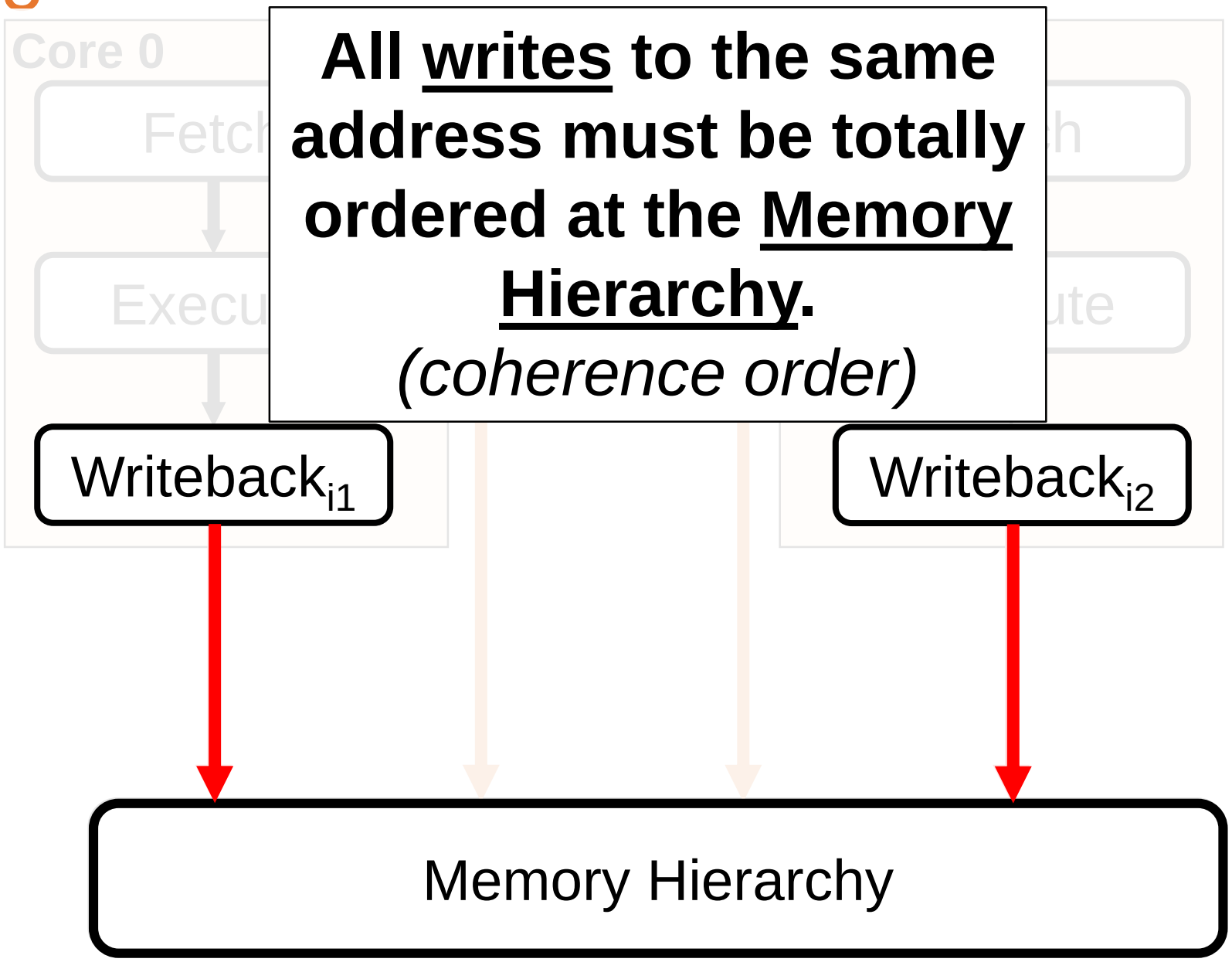
Initially, Mem[x] = 0

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

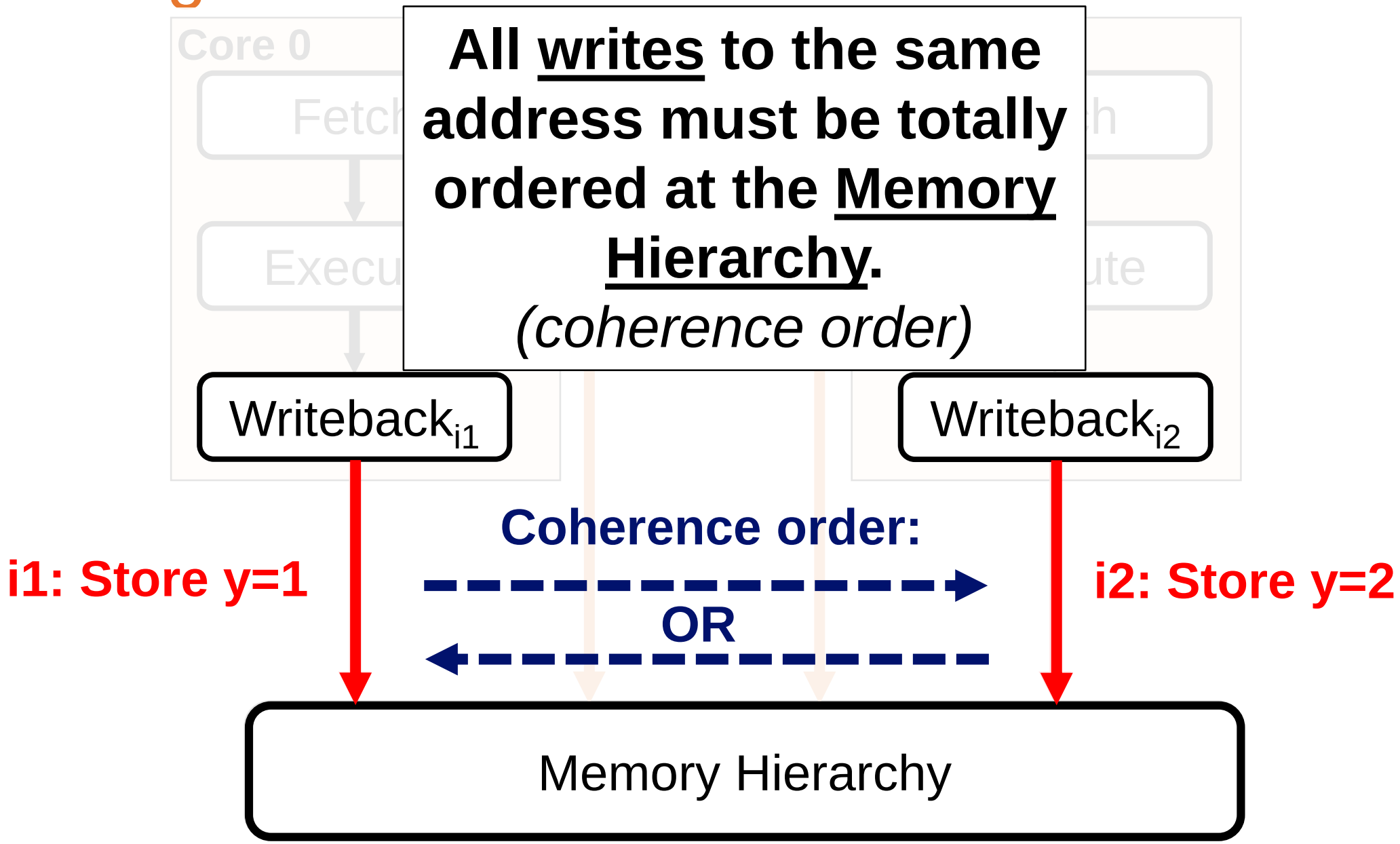
Finding Axioms



Finding Axioms



Finding Axioms



The WriteSerialization Axiom

SC_fillable.uarch, line 65

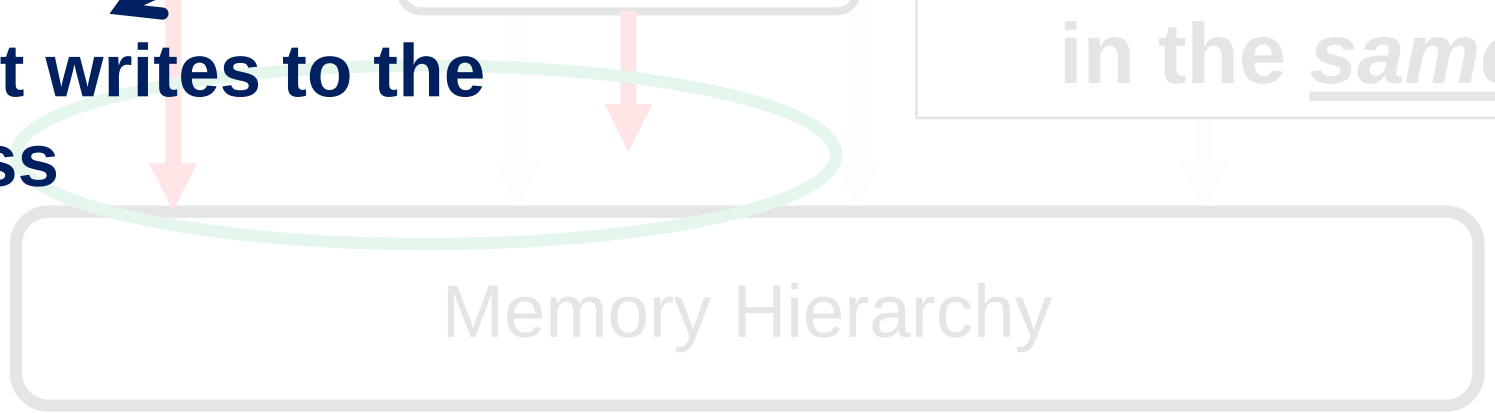
```
Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
  ( ~(SameMicroop i1 i2) /\ IsAnyWrite i1
   /\ IsAnyWrite i2 /\ SamePhysicalAddress i1 i2) =>
    (EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \/
     EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).
```

The WriteSerialization Axiom

SC_fillable.uarch, line 65

```
Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
  ( ~(SameMicroop i1 i2) /\ IsAnyWrite i1
   /\ IsAnyWrite i2 /\ SamePhysicalAddress i1 i2) =>
     (EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \/
      EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).
```

Two different writes to the same address



The WriteSerialization Axiom

SC_fillable.uarch, line 65

```
Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
  ( ~(SameMicroop i1 i2) /\ IsAnyWrite i1
   /\ IsAnyWrite i2 /\ SamePhysicalAddress i1 i2) =>
    (EdgeExists ((i1, (0,MemHier)), (i2, (0,MemHier))) \/
     EdgeExists ((i2, (0,MemHier)), (i1, (0,MemHier)))).
```

Either i1 is before i2 in coherence order, OR vice-versa.

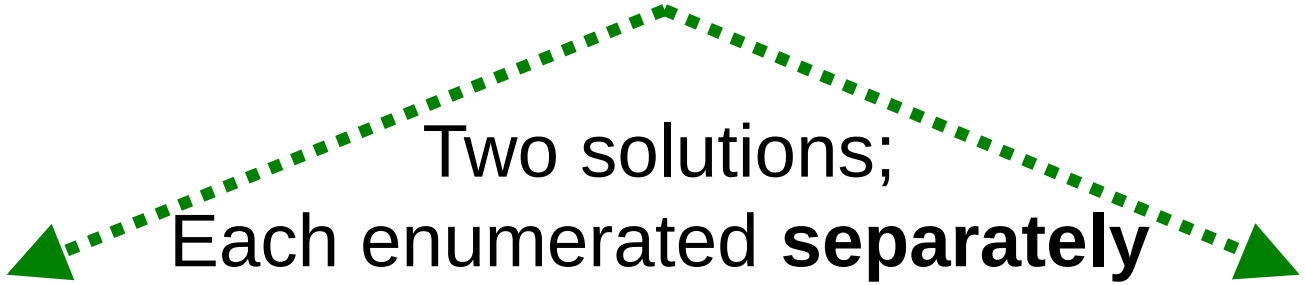
μhb Graphs for `co-mp` Using Axioms

WriteSerialization axiom

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

μhb Graphs for `co-mp` Using Axioms

WriteSerialization axiom

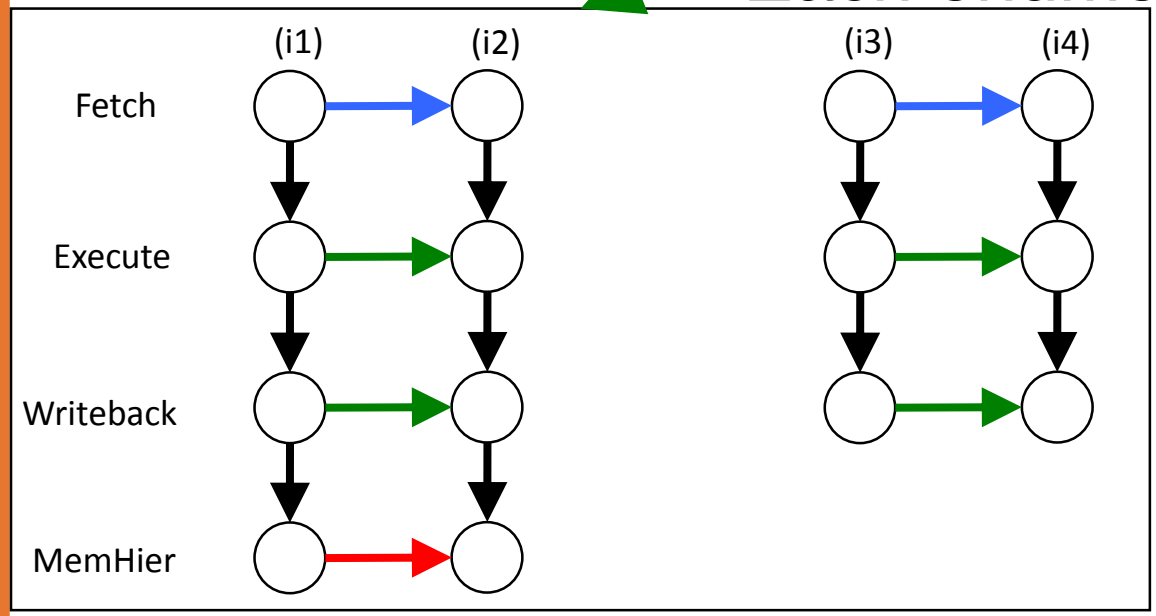


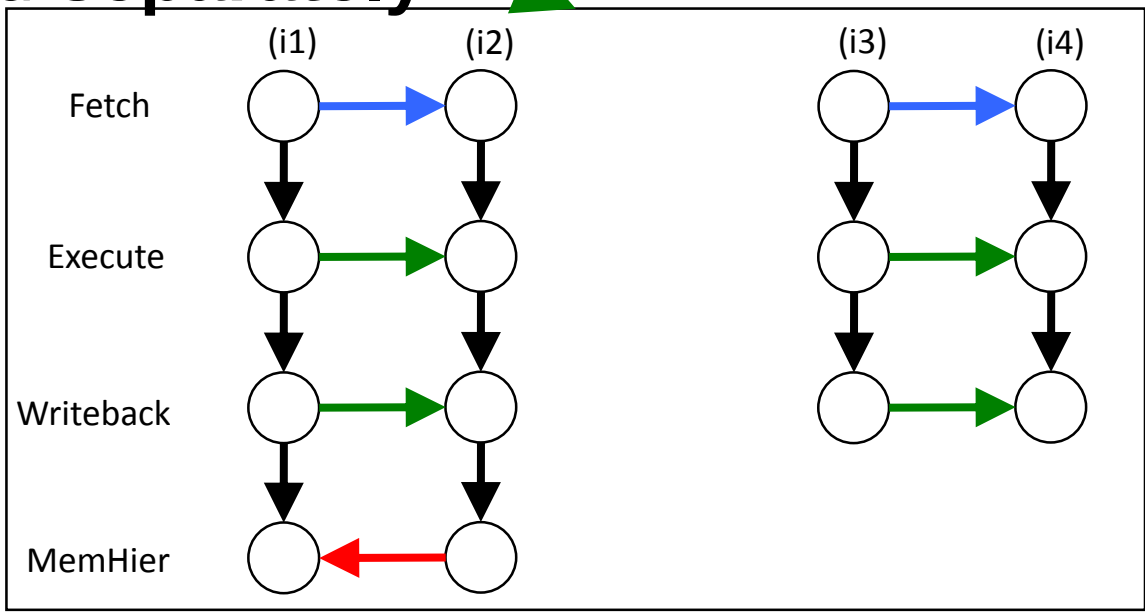
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated **separately**



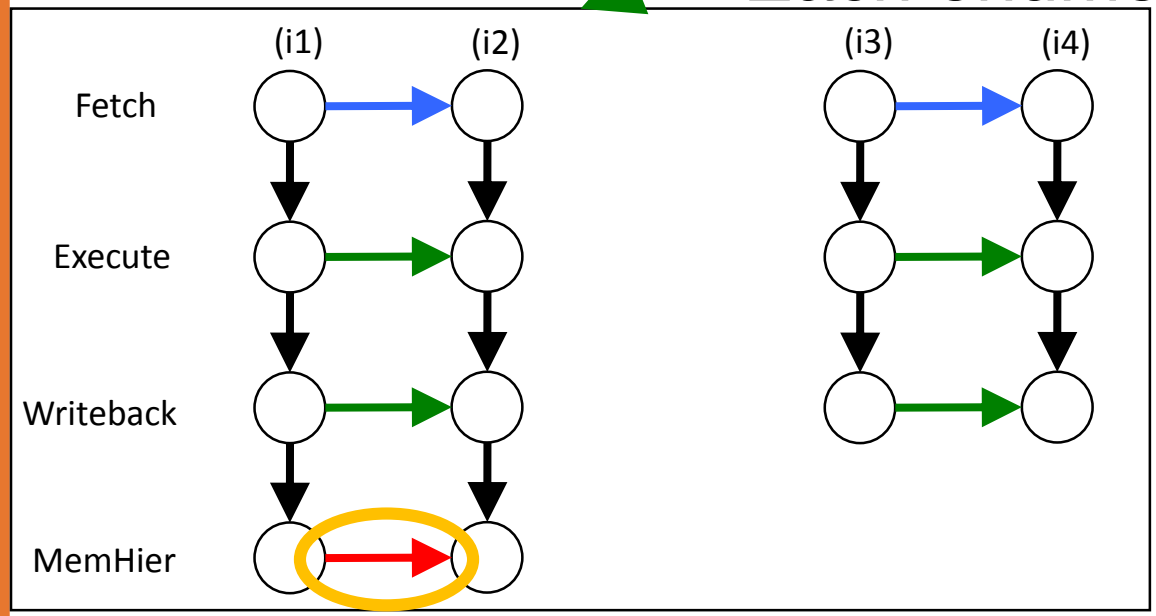


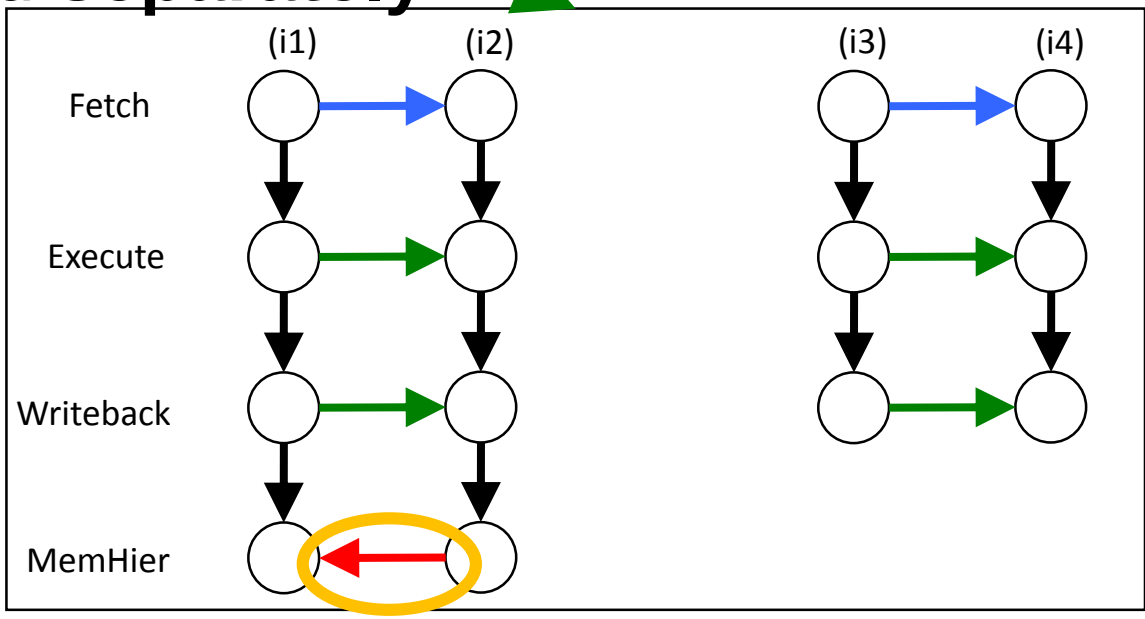
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated **separately**



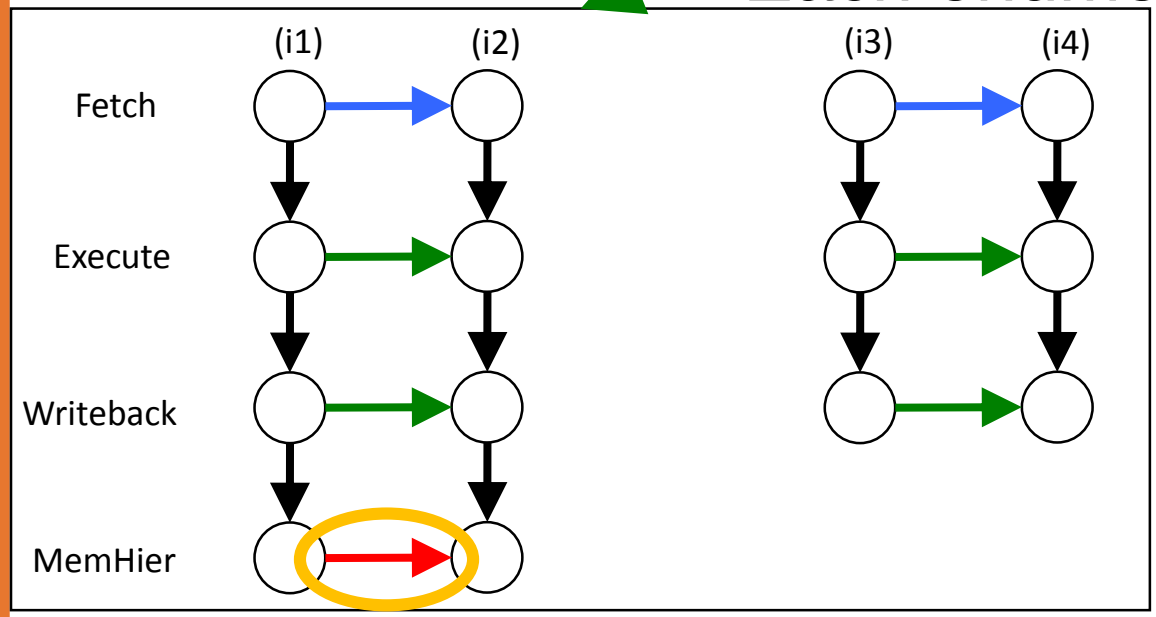


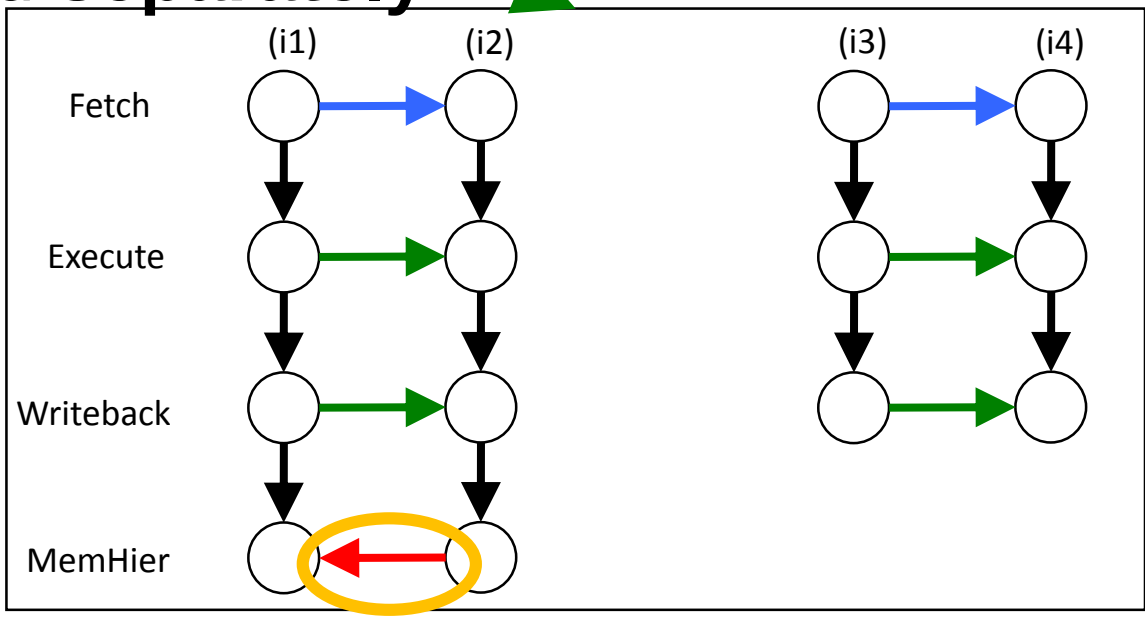
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

μhb Graphs for co-mp Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated **separately**

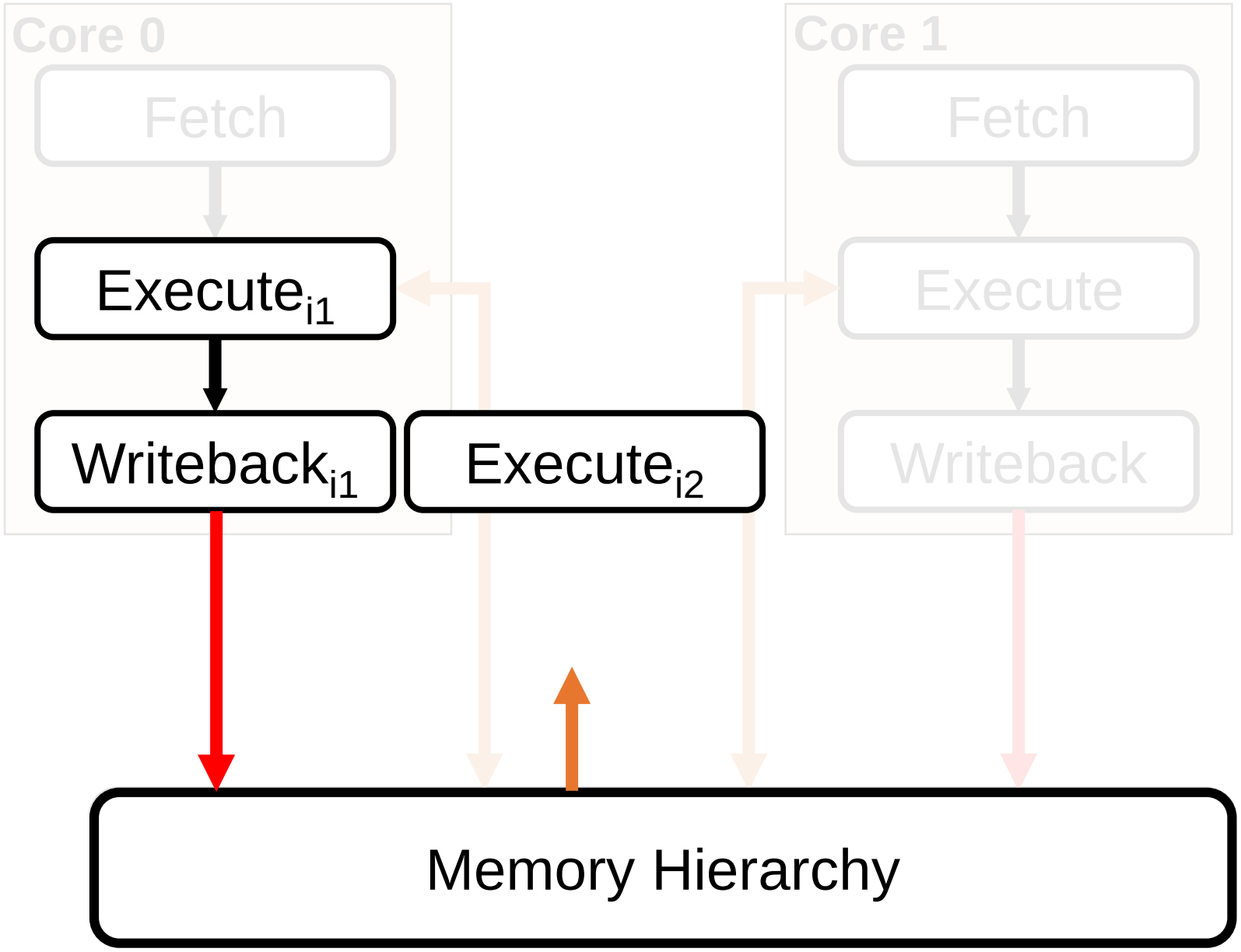




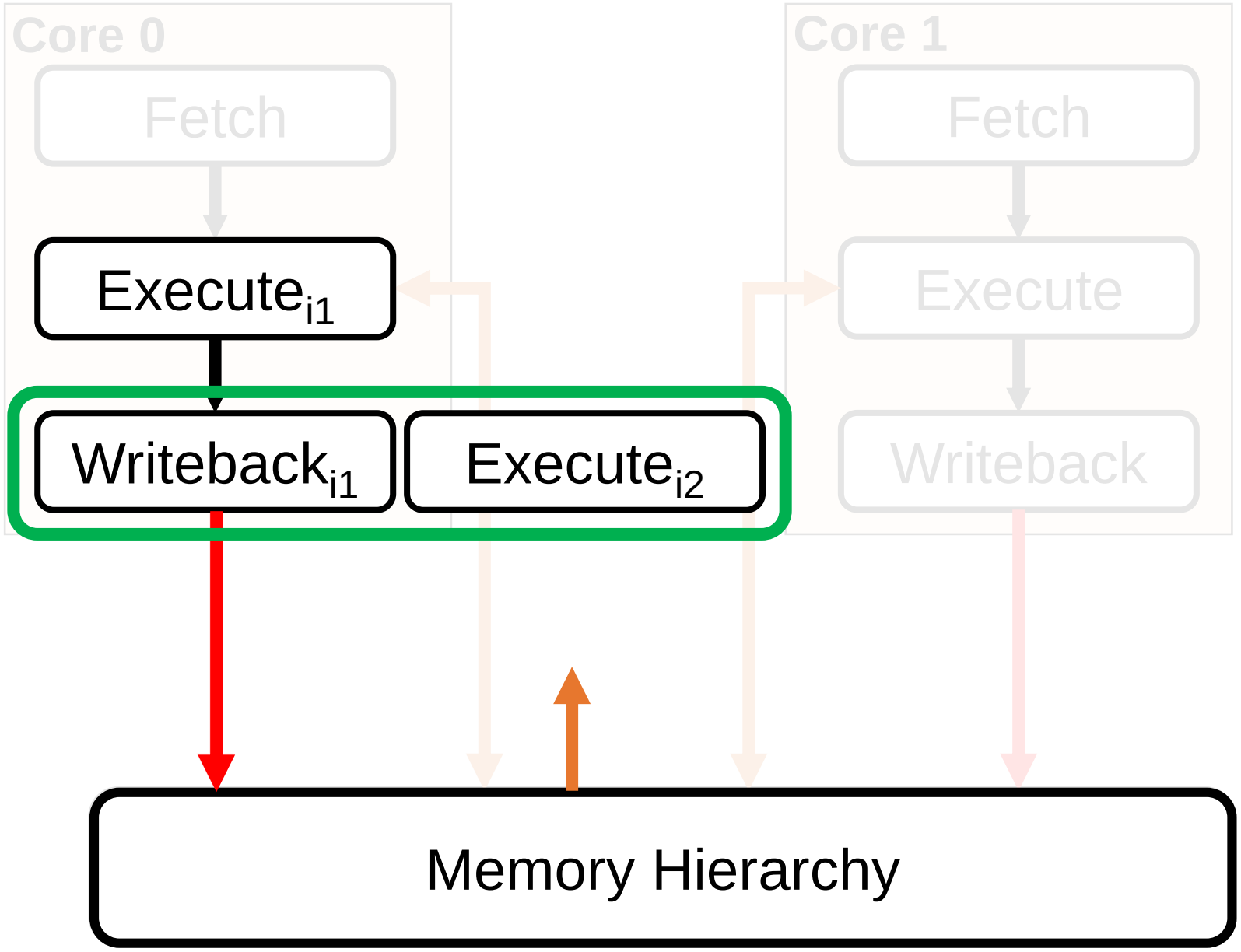
- PipeCheck examines ***all*** cases
- Will focus on left graph for clarity

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

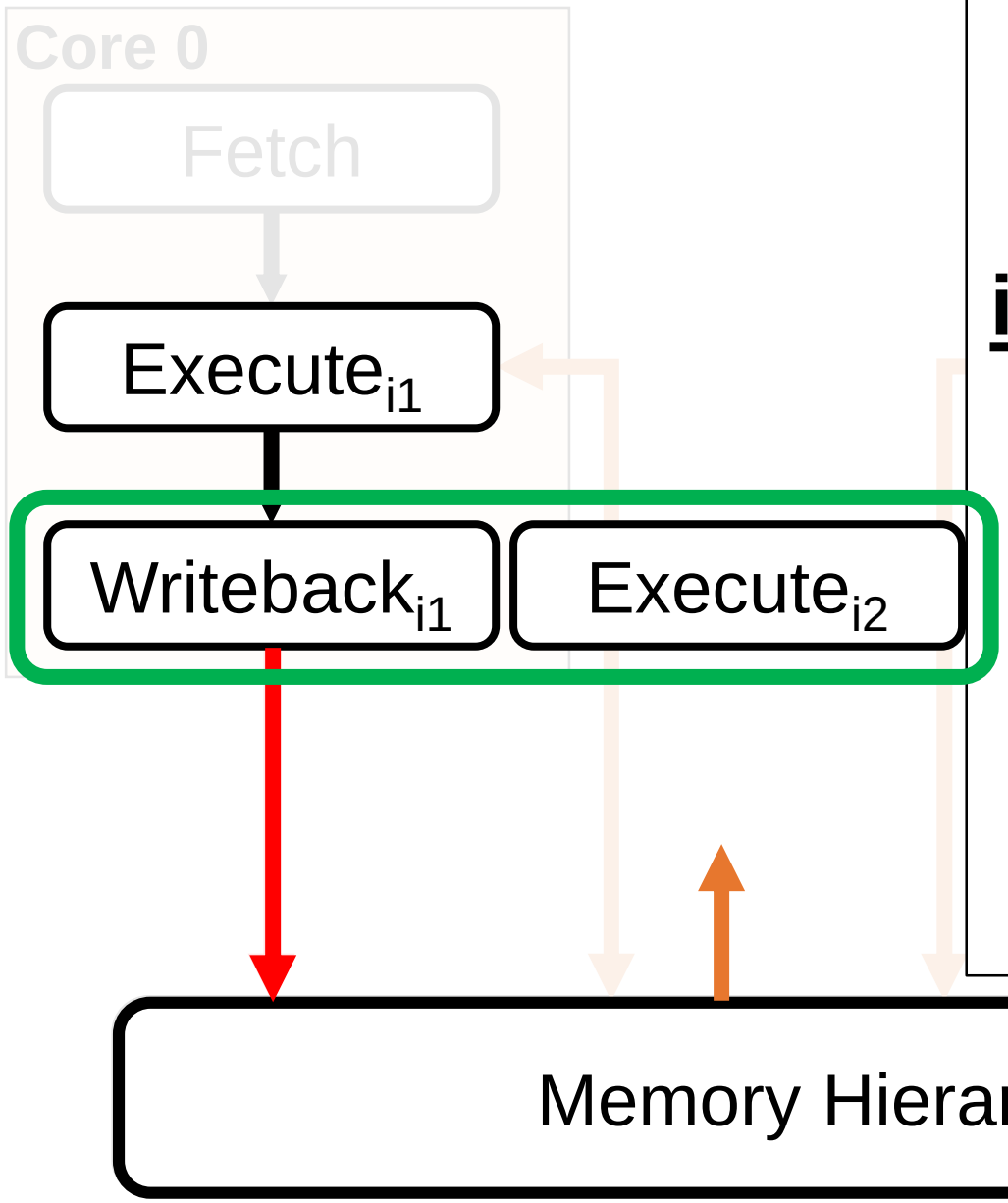
Finding Axioms



Finding Axioms



Finding Axioms



A write must reach the Memory Hierarchy before memory instructions on the same core that are after the write in program order.

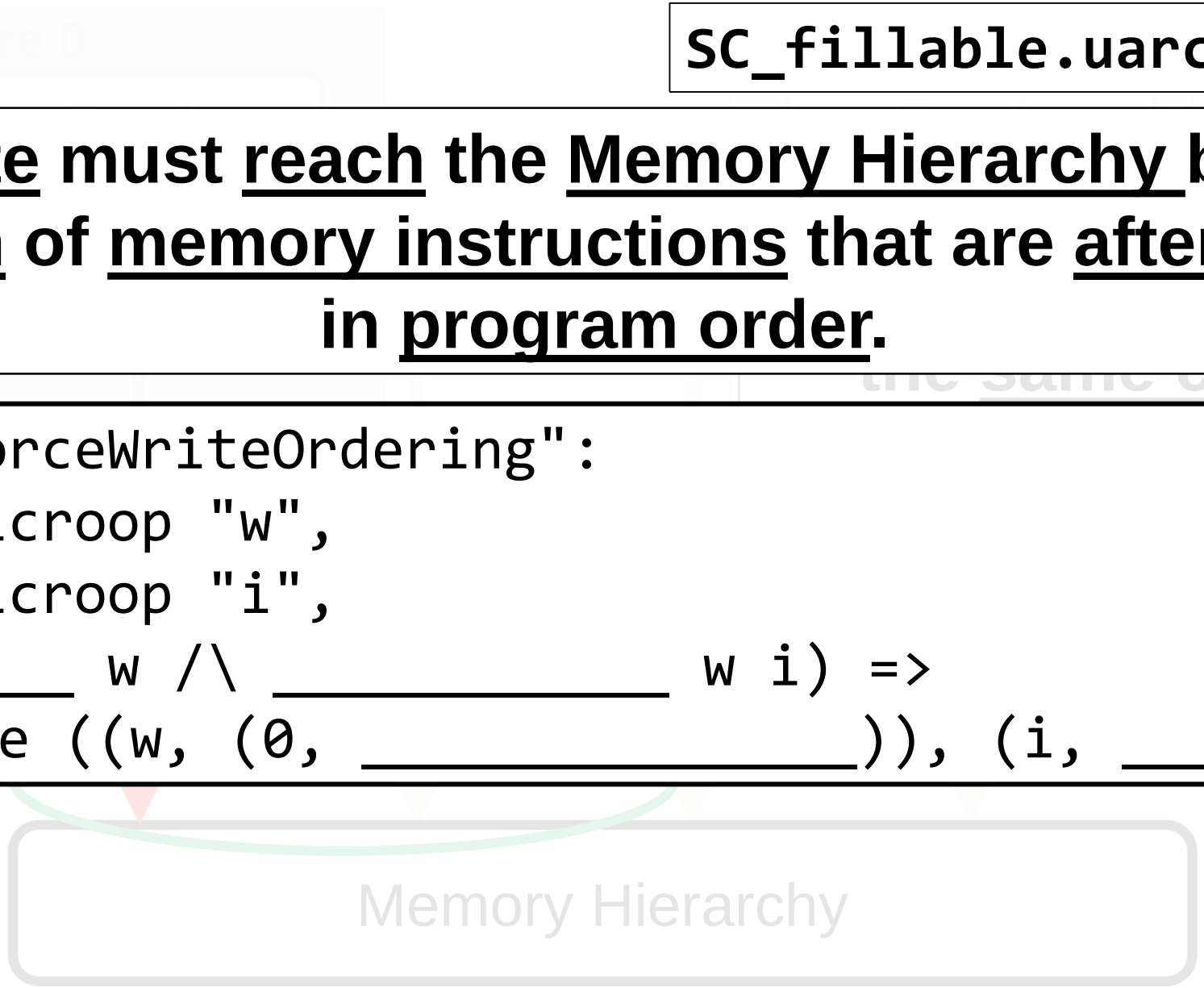
(otherwise the write could be reordered with later writes or later reads)

The Enforce_Write_Ordering Axiom

`SC_fillable.uarch, line 87`

A write must reach the Memory Hierarchy before execution of memory instructions that are after the write in program order.

```
Axiom "EnforceWriteOrdering":
  forall microop "w",
  forall microop "i",
  (__________ w /\ _____________ w i) =>
     AddEdge ((w, (0, _________________)), (i, _______)).
```



The Enforce_Write_Ordering Axiom

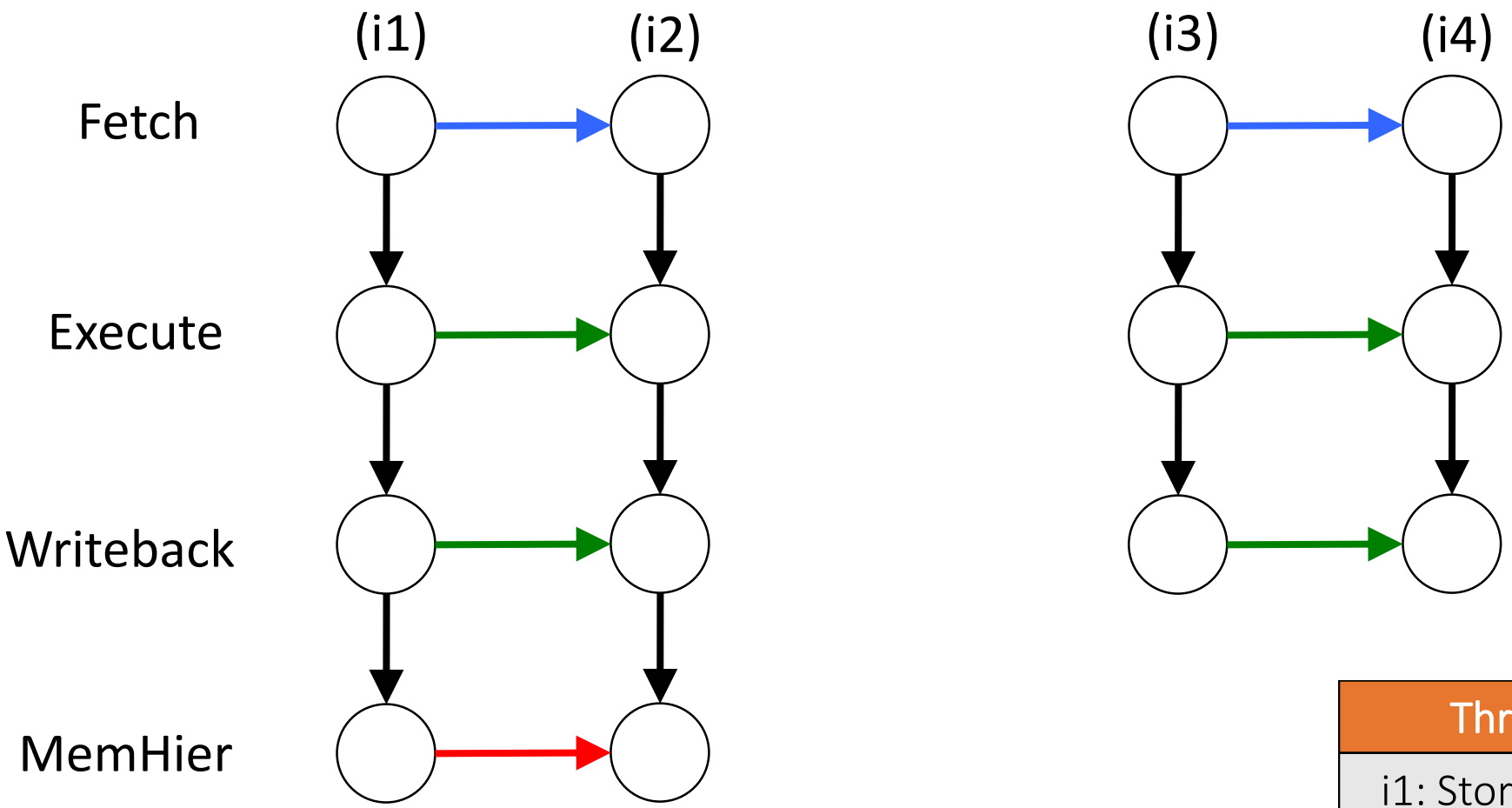
`SC_fillable.uarch, line 87`

A write must reach the Memory Hierarchy before execution of memory instructions that are after the write in program order.

```
Axiom "EnforceWriteOrdering":
  forall microop "w",
  forall microop "i",
  (IsAnyWrite w /\ ProgramOrder w i) =>
     AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)).
```


μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

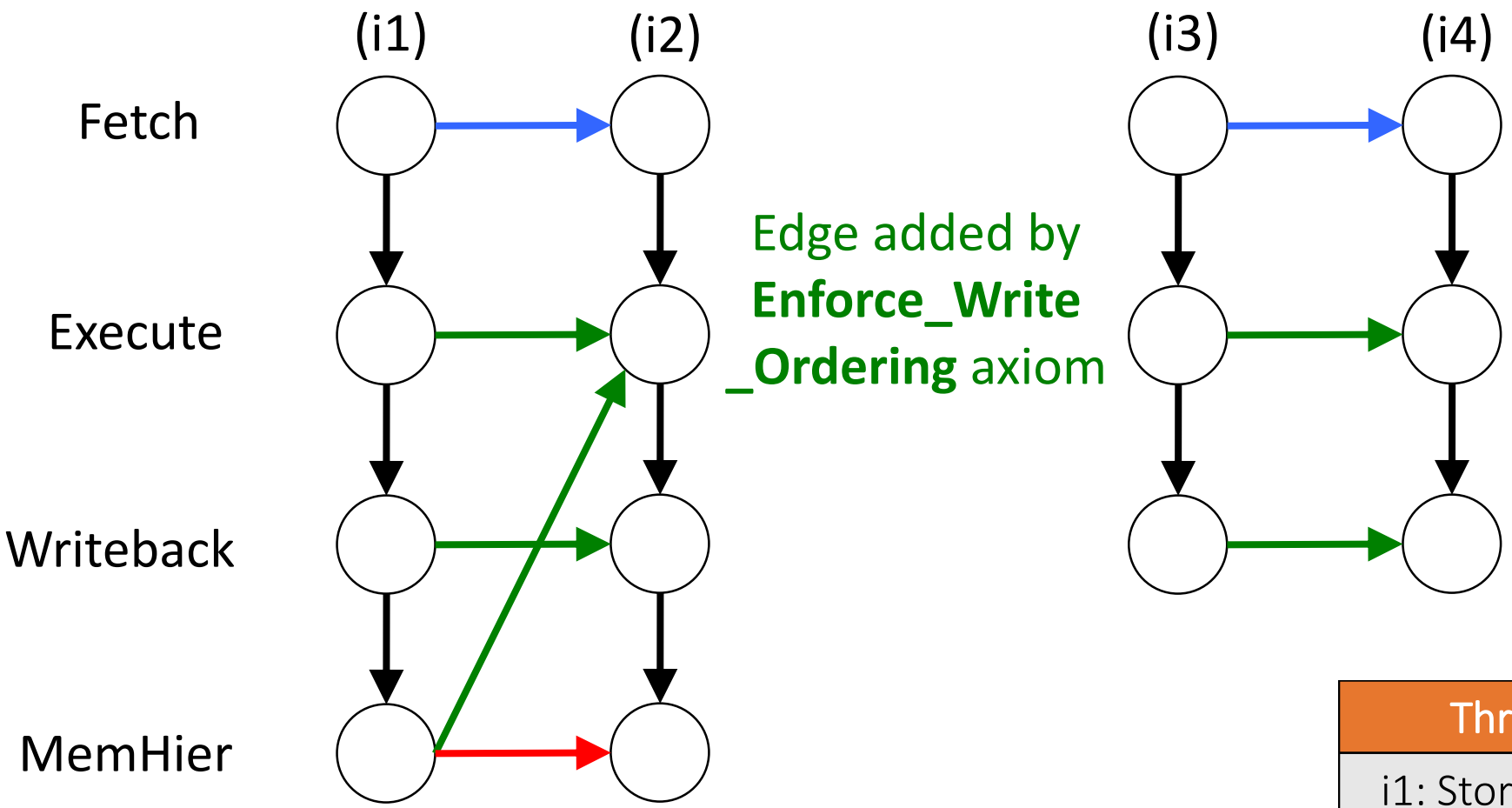


Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

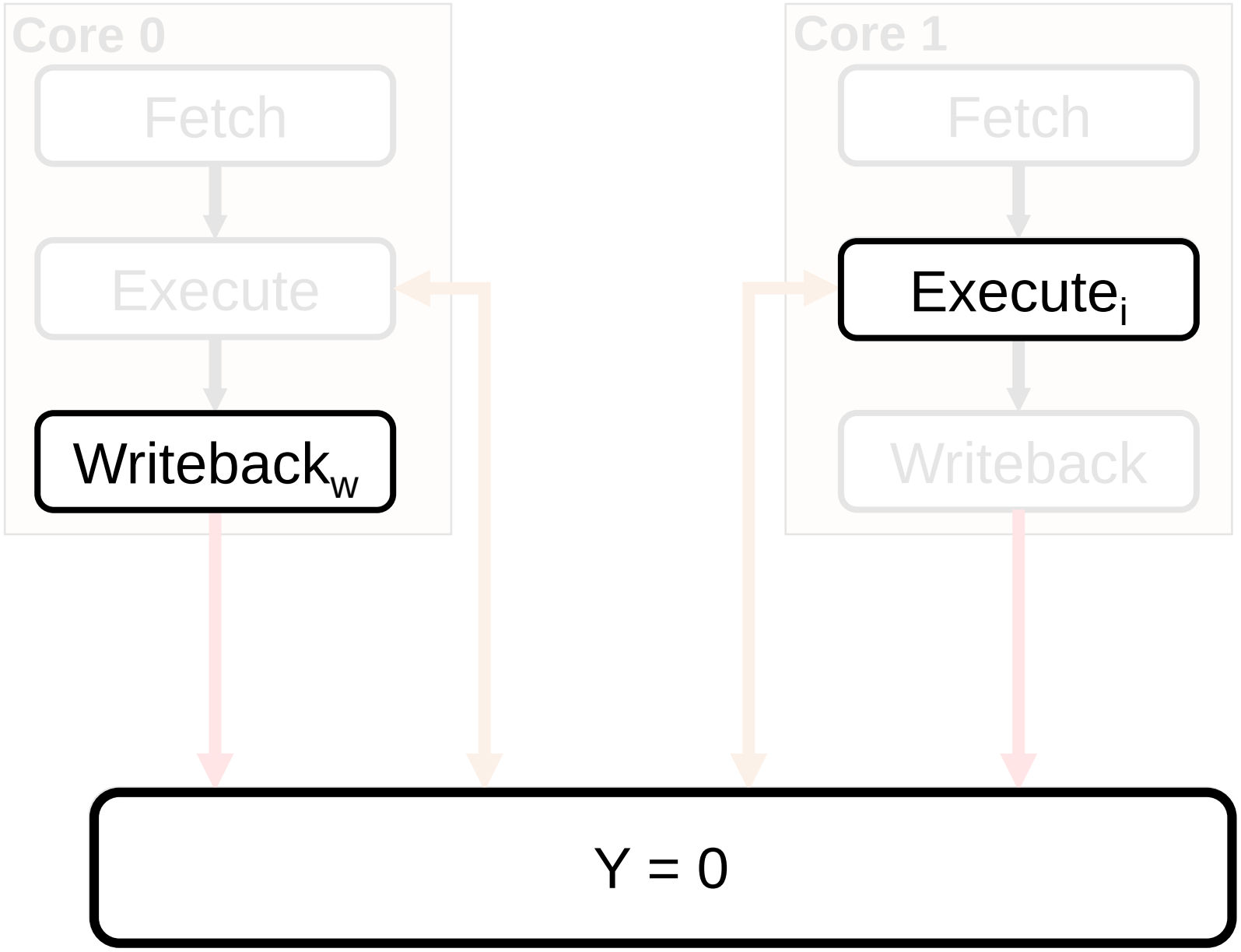
Each column represents an instruction flowing through the pipeline



Initially, Mem[x] = 0

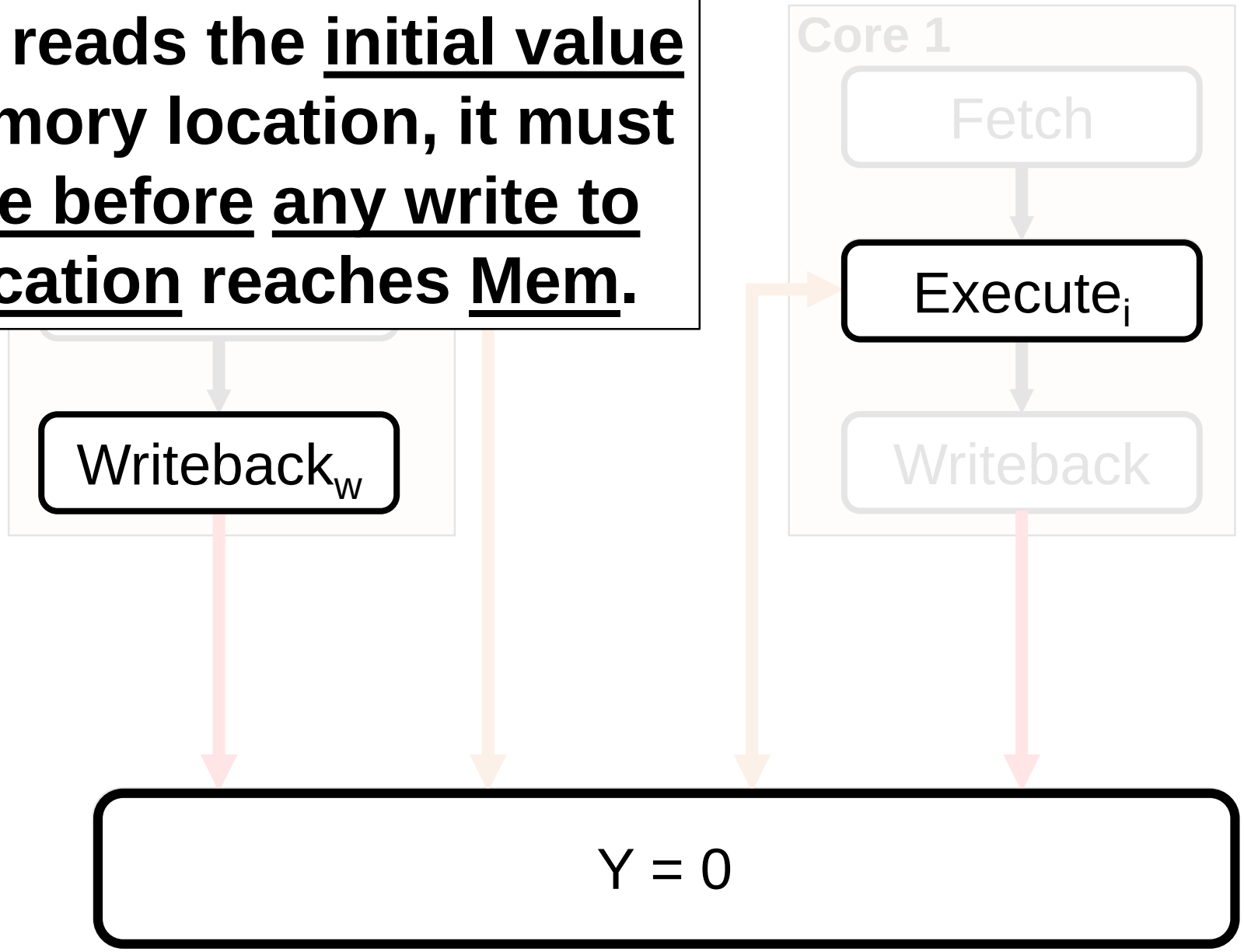
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

Finding Axioms



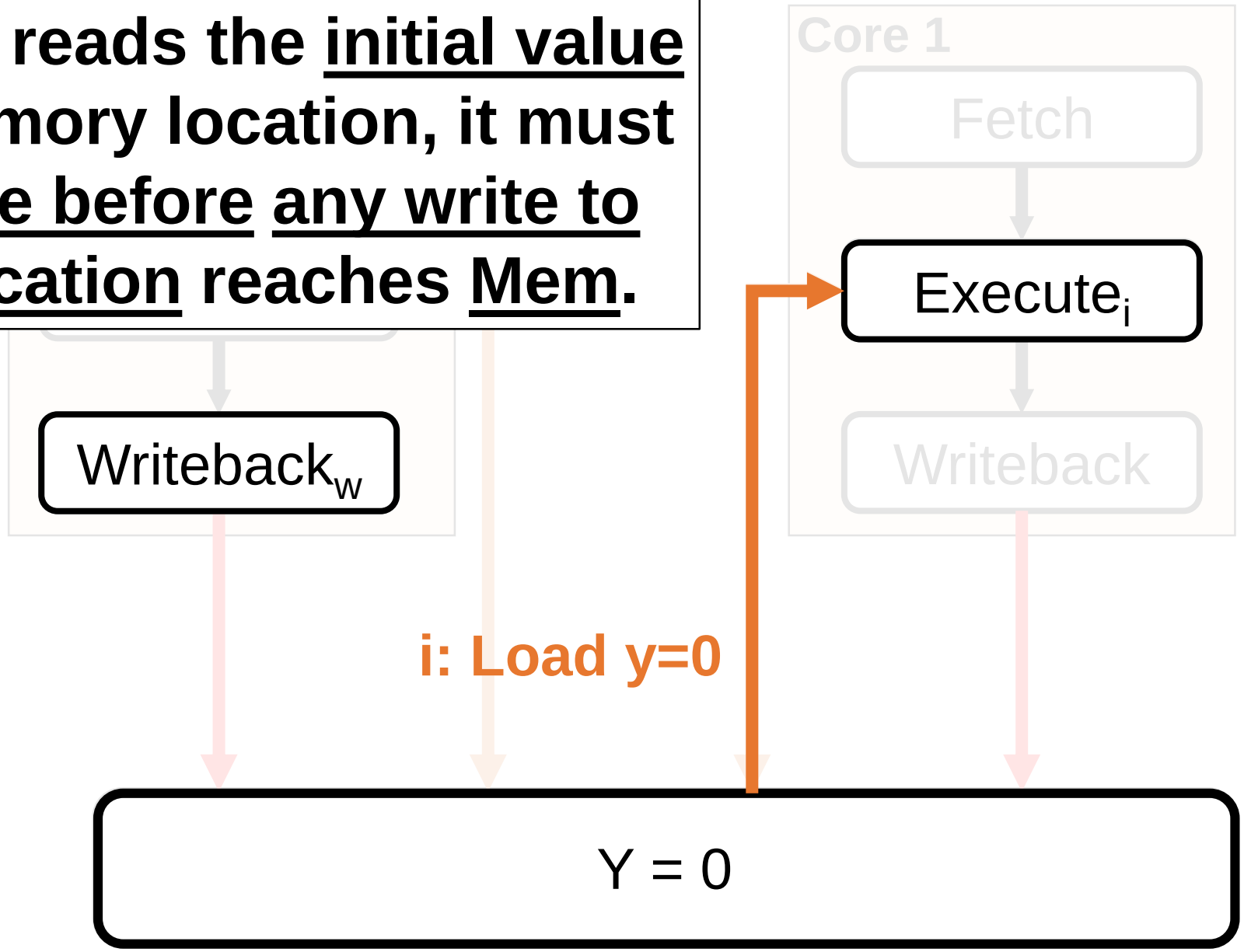
Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.



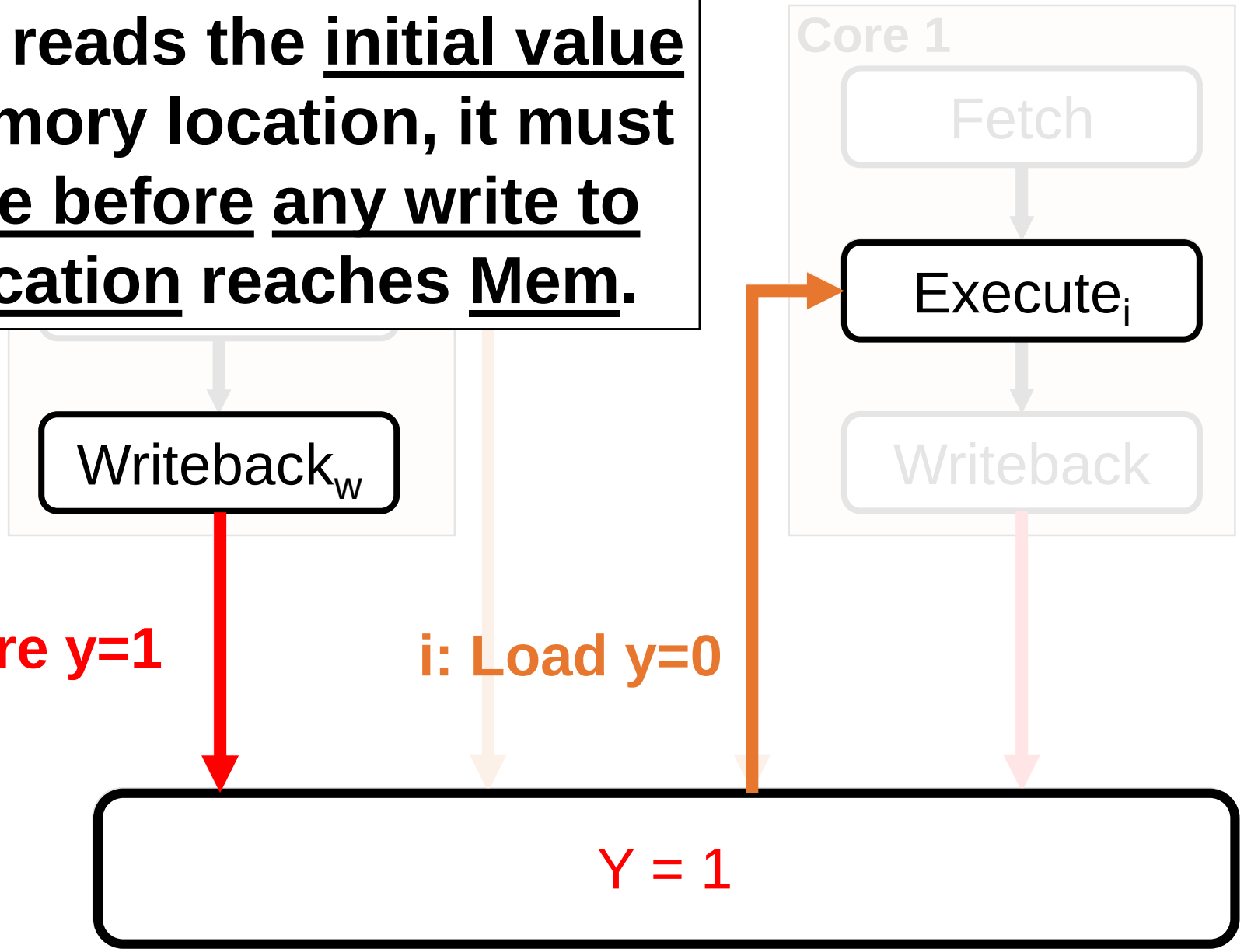
Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.



Finding Axioms

If a load reads the initial value of a memory location, it must execute before any write to that location reaches Mem.



BeforeAllWrites Macro

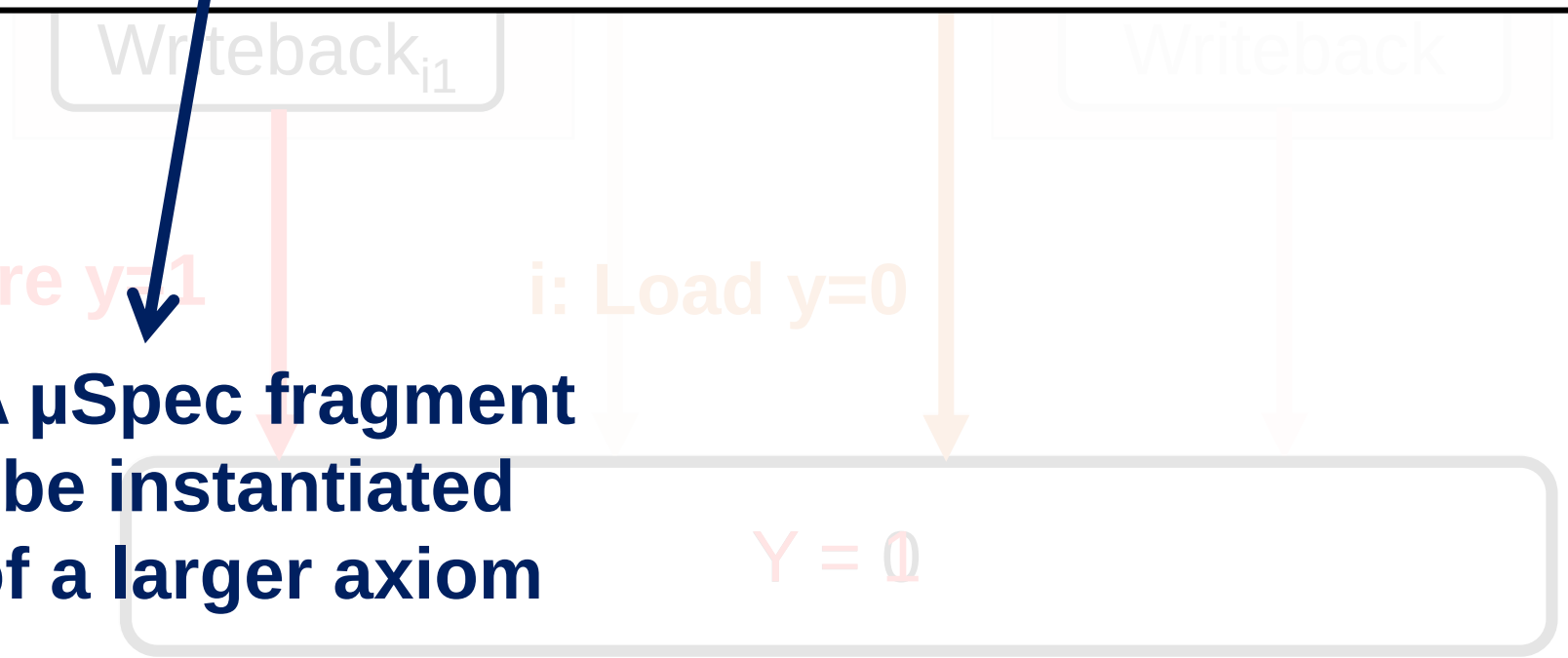
SC_fillable.uarch, line 107

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, _______), (w, (0, ________________)))).
```

BeforeAllWrites Macro

SC_fillable.uarch, line 107

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, _______), (w, (0, ________________)))).
```

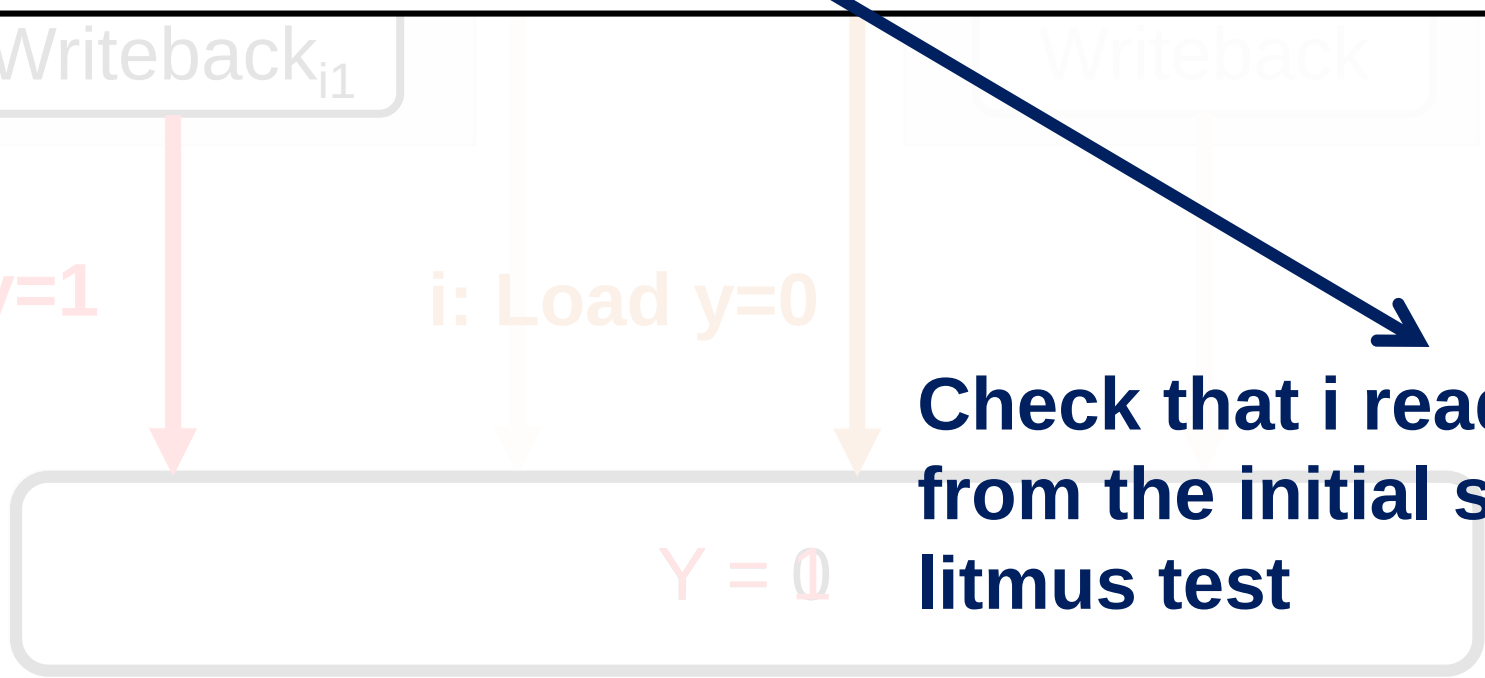


Macro: A μSpec fragment that can be instantiated as part of a larger axiom

BeforeAllWrites Macro

SC_fillable.uarch, line 107

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
   forall microop "w", (
   (IsAnyWrite w /\ SamePhysicalAddress w i
     /\ ~SameMicroop i w) =>
     AddEdge ((i, _______), (w, (0, ____________________)))).
```



Check that i reads its value from the initial state of the litmus test

BeforeAllWrites Macro

SC_fillable.uarch, line 107

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, _______), (w, (0, ________________)))).
```

If a load reads the initial value of a memory location, it must execute before any write to that addr reaches Mem.

BeforeAllWrites Macro

SC_fillable.uarch, line 107

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, Execute), (w, (0, MemoryHierarchy)))).
```

If a load reads the initial value of a memory location, it must execute before any write to that addr reaches Mem.

BeforeAllWrites Macro

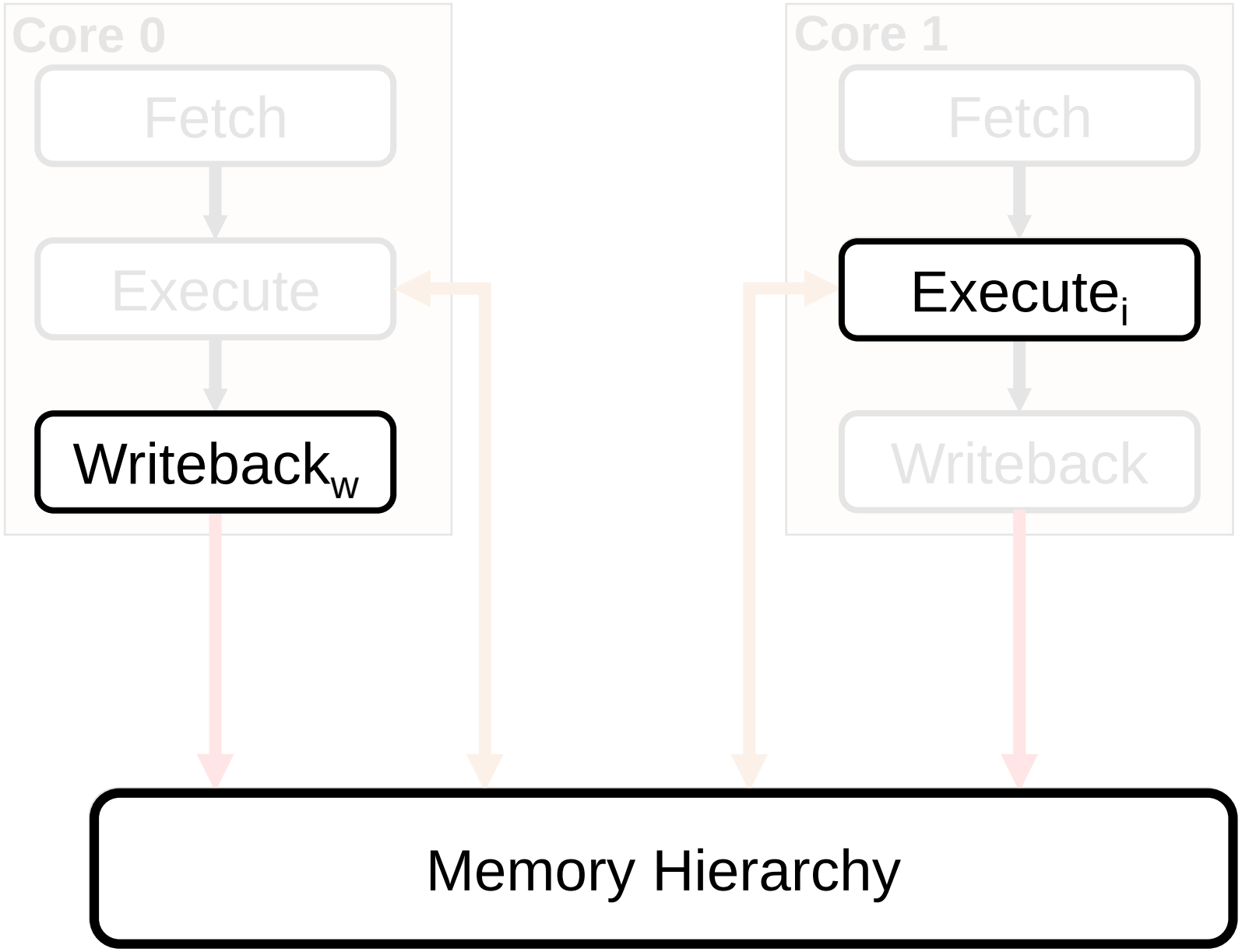
SC_fillable.uarch, line 107

```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, Execute), (w, (0, MemoryHierarchy)))).
```

If a load reads the initial value of a memory location, it must execute before any write to that addr reaches Mem.

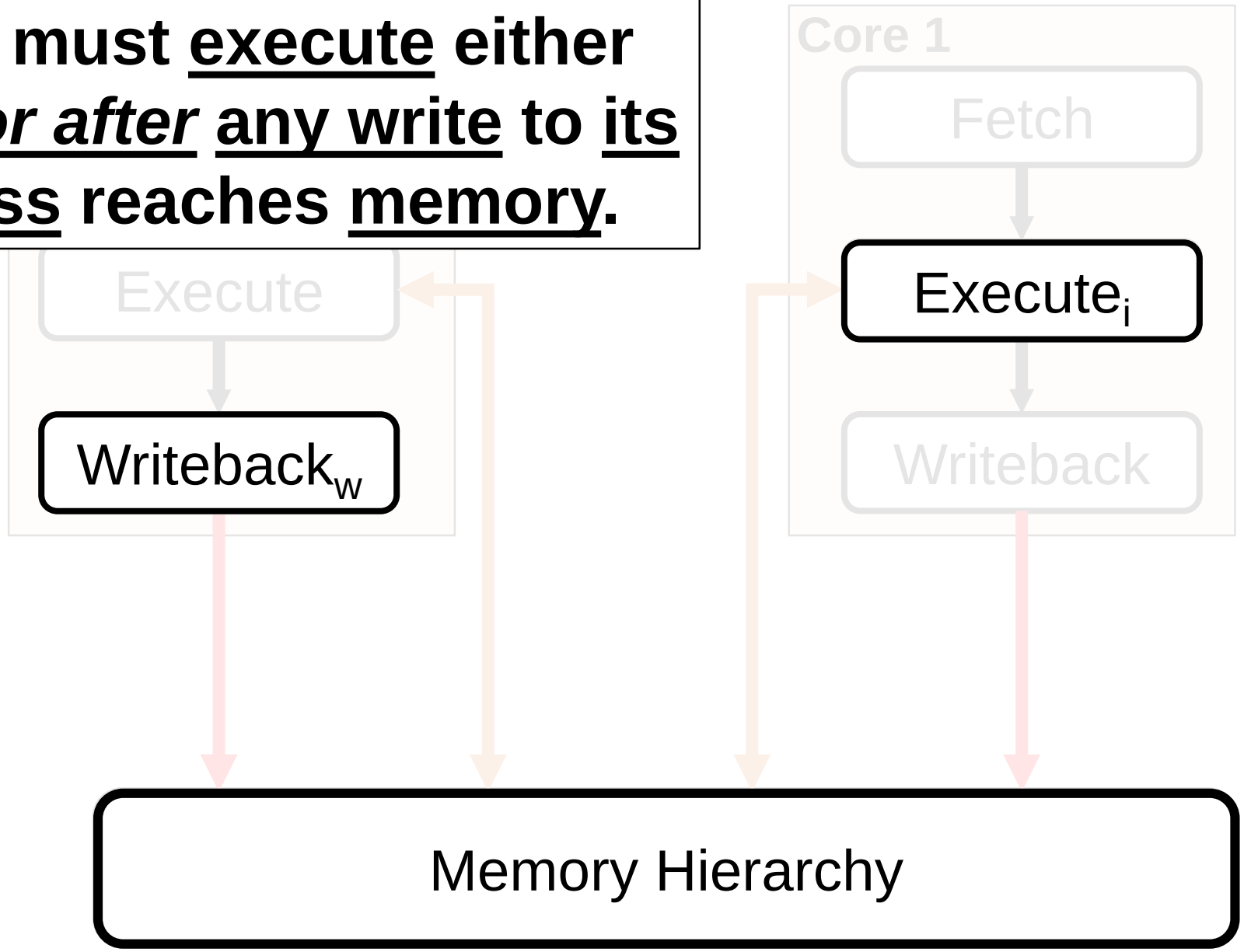
Enforce that the load executes before all writes to its address in the test

Finding Axioms



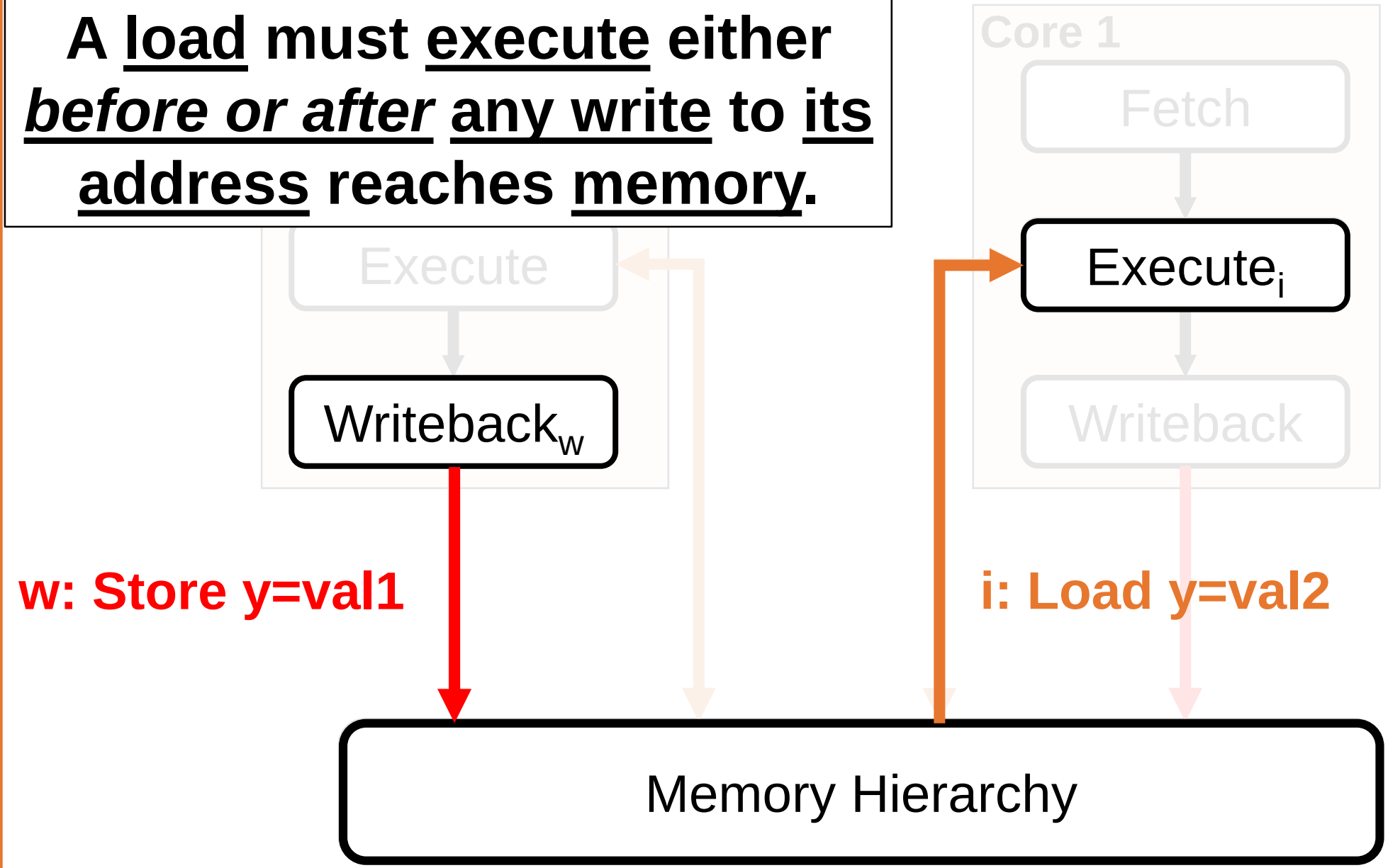
Finding Axioms

A load must execute either *before or after* any write to its address reaches memory.



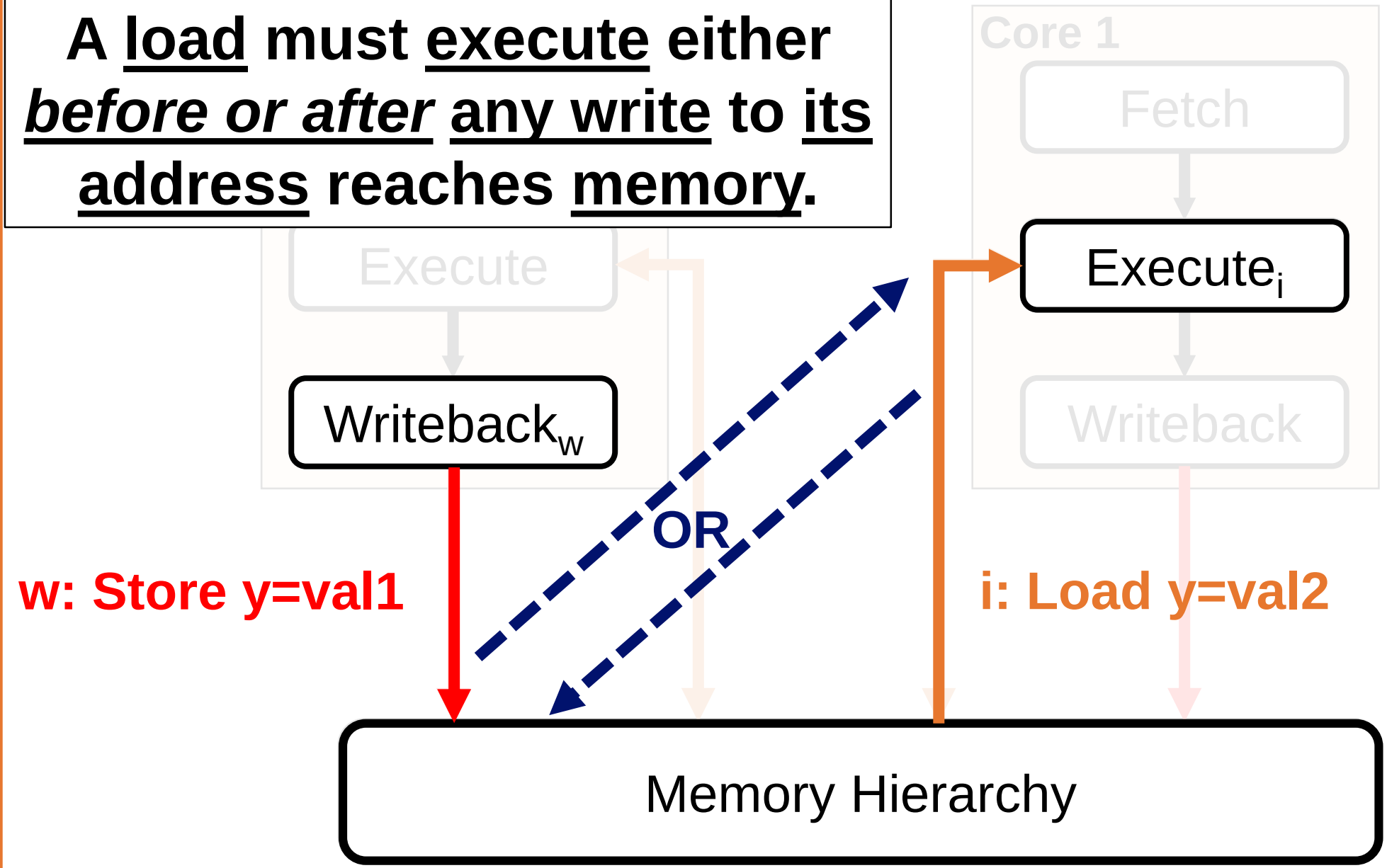
Finding Axioms

A load must execute either *before or after* any write to its address reaches memory.



Finding Axioms

A load must execute either *before or after* any write to its address reaches memory.



The Before_Or_After_Every_SameAddrWrite Macro

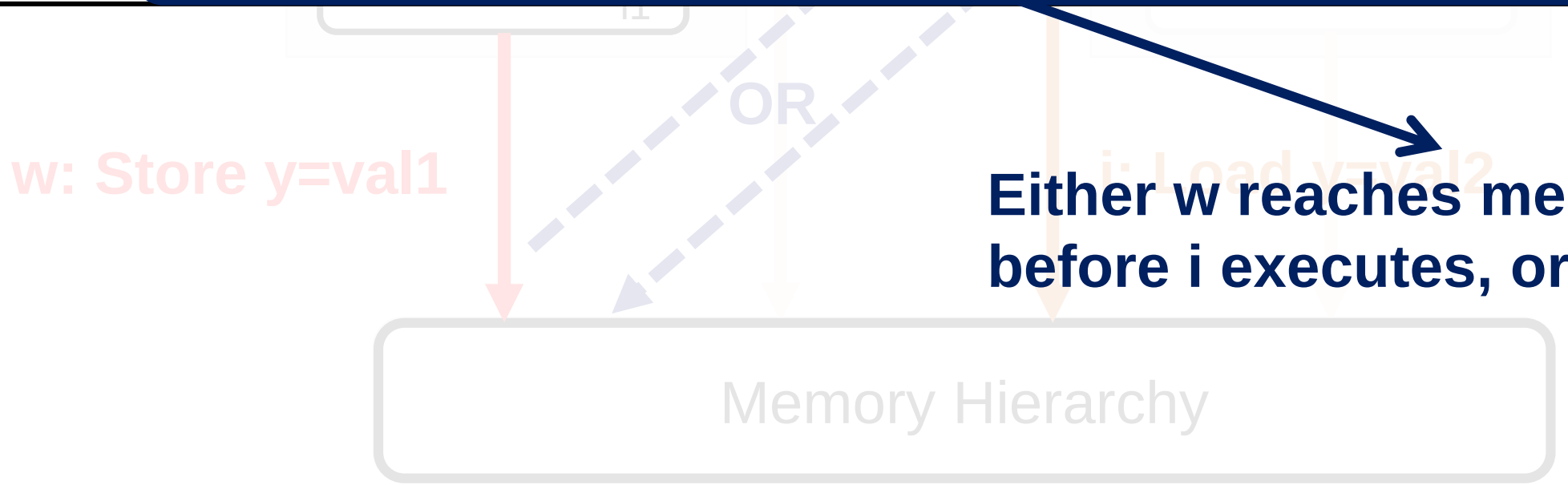
SC_fillable.uarch, line 118

```
DefineMacro "Before_Or_After_Every_SameAddrWrite":
  forall microop "w", (
    (IsAnyWrite w /\ SamePhysicalAddress w i) =>
    (AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \/
     AddEdge ((i, Execute), (w, (0, MemoryHierarchy))))).
```

The Before_Or_After_Every_SameAddrWrite Macro

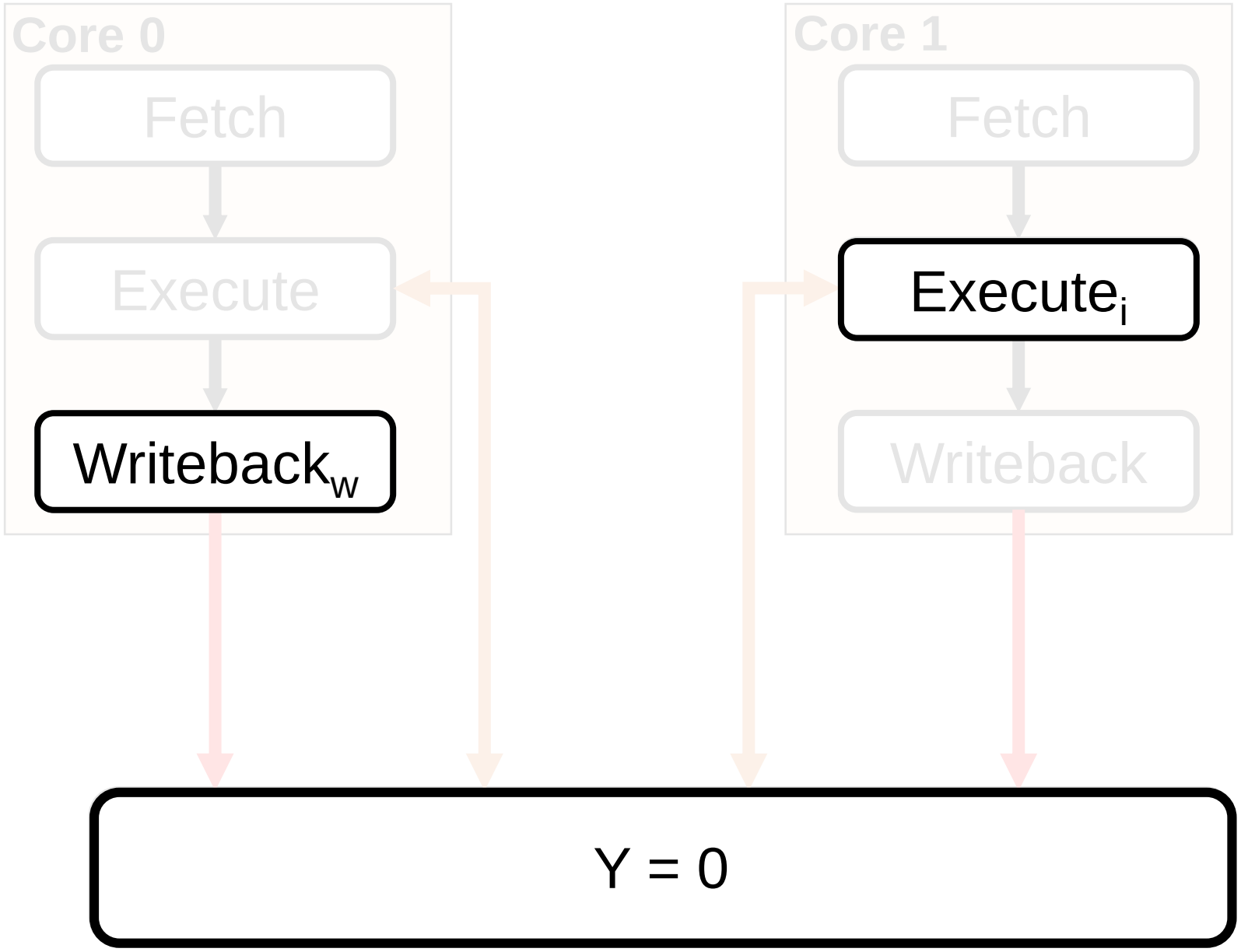
SC_fillable.uarch, line 118

```
DefineMacro "Before_Or_After_Every_SameAddrWrite":
  forall microop "w", (
    (IsAnyWrite w /\ SamePhysicalAddress w i) =>
    (AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) \/
     AddEdge ((i, Execute), (w, (0, MemoryHierarchy))))).
```



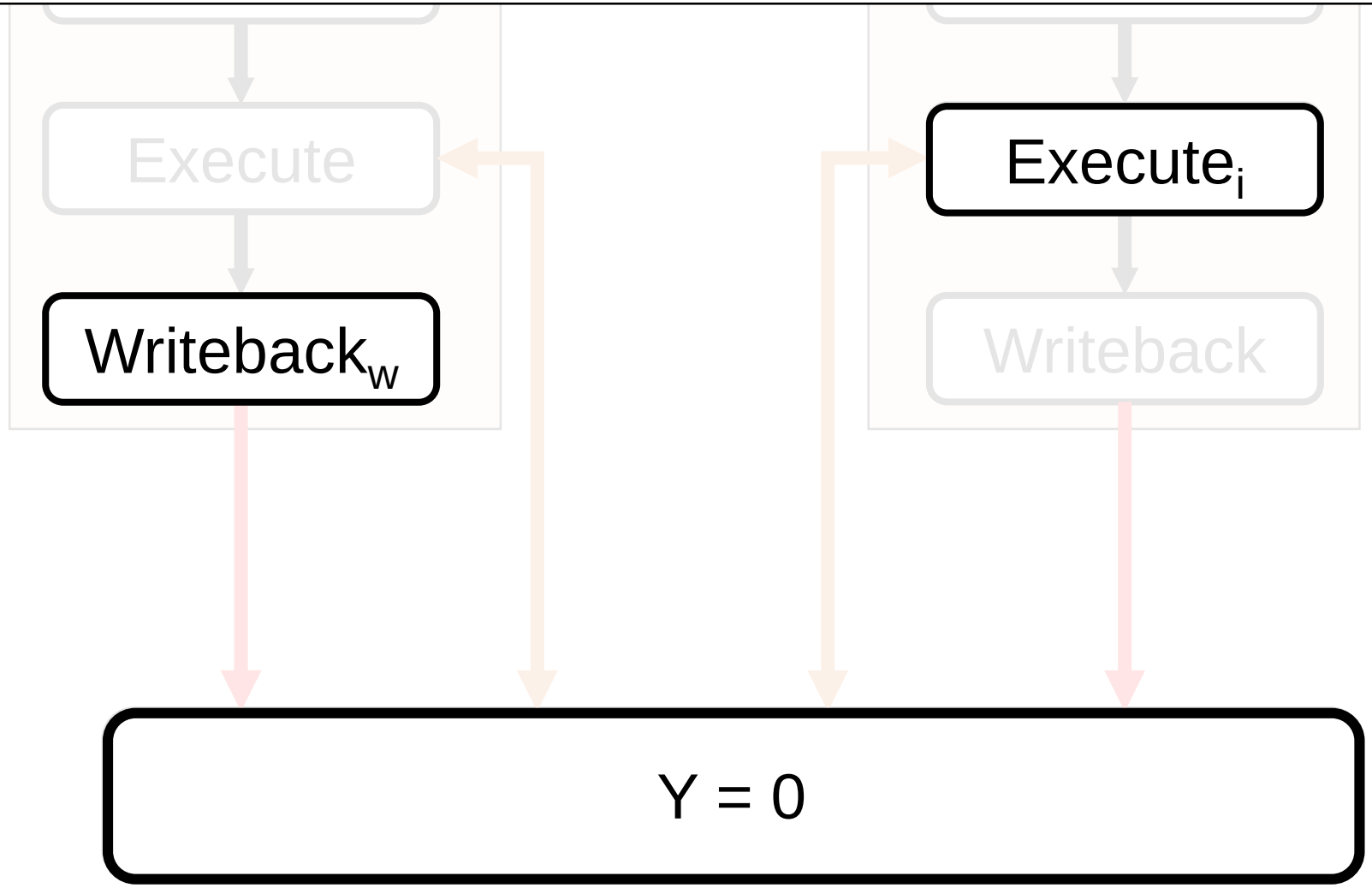
Either w reaches memory before i executes, or vice-versa.

Finding Axioms



Finding Axioms

A load must read from the latest write to that address to reach memory.

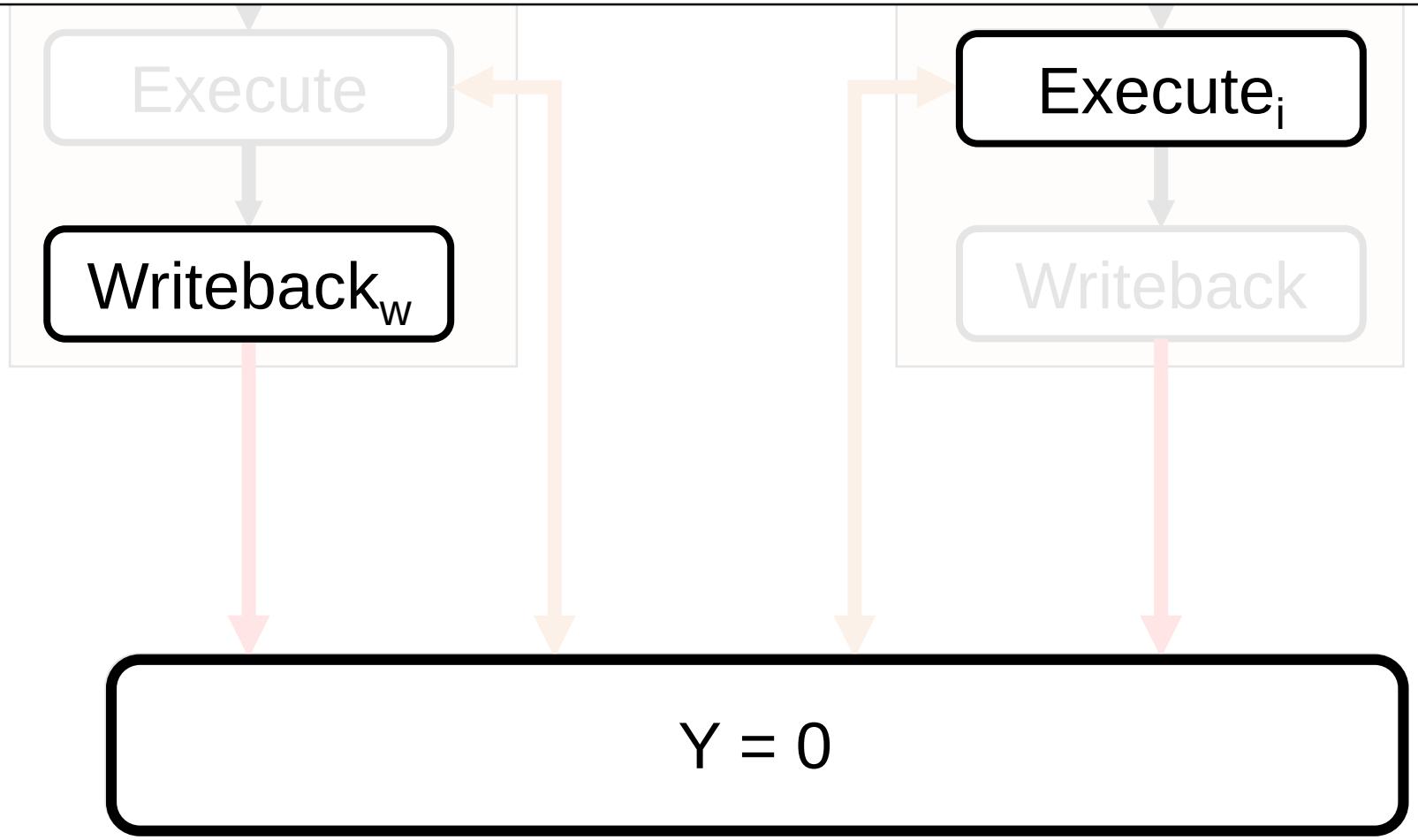


Finding Axioms

Alternatively:

1) The load must execute after the write it reads from

2) No writes to that address between the source write and the read

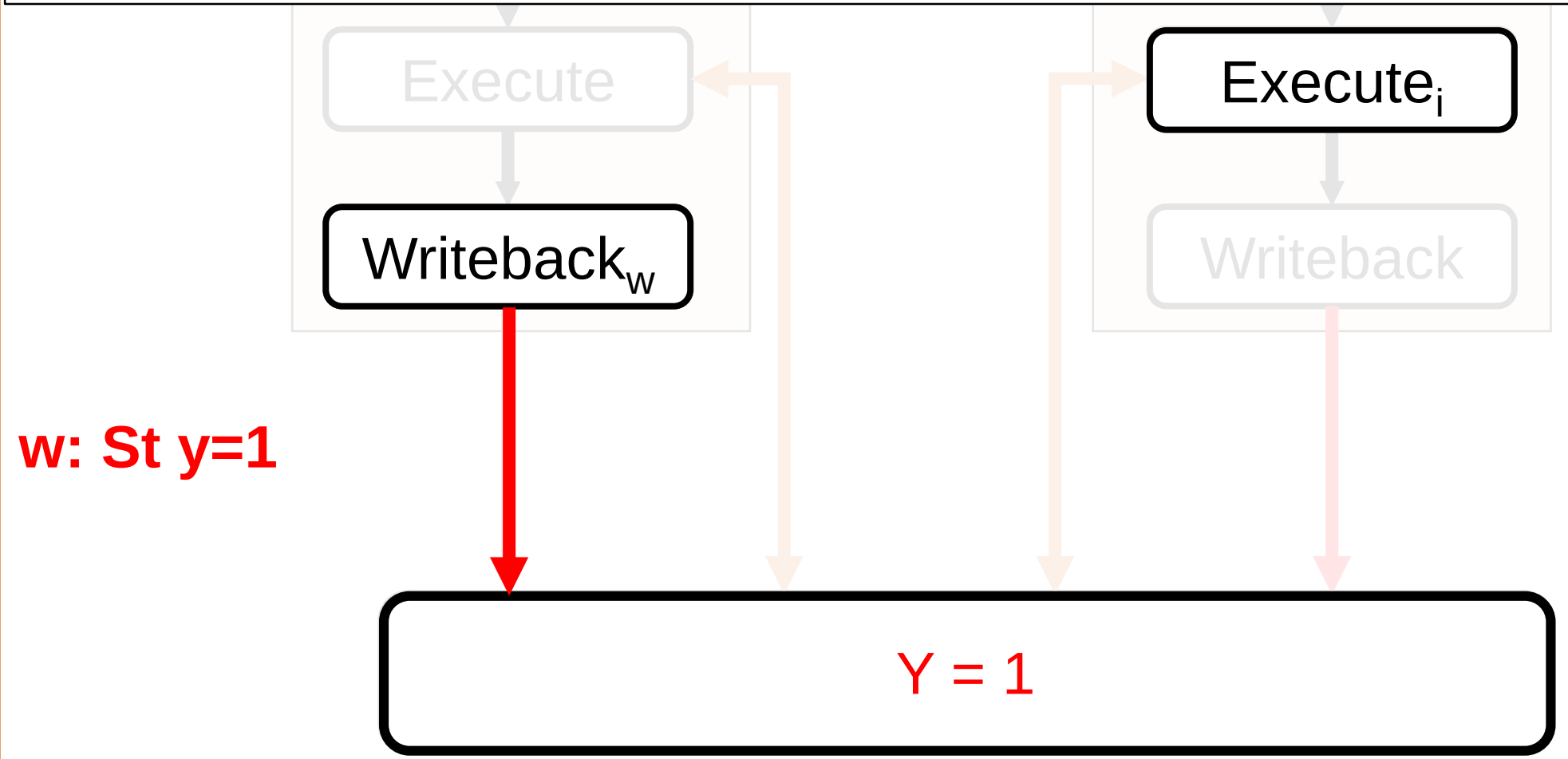


Finding Axioms

Alternatively:

1) The load must execute after the write it reads from

2) No writes to that address between the source write and the read

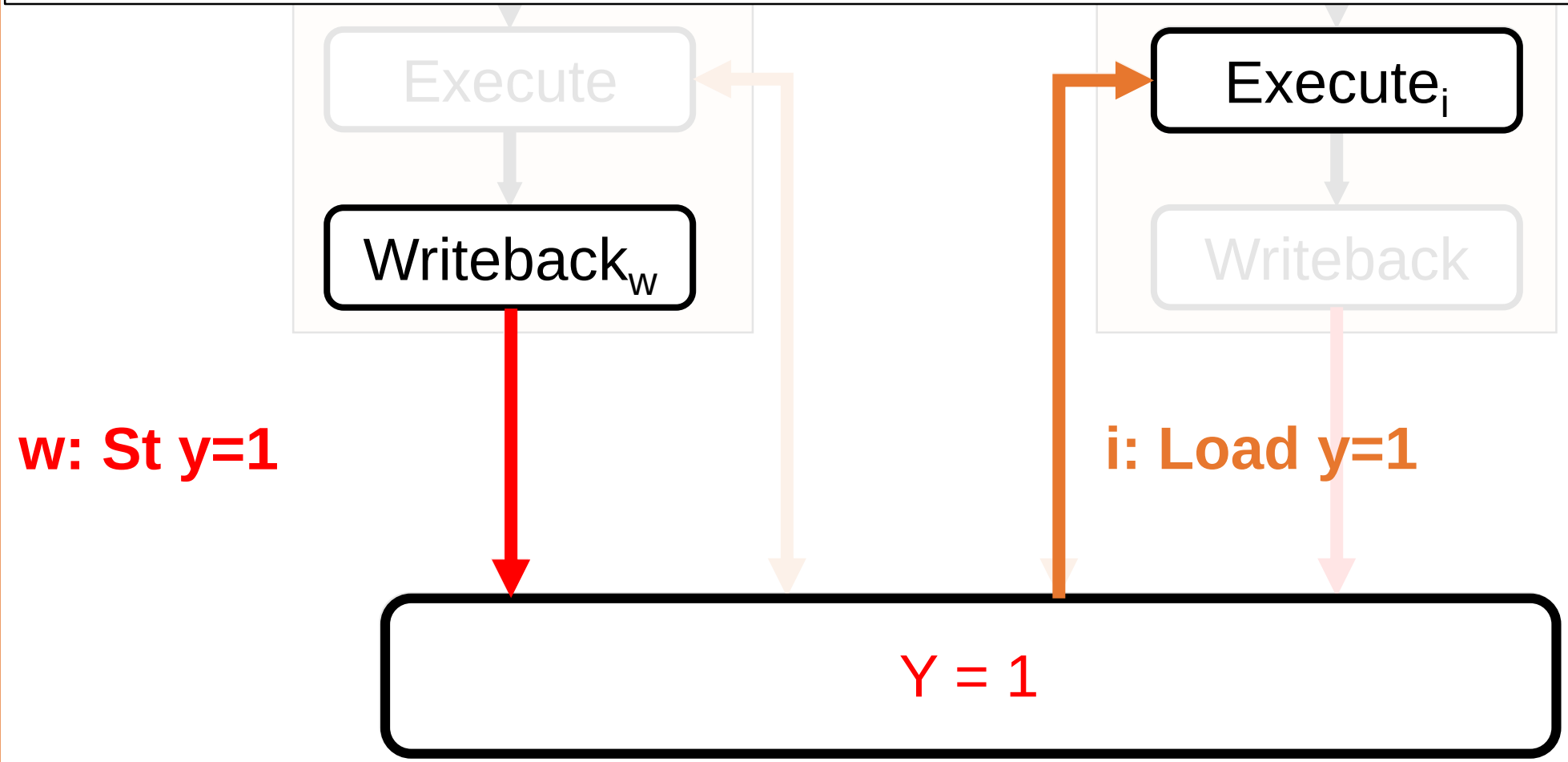


Finding Axioms

Alternatively:

1) The load must execute after the write it reads from

2) No writes to that address between the source write and the read

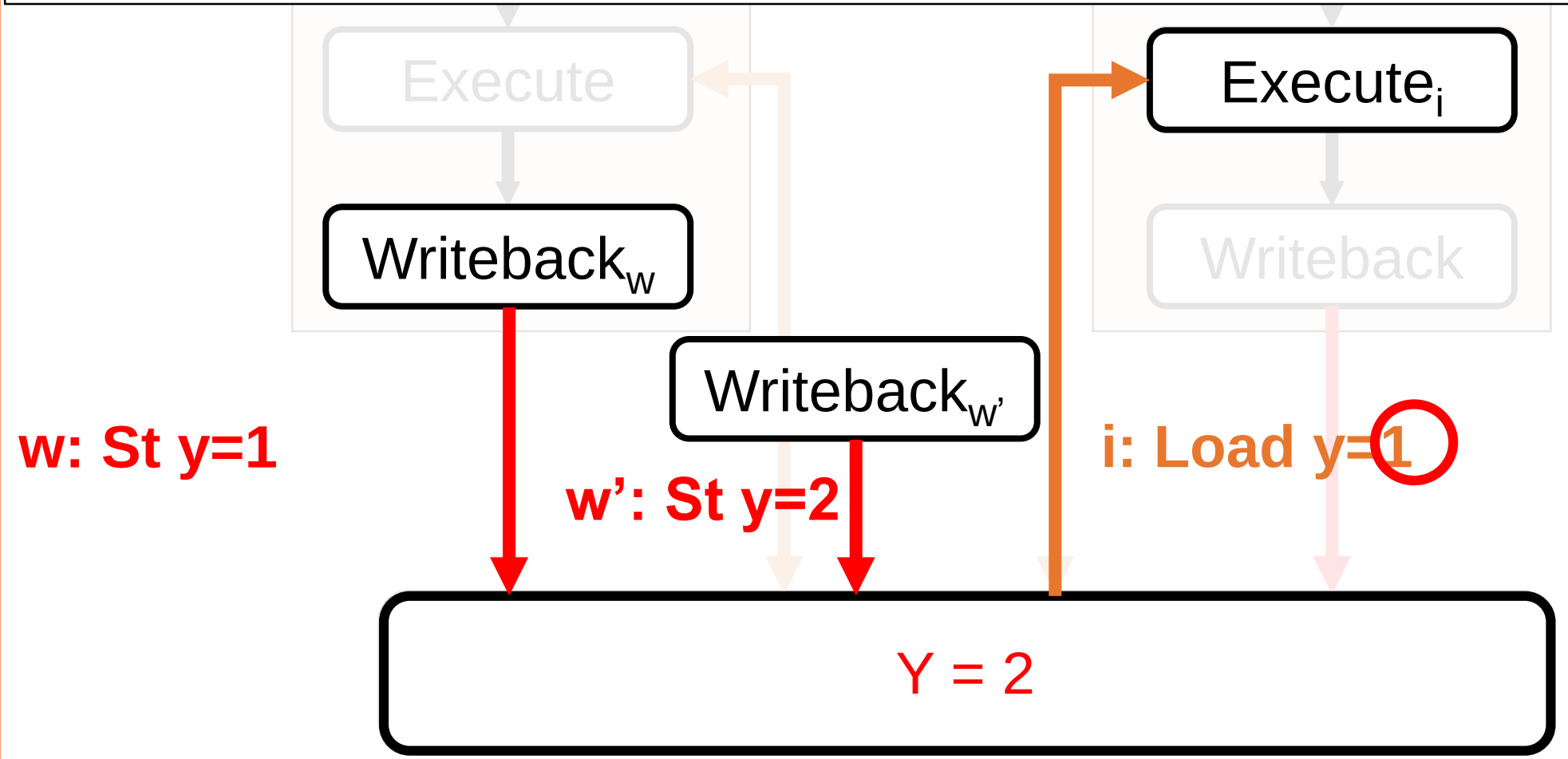


Finding Axioms

Alternatively:

1) The load must execute after the write it reads from

2) No writes to that address between the source write and the read



The No_SameAddrWrites_Btwn_Src_And_Read Macro

SC_fillable.uarch, line 135

Alternatively:

```
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
 exists microop "w", (
  IsAnyWrite w /\ ____________________ w i /\ ________ w i
  /\ AddEdge ((w, (0, MemHier)), (i, Execute)) /\
  ~(exists microop "w'",
    IsAnyWrite w' /\ ____________________ i w’ /\
    ~SameMicroop w w’
    /\ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier)));
                   ((w', (0,MemHier)), (i, Execute))])).
```

1) The load must execute after the write it reads from

2) No writes to that address between the source write and the read

The No_SameAddrWrites_Btwn_Src_And_Read Macro

Alternatively:

SC_fillable.uarch, line 135

```
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
 exists microop "w", (
  IsAnyWrite w /\ SamePhysicalAddress w i /\ SameData w i
  /\ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) /\
  ~(exists microop "w'",
    IsAnyWrite w' /\ SamePhysicalAddress i w’ /\
    ~SameMicroop w w’
    /\ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier)));
                   ((w', (0,MemHier)), (i, Execute))])).
```

The No_SameAddrWrites_Btwn_Src_And_Read Macro

SC_fillable.uarch, line 135

Alternatively:

```
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
 exists microop "w", (
  IsAnyWrite w /\ SamePhysicalAddress w i /\ SameData w i
  /\ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) /\
  ~(exists microop "w'",
    IsAnyWrite w' /\ SamePhysicalAddress i w’ /\
    ~SameMicroop w w’
    /\ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier)));
                   ((w', (0,MemHier)), (i, Execute))])).
```

Read i executes after its source write w reaches memory…

The No_SameAddrWrites_Btwn_Src_And_Read Macro

SC_fillable.uarch, line 135

Alternatively:

```
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
 exists microop "w", (
  IsAnyWrite w /\ SamePhysicalAddress w i /\ SameData w i
  /\ AddEdge ((w, (0, MemoryHierarchy)), (i, Execute)) /\
  ~(exists microop "w'",
    IsAnyWrite w' /\ SamePhysicalAddress i w’ /\
    ~SameMicroop w w’
    /\ EdgesExist [((w, (0,MemHier)), (w', (0,MemHier)));
                   ((w', (0,MemHier)), (i, Execute))])).
```

…and there are no writes w’ to that addr between the source write w and the read i.

Putting the Macros together: the Read_Values axiom

Alternatively:

SC_fillable.uarch, line 149

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (
    ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    /\
    ExpandMacro Before_Or_After_Every_SameAddrWrite
  )).
```

Putting the Macros together: the Read_Values axiom

Alternatively:

SC_fillable.uarch, line 149

```
Axiom "Read Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (
    ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    /\
    ExpandMacro Before_Or_After_Every_SameAddrWrite
  )).
```

For all reads i (same identifier used in the macros)…

Y = ∅

Putting the Macros together: the Read_Values axiom

Alternatively:

SC_fillable.uarch, line 149

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (
    ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    /\
    ExpandMacro Before_Or_After_Every_SameAddrWrite
  )).
```

…either the read executes before all writes (expand macro defined earlier)…

Y = 0

Putting the Macros together: the Read_Values axiom

Alternatively:

SC_fillable.uarch, line 149

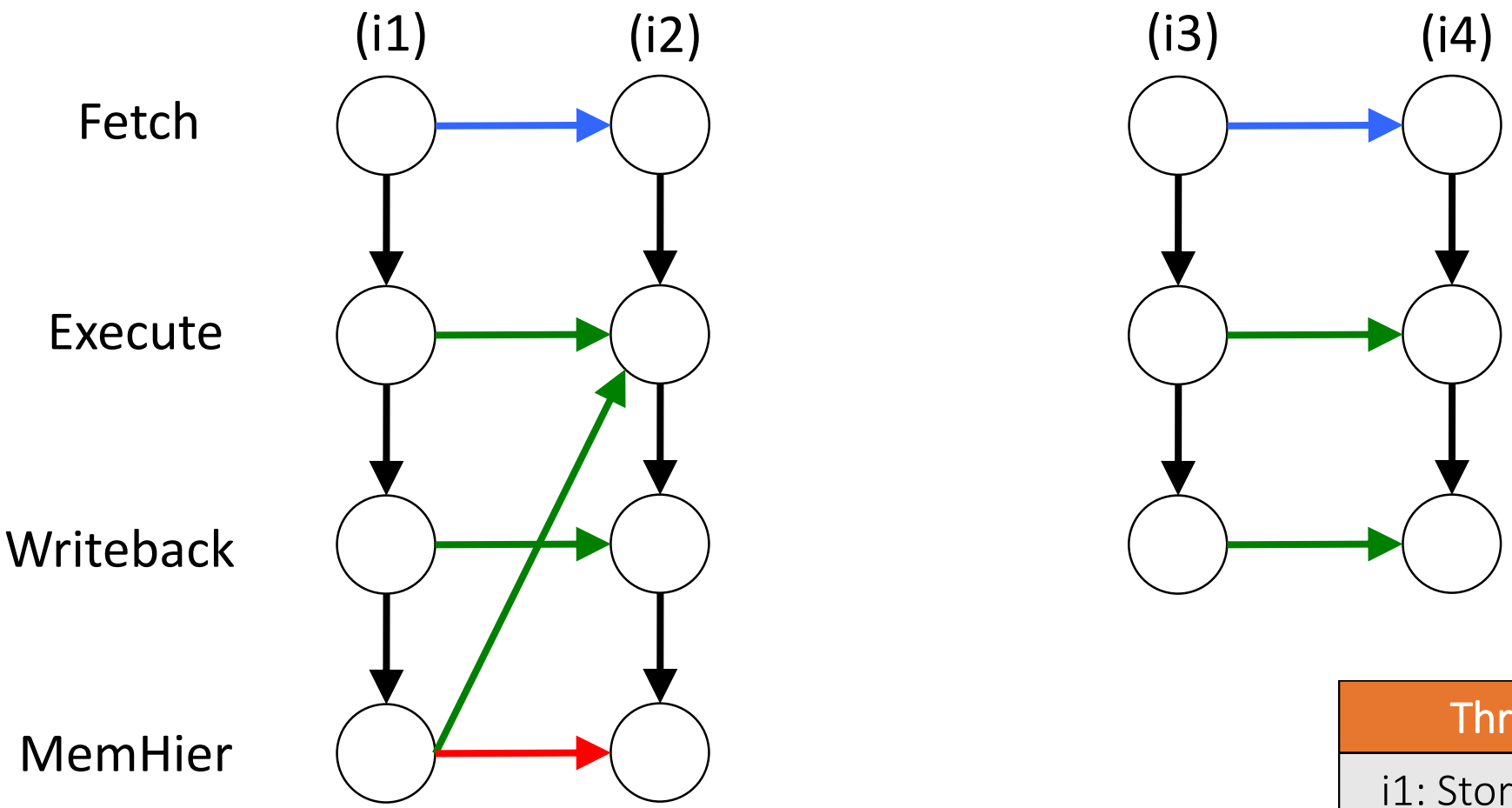
```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (
    ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    /\
    ExpandMacro Before_Or_After_Every_SameAddrWrite
  )).
```

…or the read reads from the latest write to that address

Y = 0

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

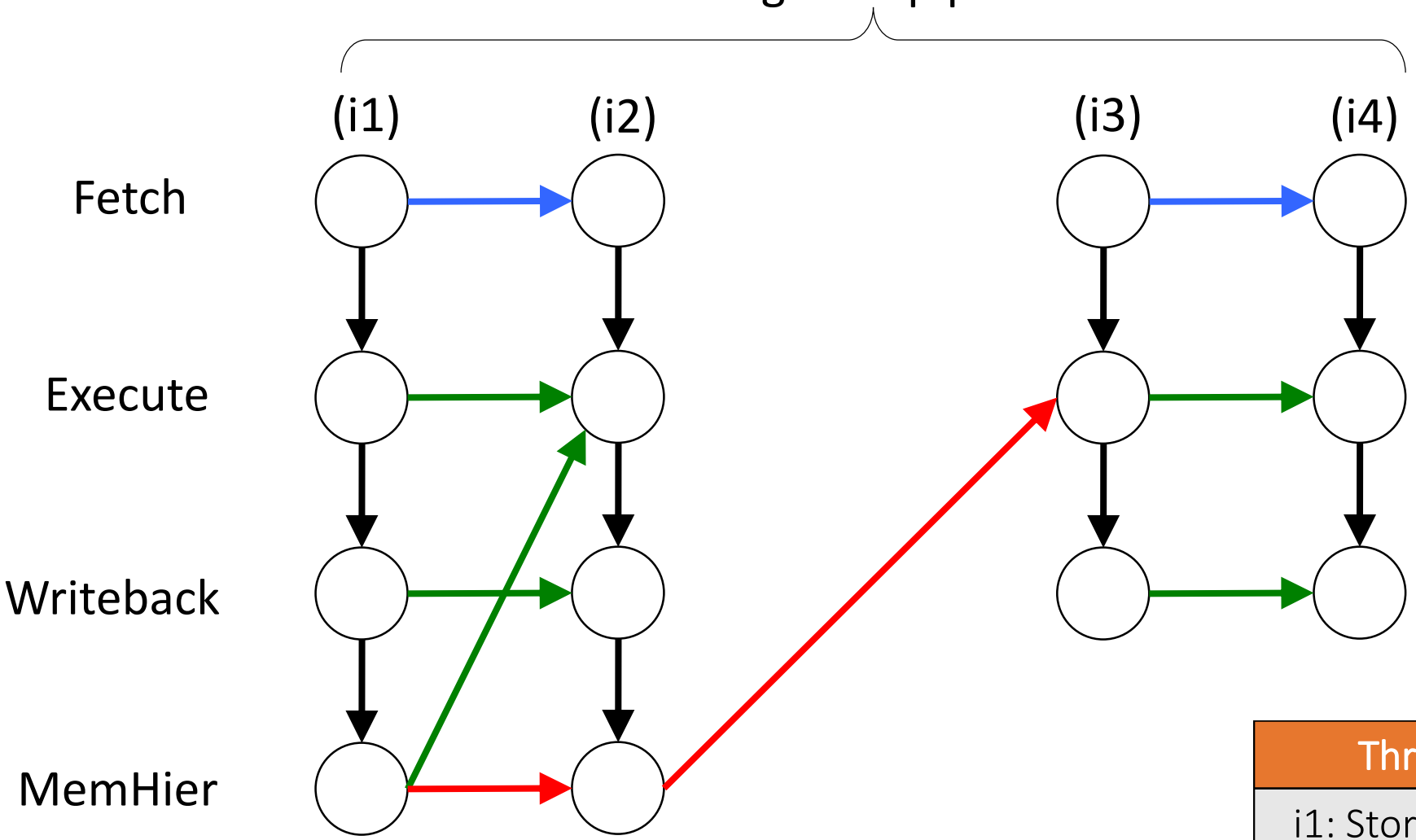


Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline



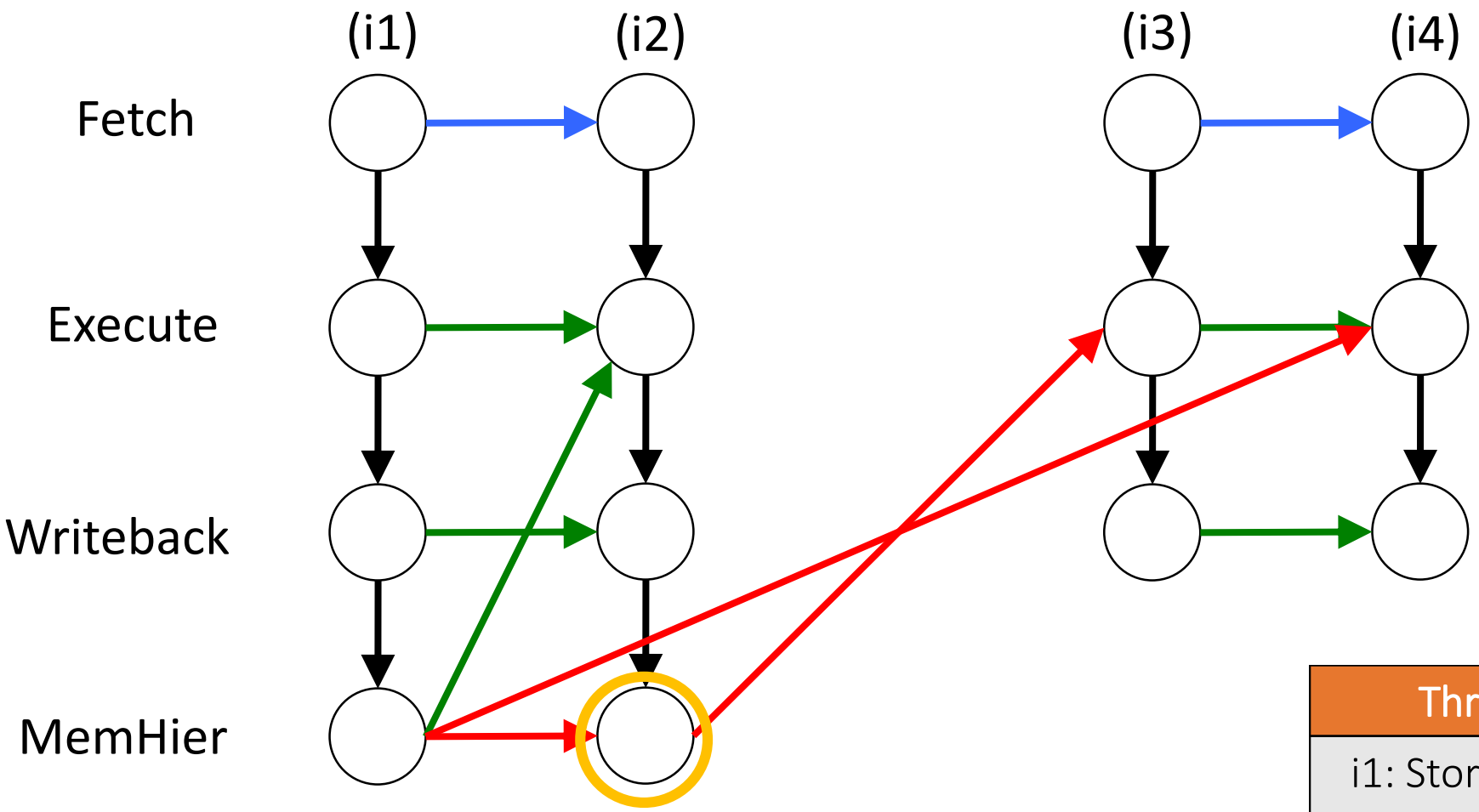
- i3 must be sourced from the write i2
- No intervening writes; **constraint satisfied**

Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline



- i4 must be sourced from i1
- But i2 intervenes! => **Constraint unsatisfiable**

Initially, Mem[x] = 0

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

μhb Graphs for co-mp Using Axioms

Each column represents an instruction flowing through the pipeline

(i1) (i2) (i3) (i4)

Cannot find an **acyclic** graph that **satisfies** all constraints =>

Forbidden Execution of co-mp is NOT observable on μarch!

MemHier

Initially, Mem[x] = 0

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

Test your completed SC uarch!

```
# Assuming you are in ~/pipecheck_tutorial/uarches/
$ check -i ../tests/SC_tests/co-mp.test -m SC_fillable.uarch

# If your uarch is valid, the above will create co-mp.pdf in your
# current directory (open pdfs from command line with evince)
# To run the solution version of the SC uarch on this test:
# (Note: this will overwrite the co-mp.pdf in your current folder)
$ check -i ../tests/SC_tests/co-mp.test -m SC.uarch -d solutions/

# If you get an error (cannot parse uarch, ps2pdf crashes, etc),
# examine your syntax or ask for help.
# If the outcome is observable ("BUG"), compare the graphs
# generated by the solution uarch to those of your uarch.

# To compare the uarches themselves:
$ diff SC_fillable.uarch solutions/SC.uarch
```

Run the entire suite of SC litmus tests!

```
# Assuming you are in ~/pipecheck_tutorial/uarches/
$ run_tests -v 2 -t ../tests/SC_tests/ -m SC_fillable.uarch

# The above will generate *.gv files in ~/pipecheck_tutorial/out/
# for all SC tests, and output overall statistics at the end. If
# the count for "Buggy" is non-zero, your uarch is faulty. Look for
the tests that output "BUG" to find out which tests fail.

# You can use gen_graph to convert gv files into PDFs:
$ gen_graph -i <test_gv_file>

# Compare your uarch with the solution SC uarch using diff to find
# discrepancies:
$ diff SC_fillable.uarch solutions/SC.uarch
```

Coffee Break!

After the break: Extending SC uarch. to TSO

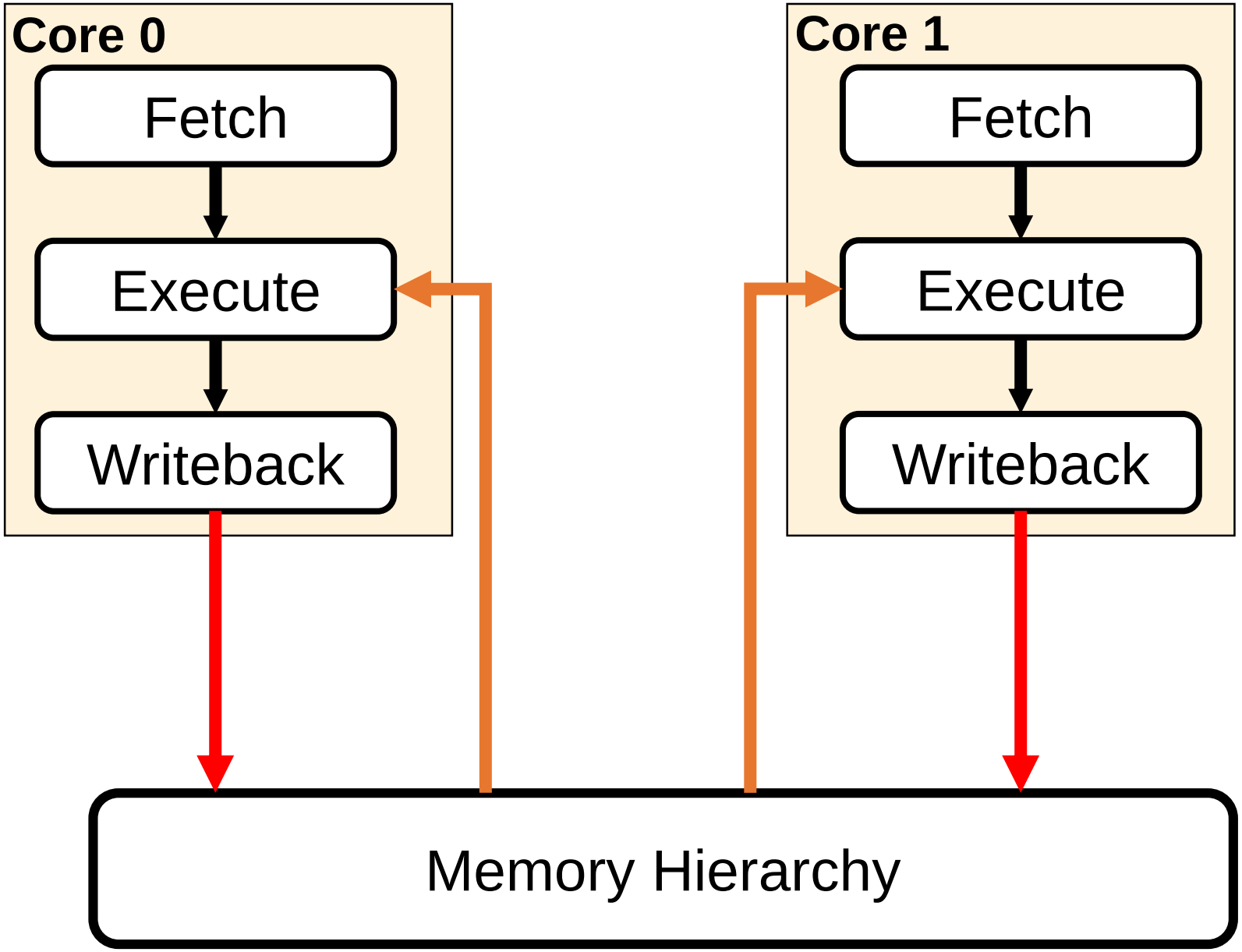
PipeCheck Hands-On Continued:

Extending SC uarch. to TSO

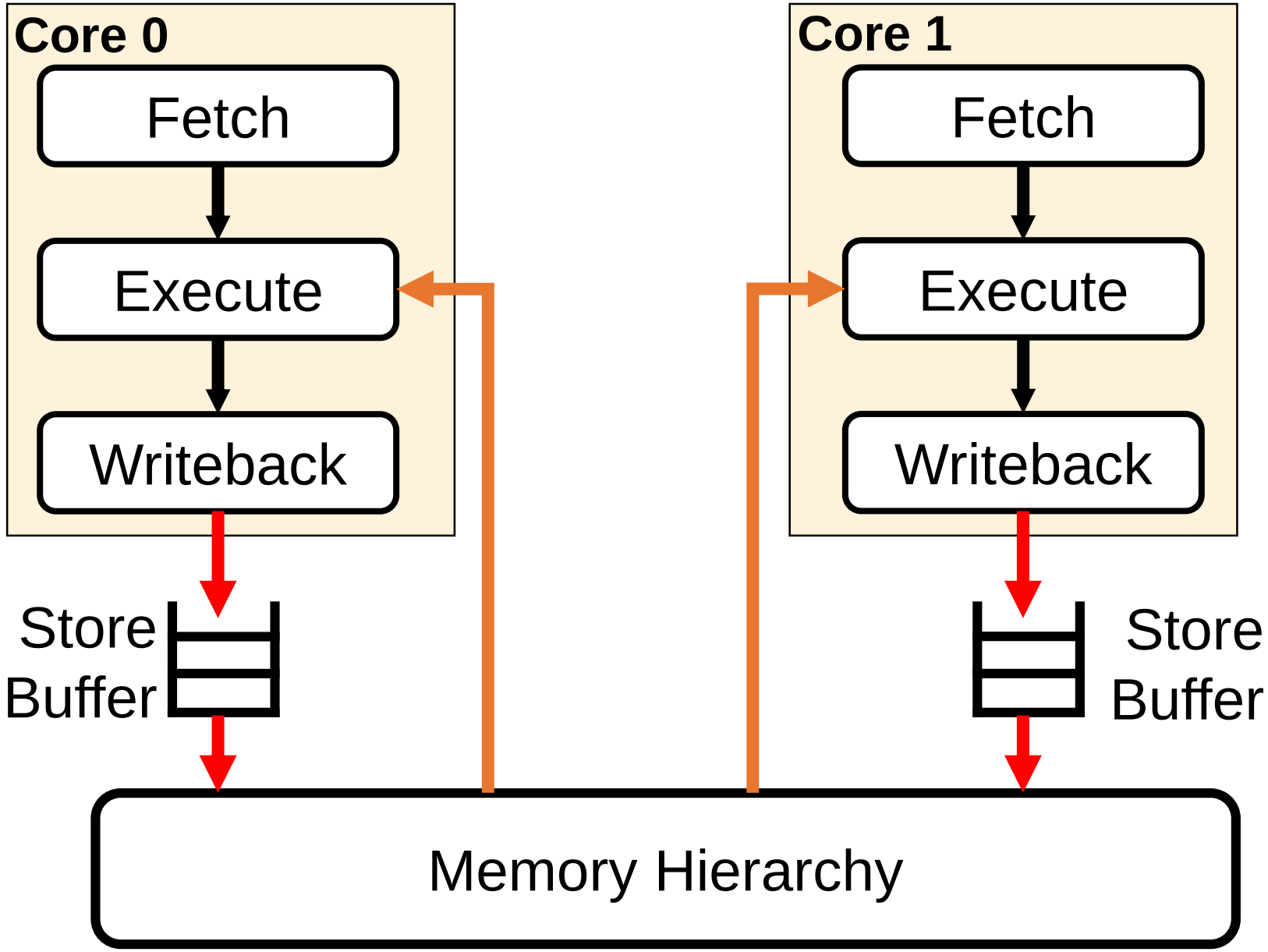
Hands-on: Moving from SC to TSO

- Reads must currently wait for prior writes to reach memory
  - **EnforceWriteOrdering** axiom
  - Low performance!
- Main motivation for TSO: store buffers to hide write latency
  - Allow reads to be ***reordered*** with writes
- Also want to allow reads to bypass value from store buffer (before value made visible to other cores)
  - Known as “read your own write early”
- **How to model this in μSpec?**

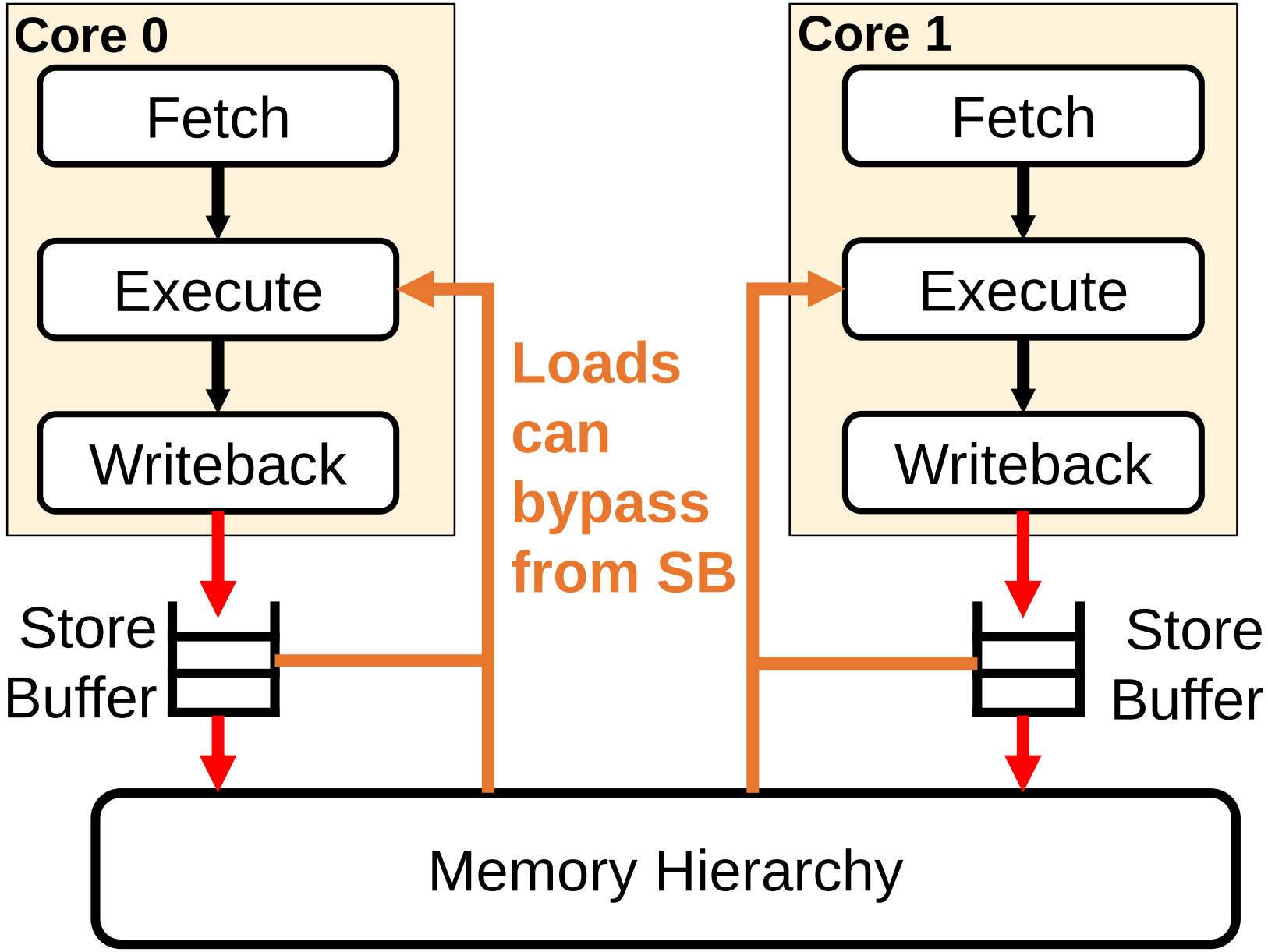
Moving from SC to TSO



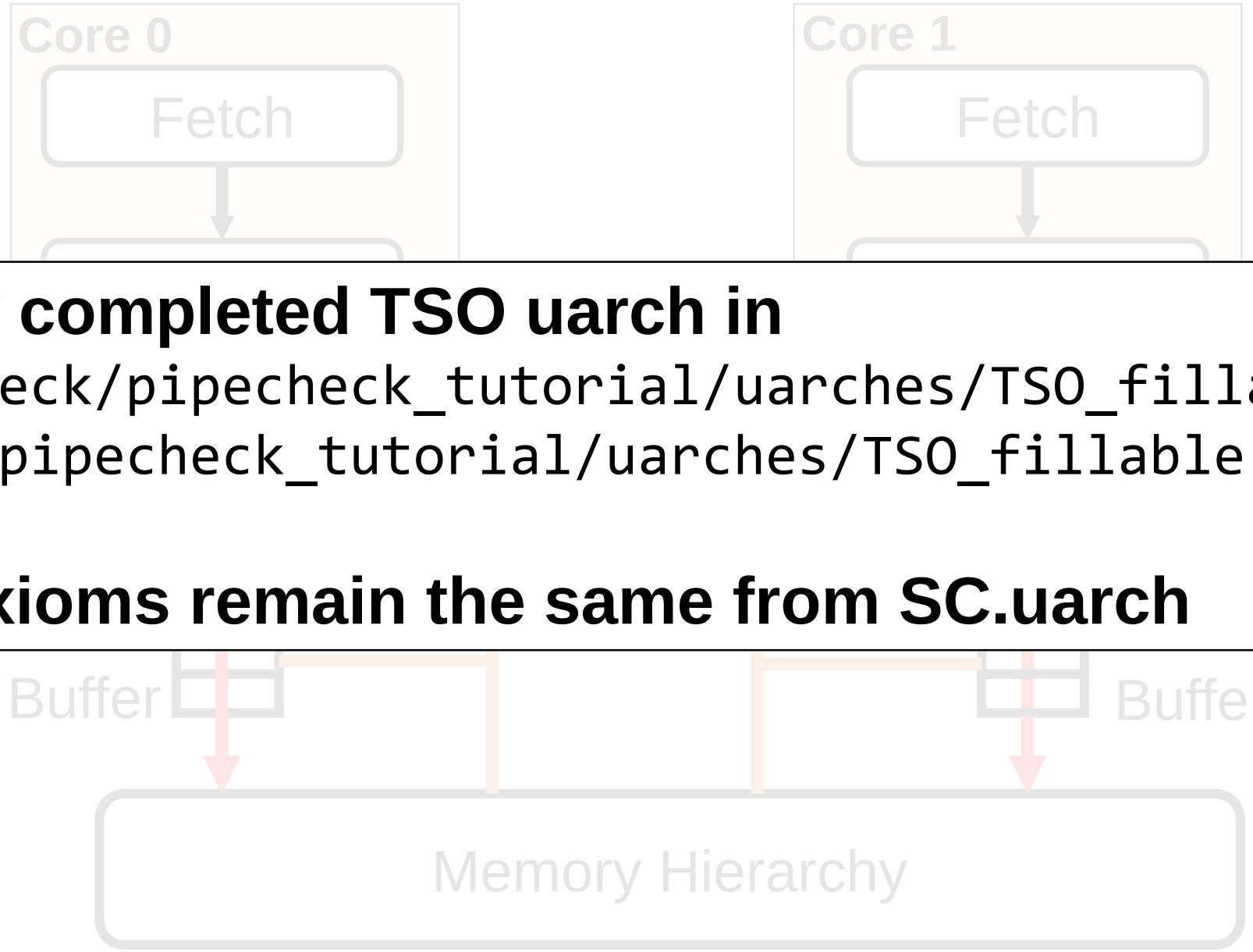
Moving from SC to TSO



Moving from SC to TSO



Moving from SC to TSO



Partially completed TSO uarch in
`/home/check/pipecheck_tutorial/uarches/TSO_fillable.uarch`
`(i.e. ~/pipecheck_tutorial/uarches/TSO_fillable.uarch)`

Some axioms remain the same from SC.uarch

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. **Add store buffer stage**

Add StoreBuffer Stage

TSO_fillable.uarch, line 39

- "StoreBuffer" stage is between Writeback and MemoryHierarchy
- Solution:

```
StageName _ "_____________".
StageName _ "MemoryHierarchy".
```

Add StoreBuffer Stage

TSO_fillable.uarch, line 39

- "StoreBuffer" stage is between Writeback and MemoryHierarchy
- Solution:

```
StageName 3 "StoreBuffer".
StageName 4 "MemoryHierarchy".
```

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. **Make writes go through SB before memory**

Writes Go Through SB

TSO_fillable.uarch, line 55

- Modify **Writes_Path** axiom so stores go WB → SB →MemHier
- Solution:

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute),      "path");
          ((i, Execute), (i, Writeback),    "path");
          ((i, _________), (i, _____________),    "path");
          ((i, _____________), (i, (0, _________________)),
              "path")
          ].
```

Writes Go Through SB

TSO_fillable.uarch, line 55

- Modify **Writes_Path** axiom so stores go WB → SB →MemHier
- Solution:

```
Axiom "Writes_Path":
forall microops "i",
IsAnyWrite i =>
AddEdges [((i, Fetch), (i, Execute),     "path");
          ((i, Execute), (i, Writeback),    "path");
          ((i, Writeback), (i, StoreBuffer),    "path");
          ((i, StoreBuffer), (i, (0, MemoryHierarchy)),
              "path")
          ].
```

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. **Ensure that same-core writes go through SB in order**

Same-Core Writes Go Through SB in order

TSO_fillable.uarch, line 106

- If same-core writes go through WB in order, they should go through SB in order too
- **Hint: Use Writeback_stage_is_in_order axiom as a starting point**
- Solution:

```
Axiom "StoreBuffer_stage_is_in_order":
forall microops "i1",
forall microops "i2",
IsAnyWrite i1 /\ IsAnyWrite i2 /\ _________ i1 i2 =>
EdgeExists ((i1, _________), (i2, _________), "") =>
AddEdge ((i1, ___________), (i2, ___________), "PPO",
"darkgreen").
```

Same-Core Writes Go Through SB in order

TSO_fillable.uarch, line 106

- If same-core writes go through WB in order, they should go through SB in order too
- **Hint: Use Writeback_stage_is_in_order axiom as a starting point**
- Solution:

```
Axiom "StoreBuffer_stage_is_in_order":
forall microops "i1",
forall microops "i2",
IsAnyWrite i1 /\ IsAnyWrite i2 /\ SameCore i1 i2 =>
EdgeExists ((i1, Writeback),  (i2, Writeback), "") =>
AddEdge ((i1, StoreBuffer), (i2, StoreBuffer), "PPO",
"darkgreen").
```

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. **Enforce that write is released from SB only after all prior same-core writes have reached memory**

Same-Core Writes Reach Memory In Order

TSO_fillable.uarch, line 141

- For two same-core writes in program order, first write must reach memory before second can leave store buffer
- **Hint: Axiom should only apply to pairs of writes!**
- Solution:

```
Axiom "EnforceWriteOrdering":
  forall microop "w",
  forall microop "w'",
  (IsAnyWrite w /\ _____________ w' /\ _____________ w w') =>
    AddEdge ((w, (0, _____________)), (w', _____________),
     "one_at_a_time", "green").
```

Same-Core Writes Reach Memory In Order

TSO_fillable.uarch, line 141

- For two same-core writes in program order, first write must reach memory before second can leave store buffer
- **Hint: Axiom should only apply to pairs of writes!**
- Solution:

```
Axiom "EnforceWriteOrdering":
  forall microop "w",
  forall microop "w'",
  (IsAnyWrite w /\ IsAnyWrite w' /\ ProgramOrder w w') =>
    AddEdge ((w, (0, MemoryHierarchy)), (w', StoreBuffer),
     "one_at_a_time", "green").
```

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. **Ensure that if load is reading from memory, that core's store buffer has no entries for address of load**

Only read from Mem if SB has no same addr writes

TSO_fillable.uarch, line 169

- Create a **macro** enforcing that all writes before instr "i" in program order to address of "i" have reached mem before "i" **Execute**s
- Solution:

```
DefineMacro "STBEmpty":
  % Store buffer is empty for the address we want to read.
  forall microop "w", (
    (__________ w /\ ____________________ w i /\
     ___________ w i) =>
      AddEdge ((w, (0, _________________)), (i, _______),
          "STBEmpty", "purple")).
```

Only read from Mem if SB has no same addr writes

TSO_fillable.uarch, line 169

- Create a **macro** enforcing that all writes before instr "i" in program order to address of "i" have reached mem before "i" **Execute**s
- Solution:

```
DefineMacro "STBEmpty":
  % Store buffer is empty for the address we want to read.
  forall microop "w", (
    (IsAnyWrite w /\ SamePhysicalAddress w i /\
     ProgramOrder w i) =>
      AddEdge ((w, (0, MemoryHierarchy)), (i, Execute),
          "STBEmpty", "purple")).
```

Only read from Mem if SB has no same addr writes

`TSO_fillable.uarch, line 226`

- Now expand the macro in Read_Values axiom to ensure that SB has no entries for a load's address if it is reading from memory

Only read from Mem if SB has no same addr writes

TSO_fillable.uarch, line 226

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i => (
% Uncomment the commented lines if you add the (advanced) store buff forwarding.
%  ExpandMacro ______ \/
%  (
       ExpandMacro ________ /\
       (
          ExpandMacro BeforeAllWrites
          \/
          (
            ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
            /\
            ExpandMacro Before_Or_After_Every_SameAddrWrite
          )
       )
%  )
).
```

Only read from Mem if SB has no same addr writes

TSO_fillable.uarch, line 226

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i => (
% Uncomment the commented lines if you add the (advanced) store buff forwarding.
%  ExpandMacro ______ \/
%  (
        ExpandMacro STBEmpty /\
        (
           ExpandMacro BeforeAllWrites
           \/
           (
             ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
             /\
             ExpandMacro Before_Or_After_Every_SameAddrWrite
           )
        )
%  )
).
```

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core's store buffer has no entries for address of load
  6. **(Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores**

Forward Value from SB (Advanced)

`TSO_fillable.uarch, line 269`

- Create a **macro** that checks for a write on the same core to forward from (Execute stage -> Execute stage), and ensures the forwarding occurs **before** the write reaches memory
- Macro must also check that forwarding occurs from the latest write in program order (no intervening writes)
- Solution:

Forward Value from SB (Advanced)

TSO_fillable.uarch, line 269

```
DefineMacro "STBFwd":
  % Forward from the store buffer
  exists microop "w", (
    ___________ w /\
    _________ w i /\
    ____________________ w i /\
    _________ w i /\
    AddEdges [((w, Execute), (i, Execute), "STBFwd", "red");
              ((i, Execute), (w, (0, MemoryHierarchy)), "STBFwd",
                "purple")]) /\
    % Ensure the STB entry is the latest one.
    ~exists microop "w'",
    ___________ w' /\ ____________________ w w' /\
    _____________ w w' /\ _______________ w' i.
```

Forward Value from SB (Advanced)

TSO_fillable.uarch, line 269

```
DefineMacro "STBFwd":
  % Forward from the store buffer
  exists microop "w", (
    IsAnyWrite w /\
    SameCore w i /\
    SamePhysicalAddress w i /\
    SameData w i /\
    AddEdges [((w, Execute), (i, Execute), "STBFwd", "red");
              ((i, Execute), (w, (0, MemoryHierarchy)), "STBFwd",
                "purple")]) /\
    % Ensure the STB entry is the latest one.
    ~exists microop "w'",
    IsAnyWrite w' /\ SamePhysicalAddress w w' /\
    ProgramOrder w w' /\ ProgramOrder w' i.
```

Forward Value from SB

`TSO_fillable.uarch, line 226`

- Expand the macro in the Read_Values axiom so that forwarding from the SB is an *alternative* choice to reading from memory
- Remember to uncomment lines 231-232, and line 243!
- Solution:

TSO_fillable.uarch, line 226

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(
  ExpandMacro ______ \/
  (
    ExpandMacro STBEmpty /\
    (
      ExpandMacro BeforeAllWrites
      \/
      (
        ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
        /\
        ExpandMacro Before_Or_After_Every_SameAddrWrite
      )
    )
  )
).
```

TSO_fillable.uarch, line 226

```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(
  ExpandMacro STBFwd \/
  (
    ExpandMacro STBEmpty /\
    (
      ExpandMacro BeforeAllWrites
      \/
      (
        ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
        /\
        ExpandMacro Before_Or_After_Every_SameAddrWrite
      )
    )
  )
).
```

Hands-on: Moving from SC to TSO

- 7 changes needed to SC.uarch:
  1. Add store buffer stage
  2. Make writes go through SB before memory
  3. Ensure that same-core writes go through SB in order
  4. Enforce that write is released from SB only after all prior same-core writes have reached memory
  5. Ensure that if load is reading from memory, that core's store buffer has no entries for address of load
  6. (Advanced) Allow a core to read value of a write from its store buffer before write is made visible to other cores
  7. **Implement fence operation that flushes all prior writes to memory before any succeeding instructions can perform**

Fence Instruction Orders Write-Read pairs

`TSO_fillable.uarch, line 300`

- Add a fence instruction that flushes all prior writes in program order to memory before the fence's execute stage
- Solution:

Fence Instruction Orders Write-Read pairs

TSO_fillable.uarch, line 300

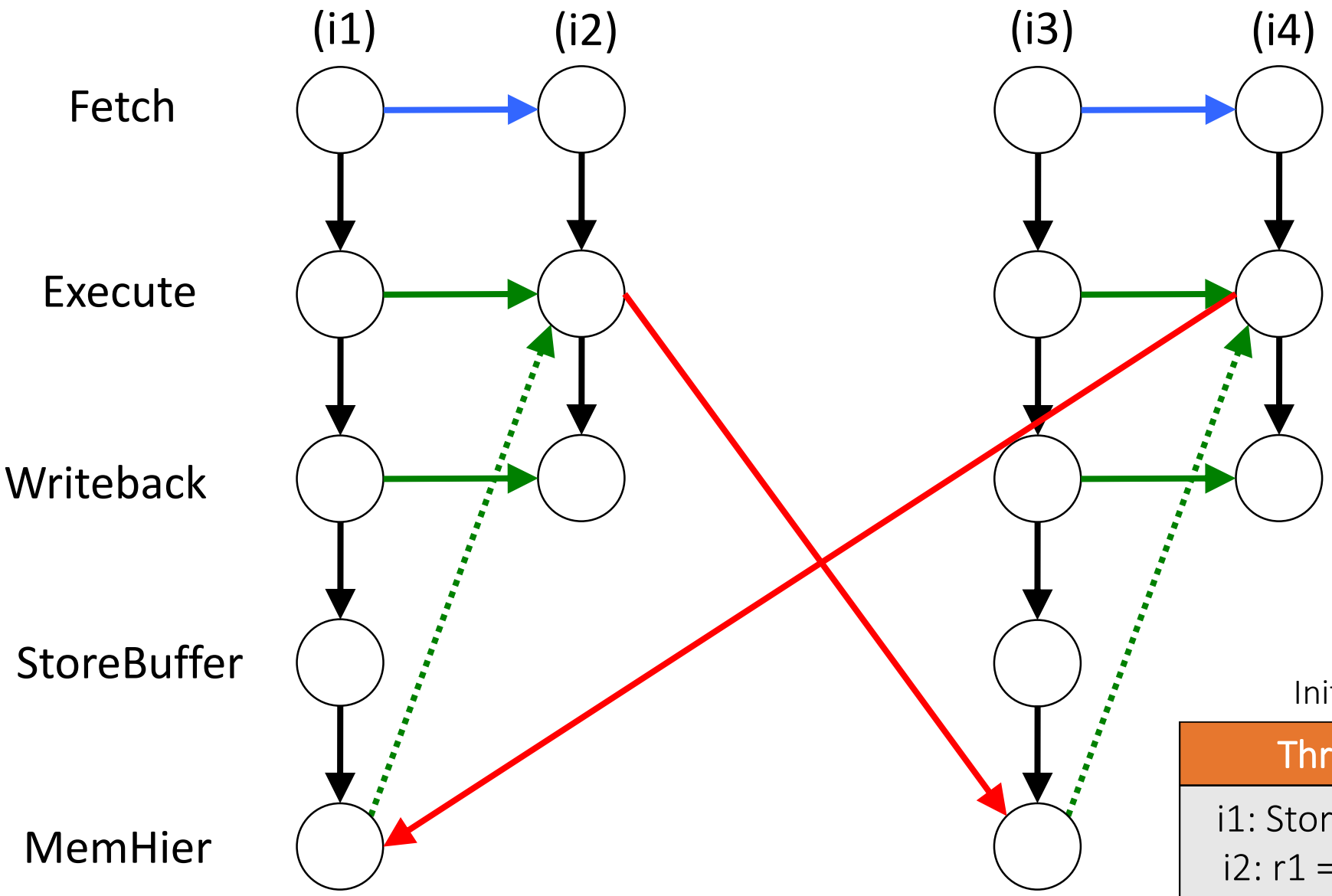
```
Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch),       (f, Execute),     "path");
          ((f, Execute),     (f, Writeback), "path")]
/\
(
  forall microops "w",
    (__________ w /\ _____________ w f) =>
      AddEdge ((w, (0, _________________)), (f, _______),
          "fence", "orange")
).
```

Fence Instruction Orders Write-Read pairs

TSO_fillable.uarch, line 300

```
Axiom "Fence_Ordering":
forall microops "f",
IsAnyFence f =>
AddEdges [((f, Fetch),        (f, Execute),      "path");
          ((f, Execute),      (f, Writeback), "path")]
/\
(
  forall microops "w",
    (IsAnyWrite w /\ ProgramOrder w f) =>
      AddEdge ((w, (0, MemoryHierarchy)), (f, Execute),
          "fence", "orange")
).
```

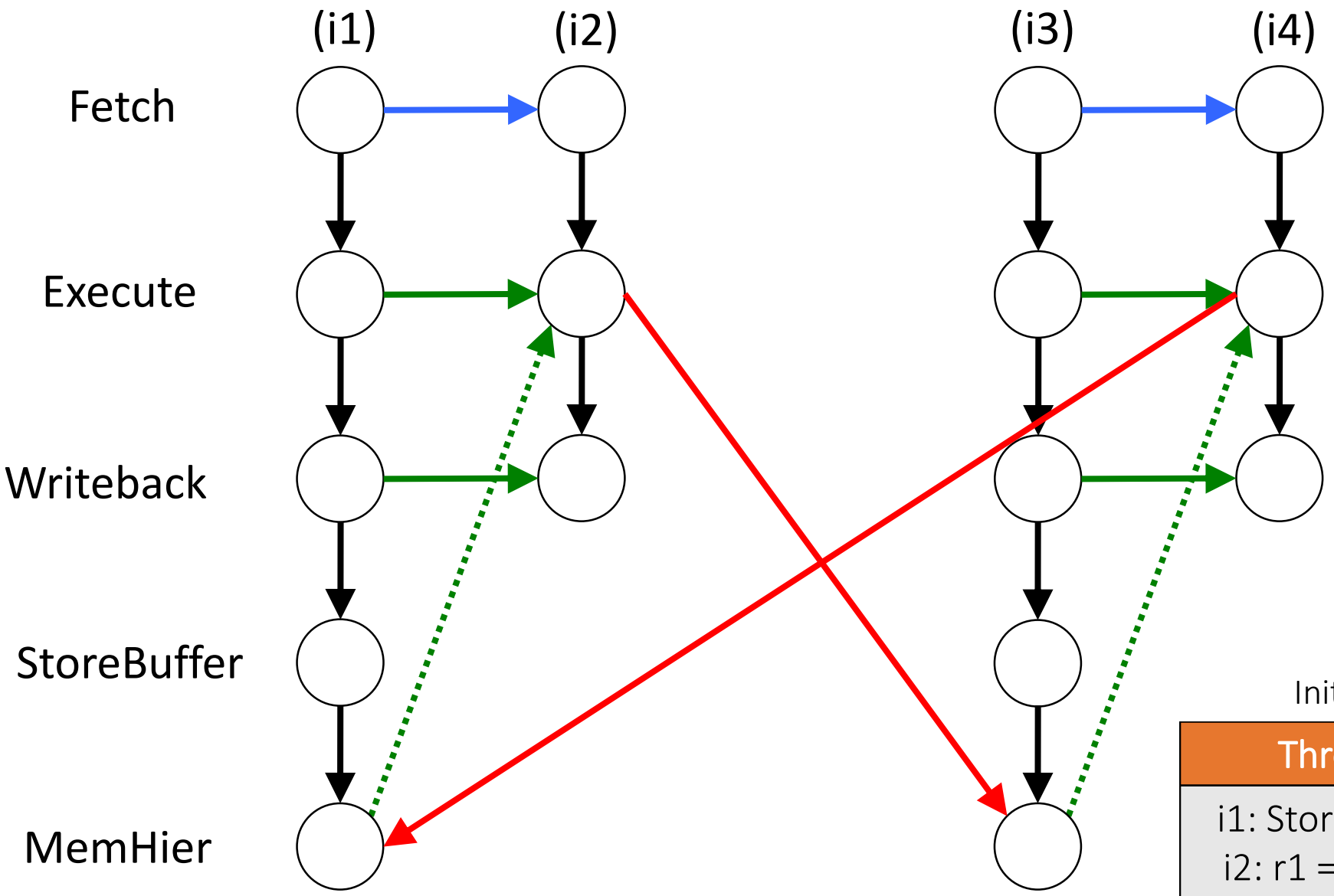
μhb Graph for sb On TSO μarch.



Initially, Mem[x] = Mem[y] = 0

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

μhb Graph for sb On TSO μarch.



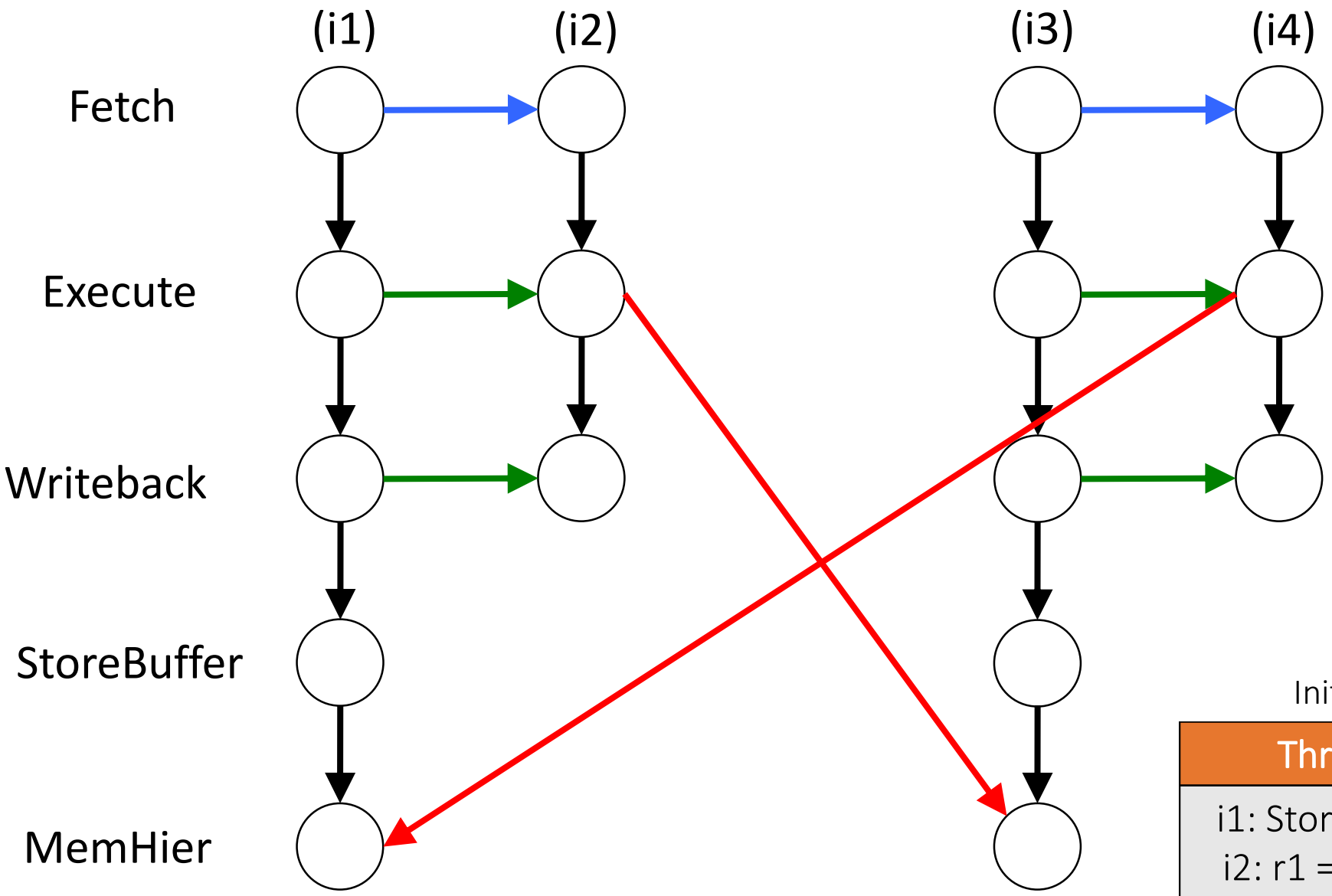
Dotted green edges order writes before later reads in our SC μarch.

These edges are not present in our TSO μarch!

Initially, Mem[x] = Mem[y] = 0

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

μhb Graph for sb On TSO μarch.



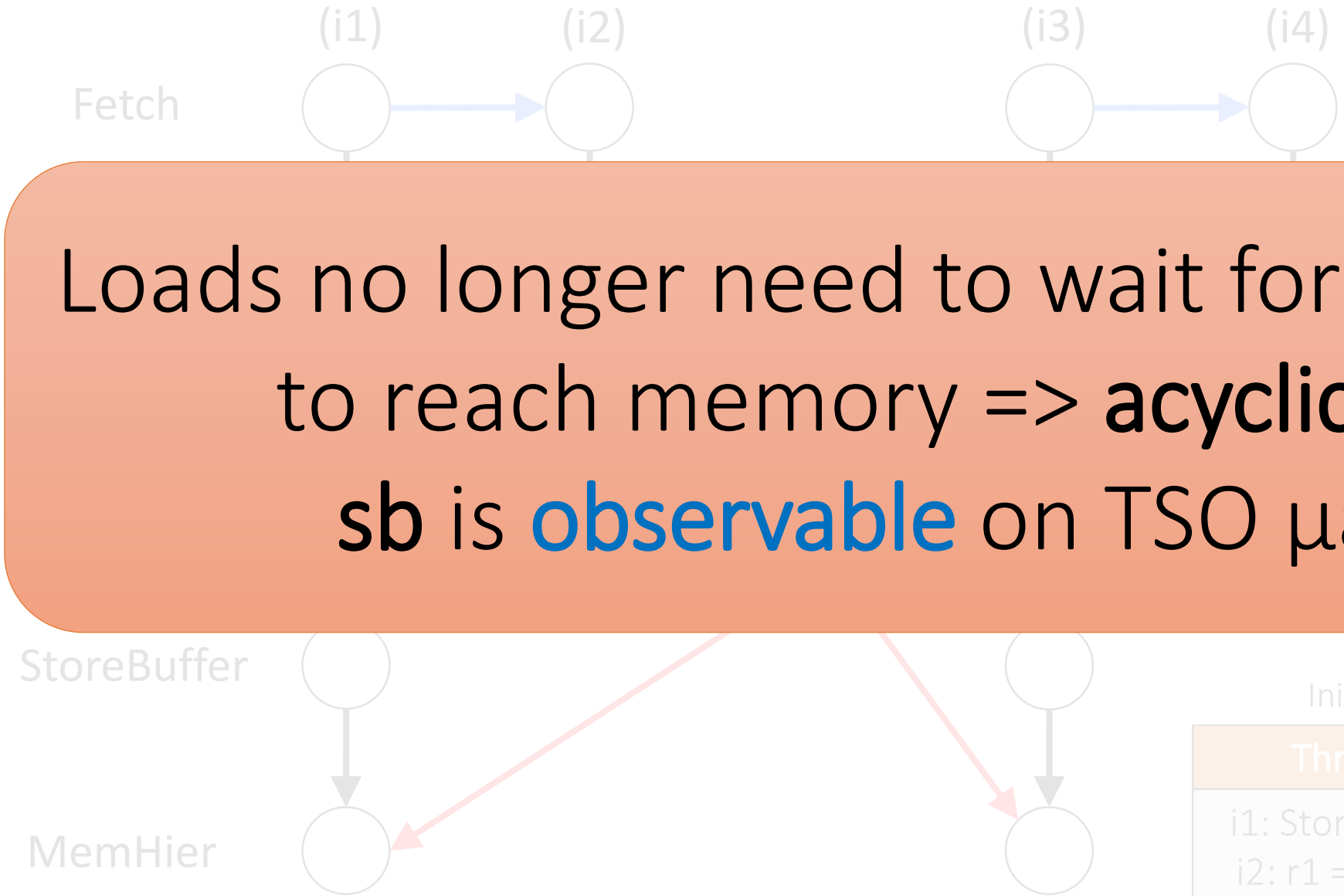
Dotted green edges order writes before later reads in our SC μarch.

These edges are not present in our TSO μarch!

Initially, Mem[x] = Mem[y] = 0

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

μhb Graph for sb On TSO μarch.



Dotted green

Loads no longer need to wait for prior writes to reach memory => **acyclic graph**

sb is observable on TSO μarch!

Initially, Mem[x] = Mem[y] = 0

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

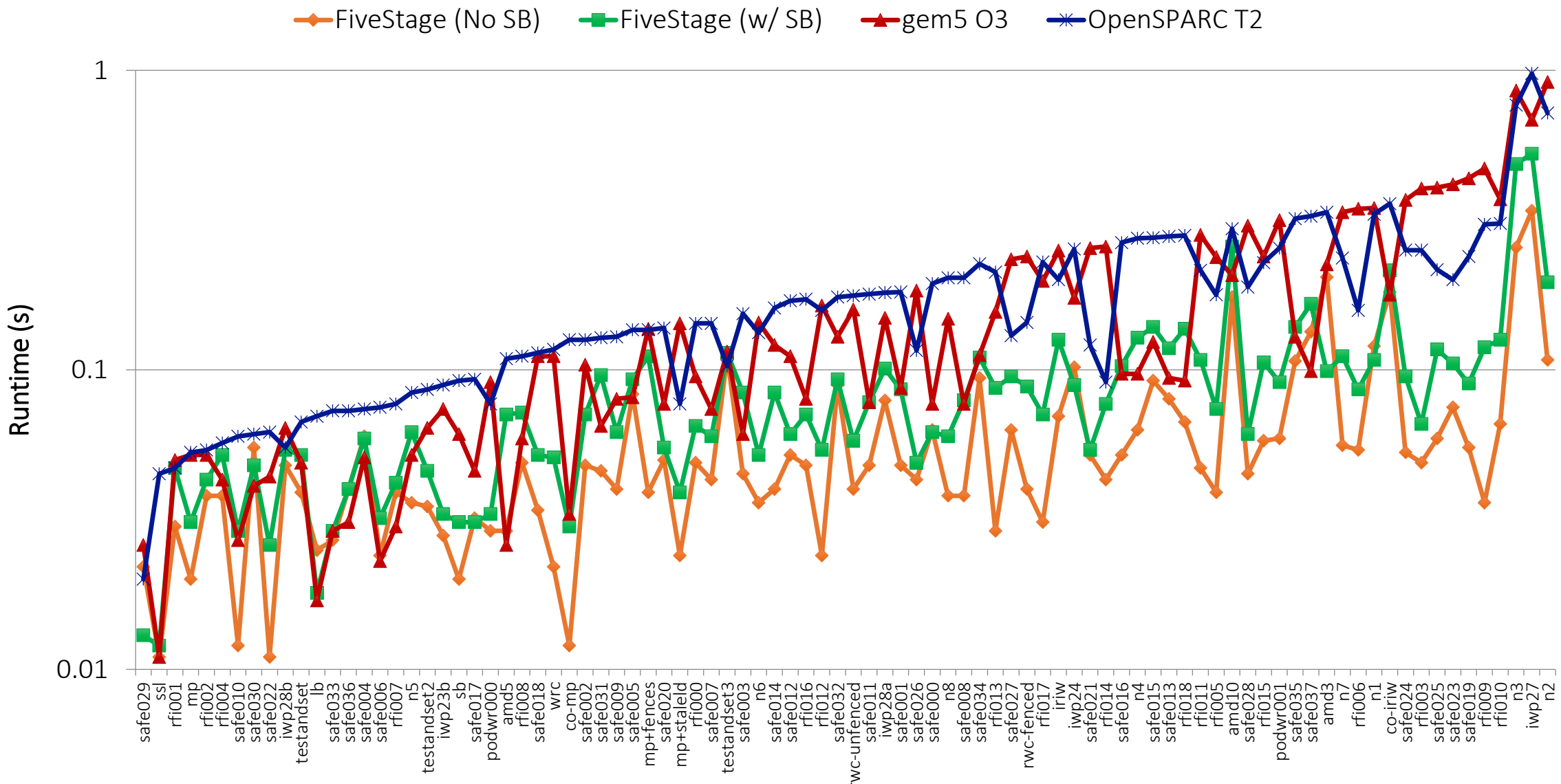
Test your completed TSO uarch!

```
# Assuming you are in ~/pipecheck_tutorial/uarches/
$ check -i ../tests/TSO_tests/sb.test -m TSO_fillable.uarch

# If your uarch is valid, the above will create sb.pdf in your
# current directory (open pdfs from command line with evince)
# To run the solution version of the TSO uarch on this test:
# (Note: this will overwrite the sb.pdf in your current folder)
$ check -i ../tests/TSO_tests/sb.test -m TSO.uarch -d solutions/

# If you get an error (cannot parse uarch, ps2pdf crashes, etc),
# examine your syntax or ask for help.
# If the outcome is not observable ("Strict"), compare the graphs
# generated by the solution uarch to those of your uarch.

# To compare the uarches themselves:
$ diff TSO_fillable.uarch solutions/TSO.uarch
```

Run the entire suite of TSO litmus tests!

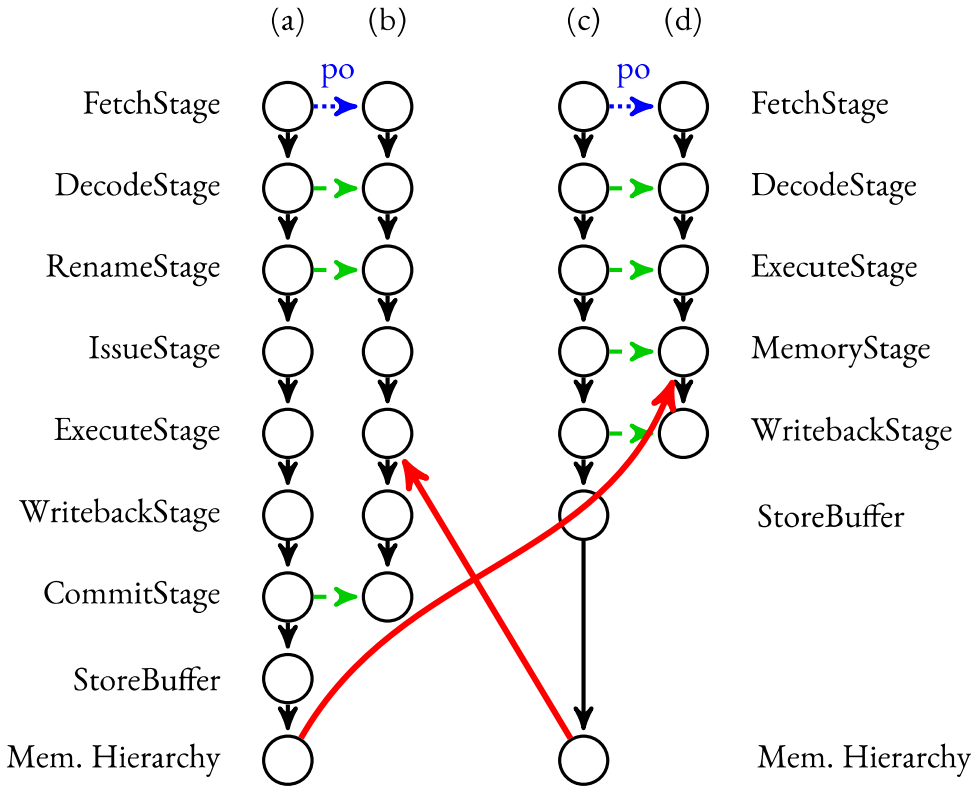
```
# Assuming you are in ~/pipecheck_tutorial/uarches/
$ run_tests -v 2 -t ../tests/TSO_tests/ -m TSO_fillable.uarch

# The above will generate *.gv files in ~/pipecheck_tutorial/out/
# for all TSO tests, and output overall statistics at the end. If
# the count for "Buggy" is non-zero, your uarch is faulty. Look for
the tests that output "BUG" to find out which tests fail.

# You can use gen_graph to convert gv files into PDFs:
$ gen_graph -i <test_gv_file>

# Compare your uarch with the solution TSO uarch using diff to find
# discrepancies:
$ diff TSO_fillable.uarch solutions/TSO.uarch
```

PipeCheck Verification Time



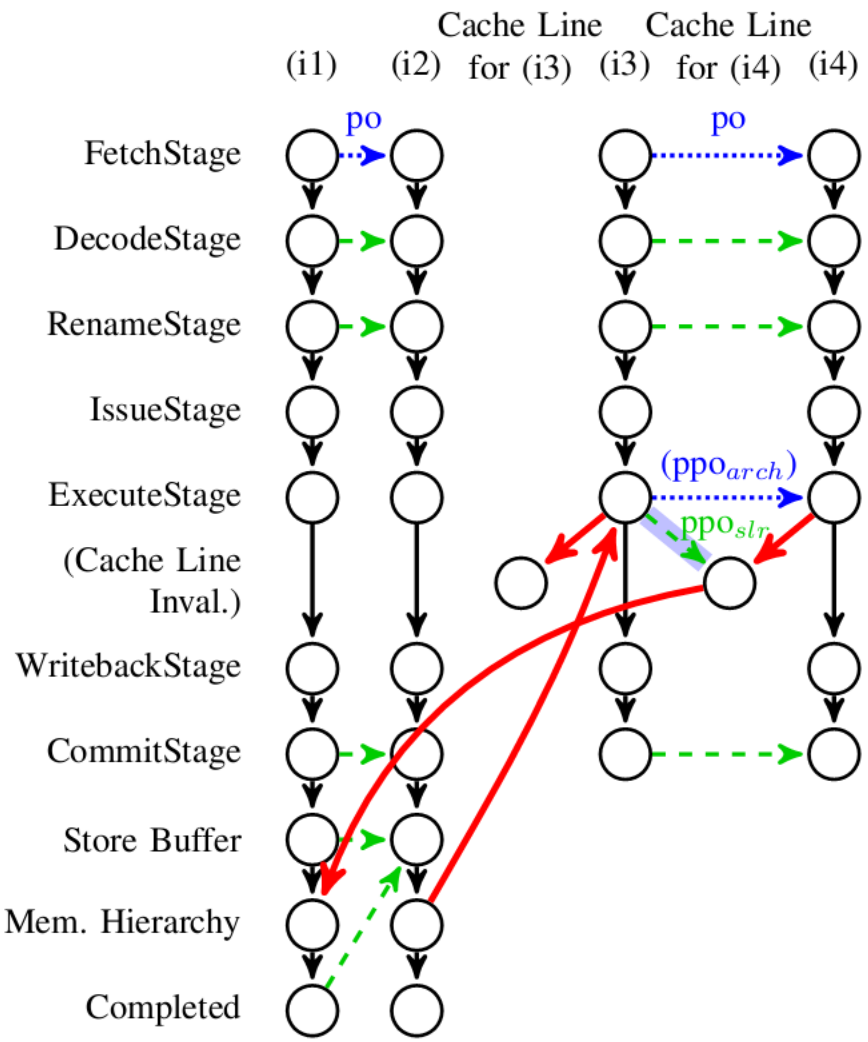
Covered the basics of what PipeCheck can do…

- But there's more!
- PipeCheck can handle heterogeneous pipelines:



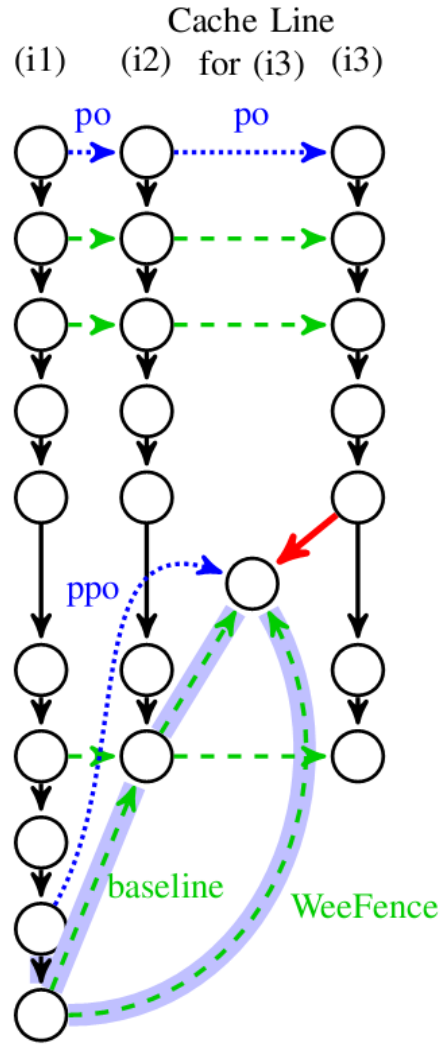
Covered the basics of what PipeCheck can do…

- …and microarchitectural optimizations…



Left: Speculative Load Reordering

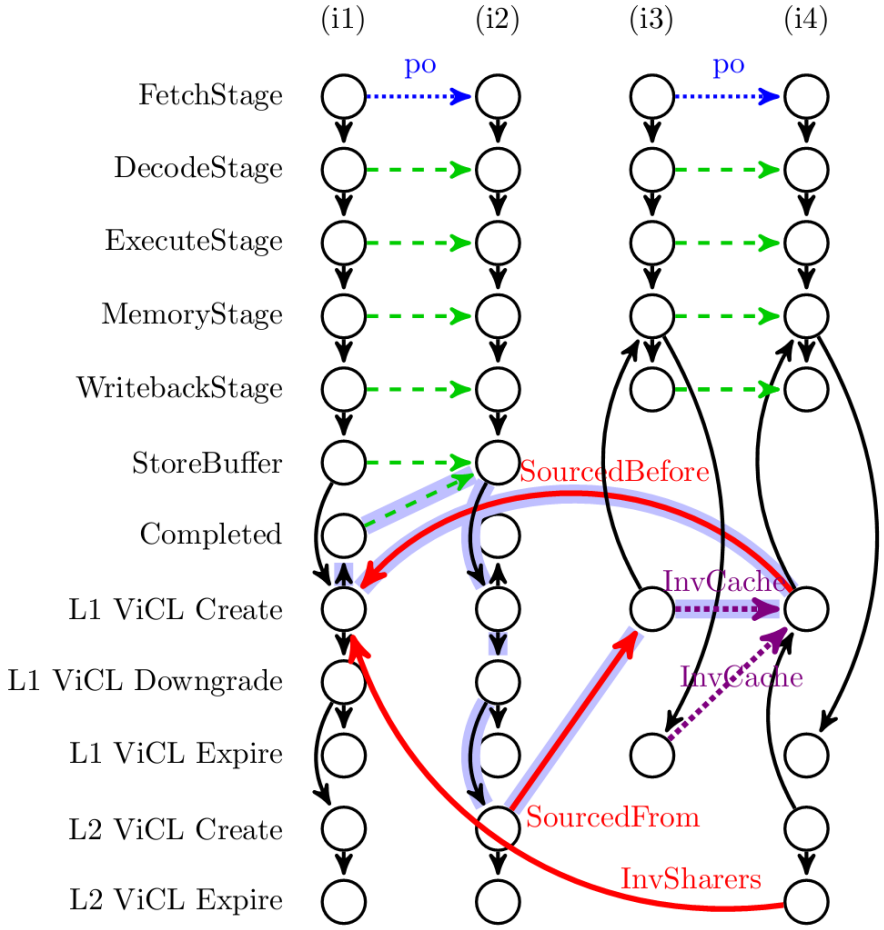
Right: Speculative Fence Retirement



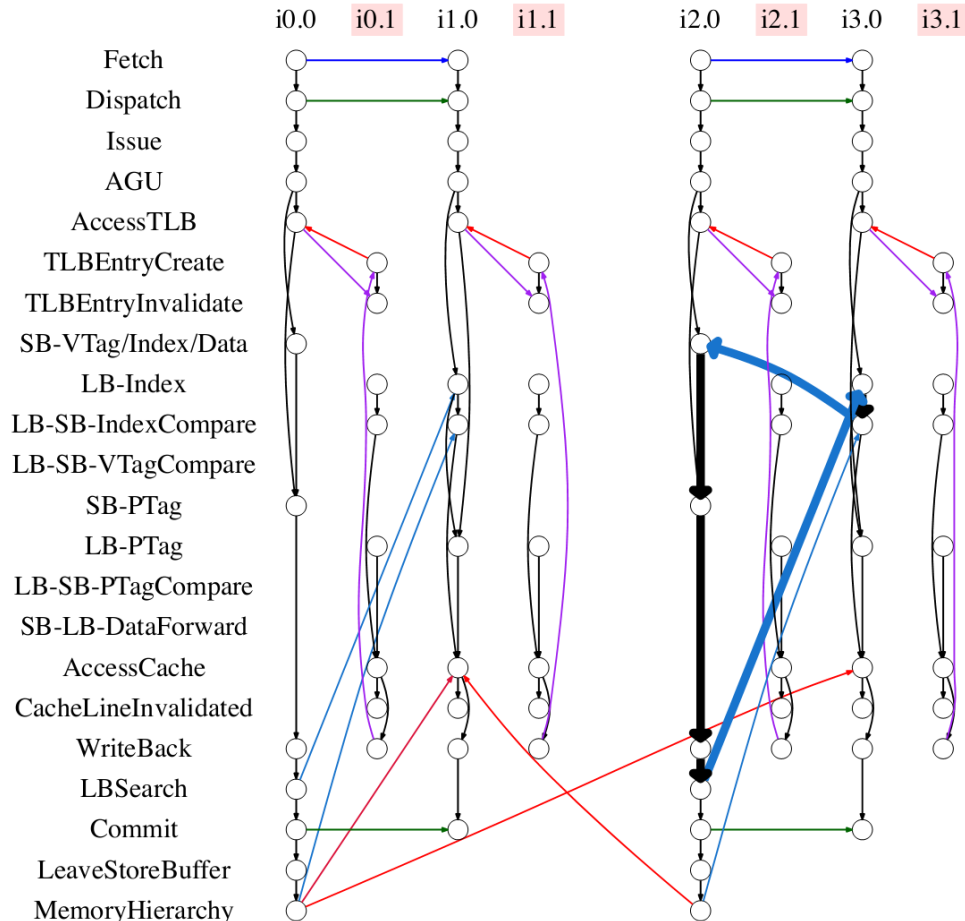
Covered the basics of what PipeCheck can do…

- …and the methodology is extensible to other types of orderings!

CCICheck: Coherence orderings that affect consistency



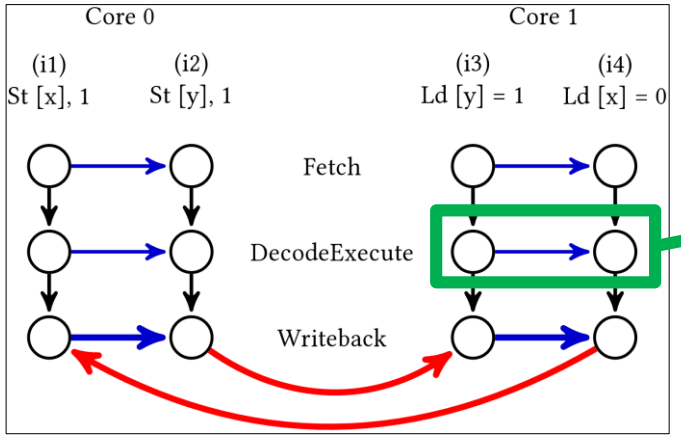
COATCheck: Addr Translation/Virtual Memory orderings that affect consistency



Does the μspec model match hardware?

- **RTLCheck:** Validate that hardware supports μspec axioms!

Axiomatic
Microarch. Verification



Abstract nodes and happens-before edges

Temporal
RTL Verification (SVA, etc)

Concrete signals and clock cycles

PipeCheck Summary

- Fast, automated verification
- Check hardware implementation against ISA spec
- Decompose HW verification into smaller per-axiom sub-problems
  - Each axiom can then be each validated w.r.t RTL independently
- Open-Sourced:

https://github.com/daniellustig/coatcheck

Repo from this tutorial:

https://github.com/ymanerka/pipecheck_tutorial

Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, …

TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA

Caroline Trippel, Yatin A. Manerkar, Daniel Lustig*,
Michael Pellauer*, Margaret Martonosi

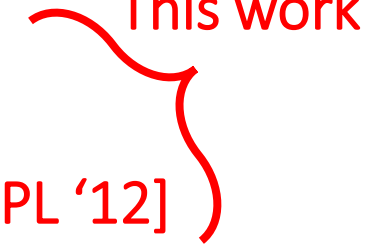
Princeton University *NVIDIA

ASPLOS 2017

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

Why is TriCheck Necessary?

- Memory model bugs are real and problematic!
  - ARM Read-after-Read Hazard [Alglave et al. TOPLAS14]
  - RISC-V ISA draft specification was incompatible with C11
  - C11→POWER/ARMv7 "trailing-sync" compiler mapping **[Batty et al. POPL '12]**
  - C11→POWER/ARMv7 "leading-sync" compiler mapping [Lahav et al. PLDI17]

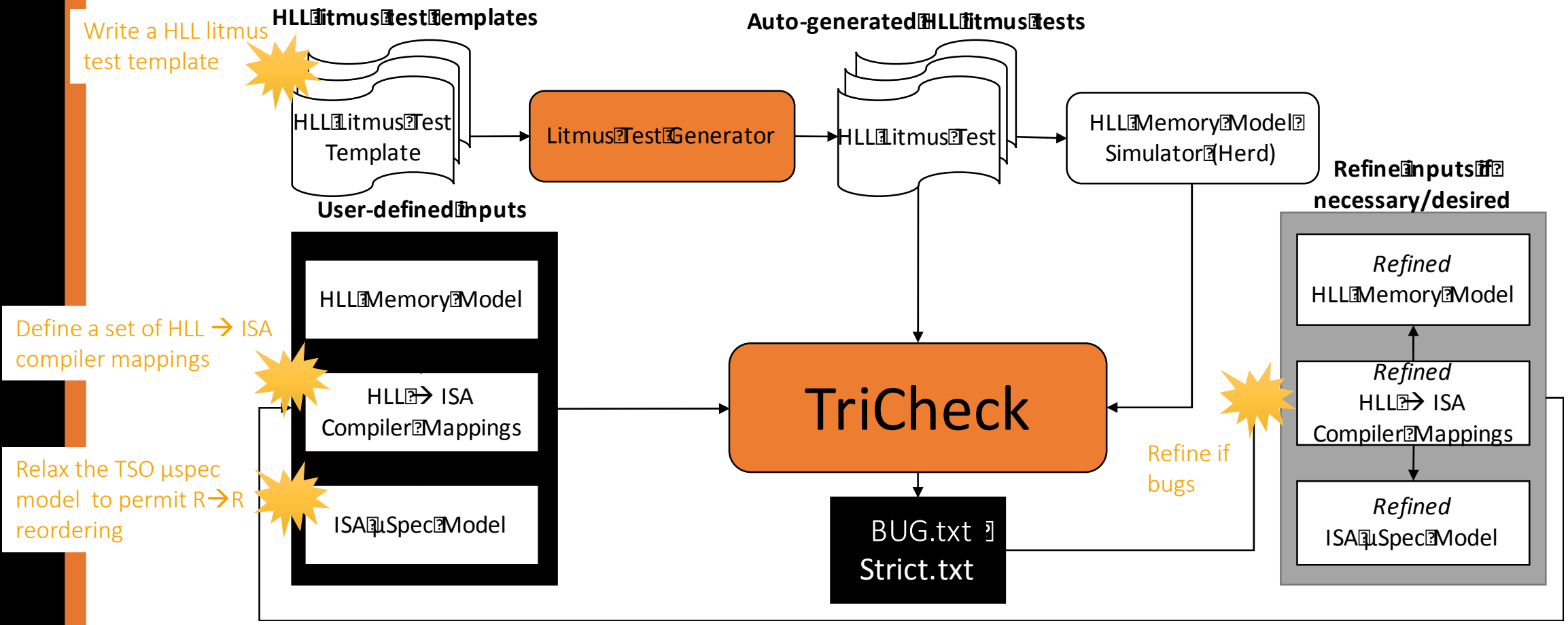


- ISAs are an important and still-fluid design point!
  - Often, ISAs designed in light of desired HW optimizations
  - ISA places some constraints on hardware and some on compiler
  - Many industry memory models are still evolving: C11, ARMv7 vs. ARMv8
  - New ISAs are designed, e.g., RISC-V CPUs, specialized accelerators
- Correctness requires cooperation of the whole stack

TriCheck Key Ideas

- **First tool capable of full stack memory model verification**
  - Any layer can introduce real bugs
- **Litmus Tests + Auto-generators**
  - Comprehensive families of tests across HLL ordering options, compiler mapping variations, ISA options
- **Happens-before, graph-based analysis**
  - Nodes are memory accesses & ordering primitives
  - Edges are event orders discerned via memory model relations
- **Efficient top-to-bottom analysis: Runtime in seconds or minutes**
  - Fast enough to find real bugs; Interactive design process

TriCheck Overview



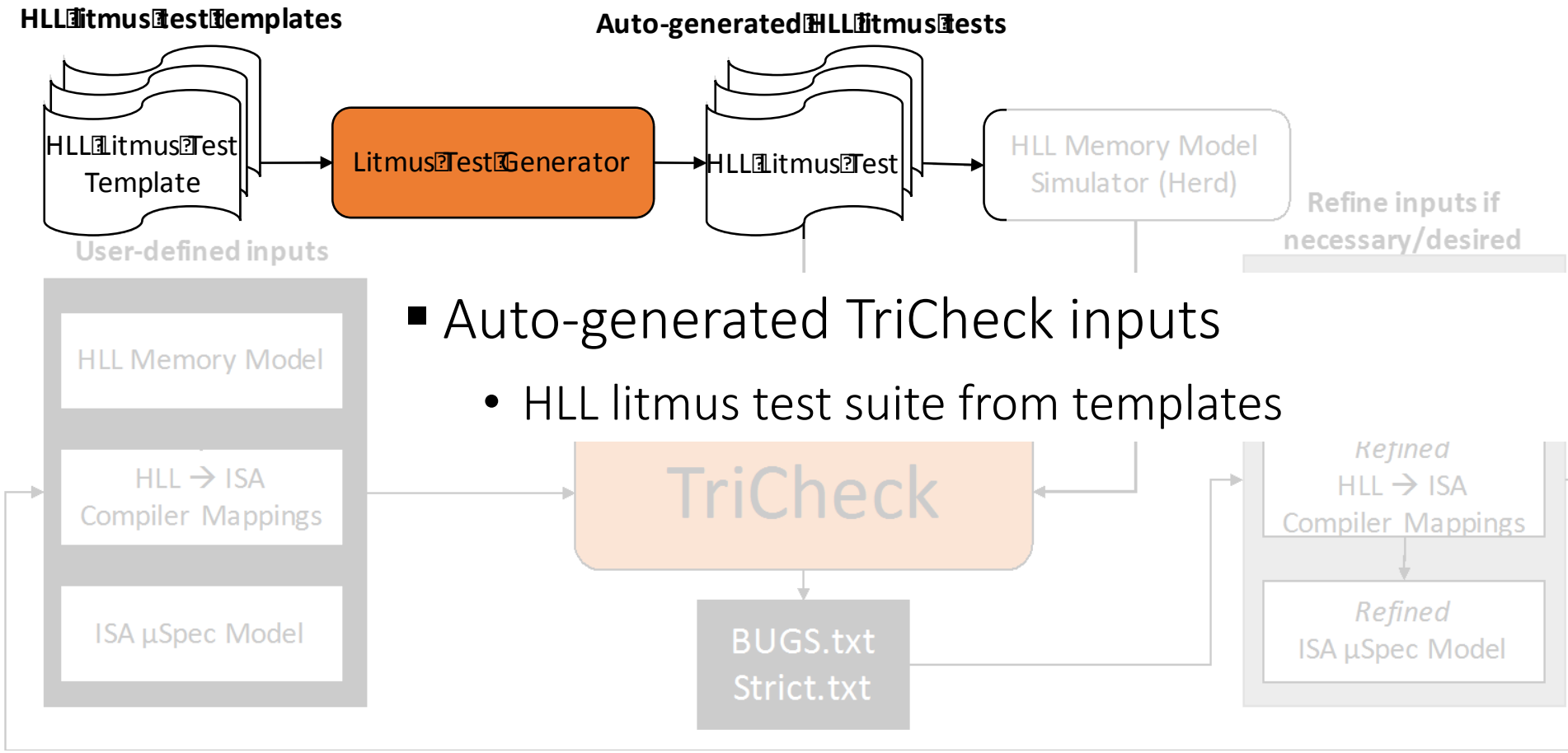
Outline

- TriCheck Introduction
- Auto-generating HLL litmus tests
- User-defined TriCheck inputs
- Iterative ISA design example
- Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
- Ongoing Work & Conclusions

NOTE: Before running TriCheck, define the $TRICHECK_HOME environment variable and install the parallel utility:

```
sudo apt-get update
export TRICHECK_HOME=/home/check/TriCheck
sudo apt-get install parallel
```

Auto-generating HLL litmus tests



- Auto-generated TriCheck inputs
  - HLL litmus test suite from templates

* Omitting *consume*

Litmus test templates

- HLL is generally meant to compile/map to a variety of ISAs
  - For a given litmus test, we want to evaluate all possible HLL-level formulations and ordering options
  - Translates to evaluating a variety of compiler mapping and ISA options
- HLL litmus tests with placeholders for HLL-specific memory model ordering primitives
- E.g., C11 features the *atomic* type and allows programmers to place ordering constraints on memory accesses to *atomic* variables
  - Stores to atomic variables can be specified as *relaxed, release,* or *seq_cst*
  - Loads of atomic variables can be specified as *relaxed, acquire*, or seq_cst*
- Litmus test templates path: $TRICHECK_HOME/tests/templates

$TRICHECK_HOME/tests/templates/mp.litmus

```
C <TEST>

{
[x] = 0;
[y] = 0;
}

P0 (atomic_int* y, atomic_int* x) {
  atomic_store_explicit(x,1,memory_order_<ORDER_STORE>);
  atomic_store_explicit(y,1,memory_order_<ORDER_STORE>);
}

P1 (atomic_int* y, atomic_int* x) {
  int r0 = atomic_load_explicit(y,memory_order_<ORDER_LOAD>);
  int r1 = atomic_load_explicit(x,memory_order_<ORDER_LOAD>);
}

exists (0:r0=1 /\ 1:r1=0)
```

Processor/Core ID

Message Passing (MP)	
T0	T1
W x ← 1	R y ← 1
W y ← 1	R x ← 0
Non-SC Outcome Forbidden	

Exercise: $TRICHECK_HOME/tests/templates/sb.litmus

```
C <TEST>

{
[x] = 0;
[y] = 0;
}

P0 (atomic_int* y, atomic_int* x) {
 // store to x
 int r0 = // load of y
}

P1 (atomic_int* y, atomic_int* x) {
 // store to y
 int r1 = // load of x
}
exists (              )
```

Store Buffering (SB)	
P0	P1
W x ← 1	W y ← 1
R y ← 0	R x ← 0
Non-SC Outcome Permitted	

Solution: $TRICHECK_HOME/tests/templates/sb.litmus

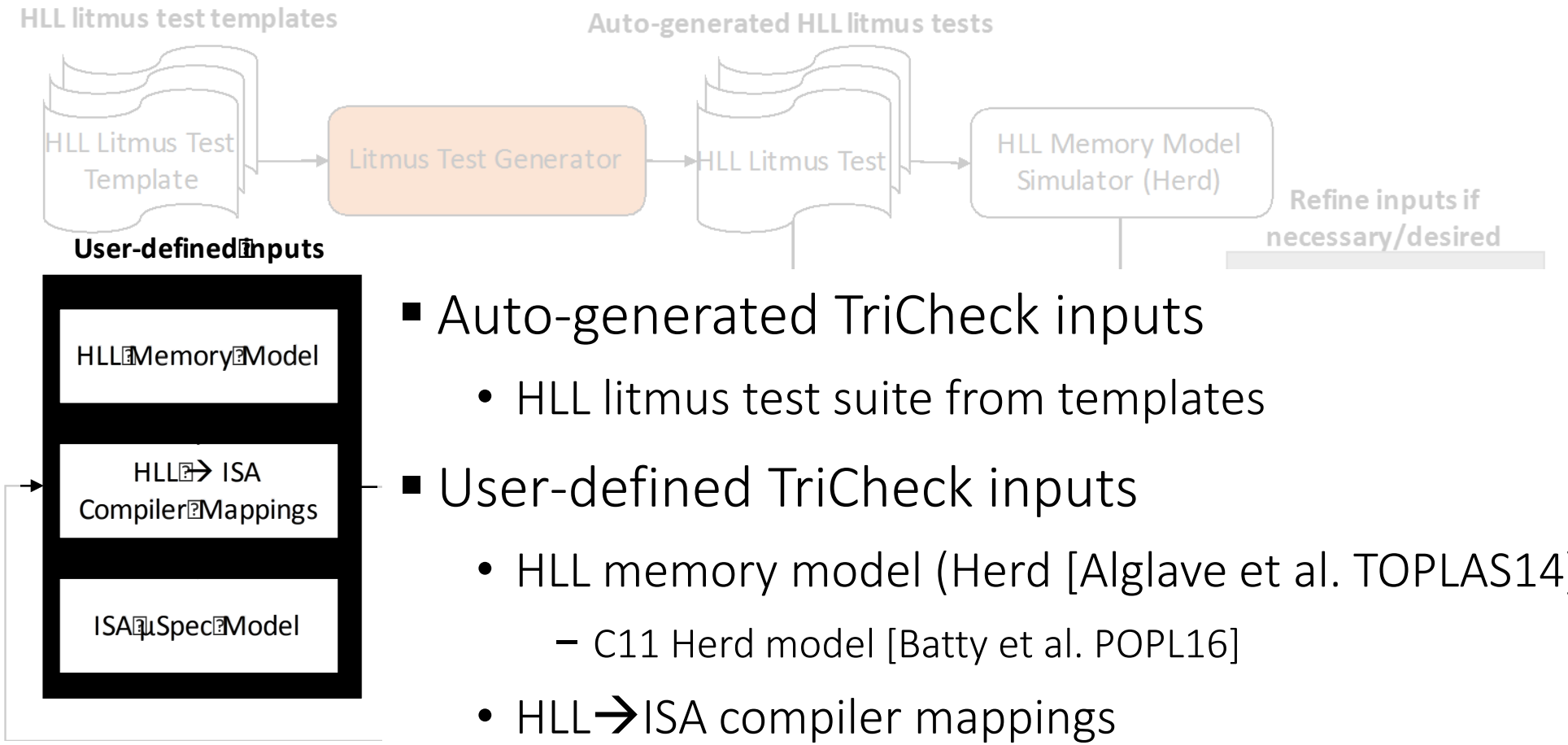
```
C <TEST>

{
[x] = 0;
[y] = 0;
}

P0 (atomic_int* y, atomic_int* x) {
  atomic_store_explicit(x,1,memory_order_<ORDER_STORE>);
  int r0 = atomic_load_explicit(y,memory_order_<ORDER_LOAD>);
}

P1 (atomic_int* y, atomic_int* x) {
  atomic_store_explicit(y,1,memory_order_<ORDER_STORE>);
  int r1 = atomic_load_explicit(x,memory_order_<ORDER_LOAD>);
}

exists (0:r0=0 /\ 1:r1=0)
```

Store Buffering (SB)	
P0	P1
W x ← 1	W y ← 1
R y ← 0	R x ← 0
Non-SC Outcome Permitted	

Outline

- TriCheck Introduction
- Auto-generating HLL litmus tests
- User-defined TriCheck inputs
- Iterative ISA design example
- Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
- Ongoing Work & Conclusions

User-defined Inputs



- Auto-generated TriCheck inputs
  - HLL litmus test suite from templates
- User-defined TriCheck inputs
  - HLL memory model (Herd [Alglave et al. TOPLAS14])
    - C11 Herd model [Batty et al. POPL16]
  - HLL→ISA compiler mappings
  - Hardware model (*μspec* DSL)

User-defined input #1: HLL memory model

- For this tutorial, we will use the C11 HLL memory model, written in herd syntax from [Batty et al., POPL16]
- C11 herd model path: $TRICHECK_HOME/util/herd/c11_partialSC.cat

User-defined inputs #2 & #3: ISA

- ISA is a contract between hardware and software
- Sliding lever between what is required by compiler and what is required by microarchitecture
- TriCheck represents ISA as an input through:
  - Compiler mappings
  - Hardware model

User-defined input #3: Hardware model

- Hardware model so we know primitives to use in compiler mappings
- Default TriCheck uarches path: $TRICHECK_HOME/uarches

- **Exercise:** open $TRICHECK_HOME/uarches /TSO-RR.uarch
  - Relax Ld-Ld order
  - Enforce Ld-Ld order only for dependent operations
    - Address dependencies – affect Ld-Ld, Ld-St
    - Data dependencies – affect Ld-St
    - Control dependencies – affect Ld-Ld, Ld-St

1. Modify Execute_stage_is_in_order axiom

- Modify axiom to permit Ld-Ld reordering:

  “Execute stage is in order for all pairs of operations *except* two reads”

```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ ~(_____________ /\ _____________) /\
EdgeExists ((i1, Fetch),  (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "darkgreen").
```

TSO-RR.uarch, line 57

1. Modify Execute_stage_is_in_order axiom

- Modify axiom to permit Ld-Ld reordering:

  “Execute stage is in order for all pairs of operations *except* two reads”

```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ ~(IsAnyRead i1 /\ IsAnyRead i2) /\
EdgeExists ((i1, Fetch),  (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO", "darkgreen").
```

TSO-RR.uarch, line 57

2. Enforce dependency order by default

- Relaxing Ld-Ld order requires axioms for address (addr) and control (ctrlisb) dependencies
  - Make use of HasDependency <addr|data|ctrl|ctrlisb> <i1> <i2> predicate

"If two reads are related by a dependency of type <addr|ctrlisb>, they must execute in order"

```
Axiom "Addr_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ _____________ /\ _____________ /\ HasDependency addr i1 i2 =>
  AddEdge ((i1, _______), (i2, _______), "addr_rr_dependency").

Axiom "CtrlIsb_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ _____________ /\ _____________ /\ HasDependency ctrlisb i1 i2 =>
  AddEdge ((i1, _______), (i2, _______), "ctrlisb").
```

TSO-RR.uarch, line 196

2. Enforce dependency order by default

- Relaxing Ld-Ld order requires axioms for address (addr) and control (ctrlisb) dependencies
  - Make use of HasDependency <addr|data|ctrl|ctrlisb> <i1> <i2> predicate

"If two reads are related by a dependency of type <addr|ctrlisb>, they must execute in order"

```
Axiom "Addr_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ IsAnyRead i1 /\ IsAnyRead i2 /\ HasDependency addr i1 i2 =>
  AddEdge ((i1, Execute), (i2, Execute), "addr_rr_dependency").

Axiom "CtrlIsb_Read_Read_Dependencies":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\ IsAnyRead i1 /\ IsAnyRead i2 /\ HasDependency ctrlisb i1 i2 =>
  AddEdge ((i1, Execute), (i2, Execute), "ctrlisb").
```

TSO-RR.uarch, line 196

User-defined input #2: HLL → ISA compiler mappings

- Compiler mappings have been proven correct for C11 to x86-TSO

https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html	
C/C++11 Operation	X86-TSO implementation
Load Relaxed:	MOV
Load Acquire:	MOV
Load Seq_Cst:	MOV
Store Relaxed:	MOV
Store Release:	MOV
Store Seq Cst:	MOV, MFENCE

- Path to compiler mappings file: $TRICHECK_HOME/util/compile.txt

User-defined input #2: HLL → ISA compiler mappings

- This is how we would specify the C11 to TSO.uarch compiler mappings in compile.txt:

C11/C++11 op	prefix;prefix	suffix;suffix
Read relaxed	NA	NA
Write relaxed	NA	NA
Read acquire	NA	NA
Write release	NA	NA
Read seq_cst	NA	NA
Write seq_cst	NA	MMFENCE

User-defined input #2: HLL → ISA compiler mappings

- **Exercise:** Modify these mappings for our new TSO-RR.uarch that relaxes Read→Read ordering.
  - **Hint:** Load Acquire and Load Seq_Cst require Read→Read order.

C11/C++11 op	prefix;prefix	suffix;suffix
Read relaxed	NA	NA
Write relaxed	NA	NA
Read acquire	NA	NA
Write release	NA	NA
Read seq_cst	NA	NA
Write seq_cst	NA	MMFENCE

User-defined input #2: HLL → ISA compiler mappings

- **Solution:** Modify these mappings for our new TSO-RR.uarch that relaxes Read→Read ordering.
  - **Hint:** Load Acquire and Load Seq_Cst require Read→Read order.

C11/C++11 op	prefix;prefix	suffix;suffix
Read relaxed	NA	NA
Write relaxed	NA	NA
Read acquire	NA	MMFENCE
Write release	NA	NA
Read seq_cst	NA	MMFENCE
Write seq_cst	NA	MMFENCE

Outline

- TriCheck Introduction
- Auto-generating HLL litmus tests
- User-defined TriCheck inputs
- Iterative ISA design example
- Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
- Ongoing Work & Conclusions

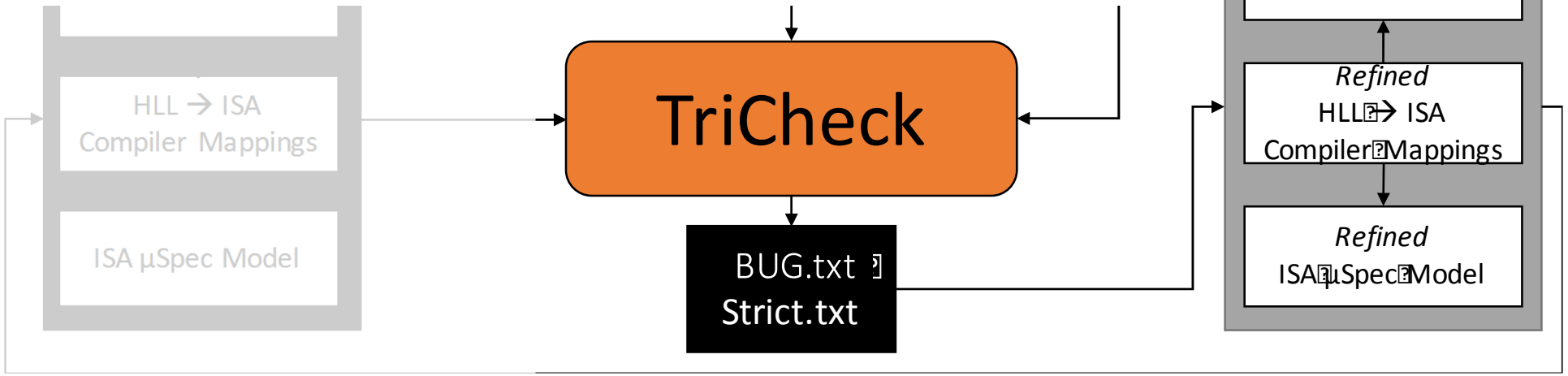
Run TriCheck On Inputs

- cd $TRICHECK_HOME/util
- ./release-generate-tests.py --all --fences
- ./release-run-all.py --pipecheck=/home/check/pipecheck_tutorial/src

Path to litmus test generator: $TRICHECK_HOME/util/release-generate-tests.py
Path to TriCheck: $TRICHECK_HOME/util/release-run-all.py

User-defined Inputs

- Each iteration: bugs analyzed to identify cause
  - Compiler bug, hardware implementation bug, ISA bug
  - Blame may be debated
  - Blame != Fix



Create BUG.txt and Strict.txt

- cd $TRICHECK_HOME/util
- ./release-parse-results.py
- cat $TRICHECK_HOME/util/results/TSO-RR.uarch/BUG.txt

Path to TriCheck output parser: $TRICHECK_HOME/util/release-parse-results.py

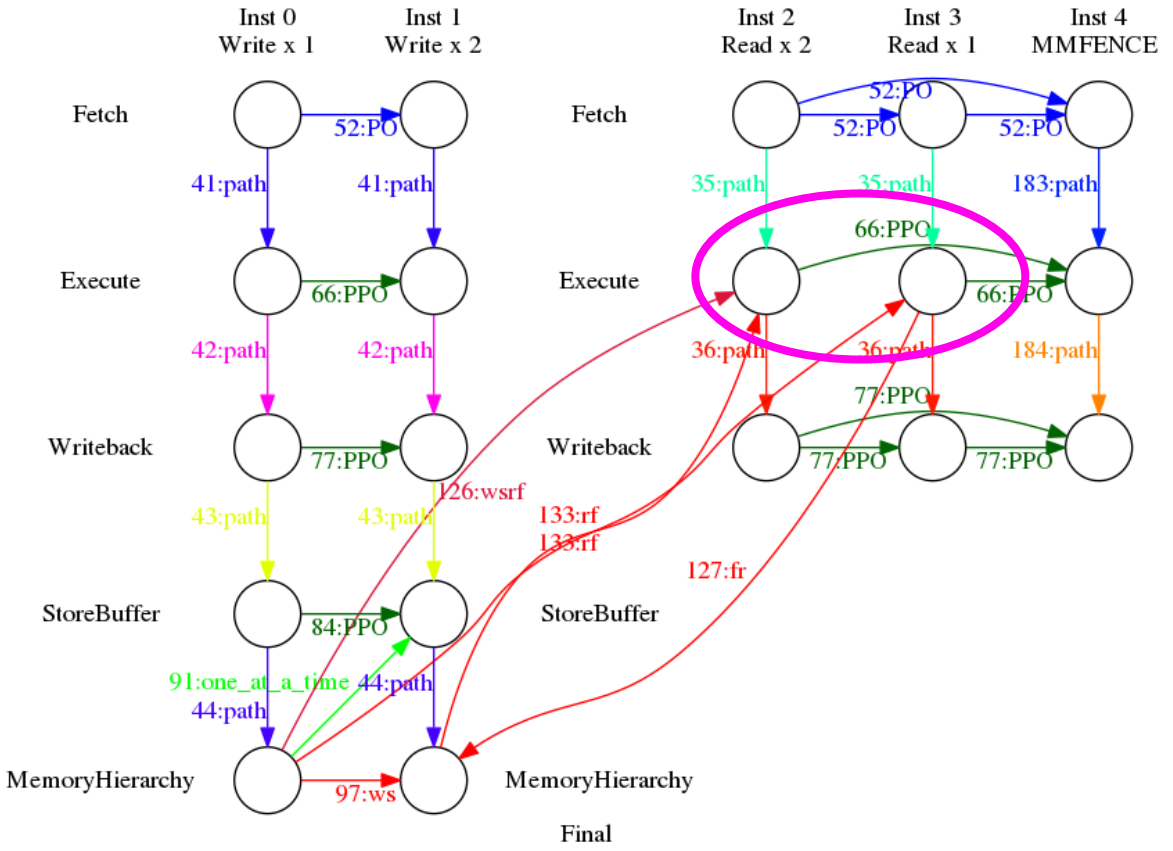
Create BUG.txt and Strict.txt

- cd $TRICHECK_HOME/util
- ./release-parse-results.py
- cat $TRICHECK_HOME/util/results/TSO-RR.uarch/BUG.txt

Bugs exist, so we must refine some combination of inputs and rerun…

Analyzing a bug

- cd $TRICHECK_HOME/util/results/TSO-RR.uarch/corr
- gen_graph -i corr_R_relaxed_fence_acquire_fence_W_relaxed_fence_relaxed_fence.test.gv
- evince corr_R_relaxed_fence_acquire_fence_W_relaxed_fence_relaxed_fence.test.pdf



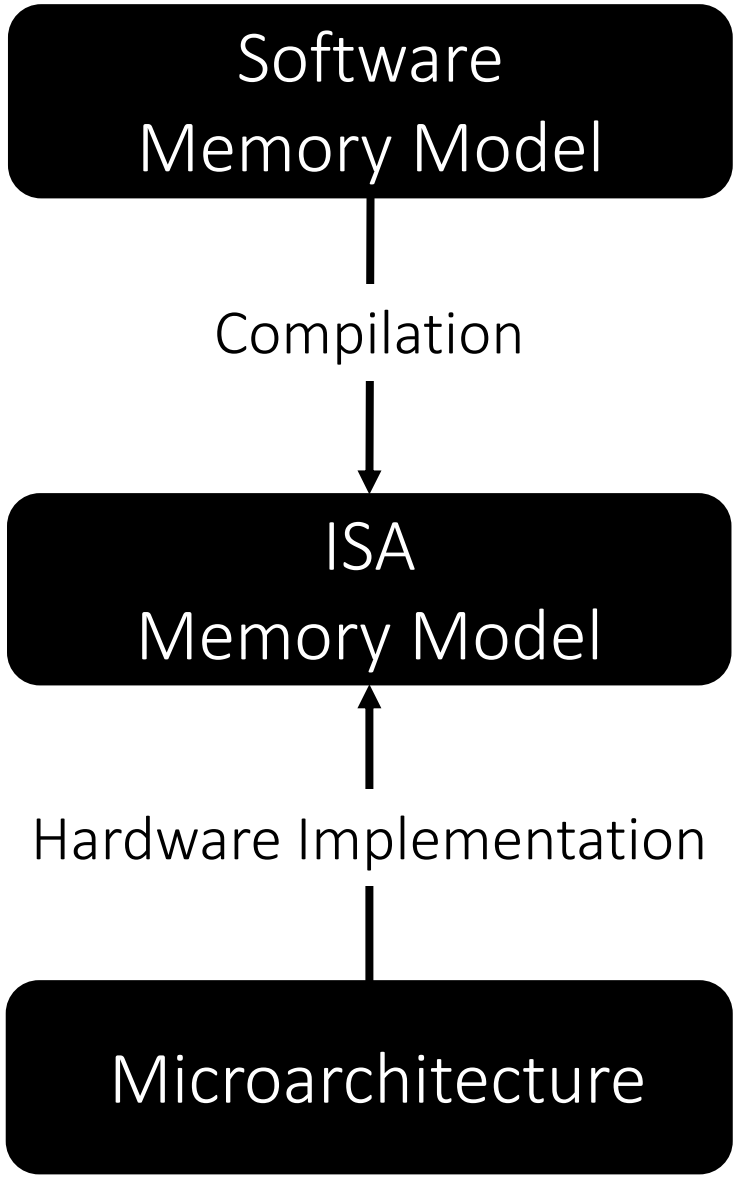
C11 requires that all same-address reads of atomic locations execute in order.

ARM Read-Read Hazard

Initial conditions: data=0, atomic *ptr=&data

Forbidden by C11: r1=2, r2=1

T0	T1
st(data,1,rlx)	st(data,2,rlx)
r1=ld(*ptr,rlx)	
r2=ld(data,rlx)	



C11/C++11	ARMv7
st(rlx)	STR
ld(rlx)	LDR
ld(acq)	LDR; DMB
...	...

C0	C1
ST [data]←1	ST [data]←2
LD [ptr]→r0	
LD [r0]→r1	
LD [data]→r2	

Two loads of the same address

Forbidden outcome observable on Cortex-A9

ARM Cortex-A9

Fixing the bug…

- ARM fixed the bug by modifying the compiler, so we'll do the same thing here…
- Modify compiler mapping in $TRICHECK_HOME/util/compile.txt

C/C++11 Operation	TSO-RR implementation
Load Relaxed:	Read, MMFENCE
Load Acquire:	Read, MMFENCE
Load Seq_Cst:	Read, MMFENCE
Store Relaxed:	Write
Store Release:	Write
Store Seq Cst:	Write, MMFENCE

```
C11/C++11 op    | prefix;prefix | suffix;suffix
Read relaxed    | NA            | MMFENCE
Write relaxed   | NA            | NA
Read acquire    | NA            | MMFENCE
Write release   | NA            | NA
Read seq_cst    | NA            | MMFENCE
Write seq_cst   | NA            | MMFENCE
```

Run TriCheck On Refined Inputs

- cd $TRICHECK_HOME/util
- rm –r $TRICHECK_HOME/util/tests/ctests/*/pipecheck
- ./release-generate-tests.py --all --fences
- ./release-run-all.py --pipecheck=/home/check/pipecheck_tutorial/src
- ./release-parse-results.py
- cat $TRICHECK_HOME/util/results/TSO-RR.uarch/BUG.txt

Outline

- TriCheck Introduction
- Auto-generating HLL litmus tests
- User-defined TriCheck inputs
- Iterative ISA design example
- Bugs Found with TriCheck: RISC-V Case Study and Compiler Mappings
- Ongoing Work & Conclusions

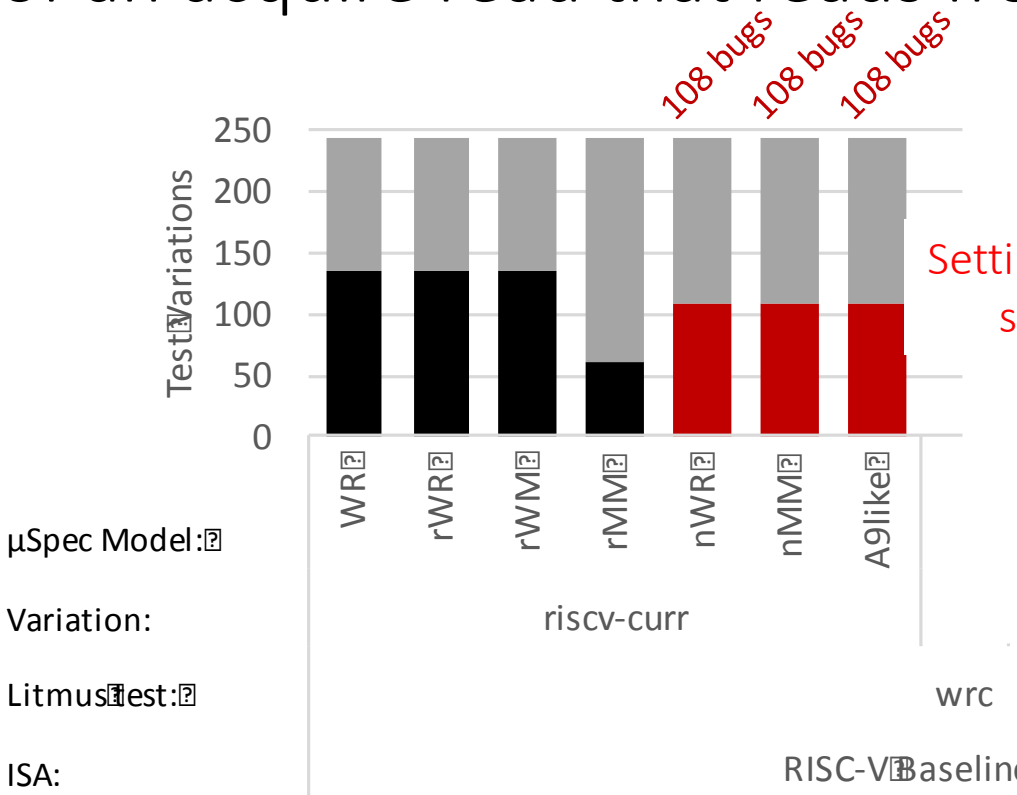
RISC-V Case Study

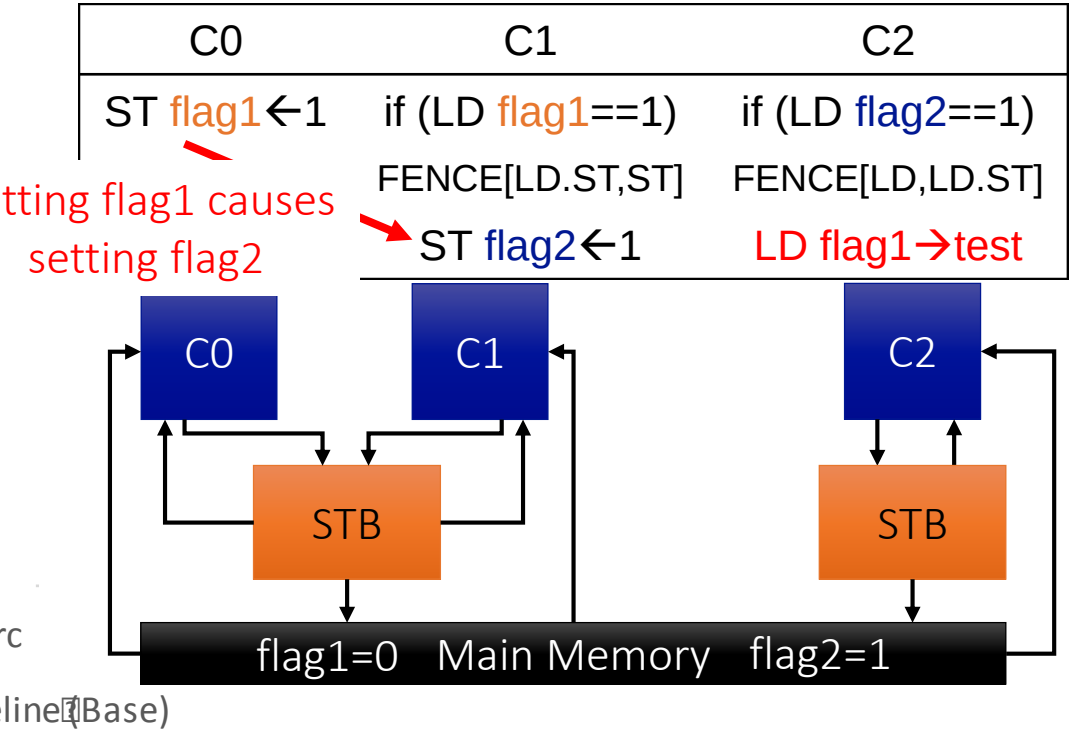
- Create μspec models for 7 distinct RISC-V implementation possibilities:
  - All abide by current RISC-V spec
  - Vary in preserved program order and store atomicity
- Started with stricter-than-spec microarchitecture: RISC-V Rocket Chip
  - TriCheck detects **bugs**: refine for correctness
  - TriCheck detects **over-strictness**: Performed legal (per RISC-V spec) hardware relaxations
- Impossible to compile C11 for RISC-V as originally specified
- Out of 1,701 tested C11 programs:
  - RISC-V-Base-compliant design allows 144 buggy outcomes
  - RISC-V-Base+A-compliant design allows 221 buggy outcomes

RISC-V Base: Lack of Cumulative Fences

Initial conditions: x=0, y=0		
T0	T1	T2
a: sw x1, (x5)	b: lw x2, (x5)	e: lw x3, (x6)
	c: fence rw, w	f: fence r, rw
	d: sw x2, (x6)	g: lw x4, (x5)
Forbidden HLL Outcome: x1=1, x2=1, x3=1, x4=0		

C11 acquire/release synchronization is transitive: accesses before a release write in program order, and observed by the releasing core prior to the release write must be ordered before the release from the viewpoint of an acquire read that reads from the release write





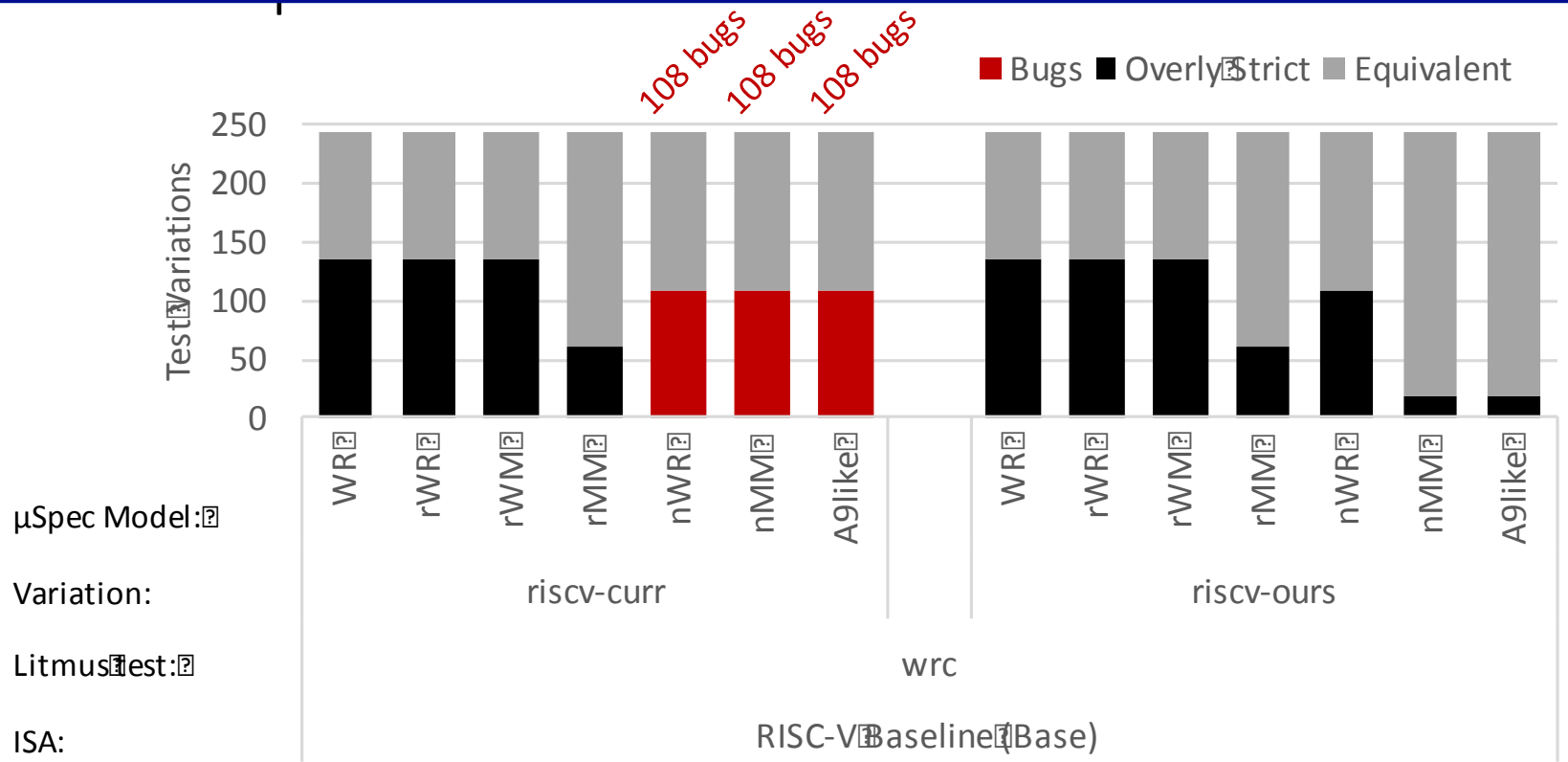
RISC-V Base: Lack of Cumulative Fences

Initial conditions: x=0, y=0		
T0	T1	T2
a: sw x1, (x5)	b: lw x2, (x5)	e: lw x3, (x6)
	c: fence rw, w	f: fence r, rw
	d: sw x2, (x6)	g: lw x4, (x5)
Forbidden HLL Outcome: x1=1, x2=1, x3=1, x4=0		

Base RISC-V ISA lacks cumulative fences

- Cumulative fence needed to enforce order between different-thread accesses
- Cannot fix bugs by modifying compiler

Our solution: add cumulative fences to the Base RISC-V ISA



More results in the paper:

- Both Base and Base+A:
  - Lack of cumulative lightweight fences
  - Lack of cumulative heavyweight fences
  - Re-ordering of same-address loads (Like tutorial example)
  - No dependency ordering, but Linux port assumes it
- Base+A only:
  - Lack of cumulative releases; no acquire-release synchronization
  - No roach-motel movement

Since publishing these results, a RISC-V Memory Model Working Group was formed to design a robust MCM specification for the RISC-V ISA that meets the needs of RISC-V users and supports C11.

As of a few days ago, the new MCM proposal passed the 45 day ratification period.

Evaluating Compiler Mappings with TriCheck

- During RISC-V analysis, we discovered two counter-examples while using the "proven-correct" *trailing-sync* mappings for compiling C11 to POWER/ARMv7
- Also incorrect: the *proof* for the C11 to POWER/ARMv7 trailing-sync compiler mappings [Manerkar et al., CoRR '16]

TriCheck Conclusions

- Memory model design choices are complicated =>
  - Verification calls for automated analysis to comprehensively tackle subtle interplay between many diverse features.
- TriCheck uncovered flaws in the RISC-V memory model…
  - But more generally, TriCheck can be used on any ISA.
- Languages and Compilers matter too…
  - TriCheck uncovered bugs in the trailing-sync compiler mapping from C11 to POWER/ARMv7

Outline

- Introduction
- Motivating Example
- Overview of Our Work
- MCM Background & Our Approach
- PipeCheck: Verifying Hardware Implementations against ISA Specs
  - Graph-based happens-before analysis of program executions on hardware
  - μspec DSL for specifying axiomatic models of hardware
- TriCheck: Expanding to HW/SW Stack Interface Issues
- Looking forward: Other uses of tools and techniques
  - CCICheck, COATCheck, SecurityCheck, …

COATCheck: Verifying Memory Ordering at the Hardware-OS Interface

Daniel Lustig, Geet Sethi[+], Michael Pellauer*,
Margaret Martonosi, Abhishek Bhattacharjee [+]

Princeton University [+]Rutgers University *NVIDIA

ASPLOS 2016

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

Simple Motivating Example

Initially: [x]=0, [y]=0	
Thread 0	**Thread 1**
St [x] ← 1 Ld [y] → r1	St [y] ← 2 Ld [x] → r2
Proposed outcome: r1=2, r2=1	

Permitted if x and y are different addresses

Initially: [x]=0, [y]=0	
Thread 0	**Thread 1**
St PA1←1 Ld PA2→r1	 St PA2←2 Ld PA1→r2
Outcome r1=2, r2=1 permitted	

Forbidden if x and y are synonyms

Initially: [x]=0, [y]=0	
Thread 0	**Thread 1**
St PA1←1 Ld PA1→r1	 St PA1←2 Ld PA1→r2
Outcome r1=2,r2=1 forbidden	

“Transistency Model”

- Memory ordering verification is fundamentally incomplete unless it explicitly accounts for address translation
- Superset of consistency which captures all address translation-aware sets of ordering rules
- Most prior techniques ignore the implications of virtual-to-physical address translation on memory ordering
  - E.g., synonyms, and page permission updates
- Microarchitectural events and OS behavior can affect memory ordering in ways for which standard memory model analysis can be fundamentally insufficient

Ongoing Work

- We've seen how memory model bugs can result in in correct program outcomes that are intermittent/unpredictable
- Currently, we are applying our techniques of exhaustive enumeration and checking of event orderings to other domains
  - Security: Is a hardware design susceptible to a given class of security exploits?
    - Hardware-aware exploit program synthesis
    - We auto-synthesized programs representative of Meltdown & Spectre
    - We also synthesized 2 new exploits related to Meltdown & Spectre but distinct
      - https://arxiv.org/abs/1802.03802
  - IoT: how do we reason about many concurrently acting IoT devices?

Takeaways

- Memory consistency modes matter
  - Reliability, correctness, and portability
  - Performance
  - Security
- Intuitive “checking” through automated verification
- Move memory model verification earlier in the design processes
- Evaluate across interfaces and design boundaries
  - If interfaces are often source of bugs
- Speed of approach enables new oppertunities
  - Comprehenseive and fast verification for iterative design

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