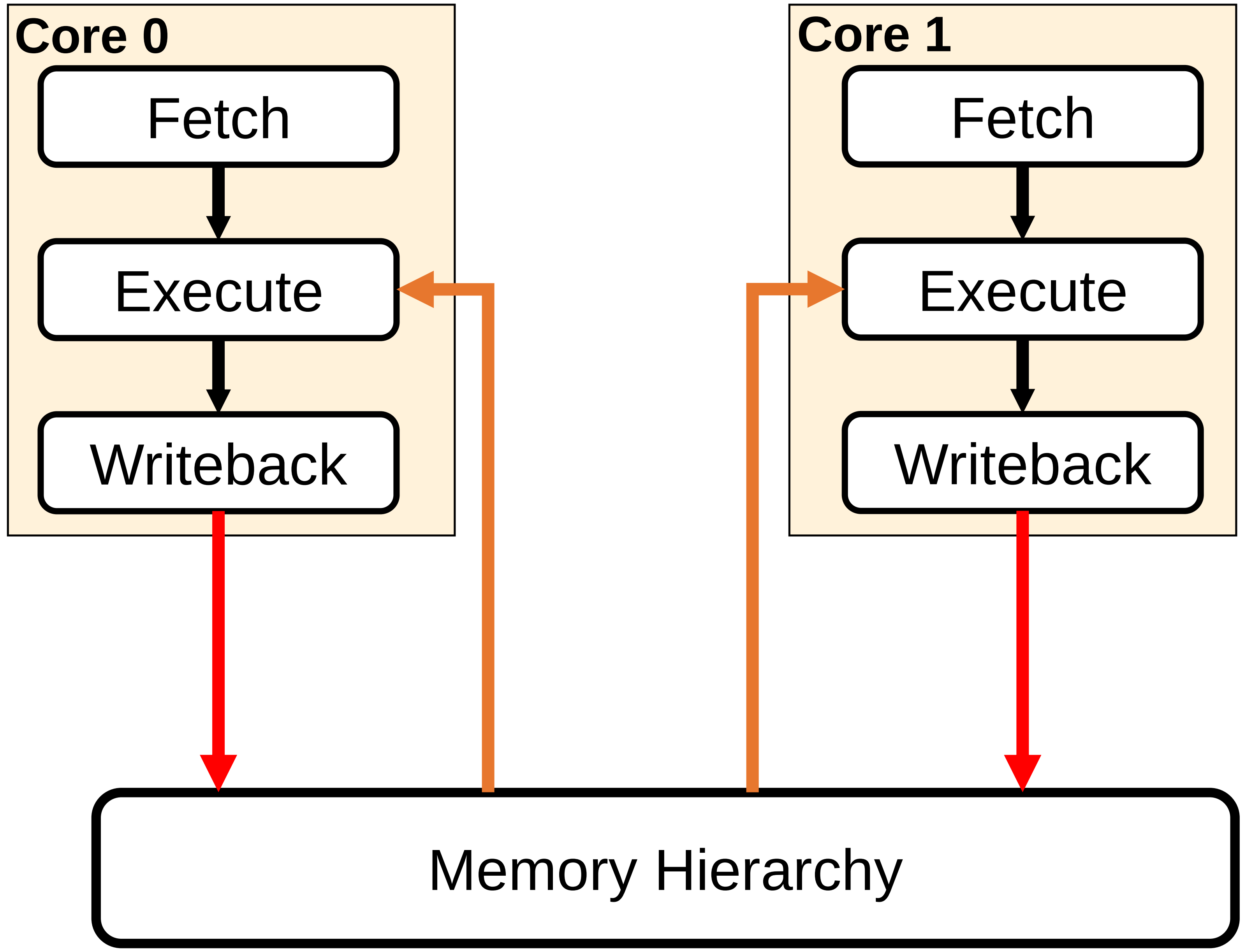
PipeCheck Hands-On

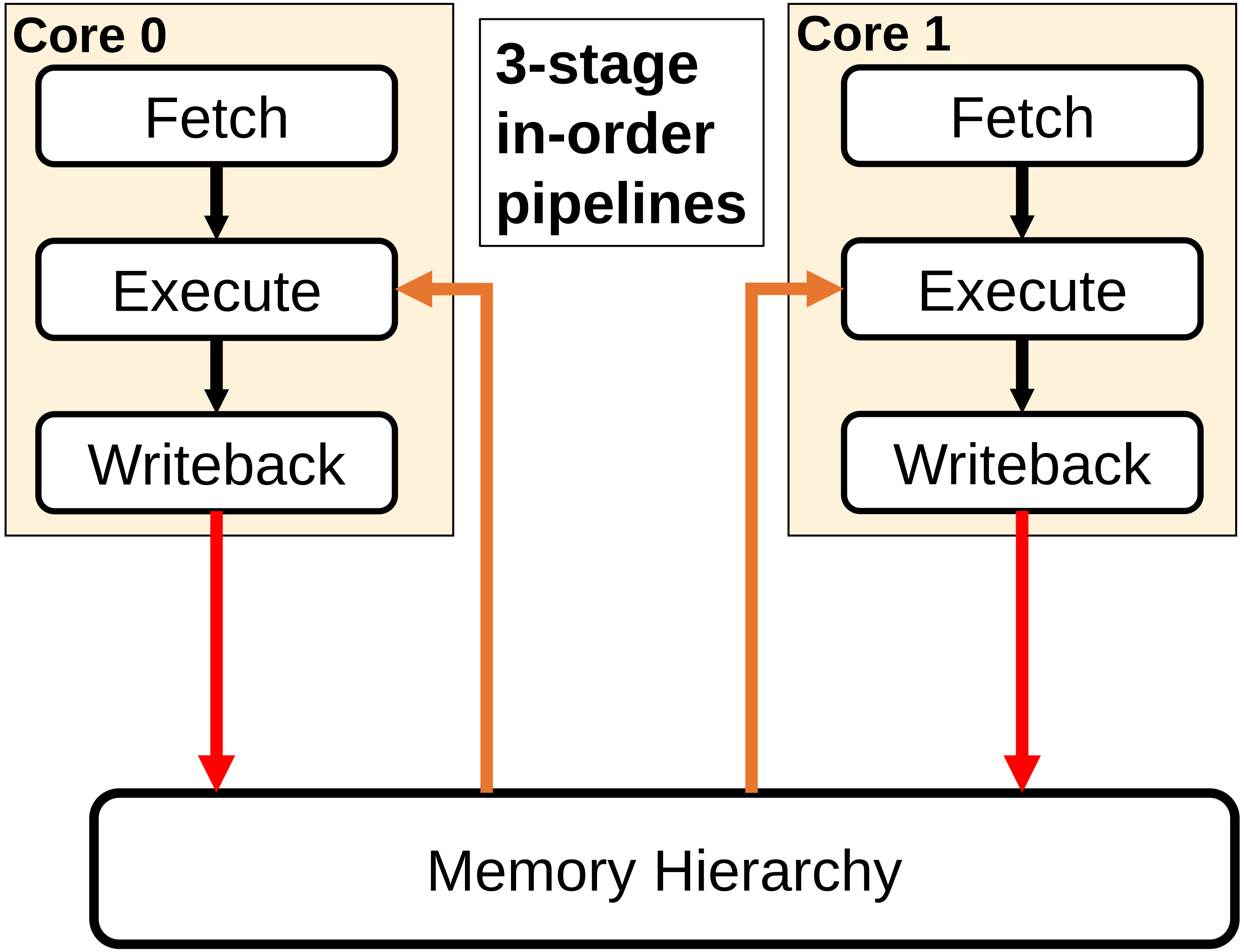
Overview

- Will take you through modelling simple uarches in µSpec
- **simpleSC:** An SC microarchitecture
  - Partially completed uarch specification in VM, you will fill in remainder
- Initially, will look at verifying individual litmus test programs
- Later, will look at verifying across all programs (an infinite space!)

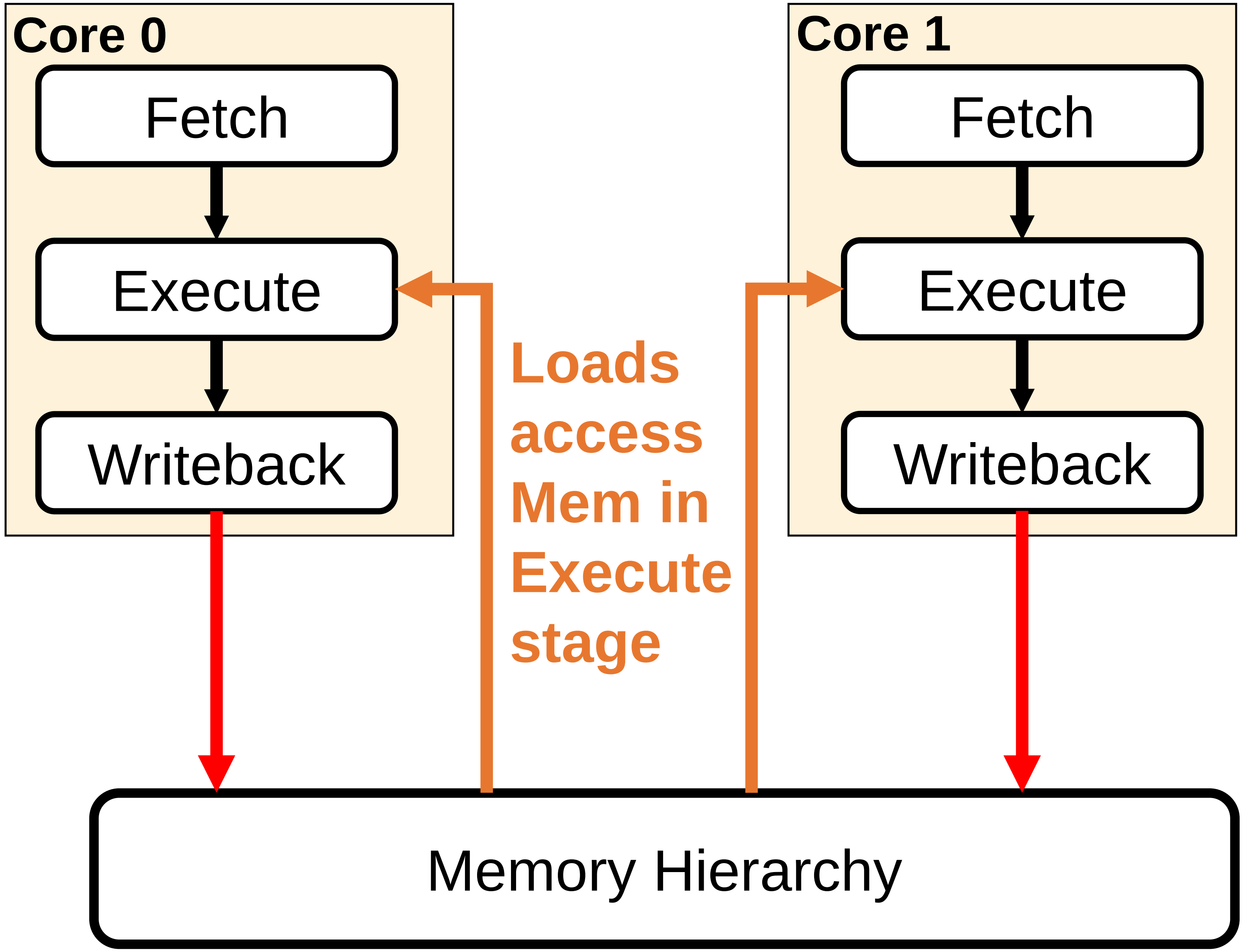
The simpleSC Microarchitecture



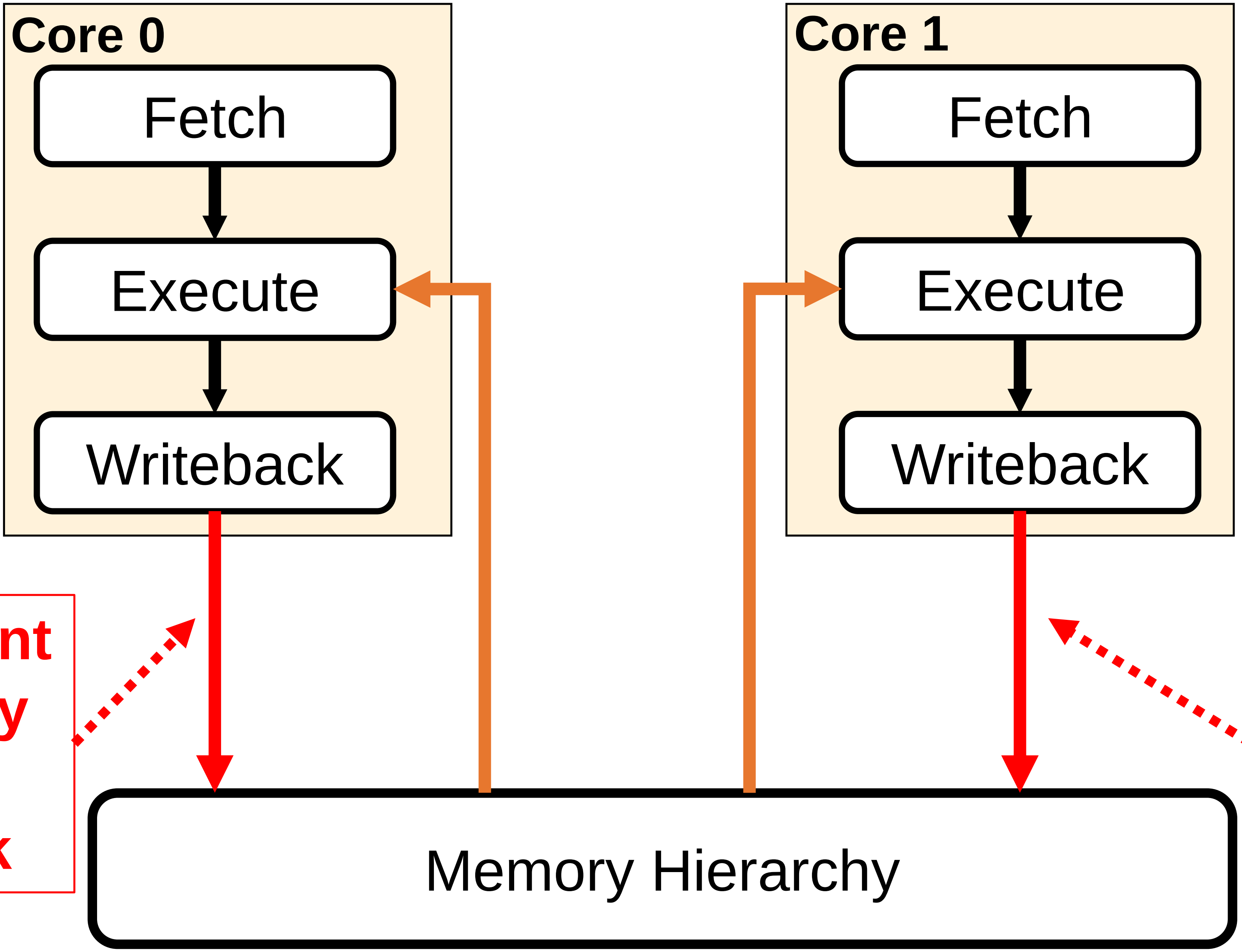
The simpleSC Microarchitecture



The simpleSC Microarchitecture



The simpleSC Microarchitecture



Stores sent to Memory during Writeback

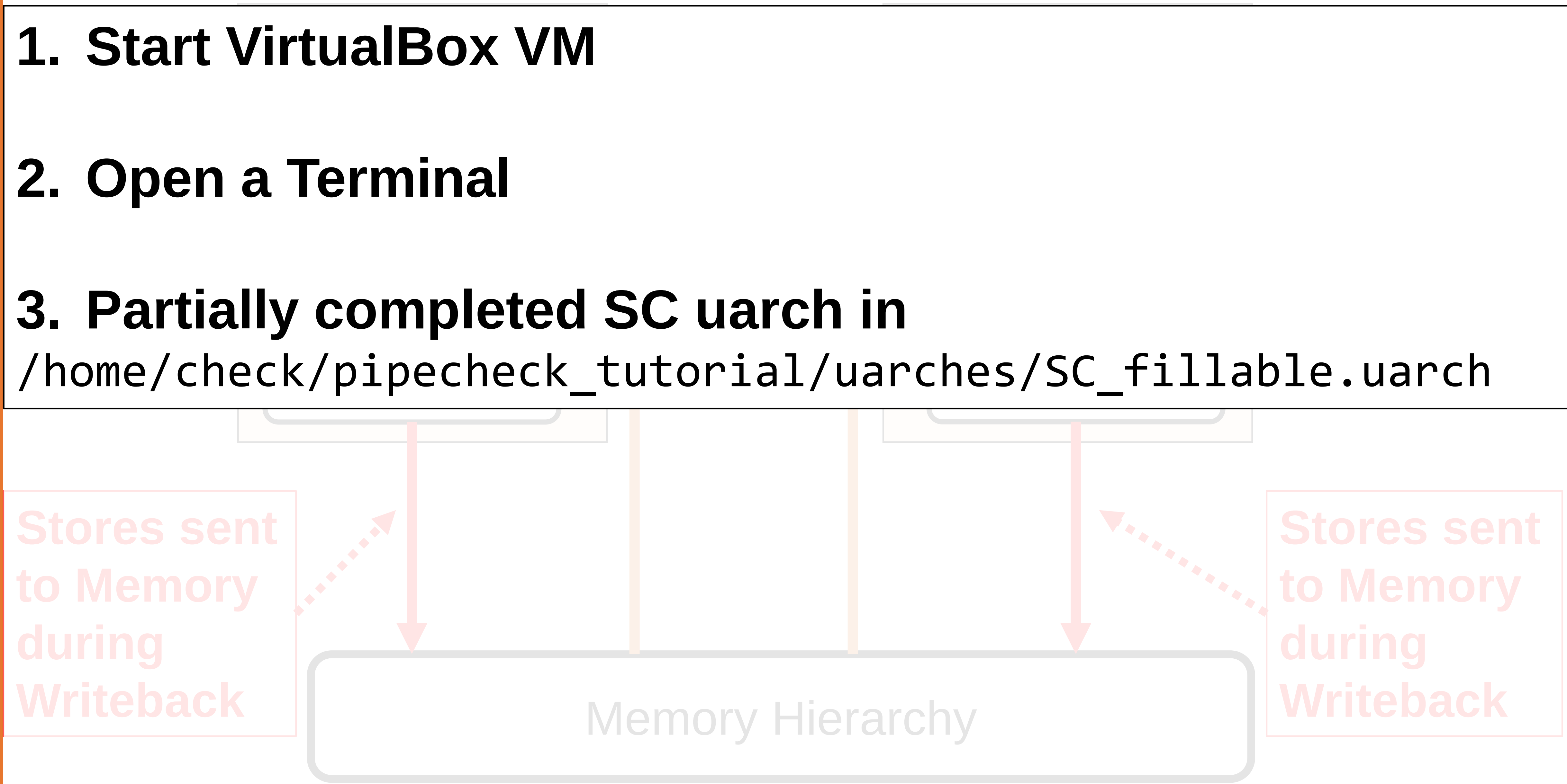
Stores sent to Memory during Writeback

The simpleSC Microarchitecture

1. **Start VirtualBox VM**

2. **Open a Terminal**

3. **Partially completed SC uarch in**
`/home/check/pipecheck_tutorial/uarches/SC_fillable.uarch`


Stores sent to Memory during Writeback
Stores sent to Memory during Writeback
Memory Hierarchy

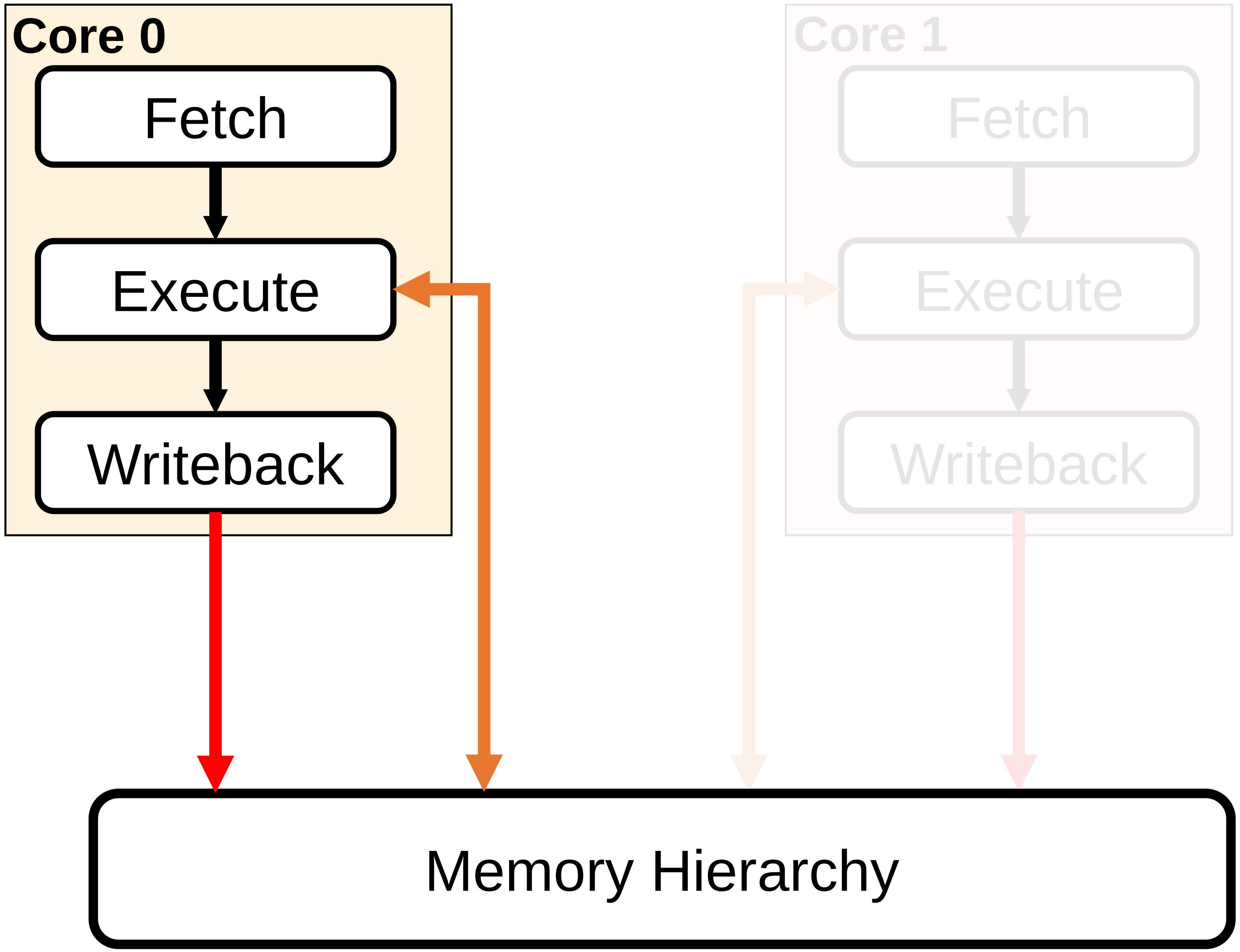
μSpec: A DSL for Specifying Microarchitectures

- Language has capabilities similar to first-order logic (FOL)
  - forall, exists, AND (/\), OR (\/), NOT (~), implication (=>)
  - Has a number of built-in predicates which take memory operations as input
    - e.g. `ProgramOrder i j` where `i` and `j` are loads/stores
    - Other predicates include `SamePhysicalAddress, SameData, IsAnyRead, …`
      - See "Check Quick Start" handout for a more extensive list
  - Predicates can also reference nodes and edges
    - e.g. `EdgeExists ((i1, Fetch), (i2, Fetch))`
    - This predicate is true iff an edge exists between `i1` and `i2`'s `Fetch` stages
  - All μhb edges are transitive (so μSpec is not a subset of FOL)

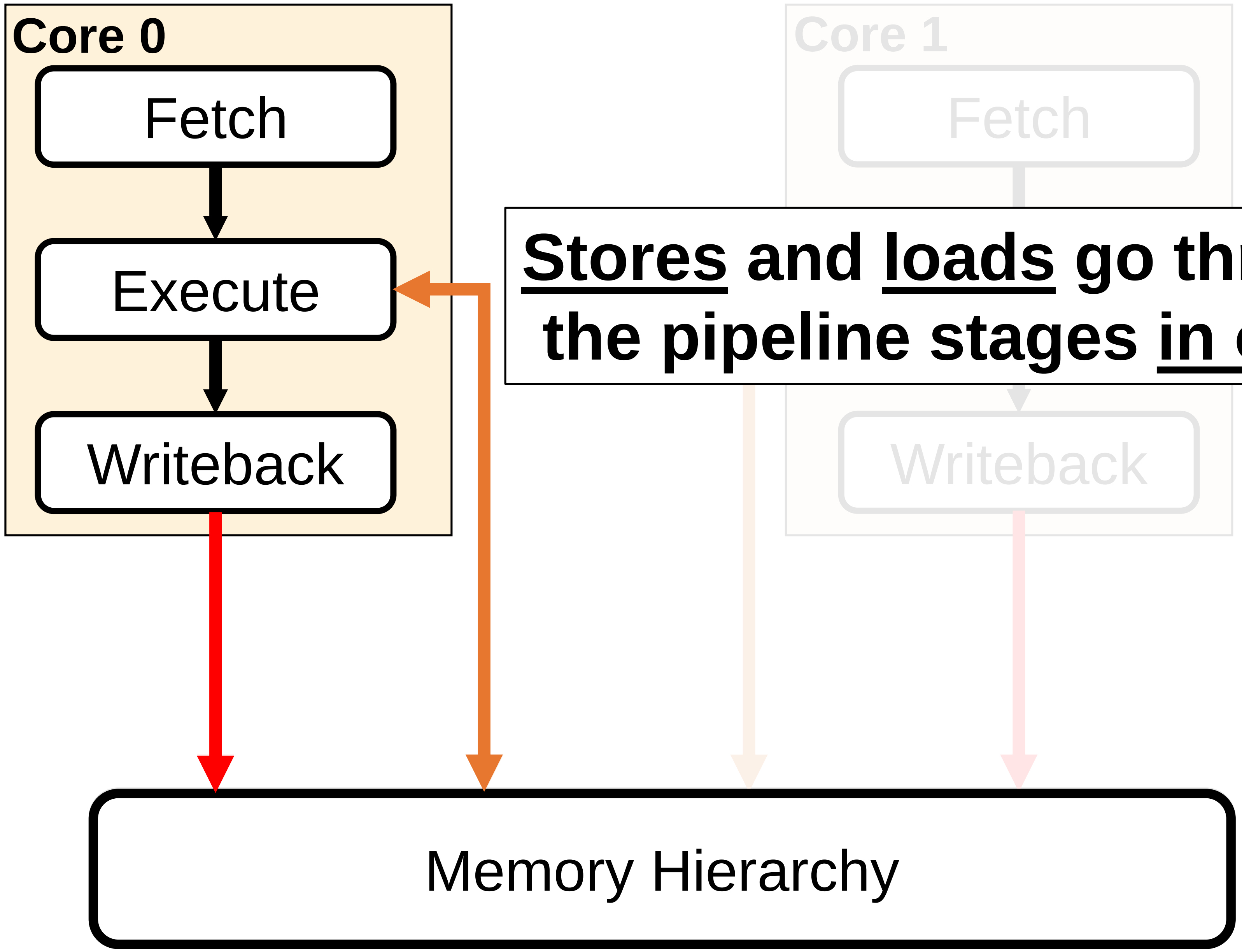
µSpec: A DSL for Specifying Microarchitectures

- Microarchitecture spec has three components:
  - Stage identifier definitions
  - Macro definitions (optional)
  - Axiom definitions
- Macros allow:
  - decomposition of axioms into smaller parts
  - reuse of uspec fragments
- Axioms are each a **partial** ordering on the events in an execution
- Job of PipeCheck is to ensure that these axioms correctly work ***together*** to uphold ISA-level MCM requirements for a litmus test

Finding Axioms

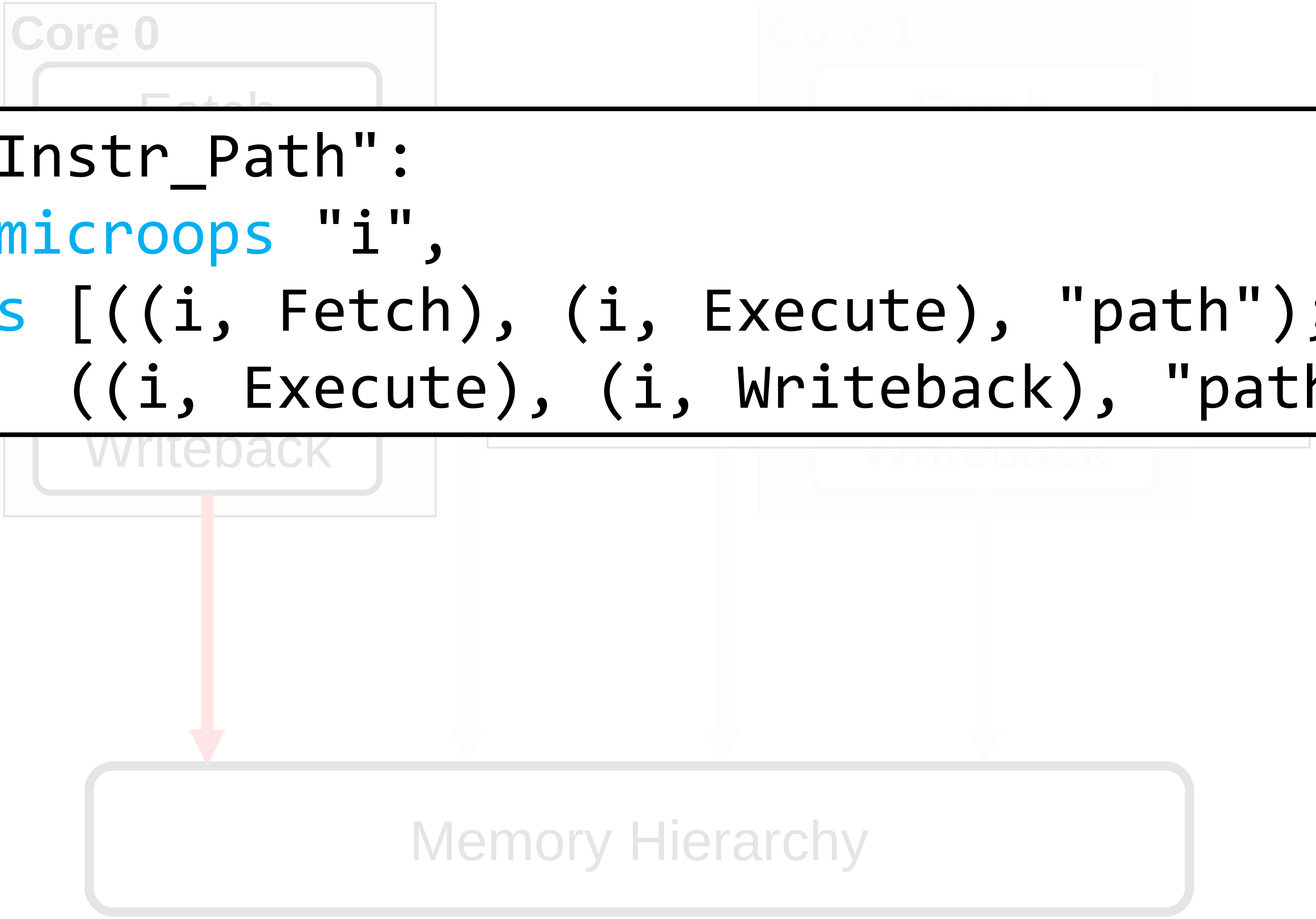


Finding Axioms



Stores and loads go through the pipeline stages in order

The Instr_Path Axiom

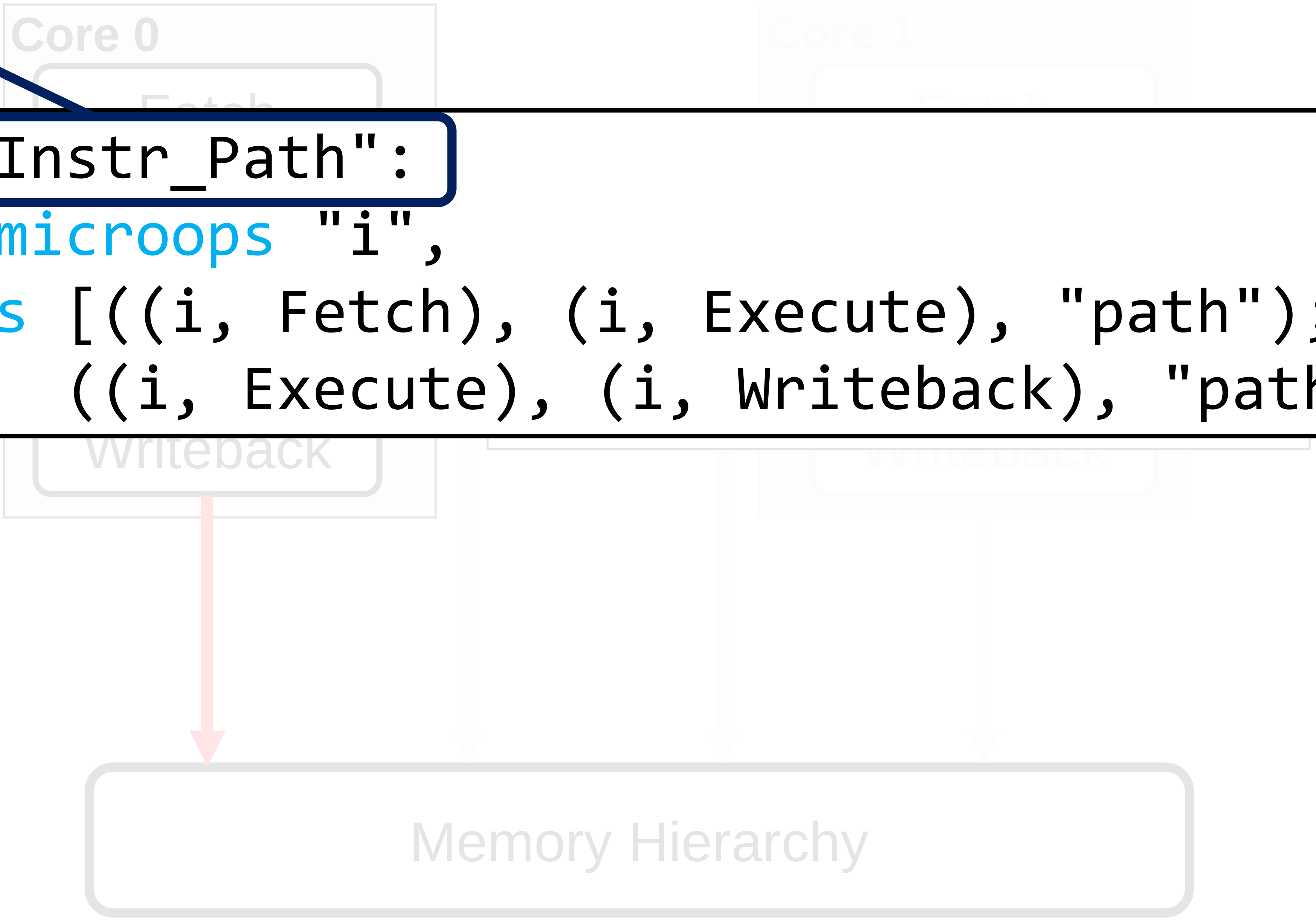


```
Axiom “Instr_Path":
forall microops "i",
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path")].
```

The Instr_Path Axiom

Axiom name

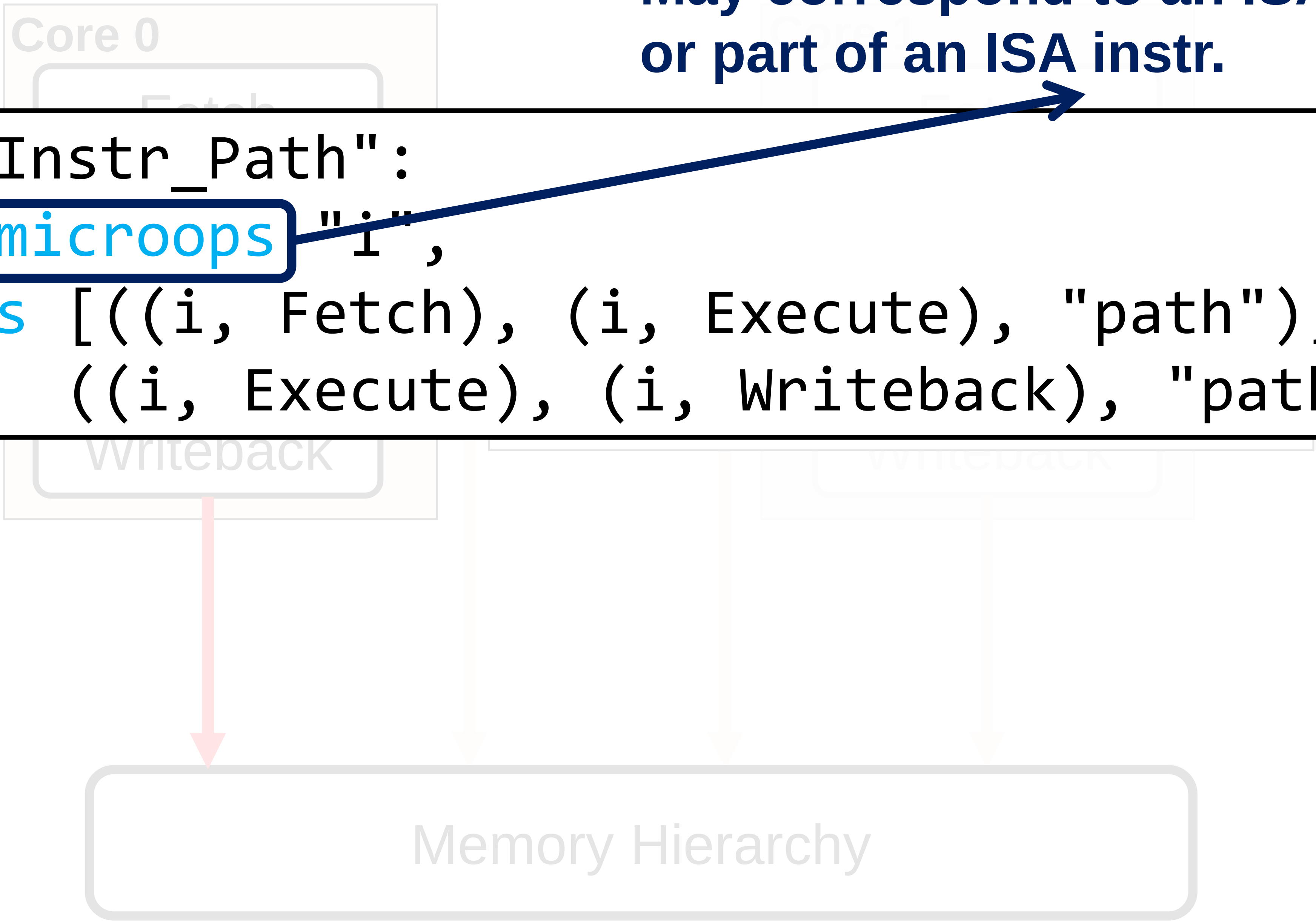
```
Axiom “Instr_Path":
forall microops "i",
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path")].
```



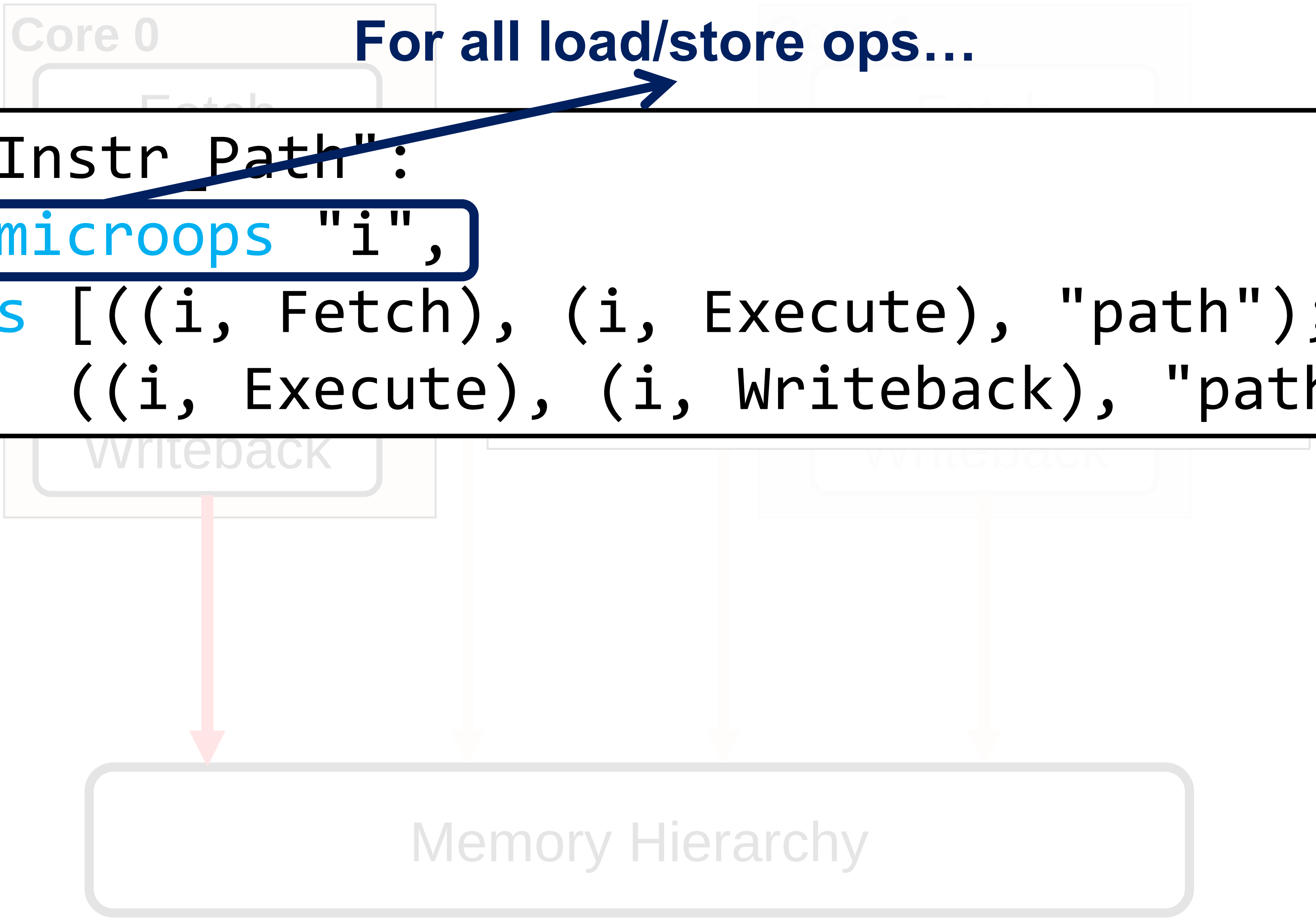
The Instr_Path Axiom

<u>Microop:</u> A single load/store op. May correspond to an ISA instr, or part of an ISA instr.

```
Axiom “Instr_Path":
forall microops "i",
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path")].
```



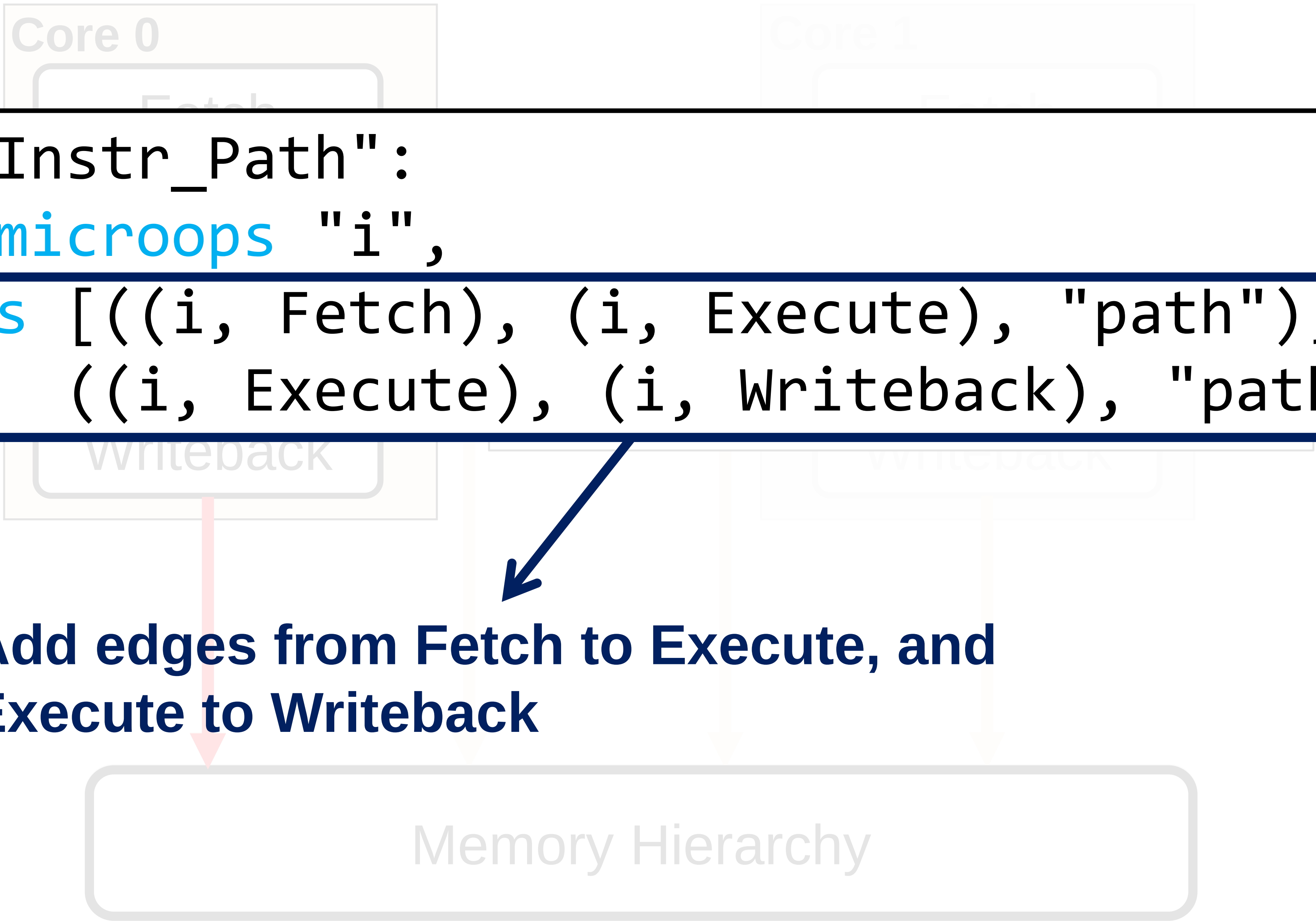
The Instr_Path Axiom



For all load/store ops…

```
Axiom “Instr_Path”:
forall microops "i",
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path")].
```

The Instr_Path Axiom



```
Axiom “Instr_Path":
forall microops "i",
AddEdges [((i, Fetch), (i, Execute), "path");
          ((i, Execute), (i, Writeback), "path")].
```

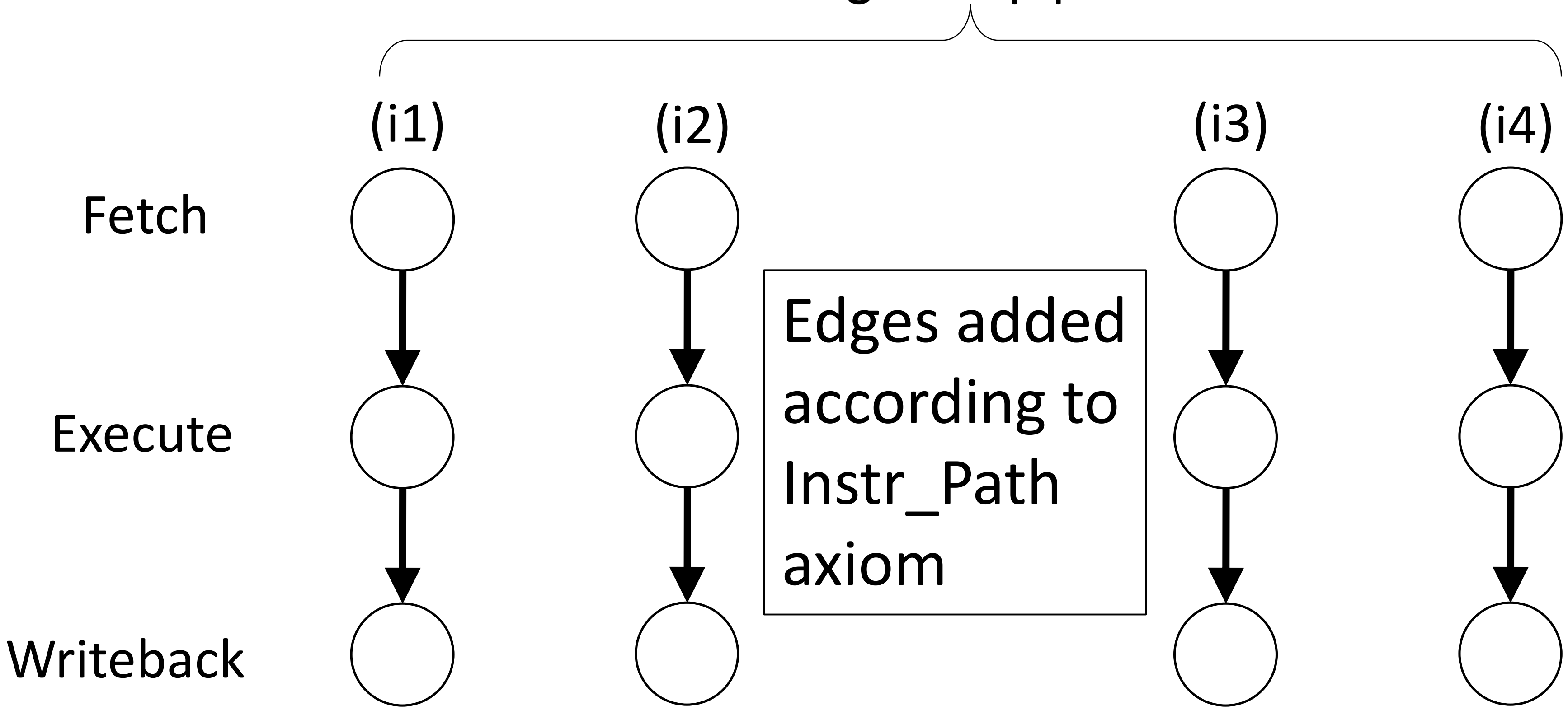
Add edges from Fetch to Execute, and Execute to Writeback

Specifying µSpec Nodes

- A node represents a particular event in a particular instruction’s execution
- Format for nodes is: **`(instr, stage/event_name)`**
- Thus, **`(i, Fetch)`** represents the fetch stage of instruction **`i`**…

μhb Graphs for `co-mp` Using Axioms

Each column represents an instruction flowing through the pipeline



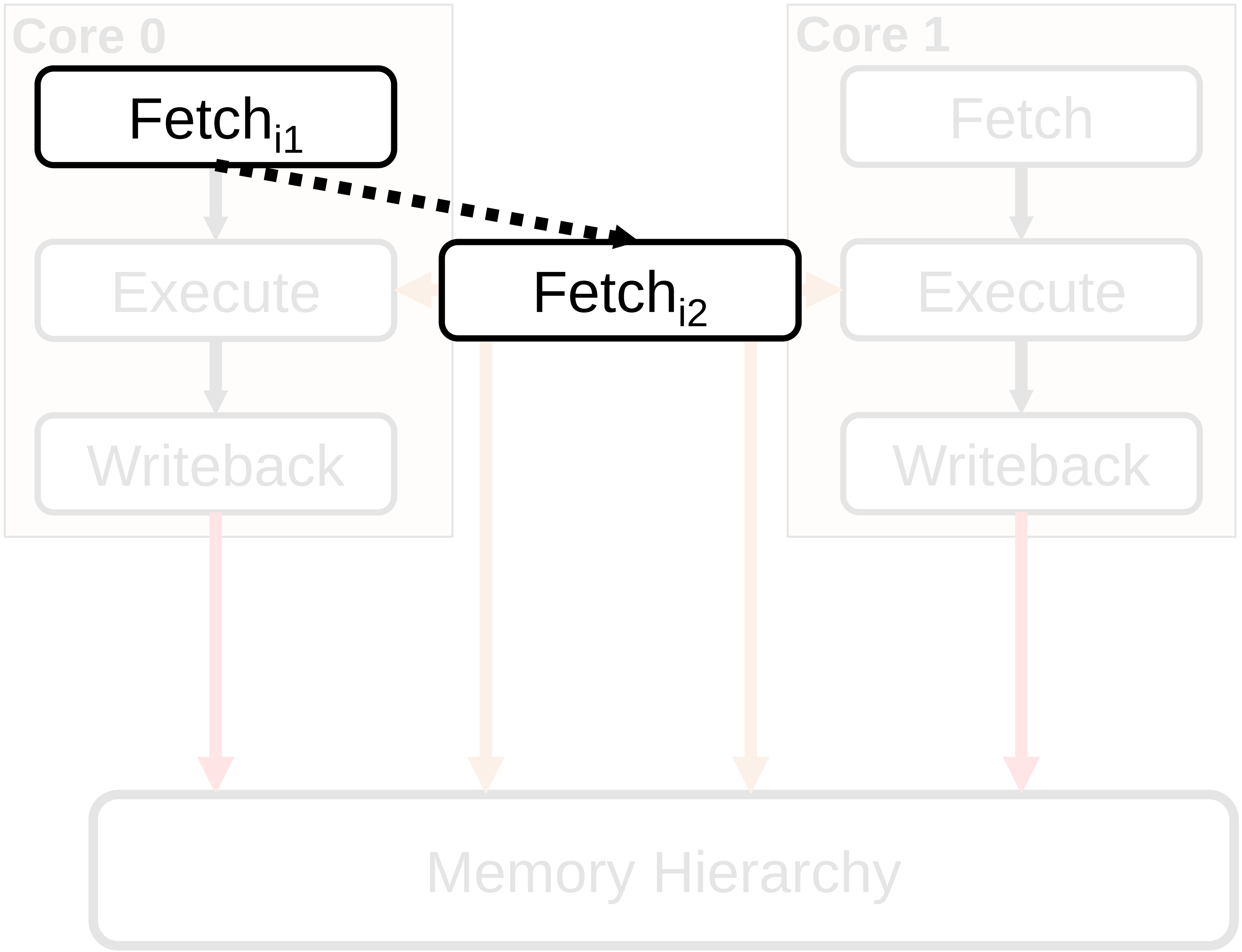
Edges added according to Instr_Path axiom

Initially, Mem[x] = 0

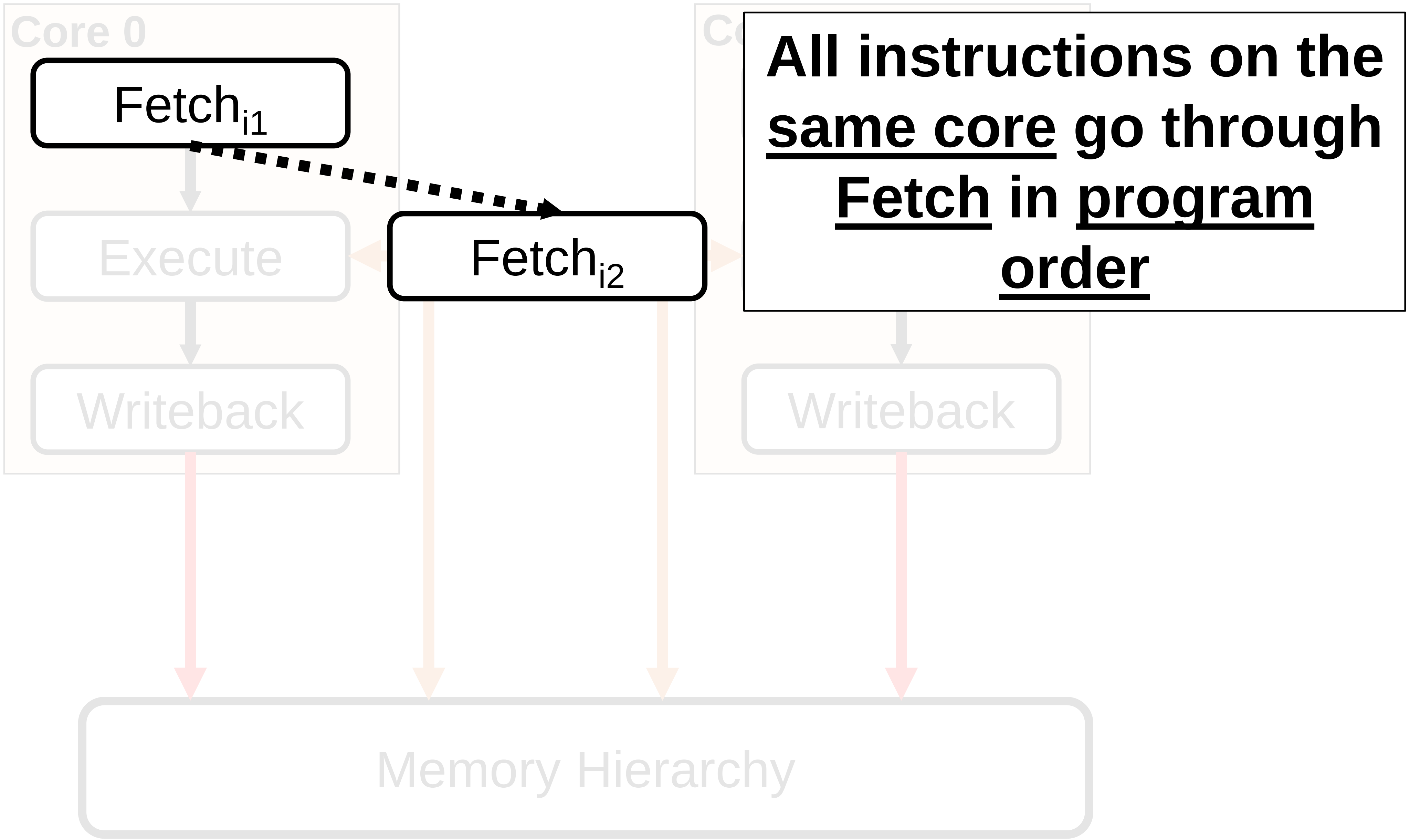
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]

SC **Forbids**: r1=2, r2=1, Mem[x] = 2

Finding Axioms

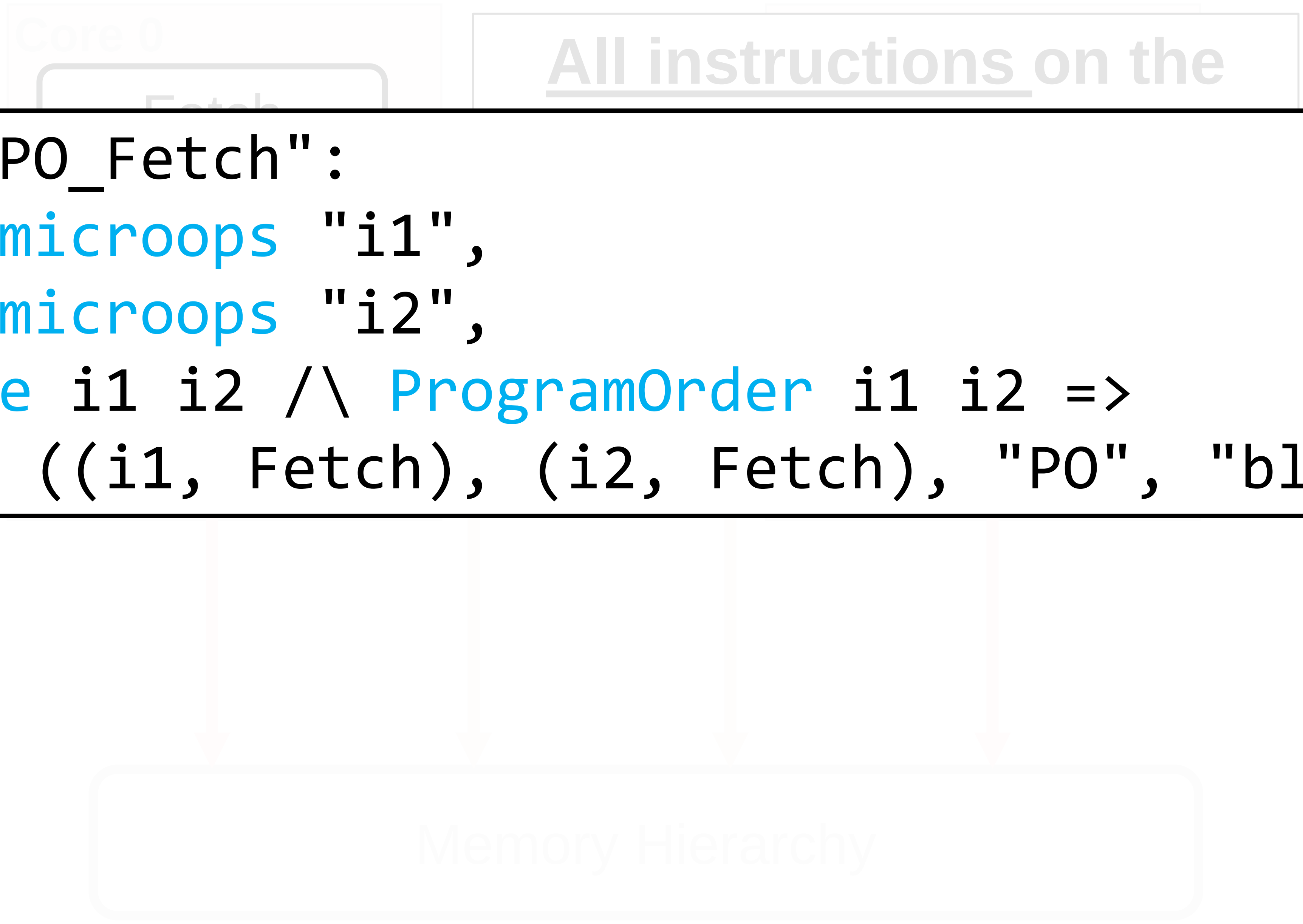


Finding Axioms



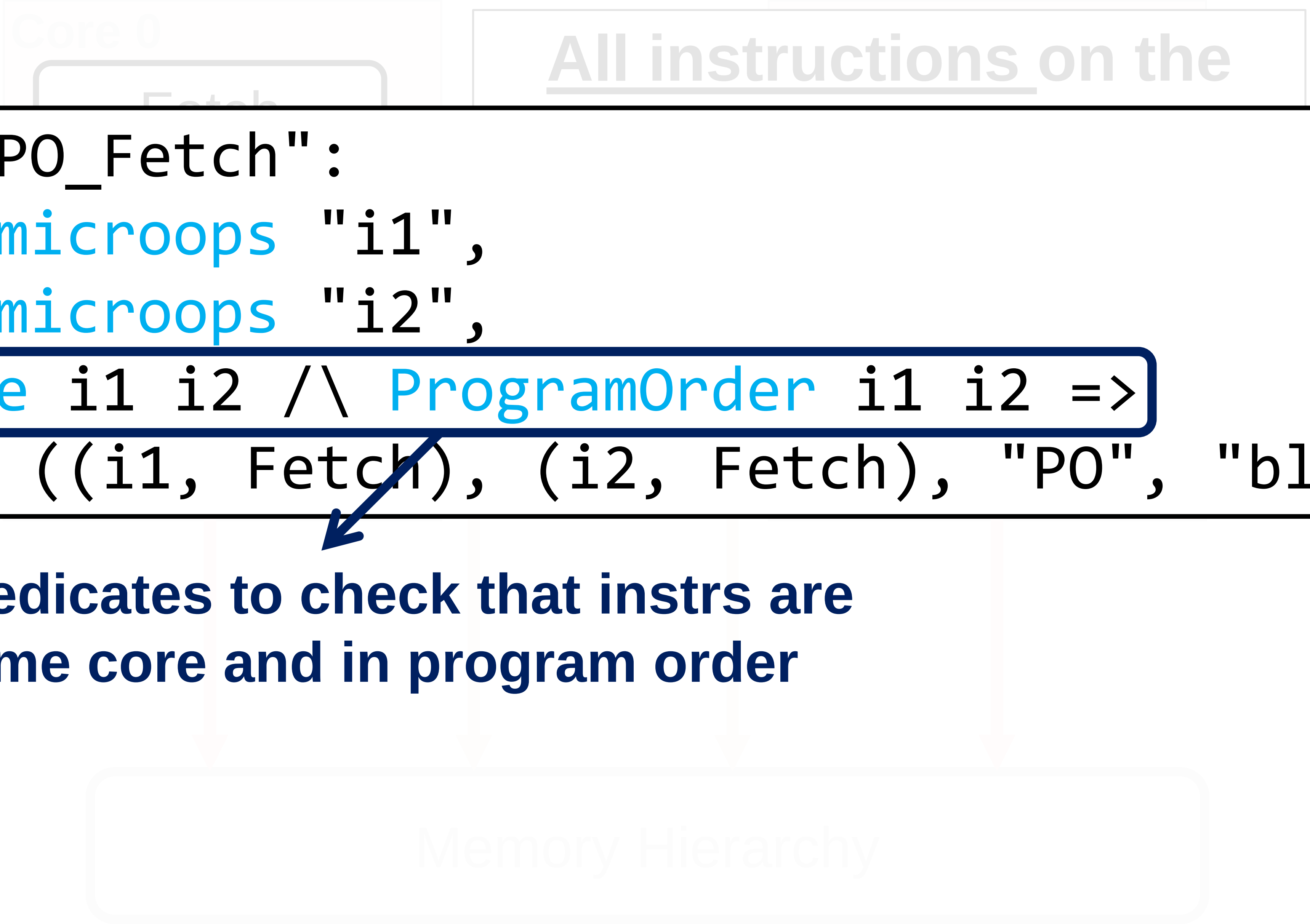
All instructions on the <u>same core</u> go through <u>Fetch</u> in <u>program order</u>

The PO_Fetch Axiom



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

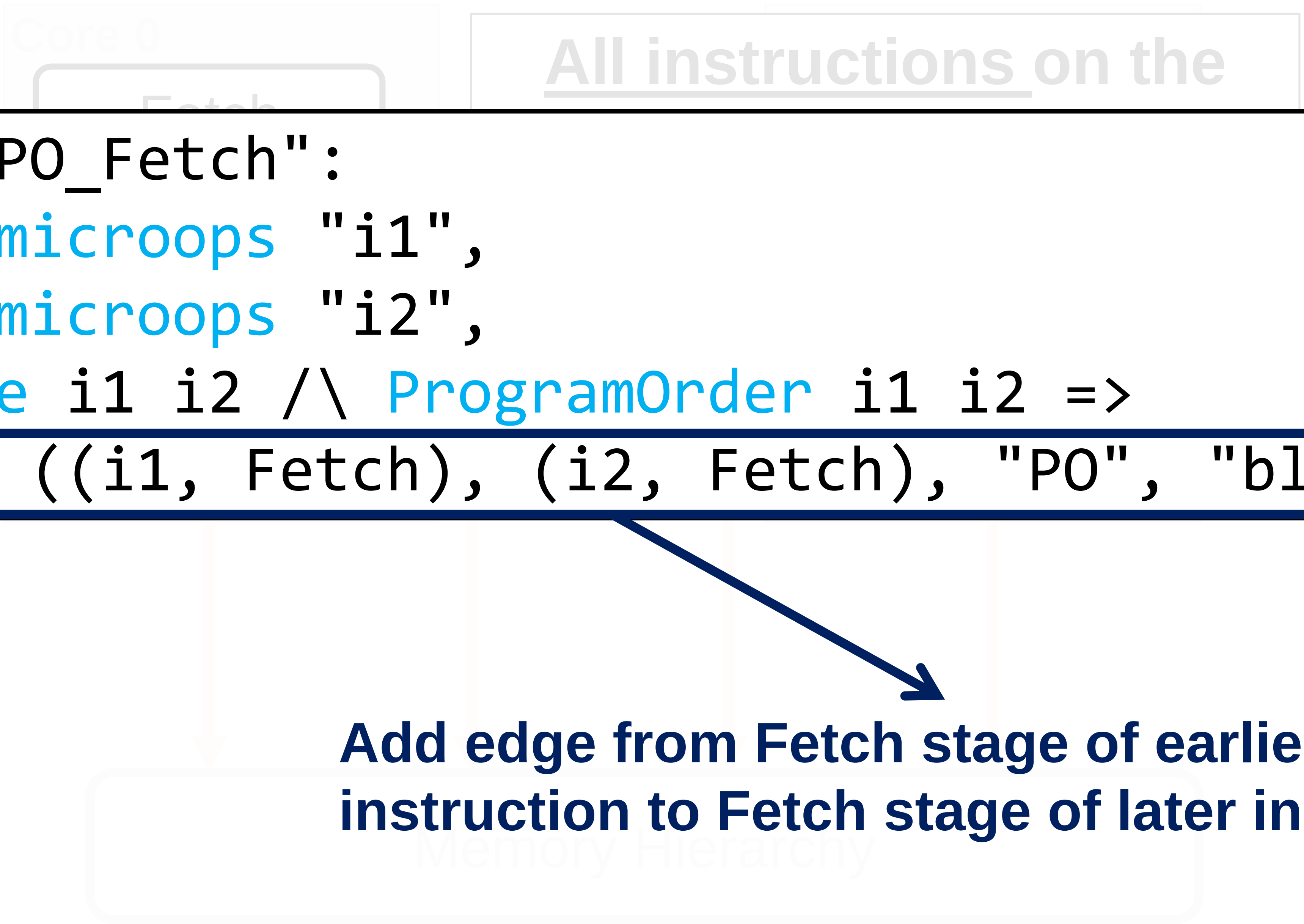
The PO_Fetch Axiom



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

Use of predicates to check that instrs are on the same core and in program order

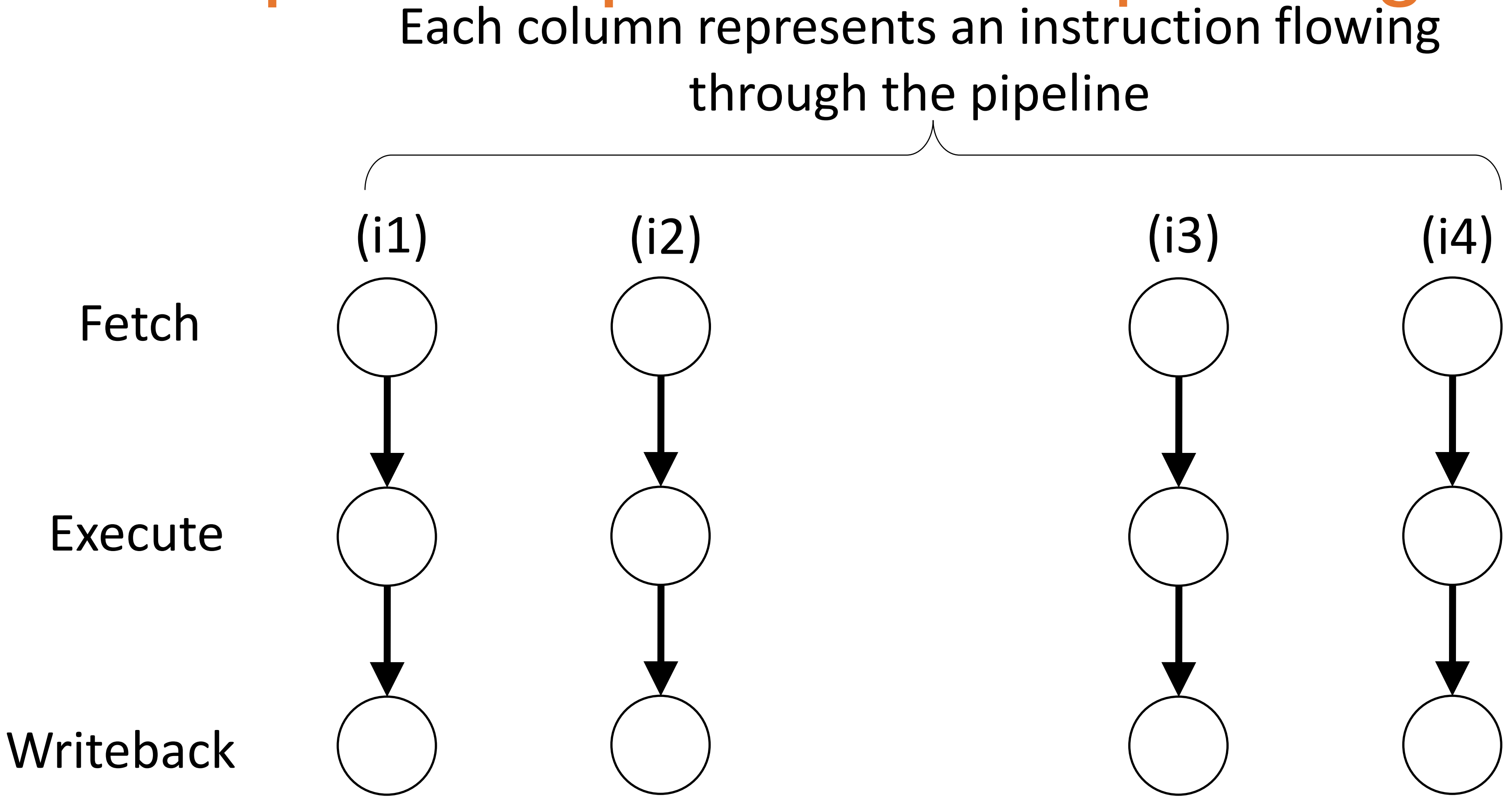
The PO_Fetch Axiom



```
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 to Fetch stage of later instruction

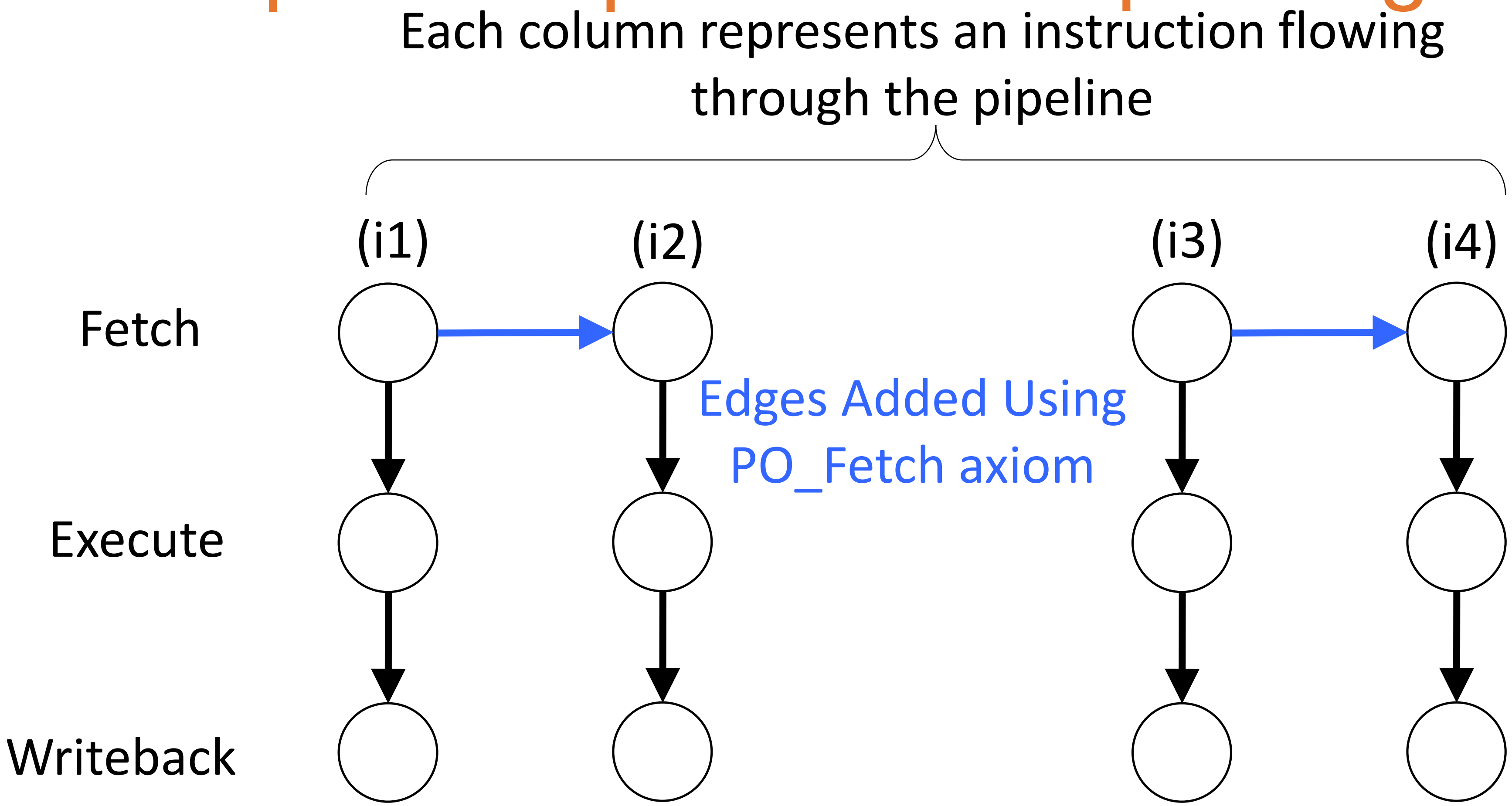
µhb Graphs for `co-mp` Using Axioms


Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback

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	

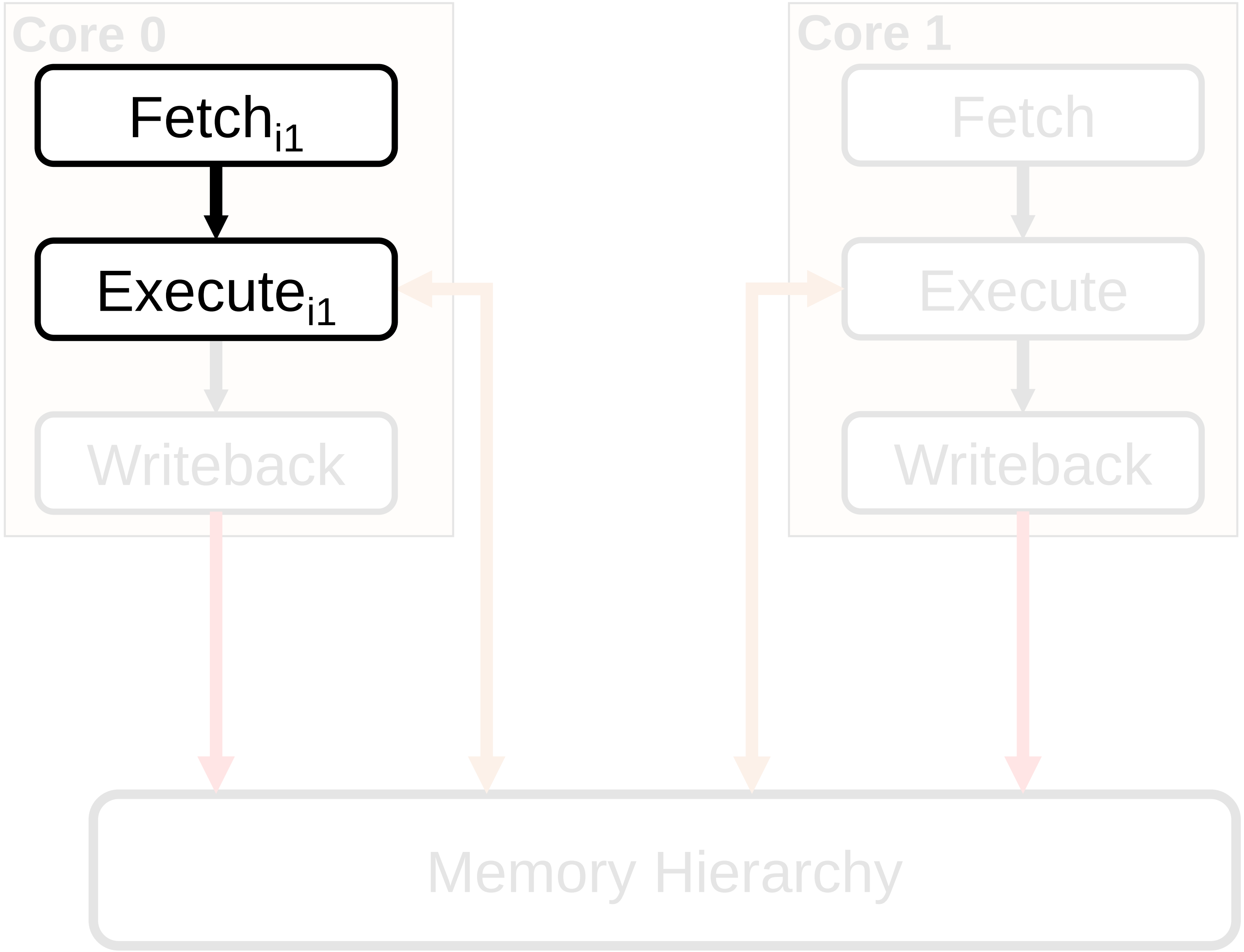
μhb Graphs for `co-mp` Using Axioms


Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback
Edges Added Using PO_Fetch axiom

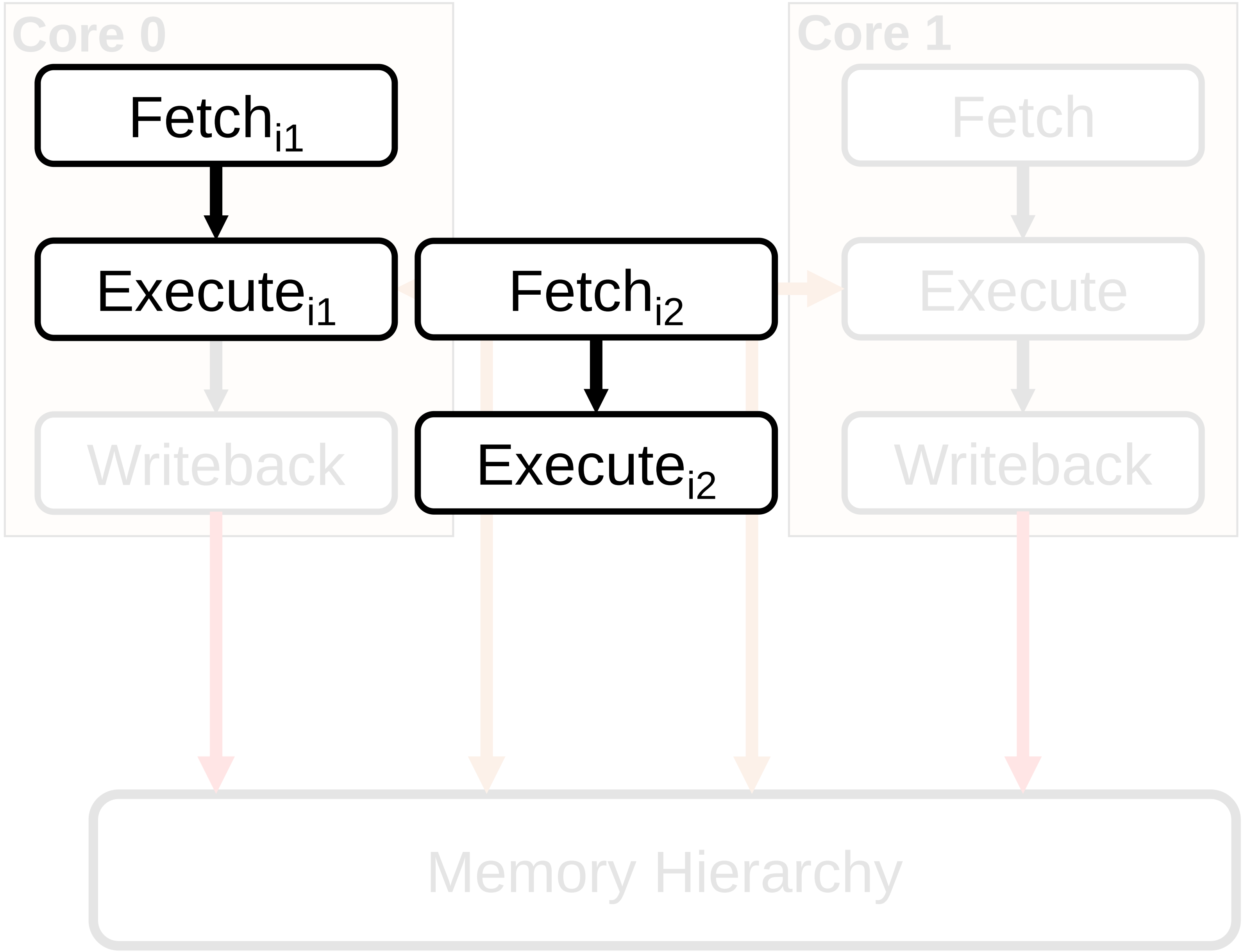
Initially, Mem[x] = 0

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	

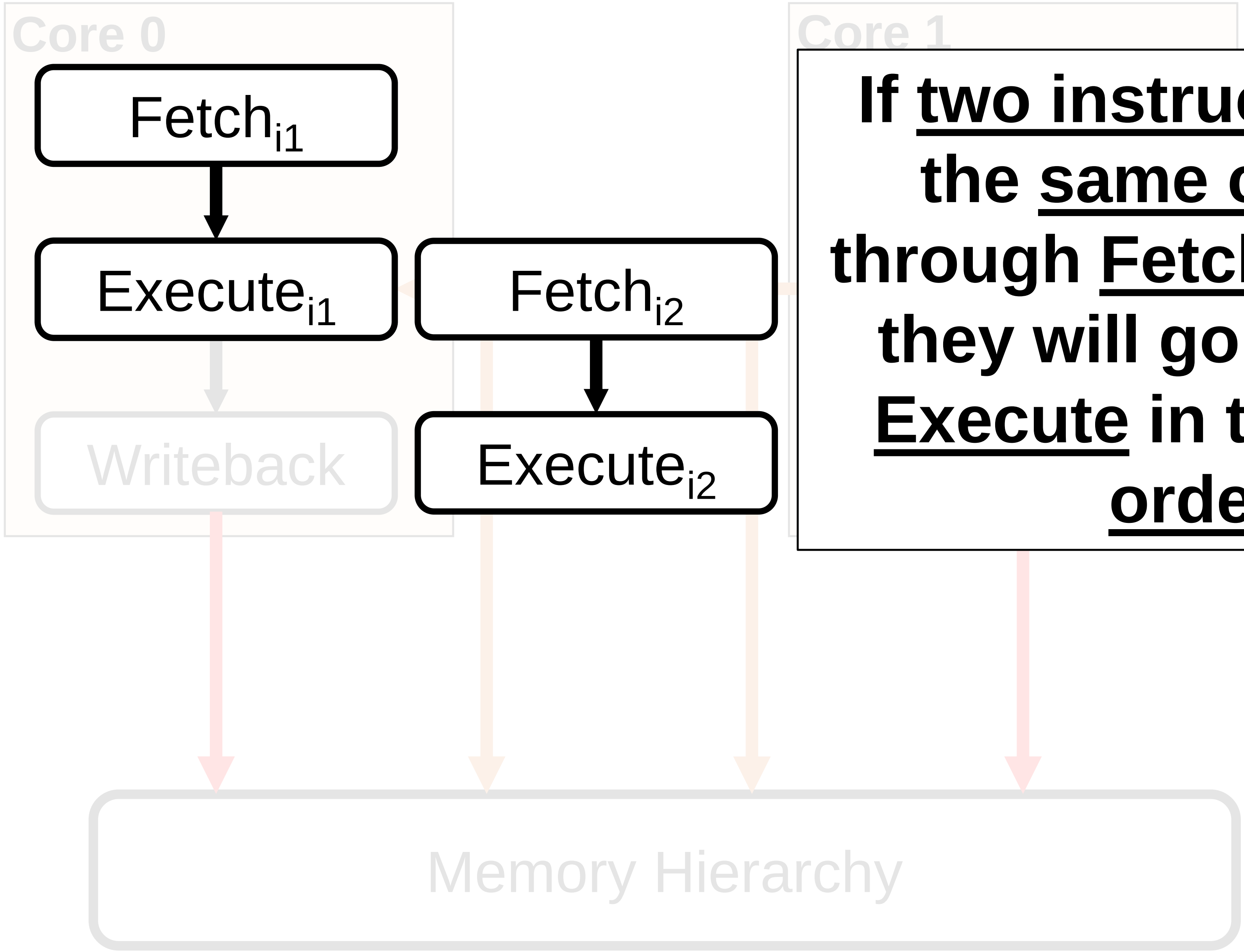
Finding Axioms



Finding Axioms

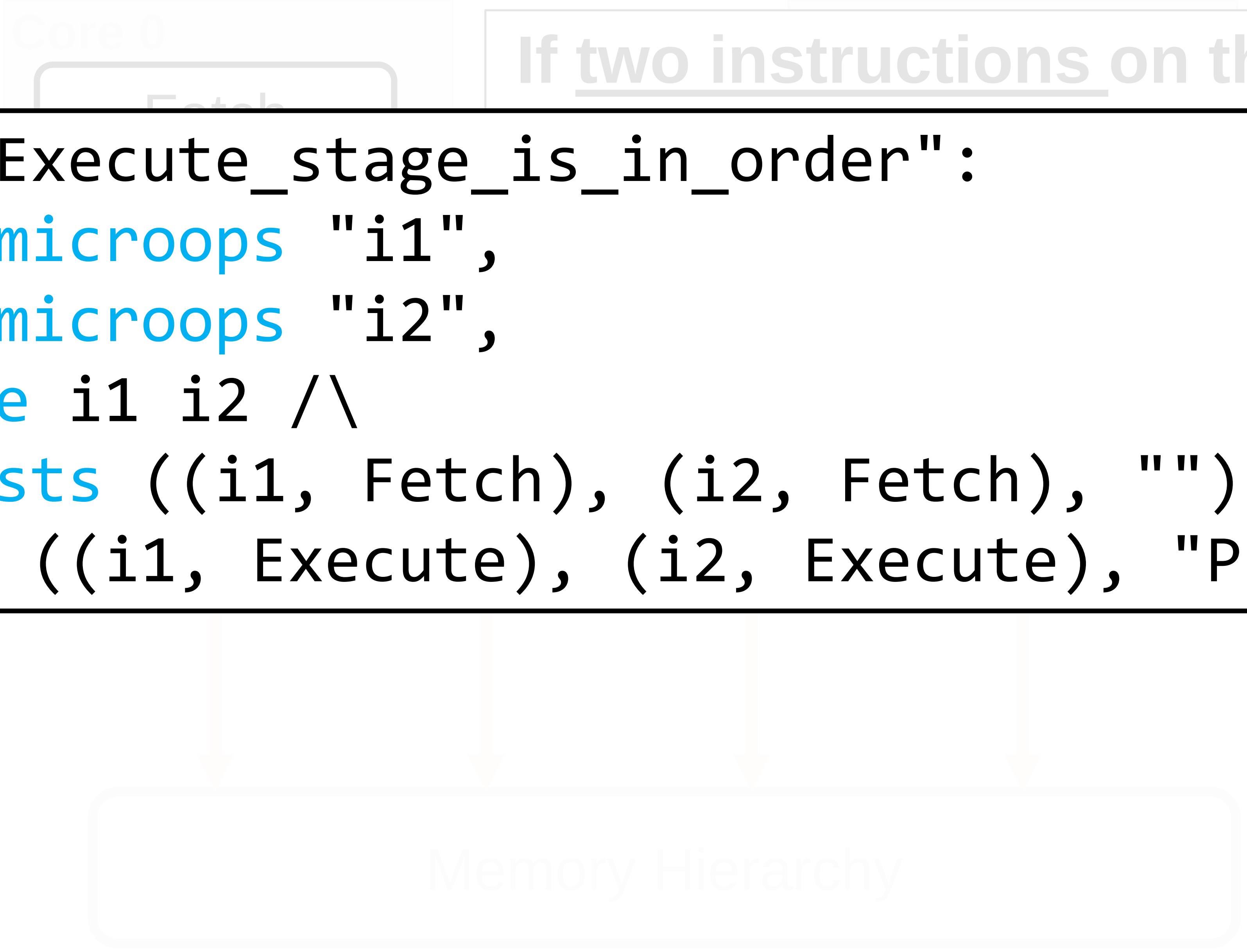


Finding Axioms



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</u>

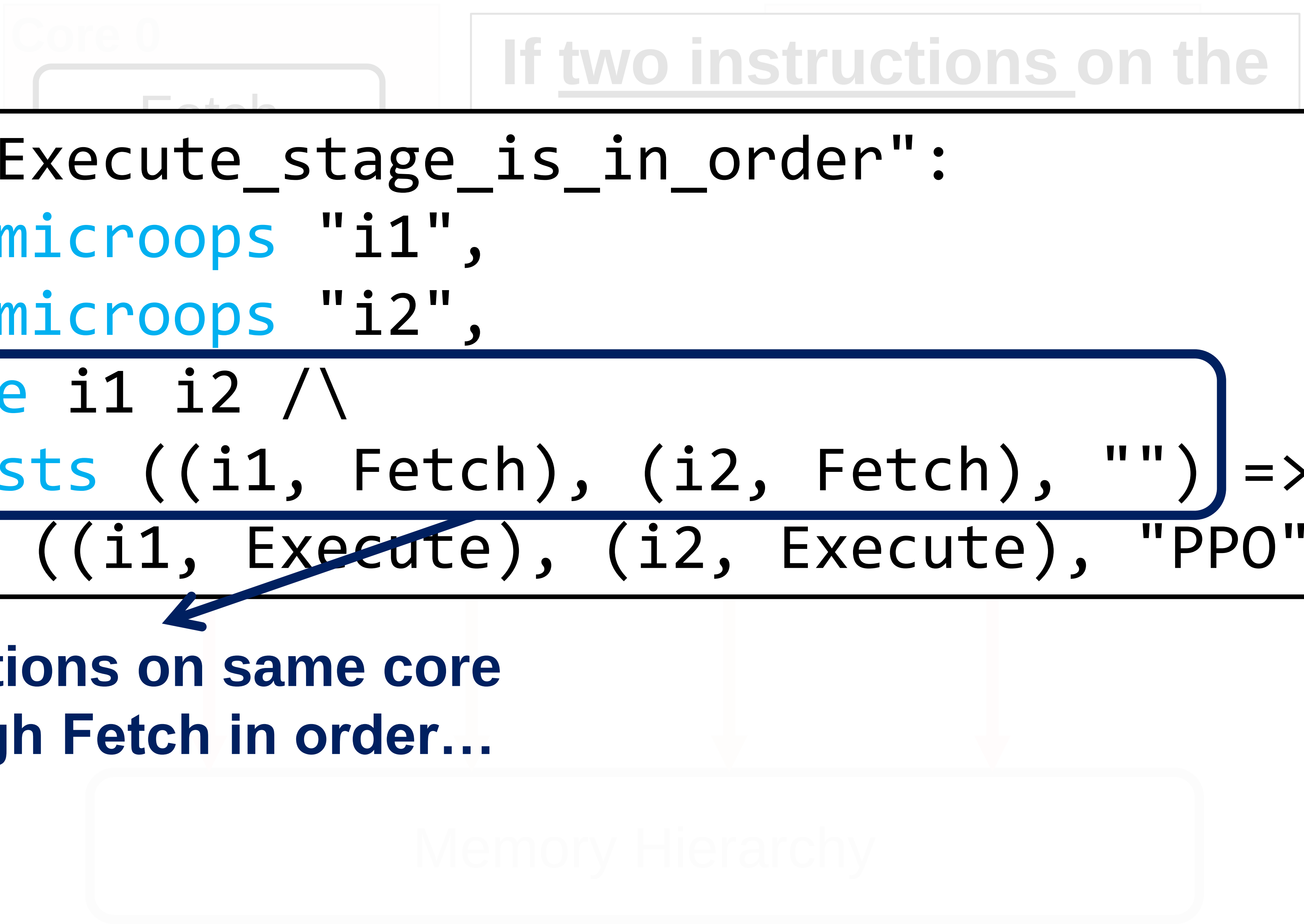
The Execute_Stage_Is_In_order Axiom



If two instructions on the

```
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



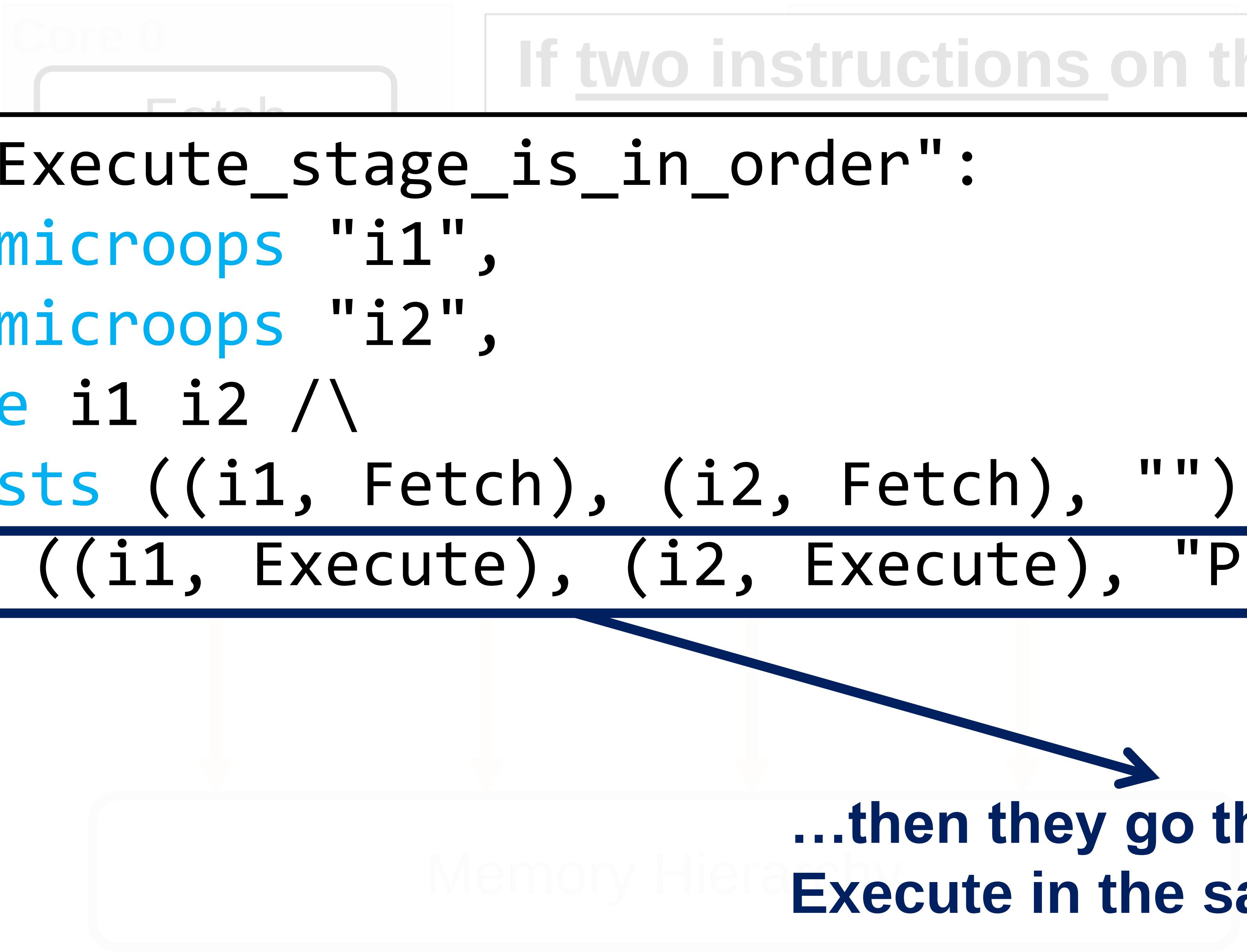
If two instructions on the

```
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 on same core go through Fetch in order…

The Execute_Stage_Is_In_order Axiom

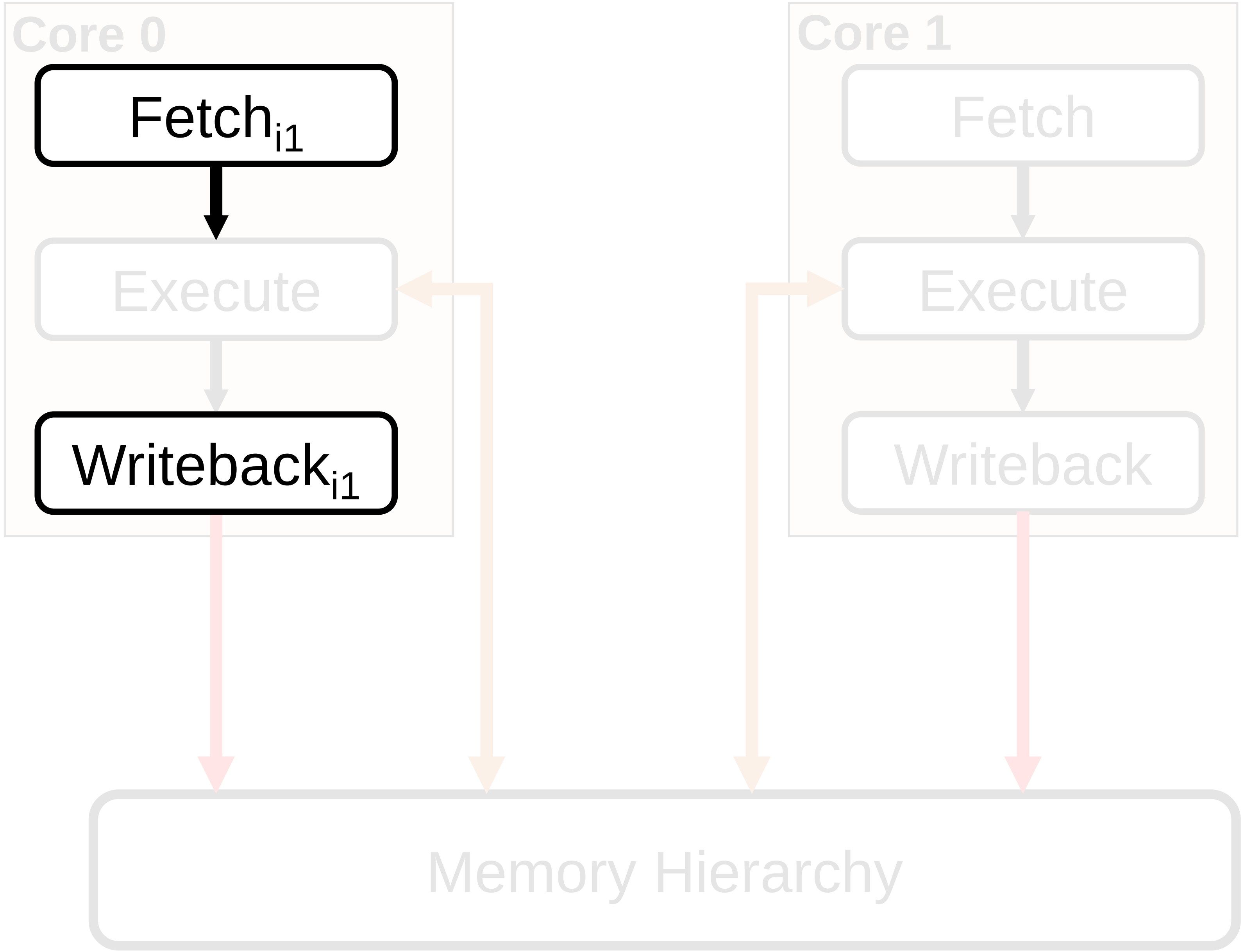
If two instructions on the



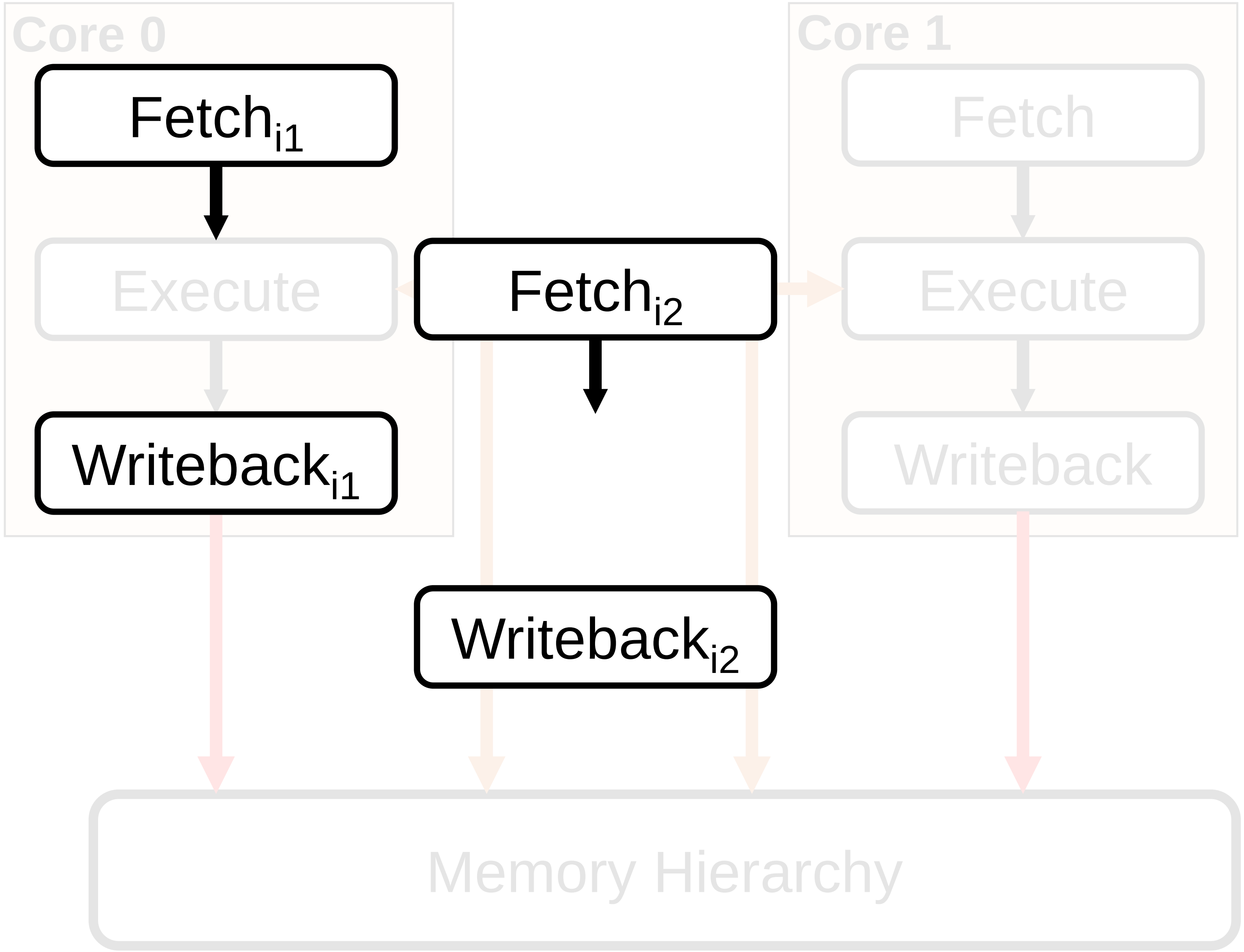
```
Axiom "Execute_stage_is_in_order":
forall microops "i1",
forall microops "i2",
SameCore i1 i2 /\
EdgeExists ((i1, Fetch), (i2, Fetch), "") =>
AddEdge ((i1, Execute), (i2, Execute), "PPO").
```

…then they go through Execute in the same order.

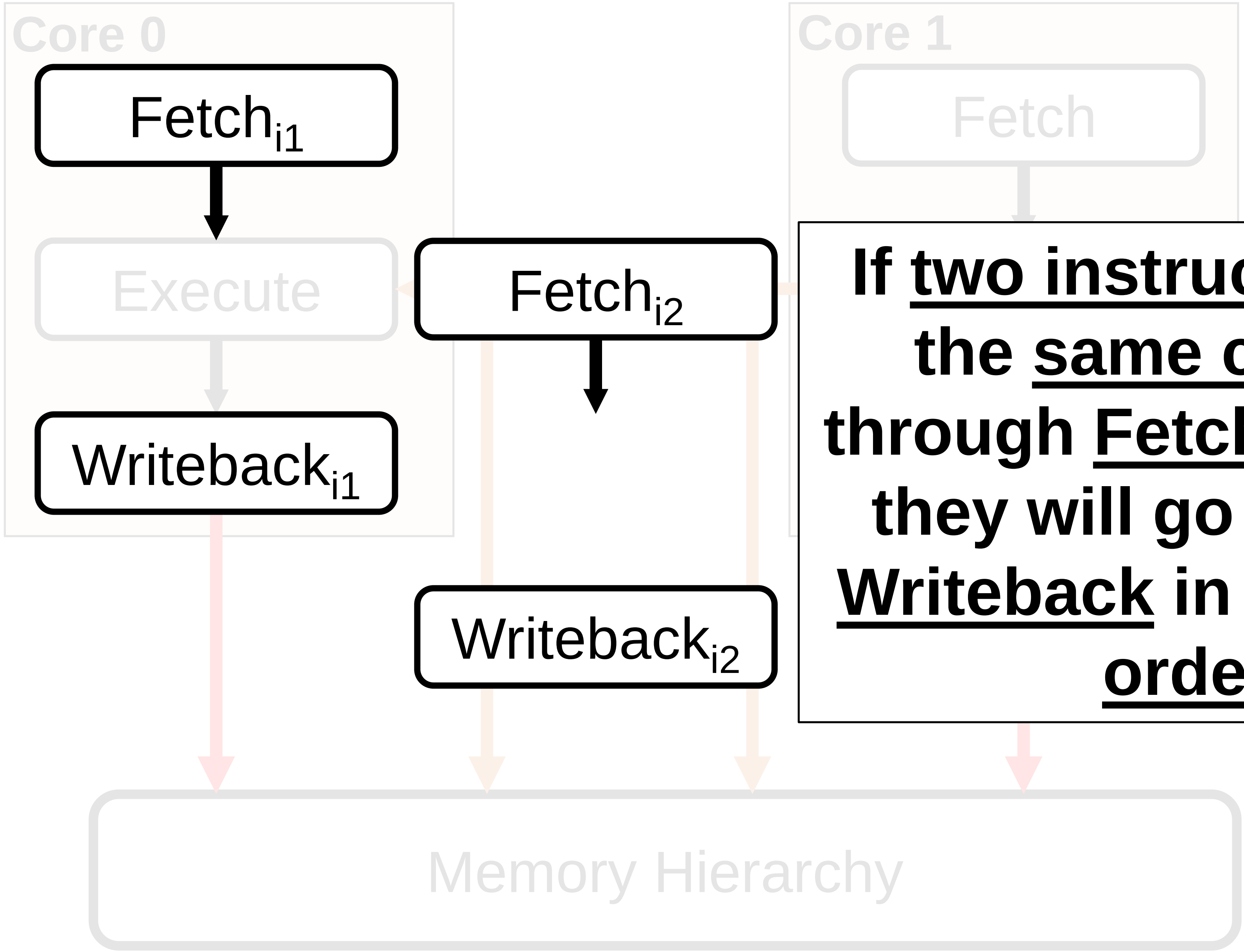
Finding Axioms



Finding Axioms



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>Writeback</u> in the <u>*same* order</u>

The Writeback_Stage_Is_In_Order Axiom

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

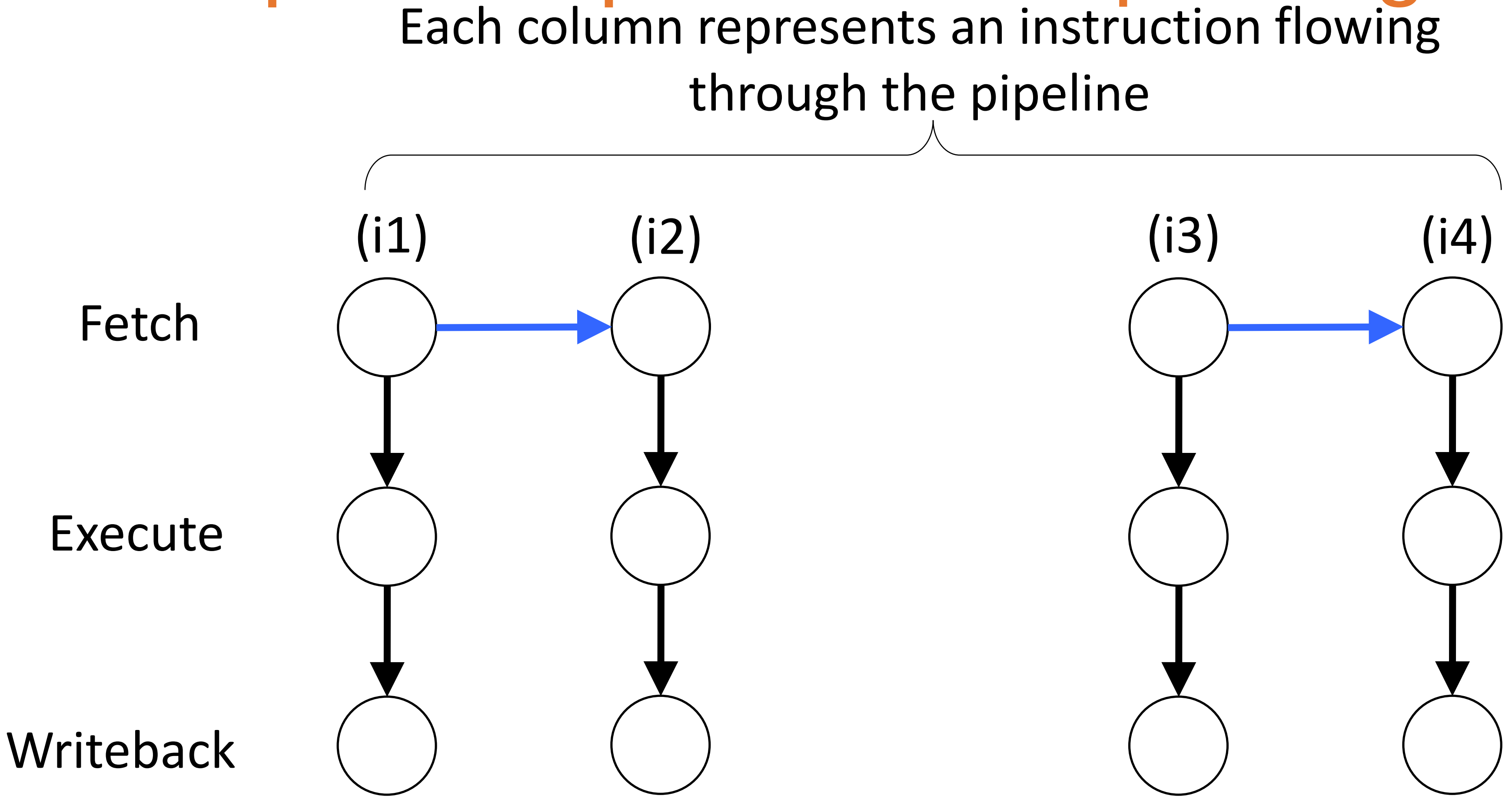
```
Axiom "Writeback_stage_is_in_order":
forall microops "i1",
forall microops "i2",
________ i1 i2 /\
EdgeExists ((i1, _____), (i2, _____), "") =>
AddEdge ((i1, _________), (i2, _________), "PPO").
```

The Writeback_Stage_Is_In_Order Axiom

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
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback

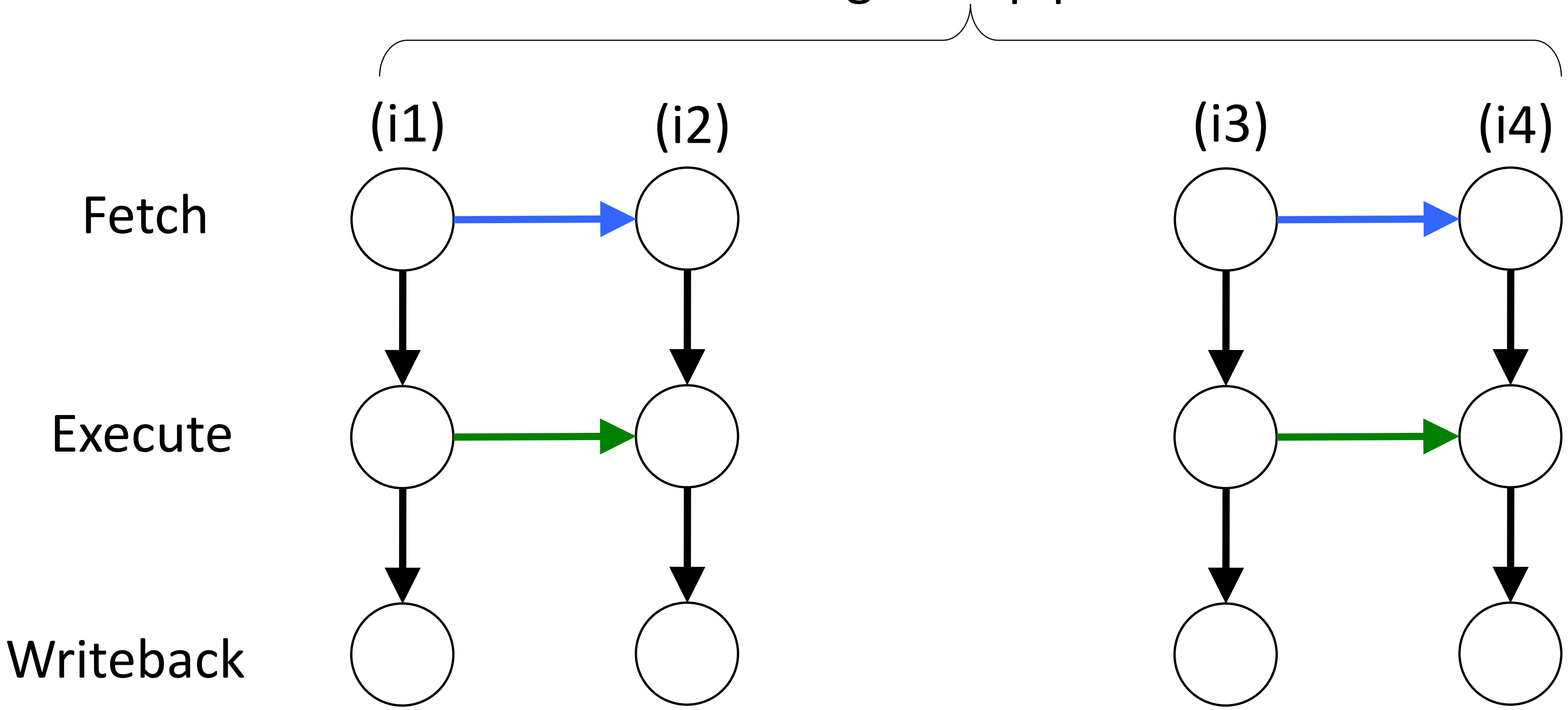
Initially, Mem[x] = 0

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

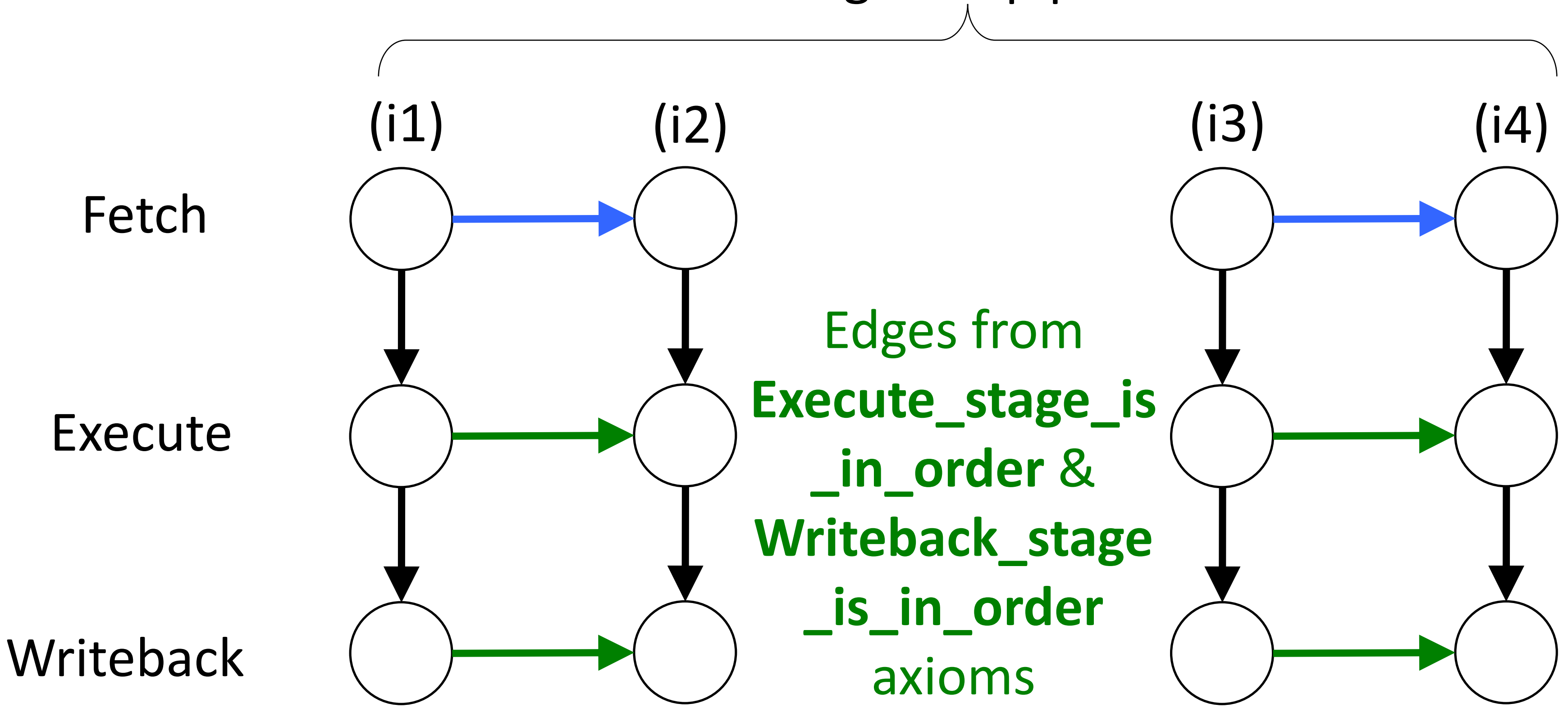
Each column represents an instruction flowing through the pipeline



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	

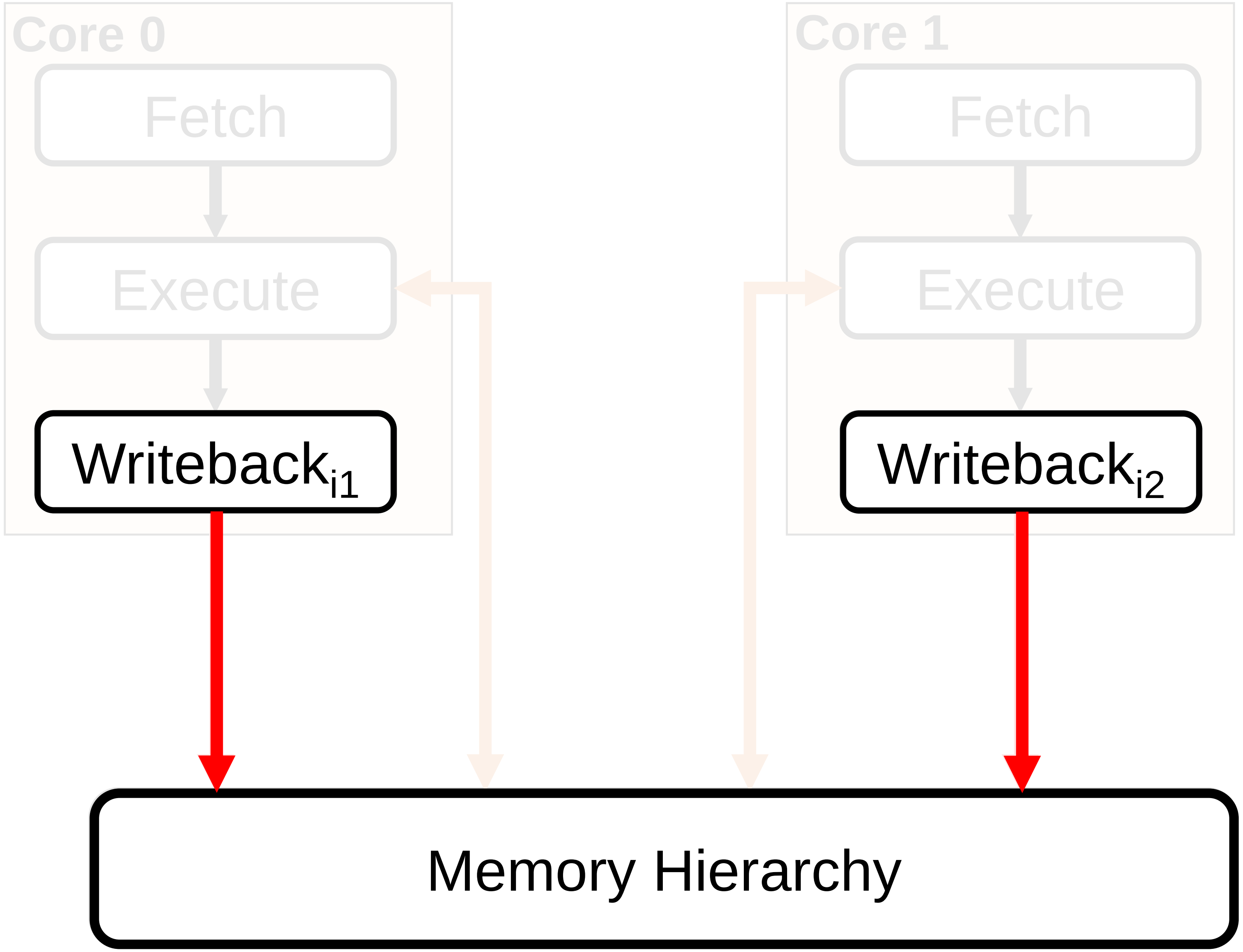
μhb Graphs for `co-mp` Using Axioms


Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback
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	

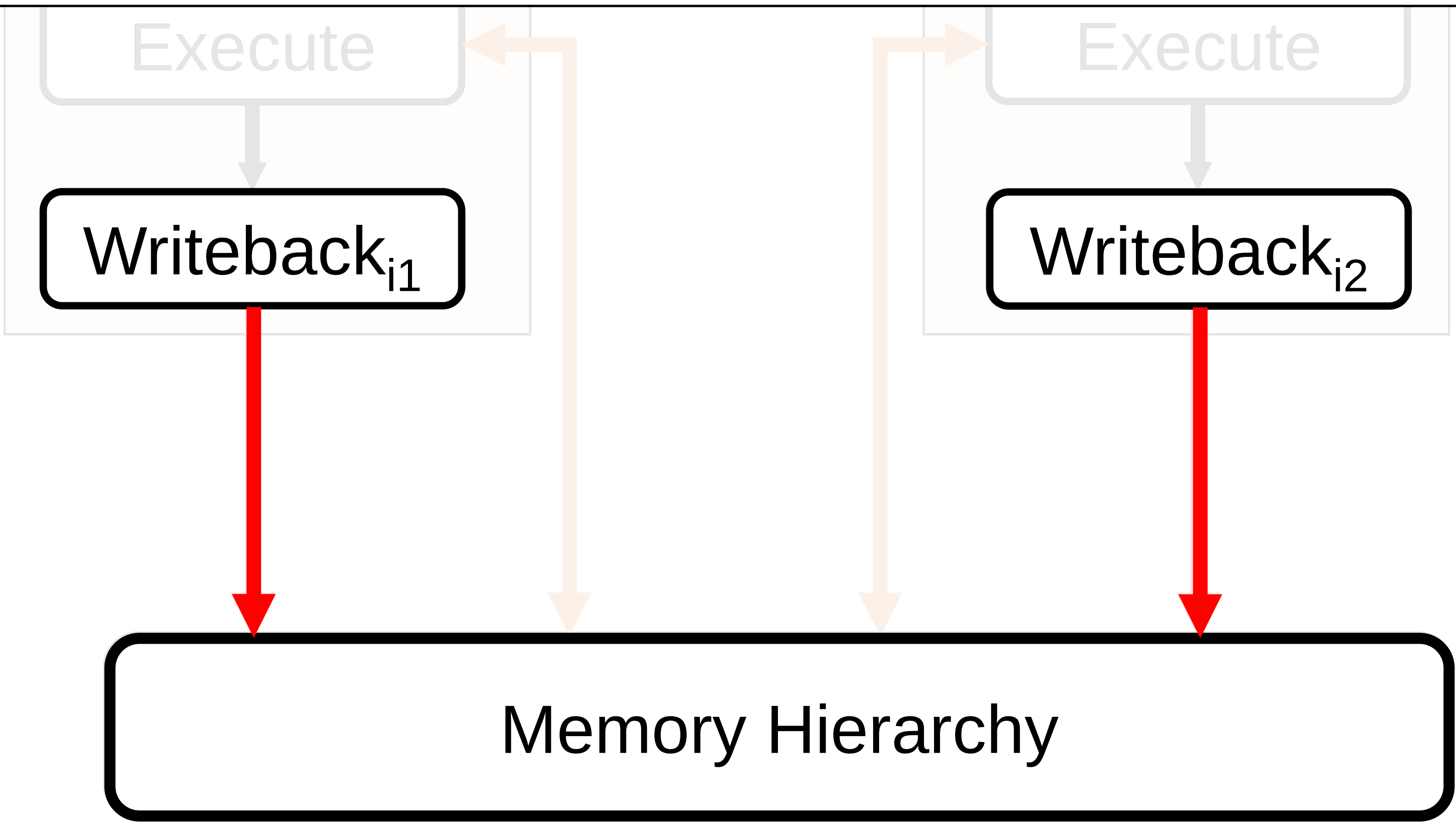
Finding Axioms



Finding Axioms

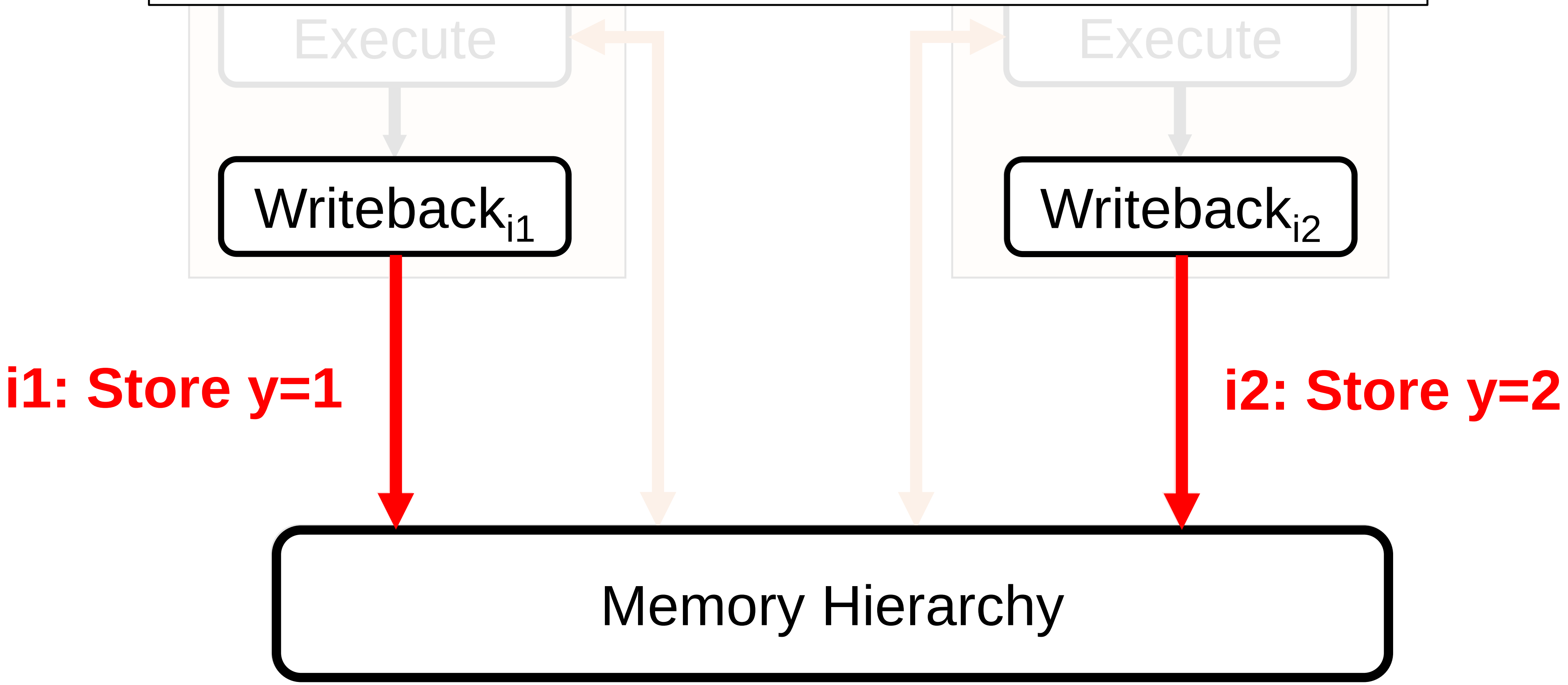
All <u>writes</u> to the <u>same address</u> must be <u>totally ordered</u> at memory.

(coherence order)



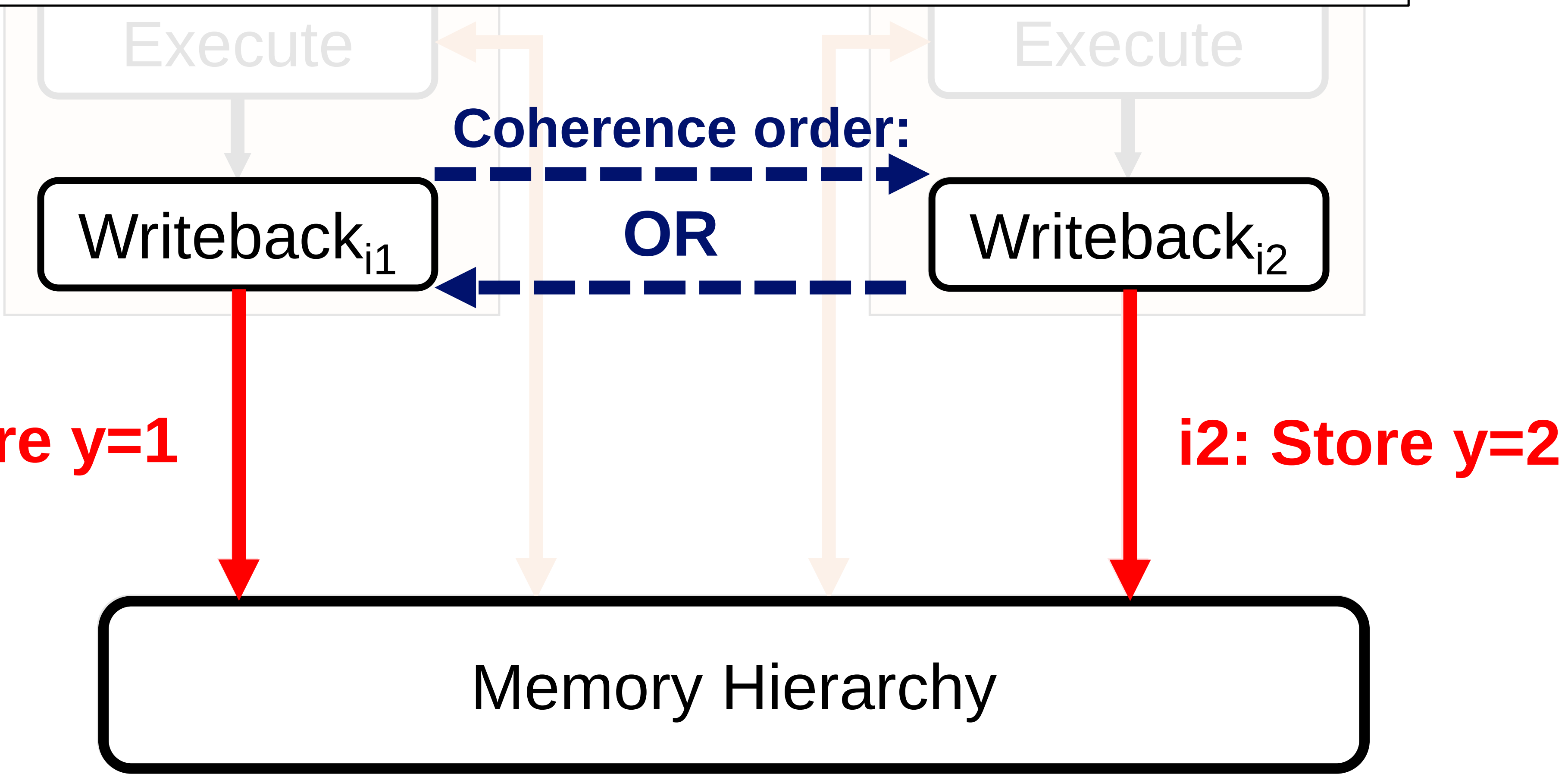
Finding Axioms

All <u>writes</u> to the <u>same address</u> must be <u>totally ordered</u> at memory.
(coherence order)

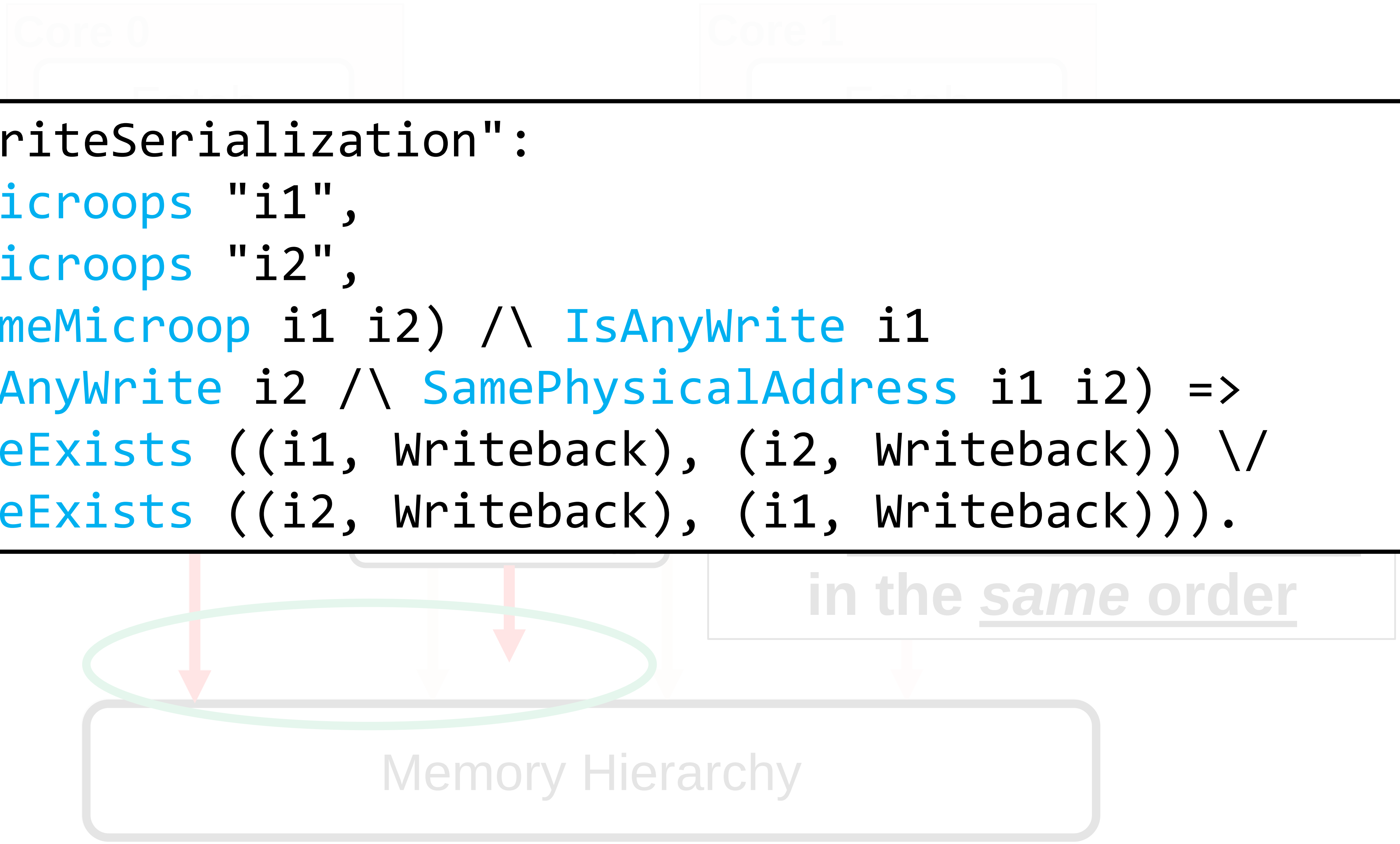


Finding Axioms

All writes to the same address must be totally ordered at memory.
(coherence order)



The WriteSerialization Axiom

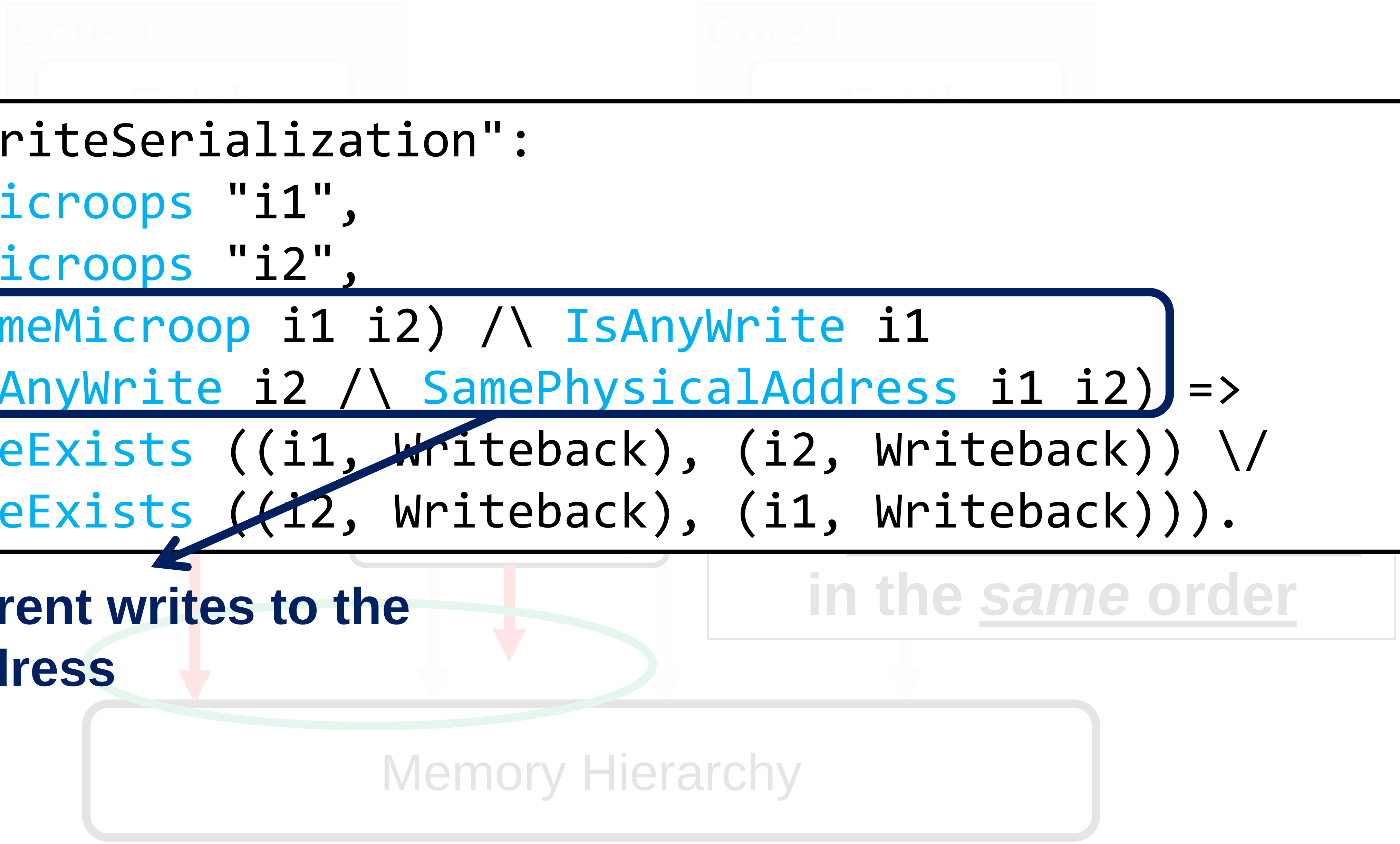


```
Axiom "WriteSerialization":
forall microops "i1",
forall microops "i2",
  ( ~(SameMicroop i1 i2) /\ IsAnyWrite i1
   /\ IsAnyWrite i2 /\ SamePhysicalAddress i1 i2) =>
    (EdgeExists ((i1, Writeback), (i2, Writeback)) \/
     EdgeExists ((i2, Writeback), (i1, Writeback))).
```

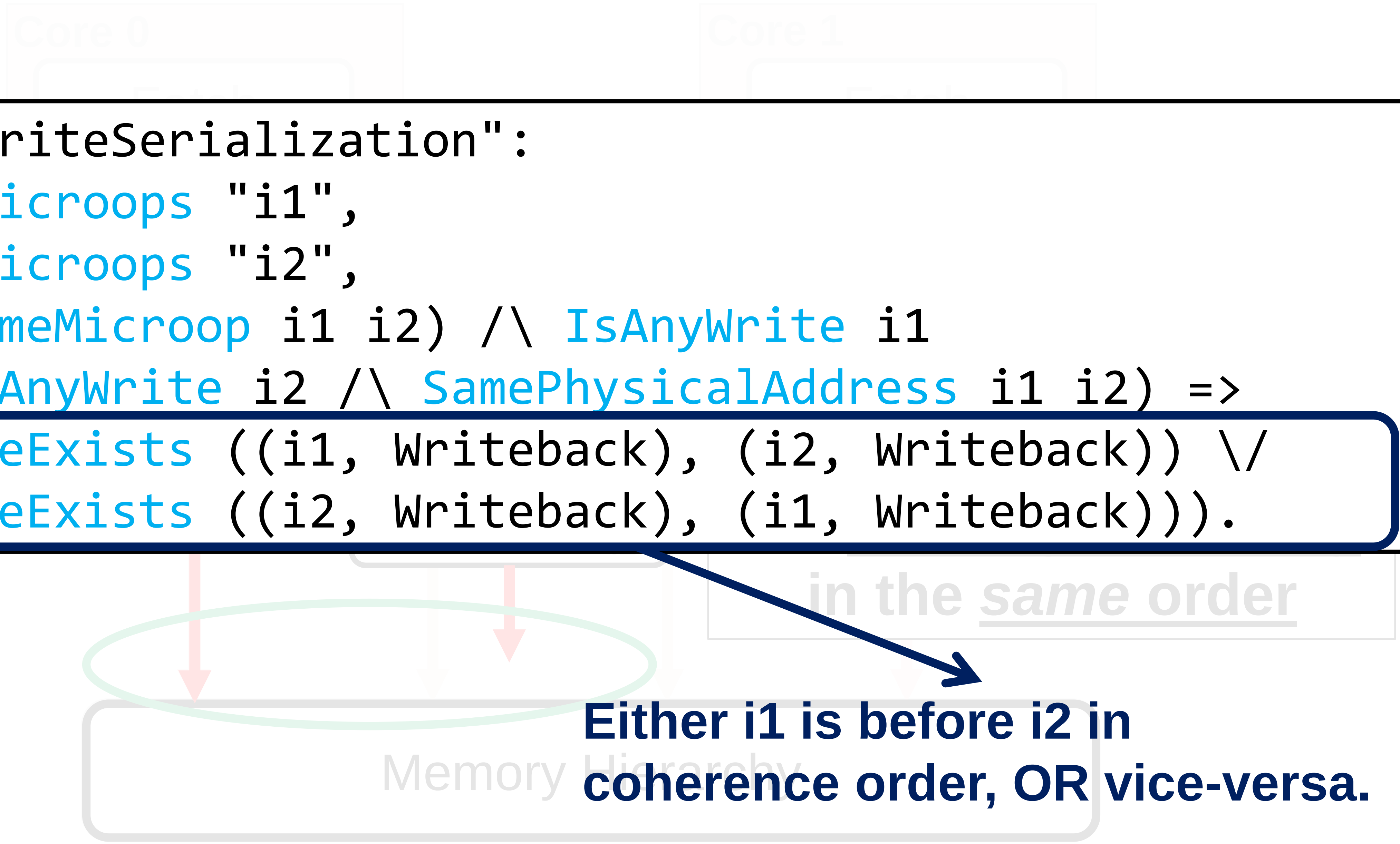
The WriteSerialization Axiom

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

Two different writes to the same address



The WriteSerialization Axiom



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

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

μhb Graphs for `co-mp` Using Axioms

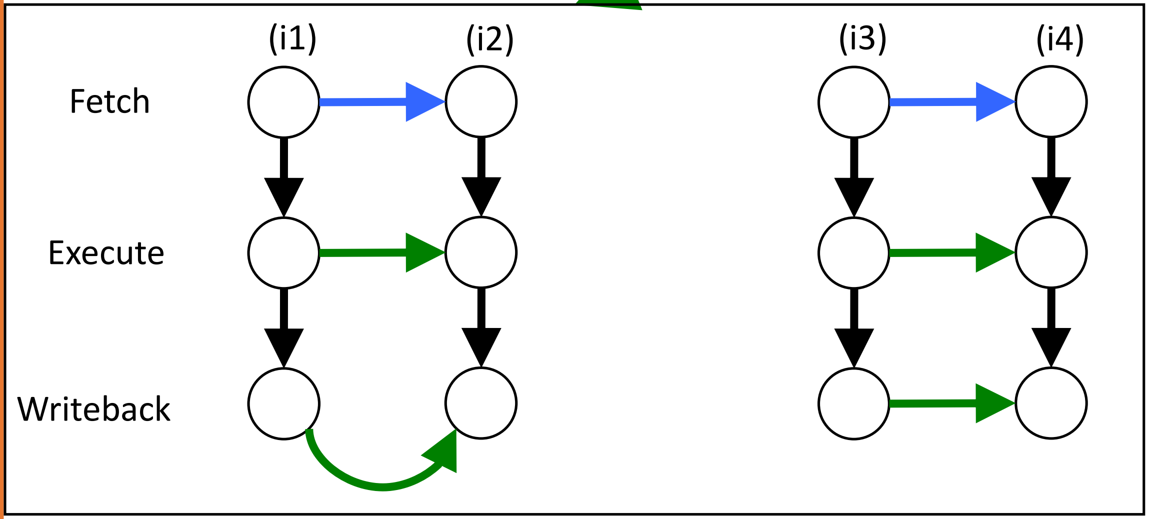
WriteSerialization axiom

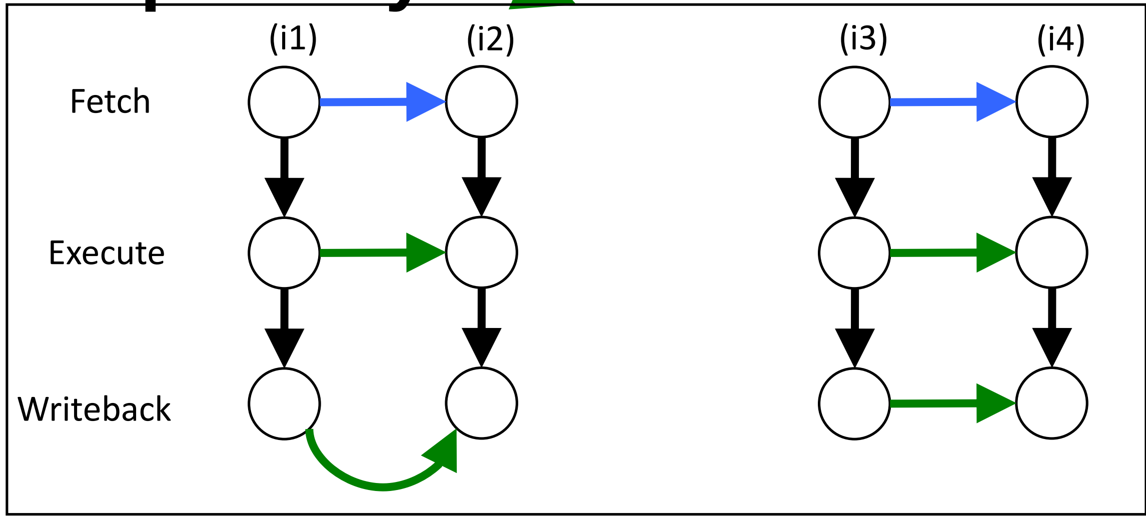
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

µhb Graphs for `co-mp` Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated **separately**



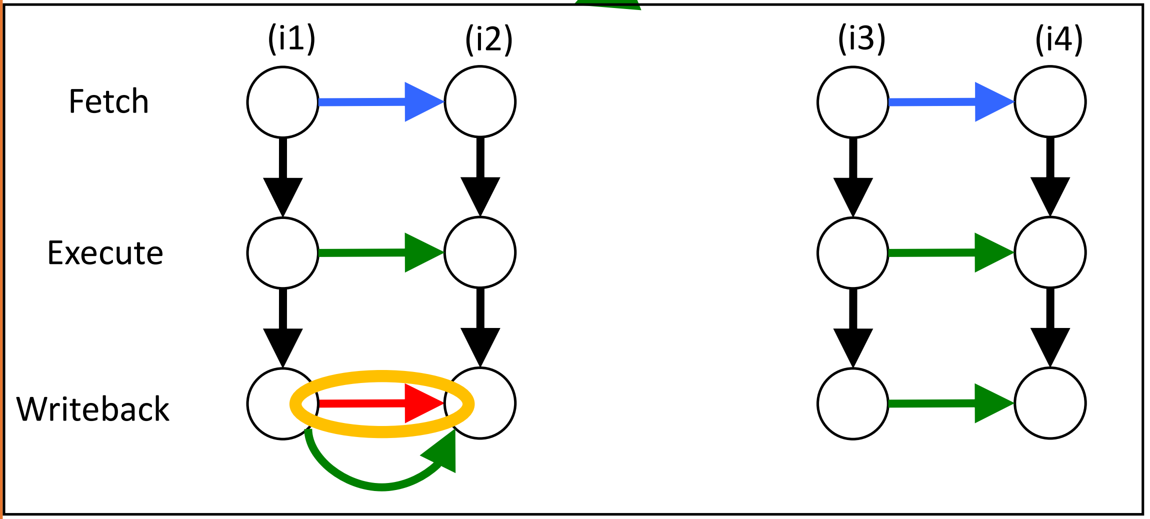


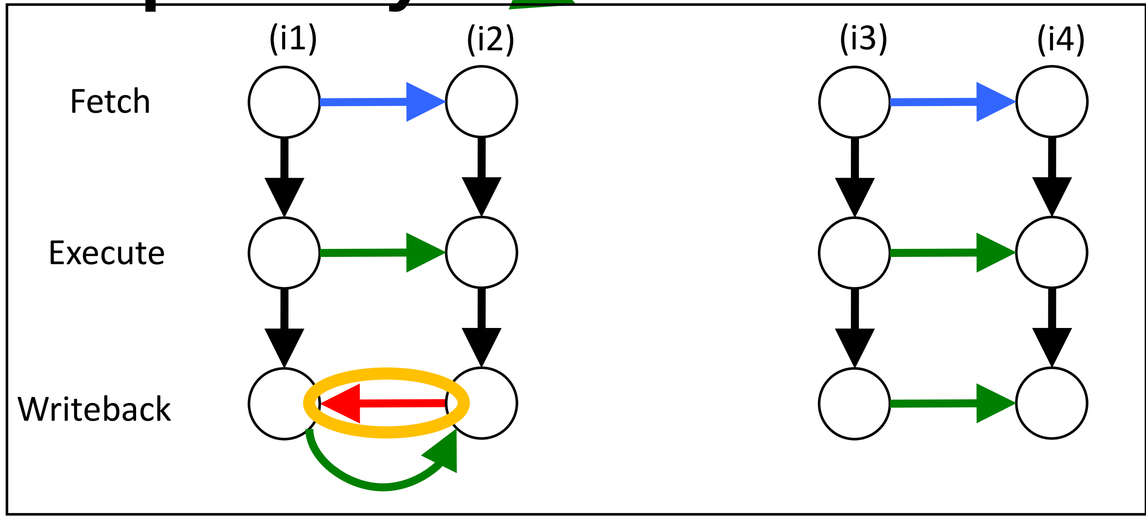
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

μhb Graphs for `co-mp` Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated **separately**



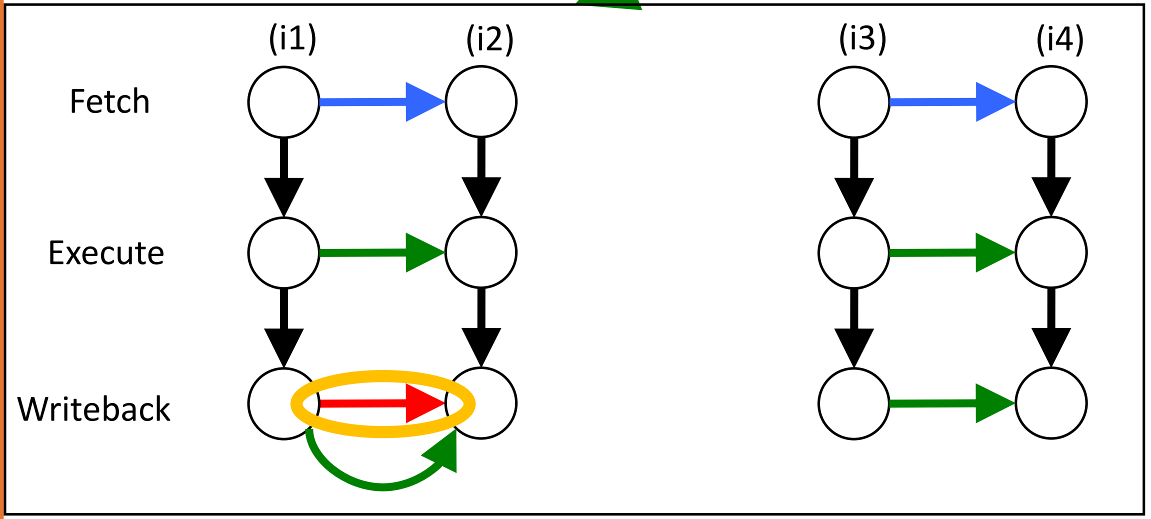


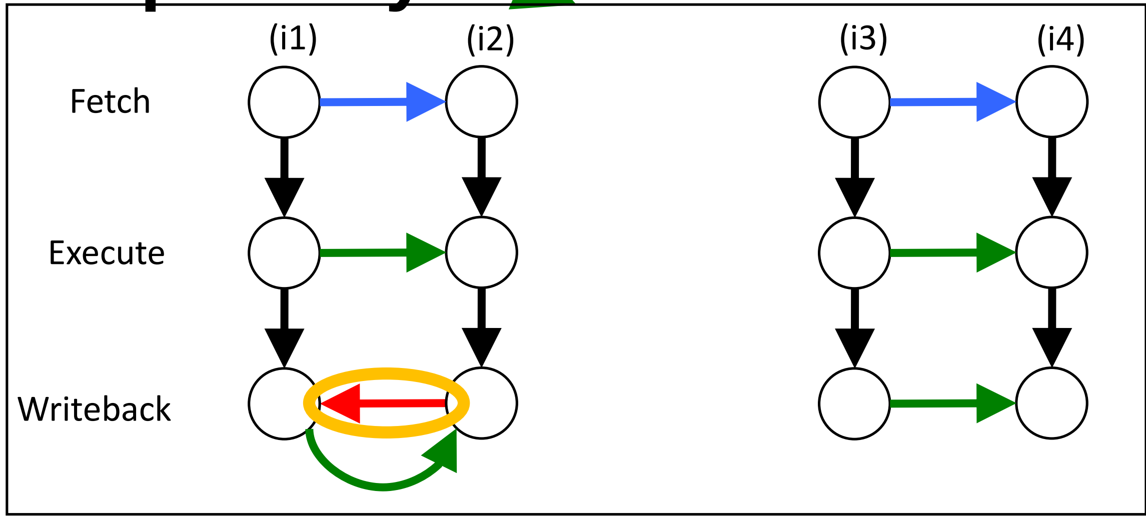
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC Forbids: r1=2, r2=1, Mem[x] = 2	

μhb Graphs for `co-mp` Using Axioms

WriteSerialization axiom

Two solutions;
Each enumerated **separately**

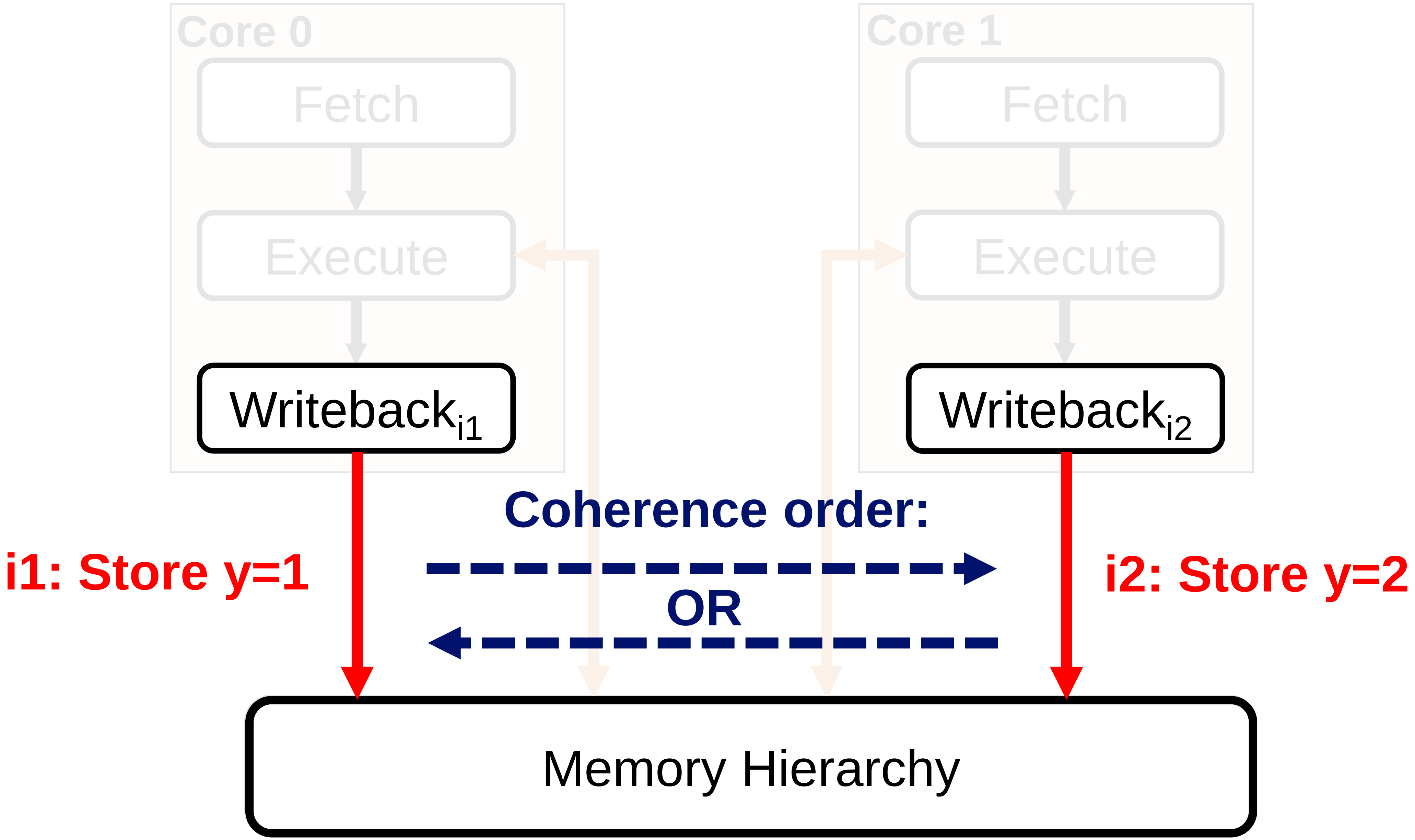




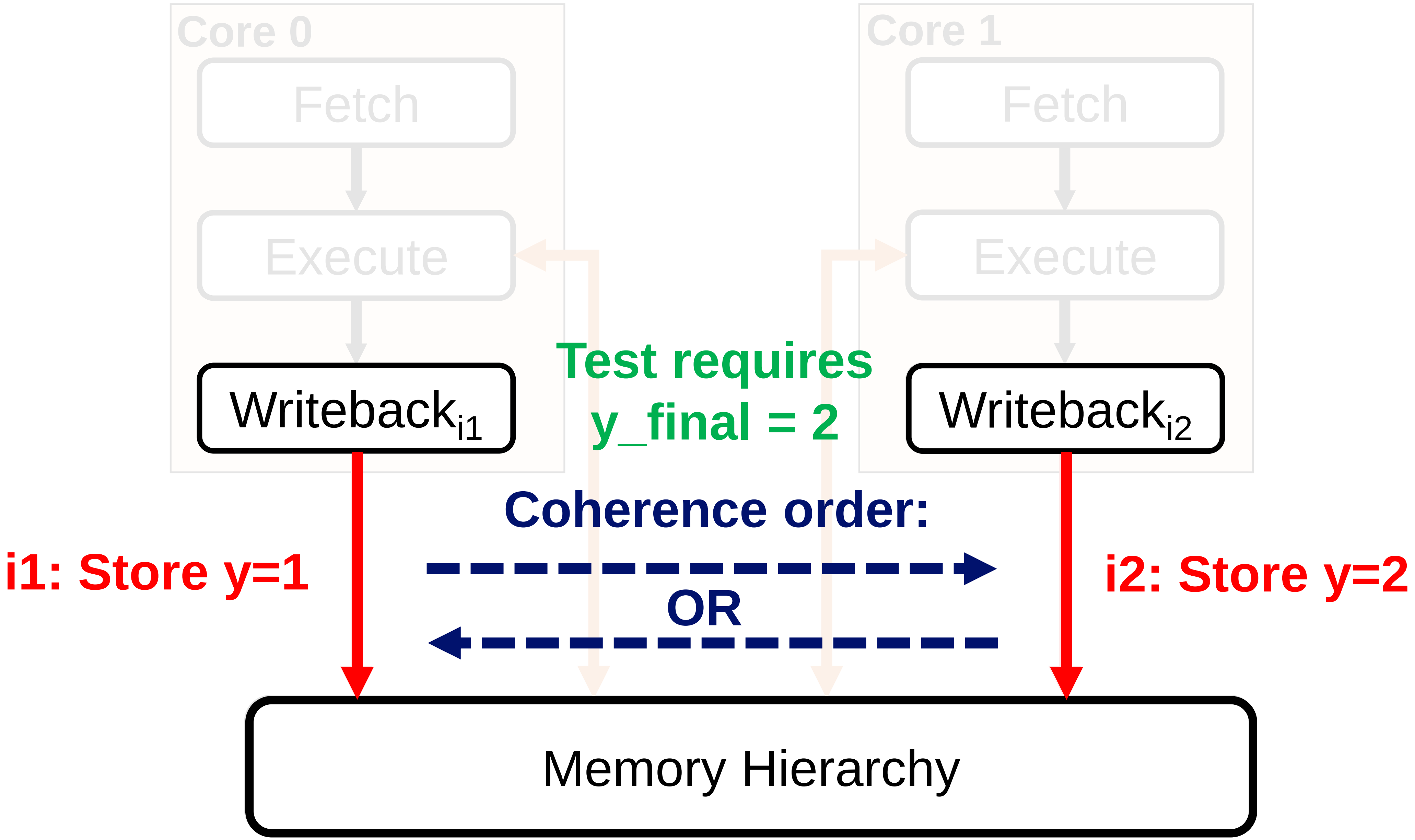
We will focus on left graph going forward

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

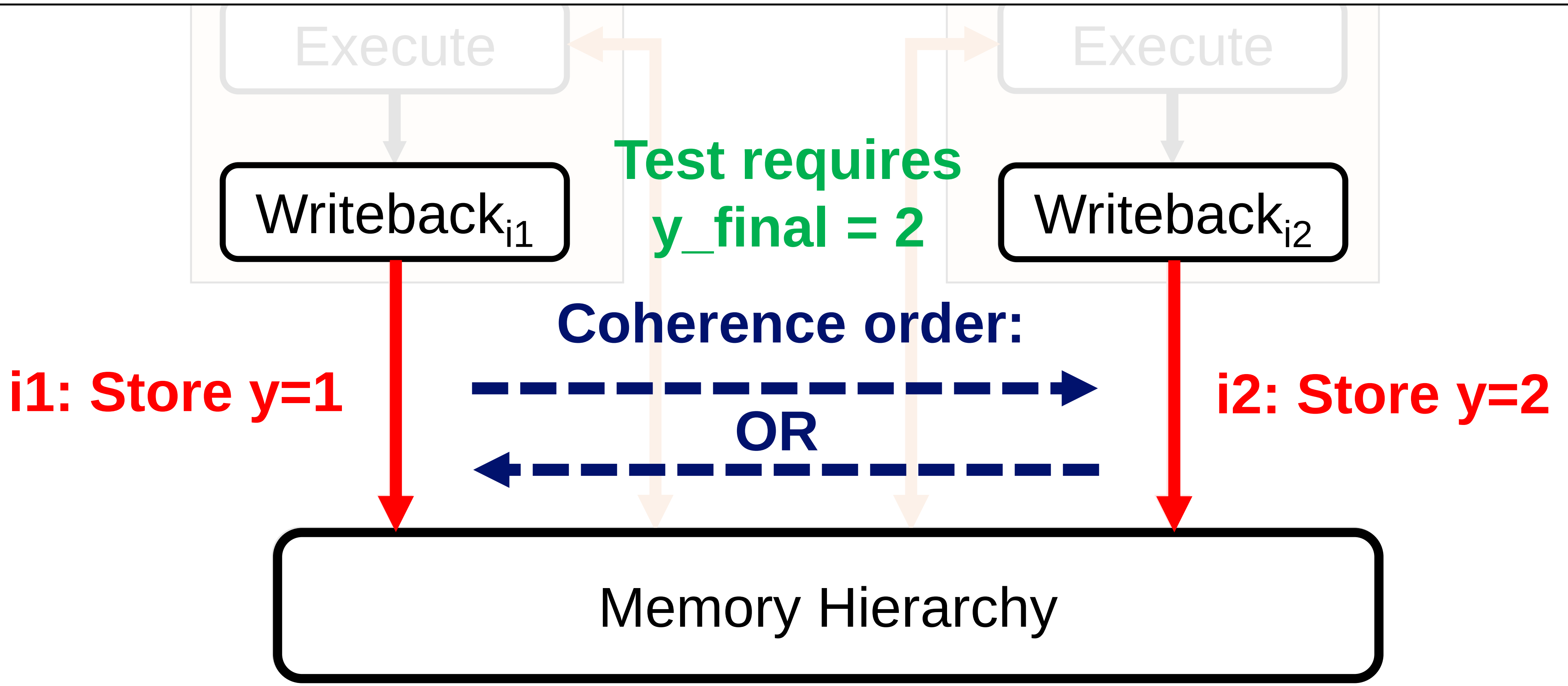


Finding Axioms



Finding Axioms

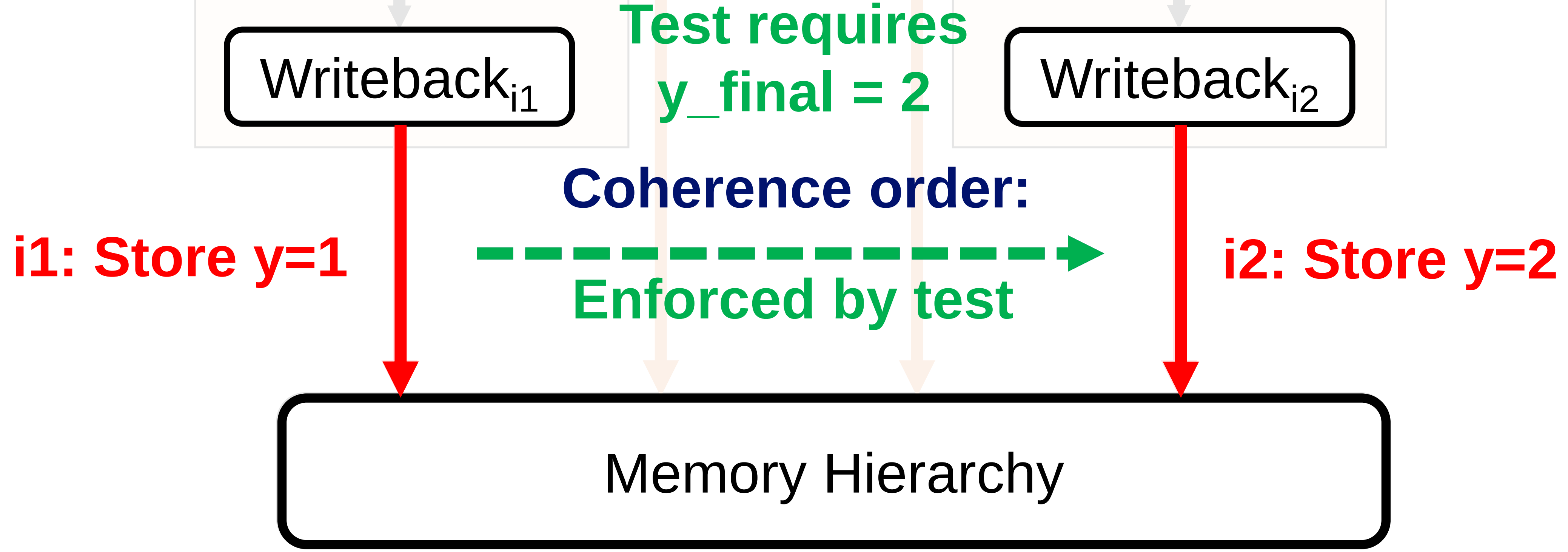
If a <u>litmus test requires</u> that an <u>address</u> has the <u>value of a certain write</u> at the <u>end of the test</u>, <u>that write</u> must be the <u>last</u> to reach memory.



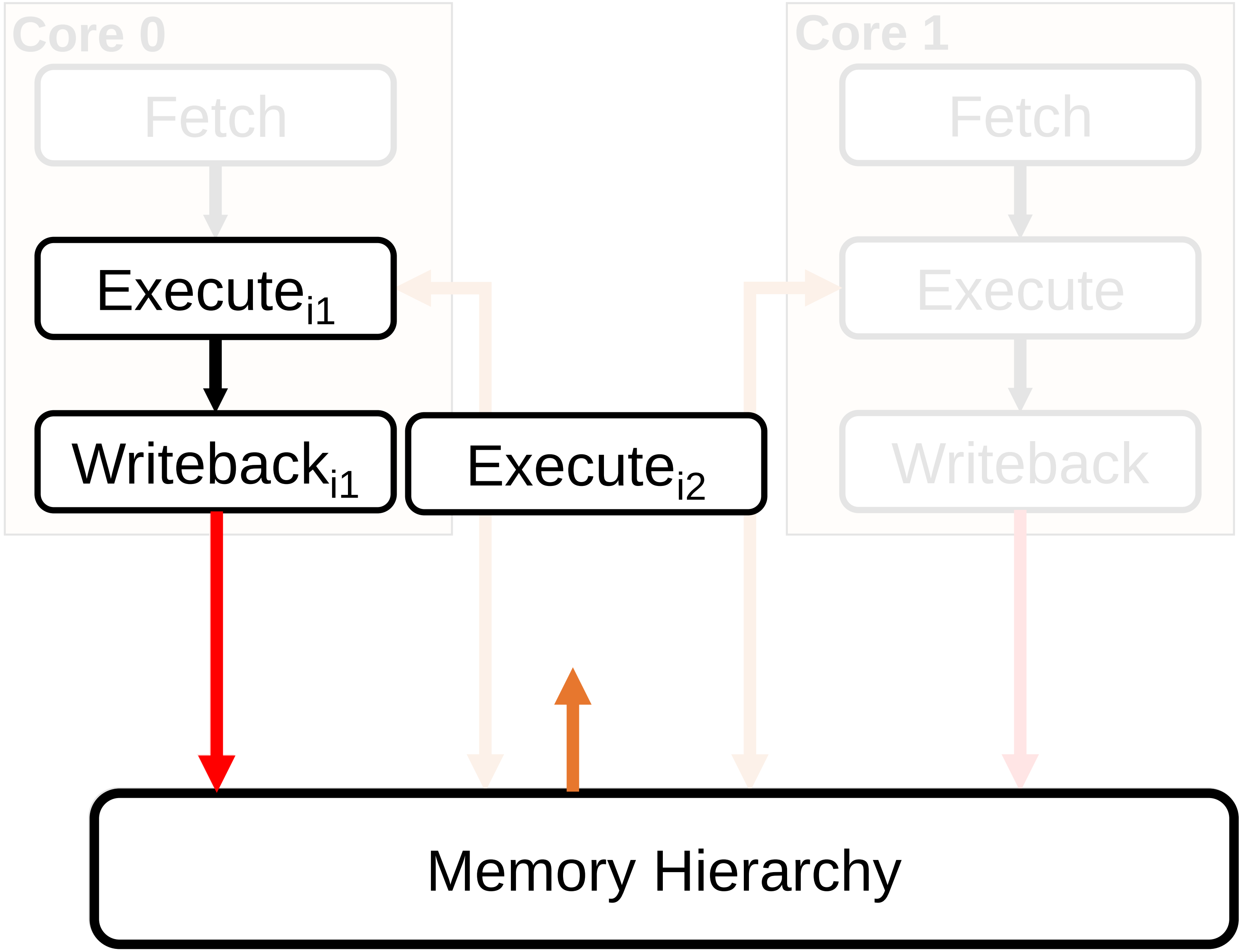
Finding Axioms

If a <u>litmus test requires</u> that an <u>address</u> has the <u>value of a certain write</u> at the <u>end of the test</u>, <u>that write</u> must be the <u>last</u> to reach memory.

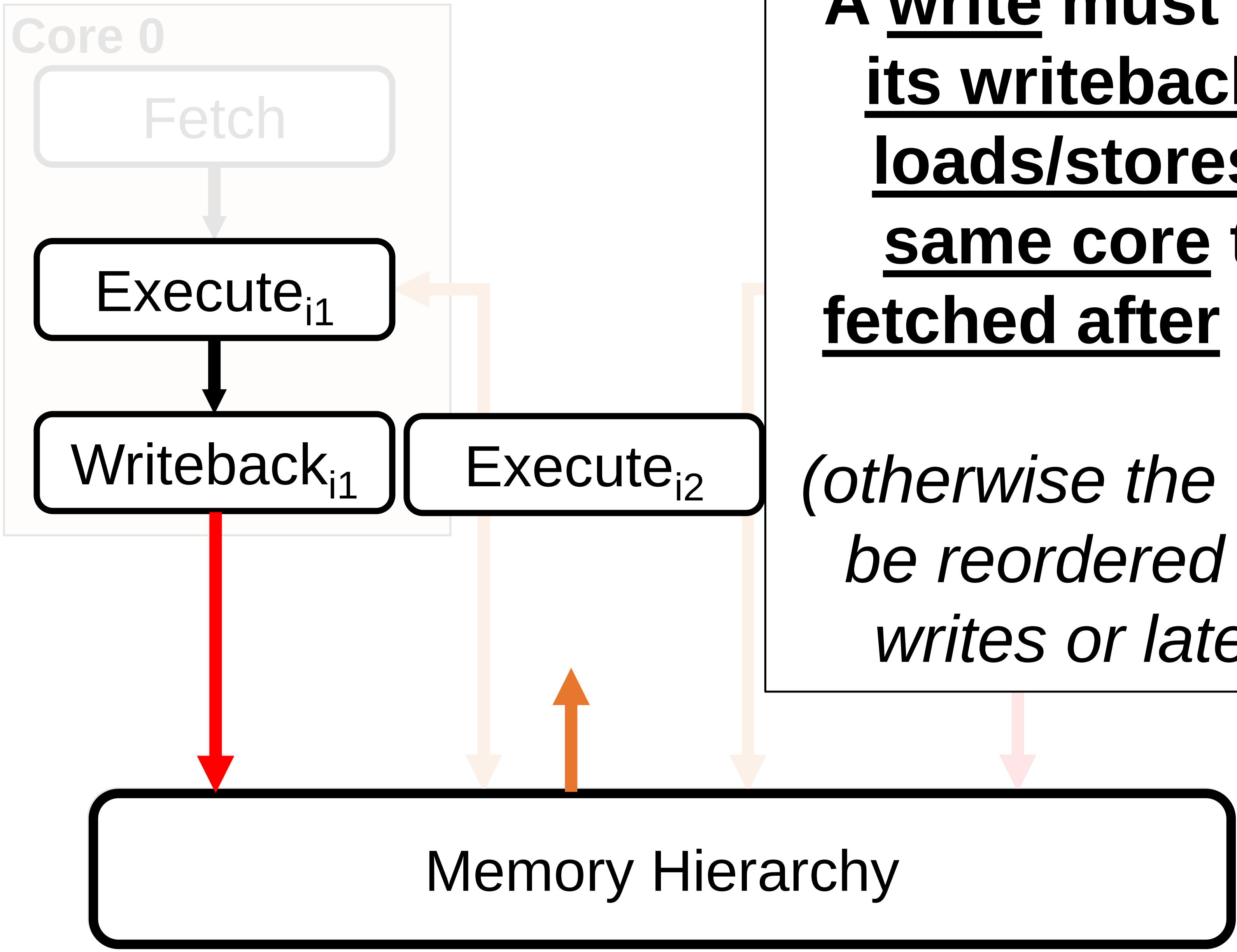
EnforceFinalWrite axiom in the µSpec



Finding Axioms



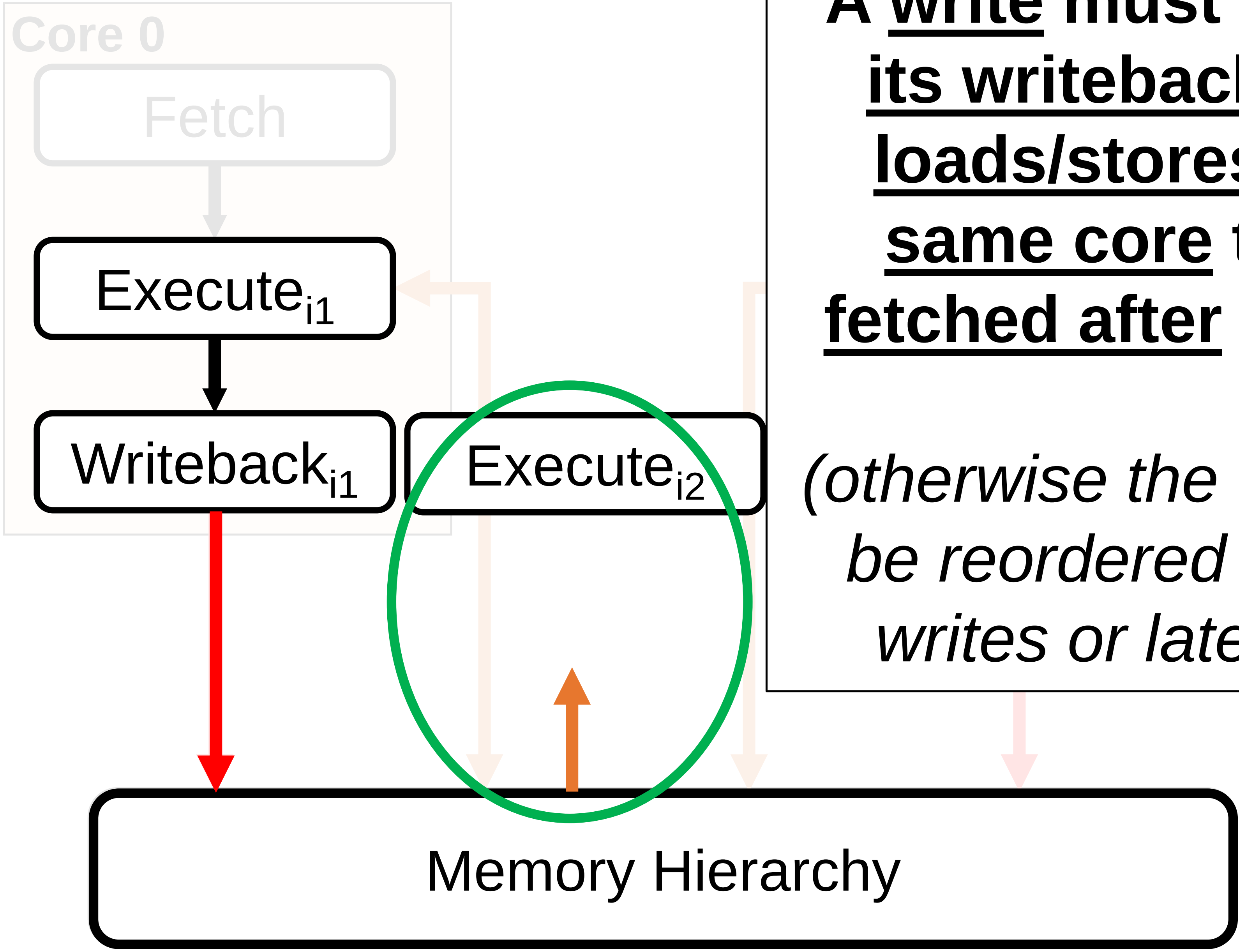
Finding Axioms



A <u>write</u> must <u>complete its writeback</u> before <u>loads/stores</u> on the <u>same core</u> that are <u>fetched after</u> the write.

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

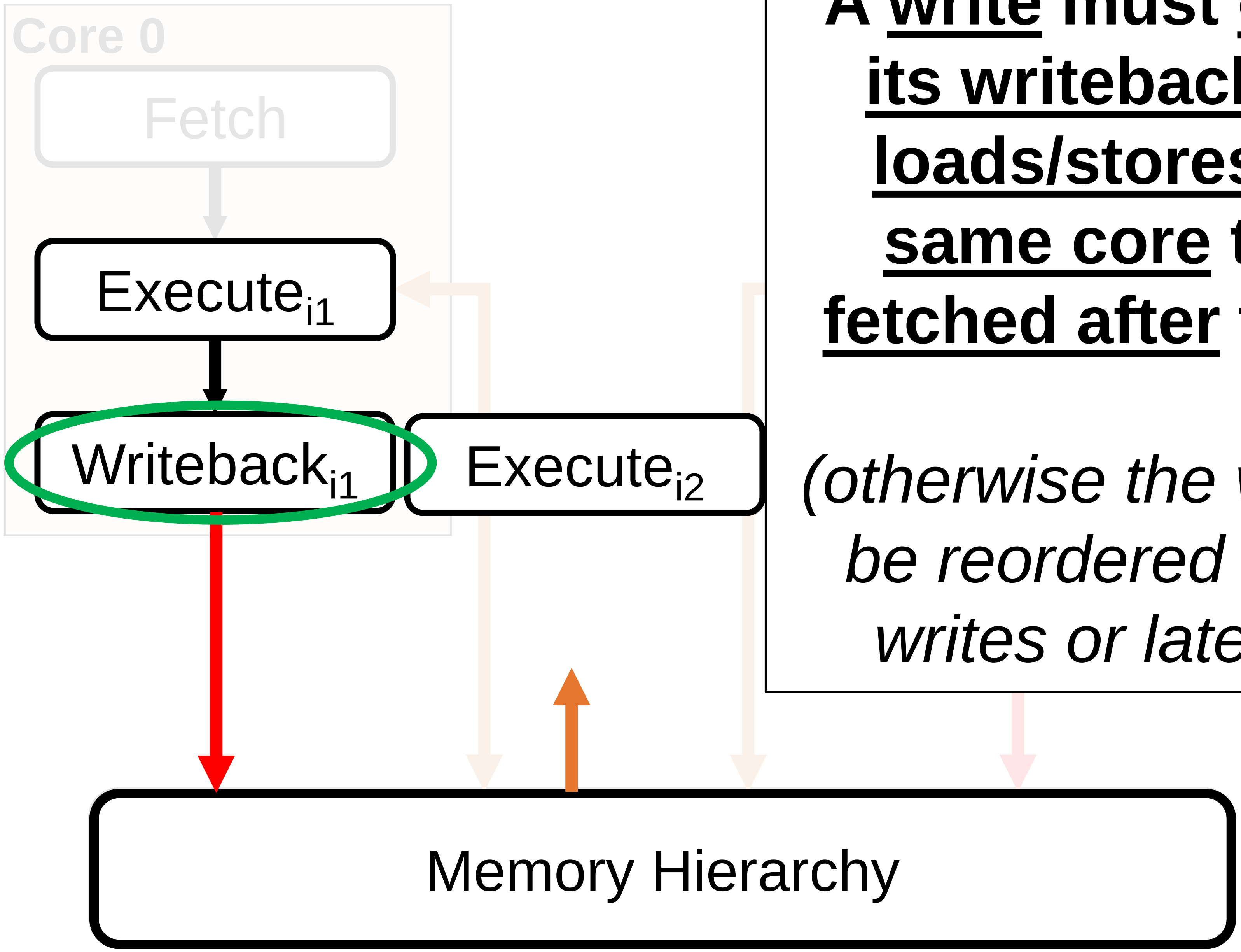
Finding Axioms



A <u>write</u> must <u>complete its writeback</u> before <u>loads/stores on the same core</u> that are <u>fetched after</u> the write.

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

Finding Axioms



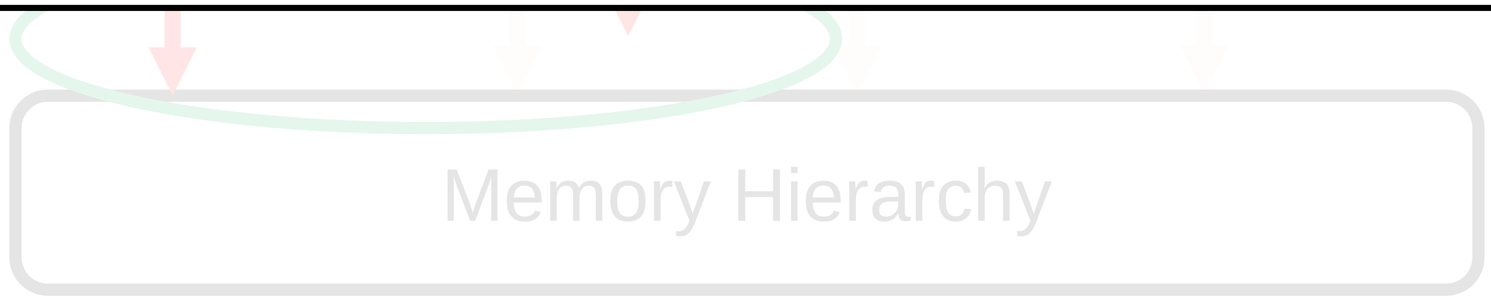
A <u>write</u> must <u>complete its writeback</u> before <u>loads/stores</u> on the <u>same core</u> that are <u>fetched after</u> the write.

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

The EnforceWritePPO Axiom

A <u>write</u> must <u>complete its writeback</u> before <u>execution</u> of <u>loads/stores</u> on the <u>same core</u> that are <u>fetched after</u> the write.

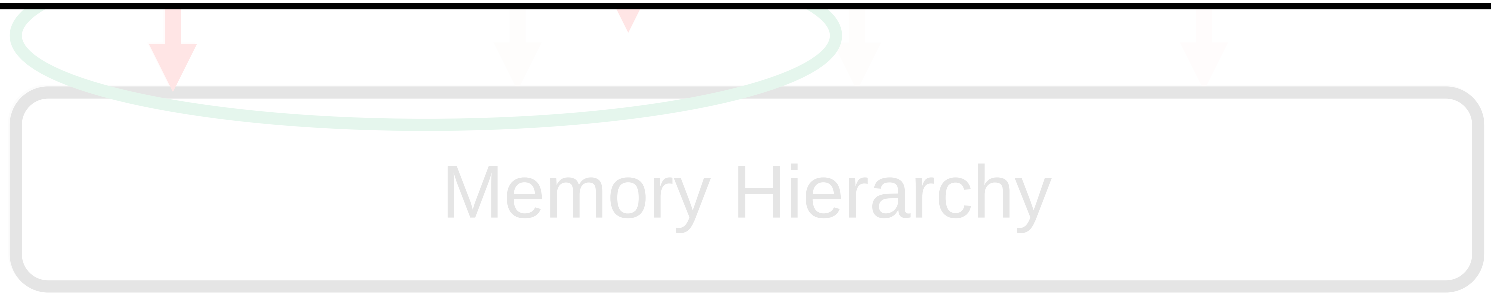
```
Axiom "EnforceWritePPO":
  forall microop "w",
  forall microop "i",
  (__________ w /\ ________ w i
   /\ EdgeExists((w, Fetch), (i, Fetch), "")) =>
          AddEdge ((w, _________), (i, _______)).
```



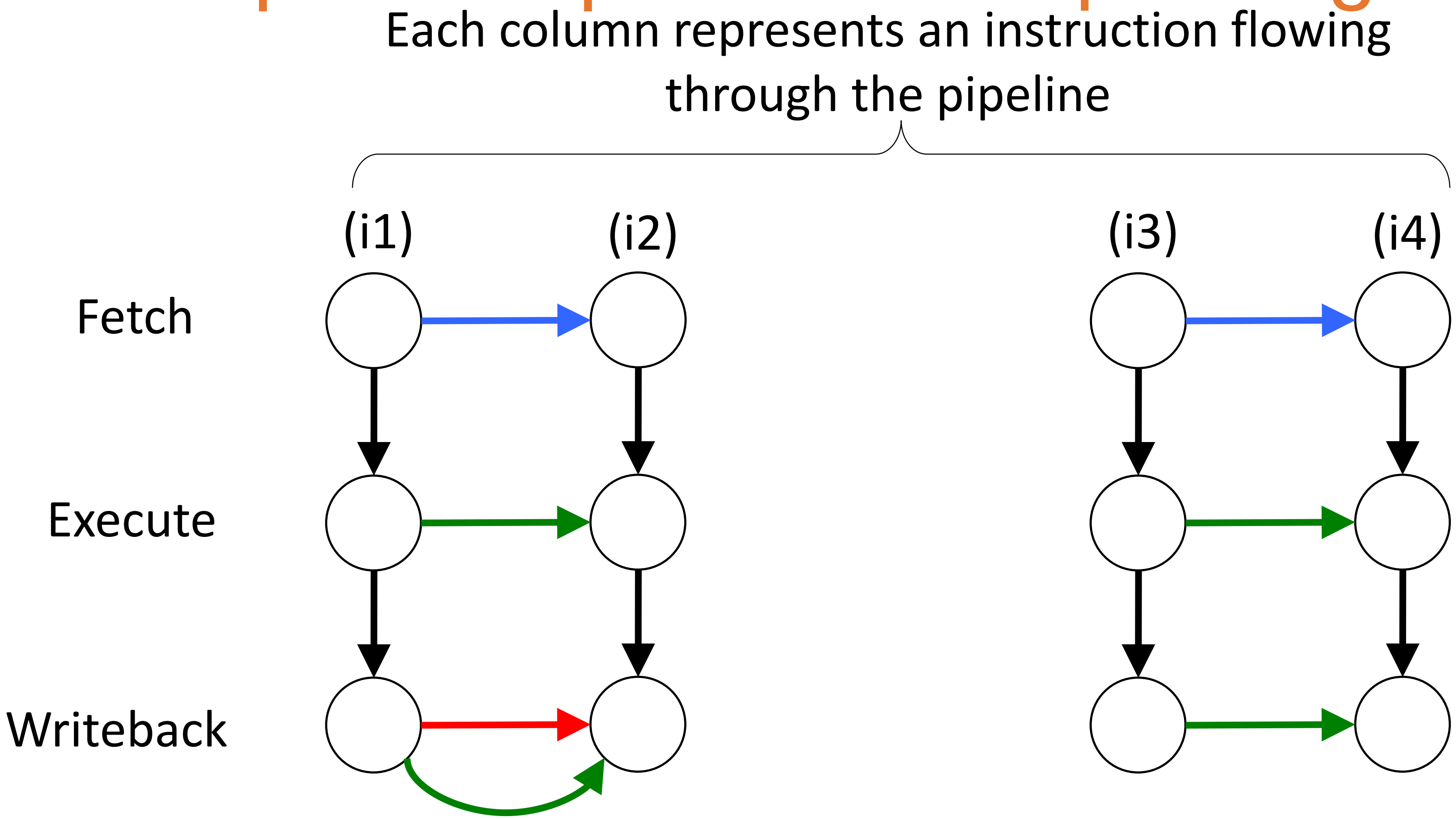
The EnforceWritePPO Axiom

A write must complete its writeback before execution of loads/stores on the same core that are fetched after the write.

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



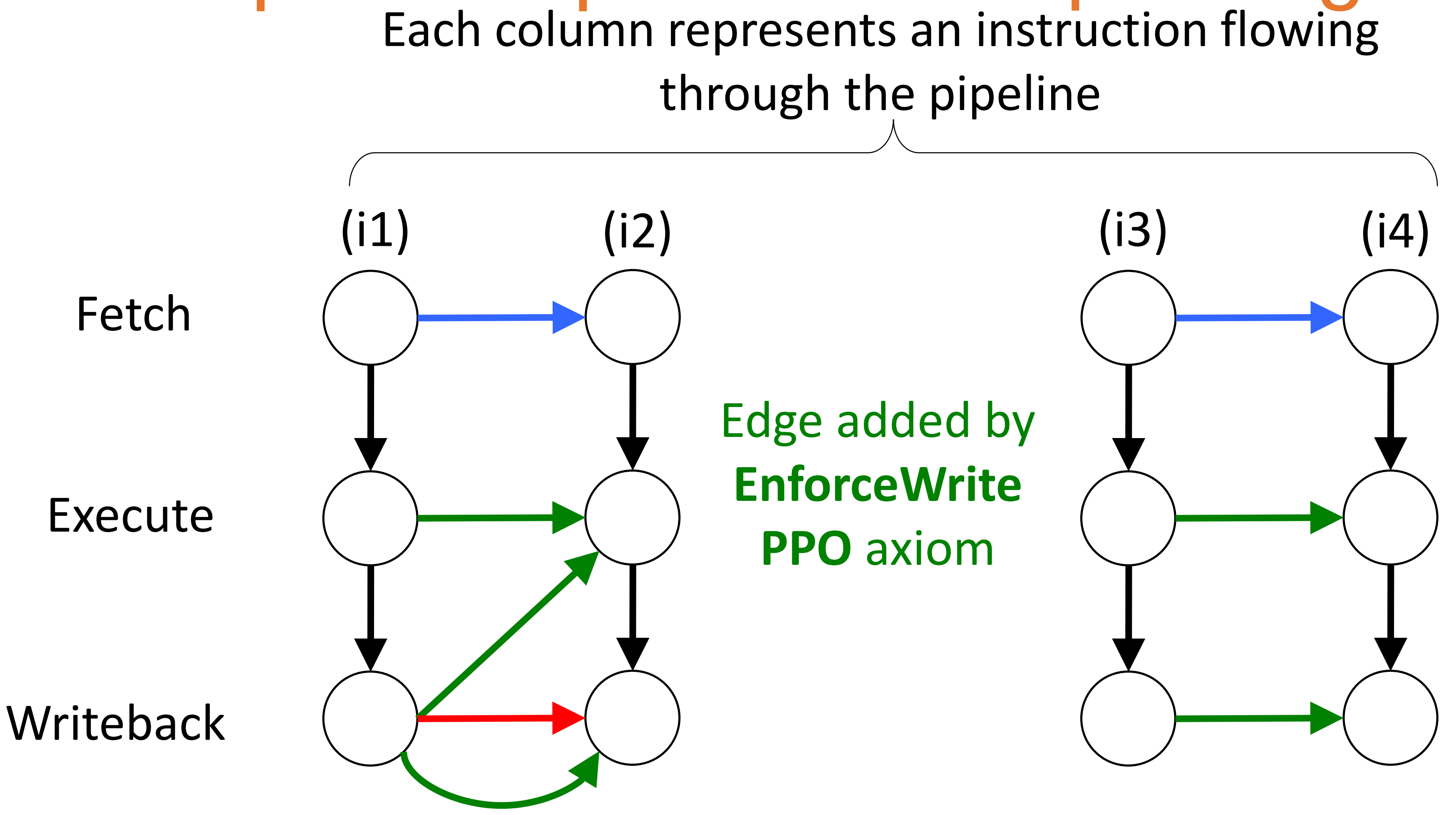
µhb Graphs for co-mp Using Axioms


Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback

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	

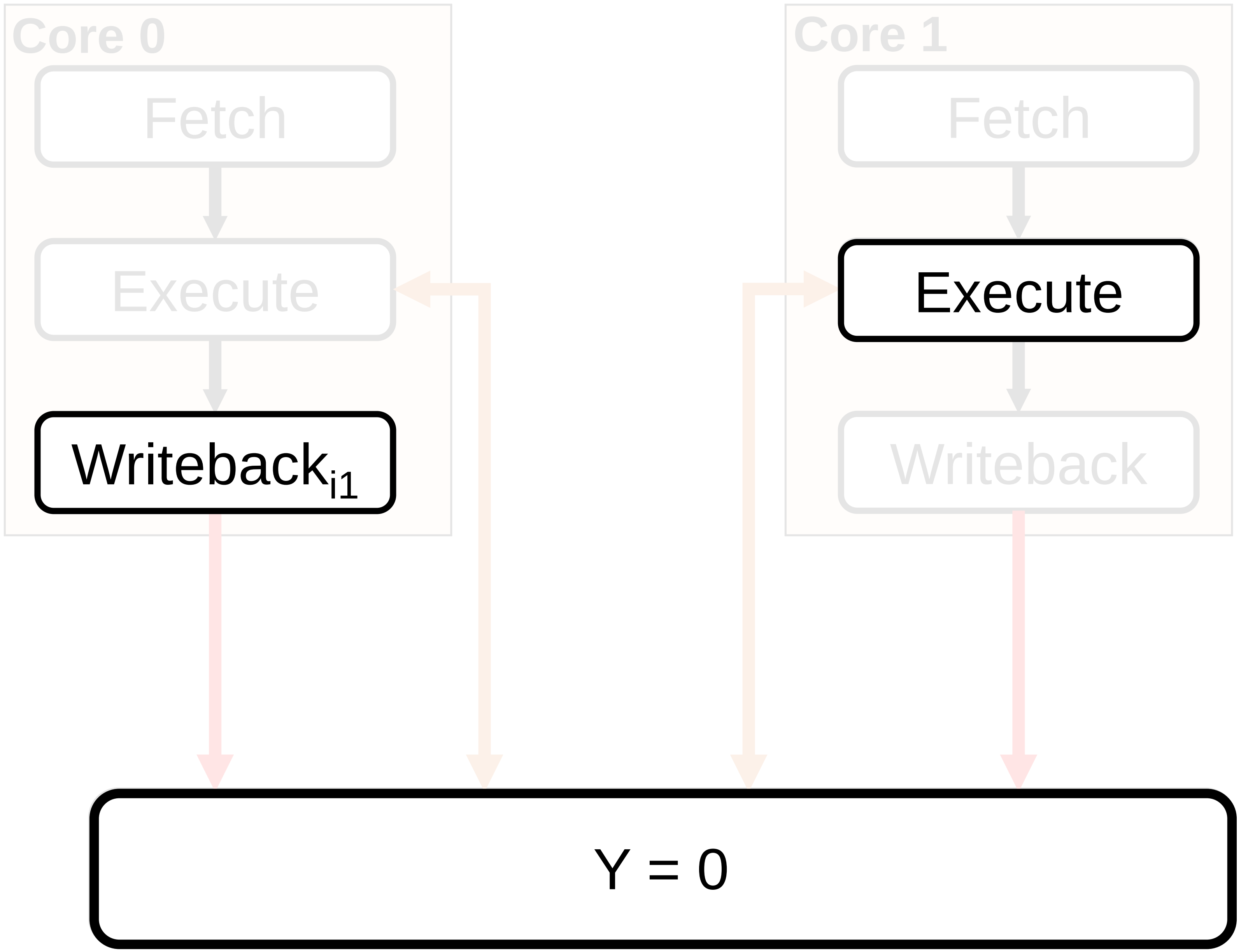
µhb Graphs for `co-mp` Using Axioms


Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback
Edge added by **EnforceWrite PPO** axiom

Initially, Mem[x] = 0

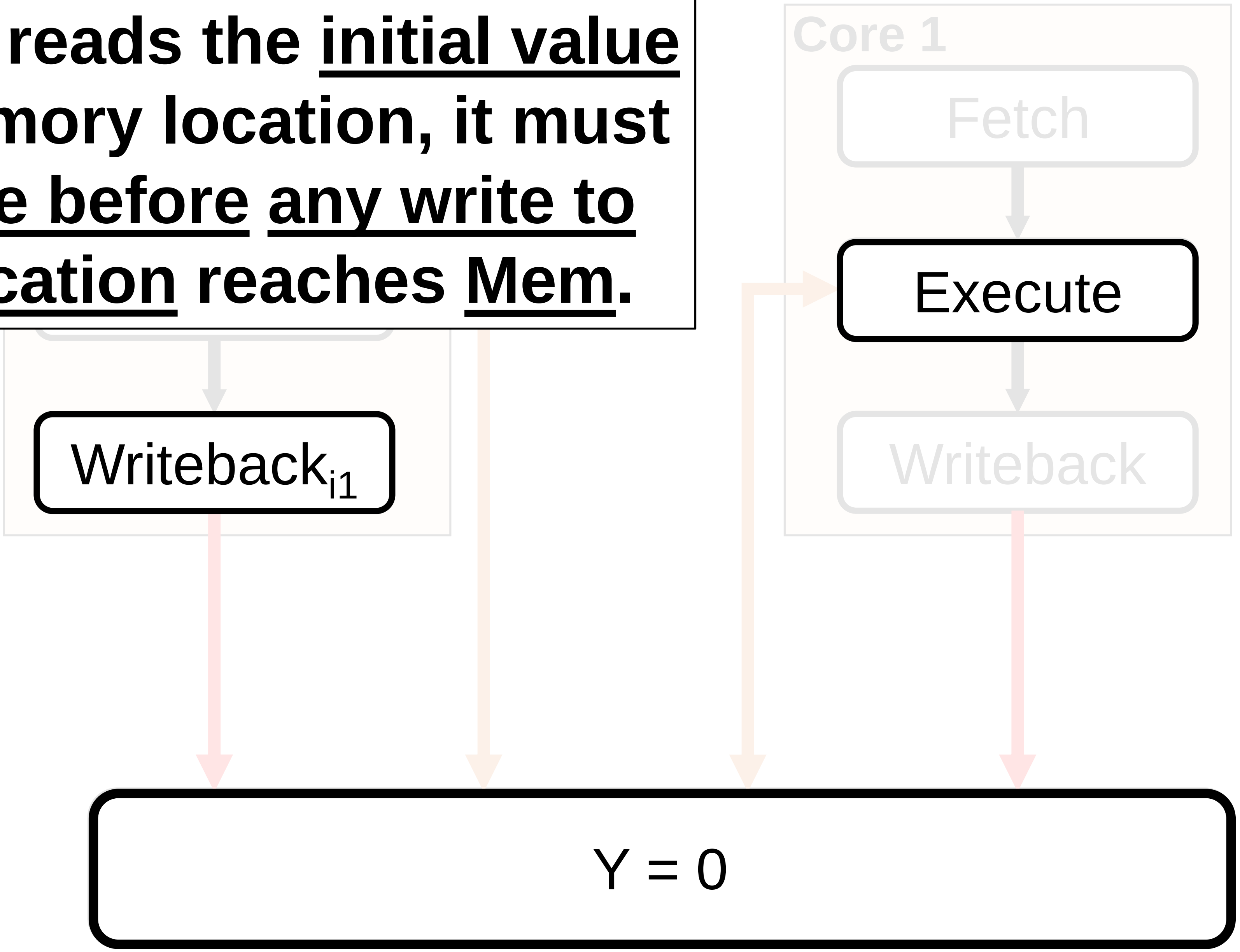
Thread 0	Thread 1
i1: Store [x] ← 1 i2: Store [x] ← 2	i3: r1 = Load [x] i4: r2 = Load [x]
SC **Forbids**: r1=2, r2=1, Mem[x] = 2	

Finding Axioms



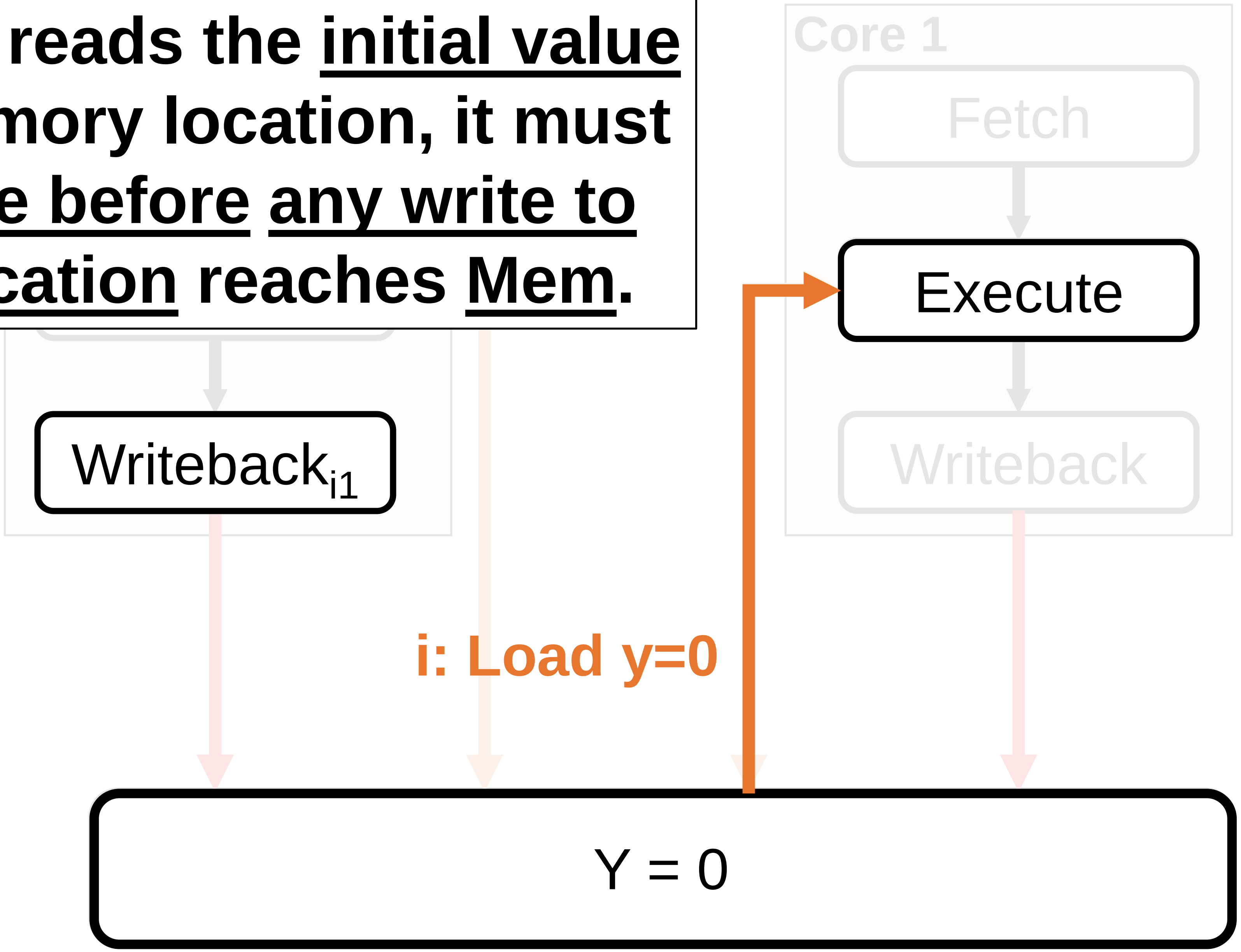
Finding Axioms

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



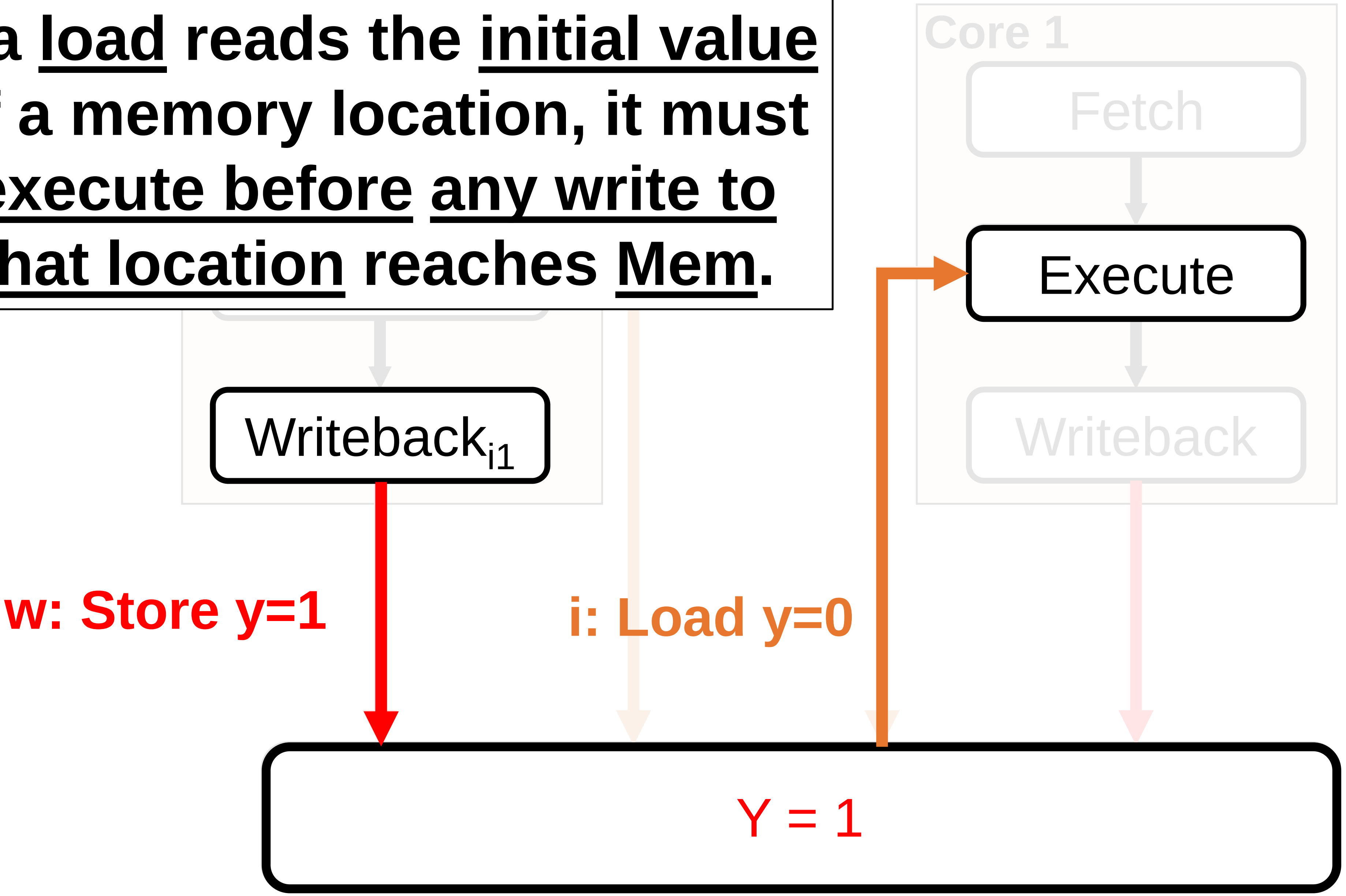
Finding Axioms

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



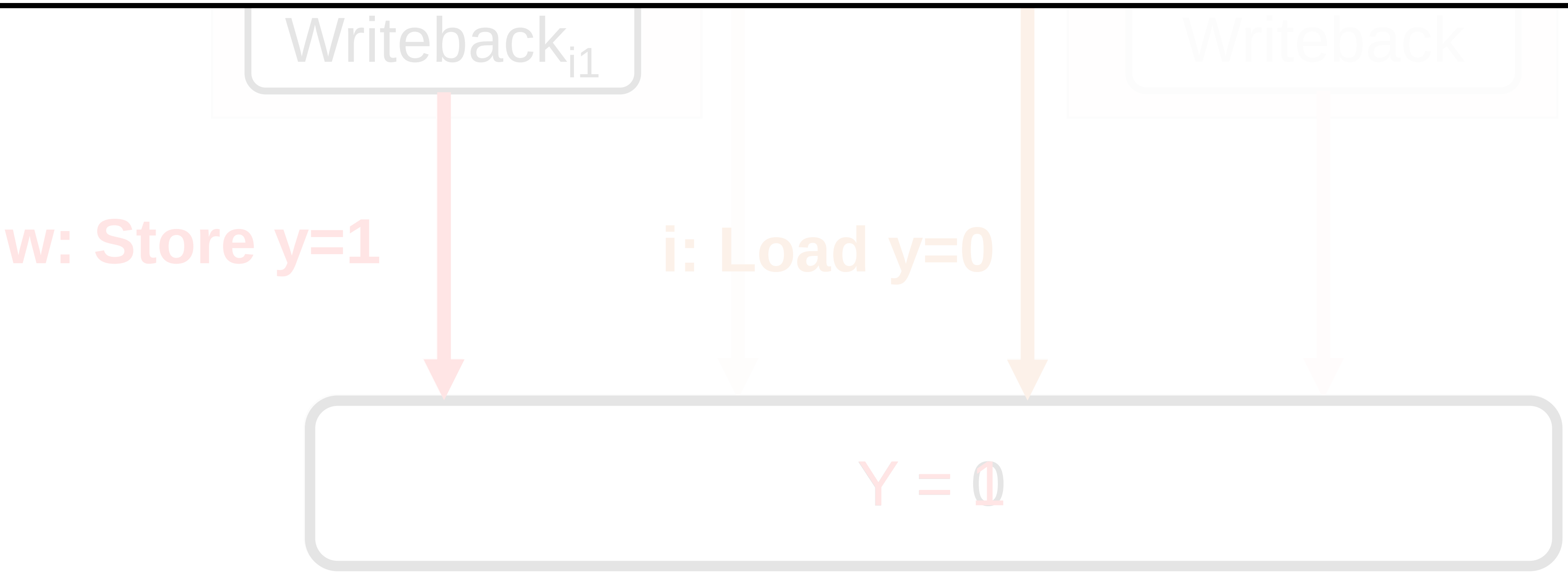
Finding Axioms

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



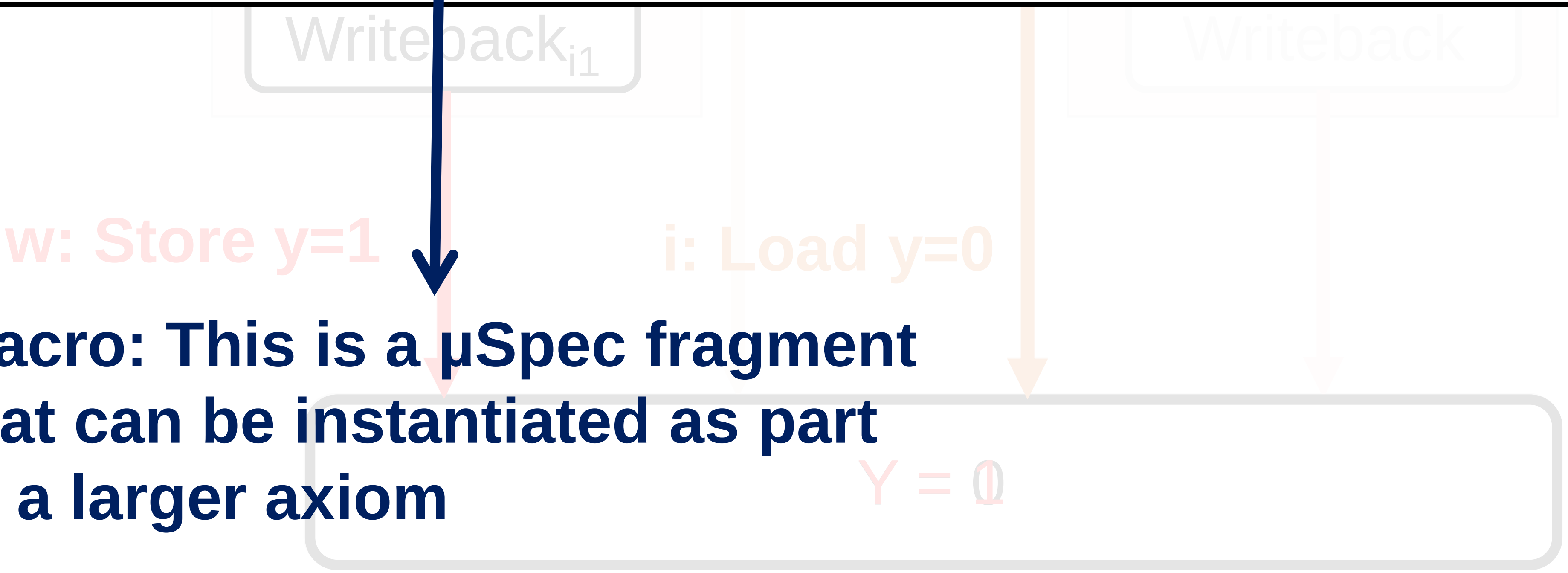
The BeforeAllWrites Macro

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



The BeforeAllWrites Macro

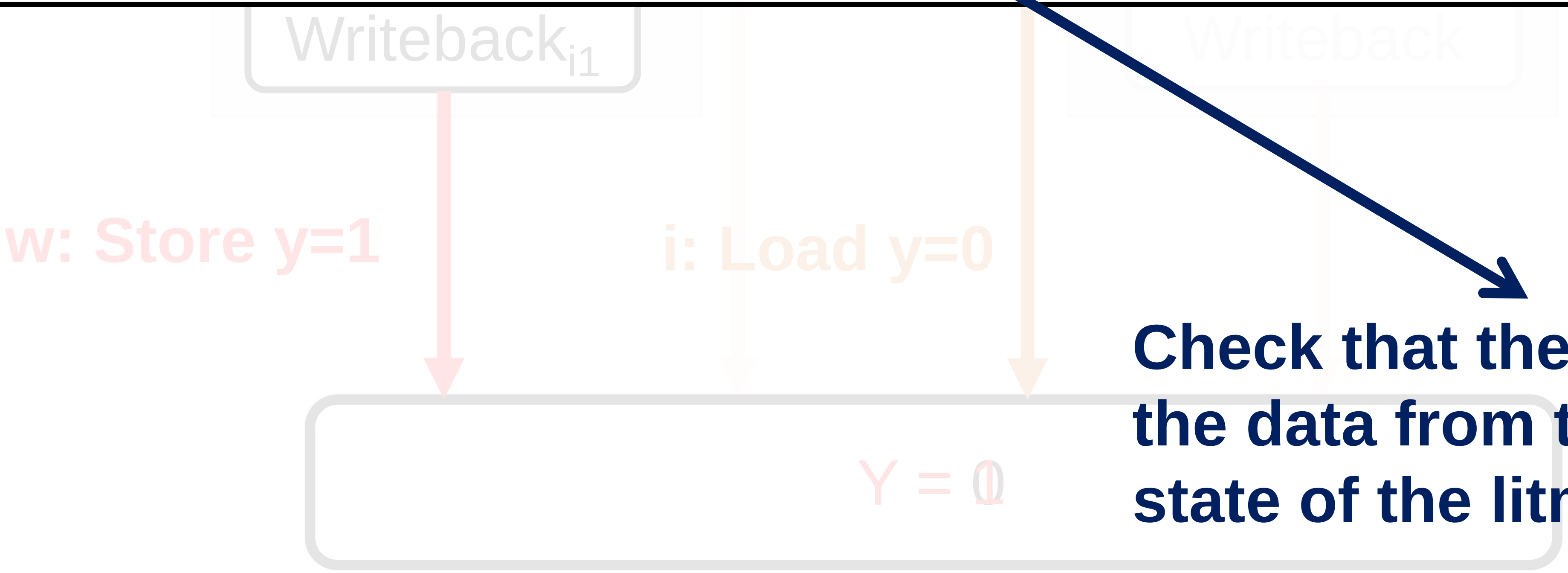
```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, _______), (w, _________))).
```



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

The BeforeAllWrites Macro

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

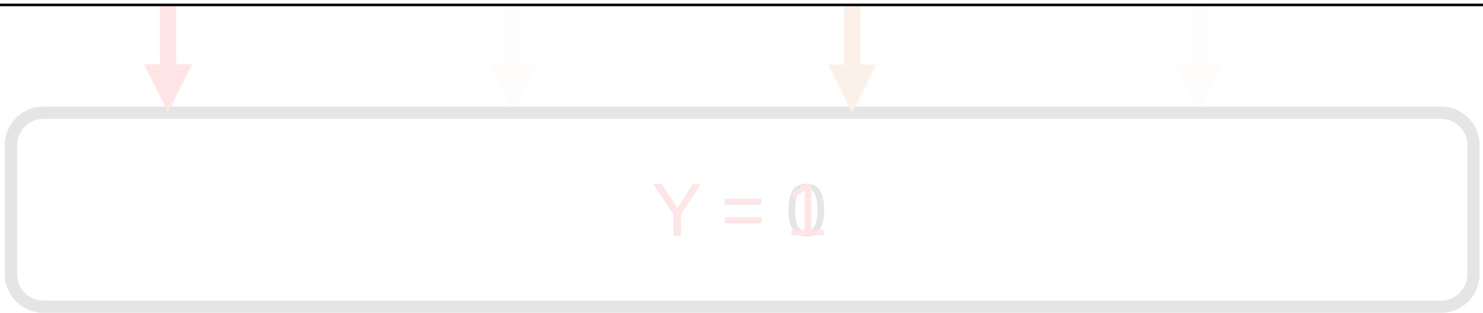


Check that the load reads the data from the initial state of the litmus test

The BeforeAllWrites Macro

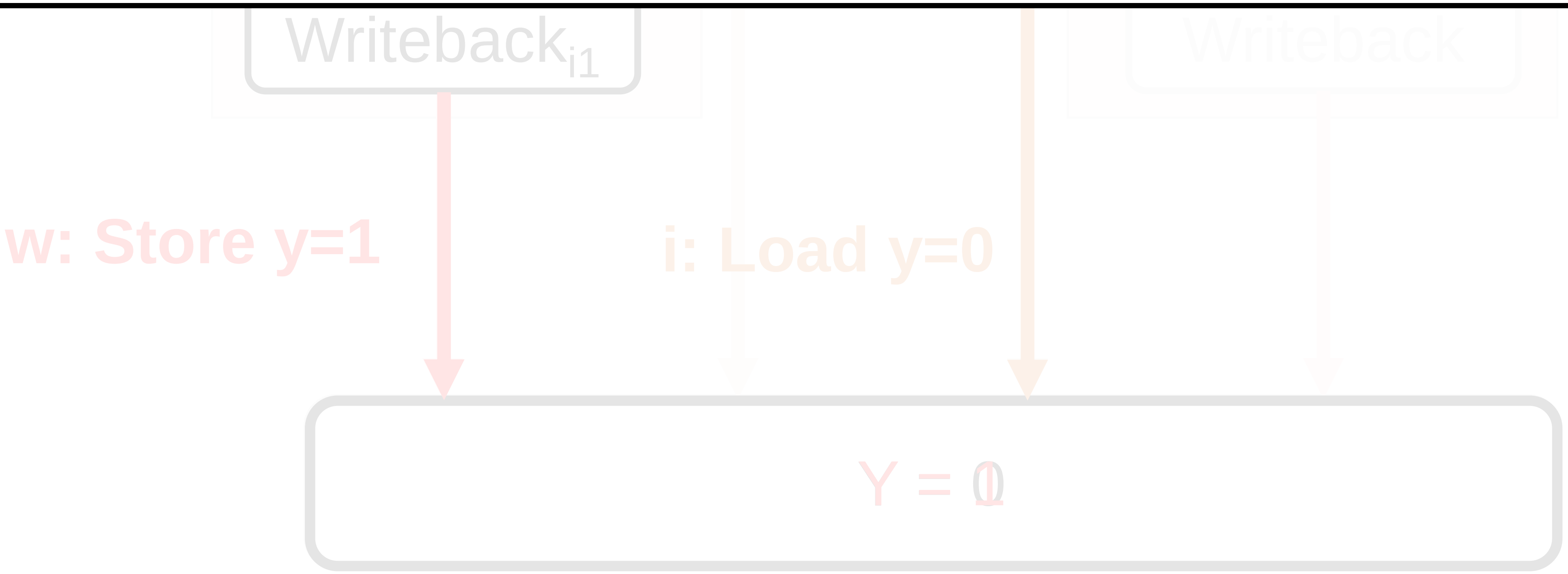
```
DefineMacro "BeforeAllWrites":
  DataFromInitialStateAtPA i /\
  forall microop "w", (
  (IsAnyWrite w /\ SamePhysicalAddress w i
    /\ ~SameMicroop i w) =>
    AddEdge ((i, _______), (w, _________))).
```

If a <u>load</u> reads the <u>initial value</u> of a memory location, it must <u>execute before</u> <u>any write to that addr</u> completes its <u>writeback</u>.



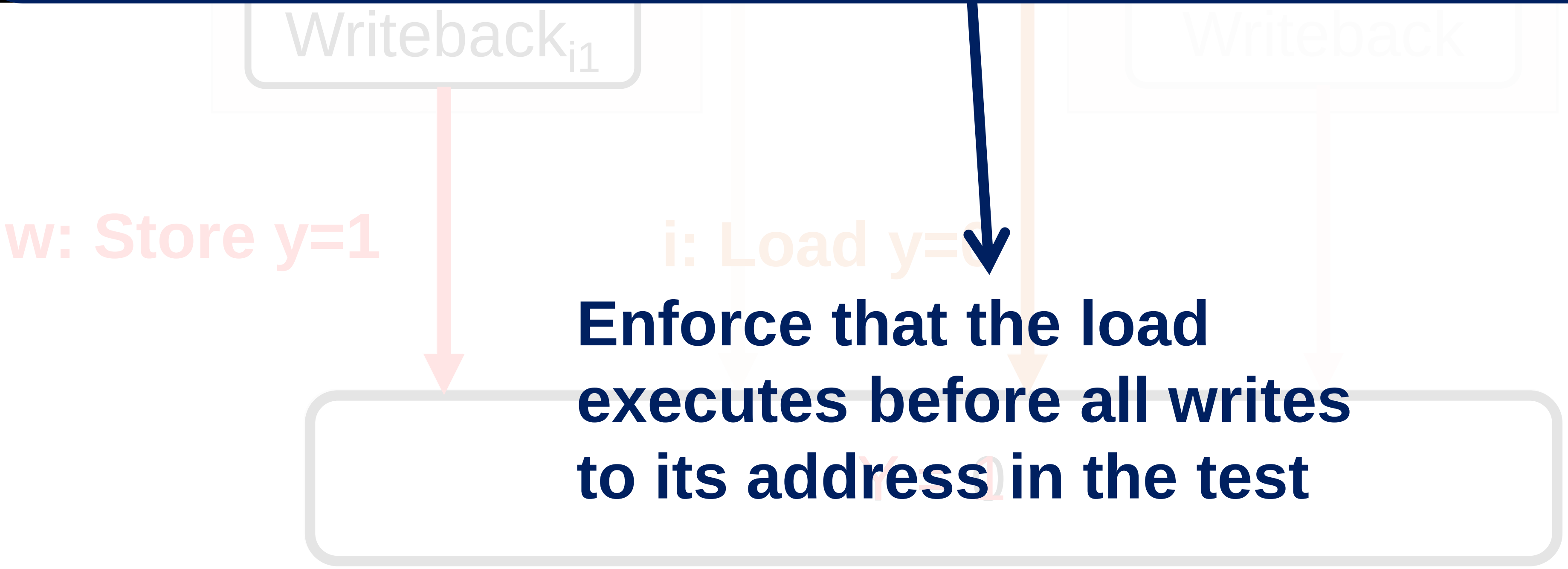
The BeforeAllWrites Macro

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



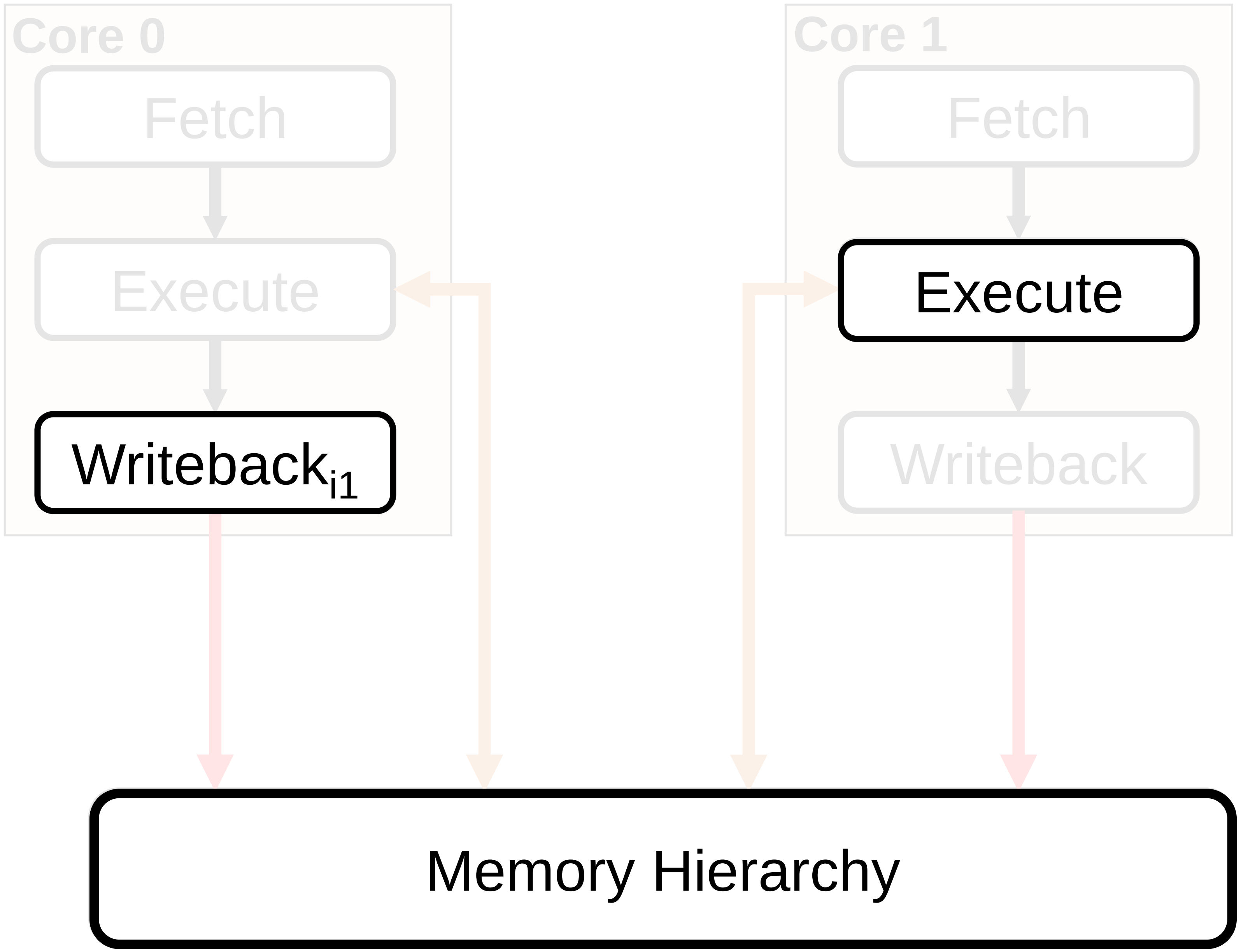
The BeforeAllWrites Macro

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



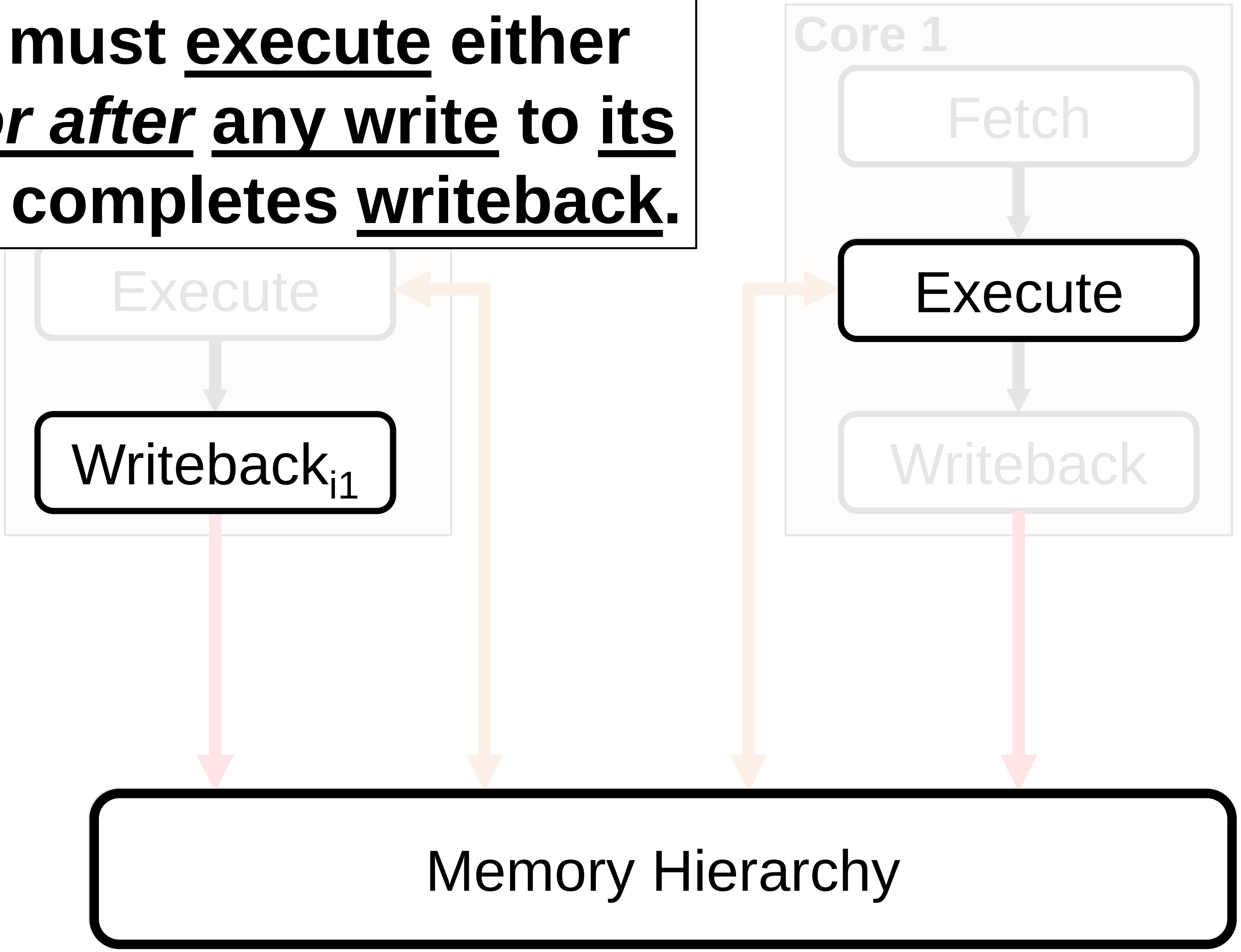
Enforce that the load executes before all writes to its address in the test

Finding Axioms



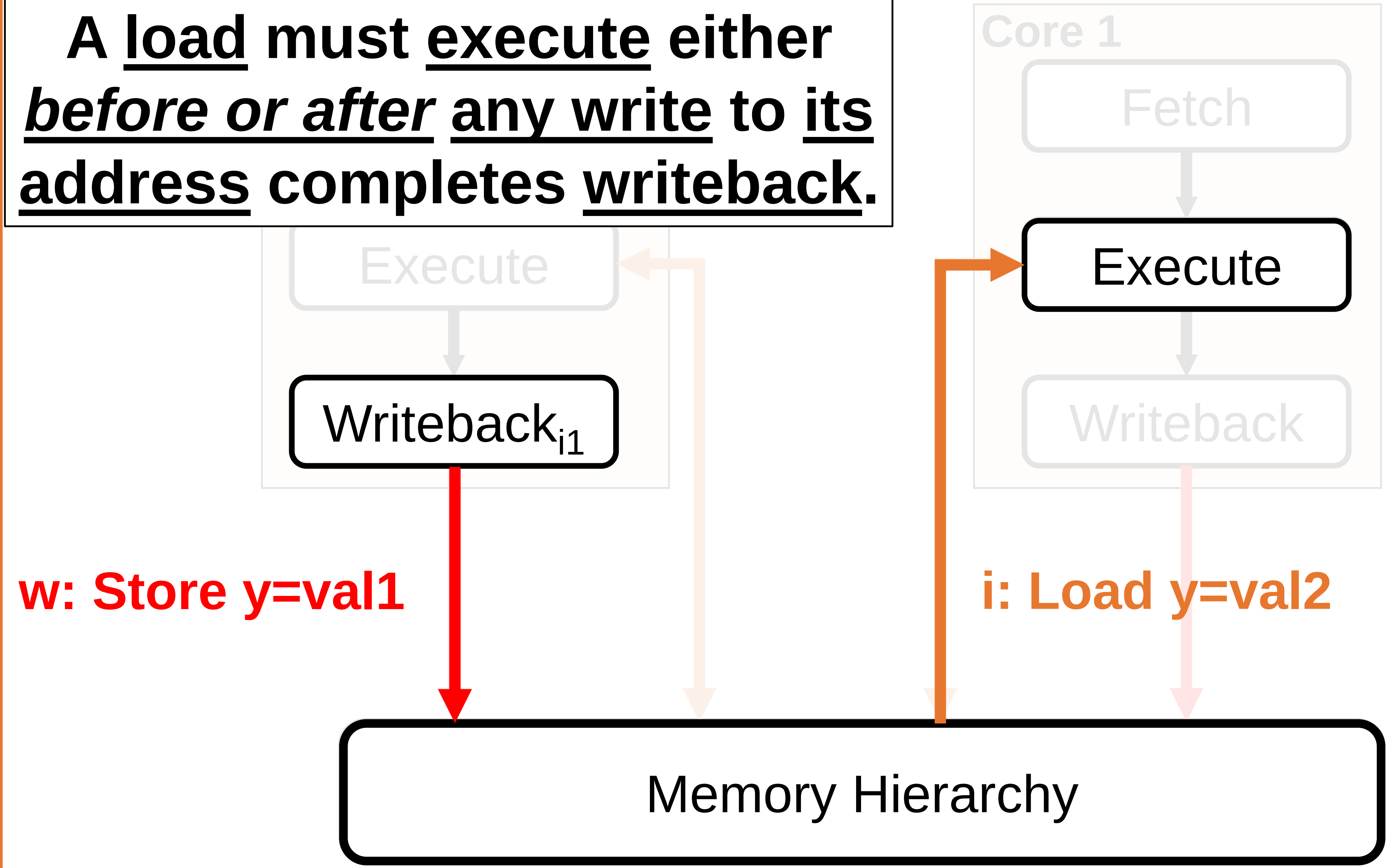
Finding Axioms

A <u>load</u> must <u>execute</u> either *<u>before or after</u>* <u>any write</u> to <u>its address</u> completes <u>writeback</u>.



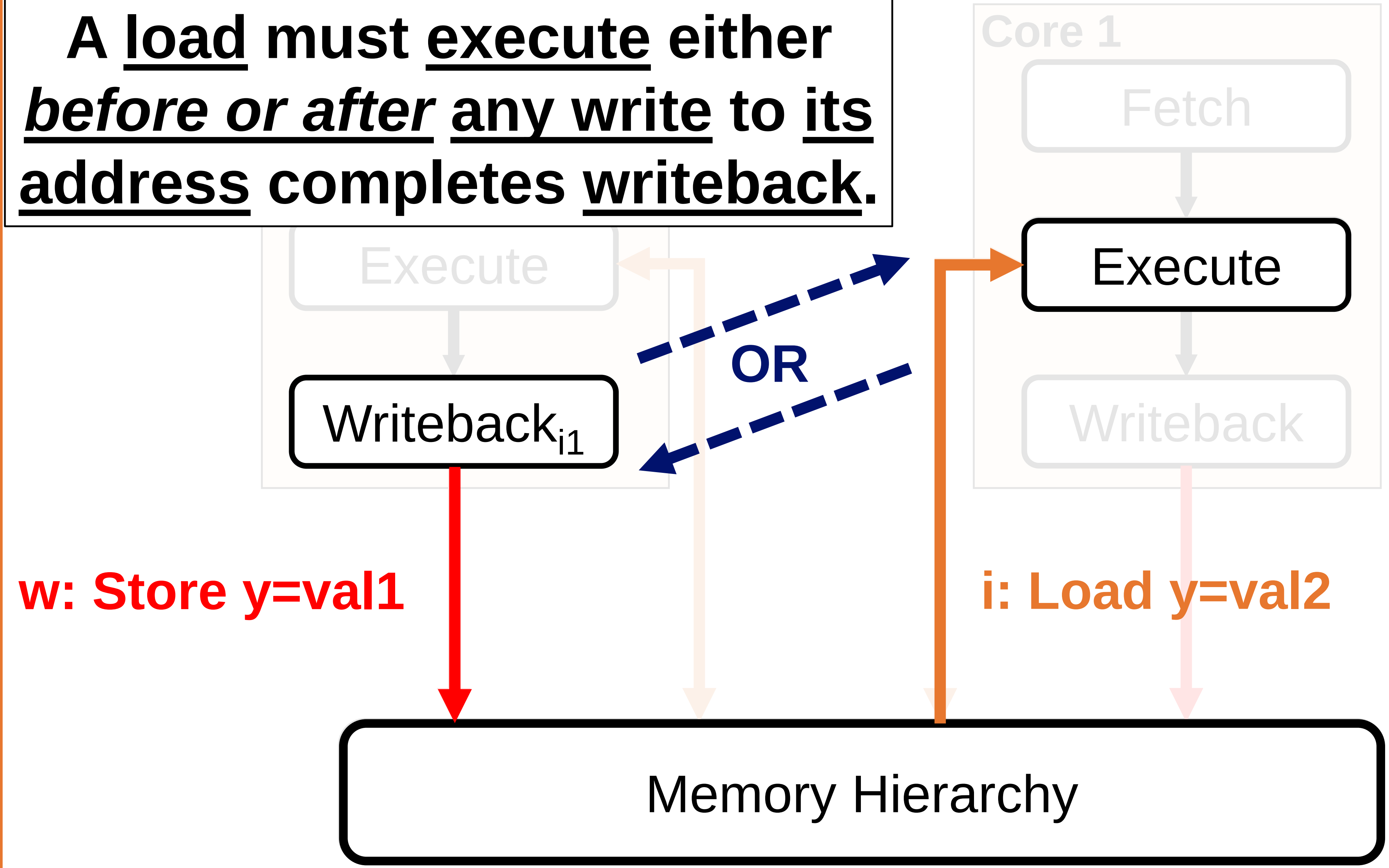
Finding Axioms

A <u>load</u> must <u>execute</u> either *<u>before or after</u>* <u>any write</u> to <u>its address</u> completes <u>writeback</u>.



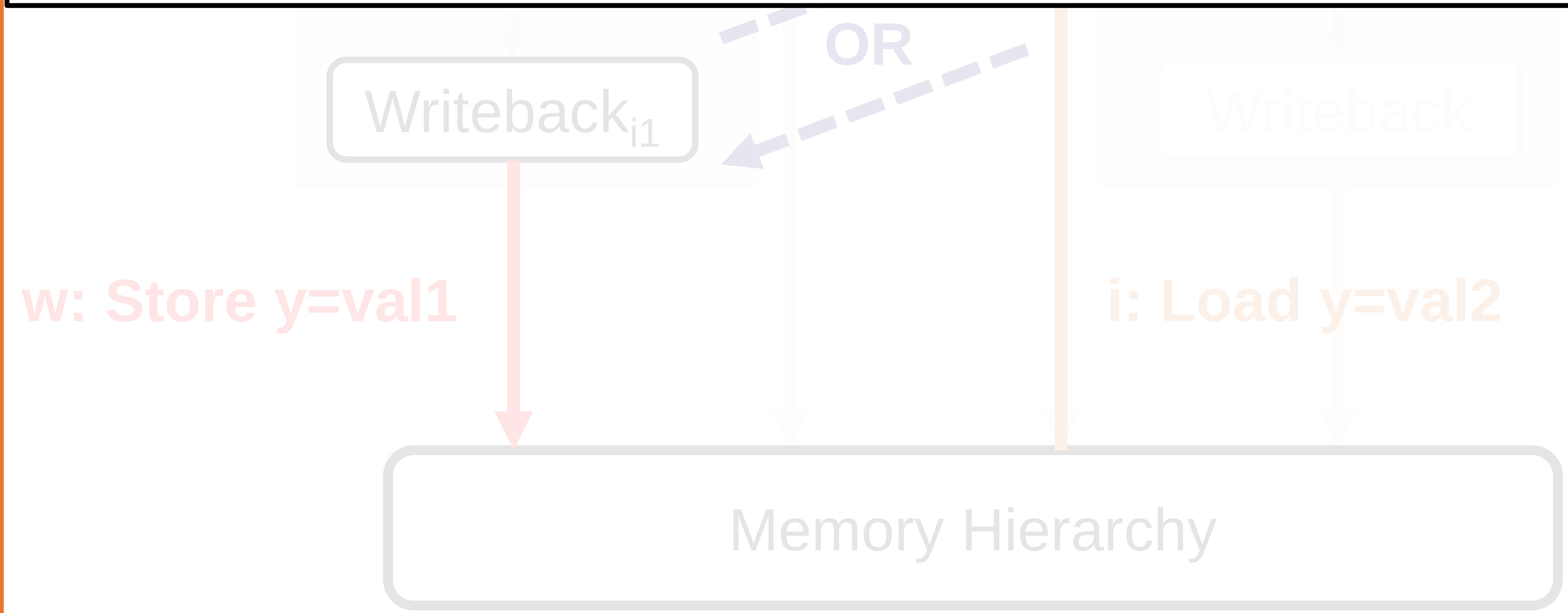
Finding Axioms

A <u>load</u> must <u>execute</u> either *<u>before or after</u>* <u>any write</u> to <u>its address</u> completes <u>writeback</u>.



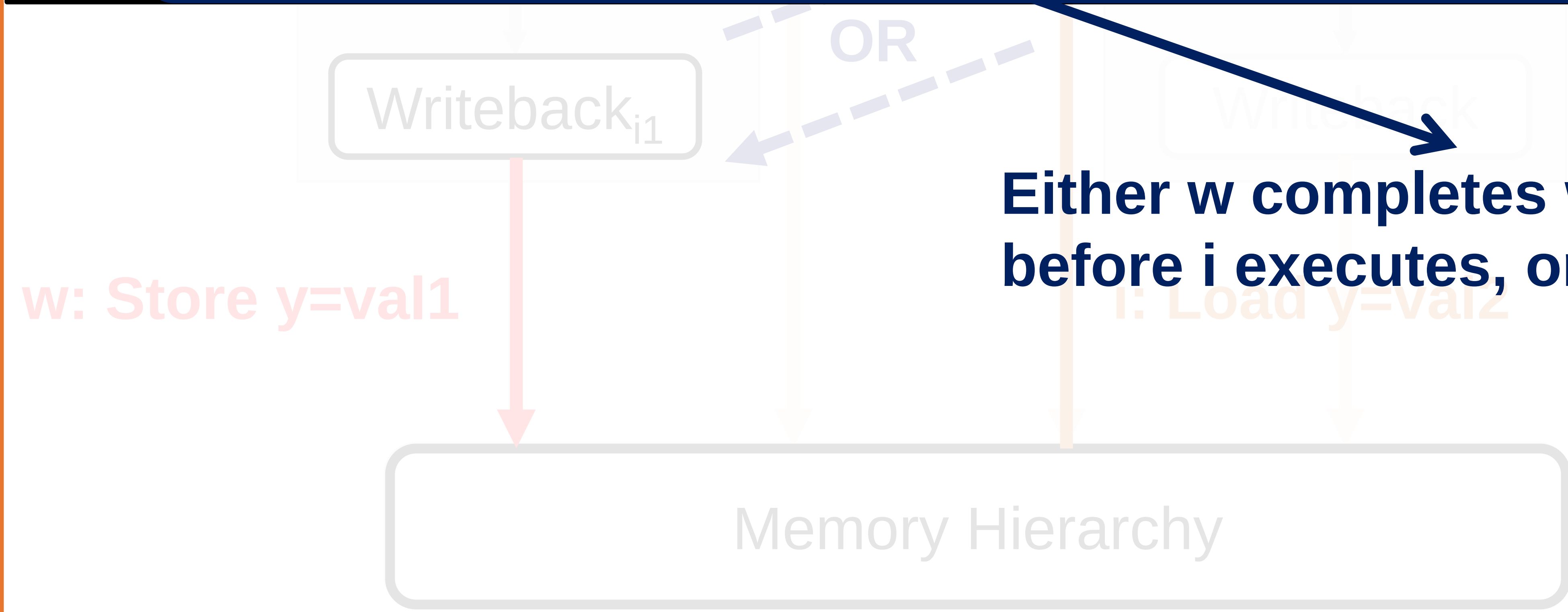
The Before_Or_After_Every_SameAddrWrite Macro

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



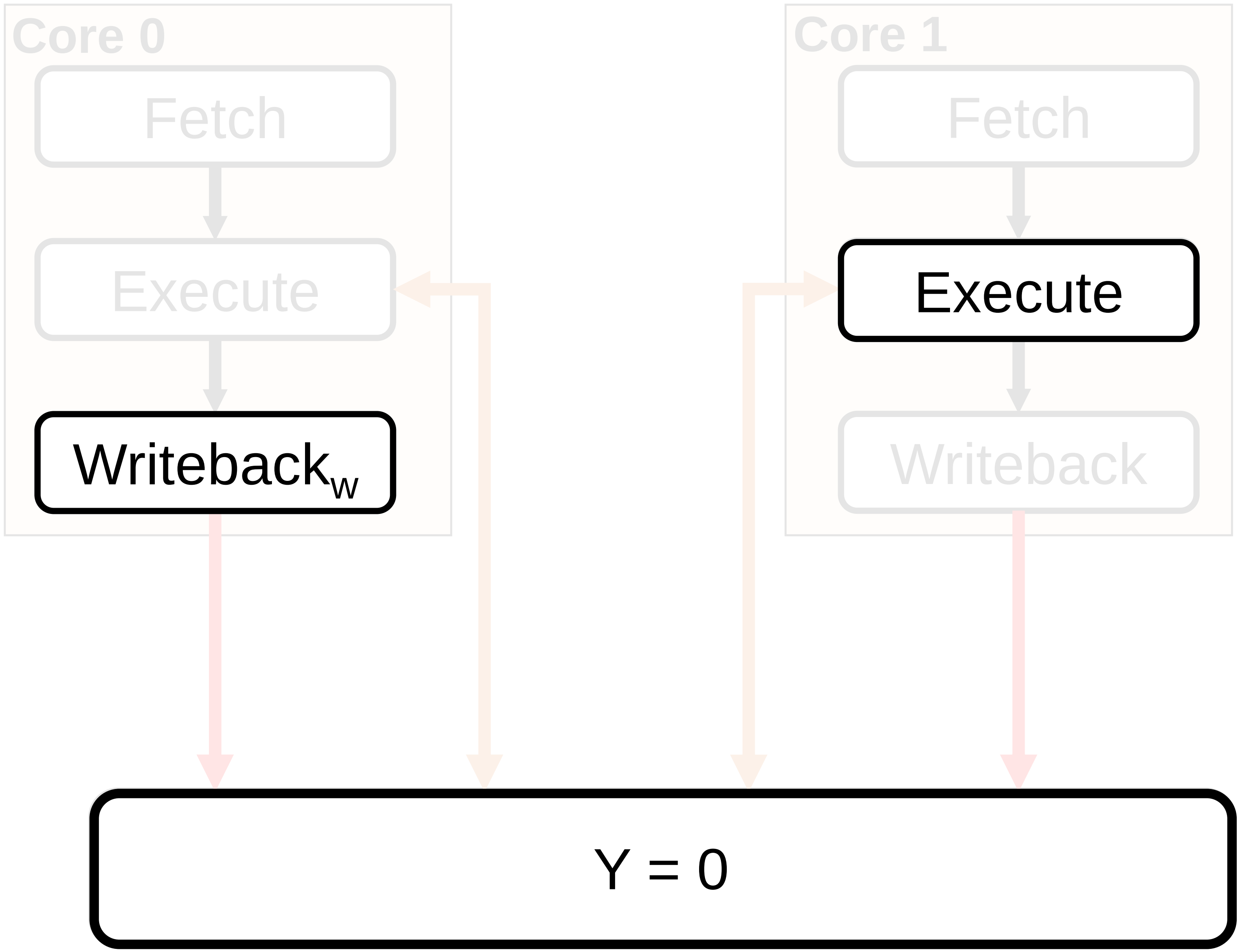
The Before_Or_After_Every_SameAddrWrite Macro

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



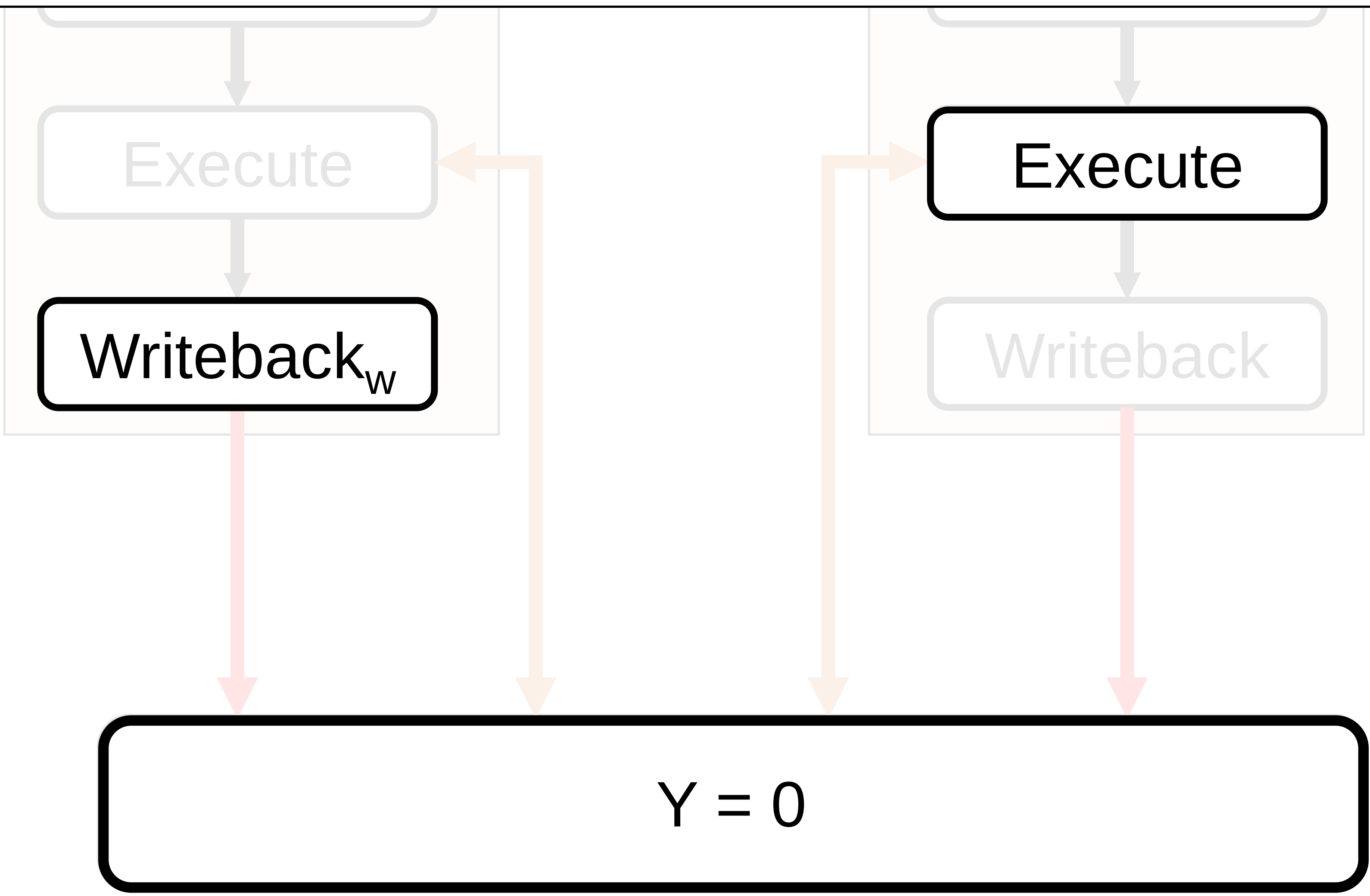
Either w completes writeback before i executes, or vice-versa.

Finding Axioms



Finding Axioms

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

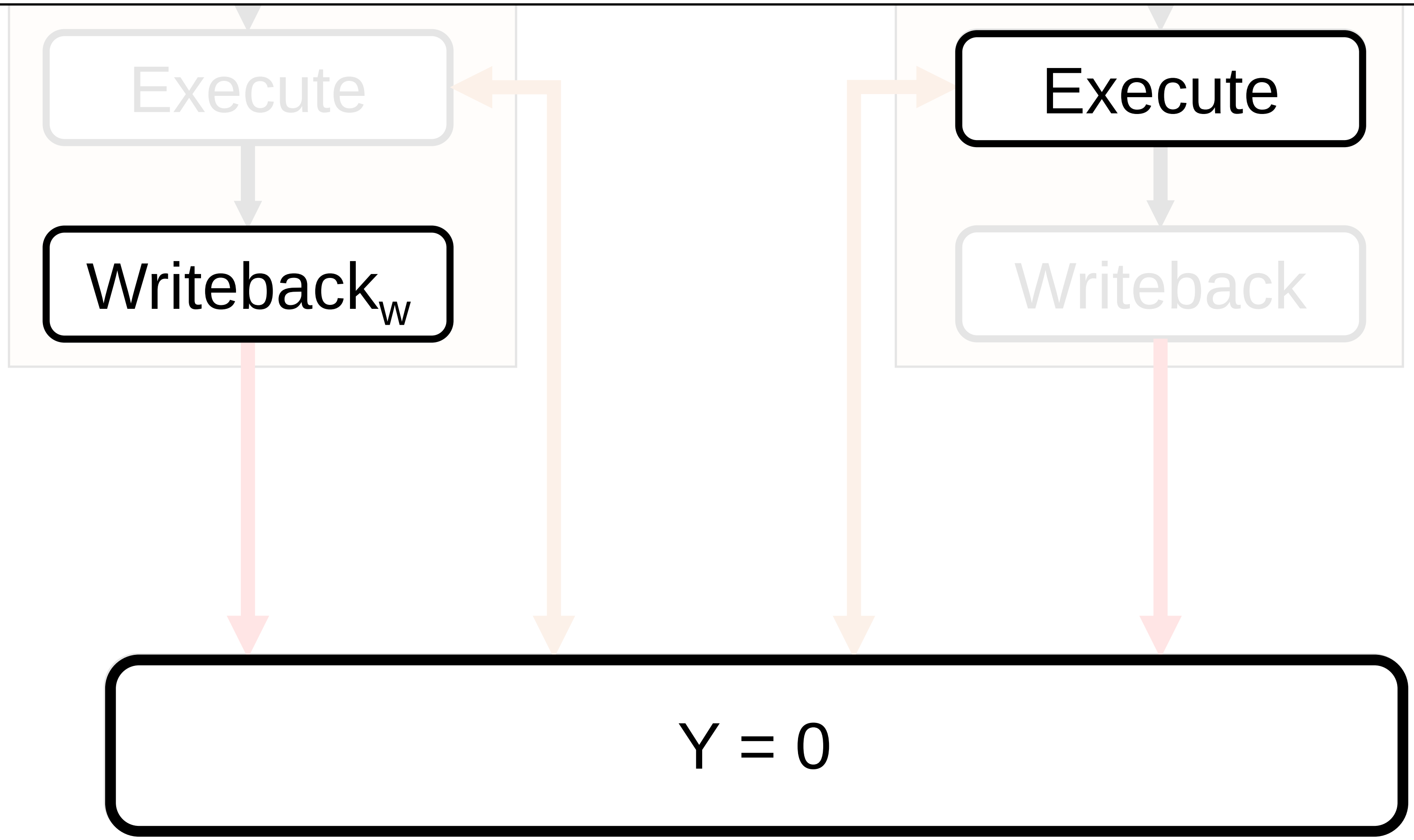


Finding Axioms

Alternatively:

1) The load must <u>execute after</u> the <u>write it reads from</u>

2) <u>No writes</u> to <u>that address</u> <u>between</u> the <u>source</u> write and the <u>read</u>

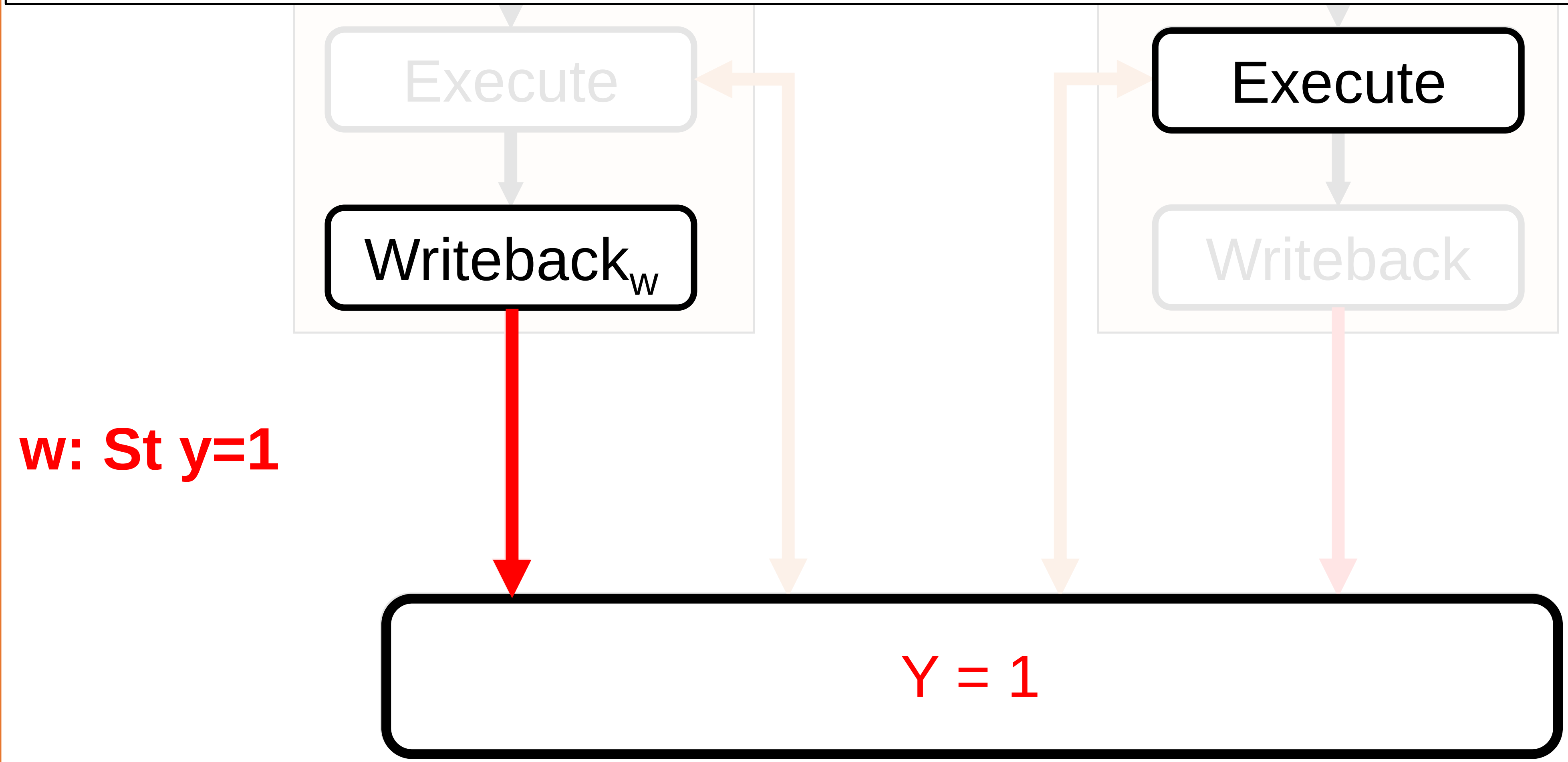


Finding Axioms

<u>Alternatively:</u>

1) The load must <u>execute after</u> the <u>write it reads from</u>

2) <u>No writes</u> to <u>that address</u> <u>between</u> the <u>source</u> write and the <u>read</u>

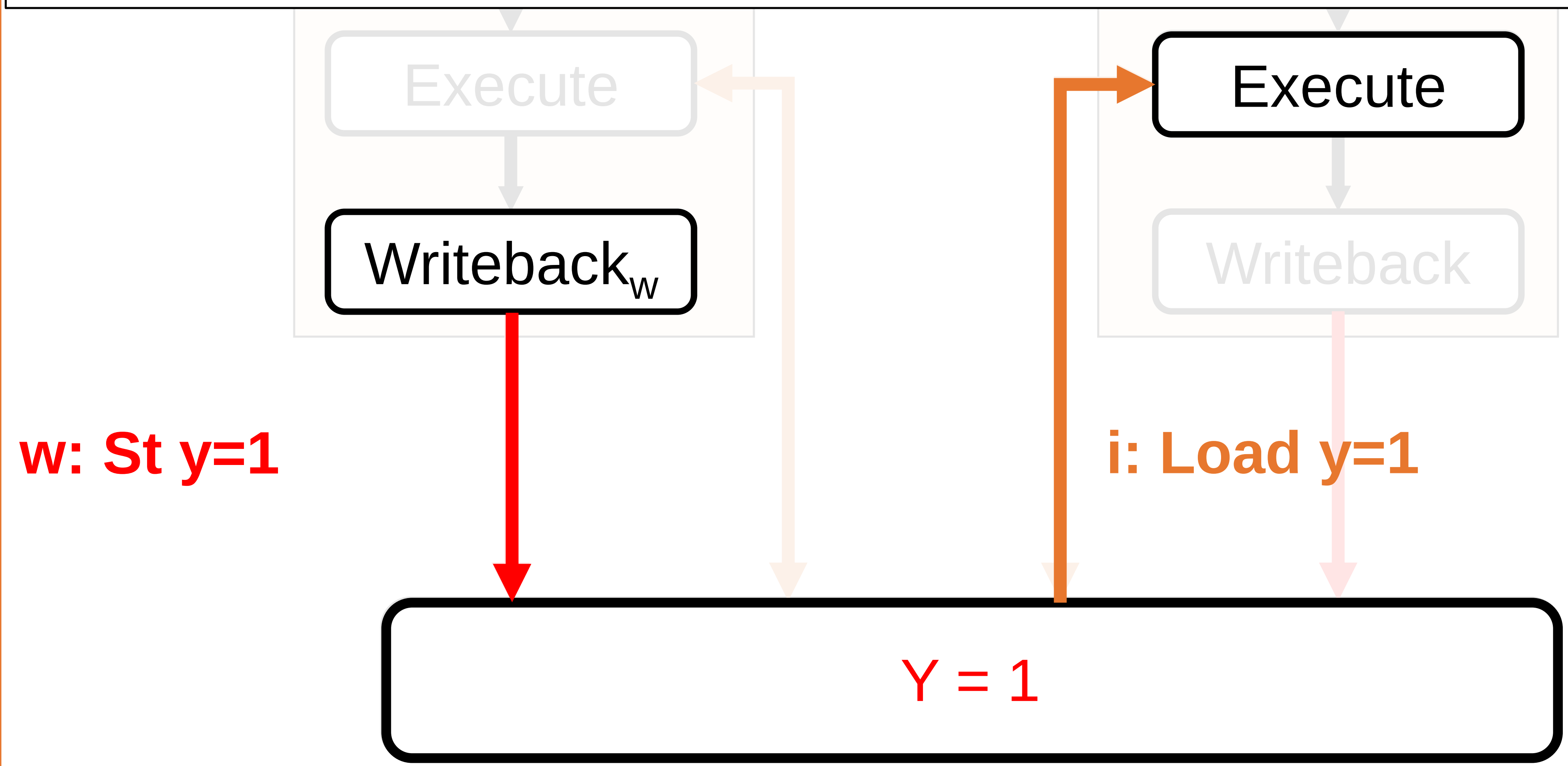


Finding Axioms

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

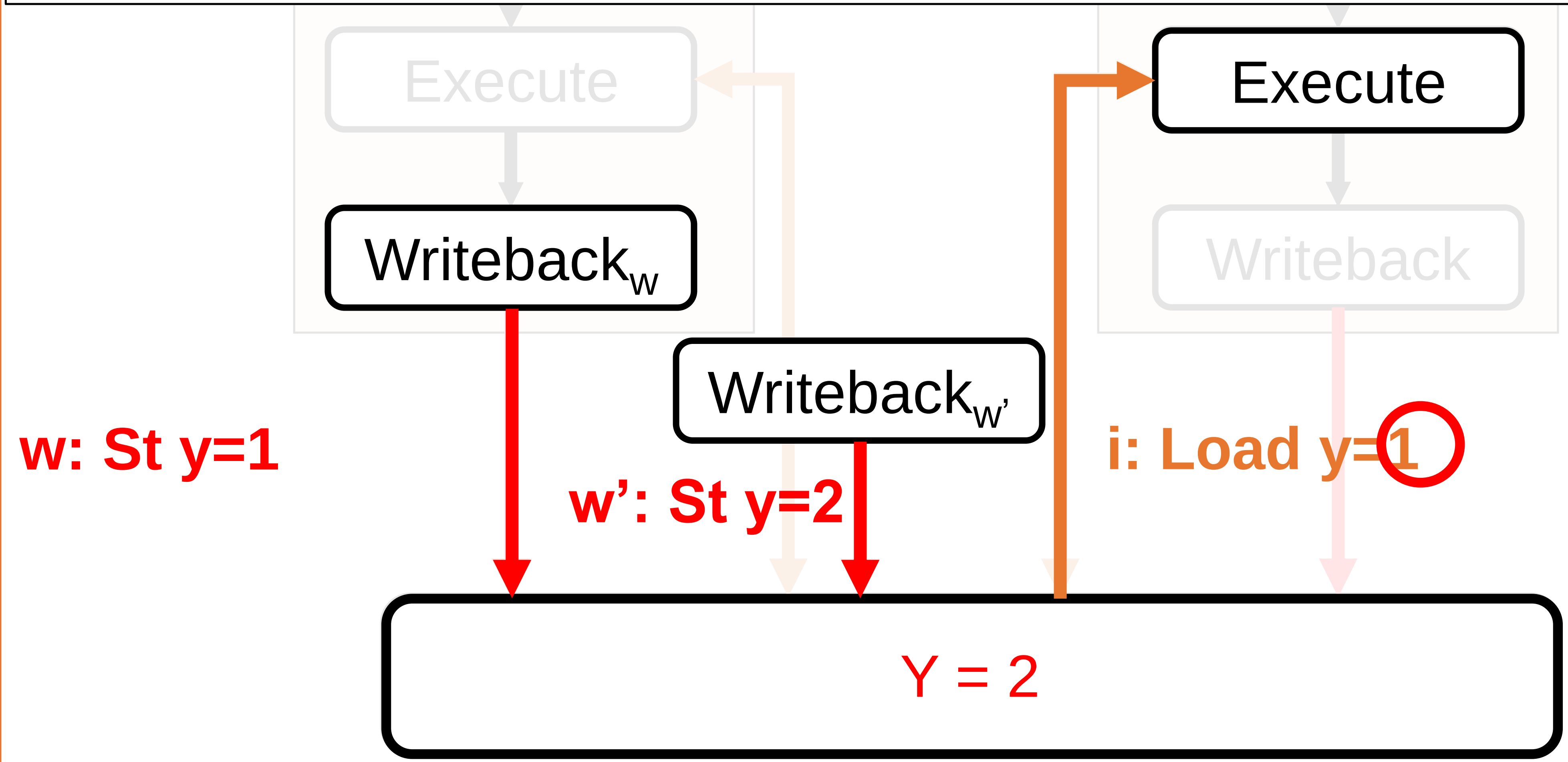


Finding Axioms

<u>Alternatively:</u>

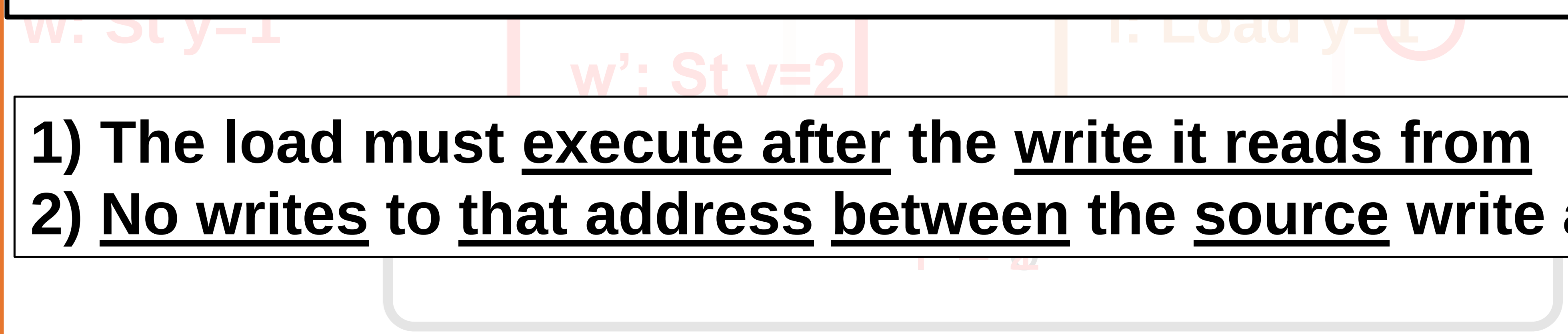
1) The load must <u>execute after</u> the <u>write it reads from</u>

2) <u>No writes</u> to <u>that address</u> <u>between</u> the <u>source</u> write and the <u>read</u>



The No_SameAddrWrites_Btwn_Src_And_Read Macro

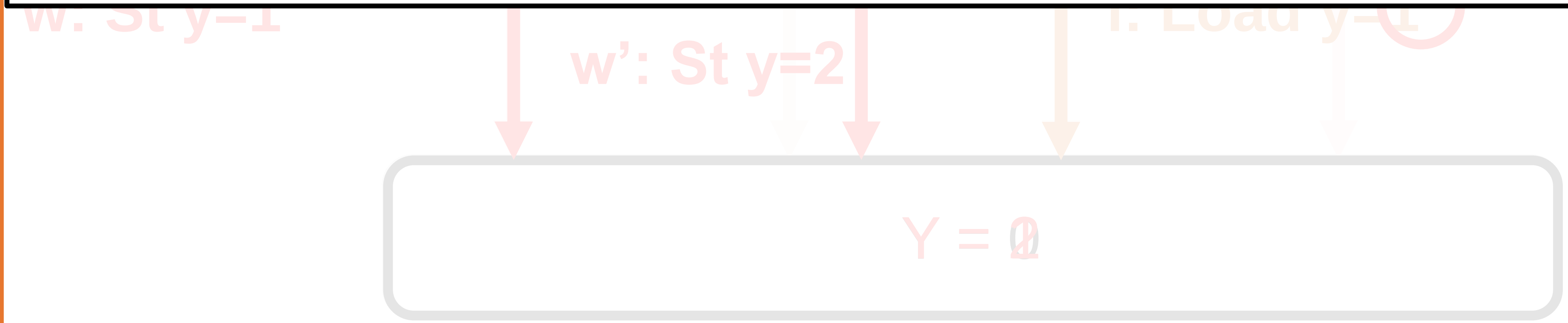
```
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
 exists microop "w", (
  IsAnyWrite w /\ __________________ w i /\ ________ w i
  /\ AddEdge ((w, Writeback), (i, Execute)) /\
  ~(exists microop "w'",
    IsAnyWrite w' /\ ___________________ i w’ /\
    ~SameMicroop w w’
    /\ EdgesExist [((w,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



1) **The load must <u>execute after</u> the <u>write it reads from</u>**
2) **<u>No writes</u> to <u>that address</u> <u>between</u> the <u>source</u> write and the <u>read</u>**

The No_SameAddrWrites_Btwn_Src_And_Read Macro

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



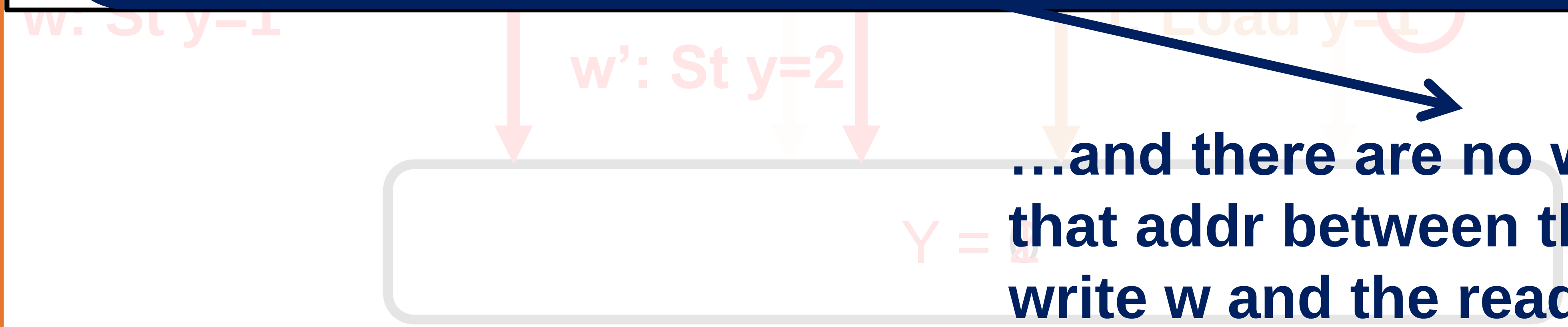
The No_SameAddrWrites_Btwn_Src_And_Read Macro

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

Read i executes after its source write w reaches memory…

The No_SameAddrWrites_Btwn_Src_And_Read Macro

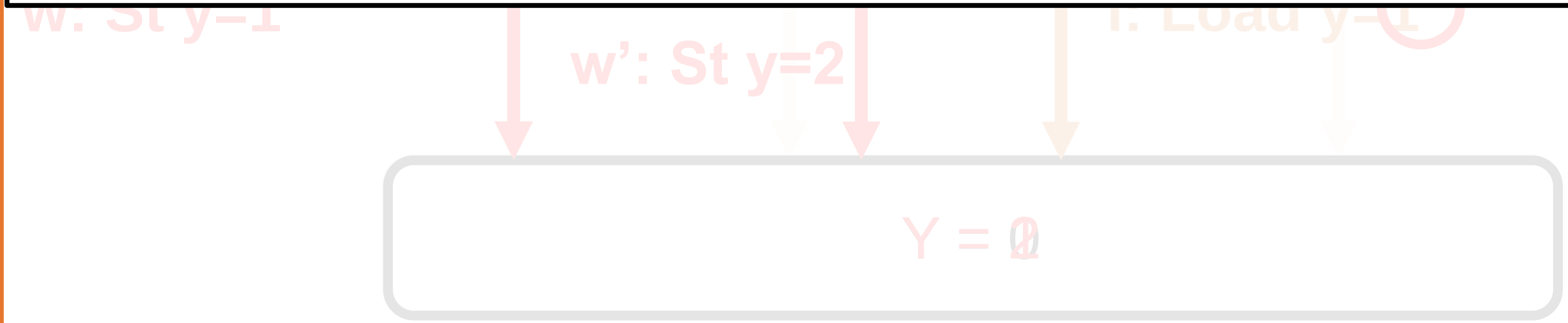
```
DefineMacro "No_SameAddrWrites_Btwn_Src_And_Read":
 exists microop "w", (
  IsAnyWrite w /\ SamePhysicalAddress w i /\ SameData w i
  /\ AddEdge ((w, Writeback), (i, Execute)) /\
  ~(exists microop "w'",
    IsAnyWrite w' /\ SamePhysicalAddress i w’ /\
    ~SameMicroop w w’
    /\ EdgesExist [((w,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

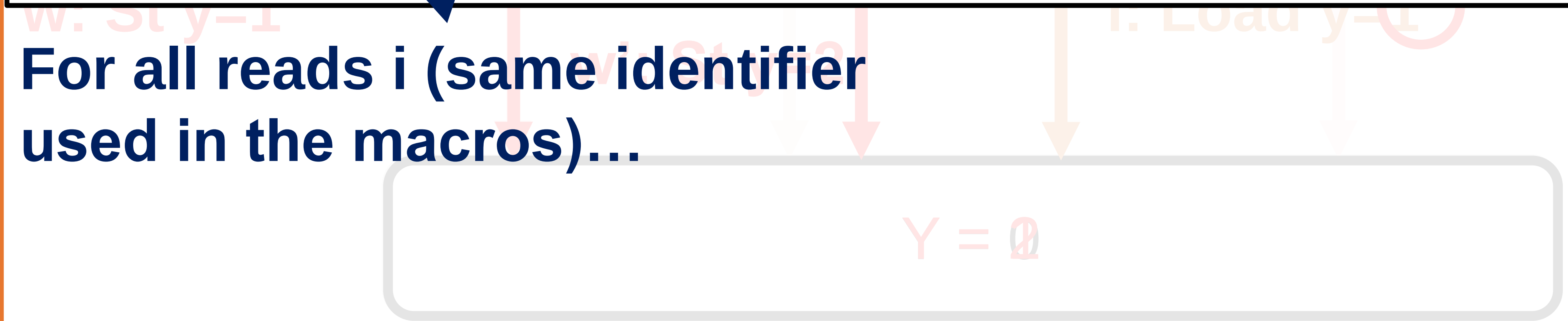
```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (
    ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    /\
    ExpandMacro Before_Or_After_Every_SameAddrWrite
  )).
```



Putting the Macros together: the Read_Values axiom

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

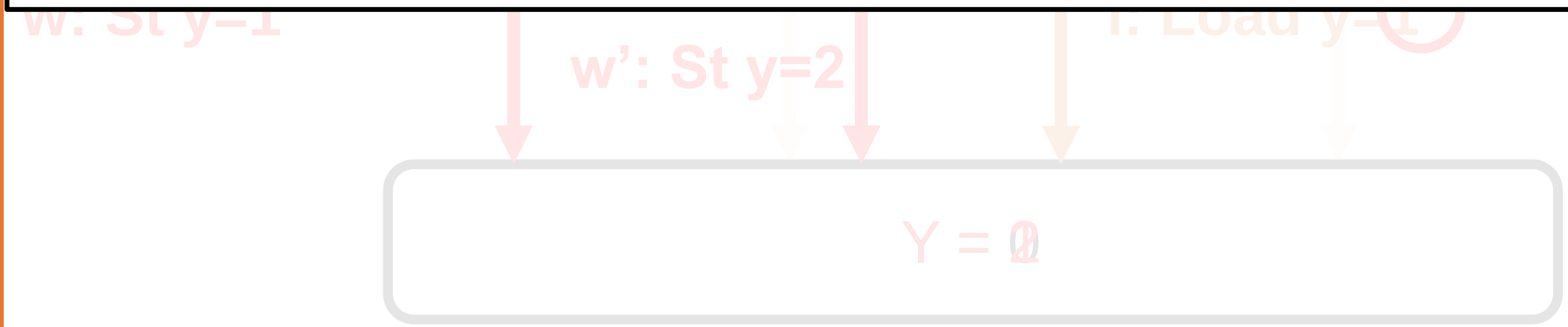
For all reads i (same identifier used in the macros)…



Putting the Macros together: the Read_Values axiom

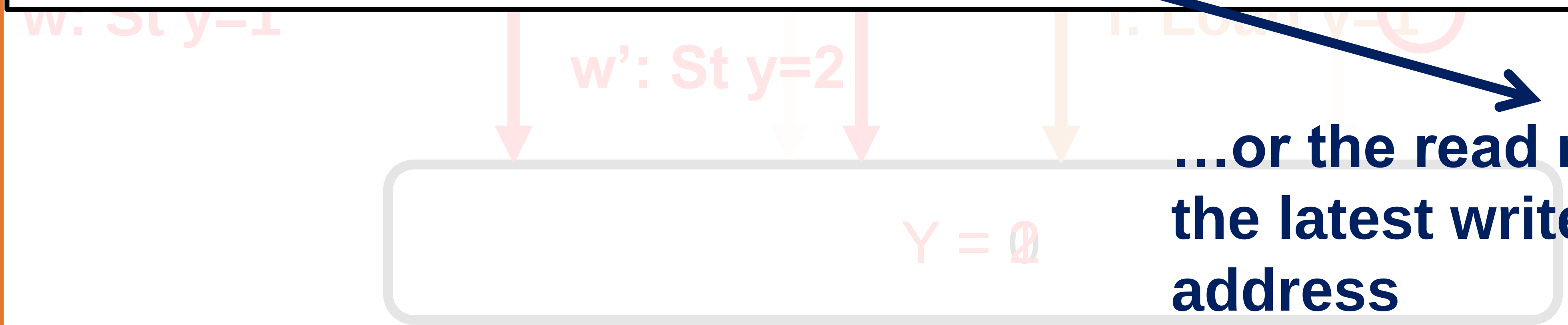
```
Axiom "Read_Values":
forall microops "i",
IsAnyRead i =>
(ExpandMacro BeforeAllWrites \/
  (
    ExpandMacro No_SameAddrWrites_Btwn_Src_And_Read
    /\
    ExpandMacro Before_Or_After_Every_SameAddrWrite
  )).
```

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



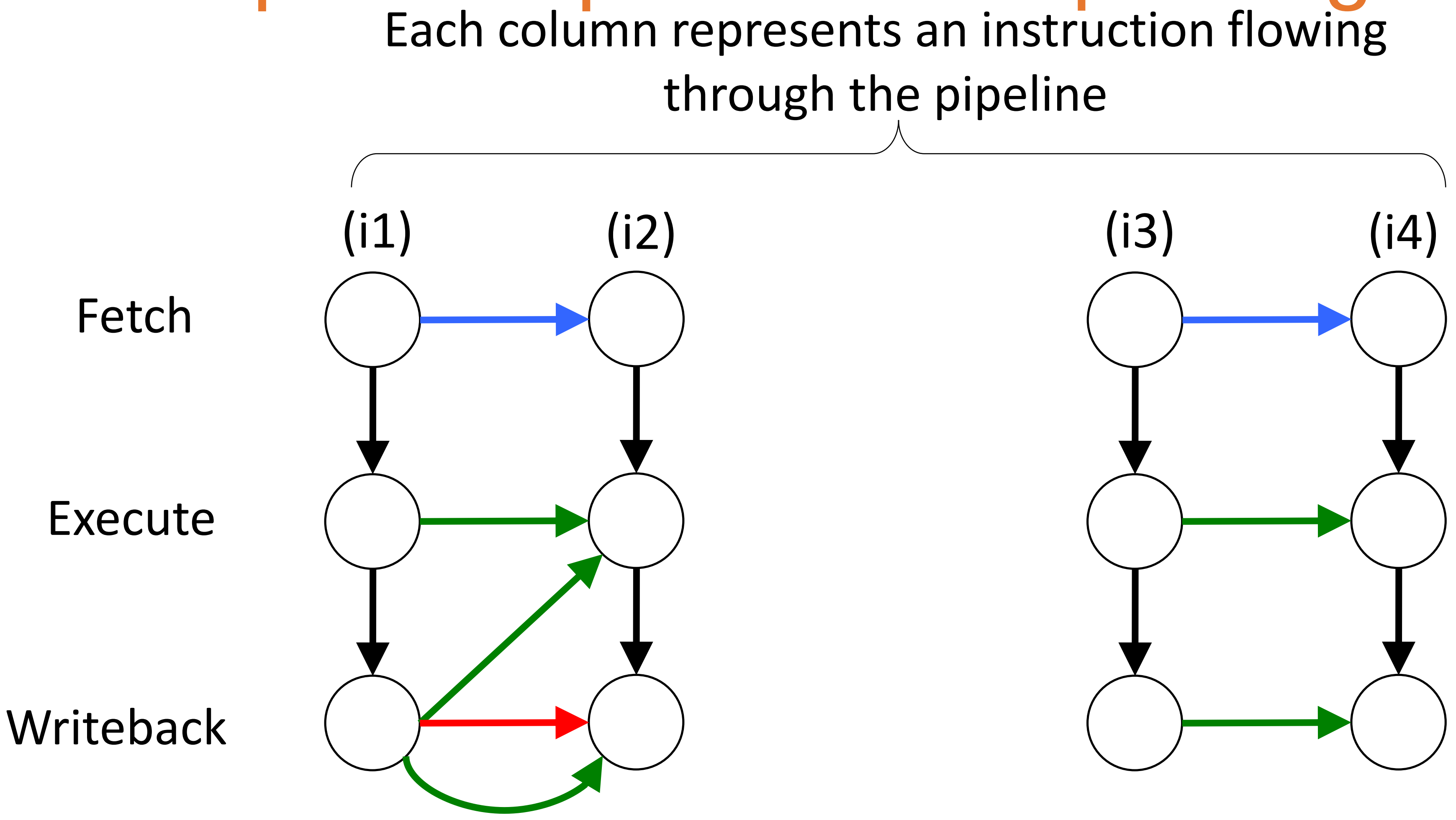
Putting the Macros together: the Read_Values axiom

```
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

μhb Graphs for `co-mp` Using Axioms

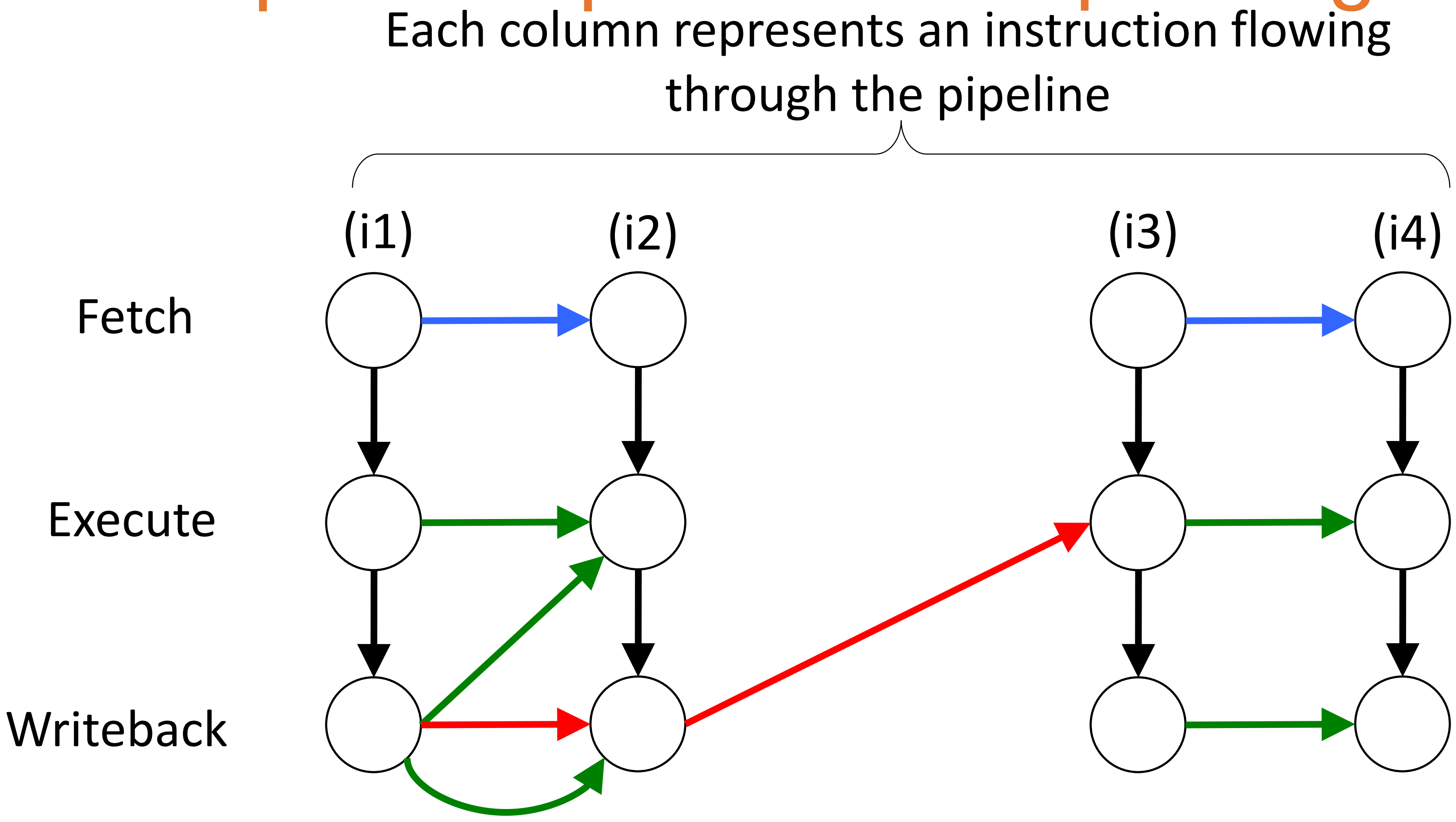


Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback

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

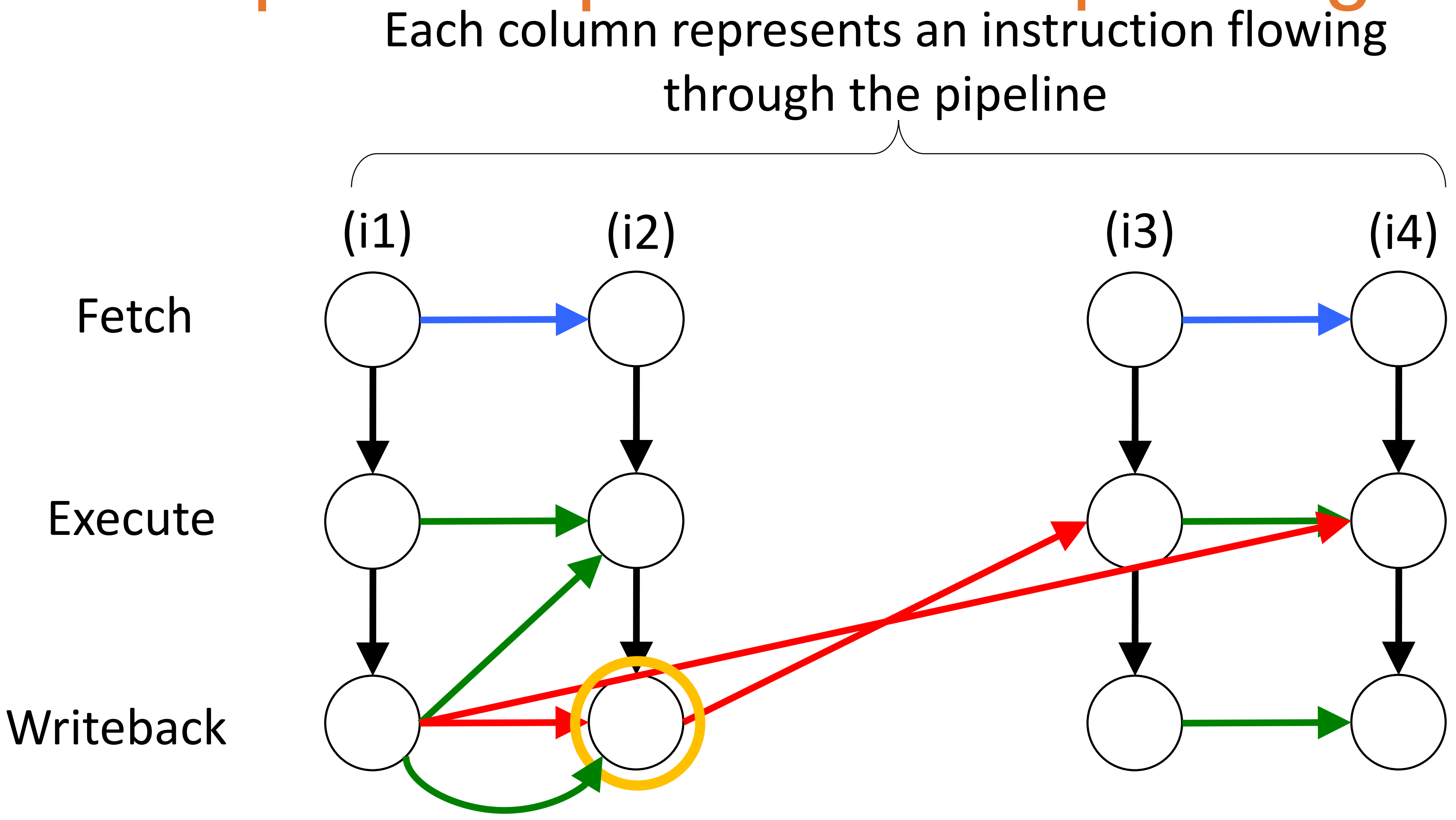

Each column represents an instruction flowing through the pipeline
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback

- 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
(i1)
(i2)
(i3)
(i4)
Fetch
Execute
Writeback

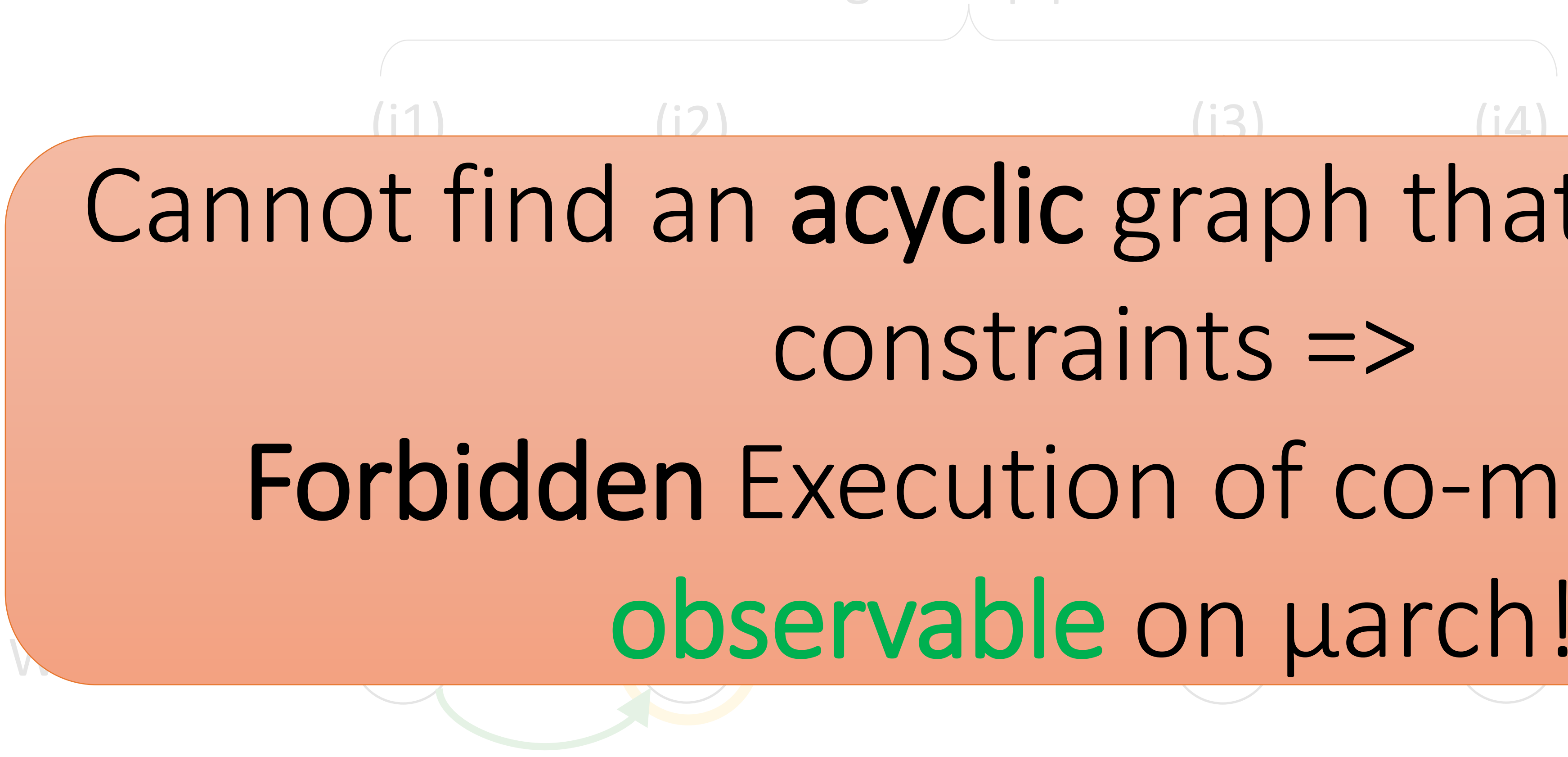
- i4 must be sourced from i1
- But i2 intervenes! => **Constraint unsatisfiable**

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



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

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

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	

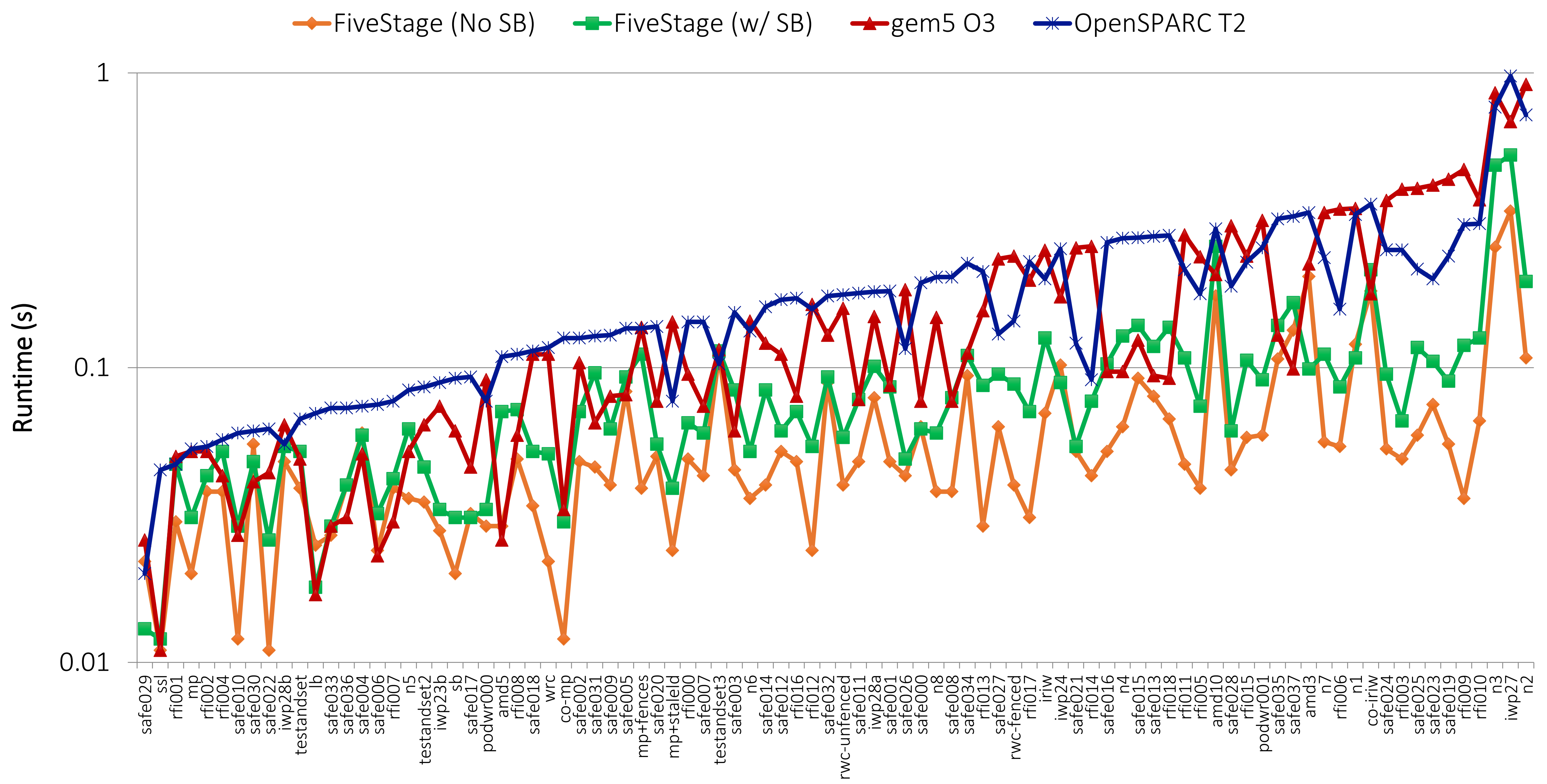
Test your completed SC uarch!

```
# 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!

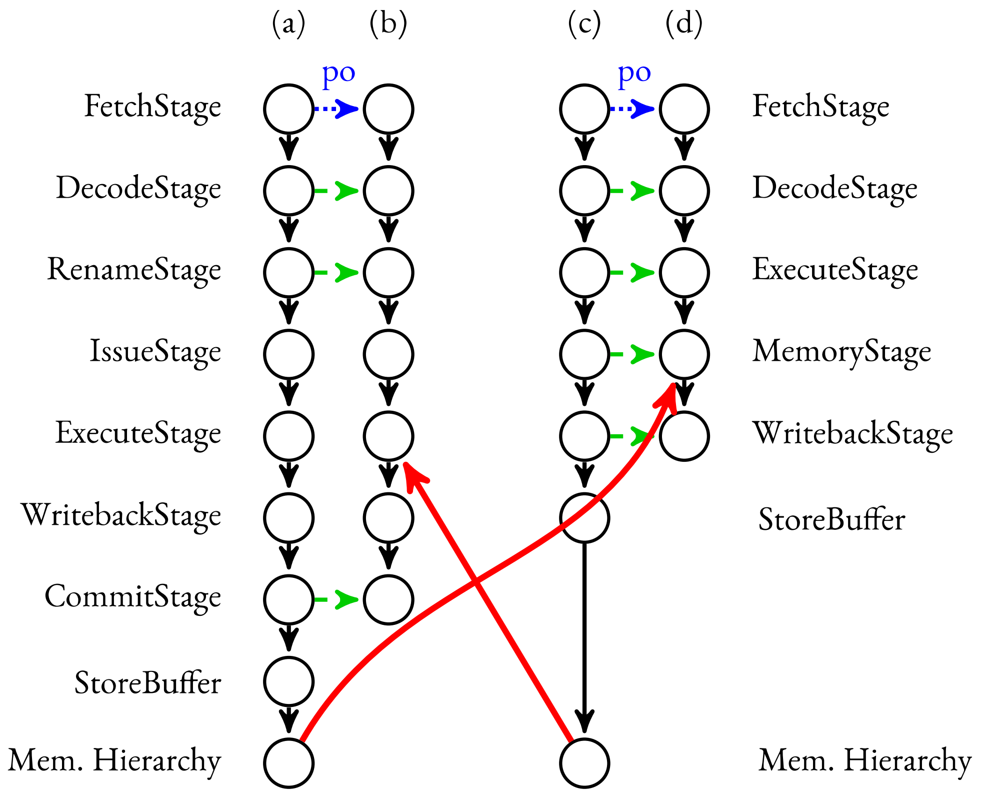
```
# Assuming you are in ~/pipecheck_tutorial/uarches/
$ run_tests –v 2 -t ../tests/SC_tests/ -m SC_fillable.uarch

# 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
```

PipeCheck Verification Time



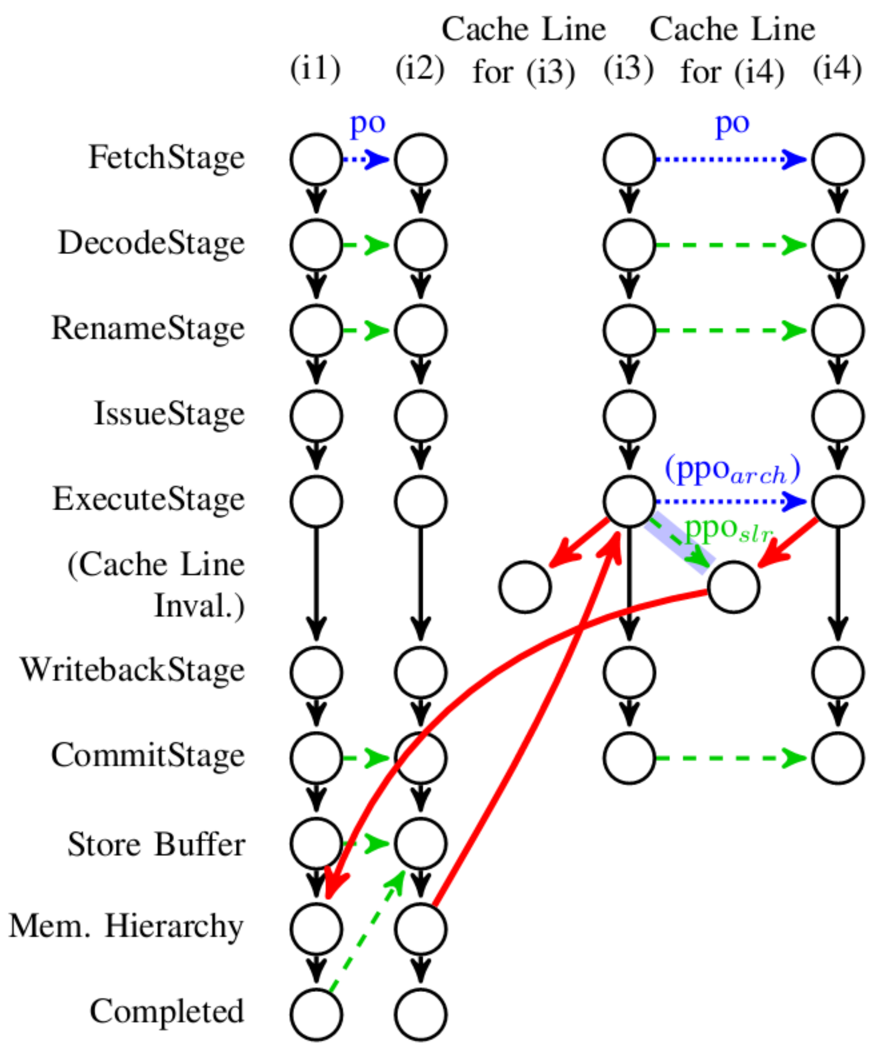
Covered the basics of what PipeCheck can do…

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



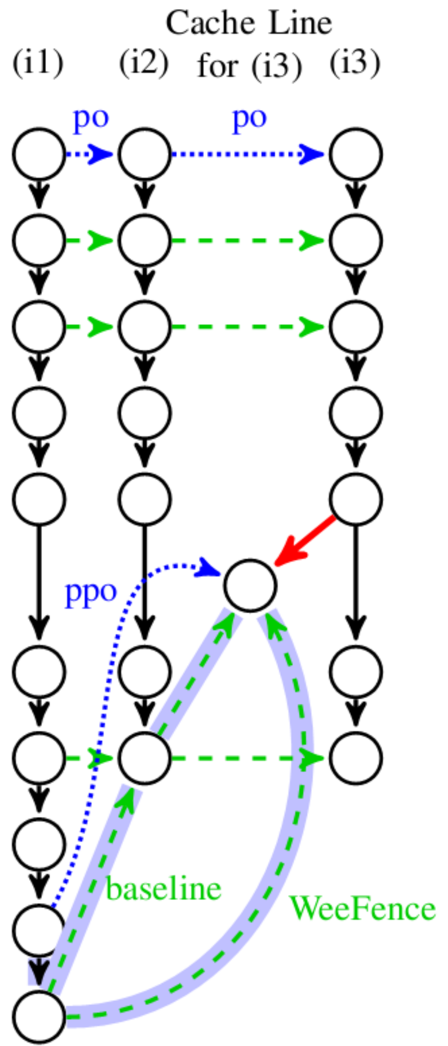
Covered the basics of what PipeCheck can do…

- …and microarchitectural optimizations…



Left: Speculative Load Reordering

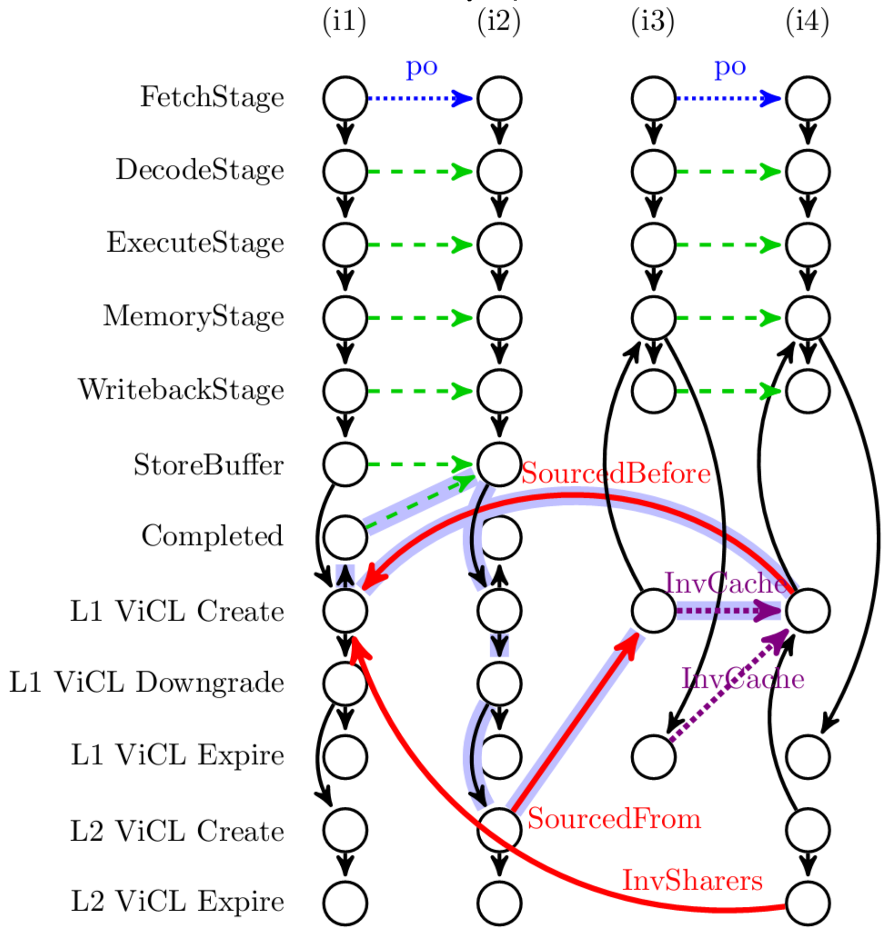
Right: Speculative Fence Retirement



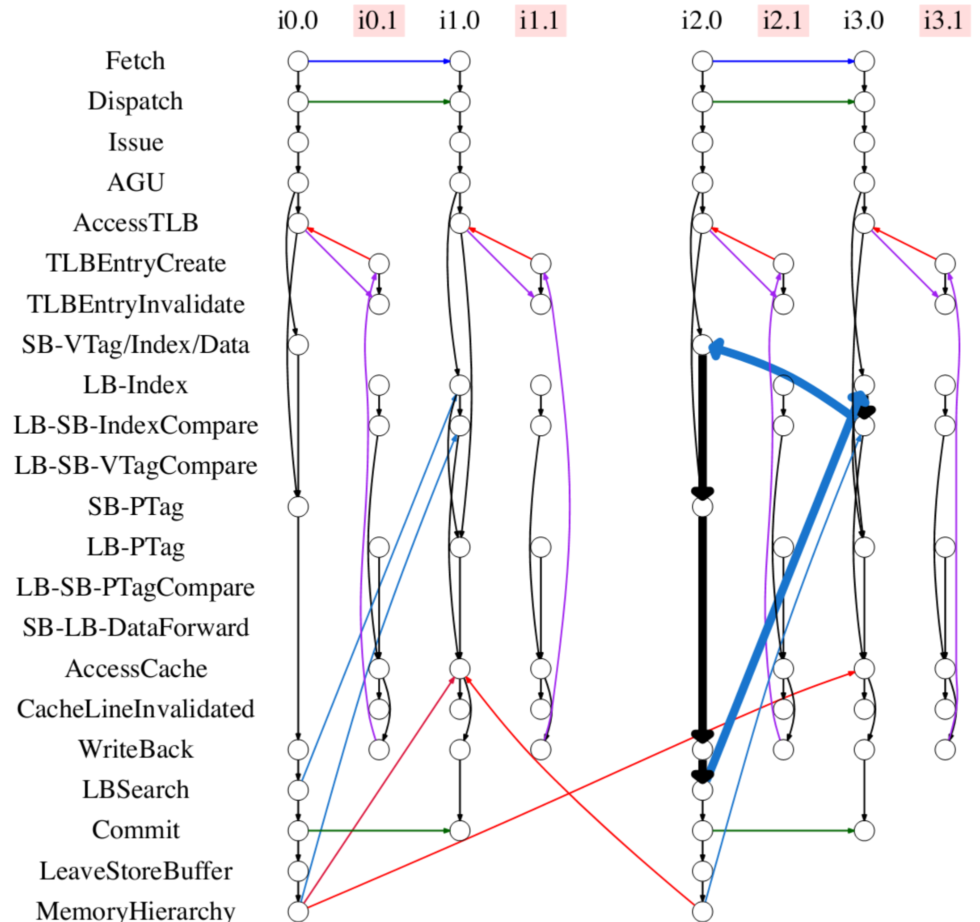
Covered the basics of what PipeCheck can do…

- …and the methodology is extensible to other ordering types, including…

<u>CCICheck:</u> Coherence orderings that affect consistency (with ViCL abstraction)



<u>COATCheck:</u> Addr Translation/Virtual Memory orderings that affect consistency



PipeCheck Summary

- Fast, automated per-program verification
- Check implementation against ISA spec
- Decompose RTL verification into smaller per-axiom sub-problems
  - More on that after the coffee break with RTLCheck!
- Open-Sourced:

https://github.com/daniellustig/coatcheck

Repo from this tutorial:

https://github.com/ymanerka/pipecheck_tutorial