Operational Semantics of Concurrent Assembly

Shaked Flur

Collaboration with: Peter Sewell
Christopher Pulte Susmit Sarkar
Luc Maranget and others

What Does this Program Do?

```
MOV X2,#1
STR X2,[X0]
STR X2,[X1]
```

What Does this Program Do?

Thread 0	Thread 1
`MOV X2,#1`	`LDR X3,[X1]`
`STR X2,[X0]`	`LDR X4,[X0]`
`STR X2,[X1]`	

Outline of the Talk

What's wrong with the reference manual?

Why operational semantics?

Why concurrency is hard?

An operational model for ARMv8

Further abstractions

TPHOLs 2009 A better x86 memory model: x86-TSO. Scott Owens, Susmit Sarkar, and Peter Sewell.

PLDI 2011 Understanding POWER Multiprocessors. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams.

POPL 2016 Modelling the ARMv8 architecture, operationally: concurrency and ISA. Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell.

POPL 2017 Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell.

POPL 2018 Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell.

POPL 2019 ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell.

POPL 2020 ARMv8-A system semantics: instruction fetch in relaxed architectures. Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell.

ARM Architecture Reference Manual is 7900 pages!!!

Informal, ambiguous, imprecise, incomplete...

$\times 1.58$

Formal Semantics

The architecture can be decomposed to two main parts:

- Instruction semantics (sequential execution)
- Memory model (multi-threading)

The architecture can be decomposed to two main parts:

- Instruction semantics (sequential execution)
- Memory model (multi-threading)

The ARM manual:

```
bits(datasize) result;
result = Zeros();
result<pos+15:pos> = imm16;
X[d] = result;
```

Sail:

```
(bit['R]) result := 0;
result := Zeros();
result[(pos+15)..pos] := imm16;
wX(d) := result;
```

MOV X2,#1

The architecture can be decomposed to two main parts:

- Instruction semantics (sequential execution)
- Memory model (multi-threading)

The ARM manual:

```
bits(datasize) result;
result = Zeros();
result<pos+15:pos> = imm16;
X[d] = result;
```

Sail:

```
(bit['R]) result := 0;
result := Zeros();
result[(pos+15)..pos] := imm16;
wX(d) := result;
```

```
MOV X2,#1
```

What is the physical thing we model?

What is the physical thing we model?

- There is no prior reliable model
- We model the architect's intent
- Our model provides formal foundation for other models

What is the physical thing we model?

- There is no prior reliable model
- We model the architect's intent
- Our model provides formal foundation for other models

How do we make our model reliable?

What is the physical thing we model?

- There is no prior reliable model
- We model the architect's intent
- Our model provides formal foundation for other models

How do we make our model reliable?

- Litmus tests with known results
- Soundness with respect to existing hardware

What is the physical thing we model?

- There is no prior reliable model
- We model the architect's intent
- Our model provides formal foundation for other models

How do we make our model reliable?

- Litmus tests with known results
- Soundness with respect to existing hardware
- Expressed in the same terms architects use

Styles of Semantic Definitions

Axiomatic semantics

Denotational semantics

Operational semantics

Styles of Semantic Definitions

Axiomatic semantics

- Candidate execution over memory events
- Apply a predicate to the execution
- The main construct here is relations

Denotational semantics

Operational semantics

Styles of Semantic Definitions

Axiomatic semantics

- Candidate execution over memory events
- Apply a predicate to the execution
- The main construct here is relations

Denotational semantics

Construct executions by applying rules to partially ordered multi-sets

Operational semantics

Styles of Semantic Definitions

Axiomatic semantics

- Candidate execution over memory events
- Apply a predicate to the execution
- The main construct here is relations

Denotational semantics

Construct executions by applying rules to partially ordered multi-sets

Operational semantics

- State machine
- Construct execution incrementally
- Relations emerge from the structure (not explicit)
- This is the language architects use

Axiomatic and Denotational Semantics

- No microarchitectural intuition
- Not a reliable foundation
- How do we add system-level features?
- Allow more behaviour than the architecture
- For well-behaved code
- Combinatorial explosion
- Better suited for higher-level programming languages

Operational Semantics

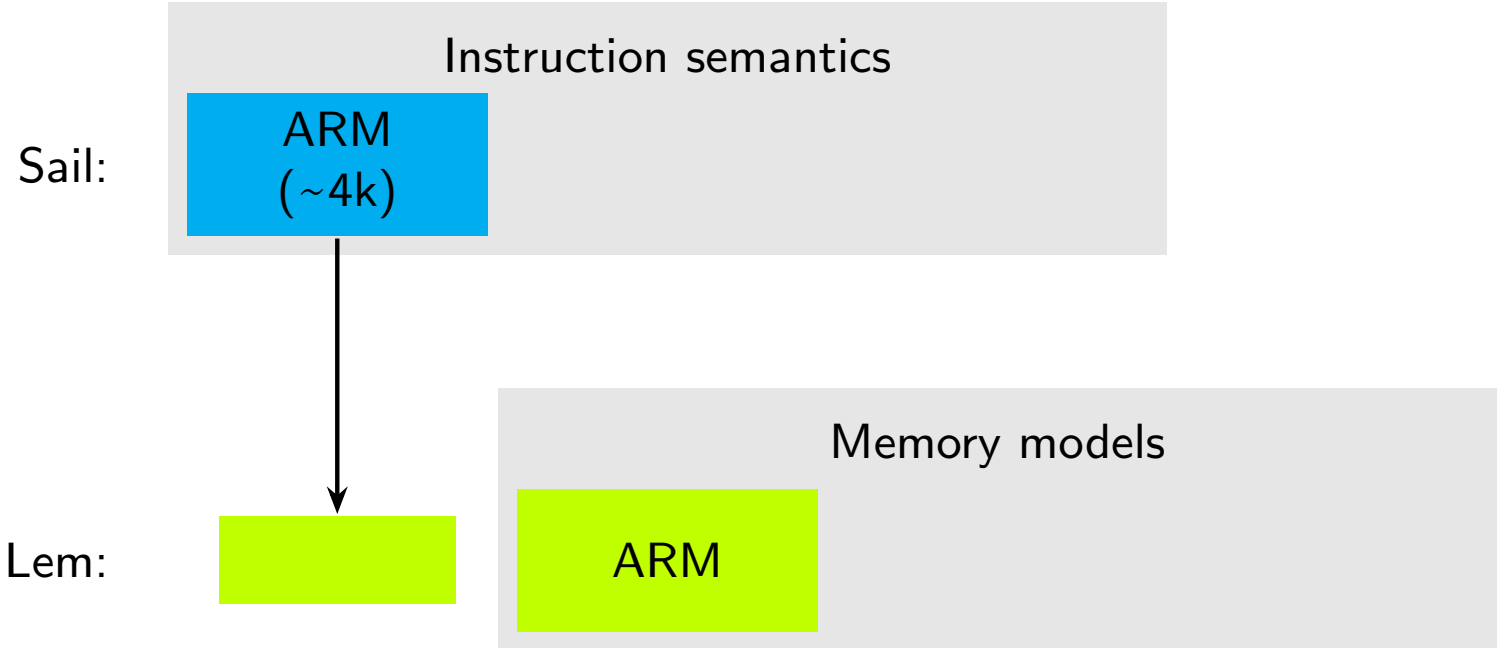
- Familiar to the architects (microarchitectural intuition)
- Captures the architects intent
- Tightly integrated with instruction semantics
- Relatively easy to extend with system-level features
- Construct valid executions incrementally

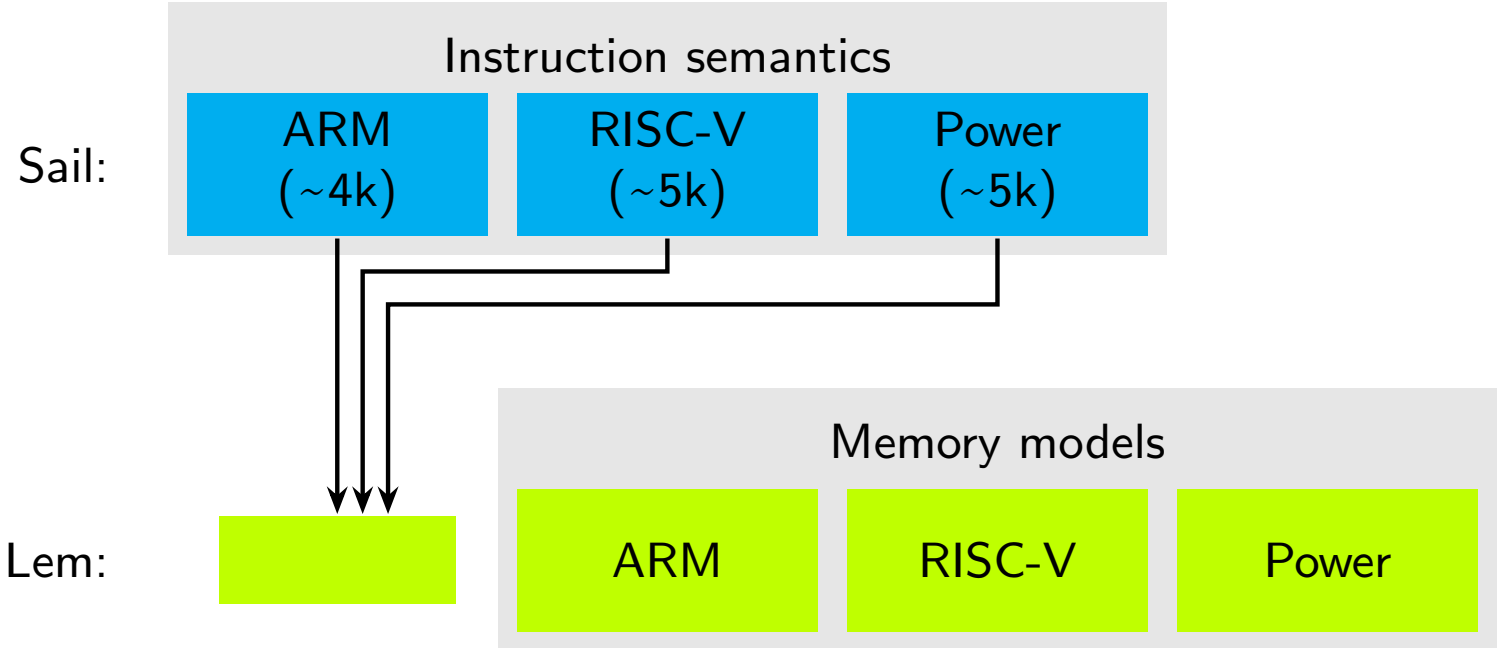
The ARM Operational Memory Model

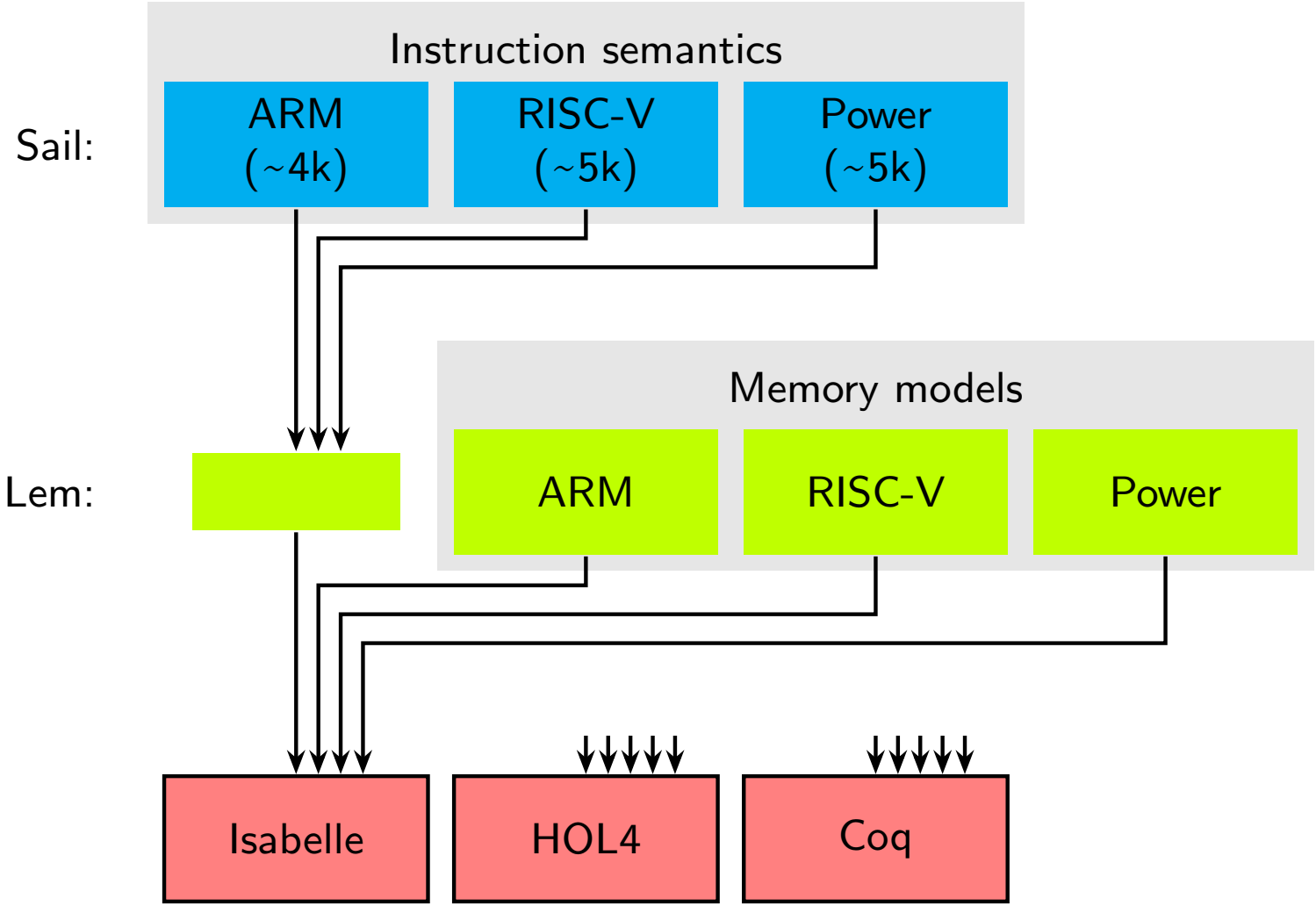
- Written in Lem
- Core functionality: ~7.5k LoC
- Utilities and other models: ~10k LoC

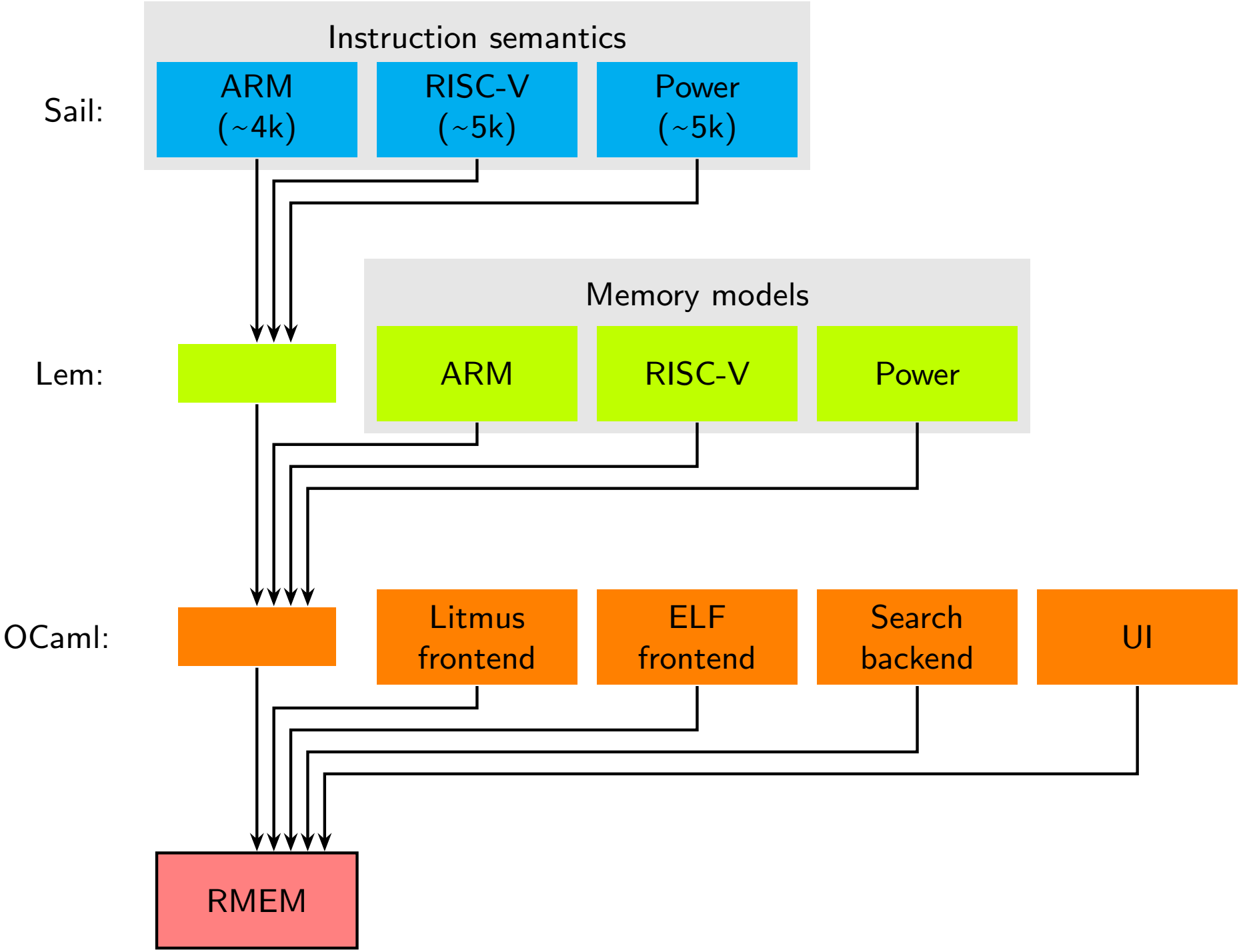
```
let reorder_pred params state feic =
  match feic.buffer_below with
  | []          -> Nothing
  | e :: below ->
      guard ((e, feic.event) NIN state.reordered) >>
      guard (reorder_pred' params feic.event e) >>
      return (e, below)
  end

let reorder_next _params state feic = fun e below ->
  let buffer =
    feic.buffer_above ++ [e; feic.event] ++ below
  in
  <| state with
    buffers =
      Map.insert feic.segment buffer state.buffers;
    reordered =
      {(feic.event, e)} union state.reordered;
  |>
```


Instruction semantics
Sail:
ARM
(~4k)
Memory models
Lem:
ARM


Instruction semantics
Sail:
ARM
(~4k)
RISC-V
(~5k)
Power
(~5k)
Memory models
Lem:
ARM
RISC-V
Power


Instruction semantics
Sail:
ARM
(~4k)
RISC-V
(~5k)
Power
(~5k)
Memory models
Lem:
ARM
RISC-V
Power
Isabelle
HOL4
Coq


Instruction semantics
Sail:
ARM (~4k)
RISC-V (~5k)
Power (~5k)
Memory models
Lem:
ARM
RISC-V
Power
OCaml:
Litmus frontend
ELF frontend
Search backend
UI
RMEM

https://www.cl.cam.ac.uk/~sf502/regressions/rmem/

RMEM r895feac40b04cf28ed416fa

cl.cam.ac.uk/~sf502/regressions/rmem/

RMEM AArch64 MP+po+ctrl | Load litmus | Load ELF | Model | Next | Back | Restart | Search | Execution | Interface | Graph | Link to this state | Help

State — 100%

Storage subsystem state (flowing):

```
0           1
|           |
|           |
|           |
--------------
      |2 (0:4:0):W 0x1000 (y)/4=1
      |
      |
```

Memory = [(1000:0:0):W 0x1000 (y)/4=0, (1000:1:0):W 0x1100 (x)/4=0]

2 flow write to memory: (0:4:0):W 0x1000 (y)/4=1

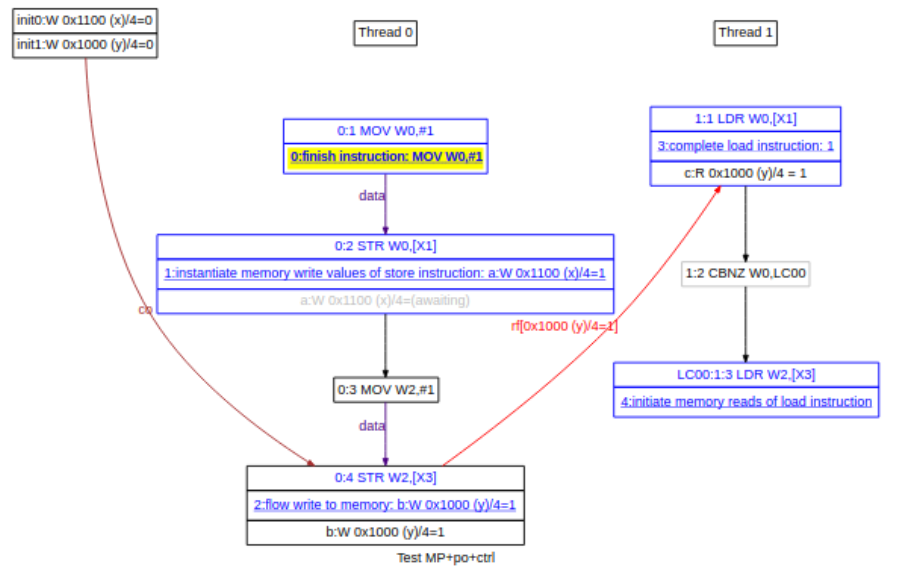
Thread 0 state:

```
read issue order: []
0:1    0x050000        MOV W0,#1           reg writes: R0=0x_63'0000000000000001
  micro_op_state: MOS_plain Done ()
  0 0:1    finish instruction: MOV W0,#1
0:2    0x050004        STR W0,[X1]         reg reads: R1=0x_63'0000000000001100 (x) from initialstate,
R0=0x_63'0000000000000001 from 0:2
  micro_op_state: MOS_plain Write_memv
  1 0:2    instantiate memory write values of store instruction: (0:2:0):W 0x1100 (x)/4=1
0:3    0x050008        MOV W2,#1           reg writes: R2=0x_63'0000000000000001
  micro_op_state: MOS_plain
0:4    0x05000c        STR W2,[X3]         mem writes: (0:4:0):W 0x1000 (y)/4=1  reg reads:
R3=0x_63'0000000000001000 (y) from initialstate, R2=0x_63'0000000000000001 from 0:4
  micro_op_state: MOS_plain
```

Thread 1 state:

```
read issue order: []
1:1    0x051000        LDR W0,[X1]         mem reads: (1:1:0):R from (0:4:0):W 0x1000 (y)/4=1  reg reads:
R1=0x_63'0000000000001000 (y) from initialstate
```

Graph — Refresh | Download .dot | centre — 100%



Sources — Download | Edit — 100%

AArch64 MP+po+ctrl

```
10 0:X1=x; 0:X3=y;
11 1:X1=y; 1:X3=x;
12 }
13  P0          | P1          ;
14  MOV W0,#1   | LDR W0,[X1] ;
15  STR W0,[X1] | CBNZ W0,LC00 ;
16  MOV W2,#1   | LC00:       ;
17  STR W2,[X3] | LDR W2,[X3] ;
18 exists
19 (1:X0=1 /\ 1:X2=0)
```

Why Concurrency is Hard?

- Computers create an illusion that single threaded code is executed sequentially
- It is not practical to do the same for multi-threaded programs
- Multi-threaded programs can observe optimisations

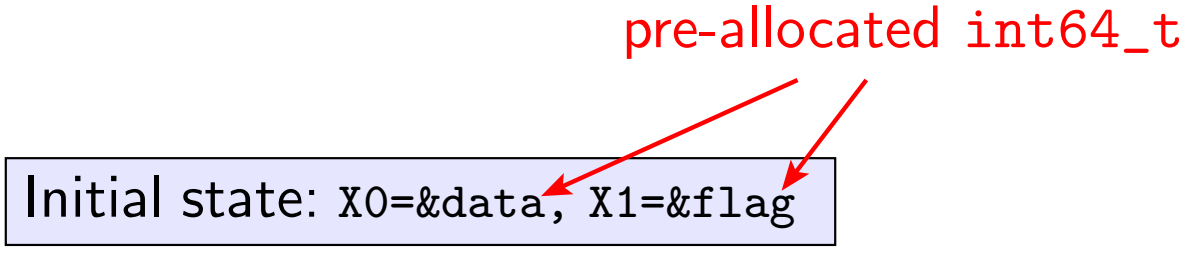
Taste of Multi-threaded Behaviour

Out of Order Execution

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Out of Order Execution

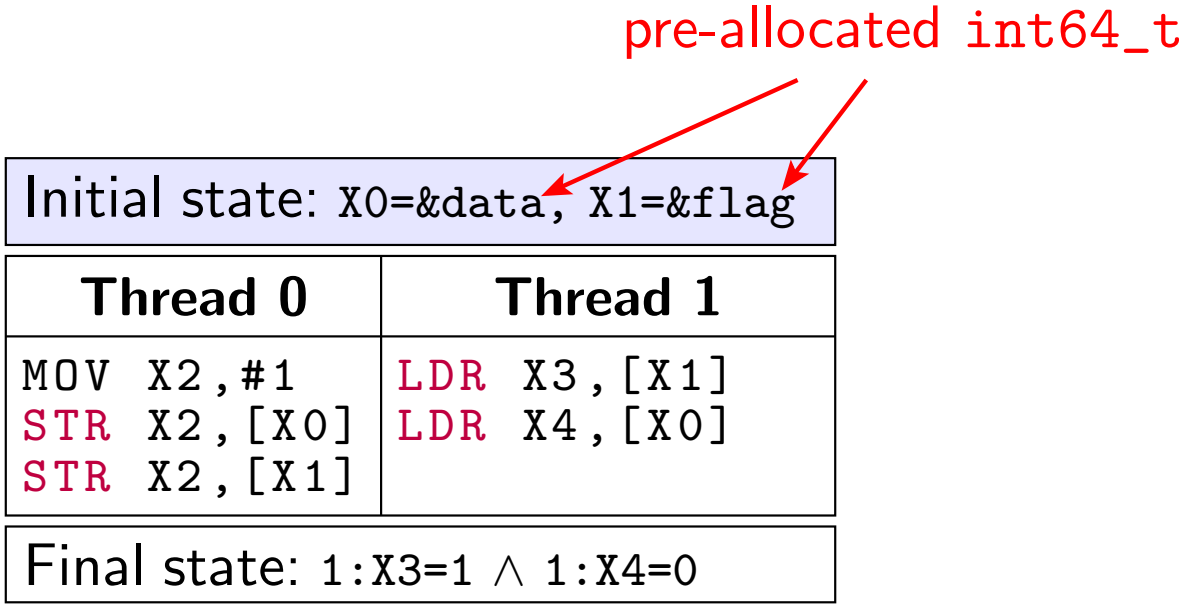


Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Out of Order Execution



Out of Order Execution

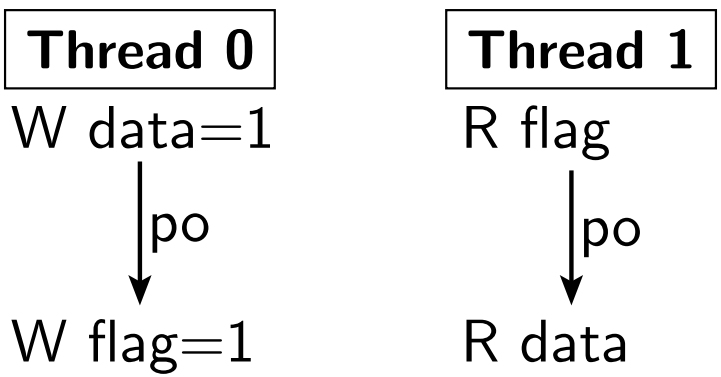
pre-allocated `int64_t`

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Final state: `1:X3=1` $\wedge$ `1:X4=0`



Out of Order Execution

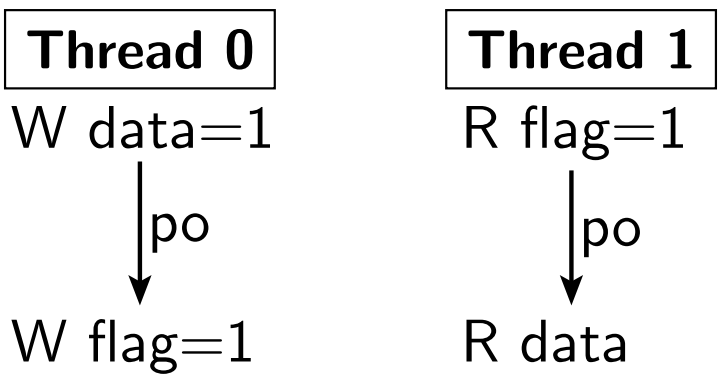
pre-allocated `int64_t`

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Final state: `1:X3=1` $\wedge$ `1:X4=0`



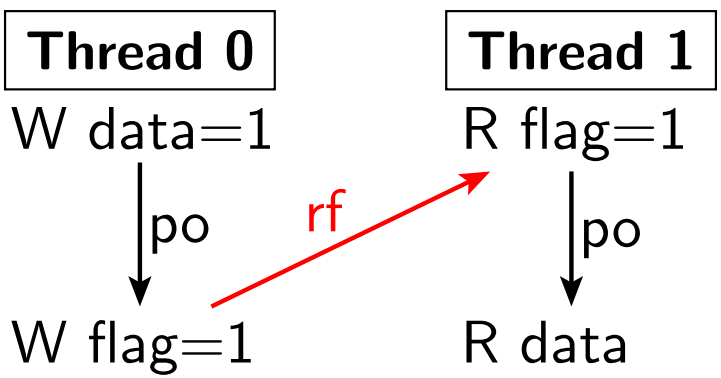
Out of Order Execution

pre-allocated `int64_t`

Initial state: `X0=&data, X1=&flag`	
Thread 0	**Thread 1**

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Final state: `1:X3=1` $\wedge$ `1:X4=0`	



Out of Order Execution

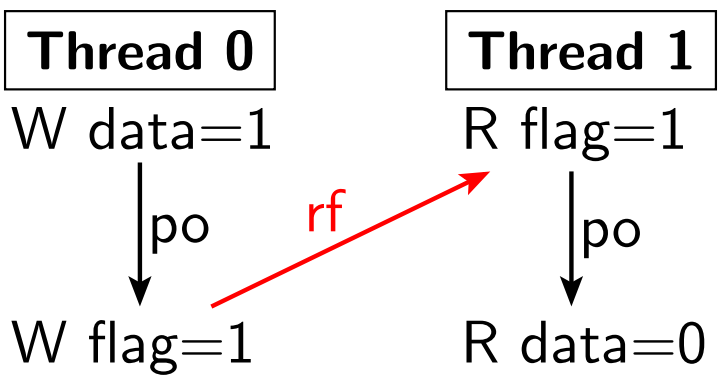
pre-allocated `int64_t`

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Final state: `1:X3=1` $\wedge$ `1:X4=0`



Out of Order Execution

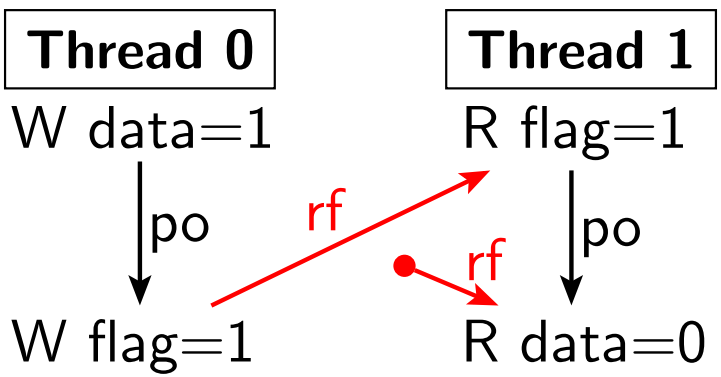
pre-allocated `int64_t`

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Final state: `1:X3=1` $\wedge$ `1:X4=0`



Out of Order Execution

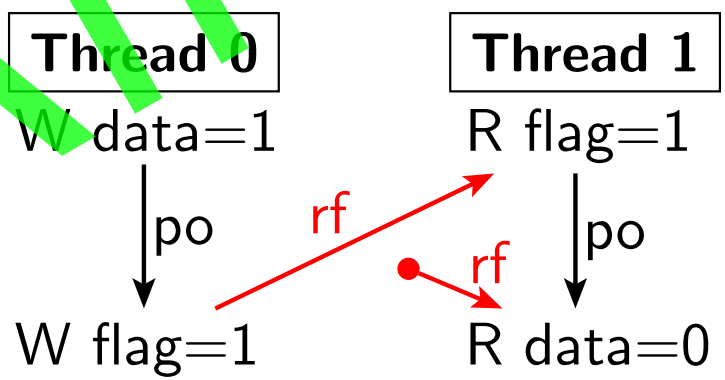
pre-allocated `int64_t`

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`LDR X4,[X0]`

Final state: `1:X3=1` $\wedge$ `1:X4=0`



Forcing In Order Execution

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`DMB SY//barrier`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`AND X4,X3,#0//address`
	`LDR X5,[X0,X4]`

Final state: `1:X3=1` $\wedge$ `1:X5=0`

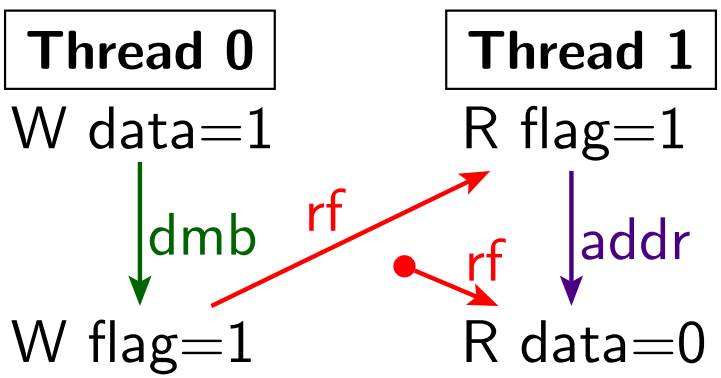
Forcing In Order Execution

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`DMB SY//barrier`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`AND X4,X3,#0//address`
	`LDR X5,[X0,X4]`

Final state: `1:X3=1` $\wedge$ `1:X5=0`



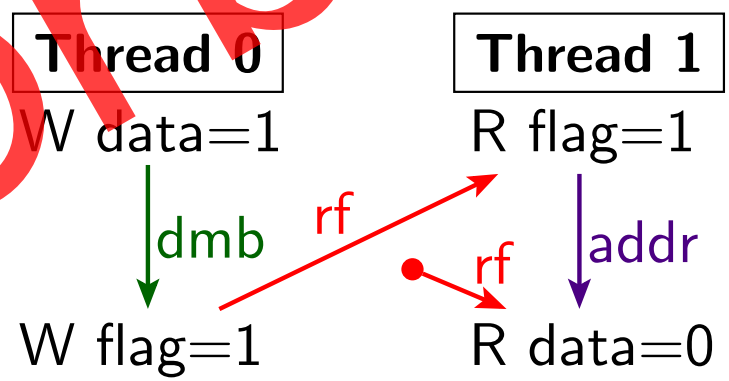
Forcing In Order Execution

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`DMB SY//barrier`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`AND X4,X3,#0//address`
	`LDR X5,[X0,X4]`

Final state: `1:X3=1 ∧ 1:X5=0`



Speculative Execution (Load)

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`DMB SY`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`CBZ X3,end//branch`
	`LDR X4,[X0]`
	`end:`

Final state: `1:X3=1` $\wedge$ `1:X4=0`

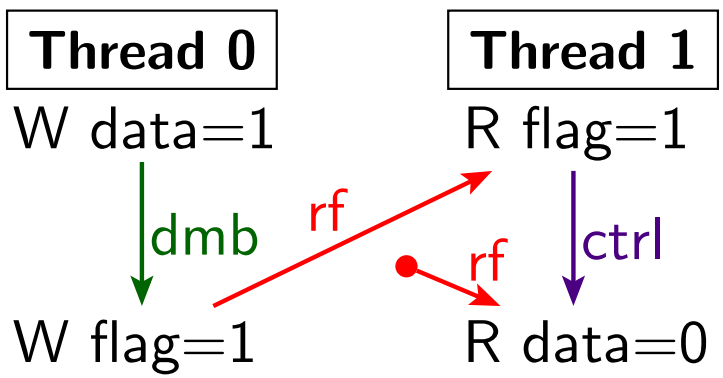
Speculative Execution (Load)

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1

`MOV X2,#1`
`STR X2,[X0]`
`DMB SY`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`CBZ X3,end//branch`
	`LDR X4,[X0]`
	`end:`

Final state: `1:X3=1` $\wedge$ `1:X4=0`



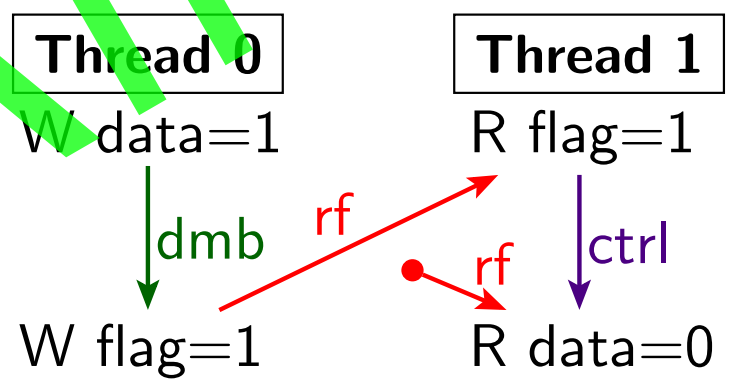
Speculative Execution (Load)

Initial state: `X0=&data, X1=&flag`

Thread 0	Thread 1
`MOV X2,#1`	`LDR X3,[X1]`
`STR X2,[X0]`	`CBZ X3,end//branch`
`DMB SY`	`LDR X4,[X0]`
`STR X2,[X1]`	`end:`

Final state: `1:X3=1 /\ 1:X4=0`

Allowed



Speculative Execution (Store)

Initial state: `X0=&a, X1=&b`

Thread 0	Thread 1

`LDR X2,[X0]`
`DMB SY`
`MOV X3,#1`
`STR X3,[X1]`
	`LDR X2,[X1]`
	`CBZ X2,end`
	`MOV X3,#1`
	`STR X3,[X0]`
	`end:`

Final state: `0:X2=1` $\wedge$ `1:X2=1`

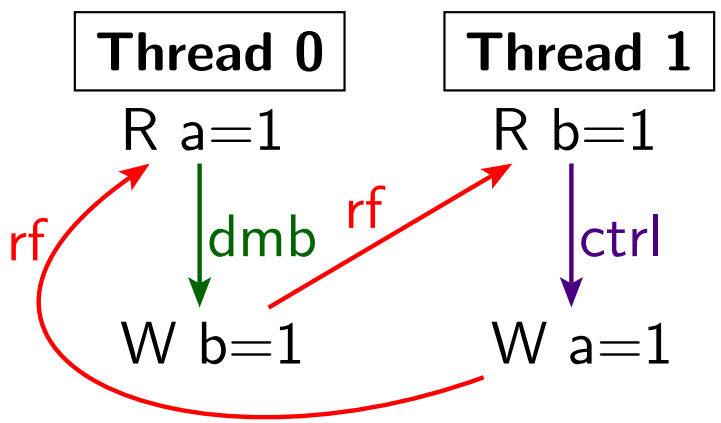
Speculative Execution (Store)

Initial state: `X0=&a, X1=&b`

Thread 0	Thread 1

`LDR X2,[X0]`
`DMB SY`
`MOV X3,#1`
`STR X3,[X1]`
	`LDR X2,[X1]`
	`CBZ X2,end`
	`MOV X3,#1`
	`STR X3,[X0]`
	`end:`

Final state: `0:X2=1 ∧ 1:X2=1`

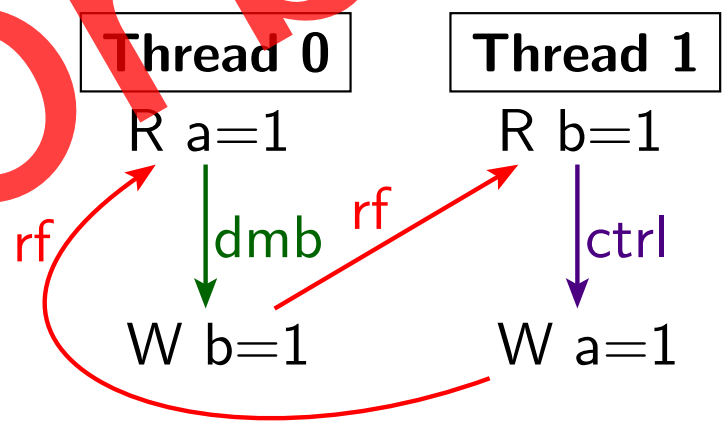


Speculative Execution (Store)

Initial state: `X0=&a, X1=&b`

Thread 0	Thread 1
`LDR X2,[X0]`	`LDR X2,[X1]`
`DMB SY`	`CBZ X2,end`
`MOV X3,#1`	`MOV X3,#1`
`STR X3,[X1]`	`STR X3,[X0]`
	`end:`

Final state: `0:X2=1 ∧ 1:X2=1`



Write Forwarding

Initial state: X0=&data, X1=&flag, X2=&temp

Thread 0	Thread 1

MOV X3,#1
STR X3,[X0]
DMB SY
STR X3,[X1]
	LDR X3,[X1]
	CBZ X3,end
	MOV X4,#1
	STR X4,[X2]
	LDR X5,[X2]
	AND X6,X5,#0
	LDR X7,[X0,X6]
	end:

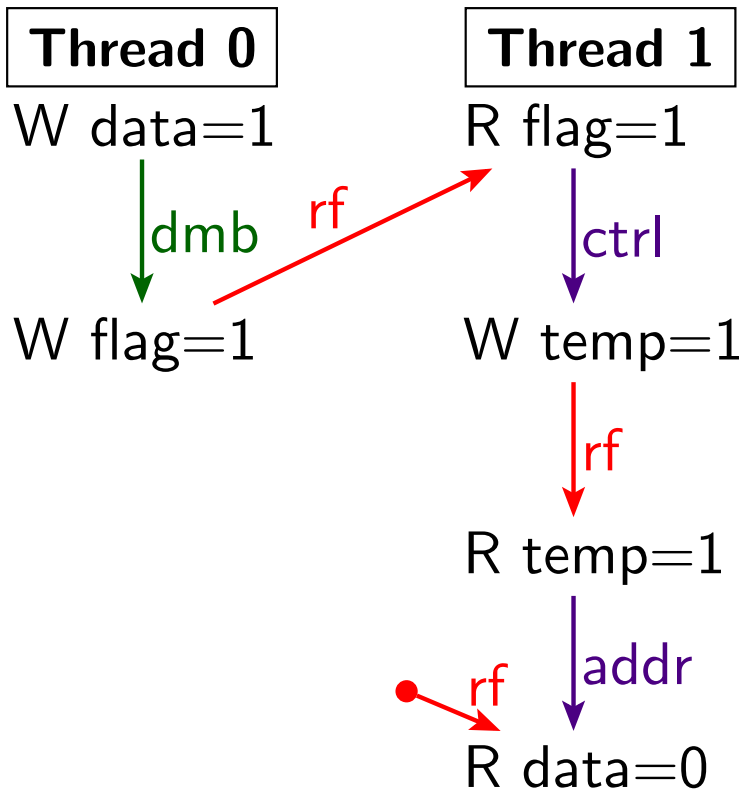
Final state: 1:X3=1 $\wedge$ 1:X5=1 $\wedge$ 1:X7=0

Write Forwarding

Initial state: `X0=&data, X1=&flag, X2=&temp`

Thread 0	Thread 1
`MOV X3,#1`	`LDR X3,[X1]`
`STR X3,[X0]`	`CBZ X3,end`
`DMB SY`	`MOV X4,#1`
`STR X3,[X1]`	`STR X4,[X2]`
	`LDR X5,[X2]`
	`AND X6,X5,#0`
	`LDR X7,[X0,X6]`
	`end:`

Final state: `1:X3=1` $\wedge$ `1:X5=1` $\wedge$ `1:X7=0`

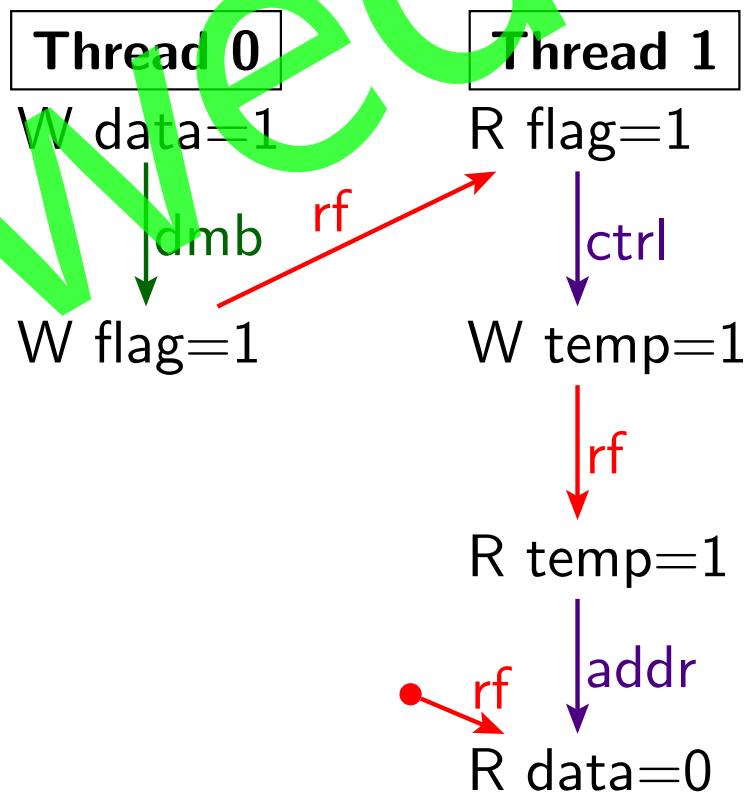


Write Forwarding

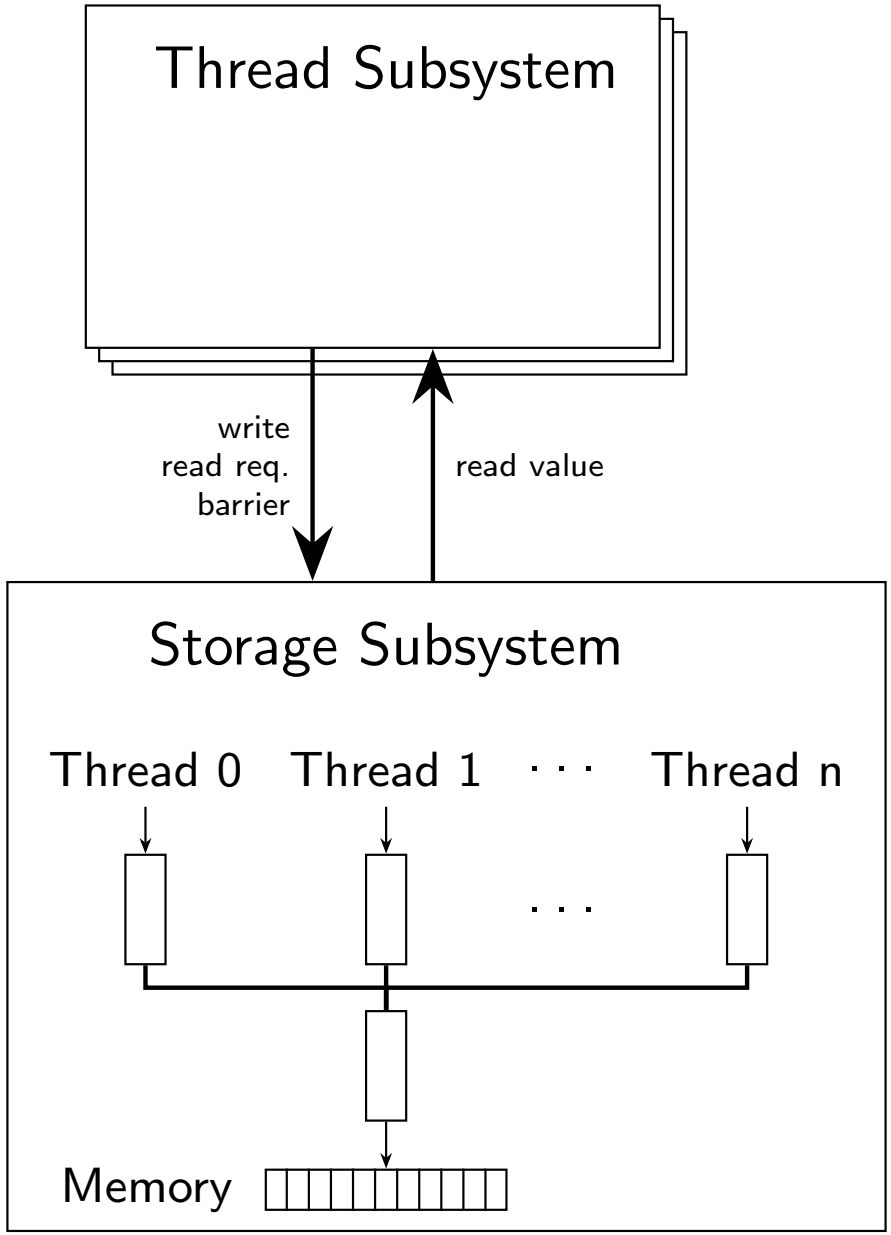
Initial state: `X0=&data, X1=&flag, X2=&temp`

Thread 0	Thread 1
`MOV X3,#1`	`LDR X3,[X1]`
`STR X3,[X0]`	`CBZ X3,end`
`DMB SY`	`MOV X4,#1`
`STR X3,[X1]`	`STR X4,[X2]`
	`LDR X5,[X2]`
	`AND X6,X5,#0`
	`LDR X7,[X0,X6]`
	`end:`

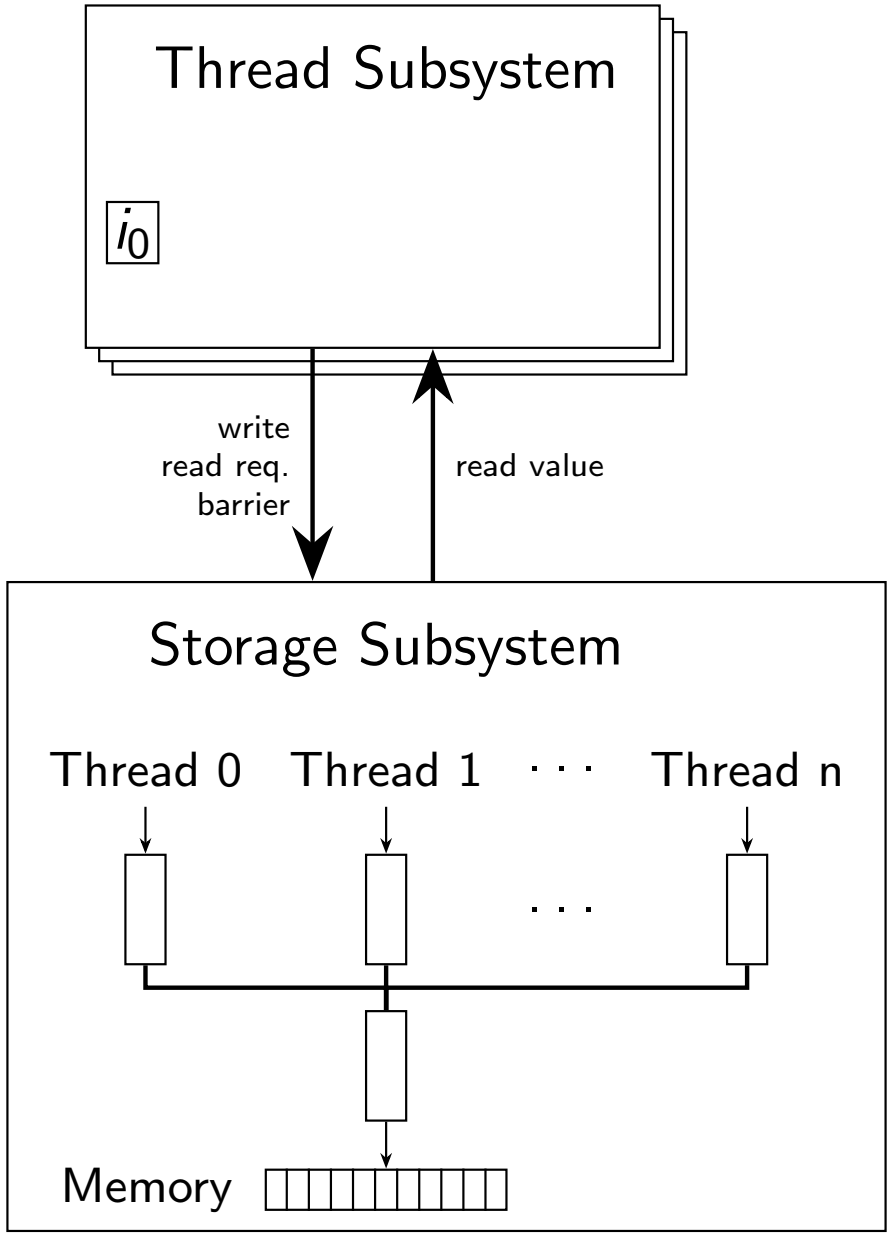
Final state: `1:X3=1 ∧ 1:X5=1 ∧ 1:X7=0`



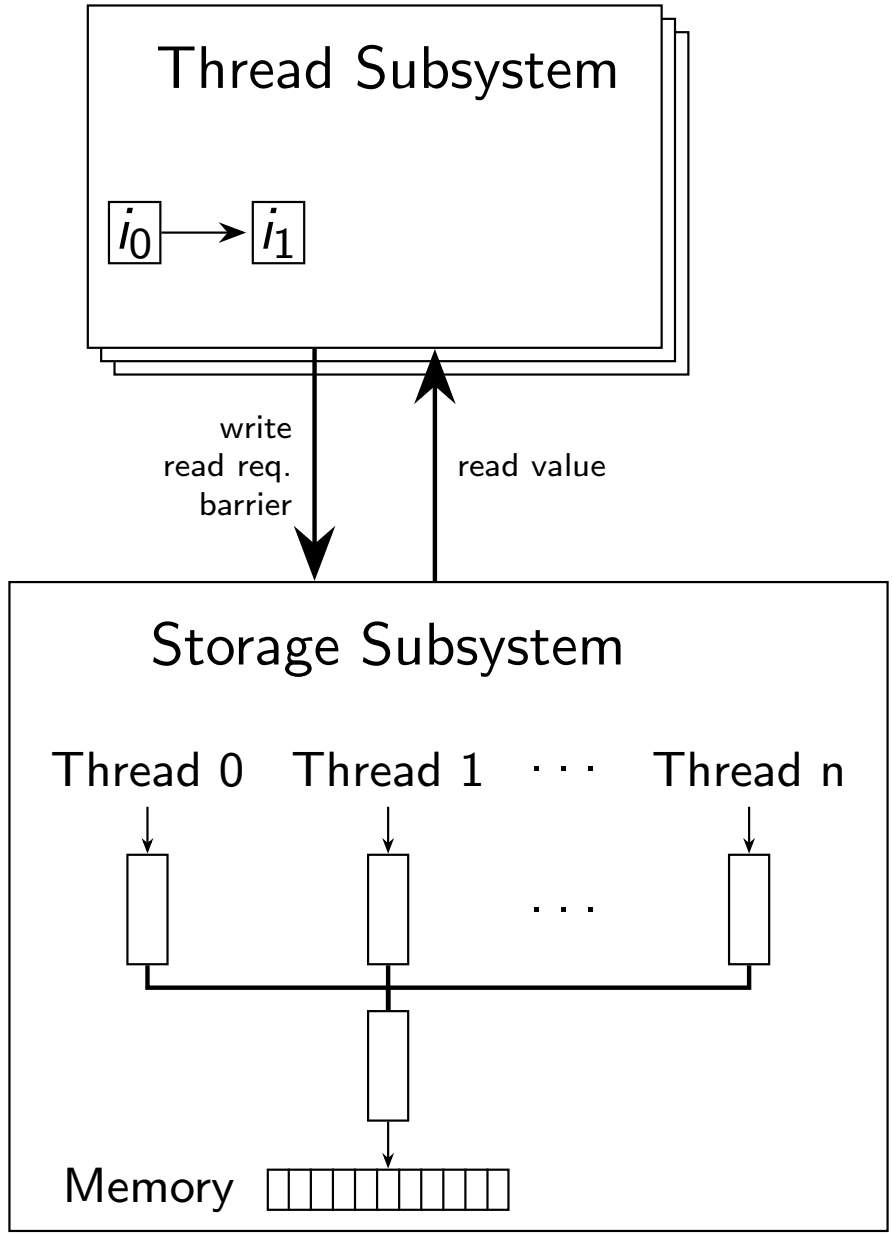
Model State


Thread Subsystem
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

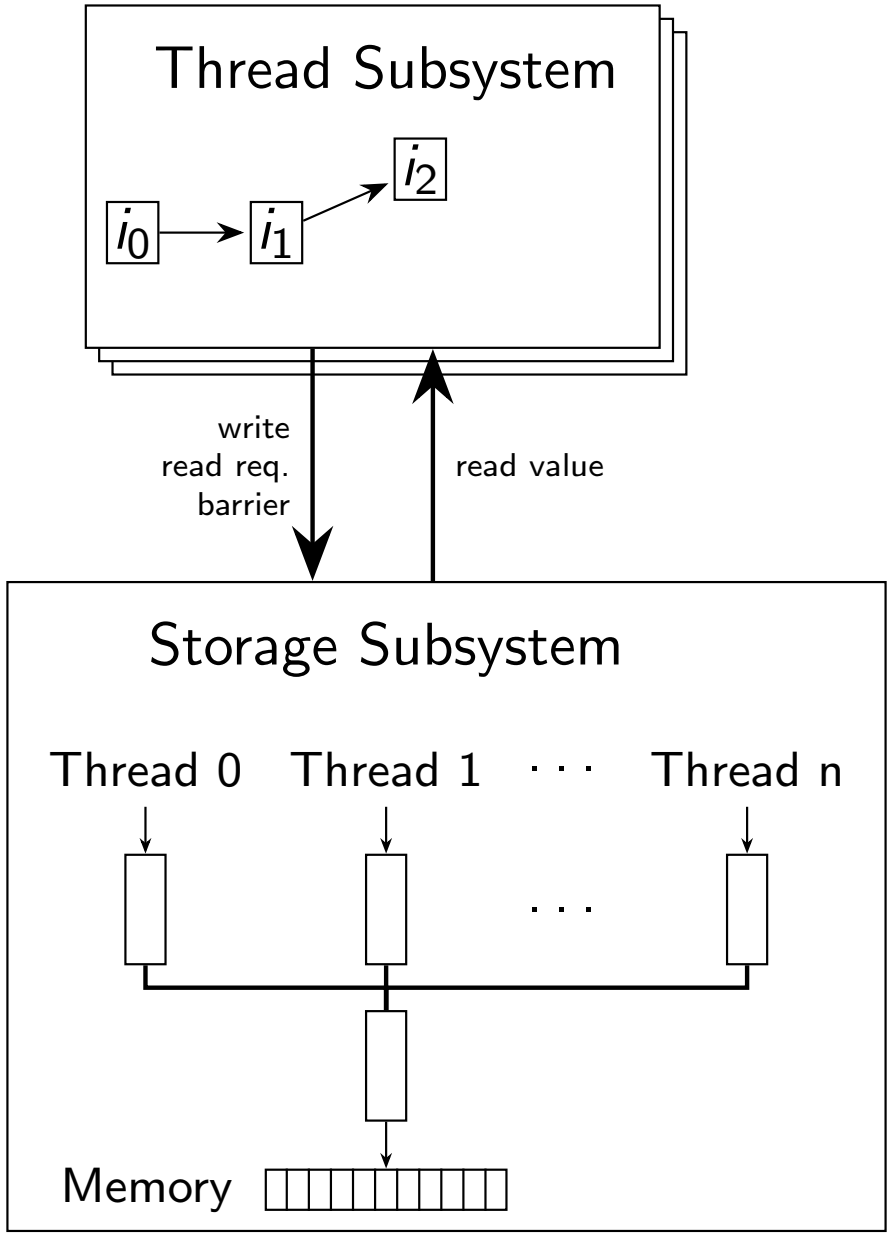
Model State


Thread Subsystem
i_0
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

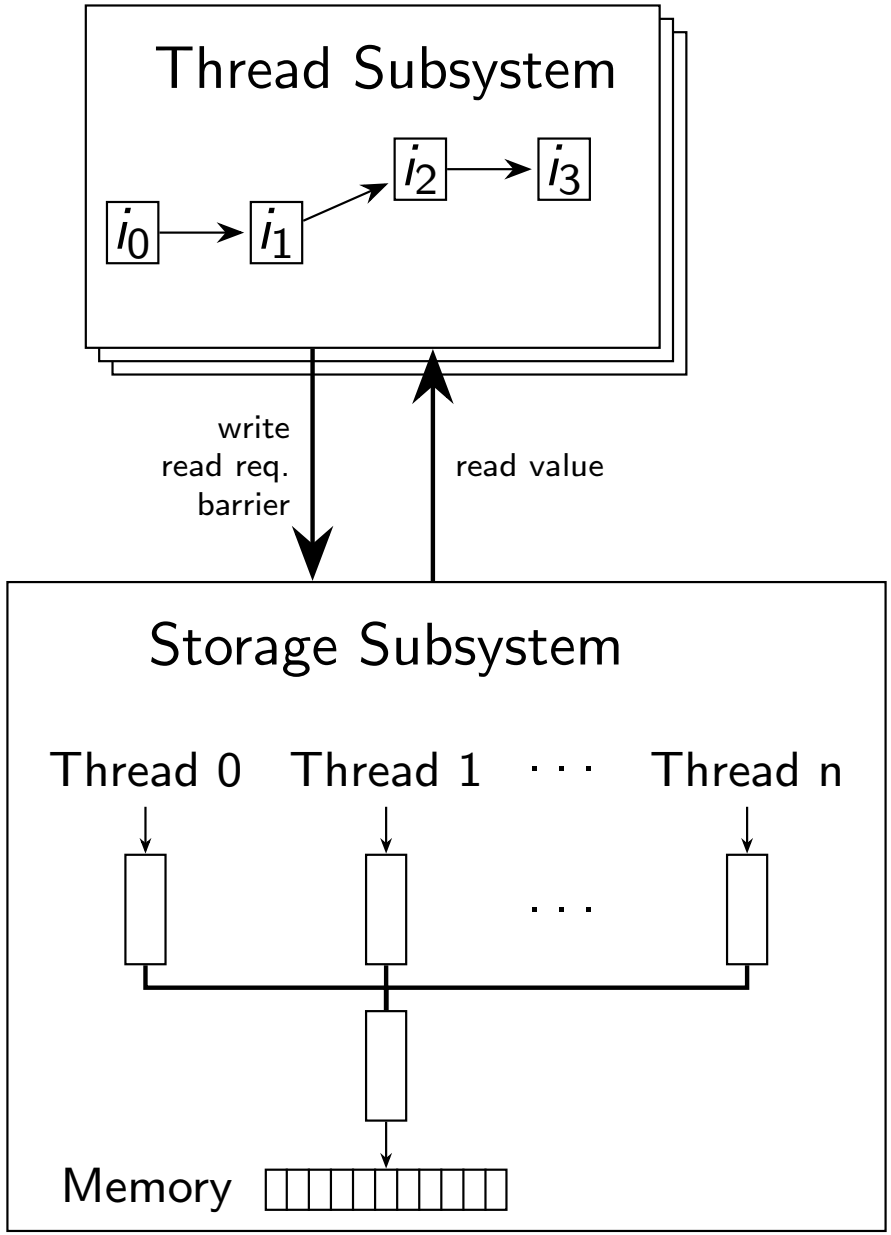
Model State


Thread Subsystem
i_0
i_1
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

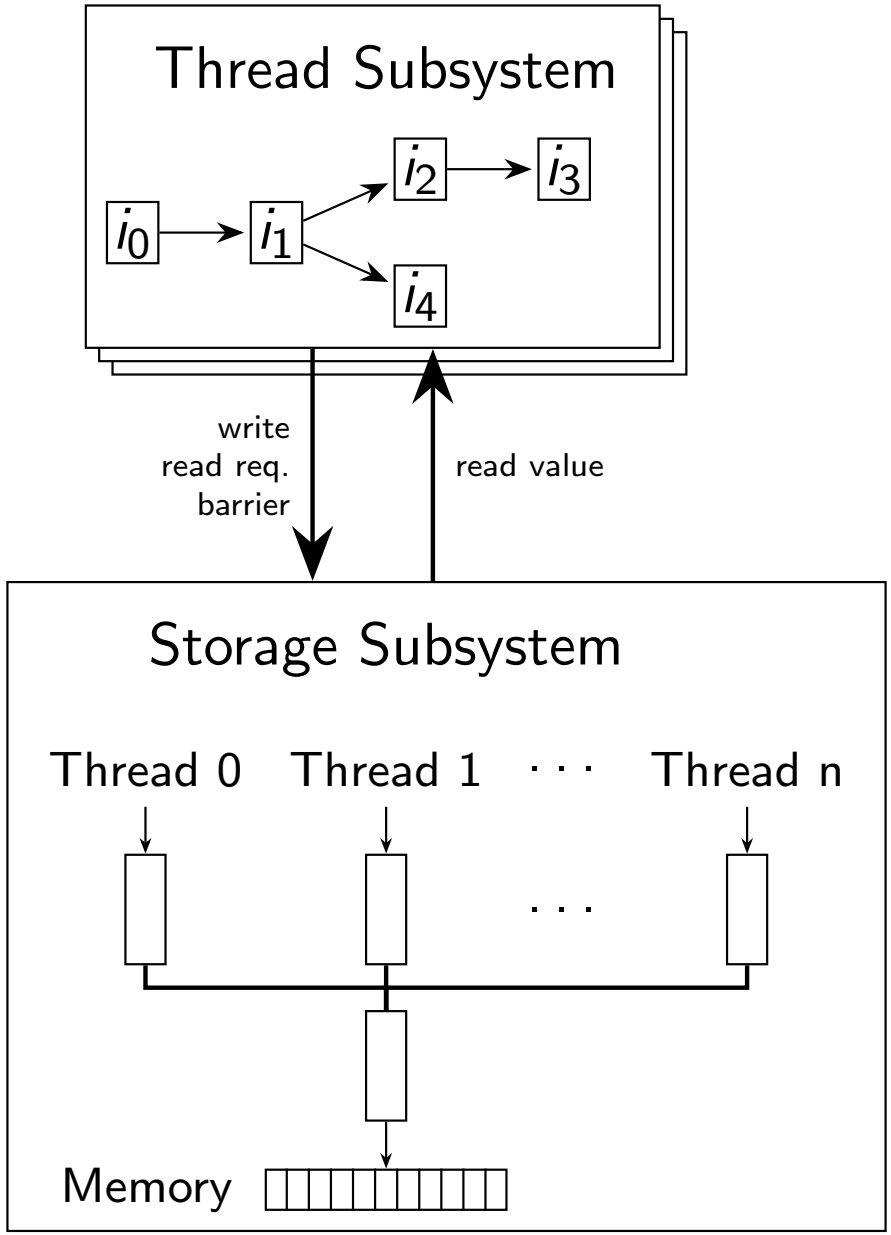
Model State


Thread Subsystem
i_0
i_1
i_2
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

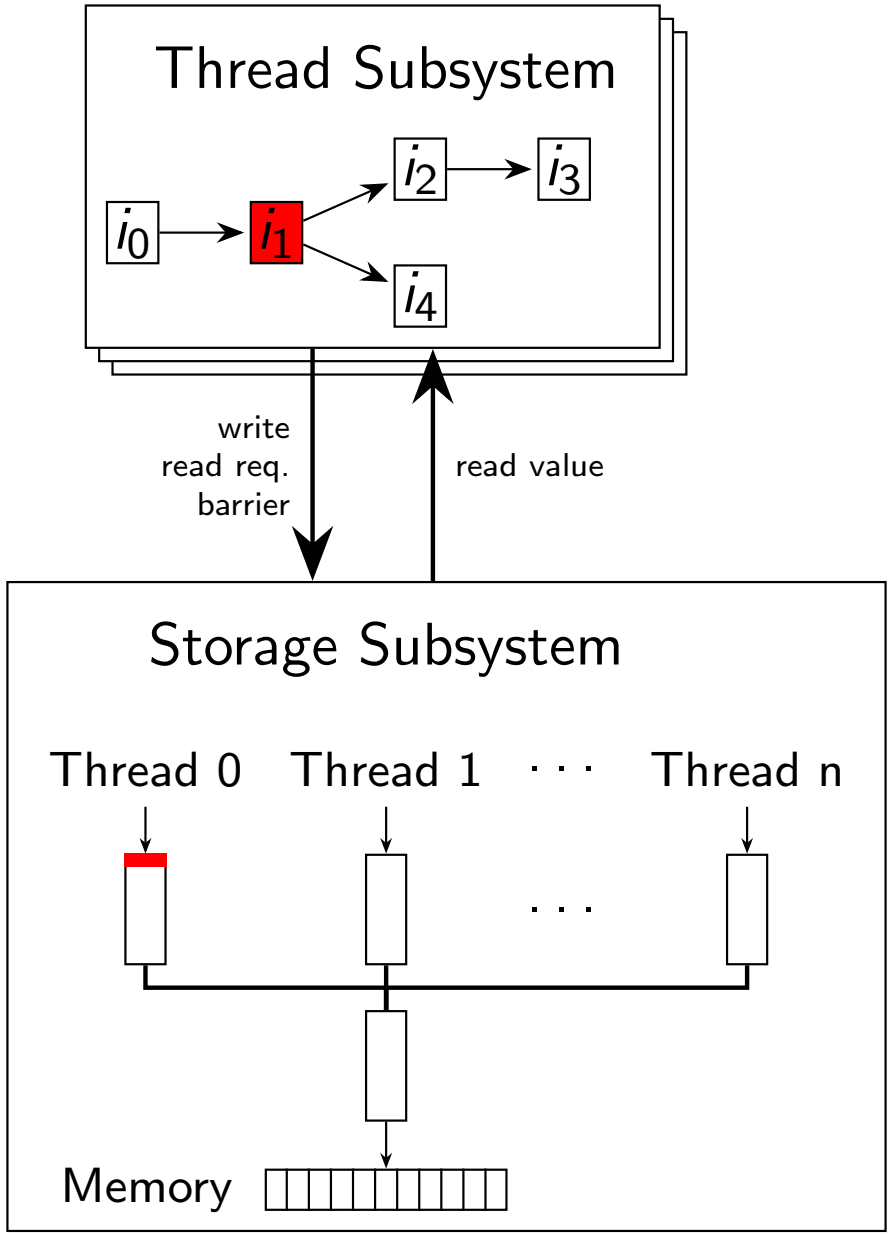
Model State


Thread Subsystem
i_0
i_1
i_2
i_3
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

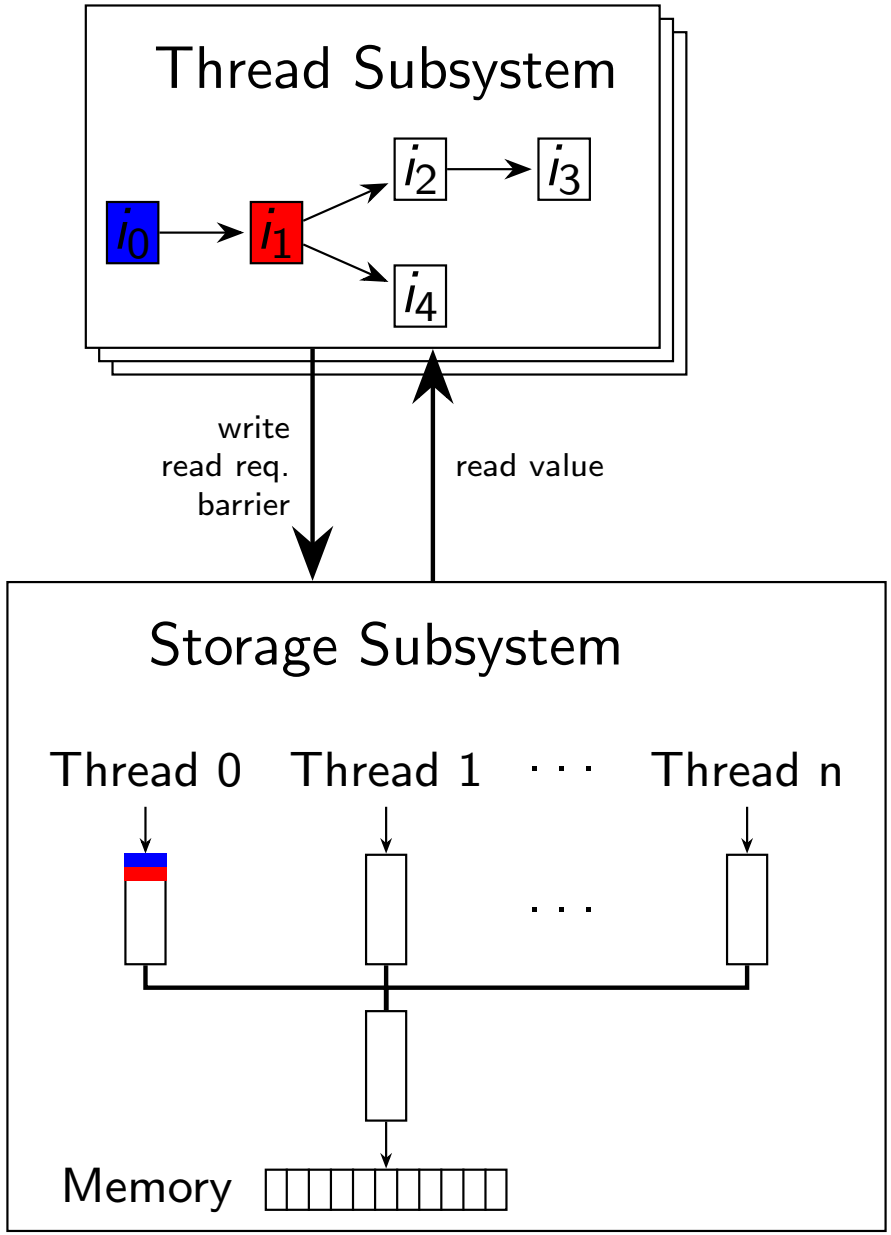
Model State


Thread Subsystem
i_0
i_1
i_2
i_3
i_4
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

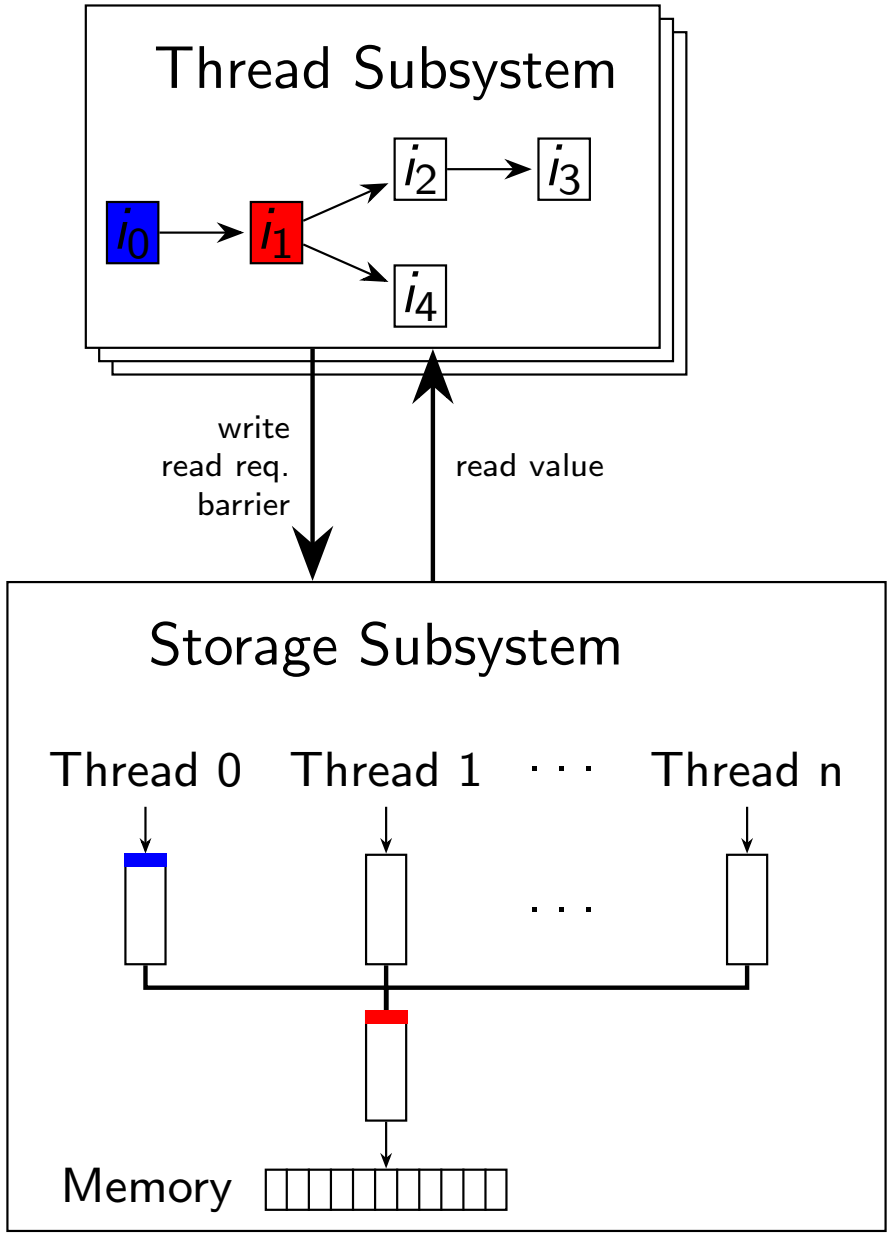
Model State


Thread Subsystem
i_0
i_1
i_2
i_3
i_4
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

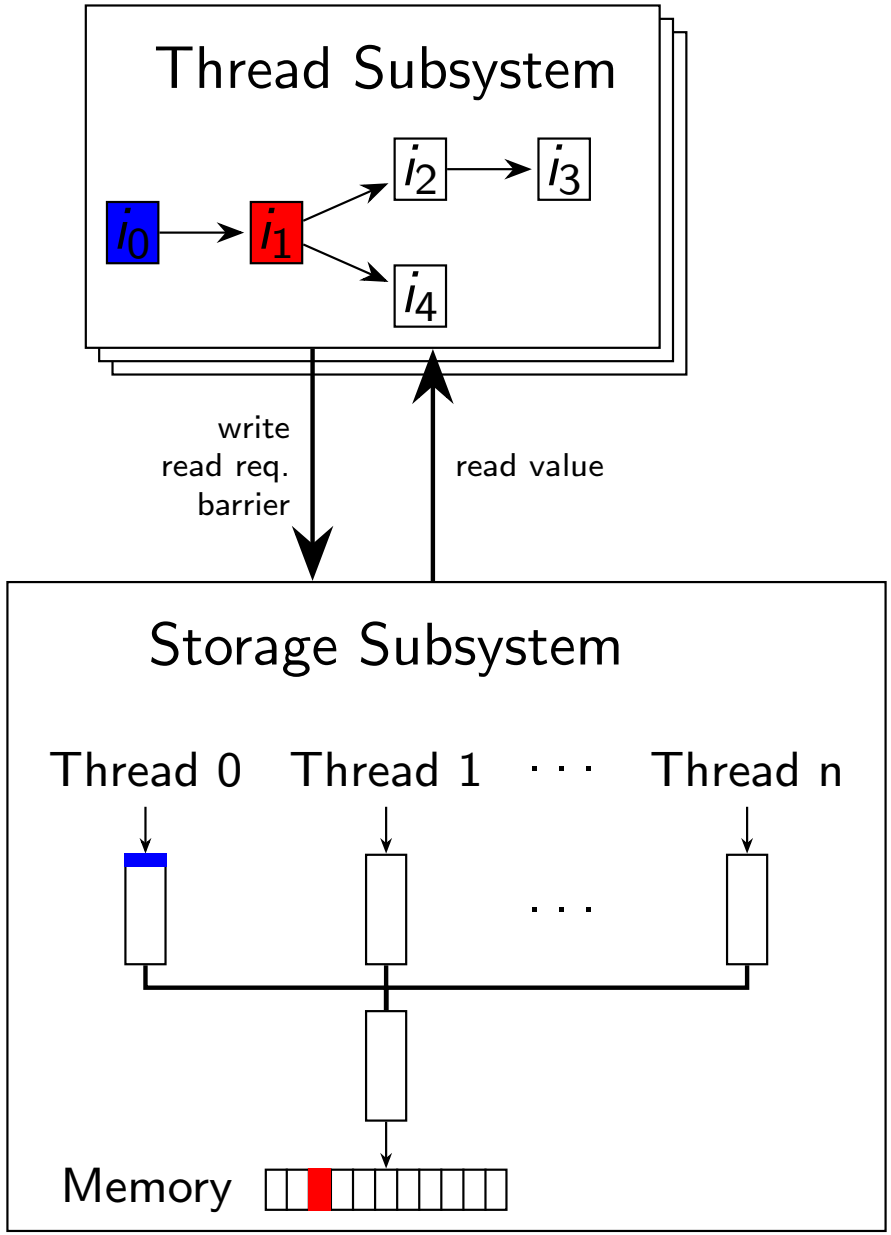
Model State


Thread Subsystem
i_0
i_1
i_2
i_3
i_4
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

Model State


Thread Subsystem
i_0
i_1
i_2
i_3
i_4
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

Model State


Thread Subsystem
i_0
i_1
i_2
i_3
i_4
write
read req.
barrier
read value
Storage Subsystem
Thread 0
Thread 1
· · ·
Thread n
· · ·
Memory

Thread transitions

Execute a store

Condition: a store can be executed if

1. all previous stores and loads to the same location have been executed;
2. all previous branches are determined; and
3. all previous barriers have been executed.

Action: add a write to the top of the buffer associated with the thread.

Execute a barrier

Condition: a barrier can be executed if

1. all previous stores and loads to the same location have been executed; and
2. all previous branches are determined;

Action: add a barrier to the top of the buffer associated with the thread.

...

Storage transitions

Flow event
The bottom most event of a non-root buffer can flow to the top of the root buffer, and the bottom most event of the root buffer can flow into memory.

Satisfy a read
A read that is adjacent to a write (read is higher) to the same location can be satisfied by the write.

Reorder events
Two adjacent memory accesses in a buffer can be reordered with each other, if they are to different memory locations.

...

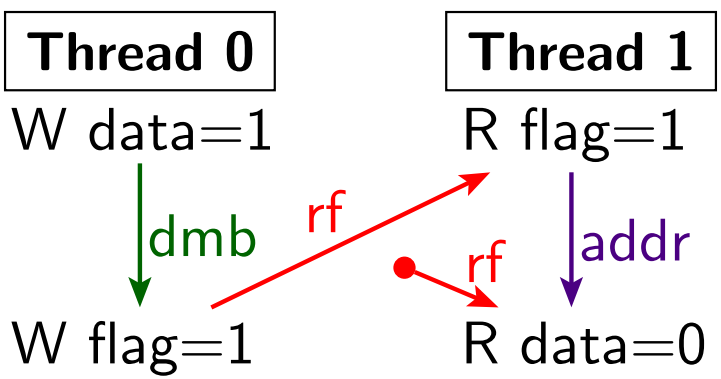
Example

Initial state: `X0=&data, X1=&flag, 0:X2=1`

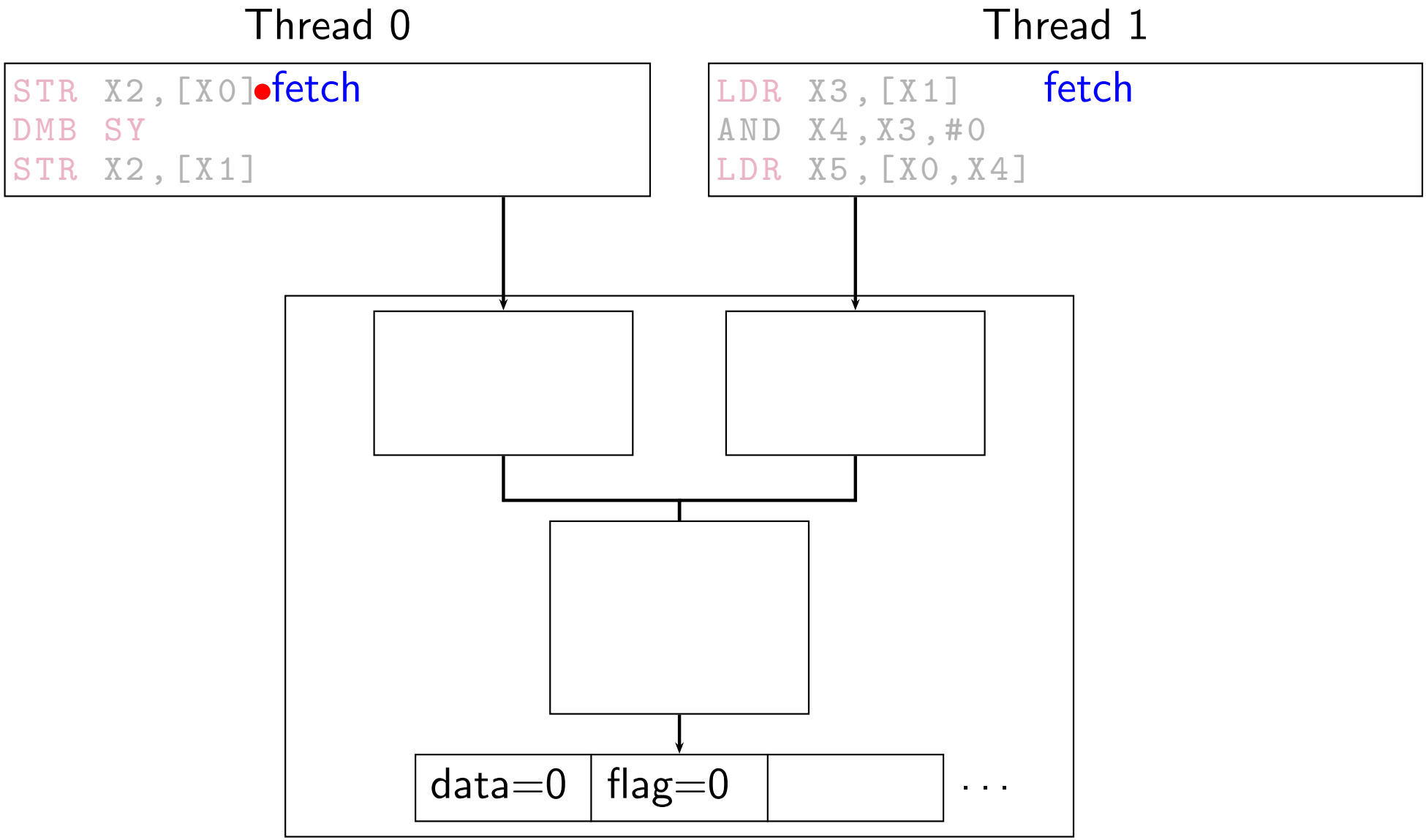
Thread 0	Thread 1

`STR X2,[X0]`
`DMB SY`
`STR X2,[X1]`
	`LDR X3,[X1]`
	`AND X4,X3,#0`
	`LDR X5,[X0,X4]`

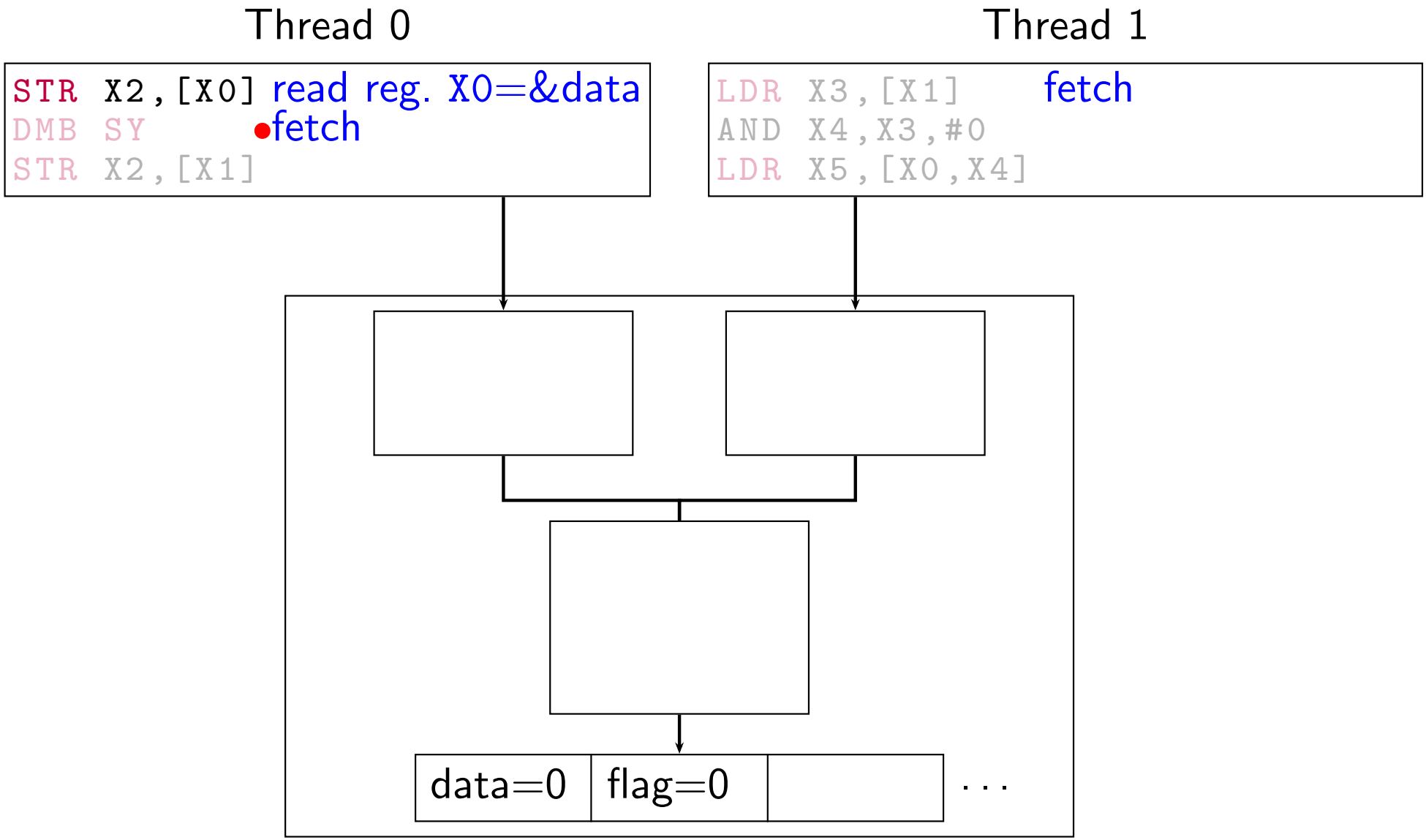
Forbidden: `1:X3=1` $\wedge$ `1:X5=0`



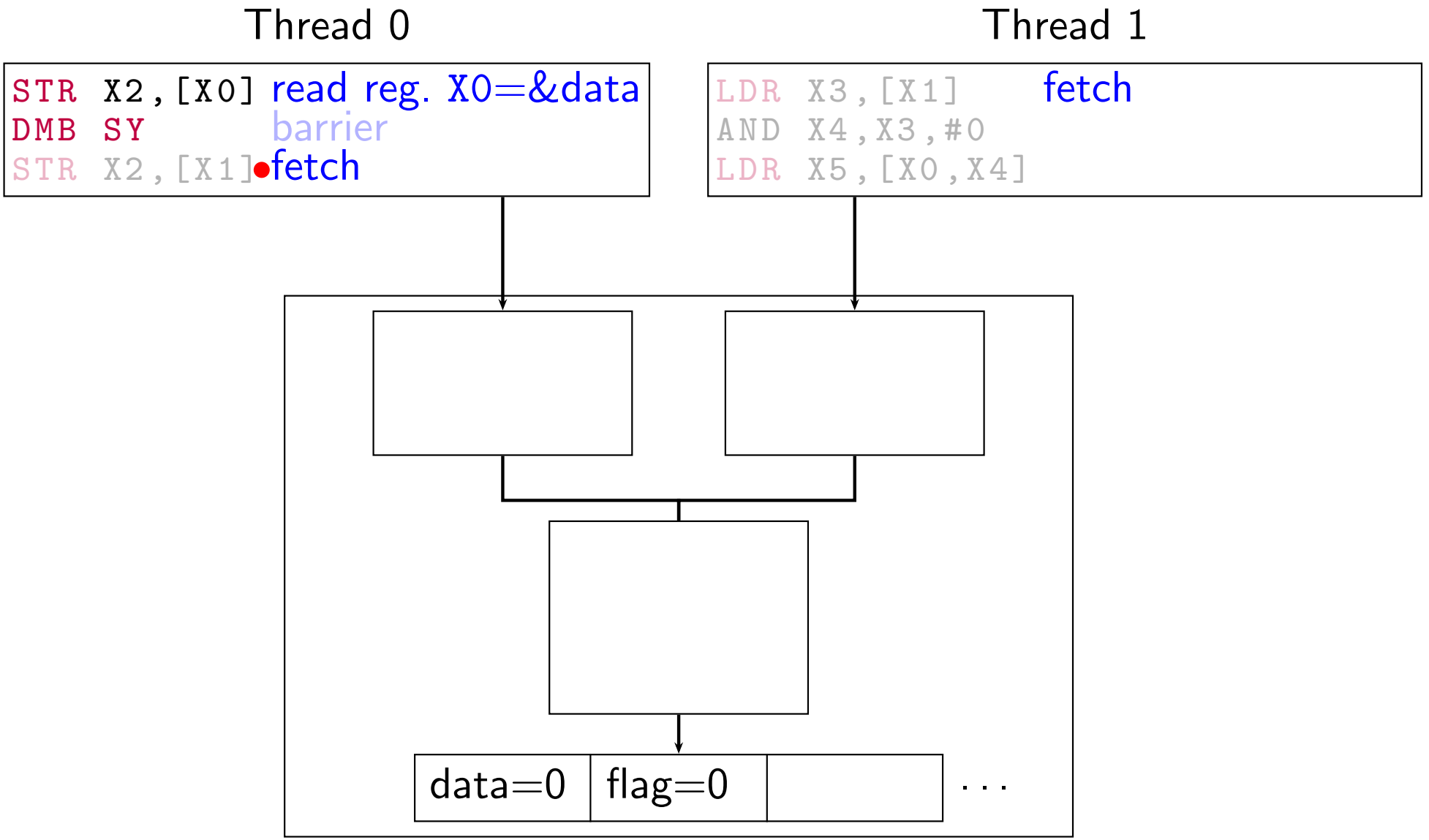
Example


Thread 0
Thread 1
STR X2,[X0]
fetch
DMB SY
STR X2,[X1]
LDR X3,[X1]
fetch
AND X4,X3,#0
LDR X5,[X0,X4]
data=0
flag=0
· · ·

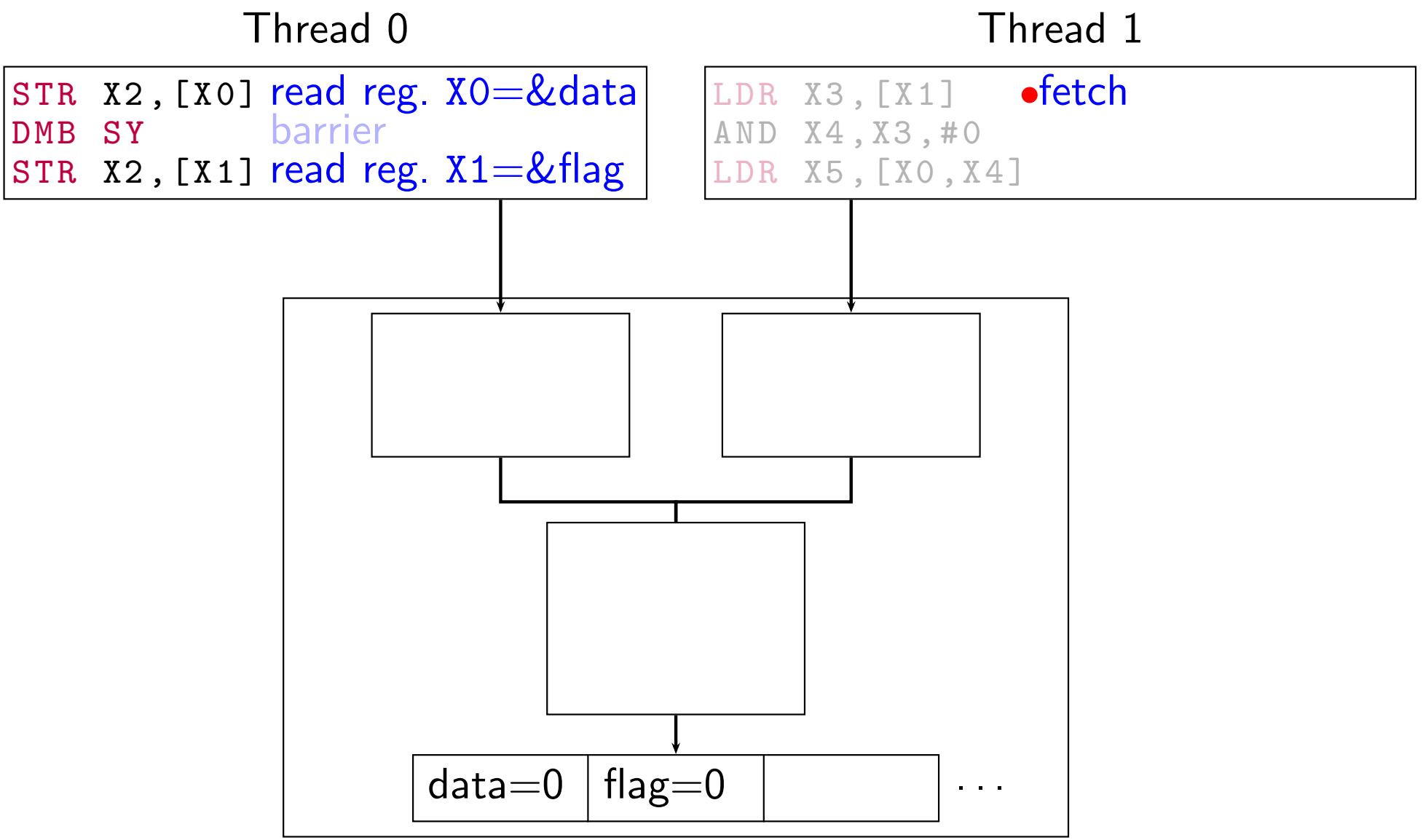
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY
fetch
STR X2,[X1]
LDR X3,[X1]
fetch
AND X4,X3,#0
LDR X5,[X0,X4]
data=0
flag=0
· · ·

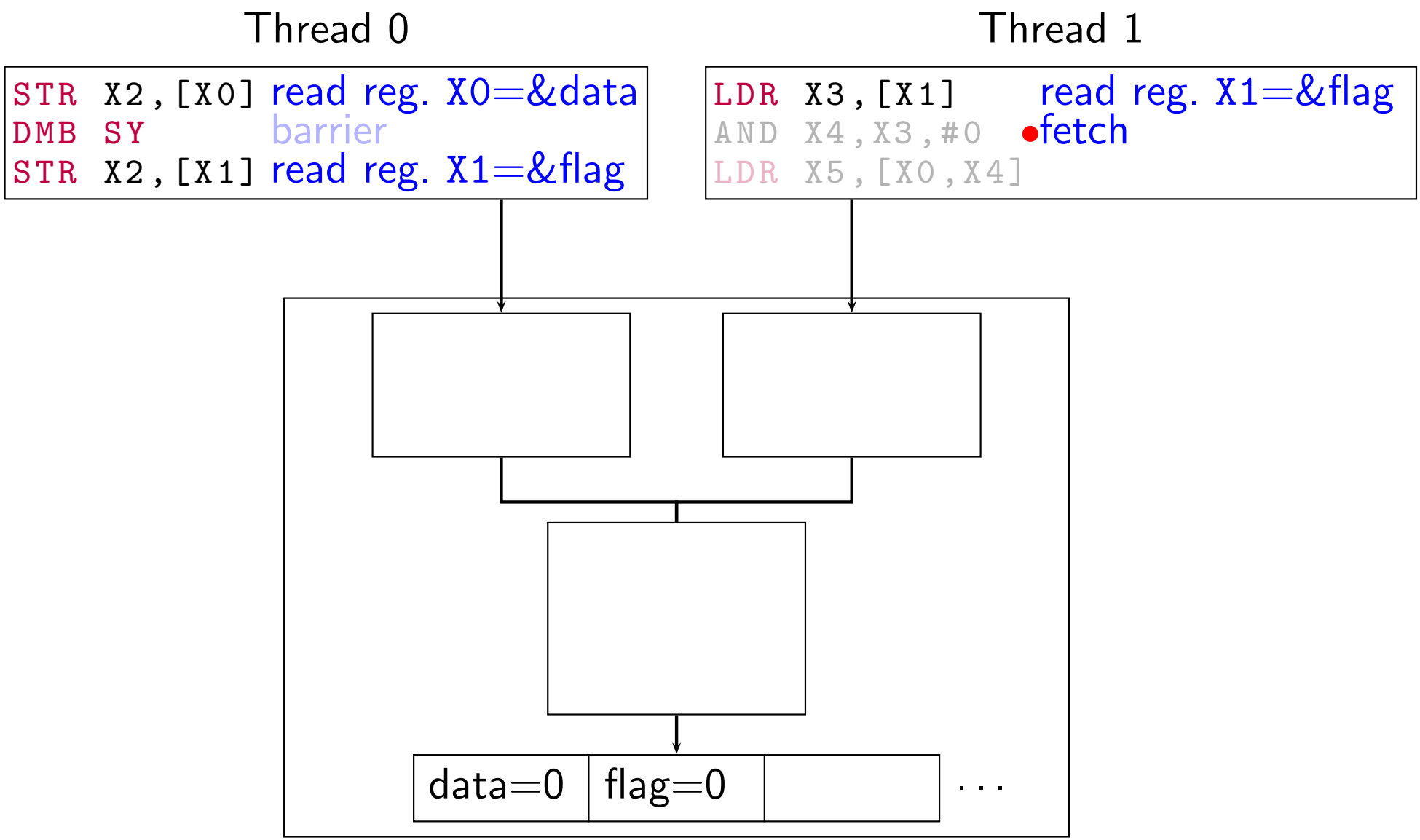
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] fetch
LDR X3,[X1] fetch
AND X4,X3,#0
LDR X5,[X0,X4]
data=0
flag=0
· · ·

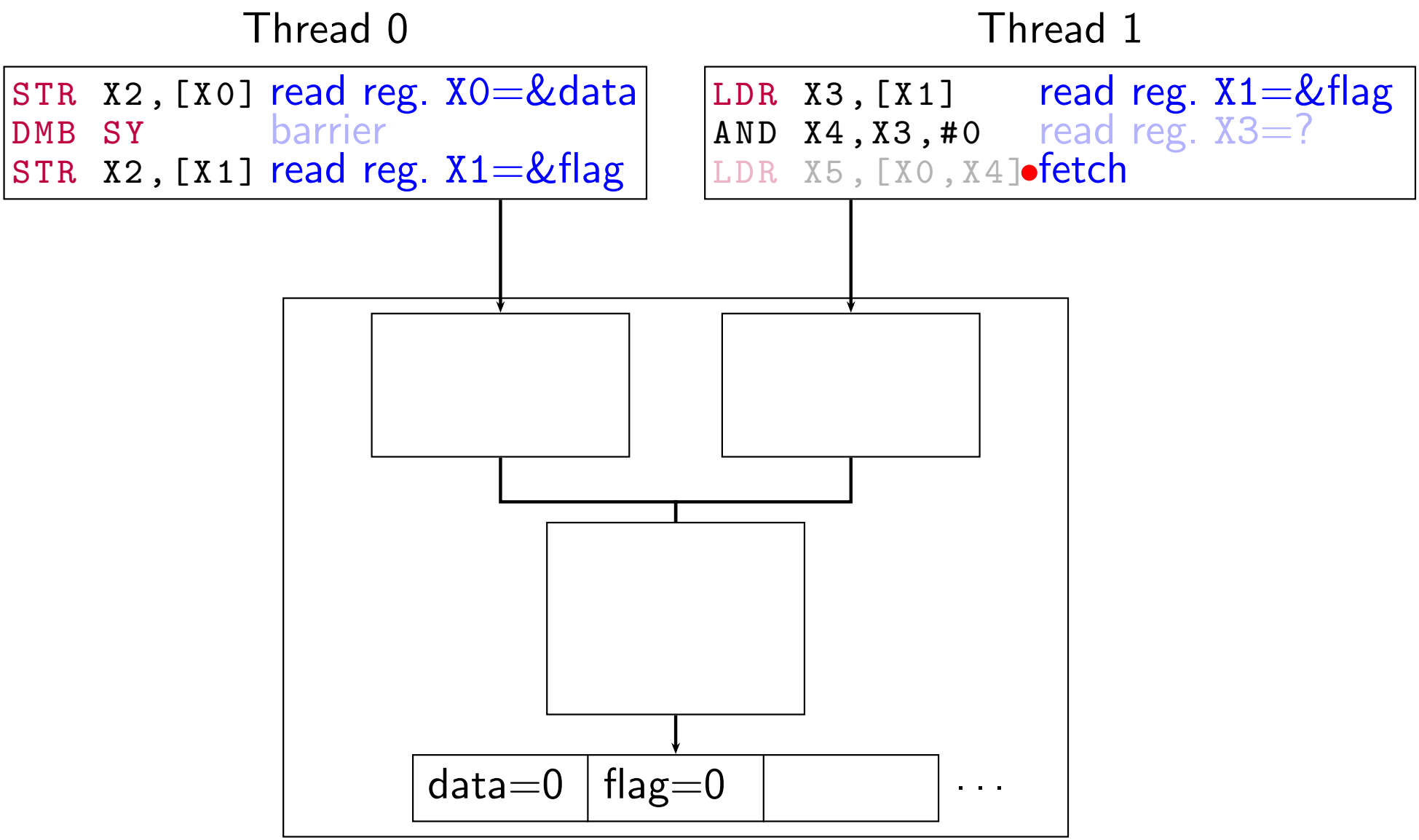
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] •fetch
AND X4,X3,#0
LDR X5,[X0,X4]
data=0
flag=0
· · ·

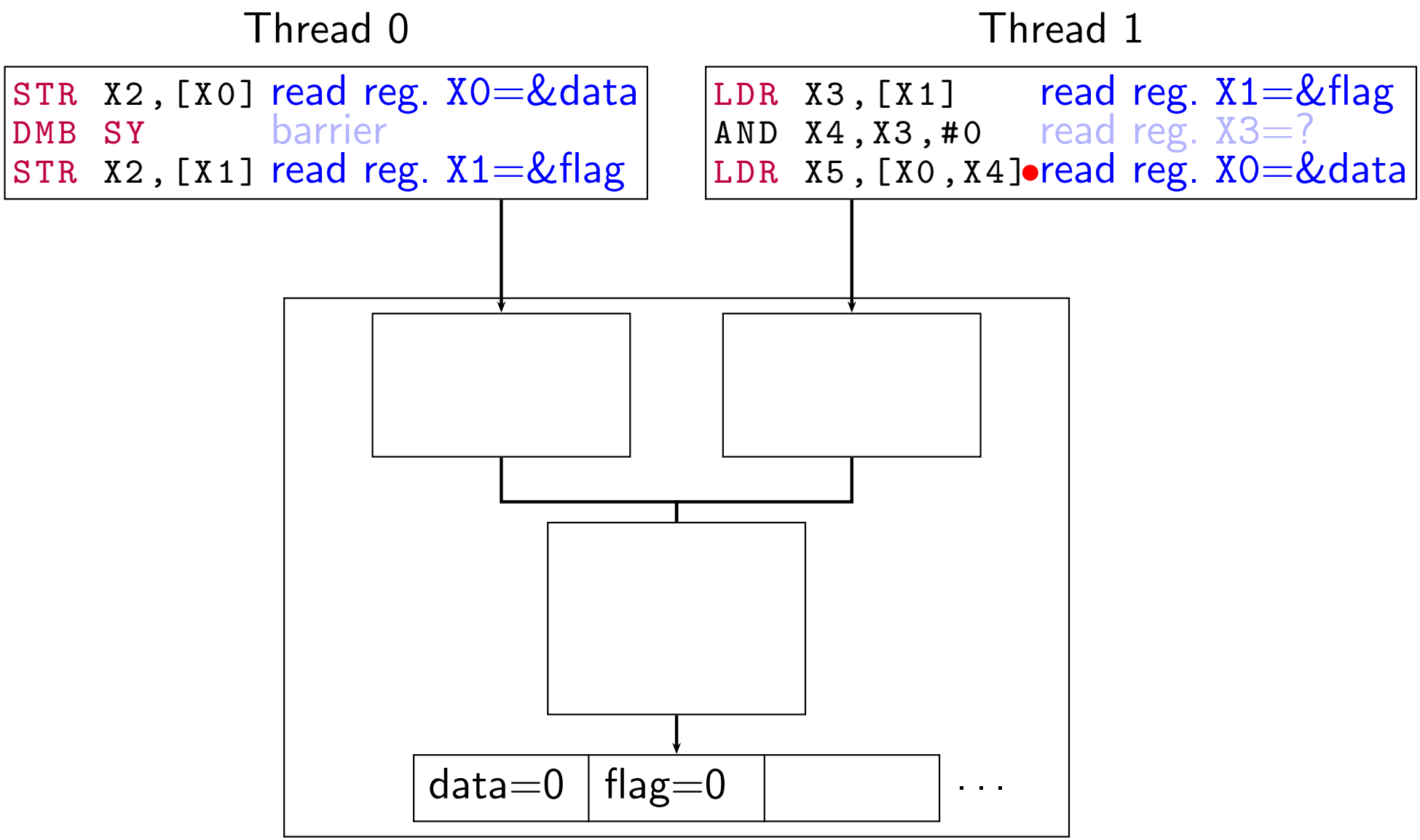
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] read reg. X1=&flag
AND X4,X3,#0 fetch
LDR X5,[X0,X4]
data=0
flag=0
· · ·

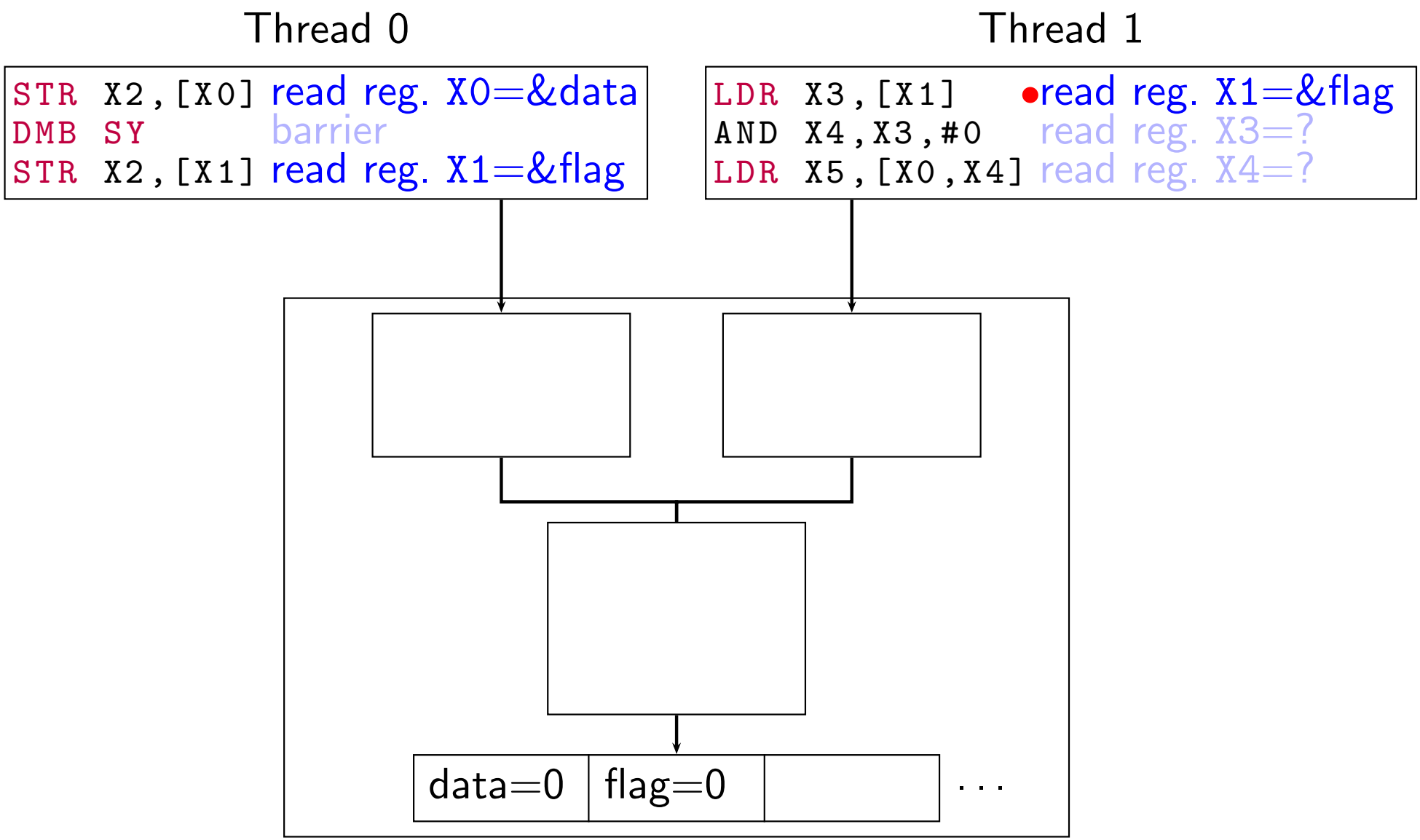
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] read reg. X1=&flag
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] fetch
data=0
flag=0
· · ·

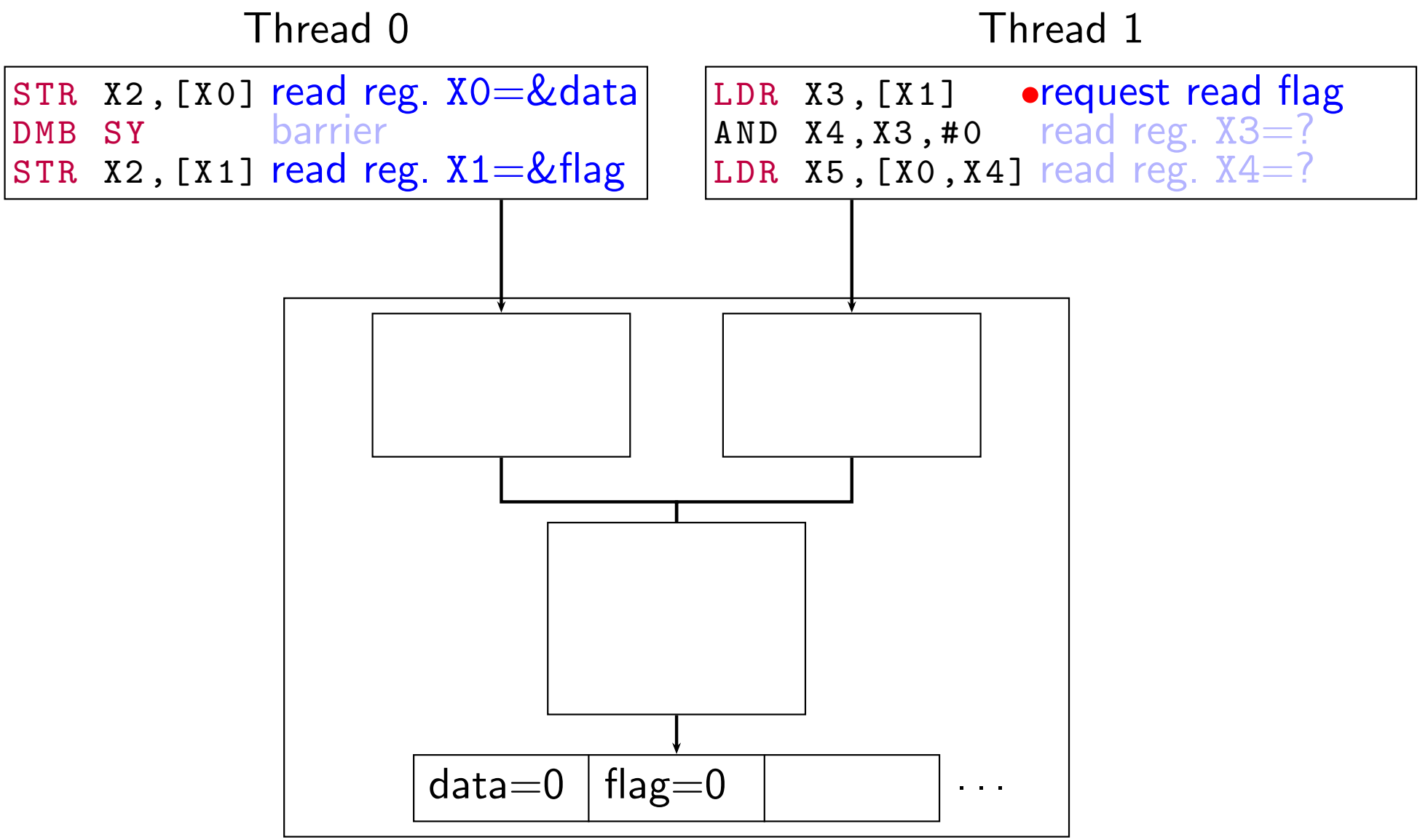
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] read reg. X1=&flag
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X0=&data
data=0
flag=0
· · ·

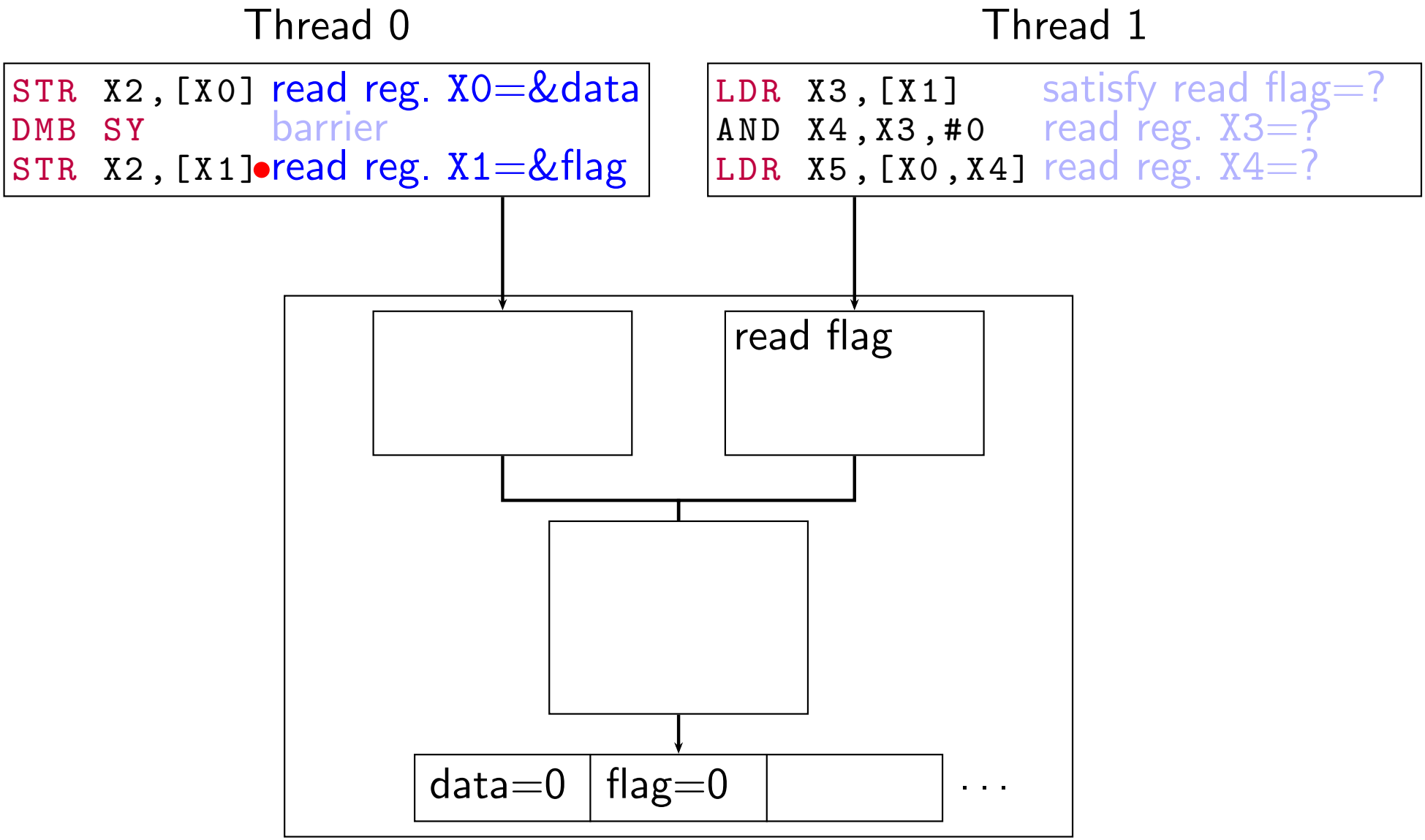
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] read reg. X1=&flag
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
data=0
flag=0
· · ·

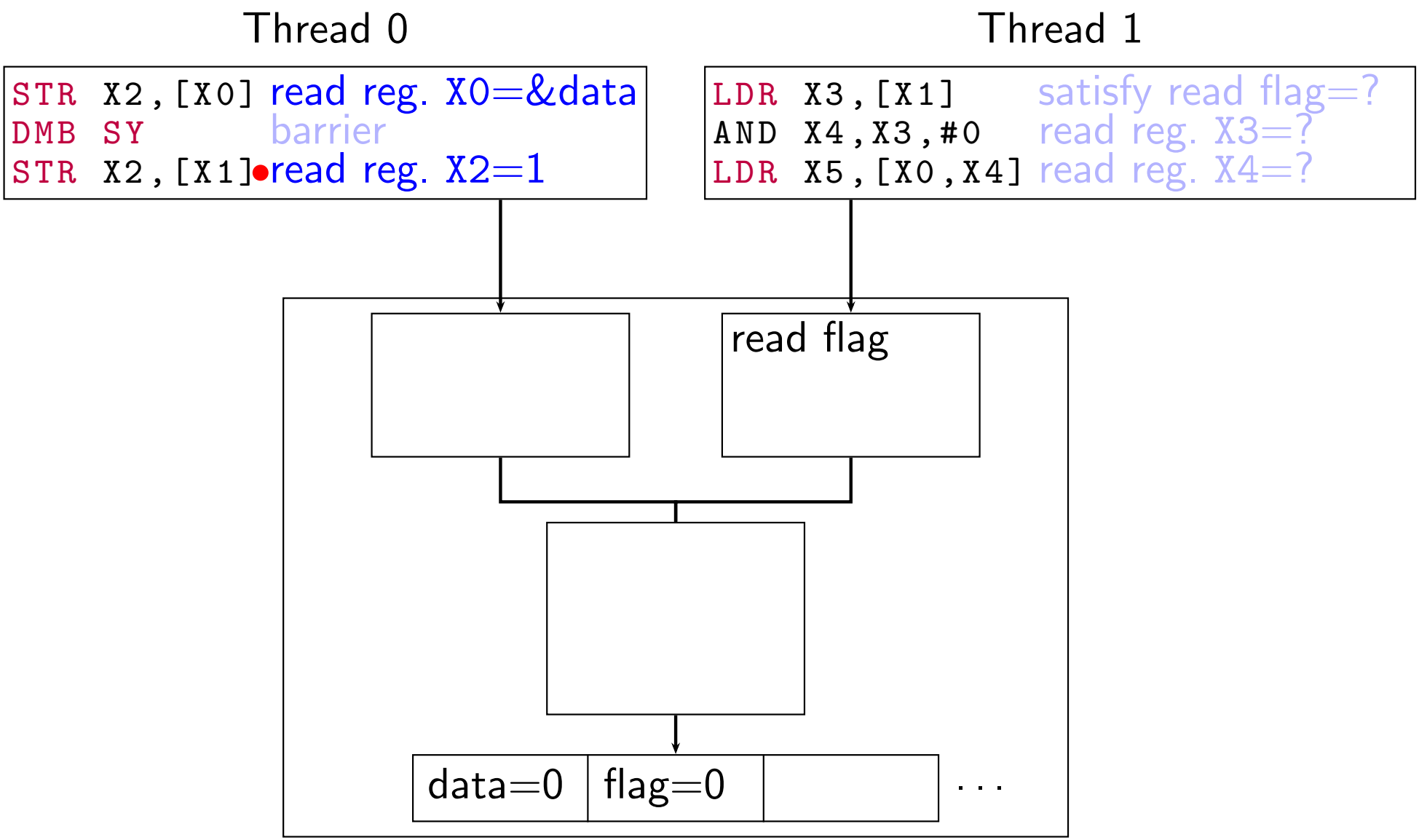
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] •request read flag
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
data=0
flag=0
· · ·

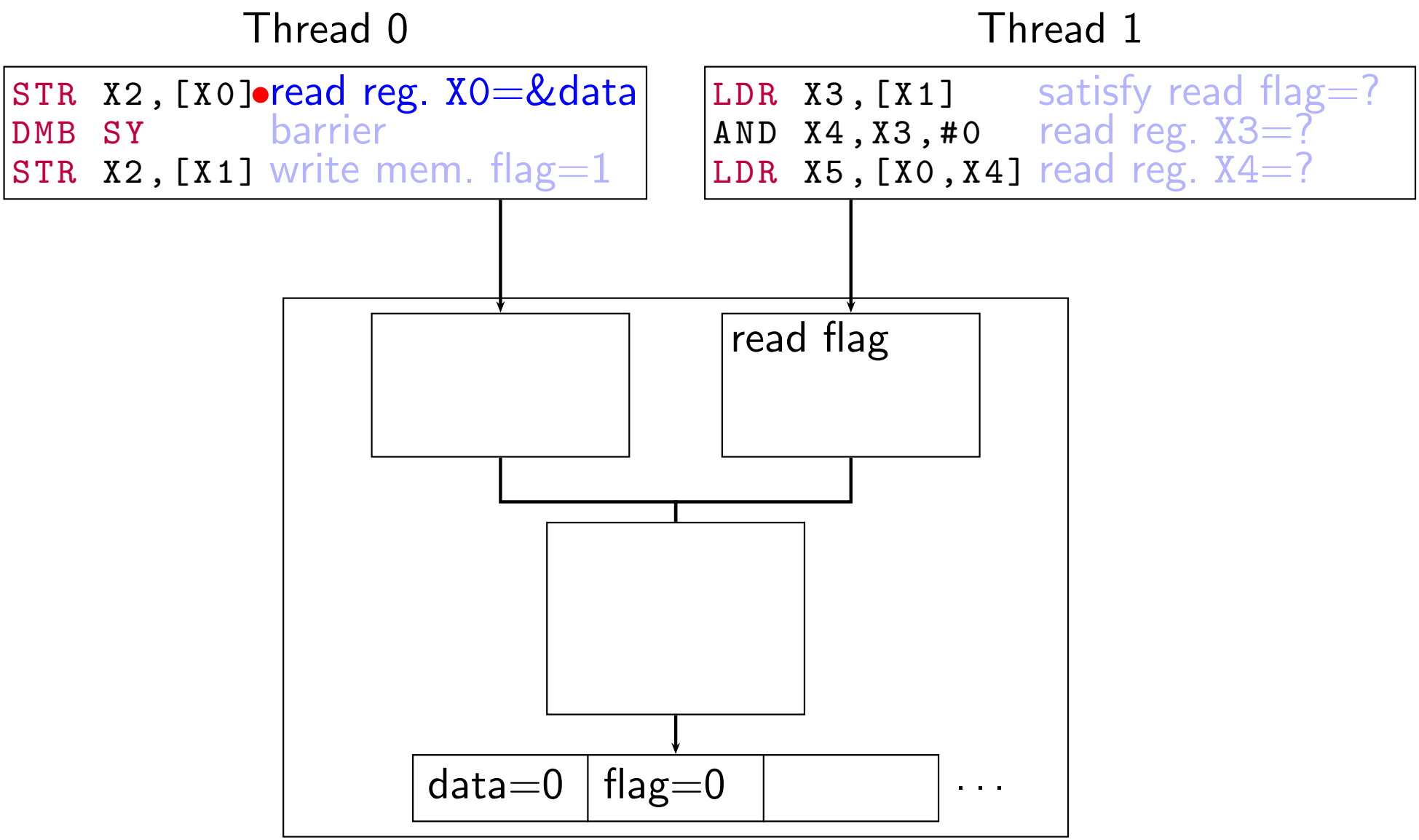
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X1=&flag
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
read flag
data=0
flag=0
· · ·

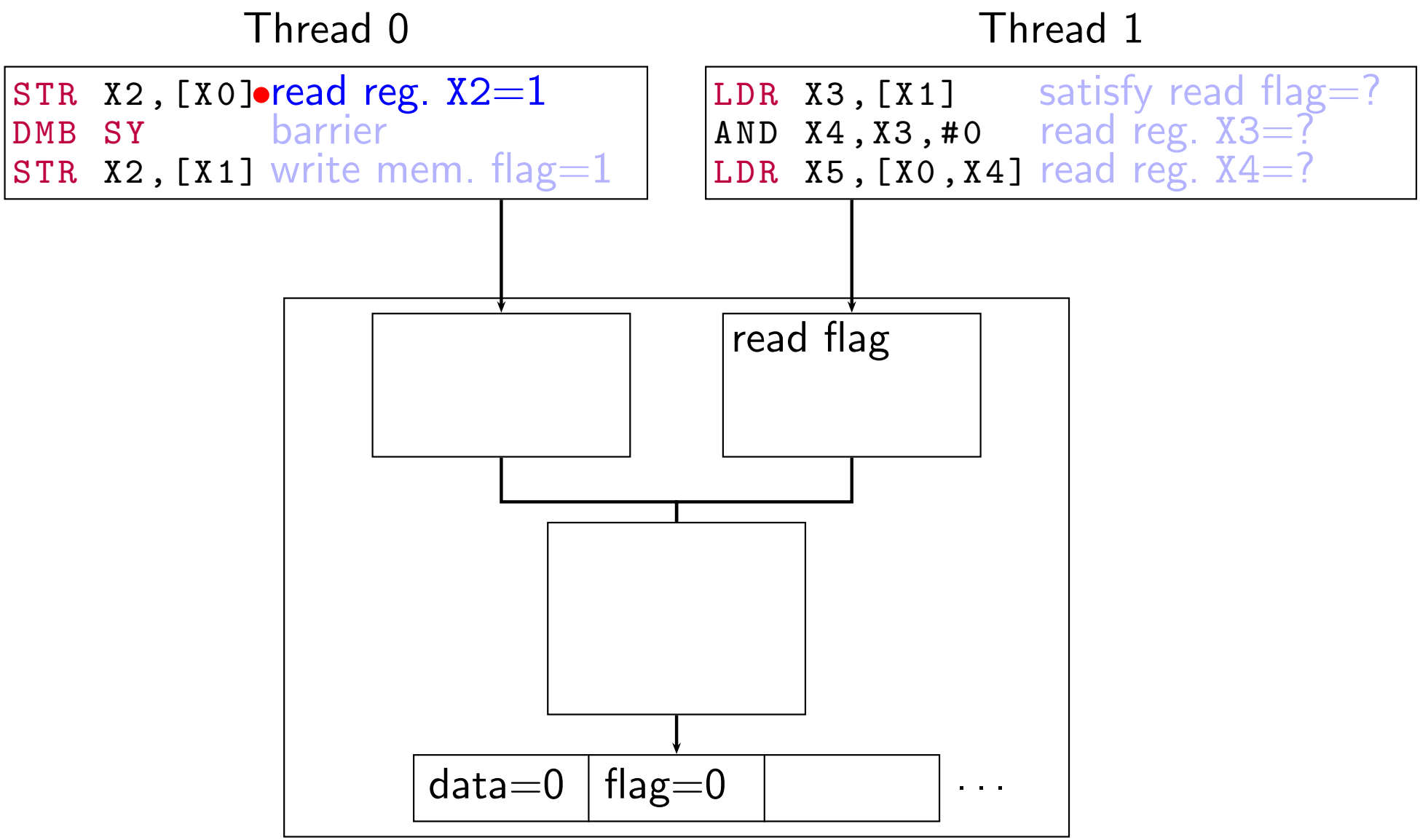
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] read reg. X2=1
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
read flag
data=0
flag=0
· · ·

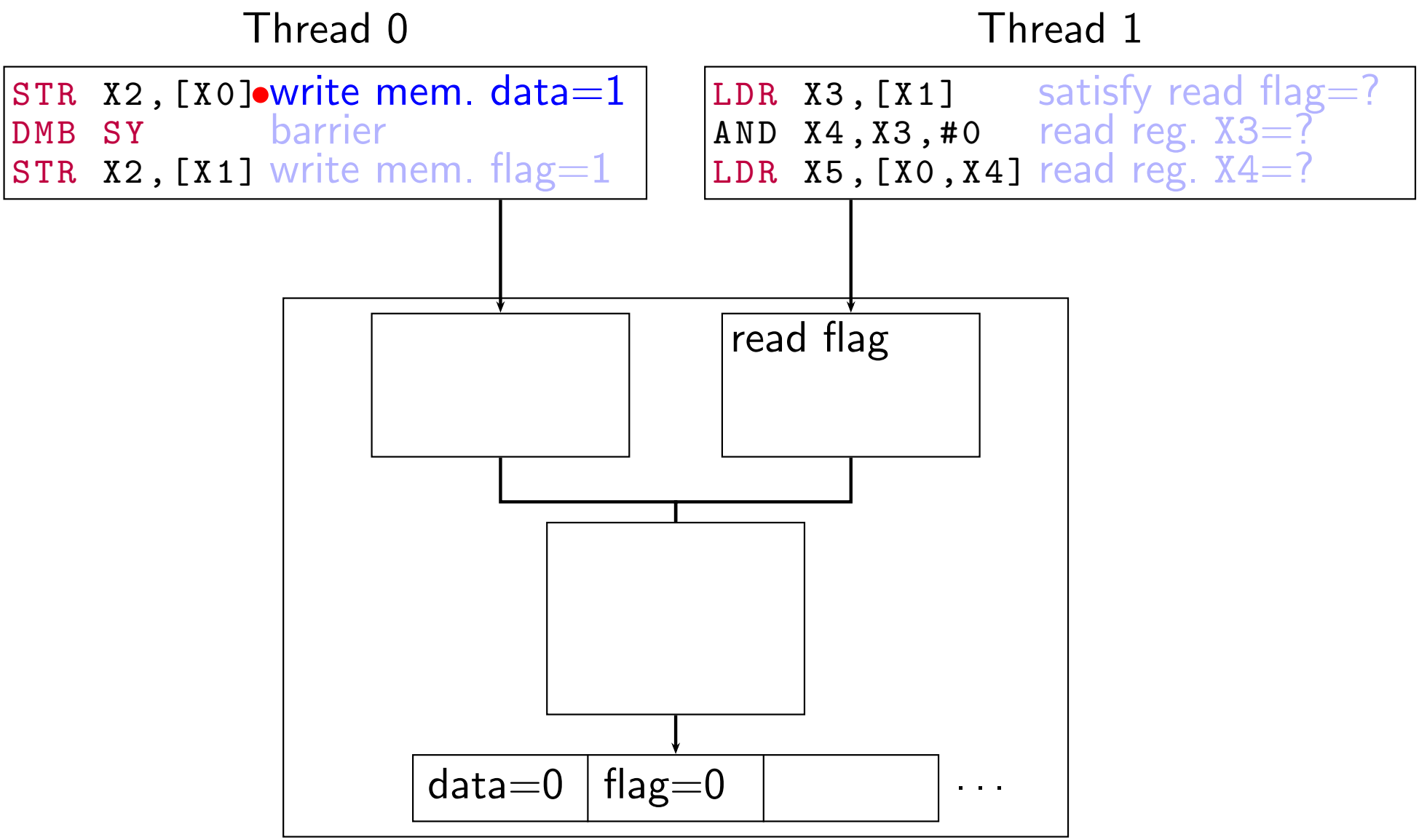
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X0=&data
DMB SY barrier
STR X2,[X1] write mem. flag=1
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
read flag
data=0
flag=0
· · ·

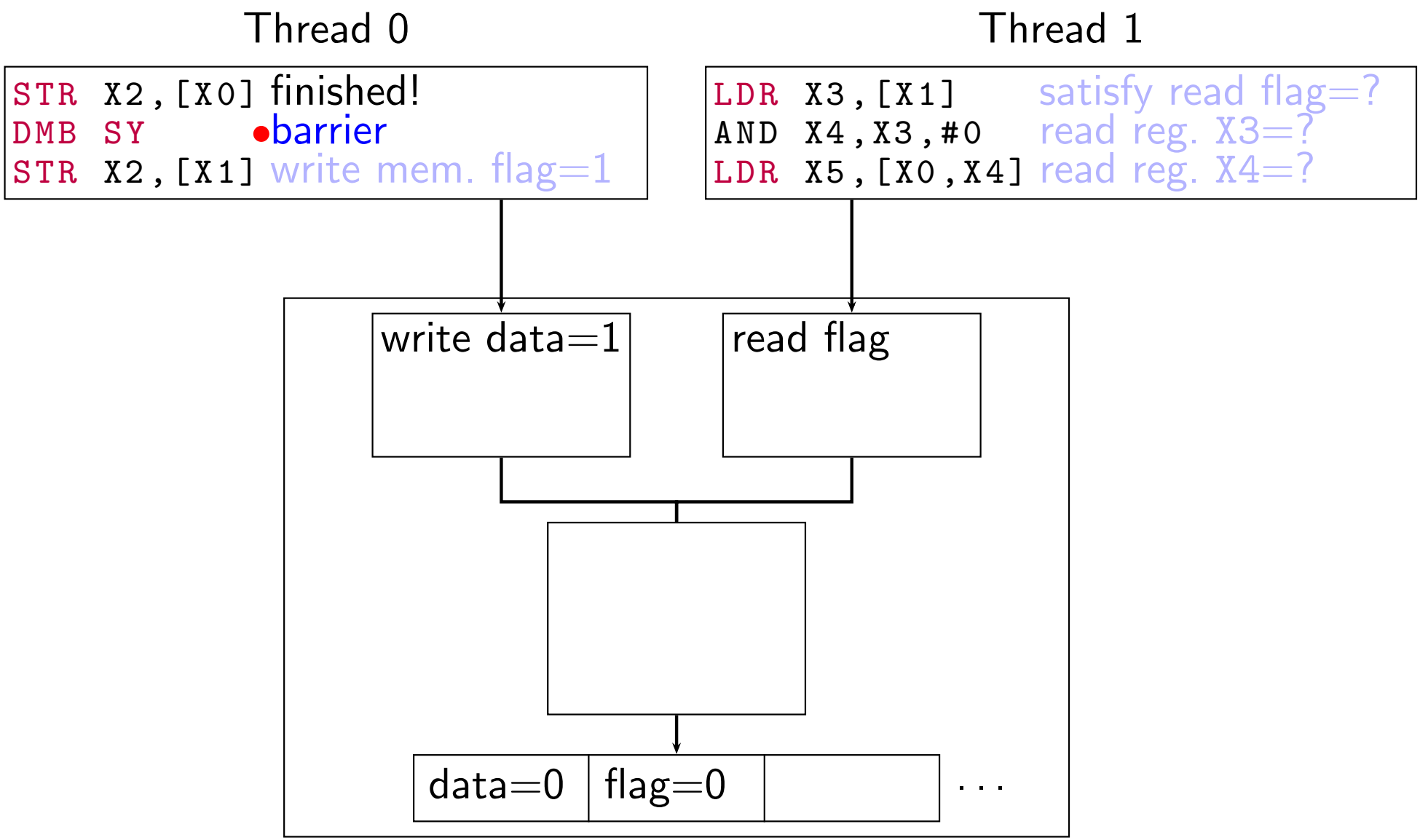
Example


Thread 0
Thread 1
STR X2,[X0] read reg. X2=1
DMB SY barrier
STR X2,[X1] write mem. flag=1
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
read flag
data=0
flag=0
· · ·

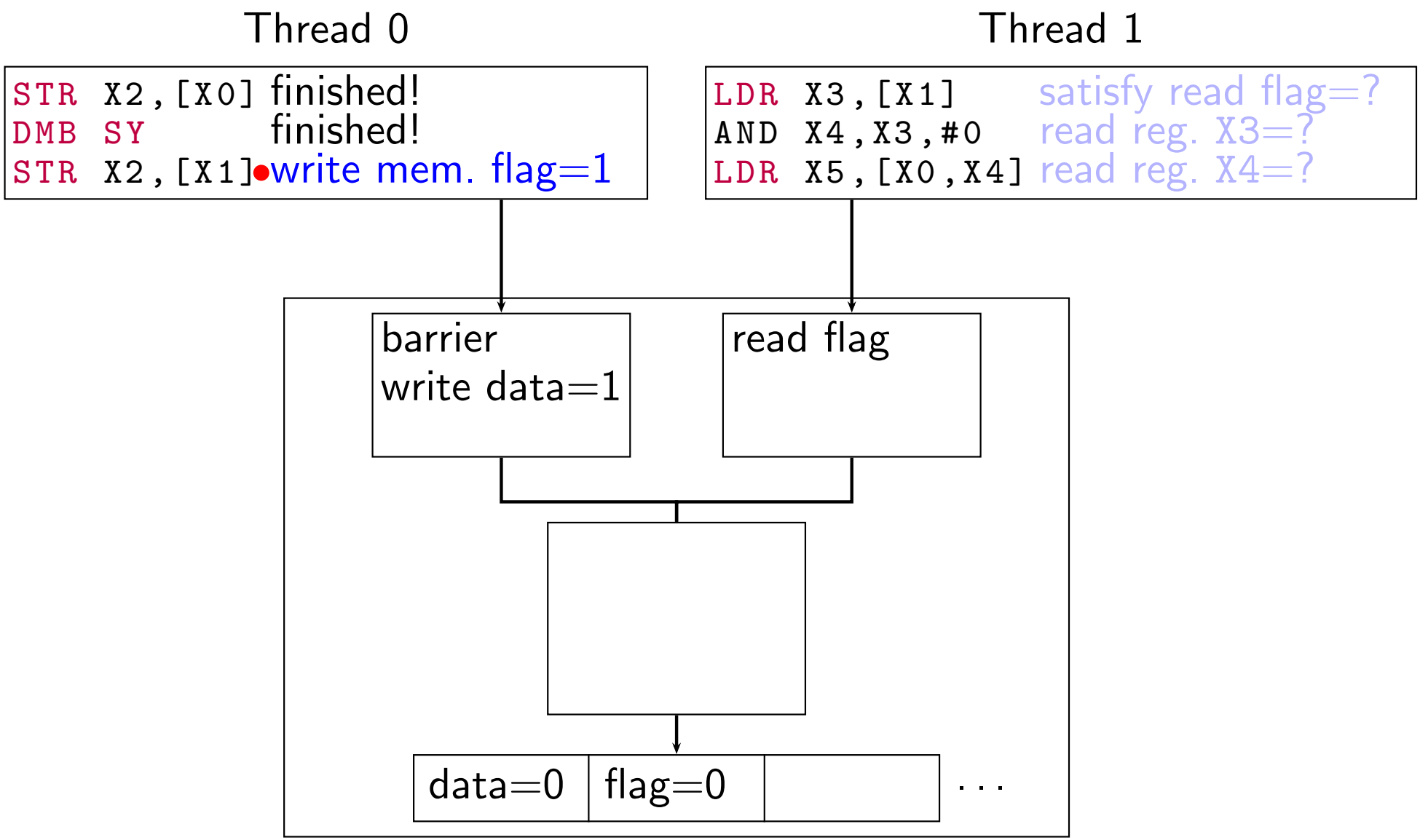
Example


Thread 0
Thread 1
STR X2,[X0] write mem. data=1
DMB SY barrier
STR X2,[X1] write mem. flag=1
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
read flag
data=0
flag=0
· · ·

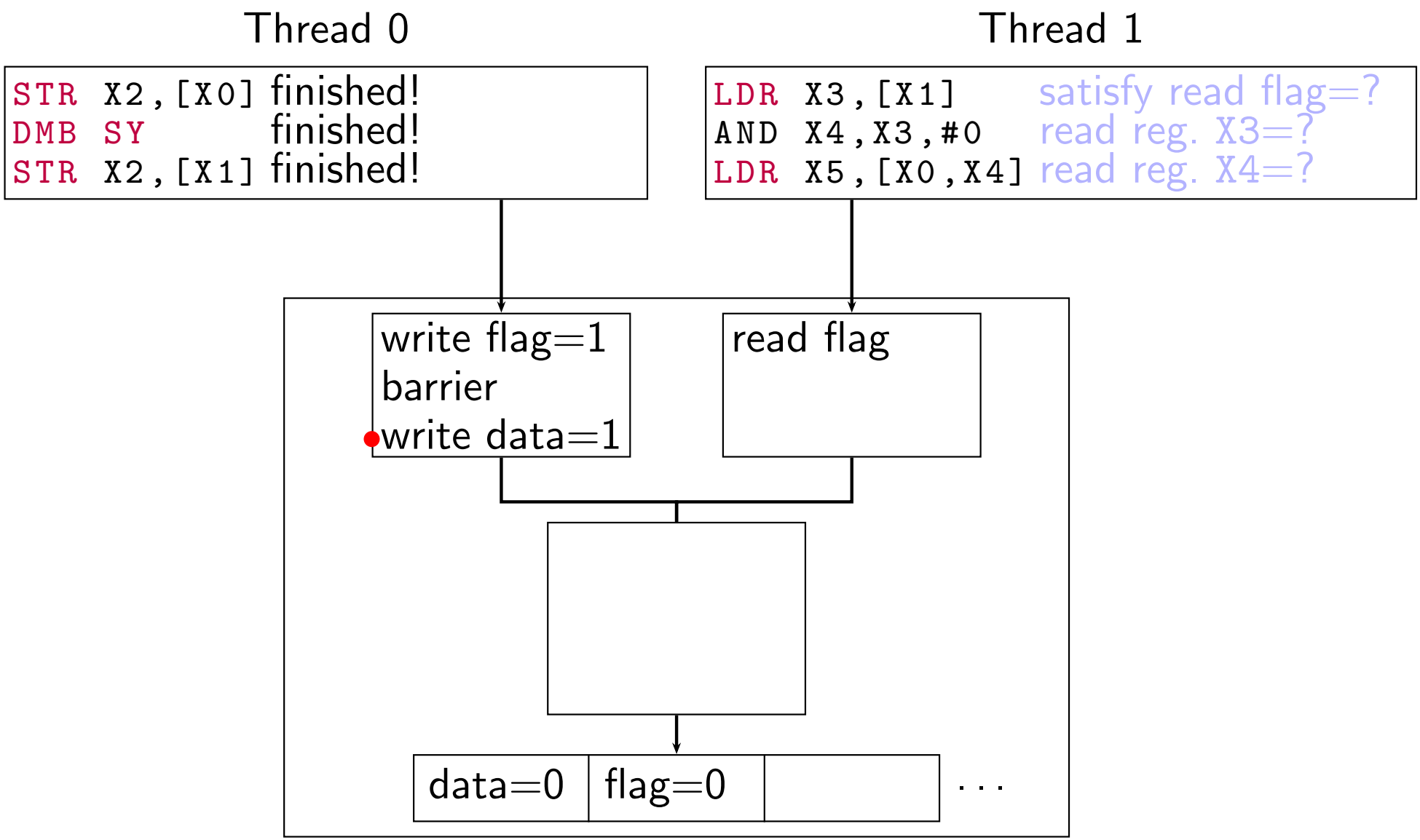
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY barrier
STR X2,[X1] write mem. flag=1
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
write data=1
read flag
data=0
flag=0
· · ·

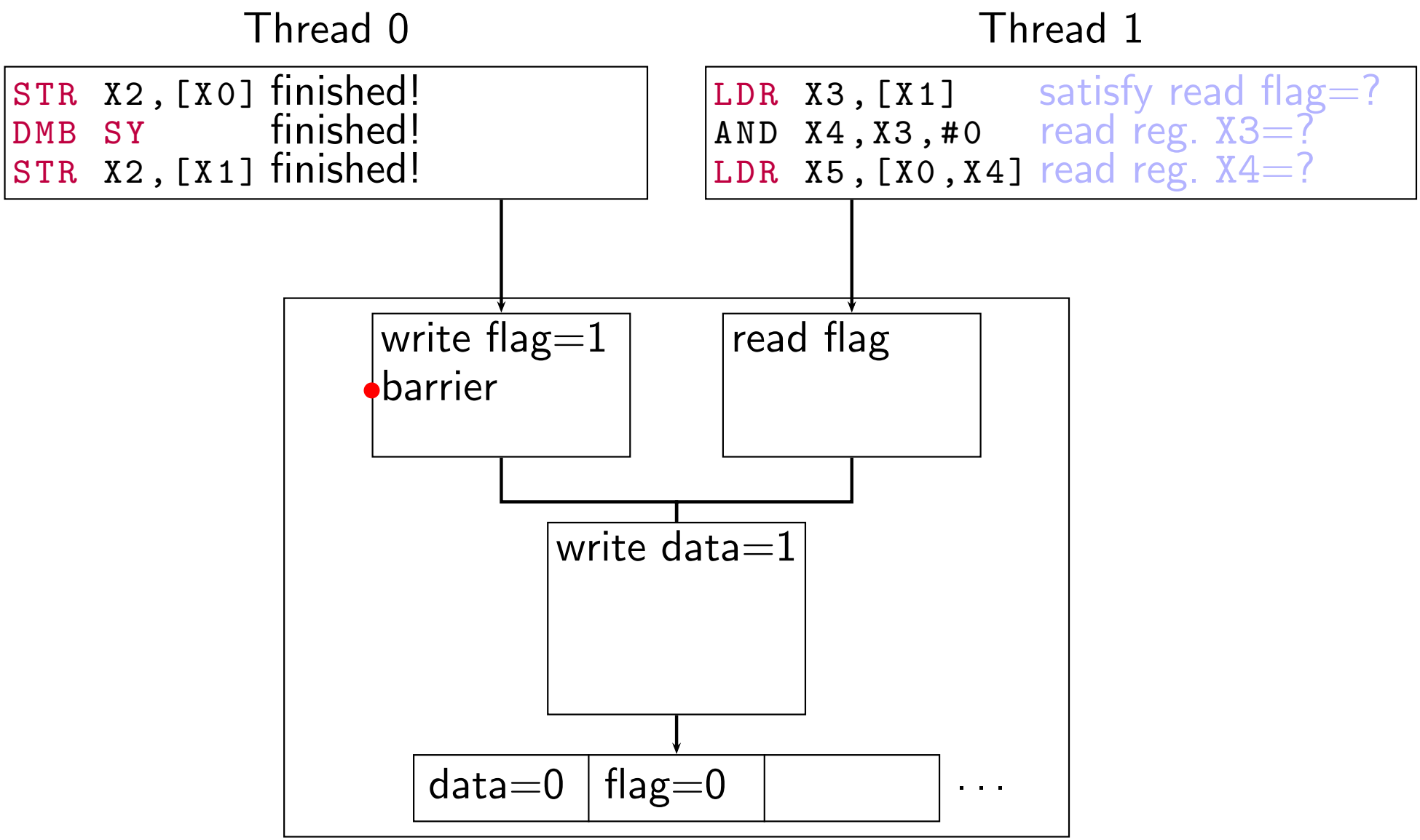
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] write mem. flag=1
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
barrier
write data=1
read flag
data=0
flag=0
· · ·

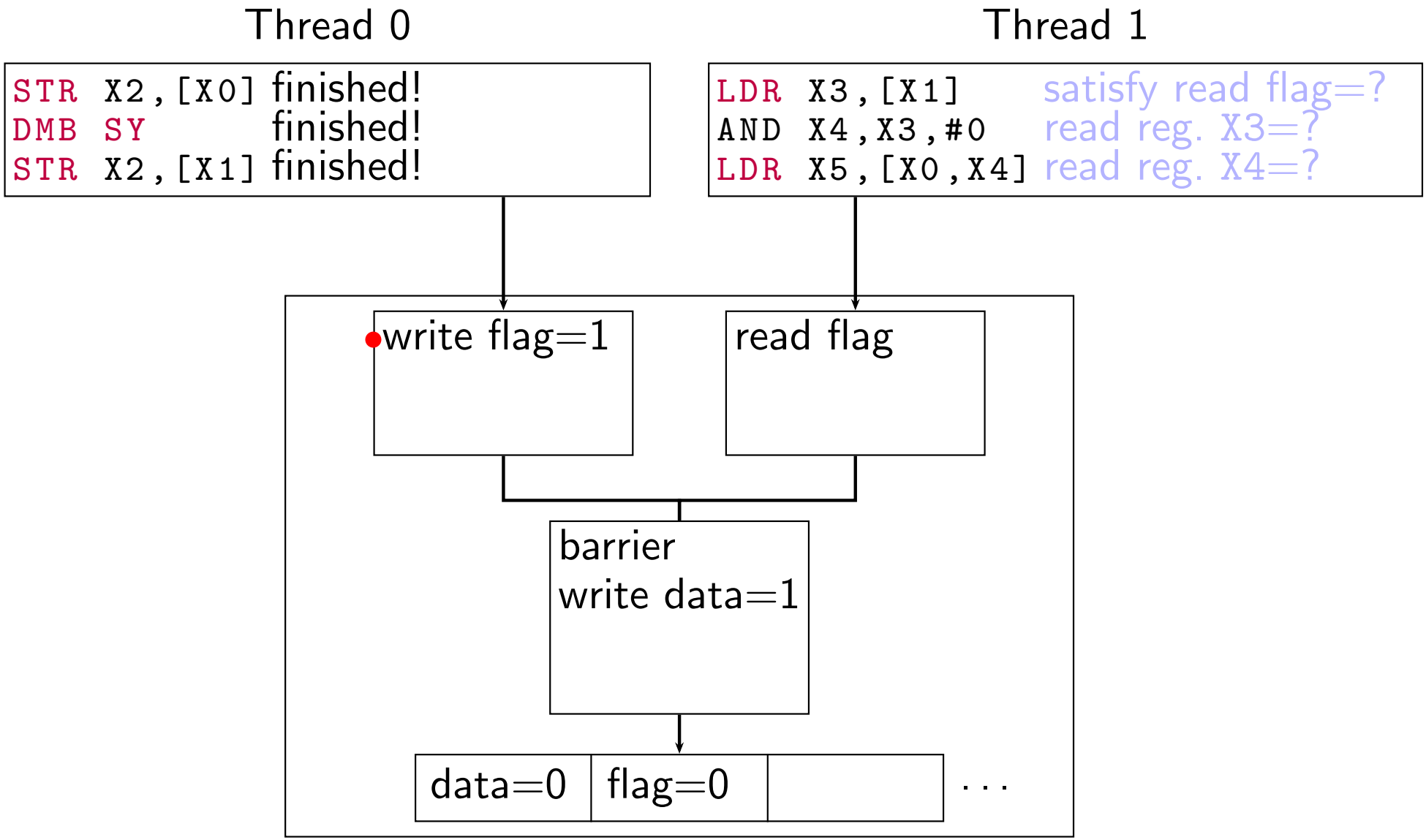
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
write flag=1
barrier
write data=1
read flag
data=0
flag=0
· · ·

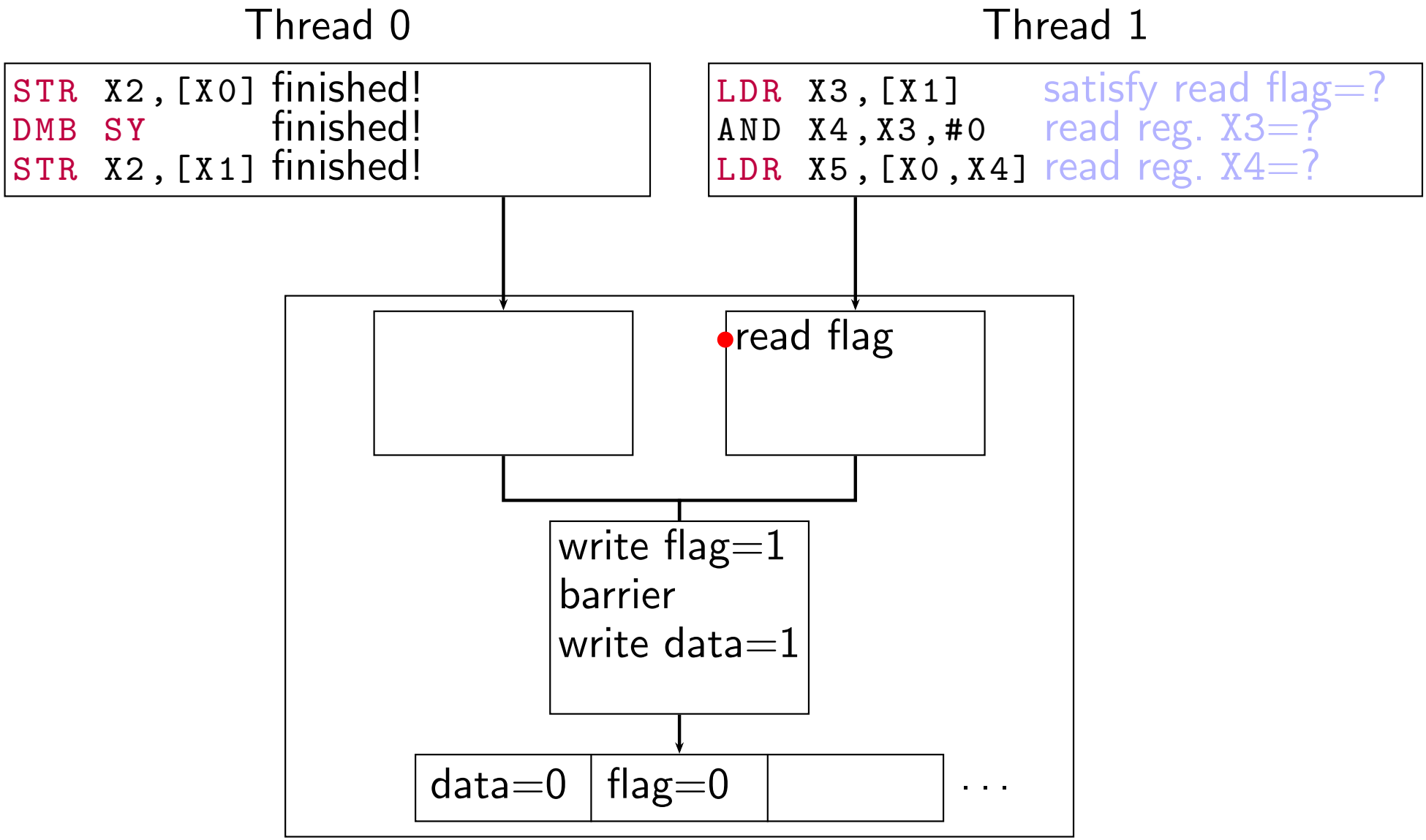
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
write flag=1
barrier
read flag
write data=1
data=0
flag=0
· · ·

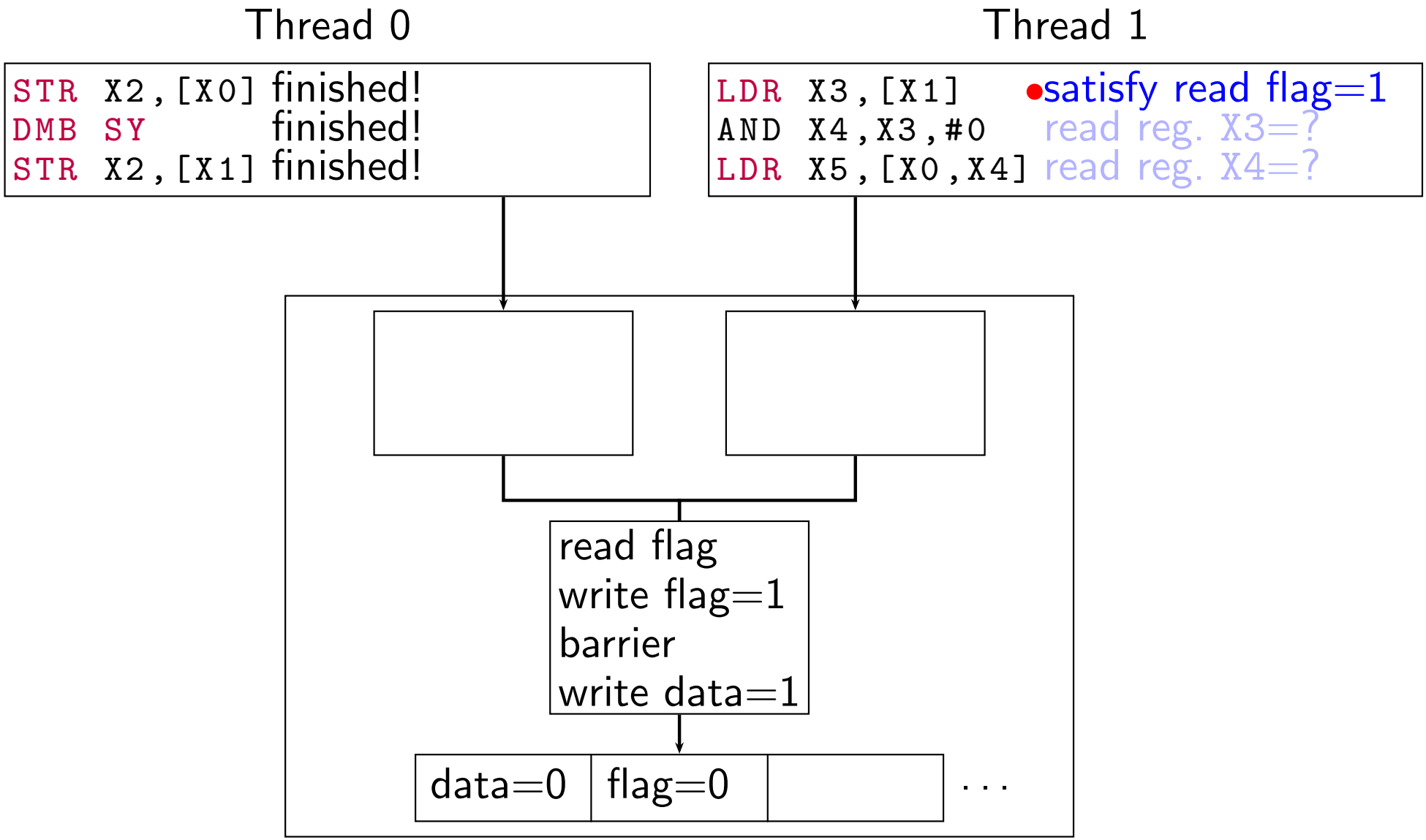
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
write flag=1
read flag
barrier
write data=1
data=0
flag=0
· · ·

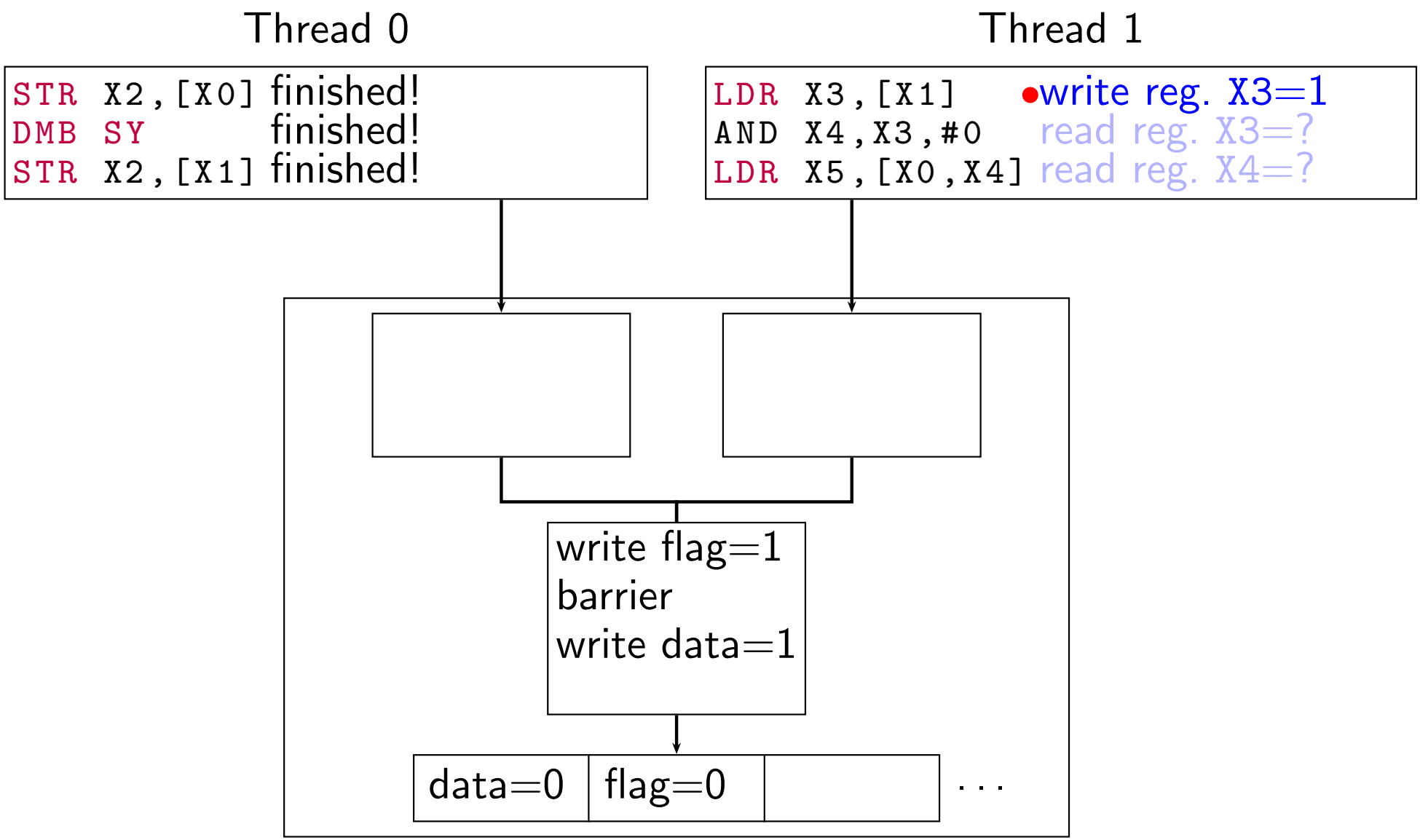
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] satisfy read flag=?
AND X4,X3,#0 read reg. X3=?
LDR X5,[X0,X4] read reg. X4=?
read flag
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

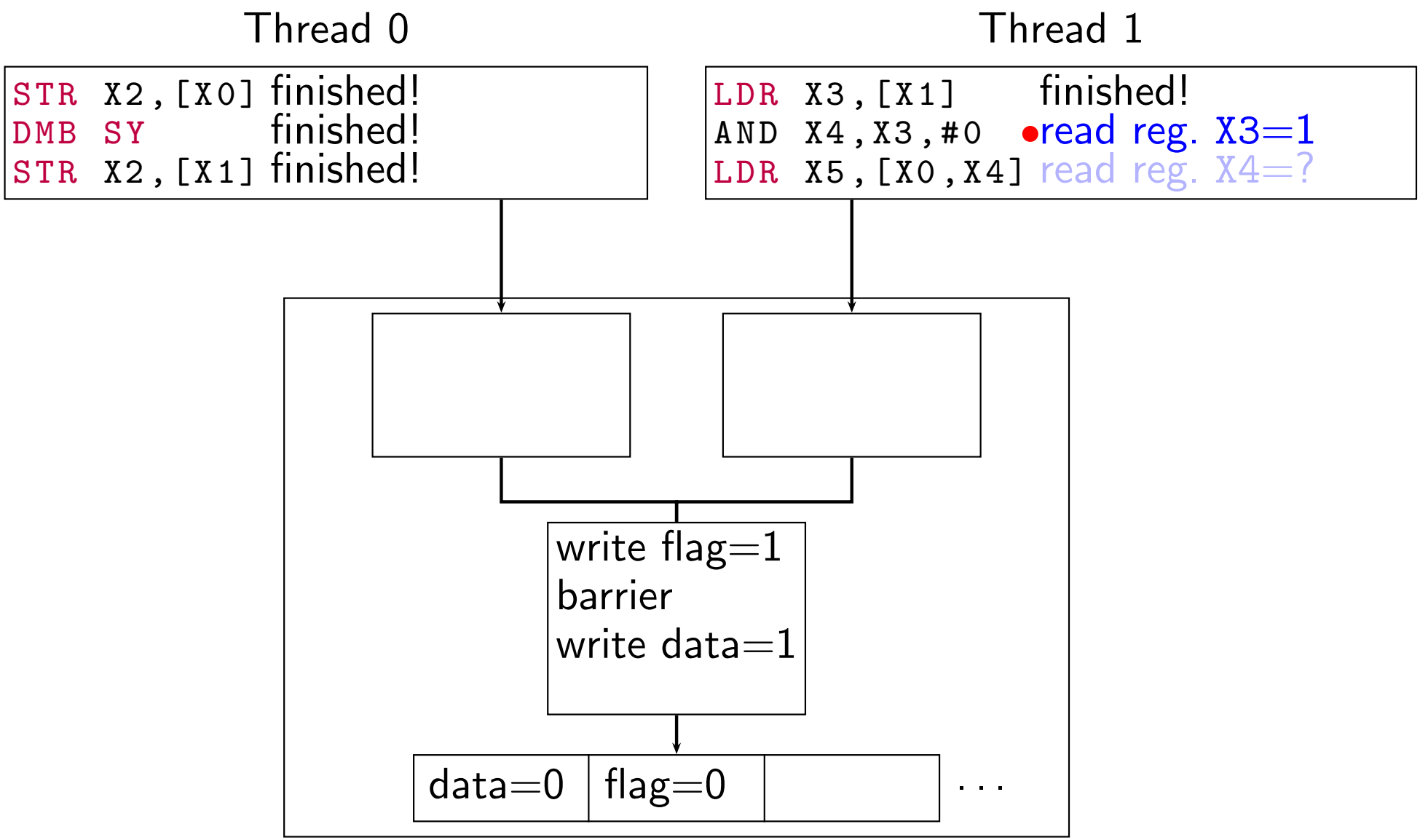
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1]
satisfy read flag=1
AND X4,X3,#0
read reg. X3=?
LDR X5,[X0,X4]
read reg. X4=?
read flag
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

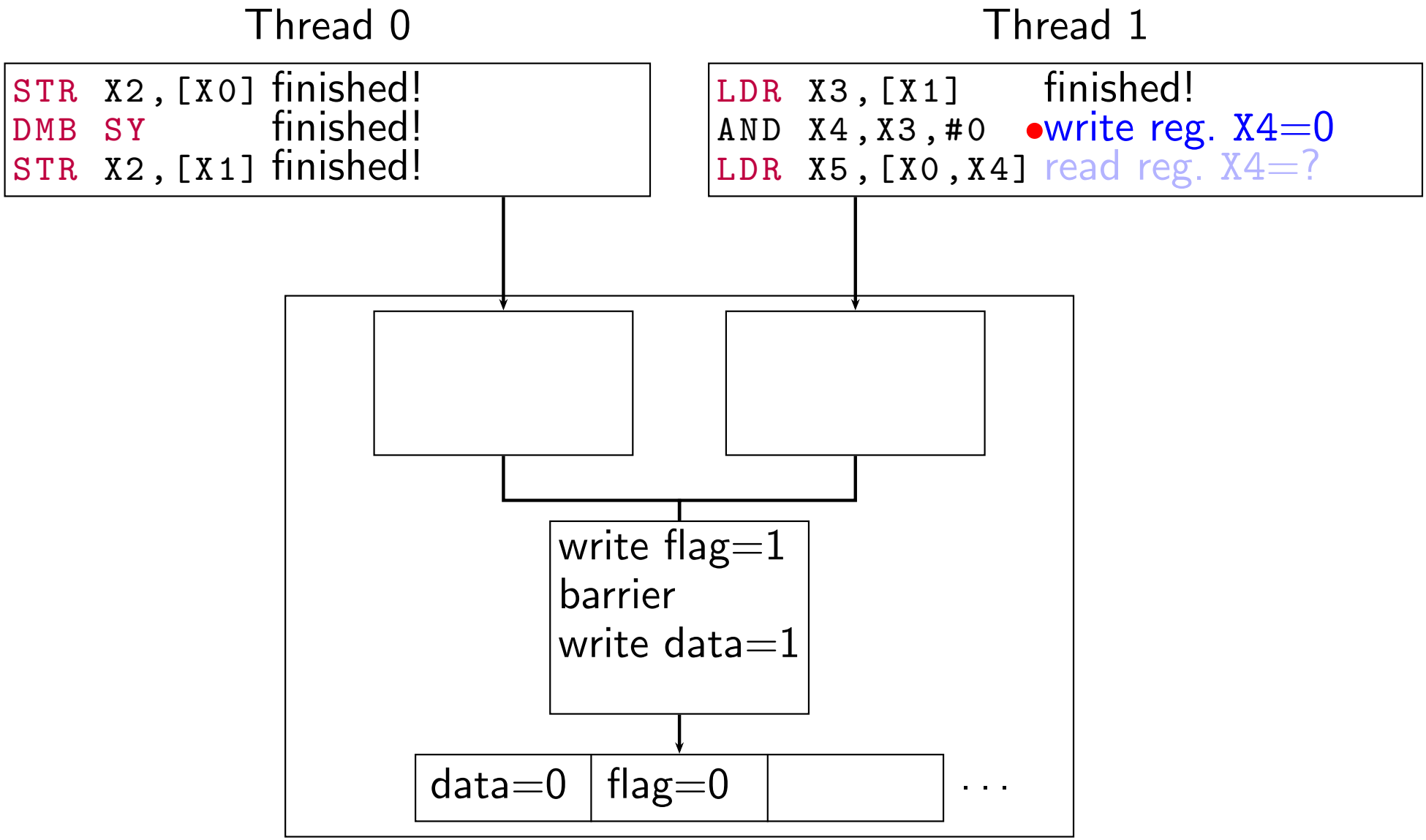
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1]
write reg. X3=1
AND X4,X3,#0
read reg. X3=?
LDR X5,[X0,X4]
read reg. X4=?
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

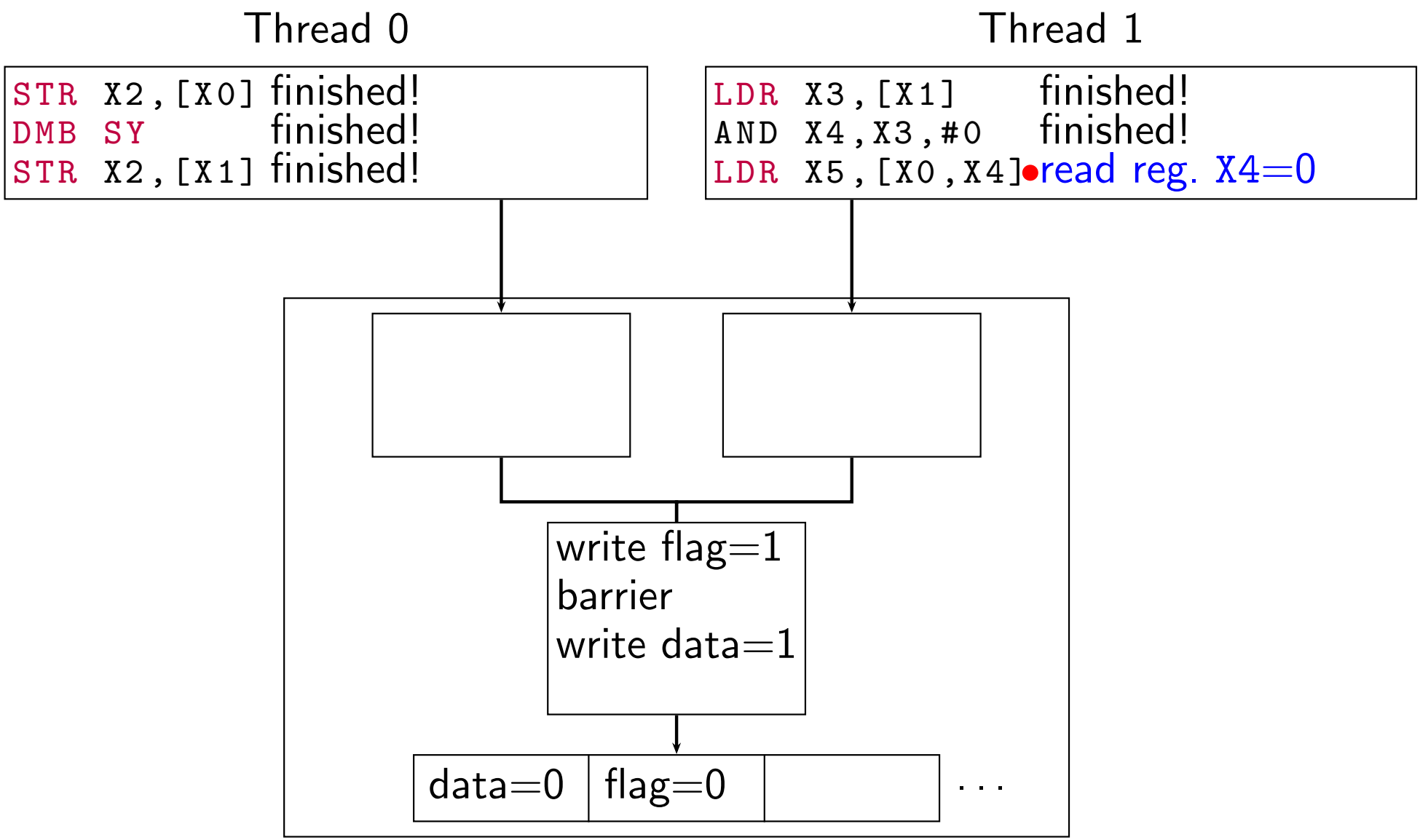
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 read reg. X3=1
LDR X5,[X0,X4] read reg. X4=?
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

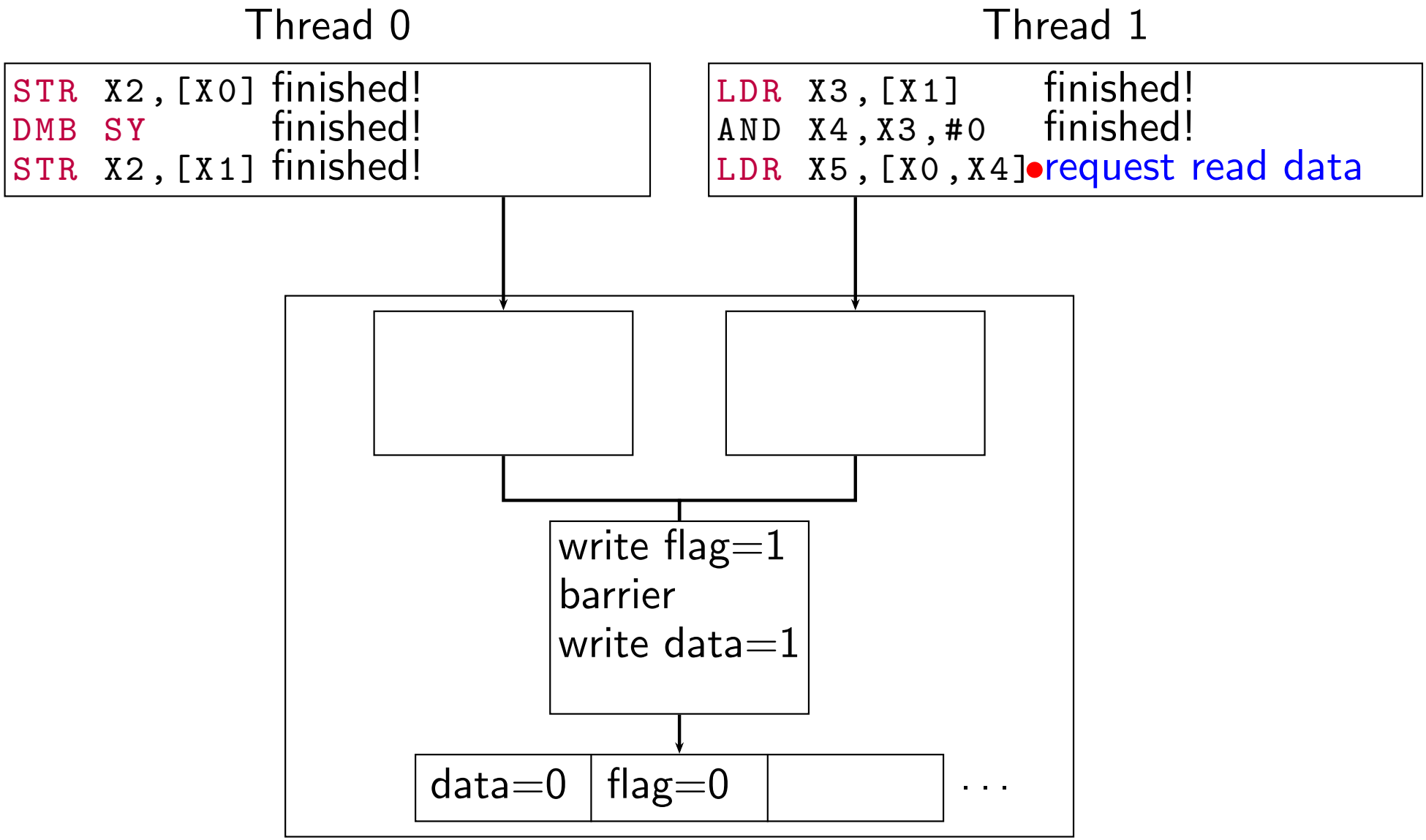
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 write reg. X4=0
LDR X5,[X0,X4] read reg. X4=?
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

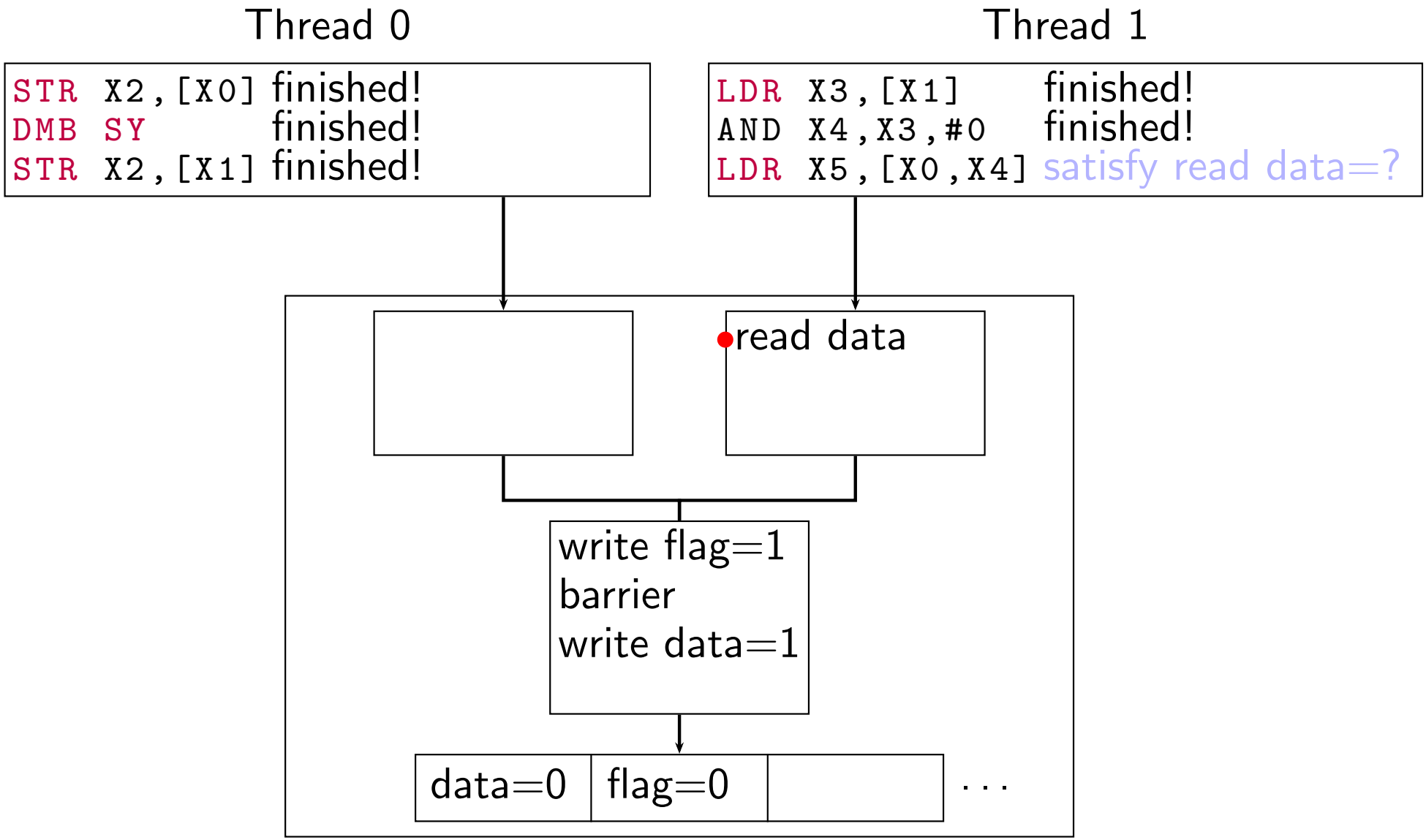
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] read reg. X4=0
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

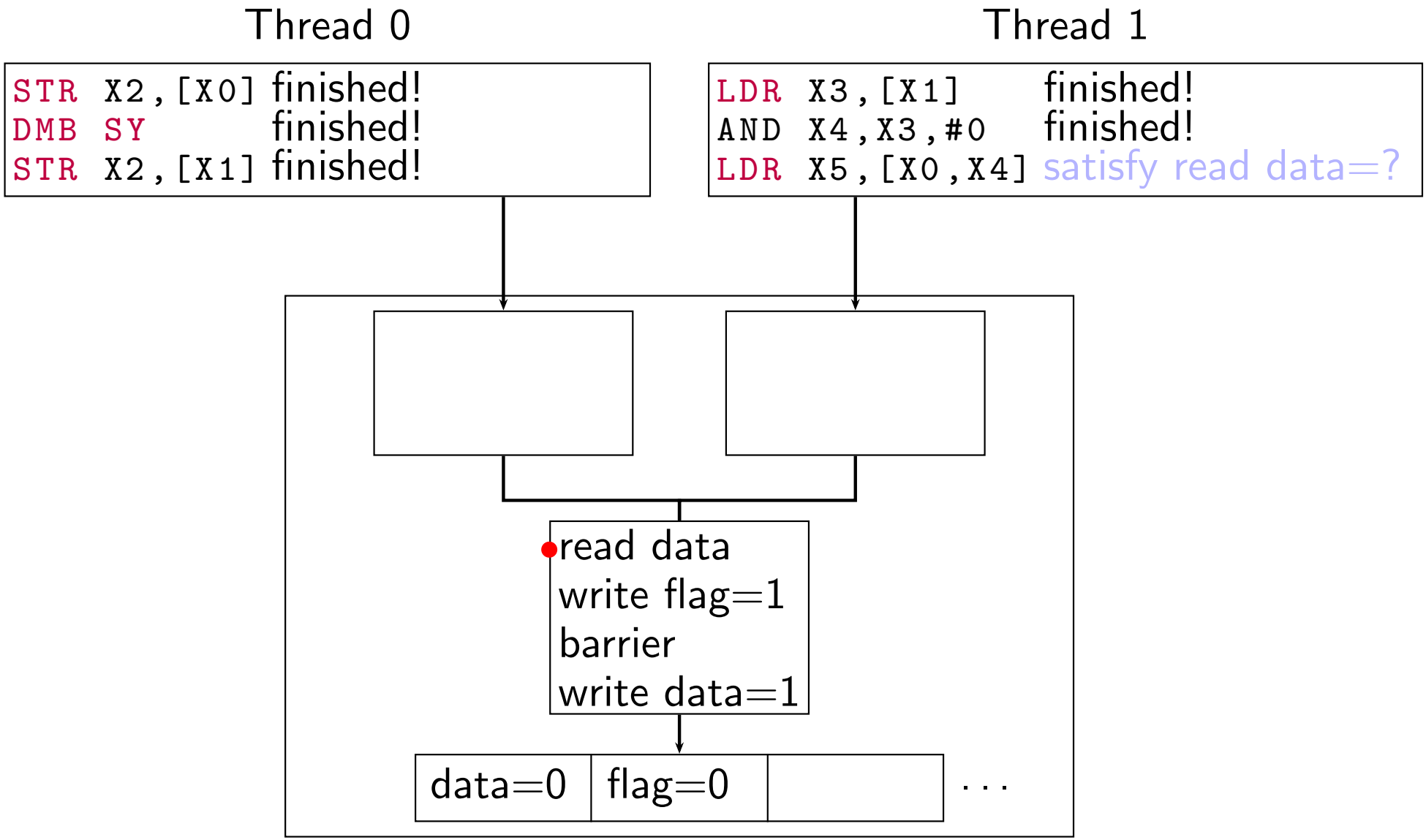
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] request read data
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

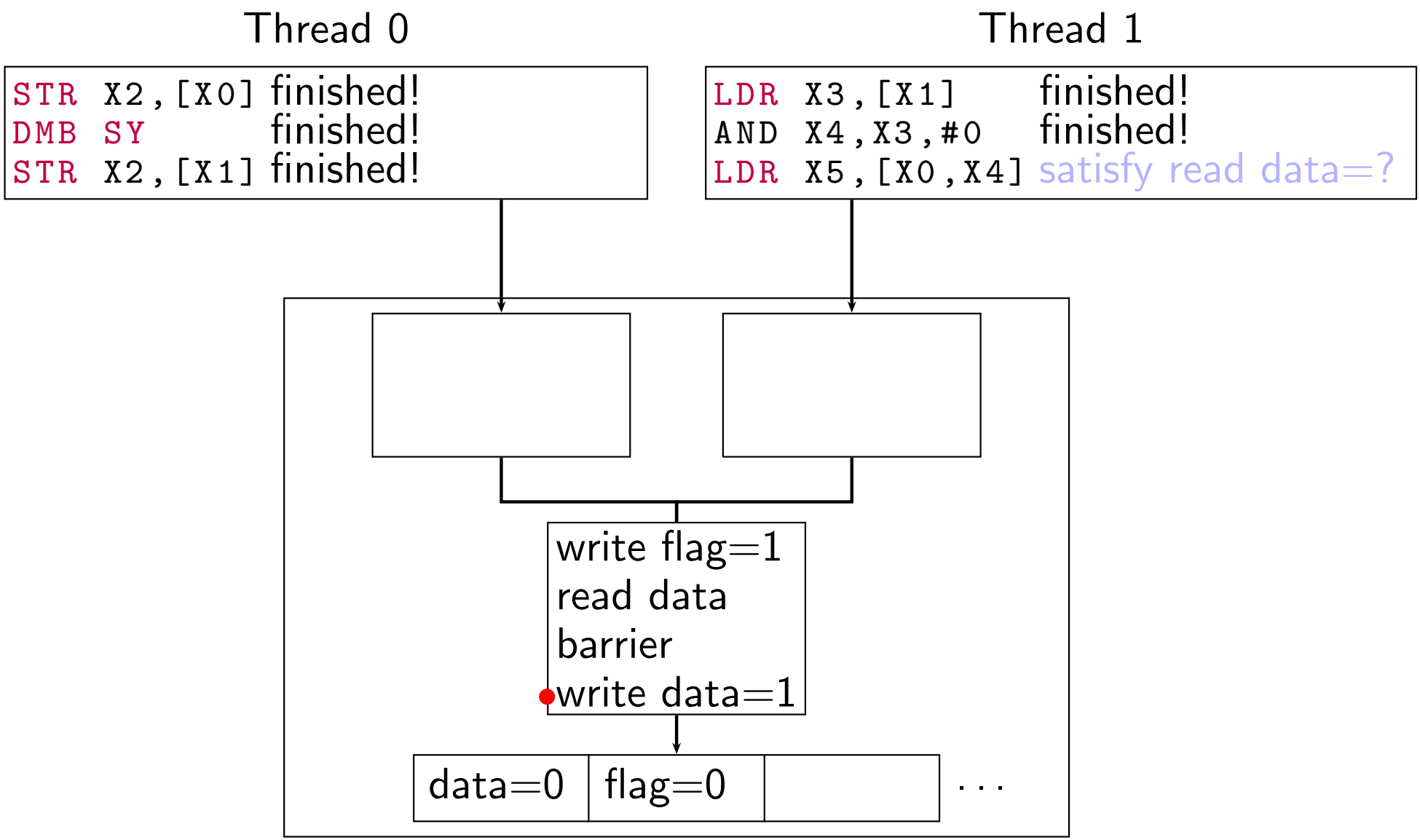
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] satisfy read data=?
read data
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

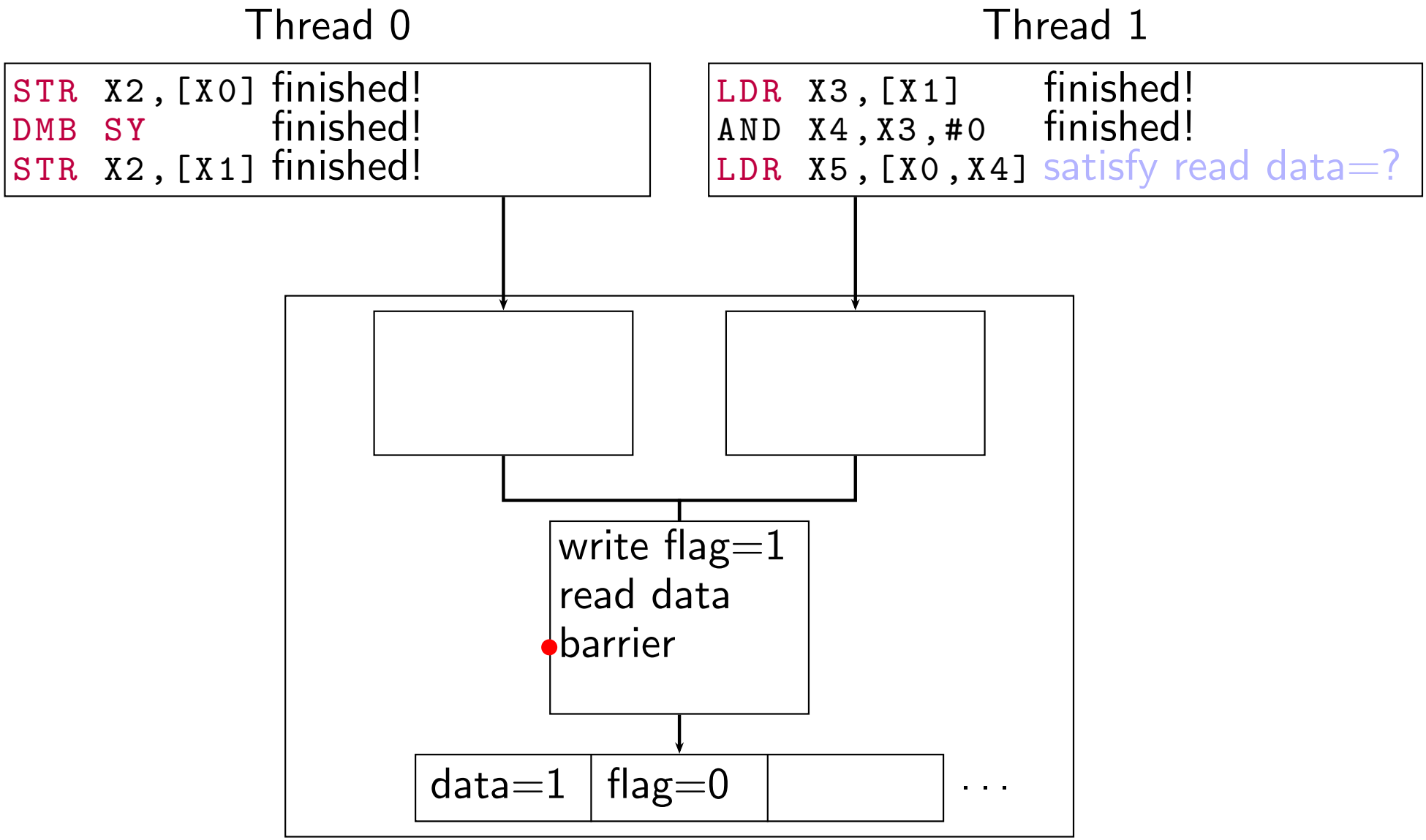
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] satisfy read data=?
read data
write flag=1
barrier
write data=1
data=0
flag=0
· · ·

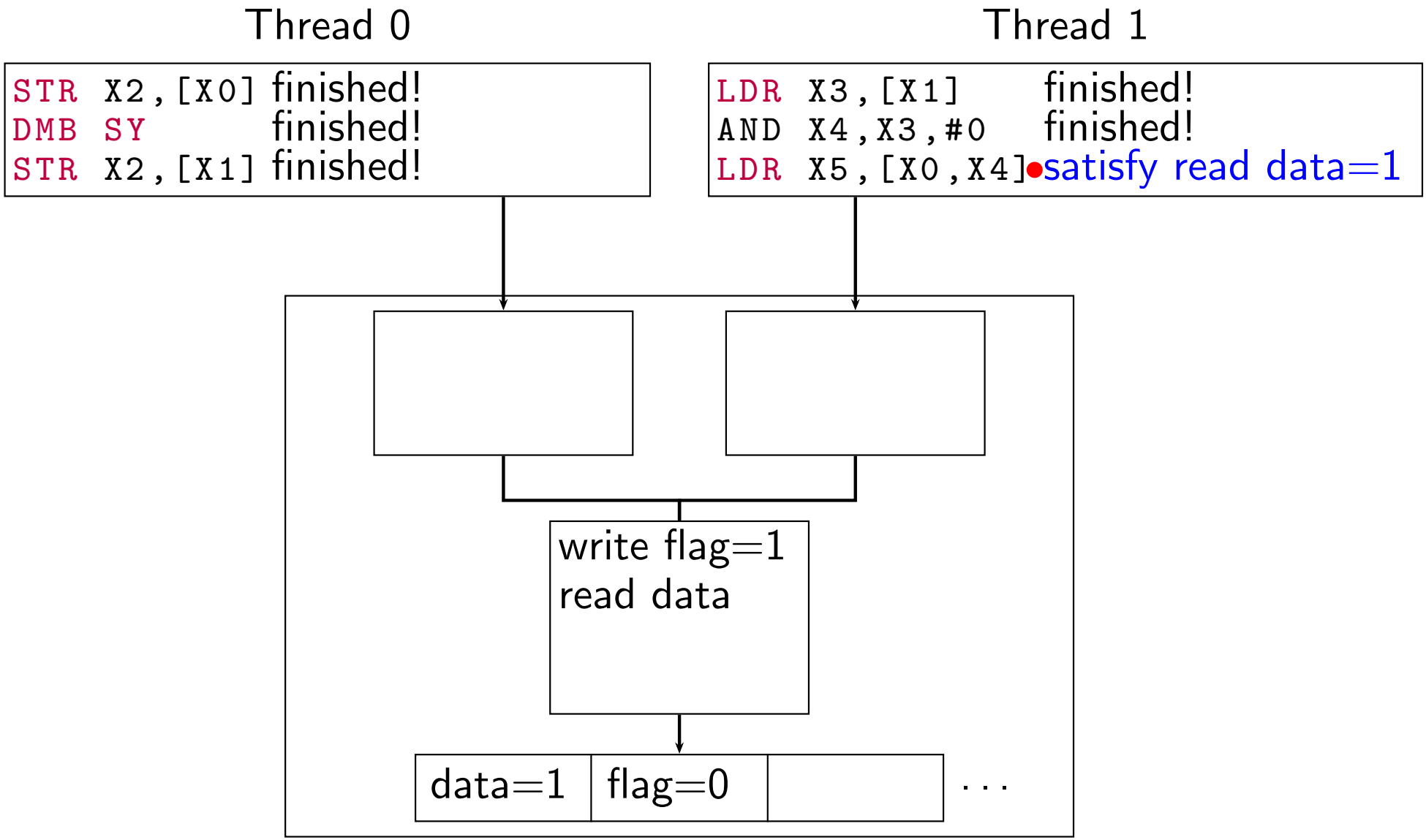
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] satisfy read data=?
write flag=1
read data
barrier
write data=1
data=0
flag=0
· · ·

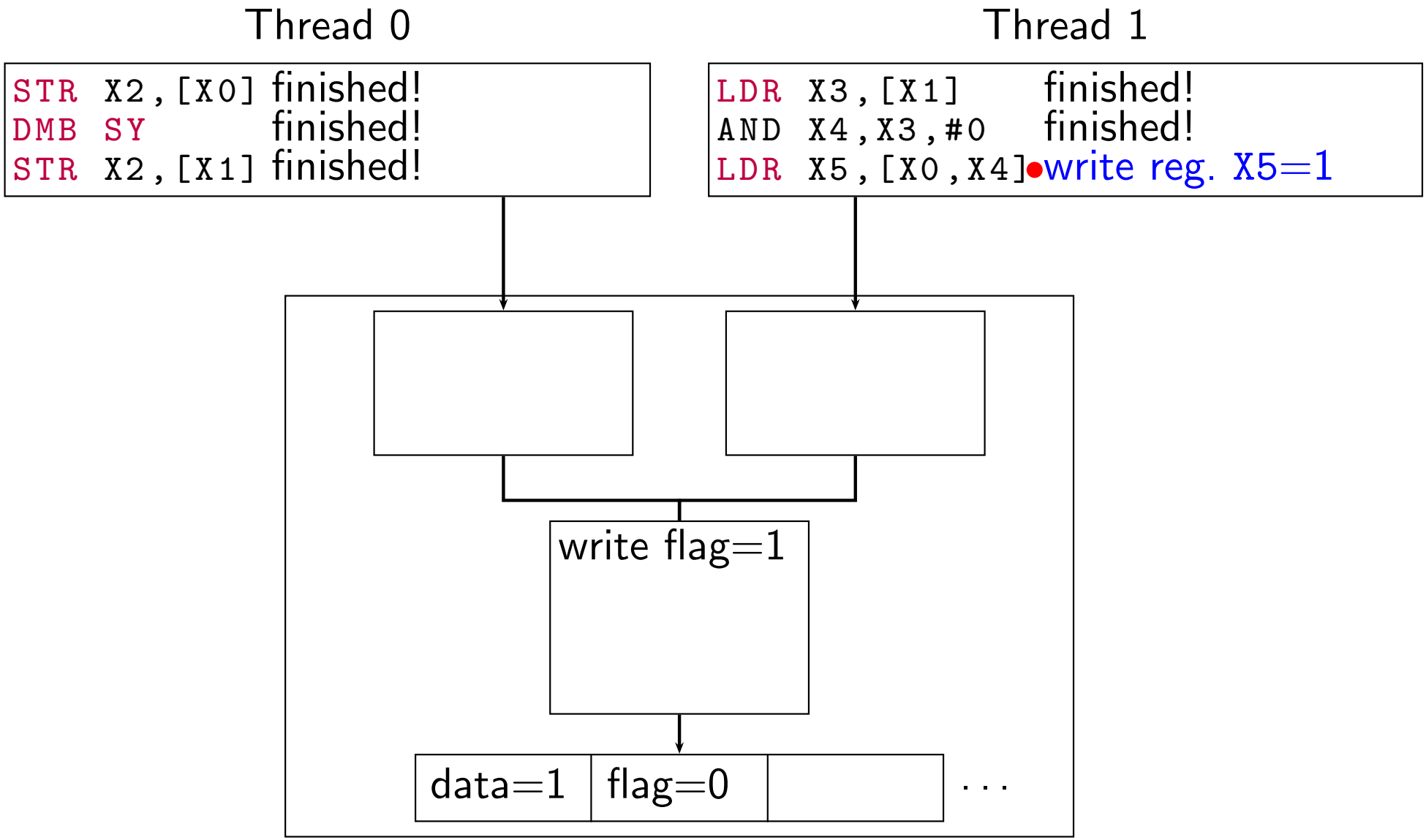
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] satisfy read data=?
write flag=1
read data
barrier
data=1
flag=0
· · ·

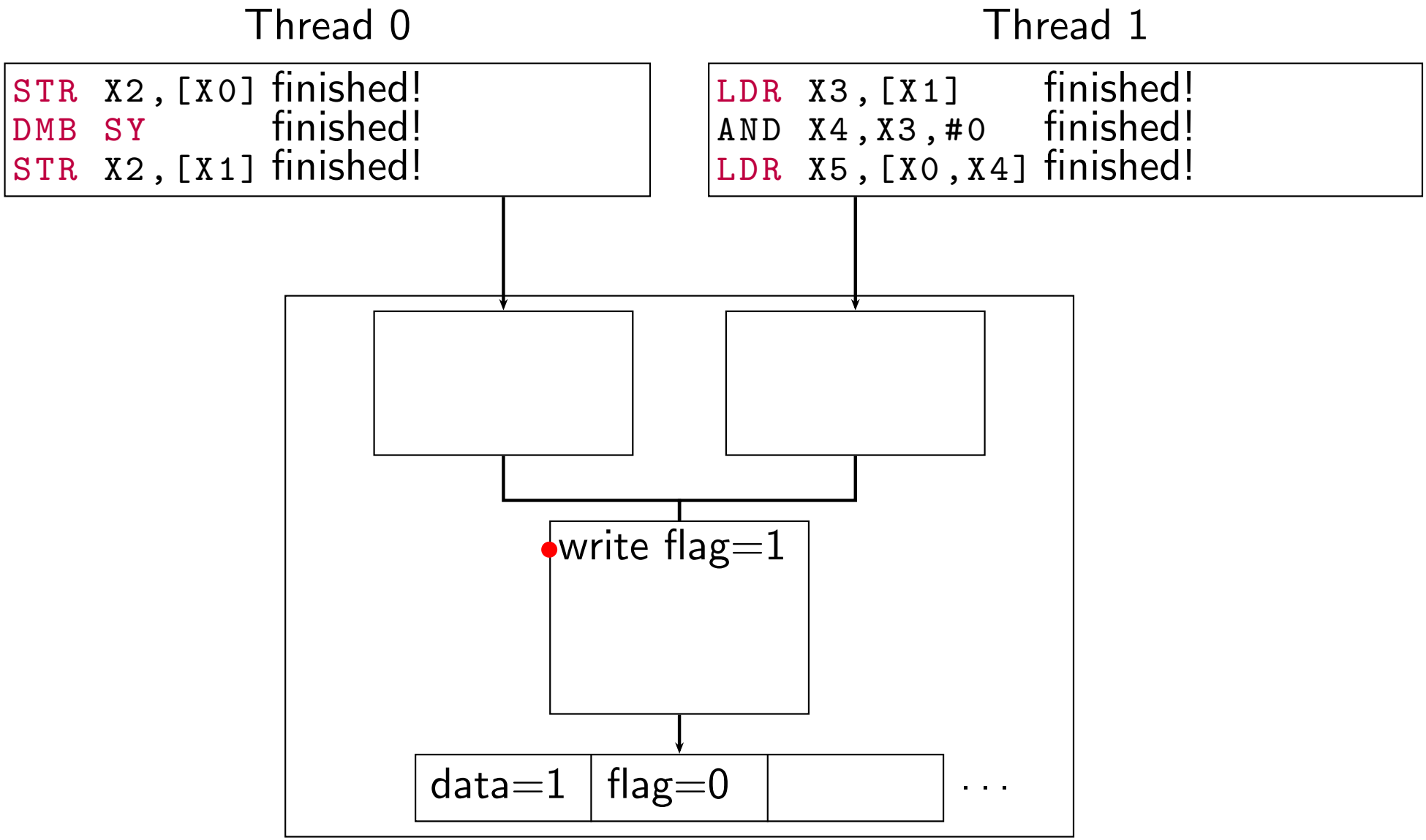
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] satisfy read data=1
write flag=1
read data
data=1
flag=0
· · ·

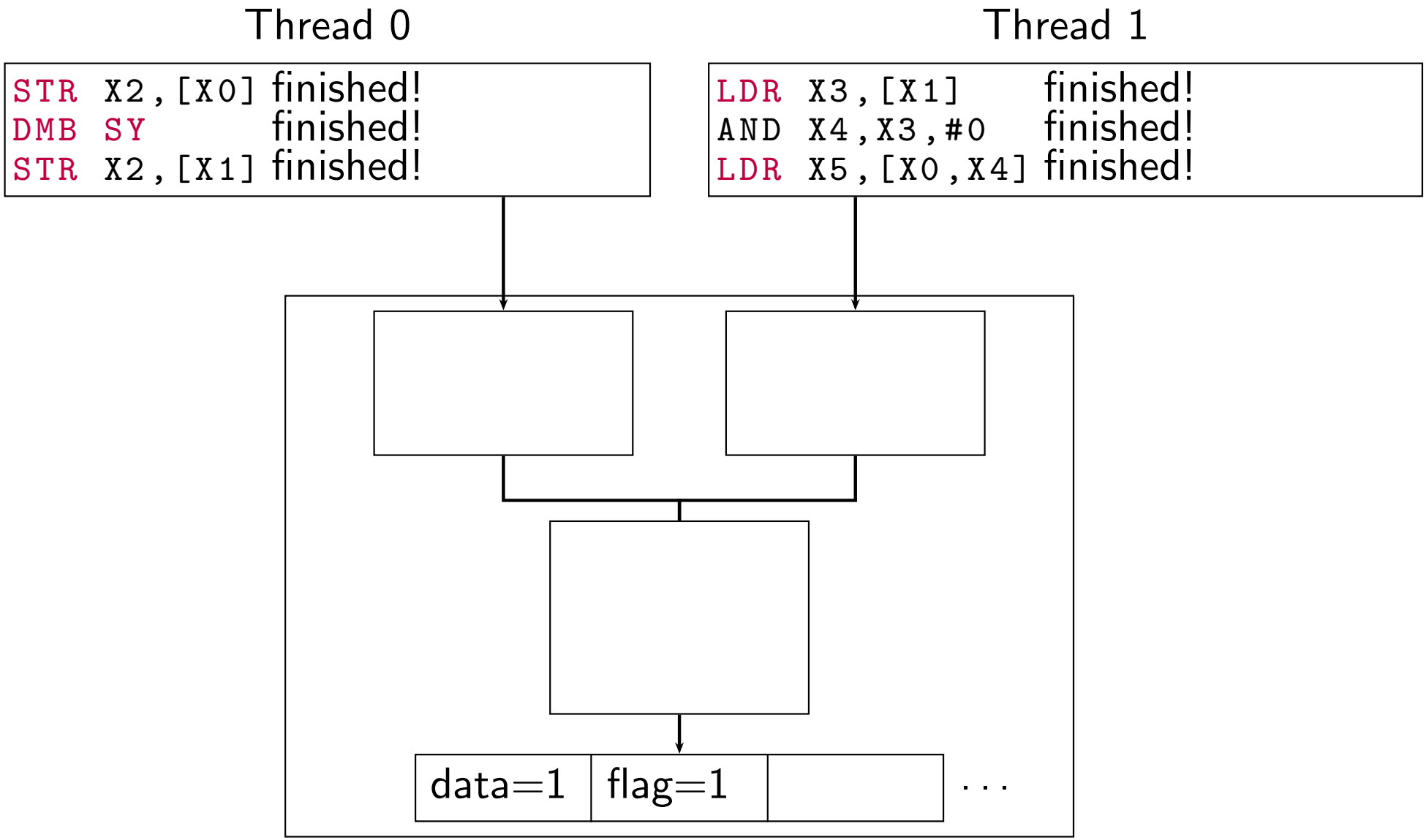
Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] write reg. X5=1
write flag=1
data=1
flag=0
· · ·

Example

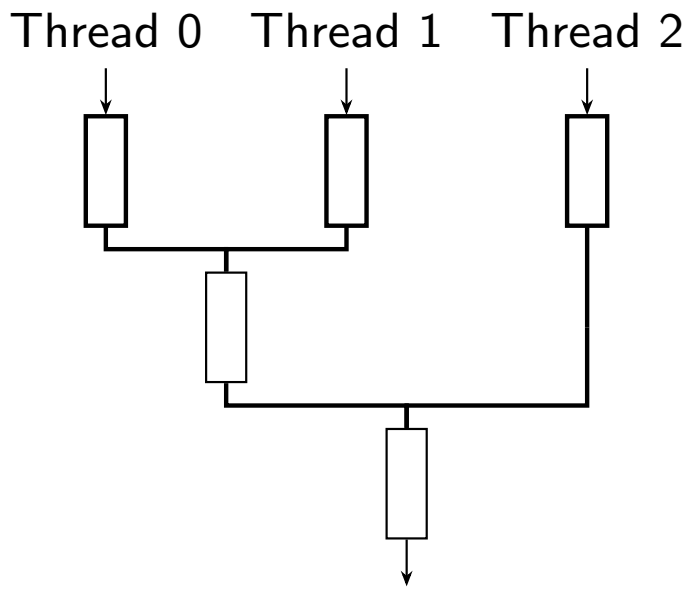

Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] finished!
write flag=1
data=1
flag=0
· · ·

Example


Thread 0
Thread 1
STR X2,[X0] finished!
DMB SY finished!
STR X2,[X1] finished!
LDR X3,[X1] finished!
AND X4,X3,#0 finished!
LDR X5,[X0,X4] finished!
data=1
flag=1
· · ·

Multi-copy Atomicity

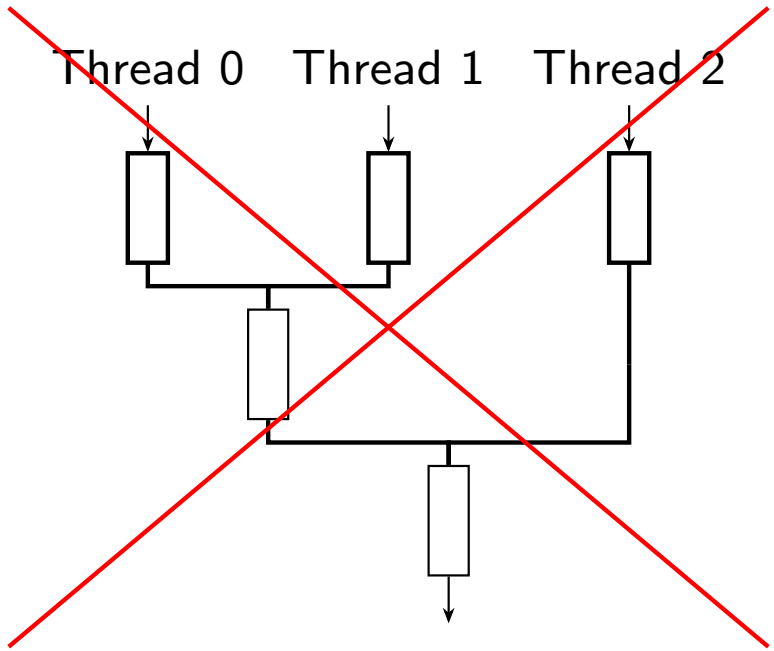
Behaviour that arise from buffer topologies other than the flat one:



This turned out to be more complicated than ARM expected.

Multi-copy Atomicity

Behaviour that arise from buffer topologies other than the flat one:



This turned out to be more complicated than ARM expected.

More Abstract Models

The Flat Model

- Similar thread sub-system
- Storage is a simple array (no buffers)

More Abstract Models

The Flat Model

- Similar thread sub-system
- Storage is a simple array (no buffers)

The Promising Model

- Instead of speculation, promise stores
- Instructions are executed in one step
- Instructions are executed in-order (almost)
- More suitable as basis for program logic

More Abstract Models

The Flat Model

- Similar thread sub-system
- Storage is a simple array (no buffers)

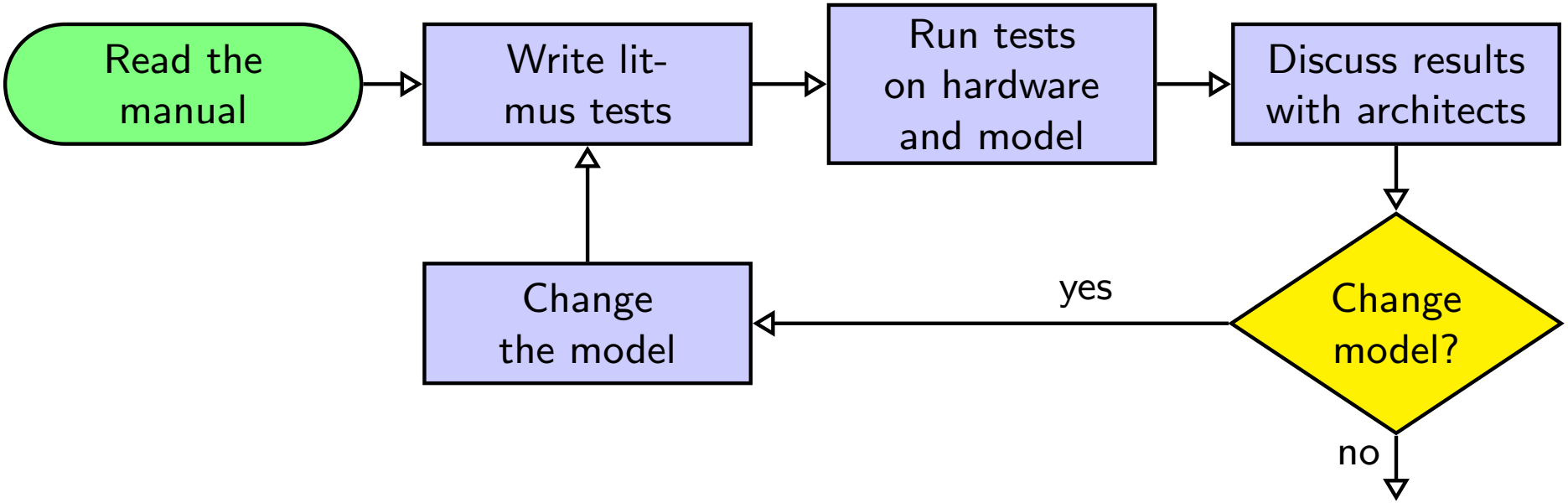
The Promising Model

- Instead of speculation, promise stores
- Instructions are executed in one step
- Instructions are executed in-order (almost)
- More suitable as basis for program logic

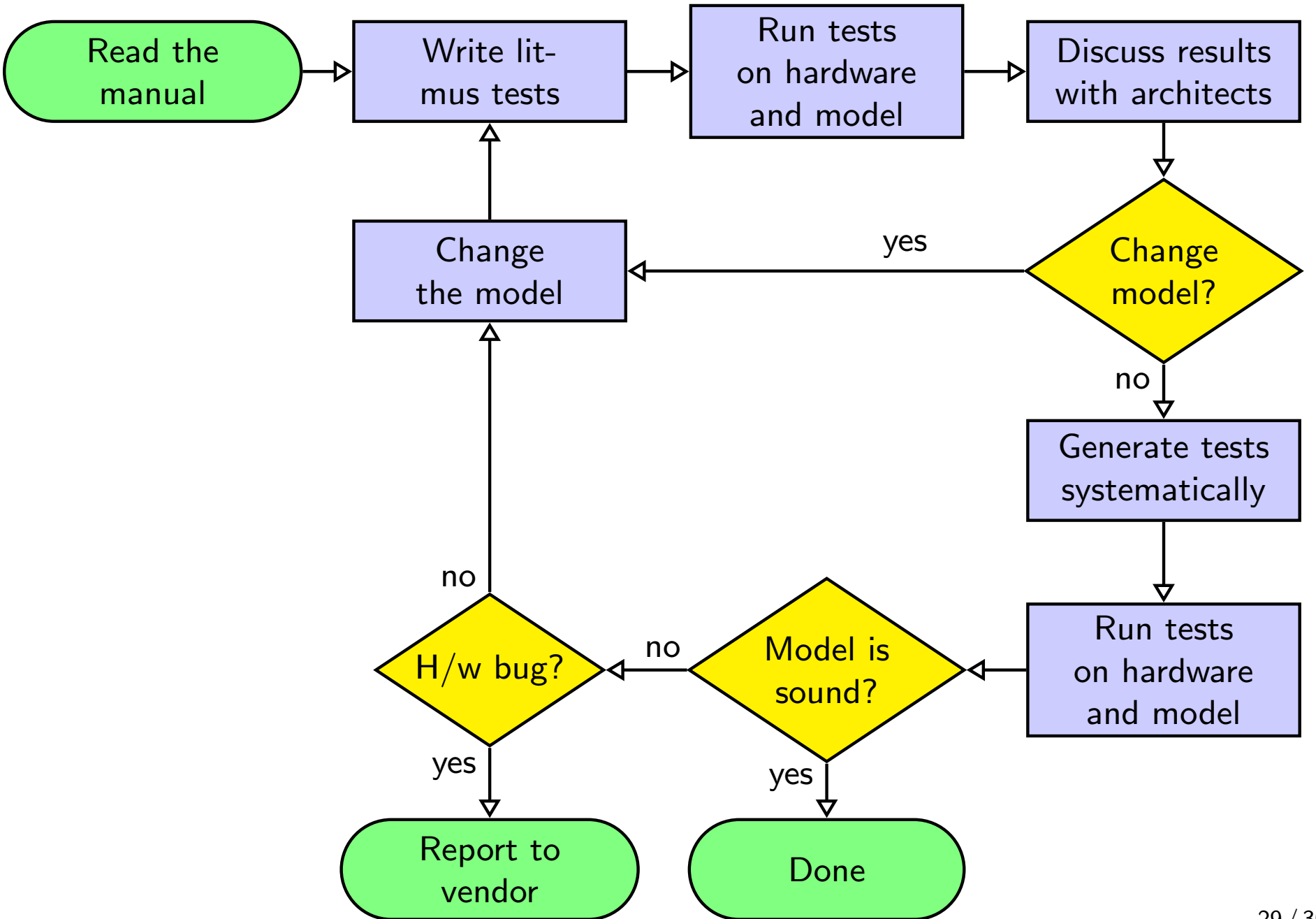
Axiomatic Model

- Very concise
- Appears in the ARM reference manual

Research Methodology



Research Methodology



Contribution

- As a result of my work ARM has changed the memory model
- A reliable model of the ARMv8 architecture
- Machine readable definitions
- A tool, derived from the definitions (RMEM)
- Bugs in commercial hardware
- Similar work for RISC-V

Contribution

- As a result of my work ARM has changed the memory model
- A reliable model of the ARMv8 architecture
- Machine readable definitions
- A tool, derived from the definitions (RMEM)
- Bugs in commercial hardware
- Similar work for RISC-V

Thank you