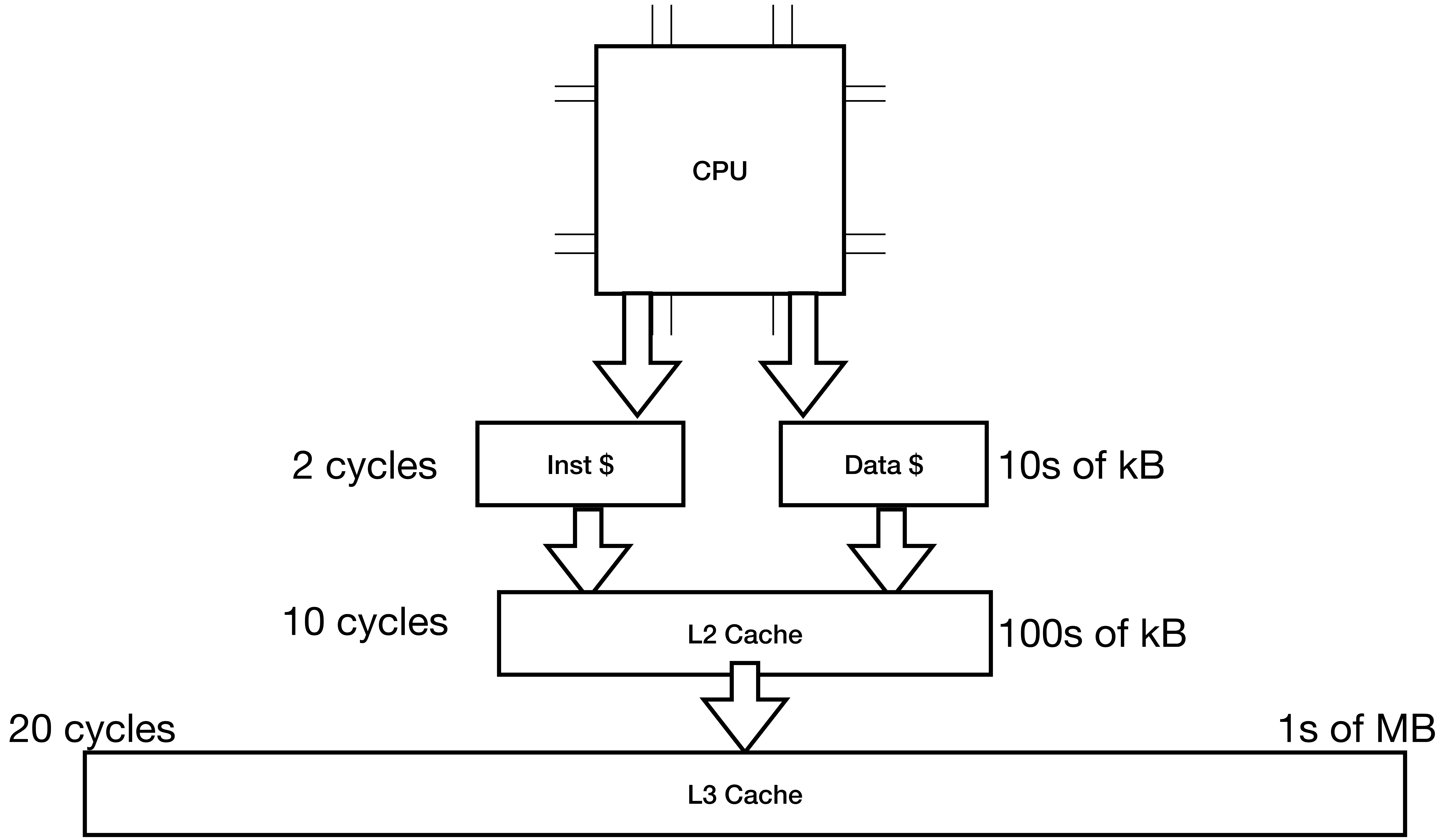
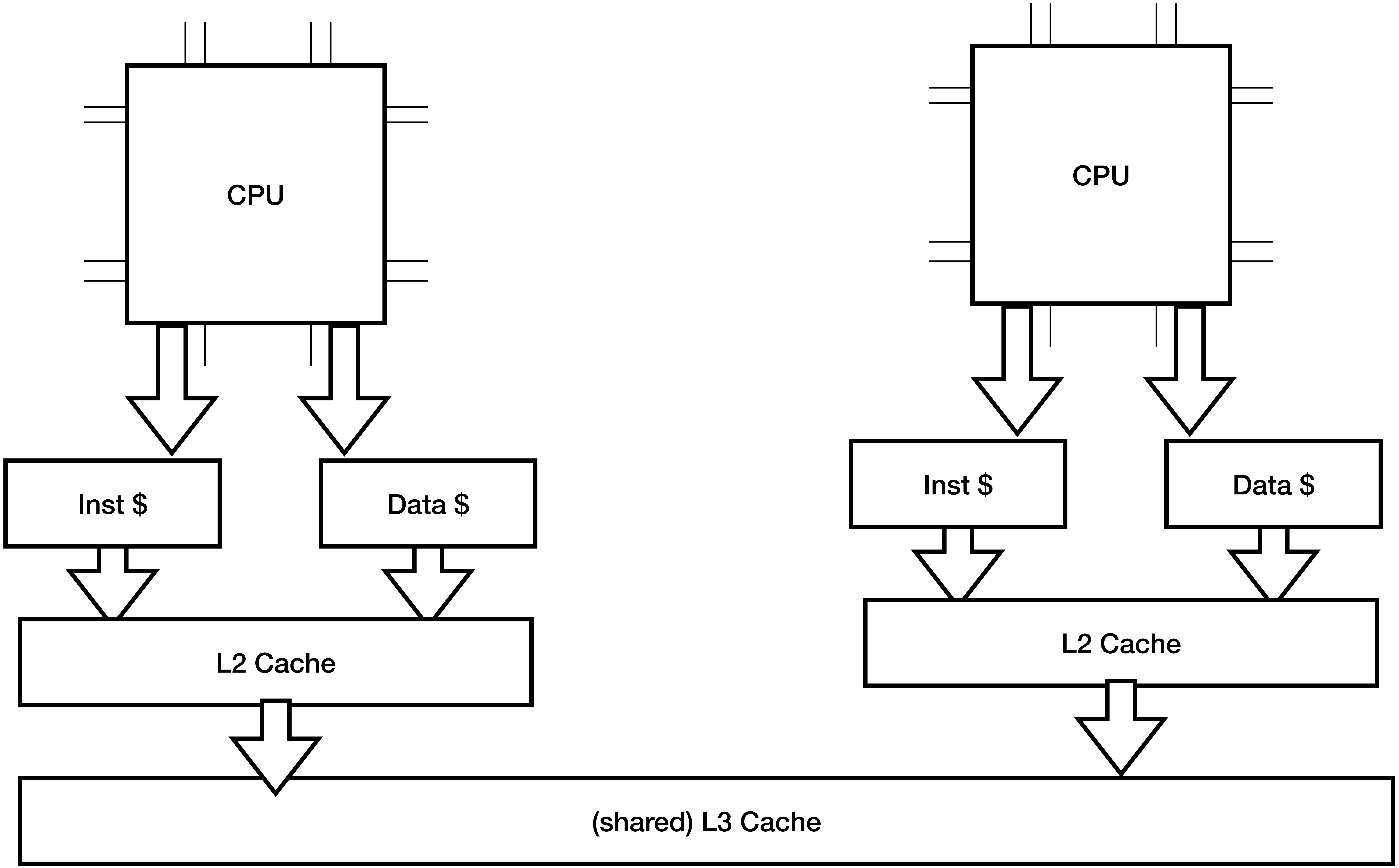
Caches: Coherence, Synchronization, Consistency

Outline for Today

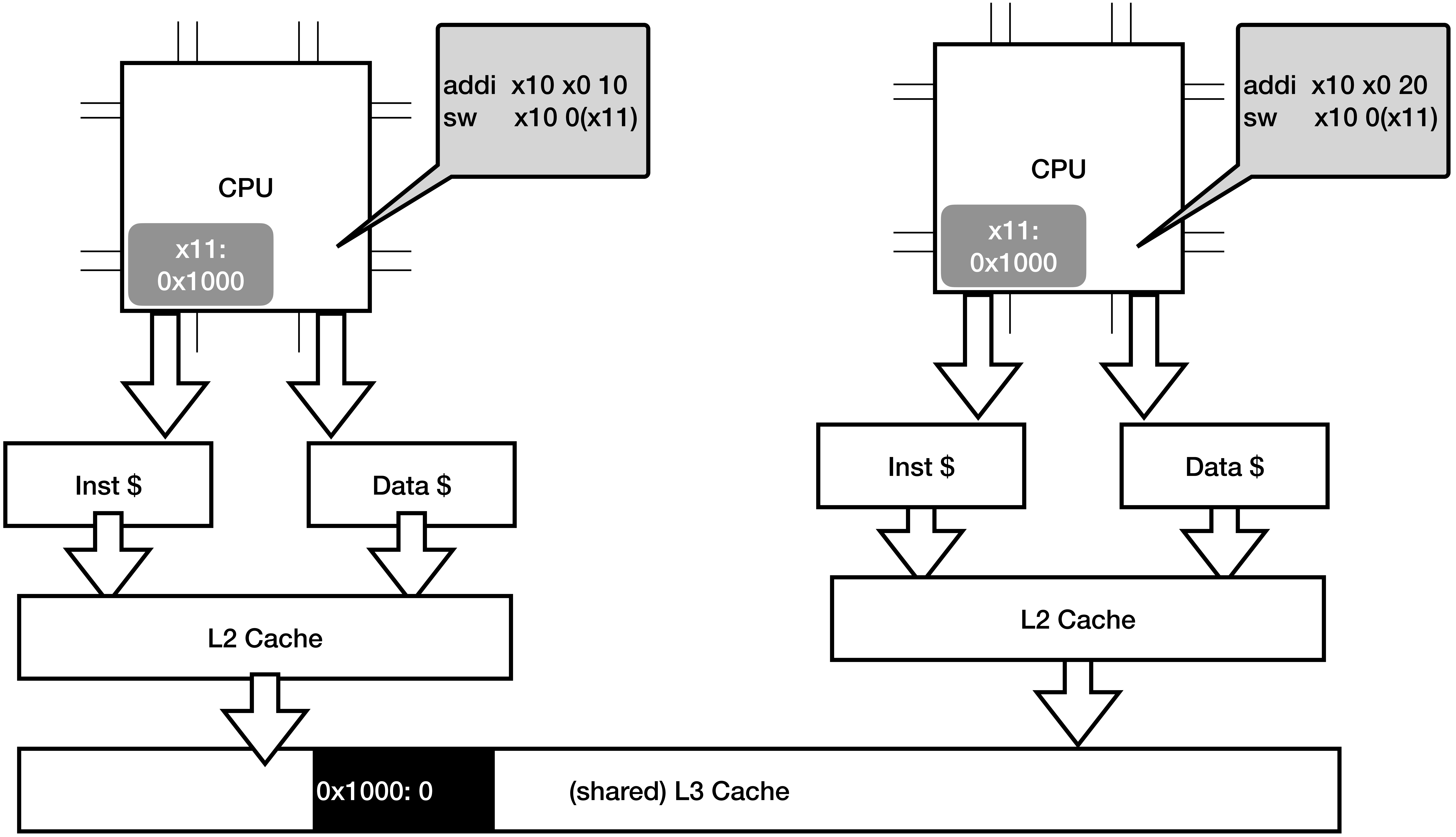
- Revisit the cache hierarchy
- Describing snooping coherence protocol
- Implementing fences in hardware
- Highlighting relaxed memory models


CPU
2 cycles
Inst $
Data $
10s of kB
10 cycles
L2 Cache
100s of kB
20 cycles
1s of MB
L3 Cache

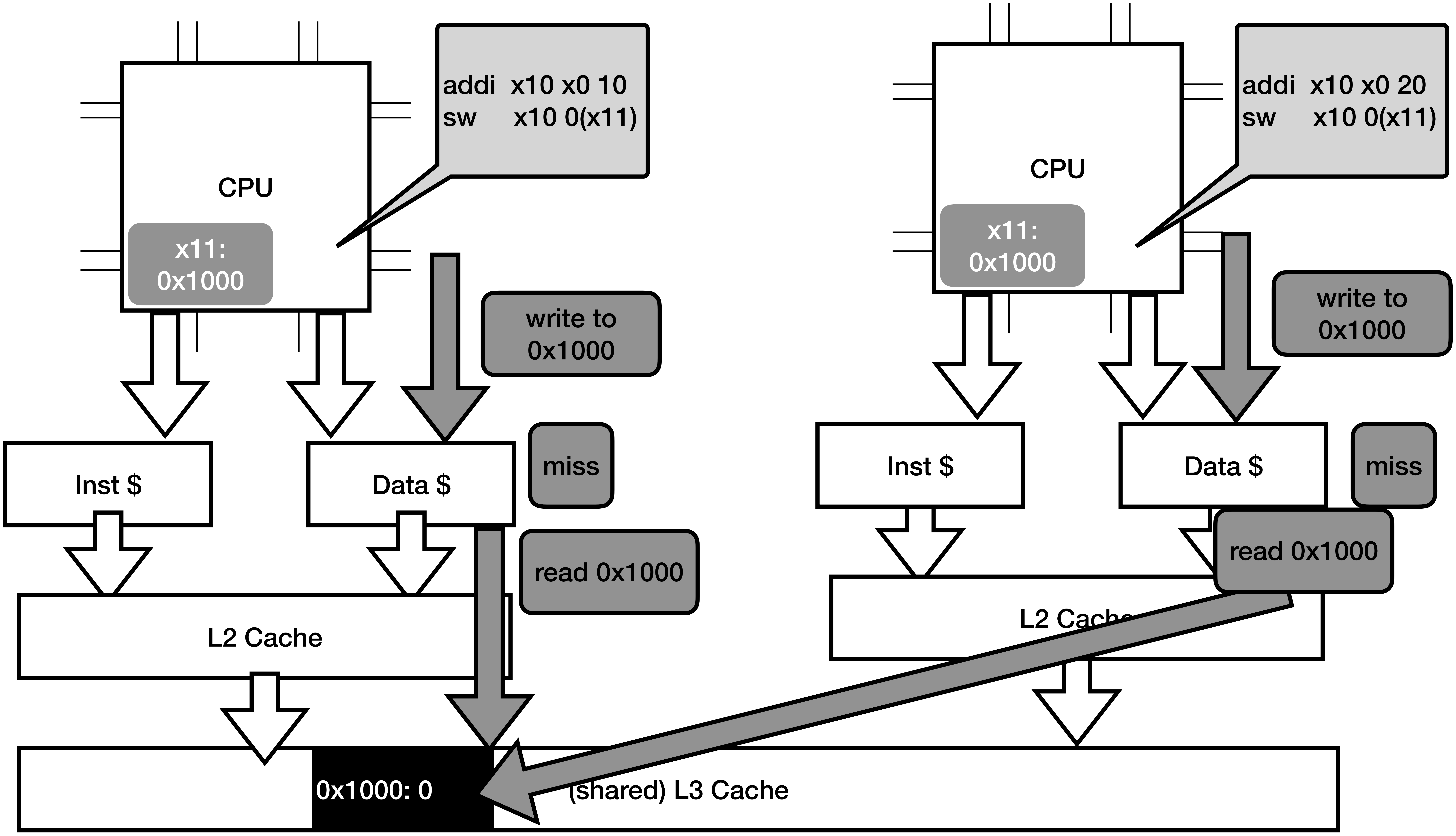

CPU
CPU
Inst $
Data $
Inst $
Data $
L2 Cache
L2 Cache
(shared) L3 Cache

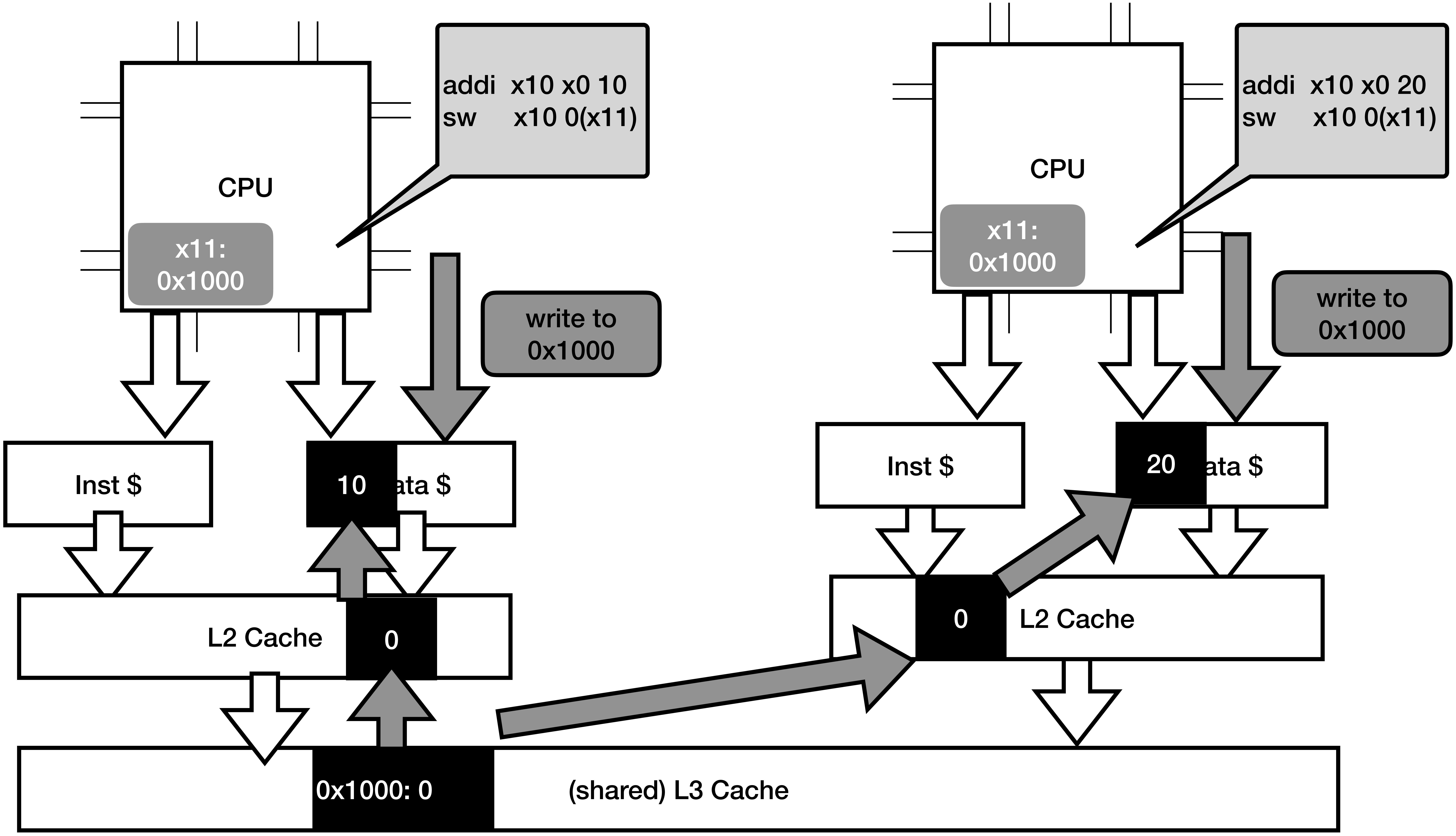
Chat with your neighbor!

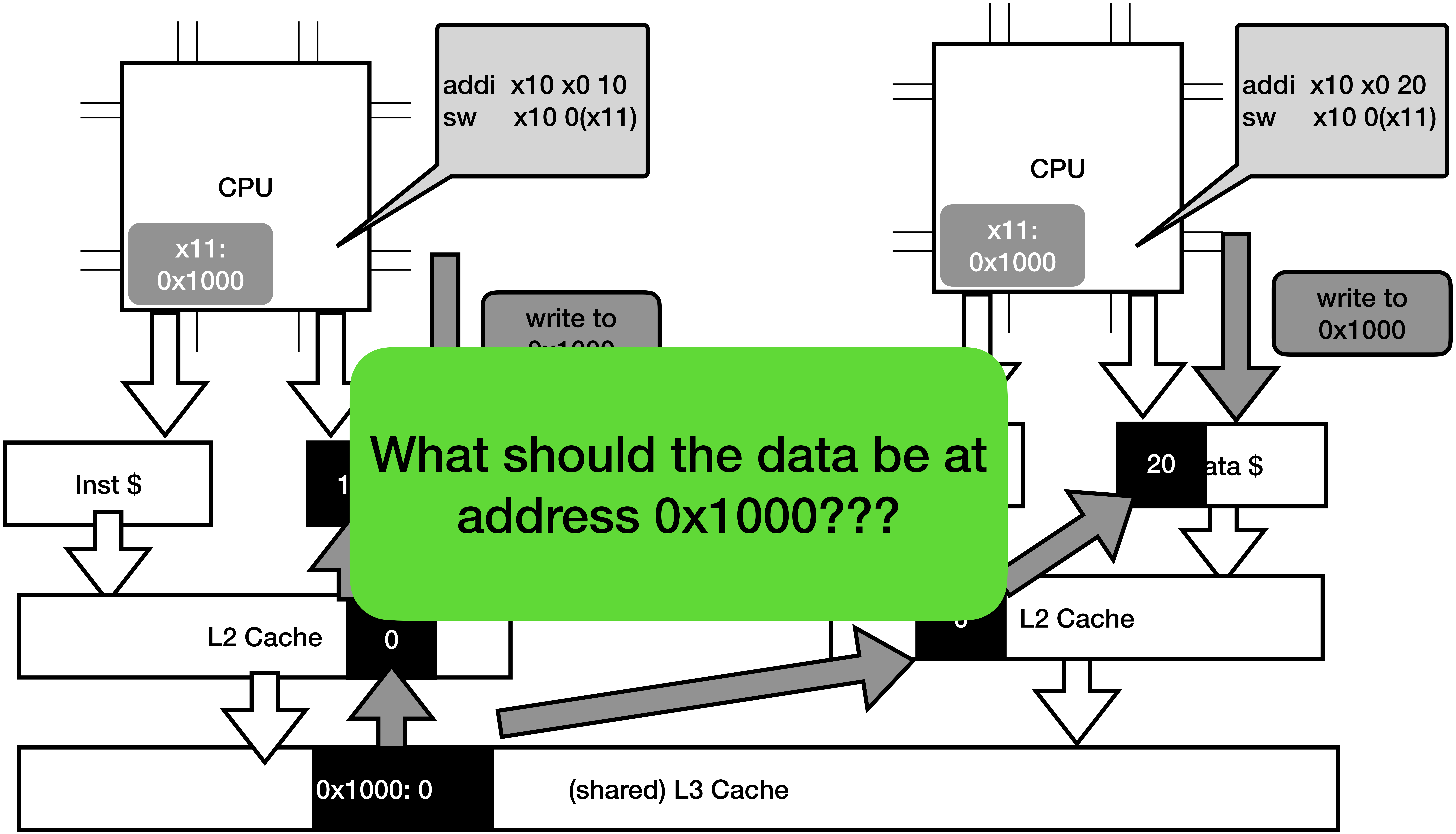
- What about this cache hierarchy works?
- What about this cache hierarchy doesn’t work?



addi x10 x0 10
sw x10 0(x11)
CPU
x11: 0x1000
Inst $
Data $
L2 Cache
addi x10 x0 20
sw x10 0(x11)
CPU
x11: 0x1000
Inst $
Data $
L2 Cache
0x1000: 0
(shared) L3 Cache


CPU
x11: 0x1000
addi x10 x0 10
sw x10 0(x11)
write to 0x1000
Inst $
Data $
miss
read 0x1000
L2 Cache
CPU
x11: 0x1000
addi x10 x0 20
sw x10 0(x11)
write to 0x1000
Inst $
Data $
miss
read 0x1000
L2 Cache
0x1000: 0
(shared) L3 Cache


addi x10 x0 10
sw x10 0(x11)
CPU
x11: 0x1000
write to 0x1000
Inst $
10
ata $
L2 Cache
0
0x1000: 0
(shared) L3 Cache
addi x10 x0 20
sw x10 0(x11)
CPU
x11: 0x1000
write to 0x1000
Inst $
20
ata $
0
L2 Cache

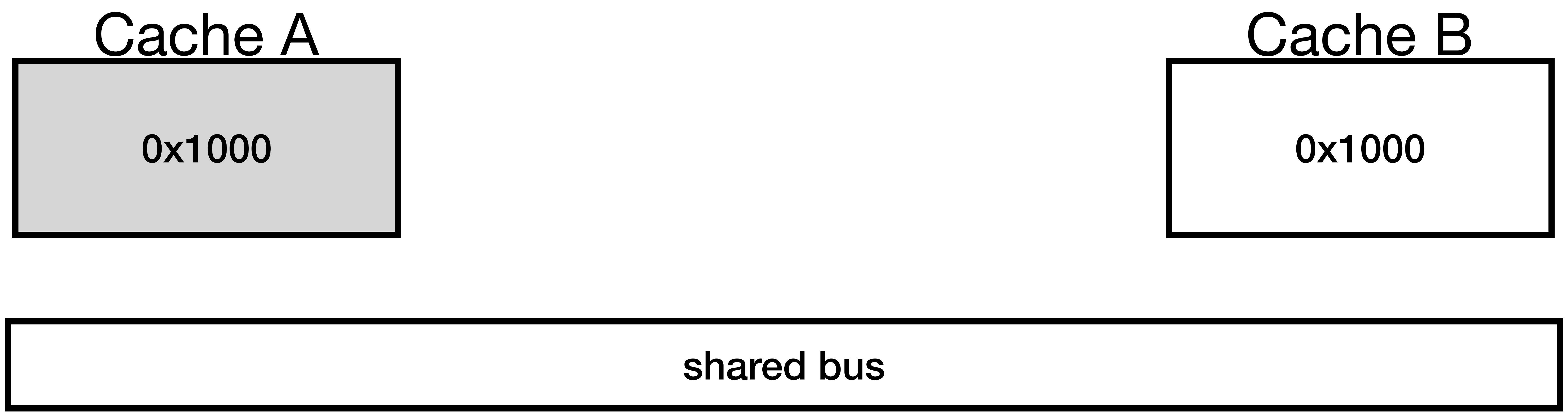


What should the data be at address 0x1000???

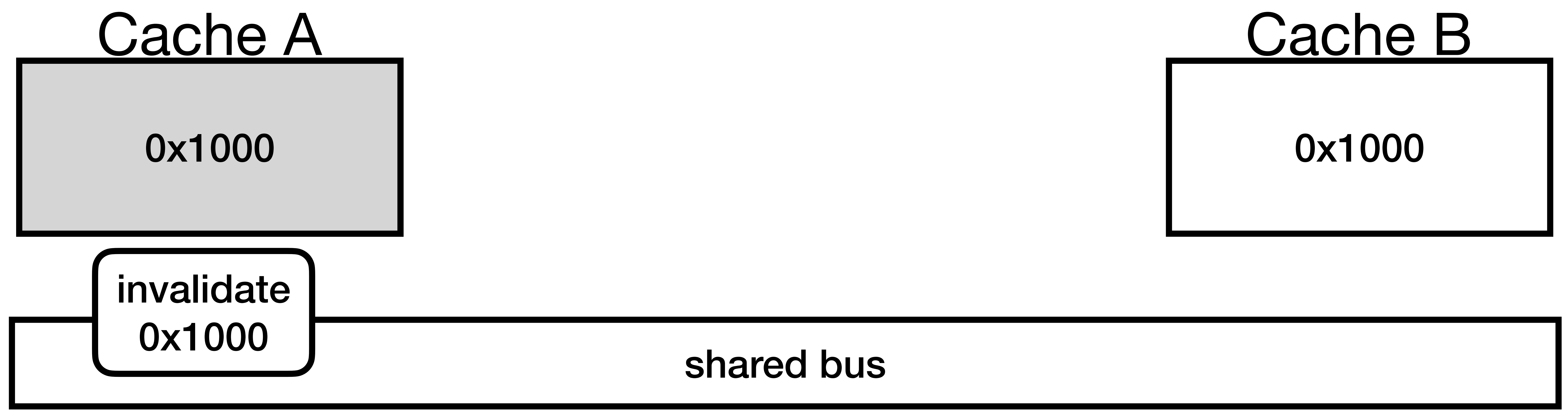
Definitions

- Coherence: what value should be returned by a read?
- Synchronization: how can software reason about the state of data?
- Consistency: in what order can memory operations occur?

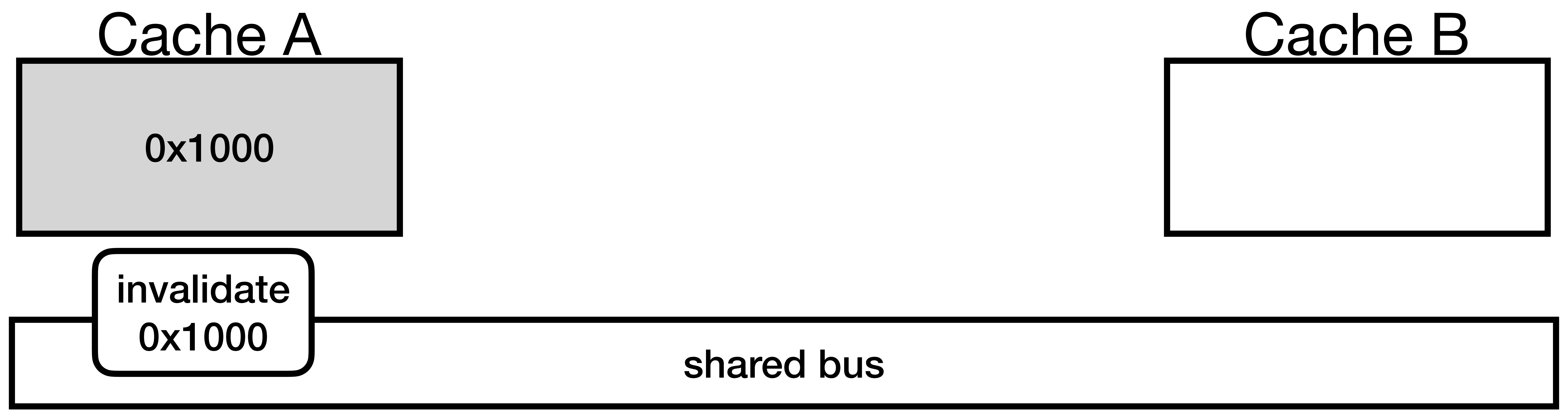
Snooping Coherence



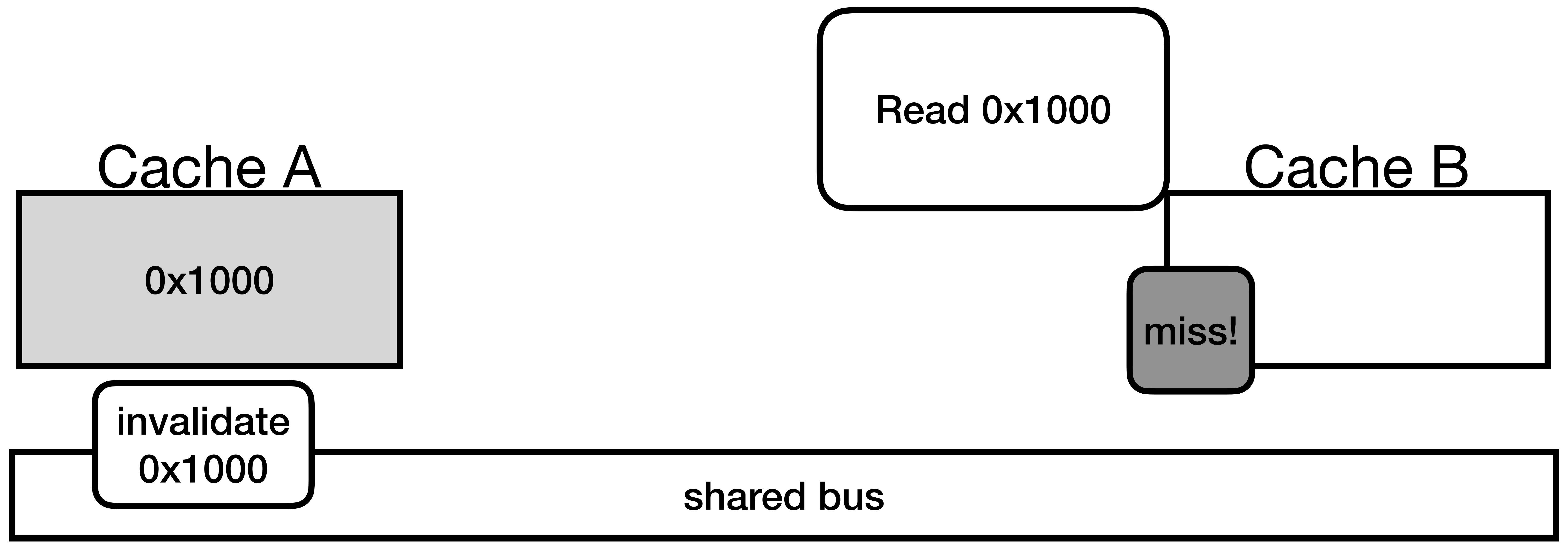
Snooping Coherence



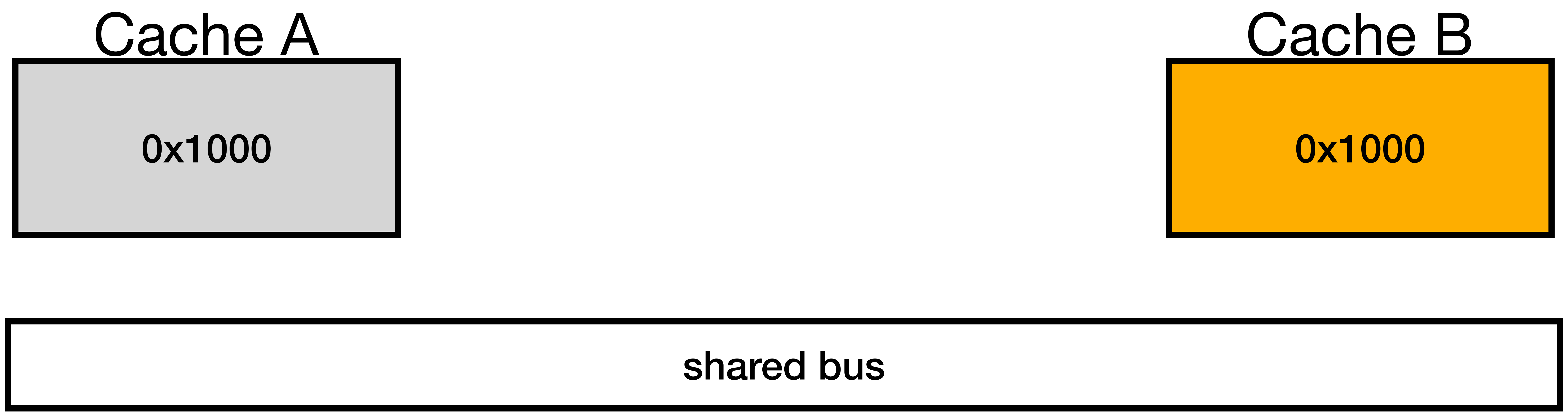
Snooping Coherence



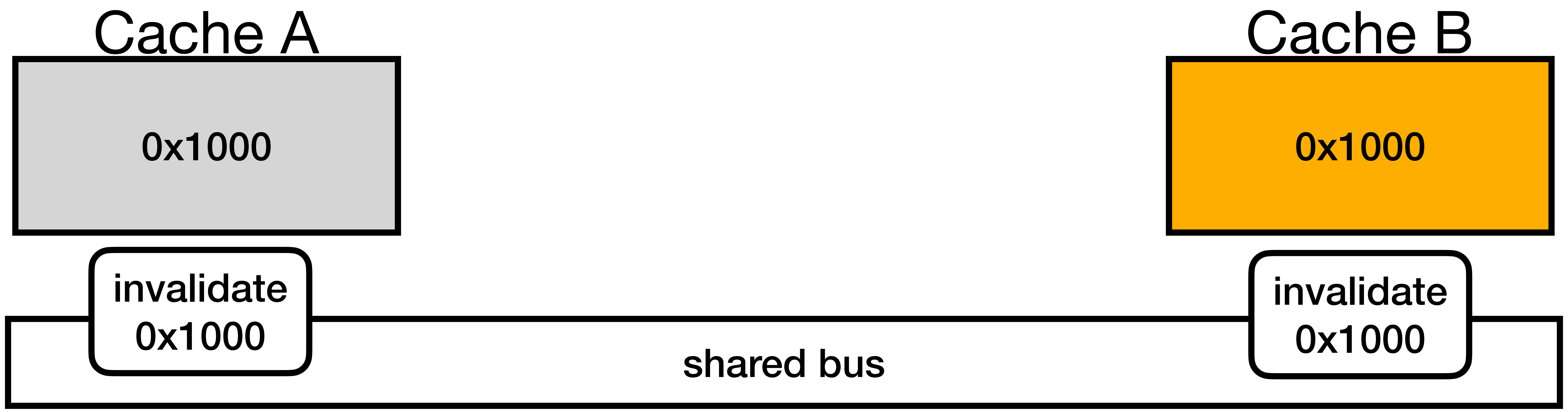
Snooping Coherence



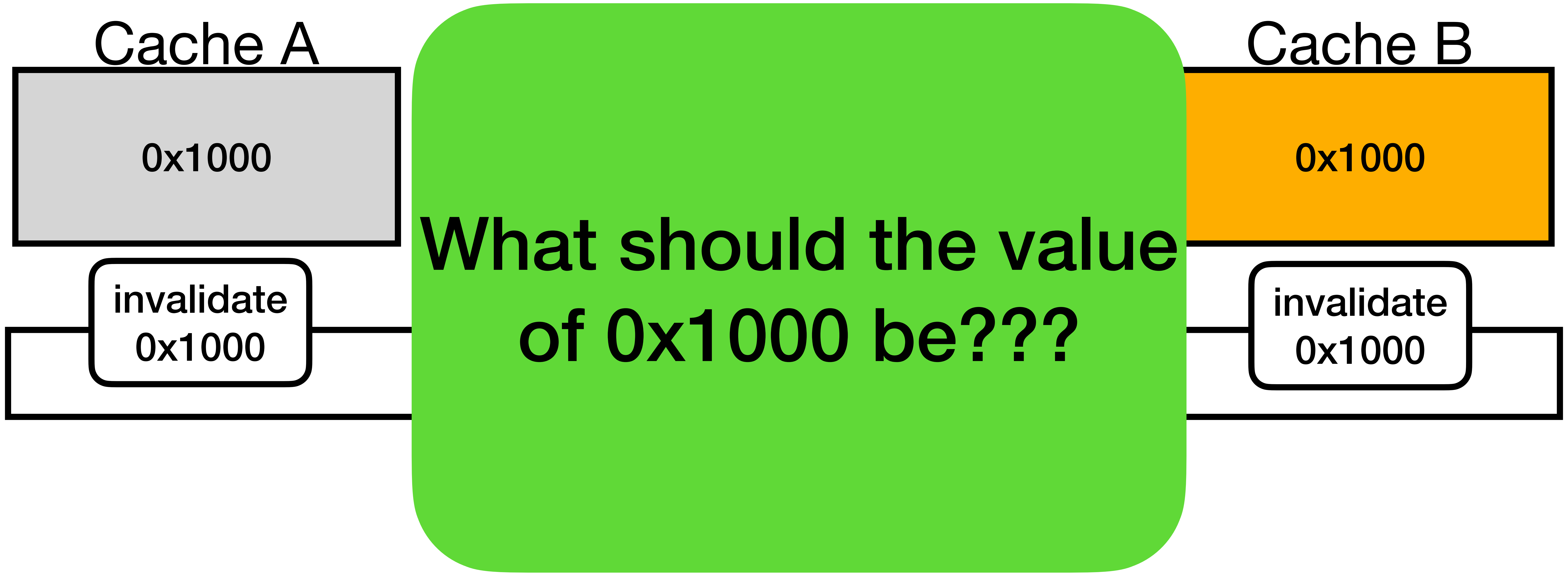
Snooping Coherence



Snooping Coherence

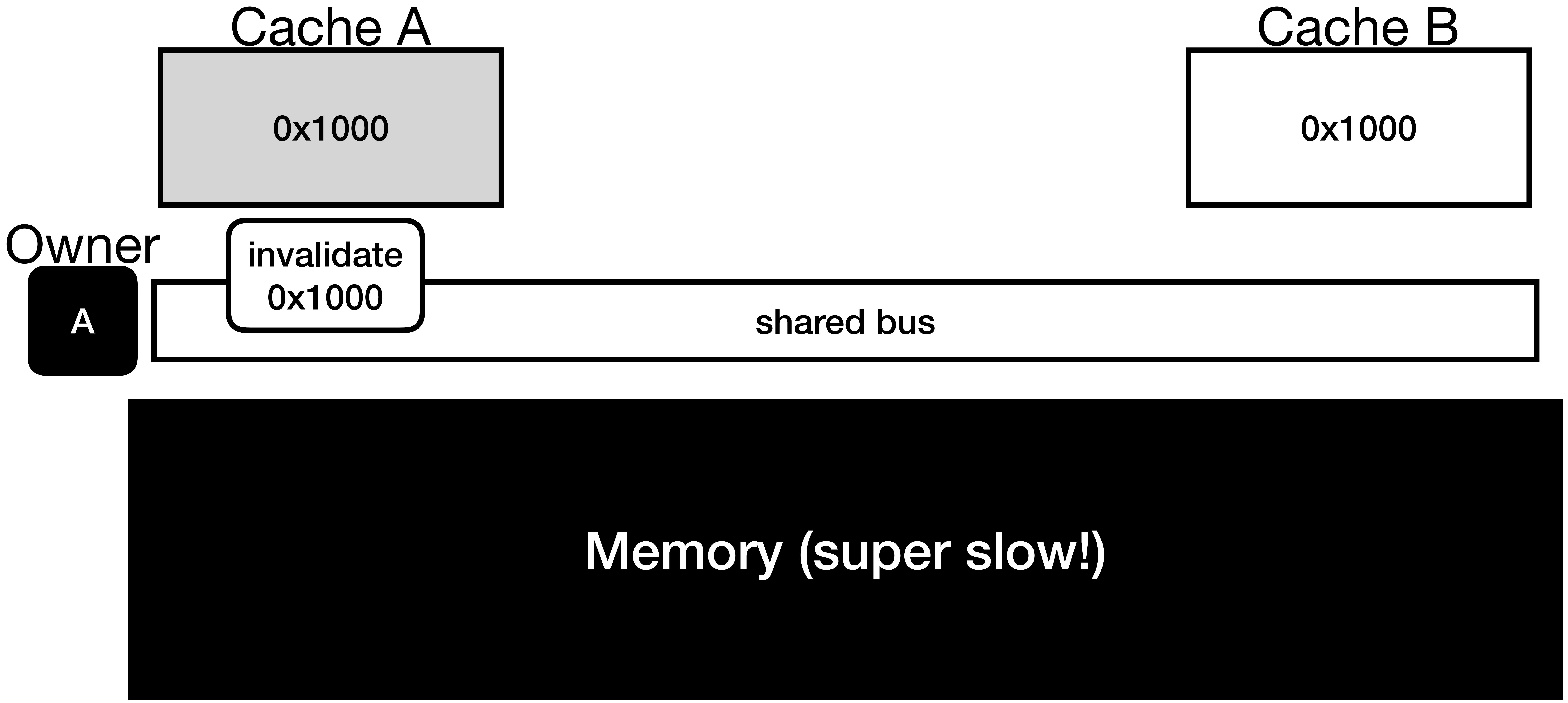


Snooping Coherence

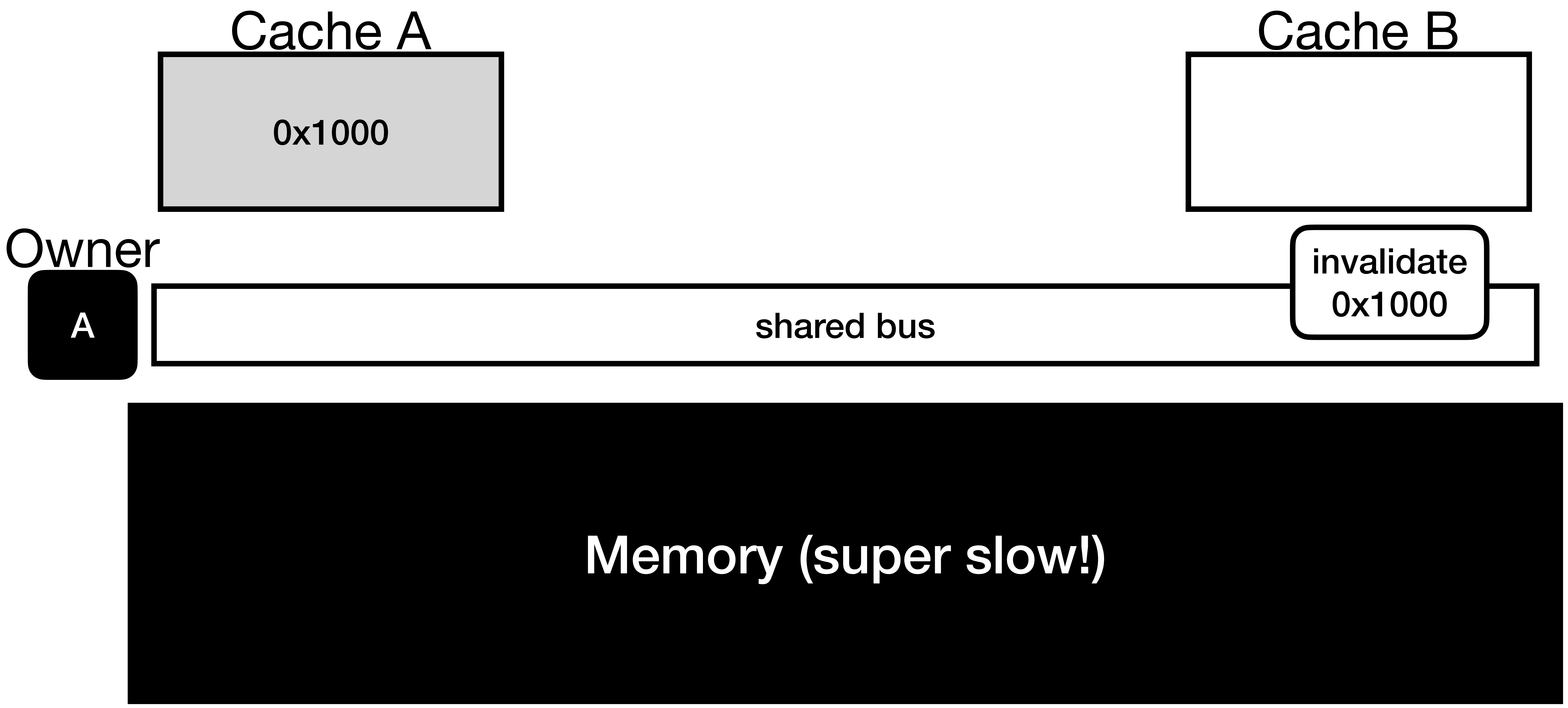


What should the value of 0x1000 be???

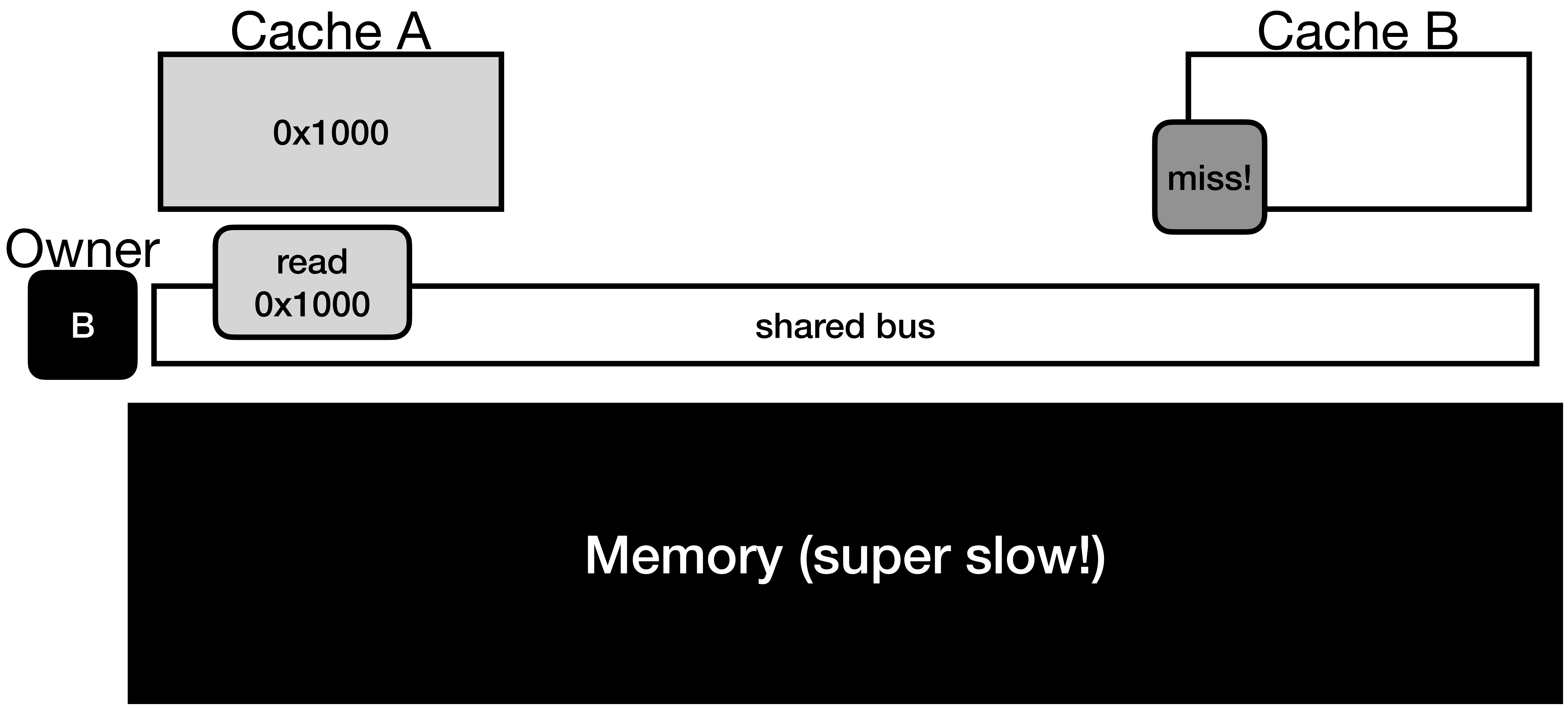
Snooping Caches



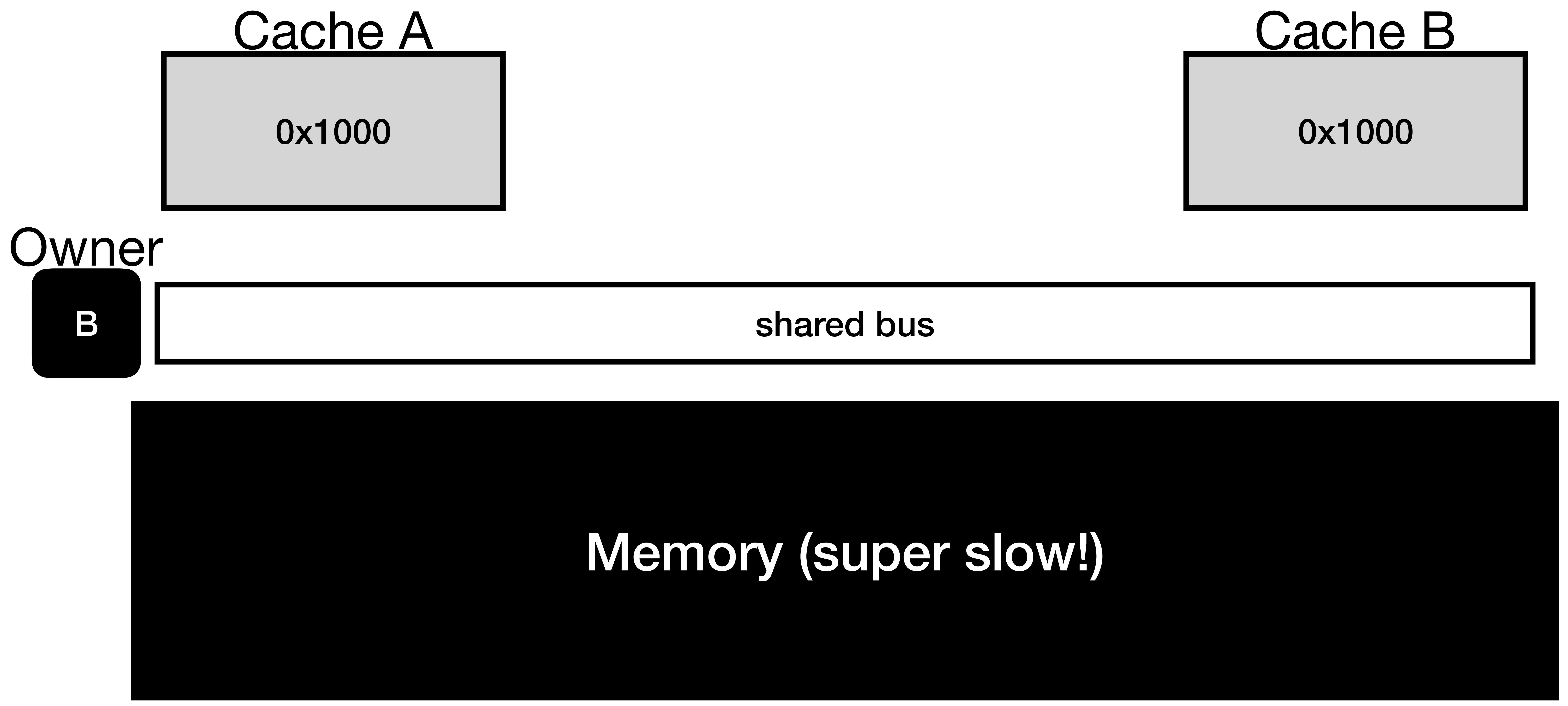
Snooping Caches



Snooping Caches



Snooping Caches



Snooping Coherence

- Scheme is called *write-invalidate*
- The consistency model from this scheme is called *write serialization*
- “Snoops” are messages across the shared bus

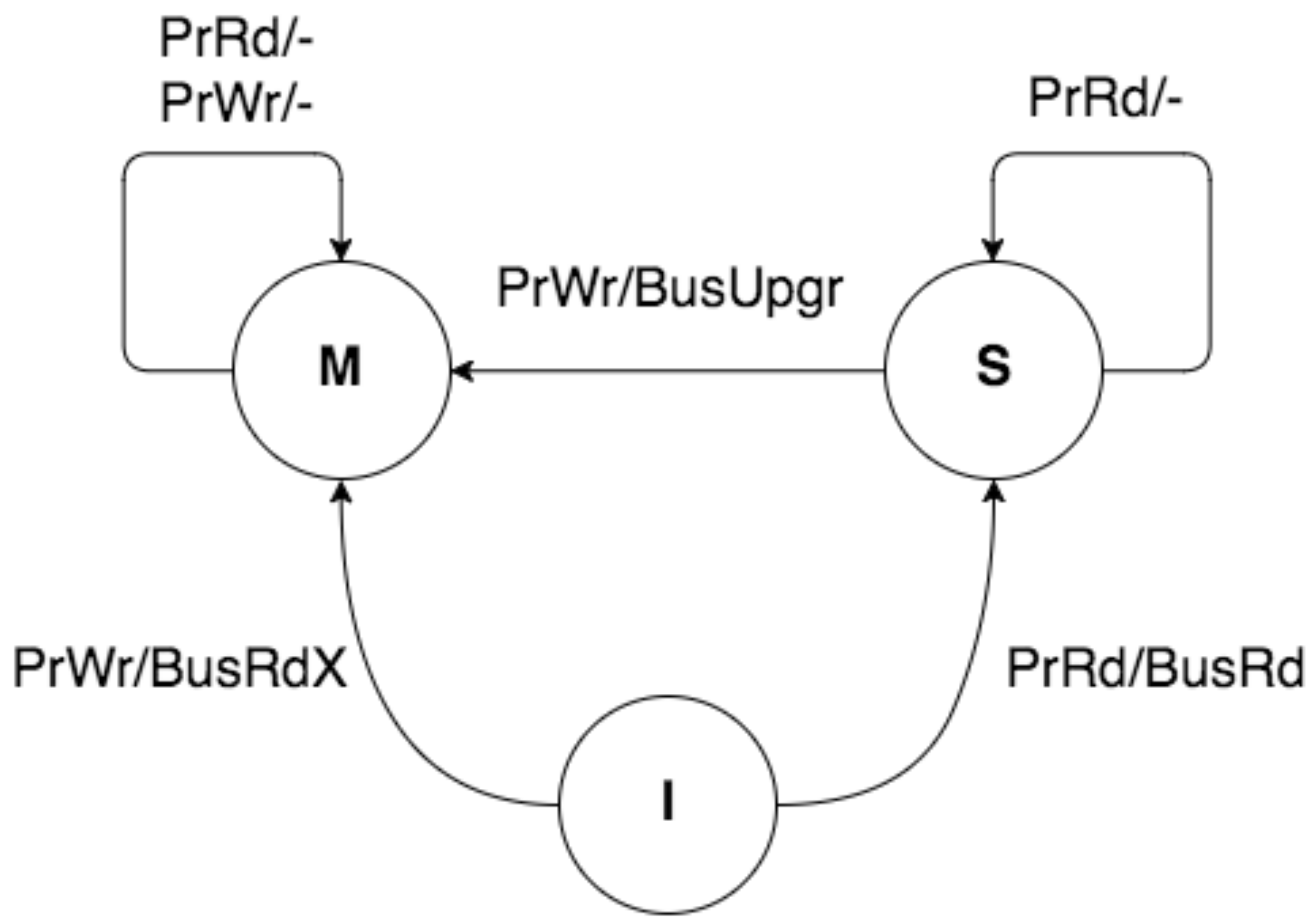
Snooping Coherence (from CPU)

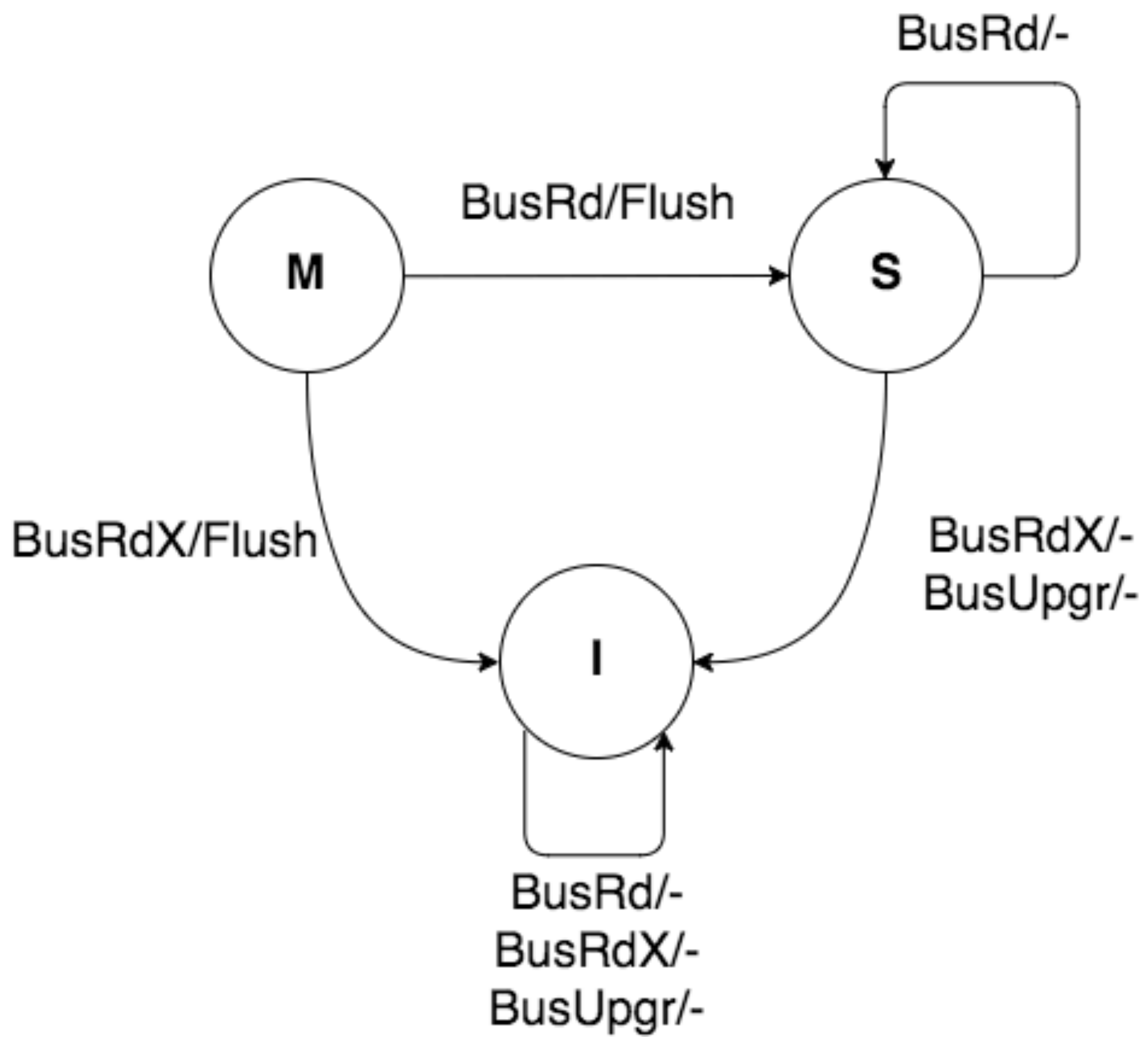
	modified	shared	invalid
read hit	read data in local cache	read data in local cache	n/a
read miss	place read miss on bus; replace data; writeback on bus	place read miss on bus; replace data	place read miss on bus
write hit	write data in local cache	place invalidate on bus; transition to modified state	n/a
write miss	writeback block; place write miss on bus	place write miss on bus	place write miss on bus

Snooping Coherence (from bus)

	modified	shared	invalid
read miss (snoop)	read data and place on bus; writeback data; change to shared state	allow shared cache or memory to service miss	n/a
invalidate	n/a	change to invalid state	n/a
write miss (snoop)	abort other memory operation; writeback data; change to invalid state	change to invalid state	n/a

Snooping Coherence





https://en.wikipedia.org/wiki/MSI_protocol#State_Machine

Snooping Coherence

- Protocol called MSI (Modified, Shared, Invalid)
- Less bus traffic in MESI (Modified, Exclusive, Shared, Invalid) protocol
- Even less bus traffic in MOESI (Modified, Owned, Exclusive, Shared, Invalid) protocol

Chat with your neighbor!

- What are the advantages of snooping?
- What are the disadvantages of snooping?
- How would you describe the consistency model of snooping coherence?

Coherence, Synchronization, Consistency

- We have a protocol to maintain multiple copies of data coherently
- We have described how to use coherence to implement write serialization
- We do *not* have a way for the processor to use the memory hierarchy to implement synchronization features (fences, atomic ops, etc.)
- We do *not* have any other way to reason about consistency in the memory hierarchy

Synchronization: Fences

```
sw x0, 0(x10) # dcache miss, l2 miss, l3 hit

sw x0, 0(x11) # dcache hit

sw x0, 0(x12) # dcache miss, l2 hit
```

If we use coherence for consistency, in what order are these operations going to appear in memory?

Synchronization: Fences

```
sw x0, 0(x10) # cache miss, l2 miss, l3 hit

sw x0, 0(x11) # dcache hit

FENCE

sw x0, 0(x12) # dcache miss, l2 hit
```

Ensures that all memory operations before the fence happen before the memory operations after the fence

Synchronization: Fences

0
store buffer
0x100000
0x200000
0x300000
0x400000
FENCE
memory hierarchy
0
0x100000
0x200000
0x400000
1
0x100000
0x300000
0x400000
bus

Synchronization

- We can leverage protocols in the memory hierarchy to expose high-level programming semantics to the application
    - Fences
    - Conditional operations
    - Atomic operations
    - Increment operations
    - etc…

Synchronization: Fences (Summary)

- Memory operations can appear out of order in the cache hierarchy due to the cache state (e.g., miss then hit, etc.)
- Fences are instructions (software) that provide order to memory operations (hardware)
- We can use coherence with the store buffer to implement fences! Not how it’s done in gem5…

Memory Consistency

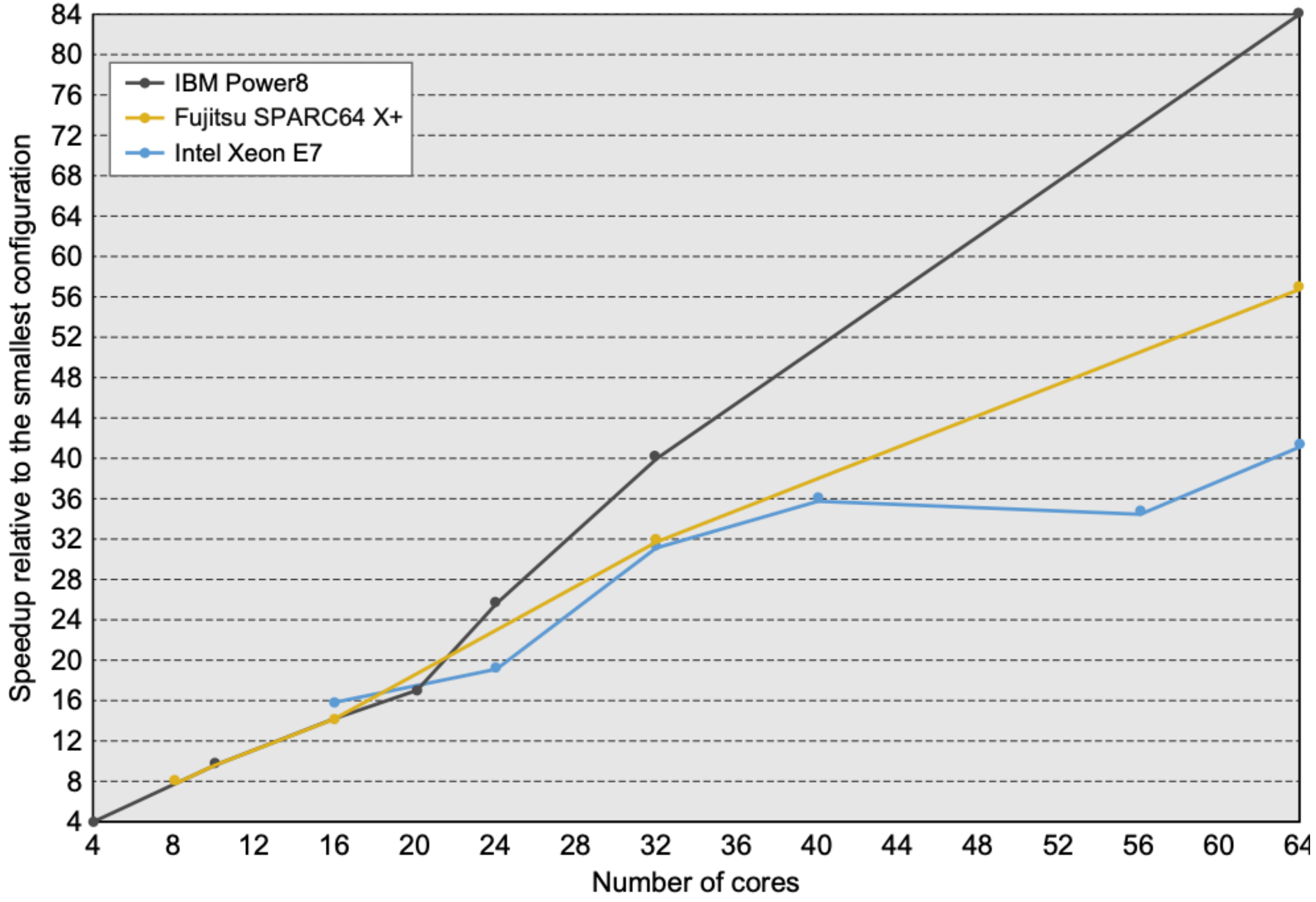
- Programs may not be written with the most efficient memory ordering…
- What if the hardware could just finish operations whenever they were ready?
- Software could do all coordination (i.e., fences everywhere its needed, etc.)

- May this is a bit extreme…
- Relax consistency defines different ways in which hardware will re-order operations!
- Software developers will write architecture specific applications based on consistency models

Memory Consistency

- Operations: R->W, R->R, W->R, W->W
- What if we allow some of these orderings to occur out of order?
- Total store order (TSO): W->R may appear as R->W
- Partial store order (PSO): TSO and W1->W2 may appear W2->W1
- Weak ordering: all operations may appear in any order

Memory Consistency



Summary

- We covered the cache hierarchy for multiprocessors
- We defined coherence
- We described snooping, and built a snooping-based coherence protocol
- We used fences as a case study for how the processor and memory hierarchy allows software to implement synchronization
- We defined different memory consistency models, which allow for varying degrees of reordering