Lecture 17:
A More Sophisticated Snooping Multi-Processor

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

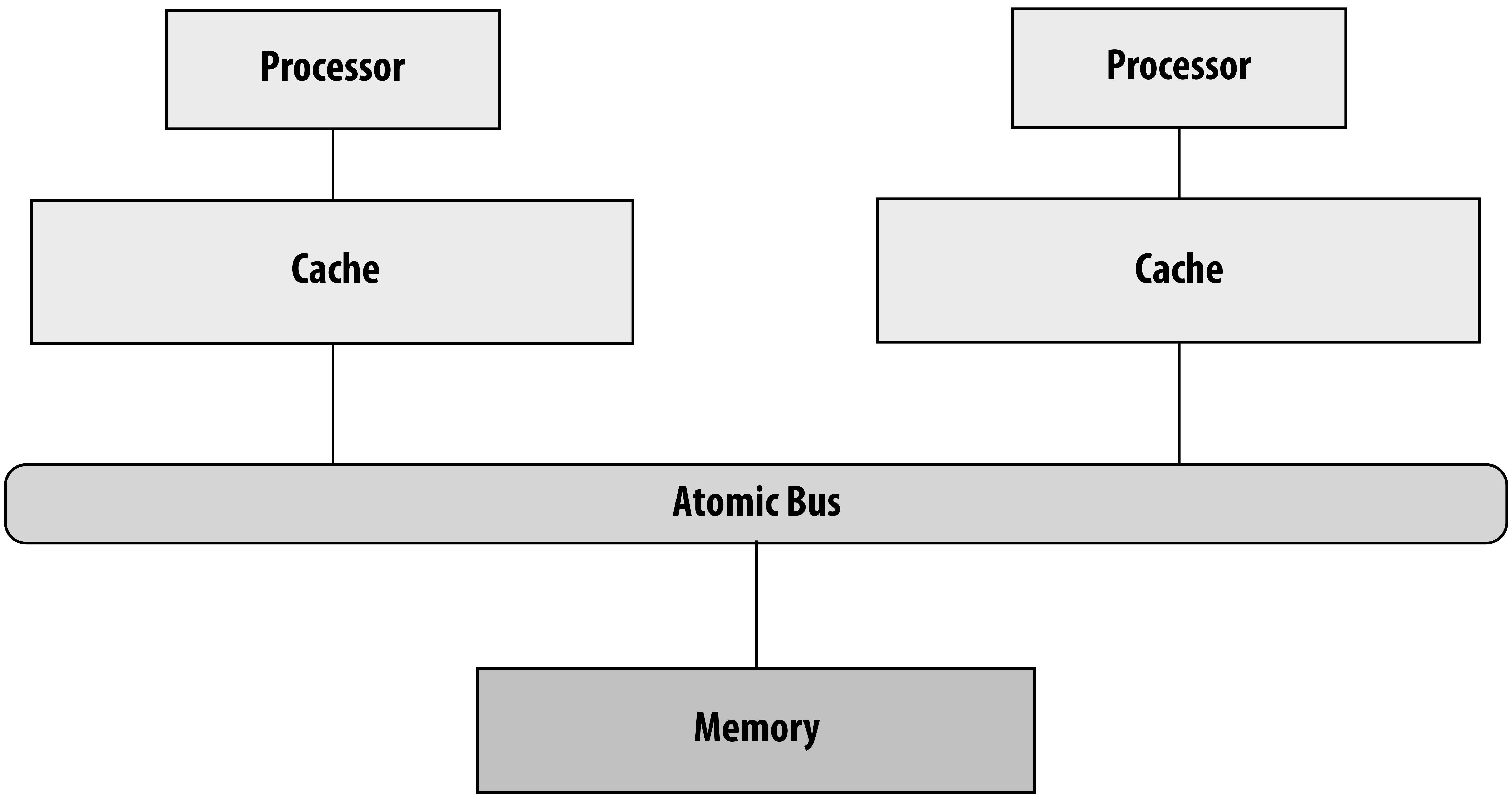
Announcements

- Michael will be giving lecture next class
  - Interconnection networks

- Gentle reminder to come talk to us about project ideas

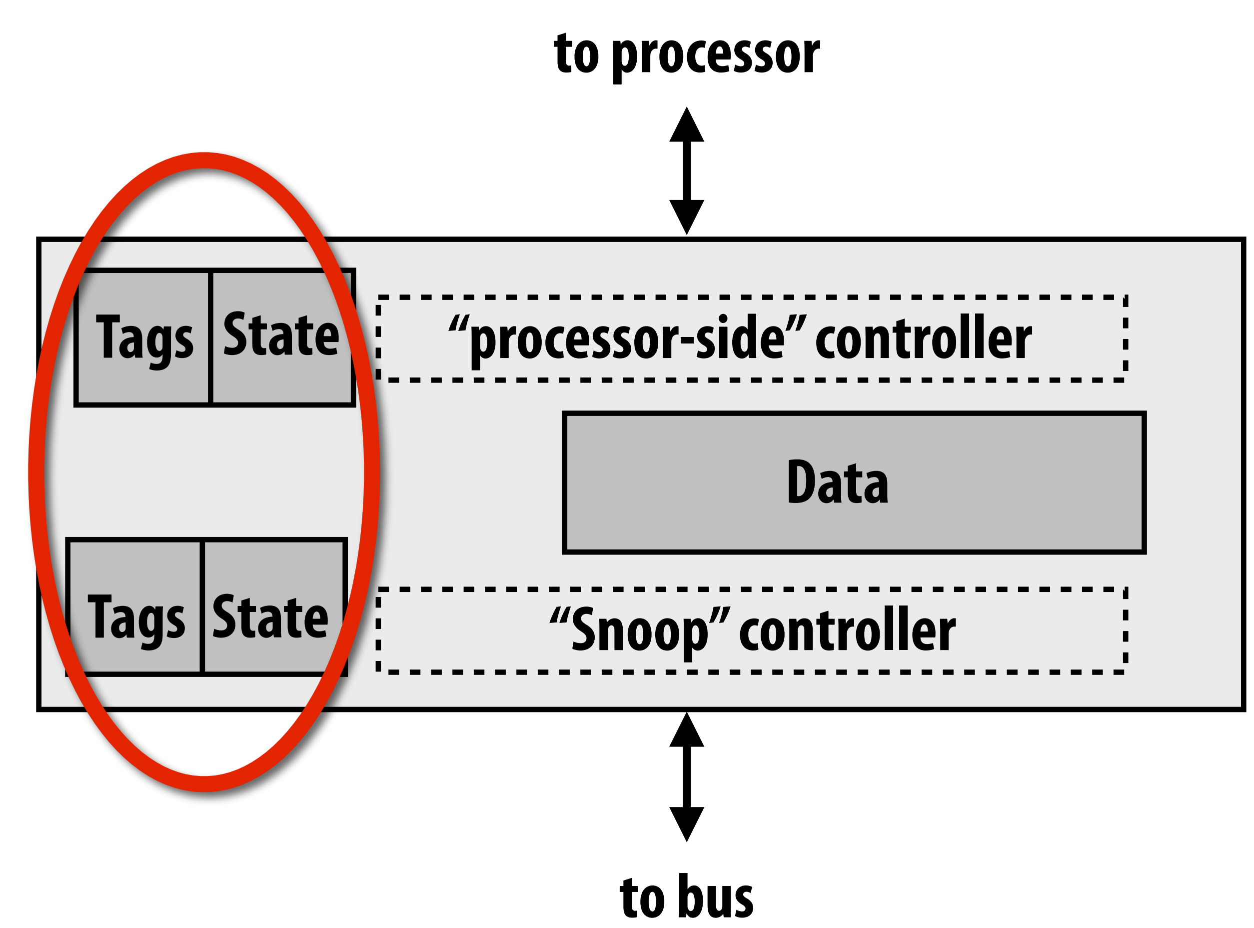
Last time

- We implemented a very simple cache-coherent multi-processor around a shared atomic bus



Key issues

We addressed the issue of contention for access to tags by duplicating tags.

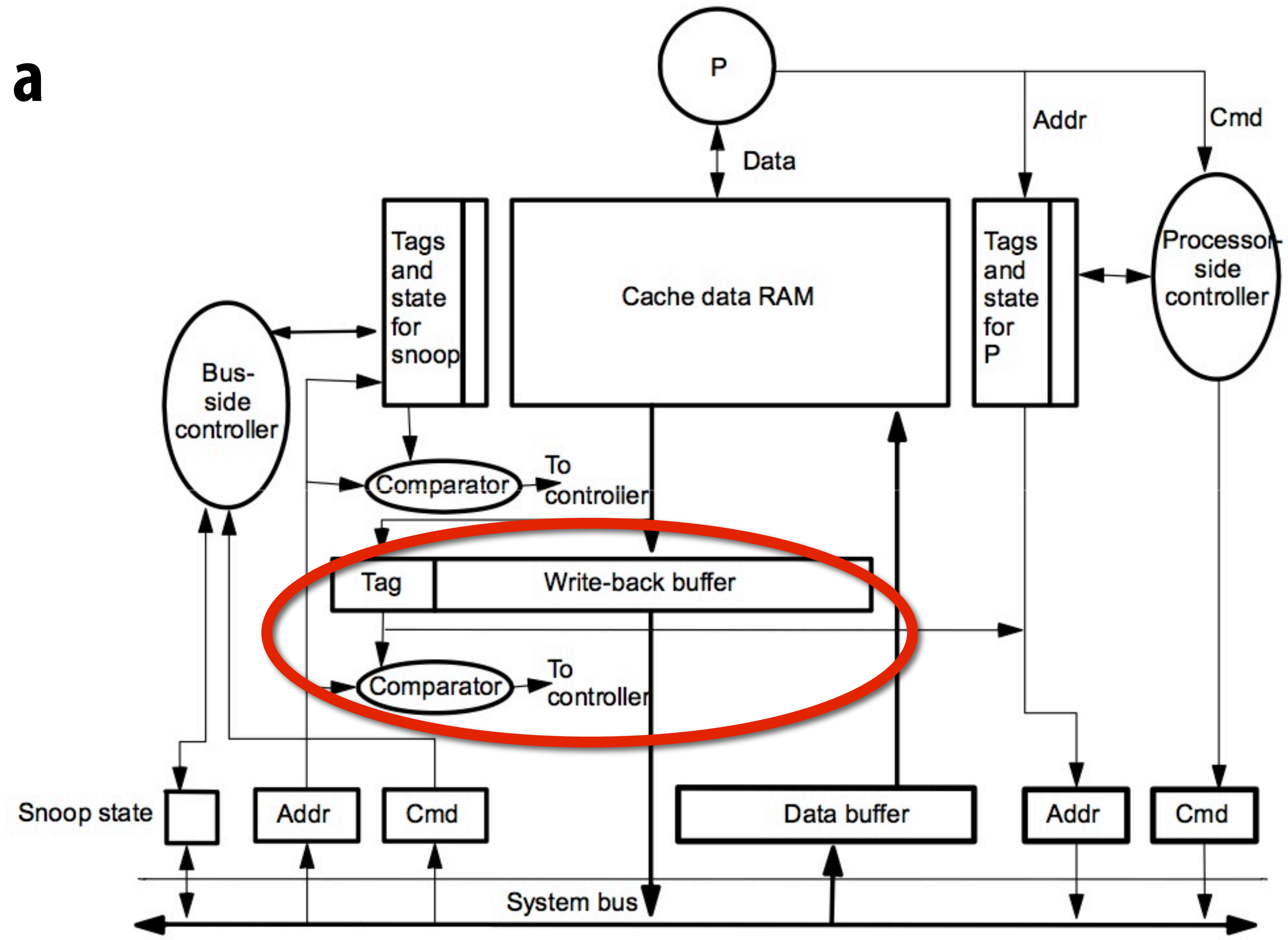


We described how snoop results are collectively reported to a cache via shared, dirty, and valid lines on the bus.

Key issues

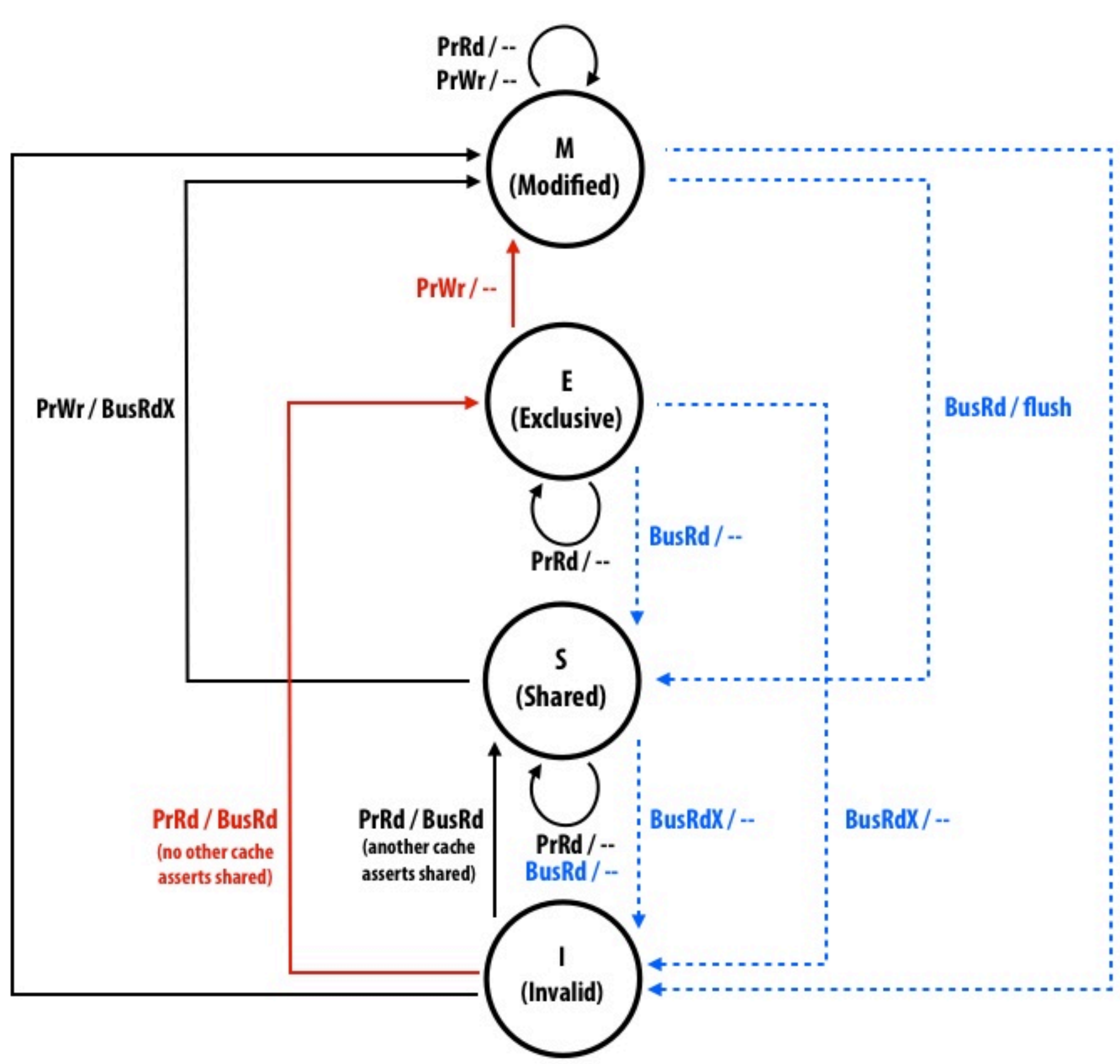
We addressed correctness issues due to a write-back buffer by checking both the cache tags and the write-back buffer when snooping.

(and also added the ability to cancel pending bus transfer requests).



Key issues

We talk about ensuring write serialization: processor is held up by the cache until the "make exclusive" transaction appears on the bus. (write commits)



We talked about how coherence protocol state transitions are not atomic machine operations (even though the bus itself it atomic) leading to possible race conditions.

We discussed deadlock, livelock, and starvation

Situation 1:

P1 has a modified copy of cache block B

P1 is waiting for the bus to issue BusRdX on cache block A

BusRd for B appears on bus while P1 is waiting

FETCH DEADLOCK!

To avoid deadlock, P1 must be able to service incoming transactions while waiting to issue its own requests

Correctness?

Situation 2:

Two processors simultaneously write to cache block B

P1 acquires bus ("wins bus"), issues BusRdX

P2 invalidates in response to P1's BusRdX

Before P1 performs the write (updates block), P2 acquires bus and issues BusRdX

P1 invalidates in response to P2's BusRdX

LIVELOCK!

To avoid livelock, write that obtains exclusive ownership must be allowed to complete before exclusive ownership is relinquished.

Starvation

- **Multiple processors competing for bus access**
  - Must be careful to avoid (or minimize likelihood of) starvation
- **FIFO arbitration**
  - Eliminates starvation
- **Priority-based heuristics**
  - Reduce likelihood of starvation

Source of the complexity: parallelism

- **Processor, cache, bus, memory all are resources operating in parallel**
  - **Often contending for shared resources:**
    - **Processor and bus contending for cache**
    - **Caches and memory contenting for bus access**

- **"Memory operations" that are <u>abstracted</u> by the architecture as atomic are <u>implemented</u> via multiple transactions involving all of these clients**

- **Performance optimization often entails splitting operations into several smaller transactions**
  - **Splitting work into smaller transactions reveals more parallelism (recall pipelining example)**
  - **Cost: more hardware needed to exploit additional parallelism**
  - **Cost: more care needed to ensure abstractions still hold (the machine is correct)**

Today's topic

More of the same...

but now we will build the system around a non-atomic bus.

Optimize

Re-evaluate correctness

Optimize

Re-evaluate correctness

[and so on...]

What you should know

- How deadlock and livelock might occur in both atomic bus and non-atomic bus-based systems (what are possible solutions for avoiding it?)
- Why is an atomic bus likely insufficient for our needs
- The main components of a split-transaction bus, how transactions are split into requests and responses
- The role of queues in a parallel system

Transaction on an atomic bus

1. Client is granted bus access (result of arbitration)
2. Client places command on bus (may also place data on bus)

Problem: bus is idle while response is pending (decreases effective bus bandwidth)

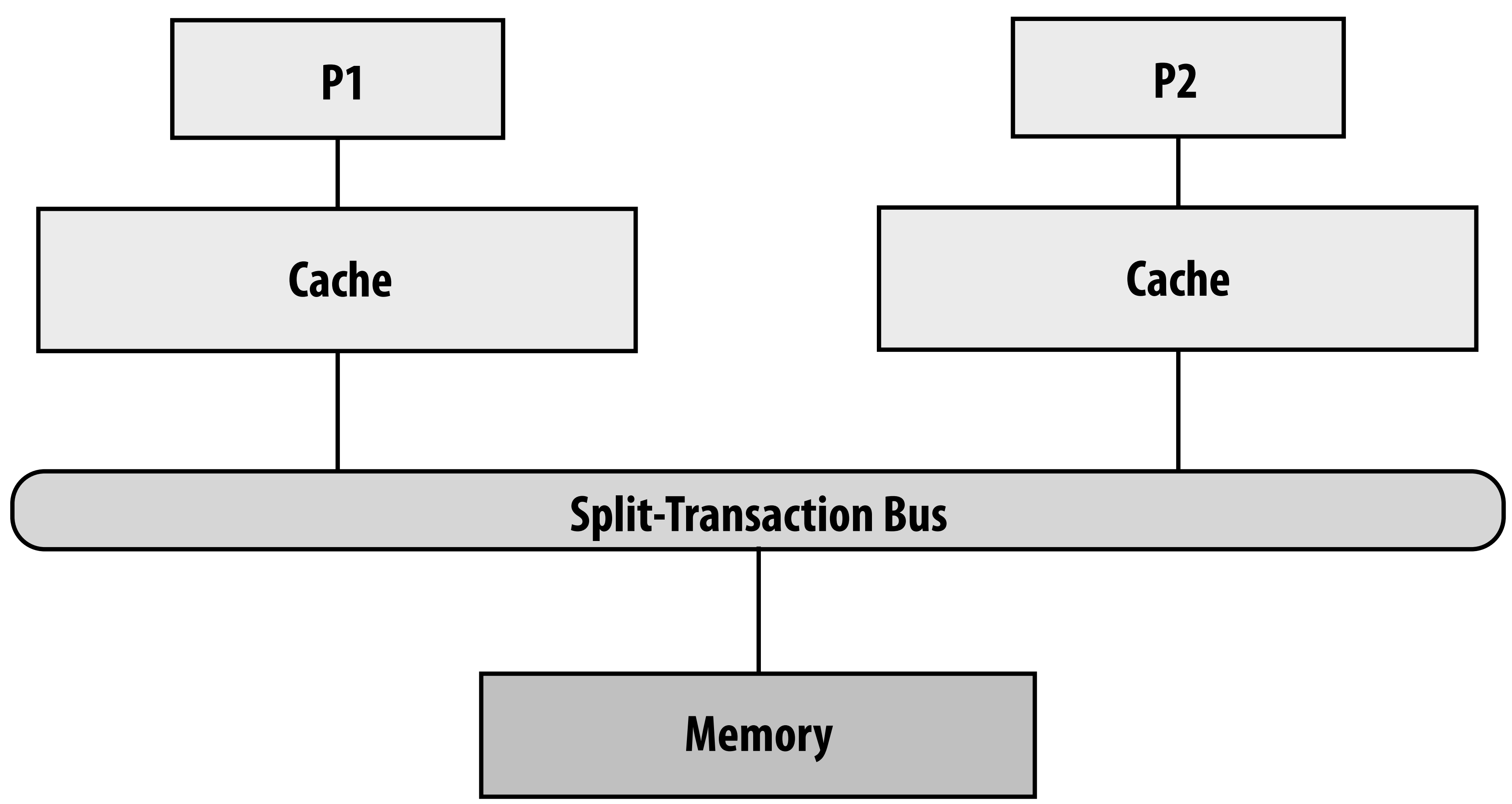
This is bad, because the bus is often a limited, shared resource in a multi-processor system.

3. Response by another bus client placed on bus
4. Next client obtains bus access (arbitration)

Split-transaction bus

Bus transactions are split into two separate request and response sub-transactions.

Other transactions can intervene.



Consider:
Read miss to A by P1
Bus upgrade of B by P2

P1 gains access to bus

P1 sends BusRd command
[memory starts fetching data]

P2 gains access to bus

P2 sends BusUpg command

Memory gains access to bus

Memory places A on bus

New issues

1. How to match requests with responses?

2. Conflicting requests on bus

   Consider:

   - P1 has outstanding request for block A
   - Before response to P1 occurs, P2 makes request for block A

3. Flow control: how many requests can be outstanding at a time, and what should be done when buffers fill up?

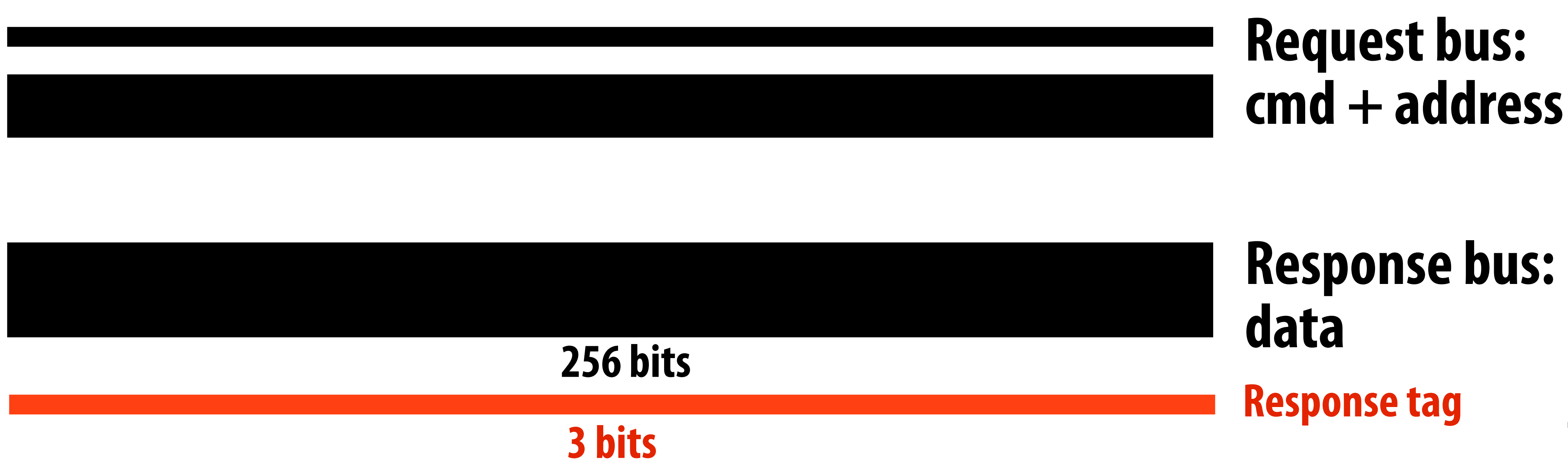
4. When are snoop results reported? During the request? During the response?

A basic design
(follows design discussed in textbook section 6.4)

- **Up to eight outstanding requests at a time**
- **Responses need not be in the same order as requests**
    - **But request order establishes the total order for the system**
- **Flow control via negative acknowledgements (NACKs)**
    - **When a buffer is full, client can NACK a transaction, causing a retry**

Initiating a request

Can think of a split-transaction bus as two separate buses: a request bus and a response bus.



Request Table
(copy maintained by each cache)

Transaction tag is index into table →

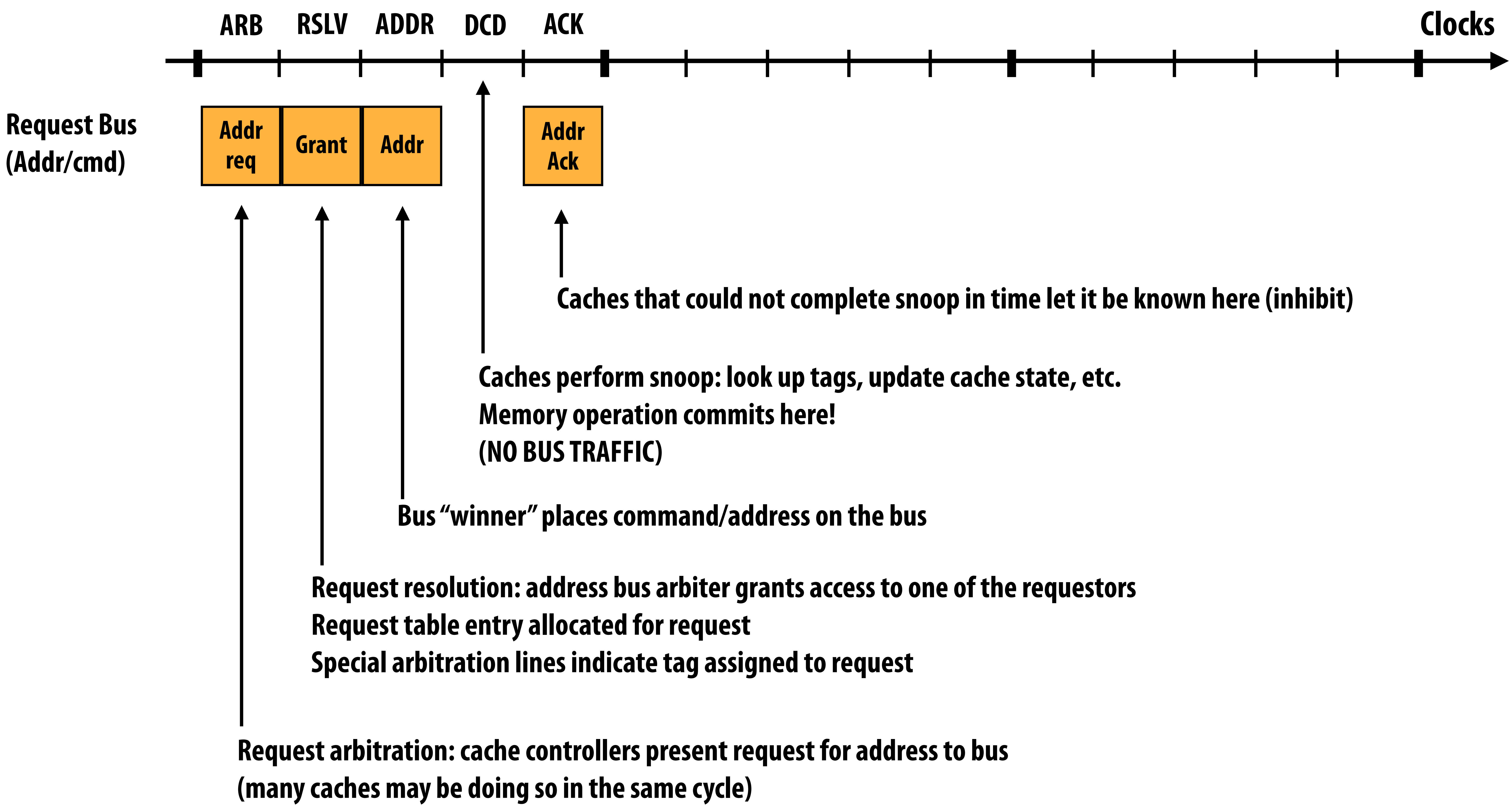
Requestor	Addr	State
P0	0xbeef	

Step 1: Requestor asks for request bus access

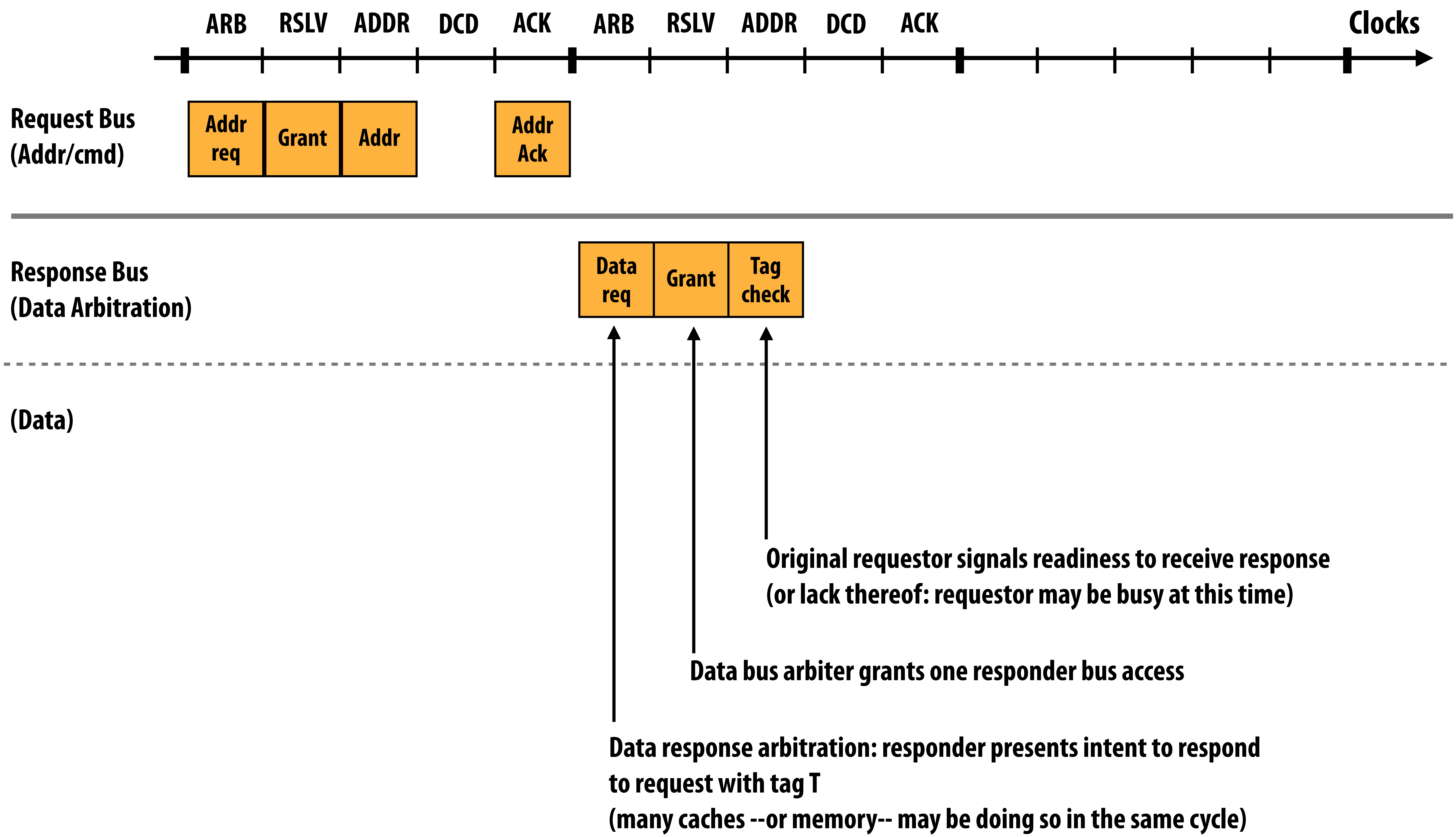
Step 2: Bus arbiter grants access, assigns transaction a tag

Step 3: Requestor places command + address on the request bus

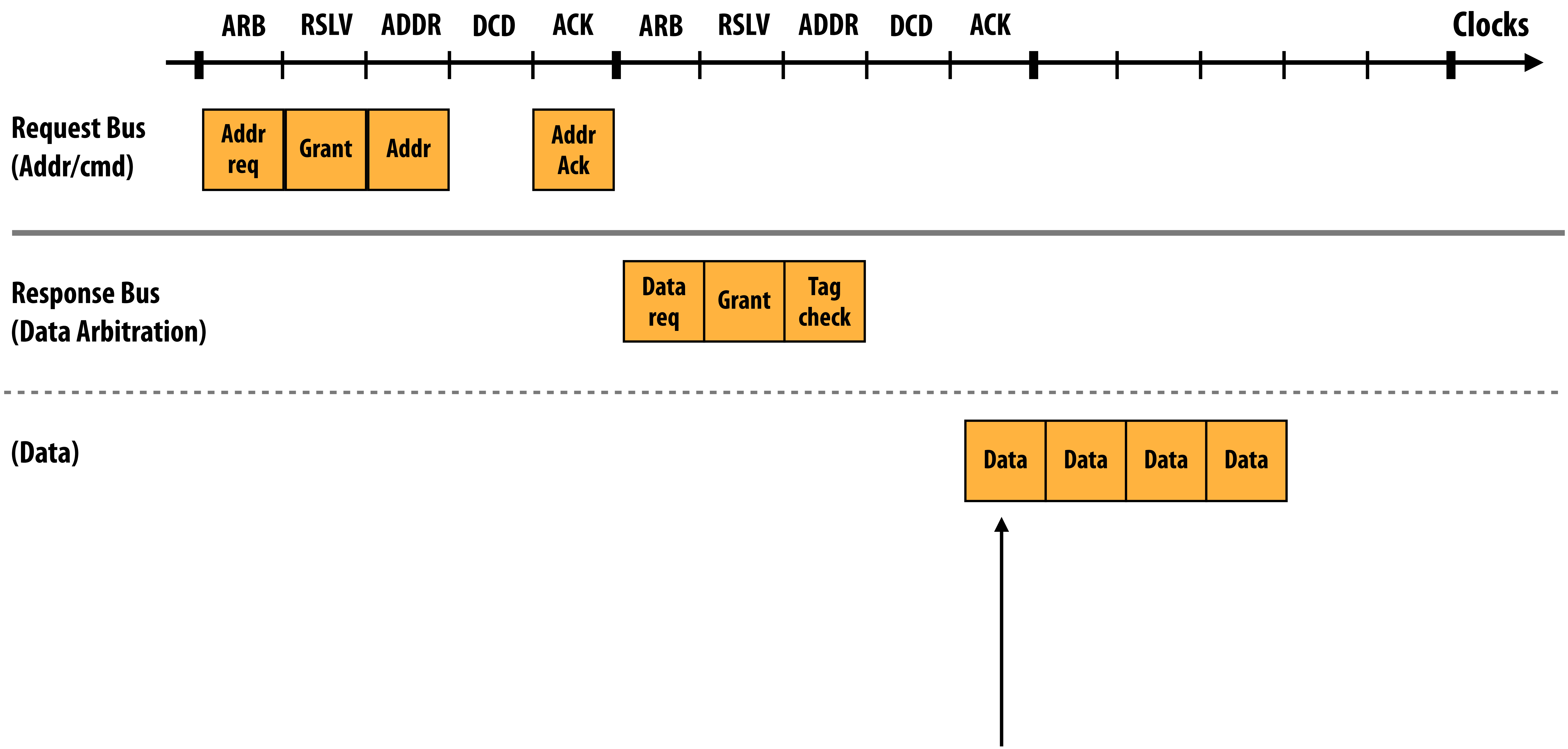
Read miss: cycle-by-cycle bus behavior (phase 1)



Read miss: cycle-by-cycle bus behavior (phase 2)



Read miss: cycle-by-cycle bus behavior (phase 3)



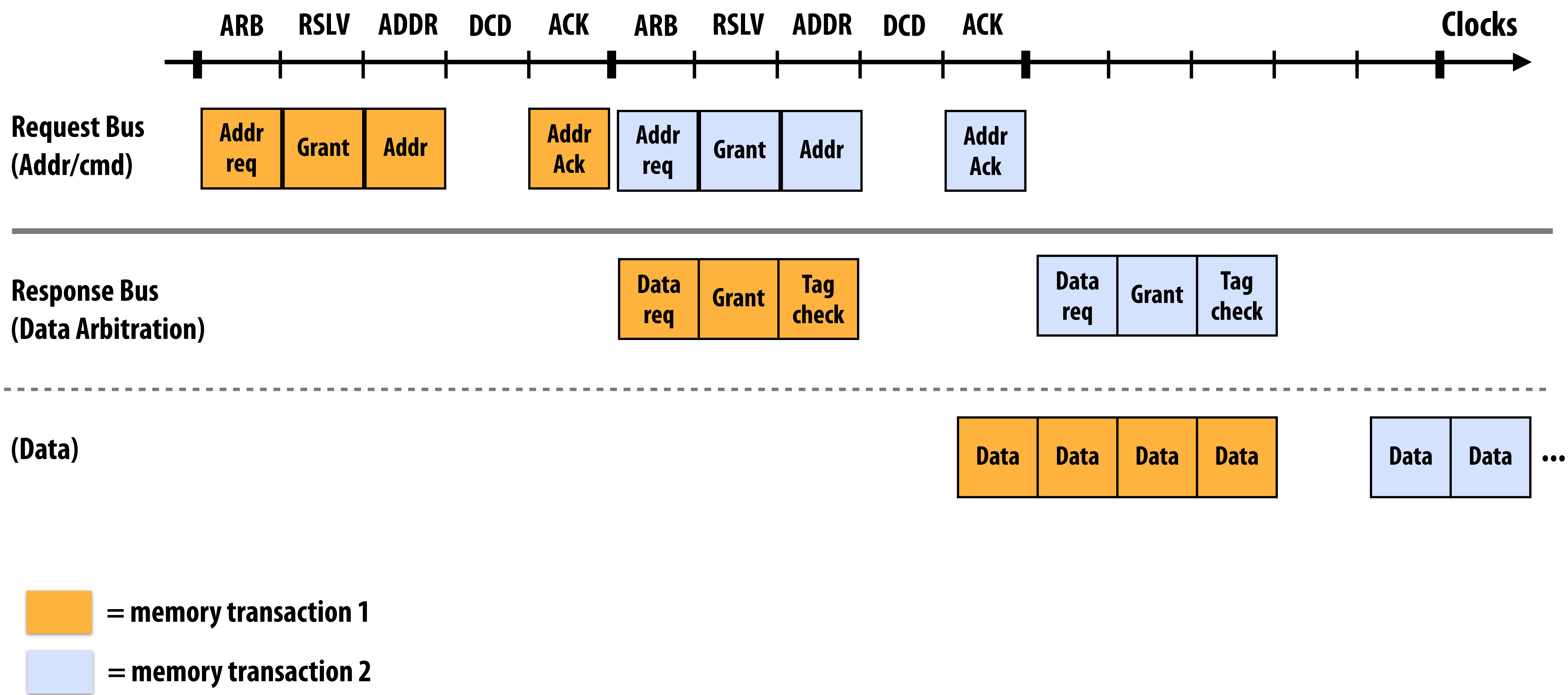
Responder places response data on data bus

Caches present snoop result for request with the data

Request table entry is freed

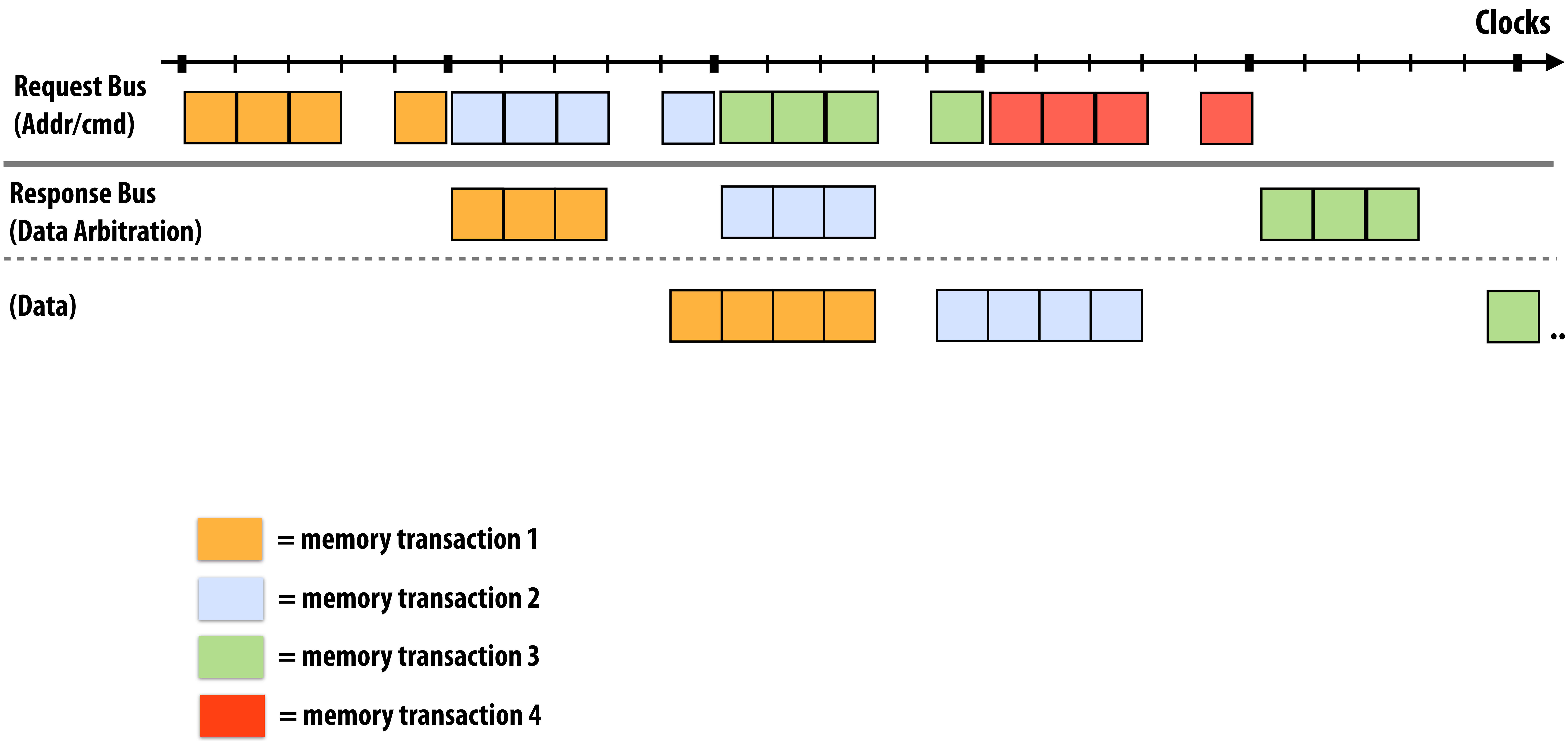
Here: assume 128 byte cache lines → 4 cycles on 256 bit bus

Pipelined transactions



Note: write-backs and BusUpg transactions do not have a response component (write backs acquire access to both request address and data bus as part of the request)

Pipelined transactions



Dealing with key issues

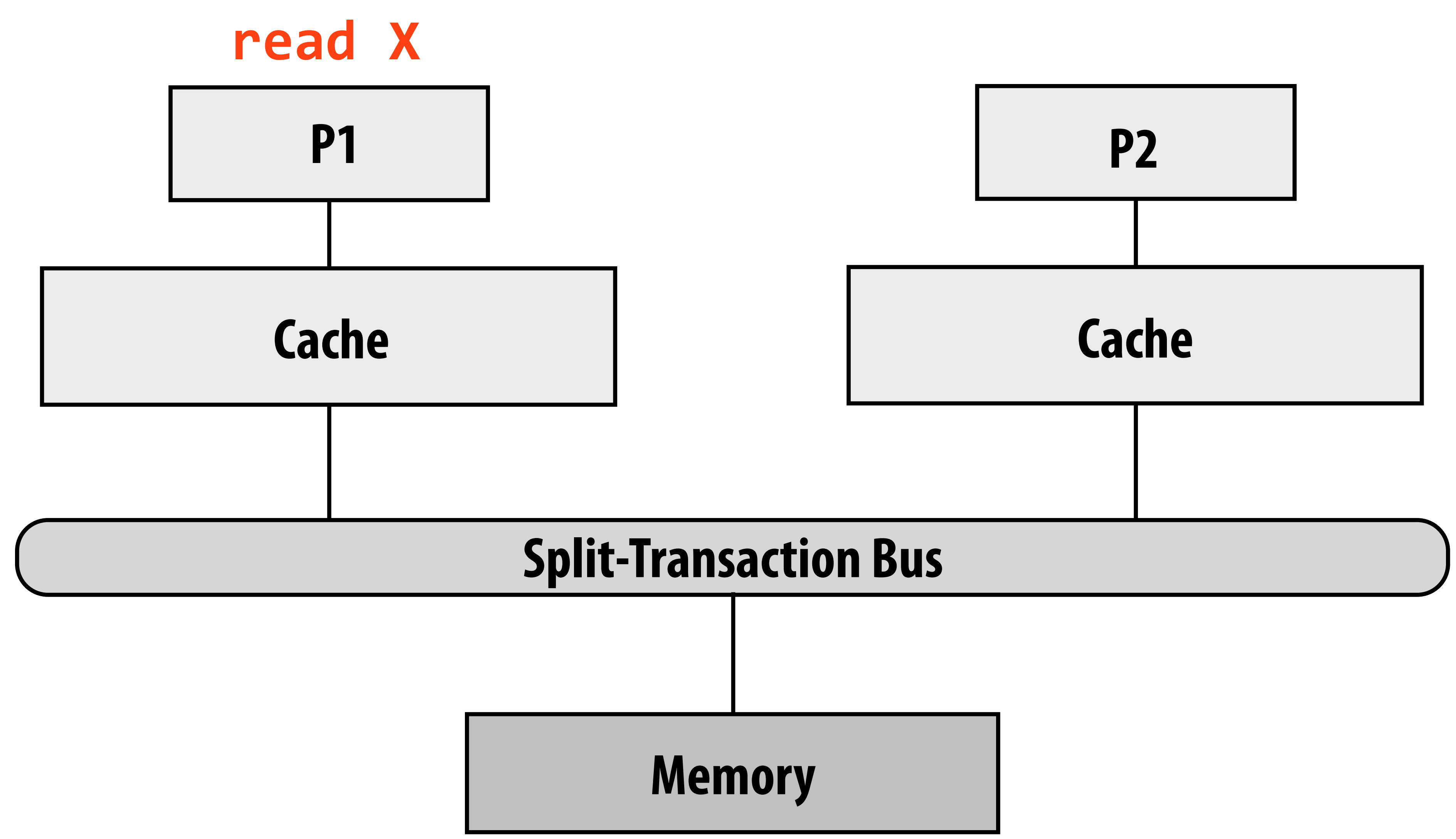
- **Conflicting requests**
  - Avoid conflicting requests by disallowing them
  - Each cache has a copy of the request table
  - Policy: caches do not make requests that conflict with requests in the request table

- **Flow control:**
  - Caches/memory have buffers for receiving data off the bus
  - If the buffer fills, client NACKs relevant requests or responses
  - Triggers a later retry

Situation 1: P1 read miss to X, transaction involving X is outstanding on bus

P1 Request Table

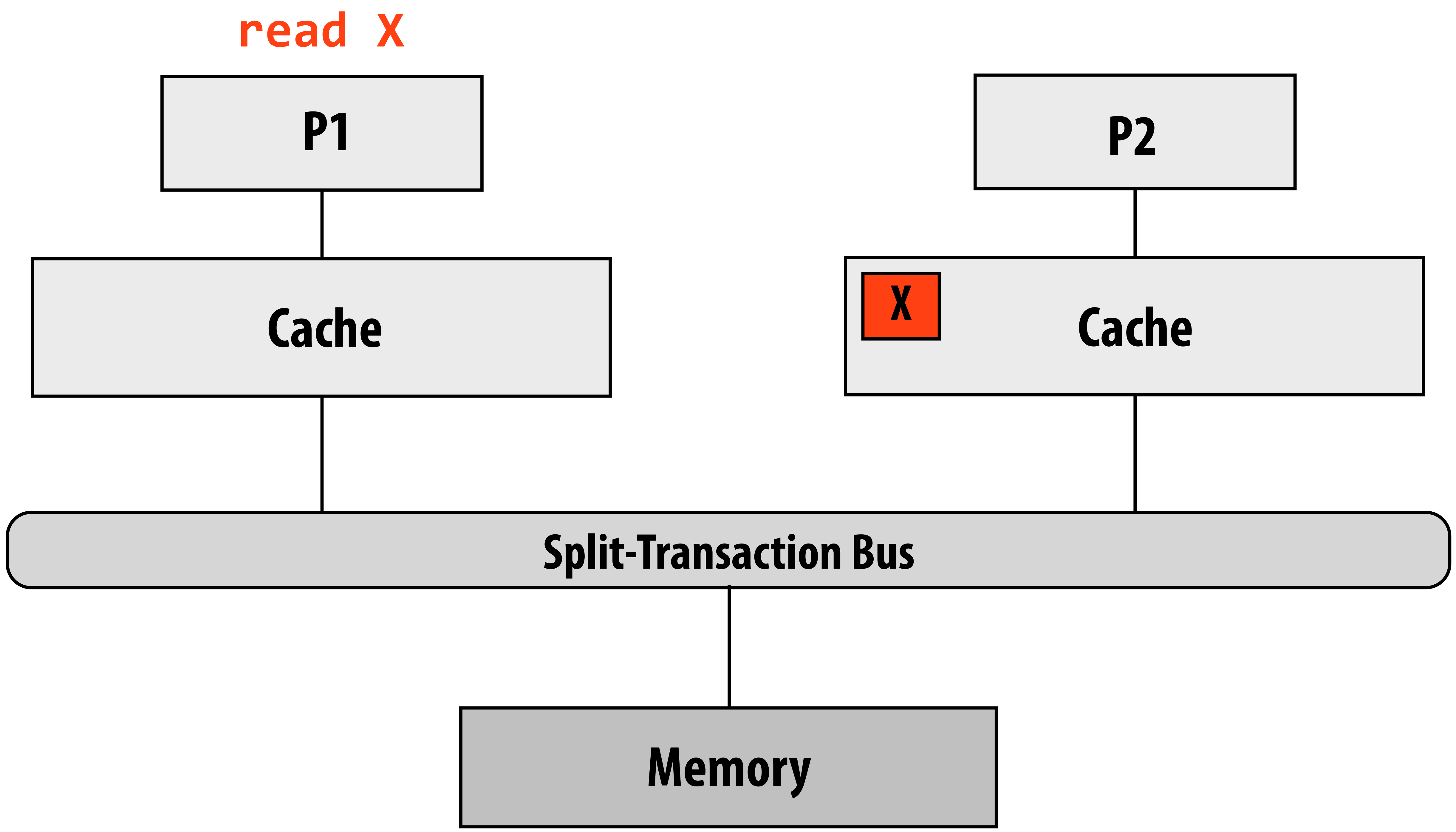
Requestor	Addr	State
P2	X	Op: BusRdX , share



If there is a conflicting outstanding request (as determined by checking the request table), cache must hold request until conflict clears

If outstanding request is a read: not a conflict. No need to make new request, just listen for the response to the previous one.

Situation 2: P1 read miss to X, X dirty in P2's cache

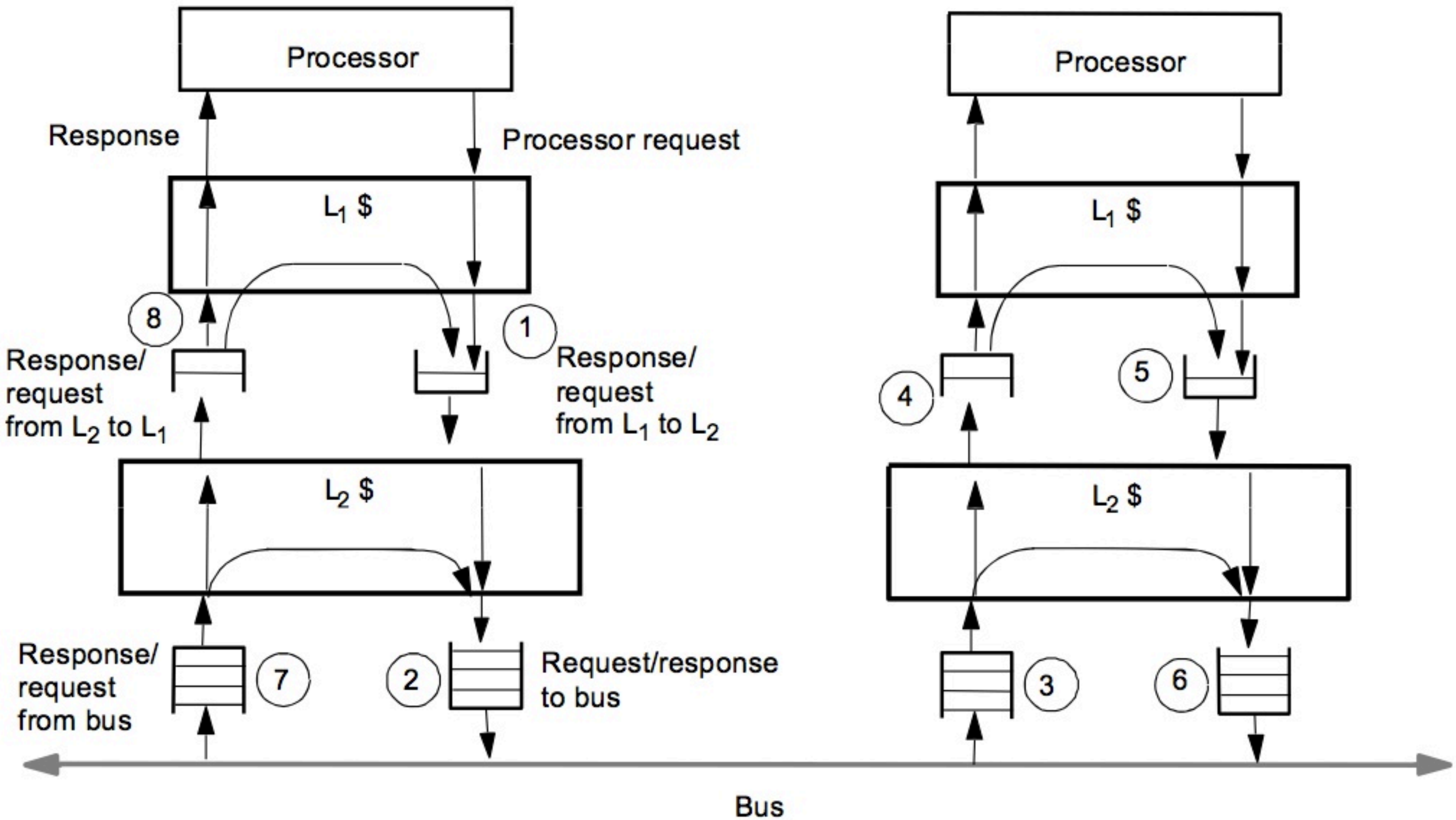


P1 wins request bus, issues BusRd request on bus

Caches begin snooping, memory may begin fetch

What happens next?

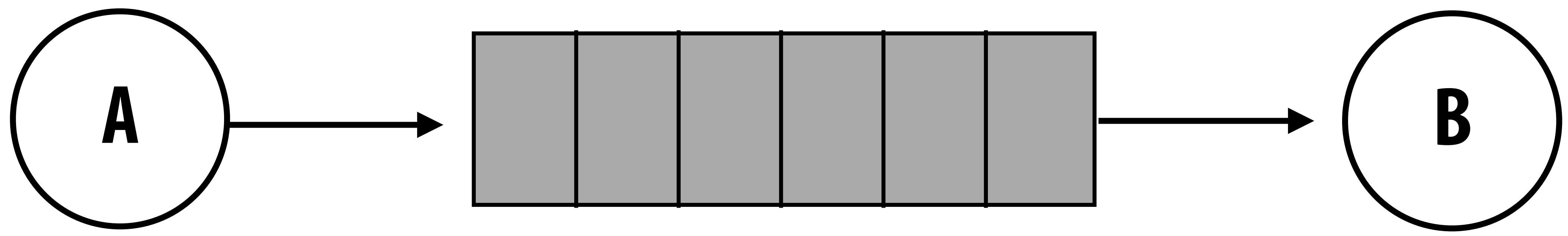
Multi-level cache hierarchies



Assume one outstanding memory request per processor.

Consider fetch deadlock problem: cache must be able to service requests while waiting on response to its own request (hierarchies increase response delay)

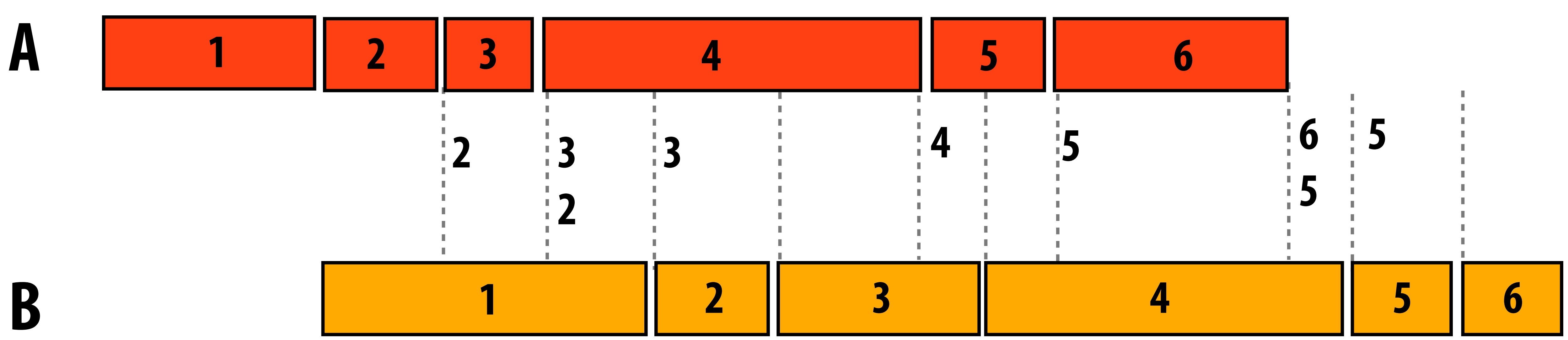
Aside: why do we have queues?



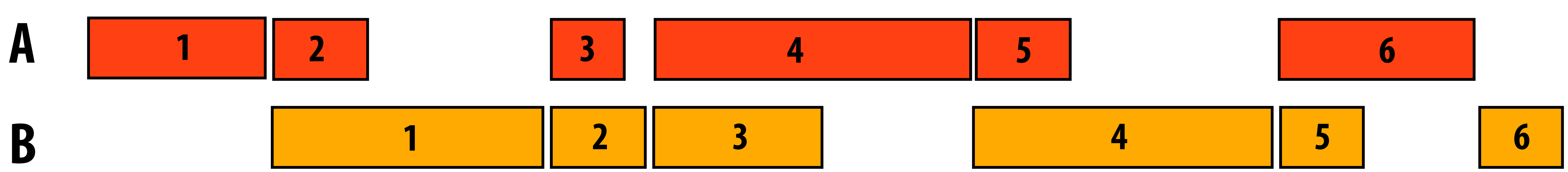
To accommodate variable (unpredictable) rates of production and consumption.

As long as A and B, on average, produce and consume at the same rate, both workers can run at full rate.

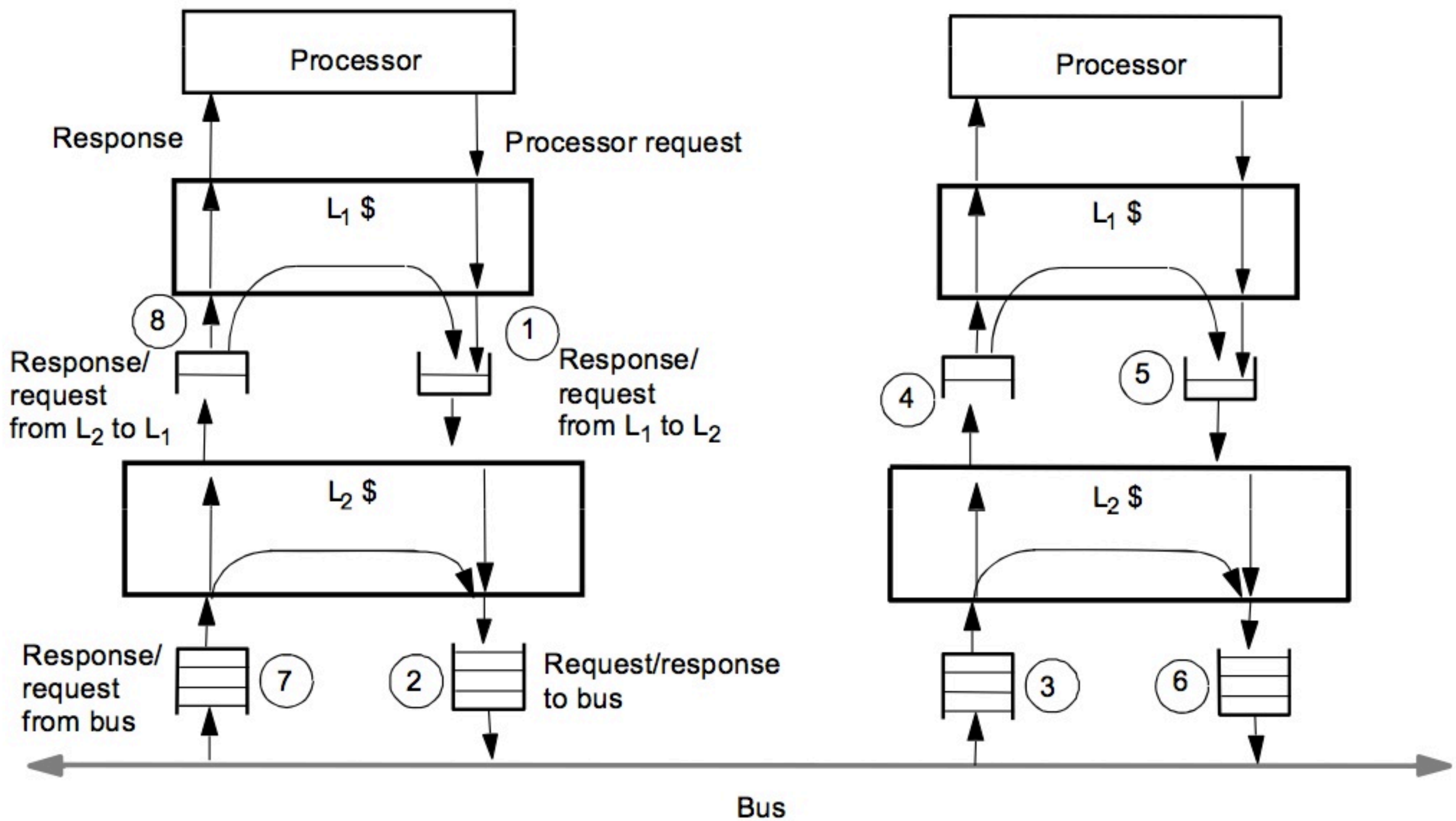
With queue of size 2: A and B never stall



No queue: stalls exist



Multi-level cache hierarchies

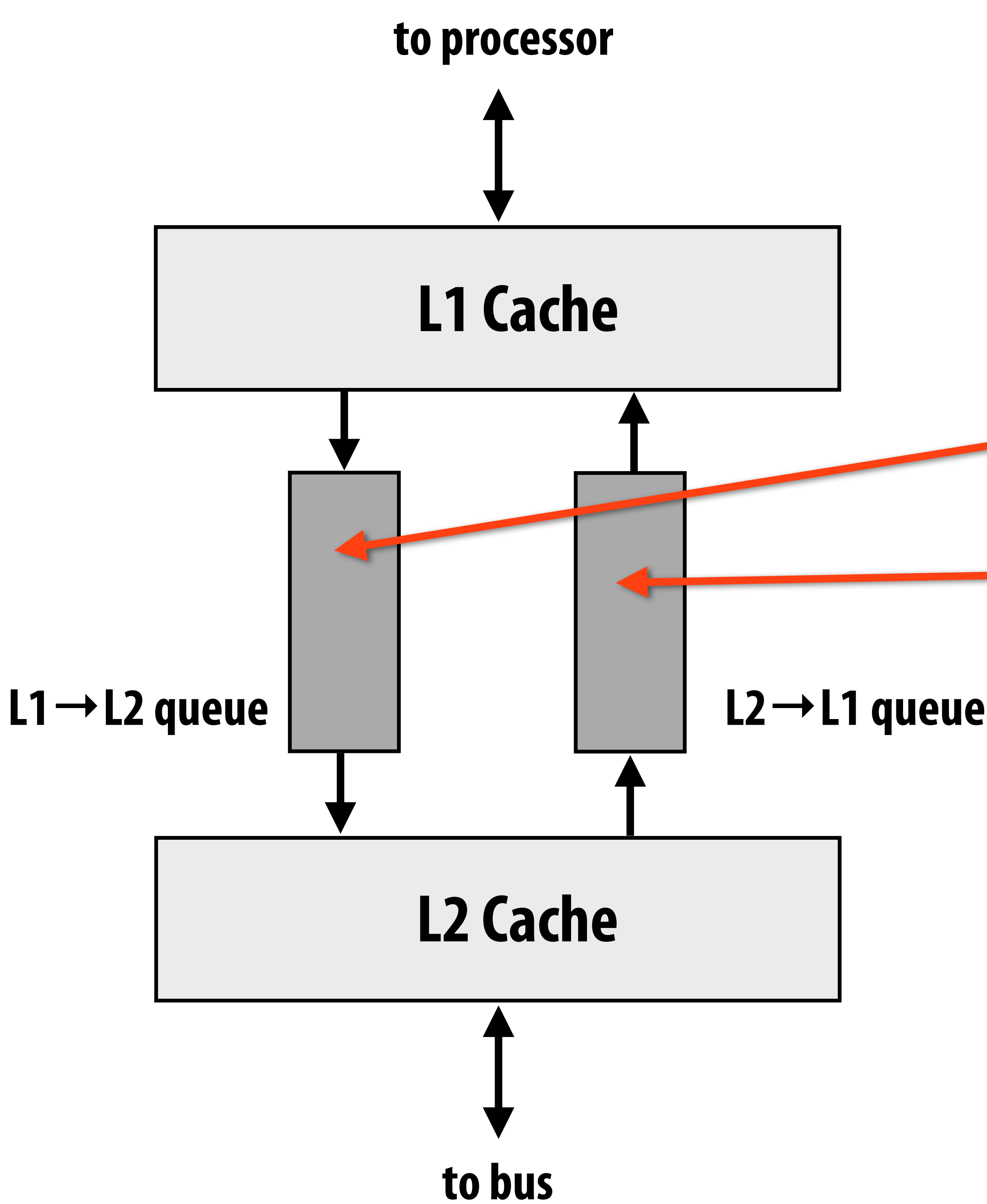


Assume one outstanding memory request per processor.

Consider fetch deadlock problem: cache must be able to service requests while waiting on response to its own request (hierarchies increase response delay)

Ideally, would like buffering at each cache for all requests that can be outstanding on bus.

Buffer deadlock



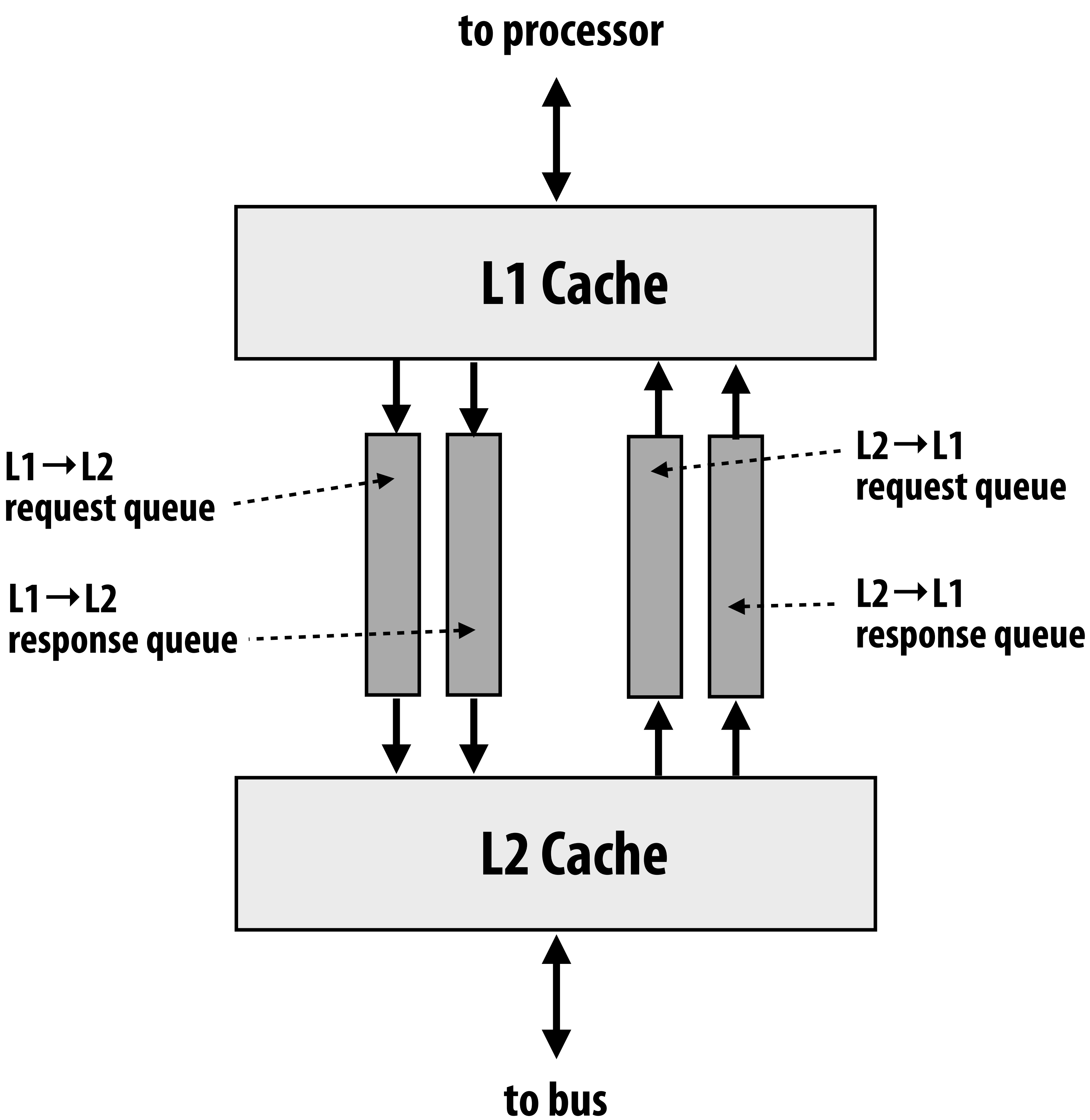
Outgoing read request (initiated by this processor)

Incoming read request (due to another cache) **

Both requests generate responses that require space in the other queue (circular dependency)

** will only occur if L1 is write back

Avoiding buffer deadlock



Classify all transactions as requests and responses

Responses can be completed without generating further transactions

While stalled attempting to send a request, cache must be able to service <u>responses</u>.

Responses will make progress (they generate no new work so there's no circular dependence), eventually freeing up resources for requests

** will only occur if L1 is write back

Putting it all together

Class exercise: describe everything that might occur during the execution of this statement

```
int x = 10;     // assume write to memory, not stored in register
```

Class exercise: describe everything that might occur during the execution of this statement

```
int x = 10;
```

Virtual address to physical (TLB lookup)
TLB miss
TLB update (might involve OS)
OS may need to swap in page (load from disk to physical address)
Cache lookup
Line not in cache (need to generate BusRdX)
Arbitrate for bus
Win bus, place address, command on bus
Another cache or memory decides it must respond (assume memory)
Memory request sent to memory controller
Memory controller is itself a scheduler
Memory checks active row. Changes active row into row buffer
Values read from row buffer
Memory arbitrates for data bus
Memory wins bus
Memory puts data on bus
Cache grabs data, updates cache line and tags, moves line into Exclusive state
Processor notified data exists
Instruction proceeds