

**Lecture 16:**

# **A More Sophisticated Snooping-Based Multi-Processor**

---

**Parallel Computer Architecture and Programming**  
**CMU 15-418/15-618, Spring 2014**

# Tunes

## **“The Projects”** **Handsome Boy Modeling School** **(So... How’s your Girl?)**

*“There’s a lot about the projects I do adore.”*

*- Dan the Automator*

# Final projects / expectations

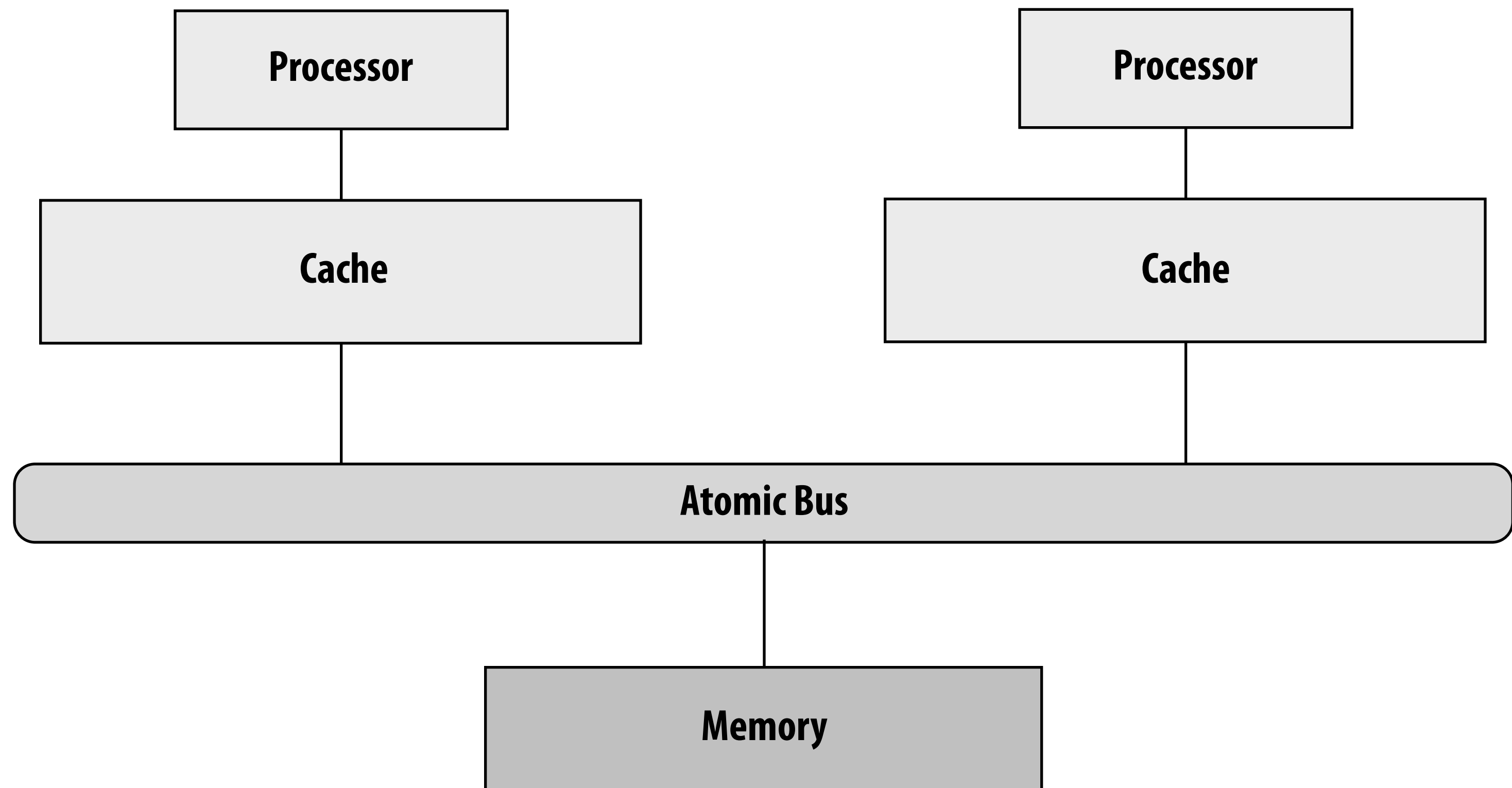
- **Project proposals are due Friday April 4 (but you are welcome to submit early to get feedback)**
  - 5-6 weeks of effort: expecting at least 2 class assignments worth of work
- **The parallelism competition is on Friday May 10th during the final exam slot.**
  - ~20-25 “finalist” projects will present to judges during this time
  - Other projects will be presented to staff later in the day
  - Final writes are due end of day on May 10th. (no late days)
- **Your grade is independent of the parallelism competition results**
  - It is based on your work, your writeup, and your presentation
- **You are encouraged to design your own project**
  - Contact staff as early as possible about equipment needs
  - This is a list of project ideas on the web site to help
  - <http://15418.courses.cs.cmu.edu/spring2014/article/12>

# Photo competition winner: “75% utilization”



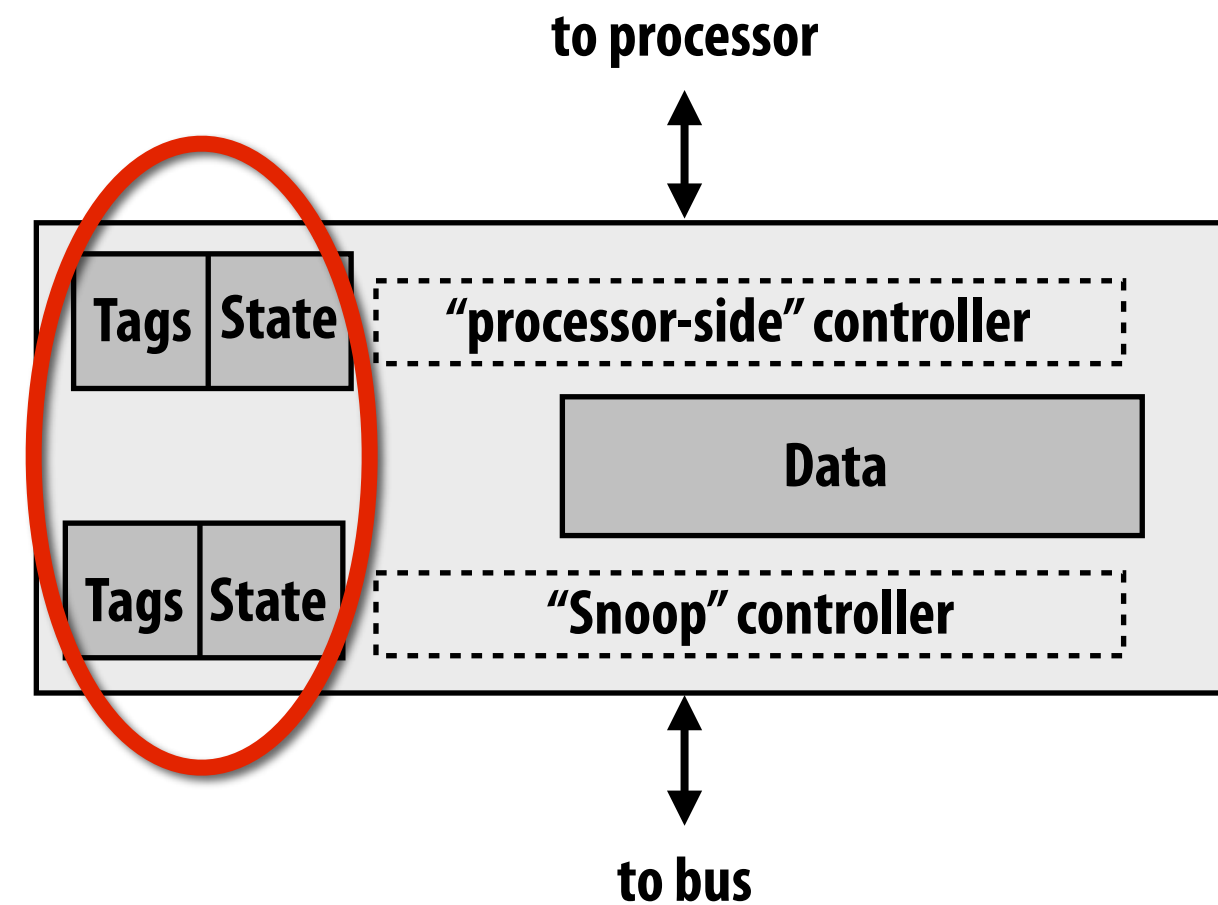
**“My work aims to capture the essential human struggle that pits ensuring good load balance against the overheads incurred by using resources to orchestrate a computation.” - Rokhini**

**Last time we implemented a very simple cache-coherent multi-processor using an atomic bus as an interconnect**



# Key issues

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



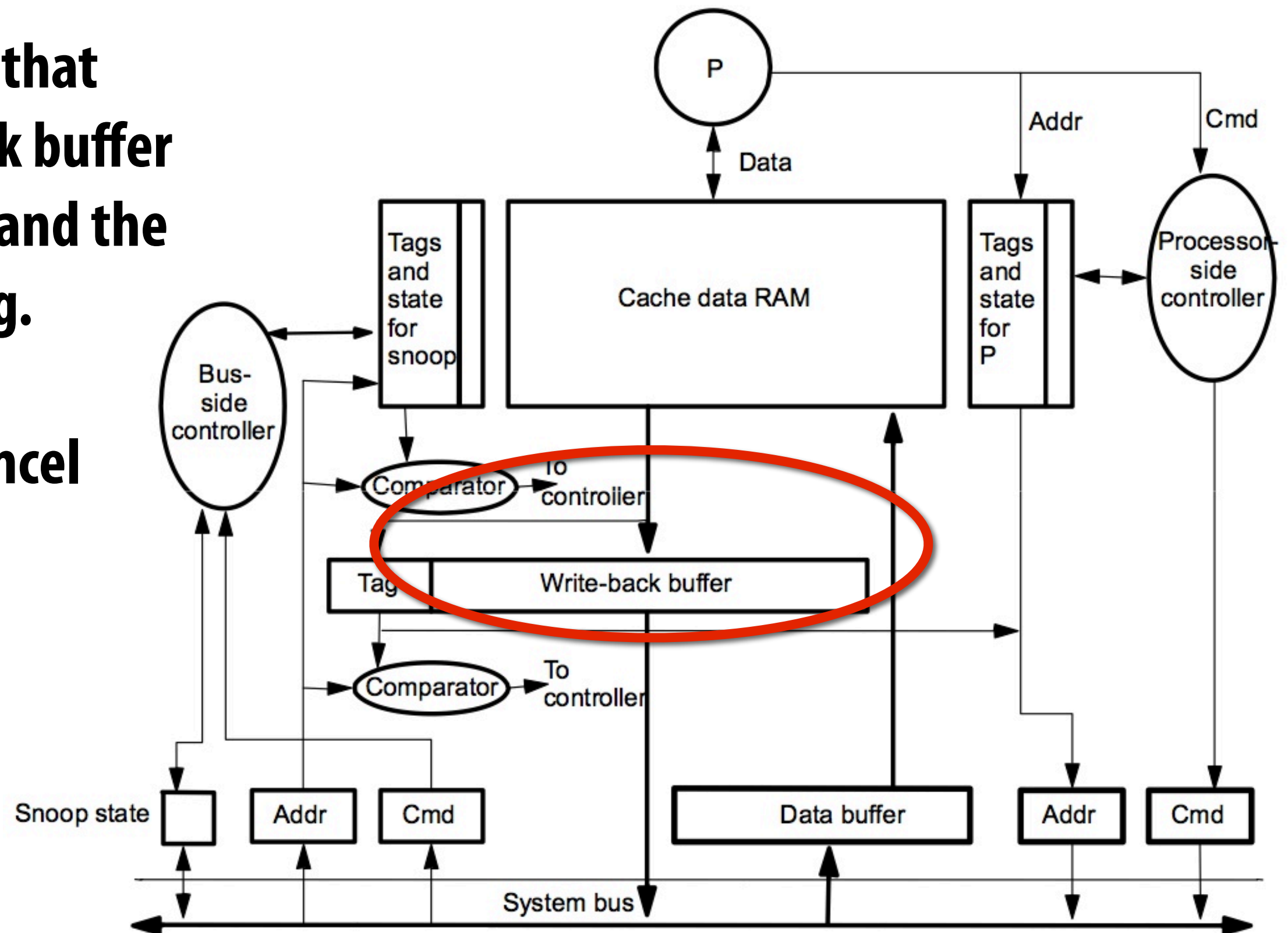
<div></div>	Address
<div></div>	Data
<div></div>	Shared
<div></div>	Dirty
<div></div>	Snoop-valid

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

# Key issues

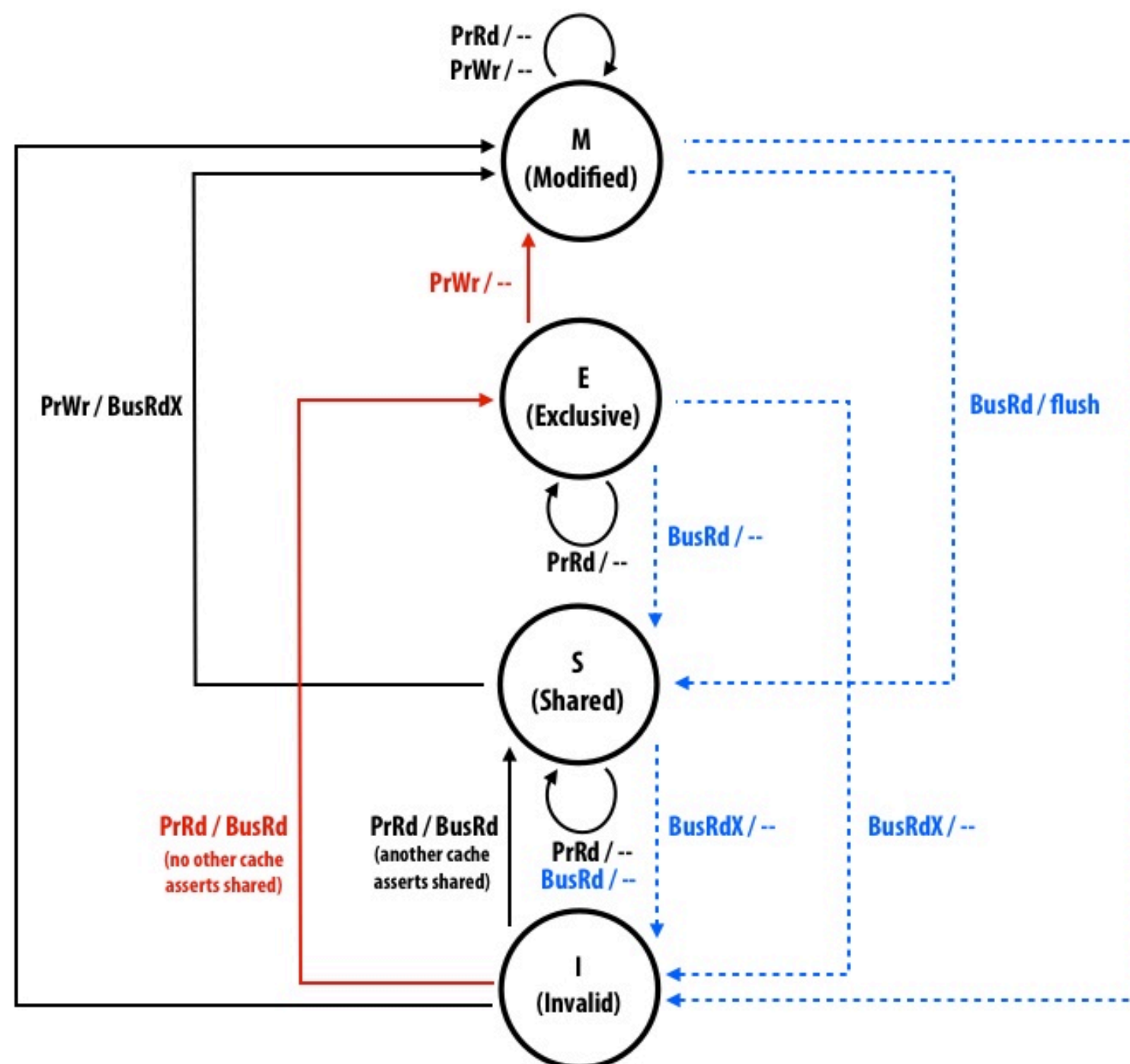
We addressed correctness issues that arise from the use of a write-back buffer by checking both the cache tags and the write-back buffer when snooping.

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



# Key issues

**We talked about ensuring write serialization: processor is held up by the cache until the “I want exclusive access” transaction appears on the bus (this is when the write “commits”).**



**We talked about how coherence protocol state transitions are not really atomic operations in a real machine (even though the bus itself is atomic), leading to possible race conditions.**

# **We discussed deadlock, livelock, and starvation**

## **Situation 1:**

**P1 has a modified copy of cache line X**

**P1 is waiting for the bus to issue BusRdX on cache line Y**

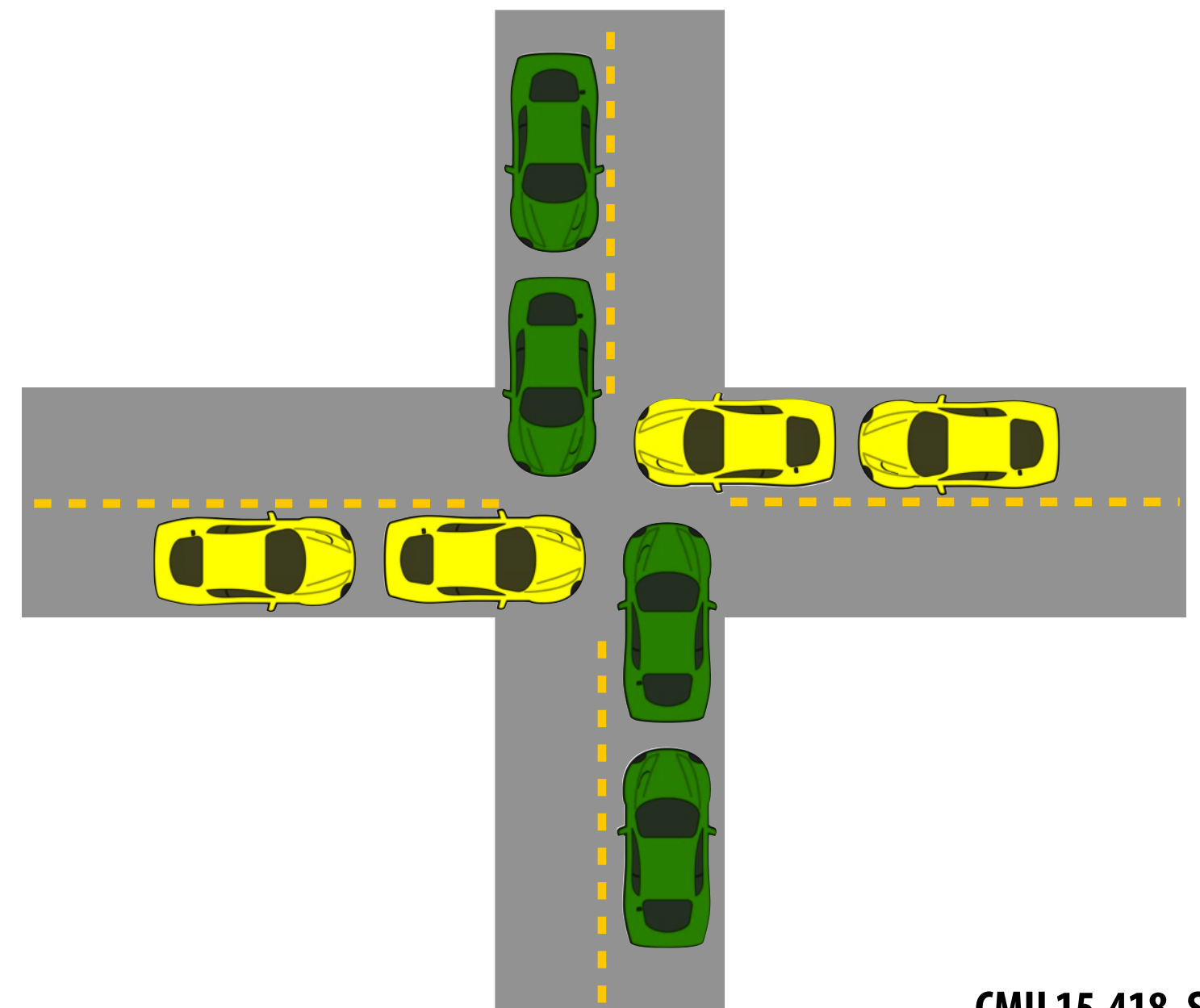
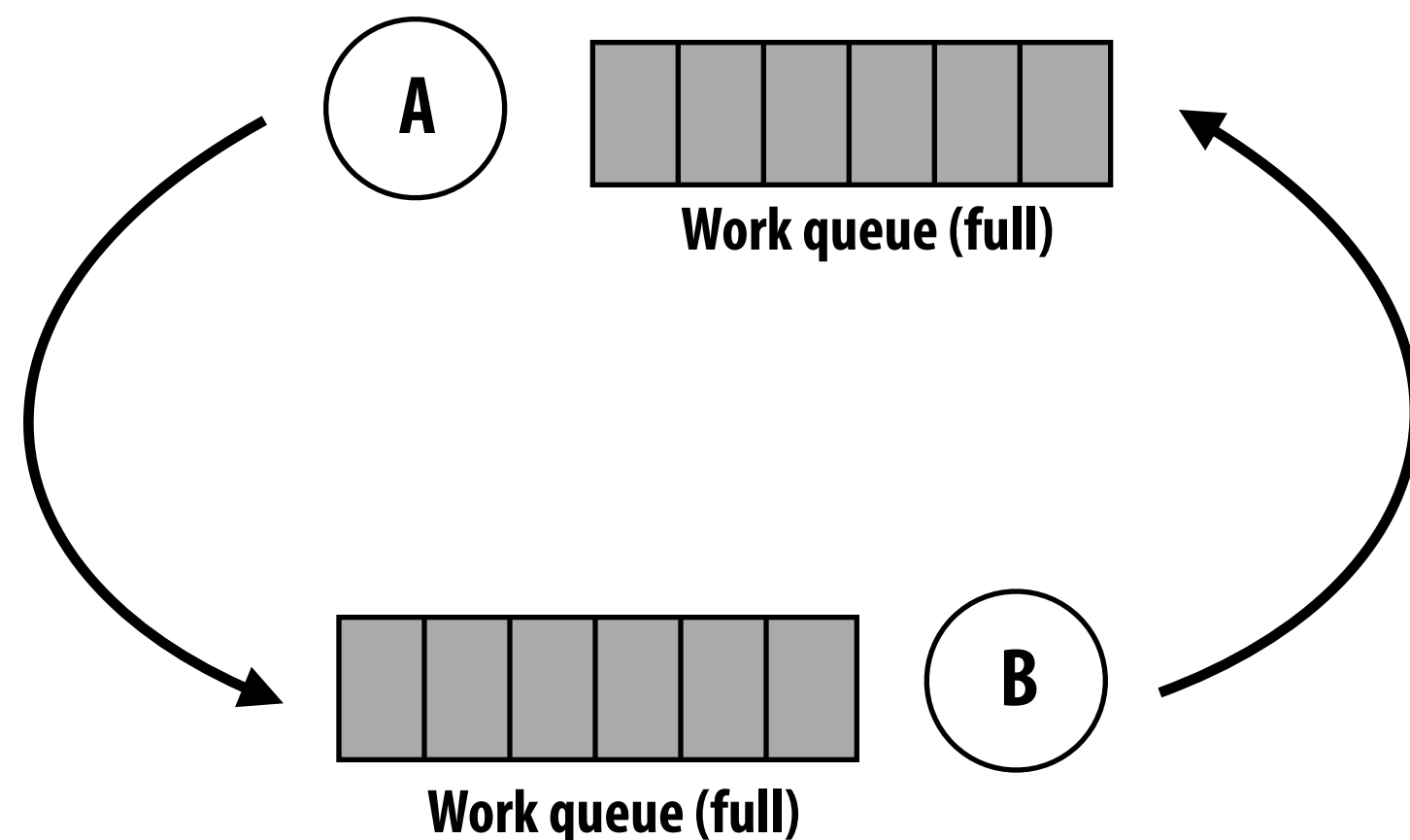
**BusRd for X appears on bus while P1 is waiting**

***FETCH DEADLOCK!***

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

# Required conditions for deadlock

1. **Mutual exclusion:** one processor can hold a given resource at once
2. **Hold and wait:** processor must hold the resource while waiting for other resources needed to complete an operation
3. **No preemption:** processors don't give up resources until operation they wish to perform is complete
4. **Circular wait:** waiting processors have mutual dependencies (a cycle exists in the resource dependency graph)



# Livelock

## Situation 2:

**Two processors simultaneously write to cache line X (in S state)**

**P1 acquires bus access (“wins bus”), issues BusRdX**

**P2 invalidates its copy of the line 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.***

# **Today's topic**

**Today we will build the system around non-atomic  
bus transactions.**

# What you should know

- **What is the major performance issue with atomic bus transactions that motivates moving to a more complex non-atomic system?**
- **Who should know the main components of a split-transaction bus, and how transactions are split into requests and responses**
- **How deadlock and livelock might occur in both atomic bus and non-atomic bus-based systems (what are possible solutions for avoiding it?)**
- **The role of queues in a parallel system (today is yet another example)**

# 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 interconnect is a limited,  
shared resource in a multi-processor system.**

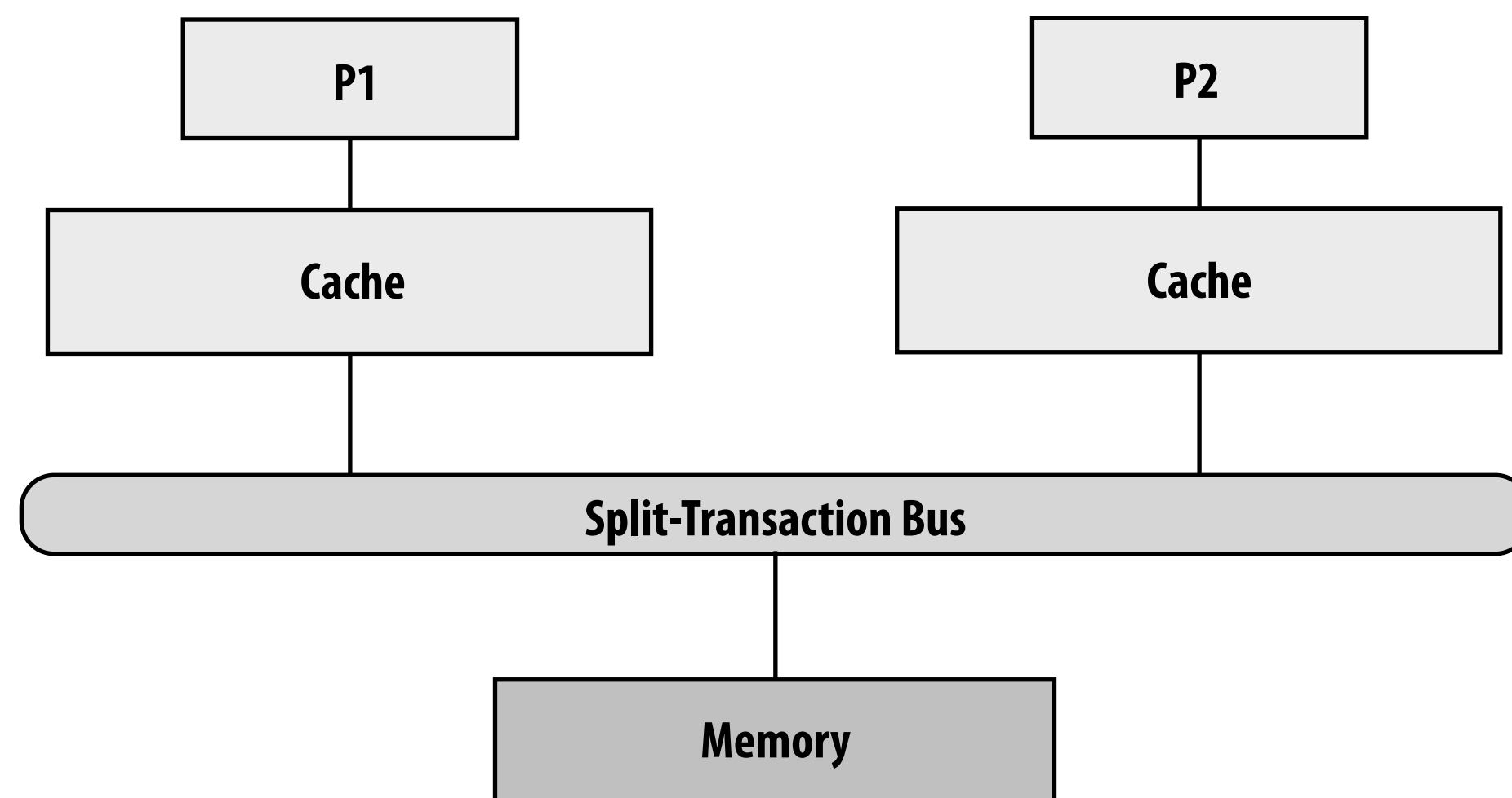
**(So it is important to use it as efficiently as possible)**

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

# Split-transaction bus

**Bus transactions are split into two transactions: a request and a response**

**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 arise due to split transactions**

- 1. How to match requests with responses?**
- 2. How to handle conflicting requests on bus?**

**Consider:**

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

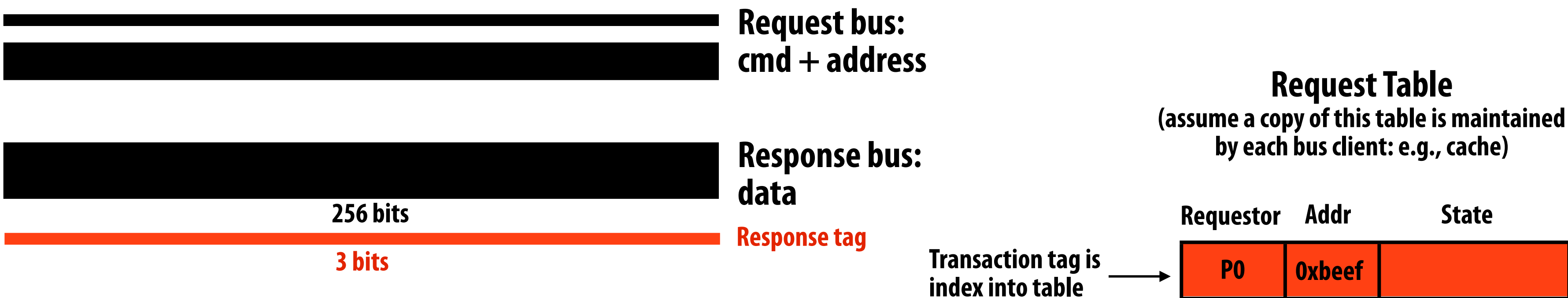
- 3. Flow control: how many requests can be outstanding at a time, and what should be done when buffers fill up?**
- 4. When are snoop results reported? During the request? or during the response?**

# A basic design

- **Up to eight outstanding requests at a time (system wide)**
- **Responses need not occur 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

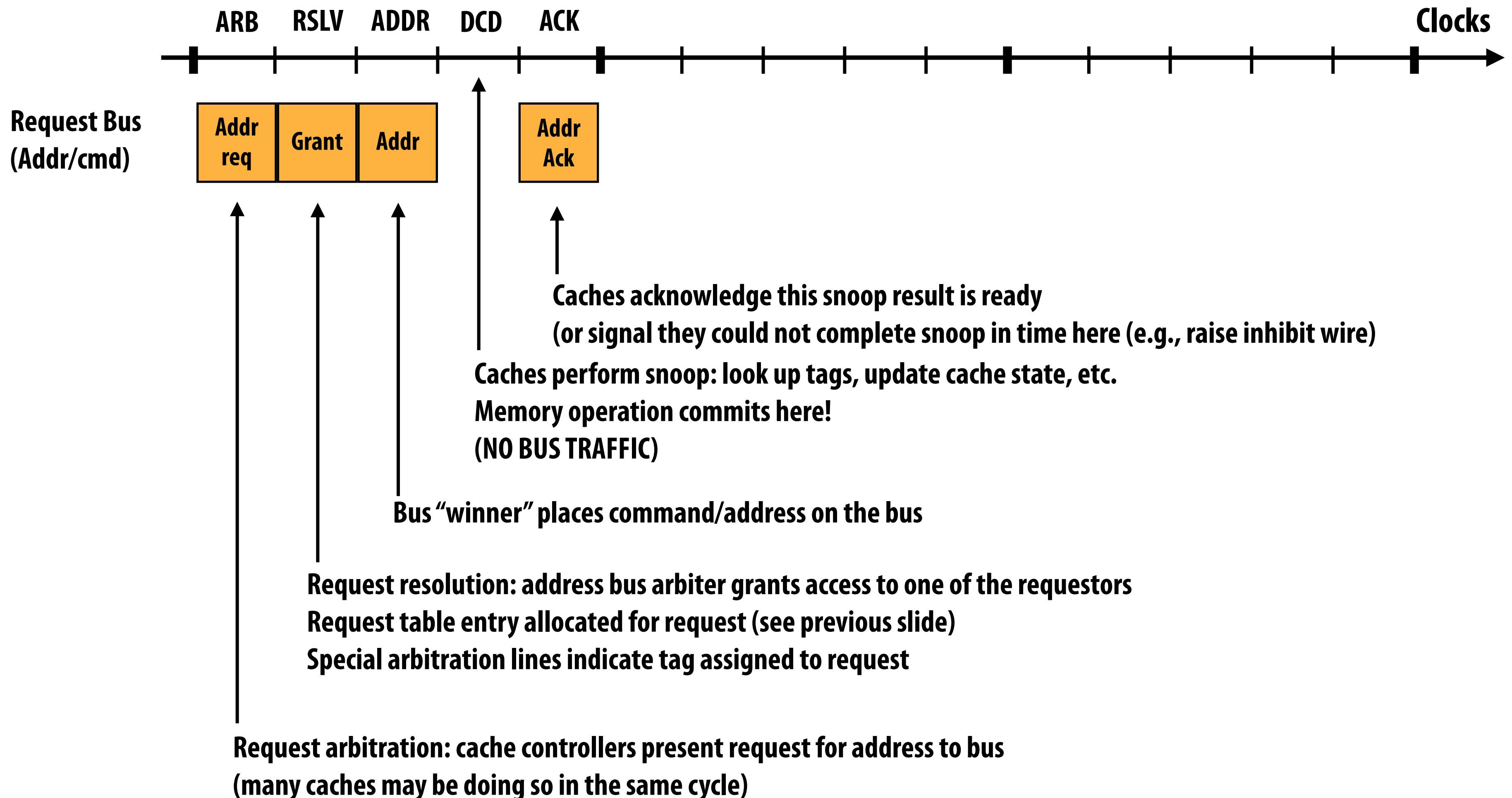
# Initiating a request

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

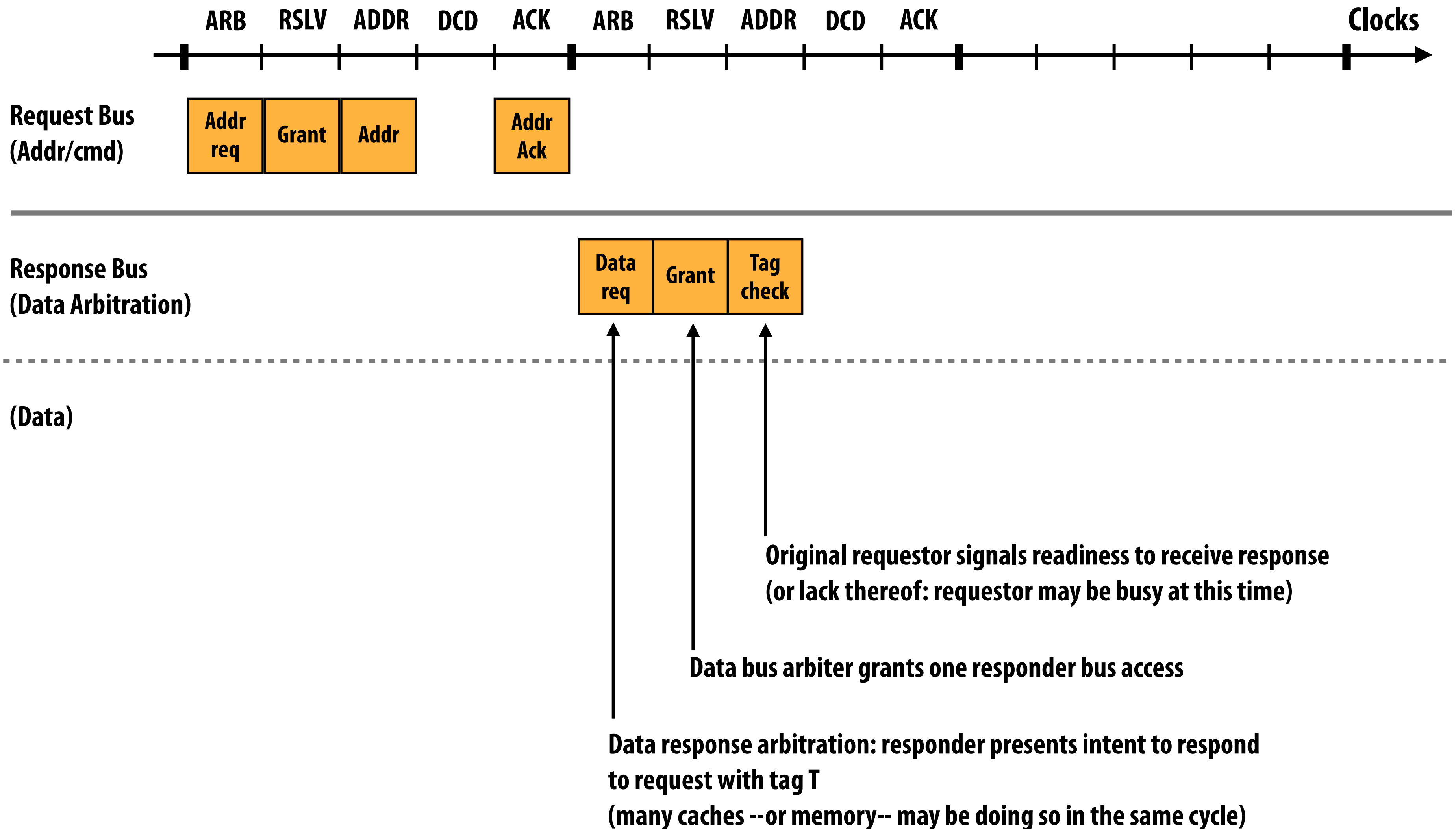


- Step 1: Requestor asks for request bus access
- Step 2: Bus arbiter grants access, assigns transaction a tag
- Step 3: Requestor places command + address on the request bus

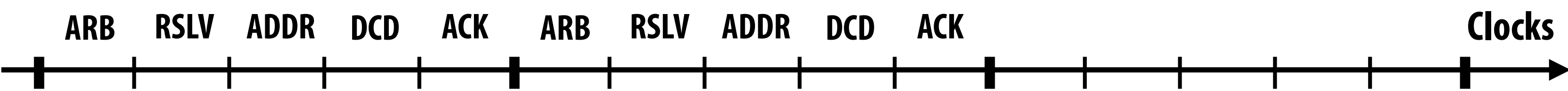
# Read miss: cycle-by-cycle bus behavior (phase 1)



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



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



Request Bus  
(Addr/cmd)



Response Bus  
(Data Arbitration)

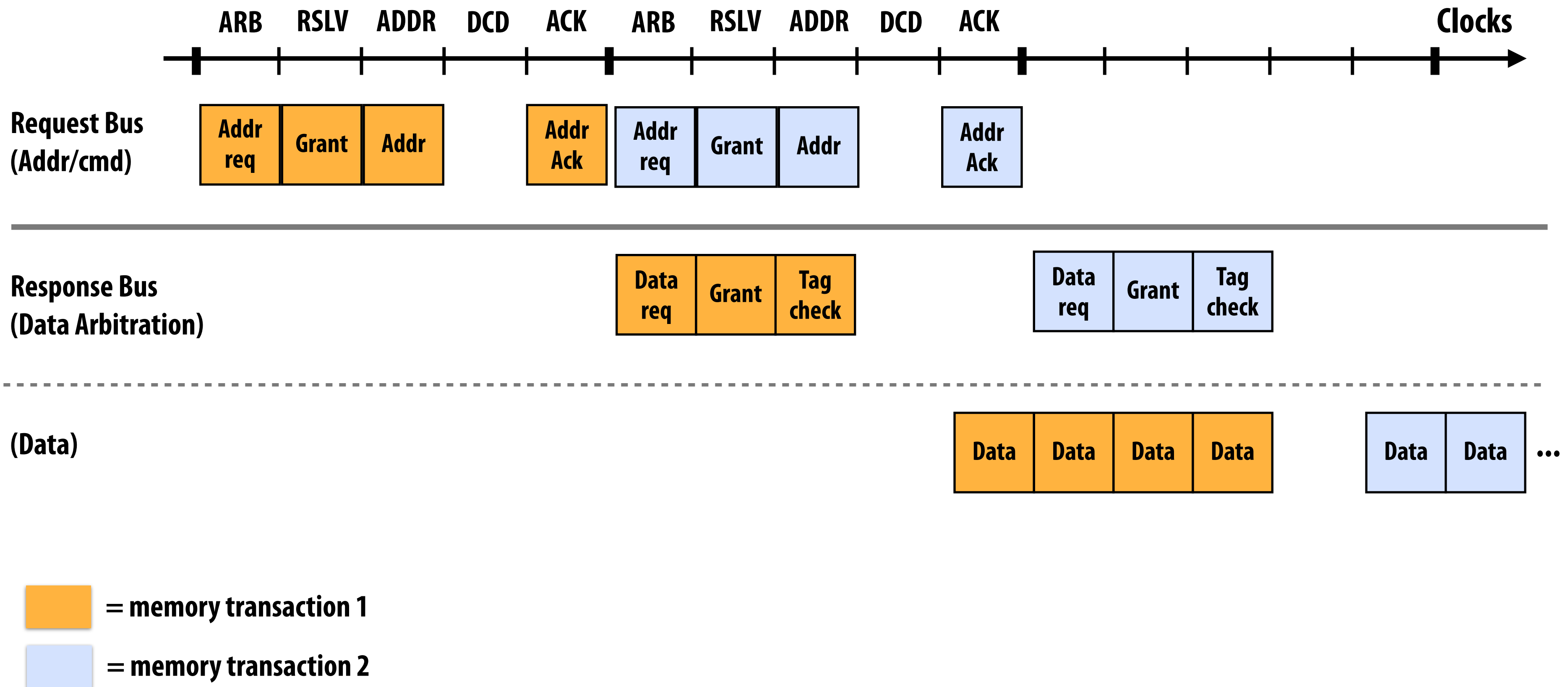


(Data)



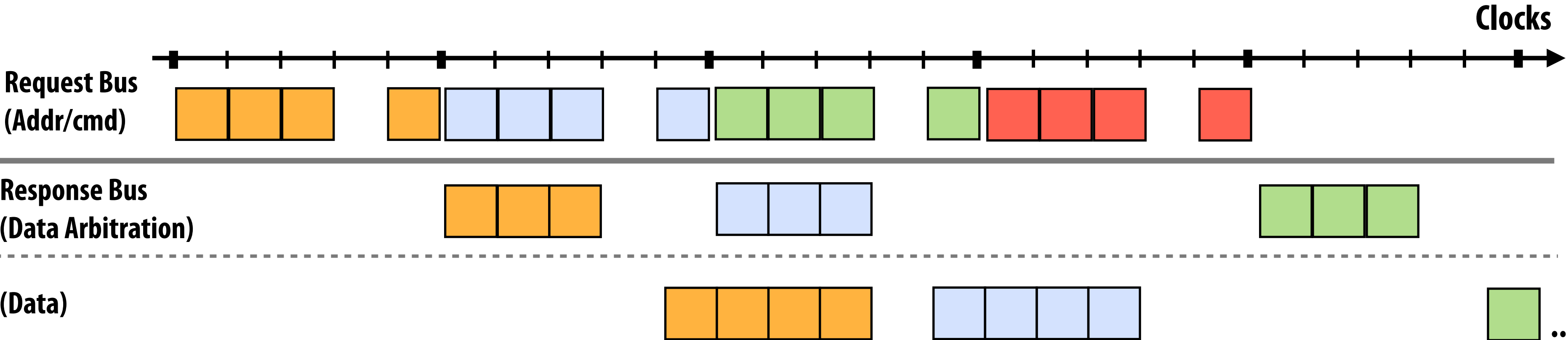
Responder places response data on data bus  
Caches present snoop result for request with the data  
Request table entry is freed  
Here: assume 128 byte cache lines → 4 cycles on 256 bit bus


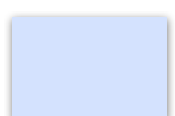
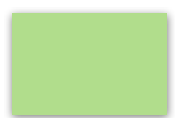

# Pipelined transactions



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

# Pipelined transactions



-  = memory transaction 1
-  = memory transaction 2
-  = memory transaction 3
-  = memory transaction 4

# Key issues to resolve

## ■ Conflicting requests

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

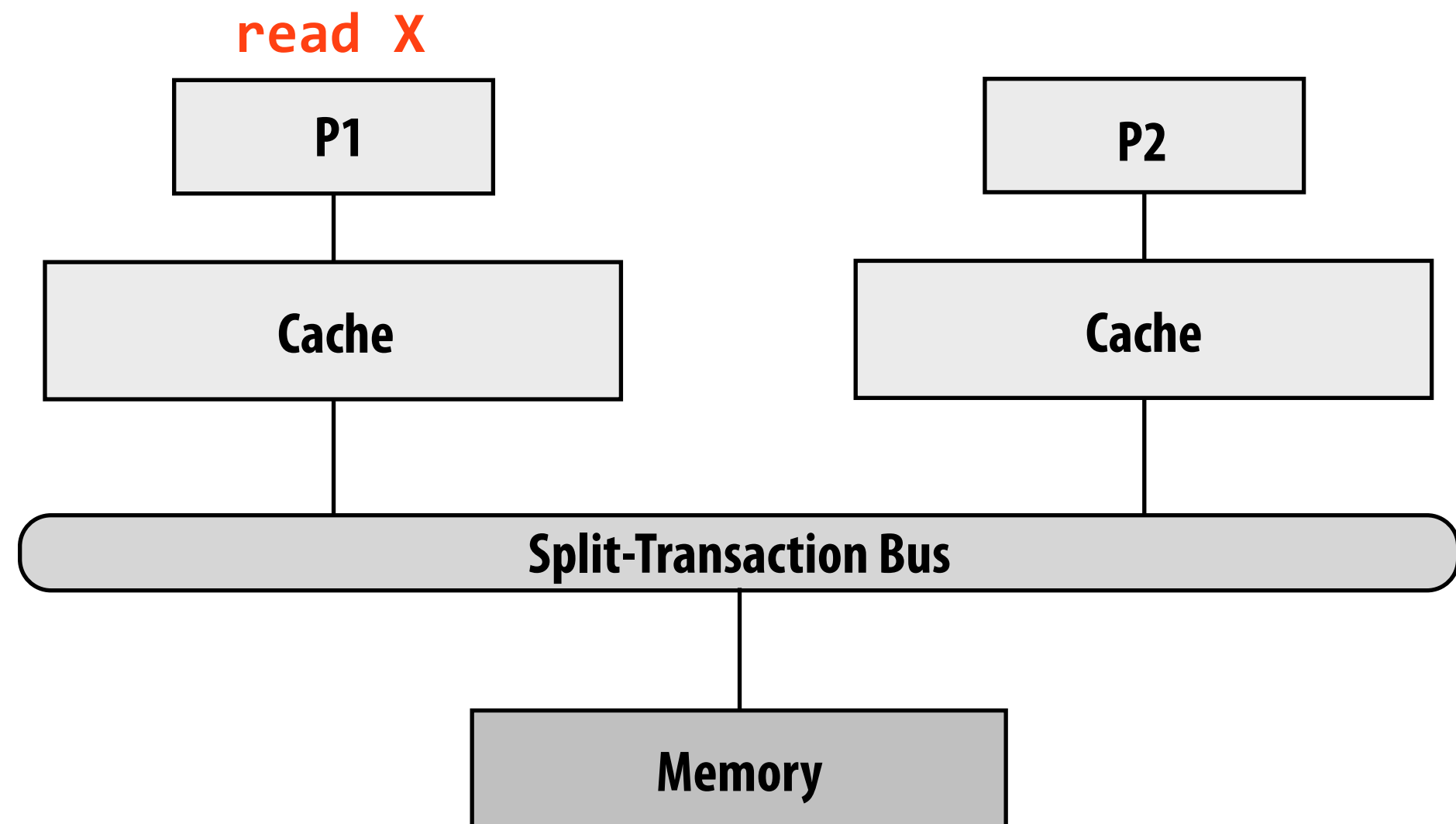
## ■ Flow control:

- Caches/memory have buffers for receiving data off the bus
- If the buffer fills, client NACKs relevant requests or responses (NACK = negative acknowledgement)
- Triggers a later retry

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

P1 Request Table

Requestor	Addr	State
P2	X	Op: BusRdX

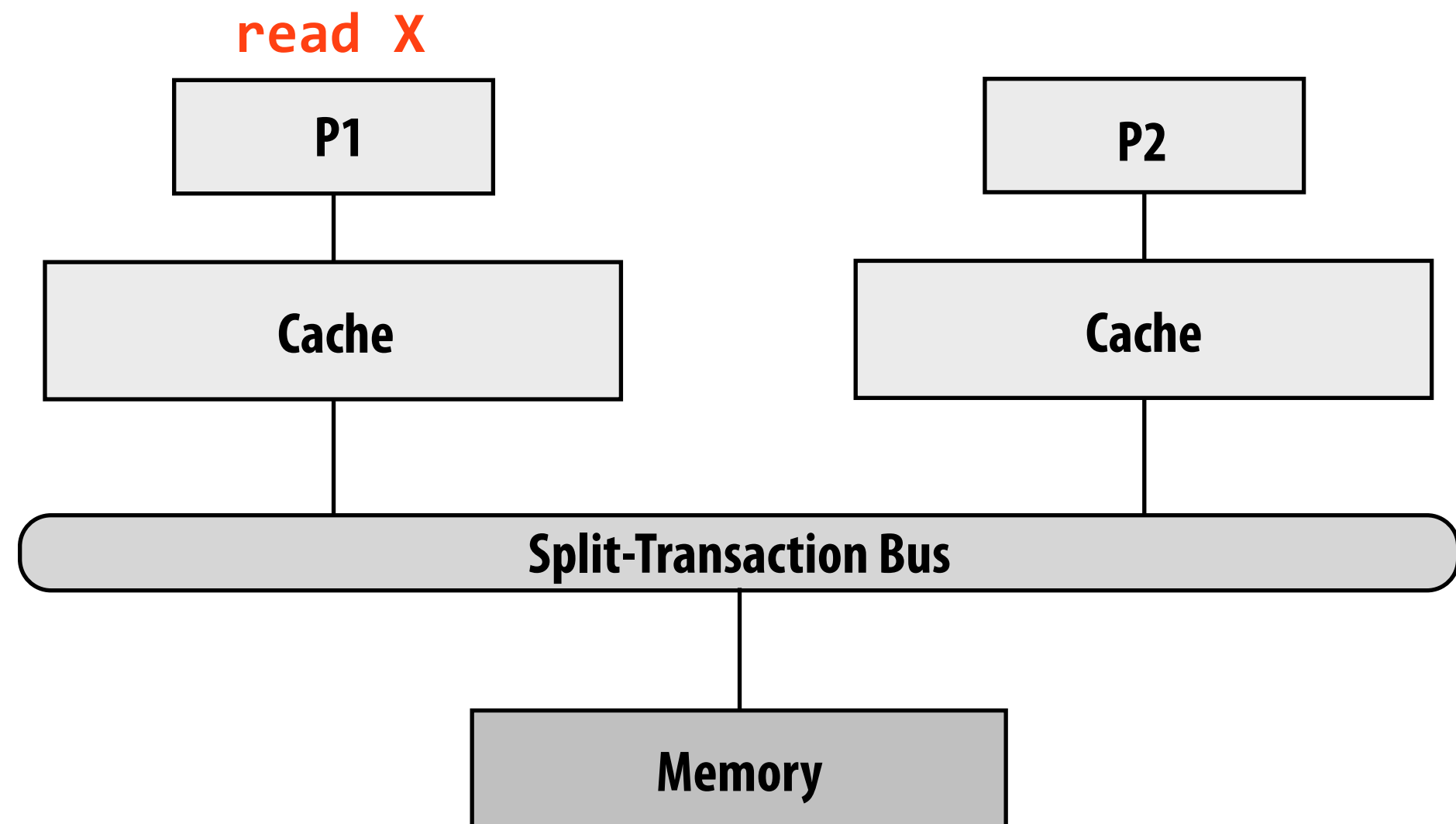


If there is a conflicting outstanding request (as determined by checking the request table), cache must hold request until conflict clears

# Situation 2: P1 read miss to X, read transaction involving X is outstanding on bus

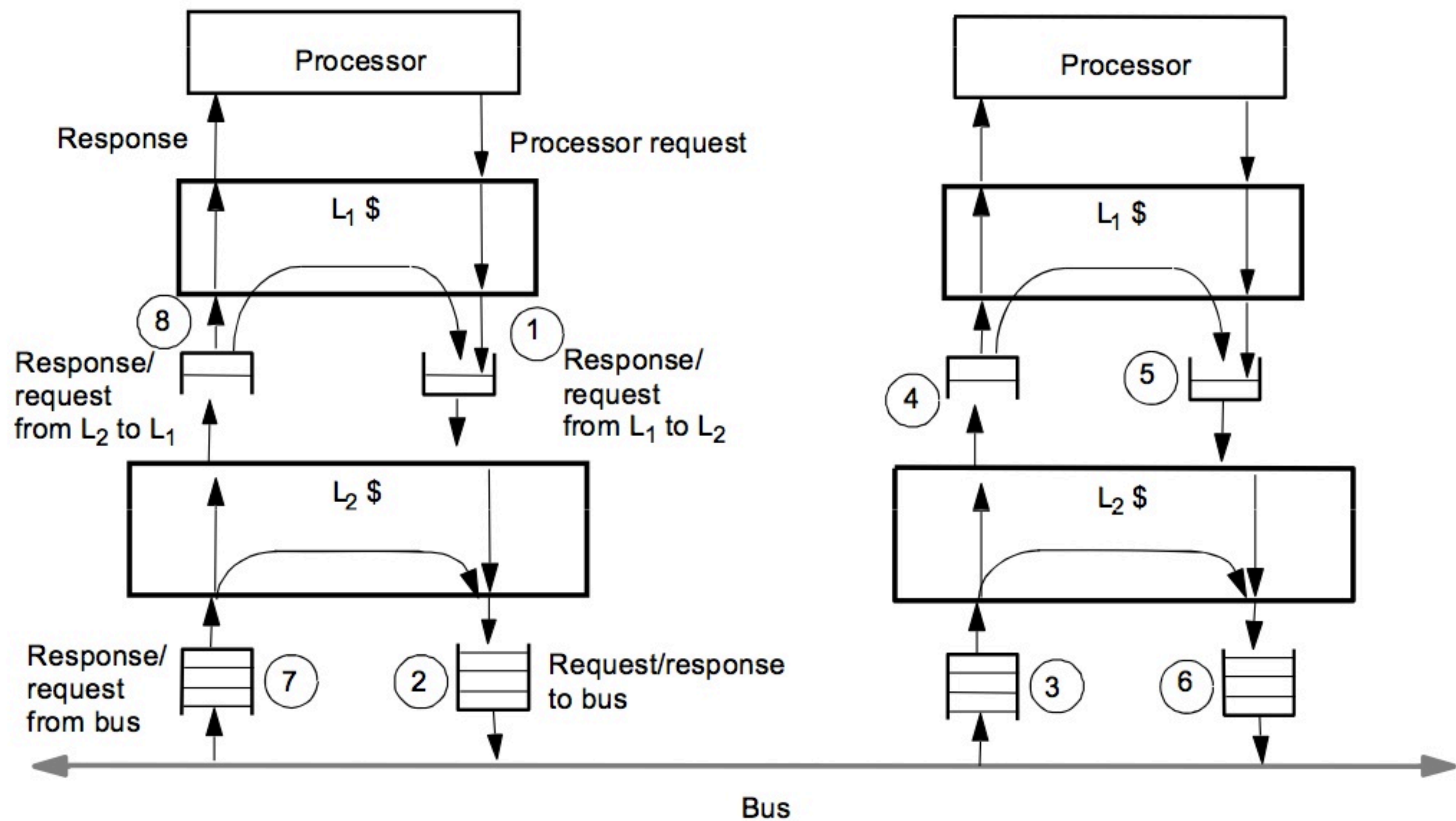
P1 Request Table

Requestor	Addr	State
P2	X	Op: BusRd , <b>share</b>

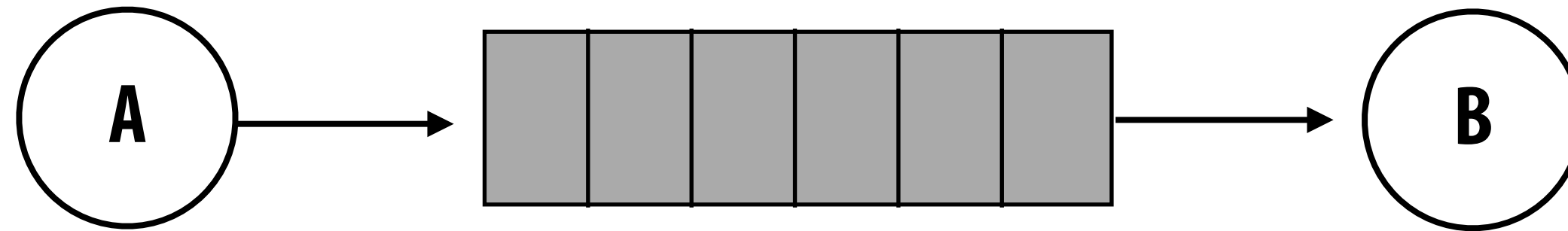


**If outstanding request is a read: there is no conflict. No need to make a new bus request, just listen for the response to the outstanding one.**

# Multi-level cache hierarchies



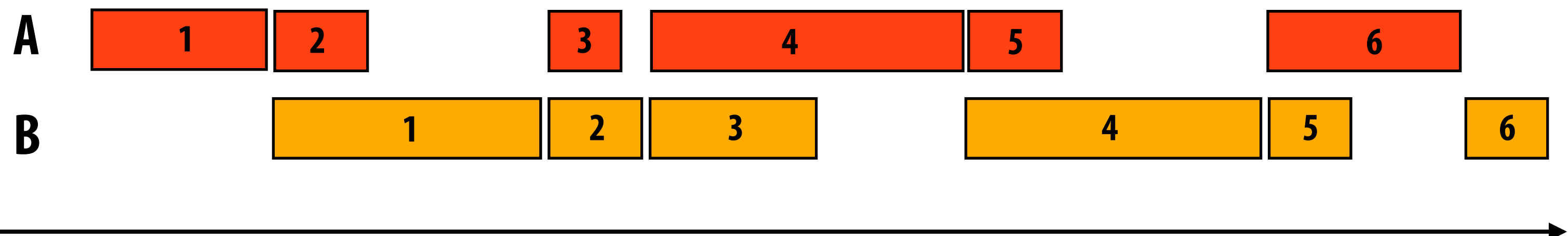
# Why do we have queues?



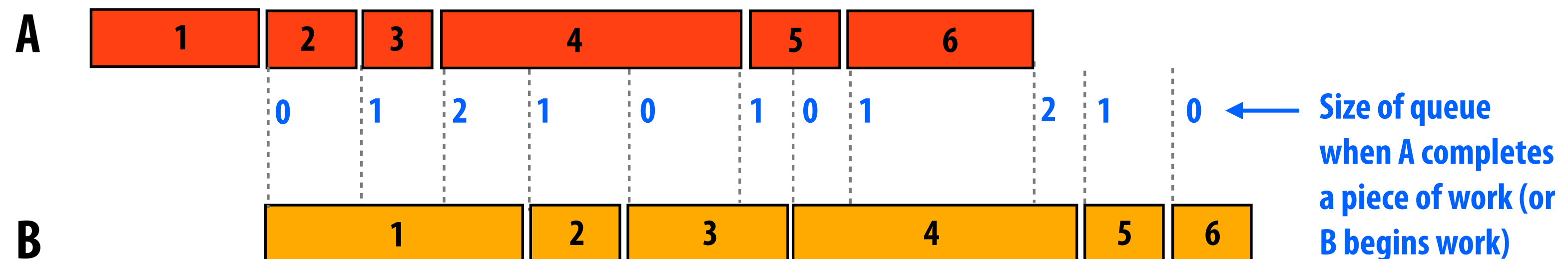
To accommodate variable (unpredictable) rates of production and consumption.

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

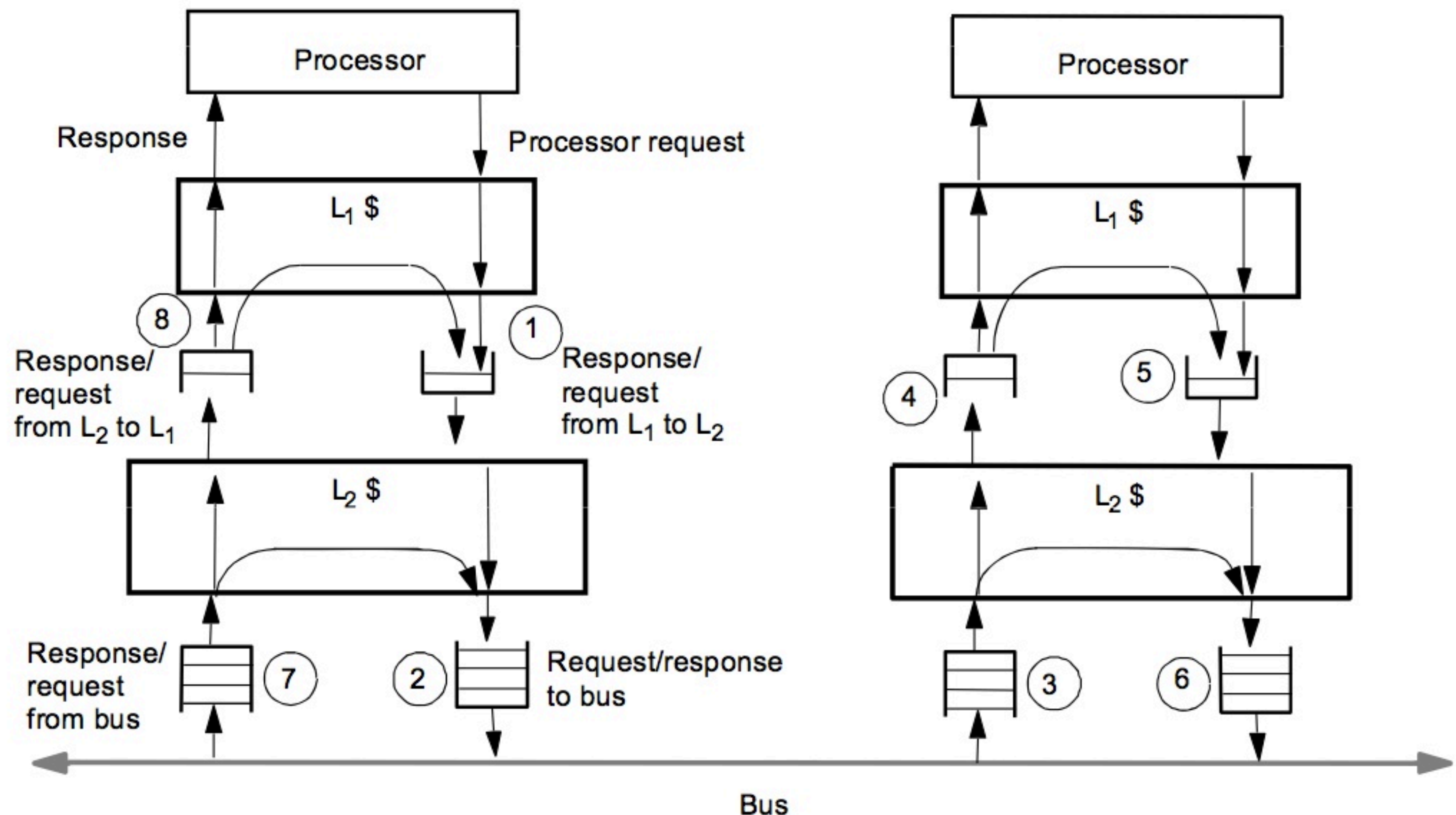
No queue:  
stalls exist



With queue of  
size 2: A and B  
never stall



# Consider fetch deadlock problem

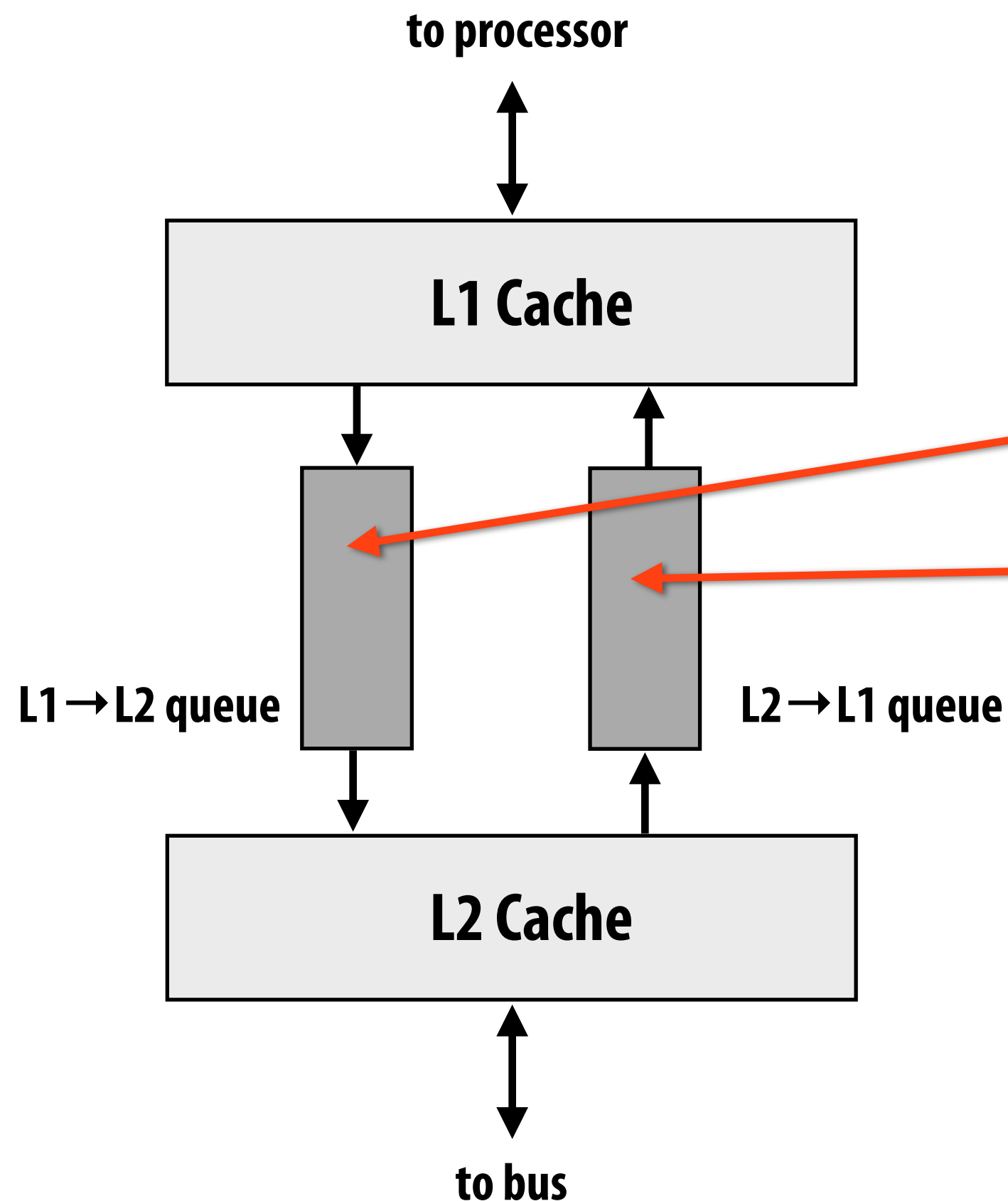


**Assume one outstanding memory request per processor.**

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

# Deadlock due to full queues

Assume buffers are sized so that max queue size is one message.



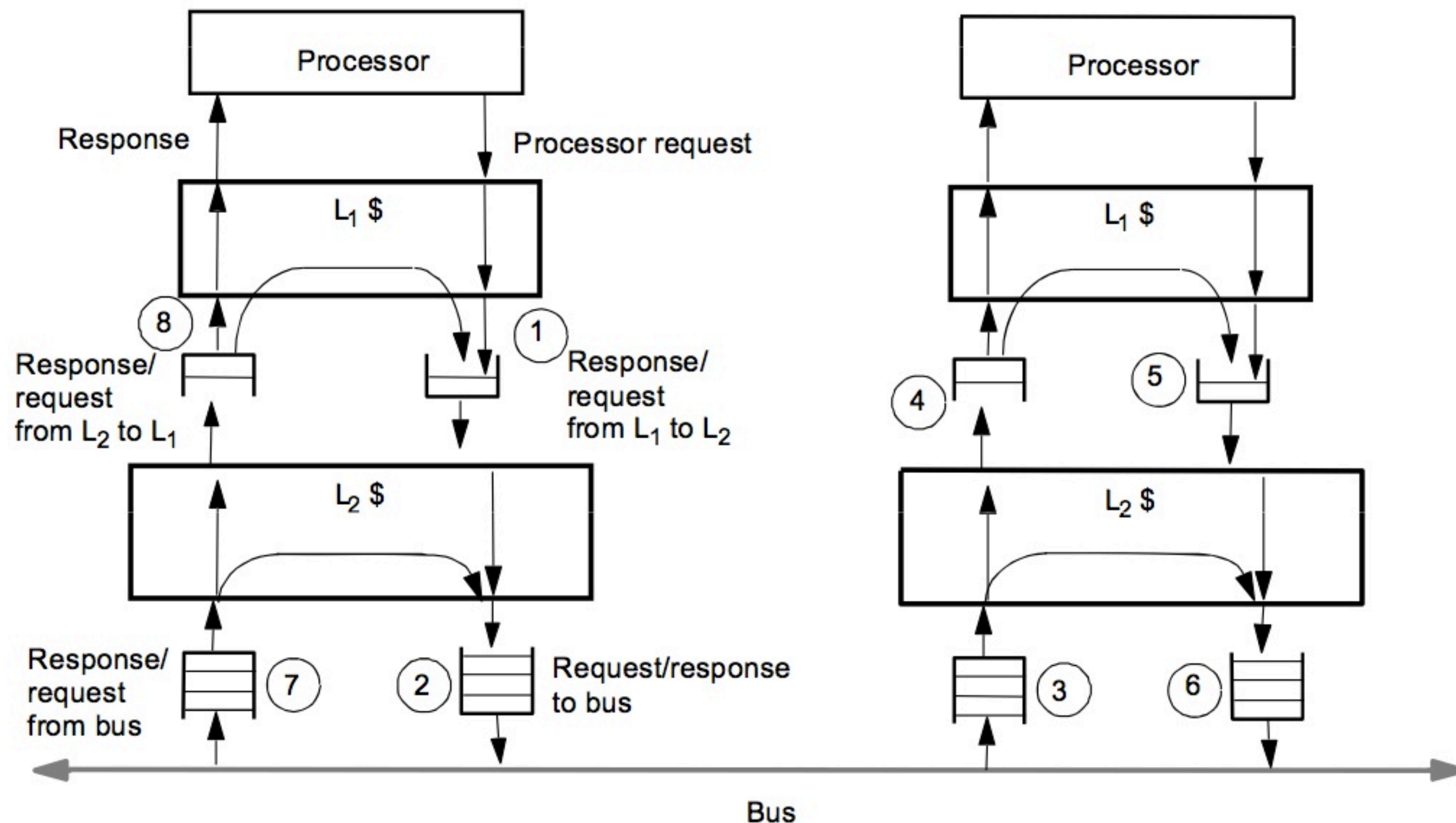
Outgoing read request (initiated by this processor)

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

# Multi-level cache hierarchies

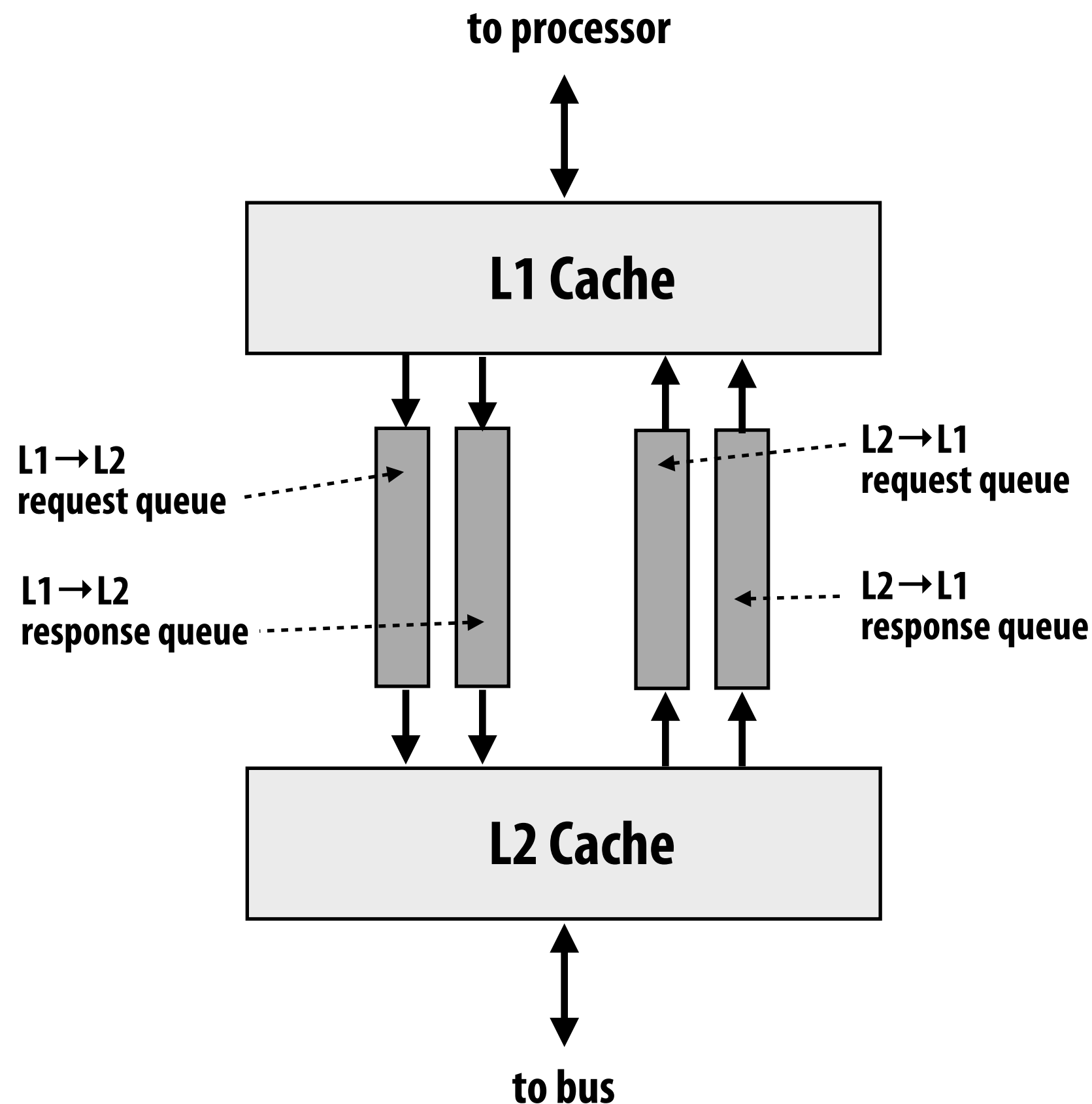


**Assume one outstanding memory request per processor.**

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

**Sizing all buffers to accommodate the maximum number of outstanding requests on bus is one solution to avoiding deadlock. But an expensive one!**

# Avoiding buffer deadlock with separate request/response queues



System classifies all transactions as requests or responses

Key insight: responses can be completed without generating further transactions!

**Requests INCREASE queue length**

**But responses REDUCE queue length**

While stalled attempting to send a request, cache must be able to service responses.

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

# Putting it all together

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

```
int x = 10;           // assume this is a write to memory (value not  
                      // stored in register)
```

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

**int x = 10;**

**Virtual address to physical address conversion (TLB lookup)**

**TLB miss**

**TLB update (might involve OS)**

**OS may need to swap in page to get the appropriate page table (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 in row buffer. May need to activate new row.**

**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 is notified data exists**

**Instruction proceeds**