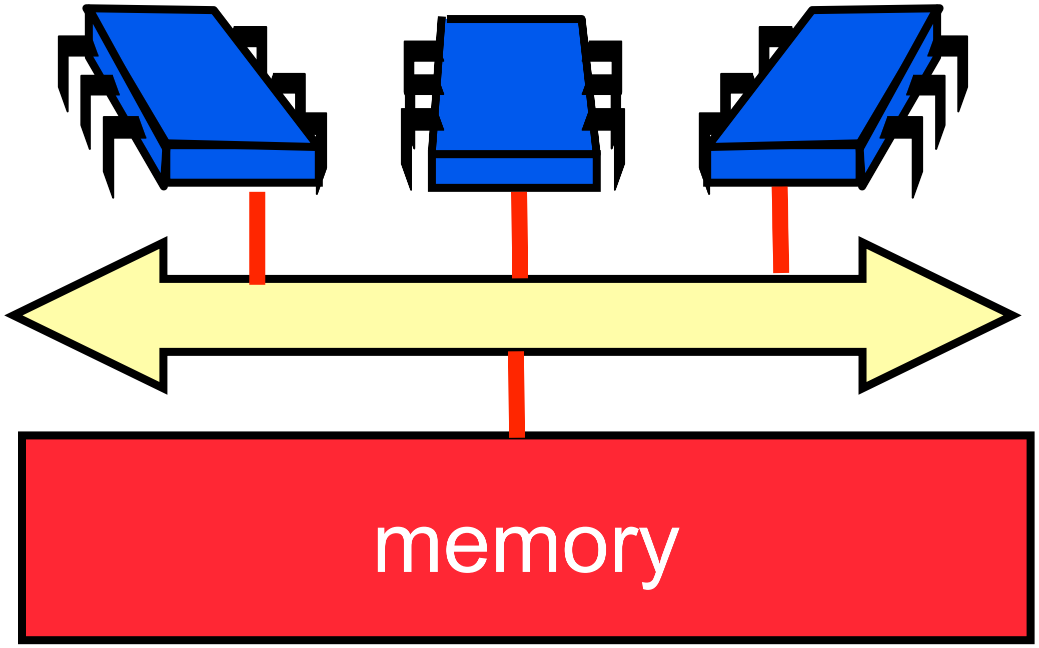
CS-206 Concurrency

Lecture 2
Multiprocessors



Spring 2015
Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/

Where are We?

		Lecture & Lab		
M	T	W	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	2	25-Feb	26-Feb	27-Feb
2-Mar	3-Ma	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- Today
  - Multicore Architecture
  - Memory Hierarchies
- Next Wednesday
  - Concurrency vs. Parallelism
  - Performance
- Labs
  - HW1: Today

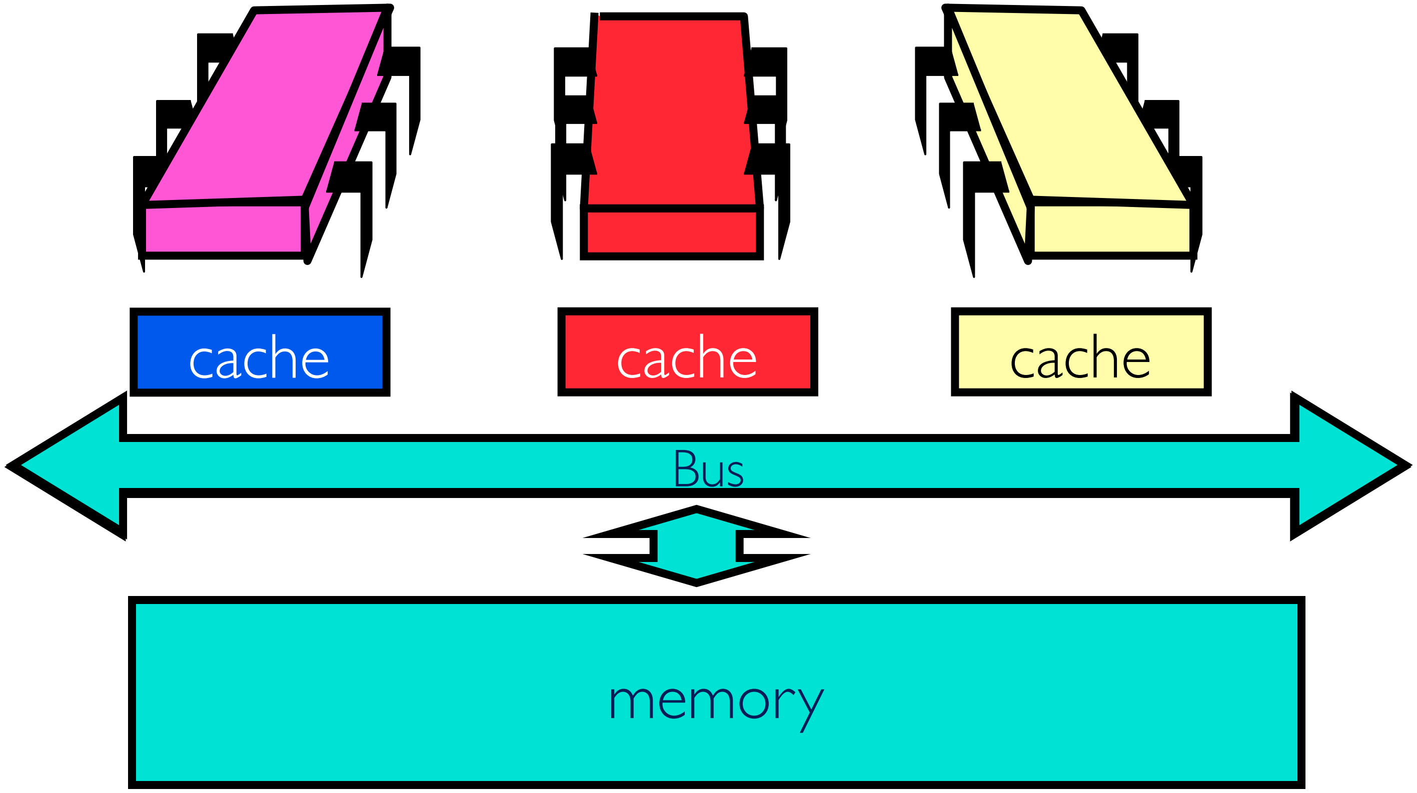
Multiprocessor Architecture

- Abstract models are (mostly) OK to understand algorithm correctness and progress
- To understand how concurrent algorithms actually perform
- You need to understand something about multiprocessor architectures
- Detailed nuts & bolts? next year

Pieces

- Processors
- Threads
- Interconnect
- Memory
- Caches

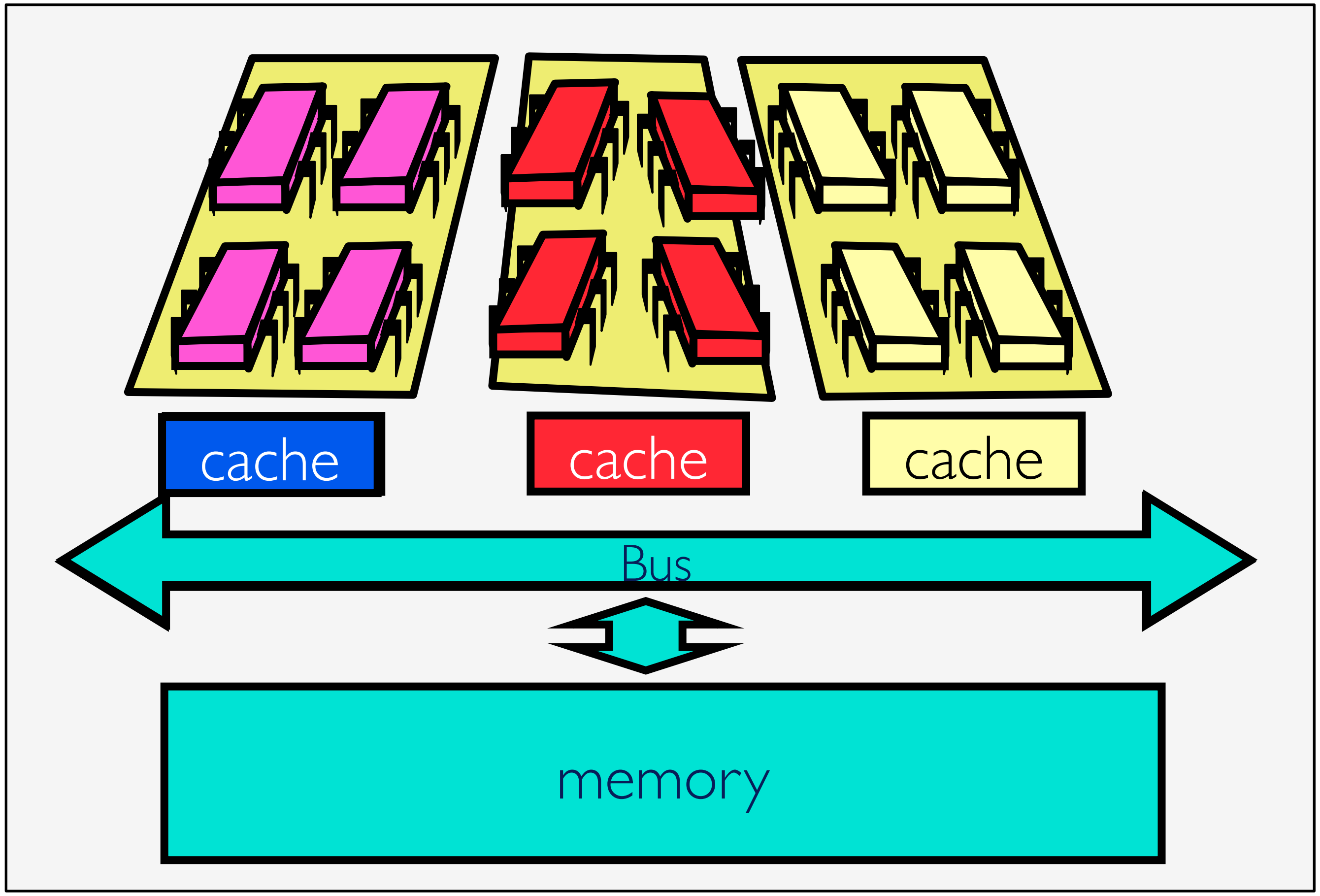
Old-School Multiprocessor



Old School

- Processors on different chips
- Processors share off chip memory resources
- Communication between processors typically slow

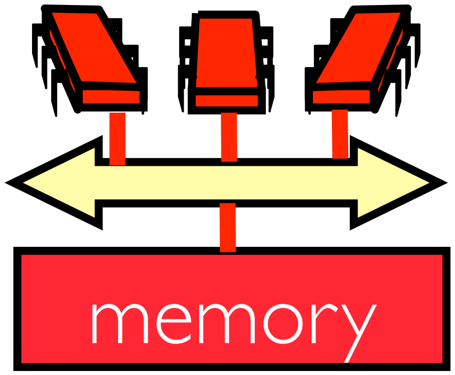
Multicore Architecture



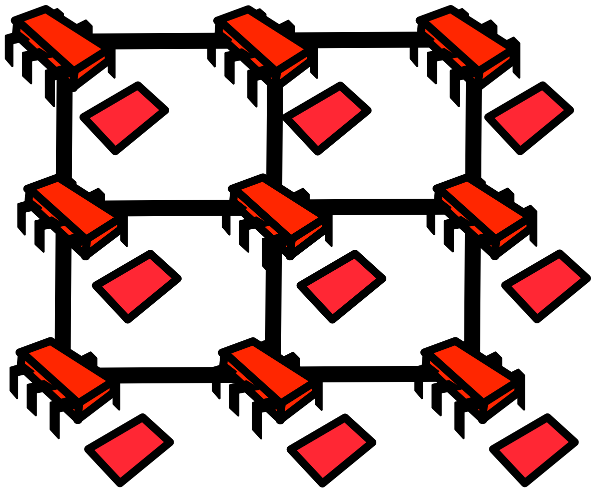
Multicore

- All Processors on same chip
- Processors share on chip memory resources
- Communication between processors now very fast

SMP vs NUMA



SMP



NUMA

- SMP: symmetric multiprocessor
- NUMA: non-uniform memory access
- CC-NUMA: cache-coherent …

Future Multicores

- These days: no longer SMP
  - Machines are getting bigger
  - Bus becomes a bottleneck (later in CS-370)
- All machines are NUMA
  - Most NUMA machines: multiple chips, single board
  - E.g., Facebook server
  - Some NUMA machines: multiple boards, entire rack
  - E.g., Oracle SPARC servers
- For this course, we assume SMP
  - Simplify the model for programming

Understanding the Pieces

- Lets try to understand what the pieces that make the multiprocessor machine are
- And how they fit together

Processors

- Cycle:
  - Fetch and execute one instruction
- Cycle times change
  - 1980: 10 million cycles/sec
  - 2005: 3,000 million cycles/sec

Computer Architecture

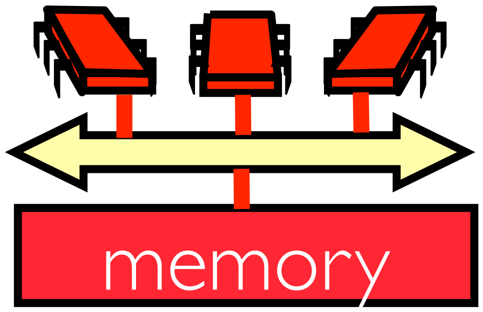
- Measure time in cycles
  - Absolute cycle times change
- Memory access: ~100s of cycles
  - Changes slowly
  - Mostly gets worse

Threads

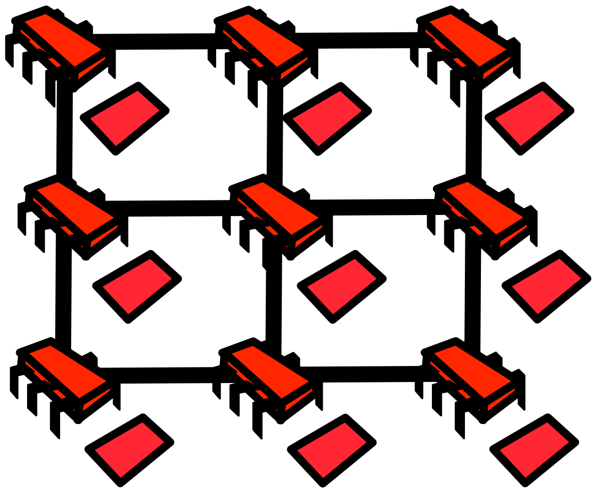
- Execution of a sequential program
- Software, not hardware
- A processor can run a thread
- Put it aside
  - Thread does I/O
  - Thread runs out of time
- Run another thread

Interconnect

- Bus
  - Like a tiny Ethernet
  - Broadcast medium
  - Connects
    - Processors to memory
    - Processors to processors
- Network
  - Tiny LAN
  - Mostly used on large machines



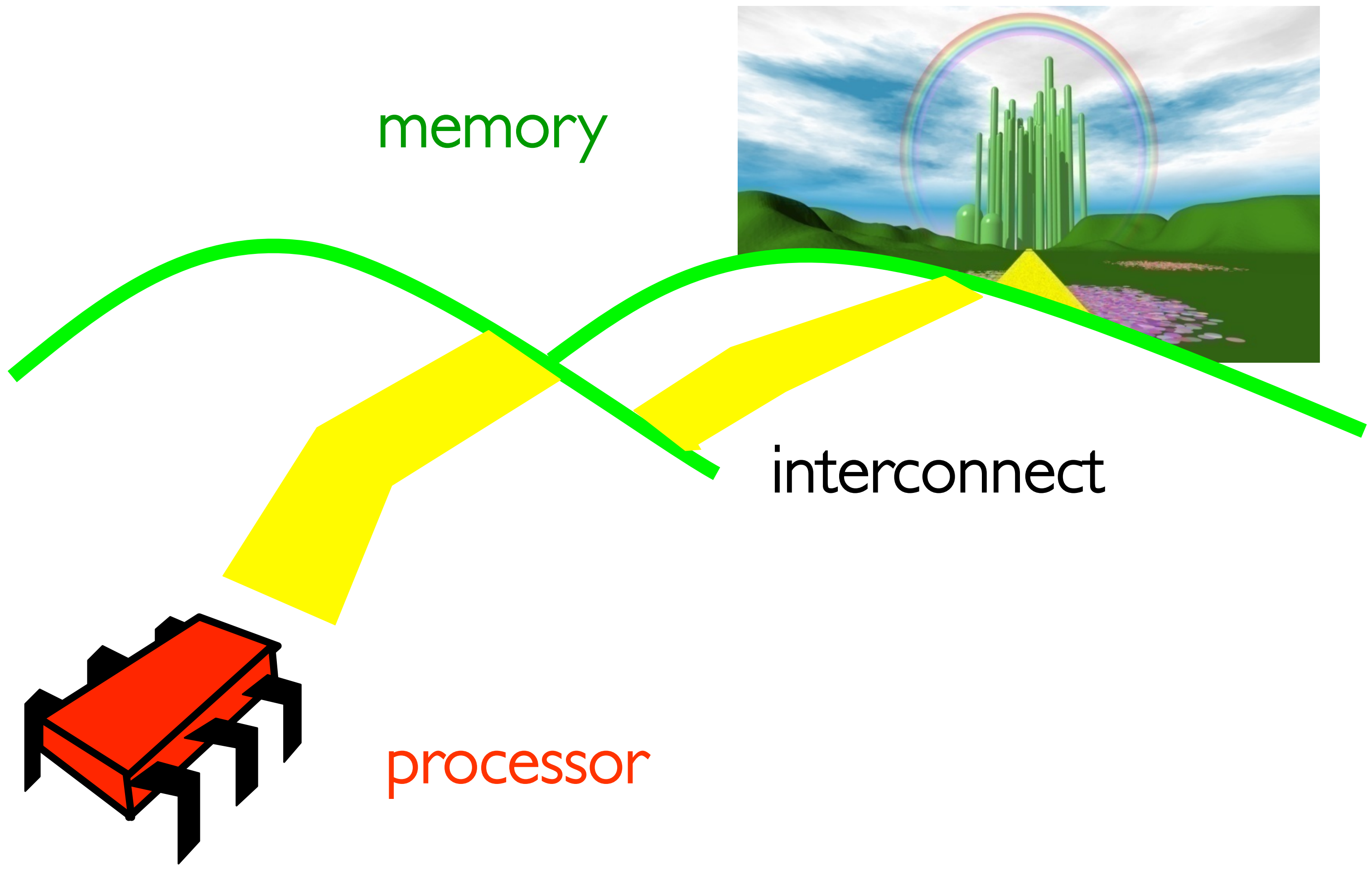
SMP



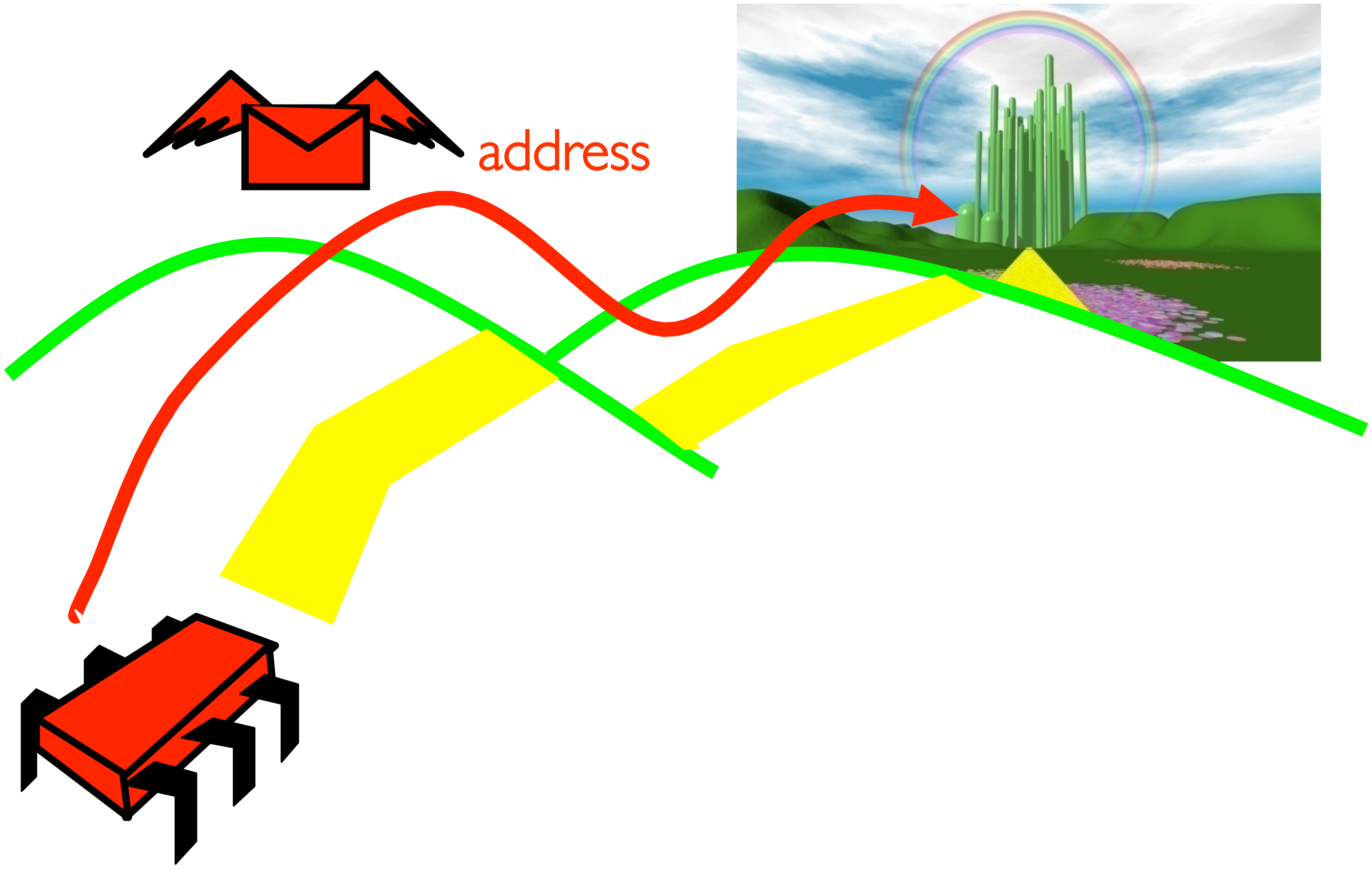
Interconnect

- Interconnect is a finite resource
- Processors can be delayed if others are consuming too much
- Avoid algorithms that use too much bandwidth

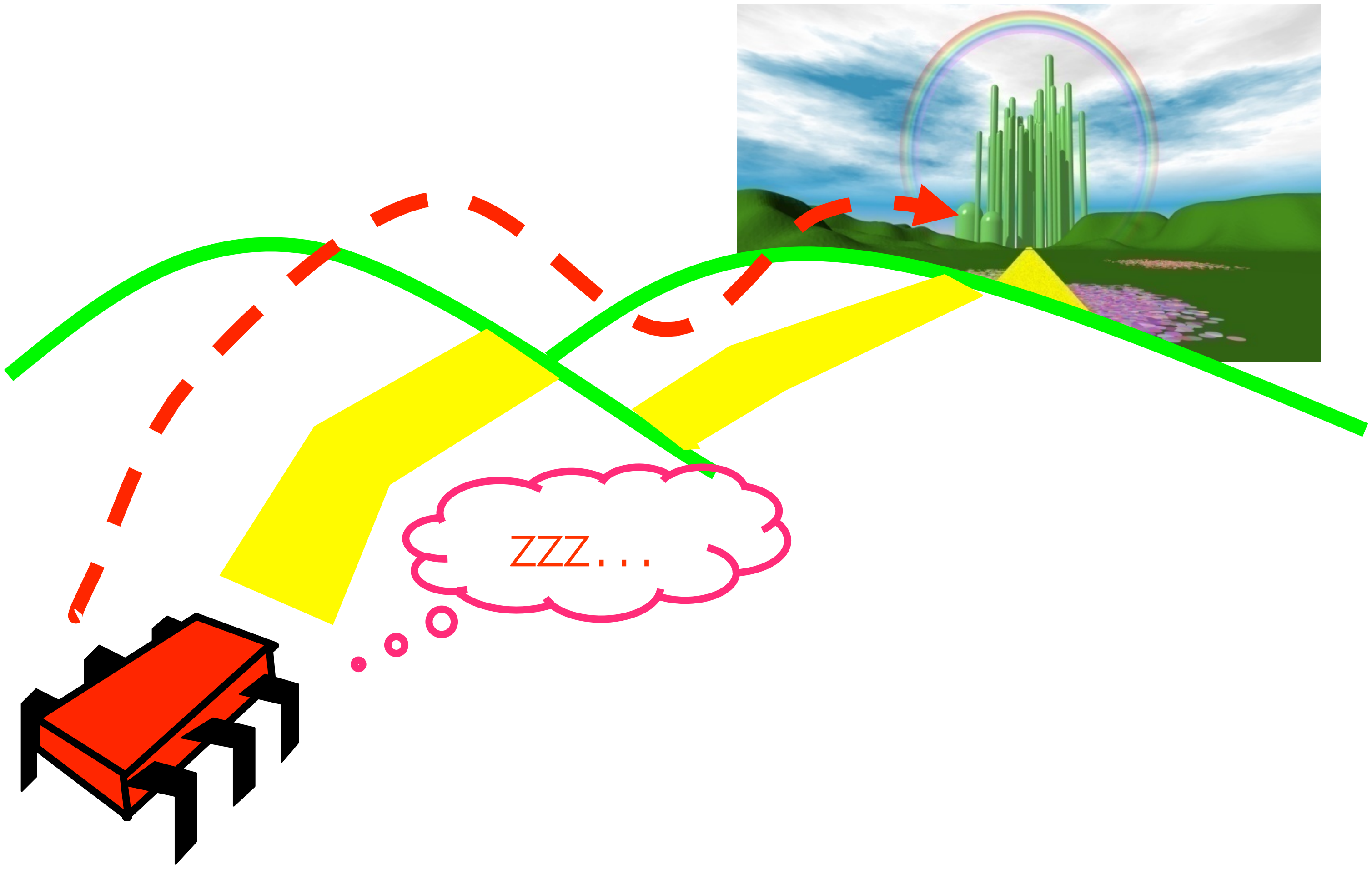
Processor and Memory are Far Apart



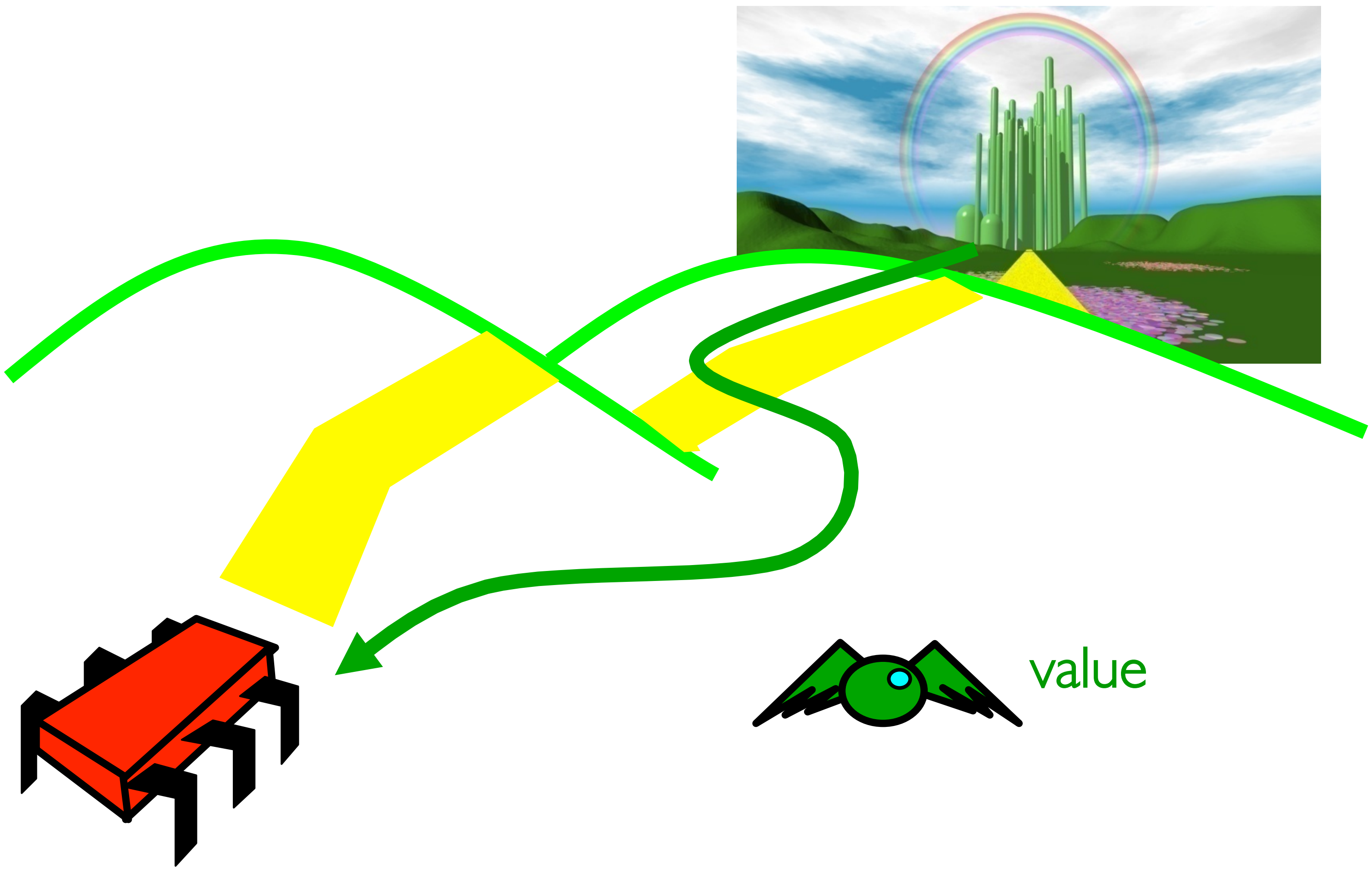
Reading from Memory



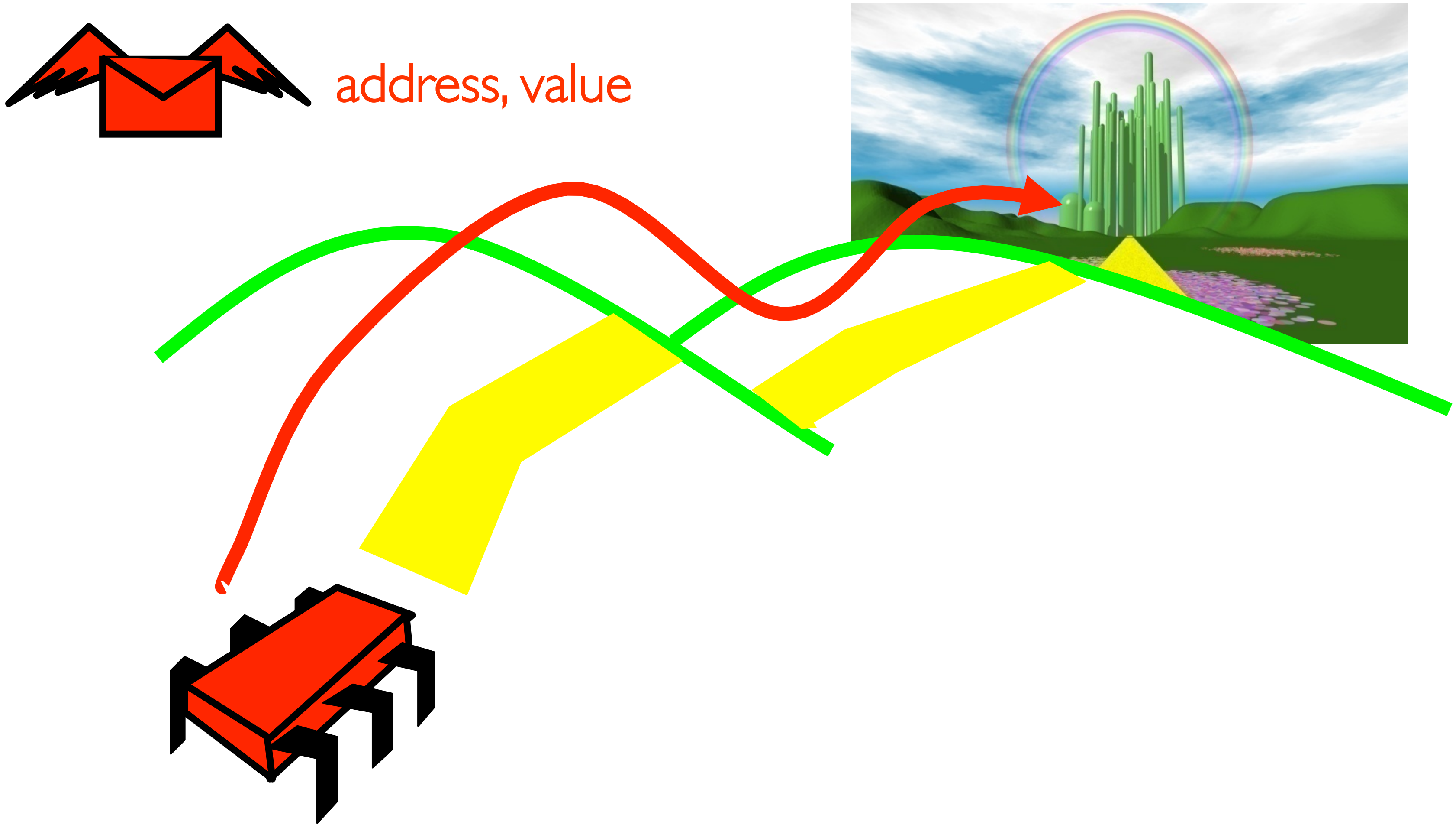
Reading from Memory



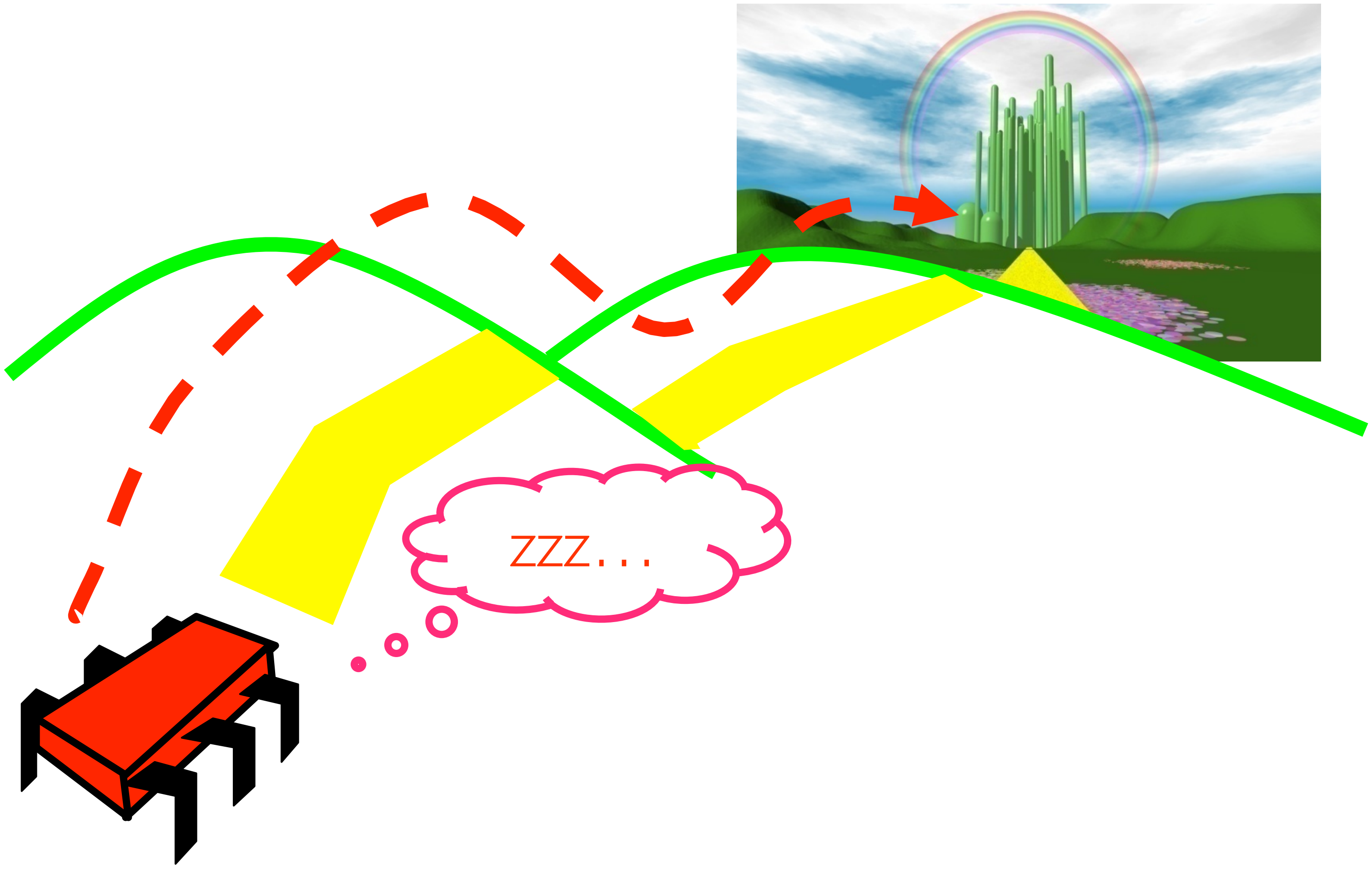
Reading from Memory



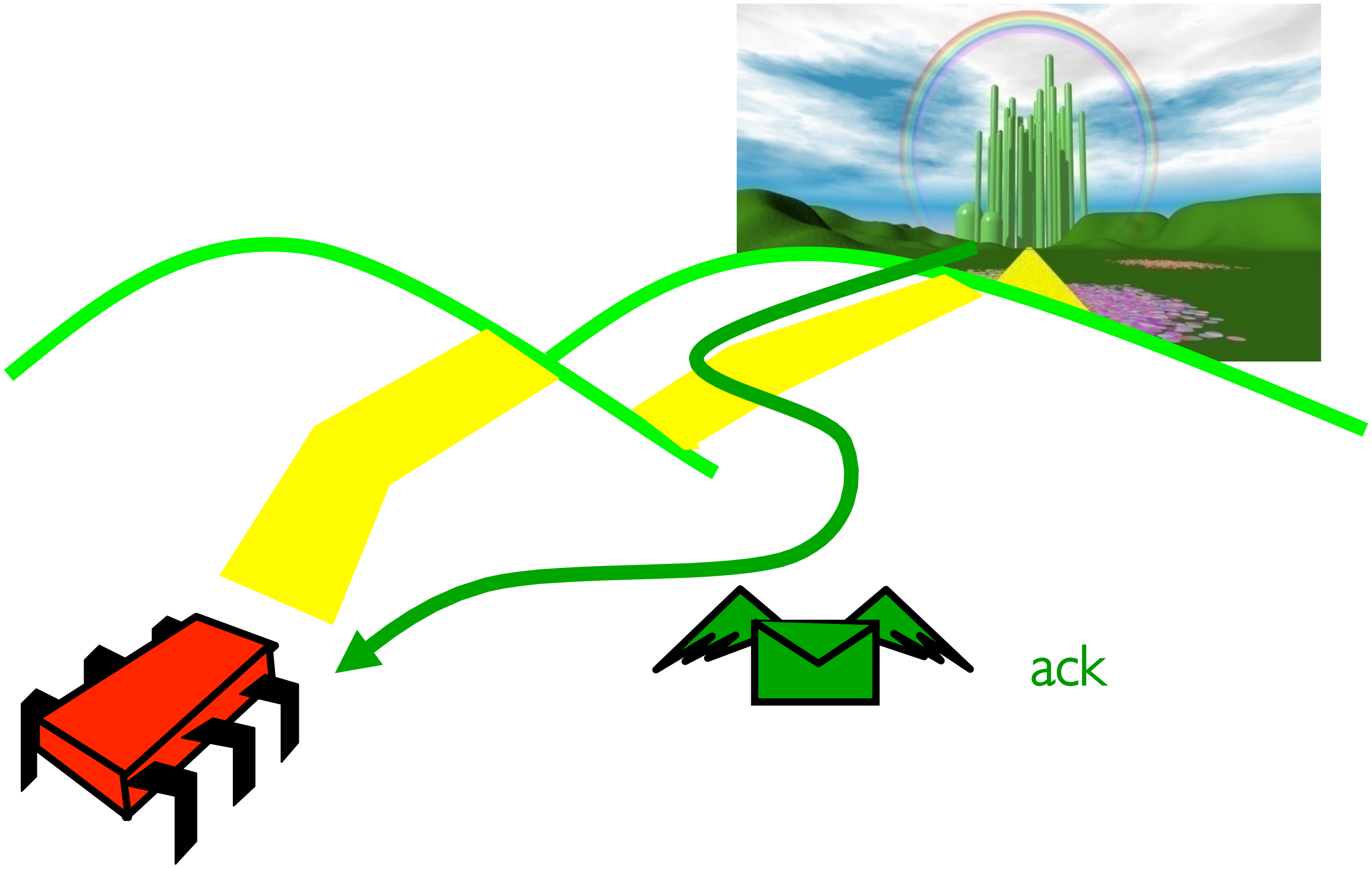
Writing to Memory



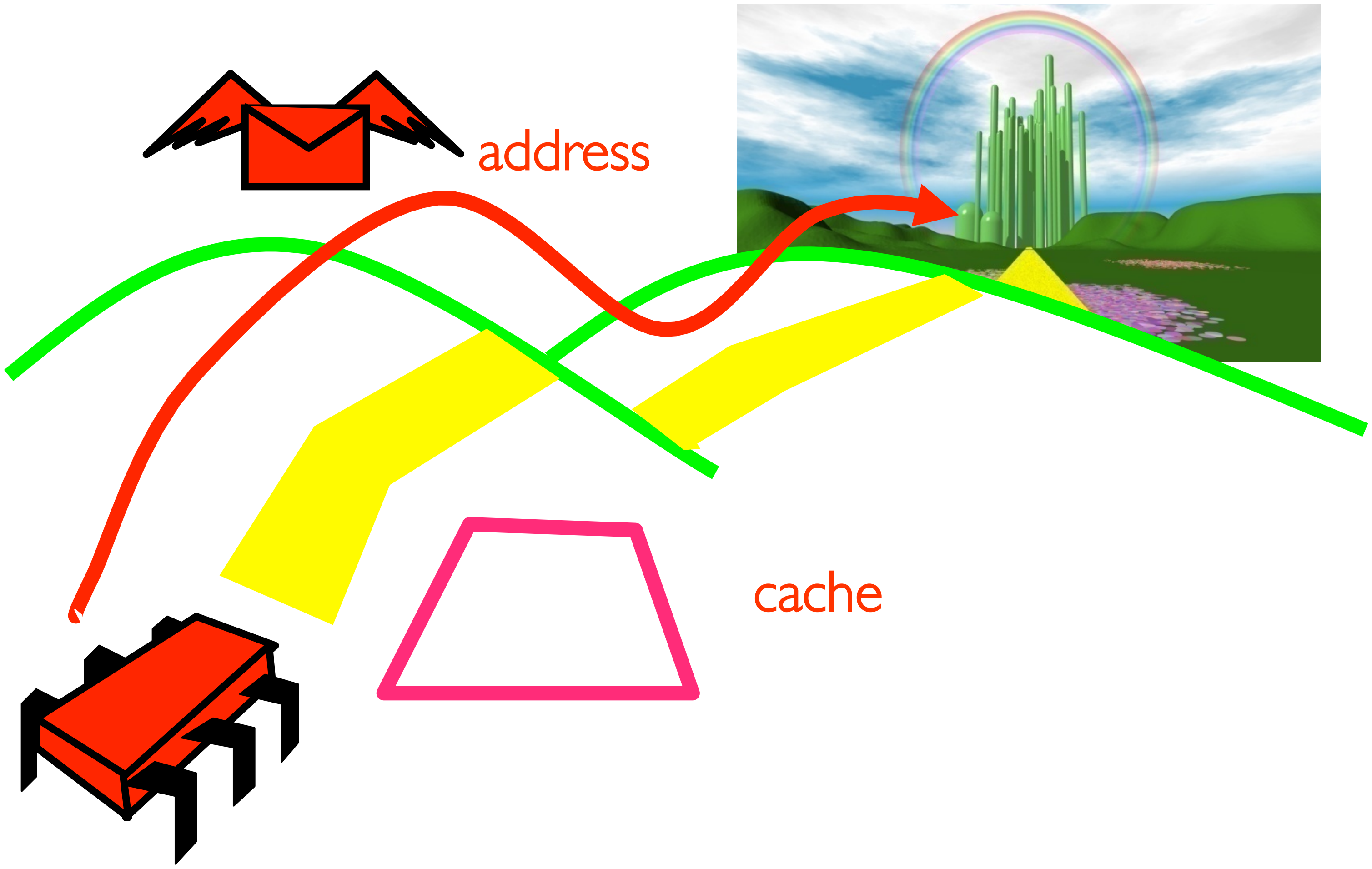
Writing to Memory



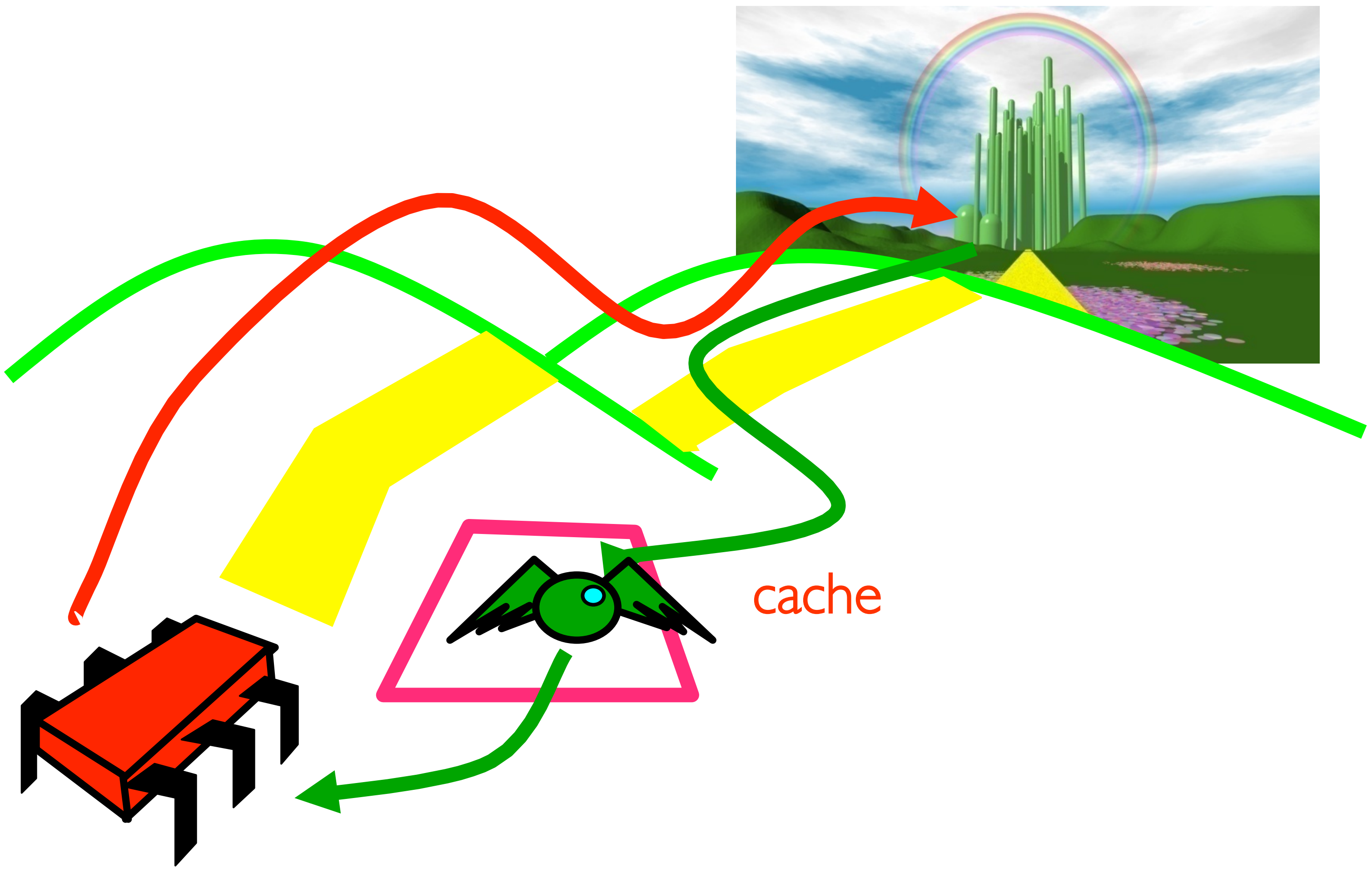
Writing to Memory



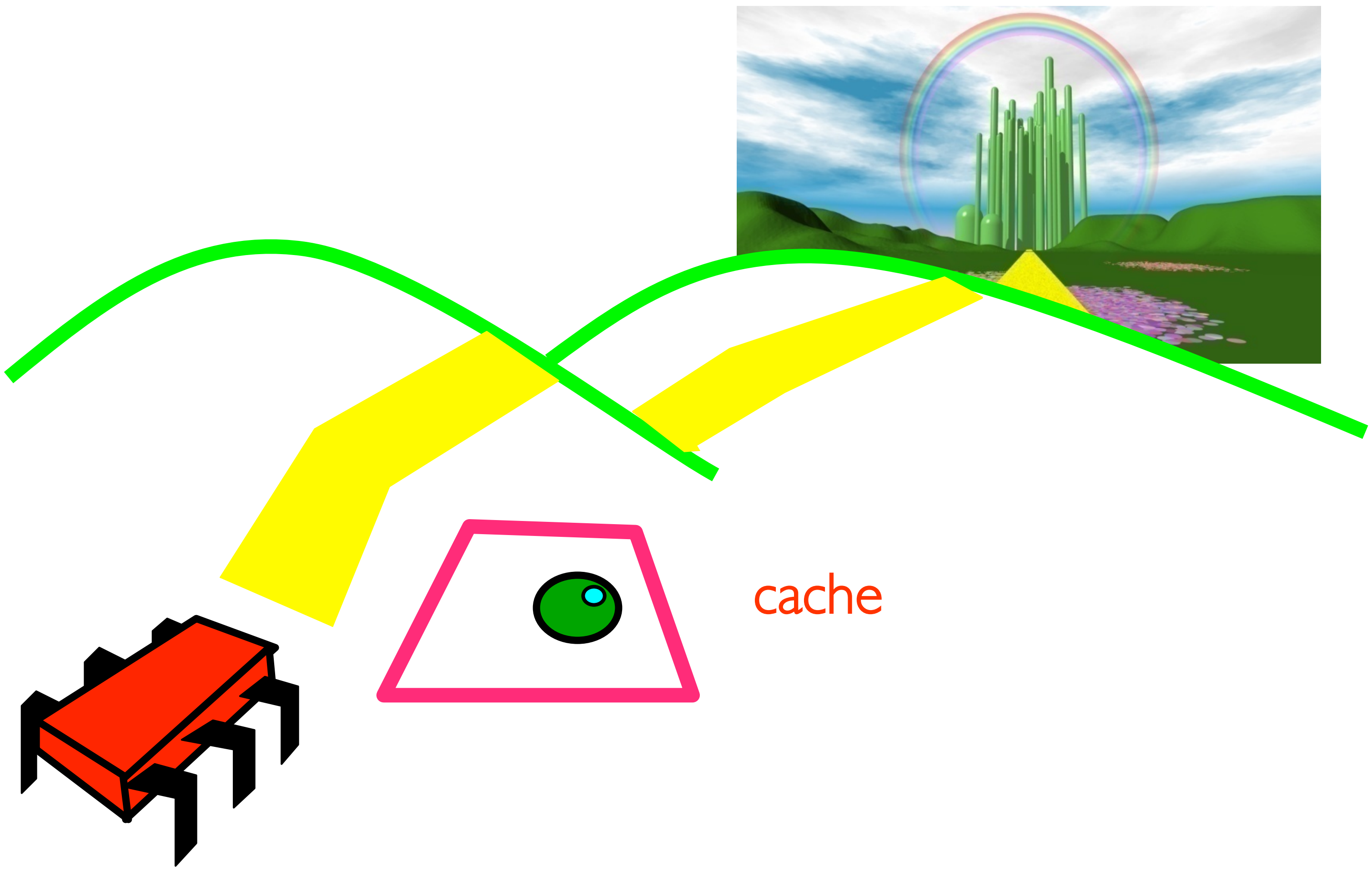
Cache: Reading from Memory



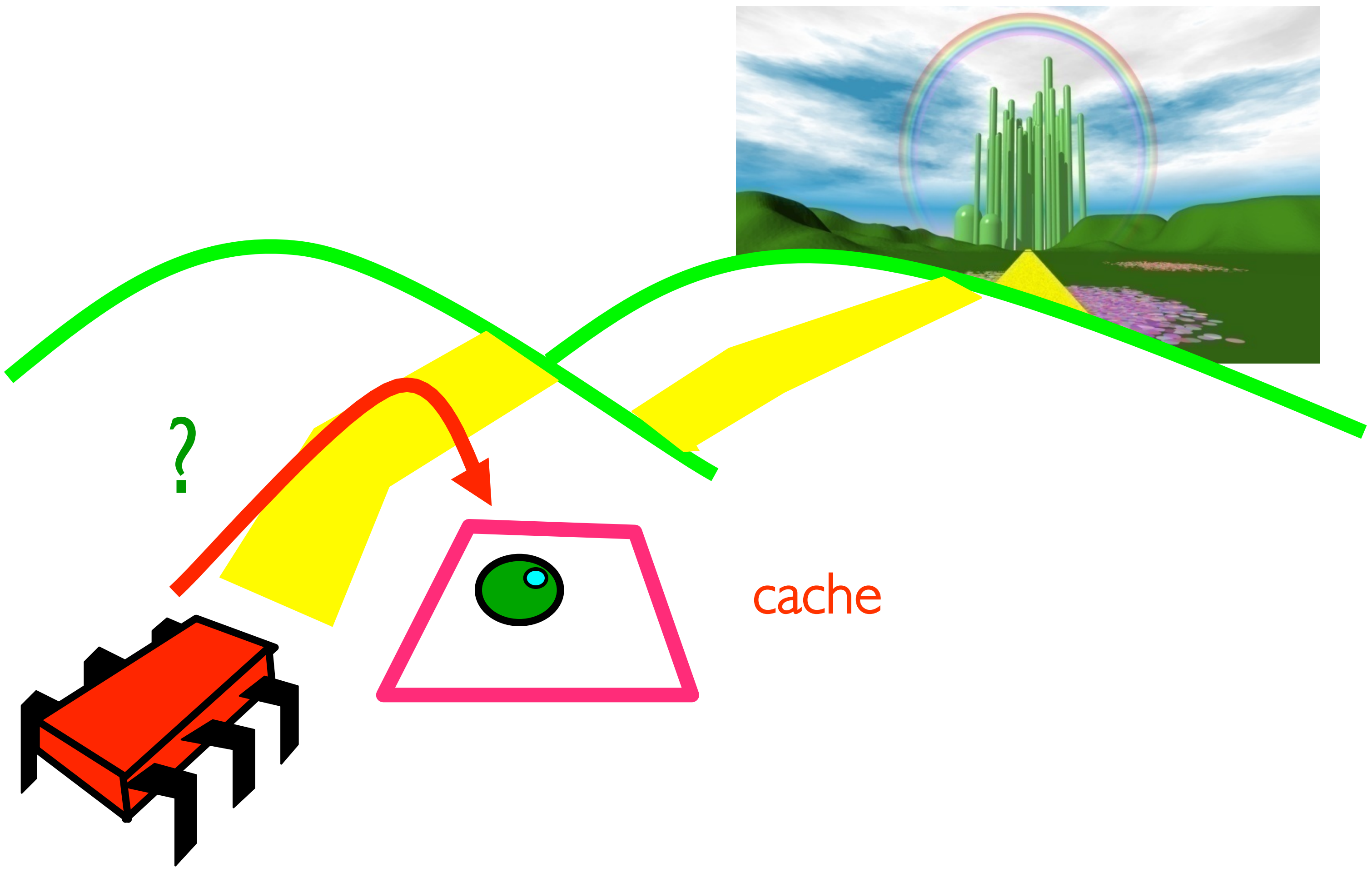
Cache: Reading from Memory



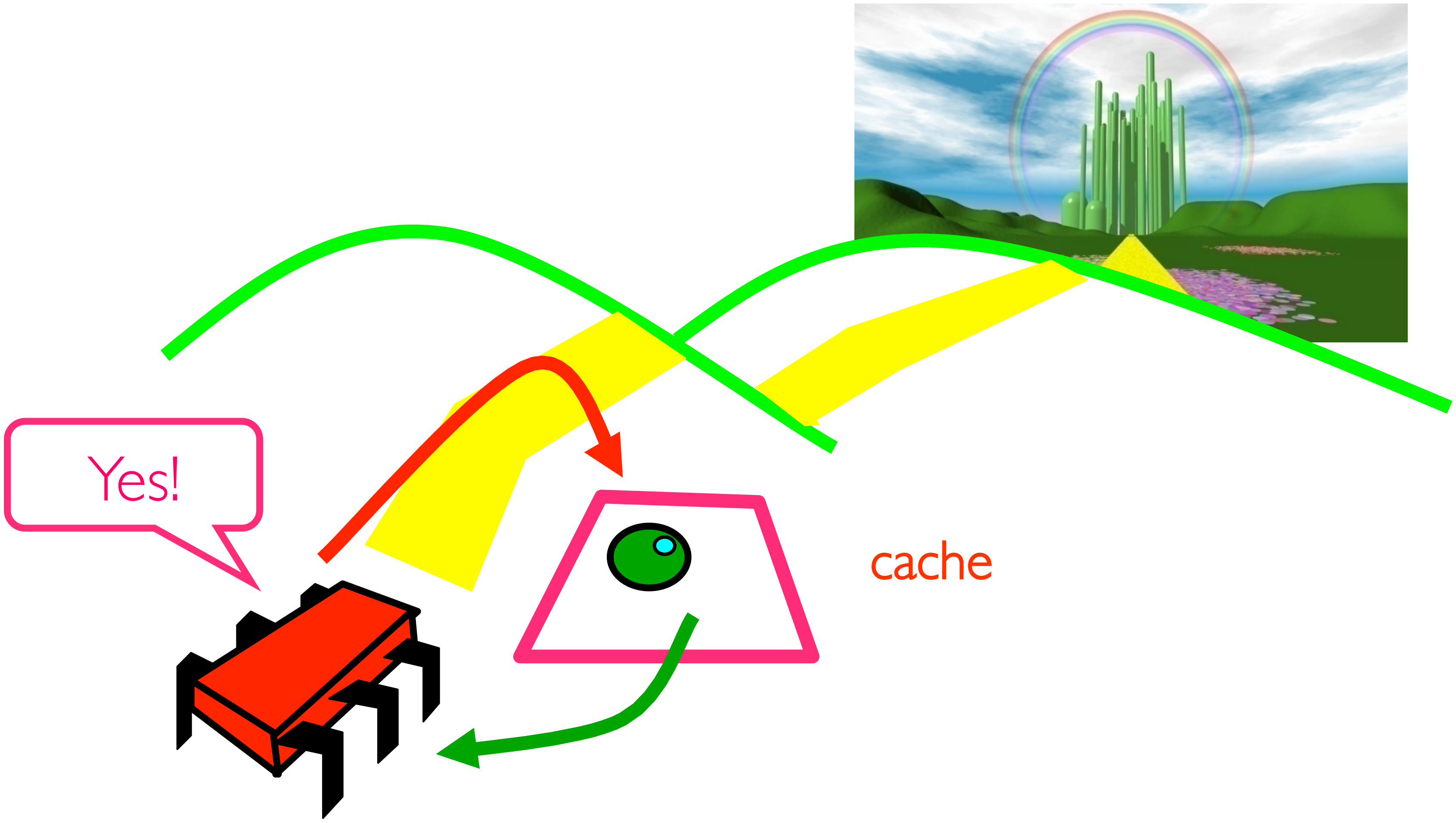
Cache: Reading from Memory



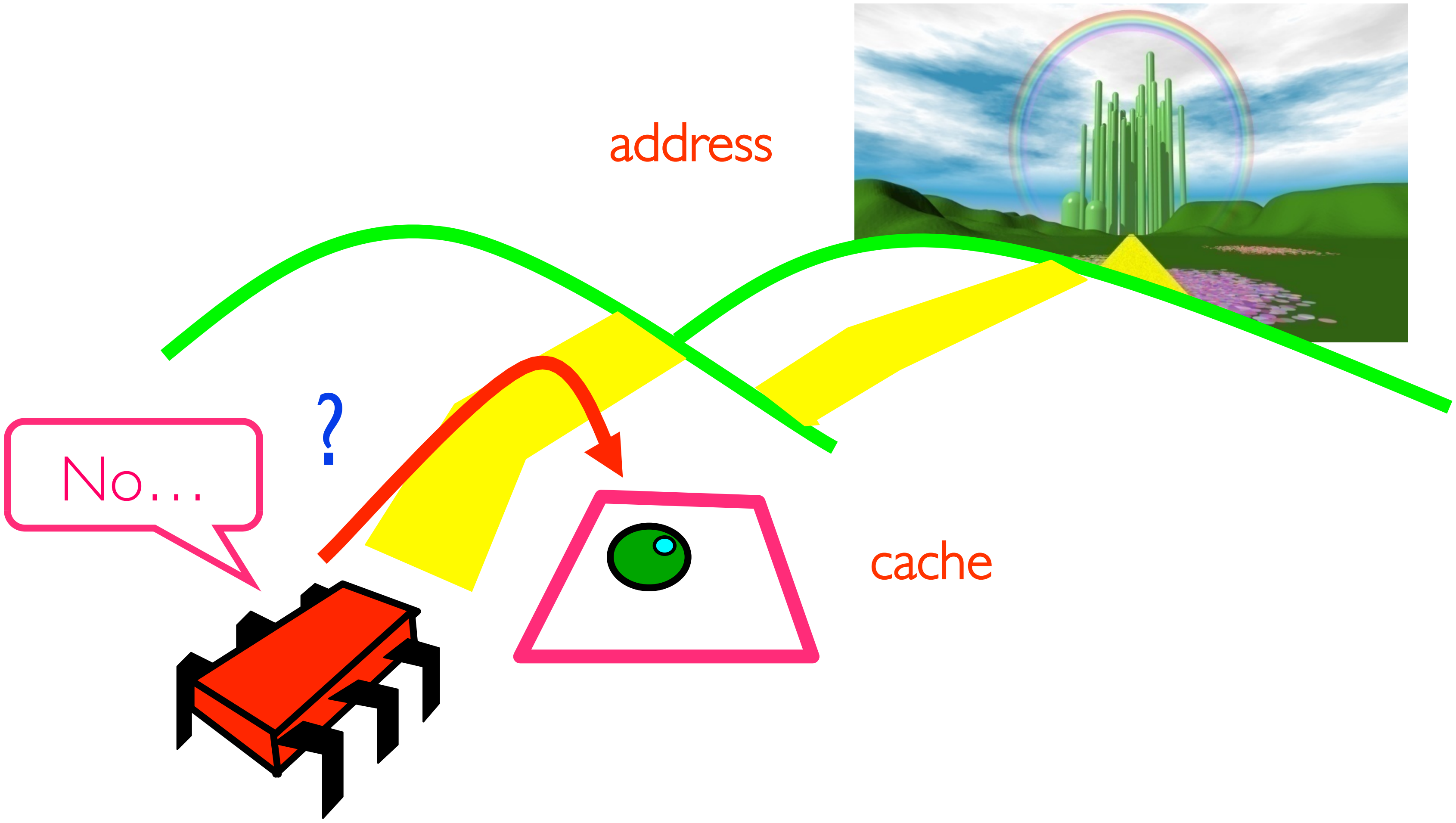
Cache Hit



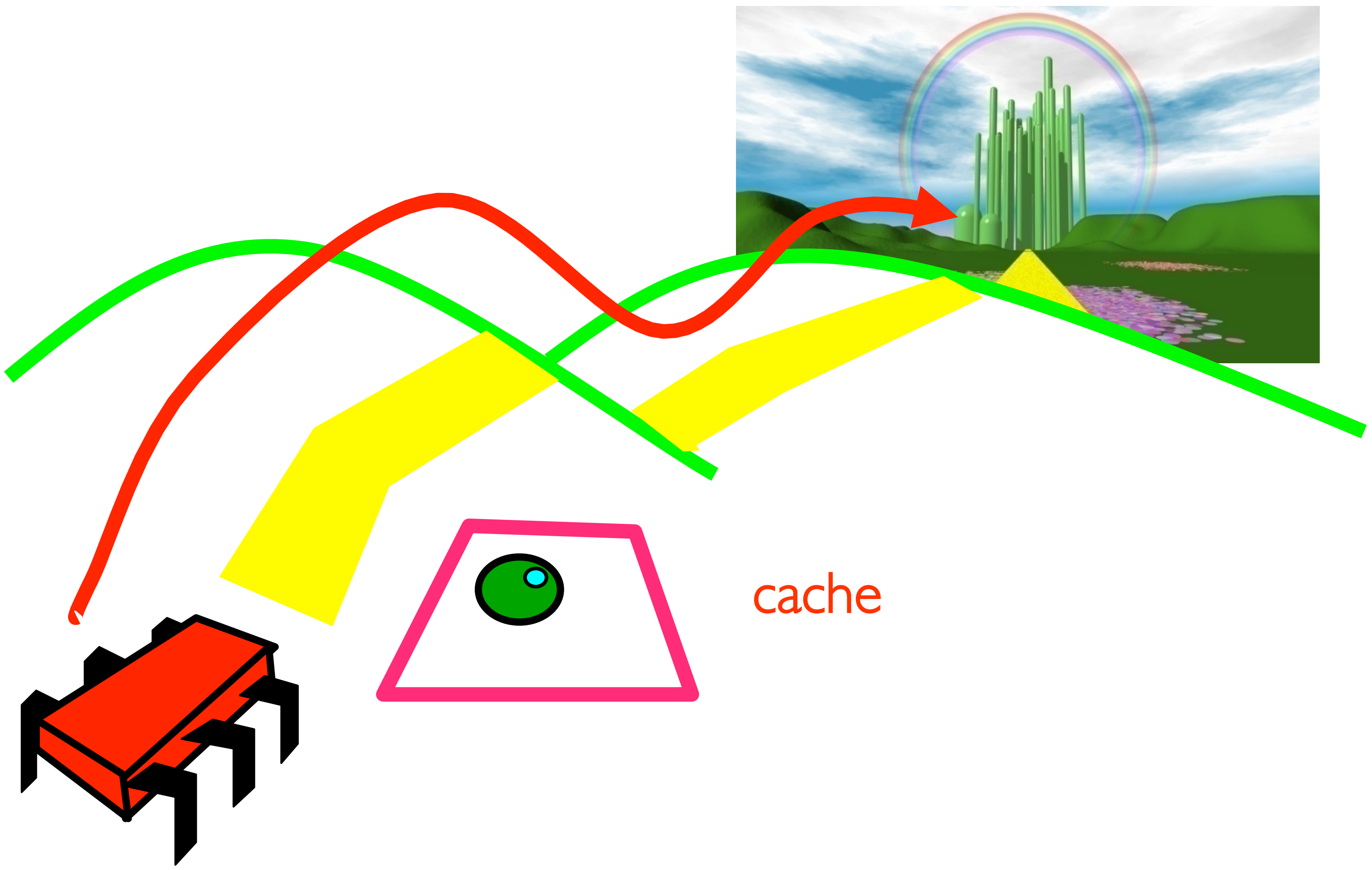
Cache Hit



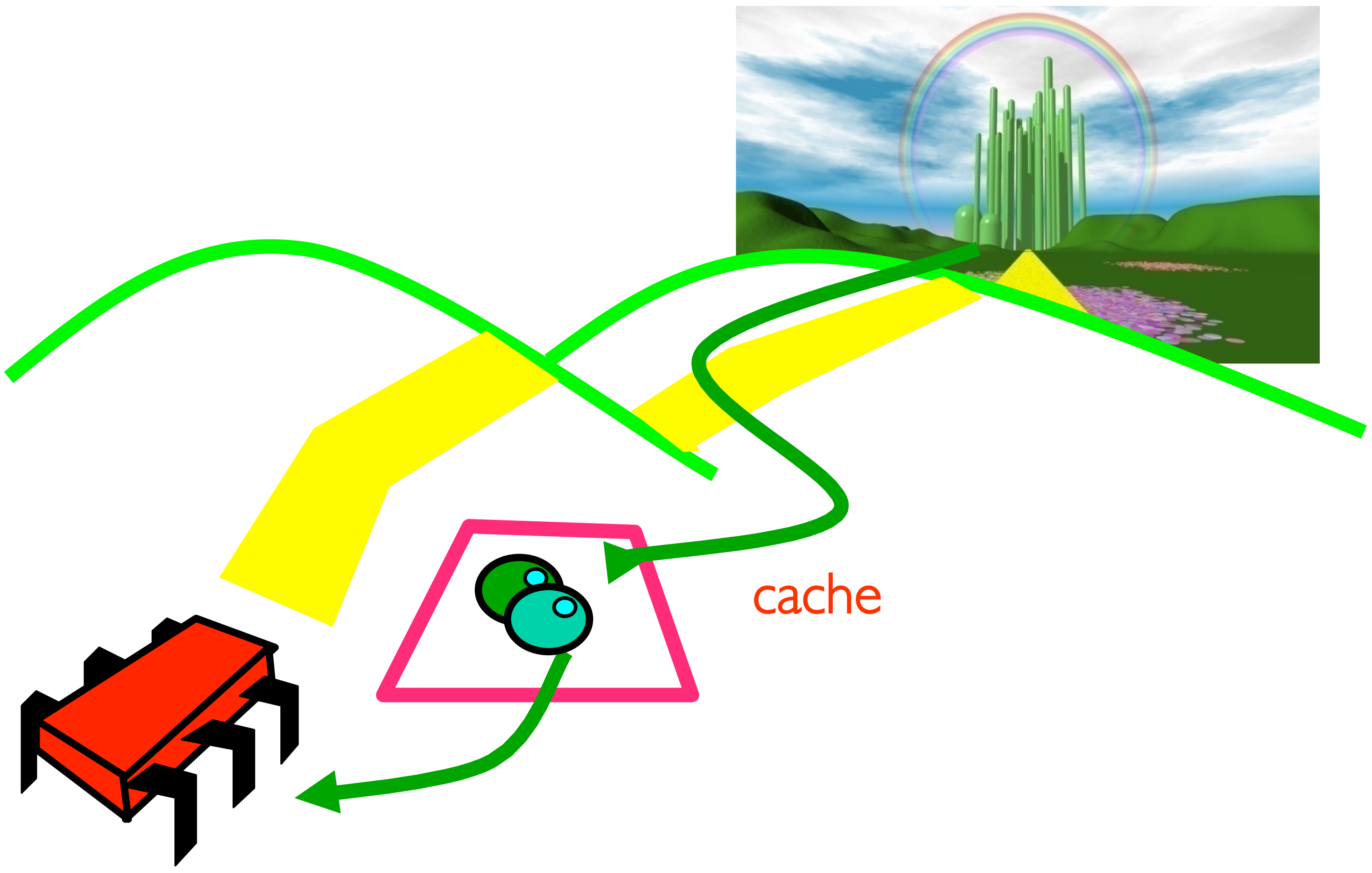
Cache Miss



Cache Miss



Cache Miss



Sometimes you have to spin for shared data….

- E.g., variable "x" is shared. The initial of value "x" is 0. Thread 1 will write a non-zero value. Thread 2 is "spins" waiting for Thread 1 to write the value

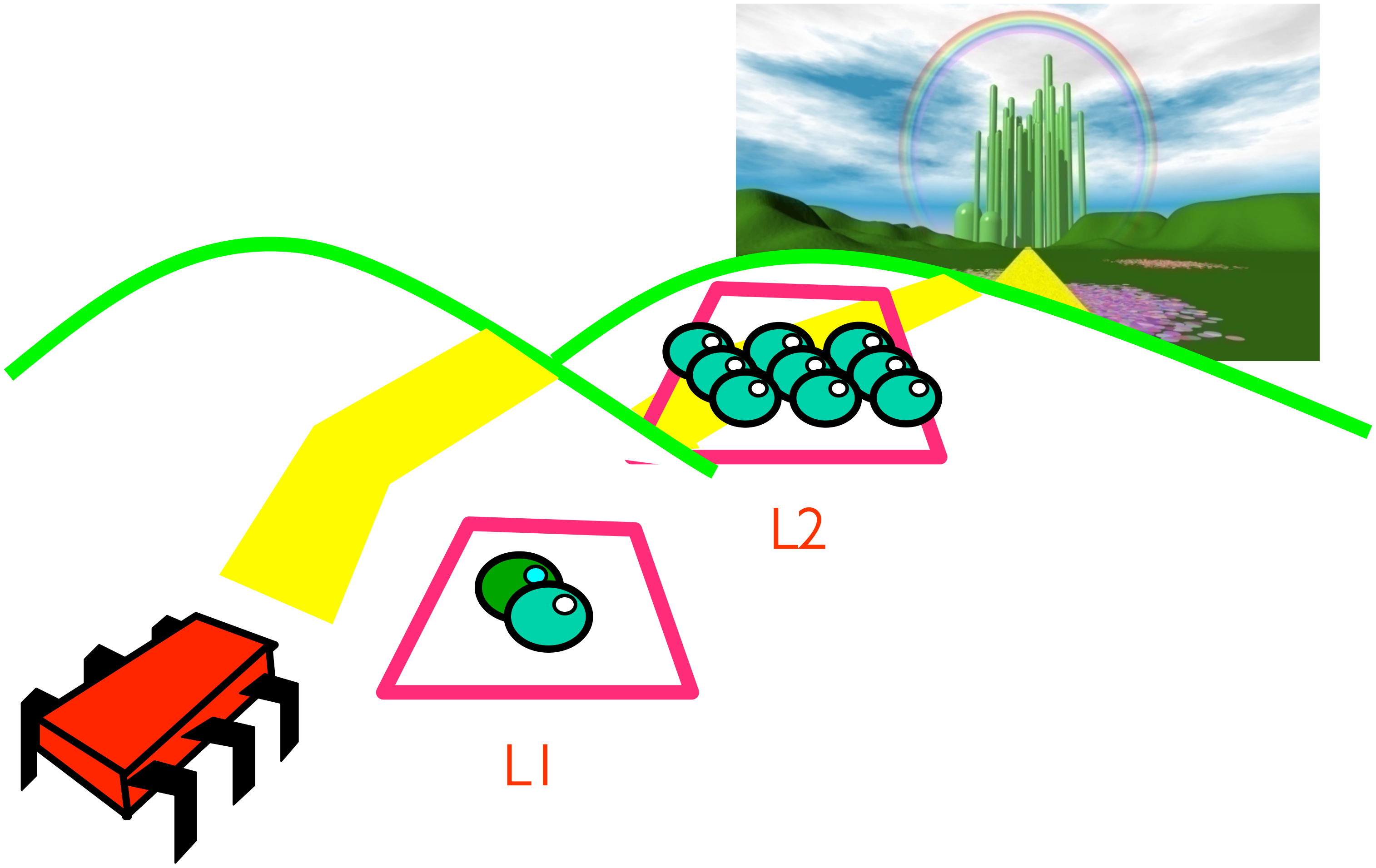
Thread 1	Thread 2
…..	while (x == 0) { /* wait */ }
x = 3;	….. = x;

Thread 2 keeps reading "x", hitting in the cache

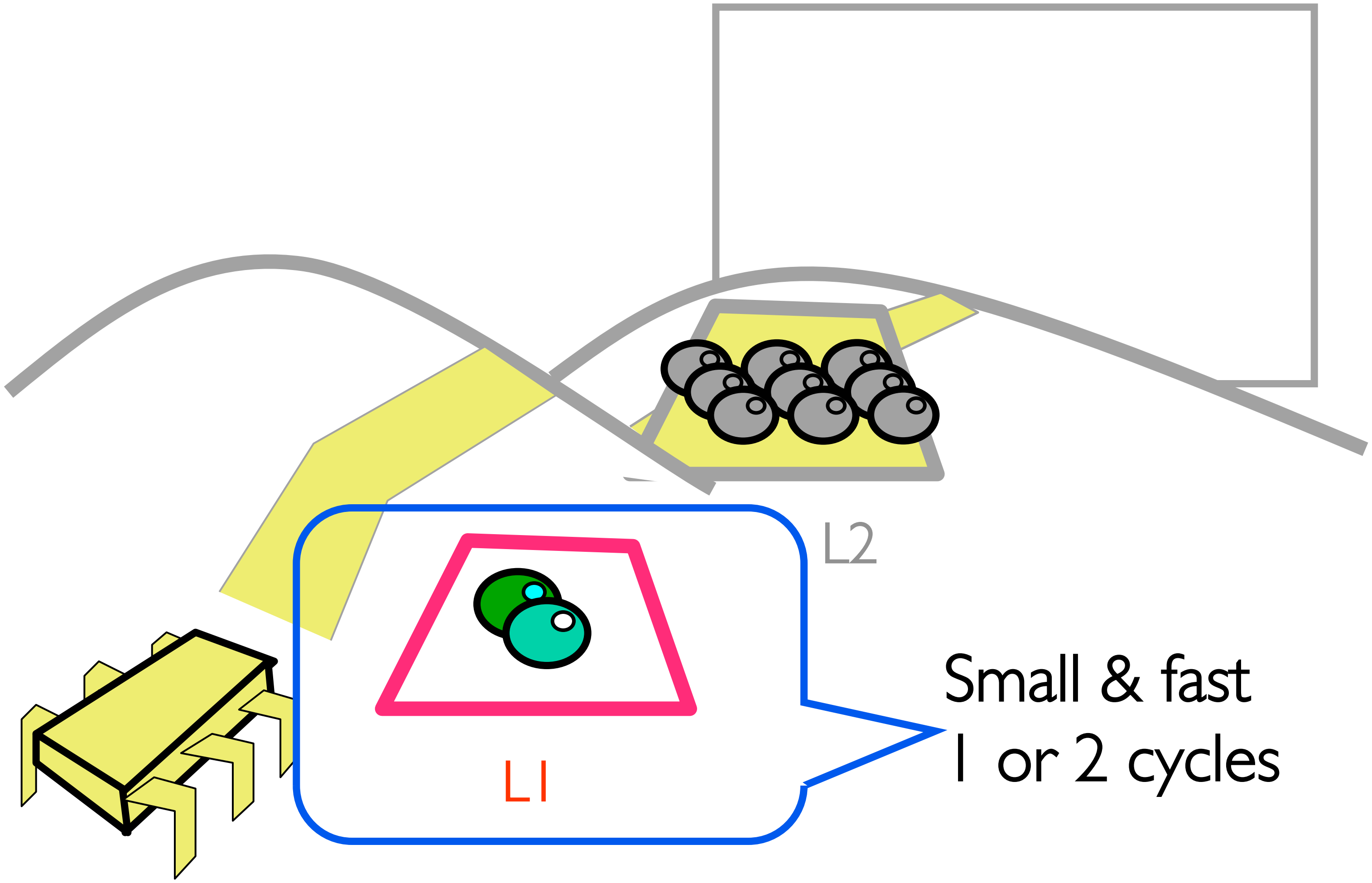
Local Spinning

- With caches, spinning becomes practical
- First time
  - Load variable into cache
- As long as it doesn't change
  - Hit in cache (no interconnect used)
- When it changes
  - One-time cost
  - See cache coherence (later)

L1 and L2 Caches



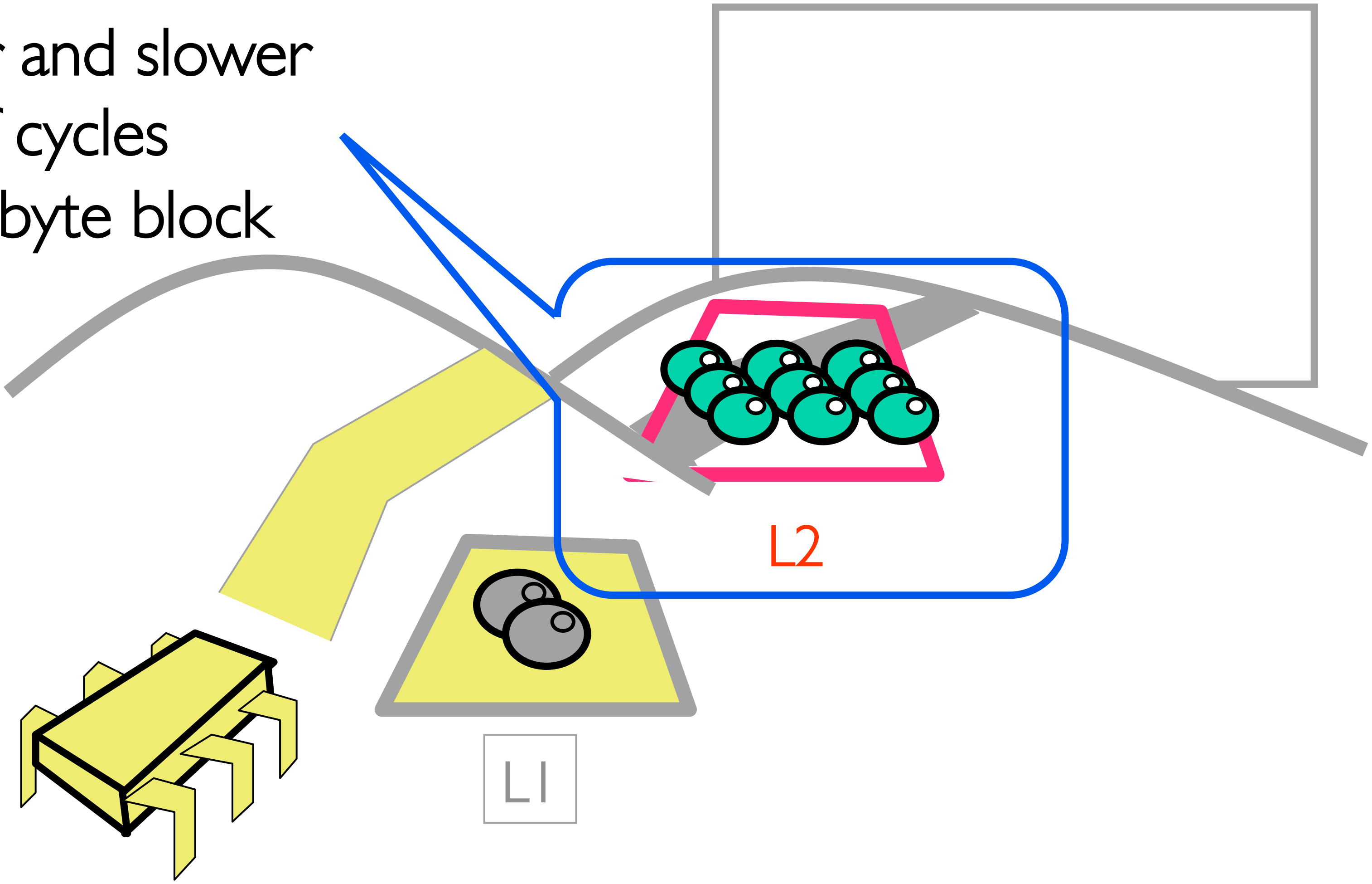
L1 and L2 Caches



Small & fast
1 or 2 cycles

L1 and L2 Caches

Larger and slower
10s of cycles
~128 byte block

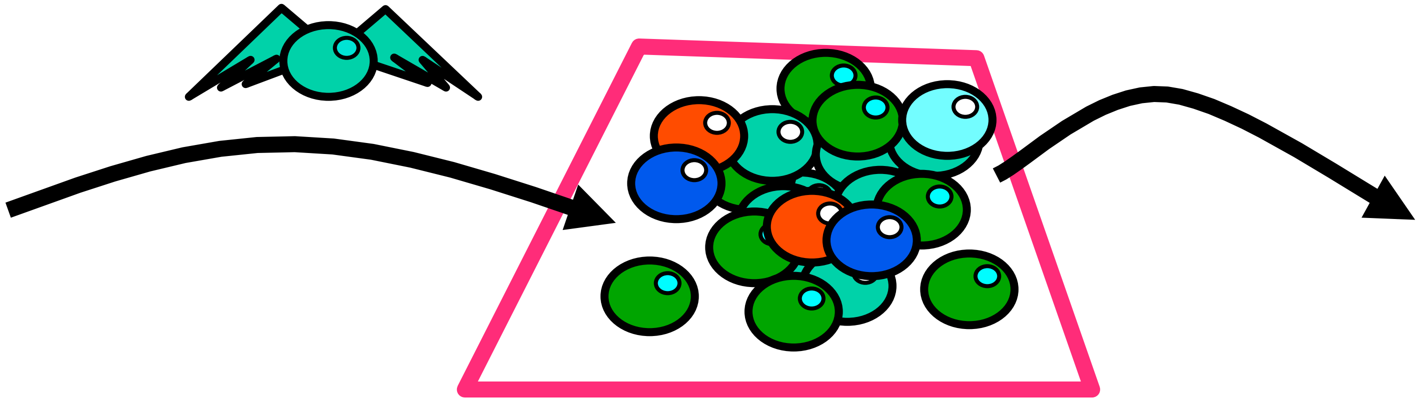


When a Cache Becomes Full…

- Need to make room for new entry
- By evicting an existing entry
- Need a replacement policy
  - Usually some kind of least recently used heuristic

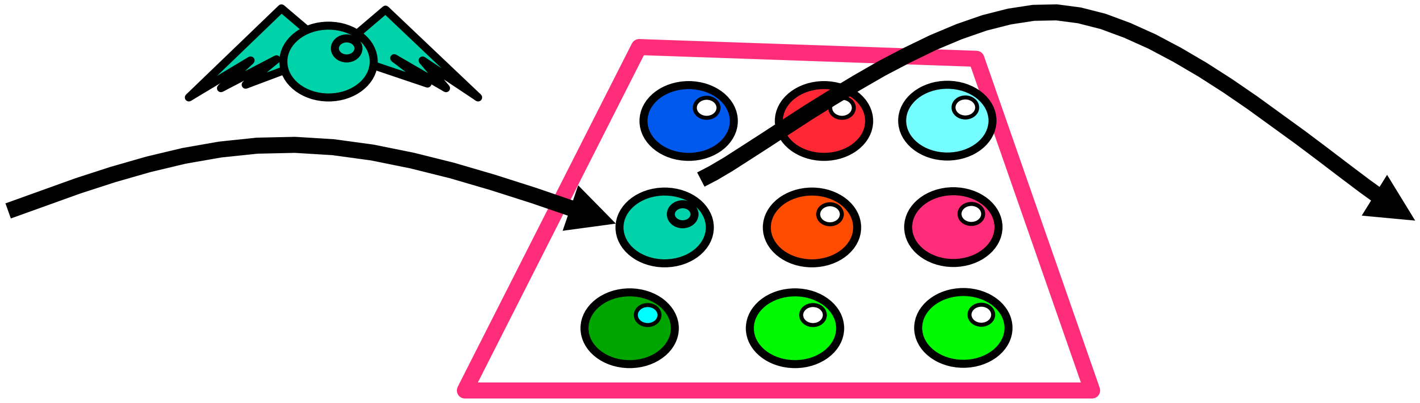
Fully Associative Cache

- Any line can be anywhere in the cache
  - Advantage: can replace any line
  - Disadvantage: hard to find lines



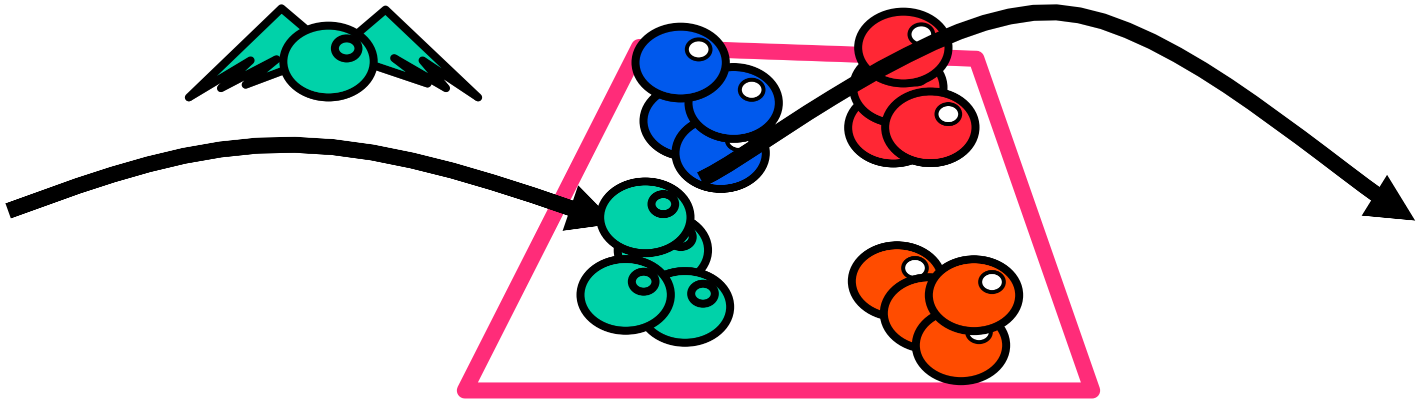
Direct Mapped Cache

- Every address has exactly 1 slot
  - Advantage: easy to find a line
  - Disadvantage: must replace fixed line



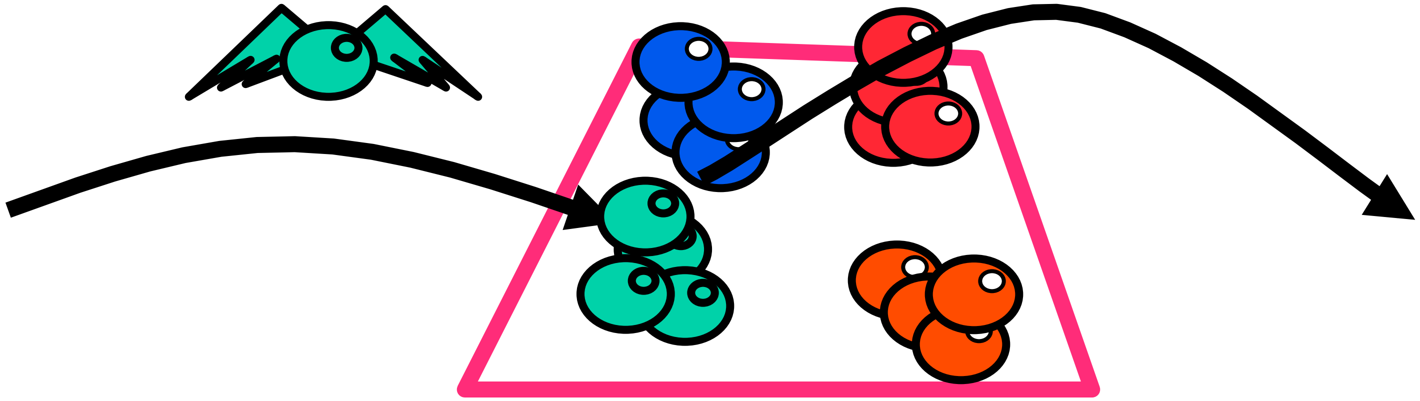
K-way Set Associative Cache

- Each slot holds k lines
  - Advantage: pretty easy to find a line
  - Advantage: some choice in replacing line



Multicore Set Associativity

- L2, lower levels can be more associative (e.g., > 16 ways)
  - Why? Because cores share sets
  - Threads cut effective size if accessing different data



Example: Average Memory latency

w/o cache, all accesses go to memory:

Average Memory Access Time = memory latency

with one level cache:

Average Memory Access Time =

cache hit time x (1 – cache miss rate) +

memory latency x cache miss rate

- With a cache has a miss rate of 10%, hit time of 2 cycles and a memory latency off 100 cycles, what is AMAT?

Example: Average Memory latency

Assume the following params:

L1 hit time = 2 cycles, L1 miss rate = 5%

L2 hit time = 10 cycles, L2 miss rate = 2%

Memory latency = 200 cycles

- What is AMAT?

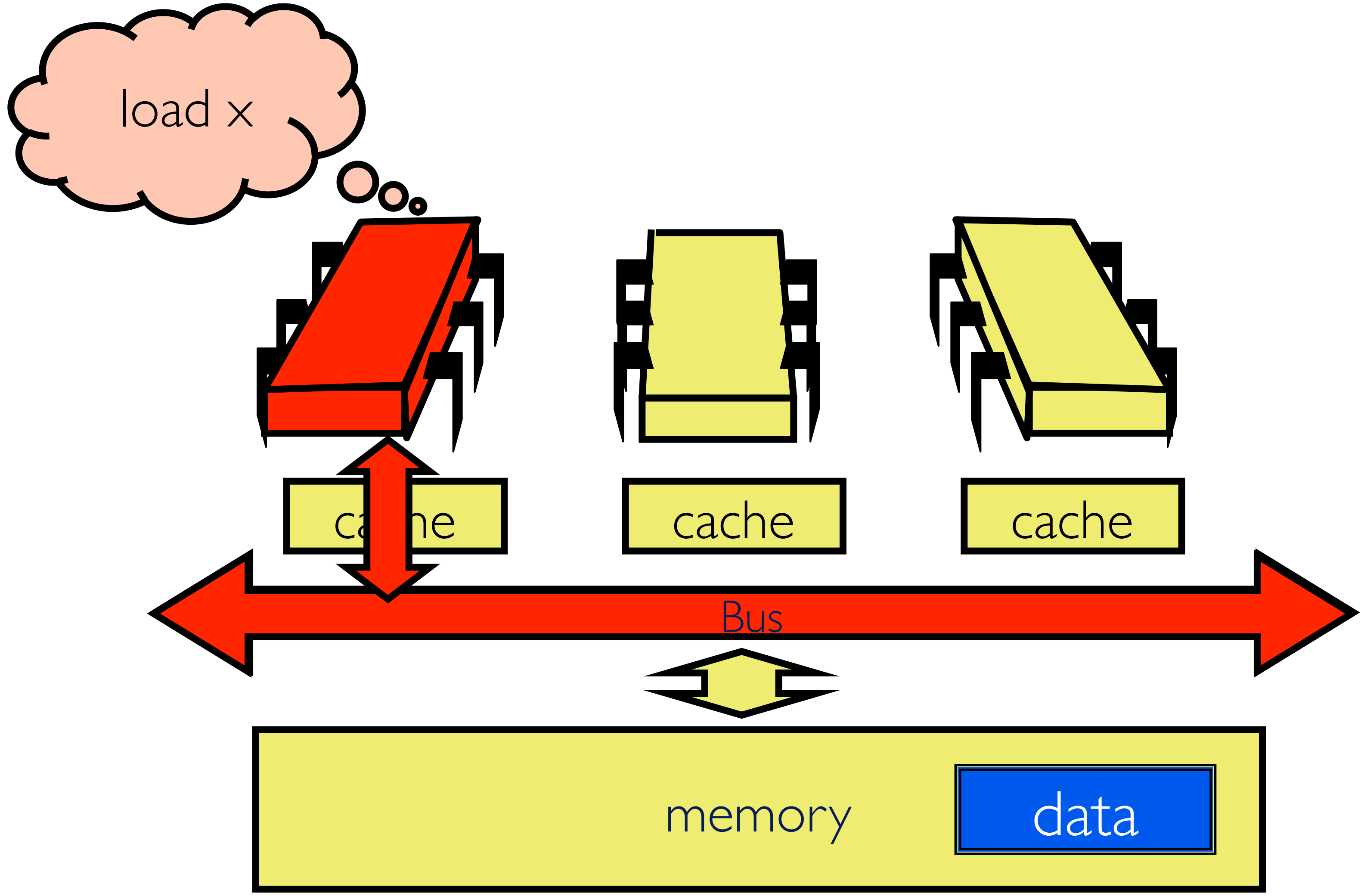
Sources of Cache Misses

- ▶From CS-208
  - ▷3 C's: Compulsory, conflict, capacity misses
- ▶4th type of miss:
  - ▷Coherence: reads/writes from multiple processors

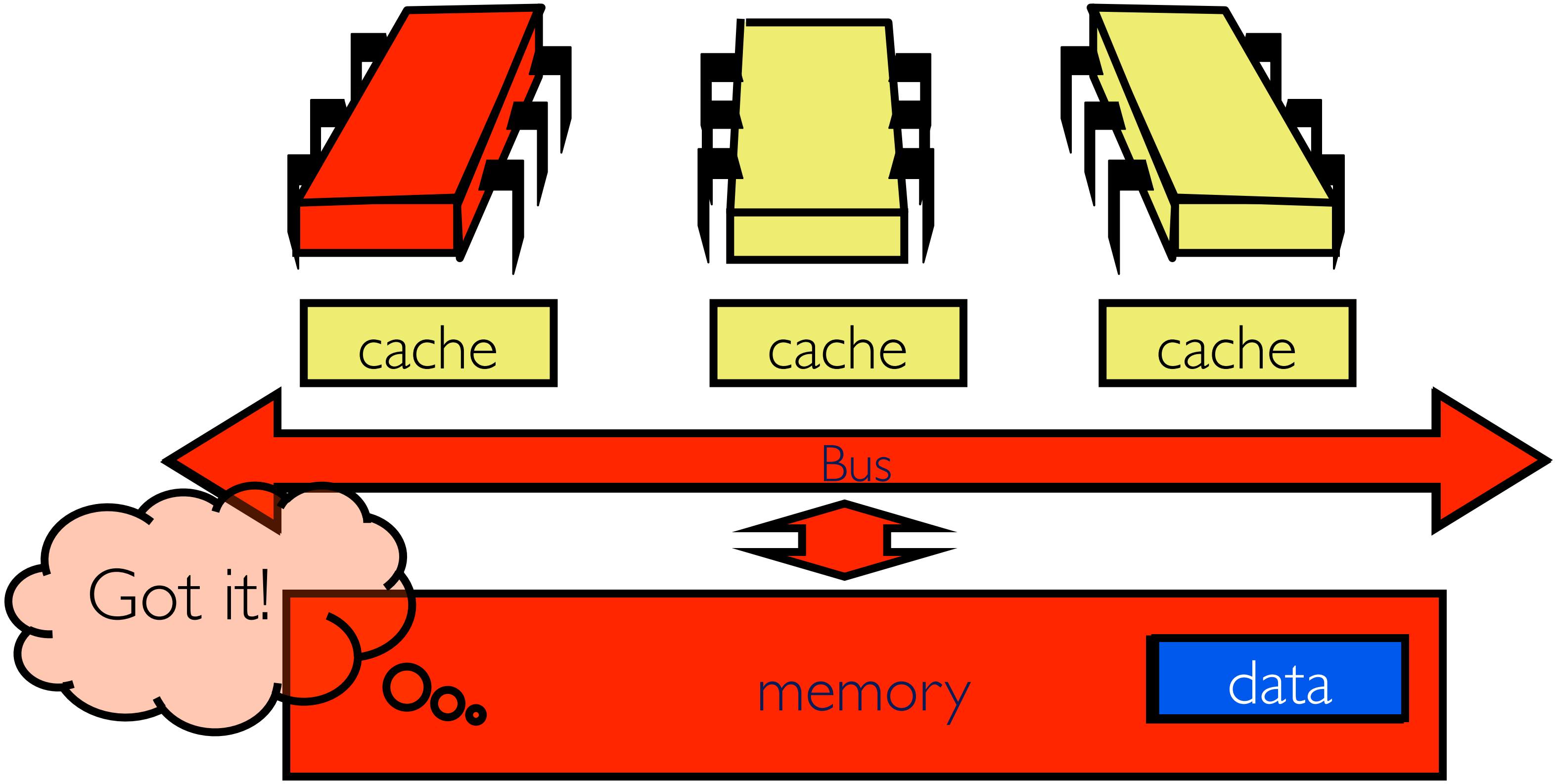
Cache Coherence

- A and B both cache address x
- A writes to x
  - Updates cache
- How does B find out?
- Many coherence protocols in products (will see in CS-370)

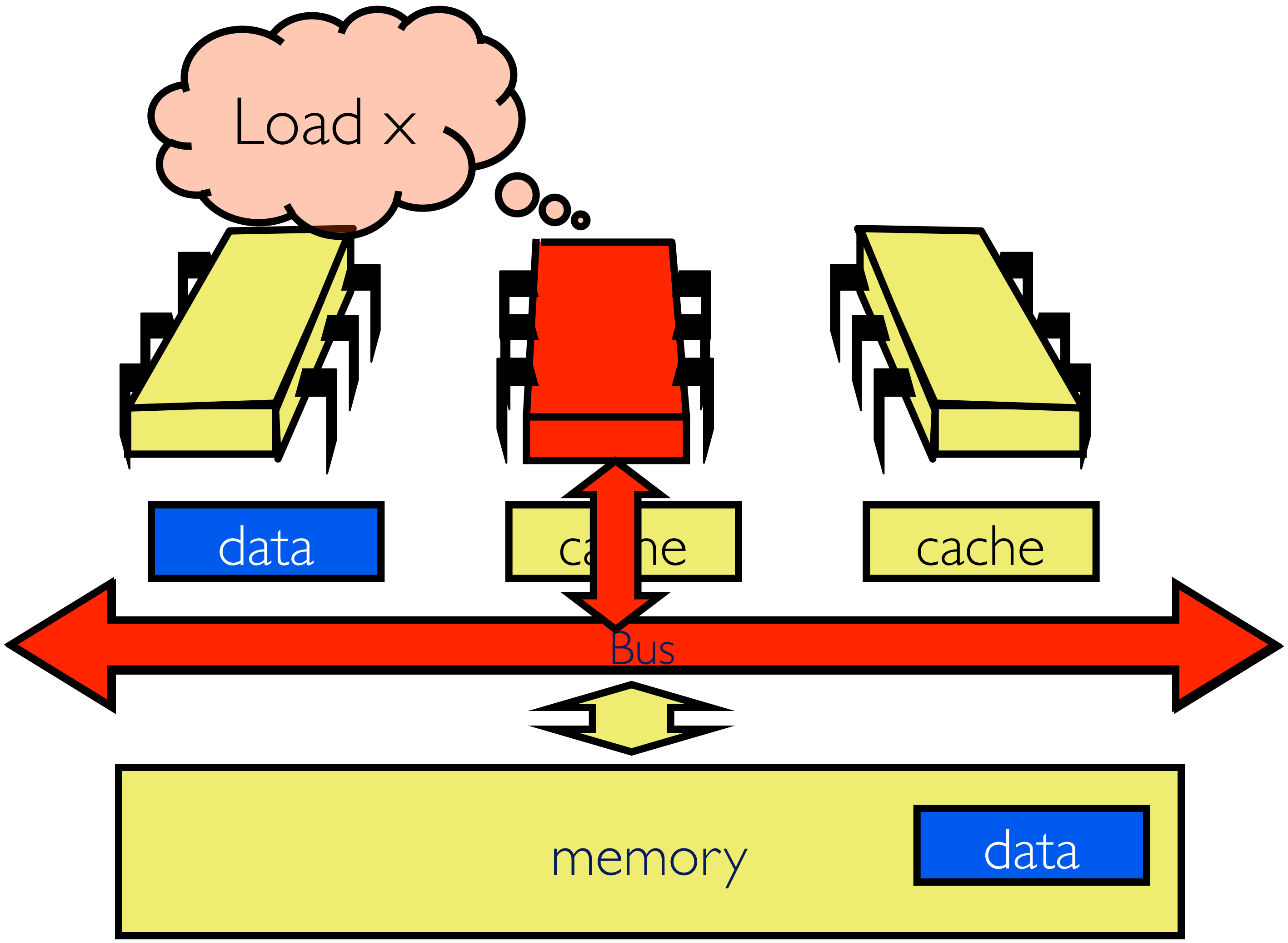
Processor Issues Load Request



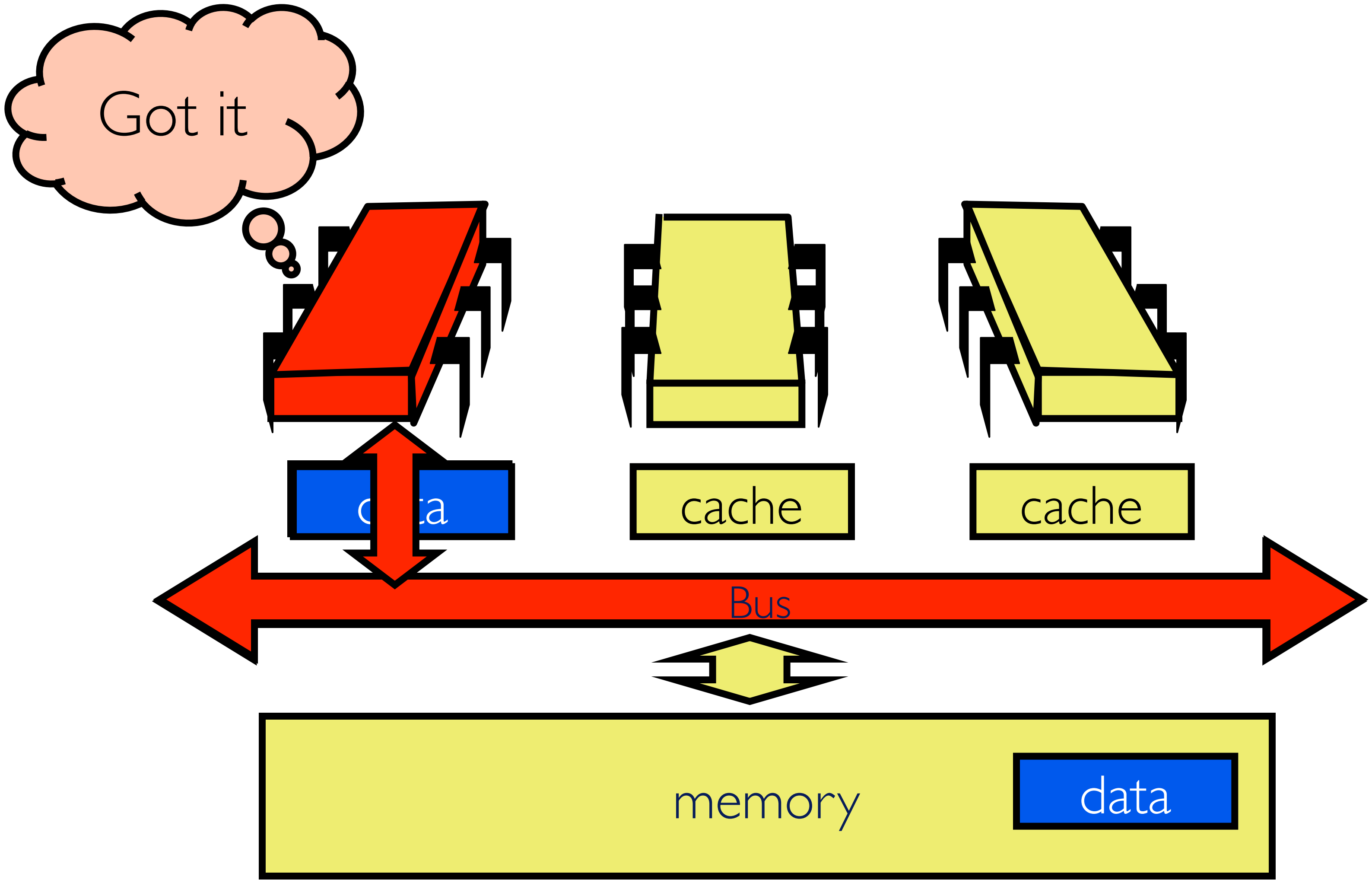
Memory Responds



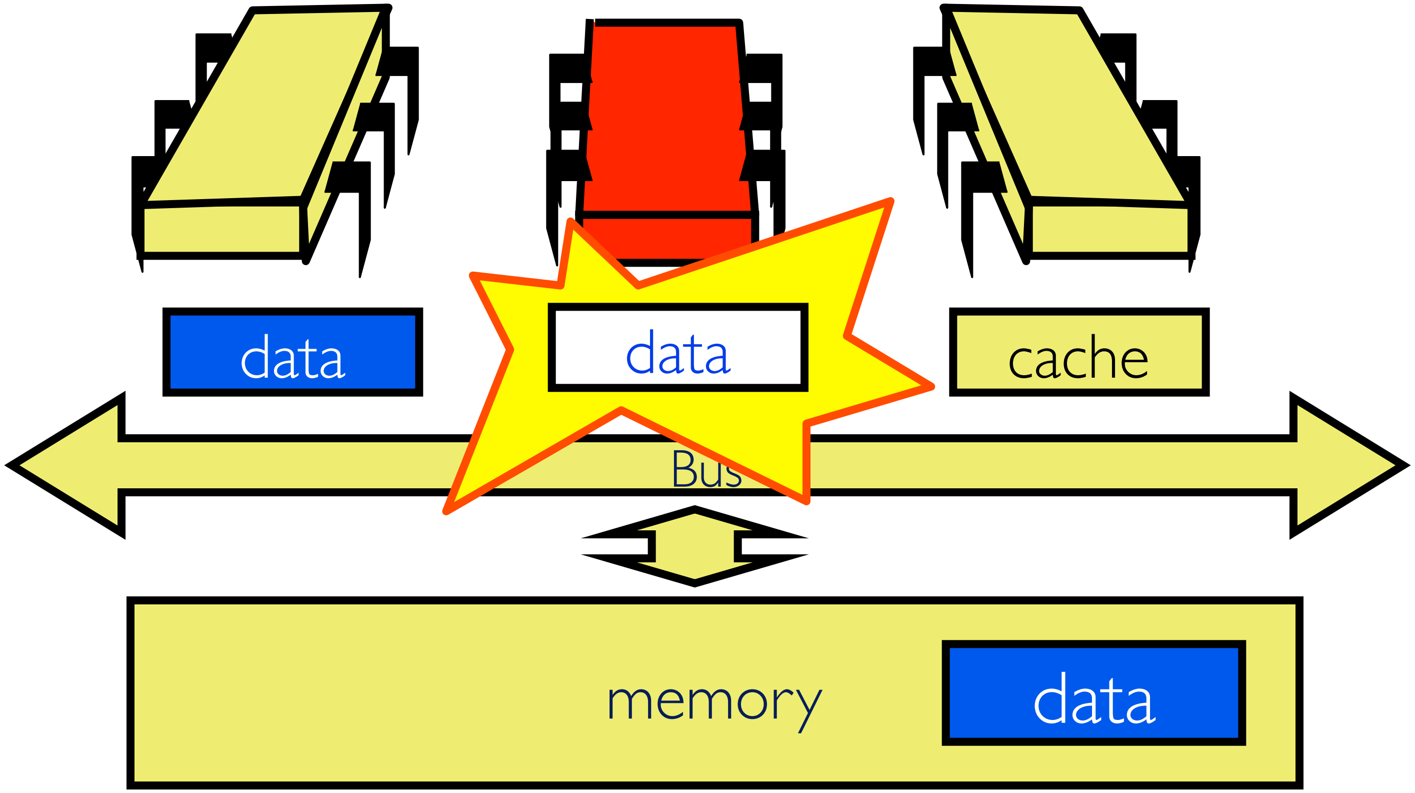
Processor Issues Load Request



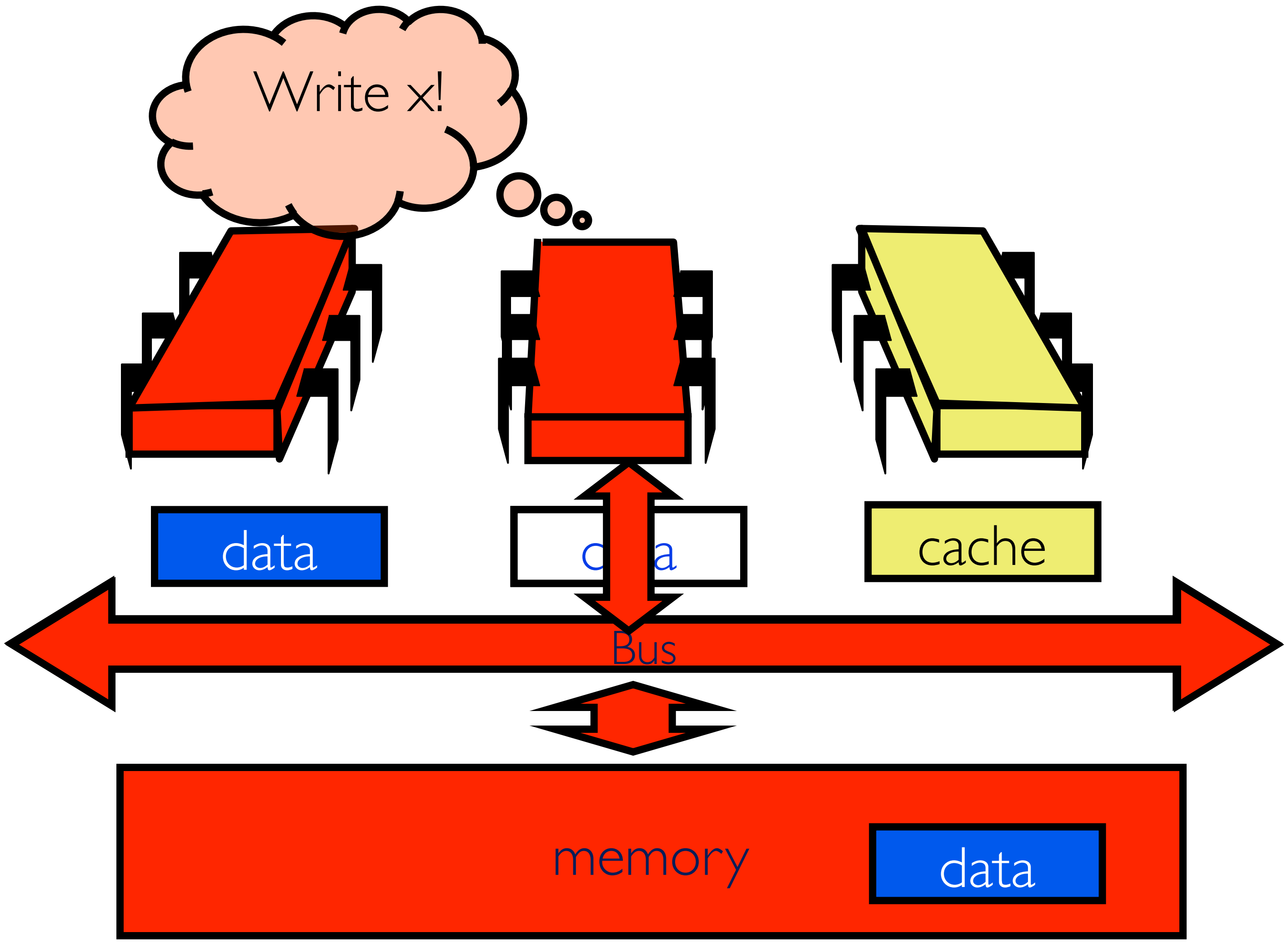
Other Processor Responds



Modify Cached Data



Write-Through Cache



Write-Through Caches

- Immediately broadcast changes
  - Memory is up-to-date
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes …

Write-Through Caches

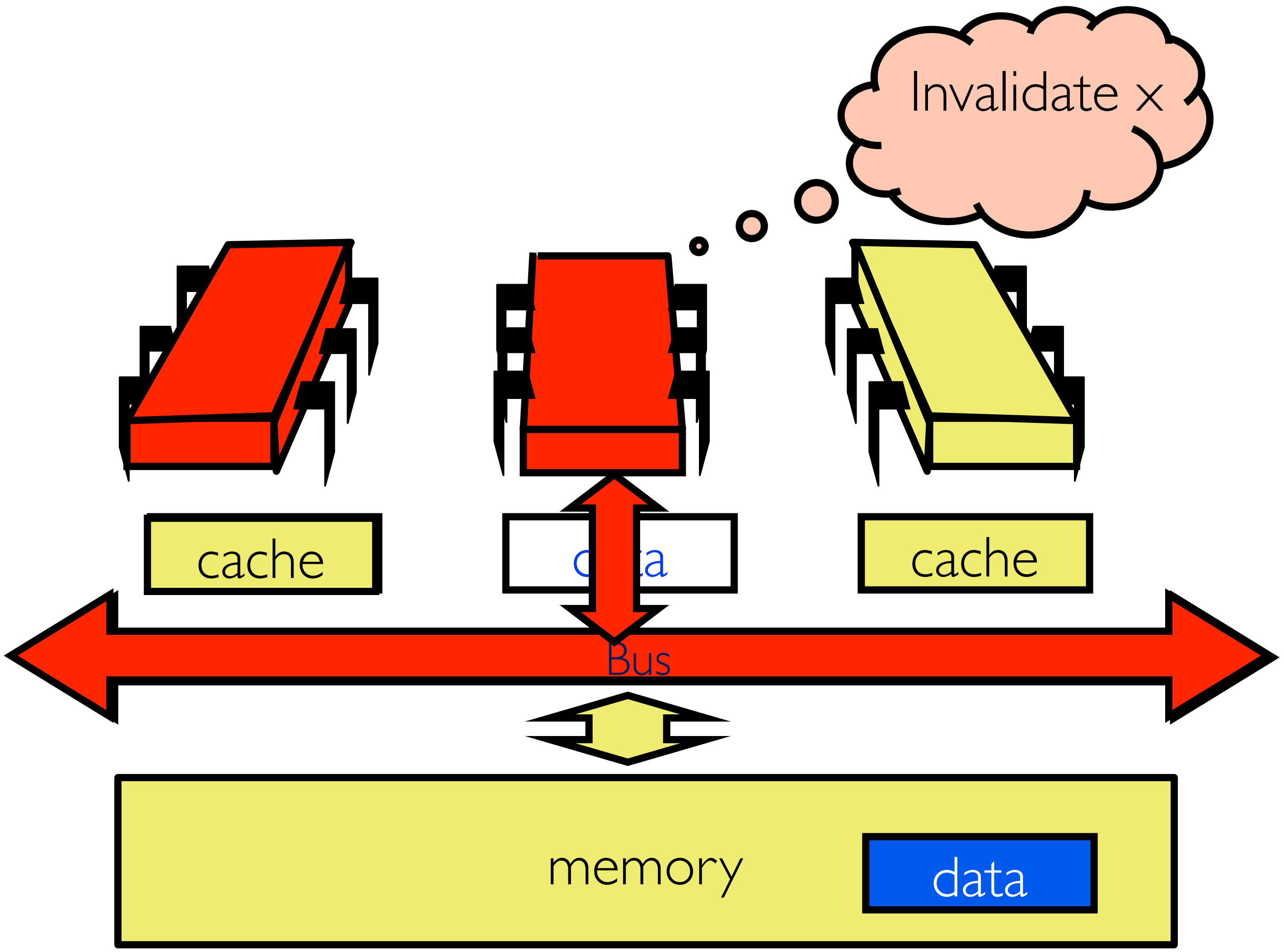
- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes …

"show stoppers"

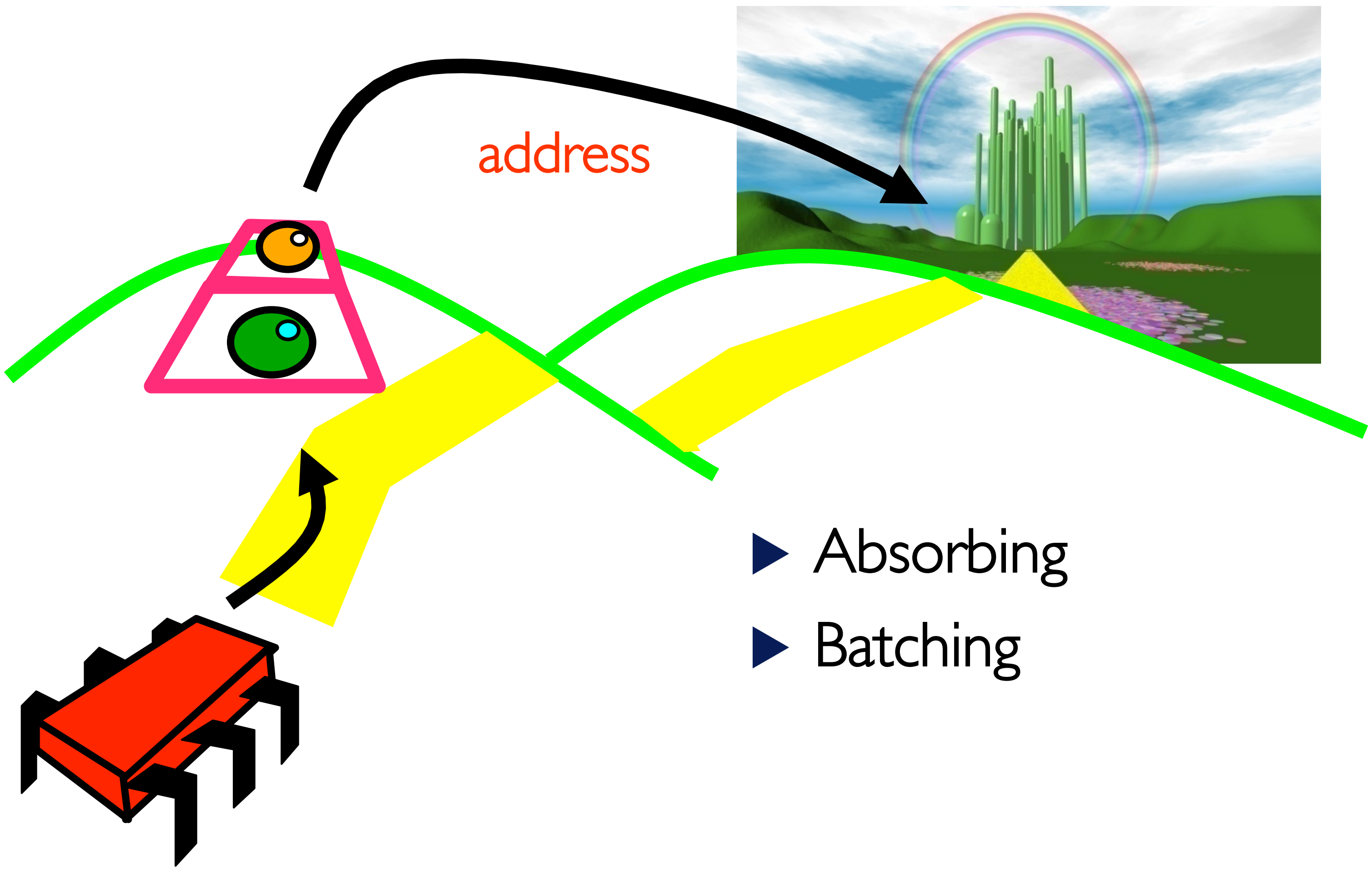
Write-Back Caches

- Accumulate changes in cache
- Write back when block evicted
  - Need the cache for something else
  - Another processor wants it

Invalidate



Write Buffers

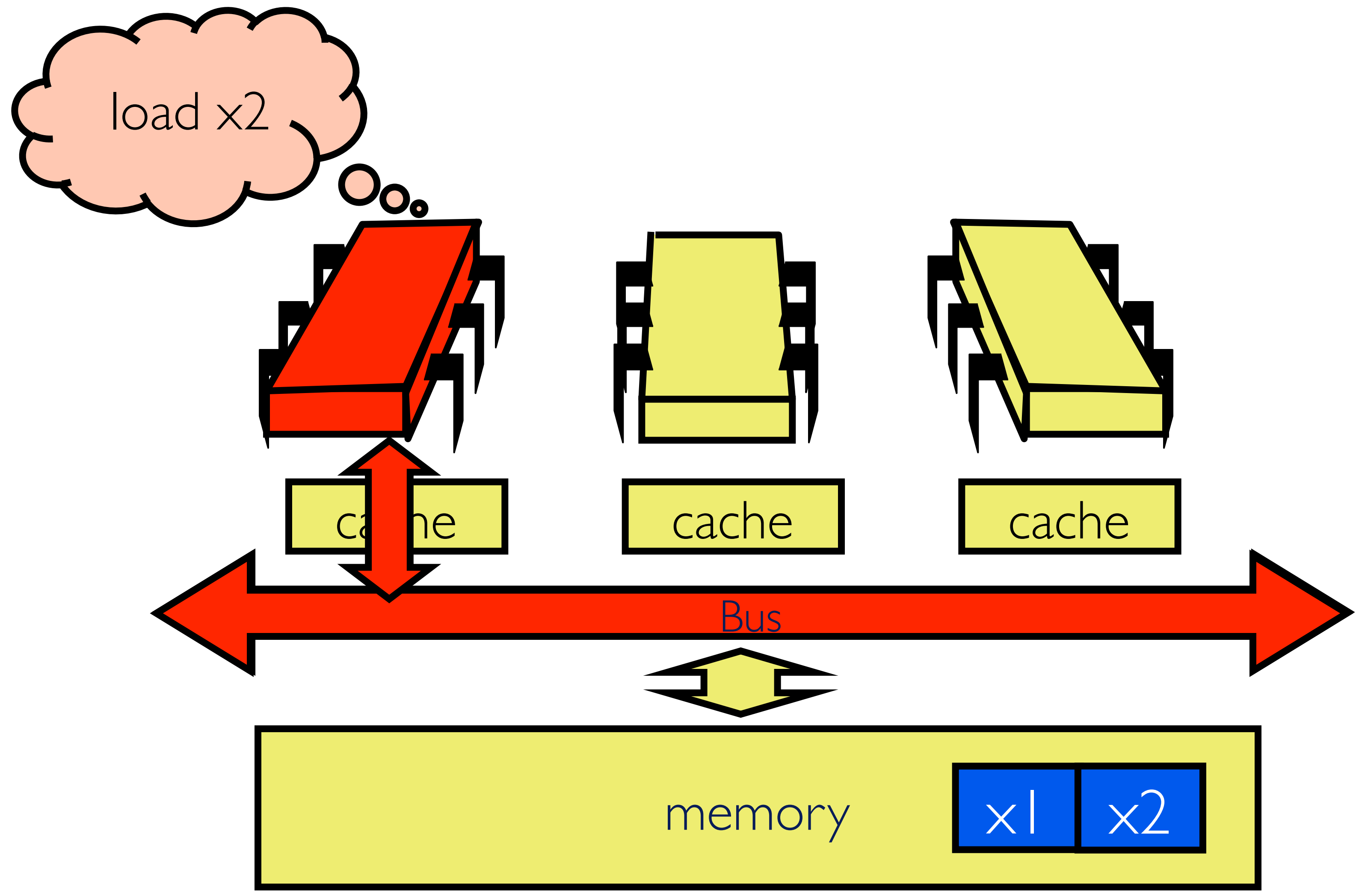


- Absorbing
- Batching

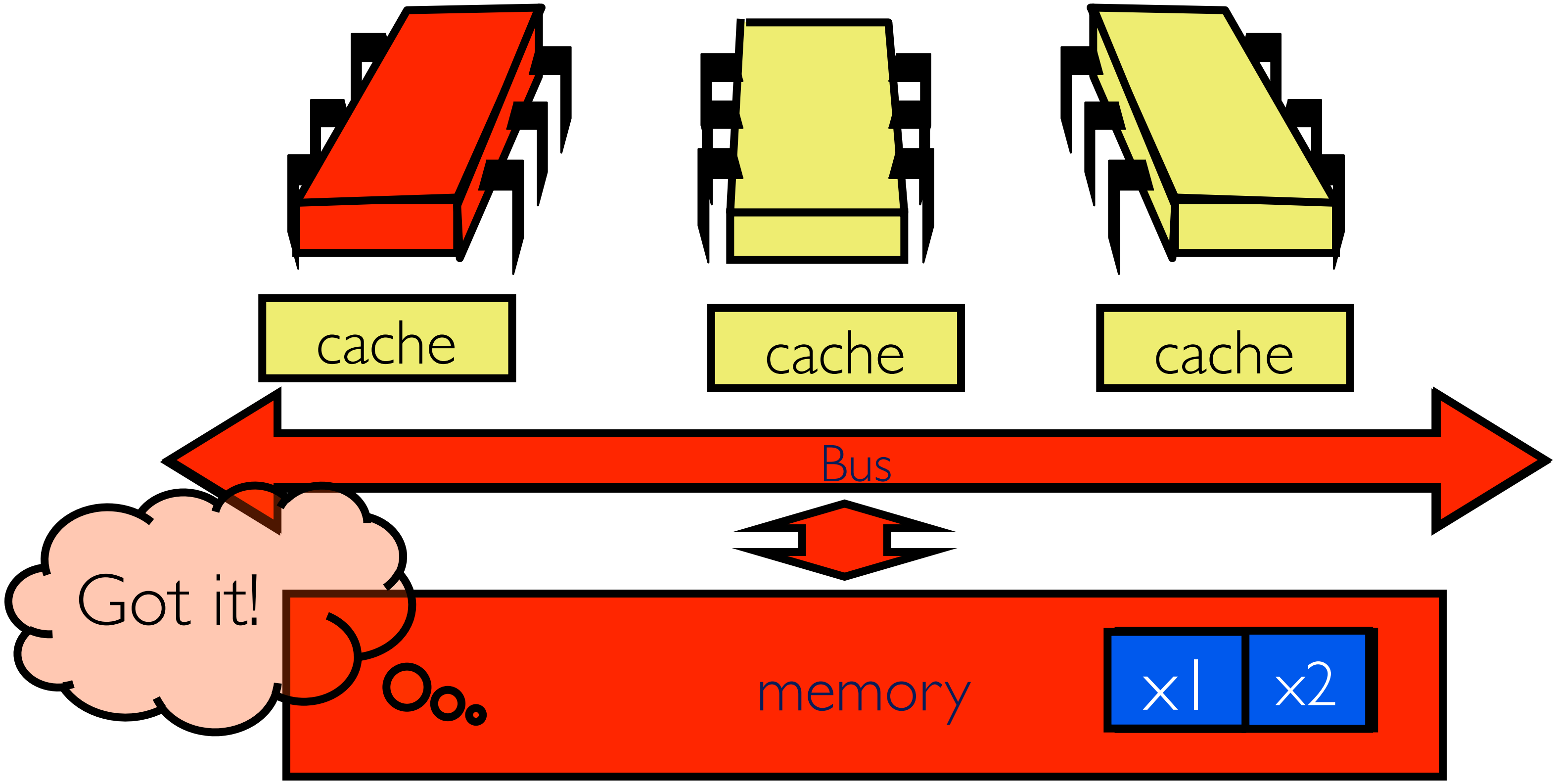
Coherence Misses

- "True Sharing"
  - When two processors read/write same variable "x"
  - Communicate a new value of "x" from one thread to another
  - Inherent to the computation
- "False Sharing"
  - Reading and writing two distinct variables "x1" and "x2"
  - Happen to reside in the same cache block
  - E.g., x1 and x2 are single words, but a 64-byte block can hold 8 words, and contains both "x1" and "x2"
- False sharing is unnecessary and can reduce performance

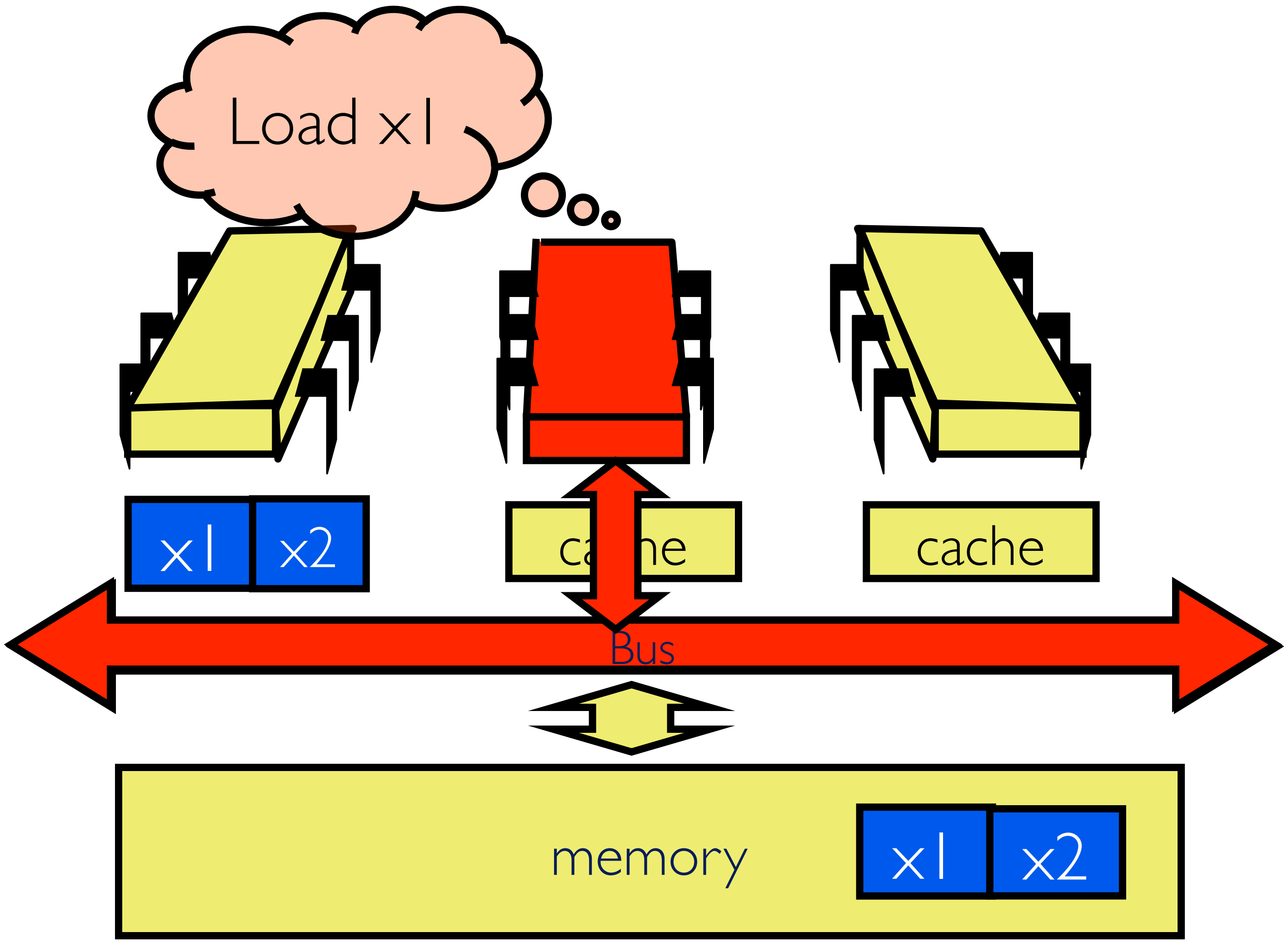
Processor Issues Load Request



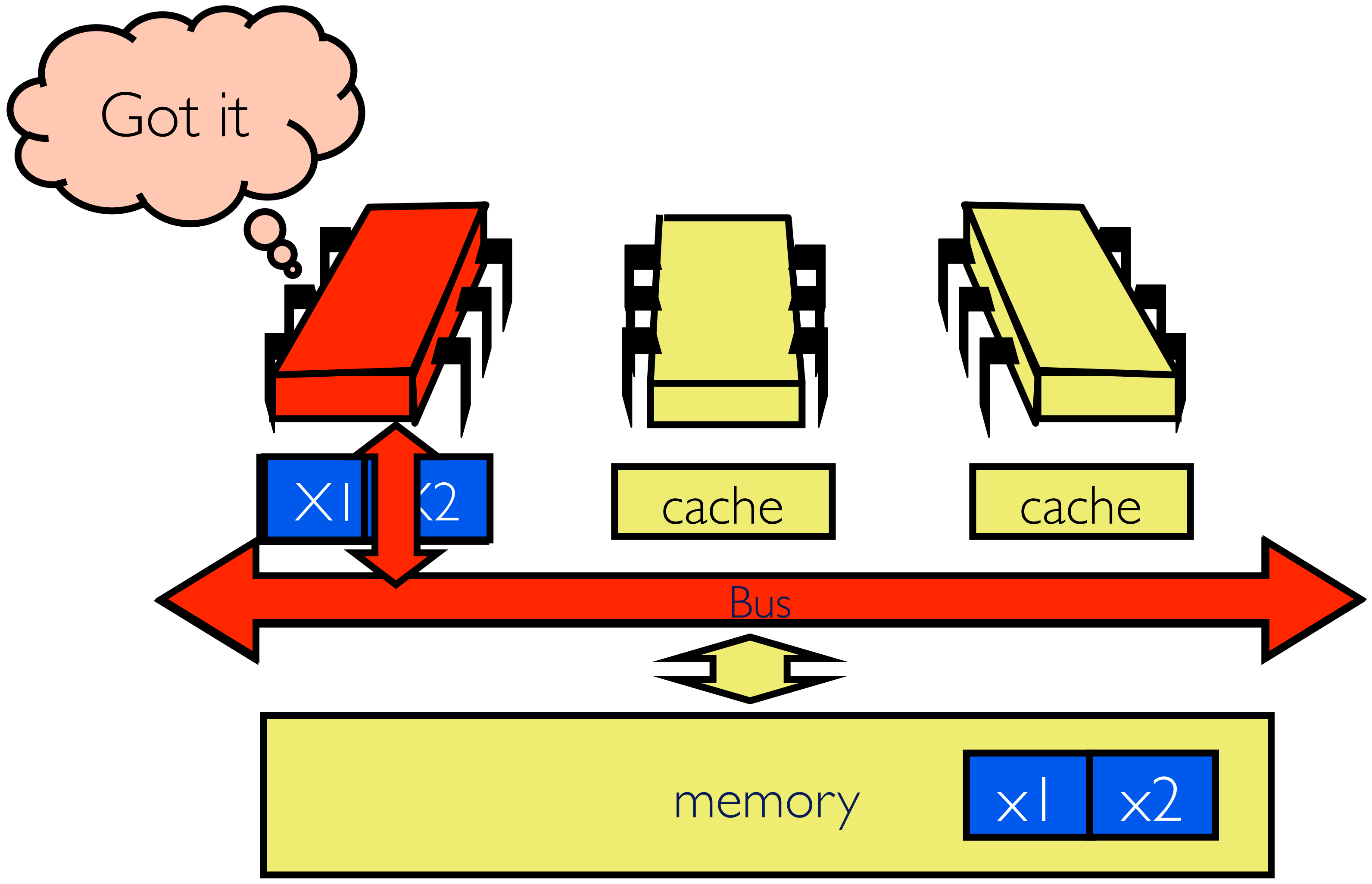
Memory Responds



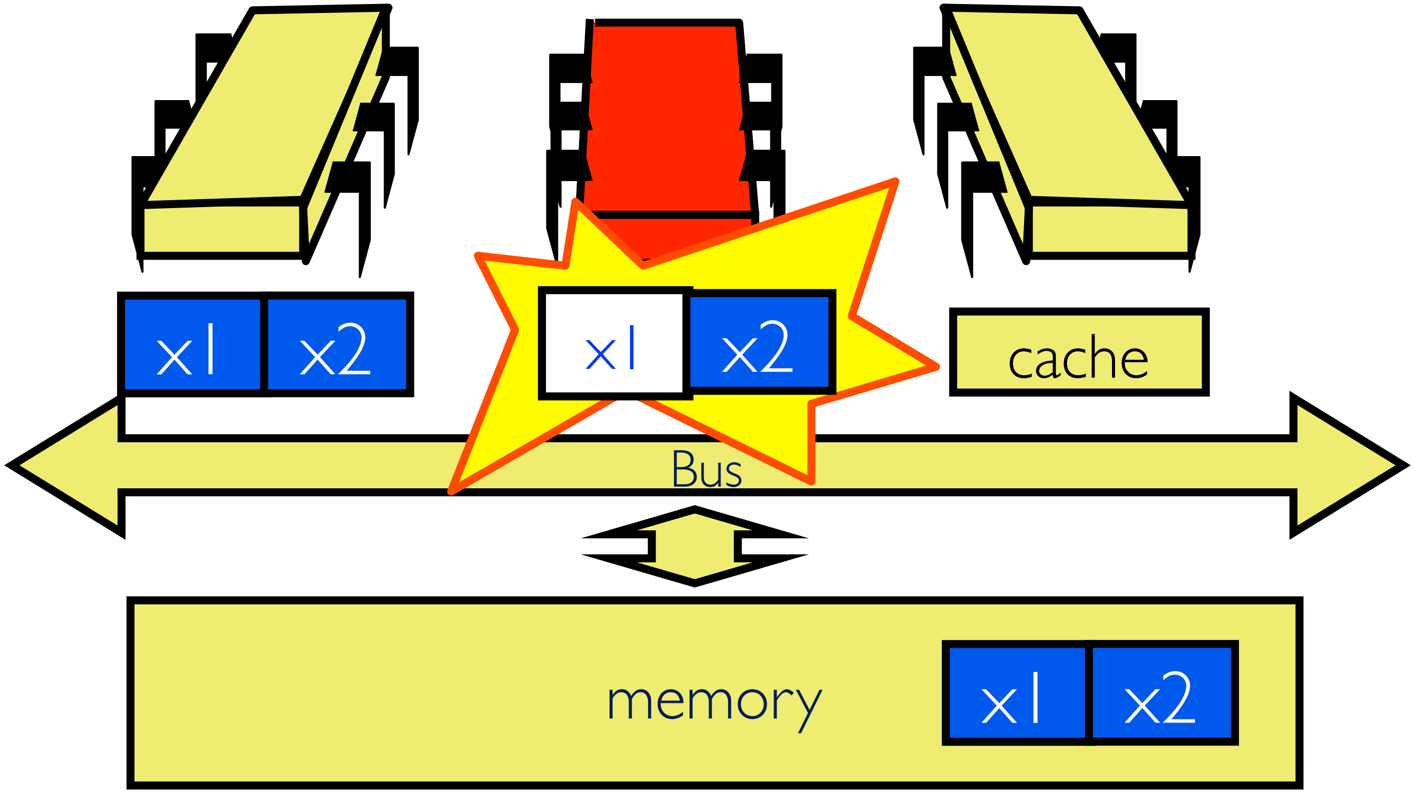
Processor Issues Load Request



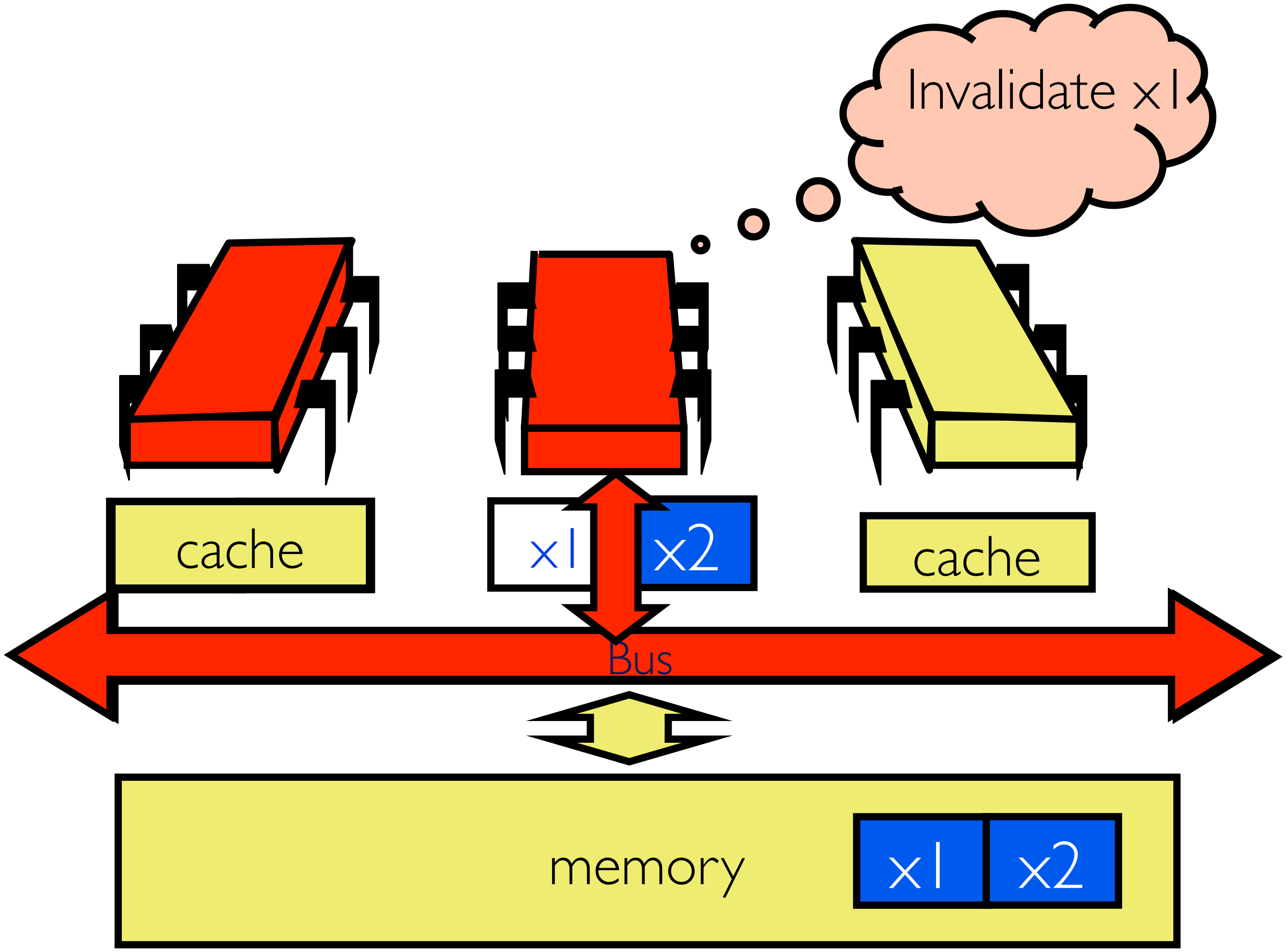
Other Processor Responds



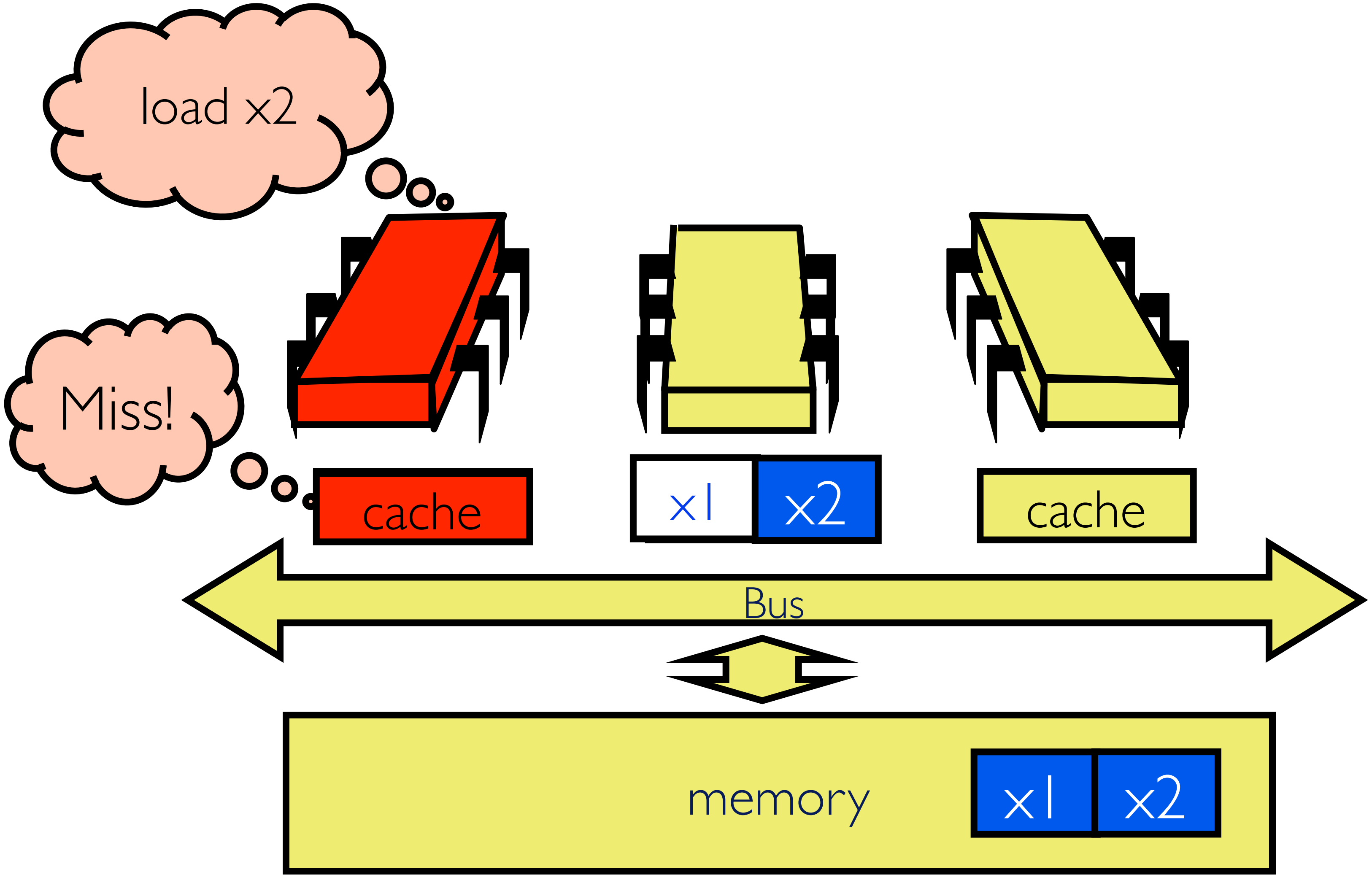
Modify Cached Data



Invalidate



False Sharing



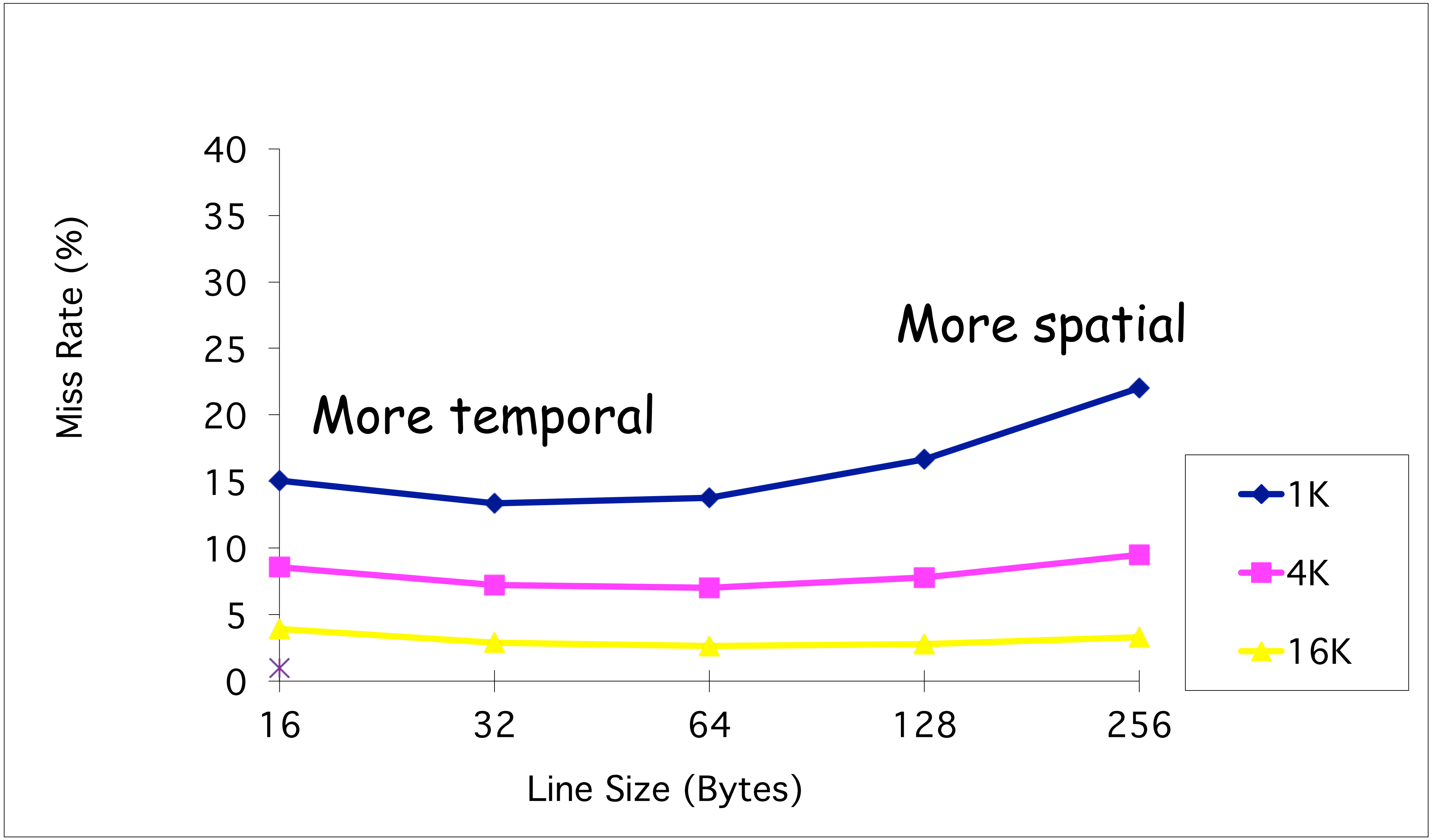
False Sharing causes extra misses!

- One miss to get rid of the copy in the first processor
  - Because of a write to “x2”
  - Even though the first processor only needs “x1”
- One miss for the processor to read “x1” back again
- A total of two misses just because of a write to “x2”
- A write to “x1” again would cause two misses

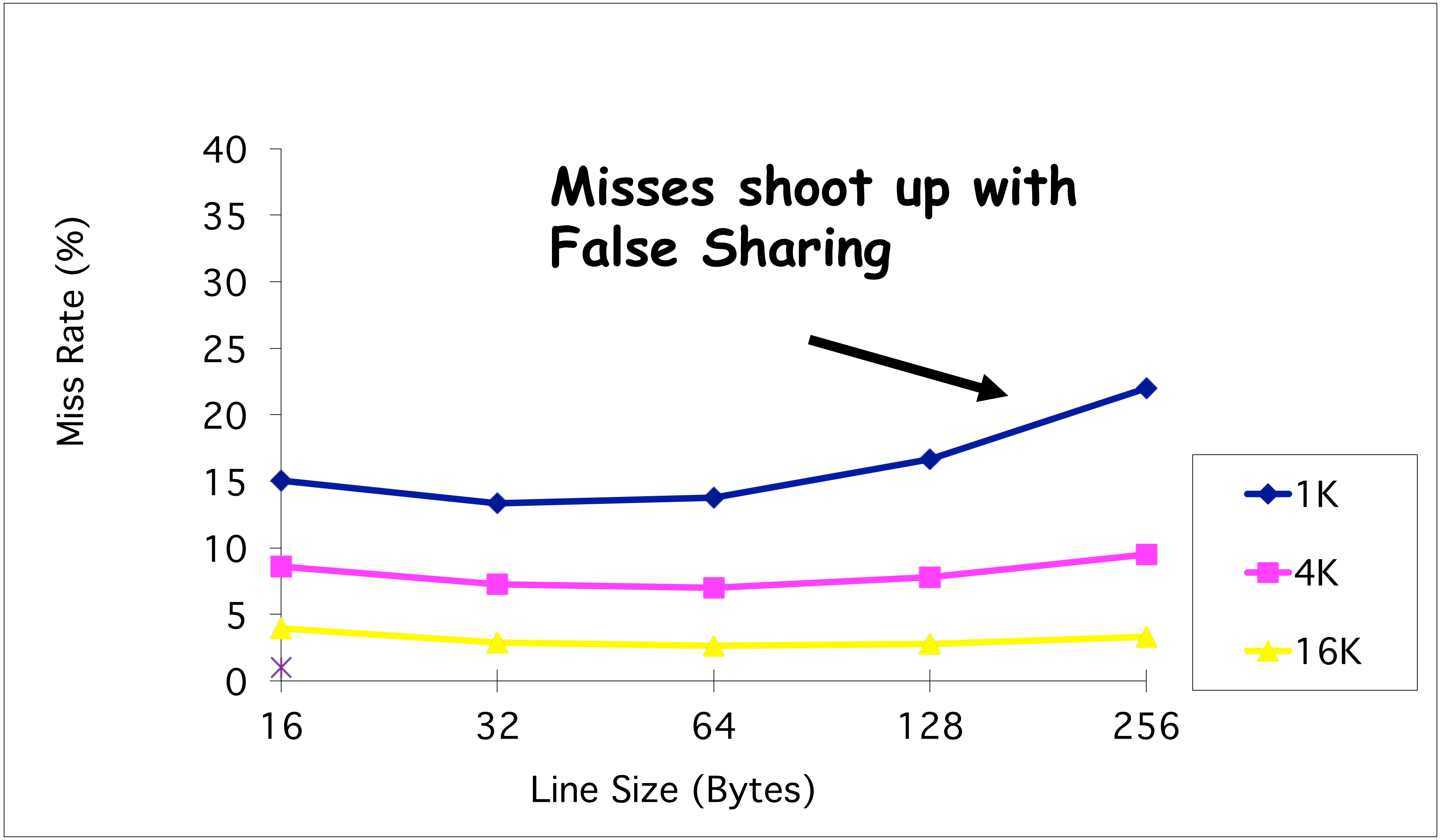
False Sharing gets worse with Bigger Blocks!

- The larger the block, the more likely threads will be reading/writing the same block at the same time
- When writing software, should try to divide up data structures to avoid False Sharing
  - Threads should not sharing near-neighbor variables in memory
  - We will talk about this when we start writing parallel software

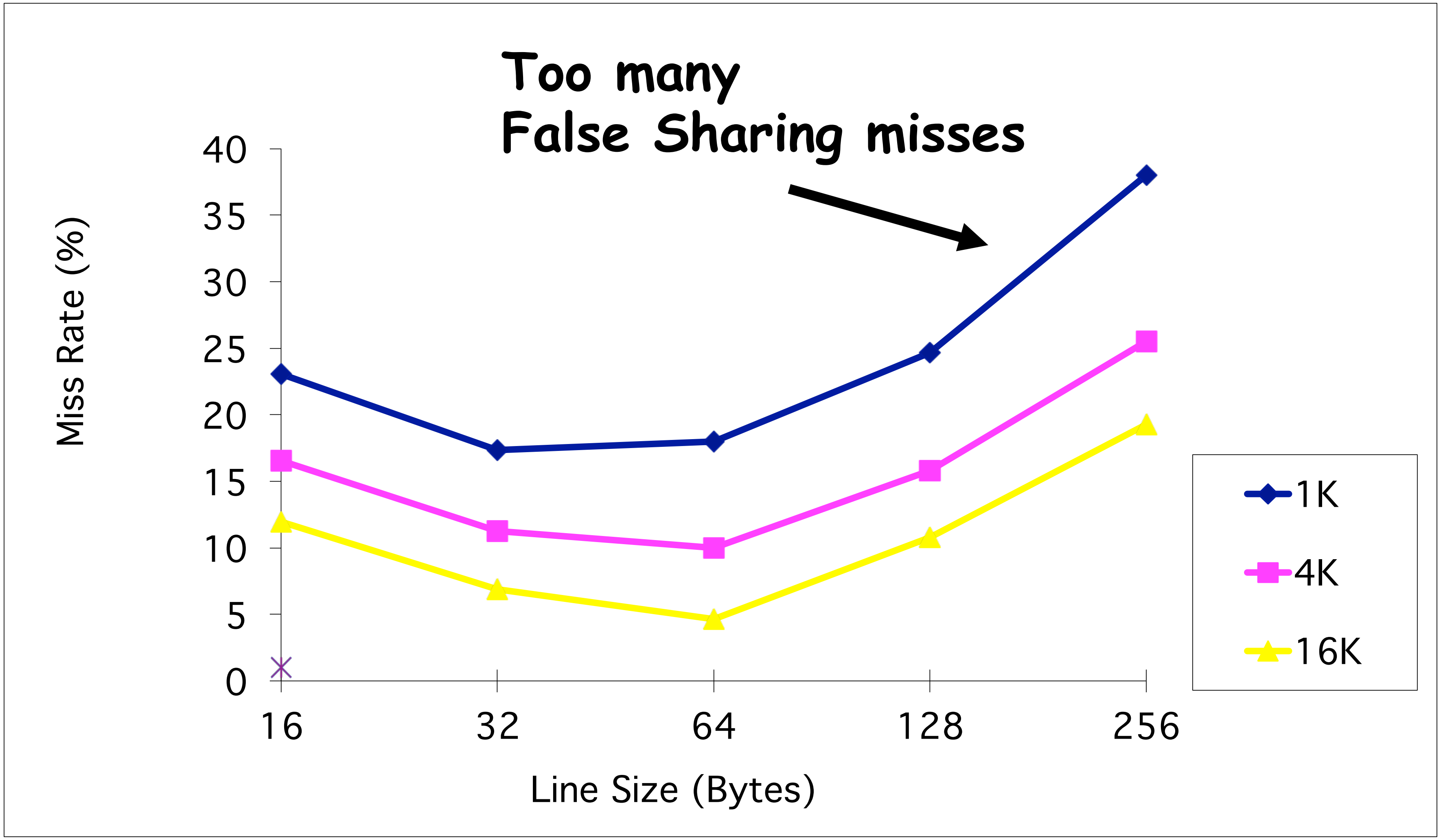
Miss rate vs. Block Size: Uniprocessor (CS-208)



Miss Rate vs. Block Size: 4 Processors



Miss Rate vs. Block Size: 16 Processors



Summary

- Most multiprocessors use shared memory
- We will assume an SMP, simple multiprocessor model
- Must know how to platform works to construct software
- Must understand the bottlenecks
- Next week, we will see how to think parallel