Concurrency:
Foundations and Algorithms

Multiprocessor Architecture Basics

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

*Based on slides by **Maurice Herlihy** and **Nir Shavit***

Institut d'informatique
Université de Neuchâtel

Multiprocessor Architecture

- Abstract models are (mostly) OK to understand algorithm correctness
- To understand how concurrent algorithms perform…
- You need to understand something about **multiprocessor architectures**

Pieces

- Processors
- Threads
- Interconnect
- Memory
- Caches

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 (cycles get faster!)

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 (preempted by OS)
- Run another thread

Interconnect

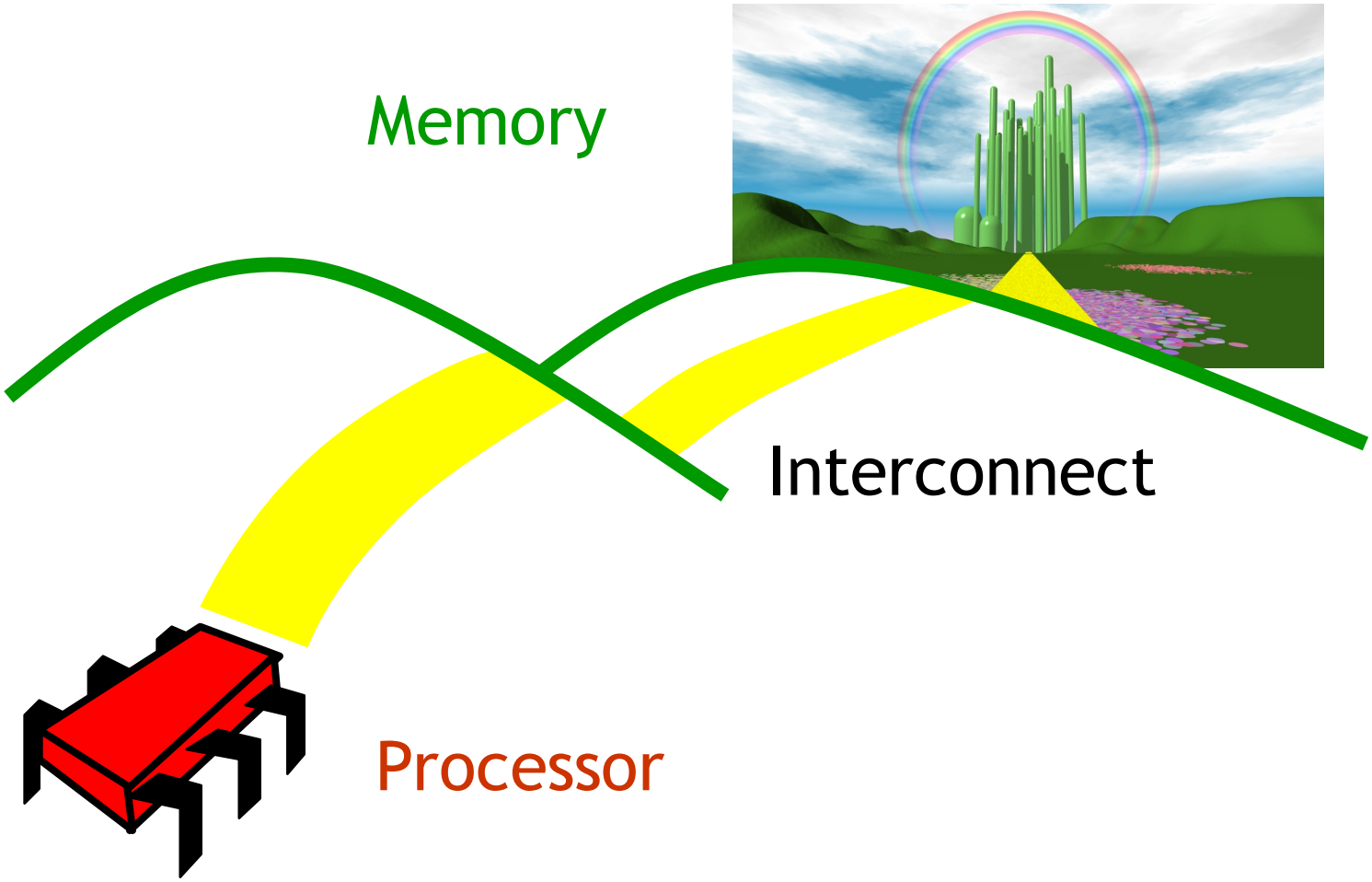
- Bus
  - Like a tiny Ethernet
  - Broadcast medium
  - Connects
    - Processors to memory
    - Processors to processors
- Network
  - Tiny LAN
  - Mostly used on large machines

What is interconnecting components of machine

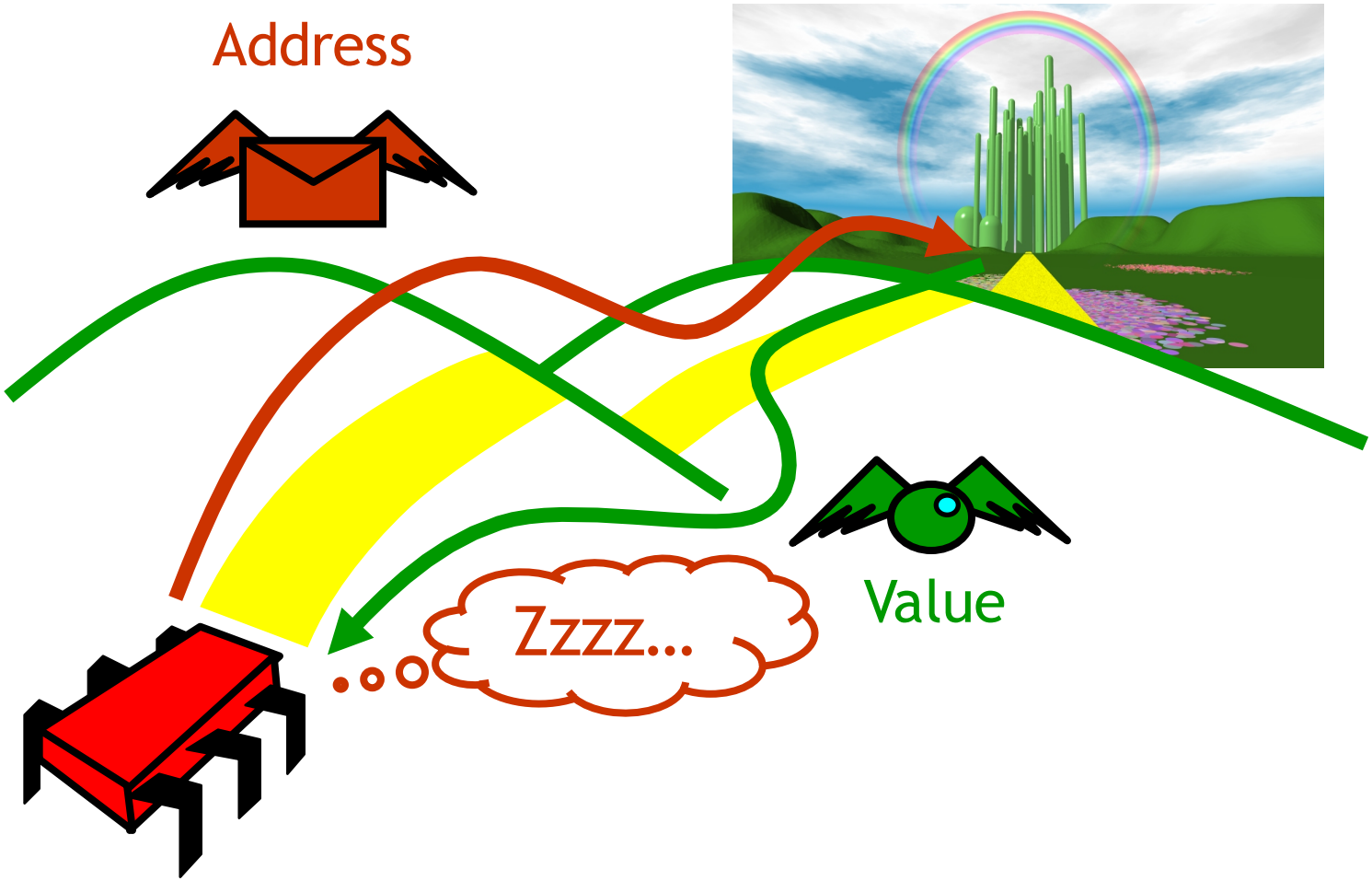
Interconnect

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

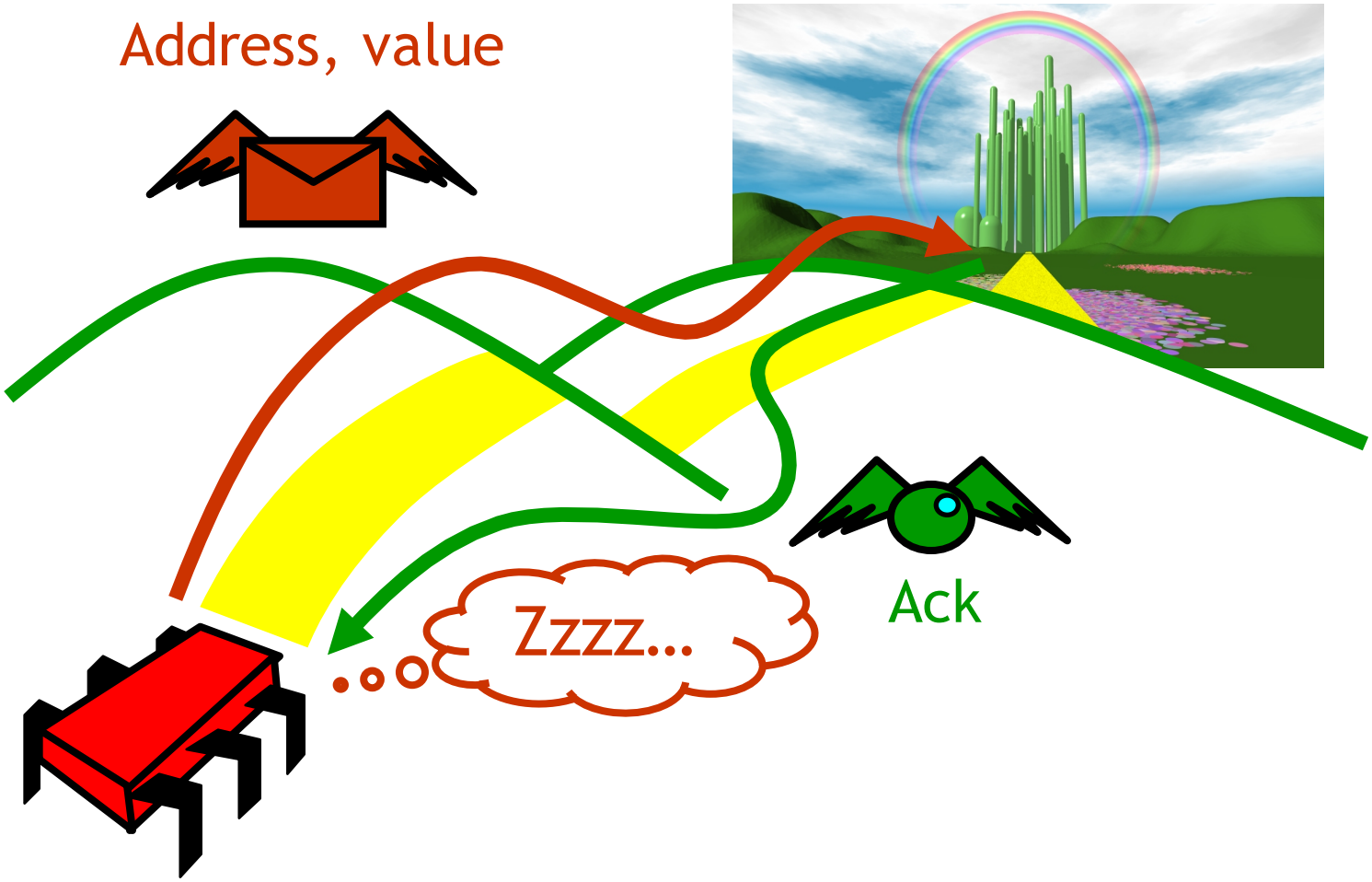
CPU and Memory are Far Apart



Reading from Memory



Writing to Memory



Remote Spinning

- Thread waits for a bit in memory to change
  - E.g., tries to dequeue from an empty buffer, tries to acquire lock owned by another thread
- Spins
  - Repeatedly rereads flag bit
- Huge waste of interconnect bandwidth
  - Generates continuous traffic on bus

Analogy

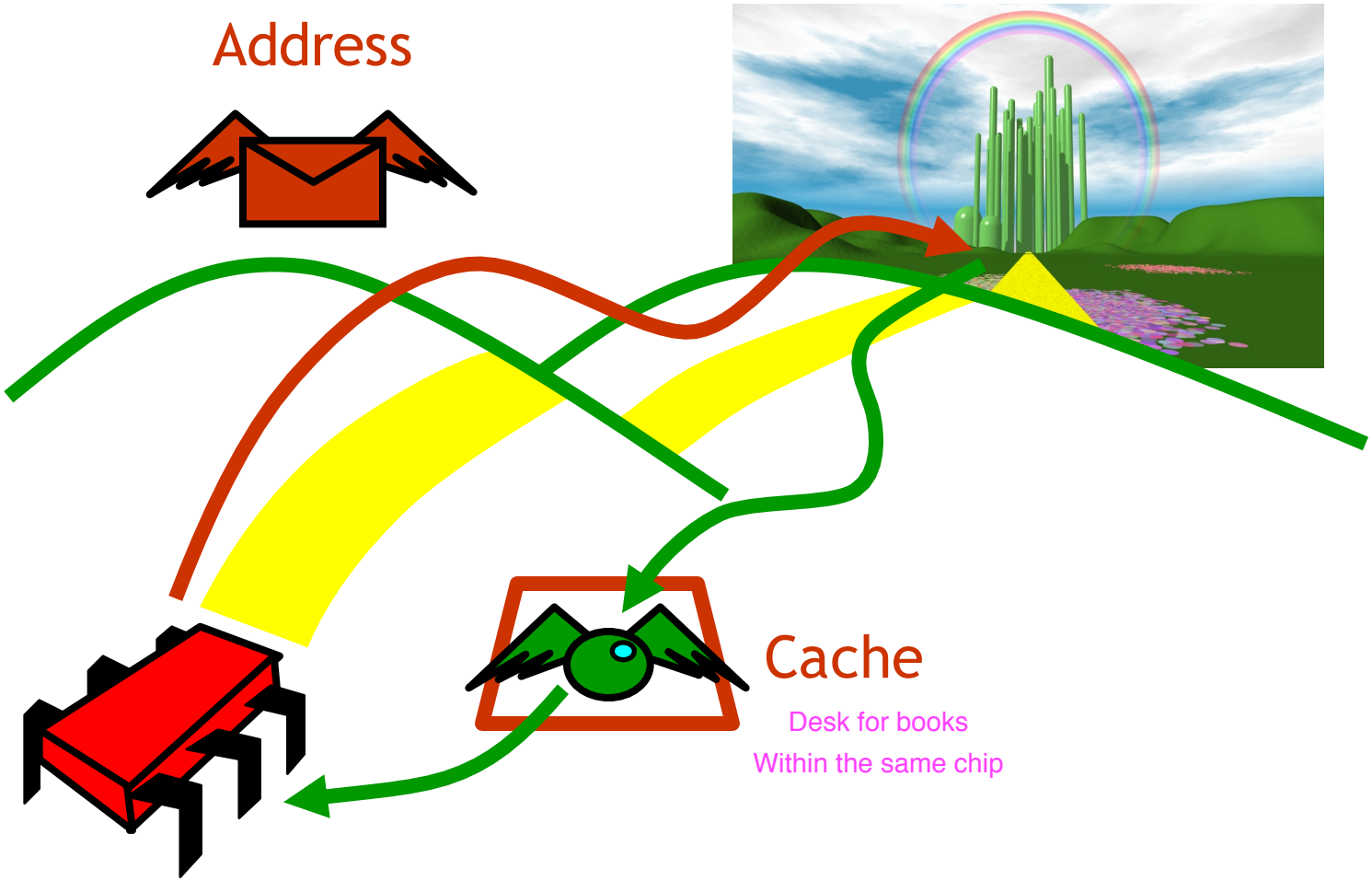
- In the days before the Internet…
- Alice is writing a paper on aardvarks
- Sources are in university library
  - Request book by campus mail
  - Book arrives by return mail
  - Send it back when not in use

- She spends a lot of time in the mail room

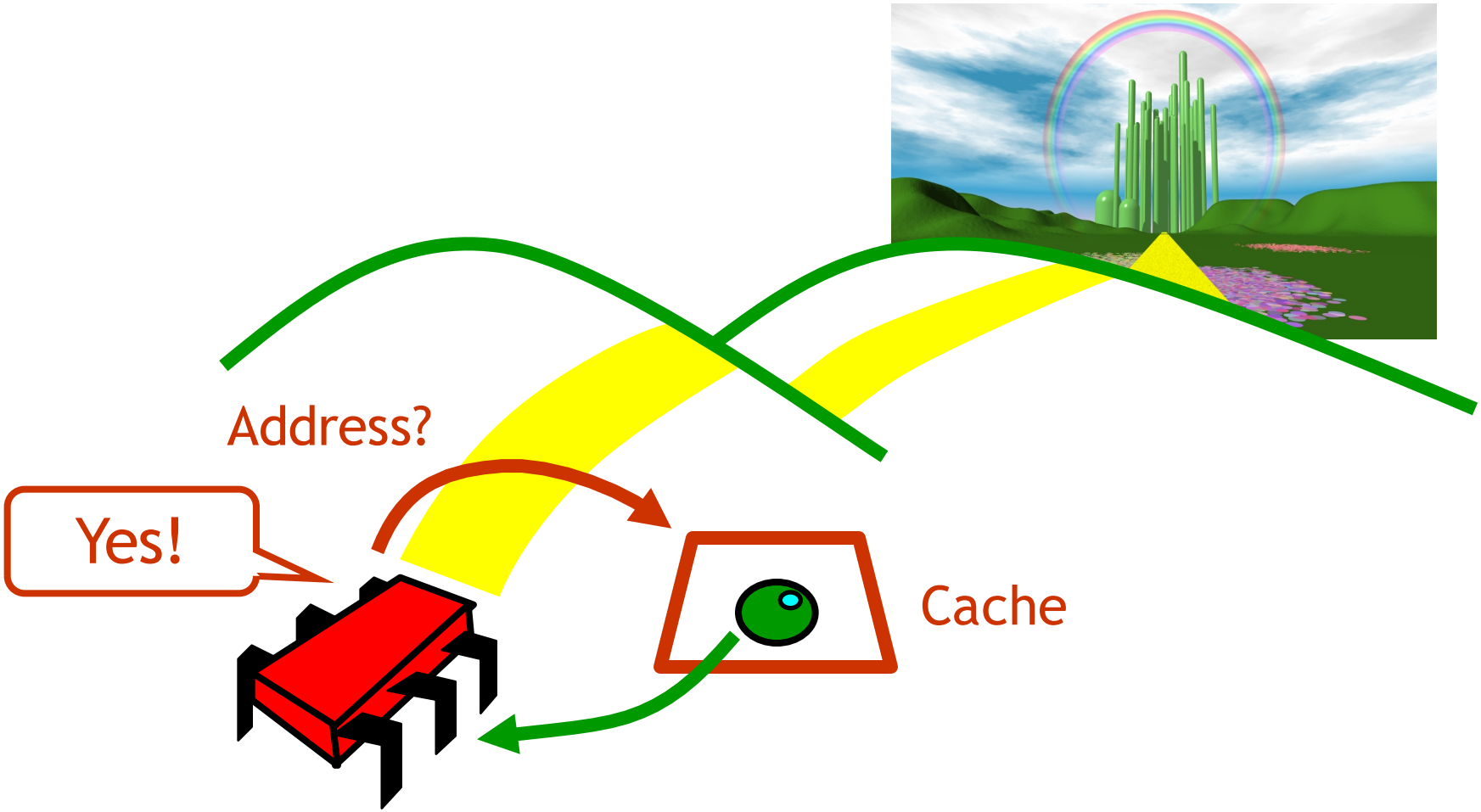
Analogy II

- Alice buys
  - A desk
    - In her office
    - To keep the books she is using now
  - A bookcase
    - In the hall
    - To keep the books she will need soon

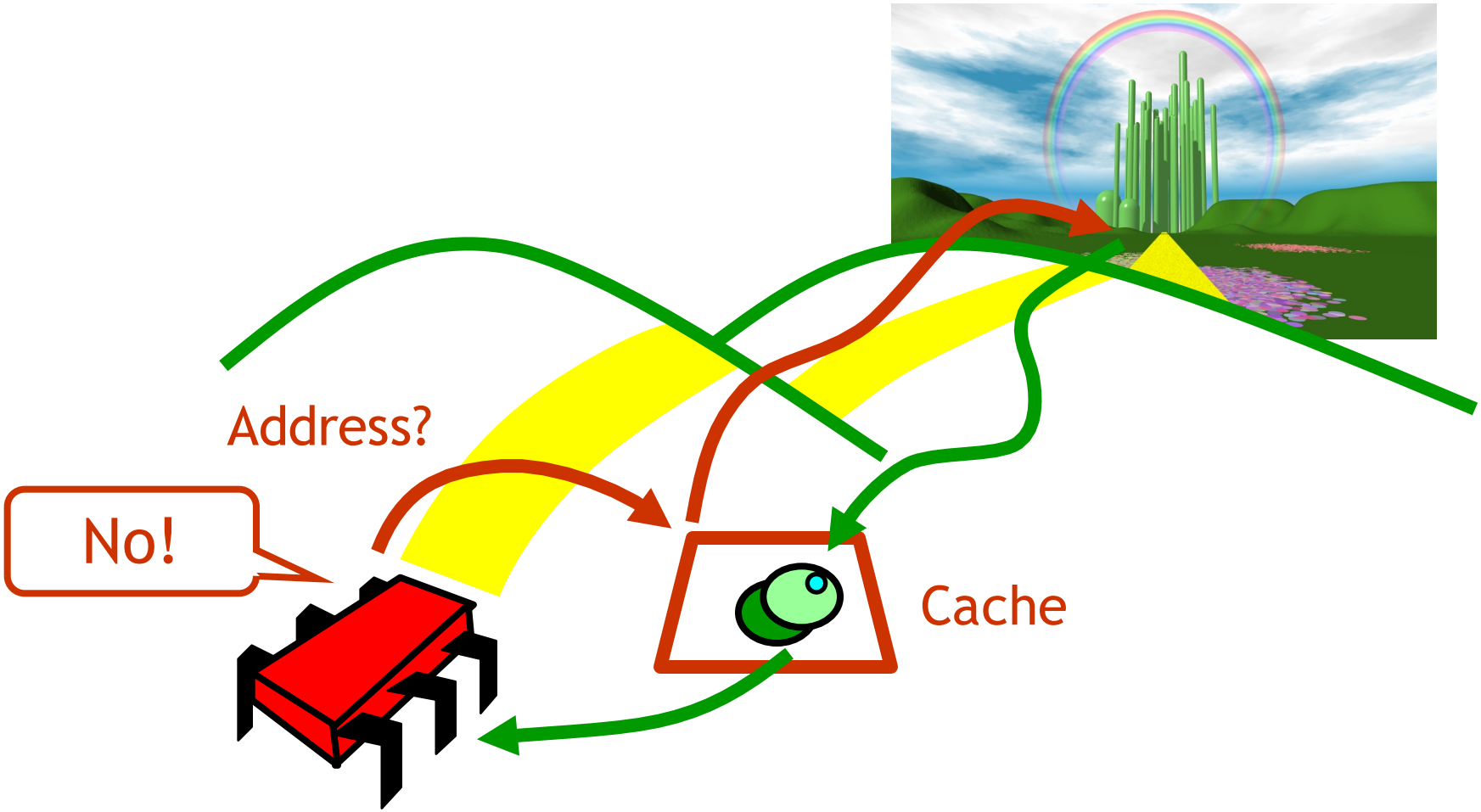
Cache: Reading from Memory



Cache Hit

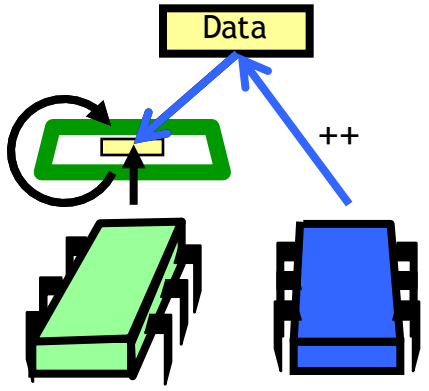


Cache Miss

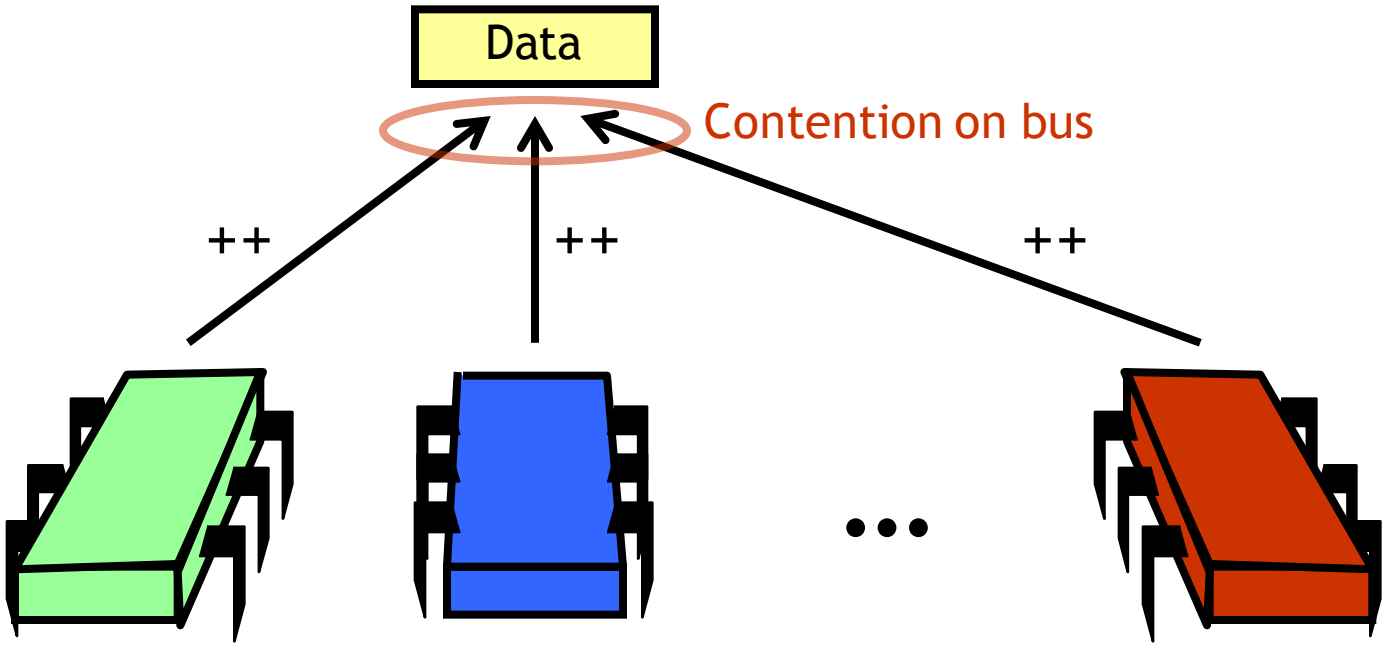


Local Spinning

- With caches, spinning becomes practical
- First time
  - Load flag bit into cache
- As long as it does not change
  - Hit in cache (no interconnect used)
- When it changes
  - One-time cost
  - See cache coherence below



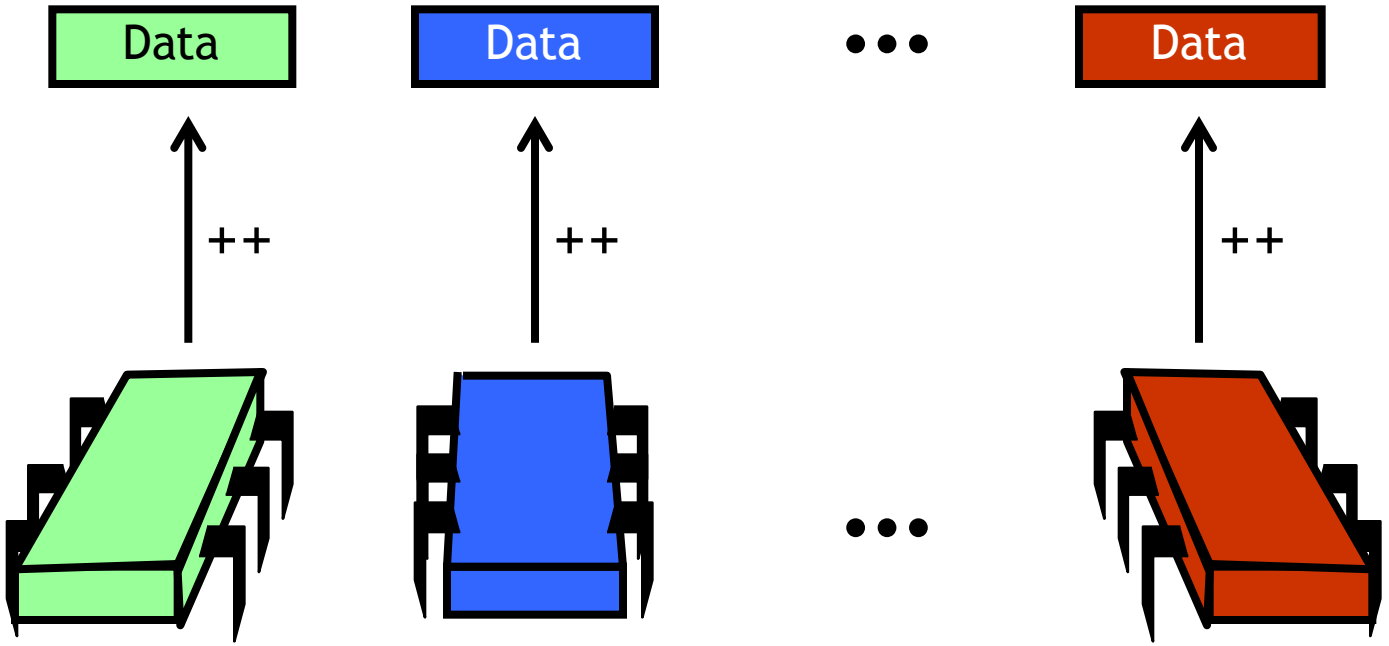
Understanding Cache Behavior



Analogy III

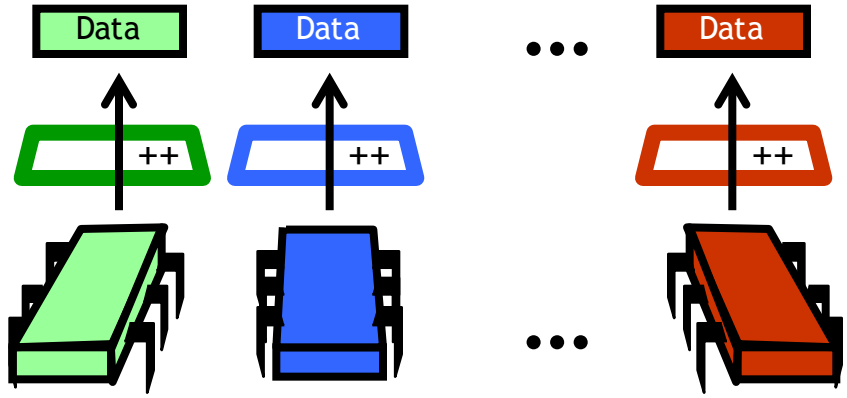
- Alice and Bob are both writing research papers on aardvarks
- Alice has encyclopedia volume AA-AC
  - Bob asks library for it
  - Library asks Alice to return it
  - Alice returns it & re-requests it
  - Library asks Bob to return it…
- It is good to avoid memory contention
  - Idle cores, consumes interconnect bandwidth

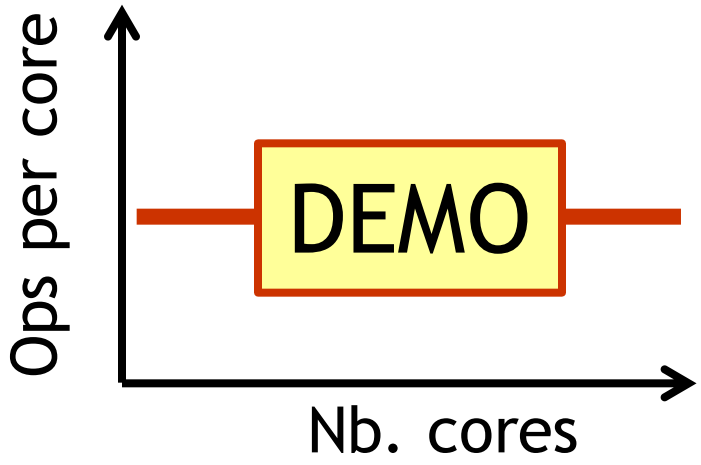
Understanding Cache Behavior



Understanding Cache Behavior

- Expected behavior
  - Data is not shared
  - Cores should read from and write to cache
  - No contention on bus
  - Throughput per thread should be constant if enough cores are available





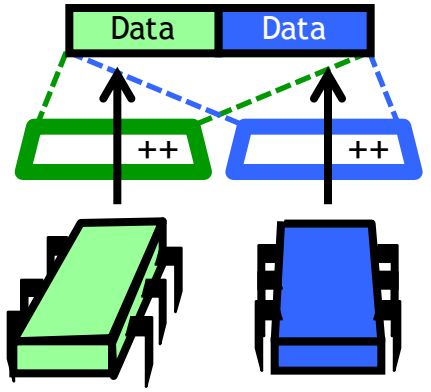
Granularity and Locality

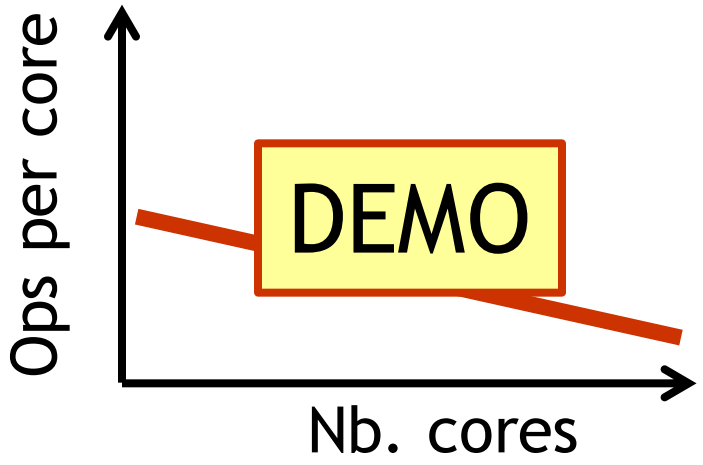
- Caches operate at a larger granularity than a word word is native size of data on CPU (word on 64bit CPU is 64bit)
- Cache line: fixed-size block containing the address
  - E.g., 16 words
- If you use an address now, you will probably use a nearby address soon
  - In the same cache line

Understanding Cache Behavior

Same kind of contention, but transparently created by the cache

- Observed behavior
  - Variables are in the same cache line
  - Cache lines are shared
  - Modification of one variable invalidates full cache line
  - Every write invalidates caches of cores sharing same cache line





Analogy IV

False sharing

- Alice is still writing a research paper on aardvarks
- Carol is writing a tourist guide to the German city of Aachen
- No conflict?
  - Library deals with volumes, not articles
  - Both require same encyclopedia volume

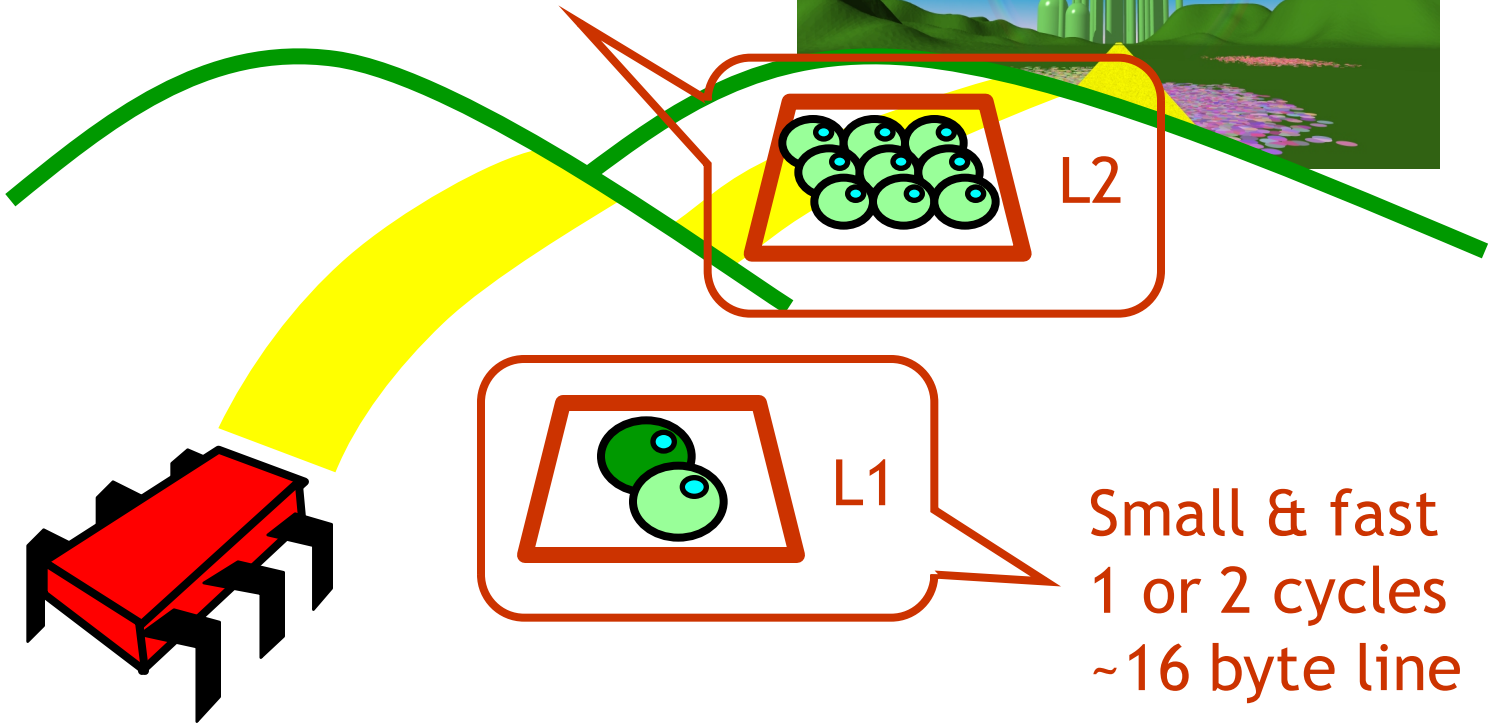
False Sharing

- Two processors may conflict over disjoint addresses
  - If those addresses lie on the same cache line
- Large cache line size
  - Increases locality
  - But also increases likelihood of false sharing
- Sometimes need to “scatter” data to avoid this problem

L1 and L2 Caches

Usually shared among multiple cores on the same processor

Larger and slower
10s of cycles
~1K line size



Small & fast
1 or 2 cycles
~16 byte line

Hit Ratio

Indicates how frequently you try to access a memory location and find it in the cache

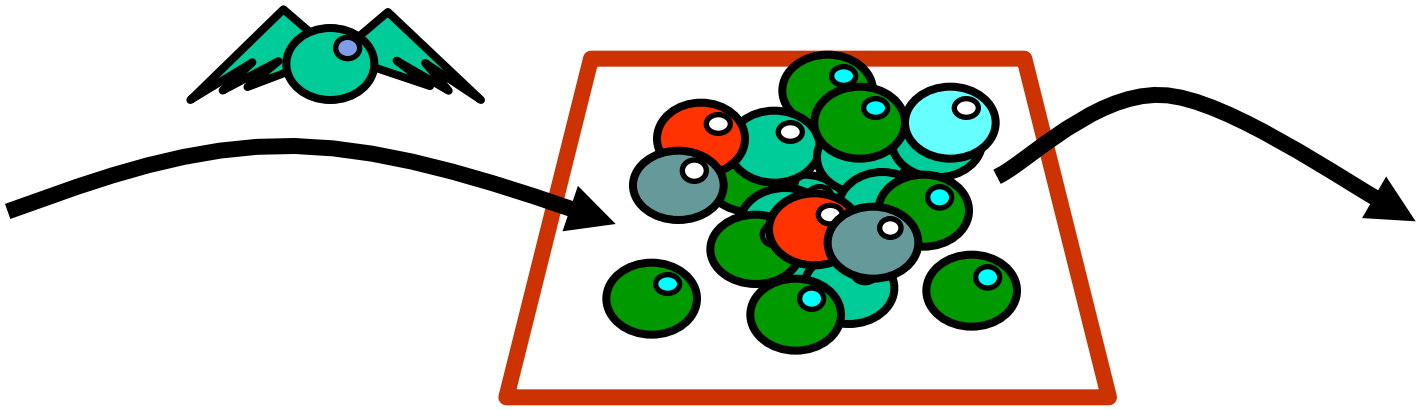
- If you use an address now, you will probably use it again soon
  - Fetch from cache, not memory
- Hit ratio: proportion of requests that hit in the cache
  - Measure of effectiveness of caching mechanism
  - Depends on locality of application

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

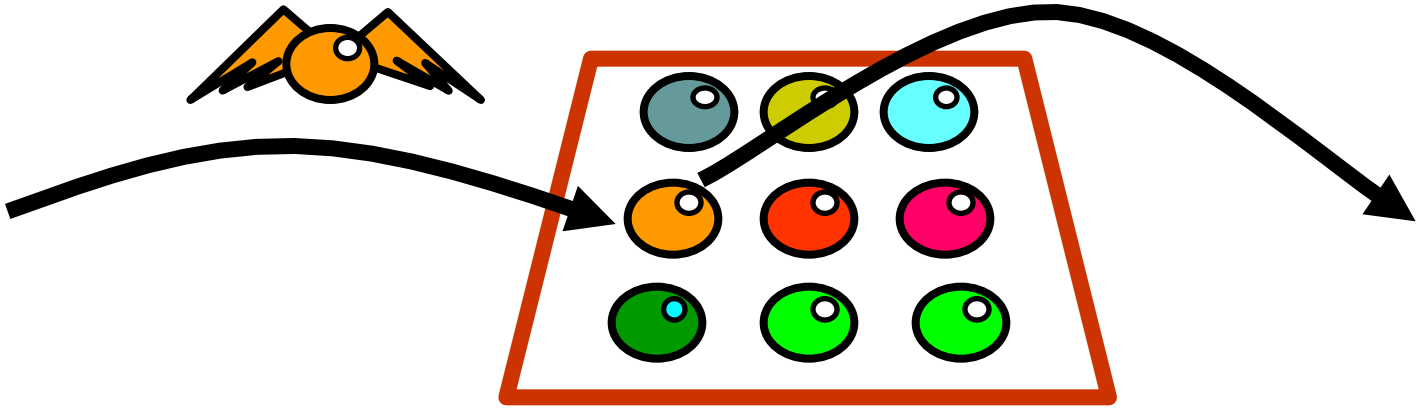
Fully Associative Cache

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



Direct Mapped Cache

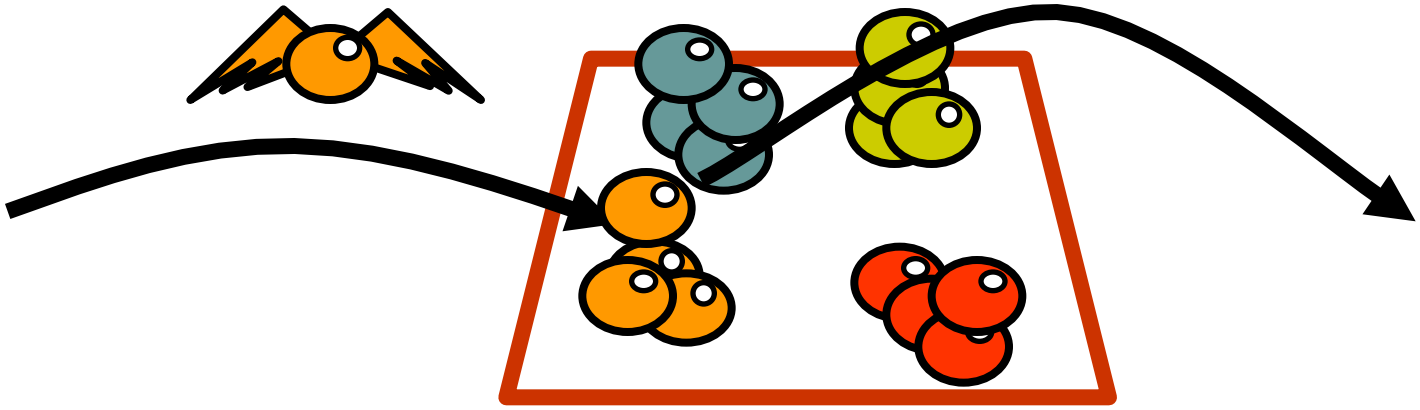
- Every address has exactly 1 slot
  - Advantage: easy to find a line
  - Disadvantage: must replace fixed line



K-way Set Associative Cache

What you have in most systems

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



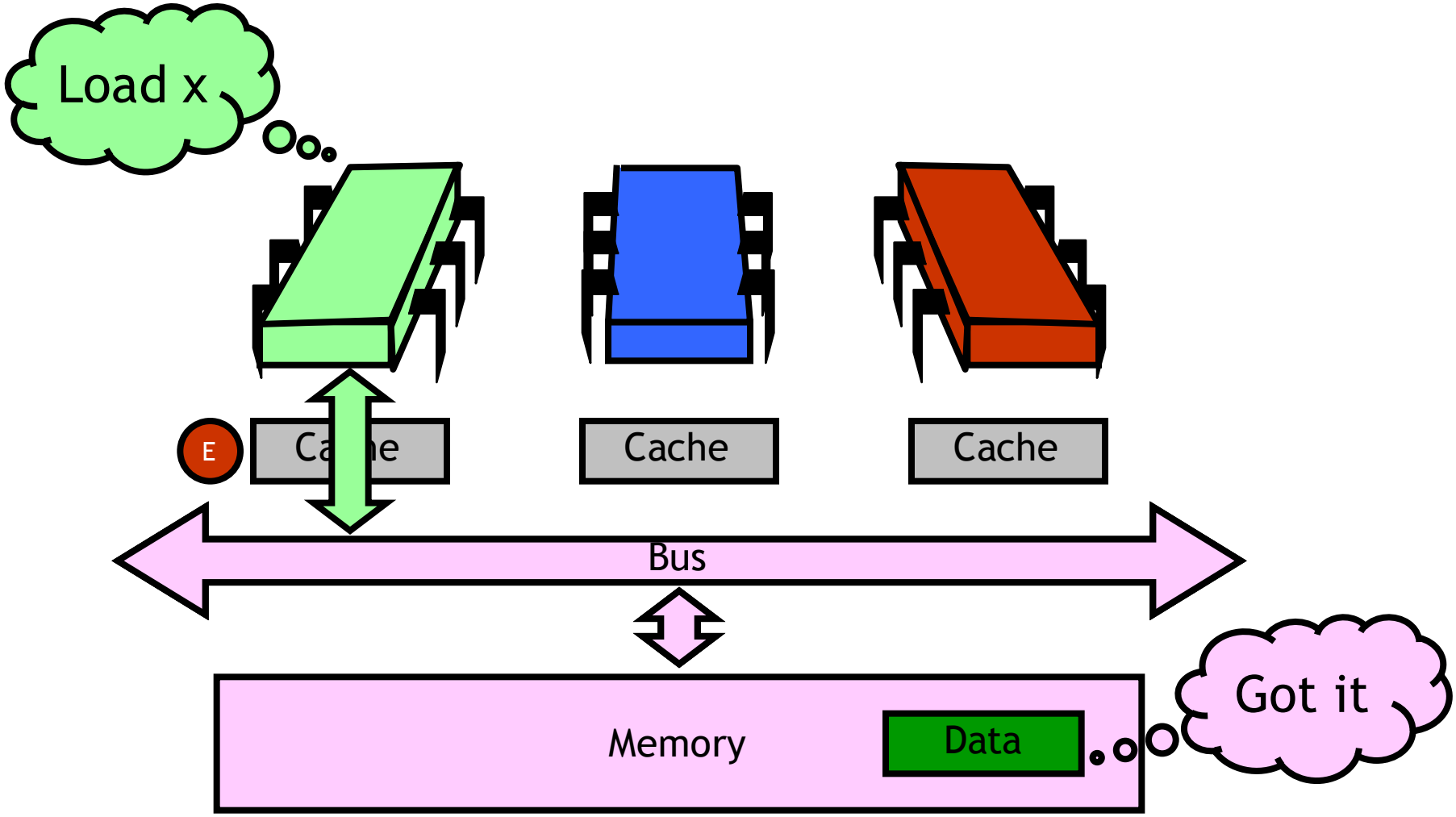
Cache Coherence

- Processor **A** and **B** both cache address **x**
- **A** writes to **x**
  - Updates cache
- How does **B** find out?
- Many cache coherence protocols in literature

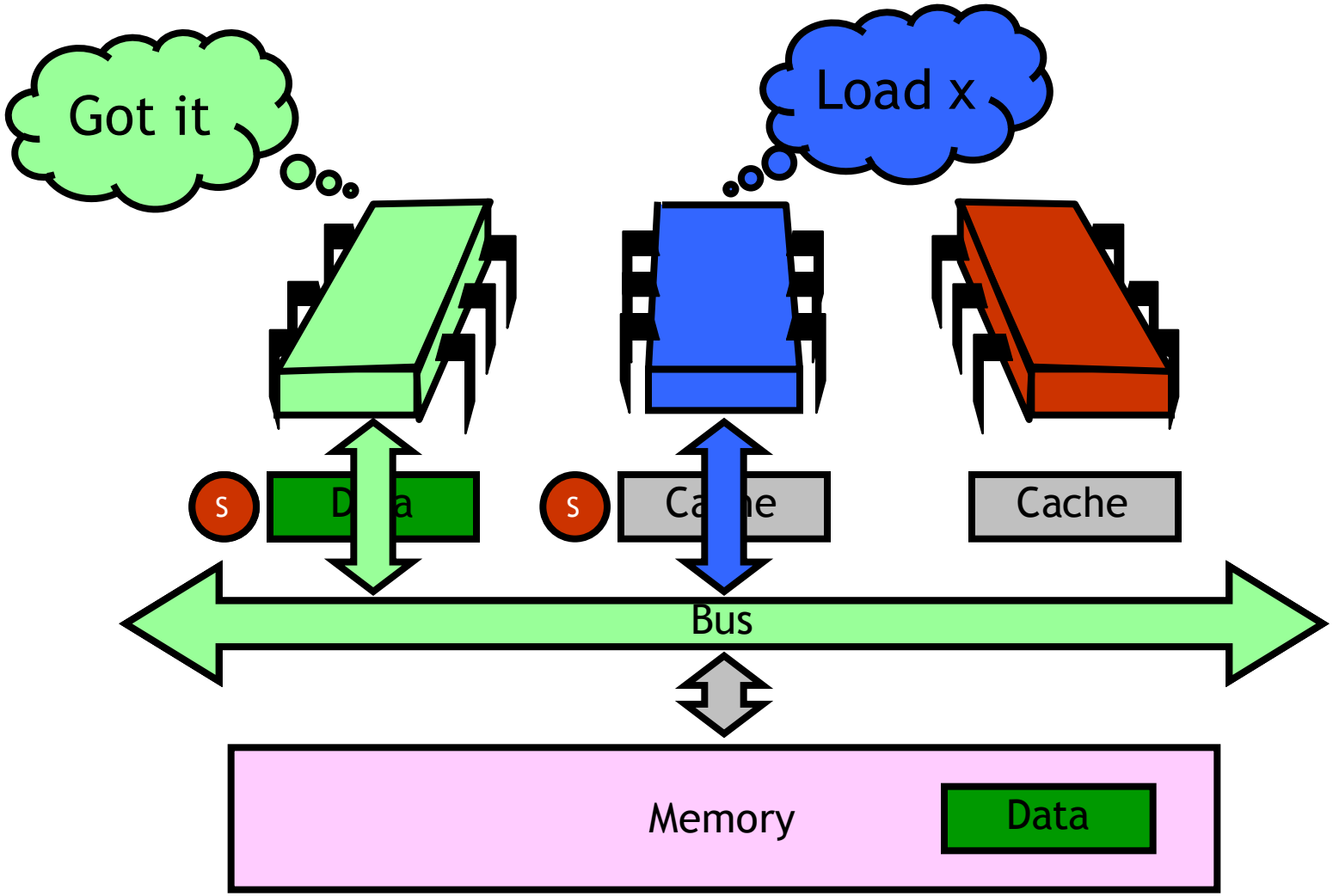
MESI

- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy
- Shared
  - Not modified, may be cached elsewhere
- Invalid
  - Cache content not meaningful

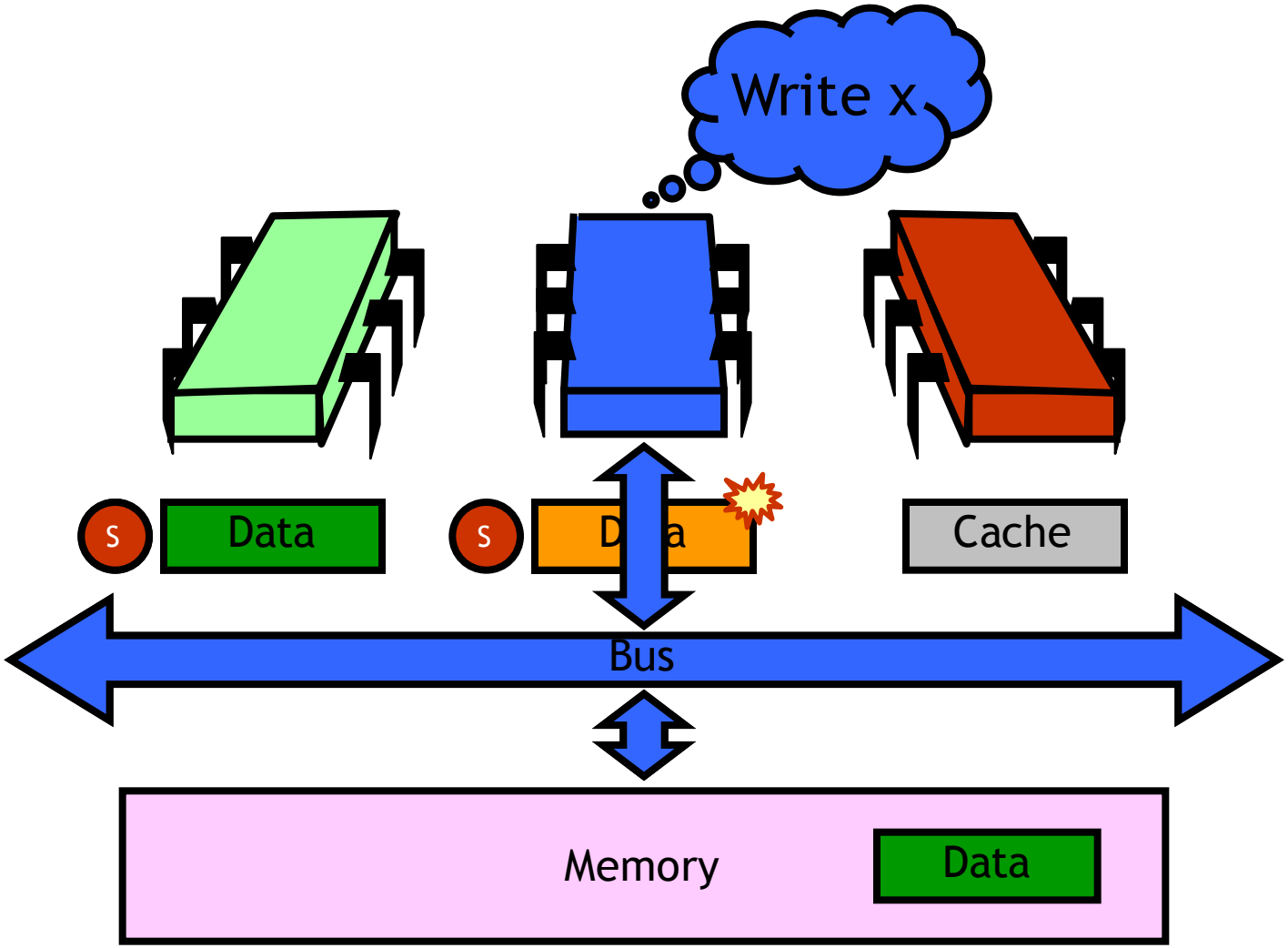
Processor Issues Load Request



2nd Processor Loads Data



Write Data: Write-Through Cache



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…

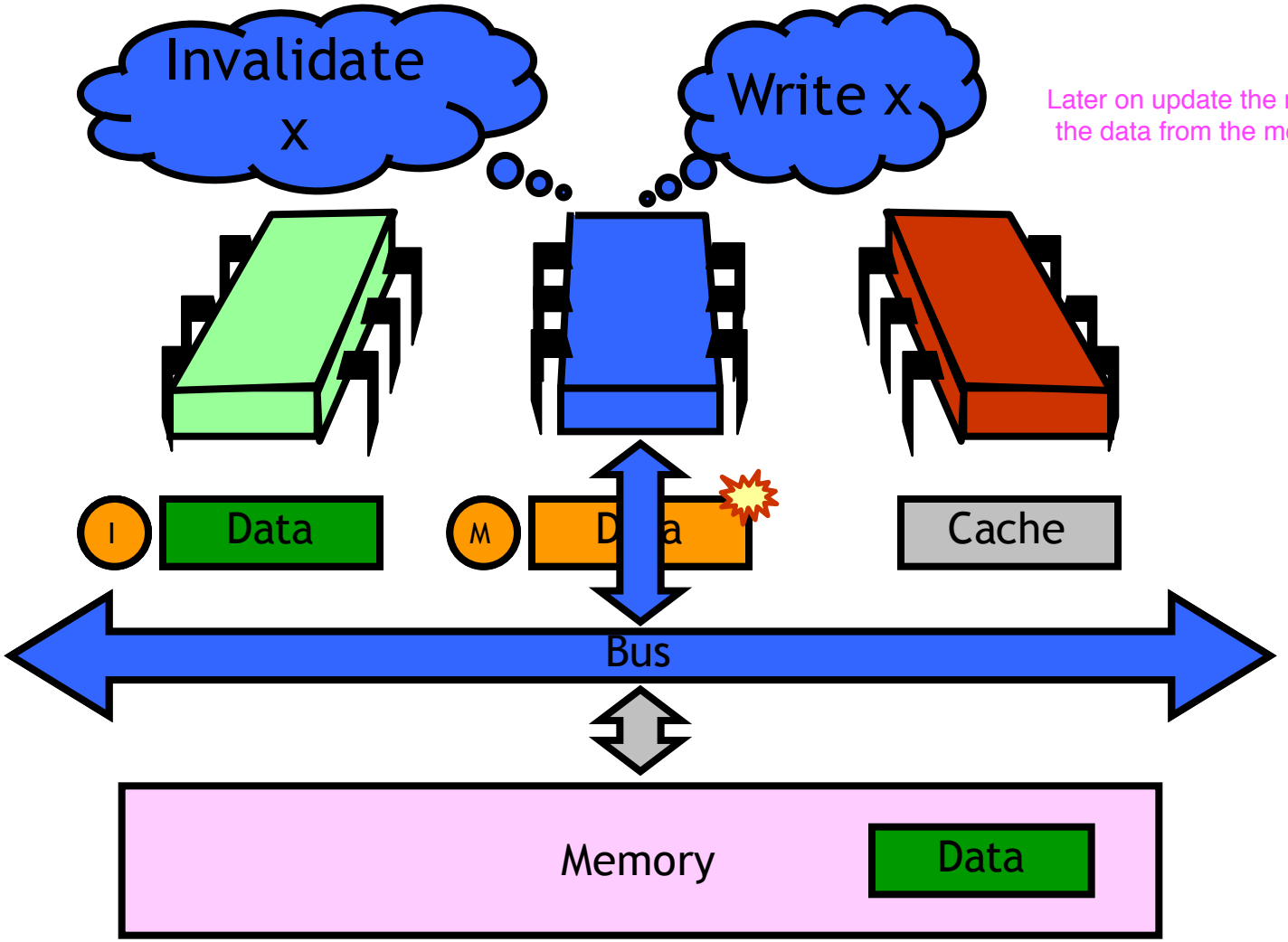
(1)

Write-Back Caches

- Accumulate changes in cache
  - Invalidate other copies
- Write back when line evicted
  - Need the cache for something else
  - Another processor wants it

If it's shared, we have to let the other theads know via the bus

Write Data: Write-Back Cache

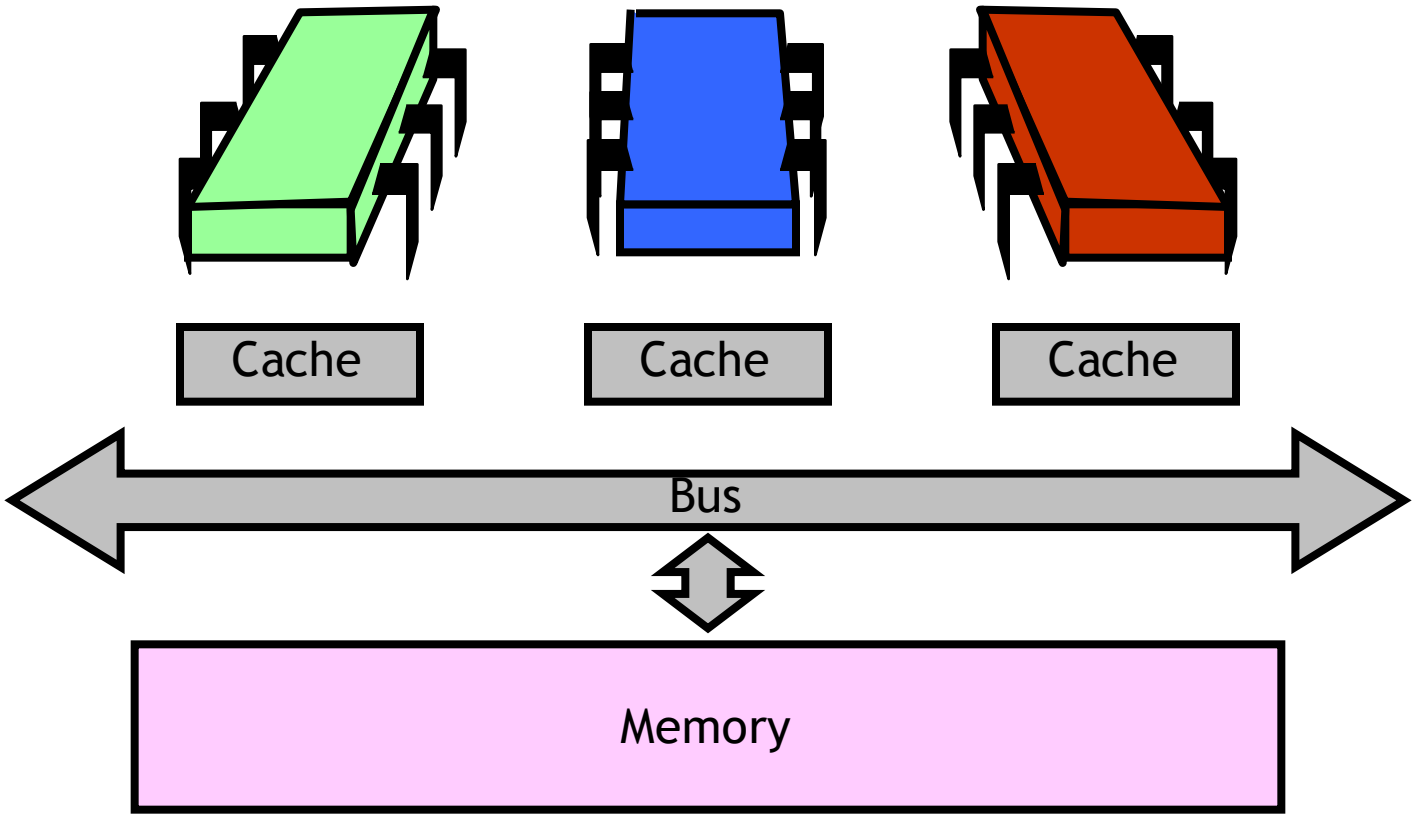


Later on update the main memory with the data from the modified cache line

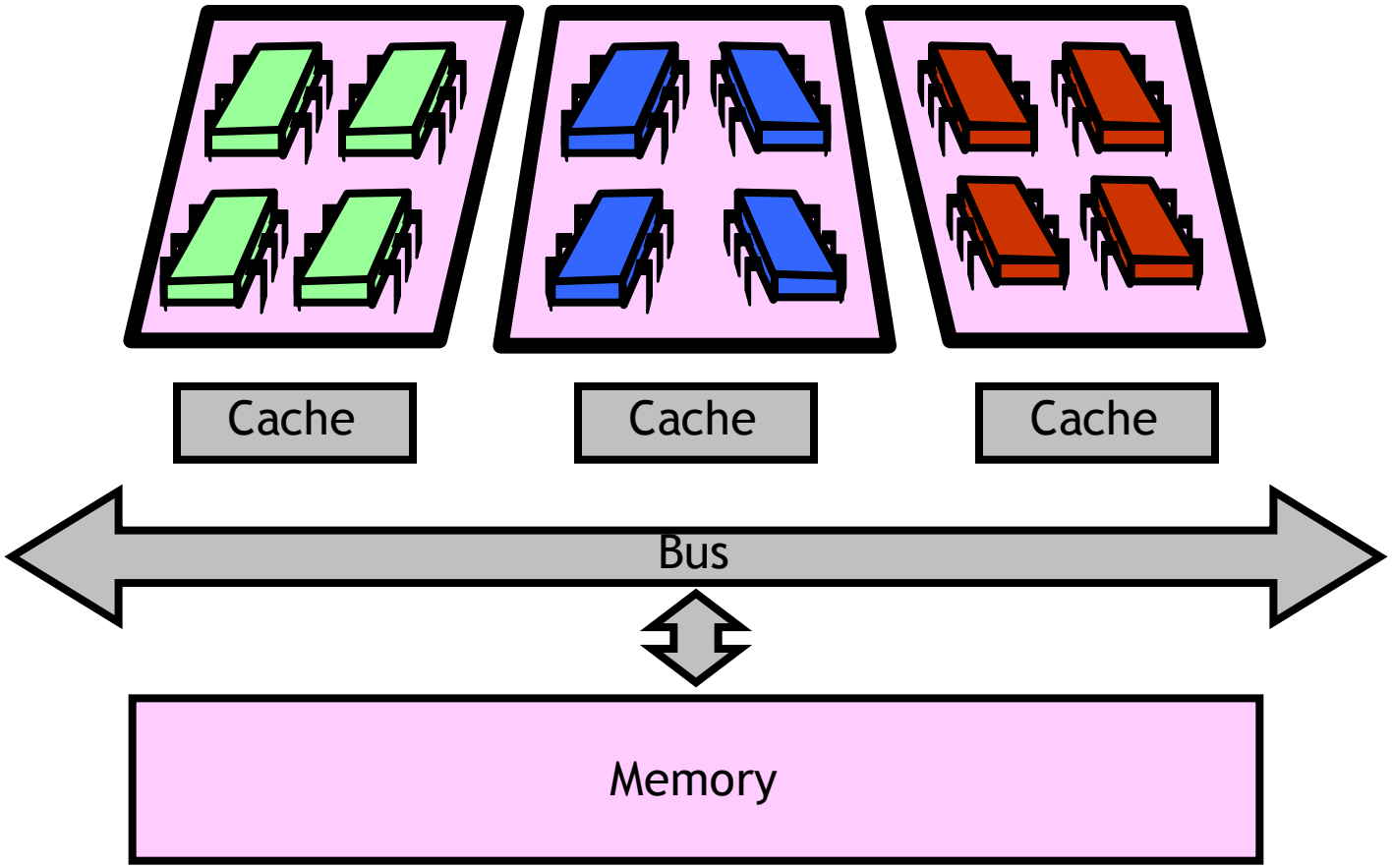
Multicore Architectures

- The university president
  - Alarmed by fall in productivity
- Puts Alice, Bob, and Carol in same corridor
  - Private desks
  - Shared bookcase
- Contention costs go way down

Old-School Multiprocessor



Multicore Architecture



Multicore

- Private L1 caches
- Shared L2 caches
- Communication between same-chip processors now very fast
- Different-chip processors still not so fast

NUMA Architectures

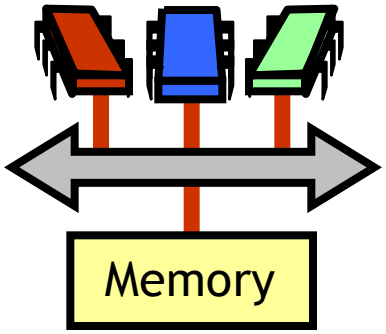
Getting back into fashion

- Alice and Bob transfer to NUMA State University
- No centralized library
- Each office basement holds part of the library
- Alice has volumes that start with A
  - Aardvark papers are convenient: run downstairs
  - Zebra papers are inconvenient: run across campus

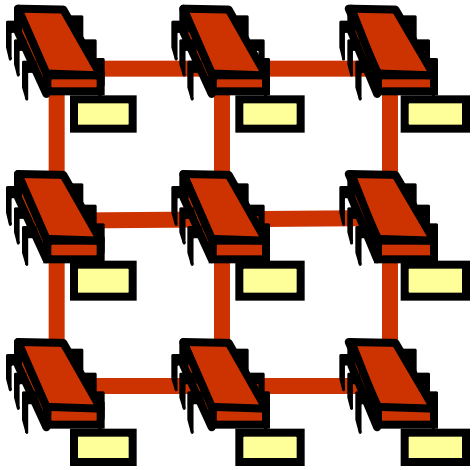
SMP vs. NUMA

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

Different processors each have a piece of the memory.



SMP



NUMA

Spinning Again

- NUMA without cache
  - OK if local variable
  - Bad if remote
- CC-NUMA
  - Cache-coherent
  - Like SMP

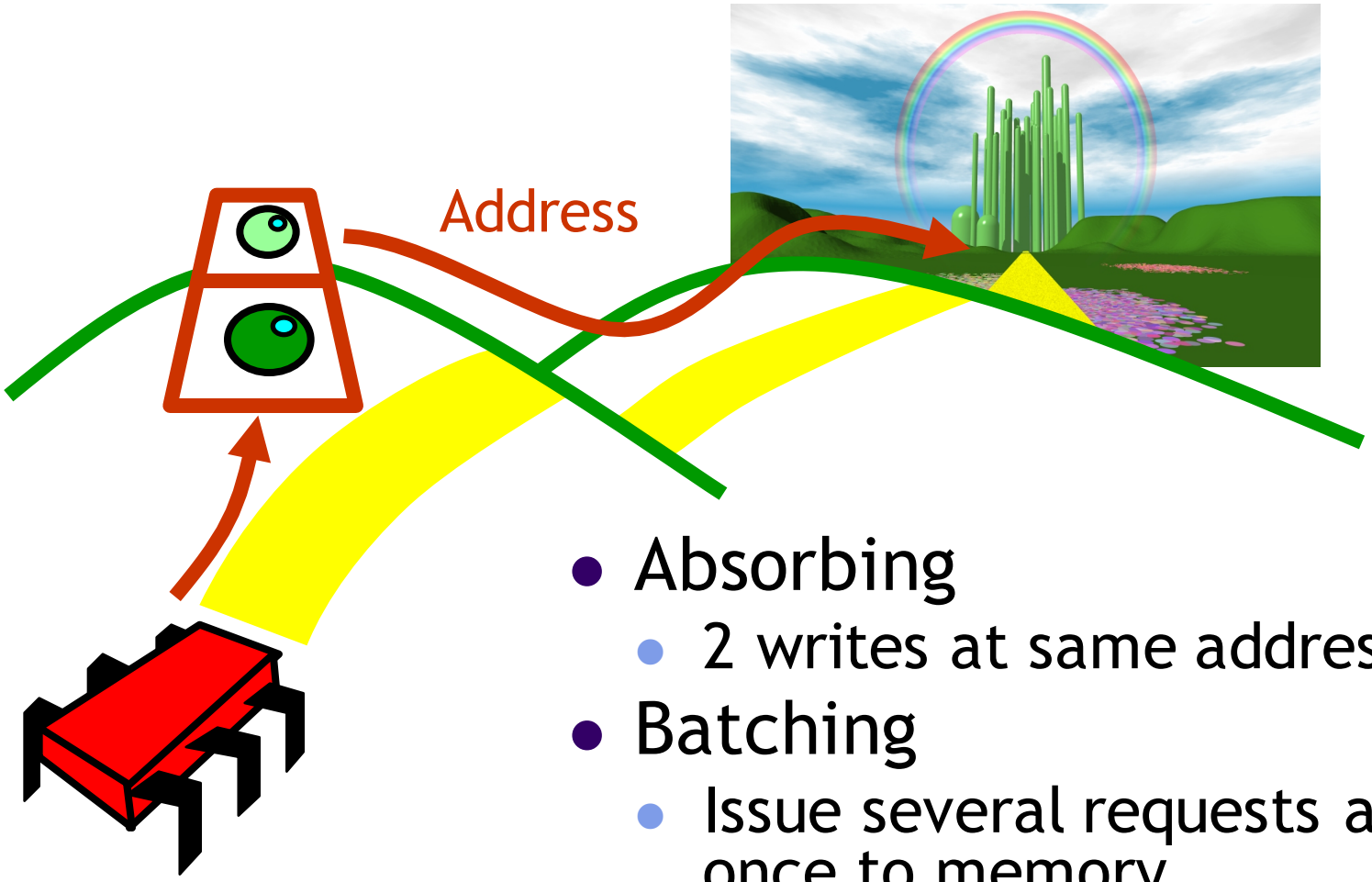
Relaxed Memory

- Remember the flag principle?
  - Alice and Bob's flag variables false
- Alice writes true to her flag and reads Bob's
- Bob writes true to his flag and reads Alice's
- One must see the other's flag true

Not Necessarily So

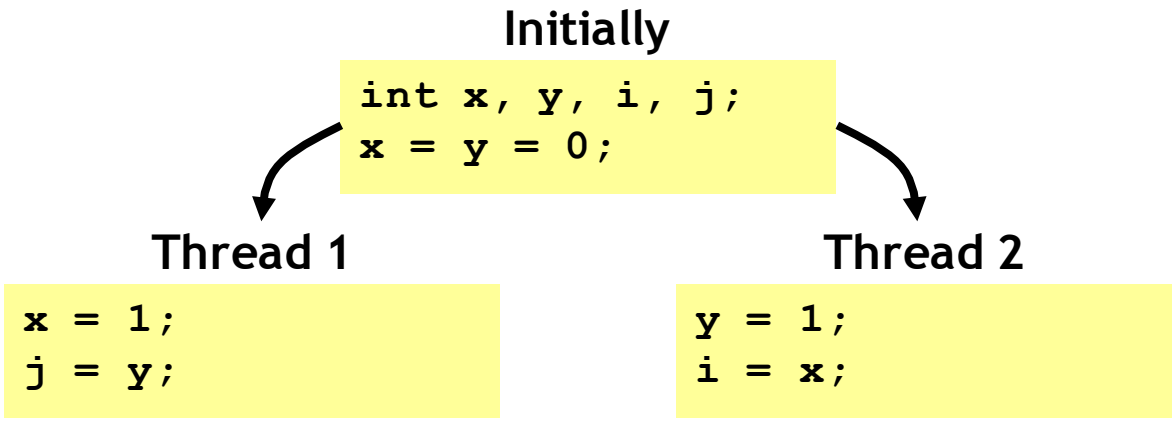
- Sometimes the compiler reorders memory operations
- Processors also have write buffers
- Can improve
  - Cache performance
  - Interconnect use
- But unexpected concurrent interactions

Write Buffers



- Absorbing
  - 2 writes at same address
- Batching
  - Issue several requests at once to memory

Memory Models



Value of `i` and `j`?

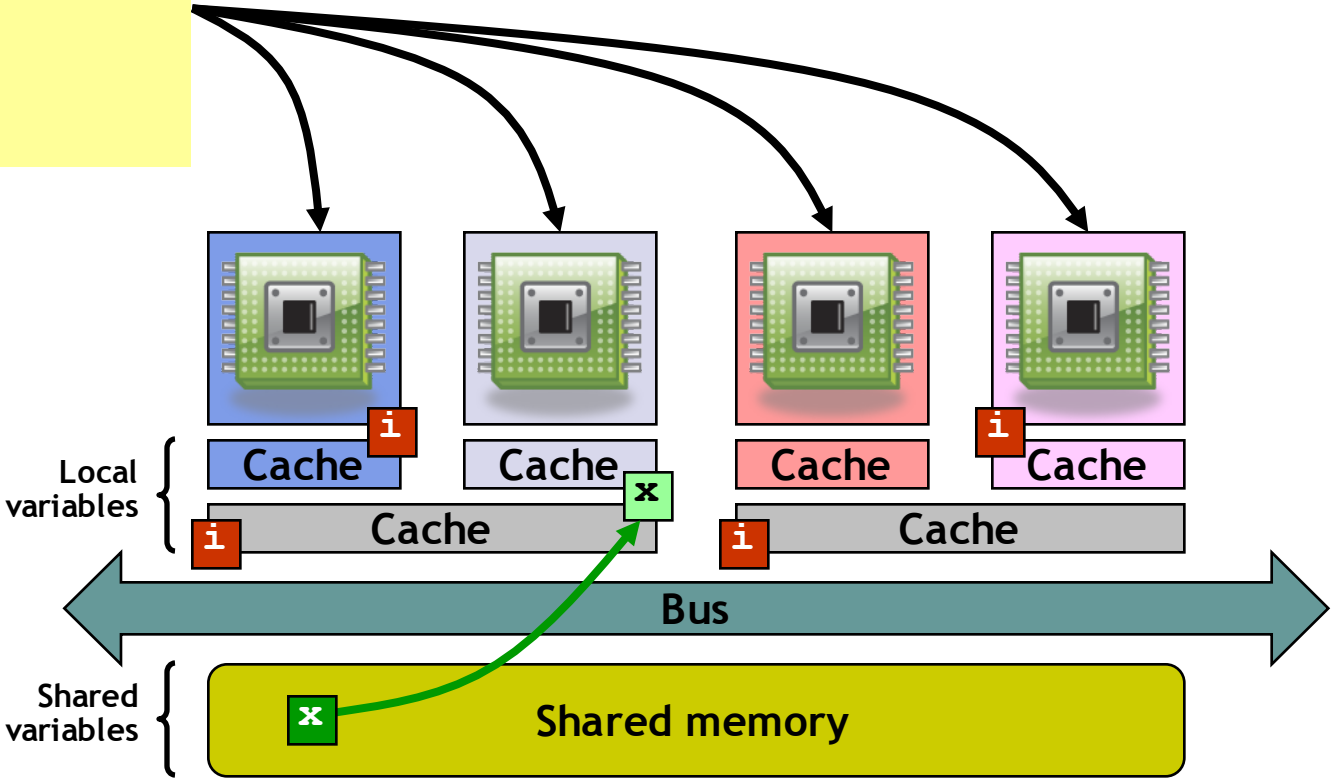
Can we have?
`i == 0 && j == 0`

In theory no; In practice yes because of reordering

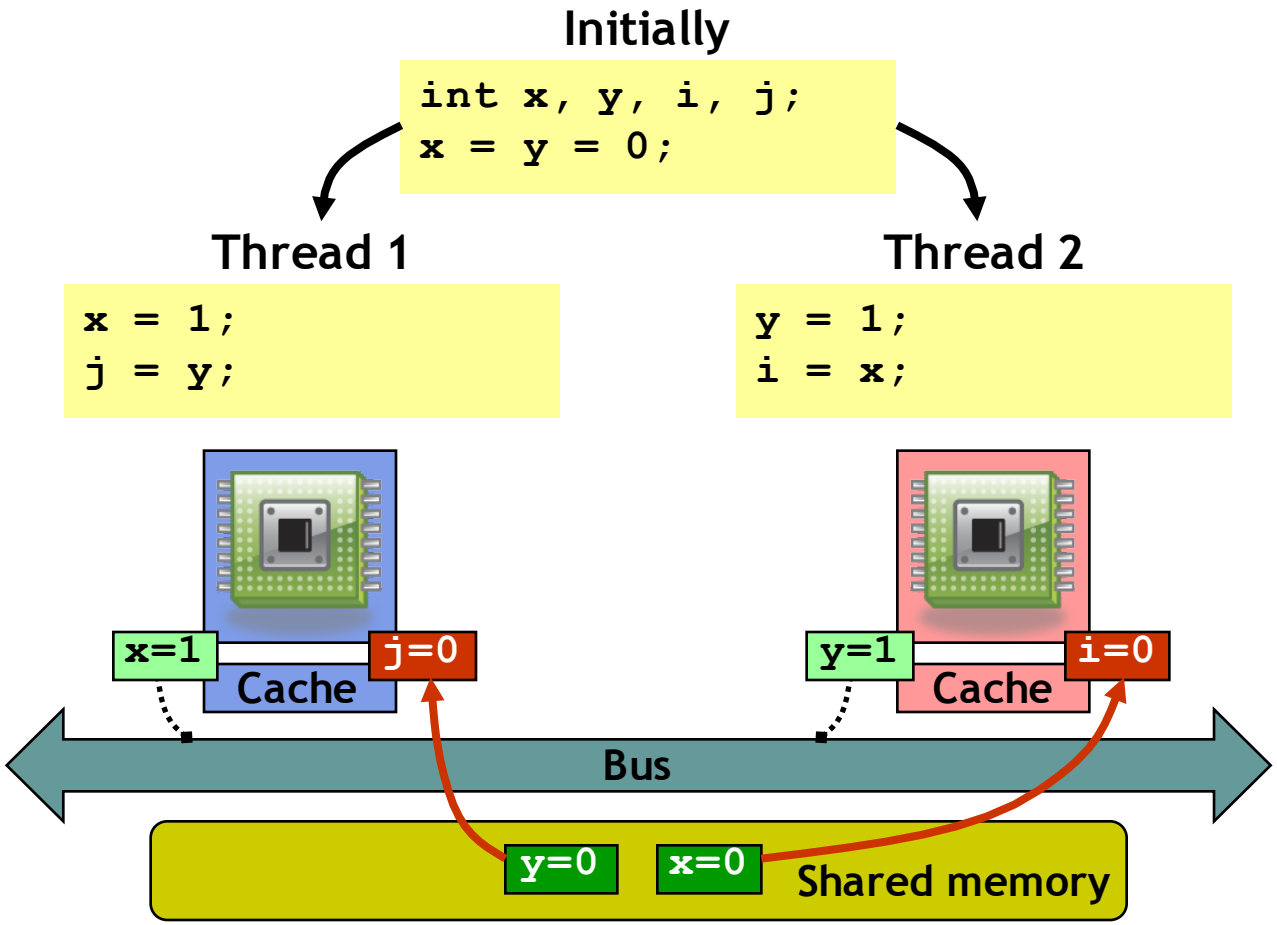
Where Things Fit

```
class SharedObject {
  int x;

  void f() {
    int i;
    …
  }
}
```



Where Things Fit



This is Not Wrong!

- We are assuming sequentially consistent behavior
  - But time is a relative dimension!
- Computers don't care about your intuition regarding time across multiple threads
  - Compilers, processors, caches can reorder instructions
- Preventing reordering must be explicitly requested
  - Using synchronization operations (lock/unlock)
  - Using memory barriers

Memory Barriers

- A processor can execute hundreds of instructions during a memory access
  - Why delay on every memory write?
  - Instead, keep value in register or cache

```
for (i = 0; i < 10000; i++)
  x += i;
```

Typically: `x` and `i` not written to memory at each iteration

- Memory barrier instruction (**expensive**)
  - Flush unwritten caches
  - Bring caches up to date
  - Added by compiler (synchronization, **`volatile`**)

Memory Barriers

- **LD1 L/L LD2**

  LD1's data loaded before data accessed by LD2 and all subsequent loads are loaded

- **ST1 S/S ST2**

  ST1's data visible to other processors before data associated with ST2

- **LD1 L/S ST2**

  LD1's data loaded before data associated with ST2 and all subsequent store instructions are flushed

- **ST1 S/L LD2**

  ST1's data visible to other processors before data accessed by LD2 and all subsequent loads are loaded

Memory Barriers Example (Java)

```
class X {
  int a, b;
  volatile int v, u;
  void f() {
    int i, j;

    i = a;
    j = b;
    i = v;
    // L/L
    j = u;
    // L/S
    a = i;
    b = j;
    v = i;
    // S/S
    u = j;
    // S/L
    i = u;
    j = b;
    a = i;
  }
}
```

The keyword `volatile` asks compiler to keep variable up-to-date and inhibits reordering & other optimizations

The compiler inserts memory barriers where necessary

RMW Atomic Operations

- Read-modify-write operation combines…
  - Read from memory
  - Modify value
  - Write to memory

  … atomically
- Supported by modern processors
  - Atomic increment/decrement, test-and-set, compare-and-set, etc.
- In Java: `java.util.concurrent.atomic`

Atomic-Inc/Dec

```
public class AtomicInteger {
  int value;

  public synchronized int
  incrementAndGet() {
    value = value + 1;
    return value;
  }
  public synchronized int
  decrementAndGet() {
    return --value;
  }
}
```

Package java.util.concurrent.atomic

Increment value

```
; x86
LOCK INC …
LOCK DEC …
LOCK XADD …
```

Decrement value (pre-decrement operator is not atomic!)

Get-and-Set

```
public class AtomicBoolean {
  boolean value;

  public synchronized boolean
  getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Set new value and return old value

```
; x86
LOCK XCHG …
```

Compare-and-Set

aka compare and swap

```
public class AtomicInteger {
  int value;

  public synchronized boolean
  compareAndSet(int expValue,
                int newValue) {
    if (value == expValue) {
      value = newValue;
      return true;
    }
    return false;
  }
}
```

Set new value and return true if old value matches expected value, return false otherwise

```
; x86
LOCK CMPXCHG …
```

The Power of RMW Operations

- Consensus number: maximum number of threads for which the object can solve the "consensus problem"
  Multiple threads that are proposing value have to agree on one
- Get-and-set is weaker than compare-and-set
  - Get-and-set: consensus number 2
    You can have no more than 2 threads agree on a value
  - Compare-and-set: consensus number ∞
- Can you:
  - Implement atomic-inc using compare-and-set?
    Exam question
  - Implement get-and-set using compare-and-set?
  - Implement compare-and-set using get-and-set?
    No, strictly weaker

Summary

- Hardware is more complex than our ideal model
  - Weaken consistency for performance
- Cache may lead to reordering of operations
  - Bypassing the cache (`volatile`) is expensive
  - Memory barriers to flush or update cache
  - Beware of false sharing!
- RMW operations helpful for synchronization
  - We will mainly use compare-and-set to implement lock-free/wait-free concurrent data structures