NTNU – Trondheim
Norwegian University of
Science and Technology

Cache coherence

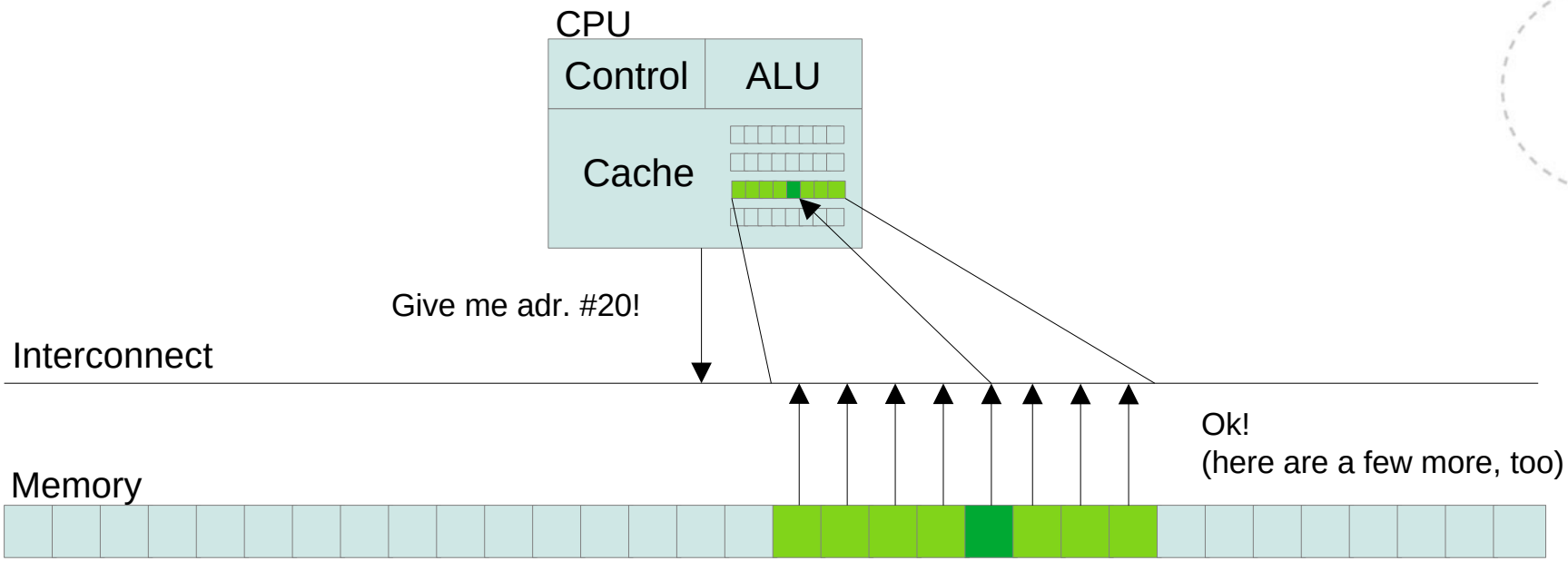
Jan.Christian.Meyer@ntnu.no

Our topic today

- Last time, we ended on a brief discussion of *false sharing*
- We skipped a few details about what happens when two cores are fighting for a cache line
  - The book actually touched upon this all the way back in Chapter 2
  - It is high time that we talk about it in lecture as well

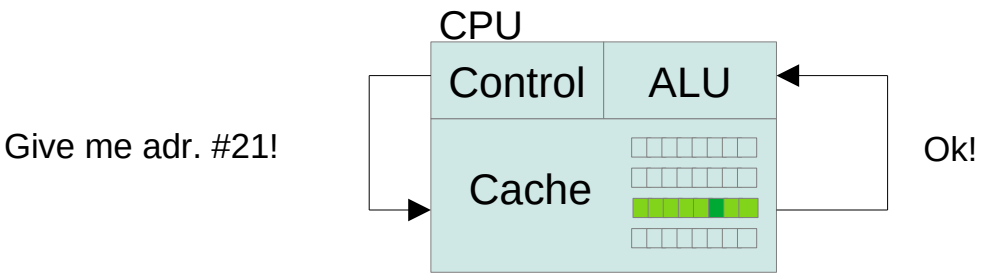
A quick recap

- When one CPU wants to read an un-cached value...



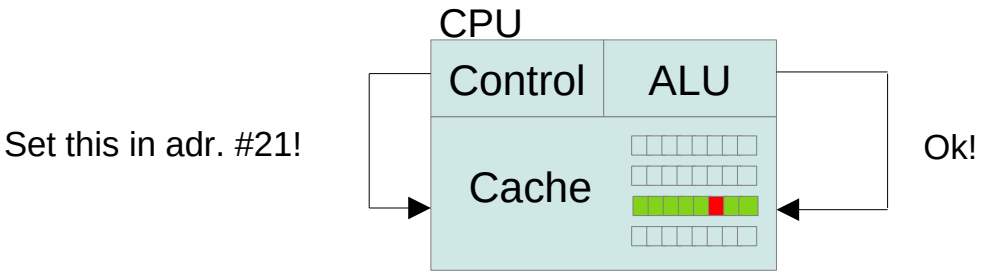
A quick recap

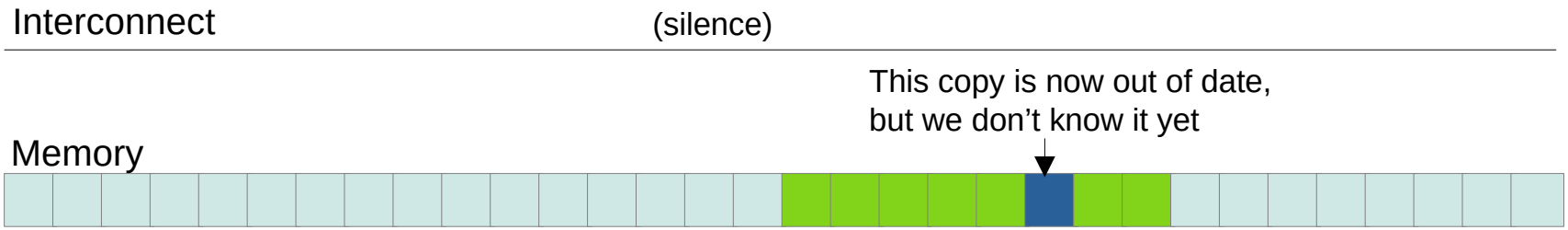
- When one CPU wants to read another (now cached) value...



Writing

- When one CPU wants to *write* a cached value...





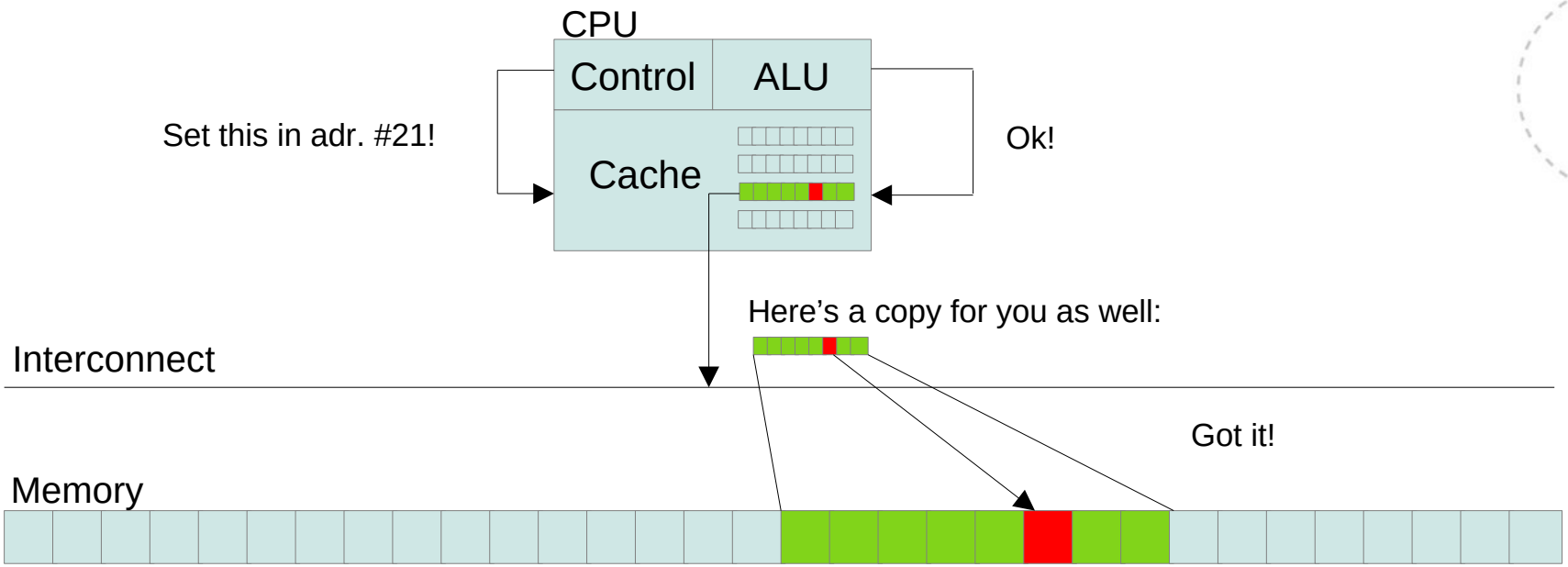
We have two options

- Push write operations straight through the cache memory, and update main memory as soon as possible
  - + Simple and inexpensive implementation
  - - May create constant memory traffic
- Delay write operations in memory, continue working with the cached copy
  - + Doesn't do unneccessary work
  - - Requires more complex circuitry and wiring

(it's a silmilar tradeoff to eager vs. lazy evaluation in programming languages)

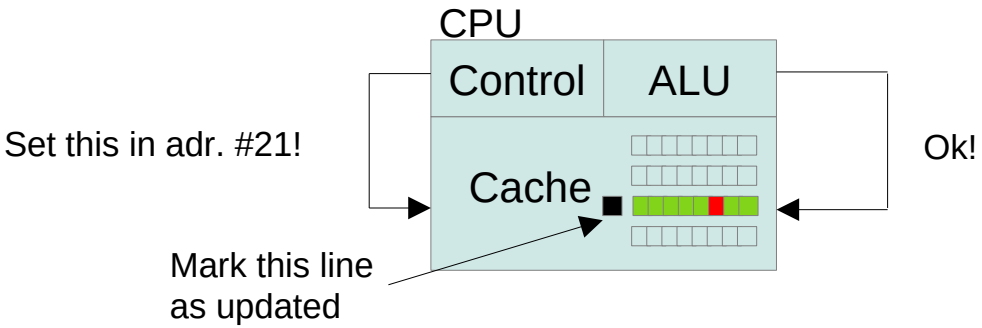
Write-through caching

- Write-through caches immediately pass their updates to main memory via the interconnect



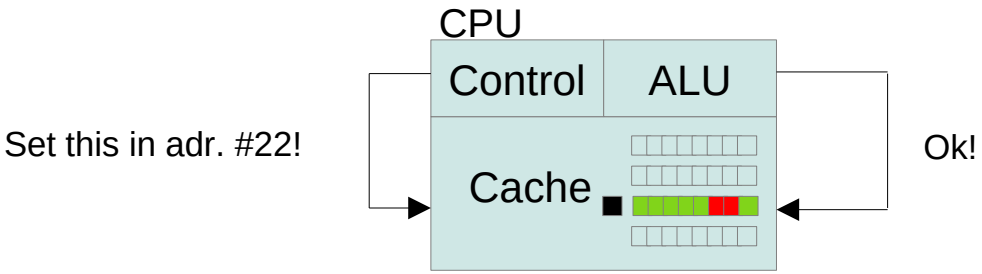
Write-back caching

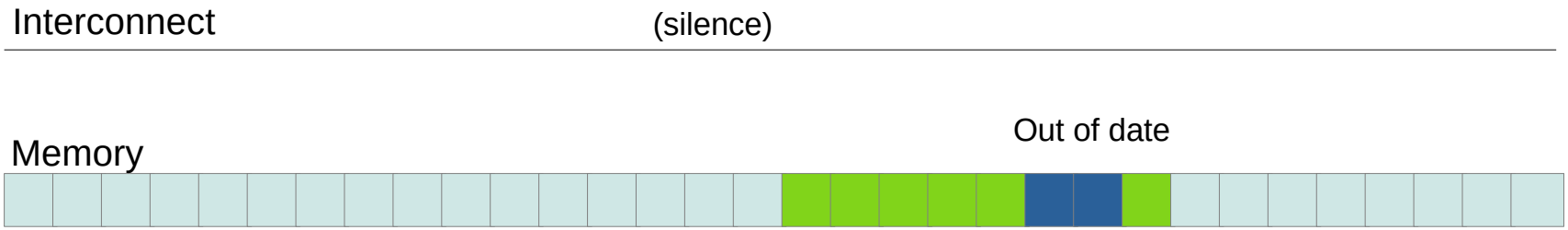
- Write-back caches retain their updates until it is convenient to report them back to main memory



Write-back caching

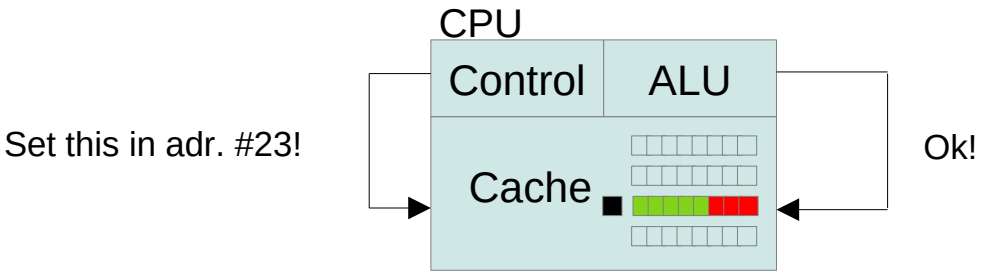
- Write-back caches retain their updates until it is convenient to report them back to main memory

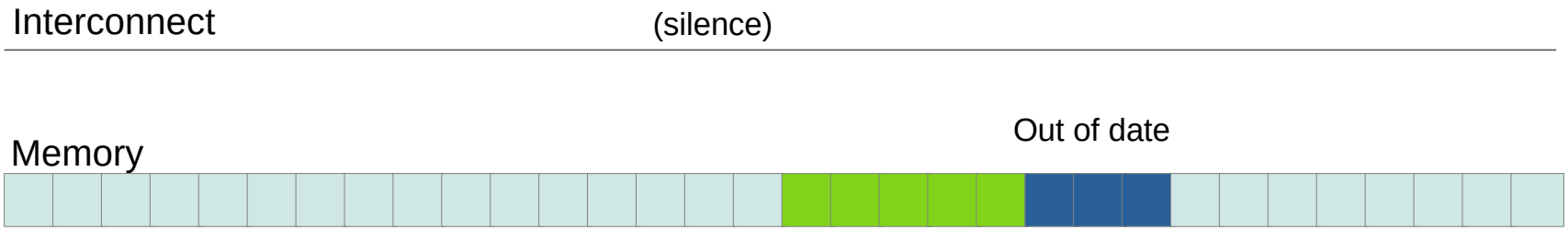




Write-back caching

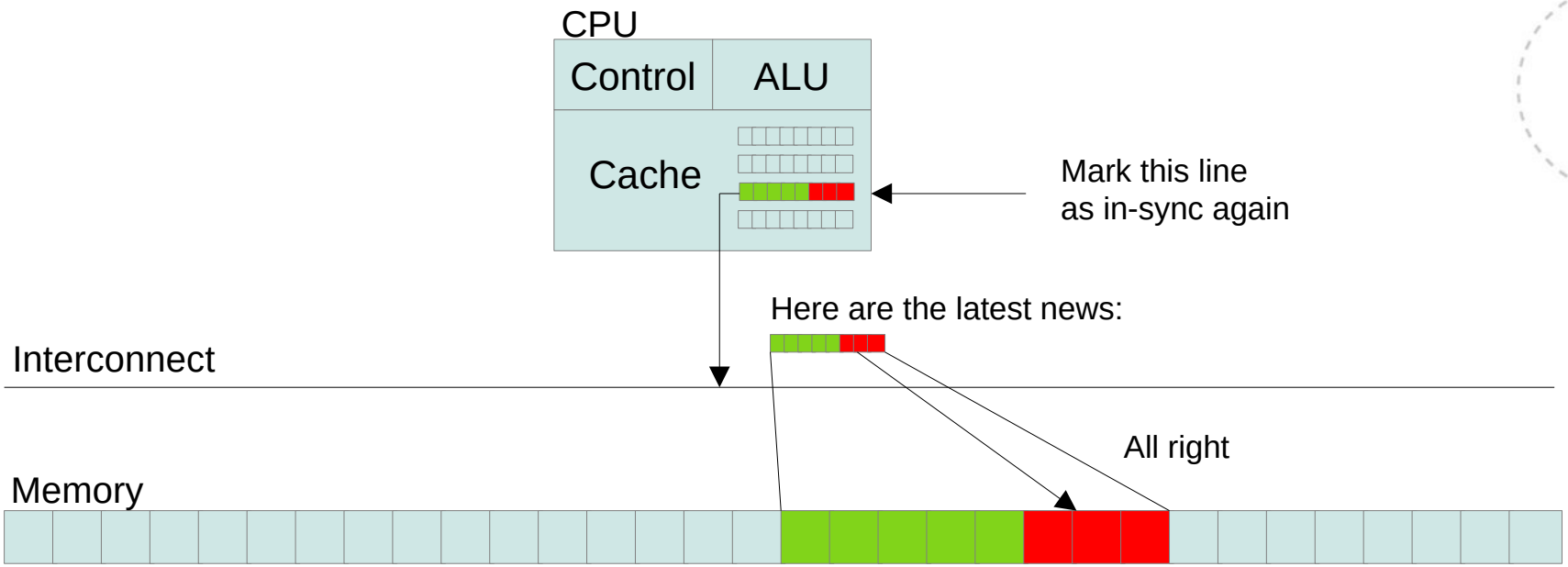
- Write-back caches retain their updates until it is convenient to report them back to main memory





Write-back caching

- Write-back caches retain their updates until it is convenient to report them back to main memory

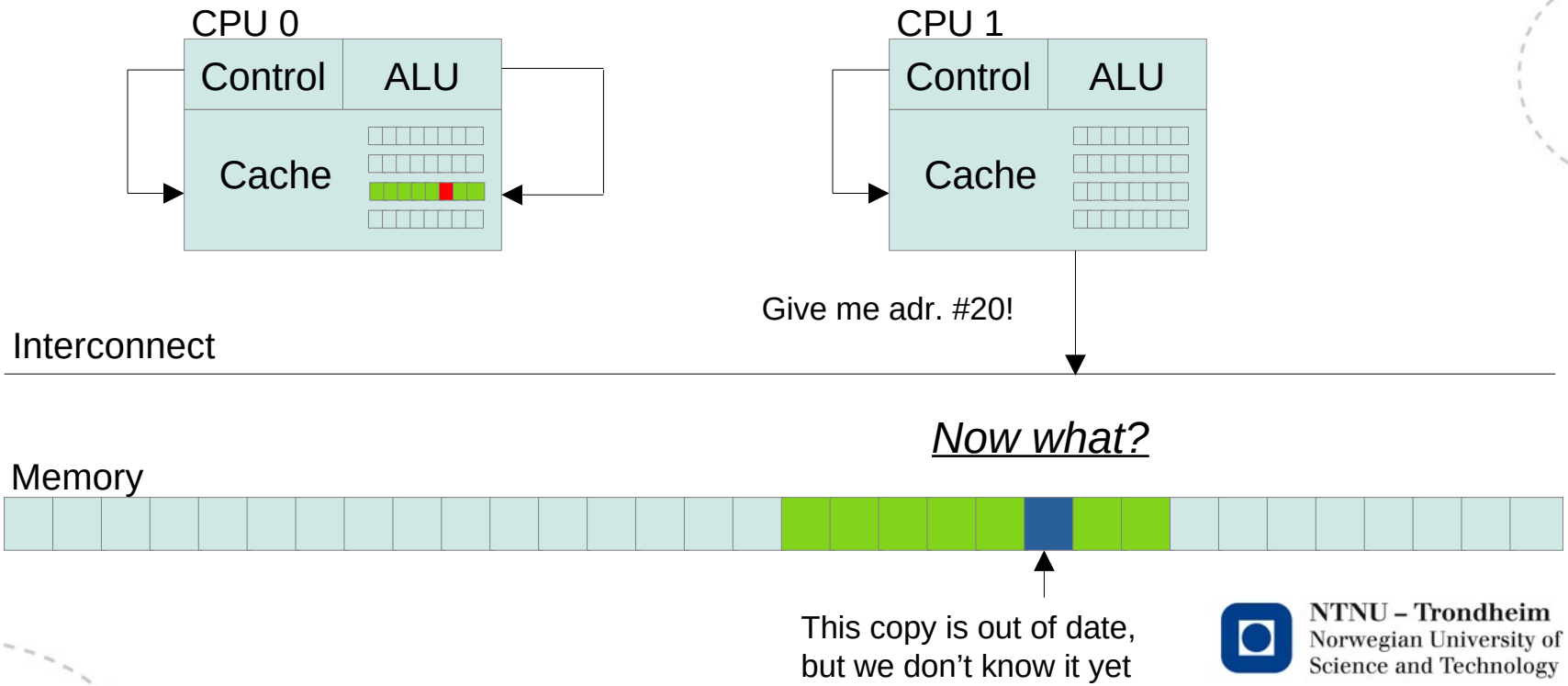


This is fine for *one* CPU

- It doesn’t matter if main memory reflects the cache state
  - ...as long as we have the cached copy to work with anyway, memory can update at its own leisurely pace
  - We’ll have to wait for it to catch up at I/O operations and other things that have direct memory access
  - That’s ok, those are slower than memory anyway

It’s less fine for *many* CPUs

- Effectively, the other CPUs are devices with direct memory access that work at the same speed:



Cache coherence

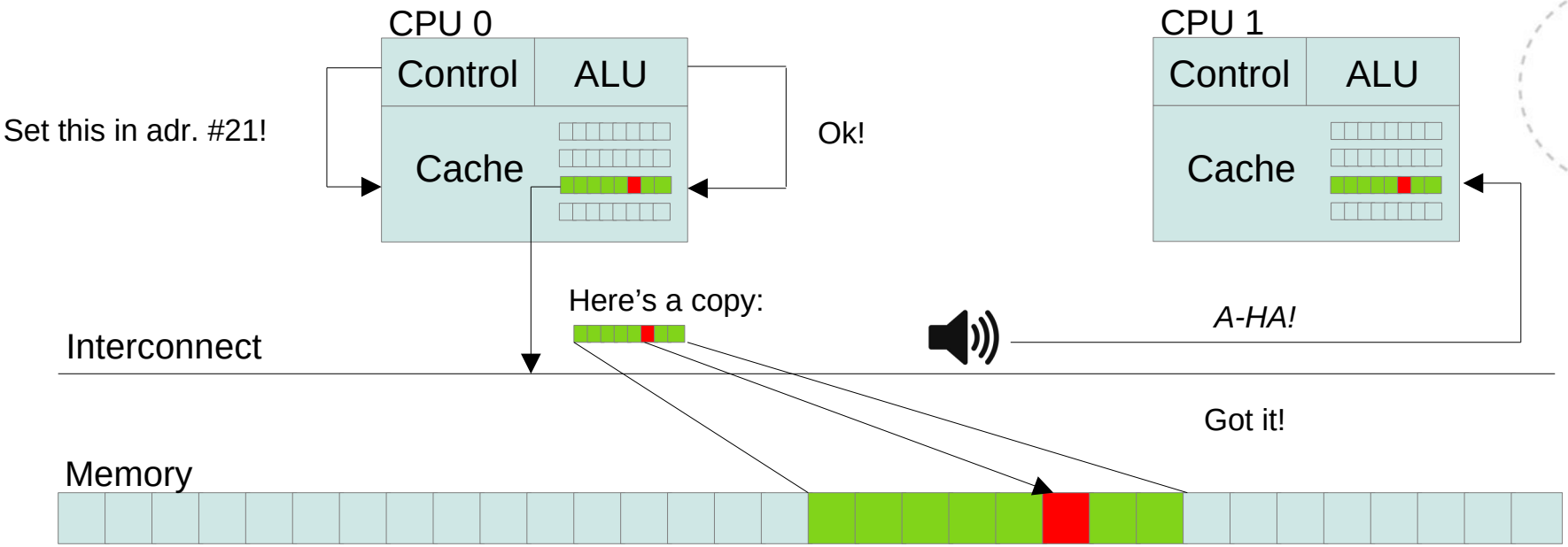
- The issue of keeping multiple caches up to date with each others' modified values is called the *cache coherence* problem
- Its solutions fall into two categories
  - *Snooping*

    i.e. allowing CPUs to listen in on each others' memory traffic via shared branches of the interconnect
  - *Directory*

    i.e. maintaining a centralized registry of cache lines and the various states they are in

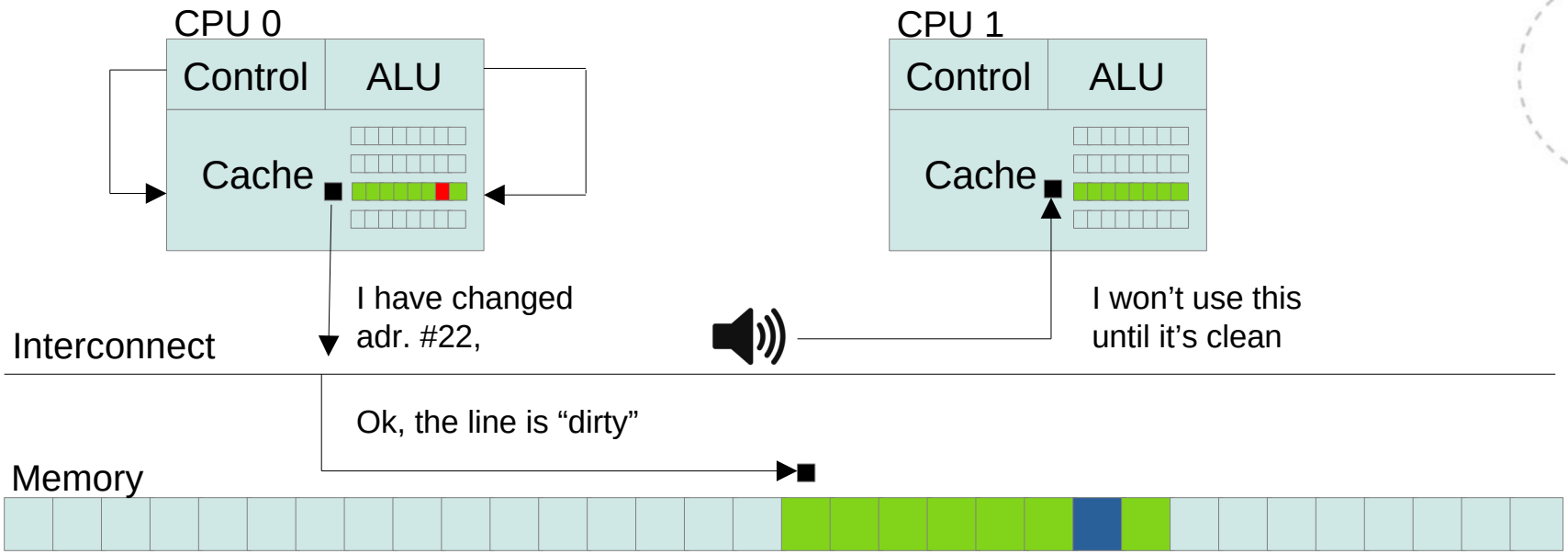
Snooping
(with write-through)

- This approach comes from machines with a single, shared memory bus as interconnect



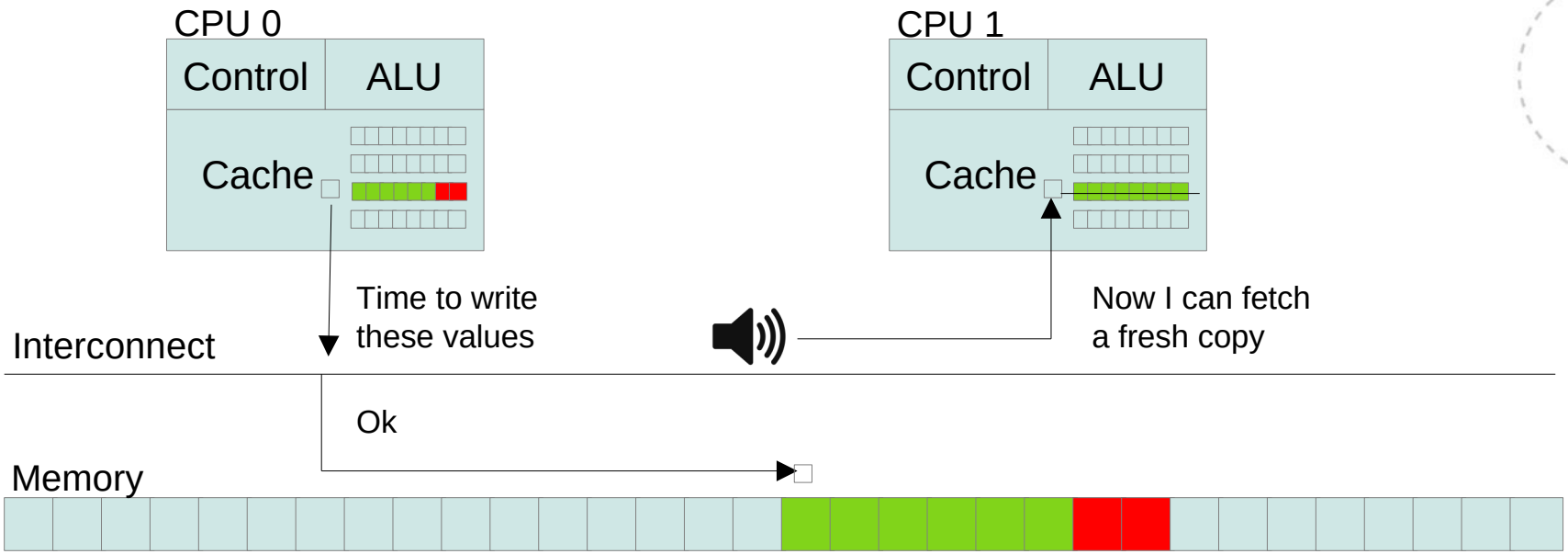
Snooping
(with write-back)

- While the cache line is out of date, only 1 bit needs to be broadcast



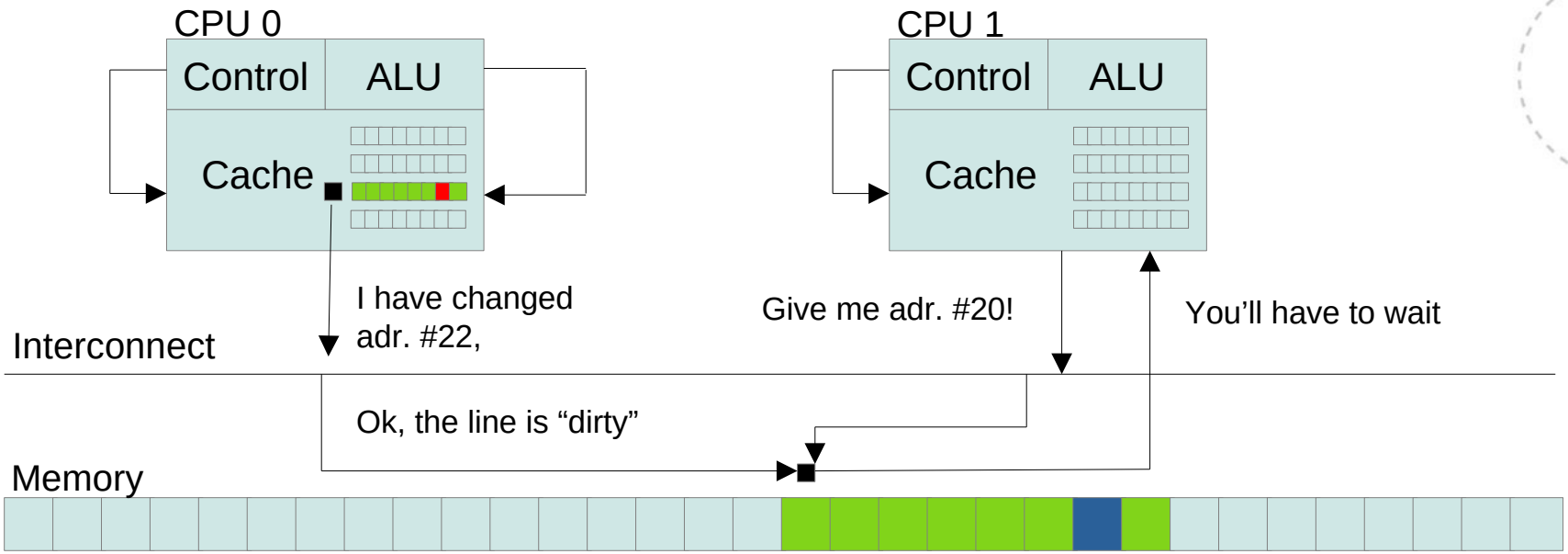
Snooping
(with write-back)

- While the cache line is out of date, only 1 bit needs to be broadcast



Snooping
(with write-back)

- If CPU 1 wasn't listening, the 'dirty bit' still tells us that the line hasn't been written back to memory



The issue with snooping

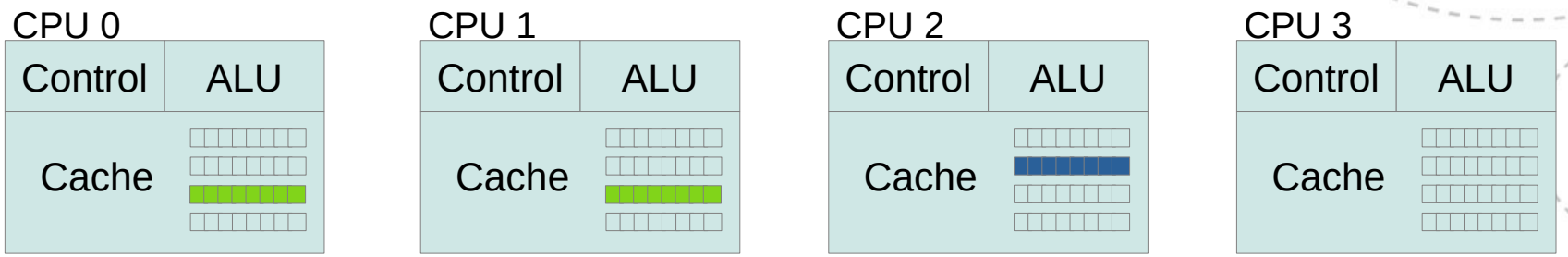
- Even though it's easiest to think of that way, snoopy coherence protocols don't actually require a single, shared bus
  - They just need the ability to make broadcast operations
- We've looked at the cost of broadcast operations
  - When the interconnect has a large, multi-level structure, their cost increases with the number of participants
- Snooping is fast...
  - ...but only for numbers that are small enough for efficient broadcast

Directory

- If we can afford more than 1 additional bit of information per cache line, we can make a table
  - To record which processors have copies of what memory
- It will need
  - an entry for each memory block we want to track
  - a few bits to record its state

    (exclusive-or-shared, modified, uncached)
  - a bit vector with one bit for each processor

A directory state



Directory

Block	cpus	state
[0:7]	0010	Excl.
[8:15]	0000	uncached
[16:23]	1100	Shared
[24:31]	0000	uncached

Directory granularity

- As you can see, directory entries for every tiny cache line lead to a spectacularly large table
  - That's a directory with 'full bit vector' format
  - Table size grows as (memory amount) x (nr. of cpus)
- For bigger systems, we can divide memory in bigger chunks
  - Directory bit vector now indicates that "one or more out of cpus 0-7 has a (modified) copy"
  - Those 8 can sort out coherence between themselves, using another mechanism
  - It is still clear where to send the request when other cpus want something from the memory block
  - That's a directory with 'coarse bit vector' format

The directory requires memory

- Some systems allocate a fixed part of general system RAM for use as directory
  - Hardly noticeable for small/medium scale systems
  - Leads to upset customers of large systems, when they discover that they can't allocate all the RAM they purchased
- Other systems require the installation of additional memory banks just for the directory
  - Customers aren't upset when the cost is made visible, but systems grow more expensive

A recurring theme

- Moving data takes (much) more time than carrying out operations
- The further it has to move, the longer it takes
  - Message passing can reach very far, but it's expensive
  - Shared memory at laptop scale looks cheaper, because all the memory sits (relatively) close to the cores
  - When the shared memory systems grow bigger, they develop the same issues with cost and distance
- High performance computing is like real estate business
  - It's all about location, location, and location