Multicore Architectures

Week 1, Lecture 2

VirginiaTech
Invent the Future

Multicore Landscape

- Intel
  - Dual and quad-core Pentium family.
  - 80-core demonstration last year.
- AMD
  - Dual, triple (?!), and quad-core Opteron family.
- IBM
  - Dual and quad-core Power family
  - Cell Broadband Engine
- Sun
  - (Multithreaded) Niagara: 8 cores (64 threads).

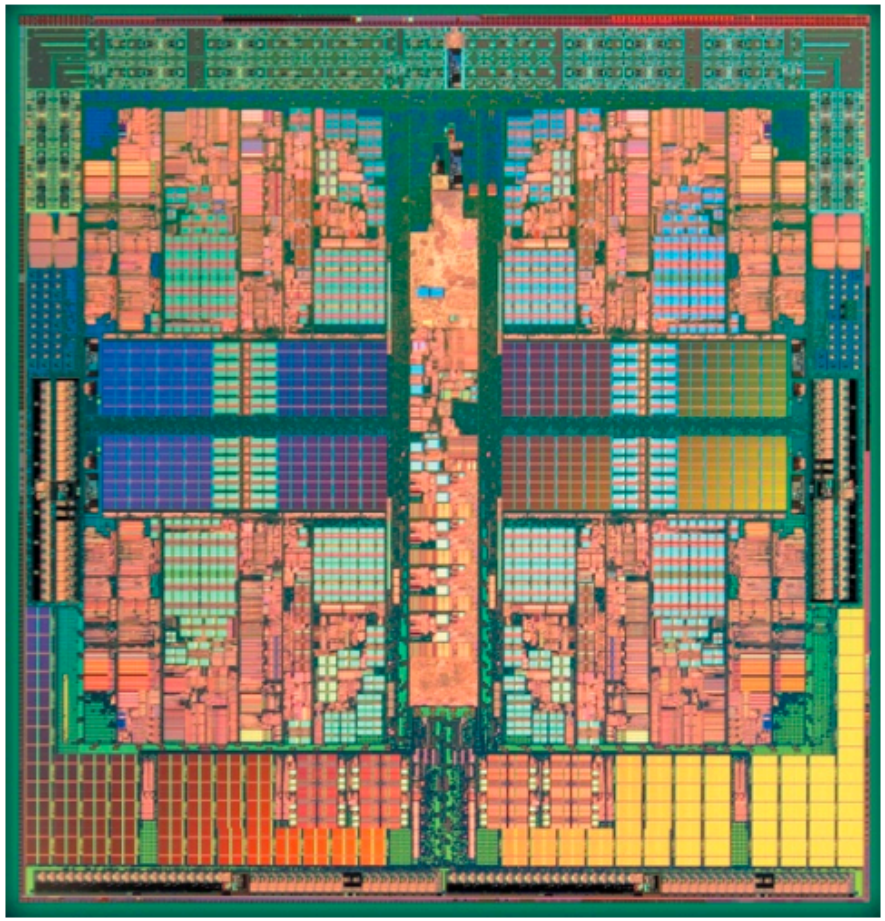
Multicore Classes

- Homogenous Multicore
  - Replication of the same processor type on the die as a shared memory multiprocessor.
  - Examples: AMD and Intel dual- and quad-core processors
- Heterogeneous/Hybrid Multicore
  - Different processor types on a die
  - Example: IBM Cell Broadband Engine
    - One standard core (PowerPC) + 8 specialiced cores (SPEs)
  - Rumors: Intel and AMD to follow suit.
    - Combining GPU-like processors with standard multicore cores

Homogeneous Multicore

- The major mainstream chips are all of this form
  - Examples: Intel, IBM, Sun, AMD, and so on.
- Take a single “core”, stamp it out lots of times on a die.
- This will work for now, but it likely will *not* scale to high core counts. Why?

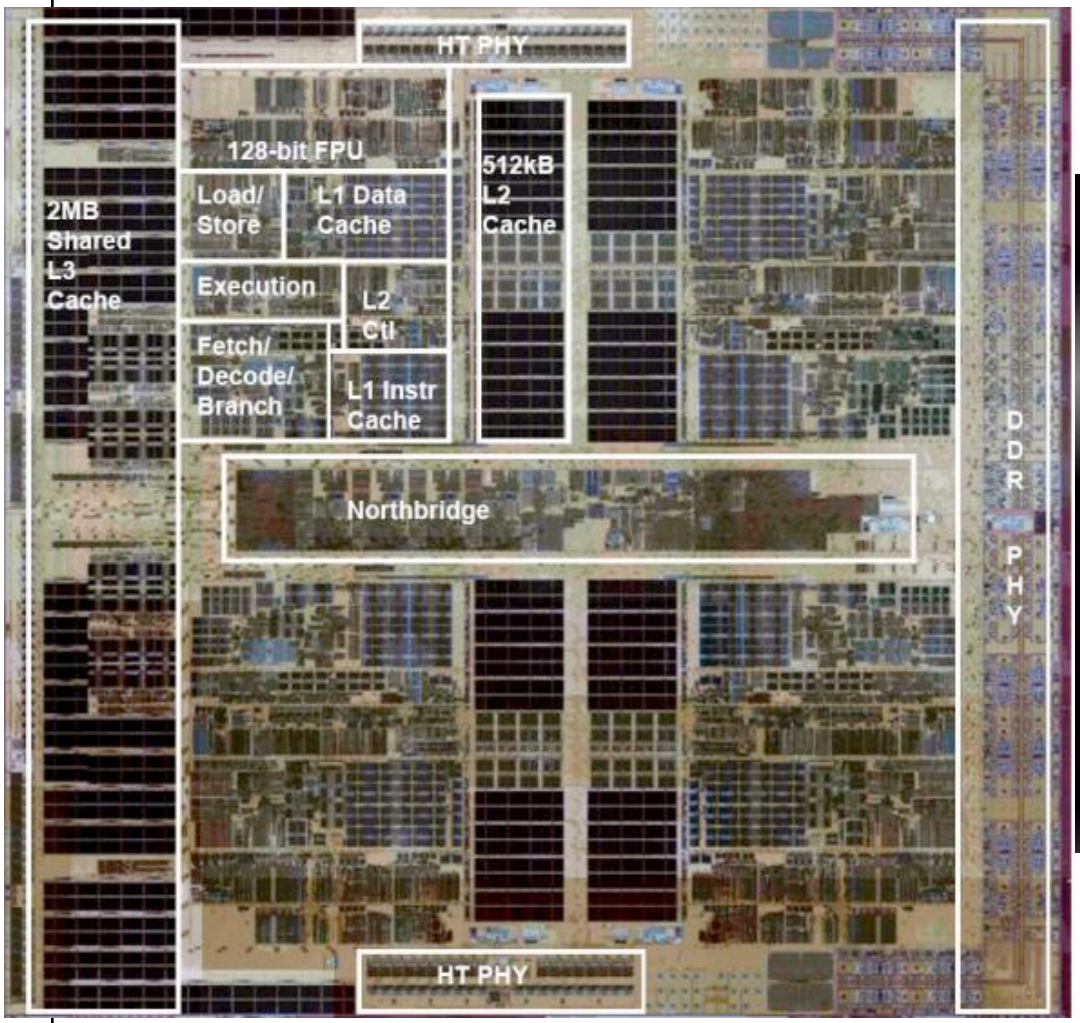
AMD Dual-, Triple-, and Quad-Core

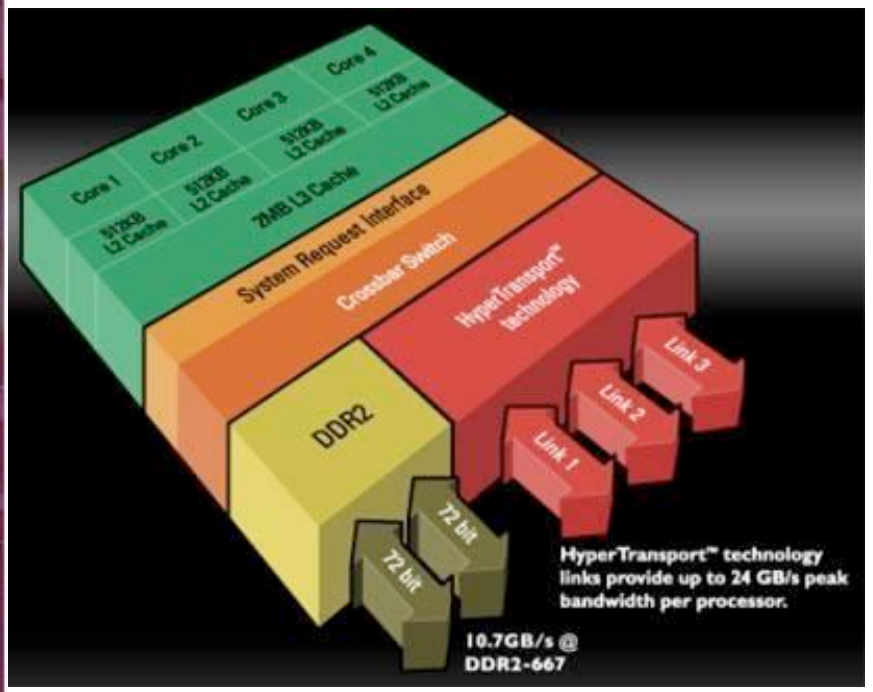


AMD Triple-Core?

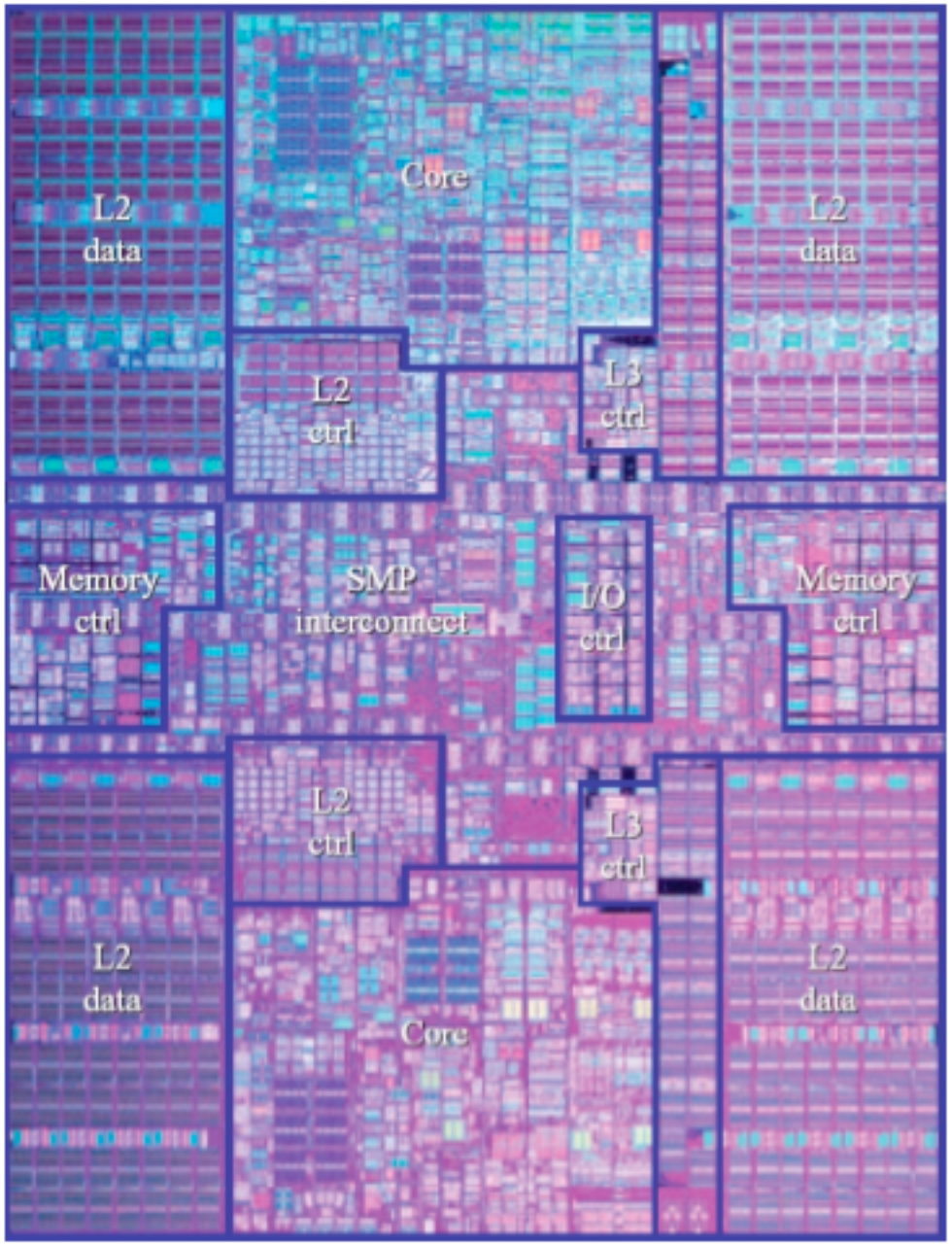
- Actually quad-core processors with one core disabled.
- Manufacturing defects that kill one core but leave the rest functional would be thrown out as failed quad core.
  - Why not resell them as triple core?
- Not a new idea …
  - Lower clock-rate versions of processors are identical to their higher clock-rate siblings but were part of a batch that failed to meet certain manufacturing tolerances but were otherwise fine.

AMD Rev. H Quad-Core



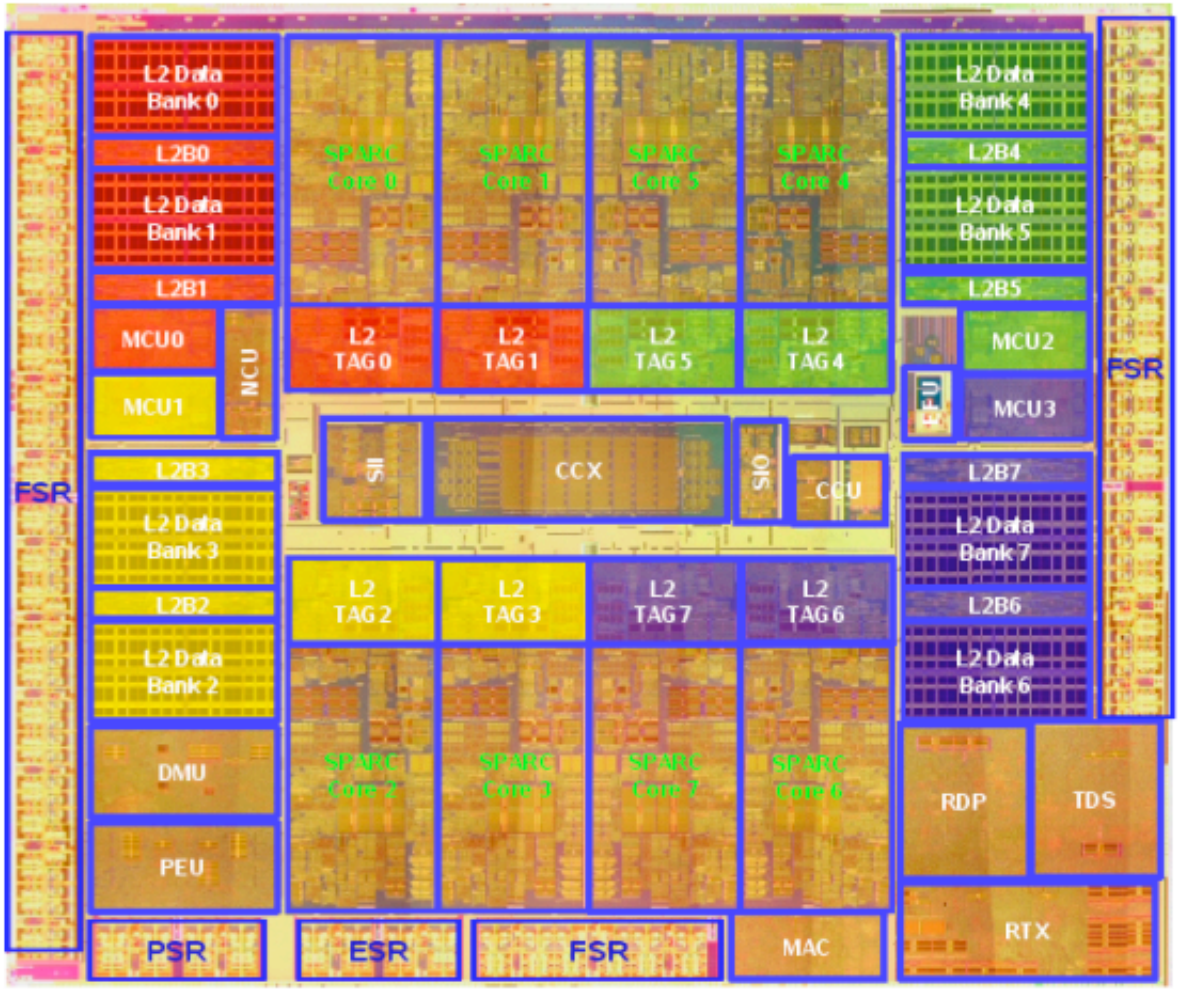


IBM: Power6

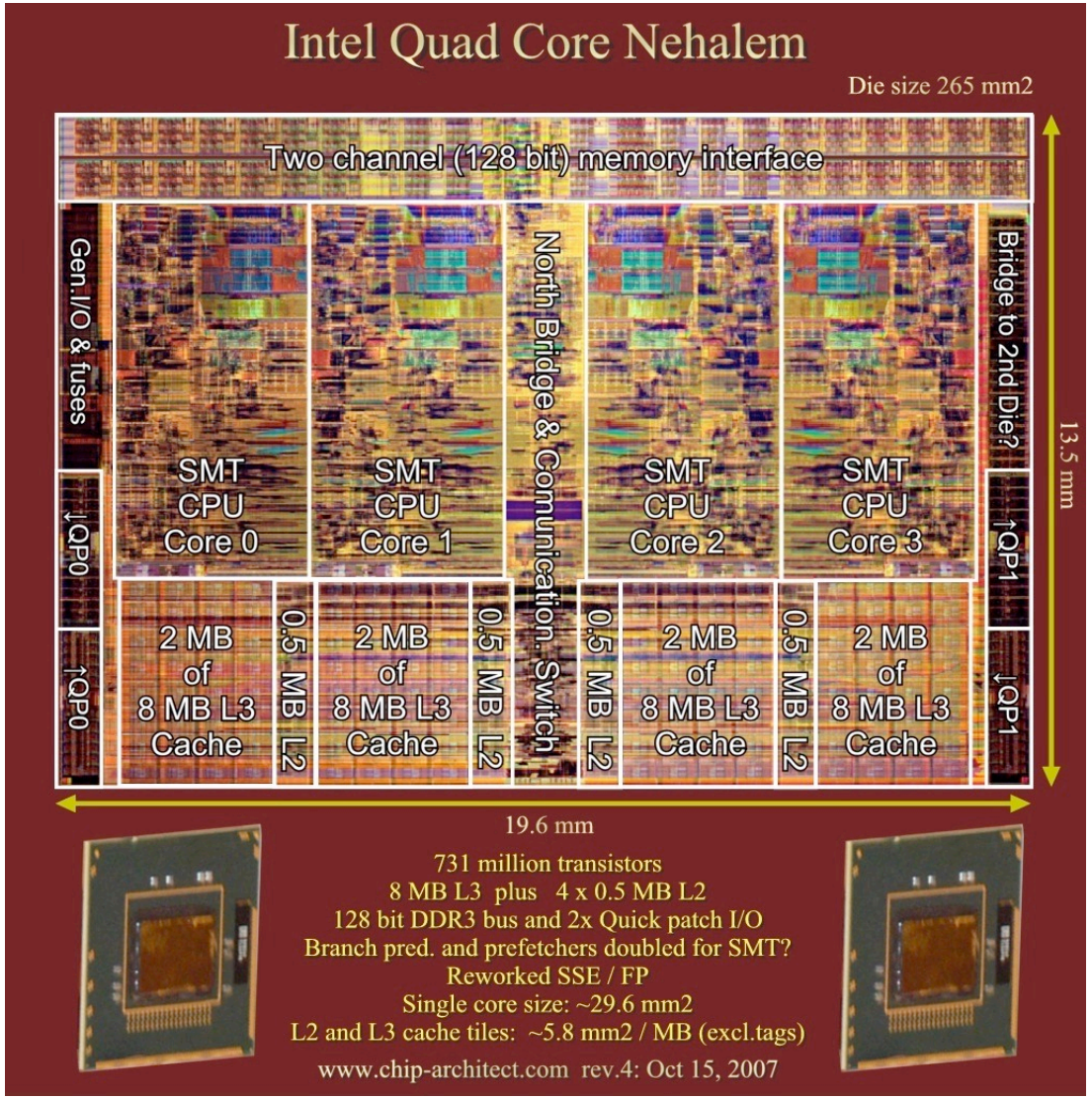


Sun Niagara

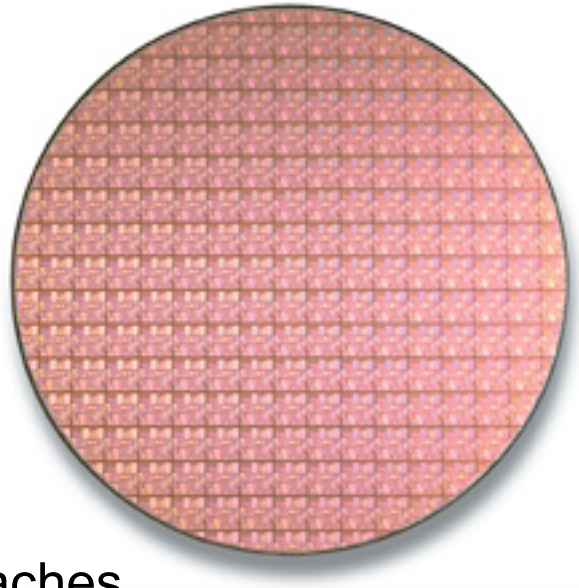
8 cores, 8 threads per core = 64 threads



Intel Nehalem



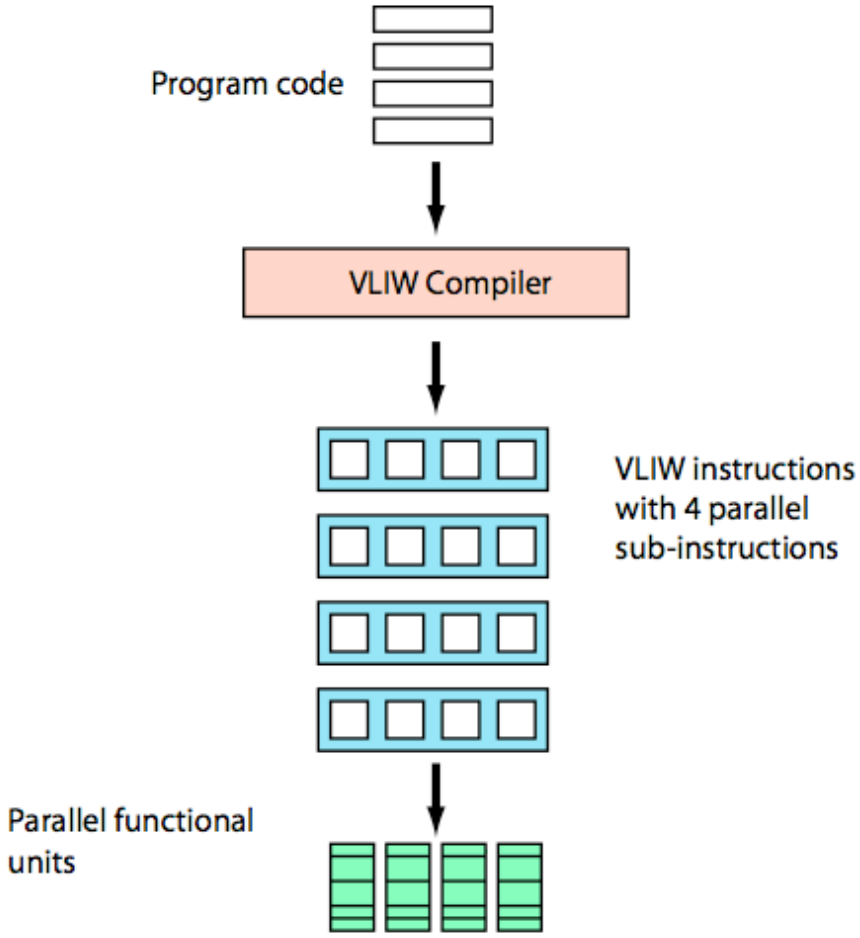
Dies



- Observations
  - Core replication obvious.
  - Big chunks of real estate dedicated to caches.
  - Increasing real estate dedicated to memory and bus logic. Why?
  - In the past, SMPs were usually 2 or 4 CPUs, so this logic was on other chips on the motherboard. Even with big fat SMPs, the logic was off on a different board (often this was what the $$$ went for in the expensive SMPs.)

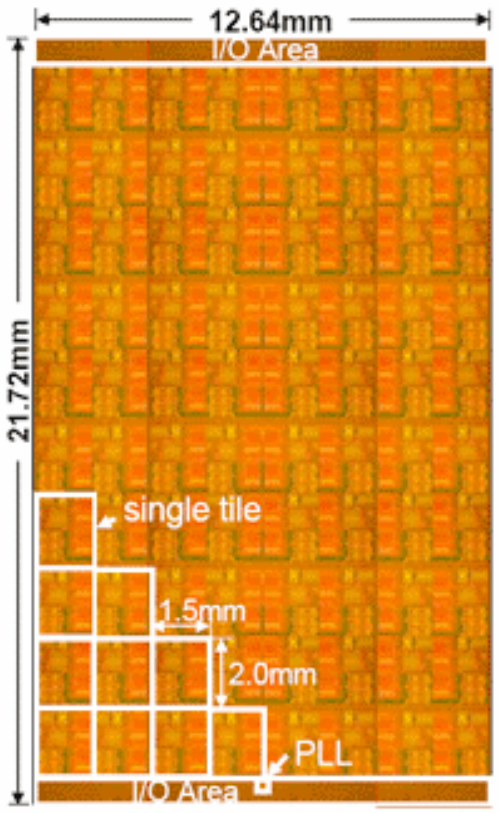
Wait a Minute! There's More!

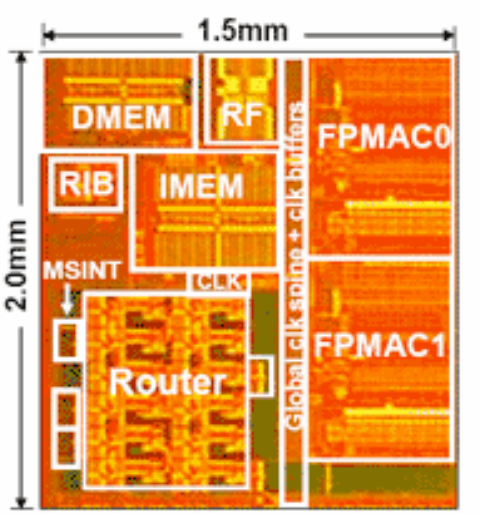
- Itanium processors are based on the concept of VLIW (Very Long Instruction Word).
  - Each instruction is a 128-bit word composed of four 32-bit instructions to be executed concurrently.
  - Difficulty of identifying parallelism is left to the compiler instead of hardware or programmer.
  - VLIW dead with "Itanic"?



VLIW Lives!

- VLIW *not* dead.
- Intel Terascale demo processor
  - 80 VLIW cores instead of 80 superscalar cores similar to those found in the Core2 architecture more commonly found today.





Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm²
Tile area	3mm²
Package	1248 pin LGA, 14 layers, 343 signal pins

Multicore: Present

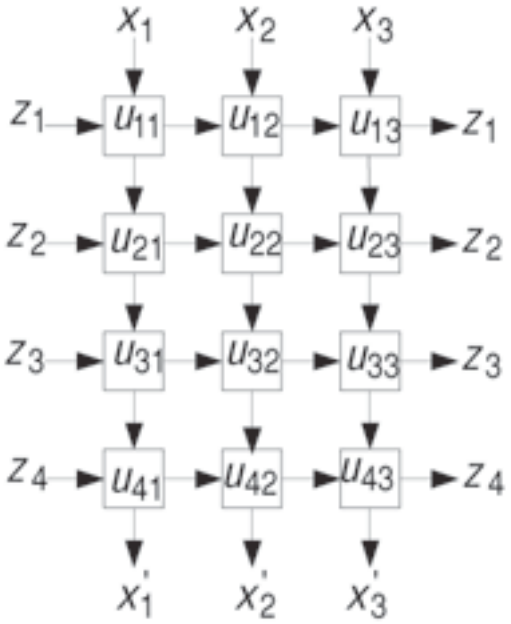
- Core count is increasing.
- Operating systems schedule processes out to the various cores in the same way they always have on traditional multiprocessor systems.
- Applications get increased performance for free by reducing the # processes per core yielding a decrease in contention for processor resources.

Resource Contention

- Assume each process or thread has its own CPU.
  - Example: The little clock in the corner of the screen has its own CPU.
- Result: Massive contention for resources when simultaneous threads all "fight" for off-chip resources.
- Process preemption when doing something slow, e.g., going to memory. Other useful work can occur.
- If a processor exists for each process, everyone could proceed at the same time yet will still be fighting for a bus that is *not* growing at the same rate as the core count. Same with caches.

Blast Back to the Past … 70s and 80s

- Network topology will matter again!
  - Mesh-based algorithms, hypercubes, tori, etc.
  - 80-core Intel Terascale demo processor had a huge component related to interconnection topology.
  - Systolic algorithms are making a comeback!



Reminder: Reading Assignment

- Brian Hayes, "Computing in a Parallel Universe," *American Scientist*, November-December 2007. http://www.americanscientist.org/issues/pub/computing-in-a-parallel-universe
- Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, 30(3), March 2005. http://gotw.ca/publications/concurrency-ddj.htm

Why Memory Matters

Yes for “traditional multicore” …

No for “emerging multicore” …

VirginiaTech
Invent the Future

Memory Architecture in Multicore

- As you saw in one of the readings …
  - The cache is still a key performance feature.
    - … but interesting ideas exist that will turn the usual memory architecture upside down.
    - … hence, we look at traditional memory architecture so we can compare and contrast it with emerging memory architecture of the Cell, GPGPU, and so on.

Caches: The Multicore Feature of Interest

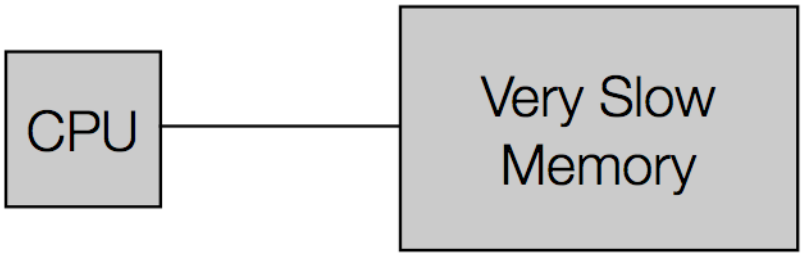
- Introduced in the 1960s as a way to overcome the "memory wall".
  - Memory speeds did not advance as fast as the speed of functional units.
  - Consequence: Processor outruns memory, leading to decreased utilization.
    - Utilization = $(t_{total} - t_{idle}) / t_{total}$
- What happens when you go out to main memory? Idle time.
  - Decreased utilization = less work per unit time.
  - This costs money because time=$$ … so time doing nothing is $$$ wasted.

Idle Time

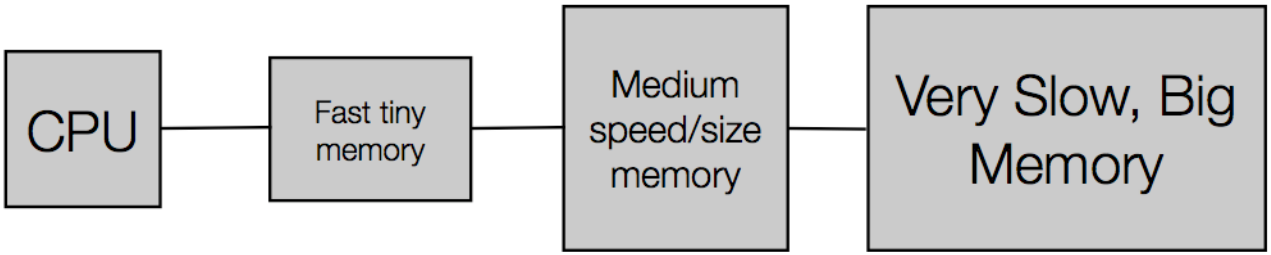
- Caches were *not* the sole fix for idle time.
  - Preemptive multitasking and time sharing were actually the dominant methods in the early days.
  - But every program inevitably has to go out to memory, and you do not always have enough jobs to swap in while others wait for memory.
  - Plus, do you really want to be preempted every time you ask for data? Of course not!
- So, caches important (for now :-) …

Caches: Quick Overview

- Traditional Single-Level Memory

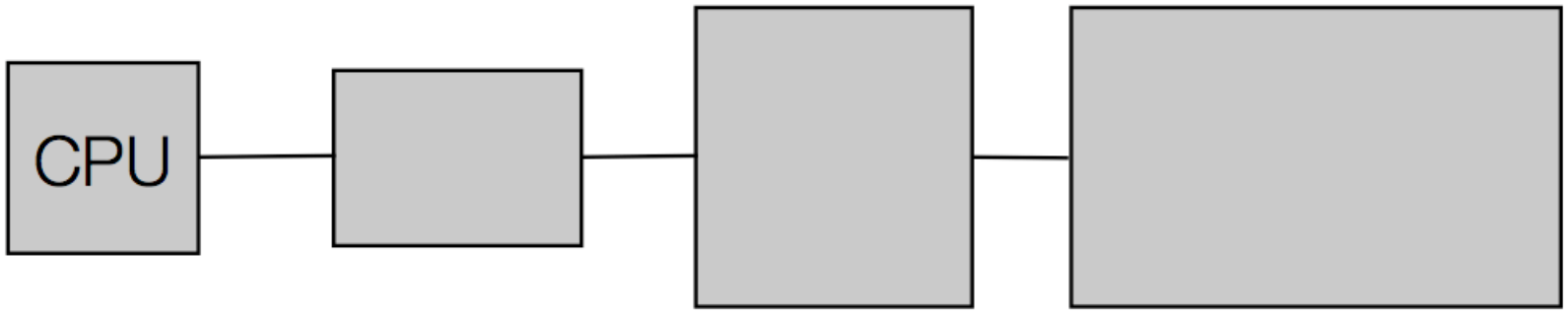


- Multiple Memory Levels



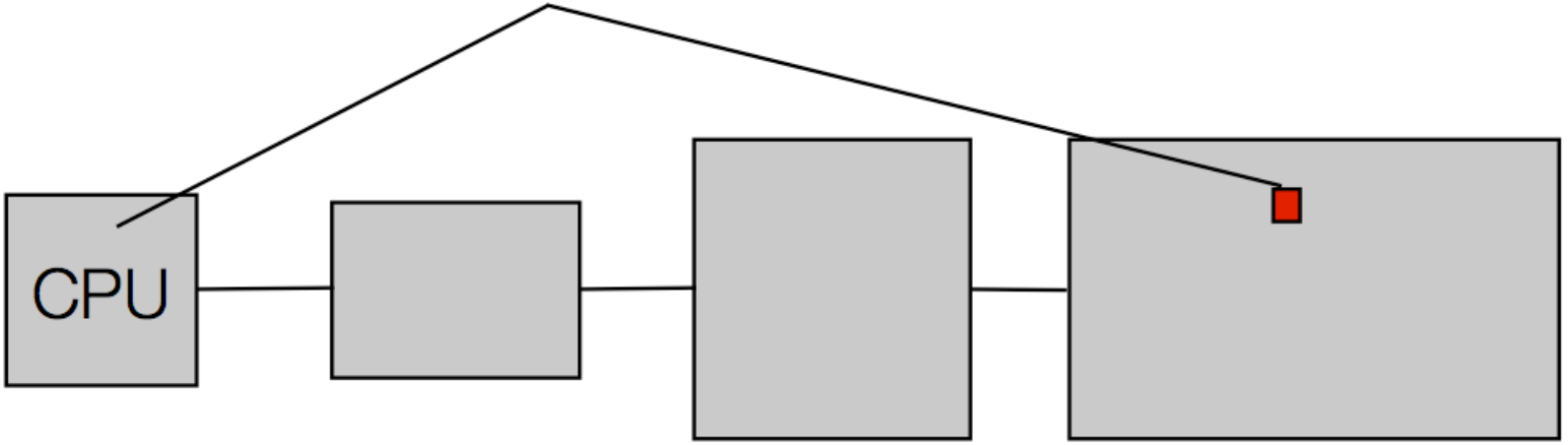
Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



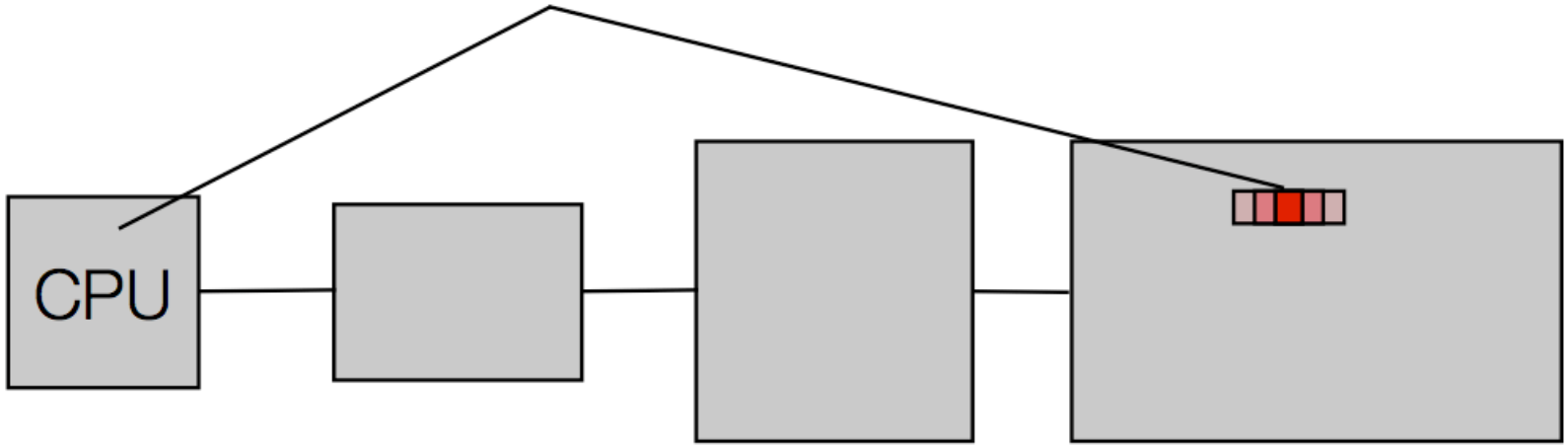
Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



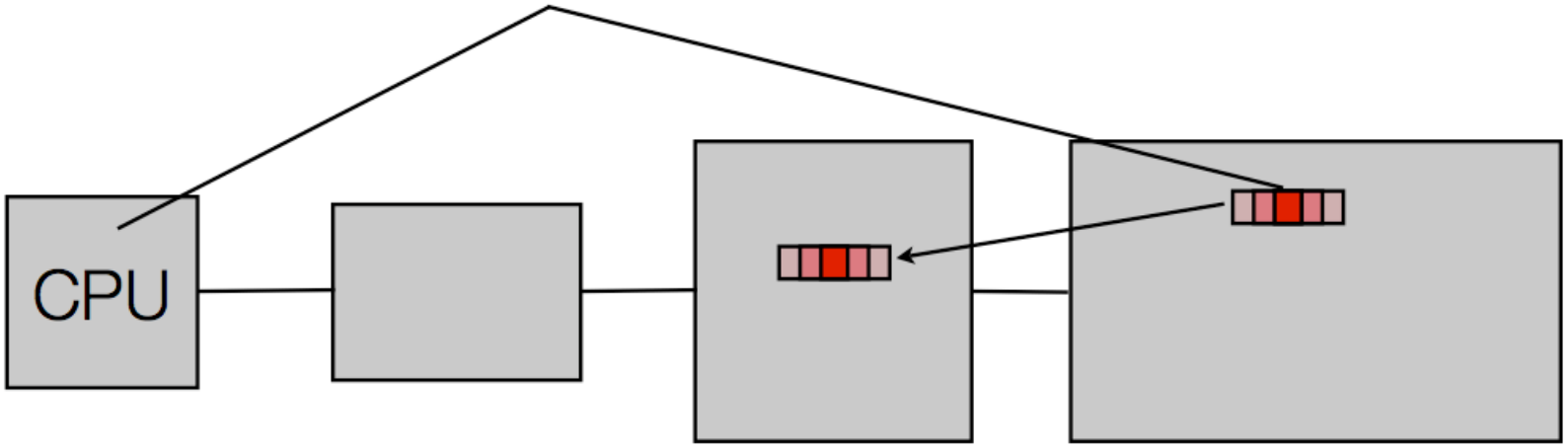
Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



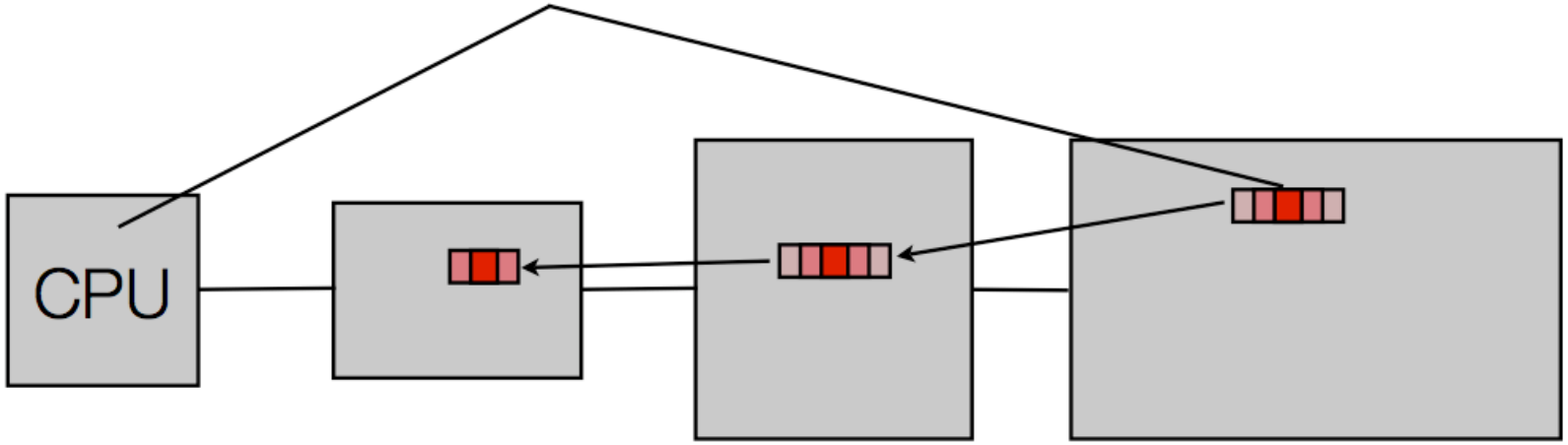
Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



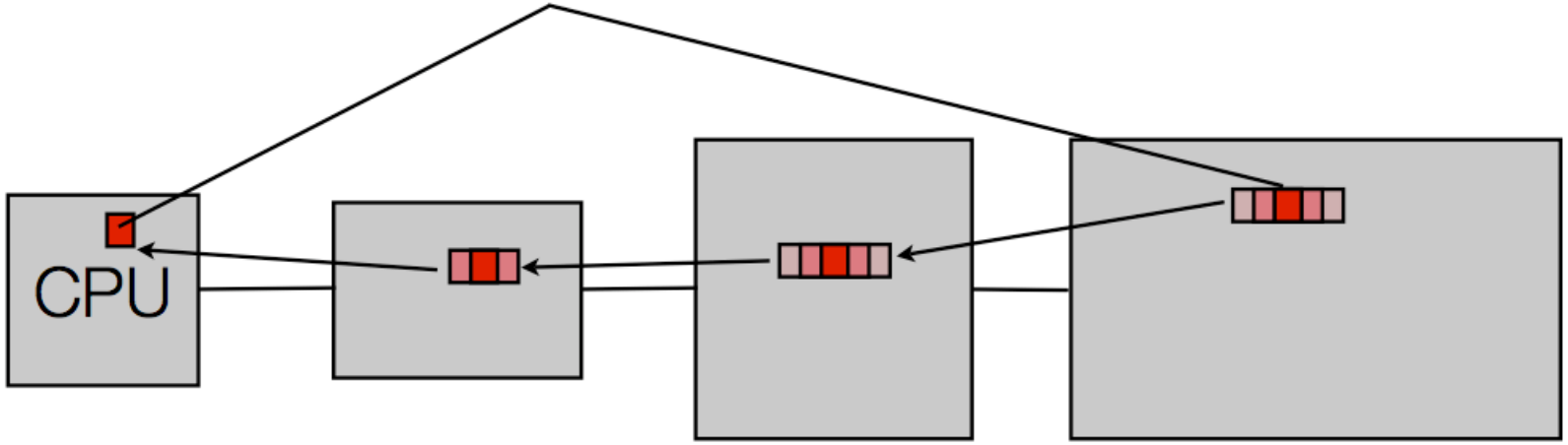
Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



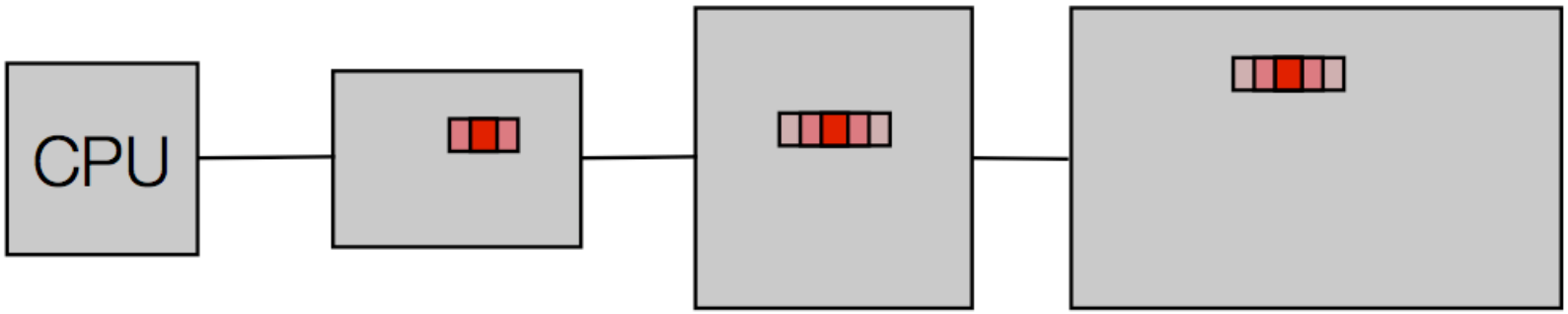
Caches: In Action

- Access a location in memory
- Copy the location and its neighbors into the faster memories closer to the CPU.



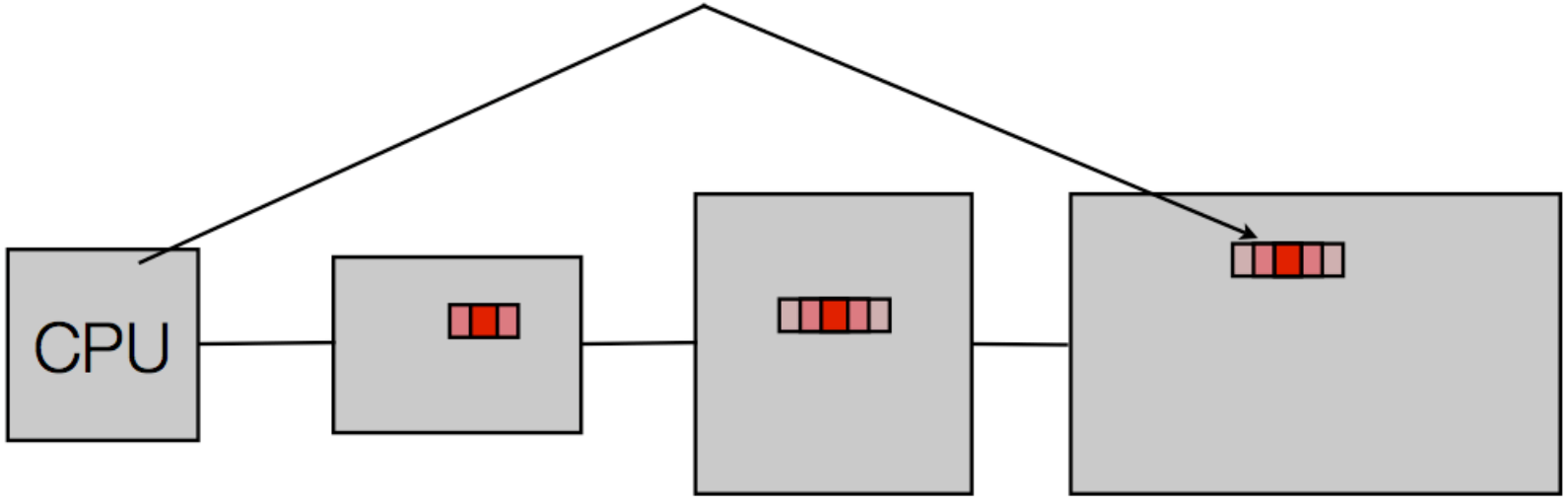
Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



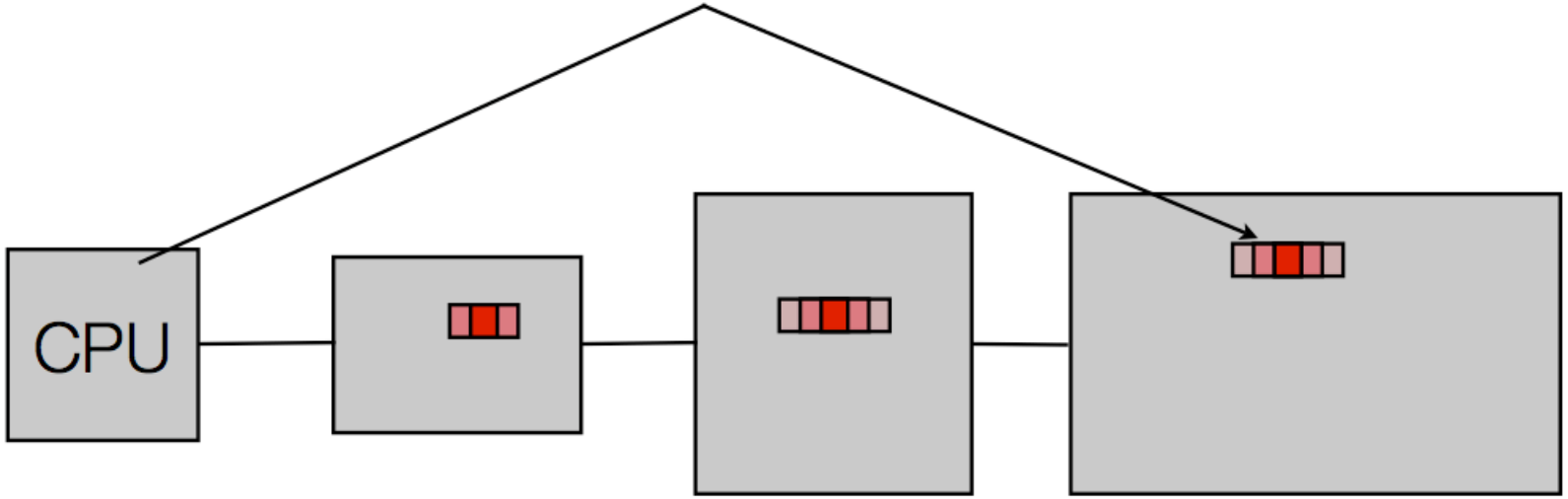
Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



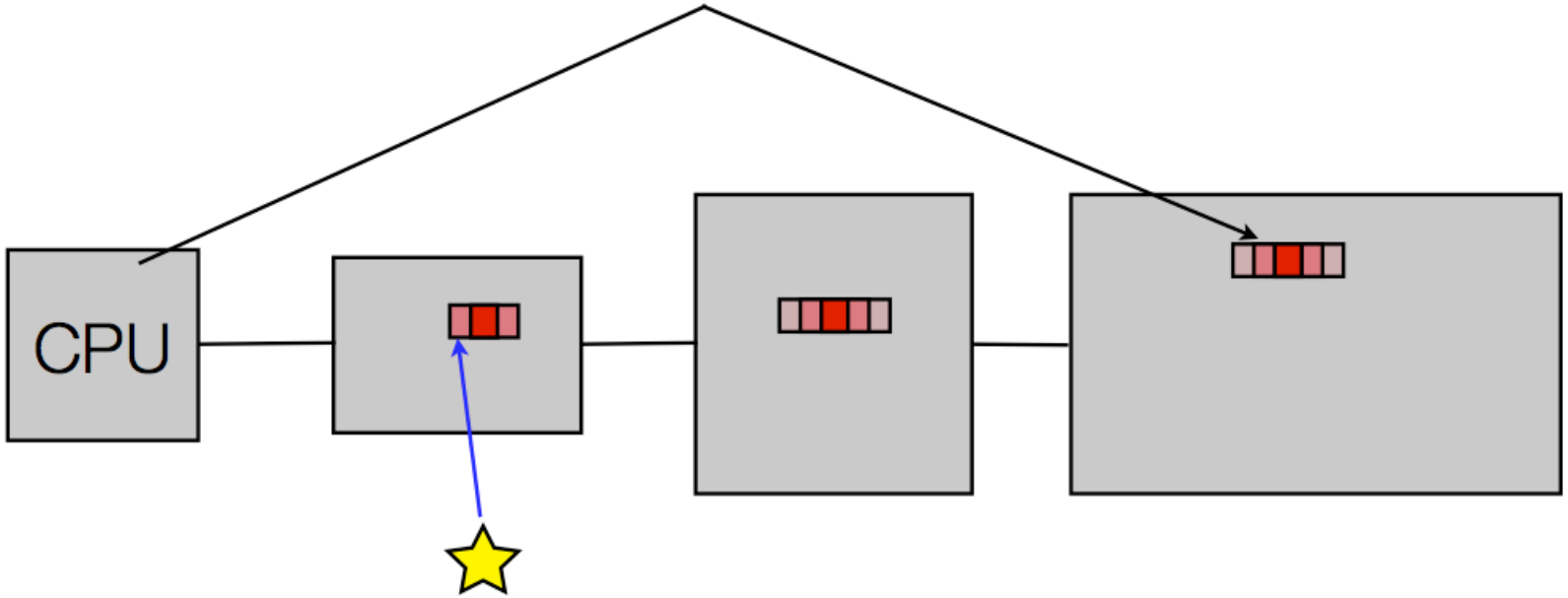
Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



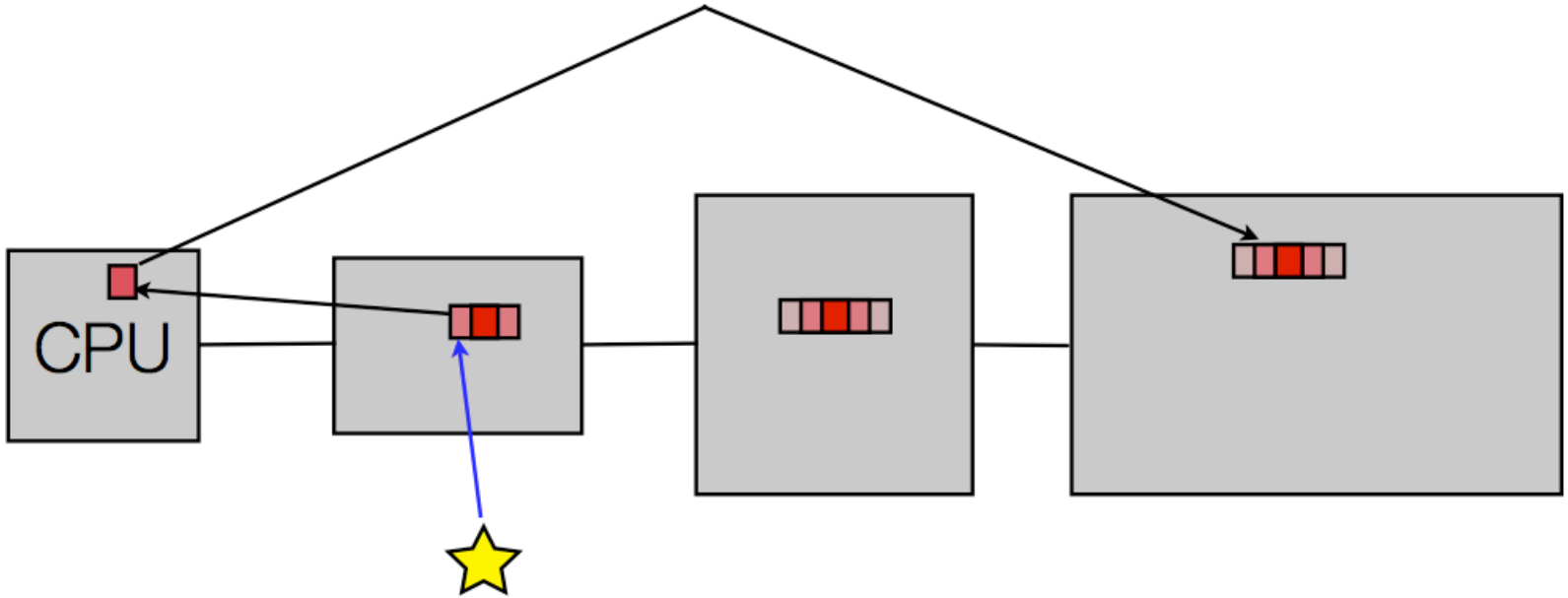
Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.



Caches: In Action

- Next time you access memory, if you already pulled the address into one of the cache levels in a previous access, the value is provided from the cache.

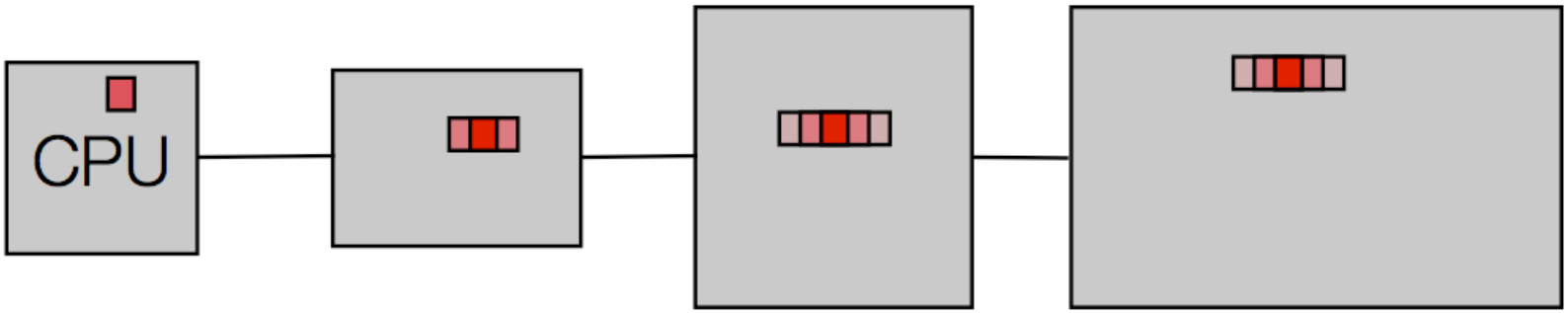


Why Do Caches Work? Locality

- Spatial and Temporal
  - Locations near each other in *space* (address) are highly likely to be accessed near each other in *time*.
- Cost?
  - A high cost for one access but amortize this out with faster accesses afterwards.
- Burden?
  - The *machine* makes sure that memory is kept consistent. If a part of cache must be reused, the cache system writes the data back to main memory before overwriting it with new data.
  - Hardware cache design deals with managing mappings between the different levels and deciding when to write back down the hierarchy.

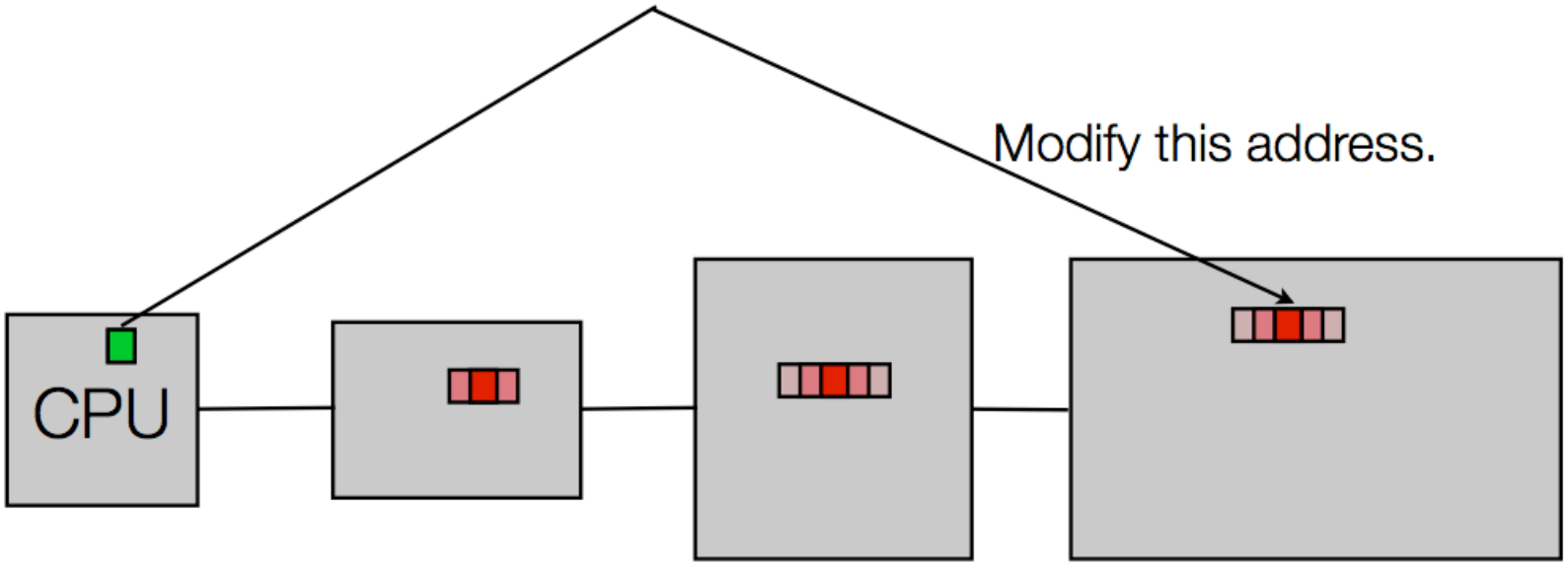
Caches: Memory Consistency?

- What happens when you modify something in memory?



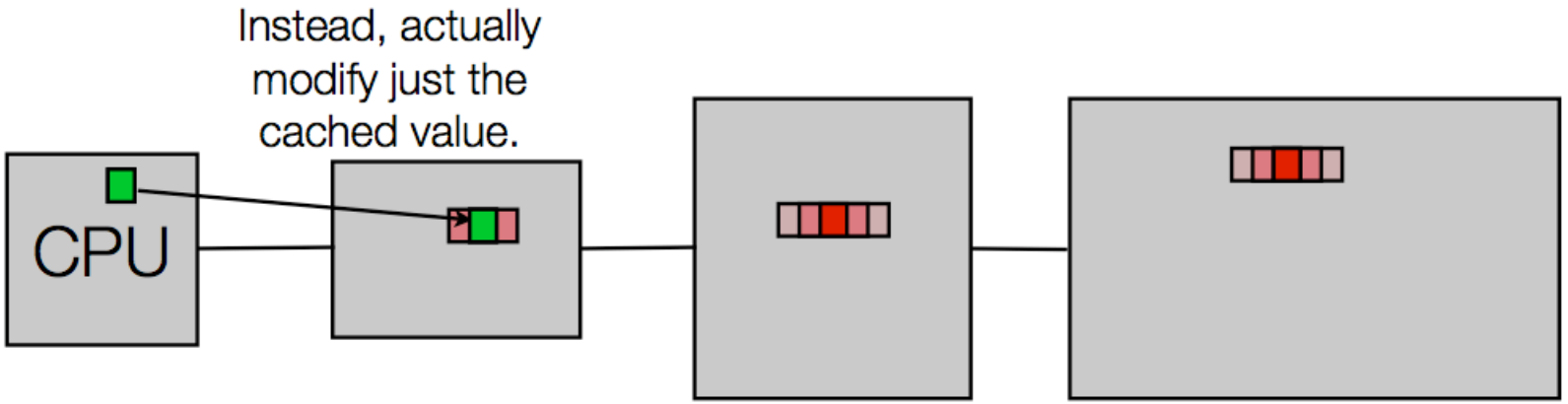
Caches: Memory Consistency?

- What happens when you modify something in memory?



Caches: Memory Consistency?

- What happens when you modify something in memory?
- Writes to memory become cheap. Only go to slow memories when needed. Called **write-back memory.**

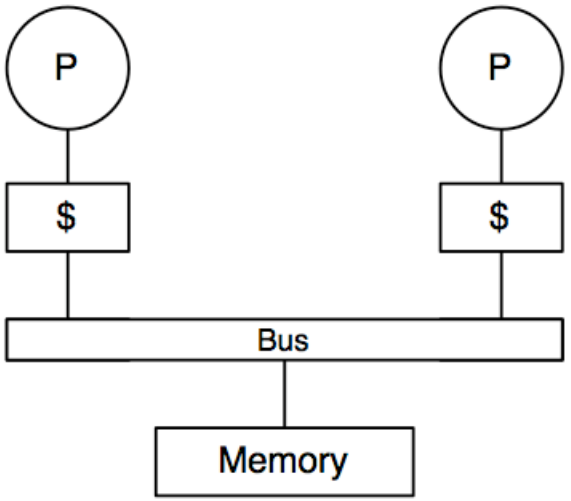


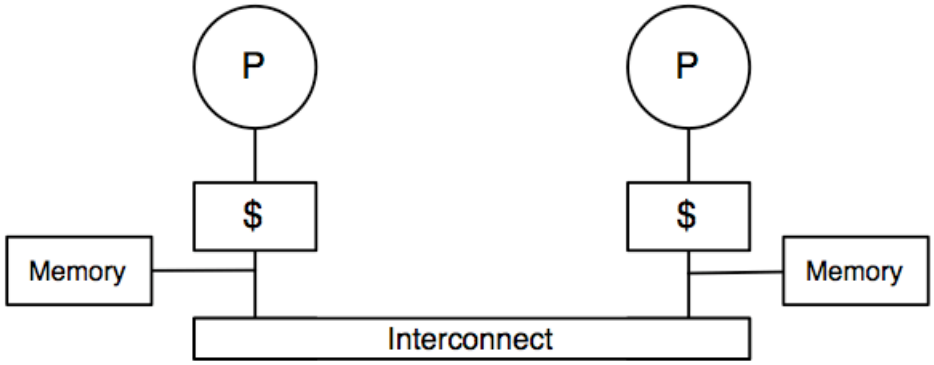
Caches: Memory Consistency?

- Eventually written values must make it back to the main store.
- When?
  - Typically, when a cache block is replaced due to a cache miss, where new data must take the place of old.
- The programmer does NOT see this.
  - Hardware takes care of all this … but things can go wrong very quickly when you modify this model.
    - Forecast for emergent chip multiprocessors: Cell, GPGPU, Terascale, and so on.

Common Memory Models

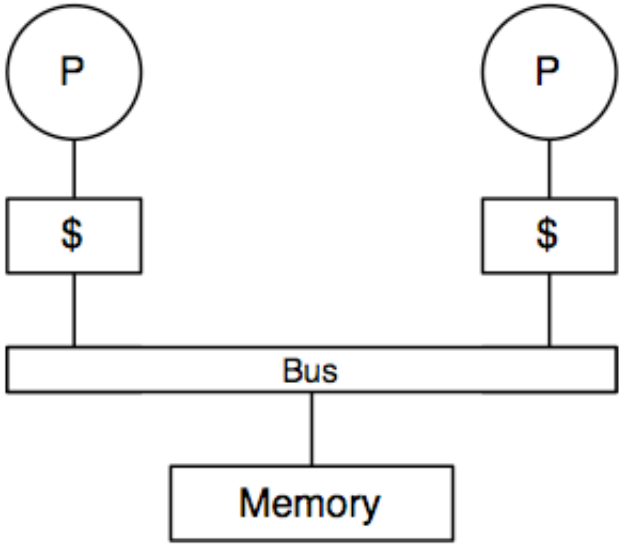
- Shared Memory Architecture
- Distributed Memory Architecture
- Which is Intel? Which is AMD? Others?



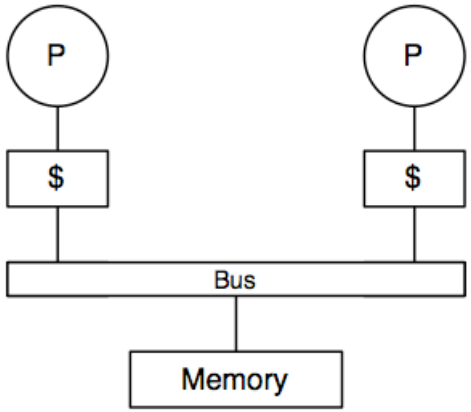


Memory Models: Shared Memory

- Before
  - Only one processor has access to modify memory.
- How do we avoid problems when multiple cache hierarchies see the same memory?

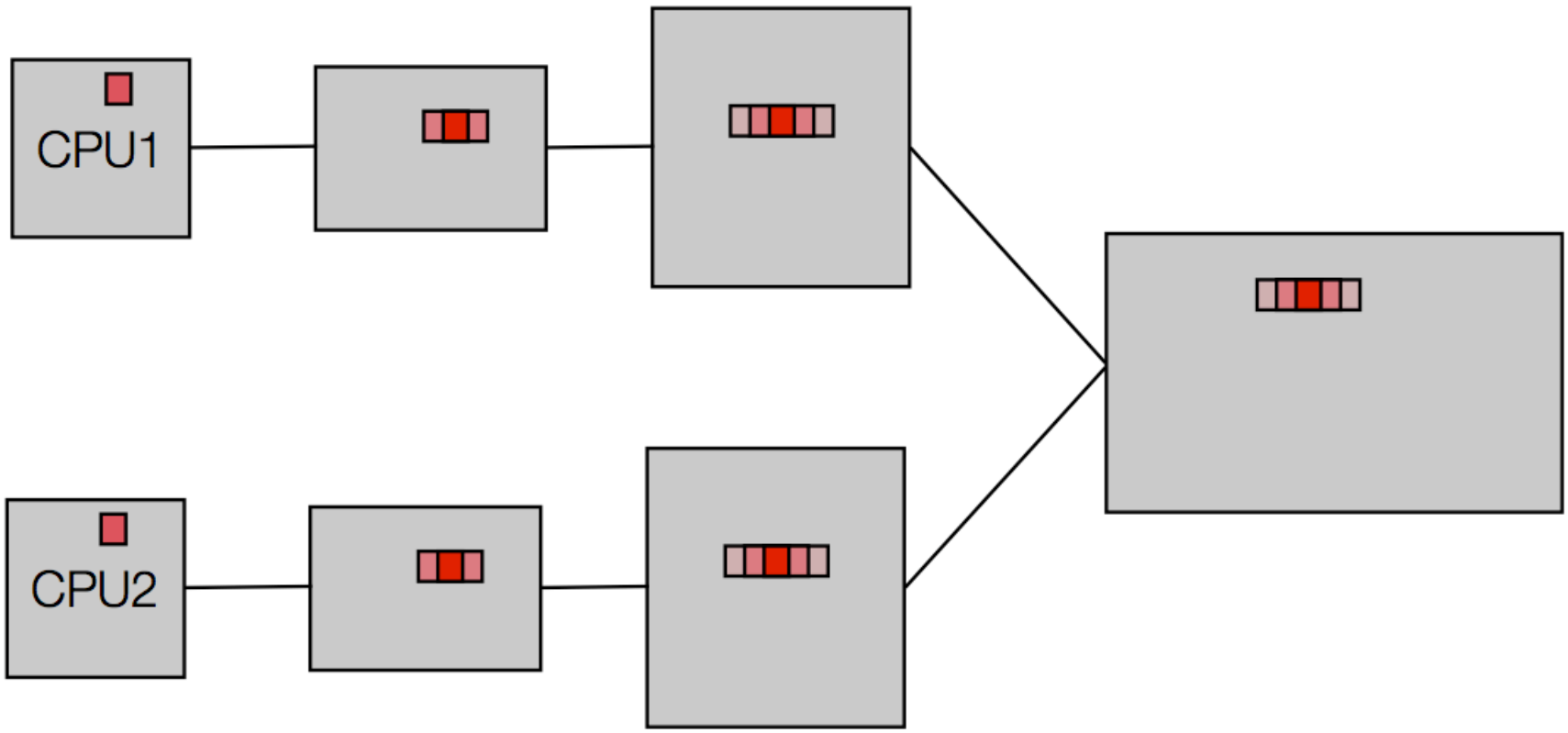


Caching Issues

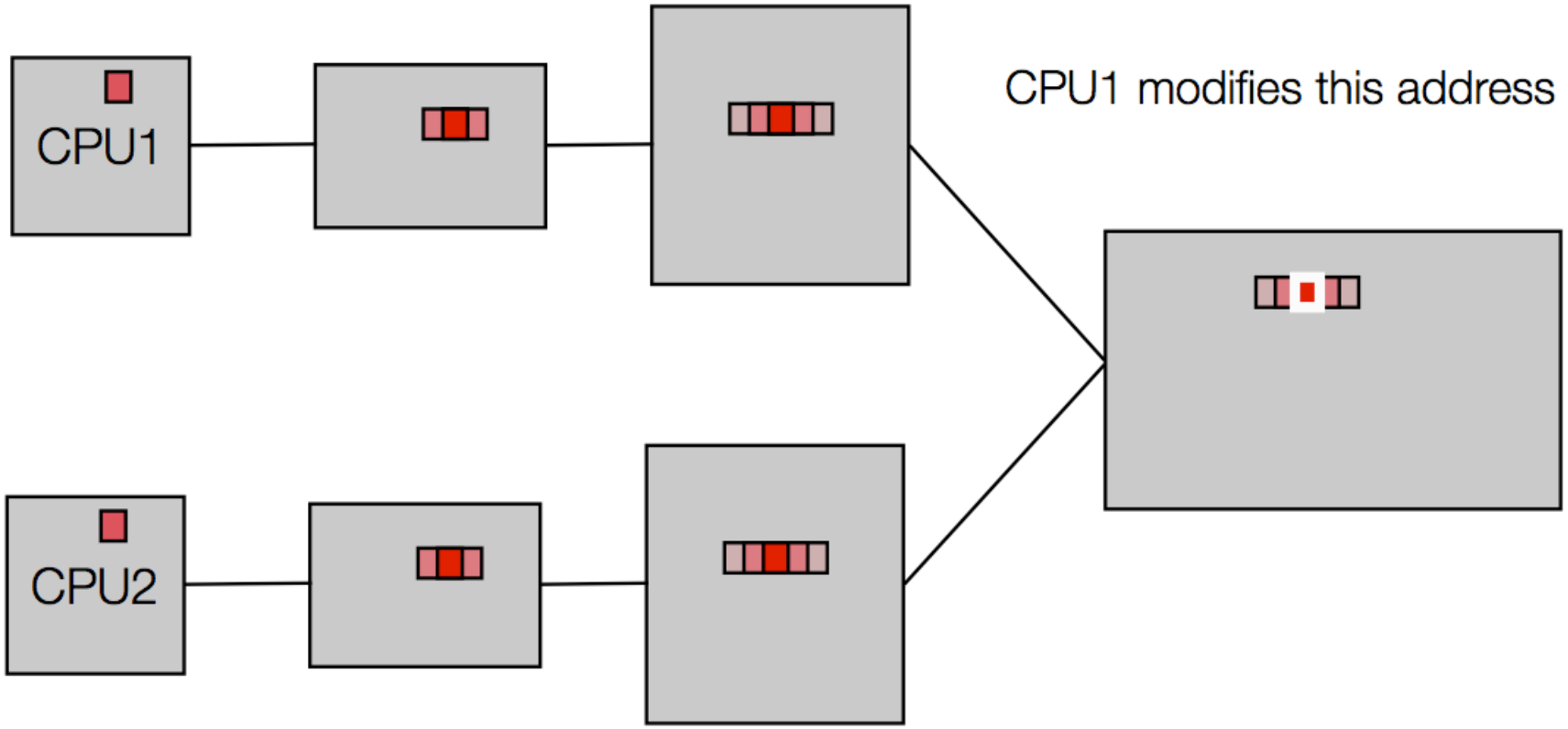


- Assume two processors load locations that are neighbors, so data is replicated in the local processor caches.
- Now, let one processor modify a value.
- The memory view is now inconsistent. One processor sees one version of memory, the other sees a different version.
- How do we resolve this *in hardware* such that the advantages of caches are still seen by application developers in terms of performance while ensuring a consistent (or, coherent) view of memory?

Caching Issues

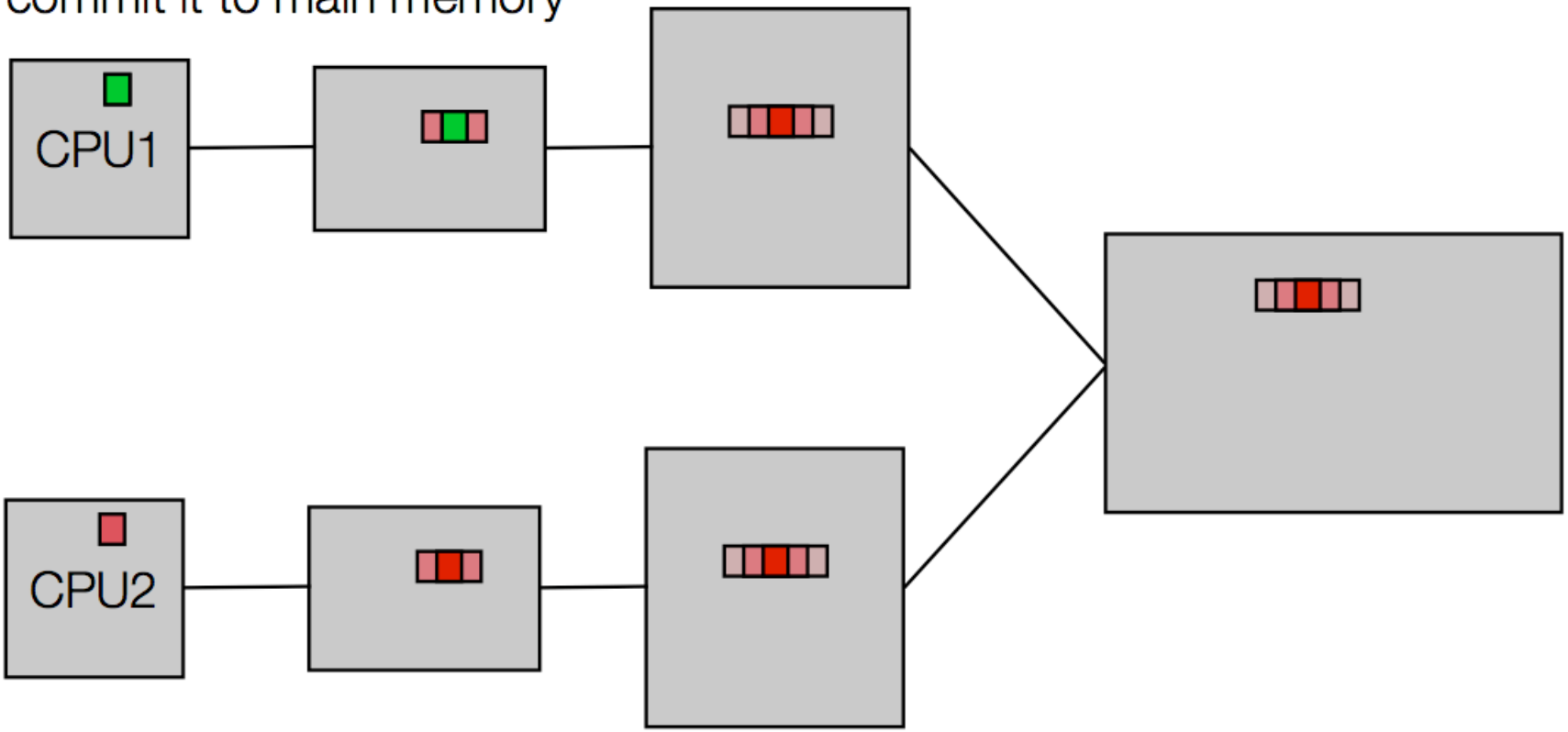


Caching Issues

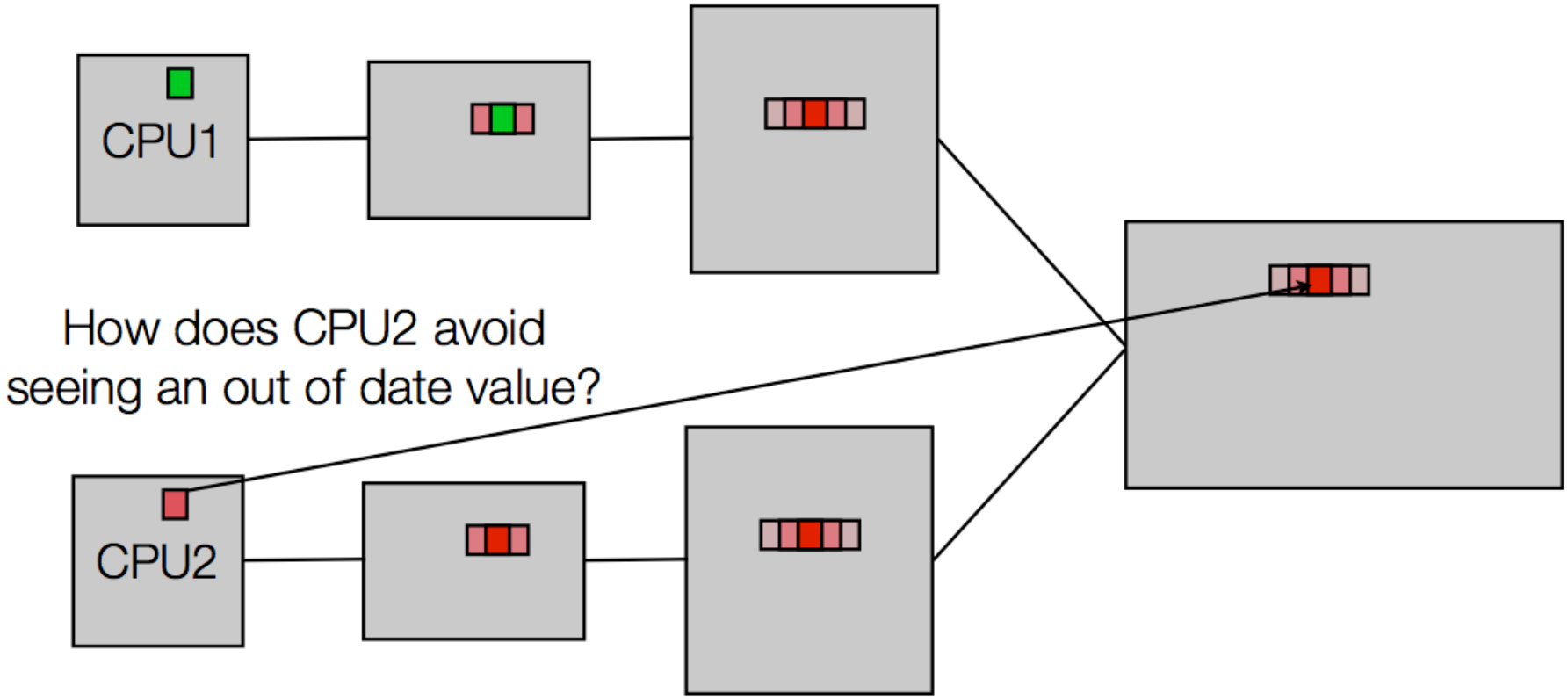


Caching Issues

But so far has no reason to commit it to main memory



Caching Issues



Caching: Memory Consistency?

- Easy to see "memory consistency" problem if we restrict each cache hierarchy to being isolated from the others, ***only sharing main memory***.
- Key insight
  - Make this inconsistency "go away" by making the caches aware of each other.

What is Memory Coherence?

- Definition (Courtesy: "Parallel Computer Architecture" by Culler and Singh)
  1. Operations issued by any particular process occur in the order in which they were issued to the memory system by that process.
  2. The value returned by each read operation is the value written by the last write to that location in the serial order.
- Assumption: *The above requires a hypothetical ordering for all read/write operations by all processes into a total order that is consistent with the results of the overall execution.*
- Sequential Consistency (SC)
  - The memory coherence hardware assists in enforcing SC.

Implicit Properties of Coherence

- The key to solving the cache coherence problem is the hardware implementation of a *cache coherence protocol.*
- A cache coherence protocol takes advantage of two hardware features
  1. State annotations for each cache block (often just a couple bits per block).
  2. Exclusive access to the bus by any accessing process.

Bus Properties

- All processors on the bus see the same activity.
- So, every cache controller sees bus activity in the same *order.*
- Serialization at the bus level results from the phases that compose a bus transaction:
  - *Bus arbitration:* The bus arbiter grants exclusive access to issue commands onto the bus.
  - *Command/address:* The operation to perform (“Read”, “Write”), and the address.
  - *Data:* The data is then transferred.

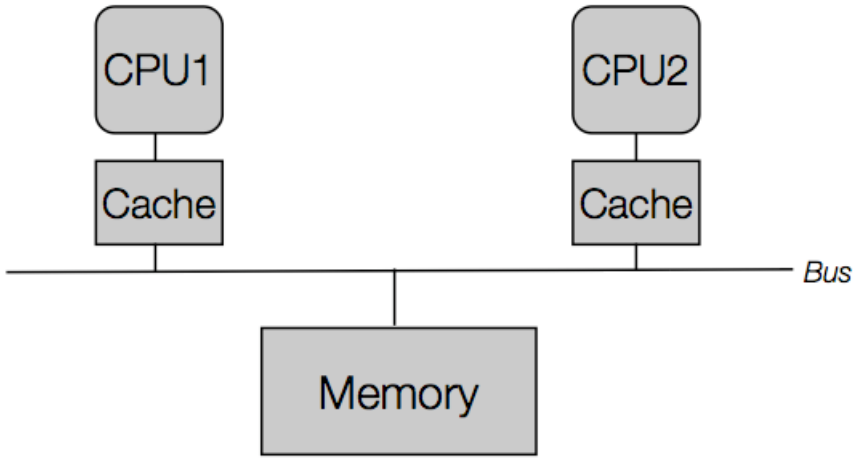
Granularity

- Cache coherence applies at the *block level.*
- Recall that when you access a location in memory, that location *and* its neighbors are pulled into the cache(s). These are *blocks.*

Note: To simplify the discussion, we will only consider a single level of cache. The same ideas translate to deeper cache hierarchies.

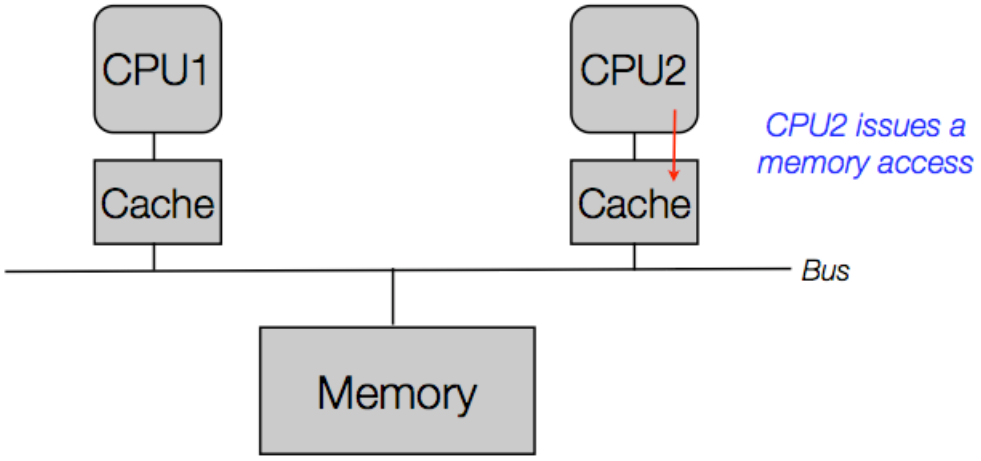
Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



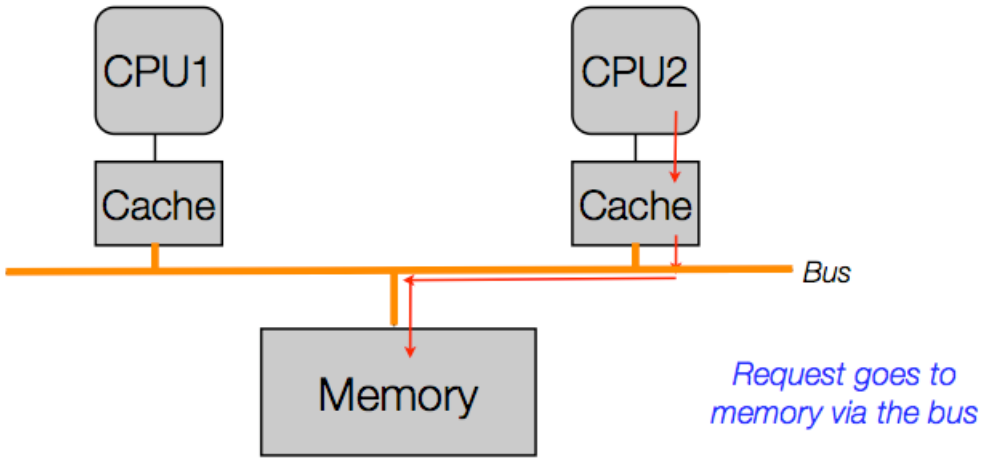
Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



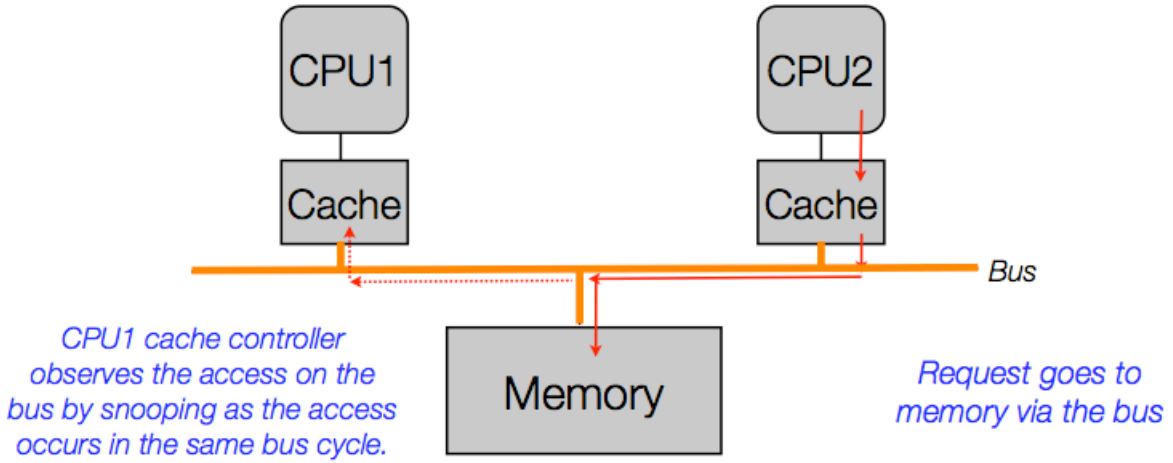
Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



Cache Coherency via the Bus

- Key Idea: Bus Snooping
  - All CPUs on the bus can see activity on the bus regardless of if they initiated it.



Invalidation vs. Update

- A cache controller snoops and sees a write to a location that it has a now-outdated copy of.
  - What does it do?
- **Invalidation**
  - Mark cache block as *invalid*, so when CPU accesses it again, a miss will result and the updated data from main memory will be loaded. Requires one bit per block to implement.
- **Update**
  - See the write and update the caches with the value observed being written to main memory.

Write-Back vs. Write-Through Caches

- Write-Back
  - On a write miss, the CPU reads the entire block from memory where the write address is, updates the value in cache, and marks the block as modified (aka dirty).
- Write-Through
  - When the processor writes, even to a block in cache, a bus write is generated.
- Write-back is more efficient with respect to bandwidth usage on the bus, and hence, ubiquitously adopted.

Cache Coherence and Performance

- Unlike details with pipelining that only concern compiler writers, you the programmer really need to acknowledge that this is going on under the covers.

- *The coherence protocol can impact your performance.*

Cache Coherence: Performance Demo