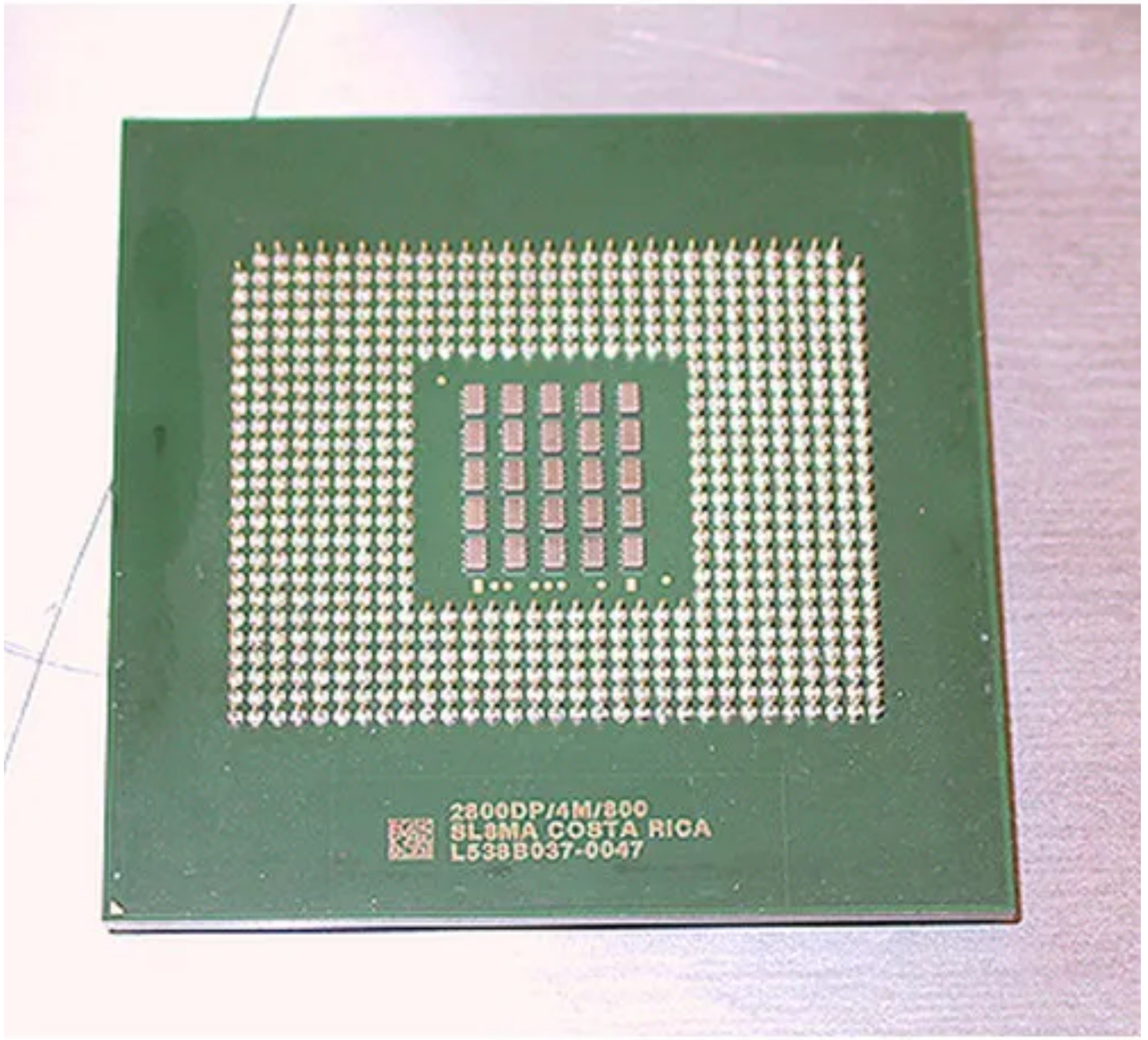
Multicore Processors

Check In 6 today!
HW3 due Friday



Intel Xeon Dual-Core Processor (2005) ➡️ first commodity multicore

From 2000—2005, techniques to exploit ILP for performance slowed down

Academics had been studying “theoretical” multiprocessor coordination since the 1950s

Growing interest in high end servers due to the “data-intensive applications on the Internet”

Image credit: https://store.flagshiptech.com/2-8ghz-4mb-800mhz-intel-xeon-dual-core-cpu-paxville-sl8ma-td428/

Outline

- What do multicore systems look like?
- Challenges of multicore systems
- Check-In 6

Revisiting Multithreaded Programs

System calls!

The operating system acts as the interface for software to interact with hardware directly!

On pthread_create, construct a process structure with the instruction address of thread_fn as the program counter and the args as the stack state

```
int main(int argc, char **argv) {
    pthread_t tid[num_threads];

    for (int i = 0; i < num_threads; i++) {
        pthread_create(&tid[i], NULL, thread_fn, (void *) NULL);
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(&tid[i], NULL);
    }

    return 0;
}
```

```
void *thread_fn(void *args) {
    // do parallel things!
    lock.acquire();
    // do sequential things!
    lock.release();
}
```

Revisiting Multithreaded Programs (under the hood)

- Because the `pthread` functions are implemented as system calls, *thread-level parallelism* is highly user-facing!
- The operating system creates process structs as variables, and they live in the operating system's reserved address space
- After the processes are created, the operating system can trigger an *interrupt* (e.g., a signal implemented in hardware as a special communication type) to other processors in the system to redirect their program counter to the new process contexts

Chat with your neighbor(s)!

Suppose a program is written with 100 threads and run on a 100-core multiprocessor. What is the expected speedup of the program relative to running the same program sequentially?

The Speedup Pitfall with Amdahl's Law

Takeaway: programs require very high percentages of parallelizable regions to fully benefit from multicore!

- To understand speedup due to thread-level parallelism, we need to understand how much of the program is parallelizable versus being sequential
- Amdahl's Law: speedup = ((1 - % parallel) + (% parallel / speedup parallel))$^{-1}$
- Example: suppose a program that is 85% parallelizable is written for 100 cores with 75 threads

Speedup = ((1 - .85) + (.85 / 75))$^{-1}$
Speedup = 1 / (.15 + .0113)
Speedup = 6.2 times speedup

100 threads?

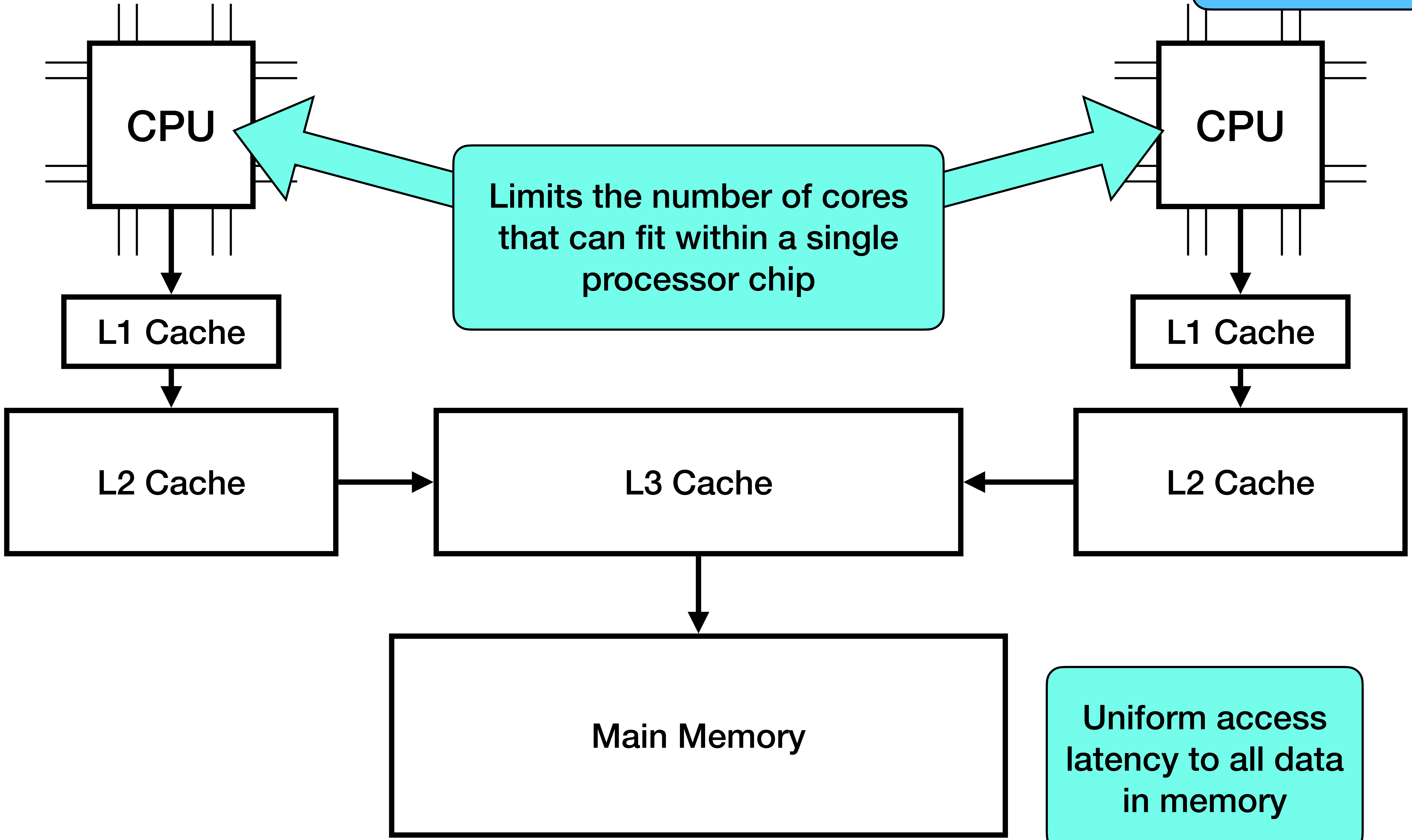
Speedup = ((1 - .85) + (.85 / 100))$^{-1}$
Speedup = 1 / (.15 + .0085)
Speedup = 6.3 times speedup

How parallel to get 80x?

$80 = ((1 - p) + (p / 100))^{-1}$
$80((1 - p) + (p / 100)) = 1$
$80(1 - p) + 80(p / 100) = 1$
$80 - 80p + .8p = 1$
$-79.2p = -79$
$p = .9974$

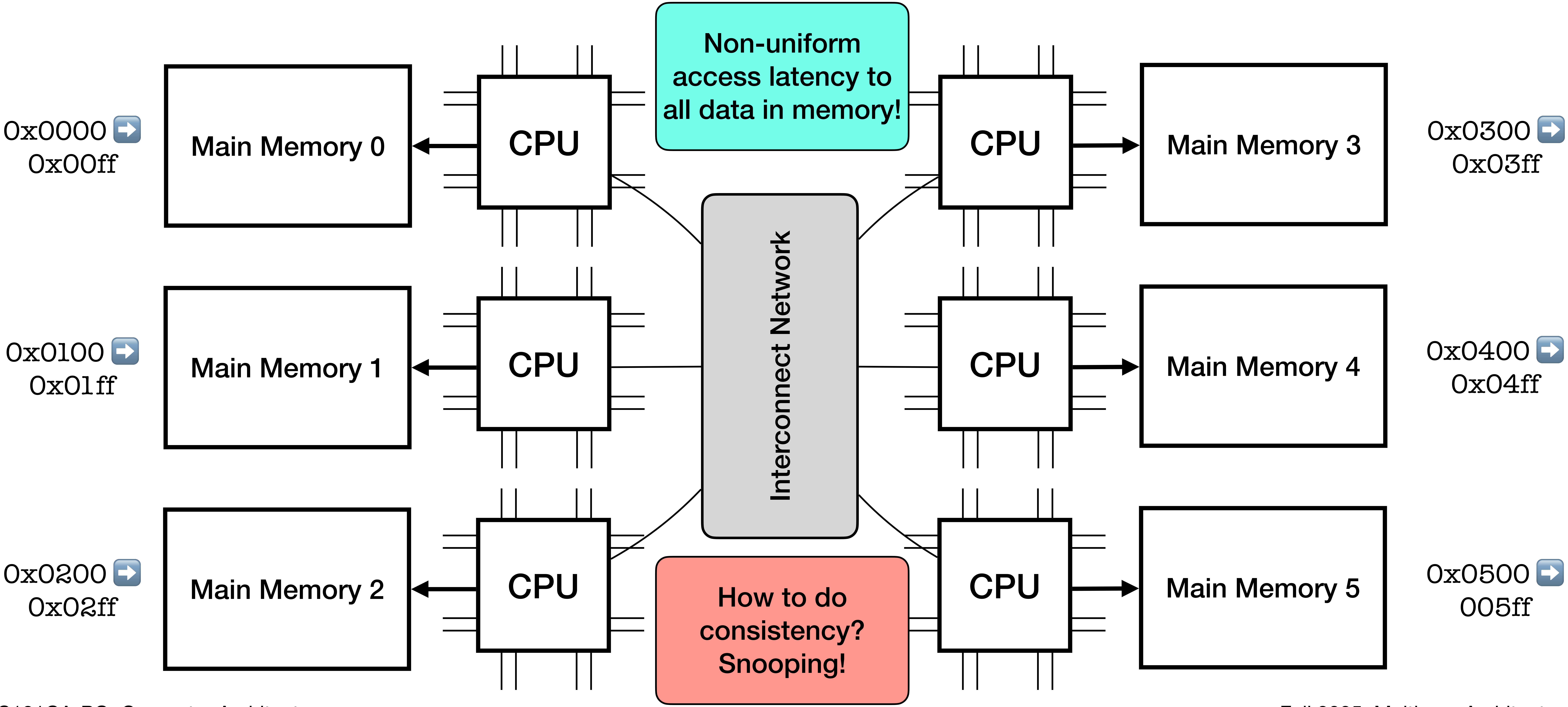
Simple Multicore Architectures

Describes a *centralized shared-memory* multiprocessor or a *symmetric* multiprocessor



Uniform access latency to all data in memory

Distributed Memory Multiprocessor Architectures



Takeaways

- Multiprocessors allow programmers to utilize thread-level parallelism without worrying about constraints within a processor
- Thread-level parallelism is limited by Amdahl's law
- Multiprocessors may be organized as centralized shared-memory multicore or distributed-memory multicore architectures