Multicore and Parallelism

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 4.10, 1.7, 1.8, 5.10, 6

How to improve performance?

We have looked at

- Pipelining

- To speed up:
  - Deeper pipelining
  - Make the clock run faster
  - Parallelism
    - Not a luxury, a necessity

Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?

A: Deeper pipeline

- E.g. 250MHz 1-stage; 500Mhz 2-stage; 1GHz 4-stage; 4GHz 16-stage

Pipeline depth limited by…

- max clock speed
- min unit of work (less work per stage ⇒ shorter clock cycle)
- dependencies, hazards / forwarding logic

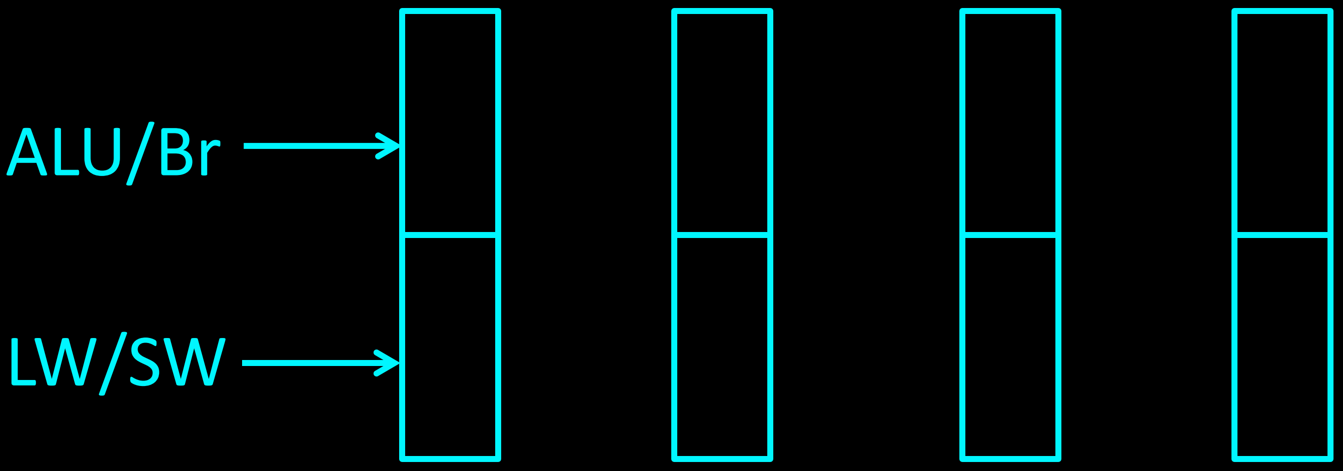
Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?

A: Multiple issue pipeline

- Start multiple instructions per clock cycle in duplicate stages



Multiple issue pipeline

Static multiple issue
aka Very Long Instruction Word
Decisions made by compiler

Dynamic multiple issue
Decisions made on the fly

Cost: More execute hardware
Reading/writing register files: more ports

Static Multiple Issue

a.k.a. Very Long Instruction Word (VLIW)

Compiler groups instructions to be issued together

- Packages them into “issue slots”

Q: How does HW detect and resolve hazards?

A: It doesn’t

→ Simple HW, assumes compiler avoids hazards

Example: Static Dual-Issue 32-bit MIPS

- Instructions come in pairs (64-bit aligned)
  - One ALU/branch instruction (or nop)
  - One load/store instruction (or nop)

MIPS with Static Dual Issue

Two-issue packets

- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop
- Delay slot: 2 instructions (1 cycle)

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Scheduling Example

Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:			1
			2
			3
			4

Speculation

- Reorder instructions
  - To fill the issue slot with useful work
  - Complicated: exceptions may occur

Optimizations to make it work

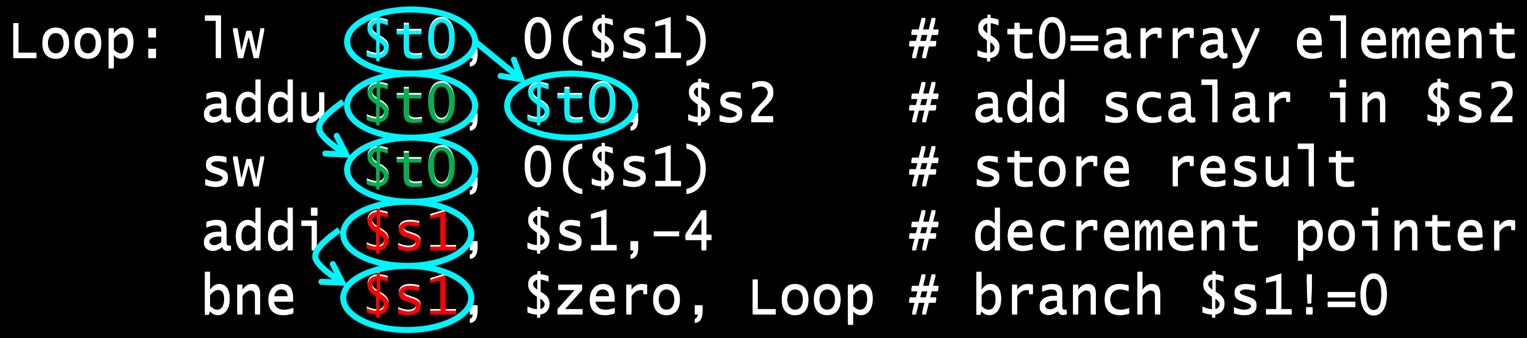
Move instructions to fill in nops

- Need to track hazards and dependencies

Loop unrolling

Scheduling Example

Schedule this for dual-issue MIPS



```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

Scheduling Example

Compiler scheduling for dual-issue MIPS…

```
Loop:  lw   $t0, 0($s1)          # $t0 = A[i]
       lw   $t1, 4($s1)          # $t1 = A[i+1]
       addu $t0, $t0, $s2        # add $s2
       addu $t1, $t1, $s2        # add $s2
       sw   $t0, 0($s1)          # store A[i]
       sw   $t1, 4($s1)          # store A[i+1]
       addi $s1, $s1, +8         # increment pointer
       bne  $s1, $s3, Loop       # continue if $s1!=end
```

8 cycles

ALU/branch slot	Load/store slot	cycle

6 cycles

Scheduling Example

Compiler scheduling for dual-issue MIPS…

```
Loop:  lw   $t0, 0($s1)           # $t0 = A[i]
       lw   $t1, 4($s1)           # $t1 = A[i+1]
       addu $t0, $t0, $s2         # add $s2
       addu $t1, $t1, $s2         # add $s2
       sw   $t0, 0($s1)           # store A[i]
       sw   $t1, 4($s1)           # store A[i+1]
       addi $s1, $s1, +8          # increment pointer
       bne  $s1, $s3, Loop        # continue if $s1!=end
```

8 cycles

ALU/branch slot	Load/store slot	cycle

5 cycles

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS…

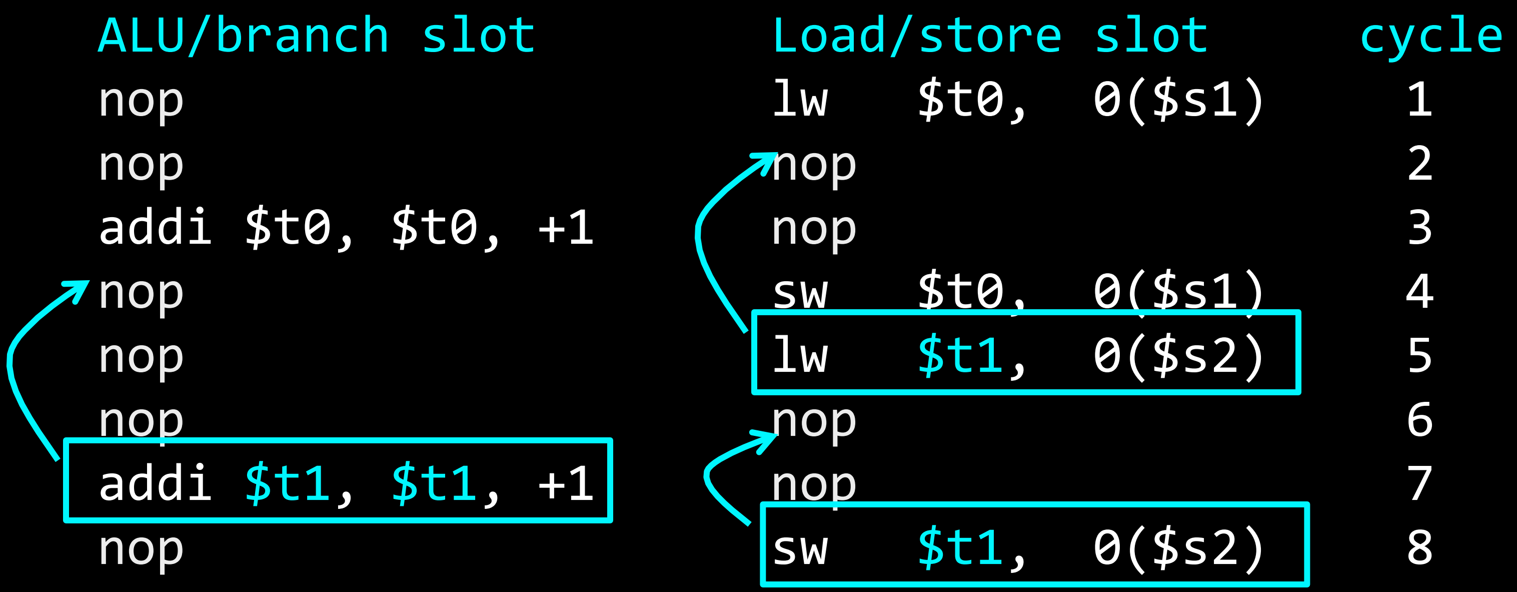
```
lw   $t0, 0($s1)          # load A
addi $t0, $t0, +1         # increment A
sw   $t0, 0($s1)          # store A
lw   $t0, 0($s2)          # load B
addi $t0, $t0, +1         # increment B
sw   $t0, 0($s2)          # store B
```

ALU/branch slot	Load/store slot	cycle
nop	lw $t0, 0($s1)	1
nop	nop	2
addi $t0, $t0, +1	nop	3
nop	sw $t0, 0($s1)	4
nop	lw $t0, 0($s2)	5
nop	nop	6
addi $t0, $t0, +1	nop	7
nop	sw $t0, 0($s2)	8

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS…

```
lw   $t0, 0($s1)          # load A
addi $t0, $t0, +1         # increment A
sw   $t0, 0($s1)          # store A
lw   $t1, 0($s2)          # load B
addi $t1, $t1, +1         # increment B
sw   $t1, 0($s2)          # store B
```



ALU/branch slot	Load/store slot	cycle
nop	lw $t0, 0($s1)	1
nop	nop	2
addi $t0, $t0, +1	nop	3
nop	sw $t0, 0($s1)	4
nop	lw $t1, 0($s2)	5
nop	nop	6
addi $t1, $t1, +1	nop	7
nop	sw $t1, 0($s2)	8

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS…

```
lw   $t0, 0($s1)          # load A
addi $t0, $t0, +1         # increment A
sw   $t0, 0($s1)          # store A
lw   $t1, 0($s2)          # load B
addi $t1, $t1, +1         # increment B
sw   $t1, 0($s2)          # store B
```

ALU/branch slot	Load/store slot	cycle
nop	lw $t0, 0($s1)	1
nop	lw $t1, 0($s2)	2
addi $t0, $t0, +1	nop	3
addi $t1, $t1, +1	sw $t0, 0($s1)	4
nop	sw $t1, 0($s2)	5

Problem: What if $s1 and $s2 are equal (***aliasing***)? Won’t work

Dynamic Multiple Issue

a.k.a. SuperScalar Processor

CPU examines instruction stream and chooses multiple instructions to issue each cycle

- Compiler can help by reordering instructions….
- … but CPU is responsible for resolving hazards

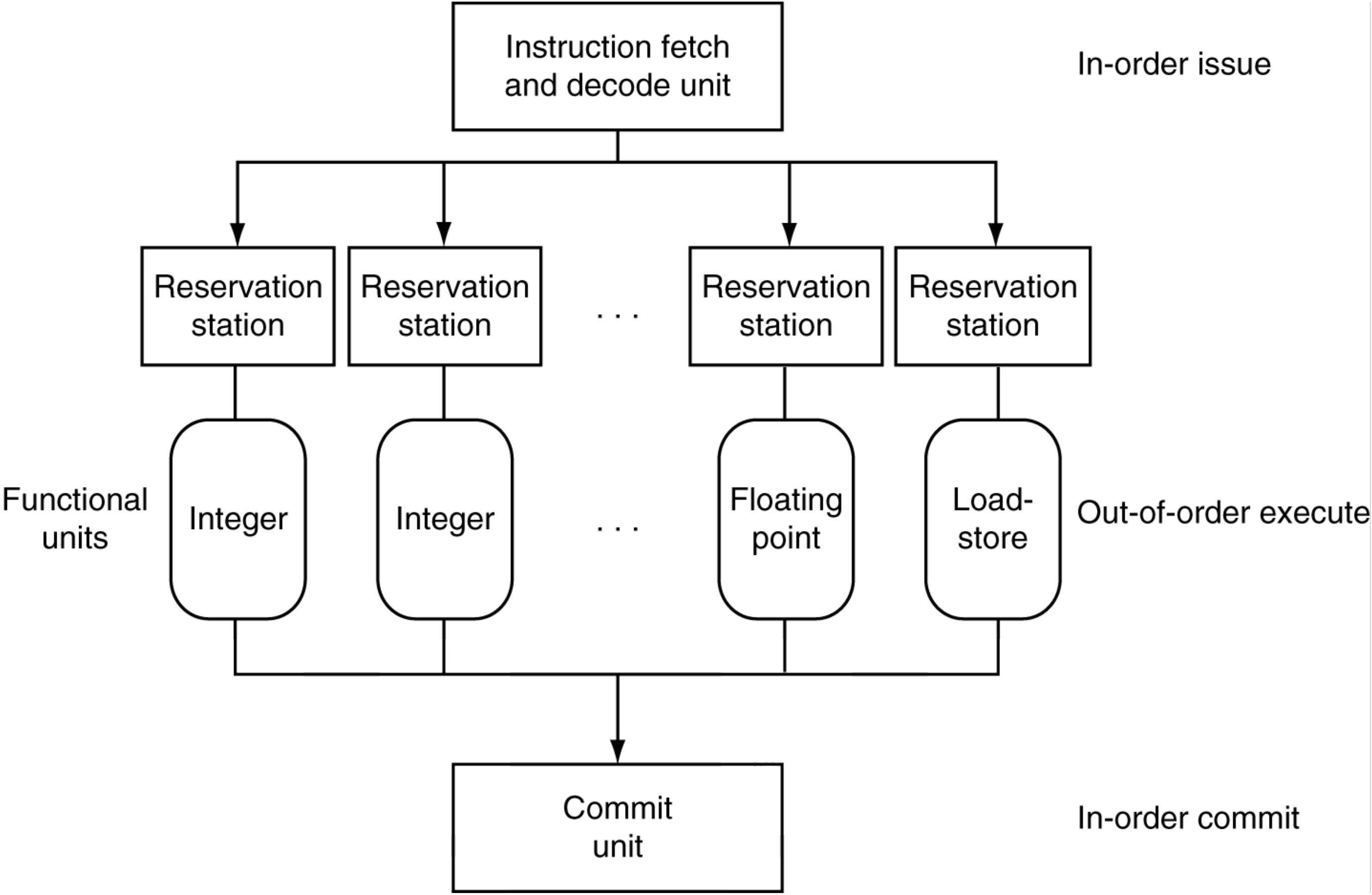
Dynamic Multiple Issue

a.k.a. SuperScalar Processor

Speculation/Out-of-order Execution

- Execute instructions as early as possible
- Aggressive register renaming
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don’t commit results until all previous insts. are retired

Dynamic Multiple Issue



Why dynamic scheduling?

To handle unpredictable stalls

- Like cache misses

Hides details of pipeline from applications

- Abstraction

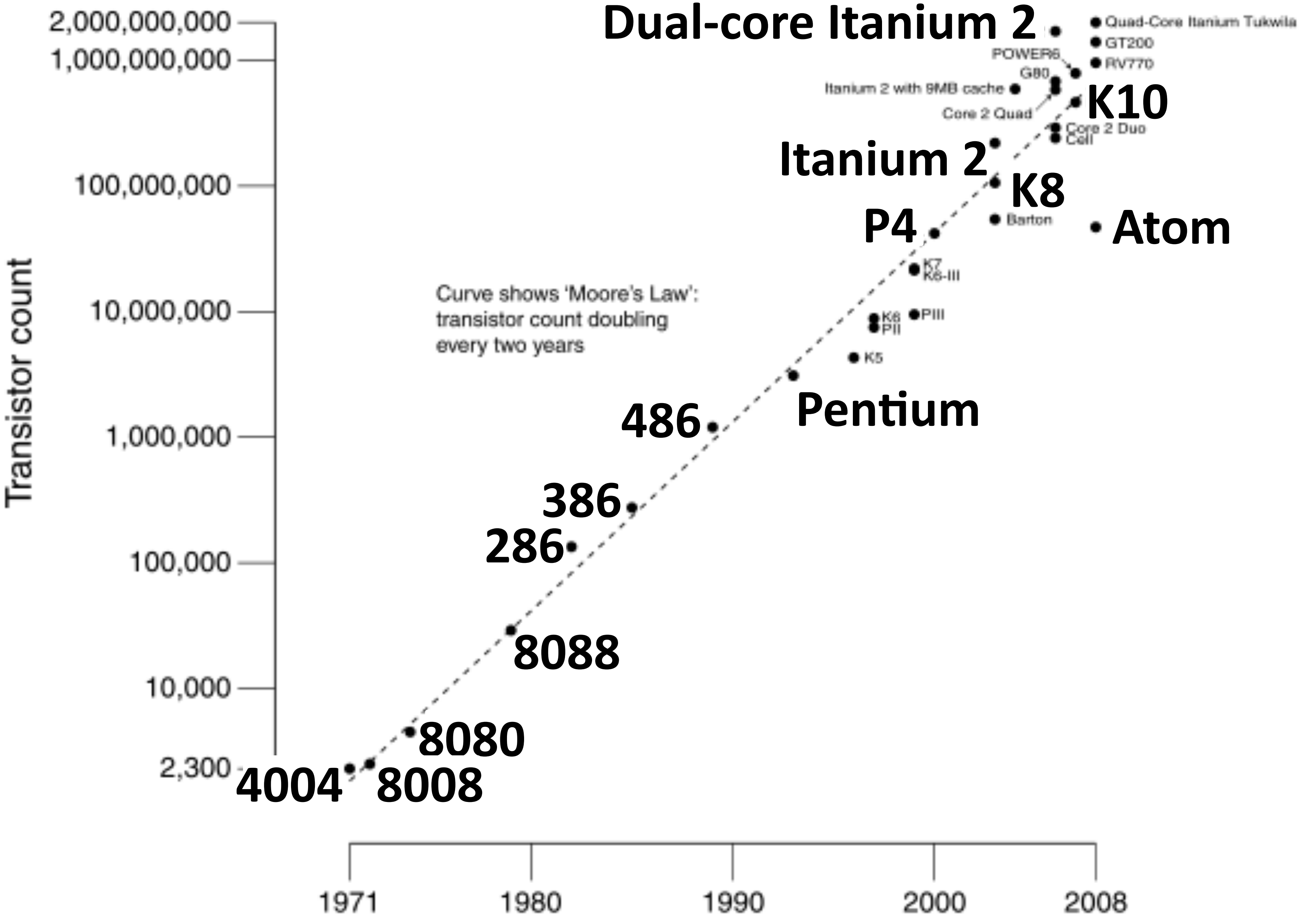
Does Multiple Issue Work?

Q: Does multiple issue / ILP work?

A: Kind of… but not as much as we'd like

Limiting factors?

- Programs dependencies
- Hard to detect dependencies → be conservative
  - e.g. Pointer Aliasing: A[0] += 1; B[0] *= 2;
- Hard to expose parallelism
  - Can only issue a few instructions ahead of PC
- Structural limits
  - Memory delays and limited bandwidth
- Hard to keep pipelines full

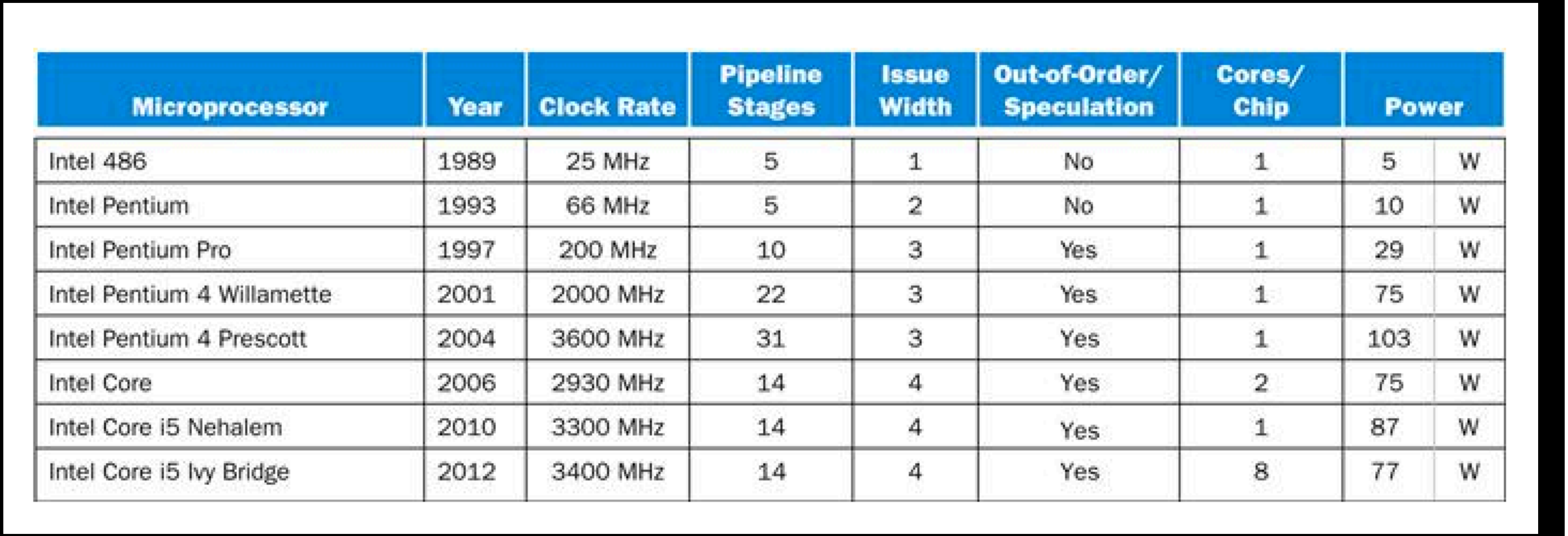


Curve shows 'Moore's Law':
transistor count doubling
every two years

Power Efficiency

Q: Does multiple issue / ILP cost much?

A: Yes. Dynamic issue & speculation requires power



Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

→ Multiple simpler cores may be better?

Why Multicore?

Moore's law

- A law about transistors
- Smaller means more transistors per die
- And smaller means faster too

But: need to worry about power too…

Power Wall

Power = capacitance * voltage^2 * frequency

approx. capacitance * voltage^3

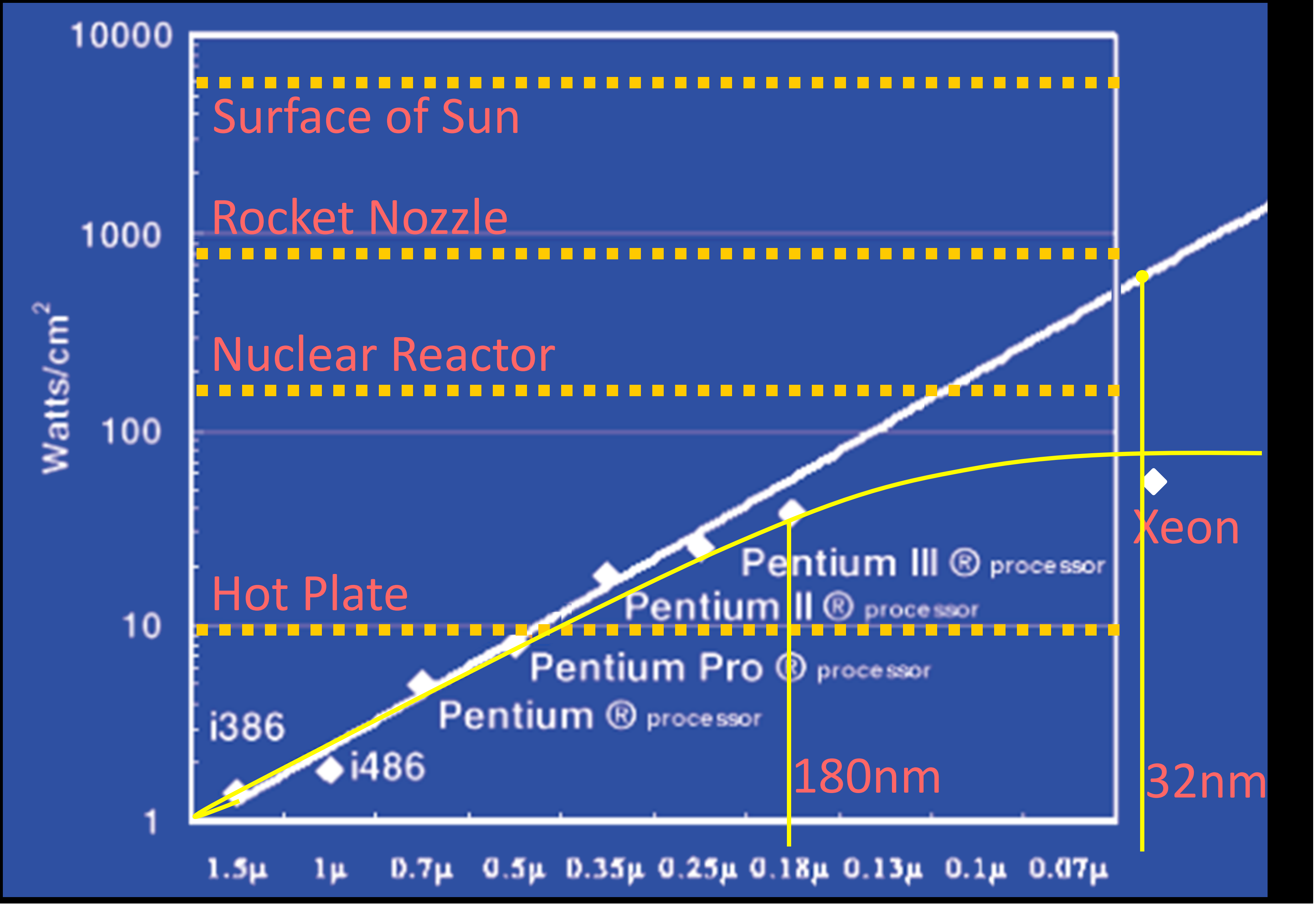
Reducing voltage helps (a lot)

Better cooling helps

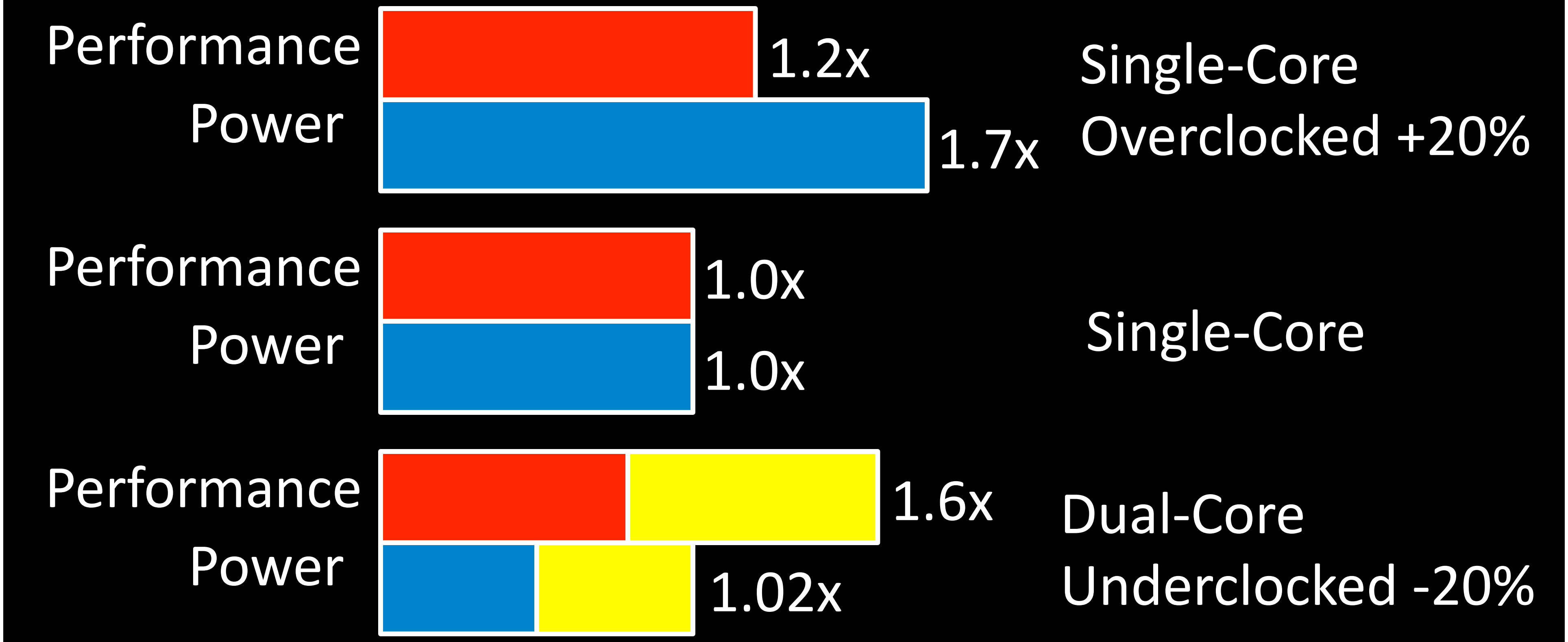
The power wall

- We can’t reduce voltage further - leakage
- We can’t remove more heat

Power Limits

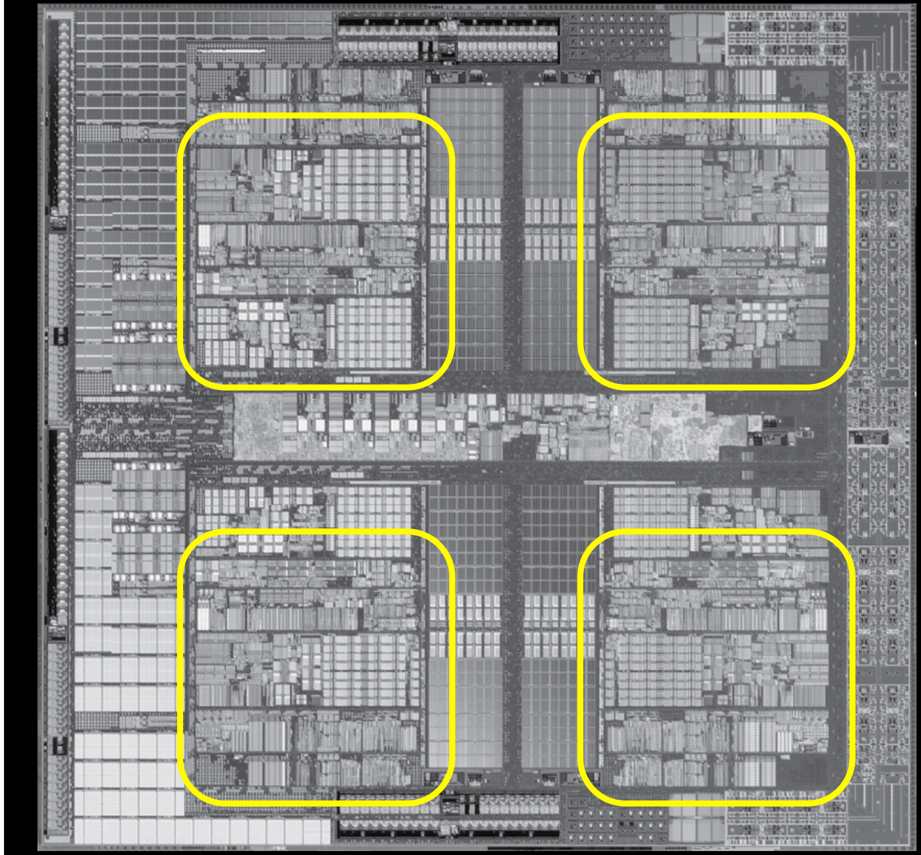


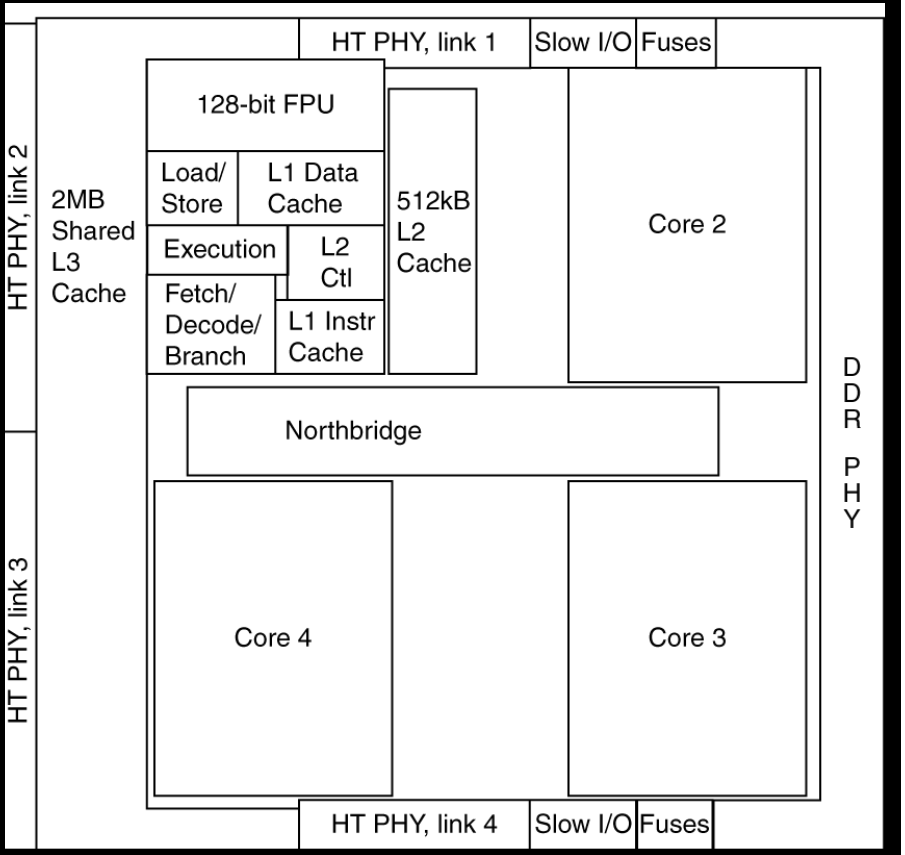
Why Multicore?



Inside the Processor

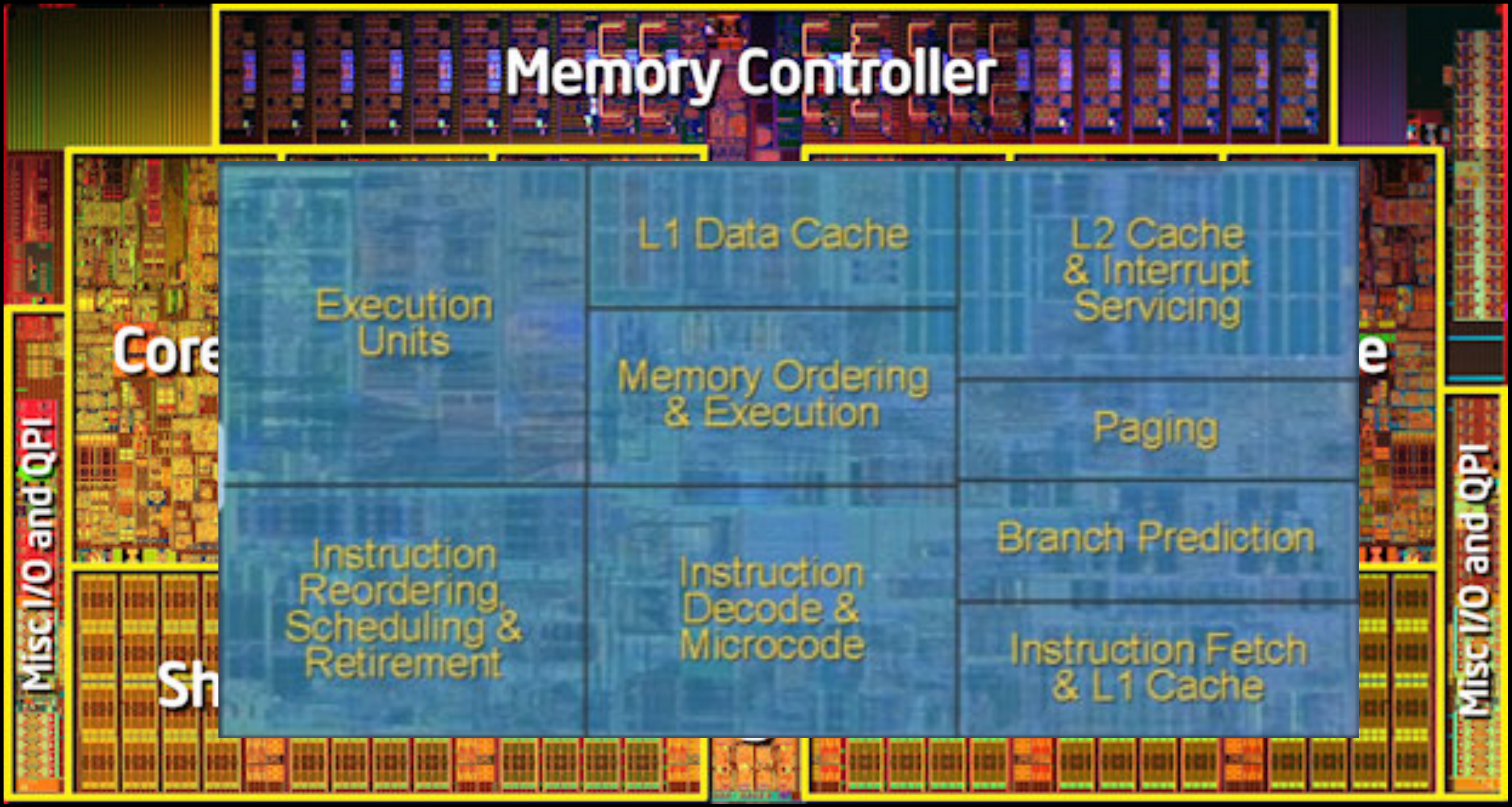
AMD Barcelona Quad-Core: 4 processor cores





Inside the Processor

Intel Nehalem Hex-Core


Memory Controller
Core
e
Sh
Misc I/O and QPI
Misc I/O and QPI
Execution Units
L1 Data Cache
L2 Cache & Interrupt Servicing
Memory Ordering & Execution
Paging
Branch Prediction
Instruction Reordering, Scheduling & Retirement
Instruction Decode & Microcode
Instruction Fetch & L1 Cache

Hyperthreading

Hyperthreads (Intel)

- Illusion of multiple cores on a single core

Switch between hardware threads for stalls

Fine grained and coarse grained

Hyperthreading

	Multi-Core	vs. Multi-Issue	vs. HT
Programs:	*N*	*1*	*N*
Num. Pipelines:	*N*	*1*	1
Pipeline Width:	*1*	*N*	*N*

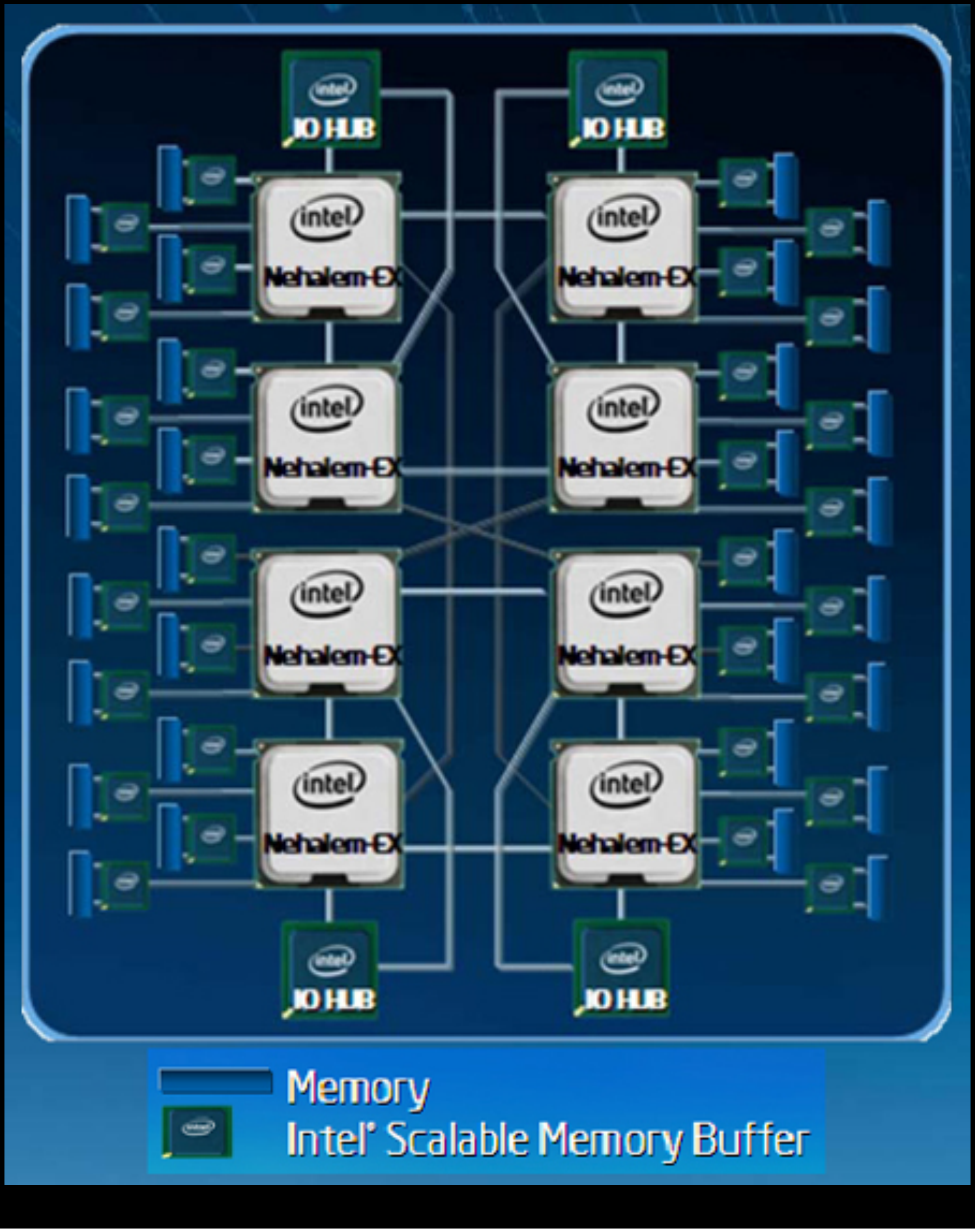
Hyperthreads

- HT = MultiIssue + extra PCs and registers – dependency logic
- HT = MultiCore – redundant functional units + hazard avoidance

Hyperthreads (Intel)

- Illusion of multiple cores on a single core
- Easy to keep HT pipelines full + share functional units

Example: All of the above



8 multiprocessors
4 core per multiprocessor
2 HT per core

Dynamic multi-issue: 4 issue
Pipeline depth: 16

Note: each processor may have multiple processing cores, so this is an example of a multiprocessor multicore hyperthreaded system

Parallel Programming

Q: So lets just all use multicore from now on!

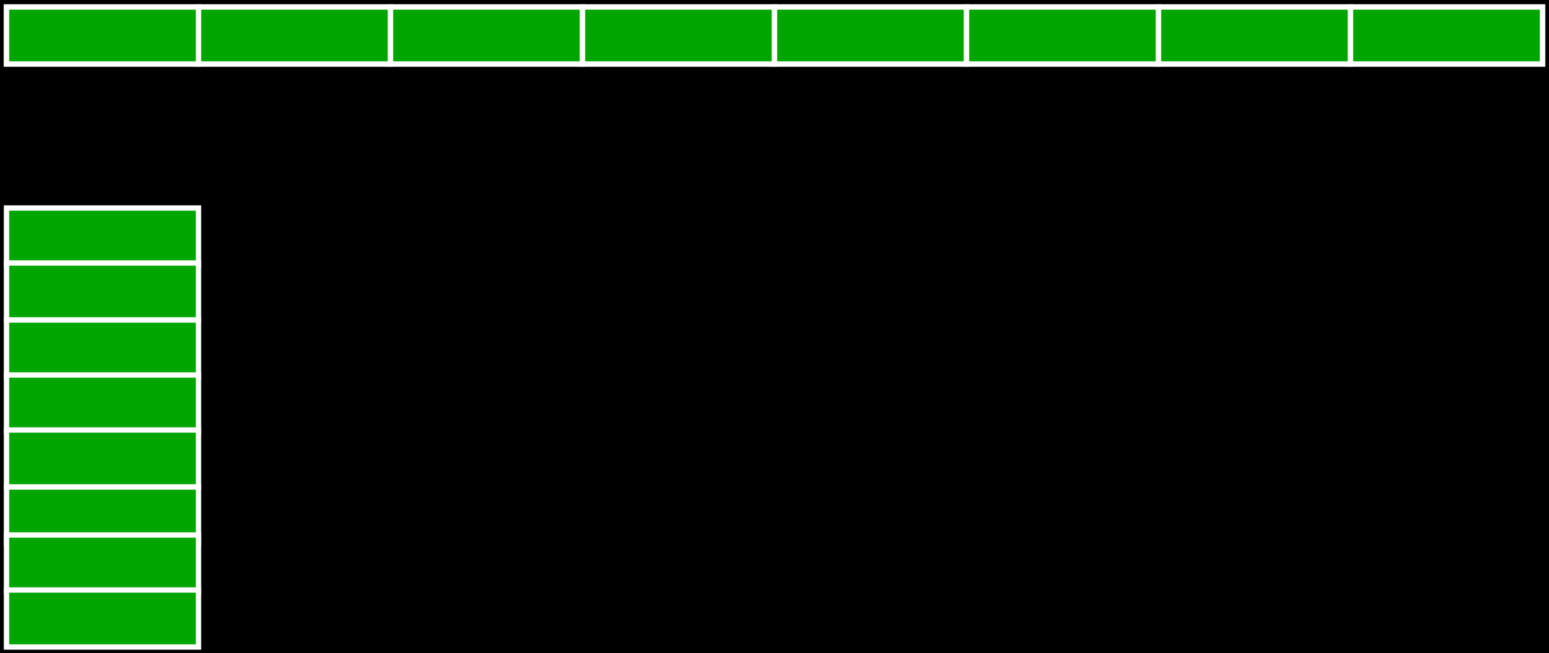
A: Software must be written as parallel program

Multicore difficulties

- Partitioning work, balancing load
- Coordination & synchronization
- Communication overhead
- How do you write parallel programs?
  - ... without knowing exact underlying architecture?

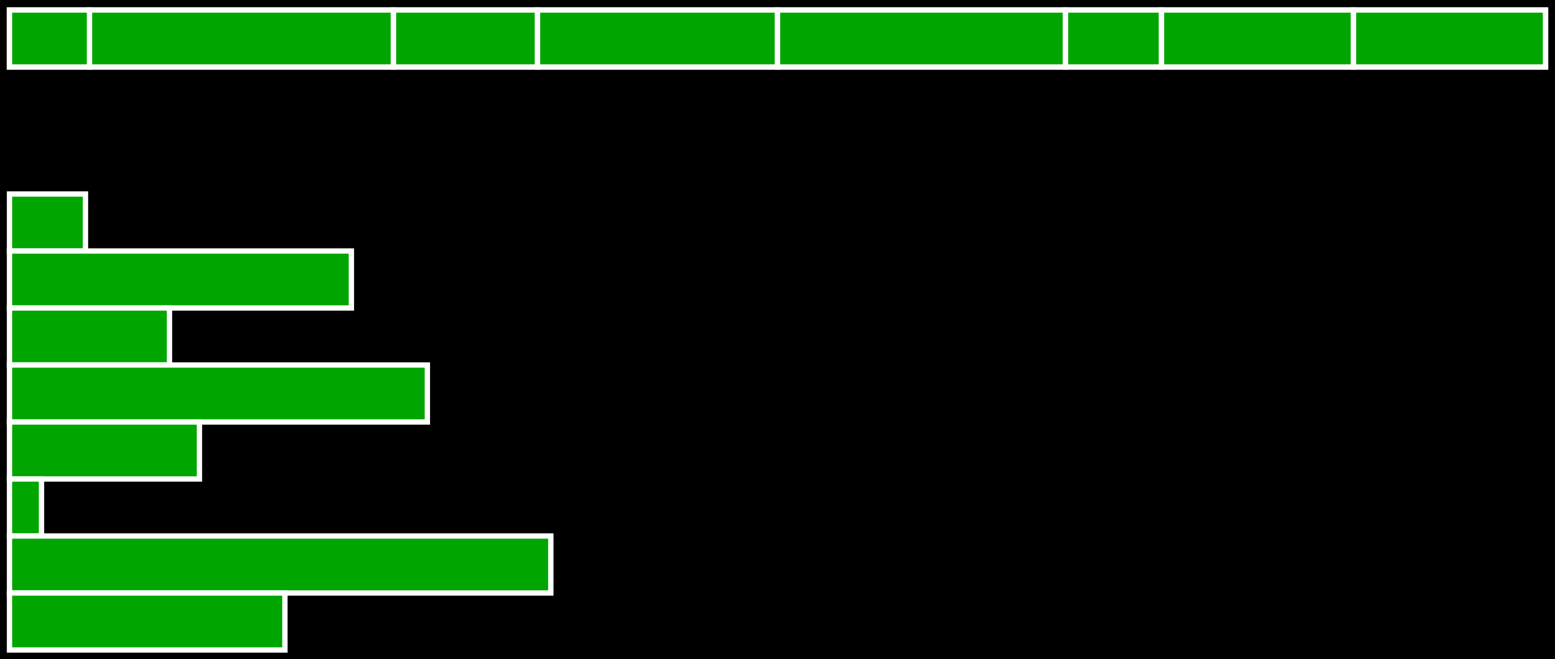
Work Partitioning

Partition work so all cores have something to do



Load Balancing

Need to partition so all cores are actually working



Amdahl's Law

If tasks have a serial part and a parallel part…

Example:

step 1: divide input data into *n* pieces

step 2: do work on each piece

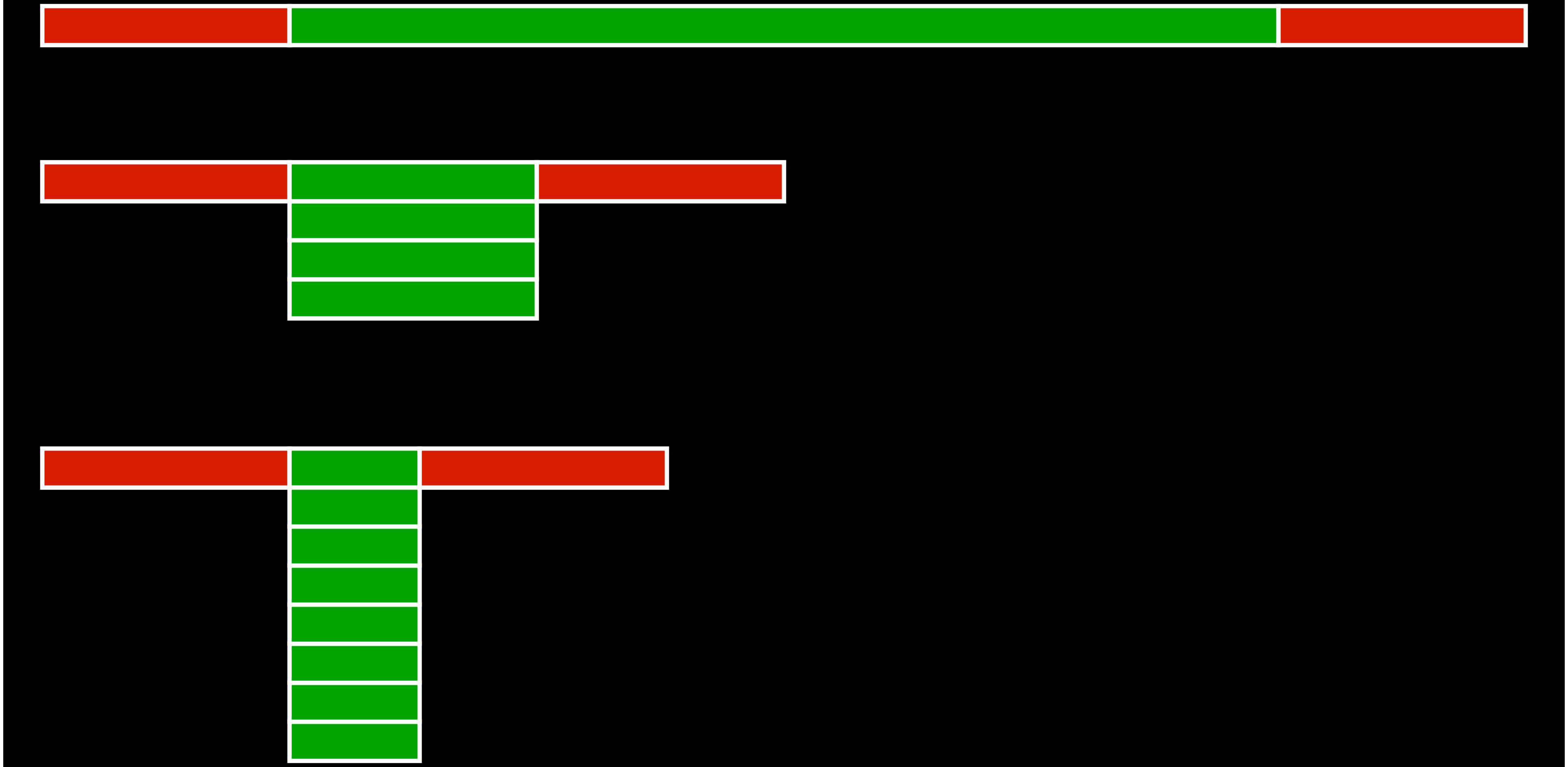
step 3: combine all results

Recall: Amdahl's Law

As number of cores increases …

- time to execute parallel part? goes to zero
- time to execute serial part? Remains the same
- *Serial part eventually dominates*

Amdahl’s Law



Pitfall: Amdahl's Law

Execution time after improvement =

$$\frac{\text{affected execution time}}{\text{amount of improvement}} + \text{execution time unaffected}$$

$$T_{\text{improved}} = \text{Taffected/improvement factor} + T_{\text{unaffected}}$$

Pitfall: Amdahl's Law

Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \text{Taffected}/\text{improvement factor} + T_{\text{unaffected}}$$

Example: multiply accounts for 80s out of 100s

- How much improvement do we need in the multiply performance to get 5× overall improvement?

$$20 = 80/n + 20$$

– ***Can't be done!***

Scaling Example

Workload: sum of 10 scalars, and 10 × 10 matrix sum

- Speed up from 10 to 100 processors?

Single processor: Time

10 processors

- Time =
- Speedup =

100 processors

- Time =
- Speedup =

Assumes load can be balanced across processors

Scaling Example

What if matrix size is 100 × 100?

Single processor: Time = $(10 + 10000) \times t_{add}$

10 processors

- Time =
- Speedup =

100 processors

- Time =
- Speedup =

Assuming load balanced

Scaling

Strong scaling vs. weak scaling

Strong scaling: scales with same problem size

Weak scaling: scales with increased problem size

Parallel Programming

Q: So lets just all use multicore from now on!

A: Software must be written as parallel program

Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- Balancing load over cores
- How do you write parallel programs?
  - ... without knowing exact underlying architecture?

HW

SW

Your career…

Synchronization

P&H Chapter 2.11 and 5.10

Parallelism and Synchronization

How do I take advantage of multiple processors; *parallelism*?

How do I write (**correct**) parallel programs*, cache coherency and synchronization*?

What primitives do I need to implement correct parallel programs?

Topics

Understand Cache Coherency

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

Parallelism and Synchronization

Cache Coherency Problem: What happens when two or more processors cache ***shared*** data?

Parallelism and Synchronization

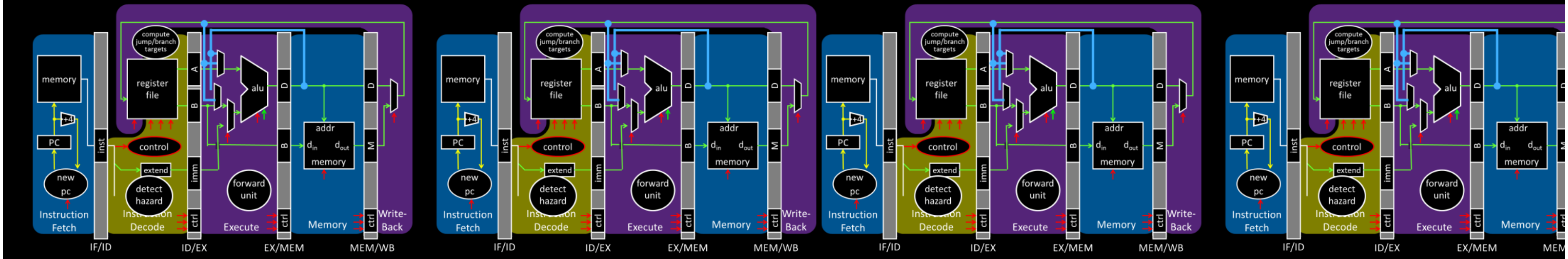
Cache Coherency Problem: What happens when two or more processors cache *shared* data?

i.e. the view of memory held by two different processors is through their individual caches

As a result, processors can see different (incoherent) values to the *same* memory location

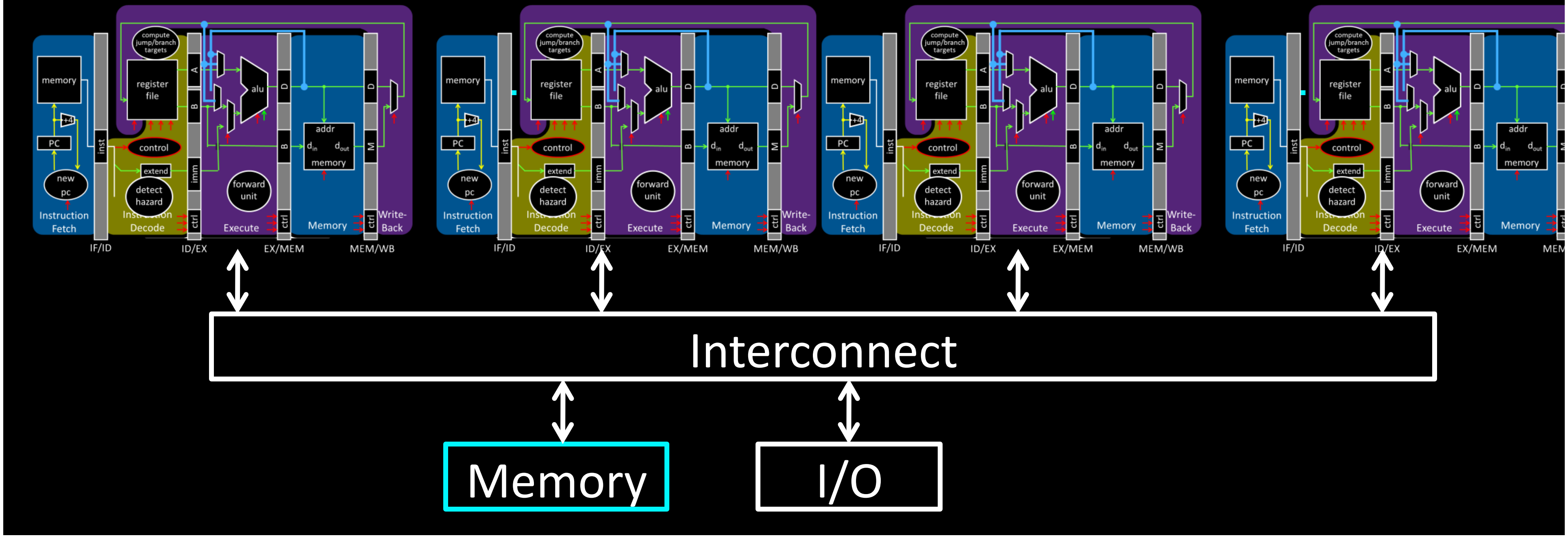
Parallelism and Synchronization

Each processor core has its own L1 cache



Parallelism and Synchronization

Each processor core has its own L1 cache



Interconnect

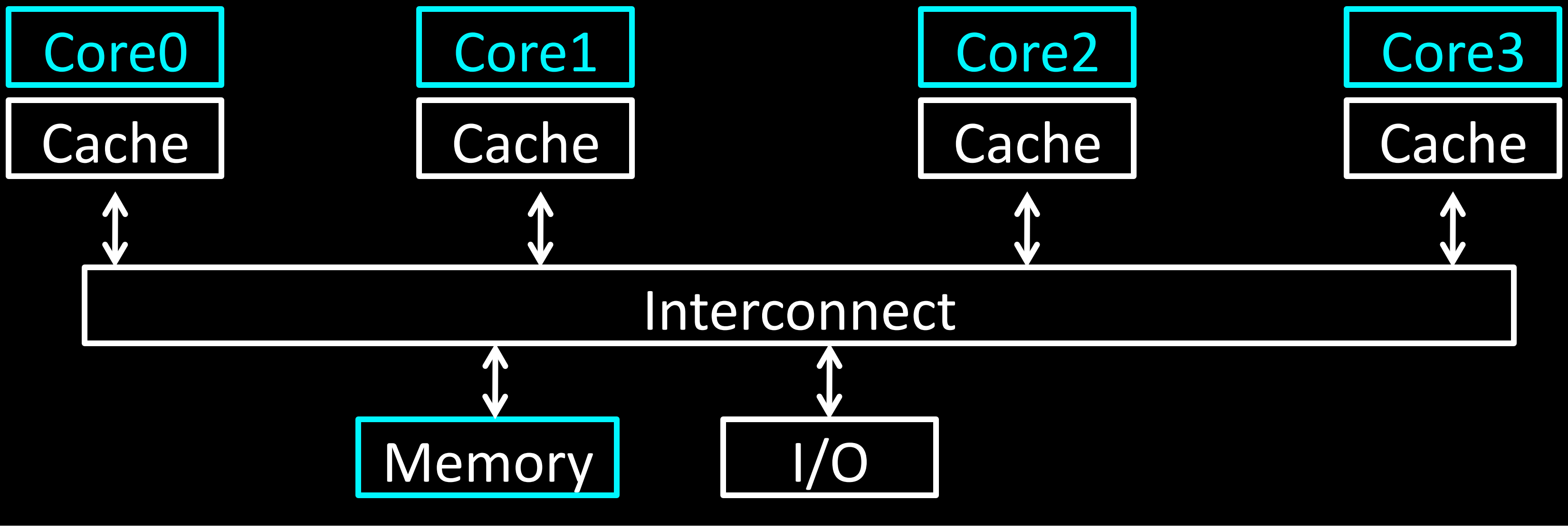
Memory

I/O

Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

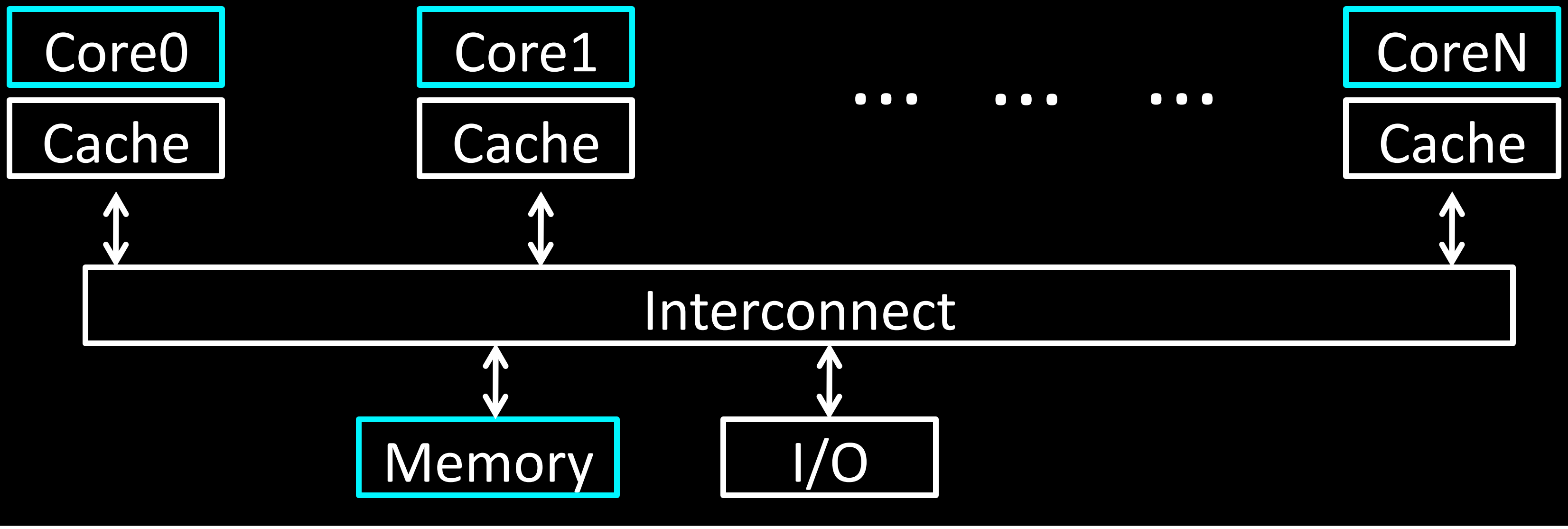
- Typical (today): 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume uniform memory access (ignore NUMA)



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume uniform memory access (ignore NUMA)



Cache Coherency Problem

Thread A (on Core0)

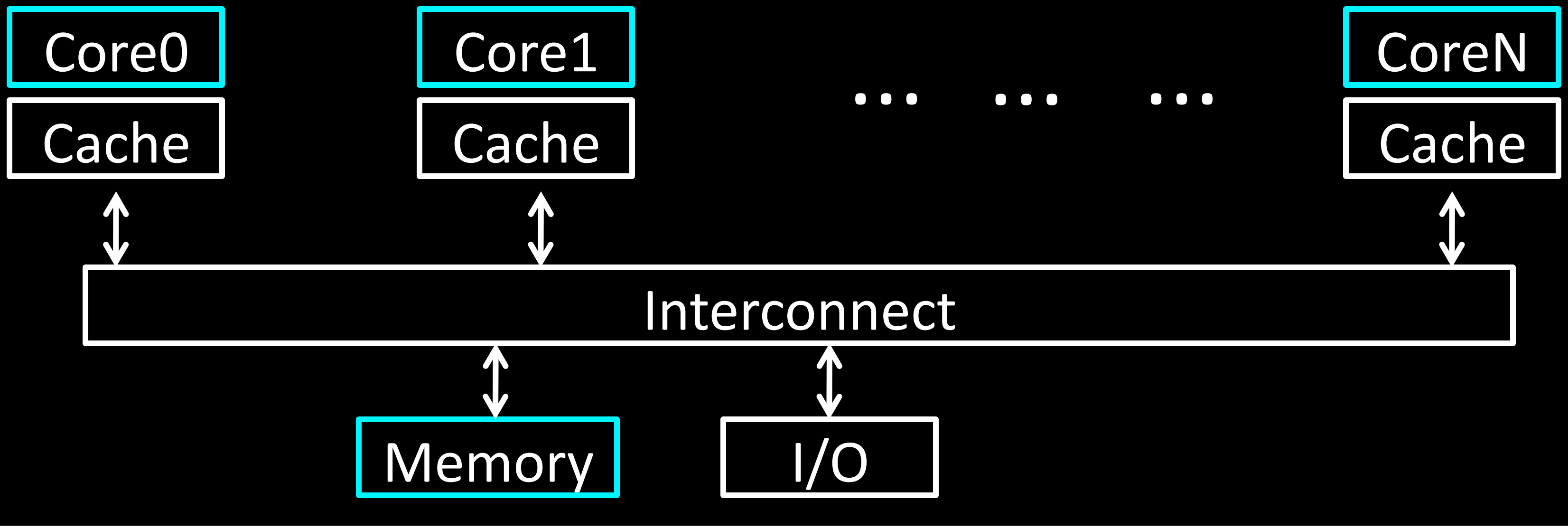
```
for(int i = 0, i < 5; i++) {
        x = x + 1;
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1;
}
```

What will the value of x be after both loops finish?

Start: x = 0



iClicker

Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        x = x + 1;
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1;
}
```

Cache Coherency Problem

Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        LW $t0, addr(x)
        ADDIU $t0, $t0, 1
        SW $t0, addr(x)
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        LW $t0, addr(x)
        ADDIU $t0, $t0, 1
        SW $t0, addr(x)
}
```

iClicker

Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        x = x + 1;
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1;
}
```

What will the value of x be after both loops finish?

a) 6

b) 8

c) 10

d) All of the above

e) None of the above