

Parallelism, Multicore, and Synchronization

CS 3410

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer. Also some slides from Amir Roth & Milo Martin in here.

Announcements

- C practice assignment
 - Due Monday, April 23rd
- P4-Buffer Overflow is due tomorrow
 - Due Wednesday, April 18th
- P5-Cache Collusion!
 - Due Friday, April 27th
 - ***Pizza party & Tournament, Monday, May 7th***
- Prelim2
 - Thursday, May 3rd, 7:30pm 185 Statler Hall

xkcd/619

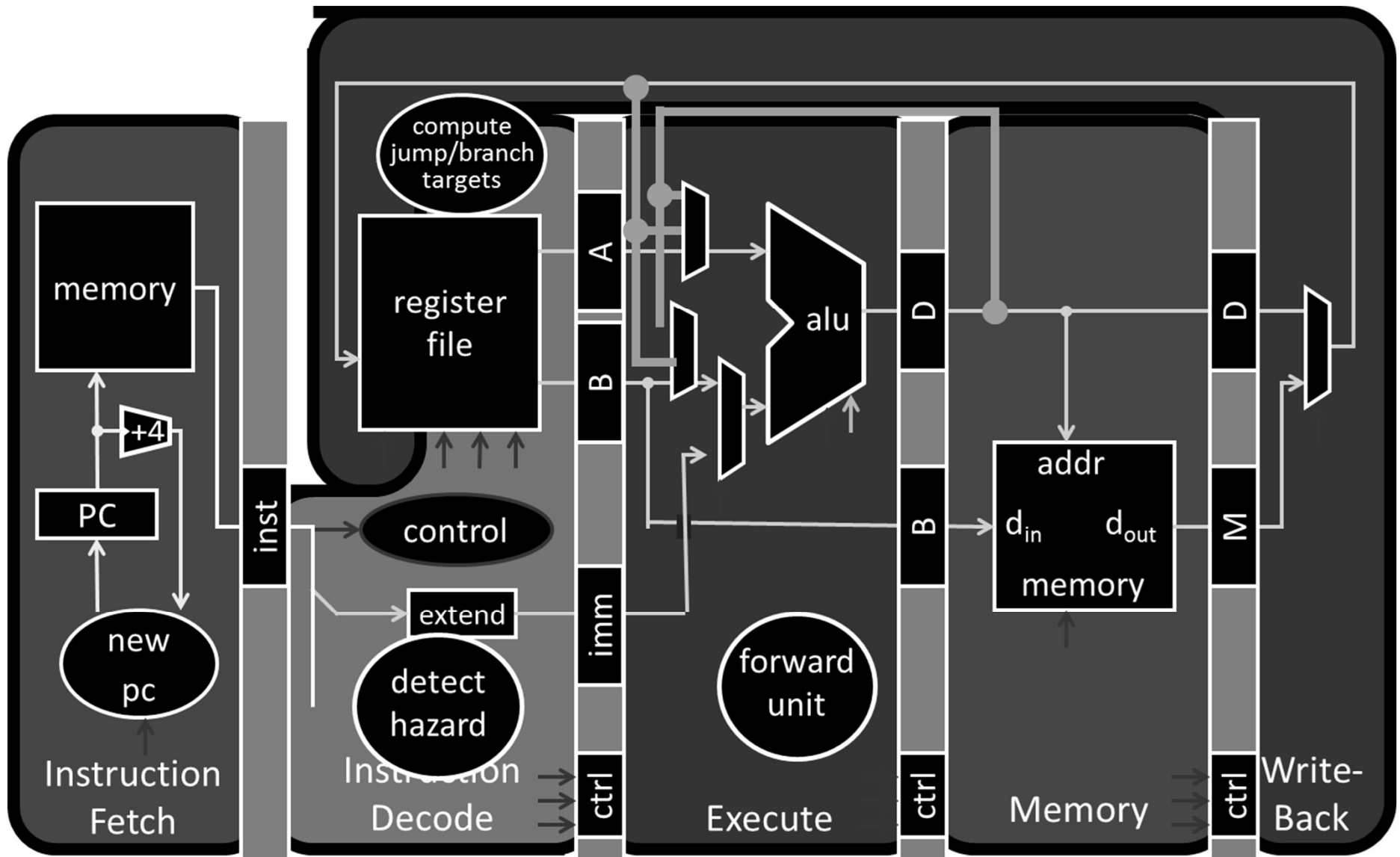
IT TOOK A LOT OF WORK, BUT THIS
LATEST LINUX PATCH ENABLES SUPPORT
FOR MACHINES WITH 4,096 CPUs,
UP FROM THE OLD LIMIT OF 1,024.

DO YOU HAVE SUPPORT FOR SMOOTH
FULL-SCREEN FLASH VIDEO YET?

NO, BUT WHO USES THAT?

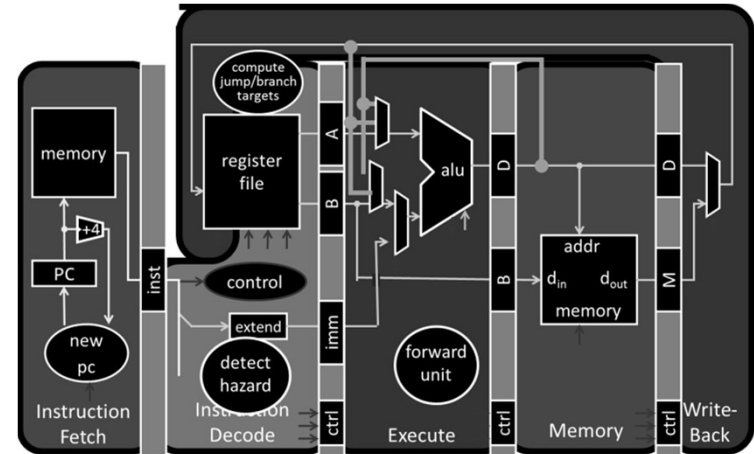
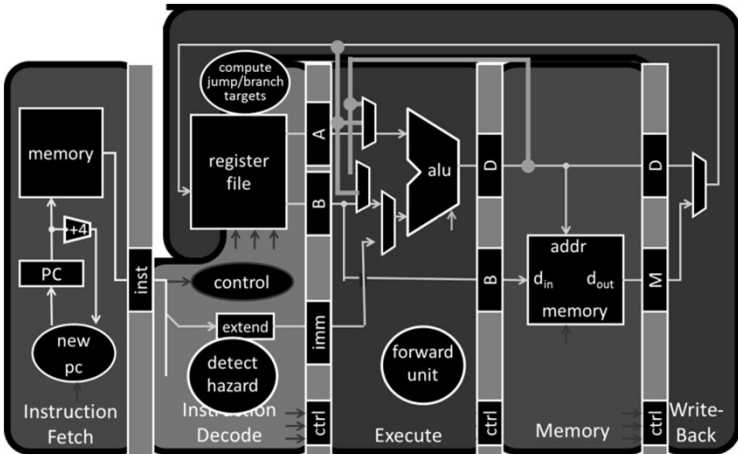
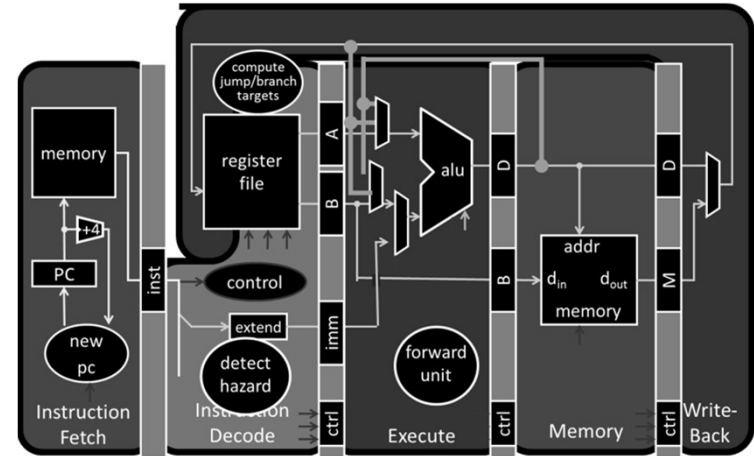
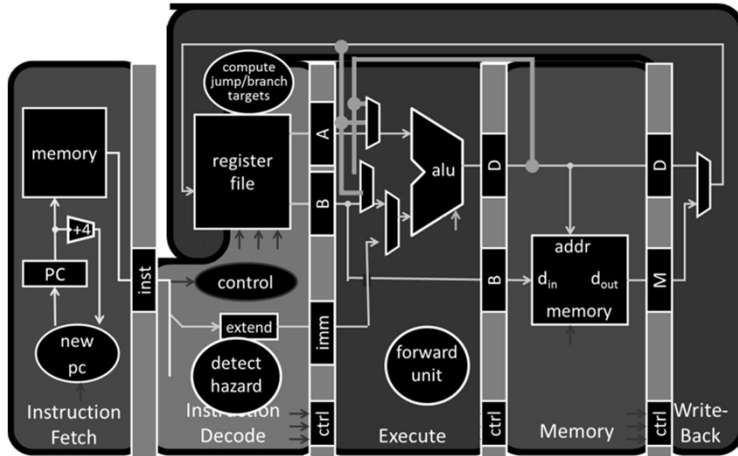


Big Picture: Multicore and Parallelism



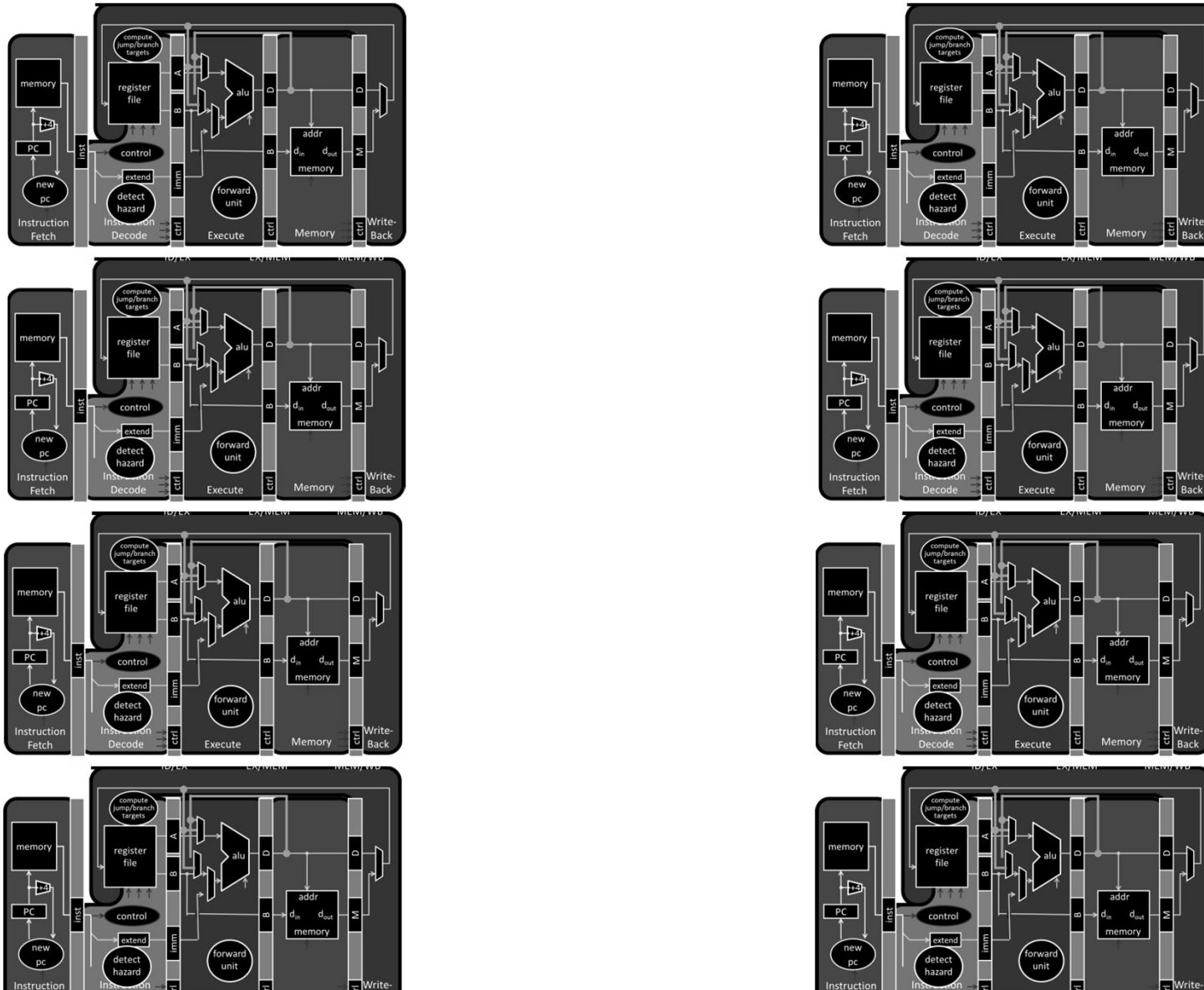
Big Picture: Multicore and Parallelism

Why do I need *four* computing cores on my phone?!



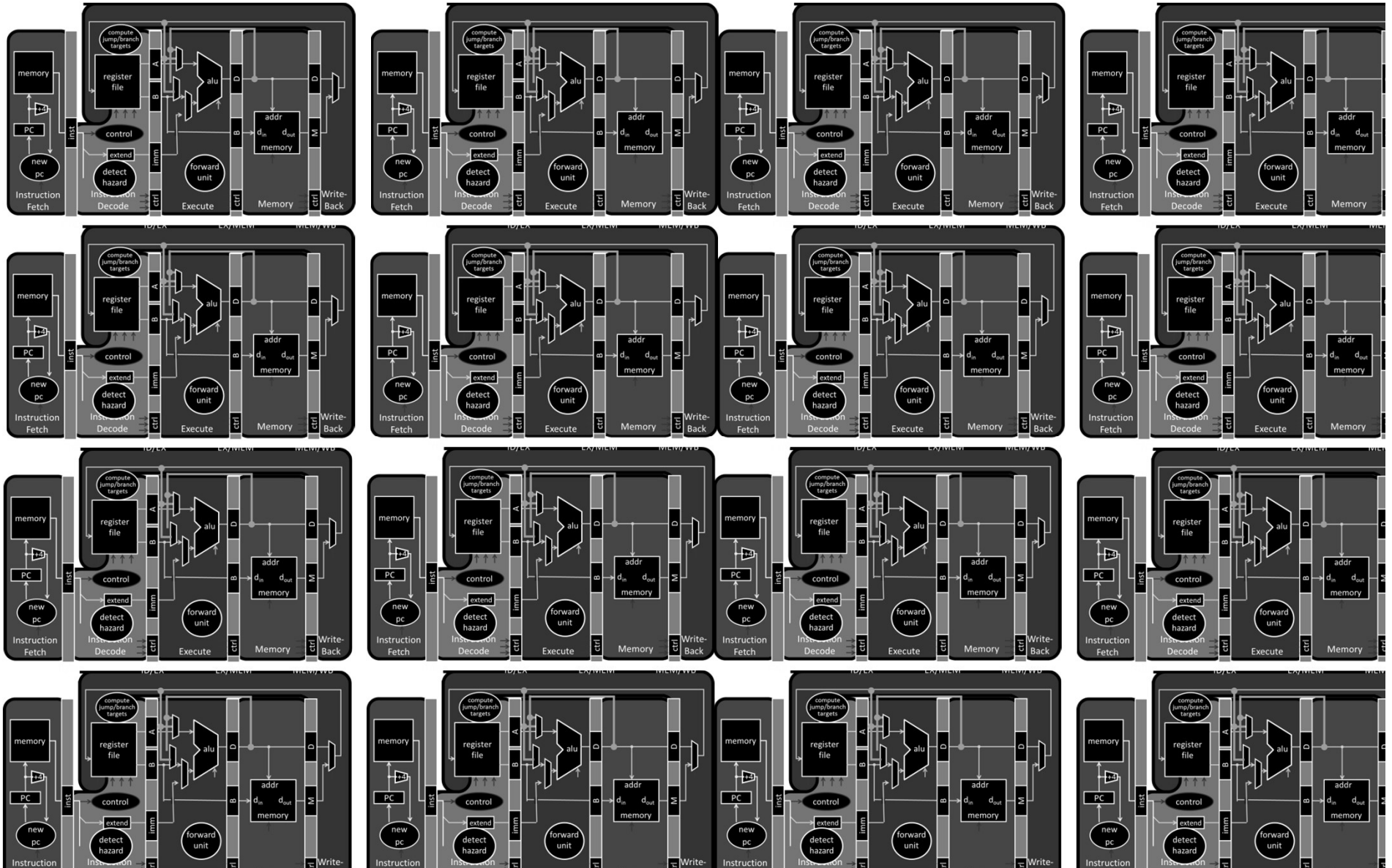
Big Picture: Multicore and Parallelism

Why do I need *eight* computing cores on my phone?!



Big Picture: Multicore and Parallelism

Why do I need *sixteen* computing cores on my phone?!



Pitfall: Amdahl's Law

Execution time after improvement =
affected execution time

amount of improvement

+ execution time unaffected

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Pitfall: Amdahl's Law

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

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Example: multiply accounts for 80s out of 100s

- Multiply can be parallelized
- How much improvement do we need in the multiply performance to get 5× overall improvement?
(a) 2x (b) 10x (c) 100x (d) 1000x (e) not possible

Pitfall: Amdahl's Law

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

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Example: multiply accounts for 80s out of 100s

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

$$20 = \frac{80}{n} + 20 \quad - \text{ Can't be done!}$$

Scaling Example

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

- Speed up from 10 to 100 processors?

Single processor: Time = $(10 + 100) \times t_{\text{add}}$

10 processors

- Time = $100/10 \times t_{\text{add}} + 10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
- Speedup = $110/20 = 5.5$

100 processors

- Time = $100/100 \times t_{\text{add}} + 10 \times t_{\text{add}} = 11 \times t_{\text{add}}$
- Speedup = $110/11 = 10$

Assumes load can be balanced across processors

Scaling Example

What if matrix size is 100×100 ?

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

10 processors

- Time = $10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
- Speedup = $10010/1010 = 9.9$

100 processors

- Time = $10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
- Speedup = $10010/110 = 91$

Assuming load balanced

Takeaway

Unfortunately, we cannot not obtain unlimited scaling (speedup) by adding unlimited parallel resources, eventual performance is dominated by a component needing to be executed sequentially.

Amdahl's Law is a caution about this diminishing return

Performance Improvement 101

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

2 Classic Goals of Architects:

↓ Clock period (↑ Clock frequency)

↓ Cycles per Instruction (↑ IPC)

Clock frequencies have stalled

Darling of performance improvement for decades

Why is this no longer the strategy?

Hitting Limits:

- Pipeline depth
- Clock frequency
- Moore's Law & Technology Scaling
- Power

Improving IPC via ILP

Exploiting Intra-instruction parallelism:

Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):

Multiple issue pipeline (2-wide, 4-wide, *etc.*)

- Statically detected by compiler (VLIW)
- Dynamically detected by HW

Dynamically Scheduled (OoO)

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 (less work per stage \Rightarrow shorter clock cycle)
- min unit of work
- dependencies, hazards / forwarding logic

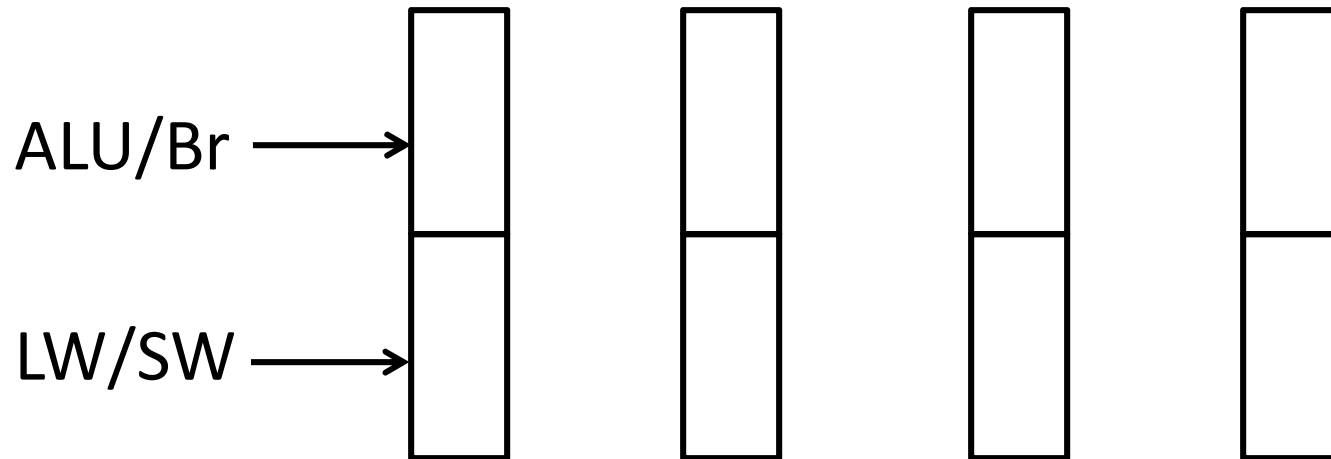
Instruction-Level Parallelism (ILP)

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”

How does HW detect and resolve hazards?

It doesn't. 😊 Compiler must avoid 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

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:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

Clicker Question: What is the IPC of this machine?

(A) 0.8 (B) 1.0 (C) 1.25 (D) 1.5 (E) 2.0

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:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1, -4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

$$\frac{5 \text{ instructions}}{4 \text{ cycles}} = \text{IPC} = 1.25$$

$$\frac{4 \text{ cycles}}{5 \text{ instructions}} = \text{CPI} = 0.8$$

Techniques and Limits of Static Scheduling

Goal: larger instruction windows (to play with)

- Predication
- Loop unrolling
- Function in-lining
- Basic block modifications (superblocks, *etc.*)

Roadblocks

- Memory dependences (aliasing)
- Control dependences

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

Compiler scheduling for dual-issue MIPS...

Loop: lw	\$t0, 0(\$s1)	# \$t0 = A[i]	} 8 cycles
lw	\$t1, 4(\$s1)	# \$t1 = A[i+1]	
addu	\$t0, \$t0, \$s2	# add \$s2	
addu	\$t1, \$t1, \$s2	# add \$s2	
sw	\$t0, -8(\$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	

ALU/branch slot	delay slot	Load/store slot	cycle
Loop: nop		lw \$t0, 0(\$s1)	1
nop		lw \$t1, 4(\$s1)	2
addu \$t0, \$t0, \$s2		nop	3
addu \$t1, \$t1, \$s2		sw \$t0, 0(\$s1)	4
addi \$s1, \$s1, +8		sw \$t1, 4(\$s1)	5
bne \$s1, \$s3, Loop		nop	6

$$\frac{6 \text{ cycles}}{8 \text{ instructions}} = \text{CPI} = 0.75$$

Scheduling Example

Compiler scheduling for dual-issue MIPS...

Loop: lw	\$t0, 0(\$s1)	# \$t0 = A[i]	} 8 cycles
lw	\$t1, 4(\$s1)	# \$t1 = A[i+1]	
addu	\$t0, \$t0, \$s2	# add \$s2	
addu	\$t1, \$t1, \$s2	# add \$s2	
sw	\$t0, -8(\$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	

ALU/branch slot	Load/store slot	cycle	
Loop: nop	lw \$t0, 0(\$s1)	1	} 5 cycles
addi \$s1, \$s1, +8	lw \$t1, 4(\$s1)	2	
addu \$t0, \$t0, \$s2	nop	3	
addu \$t1, \$t1, \$s2	sw \$t0, -8(\$s1)	4	
bne \$s1, \$s3, Loop	sw \$t1, -4(\$s1)	5	

$$\frac{5 \text{ cycles}}{8 \text{ instructions}} = \text{CPI} = 0.625$$

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS...

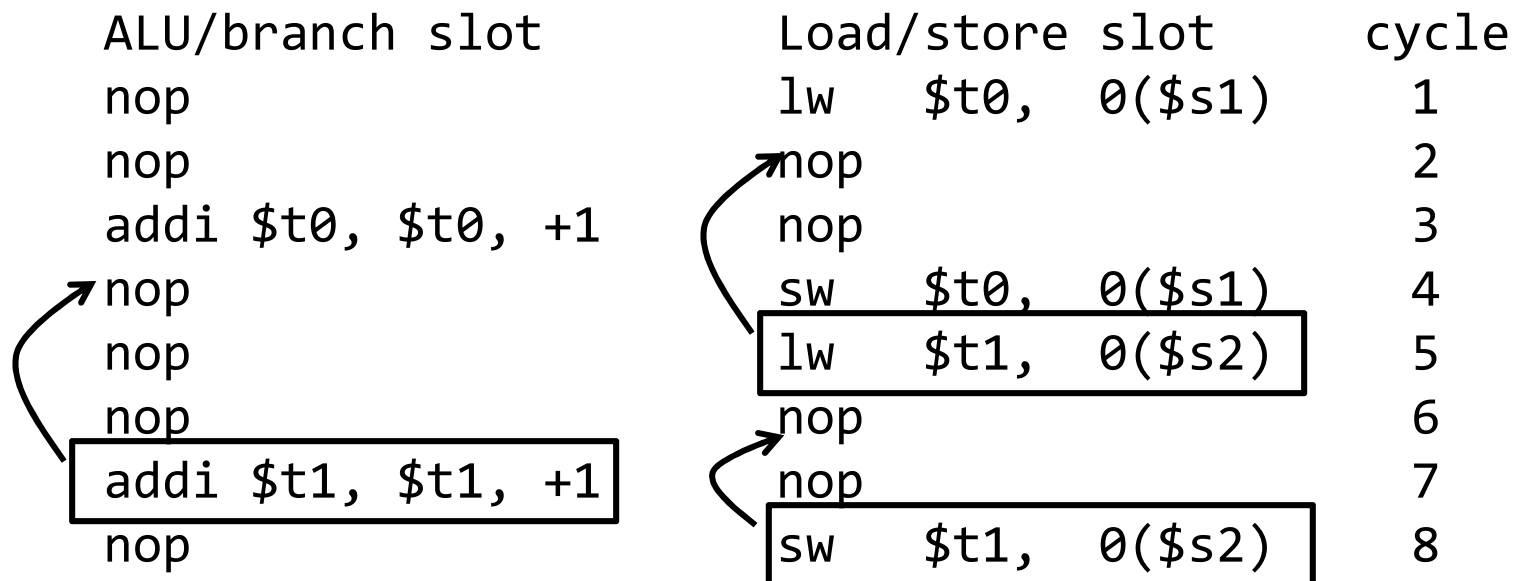
lw	\$t0, 0(\$s1)	# load A
addi	\$t0, \$t0, +1	# increment A
sw	\$t0, 0(\$s1)	# store A
lw	\$t0, 0(\$s2)	# load B
addi	\$t0, \$t0, +1	# increment B
sw	\$t0, 0(\$s2)	# store B

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
```



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

Improving IPC via ILP

Exploiting Intra-instruction parallelism:

Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):

Multiple issue pipeline (2-wide, 4-wide, *etc.*)

- Statically detected by compiler (VLIW)
- Dynamically detected by HW

Dynamically Scheduled (OoO)

Dynamic Multiple Issue

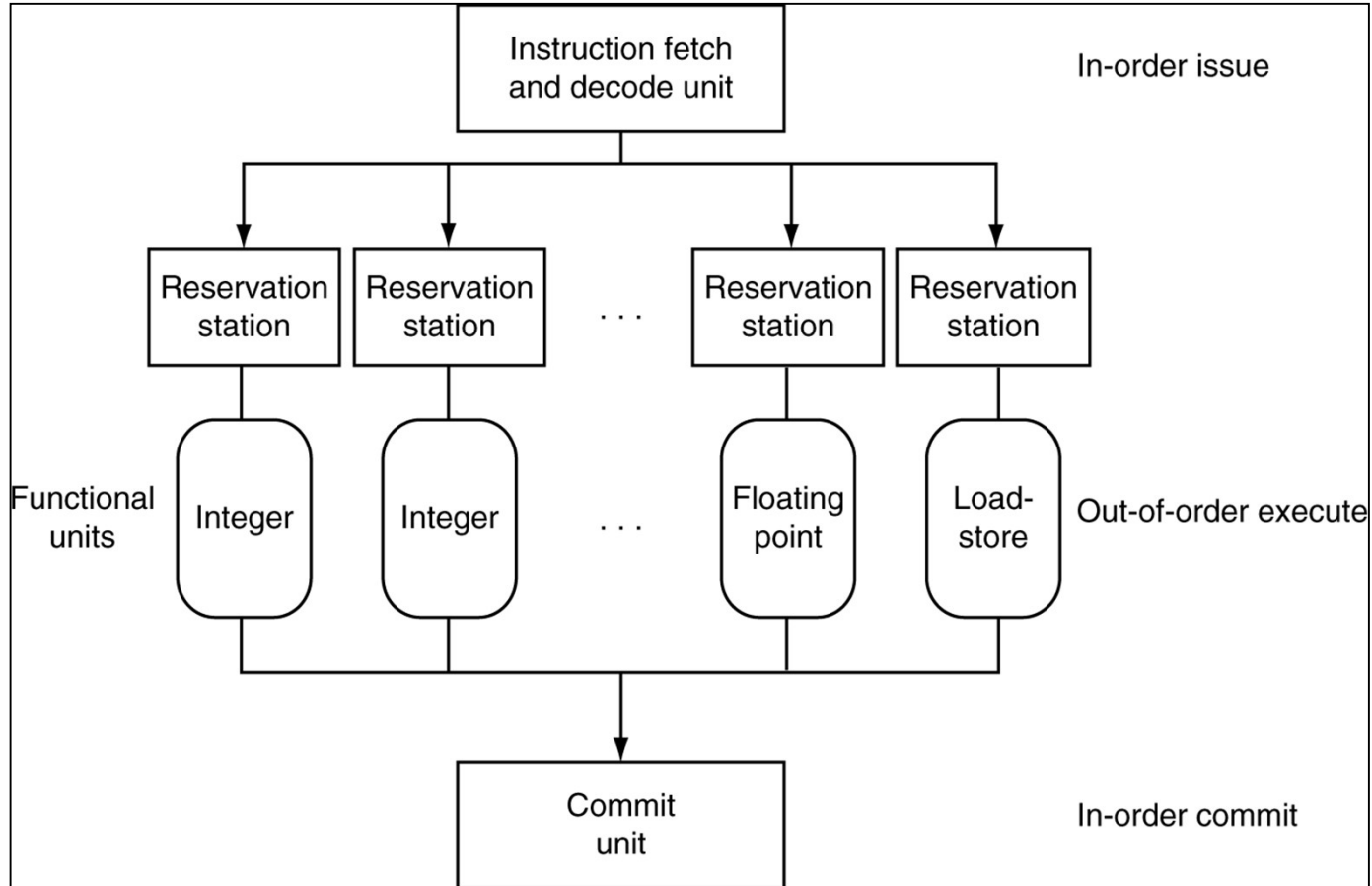
aka SuperScalar Processor (c.f. Intel)

- CPU chooses multiple instructions to issue each cycle
- Compiler can help, by reordering instructions....
- ... but CPU resolves hazards

Even better: Speculation/Out-of-order Execution

- Execute instructions as early as possible
- Aggressive register renaming (indirection to the rescue!)
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don't commit results until all previous insns committed

Dynamic Multiple Issue



Effectiveness of OoO Superscalar

It was awesome, but then it stopped improving

Limiting factors?

- Programs dependencies
- Memory dependence detection → be conservative
 - e.g. Pointer Aliasing: $A[0] += 1; B[0] *= 2;$
- Hard to expose parallelism
 - Still limited by the fetch stream of the static program
- Structural limits
 - Memory delays and limited bandwidth
- Hard to keep pipelines full, especially with branches

Improving IPC via ~~ILP~~ TLP

Exploiting Thread-Level parallelism

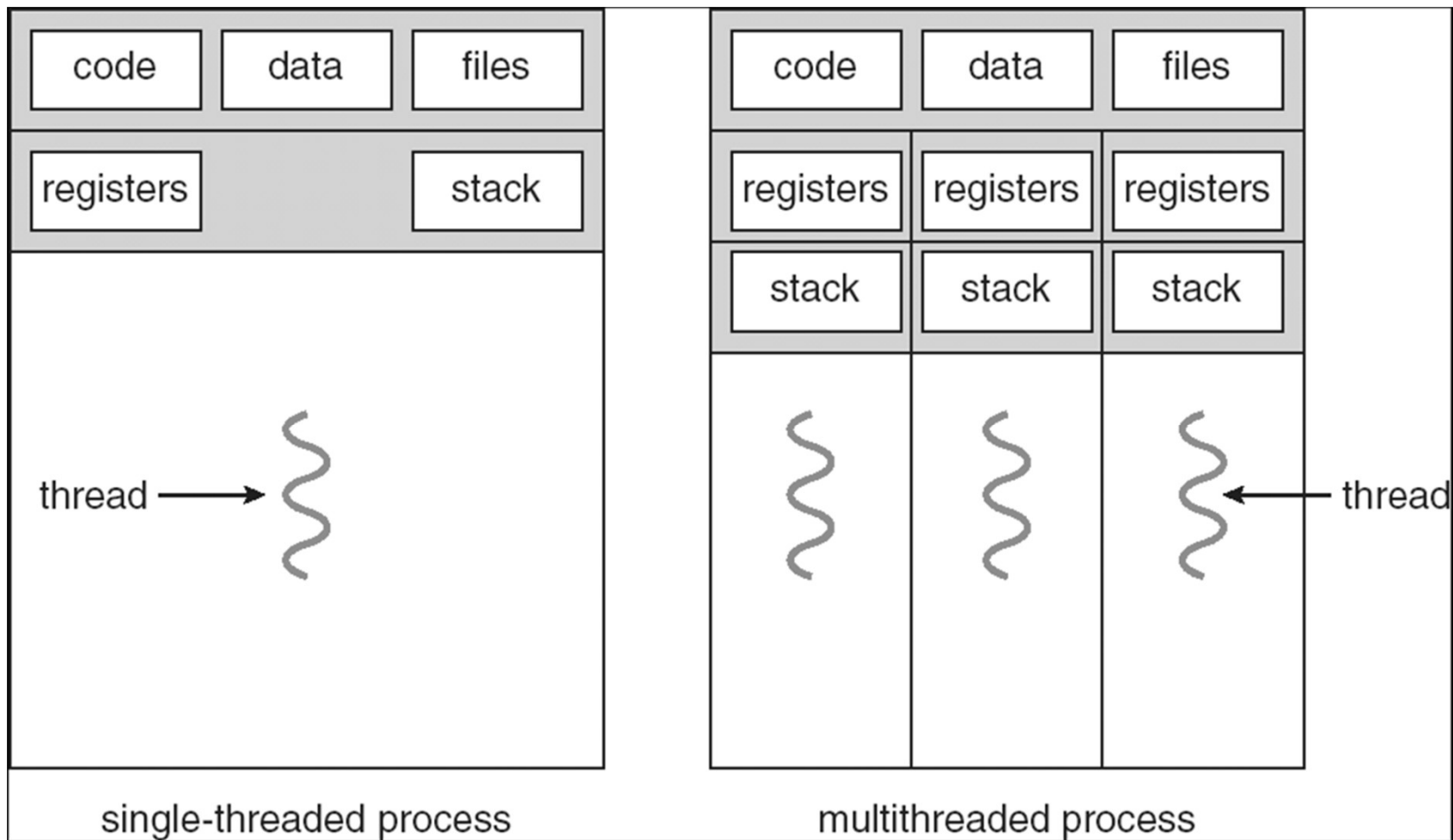
Hardware multithreading to improve utilization:

- Multiplexing multiple threads on single CPU
- Sacrifices latency for throughput
- Single thread cannot fully utilize CPU? *Try more!*
- Three types:
 - Course-grain (has preferred thread)
 - Fine-grain (round robin between threads)
 - Simultaneous (hyperthreading)

What is a thread?

Process: multiple threads, code, data and OS state

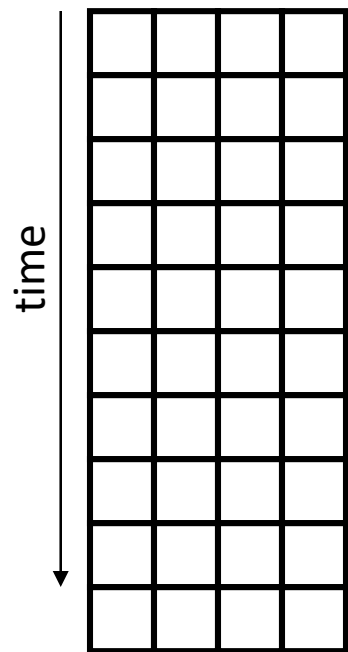
Threads: share code, data, files, **not** regs or stack



Standard Multithreading Picture

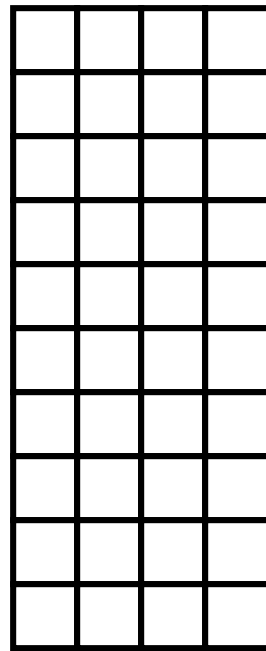
Time evolution of issue slots

- Color = thread, white = no instruction



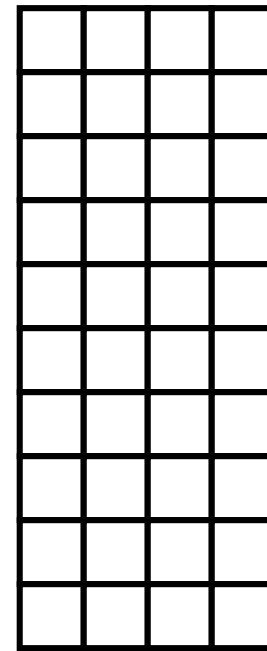
4-wide

Superscalar



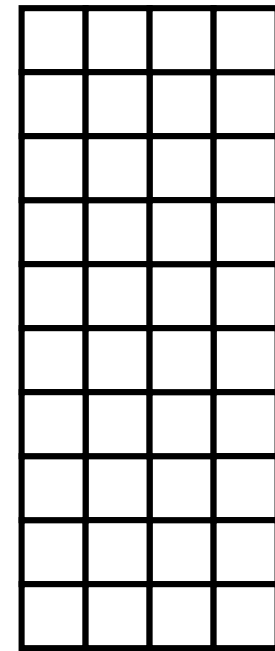
CGMT

Switch to
thread B on
thread A L2
miss



FGMT

Switch
threads
every cycle



SMT

Insns from
multiple
threads
coexist

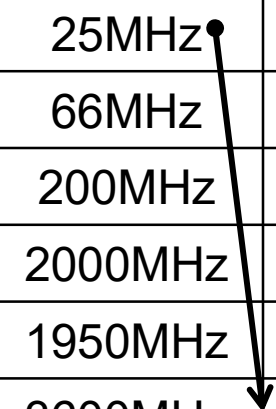
Power Efficiency

Q: Does multiple issue / ILP cost much?

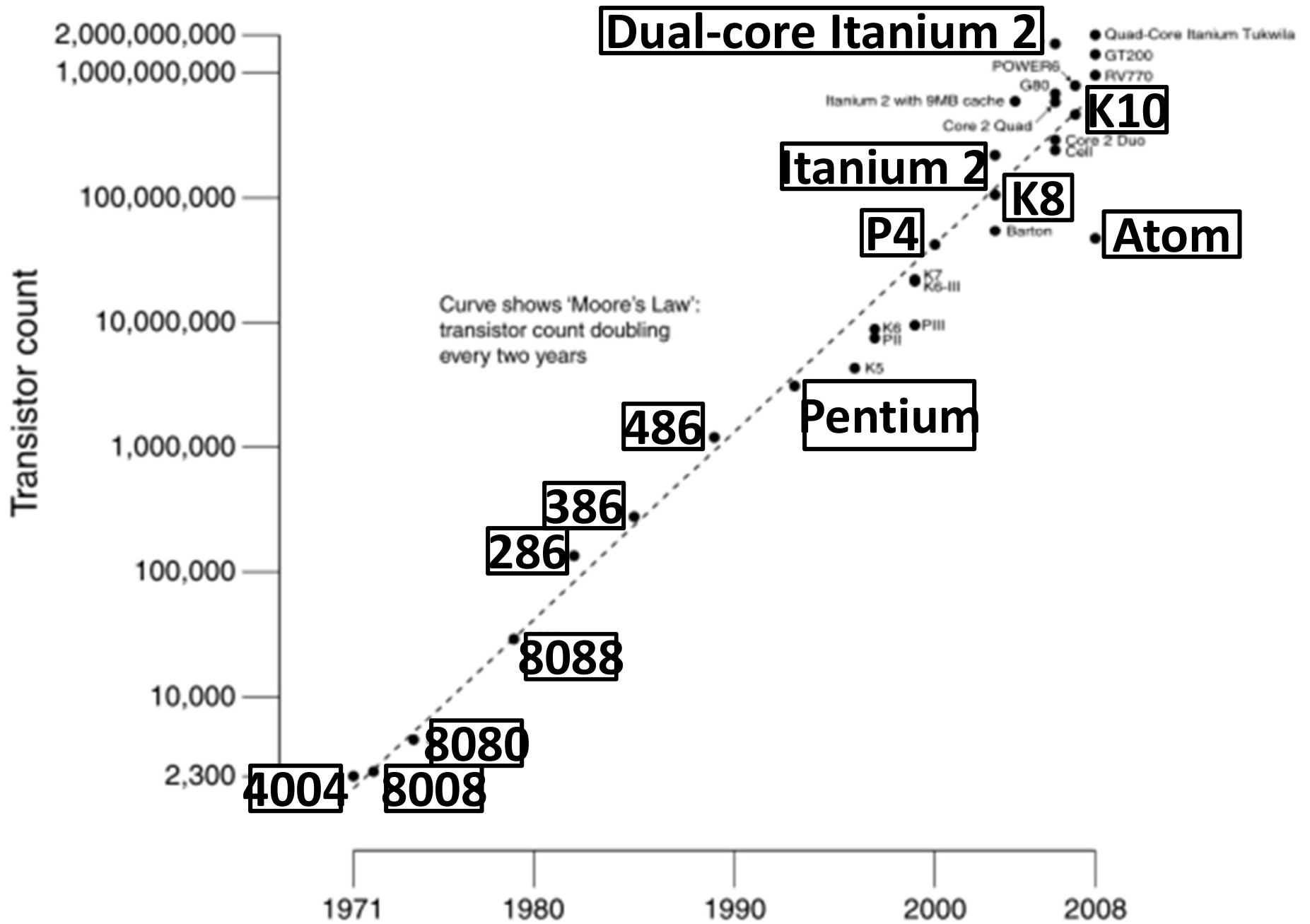
A: Yes.

→ Dynamic issue and speculation requires power

CPU	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W



Those simpler cores did something very right.



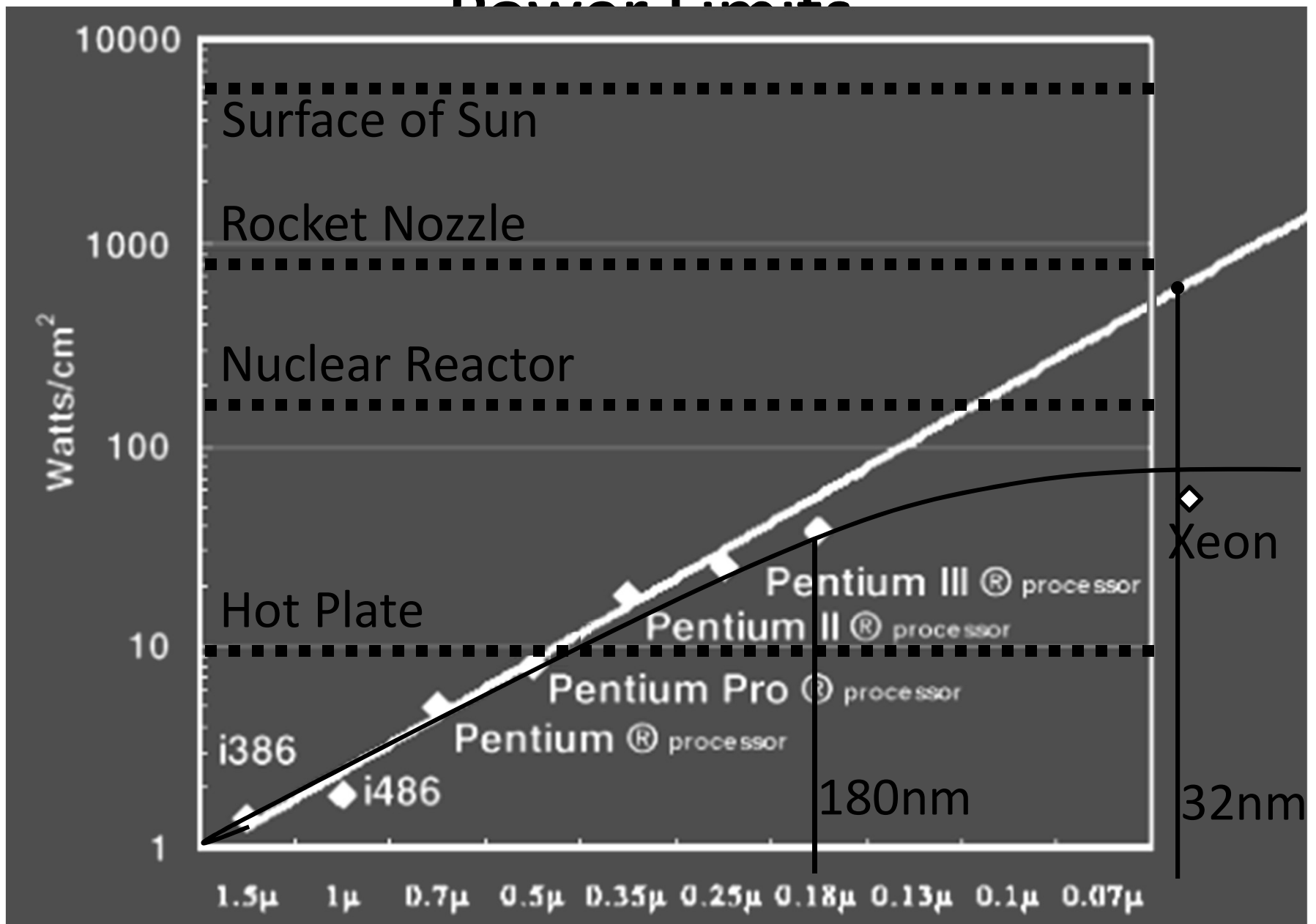
Why Multicore?

Moore's law

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

But: Power consumption growing too...

Power Limits



Power Wall

Power = capacitance * voltage² * frequency

In practice: Power ~ voltage³ Lower Frequency

Reducing voltage helps (a lot)

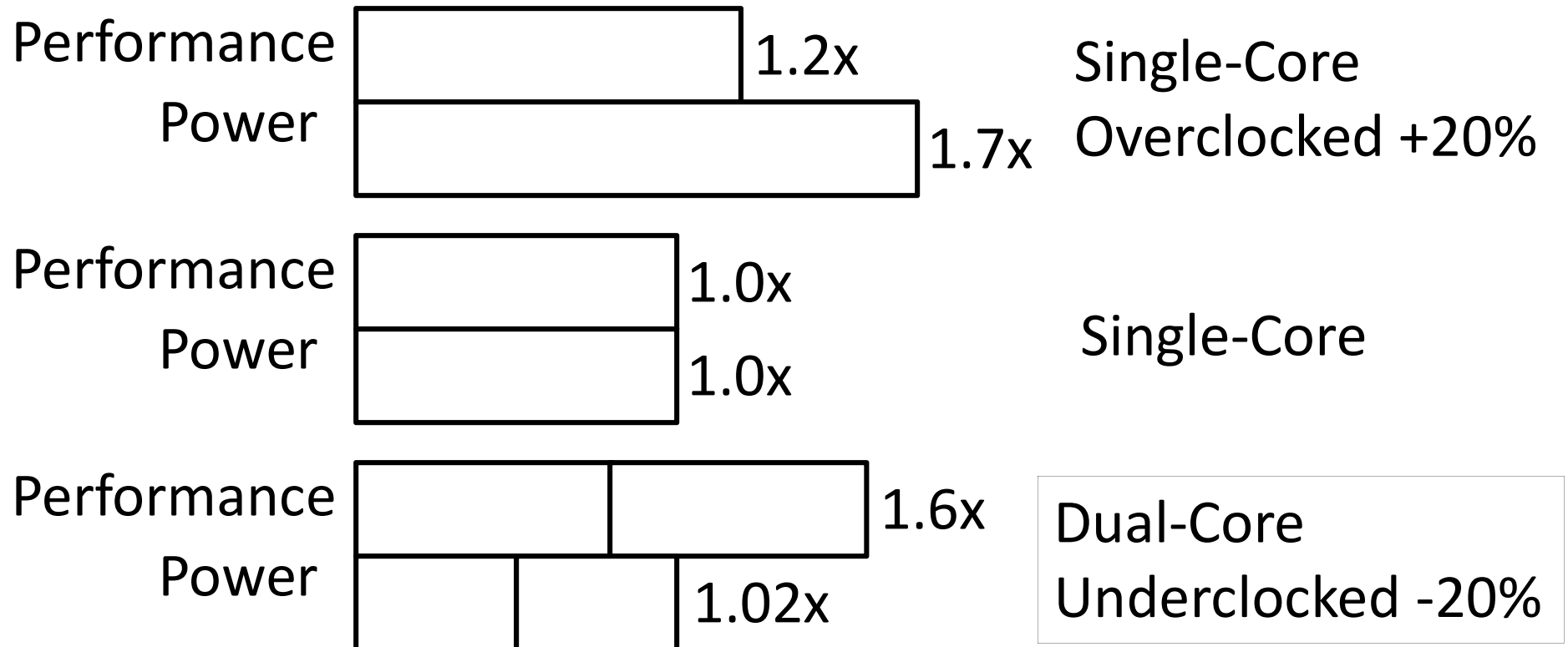
... so does reducing clock speed

Better cooling helps

The power wall

- We can't reduce voltage further
- We can't remove more heat

Why Multicore?



Power Efficiency

Q: Does multiple issue / ILP cost much?

A: Yes.

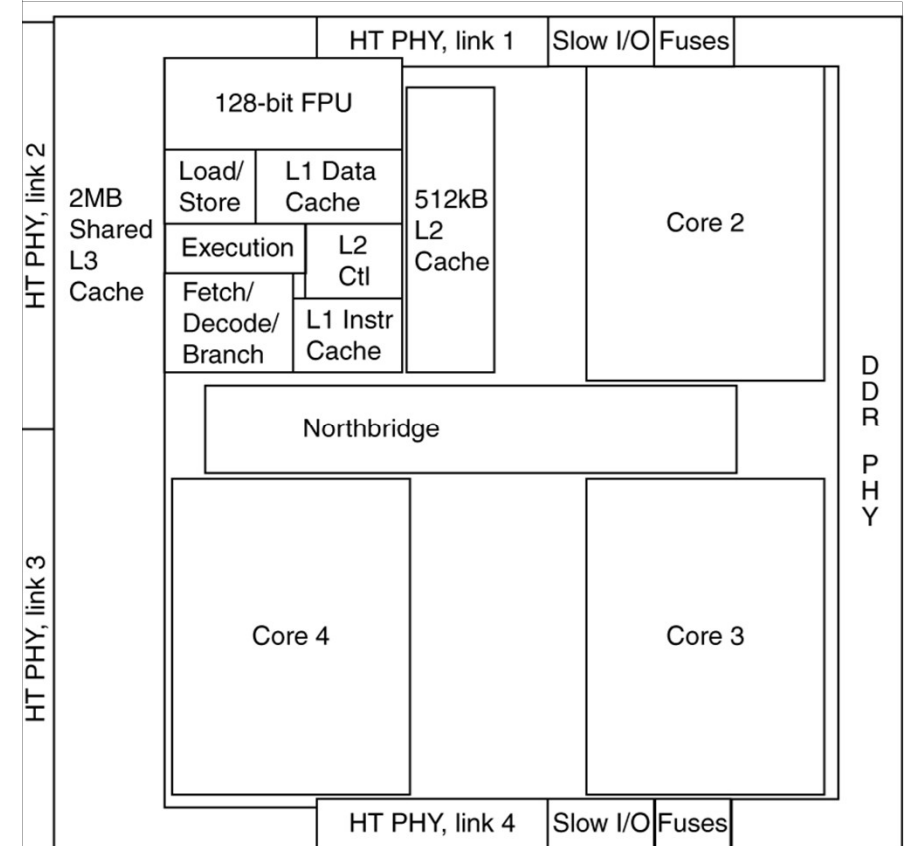
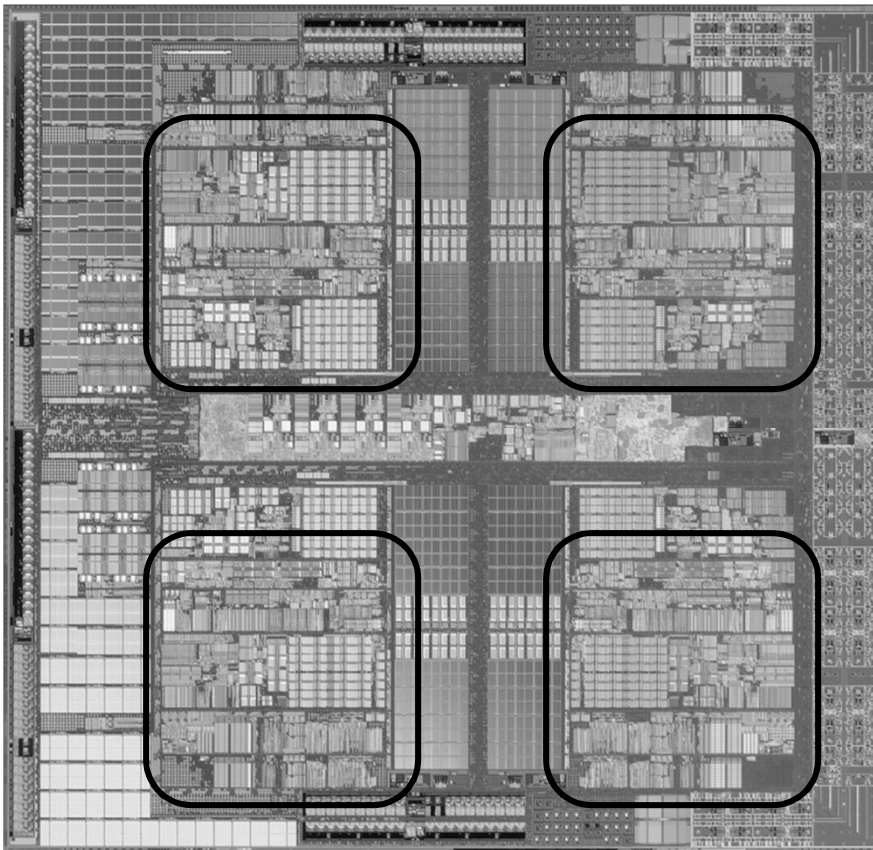
→ Dynamic issue and speculation requires power

CPU	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
Core i5 Nehal	2010	3300MHz	14	4	Yes	1	87W
Core i5 Ivy Br	2012	3400MHz	14	4	Yes	8	77W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

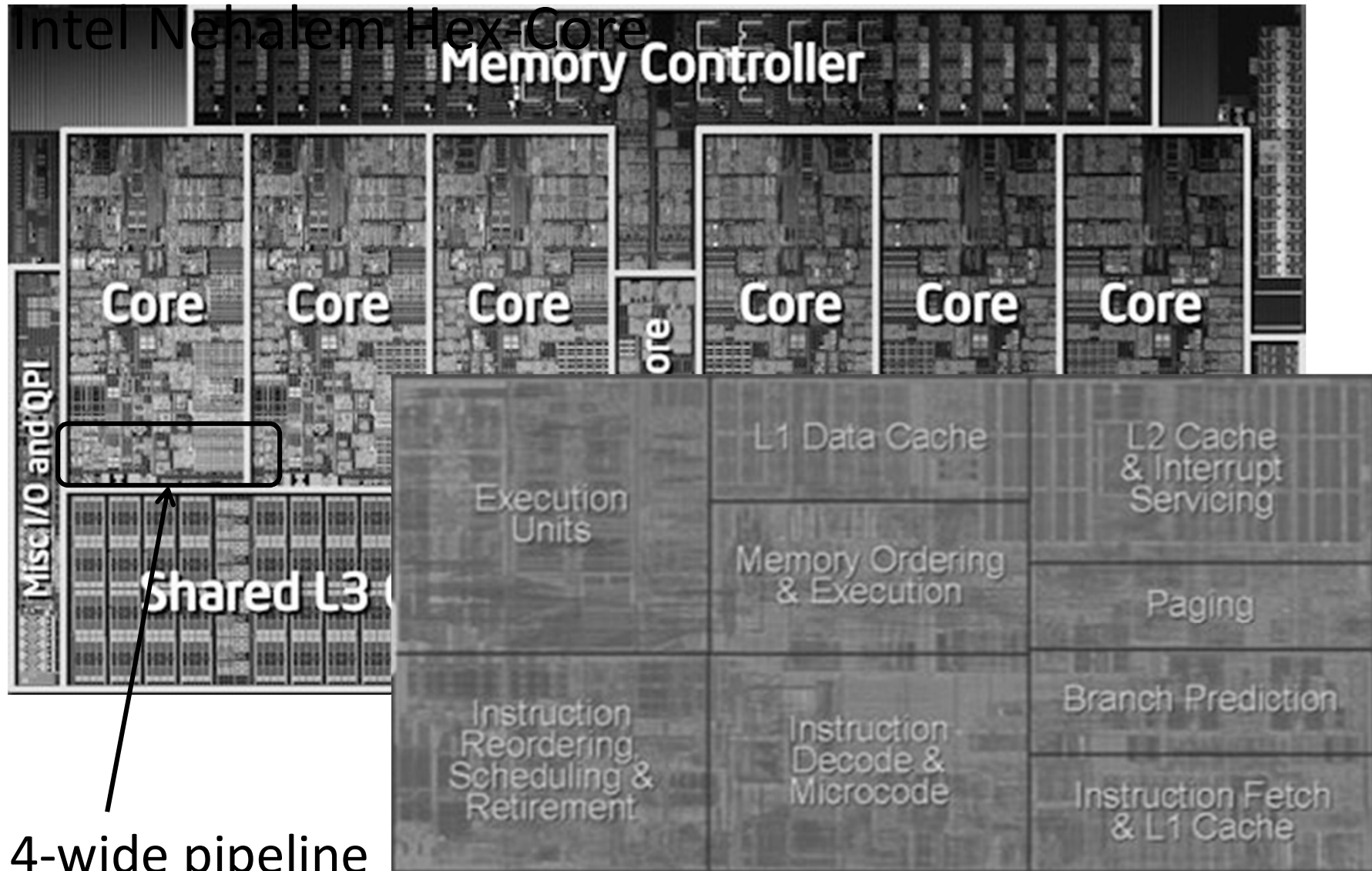
Those simpler cores did something very right.

Inside the Processor

AMD Barcelona Quad-Core: 4 processor cores



Inside the Processor



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

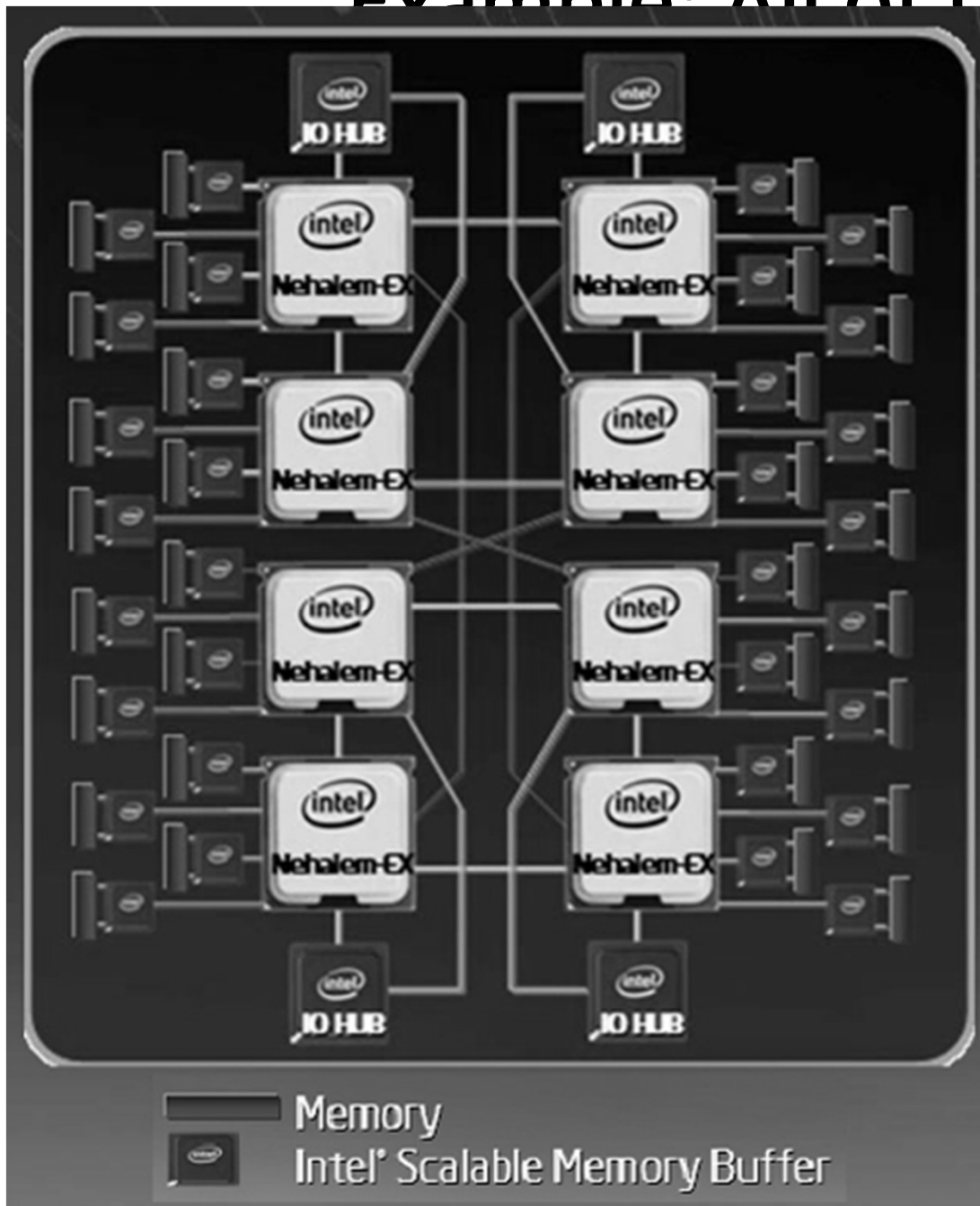
Hyperthreads (Intel)

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

Example: All of the above

8 die (aka 8 sockets)
4 core per socket
2 HT per core

Note: a socket is a processor, where 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
- Coordination & synchronization
- Communications overhead
- How do you write parallel programs?
 - ... without knowing exact underlying architecture?

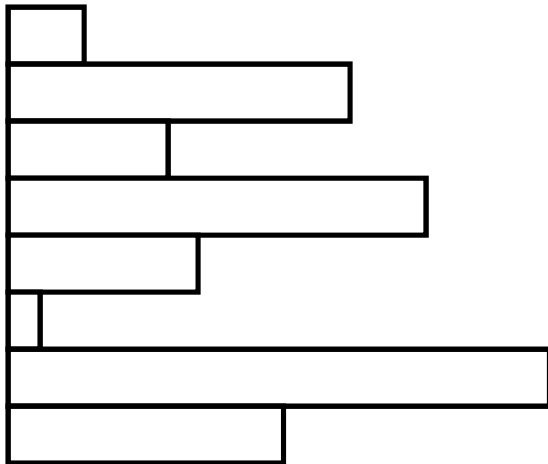
Work Partitioning

Partition work so all cores have something to do



Load Balancing **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

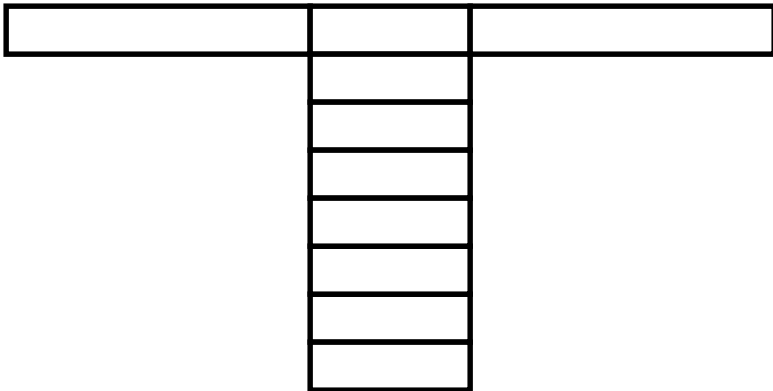
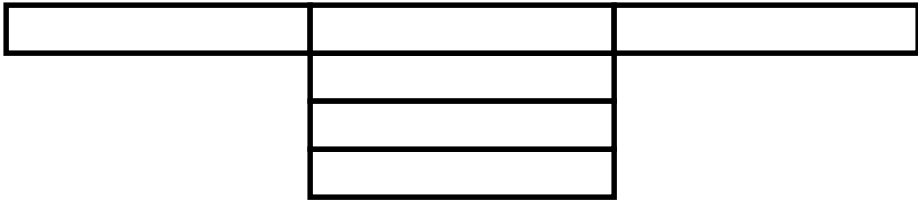
step 3: combine all results

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



Parallelism is a necessity

Necessity, not luxury

Power wall

Not easy to get performance out of

Many solutions

Pipelining

Multi-issue

Hyperthreading

Multicore

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 HW
 - How do you write parallel programs?
... without knowing exact underlying architecture?
- SW
Your
career...

Big Picture: Parallelism and Synchronization

How do I take advantage of *parallelism*?

How do I write **(correct)** parallel programs?

What primitives do I need to implement correct parallel programs?

Parallelism & Synchronization

Cache Coherency

- Processors cache *shared* data → they see different (incoherent) values for the *same* memory location

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?

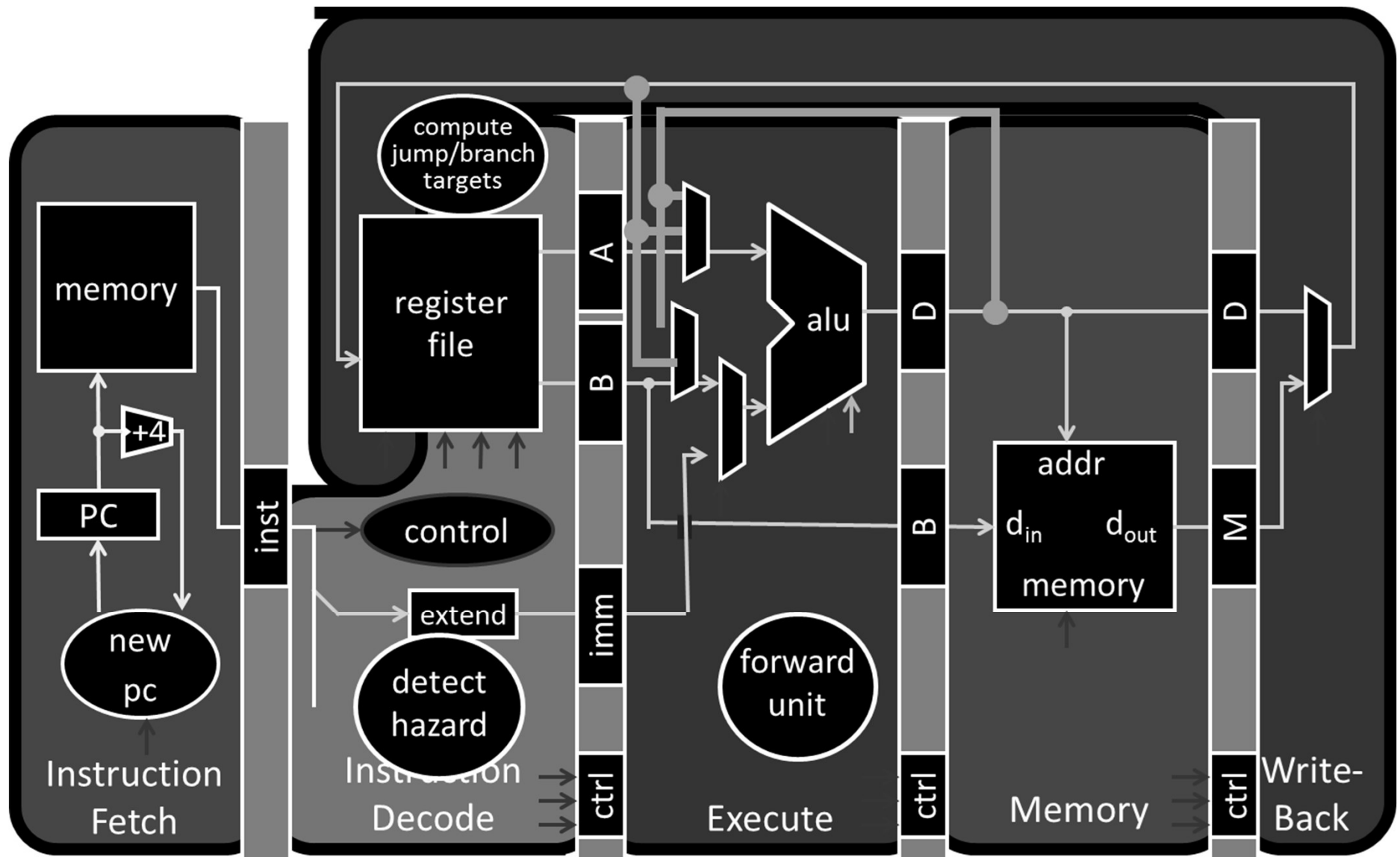
Parallelism and Synchronization

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.

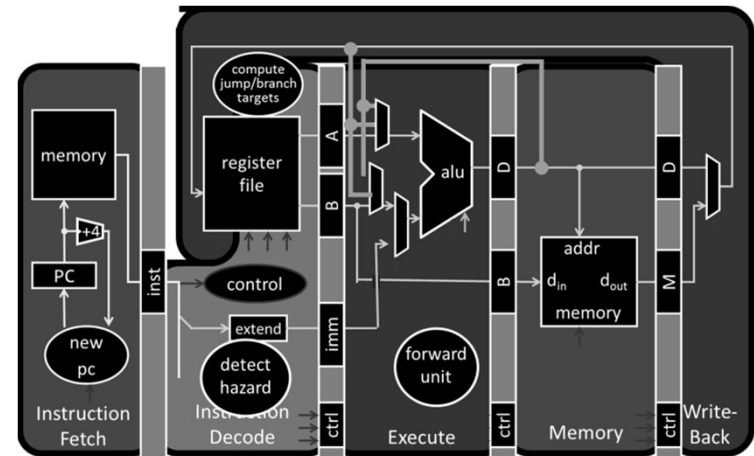
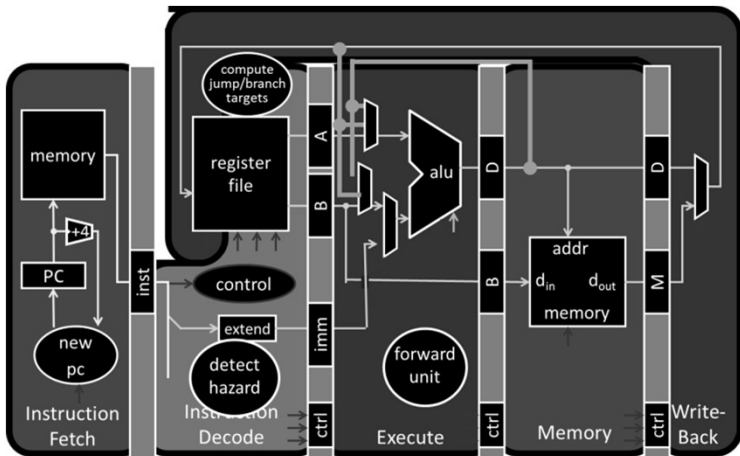
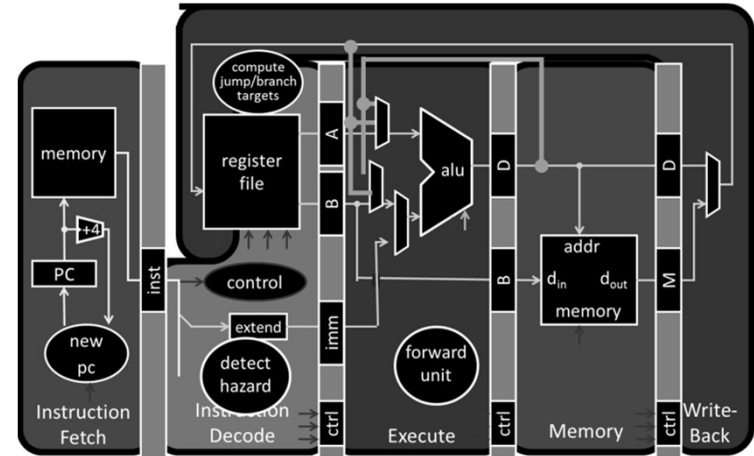
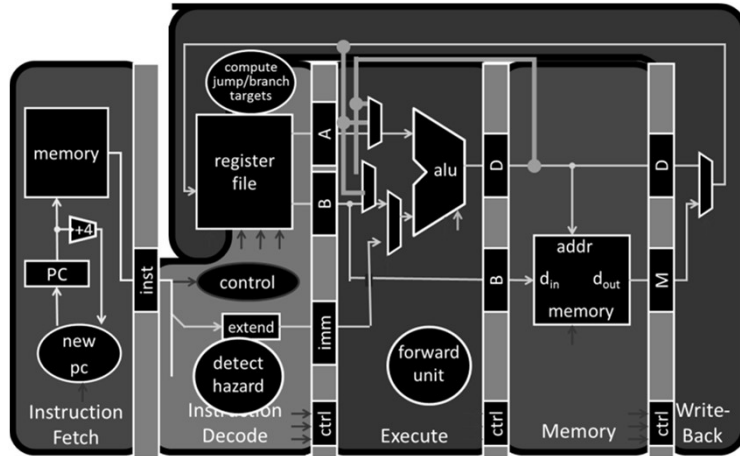
As a result, processors can see different (incoherent) values to the *same* memory location.

Parallelism and Synchronization

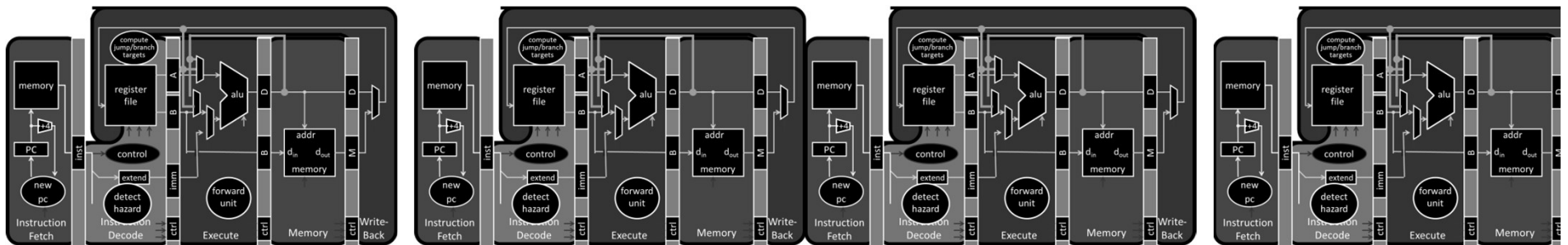


Parallelism and Synchronization

Each processor core has its own L1 cache

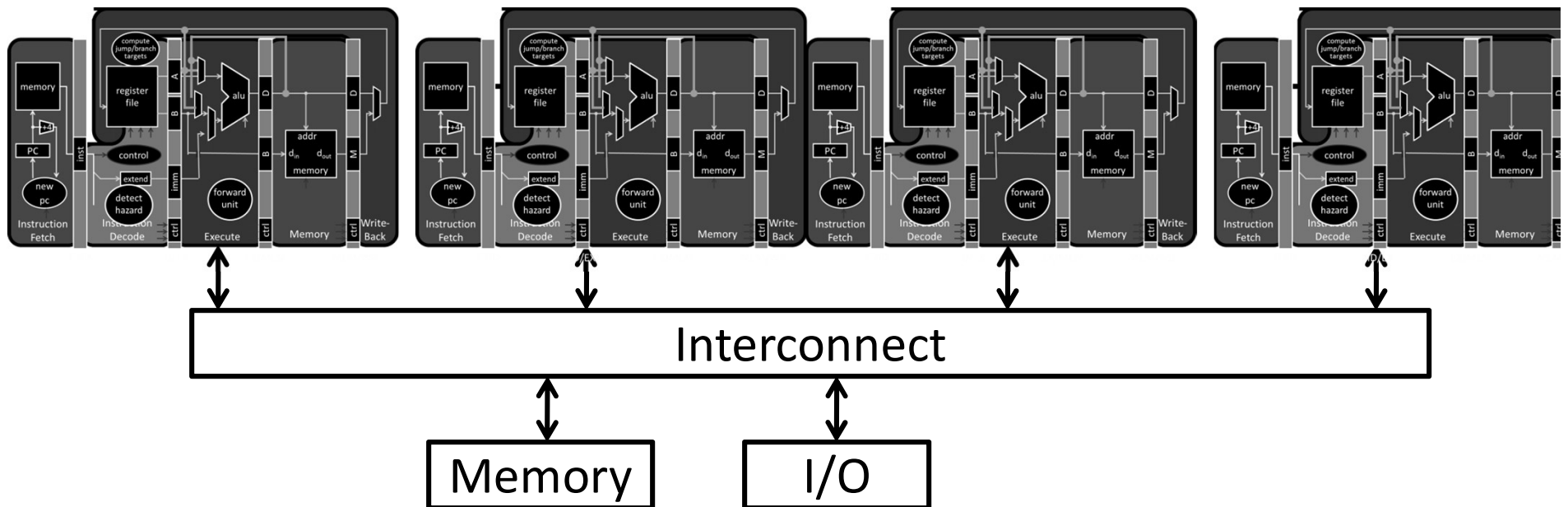


Each processor core has its own L1 cache



Parallelism and Synchronization

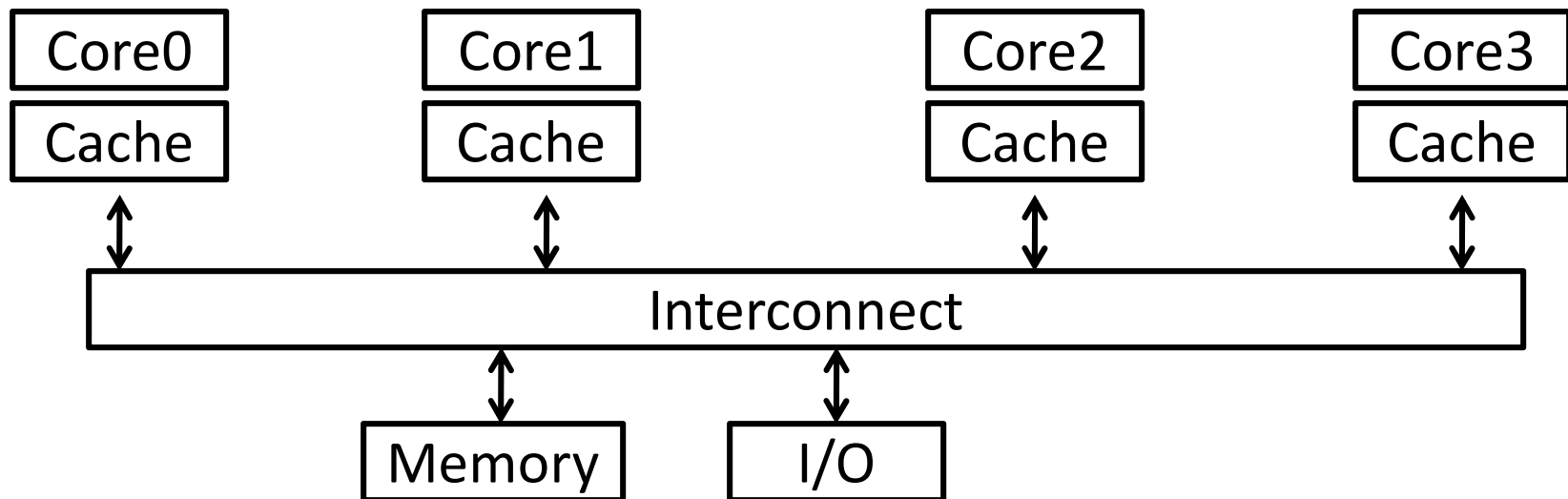
Each processor core has its own L1 cache



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

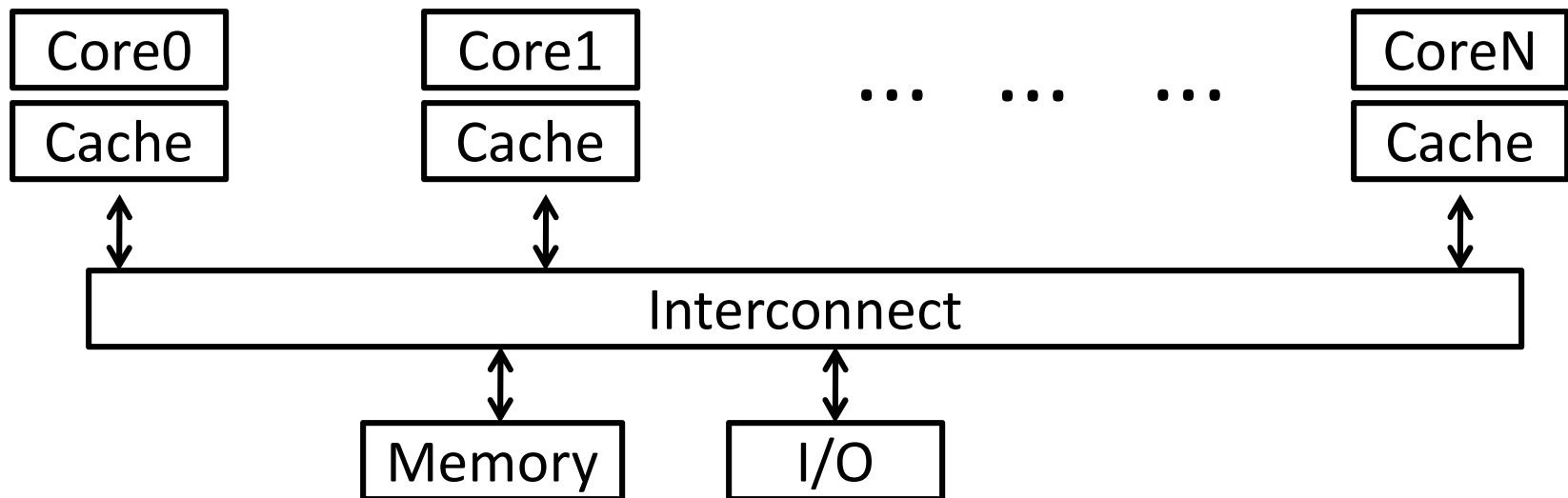
- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors



Cache Coherency Problem

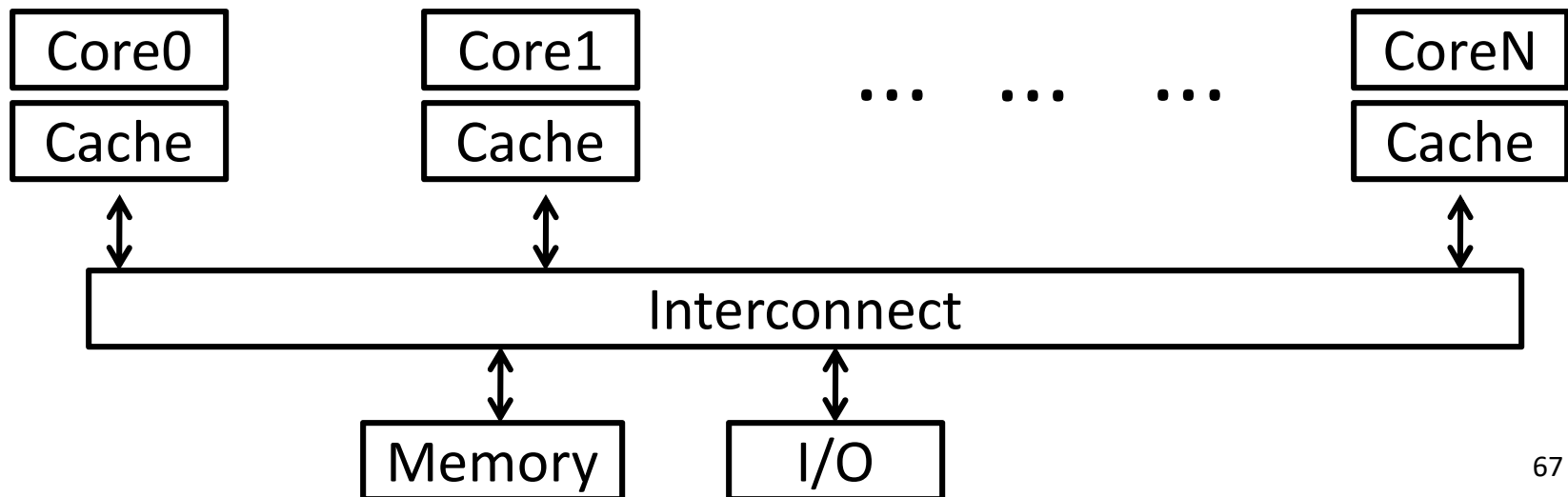
Thread A (on Core0)

```
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?



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?

(x starts as 0)

- a) 6
- b) 8
- c) 10
- d) Could be any of the above
- e) Couldn't be any of the above

Clicker Question

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?

(x starts as 0)

a) 6

b) 8

c) 10

d) Could be any of the above

e) Couldn't be any of the above

Clicker Question

Cache Coherency Problem, WB \$

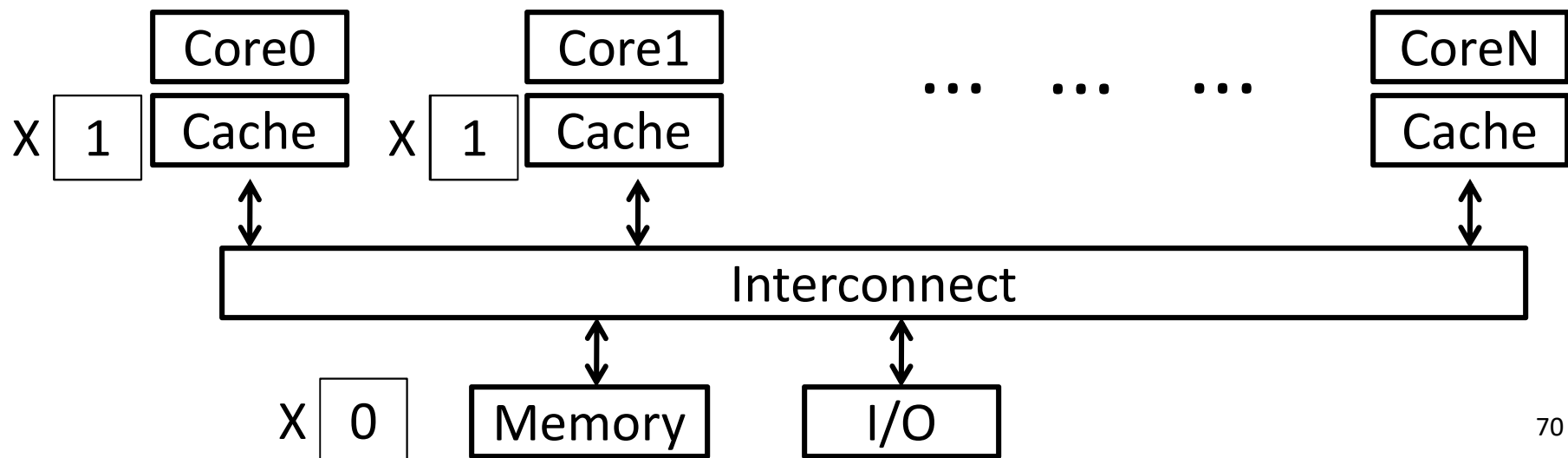
Thread A (on Core0)

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

Thread B (on Core1)

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

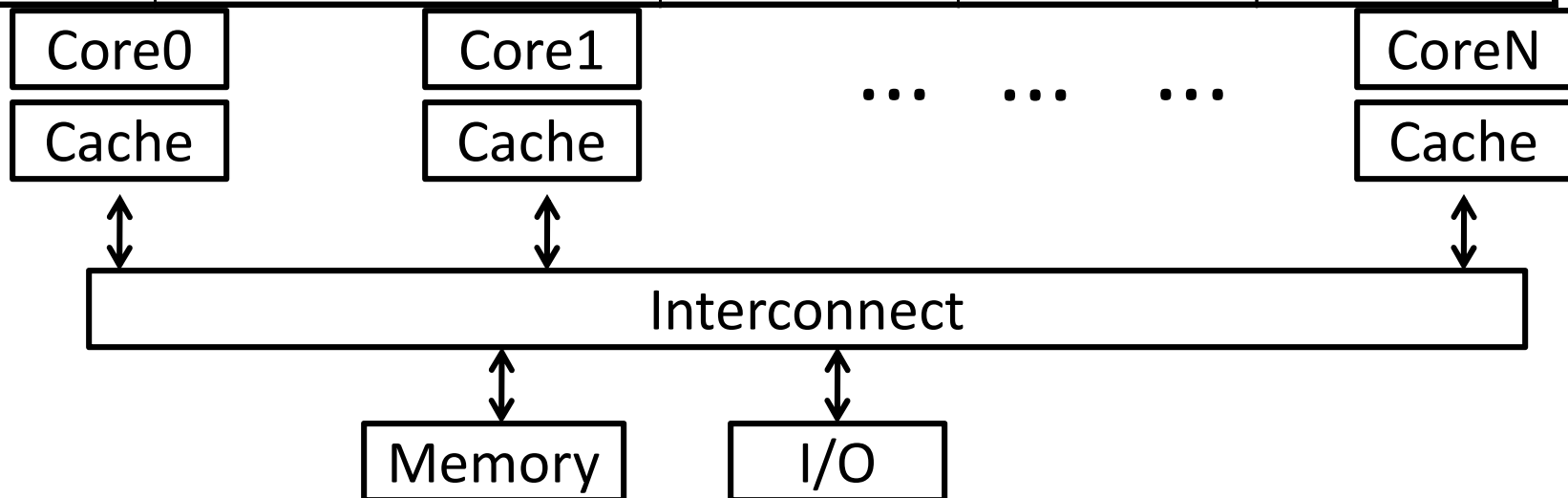
Problem!



Not just a problem for Write-Back Caches

Executing on a write-thru cache

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1



Two issues

Coherence

- What values can be returned by a read
- Need a globally uniform (consistent) view of a single memory location

Solution: Cache Coherence Protocols

Consistency

- When a written value will be returned by a read
- Need a globally uniform (consistent) view of *all memory locations relative to each other*

Solution: Memory Consistency Models

Coherence Defined

Informal: Reads return most recently written value

Formal: For concurrent processes P_1 and P_2

- P writes X before P reads X (with no intervening writes)
⇒ read returns written value
 - (preserve program order)
- P_1 writes X before P_2 reads X
⇒ read returns written value
 - (coherent memory view, can't read old value forever)
- P_1 writes X and P_2 writes X
⇒ all processors see writes in the same order
 - all see the same final value for X
 - Aka write serialization
 - (else P_A can see P_2 's write before P_1 's and P_B can see the opposite; their final understanding of state is wrong)

Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence

- Migration of data to local caches
 - Reduces bandwidth for shared memory
- Replication of read-shared data
 - Reduces contention for access

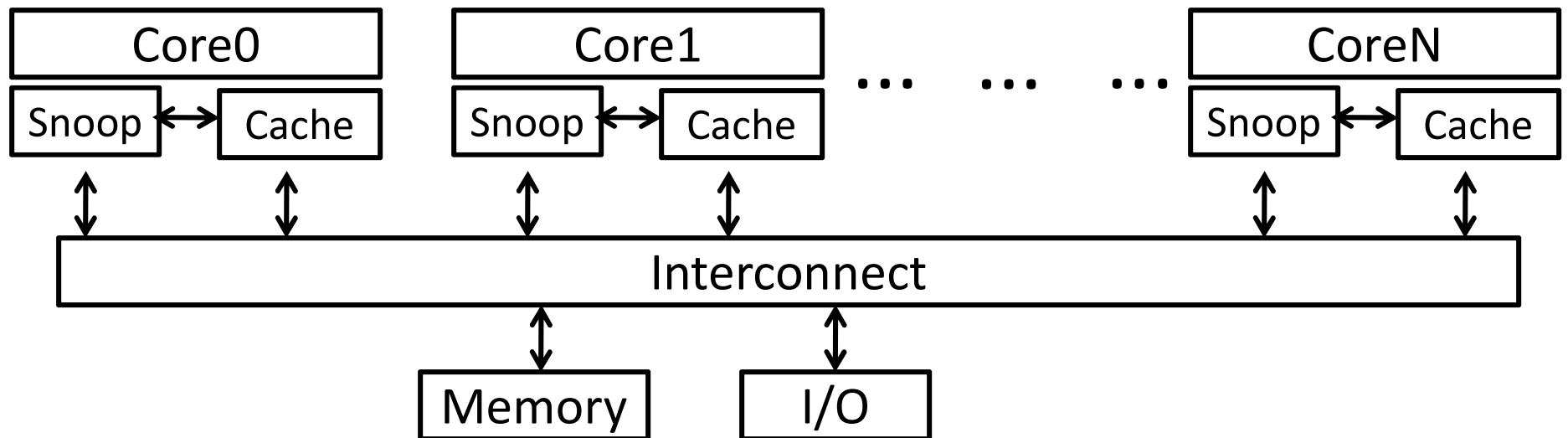
Snooping protocols

- Each cache monitors bus reads/writes

Snooping

Snooping for Hardware Cache Coherence





- All caches monitor bus and all other caches
- Bus read: respond if you have dirty data
- Bus write: update/invalidate your copy of data



Invalidating Snooping Protocols

Cache gets **exclusive access** to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read in another cache misses
 - Owning cache supplies updated value

Time Step	CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
0					0
1	CPU A reads X	Cache miss for X	0		0
2	CPU B reads X	Cache miss for X	0	0	0
3	CPU A writes 1 to X	Invalidate for X	1		0
4	CPU B read X	Cache miss for X			

Writing

Write-back policies for bandwidth

Write-invalidate coherence policy

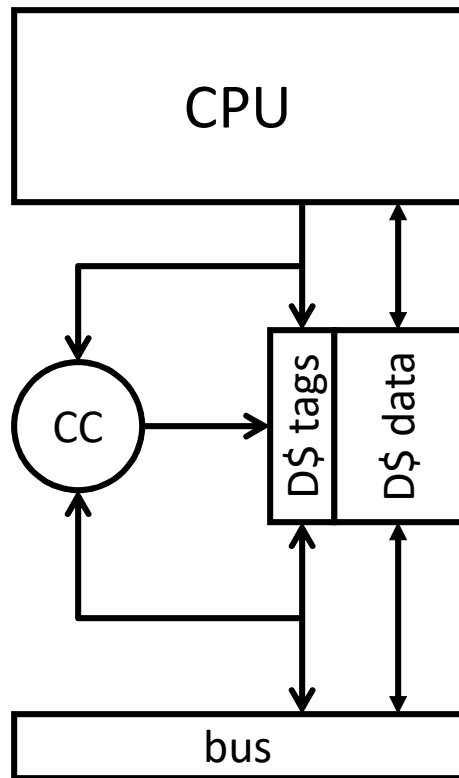
- First invalidate all other copies of data
- Then write it in cache line
- Anybody else can read it

Permits one writer, multiple readers

In reality: many coherence protocols

- Snooping doesn't scale
- Directory-based protocols
 - Caches and memory record sharing status of blocks in a directory

Hardware Cache Coherence



Coherence

- all copies have same data at all times

Coherence controller:

- Examines bus traffic (addresses and data)
- Executes **coherence protocol**
 - What to do with local copy when you see different things happening on bus

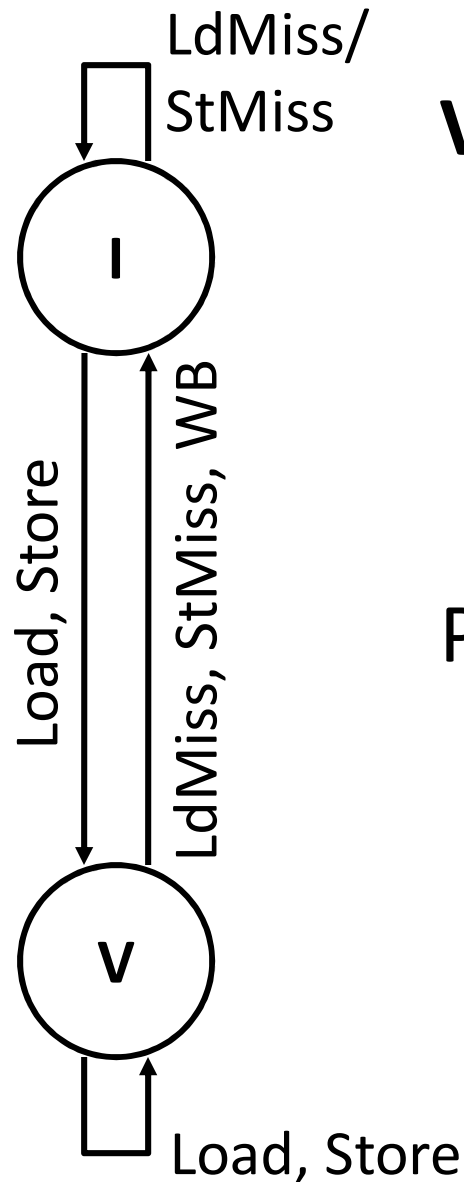
Three processor-initiated events

- **Ld**: load
- **St**: store
- **WB**: write-back

Two remote-initiated events

- **LdMiss**: read miss from ***another*** processor
- **StMiss**: write miss from ***another*** processor

VI Coherence Protocol



VI (valid-invalid) protocol:

- Two states (per block in cache)
 - **V (valid)**: have block
 - **I (invalid)**: don't have block
- + Can implement with valid bit

Protocol diagram (left)

- If *you* load/store a block: transition to **V**
- If anyone *else* wants to read/write block:
 - Give it up: transition to **I** state
 - Write-back if your own copy is dirty

VI Protocol (Write-Back Cache)

Thread A

```
lw t0, 0(r3),
ADDIU $t0,$t0,1
sw t0,0(r3)
```

Thread B

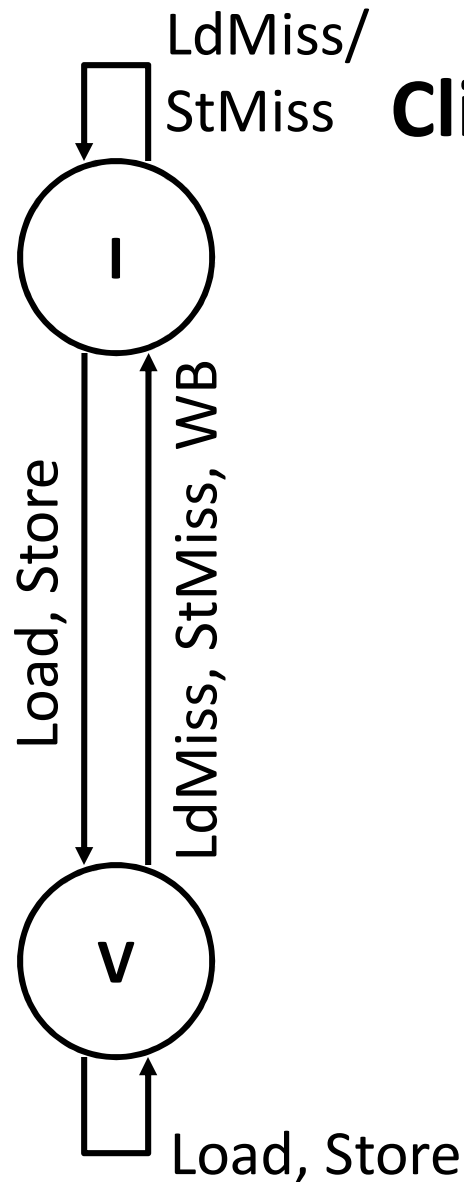
```
lw t0, 0(r3)
ADDIU $t0,$t0,1
sw t0,0(r3)
```

CPU0	CPU1	Mem
		0
V:0		0
V:1		0
I:	V:1	1
	V:2	1

lw by Thread B generates an “other load miss” event (LdMiss)

- Thread A responds by sending its dirty copy, transitioning to I

VI Coherence Question



Clicker Question:

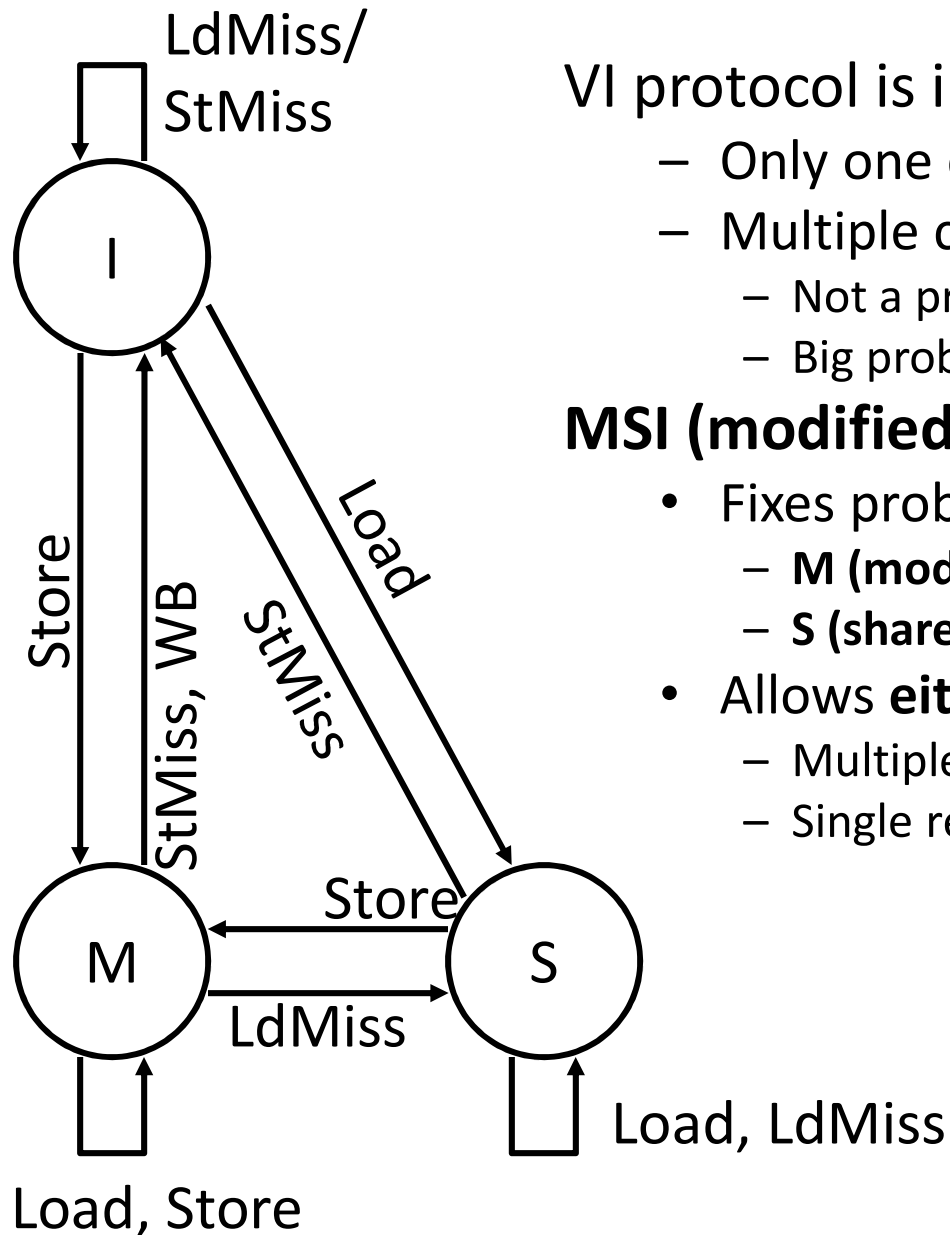
Core A loads x into a register

Core B wants to load x into a register

What happens?

- (A) they can both have a copy of X in their cache
- (B) A keeps the copy
- (C) B steals the copy from A, and this is an efficient thing to do
- (D) B steals the copy from A, and this is a sad shame
- (E) B waits until A kicks X out of its cache, then it can complete the load

VI → MSI



VI protocol is inefficient

- Only one cached copy allowed in entire system
- Multiple copies can't exist even if read-only
 - Not a problem in example
 - Big problem in reality

MSI (modified-shared-invalid)

- Fixes problem: splits “V” state into two states
 - **M (modified)**: local dirty copy
 - **S (shared)**: local clean copy
- Allows **either**
 - Multiple read-only copies (S-state) **--OR--**
 - Single read/write copy (M-state)

MSI Protocol (Write-Back Cache)

Thread A

```
lw t0, 0(r3),
ADDIU $t0,$t0,1
sw t0,0(r3)
```

Thread B

```
lw t0, 0(r3),
ADDIU $t0,$t0,1
sw t0,0(r3)
```

CPU0	CPU1	Mem
		0
S:0		0

M:1		0
-----	--	---

S:1	S:1	1
-----	-----	---

I:	M:2	1
----	-----	---

lw by Thread B generates a “other load miss” event (LdMiss)

- Thread A responds by sending its dirty copy, transitioning to **S**

sw by Thread B generates a “other store miss” event (StMiss)

- Thread A responds by transitioning to **I**

Cache Coherence and Cache Misses

Coherence introduces two new kinds of cache misses

- **Upgrade miss**
 - On stores to read-only blocks
 - Delay to acquire write permission to read-only block
- **Coherence miss**
 - Miss to a block evicted by another processor's requests

Making the cache larger...

- Doesn't reduce these type of misses
- As cache grows large, these sorts of misses dominate

False sharing

- Two or more processors sharing parts of the same block
- But *not* the same bytes within that block (no actual sharing)
- Creates pathological “ping-pong” behavior
- Careful data placement may help, but is difficult

More Cache Coherence

In reality: many coherence protocols

- Snooping: VI, MSI, MESI, MOESI, ...
 - But Snooping doesn't scale
- Directory-based protocols
 - Caches & memory record blocks' sharing status in directory
 - Nothing is free → directory protocols are slower!

Cache Coherency:

- requires that reads return most recently written value
- Is a hard problem!

Takeaway: Summary of cache coherence

Informally, Cache Coherency requires that reads return most recently written value

Cache coherence hard problem

Snooping protocols are one approach

Next Goal: Synchronization

Is cache coherency sufficient?

i.e. Is cache coherency (***what*** values are read) sufficient to maintain consistency (***when*** a written value will be returned to a read). Both coherency and consistency are required to maintain consistency in shared memory programs.

Are We Done Yet?

Thread A

lw t0, 0(r3)

ADDIU \$t0, \$t0, 1

sw t0, 0(x)

Thread B

lw t0, 0(r3)

ADDIU \$t0, \$t0, 1

sw t0, 0(x)

CPU0	CPU1	Mem
------	------	-----

		0
S:0		0

S:0	S:0	0
-----	-----	---

I:	M:1	0
----	-----	---

M:1	I:	1
-----	----	---

What just happened???

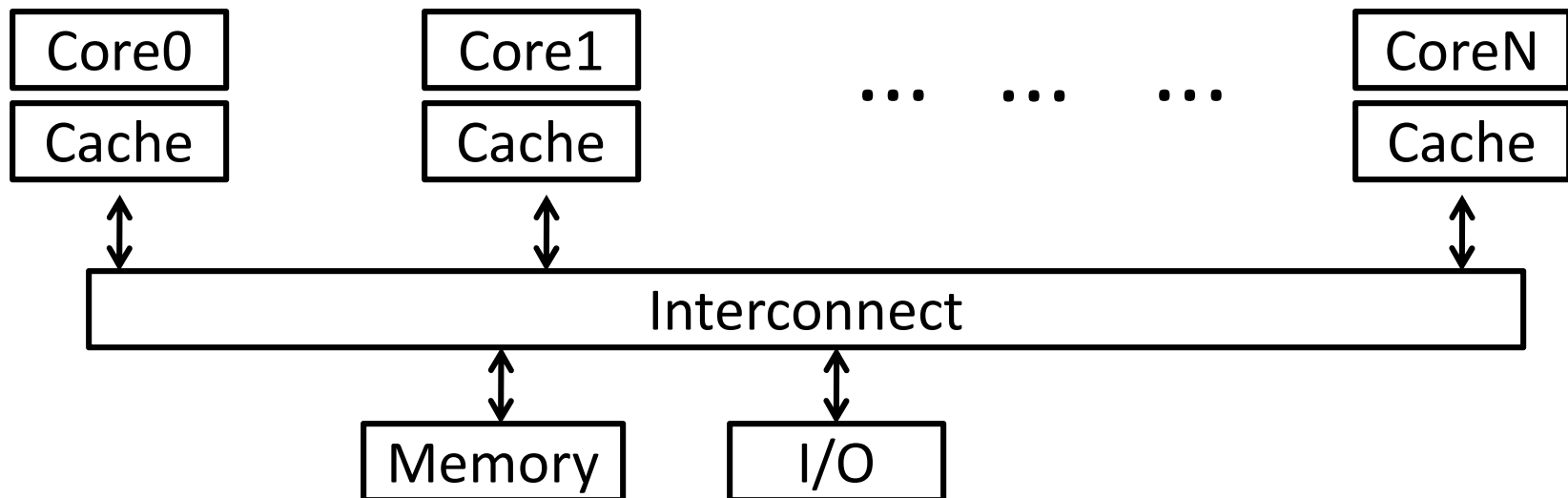
Is Cache Coherency Protocol Broken??

Is Cache Coherency Sufficient?

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)
}

} Very expensive and difficult to maintain consistency }



Clicker Question

The Previous example shows us that

- a) Caches can be incoherent even if there is a coherence protocol.
- b) Cache coherence protocols are not rich enough to support multi-threaded programs
- c) Coherent caches are not enough to guarantee expected program behavior.
- d) Multithreading is just a really bad idea.
- e) All of the above

Clicker Question

The Previous example shows us that

- a) Caches can be incoherent even if there is a coherence protocol.
- b) Cache coherence protocols are not rich enough to support multi-threaded programs
- c) Coherent caches are not enough to guarantee expected program behavior.
- d) Multithreading is just a really bad idea.
- e) All of the above

Programming with Threads

Need it to exploit multiple processing units

...to parallelize for multicore

...to write servers that handle many clients

Problem: hard even for experienced programmers

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Needed: synchronization of threads

Programming with threads

Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

Problem: hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency is **not** sufficient...

Need explicit synchronization to make sense of concurrency!

Programming with Threads

Concurrency poses challenges for:

Correctness

- Threads accessing shared memory should not interfere with each other

Liveness

- Threads should not get stuck, should make forward progress

Efficiency

- Program should make good use of available computing resources (e.g., processors).

Fairness

- Resources apportioned fairly between threads

Example: Multi-Threaded Program

Apache web server

```
void main() {  
    setup();  
    while (c = accept_connection()) {  
  
        req = read_request(c);  
        hits[req]++;  
        send_response(c, req);  
  
    }  
    cleanup();  
}
```

Example: web server

Each client request handled by a separate thread
(in parallel)

- Some shared state: hit counter, ...

```
Thread 52  
read hits  
addiu  
write hits
```

```
Thread 205  
read hits  
addiu  
write hits
```

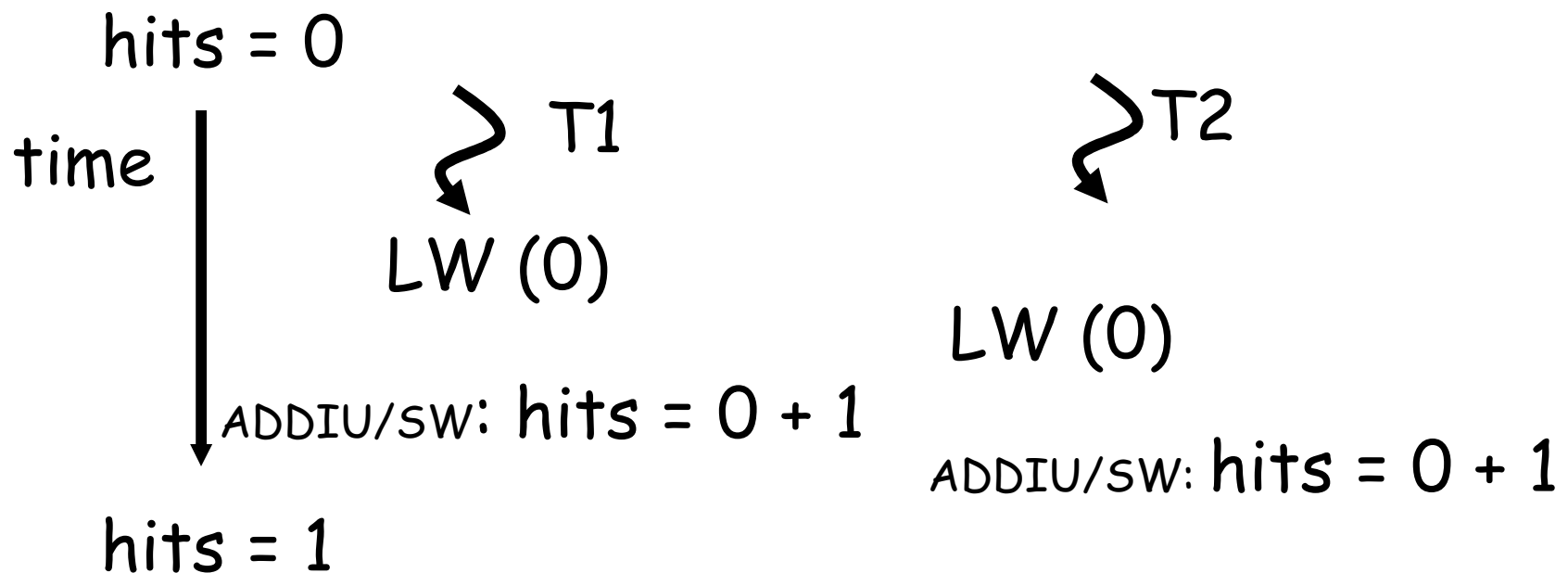
(look familiar?)

Timing-dependent failure \Rightarrow race condition

- hard to reproduce \Rightarrow hard to debug

Two threads, one counter

Possible result: lost update!



Timing-dependent failure \Rightarrow race condition

- Very hard to reproduce \Rightarrow Difficult to debug

Race conditions

Timing-dependent error involving access to shared state

Race conditions depend on how threads are scheduled

- i.e. who wins “races” to update state

Challenges of Race Conditions

- Races are intermittent, may occur rarely
- Timing dependent = small changes can hide bug

Program is correct *only* if *all possible* schedules are safe

- Number of possible schedules is huge
- Imagine adversary who switches contexts at worst possible time

Critical Sections

What if we can designate parts of the execution as critical sections

- Rule: only one thread can be “inside” a critical section

Thread 52

```
CSEnter()  
read hits  
addi  
write hits  
CSExit()
```

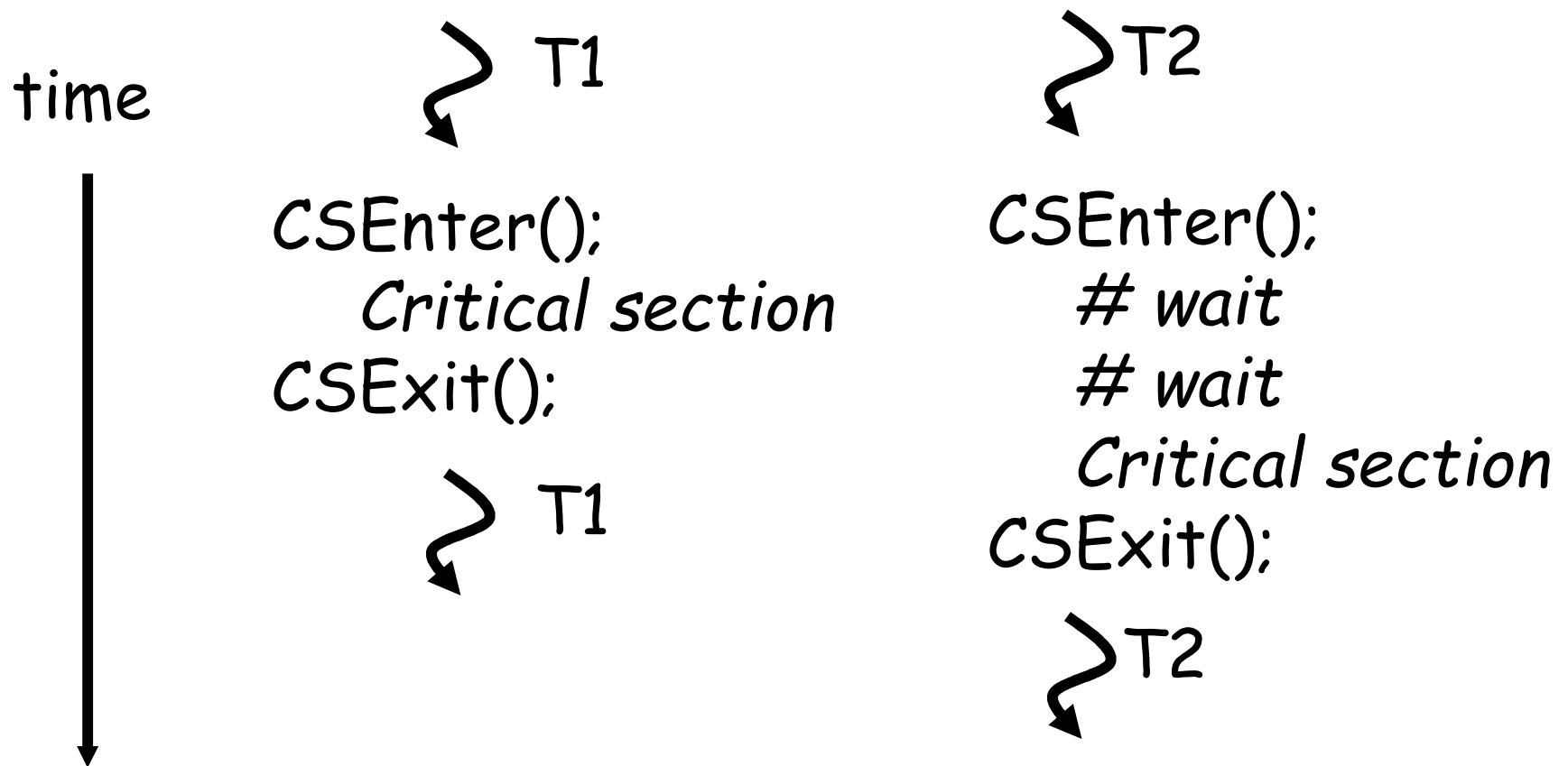
Thread 205

```
CSEnter()  
read hits  
addi  
write hits  
CSExit()
```

Critical Sections

To eliminate races: use *critical sections* that only one thread can be in

- Contending threads must wait to enter



Mutual Exclusion Lock (Mutex)

Implement CSEnter and CSExit; ie. a critical section

- Only one thread can hold the lock at a time
“I have the lock”

Mutual Exclusion Lock (mutex)

lock(m): wait till it becomes free, then lock it

unlock(m): unlock it

```
safe_increment() {  
    pthread_mutex_lock(&m);  
    hits = hits + 1;  
    pthread_mutex_unlock(&m);  
}
```

Mutexes

Only one thread can hold a given mutex at a time

Acquire (lock) mutex on entry to critical section

- Or block if another thread already holds it

Release (unlock) mutex on exit

- Allow **one** waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(&m);  
pthread_mutex_lock(&m);  
pthread_mutex_lock(&m);    # wait  
    hits = hits+1;          # wait  
pthread_mutex_unlock(&m);    hits = hits+1;  
                                pthread_mutex_unlock(&m);  
    ↪ T1                                ↪ T2
```

Next Goal

How to implement mutex locks?

What are the hardware primitives?

Then, use these mutex locks to implement critical sections, and use critical sections to write parallel safe programs

Hardware Support for Synchronization

Atomic read & write memory operation

- Between read & write: *no writes to that address*

Many atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional (pair of insns)
(MIPS, ARM, PowerPC, DEC Alpha, ...)

Synchronization in MIPS

Load linked: `LL rt, offset(rs)`

“I want the value at address X. Also, start monitoring any writes to this address.”

Store conditional: `SC rt, offset(rs)`

“If no one has changed the value at address X since the LL, perform this store and tell me it worked.”

- Data at location has not changed since the LL?
 - SUCCESS:
 - Performs the store
 - Returns 1 in rt
- Data at location has changed since the LL?
 - FAILURE:
 - Does not perform the store
 - Returns 0 in rt

Synchronization in MIPS

Load linked: LL rt, offset(rs)

Store conditional: SC rt, offset(rs)

- Succeeds if location not changed since the LL
 - Returns 1 in rt
- Fails if location is changed
 - Returns 0 in rt

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in \$t0, causing the code to try again.

i.e. use this value 0 in \$t0 to try again.

Synchronization in MIPS

Load linked: LL rt, offset(rs)

Store conditional: SC rt, offset(rs)

- Succeeds if location not changed since the LL
 - Returns 1 in rt
- Fails if location is changed
 - Returns 0 in rt

Example: atomic incrementor

i++



LW \$t0, 0(\$s0)

ADDIU \$t0, \$t0, 1 →

SW \$t0, 0(\$s0)

atomic(i++)



try: LL \$t0, 0(\$s0)

ADDIU \$t0, \$t0, 1

SC \$t0, 0(\$s0)

BEQZ \$t0, try

Value in memory changed between **LL** and **SC** ?

→ **SC** returns **0** in **\$t0** → **retry**

Atomic Increment in Action

Load linked: LL \$t0, offset(\$s0)

Store conditional: SC \$t0, offset(\$s0)

Time	Thread A	Thread B	Thread A \$t0	Thread B \$t0	Mem [\$s0]
0					0
1	try: LL \$t0, 0(\$s0)				
2		try: LL \$t0, 0(\$s0)			
3	ADDIU \$t0, \$t0, 1				
4		ADDIU \$t0, \$t0, 1			
5	SC \$t0, 0(\$s0)				
6	BEQZ \$t0, try				
7		SC \$t0, 0 (\$s0)			
8		BEQZ \$t0, try			

Atomic Increment in Action

Load linked: LL \$t0, offset(\$s0)

Store conditional: SC \$t0, offset(\$s0)

Time	Thread A	Thread B	Thread A \$t0	Thread B \$t0	Mem [\$s0]
0					0
1	try: LL \$t0, 0(\$s0)		0		0
2		try: LL \$t0, 0(\$s0)		0	0
3	ADDIU \$t0, \$t0, 1		1	0	0
4		ADDIU \$t0, \$t0, 1	1	1	0
5	SC \$t0, 0(\$s0)		1	1	1
6	BEQZ \$t0, try		1	1	1
7		SC \$t0, 0 (\$s0)	1	0	1
8		BEQZ \$t0, try	1	0	1

Success!

Failure!

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

```
mutex_lock(int *m) {  
    while(test_and_set(m)){  
    }  
}
```

```
int test_and_set(int *m) {  
    {  
        old = *m;  
        *m = 1;  
    } LL SC Atomic  
    return old;  
}
```

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

```
mutex_lock(int *m) {  
    while(test_and_set(m)){  
    }  
}
```

```
int test_and_set(int *m) {  
try: → LI $t0, 1  
      LL $t1, 0($a0)  
      SC $t0, 0($a0) ← BEQZ $t0, try  
      MOVE $v0, $t1  
}
```

Mutex from LL and SC

Linked load / Store Conditional

`m = 0;` // `m=0` means lock is free; otherwise, if `m=1`, then lock locked

```
mutex_lock(int *m) {  
    while(test_and_set(m)){  
    }  
}
```

```
int test_and_set(int *m) {  
    try:  
        LI $t0, 1  
        LL $t1, 0($a0)  
        SC $t0, 0($a0)  
        BEQZ $t0, try  
        MOVE $v0, $t1
```

Mutex from LL and SC

Linked load / Store Conditional

`m = 0;` // `m=0` means lock is free; otherwise, if `m=1`, then lock locked

```
mutex_lock(int *m) {  
    test_and_set:  
        LI $t0, 1  
        LL $t1, 0($a0)  
        BNEZ $t1, test_and_set  
        SC $t0, 0($a0)  
        BEQZ $t0, test_and_set  
}
```

```
mutex_unlock(int *m) {  
    *m = 0;  
}
```

Mutex from LL and SC

Linked load / Store Conditional

`m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked`

```
mutex_lock(int *m) {
```

```
    test_and_set:
```

```
        LI $t0, 1
```

```
        LL $t1, 0($a0)
```

```
        BNEZ $t1, test_and_set
```

```
        SC $t0, 0($a0)
```

```
        BEQZ $t0, test_and_set
```

```
}
```

```
mutex_unlock(int *m) {
```

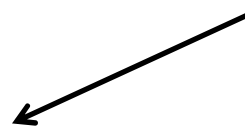
```
    SW $zero, 0($a0)
```

```
}
```

This is called a

Spin lock

Aka spin waiting



Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

mutex_lock(int *m) {

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							0
1	try: LI \$t0, 1	try: LI \$t0, 1					
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)					
3	BNEZ \$t1, try	BNEZ \$t1, try					
4		SC \$t0, 0 (\$a0)					
5	SC \$t0, 0(\$a0)						
6	BEQZ \$t0, try	BEQZ \$t0, try					
7							

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

mutex_lock(int *m) {

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							0
1	try: LI \$t0, 1	try: LI \$t0, 1	1		1		0
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	0	1	0	0
3	BNEZ \$t1, try	BNEZ \$t1, try	1	0	1	0	0
4		SC \$t0, 0 (\$a0)	0	0	1	0	1
5	SC \$t0, 0(\$a0)		0	0	1	0	1
6	BEQZ \$t0, try	BEQZ \$t0, try	0	0	1	0	1
7							

Mutex from LL and SC

Linked load / Store Conditional

$m = 0$; // $m=0$ means lock is free; otherwise, if $m=1$, then lock locked

`mutex_lock(int *m) {`

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							0
1	try: LI \$t0, 1	try: LI \$t0, 1	1		1		0
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	0	1	0	0
3	BNEZ \$t1, try	BNEZ \$t1, try	1	0	1	0	0
4		SC \$t0, 0 (\$a0)	0	0	1	0	1
5	SC \$t0, 0(\$a0)		0	0	1	0	1
6	BEQZ \$t0, try	BEQZ \$t0, try	0	0	1	0	1
	try: LI \$t0, 1	Critical section					

Failed to get mutex lock – try again

Success grabbing mutex lock!
Inside Critical section

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

```
mutex_lock(int *m) {
```

```
    test_and_set:
```

```
        LI $t0, 1
```

```
        LL $t1, 0($a0)
```

```
        BNEZ $t1, test_and_set
```

```
        SC $t0, 0($a0)
```

```
        BEQZ $t0, test_and_set
```


```
}
```

```
mutex_unlock(int *m) {
```

```
    SW $zero, 0($a0)
```

```
}
```

This is called a
Spin lock
Aka spin waiting



Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

mutex_lock(int *m) {

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							1
1	try: LI \$t0, 1	try: LI \$t0, 1					
2							
3							
4							
5							
6							
7							
8							
9							

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

mutex_lock(int *m) {

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							1
1	try: LI \$t0, 1	try: LI \$t0, 1	1		1		1
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	1	1	1	1
3	BNEZ \$t1, try	BNEZ \$t1, try	1	1	1	1	1
4	try: LI \$t0, 1	try: LI \$t0, 1	1	1	1	1	1
5	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	1	1	1	1
6	BNEZ \$t1, try	BNEZ \$t1, try	1	1	1	1	1
7	try: LI \$t0, 1	try: LI \$t0, 1	1	1	1	1	1
8	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	1	1	1	1
9	BNEZ \$t1, try	BNEZ \$t1, try	1	1	1	1	1

Now we can write parallel and correct programs

Thread A

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

```
    mutex_lock(m);
```

```
        x = x + 1;
```

```
    mutex_unlock(m);
```

```
}
```

Thread B

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

```
    mutex_lock(m);
```

```
        x = x + 1;
```

```
    mutex_unlock(m);
```

```
}
```

Alternative Atomic Instructions

Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional

(MIPS, ARM, PowerPC, DEC Alpha, ...)

Synchronization

Synchronization techniques

clever code

- must work despite adversarial scheduler/interrupts
- used by: hackers
- also: noobs

disable interrupts

- used by: exception handler, scheduler, device drivers, ...

disable preemption

- dangerous for user code, but okay for some kernel code

mutual exclusion locks (mutex)

- general purpose, except for some interrupt-related cases

Summary

Need parallel abstractions, especially for multicore

Writing correct programs is hard

Need to prevent data races

Need critical sections to prevent data races

Mutex, mutual exclusion, implements critical section

Mutex often implemented using a lock abstraction

Hardware provides synchronization primitives such as **LL** and **SC** (load linked and store conditional) instructions to efficiently implement locks

Next Goal

How do we use synchronization primitives to build concurrency-safe data structure?

Producer/Consumer Example (1)

Access to shared data must be synchronized

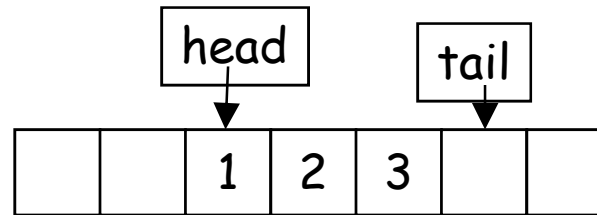
- Goal: enforce datastructure **invariants**

```
// invariant:
```

```
// data is in A[h ... t-1]
```

```
char A[100];
```

```
int h = 0, t = 0;
```



```
// producer: add to list tail
```

```
void put(char c) {
```

```
    A[t] = c;
```

```
    t = (t+1)%n;
```

```
}
```

Producer/Consumer Example (1)

Access to shared data must be synchronized

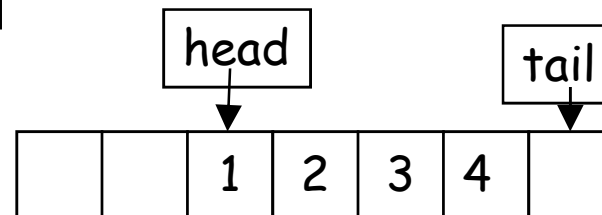
- Goal: enforce datastructure **invariants**

```
// invariant:
```

```
// data is in A[h ... t-1]
```

```
char A[100];
```

```
int h = 0, t = 0;
```



```
// producer: add to list tail // consumer: take from list head
```

```
void put(char c) {
```

```
    A[t] = c;
```

```
    t = (t+1)%n;
```

```
}
```

```
char get() {
```

```
    while (h == t) { };
```

```
    char c = A[h];
```

```
    h = (h+1)%n;
```

```
    return c;
```

```
}
```

Producer/Consumer Example (1)

Access to shared data must be synchronized

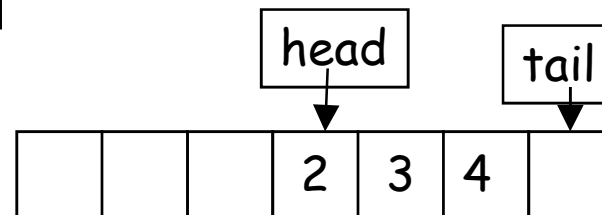
- Goal: enforce datastructure **invariants**

```
// invariant:
```

```
// data is in A[h ... t-1]
```

```
char A[100];
```

```
int h = 0, t = 0;
```



```
// producer: add to list tail // consumer: take from list head
```

```
void put(char c) {
```

```
    A[t] = c;
```

```
    t = (t+1)%n;
```

```
}What is wrong with code?
```

```
char get() {
```

```
    while (h == t) { };
```

```
    char c = A[h];
```

```
    h = (h+1)%n;
```

```
    return c;
```

```
}
```

- Will lose update to **t** and/or **h**
- Invariant is not upheld
- Will produce if full
- Will consume if empty
- All of the above

Producer/Consumer Example (1)

Access to shared data must be synchronized

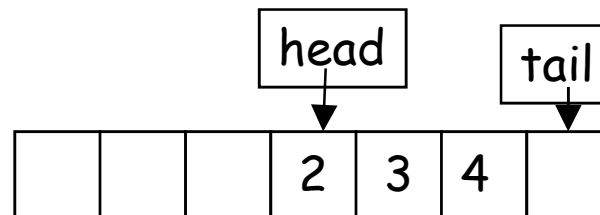
- Goal: enforce datastructure **invariants**

```
// invariant:
```

```
// data is in A[h ... t-1]
```

```
char A[100];
```

```
int h = 0, t = 0;
```



```
// producer: add to list tail // consumer: take from list head
```

```
void put(char c) {
```

```
    A[t] = c;
```

```
    t = (t+1)%n; ←
```

```
}
```

```
char get() {
```

```
    → while (h == t) { };
```

```
    char c = A[h];
```

```
    → h = (h+1)%n;
```

```
    return c;
```

```
}
```

What is wrong with the code? }

Could miss an update to t or h

Breaks invariant: only produce if not full and only consume if not empty

→ *Need to synchronize access to shared data*

Producer/Consumer Example (1)

Access to shared data must be synchronized

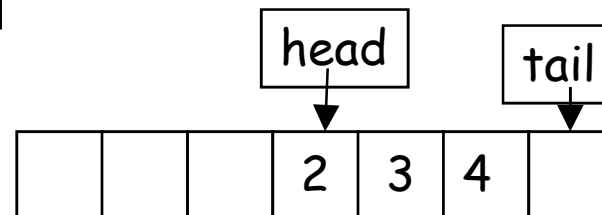
- Goal: enforce datastructure **invariants**

// invariant:

// data is in $A[h \dots t-1]$

char A[100];

int h = 0, t = 0;



// producer: add to list tail // consumer: take from list head

```
void put(char c) {  
    A[t] = c; ← acquire-lock()  
    t = (t+1)%n;  
    ← acquire-lock()  
}
```

```
char get() {  
    ← acquire-lock()  
    while (h == t) { };  
    char c = A[h];  
    h = (h+1)%n;  
    return c; ← acquire-lock()  
}
```

Rule of thumb: all access and updates that can affect invariant become critical sections

Protecting an invariant Example (2)

// invariant: (protected by mutex *m*)

// data is in A[h ... t-1]

pthread_mutex_t *m = pthread_mutex_create();

char A[100];

int h = 0, t = 0;

Does this fix
work?

```
// producer: add to list tail // consumer: take from list head
void put(char c) {                char get() {
    pthread_mutex_lock(m);          pthread_mutex_lock(m);
    A[t] = c;                       while(h == t) {}
    t = (t+1)%n;                   char c = A[h];
    pthread_mutex_unlock(m);        h = (h+1)%n;
}                                   pthread_mutex_unlock(m);
                                   return c;
}
```

Rule of thumb: all access and updates that can affect invariant become critical sections

Protecting an invariant Example (2)

// invariant: (protected by mutex *m*)

// data is in A[h ... t-1]

pthread_mutex_t *m = pthread_mutex_create();

char A[100];

int h = 0, t = 0;

Does this fix
work?

BUG: Can't wait while holding lock

// producer: add to list tail // consumer: take from list head

void put(char c) {

pthread_mutex_lock(m);

A[t] = c;

t = (t+1)%n;

pthread_mutex_unlock(m);

}

char get() {

pthread_mutex_lock(m);

while(h == t) {}

char c = A[h];

h = (h+1)%n;

pthread_mutex_unlock(m);

return c;

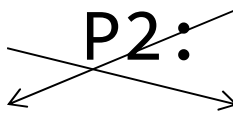
Rule of thumb: all access and updates that can affect invariant become critical sections

Guidelines for successful mutexing

Insufficient locking can cause races

- Skimping on mutexes? Just say no!

Poorly designed locking can cause deadlock

P1: lock(m1);		P2: lock(m2);	Circular Wait
lock(m2);		lock(m1);	

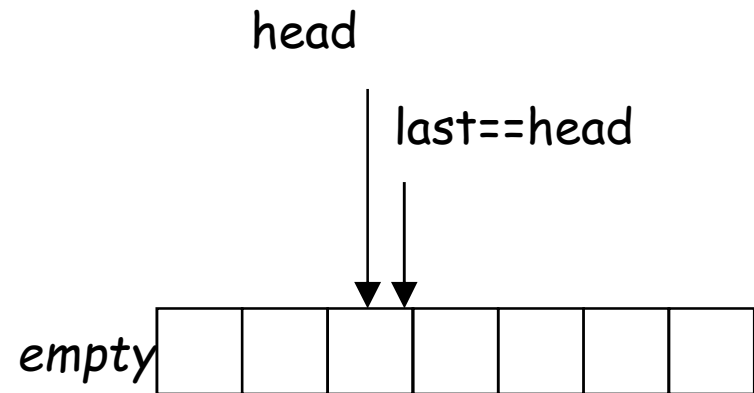
- know why you are using mutexes!
- acquire locks in a consistent order to avoid cycles
- use lock/unlock like braces (match them lexically)
 - lock(&m); ...; unlock(&m)
 - watch out for return, goto, and function calls!
 - watch out for exception/error conditions!

Beyond mutexes Example (3)

Writers must check for full buffer
& Readers must check for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    while(empty) {}  
    acquire(L);  
    char c = A[h];  
    h = (h+1)%n;  
    release(L);  
    return c;  
}
```



Beyond mutexes Example (3)

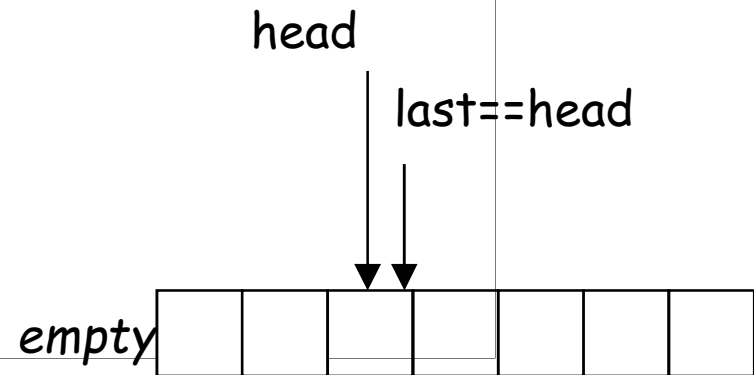
Writers must check for full buffer

& Readers must check for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    while (h == t) { };  
    acquire(L);  
    char c = A[h];  
    h = (h+1)%n;  
    release(L);  
    return c;  
}
```

Cannot check condition while
Holding the lock,
BUT, empty condition may no
longer hold in critical section



Dilemma: Have to check while holding lock,

Beyond mutexes Example (3)

Writers must check for full buffer

& Readers must check for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    acquire(L);  
    while (h == t) { };  
    char c = A[h];  
    h = (h+1)%n;  
    release(L);  
    return c;  
}
```

Dilemma: Have to check while holding lock,
but cannot wait while hold lock

Beyond mutexes Example (4)

Writers must check for full buffer

& Readers must check for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    do {  
        acquire(L);  
        empty = (h == t);  
        if (!empty) {  
            c = A[h];  
            h = (h+1)%n;  
        }  
        release(L);  
    } while (empty);  
    return c;  
}
```

Does this work?

a) Yes

b) no

Beyond mutexes Example (4)

Writers must check for full buffer

& Readers must check for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    do {  
        acquire(L);  
        empty = (h == t);  
        if (!empty) {  
            c = A[h];  
            h = (h+1)%n;  
        }  
        release(L);  
    } while (empty);  
    return c;  
}
```

It works.

But, it is wasteful

Due to the spinning

Language-level Synchronization

Condition variables

Use [Hoare] a condition variable to wait for a condition to become true (without holding lock!)

`wait(m, c) :`

- atomically release *m* and sleep, waiting for condition *c*
- wake up holding *m* sometime after *c* was signaled

`signal(c) :` wake up one thread waiting on *c*

`broadcast(c) :` wake up all threads waiting on *c*

POSIX (e.g., Linux): `pthread_cond_wait`,
`pthread_cond_signal`, `pthread_cond_broadcast`

Using a condition variable Example (5)

`wait(m, c)` : release m, sleep until c, wake up holding m

`signal(c)` : wake up one thread waiting on c

```
cond_t *not_full = ...;
cond_t *not_empty = ...;
mutex_t *m = ...;
```

```
void put(char c) {
    lock(m);
    while ((t-h) % n == 1)
        wait(m, not_full);
    A[t] = c;
    t = (t+1) % n;
    unlock(m);
    signal(not_empty);
}
```

```
char get() {
    lock(m);
    while (t == h)
        wait(m, not_empty);
    char c = A[h];
    h = (h+1) % n;
    unlock(m);
    signal(not_full);
    return c;
}
```

Monitors

A Monitor is a concurrency-safe datastructure, with...

- one mutex
- some condition variables
- some operations

All operations on monitor acquire/release mutex

- one thread in the monitor at a time

Ring buffer was a monitor

Java, C#, etc., have built-in support for monitors

Java concurrency

Java objects can be monitors

- “synchronized” keyword locks/releases the mutex
- Has one (!) builtin condition variable
 - `o.wait()` = `wait(o, o)`
 - `o.notify()` = `signal(o)`
 - `o.notifyAll()` = `broadcast(o)`
- Java `wait()` can be called even when mutex is not held. Mutex not held when awoken by `signal()`.
Useful?

Language-level Synchronization

Lots of synchronization variations...

Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

Semaphores

- N threads can hold lock at the same time

Monitors

- Concurrency-safe data structure with 1 mutex
- All operations on monitor acquire/release mutex
- One thread in the monitor at a time

Message-passing, sockets, queues, ring buffers, ...

- transfer data and synchronize

Summary

Hardware Primitives: test-and-set, LL/SC, barrier, ...
... used to build ...

Synchronization primitives: mutex, semaphore, ...
... used to build ...

Language Constructs: monitors, signals, ...