Computer Architecture

Week 14: Parallelism, Multi Core, Synchronization

Fenerbahçe University

Professor & TAs

Prof: Dr. Vecdi Emre Levent

Office: 311

Email: emre.levent@fbu.edu.tr

TA: Arş. Gör. Uğur Özbalkan

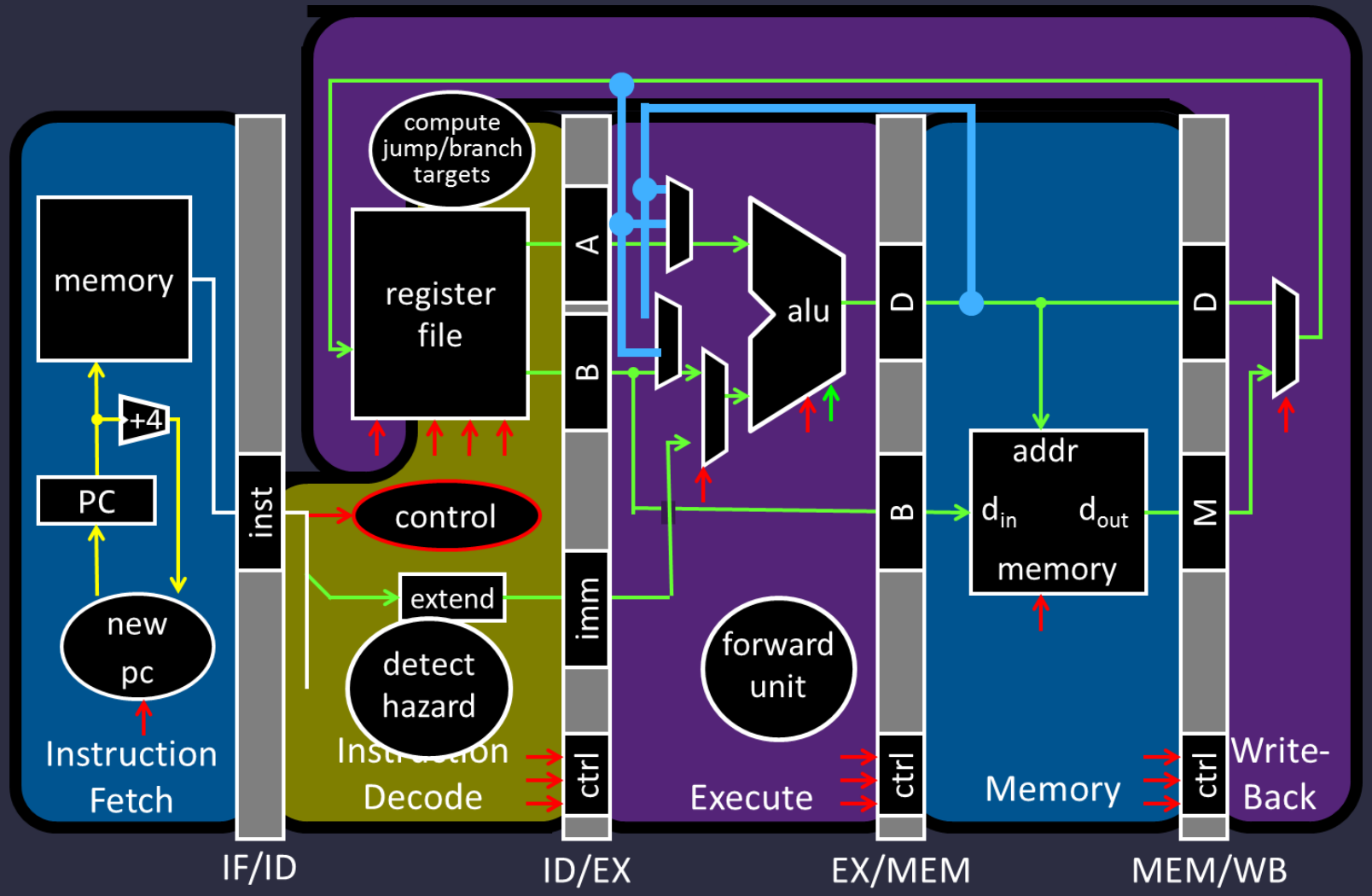
Office: 311

Email: ugur.ozbalkan@fbu.edu.tr

Course Plan

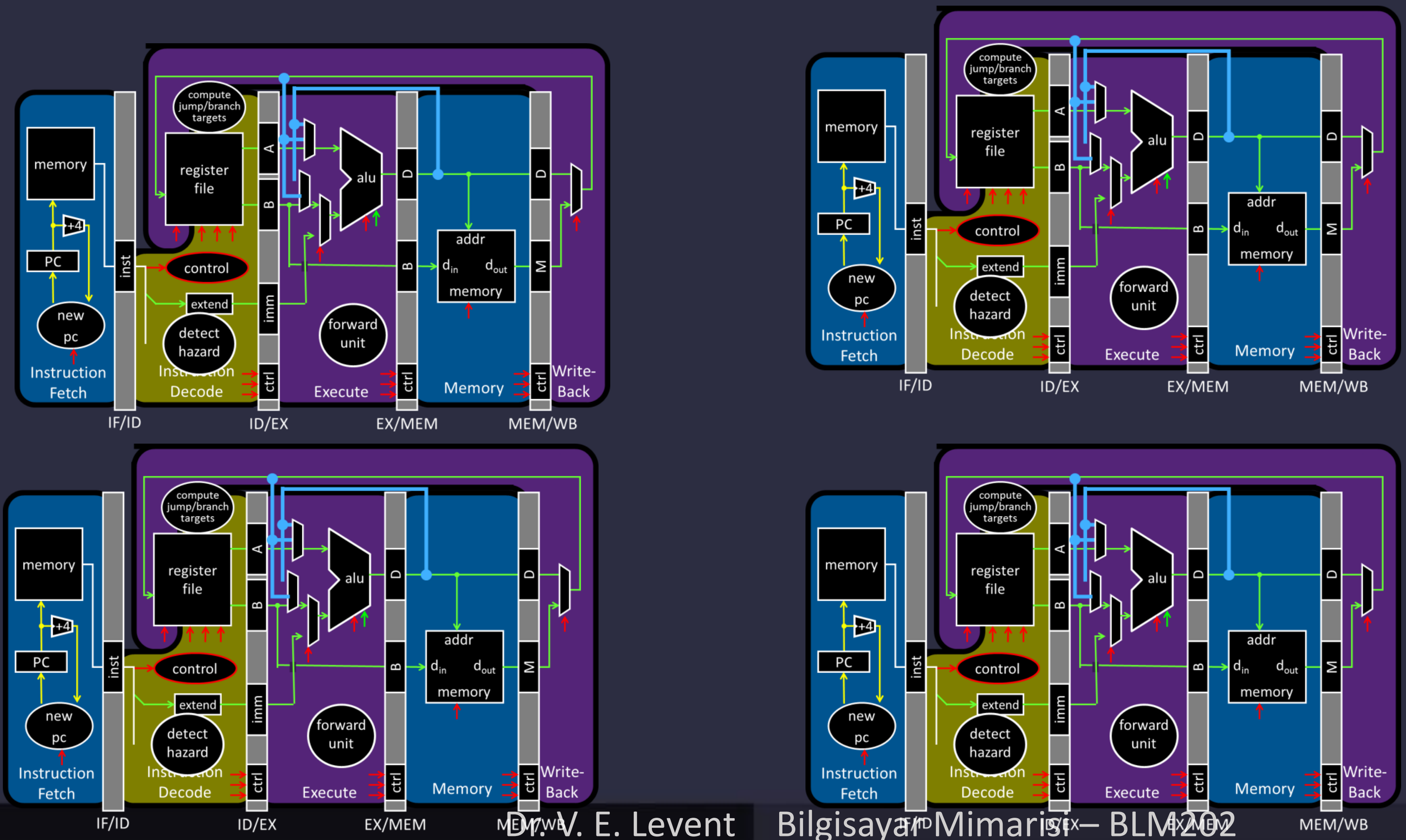
- Parallelism
- Multi Core
- Synchronization

Big Picture: Multicore and Parallelism

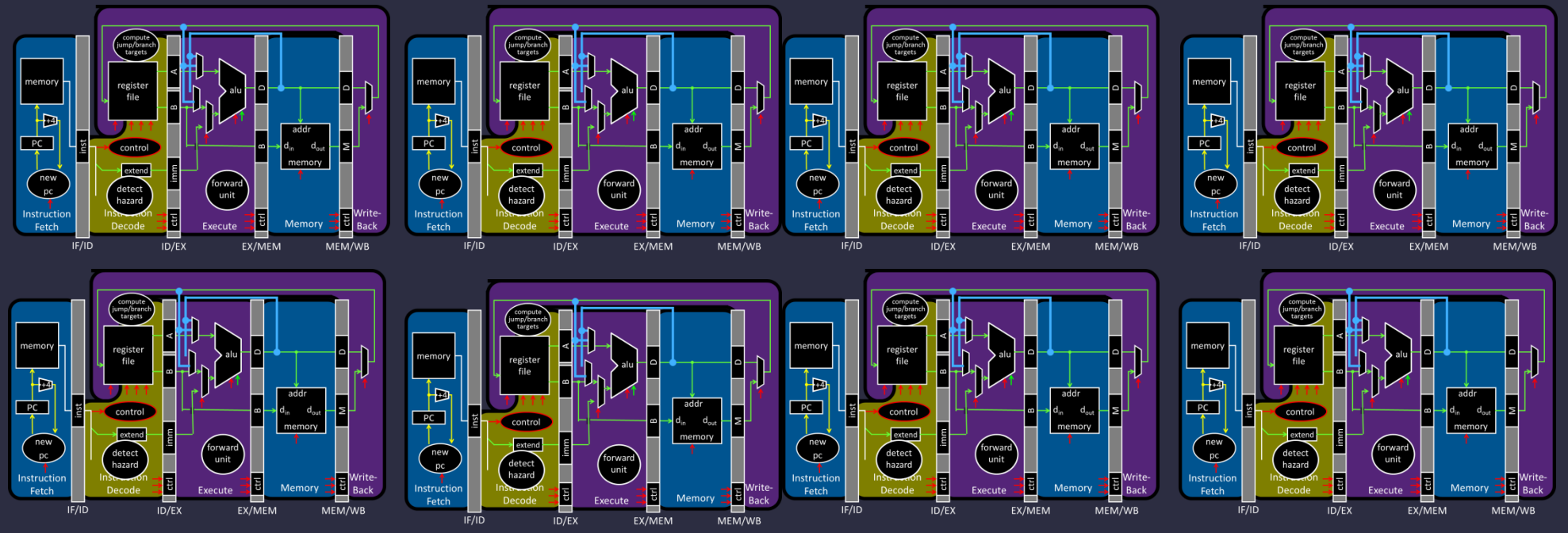


Big Picture: Multicore and Parallelism

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

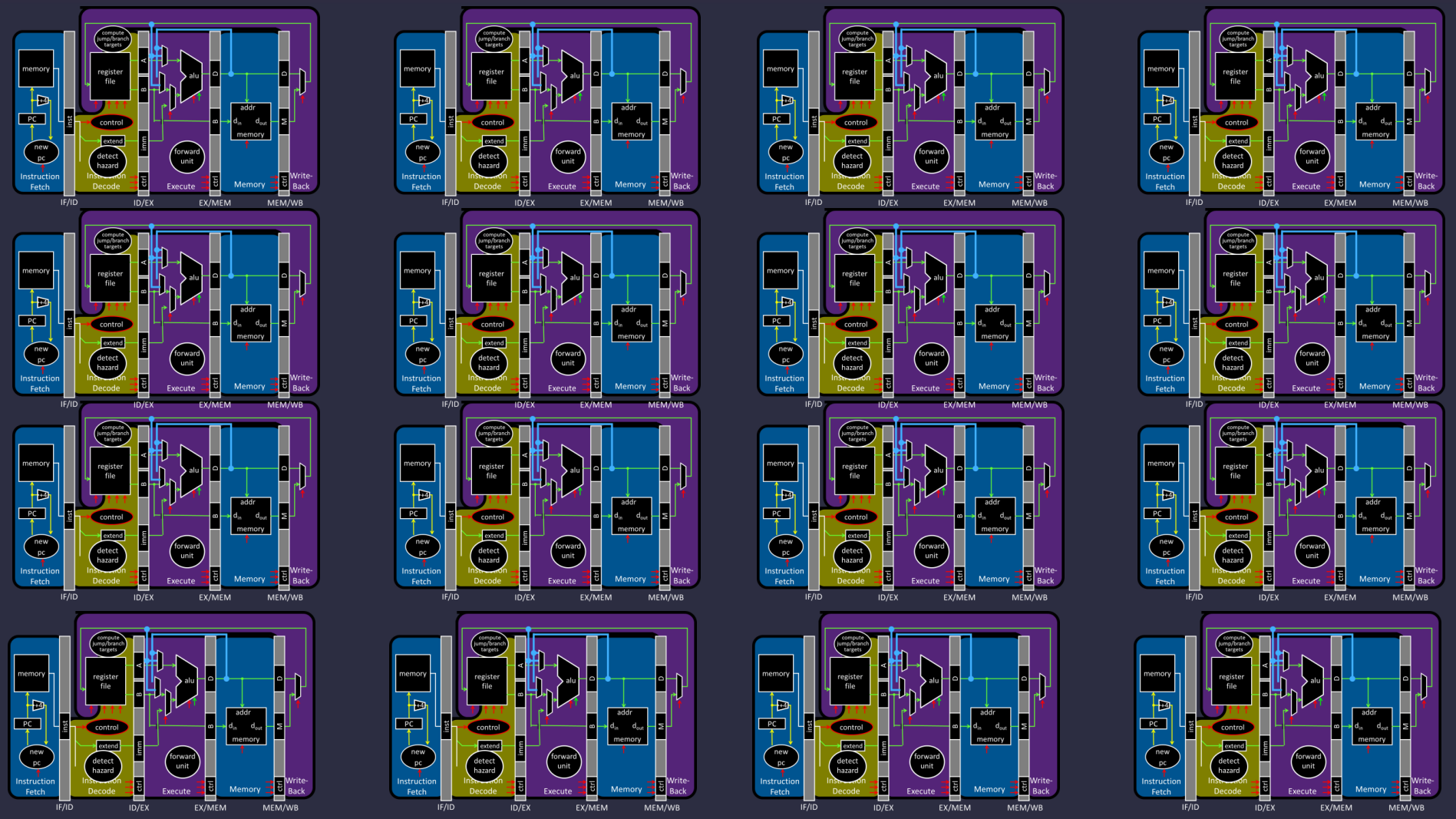


Big Picture: Multicore and Parallelism



Big Picture: Multicore and Parallelism

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



Pitfall: Amdahl's Law

Execution time after improvement =

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

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

Scaling Example

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

- Speed up from 10 to 100 processors?

Single processor: Time = (10 + 100) × t_{add}

10 processors

- Speedup = 110/10× t_{add}

100 processors

- Speedup = 110/100 × t_{add}

Assumes load can be balanced across processors

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.

Performance Improvement 101

2 Classic Goals of Architects:

- ↓ Clock period (↑ Clock frequency)
- ↓ Cycles per Instruction (↑ IPC)

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

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
  - 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

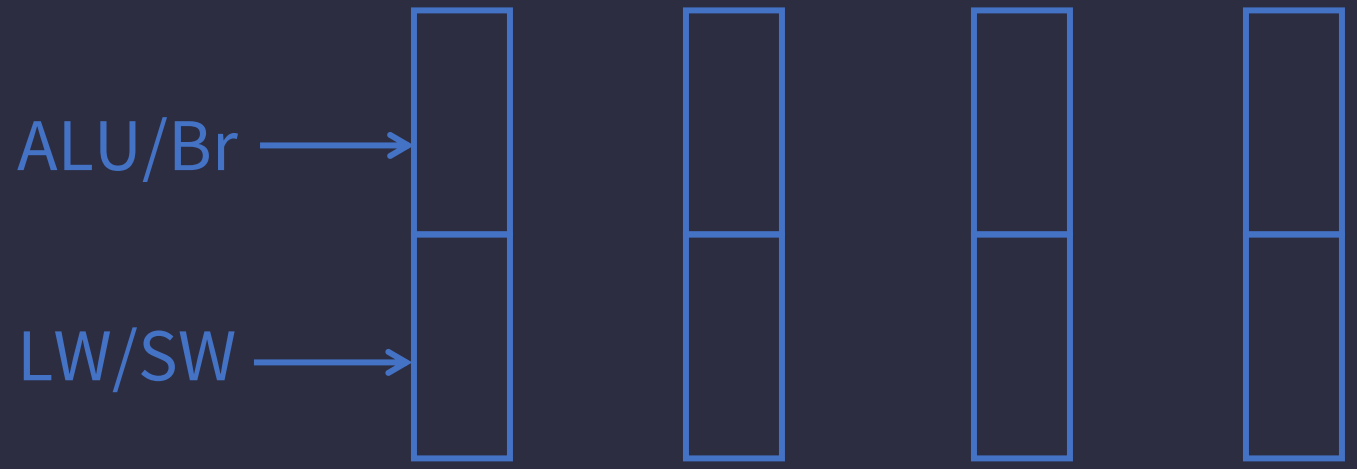
Instruction-Level Parallelism (ILP)

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



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 RISC-V

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

RISC-V 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

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

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

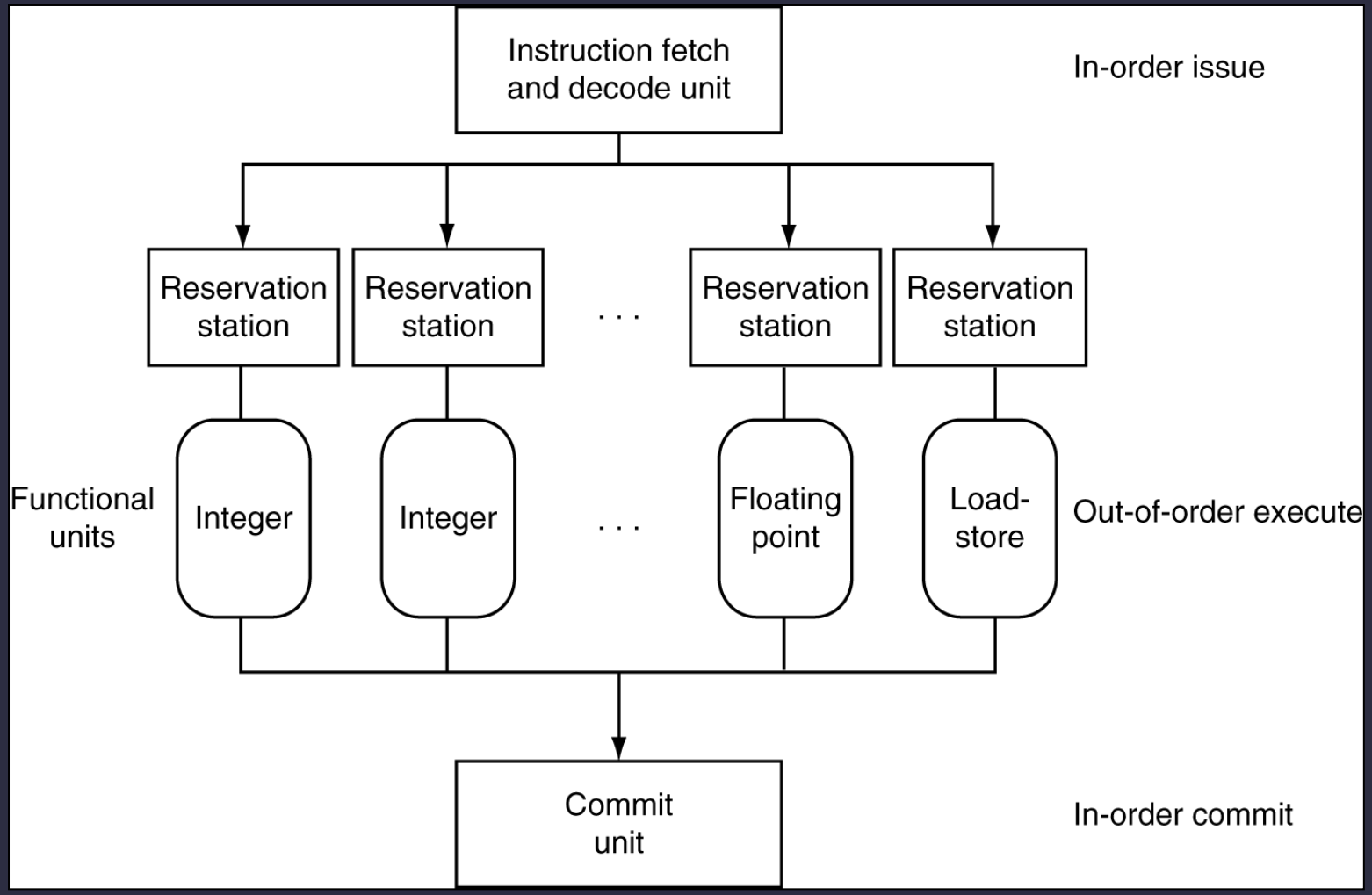
aka SuperScalar Processor (c.f. Intel)

- 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

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 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.

Improving IPC via ILP TLP

Exploiting Thread-Level parallelism

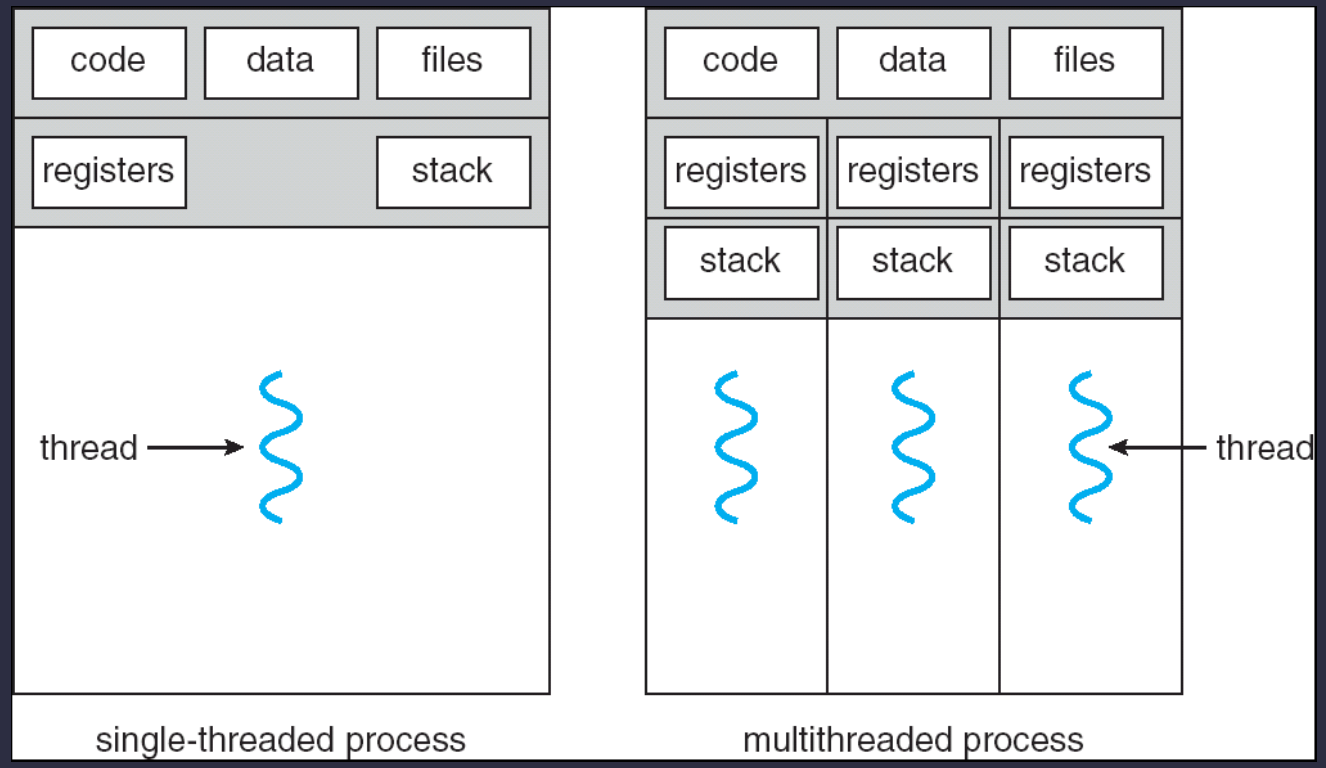
Hardware multithreading to improve utilization:

- 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

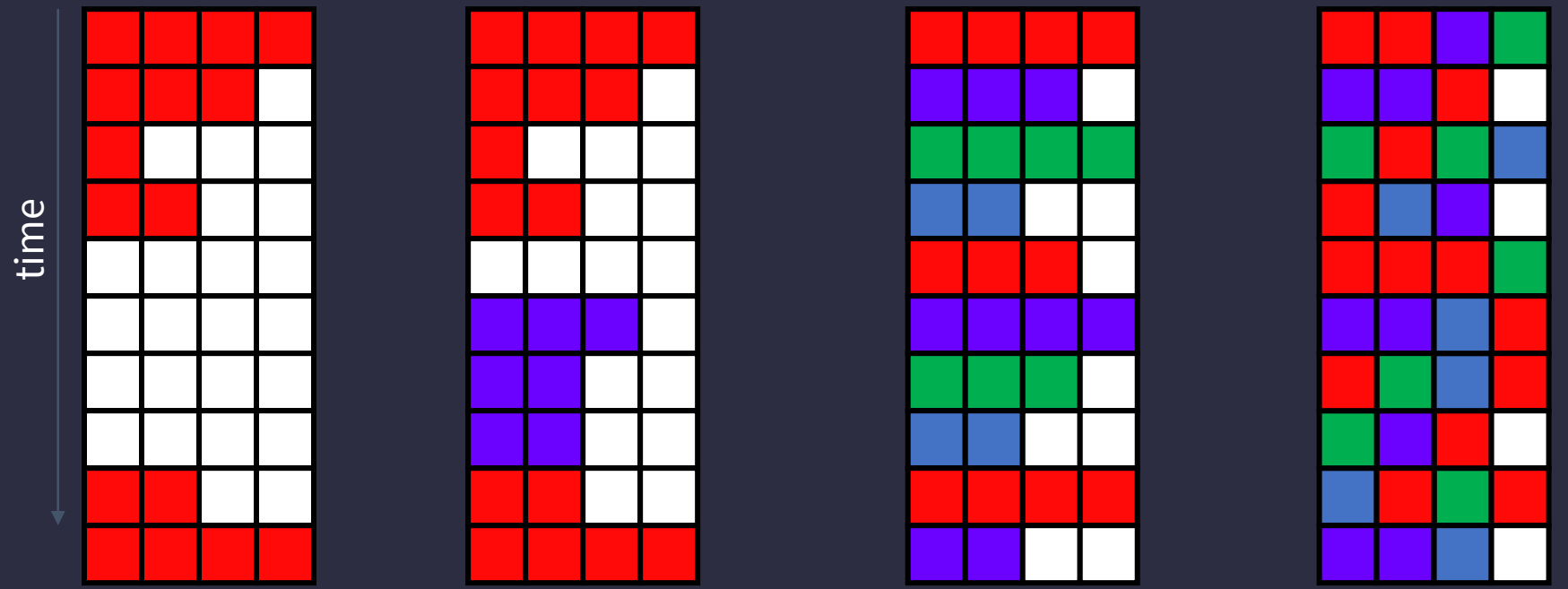
Threads: share code, data, files, **not** regs or stack



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

Hyperthreading

Multi-Core vs. Multi-Issue vs. HT

Programs:	Num. Pipelines:	Pipeline Width:
N	*1*	*N*
N	*1*	1
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

Parallel Programming

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

A: Software must be written as parallel program

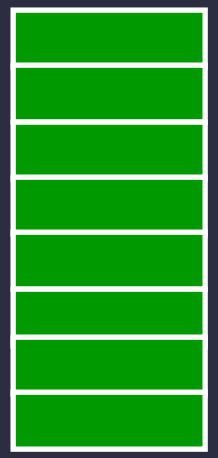
Multicore difficulties

- 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*

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
- How do you write parallel programs?

  ... without knowing exact underlying architecture?

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

- HW support for synchronization

How to write parallel programs

- Threads and processes

Parallelism and Synchronization

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

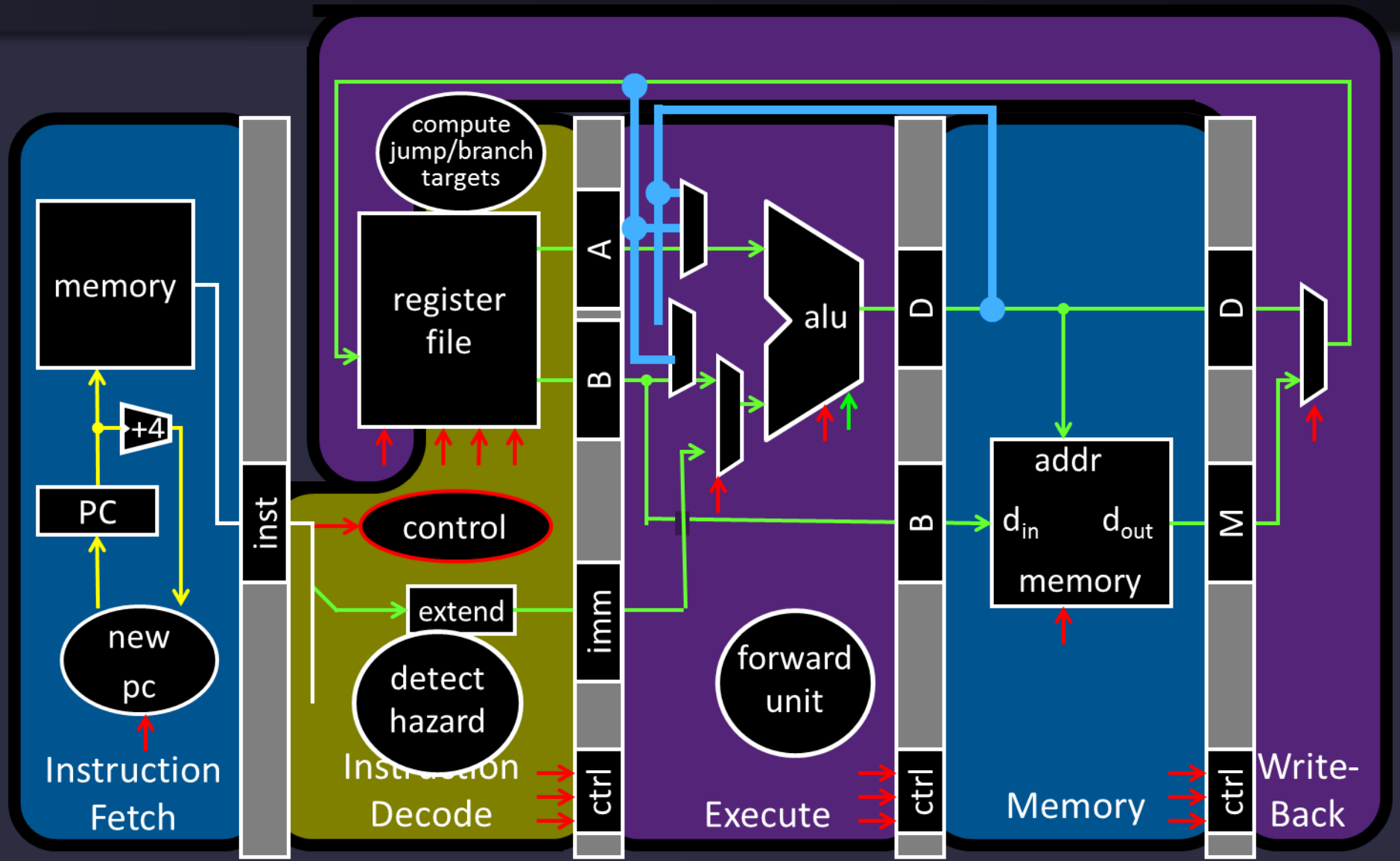
Parallelism and Synchronization

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

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

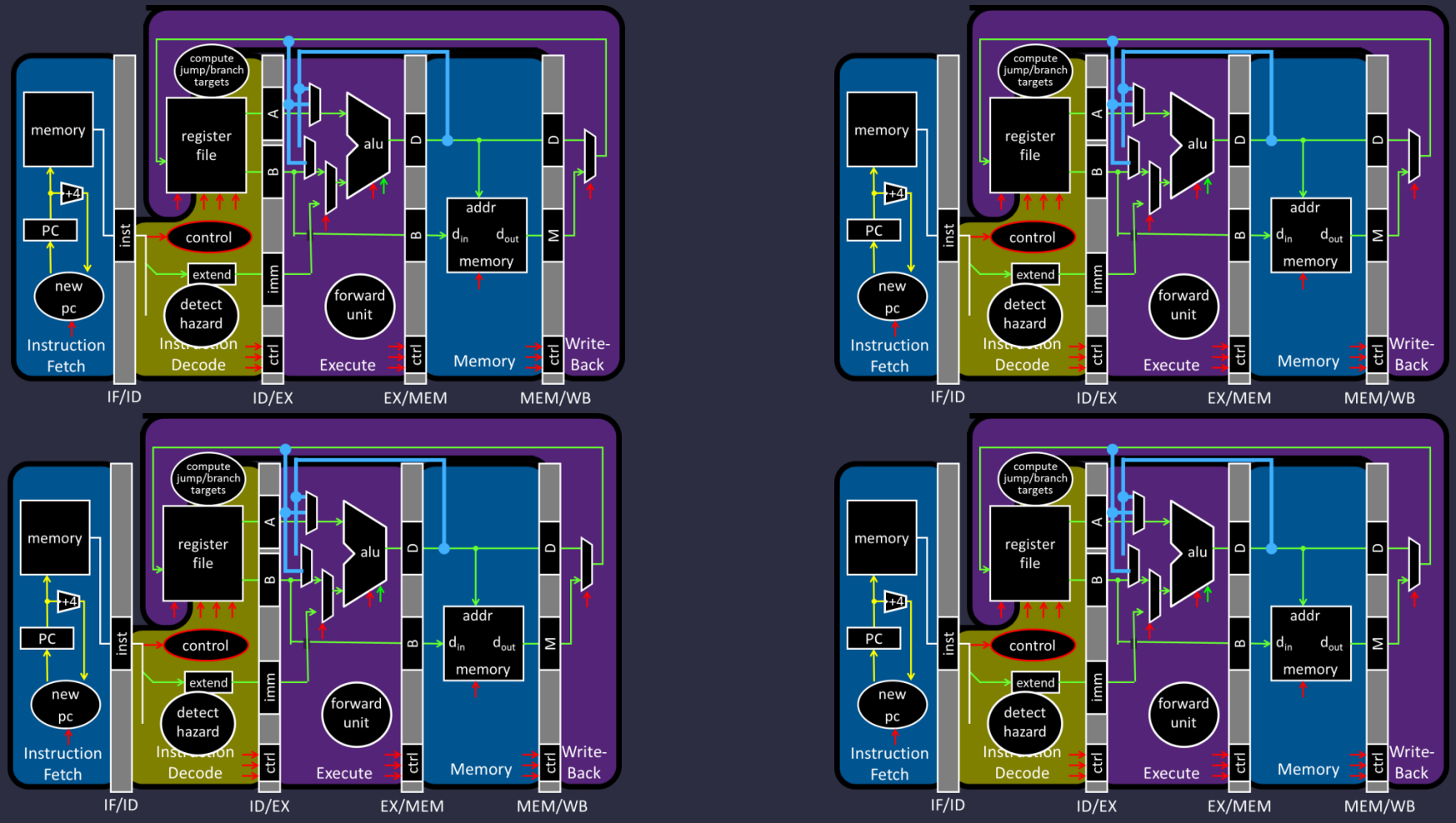
As a result, processors can see different (incoherent) values to the *same* memory location.

Parallelism and Synchronization



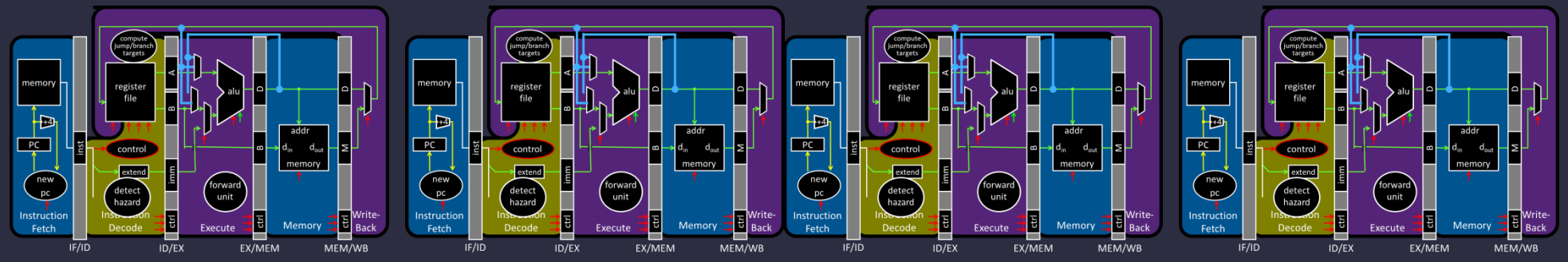
Parallelism and Synchronization

Each processor core has its own L1 cache



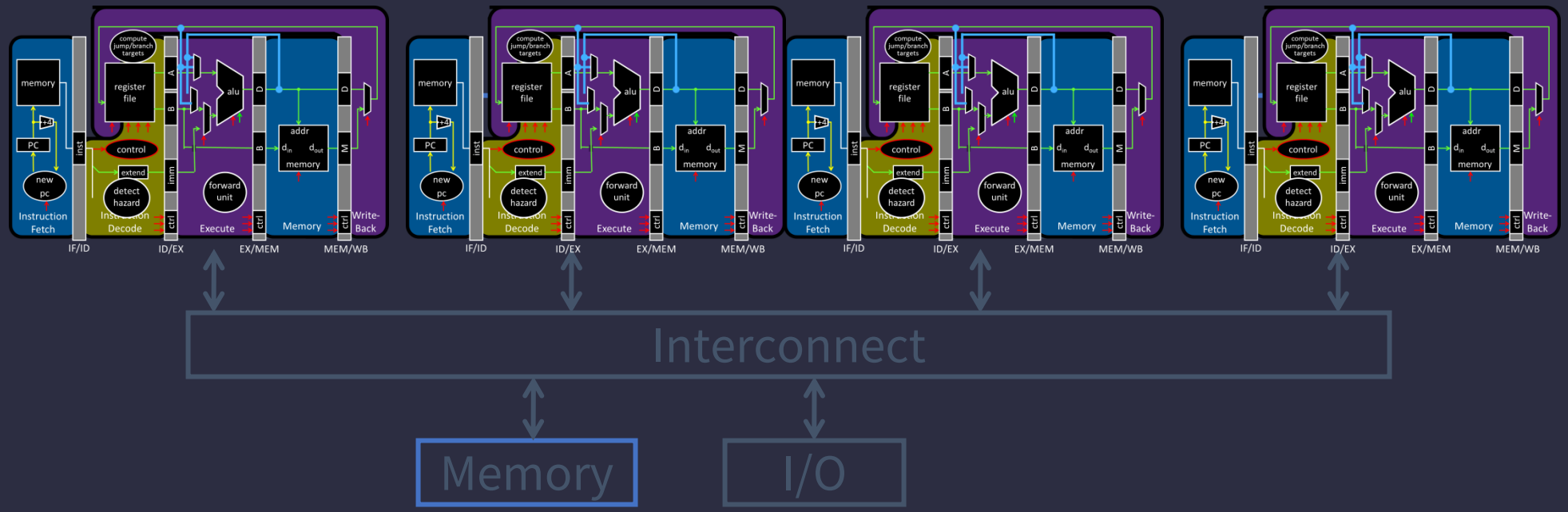
Parallelism and Synchronization

Each processor core has its own L1 cache



Parallelism and Synchronization

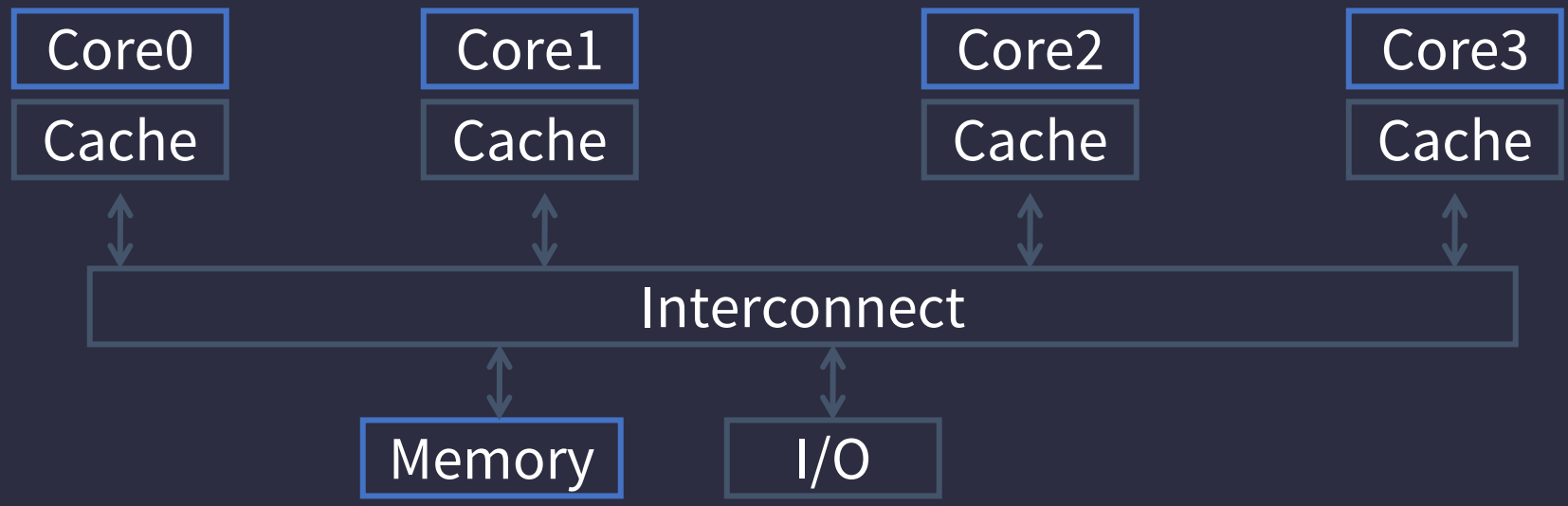
Each processor core has its own L1 cache



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



Cache Coherency Problem

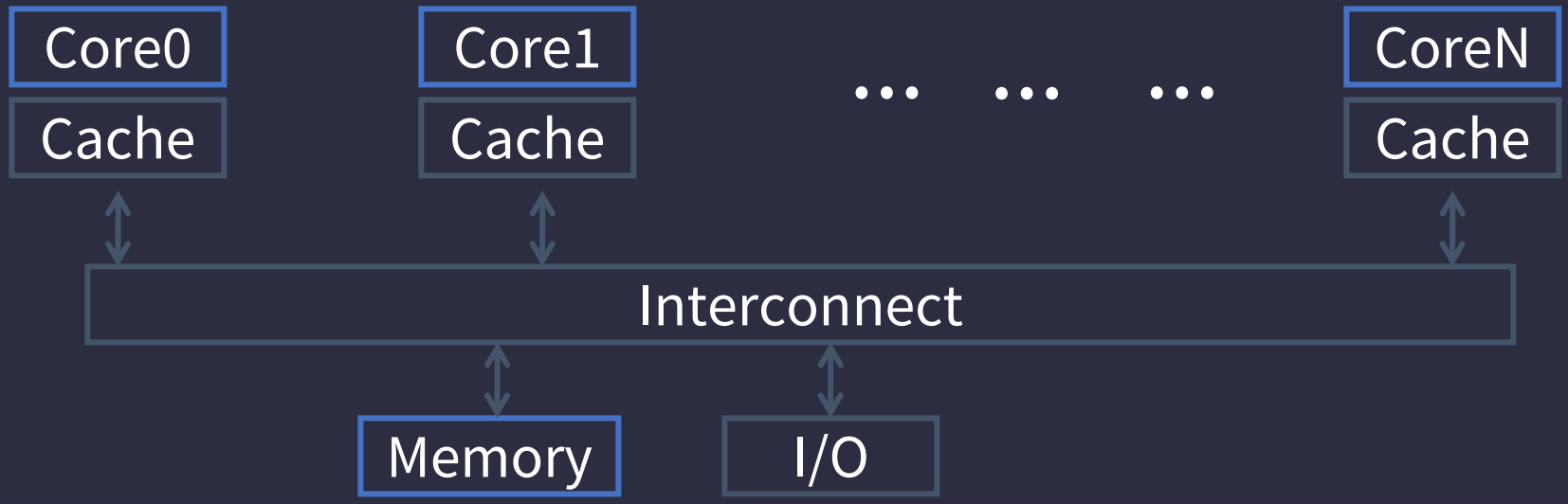
Thread A (on Core0)

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

Thread B (on Core1)

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

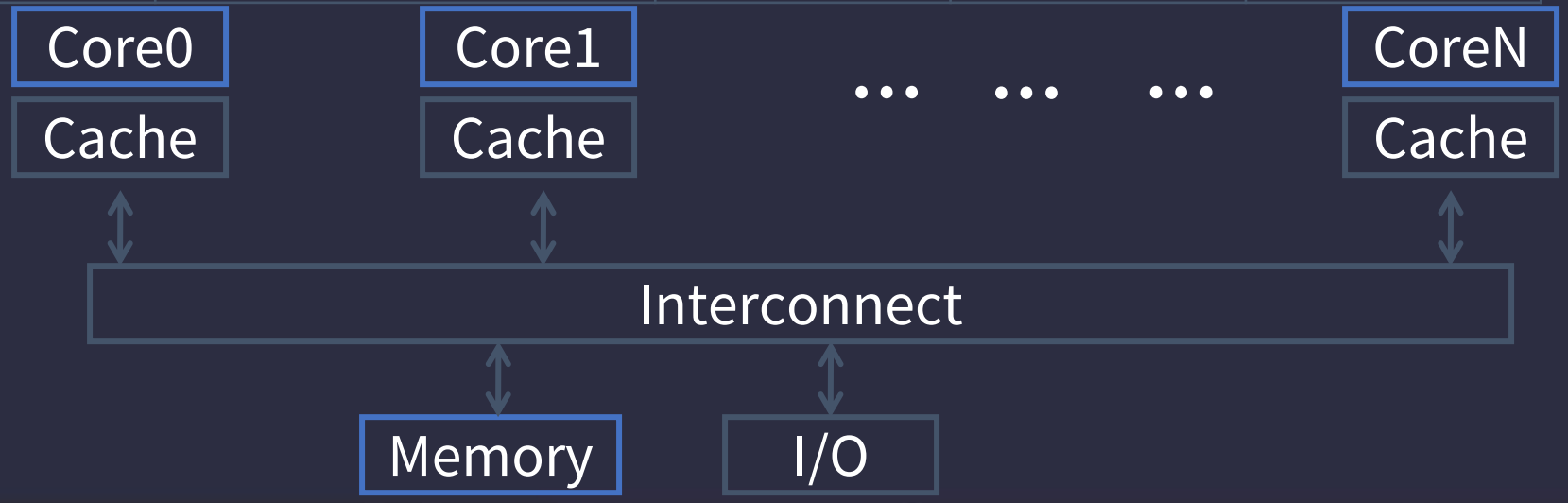
What will the value of x be after both loops finish?



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)
  $\Rightarrow$ read returns written value
  - (preserve program order)
- P_1 writes X before P_2 reads X
  $\Rightarrow$ read returns written value
  - (coherent memory view, can't read old value forever)
- P_1 writes X and P_2 writes X
  $\Rightarrow$ 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

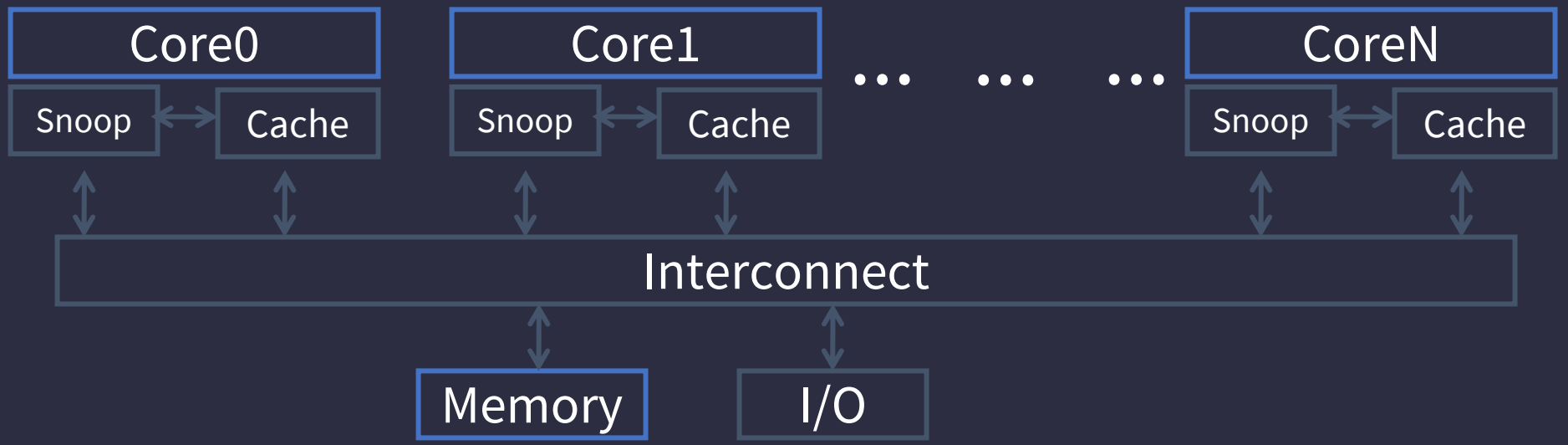
Snooping protocols

- 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	1	1	1