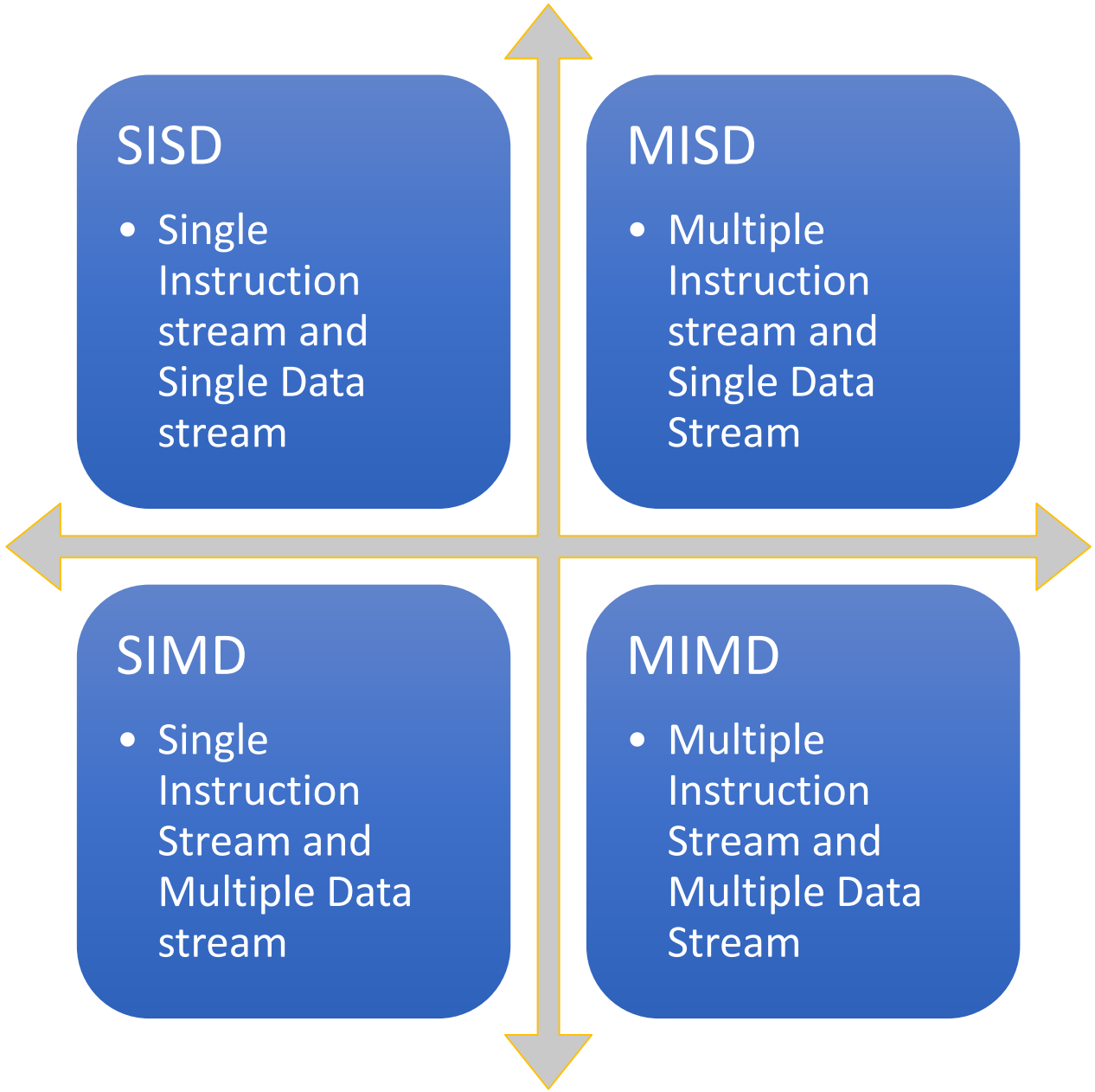
FLYNN'S TAXONOMY OF COMPUTING ARCHITECTURES



TAXONOMY

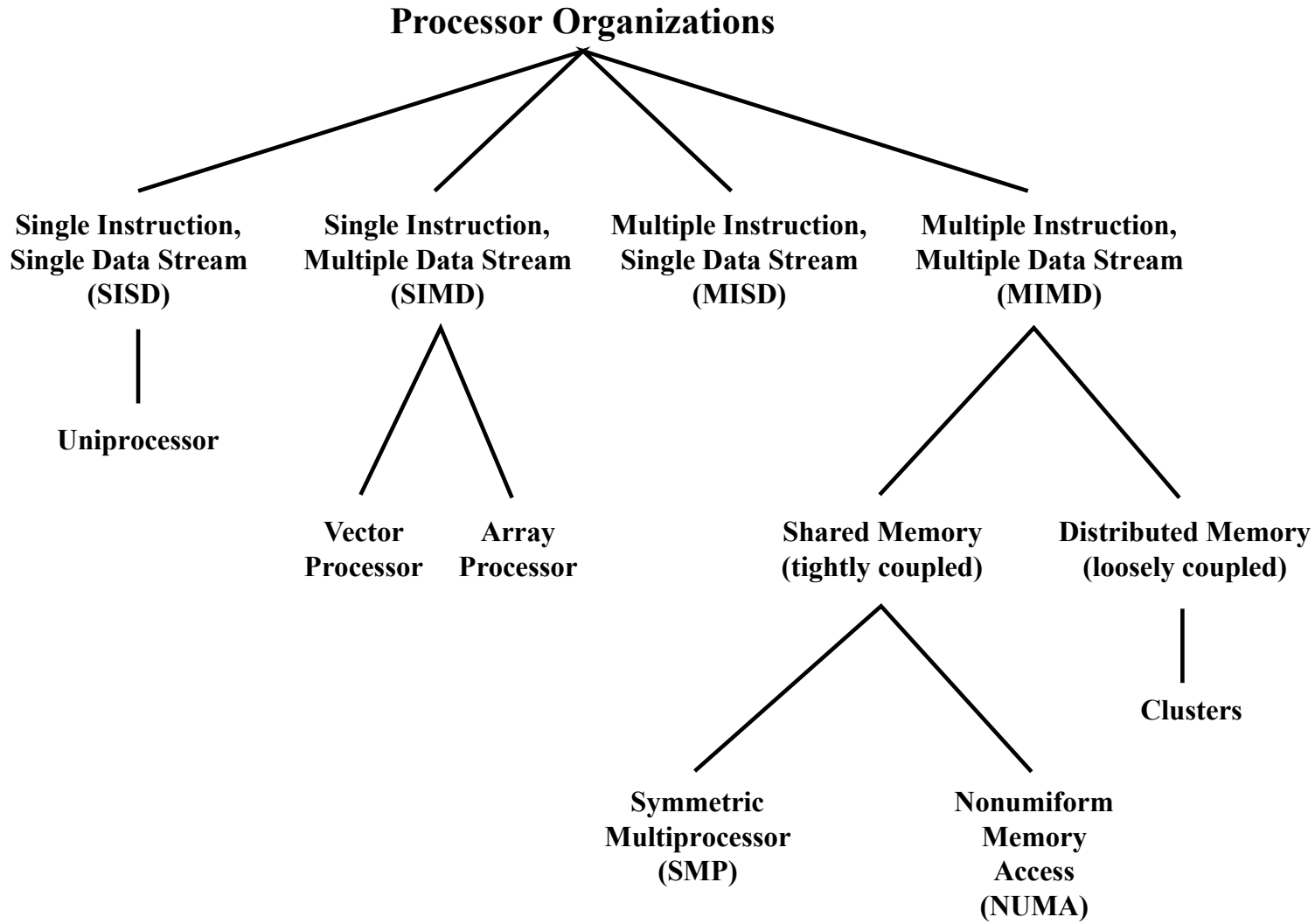


Figure 17.1 A Taxonomy of Parallel Processor Architectures

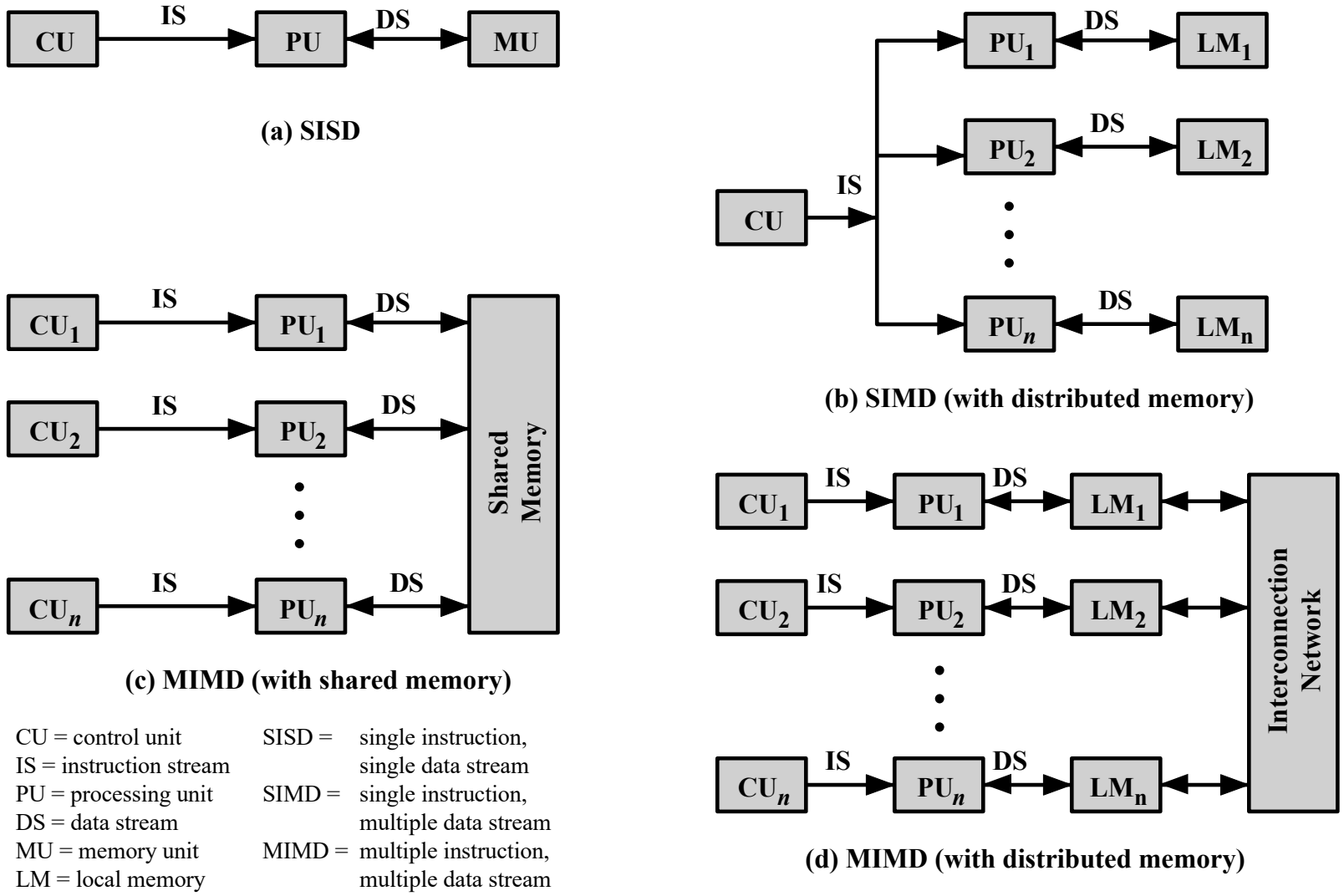


Figure 17.2 Alternative Computer Organizations

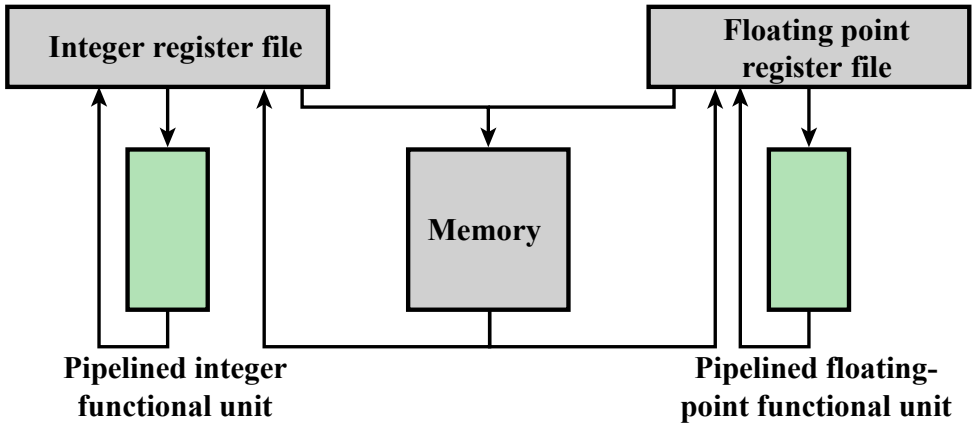
Advanced Architectural Concepts – Exploiting Parallelism

- **Can we achieve CPI < 1? (i.e., can we have IPC > 1?)**
- Most State-of-the-Art Microprocessors do achieve IPC > 1 (***Multiple-Issue Processors***).
- "Superscalar" execution or Instruction Level Parallelism (ILP)
- "Out-of-order" Execution => Instruction Window and Prefetch => Reorder Buffers
- "VLIW" Ex: Intel/HP Titanium

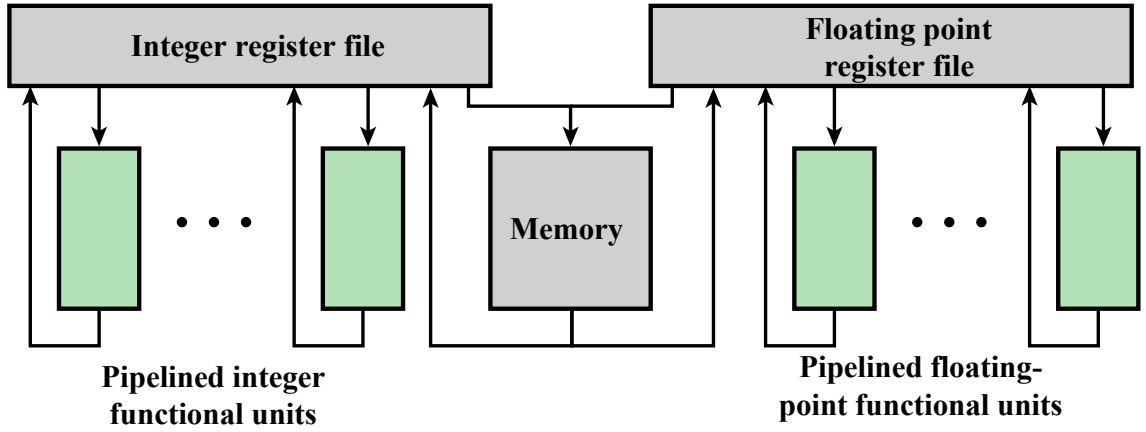
How ILP Works

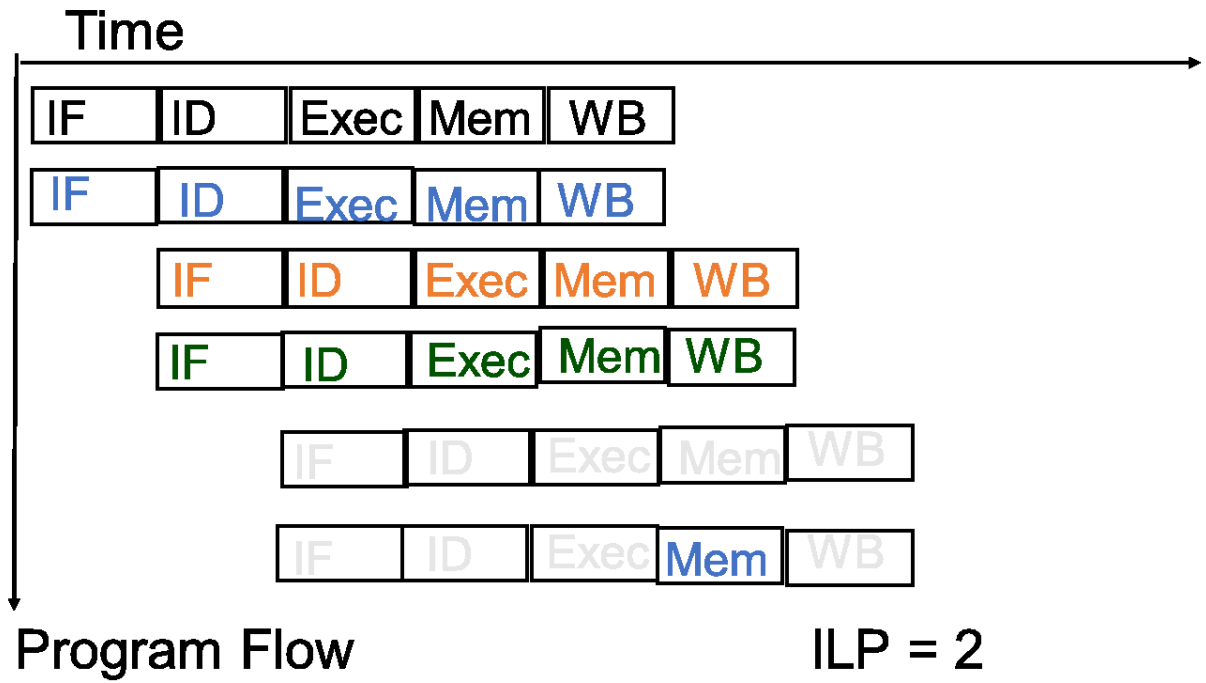
- Issuing multiple instructions per cycle would require fetching multiple instructions from memory per cycle => called Superscalar degree or Issue width
- To find independent instructions, we must have a big pool of instructions to choose from, called instruction buffer (IB). As IB length increases, complexity of decoder (control) increases that increases the datapath cycle time
- Prefetch instructions sequentially by an IFU that operates independently from datapath control. Fetch instruction (PC)+L, where L is the IB size or as directed by the branch predictor.

HIGH-LEVEL VIEW OF SUPERSCALAR ORGANIZATION



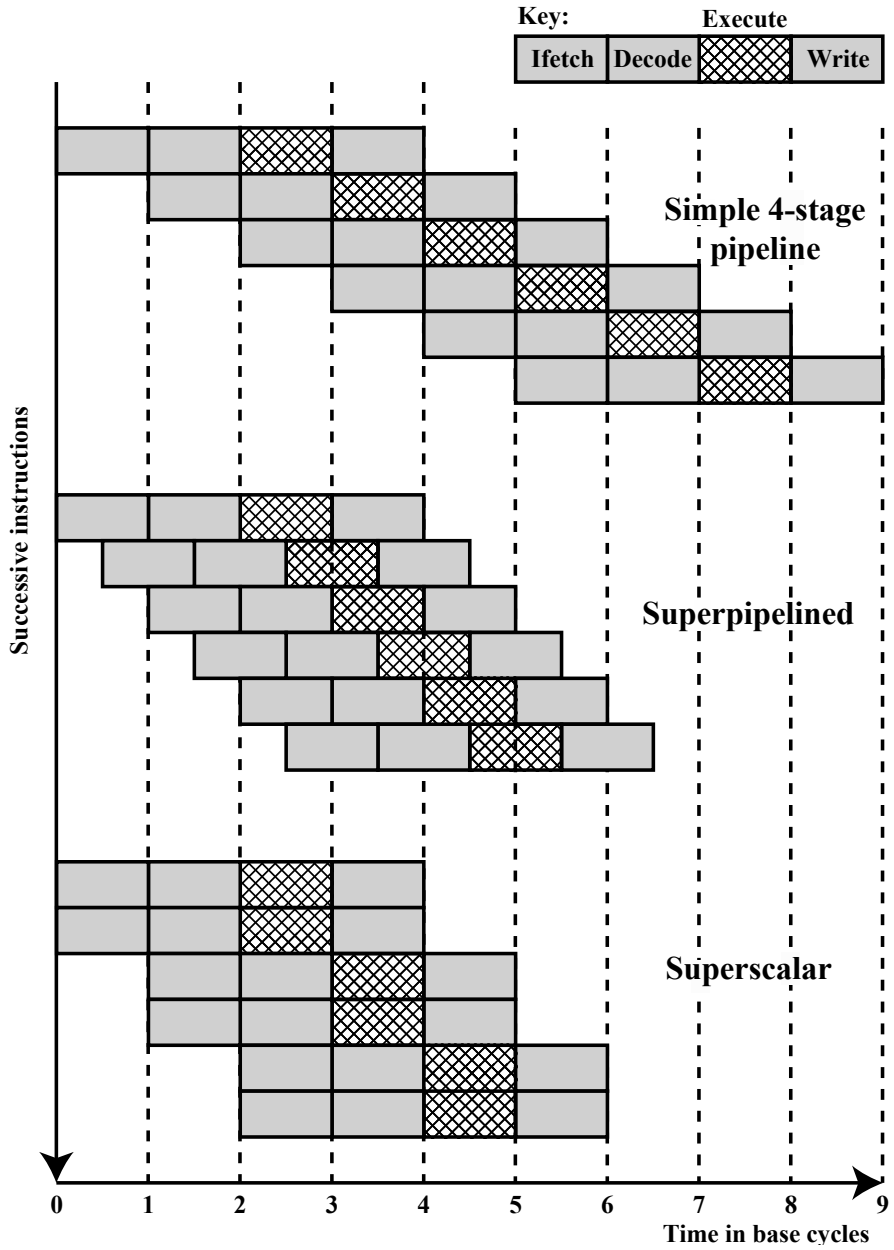
(a) Scalar organization





EX: Pentium, SPARC, MIPS 10000, IBM Power PC

SUPERSCALAR VS SUPER-PIPELINING



An Opportunity...

- But consider:

```
ADD r1, r2 -> r3
ADD r4, r5 -> r6
```

  - Why not execute them ***at the same time***? (We can!)

- What about:

```
ADD r1, r2 -> r3
ADD r4, r3 -> r6
```

  - In this case, ***dependences*** prevent parallel execution

- What about three instructions at a time?
  - Or four instructions at a time?

How many Checks?

- For two instructions: 2 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2    (2 checks)
```

- For three instructions: 6 checks

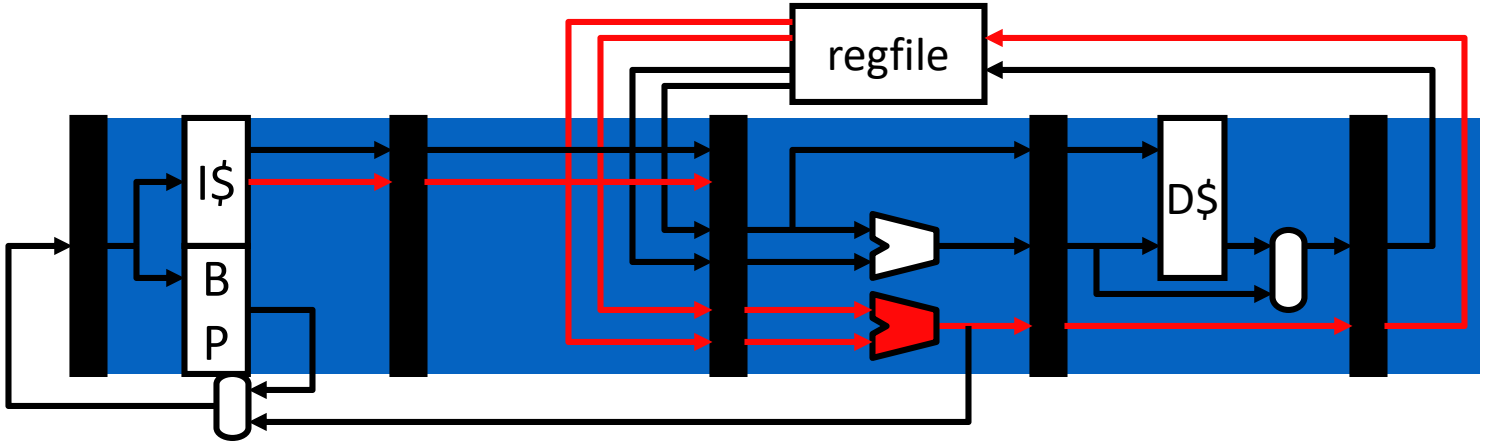
```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2    (2 checks)
ADD src1_3, src2_3 -> dest_3    (4 checks)
```

- For four instructions: 12 checks

```
ADD src1_1, src2_1 -> dest_1
ADD src1_2, src2_2 -> dest_2    (2 checks)
ADD src1_3, src2_3 -> dest_3    (4 checks)
ADD src1_4, src2_4 -> dest_4    (6 checks)
```

- Plus checking for load-to-use stalls from prior *n* loads

A Typical Dual-Issue Pipeline (2 of 2)



- Multi-ported register file
  - Larger area, latency, power, cost, complexity
- Multiple execution units
  - Simple adders are easy, but bypass paths are expensive
- Memory unit
  - Single load per cycle (stall at decode) probably okay for dual issue
  - Alternative: add a read port to data cache
    - Larger area, latency, power, cost, complexity

SUPERSCALAR CHALLENGES - FRONT END

- **Superscalar instruction fetch**
  - Modest: fetch multiple instructions per cycle
  - Aggressive: buffer instructions and/or predict multiple branches
- **Superscalar instruction decode**
  - Replicate decoders
- **Superscalar instruction issue**
  - Determine when instructions can proceed in parallel
  - More complex stall logic - order $\mathbf{N^2}$ for *N*-wide machine
  - Not all combinations of types of instructions possible
- **Superscalar register read**
  - Port for each register read (4-wide superscalar ➔ 8 read "ports")
  - Each port needs its own set of address and data wires
    - Latency & area $\propto$ #ports^2

Superscalar Challenges - Back End

- **Superscalar instruction execution**
  - Replicate arithmetic units (but not all, for example, integer divider)
  - Perhaps multiple cache ports (slower access, higher energy)
    - Only for 4-wide or larger (why? only ~35% are load/store insn)
- **Superscalar bypass paths**
  - More possible sources for data values
  - Order (N^2 * P) for *N*-wide machine with execute pipeline depth *P*
- **Superscalar instruction register writeback**
  - One write port per instruction that writes a register
  - Example, 4-wide superscalar ➔ 4 write ports
- **Fundamental challenge:**
  - Amount of ILP (instruction-level parallelism) in the program
  - Compiler must schedule code and extract parallelism

Multiple-Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - \+ Executes unmodified sequential programs
  - – Hardware must figure out what can be done in parallel
  - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)
- **Very Long Instruction Word (VLIW)**
  - **\- Compiler identifies independent instructions**, new ISA
  - \+ Hardware can be simple and perhaps lower power
  - E.g., TransMeta Crusoe (4-wide), most DSPs
  - **Variant: Explicitly Parallel Instruction Computing (EPIC)**
    - A bit more flexible encoding & some hardware to help compiler
    - E.g., Intel Itanium (6-wide)
- **Dynamically-scheduled superscalar**
  - **Hardware extracts more ILP by on-the-fly reordering of instructions**
  - Intel Atom/Core/Xeon, AMD Opteron/Ryzen, some ARM A-series

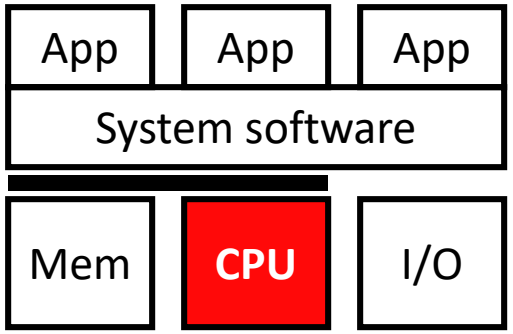
Another Opportunity for Parallelism

- Key idea: Allow instructions behind stall to proceed

```
DIVD F0,F2,F4
ADDD F10,F0,F8
MULD F12,F8,F14
SUBD F6,F16,F18
```

  - Enables out-of-order execution => out-of-order completion
  - ID stage checks for hazards. If no hazards, issue the instruction for execution.

    Scoreboard dates to CDC 6600 in 1963



- Code scheduling
  - To reduce pipeline stalls
  - To increase ILP (insn level parallelism)
- Static scheduling by the compiler
  - Approach & limitations
- Dynamic scheduling in hardware
  - Register renaming
  - Instruction selection
  - Handling memory operations

Dynamic Instruction Scheduling

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
  - Software based instruction scheduling → static scheduling
  - Hardware based instruction scheduling → dynamic scheduling

- What information does the compiler not know that makes static scheduling difficult?
  - Anything that is determined at run time: Variable-length operation latency, memory address, branch direction, exceptions,

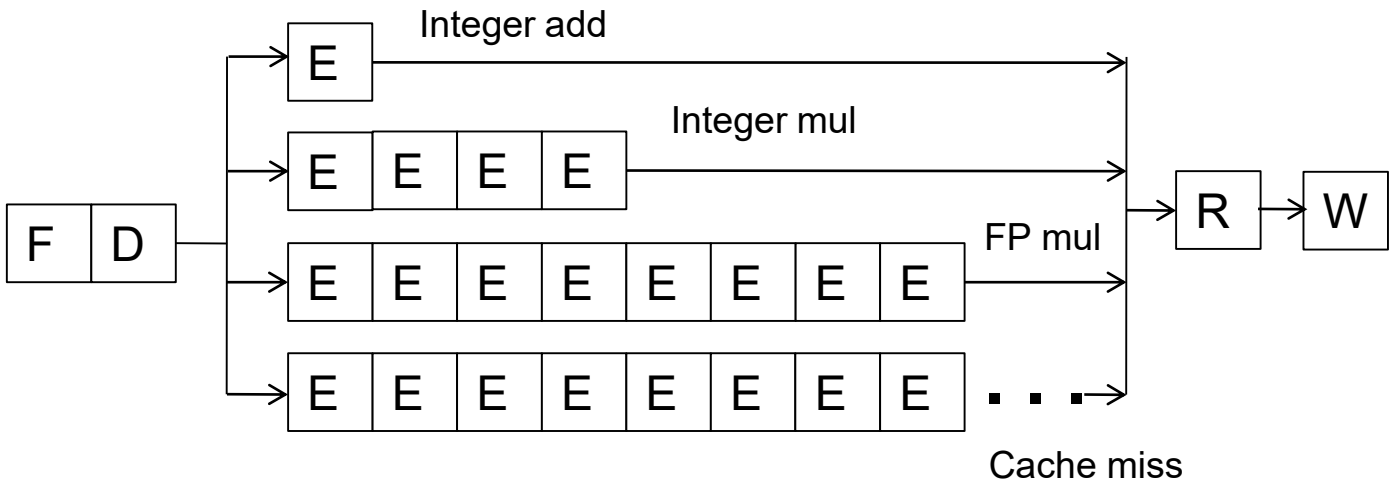
- Hardware has knowledge of dynamic events on a per-instruction basis (i.e., at a very fine granularity)
  - Cache misses
  - Branch mispredictions
  - Load/store addresses

- Wouldn't it be nice if hardware did the scheduling of instructions?

Dynamic Instruction Scheduling

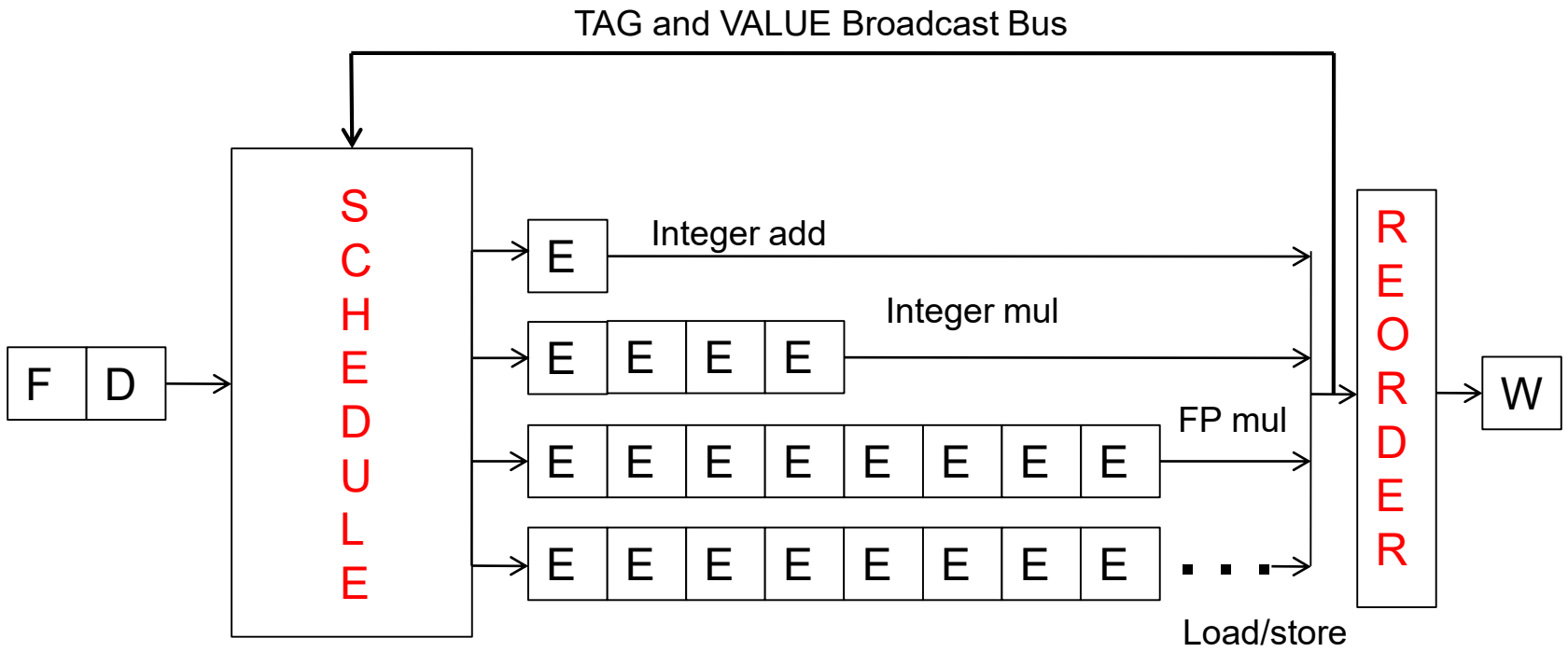
- **Dynamically-scheduled processors**
  - Also called “out-of-order” processors
  - Hardware re-schedules instructions… i.e. Instruction scheduling is done by the hardware on-the-fly during execution
  - Looks at a “window” of instructions waiting to execute
    - Each cycle, picks the next ready instruction(s)
    - …within a sliding window of Von Neumann instructions.
  - As with pipelining and superscalar, ISA unchanged
    - Same hardware/software interface, appearance of in-order
  - Increases scheduling scope
    - Does loop unrolling transparently!
    - Uses branch prediction to “unroll” branches
- Examples:
  - Pentium Pro/II/III (3-wide), Core 2 (4-wide),
    Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)

An In-order Pipeline



- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to a functional unit

Two Humps in a Modern Pipeline



- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Out-of-Order Execution

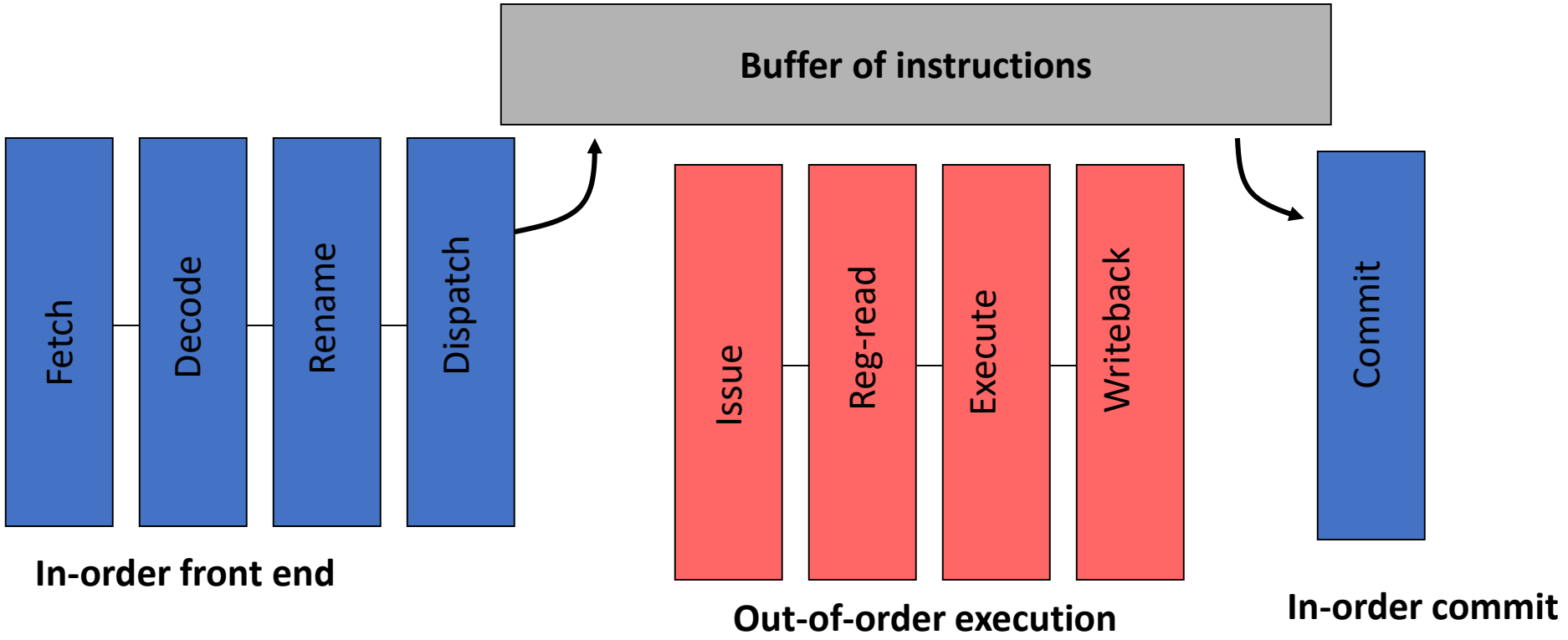
- Key to understanding out-of-order execution:
  - **Data dependencies**

- Two steps to enable out-of-order execution:

  Step #1: Register renaming – to avoid “false” dependencies

  Step #2: Dynamically schedule – to enforce “true” dependencies

Out-of-Order Pipeline



- **Rename**: Resolve false (output and anti-dependence) data dependencies
- **Dispatch**: Act of sending an instruction to a functional unit
- **Commit**: Retire the Instructions (Actual Writeback to the Architectural Register)

DEPENDENCE TYPES

- **RAW** (Read After Write) = "true dependence" (true)

  mul r0 * r1 → **r2**

  …

  add **r2** + r3 → r4

- **WAW** (Write After Write) = "output dependence" (false)

  mul r0 * r1 → **r2**

  …

  add r1 + r3 → **r2**

- **WAR** (Write After Read) = "anti-dependence" (false)

  mul r0 * **r1** → r2

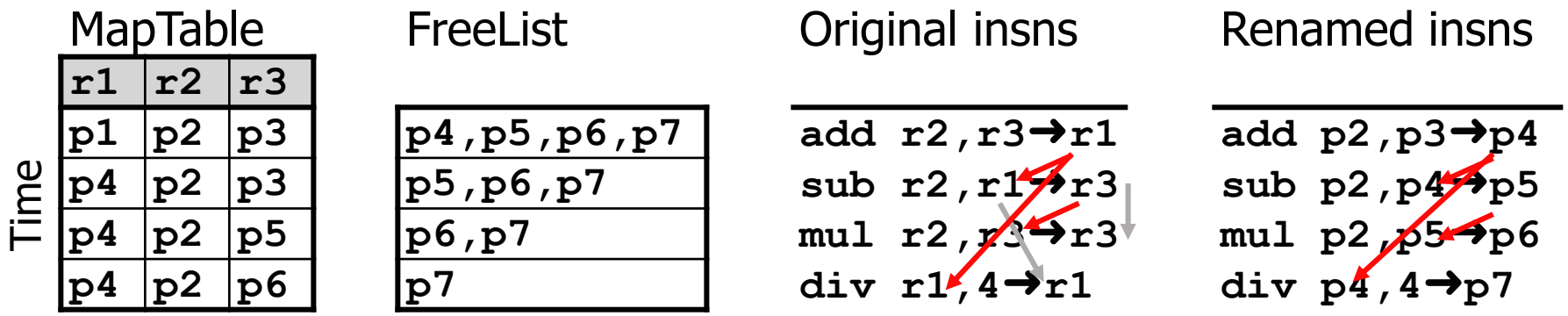
  …

  add r3 + r4 → **r1**

- WAW & WAR are "false", Can be **totally eliminated** by "renaming"

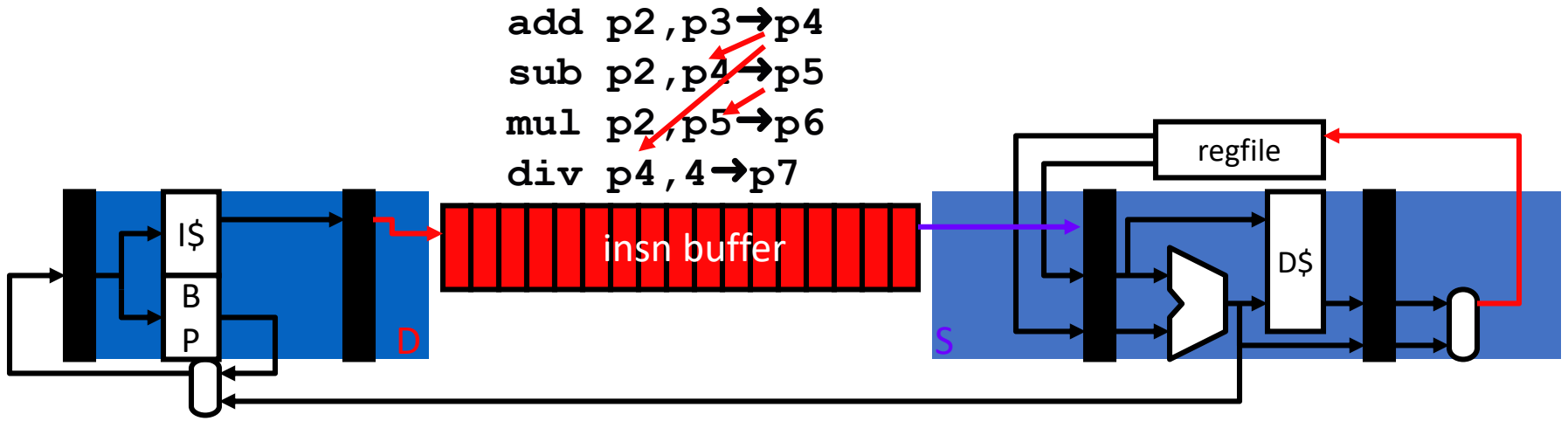
Step #1: Register Renaming

- To eliminate register conflicts/hazards
- "Architected" vs "Physical" registers – level of indirection
  - Names: `r1,r2,r3`
  - Locations: `p1,p2,p3,p4,p5,p6,p7`
  - Original mapping: `r1`→`p1`, `r2`→`p2`, `r3`→`p3`; `p4`–`p7` are "available"



  - Renaming – conceptually write each register once
    - \+ Removes **false** dependences
    - \+ Leaves **true** dependences intact!
  - When to reuse a physical register? After overwriting instruction is done.

STEP #2: DYNAMIC SCHEDULING



Ready Table

Time

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

```
add p2,p3→p4
sub p2,p4→p5 and div p4,4→p7
mul p2,p5→p6
```

- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - Also called "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
  - Execute oldest "ready" instruction, set output as "ready"

What is Multiprocessing

- Parallelism at the Instruction Level is limited because of data dependency => *Speed up is limited!!*
- Abundant availability of program level parallelism, like Loop Level Parallelism.
- How about employing multiple processors to execute the loops => Parallel processing or Multiprocessing.
- With billion transistors on a chip, we can put a few CPUs in one chip => Chip multiprocessor.

HARDWARE MULTITHREADING

- We need to develop a hardware multithreading technique because switching between threads in software is very time-consuming (Why?), so not suitable for main memory (instead of I/O) access, Ex: Multitasking

- Develop multiple PCs and register sets on the CPU so that thread switching can occur without having to store the register contents in main memory (stack, like it is done for context switching).

- Several threads reside in the CPU simultaneously, and execution switches between the threads on main memory access.

- How about both multiprocessors and multithreading on a chip? => Network Processor

- How can we guarantee no dependencies between instructions in a pipeline?
  - One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

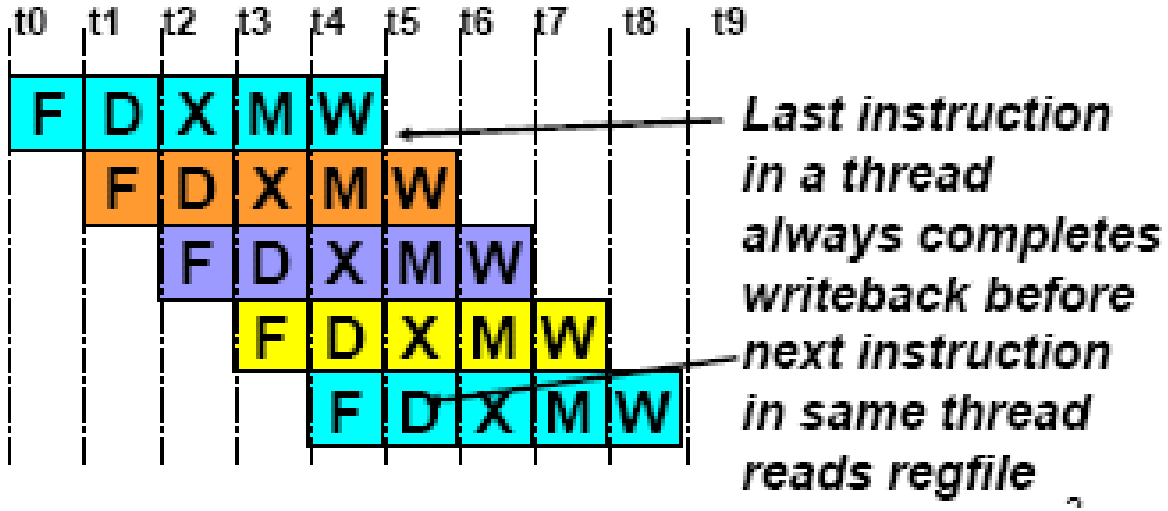
T1: LW r1, 0(r2)

T2: ADD r7, r1, r4

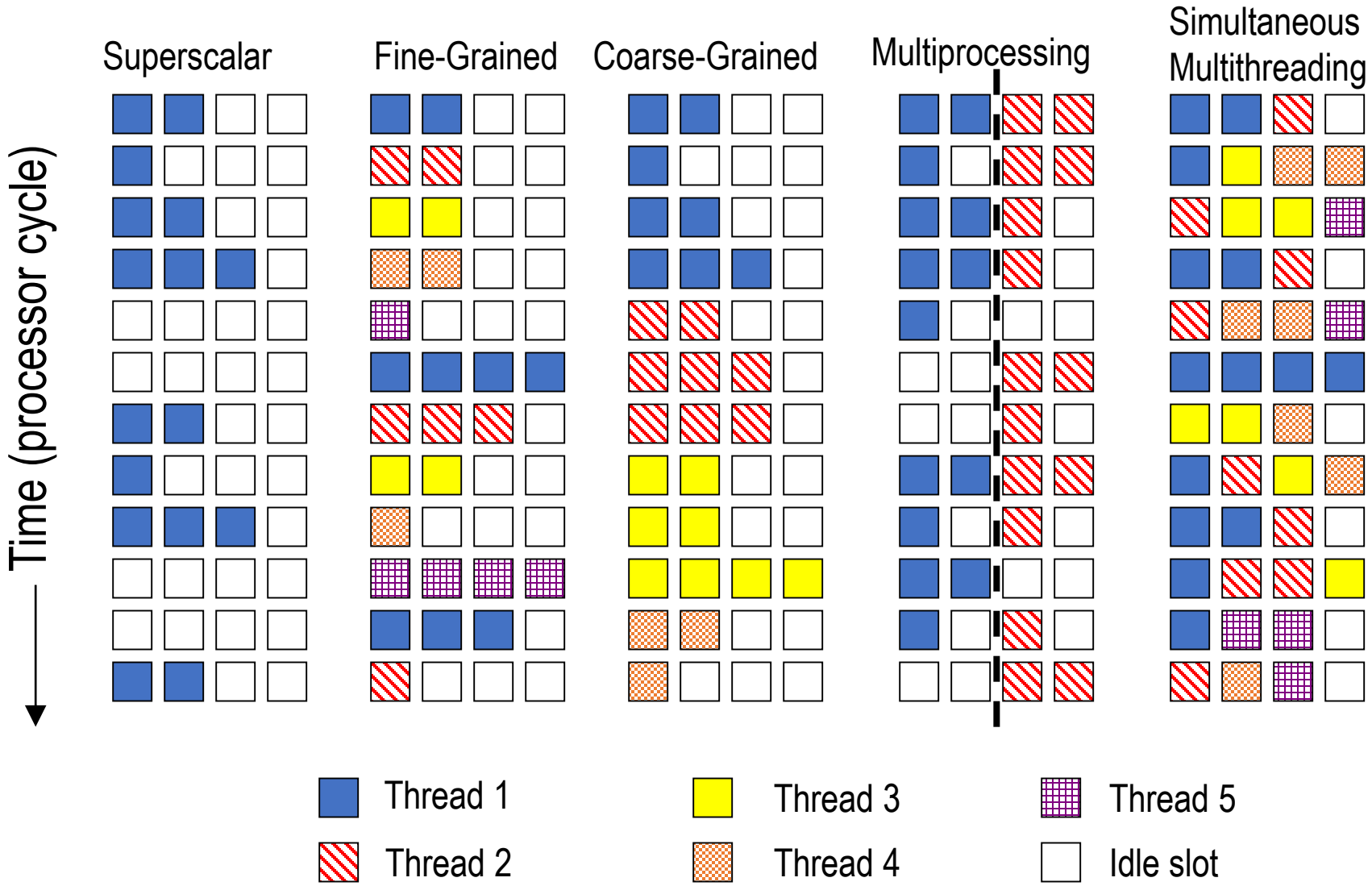
T3: XORI r5, r4, #12

T4: SW 0(r7), r5

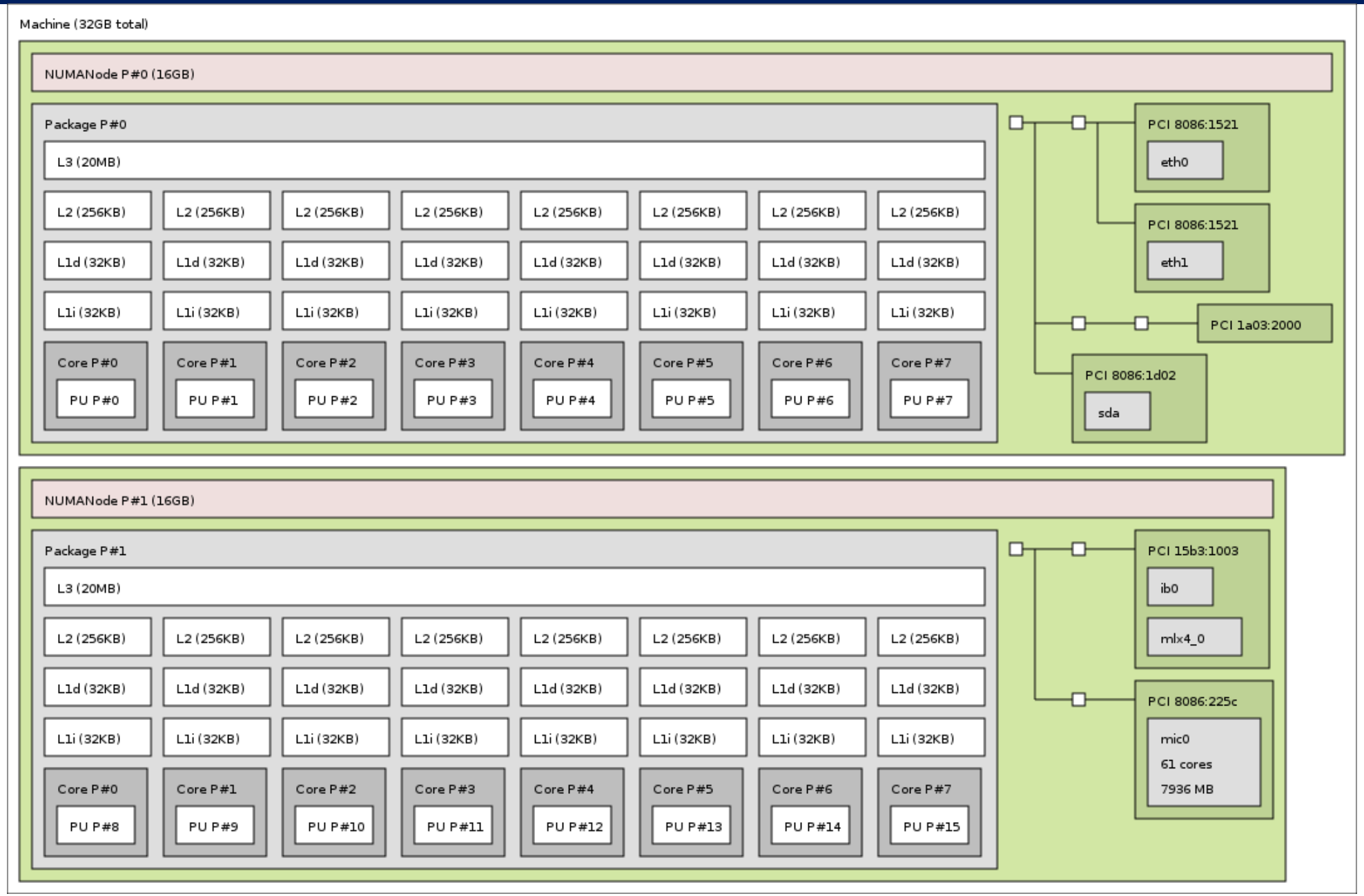
T1: LW r5, 12(r1)



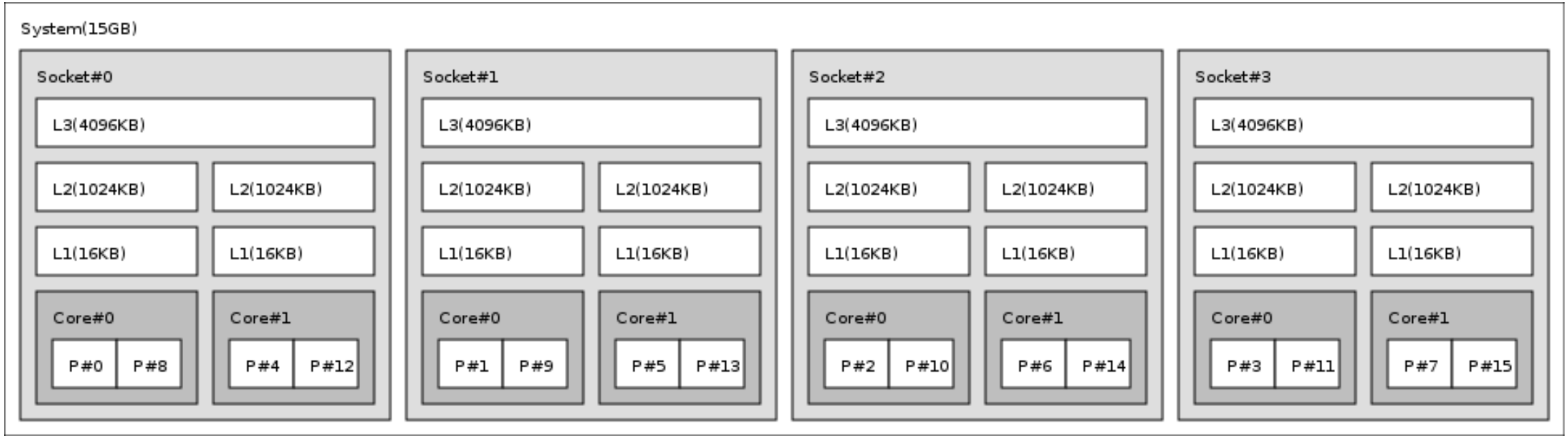
Advanced Systems --Architectural Comparisons (cont.)



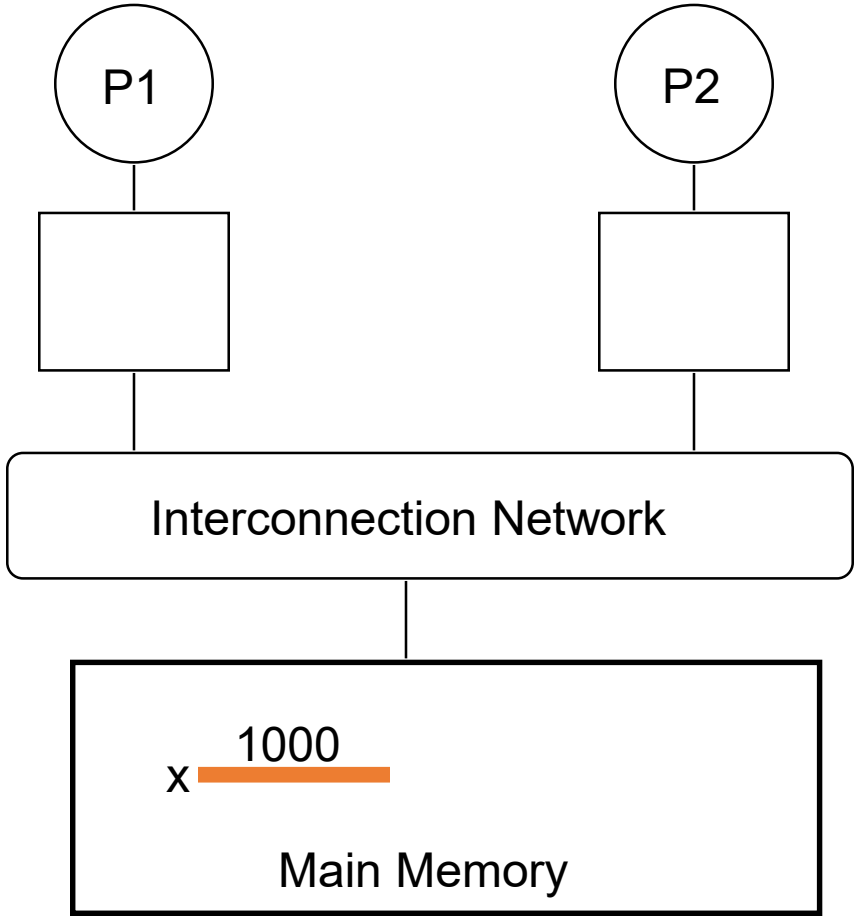
Memory Hierarchy in Modern Servers



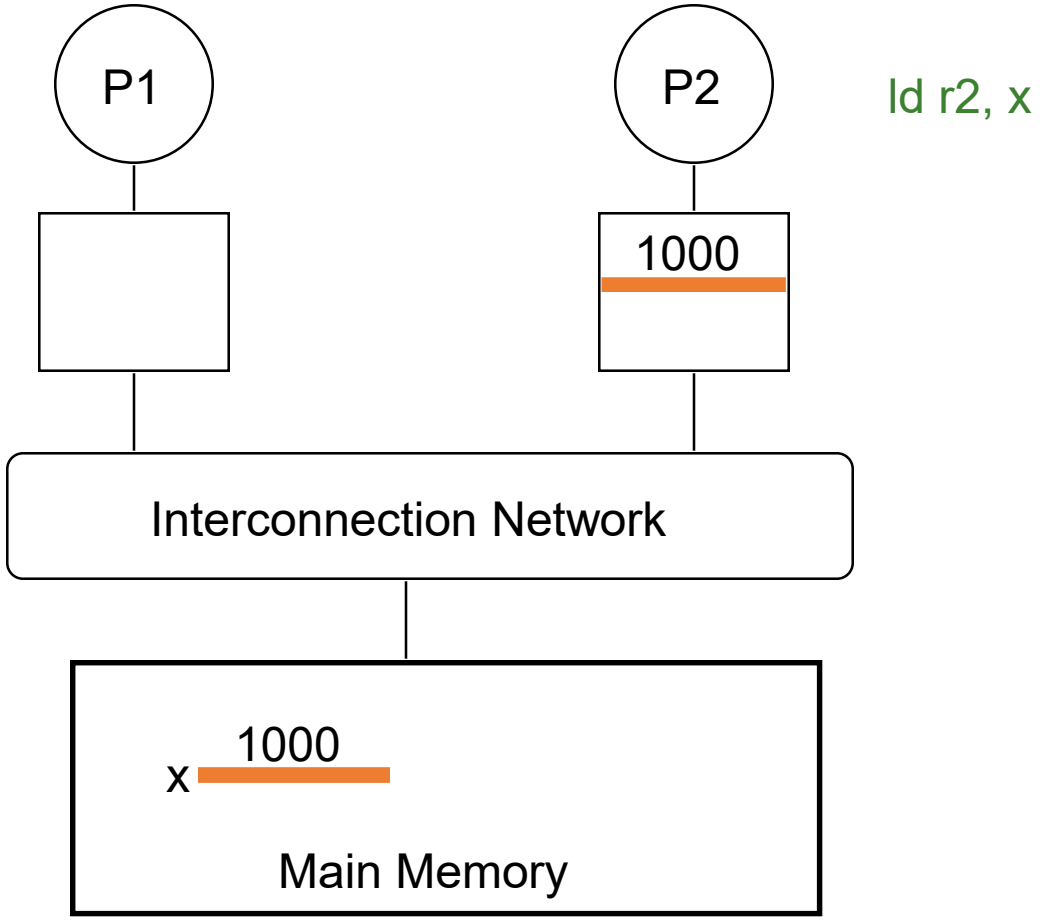
Memory Hierarchy in Modern Servers



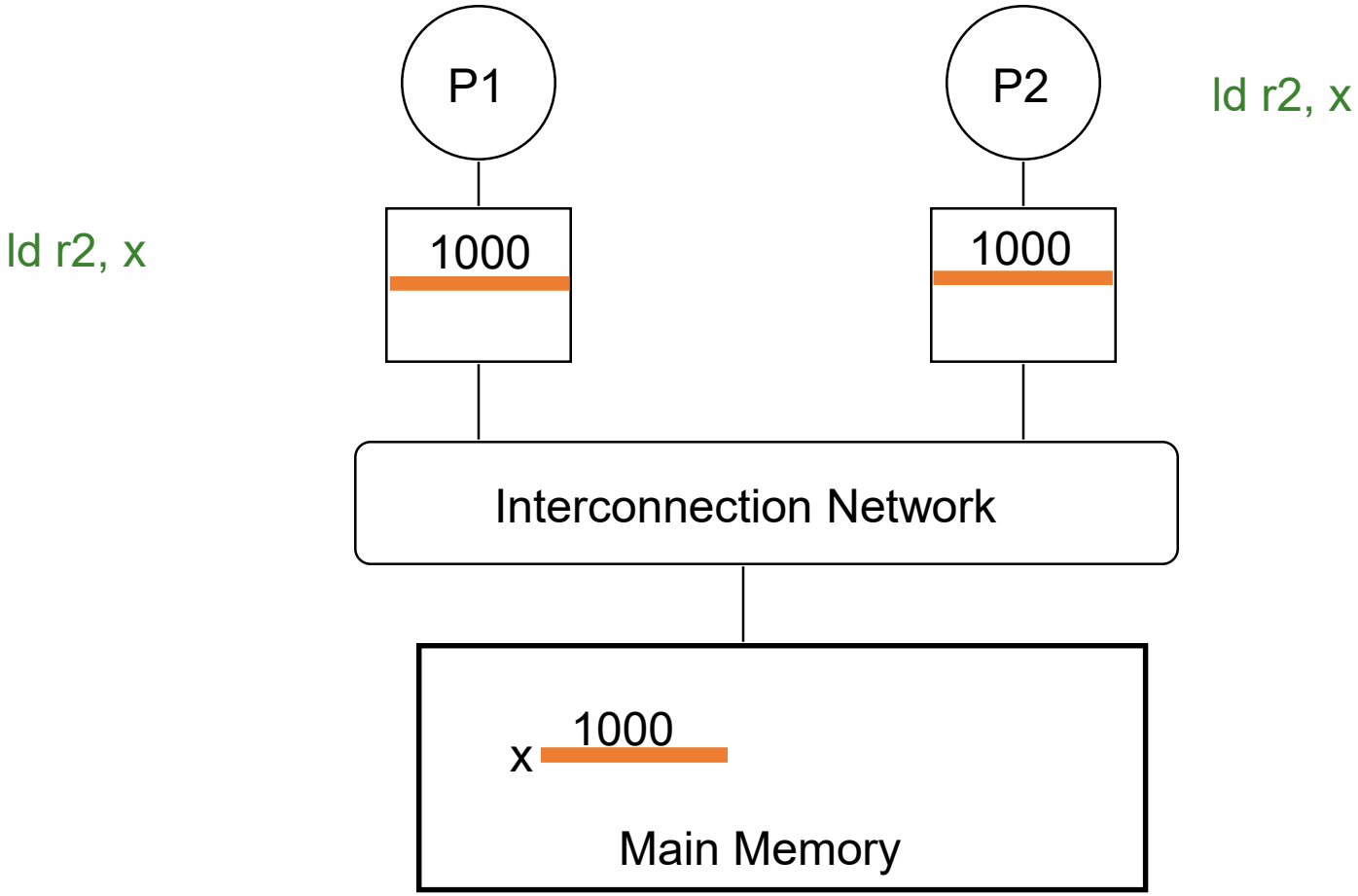
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



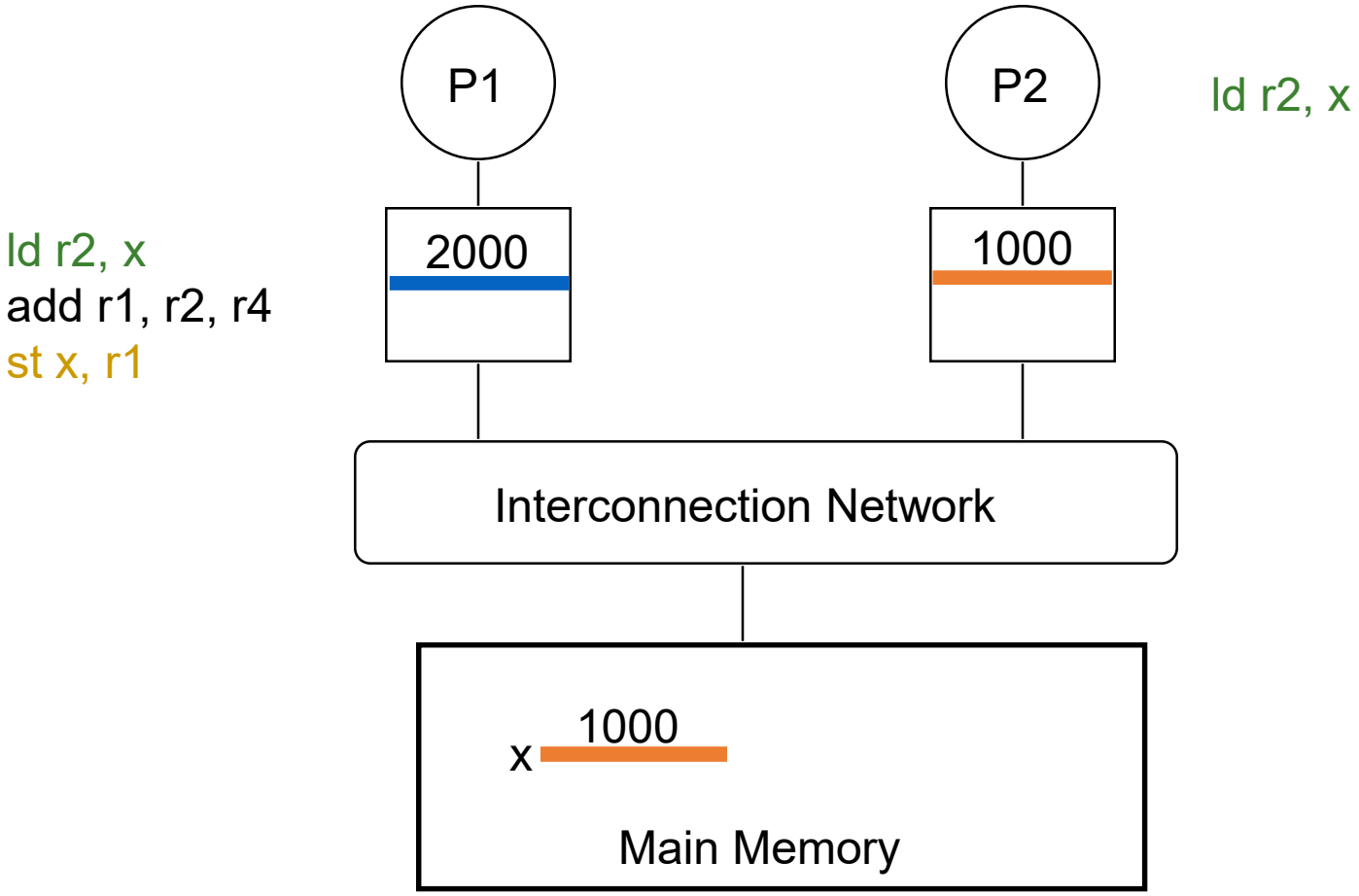
The Cache Coherence Problem



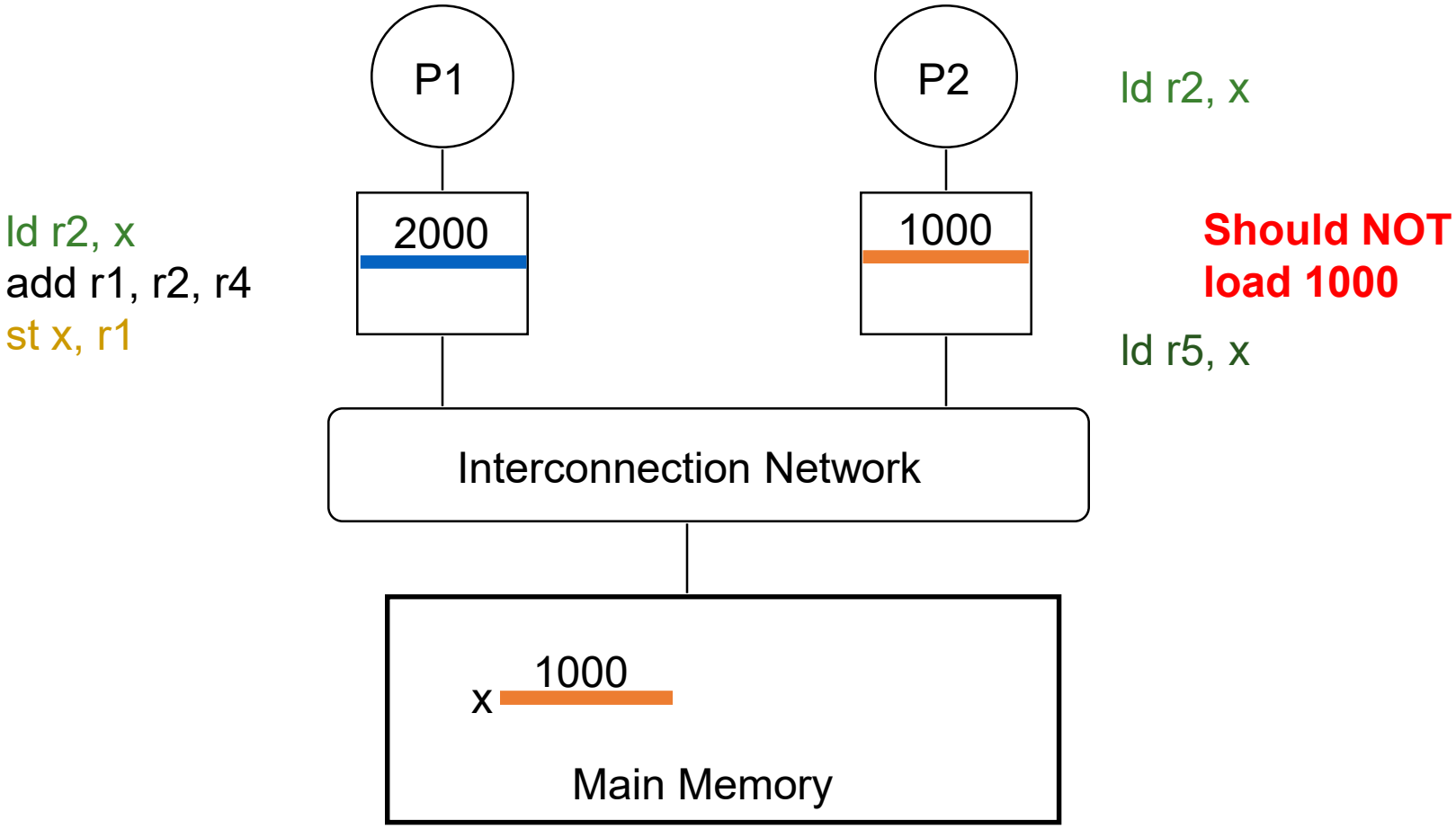
The Cache Coherence Problem



The Cache Coherence Problem



The Cache Coherence Problem



Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate
  - **Write serialization**: provide a consistent global order seen by all processors

- Need a global point of serialization for this store ordering

HARDWARE CACHE COHERENCE

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy

COHERENCE: UPDATE VS. INVALIDATE

- How can we *safely update replicated data?*
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy isn't valid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)

COHERENCE: UPDATE VS. INVALIDATE (II)

- **On a Write:**
  - Read block into cache as before

Update Protocol:

- Write to block, and simultaneously broadcast written data to sharers
- (Other nodes update their caches if data was present)

Invalidate Protocol:

- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes clear block from cache)

UPDATE VS. INVALIDATE TRADEOFFS

- Which do we want?
  - Write frequency and sharing behavior are critical
- **Update**
  - \+ If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
  - – If data is rewritten without intervening reads by other cores, updates were useless
  - – Write-through cache policy ➔ bus becomes bottleneck
- **Invalidate**
  - \+ After invalidation broadcast, core has exclusive access rights
  - \+ Only cores that keep reading after each write retain a copy
  - – If write contention is high, leads to ping-ponging (rapid mutual invalidation-reacquire)

Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, single point of serialization for all requests
  - Processors observe other processors' actions
    - E.g.: P1 makes "read-exclusive" request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1