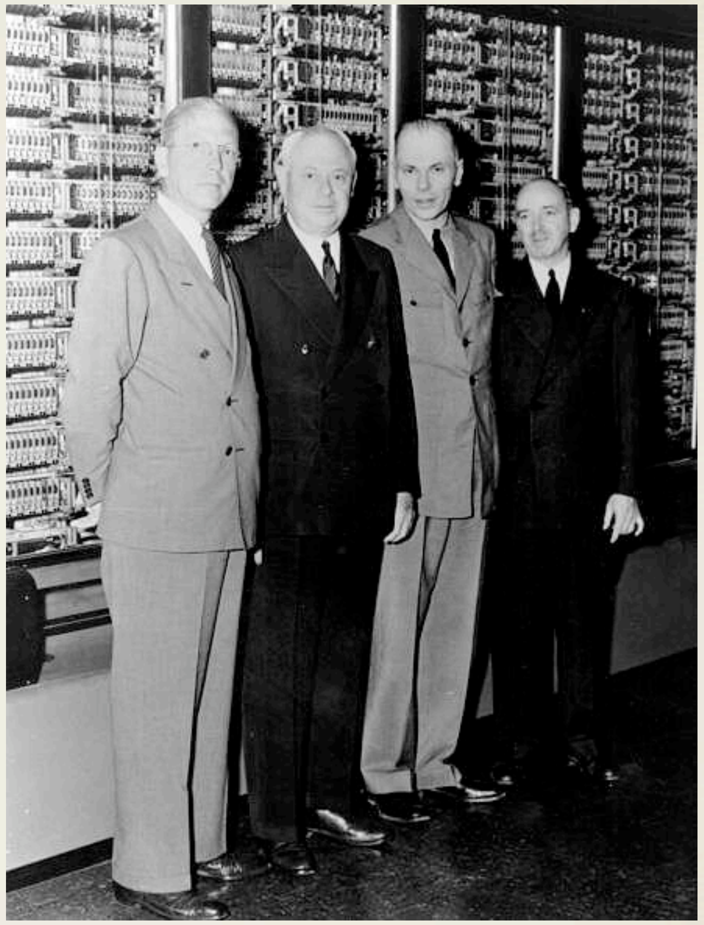
VON NEUMANN COMPUTING

Past and Future

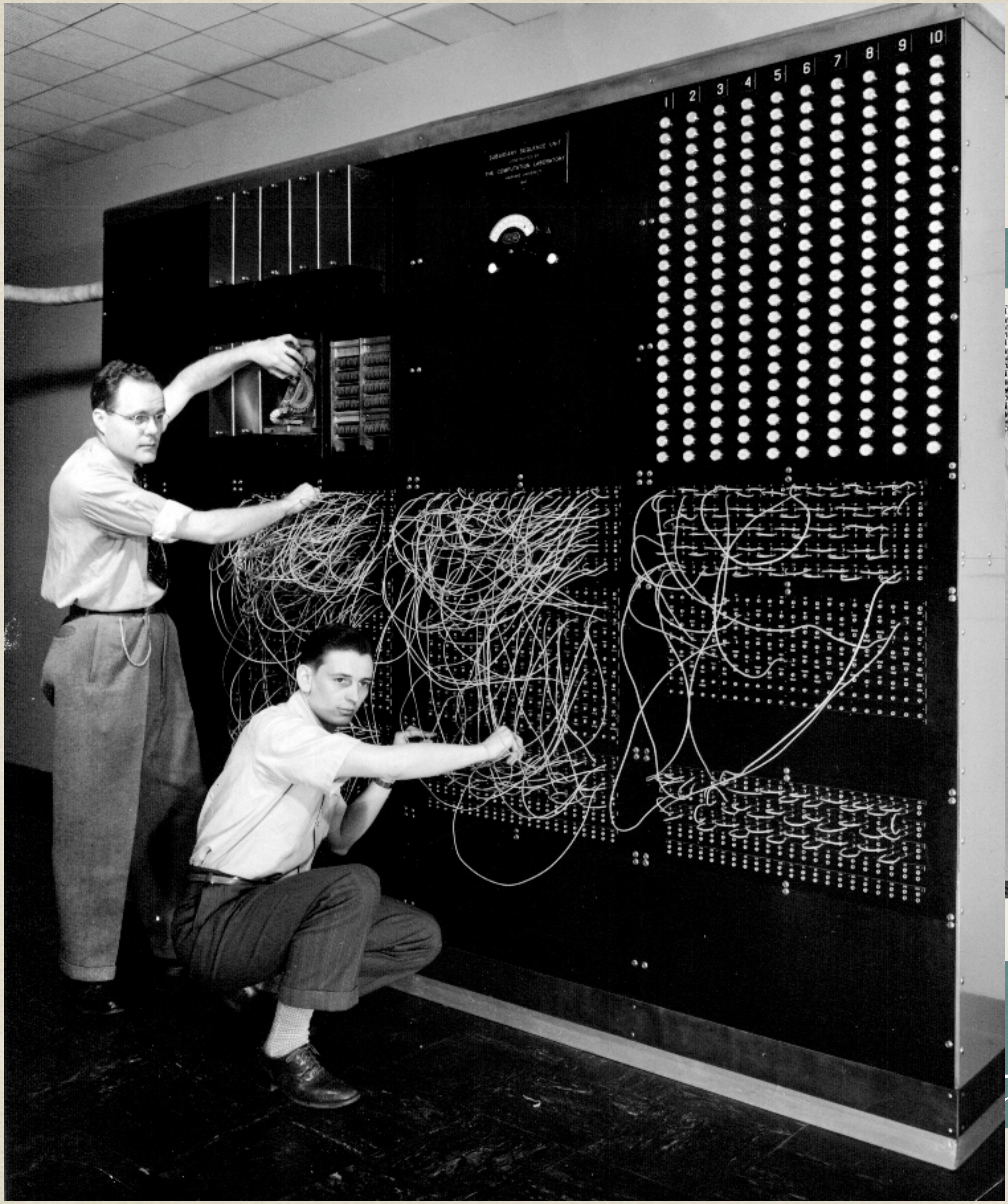
What is not VN

Think of this as an ASIC



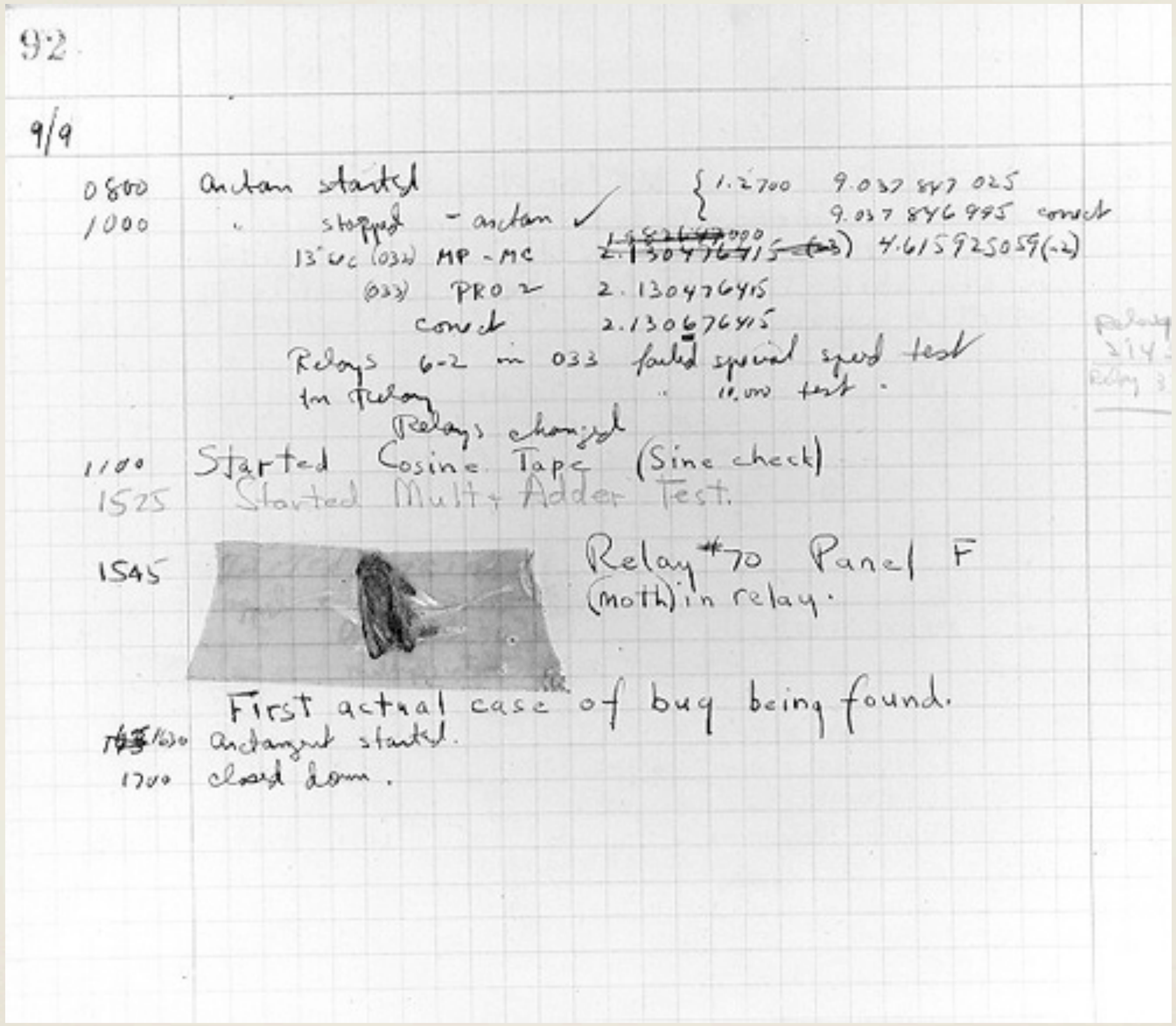
1944, Harvard Mark I: hard-wired calculator for ballistic firing calculations

VN



d calculator
tions

The first bug



92

9/9

0800 Anctan started {1.2700 9.037 847 025
1000 " stopped - anctan ✓ 9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 11.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

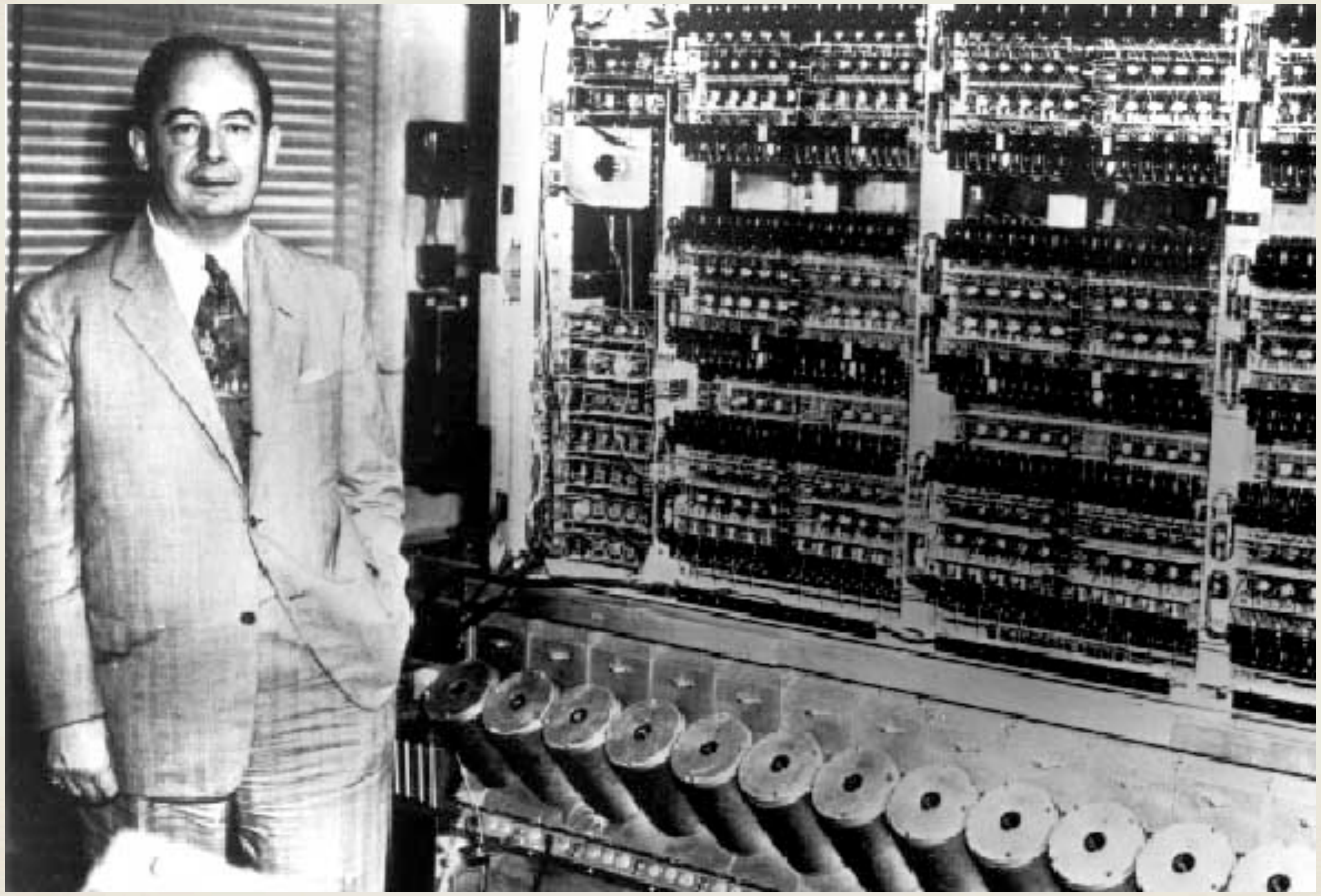
Relay 2145
Relay 3

From Grace Hopper's notebook, 1945

What do we mean by VN computing?

- Stored program / data
- Fetch / Execute
  - Conditional branch

VN



1942, ENIAC: 18K vacuum tubes, 180Kw power, base-10 arithmetic, clock-cycle ~ 5KHz

A VN Execution Algorithm

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

Topics for the next two hours

- How do you make VN execution efficient?
  - exploit locality, parallelism, predictability
- What constrains VN execution?
  - limits on locality, parallelism, and predictability
- Where do we go in a post-VN world?
  - Multiprocessors

VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

To save on cost, memory is slow

VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

To save on cost, memory is slow

This interface is too general

VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

To save on cost, memory is slow

This interface is too general

This process is sequential!

VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

To save on cost, memory is slow

Caches: Making memory appear dense *and* fast

Why is it slow? - *The Memory Wall*

Memory technologies:
1T memory cell: DRAM
6T memory cell: SRAM
12-15T memory cell: FF

others:
3,4,5T SRAM-ish cells

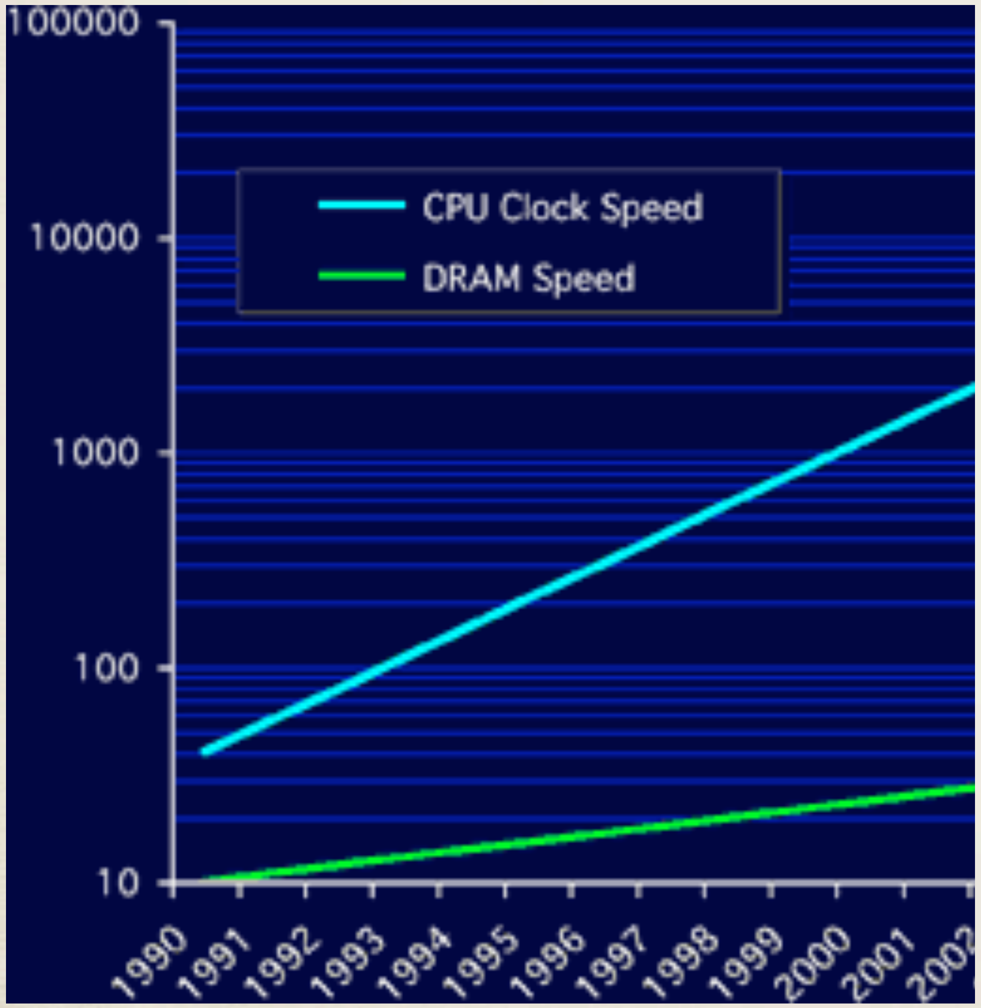
Dense
Slow

Sparse
Fast

Memory/CPU communication technologies

- On-chip
  - Pro: Fast, wide, “controllable”
  - Con: Limited
- Off-chip wide:
  - Pro: well understood, lots of pins for communication
  - Con: hard to keep in sync, thus slow per-pin speed
- Off-chip narrow:
  - Pro: less pins (pins are expensive), high per-pin speed
  - Con: more difficult to engineer, IP heavy landscape

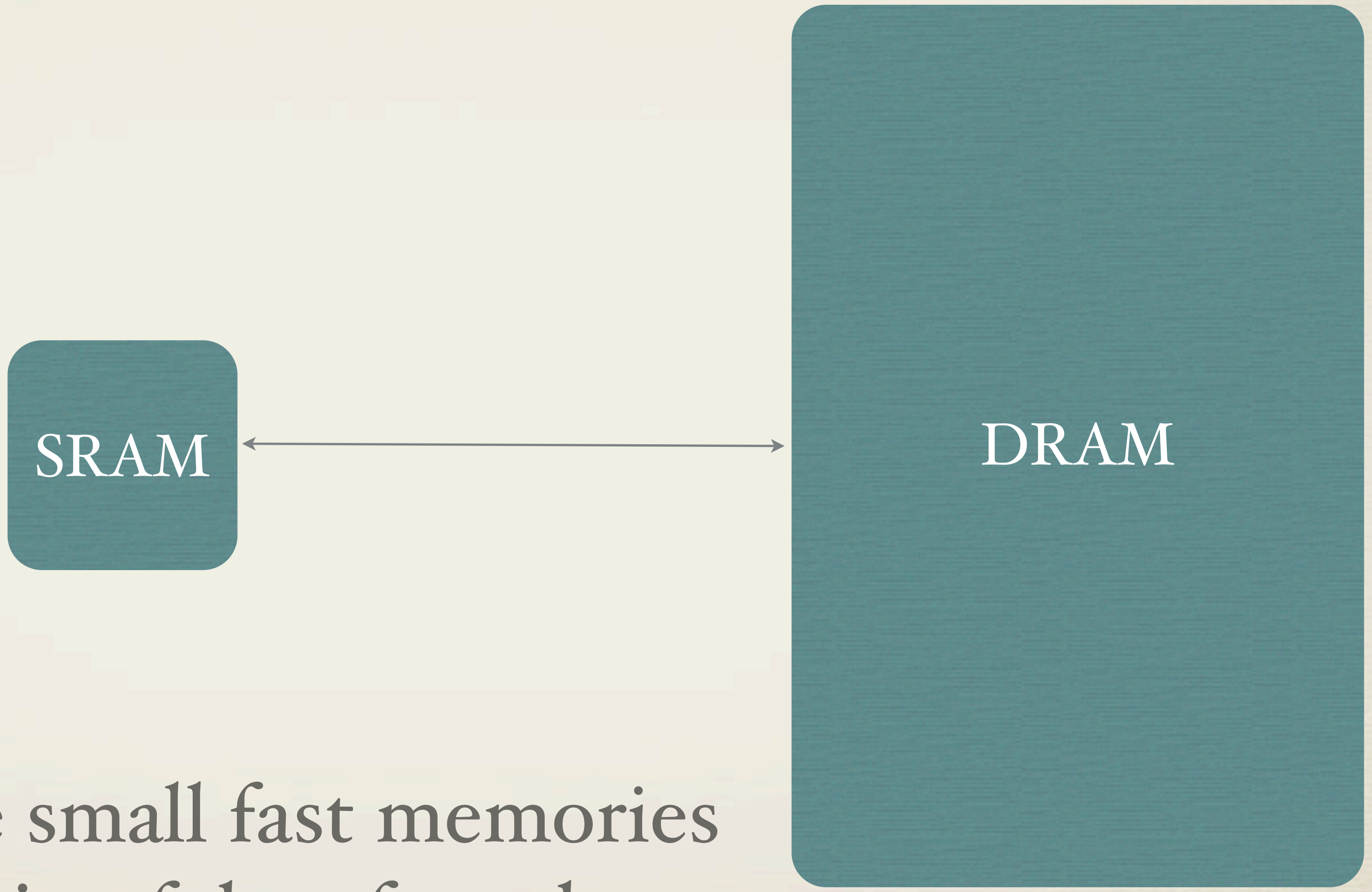
The memory wall



Historically, processor speed increased 60%/year

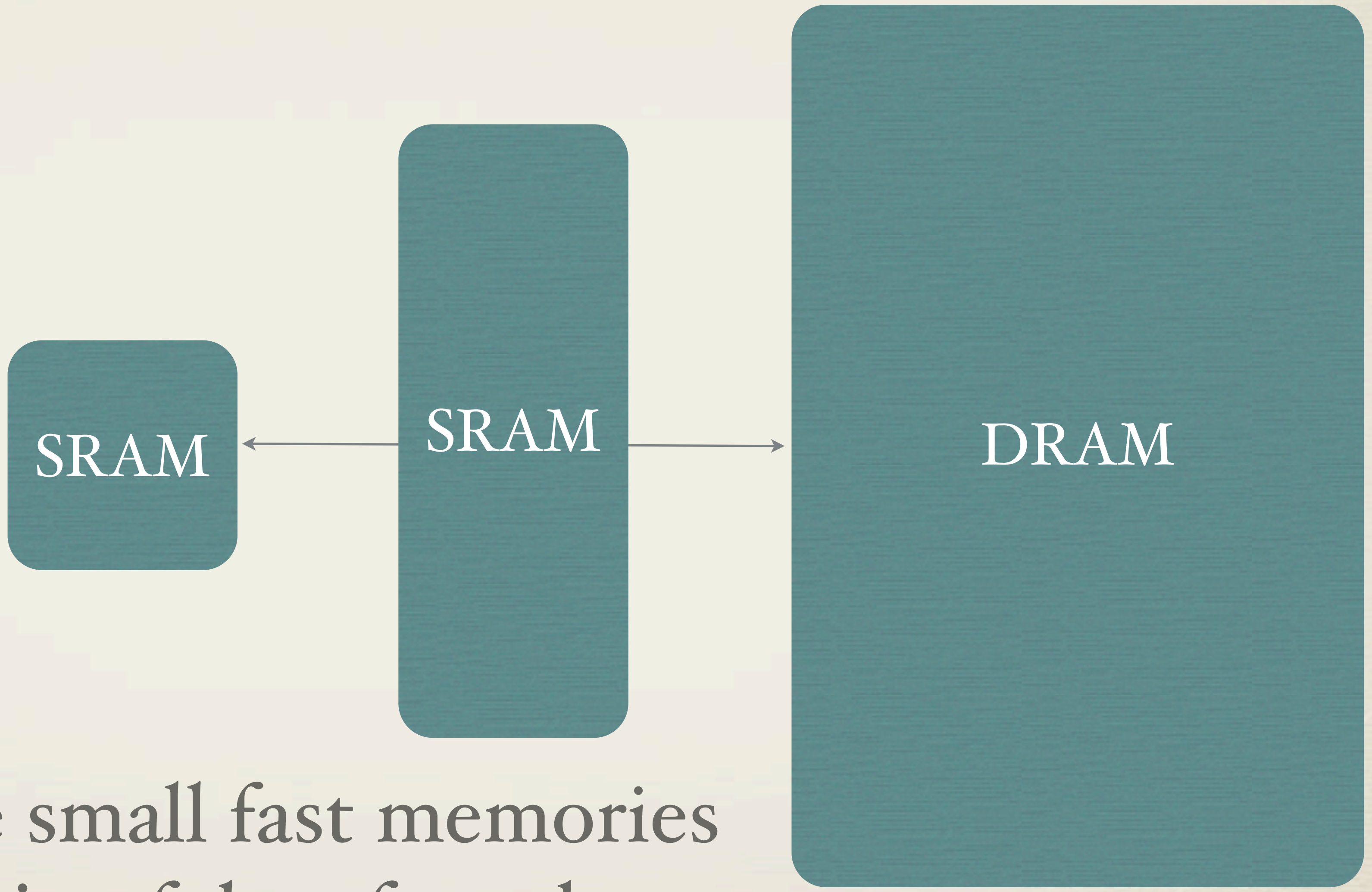
Memory "speed" ~ 9%

Favored solution: caching



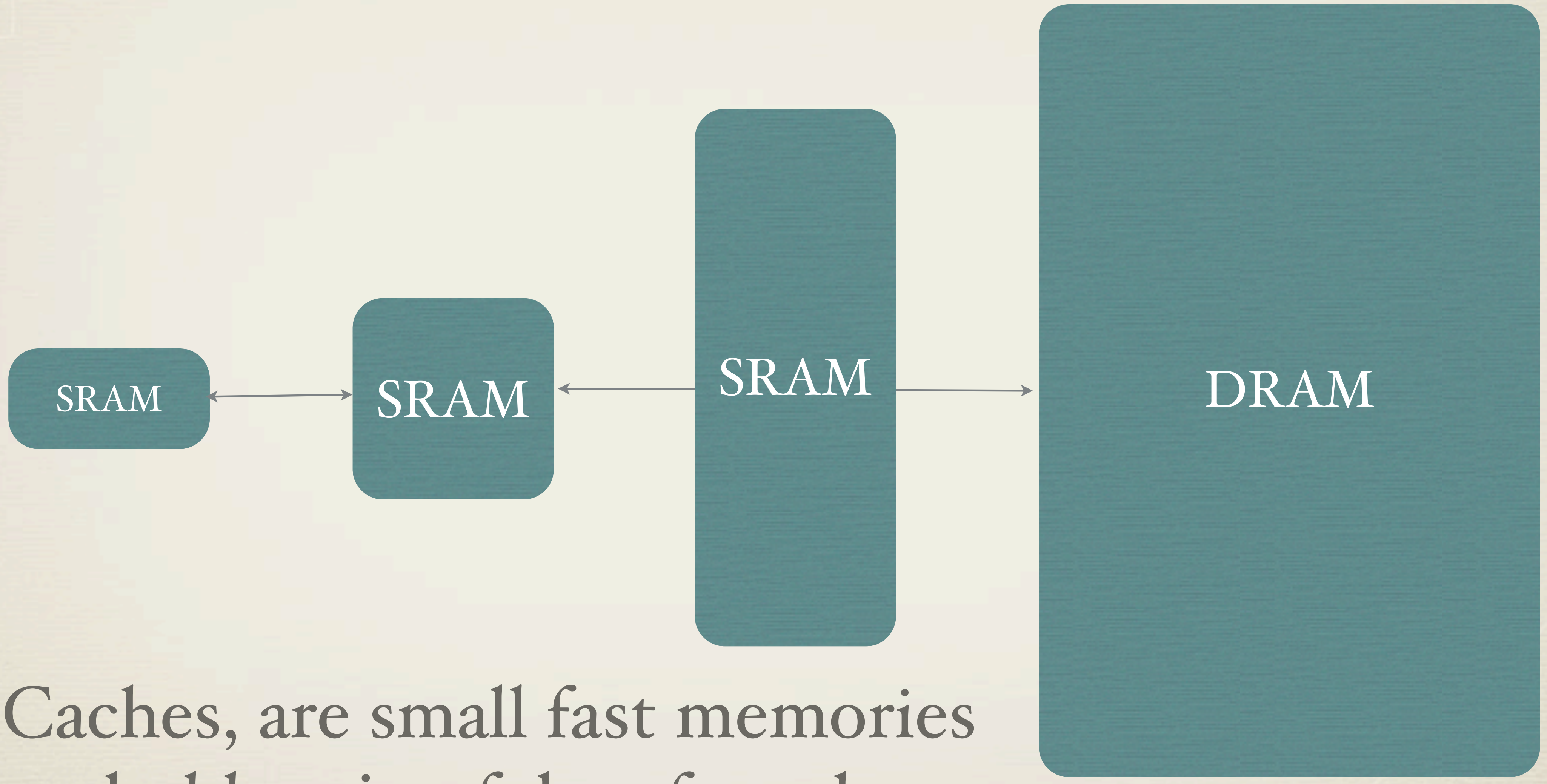
Caches, are small fast memories that hold *copies* of data from larger, slow devices

Favored solution: caching



Caches, are small fast memories that hold *copies* of data from larger, slow devices

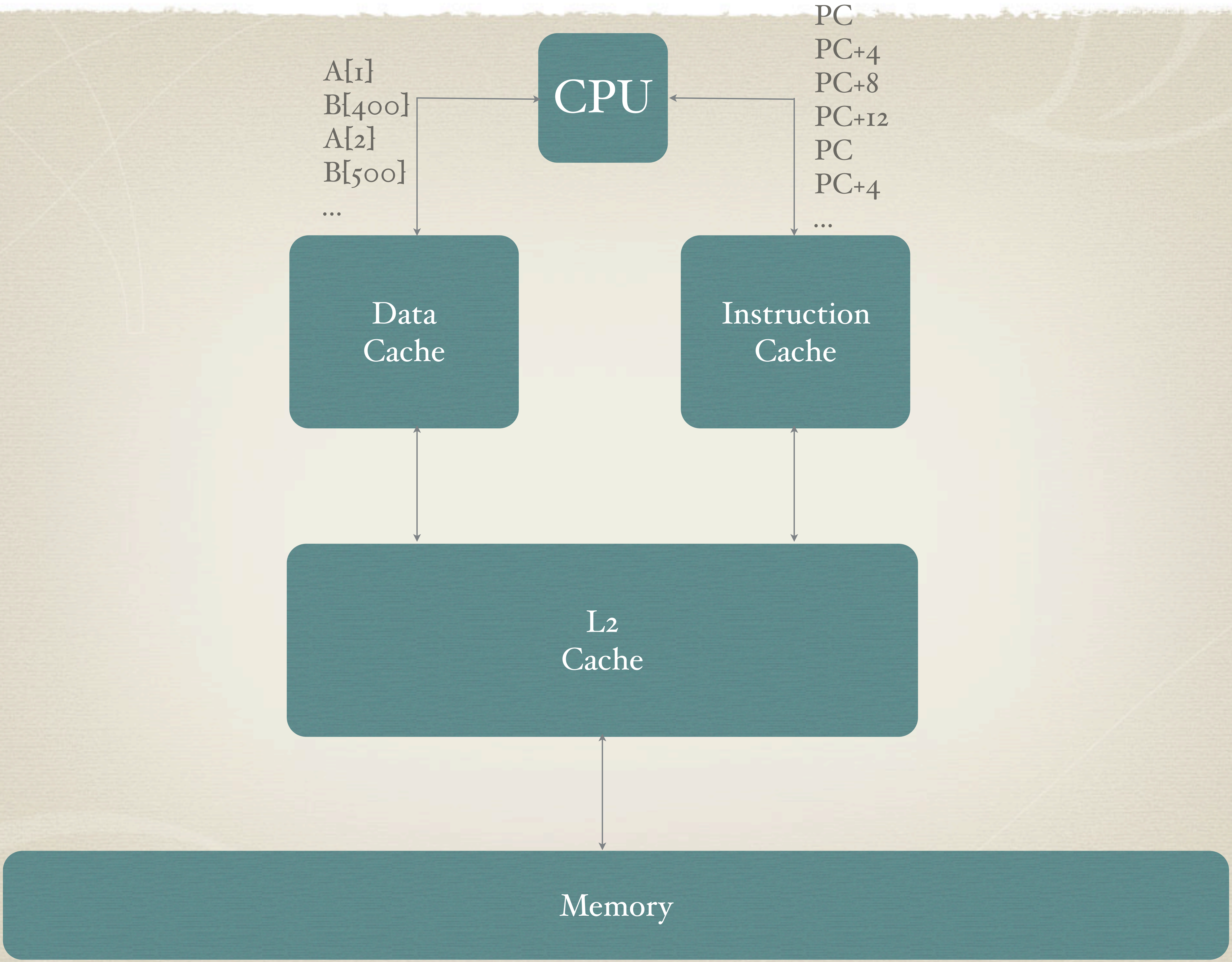
Favored solution: caching



Caches, are small fast memories that hold *copies* of data from larger, slow devices

What goes in a cache?

- Items that are accessed, and items *around* those just accessed
- Why does this work?
  - Temporal locality: *we'll likely see this thing again*
  - Spatial locality: *we'll likely need something nearby*


A[1]
B[400]
A[2]
B[500]
...
CPU
PC
PC+4
PC+8
PC+12
PC
PC+4
...
Data Cache
Instruction Cache
L2 Cache
Memory

What limits caching?

- The three C's: Capacity, Conflicts, and Cold misses
  - Capacity: need more cache
  - Conflicts: need different cache geometry
  - Cold: need to guess what to put into the cache before it is requested

What limits caching

- Cache size: ultimately, a larger cache is a slower cache (wire delay)
- Conflict misses just happen: even the best hash functions in the world collide on something
- Cold misses can't be totally eliminated: A [B[n]]

Summary: memory

- People need a dense fast storage technology
  - It doesn't exist. Thus, we try and approximate it with a caching system
- Caching is highly effective for small-footprint applications (read: not databases!)
- Caching has its limitations: inherent non-locality left in instruction and data streams

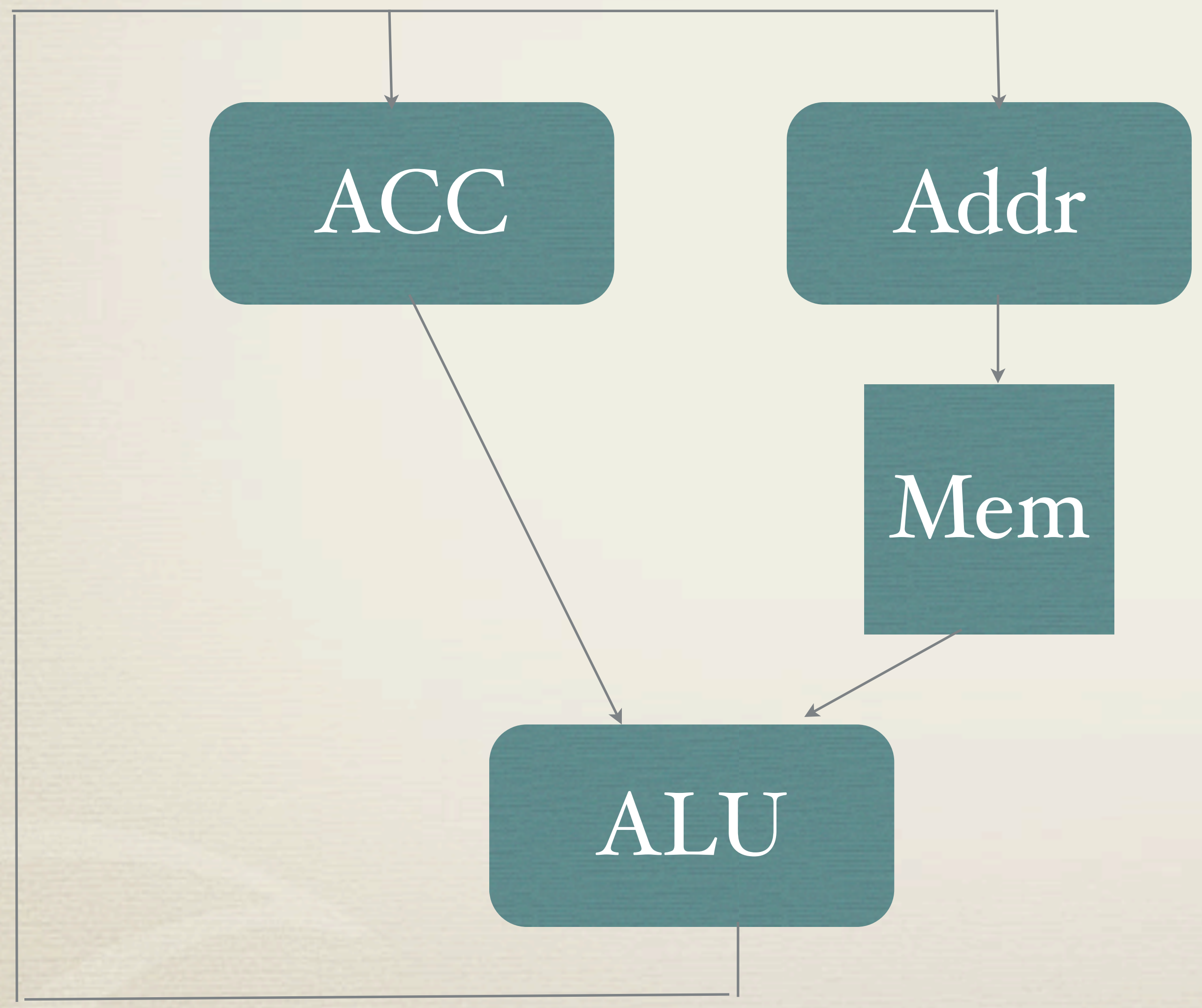
VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

This interface is too general

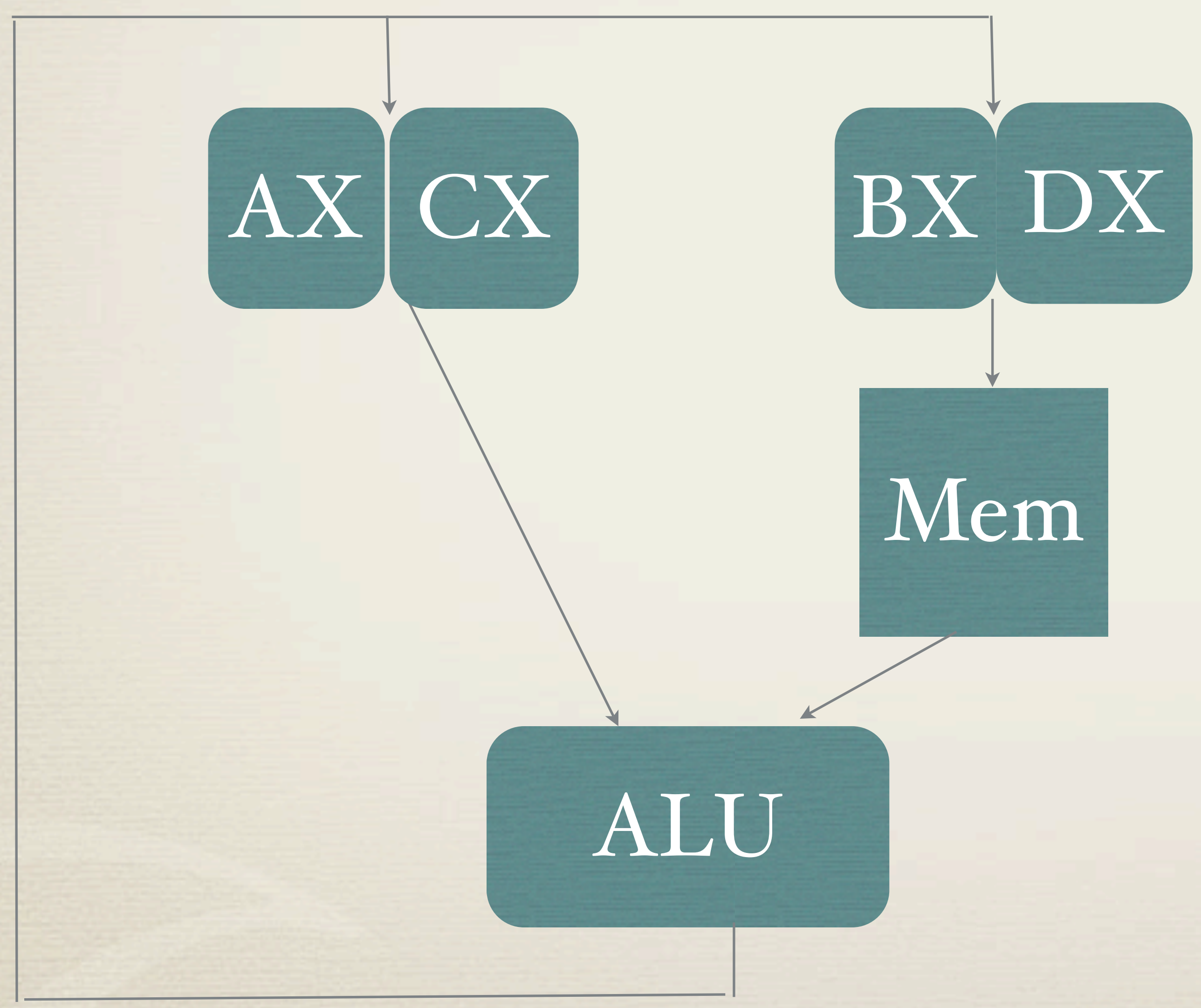
In the beginning...

In the beginning there was the accumulator



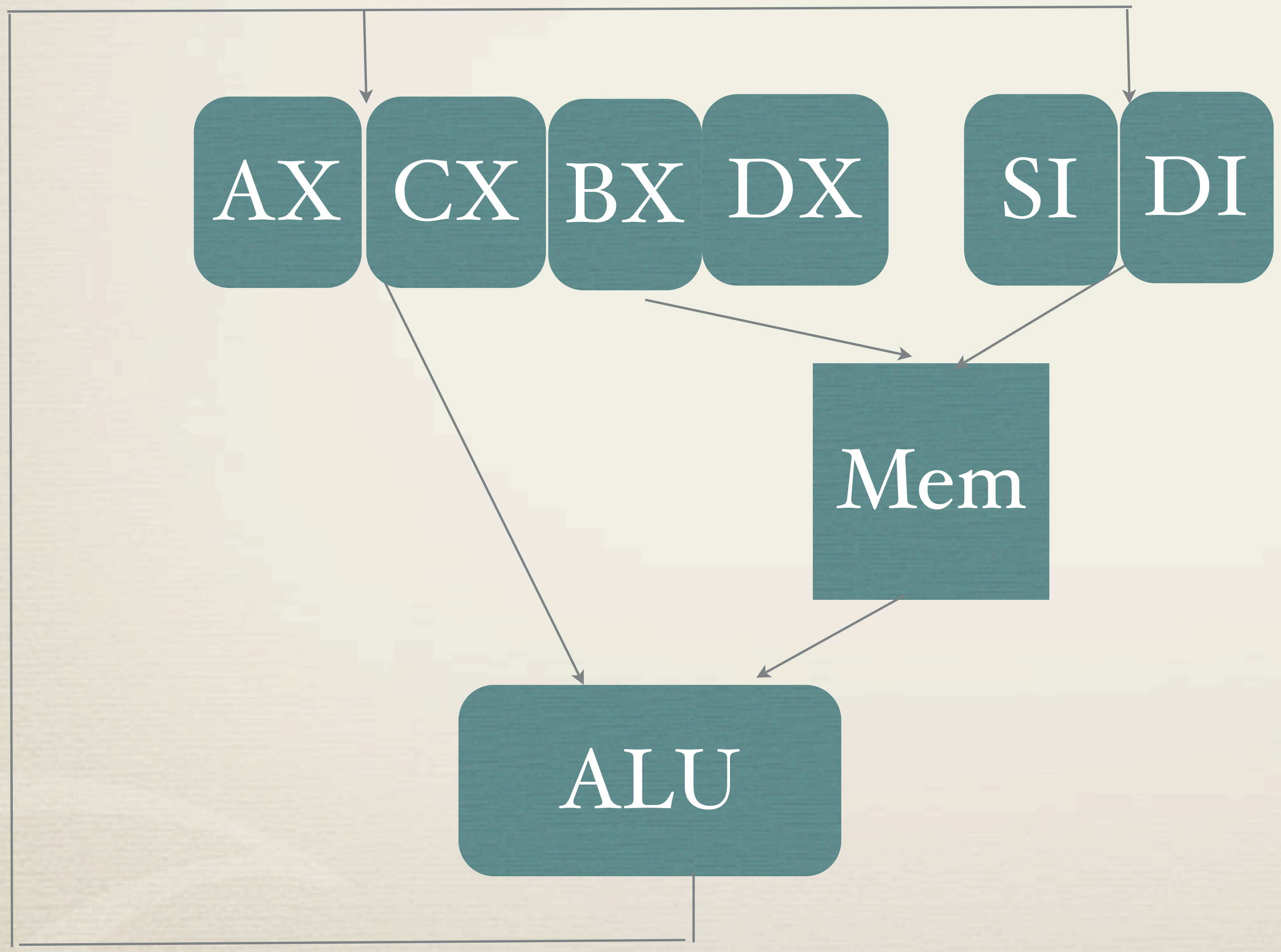
ADD
  ; ACC <- ACC + M[Addr]
MOVE-TO-ADDR
  ; Addr = Acc
MEM-TO-ACC
  ; Acc = M[Addr]
CONSTANT-TO-ADDR #
  ; Addr = #
ACC-TO-MEM
  ; M[Addr] = Acc

Then they thought two would be nice



And maybe a few more

And some generality



CISC

- Complex Instruction Set Computing
- What were they thinking?
  - Assembly programmers would like a rich, expressive instruction set. Something that would let them type:
    - ADD R1, R2, R3
    - ADD A, B, C
    - etc

CISC

- In reality, almost all code is written in a high-level language.
- Compilers do a poor job of assigning instructions to expressions => no wonder, its NP hard!
- Complex instructions are rarely used, yet need to be supported for legacy binary compatibility => bear to support!

RISC

- Reduced instruction set architecture
- Few basic operands: add, nand, xor, load, store
- Many general-purpose registers with no special-case semantics => semantics can be done in software only
- In the extreme, expose complexities of hardware to software: branch delay slots, load interlocks

CISC or RISC?

- Many academics will argue RISC is better
  - Simpler hardware => easier to build
  - Simpler hardware => tune for speed
- Truth is, people don’t buy processors for their elegance
- CISC can, with effort, be made just as fast as RISC
  - Binary compatibility matters

Summary: execution interface

- Constrain operation semantics to simplify and speedup software
- Register files: software managed caches
- RISC vs. CISC: the market decides the winner, not the technology

VN execution inefficiencies

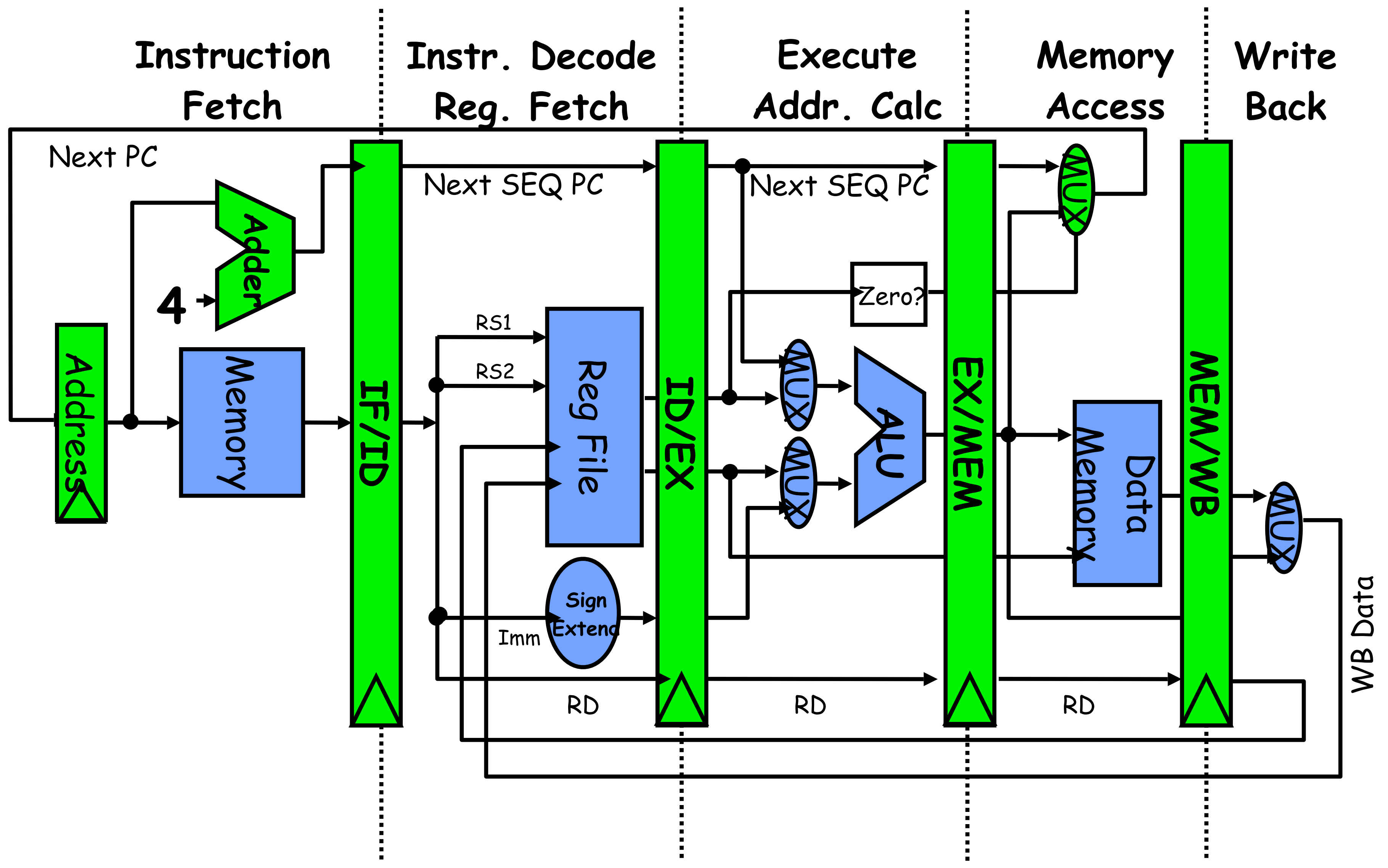
```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

This process is sequential!

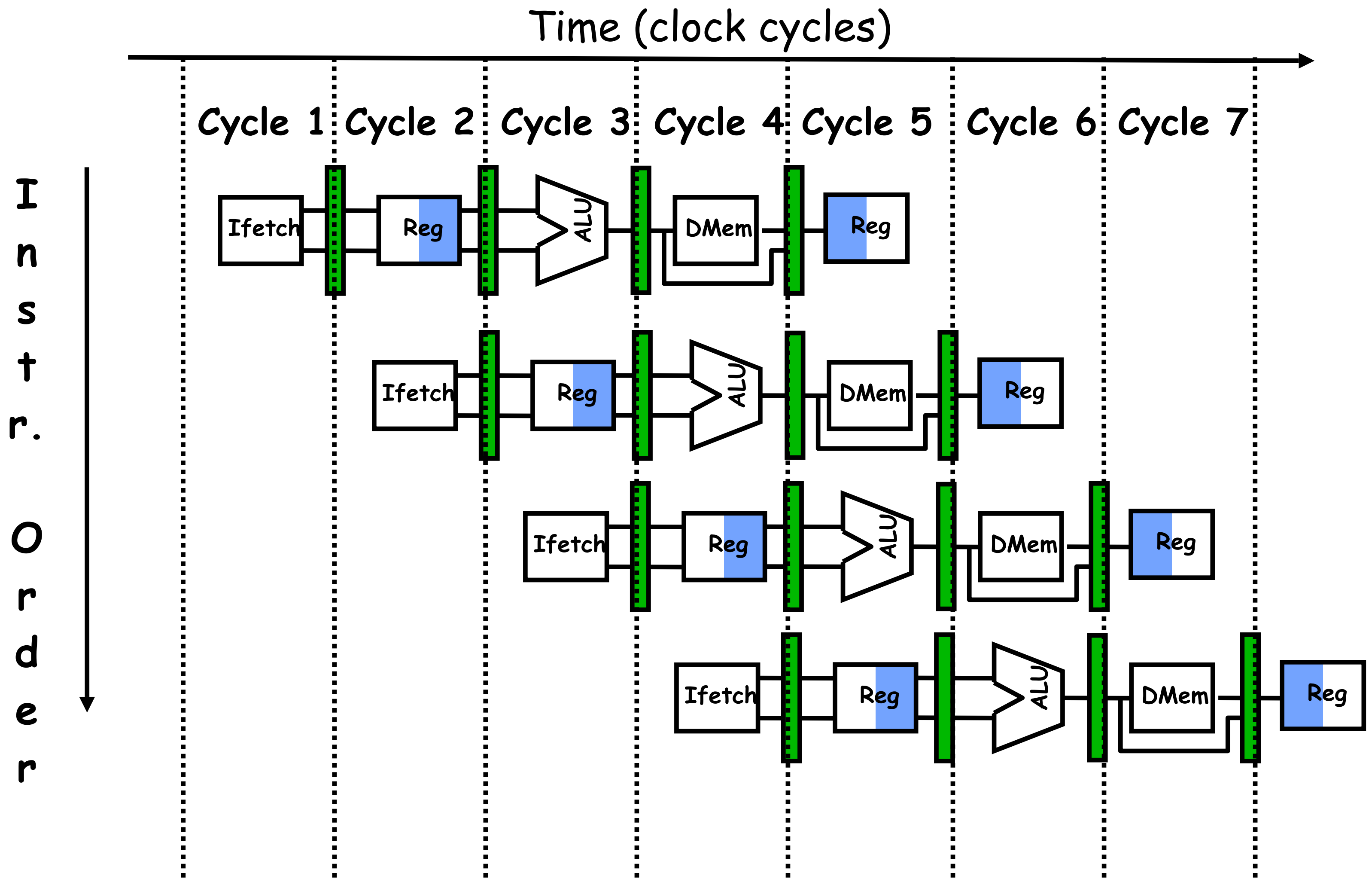
Instruction-Level Parallelism (ILP)

- Fine-grained parallelism
- Obtained by:
    - instruction overlap in a pipeline
    - executing instructions in parallel (later, with multiple instruction issue)
- In contrast to:
    - loop-level parallelism (medium-grained)
    - process-level or task-level or thread-level parallelism (coarse-grained)

Classic 5-Stage MIPS



Pipelining


Time (clock cycles)
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Instr. Order
Ifetch
Reg
ALU
DMem
Reg

Pipelining

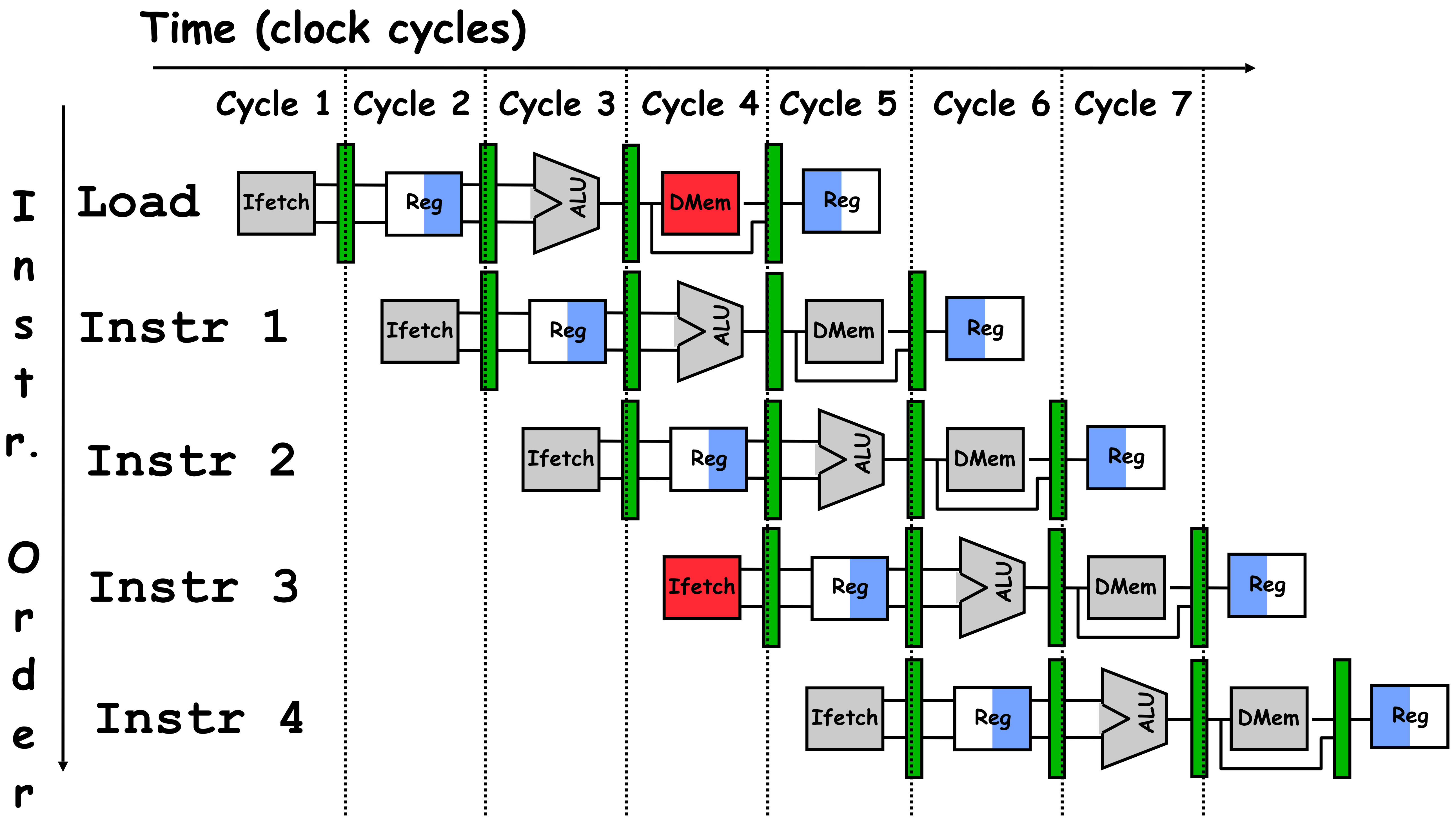
- Not that simple!
  - pipeline hazards (structural, data, control)
    - place a soft “limit” on the number of stages
  - increase instruction latency (a little)
    - write & read pipeline registers for data that is computed in a stage
    - time for clock & control lines to reach all stages
    - all stages are the same length which is determined by the longest stage
    - stage length determines clock cycle time

- IBM Stretch (1961): the first general-purpose pipelined computer

Hazards

- Structural hazards
- Data hazards
- Control hazards
- What happens on a hazard
  - instruction that caused the hazard & previous instructions complete
  - all subsequent instructions stall until the hazard is removed (in-order execution)
  - instructions that depend on that instruction stall (out-of-order execution)

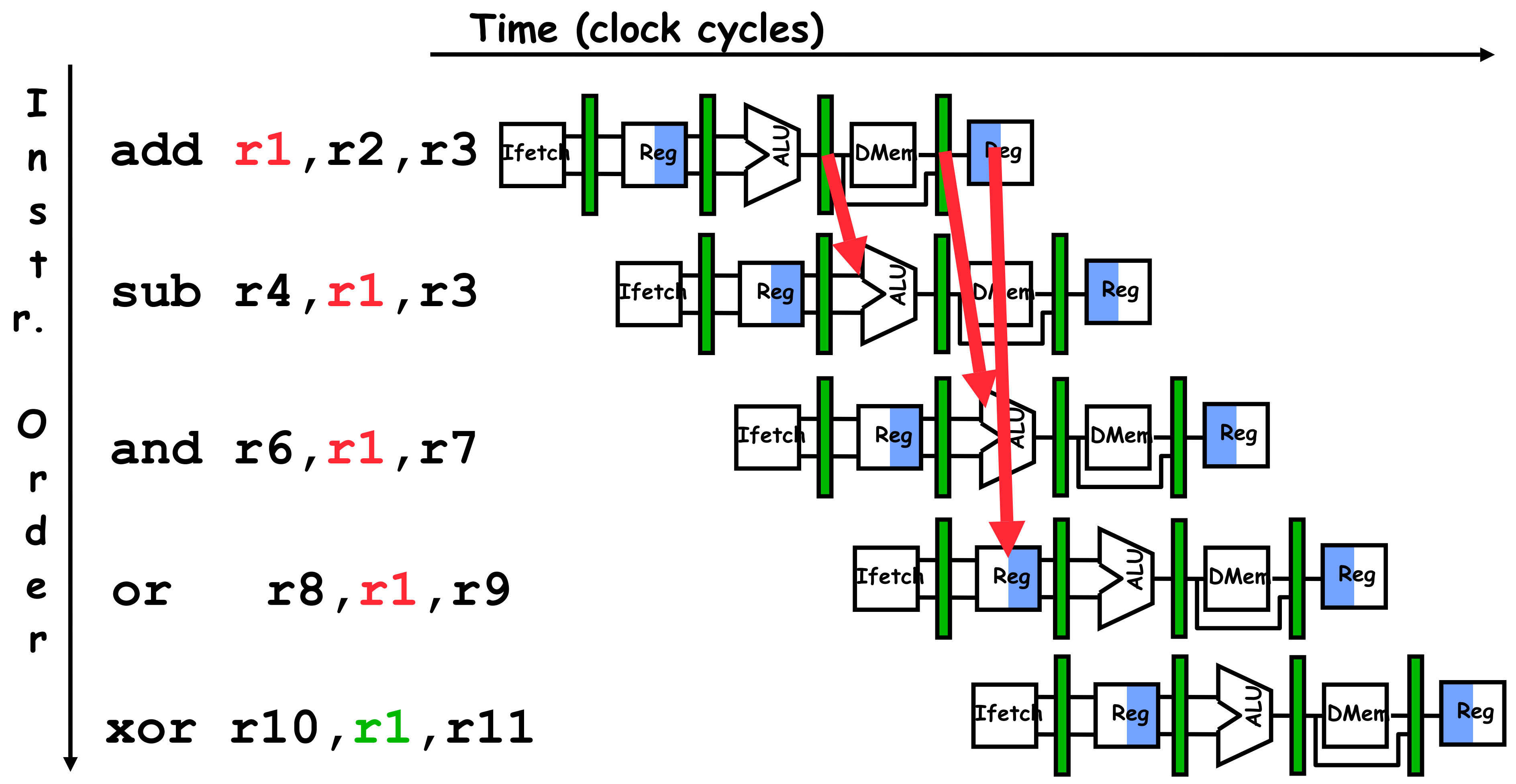
One Memory Port/Structural Hazards



Data Hazards

- Cause:
  - an instruction early in the pipeline needs the result produced by an instruction farther down the pipeline before it is written to a register
  - would not have occurred if the implementation was not pipelined
- Types
  - RAW (data: flow), WAR (name: antidependence), WAW (name: output)
- HW solutions
  - forwarding hardware (eliminate the hazard)
  - stall via pipelined interlocks
- Compiler solution
  - code scheduling (for loads)

Forwarding



Time (clock cycles)
I n s t r. O r d e r
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or r8,r1,r9
xor r10,r1,r11
Ifetch
Reg
ALU
DMem
Reg

Forwarding Implementation

- Forwarding unit checks to see if values must be forwarded:
    - between instructions in ID and EX
        - compare the R-type destination register number in EX/MEM pipeline register to each source register number in ID/EX
    - between instructions in ID and MEM
        - compare the R-type destination register number in MEM/WB to each source register number in ID/EX
- If a match, then forward the appropriate result values to an ALU source
    - bus a value from EX/MEM or MEM/WB to an ALU source

Control Hazards

- Cause: condition & target determined after next fetch
- Early HW solutions
  - stall
  - assume an outcome & flush pipeline if wrong
  - move branch resolution hardware forward in the pipeline
- Compiler solutions
  - code scheduling
  - static branch prediction
- Today's HW solutions
  - dynamic branch prediction

Pipeline Performance

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

Review: Types of Data Hazards

Consider executing a sequence of

$$r_k \leftarrow (r_i) \text{ op } (r_j)$$

type of instructions

Data-dependence

$$\begin{tikzcd}[column sep=small, row sep=small] r_3 \arrow[drr] & \leftarrow & (r_1) & \text{op} & (r_2) \\ r_5 & \leftarrow & (r_3) & \text{op} & (r_4) \end{tikzcd}$$

Read-after-Write (RAW) hazard

Anti-dependence

$$\begin{tikzcd}[column sep=small, row sep=small] r_3 & \leftarrow & (r_1) \arrow[dll] & \text{op} & (r_2) \\ r_1 & \leftarrow & (r_4) & \text{op} & (r_5) \end{tikzcd}$$

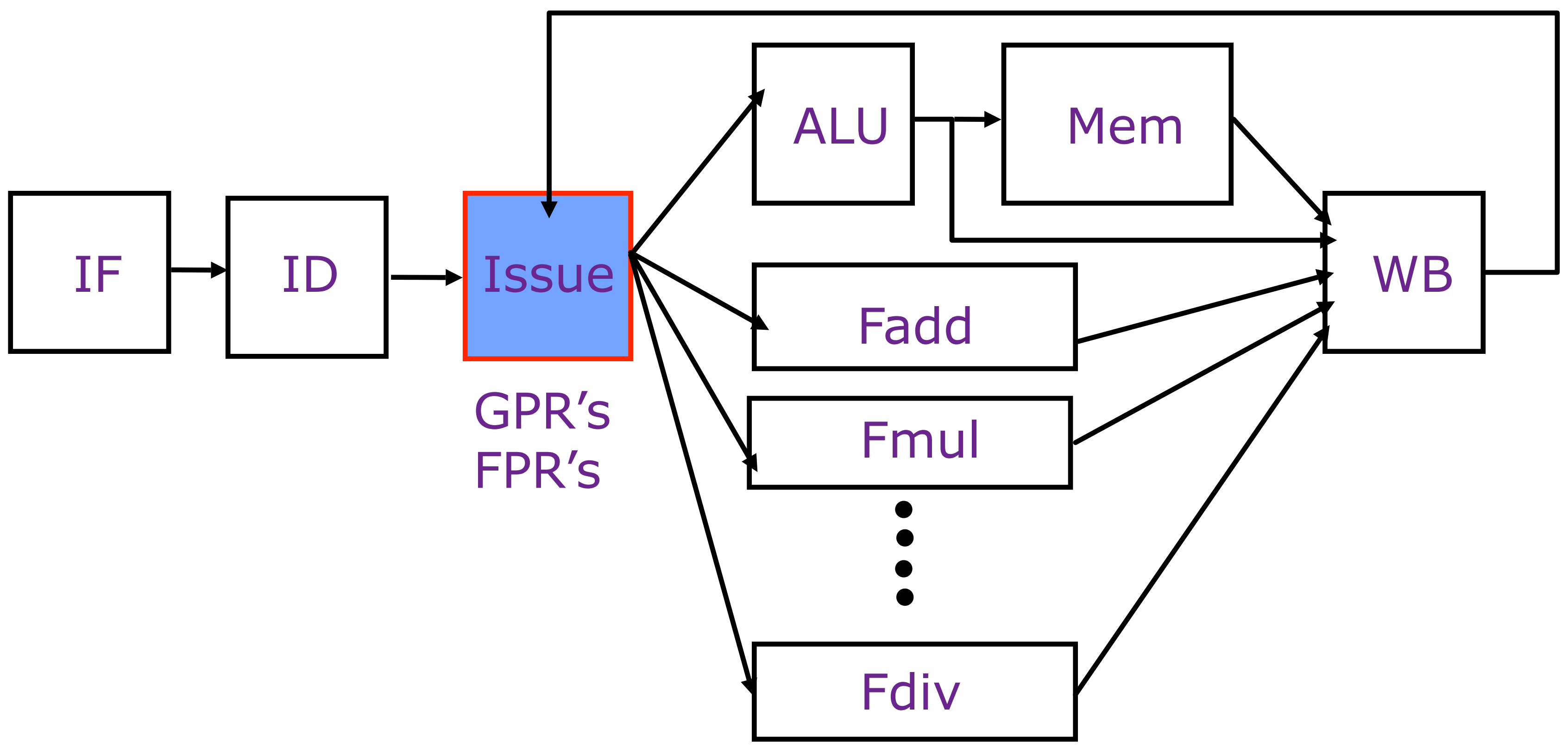
Write-after-Read (WAR) hazard

Output-dependence

$$\begin{tikzcd}[column sep=small, row sep=small] r_3 \arrow[d, bend right] & \leftarrow & (r_1) & \text{op} & (r_2) \\ r_3 & \leftarrow & (r_6) & \text{op} & (r_7) \end{tikzcd}$$

Write-after-Write (WAW) hazard

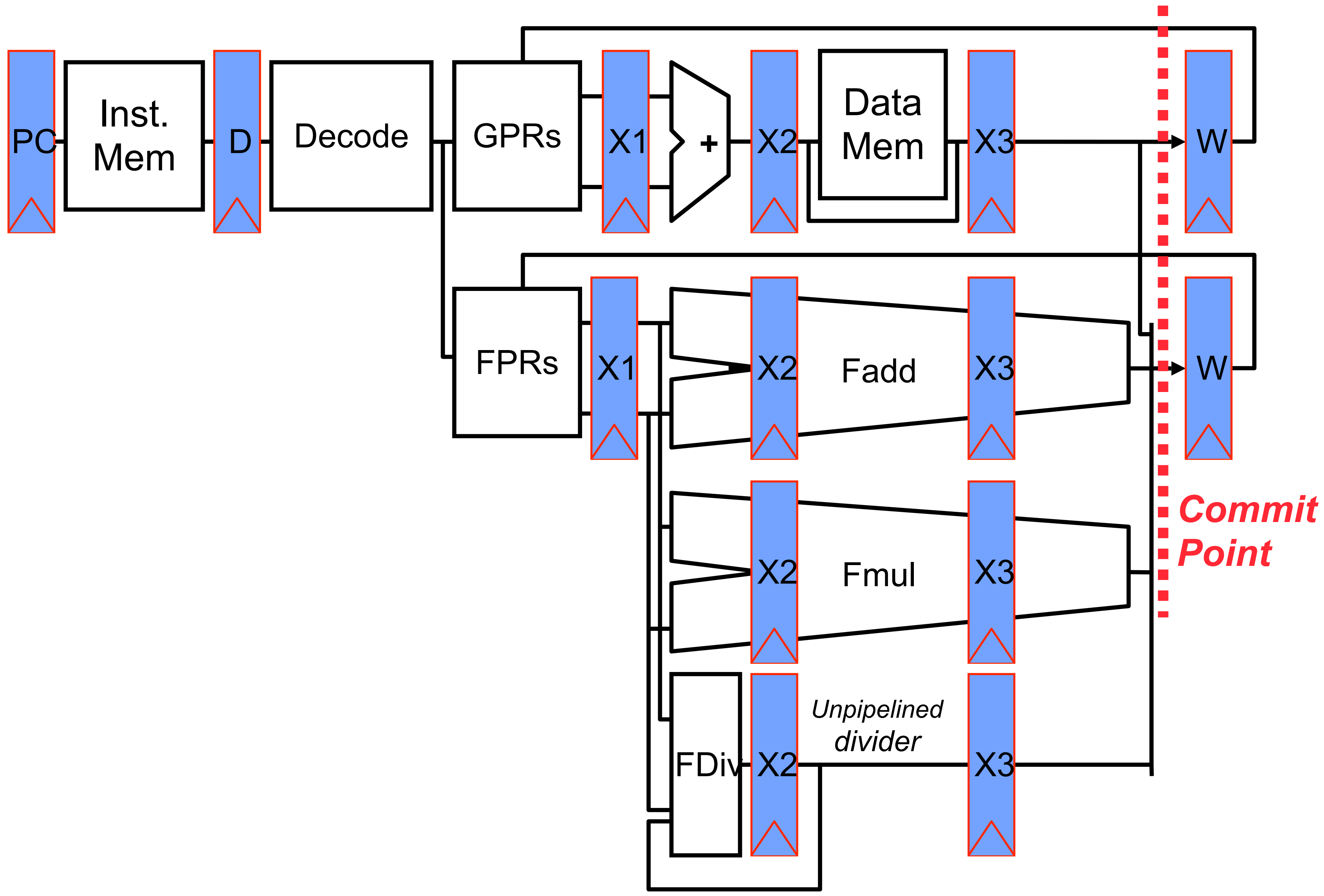
Complex Pipelining



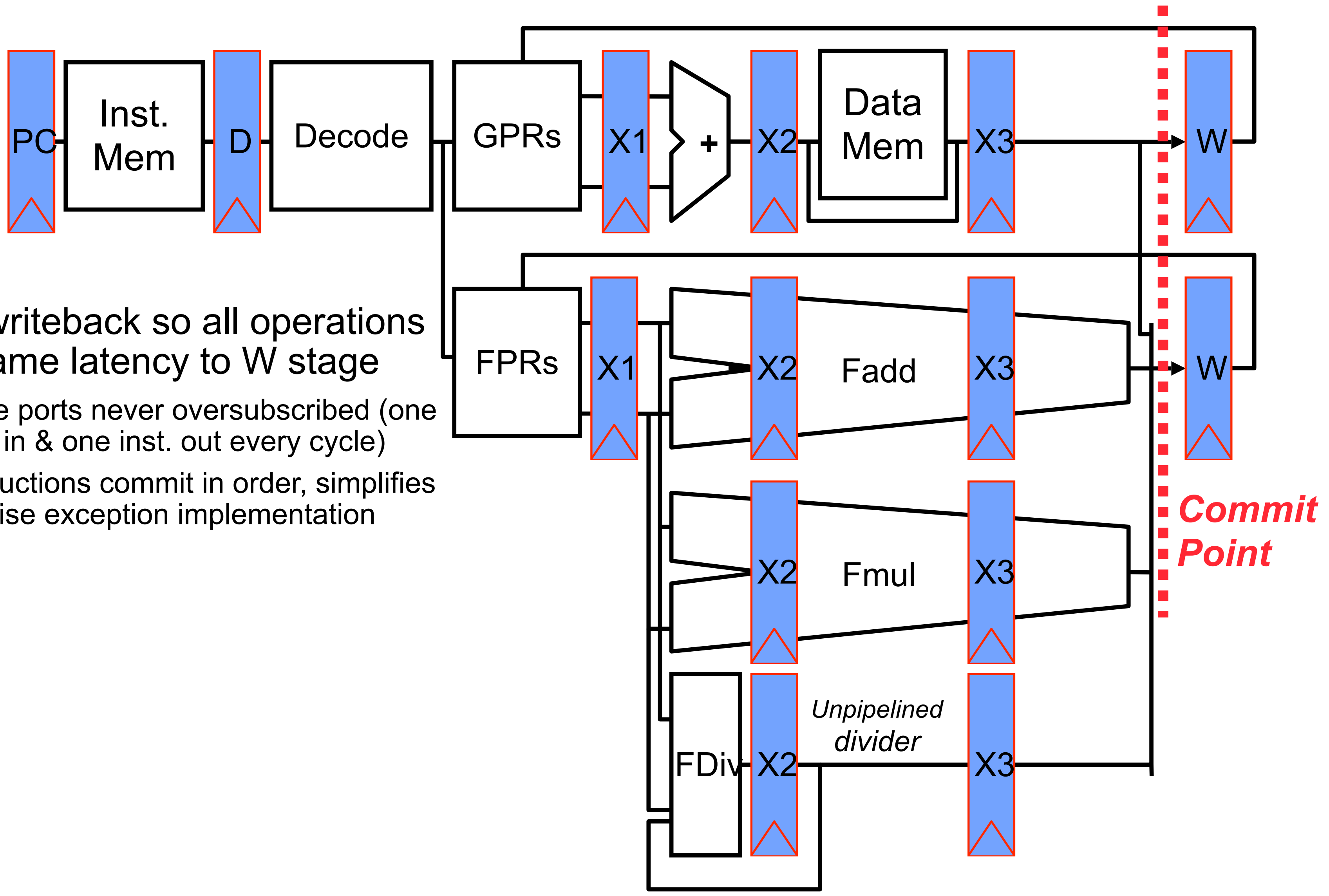
Pipelining becomes complex when we want high performance in the presence of:

- Long latency or partially pipelined floating-point units
- Multiple function and memory units
- Memory systems with variable access time
- Precise exceptions

Complex In-Order Pipeline

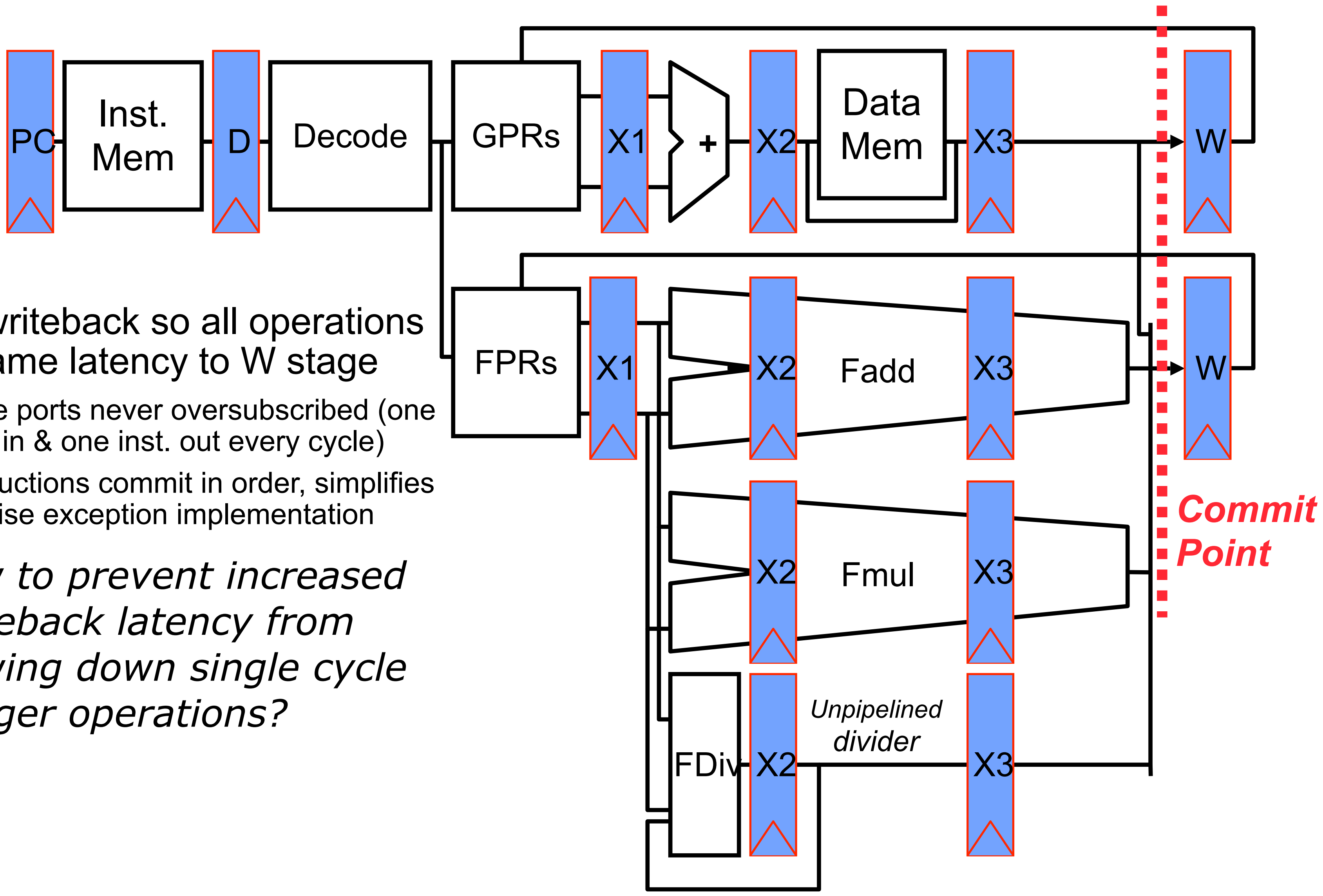


Complex In-Order Pipeline



- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Instructions commit in order, simplifies precise exception implementation

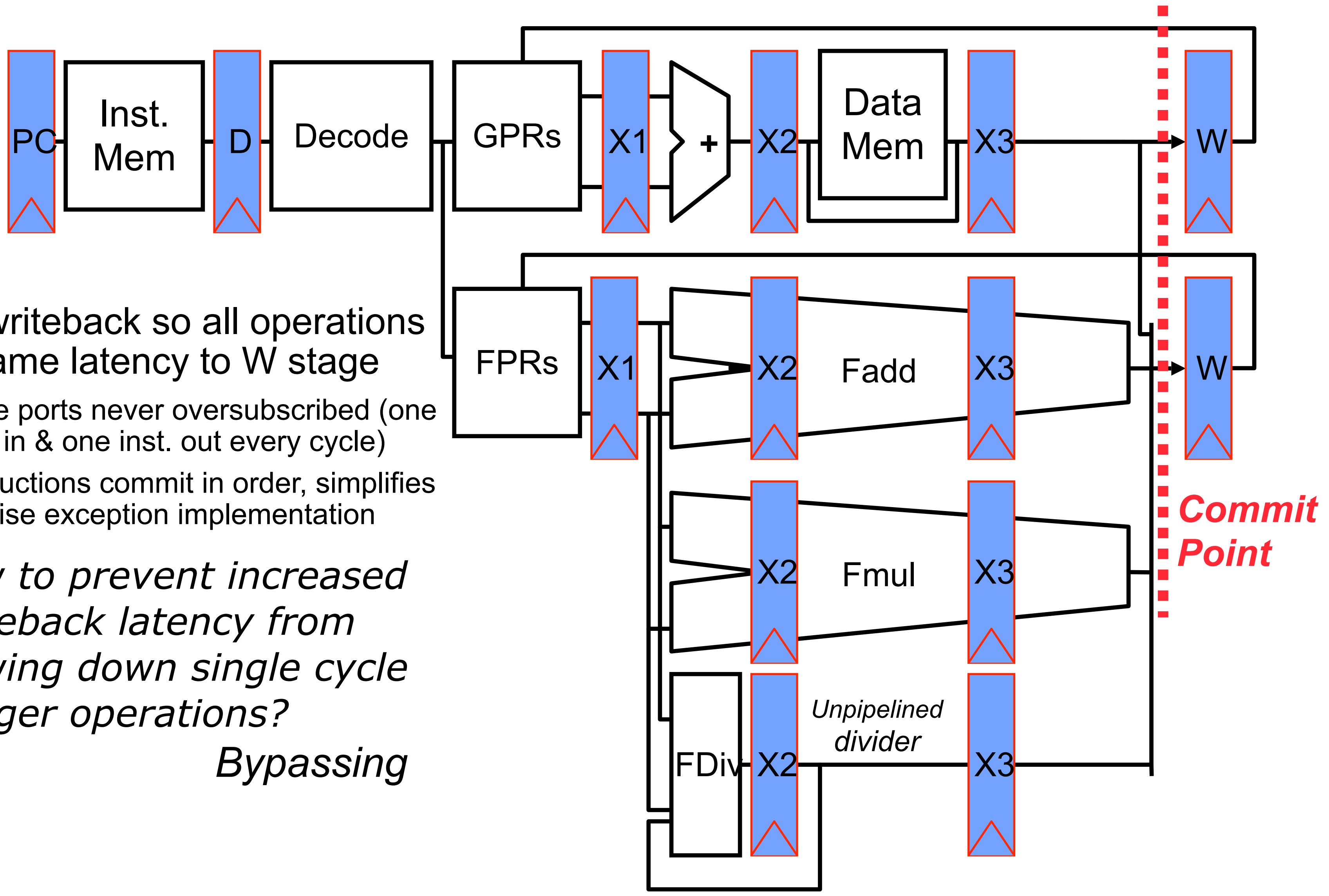
Complex In-Order Pipeline



- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Instructions commit in order, simplifies precise exception implementation

How to prevent increased writeback latency from slowing down single cycle integer operations?

Complex In-Order Pipeline

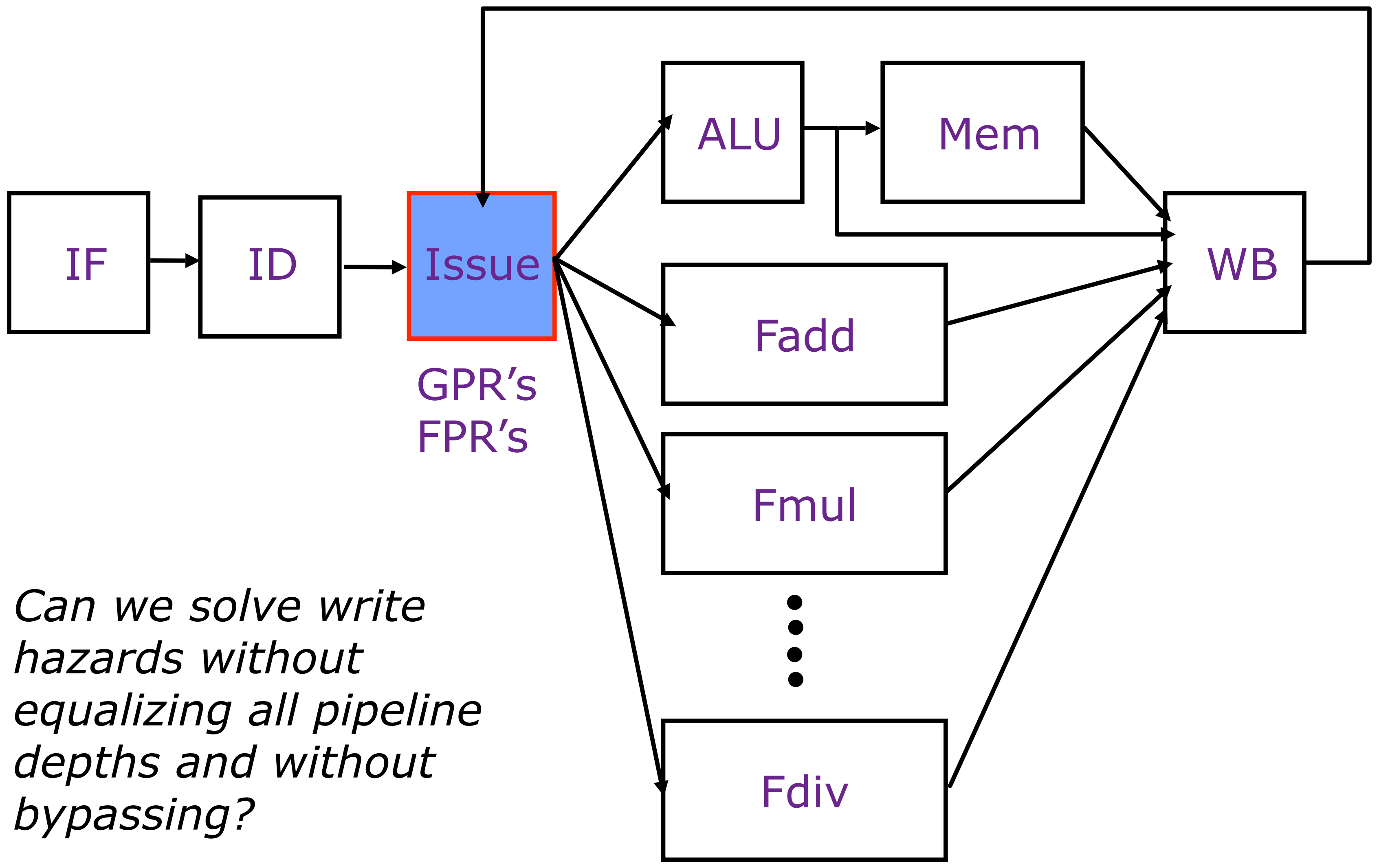


- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Instructions commit in order, simplifies precise exception implementation

How to prevent increased writeback latency from slowing down single cycle integer operations?

Bypassing

Complex In-Order Pipeline



Can we solve write hazards without equalizing all pipeline depths and without bypassing?

When is it Safe to Issue an Instruction?

- Suppose a data structure keeps track of all the instructions in all the functional units
- The following checks need to be made before the Issue stage can dispatch an instruction
    - Is the required function unit available?
    - Is the input data available? $\Rightarrow$ RAW?
    - Is it safe to write the destination? $\Rightarrow$ WAR? WAW?
    - Is there a structural conflict at the WB stage?

Scoreboard for In-Order Issue

Busy[FU#] : a bit-vector to indicate FU's availability.
(FU = Int, Add, Mult, Div)

These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending.

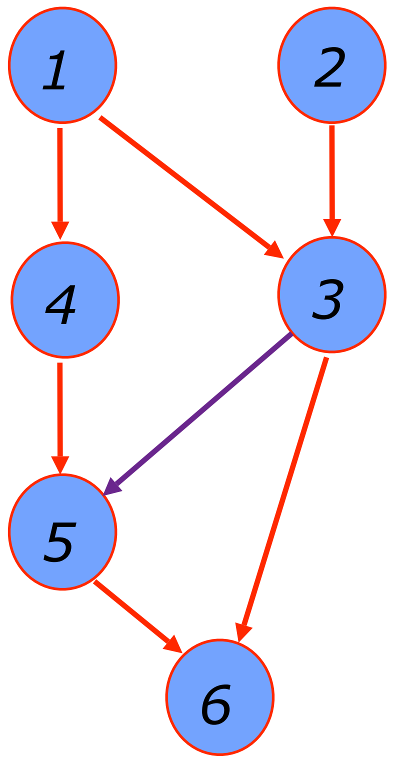
These bits are set to true by the Issue stage and set to false by the WB stage

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?
RAW?
WAR?
WAW?

In-Order Issue Limitations

					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4,	F2,	F8	*4*
6	ADDD	F10,	F6,	F4	*1*

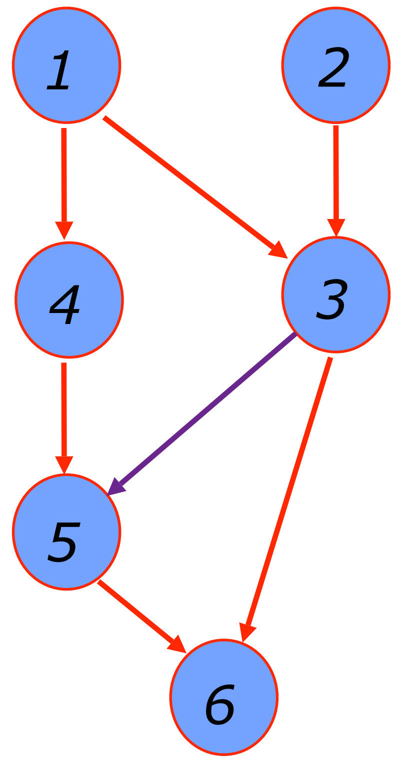

$$\begin{tikzcd} 1 \arrow[d] \arrow[dr] & 2 \arrow[d] \\ 4 \arrow[d] & 3 \arrow[dl] \arrow[ddl] \\ 5 \arrow[d] & \\ 6 & \end{tikzcd}$$

In-order: 1 (2,<u>1</u>) . . . . . . <u>2</u> 3 4 <u>4</u> <u>3</u> 5 . . . <u>5</u> 6 <u>6</u>

(<u>underline</u> indicates cycle when instruction writes back)

In-Order Issue Limitations

					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4,	F2,	F8	*4*
6	ADDD	F10,	F6,	F4	*1*

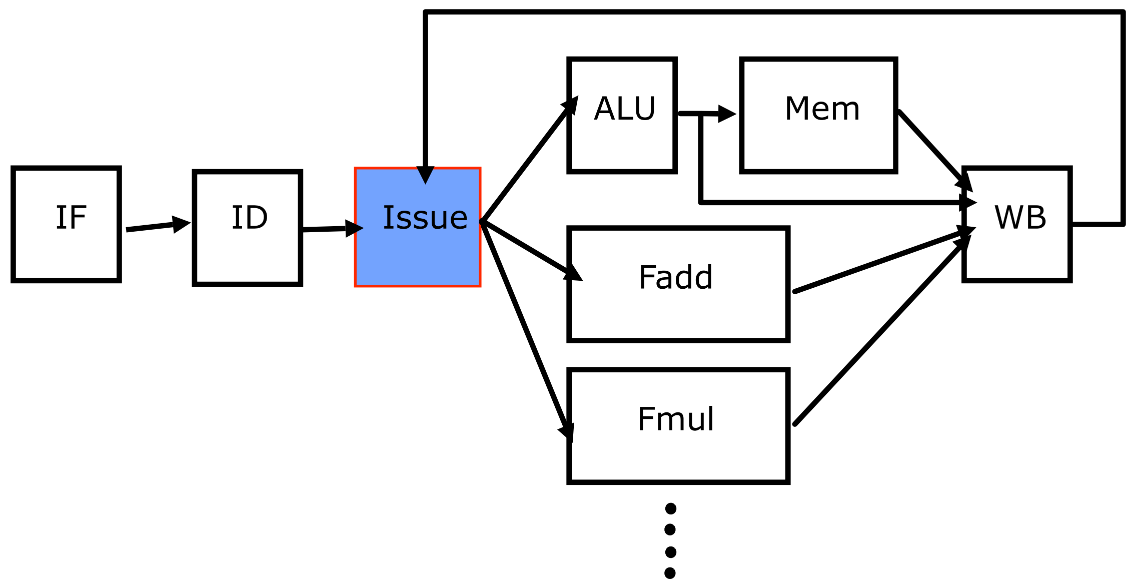


In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

In-order restriction prevents instruction 4 from being dispatched

($\underline{\text{underline}}$ indicates cycle when instruction writes back)

Out-of-Order Issue



- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
- Any instruction in buffer whose RAW hazards are satisfied can be issued (for now at most one dispatch per cycle). On a write back (WB), new instructions may get enabled.

In-Order Issue Limitations Again

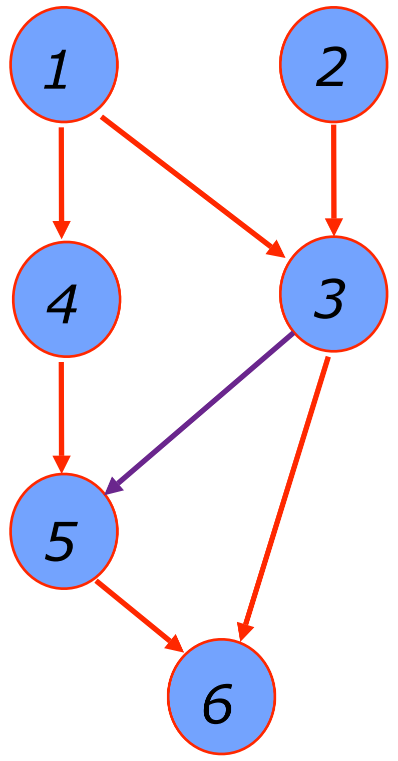
					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4,	F2,	F8	*4*
6	ADDD	F10,	F6,	F4	*1*

$$\begin{tikzcd}
1 \arrow[d] \arrow[drr] & & 2 \arrow[d] \\
4 \arrow[d] & & 3 \arrow[dll] \arrow[ddl] \\
5 \arrow[dr] & & \\
& 6 &
\end{tikzcd}$$

In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

In-Order Issue Limitations Again

					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4,	F2,	F8	*4*
6	ADDD	F10,	F6,	F4	*1*


$$\begin{tikzcd}
1 \arrow[d] \arrow[rd] & 2 \arrow[d] \\
4 \arrow[d] & 3 \arrow[ld] \arrow[d] \\
5 \arrow[r] & 6
\end{tikzcd}$$

In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$
Out-of-order: 1 (2,$\underline{1}$) 4 $\underline{4}$ . . . . $\underline{2}$ 3 . . $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

How many instructions can be in the pipeline?

Which features of an ISA limit the number of instructions in the pipeline?

____

Which features of a program limit the number of instructions in the pipeline?

____

How many instructions can be in the pipeline?

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

Which features of a program limit the number of instructions in the pipeline?

How many instructions can be in the pipeline?

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

Which features of a program limit the number of instructions in the pipeline?

Control transfers

How many instructions can be in the pipeline?

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

Which features of a program limit the number of instructions in the pipeline?

Control transfers

Out-of-order dispatch by itself does not provide any significant performance improvement !

Overcoming the Lack of Register Names

Floating Point pipelines often cannot be kept filled with small number of registers.

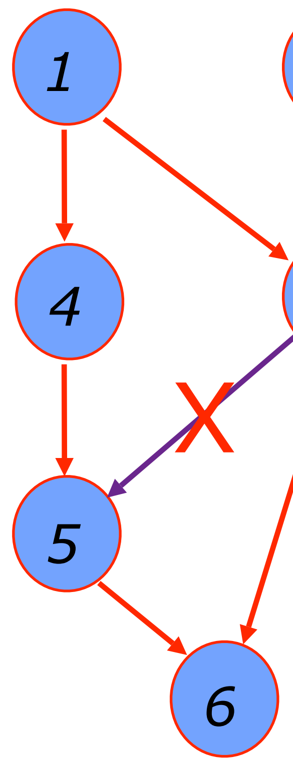
IBM 360 had only 4 Floating Point Registers

Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility ?

Robert Tomasulo of IBM suggested an ingenious solution in 1967 based on on-the-fly *register renaming*

ILP via *Renaming*

					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4’,	F2,	F8	*4*
6	ADDD	F10,	F6,	F4’	*1*

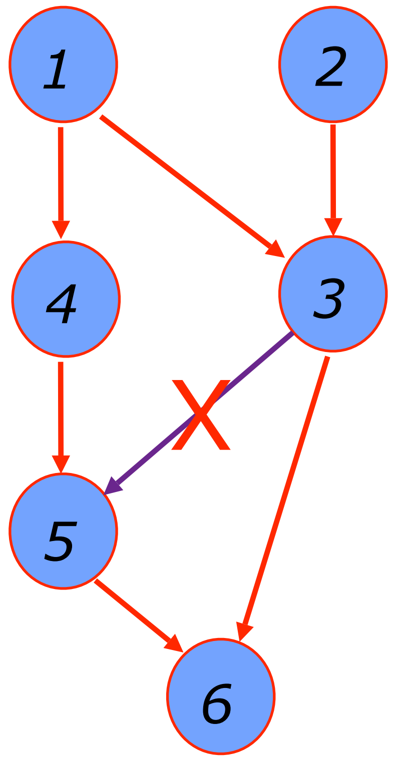


In-order: 1 (2,$\underline{1}$) . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$
Out-of-order: 1 (2,$\underline{1}$) 4 $\underline{4}$ 5 . . . $\underline{2}$ (3,$\underline{5}$) $\underline{3}$ 6 $\underline{6}$

Any antidependence can be eliminated by renaming.
(renaming $\Rightarrow$ additional storage)
Can it be done in hardware?

ILP via *Renaming*

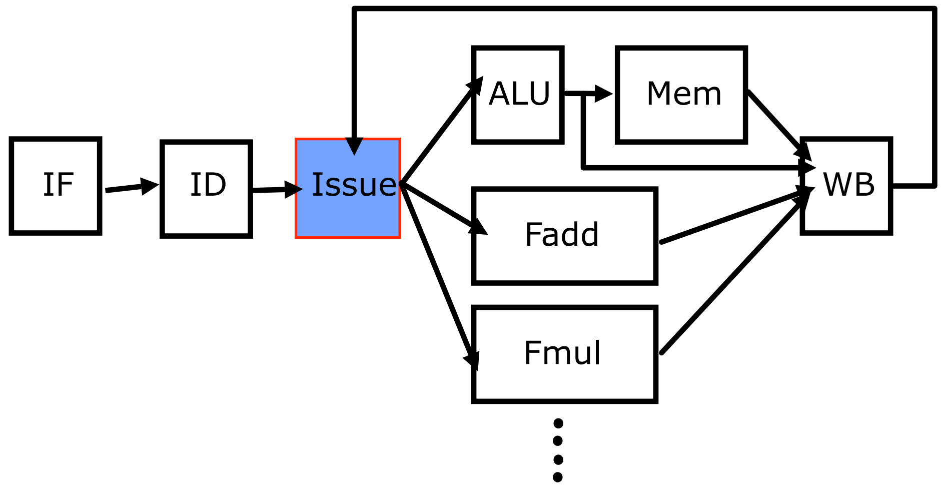
					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4’,	F2,	F8	*4*
6	ADDD	F10,	F6,	F4’	*1*



In-order: 1 (2,$\underline{1}$) . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$
Out-of-order: 1 (2,$\underline{1}$) 4 $\underline{4}$ 5 . . . $\underline{2}$ (3,$\underline{5}$) $\underline{3}$ 6 $\underline{6}$

Any antidependence can be eliminated by renaming.
(renaming $\Rightarrow$ additional storage)
Can it be done in hardware? *yes!*

Register Renaming



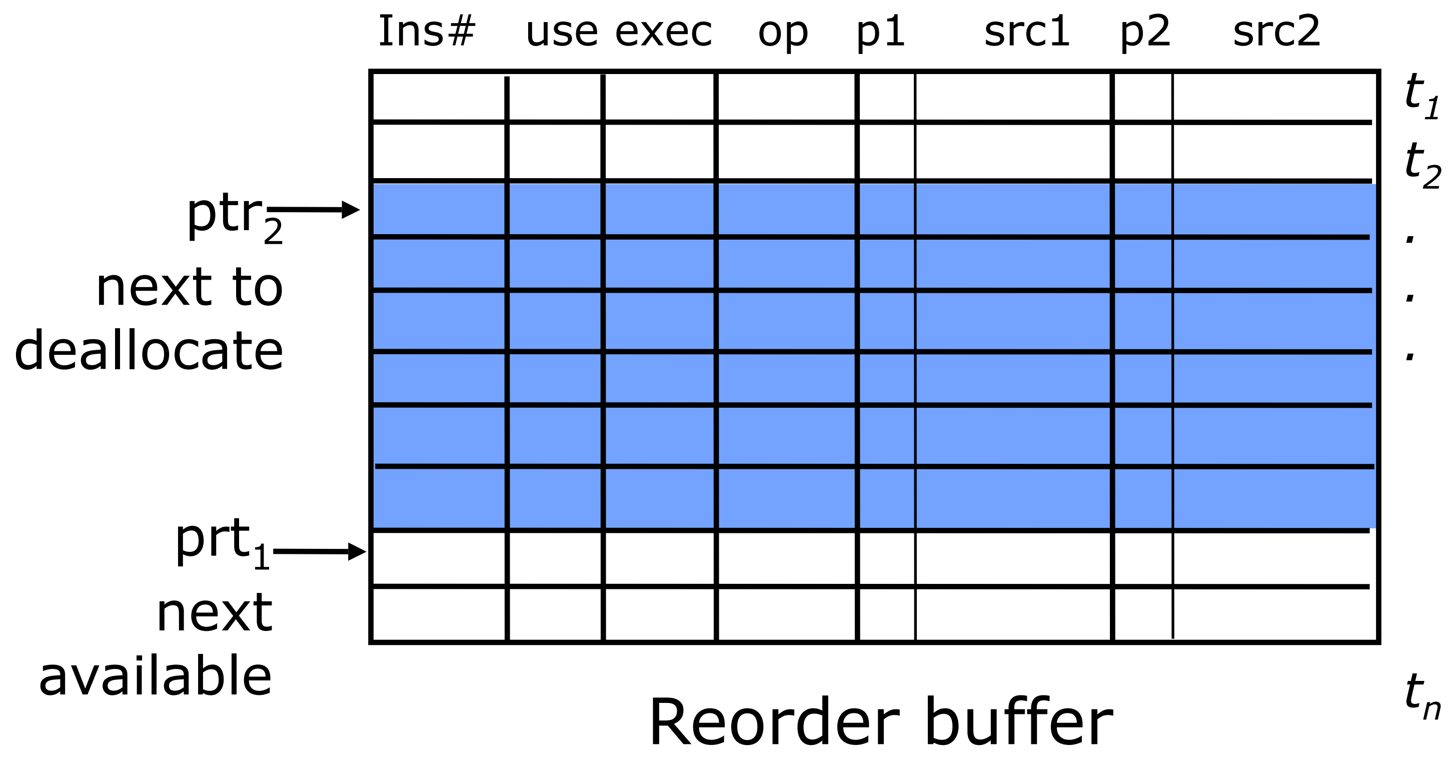
- Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB)

  ⇒ renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.

  ⇒ Out-of-order or dataflow execution

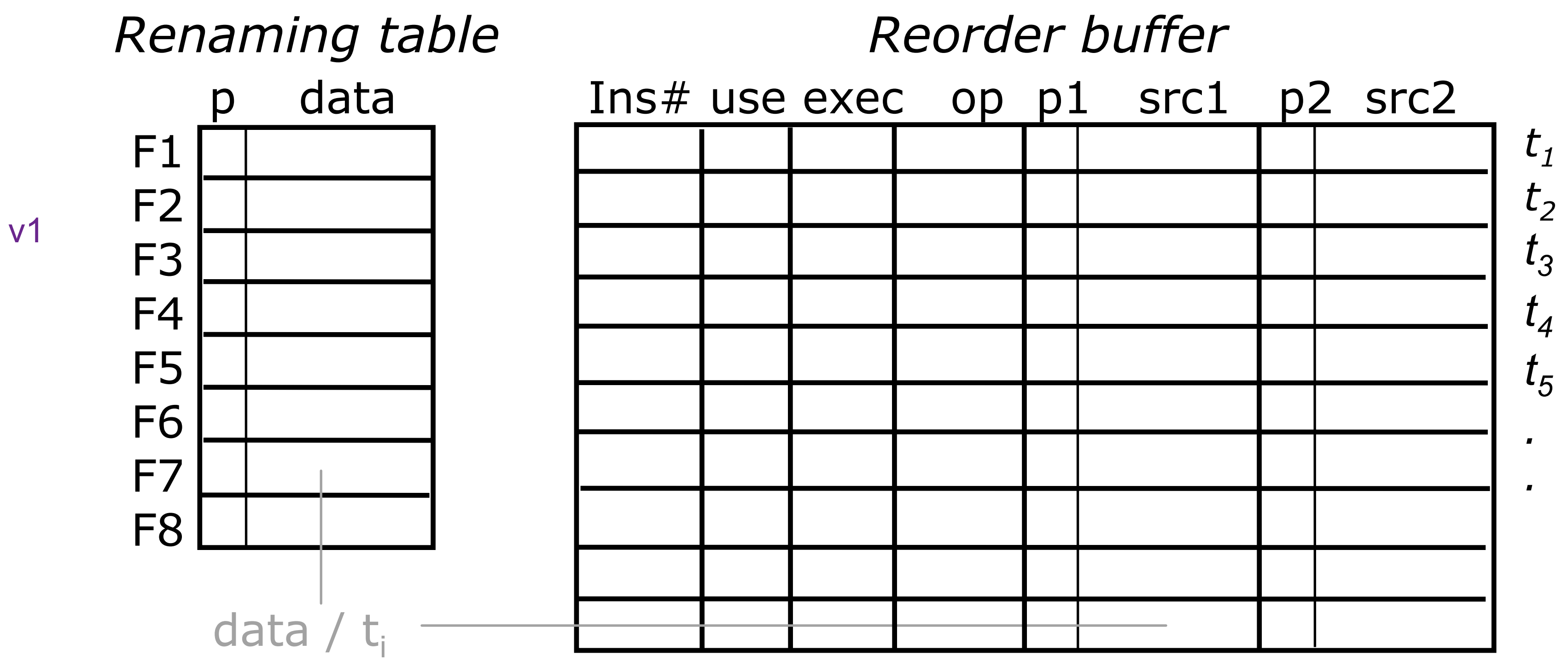
Dataflow Execution



Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

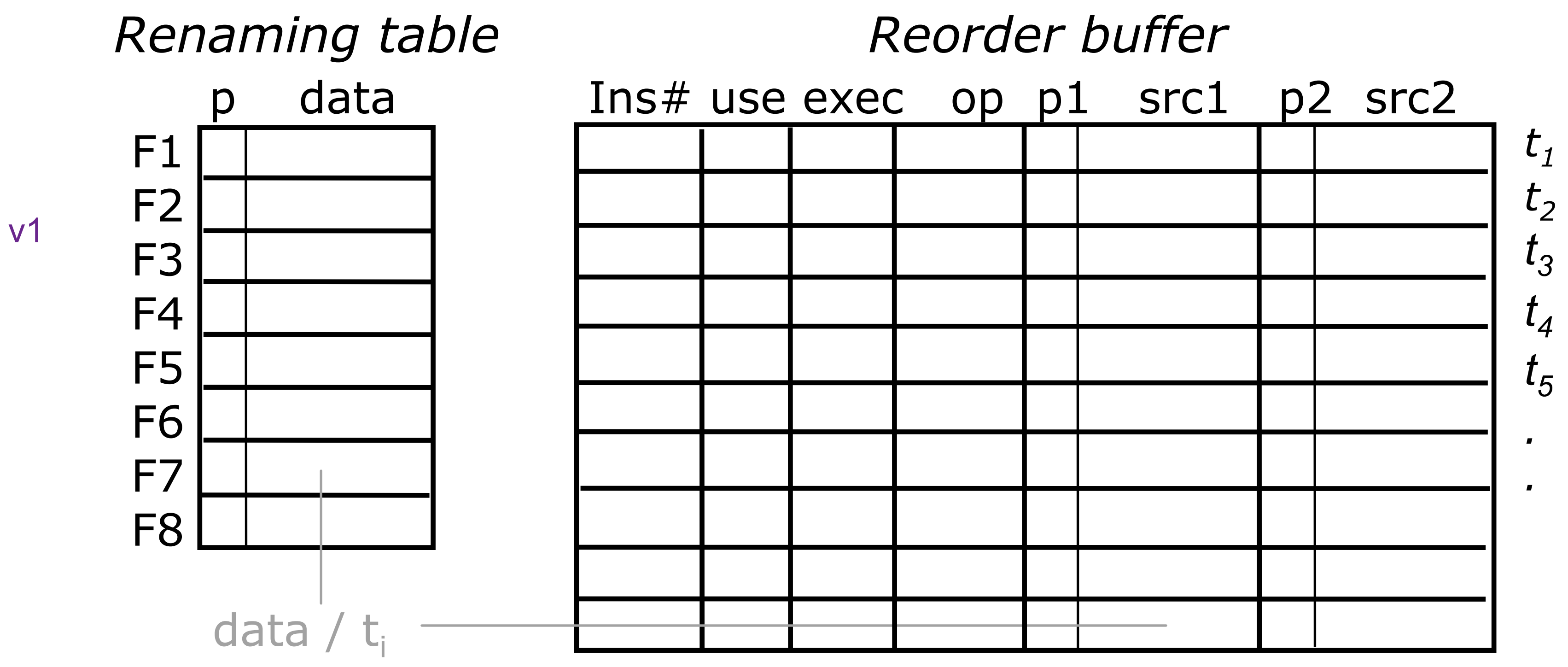
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
- *When can a name be reused?*

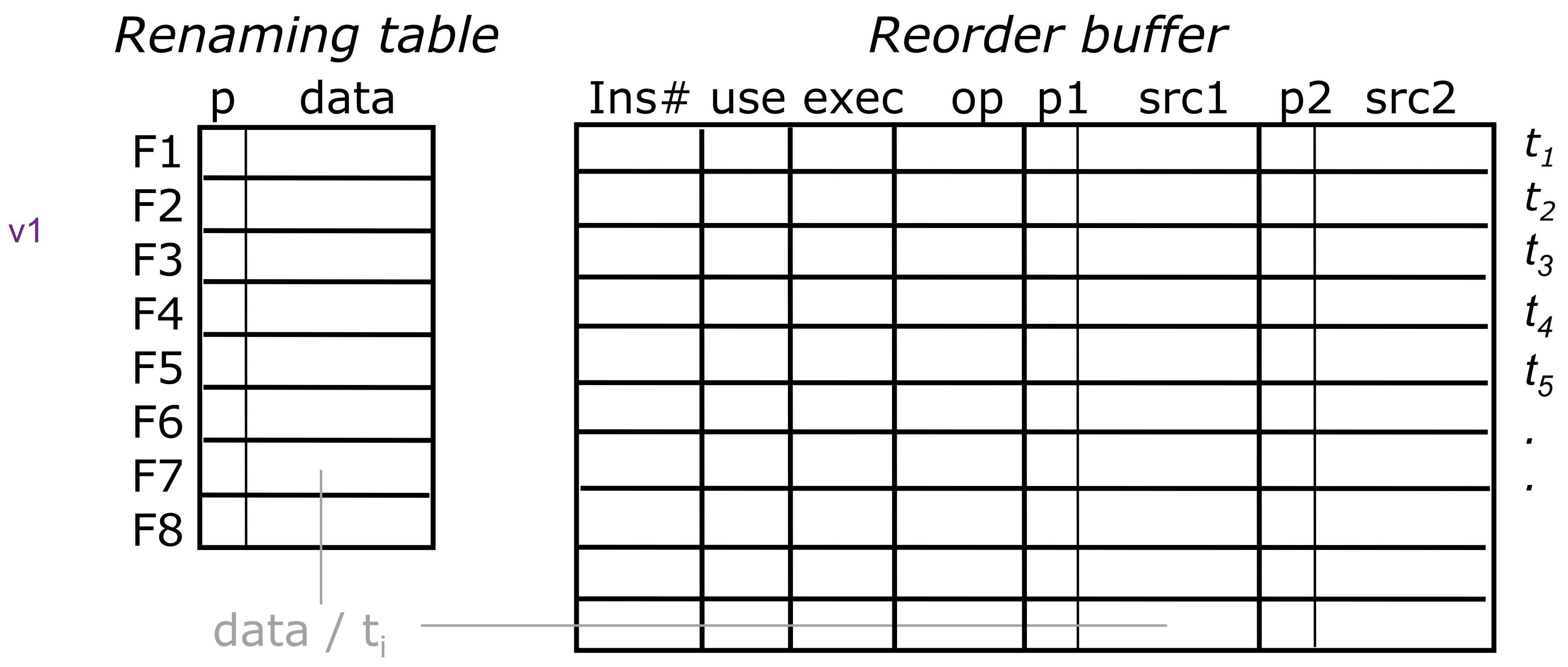
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*

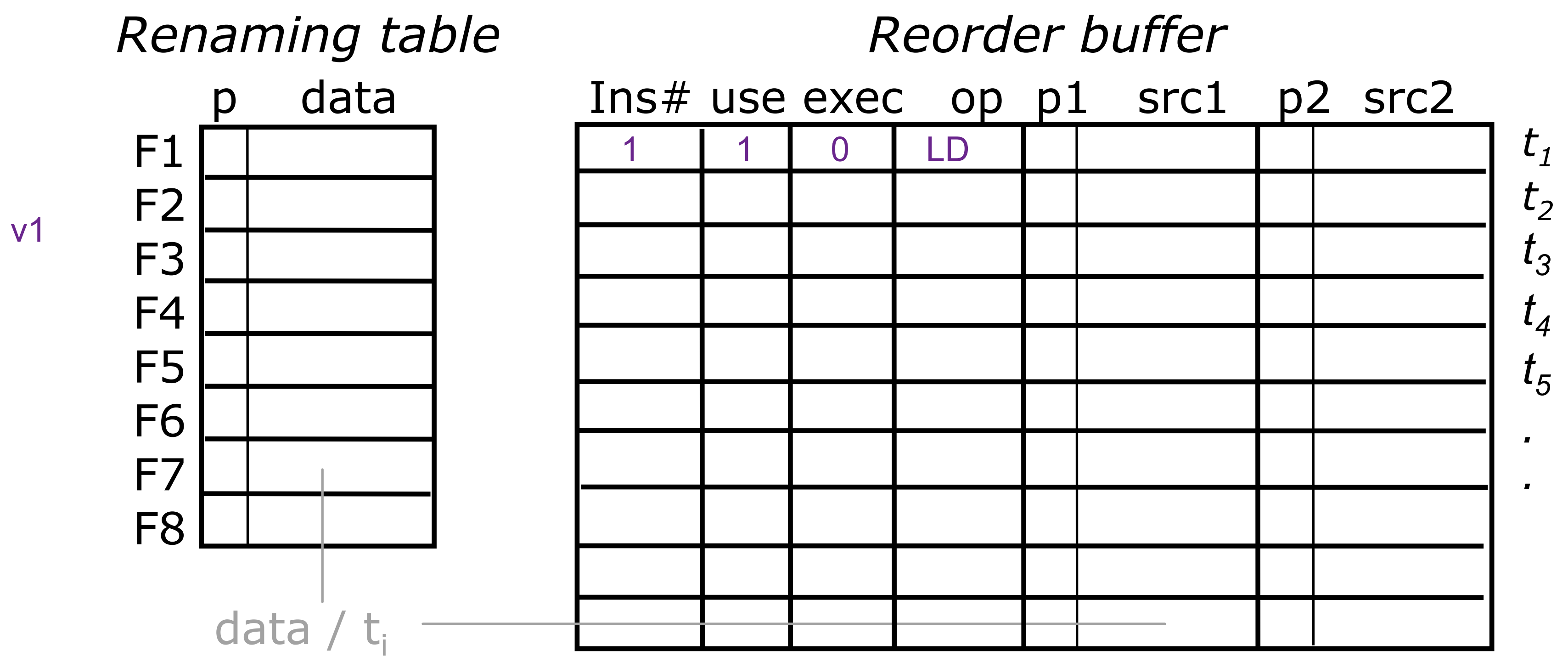
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

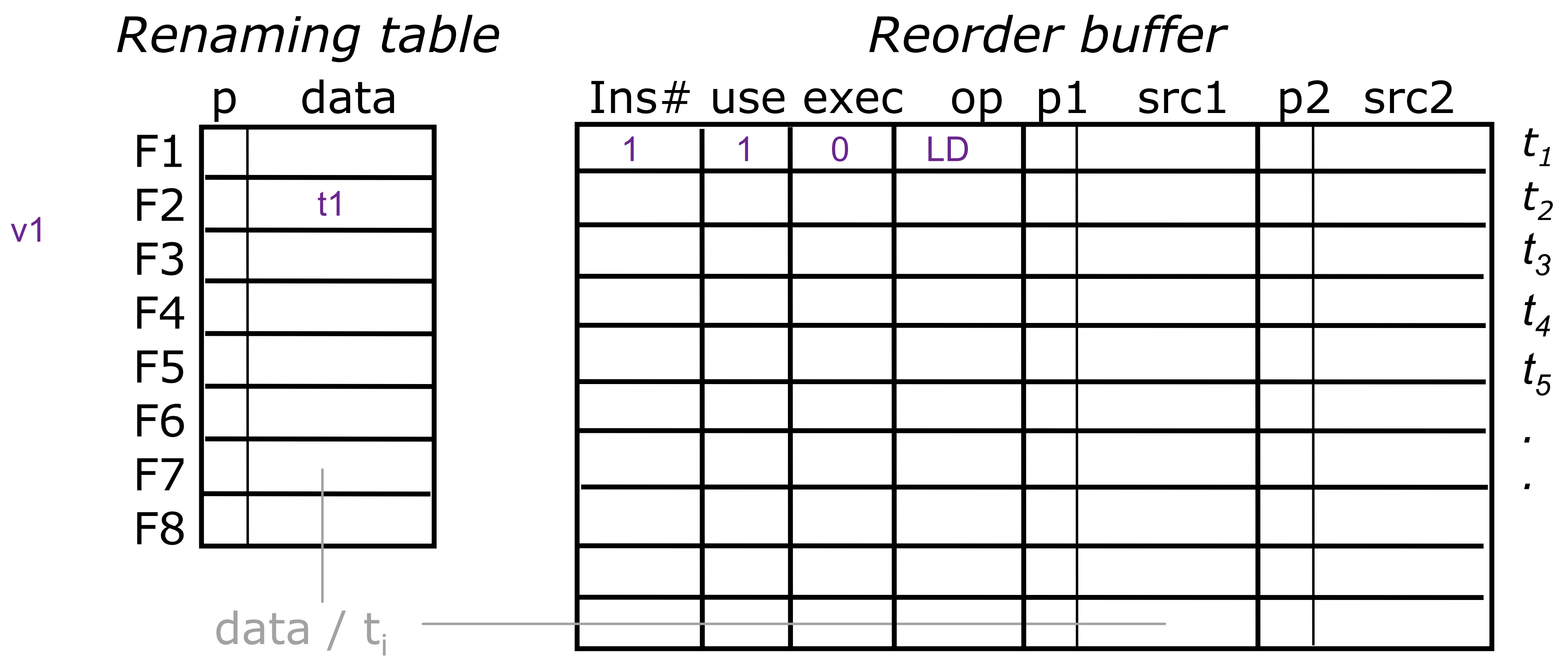
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

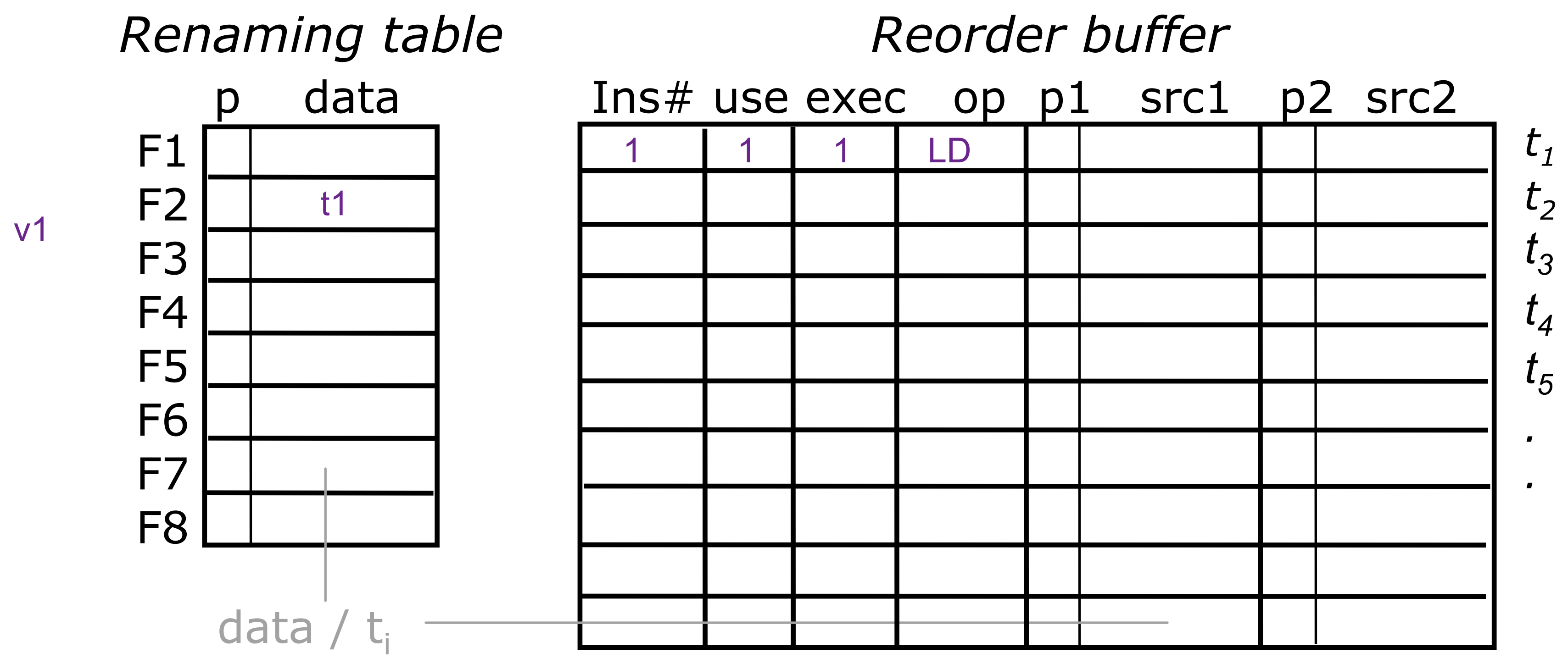
Renaming and Out-of-Order Issue



```
1    LD     F2,   34(R2)
2    LD     F4,   45(R3)
3    MULTD  F6,   F4,    F2
4    SUBD   F8,   F2,    F2
5    DIVD   F4,   F2,    F8
6    ADDD   F10,  F6,    F4
```

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

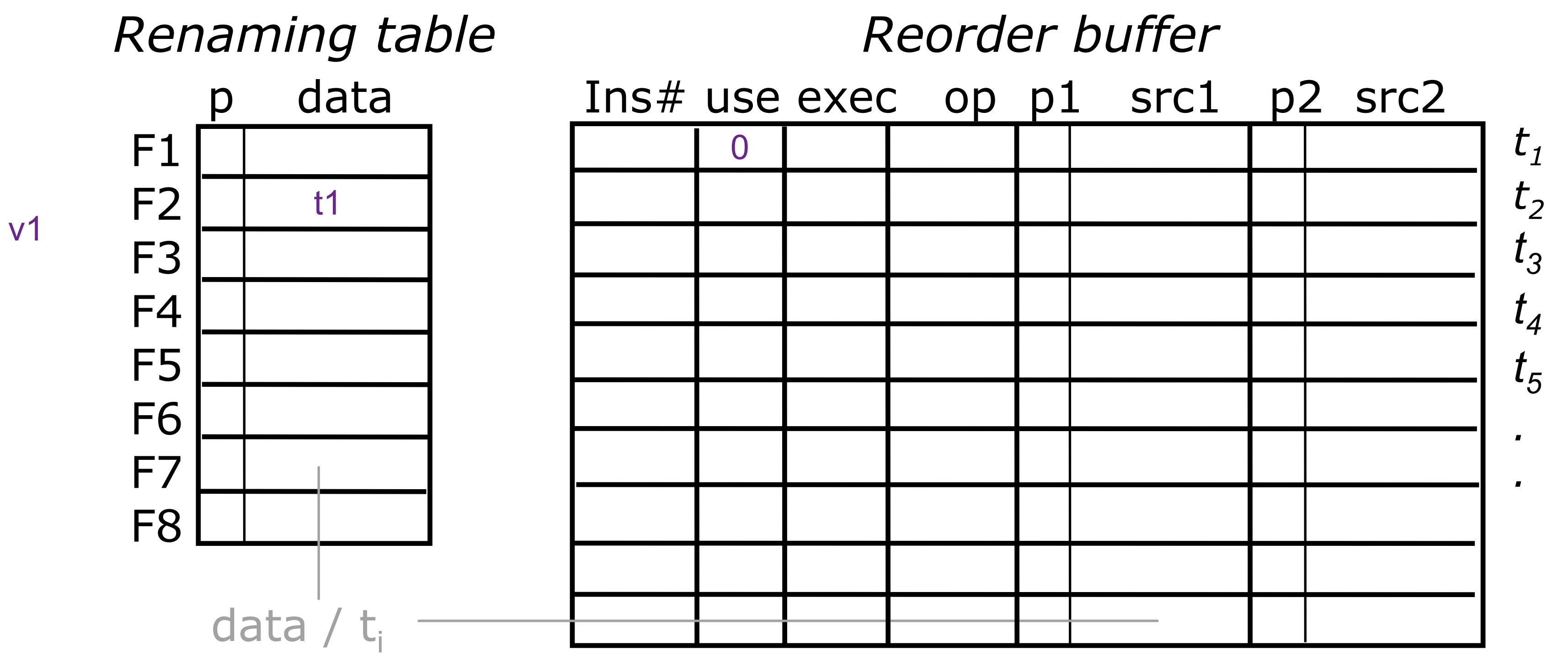
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

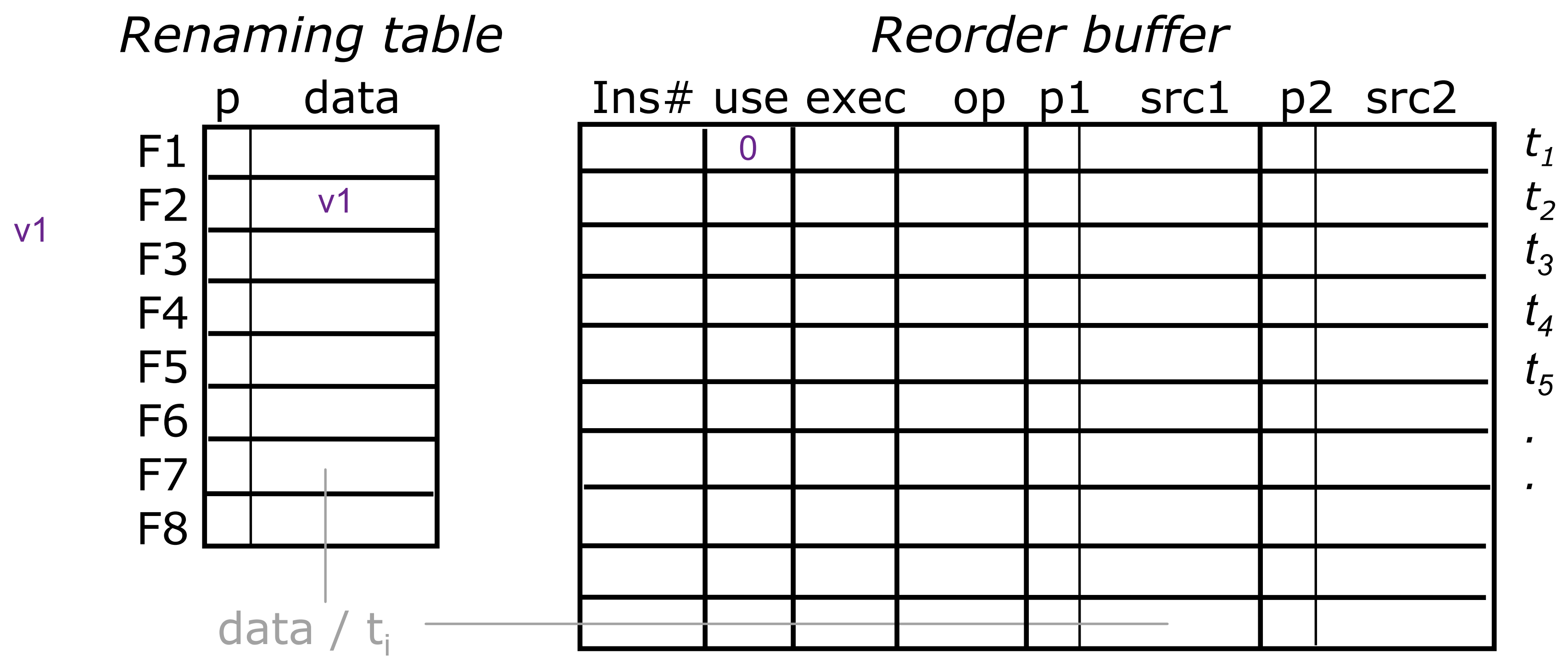
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

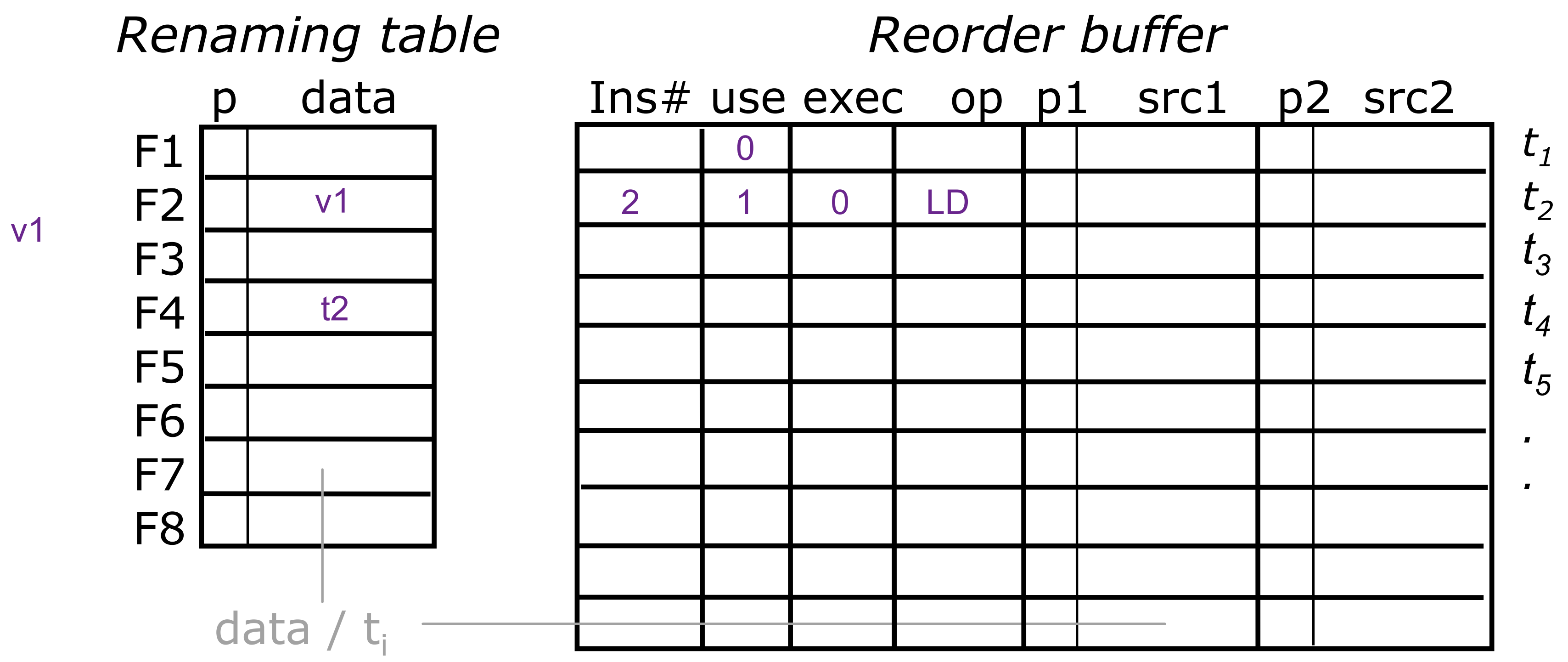
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

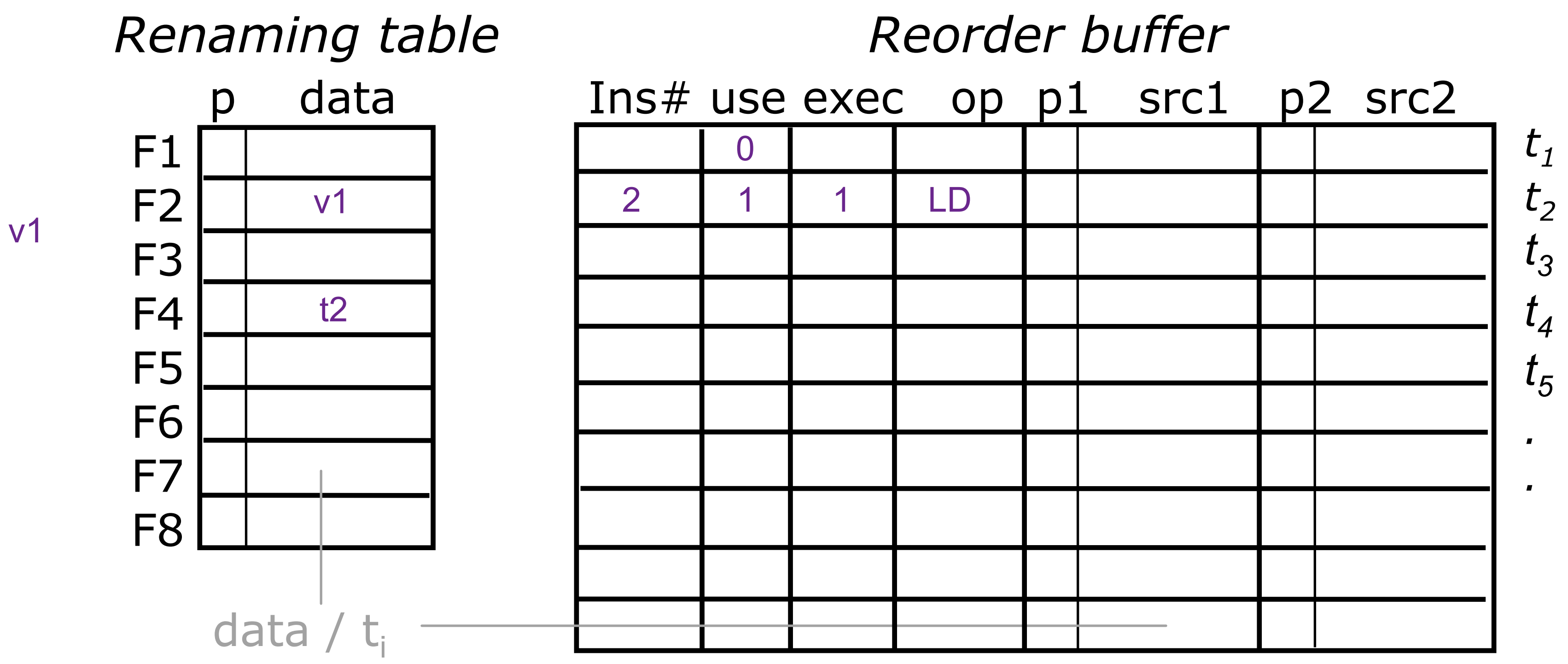
Renaming and Out-of-Order Issue



```
1        LD      F2,     34(R2)
2        LD      F4,     45(R3)
3        MULTD   F6,     F4,      F2
4        SUBD    F8,     F2,      F2
5        DIVD    F4,     F2,      F8
6        ADDD    F10,    F6,      F4
```

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

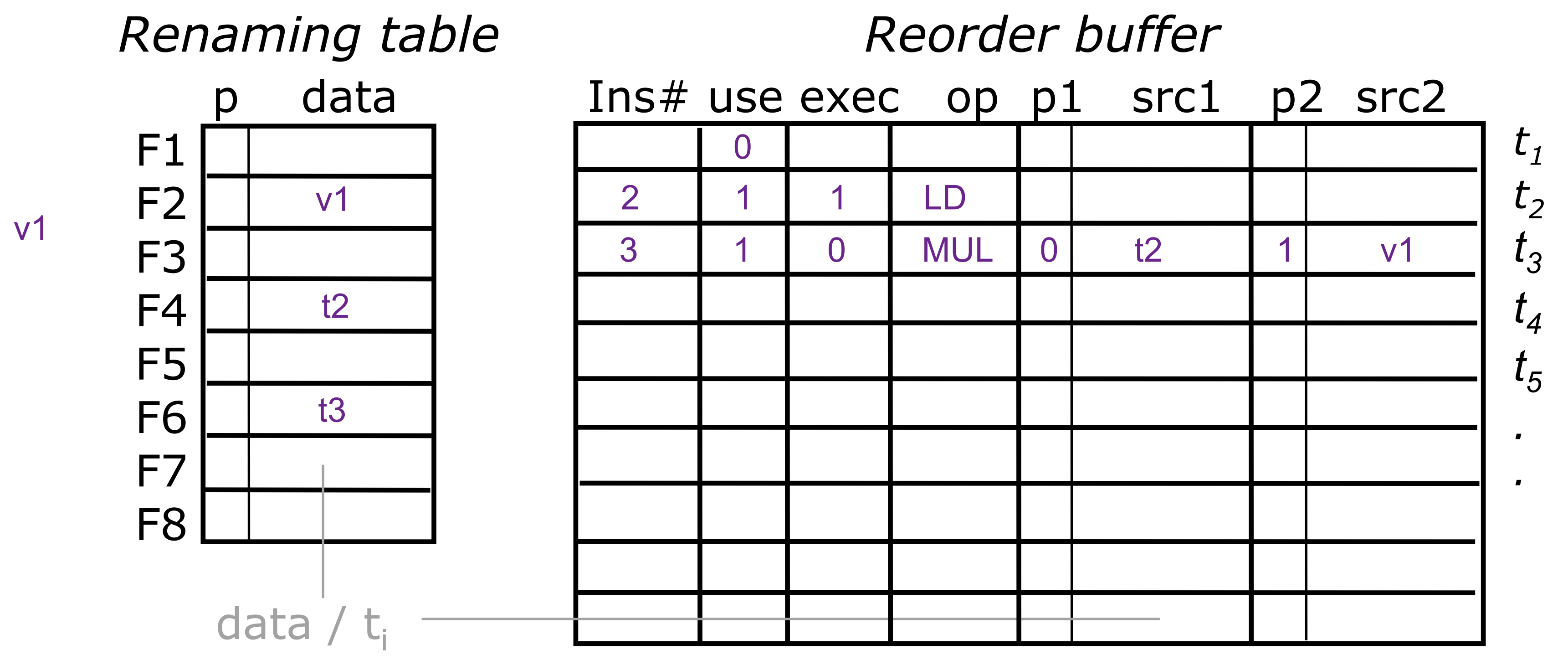
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

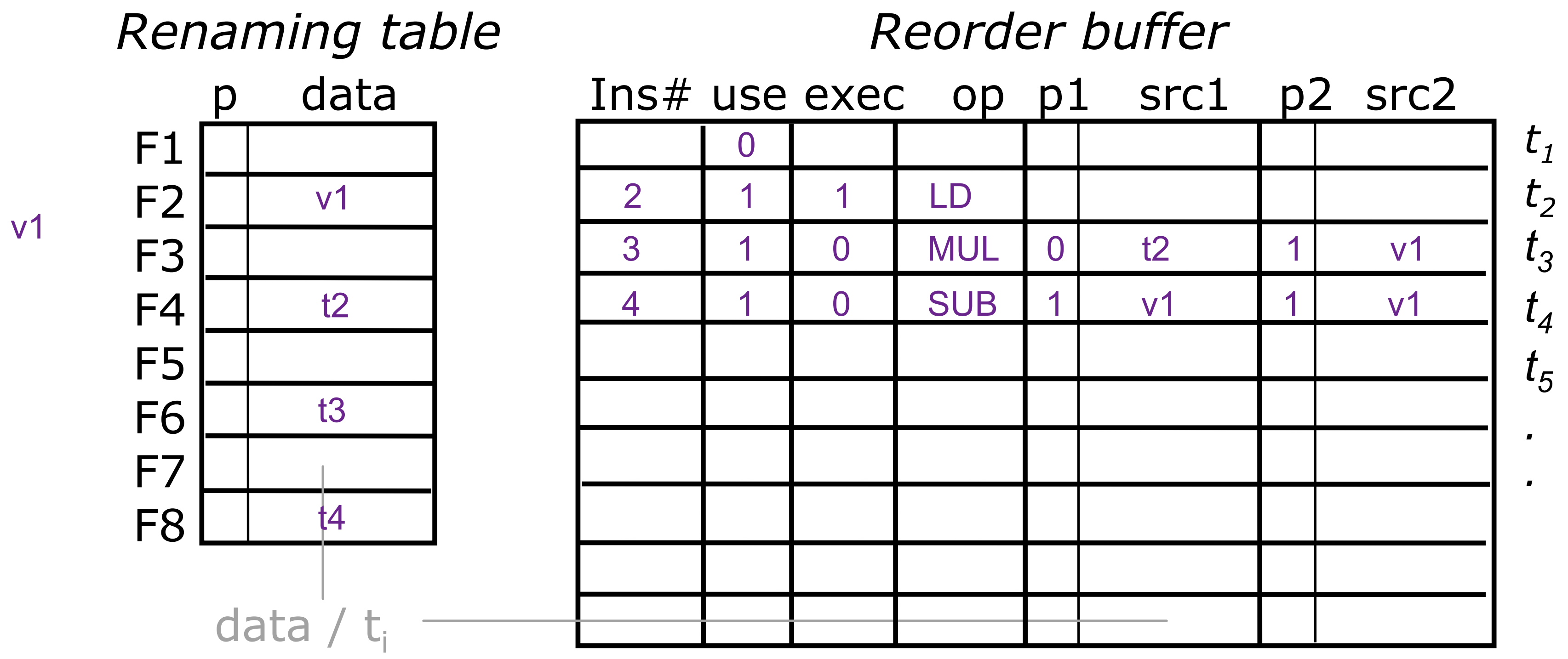
Renaming and Out-of-Order Issue



1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue



Renaming table

	p	data
F1		
F2		v1
F3		
F4		t2
F5		
F6		t3
F7		
F8		t4

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	1	0	SUB	1	v1	1	v1	t_4
								t_5
								.
								.

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue

Renaming table

v1

	p	data
F1		
F2		v1
F3		
F4		t2
F5		
F6		t3
F7		
F8		t4

data / t_i

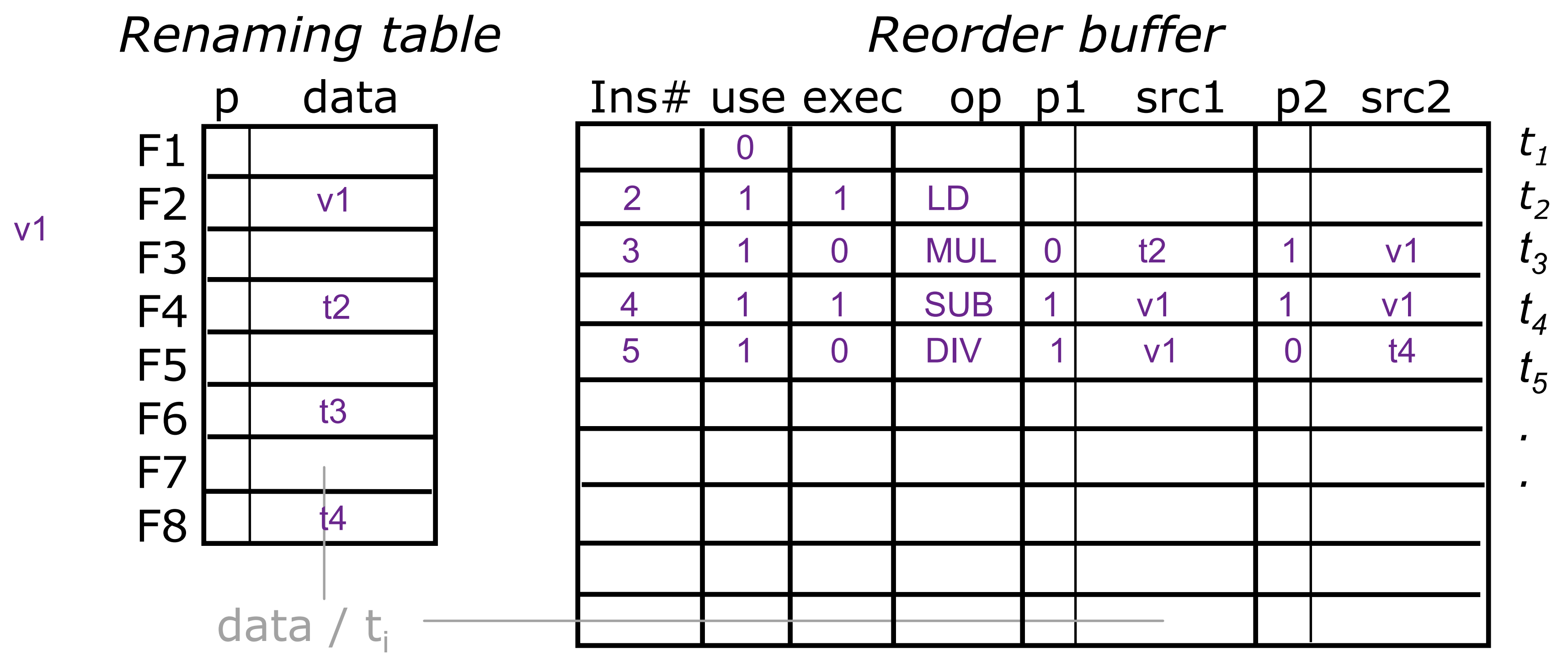
Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	1	1	SUB	1	v1	1	v1	t_4
								t_5
								.
								.

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue



Renaming table

	p	data
F1		
F2		v1
F3		
F4		t2
F5		
F6		t3
F7		
F8		t4

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	1	1	SUB	1	v1	1	v1	t_4
5	1	0	DIV	1	v1	0	t4	t_5
								.
								.

```
1     LD     F2,   34(R2)
2     LD     F4,   45(R3)
3     MULTD  F6,   F4,    F2
4     SUBD   F8,   F2,    F2
5     DIVD   F4,   F2,    F8
6     ADDD   F10,  F6,    F4
```

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue

v1

Renaming table

	p	data
F1		
F2		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		t4

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	1	1	SUB	1	v1	1	v1	t_4
5	1	0	DIV	1	v1	0	t4	t_5
								.
								.

data / t_i

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue

v1

Renaming table

	p	data
F1		
F2		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		t4

Reorder buffer

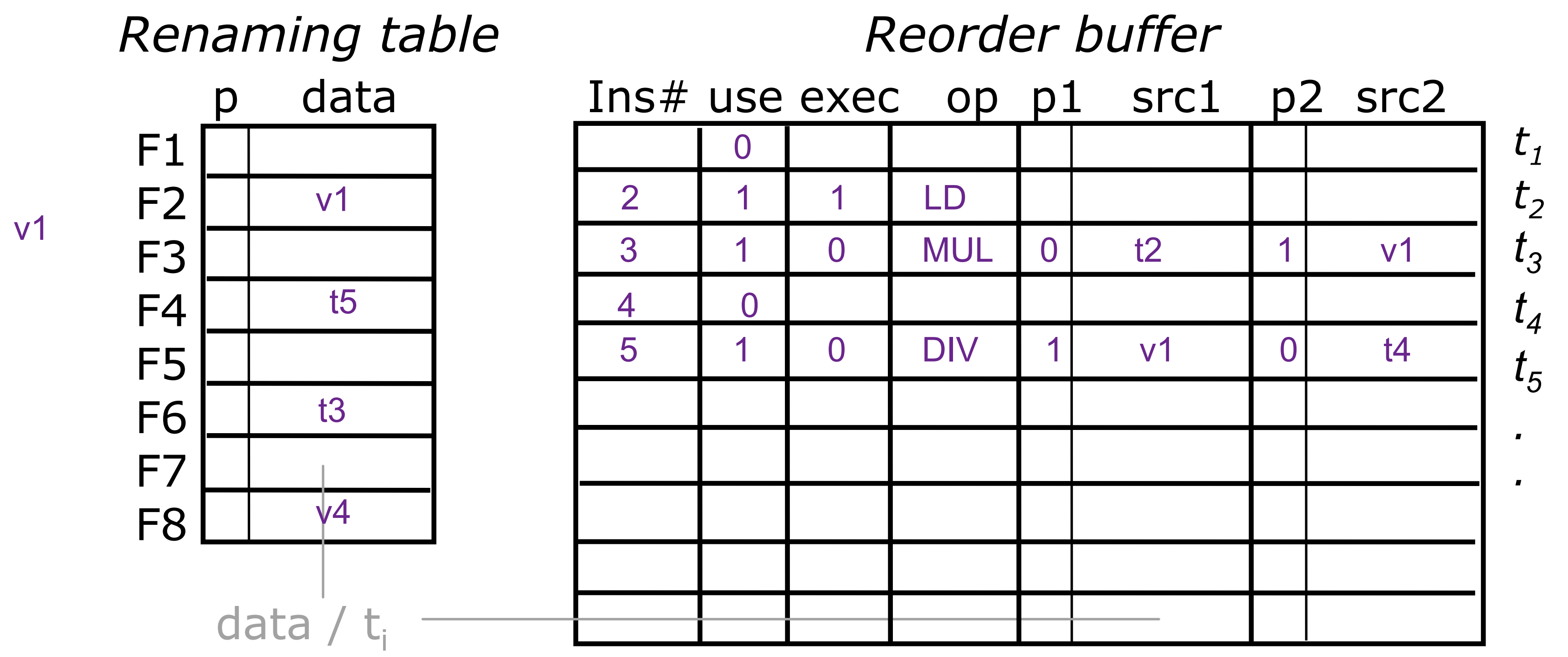
Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	0							t_4
5	1	0	DIV	1	v1	0	t4	t_5
								.
								.

data / t_i

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue



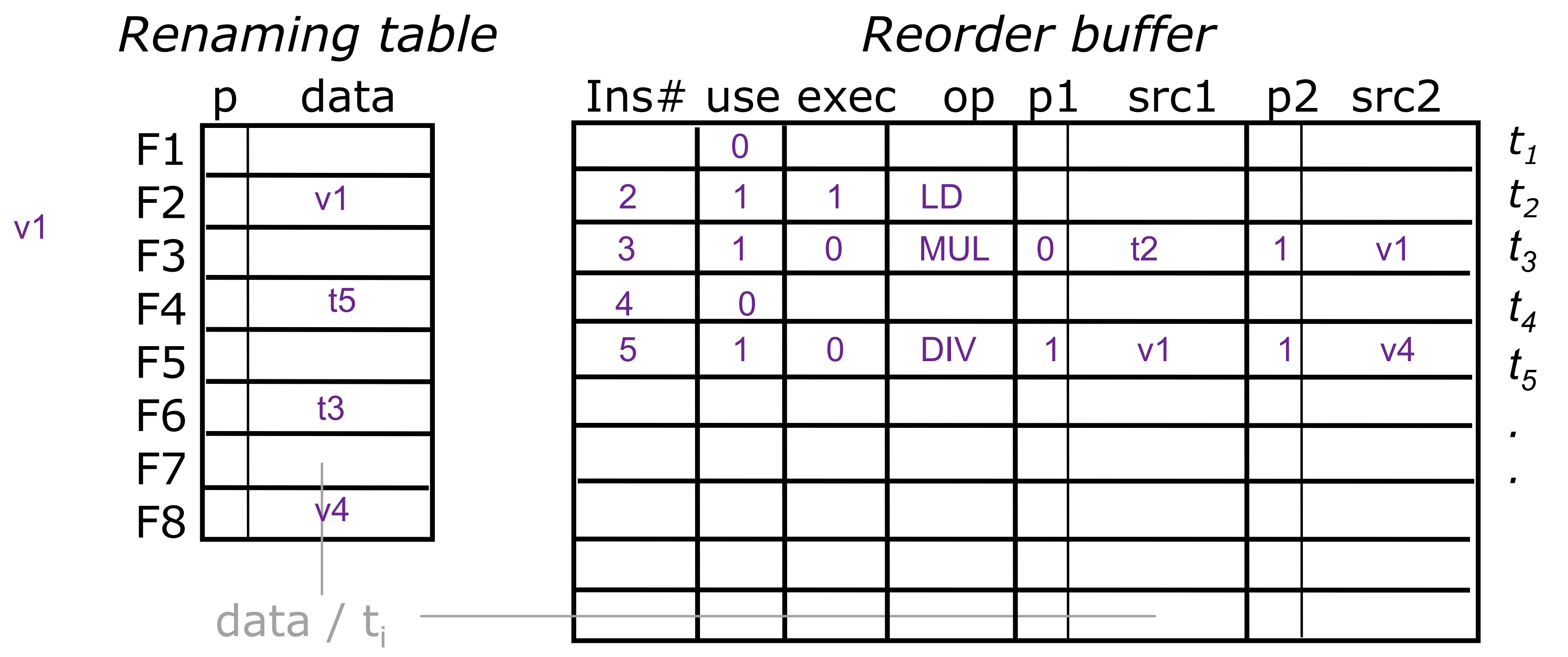
	p	data
F1		
F2		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		v4

Ins#	use	exec	op	p1	src1	p2	src2
	0						
2	1	1	LD				
3	1	0	MUL	0	t2	1	v1
4	0						
5	1	0	DIV	1	v1	0	t4

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue



Renaming table

	p	data
F1		
F2		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		v4

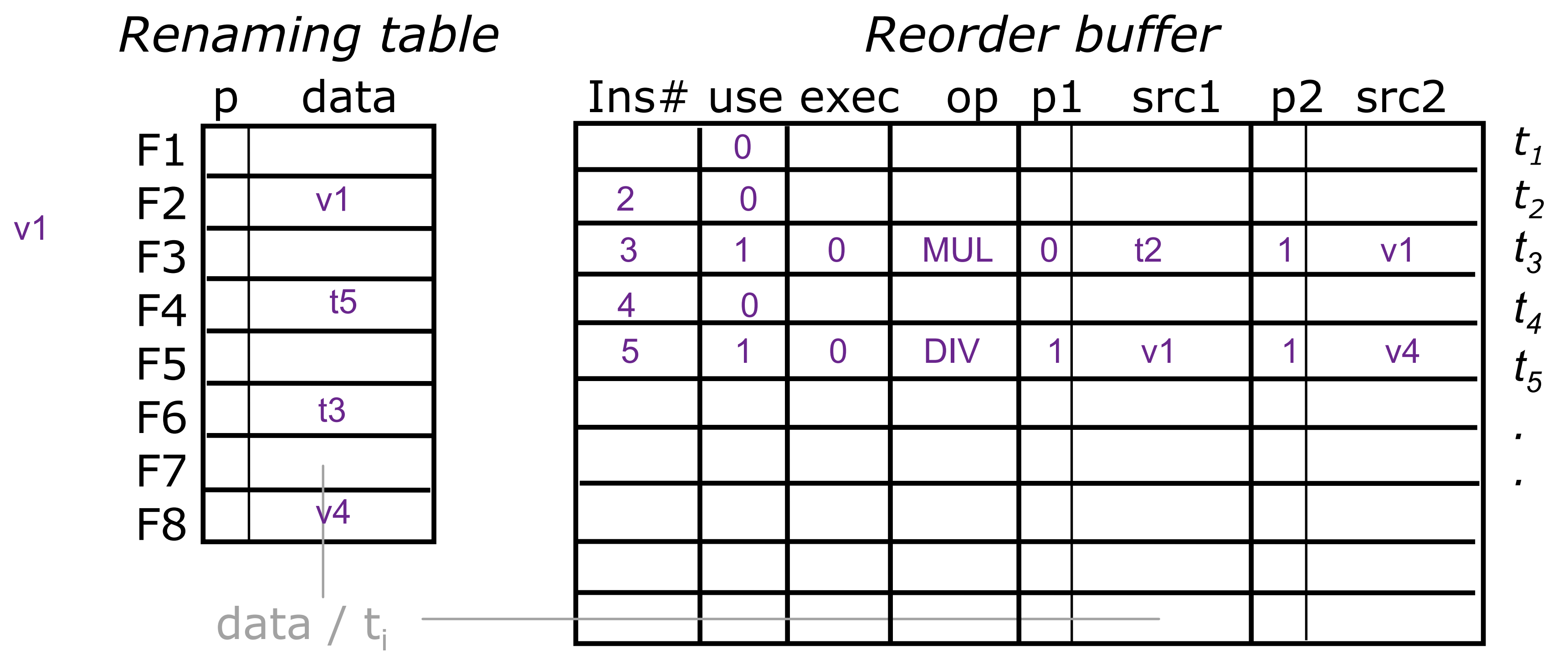
Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	1	1	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	0							t_4
5	1	0	DIV	1	v1	1	v4	t_5
								.
								.

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue



Renaming table

	p	data
F1		
F2		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		v4

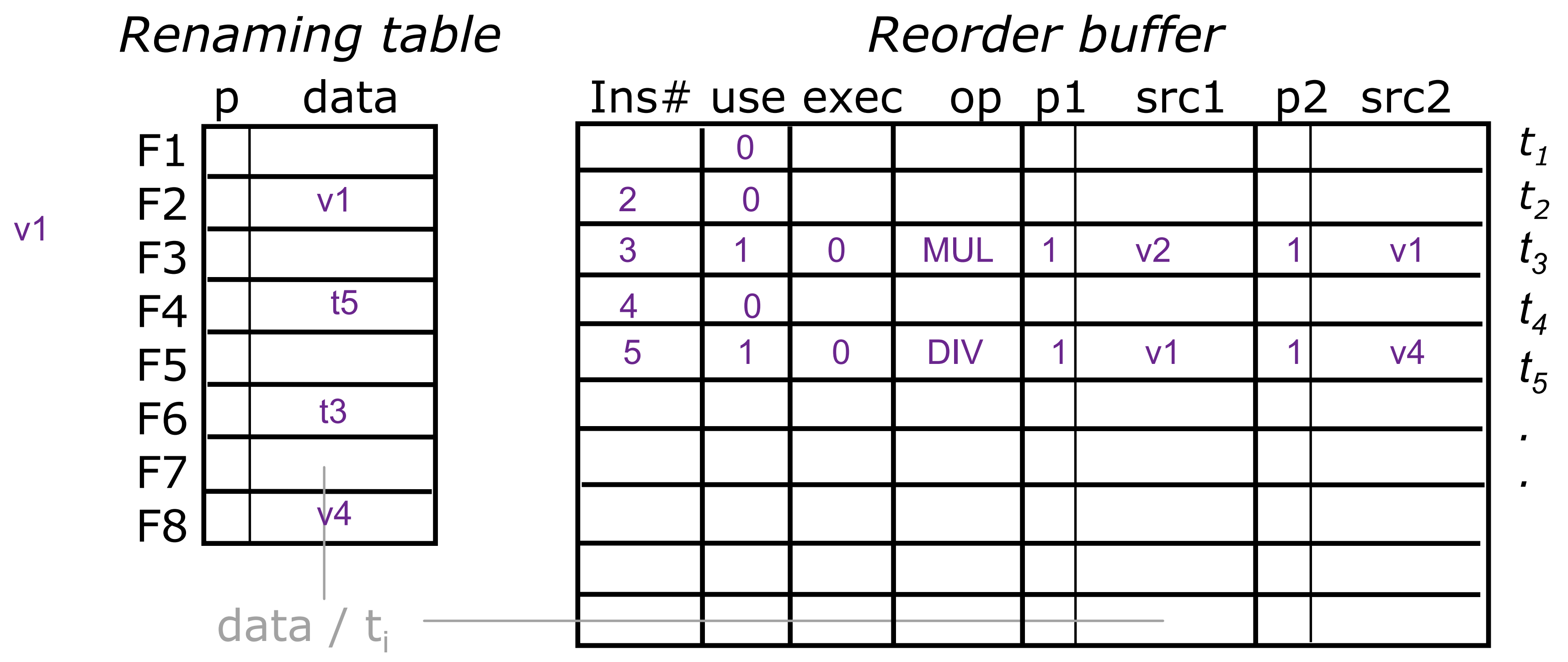
Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	0							t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	0							t_4
5	1	0	DIV	1	v1	1	v4	t_5
								.
								.

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Renaming and Out-of-Order Issue



Renaming table

	p	data
F1		
F2		v1
F3		
F4		t5
F5		
F6		t3
F7		
F8		v4

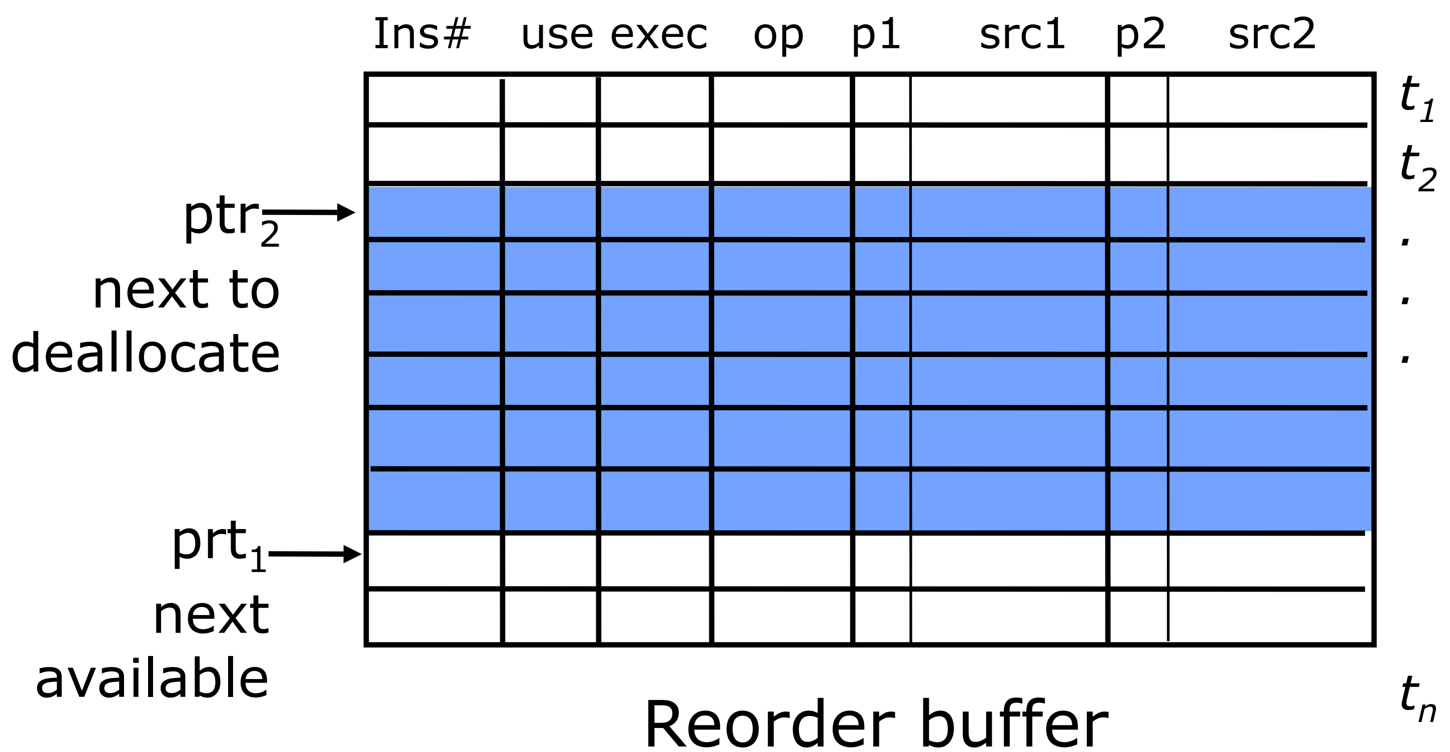
Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
	0							t_1
2	0							t_2
3	1	0	MUL	1	v2	1	v1	t_3
4	0							t_4
5	1	0	DIV	1	v1	1	v4	t_5
								.
								.

1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- *When are names in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Simplifying Allocation & Deallocation



Reorder buffer

Instruction buffer is managed circularly

- "exec" bit is set when instruction begins execution
- When an instruction completes, its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

Effectiveness?

Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

Reasons

1. Effective on a very small class of programs
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Effectiveness?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

Reasons

1. Effective on a very small class of programs
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Control transfers

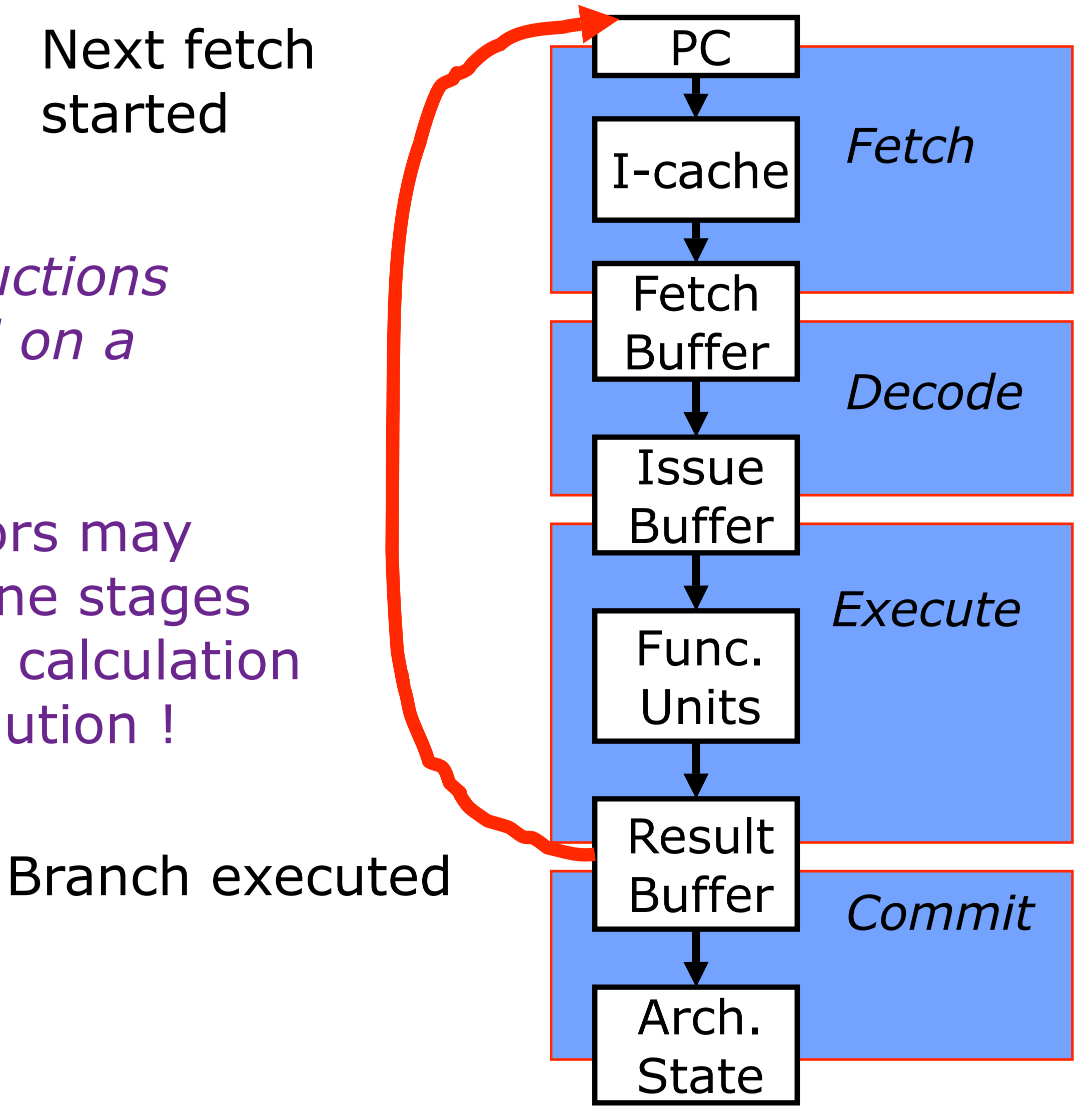
VN execution inefficiencies

```
while ( 1 ) {
   instruction = memory [ state.PC ];
   state = exec(instruction, state);
}
```

Branch Penalty

How many instructions need to be killed on a misprediction?

Modern processors may have > 10 pipeline stages between next pc calculation and branch resolution !



Average Run-Length Between Branches

Average dynamic instruction mix from SPEC92:

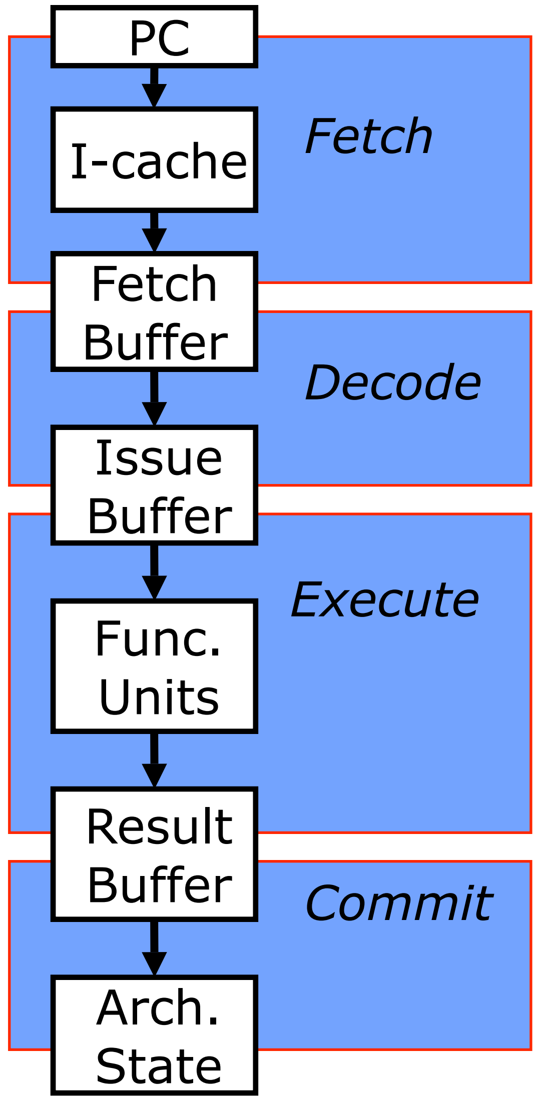
	SPECint92	SPECfp92
ALU	39 %	13 %
FPU Add		20 %
FPU Mult		13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92:	*compress, eqntott, espresso, gcc , li*
SPECfp92:	*doduc, ear, hydro2d, mdijdp2, su2cor*

What is the average *run length* between branches?

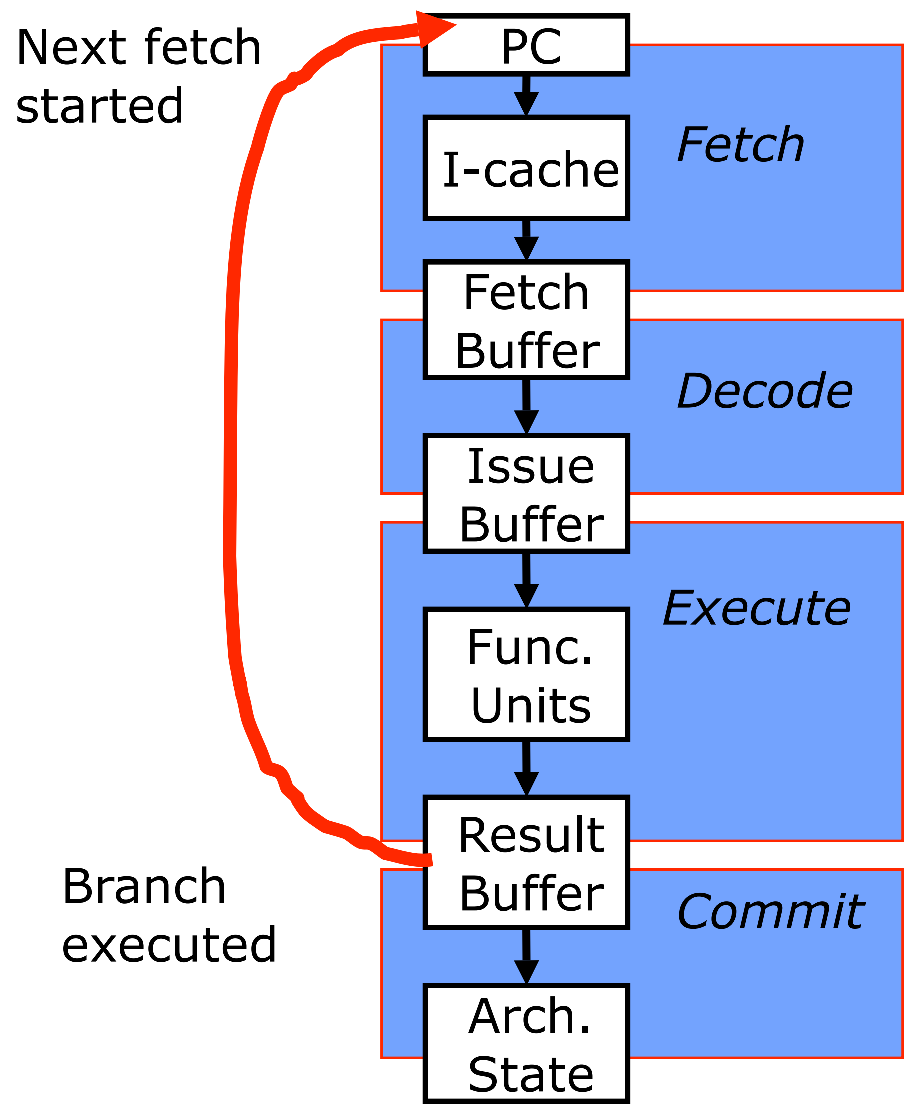
Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !



Control Flow Penalty

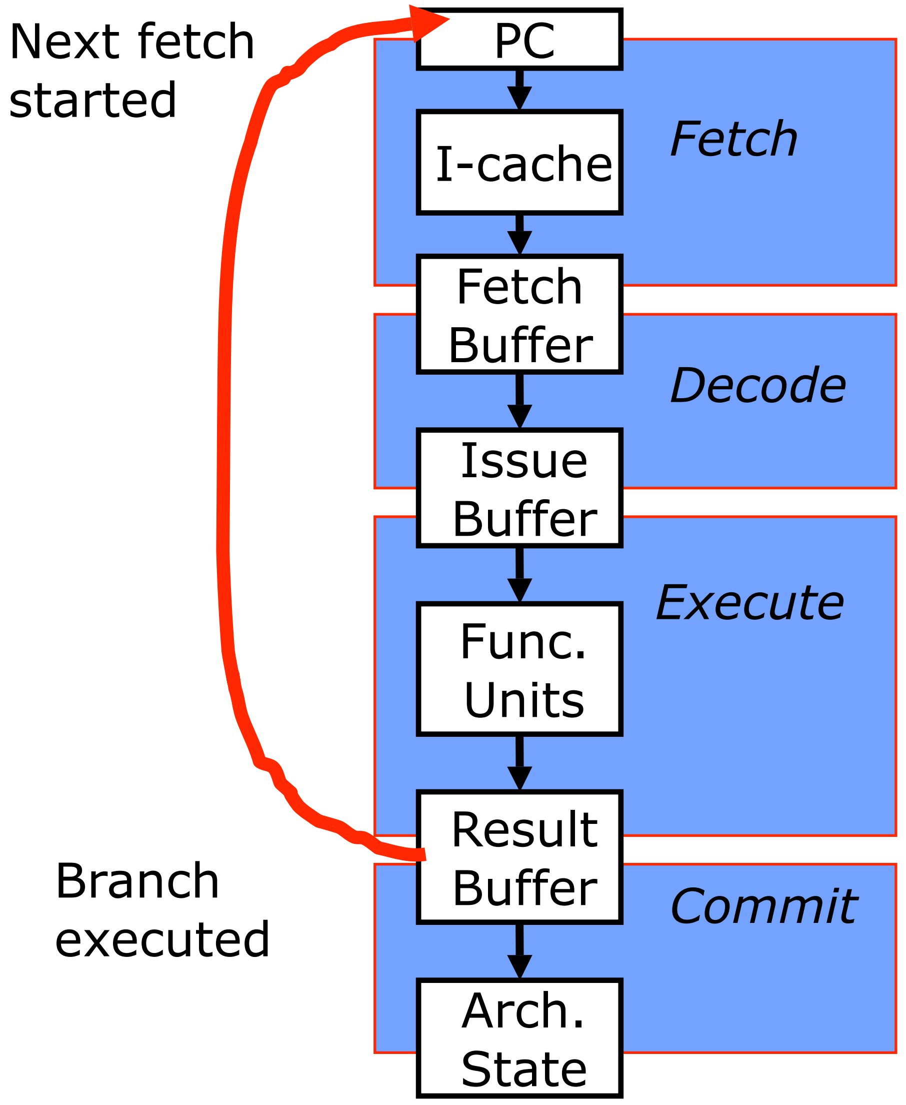
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !



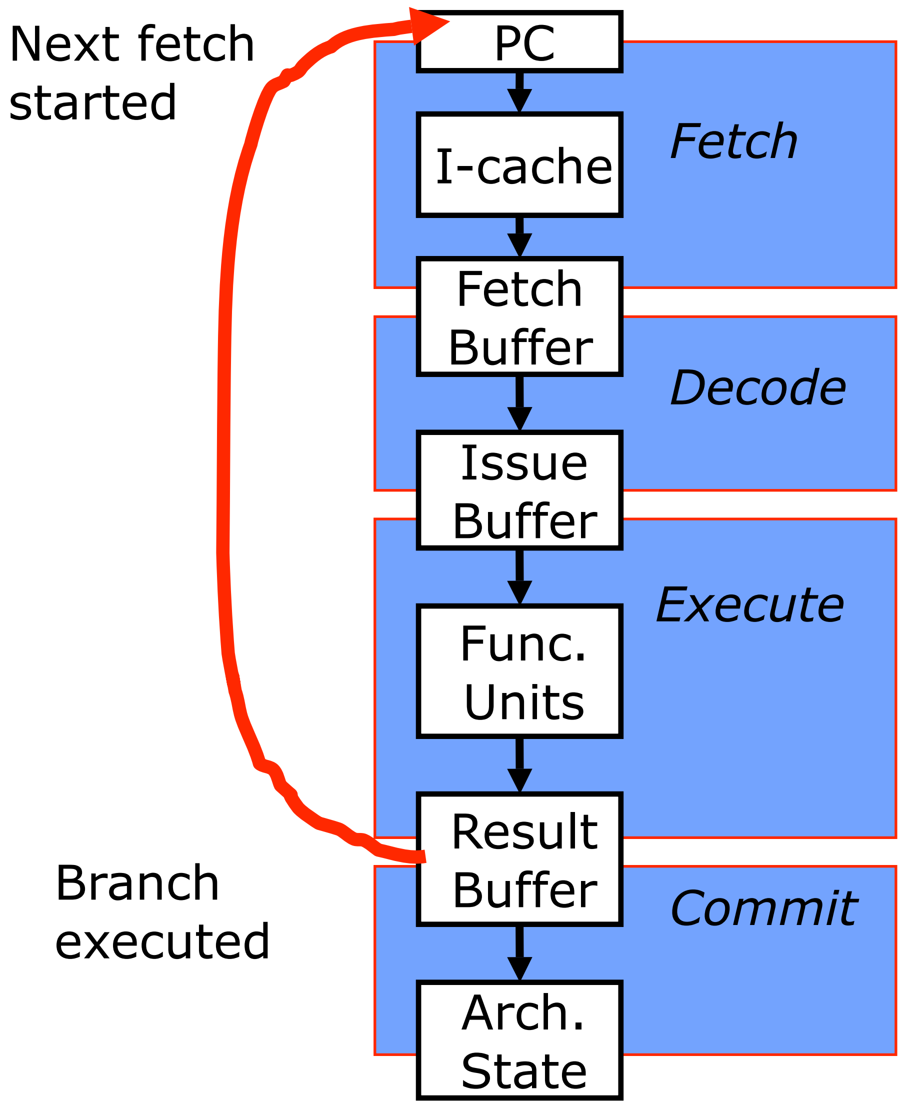
Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?



Control Flow Penalty



Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?

~ Loop length x pipeline width

Reducing Control Flow Penalty

Software solutions

- *Eliminate branches - loop unrolling*

  Increases the run length
- *Reduce resolution time - instruction scheduling*

  Compute the branch condition as early as possible (of limited value)

Hardware solutions

- Find something else to do - *delay slots*

  Replaces pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*

  Speculative execution of instructions beyond the branch

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

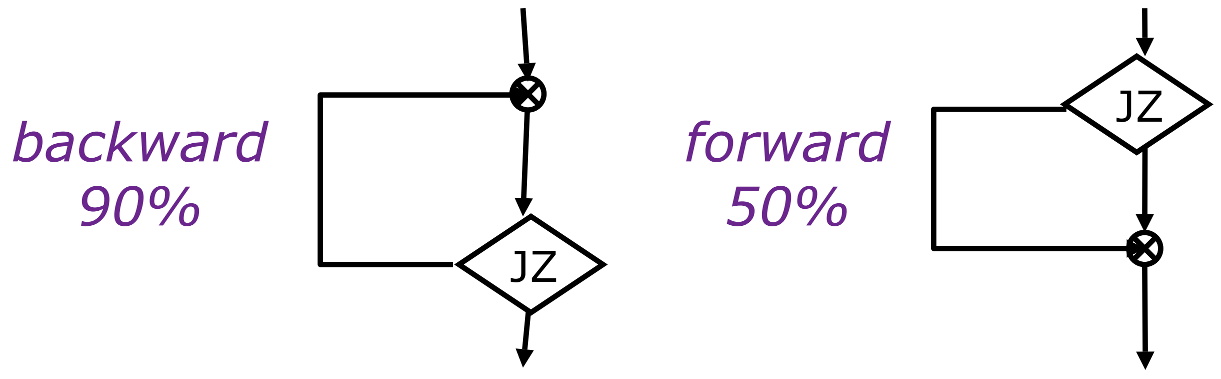
- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch

Static Branch Prediction

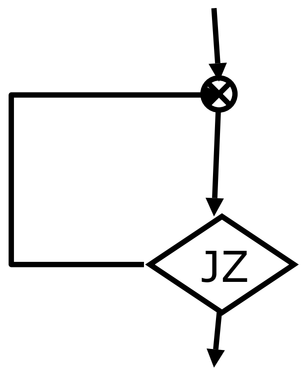
Overall probability a branch is taken is ~60-70% but:



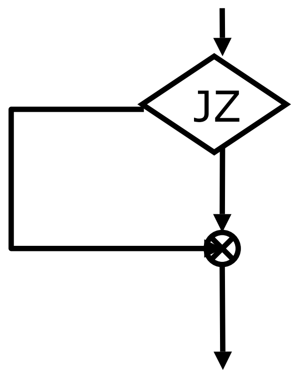
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%

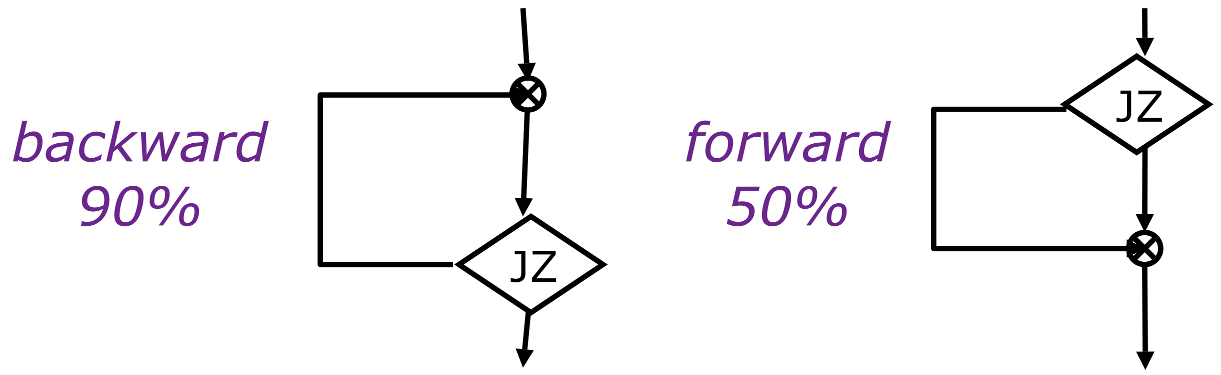


ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:



ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
typically reported as ~80% accurate

Dynamic Branch Prediction:
learning based on past behavior

Temporal correlation

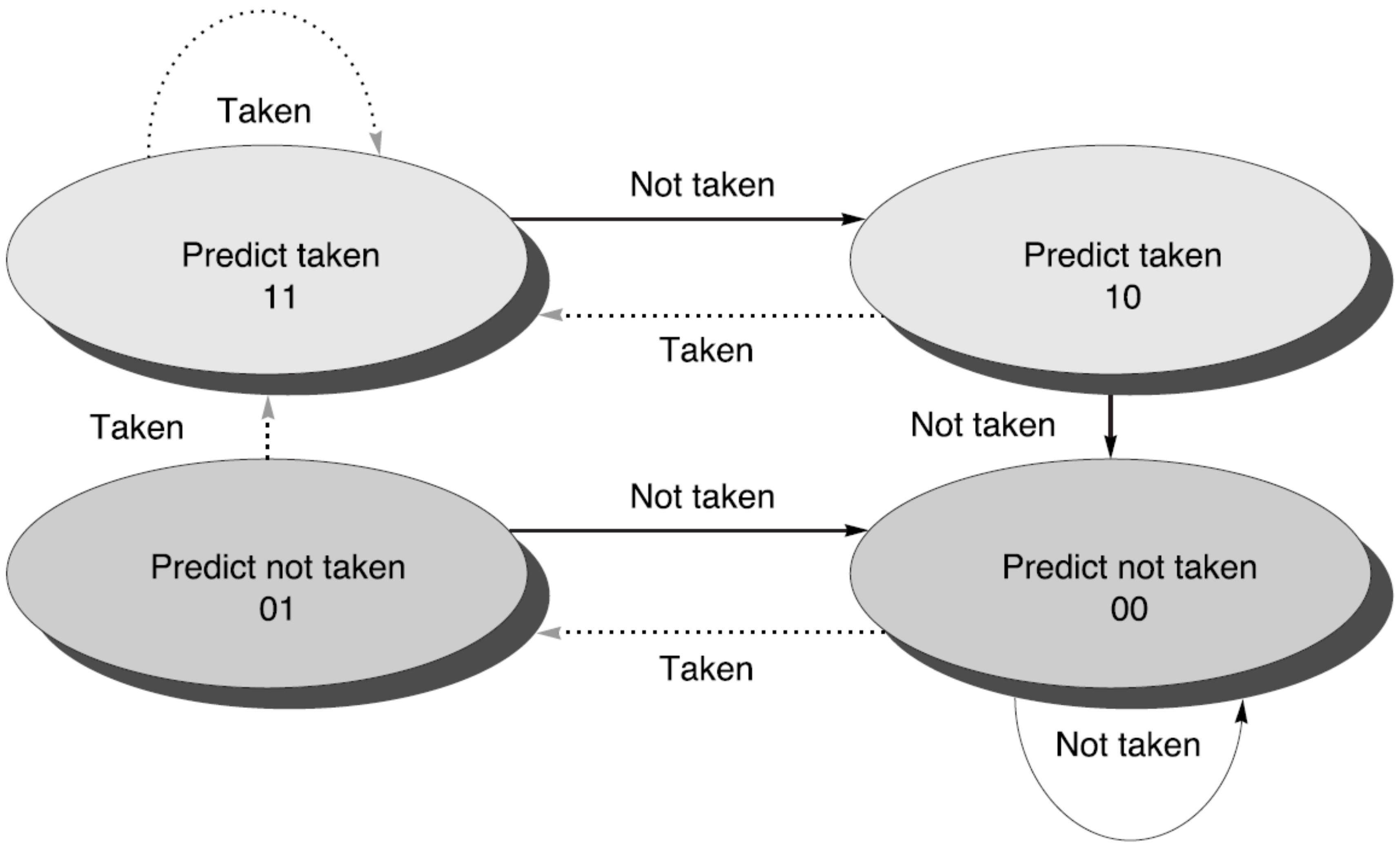
The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*

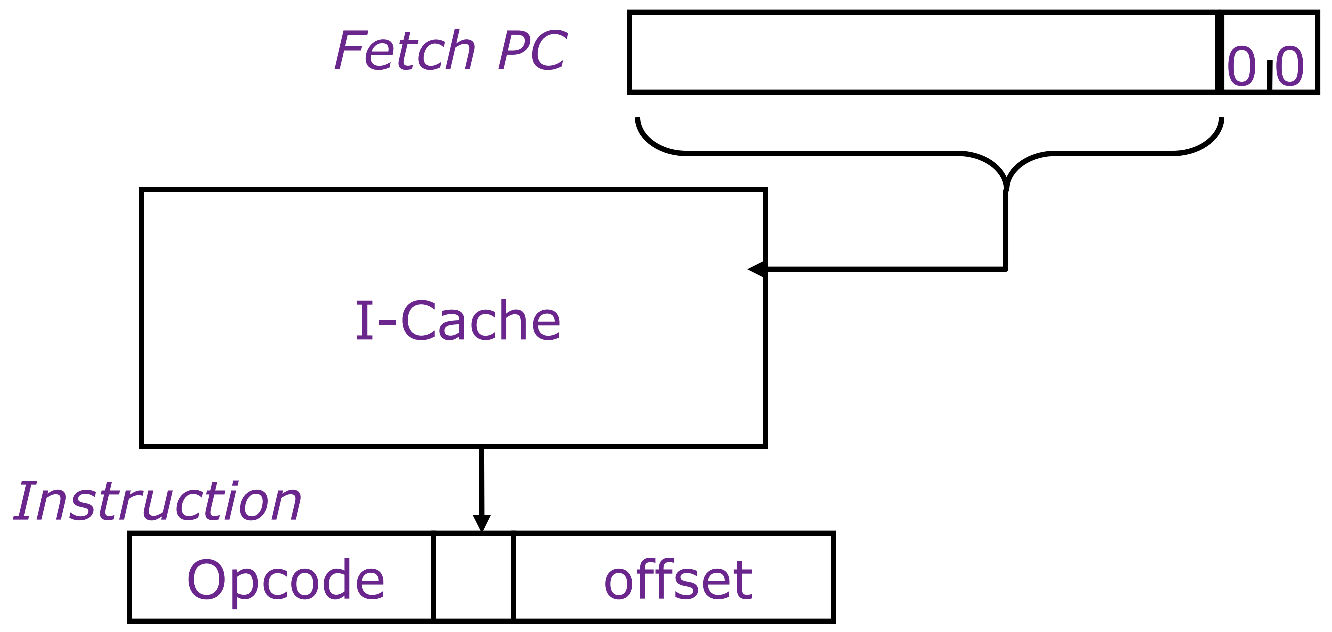
Branch Prediction Bits

- ☑ Assuming 2-bit predictor (saturating counter)

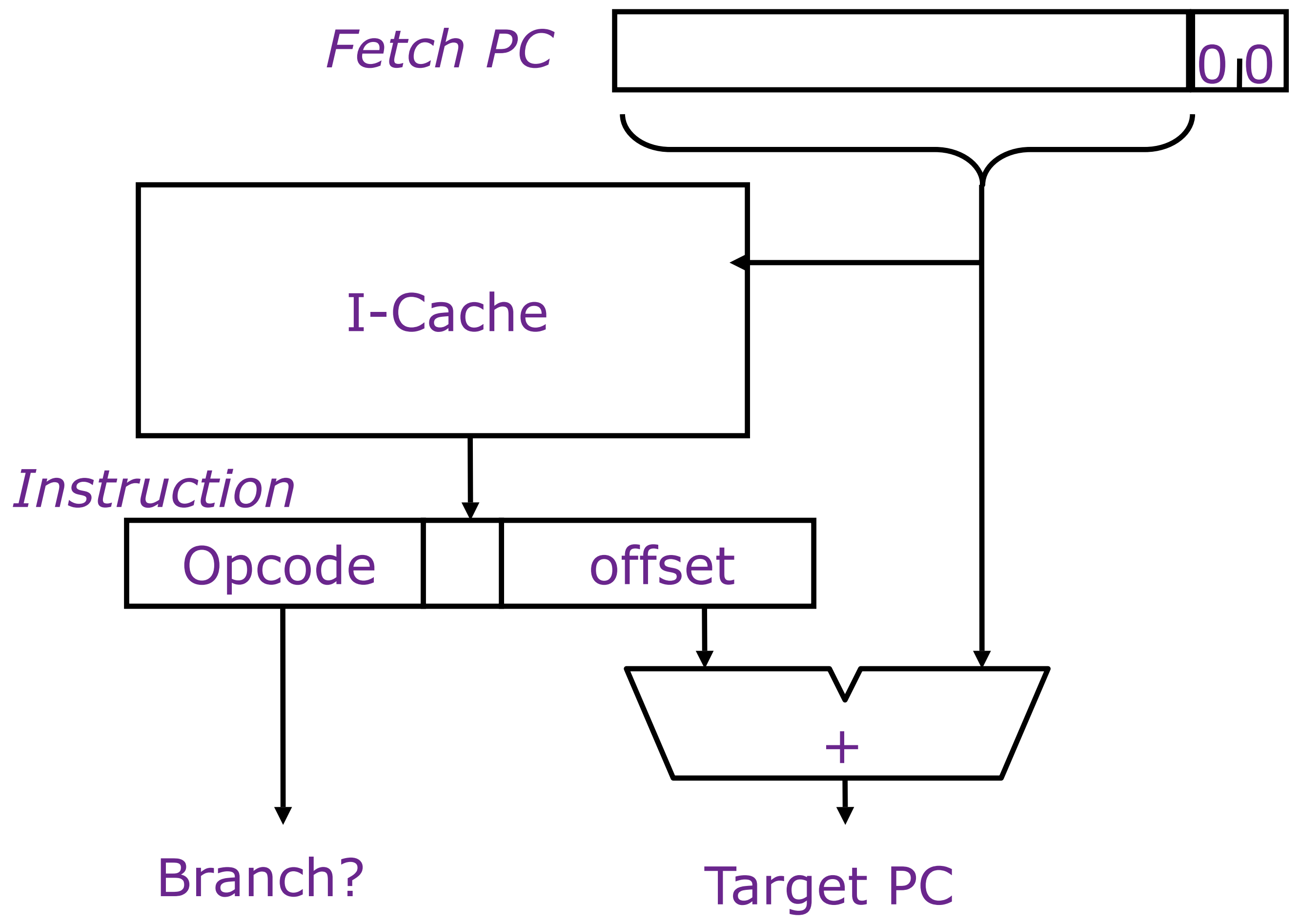


Branch History Table

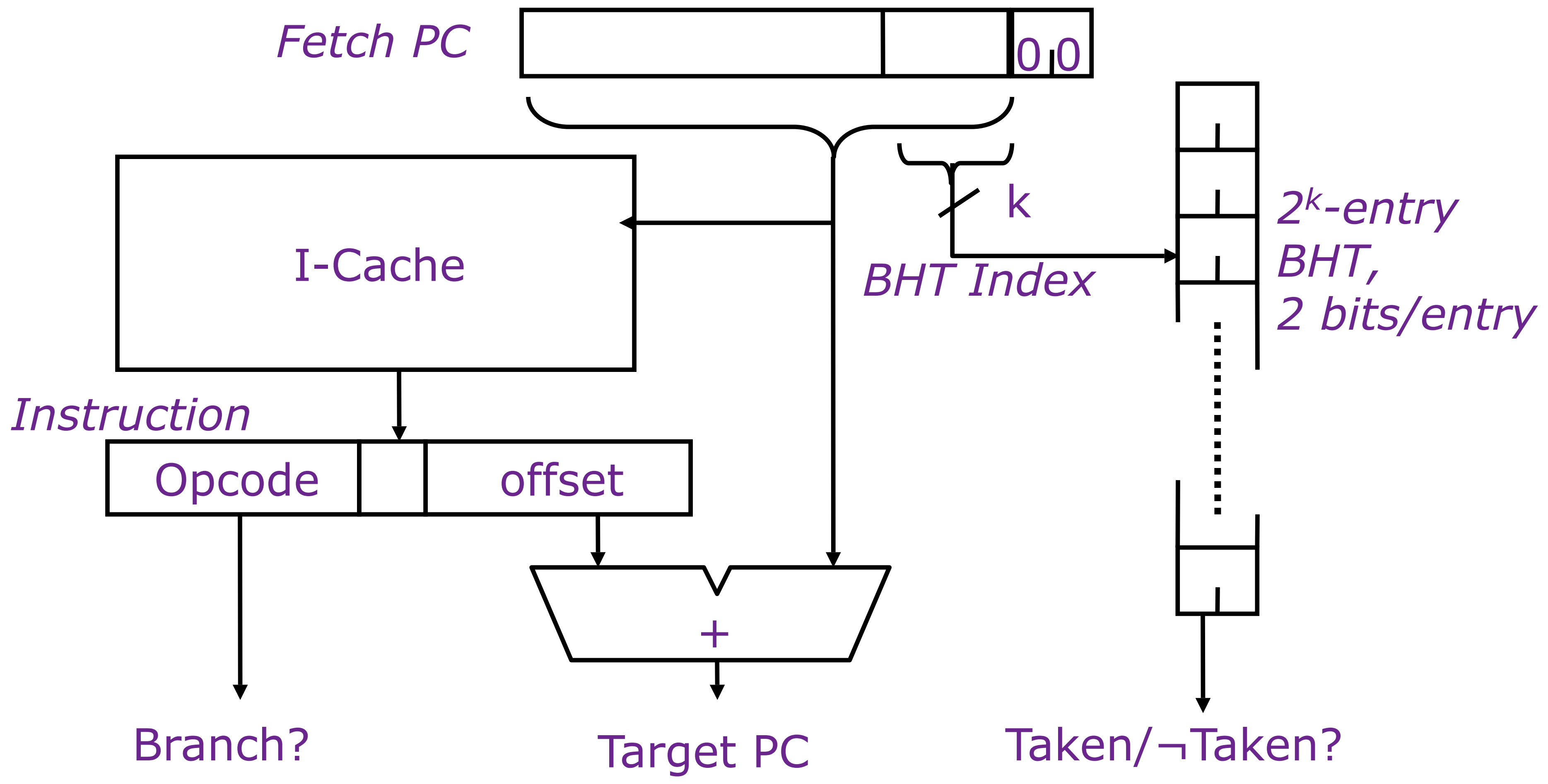
Branch History Table



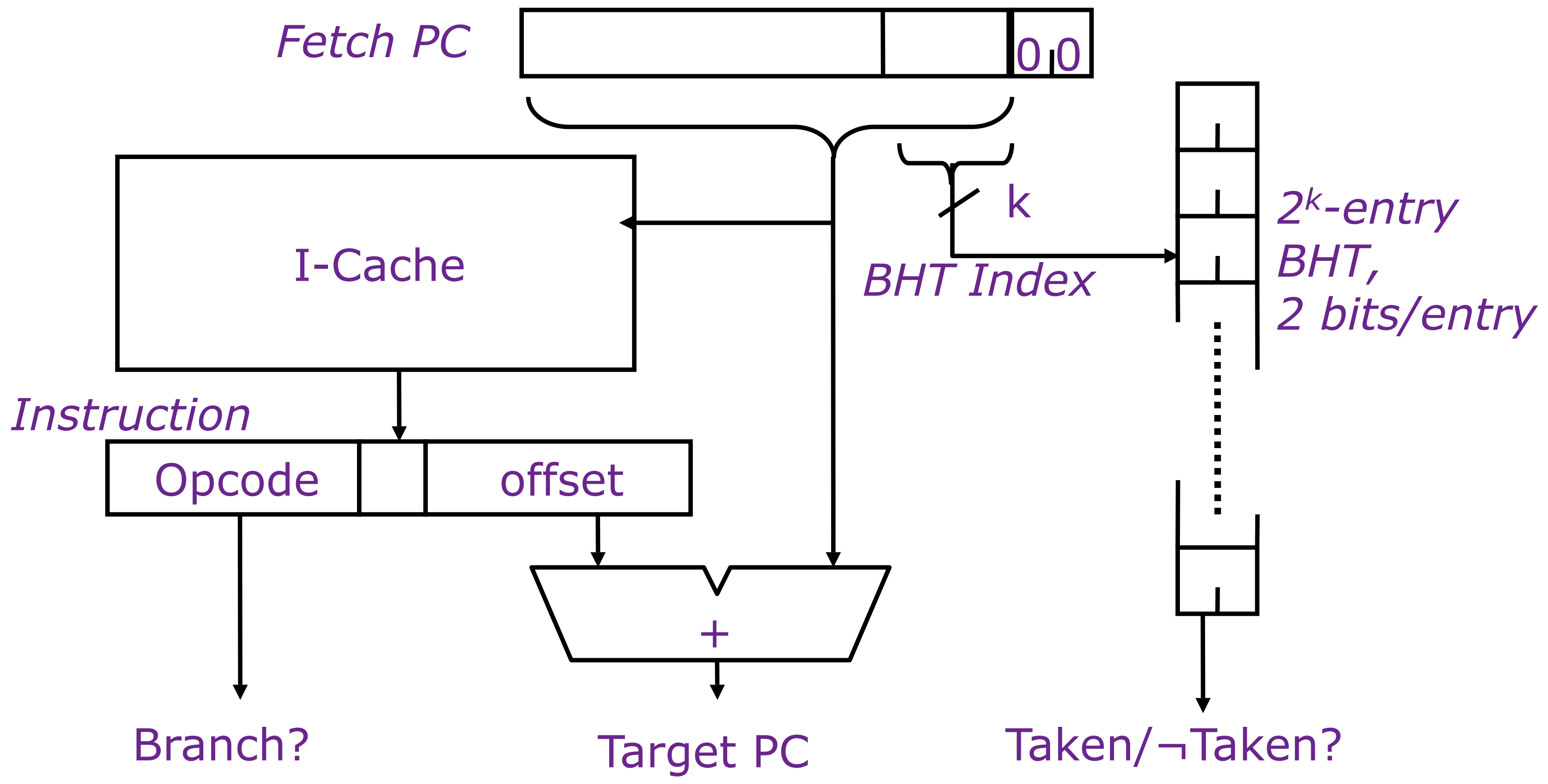
Branch History Table



Branch History Table



Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploiting Spatial Correlation

Yeh and Patt, 1992

```
if (x[i] < 7) then
      y += 1;
if (x[i] < 5) then
      c -= 4;
```

If first condition false, second condition also false

Exploiting Spatial Correlation

Yeh and Patt, 1992

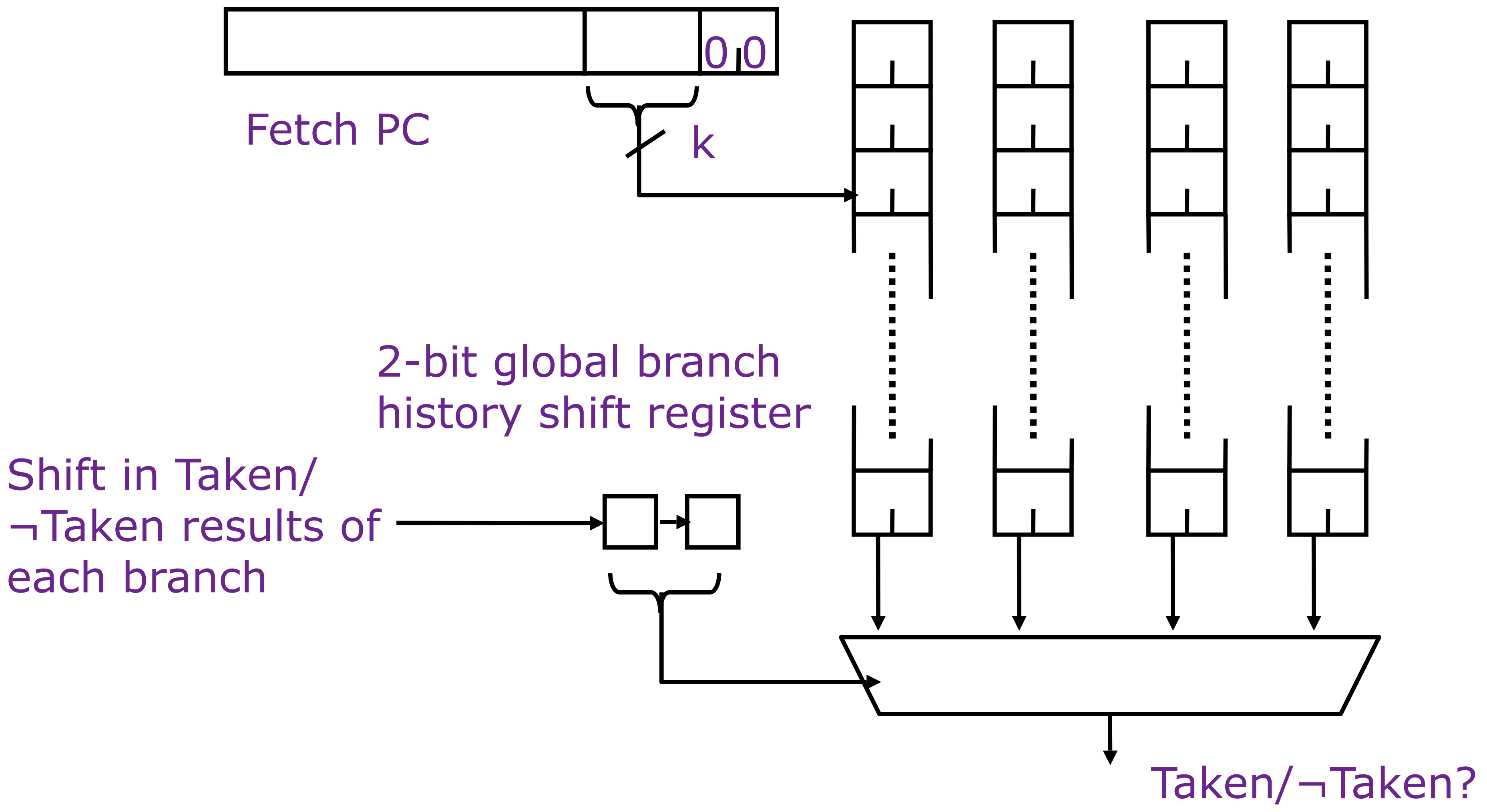
```
if (x[i] < 7) then
      y += 1;
if (x[i] < 5) then
      c -= 4;
```

If first condition false, second condition also false

History register, H, records the direction of the last N branches executed by the processor

Two-Level Branch Predictor

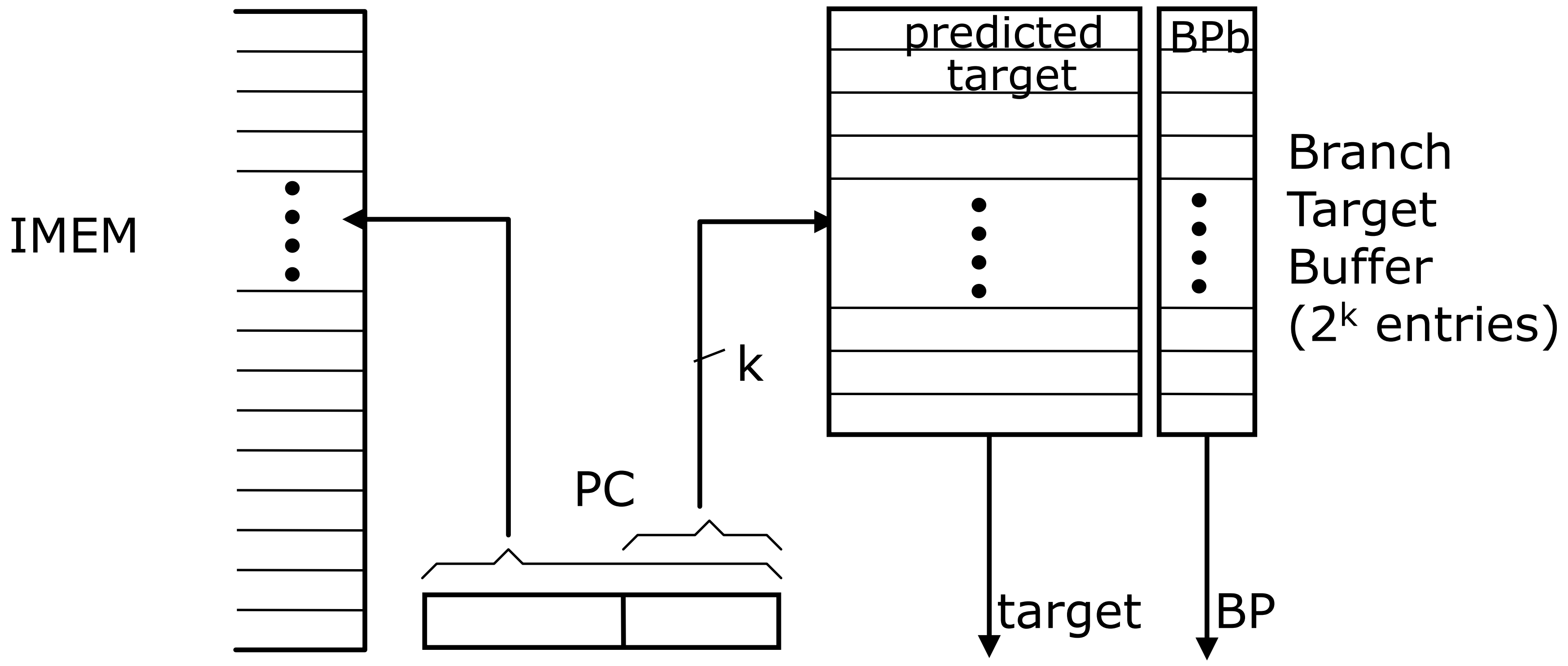
Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



0 0
Fetch PC
k
2-bit global branch history shift register
Shift in Taken/ ¬Taken results of each branch
Taken/¬Taken?

What About Target Address?

What About Target Address?



BP bits are stored with the predicted target address.

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Subroutine Address Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

Subroutine Address Stack

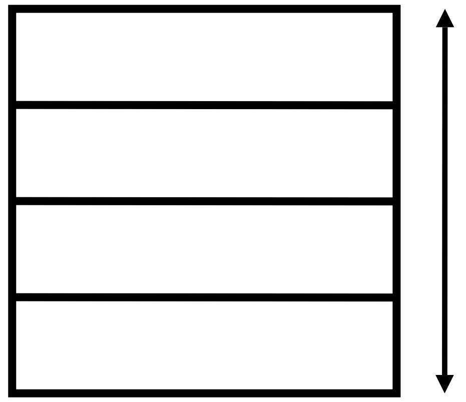
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```

Subroutine Address Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```

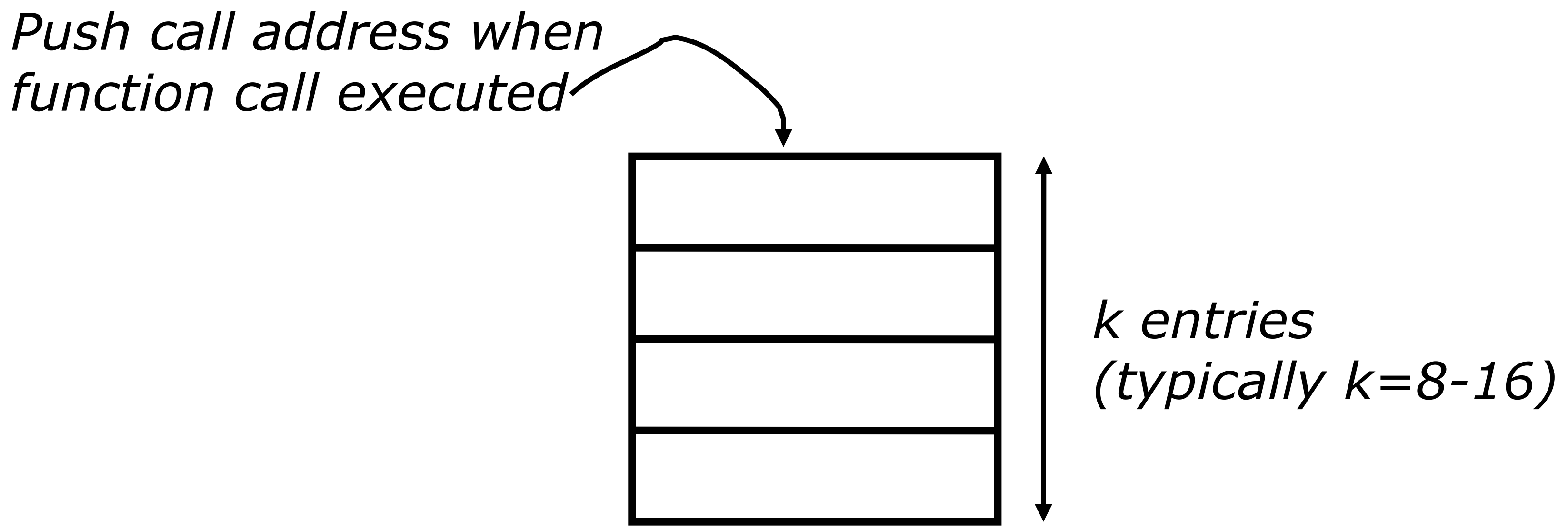


k entries (typically k=8-16)

Subroutine Address Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

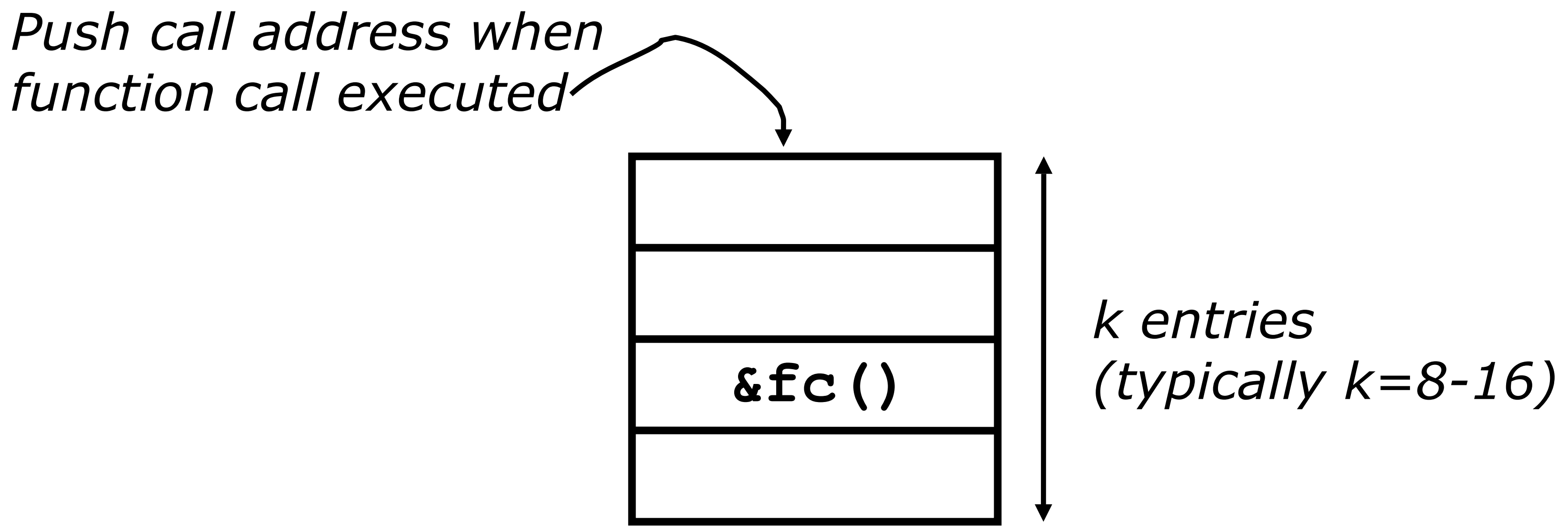
```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



Subroutine Address Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

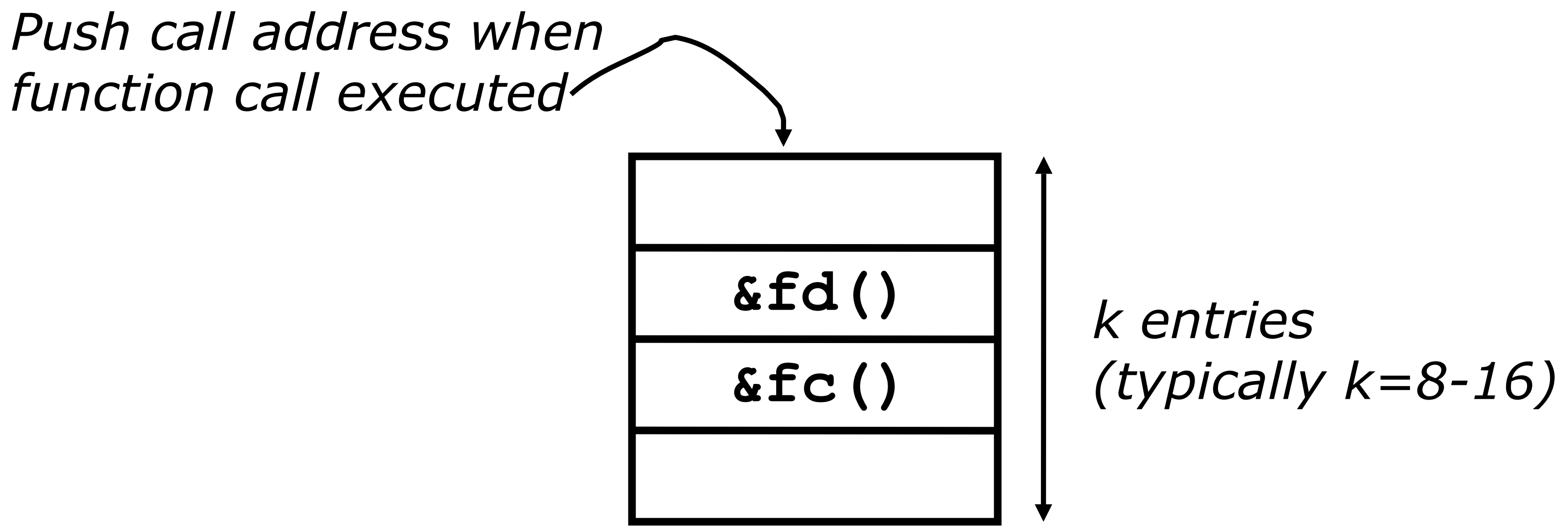
```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```


Push call address when function call executed
&fc()
k entries (typically k=8-16)

Subroutine Address Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

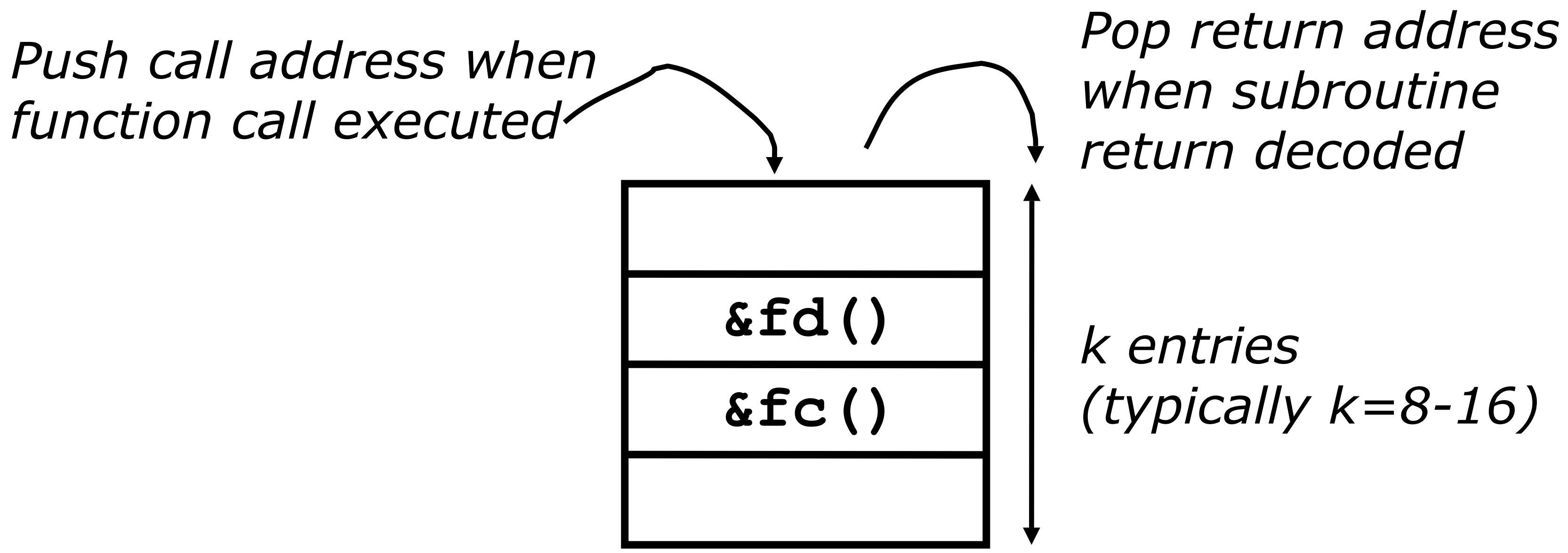
```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```



Subroutine Address Stack

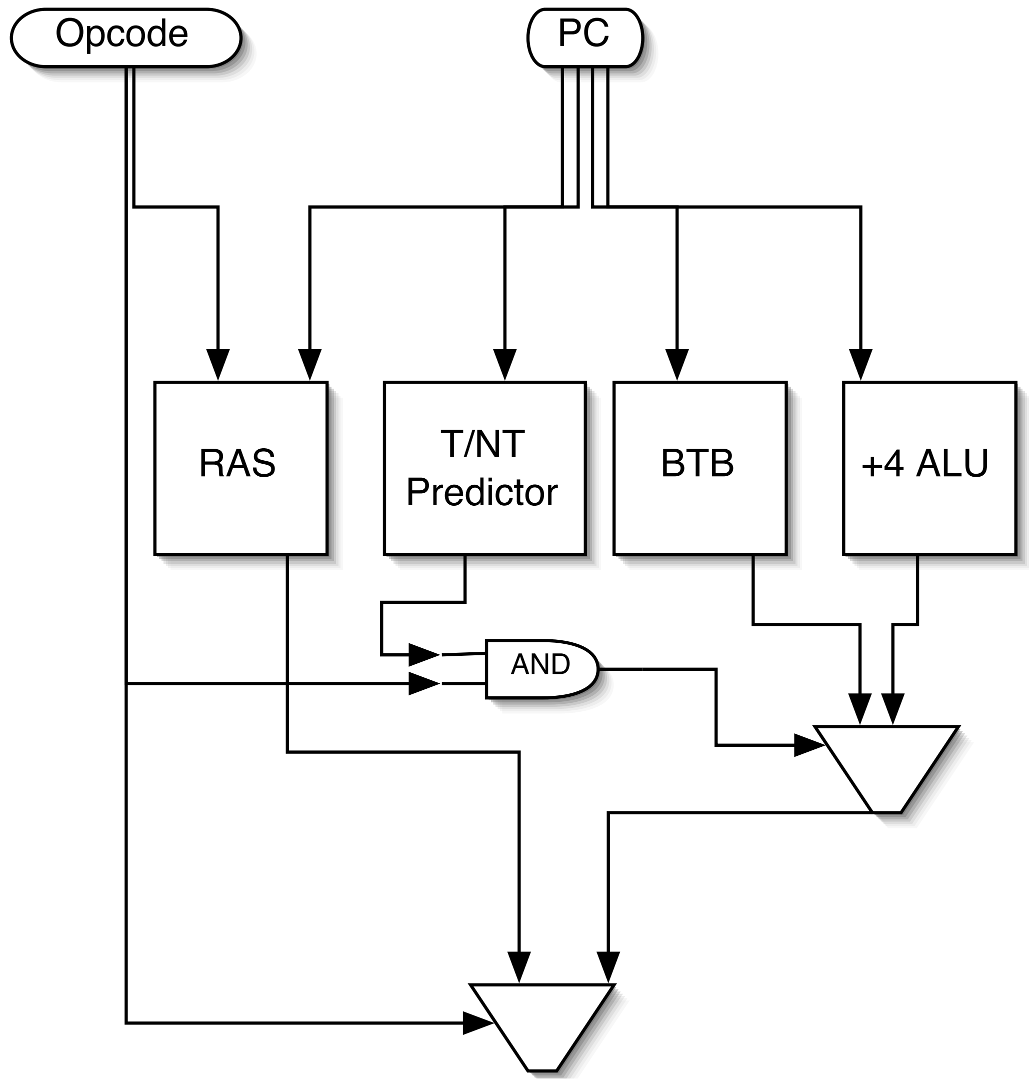
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }

fb() { fc(); }

fc() { fd(); }
```



Push call address when function call executed
Pop return address when subroutine return decoded
&fd()
&fc()
k entries (typically k=8-16)

Putting It Together



Misprediction Recovery

In-order execution machines:

- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Misprediction Recovery

In-order execution machines:

– Assume no instruction issued after branch can write-back before branch resolves

– Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

Misprediction Recovery

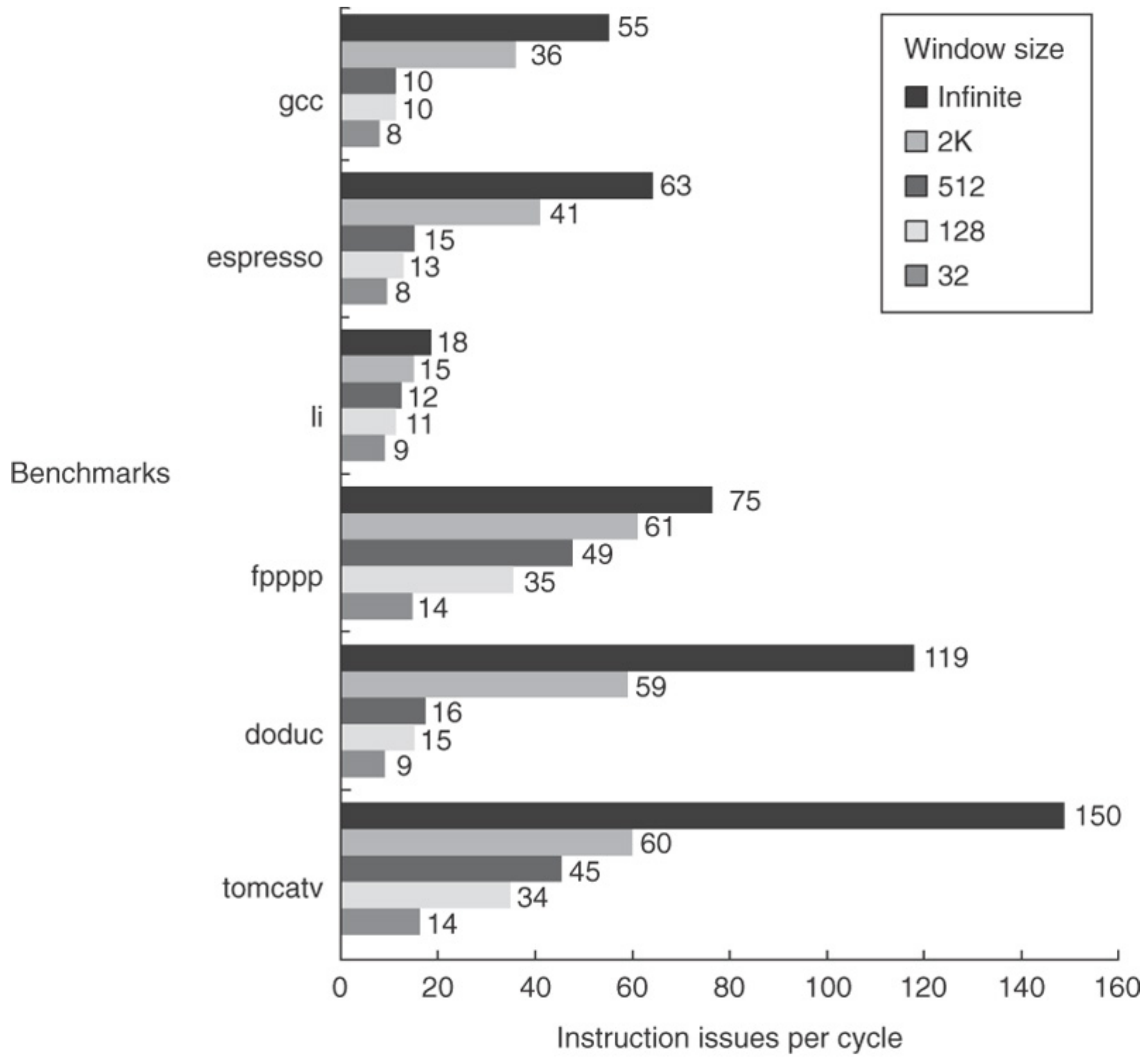
In-order execution machines:

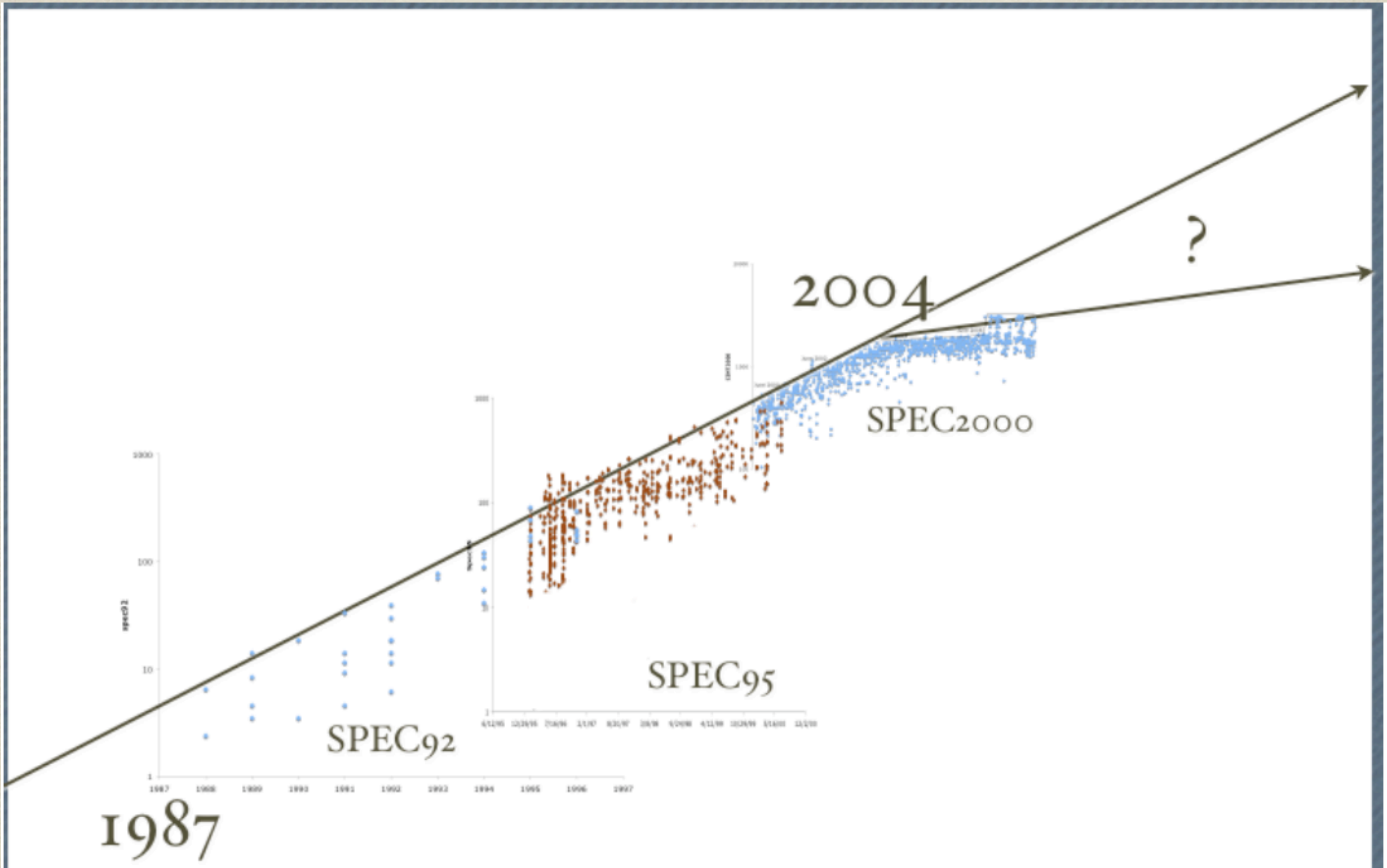
- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

- Multiple instructions following branch in program order can complete before branch resolves

Limits of ILP





?
2004
SPEC2000
SPEC95
SPEC92
1987
spec92

Flynn's Taxonomy

M.J. Flynn, "Very High-Speed Computers", *Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

- Flynn classified by data and control streams in 1966

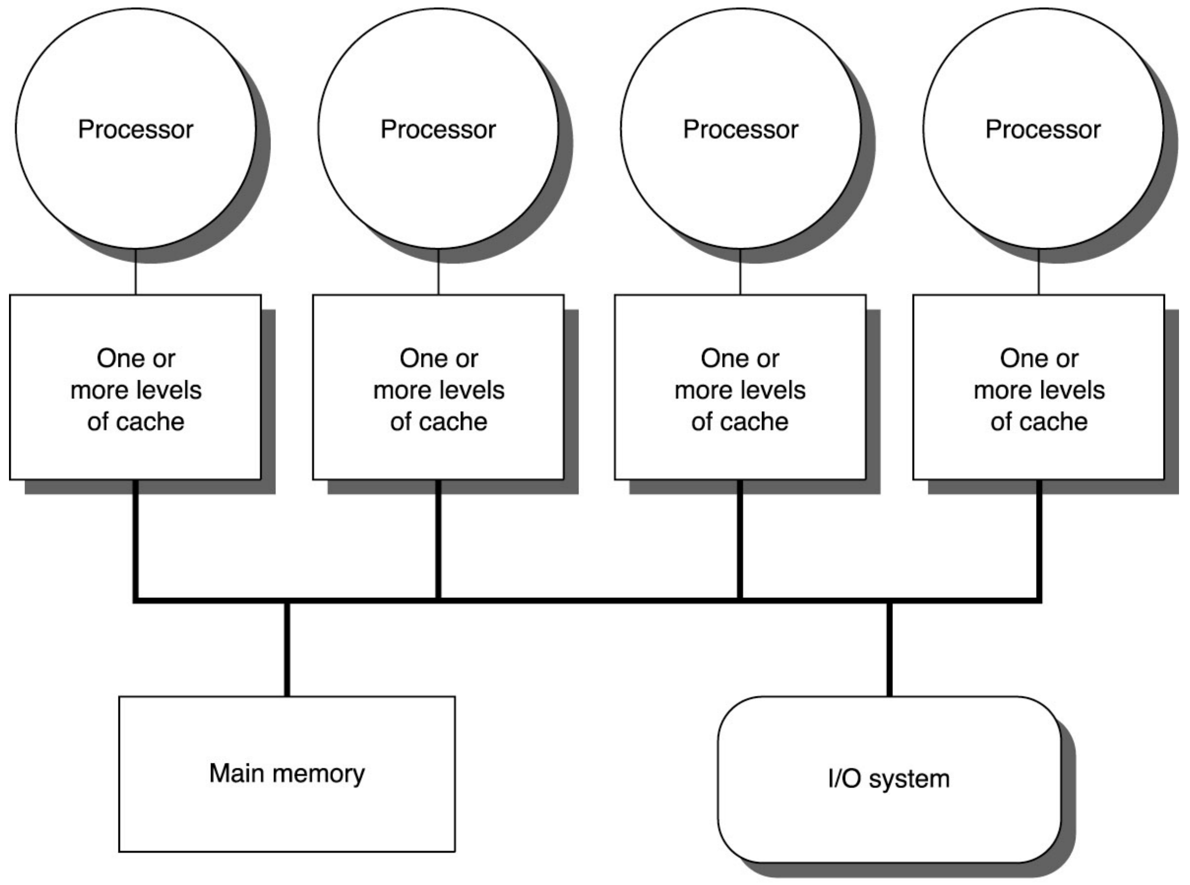
Single Instruction, Single Data (SISD) (Uniprocessor)	Single Instruction, Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction, Single Data (MISD) (Stream?)	Multiple Instruction, Multiple Data MIMD (Clusters, SMP servers)

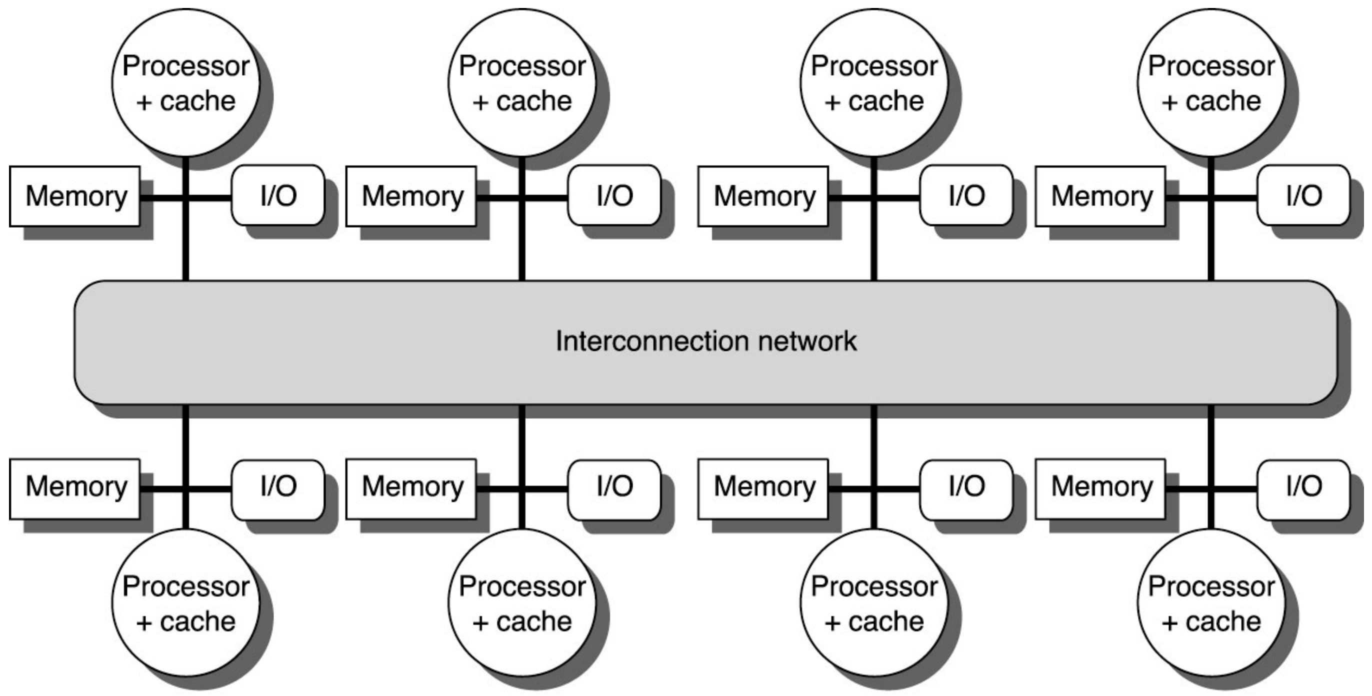
- SIMD ⇒ Data-Level Parallelism
- MIMD ⇒ Thread-Level Parallelism
- MIMD popular because
  - Flexible: N programs or 1 multithreaded program
  - Cost-effective: same MPU in desktop & MIMD machine

MIMD Multiprocessors

Centralized Shared Memory

Distributed Shared Memory


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system



Centralized-Memory Machines

- Also “Symmetric Multiprocessors” (SMP)
- “Uniform Memory Access” (UMA)
  - –All memory locations have similar latencies
  - –Data sharing through memory reads/writes
  - –P1 can write data to a physical address A,
    P2 can then read physical address A to get that data

- Problem: Memory Contention
  - –All processor share the one memory
  - –Memory bandwidth becomes bottleneck
  - –Used only for smaller machines
    - »Most often 2,4, or 8 processors

Distributed-Memory Machines

- Two kinds
  - Distributed Shared-Memory (DSM)
    - » All processors can address all memory locations
    - » Data sharing like in SMP
    - » Also called NUMA (non-uniform memory access)
    - » Latencies of different memory locations can differ (local access faster than remote access)
  - Message-Passing
    - » A processor can directly address only local memory
    - » To communicate with other processors, must explicitly send/receive messages
    - » Also called multicomputers or clusters
- Most accesses local, so less memory contention (can scale to well over 1000 processors)

Message-Passing Machines

- A cluster of computers
  - Each with its own processor and memory
  - An interconnect to pass messages between them
  - Producer-Consumer Scenario:
    - P1 produces data D, uses a SEND to send it to P2
    - The network routes the message to P2
    - P2 then calls a RECEIVE to get the message
  - Two types of send primitives
    - Synchronous: P1 stops until P2 confirms receipt of message
    - Asynchronous: P1 sends its message and continues
  - Standard libraries for message passing:
    Most common is MPI – Message Passing Interface

Shared Memory vs. Message Passing

Shared memory

+ simple parallel programming model
  - » global shared address space
  - » not worry about data locality ***but***

    get better performance when program for data placement

    lower latency when data is local

    - ***but*** can do data placement if it is crucial, but don't have to
  - » hardware maintains data coherence
    - synchronize to order processor's accesses to shared data
  - » like uniprocessor code so parallelizing by programmer or compiler is easier

⇒ can focus on program semantics, not interprocessor communication

Shared Memory vs. Message Passing

Shared memory

- + low latency (no message passing software) ***but***
    - *overlap of communication & computation*
    - *latency-hiding techniques can be applied to message passing machines*
- + higher bandwidth for small transfers ***but***
    - *usually the only choice*

Shared Memory vs. Message Passing

Message passing

- + abstraction in the programming model encapsulates the communication costs ***but***
  - *more complex programming model*
  - *additional language constructs*
  - *need to program for nearest neighbor communication*
- + no coherency hardware
- + good throughput on large transfers ***but***
  - *what about small transfers?*
- + more scalable (memory latency doesn't scale with the number of processors) ***but***
  - *large-scale SM has distributed memory also*
    - ***hah!*** so you're going to adopt the message-passing model?

Shared Memory vs. Message Passing

Why there was a debate

- little experimental data
- not separate implementation from programming model
- can emulate one paradigm with the other
  - » MP on SM machine
    message buffers in local (to each processor) memory
    copy messages by ld/st between buffers
  - » SM on MP machine
    ld/st becomes a message copy
    sloooooooooow

Who won?

Challenges of Parallel Processing

- Big challenge is % of program that is inherently sequential
  - What does it mean to be inherently sequential?
- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?

  a. 10%

  b. 5%

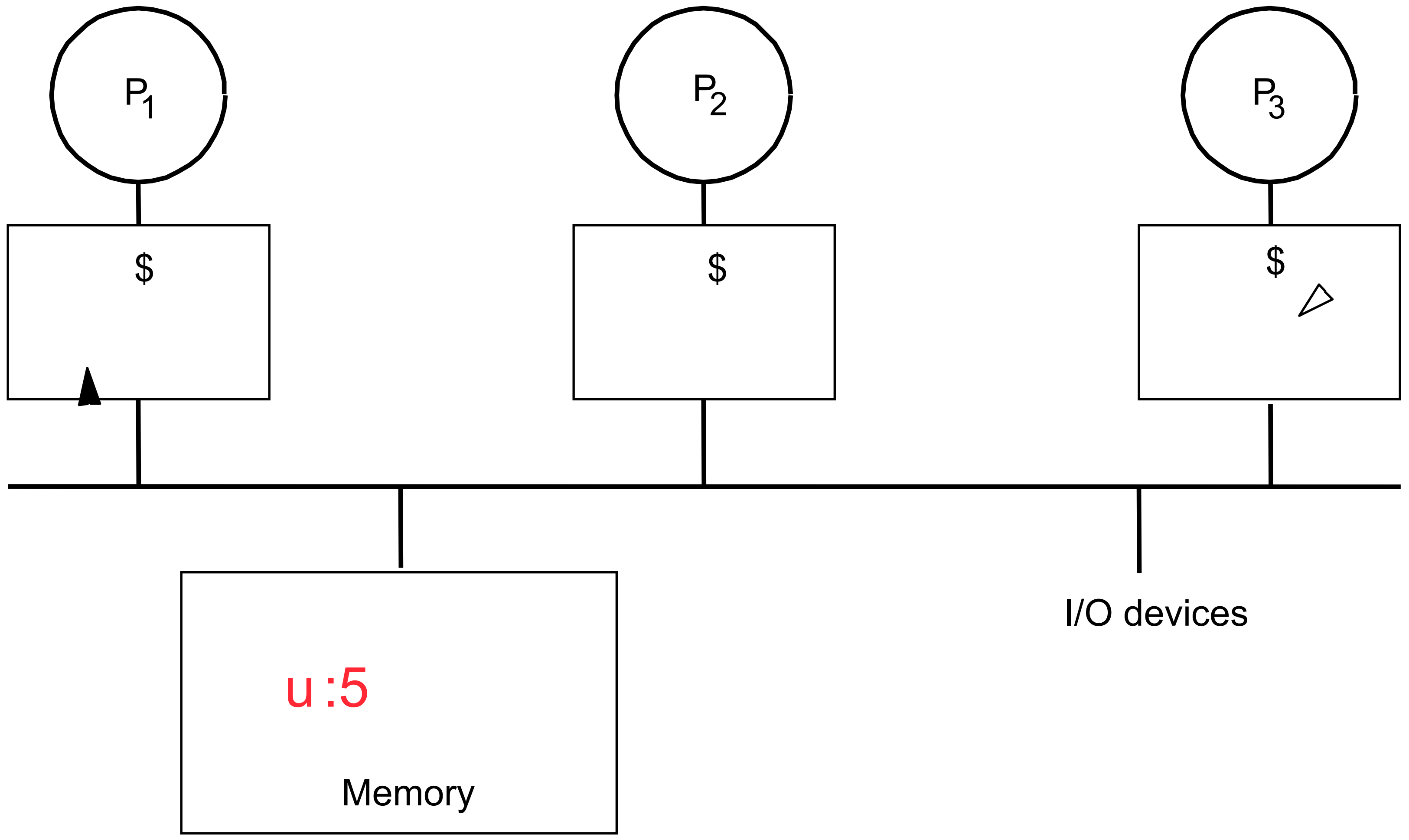
  c. 1%

  d. <1%

Symmetric Shared-Memory Architectures

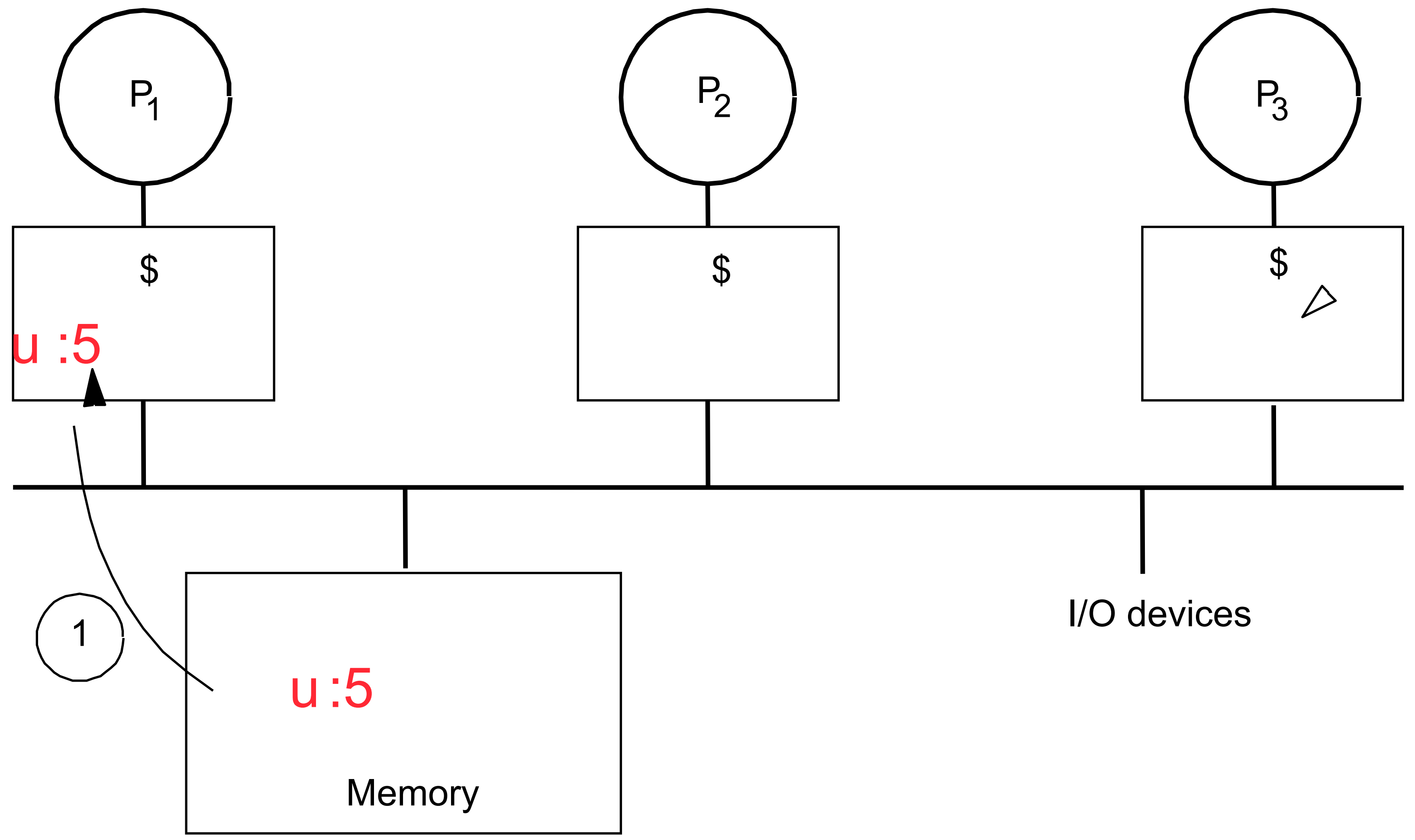
- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data
  $\Rightarrow$ reduces latency to shared data, memory bandwidth for shared data,
  and interconnect bandwidth
  $\Rightarrow$ cache coherence problem

Example Cache Coherence Problem



– Processors see different values for u after event 3
– With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  » Processes accessing main memory may see very stale value
– Unacceptable for programming, and its frequent!

Example Cache Coherence Problem



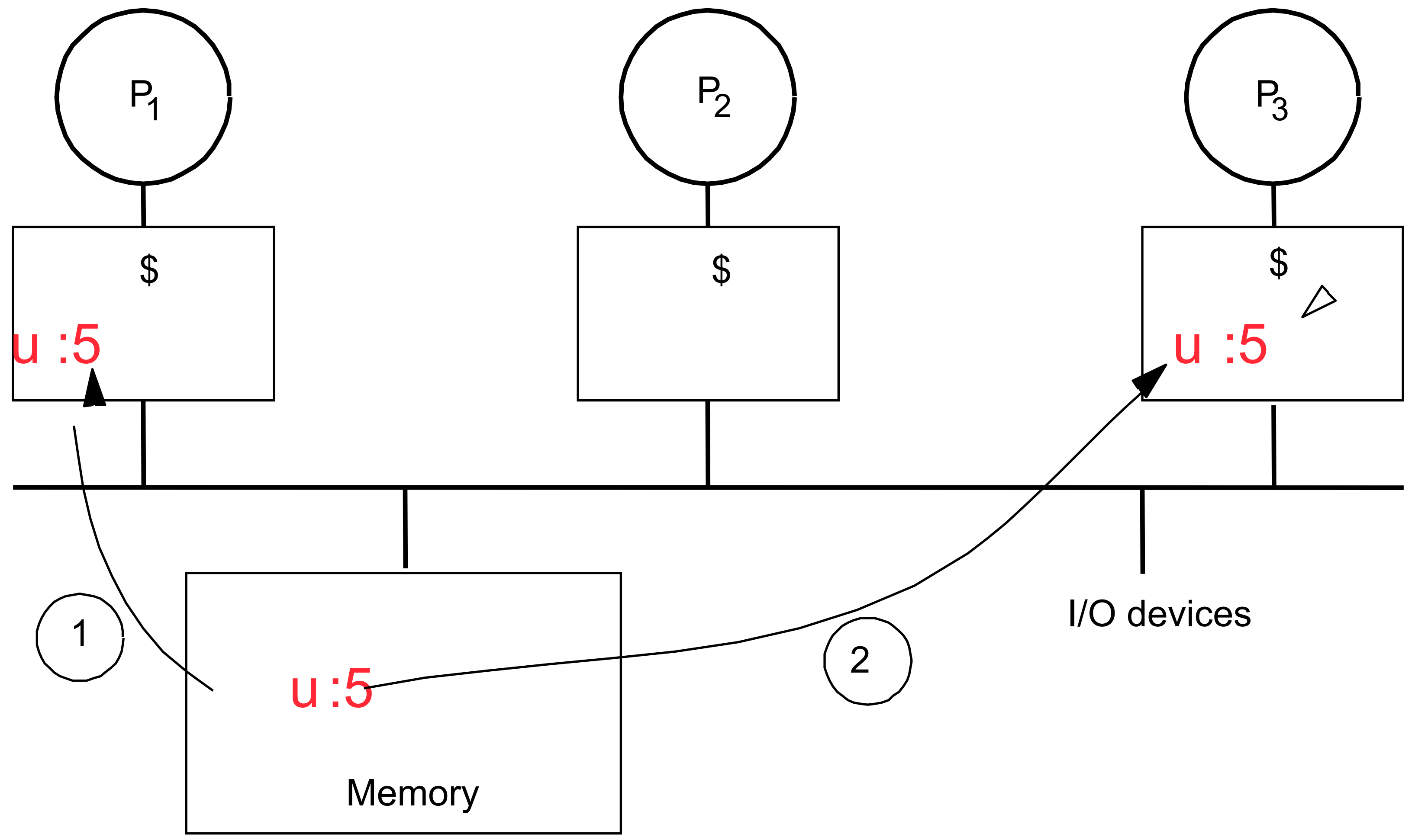
–Processors see different values for u after event 3

–With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when

»Processes accessing main memory may see very stale value

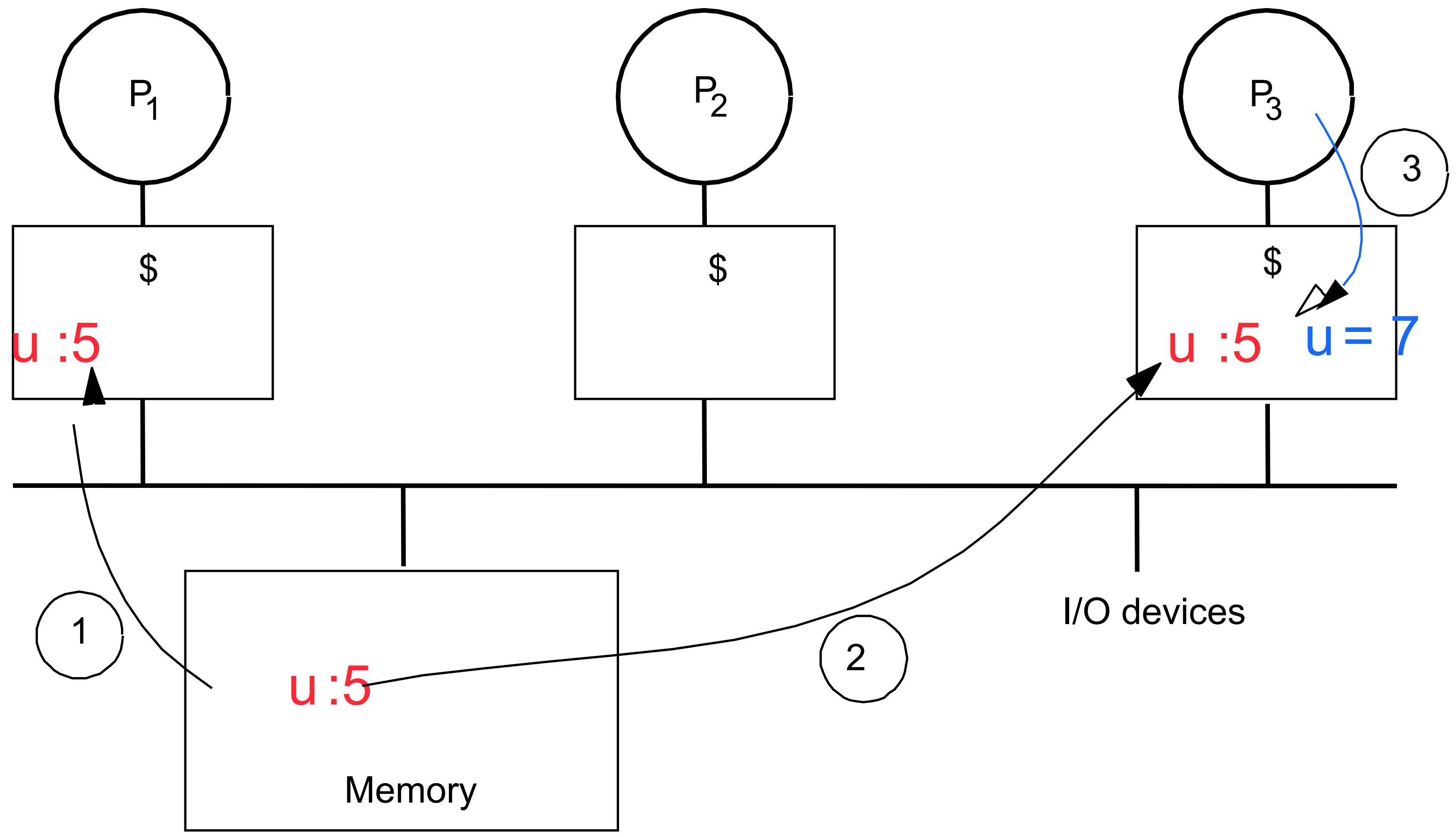
–Unacceptable for programming, and its frequent!

Example Cache Coherence Problem



– Processors see different values for u after event 3
– With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  » Processes accessing main memory may see very stale value
– Unacceptable for programming, and its frequent!

Example Cache Coherence Problem



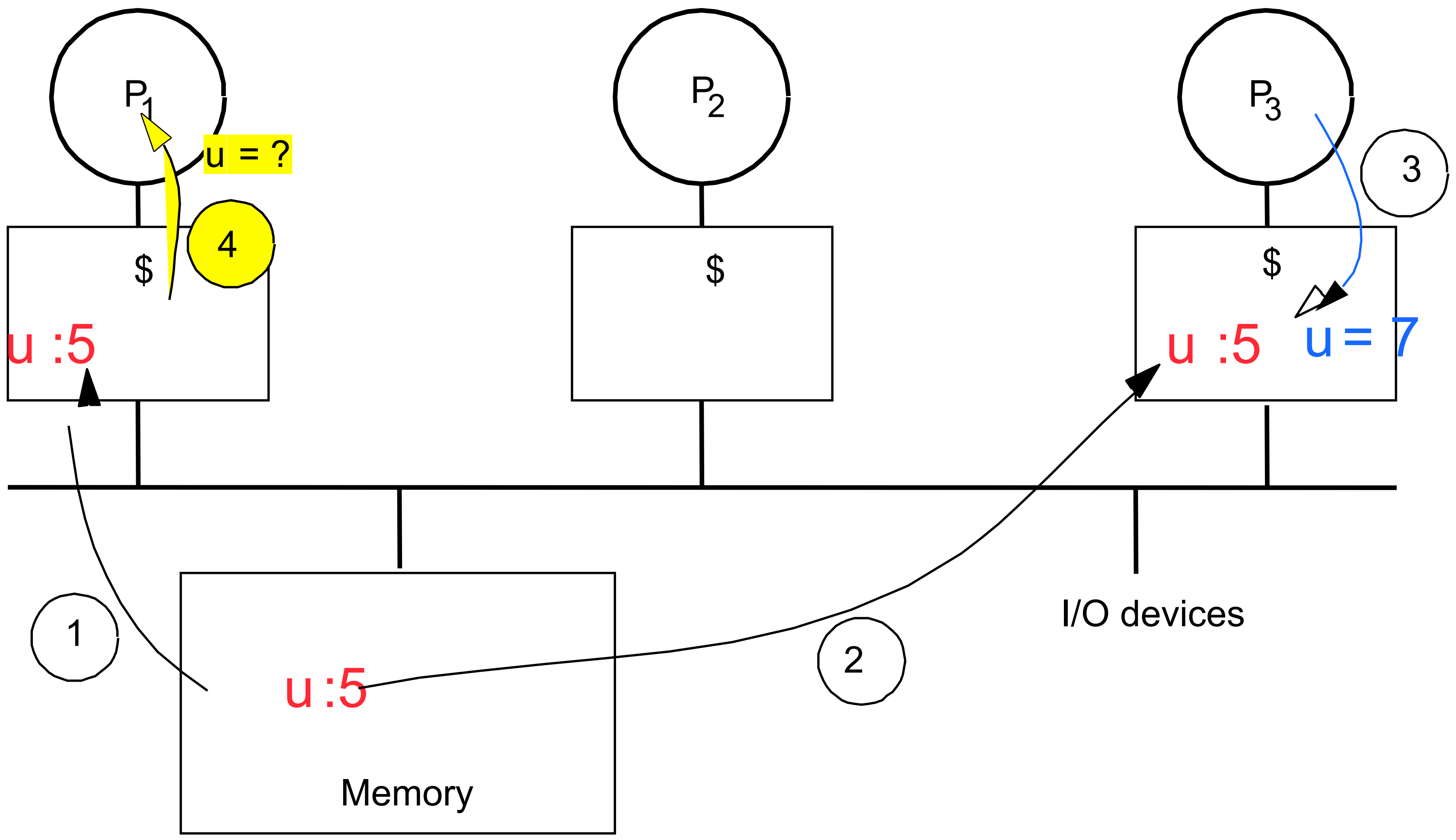
–Processors see different values for u after event 3

–With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when

»Processes accessing main memory may see very stale value

–Unacceptable for programming, and its frequent!

Example Cache Coherence Problem



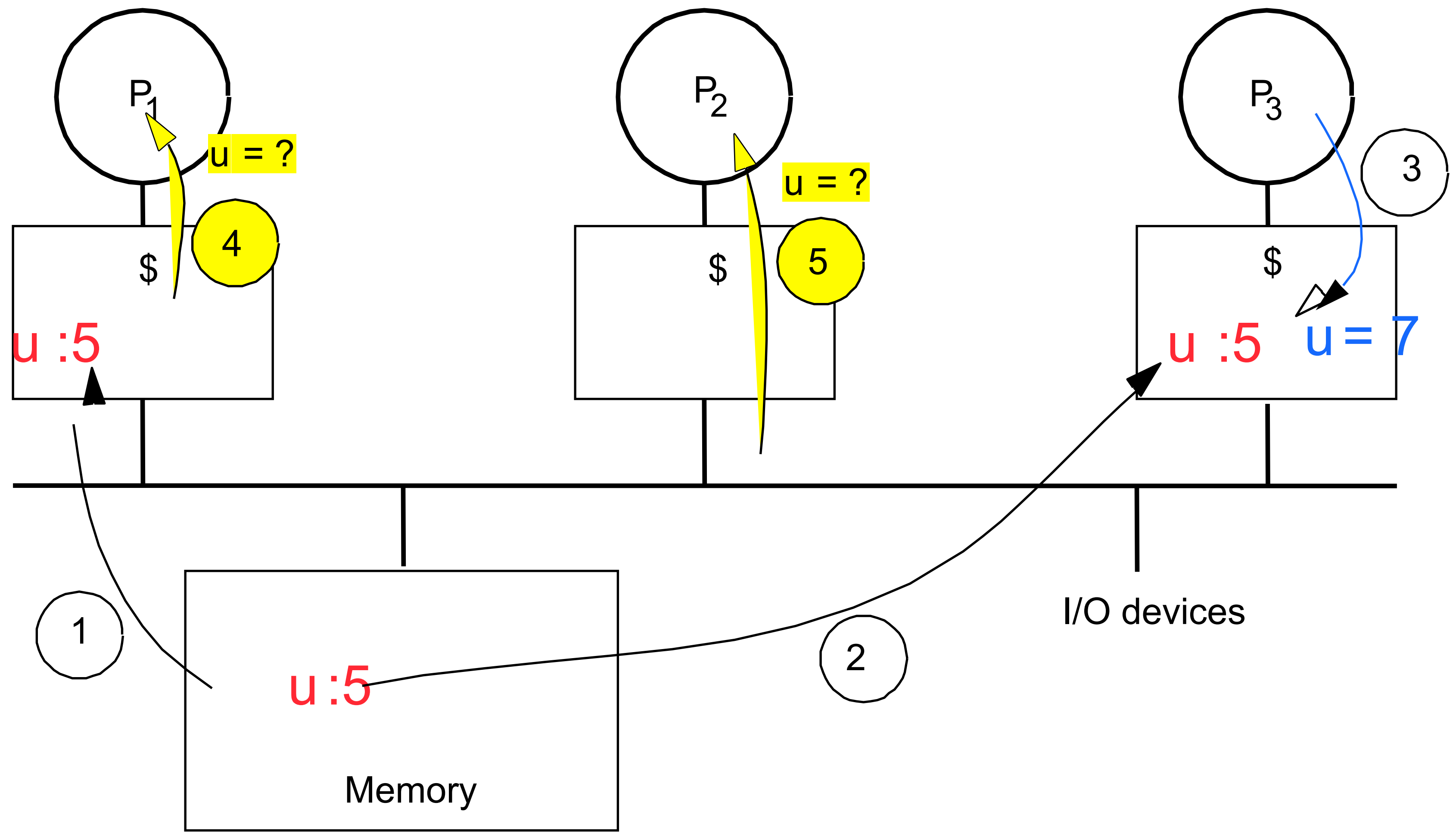
– Processors see different values for u after event 3

– With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when

  » Processes accessing main memory may see very stale value

– Unacceptable for programming, and its frequent!

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - »Processes accessing main memory may see very stale value
- Unacceptable for programming, and its frequent!

Cache Coherence Definition

- A memory system is coherent if
  1. A read R from address X on processor P1 returns the value written by the most recent write W to X on P1 if no other processor has written to X between W and R.
  2. If P1 writes to X and P2 reads X after a sufficient time, and there are no other writes to X in between, P2’s read returns the value written by P1’s write.
  3. Writes to the same location are serialized: two writes to location X are seen in the same order by all processors.

Cache Coherence Definition

- Property 1. preserves program order
  - It says that in the absence of sharing, each processor behaves as a uniprocessor would
- Property 2. says that any write to an address must eventually be seen by all processors
  - If P1 writes to X and P2 keeps reading X, P2 must eventually see the new value
- Property 3. preserves causality
  - Suppose X starts at 0. Processor P1 increments X and processor P2 waits until X is 1 and then increments it to 2. Processor P3 must eventually see that X becomes 2.
  - If different processors could see writes in different order, P2 can see P1’s write and do its own write, while P3 first sees the write by P2 and then the write by P1. Now we have two processors that will forever disagree about the value of A.

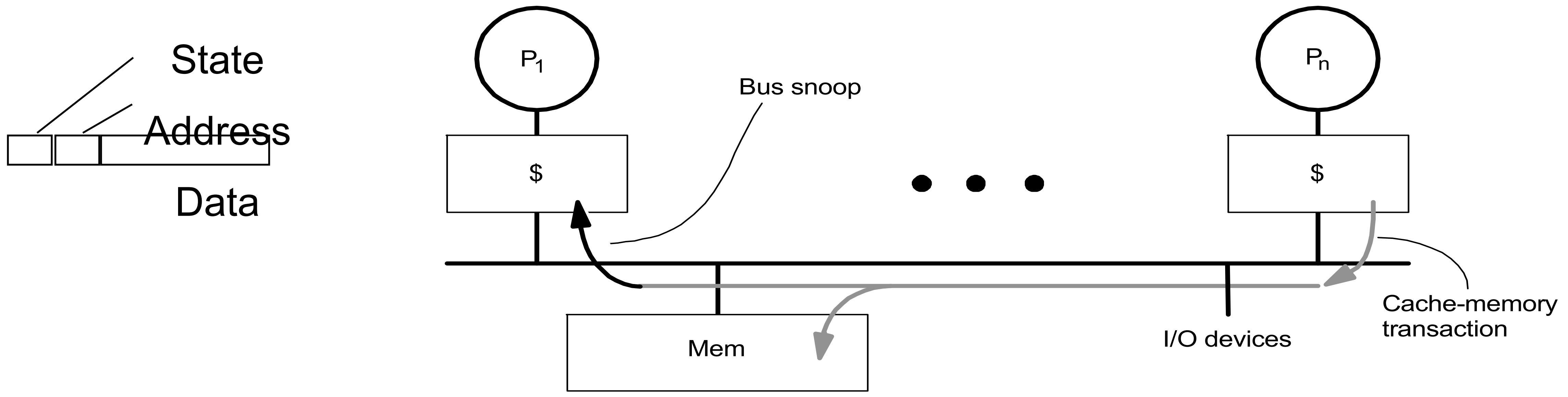
Maintaining Cache Coherence

- Hardware schemes
  - Shared Caches
    - Trivially enforces coherence
    - Not scalable (L1 cache quickly becomes a bottleneck)
  - Snooping
    - Needs a broadcast network (like a bus) to enforce coherence
    - Each cache that has a block tracks its sharing state on its own
  - Directory
    - Can enforce coherence even with a point-to-point network
    - A block has just one place where its full sharing state is kept

Snooping

- Typically used for bus-based (SMP) multiprocessors
  - Serialization on the bus used to maintain coherence property 3
- Two flavors
  - Write-update (write broadcast)
    - » A write to shared data is broadcast to update all copies
    - » All subsequent reads will return the new written value (property 2)
    - » All see the writes in the order of broadcasts
      One bus == one order seen by all (property 3)
  - Write-invalidate
    - » Write to shared data forces invalidation of all other cached copies
    - » Subsequent reads miss and fetch new value (property 2)
    - » Writes ordered by invalidations on the bus (property 3)

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update all copies on write

Snooping Implementation

How the bus is used

- broadcast medium (total ordering, yay)
- entire coherency operation is atomic wrt other processors
  - » **keep-the-bus protocol**: master holds the bus until the entire operation has completed
  - » **split-transaction buses**:
    - request & response are different phases
    - state value that indicates that an operation is in progress
    - do not initiate another operation for a cache block that has one in progress

Update vs. Invalidate

- A burst of writes by a processor to one addr
  - Update: each sends an update
  - Invalidate: possibly only the first invalidation is sent
- Writes to different words of a block
  - Update: update sent for each word
  - Invalidate: possibly only the first invalidation is sent
- Producer-consumer communication latency
  - Update: producer sends an update,
    consumer reads new value from its cache
  - Invalidate: producer invalidates consumer’s copy,
    consumer’s read misses and has to request the block
- Which is better depends on application
  - But write-invalidate is simpler and implemented in most MP-capable processors today

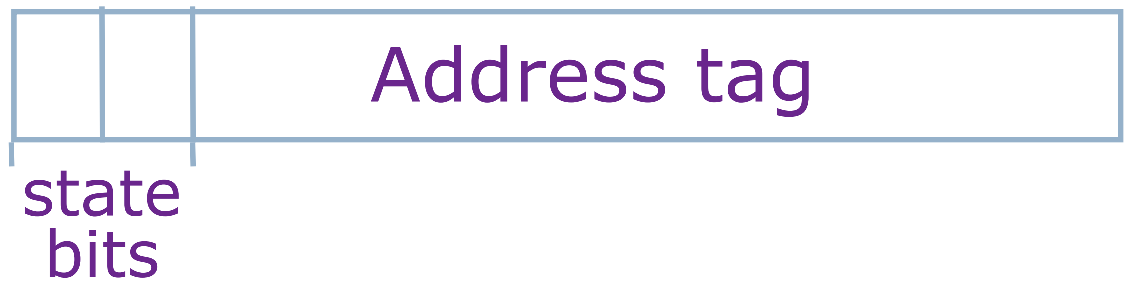
MSI Snoopy Protocol

- State of block B in cache C can be
  - Invalid: B is not cached in C
    - To read or write, must make a request on the bus
  - Modified: B is dirty in C
    - C has the block, no other cache has the block, and C must update memory when it displaces B
    - Can read or write B without going to the bus
  - Shared: B is clean in C
    - C has the block, other caches have the block, and C need not update memory when it displaces B
    - Can read B without going to bus
    - ***To write, must send an upgrade request to the bus***

Cache State Transition Diagram

The MSI protocol

Each cache line has a tag

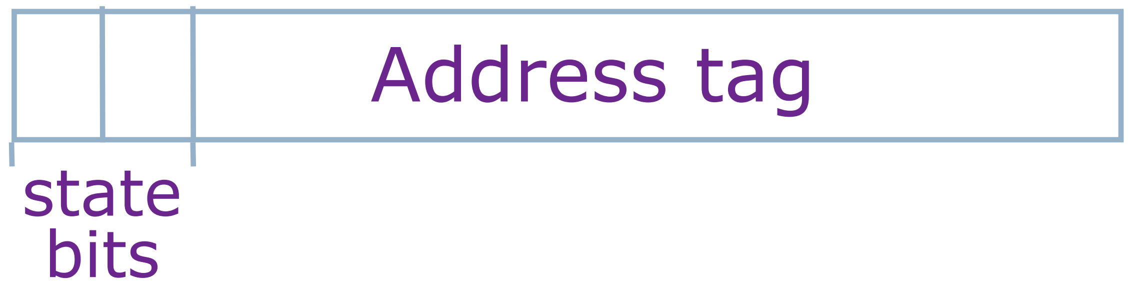


M: Modified
S: Shared
I: Invalid

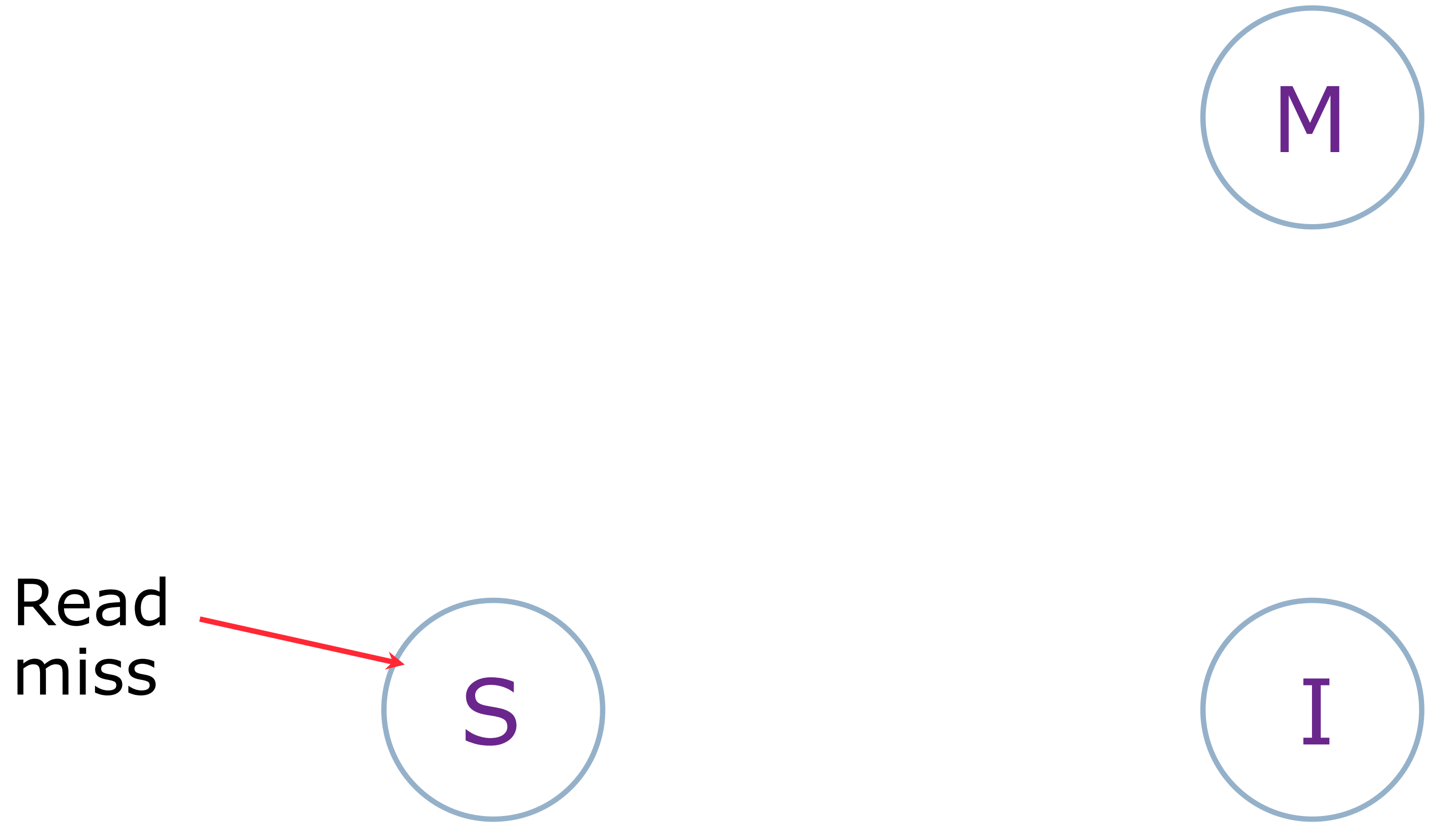
Cache state in processor P_1

Cache State Transition Diagram
The MSI protocol

Each cache line has a tag



M: Modified
S: Shared
I: Invalid

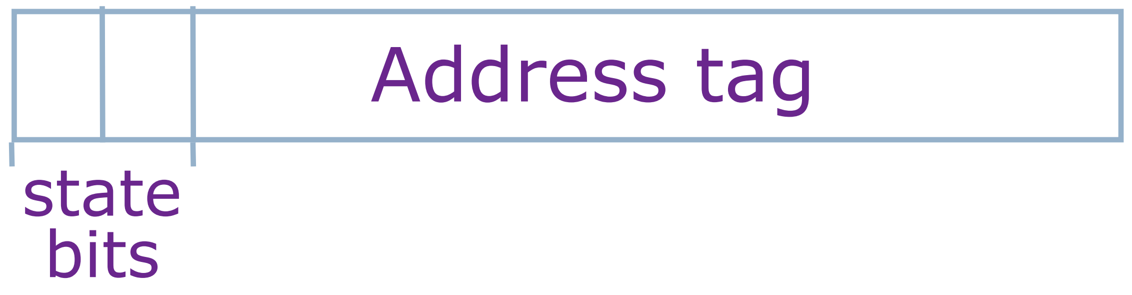


Cache state in processor P_1

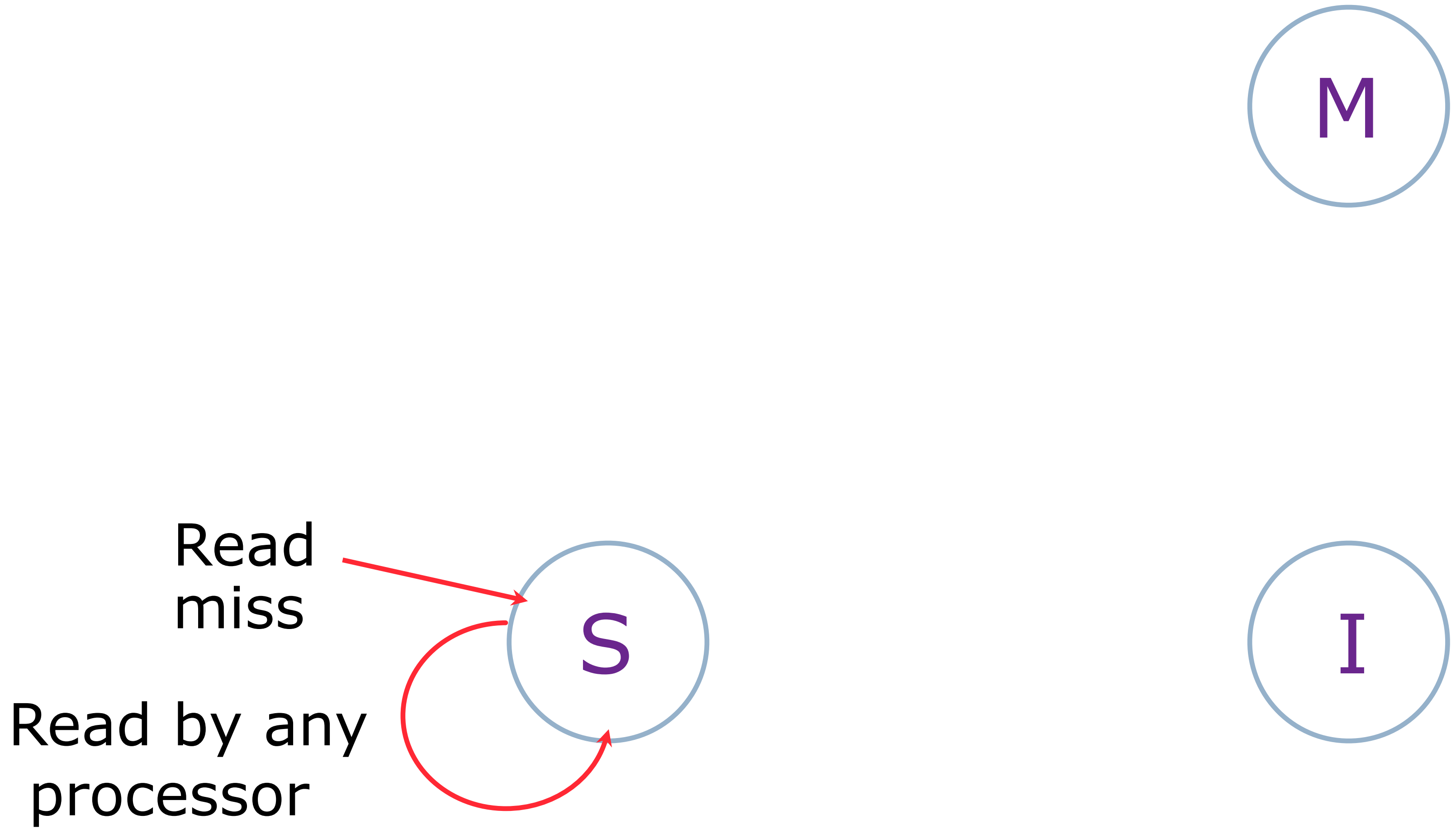
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag



M: Modified
S: Shared
I: Invalid

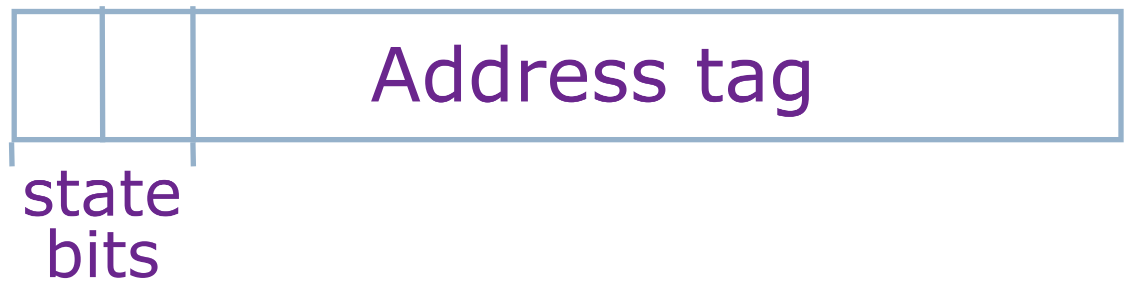


Cache state in processor P_1

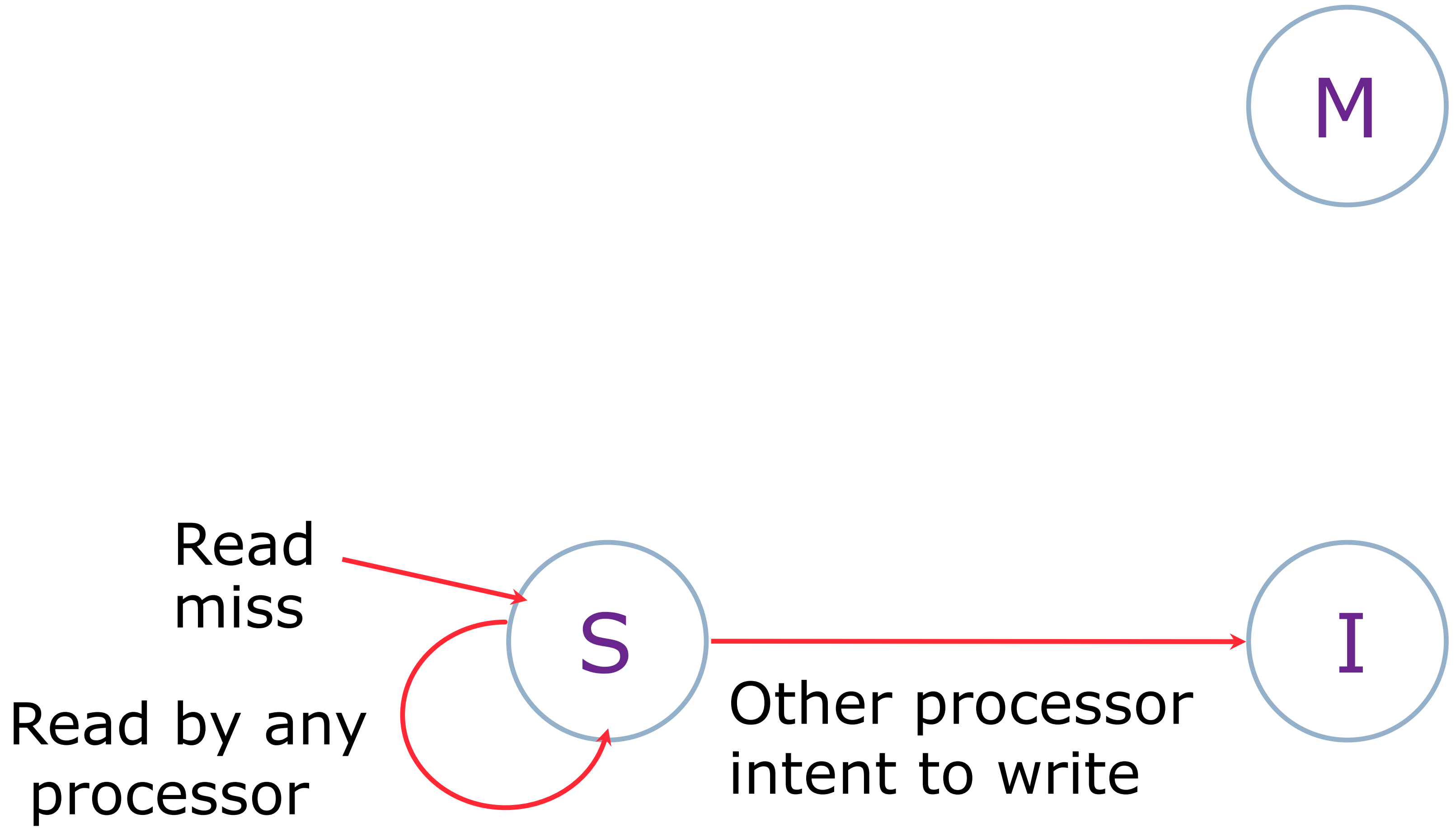
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag



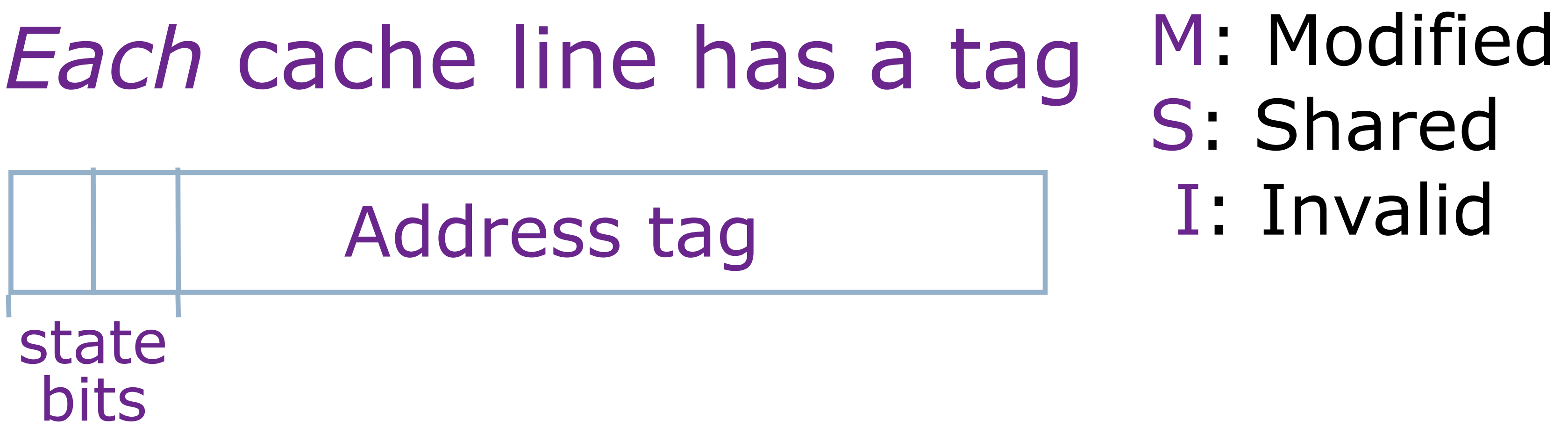
M: Modified
S: Shared
I: Invalid

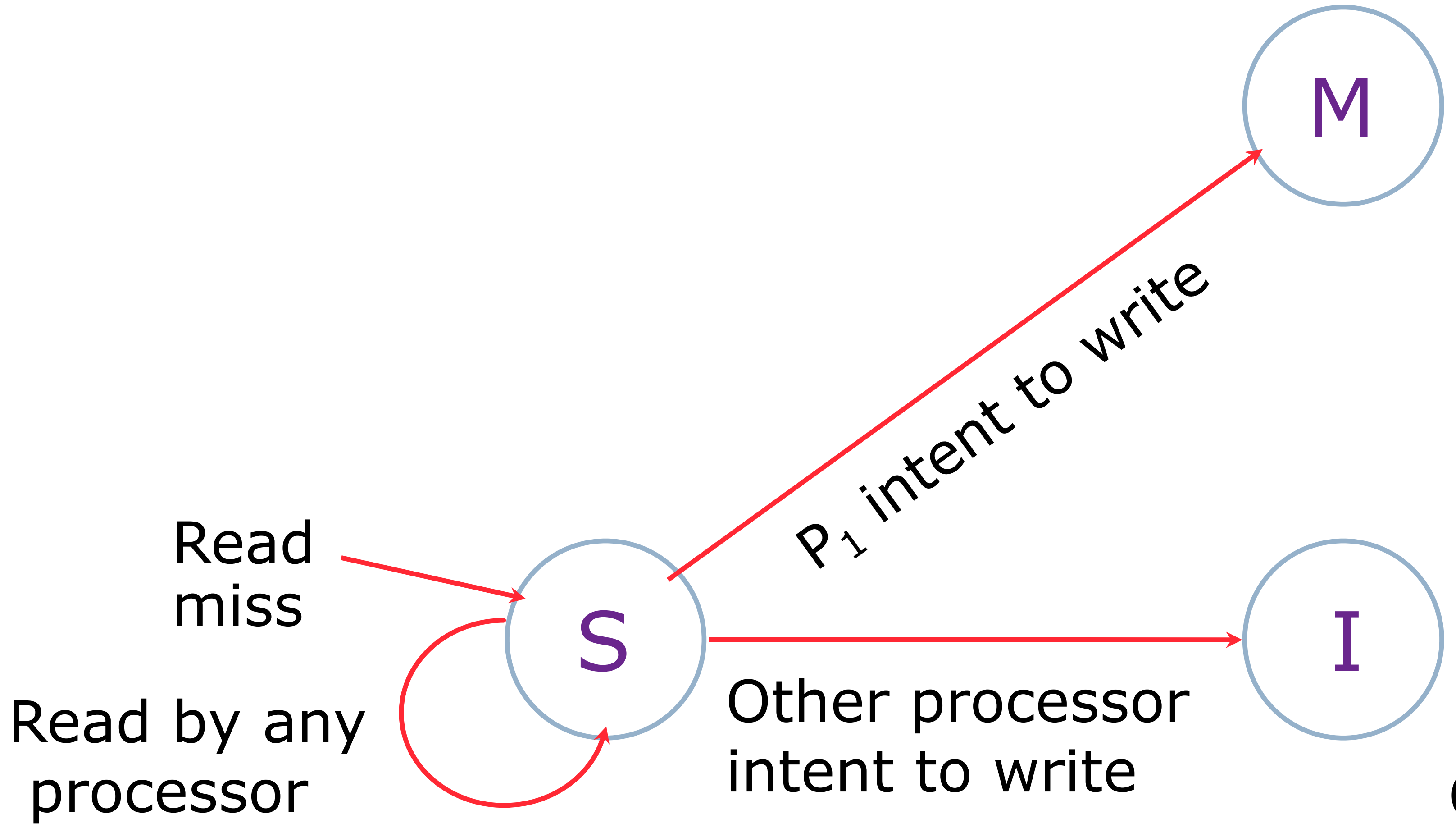


Cache state in processor P_1

Cache State Transition Diagram

The MSI protocol



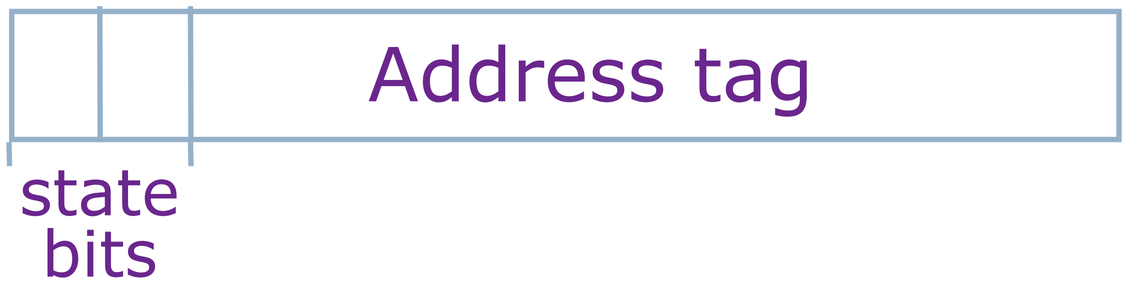


Cache state in processor P_1

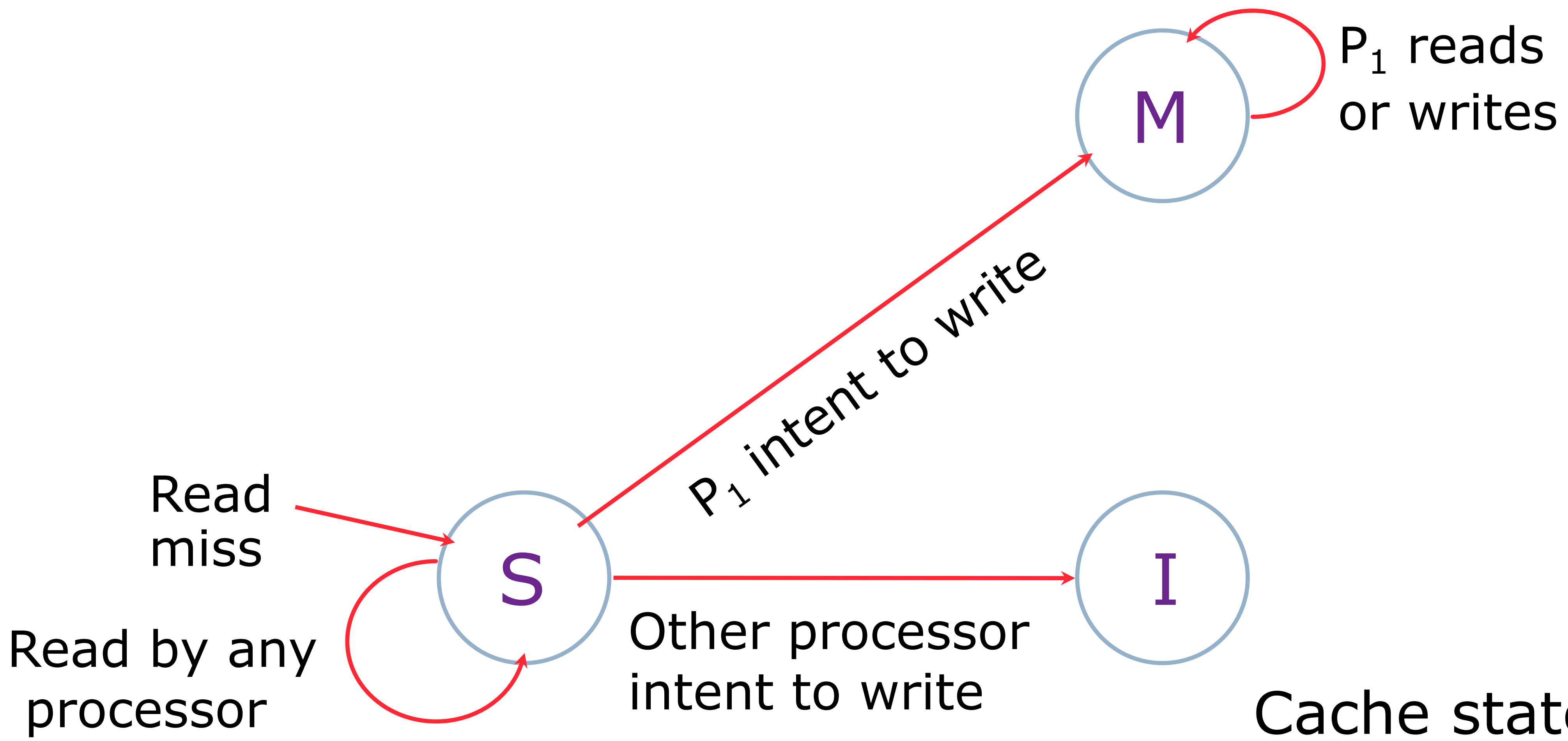
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag



M: Modified
S: Shared
I: Invalid

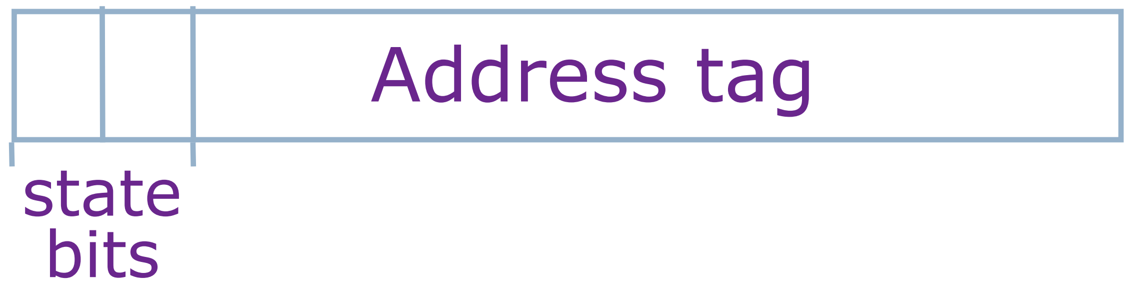


Cache state in processor P_1

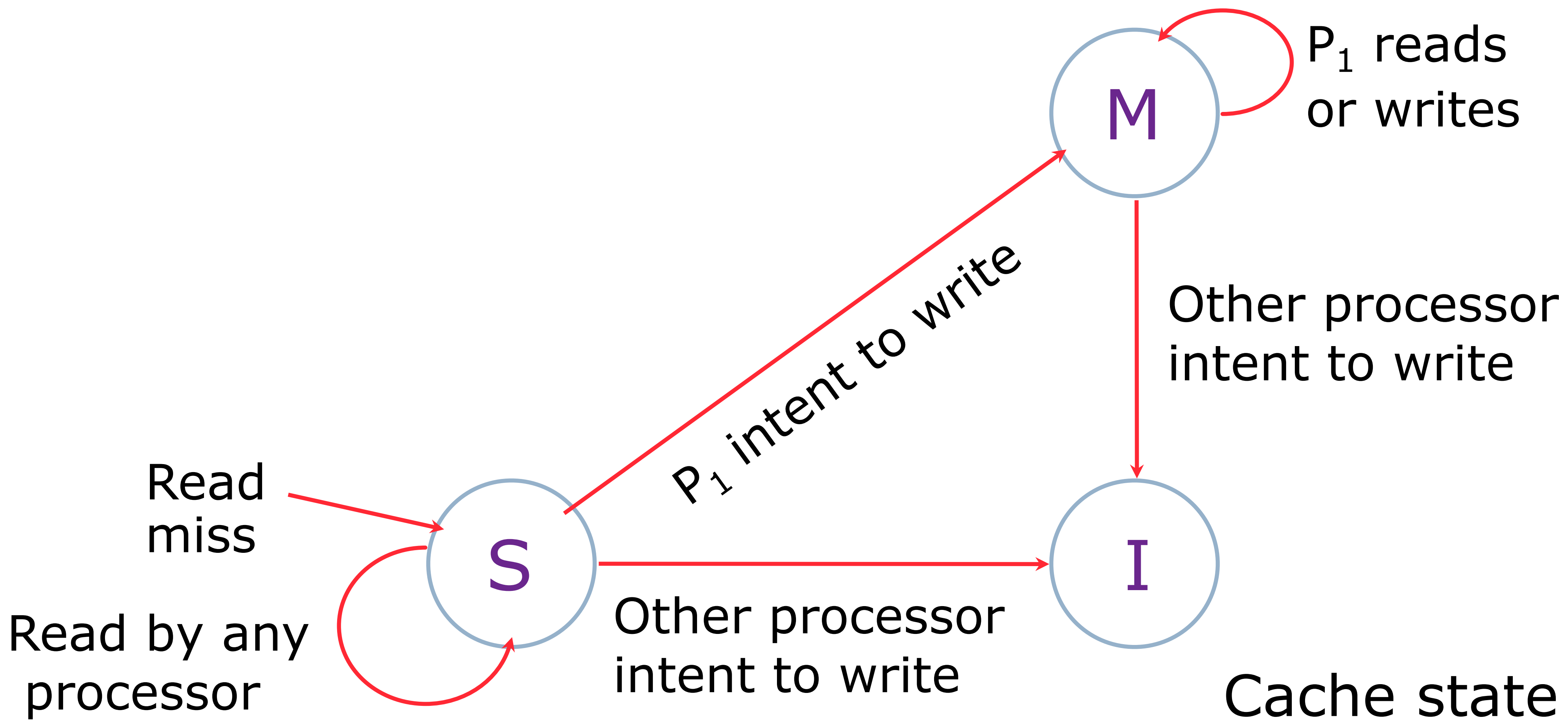
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag



M: Modified
S: Shared
I: Invalid

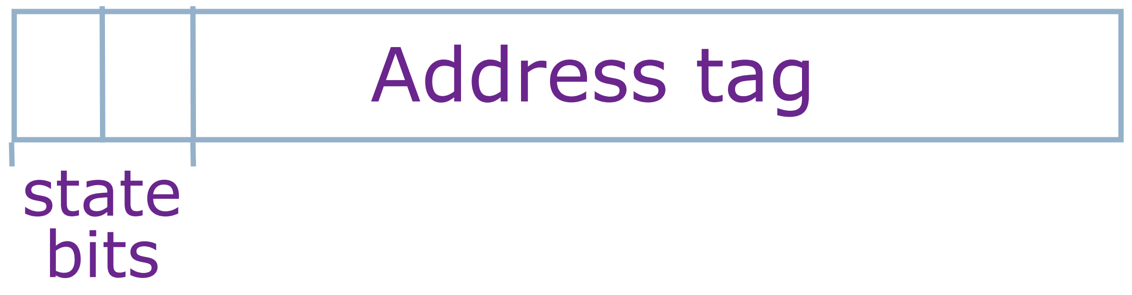


Cache state in processor P_1

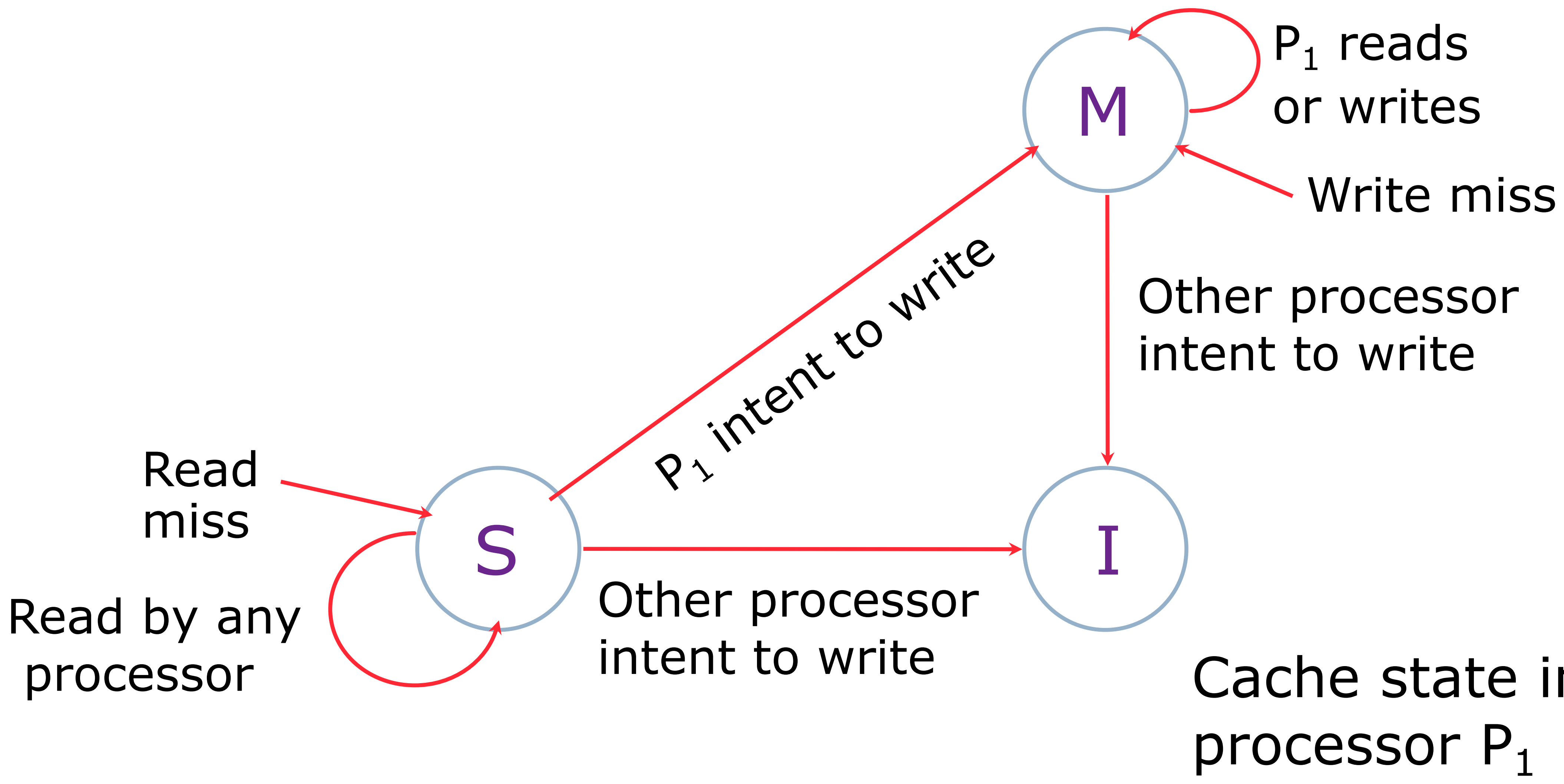
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag



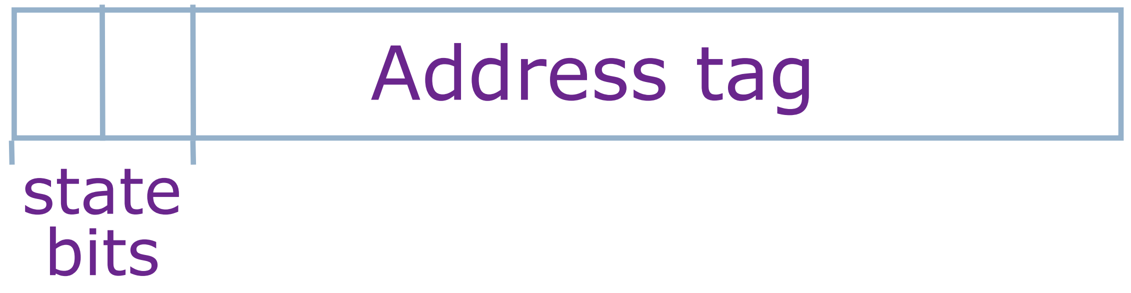
M: Modified
S: Shared
I: Invalid


M
S
I
P1 reads or writes
Write miss
Other processor intent to write
P1 intent to write
Read miss
Read by any processor
Other processor intent to write
Cache state in processor P1

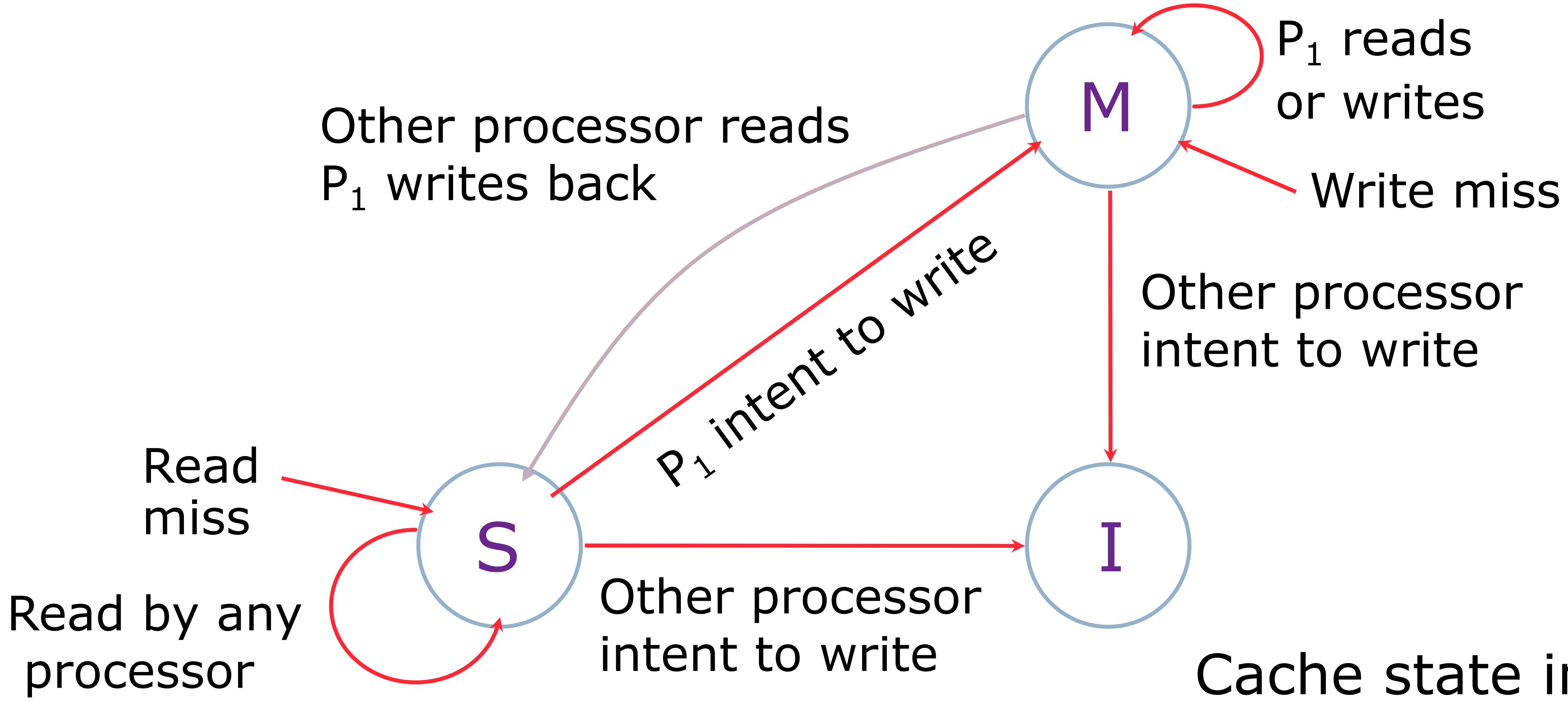
Cache State Transition Diagram

The MSI protocol

Each cache line has a tag



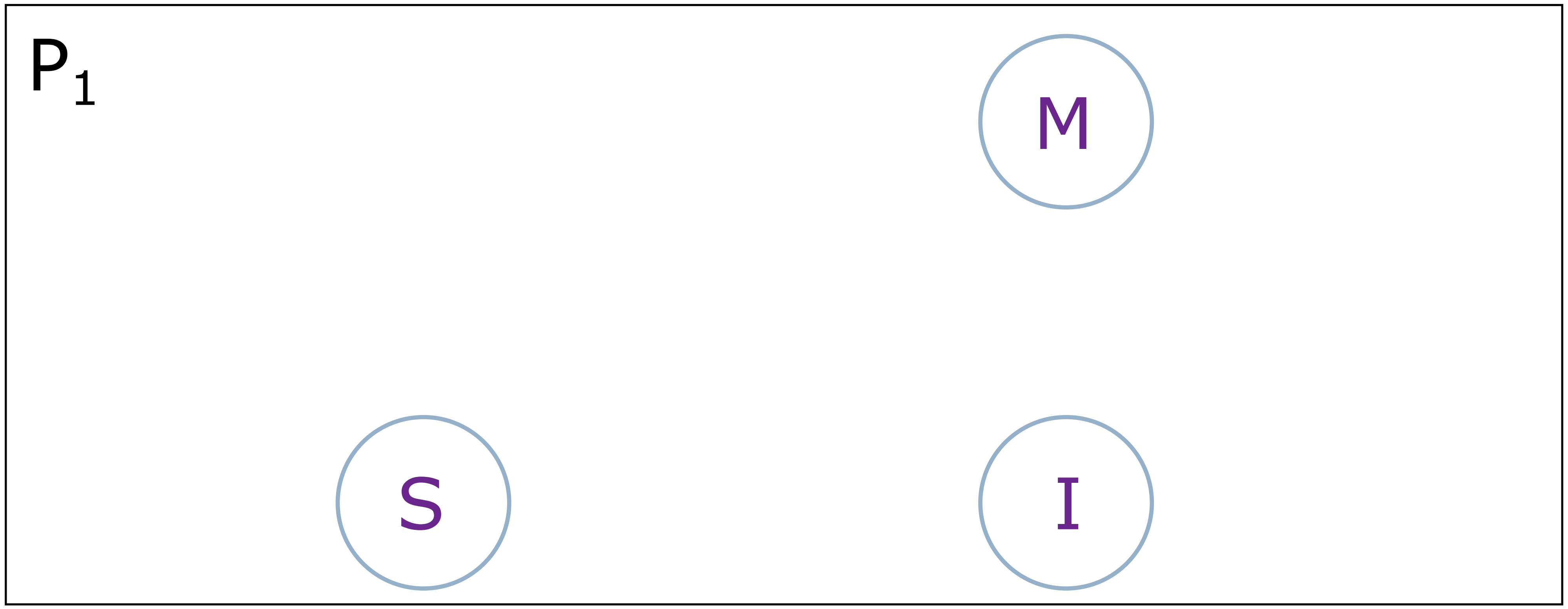
M: Modified
S: Shared
I: Invalid

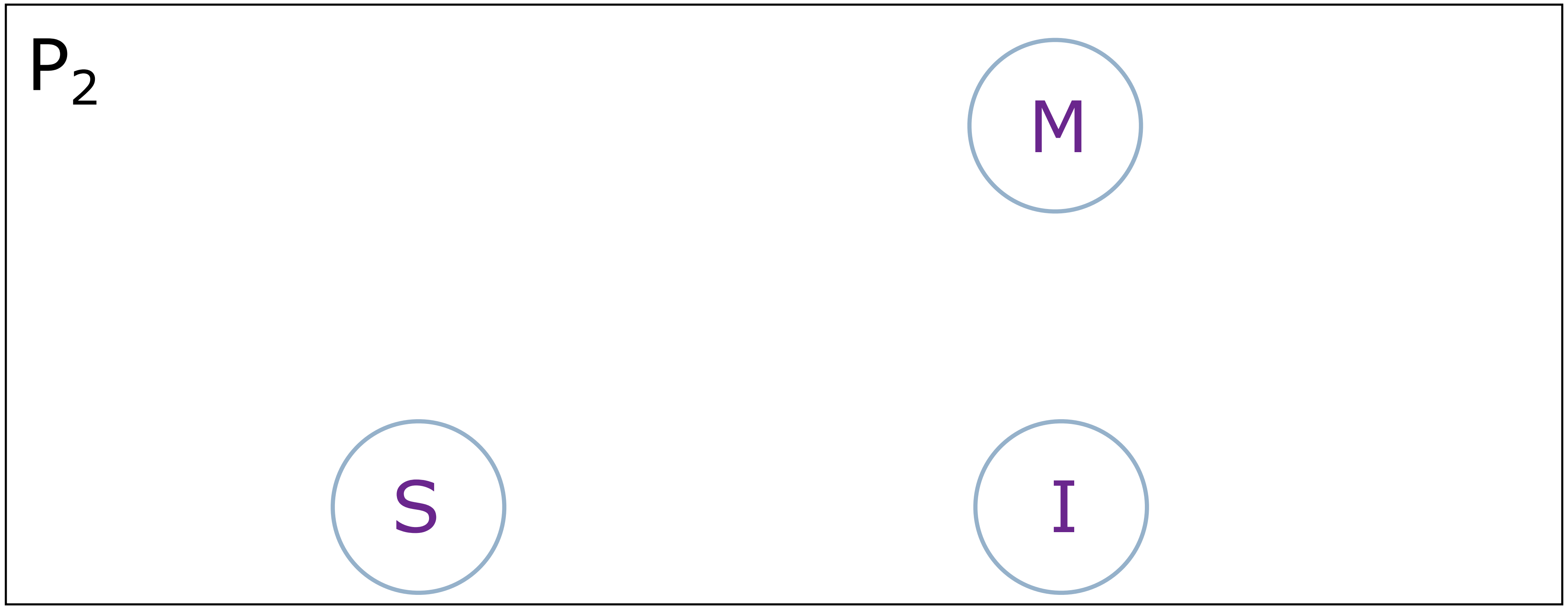


M
S
I
P1 reads or writes
Write miss
Other processor reads P1 writes back
P1 intent to write
Other processor intent to write
Read miss
Read by any processor
Other processor intent to write

Cache state in processor P_1

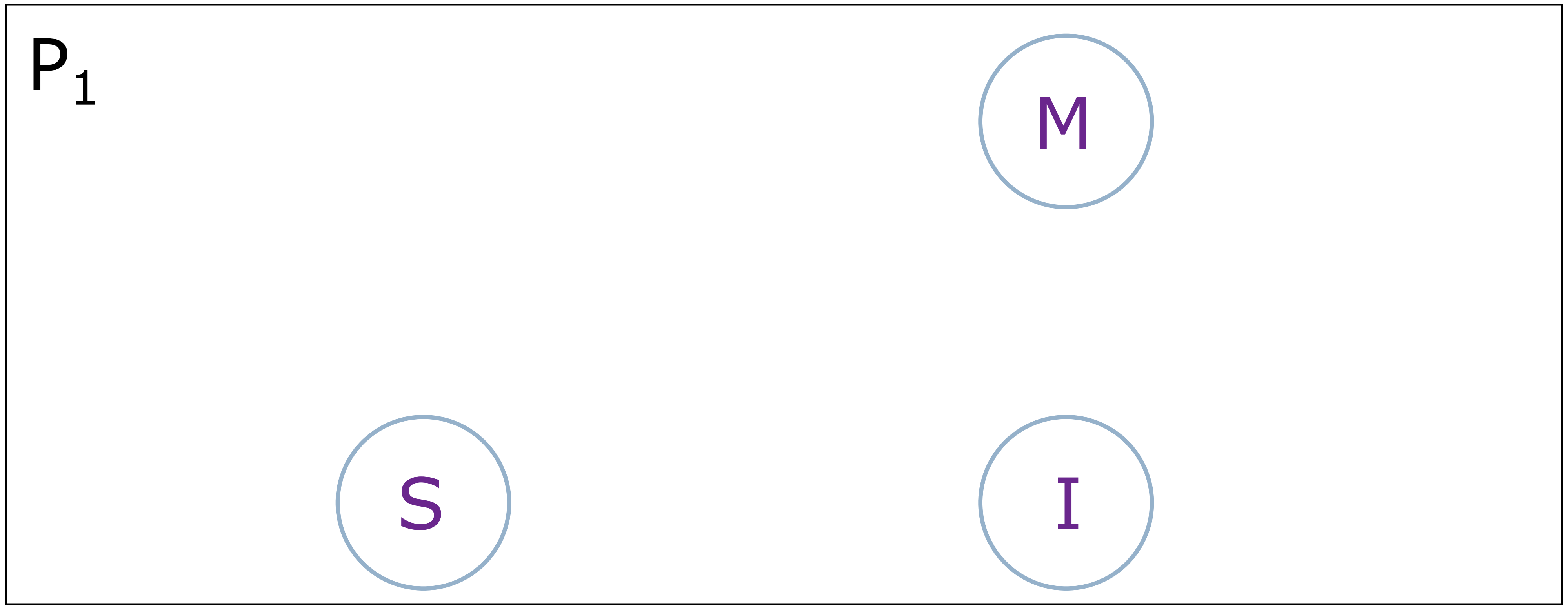
Two Processor Example
(Reading and writing the same cache line)

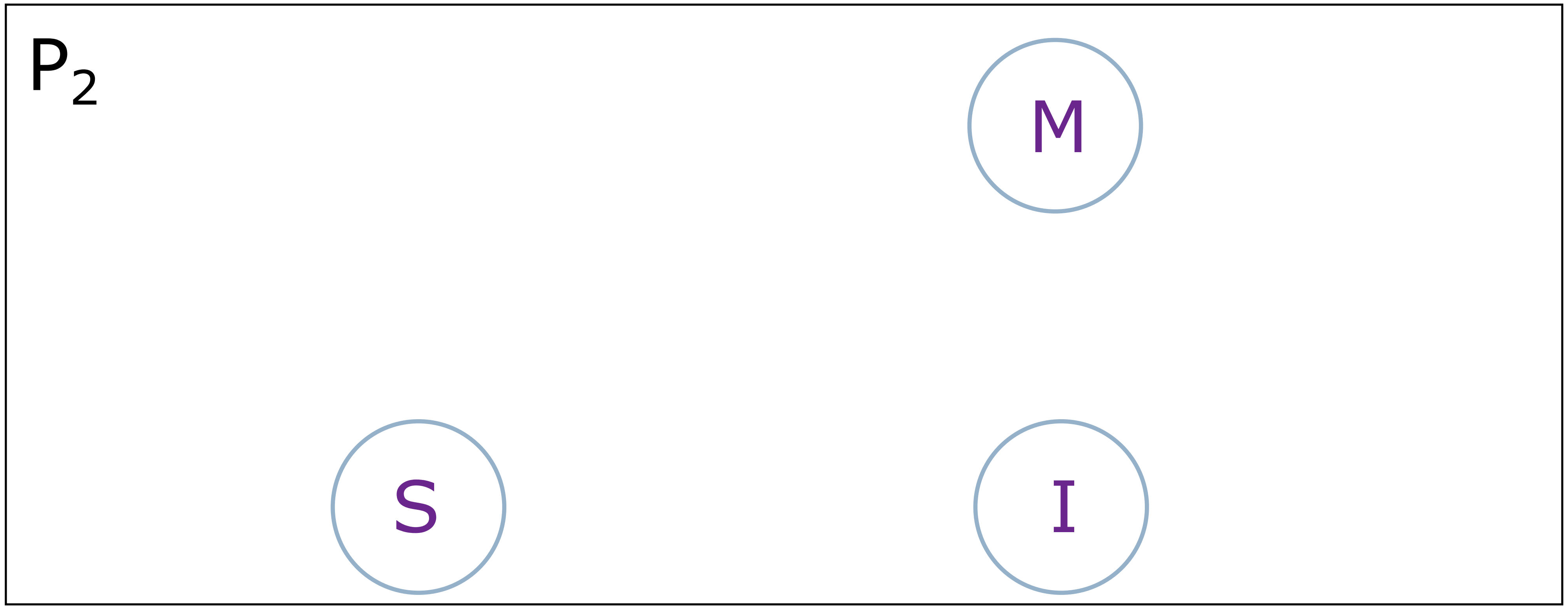




Two Processor Example
(Reading and writing the same cache line)

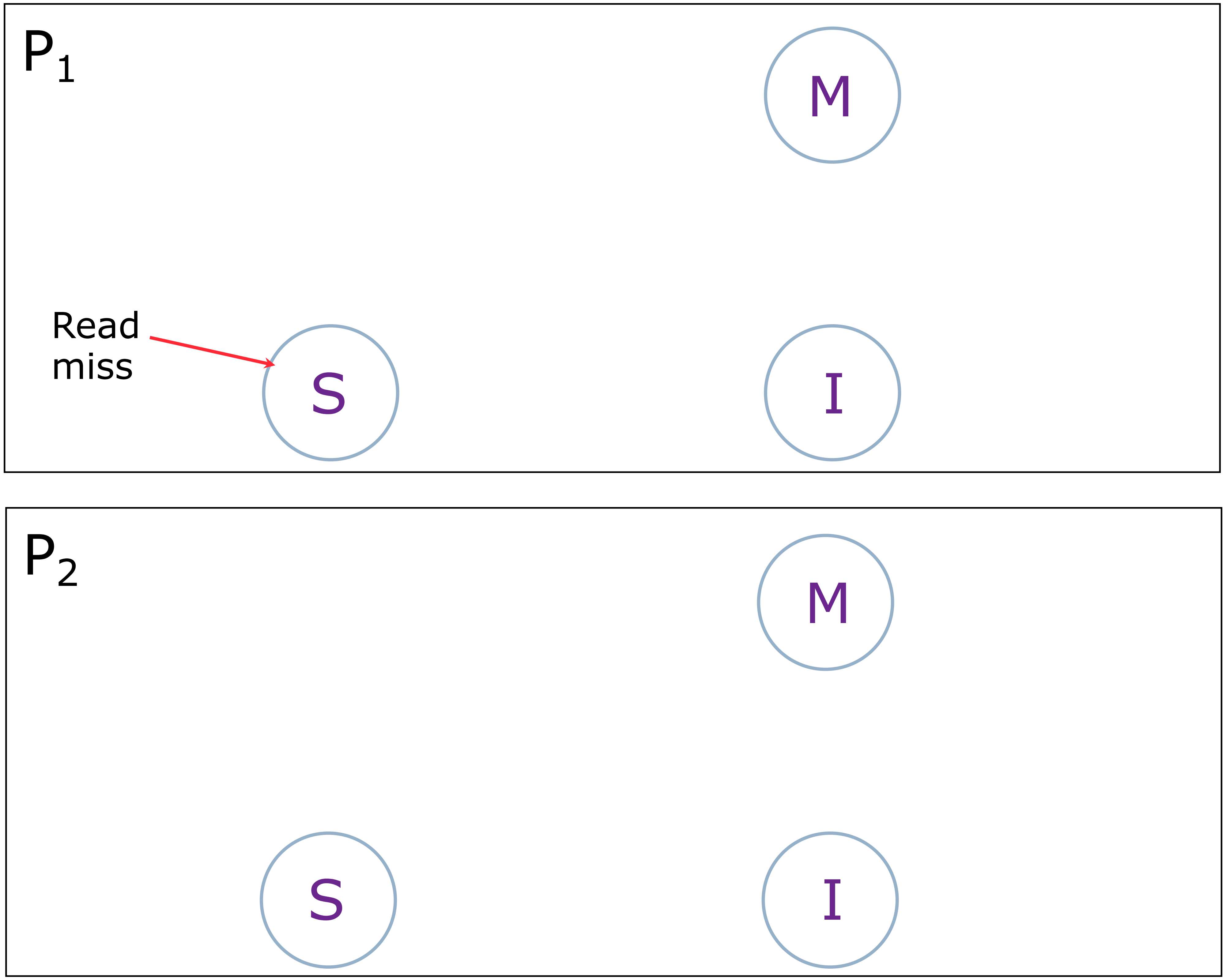
P_1 reads





Two Processor Example
(Reading and writing the same cache line)

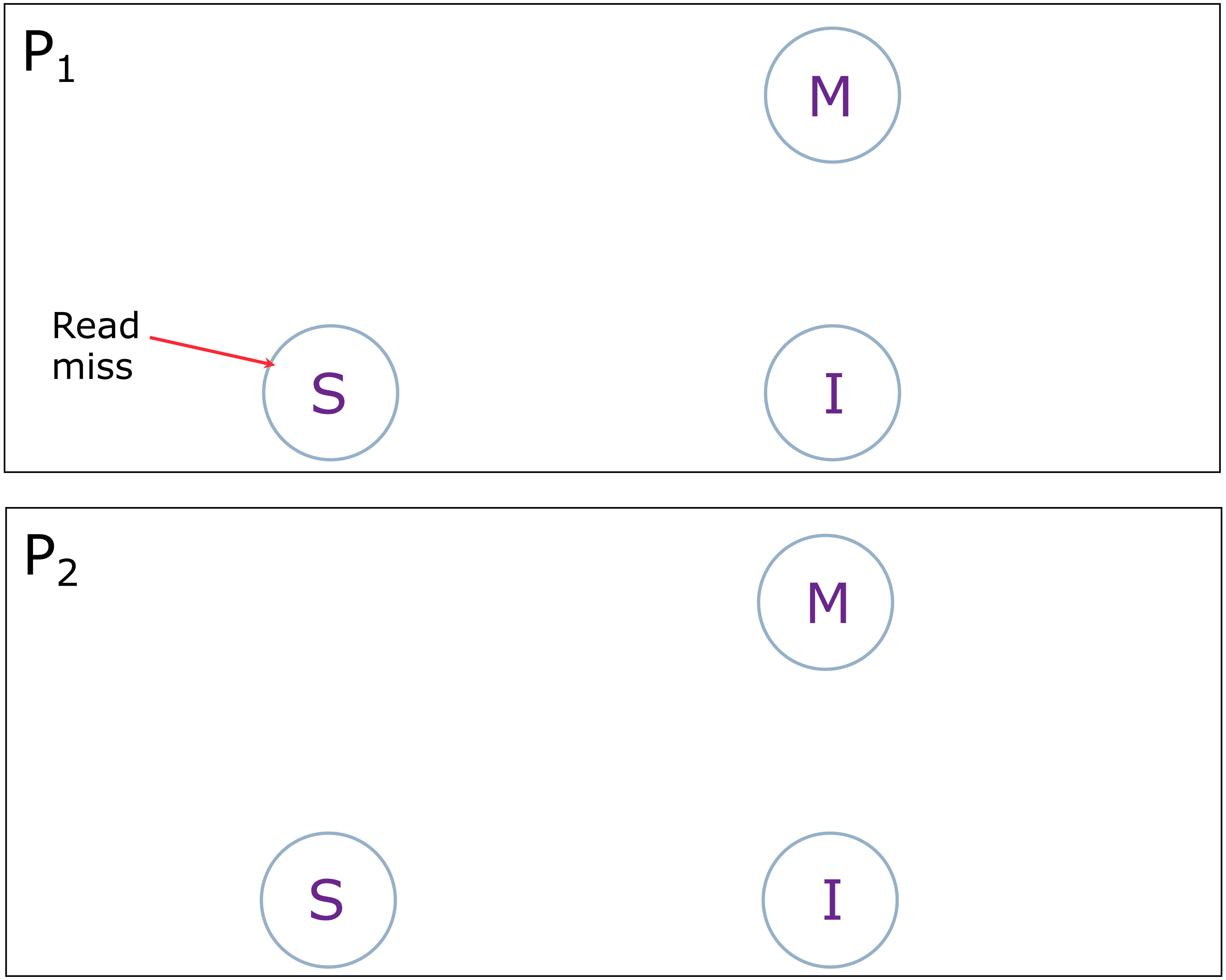
P_1 reads



Two Processor Example
(Reading and writing the same cache line)

P_1 reads

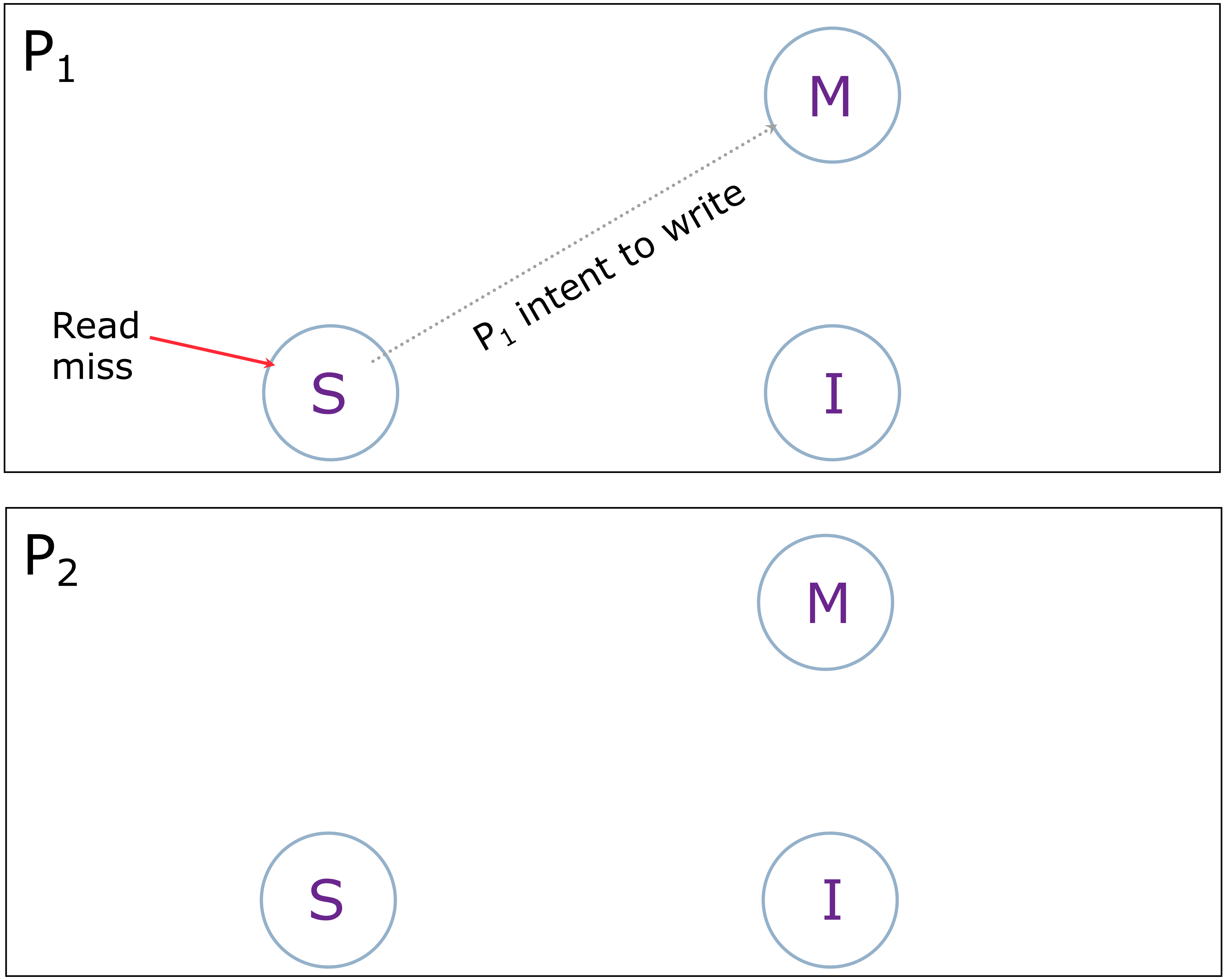
P_1 writes



Two Processor Example
(Reading and writing the same cache line)

P_1 reads

P_1 writes

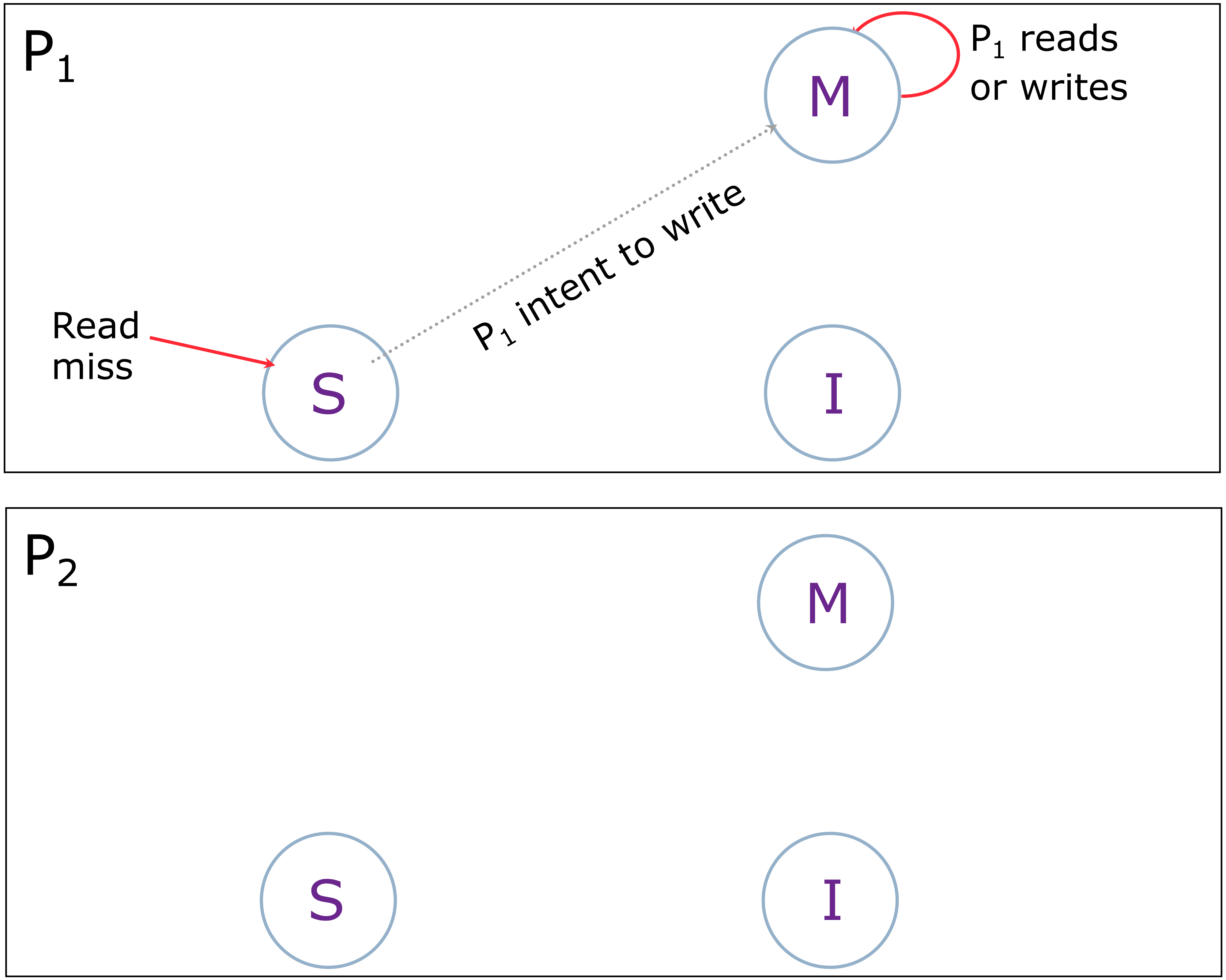


Two Processor Example

(Reading and writing the same cache line)

P_1 reads

P_1 writes

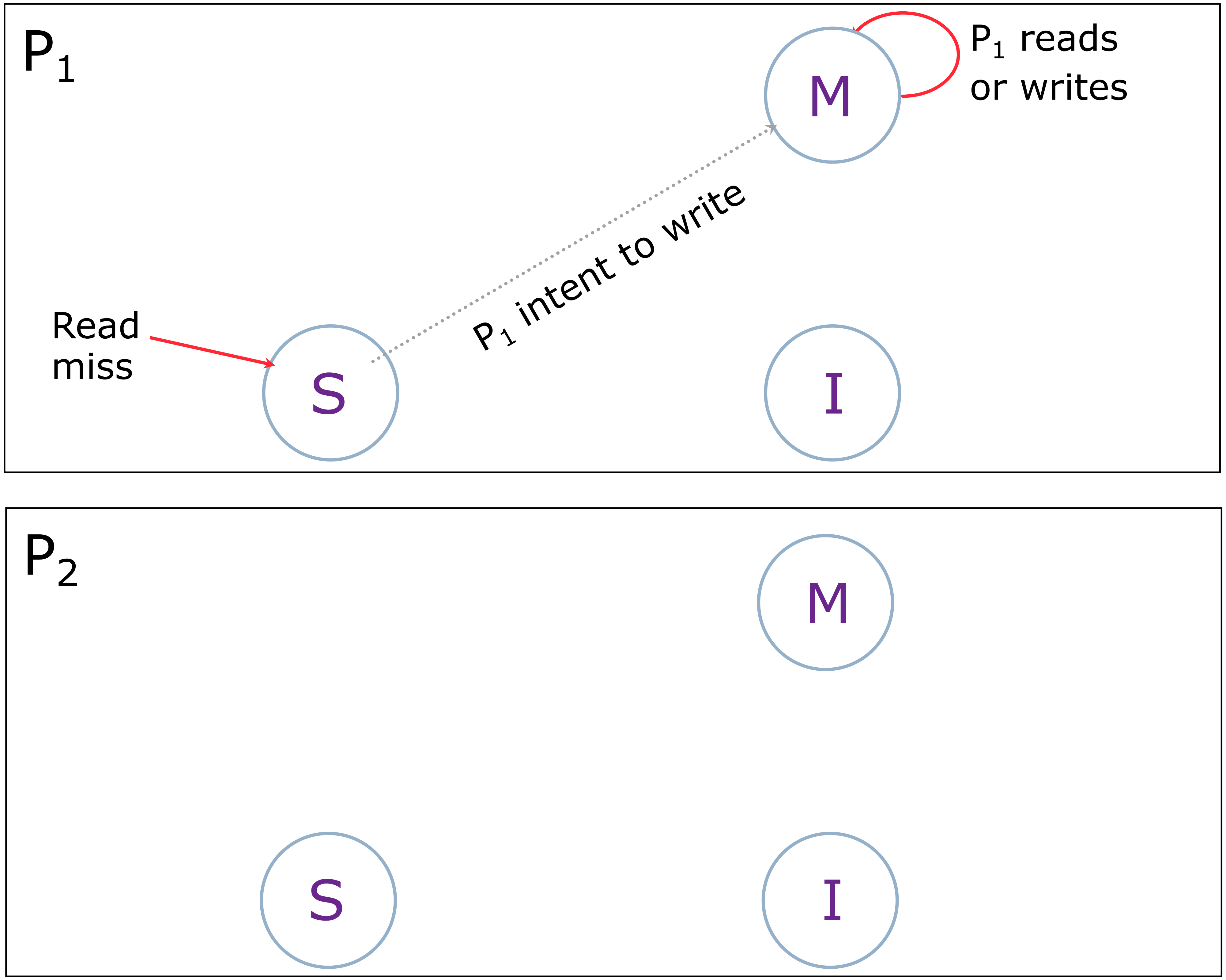


Two Processor Example
(Reading and writing the same cache line)

P_1 reads

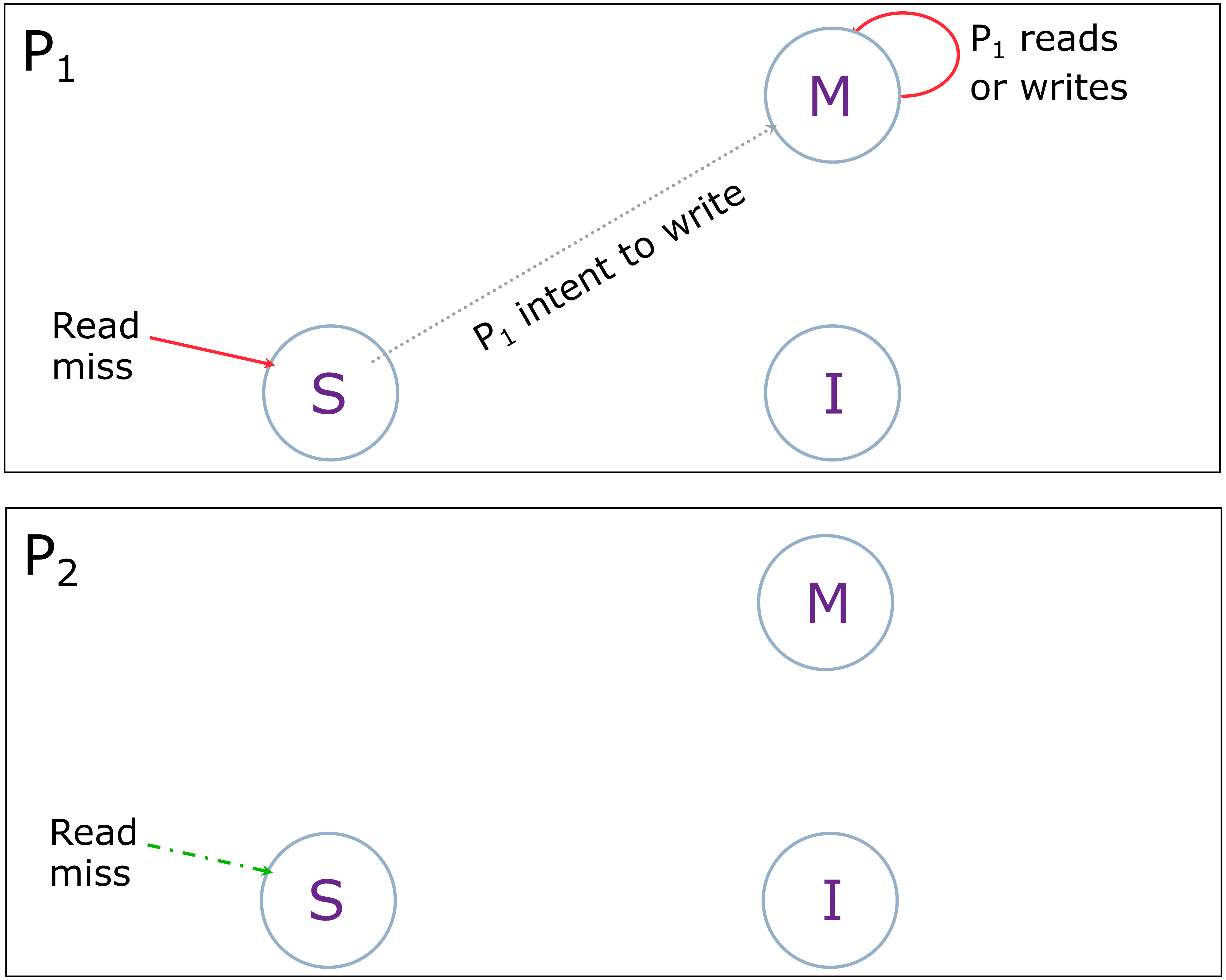
P_1 writes

P_2 reads



Two Processor Example
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads

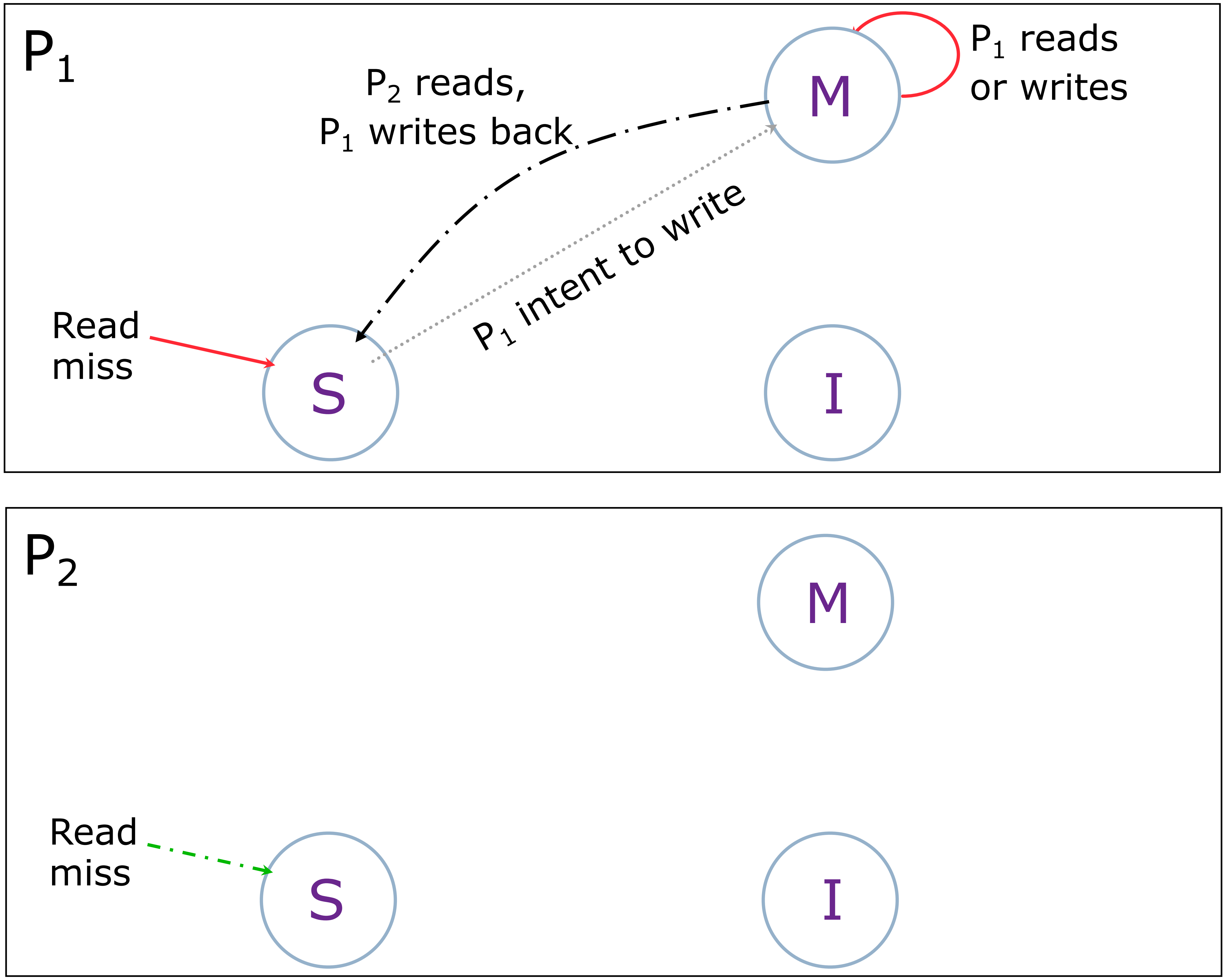


Two Processor Example
(Reading and writing the same cache line)

P_1 reads

P_1 writes

P_2 reads



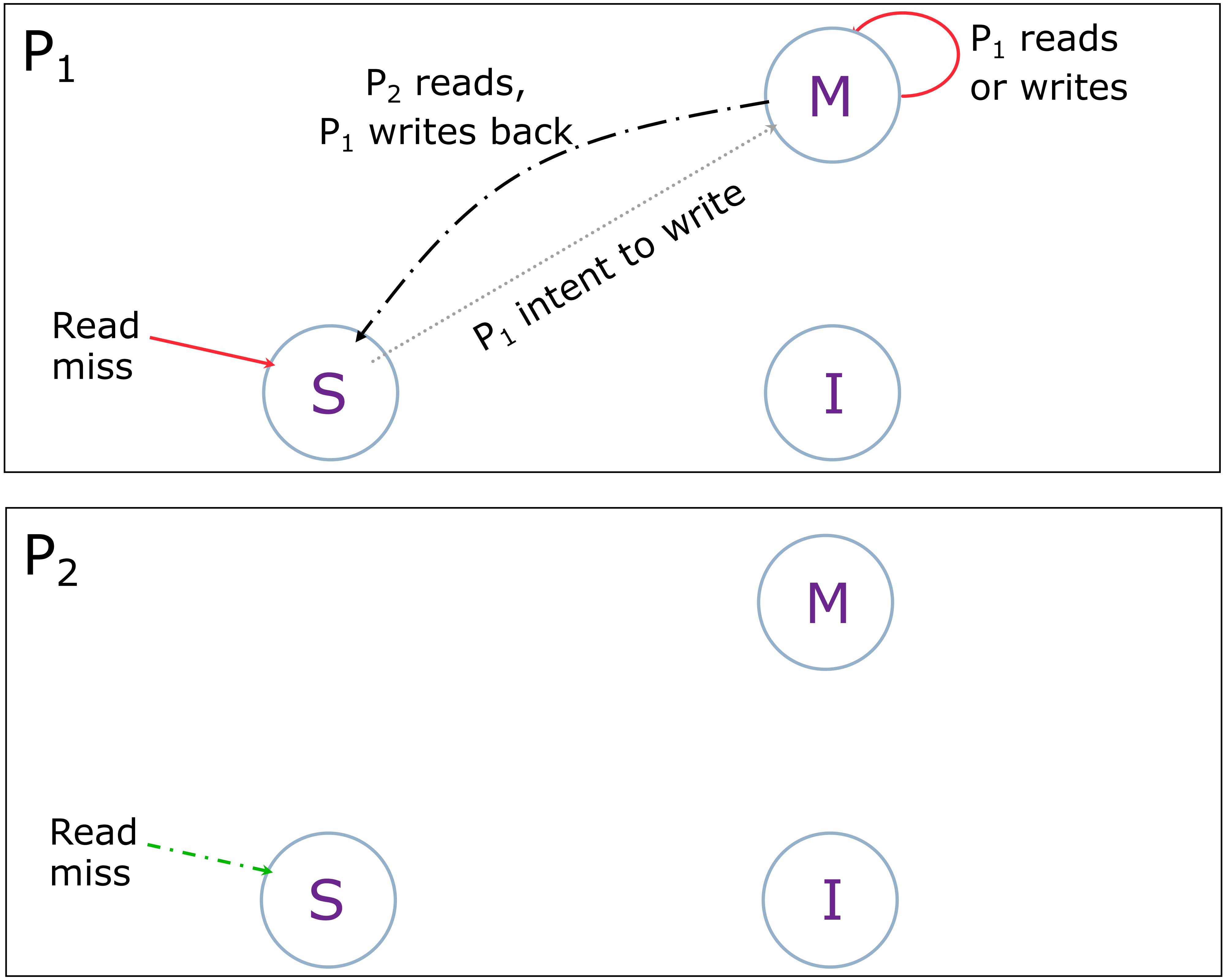
Two Processor Example
(Reading and writing the same cache line)

P_1 reads

P_1 writes

P_2 reads

P_2 writes



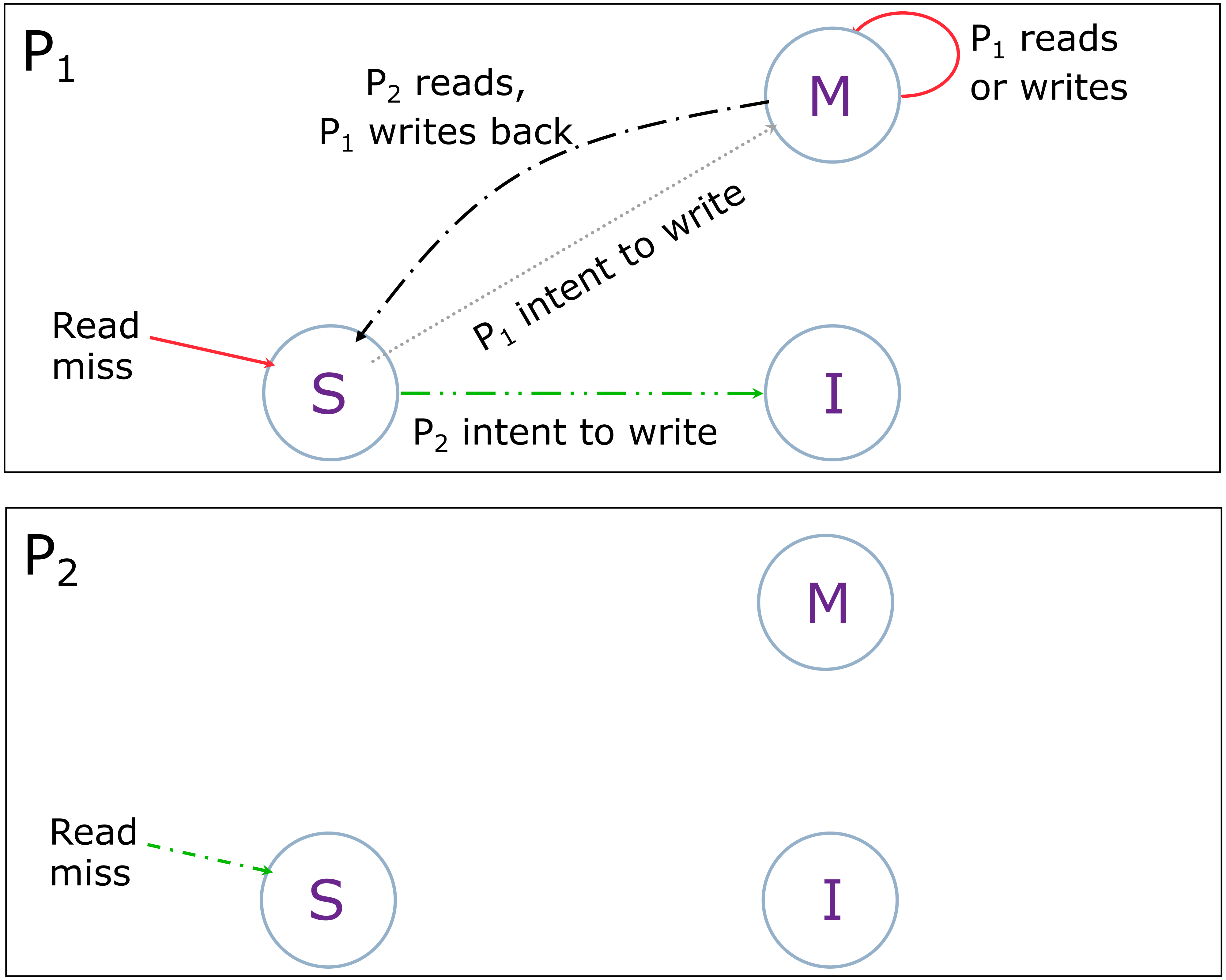
Two Processor Example
(Reading and writing the same cache line)

P_1 reads

P_1 writes

P_2 reads

P_2 writes



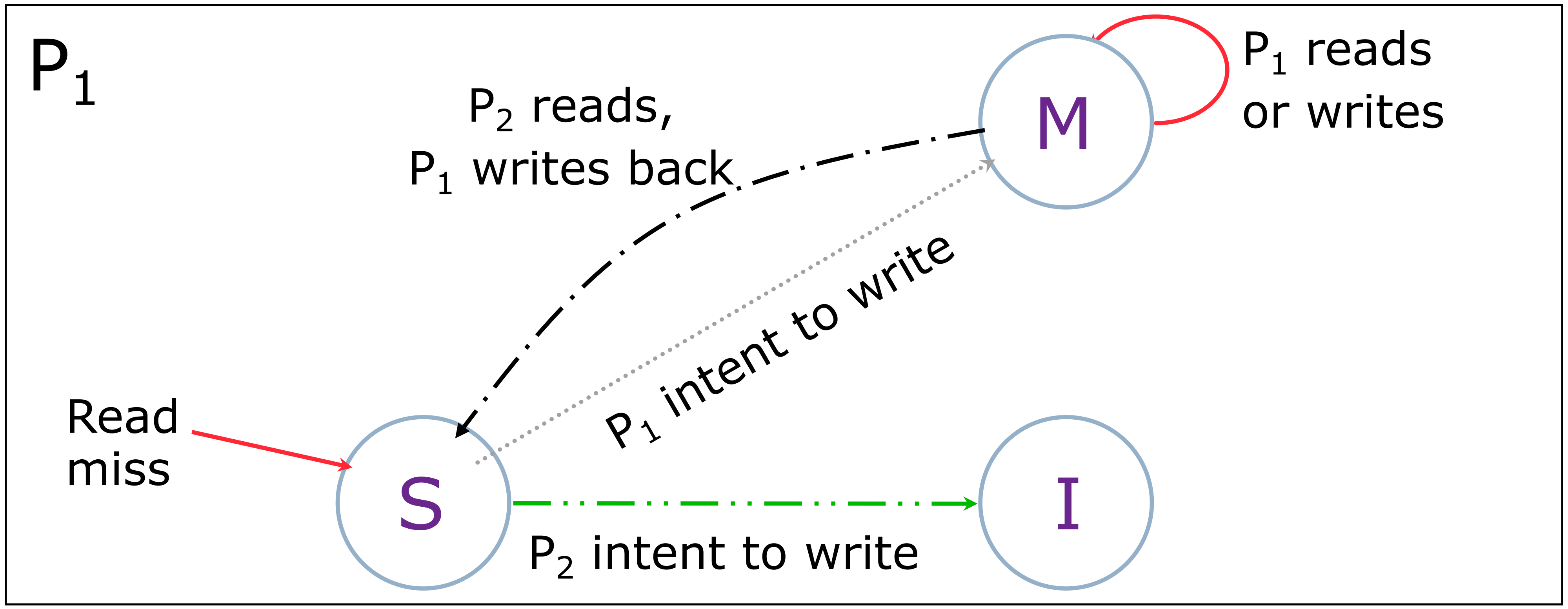
Two Processor Example
(Reading and writing the same cache line)

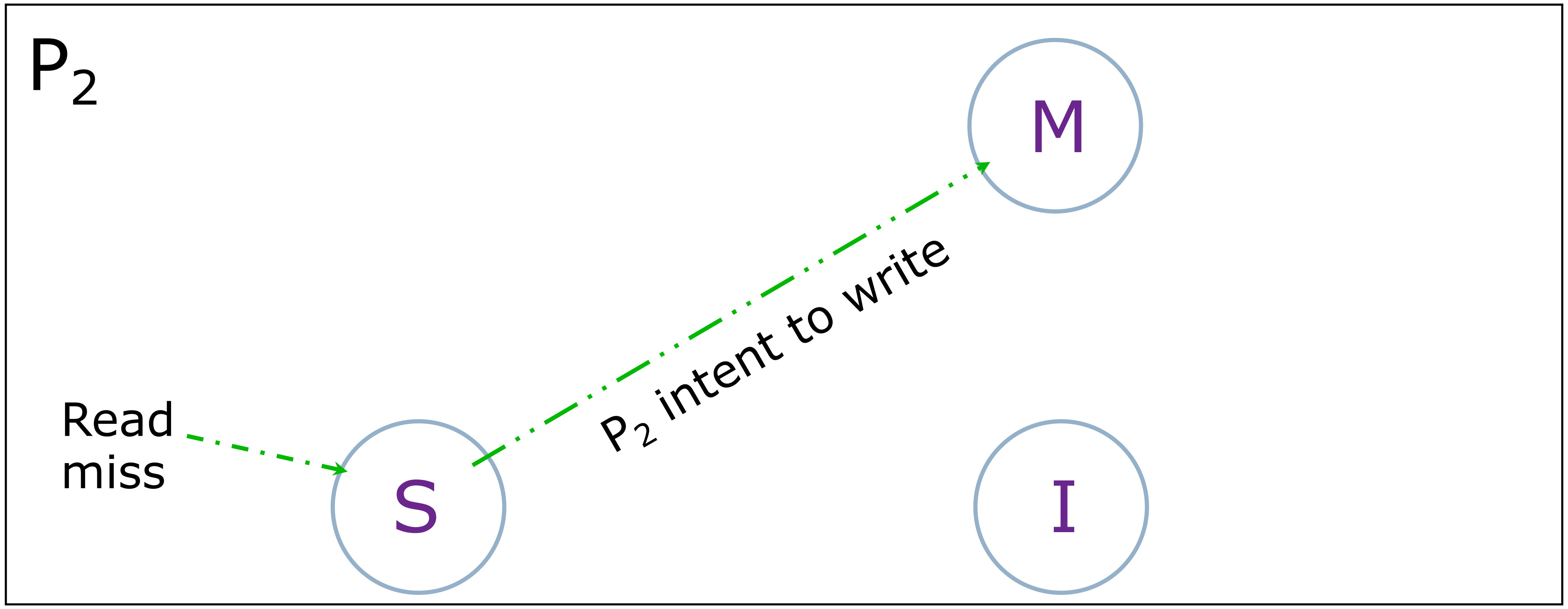
P_1 reads

P_1 writes

P_2 reads

P_2 writes





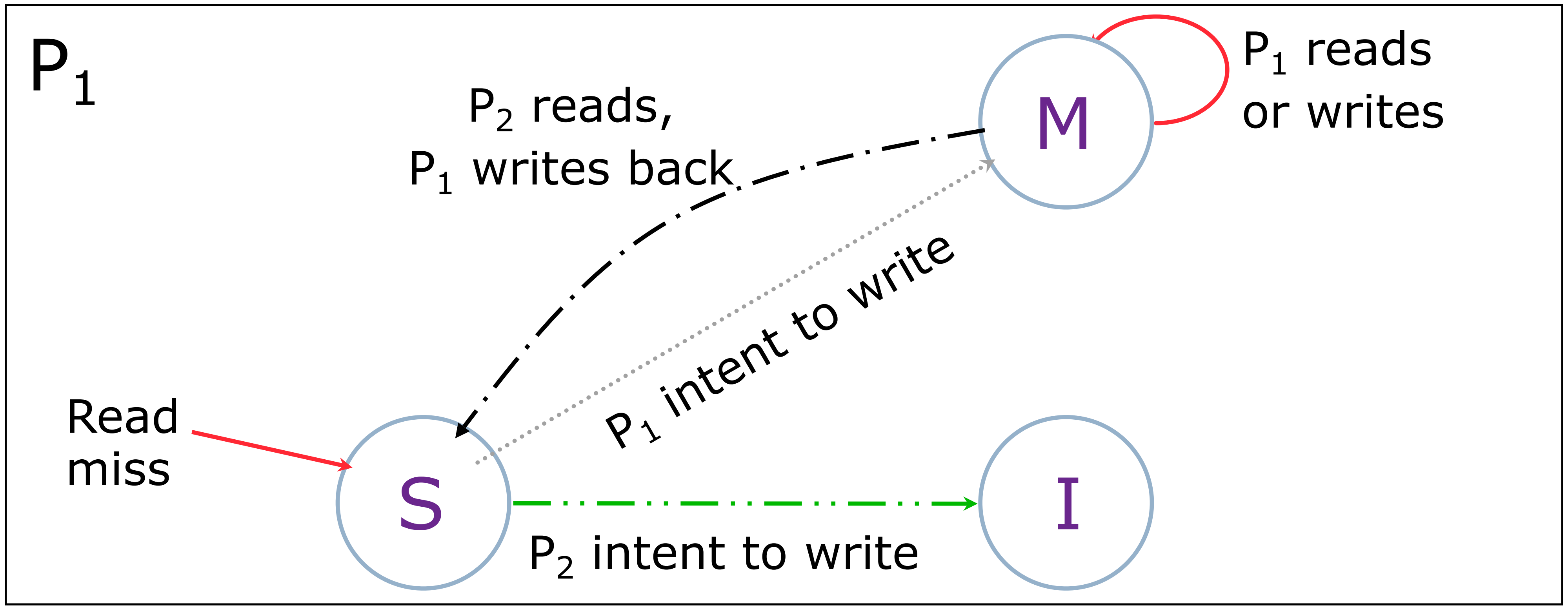
Two Processor Example
(Reading and writing the same cache line)

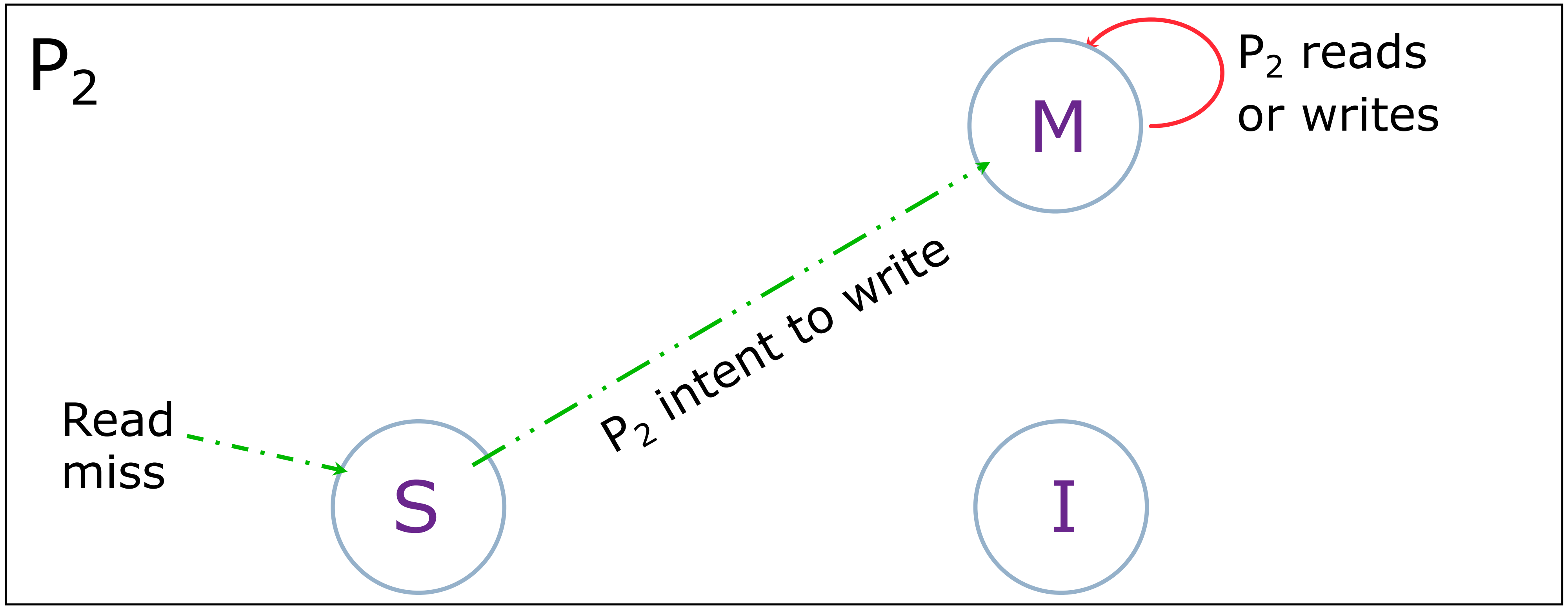
P_1 reads

P_1 writes

P_2 reads

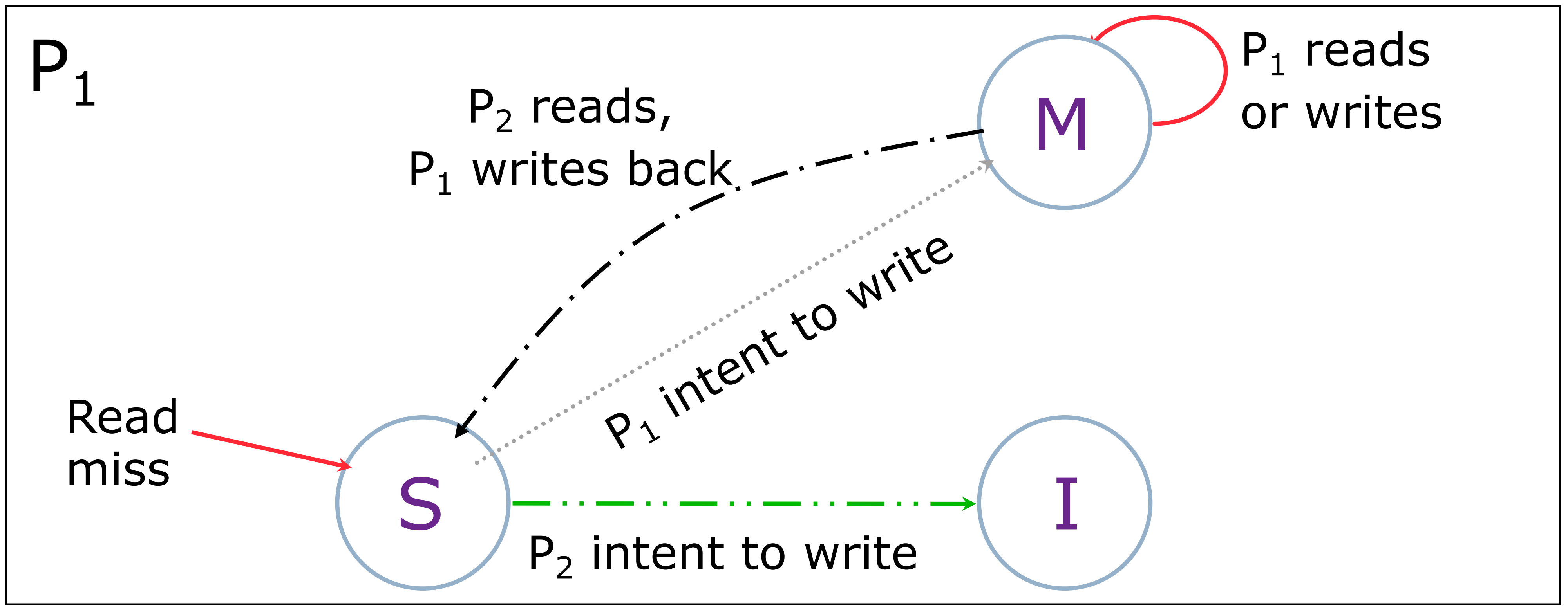
P_2 writes

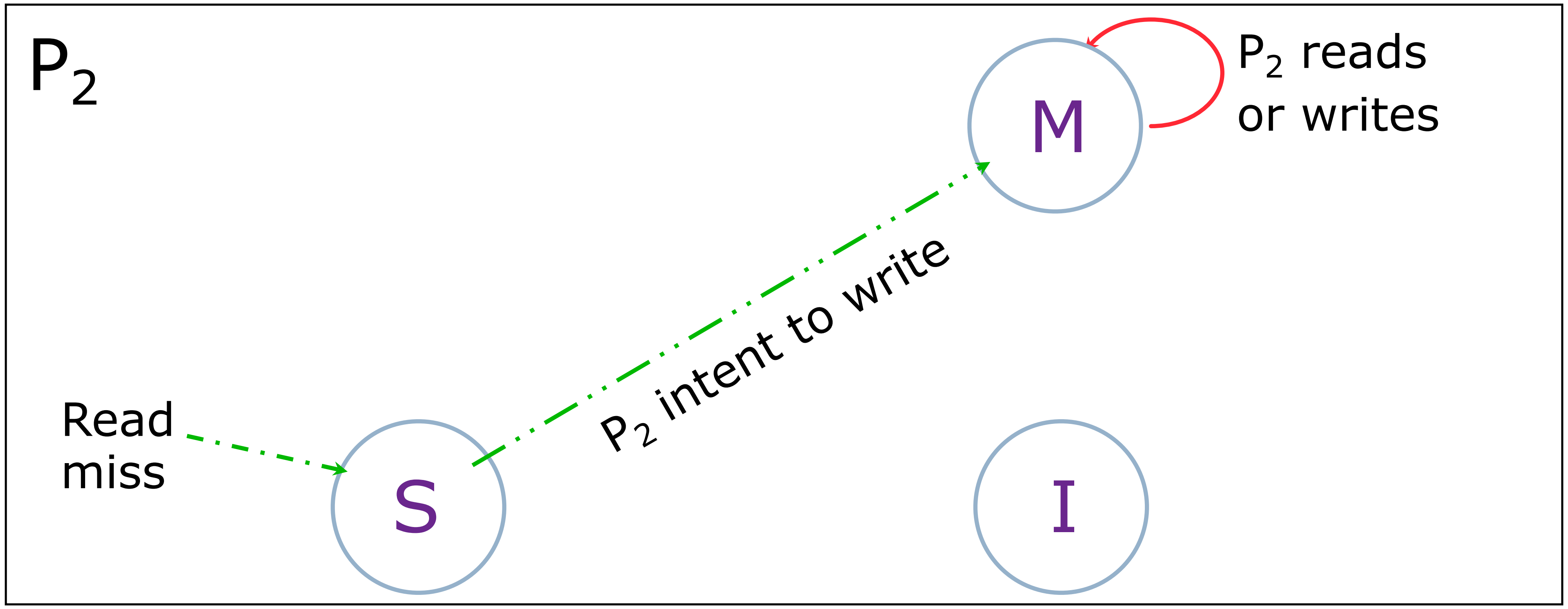




Two Processor Example
(Reading and writing the same cache line)

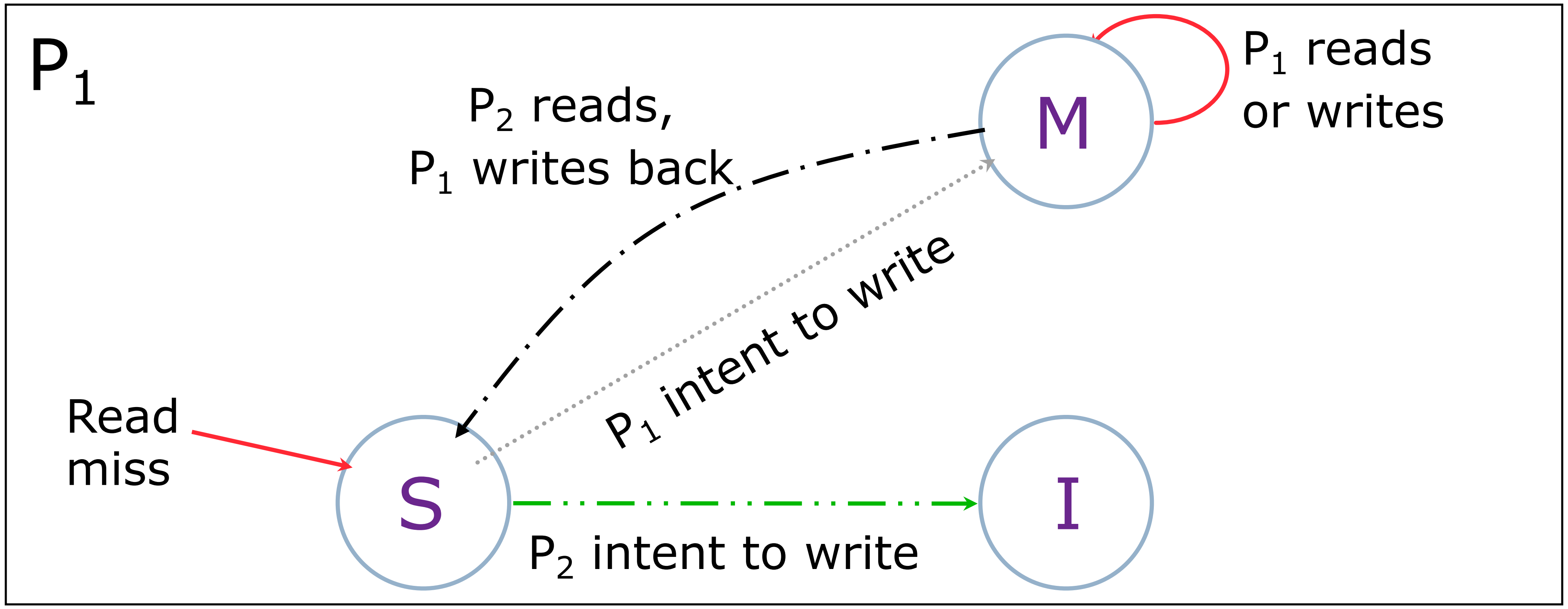
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads

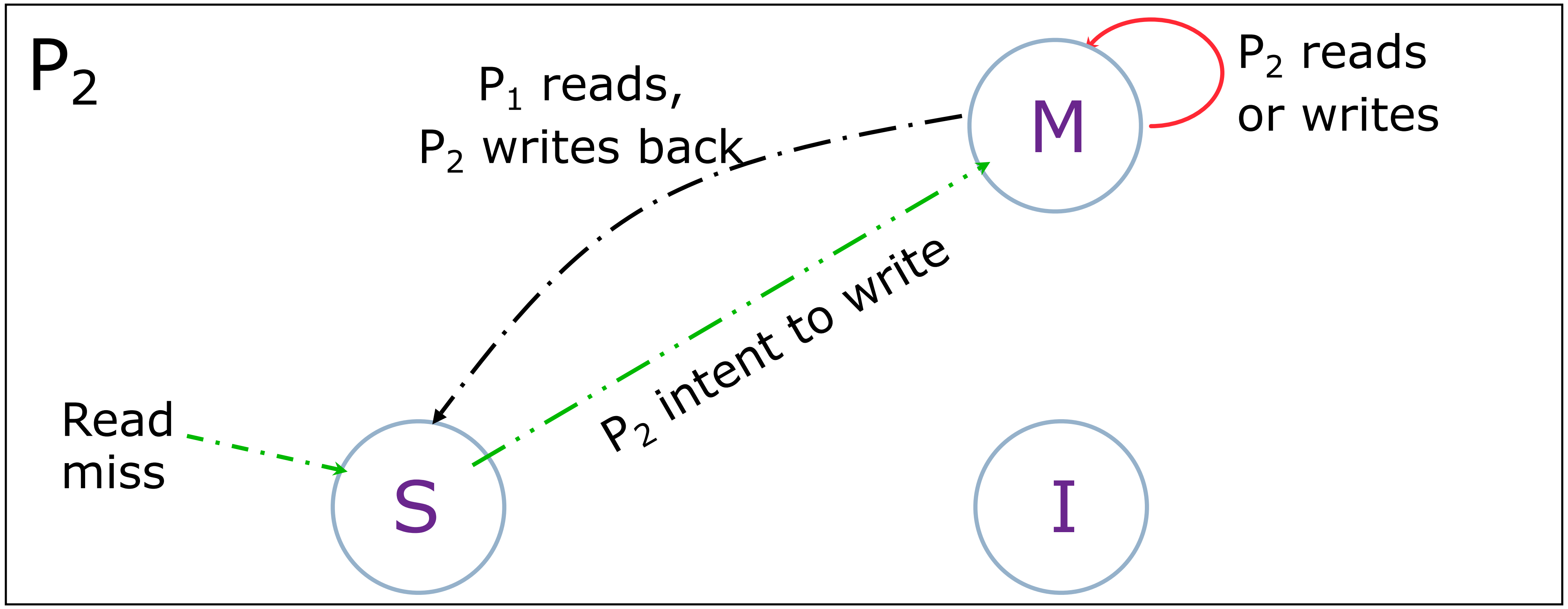




Two Processor Example
(Reading and writing the same cache line)

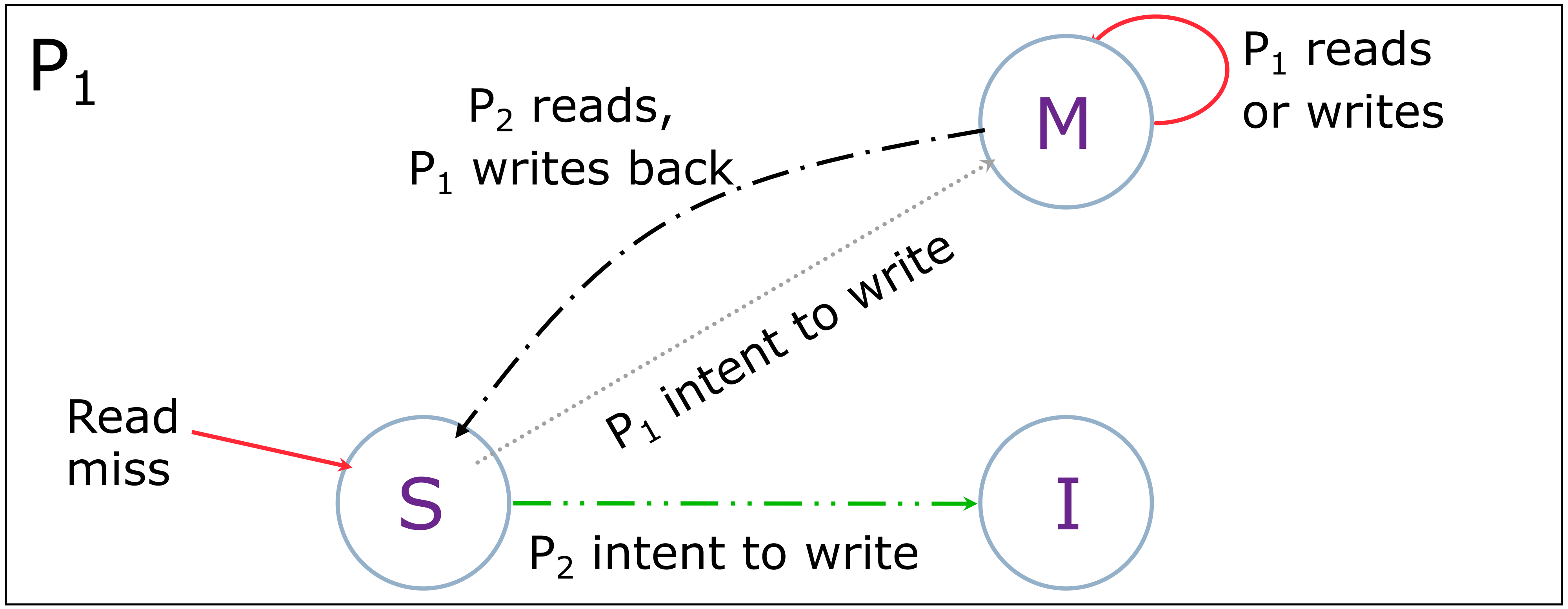
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads

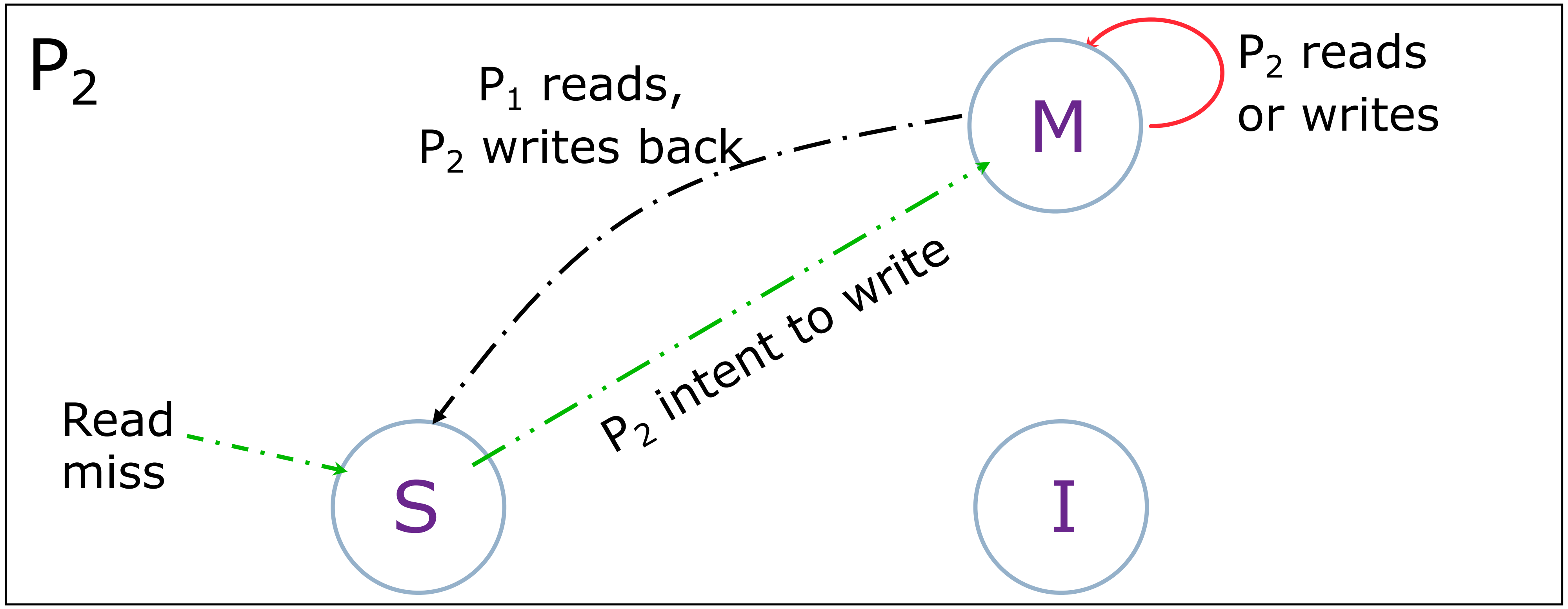




Two Processor Example
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes





Two Processor Example
(Reading and writing the same cache line)

P_1 reads

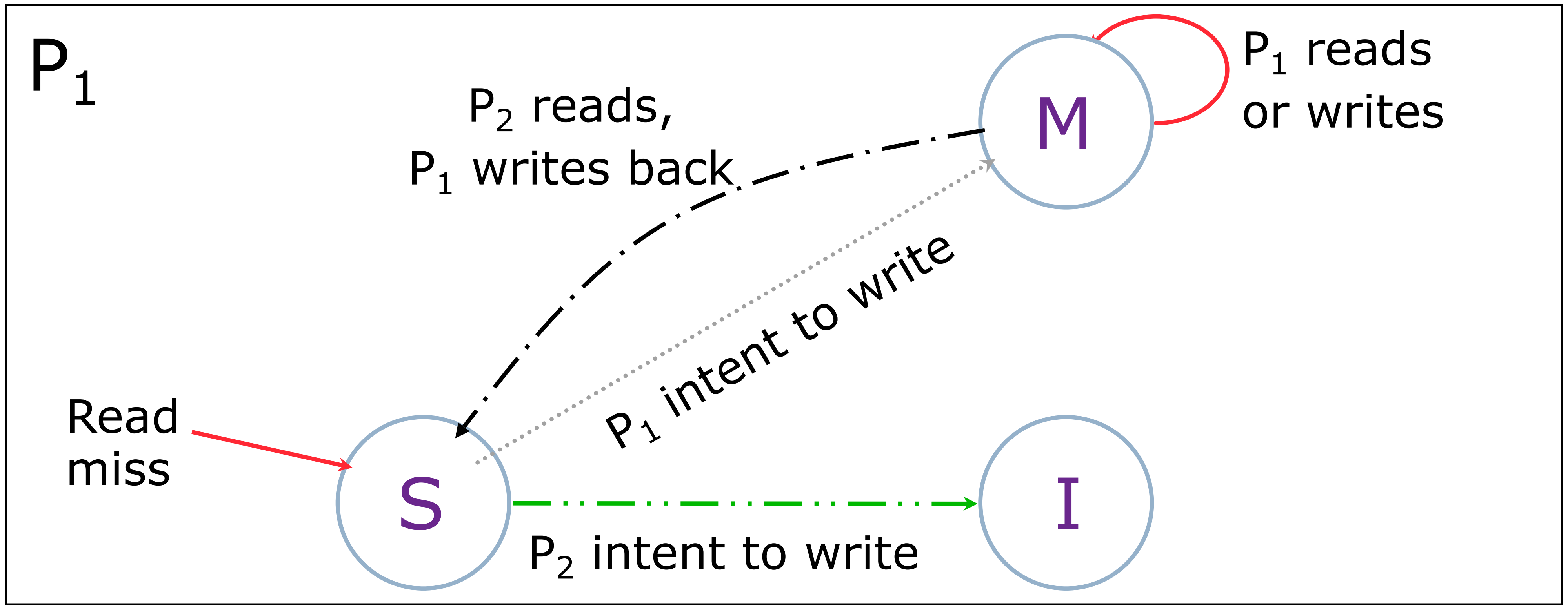
P_1 writes

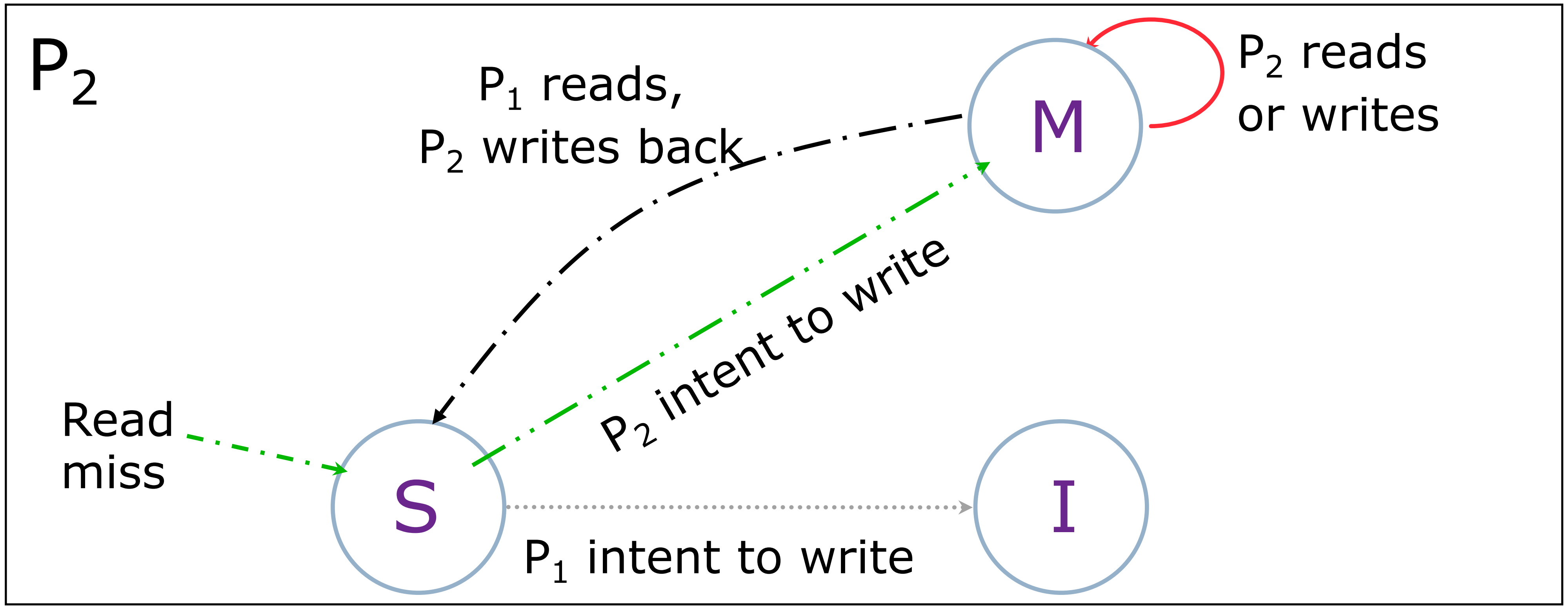
P_2 reads

P_2 writes

P_1 reads

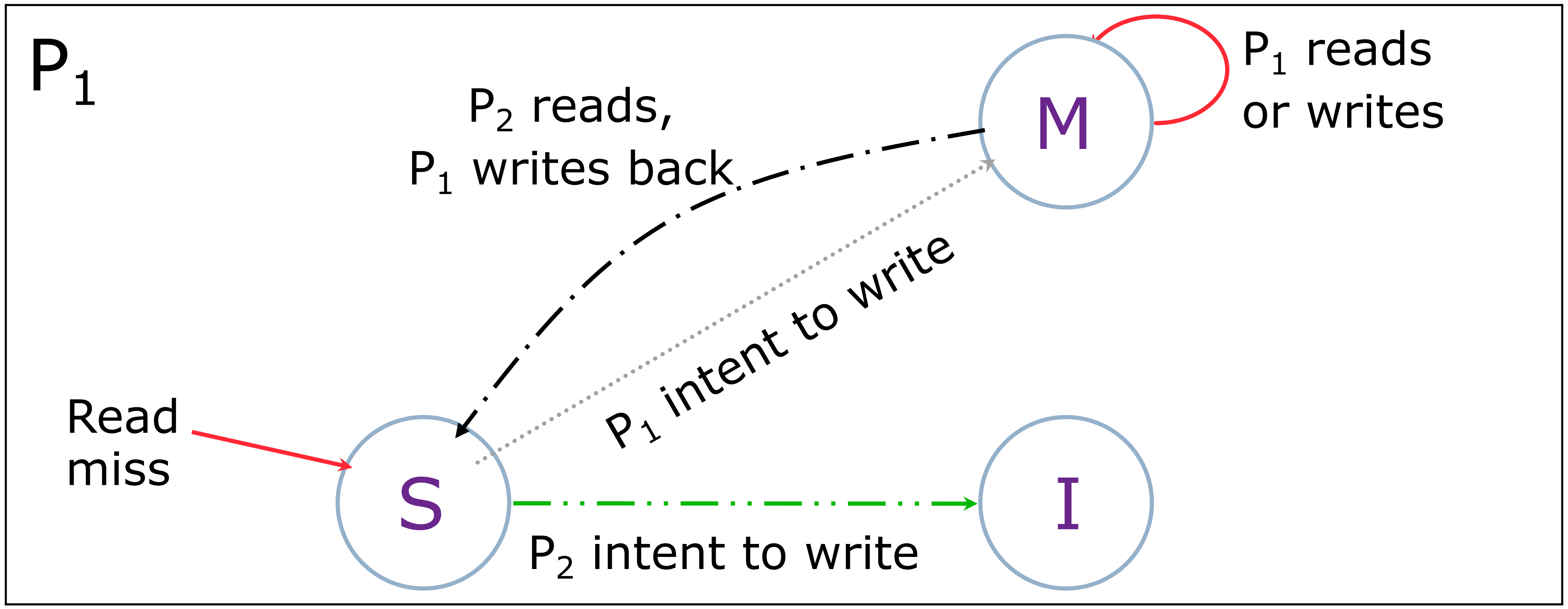
P_1 writes

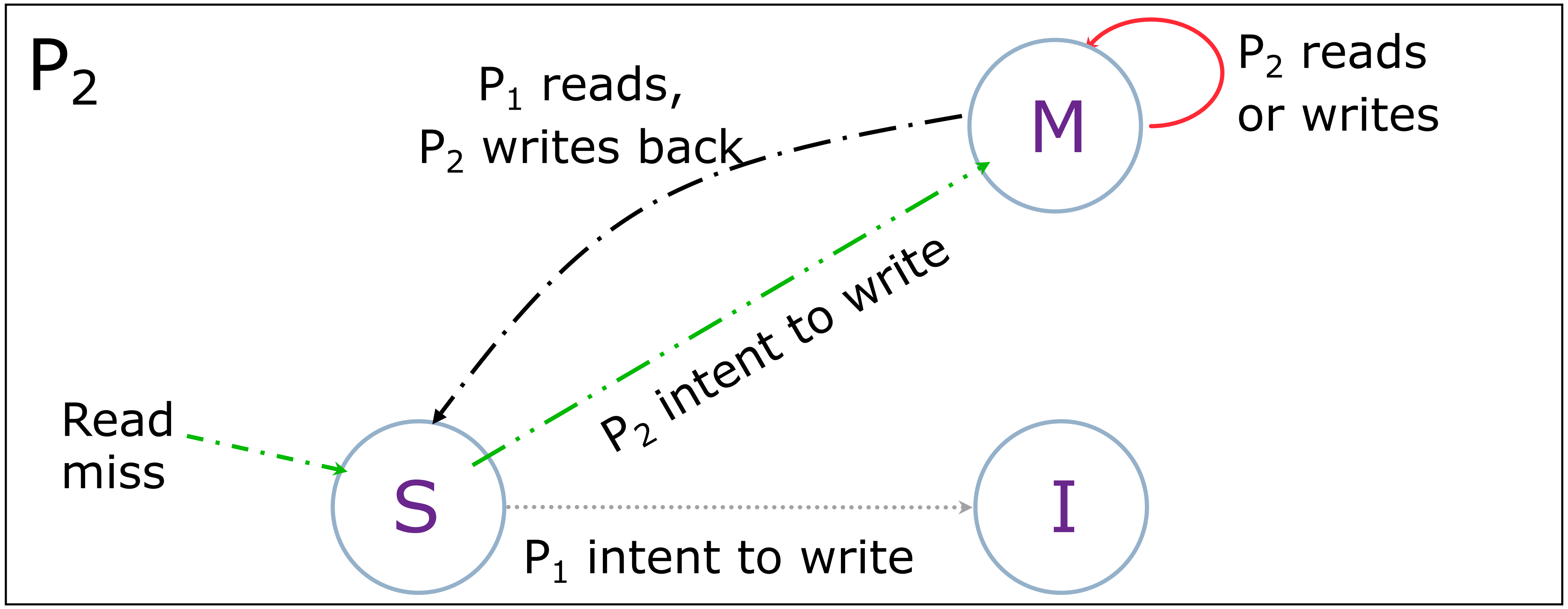




Two Processor Example
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes





Two Processor Example
(Reading and writing the same cache line)

P_1 reads

P_1 writes

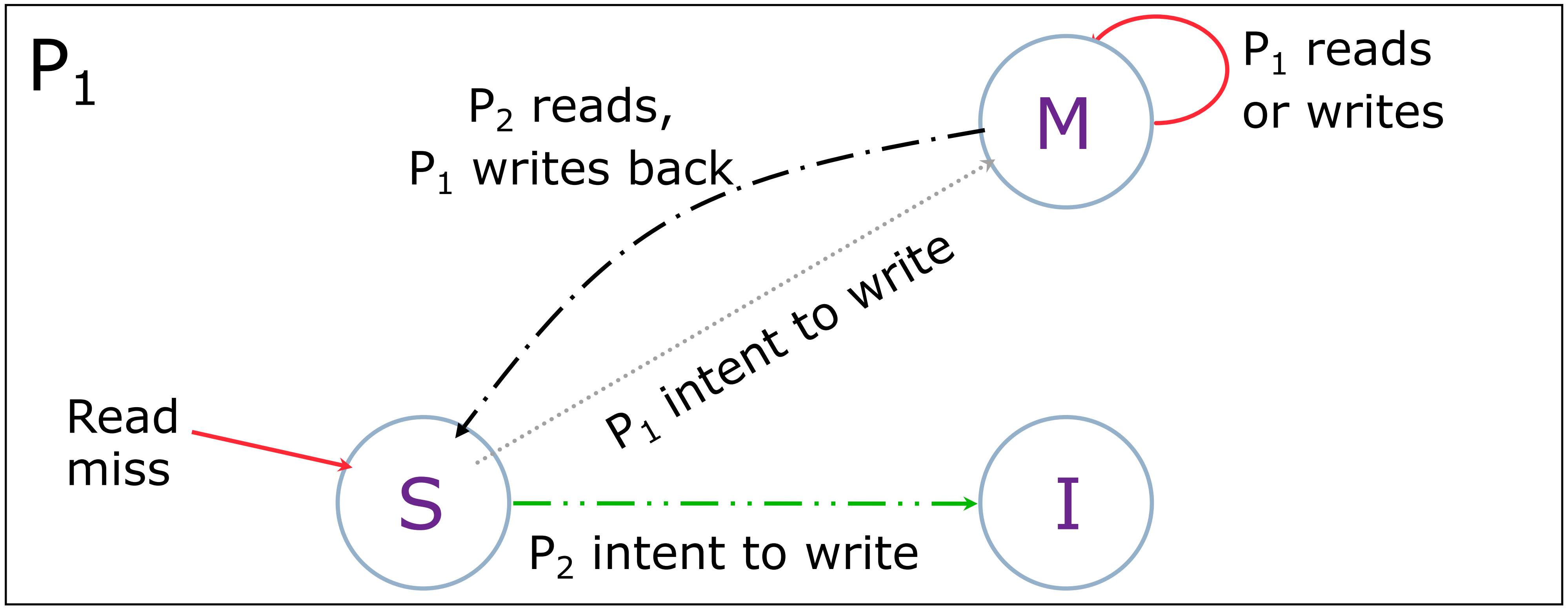
P_2 reads

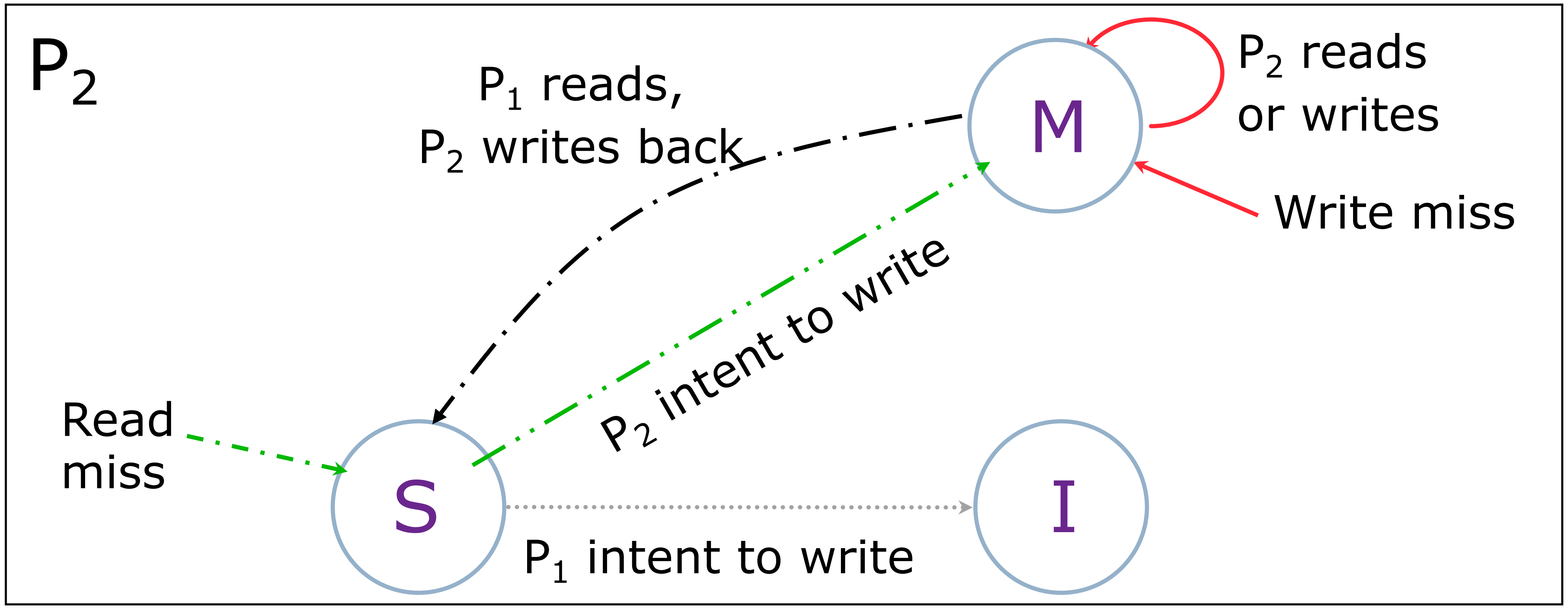
P_2 writes

P_1 reads

P_1 writes

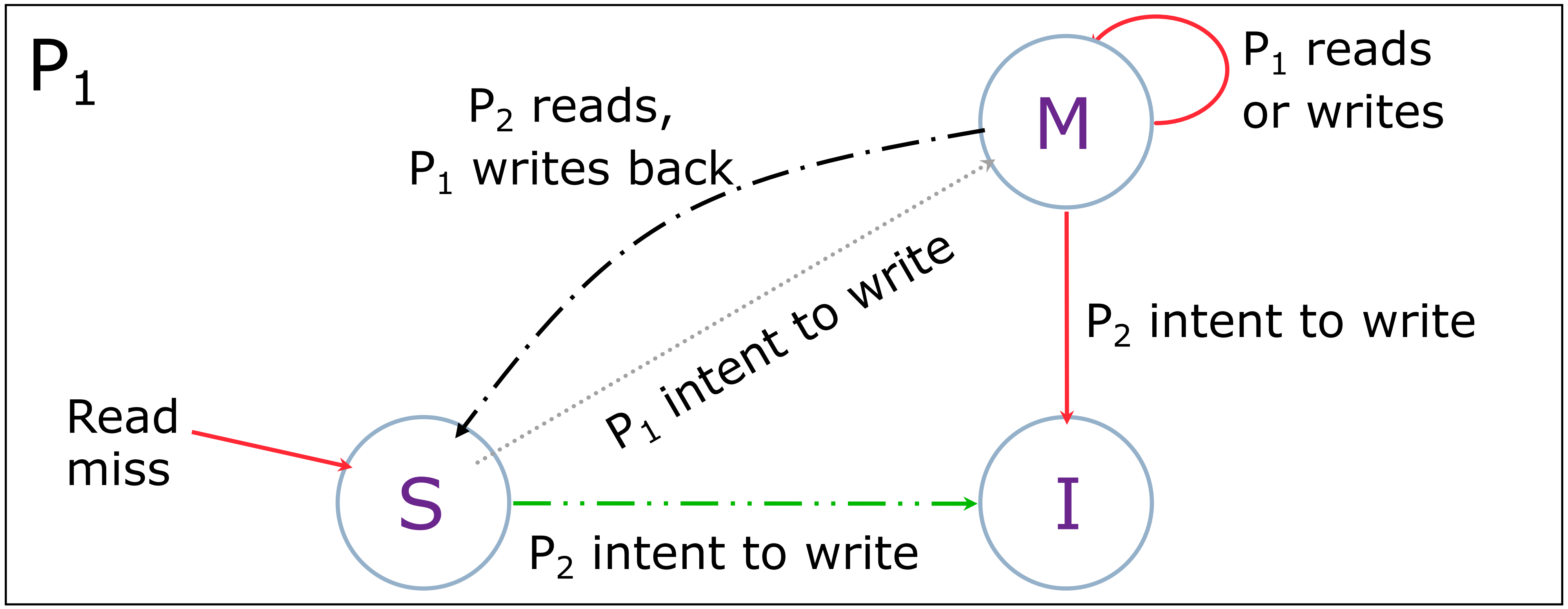
P_2 writes

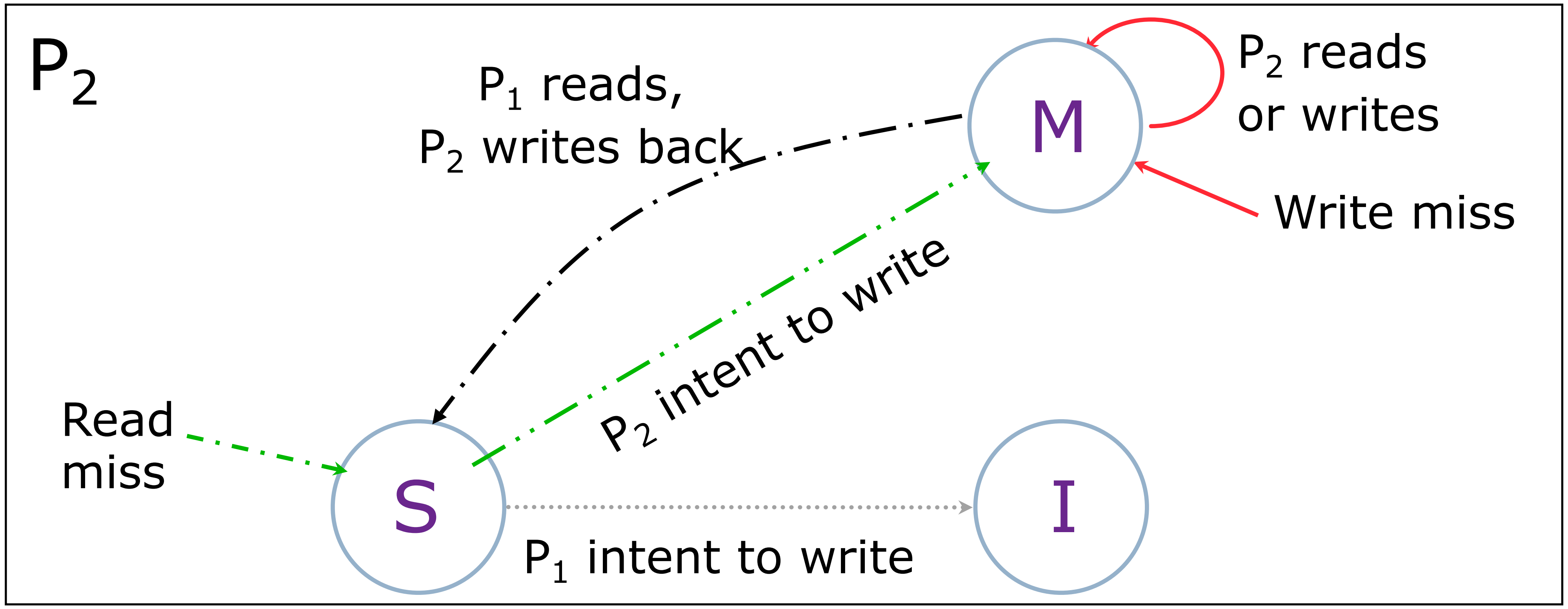




Two Processor Example
(Reading and writing the same cache line)

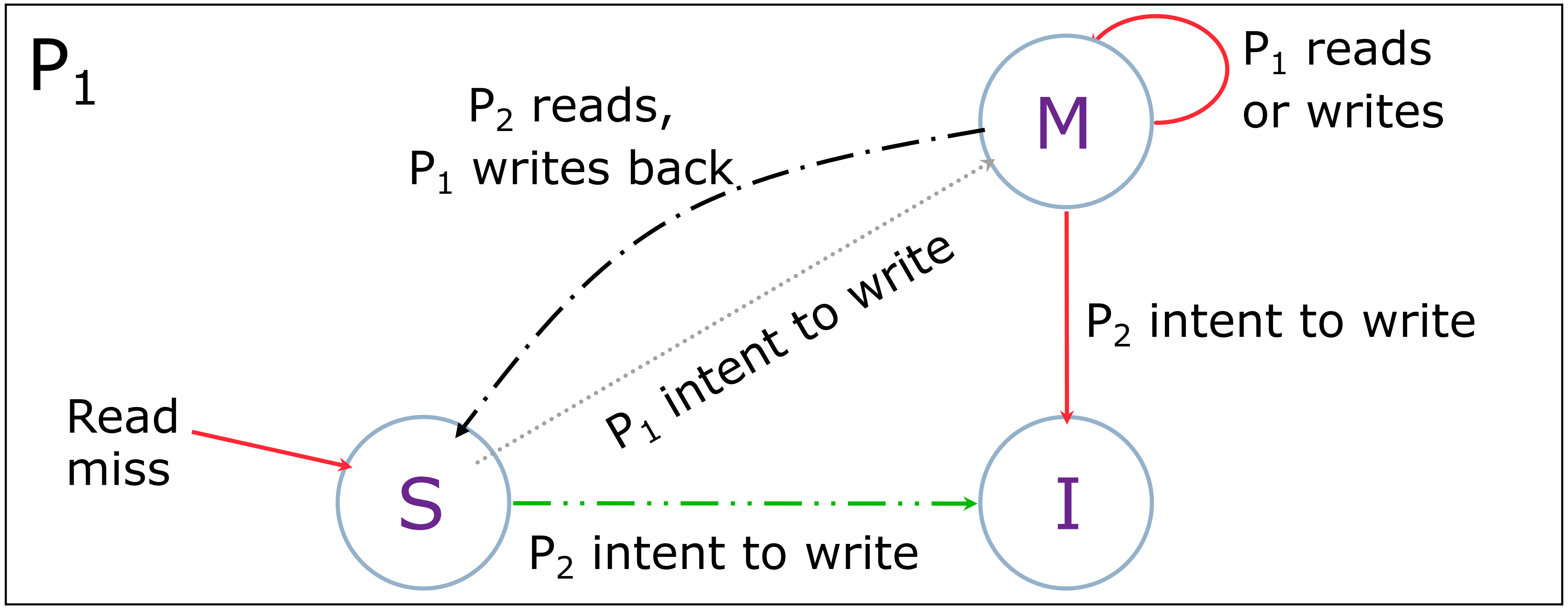
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes

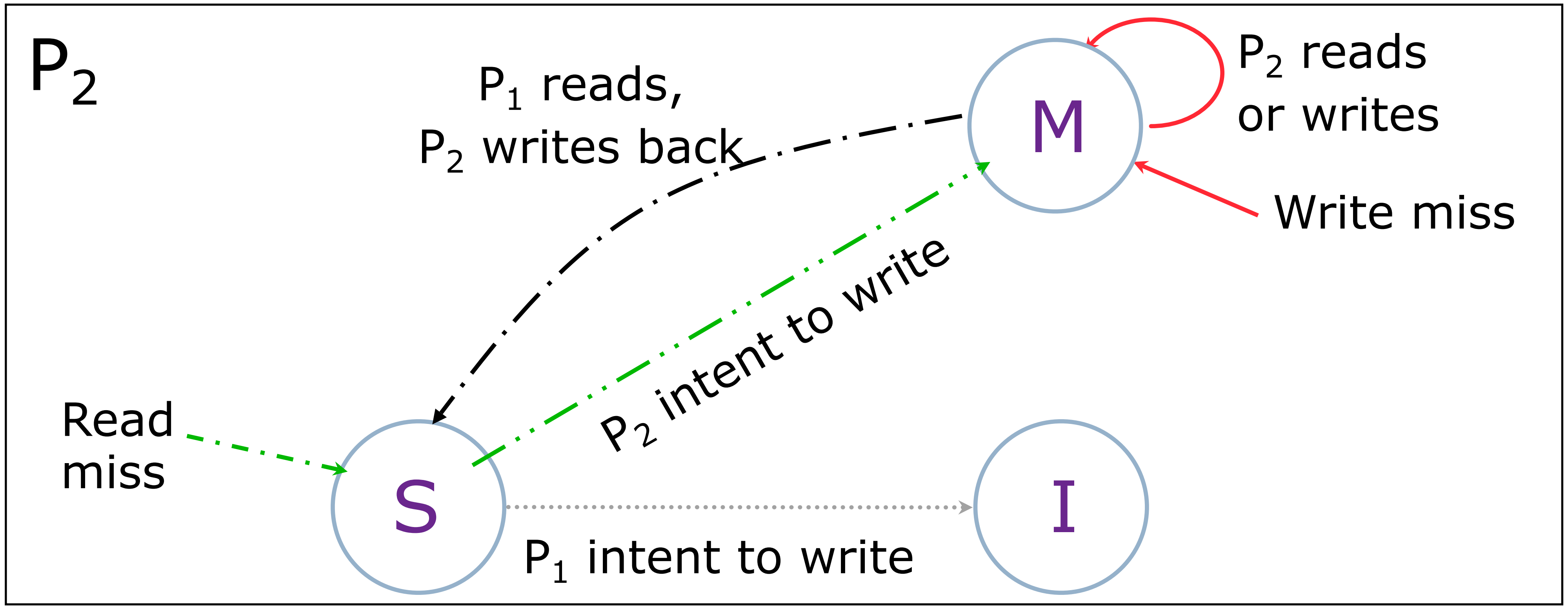




Two Processor Example
(Reading and writing the same cache line)

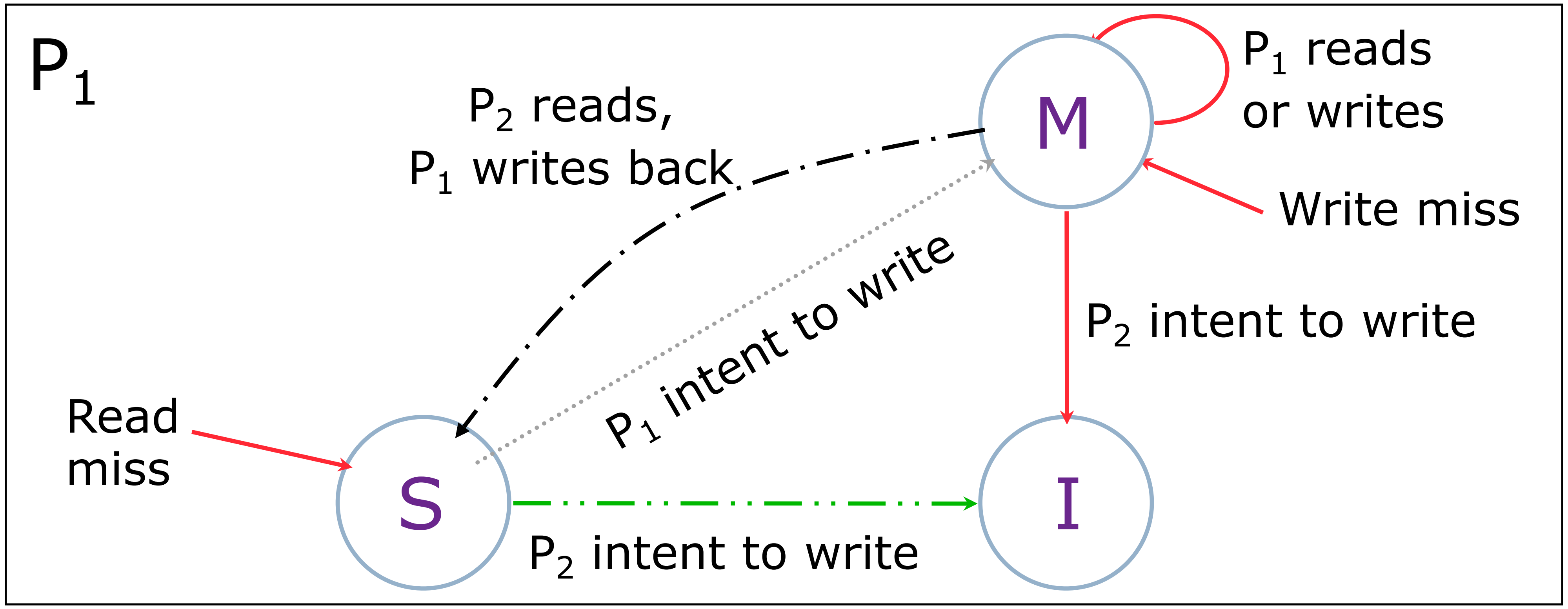
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes

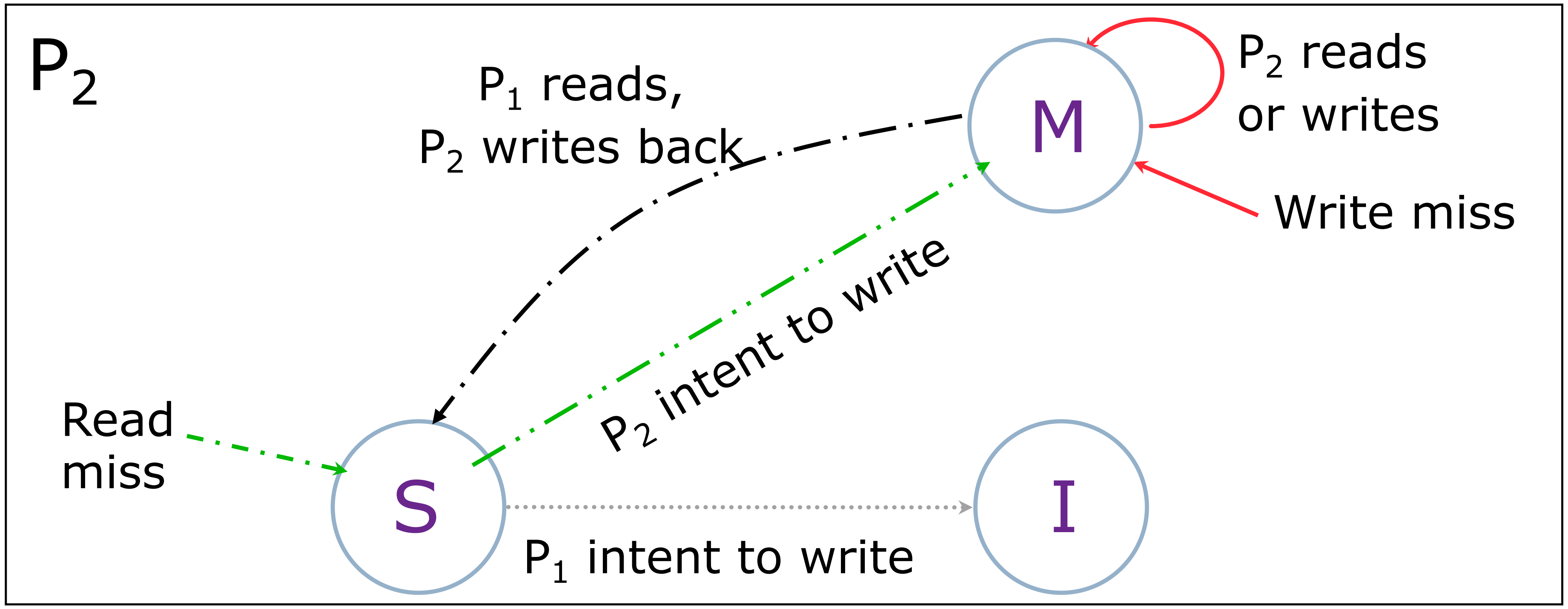




Two Processor Example
(Reading and writing the same cache line)

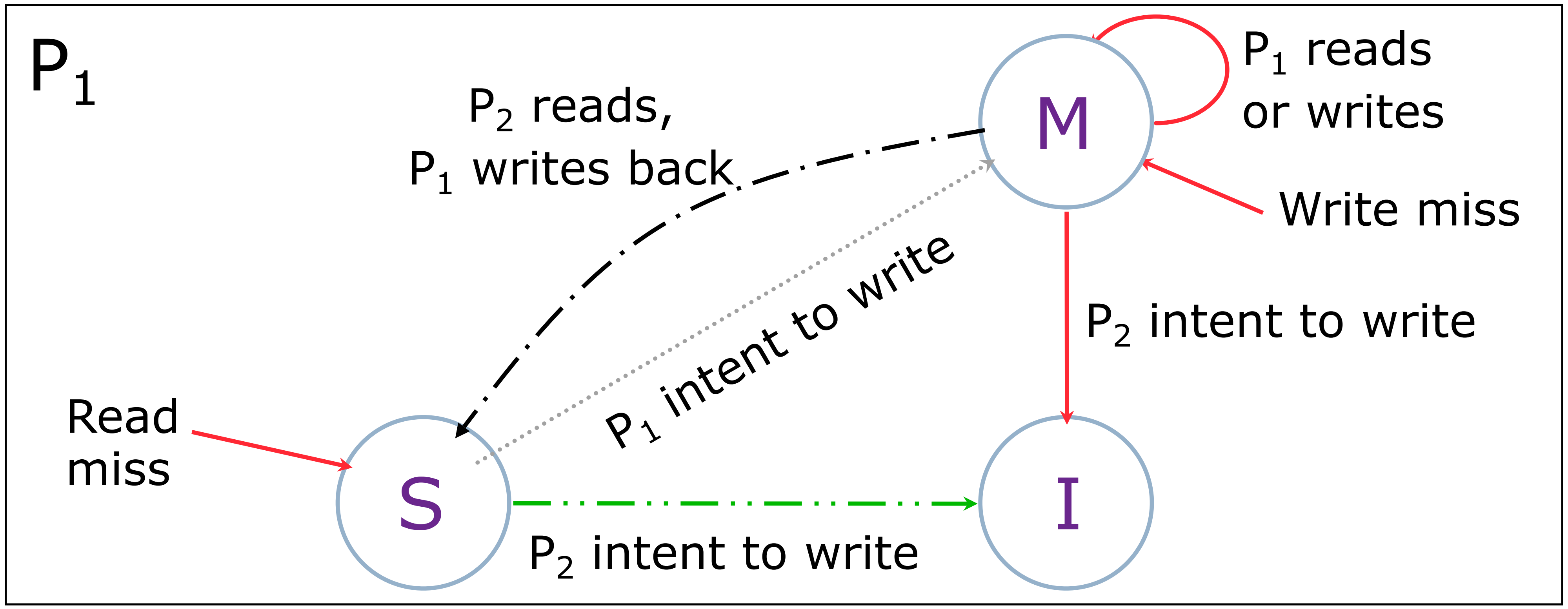
P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes

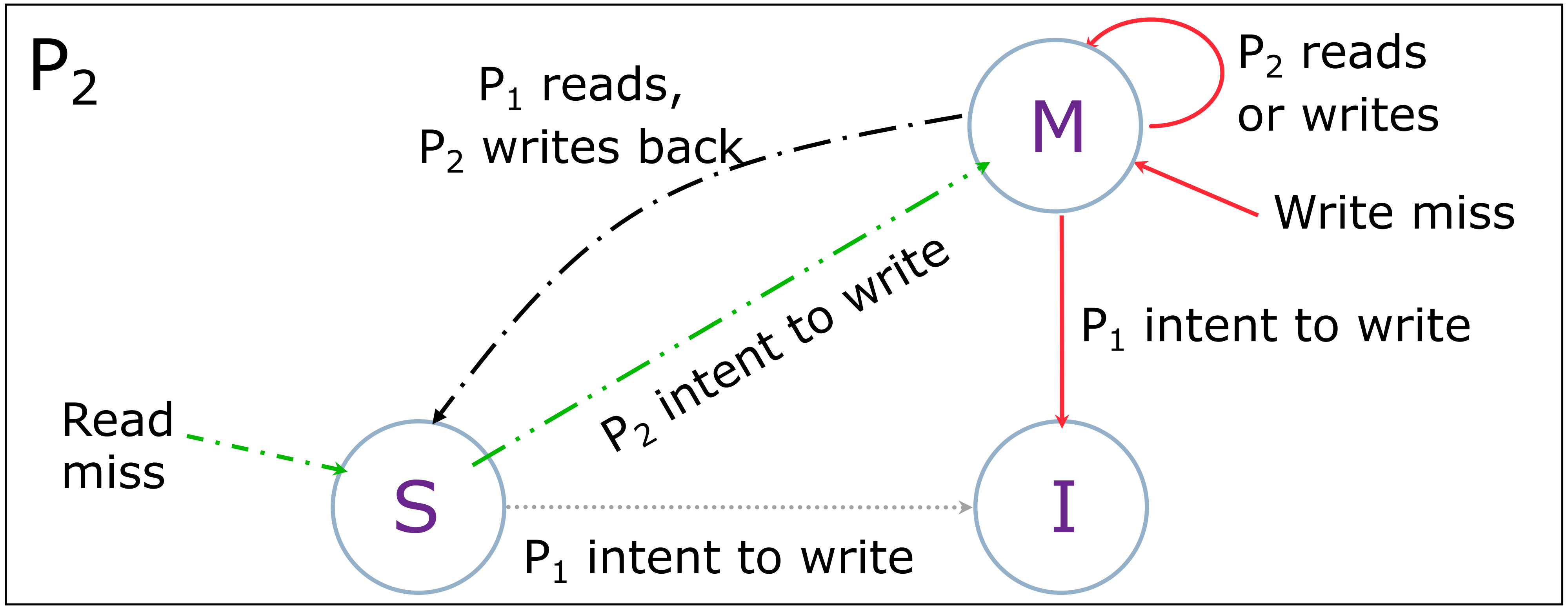




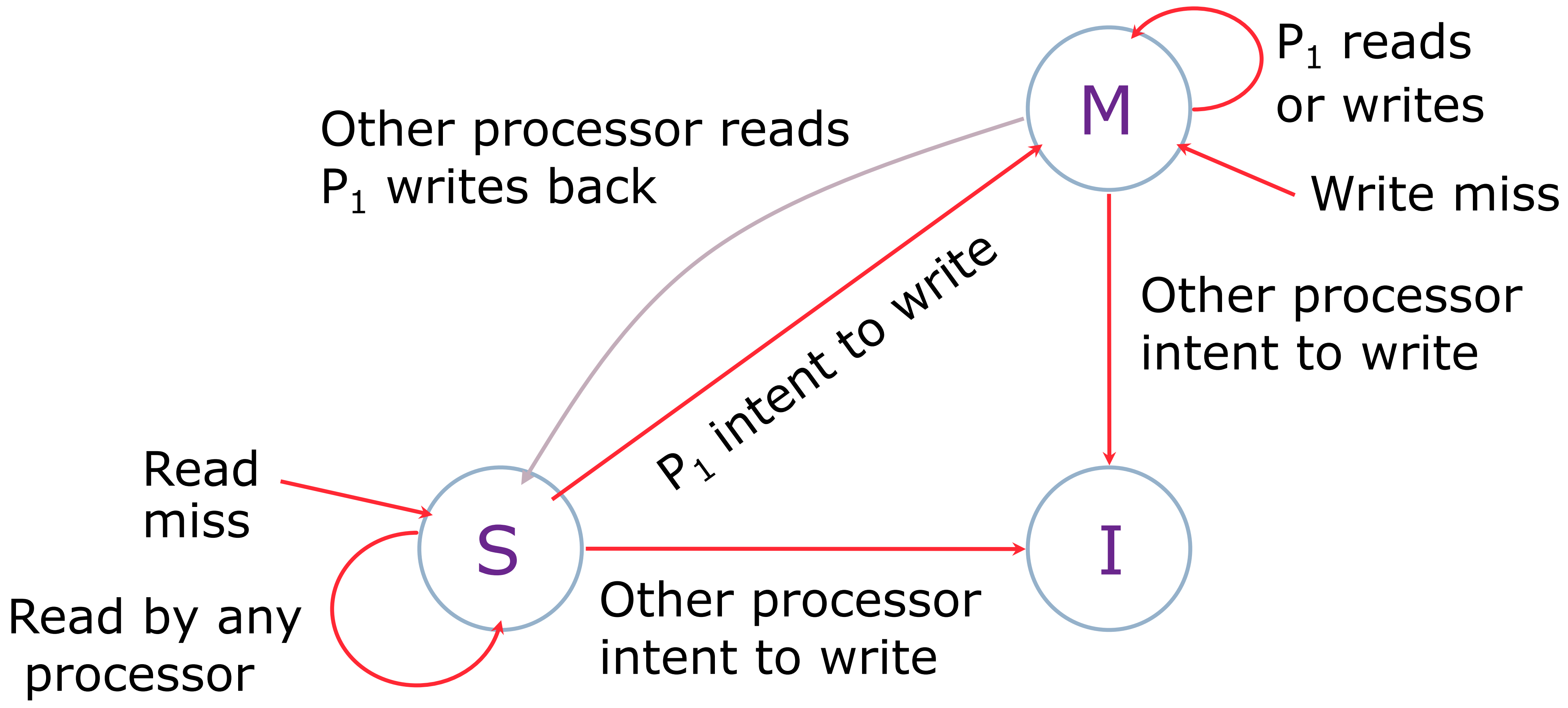
Two Processor Example
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes



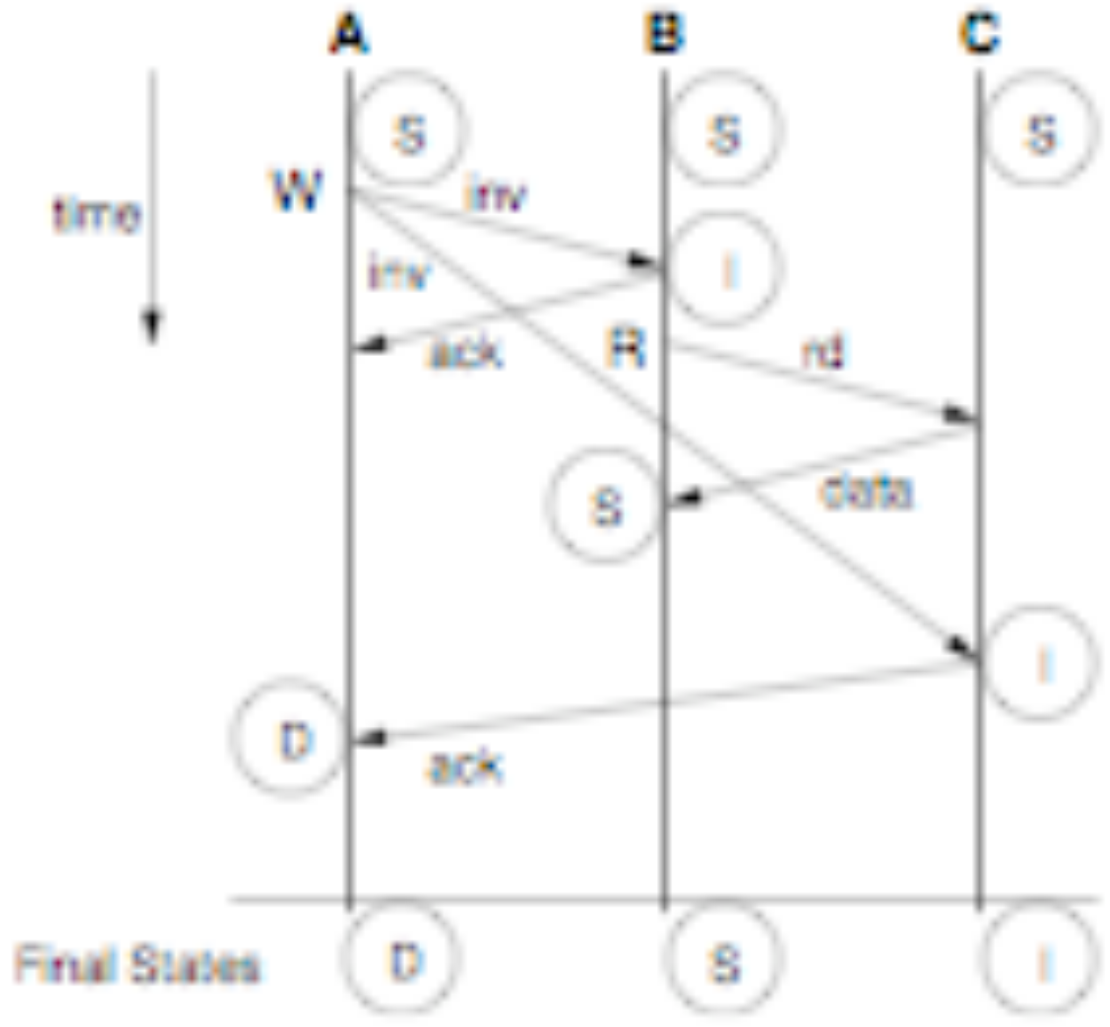


Observation

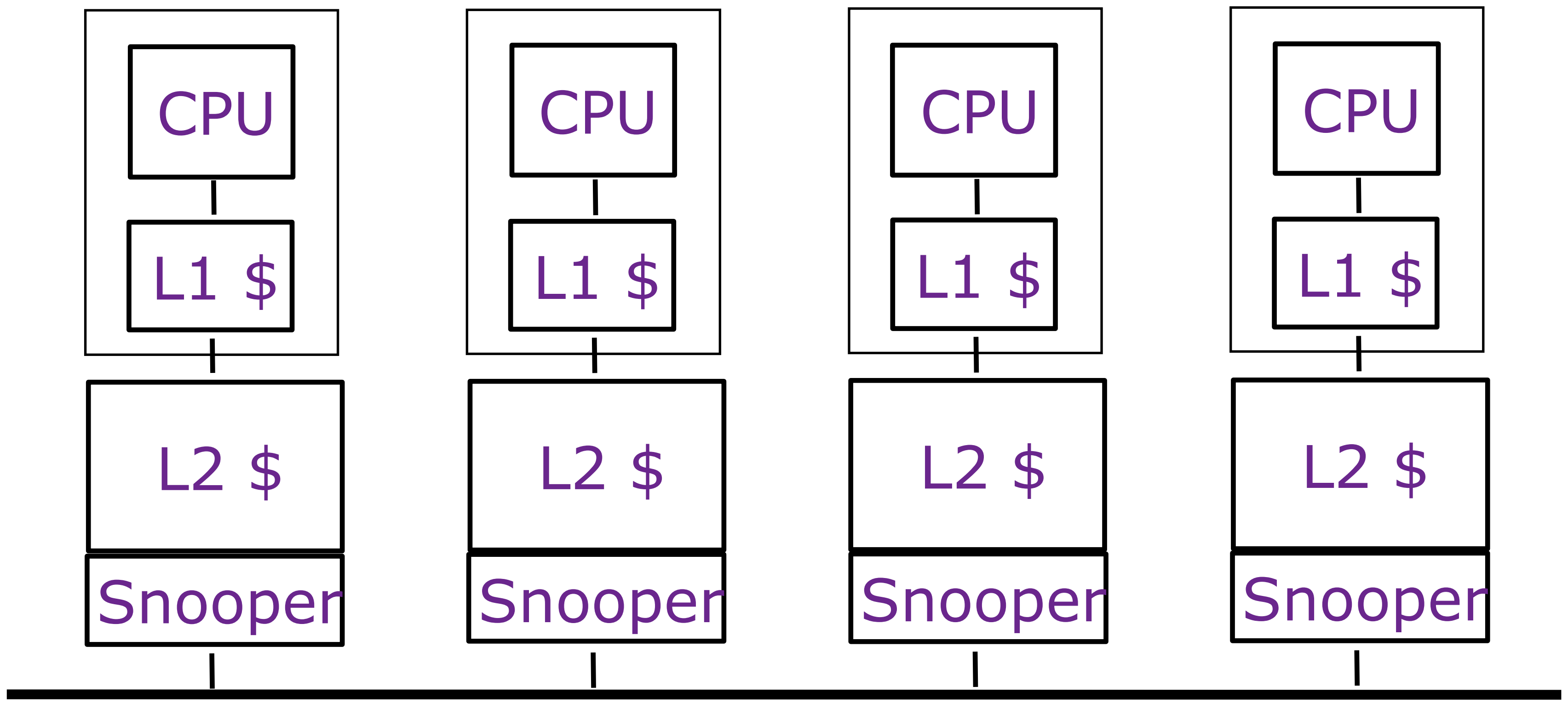

M
P1 reads or writes
Other processor reads P1 writes back
Write miss
P1 intent to write
Other processor intent to write
Read miss
S
Read by any processor
Other processor intent to write
I

- If a line is in the M state then no other cache can have a copy of the line!
  - Memory stays coherent, multiple differing copies cannot exist

Serialization is Important



Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- *Inclusion property:* entries in L1 must be in L2

  invalidation in L2 $\Rightarrow$ invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

What problem could occur?

Coherency Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU of modified block in different cache
   - Miss would still occur if block size were 1 word
2. False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared
     $\Rightarrow$ miss would not occur if block size were 1 word

Example: True v. False Sharing v. Hit?

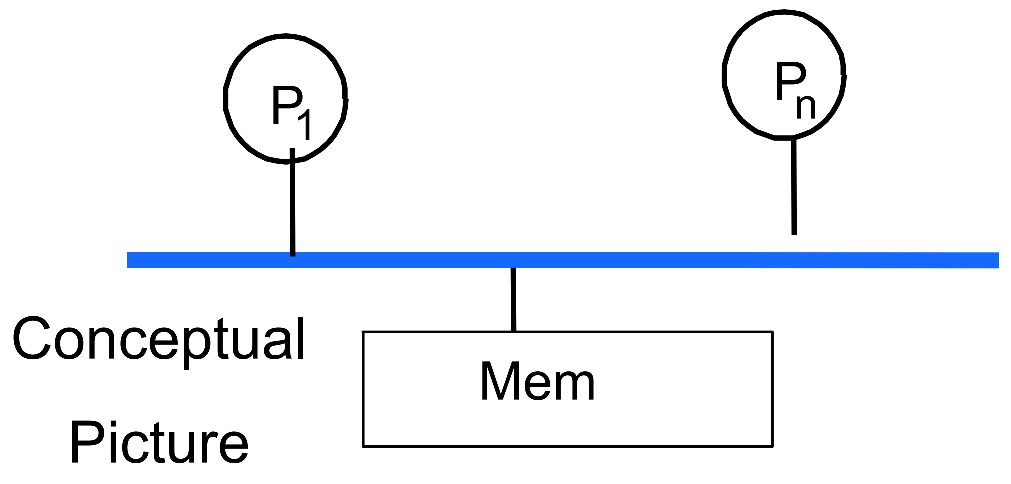
- Assume x1 and x2 in same cache block.
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	**True, False, Hit? Why?**
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

Coherence is not enough

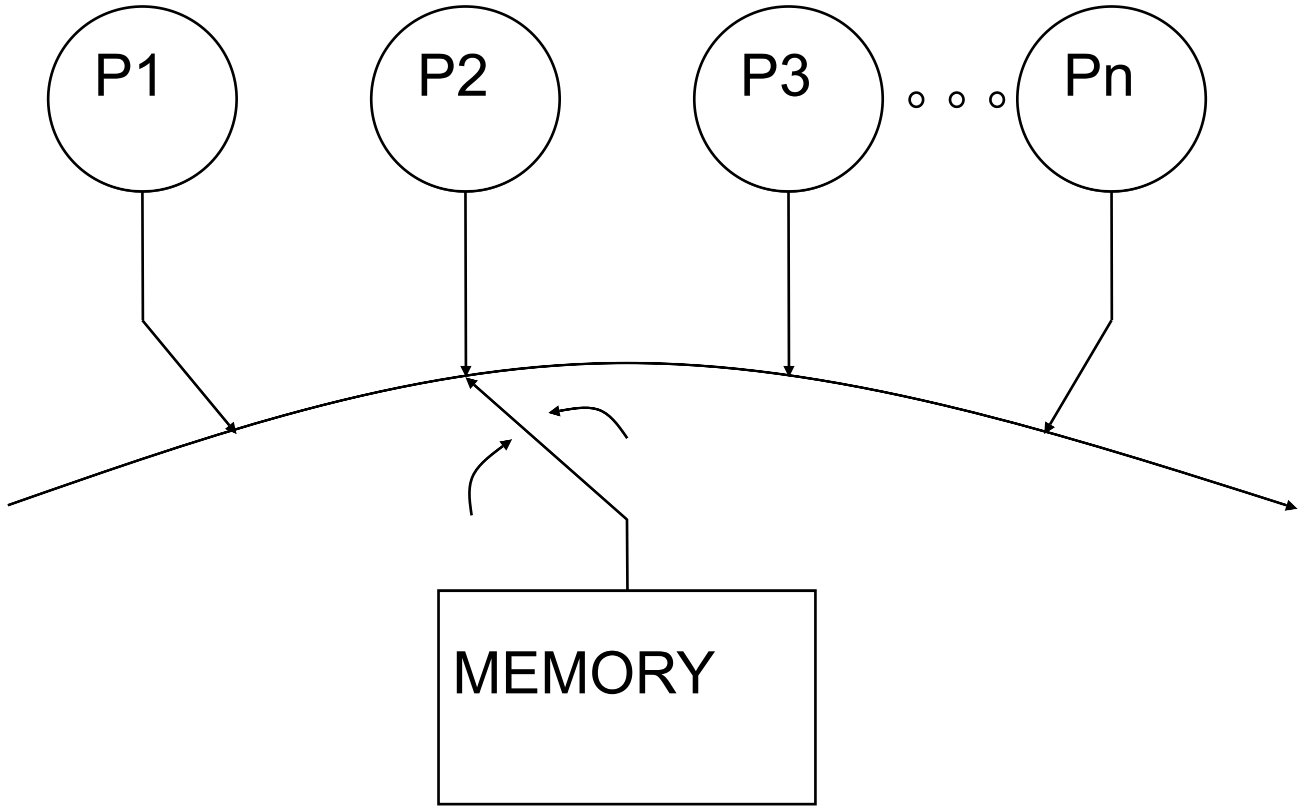
P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



Implicit Memory Model

- Sequential consistency (SC) [Lamport]
  - Result of an execution appears as if
    - All operations executed in some sequential order
    - Memory operations of each process in program order



- No caches, no write buffers

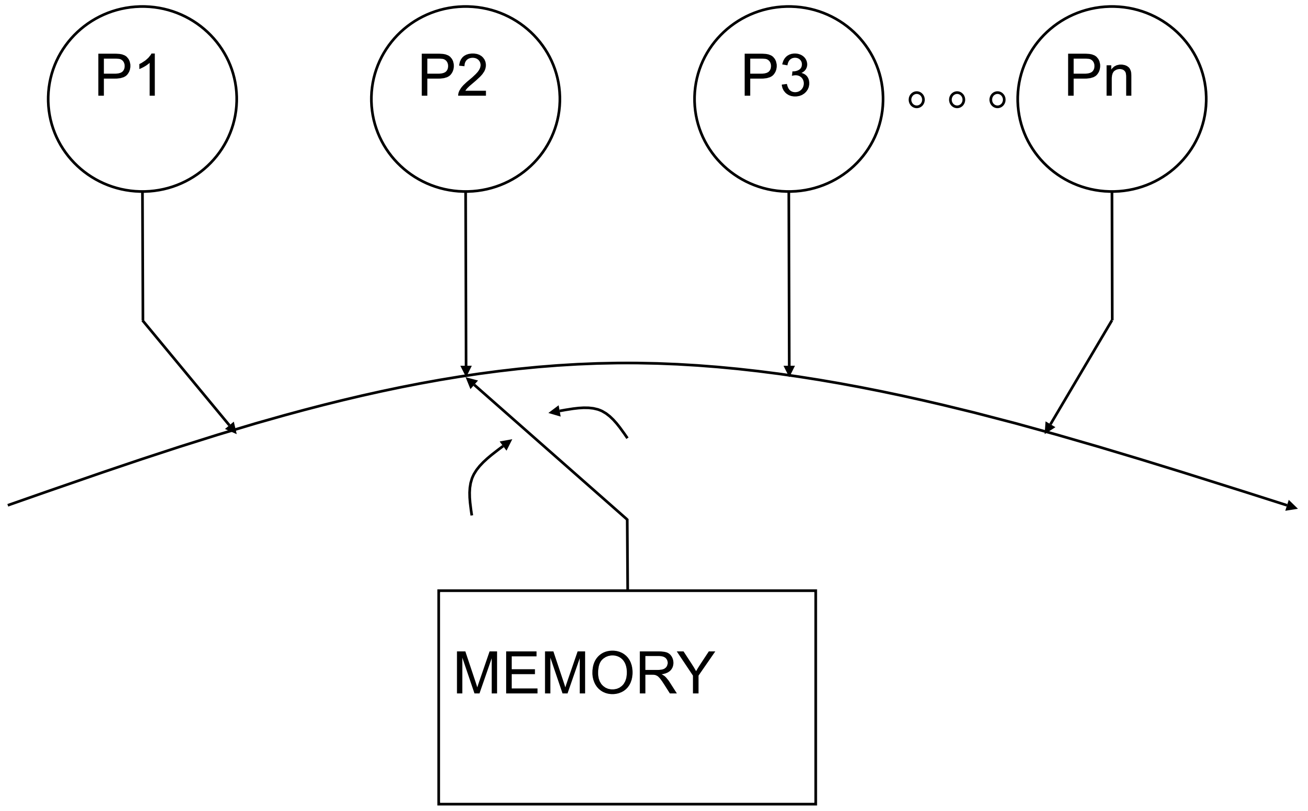
Implicit Memory Model

- Sequential consistency (SC) [Lamport]
  - Result of an execution appears as if
    - All operations executed in some sequential order
    - Memory operations of each process in program order

Two aspects:

Program order

Atomicity



- No caches, no write buffers

Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

```
T1:                              T2:
    Store (X), 1   (X = 1)           Load R1, (Y)
    Store (Y), 11  (Y = 11)          Store (Y'), R1  (Y'= Y)
                                           Load R2, (X)
                                           Store (X'), R2  (X'= X)
```

what are the legitimate answers for X′ and Y′ ?

$(X',Y') \in \{(1,11), (0,10), (1,10), (0,11)\}$?

Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:
    Store (X), 1 *(X = 1)*
    Store (Y), 11 *(Y = 11)*

T2:
    Load R_1, (Y)
    Store (Y'), R_1 *(Y'= Y)*
        Load R_2, (X)
        Store (X'), R_2 *(X'= X)*

what are the legitimate answers for X' and Y' ?

$(X',Y') \varepsilon \{(1,11), (0,10), (1,10), \cancel{(0,11)}\}$?

Sequential Consistency

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:

Store (X), 1 *(X = 1)*
Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)
Store (Y'), R_1 *(Y'= Y)*
Load R_2, (X)
Store (X'), R_2 *(X'= X)*

what are the legitimate answers for X' and Y' ?

(X',Y') ε {(1,11), (0,10), (1,10), ~~(0,11)~~} ?

If y is 11 then x cannot be 0

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies ( ) $\longrightarrow$

What are these in our example ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)

Store (Y’), R_1 *(Y’= Y)*

Load R_2, (X)

Store (X’), R_2

(X’= X)

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (   ) $\longrightarrow$

What are these in our example ?

T1:

Store (X), 1   *(X = 1)*

Store (Y), 11 *(Y = 11)*

(X′= X)

T2:

$$\begin{tikzcd}\text{Load } R_1, (Y) \arrow[d, bend right] \\ \text{Store } (Y'), R_1 \ \textit{(Y'= Y)}\end{tikzcd}$$

Load R_2, (X)

Store (X′), R_2

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (   ) $\longrightarrow$

What are these in our example ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

(X′= X)

T2:

$$\begin{tikzcd}[row sep=small] \text{Load } R_1, (Y) \arrow[d, bend right] \\ \text{Store } (Y'), R_1 \ \textit{(Y'= Y)} \end{tikzcd}$$

$$\begin{tikzcd}[row sep=small] \text{Load } R_2, (X) \arrow[d, bend right] \\ \text{Store } (X'), R_2 \end{tikzcd}$$

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies ( ) $\longrightarrow$

What are these in our example ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)

Store (Y'), R_1 *(Y'= Y)*

Load R_2, (X)

Store (X'), R_2

$\longrightarrow$ additional SC requirements

(X'= X)

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies (   ) $\longrightarrow$

What are these in our example ?

T1:

$$\begin{tikzcd}[row sep=small]
\text{Store (X), 1} \quad \textit{(X = 1)} \arrow[d, bend right] \\
\text{Store (Y), 11} \quad \textit{(Y = 11)}
\end{tikzcd}$$

T2:

$$\begin{tikzcd}[row sep=small]
\text{Load } R_1\text{, (Y)} \arrow[d, bend right] \\
\text{Store (Y'), } R_1 \quad \textit{(Y'= Y)} \\
\text{Load } R_2\text{, (X)} \arrow[d, bend right] \\
\text{Store (X'), } R_2
\end{tikzcd}$$

$\longrightarrow$ additional SC requirements

(X'= X)

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies ( ) $\longrightarrow$

What are these in our example ?

T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)

Store (Y'), R_1 *(Y'= Y)*

Load R_2, (X)

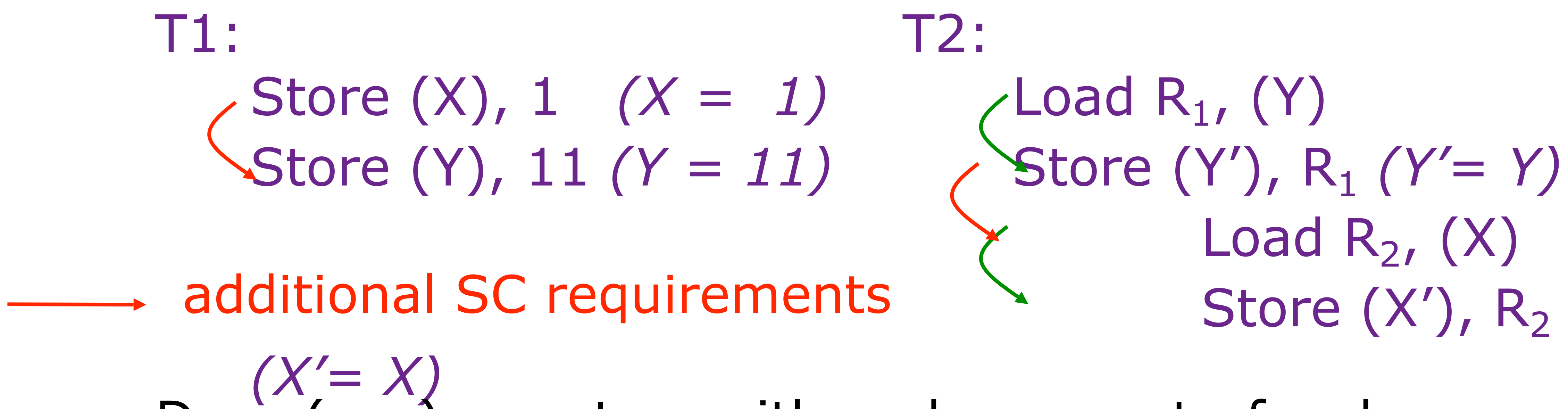
Store (X'), R_2 *(X'= X)*

$\longrightarrow$ additional SC requirements

Sequential Consistency

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies ($\longrightarrow$)

What are these in our example ?



T1:

Store (X), 1 *(X = 1)*

Store (Y), 11 *(Y = 11)*

T2:

Load R_1, (Y)

Store (Y'), R_1 *(Y'= Y)*

Load R_2, (X)

Store (X'), R_2

$\longrightarrow$ additional SC requirements

(X'= X)

Does (can) a system with caches or out-of-order execution capability provide a *sequentially consistent* view of the memory ?

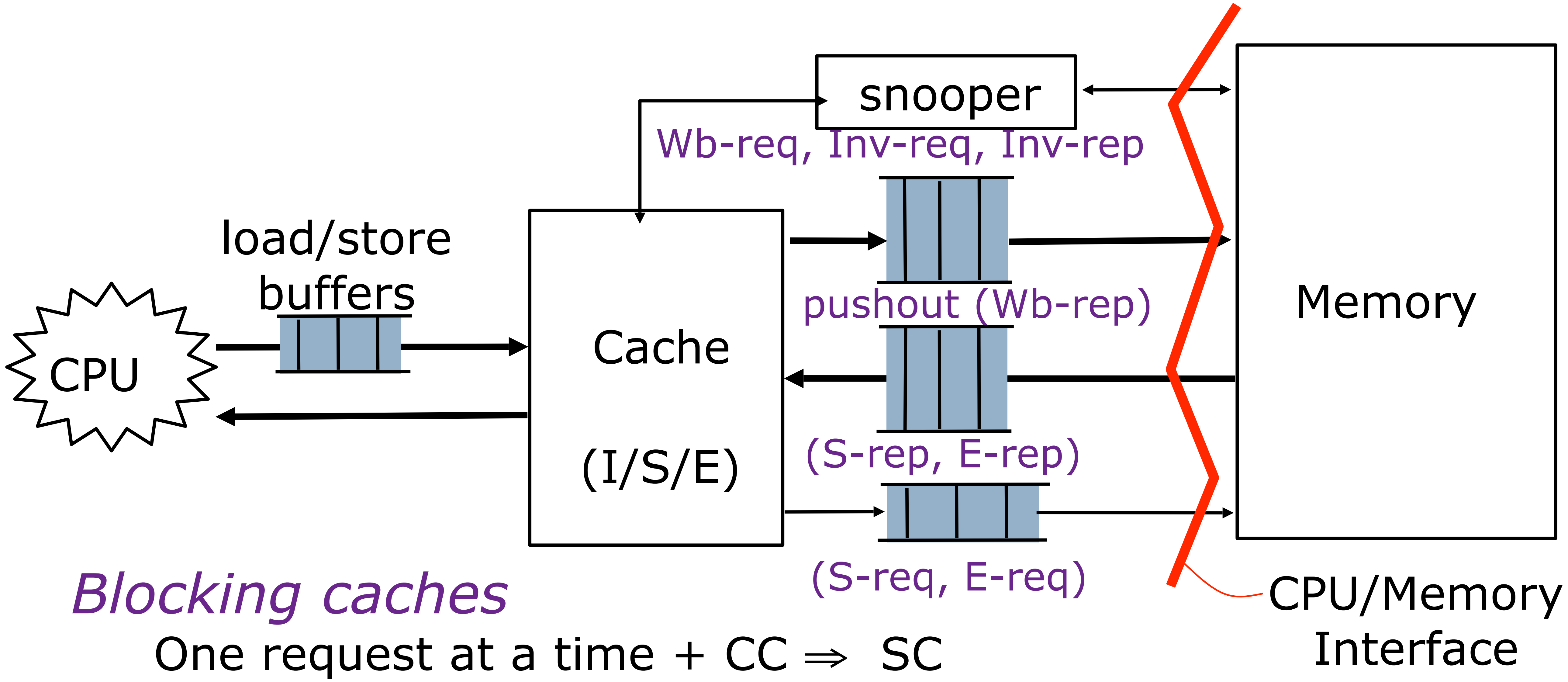
Sequential Consistency

- SC constrains all memory operations:
    - » Write → Read
    - » Write → Write
    - » Read → Read, Write

- Simple model for reasoning about parallel programs
- But, intuitively reasonable reordering of memory operations in a uniprocessor may violate sequential consistency model

- Modern microprocessors reorder operations all the time to obtain performance (write buffers, overlapped writes,non-blocking reads…).
- Question: how do we reconcile sequential consistency model with the demands of performance?

Out-of-Order Loads/Stores & CC



Blocking caches

One request at a time + CC $\Rightarrow$ SC

Non-blocking caches

Multiple requests (different addresses) concurrently + CC

$\Rightarrow$ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address

SC is fragile

- Many common optimizations break it…
- Write Buffer
- Out-of-order execution
- Forwarding

Notes

- Sequential consistency is not really about memory operations from different processors (although we do need to make sure memory operations are atomic).
- Sequential consistency is not really about dependent memory operations in a single processor's instruction stream (these are respected even by processors that reorder instructions).
- The problem of relaxing sequential consistency is really all about independent memory operations in a single processor's instruction stream that have some high-level dependence (such as locks guarding data) that should be respected to obtain correct results.

Relaxing Program Orders

- Weak ordering:
  - Divide memory operations into data operations and synchronization operations
  - Synchronization operations act like a fence:
    - All data operations before synch in program order must complete before synch is executed
    - All data operations after synch in program order must wait for synch to complete
    - Synchs are performed in program order
  - Implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed
  - Example: PowerPC has SYNC instruction (caveat: semantics somewhat more complex than what we have described…)

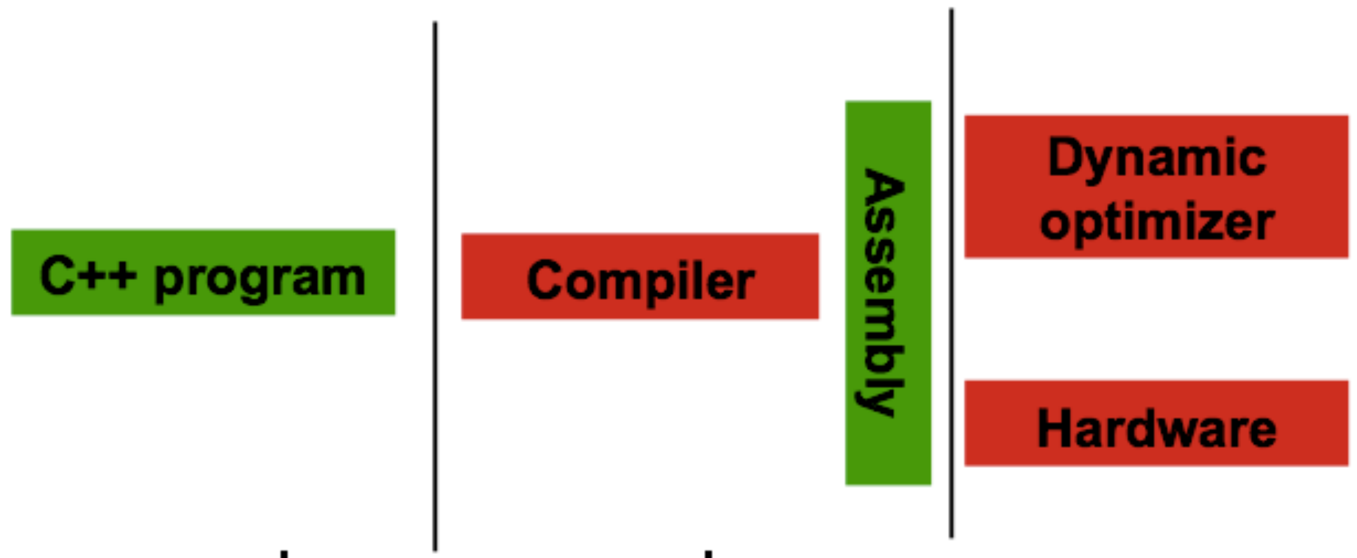
Another model: Release consistency

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
  - but accesses after release in program order do not have to wait for release
  - operations which follow release and which need to wait must be protected by an acquire

Some Current System-Centric Models

Relaxation:	W →R Order	W →W Order	R →RW Order	Read Others' Write Early	Read Own Write Early	Safety Net
IBM 370	✓					serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	synchronization
RCsc	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

It is all about the interfaces



Language				
Compiler	*Strong* -reordering has to obey language model -not much room for reordering	*Strong* -reordering has to obey language model *-needs to insert fences to map language model to weaker ISA model* -not much room for reordering	*Weak* -room for reordering *-no worries about the HW reordering*	*Weak* -room for reordering
Hardware	*Strong* -reordering has to obey ISA or not be visible to SW	*Weak* -reordering may be visible to SW -room for reordering	*Strong* -reordering has to obey ISA or not be visible to SW	*Weak* -reordering may be visible to SW -room for reordering

Synchronization

- Shared counter/sum update example
  - Use a mutex variable for mutual exclusion
  - Only one processor can own the mutex
    - » Many processors may call lock(), but only one will succeed (others block)
    - » The winner updates the shared sum, then calls unlock() to release the mutex
    - » Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - » As a shared variable (1 – owned, 0 –free)

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - Need a way to ***atomically***
    see that the mutex is 0 ***and*** set it to 1

Atomic Read-Update Instructions

- Atomic exchange instruction
  - E.g., EXCH R1,78(R2) will swap content of register R1 and mem location at address 78+R2
  - To acquire a mutex, 1 in R1 and EXCH
    - » Then look at R1 and see whether mutex acquired
    - » If R1 is 1, mutex was owned by somebody else and we will need to try again later
    - » If R1 is 0, mutex was free and we set it to 1, which means we have acquired the mutex
- Other atomic read-and-update instructions
  - E.g., Test-and-Set

LL & SC Instructions

- Atomic instructions OK, but specialized
  - E.g., SWAP can not atomically inc a counter
- Idea: provide a pair of linked instructions
- A load-linked (LL) instruction
  - Like a normal load, but also remembers the address in a special “link” register
- A store-conditional (SC) instruction
  - Like a normal store, but fails if its address is not the same as that in the link register
  - Returns 1 if successful, 0 on failure
- Writes by other processors snooped
  - If address matches link address, clear link register

Performance:
Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses
- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform a store each time

Using LL & SC

Swap R4 w/ 0(R1)

Atomic Exchange

```
swap:   mov     R3, R4
        ll      R2,0(R1)
        sc      R3,0(R1)
        beqz    R3,swap
        mov     R4,R2
```

Test if 0(R1) is zero, set to one

Atomic Test&Set

```
t&s:    mov     R3,1
        ll      R2,0(R1)
        sc      R3,0(R1)
        bnez    R2,t&s
        beqz    R3,t&s
```

Atomic Add to Shared Variable

```
upd:    ll      R2,0(R1)
        add     R3,R2,R4
        sc      R3,0(R1)
        beqz    R3,upd
```

Implementing Locks

- A simple swap (or test-and-set) works
  - But causes a lot of invalidations
    - » Every write sends an invalidation
    - » Most writes redundant (swap 1 with 1)
- More efficient: test-and-swap
  - Read, do swap only if 0
    - » Read of 0 does not guarantee success (not atomic)
    - » But if 1 we have little chance of success
  - Write only when good chance we will succeed

Large-Scale Systems: Locks

- Contention even with test-and-test-and-set
  - Every write goes to many, many spinning procs
  - Making everybody test less often reduces contention for high-contention locks but hurts for low-contention locks
  - Solution: exponential back-off
    - » If we have waited for a long time, lock is probably high-contention
    - » Every time we check and fail, double the time between checks
      - Fast low-contention locks (checks frequent at first)
      - Scalable high-contention locks (checks infrequent in long waits)
  - Special hardware support
  - Queuing locks

What Are the Problems With Locks?

- Mapping between data->locks
  - Deadlocks
  - Races
  - Composability?
- Mmm, DB?
  - Optimistic concurrency

What If you Had Multi-Word LL-SC?

- Plus the ability to execute stores speculatively
- => Transactional Memory
  - Speculative execution + monitor CC trafic

Barrier Synchronization

- All must arrive before any can leave
  - Used between different parallel sections
- Uses two shared variables
  - A counter that counts how many have arrived
  - A flag that is set when the last processor arrives

Simple Barrier Synchronization

```
lock(counterlock);
  if(count==0) release=0;  /* First resets release */
  count++;                 /* Count arrivals */
unlock(counterlock);
if(count==total){          /* All arrived */
  count=0;                 /* Reset counter */
  release = 1;             /* Release processes */
}else {                    /* Wait for more to come */
  spin(release==1);        /* Wait for release to be 1 */
}
```

- Problem: not really reusable
  - Two processes: fast and slow
  - Slow arrives first, reads release, sees 0
  - Fast arrives, sets release to 1, goes on to execute other code, comes to barrier again, resets release, starts spinning
  - Slow now reads release again, sees 0 again
  - Now both processors are stuck and will never leave

Correct Barrier Synchronization

```
localSense=!localSense;    /* Toggle local sense */
lock(counterlock);
  count++;                 /* Count arrivals */
  if(count==total){        /* All arrived */
    count=0;               /* Reset counter */
    release=localSense;    /* Release processes */
  }
unlock(counterlock);
spin(release==localSense); /* Wait to be released */
```

- Release in first barrier acts as reset for second
  - When fast comes back it does not change release, it just waits for it to become 0
  - Slow eventually sees release is 1, stops spinning, does work, comes back, sets release to 0, and both go forward.

init: localSense = 0, release = 0

Large-Scale Systems: Barriers

- Barrier with many processors
  - Have to update counter one by one – takes a long time
  - Solution: use a combining tree of barriers
    - » Example: using a binary tree
    - » Pair up processors, each pair has its own barrier
      - E.g. at level 1 processors 0 and 1 synchronize on one barrier, processors 2 and 3 on another, etc.
    - » At next level, pair up pairs
      - Processors 0 and 2 increment a count a level 2, processors 1 and 3 just wait for it to be released
      - At level 3, 0 and 4 increment counter, while 1, 2, 3, 5, 6, and 7 just spin until this level 3 barrier is released
      - At the highest level all processes will spin and a few “representatives” will be counted.
    - » Works well because each level fast and few levels
      - Only 2 increments per level, log2(numProc) levels
      - For large numProc, 2*log2(numProc) still reasonably small

Summary

* No more ideas for ILP, must go TLP
* To first order, no one understands how to program multithreaded code (approximately nobody)
* Brave new world for VN computing