Topic V29

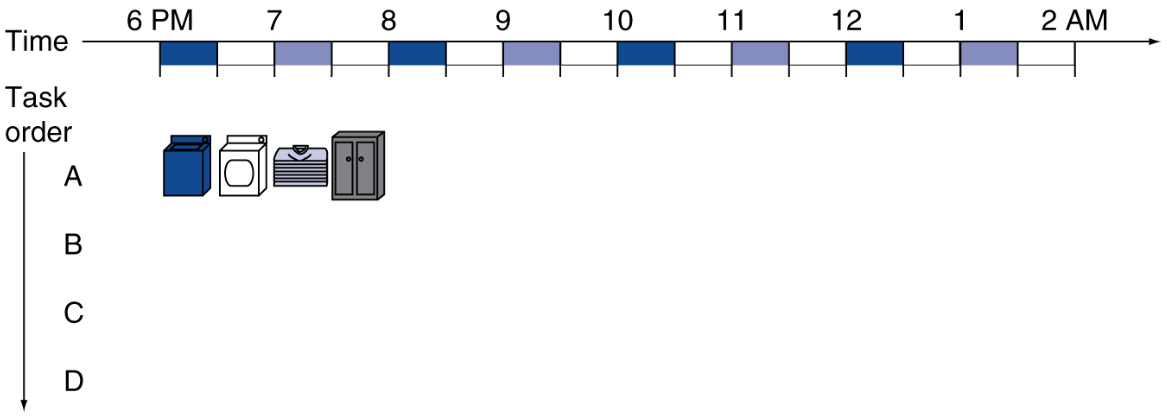
Pipelining

Reading: (Section 4.6 and 4.7)

Pipelining Analogy

Pipelined laundry: overlapping execution

Parallelism improves performance



Four loads:

What is the speedup if you need to do four loads of laundry?

$$Speedup = \frac{8}{3.5} = 2.3$$

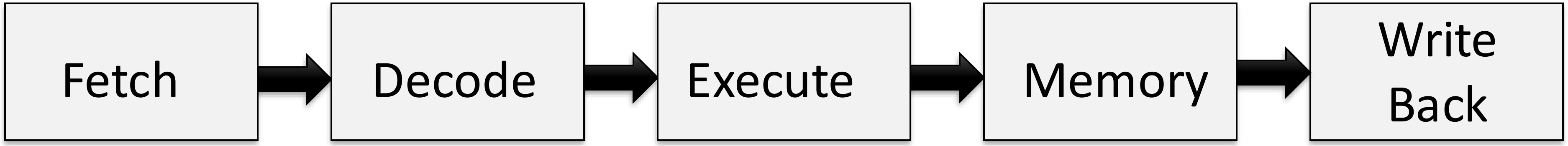
Non-stop:

$$Speedup = \frac{2n}{0.5n} + 1.5 \approx 4$$

What is the speedup if you need to do many loads of laundry?

$Speedup \approx$ number of stages

RISC-V Pipeline



Fetch → Decode → Execute → Memory → Write Back

Five stages, one step per stage

- IF: Instruction fetch from memory
- ID: Instruction decode & register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register

Pipeline Performance

Light travels less than 3cm during this time. Electrons can be much slower.

Assume time for stages is

- 100ps for register read or write
- 200ps for other stages

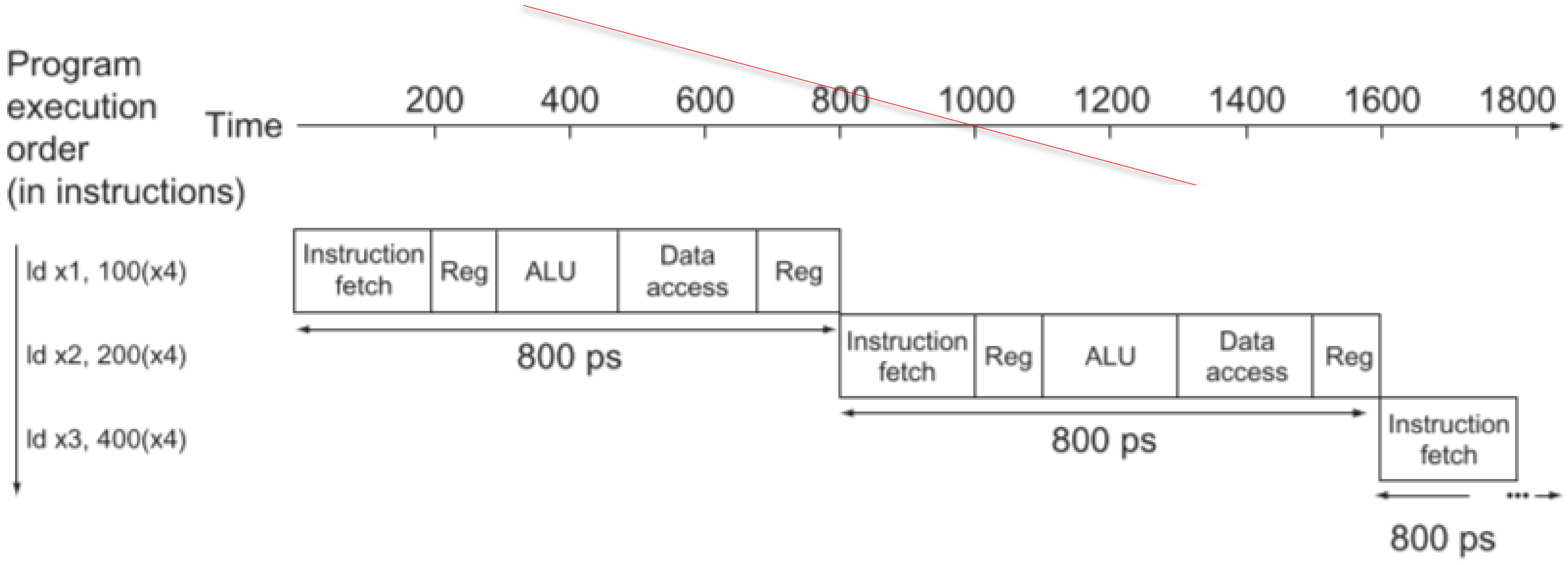
Grace Hopper nanosecond:
https://www.youtube.com/watch?v=9eyFDBPk4Yw

Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld						🟦
sd						
R-format						
beq						

Pipeline Performance

T_c: Time between completion of two consecutive instructions.



Pipeline Speedup

If all stages are balanced (i.e., all take the same time)

$$\text{Time between instructions pipelined} = \frac{\text{Time between instruction nonpipelined}}{\text{Number of stages}}$$

If not balanced, speedup is less

The speedup is due to increased throughput

Instructions/second increases

Latency does not decrease

Time to finish a single instruction is same or longer

Pipelining and ISA Design

RISC-V ISA designed for pipelining

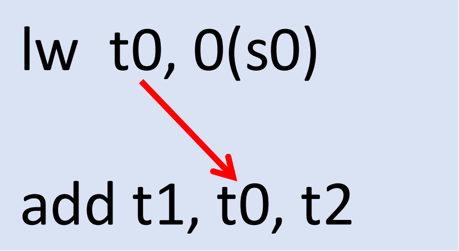
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 15-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd stage, access memory in 4th stage

From French “*hasard*”

A game of chance played with a dice.

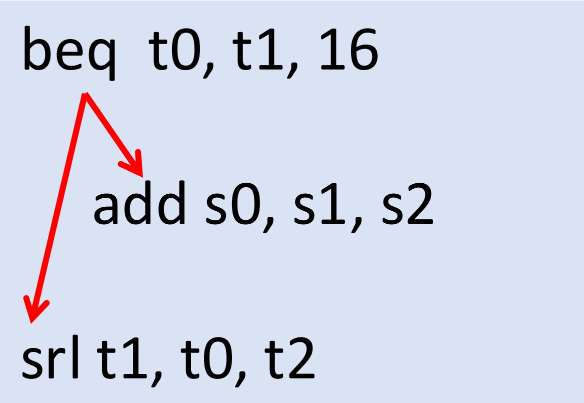
Hazards

Hazards



```
lw  t0, 0(s0)

add t1, t0, t2
```



```
beq  t0, t1, 16

  add s0, s1, s2

srl t1, t0, t2
```

Situations that prevent starting the next instruction in the next cycle

Structural hazards

- A resource that the instruction needs is busy

Data hazard

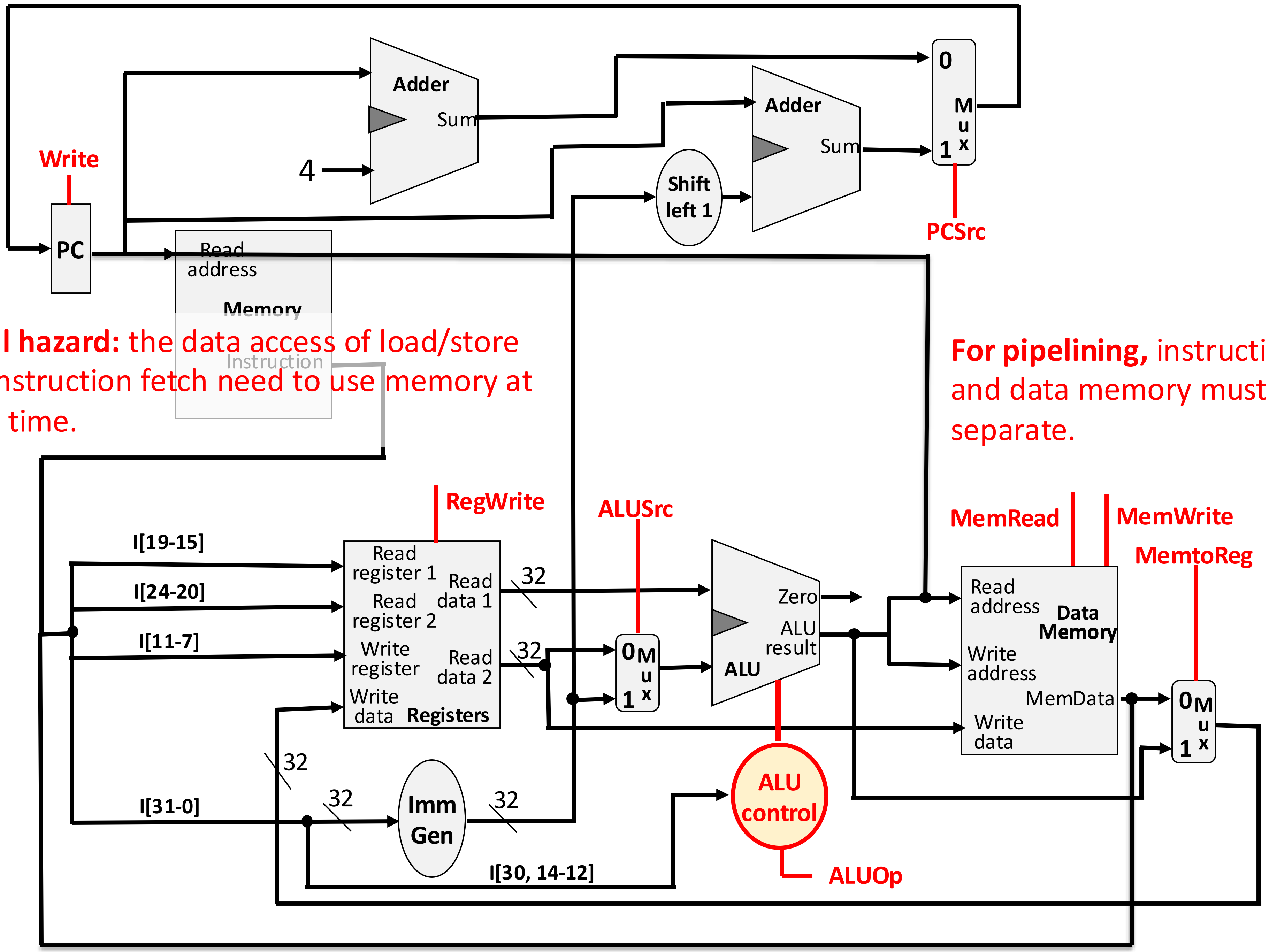
- Instruction depends on data produced by a previous instruction

Control hazard

- Control action depends on previous instruction

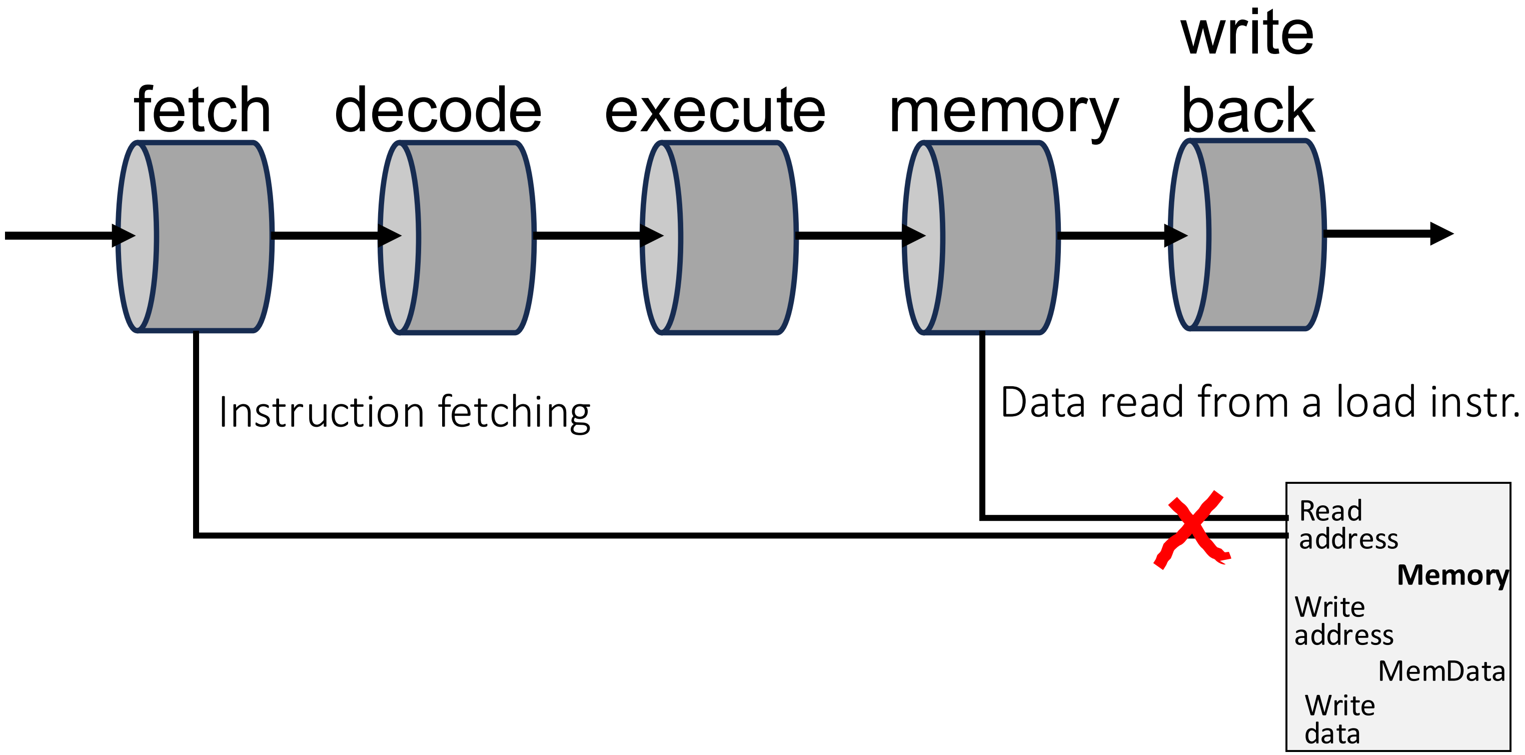
Structural Hazards

Processors do not have separate instruction and data memories…

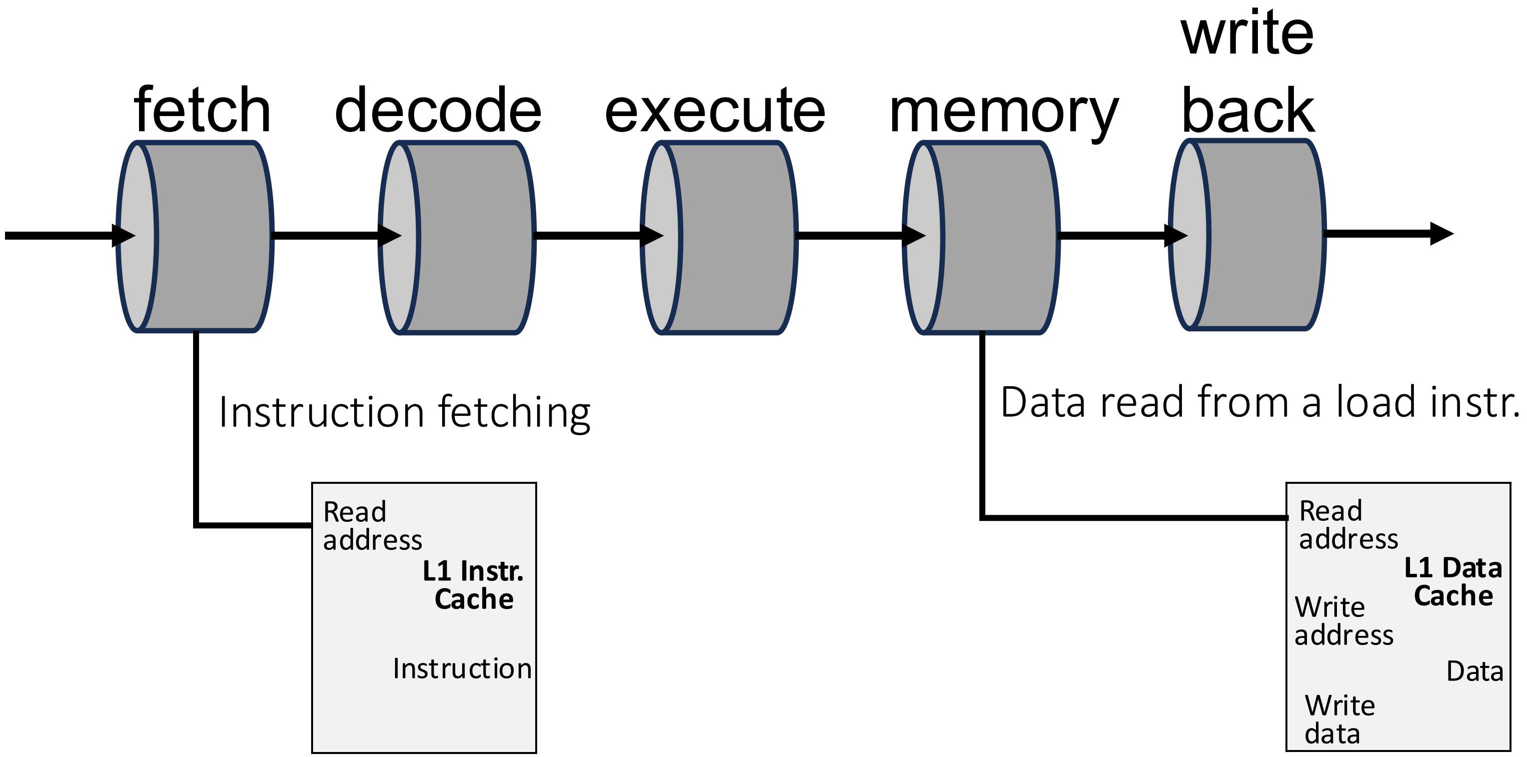


Structural hazard: the data access of load/store and the instruction fetch need to use memory at the same time.

For pipelining, instruction and data memory must be separate.



Problem: A memory cannot serve two requests on the same cycle



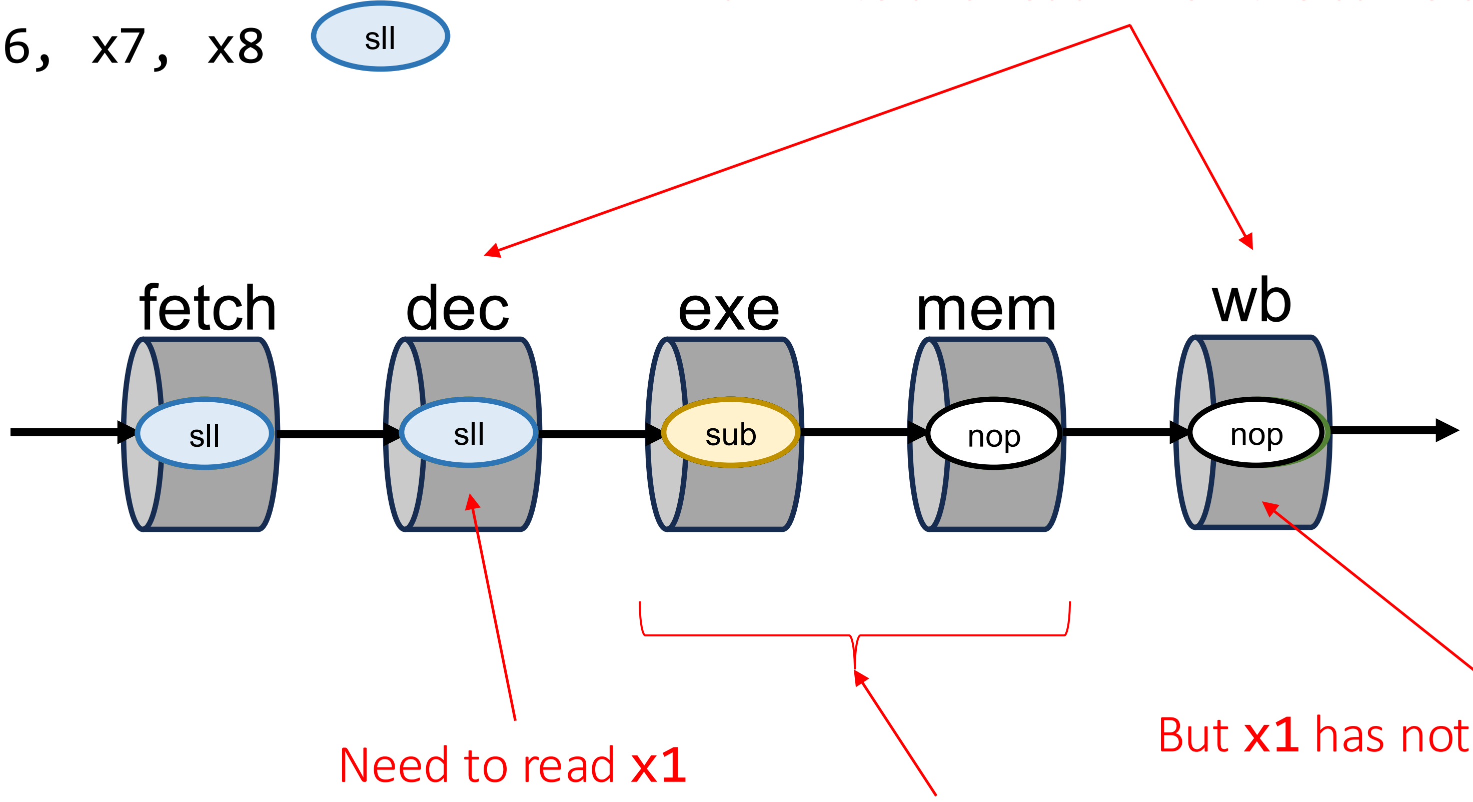
Solution: All modern processors have a separate Instruction cache (L1) and data cache (D1)

Data Hazards

An instruction depends on completion of data access by a previous instruction.

```
add    x1, x2, x3
sub    x4, x1, x5
sll    x6, x7, x8
```

Can write and read **x1** on the same cycle



Need to read **x1**

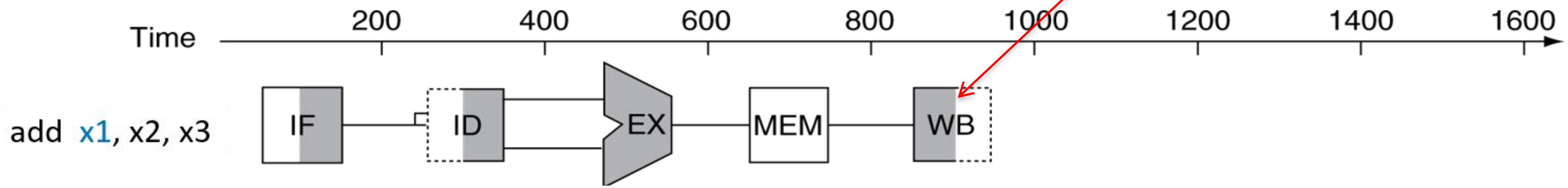
But **x1** has not been written yet

A two-cycle “bubble” in the pipeline

Data Hazards

An instruction depends on completion of data access by a previous instruction.

```
add      x1, x2, x3
sub      x4, x1, x5
```



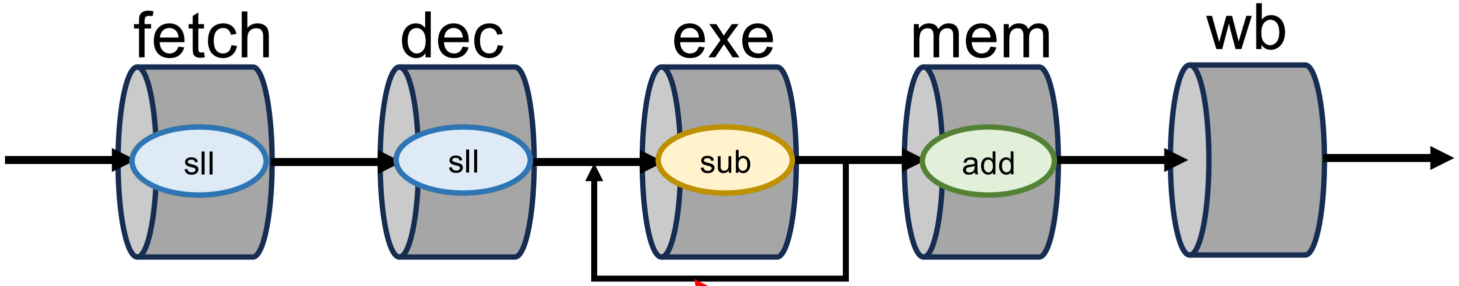
Data is available in x1

Forwarding (aka Bypassing)

An instruction depends on completion of data access by a previous instruction.

```
add    x1, x2, x3
sub    x4, x1, x5
sll   x6, x7, x8
```

No ”bubbles”

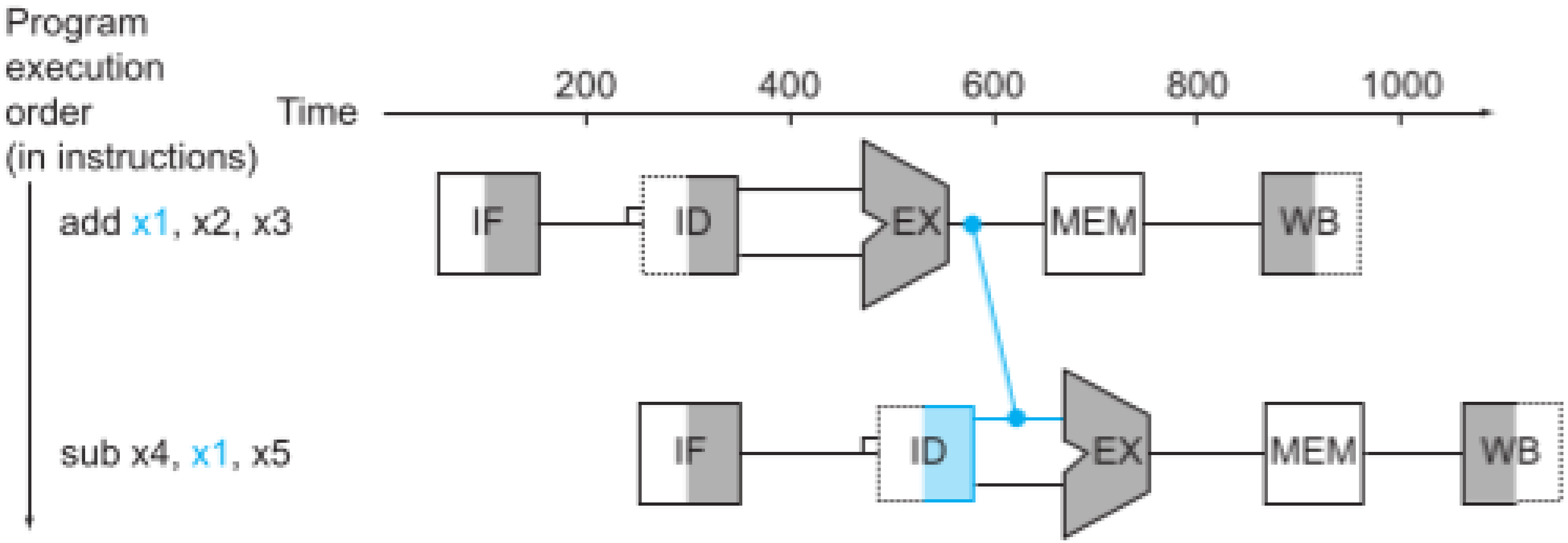


Use **x1** when it is computed

Forwarding (aka Bypassing)

Use result when it is computed

- Do not wait for it to be stored in a register
- Requires extra connections in the datapath

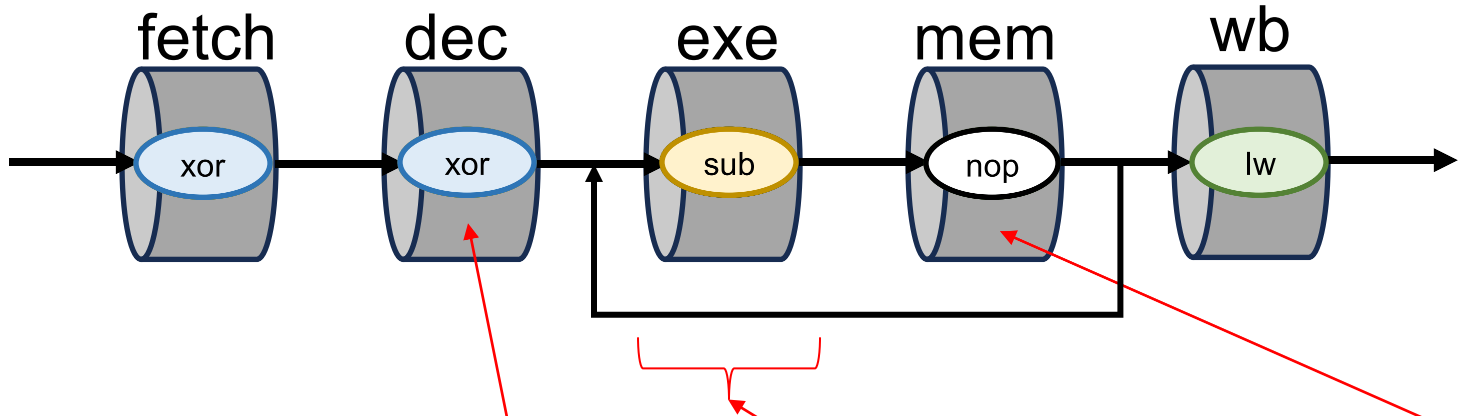


Load-use data hazard

Data Hazards

A use depends on a load

```
lw     x1, 0(x2)
sub    x4, x1, x5
xor    x6, x7, x8
```



Need to read **x1**

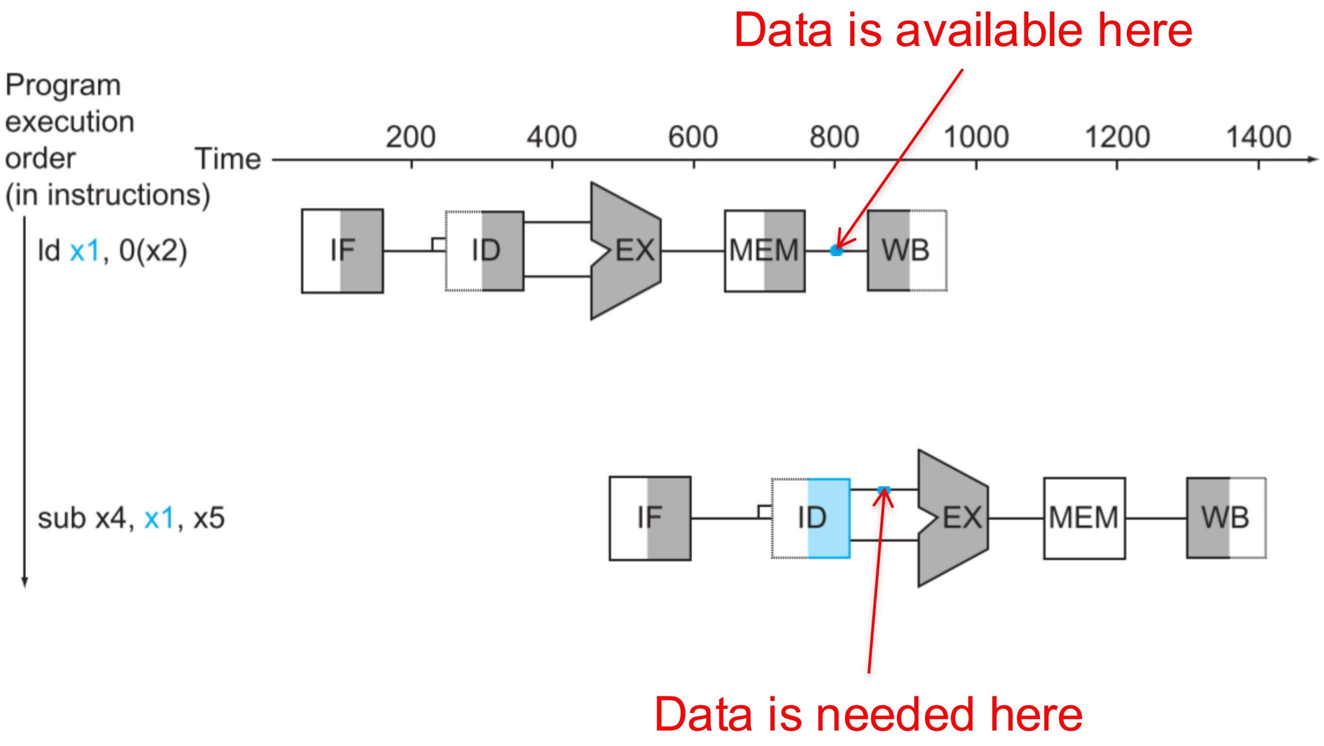
But **x1** will not be known until **lw** completes mem

A one-cycle “bubble” in the pipeline

Load-Use Data Hazard

Cannot avoid stall by forwarding

- Value is not available when needed
- Cannot send value backward in time!



Example of Code Scheduling to Avoid Stalls

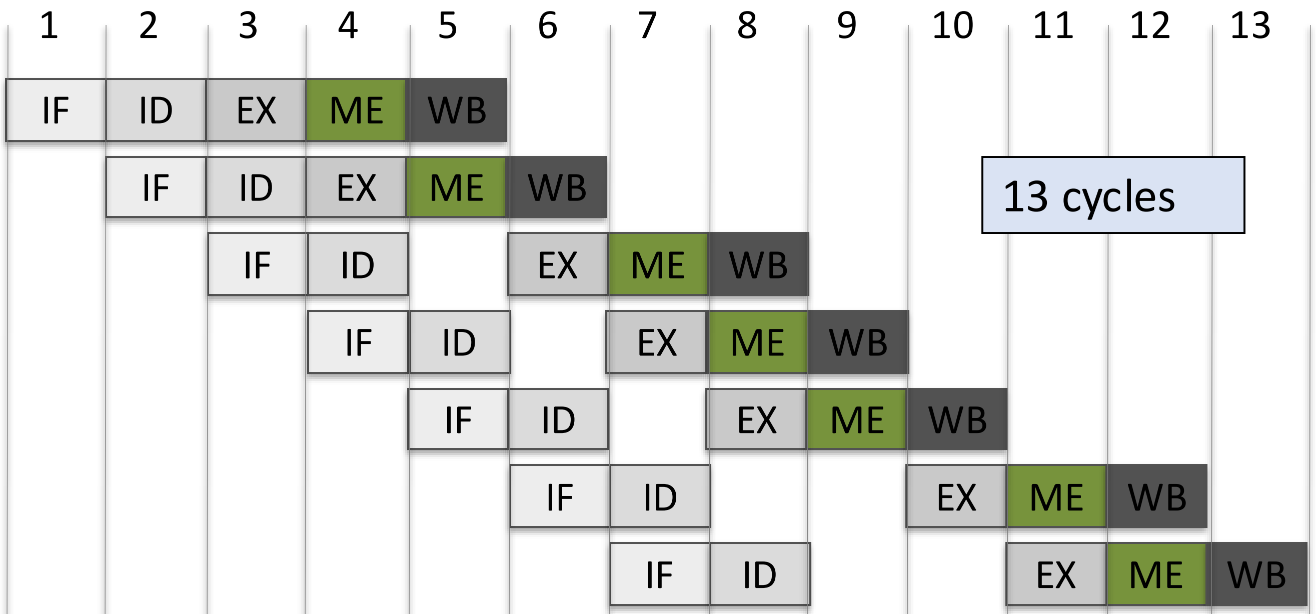
Generate assembly for the following C statements:

A = B + E;

C = B + F;

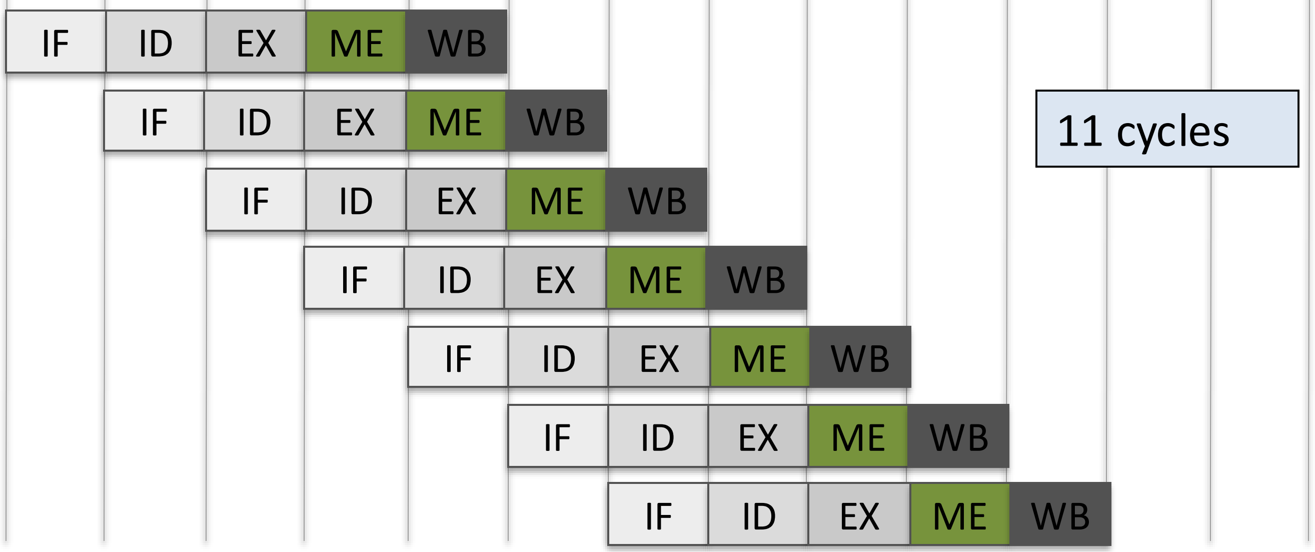
1. load B
2. load E
3. compute B+E
4. store A

1. load F
2. compute B+F
3. store C



	1	2	3	4	5	6	7	8	9	10	11	12	13
`lw t1, 0(t0)`	IF	ID	EX	ME	WB								
`lw t2, 4(t0)`		IF	ID	EX	ME	WB							
`add t3, t1, t2`			IF	ID		EX	ME	WB					
`sw t3, 12(t0)`				IF	ID		EX	ME	WB				
`lw t4, 8(t0)`					IF	ID		EX	ME	WB			
`add t5, t1, t4`						IF	ID			EX	ME	WB	
`sw t5, 16(t0)`							IF	ID			EX	ME	WB

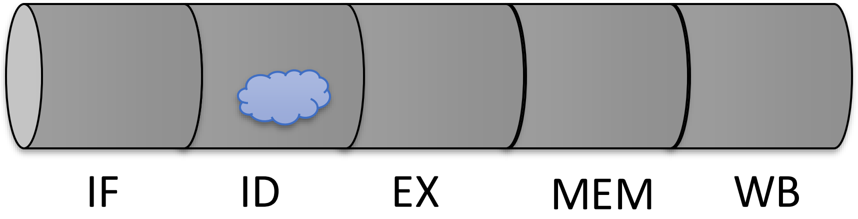
13 cycles



	1	2	3	4	5	6	7	8	9	10	11
`lw t1, 0(t0)`	IF	ID	EX	ME	WB						
`lw t2, 4(t0)`		IF	ID	EX	ME	WB					
`lw t4, 8(t0)`			IF	ID	EX	ME	WB				
`add t3, t1, t2`				IF	ID	EX	ME	WB			
`sw t3, 12(t0)`					IF	ID	EX	ME	WB		
`add t5, t1, t4`						IF	ID	EX	ME	WB	
`sw t5, 16($t0)`							IF	ID	EX	ME	WB

11 cycles

```
lw      t1, 0(t0)       ●
lw      t2, 4(t0)       ●
add     t3, t1, t2      ●
sw      t3, 12(t0)      ●
lw      t4, 8(t0)       ●
add     t5, t1, t4      ●
sw      t5, 16(t0)      ●
```



The BIG Picture

- Stalls reduce performance
  - But are needed to produce correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure