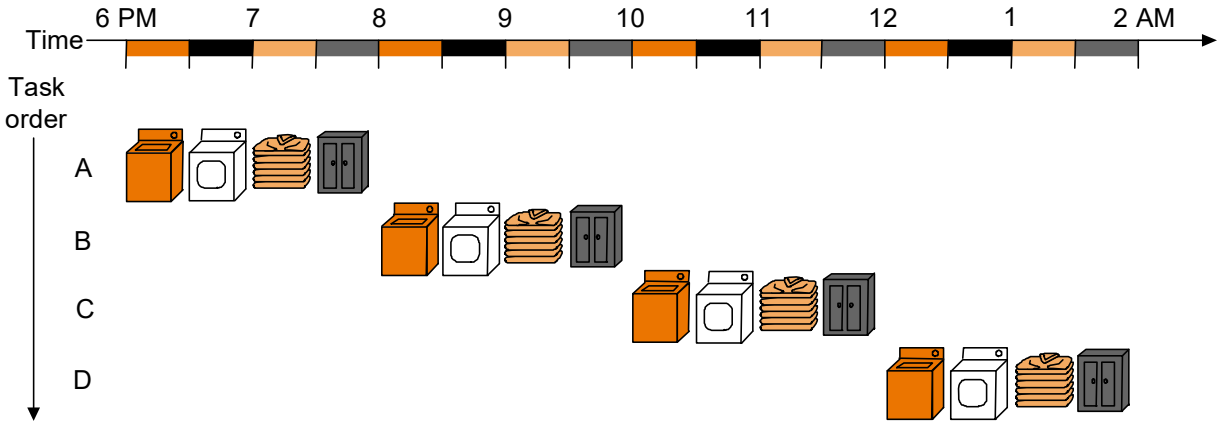
18-447 Lecture 7: Pipelined Implementation

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - getting started on pipelined implementations
- Notices
  - Lab 1, Part B, due Friday midnight
  - HW1, past due
  - Handout #5: HW 2
  - Handout #6: HW 1 solutions
- Readings
  - P&H Ch 4

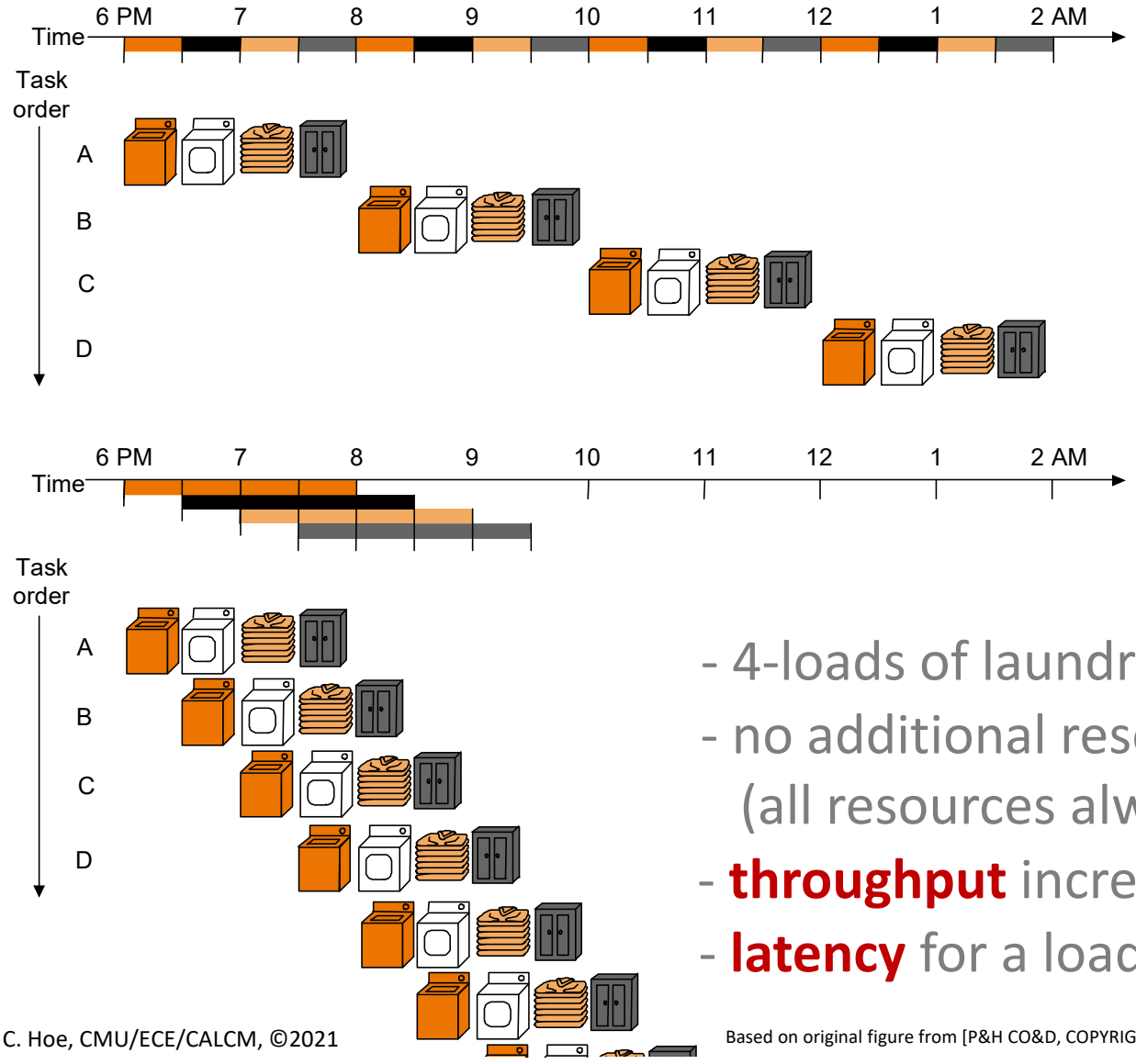
Doing laundry more quickly: in theory



1. "place one dirty load of clothes in **washer**"
2. "when washer is finished, place wet clothes in **dryer**"
3. "when dryer is finished, **you** fold dried clothes"
4. "when folding is finished, ask **friend** to put clothes away"

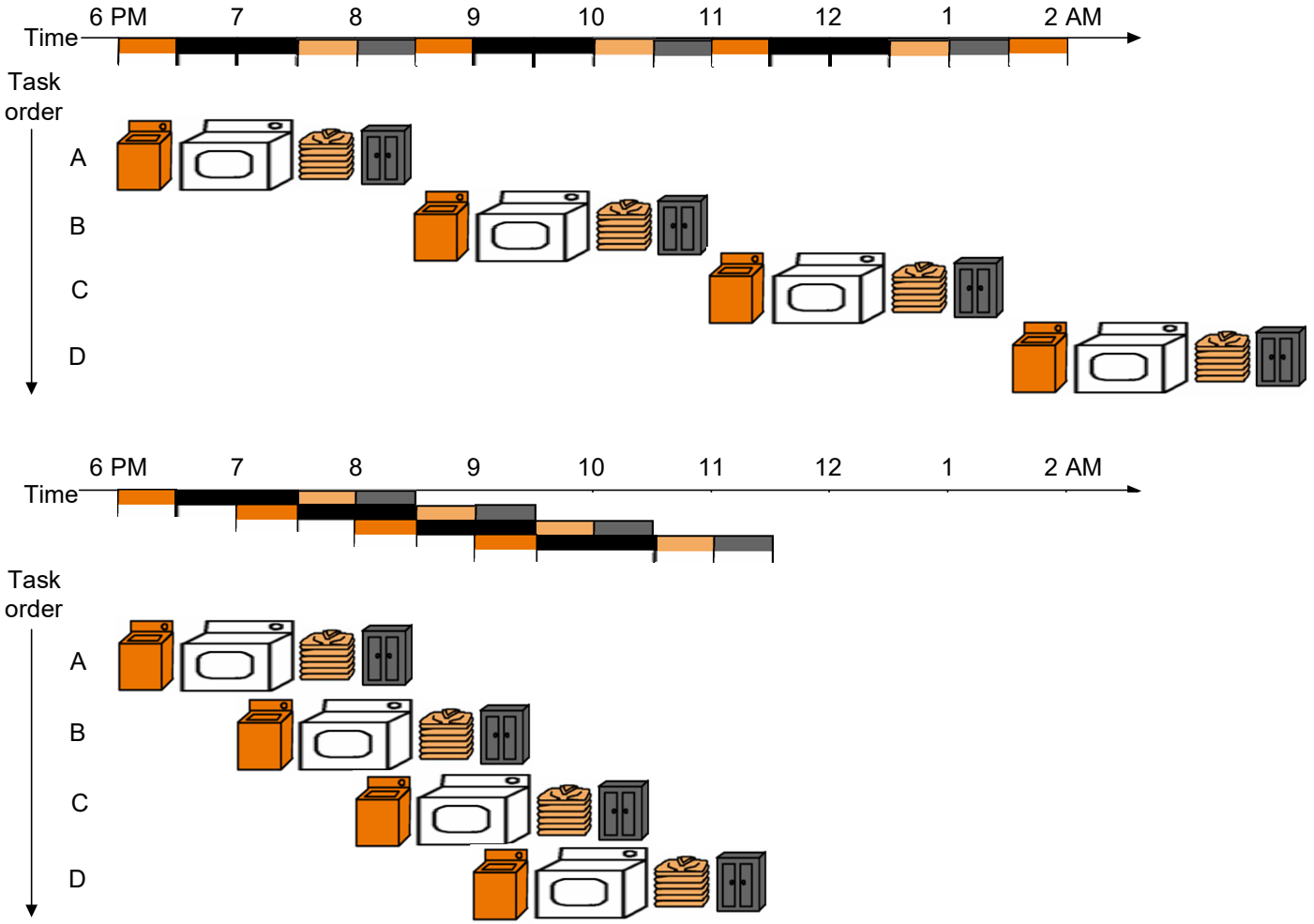
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share **resources**

Doing laundry more quickly: in theory



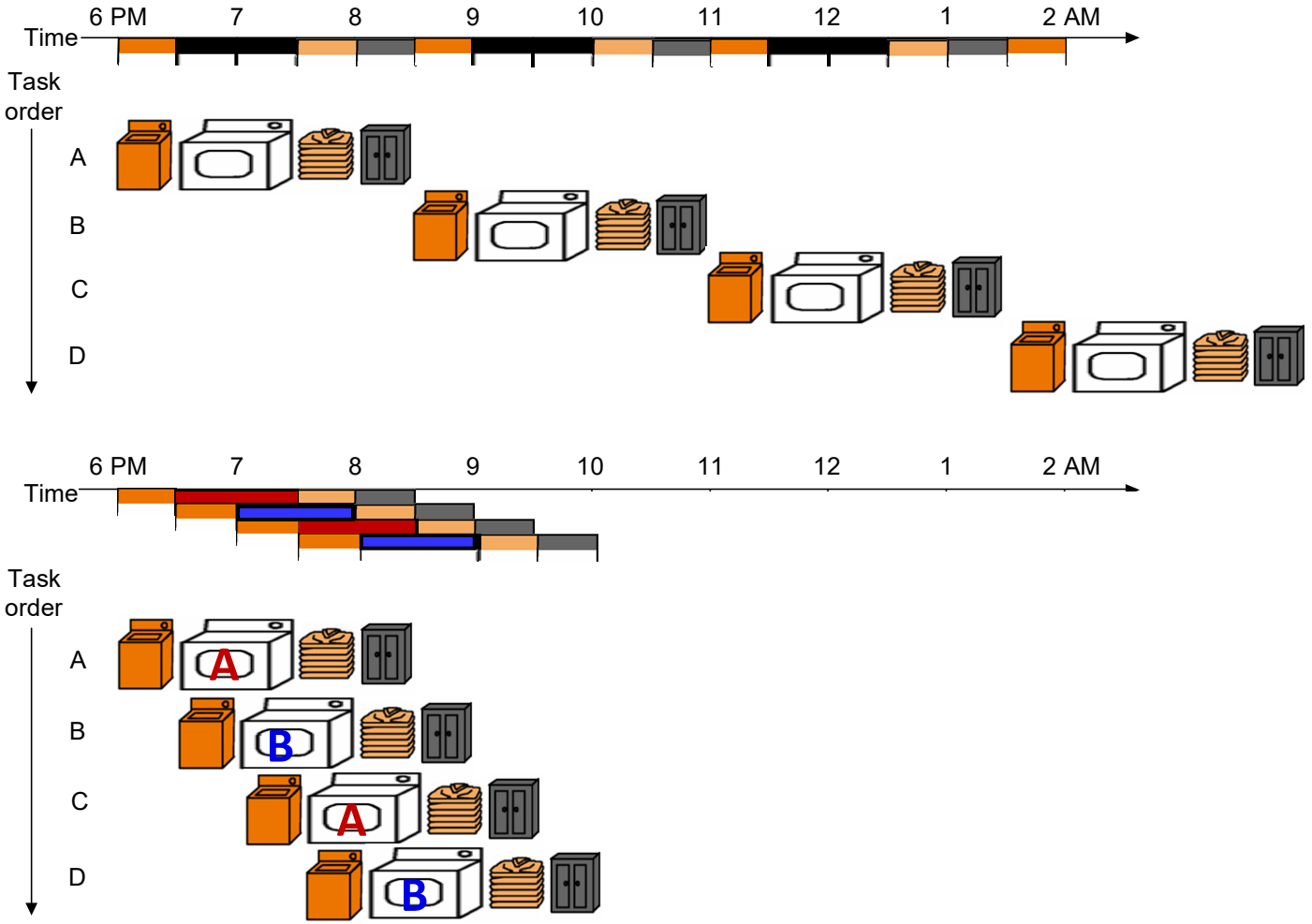
- 4-loads of laundry in parallel
- no additional resources
  (all resources always busy!)
- **throughput** increased by 4
- **latency** for a load is the same

Doing laundry more quickly: in practice



the slowest step decides **throughput**

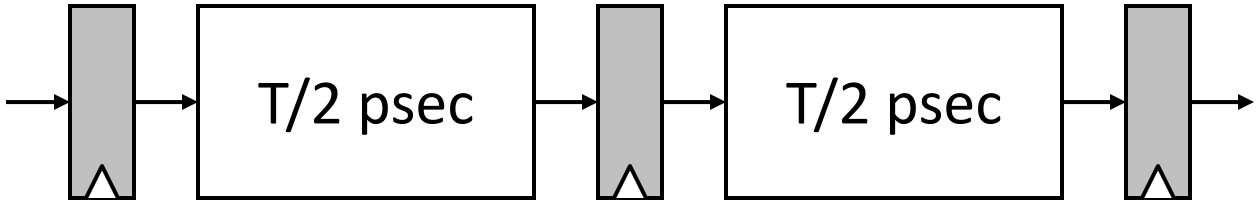
Doing laundry more quickly: in practice



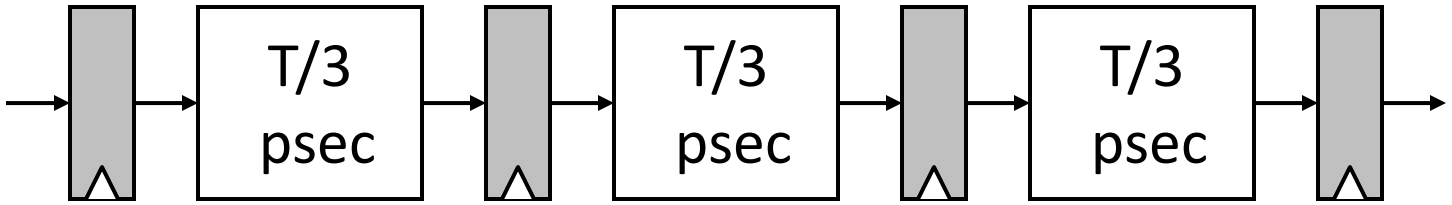
Throughput restored (2 loads per hour) using 2 dryers

(Ideal) HW Pipelining

thruput=~(1/T)



thruput=~(2/T)
speedup=2



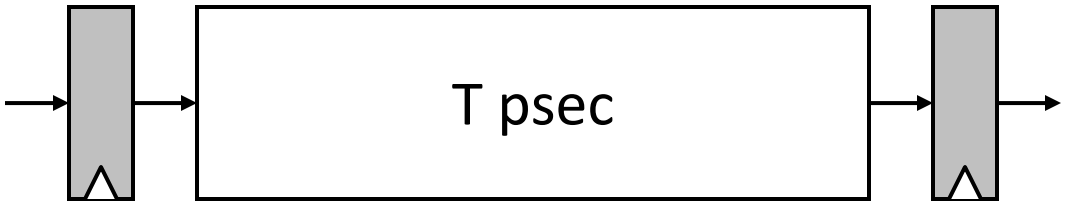
tput=~(3/T)
speedup=3

Notice: evenly divisible; no feedback wires

Performance Model

- Nonpipelined version with delay T

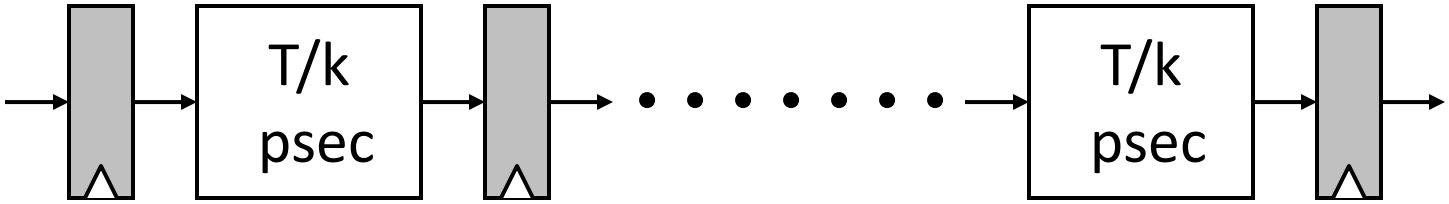
  throughput = 1/(T+S) where S = latch delay



- k-stage pipelined version

  $throughput_{k\text{-}stage} = 1 / (T/k + S)$

  $throughput_{max} = 1 / (1\ gate\ delay + S)$

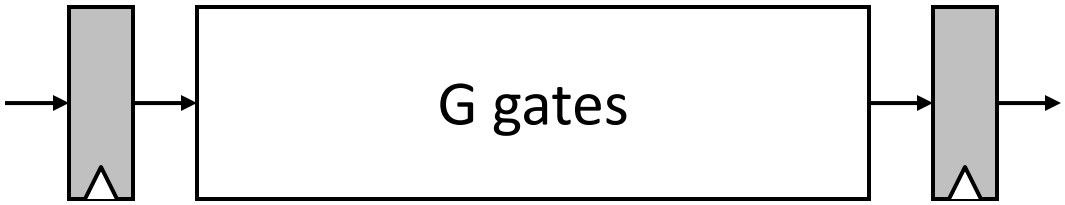


per-task latency became longer: T+kS

Cost Model

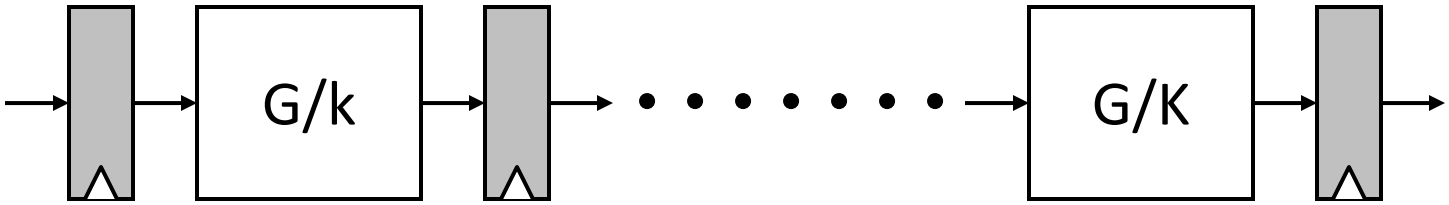
- Nonpipelined version with combinational cost G

  Cost = G+L where L = latch cost



- k-stage pipelined version

  $Cost_{k\text{-}stage} = G + Lk$



Pipeline Idealism

Motivation: Increase throughput without adding hardware cost

- Repetition of identical tasks

  same task repeated for many different inputs

- Repetition of independent tasks

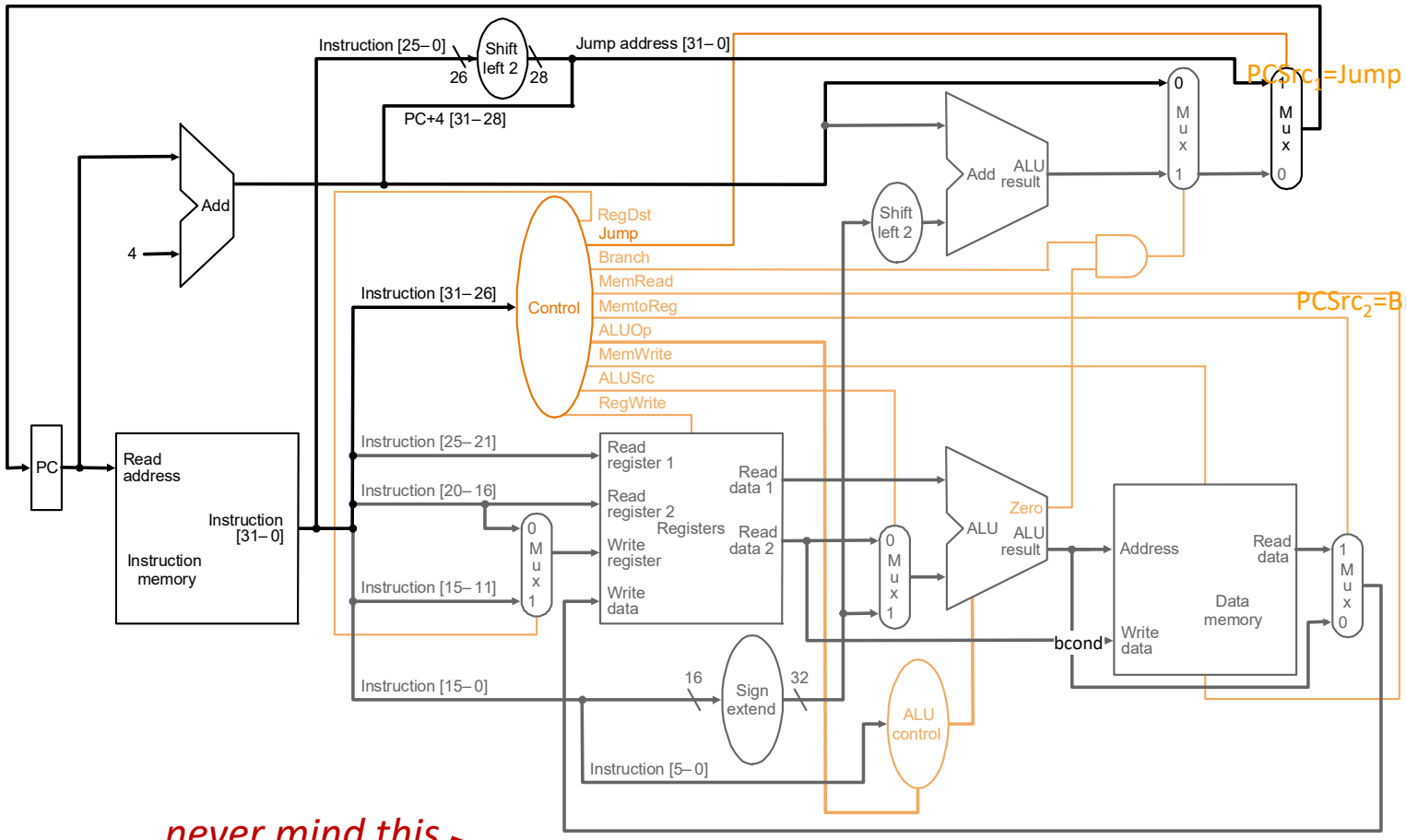
  no ordering dependencies between repeated tasks

- Uniformly partitionable suboperations

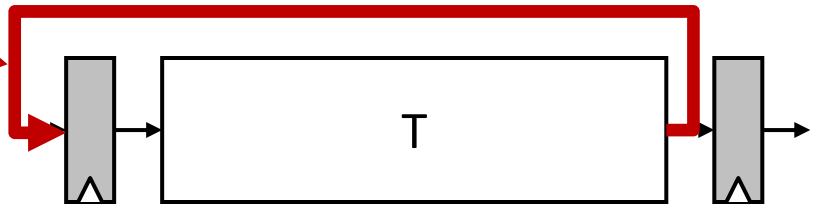
  arbitrary number and placement of boundaries

Good examples: automobile assembly line, doing laundry, but instruction execution???

Reality of Instruction Pipelining . . . .



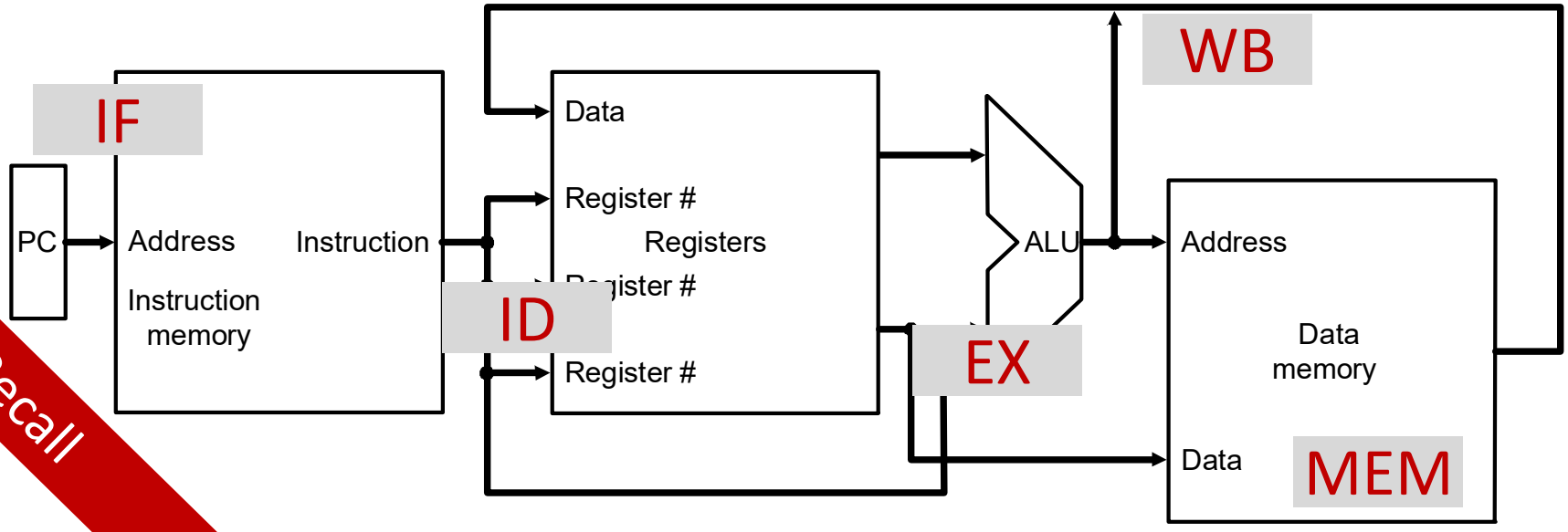
never mind this complication today



tput=~(1/T)

RISC Instruction Processing

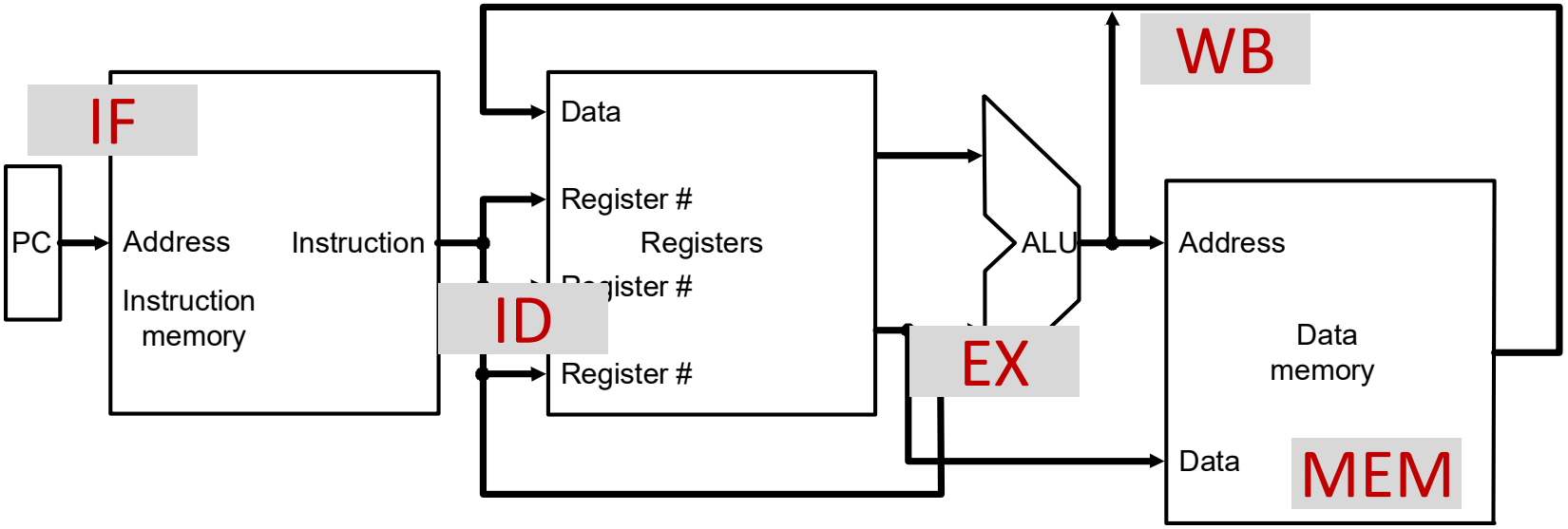
- 5 generic steps
  - instruction fetch
  - instruction decode and operand fetch
  - ALU/execute
  - memory access
  - write-back



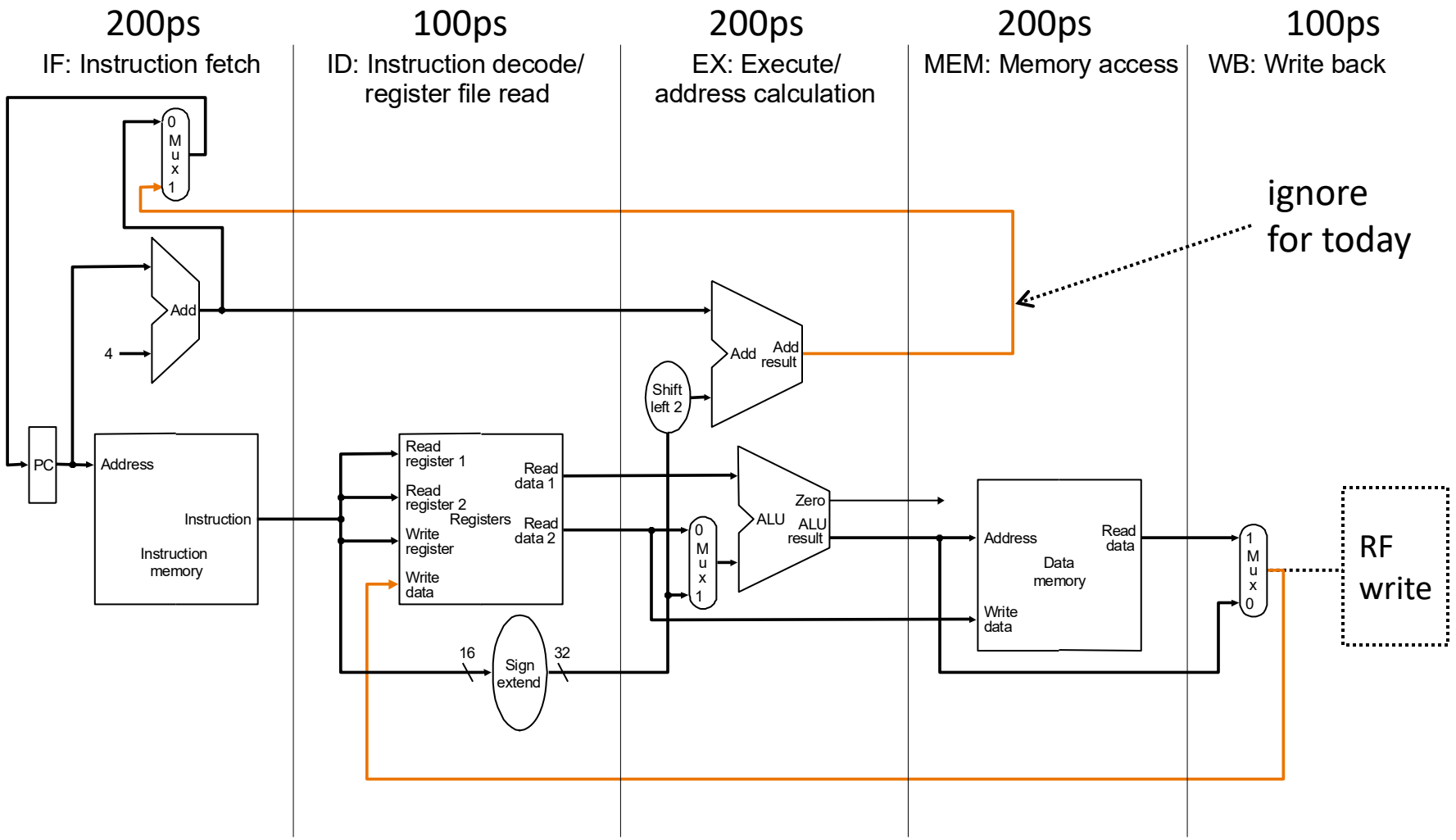
Recall

Coalescing and "External Fragmentation"

steps	IF	ID	EX	MEM	WB
R-type	√	√	√		√
I-type	√	√	√		√
LW	√	√	√	√	√
SW	√	√	√	√	
Bxx/JALR	√	√	√		--/√
JAL	√		√		√



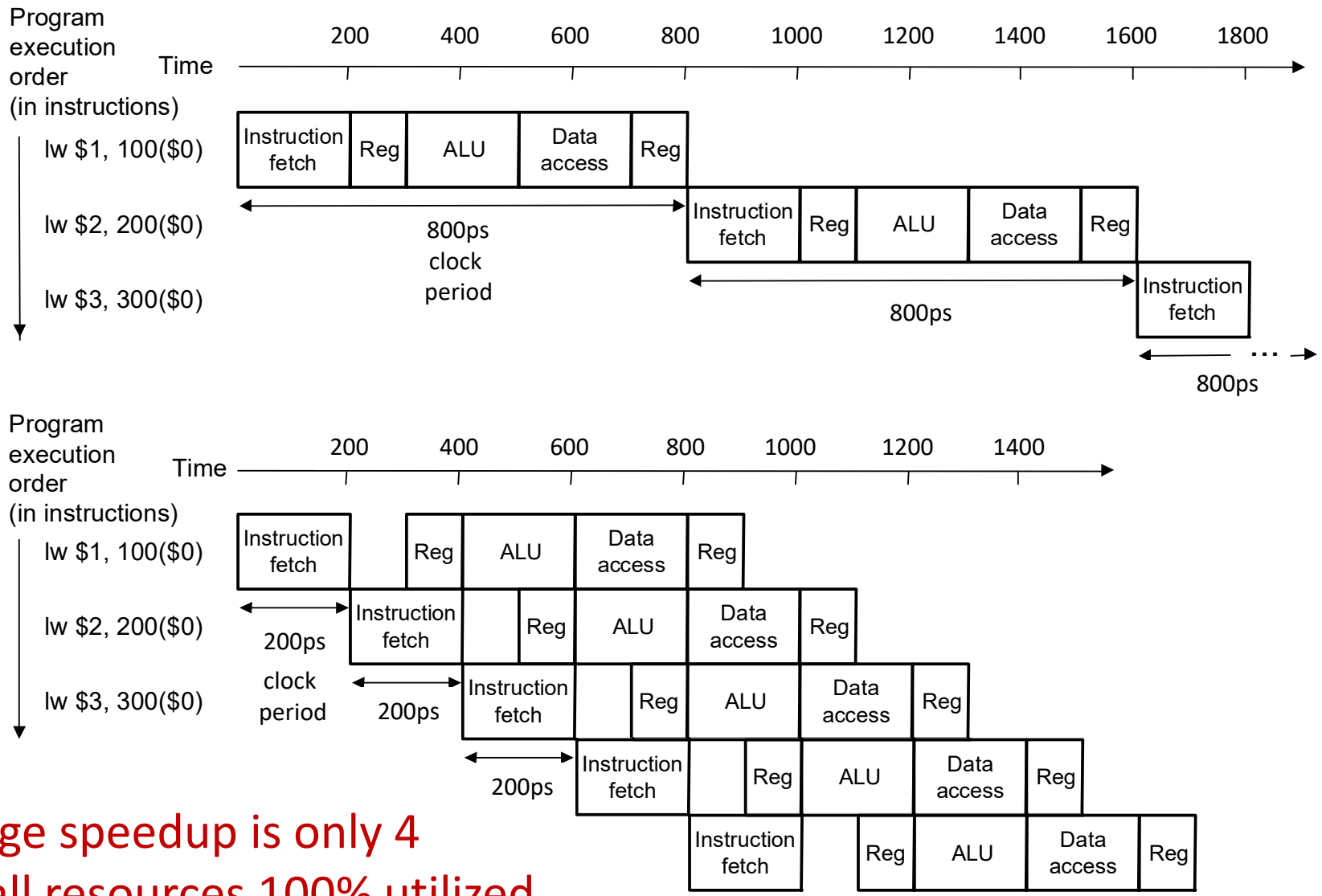
Dividing into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries

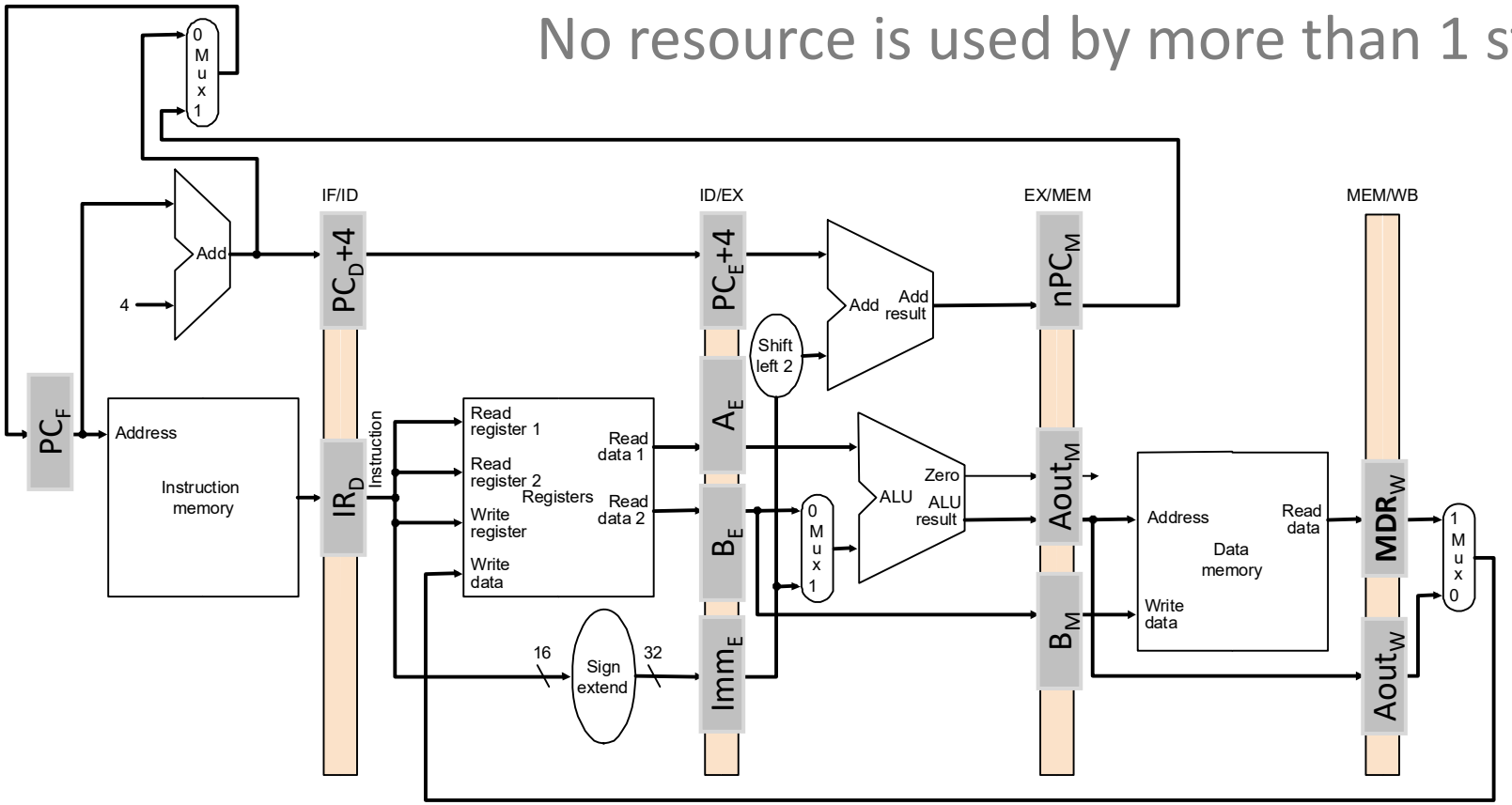
Internal and External Fragmentation



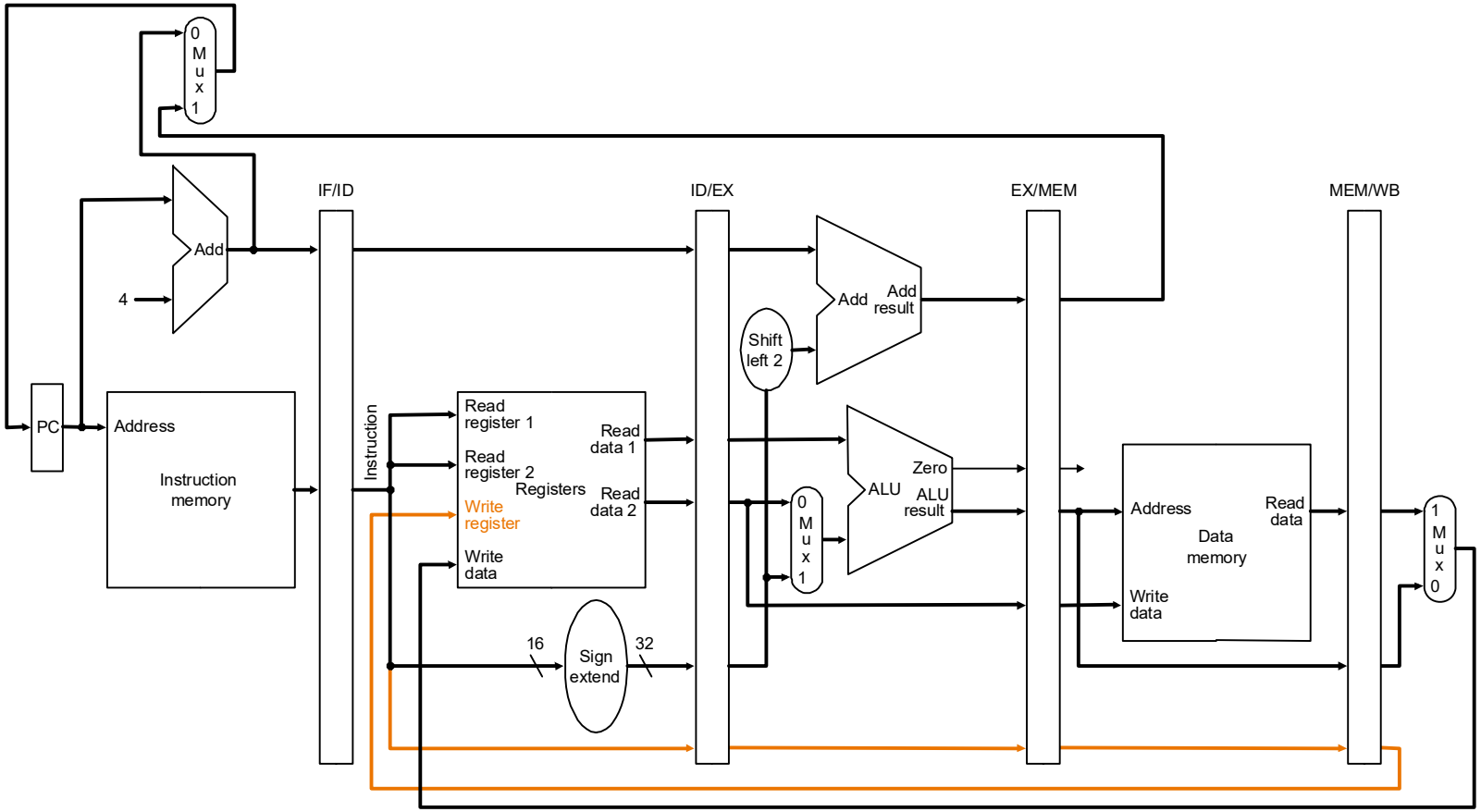
- 5-stage speedup is only 4
- Not all resources 100% utilized

Pipeline Registers

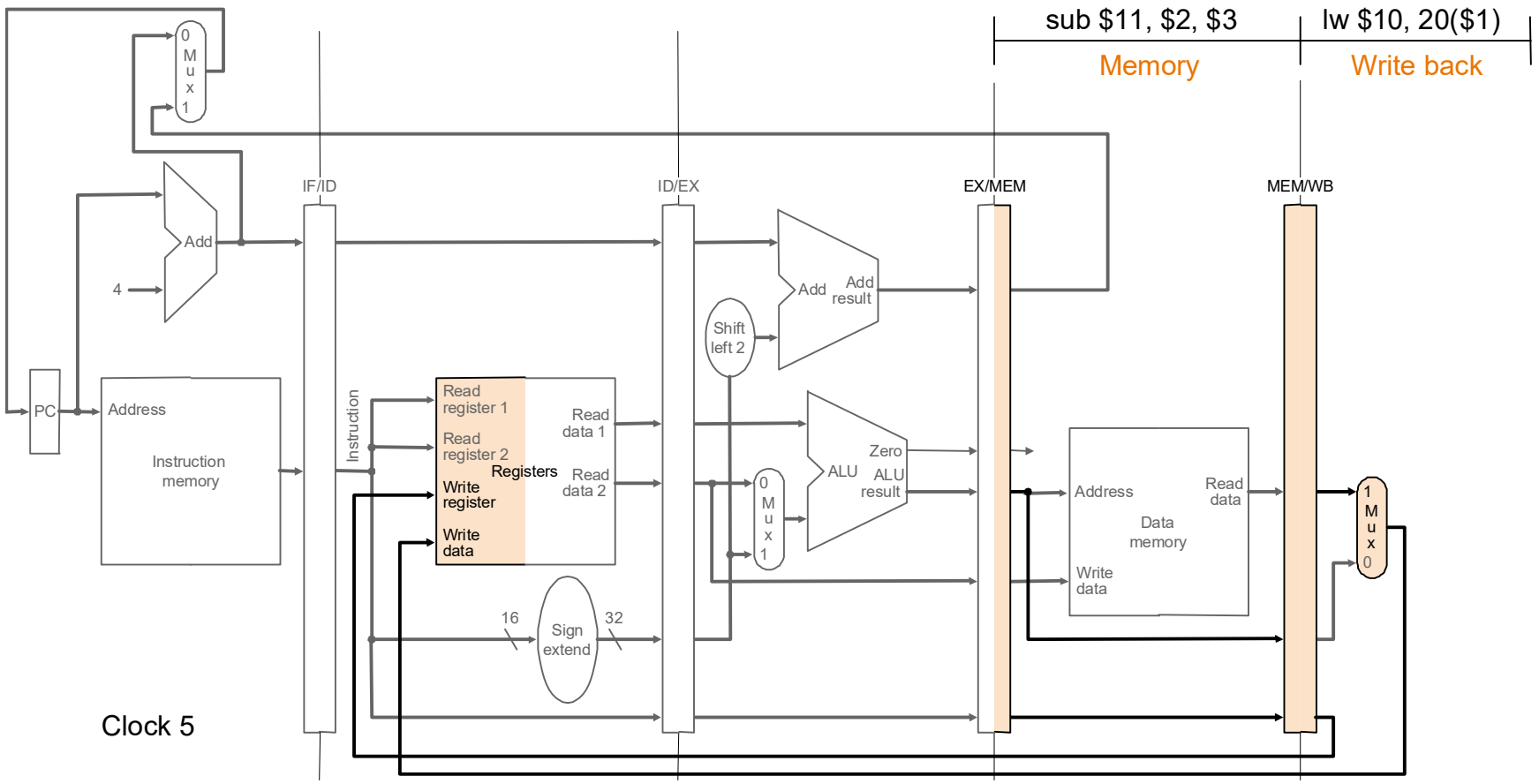
No resource is used by more than 1 stage!



Pipelined Operation



Pipelined Operation

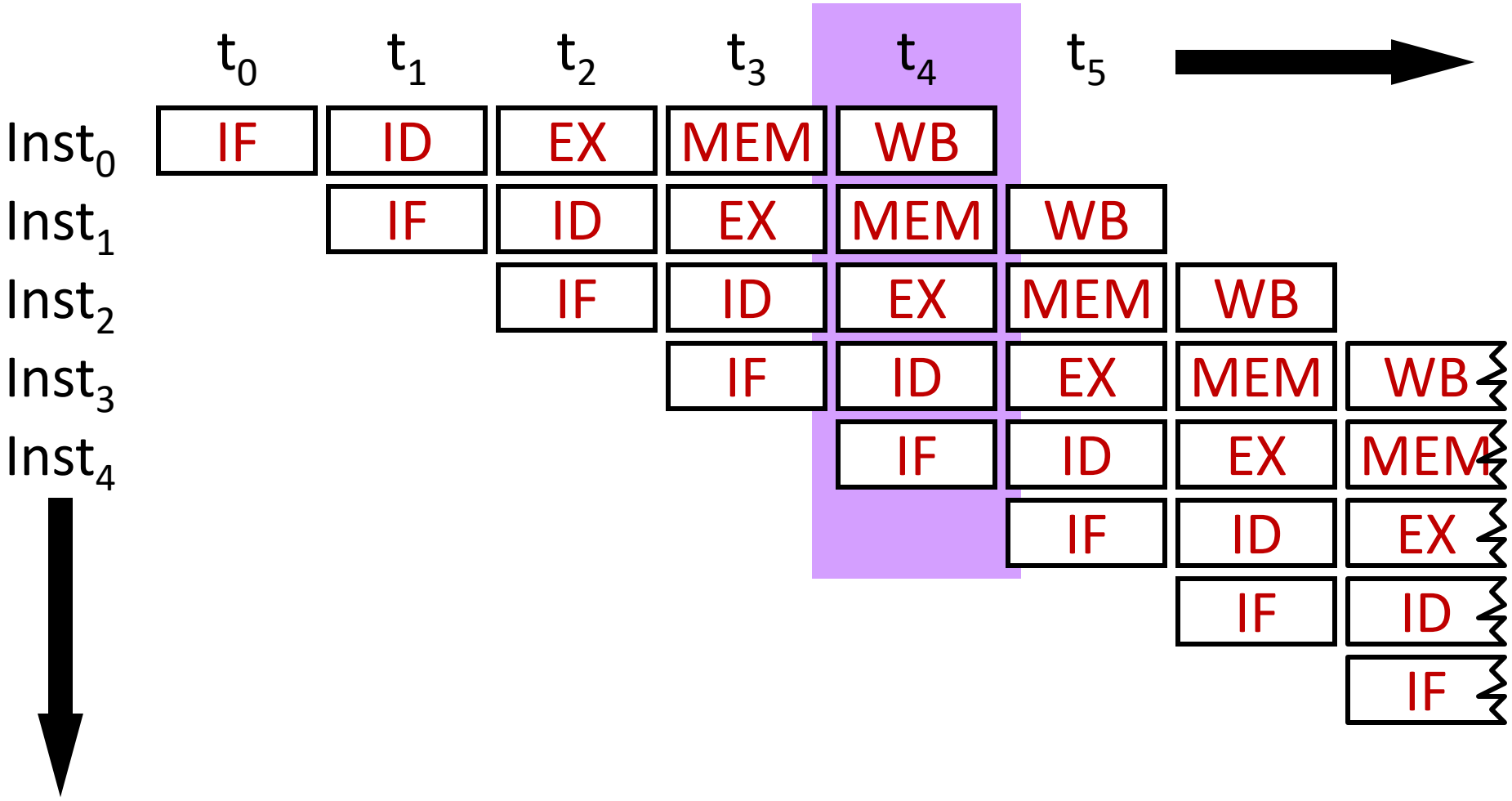


What if LW dest is **$2***?*

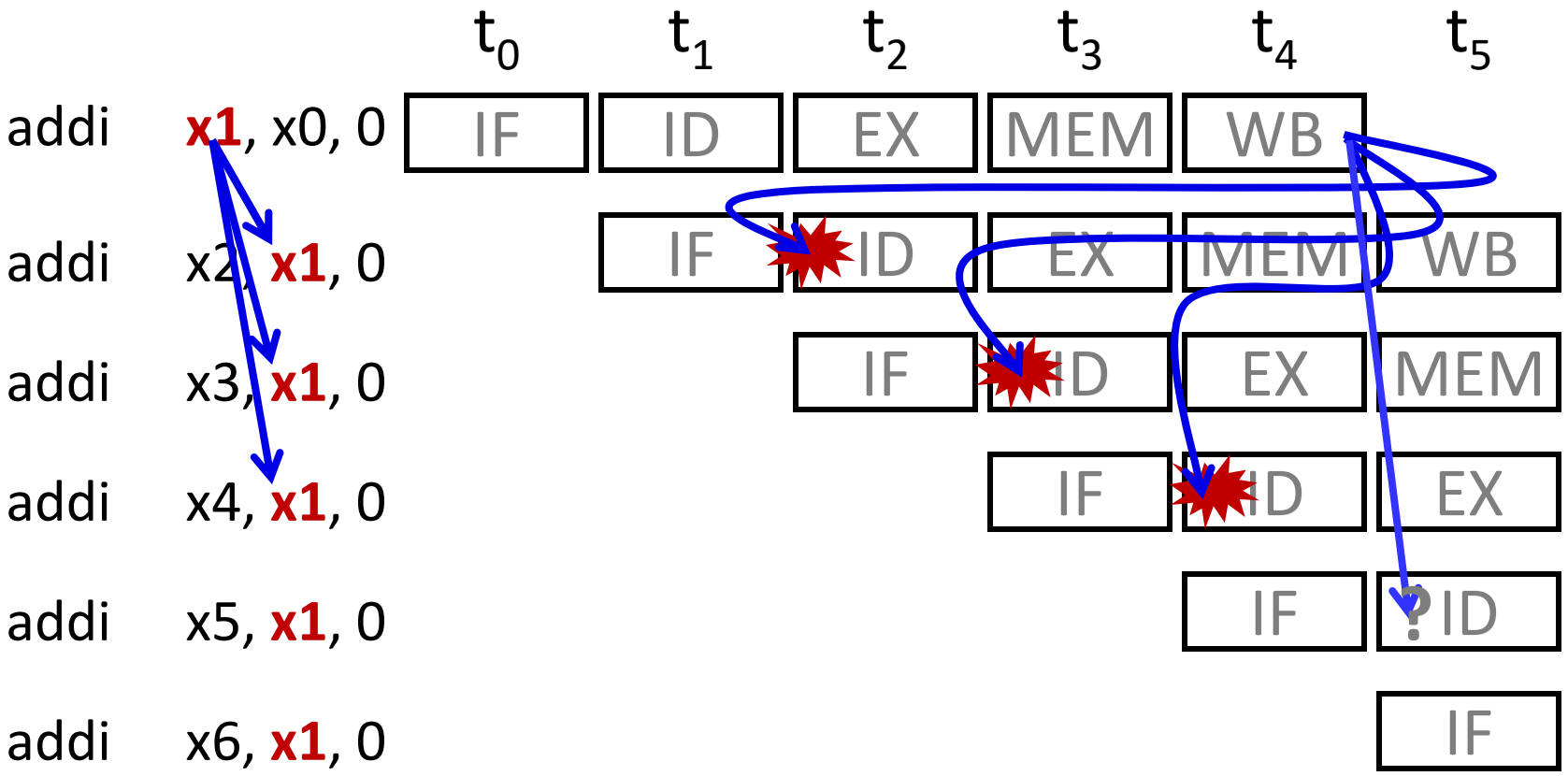
Illustrating Pipeline Operation: Resource View

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
ID		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
EX			I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM				I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB					I_0	I_1	I_2	I_3	I_4	I_5	I_6

Illustrating Pipeline Operation: Operation View



Example: Read-after-Write Hazard

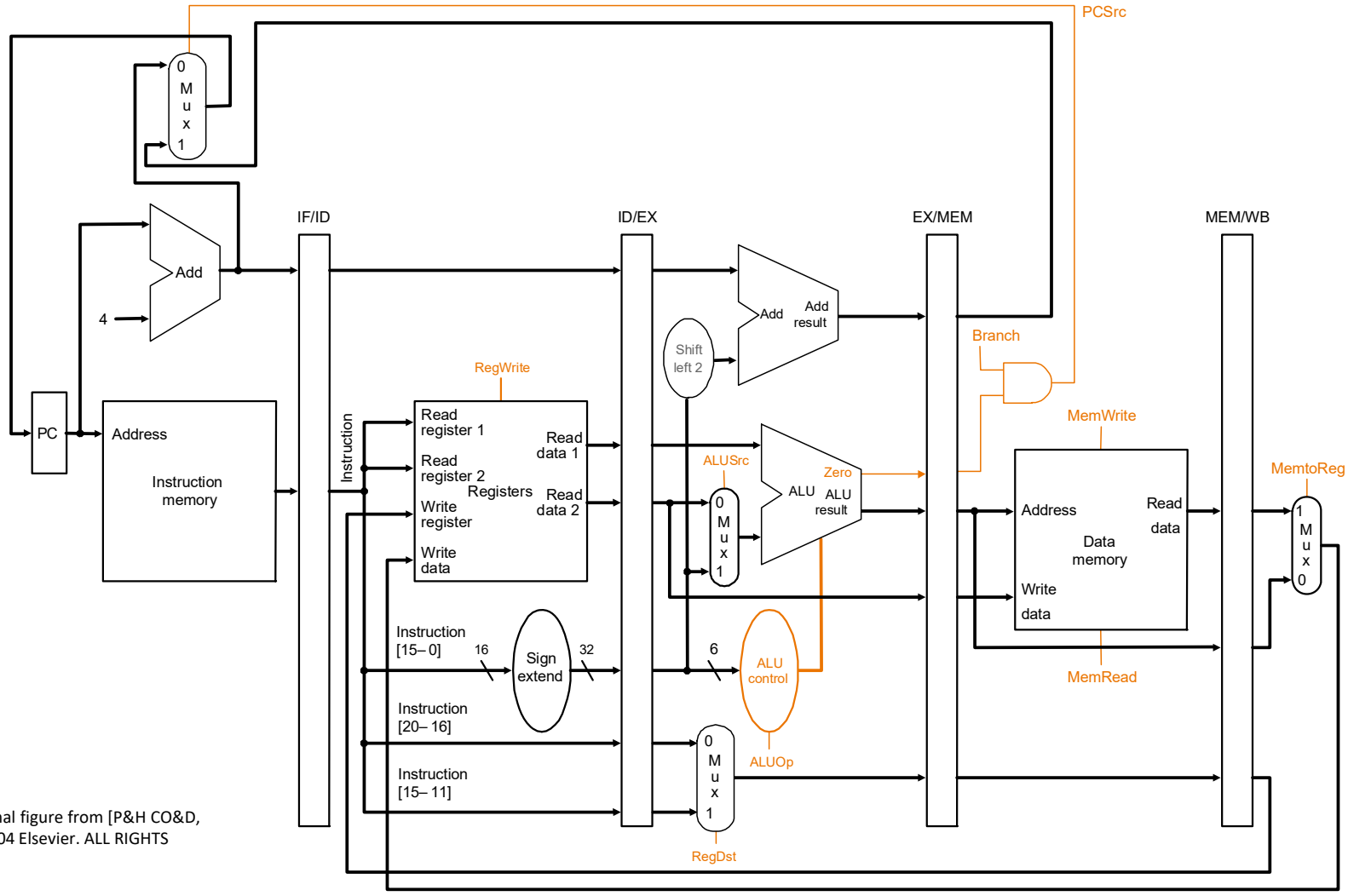


Example: Pipeline Stalls

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_4	I_4	I_4	I_5	I_6	I_7
ID		I_0	I_1	I_2	I_3	I_3	I_3	I_3	I_4	I_5	I_6
EX			I_0	I_1	I_2	∅	∅	∅	I_3	I_4	I_5
MEM				I_0	I_1	I_2	∅	∅	∅	I_3	I_4
WB					I_0	I_1	I_2	∅	∅	∅	I_3

I_2=addi x1, x0, 0 ; I_3=addi x2, x1, 0 ;

Control Points

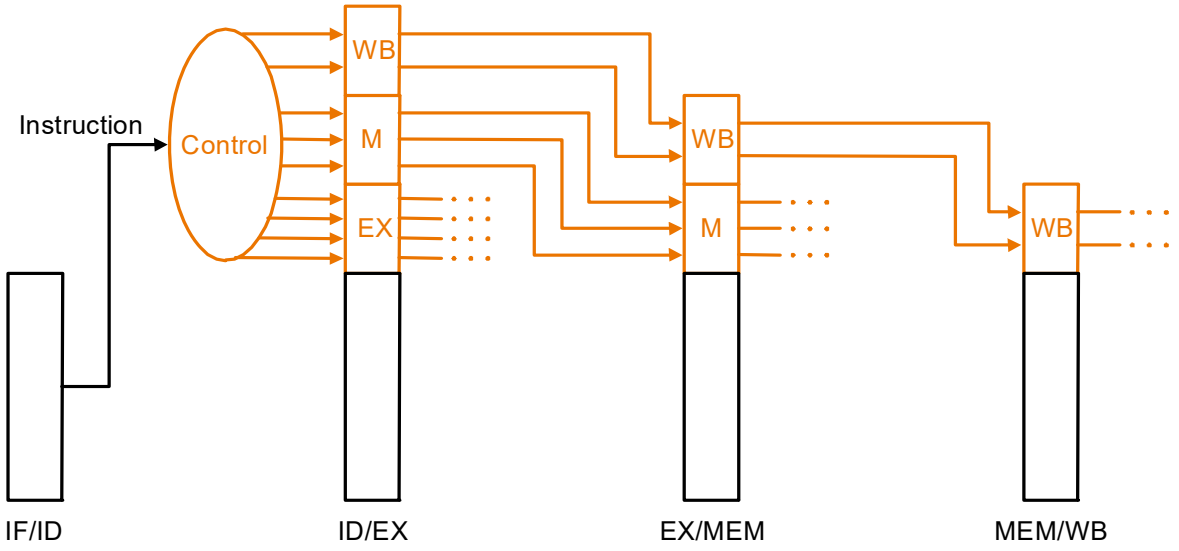


Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

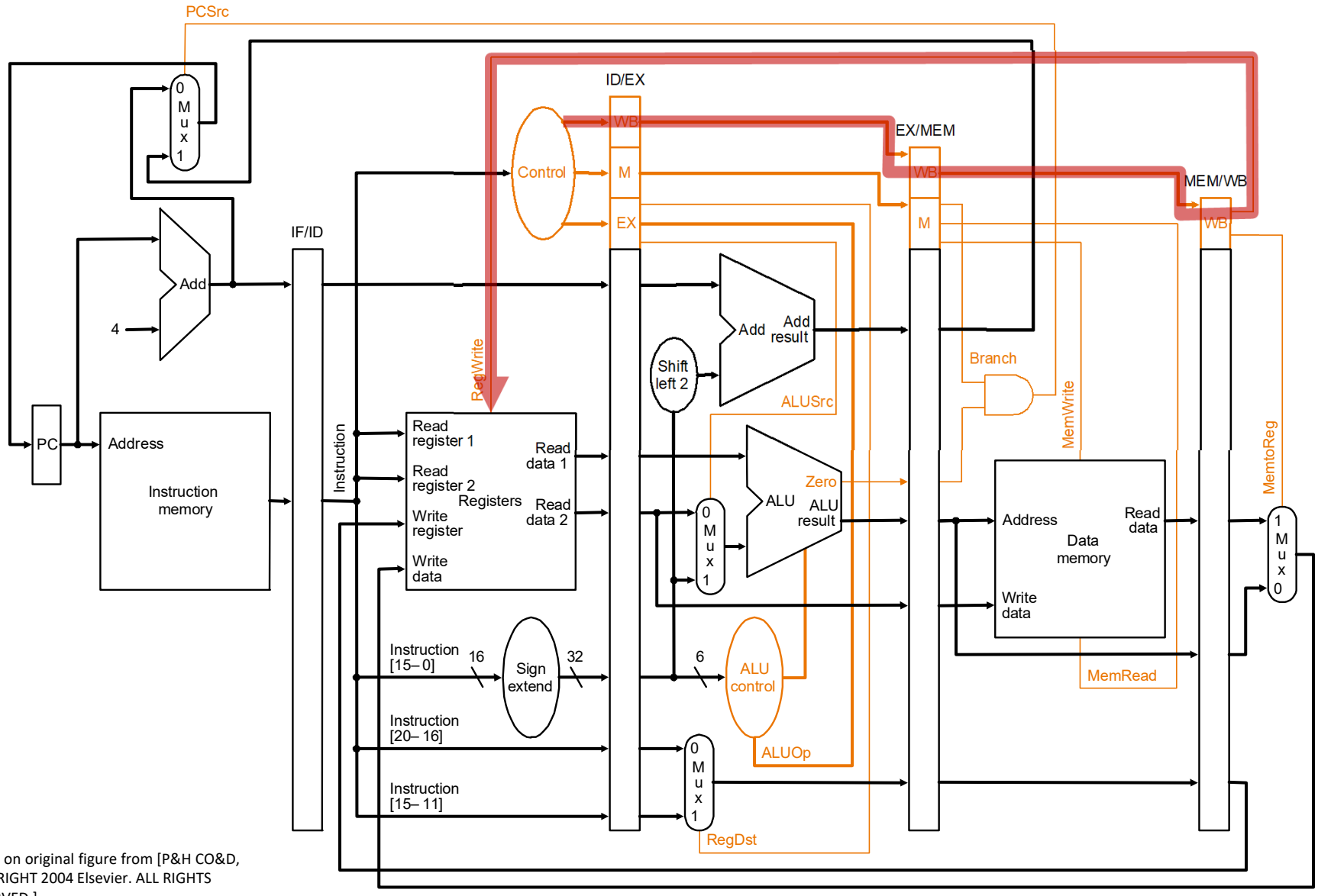
Identical set of control points as the single-cycle datapath!!

Sequential Control: Special Case

- For a given instruction
  - same control settings as single-cycle, but
  - control signals required at different cycles, depending on stage
  - decode once using the same logic as single-cycle and buffer control signals until consumed



Pipelined Control



Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

This is all there is to it (without hazards)!!

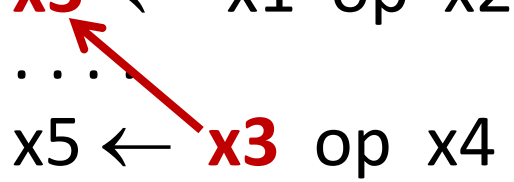
Instruction Pipeline Reality

- Not identical tasks
  - coalescing instruction types into one "multi-function" pipe
  - external fragmentation (some idle stages)
- Not uniform suboperations
  - group or sub-divide steps into stages to minimize variance
  - internal fragmentation (some too-fast stages)
- Not independent tasks
  - dependency detection and resolution
  - next lecture(s)

Even more messy if not RISC

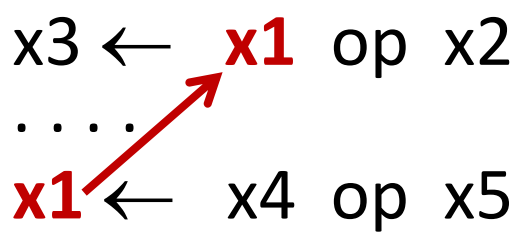
Data Dependence

Data dependence



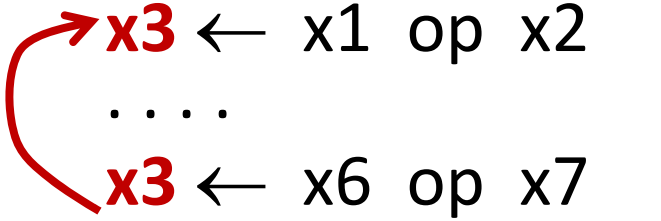
Read-after-Write (RAW)

Anti-dependence



Write-after-Read (WAR)

Output-dependence



Write-after-Write (WAW)

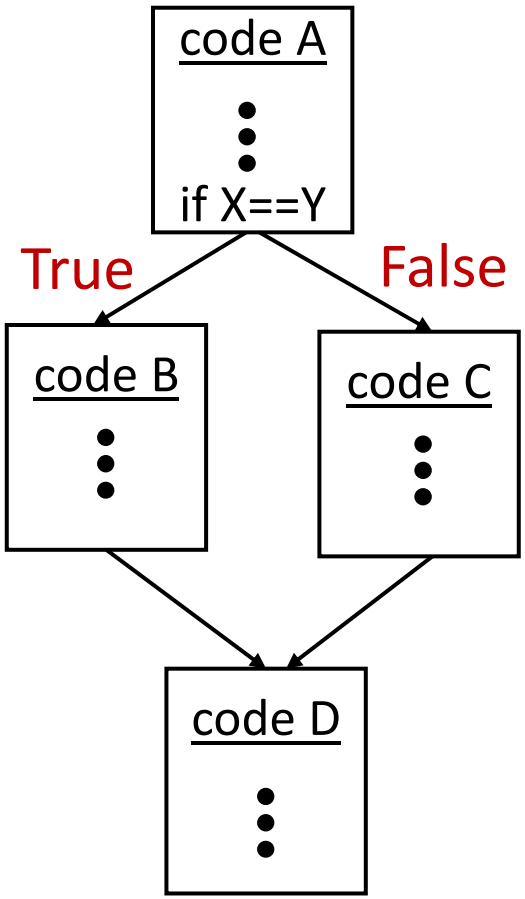
Don't forget memory instructions

Control Dependence

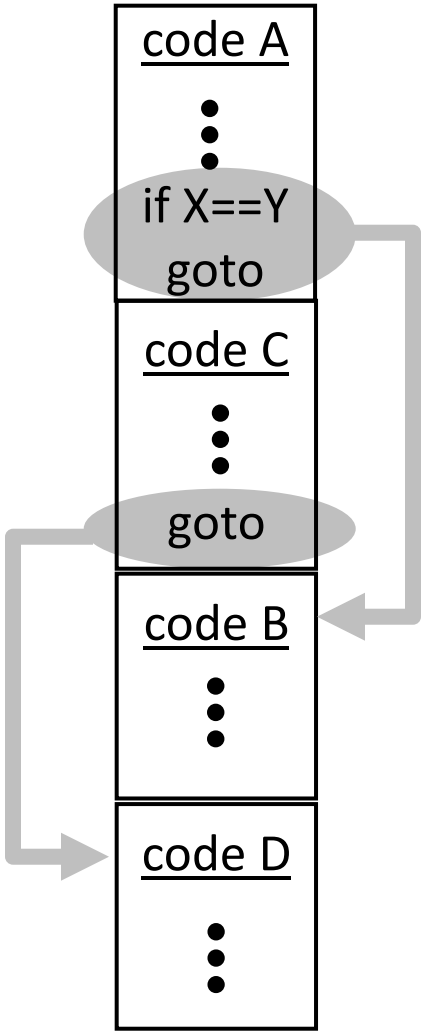
- C-Code

```
{ code A }
if X==Y then
    { code B }
else
    { code C }
{ code D }
```

Control Flow Graph



Assembly Code (linearized)



Does B or C come after A?