Lecture 7:
Pipelined CPU Implementation

Jangwoo Kim (Seoul National University)
jangwoo@snu.ac.kr

Announcement

- Homework #2 posted
  - **Due: 4/12 (Thursday)**

- Mid-term Exam
  - **Mid-term date: 6:30PM, 4/20**

What we have learned so far (1/2)

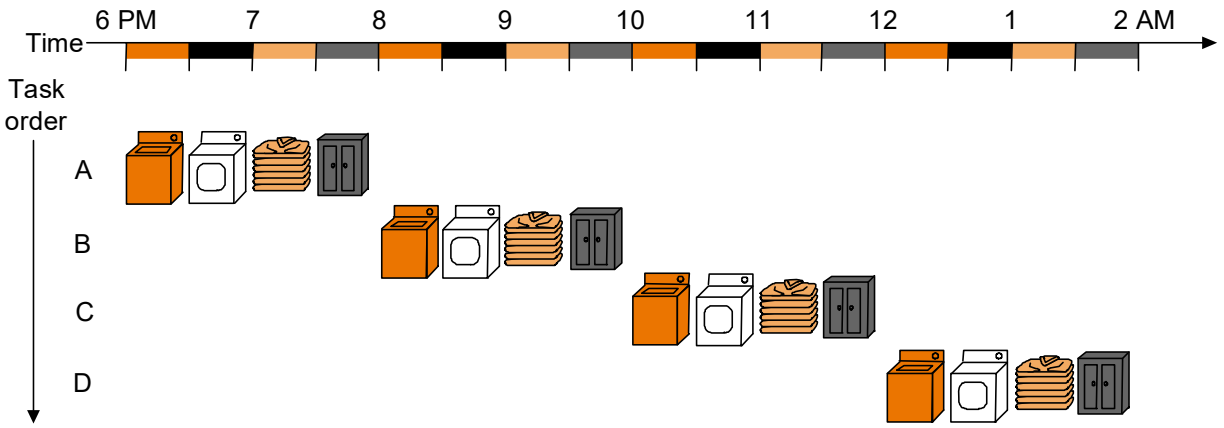
- Architecture (and Microarchitecture)
  - Instruction Set Architecture (ISA)
    - RISC vs CISC
- Types of Architecture
  - Von Newman Architecture
  - Load-Store architecture
- Architectural State (or Programmer Visible State)
  - PC, Registers, Virtual memory, …
- Several Laws
  - Moore's Law
  - Amdhal's Law
  - Iron Law
- Introduction to Verilog

What we have learned so far (2/2)

- Single-cycle implementation of CPU
  - All instructions run as slow as the slowest instruction
- Multi-cycle implementations of CPU
  - All instructions run for their own latencies
  - Still only one instruction in CPU at a given time
- Pipelined implementation of CPU
  - Actually another multi-cycle implementation of CPU, but just better!

Today's topic !!

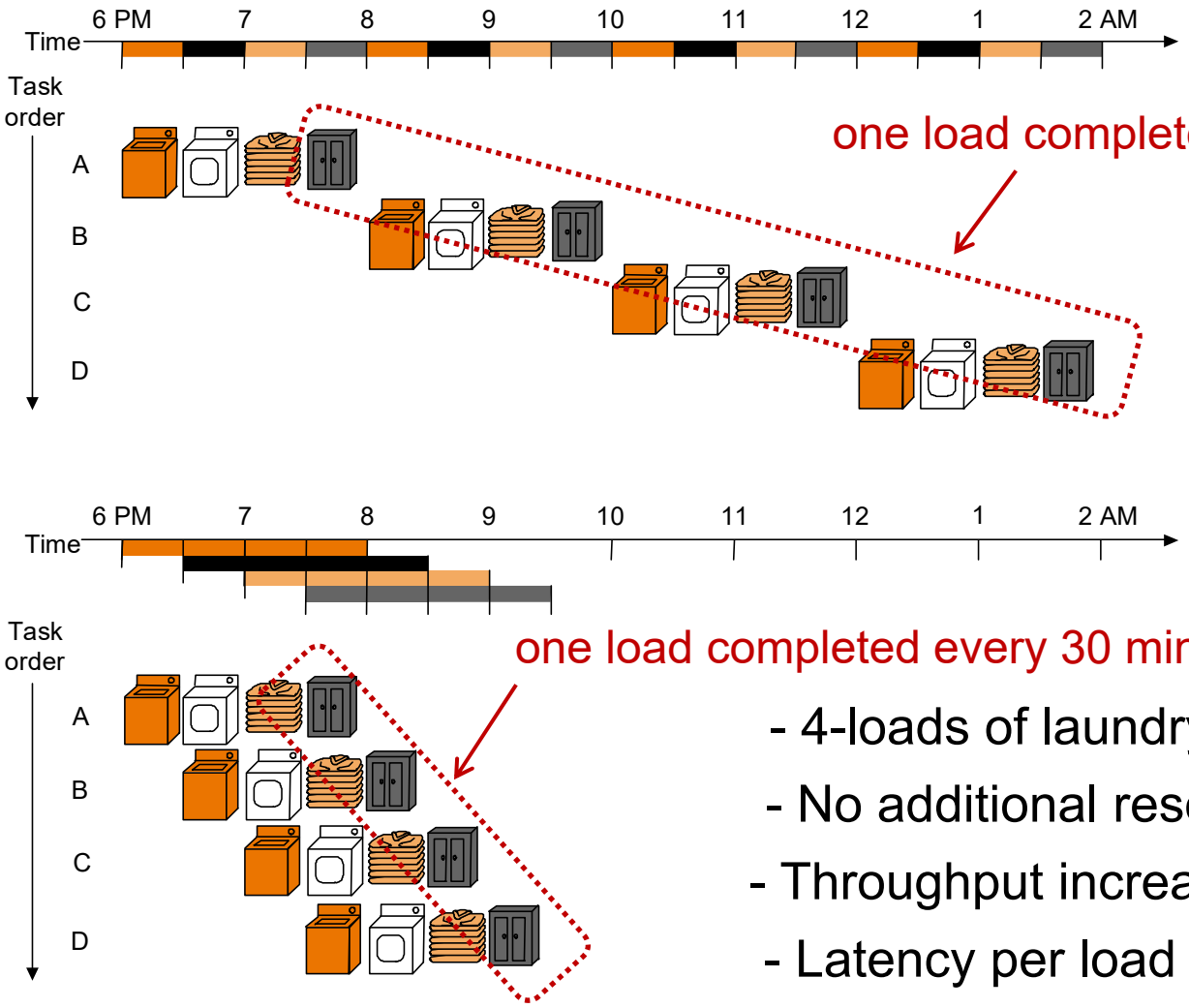
Doing laundry more quickly: in theory



- "Place one dirty load of clothes in the washer"
- "When the washer is finished, place the wet load in the dryer"
- "When the dryer is finished, place the dry load on a table and fold"
- "When folding is finished, put the clothes into the closet"

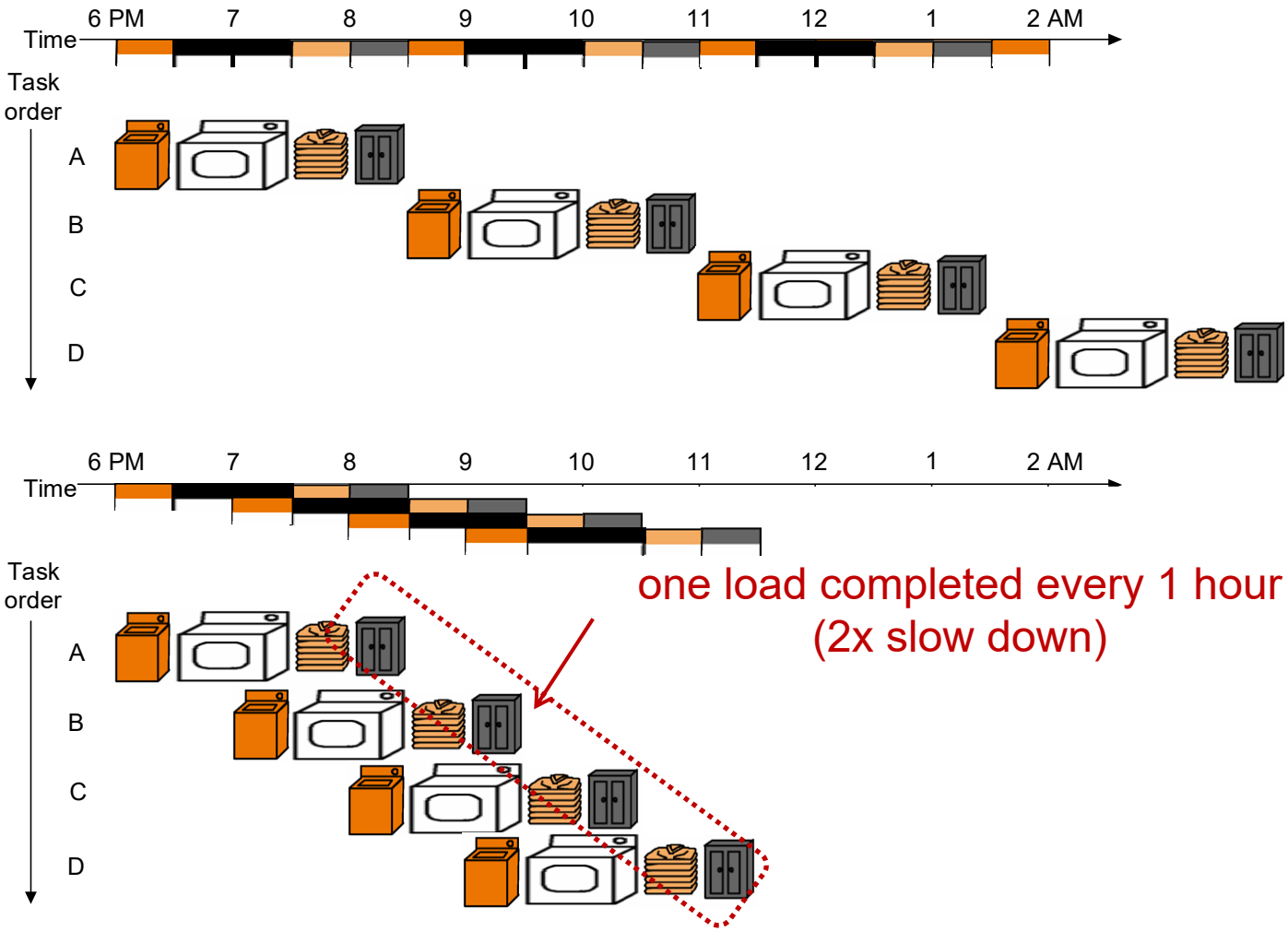
- Steps to do a load are sequentially dependent
- No dependence between different loads

Doing laundry more quickly: in theory



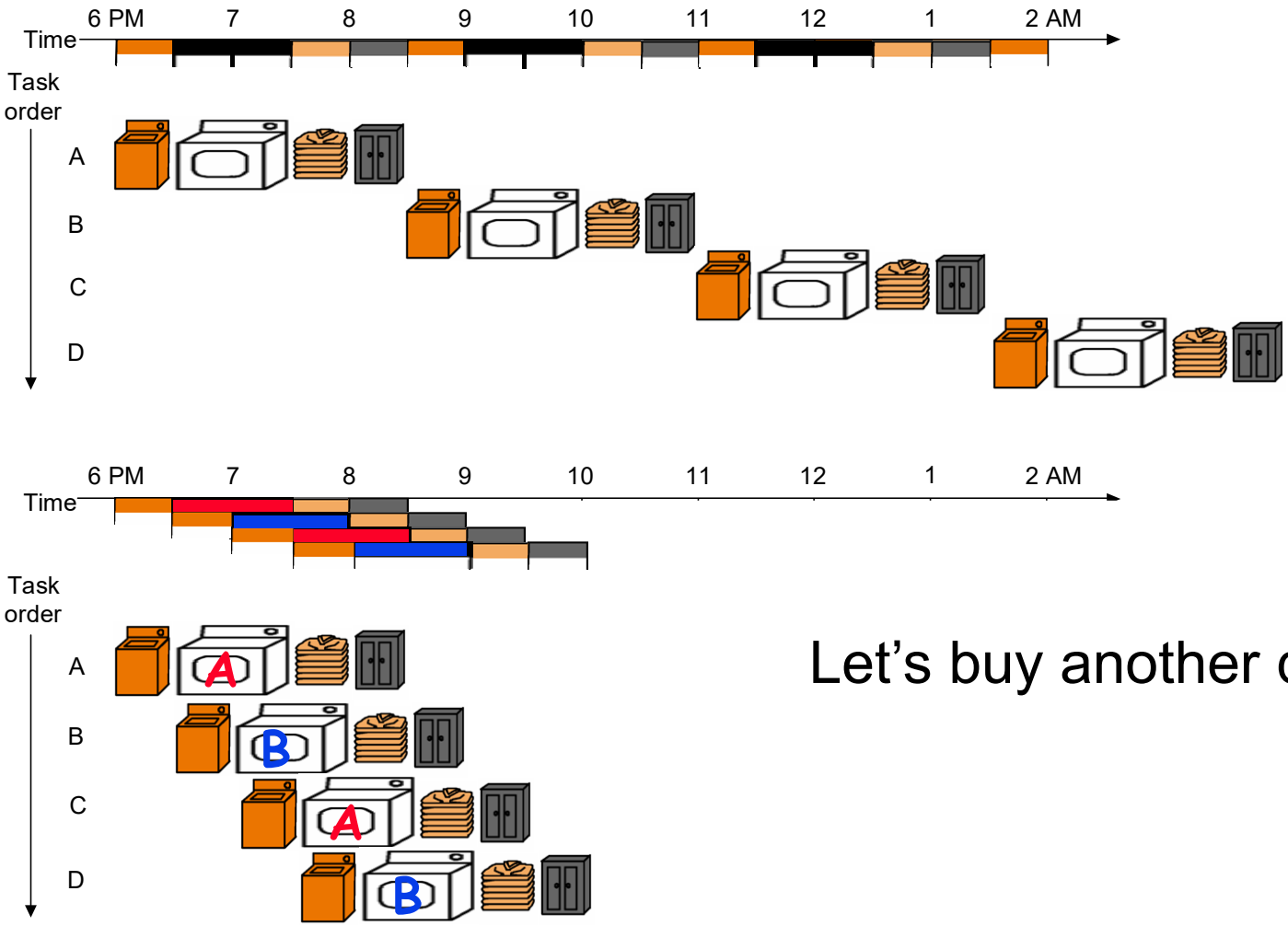
- 4-loads of laundry in parallel
- No additional resources
- Throughput increased by ~4x
- Latency per load is the same

Doing laundry more quickly: in practice



The slowest step (i.e., dryer) decides overall throughput.

Doing laundry more quickly: in practice



Let's buy another dryer.

Throughput restored (2 loads per hour) using 2 dryers.

Pipeline Idealism

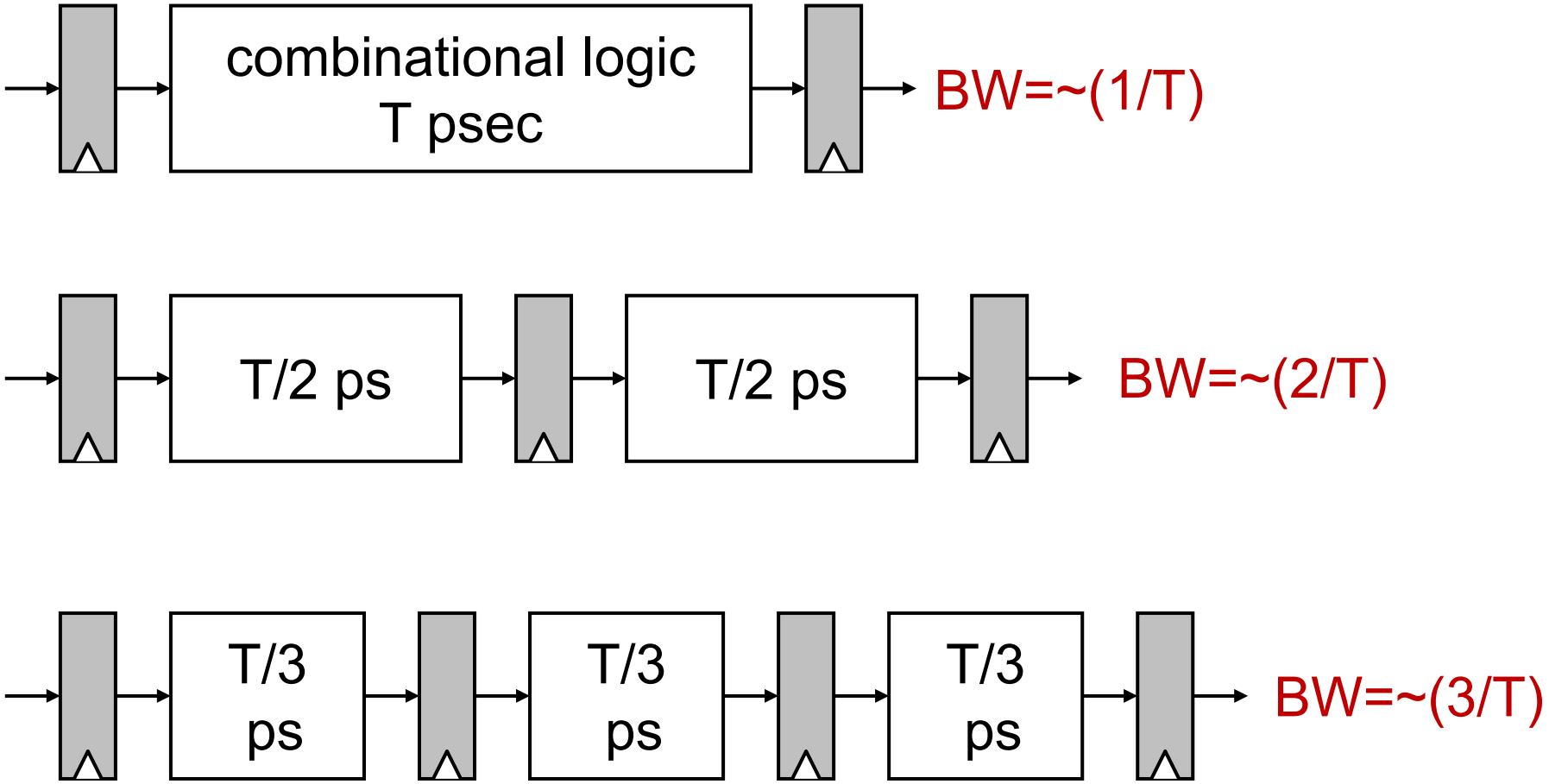
Motivation: "Increase throughput with little increase in HW"

- Repetition of identical operations // Same task!
  The same operation is repeated on a large number of different inputs
- Repetition of independent operations // Independent sub-task!
  No ordering dependencies between repeated operations
- Uniformly partitionable sub-operations // Same-length sub-task!
  Can be evenly divided into uniform-latency sub-operations
  (that do not share resources)

Good examples: automobile assembly line, doing laundry …

…. maybe instruction pipeline???

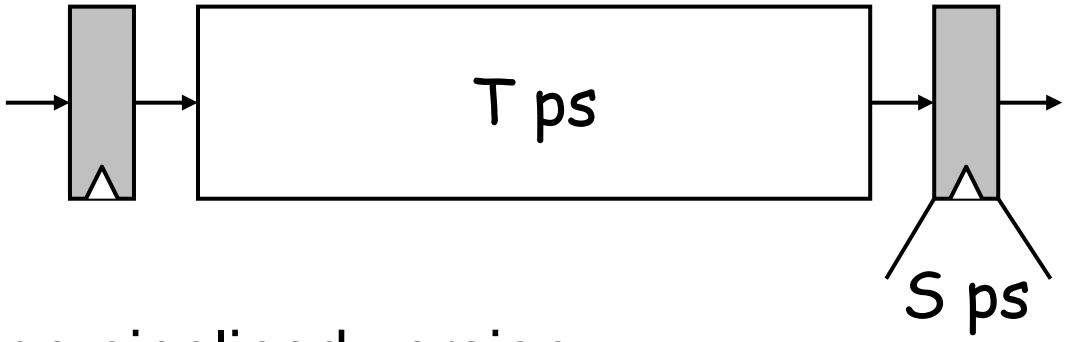
Ideal Pipelining



Performance Model

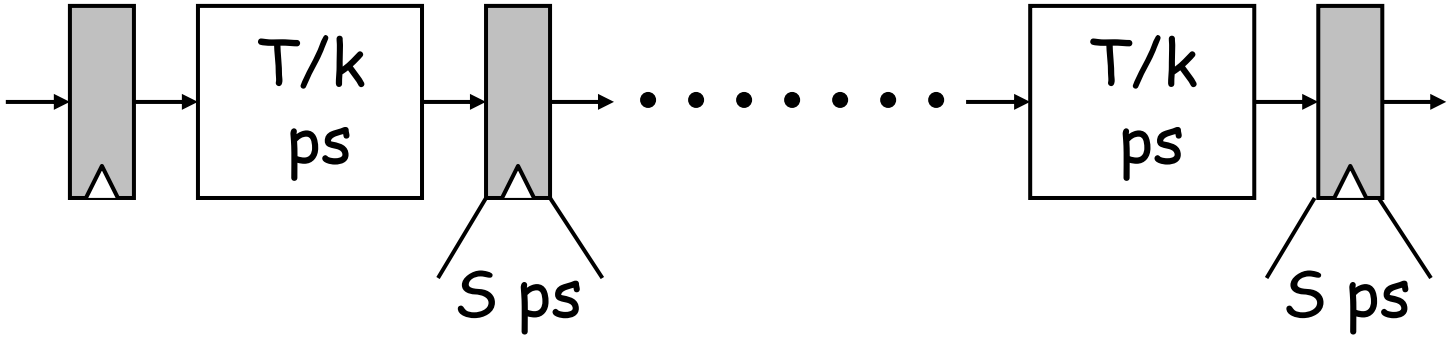
- Non-pipelined version with delay T

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



- k-stage pipelined version

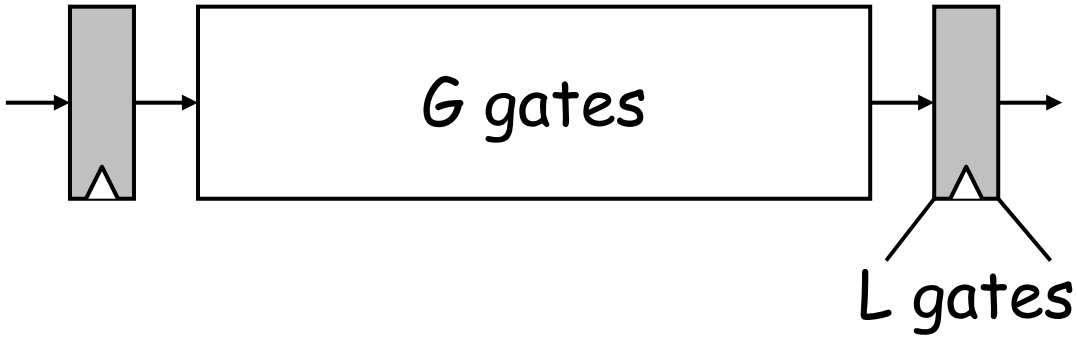
  $BW_{k\text{-stage}} = 1 / (T/k + S)$



Cost Model

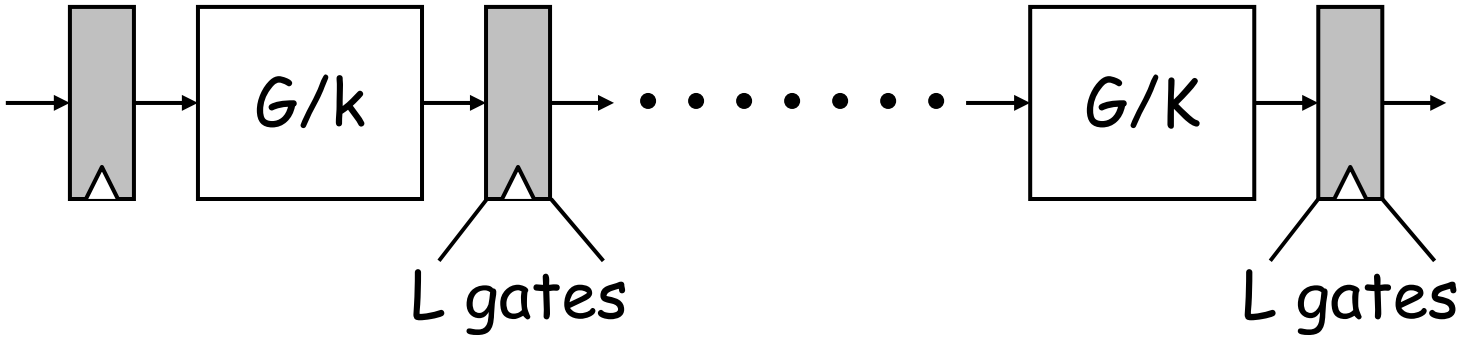
- Non-pipelined version with combinational cost G
  (cost = perf., area, power,..)

  Cost = G + L where L = latch cost



- k-stage pipelined version

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



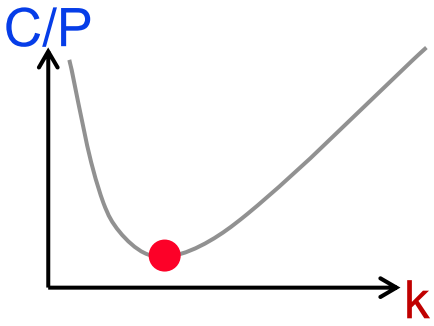
Cost/Performance Trade-off

[Peter M. Kogge, 1981]

Cost/Performance:

C/P = [L*k + G] / [1/(T/k + S)] = (L*k + G) (T/k + S)

= L*T + G*S + L*S*k + G*T/k



Optimal Cost/Performance?

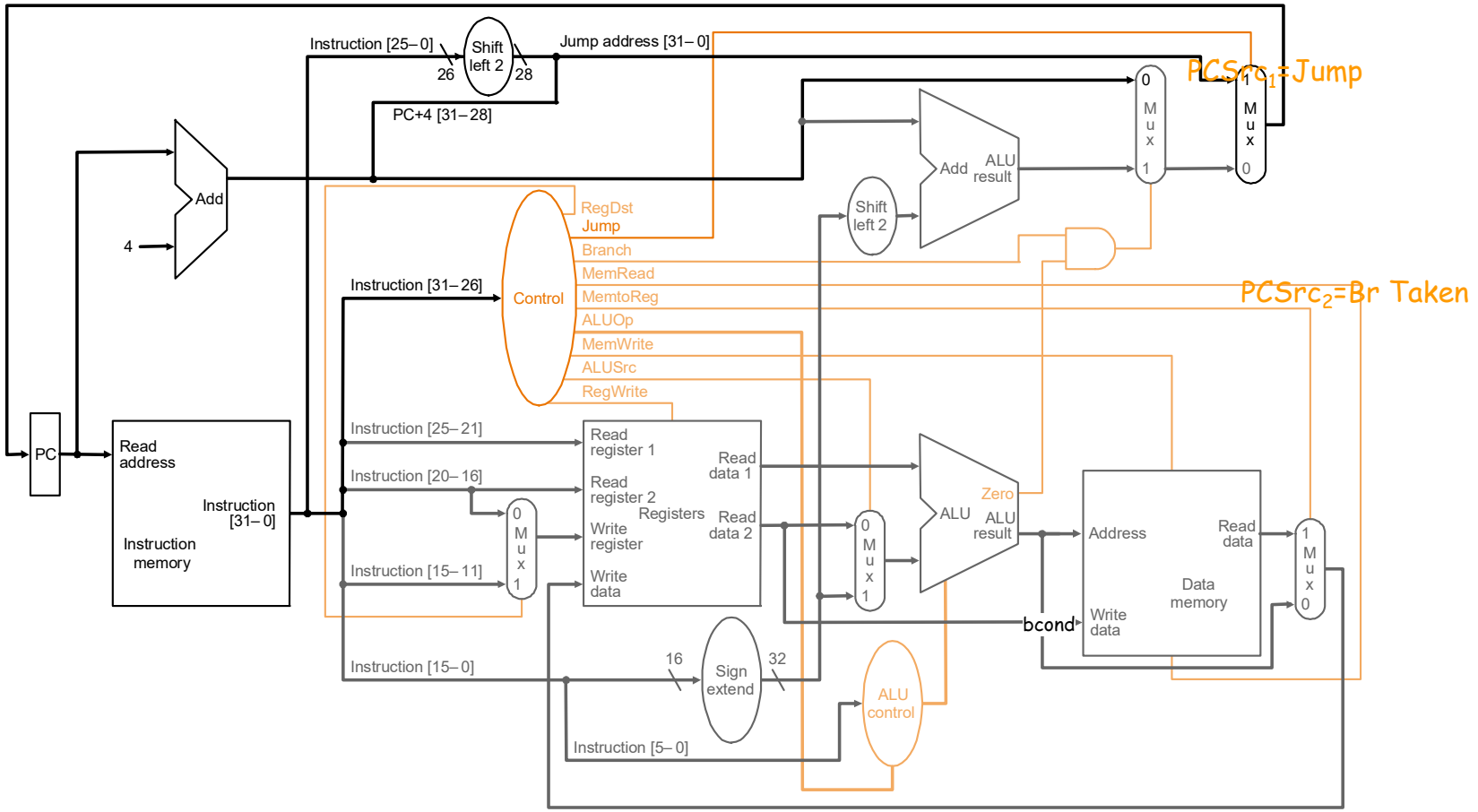
Let's find min. C/P for choice of k

$$\frac{d}{dk}\left(\frac{Lk+G}{\frac{T}{k}+S}\right) = 0 + 0 + LS - \frac{GT}{k^2}$$

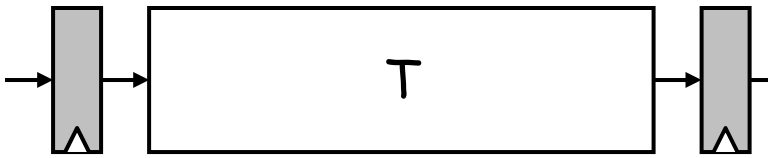
$$LS - \frac{GT}{k^2} = 0$$

$$k_{opt} = \sqrt{\frac{GT}{LS}}$$

The Reality of Instruction Pipelining….



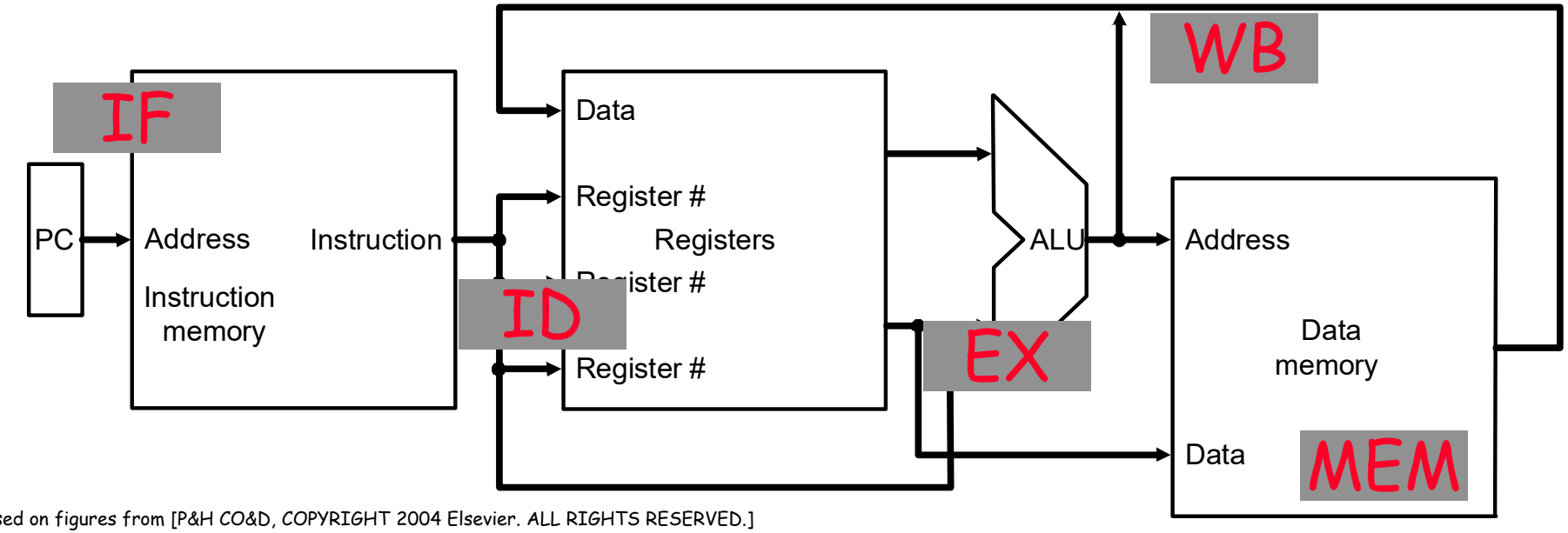
ALU operation



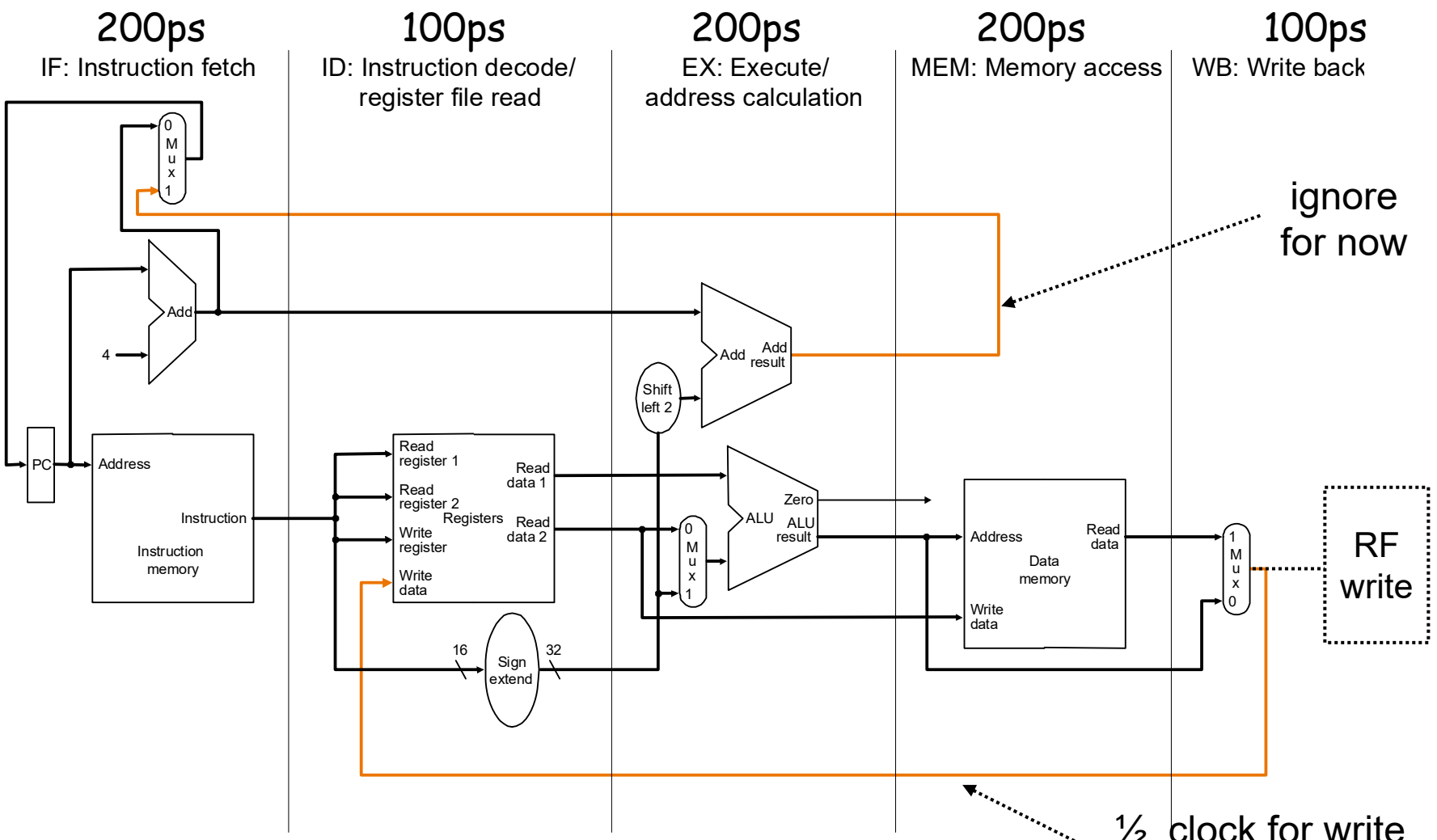
BW=~(1/T)

RISC Instruction Processing

- 5 generic steps
  - Instruction fetch
  - Instruction decode and operand fetch
  - ALU/execute
  - Memory access (not required by non-mem instructions)
  - Write-back



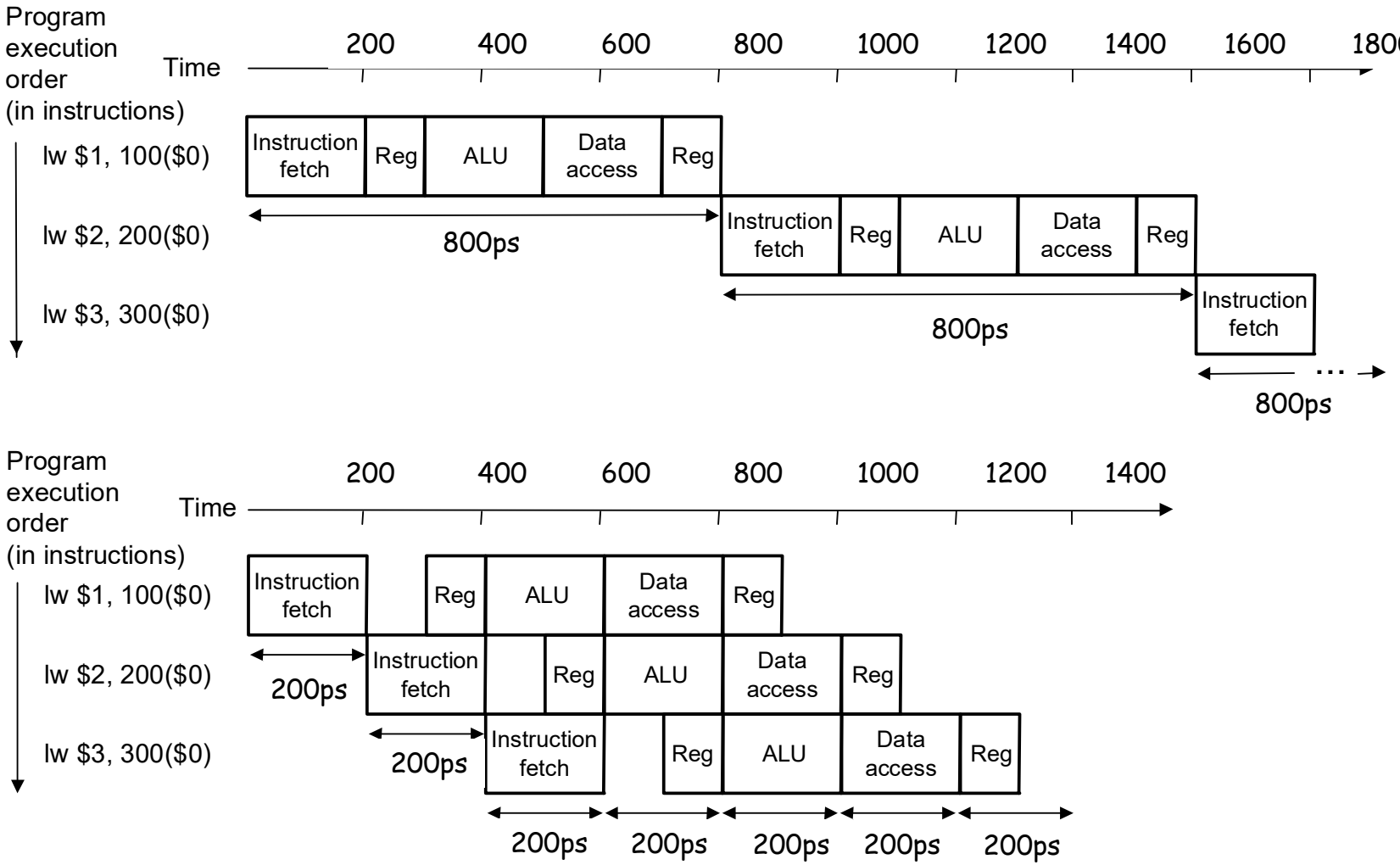
Dividing into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

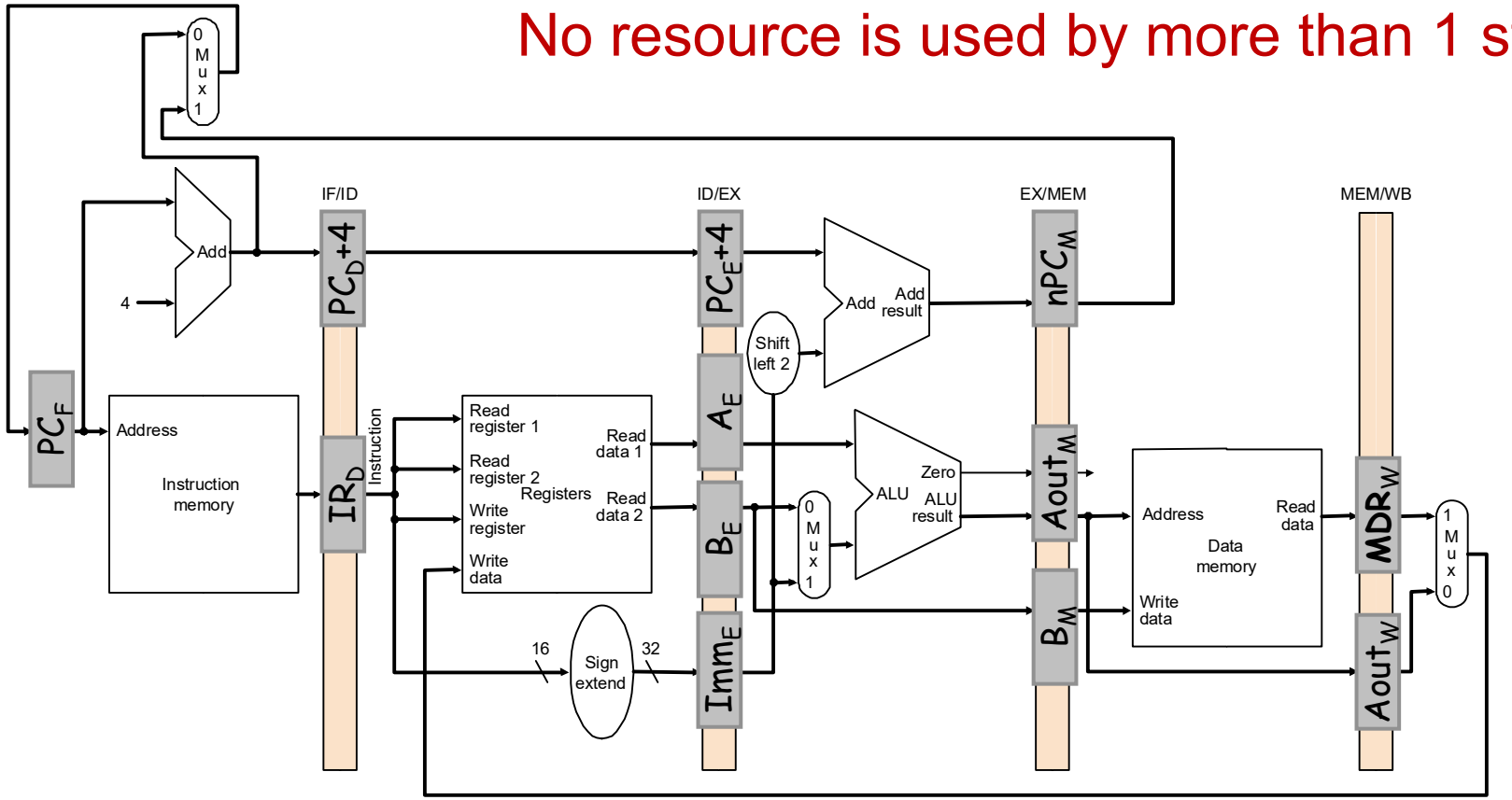
Pipelining

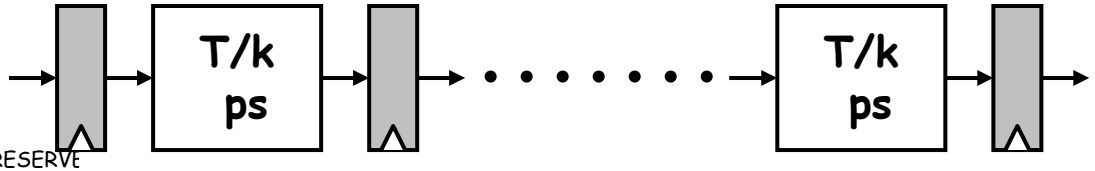


5-stage speedup is 4, not 5 as predicated by the ideal model

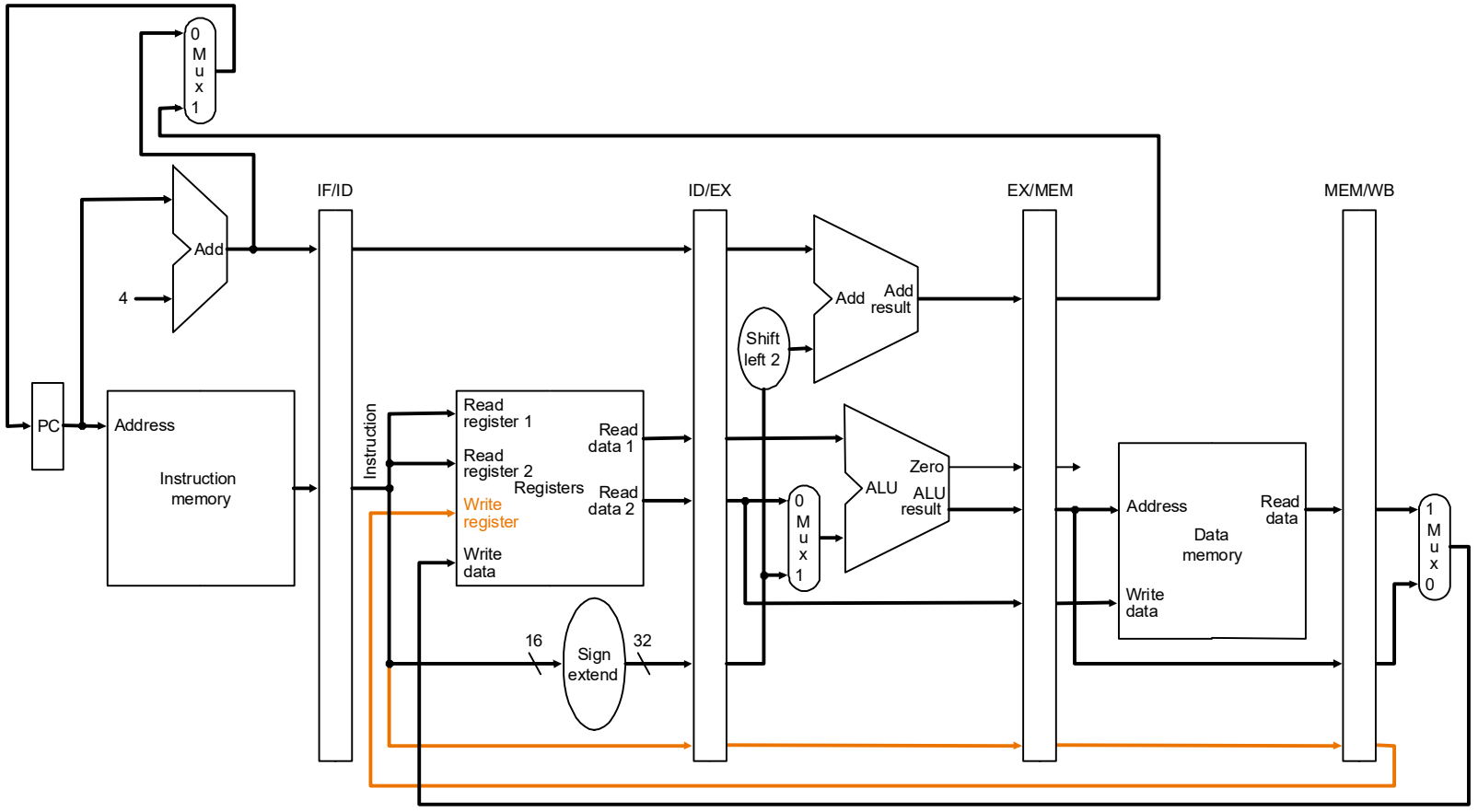
Pipeline Registers

No resource is used by more than 1 stage!



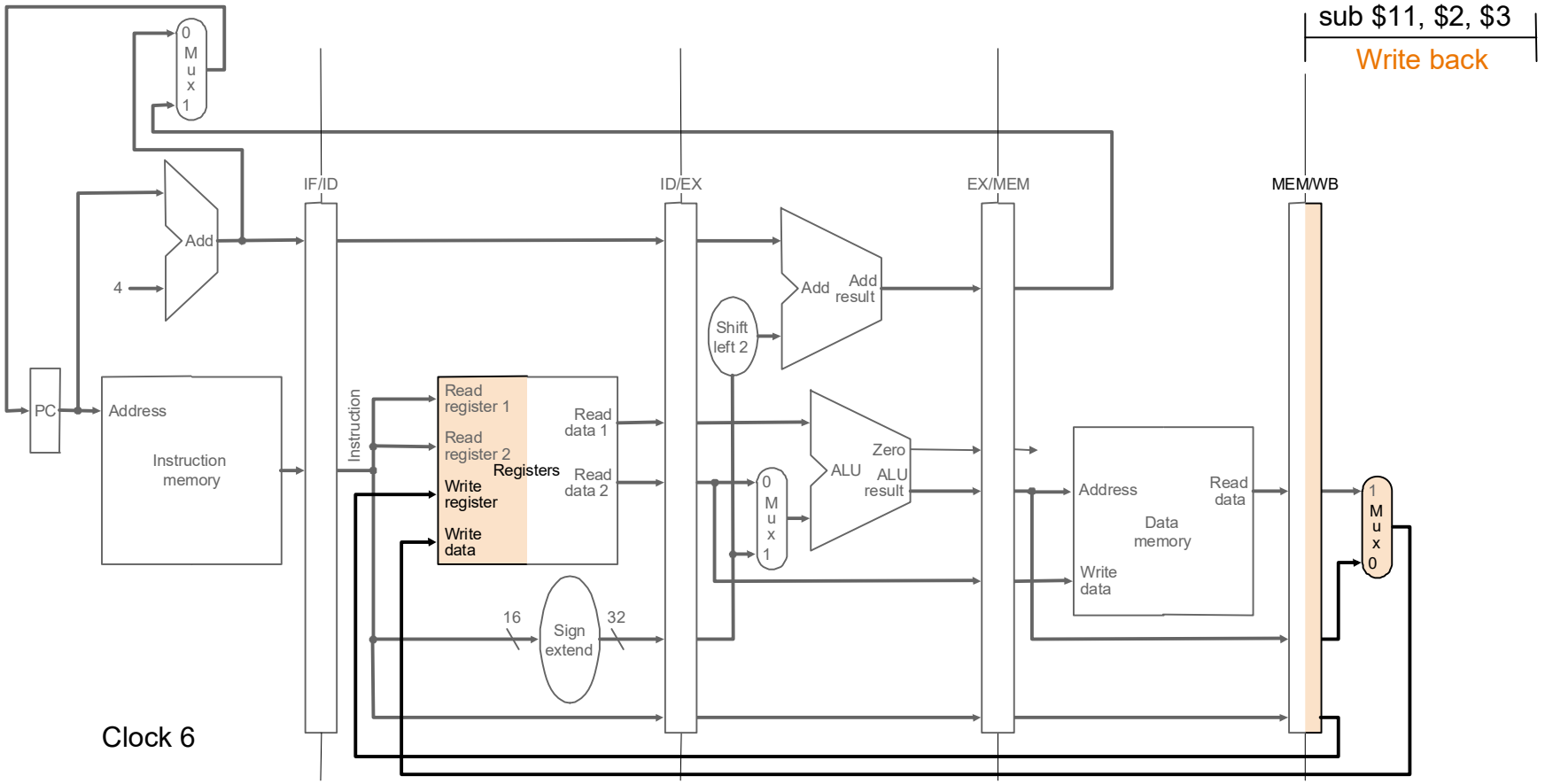


Pipelined Operation (1/2)

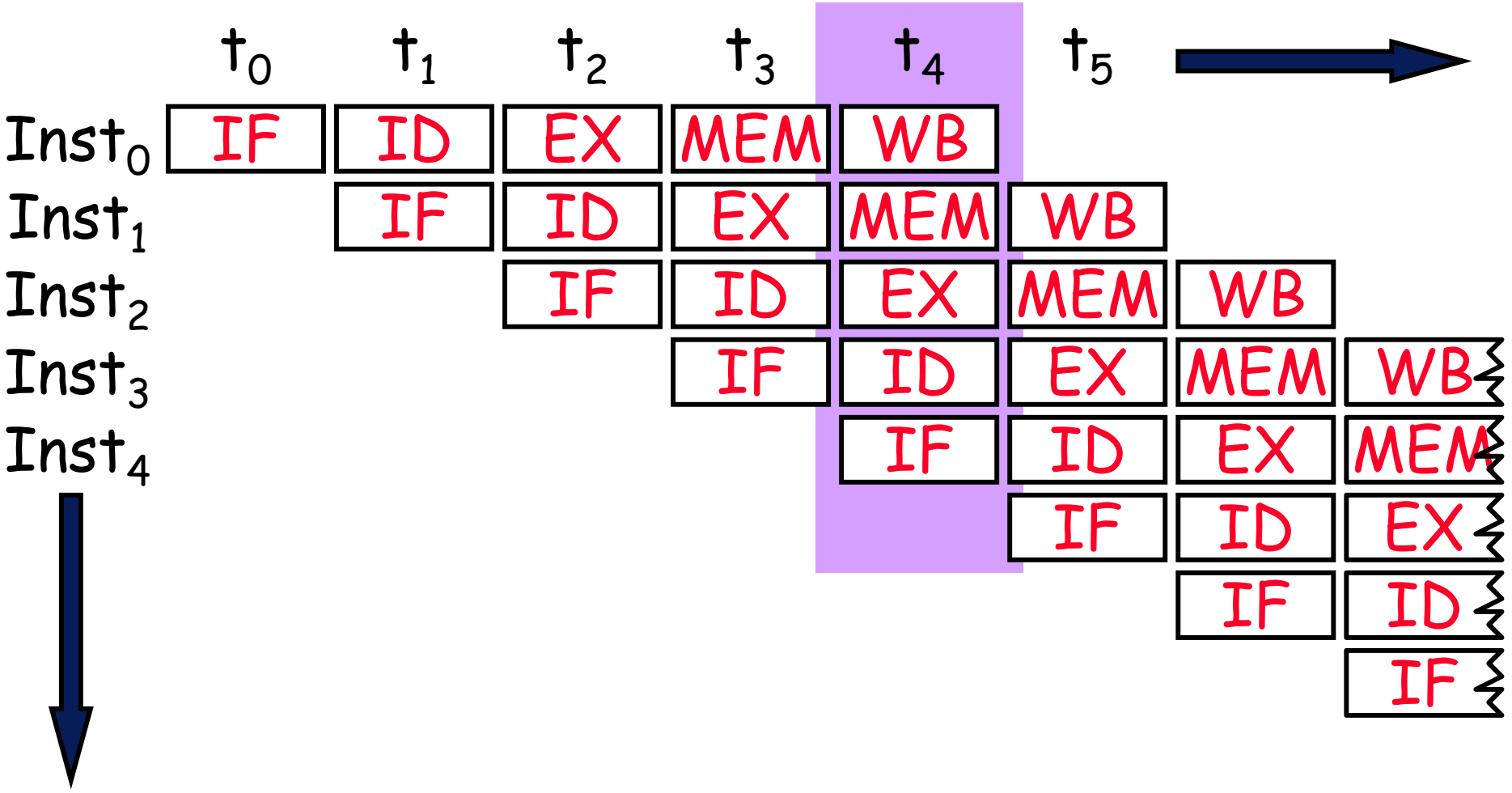


Both signal and data should be forwarded to the right state at the right time!

Pipelined Operation (2/2)



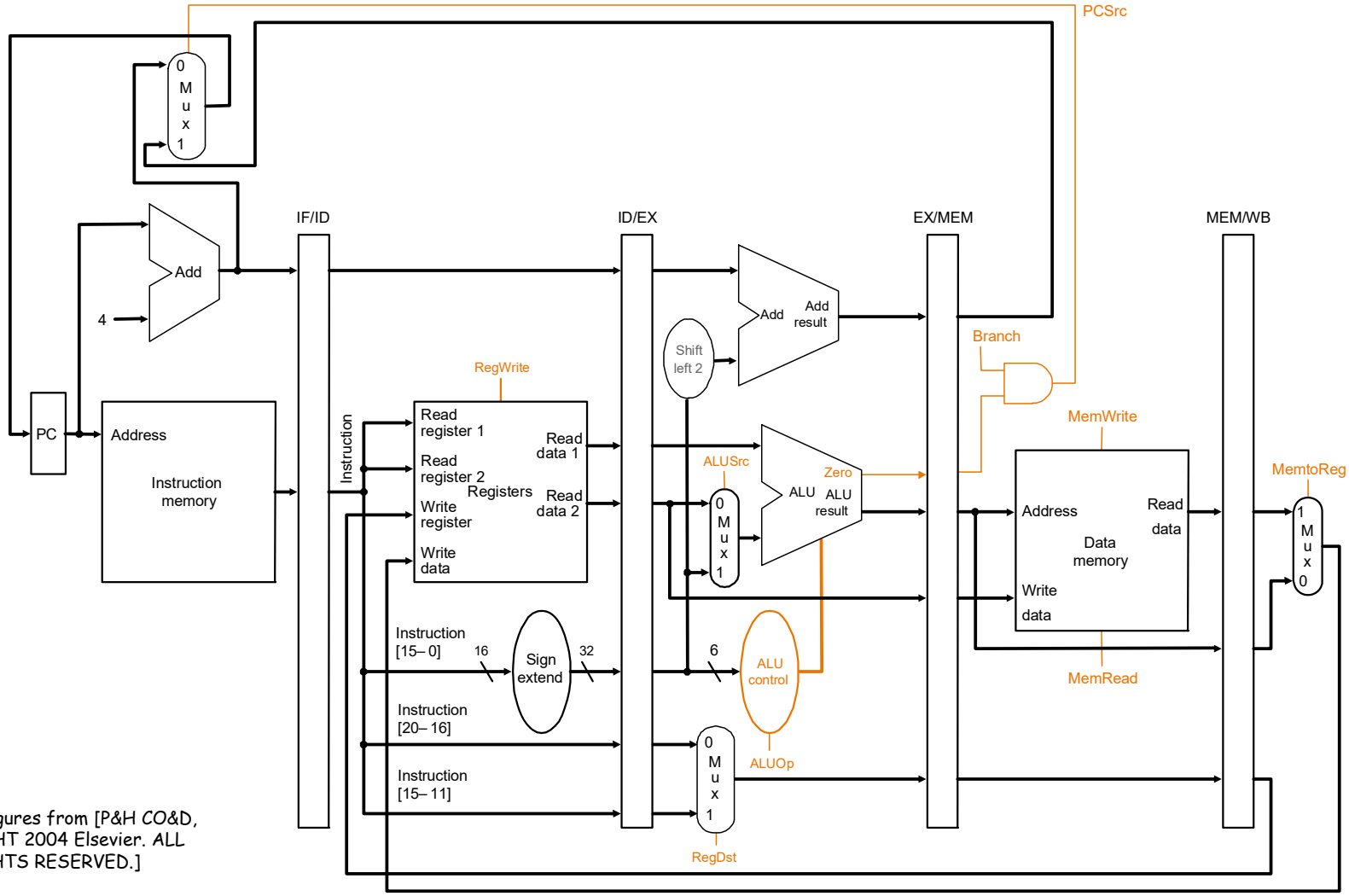
Illustrating Pipeline Operation: Operation View



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

Existing controls do not work with pipelining

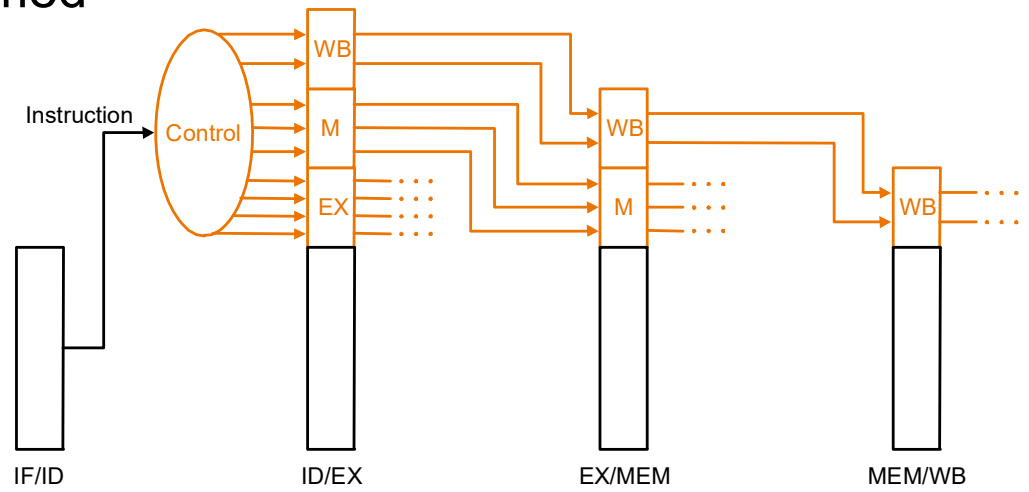


Control signals asserted/deasserted at wrong timing..

Sequential Control: Special Case

- For a given instruction
  - Same control settings as single-cycle, but
  - Control signals required at different cycles, depending on stage

(1) Decode once using the same logic as single-cycle and buffer control signals until consumed

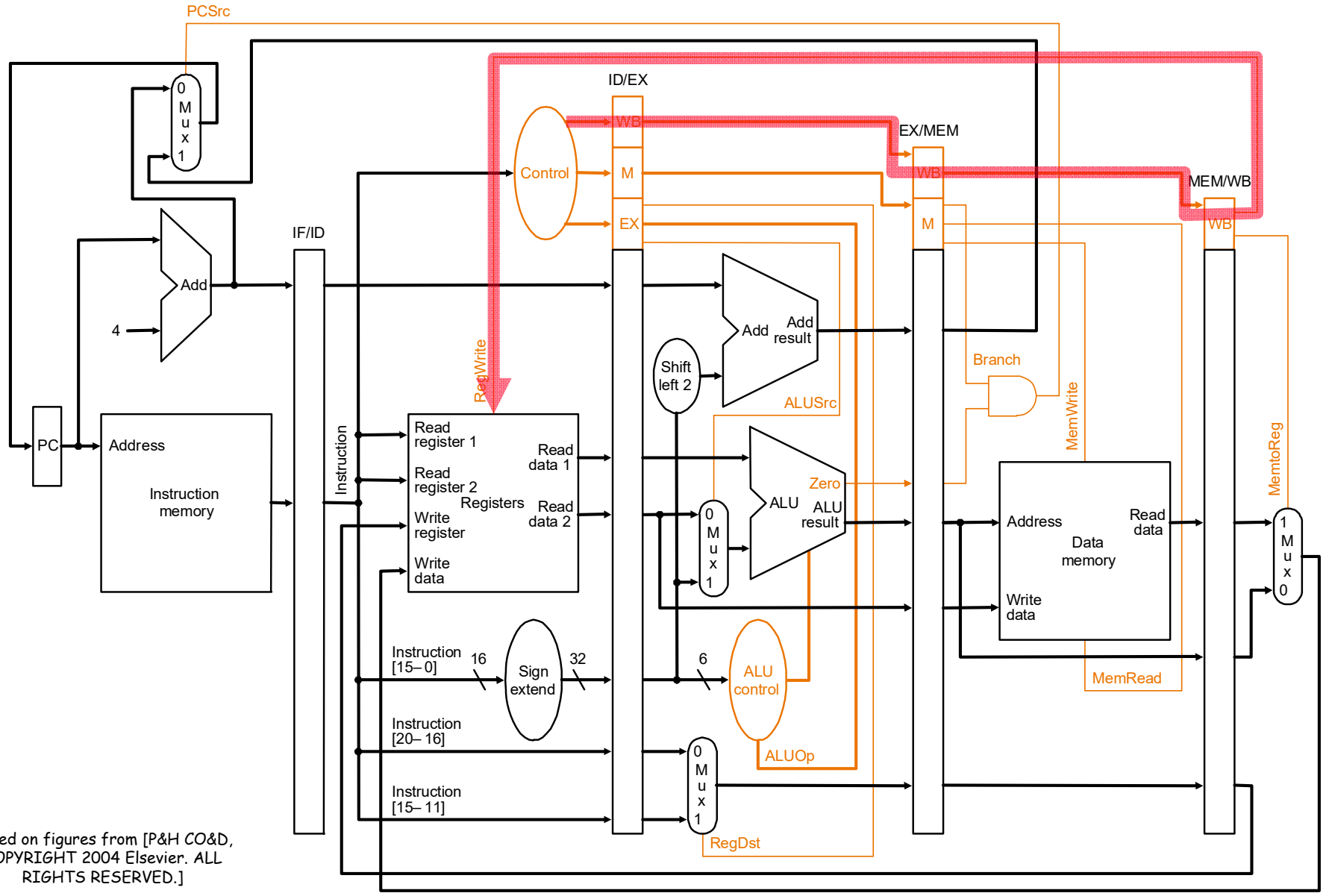


or

(2) carry relevant "instruction word/field" down the pipeline and decode locally within each stage.

Which one is better?

Pipelined Control



Reality of Instruction Pipelining

- No identical operations
  e.g., R-type, I-type & J-type
  ⇒ Unify instruction types
  - Combine instruction types to flow through "multi-function" pipe

- No independent operations
  e.g., Memory read VS. addition?
  ⇒ Remove dependency and/or busy resources
  - Duplicate contended resources
  - Inter-instruction dependency detection and resolution

- No uniform sub-operations
  e.g., Reg. read & write in 1 clock?
  Waiting data to be produced?
  ⇒ Balance pipeline stages
  - Stage-latency calculation to make balanced stages

So, MIPS ISA was made for improved pipelineability.

Question?

Announcements: *Homework #2 posted (due: 4/12)*

Reading: *Finish reading P&H Ch.4*

Handouts: *None*