

18-447

Computer Architecture  
Lecture 7: Pipelining

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 1/29/2014

# Can We Do Better than Microprogrammed Designs?

---

- What limitations do you see with the multi-cycle design?
- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening

# Can We Use the Idle Hardware to Improve Concurrency?

---

- Goal: Concurrency → throughput (more “work” completed in one cycle)
- Idea: When an instruction is using some resources in its processing phase, **process other instructions on idle resources** not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

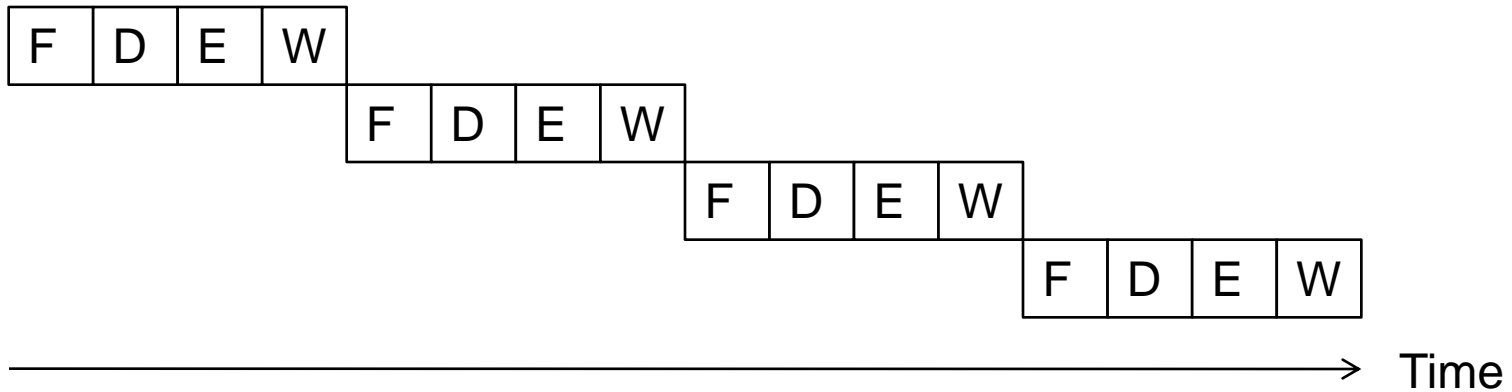
# Pipelining: Basic Idea

---

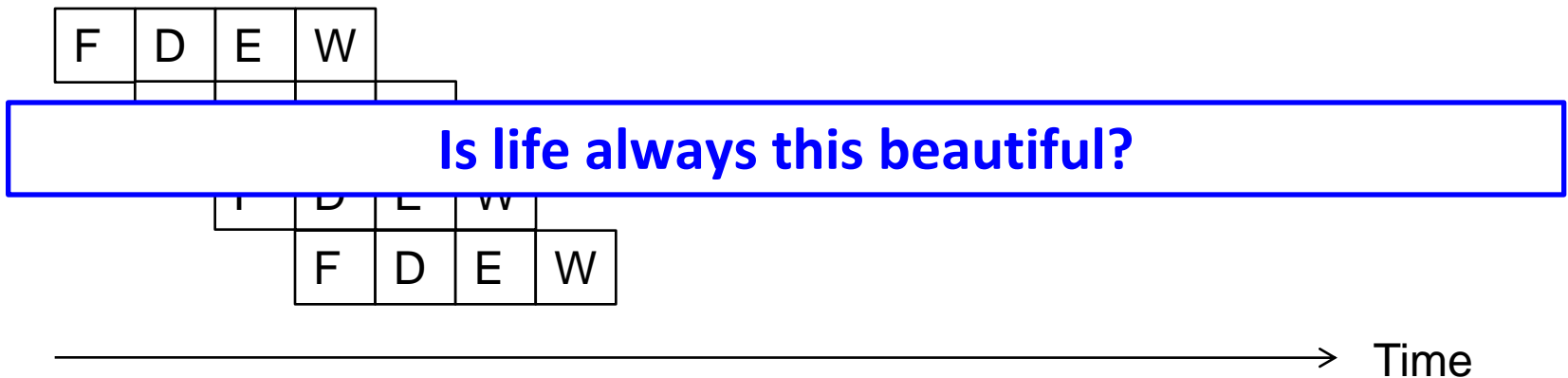
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this...

# Example: Execution of Four Independent ADDs

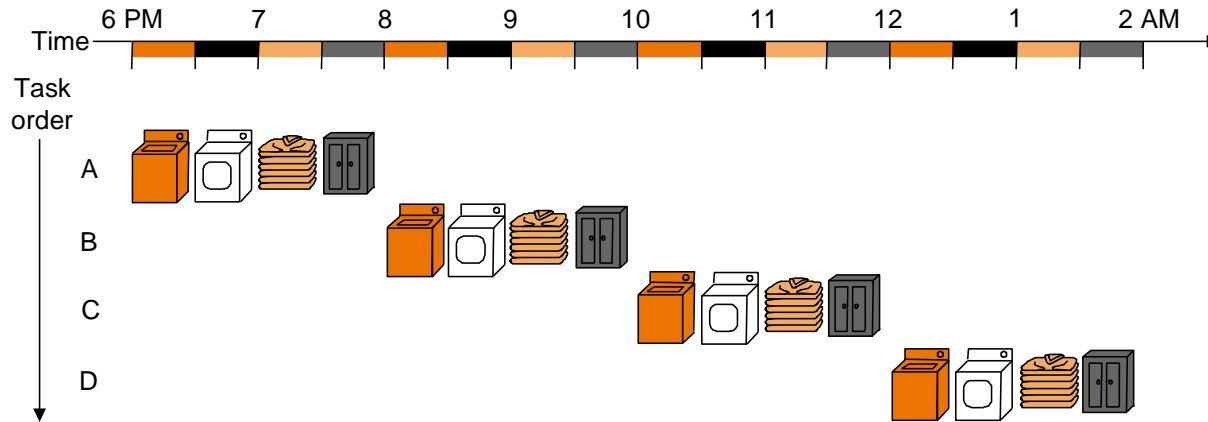
- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)

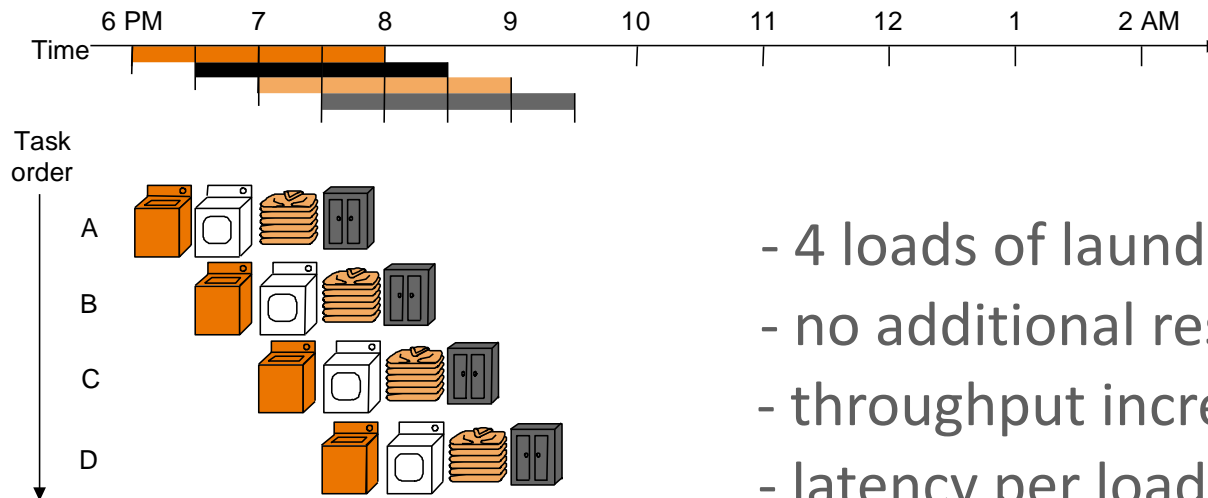
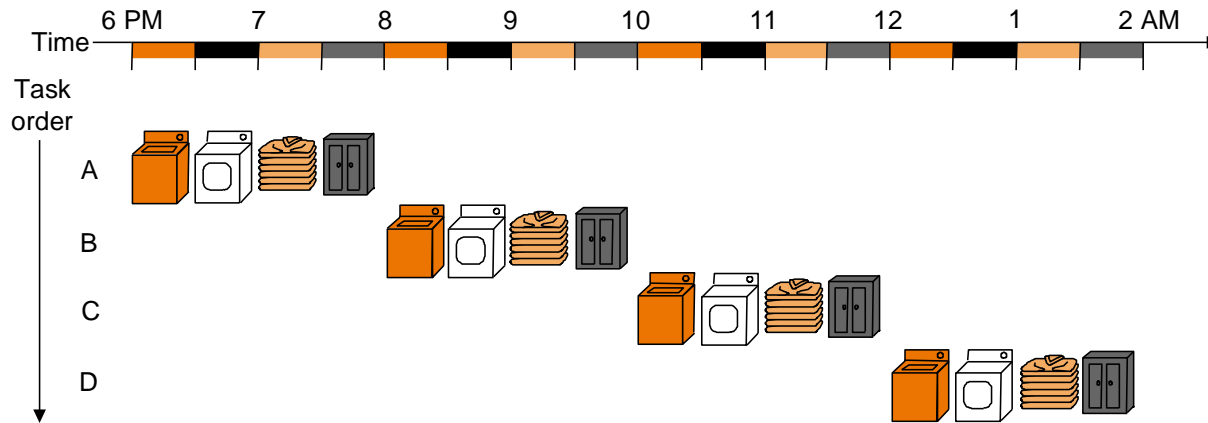


# The Laundry Analogy



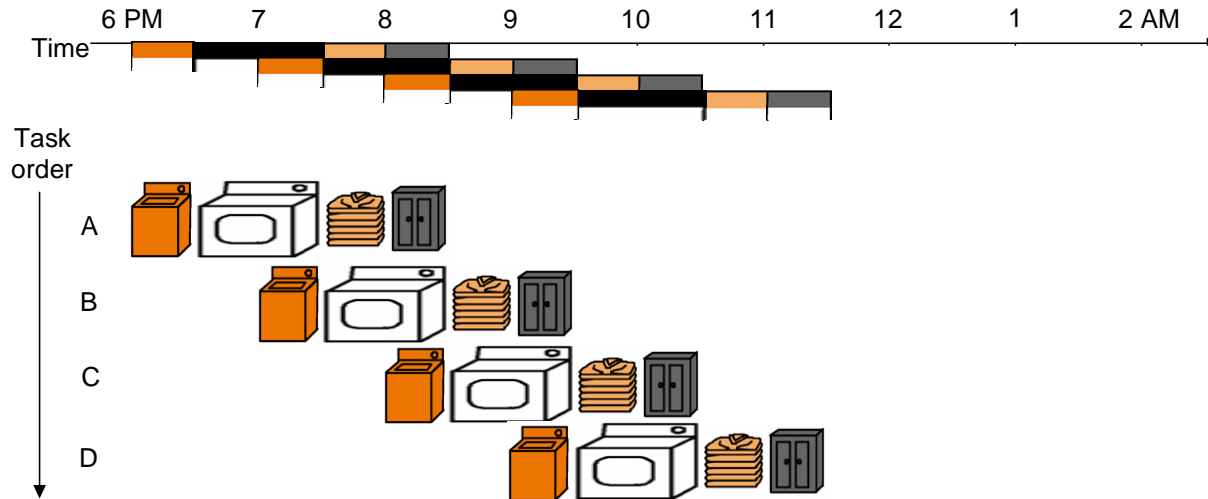
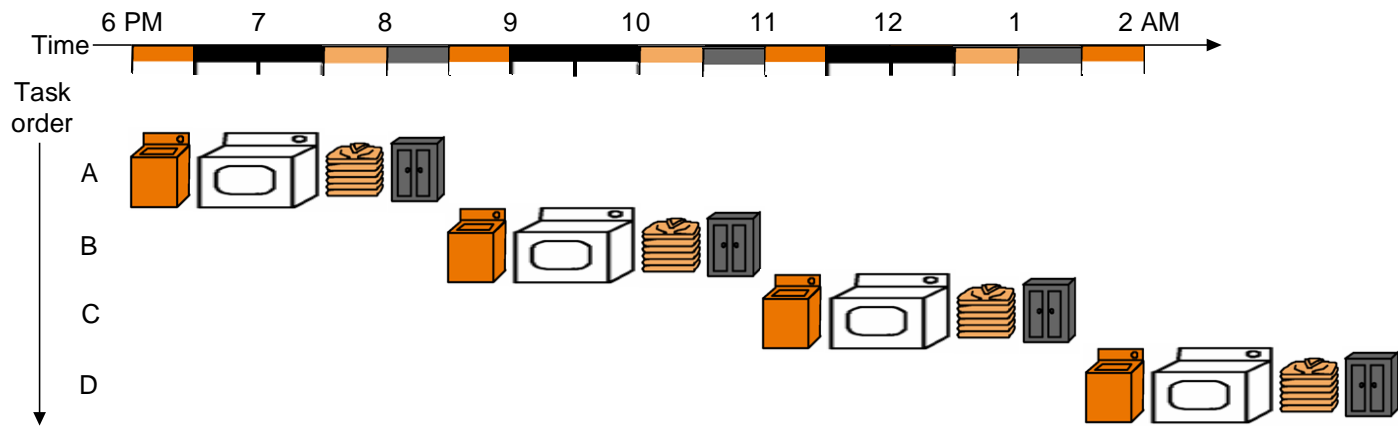
- “place one dirty load of clothes in the washer”
- “when the washer is finished, place the wet load in the dryer”
- “when the dryer is finished, take out the dry load and fold”
- “when folding is finished, ask your roommate (??) to put the clothes away”
  - steps to do a load are sequentially dependent
  - no dependence between different loads
  - different steps do not share resources

# Pipelining Multiple Loads of Laundry



- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

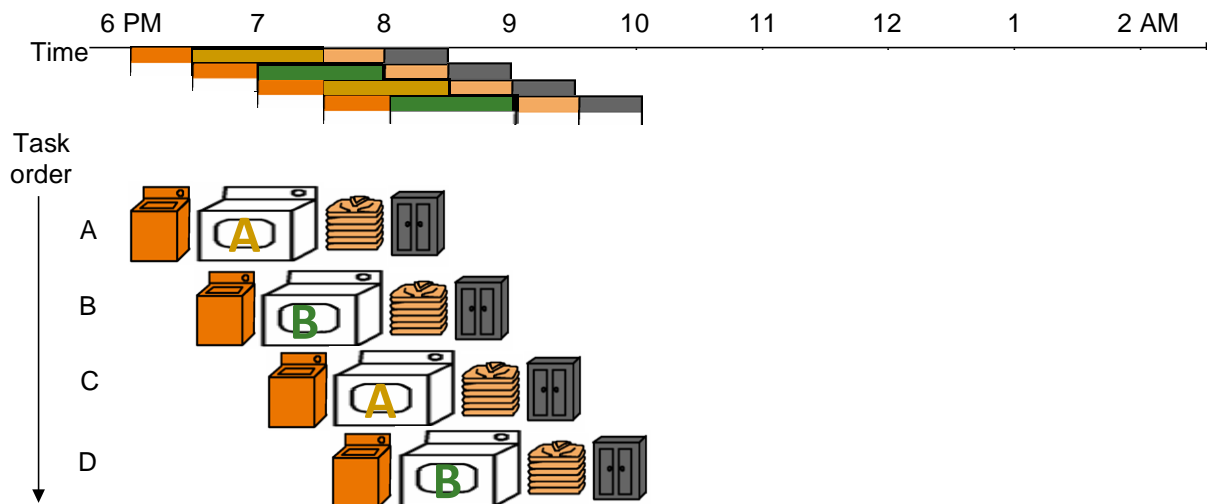
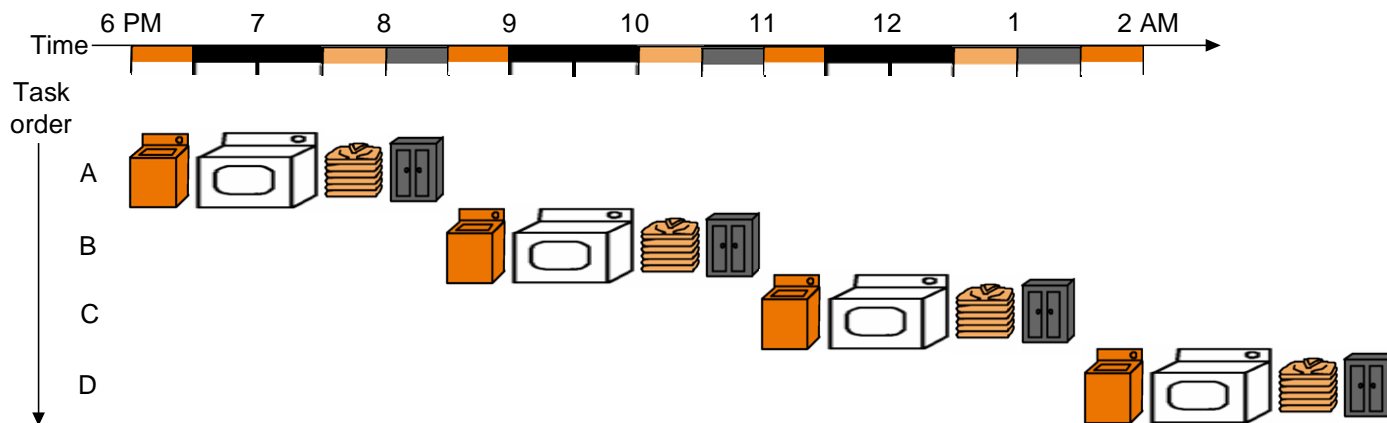
# Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput



# Pipelining Multiple Loads of Laundry: In Practice



Throughput restored (2 loads per hour) using 2 dryers

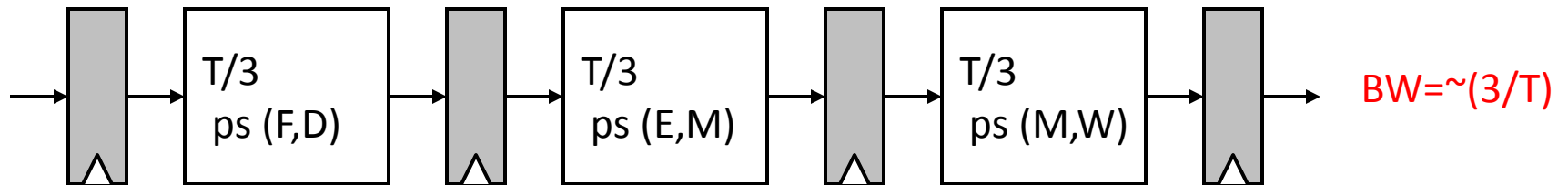
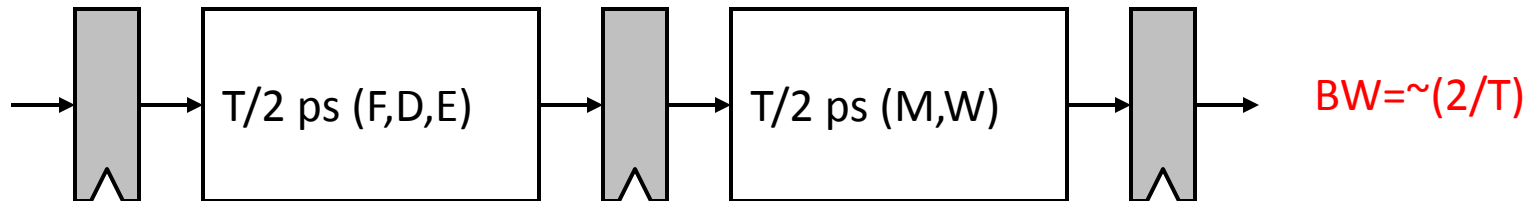
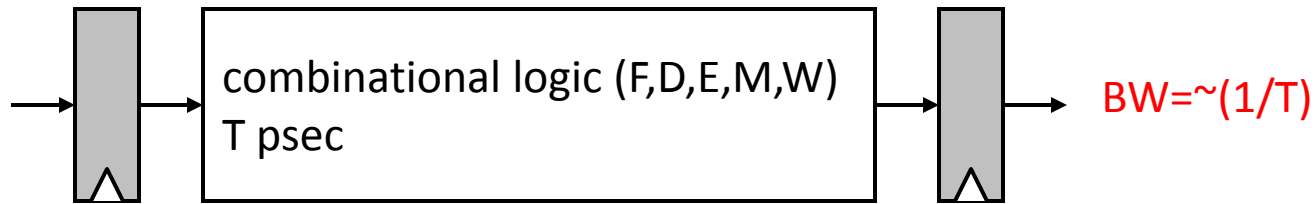
# An Ideal Pipeline

---

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs
- Repetition of **independent operations**
  - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?

# Ideal Pipelining

---



# More Realistic Pipeline: Throughput

---

- Nonpipelined version with delay  $T$

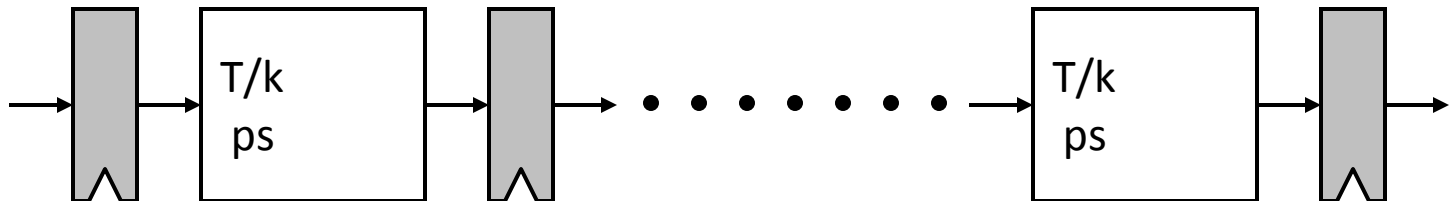
$$BW = 1/(T+S) \text{ where } S = \text{latch delay}$$



- k-stage pipelined version

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

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



# More Realistic Pipeline: Cost

---

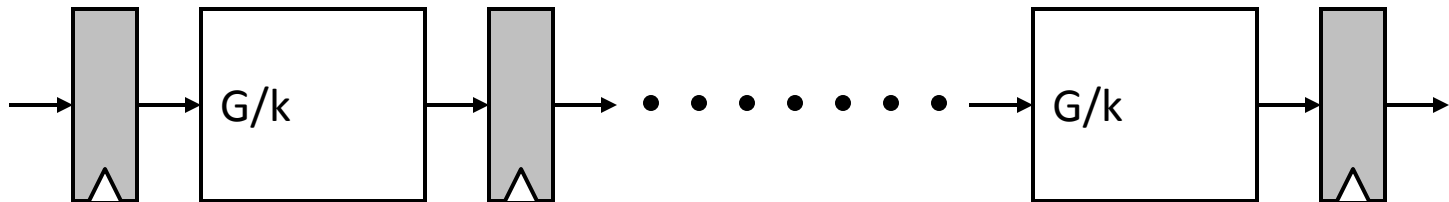
- Nonpipelined version with combinational cost  $G$

$\text{Cost} = G + L$  where  $L$  = latch cost



- $k$ -stage pipelined version

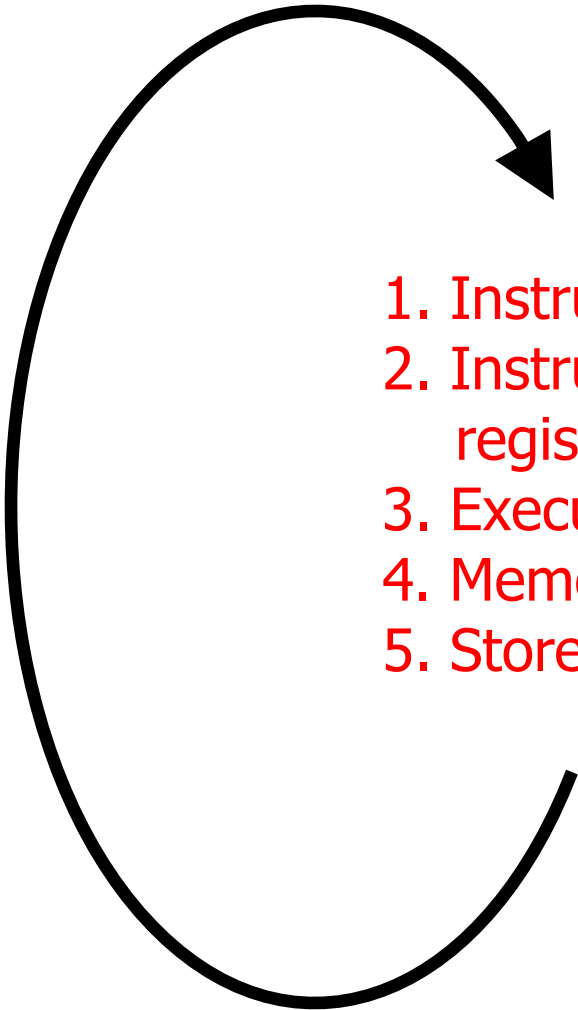
$\text{Cost}_{k\text{-stage}} = G + Lk$



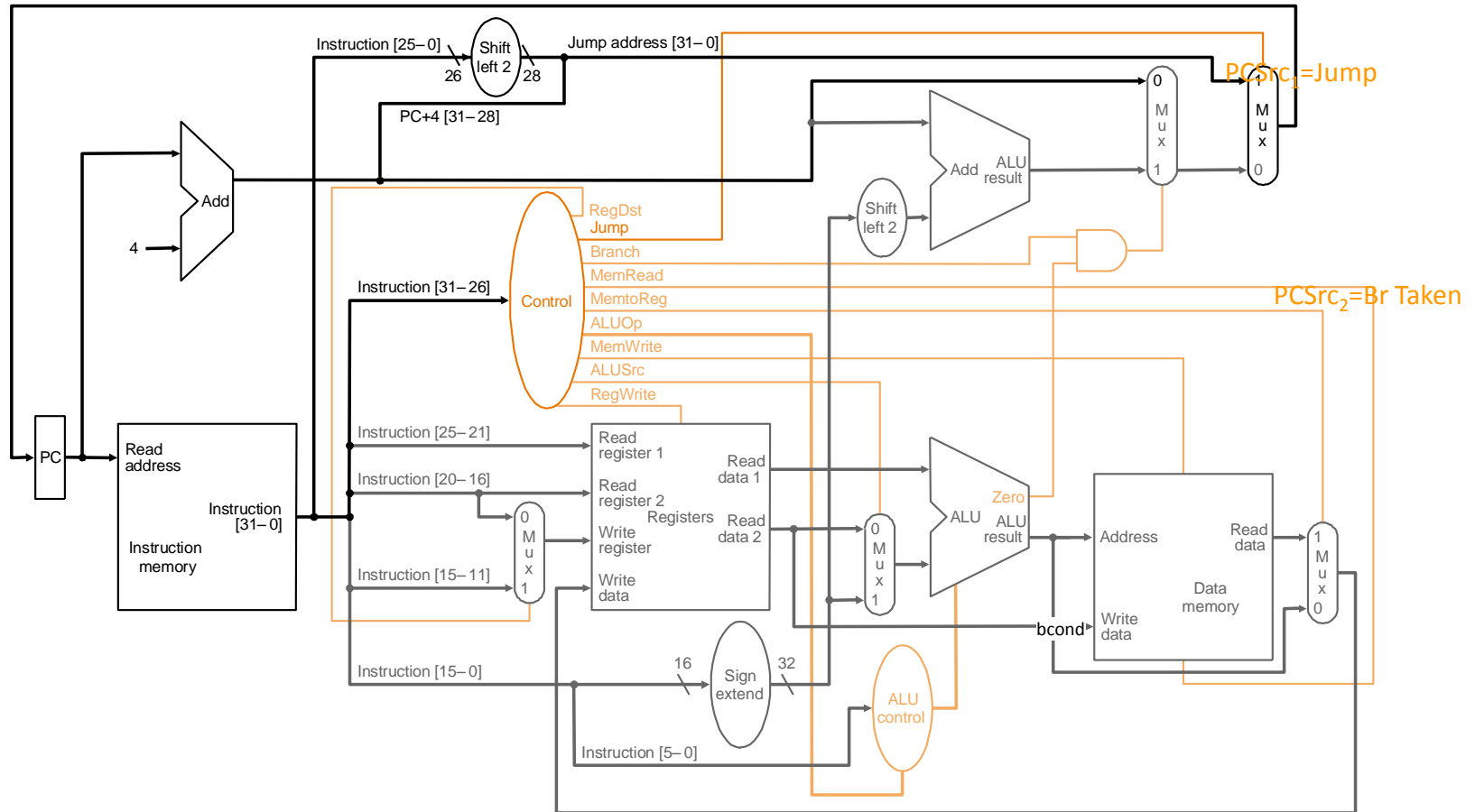
# Pipelining Instruction Processing

# Remember: The Instruction Processing Cycle

---

- 
1. Instruction fetch (IF)
  2. Instruction decode and register operand fetch (ID/RF)
  3. Execute/Evaluate memory address (EX/AG)
  4. Memory operand fetch (MEM)
  5. Store/writeback result (WB)

# Remember the Single-Cycle Uarch

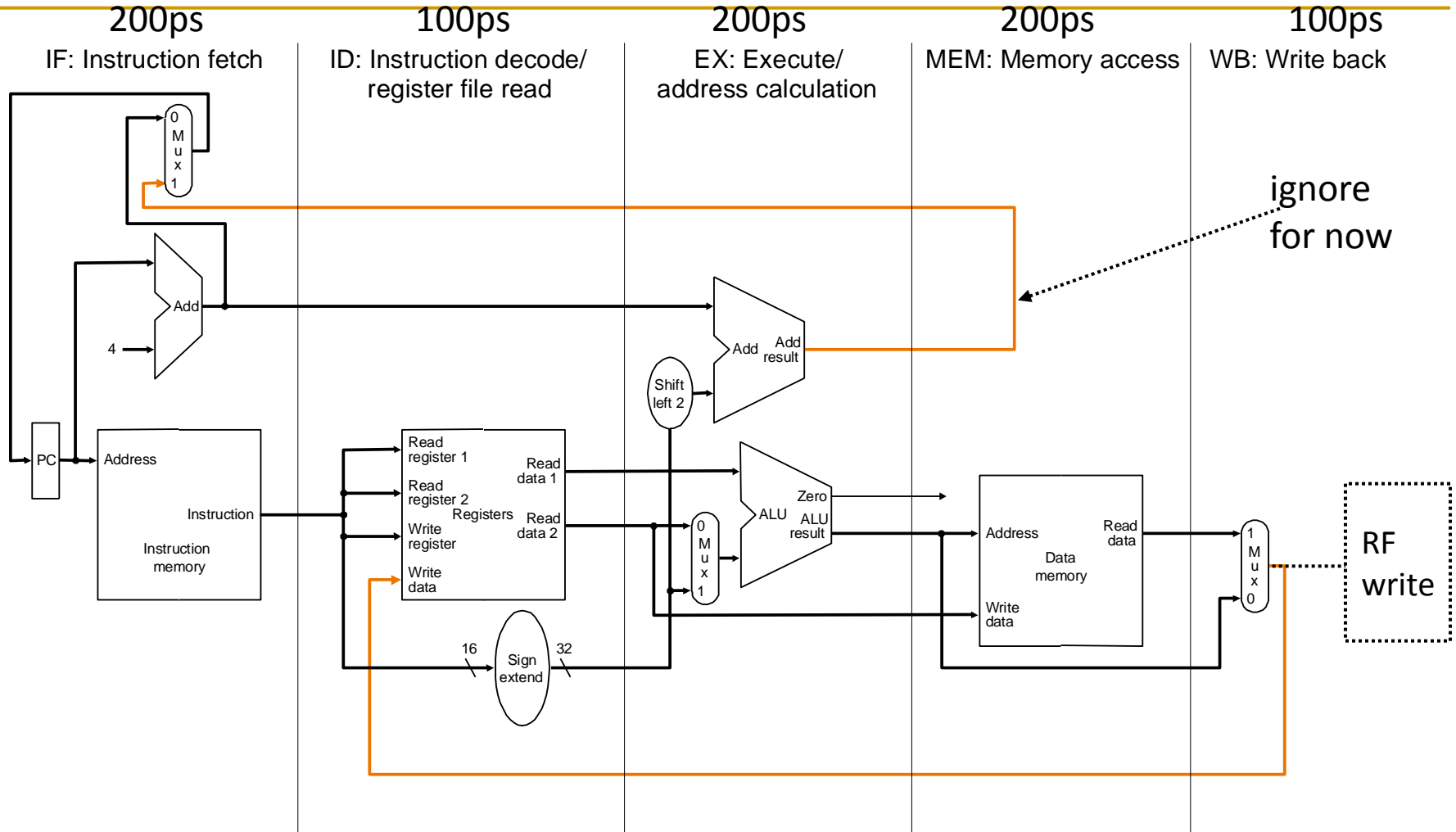


ALU operation





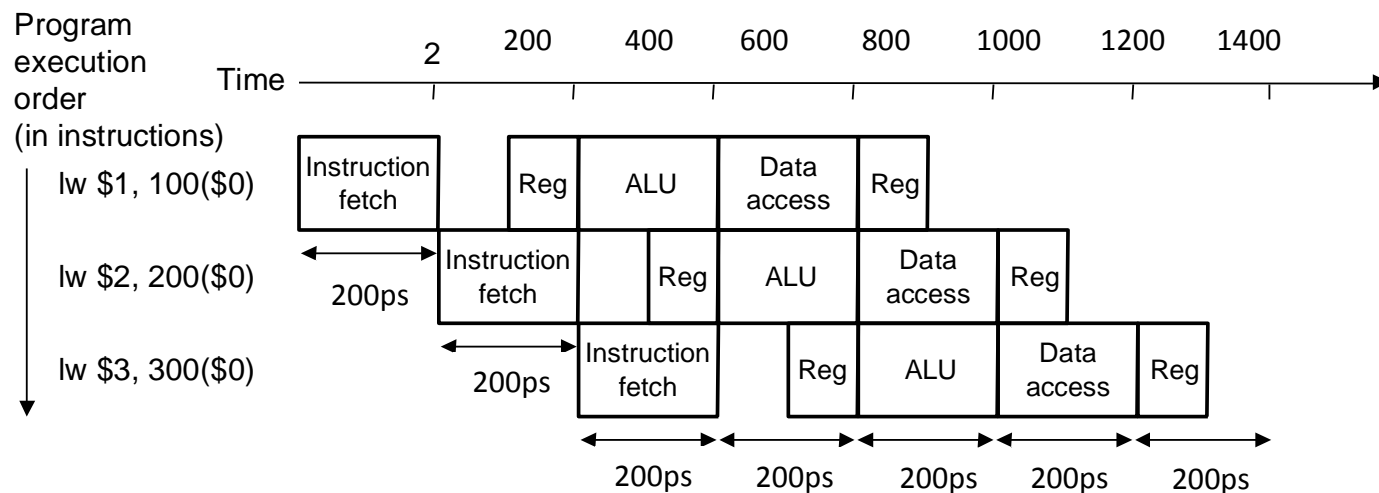
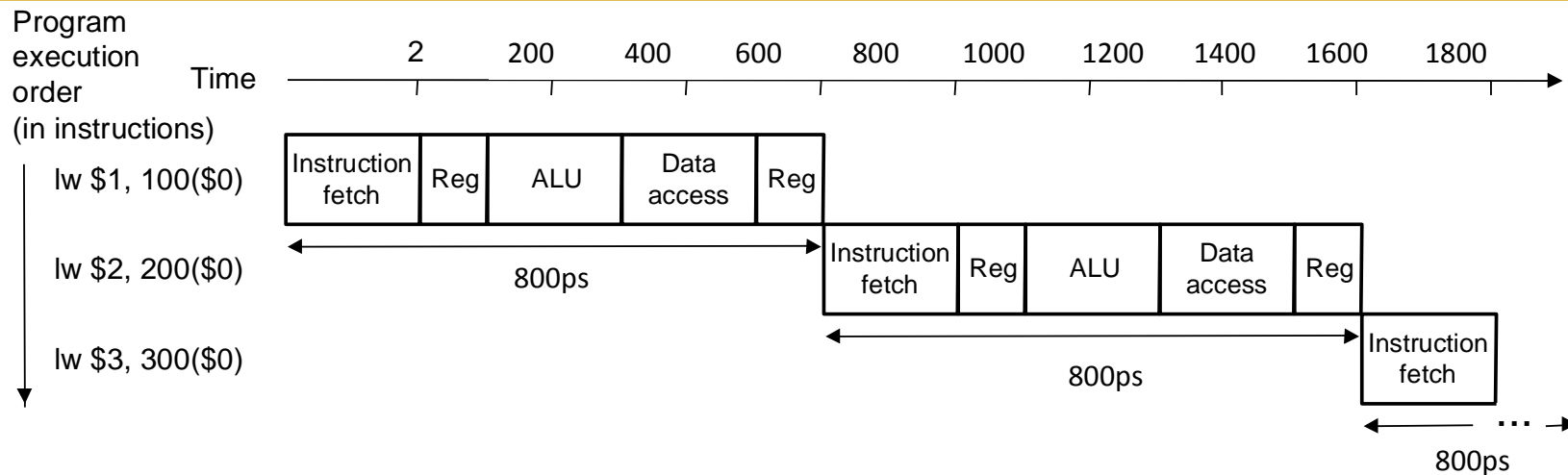
# Dividing Into Stages



Is this the correct partitioning?

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

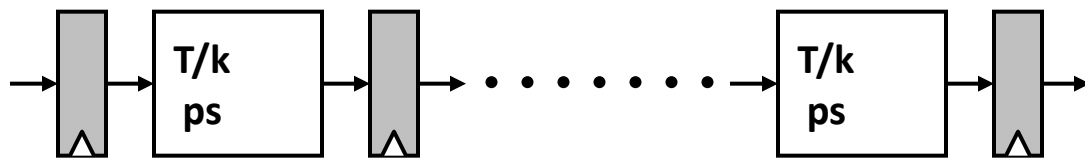
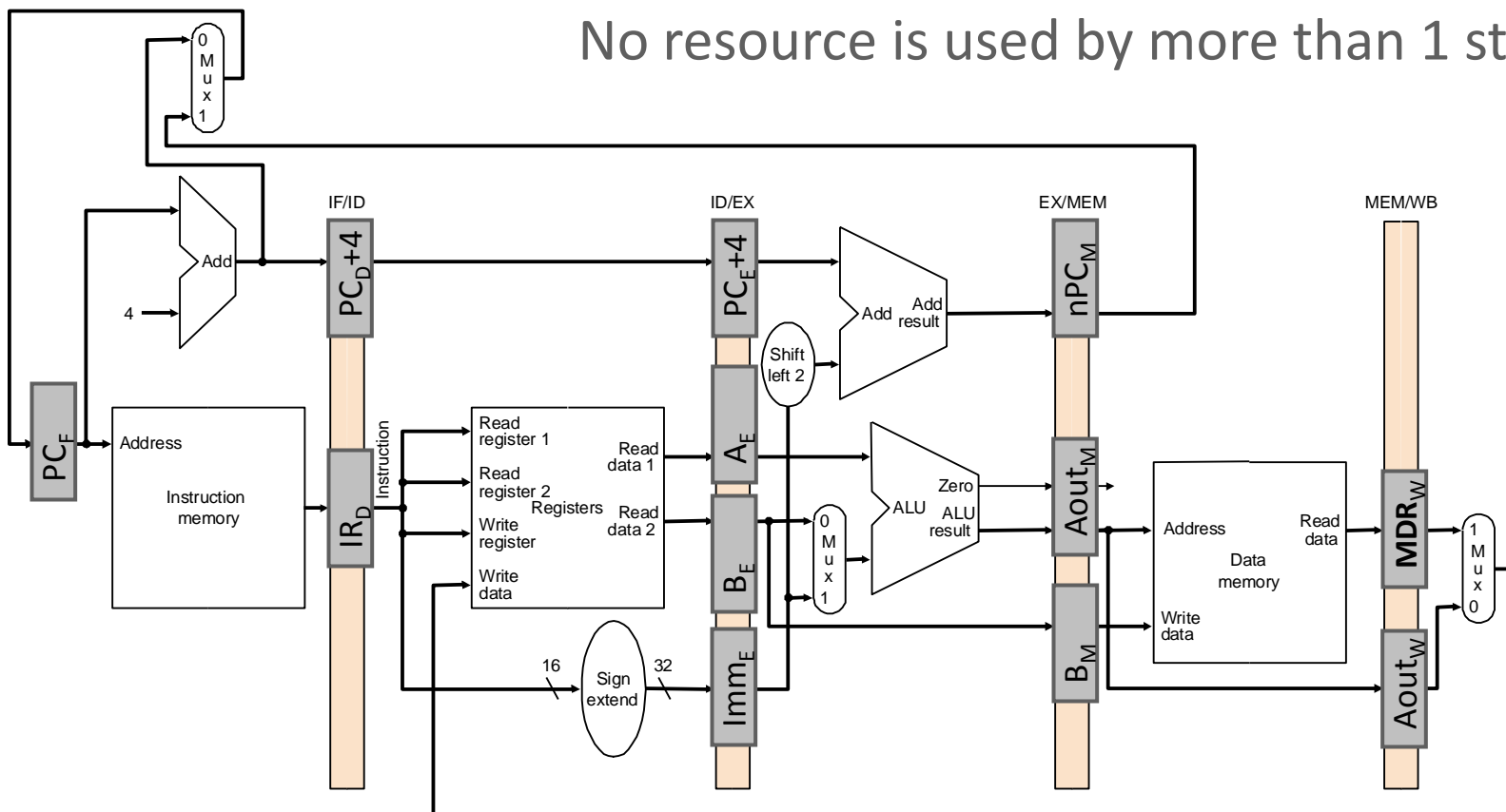
# Instruction Pipeline Throughput



5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

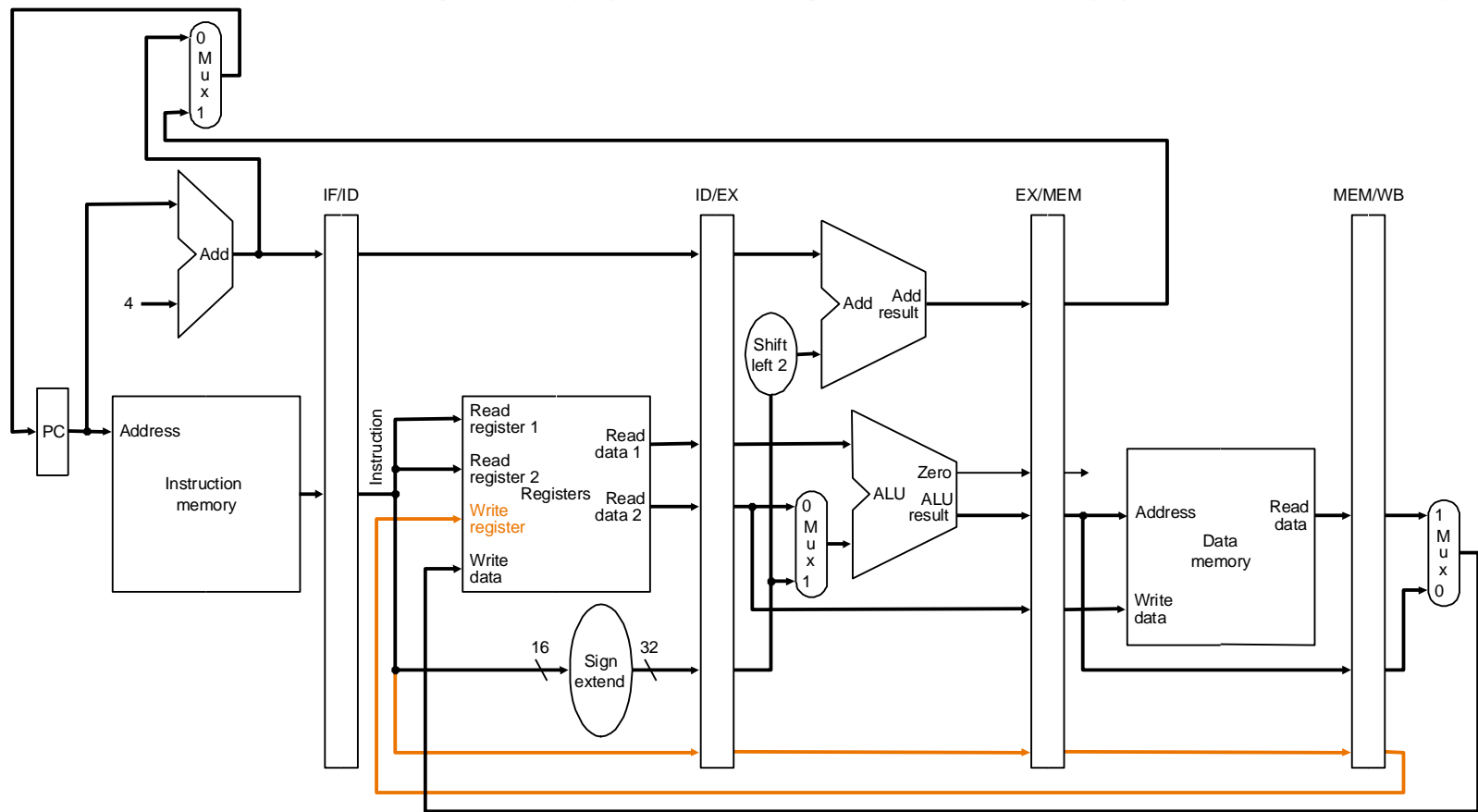
# Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than 1 stage!

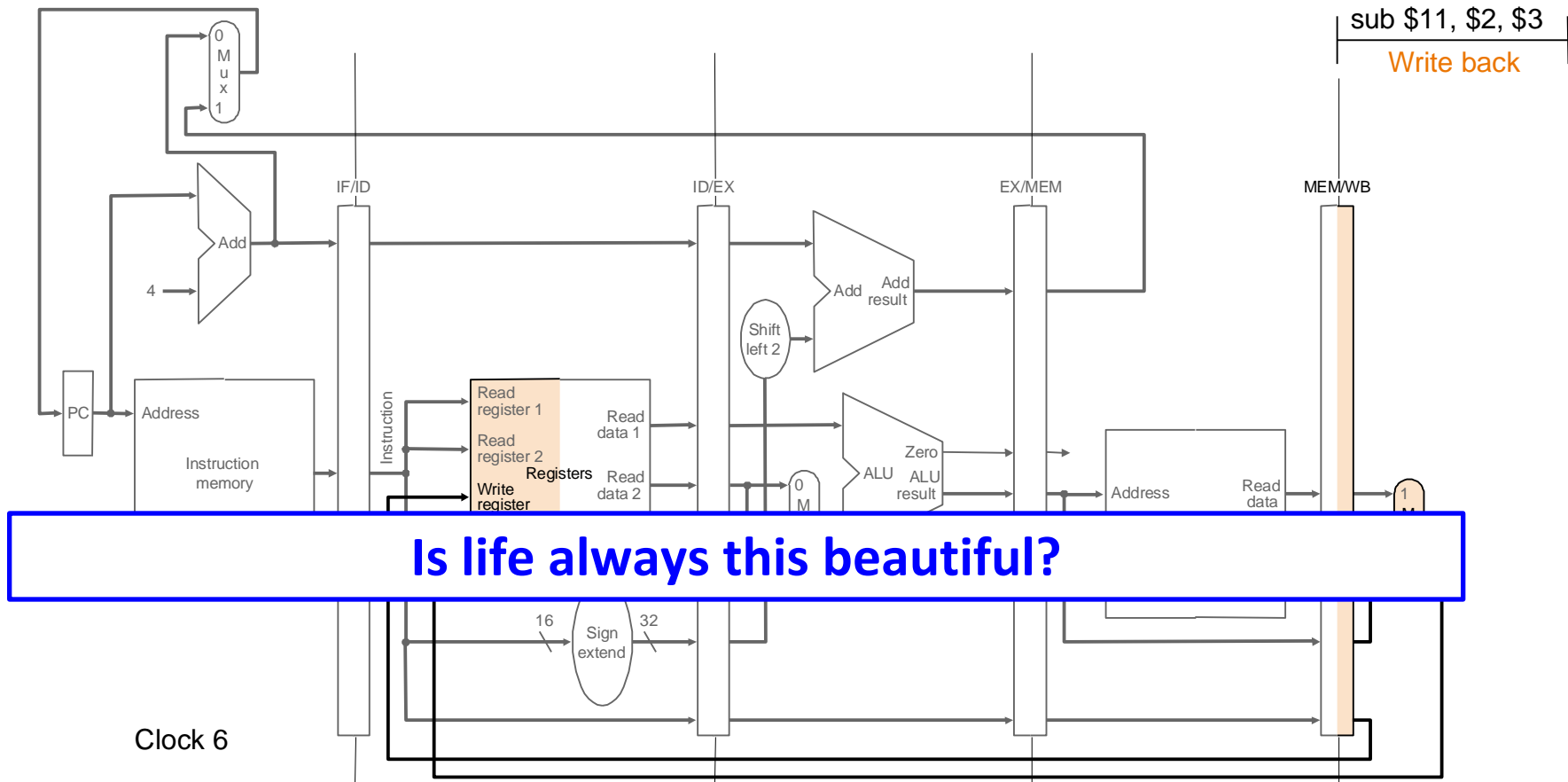


# Pipelined Operation Example

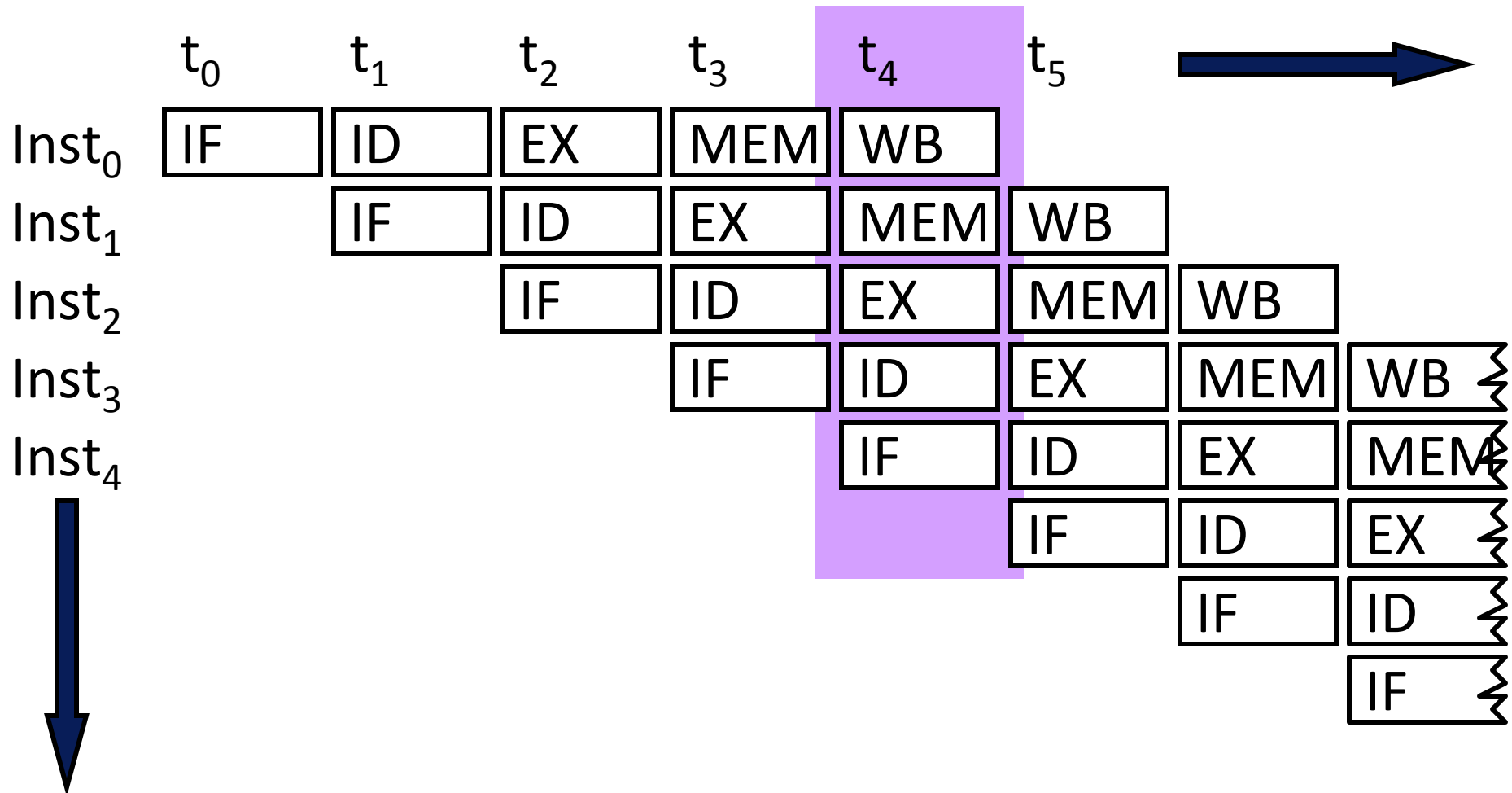
All instruction classes must follow the same path and timing through the pipeline stages. Any performance impact?



# Pipelined Operation Example



# Illustrating Pipeline Operation: Operation View

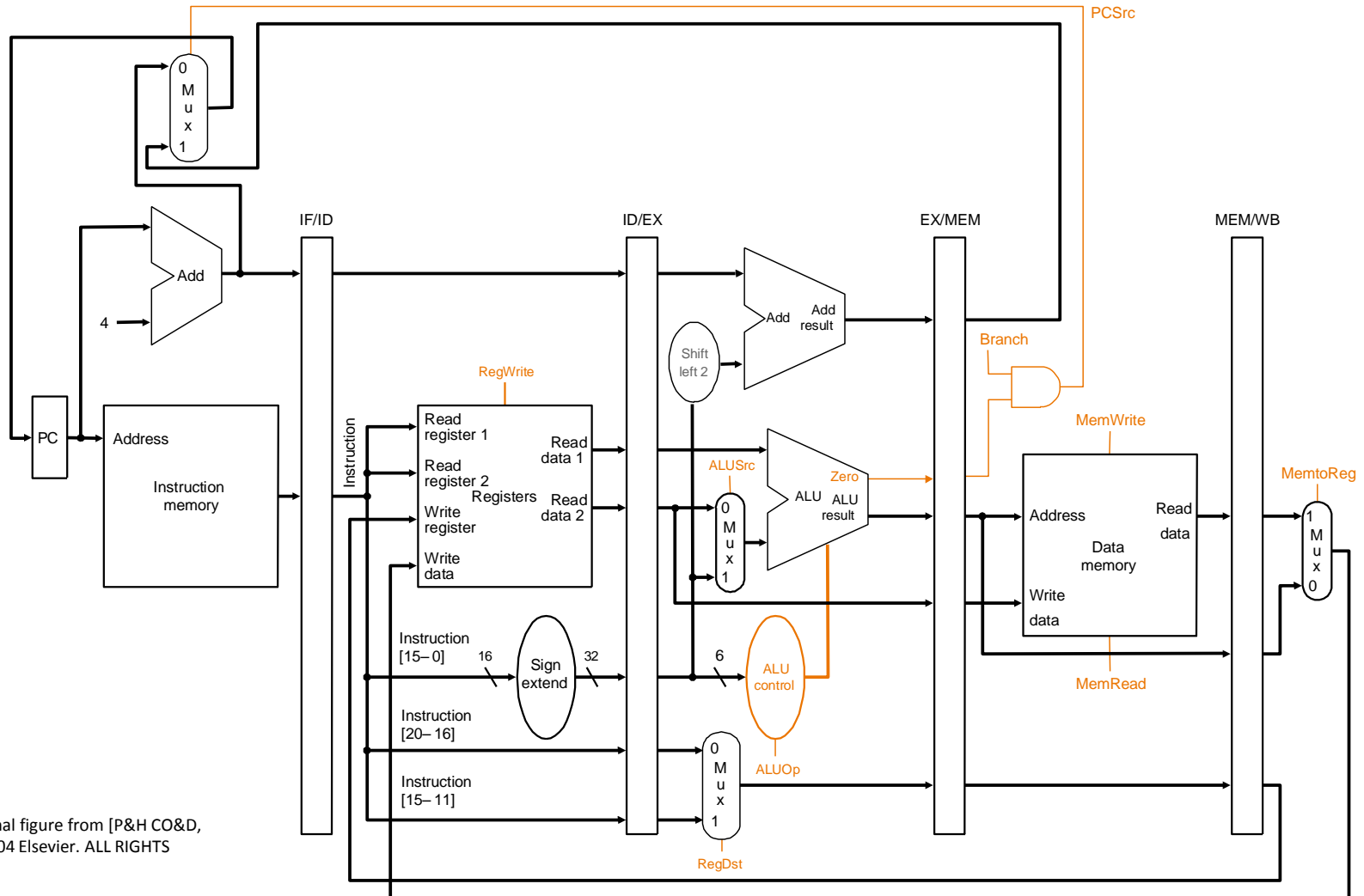


# Illustrating Pipeline Operation: Resource View

---

	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$
IF	$l_0$	$l_1$	$l_2$	$l_3$	$l_4$	$l_5$	$l_6$	$l_7$	$l_8$	$l_9$	$l_{10}$
ID		$l_0$	$l_1$	$l_2$	$l_3$	$l_4$	$l_5$	$l_6$	$l_7$	$l_8$	$l_9$
EX			$l_0$	$l_1$	$l_2$	$l_3$	$l_4$	$l_5$	$l_6$	$l_7$	$l_8$
MEM				$l_0$	$l_1$	$l_2$	$l_3$	$l_4$	$l_5$	$l_6$	$l_7$
WB					$l_0$	$l_1$	$l_2$	$l_3$	$l_4$	$l_5$	$l_6$

# Control Points in a Pipeline



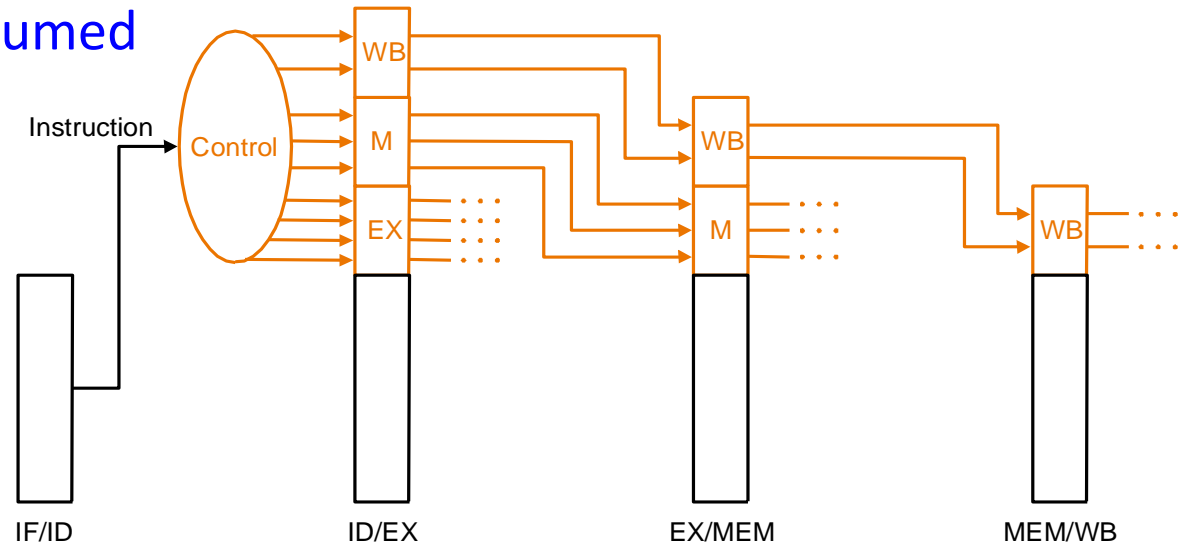
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

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



# Control Signals in a Pipeline

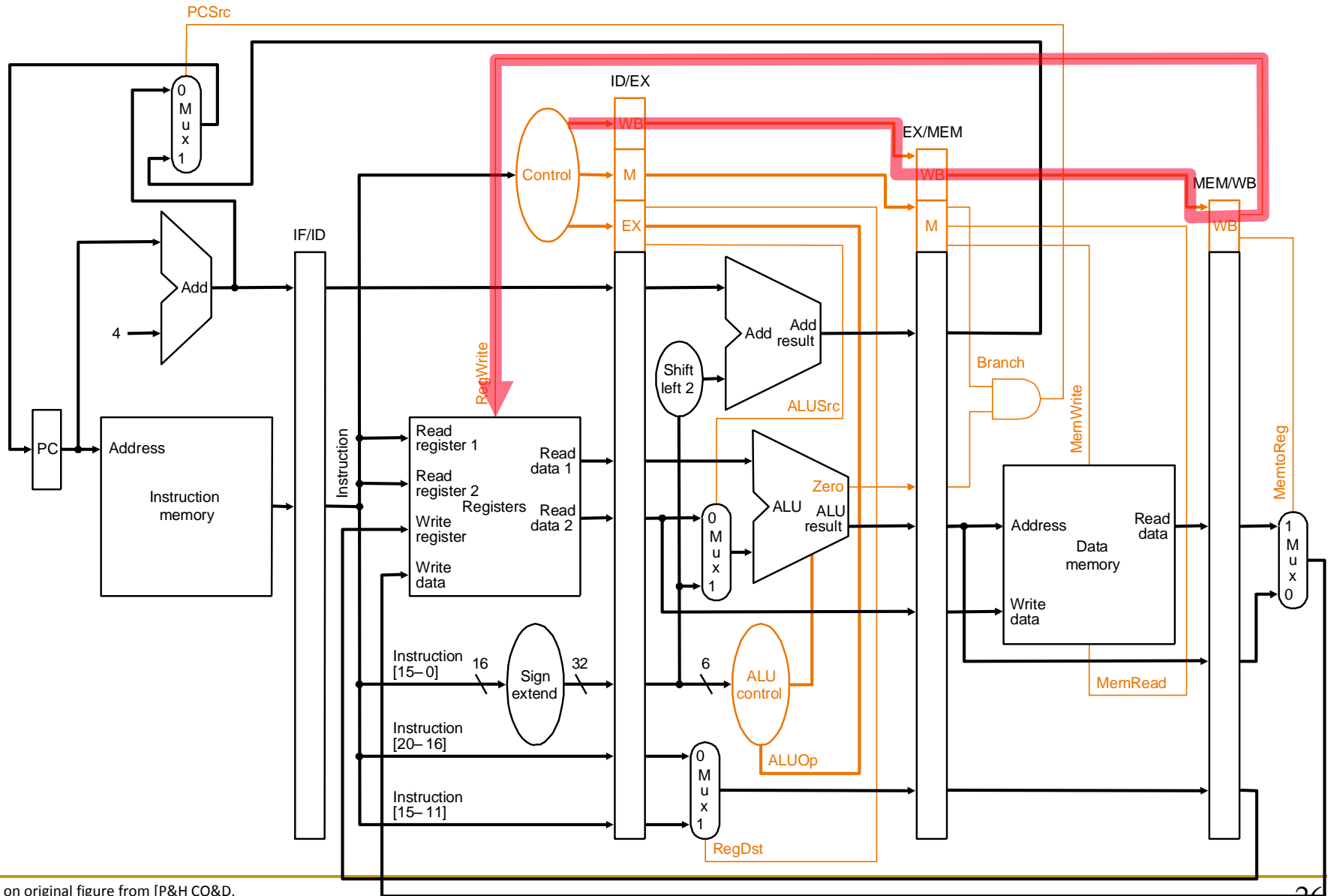
- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
- ⇒ decode once using the same logic as single-cycle and buffer control signals until consumed



- ⇒ or carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

Which one is better?

# Pipelined Control Signals



# An Ideal Pipeline

---

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs
- Repetition of **independent operations**
  - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?

# Instruction Pipeline: Not An Ideal Pipeline

---

- Identical operations ... NOT!

- ⇒ different instructions do not need all stages

- Forcing different instructions to go through the same multi-function pipe
  - external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

- ⇒ difficult to balance the different pipeline stages

- Not all pipeline stages do the same amount of work
  - internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)

- Independent operations ... NOT!

- ⇒ instructions are not independent of each other

- Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly
  - Pipeline is not always moving (it stalls)

# Issues in Pipeline Design

---

- **Balancing work in pipeline stages**
  - How many stages and what is done in each stage
- **Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow**
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- **Handling exceptions, interrupts**
- **Advanced: Improving pipeline throughput**
  - Minimizing stalls

# Causes of Pipeline *Stalls*

---

- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

# Dependences and Their Types

---

- Also called “dependency” or *less desirably* “hazard”
- Dependencies dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately

# Handling Resource Contention

---

- Happens when instructions in two pipeline stages need the same resource
- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures
- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?



# Data Dependences

---


- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)
- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

# Data Dependence Types

---

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR)

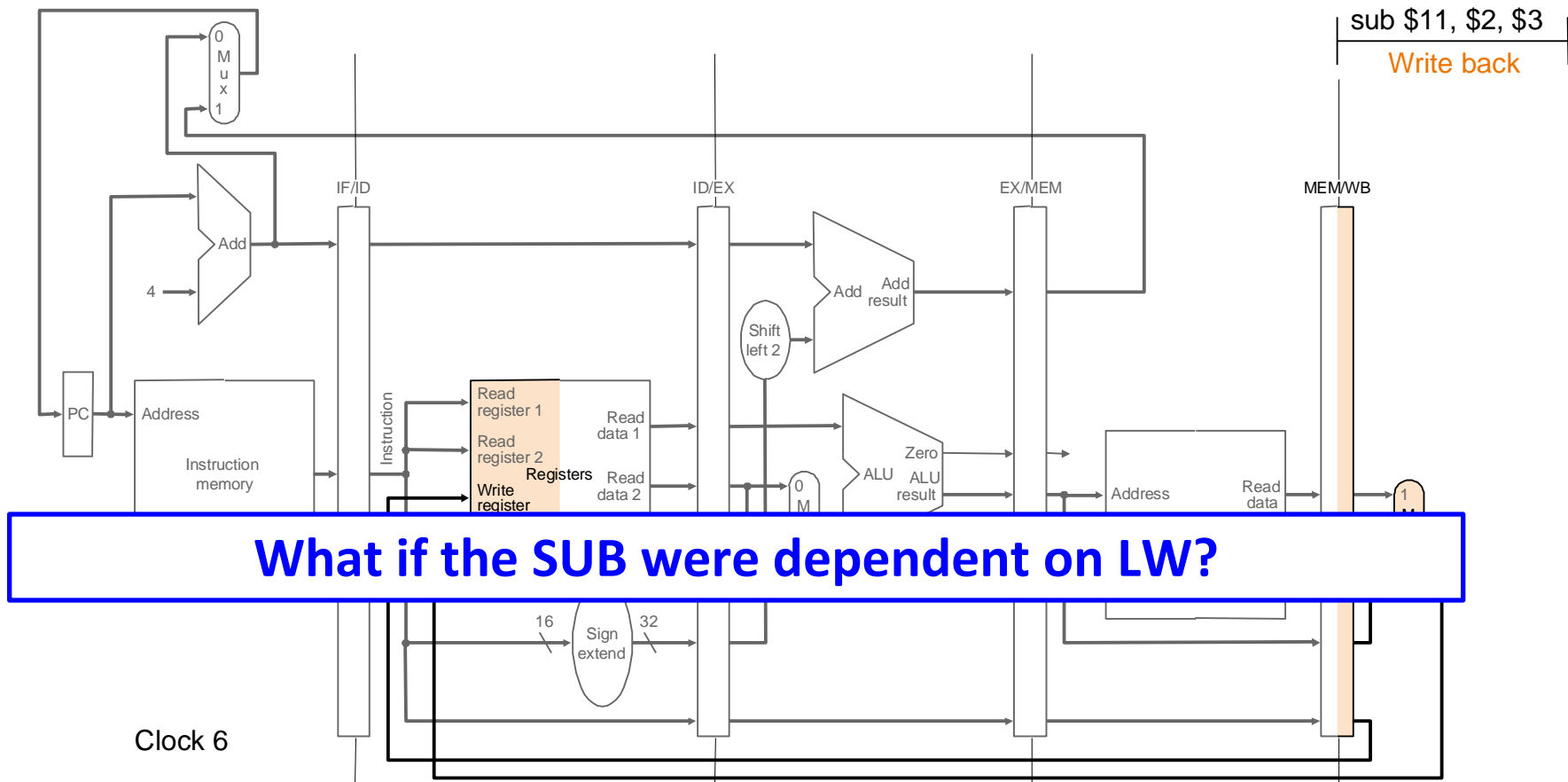
Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write  
(WAW)

# Pipelined Operation Example



# Data Dependence Handling

# Readings for Next Few Lectures

---

- P&H Chapter 4.9-4.11
- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

# How to Handle Data Dependences

---

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - **Detect and wait** until value is available in register file
  - **Detect and forward/bypass** data to dependent instruction
  - **Detect and eliminate** the dependence at the software level
    - No need for the hardware to detect dependence
  - **Predict** the needed value(s), execute “speculatively”, **and verify**
  - **Do something else** (fine-grained multithreading)
    - No need to detect

# Interlocking

---

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- Software based interlocking  
vs.
- Hardware based interlocking
- MIPS acronym?

# Approaches to Dependence Detection (I)

---

## ■ Scoreboarding

- Each register in register file has a Valid bit associated with it
- An instruction that is writing to the register resets the Valid bit
- An instruction in Decode stage checks if all its source and destination registers are Valid
  - Yes: No need to stall... No dependence
  - No: Stall the instruction

## ■ Advantage:

- Simple. 1 bit per register

## ■ Disadvantage:

- Need to stall for all types of dependences, not only flow dep.



# Not Stalling on Anti and Output Dependences

---

- What changes would you make to the scoreboard to enable this?

# Approaches to Dependence Detection (II)

---

## ■ Combinational dependence check logic

- ❑ Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
- ❑ Yes: stall the instruction/pipeline
- ❑ No: no need to stall... no flow dependence

## ■ Advantage:

- ❑ No need to stall on anti and output dependences

## ■ Disadvantage:

- ❑ Logic is more complex than a scoreboard
- ❑ Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

# Once You Detect the Dependence in Hardware

---

- What do you do afterwards?
- Observation: Dependence between two instructions is detected before the communicated data value becomes available
- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: ...

# Data Forwarding/Bypassing

---

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

# A Special Case of Data Dependence

---

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter

# Control Dependence

---

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?