

18-447

# Computer Architecture

## Lecture 7: Pipelining

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 1/30/2015

# Agenda for Today & Next Few Lectures

---

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- **Pipelining**
- **Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...**
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...

# Recap of Last Lecture

---

- Multi-cycle and Microprogrammed Microarchitectures
  - Benefits vs. Design Principles
  - When to Generate Control Signals
  - Microprogrammed Control: uInstruction, uSequencer, Control Store
  - LC-3b State Machine, Datapath, Control Structure
  - An Exercise in Microprogramming
  - Variable Latency Memory, Alignment, Memory Mapped I/O, ...
- Microprogramming
  - Power of abstraction (for the HW designer)
  - Advantages of uProgrammed Control
  - Update of Machine Behavior

# Review: A Simple LC-3b Control and Datapath

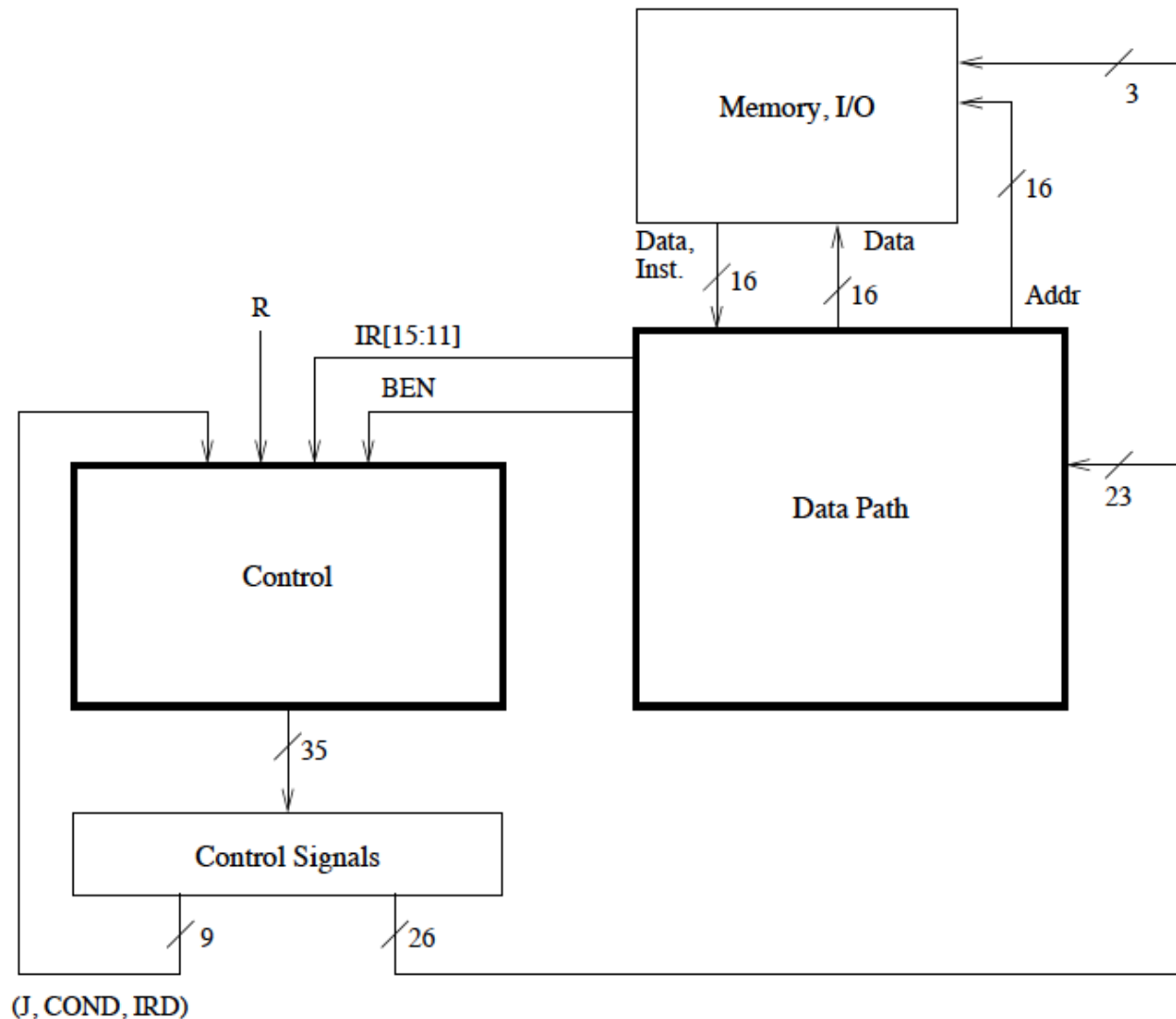
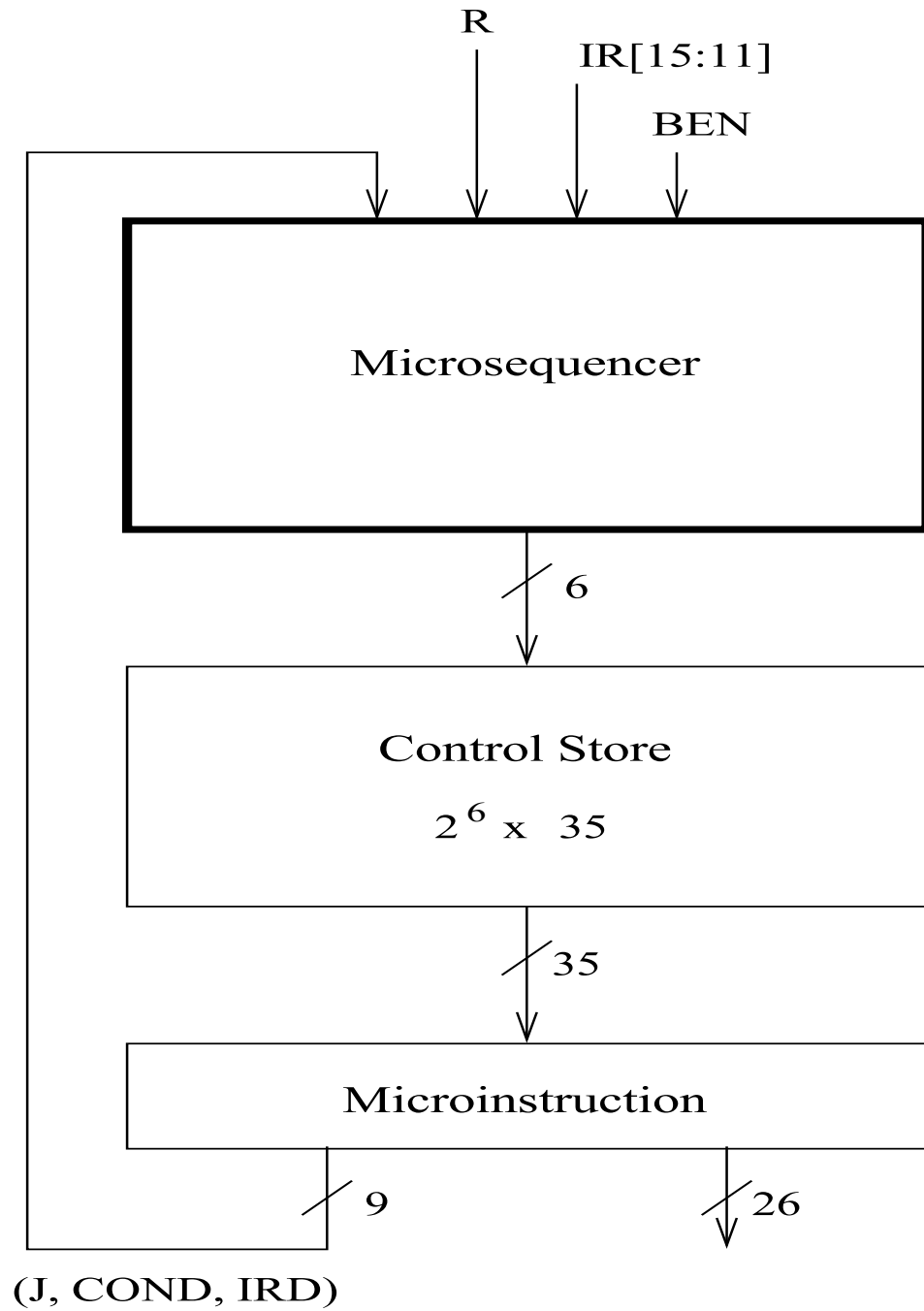


Figure C.1: Microarchitecture of the LC-3b, major components



Simple Design  
of the Control Structure



# Review: The Power of Abstraction

---

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: **microprogramming**
- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
  - **The sequence of microinstructions** needed to implement the desired operation
  - **The ability for the control logic to correctly sequence** through the microinstructions
  - **Any additional datapath elements and control signals** needed (no need if the operation can be “translated” into existing control signals)

# Review: Advantages of Microprogrammed Control

---

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a **user-invisible ISA (u-ISA)**
- Enables easy extensibility of the ISA
  - **Can support a new instruction by changing the microcode**
  - Can support complex instructions as a sequence of simple microinstructions
- Enables update of machine behavior
  - **A buggy implementation of an instruction can be fixed by changing the microcode in the field**



# Wrap Up Microprogrammed Control

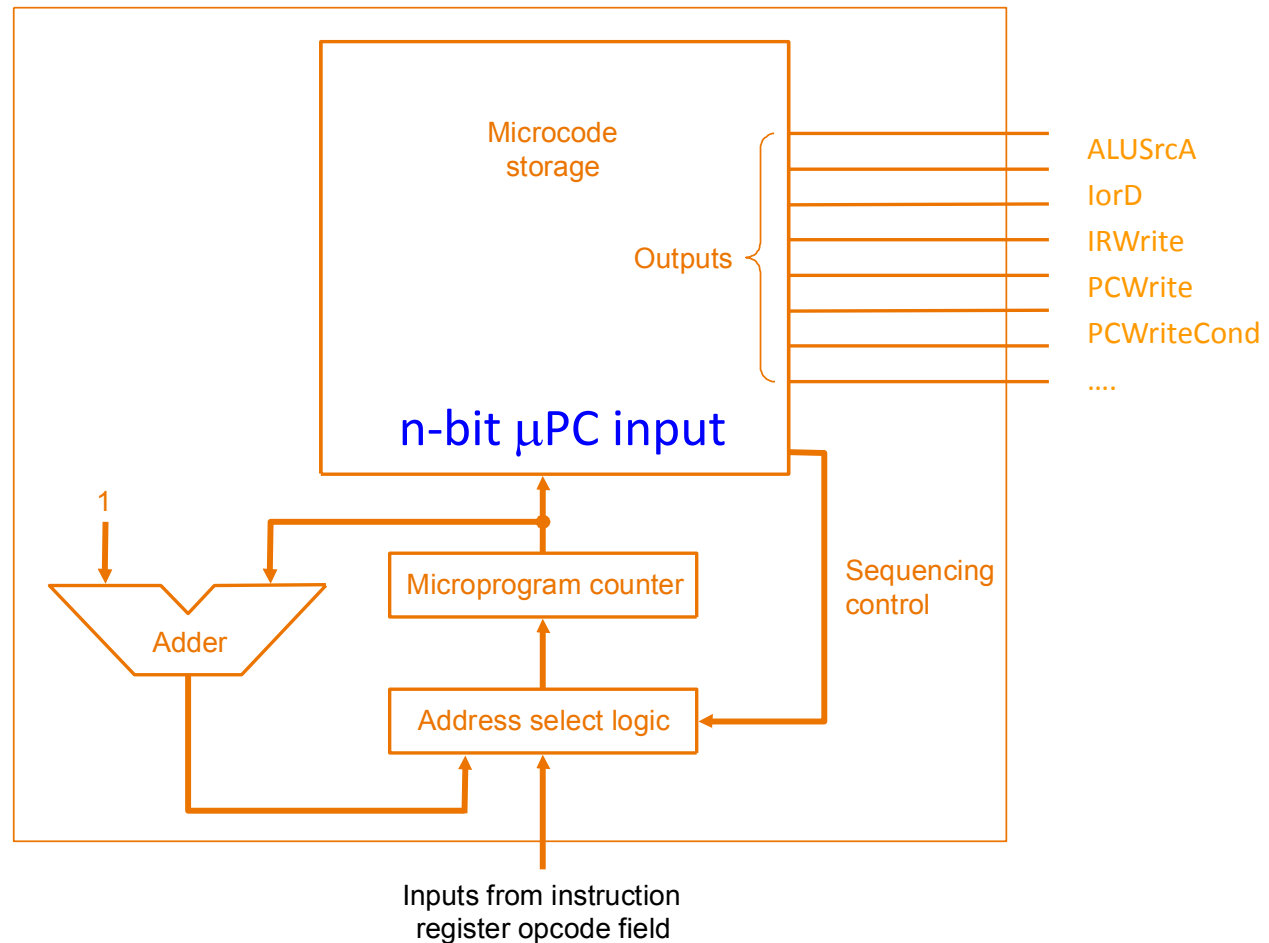
---

- Horizontal vs. Vertical Microcode
- Nanocode vs. Microcode vs. Millicode
- Microprogrammed MIPS: An Example

# Horizontal Microcode

- A single control store provides the control signals

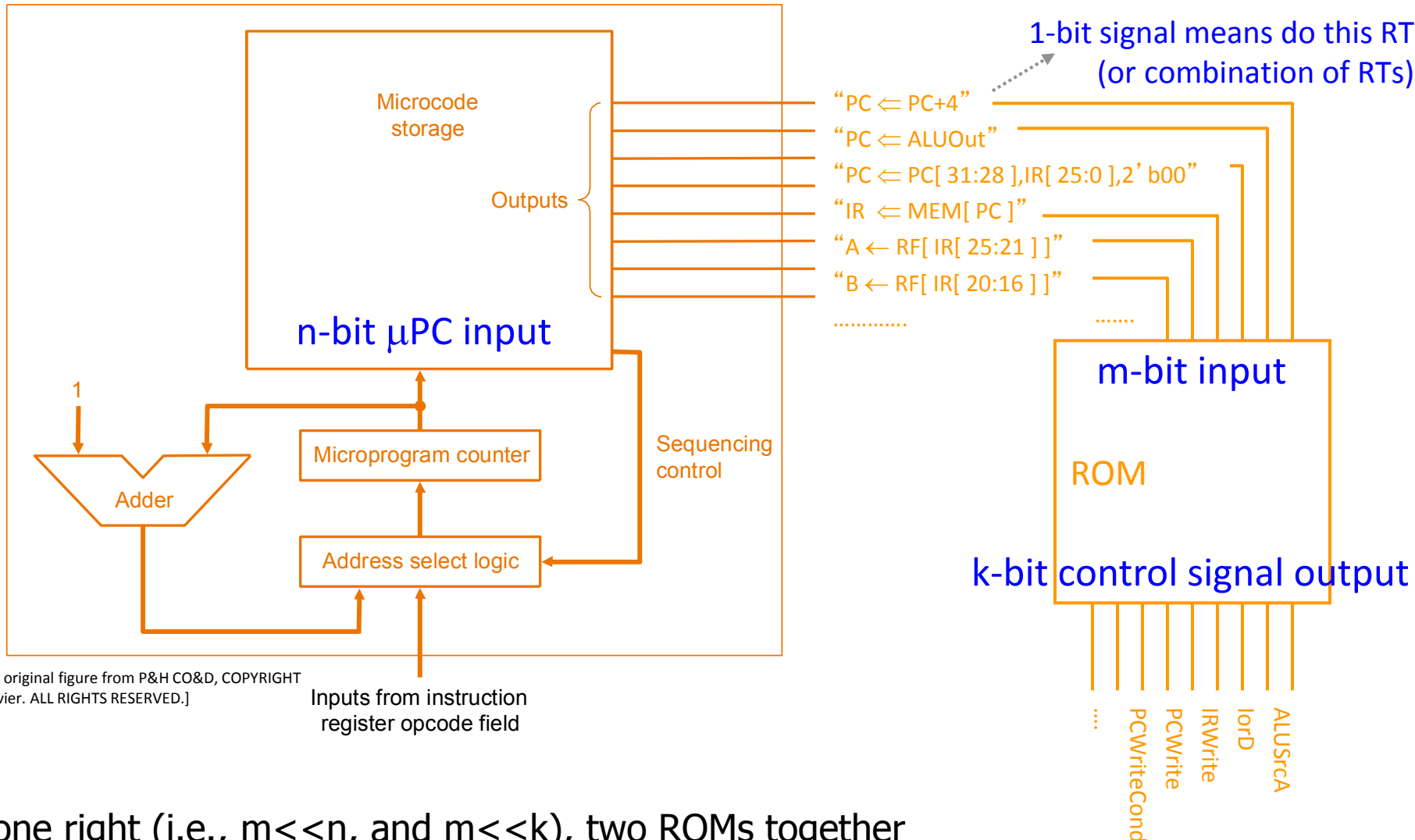
MIPS design  
From  
P&H, Appendix D



k-bit control signal output

# Vertical Microcode

- Two-level control store: the first specifies abstract operations



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

If done right (i.e.,  $m \ll n$ , and  $m \ll k$ ), two ROMs together ( $2^n \times m + 2^m \times k$  bit) should be smaller than horizontal microcode ROM ( $2^n \times k$  bit)

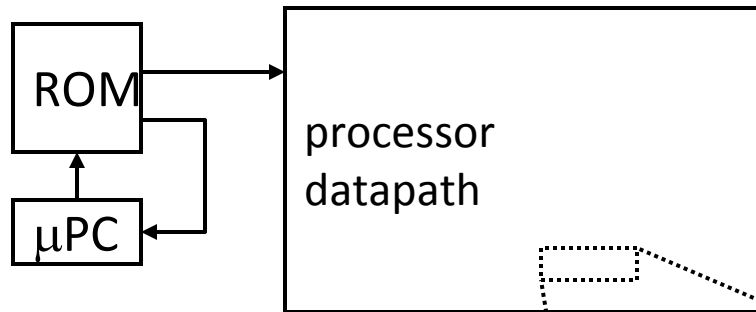
# Nanocode and Millicode

---

- **Nanocode**: a level below traditional microcode
  - microprogrammed control for sub-systems (e.g., a complicated floating-point module) that acts as a slave in a microcontrolled datapath
- **Millicode**: a level above traditional microcode
  - ISA-level subroutines that can be called by the microcontroller to handle complicated operations and system functions
  - E.g., Heller and Farrell, “**Millicode in an IBM zSeries processor**,” IBM JR&D, May/Jul 2004.
- In both cases, we avoid complicating the main u-controller
- You can think of these as “**microcode**” at different levels of abstraction

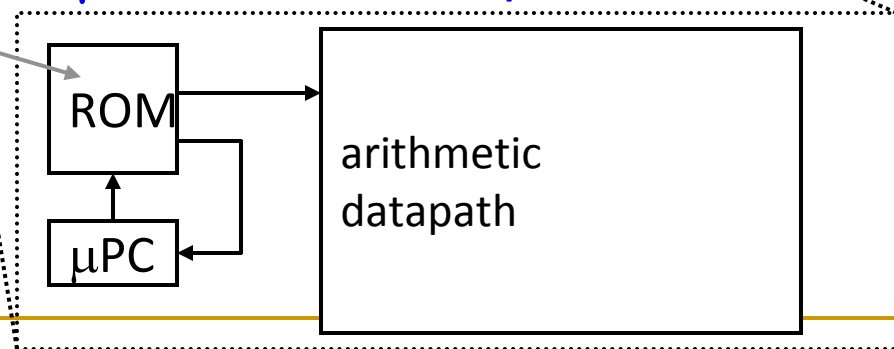
# Nanocode Concept Illustrated

a “ $\mu$ coded” processor implementation



We refer to this as “nanocode” when a  $\mu$ coded subsystem is embedded in a  $\mu$ coded system

a “ $\mu$ coded” FPU implementation



# Microcoded Multi-Cycle MIPS Design

---

- Any ISA can be implemented with a microprogrammed microarchitecture
- P&H, Appendix D: Microprogrammed MIPS design
- We will not cover this in class
- However, you can do an extra credit assignment for Lab 2

# Microcoded Multi-Cycle MIPS Design

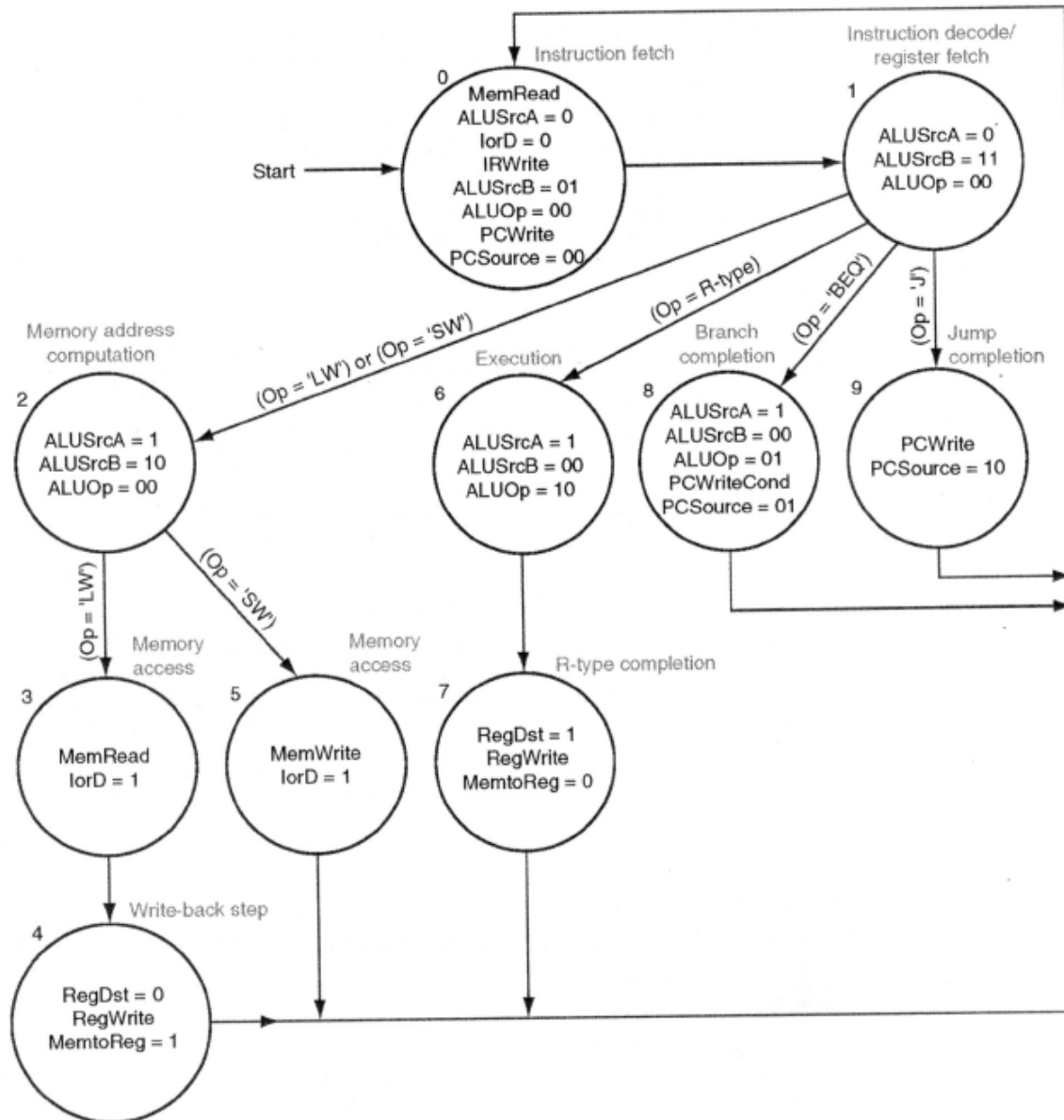
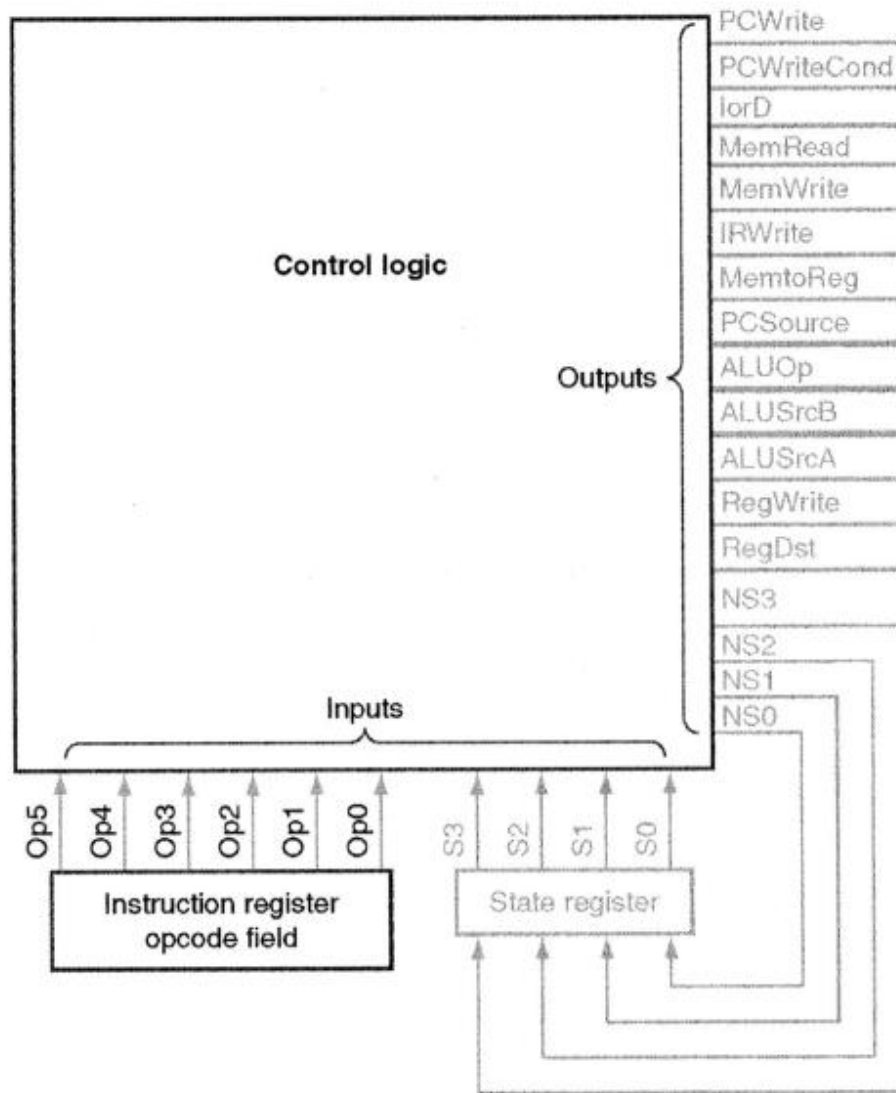


FIGURE D.3.1 The finite-state diagram for multicycle control.

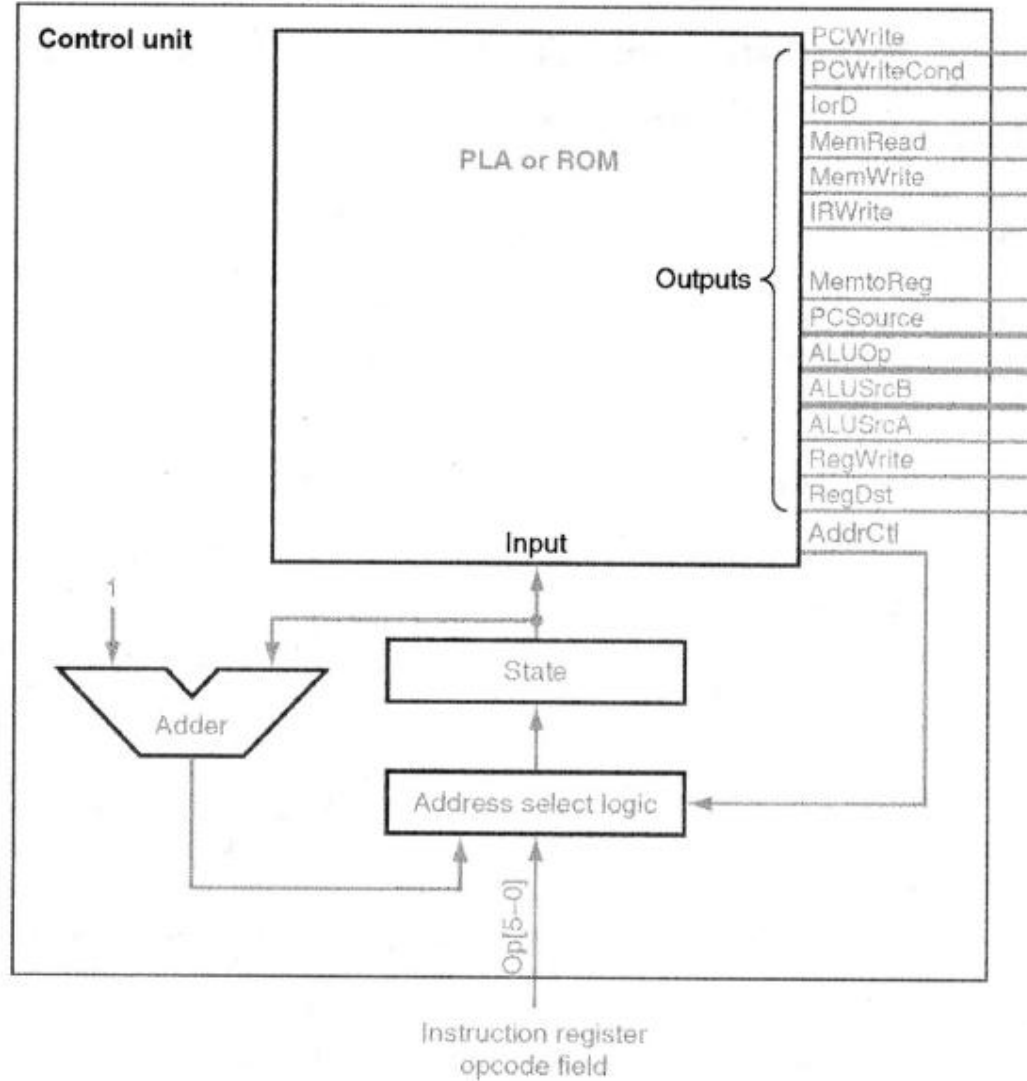
# Control Logic for MIPS FSM



**FIGURE D.3.2** The control unit for MIPS will consist of some control logic and a register to hold the state. The state register is written at the active clock edge and is stable during the clock cycle.



# Microprogrammed Control for MIPS FSM



**FIGURE D.4.1 The control unit using an explicit counter to compute the next state.** In this control unit, the next state is computed using a counter (at least in some states). By comparison, Figure D.3.2 encodes the next state in the control logic for every state. In this control unit, the signals labeled *AddrCtl* control how the next state is determined.

# Multi-Cycle vs. Single-Cycle uArch

---

- Advantages
- Disadvantages
- You should be very familiar with this right now

# Microprogrammed vs. Hardwired Control

---

- Advantages
- Disadvantages
- You should be very familiar with this right now

# Can We Do Better?

---

- 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: **More concurrency → Higher instruction throughput** (i.e., 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

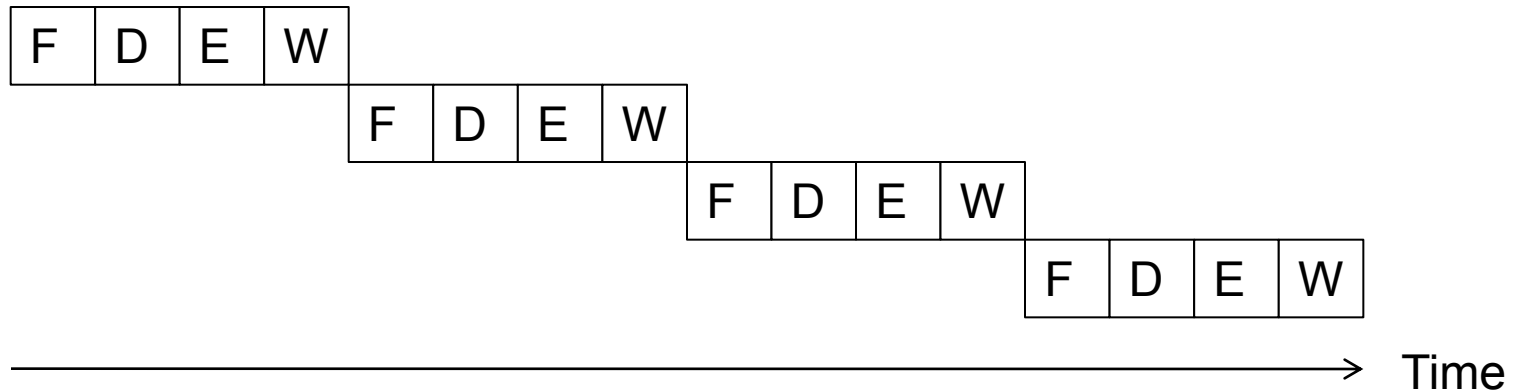
# Pipelining: Basic Idea

---

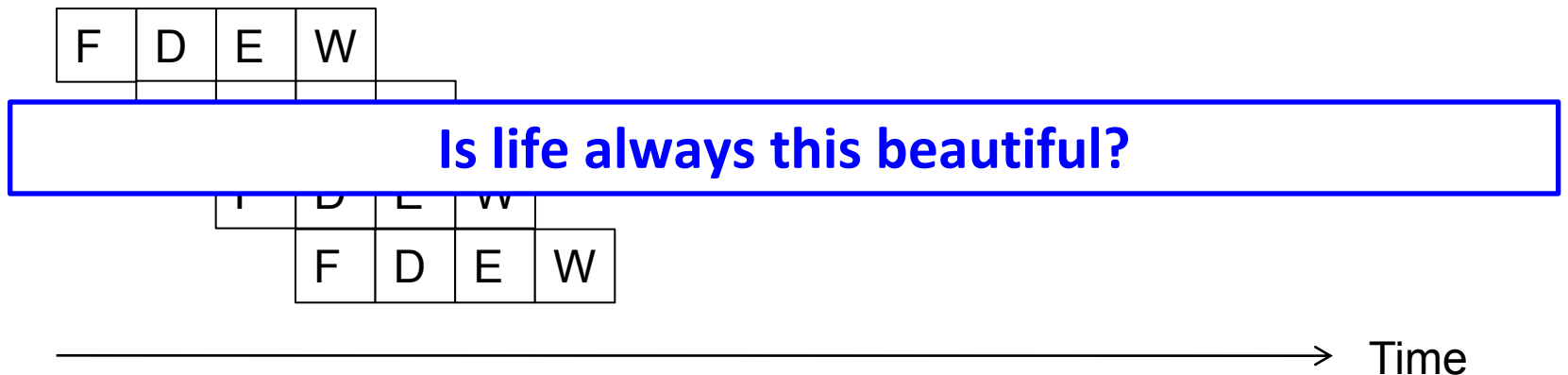
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each 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

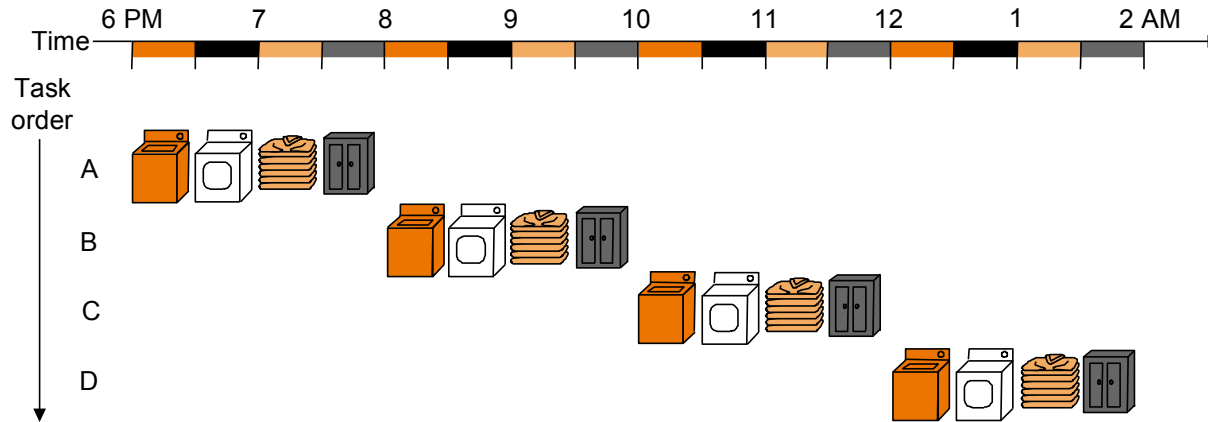


- Pipelined: 4 cycles per 4 instructions (steady state)



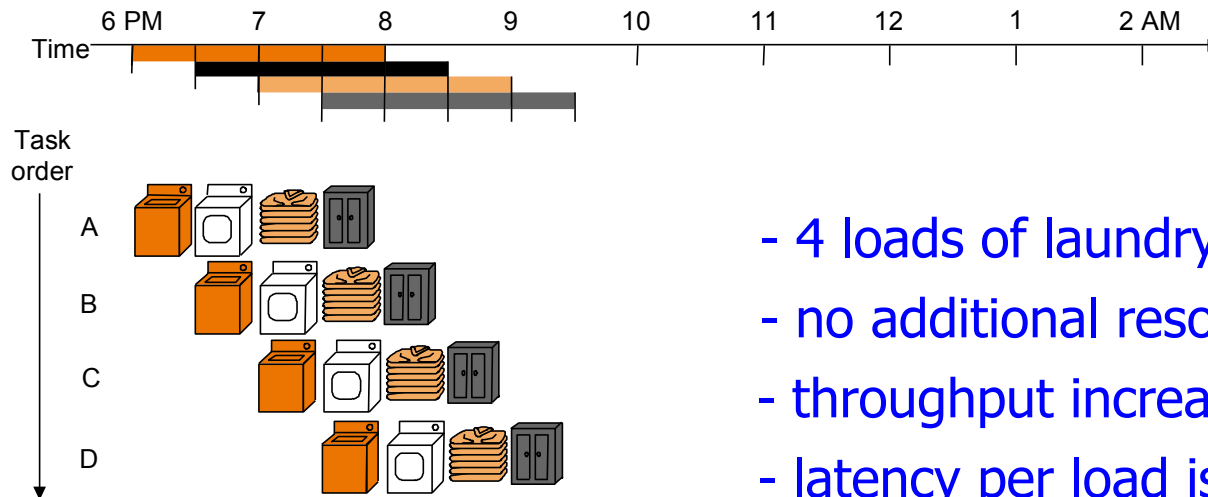
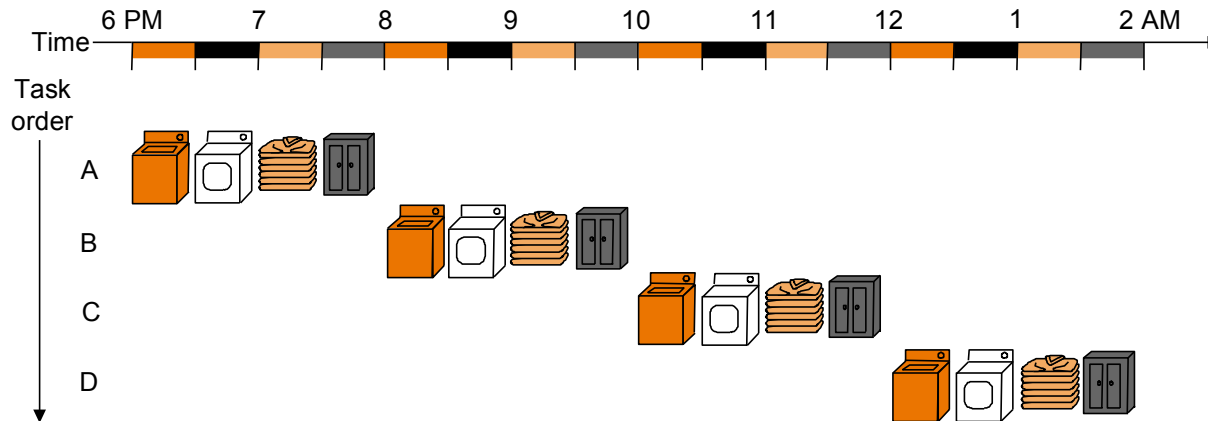


# The Laundry Analogy



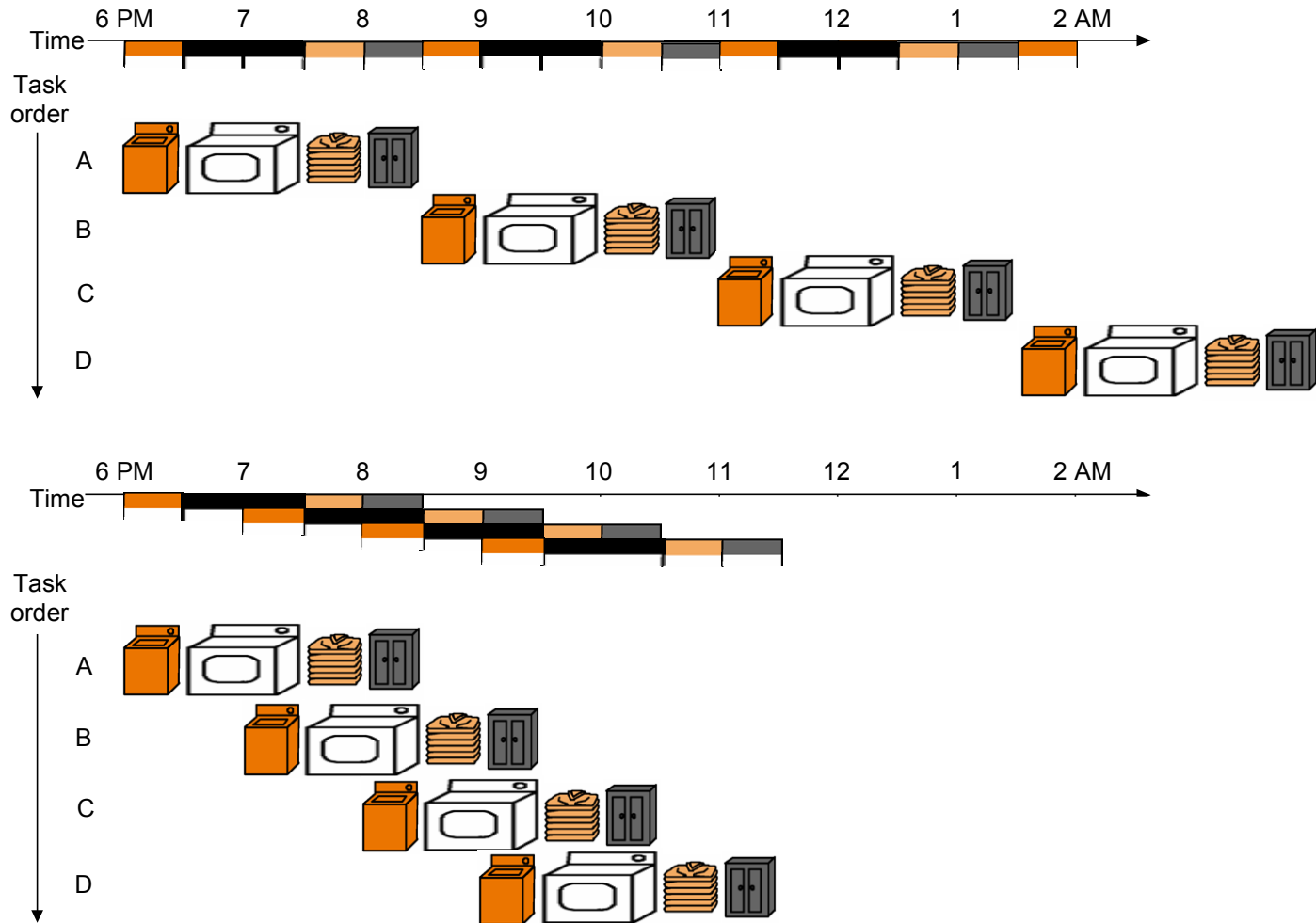
- “place one dirty load of clothes in the washer”
  - “when the washer is finished, place the wet load in the dryer”
  - “when the dryer is finished, take out the dry load and fold”
  - “when folding is finished, ask your roommate (??) to put the clothes away”
- steps to do a load are sequentially dependent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

# Pipelining Multiple Loads of Laundry



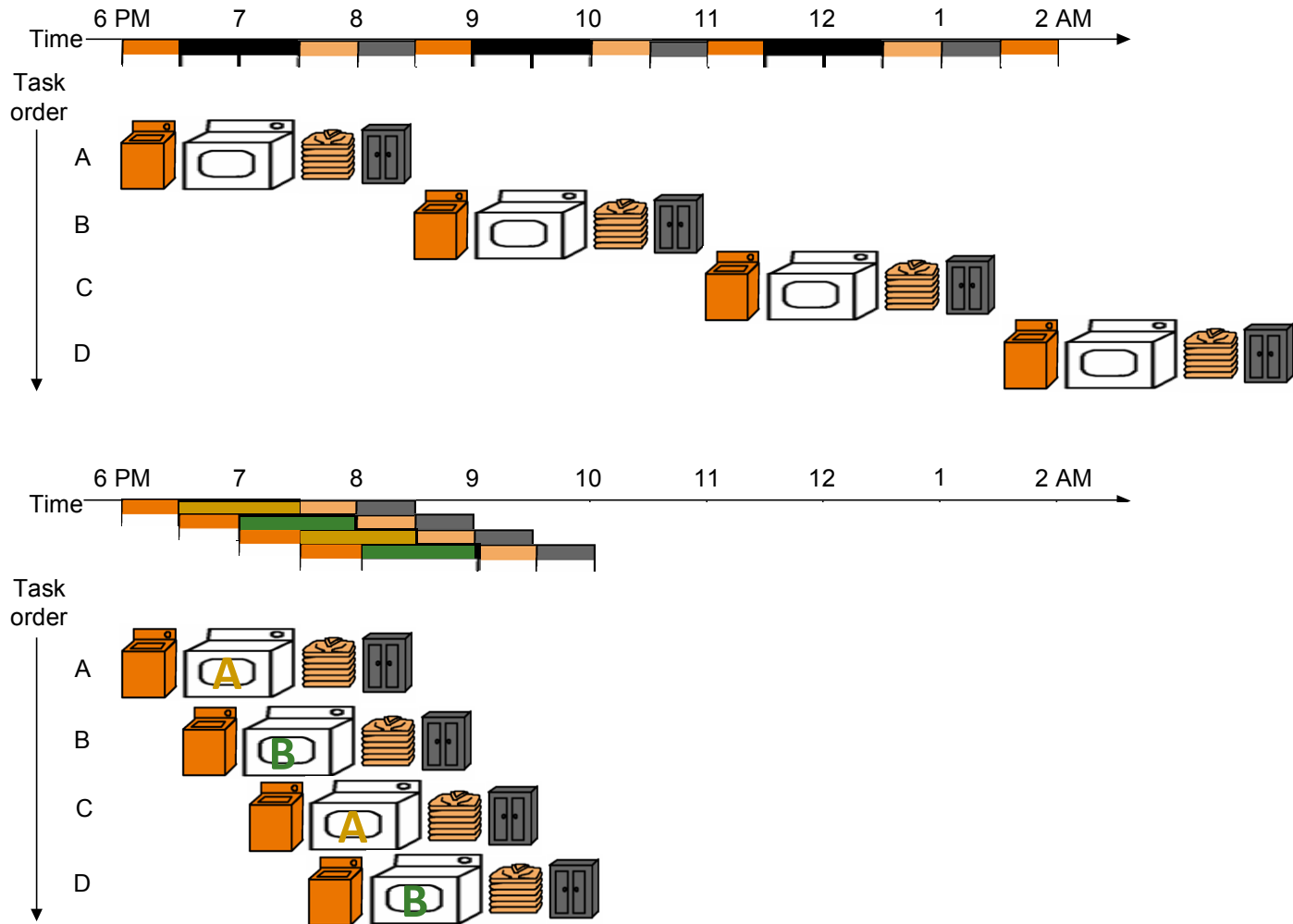
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

# 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

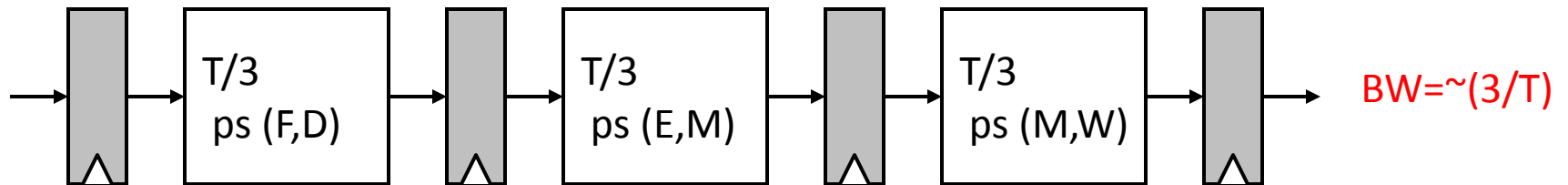
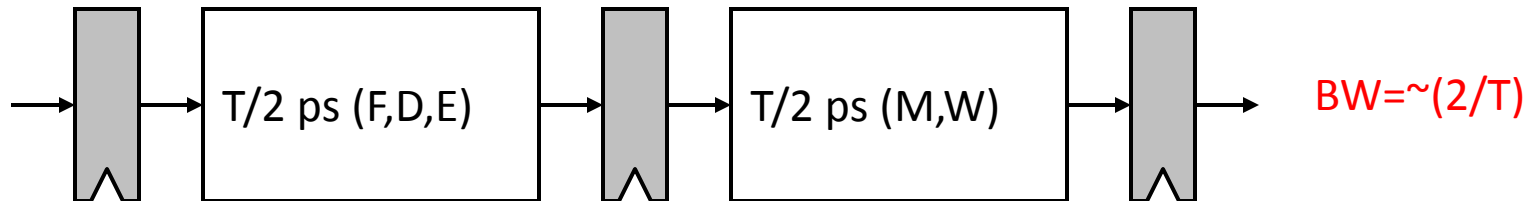
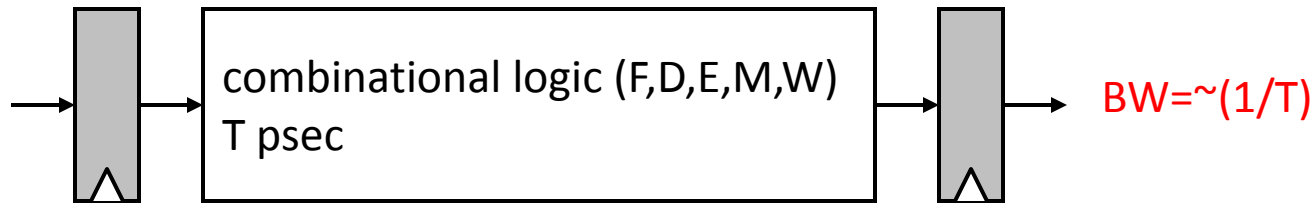
# An Ideal Pipeline

---

- 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 (e.g., all laundry loads go through the same steps)
- 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  $T$

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

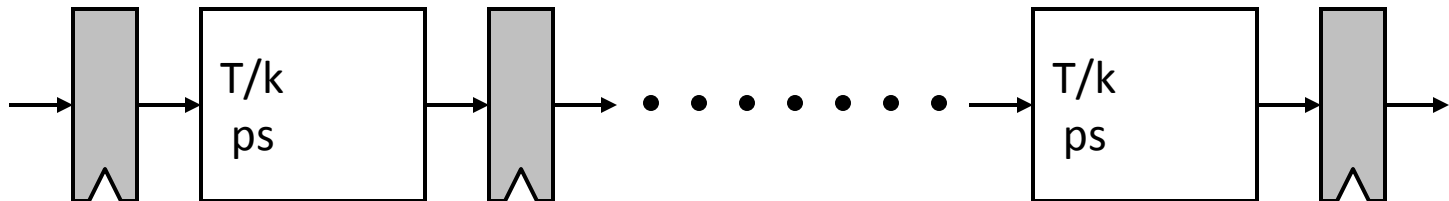


- k-stage pipelined version

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

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

**Latch delay reduces throughput  
(switching overhead b/w stages)**



# More Realistic Pipeline: Cost

---

- Nonpipelined version with combinational cost  $G$

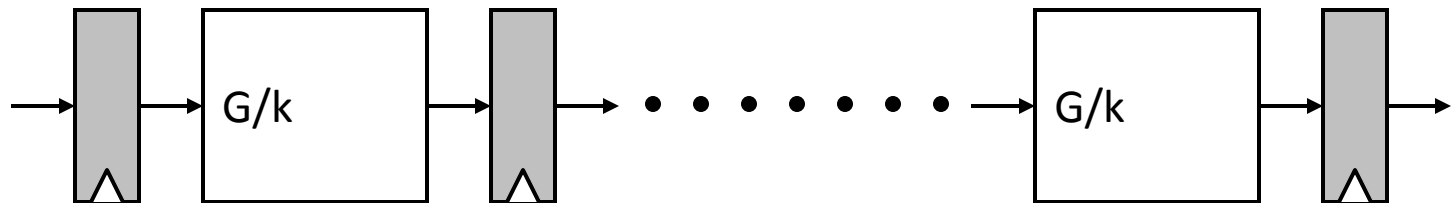
$\text{Cost} = G + L$  where  $L$  = latch cost



- k-stage pipelined version

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

**Latches increase hardware cost**

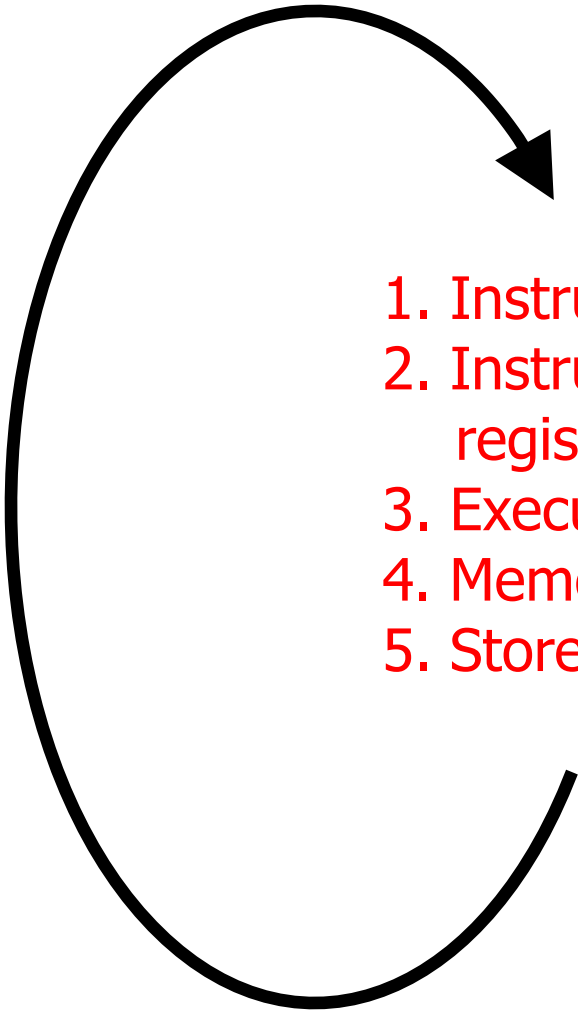




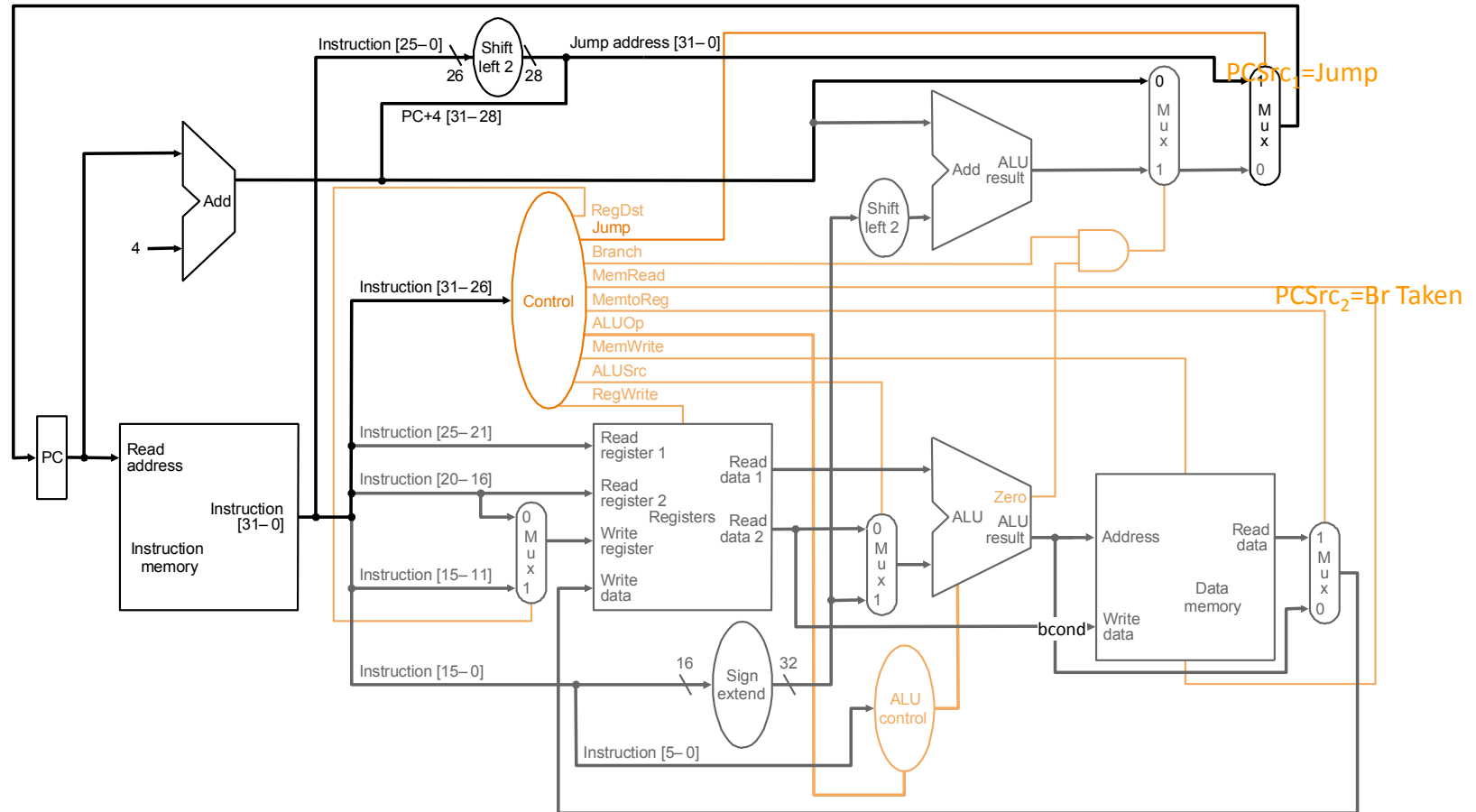
# Pipelining Instruction Processing

# 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)

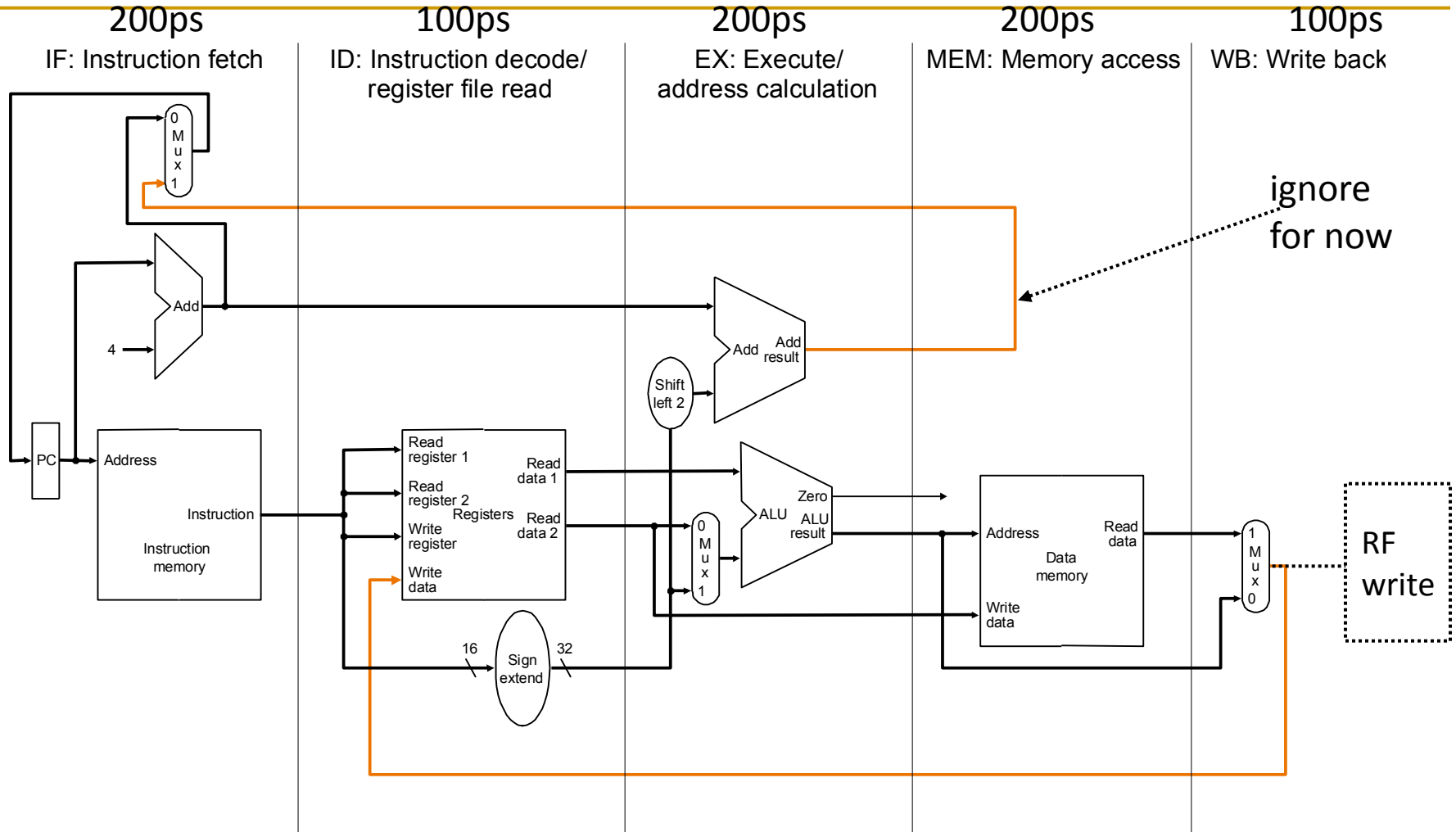
# Remember the Single-Cycle Uarch



ALU operation



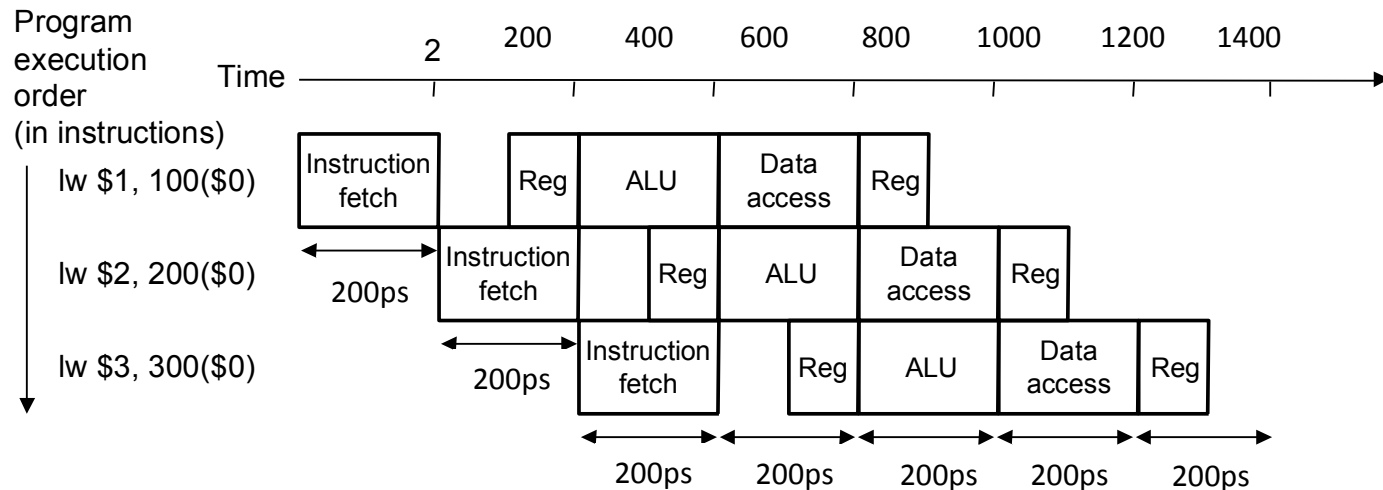
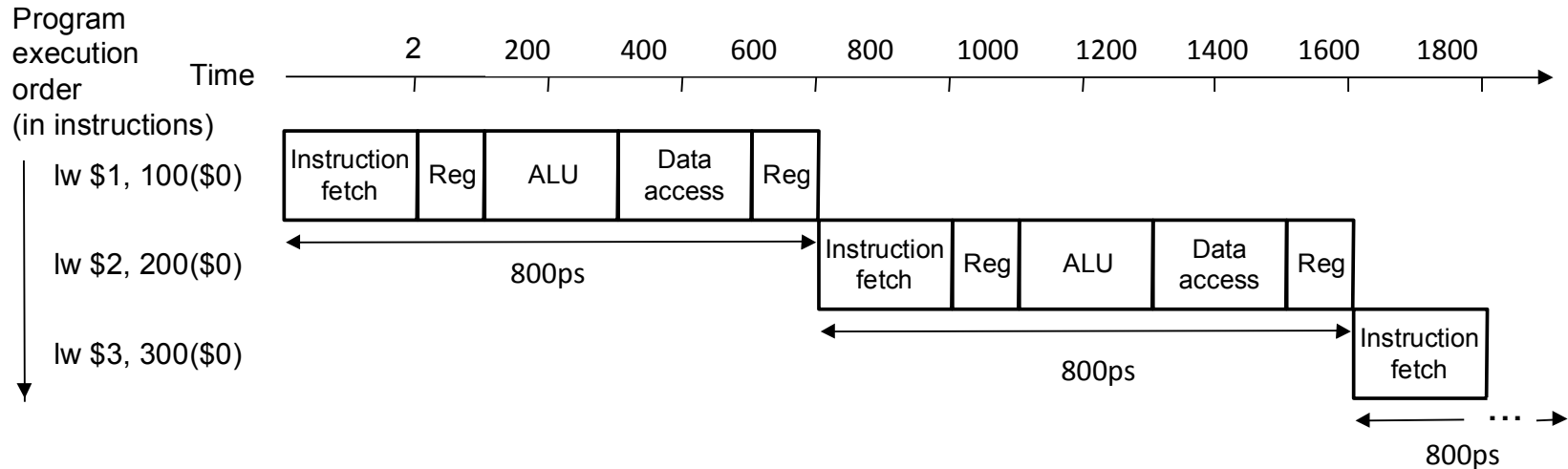
# Dividing Into Stages



Is this the correct partitioning?

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

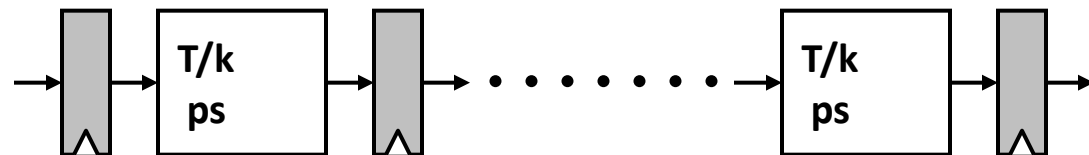
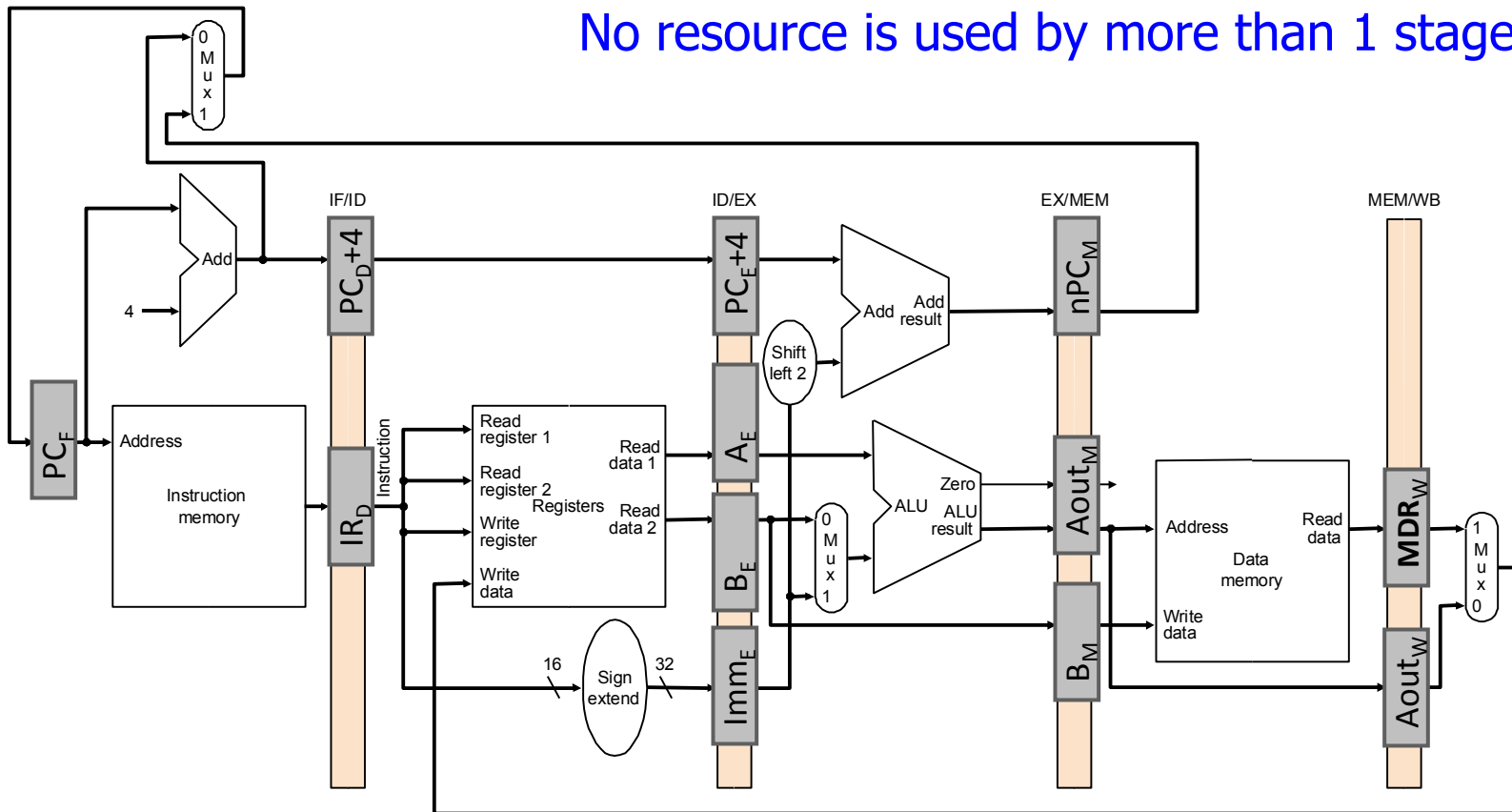
# Instruction Pipeline Throughput



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

# Enabling Pipelined Processing: Pipeline Registers

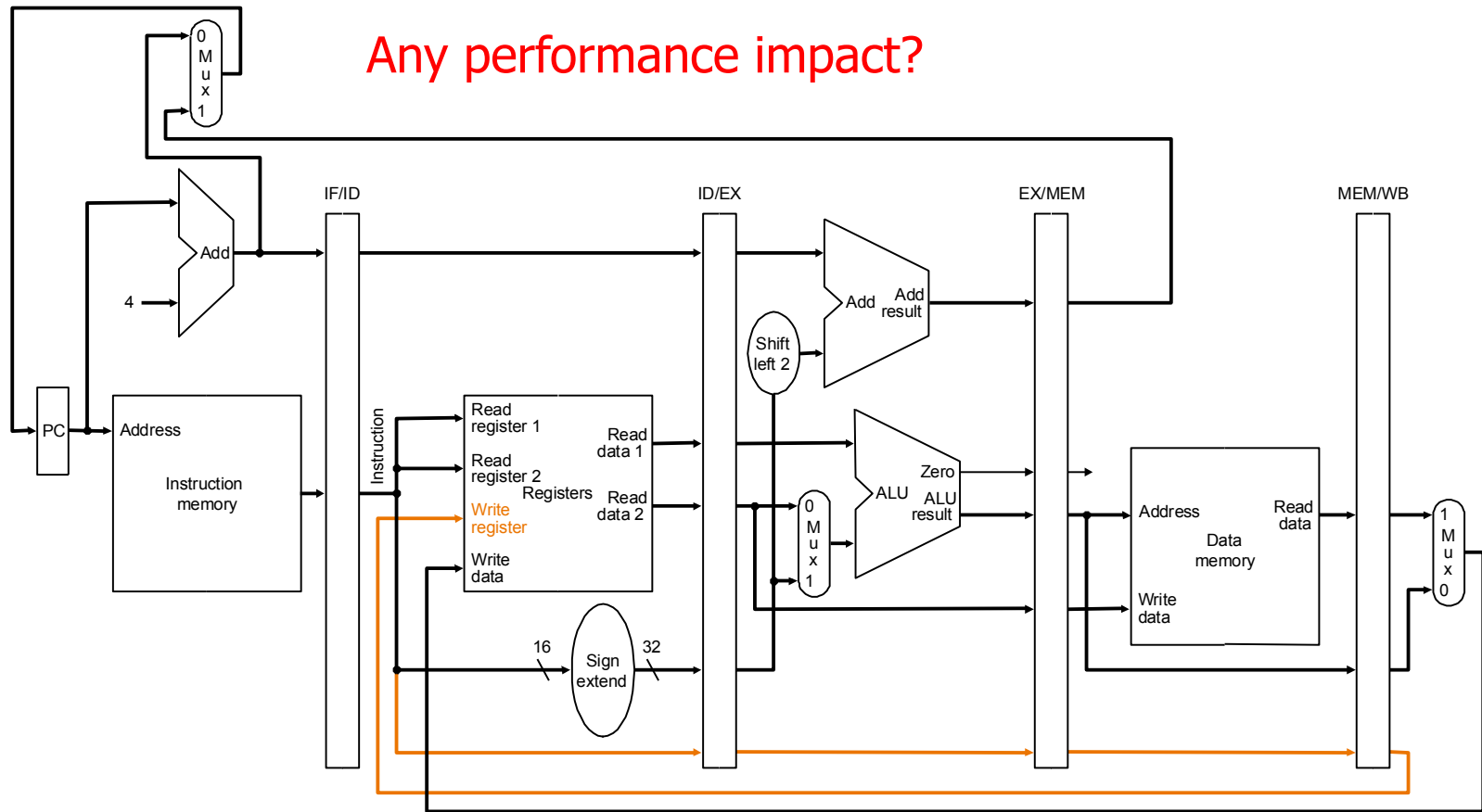
No resource is used by more than 1 stage!



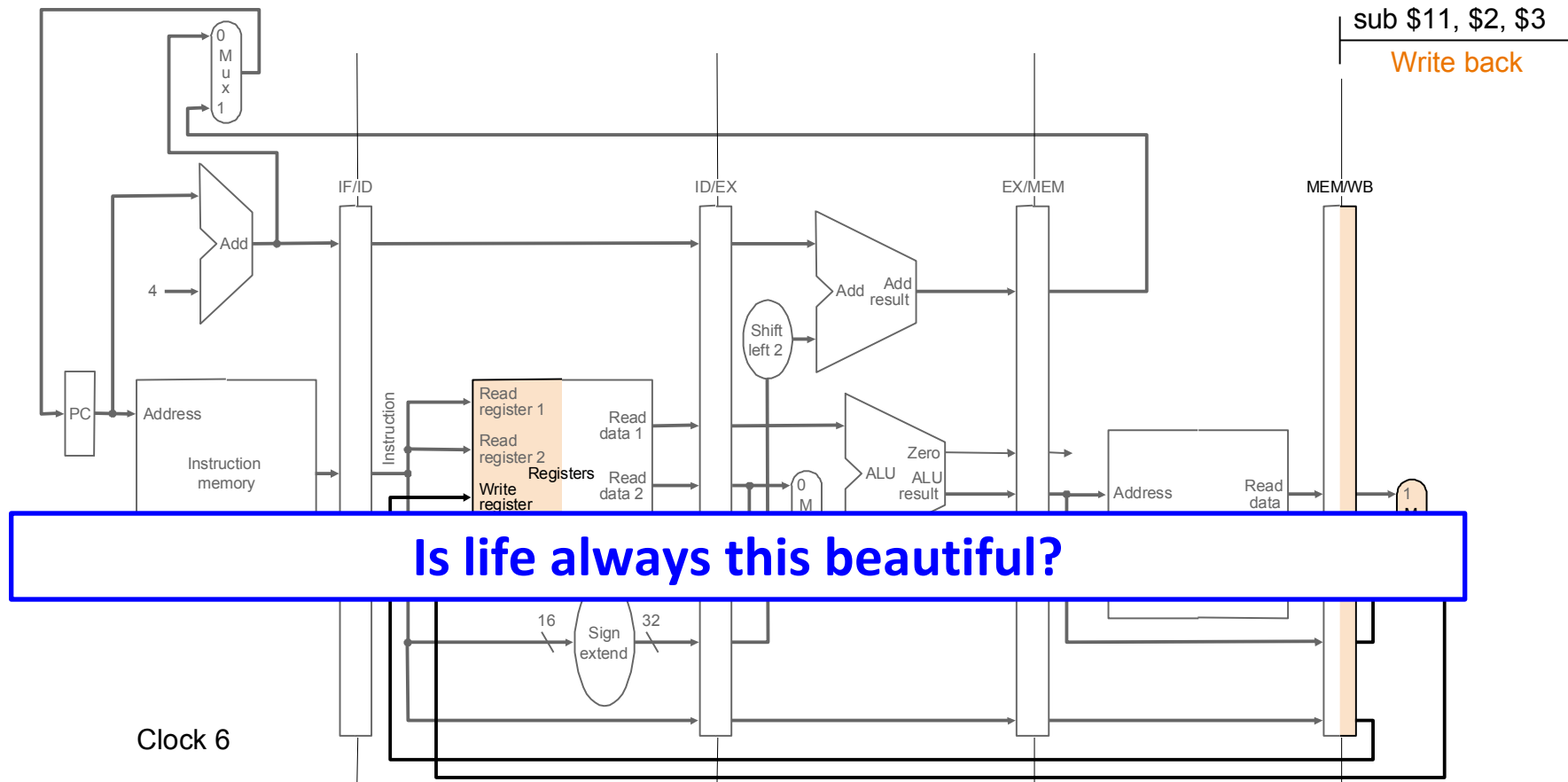
# Pipelined Operation Example

All instruction classes must follow the same path and timing through the pipeline stages.

Any performance impact?



# Pipelined Operation Example



Is life always this beautiful?

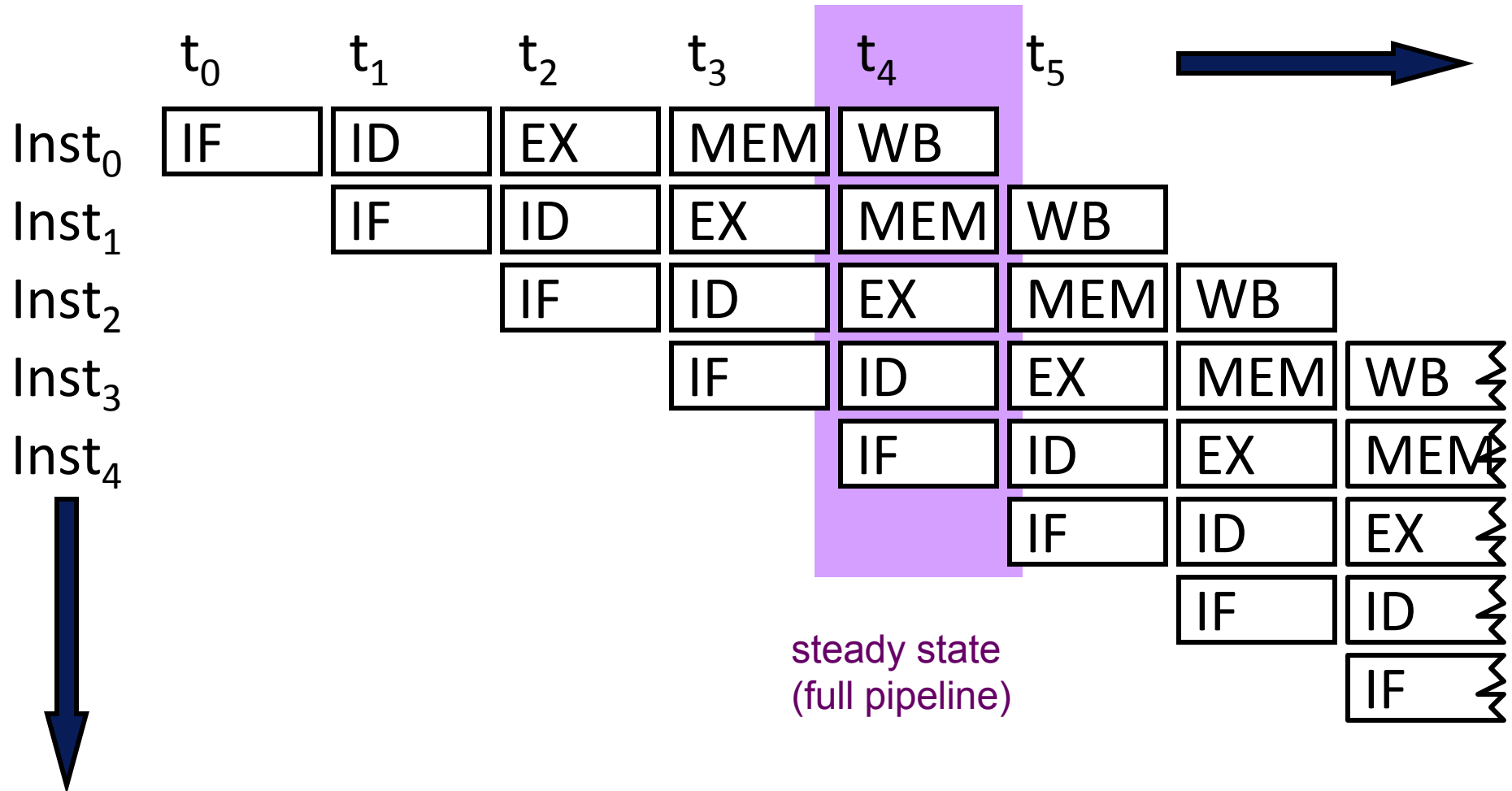
sub \$11, \$2, \$3

Write back

Clock 6



# Illustrating Pipeline Operation: Operation View

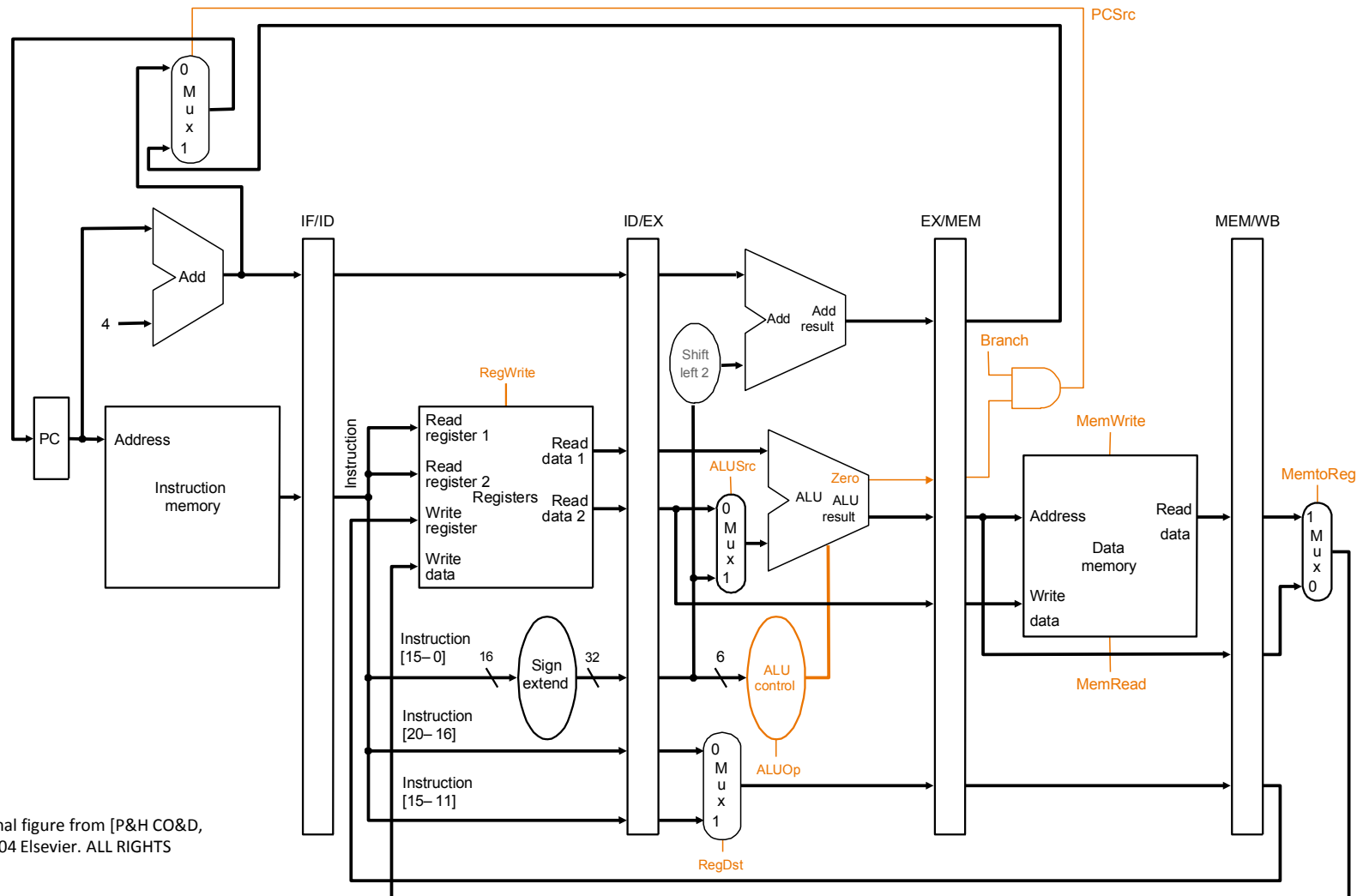


# Illustrating Pipeline Operation: Resource View

---

	t <sub>0</sub>	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>	t <sub>7</sub>	t <sub>8</sub>	t <sub>9</sub>	t <sub>10</sub>
IF	I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>	I <sub>9</sub>	I <sub>10</sub>
ID		I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>	I <sub>9</sub>
EX			I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>
MEM				I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>
WB					I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>

# Control Points in a Pipeline



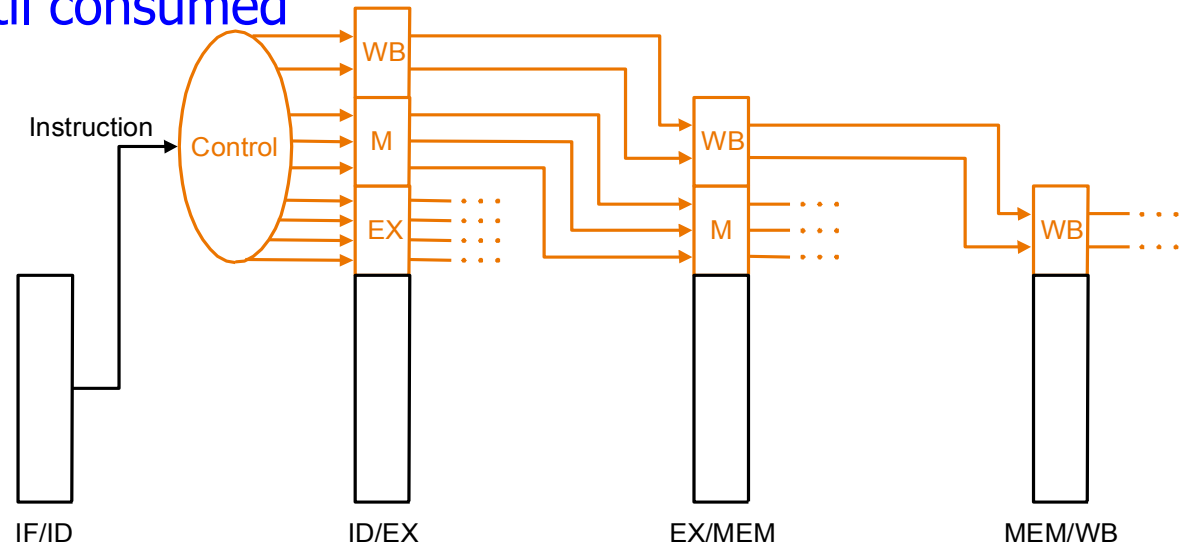
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

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

# Control Signals in a Pipeline

## ■ For a given instruction

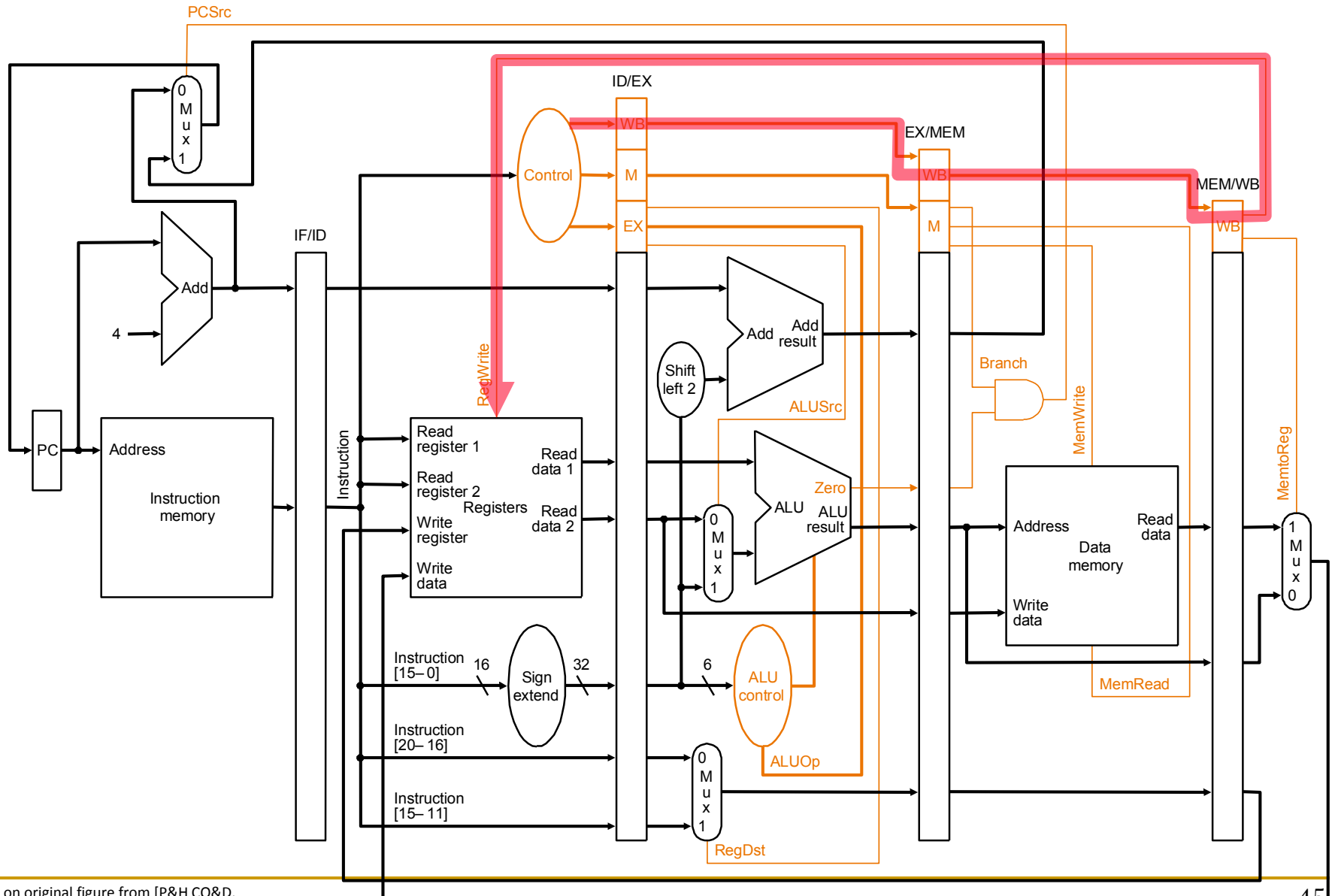
- same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
- ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



- ⇒ Option 2: 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



# Remember: 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 (e.g., all laundry loads go through the same steps)
- 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 → not all need the same stages

- Forcing different instructions to go through the same pipe stages

- external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

- ⇒ different pipeline stages → not the same latency

- Need to force each stage to be controlled by the same clock

- 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 provides correct results

- pipeline stalls (pipeline is not always moving)

# 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*

---

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

# Dependences and Their Types

---

- Also called “dependency” or *less desirably* “hazard”
- Dependences 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)

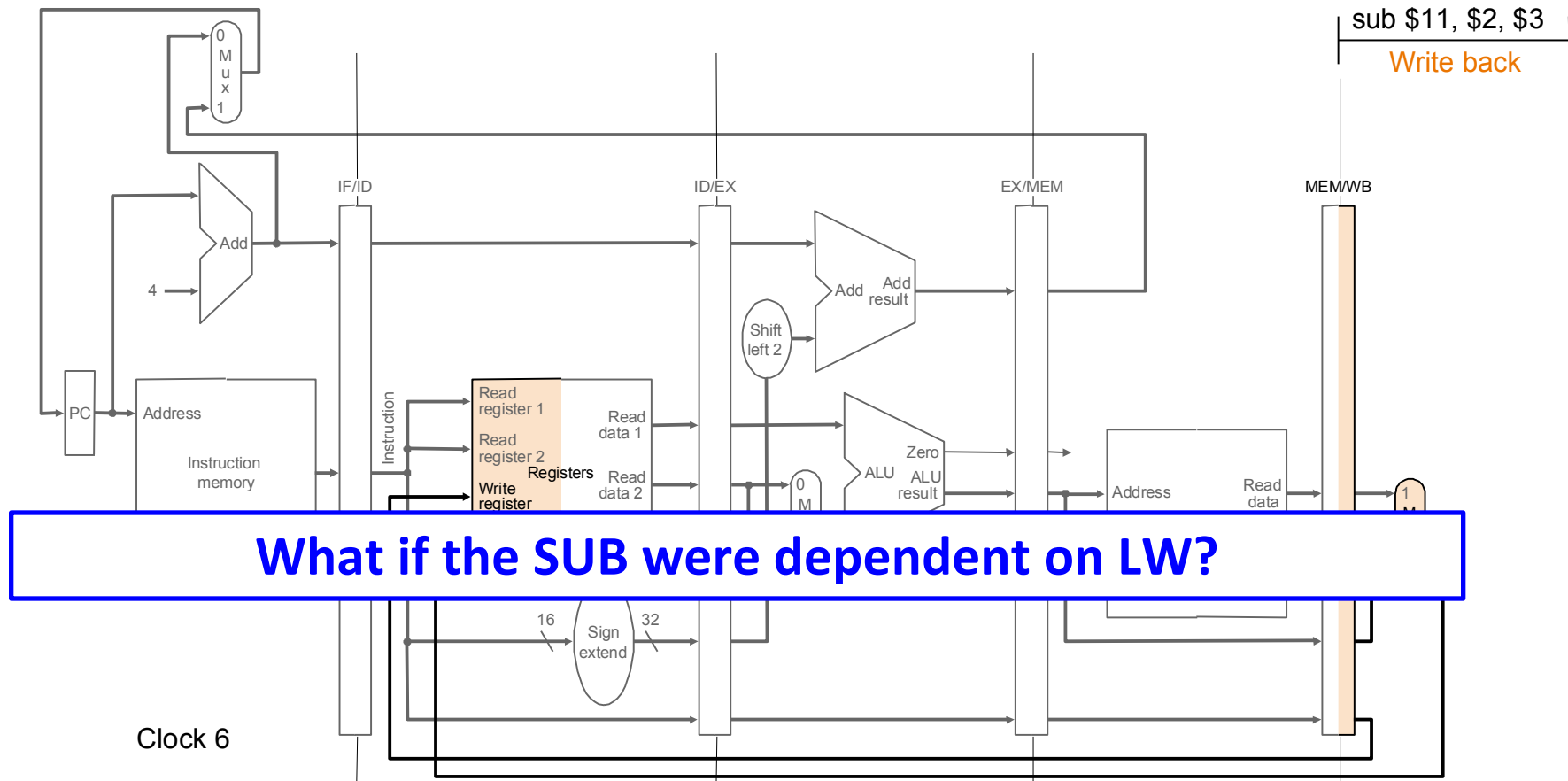
Output-dependence

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



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: ...

We did not cover the following slides in lecture.  
These are for your preparation for the next lecture.

# 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?