

# Computer Organization and Networks

(INB.06000UF, INB.07001UF)

## Chapter 5: Pipelining

Winter 2021/2022

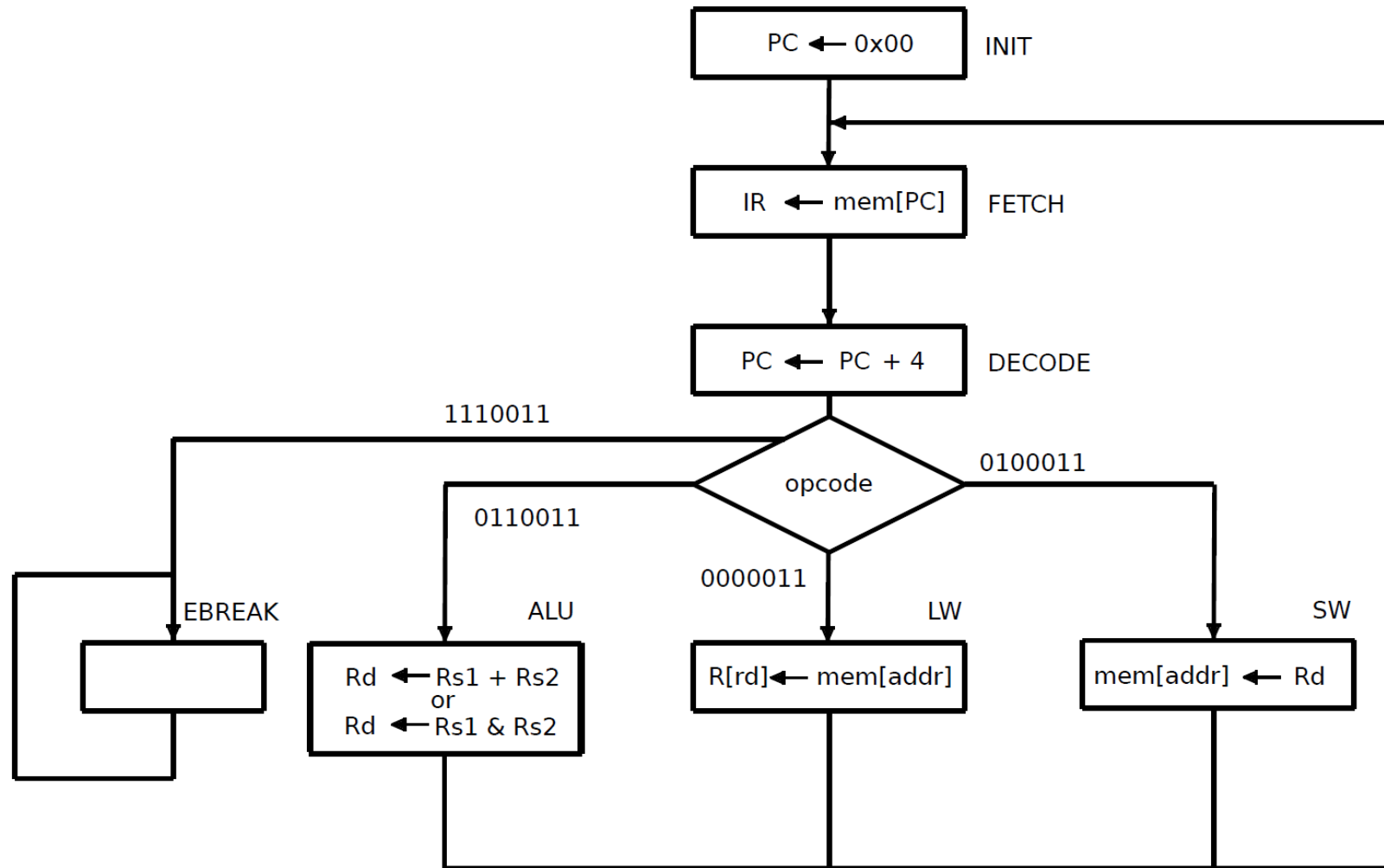


Stefan Mangard, [www.iaik.tugraz.at](http://www.iaik.tugraz.at)

# Notes on this Slide Set

- This part of the lecture is based on slides by Prof. Onur Mutlu (ETH Zürich)
- The slides have been changed significantly in several aspects
  - adaption to RISC-V
  - Addition / deletion of slides and slide content
  - Change of layout
- Original source:  
<https://safari.ethz.ch/digitaltechnik/spring2019/doku.php?id=schedule>

# Simple Fetch/Decode/Execute ASMM



# Drawbacks of the Simple Fetch/Decode/Execute Design

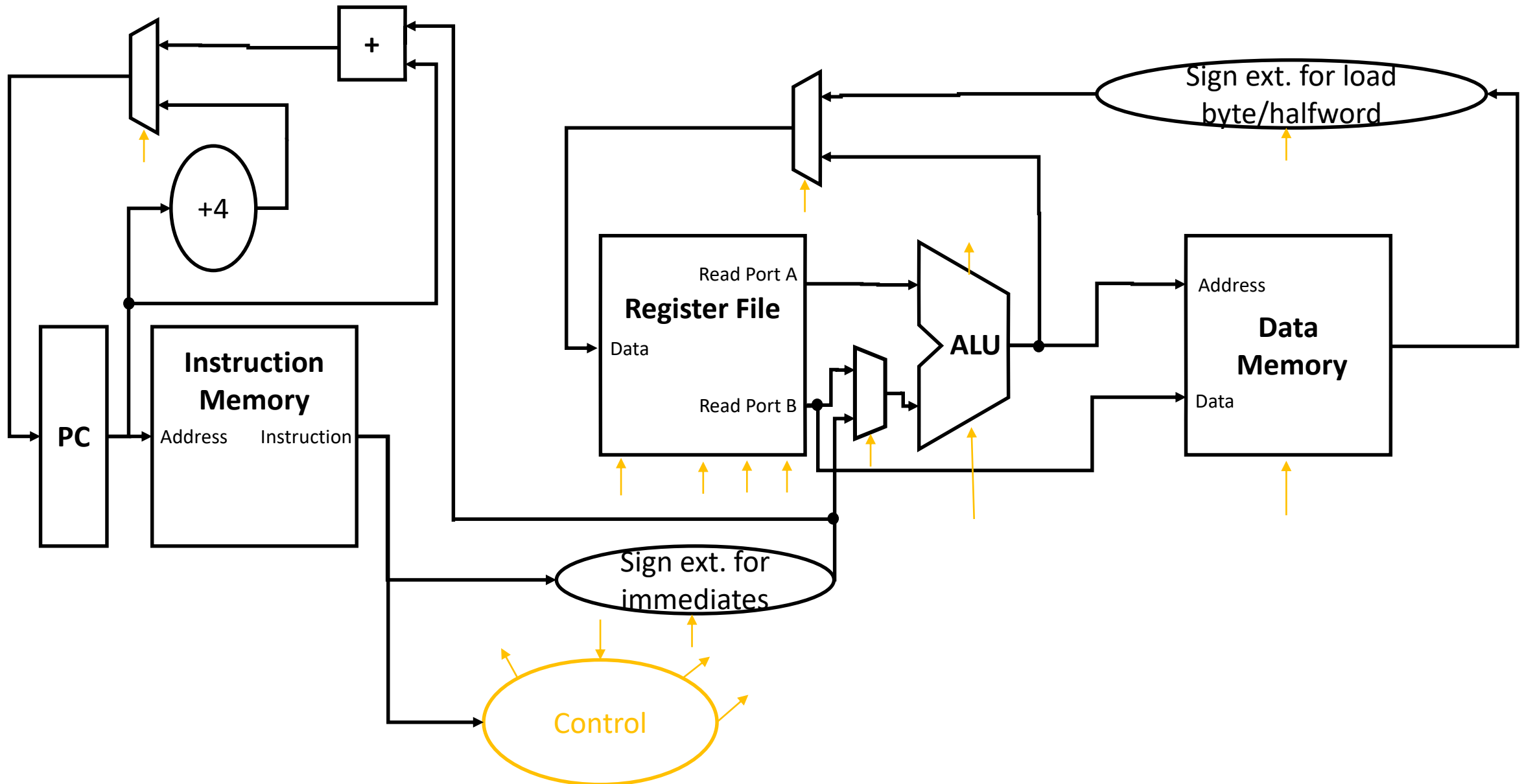
- The operations that we perform in the fetch, decode and execute stage are very different in terms of the critical path

A computation and memory lookup during the execution take much more time than the decision on which instruction to execute (decoding) → the worst-case execution stage will define the clock frequency

# Basic Idea of Multicycle Architectures

- Cut the operations that are needed for one instruction into more fine-granular operations than fetch, decode, execute
- Each instruction is a multicycle instruction and takes as many cycles as needed to perform the actions defined by the instruction
  - Instructions lead to different numbers of operations (and therefore take longer / shorter depending on their complexity)

# High-Level Overview (Single Cycle Datapath)



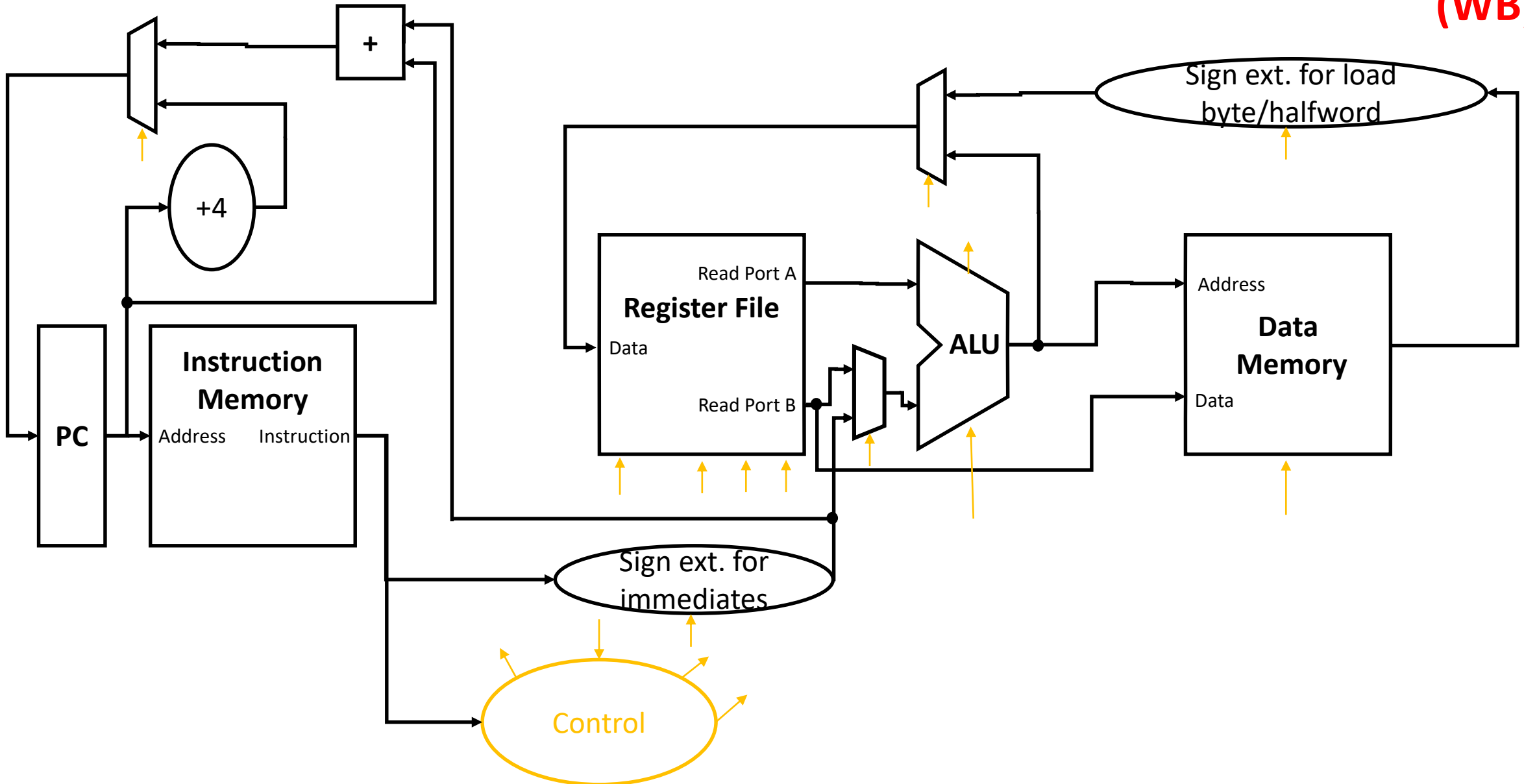
# Fetch (F)

# Decode (D)

# Execute (E)

# Memory (M)

# Write Back (WB)



# 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

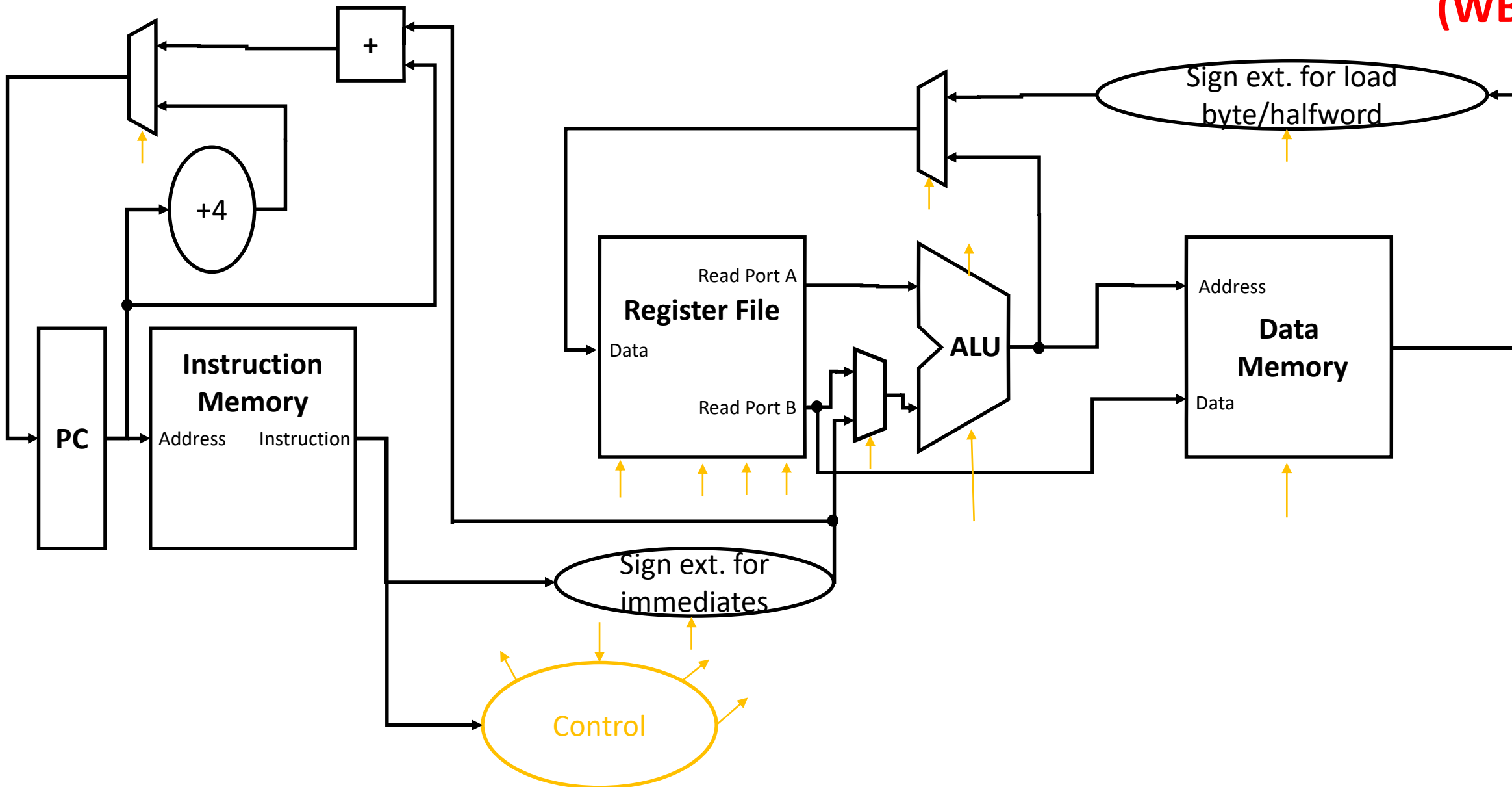
# Fetch (F)

# Decode (D)

# Execute (E)

# Memory (M)

# Write Back (WB)



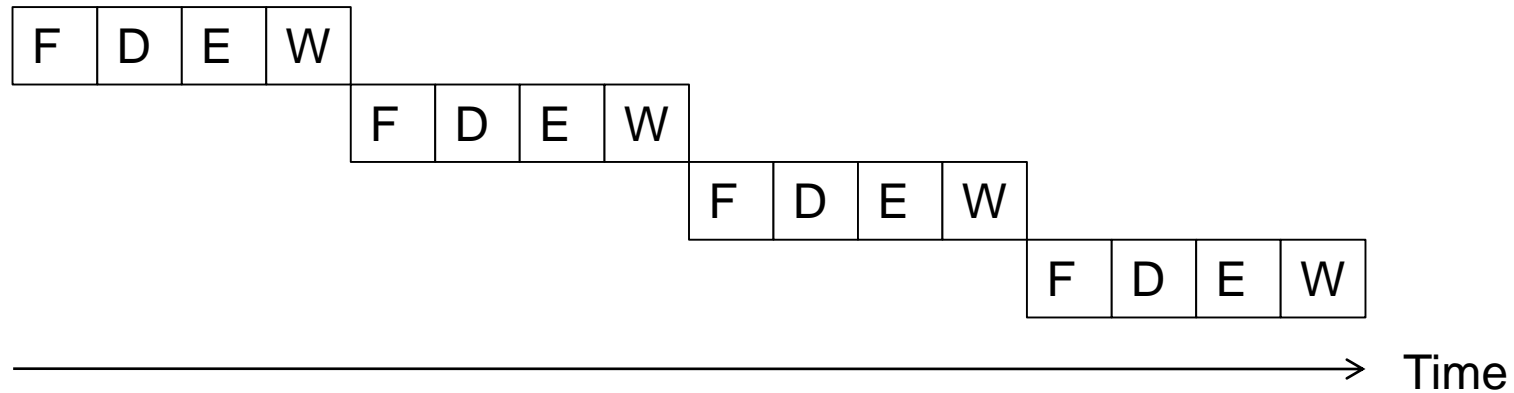
# Pipelining

# Pipelining: Basic Idea

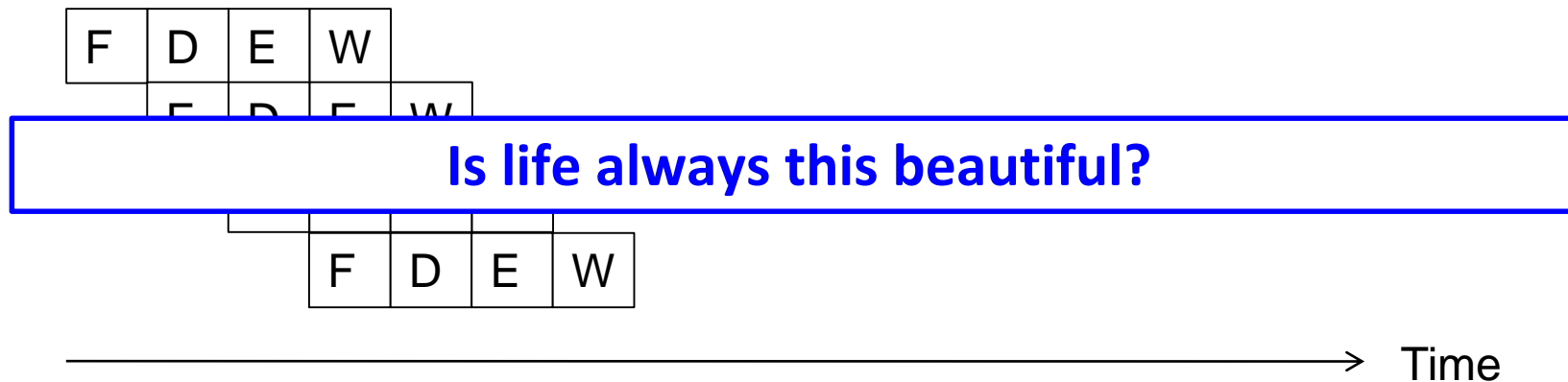
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each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

# Example: Execution of Four Independent ADDs (no memory needed)

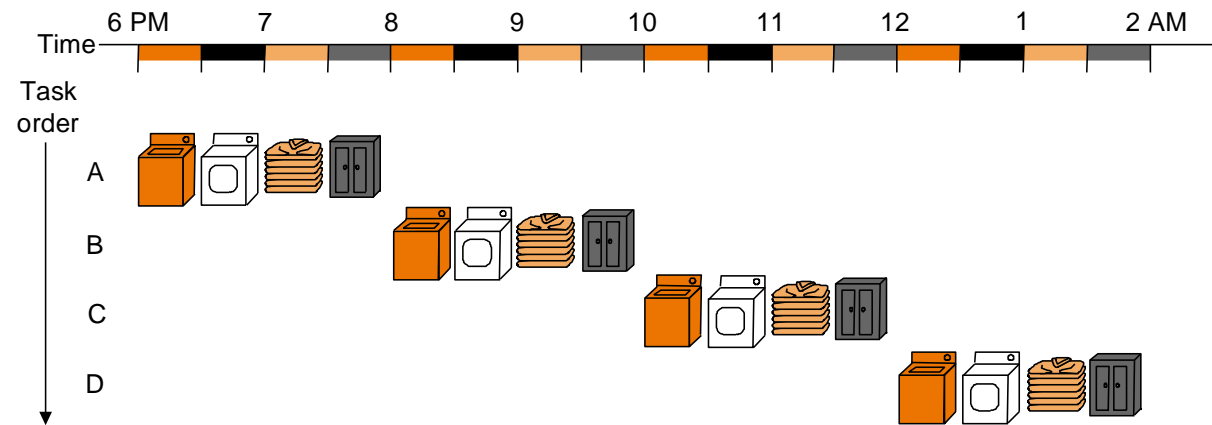
- Multi-cycle: 4 cycles per instruction



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



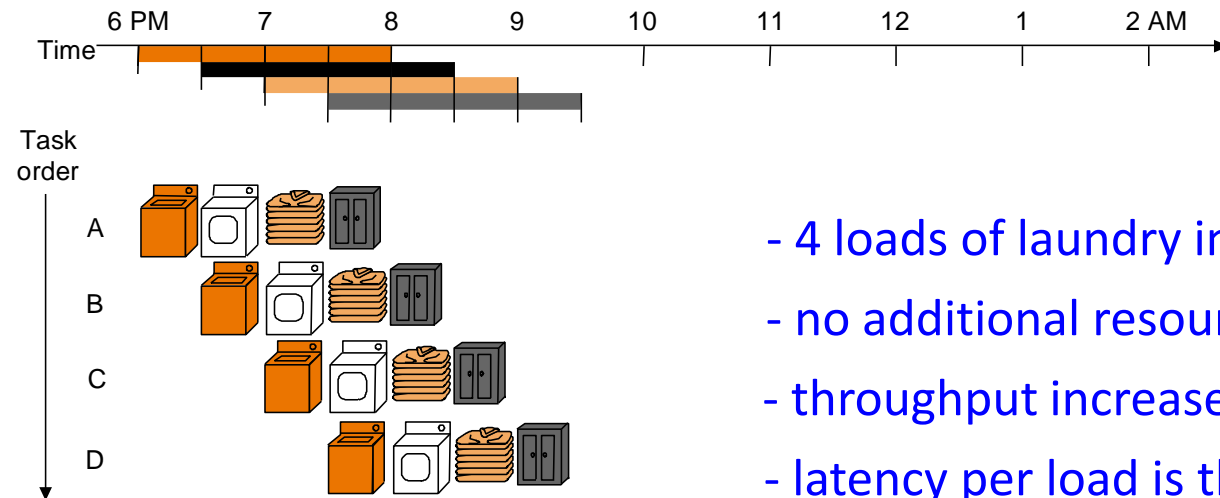
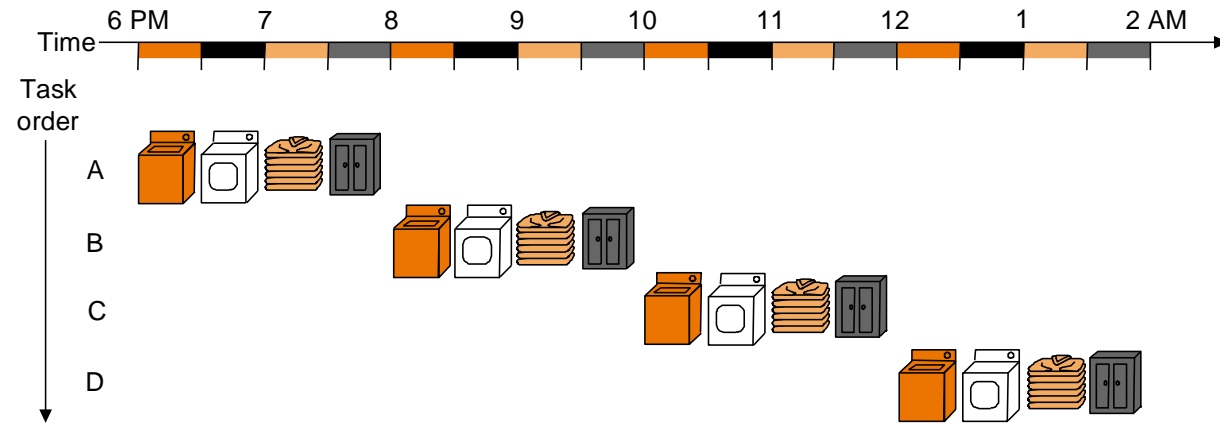
# The Laundry Analogy



- “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”

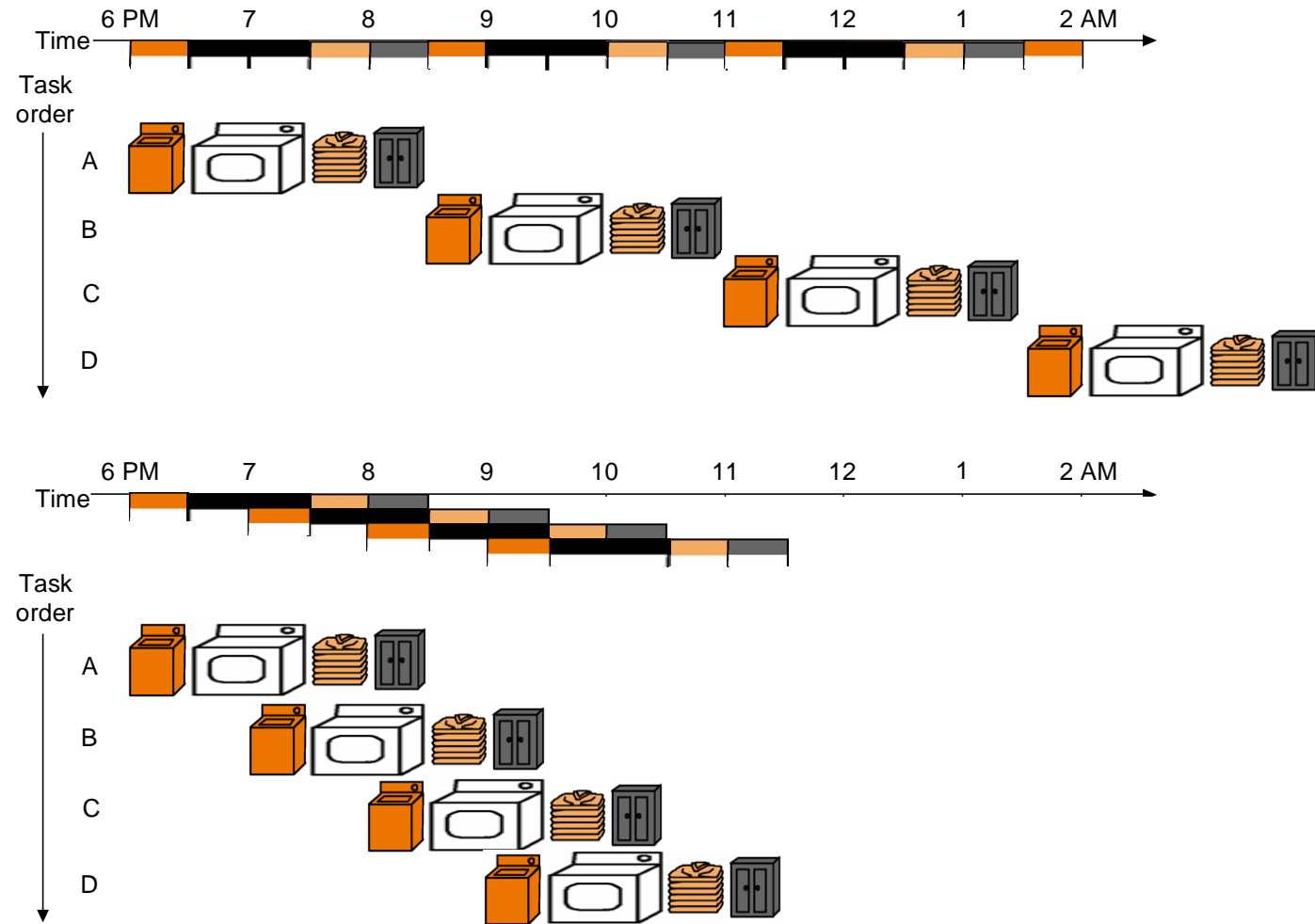
- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources

# Pipelining Multiple Loads of Laundry



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

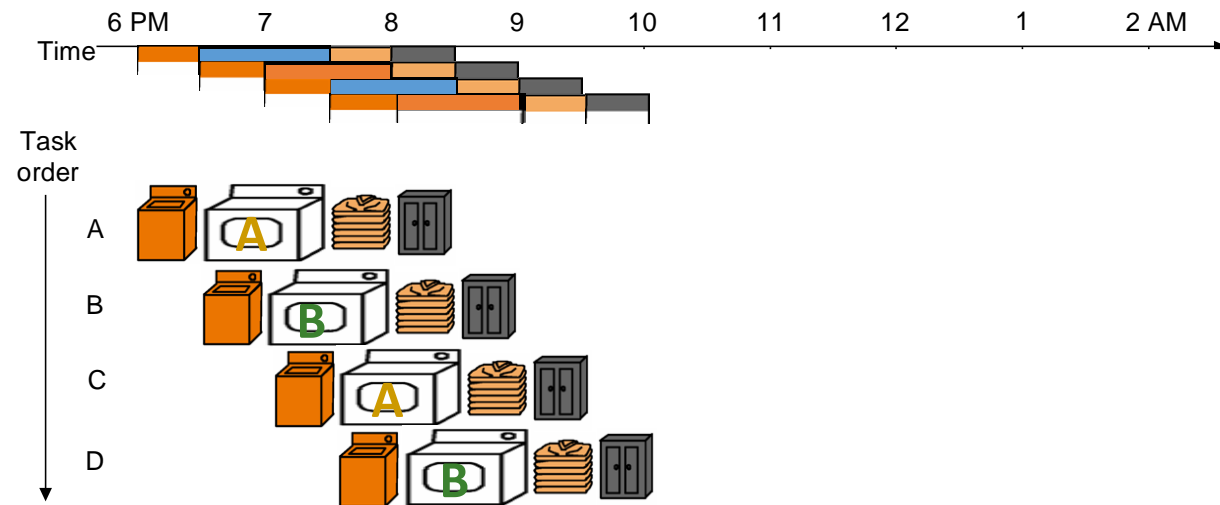
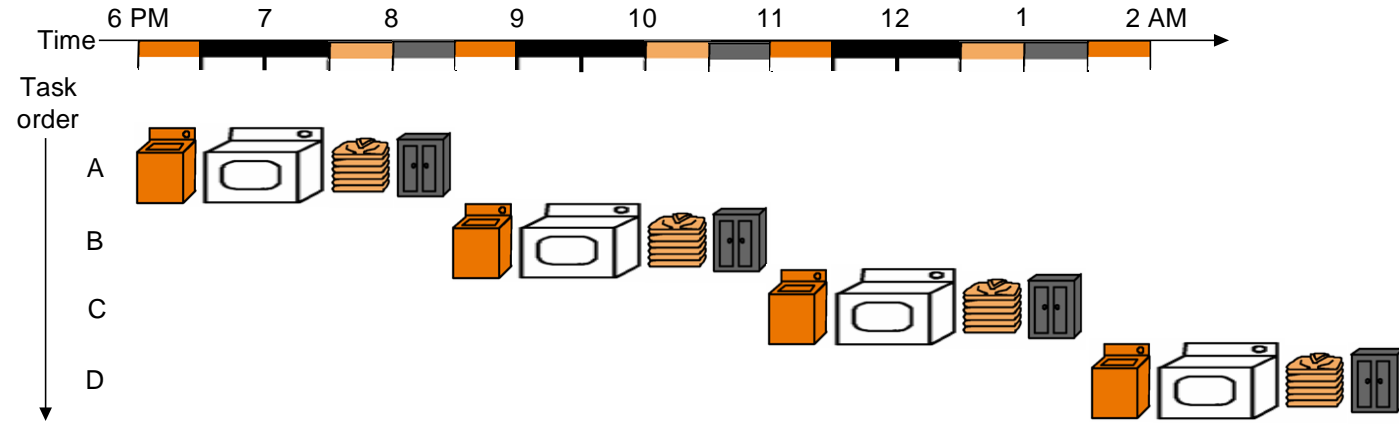
# Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput



# Pipelining Multiple Loads of Laundry: In Practice

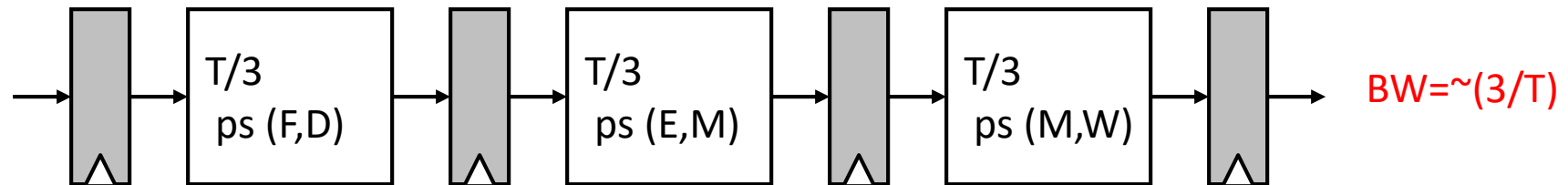
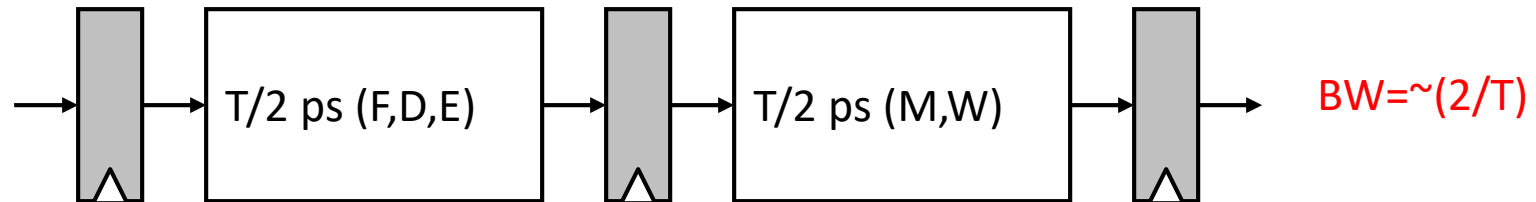


throughput restored (2 loads per hour) using 2 dryers

# 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

# Ideal Pipelining



# More Realistic Pipeline: Throughput

- Nonpipelined version with delay  $T$

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

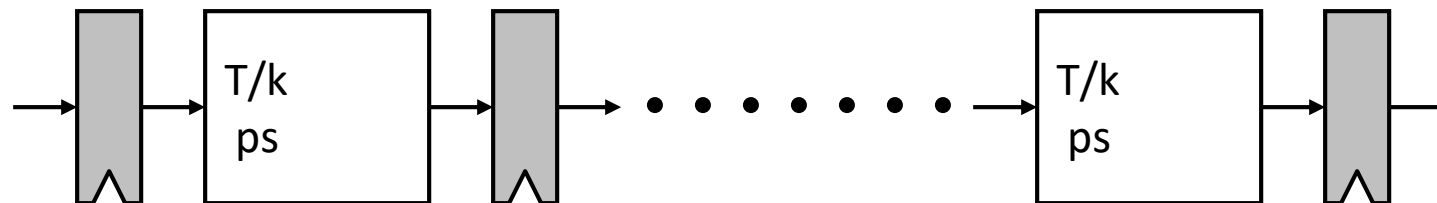


- k-stage pipelined version

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

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

**Register delay reduces throughput  
(switching overhead between stages)**



# More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost  $G$

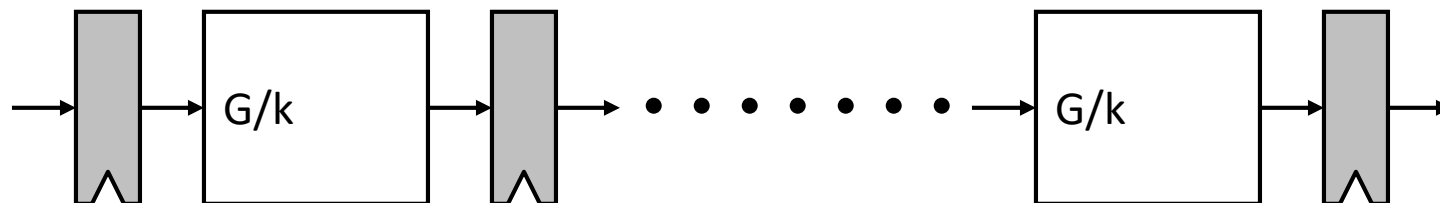
$\text{Cost} = G + L$  where  $L$  = register cost



- $k$ -stage pipelined version

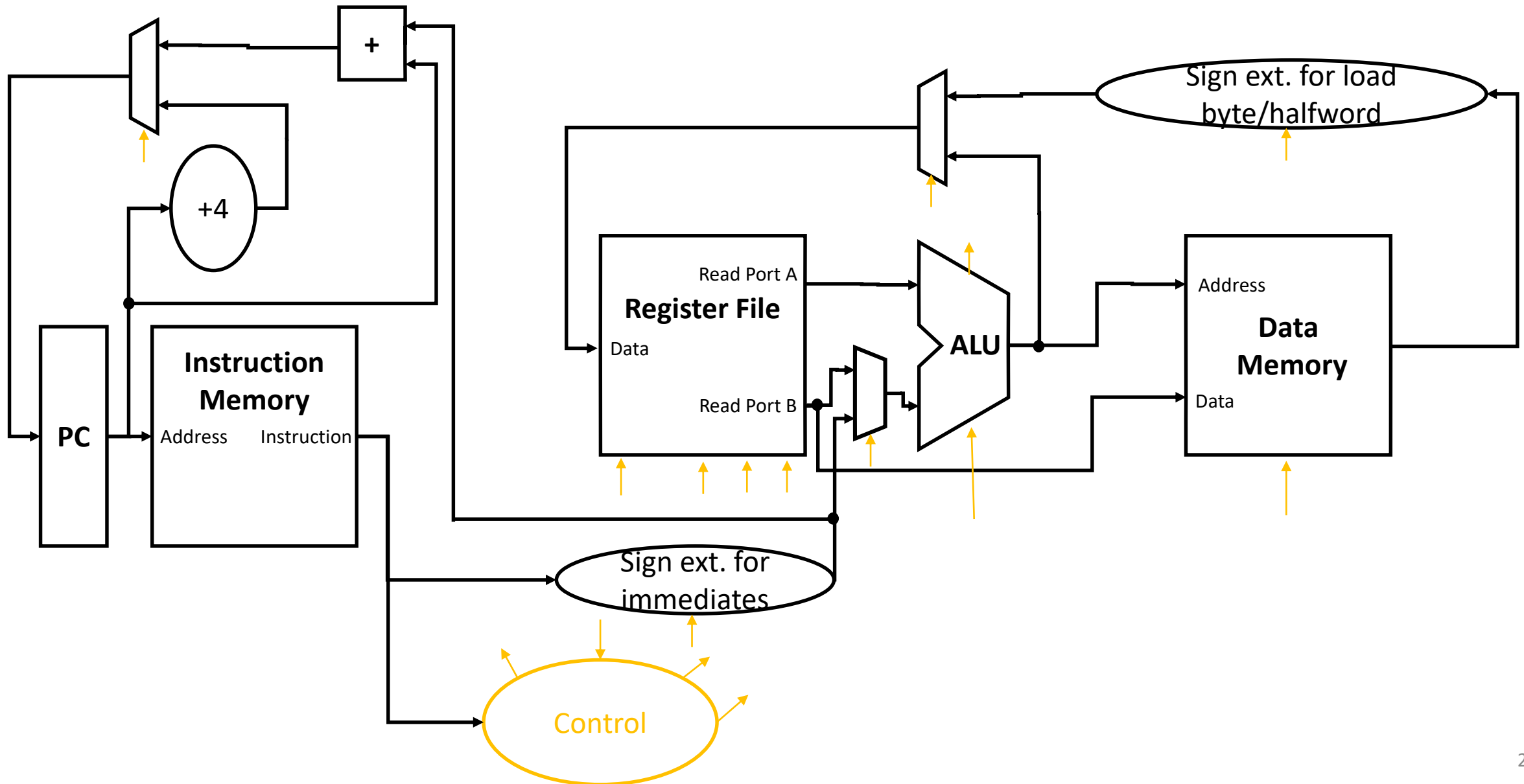
$\text{Cost}_{k\text{-stage}} = G + Lk$

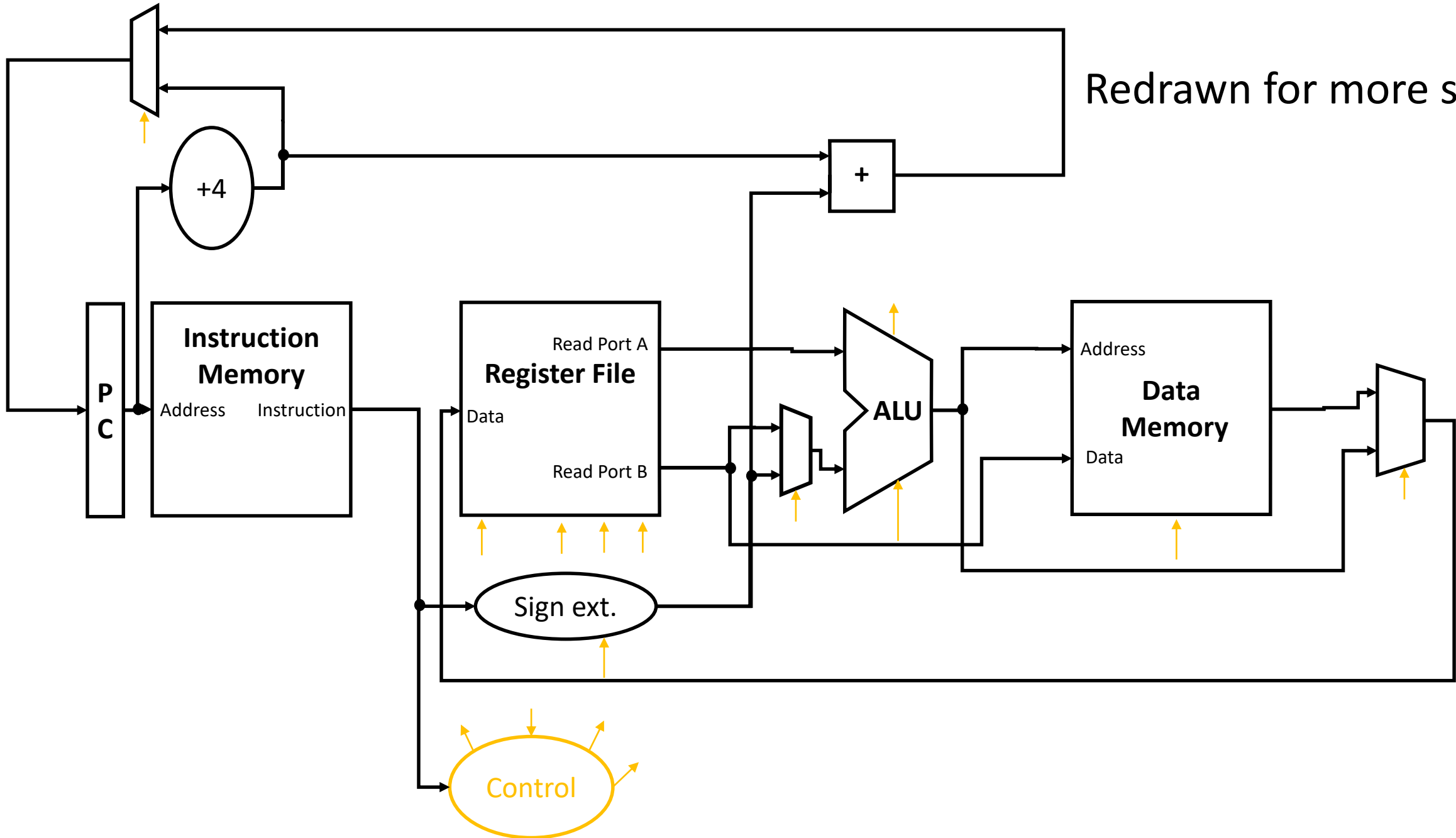
**Registers increase hardware cost**



# Pipelining Instruction Processing

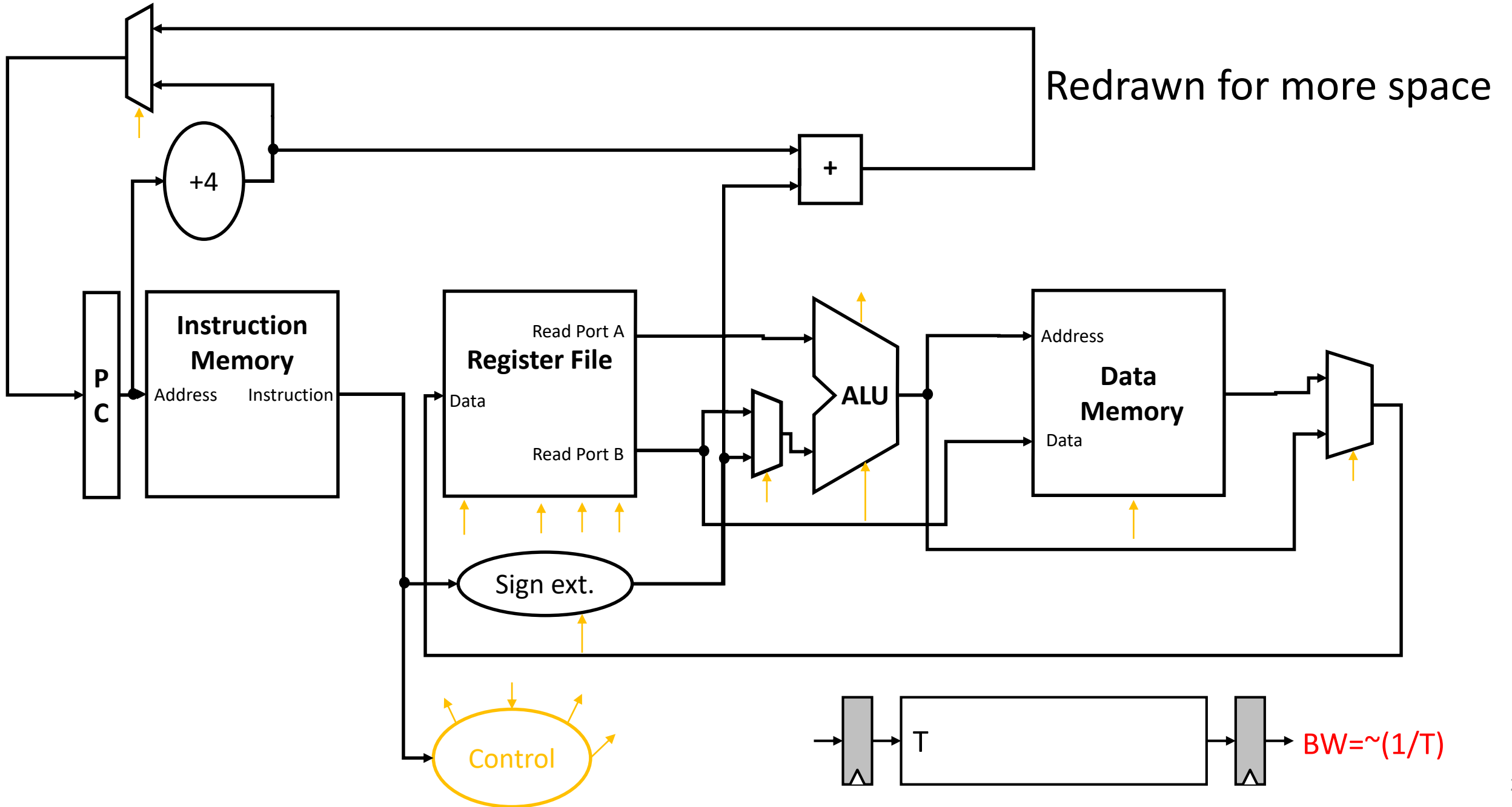
# High-Level Datapath

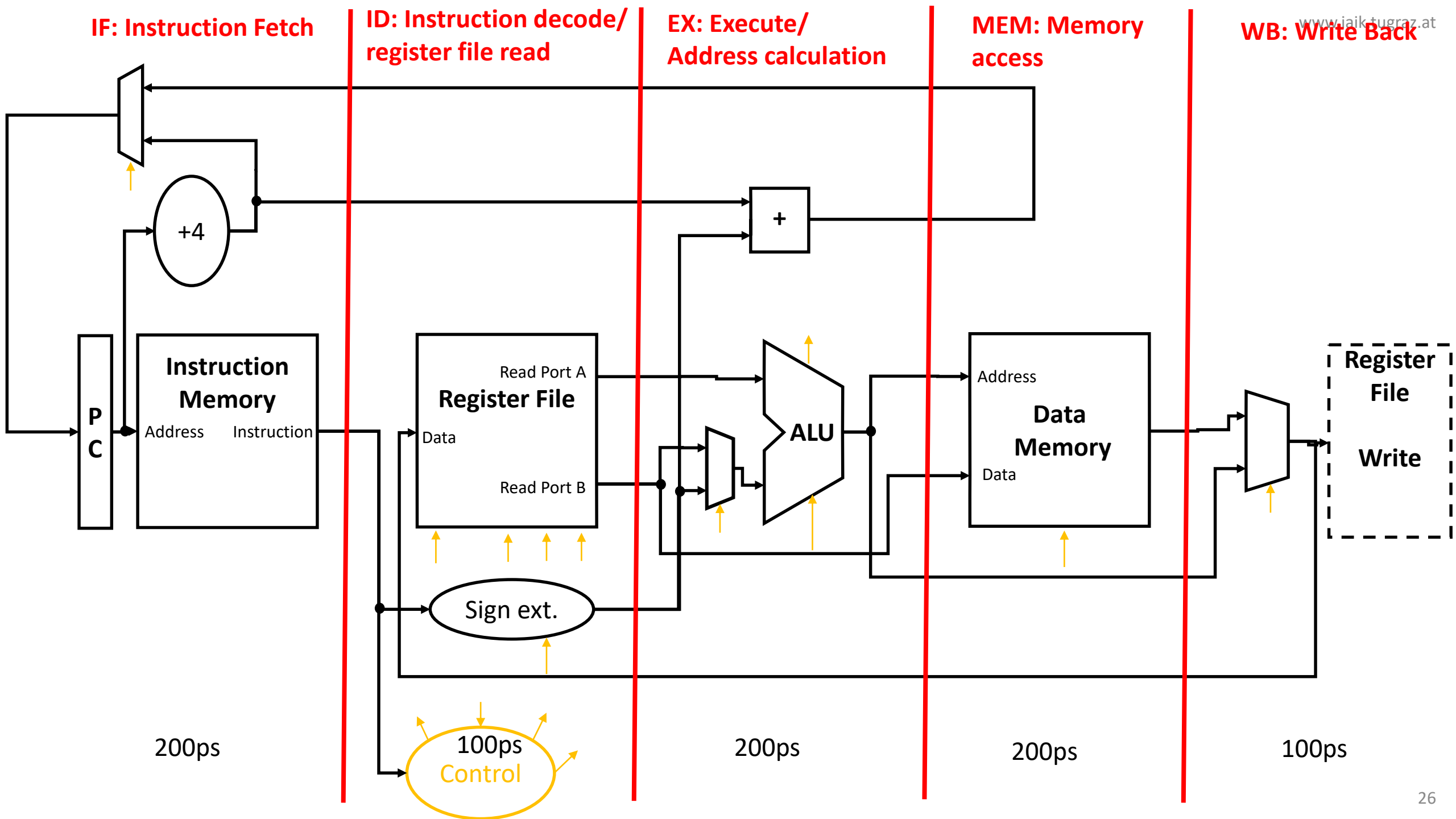




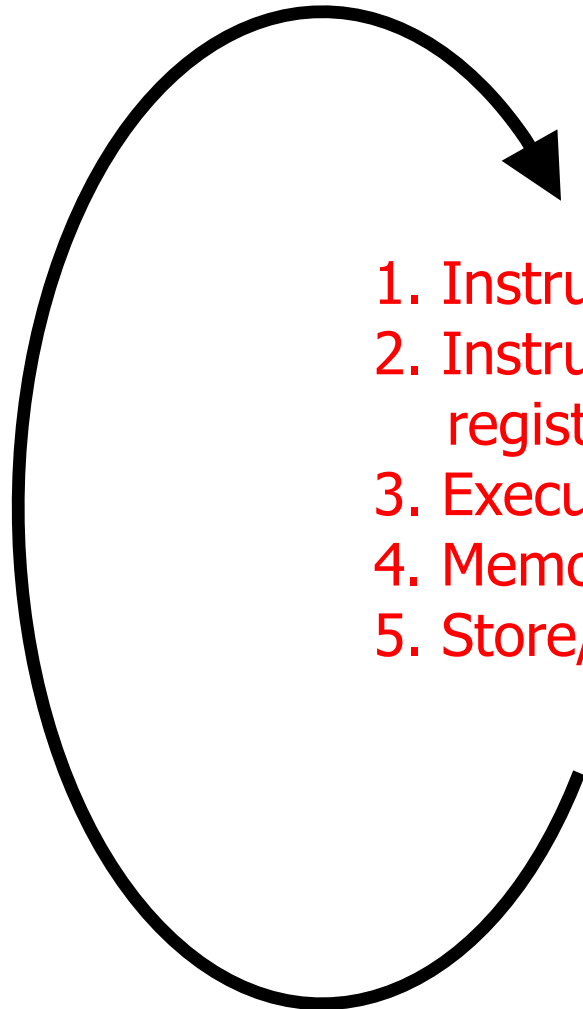
Redrawn for more space





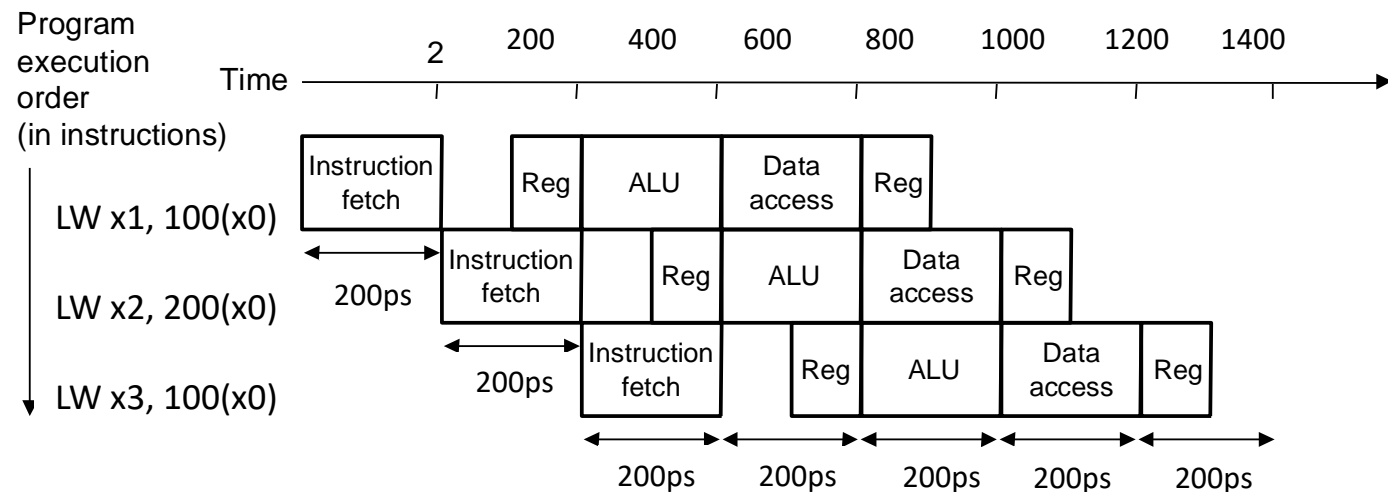
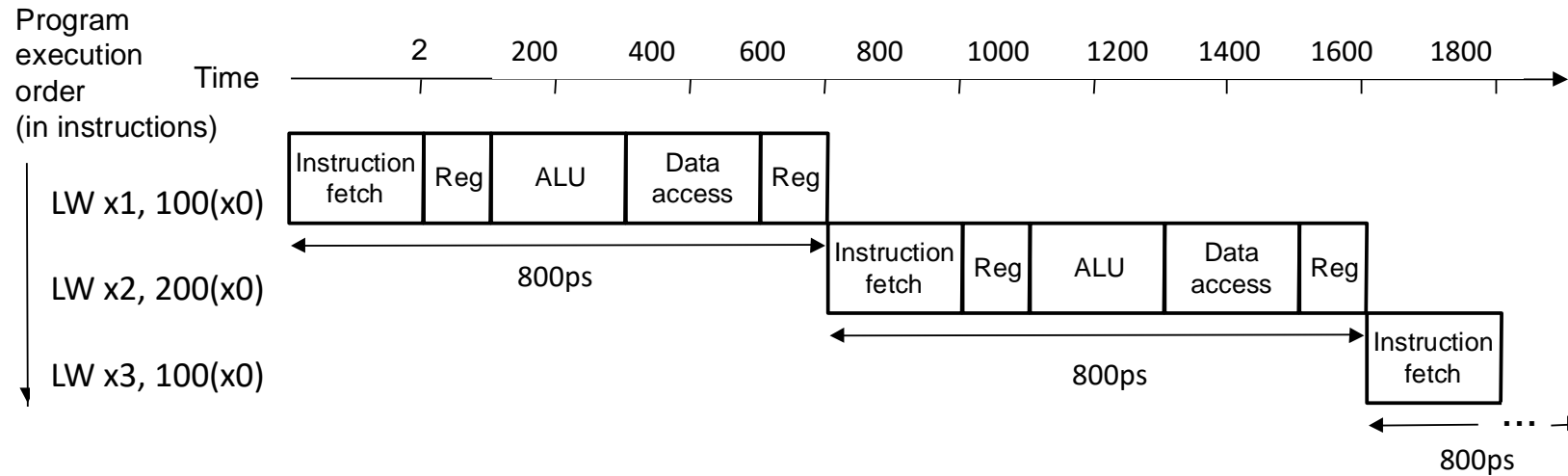


# The Instruction Processing Cycle



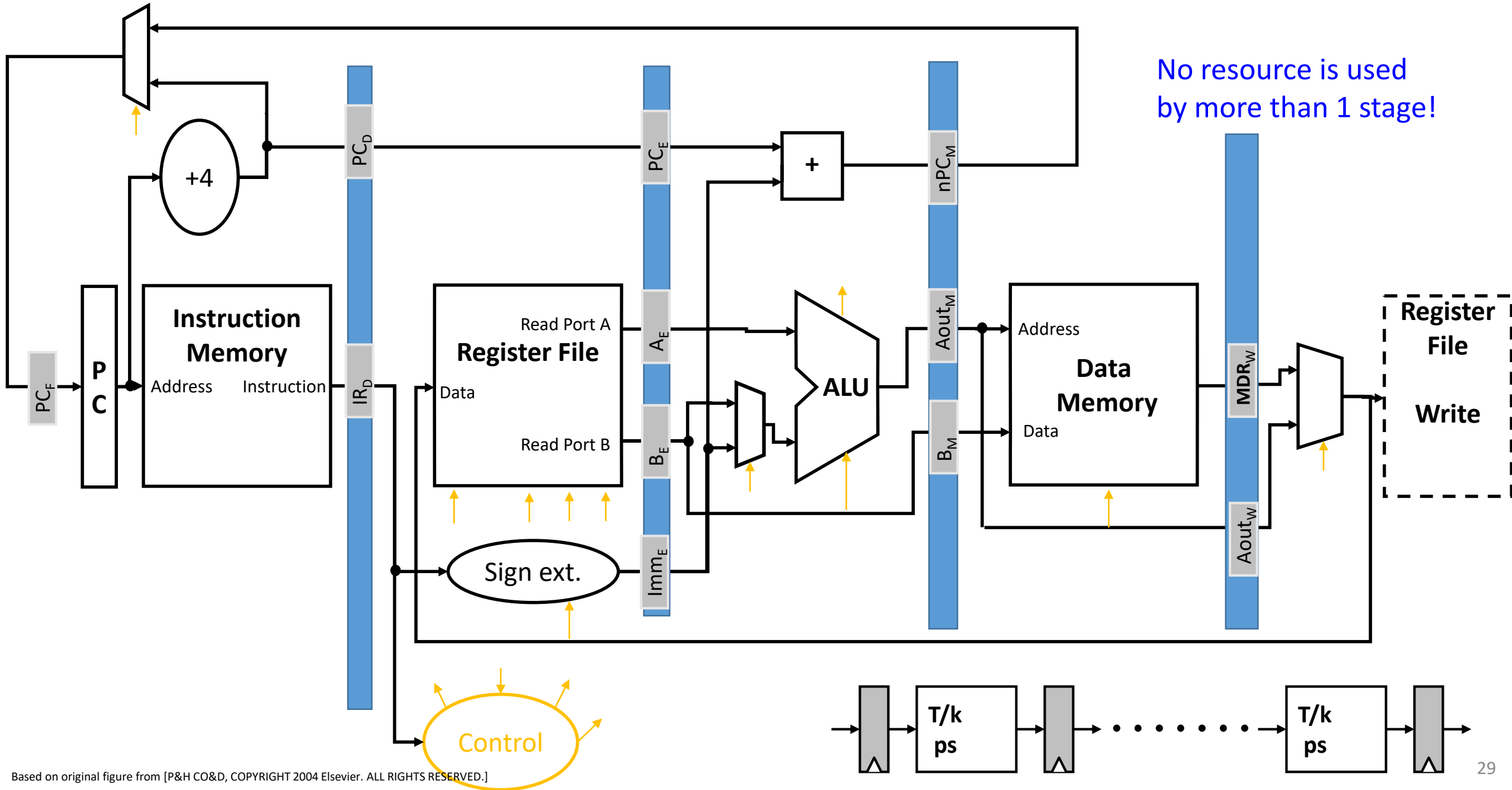
1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

# Instruction Pipeline Throughput



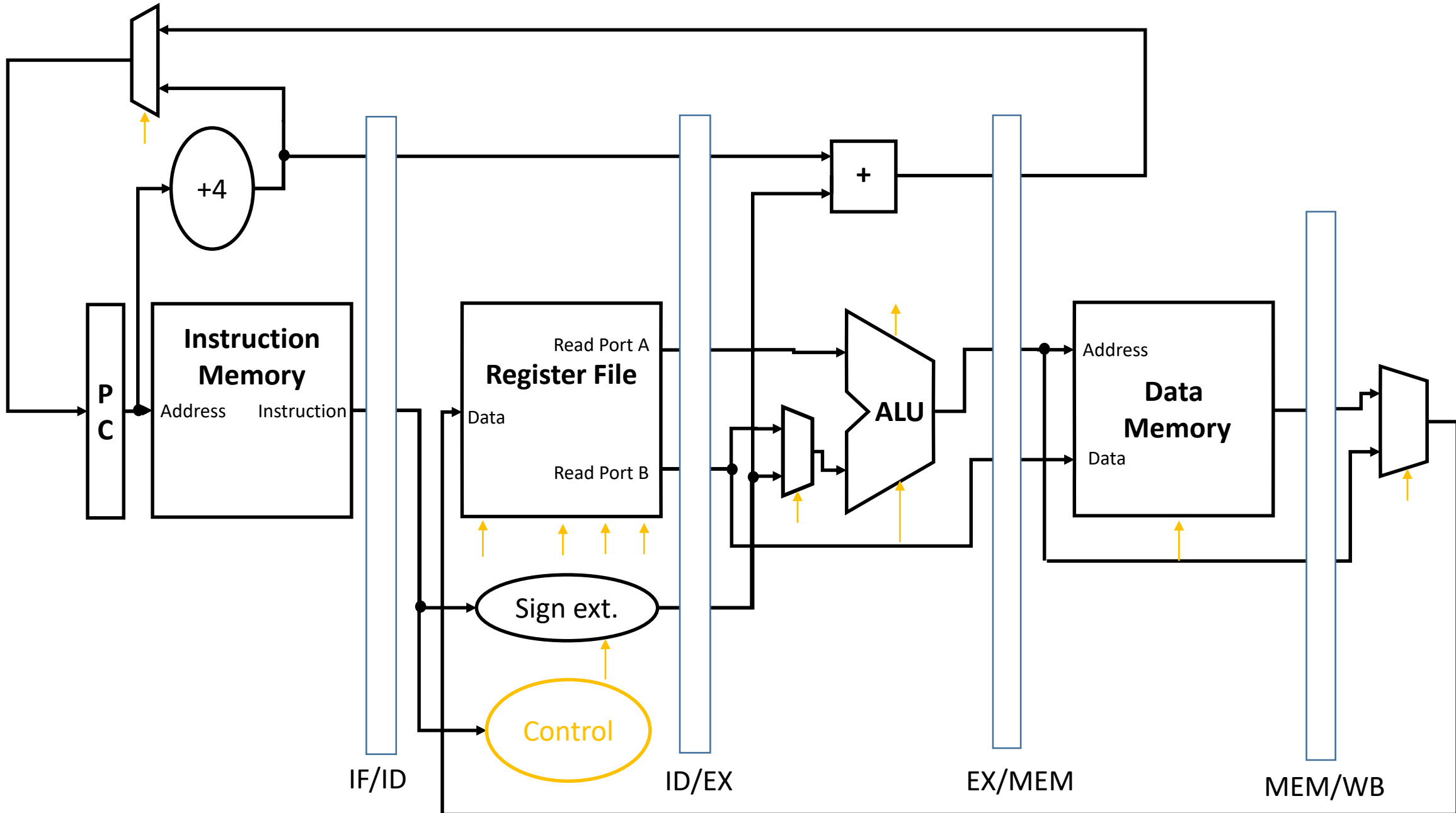
5-stage speedup is 4, not 5 as predicted by the ideal model.  
 (We complete an instruction every 200ps instead of every 800ps)

# Enabling Pipelined Processing: Pipeline Registers

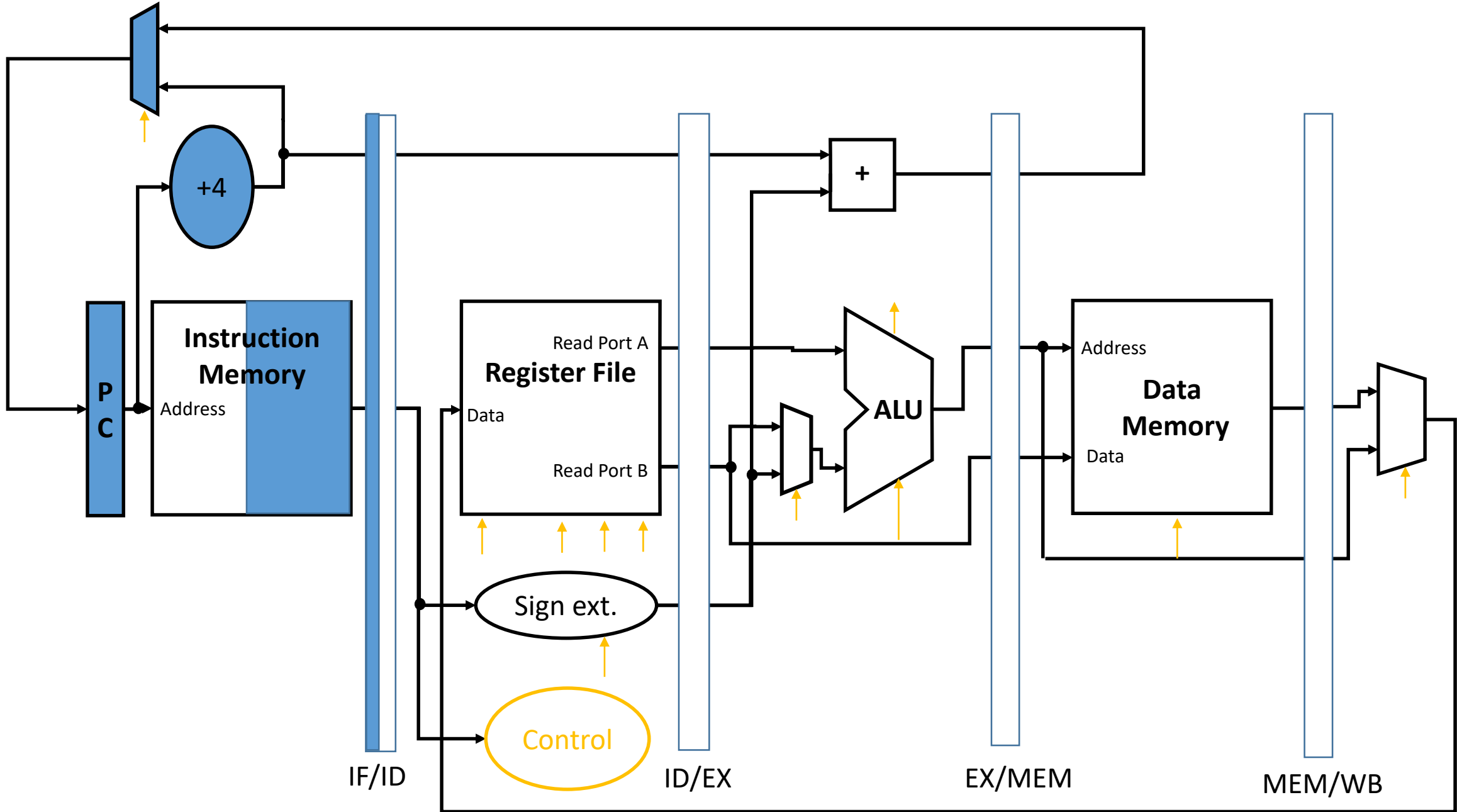


Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

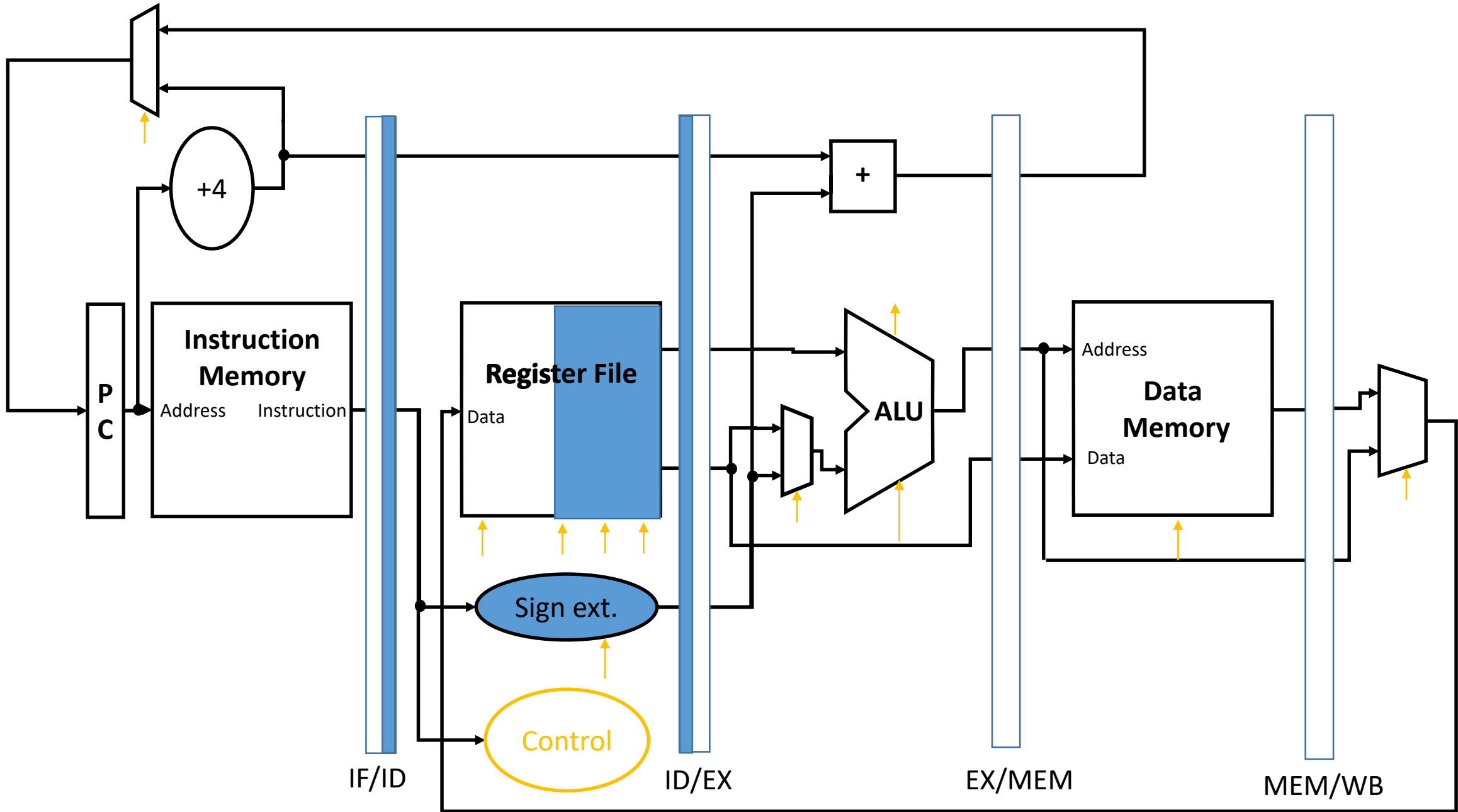
# Pipelined Operation



# LW Instruction Fetch

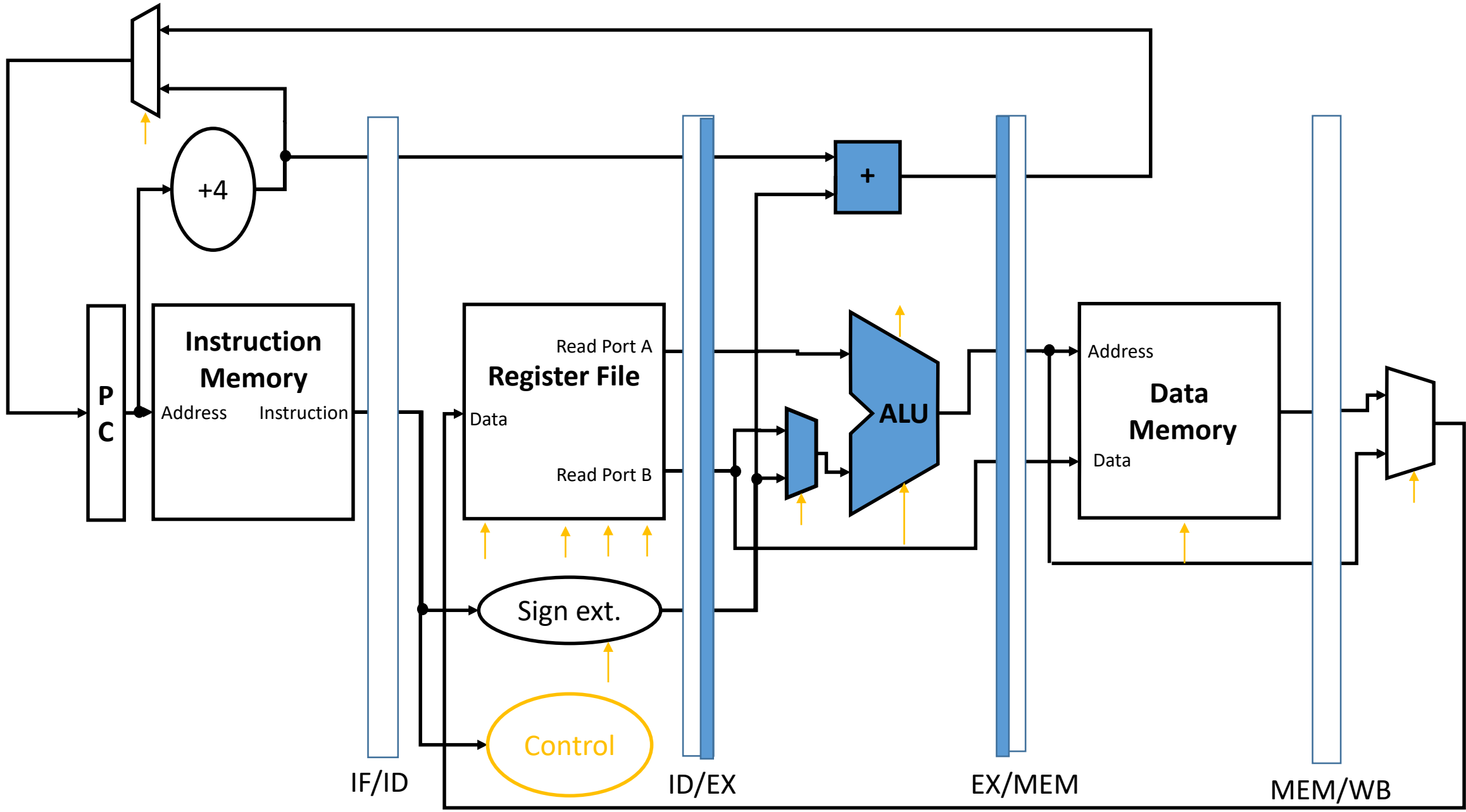


# LW Instruction Decode



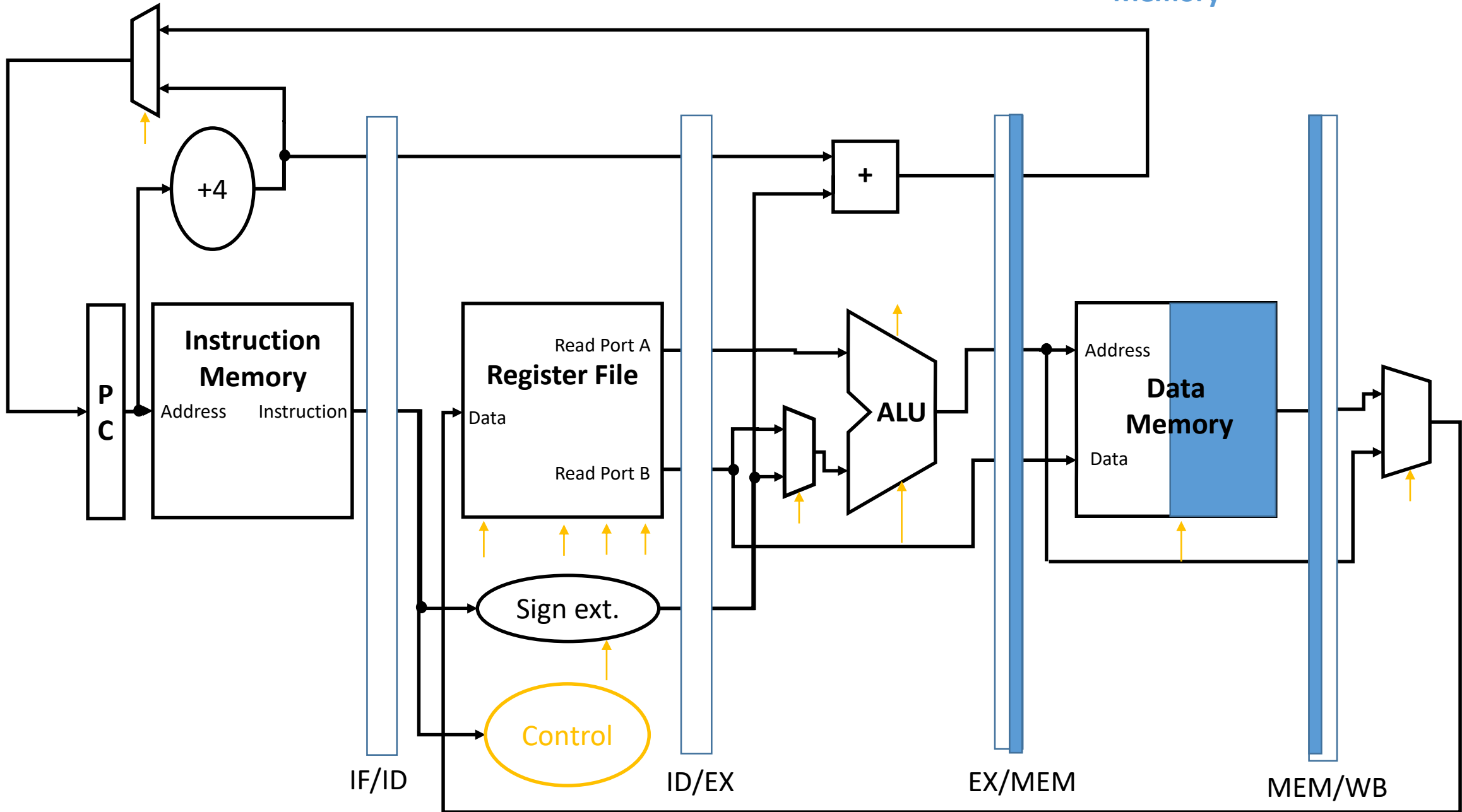


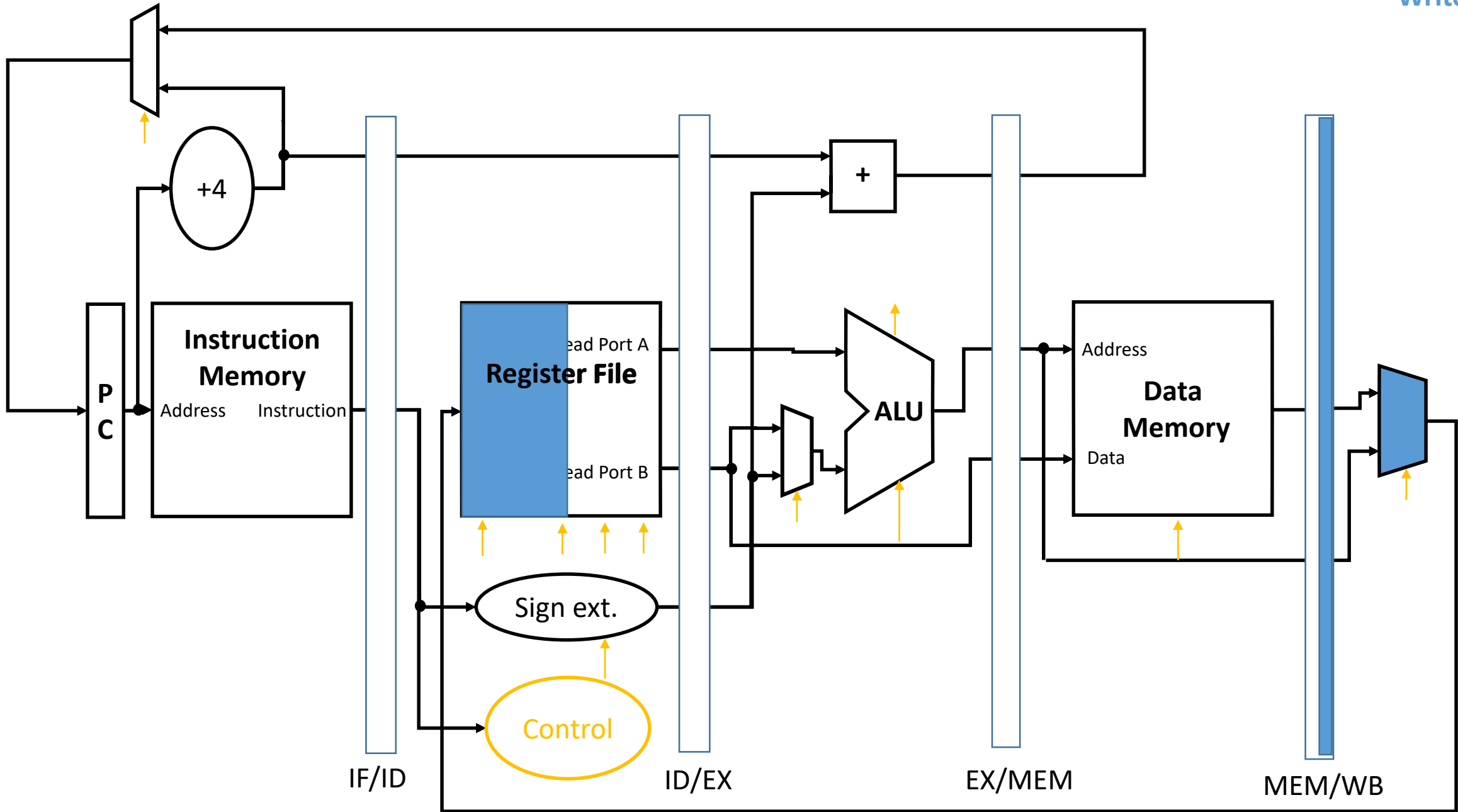
# LW Execute



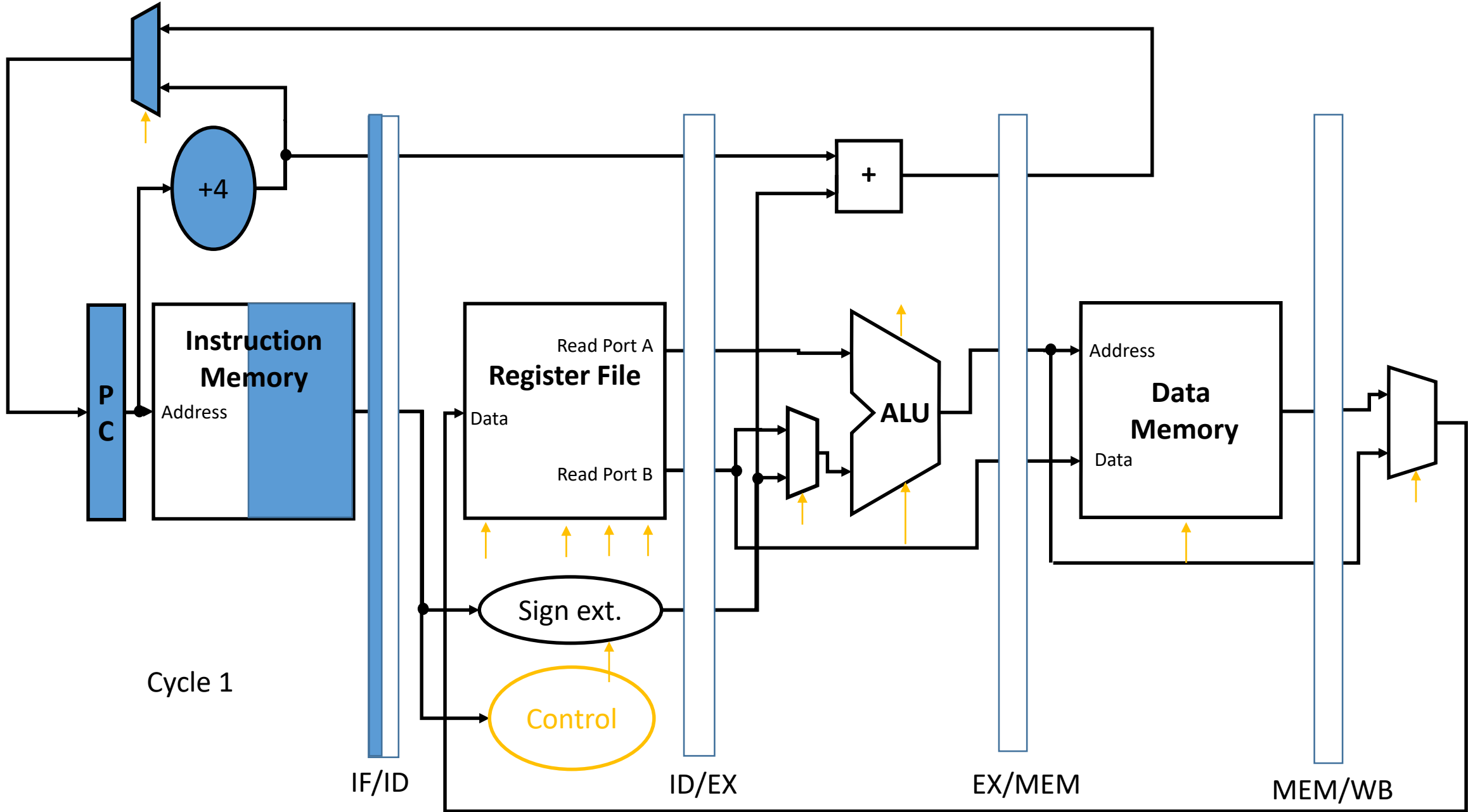
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# LW Memory





# LW x1,100(x0) Instruction Fetch

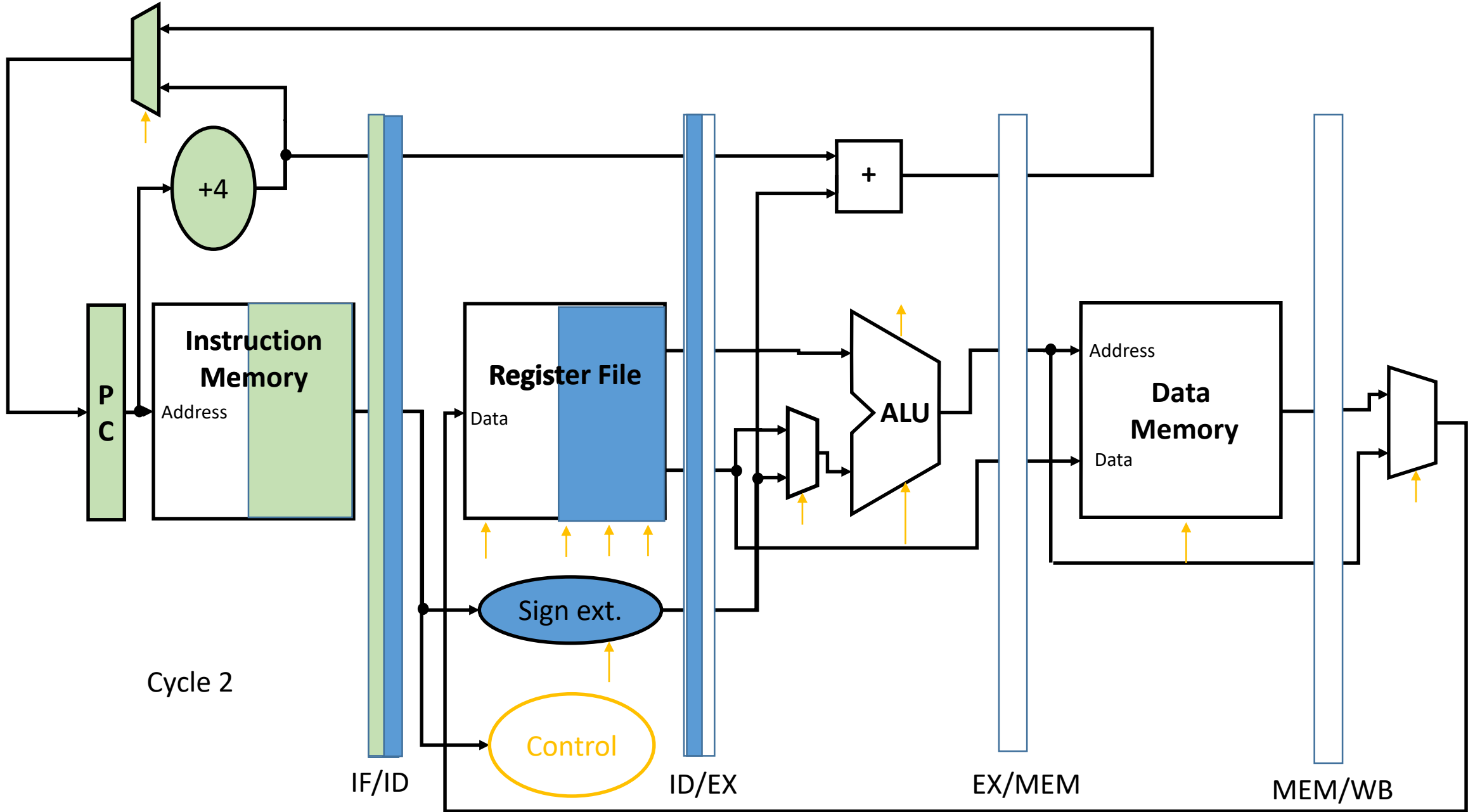


Cycle 1

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

ADD x2, x3, x4  
Instruction Fetch

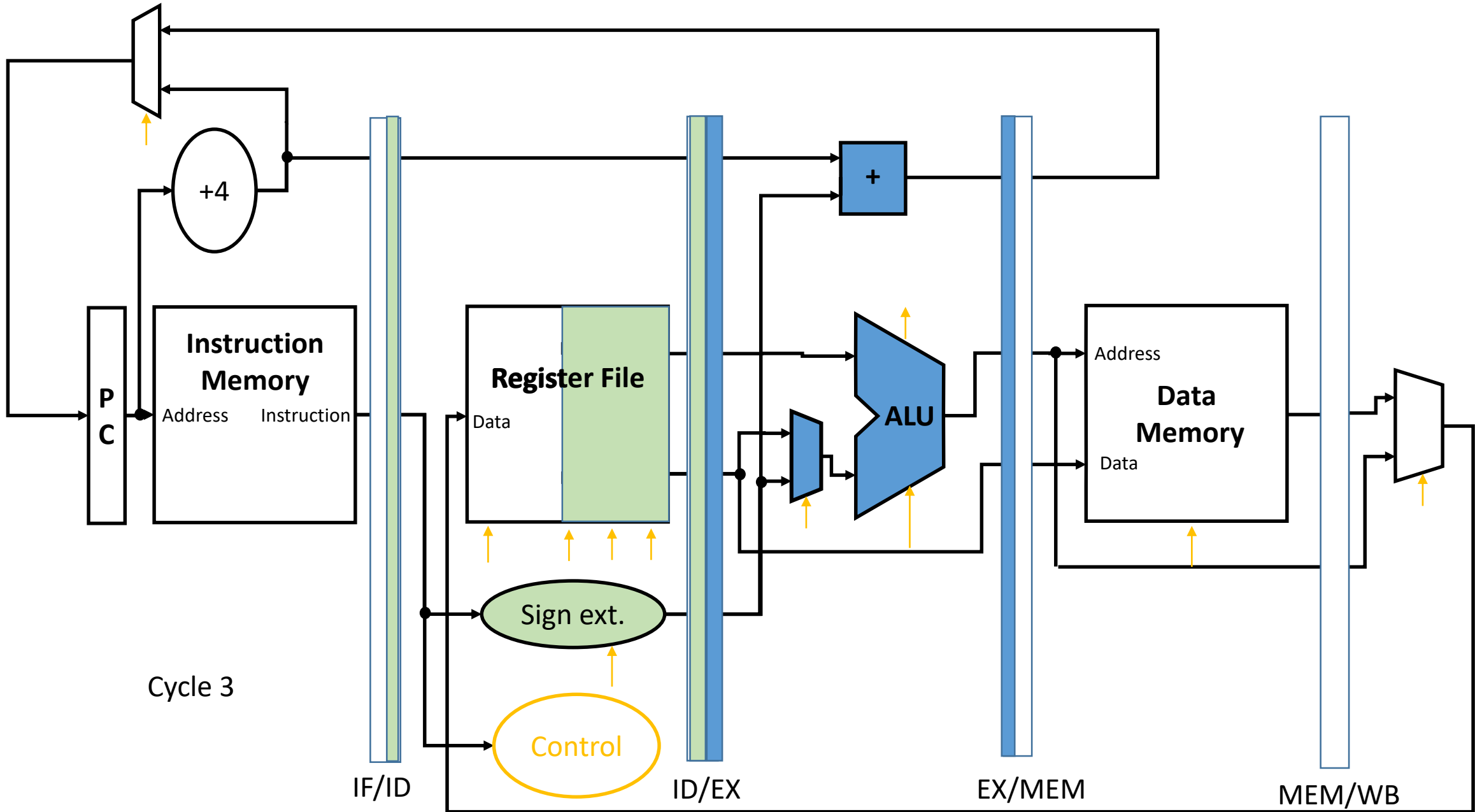
LW x1,100(x0)  
Instruction Decode



Cycle 2

ADD x2, x3, x4  
Instruction Decode

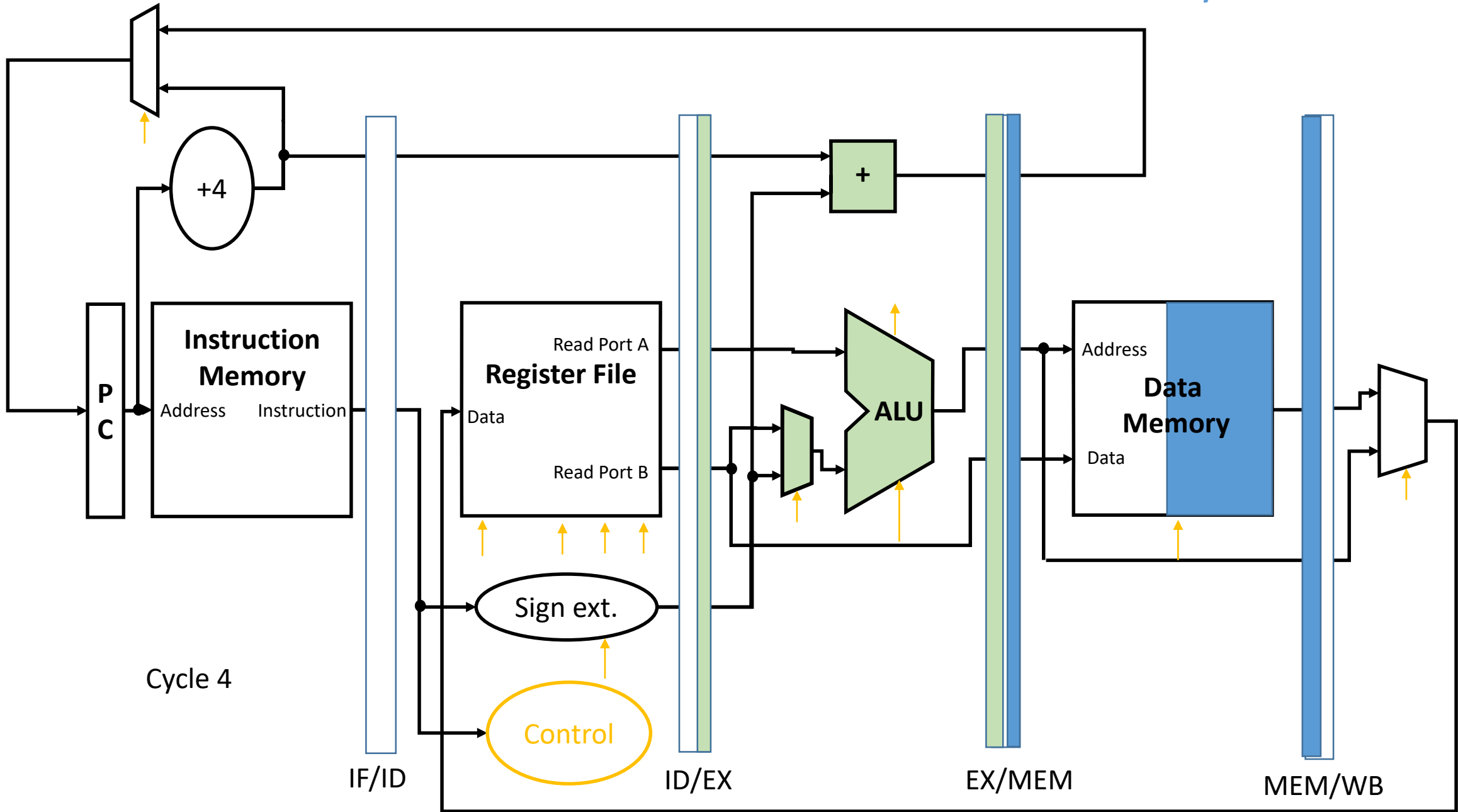
LW x1,100(x0)  
Execute



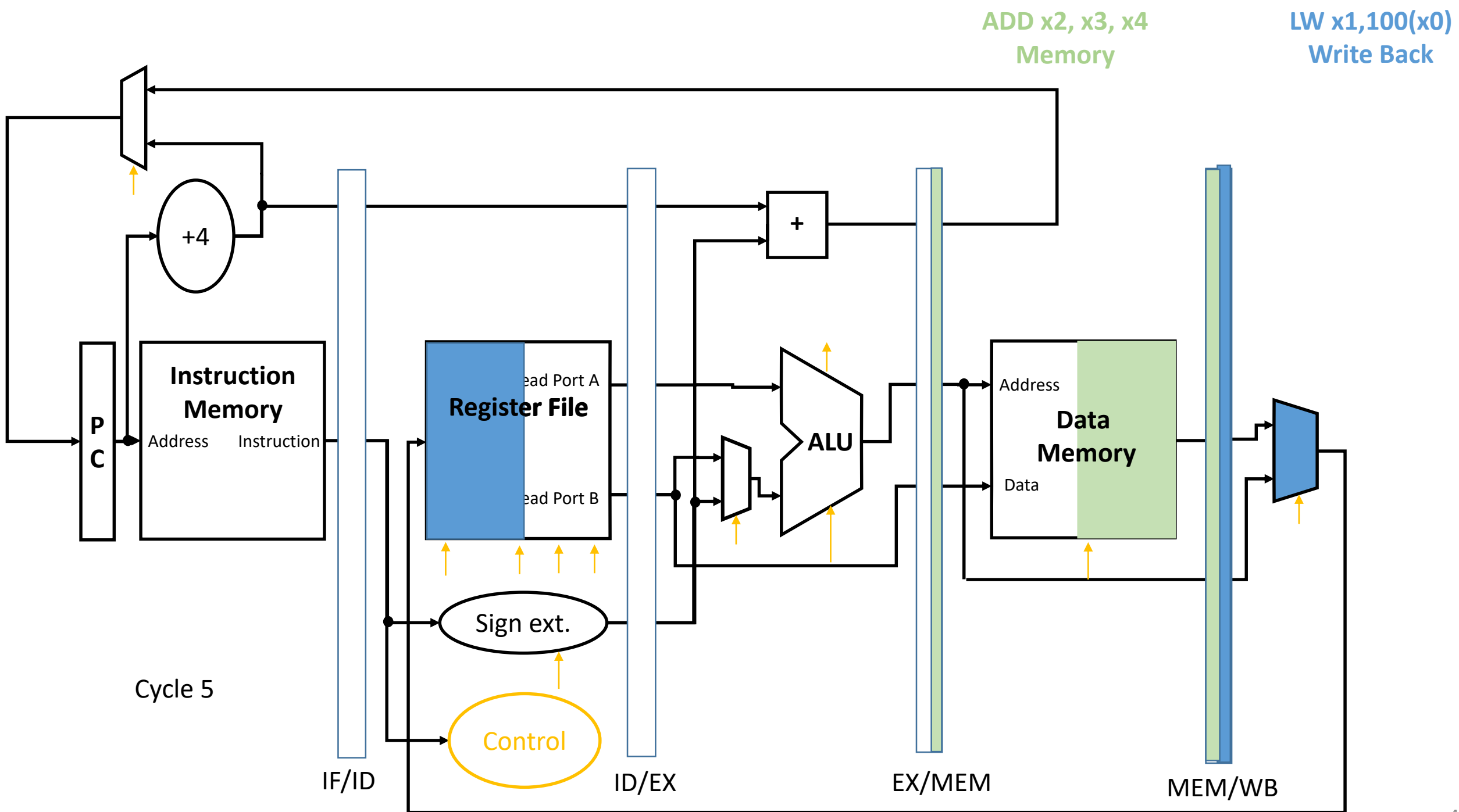
Cycle 3

ADD x2, x3, x4  
Instruction Execute

LW x1,100(x0)  
Memory

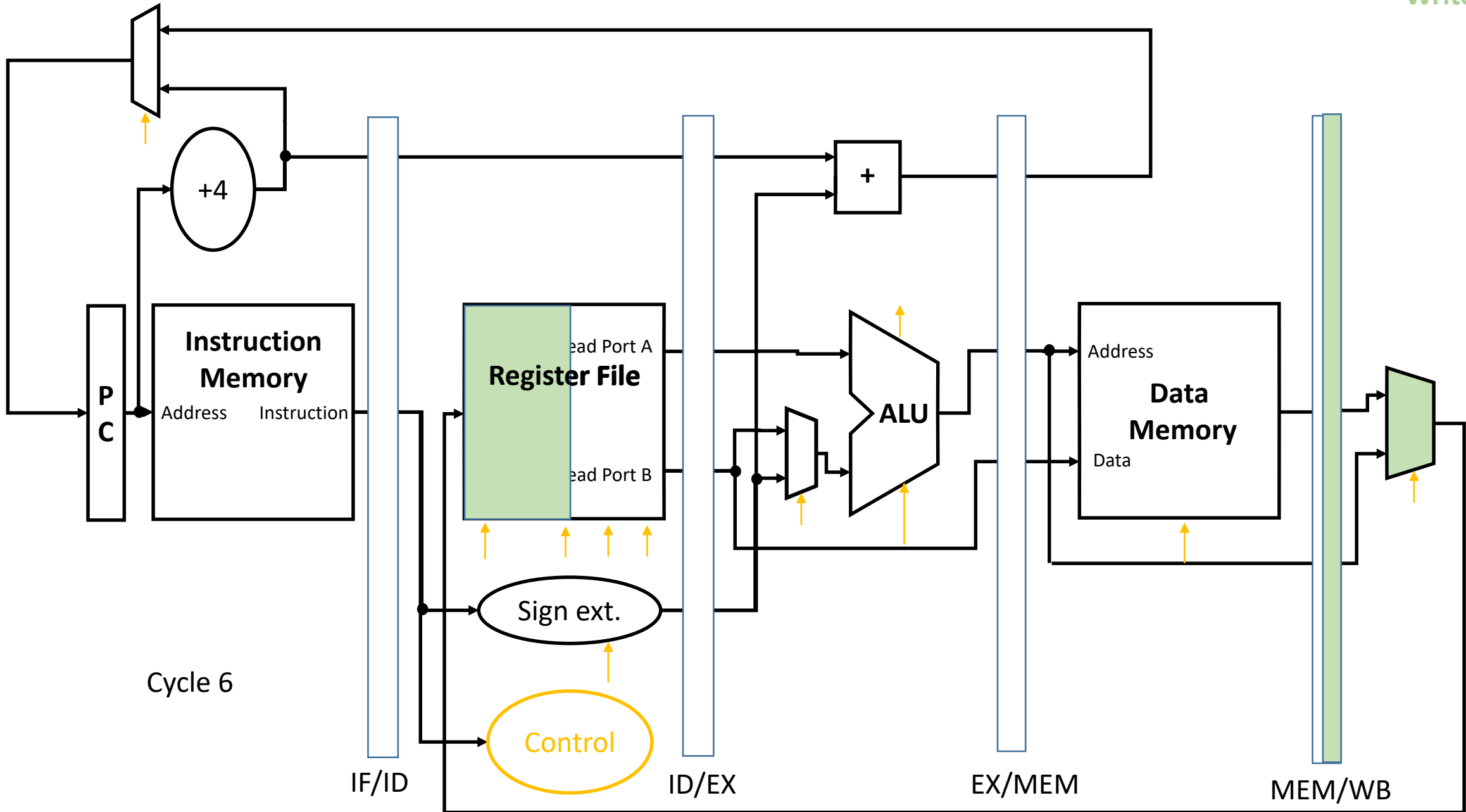


Cycle 4



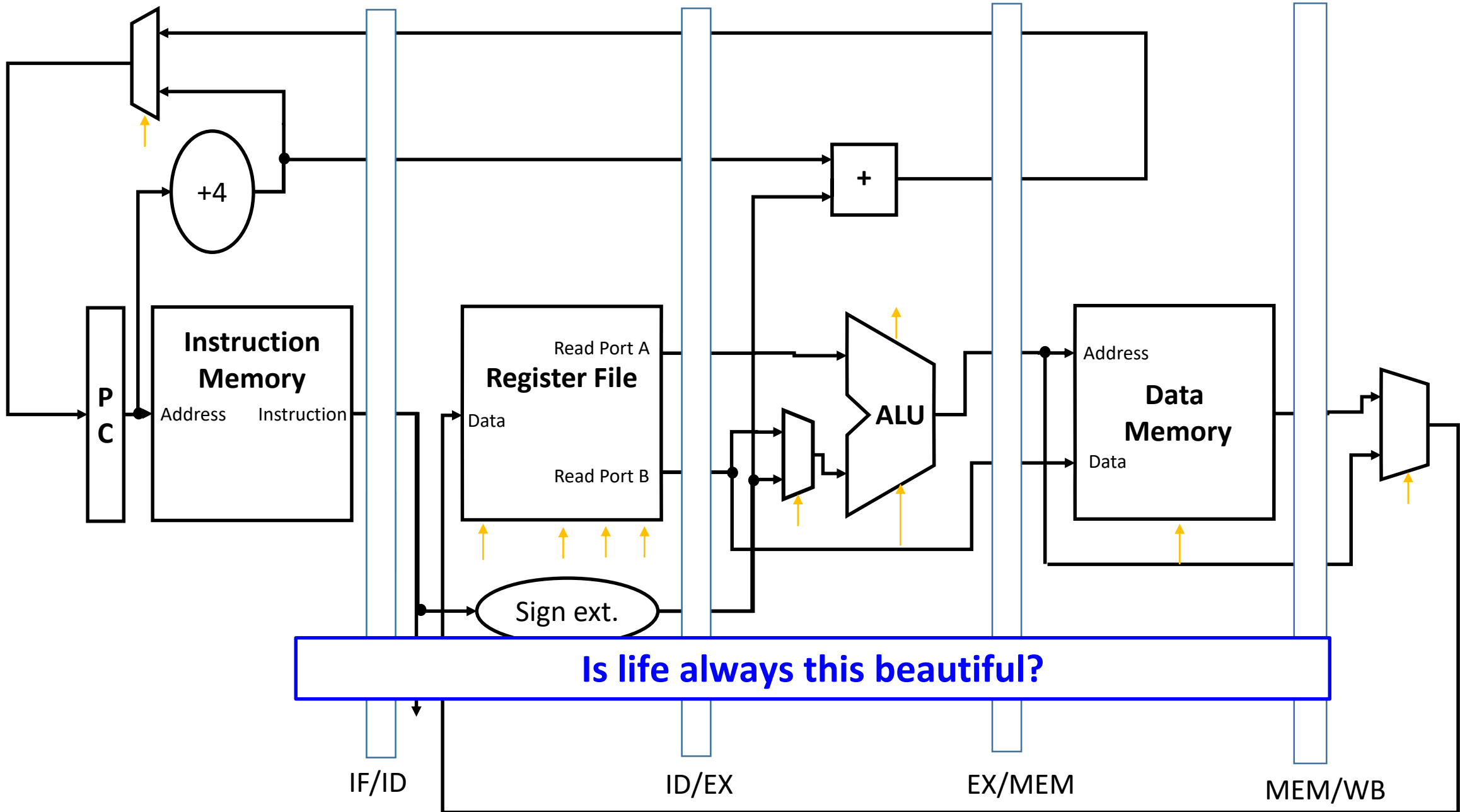


ADD x2, x3, x4  
Write Back



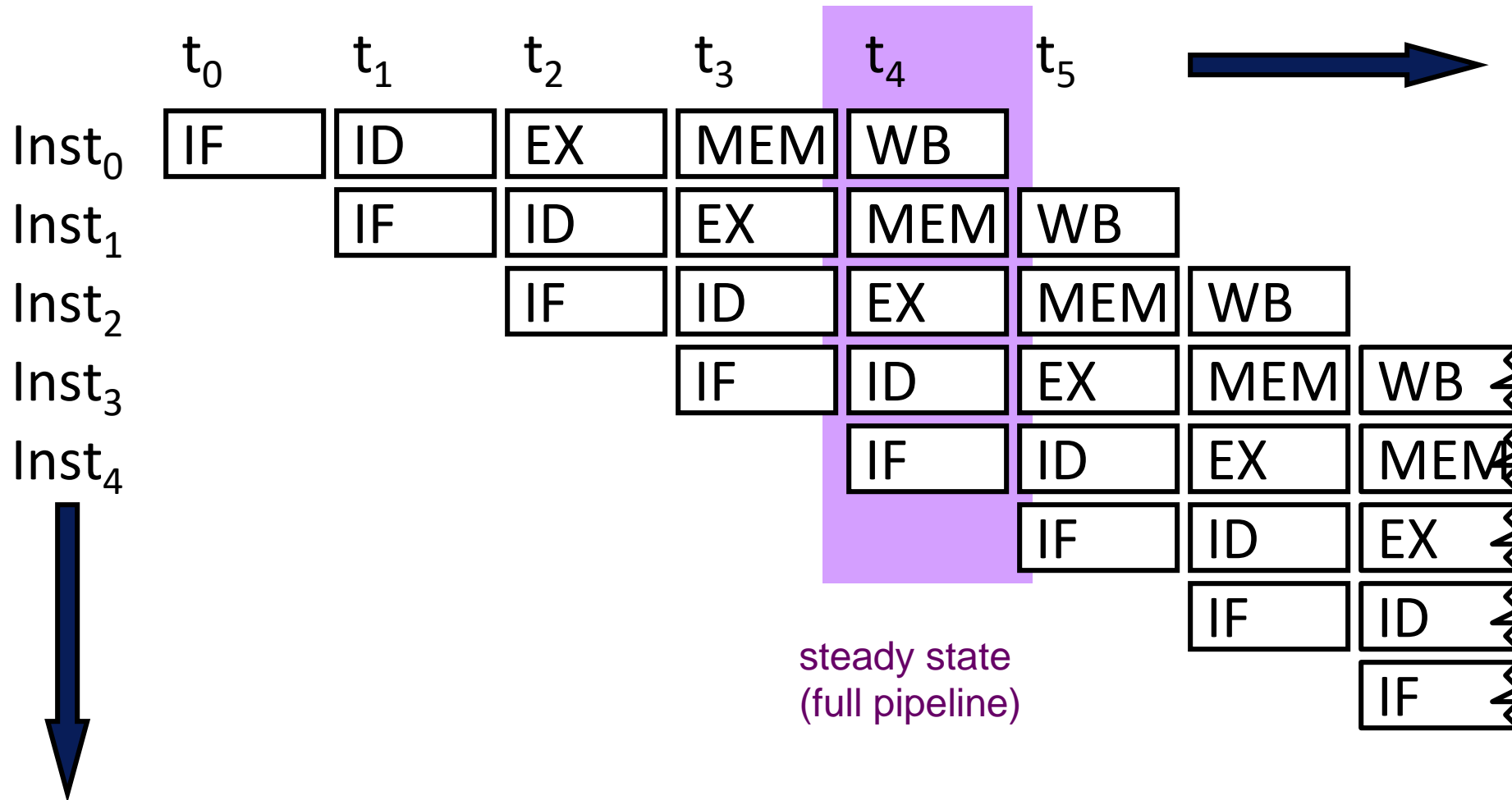
Cycle 6

# Pipelined Operation



Is life always this beautiful?

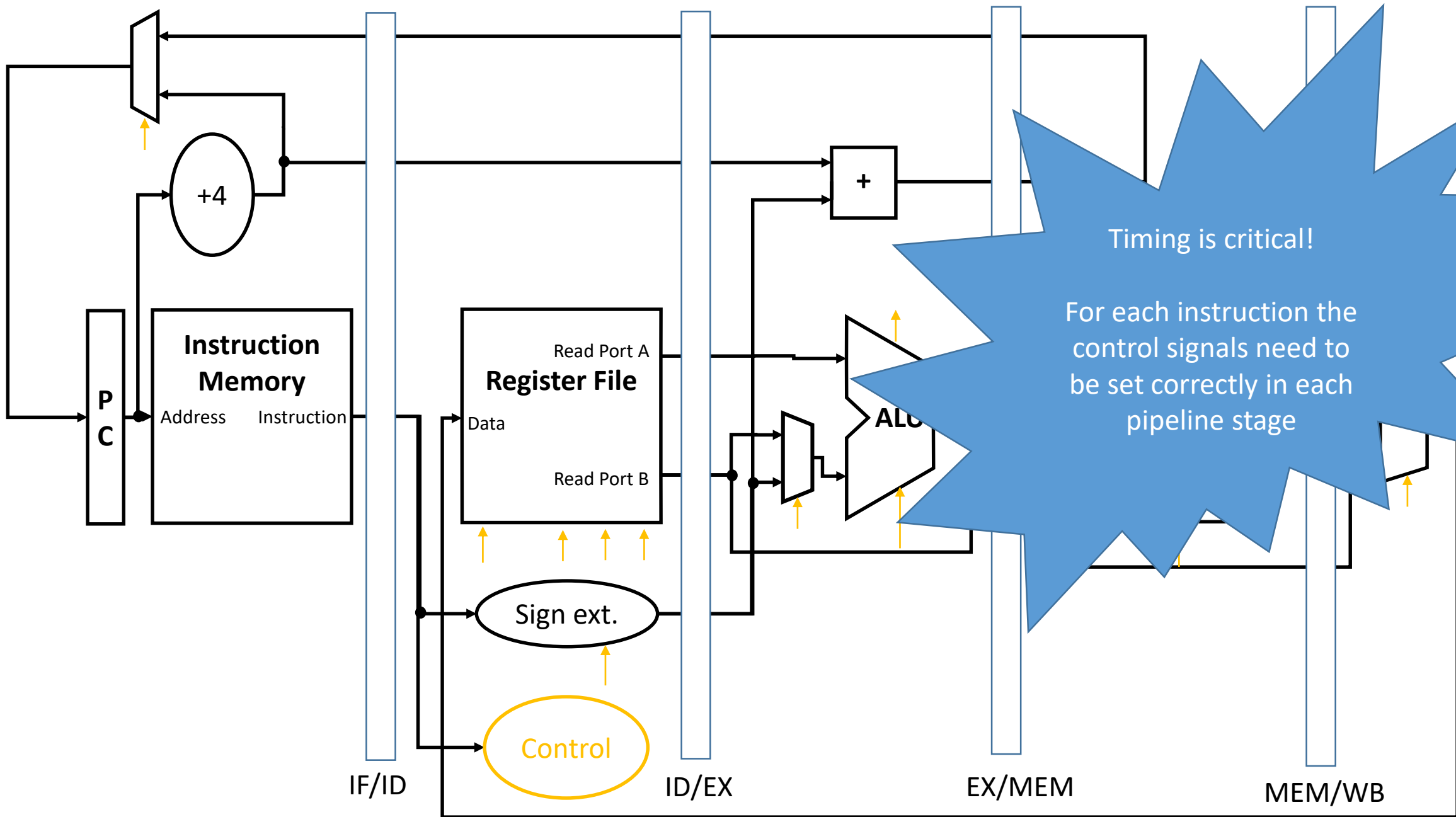
# Illustrating Pipeline Operation: Operation View



# 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					$I_0$	$I_1$	$I_2$	$I_3$	$I_4$	$I_5$	$I_6$

Note: There is the same number of control signals as in a single-cycle data path

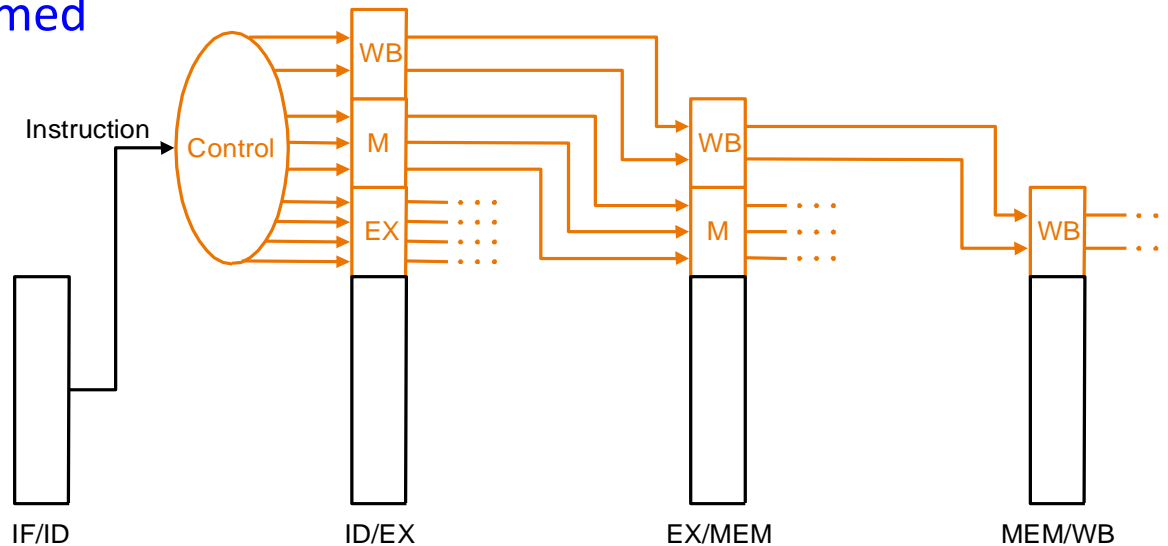


# Control Signals in a Pipeline

- For a given instruction

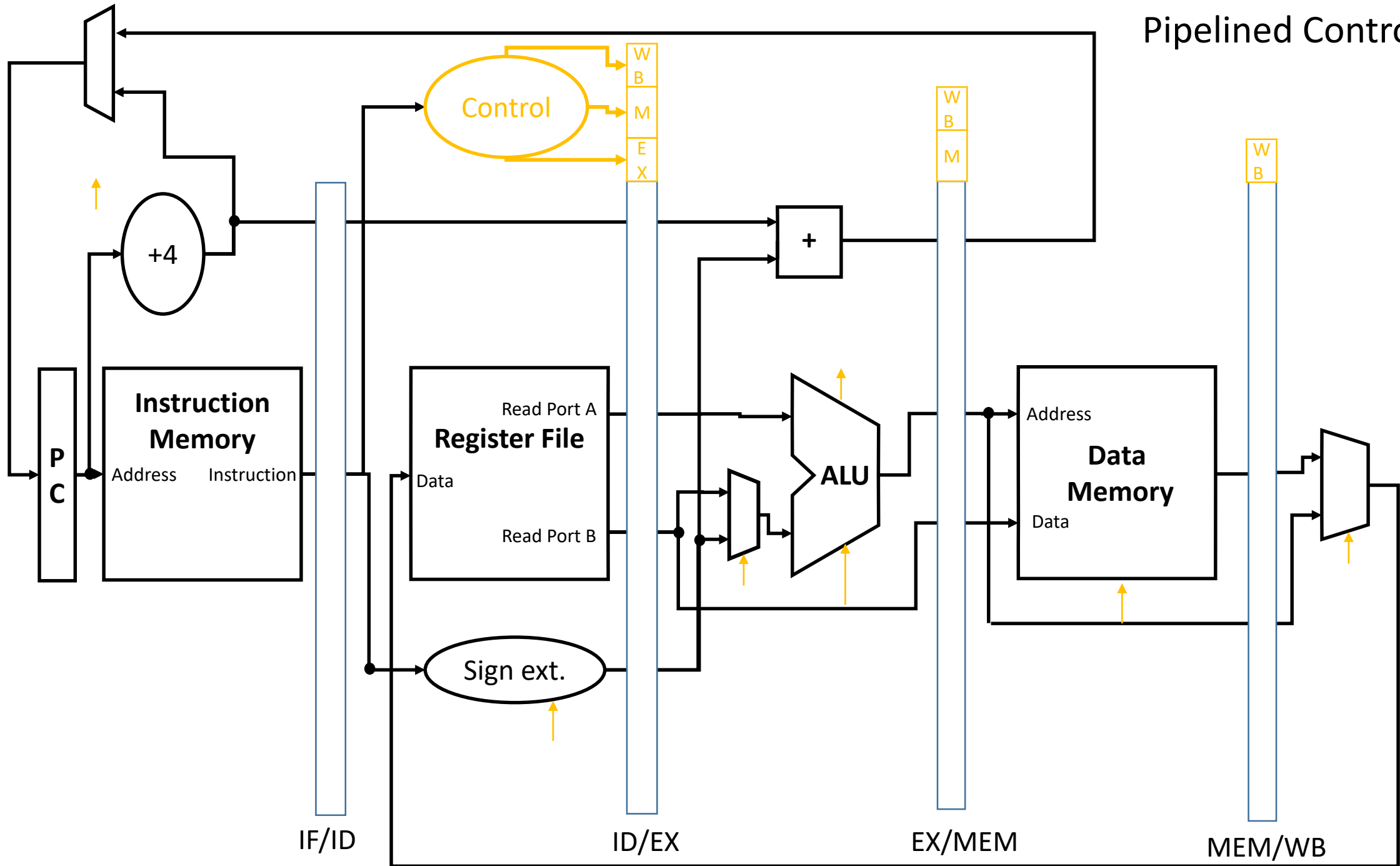
- same control signals as single-cycle, but
- control signals required at different cycles, depending on stage

⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed

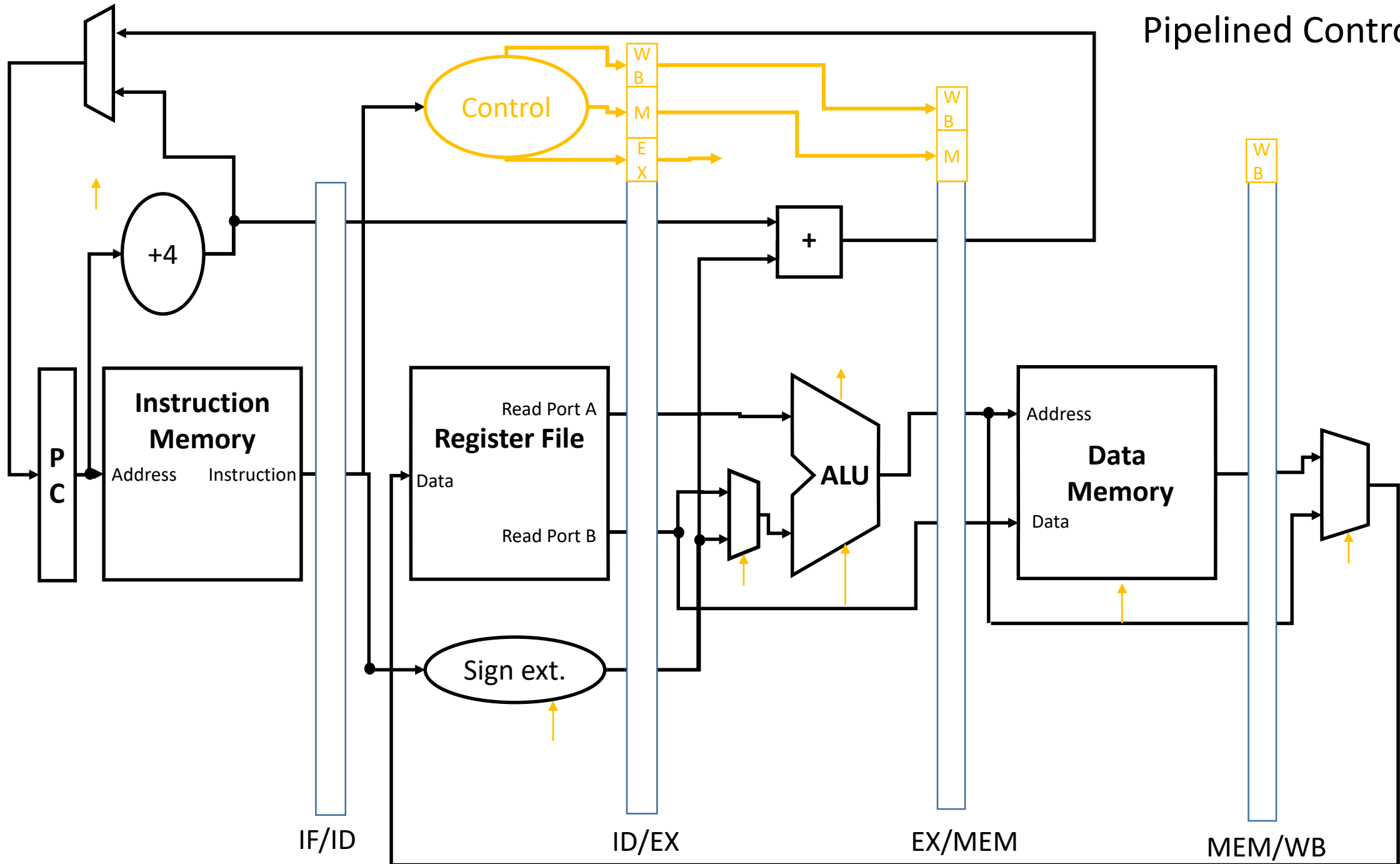


⇒ Option 2: carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

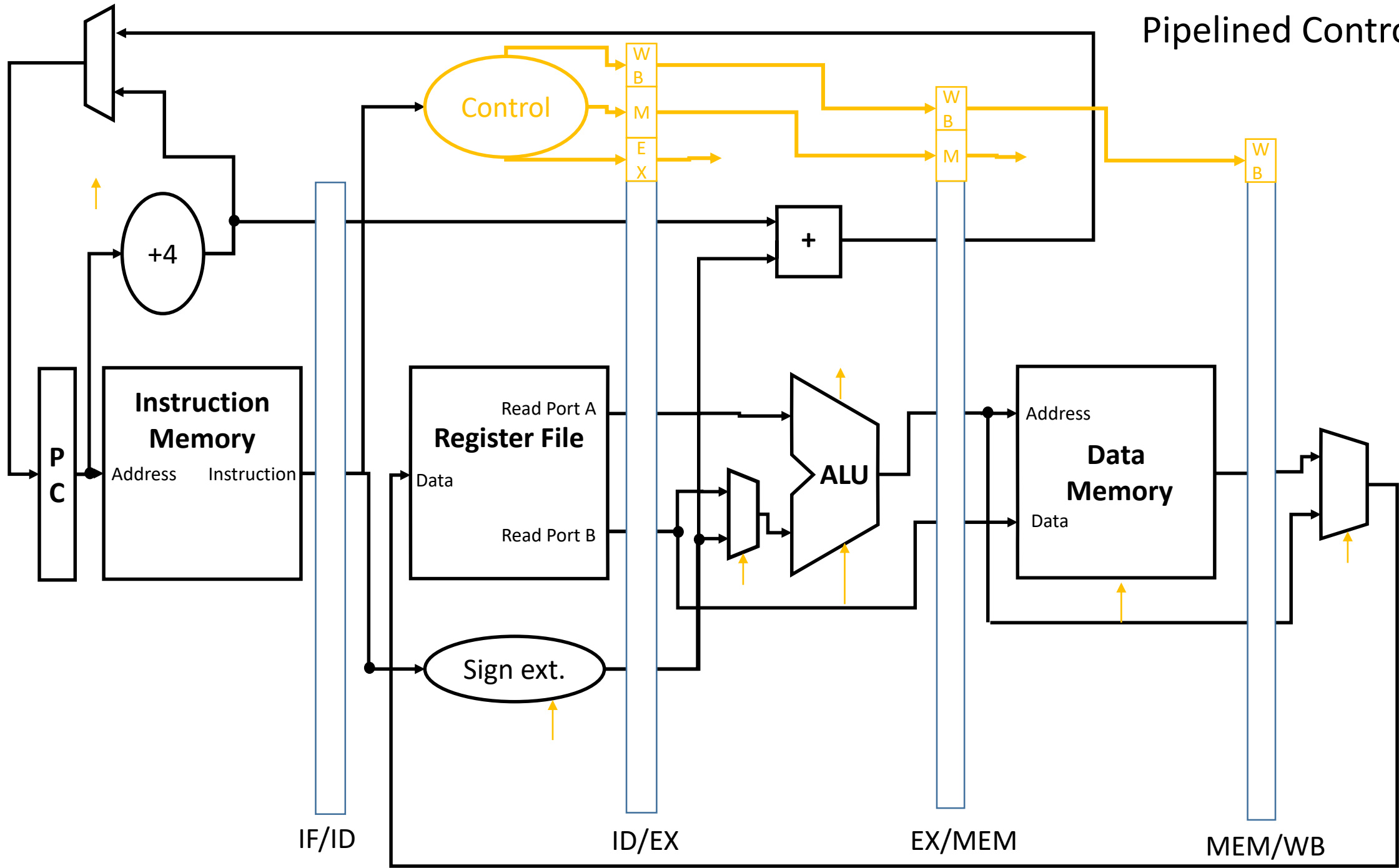
# Pipelined Control Signals



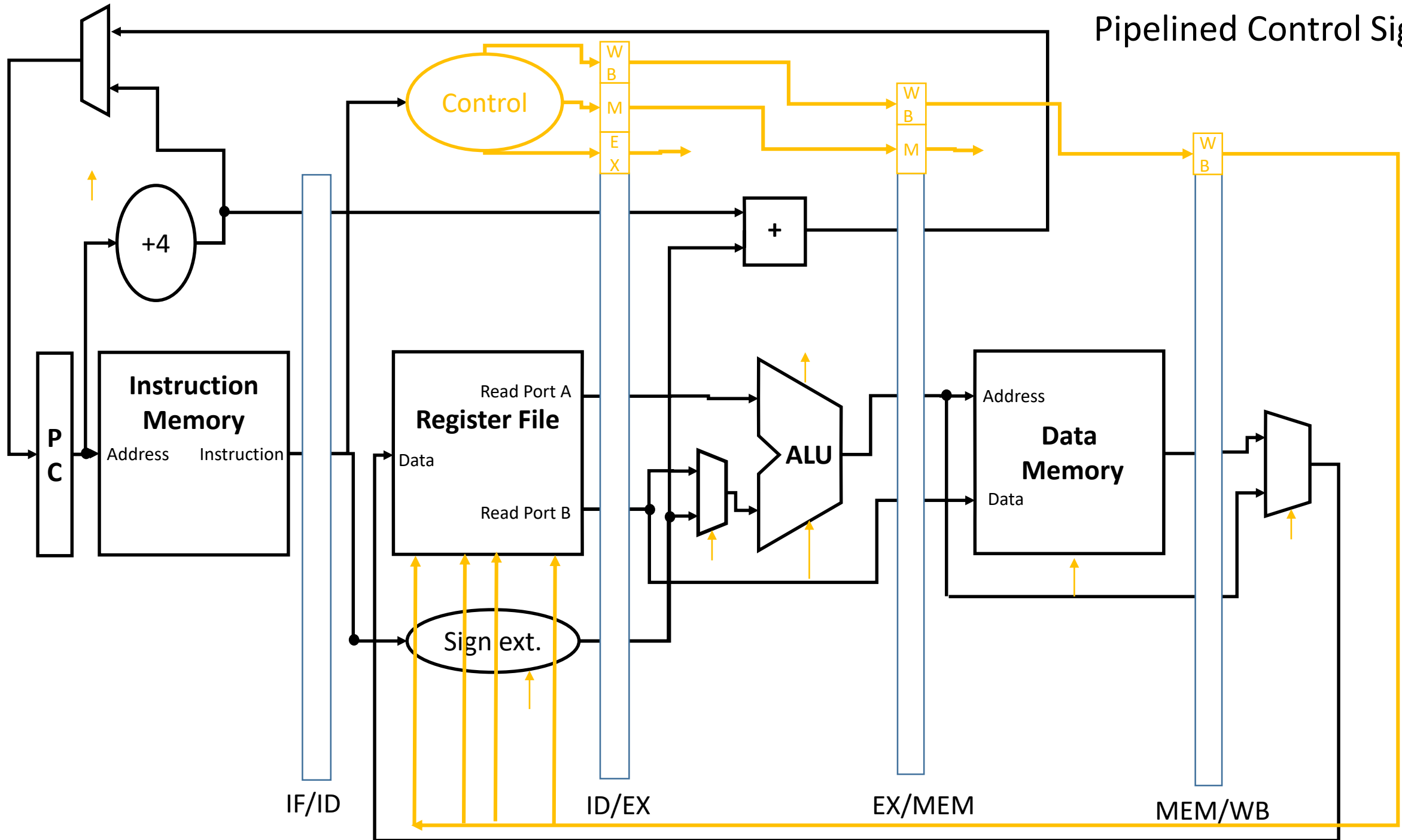
# Pipelined Control Signals







# 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

# 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)

## ■ 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)

## ■ 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)

# 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

# Causes of Pipeline *Stalls*

- Stall: A condition when the pipeline stops moving
- We need to stall the pipeline if either a needed resource or data value is not available
- Resource is not available
  - Resource contention (e.g. caused by long-latency (multi-cycle) operations)
- Data is not available
  - Dependences between instructions (also called “dependency” or “hazard”)
    - Data
    - Control

# Data Dependence Handling

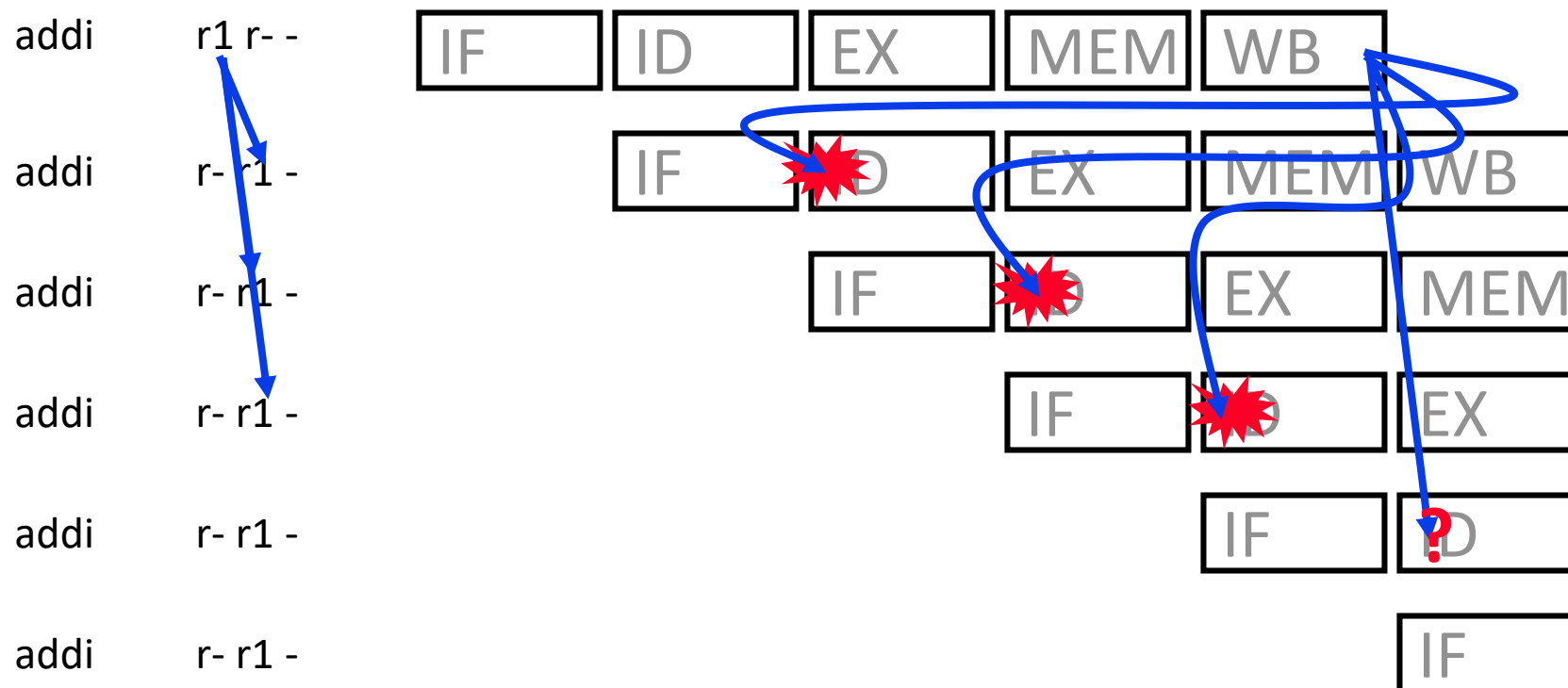
# Read-After-Write Dependency





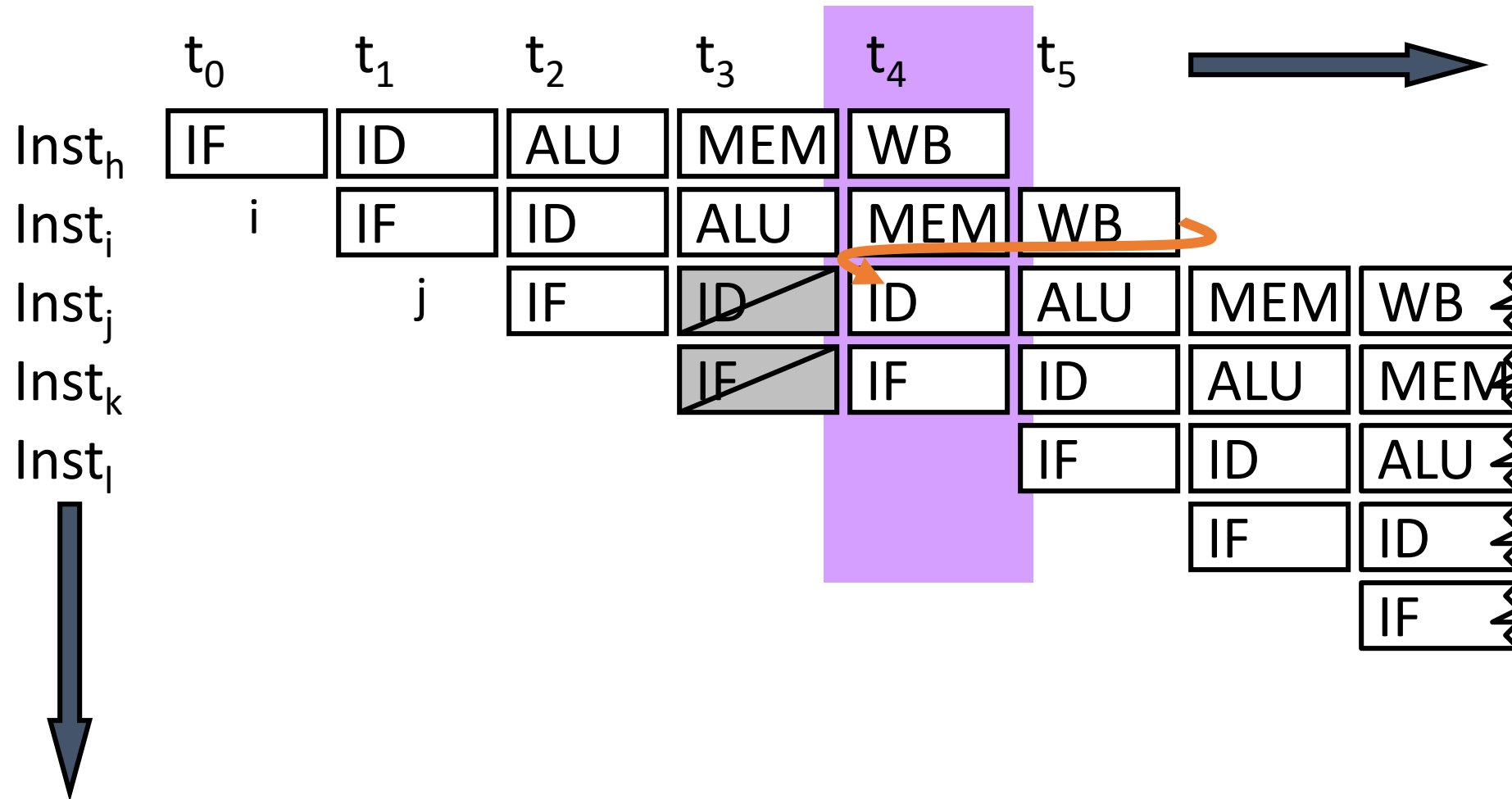
# RAW Dependence Handling

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?

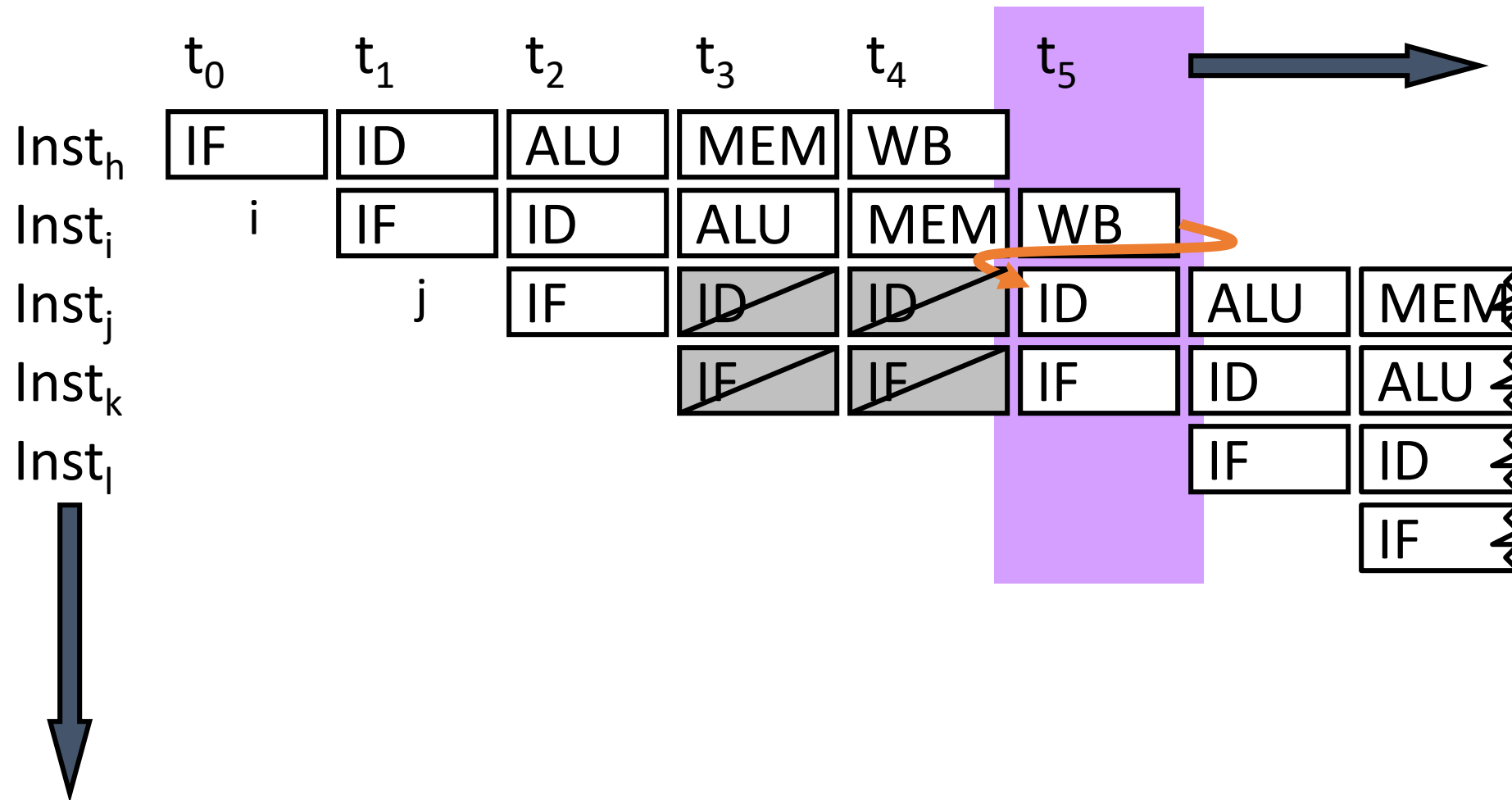




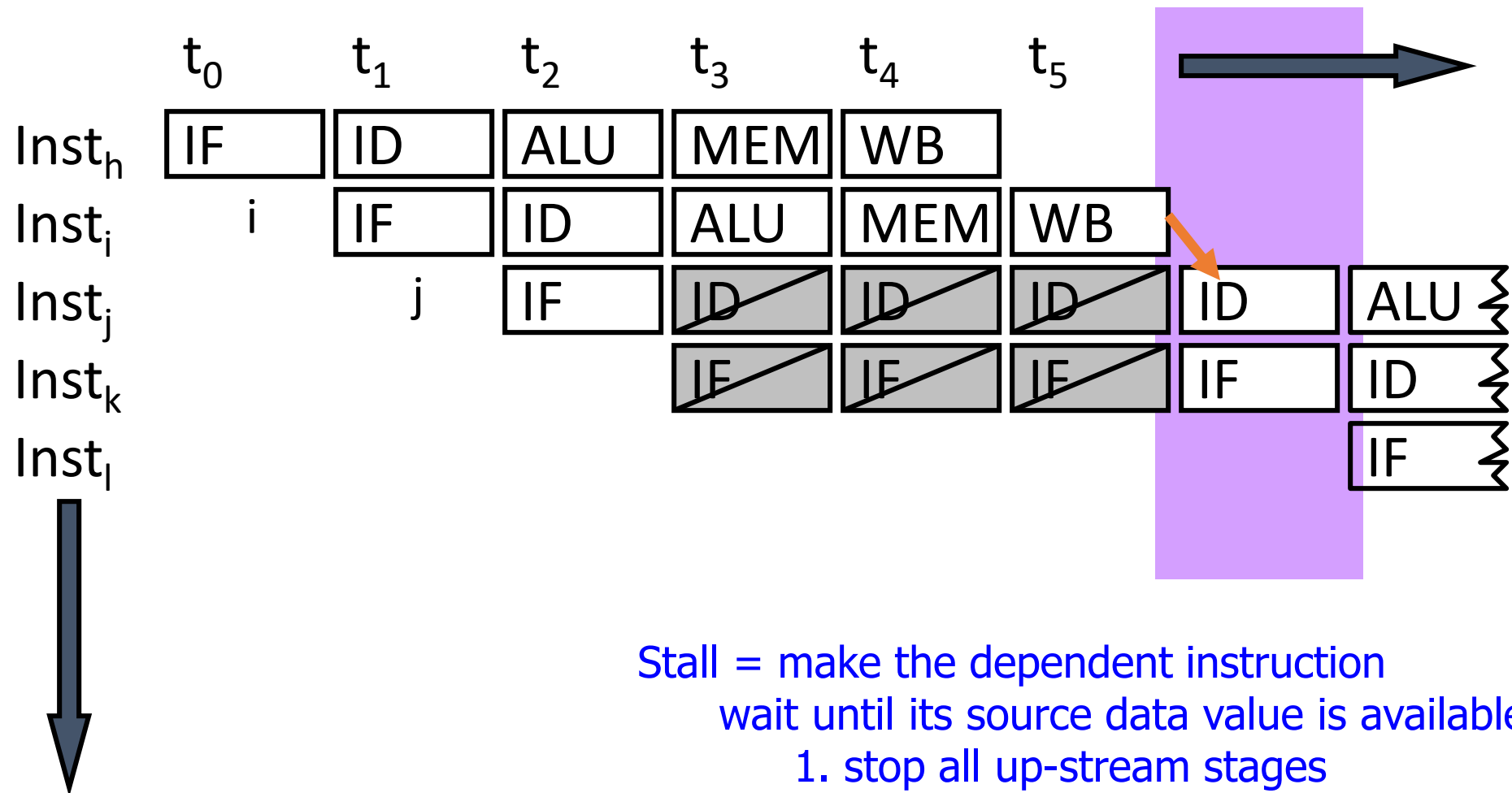
# Pipeline Stall: Resolving Data Dependence



# Pipeline Stall: Resolving Data Dependence



# Pipeline Stall: Resolving Data Dependence



# Example of Dependence Detection

- **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 registers are Valid
  - Yes: No need to stall... No dependence
  - No: Stall the instruction

# Once You Detect the Dependence in Hardware

- What do you do?
- 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

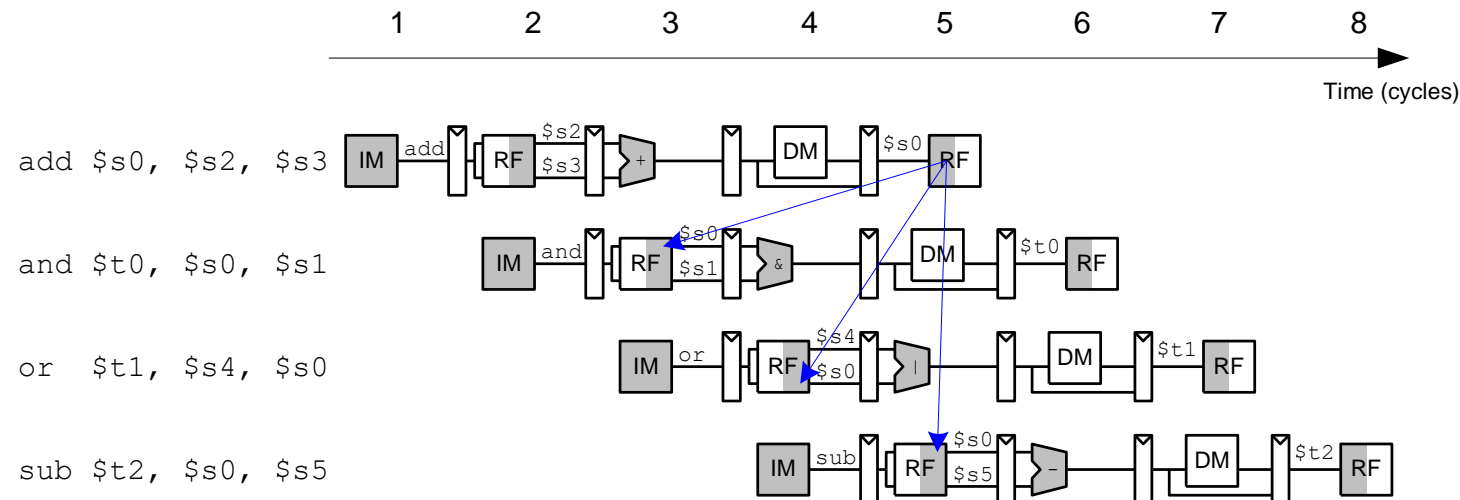




# RAW Data Dependence Example

■ One instruction writes a register (\$s0) and next instructions read this register => read after write (RAW) dependence.

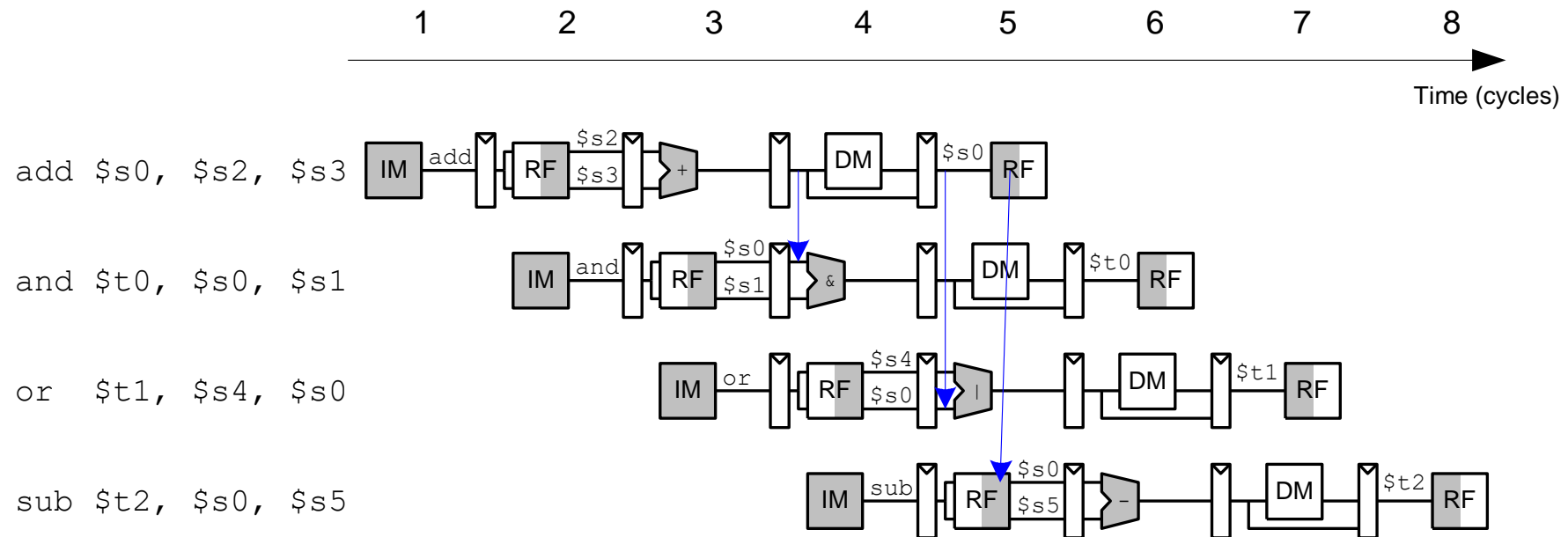
- **add** writes into \$s0 in cycle 5
- **and** requires to read \$s0 on cycle 3
- **or** requires to read \$s0 on cycle 4
- **sub** requires to read \$s0 in cycle 5



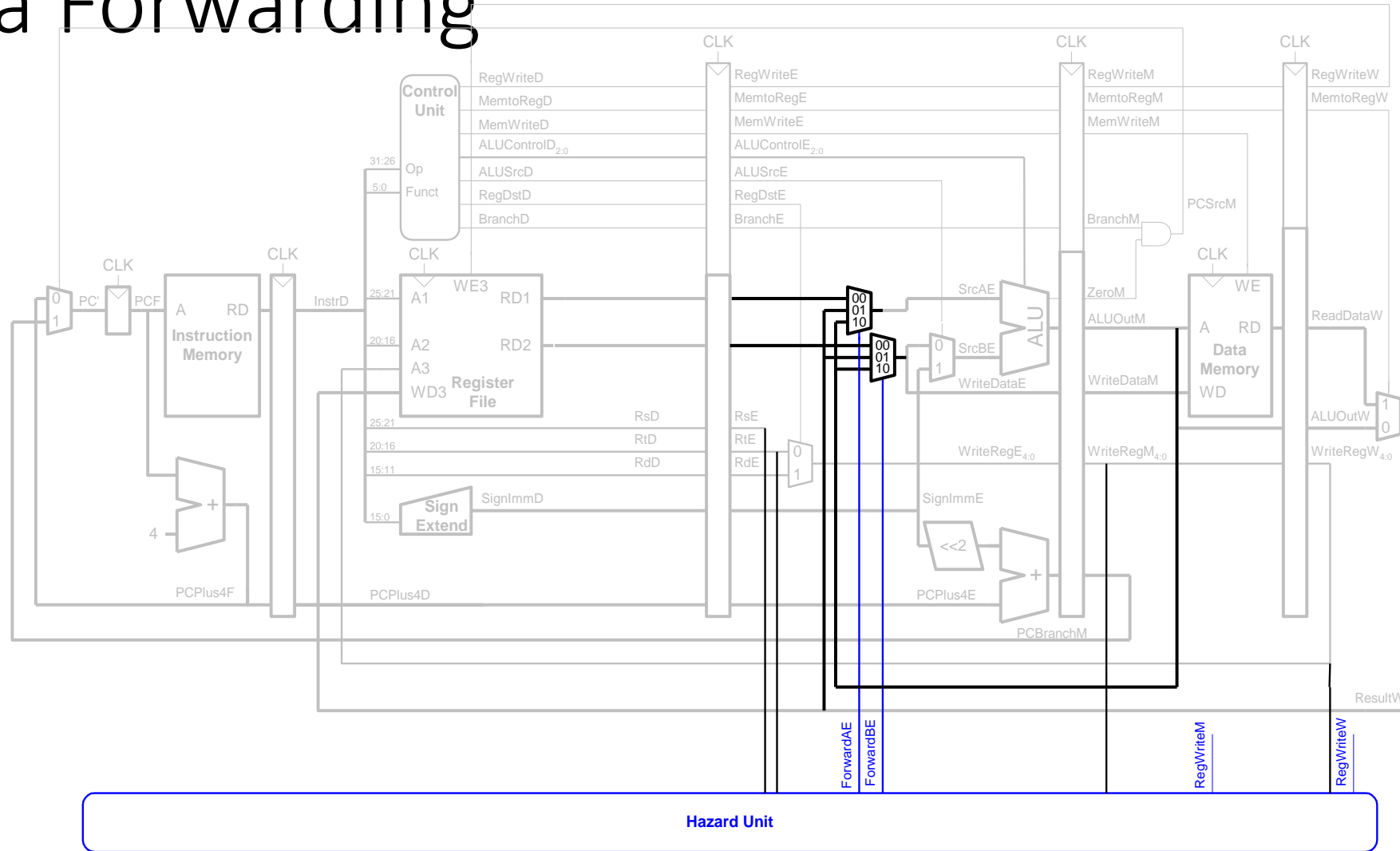
# Data Forwarding

- Also called **Data Bypassing**
- **Forward the result value to the dependent instruction as soon as the value is available**
- **Basic Idea**
  - Data values are supplied to dependent instruction as soon as it is available
  - Instruction executes when all its operands are available

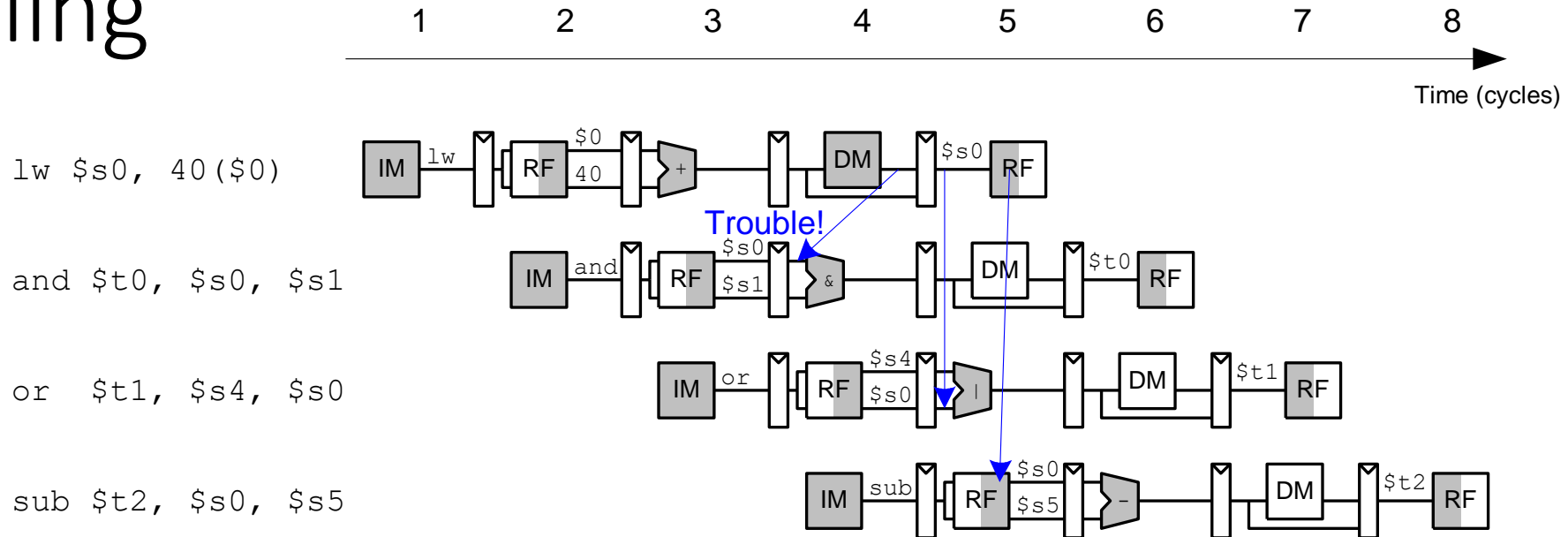
# Data Forwarding



# Data Forwarding



# Stalling



- Forwarding is sufficient to resolve RAW data dependences
- *but ... There are cases when forwarding is not possible due to pipeline design and instruction latencies*
- The `lw` instruction *does not finish* reading data until the end of the Memory stage,
- Therefore its result *cannot be forwarded* to the Execute stage of the next instruction.

# Stalling

