Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 5: Pipelining

Winter 2020/2021

TU Graz

Stefan Mangard, www.iaik.tugraz.at

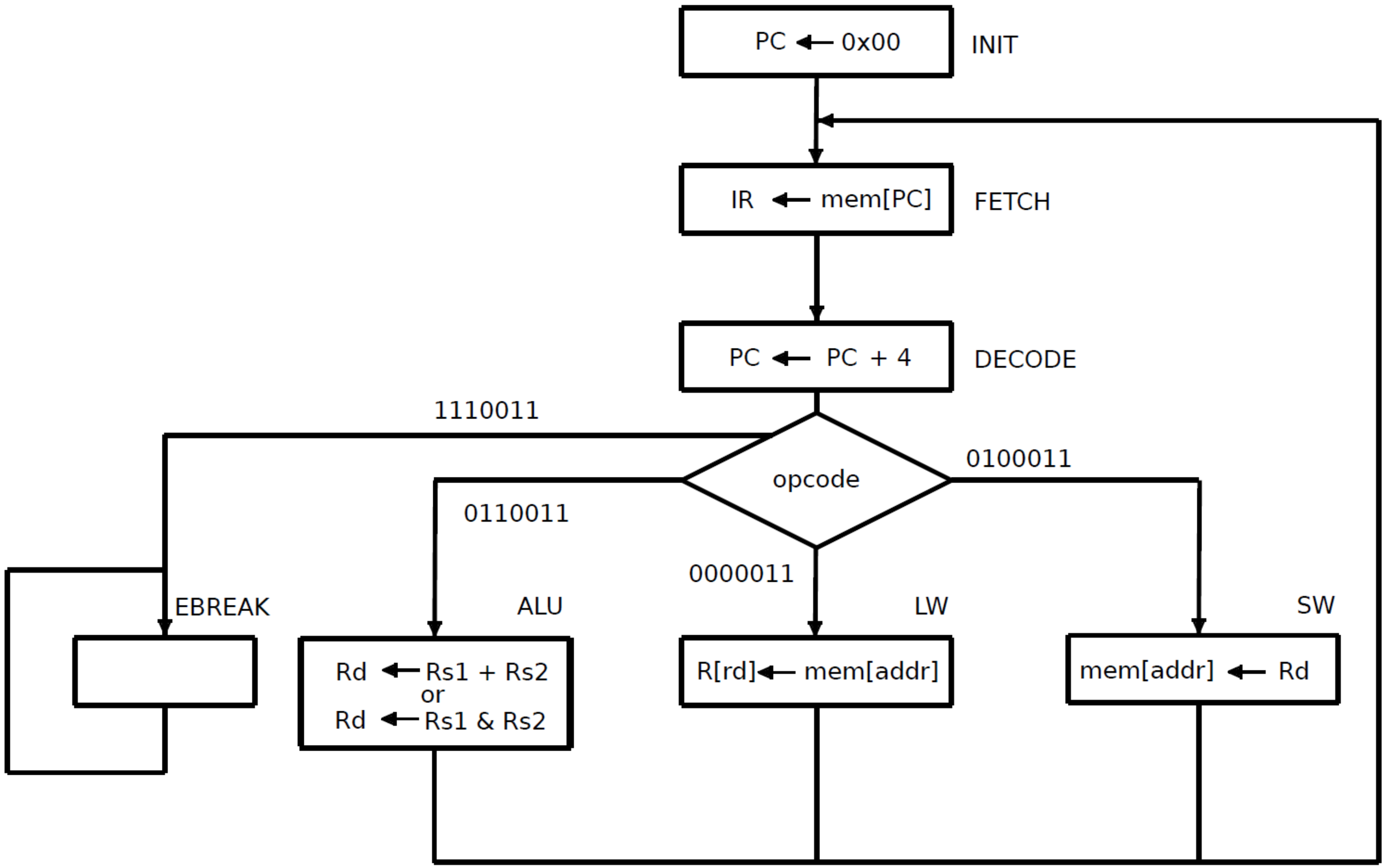
Notes on this Slide Set

- This part of the lecture is based on slides by Prof. Onur Mutlu (ETH Zürich)

- The slide sequence has been changed 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 ASM



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 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

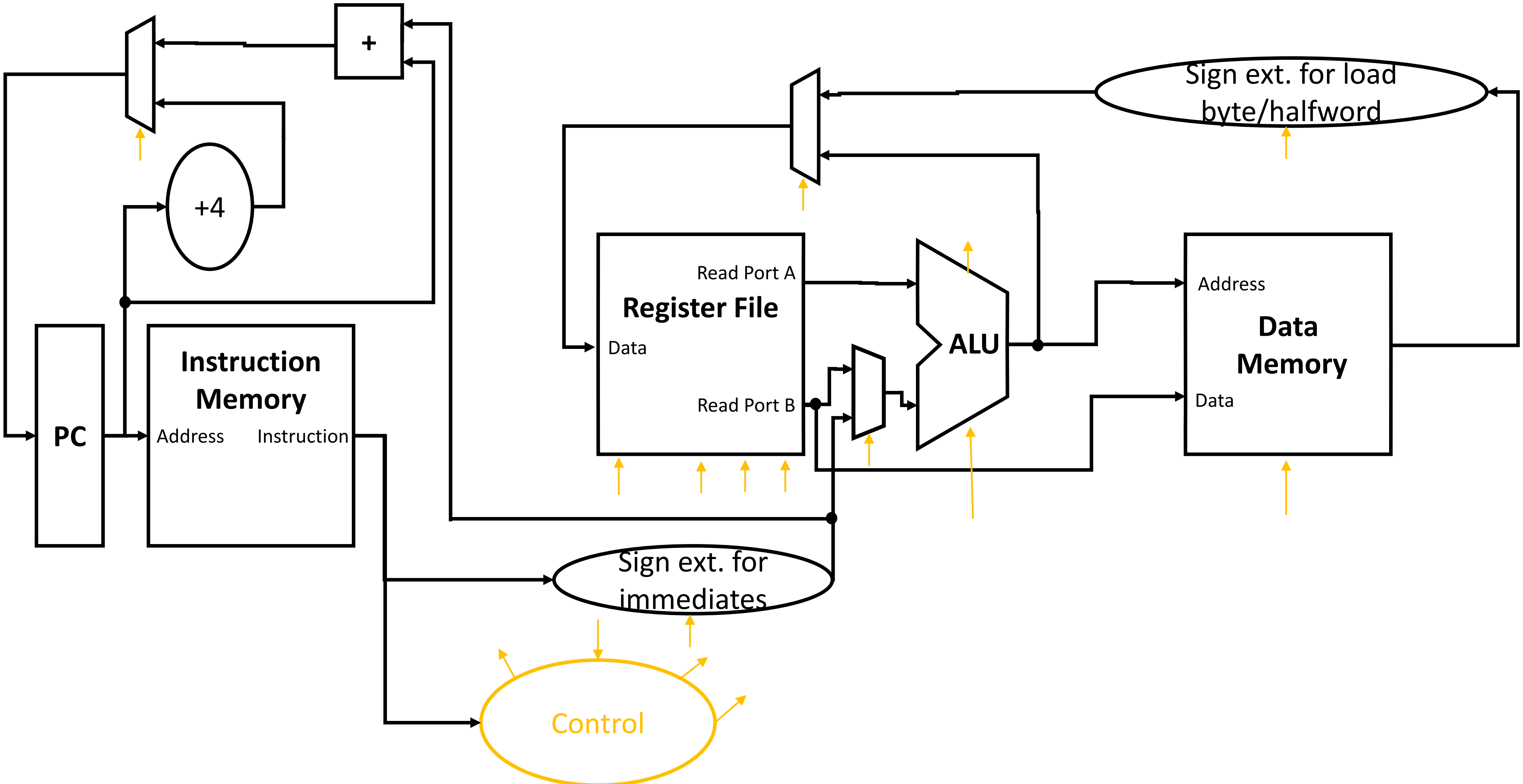
- Goal: In order to build a fast design, the goal is to build a design, where each instruction needs only as much time as it actually needs

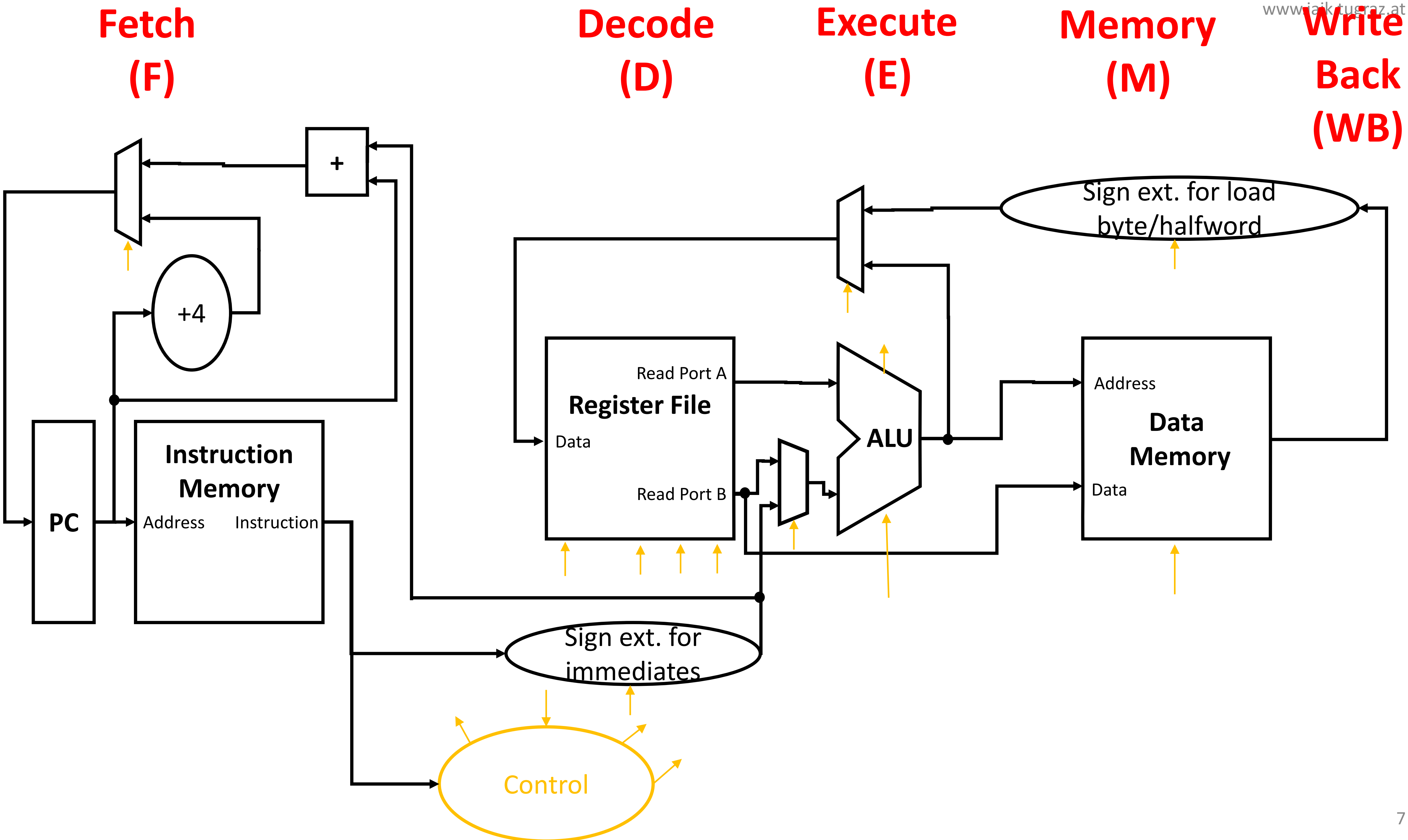
Basic Idea of Multicycle Architectures

- 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
  - Multiple state transitions per cycle
  - Each instruction leads to a different sequence of states (longer / shorter)

High-Level Overview (Single Cycle Datapath)




Fetch
(F)
Decode
(D)
Execute
(E)
Memory
(M)
Write
Back
(WB)
+
+4
Sign ext. for load byte/halfword
Read Port A
Register File
Data
Read Port B
ALU
Address
Data
Memory
Data
PC
Instruction
Memory
Address
Instruction
Sign ext. for immediates
Control

Can We Do Better?

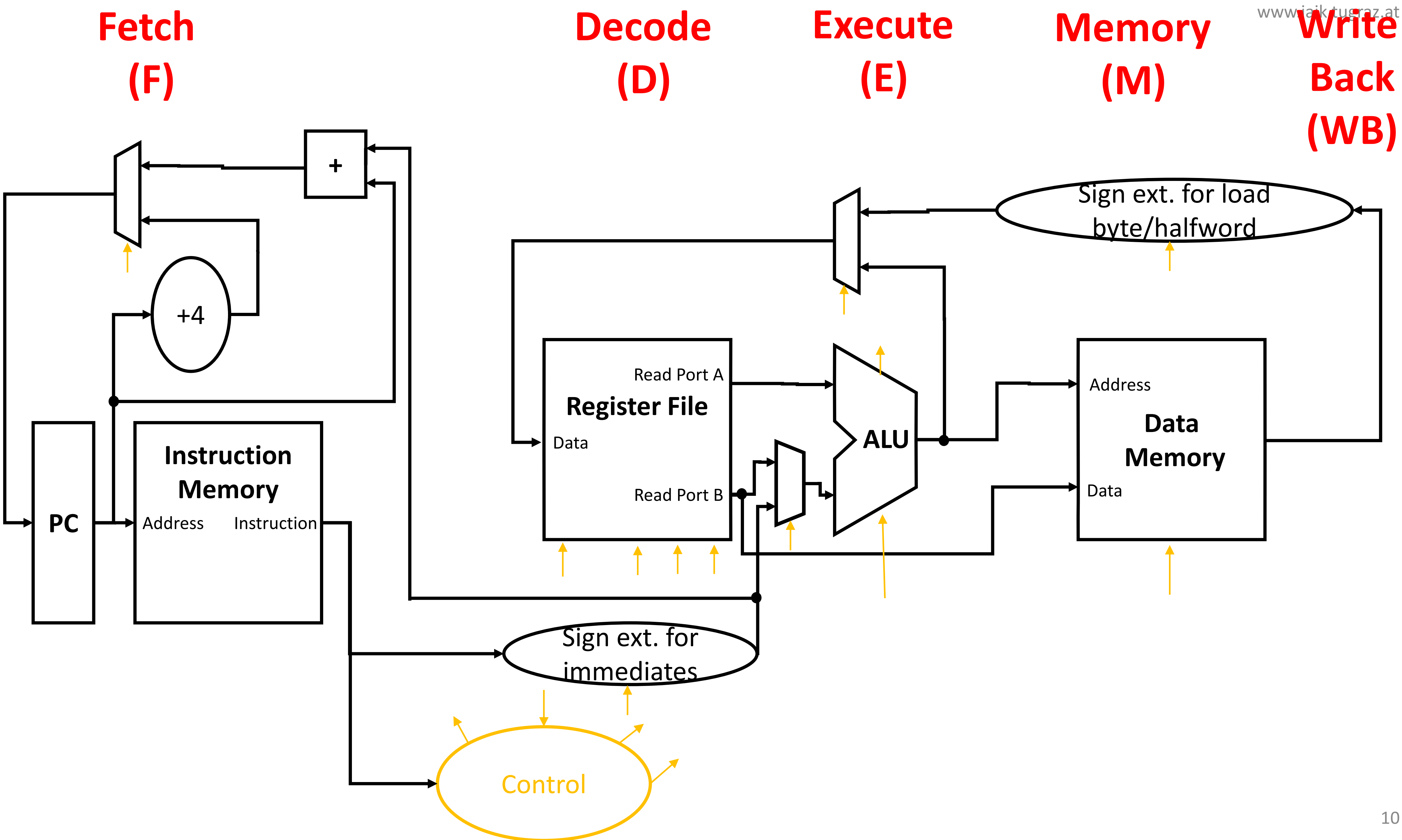
- What limitations do you see with the multi-cycle design?

- 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)
+
+4
Sign ext. for load byte/halfword
Read Port A
Register File
Data
Read Port B
ALU
Address
Data
Memory
Data
PC
Instruction
Memory
Address
Instruction
Sign ext. for immediates
Control

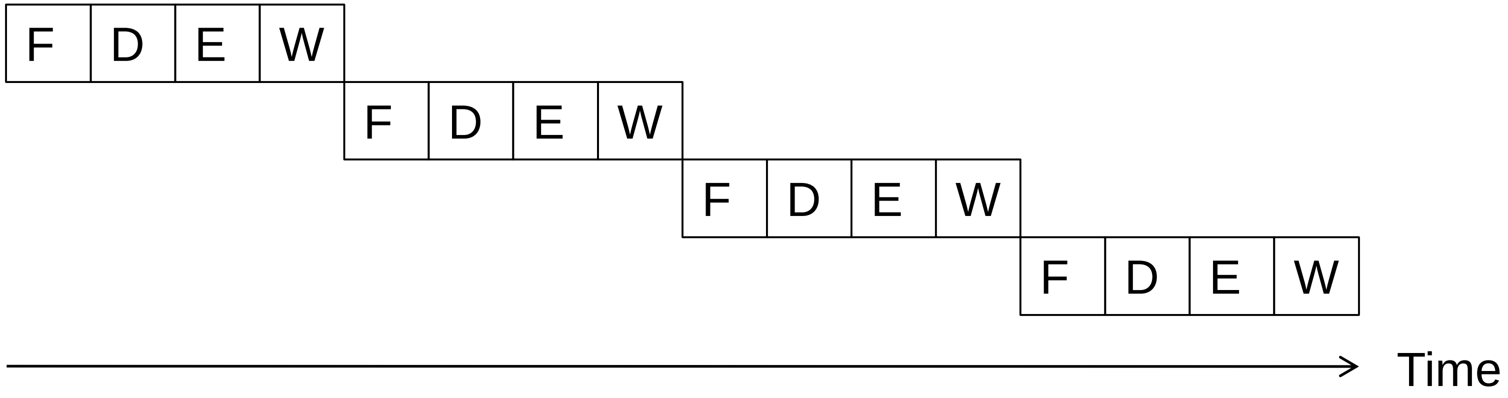
Pipelining

Pipelining: Basic Idea

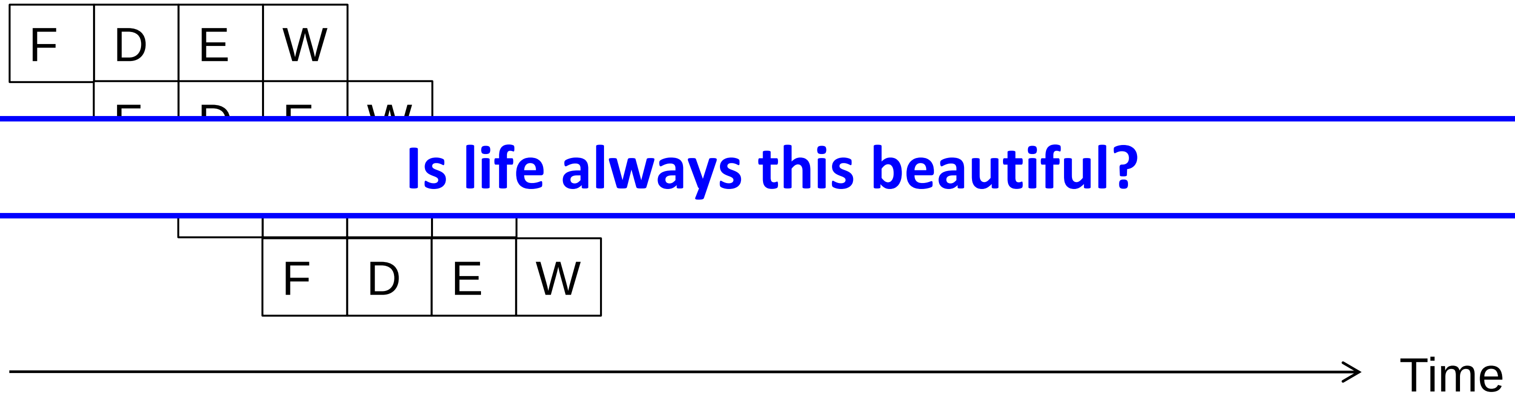
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each 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 (no memory needed)

- Multi-cycle: 4 cycles per instruction

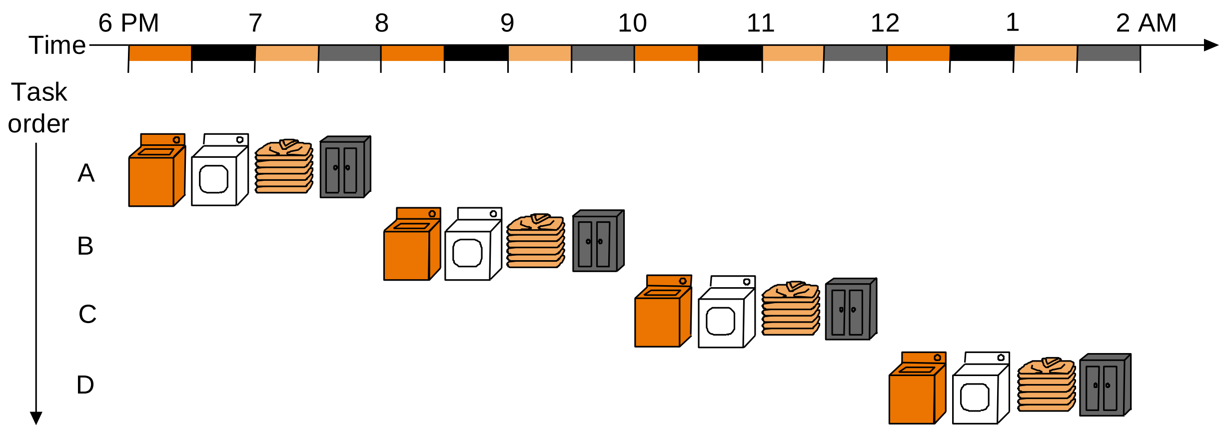


- Pipelined: 4 cycles per 4 instructions (steady state)



Is life always this beautiful?

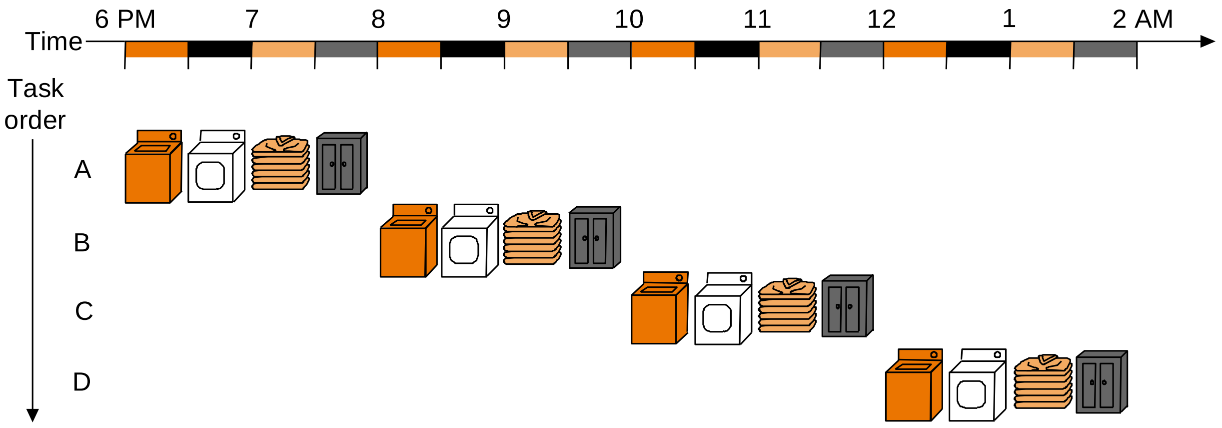
The Laundry Analogy

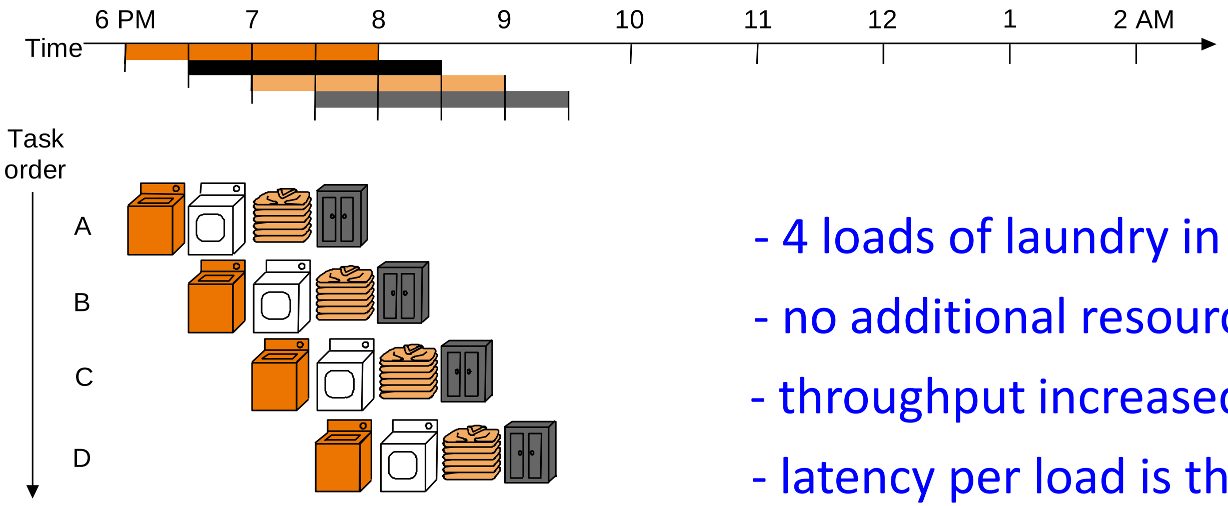


- “place one dirty load of clothes in the washer”
- “when the washer is finished, place the wet load in the dryer”
- “when the dryer is finished, take out the dry load and fold”
- “when folding is finished, ask your roommate (??) to put the clothes away”

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

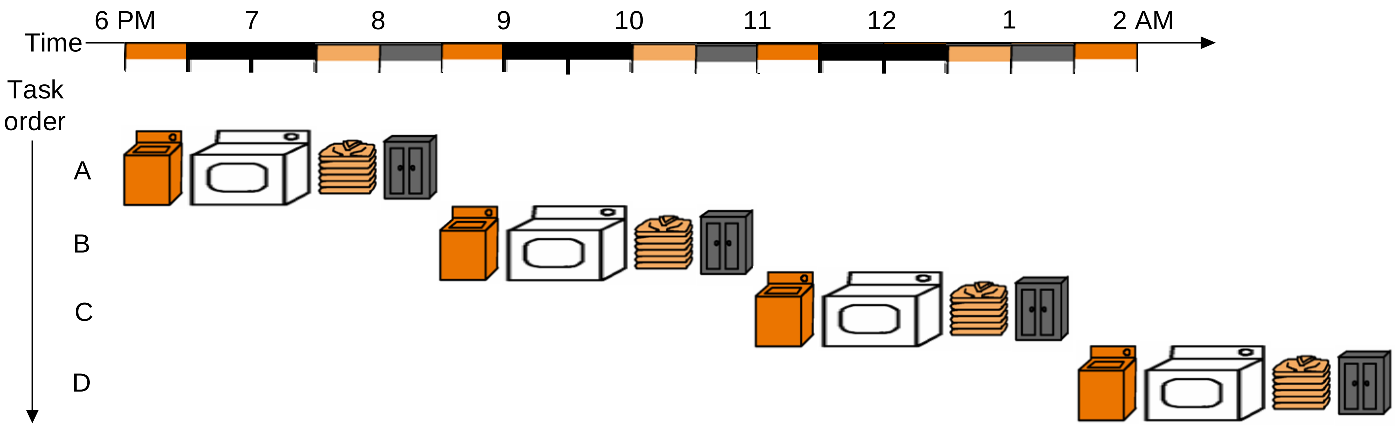
Pipelining Multiple Loads of Laundry

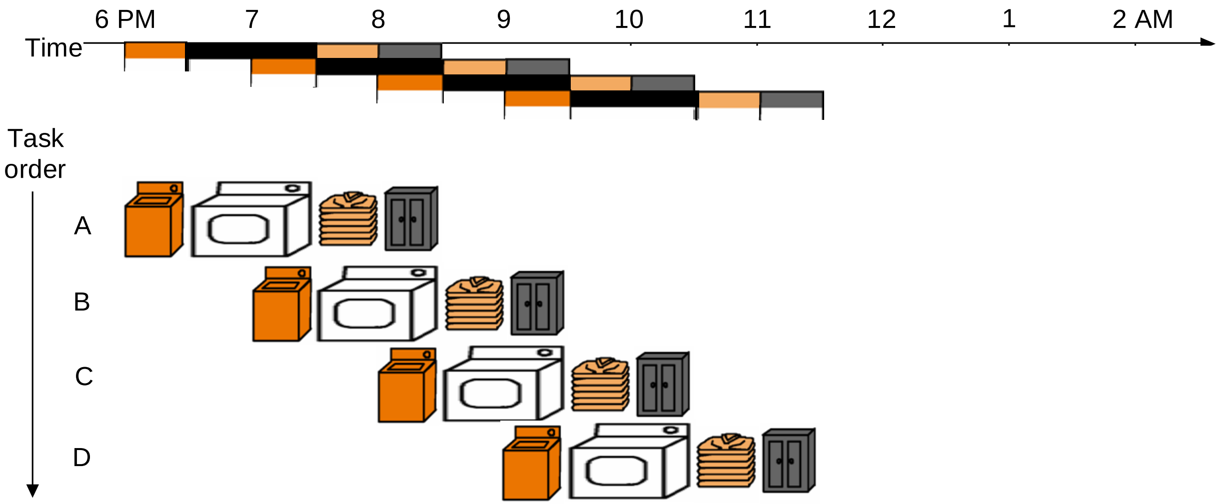




- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

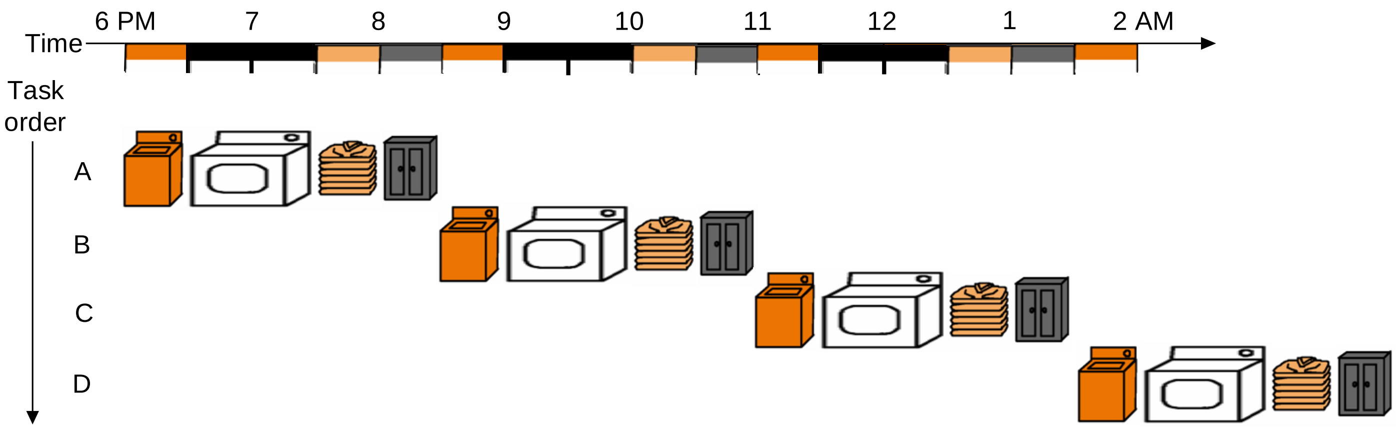
Pipelining Multiple Loads of Laundry: In Practice

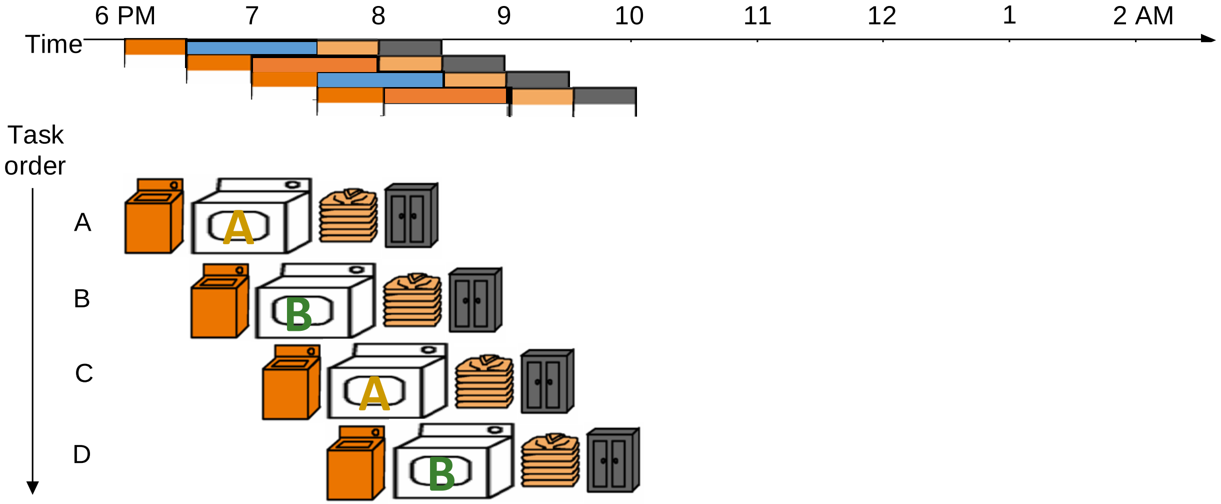




the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice





throughput restored (2 loads per hour) using 2 dryers

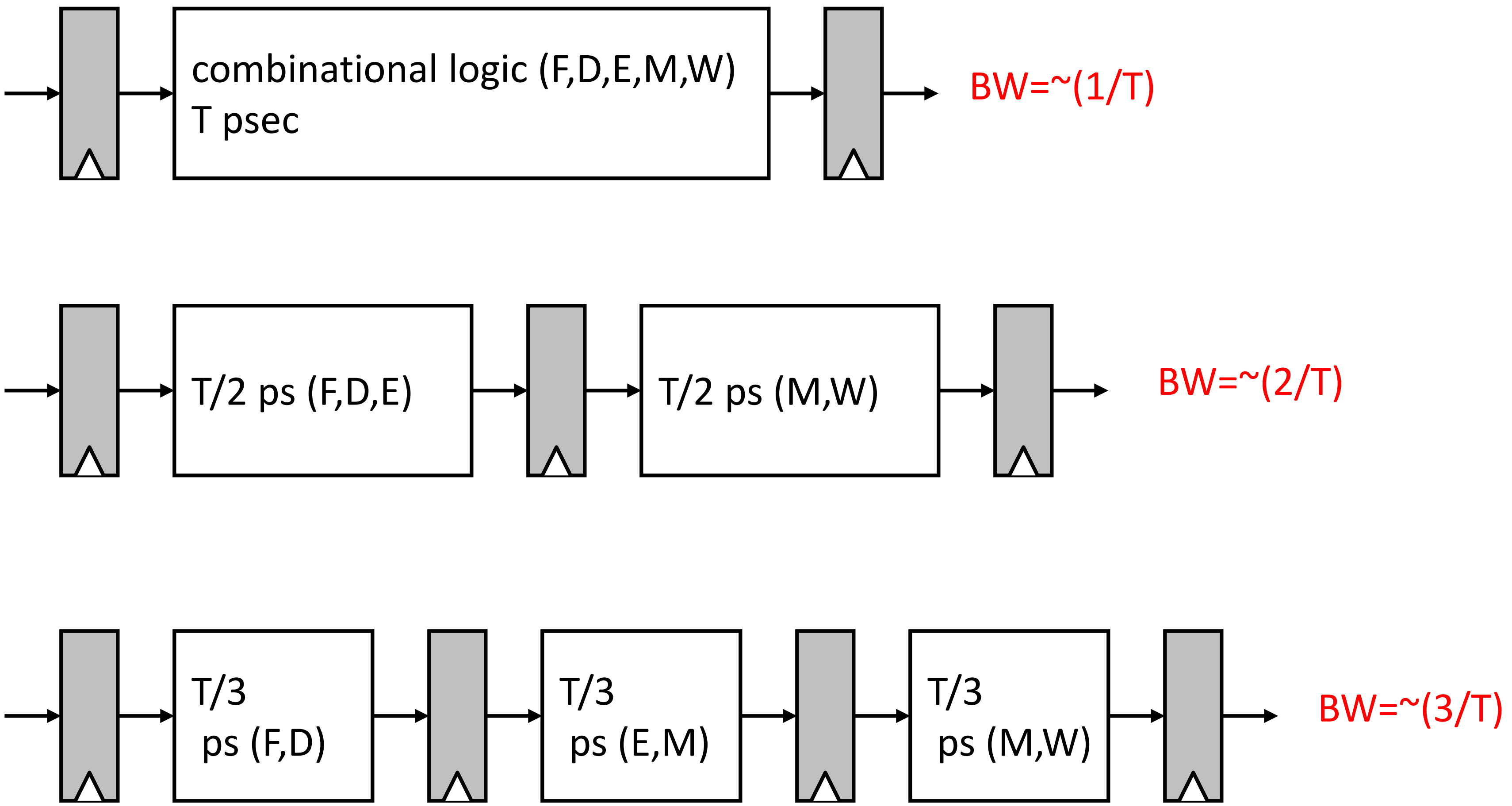
An Ideal Pipeline

- 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

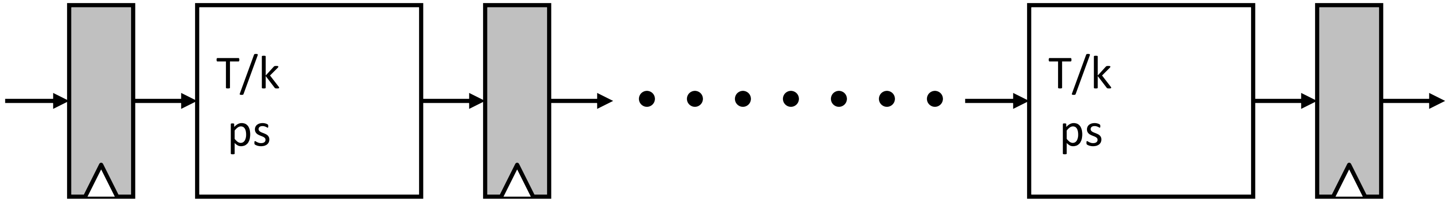
  BW = 1/(T+S) where S = register delay

- k-stage pipelined version

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

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

Register delay reduces throughput (switching overhead between stages)



More Realistic Pipeline: Cost

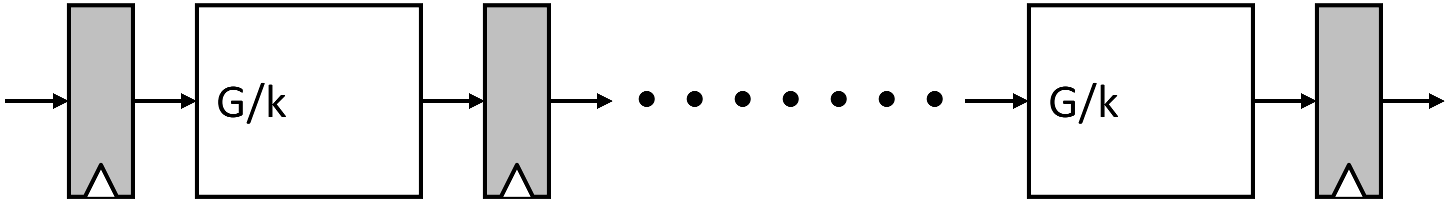
- Nonpipelined version with combinational cost G

  Cost = G+L where L = register cost

- k-stage pipelined version

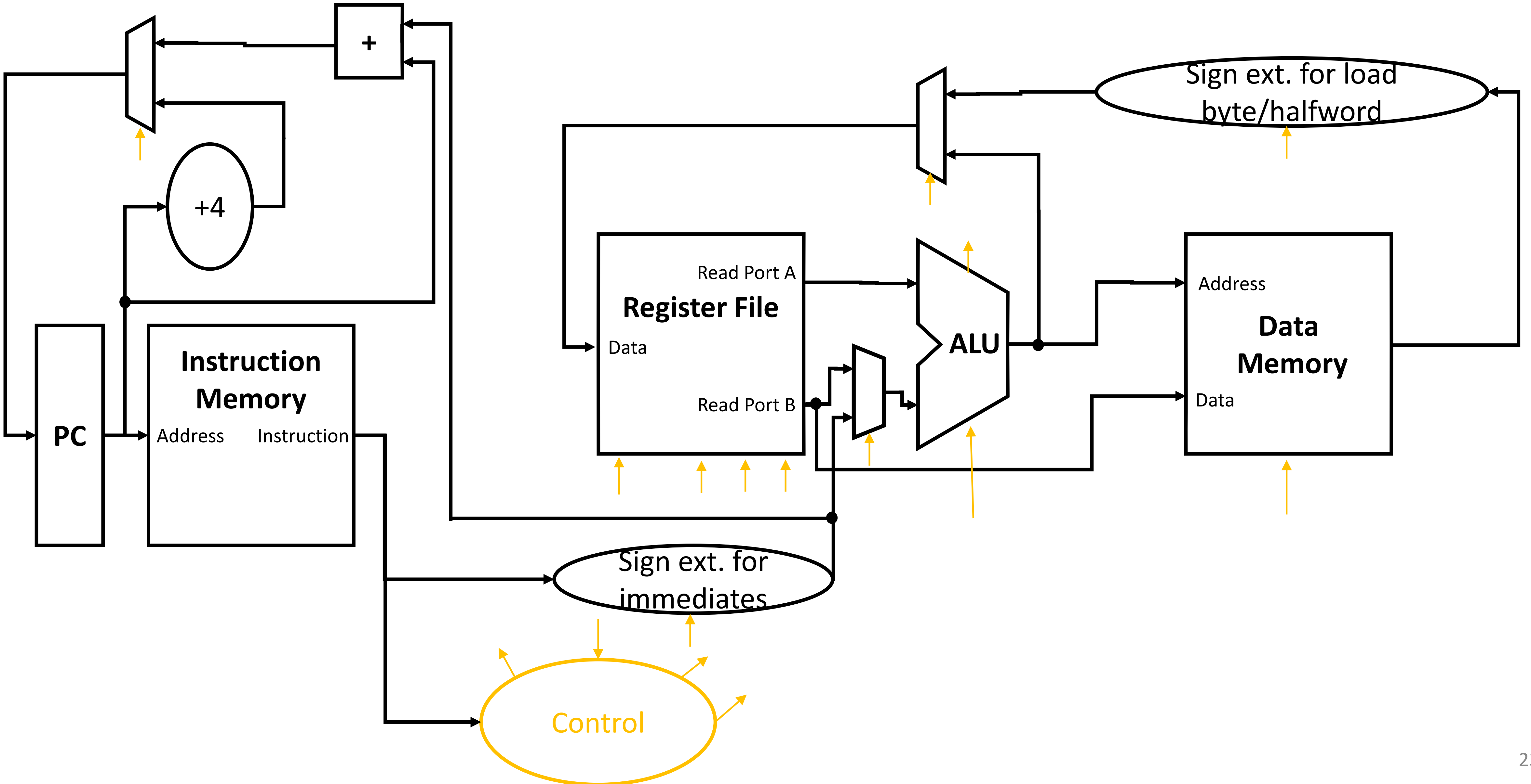
  $Cost_{k\text{-}stage} = G + Lk$

Registers increase hardware cost

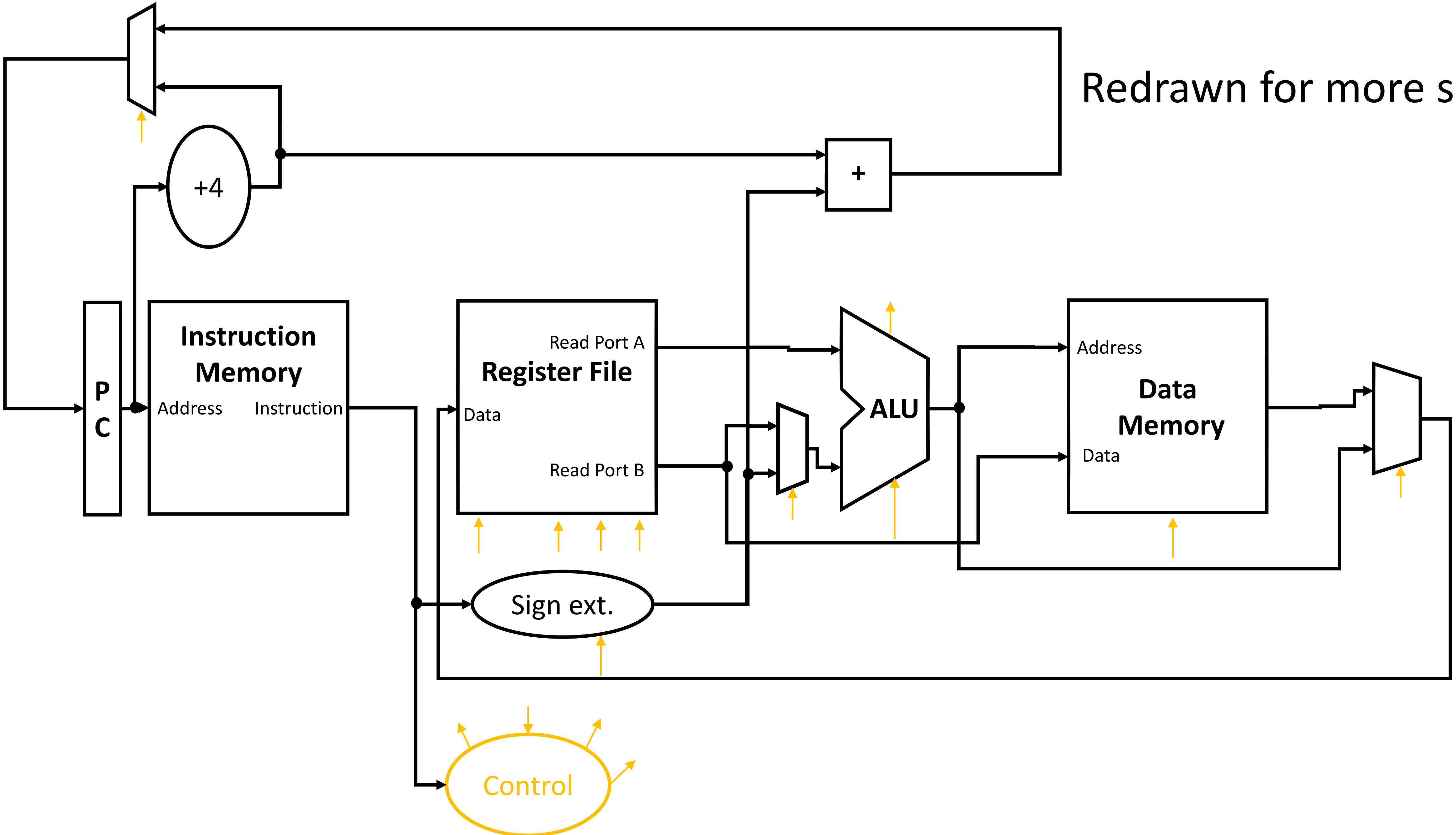


Pipelining Instruction Processing

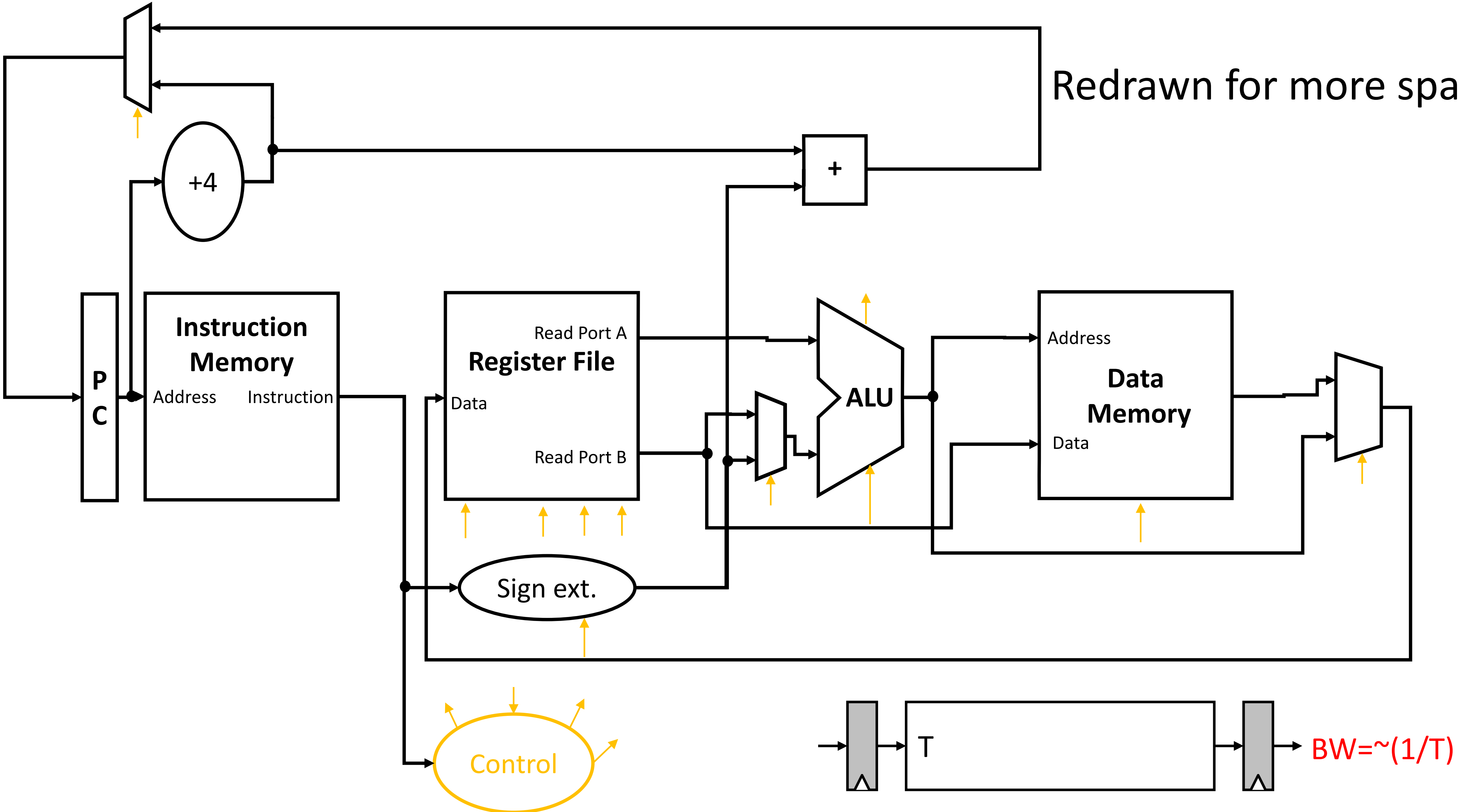
High-Level Datapath

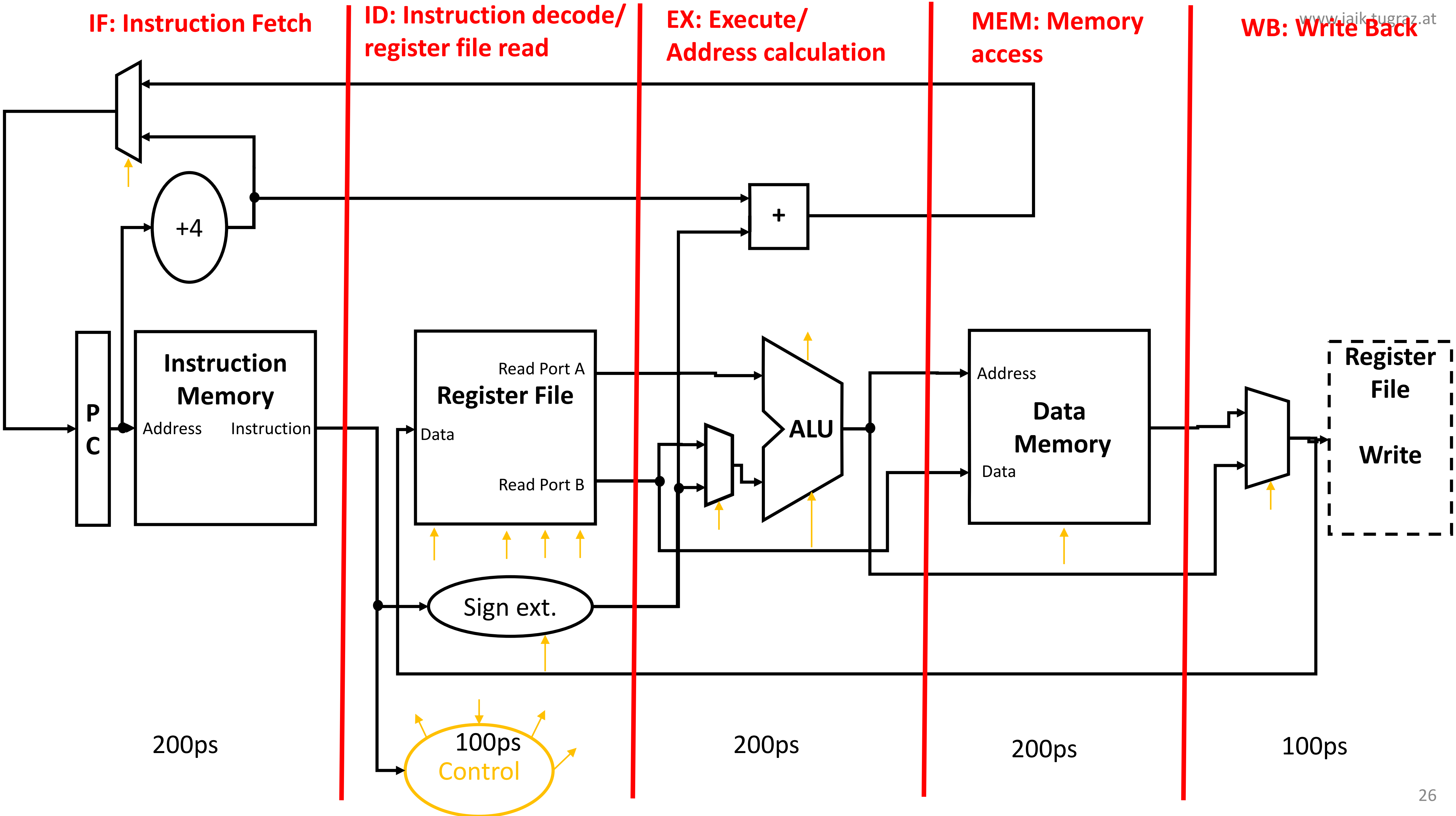


Redrawn for more space



Redrawn for more space

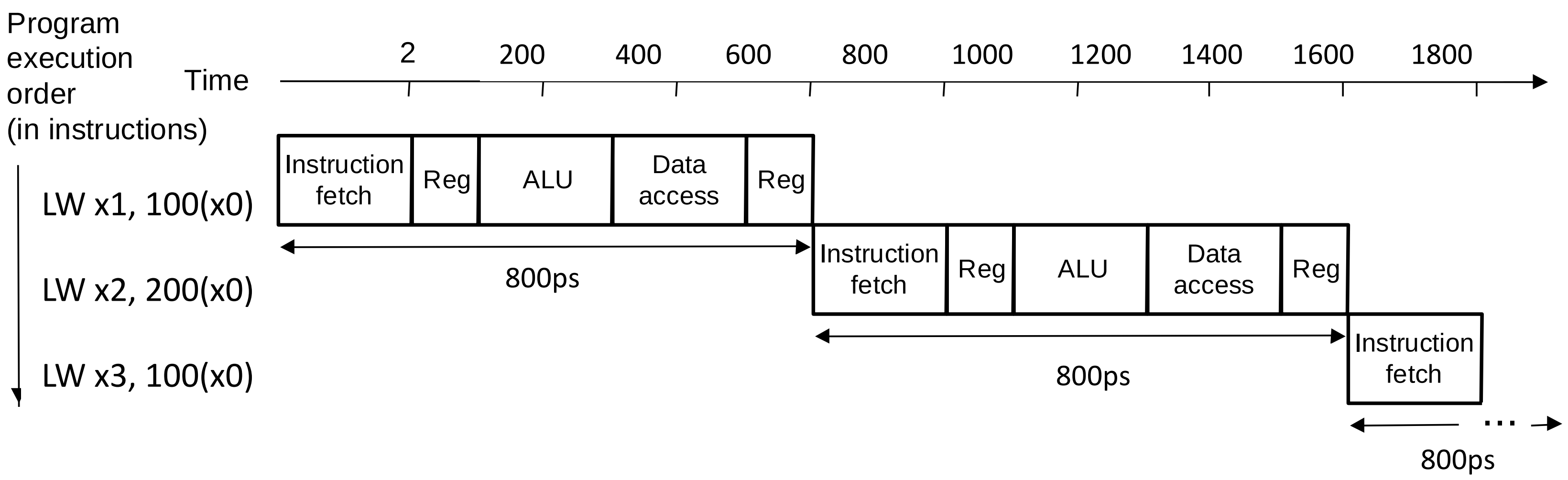


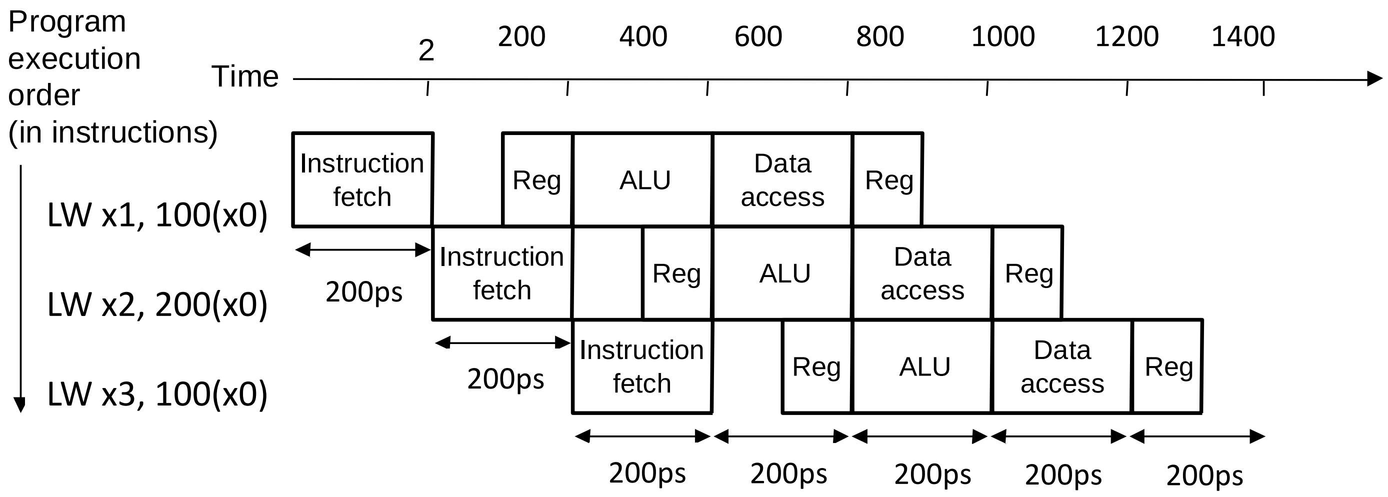

IF: Instruction Fetch
ID: Instruction decode/ register file read
EX: Execute/ Address calculation
MEM: Memory access
WB: Write Back
+4
+
P C
Instruction Memory
Address
Instruction
Read Port A
Register File
Data
Read Port B
ALU
Address
Data Memory
Data
Register File
Write
Sign ext.
Control
200ps
100ps
200ps
200ps
100ps

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

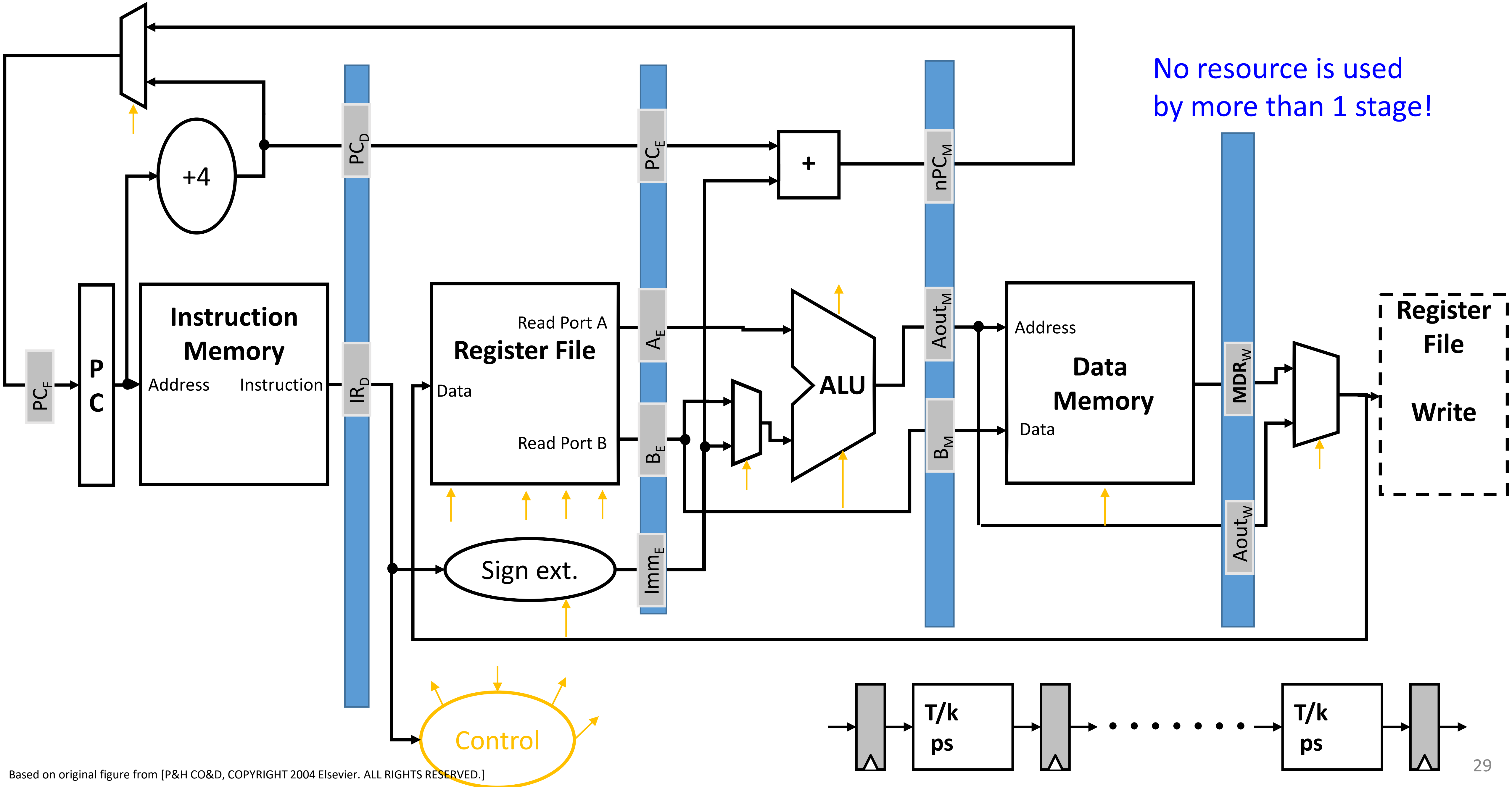




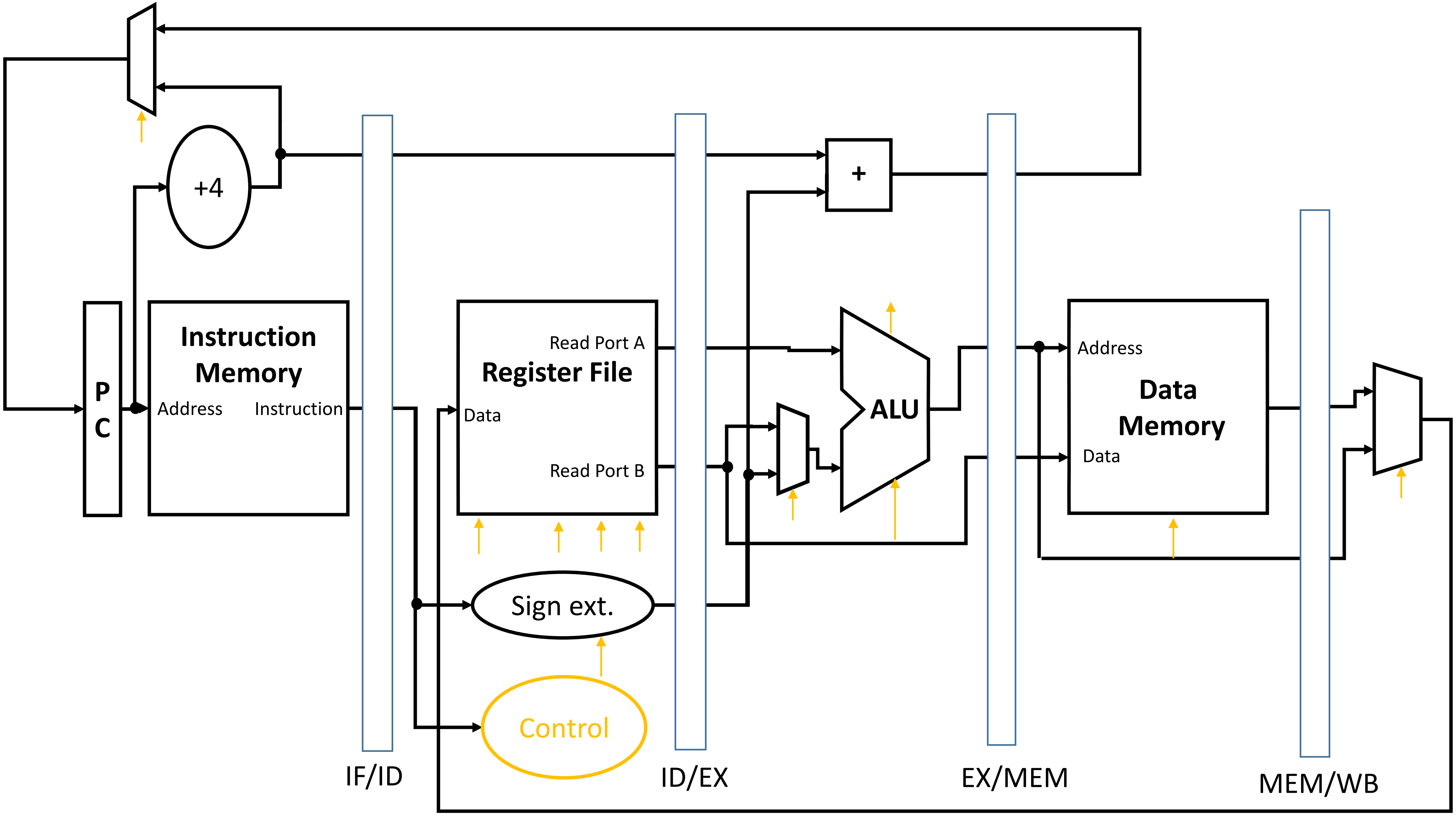
5-stage speedup is 4, not 5 as predicted by the ideal model. Why?
(We complete an instruction every 200ps instead of every 800ps)

Enabling Pipelined Processing: Pipeline Registers

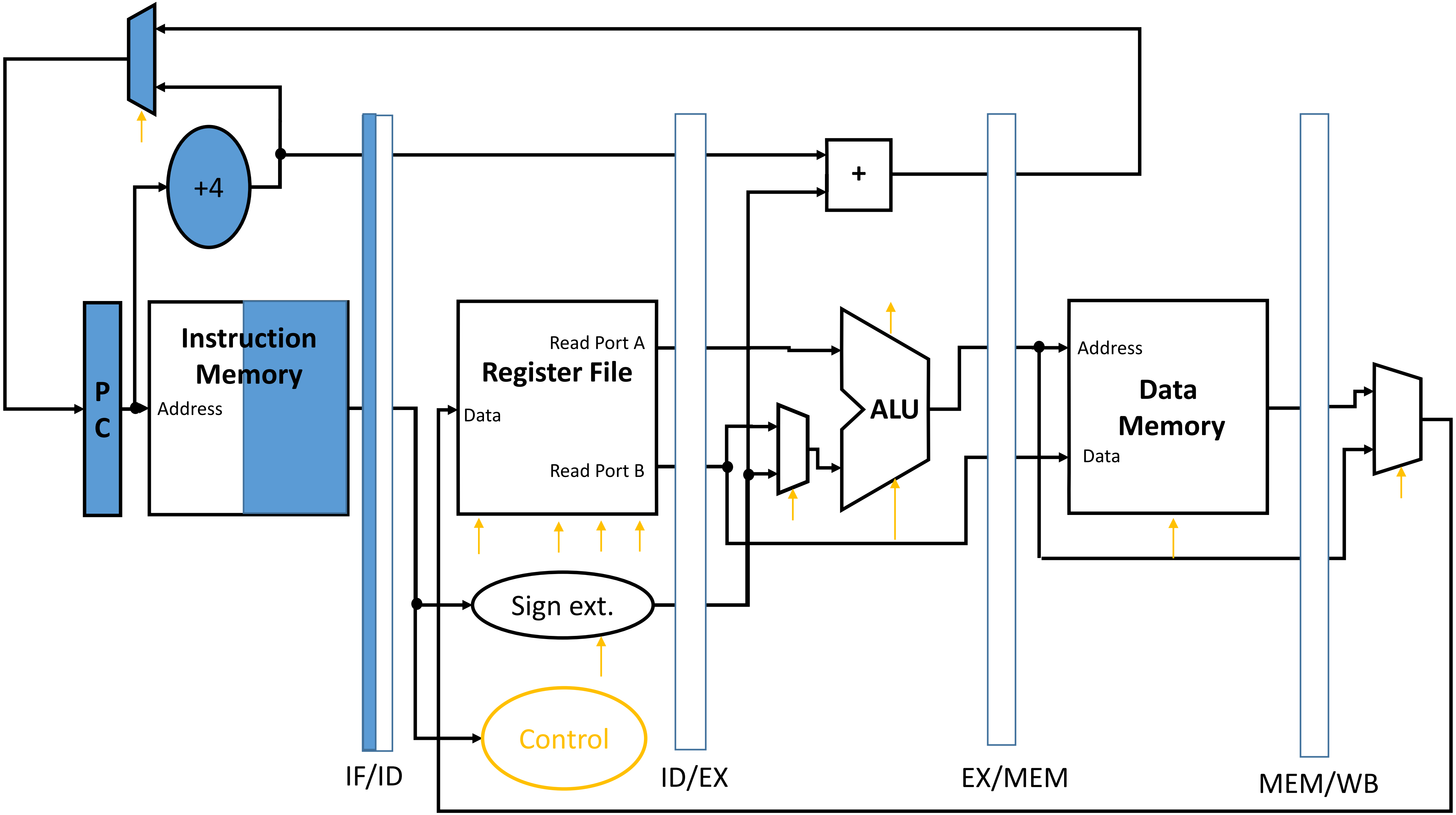
No resource is used by more than 1 stage!



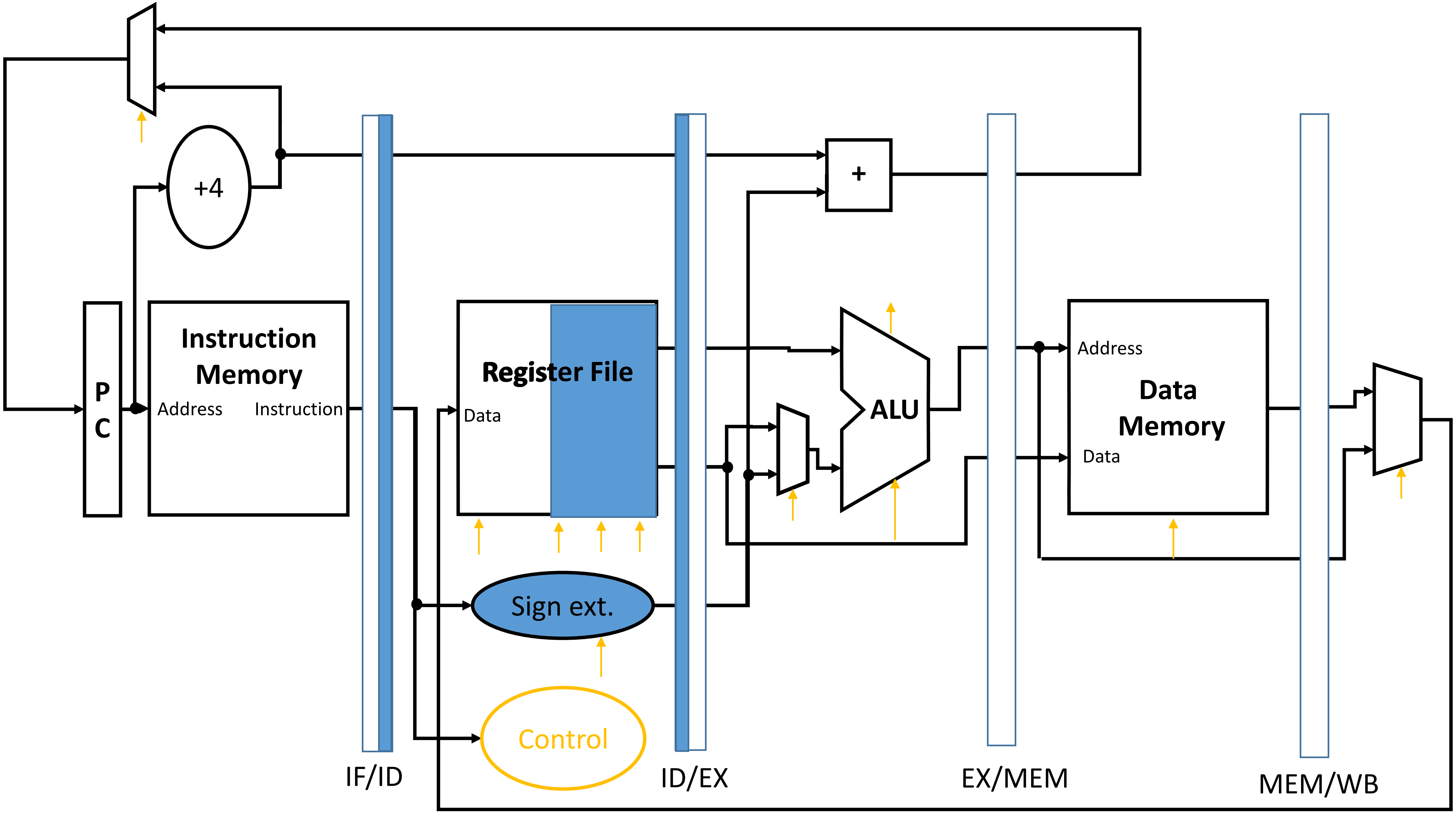
Pipelined Operation



LW
Instruction Fetch

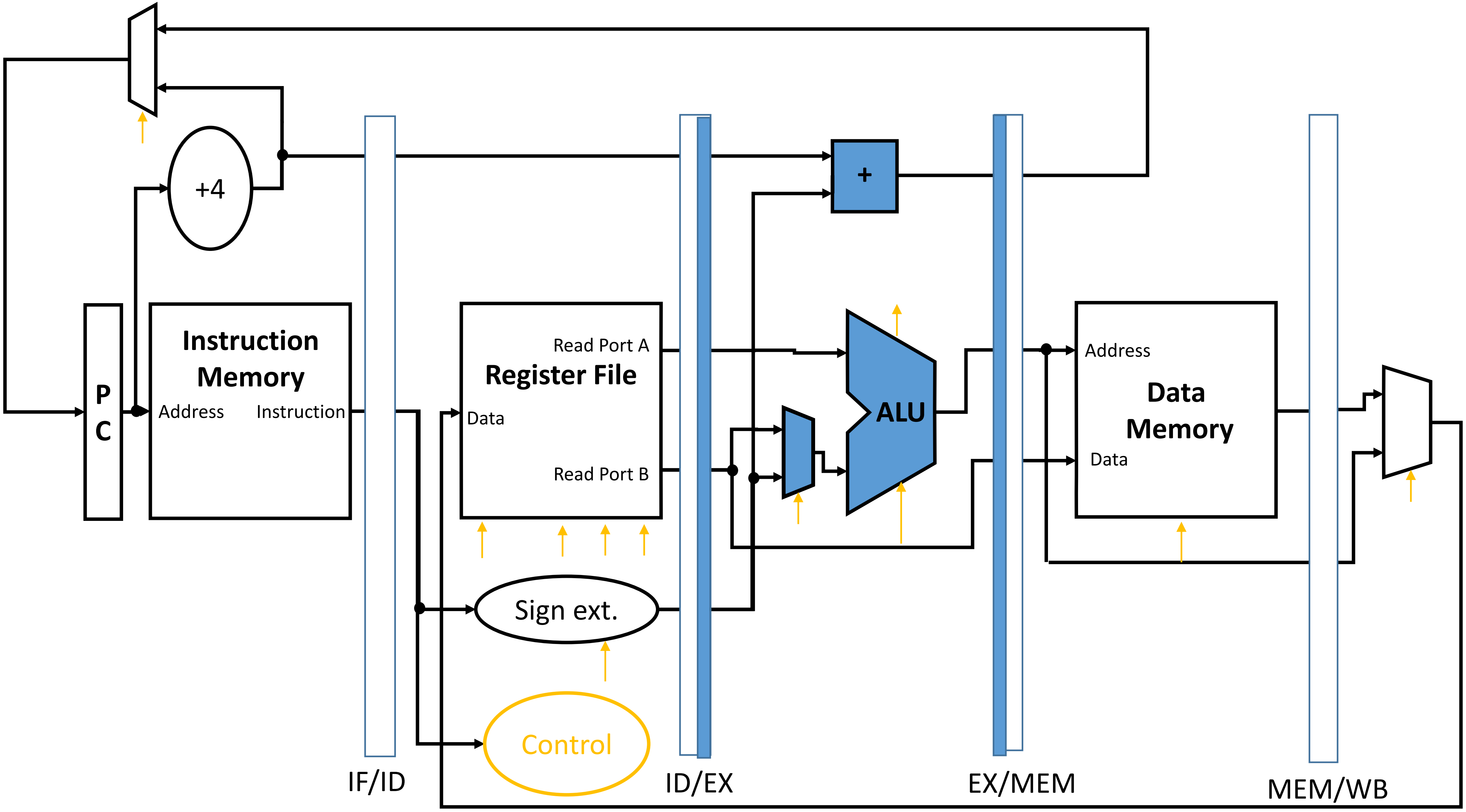


LW
Instruction Decode

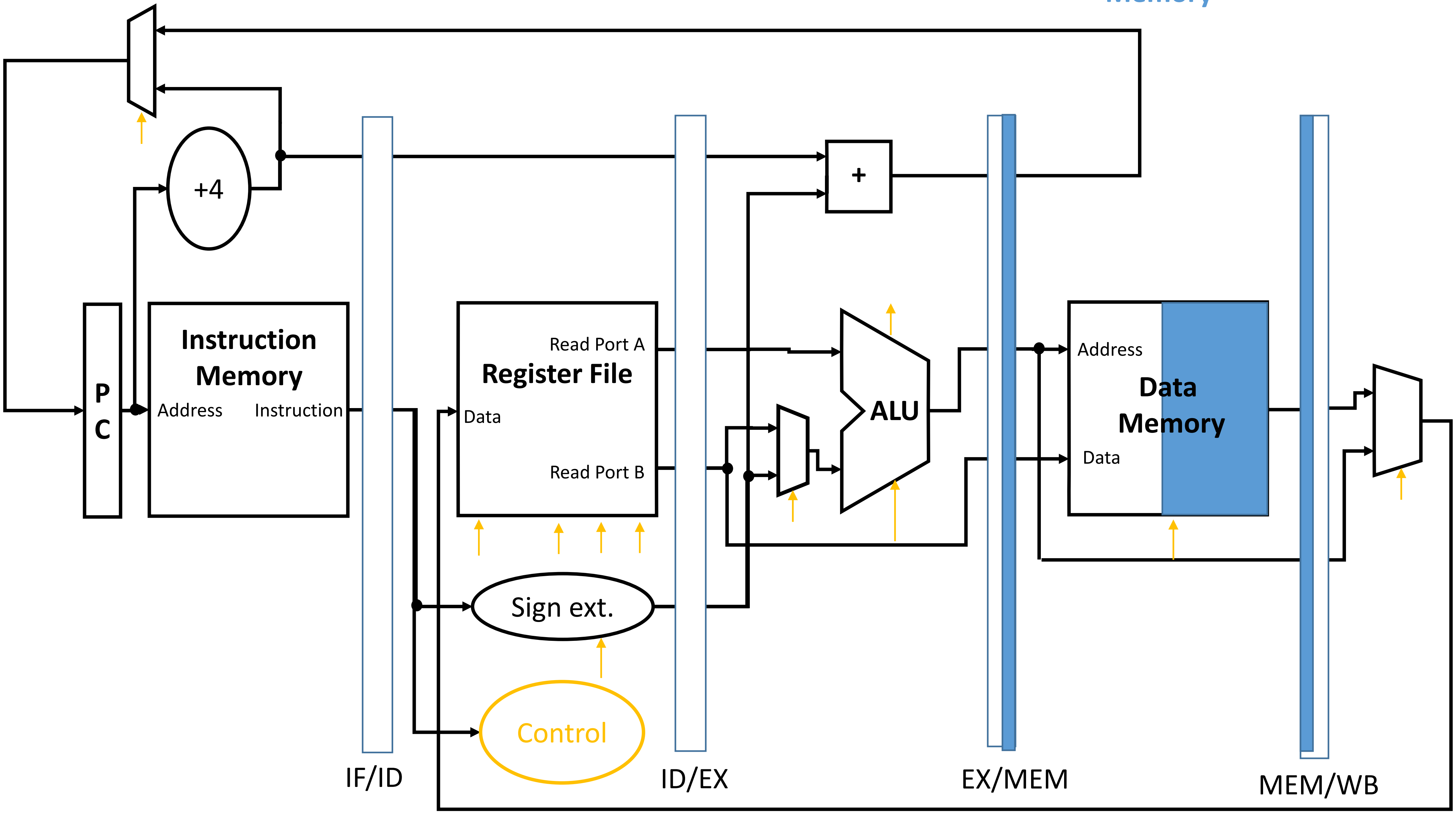


Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

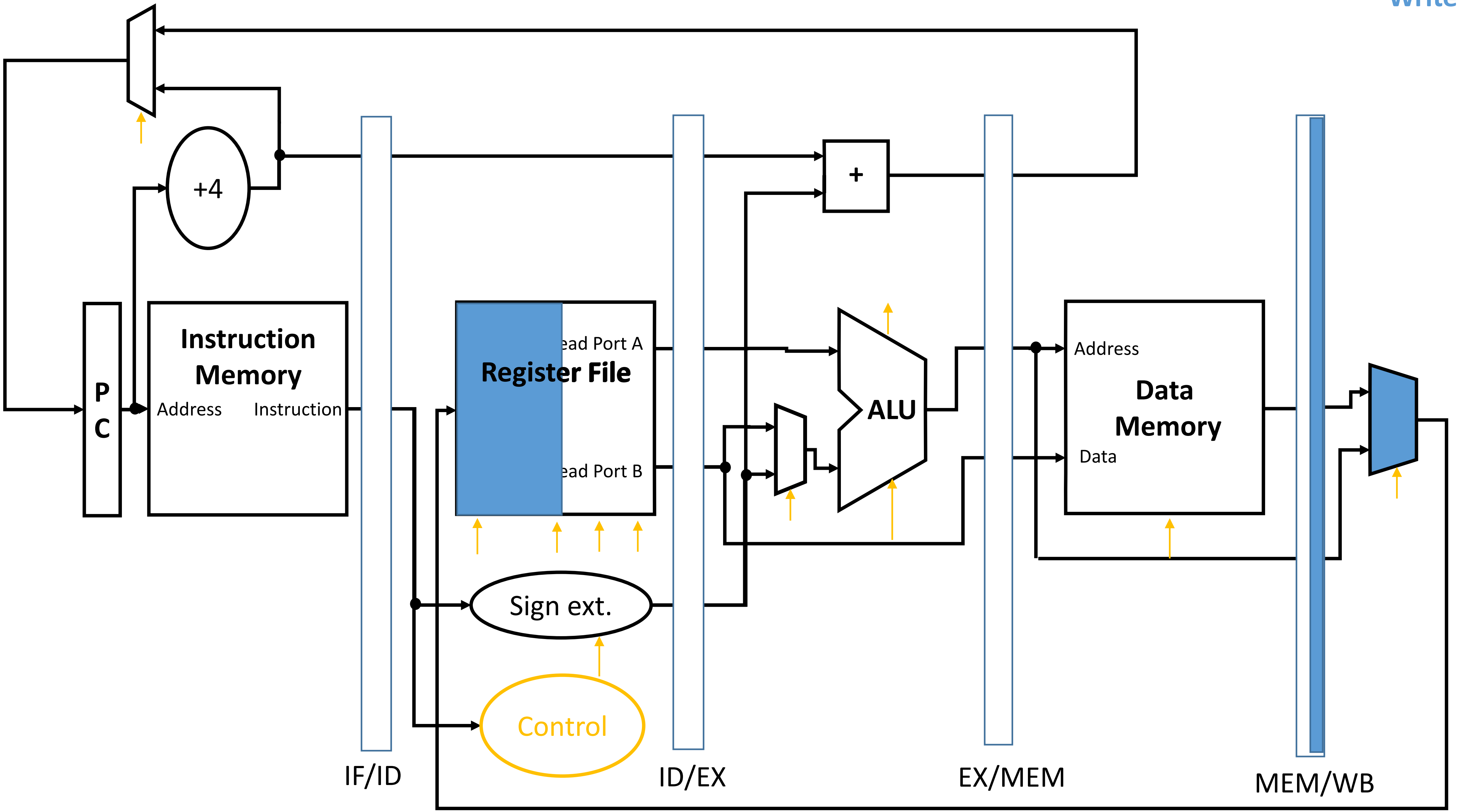
LW
Execute


+4
P
C
Instruction
Memory
Address
Instruction
Read Port A
Register File
Data
Read Port B
Sign ext.
Control
+
ALU
Address
Data
Memory
Data
IF/ID
ID/EX
EX/MEM
MEM/WB

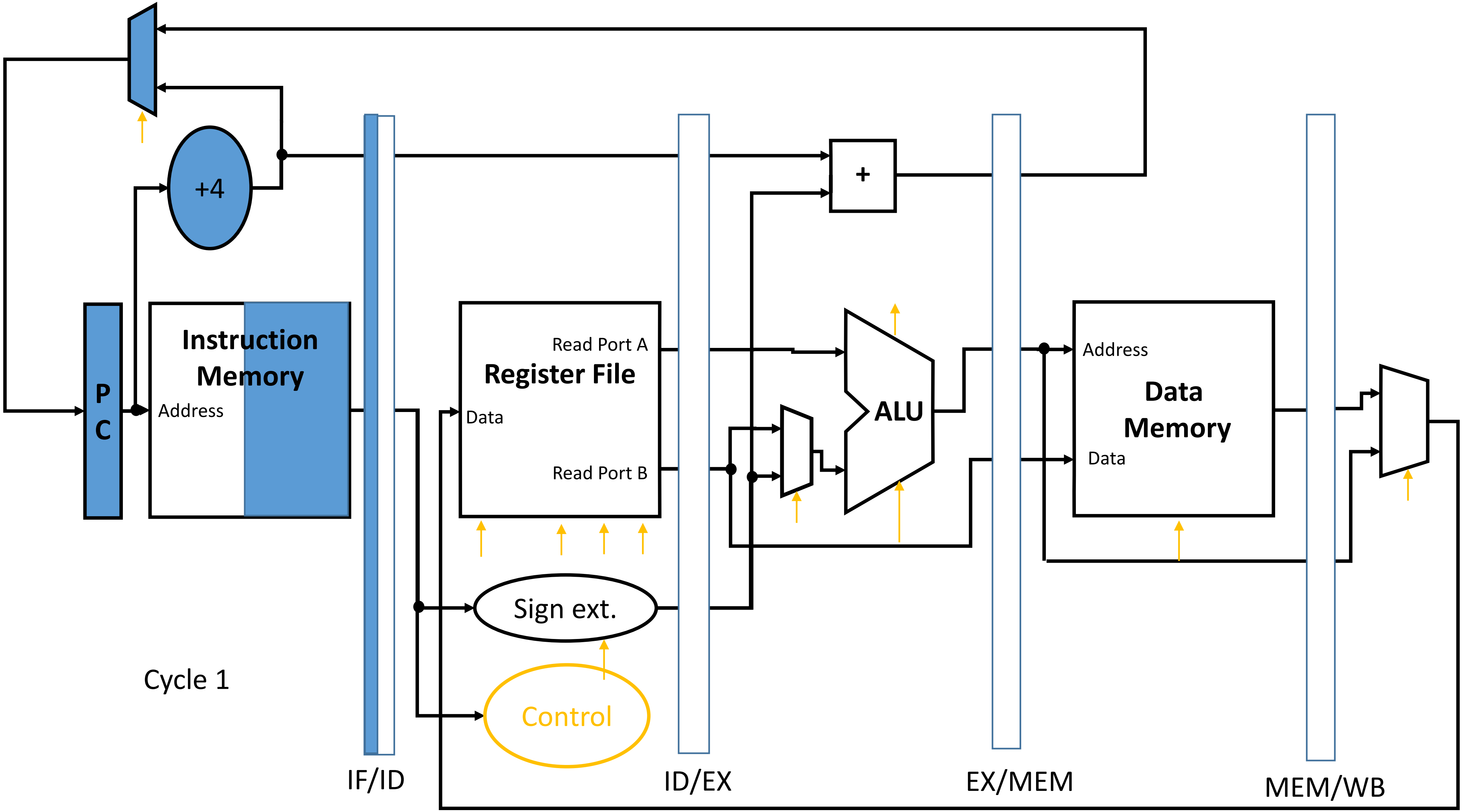
LW
Memory

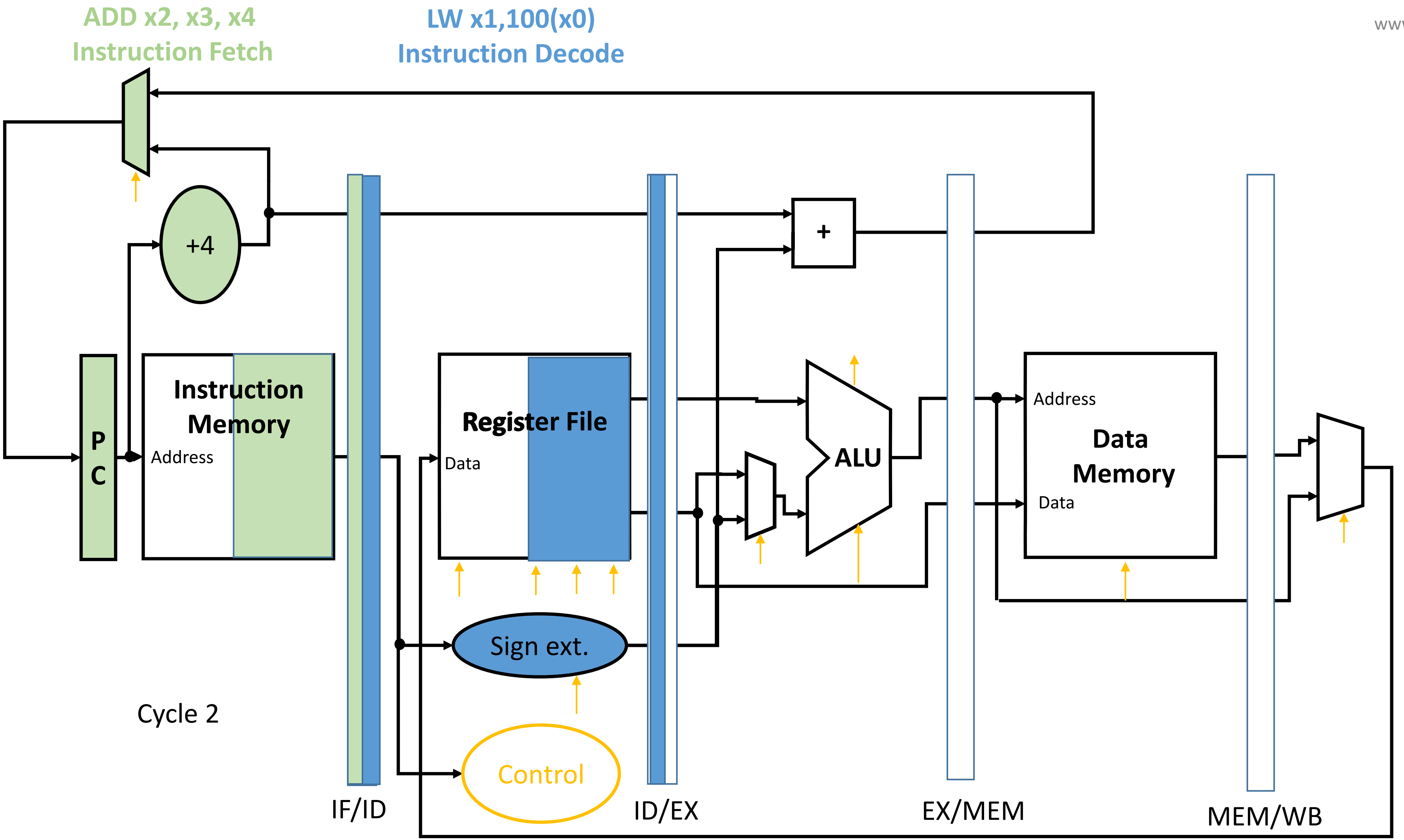

+4
+
P
C
Instruction
Memory
Address
Instruction
Read Port A
Register File
Data
Read Port B
ALU
Address
Data
Memory
Data
Sign ext.
Control
IF/ID
ID/EX
EX/MEM
MEM/WB

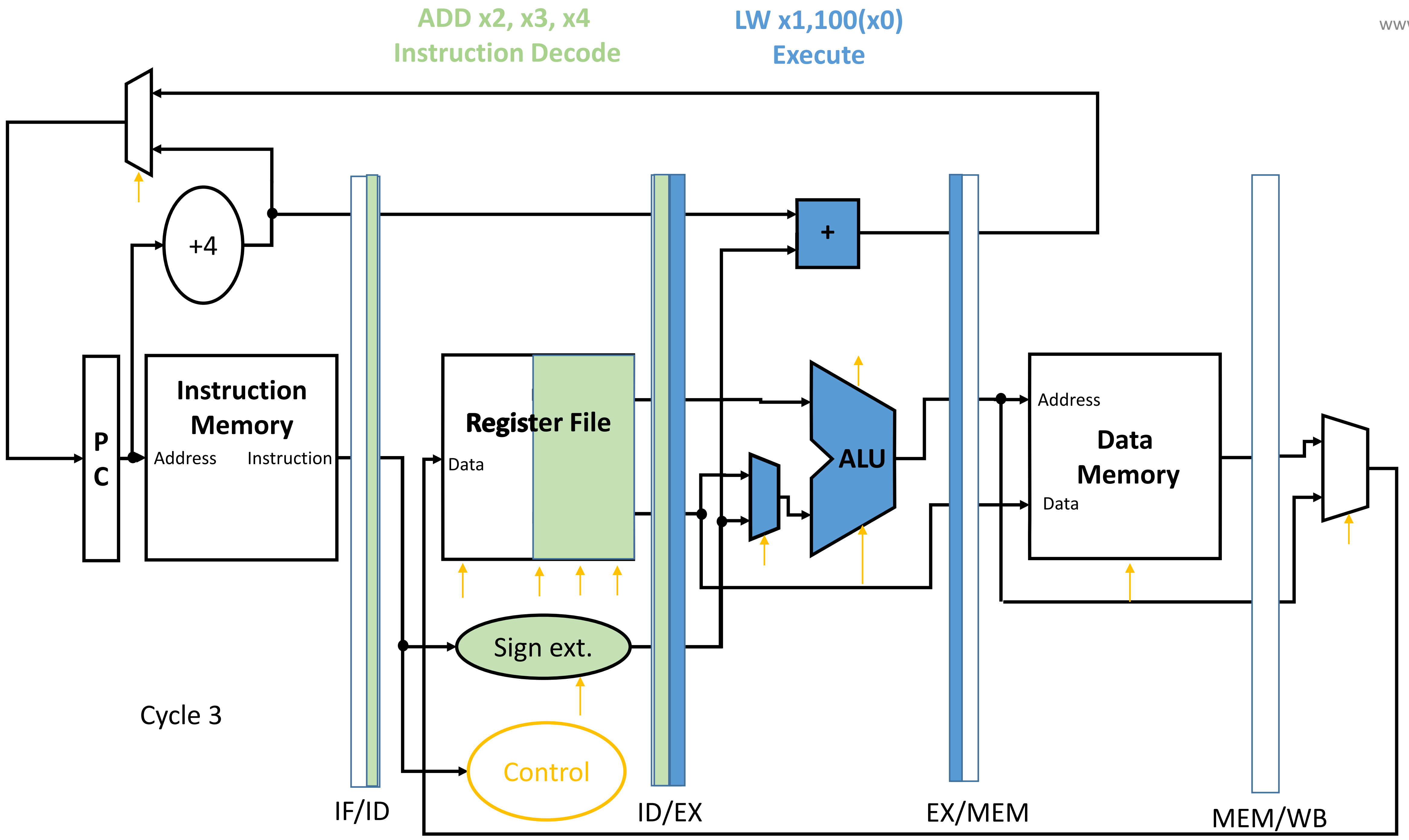
LW
Write Back

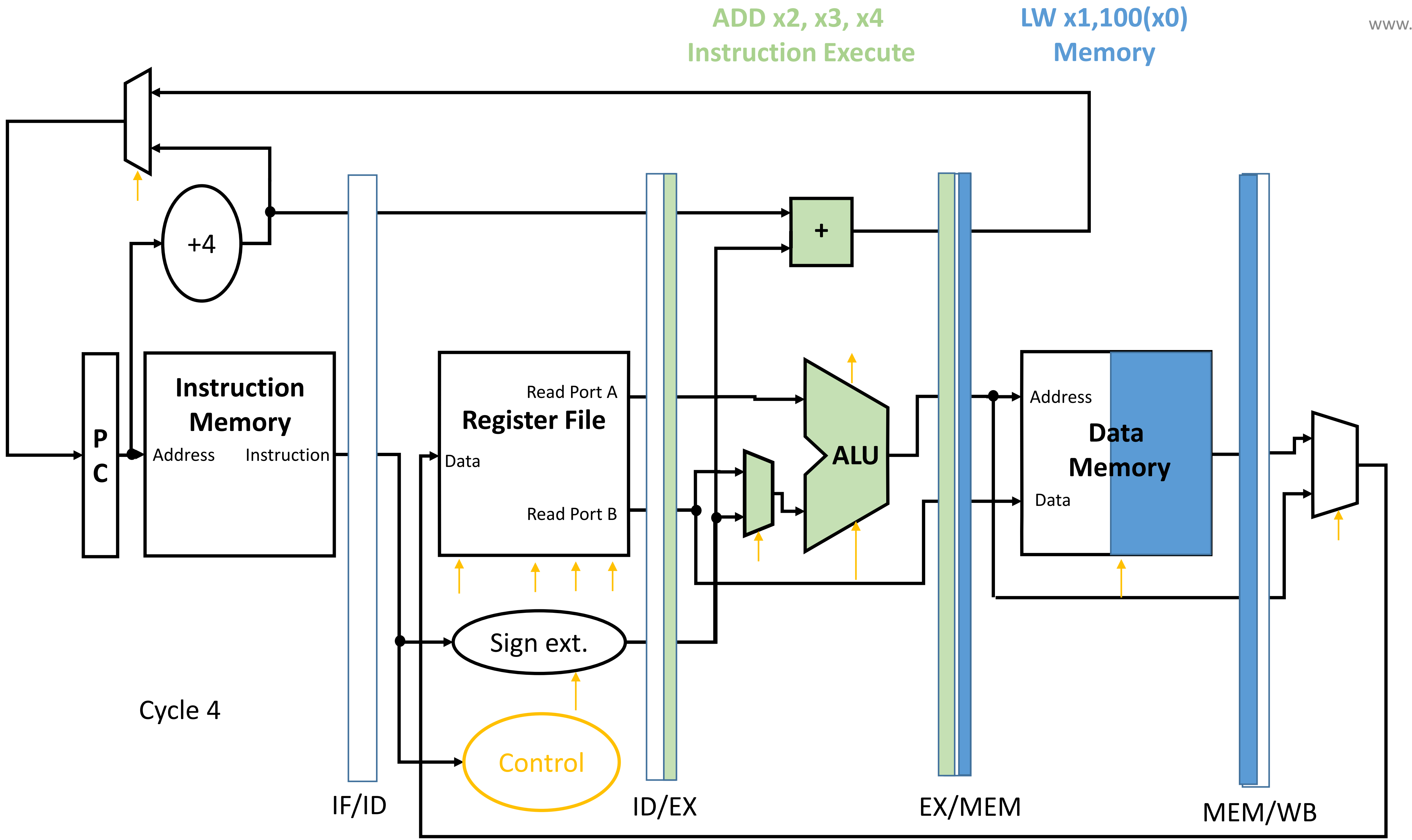


LW x1,100(x0)
Instruction Fetch



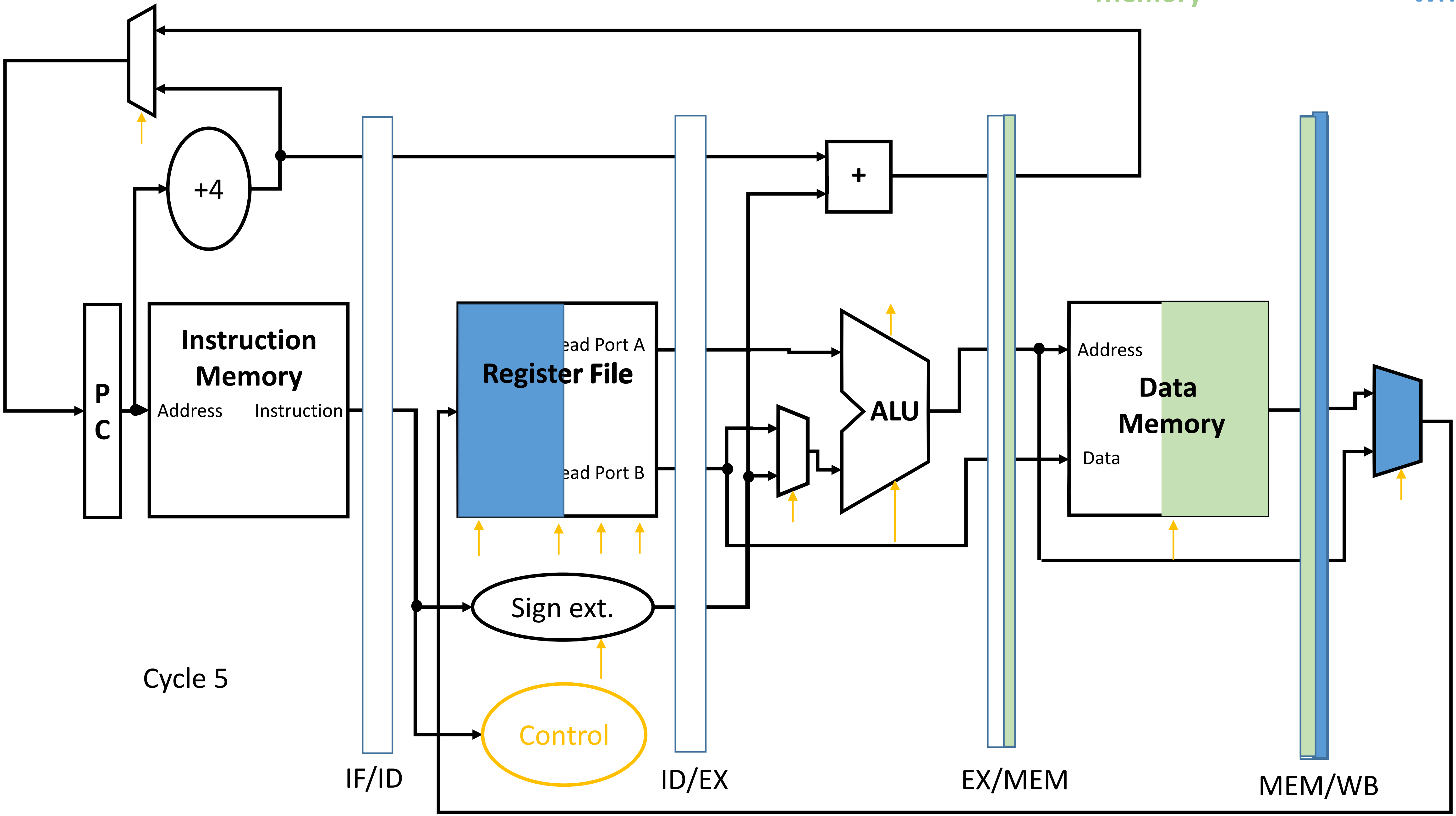

ADD x2, x3, x4
Instruction Fetch
LW x1,100(x0)
Instruction Decode
+4
P
C
Instruction
Memory
Address
Register File
Data
Sign ext.
Control
+
ALU
Address
Data
Memory
Data
IF/ID
ID/EX
EX/MEM
MEM/WB
Cycle 2


ADD x2, x3, x4
Instruction Decode
LW x1,100(x0)
Execute
+4
+
P
C
Instruction
Memory
Address
Instruction
Register File
Data
ALU
Address
Data
Memory
Data
Sign ext.
Control
Cycle 3
IF/ID
ID/EX
EX/MEM
MEM/WB


ADD x2, x3, x4
Instruction Execute
LW x1,100(x0)
Memory
+4
+
P
C
Instruction
Memory
Address
Instruction
Read Port A
Register File
Data
Read Port B
ALU
Address
Data
Memory
Data
Sign ext.
Control
Cycle 4
IF/ID
ID/EX
EX/MEM
MEM/WB

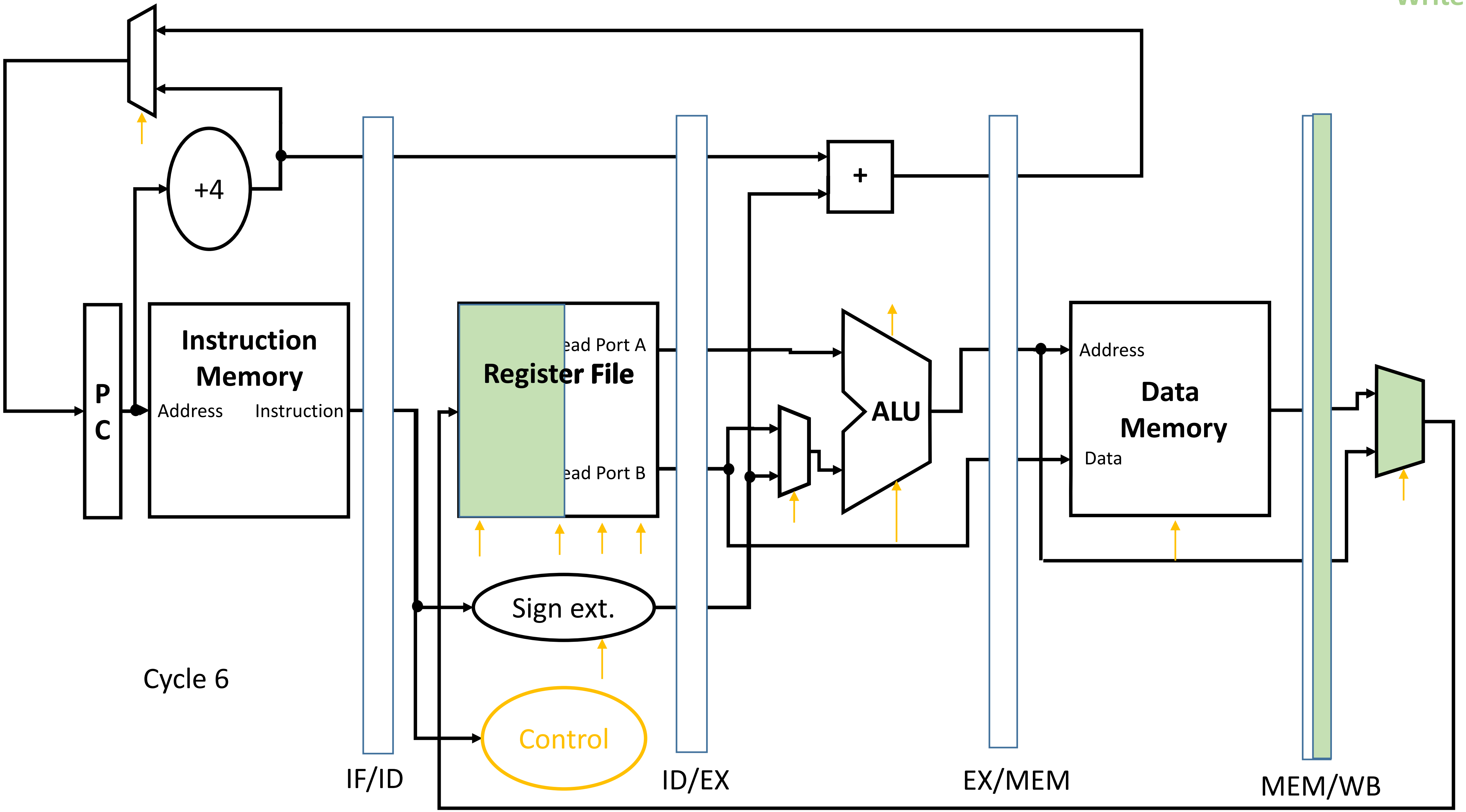
ADD x2, x3, x4
Memory

LW x1,100(x0)
Write Back



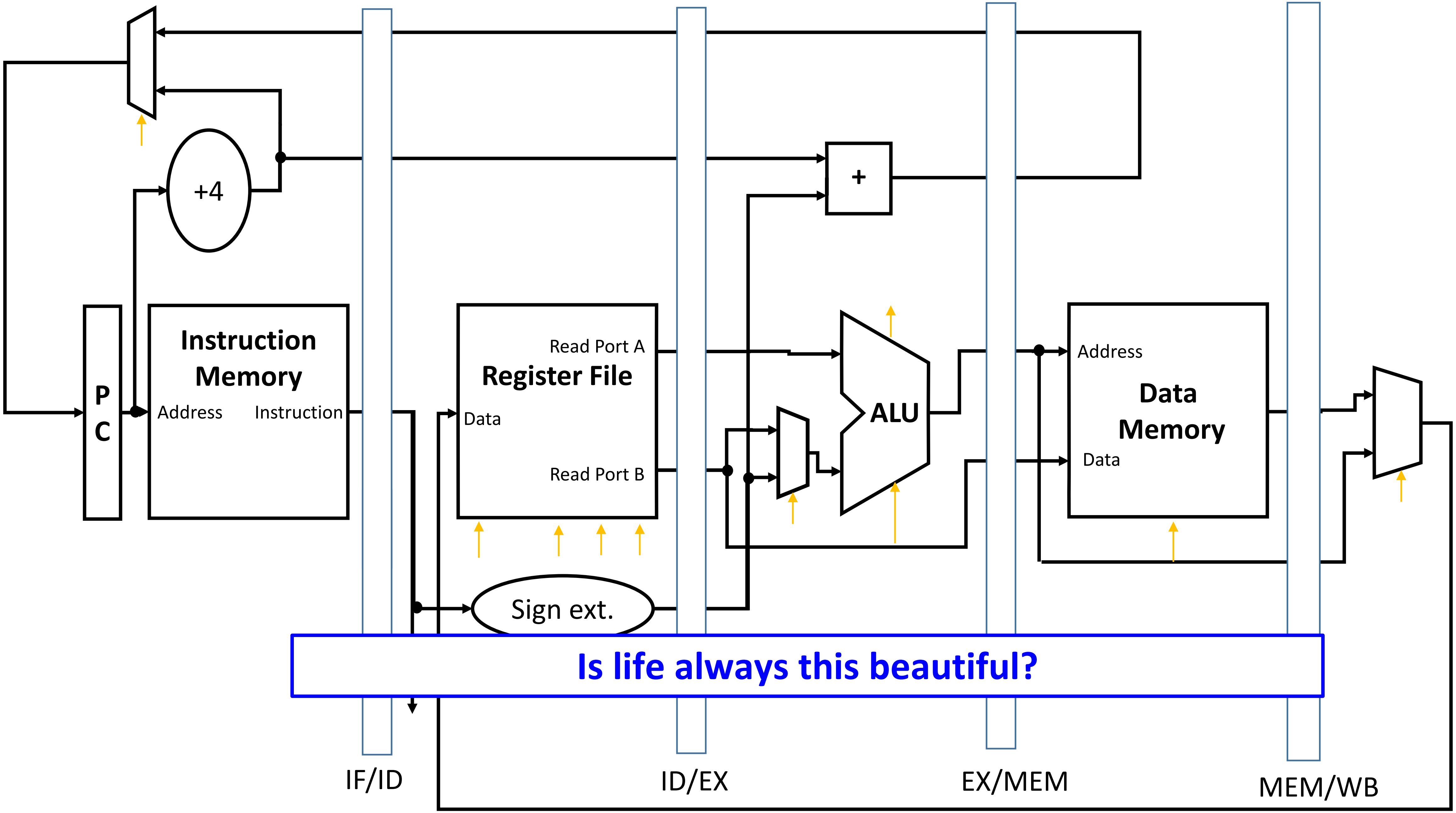
Cycle 5

ADD x2, x3, x4
Write Back



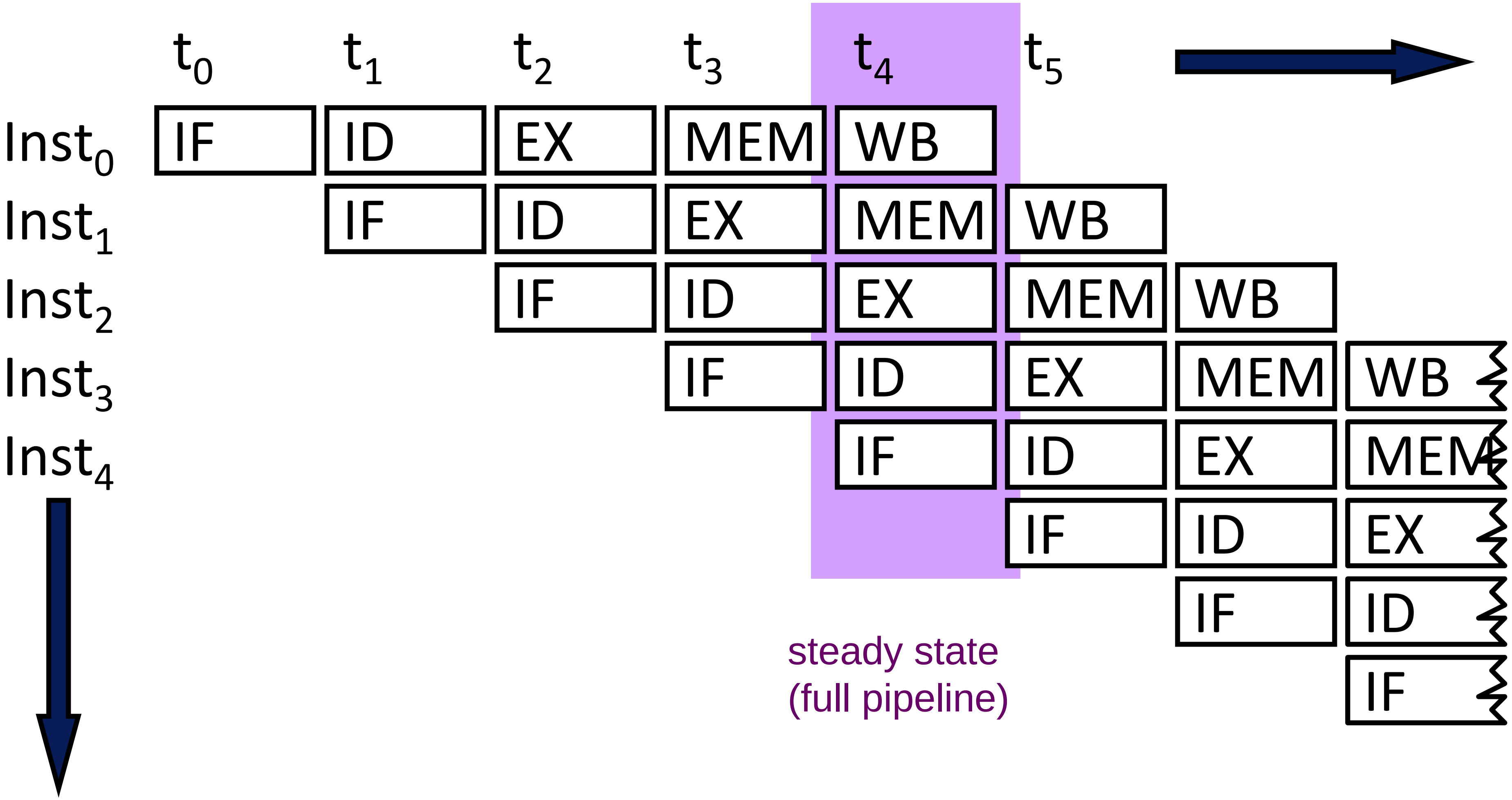
Cycle 6

Pipelined Operation



Is life always this beautiful?

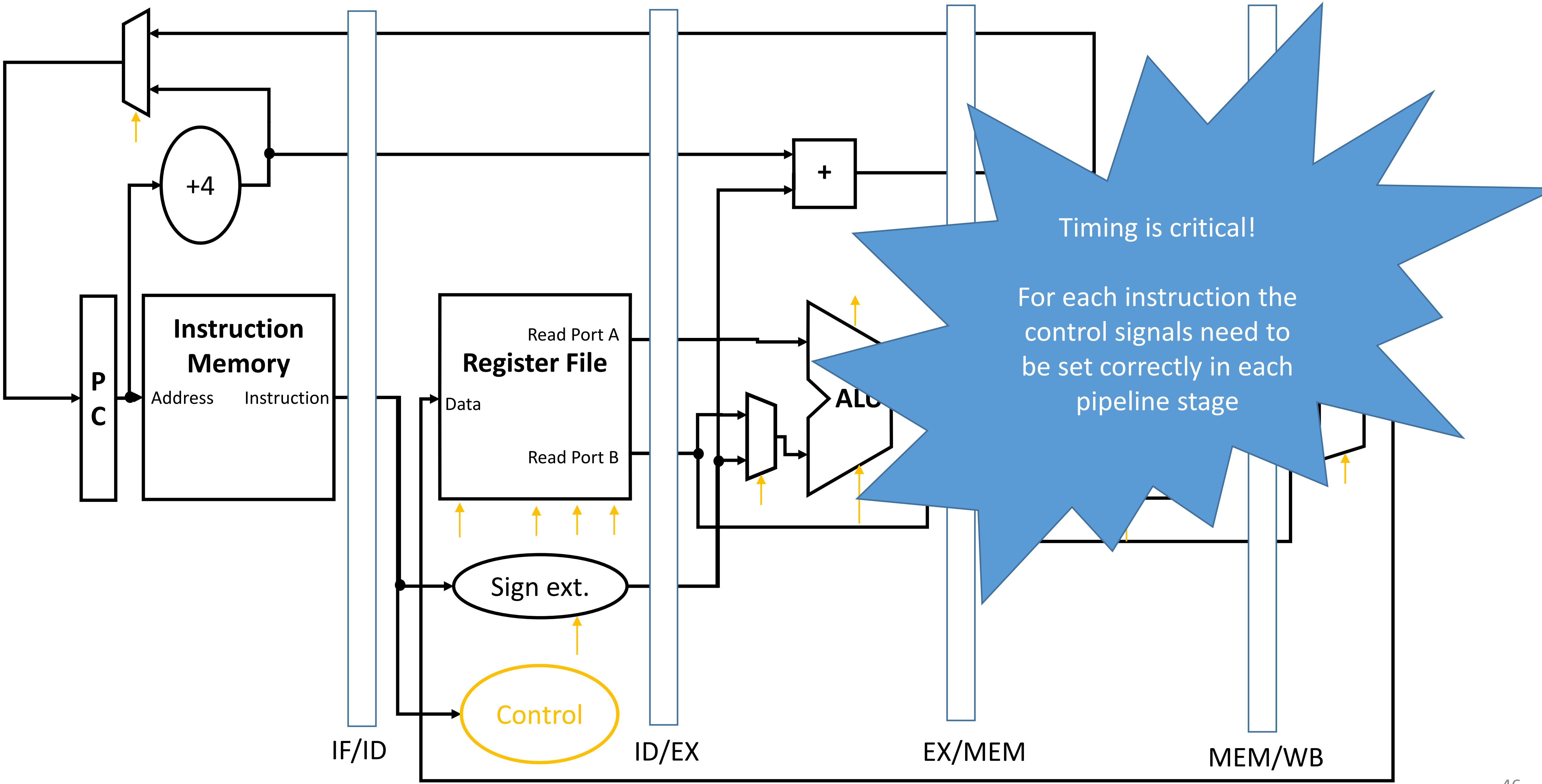
Illustrating Pipeline Operation: Operation View



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

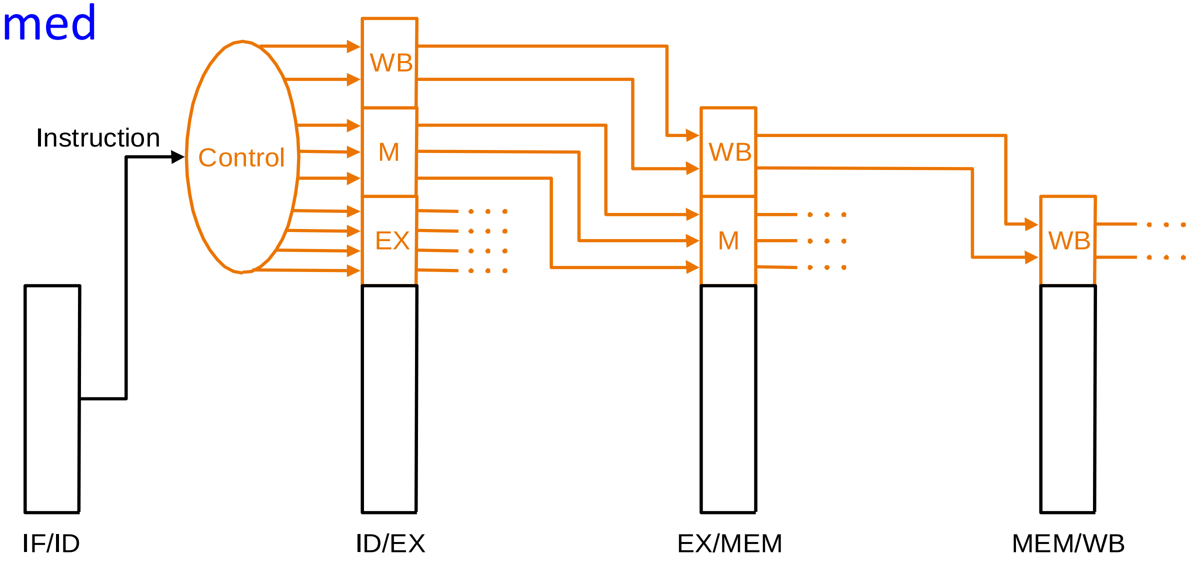
Note: There is the same number of control signals as in a single-cycle data path



Control Signals in a Pipeline

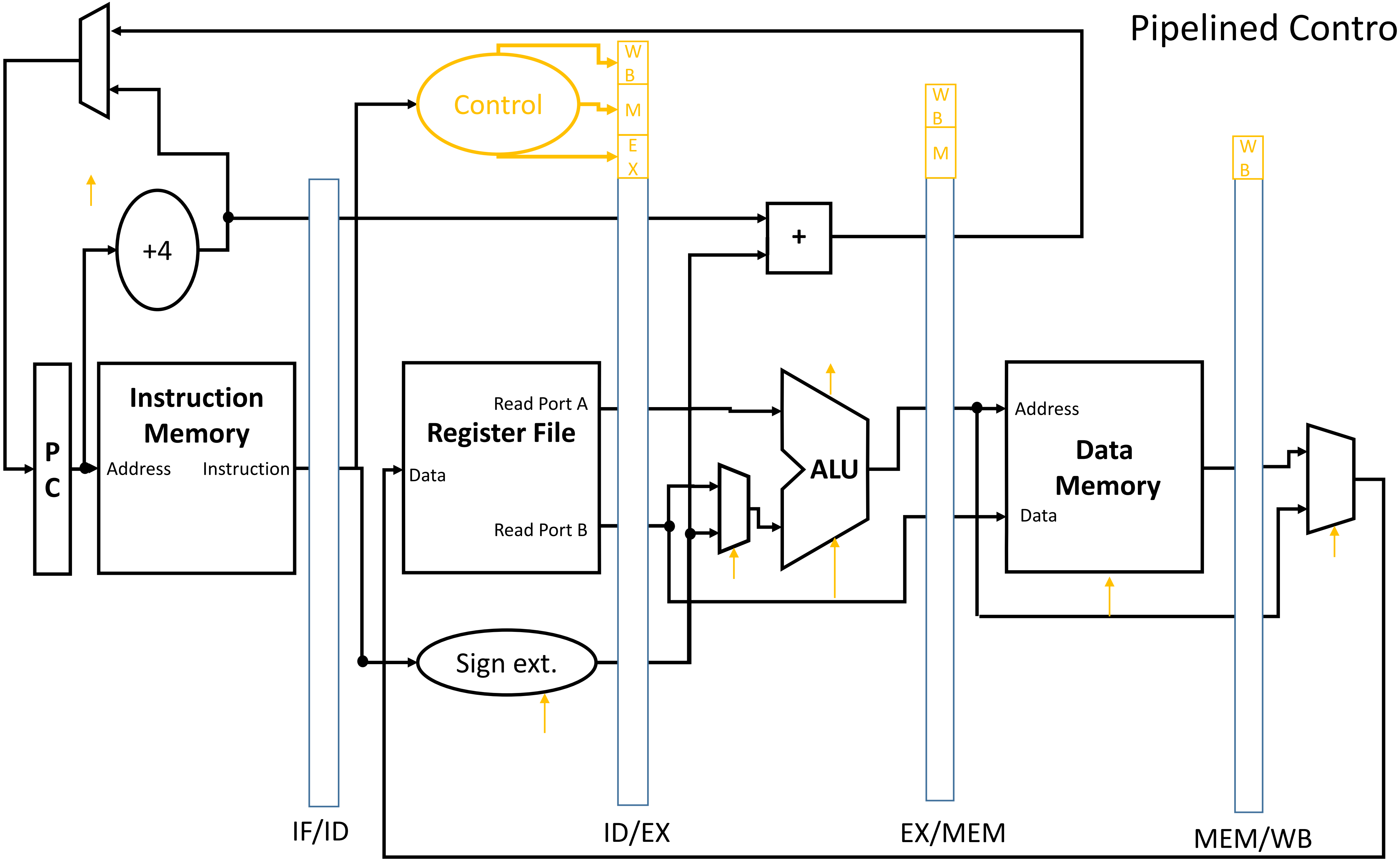
- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage

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

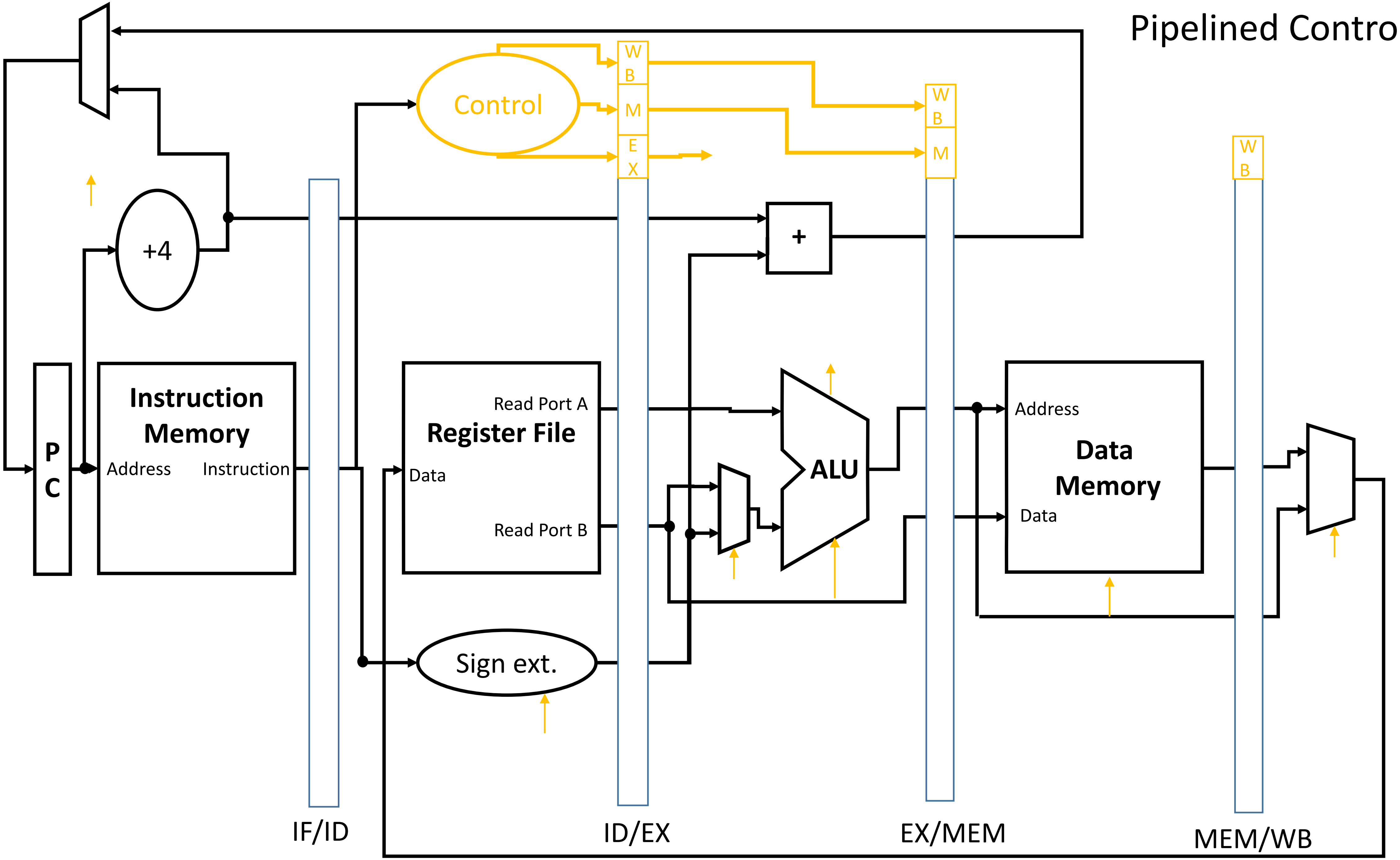


⇒Option 2: carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

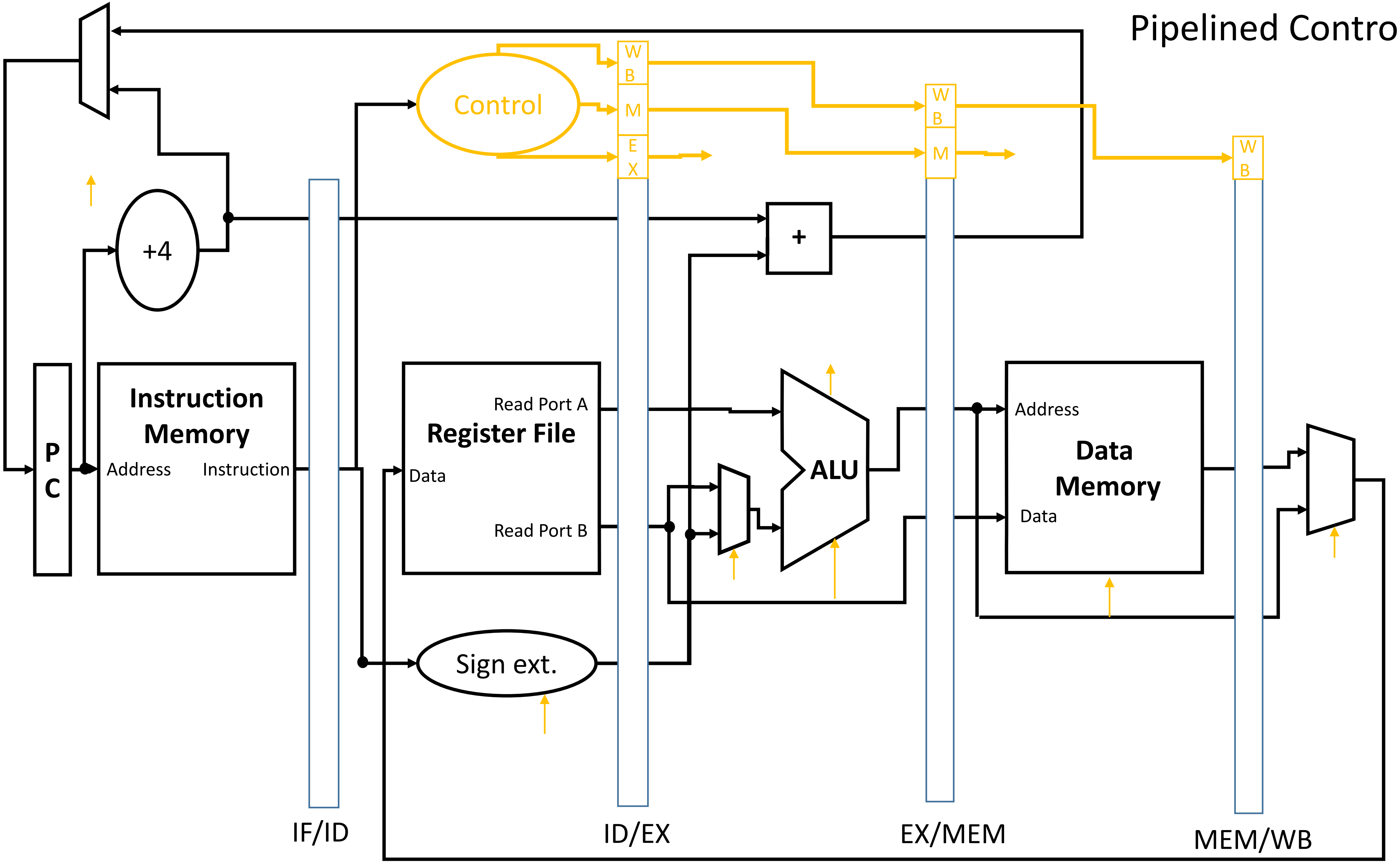
Pipelined Control Signals



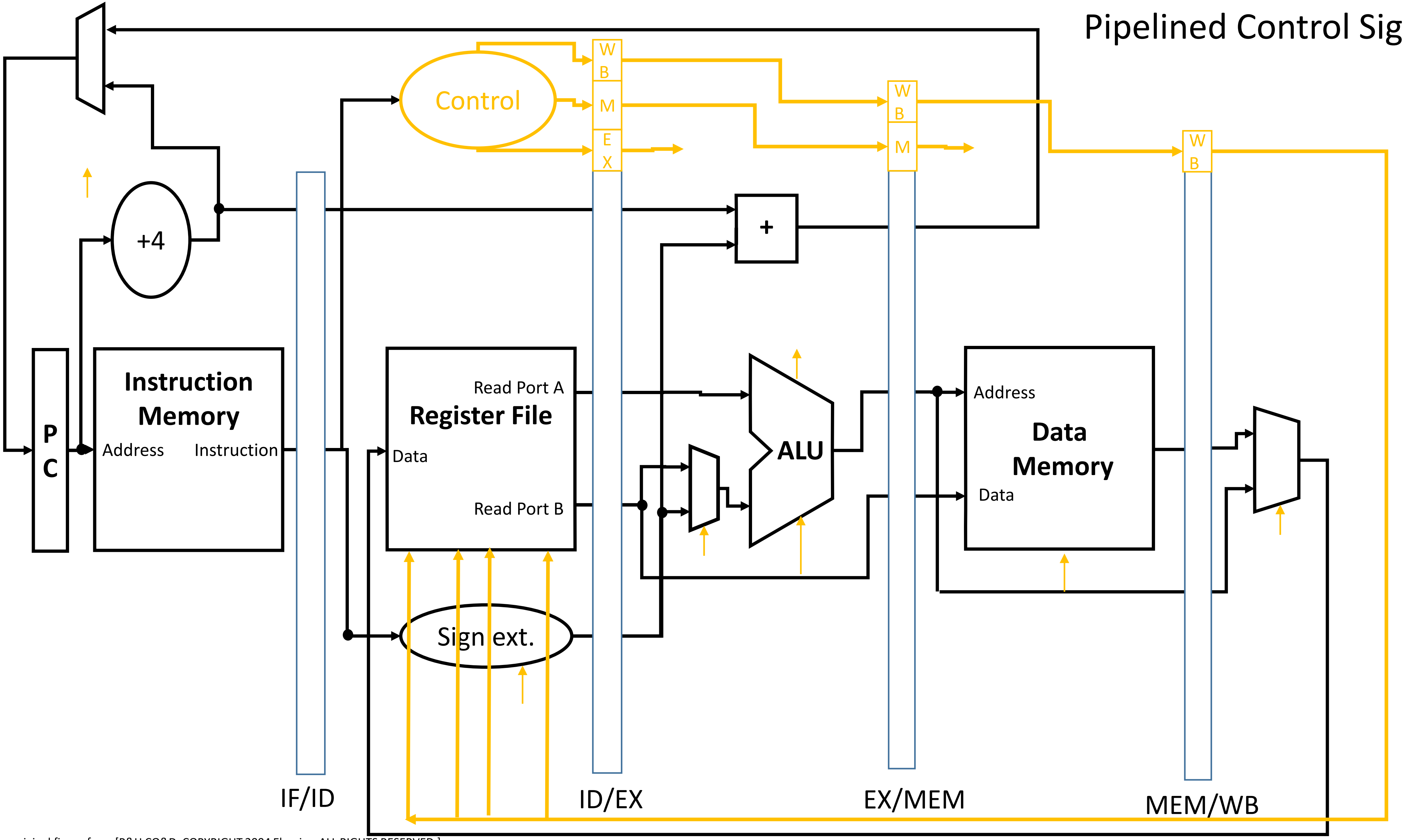
Pipelined Control Signals



Pipelined Control Signals



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 an 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

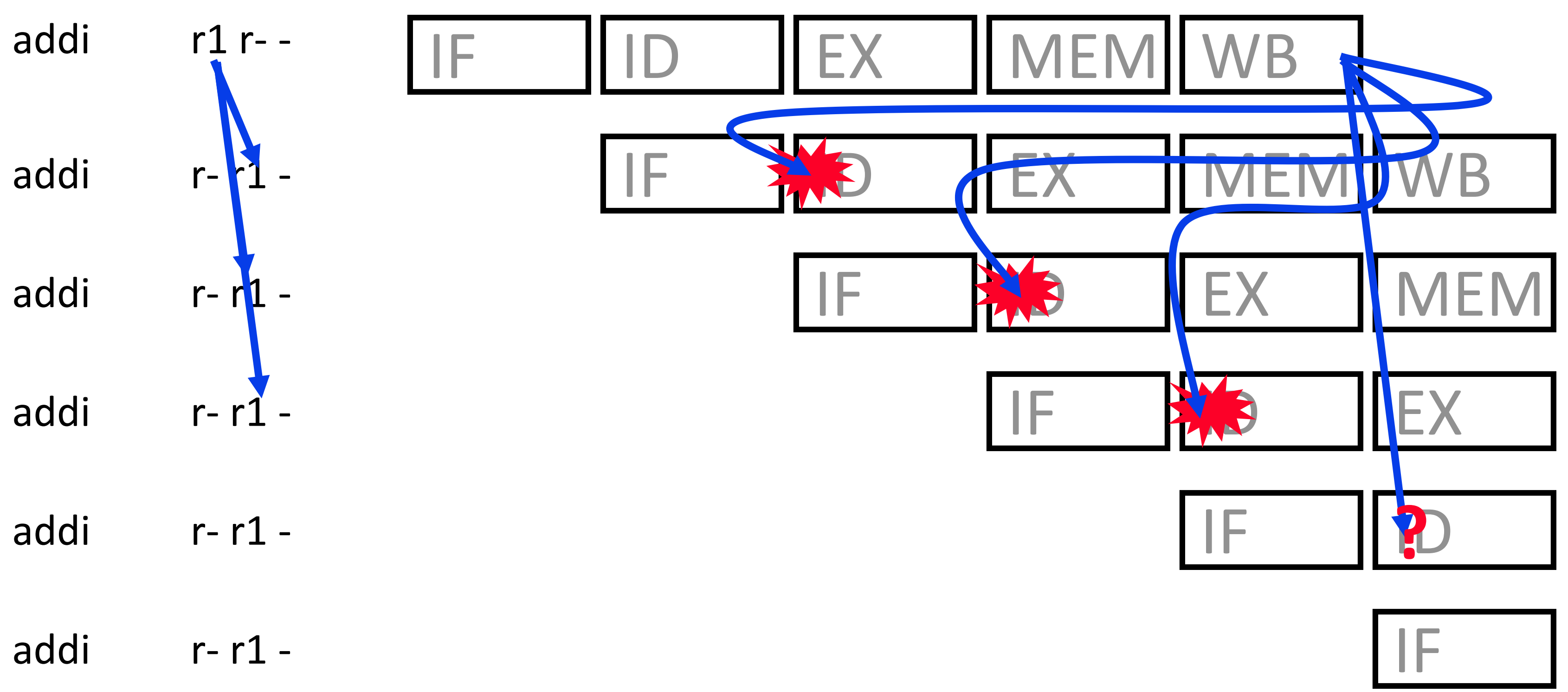
$r_3 \leftarrow r_1 \text{ op } r_2$

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

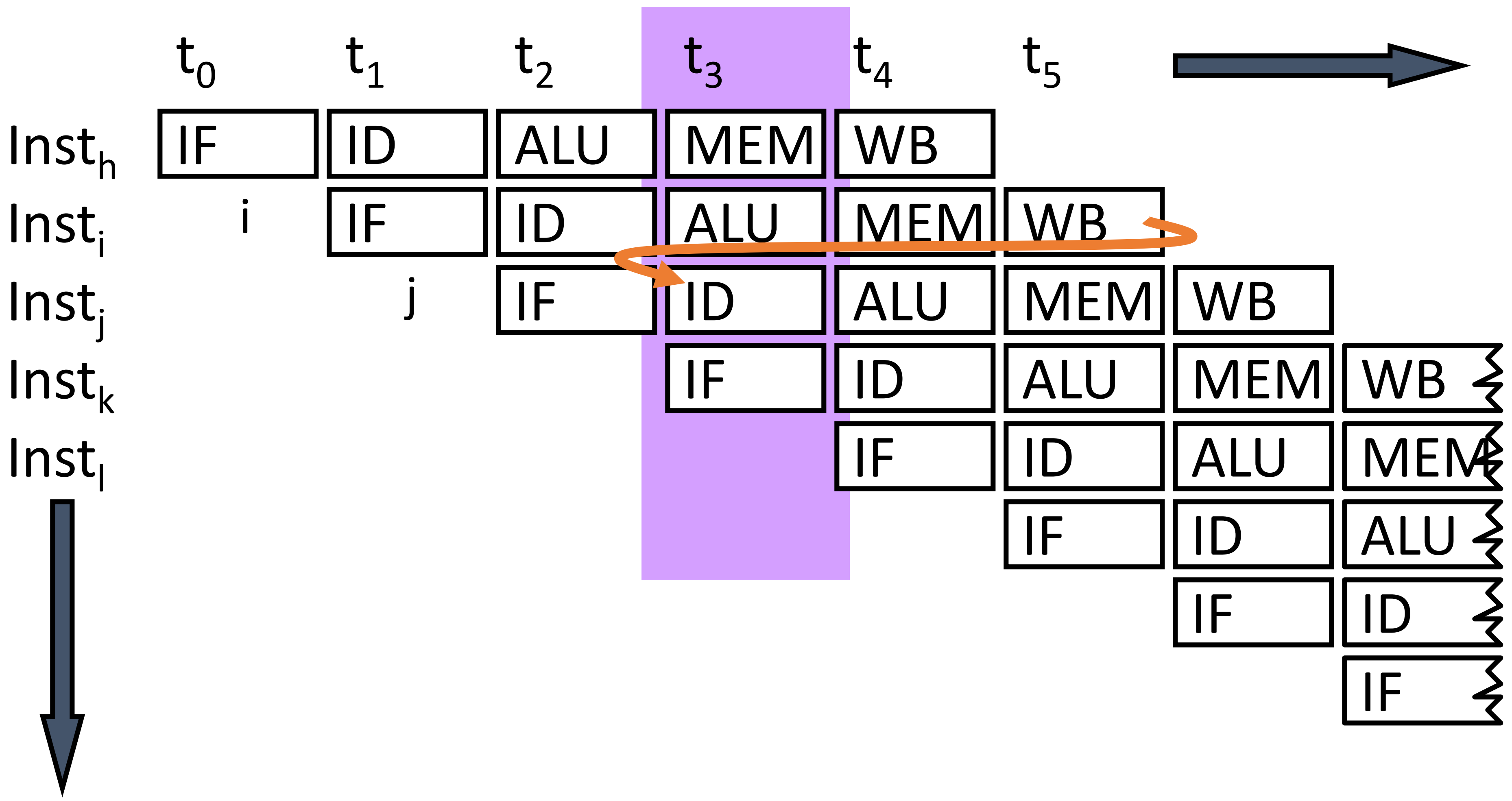
Read-after-Write (RAW)

RAW Dependence Handling

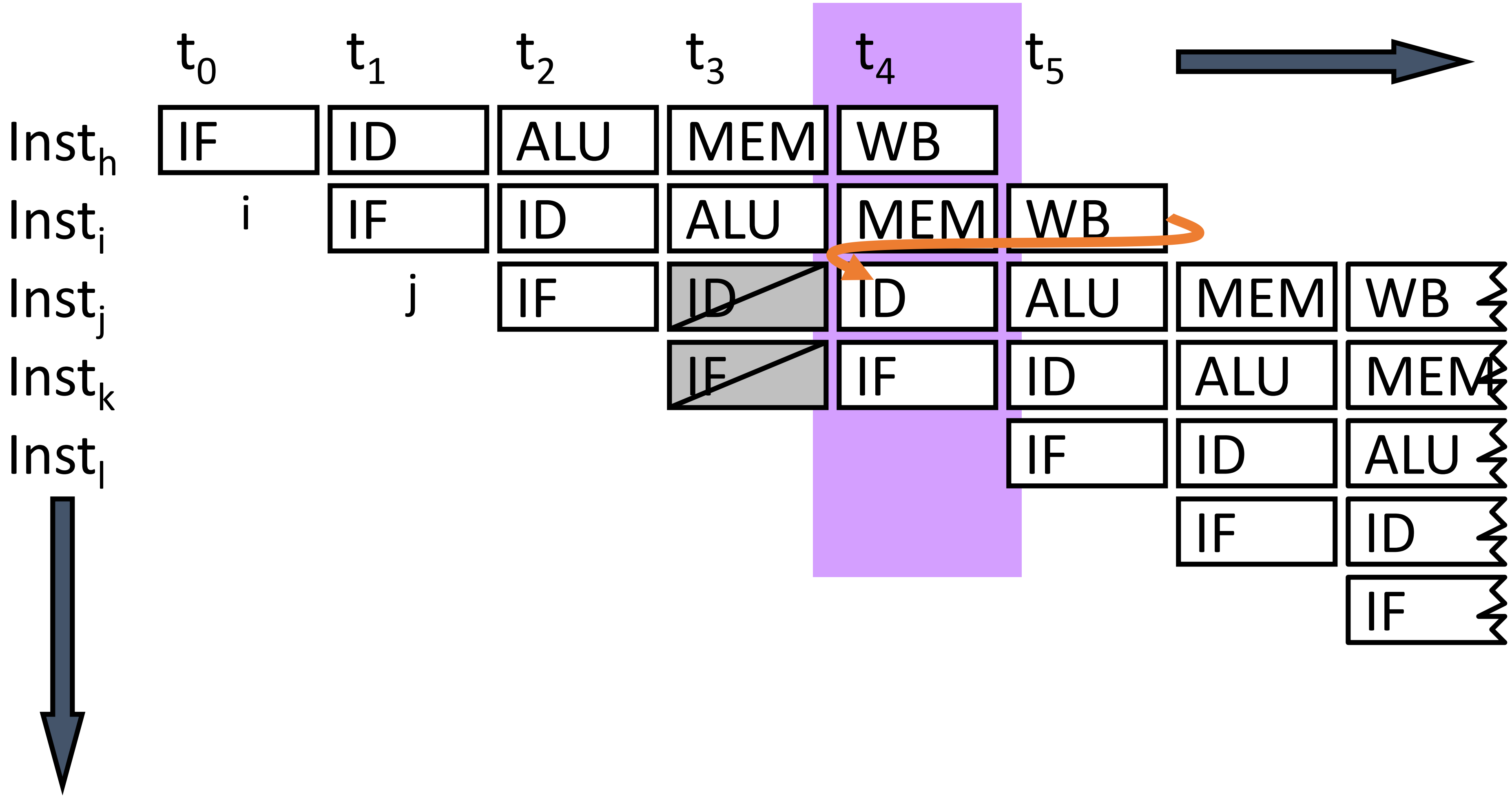
- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?



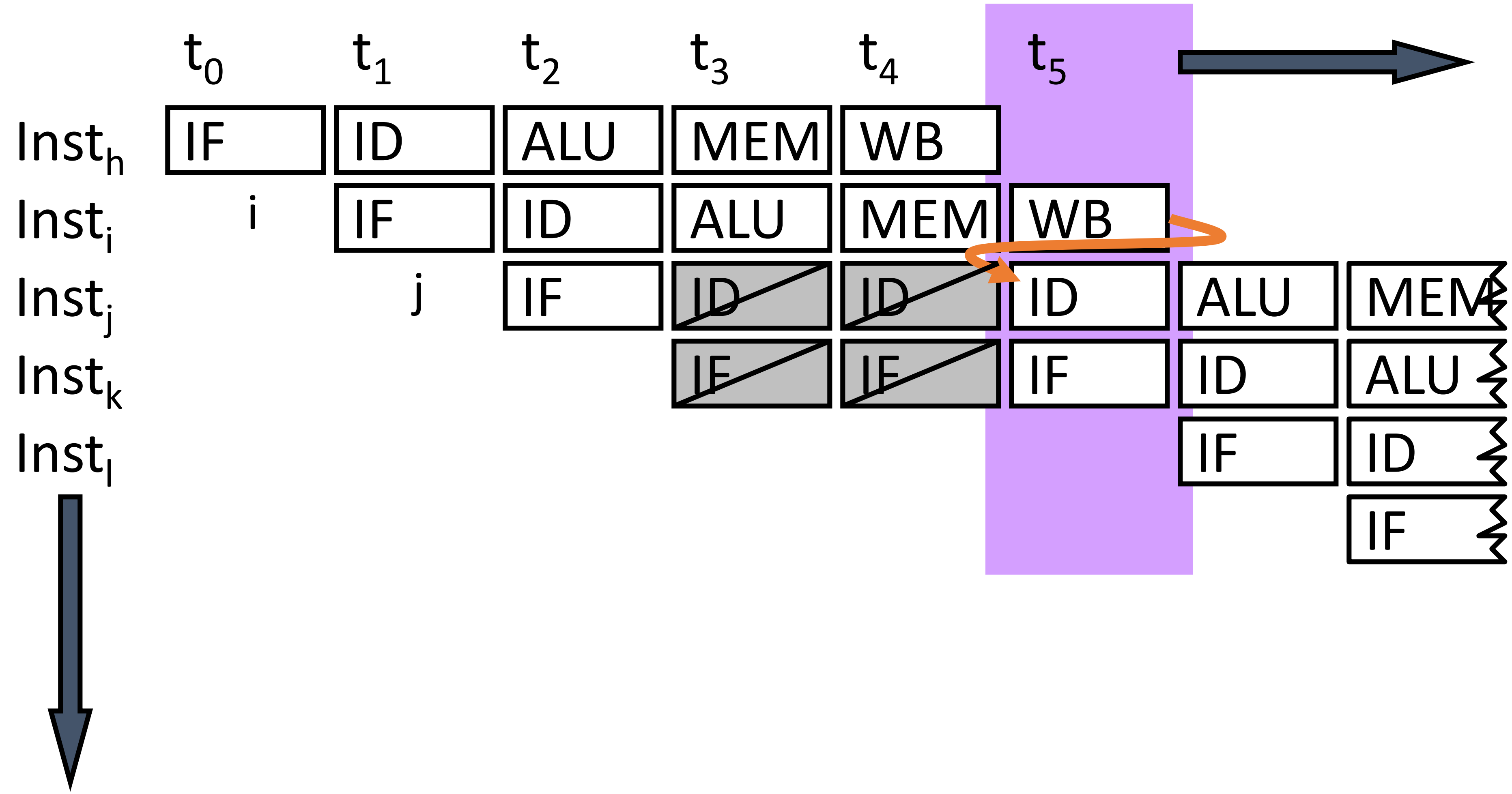
Pipeline Stall: Resolving Data Dependence



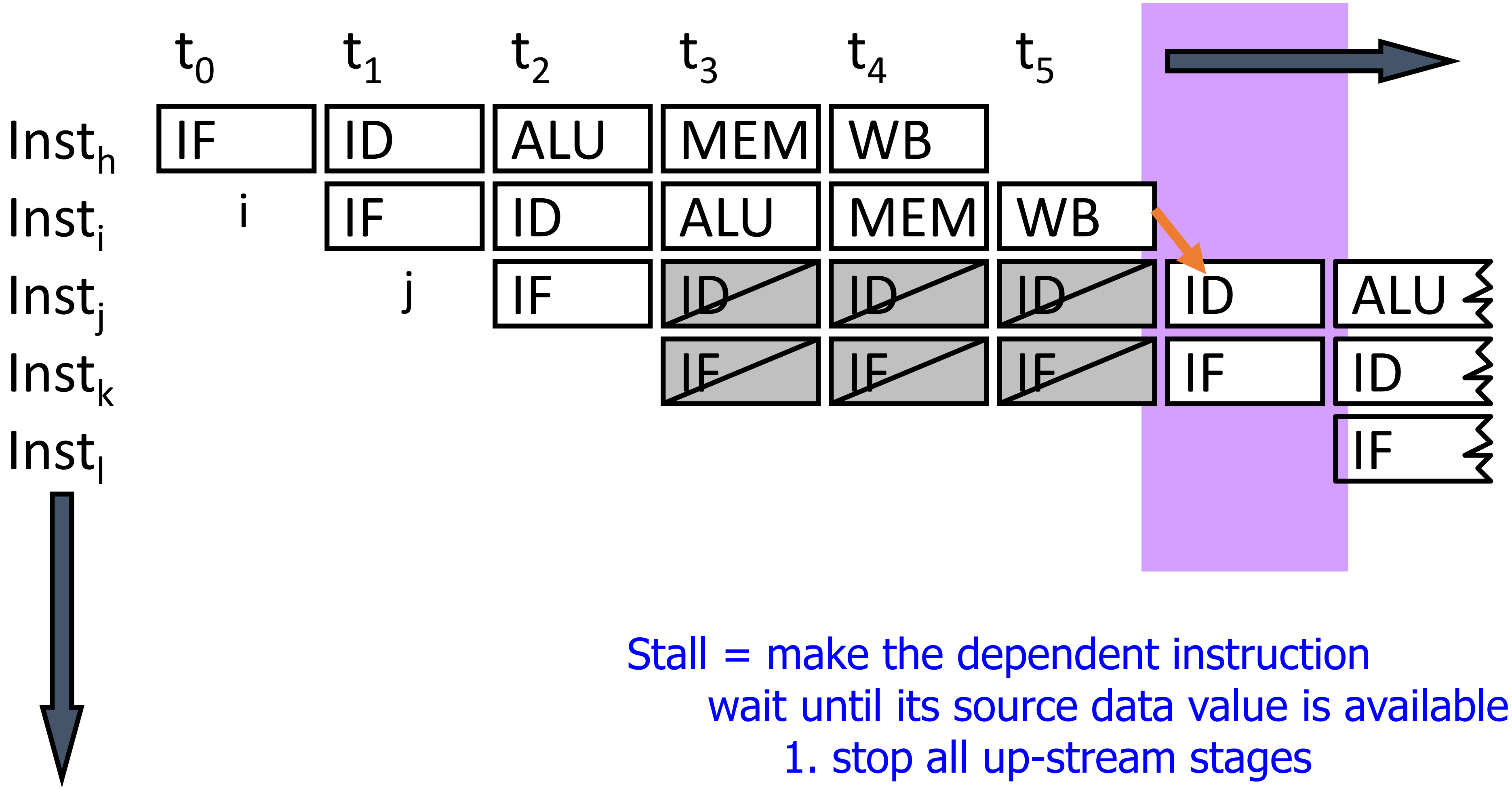
Pipeline Stall: Resolving Data Dependence



Pipeline Stall: Resolving Data Dependence



Pipeline Stall: Resolving Data Dependence



Stall = make the dependent instruction
wait until its source data value is available
1. stop all up-stream stages
2. drain all down-stream stages

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 and destination registers (in case of more complex dependencies) are Valid
    - Yes: No need to stall… No dependence
    - No: Stall the instruction

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

RISC-V vs. MIPS

- **Note:** The following example is using the MIPS instruction set, but the concepts equally hold for RISC-V

- In MIPS assembler, the $ sign stands for a register.

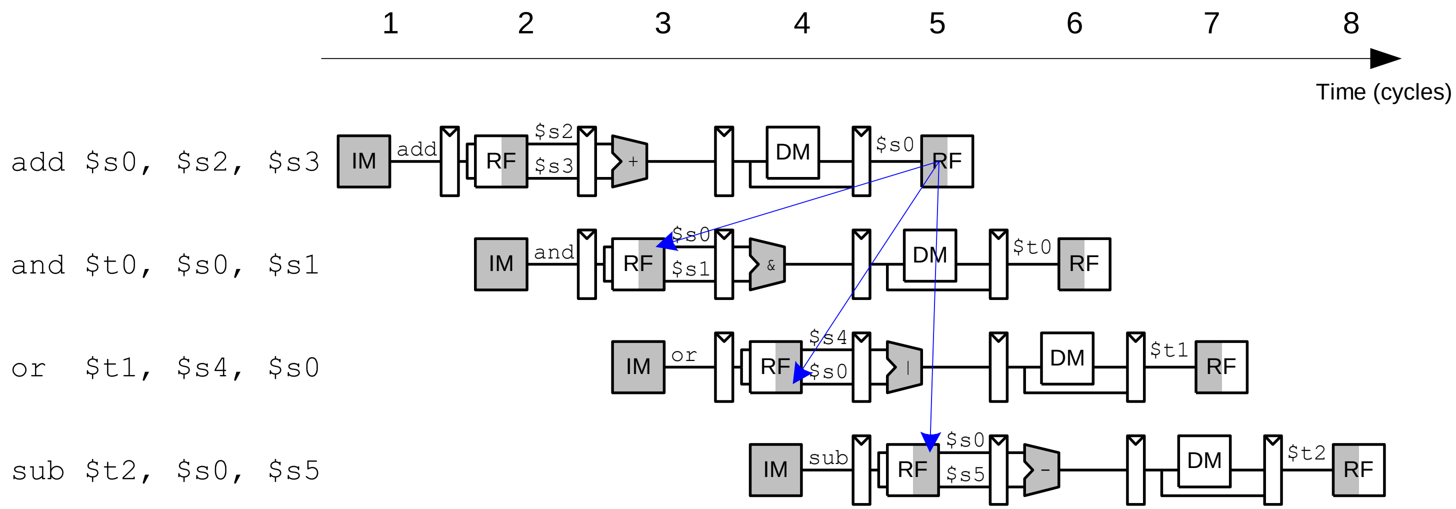
Example: load word instruction

Destination register
Offset
Address register

MIPS	**RISC-V**
`lw $1, 100($0)`	`lw r1, 100(r0)`

RAW Data Dependence Example

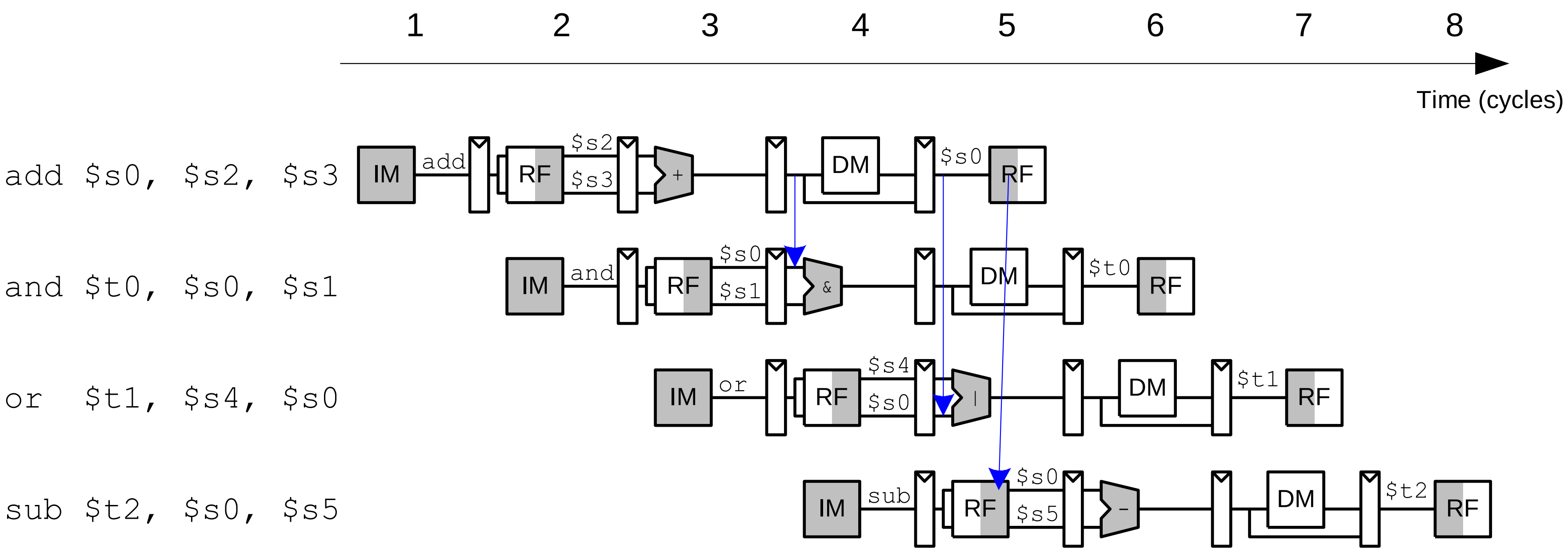
- One instruction writes a register ($s0) and next instructions read this register => read after write (RAW) dependence.
  - ***add*** writes into $s0 in the first half of cycle 5
  - ***and*** reads $s0 on cycle 3, obtaining the wrong value
  - ***or*** reads $s0 on cycle 4, again obtaining the wrong value.
  - ***sub*** reads $s0 in the second half of cycle 5, obtaining the correct value
  - subsequent instructions read the correct value of $s0



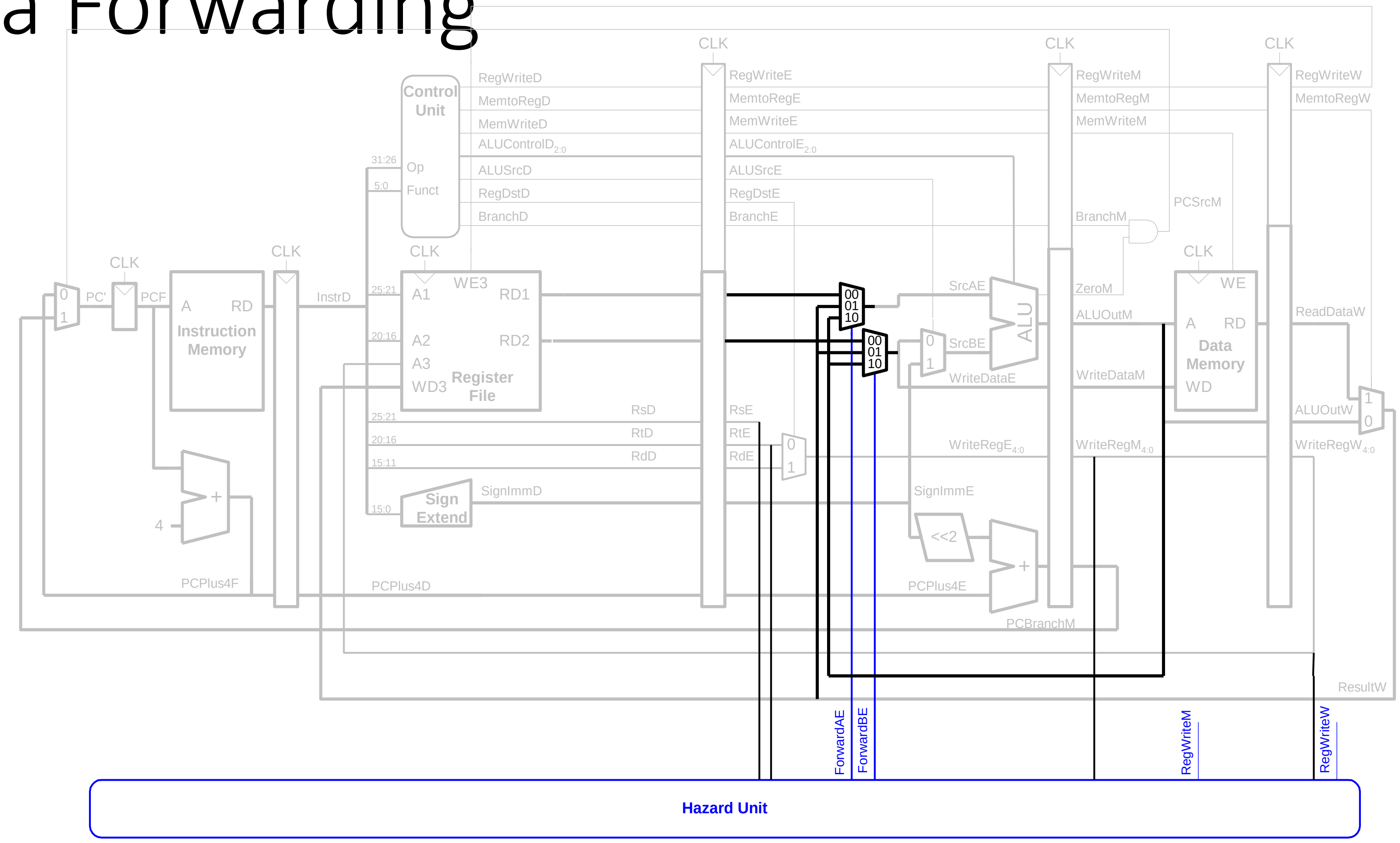
Data Forwarding

- 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

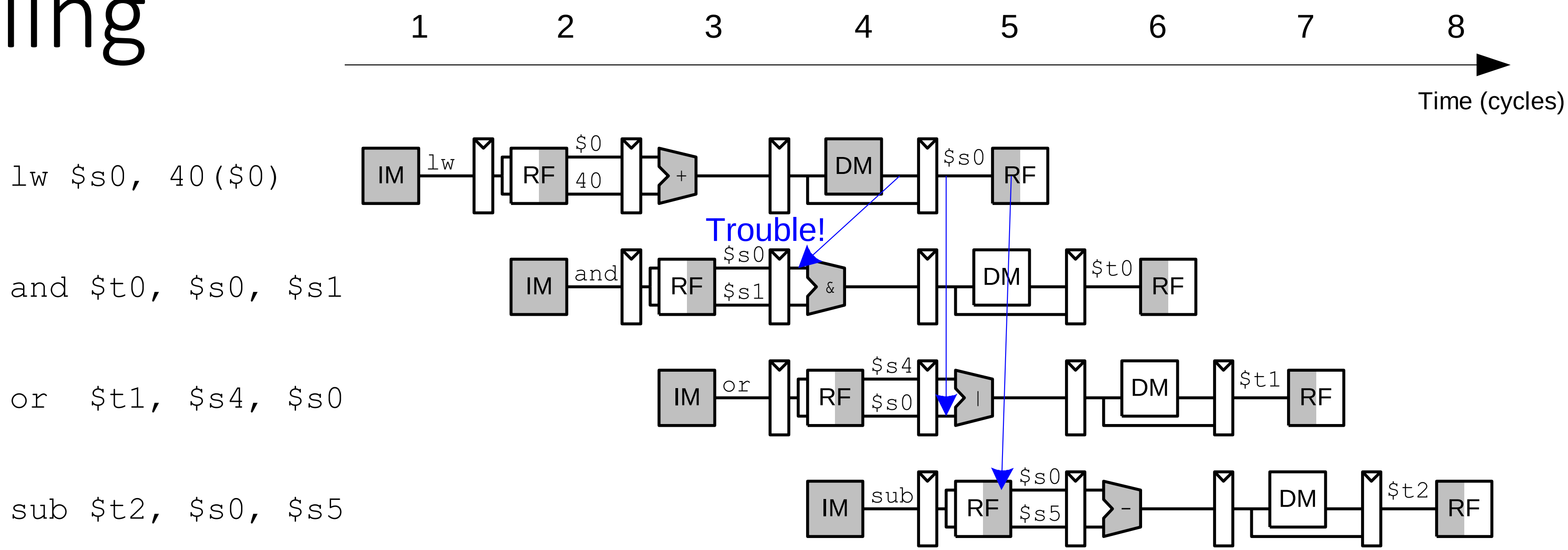
Data Forwarding



Data Forwarding

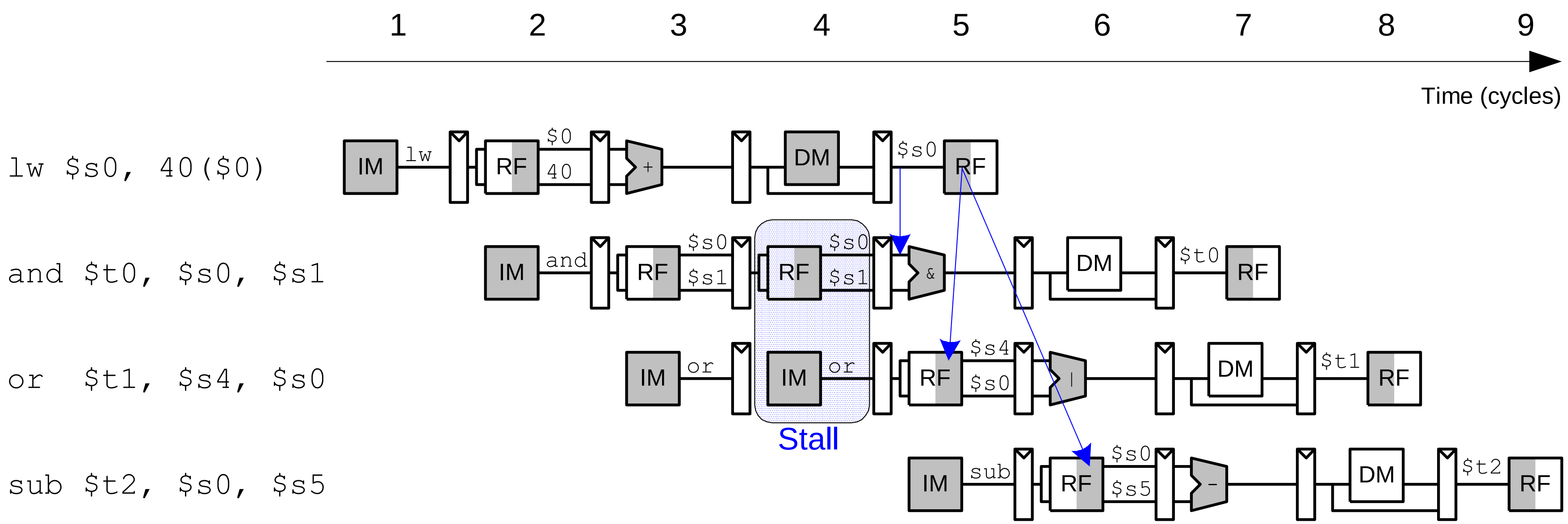


Stalling



- 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


1
2
3
4
5
6
7
8
9
Time (cycles)
lw $s0, 40($0)
IM
lw
RF
$0
40
+
DM
$s0
RF
and $t0, $s0, $s1
IM
and
RF
$s0
$s1
RF
$s0
$s1
&
DM
$t0
RF
or $t1, $s4, $s0
IM
or
IM
or
RF
$s4
$s0
|
DM
$t1
RF
Stall
sub $t2, $s0, $s5
IM
sub
RF
$s0
$s5
-
DM
$t2
RF