Computer Organization and Networks

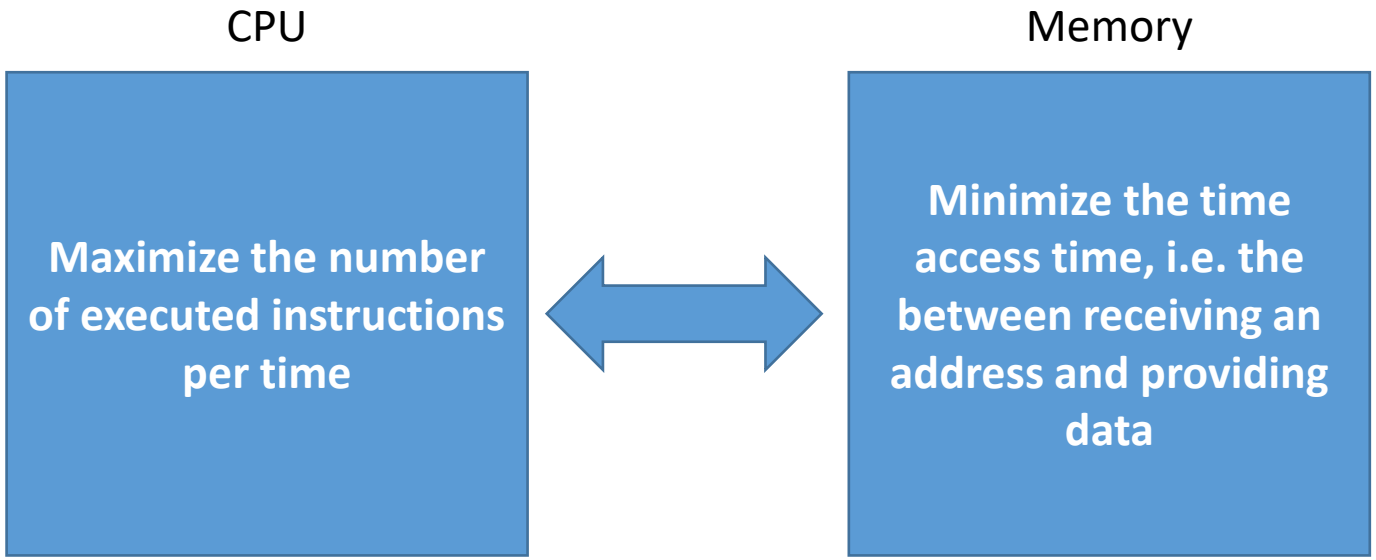
(INB.06000UF, INB.07001UF)

Chapter 10: Pipelining

Winter 2023/2024

Stefan Mangard, www.iaik.tugraz.at

The Need for Speed



- The speed of the CPU and the memory needs to be match

- We have learned in the previous chapter how to build a typical memory system → we now look at the CPU

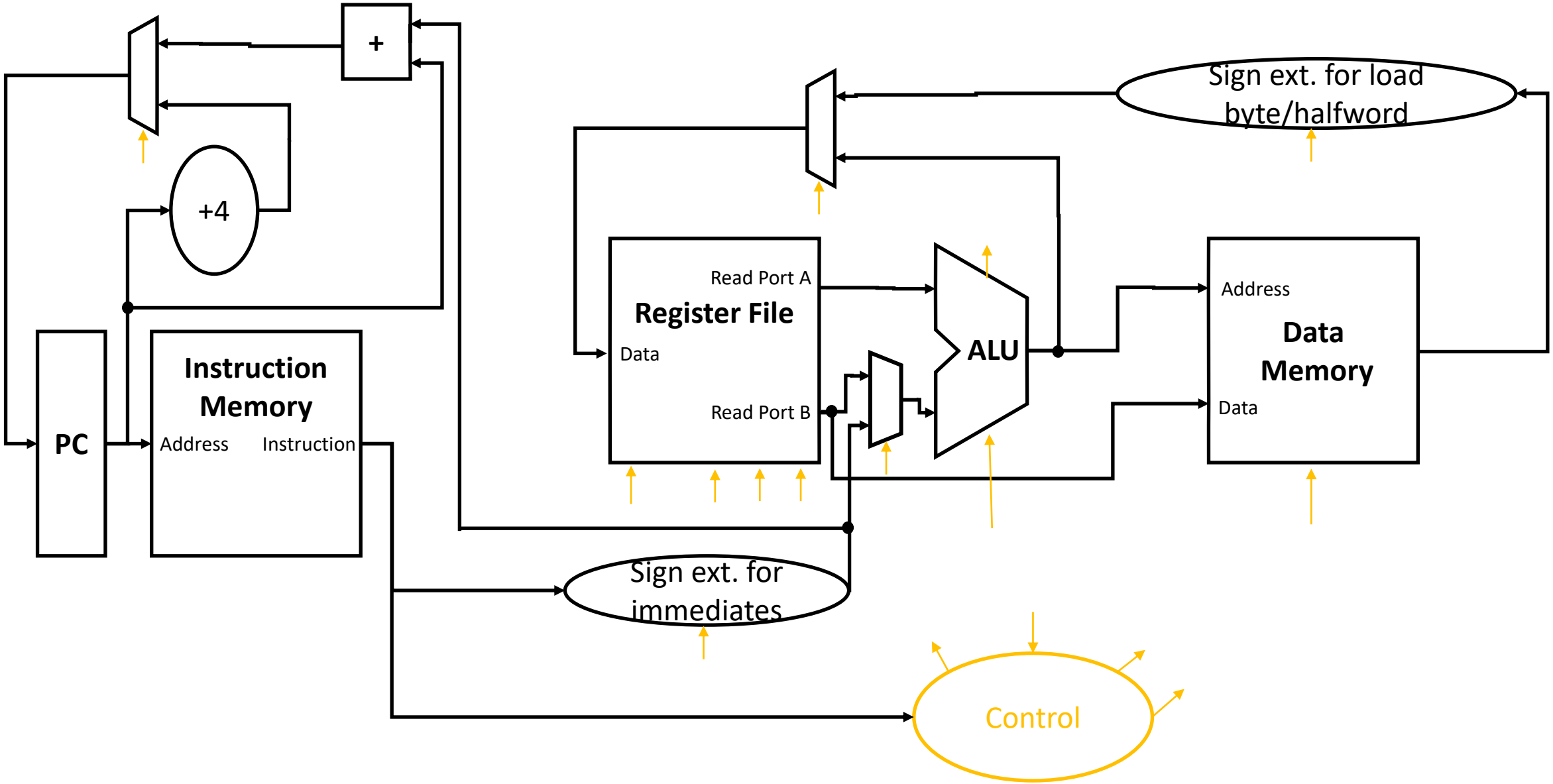
Goal

- The goal of processor design is maximizing the executed number of instructions per time

- This is determined by two factors
  - The needed clock cycles per instruction (CPI)
  - The clock frequency, which determines the number of cycles per second

- The execution time for a program with N instructions is N * CPI * (1/f)
  - f is the clock frequency (1/f is the clock period)
  - CPI is the average number of cycles per instruction

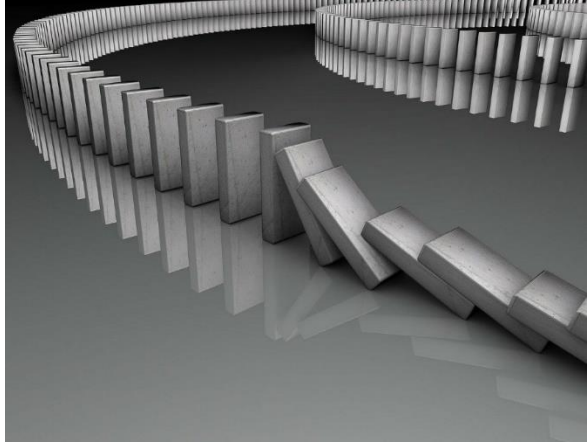
High-Level Overview (Single Cycle Datapath)

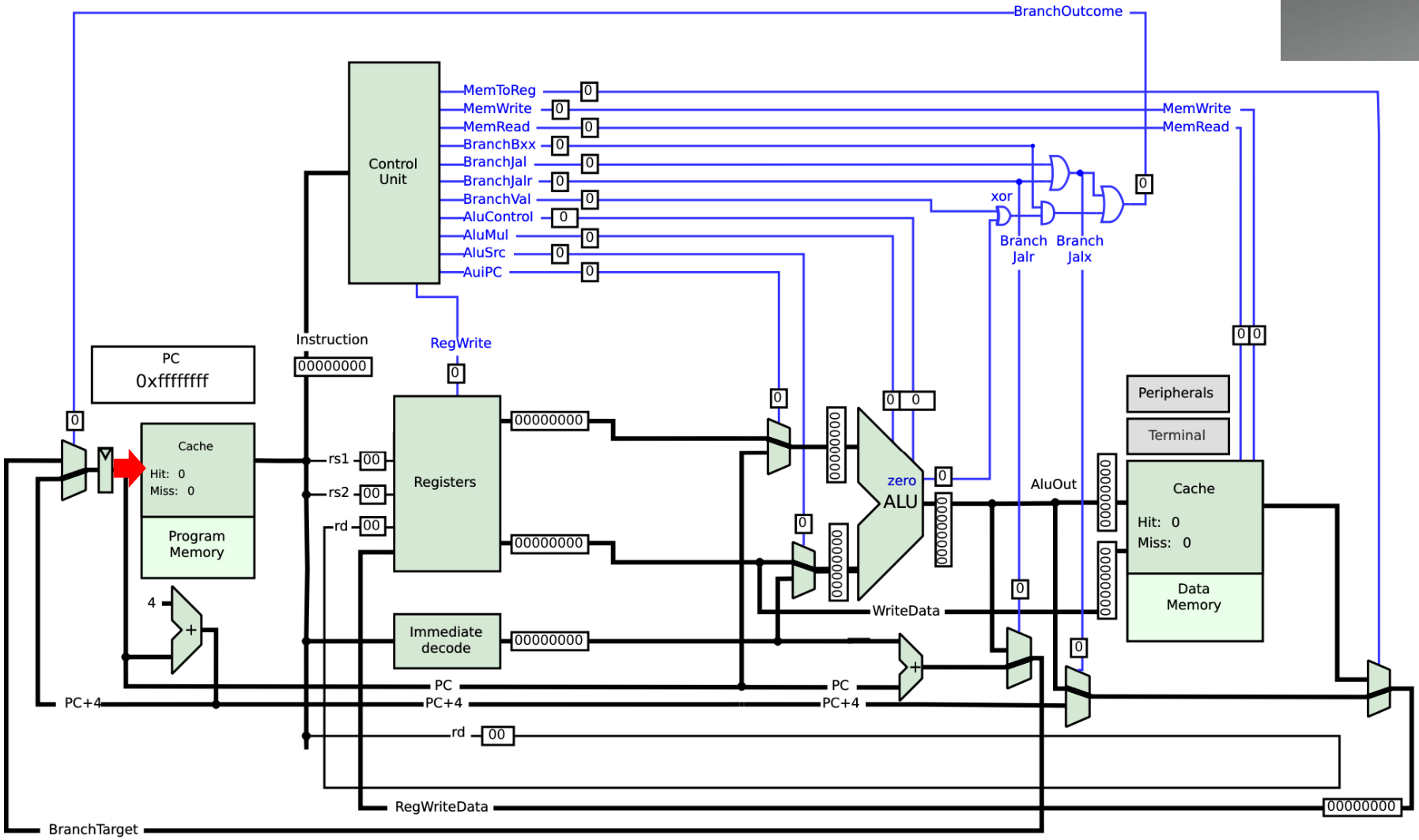


Performance of the Single-Cycle Design

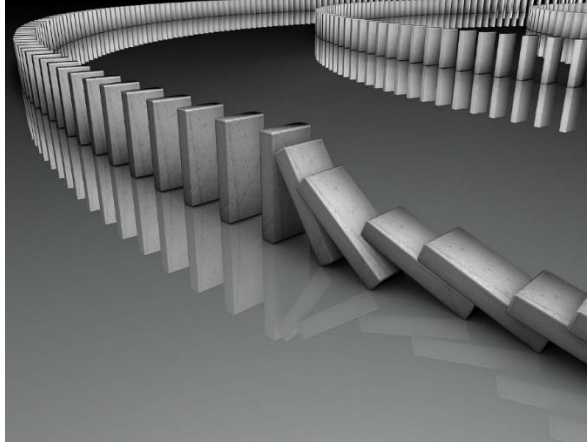
- Each instruction takes exactly one cycle to execute
- The maximum clock frequency is defined by the slowest instruction of the design
  - Remember: the critical path is the longest combinational path in the design.
  - The critical path of the slowest instruction therefore defines the clock frequency of our processor

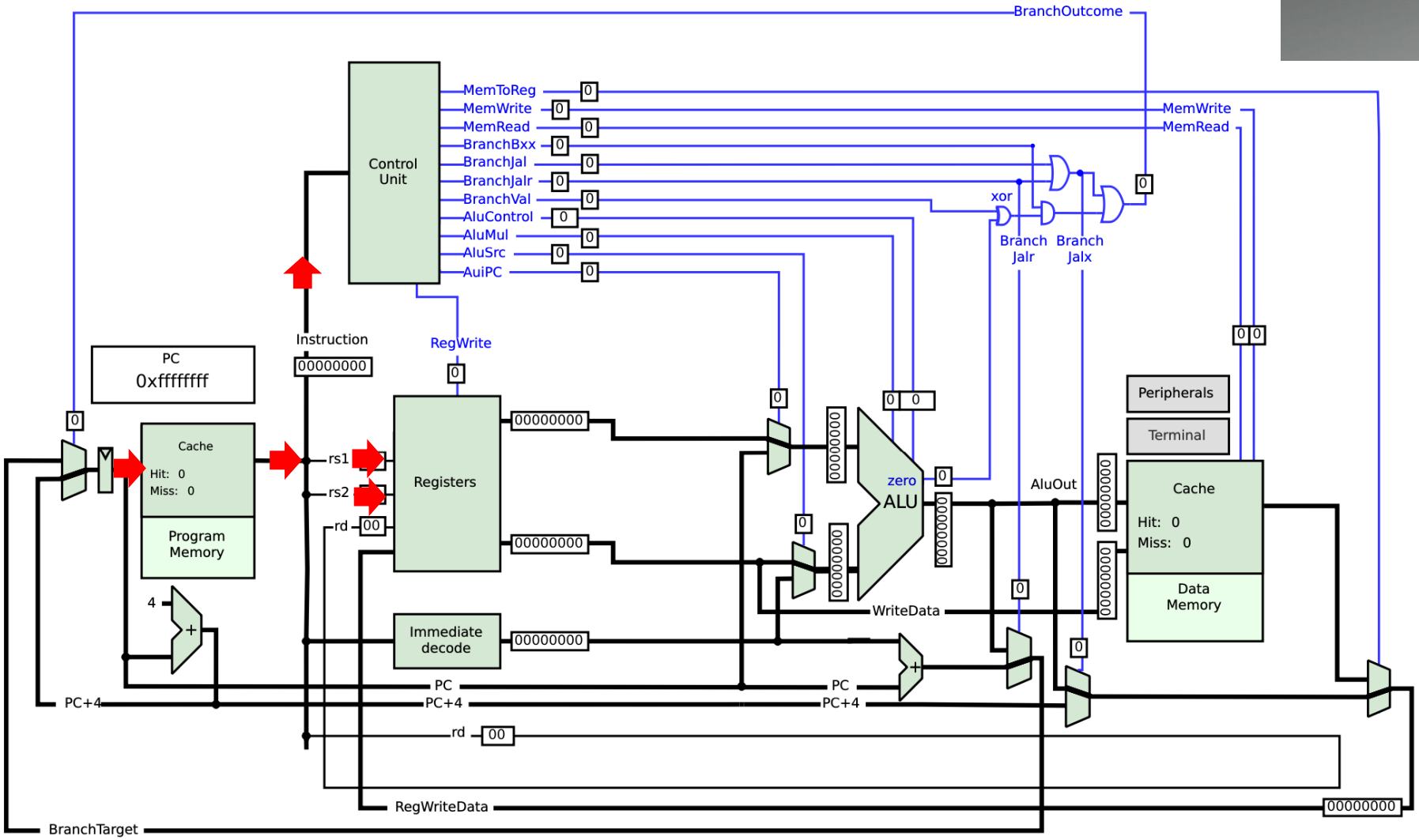
Critical Path



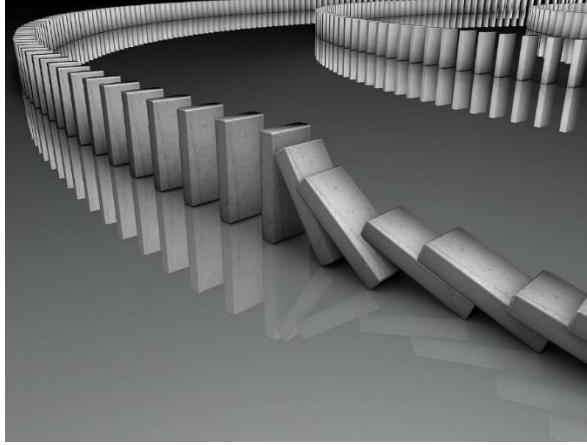


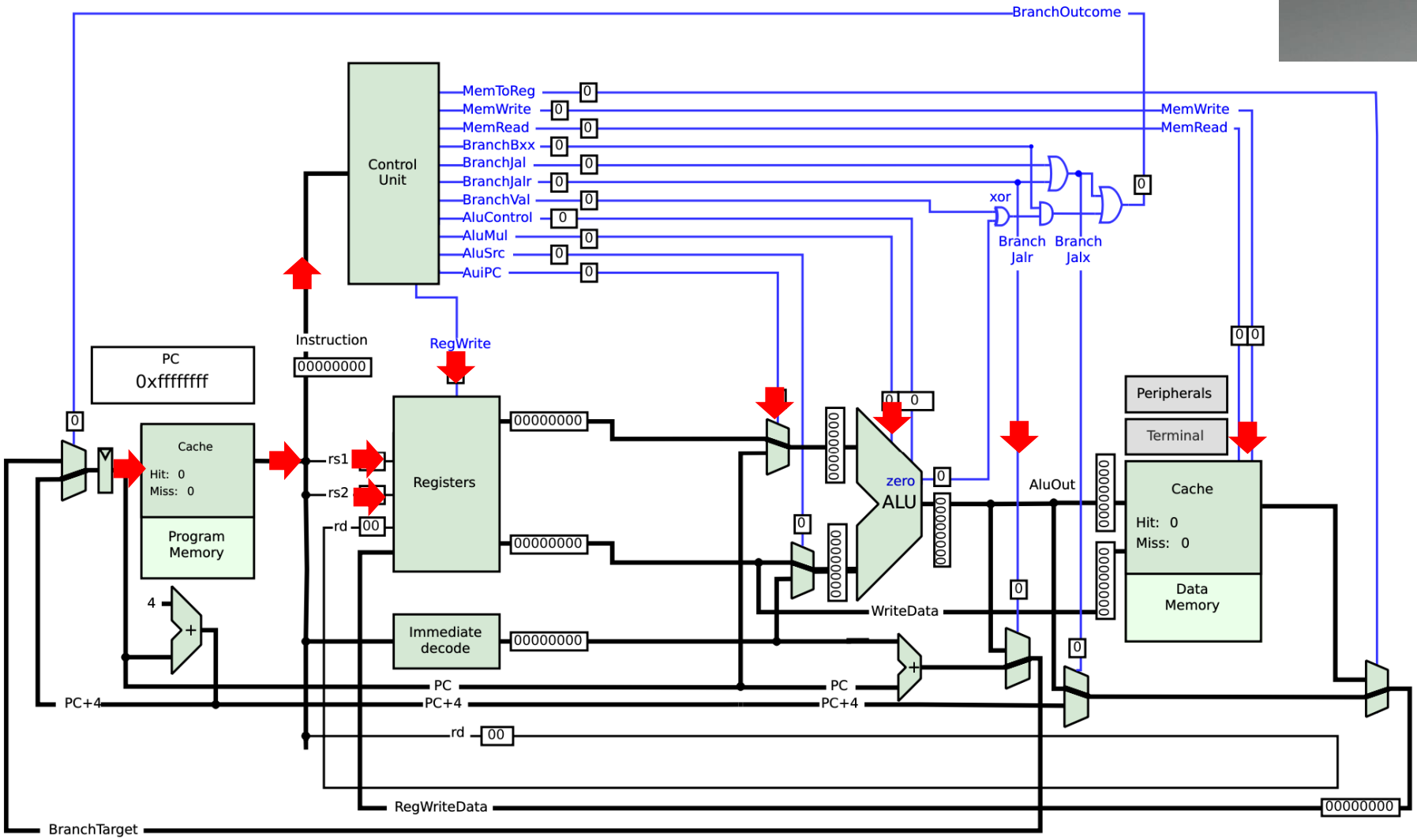
Critical Path



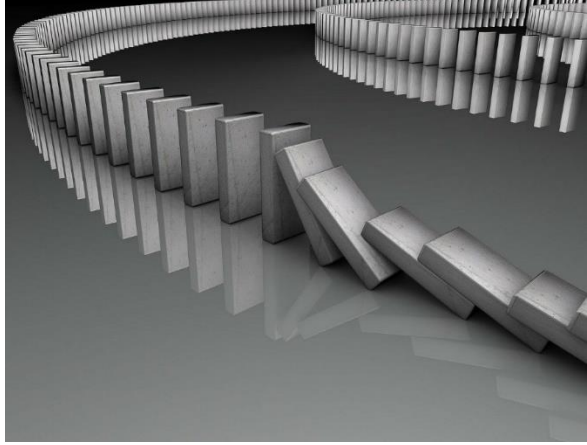


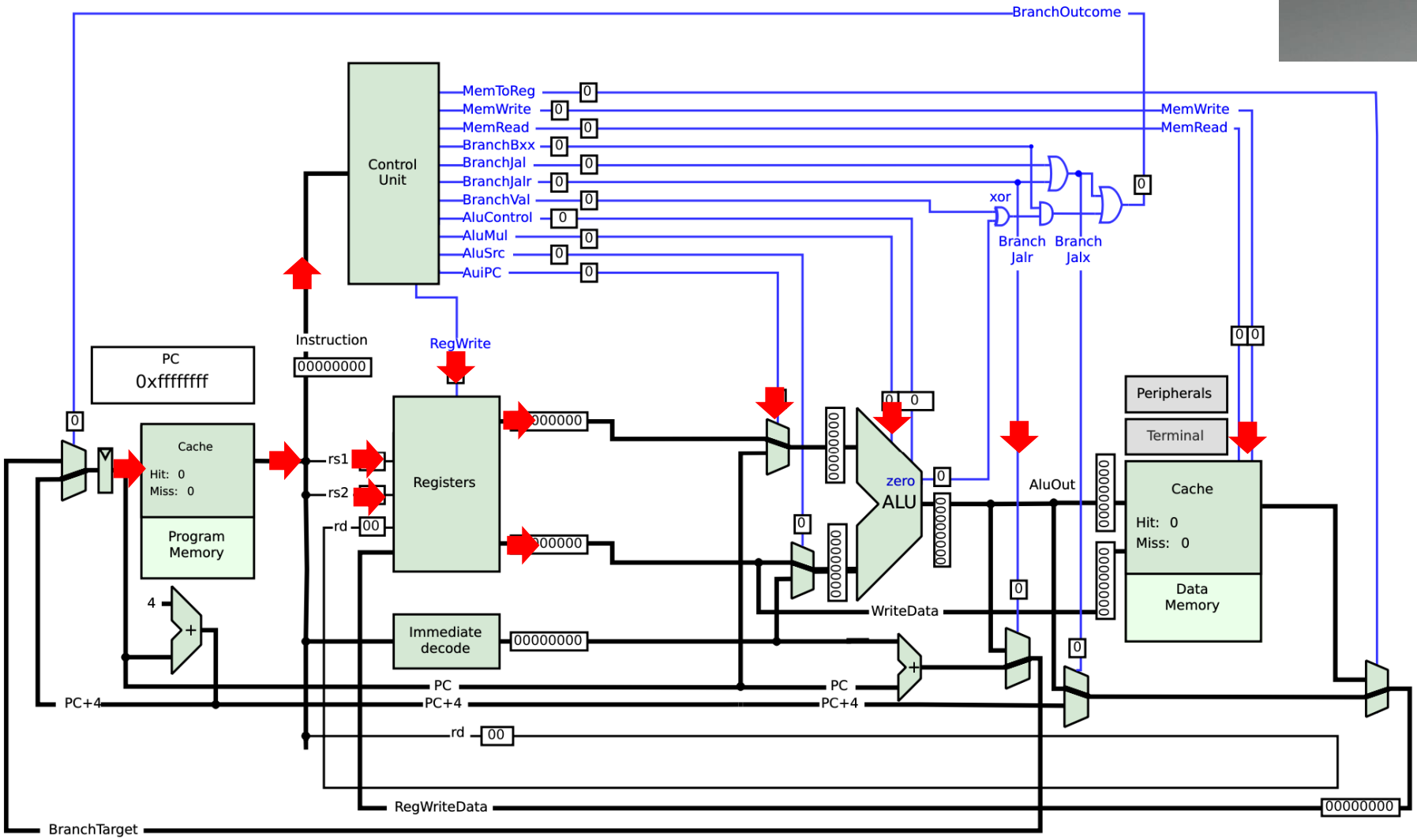
Critical Path



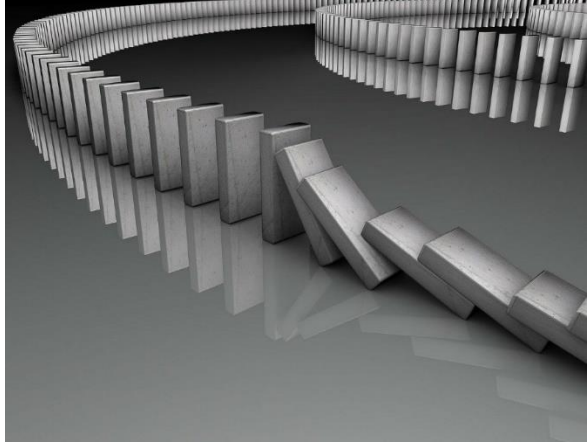


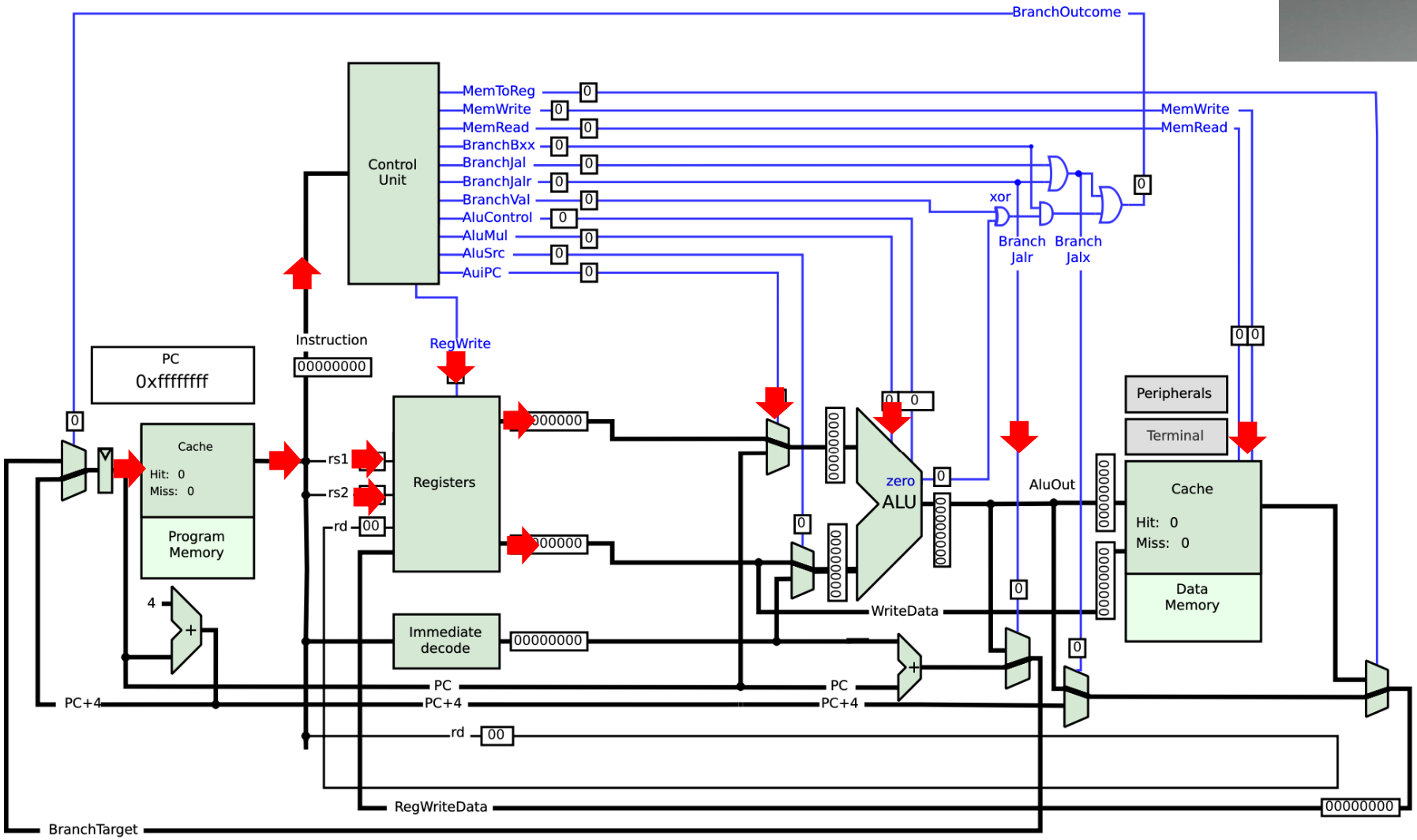
Critical Path



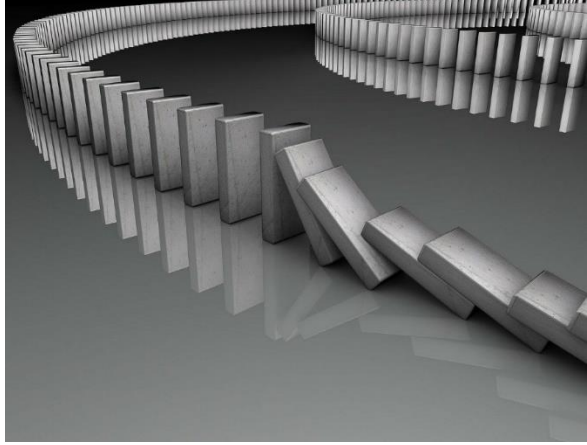


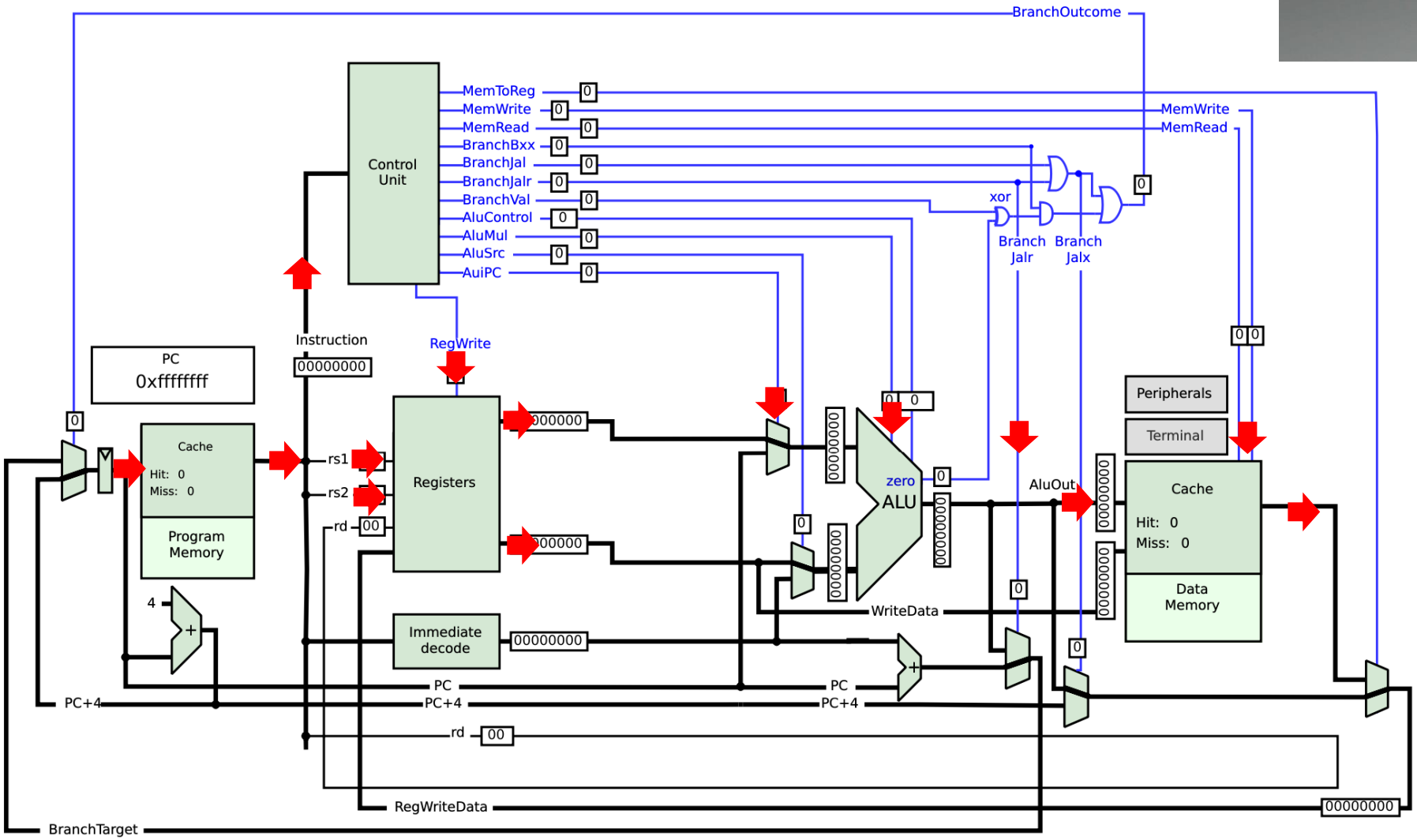
Critical Path



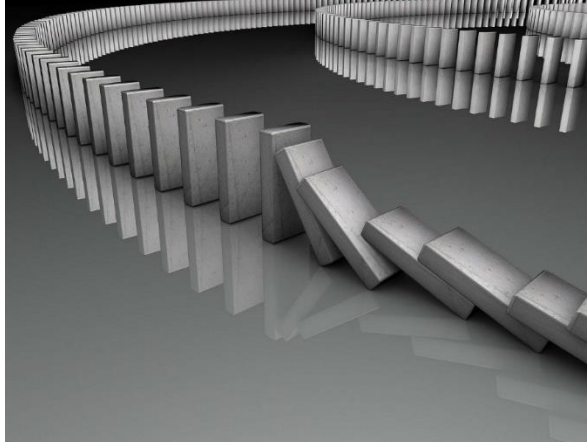


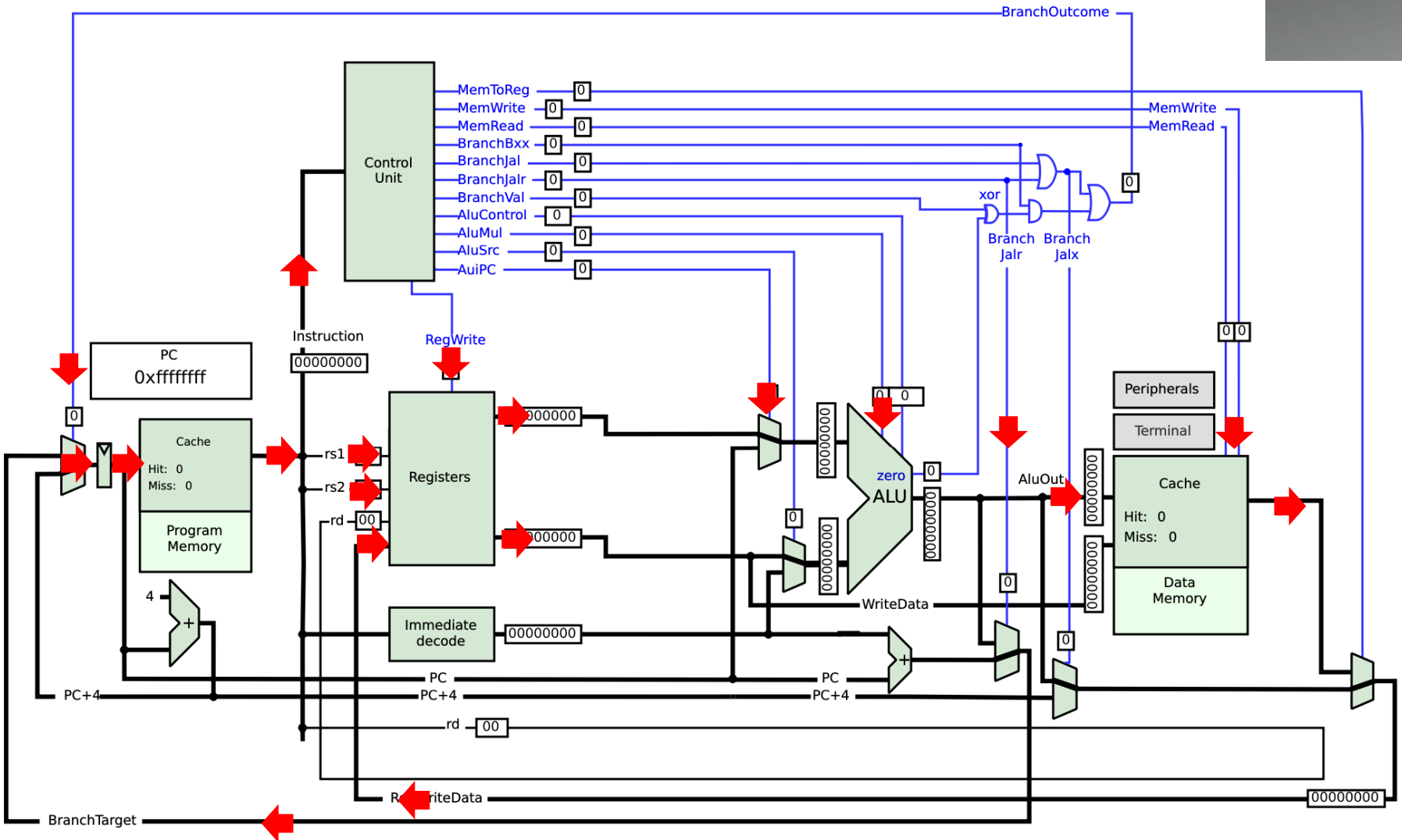
Critical Path



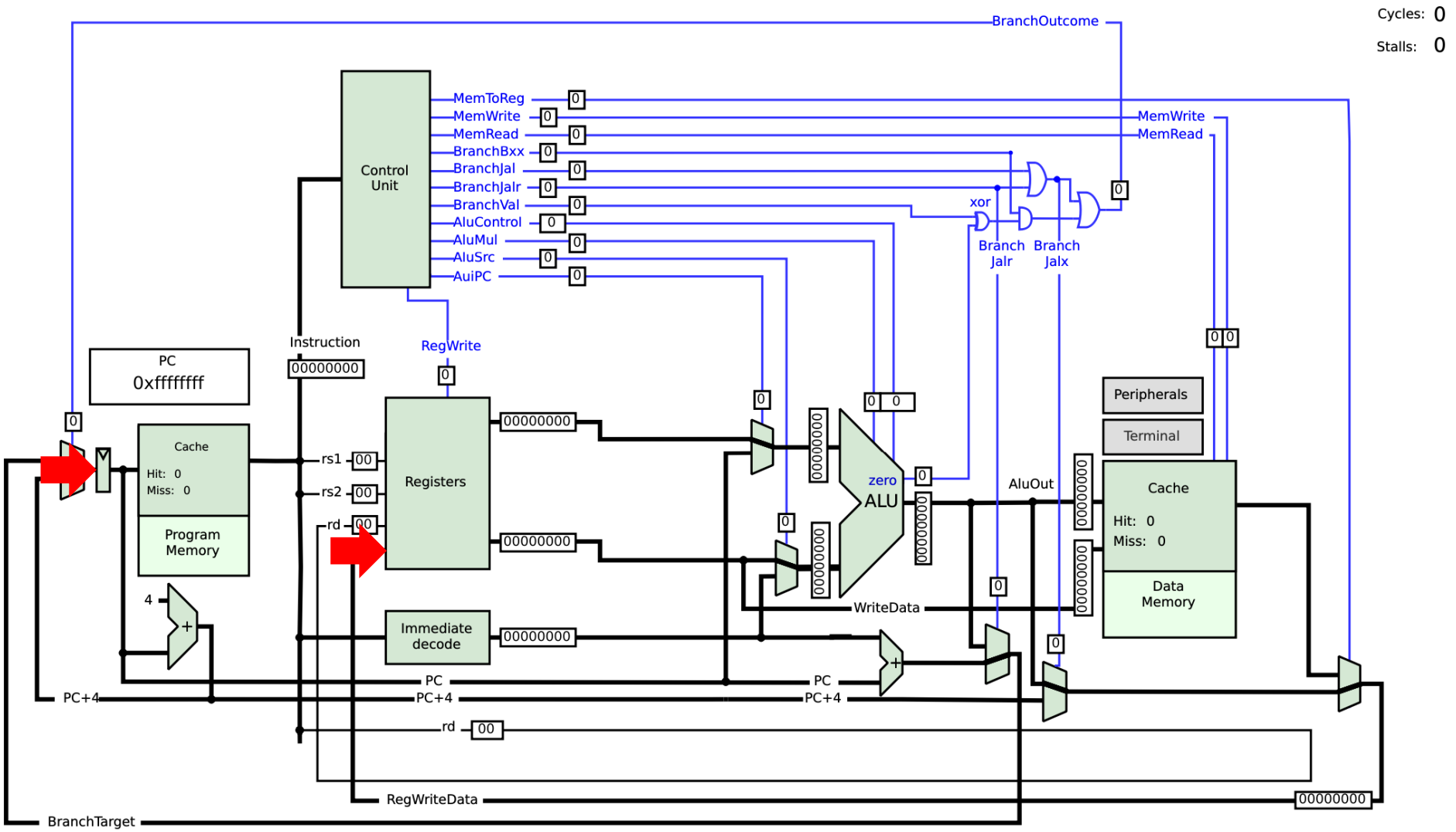


Critical Path





Critical Path



How can we improve the performance?

Note: making the building blocks (memories, logic gates, …) of the processor faster, won't make us significantly faster → we need a differ design approach

Basic Idea of Multicycle Architectures

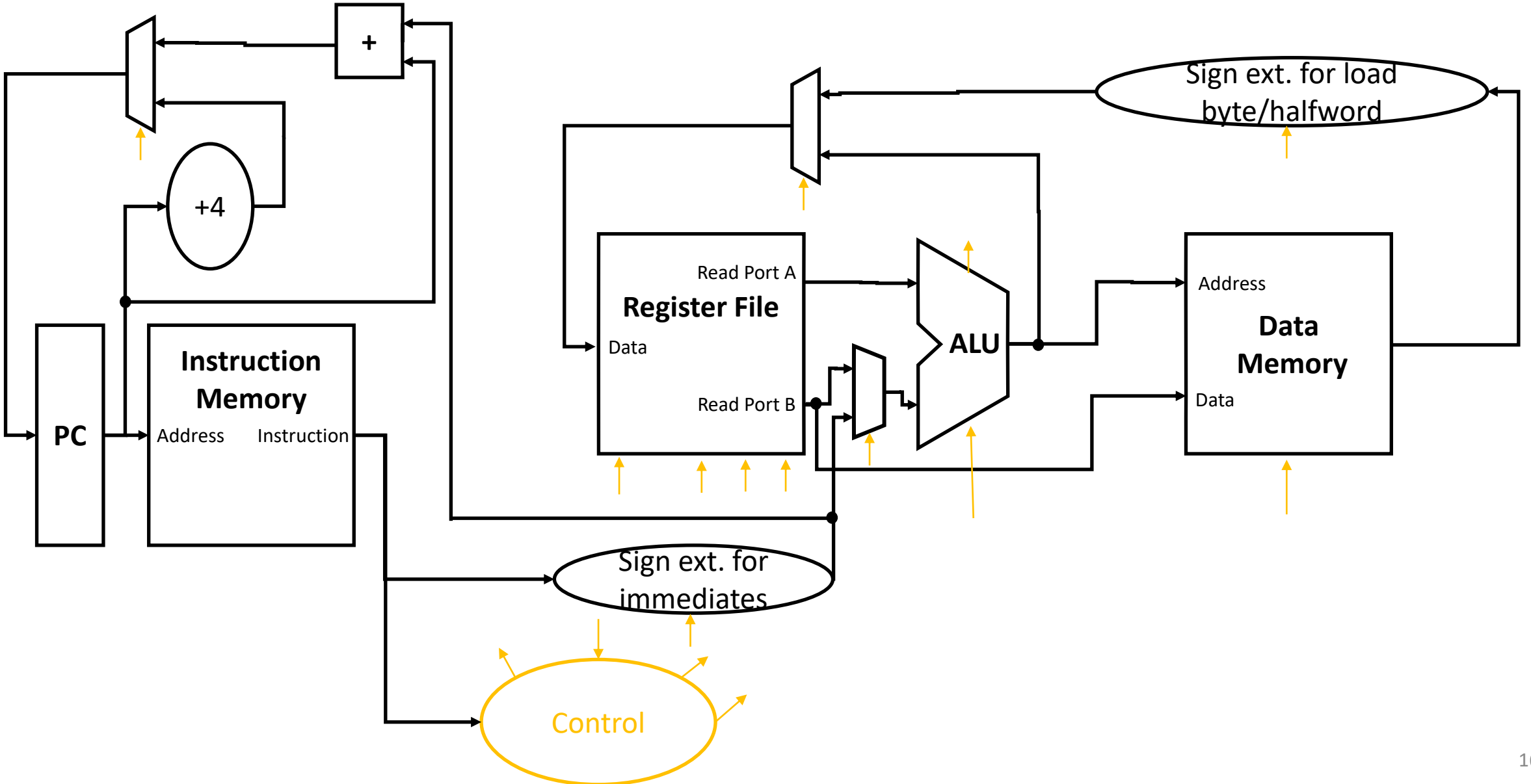
- Cut the operations that are needed for one instruction into more fine-granular operations

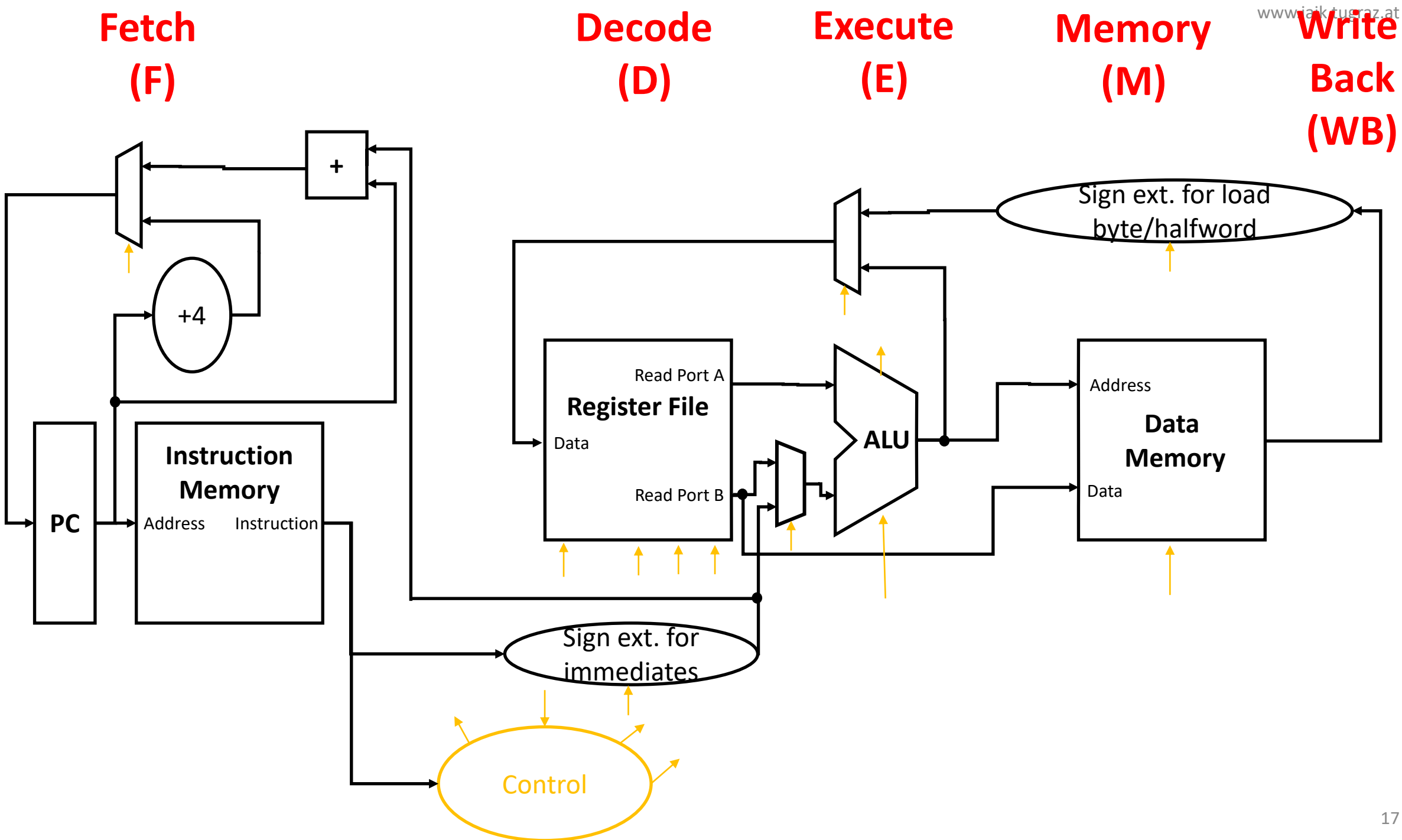
- Each instruction is a multicycle instruction and takes as many cycles as needed to perform the actions defined by the instruction

  →Instructions lead to different numbers of operations (and therefore take longer / shorter depending on their complexity)

  →The operations that are done in a clock cycle are less complex and the clock frequency can be increased

High-Level Overview (Single Cycle Datapath)




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

Fetch (F)

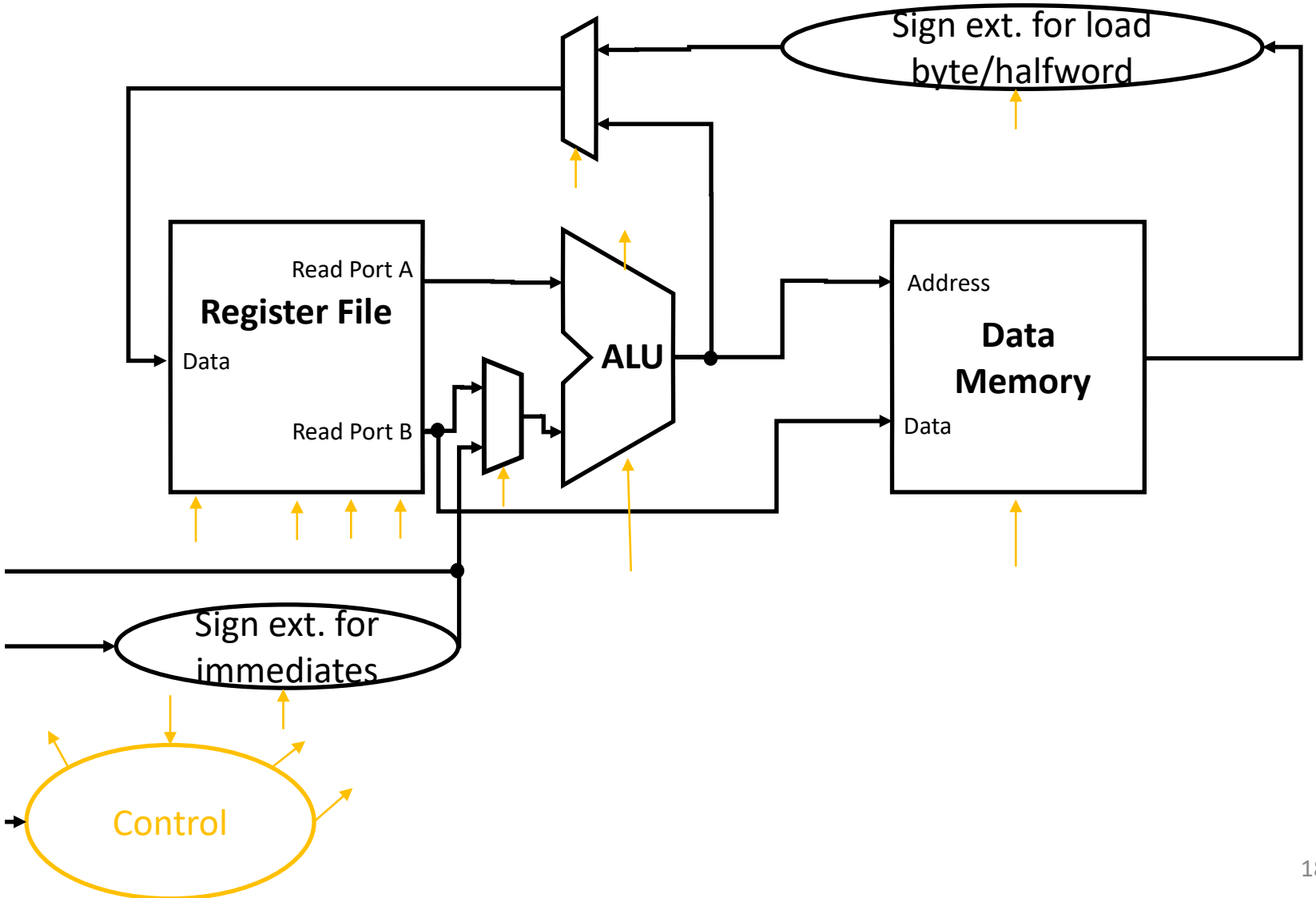
Decode (D)

Execute (E)

Memory (M)

Write Back (WB)

Given an address, read an instruction from memory



Fetch (F)

Given an address, read an instruction from memory

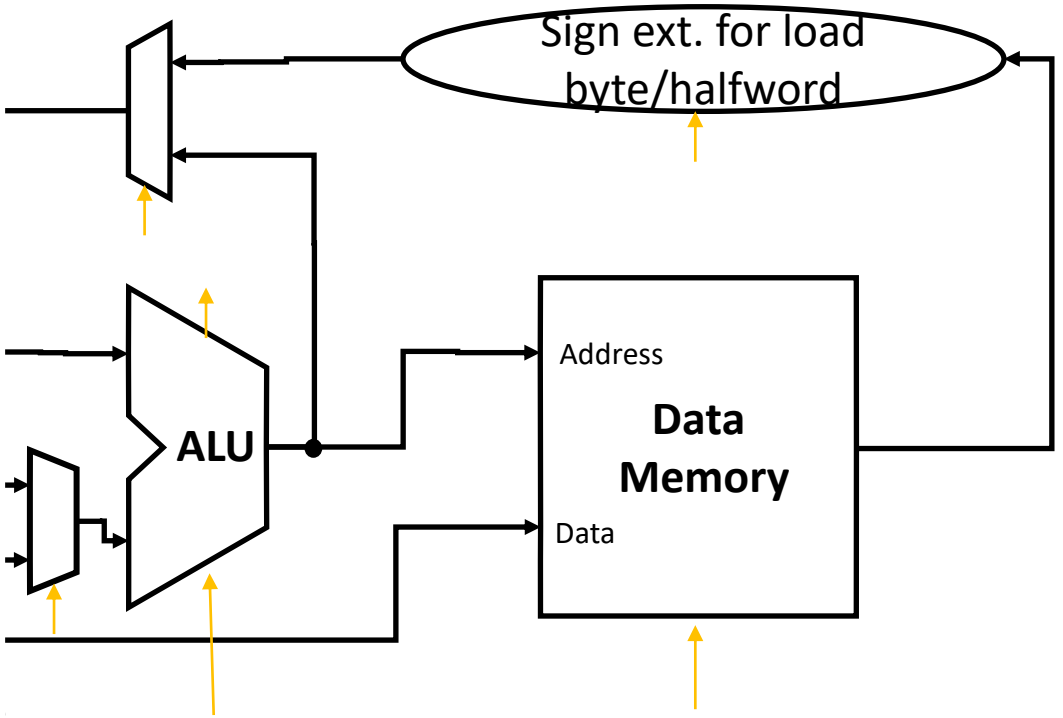
Decode (D)

Given an instruction, determine the corresponding control signals for the data path

Execute (E)

Memory (M)

Write Back (WB)



Fetch (F)

Given an address, read an instruction from memory

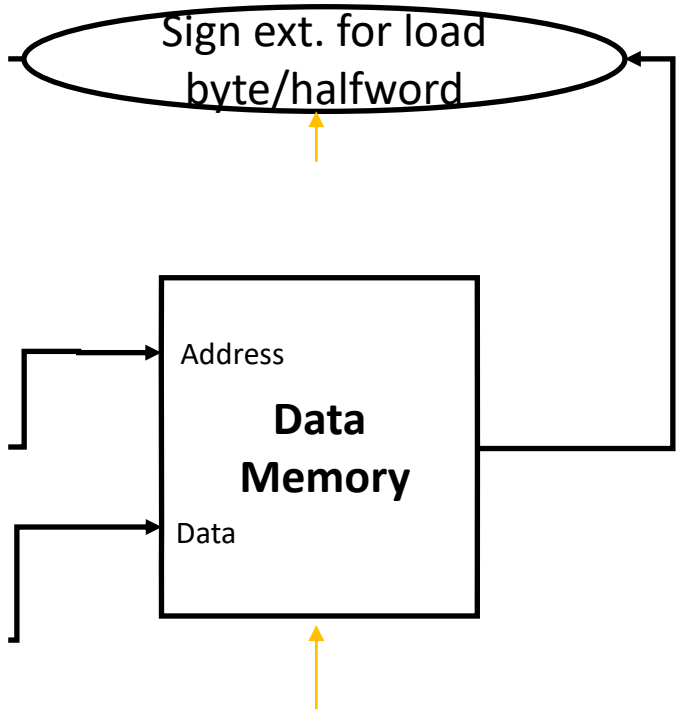
Decode (D)

Given an instruction, determine the corresponding control signals for the data path

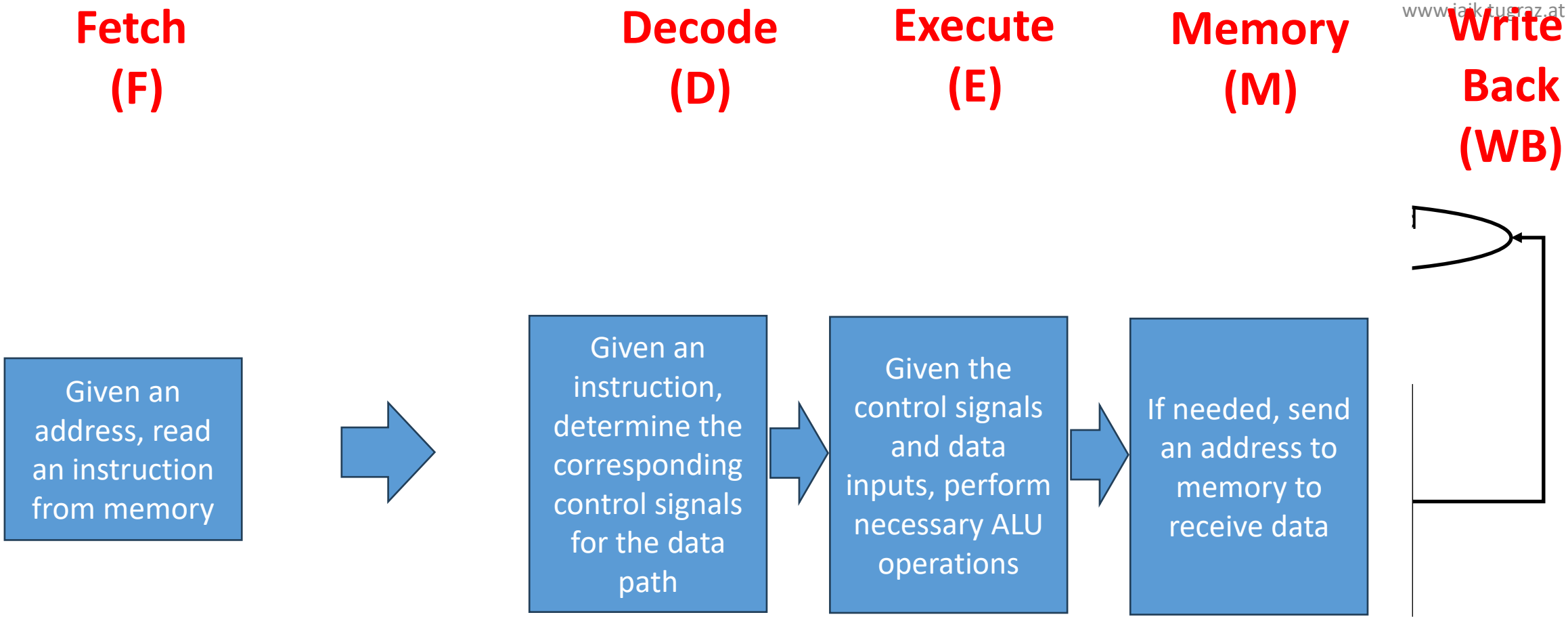
Execute (E)

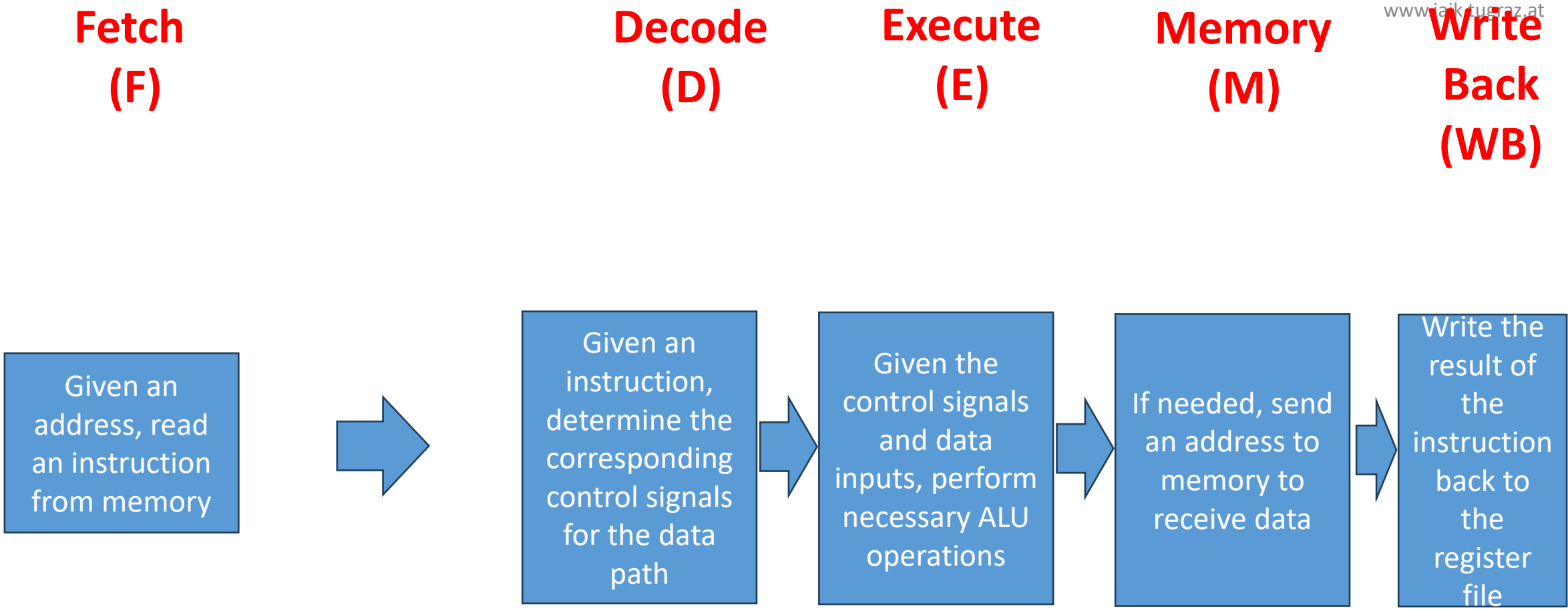
Given the control signals and data inputs, perform necessary ALU operations

Memory (M)



Write Back (WB)


Fetch (F)
Decode (D)
Execute (E)
Memory (M)
Write Back (WB)
Given an address, read an instruction from memory
Given an instruction, determine the corresponding control signals for the data path
Given the control signals and data inputs, perform necessary ALU operations
If needed, send an address to memory to receive data


Fetch (F)
Decode (D)
Execute (E)
Memory (M)
Write Back (WB)
Given an address, read an instruction from memory
Given an instruction, determine the corresponding control signals for the data path
Given the control signals and data inputs, perform necessary ALU operations
If needed, send an address to memory to receive data
Write the result of the instruction back to the register file

Observations (Part 1)

Fetch → Decode → Execute → Memory → Write Back

- We now have the situation that the “memory part” and the “write back” only needs to be done for those operations that need it
  - An add instruction does not need a memory access
  - A store instruction does not need a write back to register file
- We can also increase the clock frequency because we have smaller operations in each cycle.
- BUT overall, this is not going to give us a significant speed up

Observations (Part 2)

Fetch → Decode → Execute → Memory → Write Back

- At a given moment of time, most of the hardware is idle
  - When we do a fetch, all the hardware for decode, execute, memory, write back is not doing anything productive (its waiting for the next input)

- Goal / Idea:
  - We want more concurrency (if all circuit parts “work” (not idle), more work is done per clock cycle)
  - Concretely
    - While we decode the current instruction, we could already fetch the next instruction
    - While we execute the current instruction, we could decode the next instruction and we could fetch the instruction after the next
    - …

Pipelining

Pipelining – The Basic Idea

- Pipelining is something that is not only known in computers

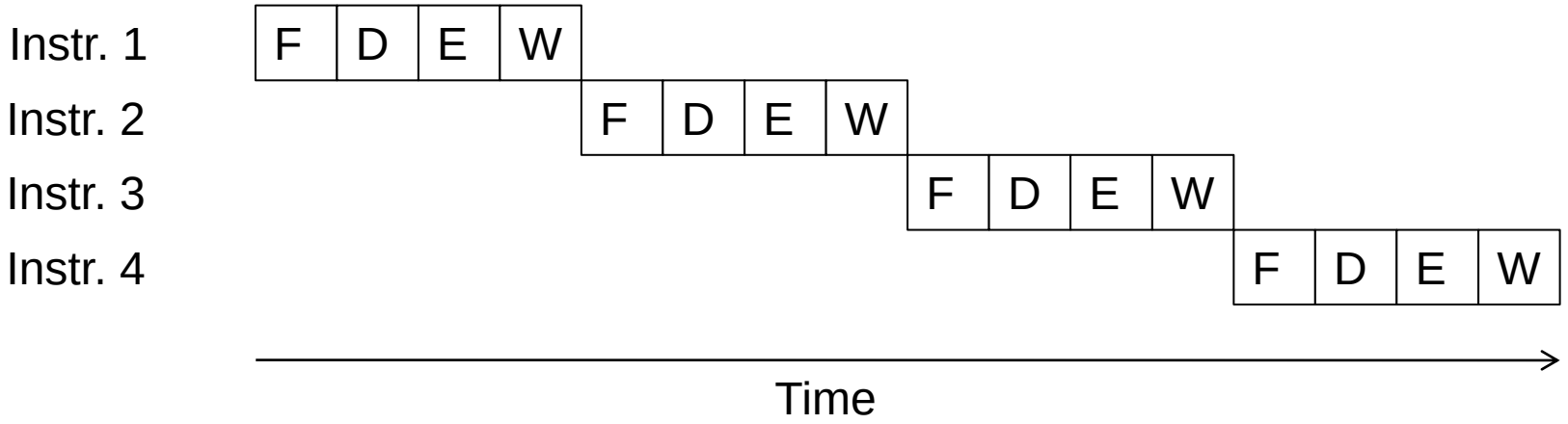
Assembly lines in car industry

Food preparation pipeline in the kitchen

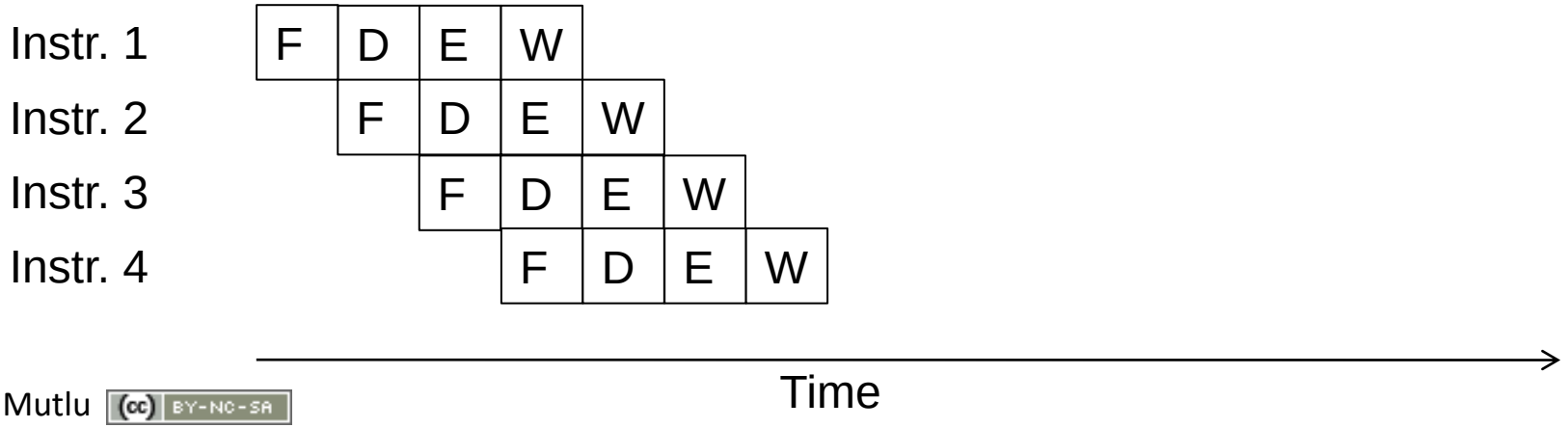
Pipelining – The Basic Idea

- Idea:
  - Divide instruction processing into different stages
  - Don’t complete the execution of one instruction before starting the execution of the next instruction
  - Process a different instruction in each stage (e.g. Stage 3 processes instruction i, Stage 2 processes instruction i-1, …) → the consecutive instructions are executed in consecutive stages
- Benefit:
  - “We use all hardware resources in each clock cycle” → Increased instruction throughput

Illustration for four simple instructions without memory access



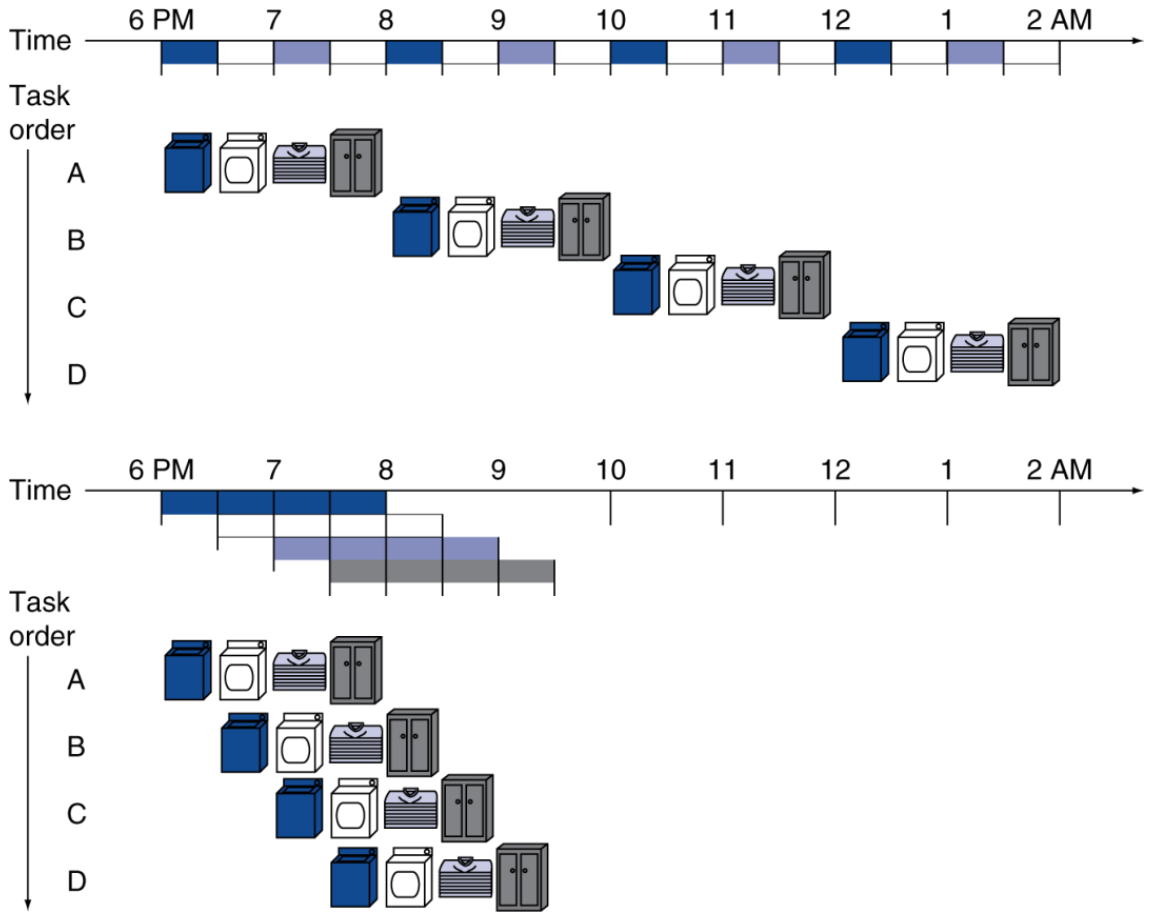
Execution time without pipelining:
16 cycles



Execution time with pipelining:
7 cycles
(1 instruction per cycle in steady state)

Figure by Onur Mutlu

The Laundry Analogy



- Speedup:
  - 7 hours instead of 16 hours in case of four loads
  - One load of washing every hour in case of non-stop washing
- Observe:
  - The processing of the stages is sequentially dependent (we can't change sequence)
  - Each stage uses different resources (no resource dependency between stages)
  - The processing of the washing loads is independent of each other (we can do any sequence of the 4 washing loads)

Properties of an Ideal Pipeline

We are given the task to perform set of operations

Ideal setup for pipelining:

- **The operations are identical:** We need to repeat the same operations over and over again (e.g. wash 10.000 loads of cloths)
- **The operations are independent:** We can perform the operations in any sequence we want
- **Uniform partitioning into suboperations is possible:** Each operation is can be divided into suboperations that take the same amount of time

Ideal Pipelining for Processors

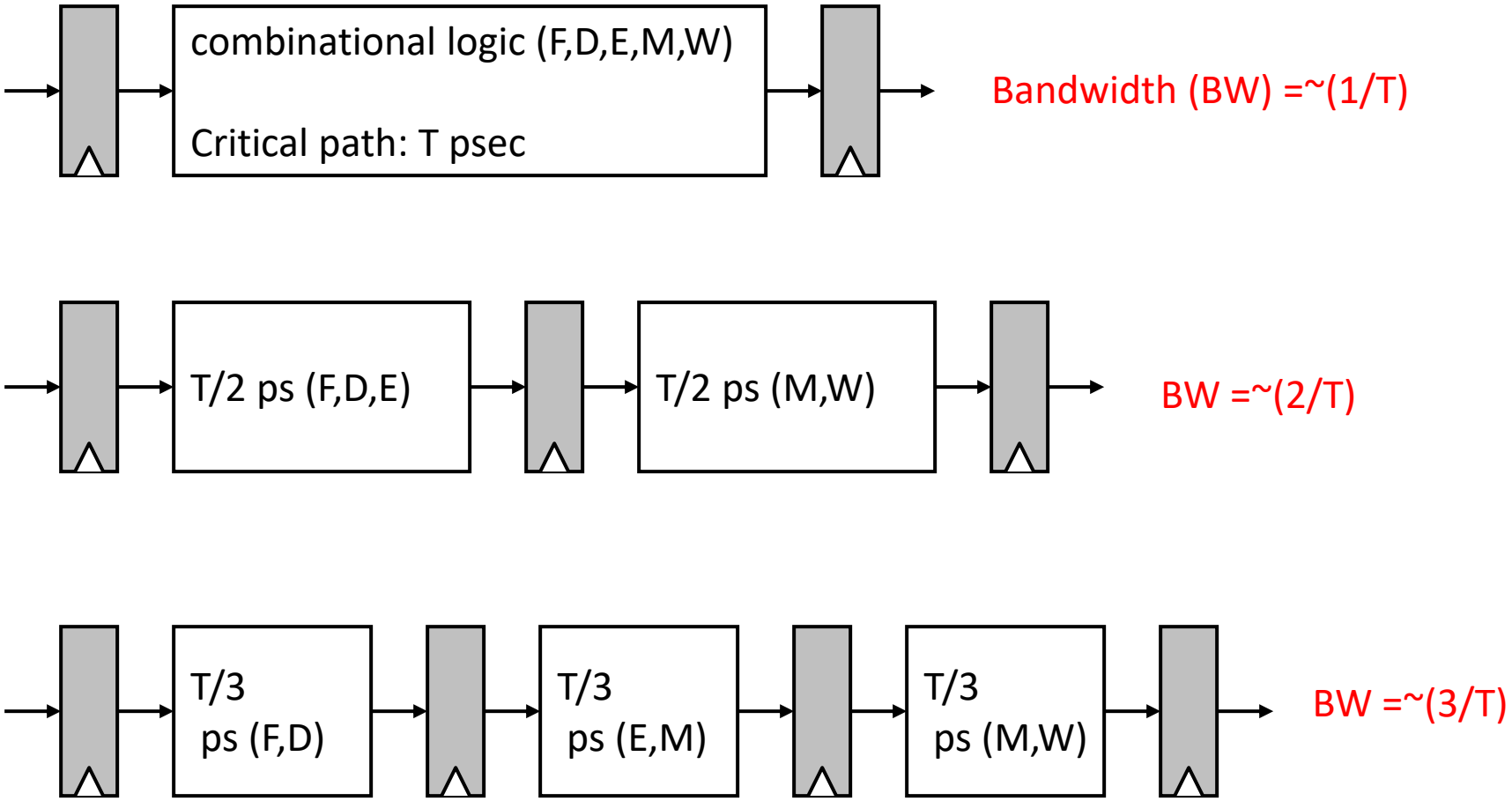


Figure by Onur Mutlu

More Realistic Pipeline: Throughput

- Non-pipelined 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)

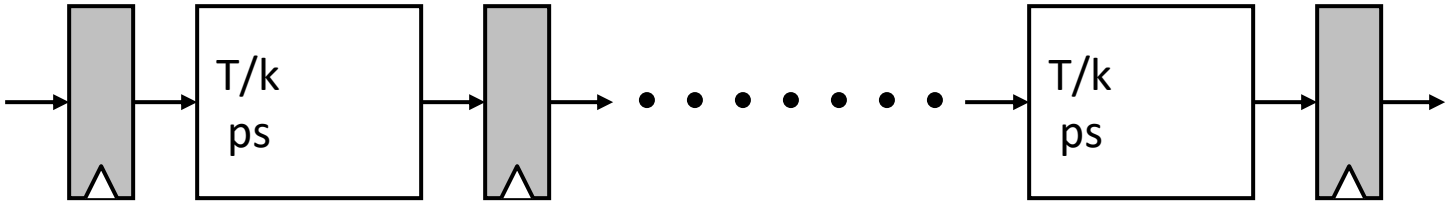


Figure by Onur Mutlu

More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost G

  Cost = G + L where L = register cost

- k-stage pipelined version

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

Registers increase hardware cost

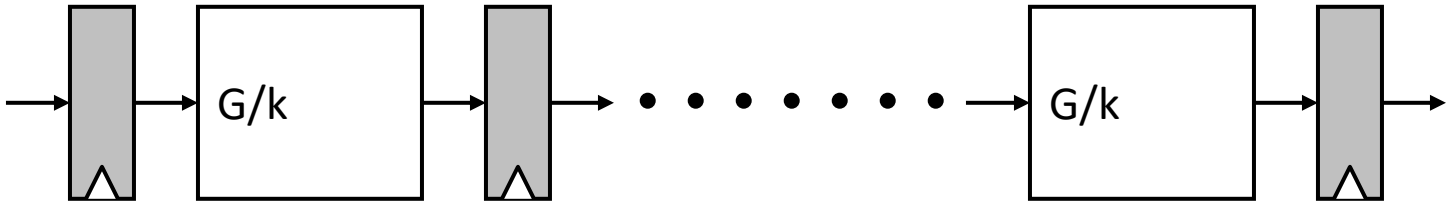
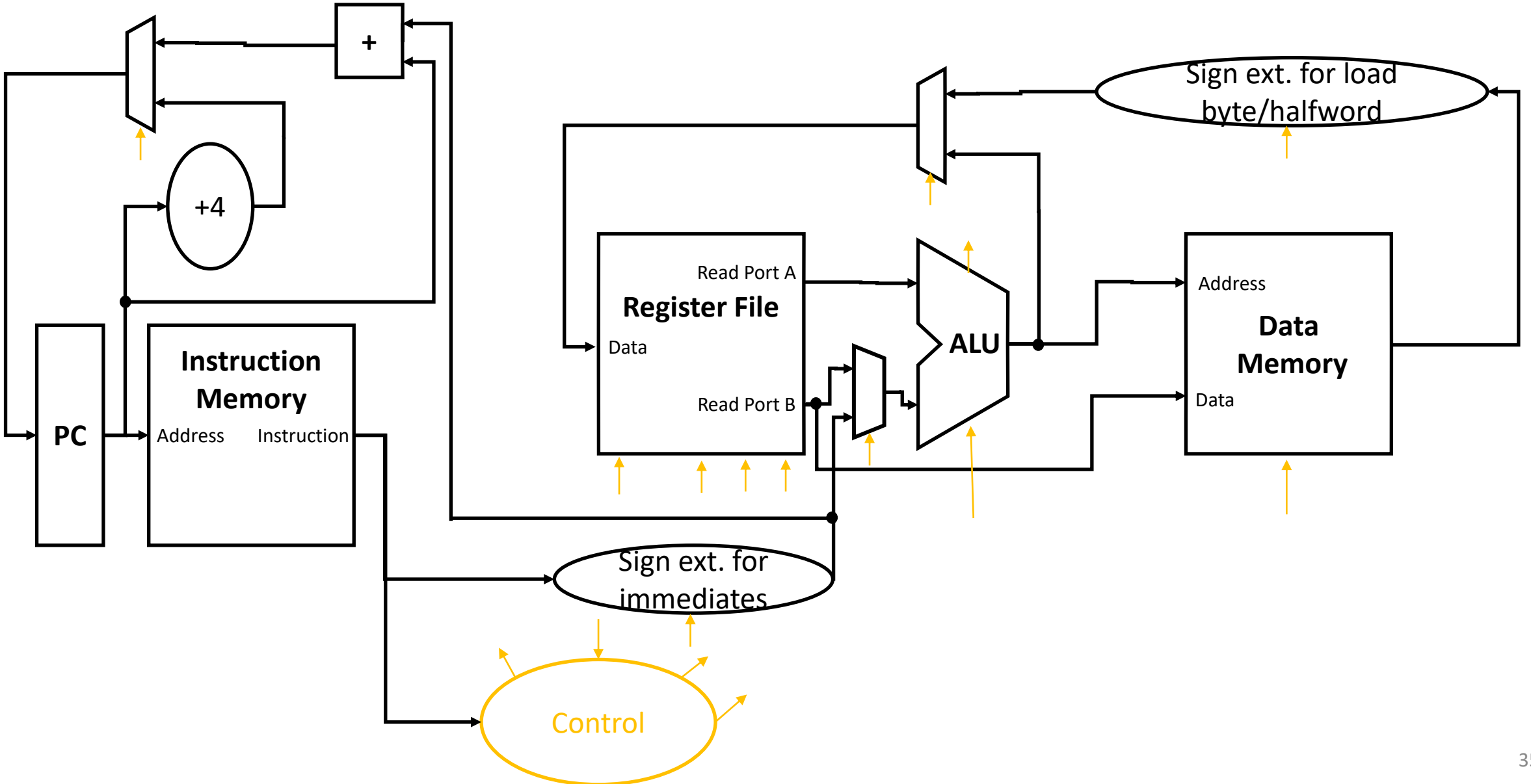


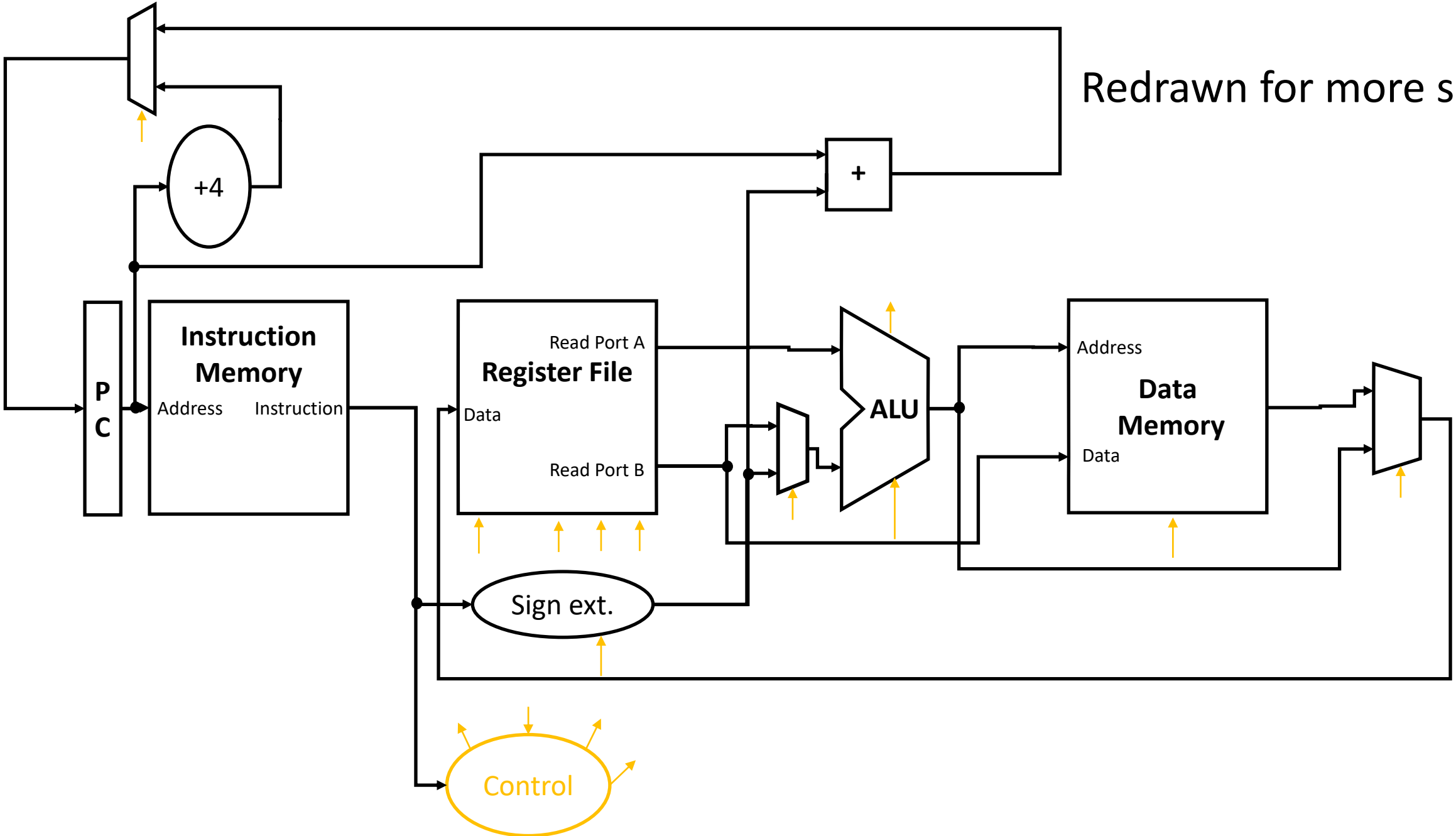
Figure by Onur Mutlu

Pipelining Instruction Processing

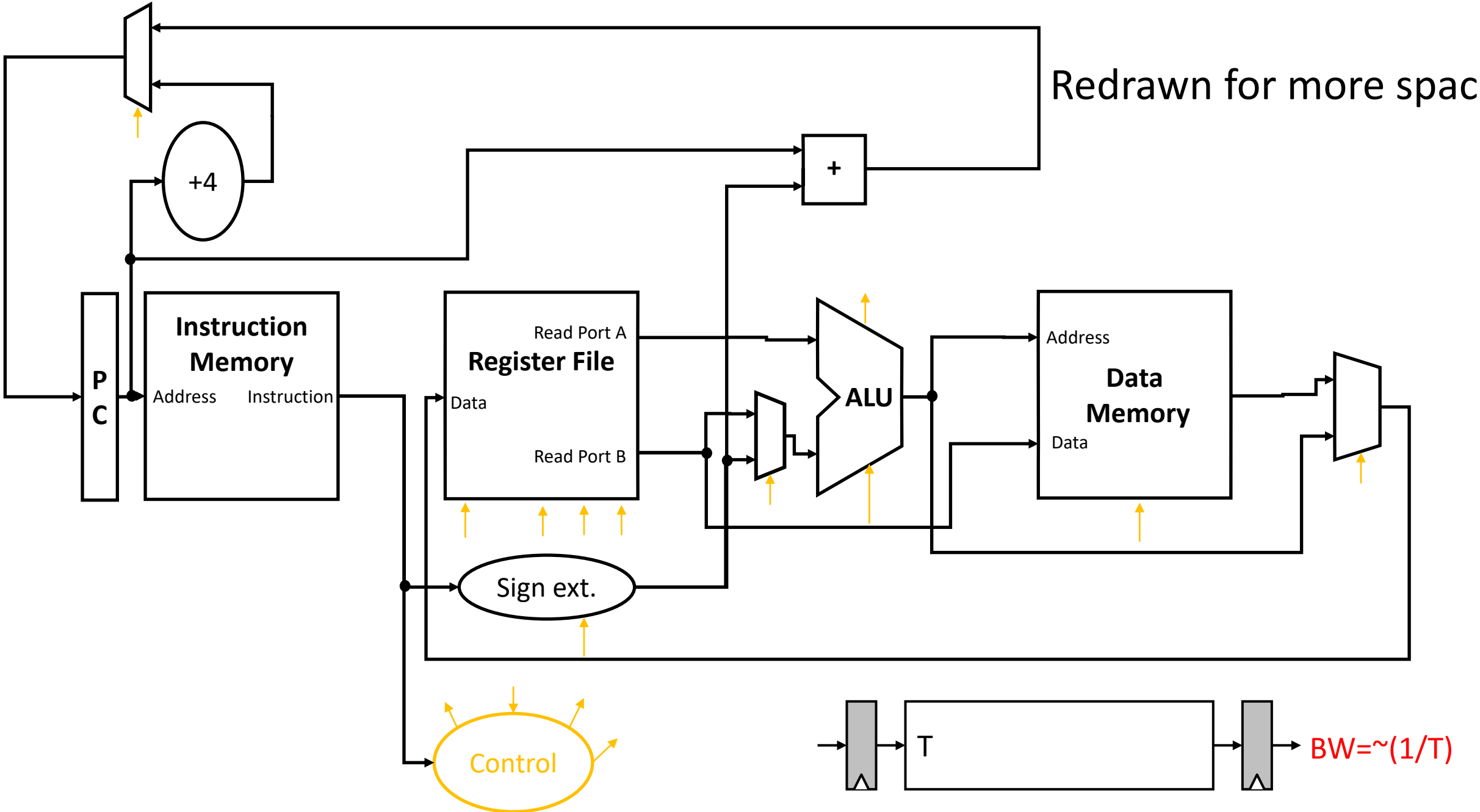
High-Level Datapath

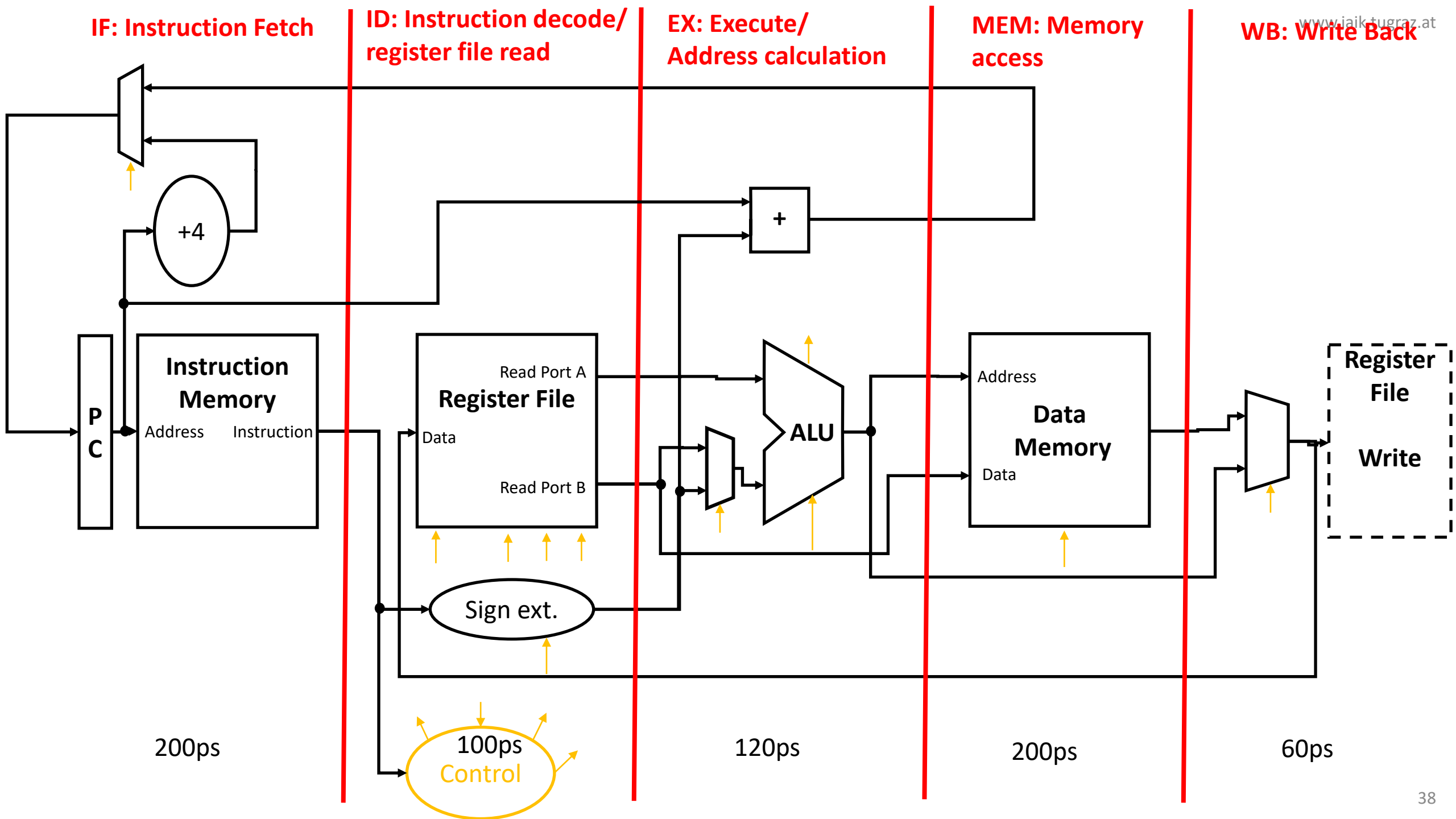


Redrawn for more space



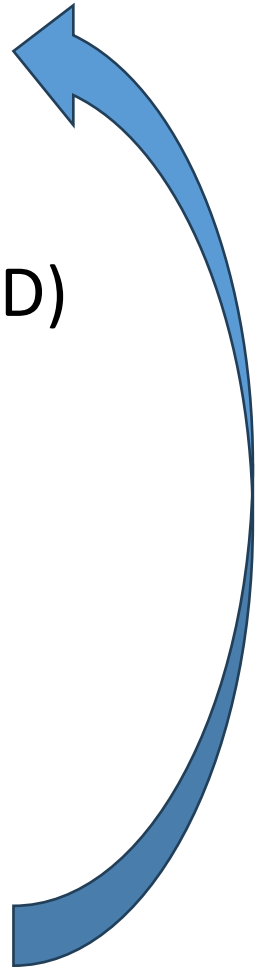
Redrawn for more space



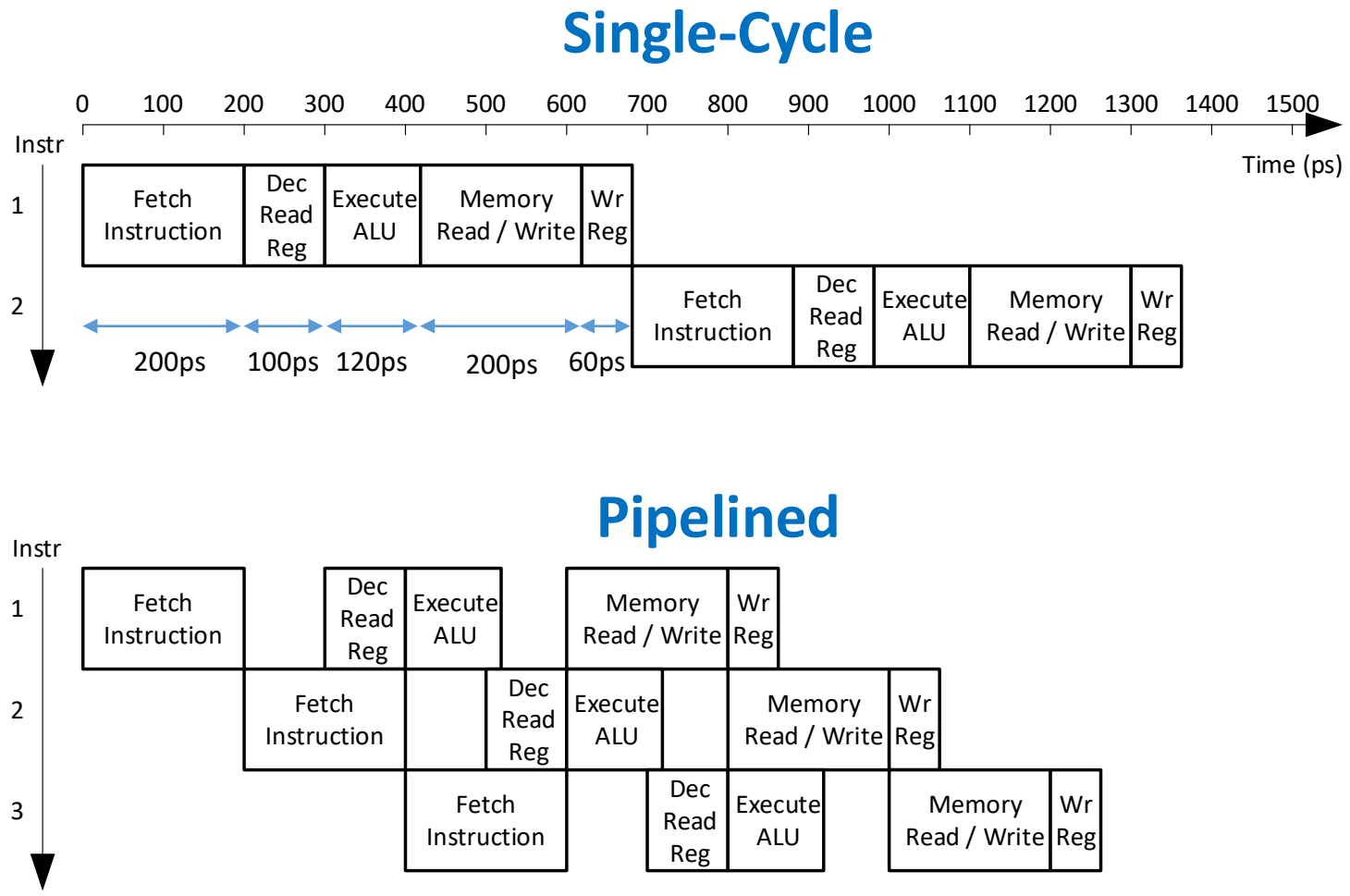

IF: Instruction Fetch
ID: Instruction decode/ register file read
EX: Execute/ Address calculation
MEM: Memory access
WB: Write Back
www.iaik.tugraz.at
+4
+
P C
Instruction Memory
Address
Instruction
Read Port A
Register File
Data
Read Port B
ALU
Address
Data Memory
Data
Register File
Write
Sign ext.
100ps
Control
200ps
120ps
200ps
60ps

The Instruction Processing

1. Instruction Fetch (IF)
2. Instruction Decode & register read (ID)
3. Execute or calculate address (EX)
4. Memory access (MEM)
5. Writeback of result (WB)

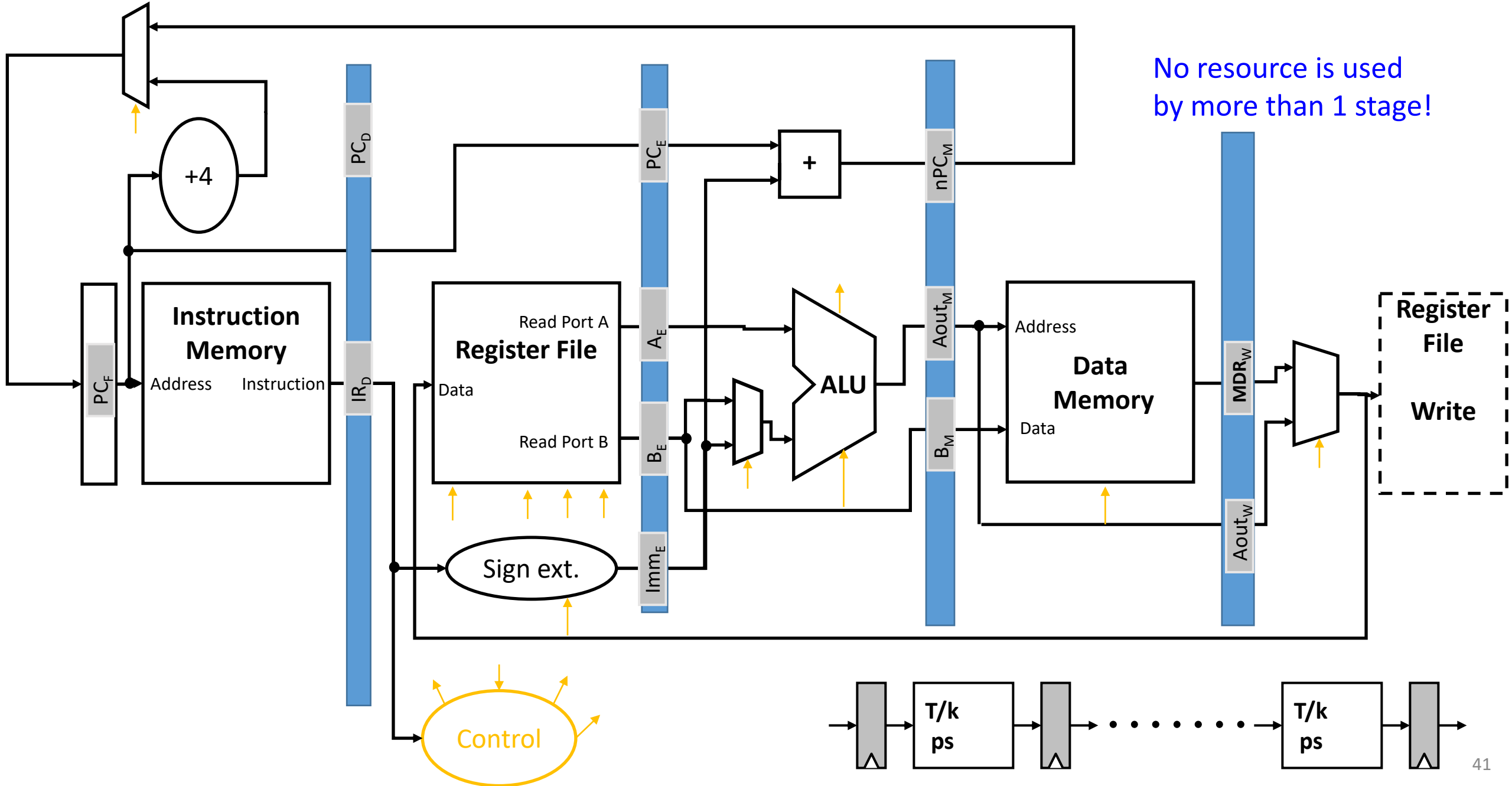


Instruction Pipeline Throughput (non-ideal example)

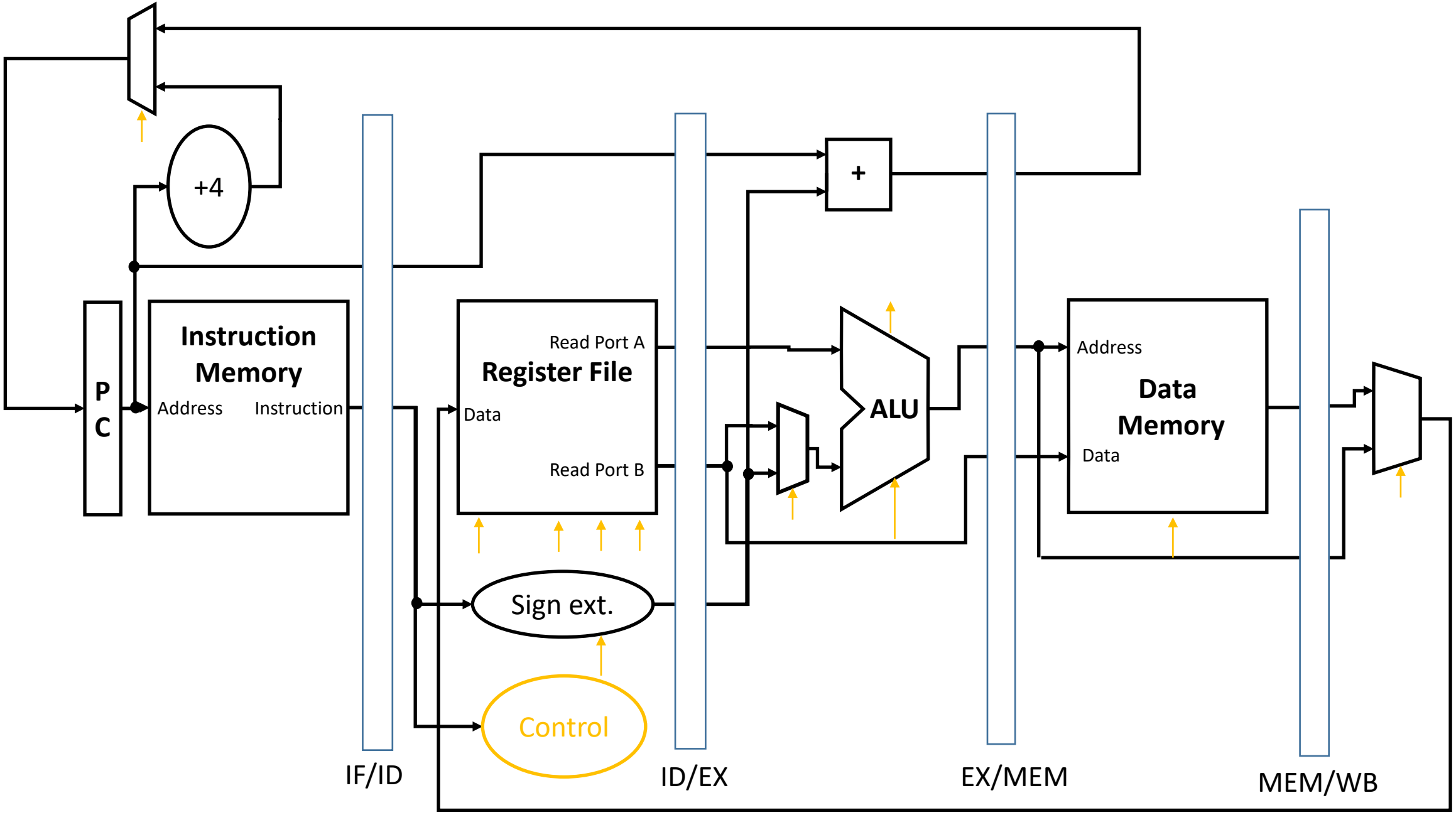


- 680ps without pipelining
- With a 5-stage pipeline, one instruction every 200ps
- Speedup is 3,4 instead of 5 in the ideal setup

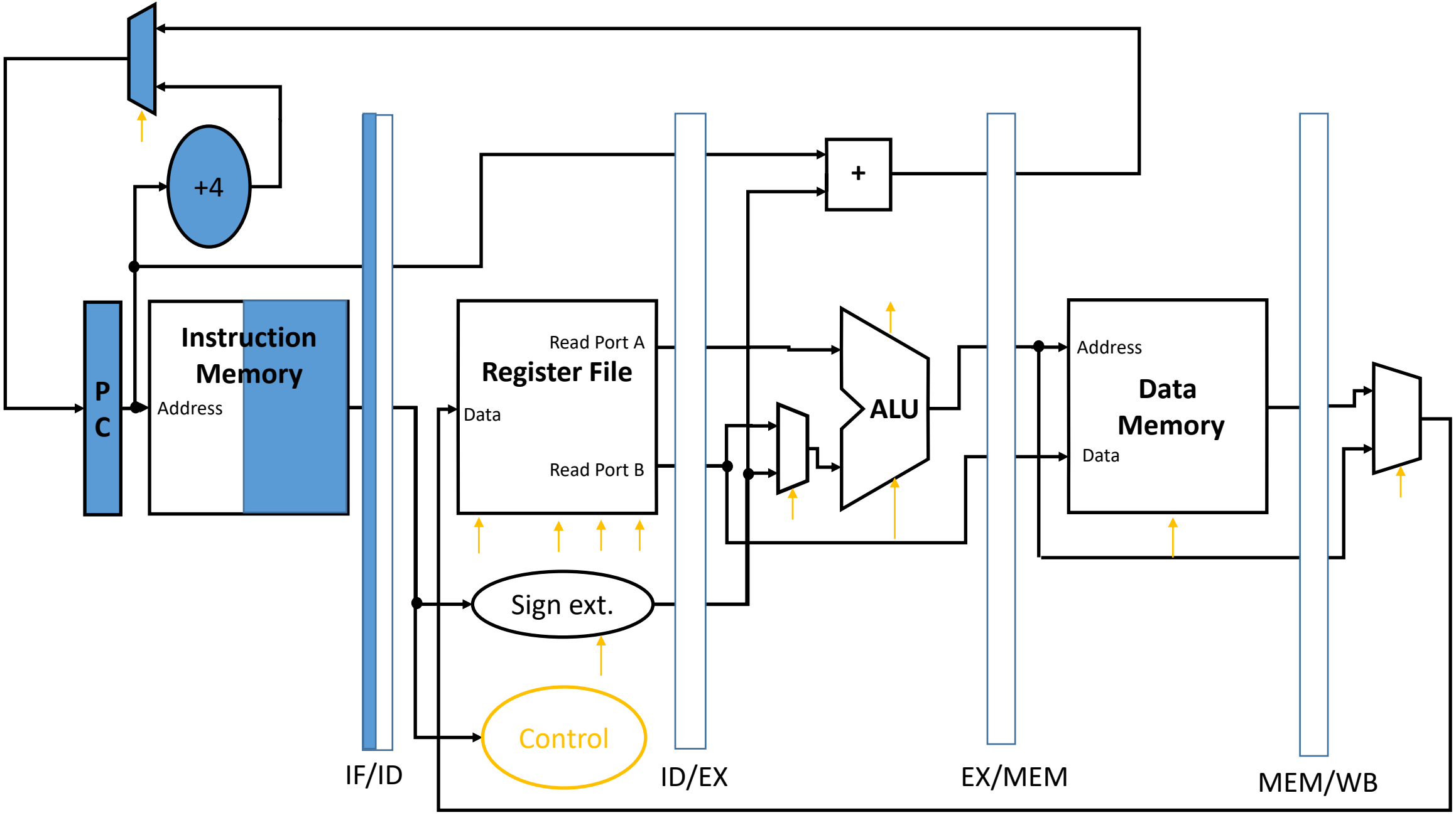
Enabling Pipelined Processing: Pipeline Registers



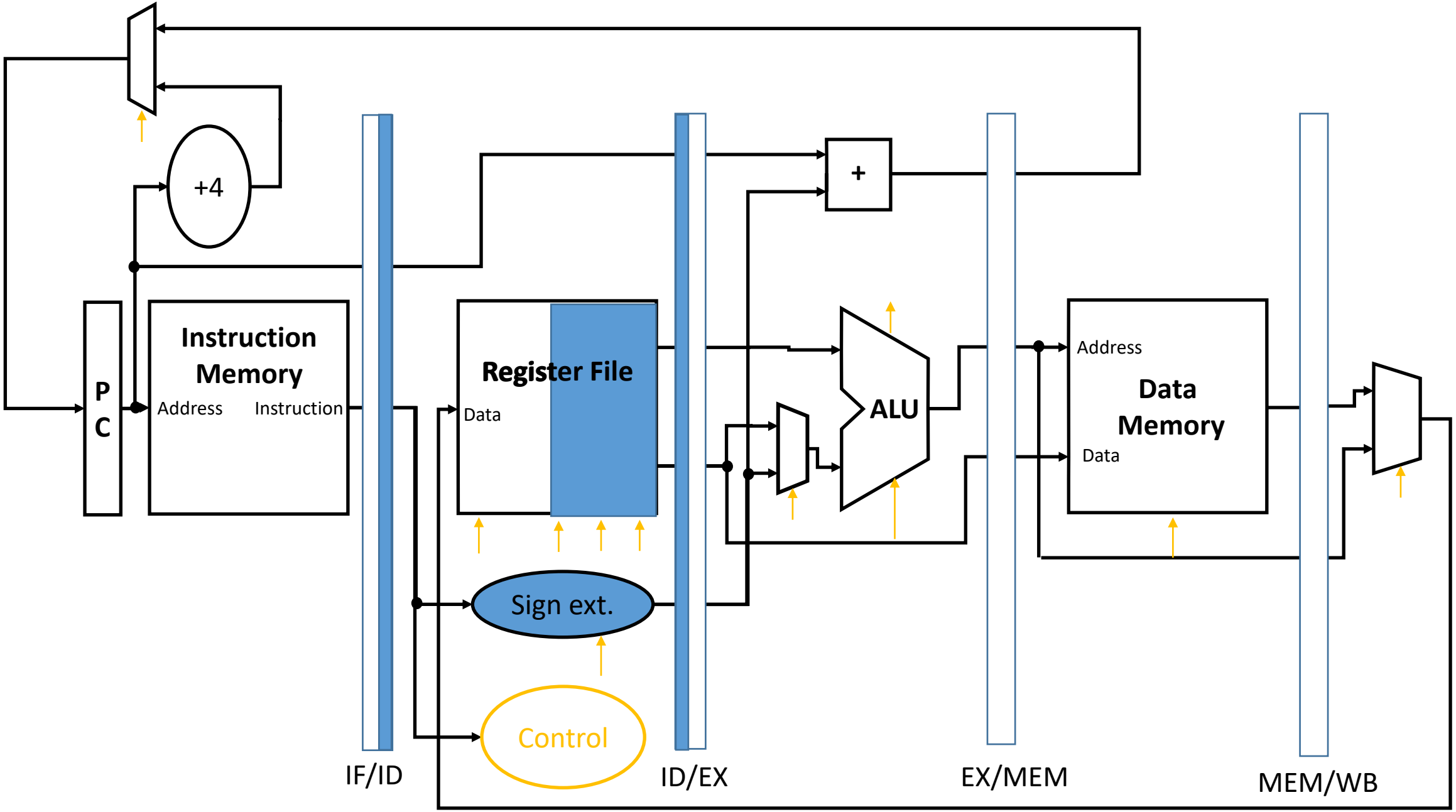
Pipelined Operation



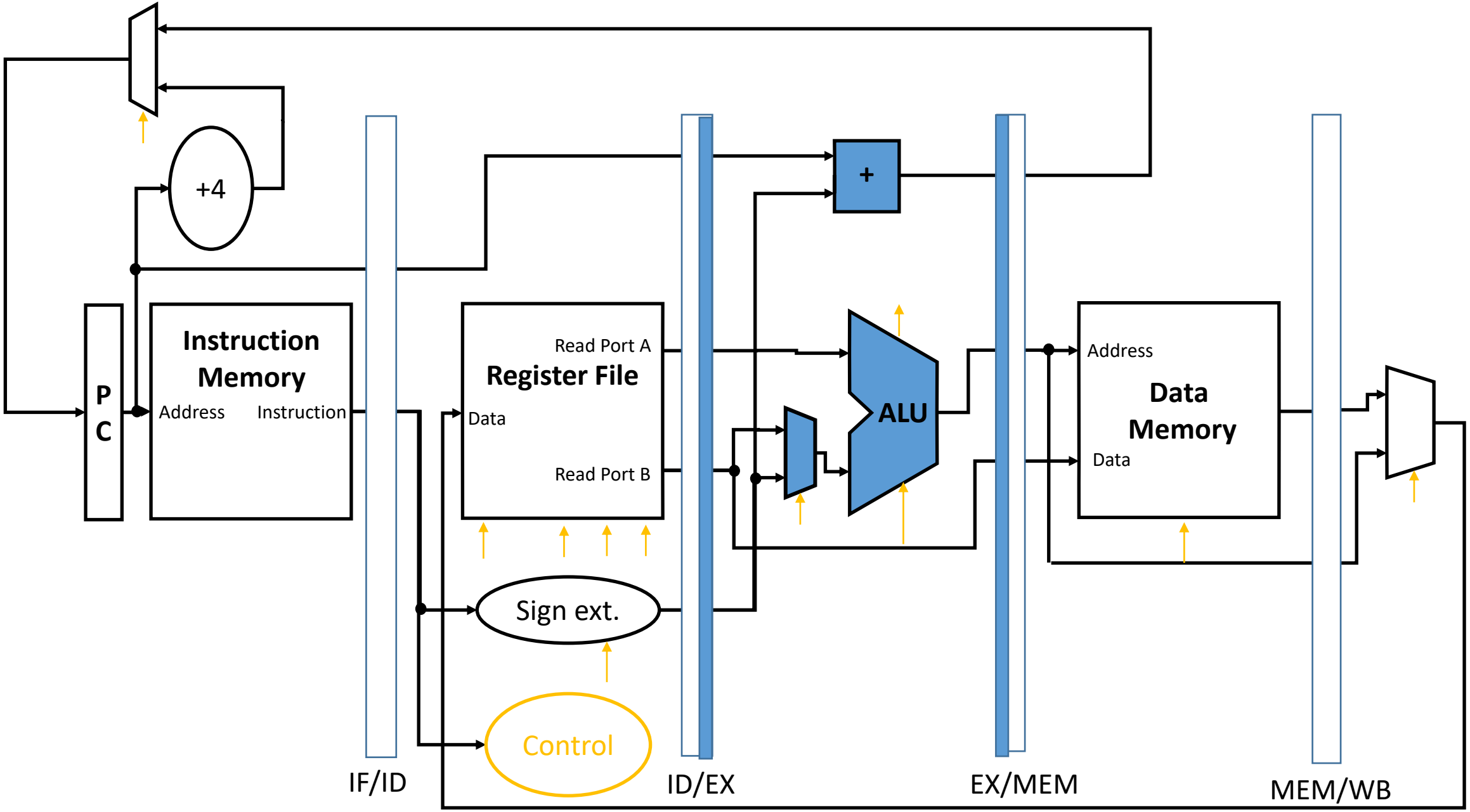
LW
Instruction Fetch

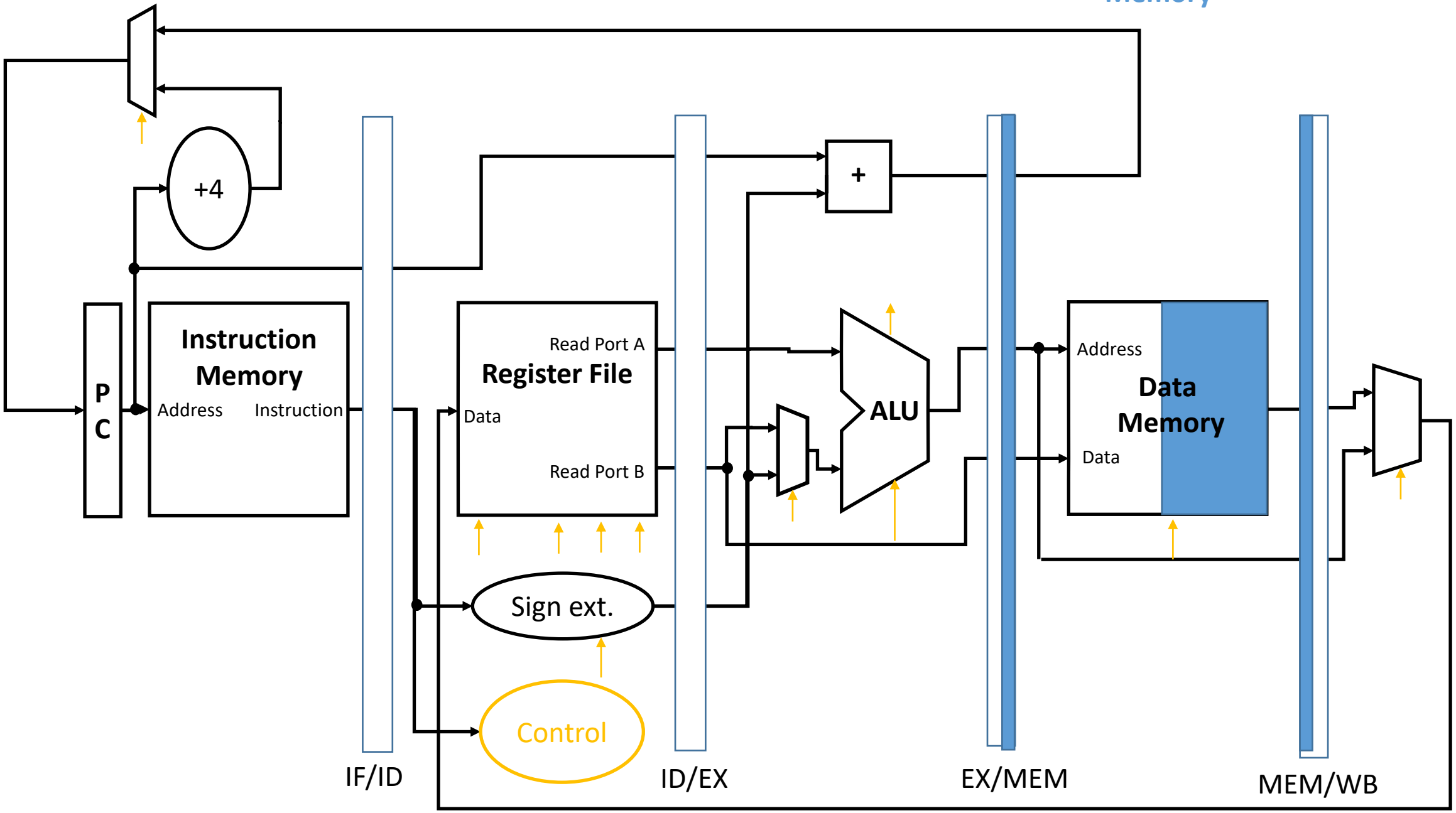


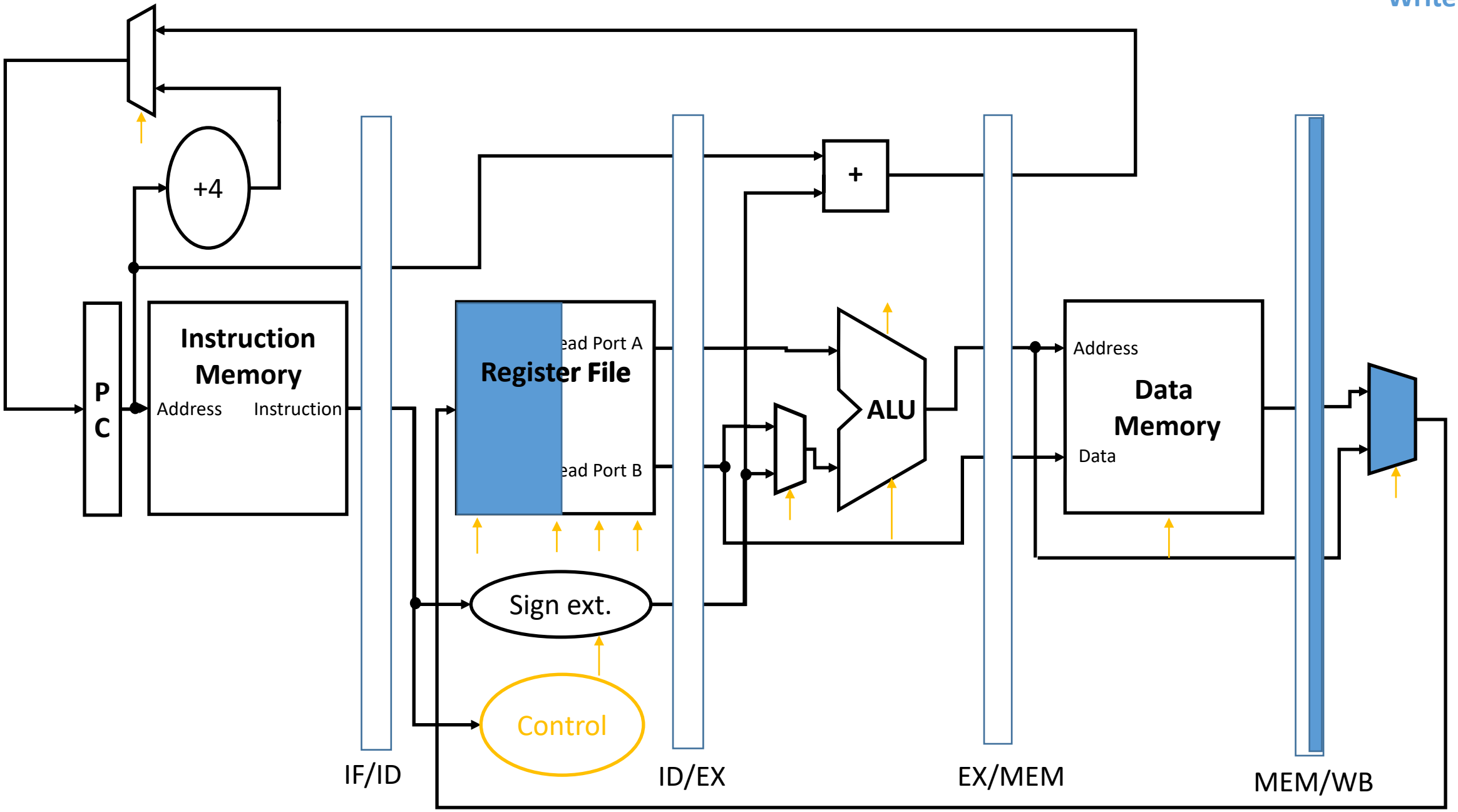
LW
Instruction Decode



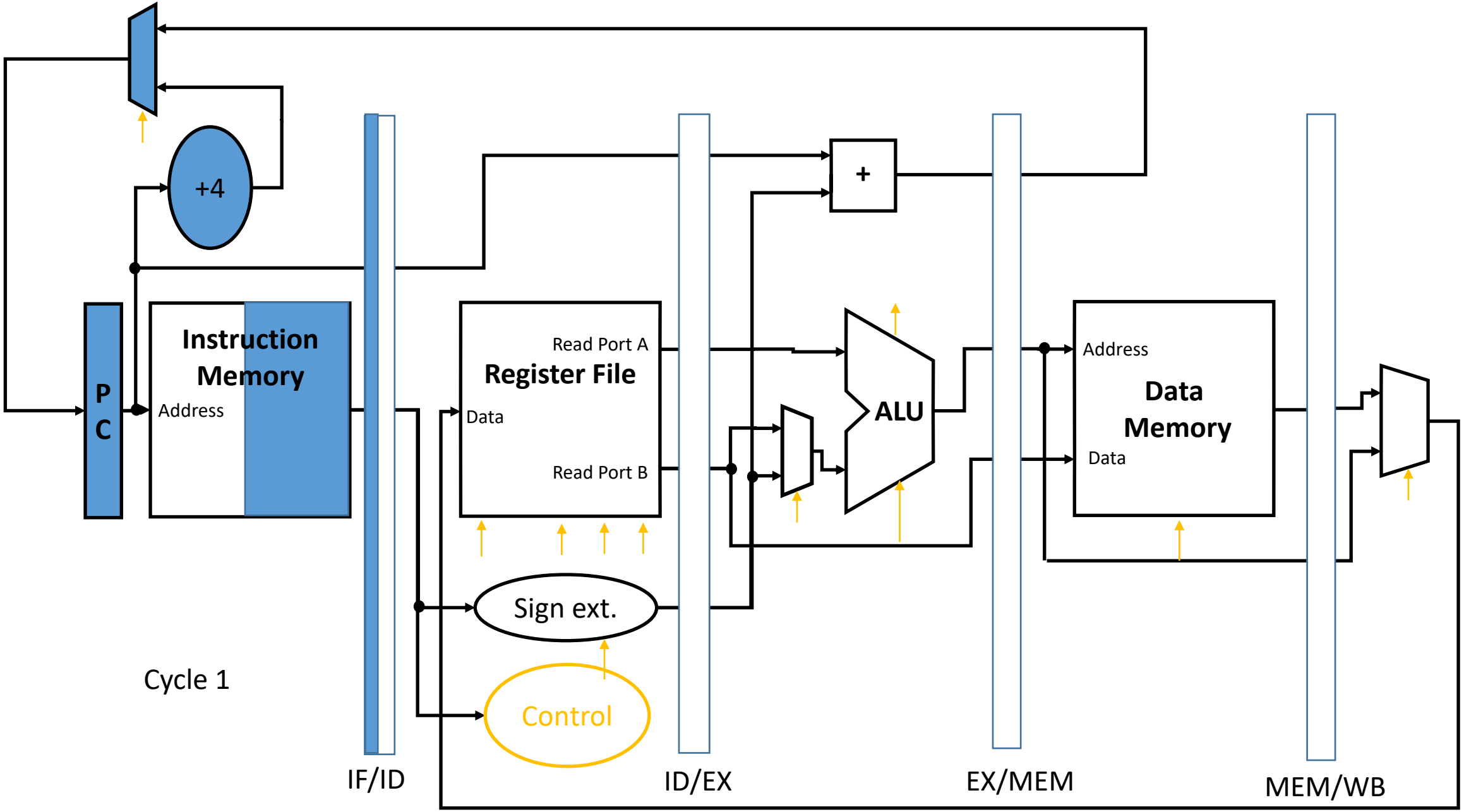
LW
Execute


+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


+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

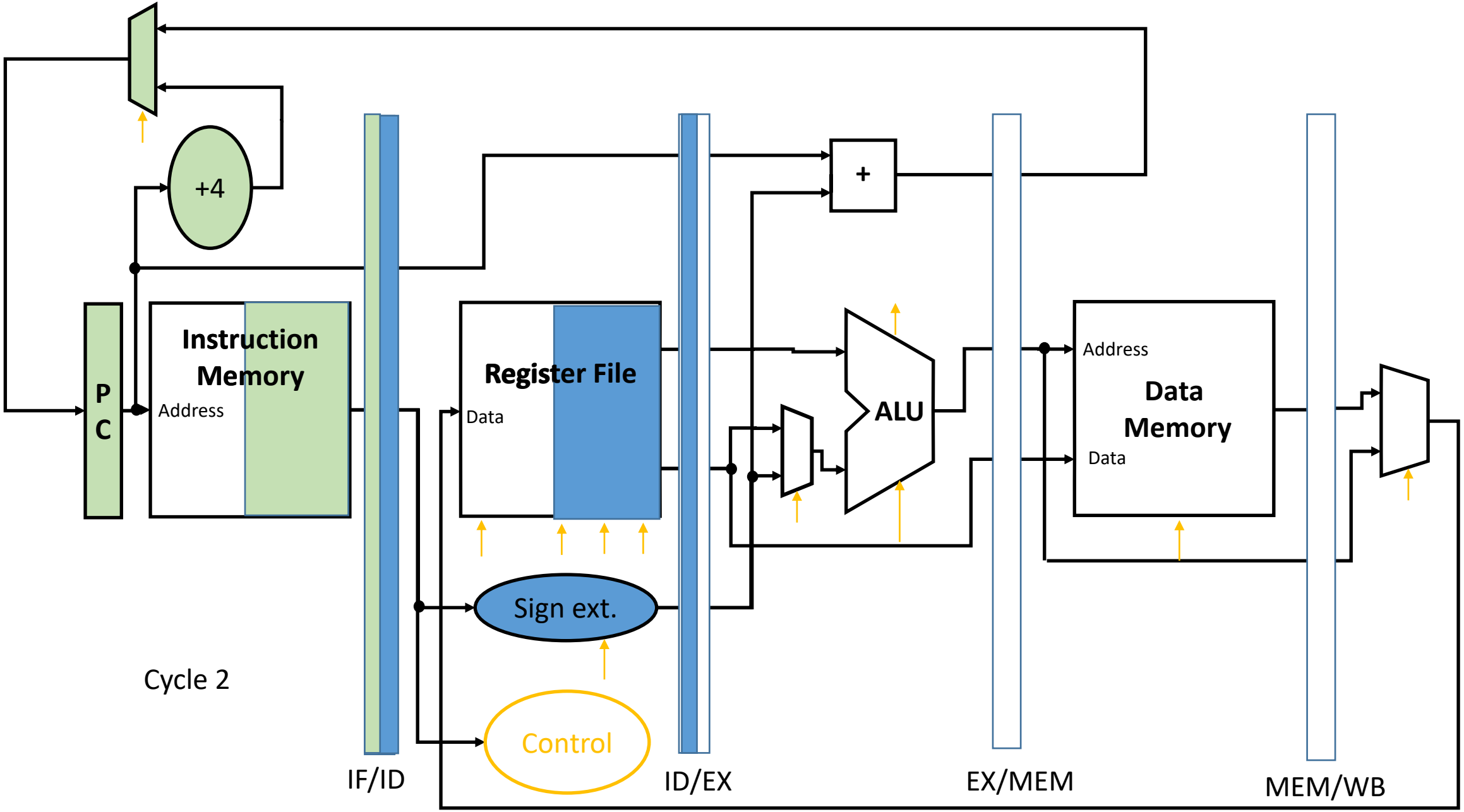

+4
PC
Instruction Memory
Address
Instruction
Register File
ead Port A
ead Port B
+
ALU
Address
Data Memory
Data
Sign ext.
Control
IF/ID
ID/EX
EX/MEM
MEM/WB

LW x1,100(x0)
Instruction Fetch



ADD x2, x3, x4
Instruction Fetch

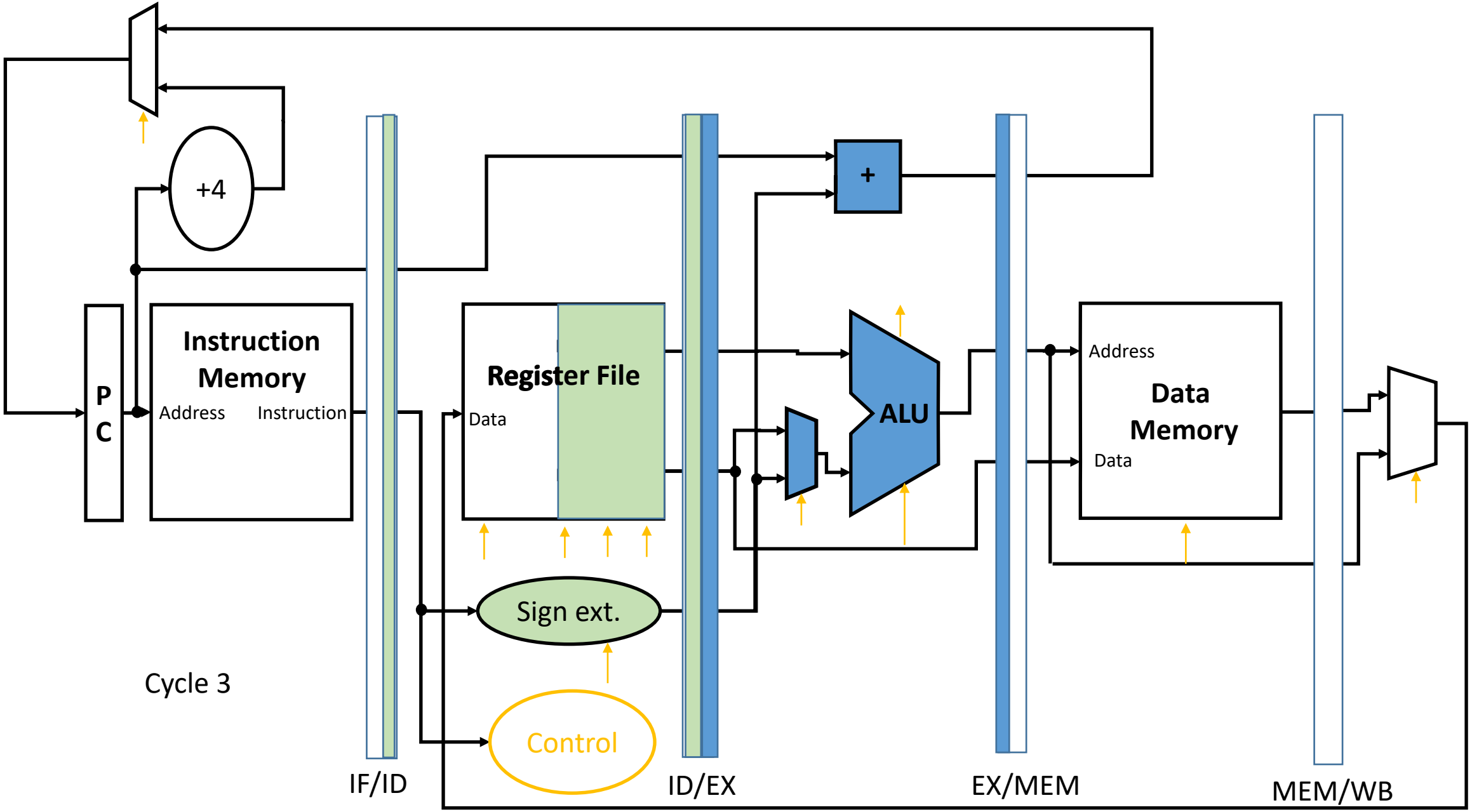
LW x1,100(x0)
Instruction Decode



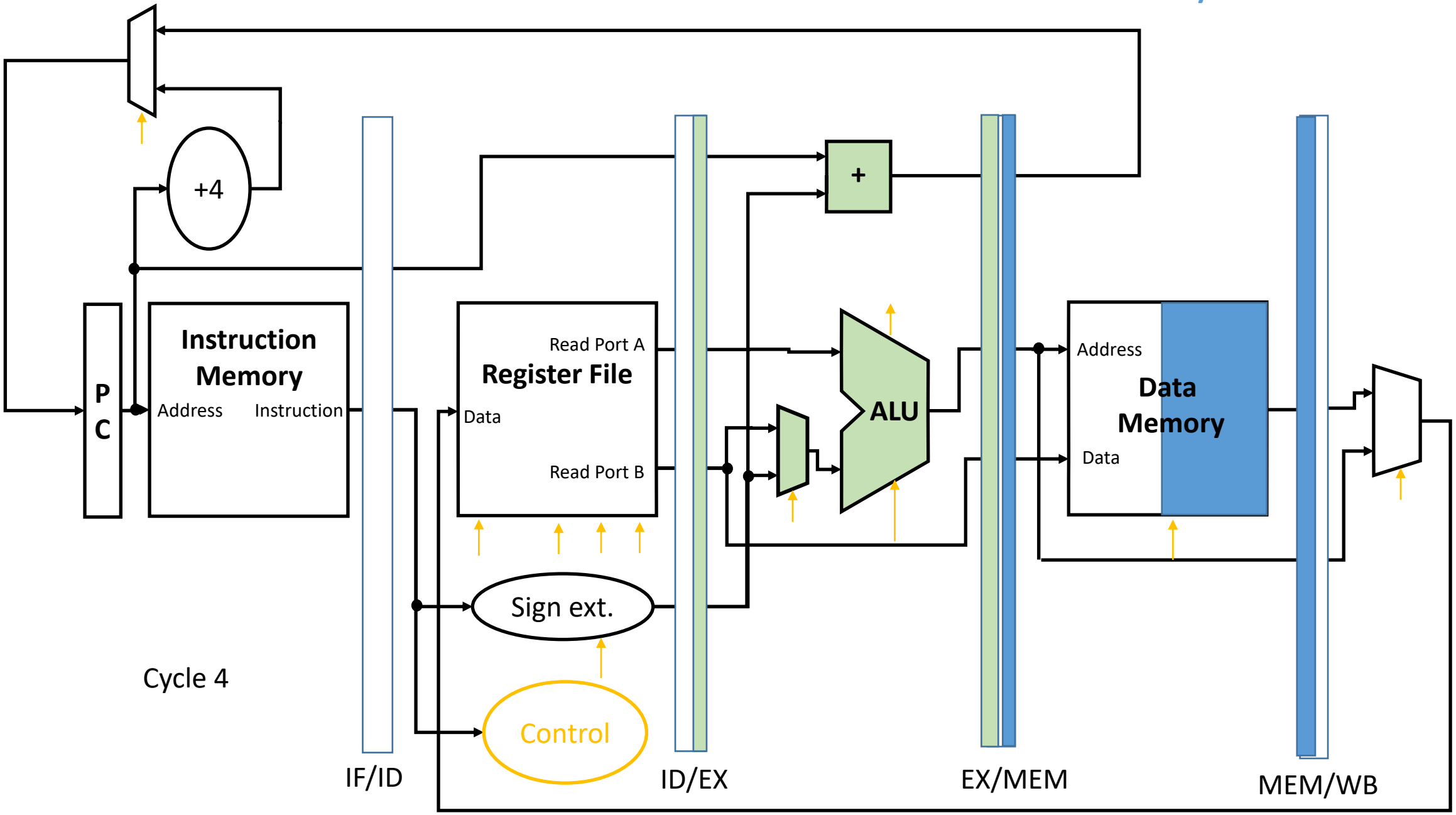
Cycle 2

ADD x2, x3, x4
Instruction Decode

LW x1,100(x0)
Execute



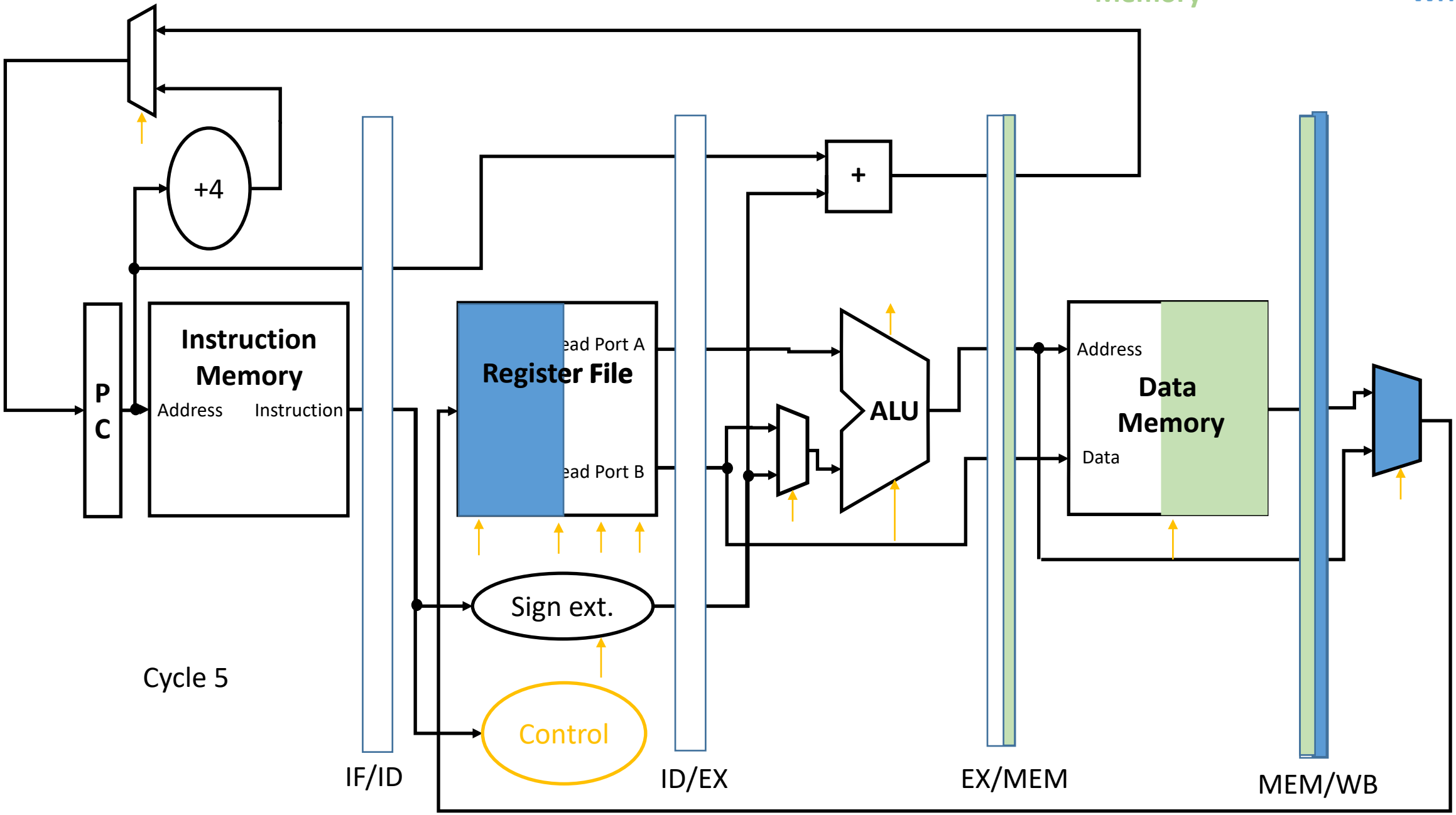
Cycle 3


+4
+
PC
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

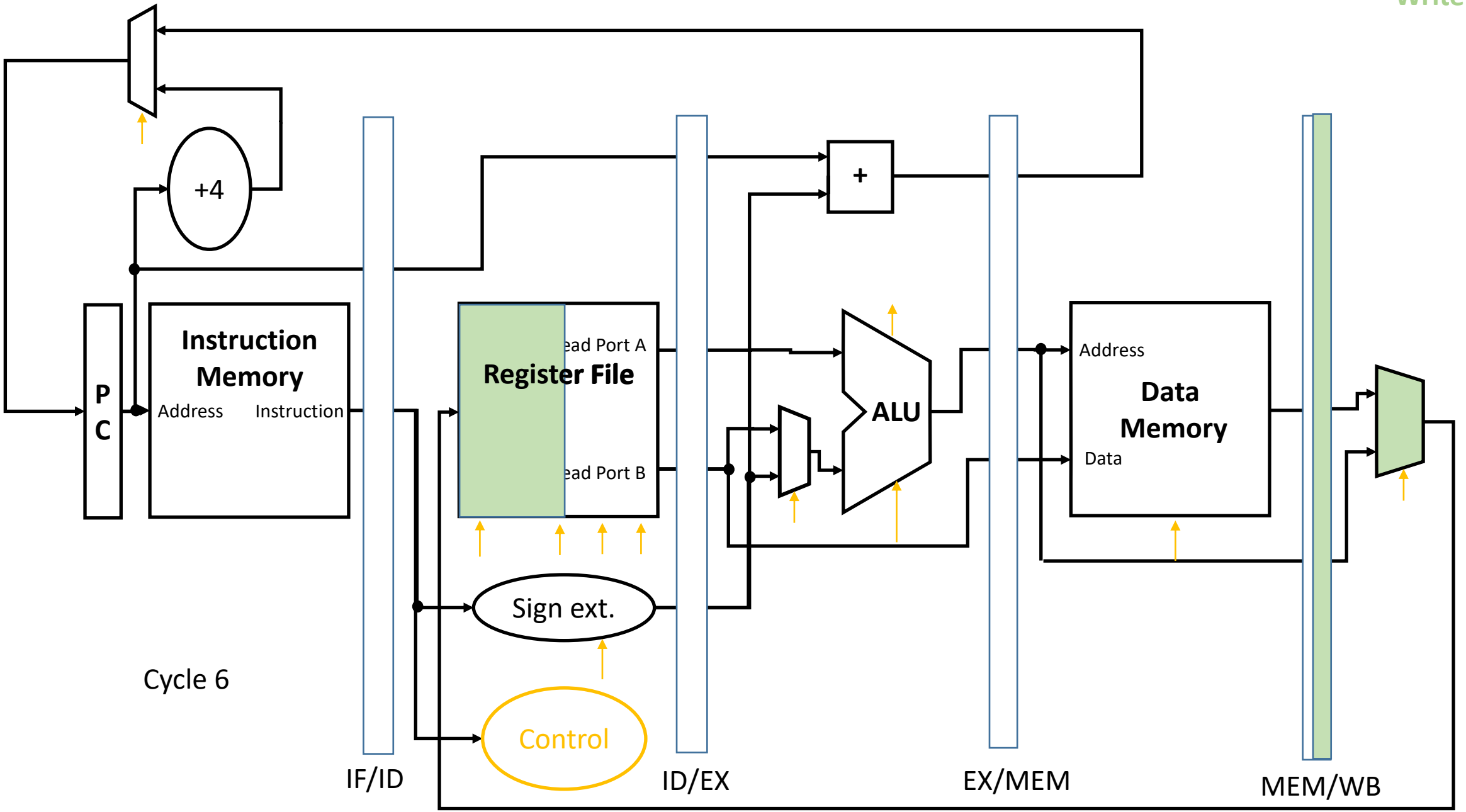
Cycle 4

ADD x2, x3, x4
Memory

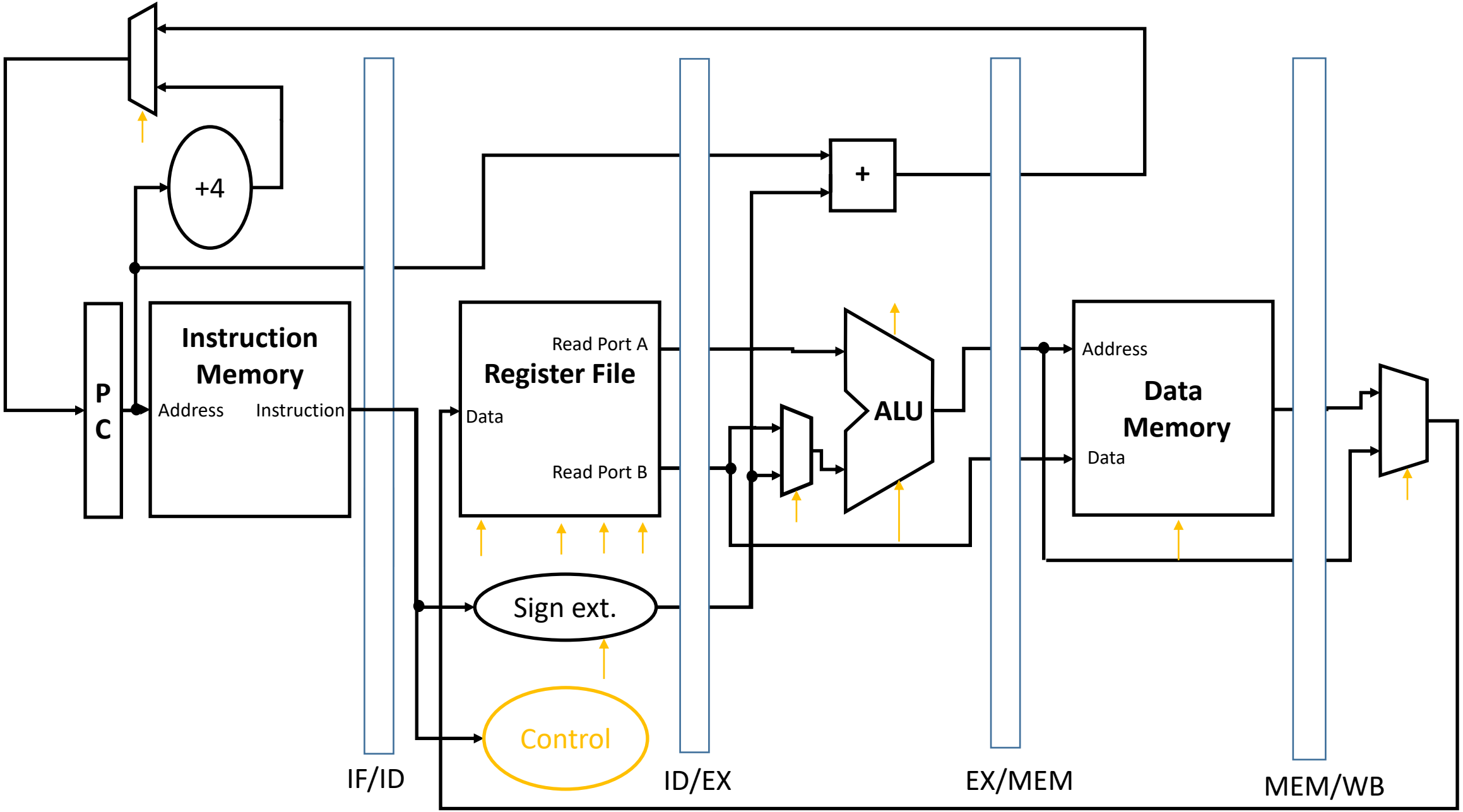
LW x1,100(x0)
Write Back



ADD x2, x3, x4
Write Back



Pipelined Operation



Illustrating Pipeline Operation: Operation View

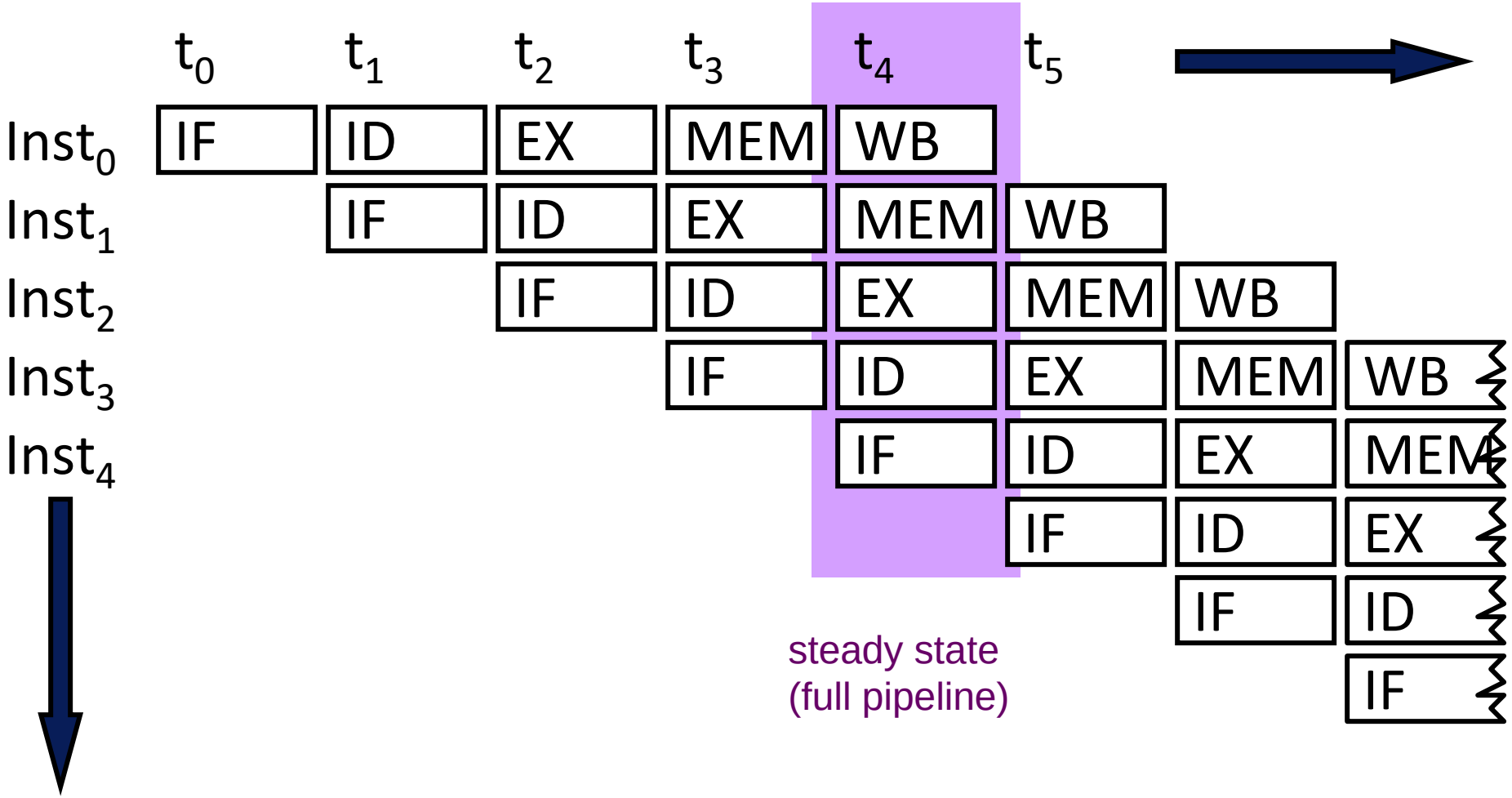


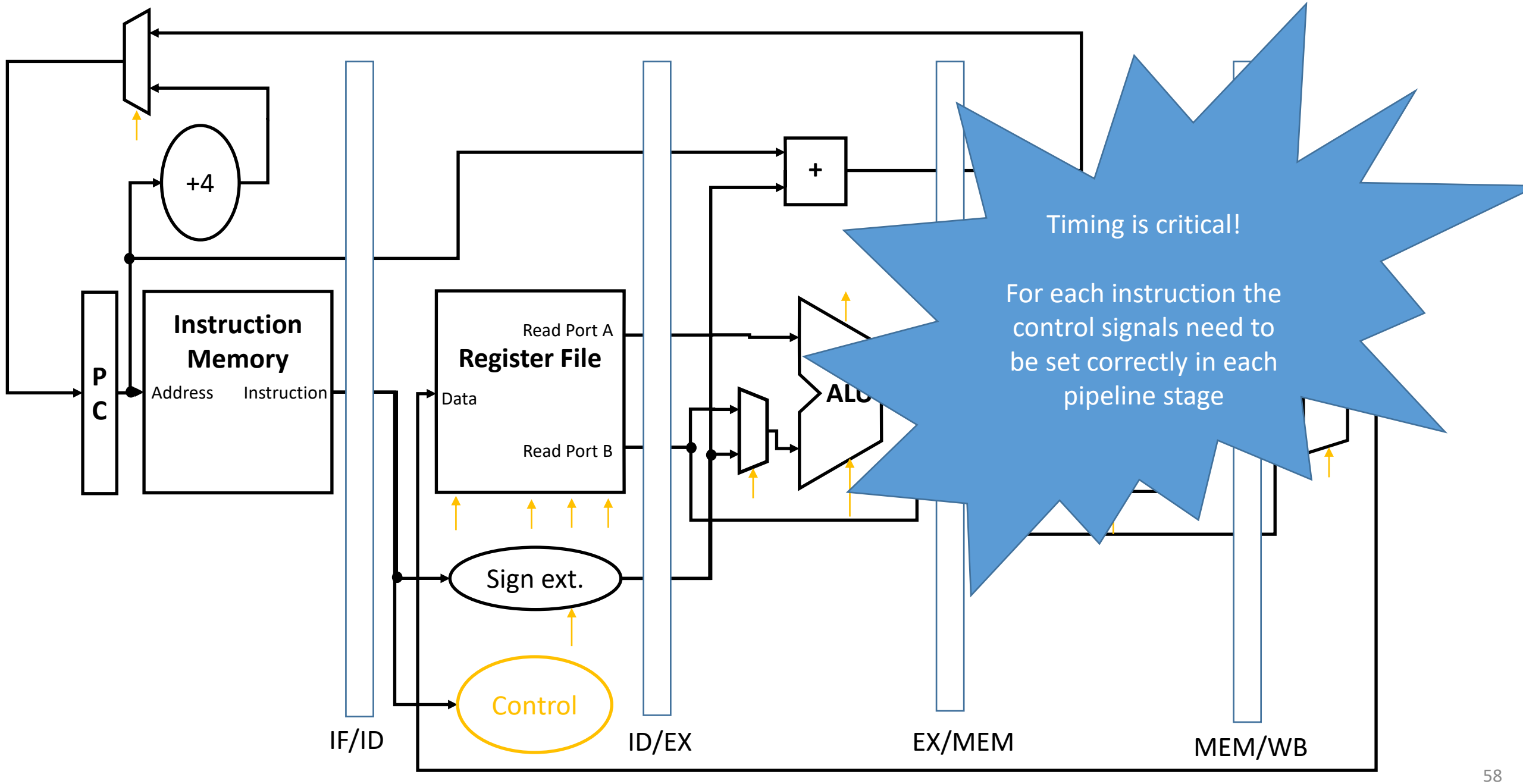
Figure by Onur Mutlu

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

Figure by Onur Mutlu

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

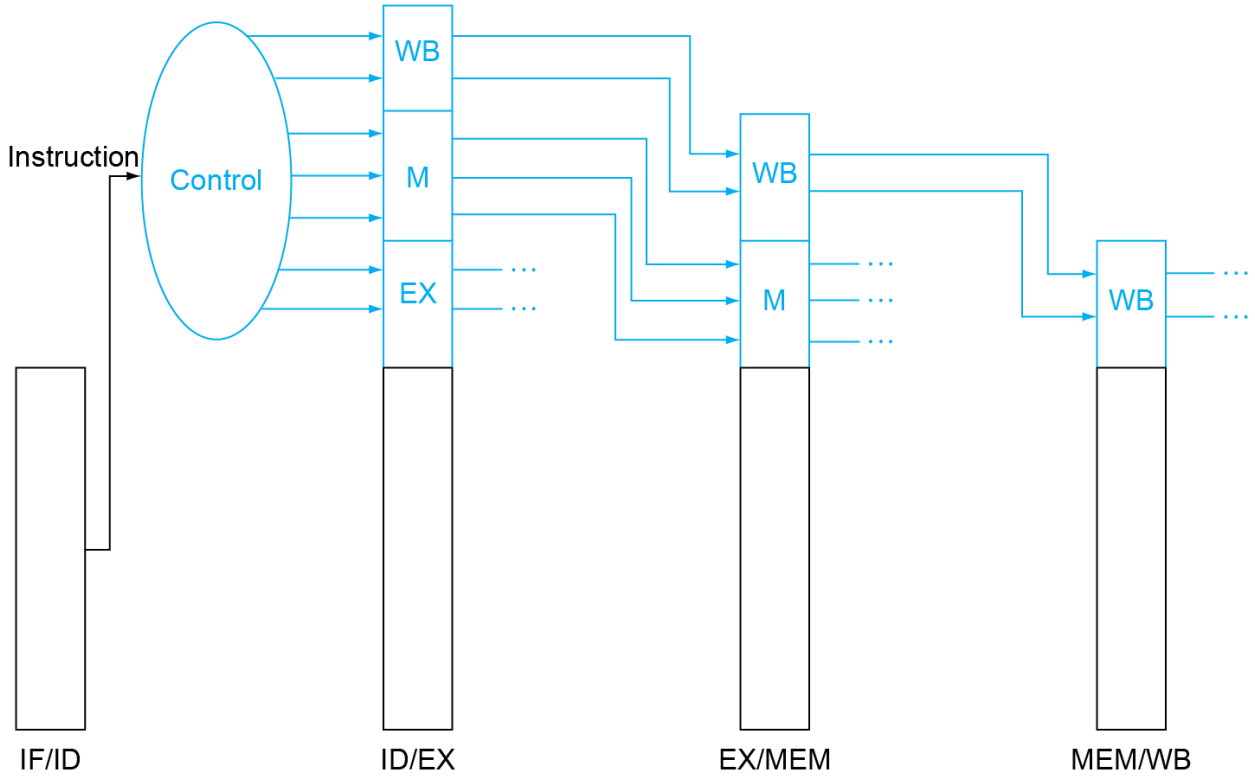


Control Signals in a Pipeline

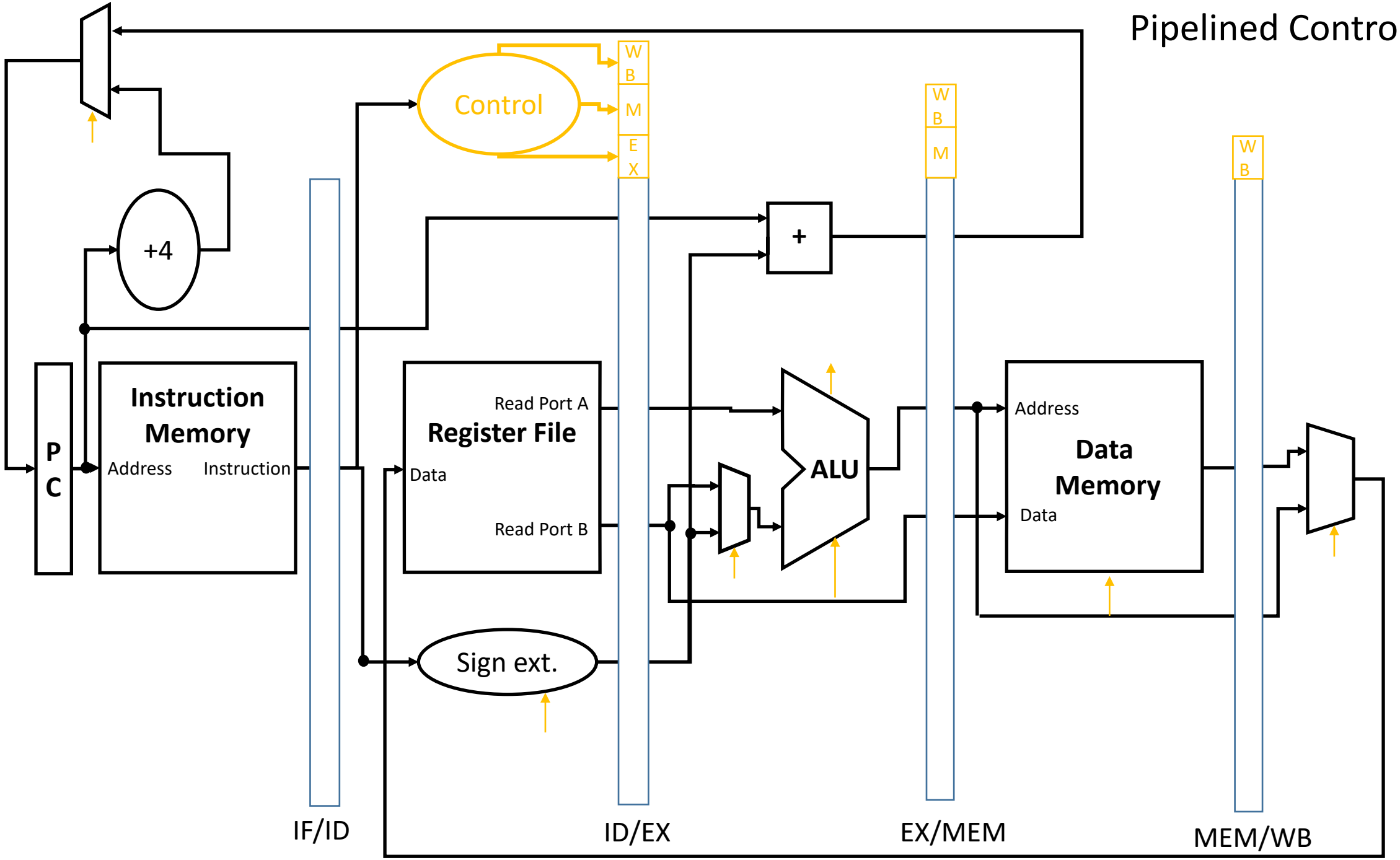
- For every instruction we need to provide the same control signals as in the single-cycle version
  BUT
  they need to provided to the right stage at the right time

- Two options:
  1. Decode the control signals once using the same decoder as in a single-cycle system and buffer the signals (see figure)

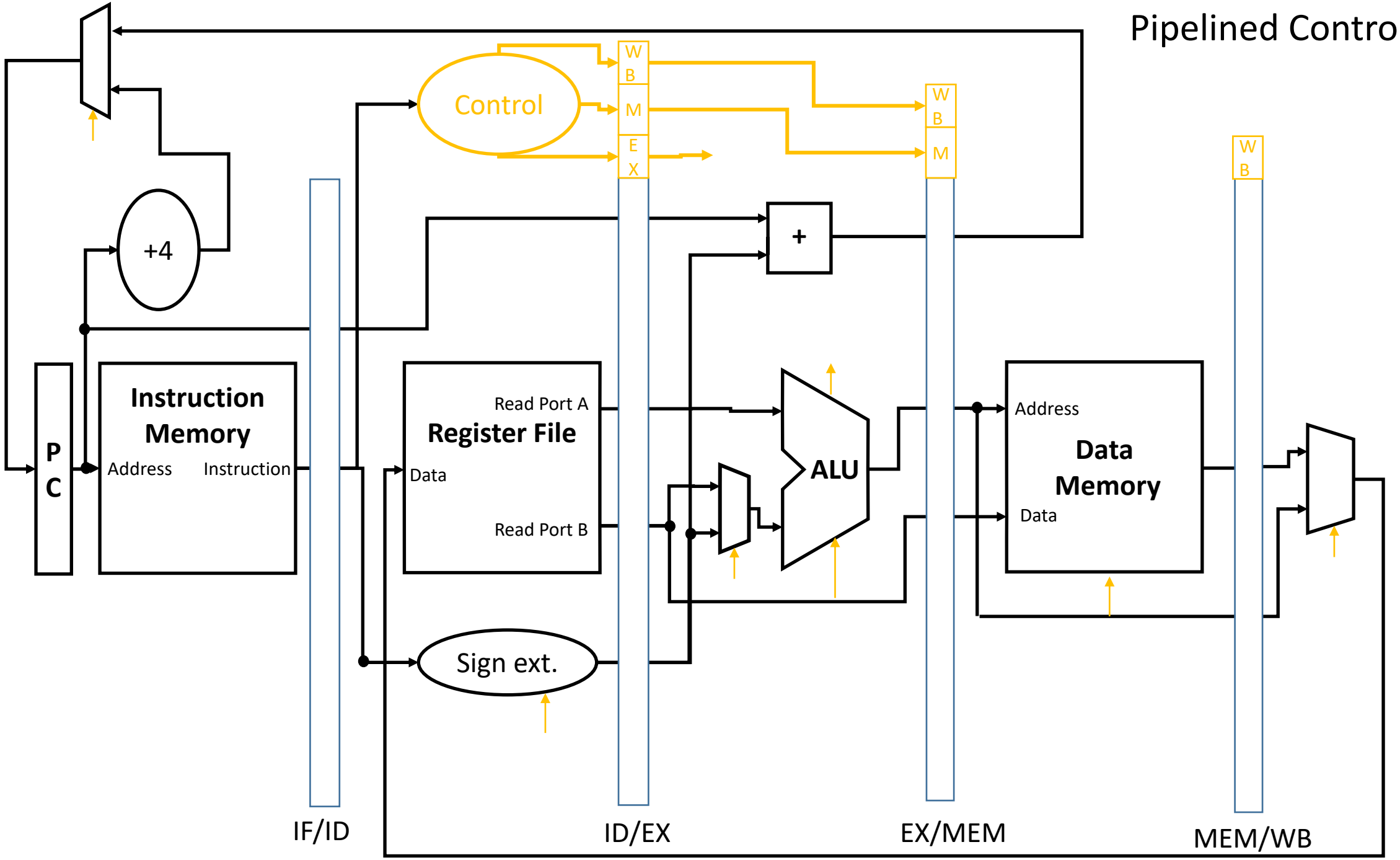
  2. Carry relevant parts of the instruction word through the pipeline and decode locally within the different stages



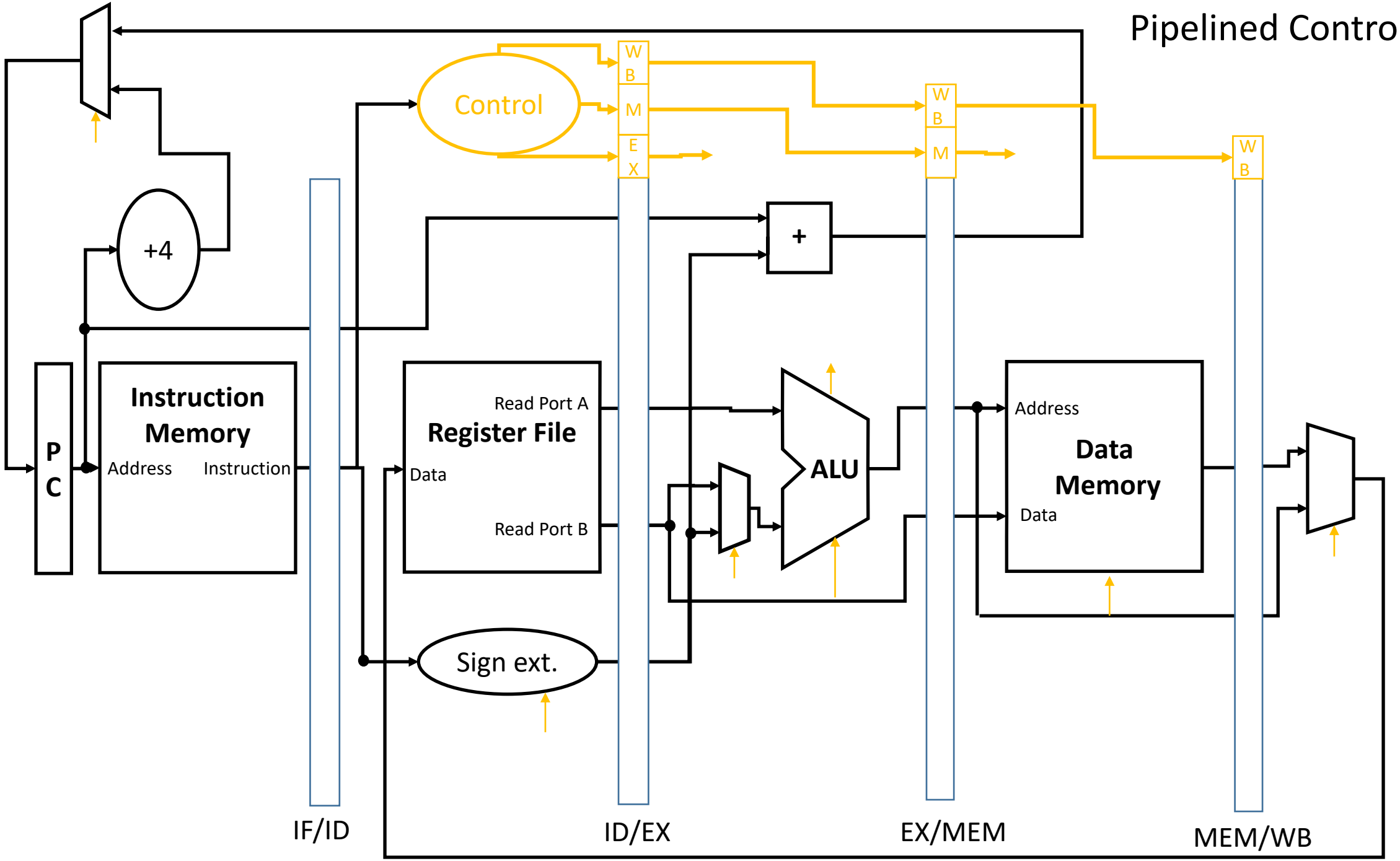
Pipelined Control Signals



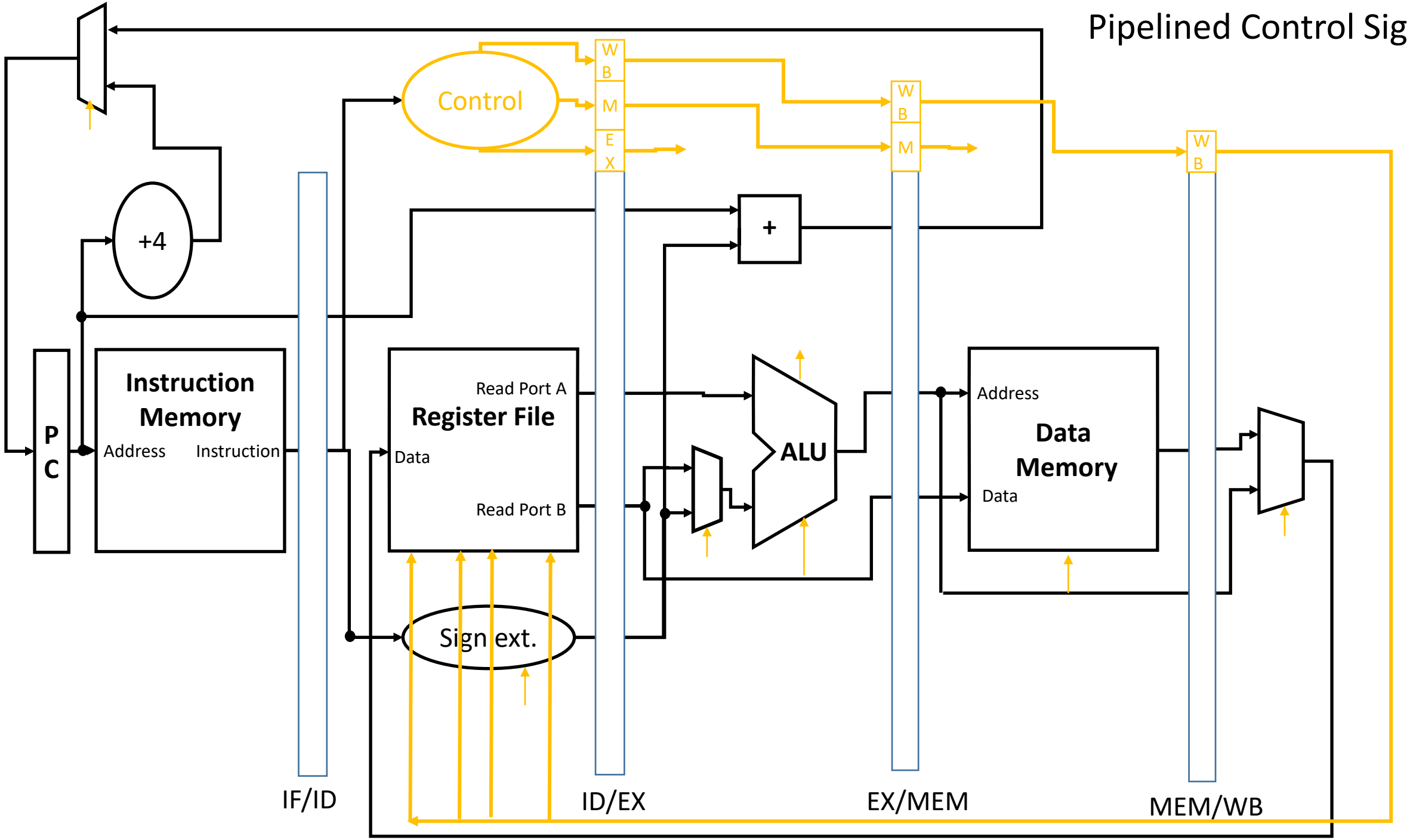
Pipelined Control Signals



Pipelined Control Signals



Pipelined Control Signals



Does Pipelining Really Work so Nicely?

Remember the Properties of an Ideal Pipeline

Ideal setup for pipelining:

- **The operations are identical:** We need to repeat the same operations over and over again (e.g. wash 10.000 loads of cloths)
- **The operations are independent:** We can perform the operations in any sequence we want
- **Uniform partitioning into suboperations is possible:** Each operation is can be divided into suboperations that take the same amount of time

Reality

- **The operations are identical → NO**

  We need to force different instruction through the pipeline architecture

- **The operations are independent → NO**

  Instructions have dependencies (e.g. operand dependencies) and we need to resolve dependencies and ensure that we compute the result correctly

- **Uniform partitioning into suboperations is possible → NO**

  We need to handle different latencies on different pipeline stages (e.g. caused by multi-cycle suboperations)

The Challenge of Pipeline Design

- Goal: Keep the pipeline moving, full, and correct under all circumstances
- Approach: Add logic (“intelligence”) around the data path to achieve the goal. This logic copes with dependencies between instructions, different latencies, exceptions, …
- What we want to prevent is a so-called pipeline stall

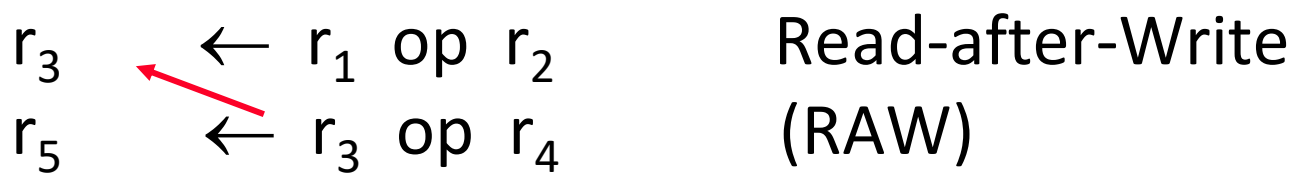
Pipeline Stall

- **Pipeline Stall:** Any condition that prevents the pipeline from moving, i.e. any condition that prevents that all instructions can move from the current stage to the next stage in the next clock cycle

- A stall is necessary, if on any pipeline stage either a needed hardware resource or data is not available (Think of a car assembly line – you have to stall the line, if it happens that on a given pipeline stage, a machine is not available, or if there is no car to work on)
  - Resource not available: e.g. a multi-cycle operation blocks a needed resource
  - Data not available: e.g. one instruction computes a result that is needed by the next instruction

Data Dependence Handling

Read-After-Write Dependency



When an instruction tries to access a registers that has not yet been written back to the register file, this is called a “data hazard”.

RAW Dependence Handling

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

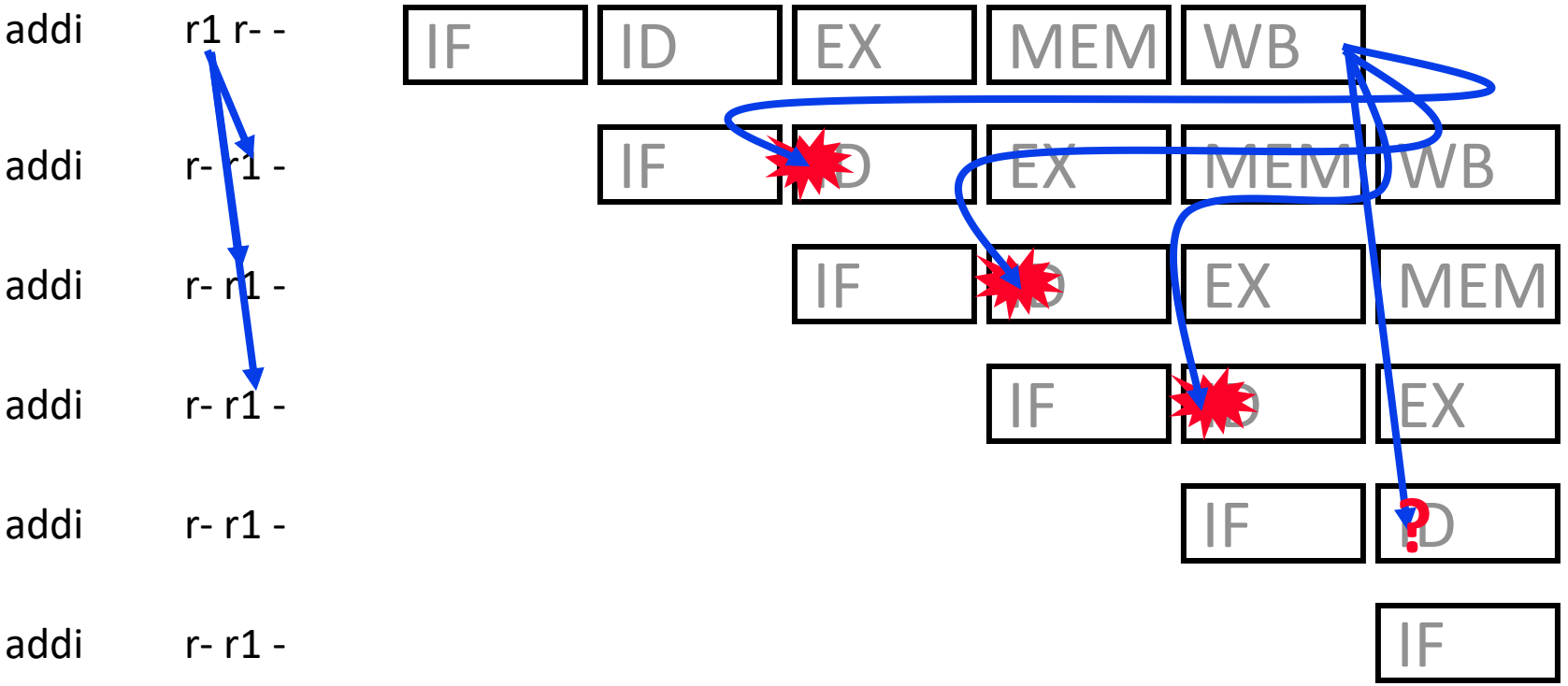


Figure by Onur Mutlu

Pipeline Stall: Resolving Data Dependence

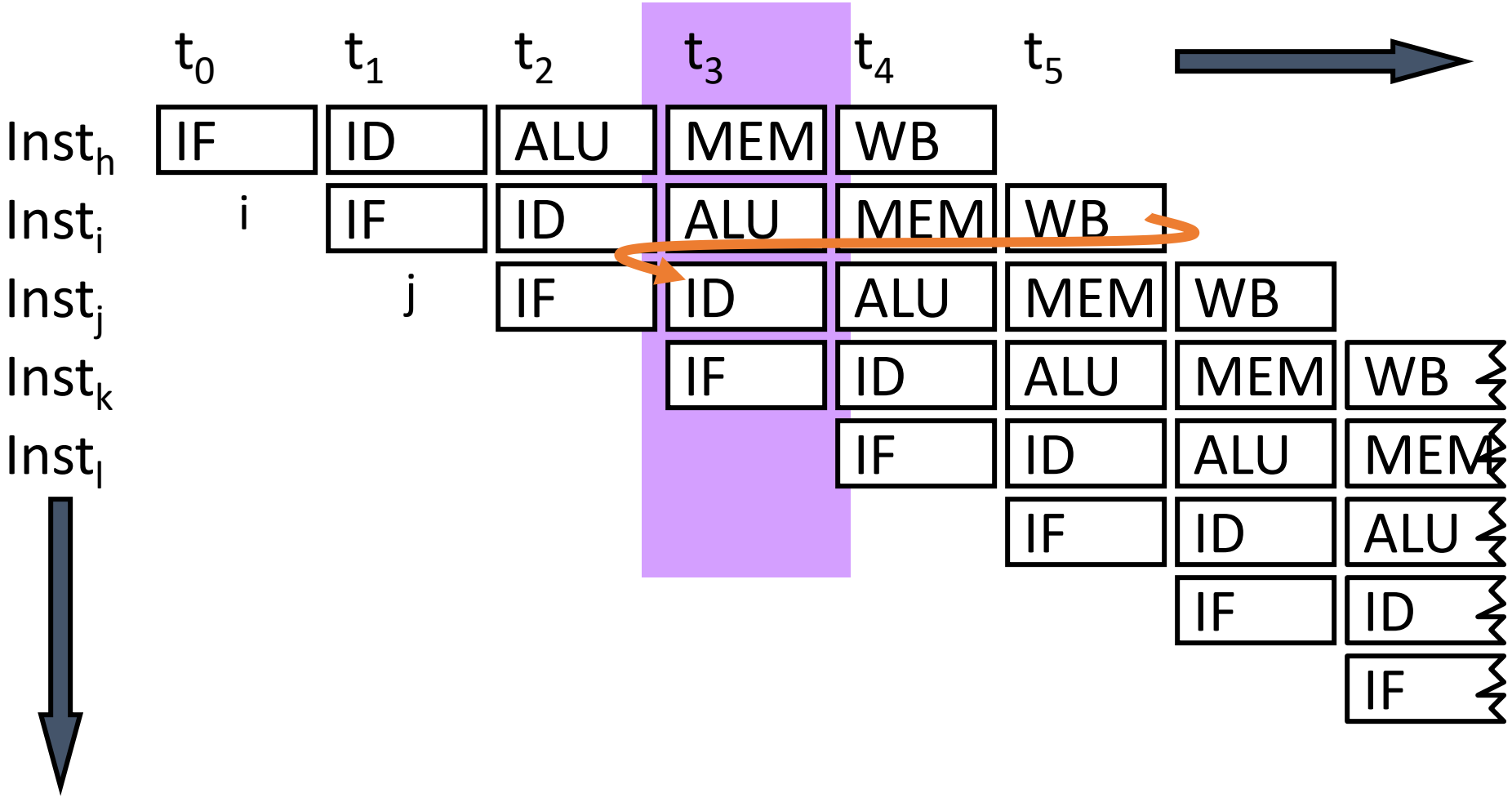


Figure by Onur Mutlu

Pipeline Stall: Resolving Data Dependence

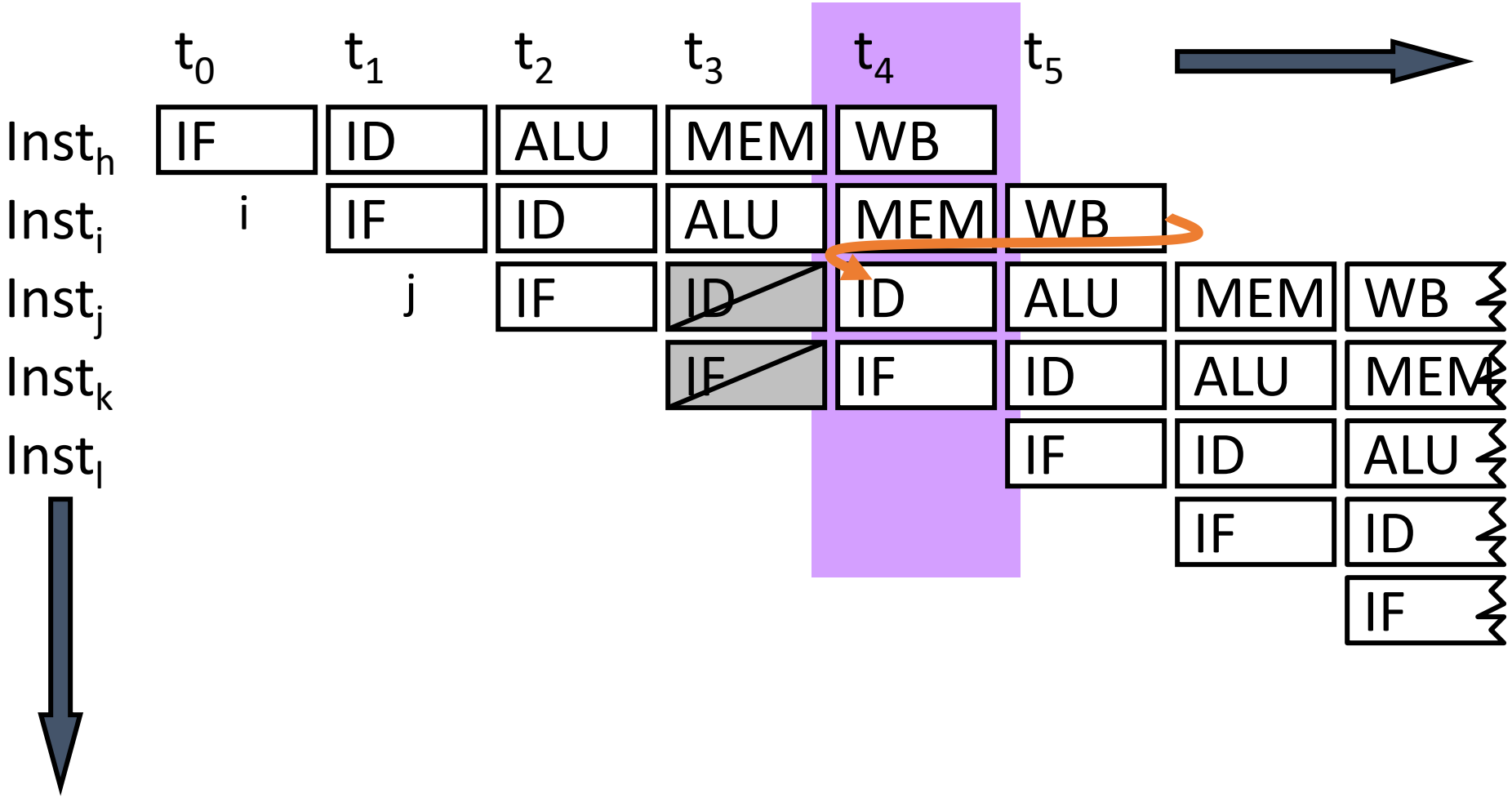


Figure by Onur Mutlu

Pipeline Stall: Resolving Data Dependence

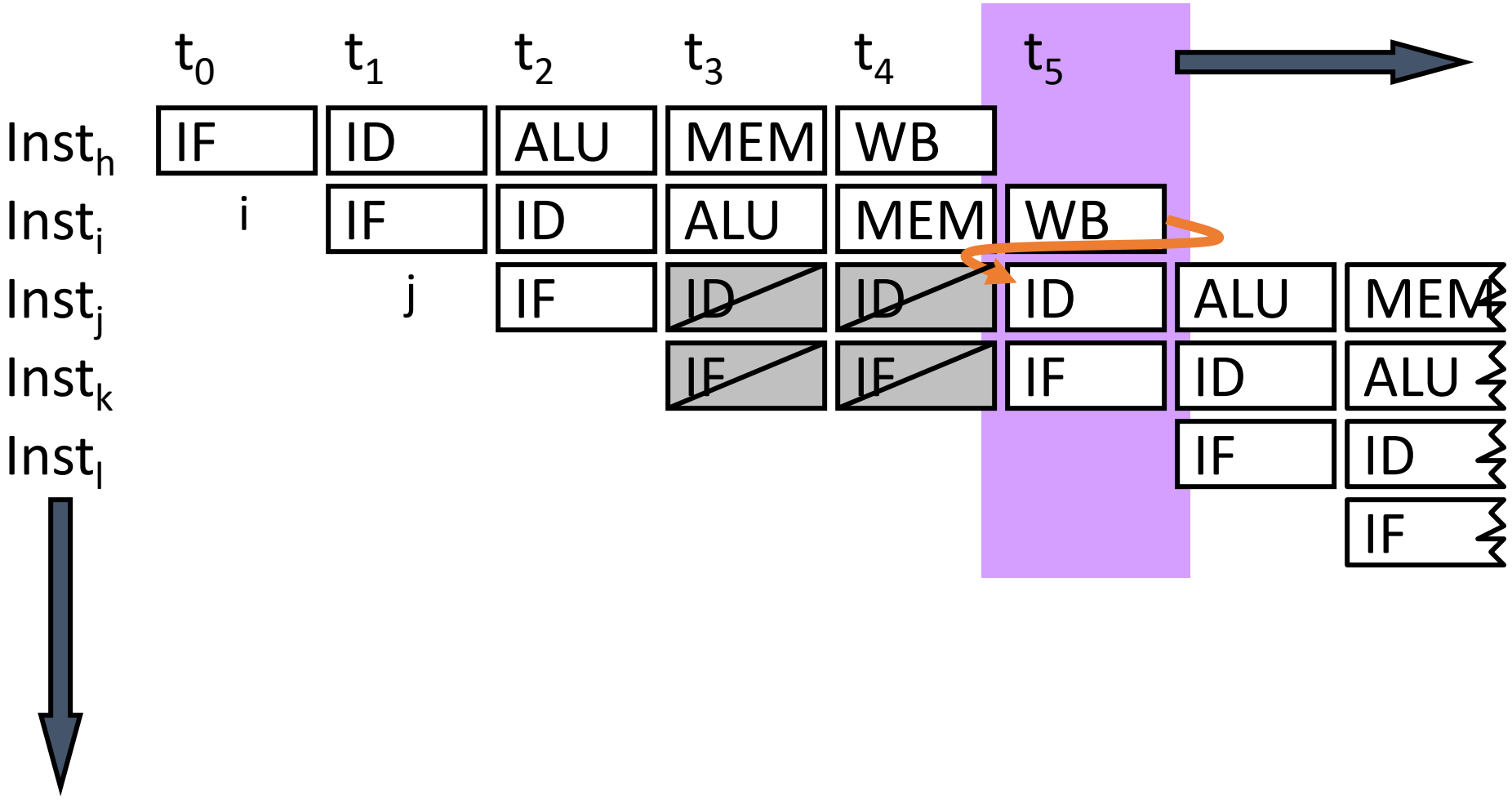
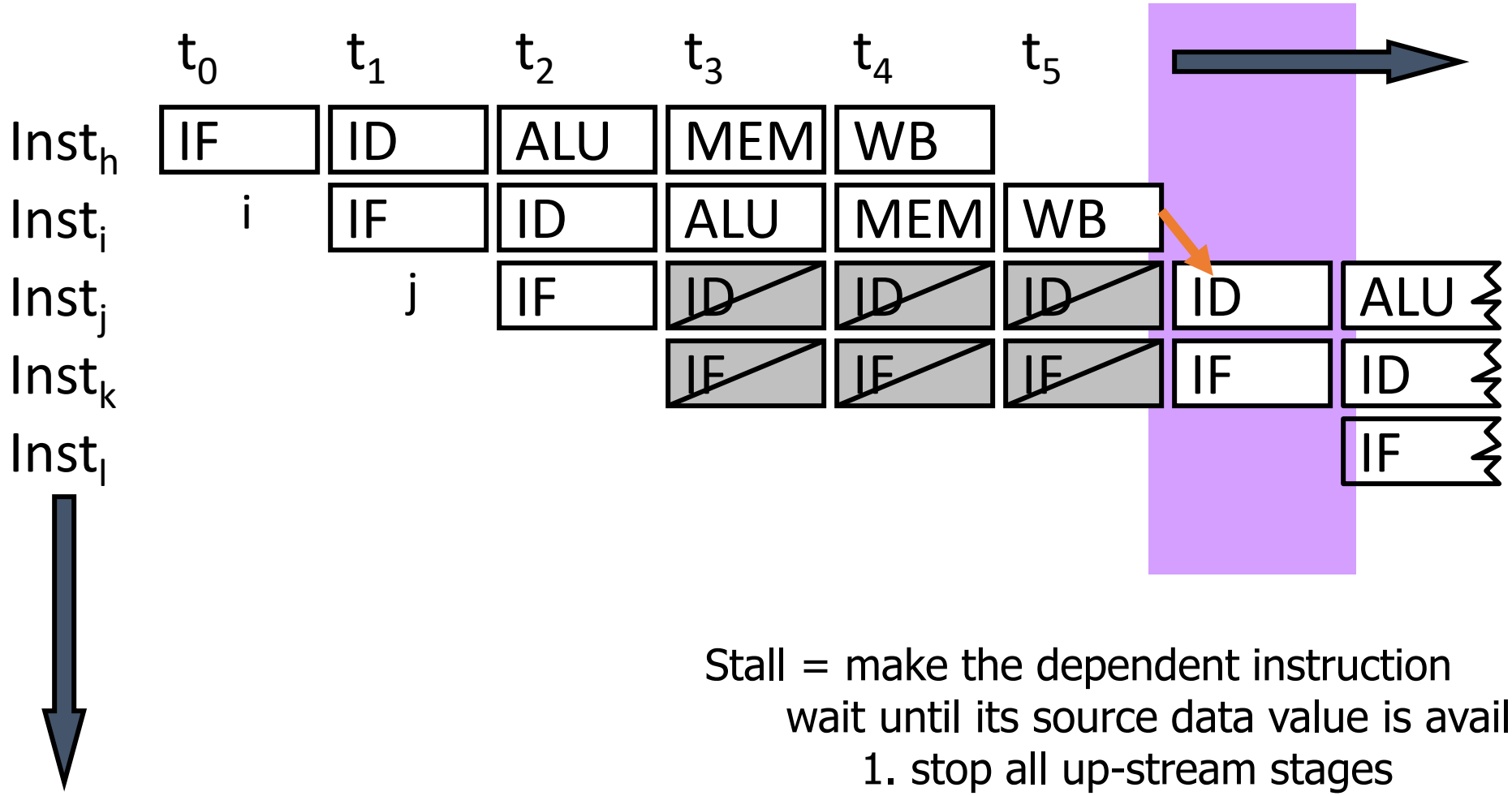


Figure by Onur Mutlu

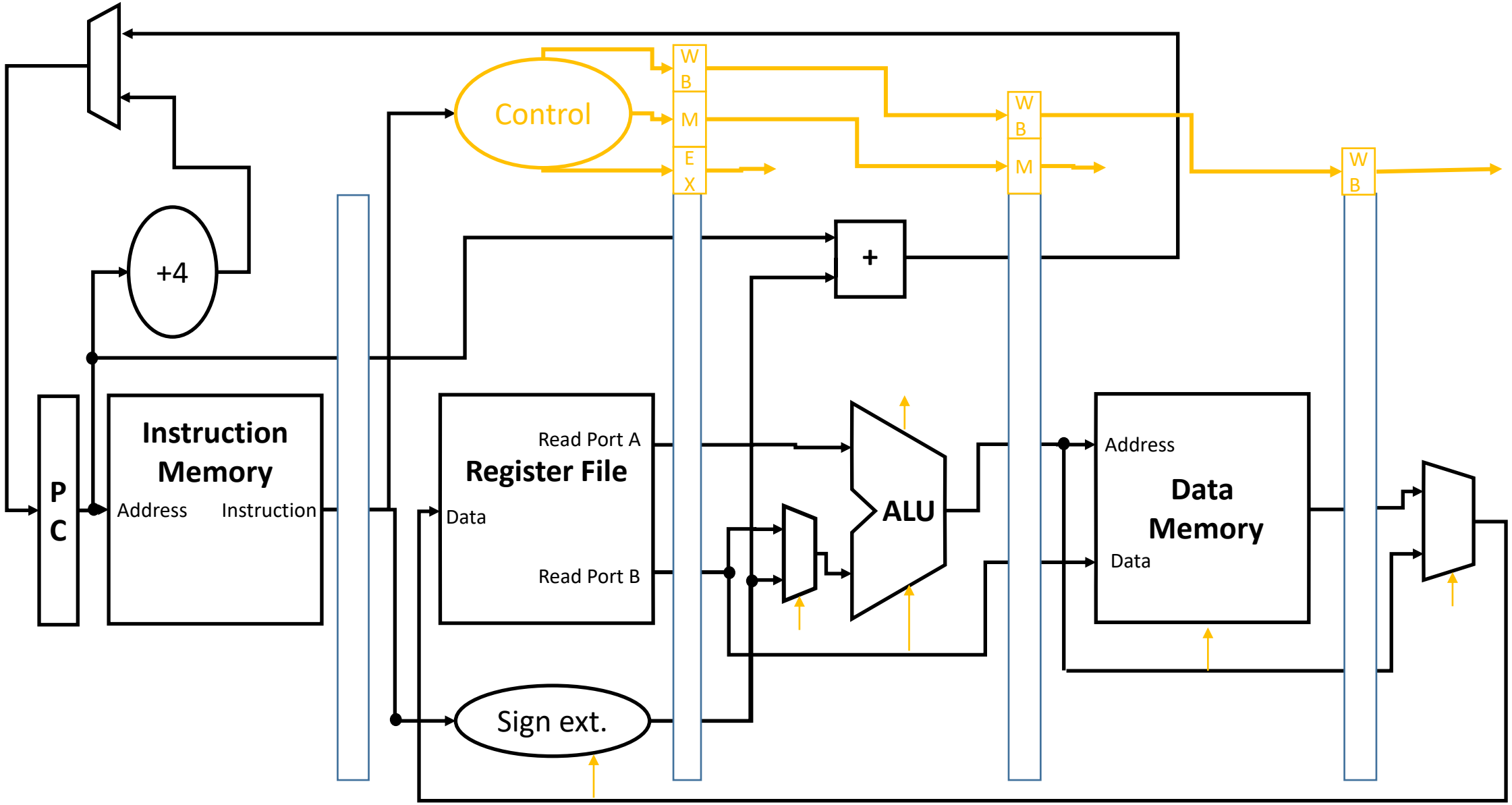
Pipeline Stall: Resolving Data Dependence



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

Figure by Onur Mutlu



- **Realizing a stall**
  - disable **PC** and **IF/ID** registers; ensure stalled instruction stays in its stage
  - Insert **"invalid"** instructions/nops into the stage following the stalled one (called "**bubbles**")

Dependence Detection

- Example Technique: Scoreboarding

- Idea:
  - Associate a Valid bit with each register of the register file
  - When an instruction is decoded that is writing to a register, it resets the corresponding 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

Dependence Handling after Detection

- Option 1: Stall the pipeline
- Option 2: Stall the pipeline only when necessary → data forwarding/bypassing

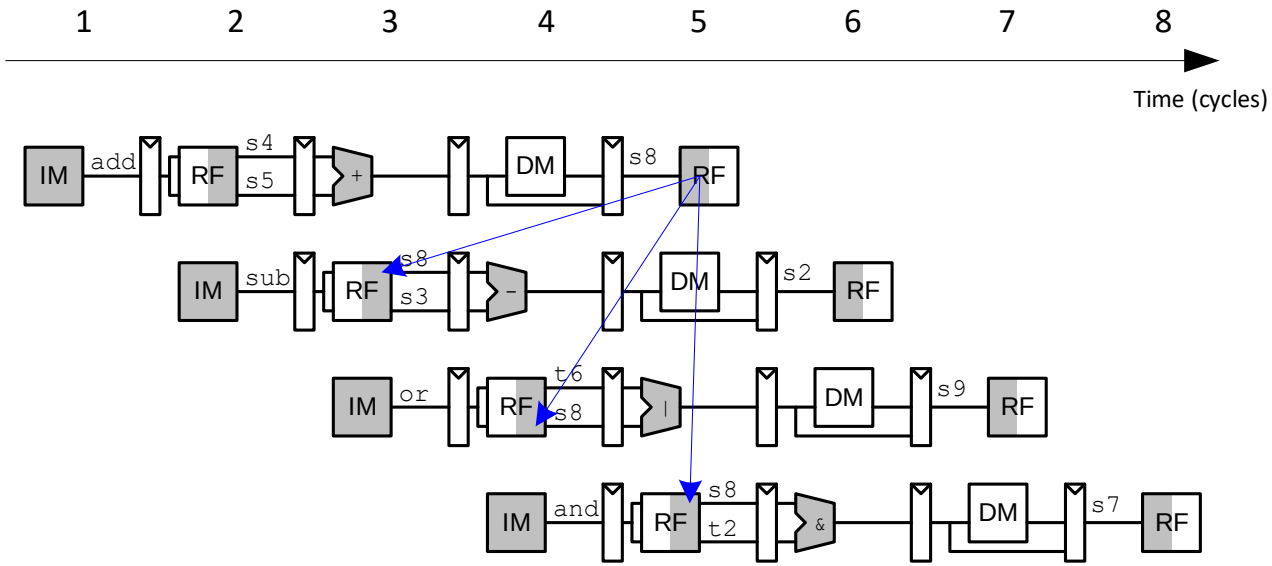
Data Forwarding/Bypassing

- **Challenge:** An instruction (the consumer) must 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/Idea:** The data value that is needed by the consumer is already available in a later pipeline stage → we can get it from there instead of the register file
- **Benefit:** The consumer can move in the pipeline until the point the value can be supplied → less stalling

RAW Data Dependence Example

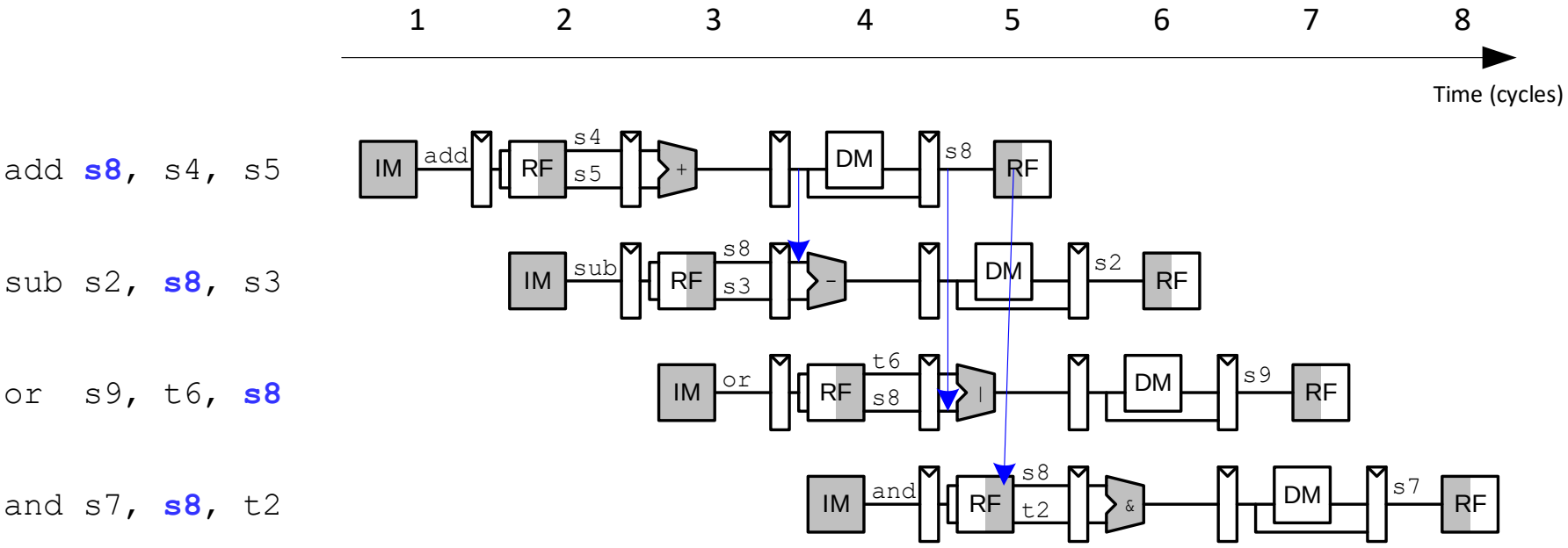
- The first instructions writes a register (s8) and next instructions read this register => read after write (RAW) dependence.
  - add writes into s8 in cycle 5
  - sub requires to read s8 on cycle 3
  - or requires to read s8 on cycle 4
  - and requires to read s8 in cycle 5

```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```

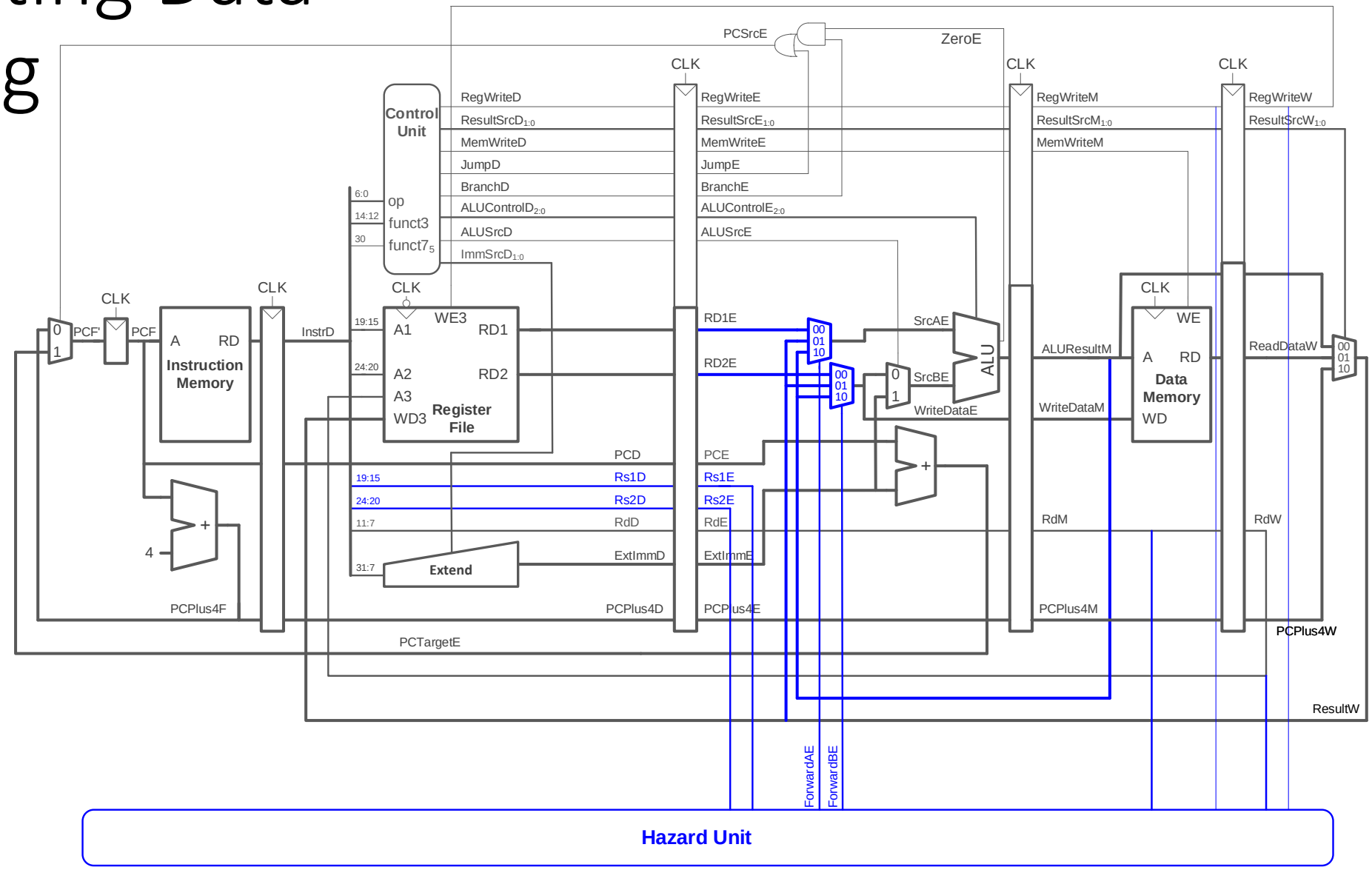


Data Forwarding

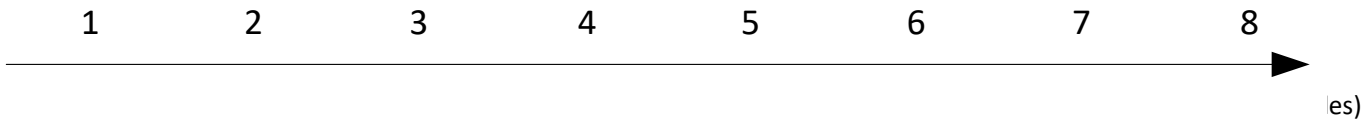
- Check if source register of the instruction transitioning to the execute stage matches the destination register of an instruction in the Memory or Writeback stage.
- If so, forward result → Data values are supplied to dependent instruction as soon as it is available



Implementing Data Forwarding

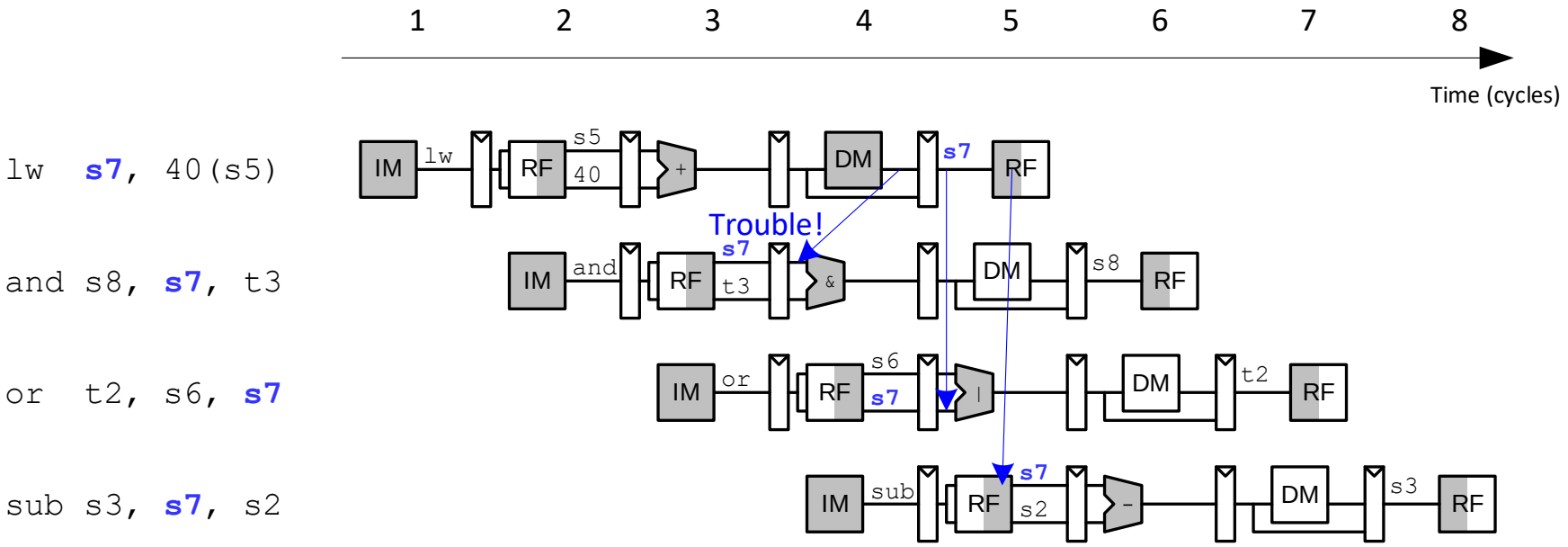


Stalling




```
lw  s7, 40(s5)

and s8, s7, t3

or  t2, s6, s7

sub s3, s7, s2
```

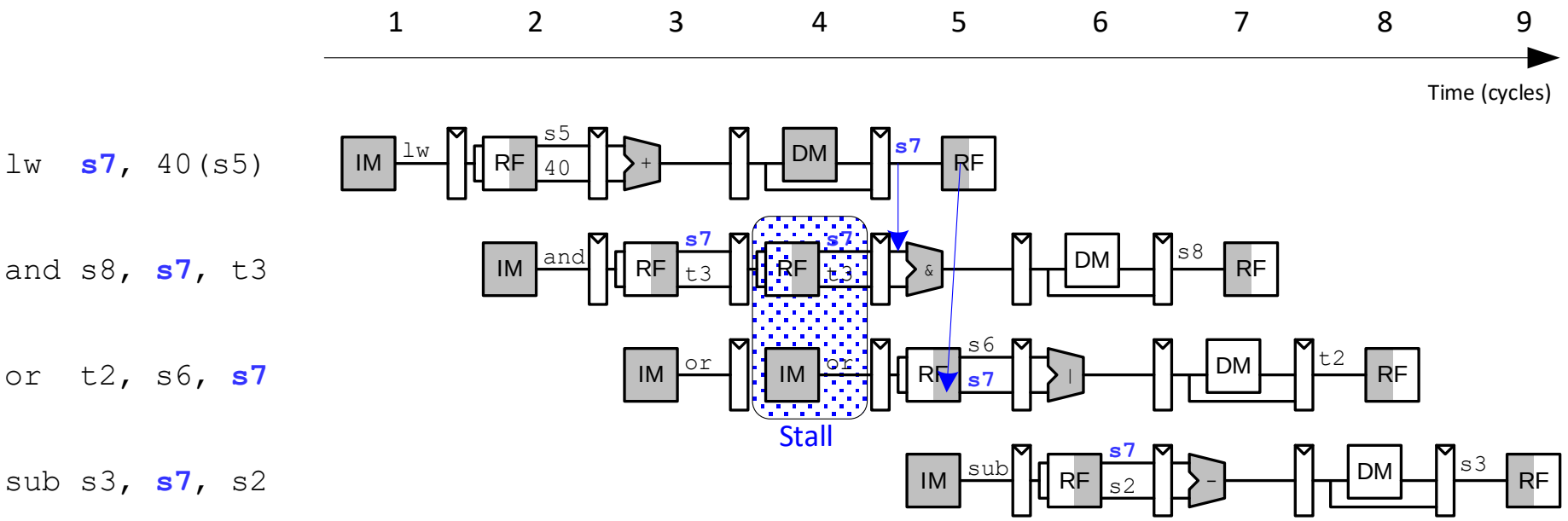
Stalling



- Forwarding works to resolve some RAW data dependences
  BUT
  there are cases when a stall cannot be avoided

- Example:
  - The lw instruction does not finish reading data until the end of the memory stage → a stall is necessary until the memory read completes

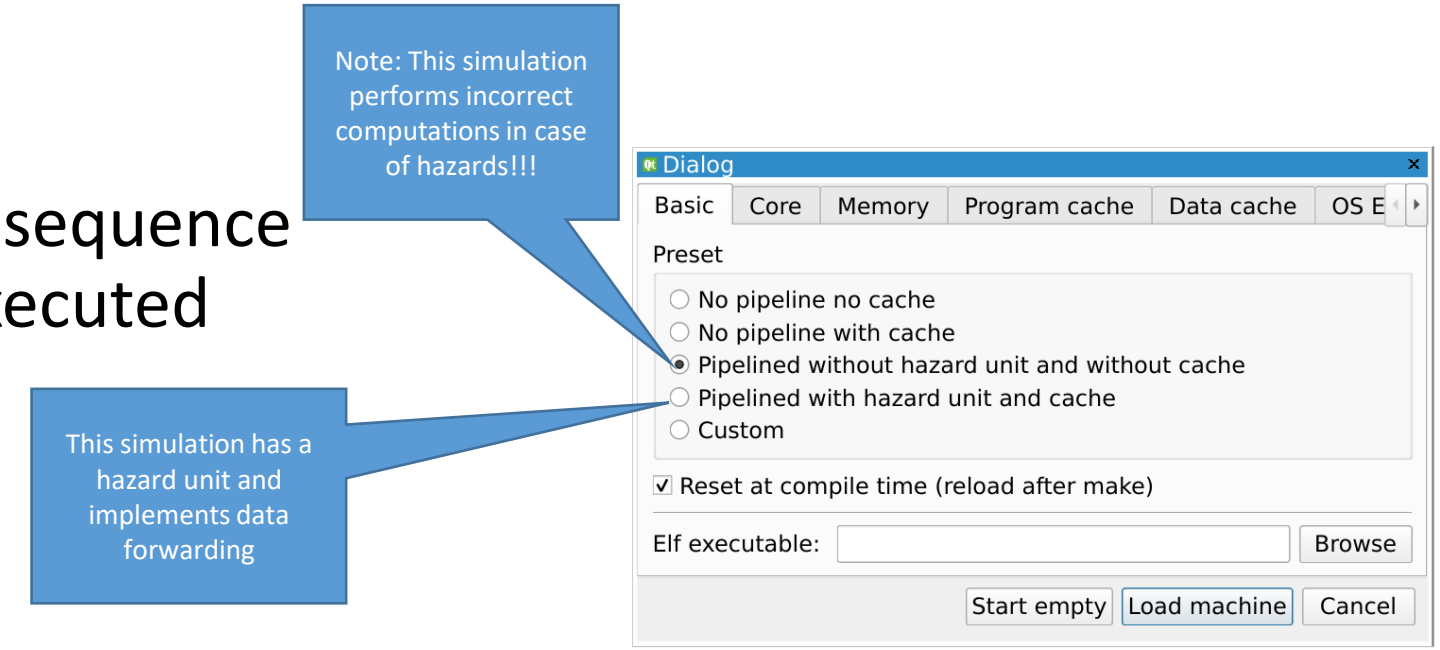
Stalling



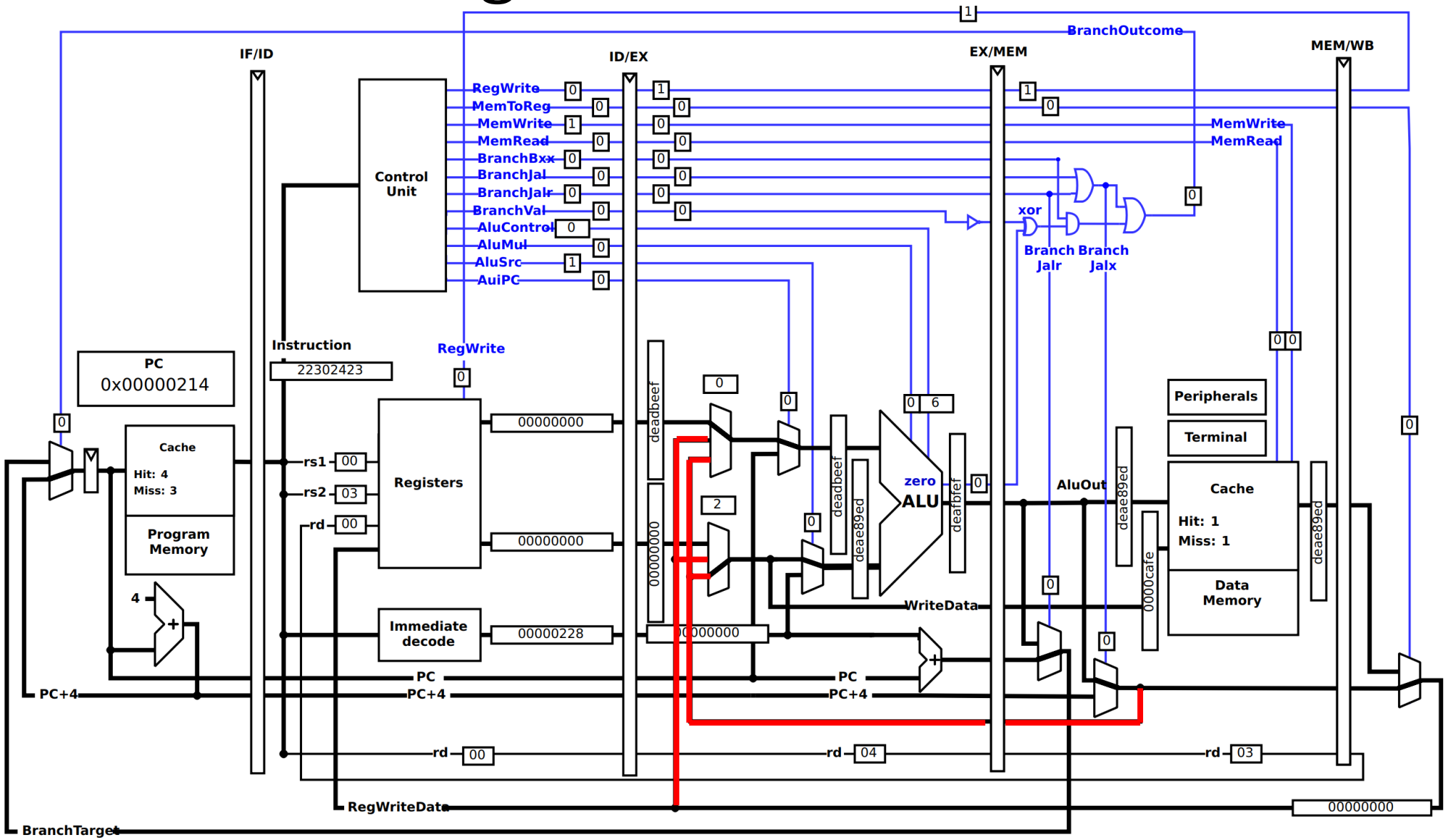
Watch the Operations of the Hardware while executing your code - QTRVSIM

Visit https://comparch.edu.cvut.cz/qtrvsim/app/ or use qtrvsim in your virtual machine

in order to visualize how a sequence of instructions becomes executed



Data Forwarding Paths in QTRVSIM



Examples

- Simulate the code examples in the directory

  con10.01_QtRVSim_pipeline_examples

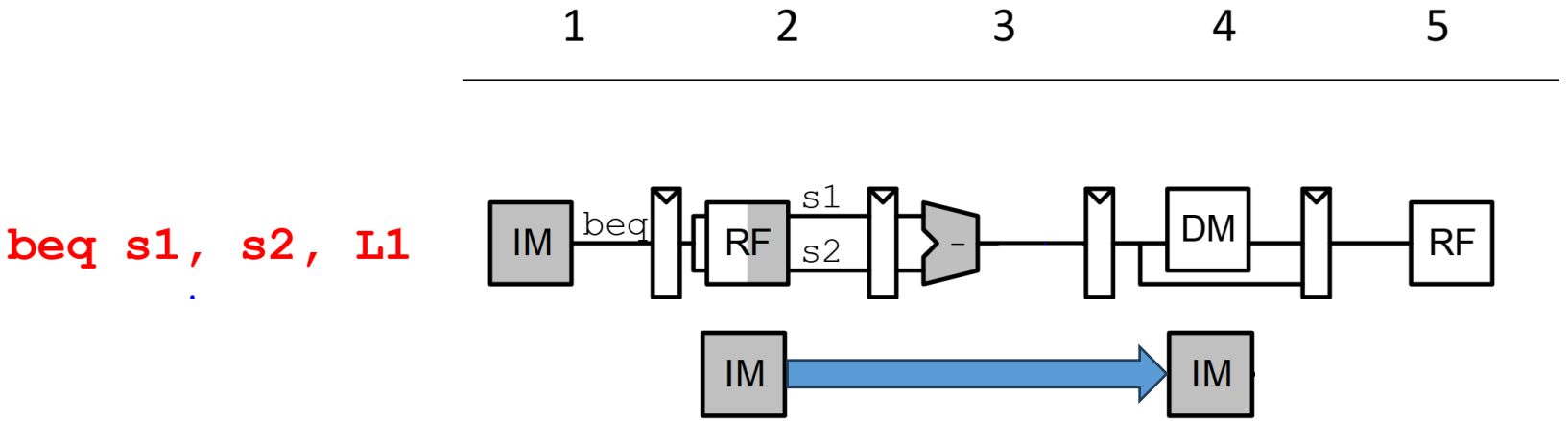
- Use the examples as starting point and change code to see stalling and forwarding

Control Hazards

Control Hazards

- To keep our pipeline full, we need fetch a new instruction in every cycle → we always need know what instruction needs to be fetched next.
- In linear code sequences, the next instruction to be fetched is simply the one the is located at the next memory address
- In case, there occur control-flow instructions (e.g. beq, bne, …), there are two possible execution paths (“branch taken” vs. “branch not taken”) and this causes a so-called “control hazard”
- A control hazard occurs when the decision of what instruction to be fetch next is not available by the time when the fetch takes place

When Do We Know the Branch Decision?



- To keep the pipeline full, we would need to fetch the next instruction in cycle 2
- We determine the branch target during the execution stage in clock cycle 3 (there we add the branch offset to the program counter and determine the new PC) → we can fetch the next instruction only in clock cycle 4
- This would mean for every branch, the pipleline would stall for 2 clock cycles

Can We Take the Decision Earlier?

- Idea: We could try to do the comparison of the of the registers (branch decision) and the calculation of the new branch target already during the decode stage.

- "Early Branch Resolution" is in principle possible, BUT
  - This would imply the need for additional forwarding paths (registers that determine the branch decision my not yet be in the register file)
  - There may still be stalls that are unavoidable: e.g.
    - a branch that directly follows the computation of a register that determines the branch target
    - a branch after a load instruction that determines the branch decision

→ "Early Branch Resolution" reduces the performance penalty of branches, but we can do better

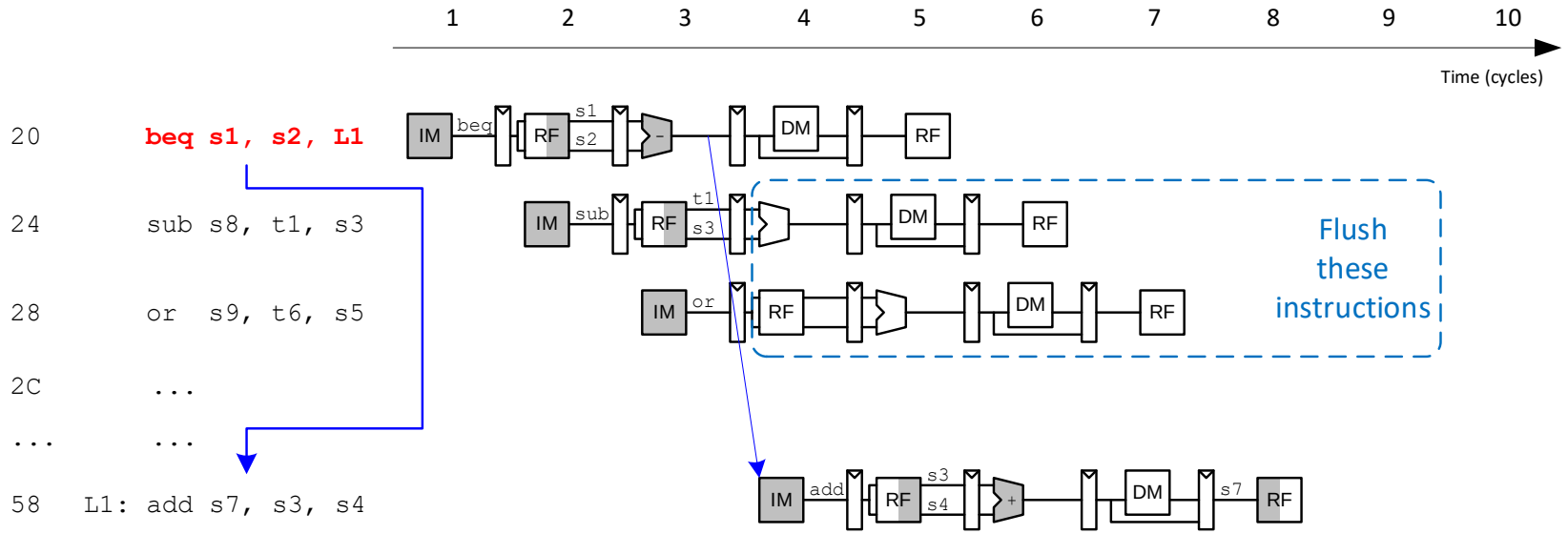
Branch Prediction

- Idea:
  - Instead of waiting for the decision, we simply try to correctly predict whether a branch is taken or not.
- Properties:
  - In case our prediction is correct, there is no performance penalty
  - In case our prediction is incorrect, we need to flush the pipeline

Example

- Let's do the default assumption that branches are not taken → we simply keep fetching the instructions following the branch

- Properties: Stall of 2 cycles if the assumption does not hold (called "misprediction penalty"); No stall if the assumption holds



Dynamic Branch Prediction

- Observe:
  - Making a static assumption, like “conditional branches will always be taken” or “conditional branches will always be not taken” are bad predictors
  - Consider a loop: there are two programming options
    - A conditional branch can be at the end of the loop and keeps jumping back to the beginning of the loop (“branch taken” occurs frequently)
    - A conditional branch can be used to exit a while loop with an unconditional branch at the end of the loop (“branch not taken” occurs frequently)

- Idea:
  - Implement a “branch predictor” that makes a dynamic prediction for every branch on whether it will be taken or not (Basic idea: “the hardware learns the behavior of the software and predicts based on past behavior”)

Branch Target Buffer

- Motivation:
  - To be able to make predicts about a specific branch, we need to story information about its history
- Implementation:
  - We can’t keep a table to store the outcomes of all branches from all potential addresses
  - Typically, a mapping function based on the lower bits of the address and additional context information is used to index a table storing information about the branching history
  - Size of the table, the indexing of the table, and the information that is stored vary for different architectures. A trade-off needs to be made between prediction accuracy and implementation costs

  (Note: The deeper the pipeline, the higher is the cost of misprediction and the higher invests are made into increasing the success rate of the prediction)

The Simplest Branch Predictor

- The simplest branch predictor is to store for a given branch if it was taken or not during the last execution (1 bit storage)
- Performance in example loop: misprediction during first and last loop

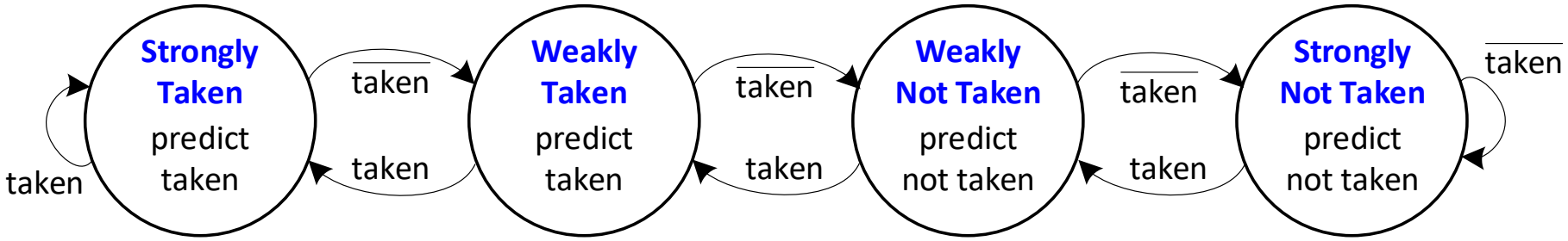
```
  addi s1, zero, 0      # s1 = sum
  addi s0, zero, 0      # s0 = i
  addi t0, zero, 10     # t0 = 10

For:                    # for (i=0; i<10; i=i+1)
  bge  s0, t0, Done
  add  s1, s1, s0       # sum = sum + i
  addi s0, s0, 1        # i = i + 1
  j    For

Done:
```

Note on misprediction in first loop:

A loop in a function is typically not only executed once (a function containing a loop is typically called several times.

Whenever the function is called, the branch predictor remembers during the last execution, the branch was taken → this leads to a misprediction during the first loop

2-Bit Branch Predictor



- 4 states encoded in 2 bit
- Only mispredicts the last branch of the loop (the branch exiting the loop)
- Solves the problem of the 1-bit branch predictor: In settings where the loop is executed multiple times, the branch is predicted correctly during the first loop

Additional Techniques in Current CPUs

We have now covered basic mechanisms for resolving data and control hazards, but current CPUs implement many more techniques to increase instruction throughput

We discuss two additional techniques
present in all large mainstream processors:

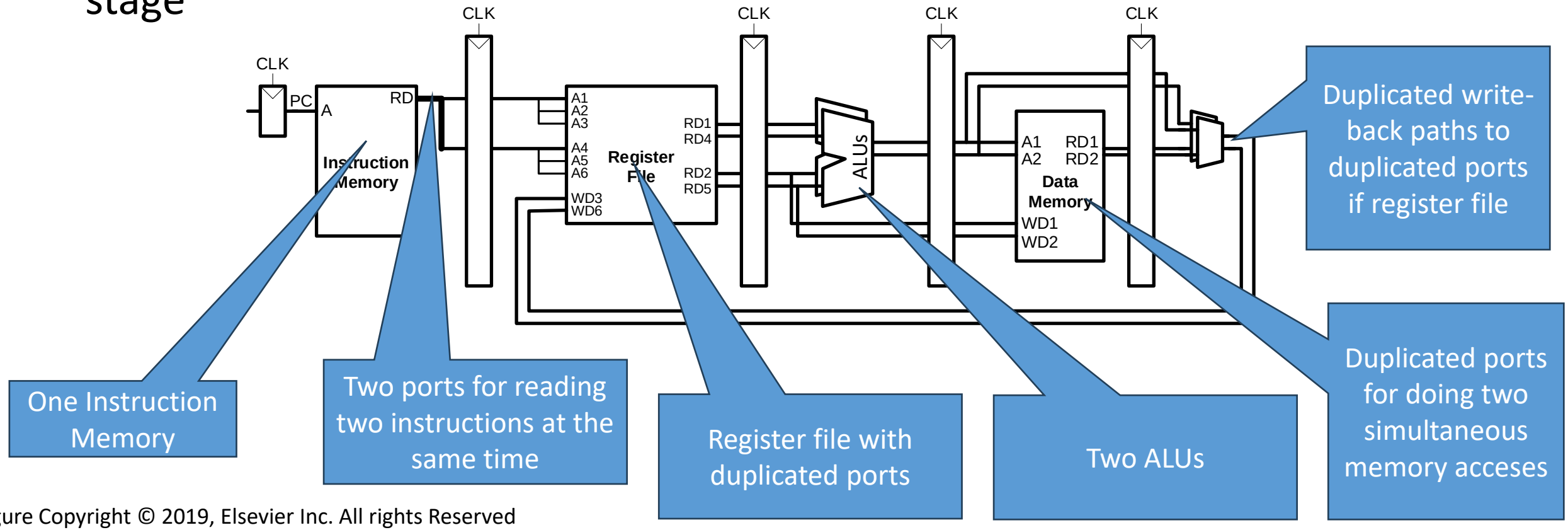
- Superscalar Designs
- Out-Of-Order Execution

Basic Idea of Superscalar Designs

- Motivation:
  - In order to speed up a pipelined design, we can introduce more and more pipeline stages (More pipeline stages → shorter critical path per stage → higher clock frequency)
  - This can't be done to an arbitrary level (remember: we need to balance the pipeline stages), and typical CPUs have roughly 10 to 20 pipeline stages
- Idea:
  - Instead of just processing one instruction on each stage, we could process multiple instructions in each stage → implement multiple units for decoding, execution, … and therefore
  - We can issue multiple instructions per cycle (IPC)

Multiple-Issue Processors

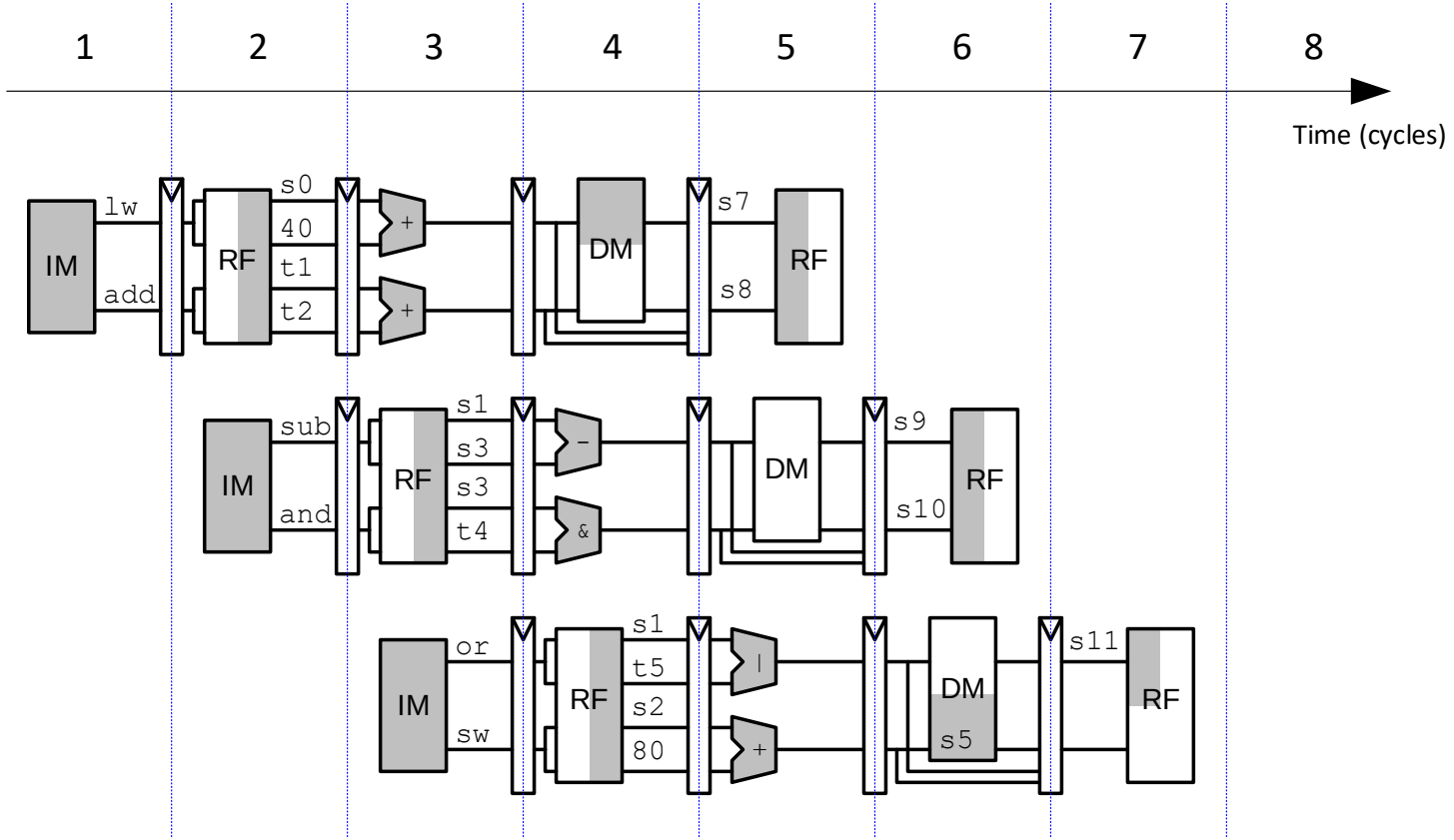
Multiple-Issue processors handle multiple instructions in each clock cycle, e.g. A dual issue-processor can handle two instructions on each stage



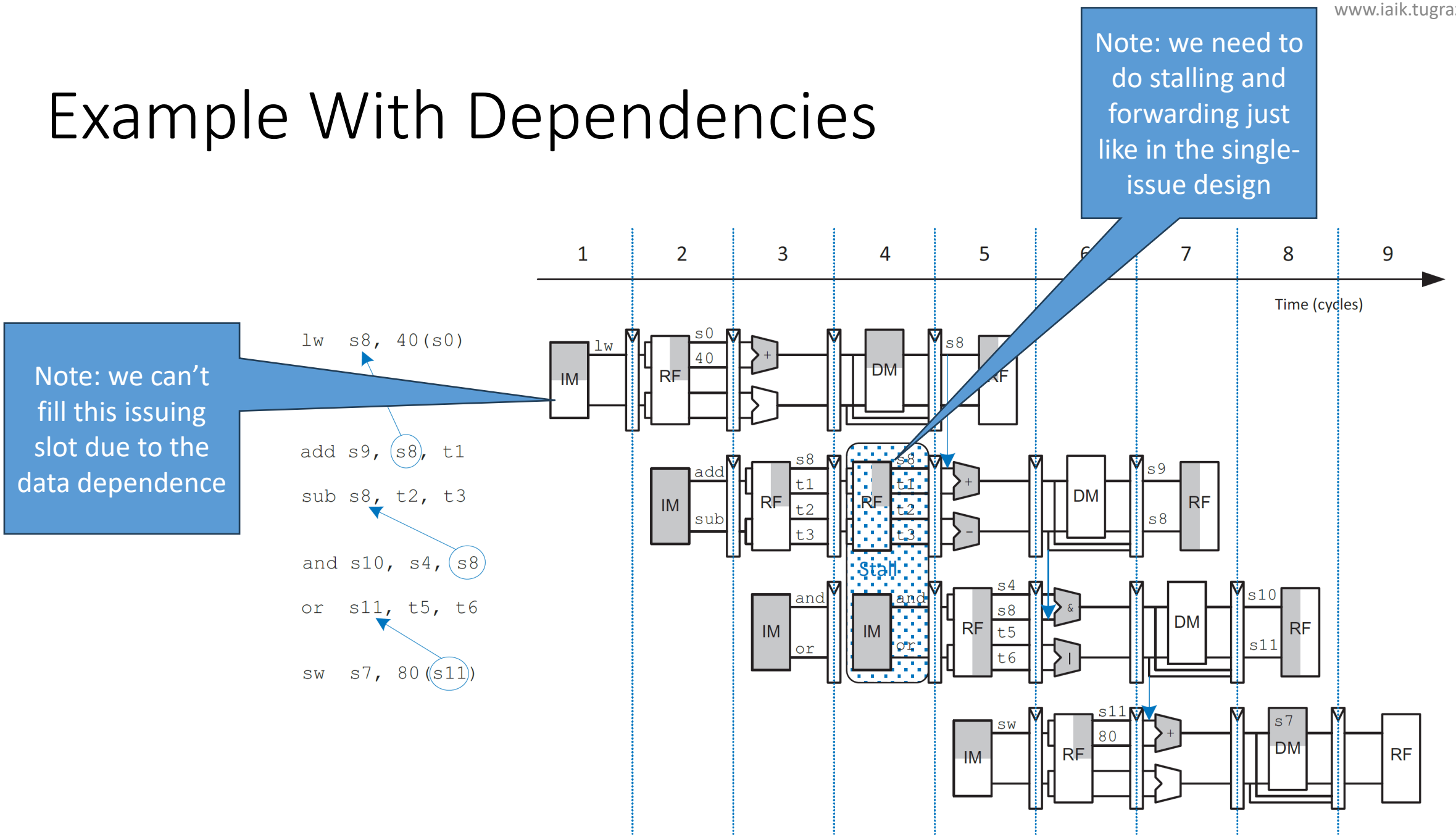
Ideal Example

- In case there are no dependencies, it is possible to execute two instructions per cycle

```
lw   s7, 40(s0)
add  s8, t1, t2

sub  s9, s1, s3
and  s10, s3, t4

or   s11, s1, t5
sw   s5, 80(s2)
```



Example With Dependencies



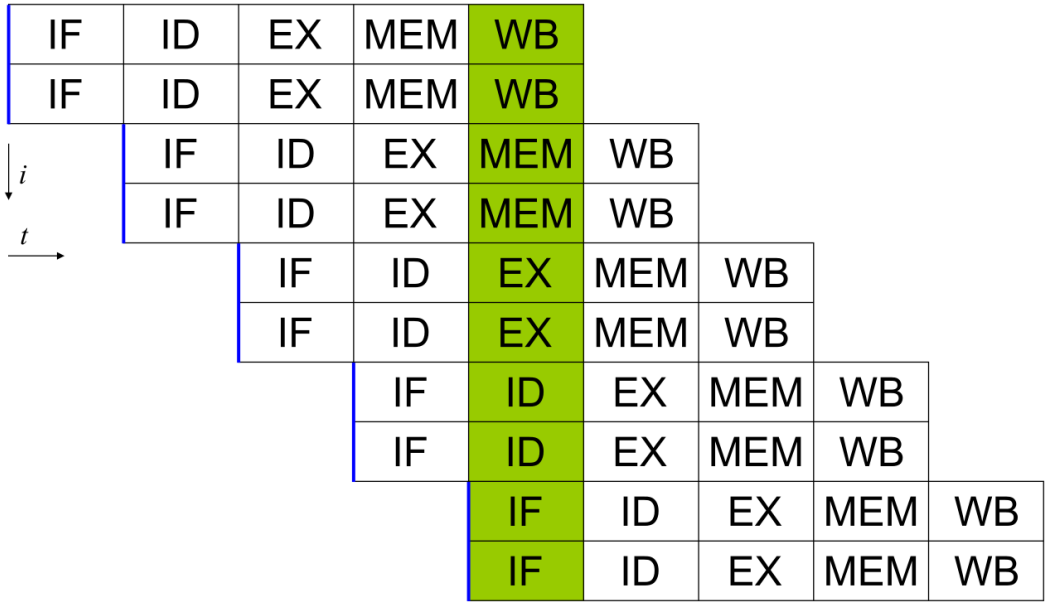
Multiple-Issue Processors

- The figure summarizes the pipeline stages in a dual-issue processor

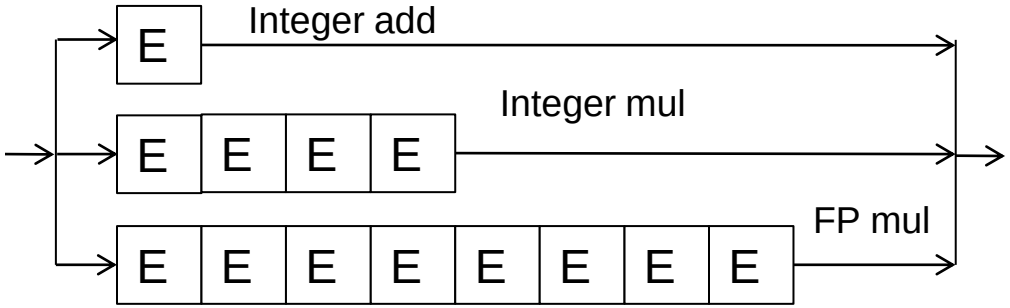
- Note:
  When scaling further up it might not be useful to replicate the entire ALU and all the ports for the memory stage

  Not every instruction requires all functional units of the execute stage or the option for memory read/write

→Designs in practice have specific compositions of functional units of the different stages



Multi-Cycle Execution



- The execution stage may contain different functional units with different execution times
- An execution stage may contain two units for add/sub, one unit for multiplication, a unit for floating point (FP) operations, …
- Note: A simple add/sub takes less time than a multiplication and a FP operation can take even more time

Figure by Onur Mutlu

Challenge

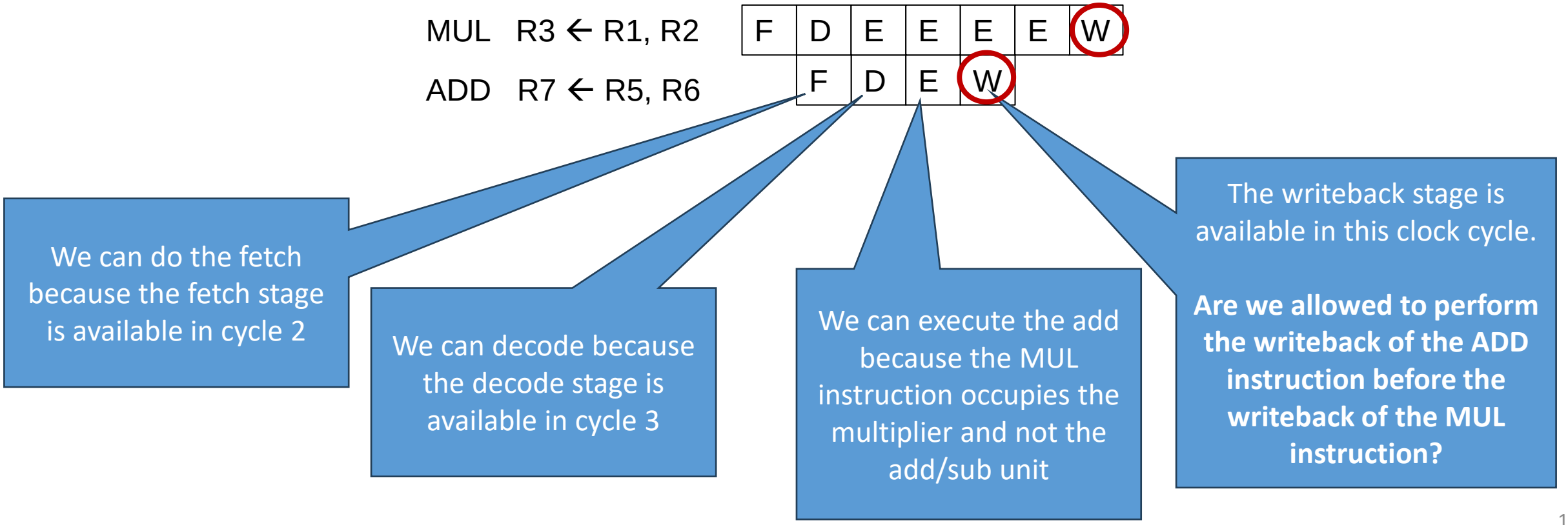
- Situation
  - Superscalar CPUs can fetch multiple instructions in each cycle
  - The instructions will then then be executed by the corresponding execution units

- Challenge
  - There will be data dependencies between the instructions that stall instructions
  - There will be multi-cycle instructions that also delay subsequent instructions

→ Significant delays can occur because of instructions that take longer to complete (e.g. a load from memory) and/or data dependencies

Example

- Assume the following two instructions are executed on the same issuing path



The Hardware-Software Contract

- In case we would complete the writeback of the ADD before the MUL, we would create an invalid state architectural state

- The ISA defines that instructions are executed one after the other

- Imagine that you as a programmer debug a program und you notice that the registers of the register file are not written one after the other, but in some “unpredictable sequence” → you would not be happy

→ The hardware needs to stick to the hardware-software contract. No matter how the microarchitecture realizes the implementation of an ISA, from a programmers view, there must not be a difference

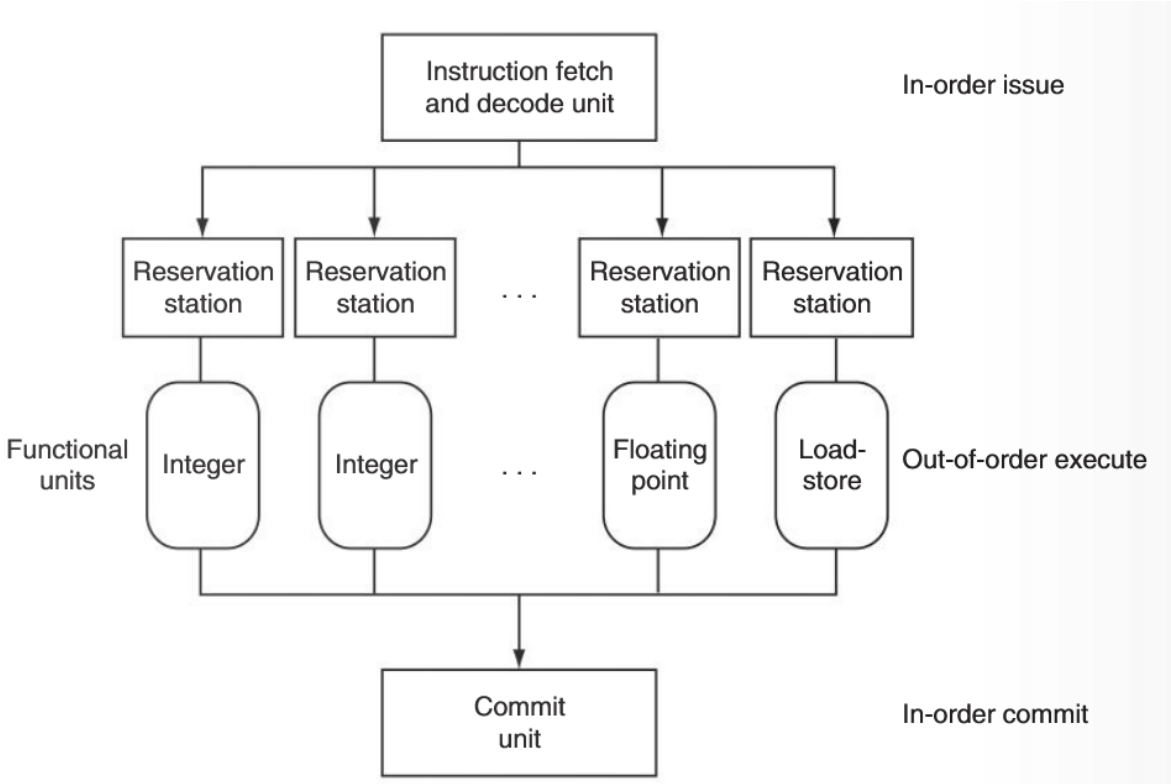
Out-Of-Order (OoO) Execution / Dynamic Instruction Scheduling

- It is an architectural requirement that instructions need to be completed in order
- Note however that completing instruction in order does not mean that we need to execute them in order
- Idea:
  - Execute instructions out-of-order and store the results in temporary registers
  - Perform the writeback from the temporary registers to the register file in order
- Motivation/Properties
  - Higher throughput through better utilization of the hardware
  - Shift the paradigm from “execute one instruction after the other” to “execute an instruction whenever it is ready to execute”, where “ready” means that the input data is available and the required execution unit is available

High Level View of Out-of-Order Execution

Basic Idea:

- The Instructions are fetched and decoded
- Dependencies need to be resolved
- Instructions that are ready for execute (input data available) wait for execution in a reservation station → out-of-order execution
- Results are provided to the commit unit that forwards the results back to the register file in order



Dependencies That We Need to be Considered for OoO Execution

- True Data Dependence / Read-after-Write (RAW)

  r3 ← r1 op r2

  r5 ← r4 op r3 “The data we need as input is not available yet”

- Anti Dependence / Write-after-Read (WAR)

  r3 ← r1 op r2

  r1 ← r4 op r5 “The register that we write to is needed as an input for another instruction”

- Output Dependence / Write-after-Write (WAW)

  r3 ← r1 op r2

  r5 ← r3 op r4

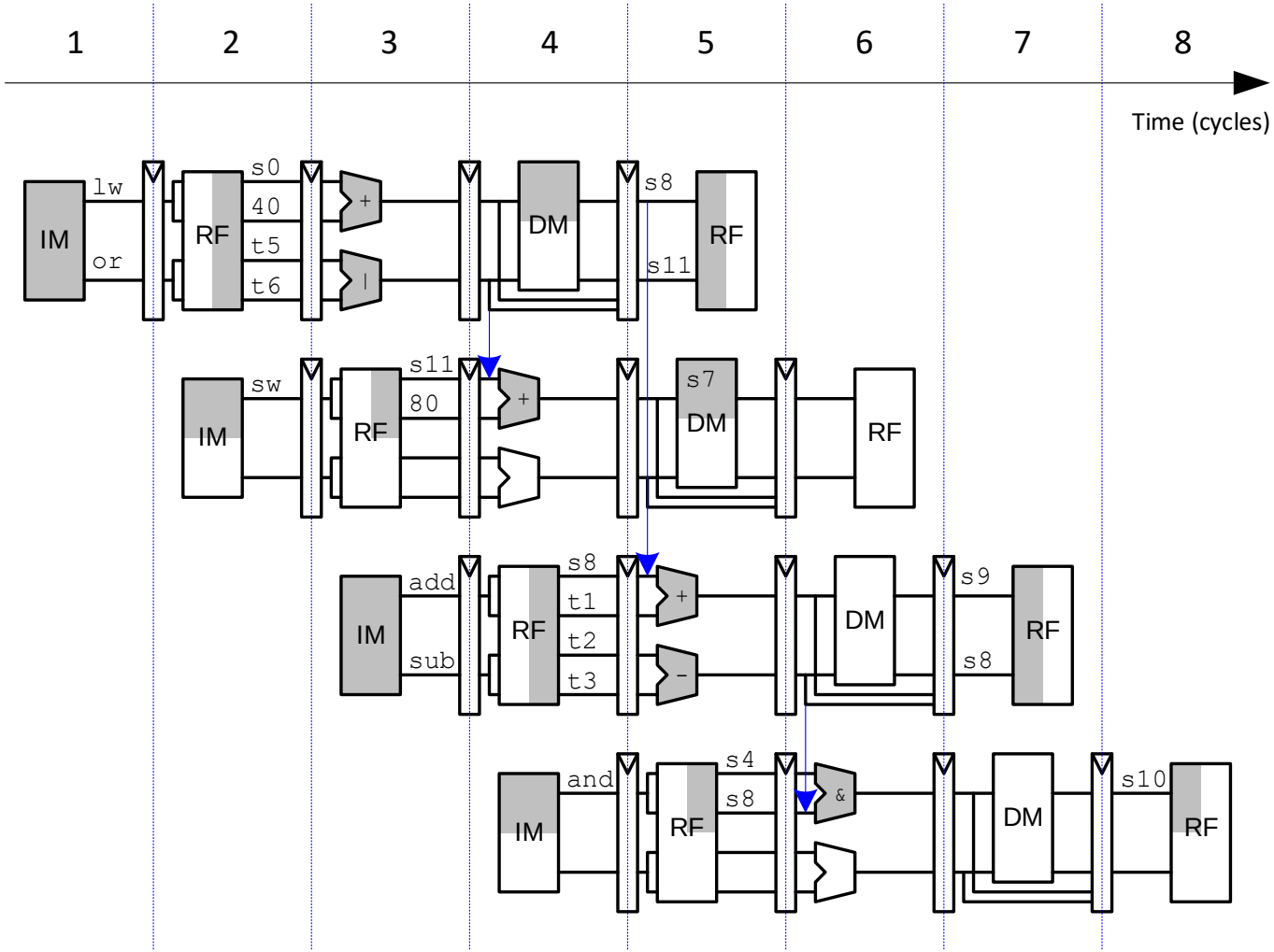
  r3 ← r6 op r7 “The register we write to is also written to by another instruction”

  (Note: the first instruction needs to be executed in any case.
  The state of the register file must be independent of the microarchitecture.
  Otherwise, we would violate the ISA. Imagine debugging your program on a hardware
  that would decide on its own which instruction is necessary to be executed and which not.
  Optimizations of code take place at compiler level.)

Example for a Dual-Issue Pipeline With OoO

The Program

```
lw   s8, 40(s0)

add s9, s8, t1
sub s8, t2, t3

and s10, s4, s8
or   s11, t5, t6

sw   s7, 80(s11)
```



Example for a Dual-Issue Pipeline With OoO

The Program

```
lw   s8, 40(s0)
add  s9, s8, t1
sub  s8, t2, t3
and  s10, s4, s8
or   s11, t5, t6
sw   s7, 80(s11)
```

Re-ordered Program

```
lw   s8, 40(s0)
or   s11, t5, t6
sw   s7, 80(s11)
add  s9, s8, t1
sub  s8, t2, t3
and  s10, s4, s8
```

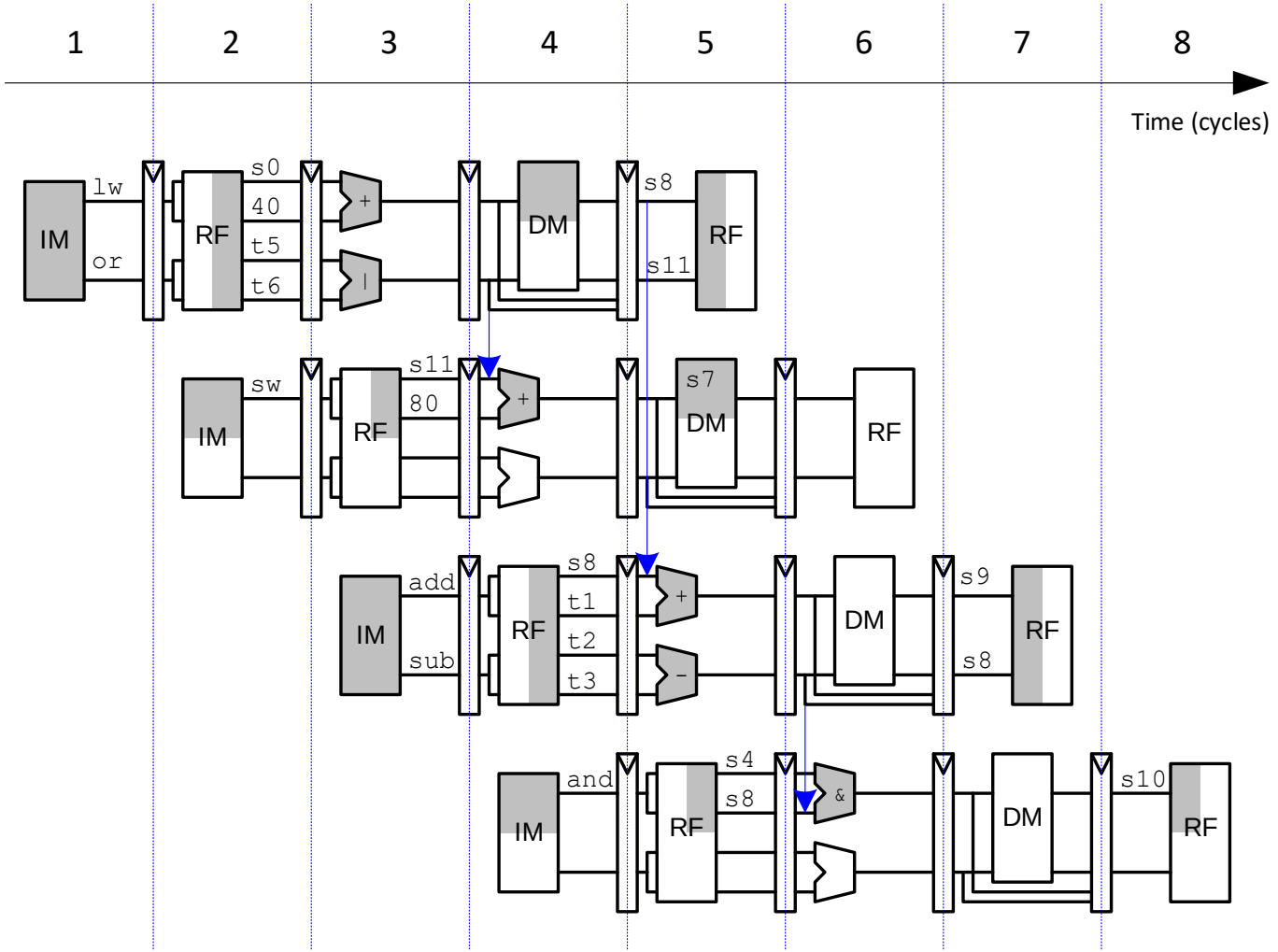
RAW

RAW

two cycle latency between load and use of **s8**

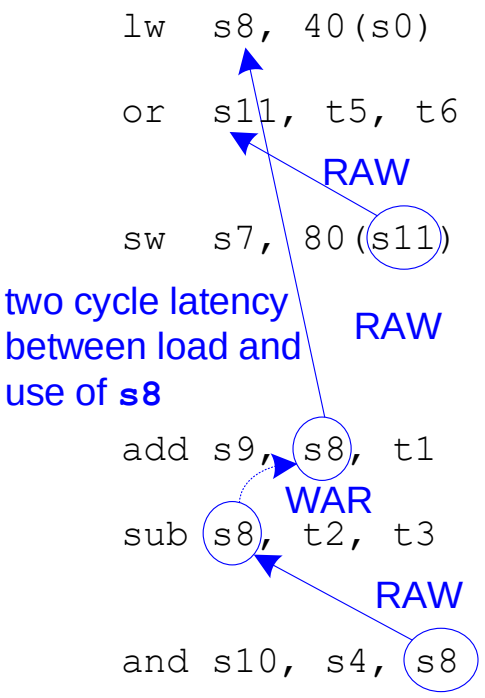
WAR

RAW



Register Renaming

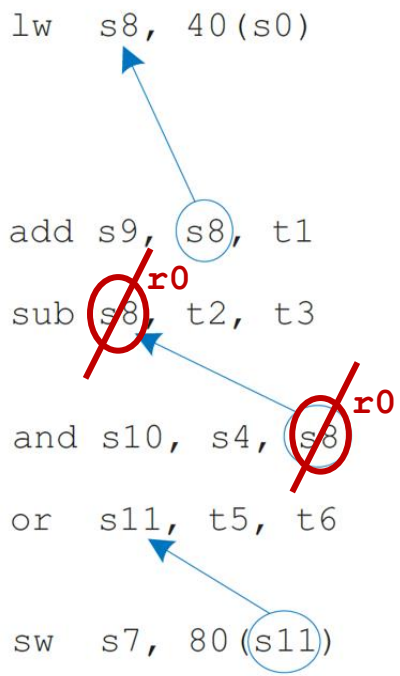
- Observe:
  - If the programmer used a different register than s8 in the sub instruction, there would be no dependence
  - Anti Dependence / Write-after-Read (WAR) and Output Dependence / Write-after-Write (WAW) are not really dependencies
  - These dependencies are only caused by the fact that we have a limited number of registers
- Idea:
  - We resolve WAR and WAW dependencies by adding more registers in the microarchitecture
  - The programmer doesn't see these registers and can't address these registers
  - The hardware simply has a larger pool of registers (larger than what is architecturally visible) – the hardware needs to know which of its registers of the pool corresponds to which register for which instruction of the software (i.e. the hardware needs to know the renaming it does)

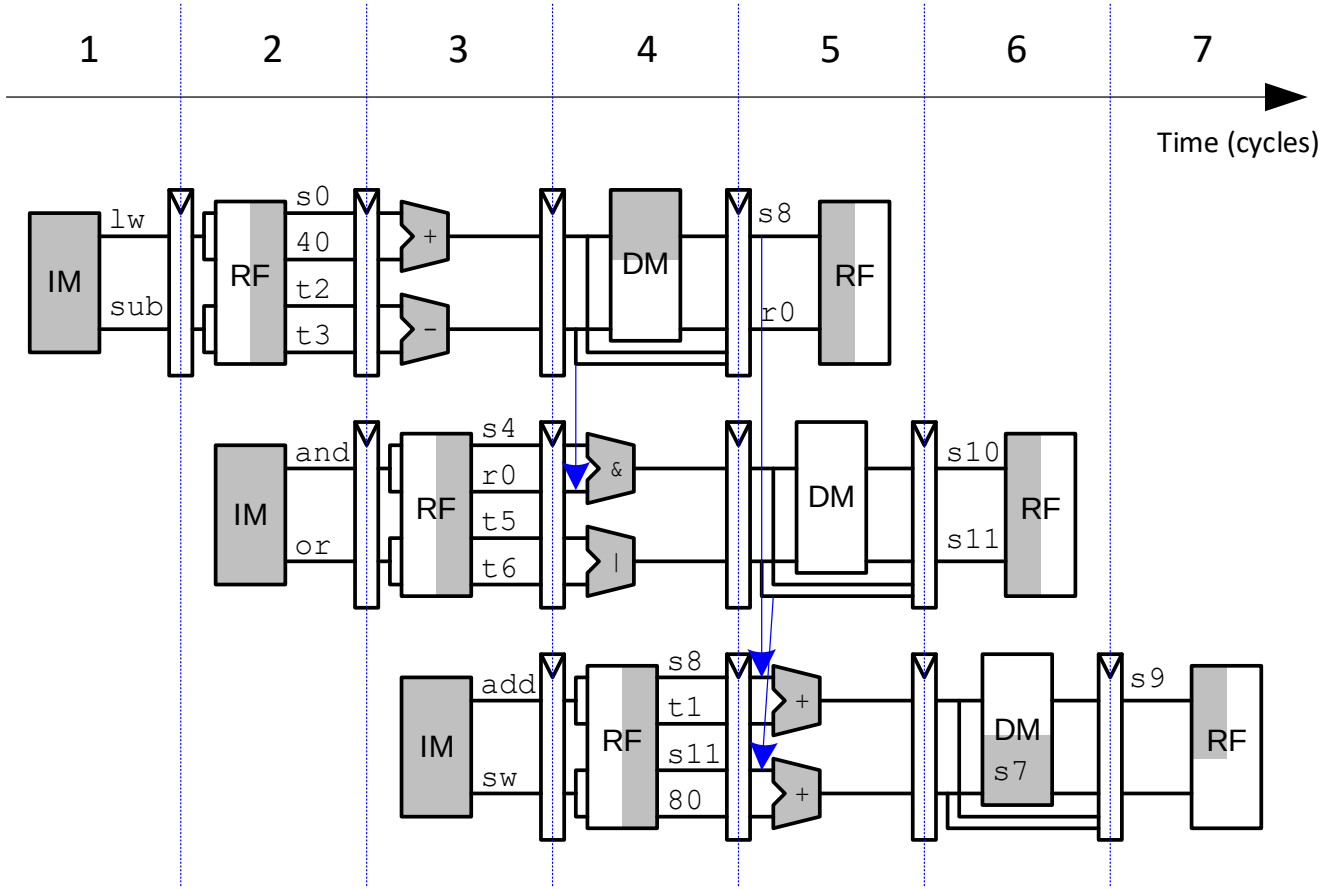


Simple Example for Register Renaming

- Assume the CPU internally has additional registers r0 … r20
- We use r0 to resolve the WAR dependence of our example program

The Program

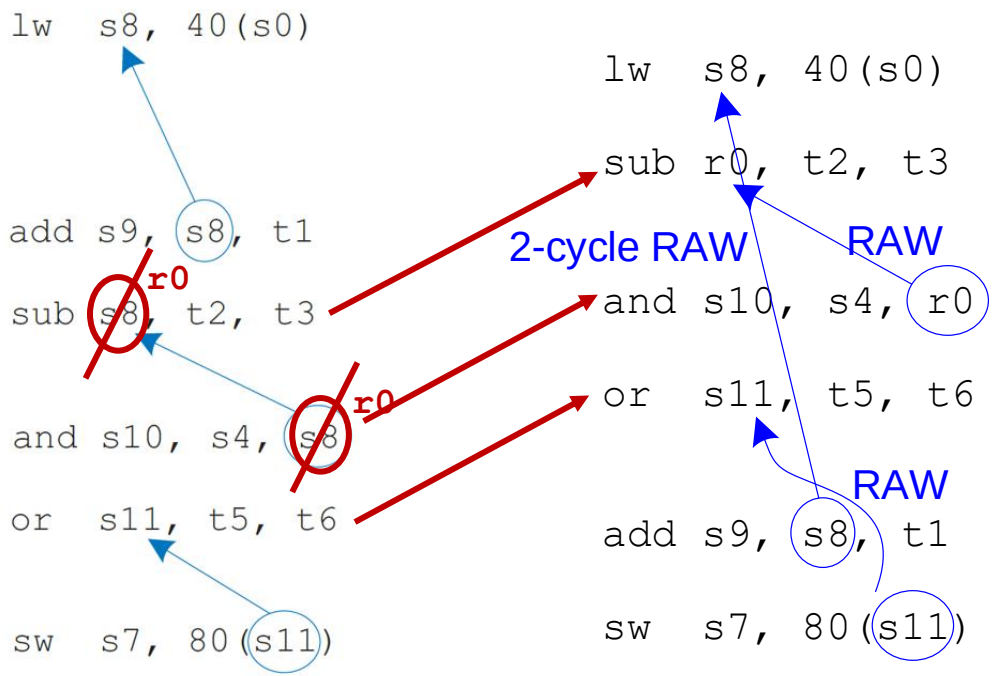


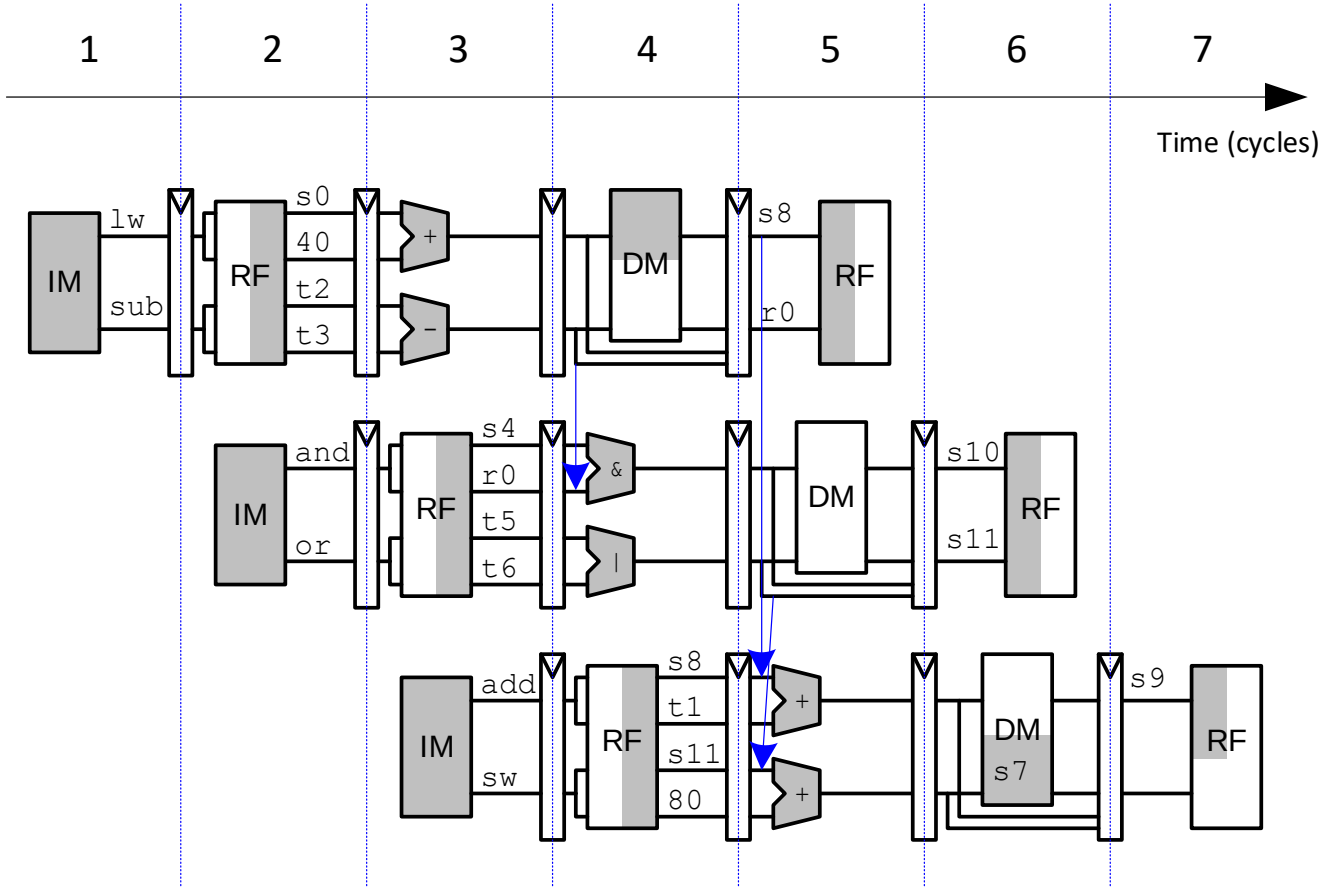


Simple Example for Register Renaming

- Assume the CPU internally has additional registers r0 … r20
- We use r0 to resolve the WAR dependence of our example program

The Program





Reorder Buffer (ROB)

- The reorder buffer is the hardware unit that stores the additional register contents
- High-Level Working principle:
  - When an instruction is decoded, it reserves an entry in the reorder buffer
  - After completion of the instruction, the reorder buffer stores the result of an instruction
  - When the oldest instruction in the reorder buffer has completed, the result of the instruction is moved to the register file

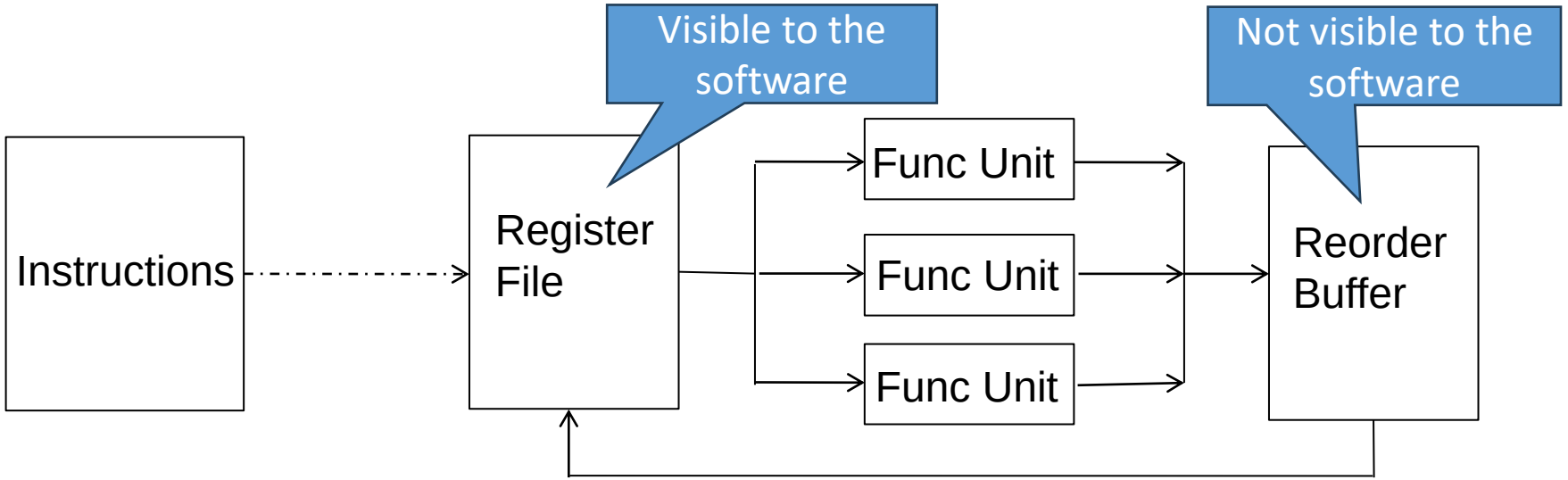


Figure by Onur Mutlu

Reorder Buffer (ROB)

- The ROB stores information about all instructions that are decoded but not yet retired/committed
- It uses valid bits to keep track of which instructions have completed executions (i.e. which results are already valid)
- It needs to store additional meta data to:
  - correctly reorder instructions back into program order
  - update the architectural state with the instruction's result(s) upon retiring
  - handle exceptions/interrupts

Reading Inputs from the Reorder Buffer

- Observe:
  - Instructions my need registers as input that are still located in the re-order buffer (results that are not committed to the register file yet)
- Idea:
  - Read input registers for instructions from the register file or directly from the reorder buffer
- Example Implementation:
  - Remember scoreboarding: We set the valid bit of the target register of an instruction as invalid during the decode stage. It remains invalid until the result is written to the register file
  - In addition to the valid bit, we store the ID of the register of reorder buffer in the register file that will store the result of the instruction. This register stores the result until it is committed.
  - Accessing the register inputs.
    - First access the register file
    - In case the register is not valid, use the ID stored in the register file to access the reorder buffer
    - Read data from the reorder buffer; delay the instruction in case the result is not ready yet

Summary

- Pipelining is the standard approach for building processors
- Large CPUs (servers, laptops, mobile phones, ...) all use superscalar designs with out-of-order execution
- For further details, interested readers are referred to “Computer Architecture - A quantitative Approach” by Hennessy, Patterson