

---

# Chapter 4: The Processor

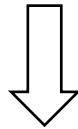
Ngo Lam Trung

[with materials from *Computer Organization and Design, 4<sup>th</sup> Edition*,  
Patterson & Hennessy, © 2008, MK  
and M.J. Irwin's presentation, PSU 2008]

# Review

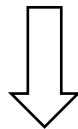
High-level  
language  
program  
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```



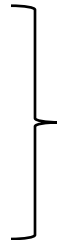
Assembly  
language  
program  
(for MIPS)

```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```



Binary machine  
language  
program  
(for MIPS)

```
000000001010000100000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
100011001111001000000000000000100
10101100111100100000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```



Performance metric

$$\text{CPU time} = \text{CPI} * \text{CC} * \text{IC}$$

CPI: cycle per instruction

CC: clock cycle

IC: instruction count

How to improve?

- IC:
- CC:
- CPI:

In this chapter

- Implementation of data path
- How to get  $\text{CPI} < 1$

# Overview

---

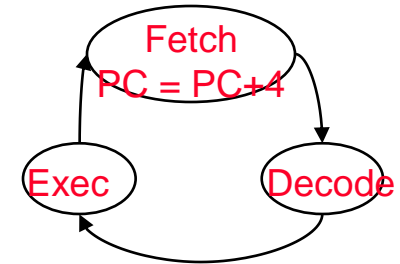
- ❑ We will examine two MIPS implementations
  - ❑ A simplified version
  - ❑ A more realistic pipelined version
- ❑ Limit to a simple subset of MIPS ISA
  - ❑ Memory reference: lw, sw
  - ❑ Arithmetic/logical: add, sub, and, or, slt
  - ❑ Control transfer: beq, j
- ❑ Implementation of real CPU with other instructions are similar to the simplified version (theoretically!)

# General instruction cycle

---

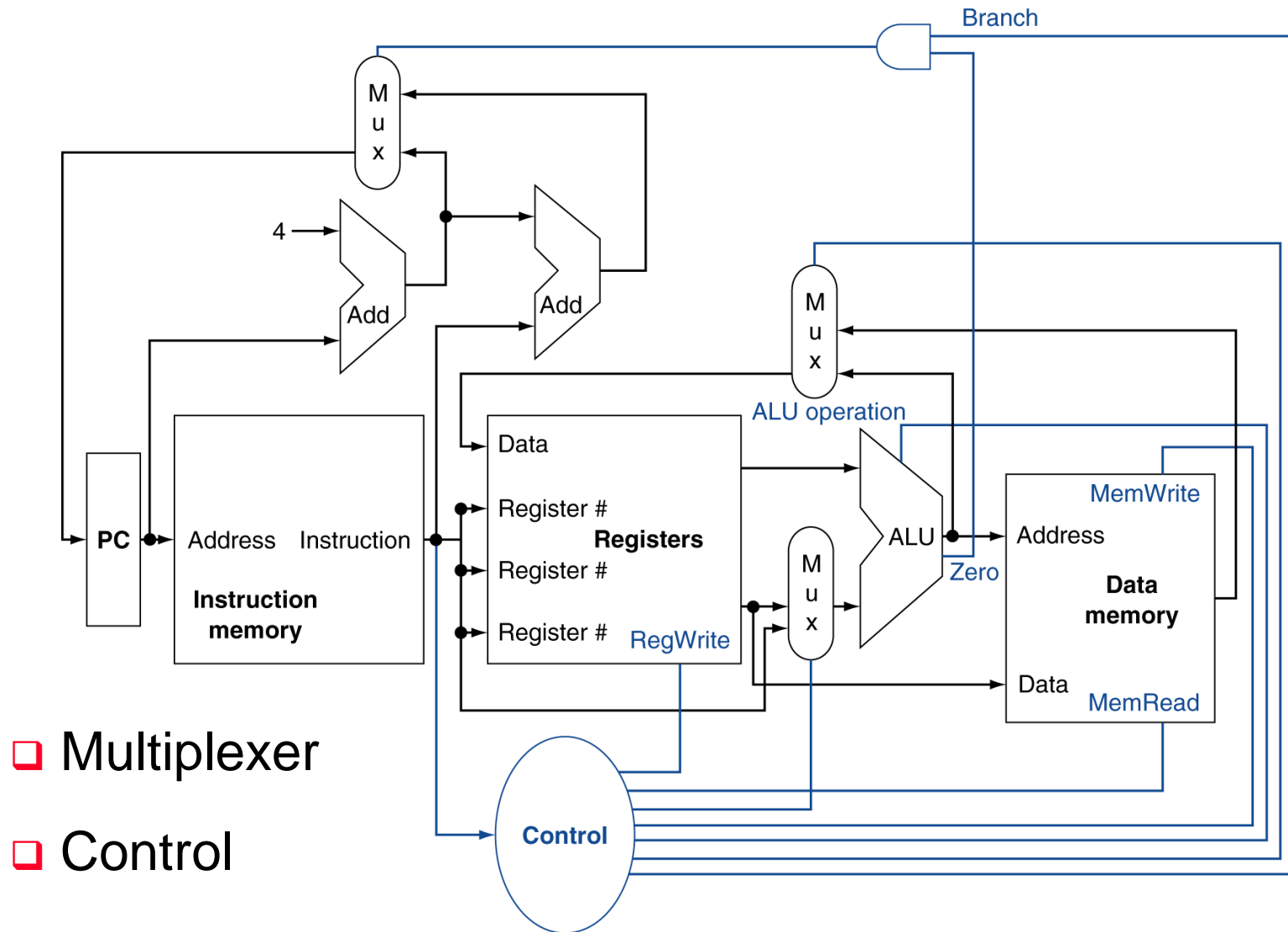
## ❑ Generic implementation

- ❑ use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
- ❑ decode the instruction (and read registers)
- ❑ execute the instruction



- ❑ All instructions (except **j**) use the ALU after reading the registers
  - ❑ ALU: Arithmetic and Logic Unit, where the arithmetic and logic operations are executed
- ❑ In this chapter: implementation of CPU that can execute the simple subset of MIPS ISA

# CPU implementation with MUXes and Control



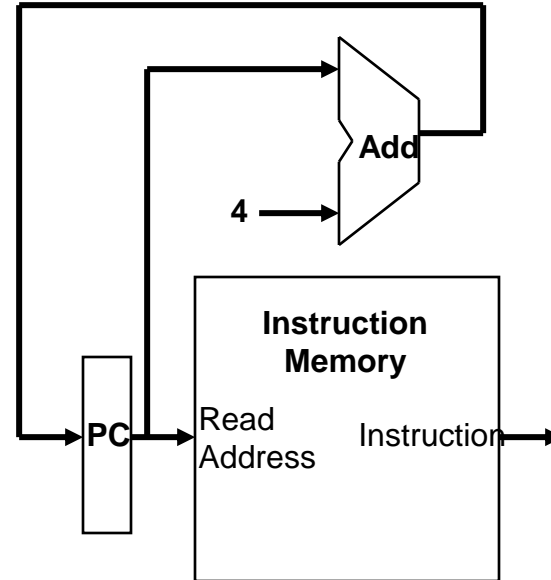
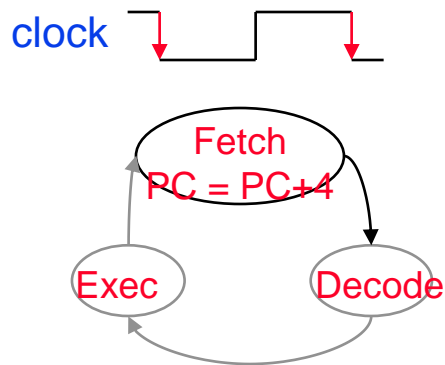
❑ Multiplexer

❑ Control

*Don't panic! We'll build this incrementally.*

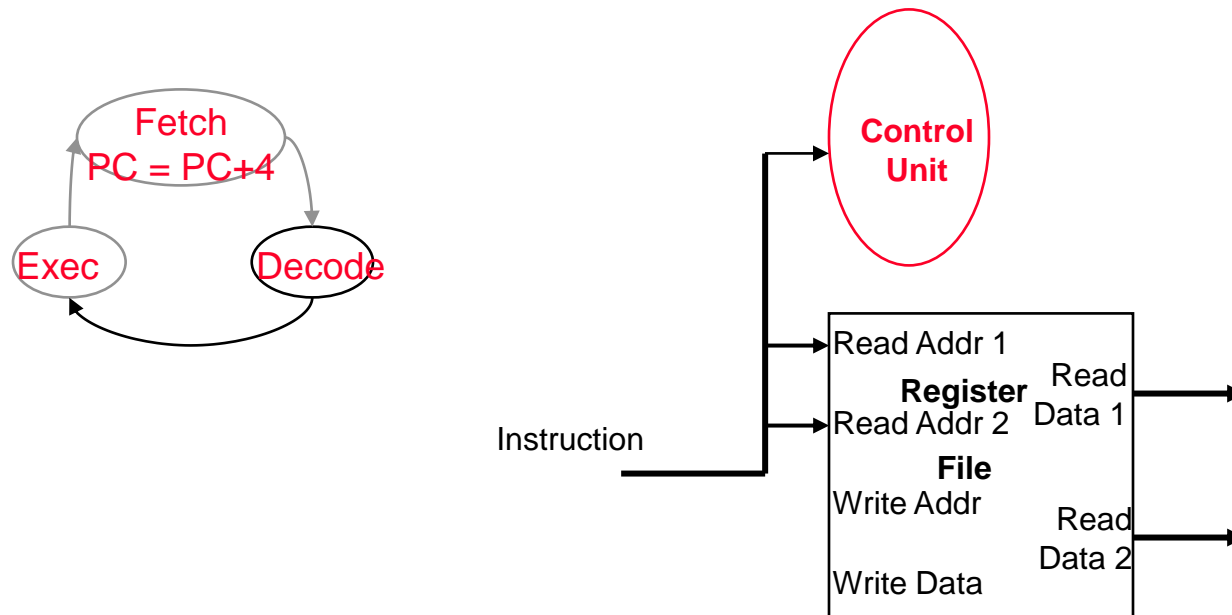
# Fetching Instructions

- ❑ Fetching instructions involves
  - ❑ reading the instruction from the Instruction Memory
  - ❑ updating the PC value to be the address of the next instruction in memory



# Decoding Instructions

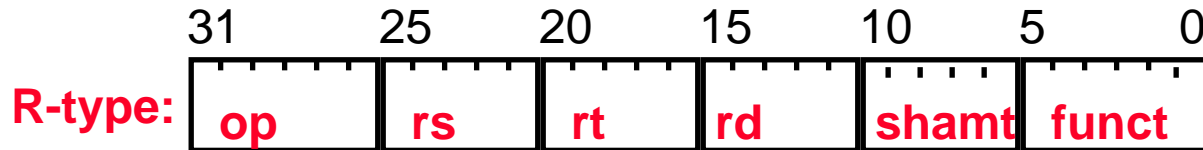
- ❑ Decoding instructions involves
  - ❑ sending the fetched instruction's opcode and function field bits to the control unit
  - ❑ The control unit send appropriate control signals to other parts inside CPU to execute the operations corresponds to the instruction



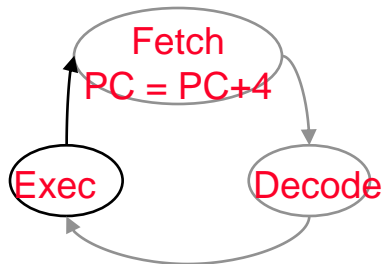
- Example: reading two values from the Register File  
→ Register File addresses are contained in the instruction

# Executing R Format Operations

□ R format operations (**add**, **sub**, **slt**, **and**, **or**)



- read two register operands **rs** and **rt**
- perform operation (**op** and **funct**) on values in **rs** and **rt**
- store the result back into the Register File (into location **rd**)



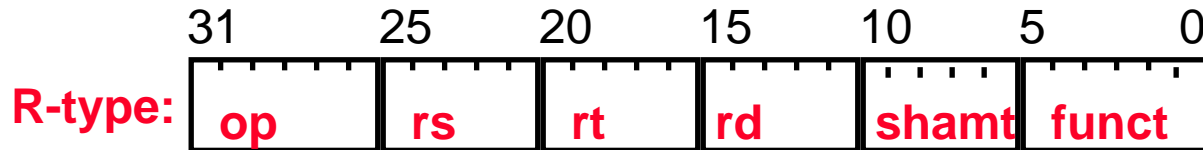
Example: **add s1, s2, s3**

- Value of s2 and s3 are sent to ALU
- ALU execute the s2 + s3 operation
- Result is store into s1

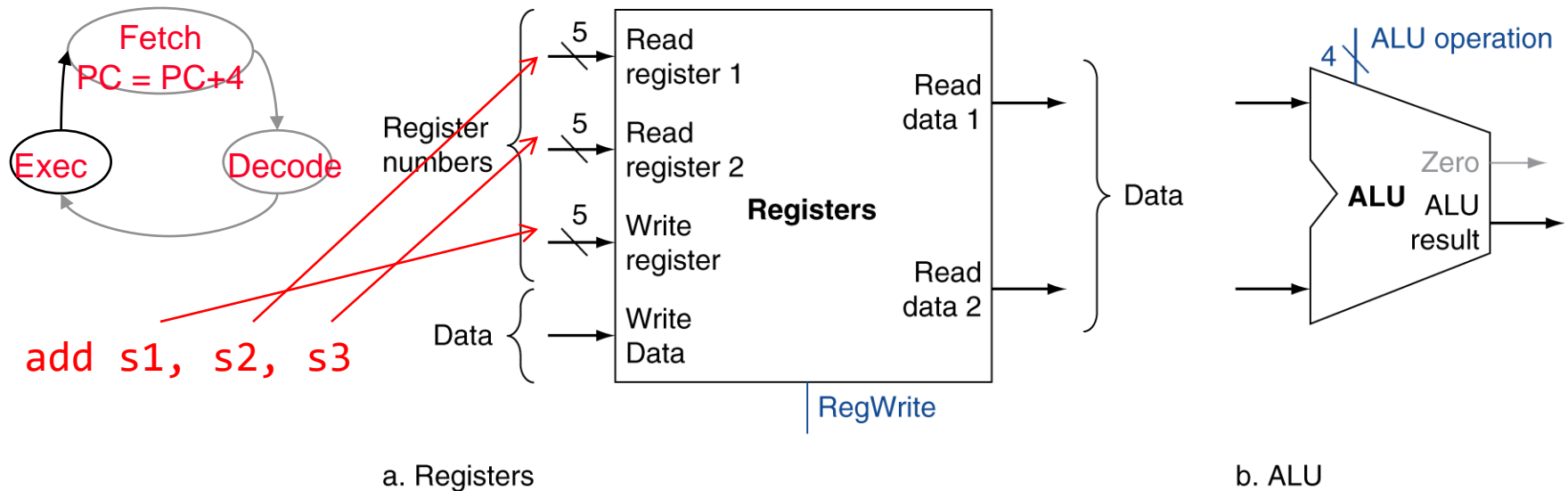


# Executing R Format Operations

□ R format operations (**add**, **sub**, **slt**, **and**, **or**)



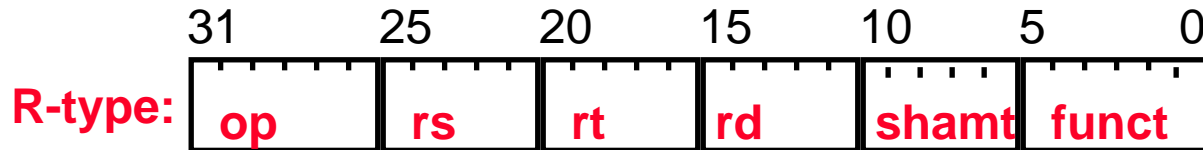
- read two register operands **rs** and **rt**
- perform operation (**op** and **funct**) on values in **rs** and **rt**
- store the result back into the Register File (into location **rd**)



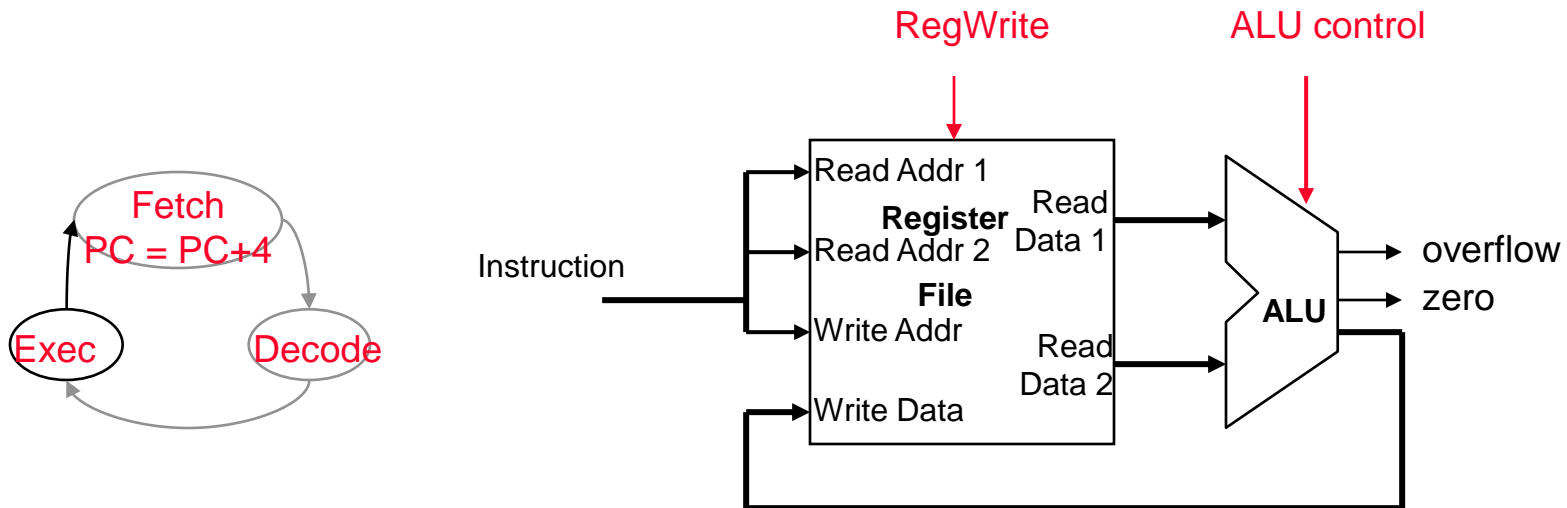
Draw connection between a and b to form the execution unit?

# Executing R Format Operations

- R format operations (**add**, **sub**, **slt**, **and**, **or**)



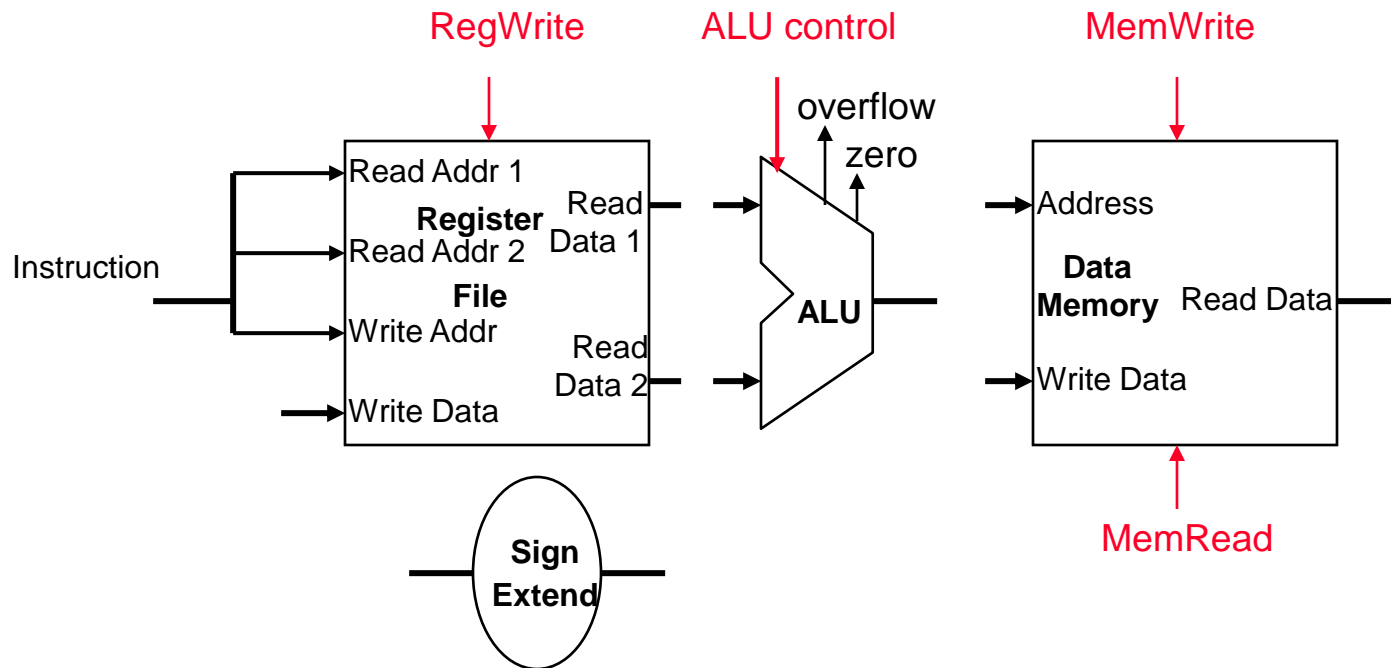
- read two register operands **rs** and **rt**
- perform operation (**op** and **funct**) on values in **rs** and **rt**
- store the result back into the Register File (into location **rd**)



- We need the **write control signal** to control when the result is written to Register File

# Executing Load and Store Operations

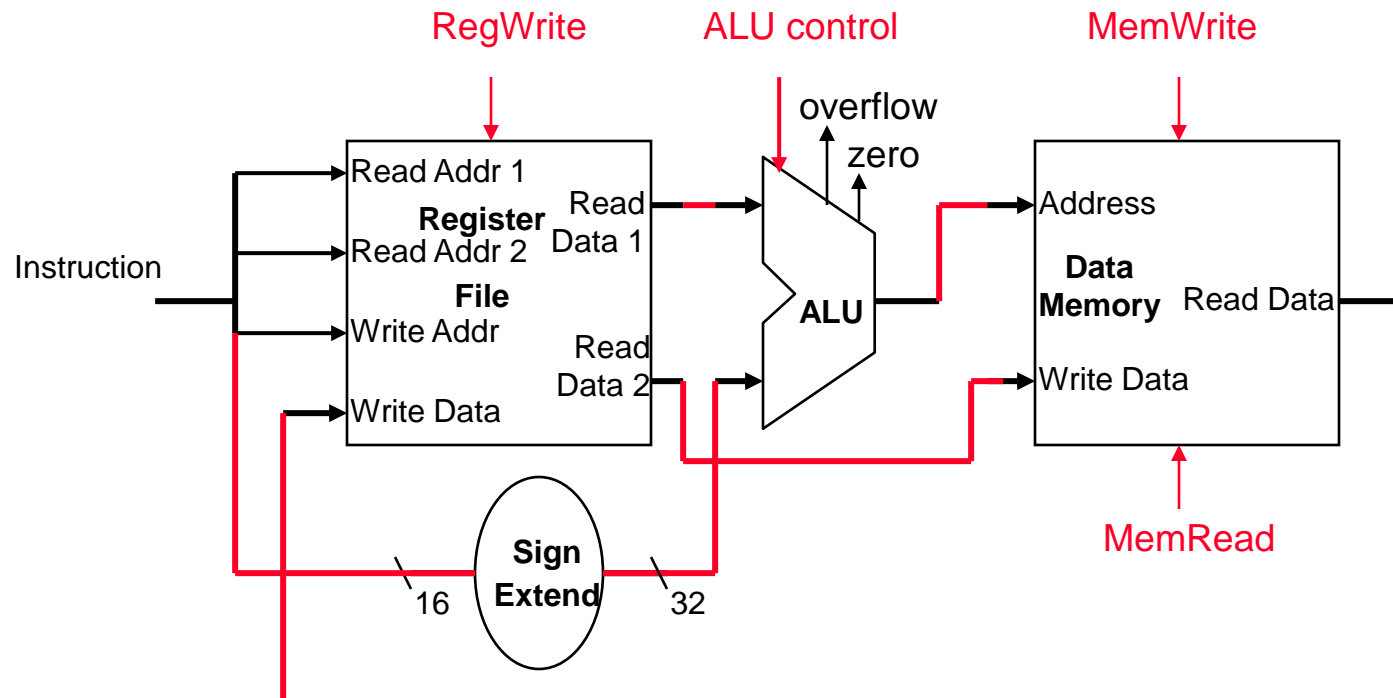
- ❑ Load and store operations involves
  - ❑ read register operands (including one base register)
  - ❑ compute memory address by adding the base to the offset
    - The 16-bit offset field in the instruction is signed-extended to 32 bit
  - ❑ **store**: read from the Register File, write to the Data Memory
  - ❑ **load**: read from the Data Memory, write to the Register File



Draw necessary connections to form execution unit?

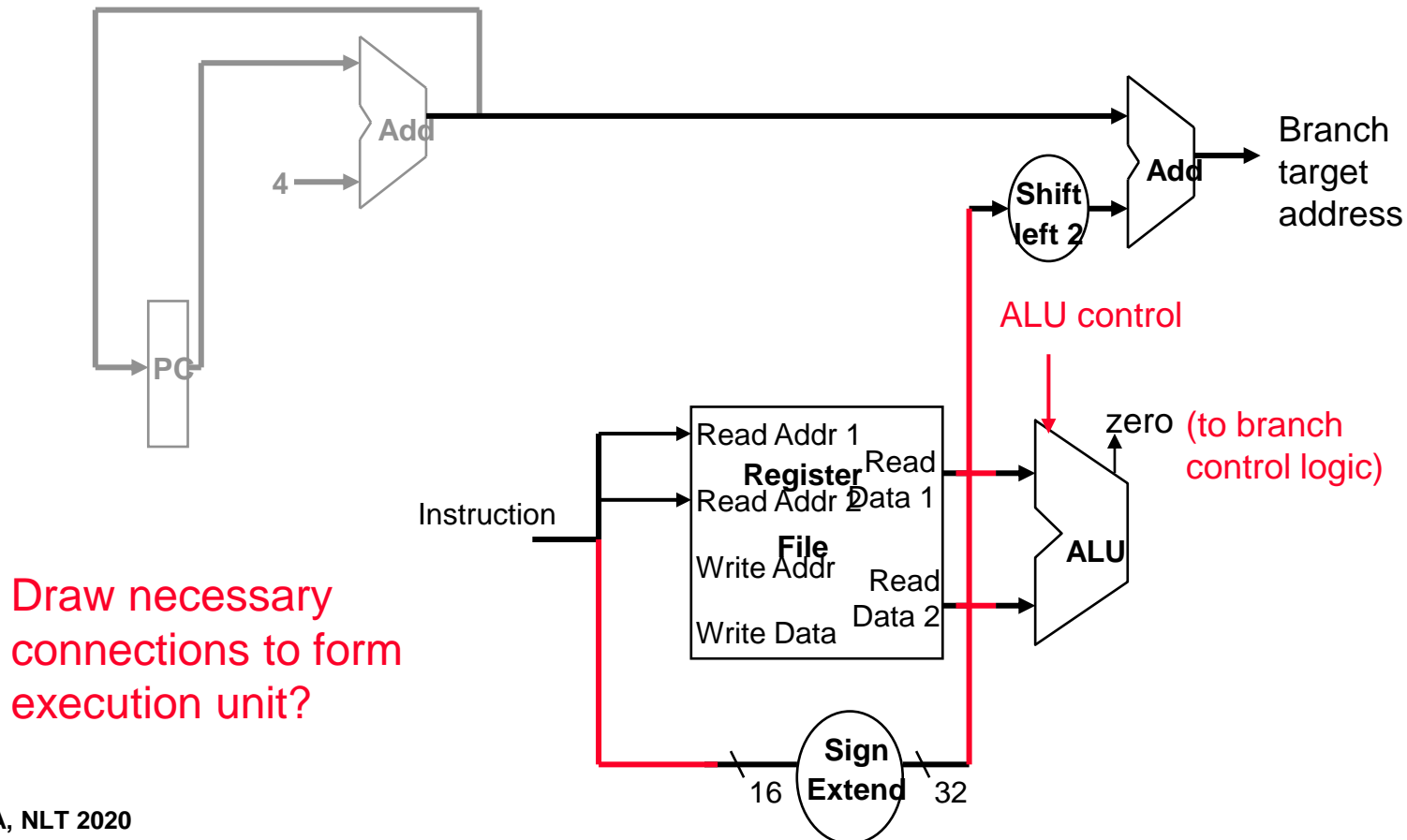
# Executing Load and Store Operations

- ❑ Load and store operations involves
  - ❑ read register operands (including one base register)
  - ❑ compute memory address by adding the base to the offset
    - The 16-bit offset field in the instruction is signed-extended to 32 bit
  - ❑ **store**: read from the Register File, write to the Data Memory
  - ❑ **load**: read from the Data Memory, write to the Register File



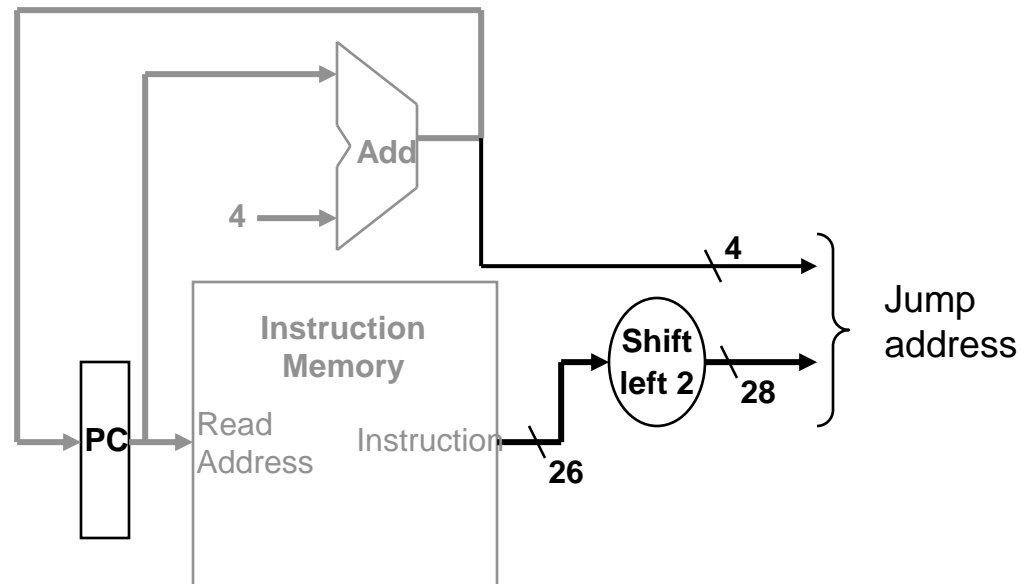
# Executing Branch Operations

- ❑ Branch operations involves
  - ❑ read register operands
  - ❑ compare the operands (subtract, check **zero** ALU output)
  - ❑ compute the branch target address: adding the updated PC to the 16-bit signed-extended offset field in the instr



# Executing Jump Operations

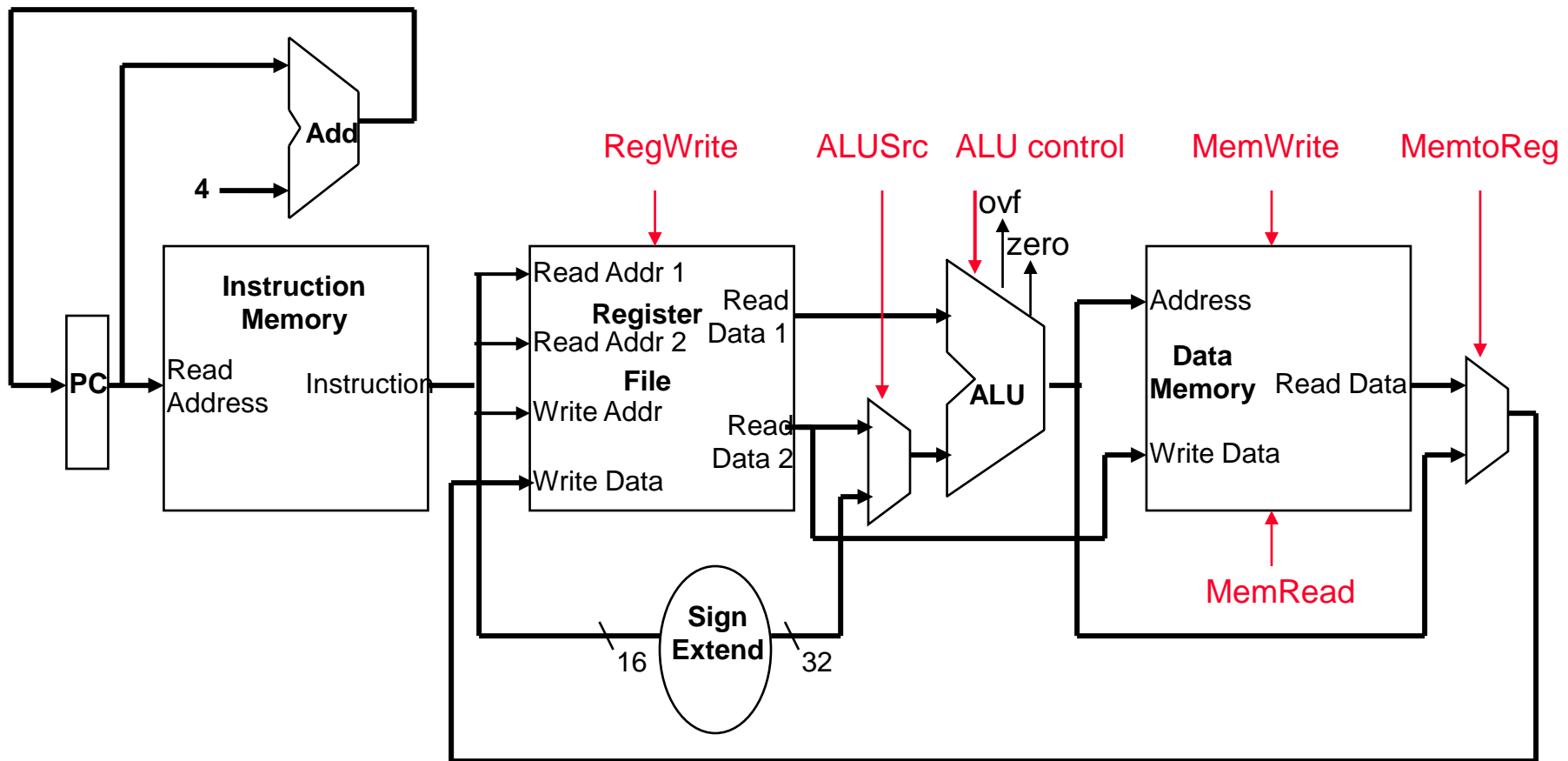
- ❑ Jump operation involves
  - ❑ keep 4 highest bits of PC
  - ❑ replace the lower 28 bits of the PC by
    - the lower 26 bits of the fetched instruction shifted left by 2 bits



# Creating a Single Datapath from the Parts

- ❑ Assemble the datapath segments and add control lines and multiplexors as needed
- ❑ **Single cycle** design – fetch, decode and execute each instructions in **one** clock cycle
  - ❑ separate Instruction Memory and Data Memory, though they are both in main memory
  - ❑ **multiplexors** needed at the input of shared elements with control lines to do the selection
  - ❑ write signals to control writing to the Register File and Data Memory

# Fetch, R, and Memory Access Portions





# Designing ALU

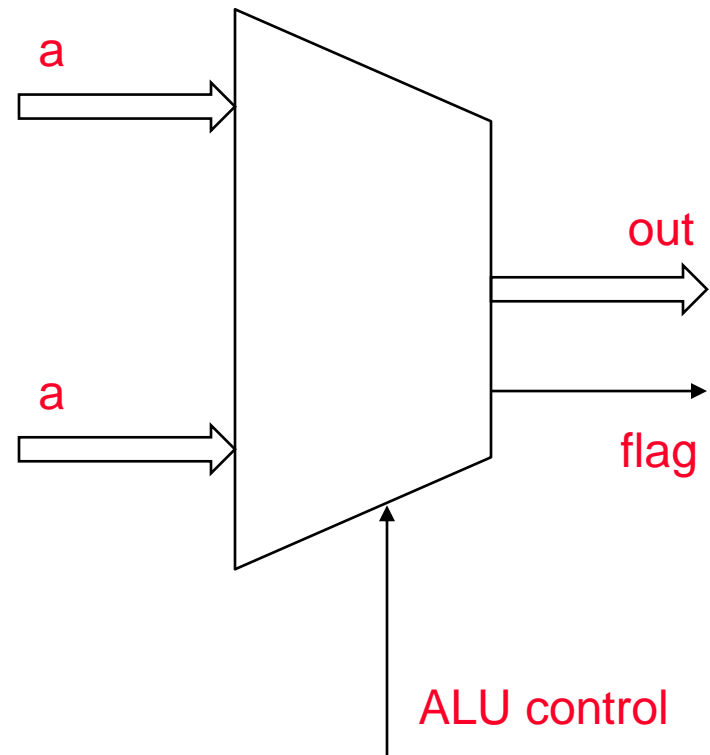
---

## ❑ Input/output

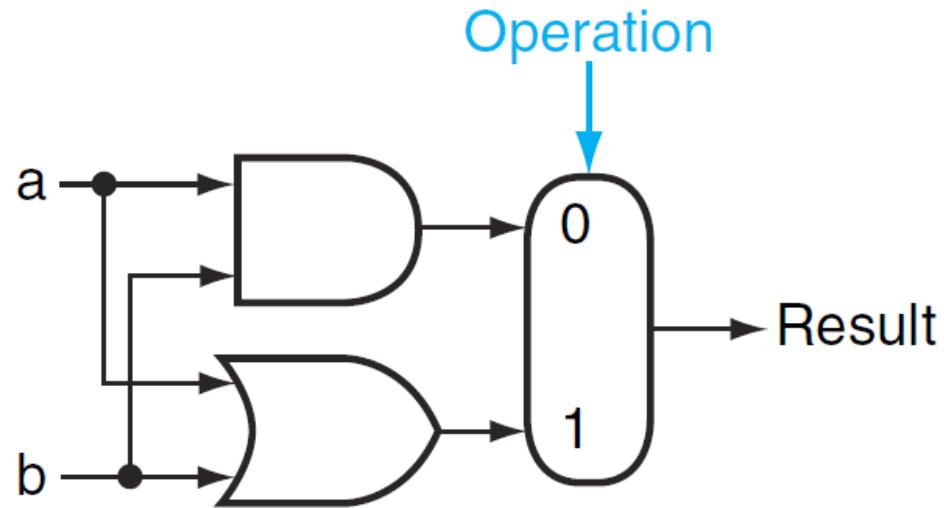
- ❑ Two data input: a, b
- ❑ ALU control signal
- ❑ Data out
- ❑ Flags out

## ❑ Operations

- ❑ And, or, nor
- ❑ Add, subtract



# 1-bit ALU with logic operation

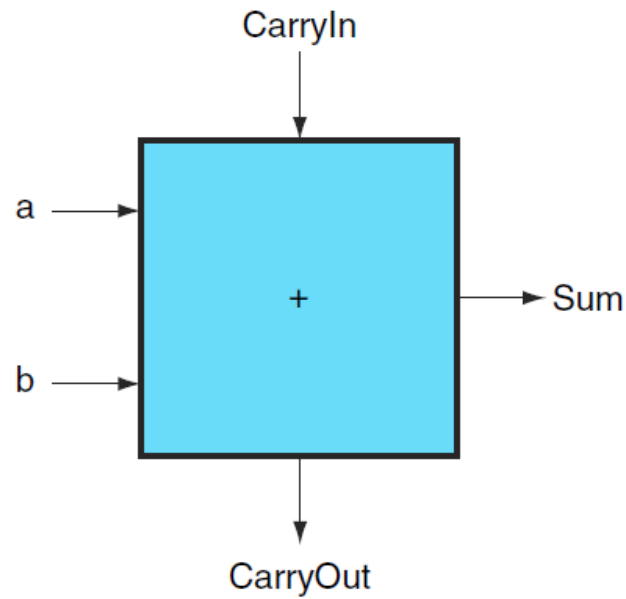


❑ What do we have if

❑ Operation = 0:

❑ Operation = 1:

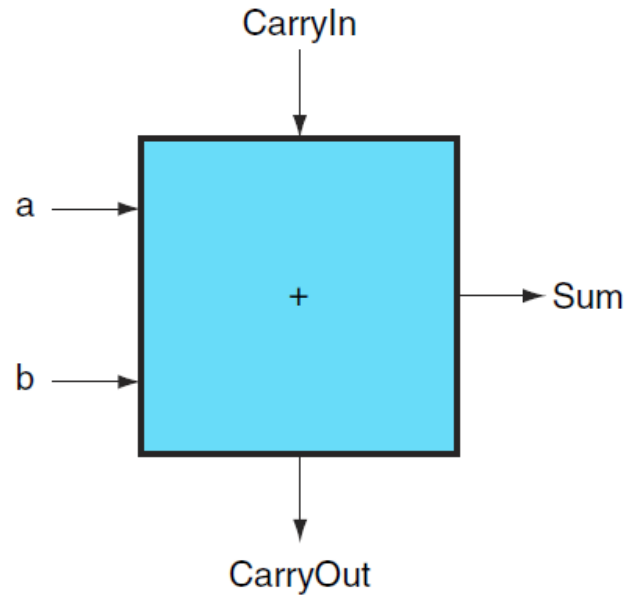
# 1-bit full-adder



Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

# 1-bit full-adder

---

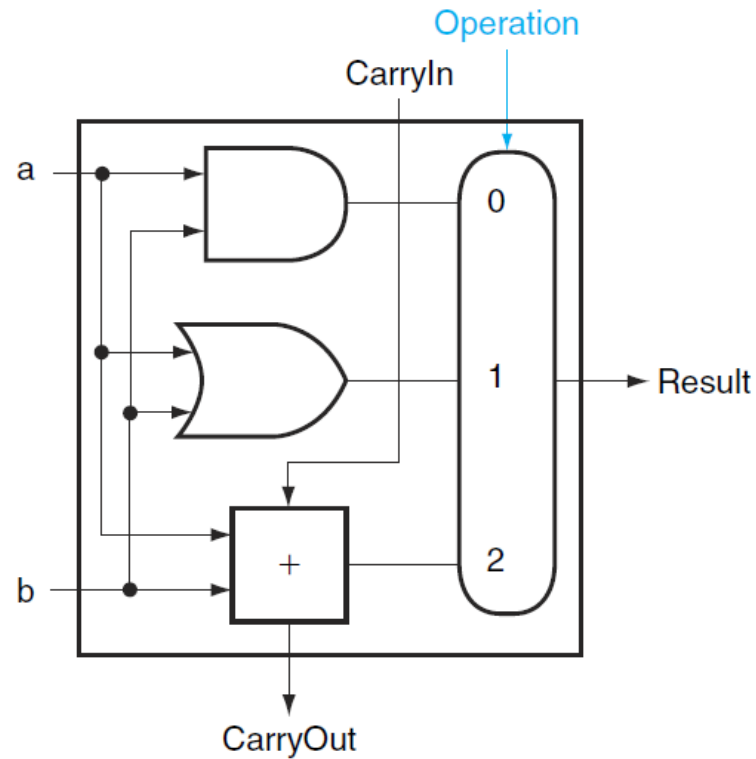


$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

➔ Combinational circuit for full-adder can be constructed

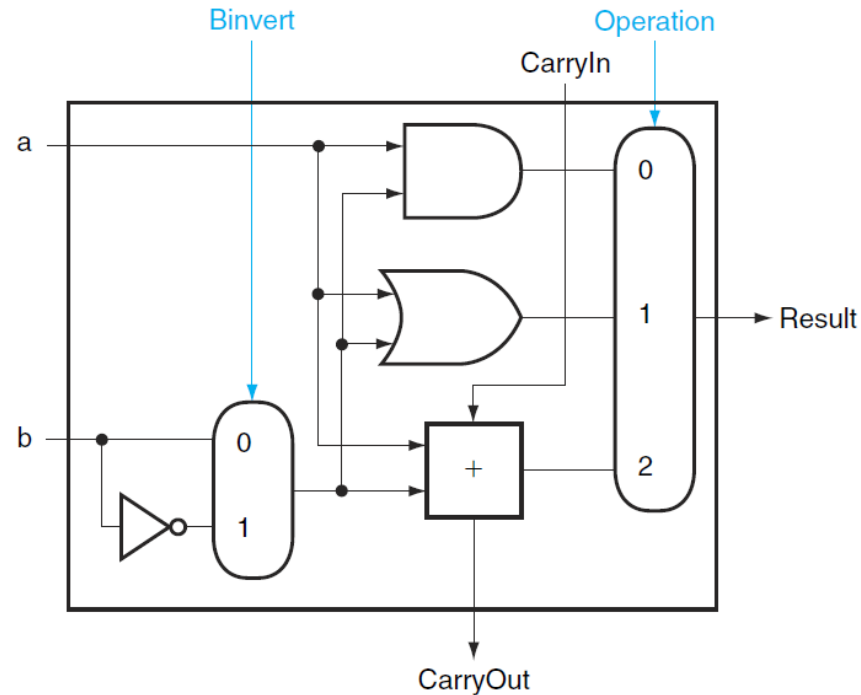
# 1-bit ALU with AND, OR, ADD



- ❑ Operation = 00:
- ❑ Operation = 01:
- ❑ Operation = 10:

# How about 1-bit ALU with AND, OR, ADD, SUB?

□  $a - b = a + (-b) = a + (2\text{'s complement of } b)$



□ For SUB operation

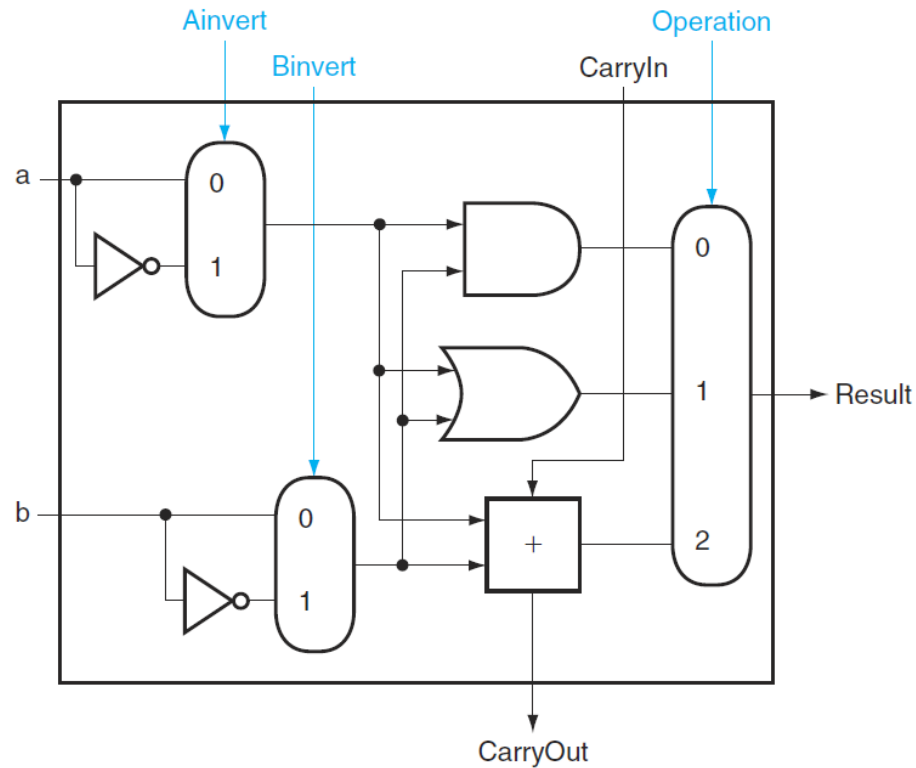
□ Operation =

□ Binvert =

□ CarryIn =

# How to add NOR operation?

□  $\overline{a + b} = \bar{a} \cdot \bar{b}$

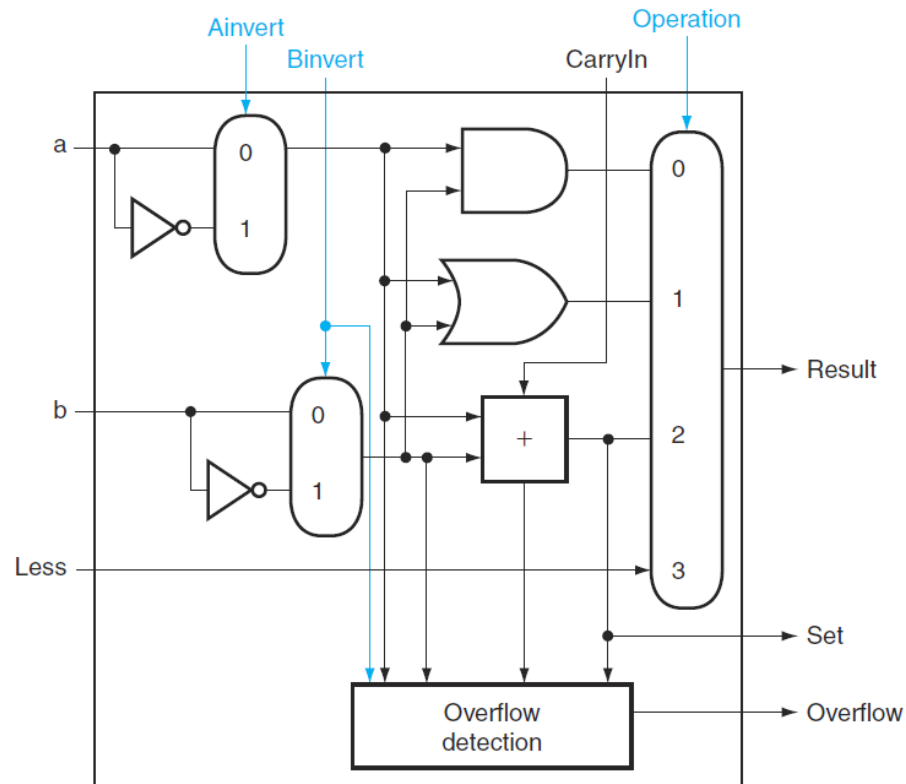


□ Find control signal for NOR operation:

# Adding SLT operation

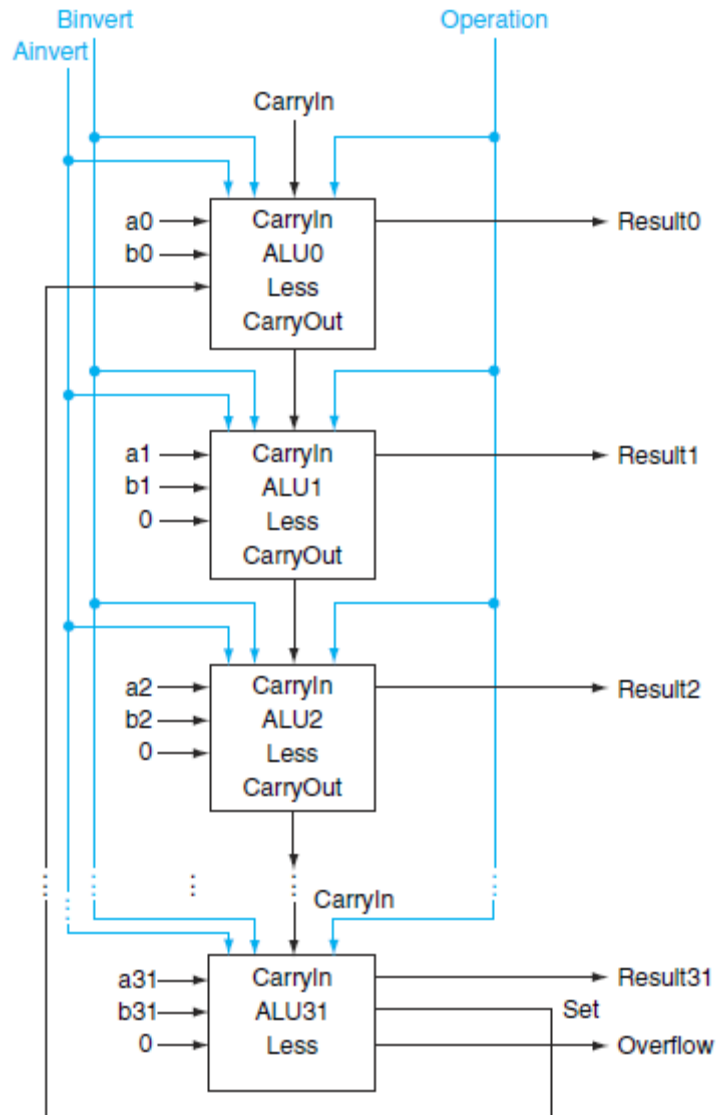
## ❑ Check highest bit of (a-b)

- ❑ 1:  $a < b \rightarrow$  set
- ❑ 0:  $a \geq b \rightarrow$  clear

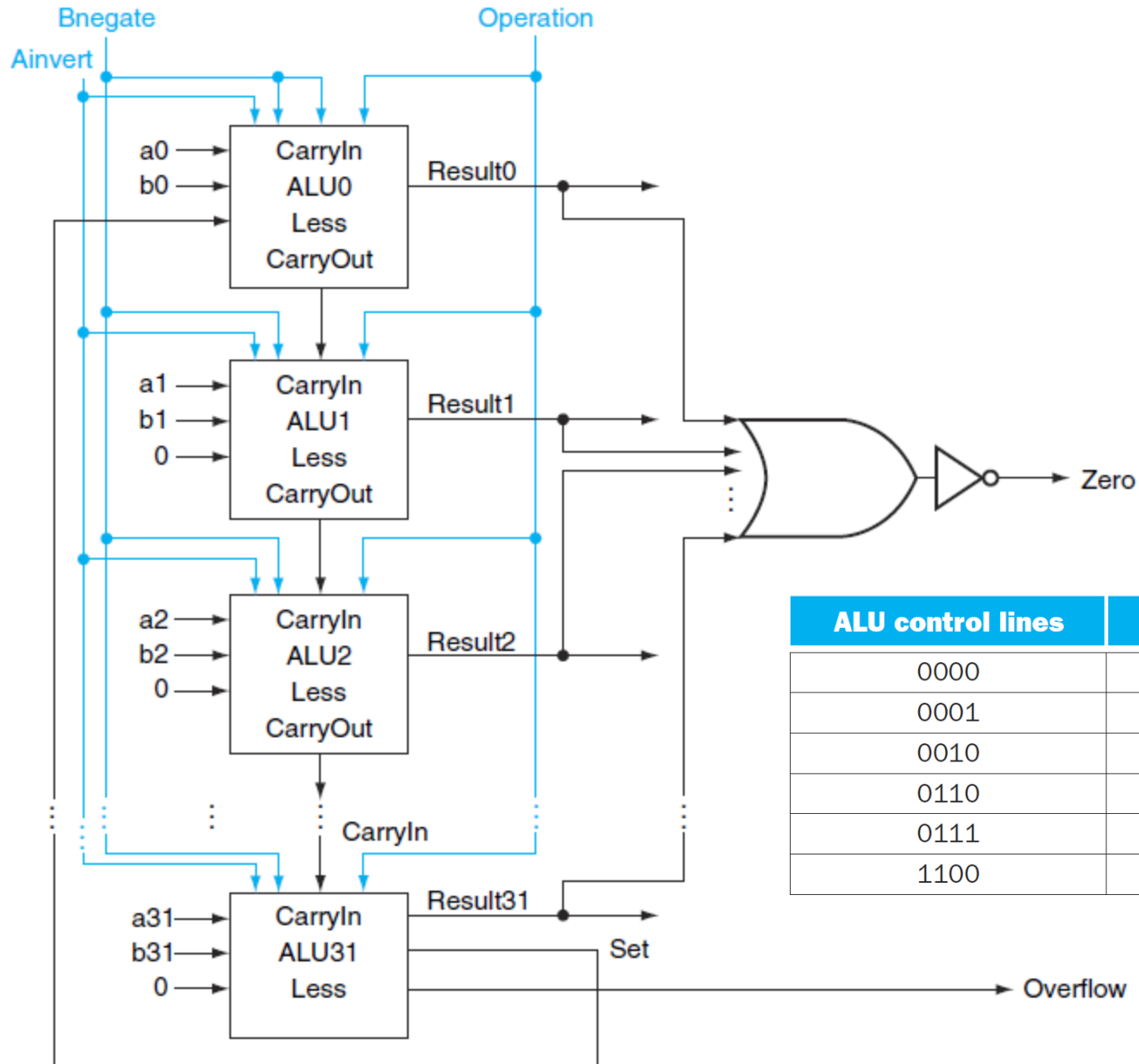




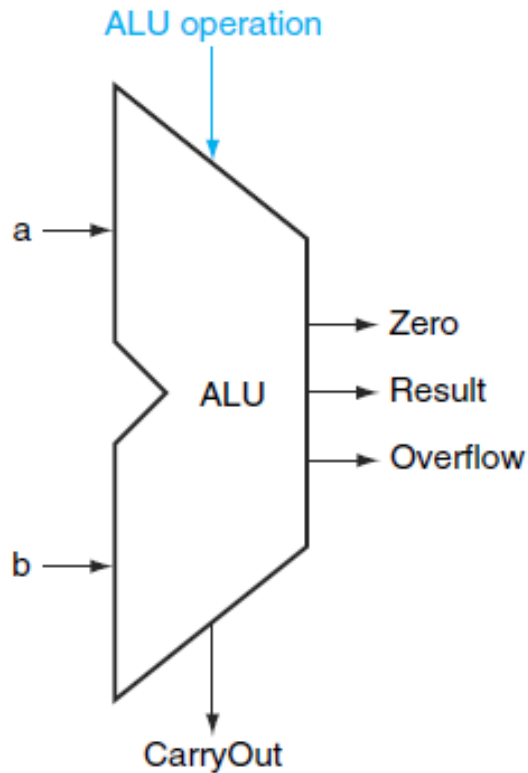
# Building 32-bit ALU from 1-bit ALUs



# Final ALU with AND, OR, NOR, ADD, SUB, SLT

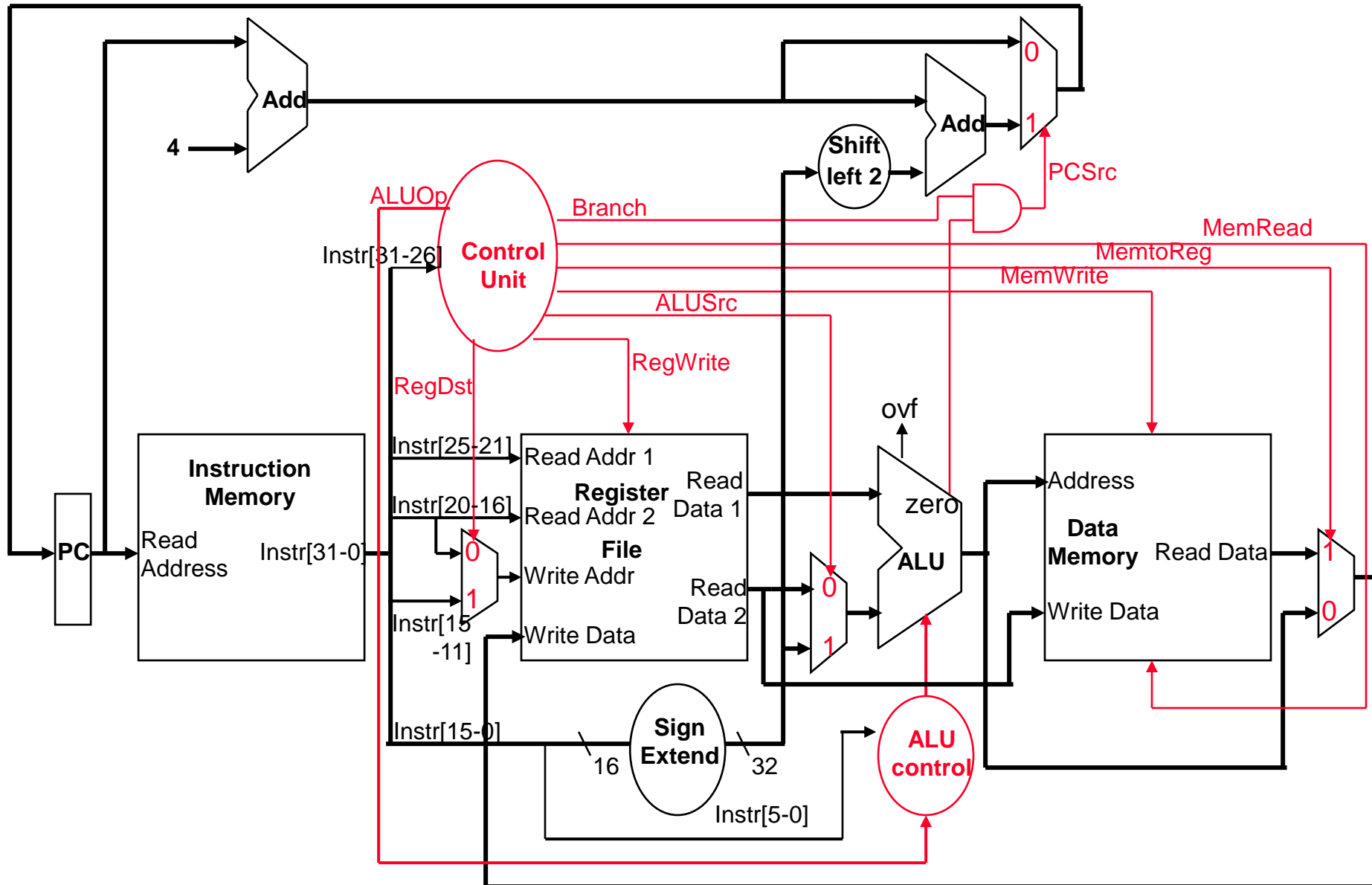


# ALU symbol and interconnection

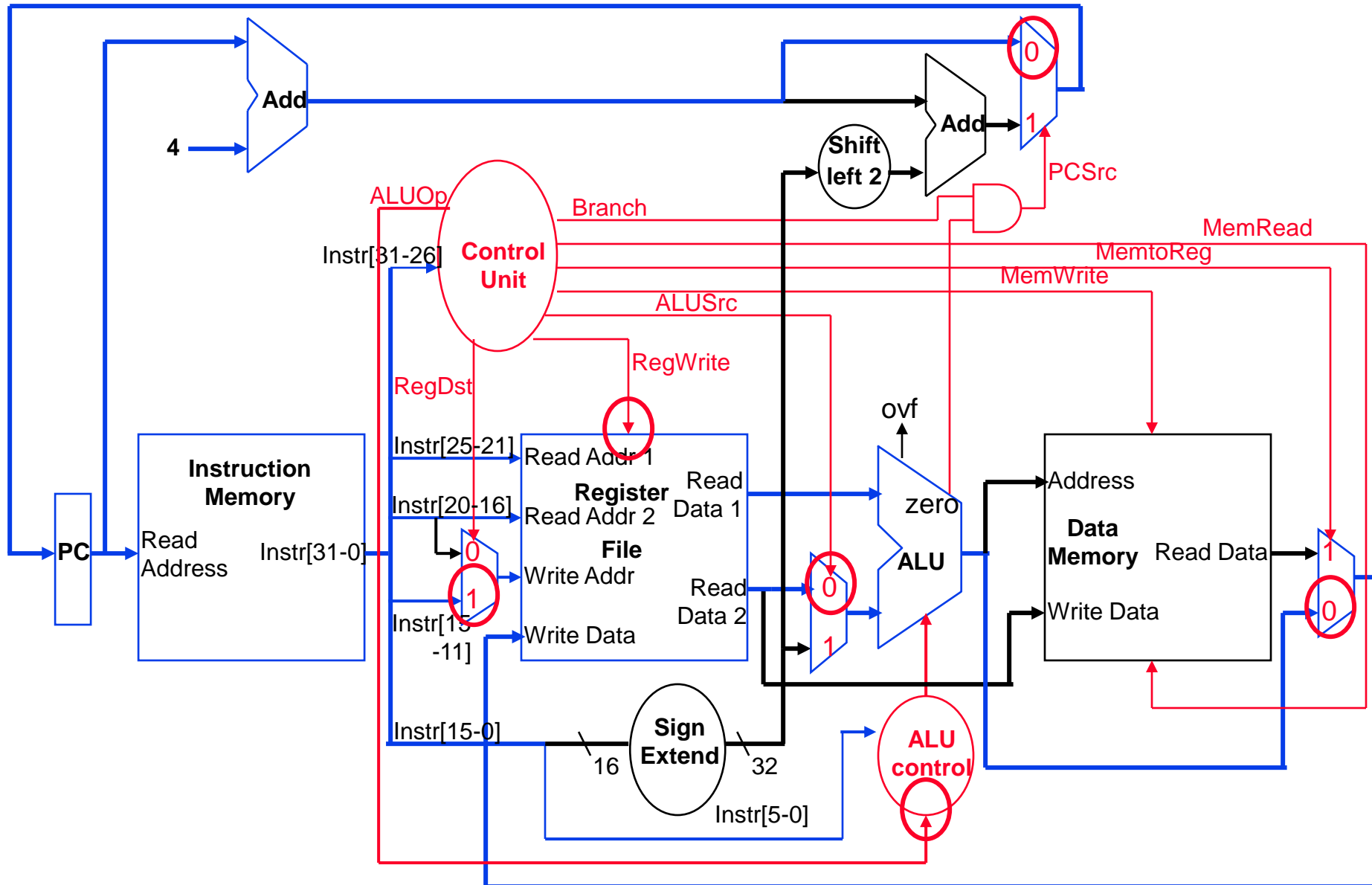


ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

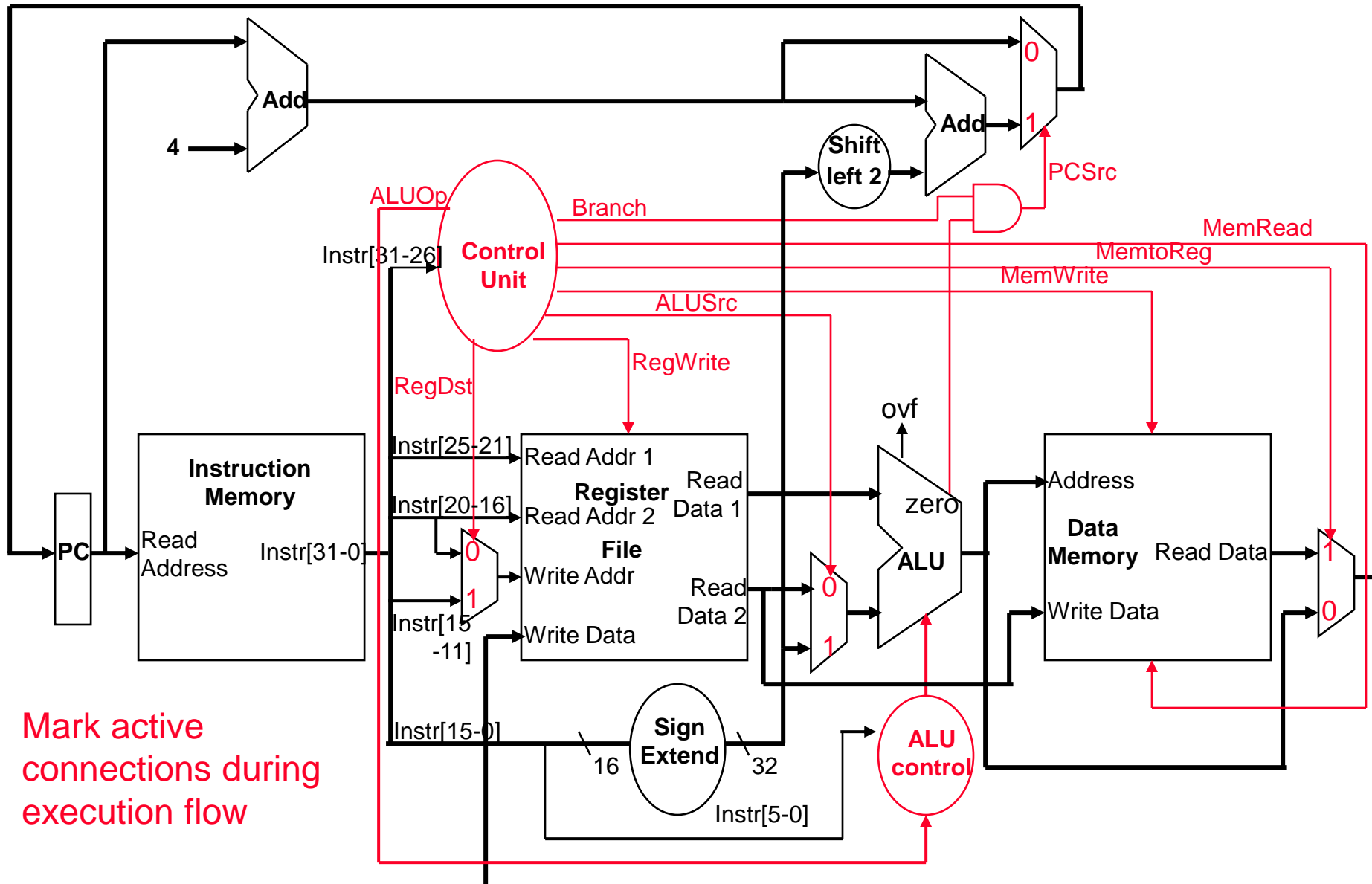
# Adding Control Unit to build single datapath



# R-type Instruction Data/Control Flow

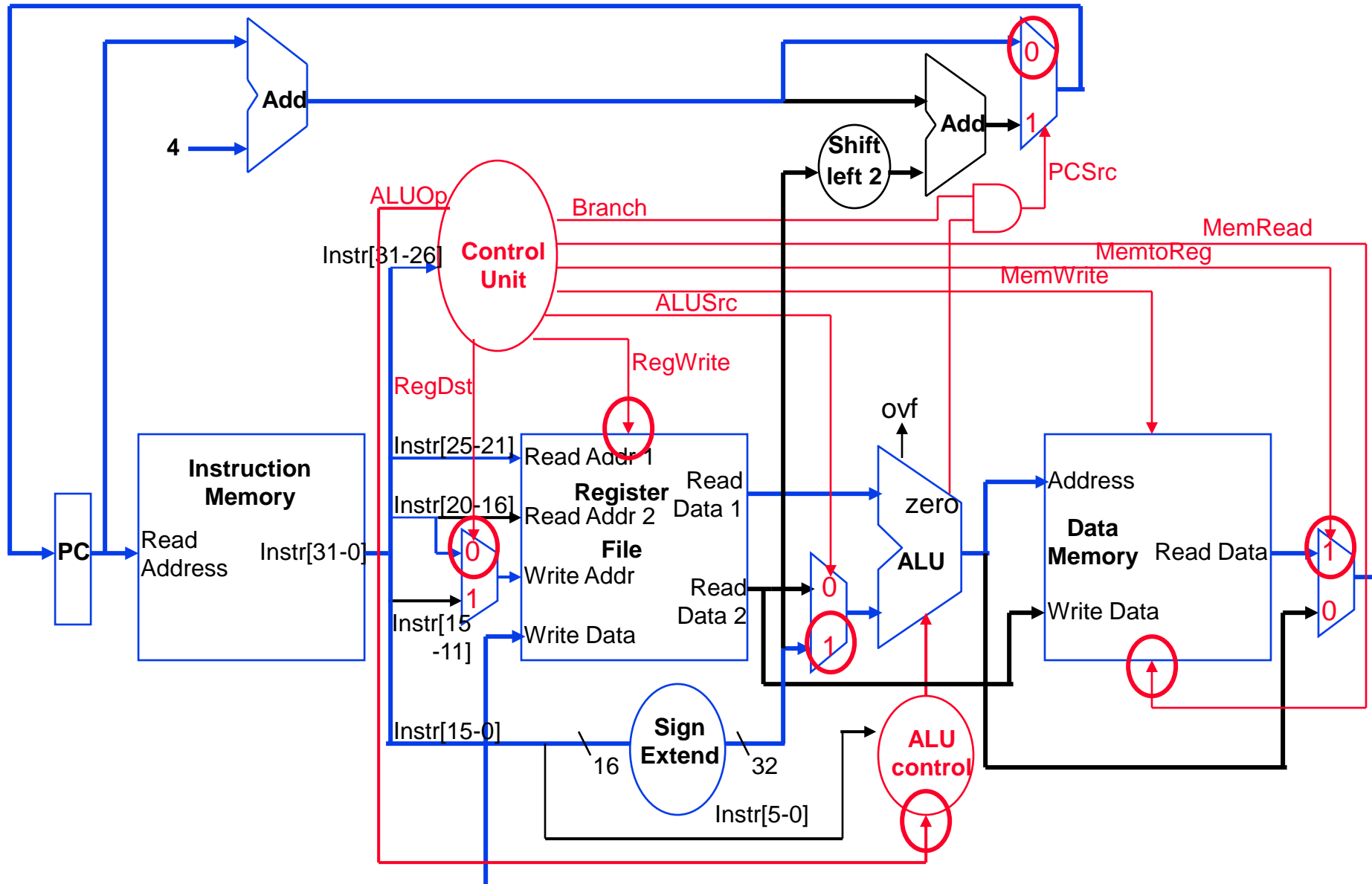


# Load Word Instruction Data/Control Flow

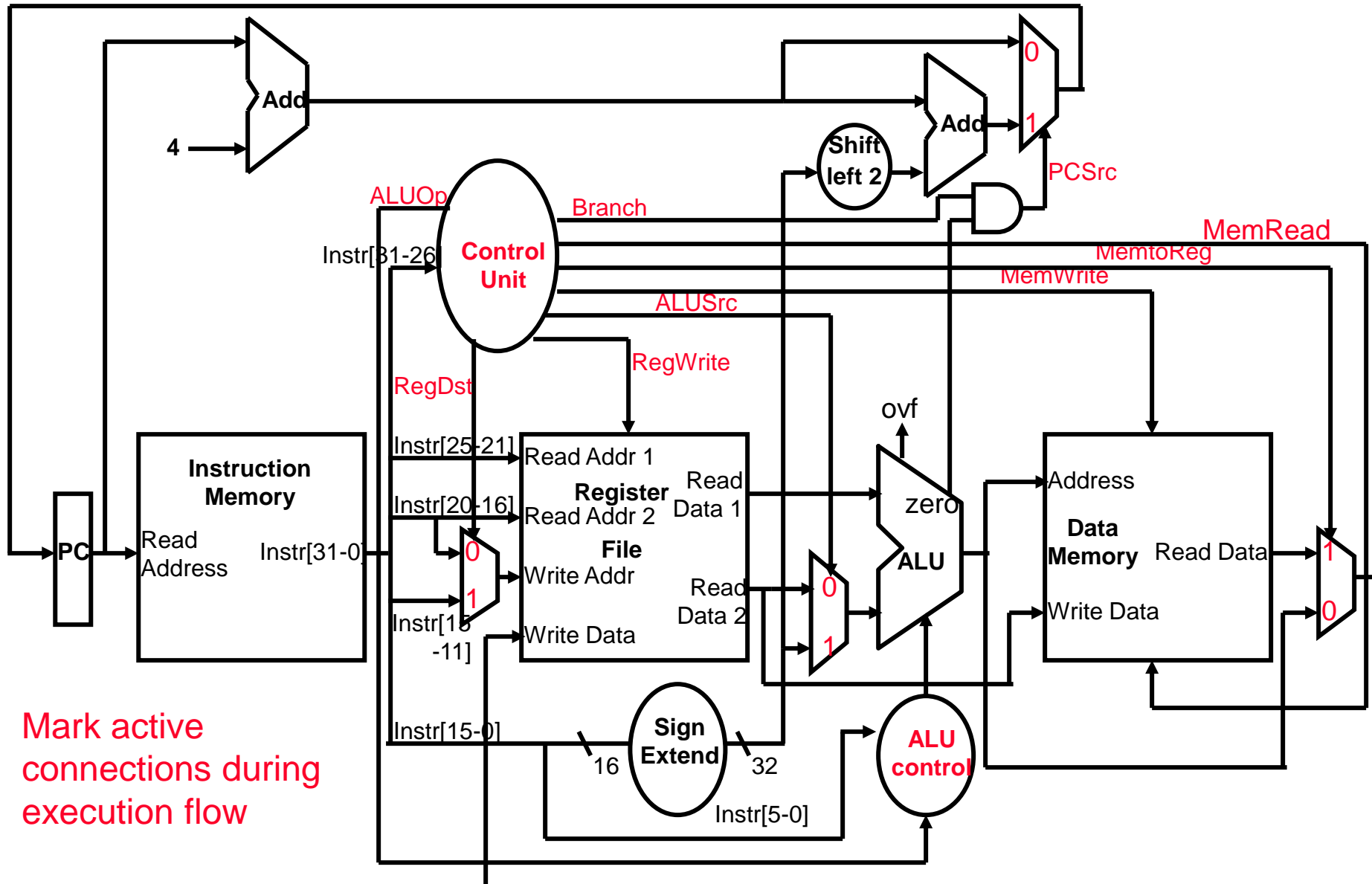


Mark active connections during execution flow

# Load Word Instruction Data/Control Flow

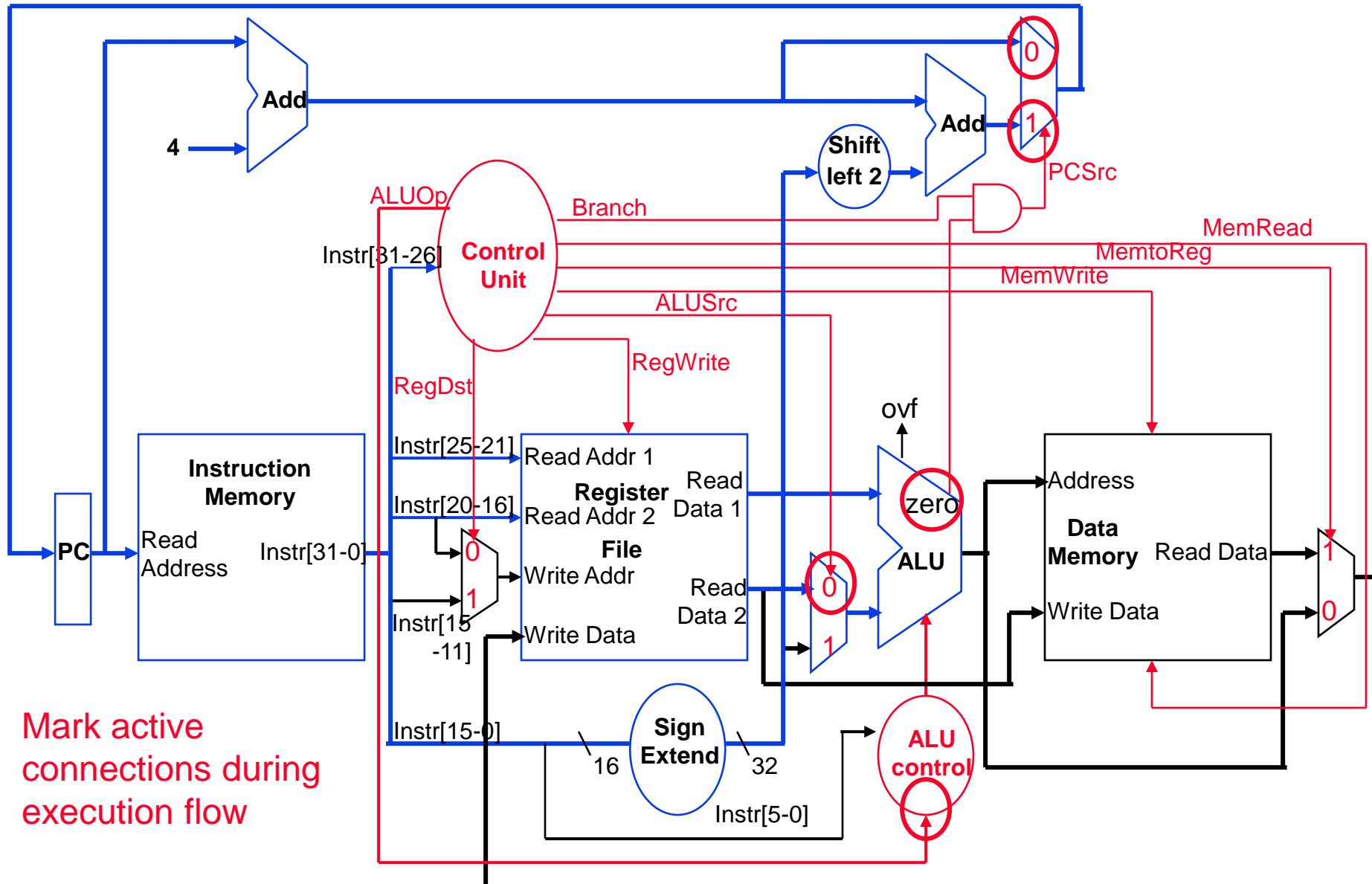


# Branch Instruction Data/Control Flow



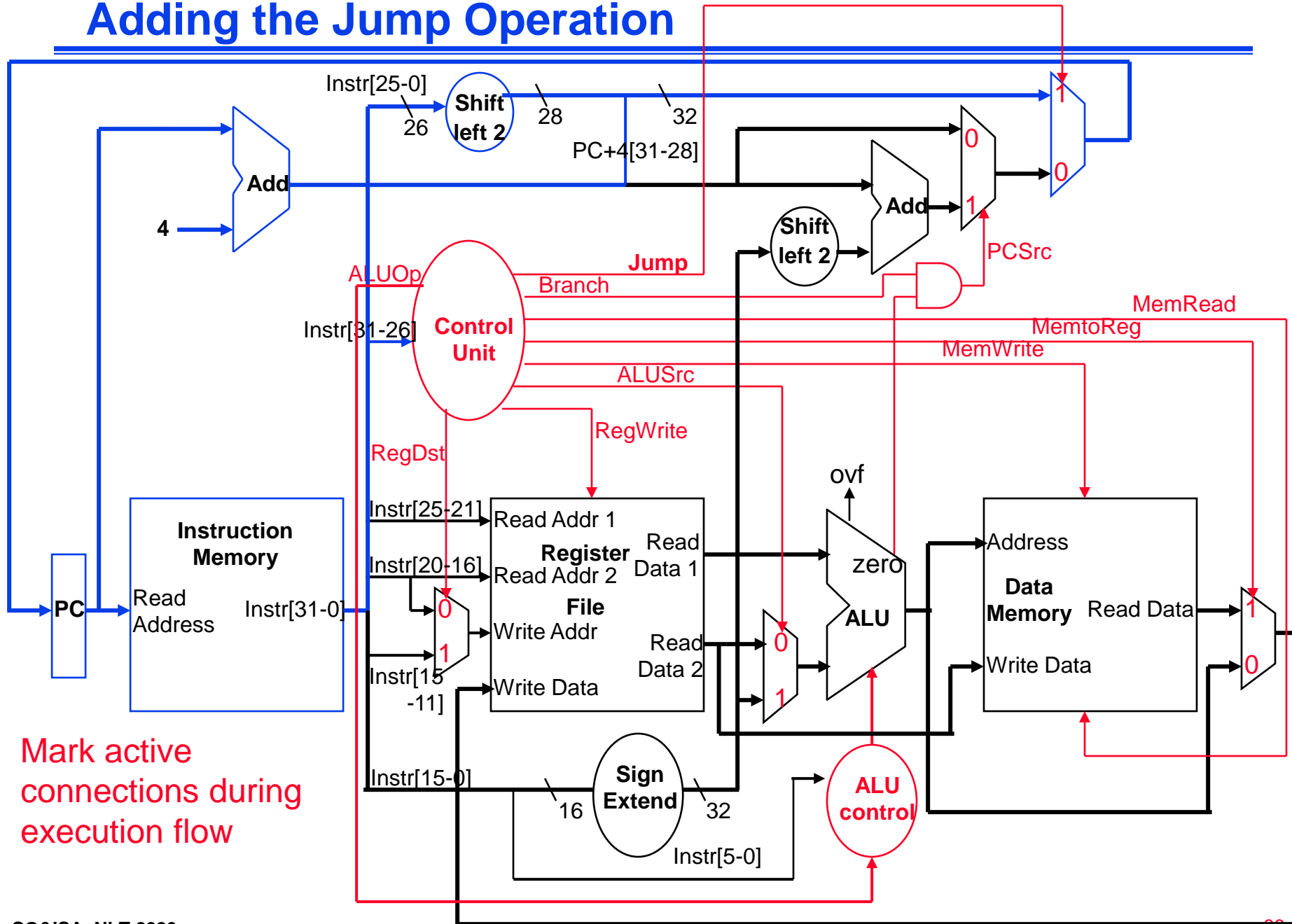


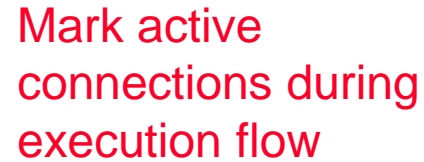
# Branch Instruction Data/Control Flow



Mark active connections during execution flow

# Adding the Jump Operation





## Instruction Times (Critical Paths)

- ❑ What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - ❑ Instruction and Data Memory (200 ps)
  - ❑ ALU and adders (200 ps)
  - ❑ Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type						
load						
store						
beq						
jump						

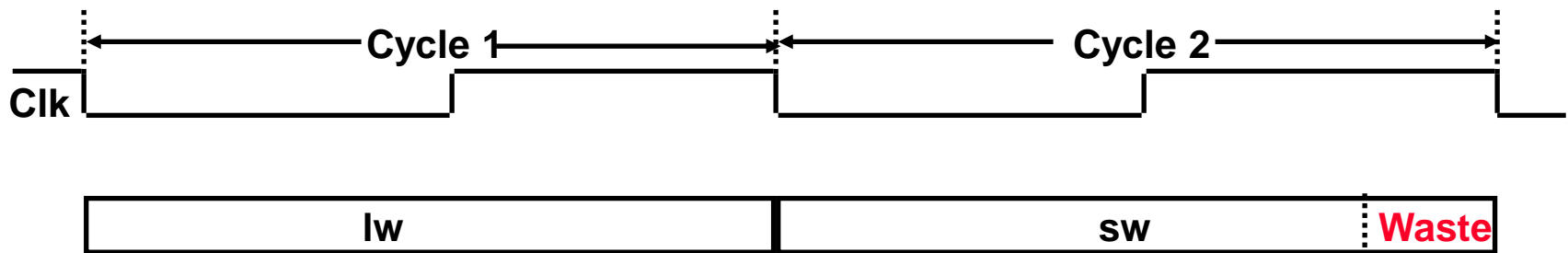
## Instruction Critical Paths for Single cycle CPU

- ❑ What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - ❑ Instruction and Data Memory (200 ps)
  - ❑ ALU and adders (200 ps)
  - ❑ Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

# Single Cycle Disadvantages & Advantages

- ❑ Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instruction
  - ❑ especially problematic for more complex instructions like floating point multiply



- ❑ May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- ❑ Is simple and easy to understand

# How Can We Make The Computer Faster?

---

- ❑ Divide instruction cycles into smaller cycles
- ❑ Executing instructions in parallel
  - ❑ With only one CPU?
- ❑ Pipelining:
  - ❑ Start fetching and executing the next instruction before the current one has completed
  - ❑ Overlapping execution

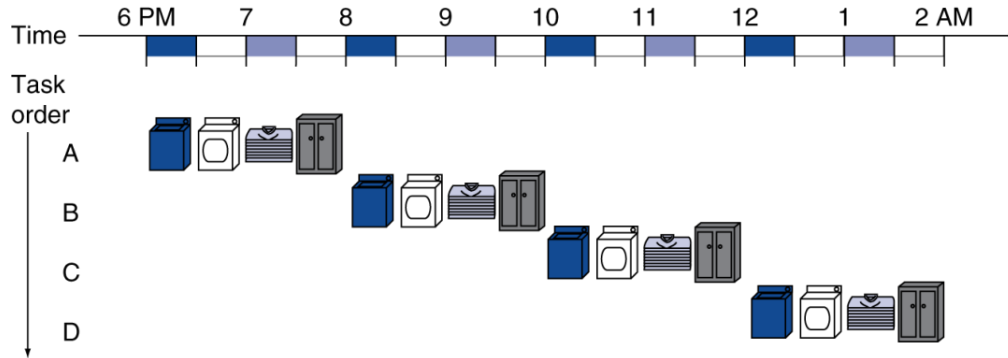
## Pipeline in real life





## A more serious example: laundry work

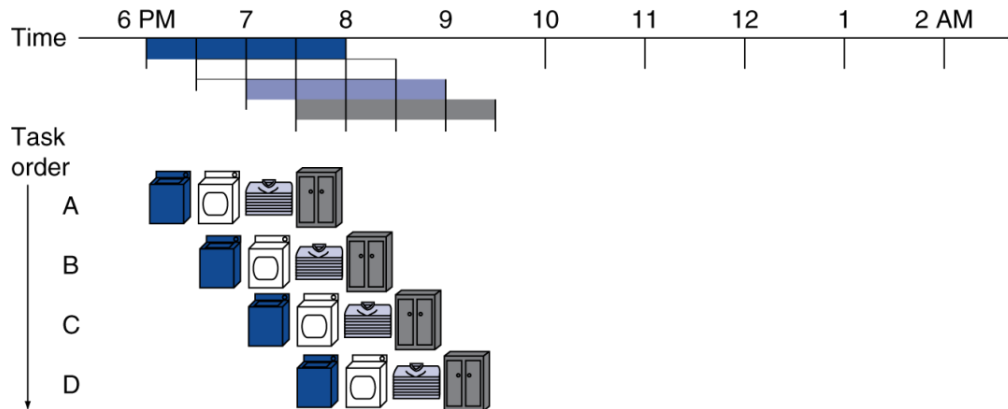
❑ Pipelined laundry boots performance up to 4 times



■ With 4 loads

$$T_{\text{normal}} = 4 * 2 = 8 \text{ hours}$$

$$T_{\text{pipeline}} = 3.5 \text{ hours}$$



■ With n loads

$$T_{\text{normal}} = n * 2 \text{ hours}$$

$$T_{\text{pipeline}} = (3+n)/2 \text{ hours}$$

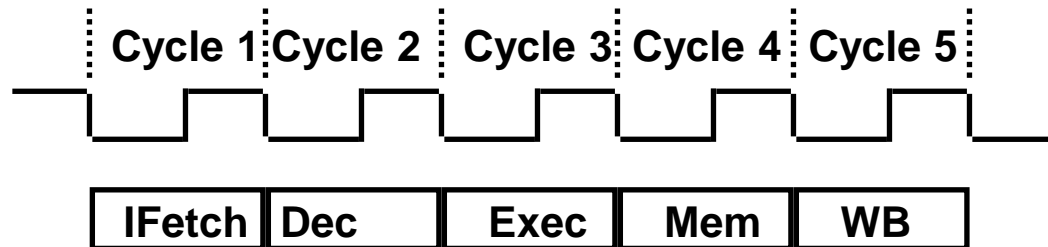
**4 stages: washing, drying, ironing, folding**

When  $n \rightarrow \infty : T_{\text{normal}} \rightarrow 4 * T_{\text{pipeline}}$

# MIPS Pipeline

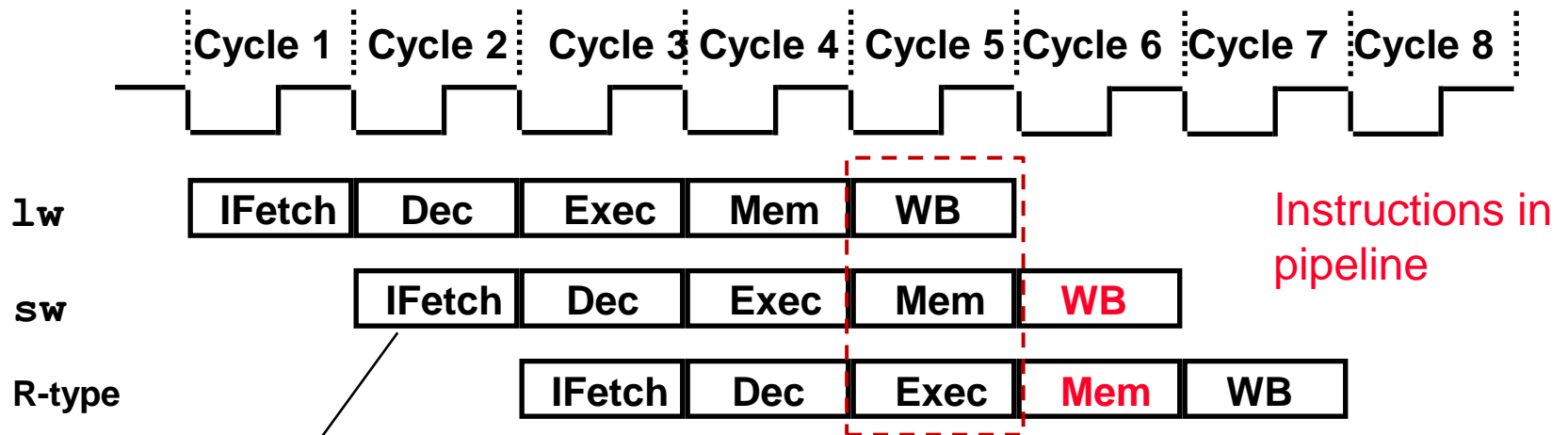
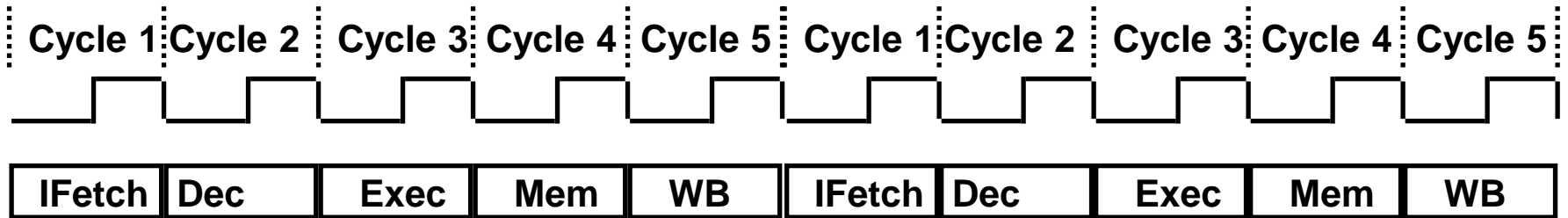
---

- ❑ Five stages, one step per stage
  - ❑ IFetch: Instruction Fetch and Update PC
  - ❑ Dec: Registers Fetch and Instruction Decode
  - ❑ Exec: Execute R-type; calculate memory address
  - ❑ Mem: Read/write the data from/to the Data Memory
  - ❑ WB: Write the result data into the register file



Execution time for a single instruction is always 5 cycles, regardless of instruction operation

# Instruction pipeline

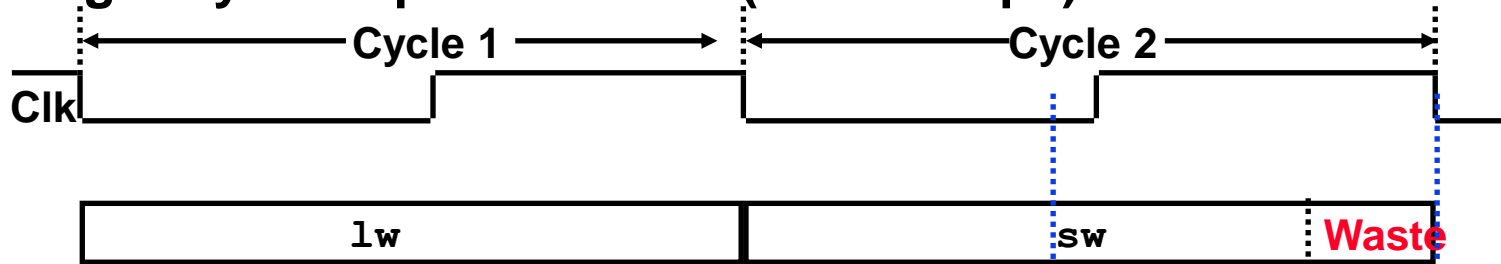


Start fetching and executing the next instruction before the current one has completed

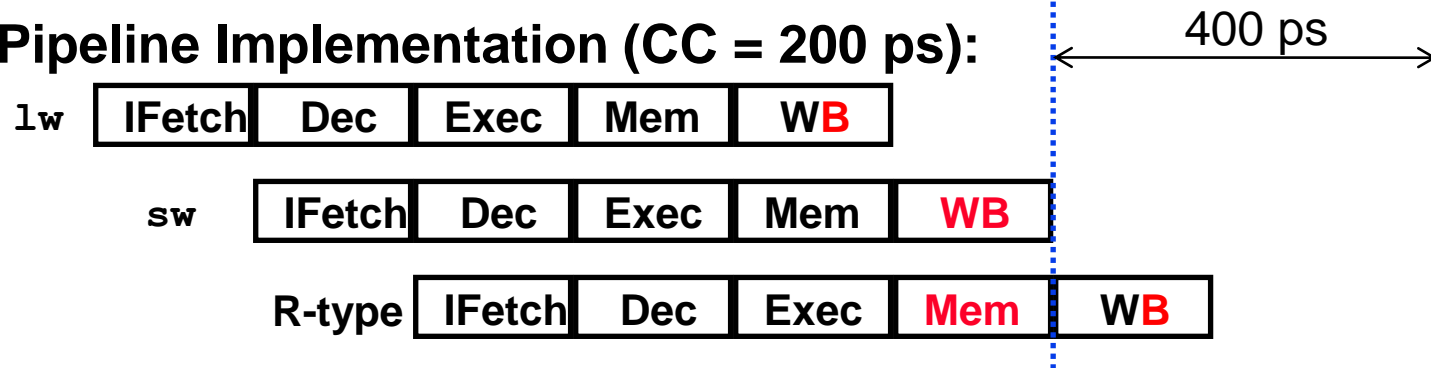
More than one instruction are executed at a time

# Single Cycle versus Pipeline

## Single Cycle Implementation (CC = 800 ps):

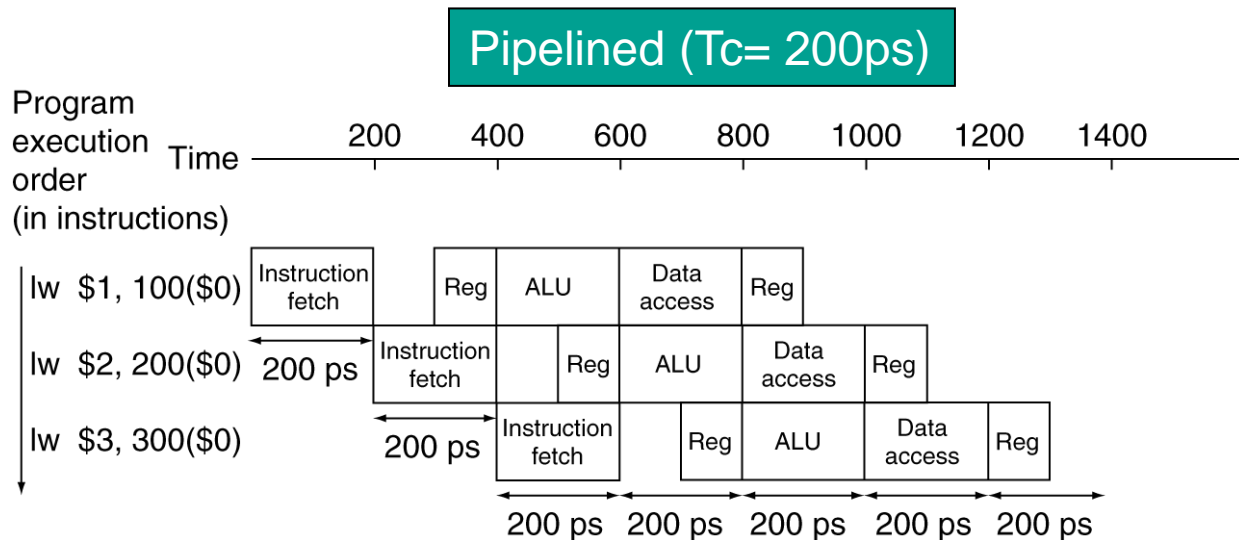
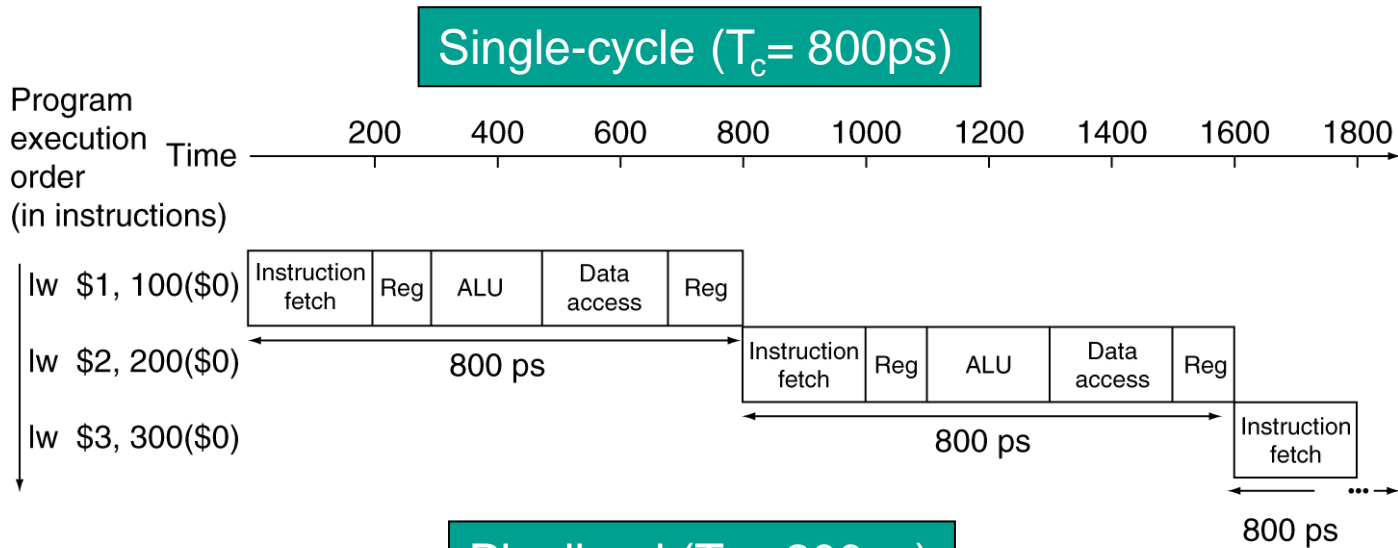


## Pipeline Implementation (CC = 200 ps):



- ❑ To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- ❑ How long does each take to complete 1,000,000 adds ?

# Example with lw instructions



# Pipeline hazards

---

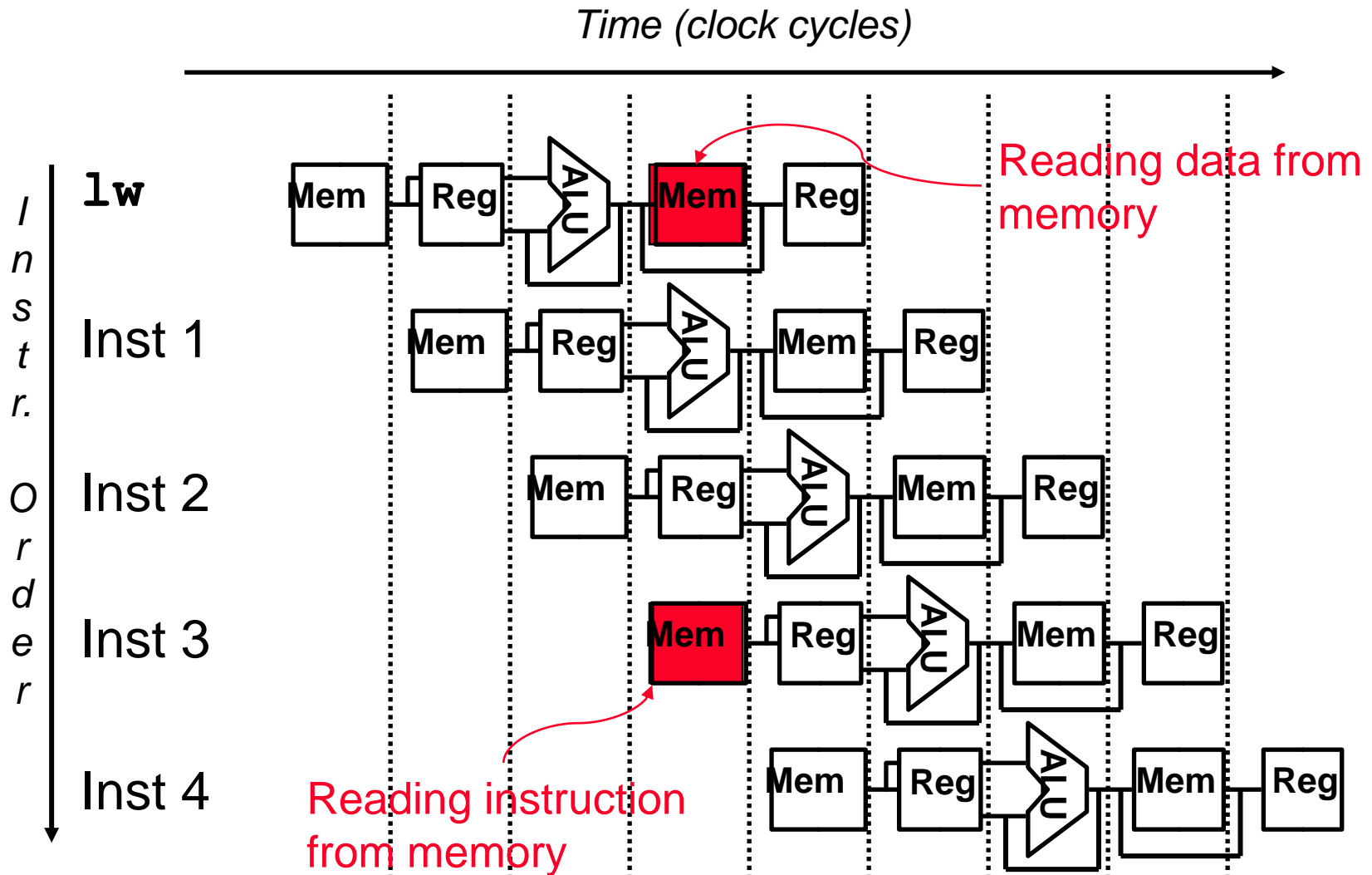
- ❑ Pipeline can lead us into troubles!!!
- ❑ Hazards: situations that prevent starting the next instruction in the next cycle
  - ❑ **structural hazards**: attempt to use the same resource by two different instructions at the same time
  - ❑ **data hazards**: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - ❑ **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions
- ❑ In most cases, hazard can be solved simply by waiting
  - ❑ but we need better solutions to take advantages of pipeline

# Structural hazard

---

- ❑ Conflict for use of a resource
- ❑ In MIPS pipeline with a single memory
  - ❑ Load/store requires data access
  - ❑ Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- ❑ Hence, pipelined datapath require separate instruction/data memories
  - ❑ Or separate instruction/data caches

# A Single Memory Would Be a Structural Hazard

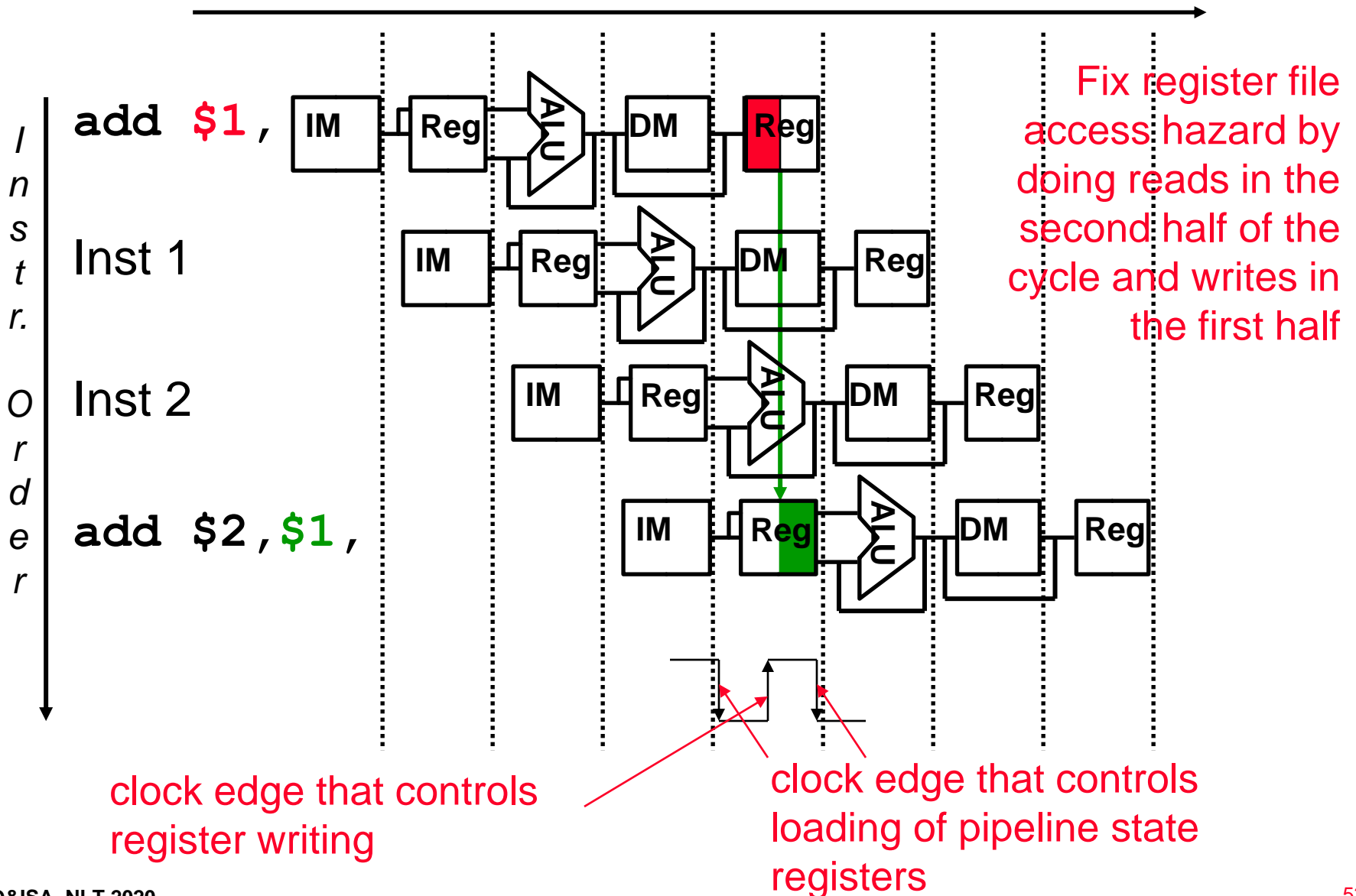


- Fix with separate instr and data memories (I\$ and D\$)



# How About Register File Access?

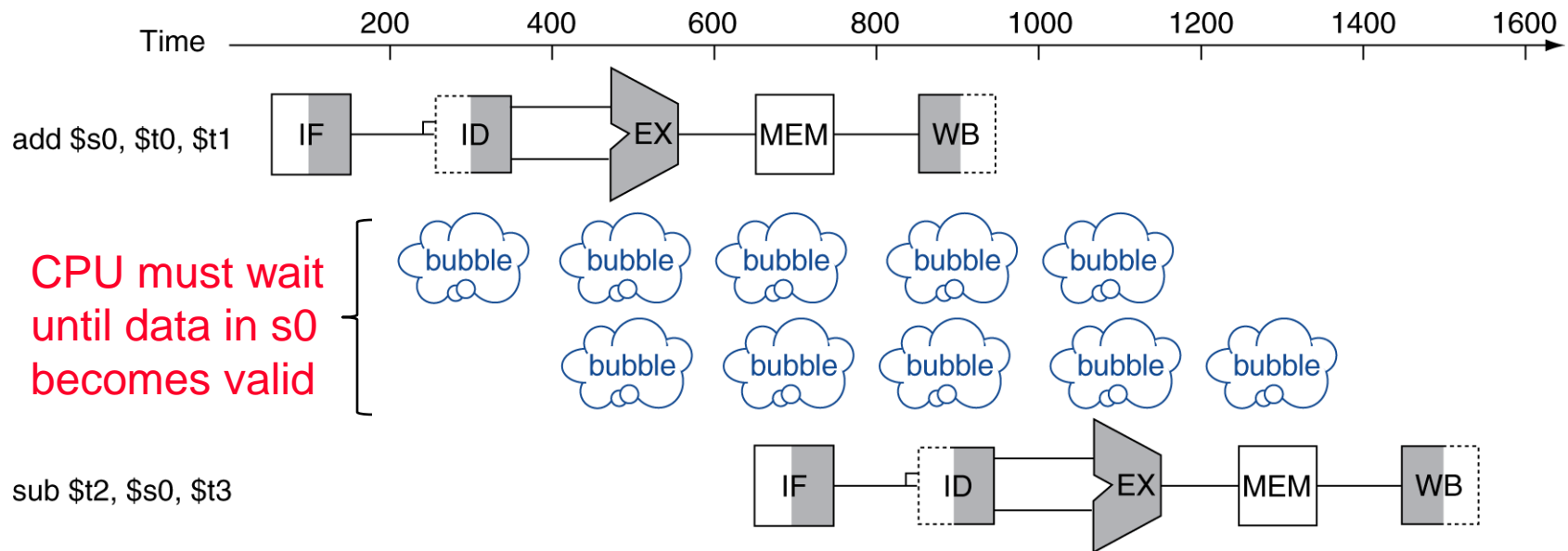
Time (clock cycles)



# Data hazard

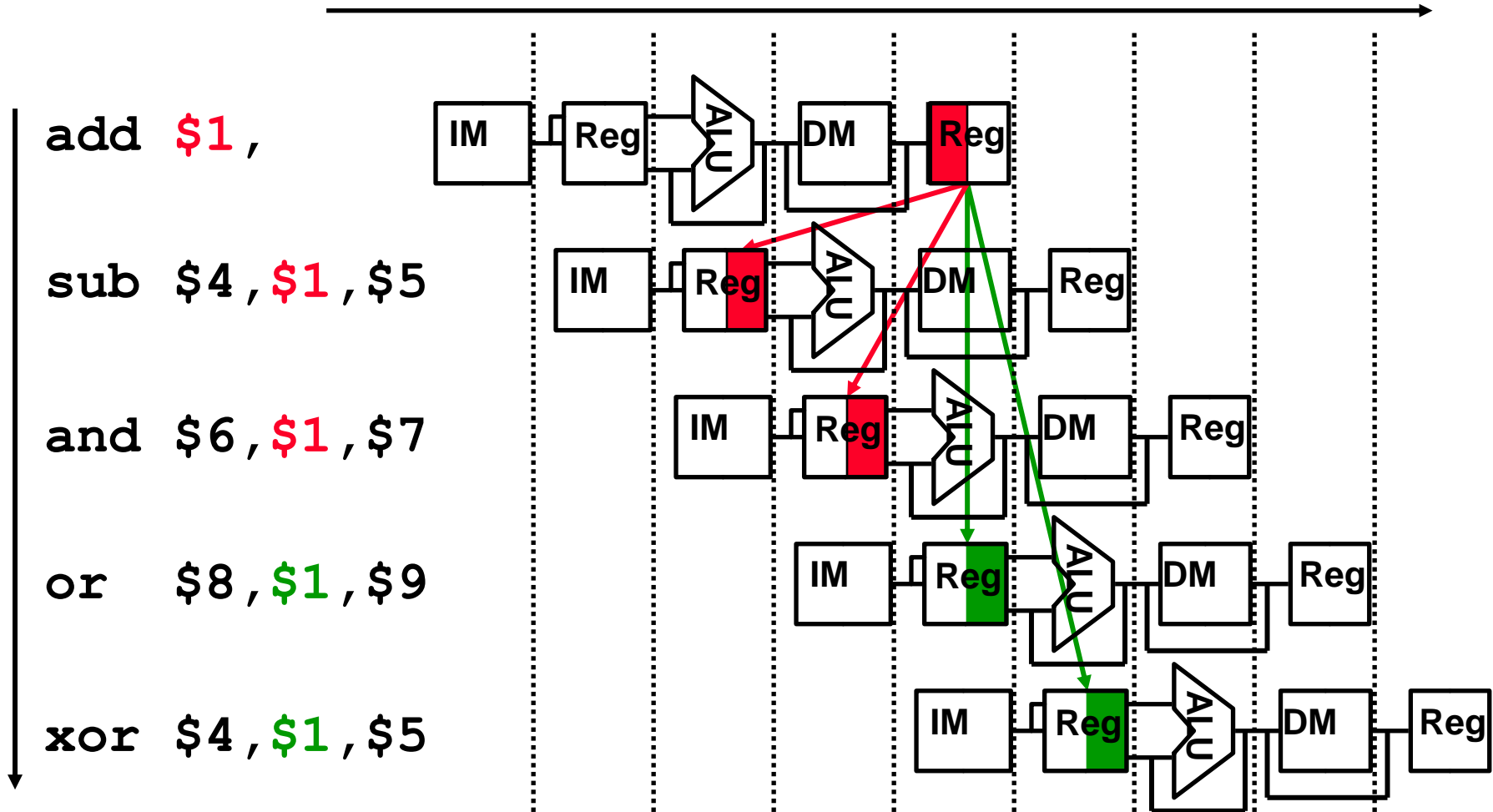
❑ An instruction depends on completion of data access by a previous instruction

❑ add      \$s0, \$t0, \$t1  
   sub      \$t2, \$s0, \$t3



## Example

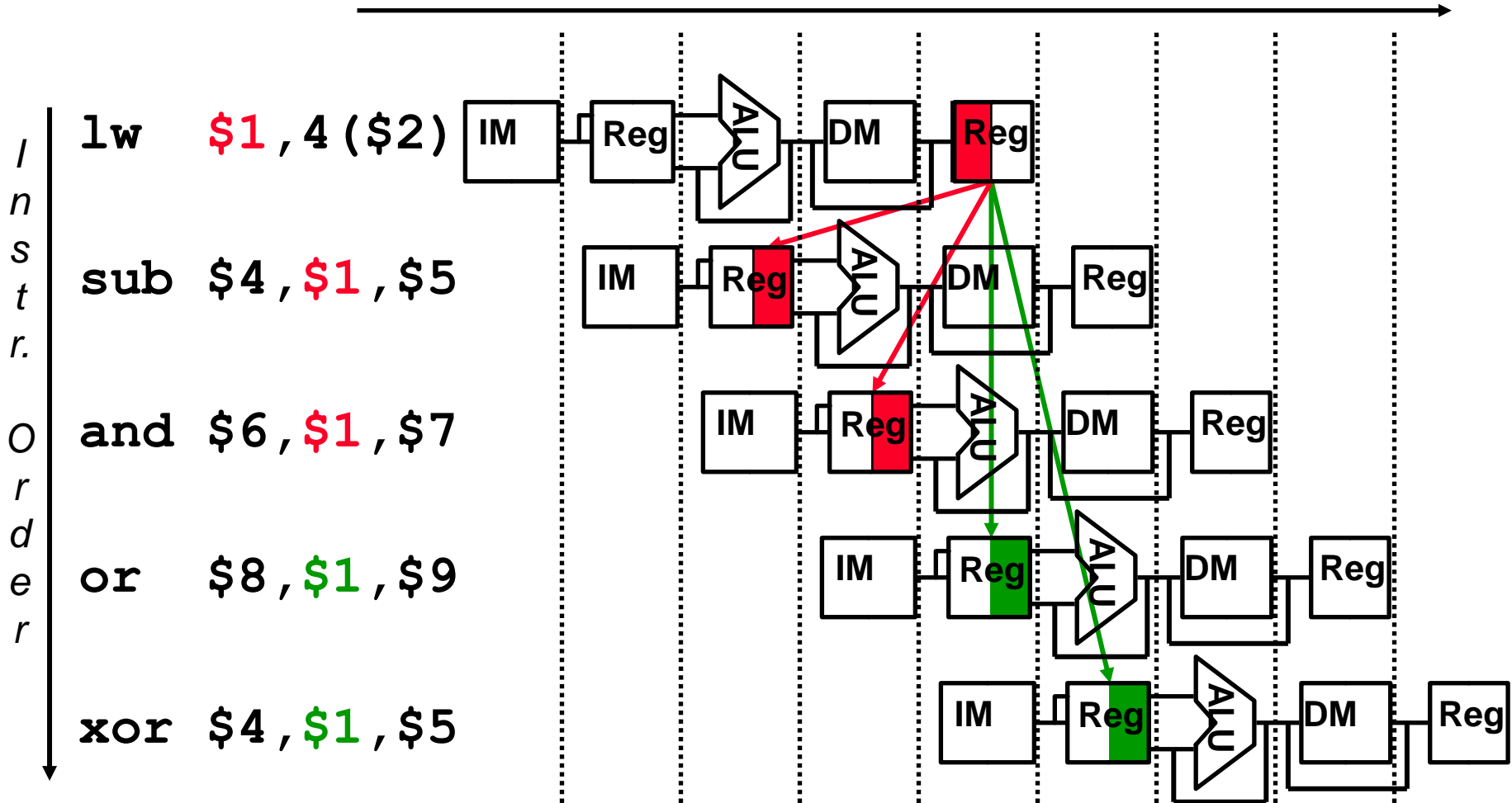
- Dependencies backward in time cause **hazards**



- Read before write **data hazard**

## Example

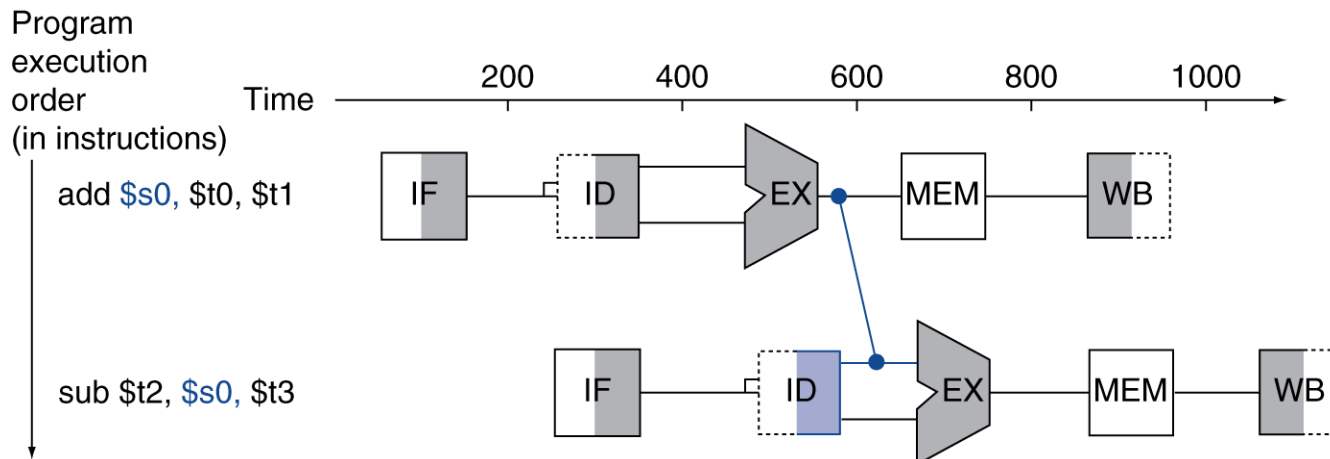
- Dependencies backward in time cause **hazards**



- Load-use data hazard**

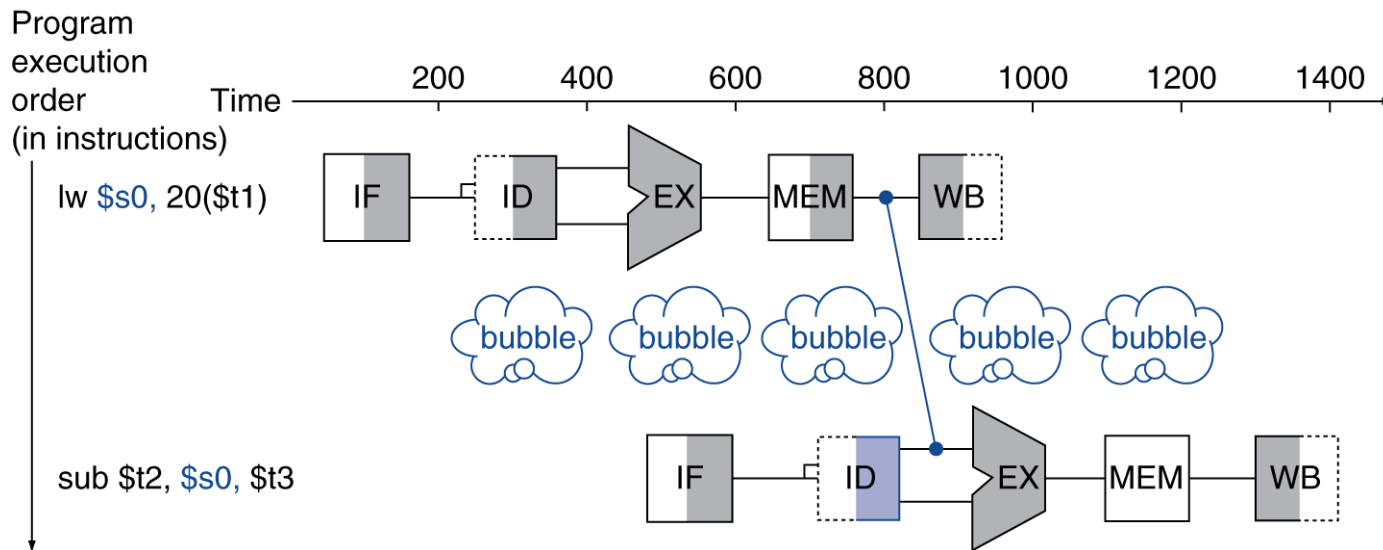
# Solving hazard with forwarding

- ❑ Use result when it is computed
  - ❑ Don't wait for it to be stored in a register
  - ❑ Requires extra connections in the datapath
- ❑ Forward from EX to EX (output to input)



# Load-Use Data Hazard

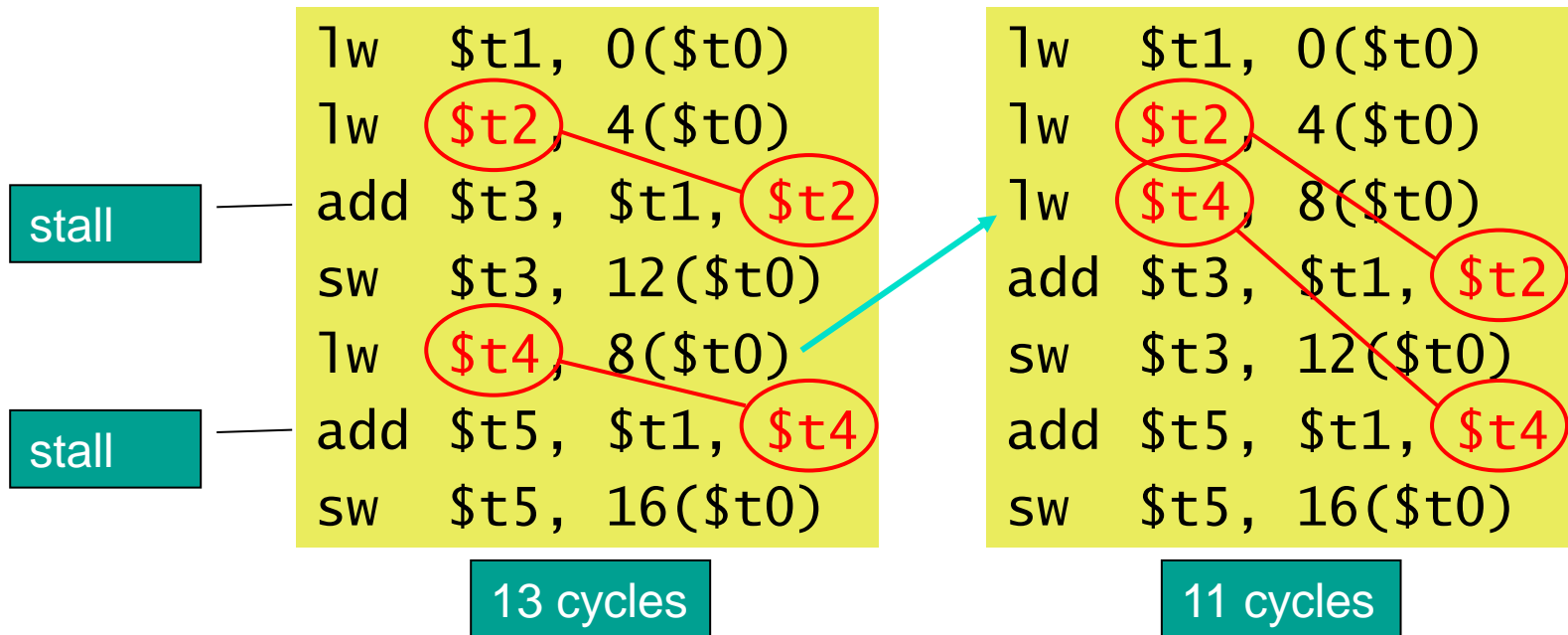
- ❑ One cycle stall is necessary
- ❑ Forward from MEM (output) to EX (input)



# Code Scheduling to Avoid Stalls

- ❑ Reorder code to avoid use of load result in the next instruction

❑ C code:      $A = B + E;$   
               $C = B + F;$



# Control Hazards

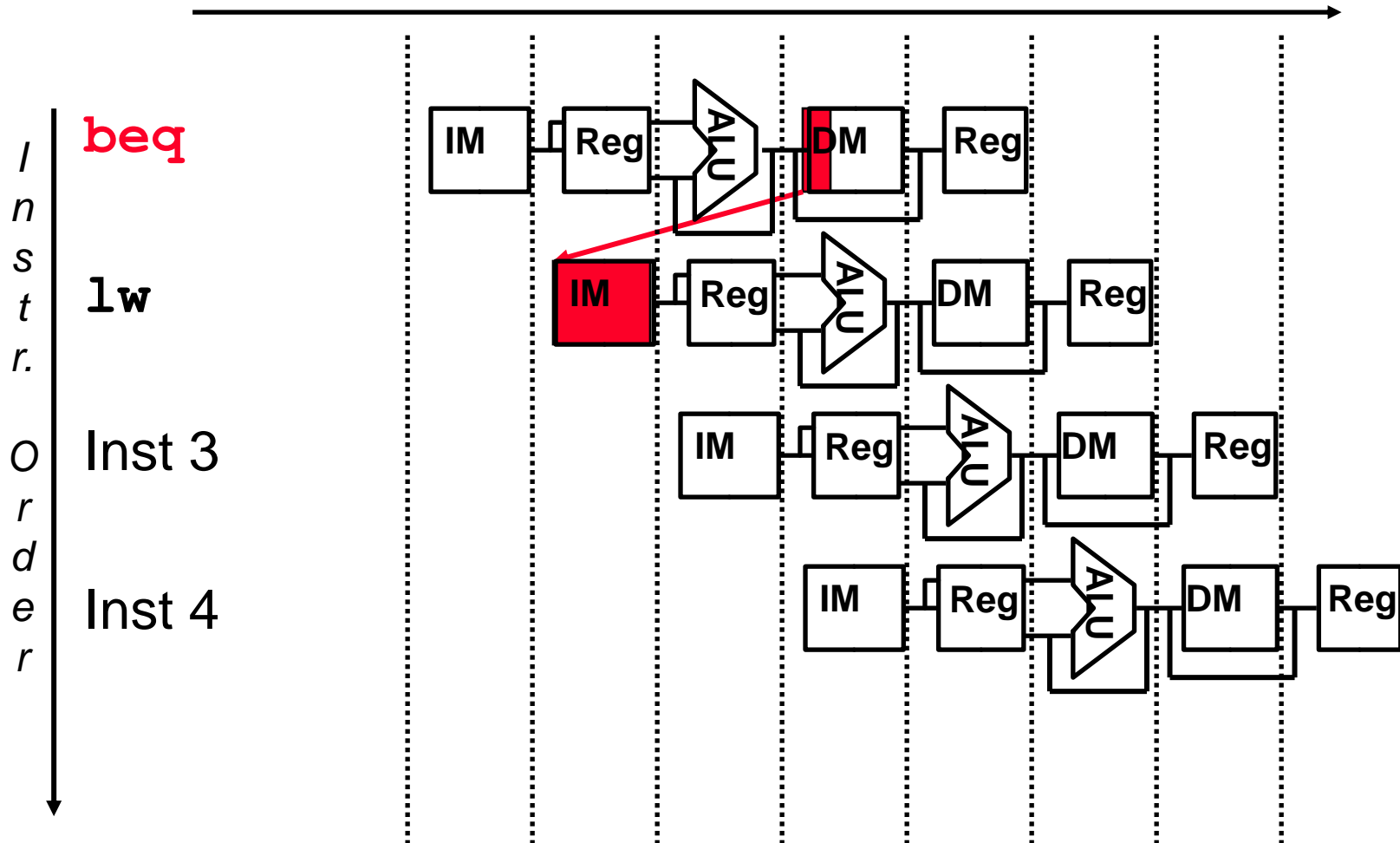
---

- ❑ Branch determines flow of control
  - ❑ Fetching next instruction depends on branch outcome
  - ❑ Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- ❑ In MIPS pipeline
  - ❑ Need to compare registers and compute target early in the pipeline
  - ❑ Add hardware to do it in ID stage



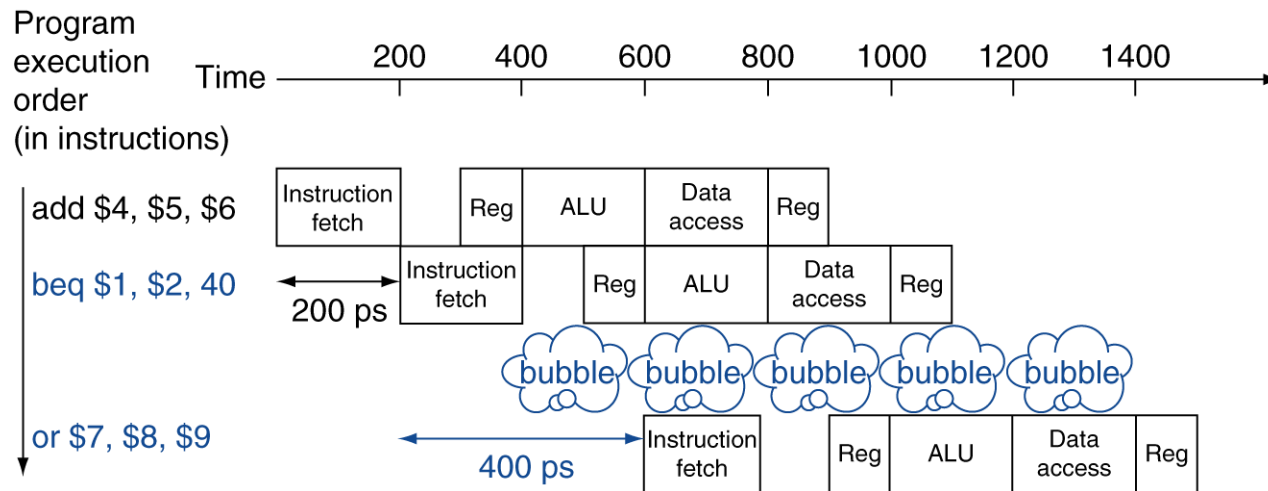
# Branch Instructions Cause Control Hazards

- Dependencies backward in time cause **hazards**



# Stall on Branch

- ❑ Naïve approach: Wait until branch outcome determined before fetching next instruction



Performance affect: assume that 17% of instructions in program are branches, if each branch take one cycle for the stall, then performance will be 17% slower. (CPI = 1.17)

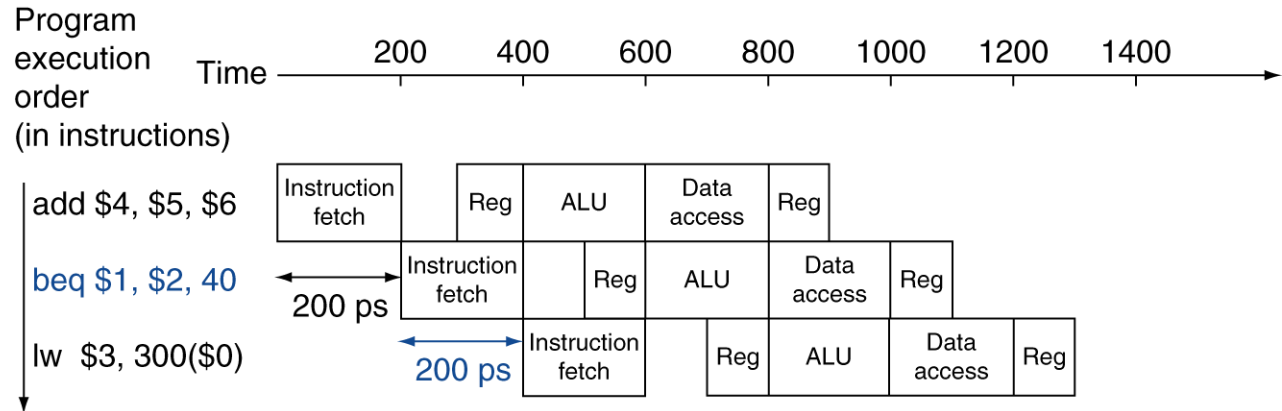
# Branch Prediction

---

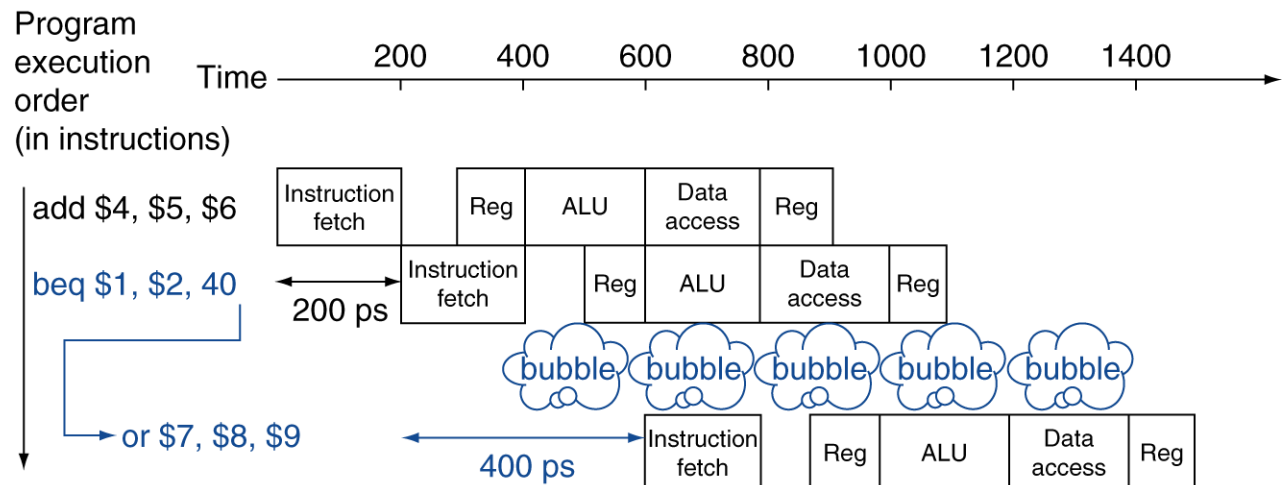
- ❑ Predict outcome of branch
- ❑ Only stall if prediction is wrong
- ❑ In MIPS pipeline
  - ❑ Can predict branches not taken
  - ❑ Fetch instruction after branch, with no delay

# MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



# More-Realistic Branch Prediction

---

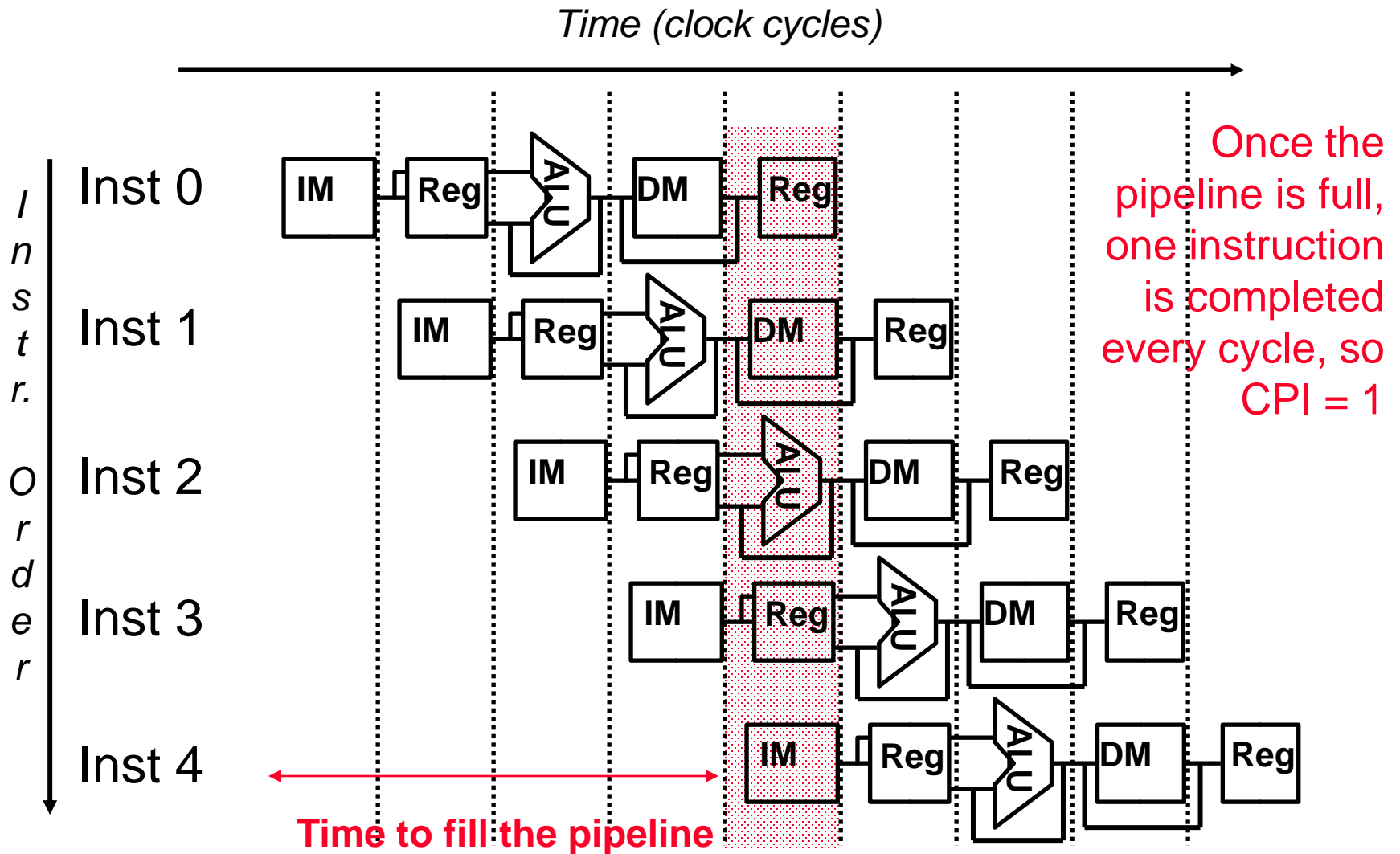
## ❑ Static branch prediction

- ❑ Based on typical branch behavior
- ❑ Example: loop and if-statement branches
  - Predict backward branches taken
  - Predict forward branches not taken

## ❑ Dynamic branch prediction

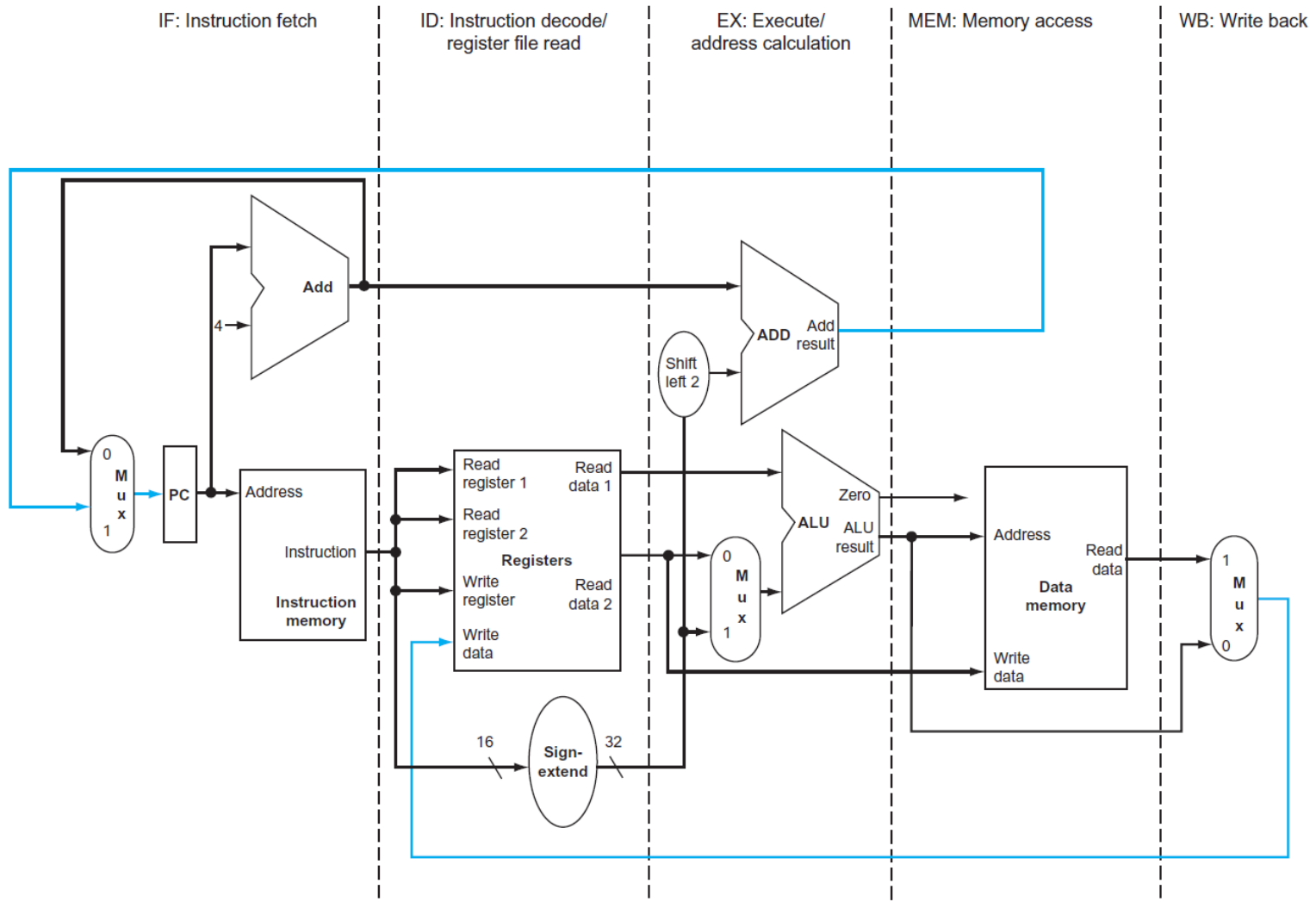
- ❑ Hardware measures actual branch behavior
  - e.g., record recent history of each branch
- ❑ Assume future behavior will continue the trend
  - When wrong, stall while re-fetching, and update history
- ❑ As good as > 90% accuracy

# Summary: Pipeline Operation



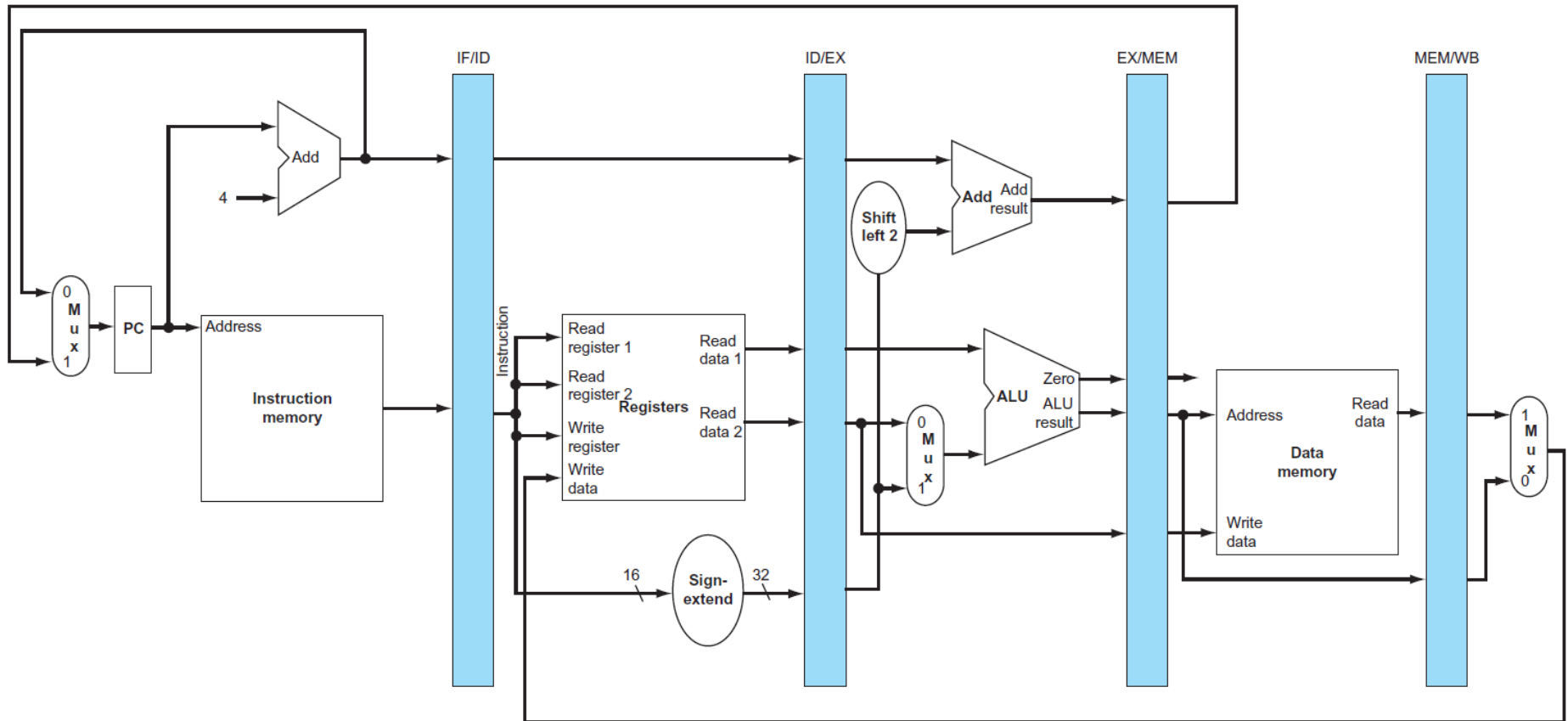
# Pipelined datapath

## ❑ How to share/isolate data between different stages?



# Pipelined datapath

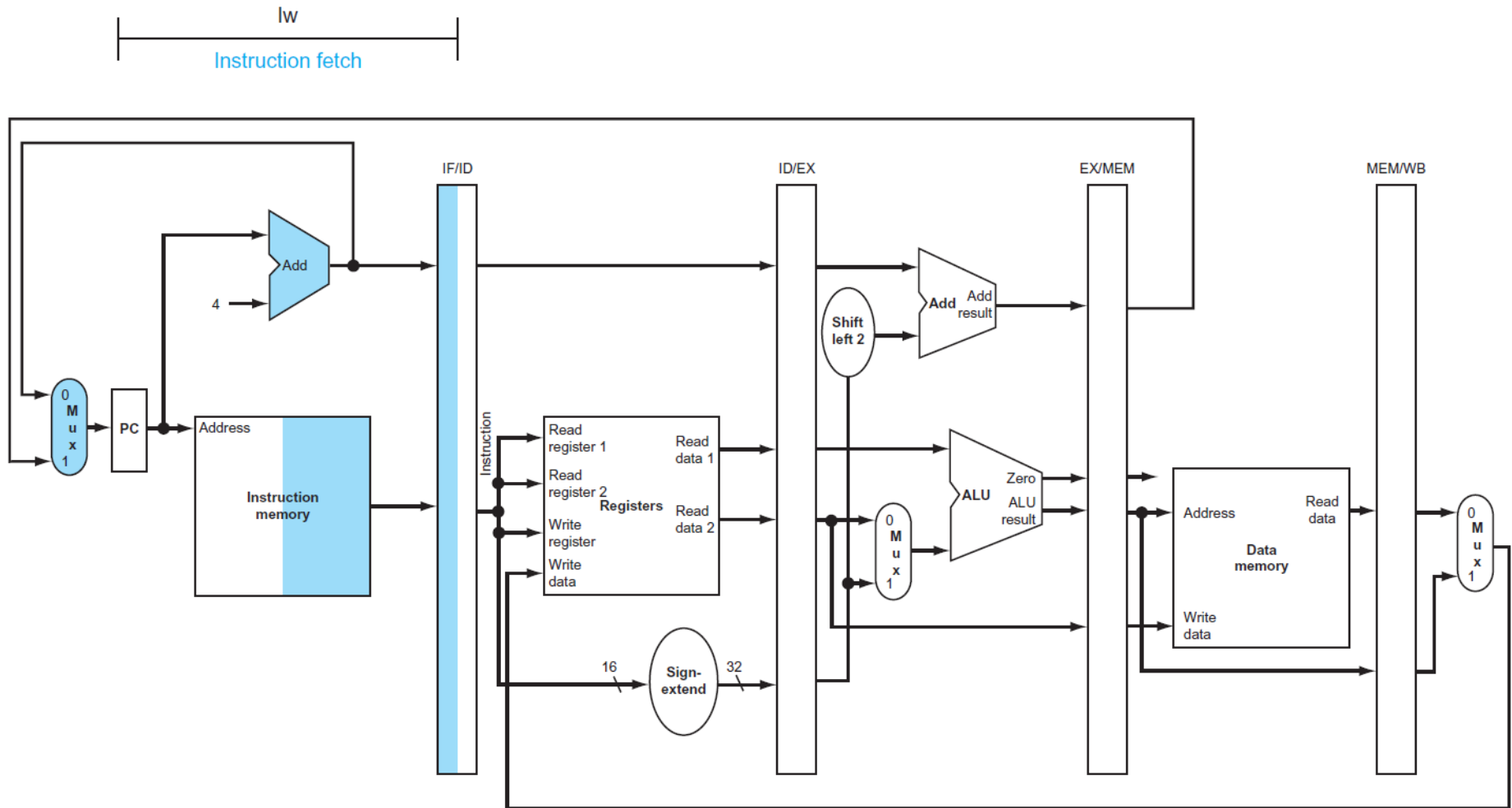
## ❑ Adding pipeline registers



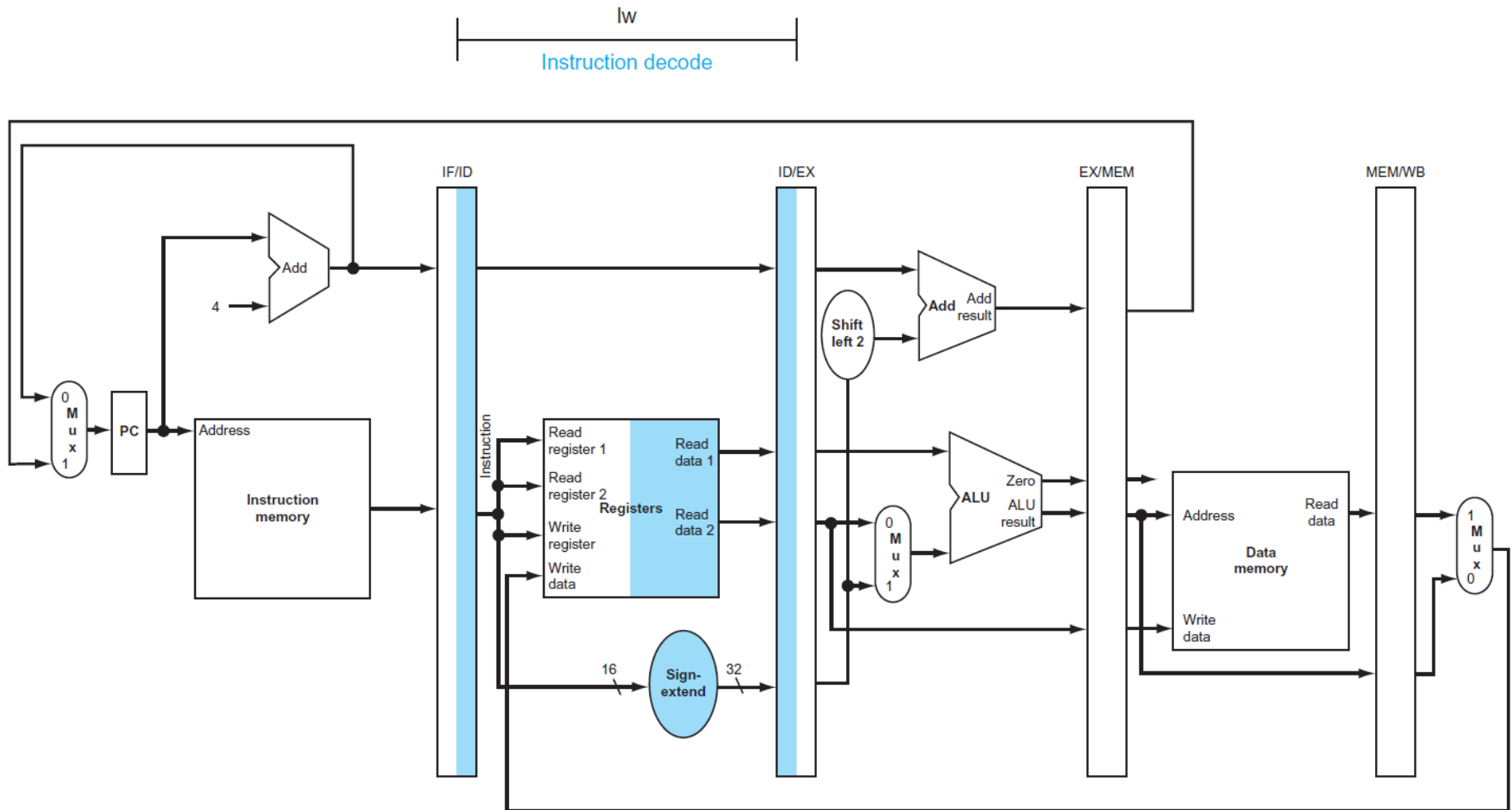
## ❑ 5 stages, why only 4 registers?



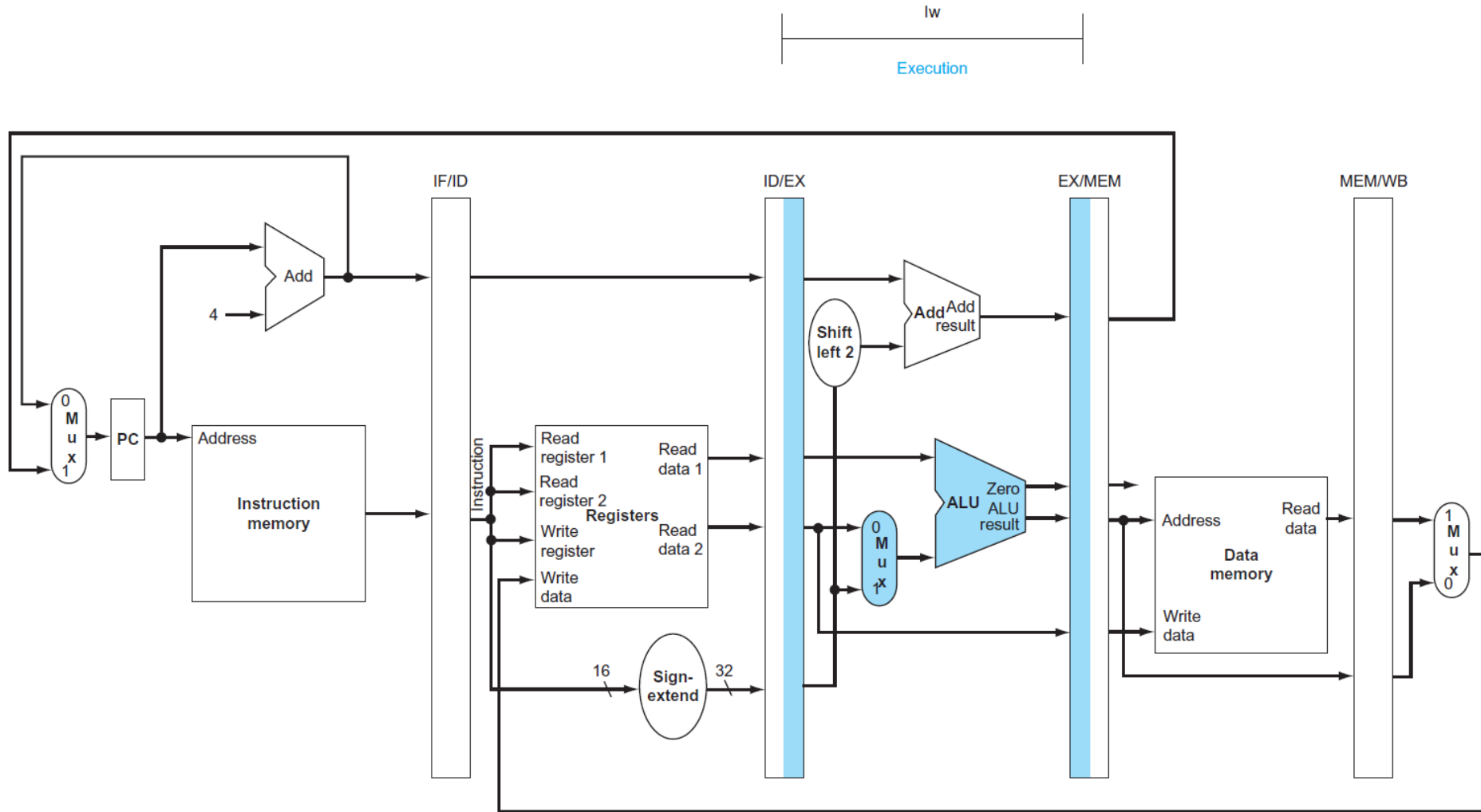
# Instruction fetch in pipeline



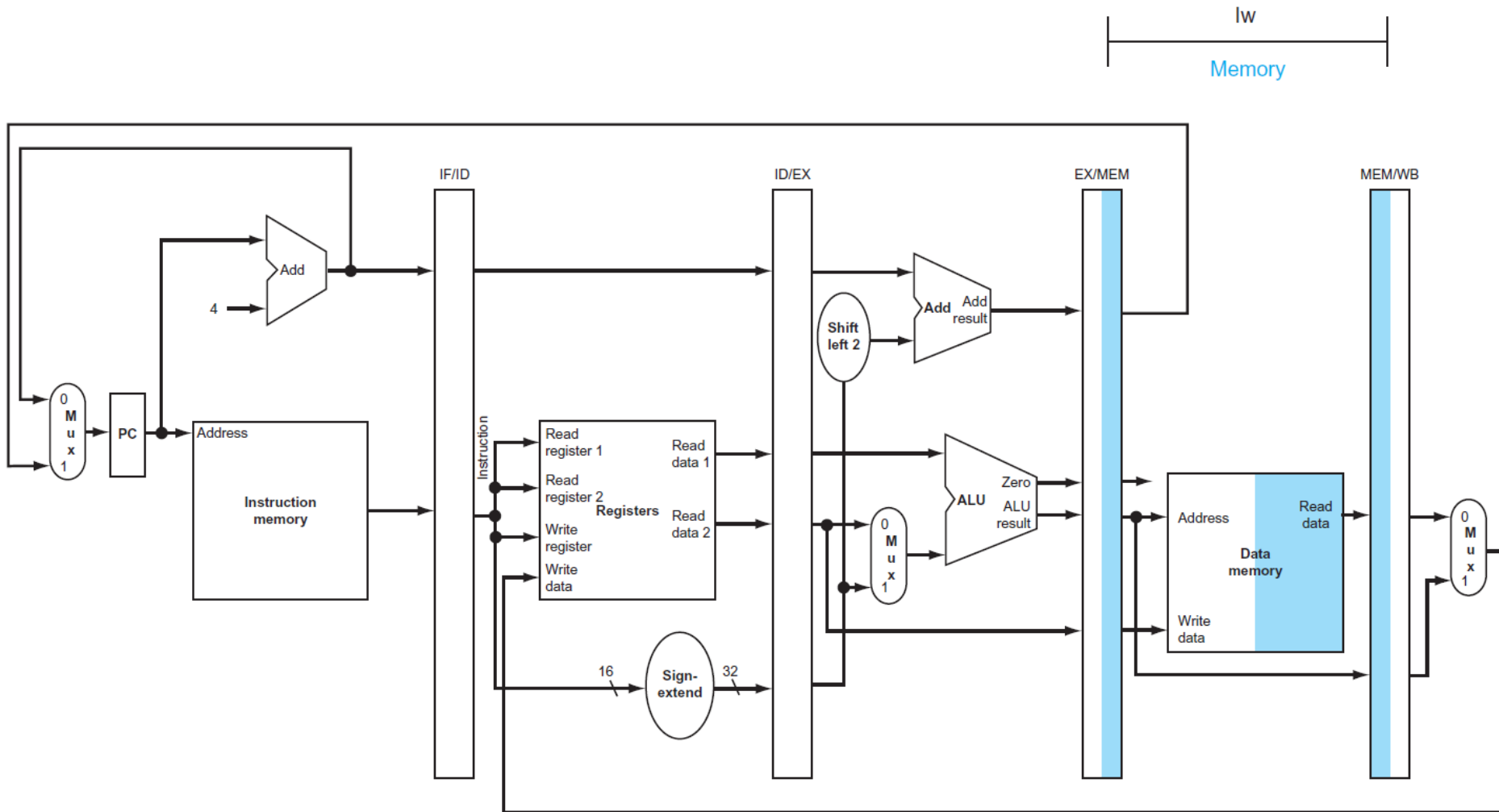
# Instruction decode



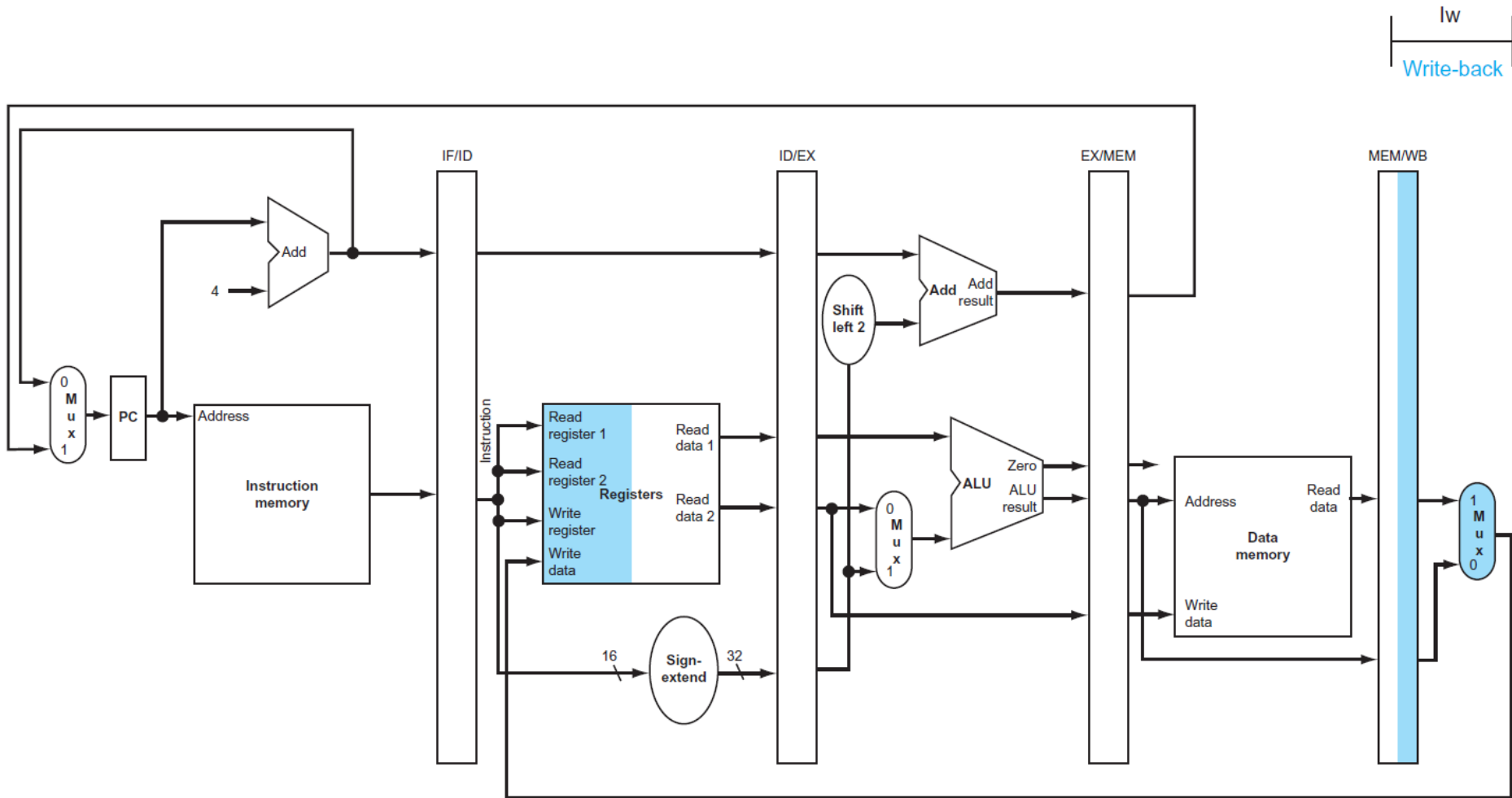
# Execution



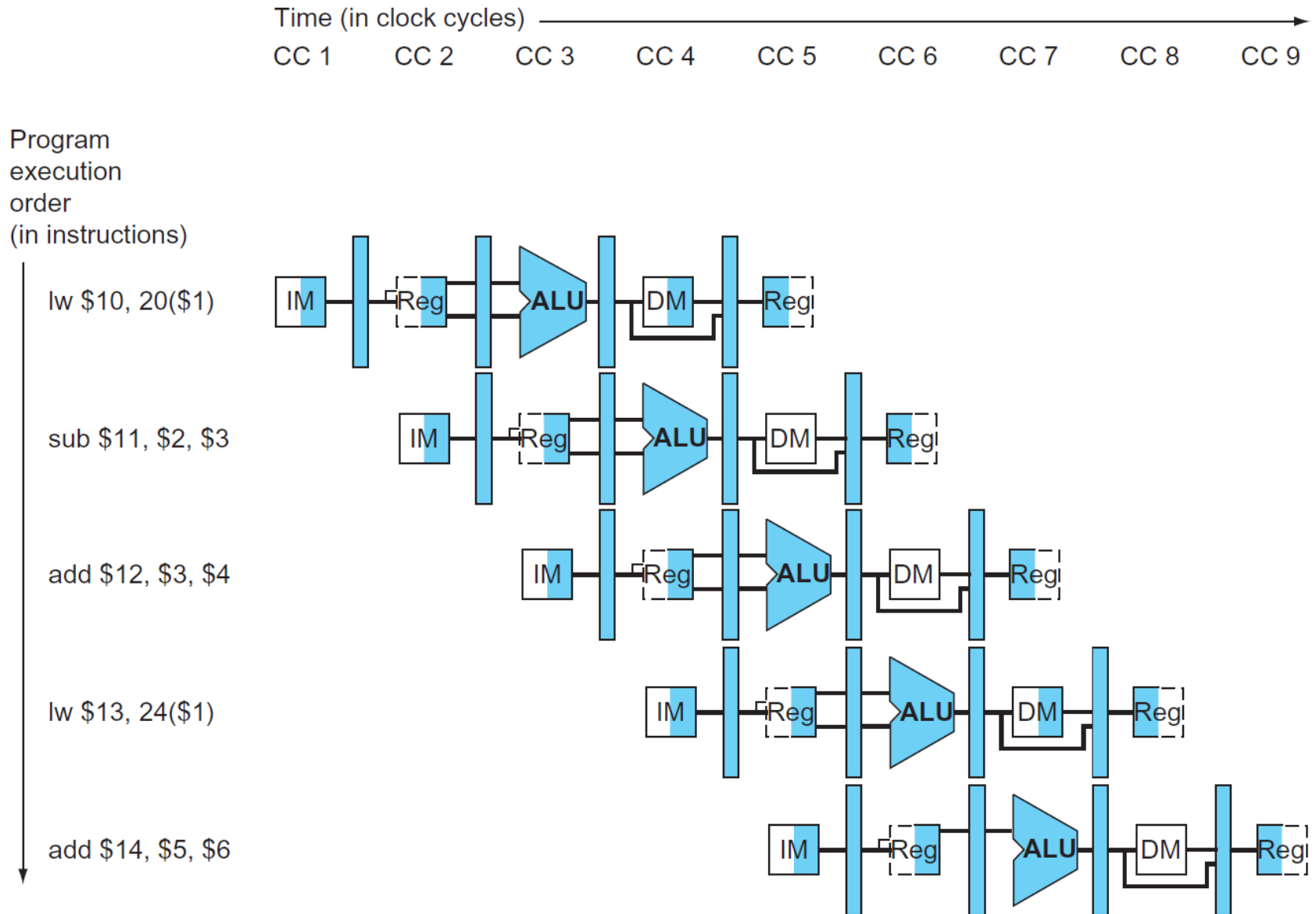
# Memory access



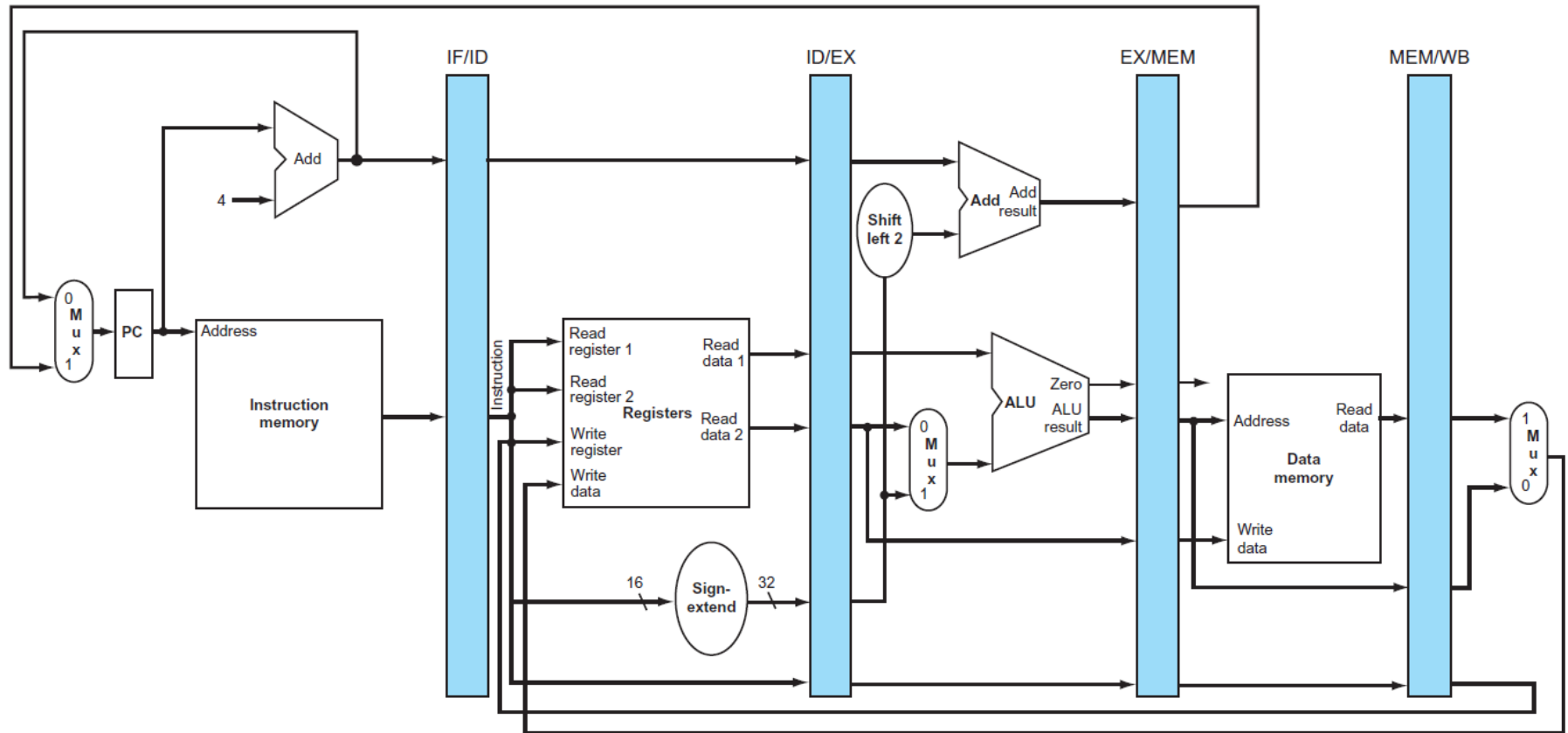
# Write-back



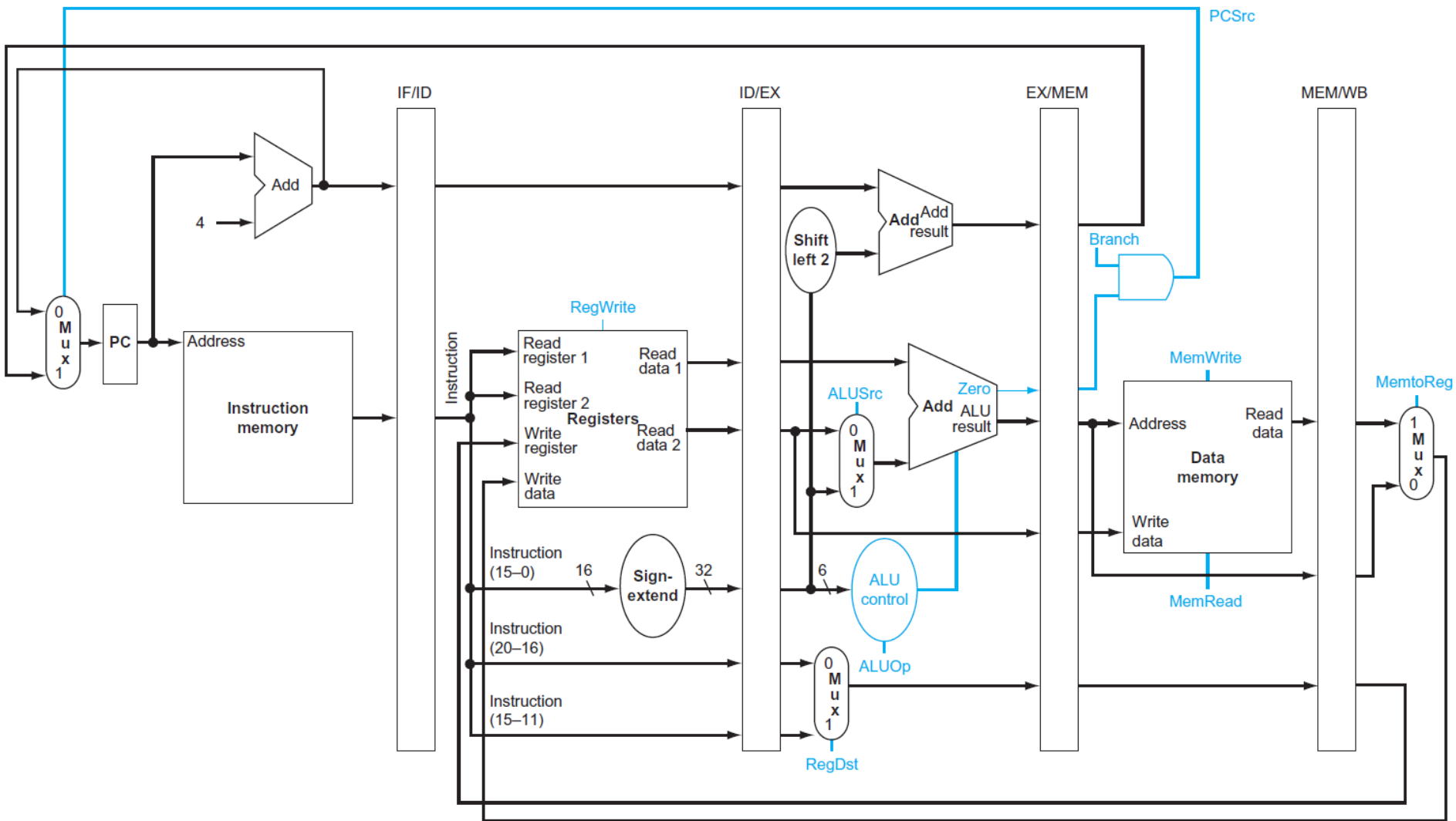
# Pipeline diagram



# Pipeline diagram

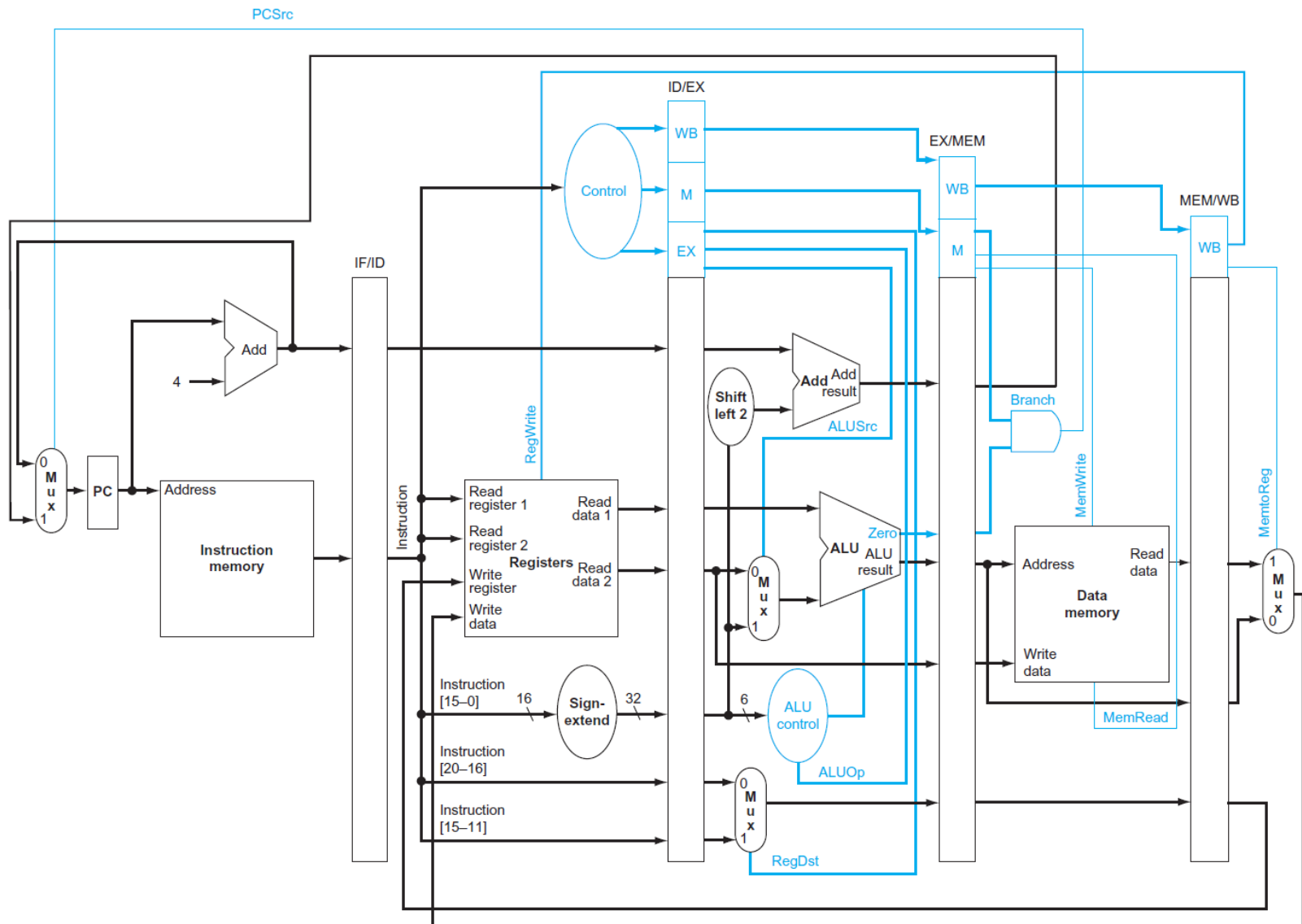


# Control signals in pipeline

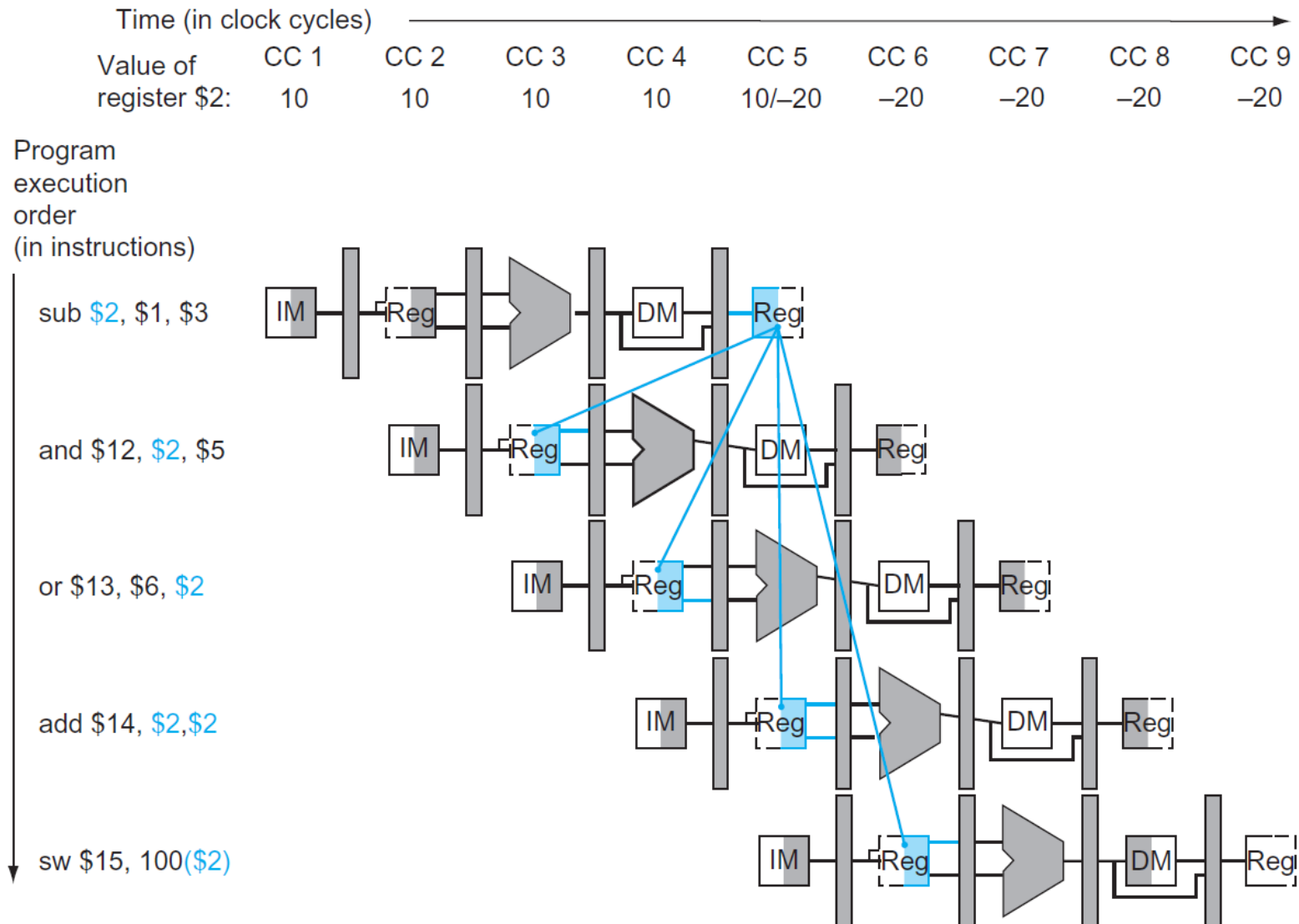




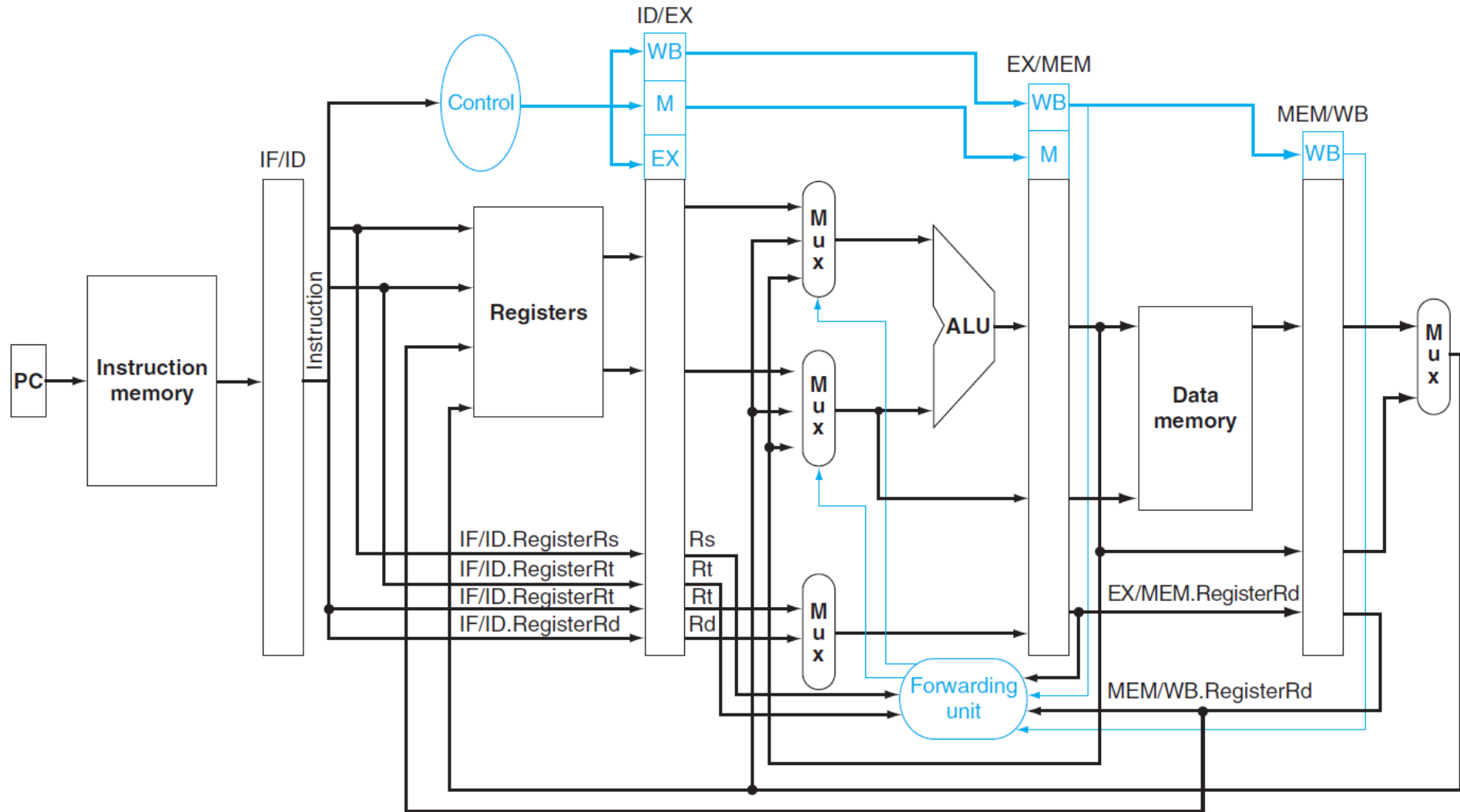
## Pipelined control



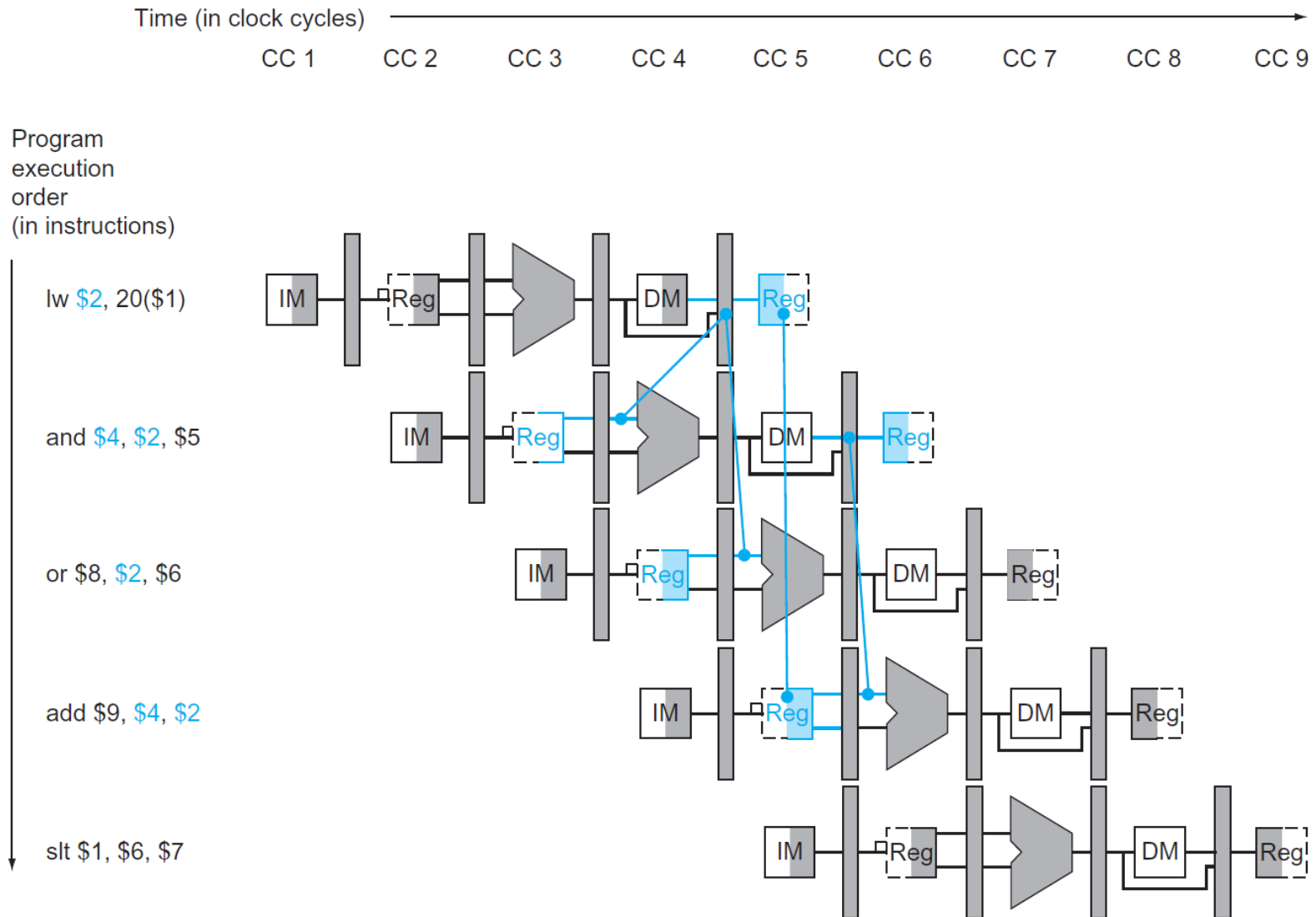
# Solving read-after-write hazard



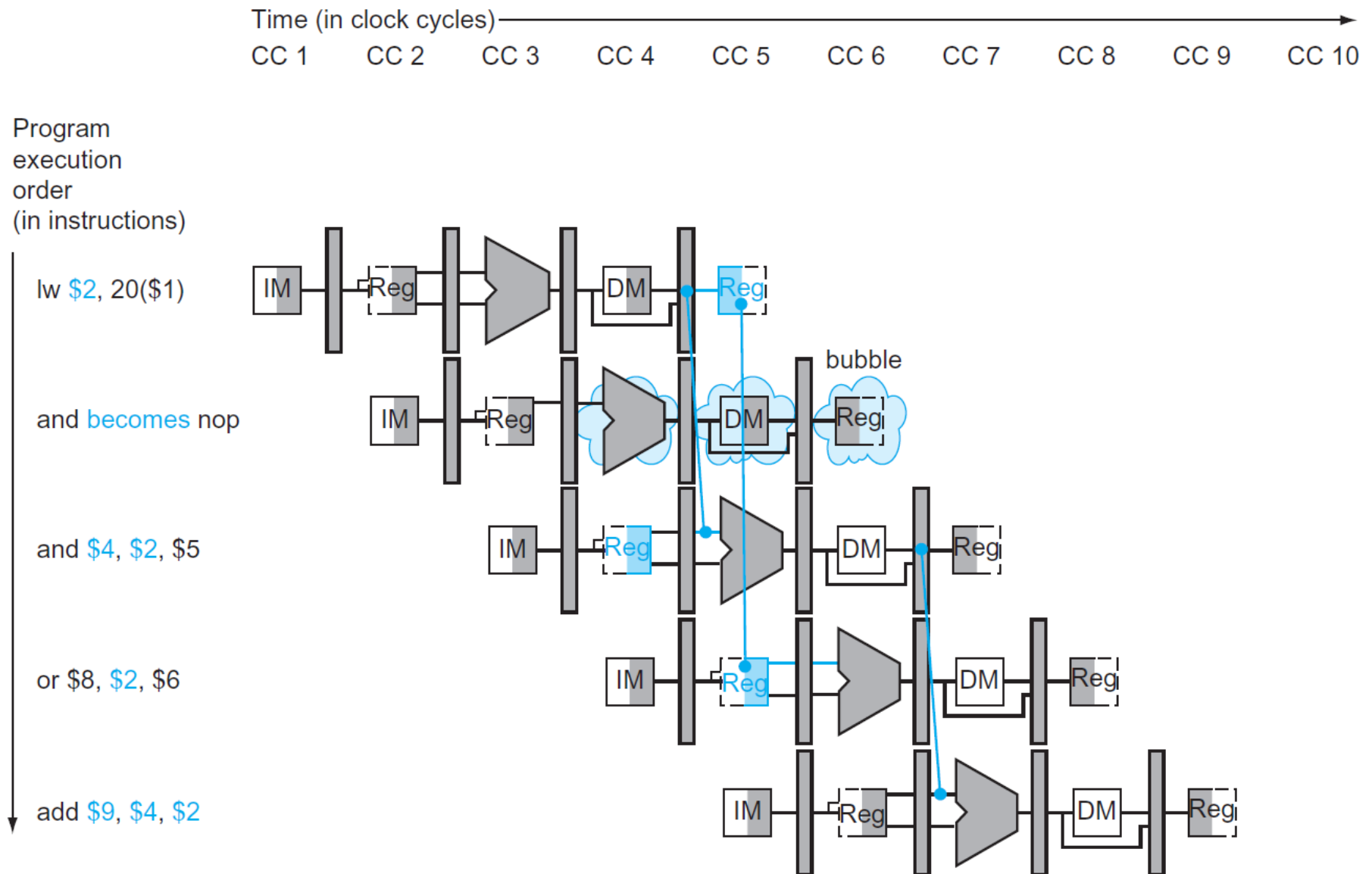
# Solving read-after-write hazard



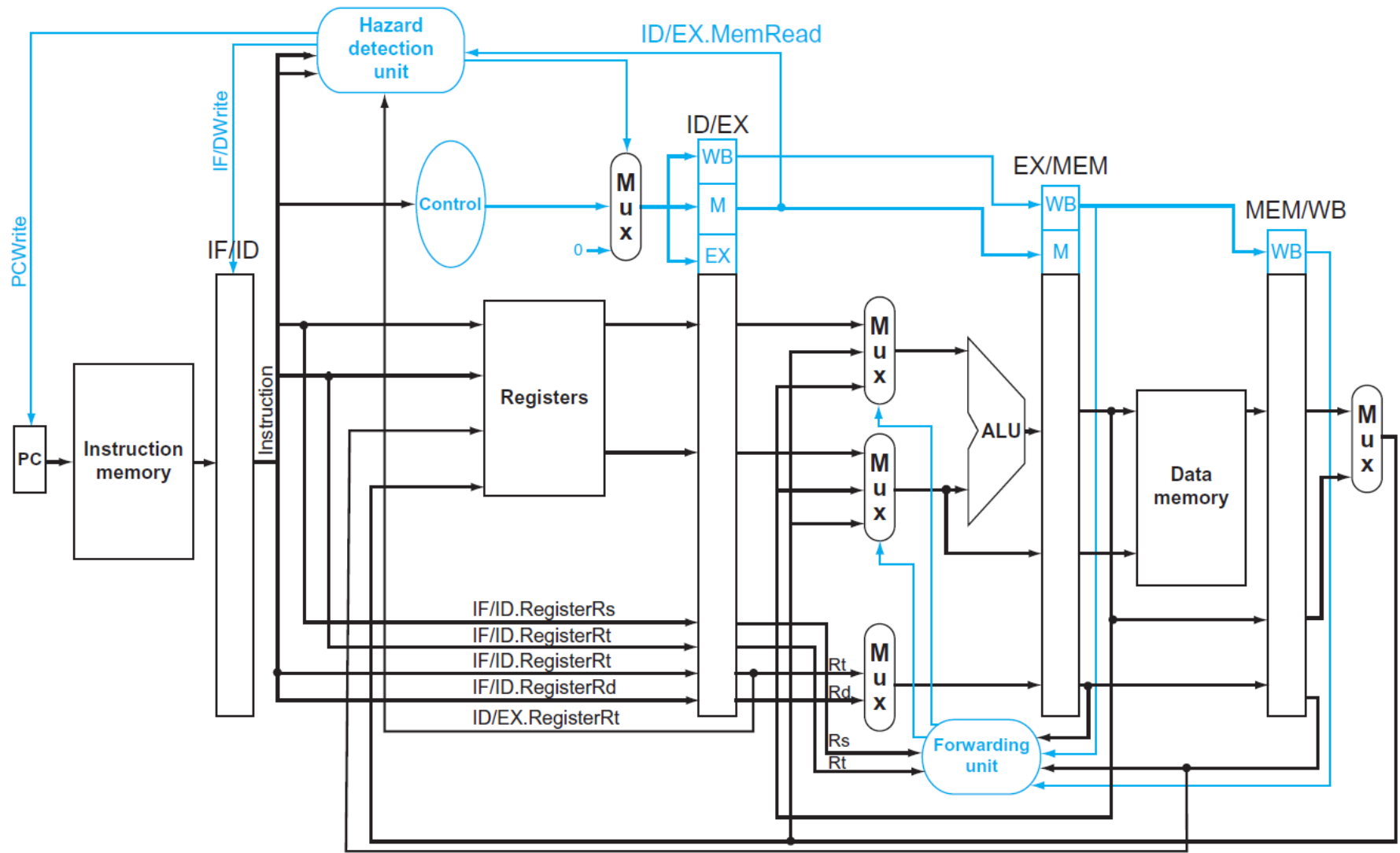
# Solving load-use hazard



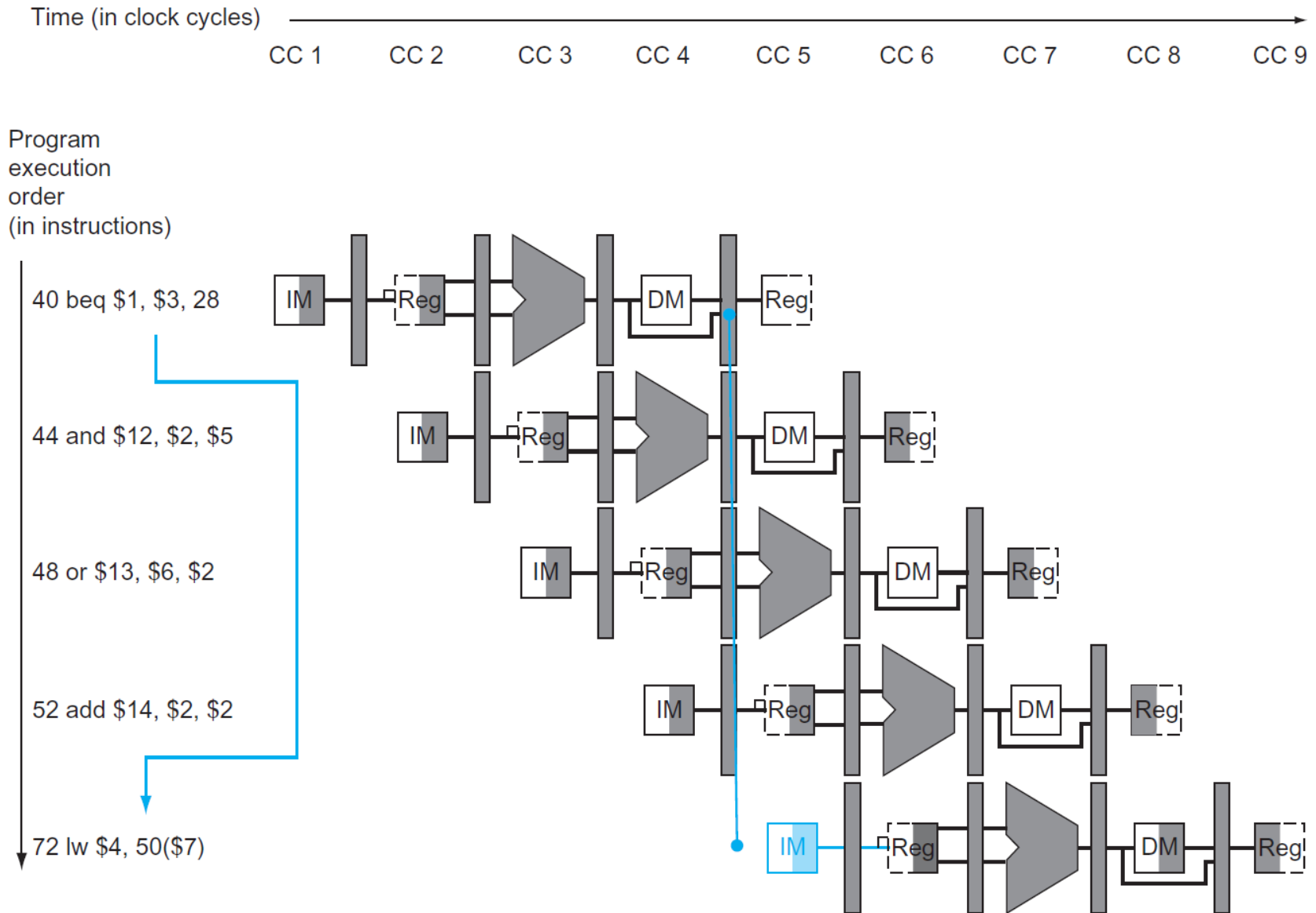
# Solving load-use hazard



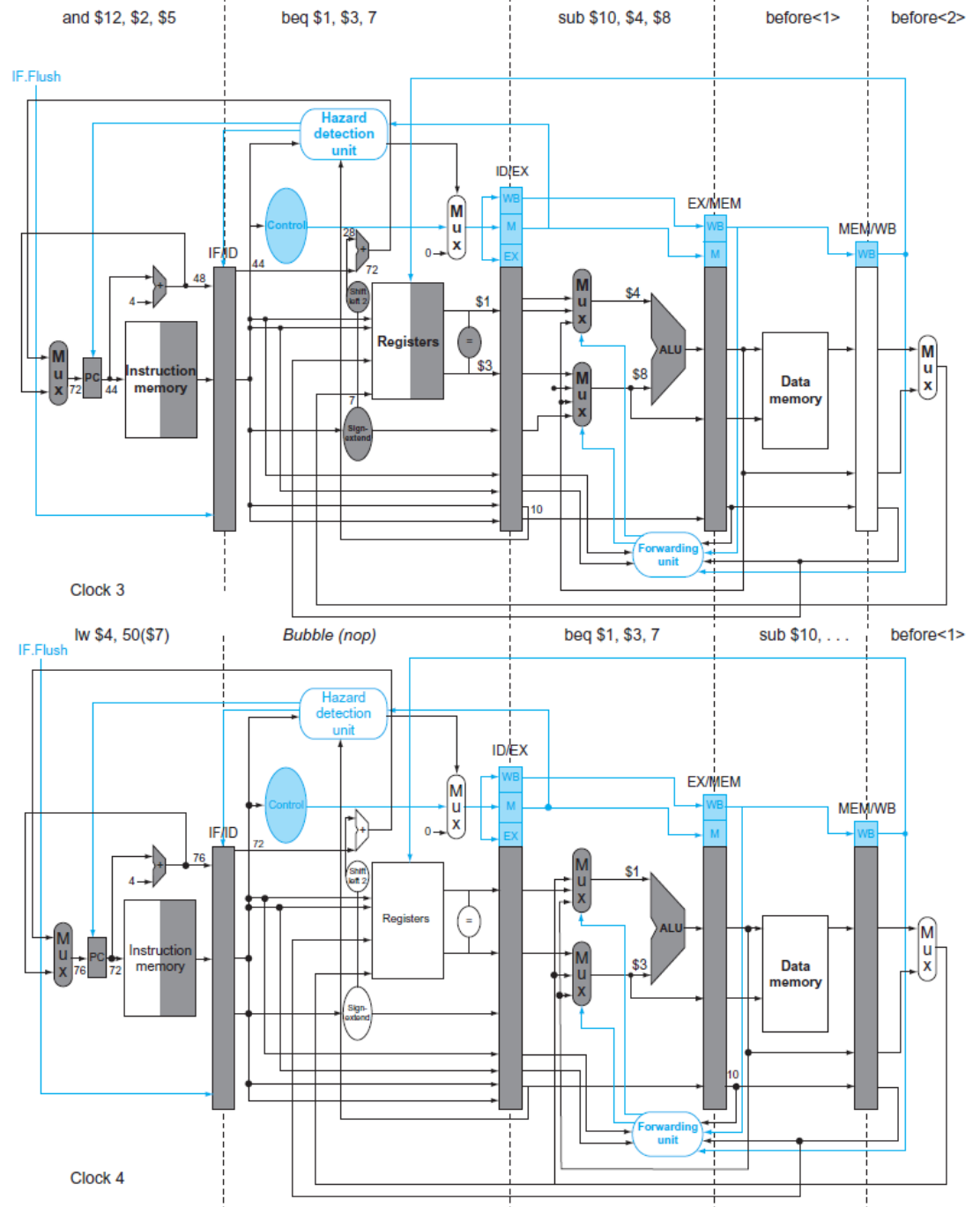
# Solving load-use hazard



# Solving control hazard



# Delayed branching





# Summary

---

- ❑ All modern day processors use pipelining
- ❑ Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- ❑ Potential speedup: a CPI of 1 and a fast CC
- ❑ Must detect and resolve hazards
  - ❑ Stalling negatively affects CPI (makes CPI less than the ideal of 1)

## Example

---

- ❑ Detect stall in the code snippets below

```
lw    $t0, 0($t0)           add    $t1, $t0, $t0
add   $t1, $t0, $t0         addi   $t2, $t0, #5
                                addi  $t4, $t1, #5

                                addi  $t1, $t0, #1
                                addi  $t2, $t0, #2
                                addi  $t3, $t0, #2
                                addi  $t3, $t0, #4
                                addi  $t5, $t0, #5
```