# Ripasso concetti base: PIPELINE BASE

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2023/2024

# Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC ← target address or PC + 4

# RISC-V Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

# RISC Pipeline – Stadi

## IF: Istruction Fetch

- IR ← Mem[PC]

Invia il program counter (PC) alla memoria e riceve (fetch) l'istruzione corrente dalla memoria.

- NPC ← PC + 4

Incrementa il PC per indirizzare l'istruzione successiva (istruzioni da 4 bytes e memoria indirizzabile al singolo byte.)

## ID: Istruction Decode

- Opcode, funct, Imm, Rd, Rs1, Rs2,.. ← IR

Decodifica l'istruzione, siccome pochi formati e registri sempre nelle stesse posizioni. Decodifica semplice => In decode si può fare altro.

- A, B ← RegisterFile[Rs1,Rs2]

Leggi il contenuto del RegisterfFle a indirizzo Rs1, Rs2

- Imm_se ← SignExt.[Imm]

Estendi il bit di segno del campo immediato: è sempre nella stessa posizione per istruz. di load e alu.

- Cond=? (A == B)

Test di ugualianza tra i dati contenuti nei due registri sorgente per possibili istruzioni branch.

- NPC = NPC + Imm_se

Calcola il possibile *branch target address* sommando al PC il campo immediato esteso di segno

# RISC Pipeline – Stadi

## EX: Execution

- ALU_output = A + Imm_se
- ALU_output = alu_op (A, B)
- ALU_output = alu_op (A + Imm_se)
- Cond =? (A == B)
- NPC = NPC + Imm_se

Load: calcola l'*indirizzo effettivo* da cui leggere il dato in memoria sommando il campo immediato al dato letto dal Register File (RF).

ALU reg-reg: Esegue l'operazione ALU sui due valori letti dal Register File

ALU reg-imm: Esegue l'operazione ALU tra immediato e dato letto dal Register File

Branch: Test di ugualinza tra i dati contenuti nei due registri sorgente per possibili istruzioni branch.

Branch: Calcola il possibile *branch target address* sommando al PC il campo immediato esteso di segno

## MEM: Memory  access / branch completion:

- PC ← NPC
- LMD ← Mem[ALU_output]
- Mem[ALU_output] ← B

Branch: Aggiorna il  program counter per puntare alla prossima istruzione.

Load: Legge il dato contenuto in memoria all'indirizzo effettivo contenuto in ALU_output e salva il valore nel *load memory data (LMD).*

Store: Scrive in memoria all'*indirizzo effettivo* contenuto in ALU_output il dato letto dal RegisterFile.

# RISC Pipeline – Stadi

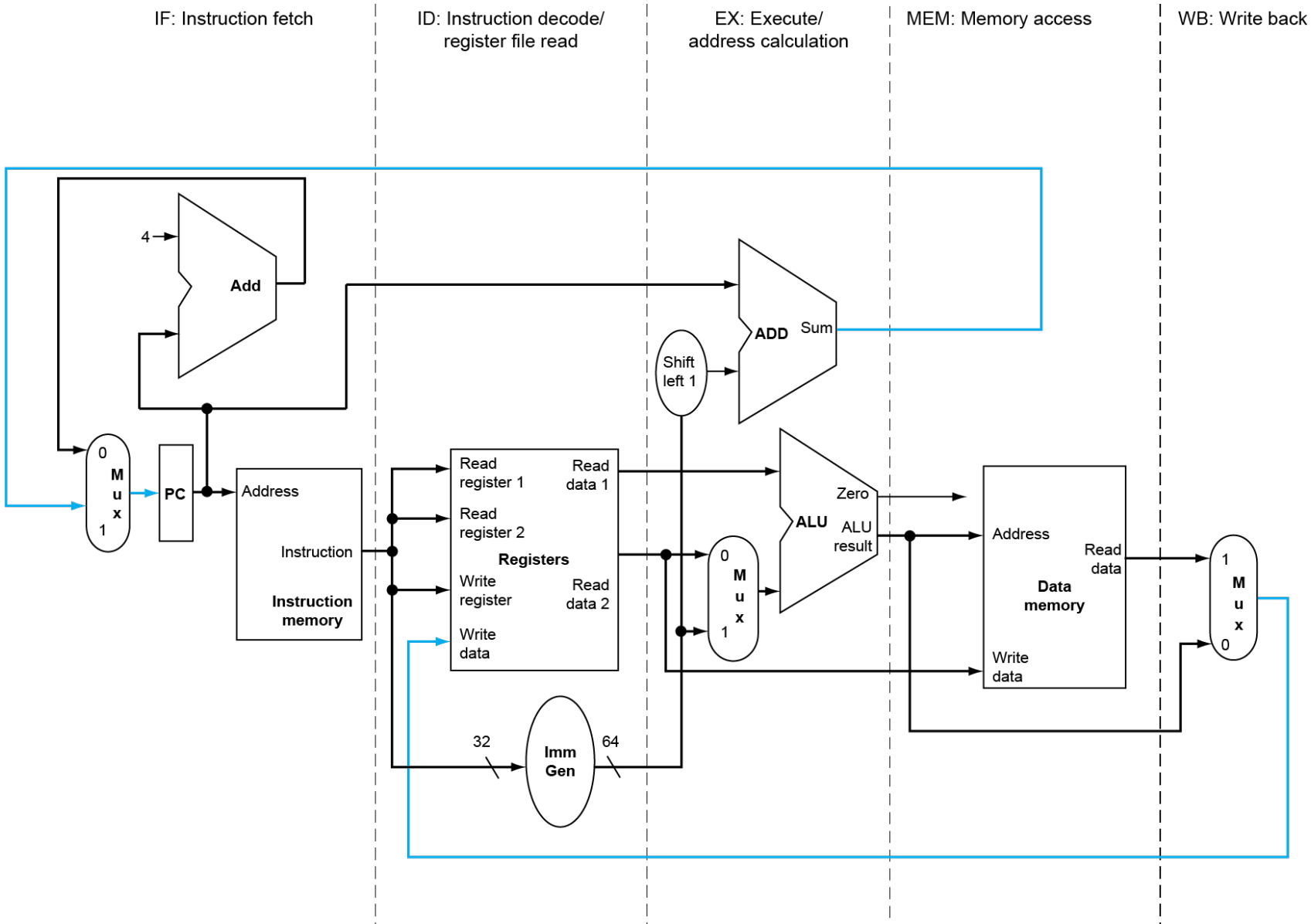
WB: Write Back

- RegisterFile[Rd] ← ALU_output
- RegisterFile[Rd] ← LMD

ALU reg-reg: memorizza il risultato dell'operazione nel Register File

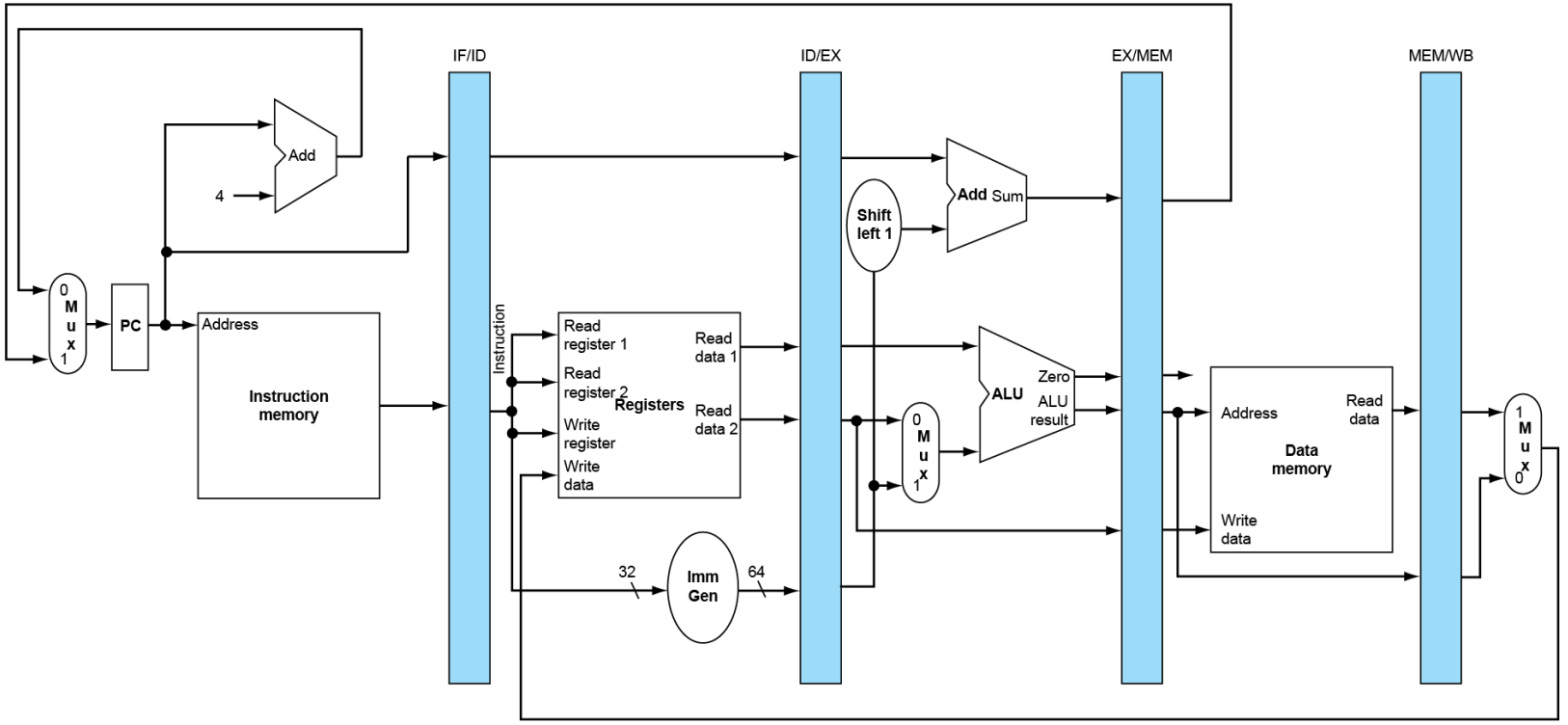
Load: Salva il dato letto nella memoria nel Register File

# RISC-V Pipelined Datapath



# Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle



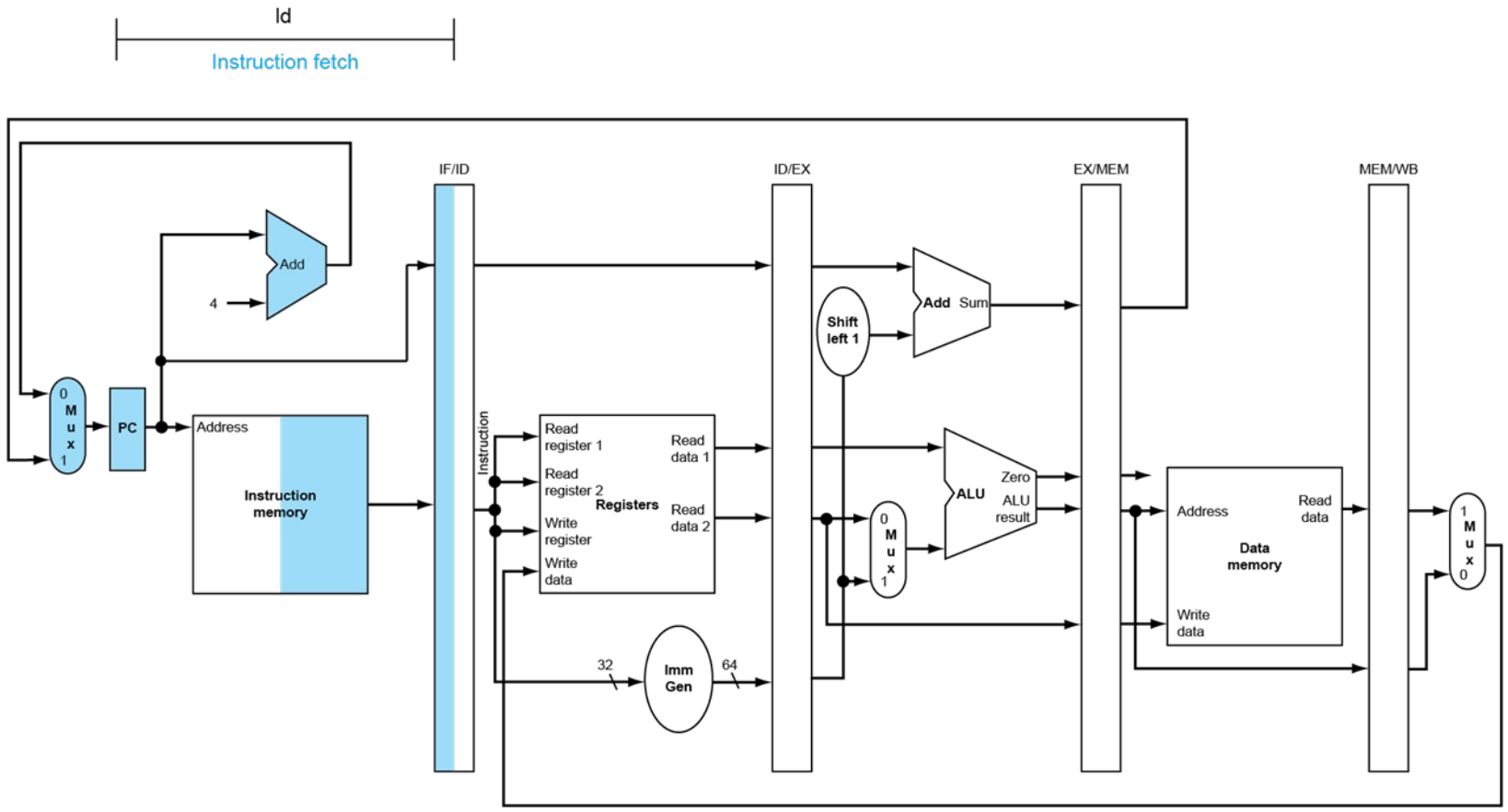
# Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $\textit{Time between instructions}_{pipelined} = \dfrac{\textit{Time between instructions}_{nonpipelined}}{\textit{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

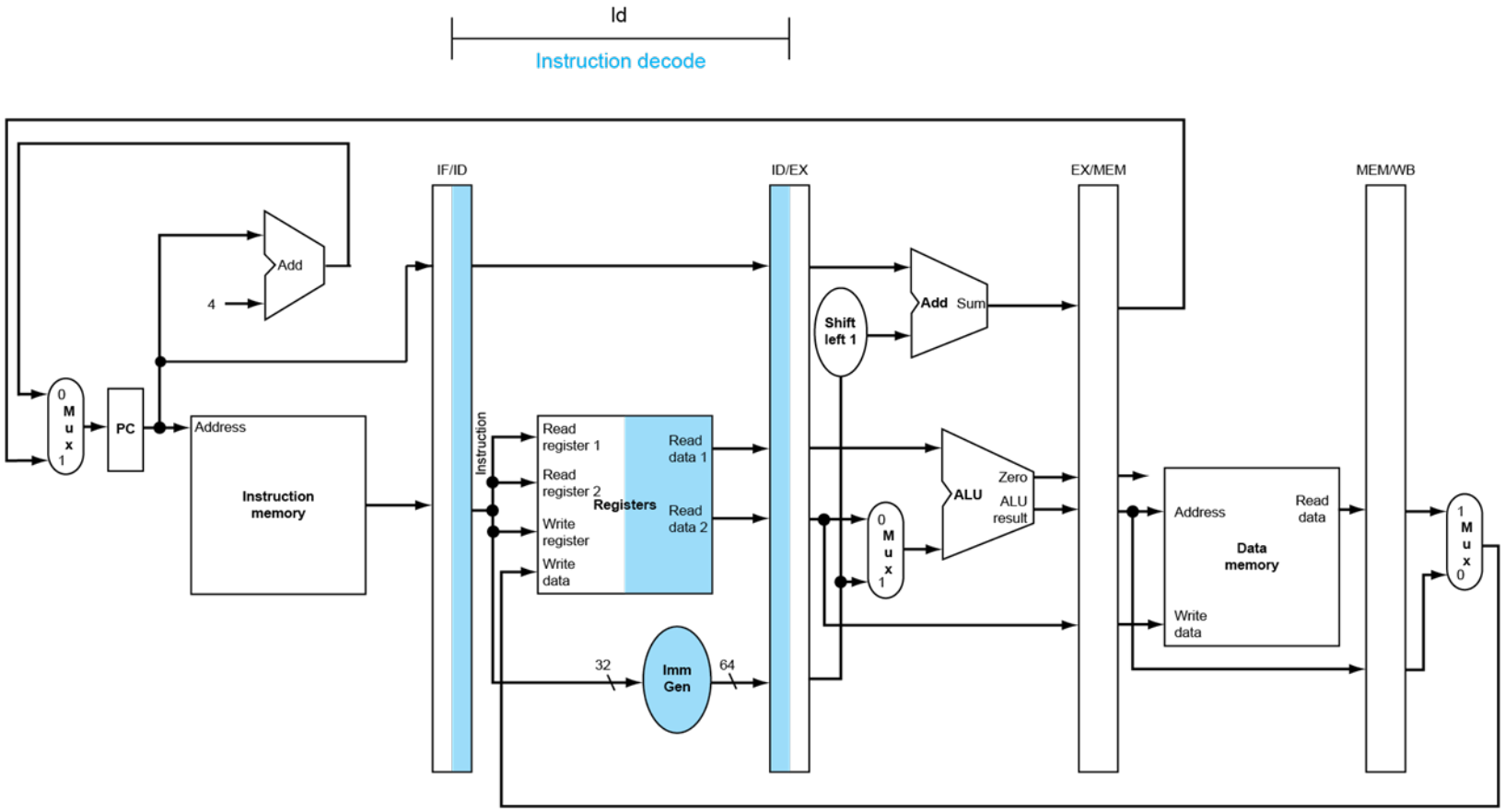
# Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3$^{rd}$ stage, access memory in 4$^{th}$ stage

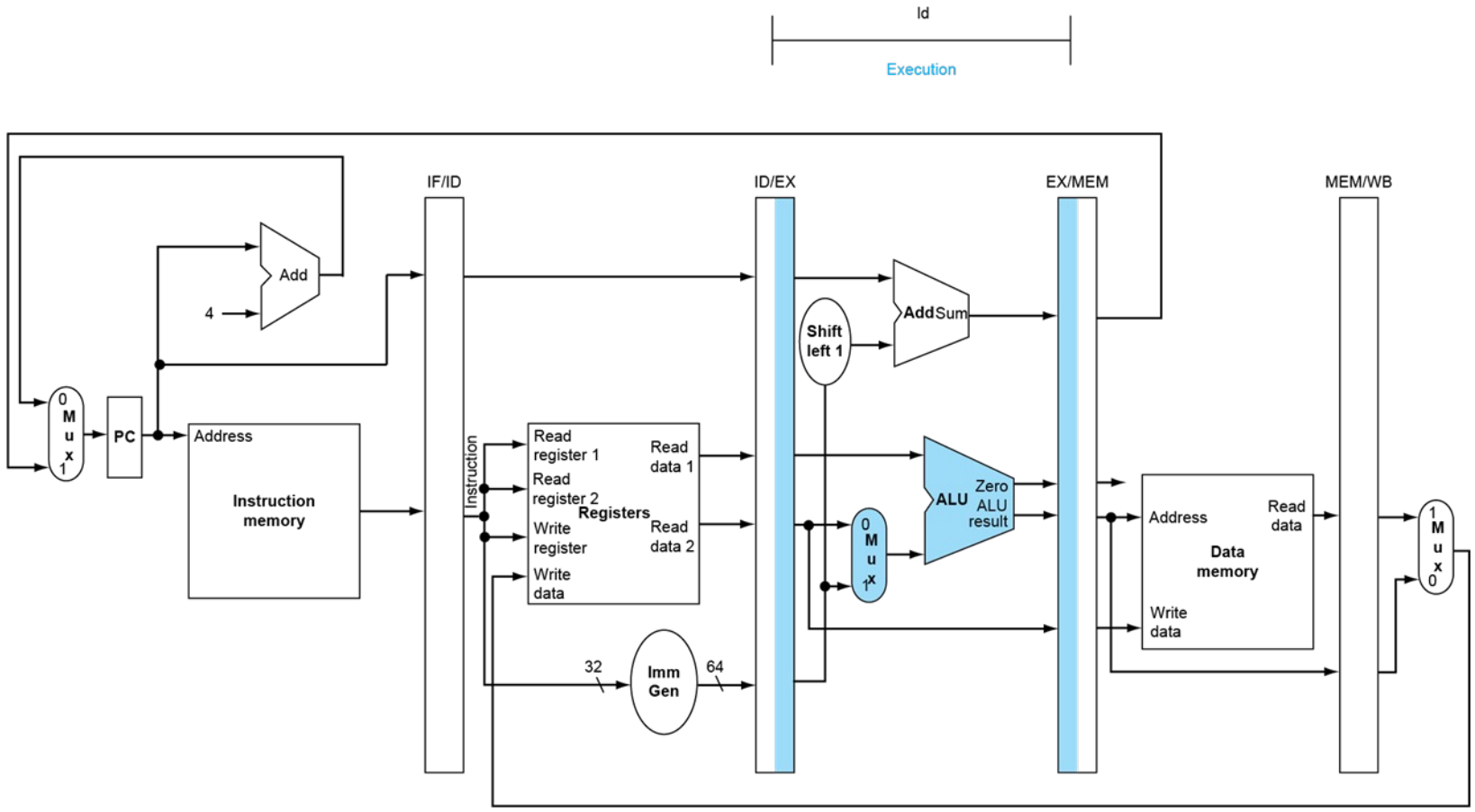
# IF for Load, Store, …



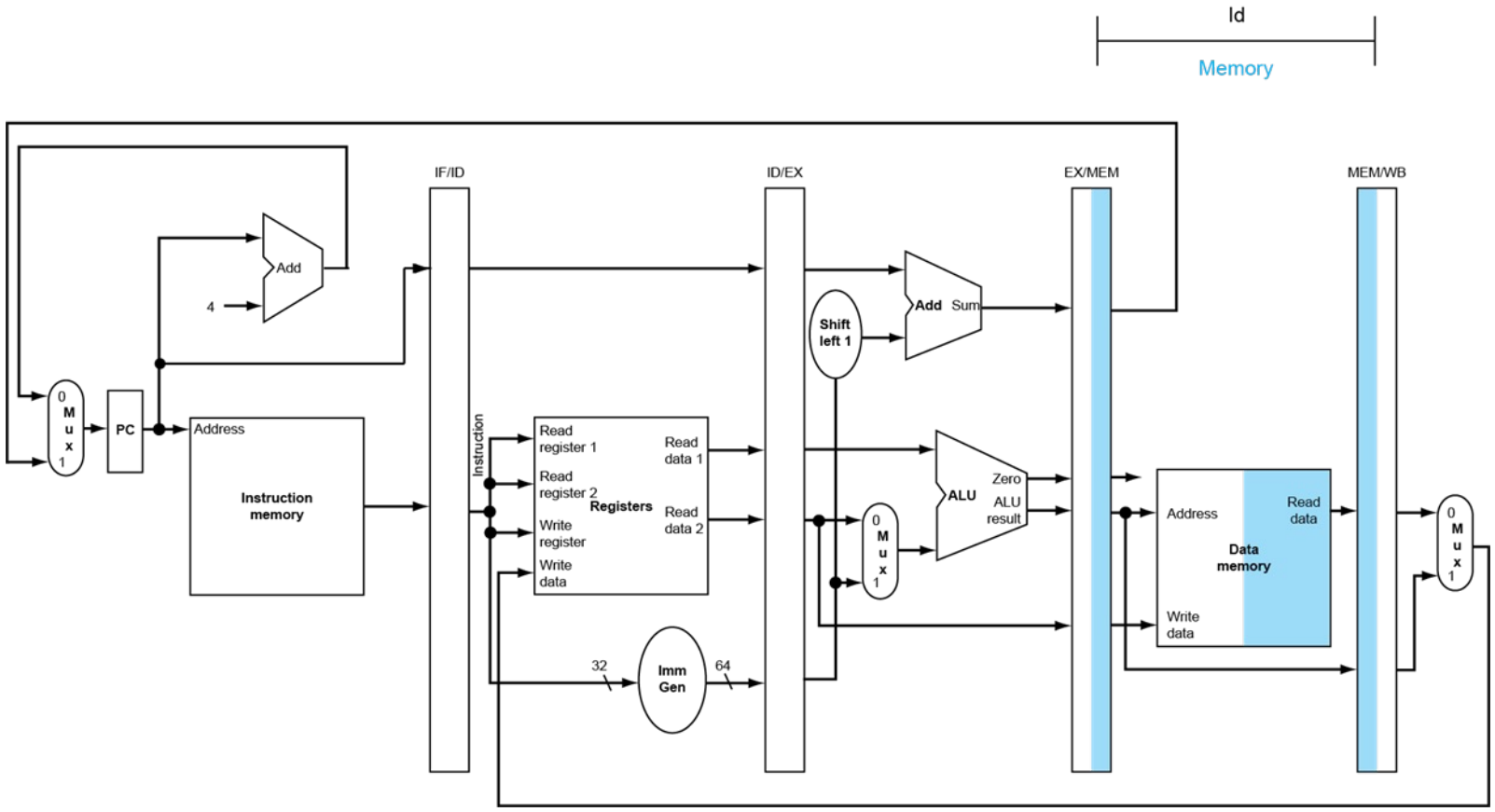
# ID for Load, Store, ...



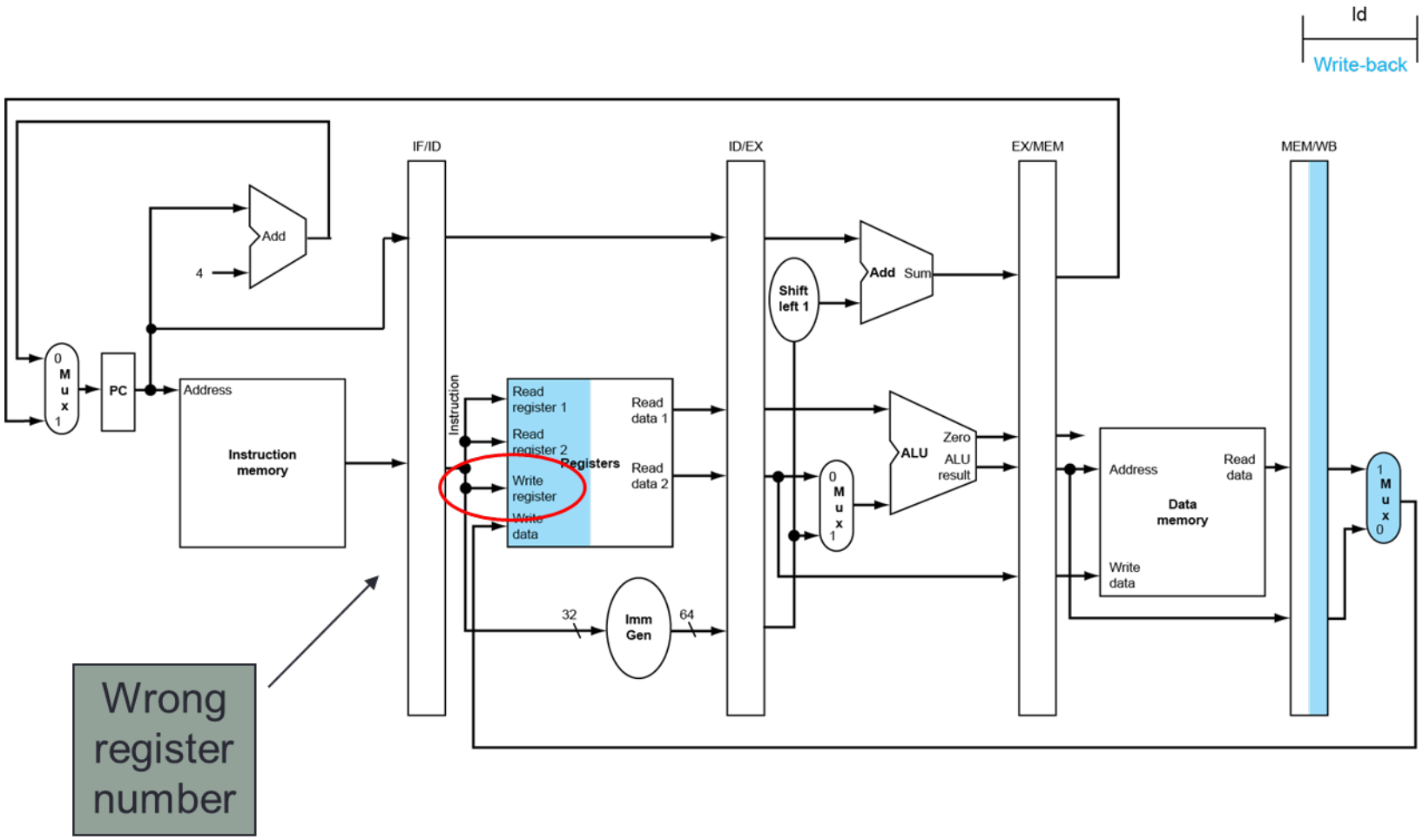
# EX for Load


ld
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add
Sum
0
M
u
x
1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0

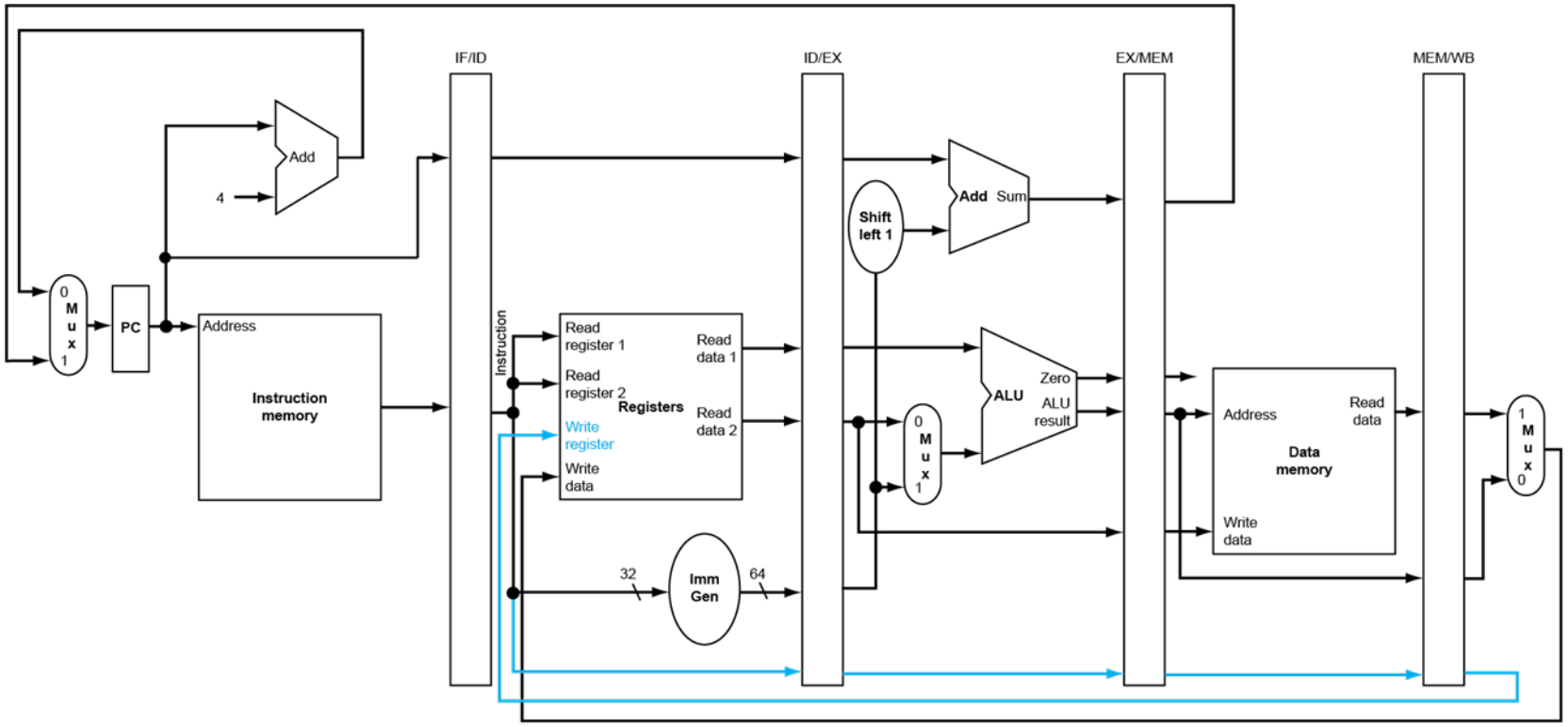
# MEM for Load


ld
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add Sum
ALU
Zero
ALU result
32
Imm Gen
64
Address
Data memory
Read data
Write data

# WB for Load

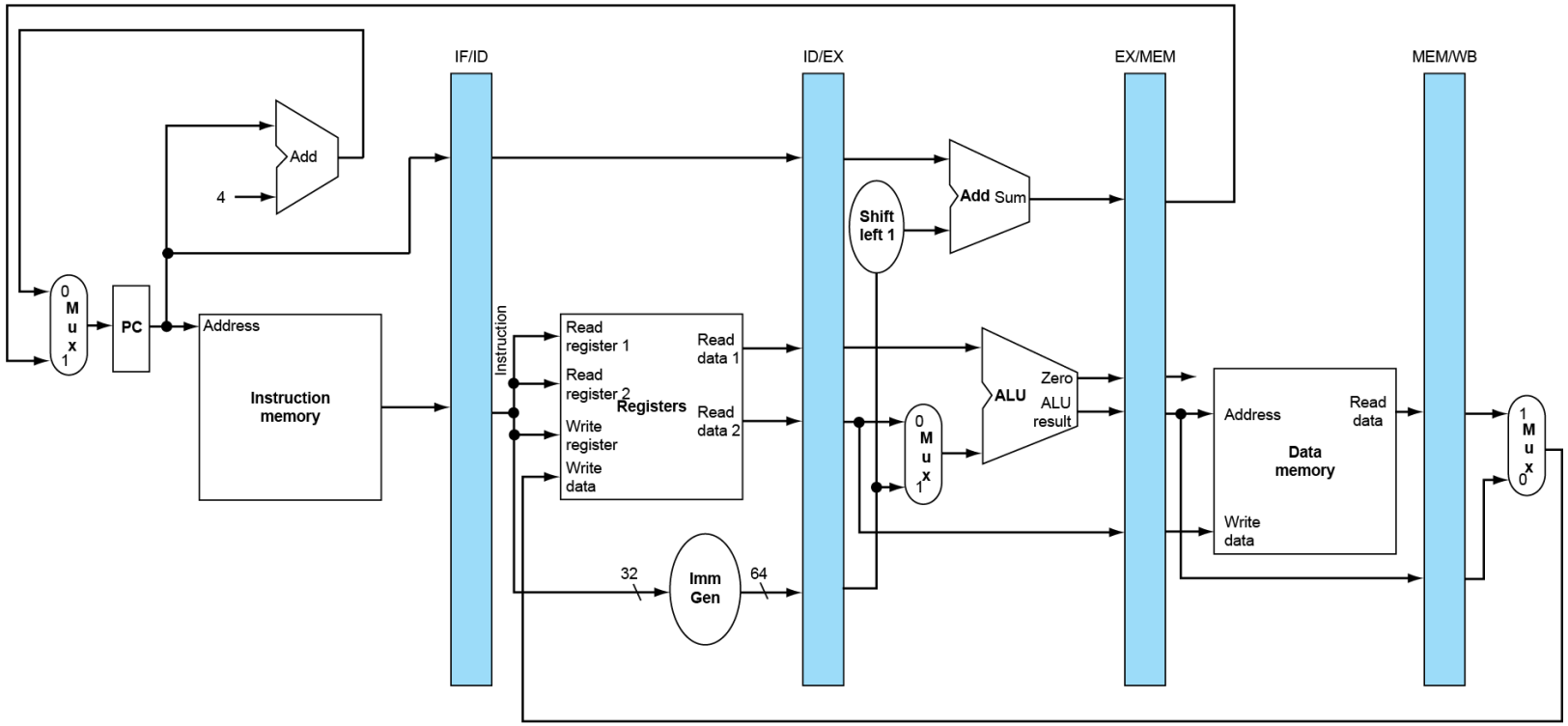

ld
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add
Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0
Wrong register number

# Corrected Datapath for Load



# Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle
  - Progress the instruction fields with the pipeline



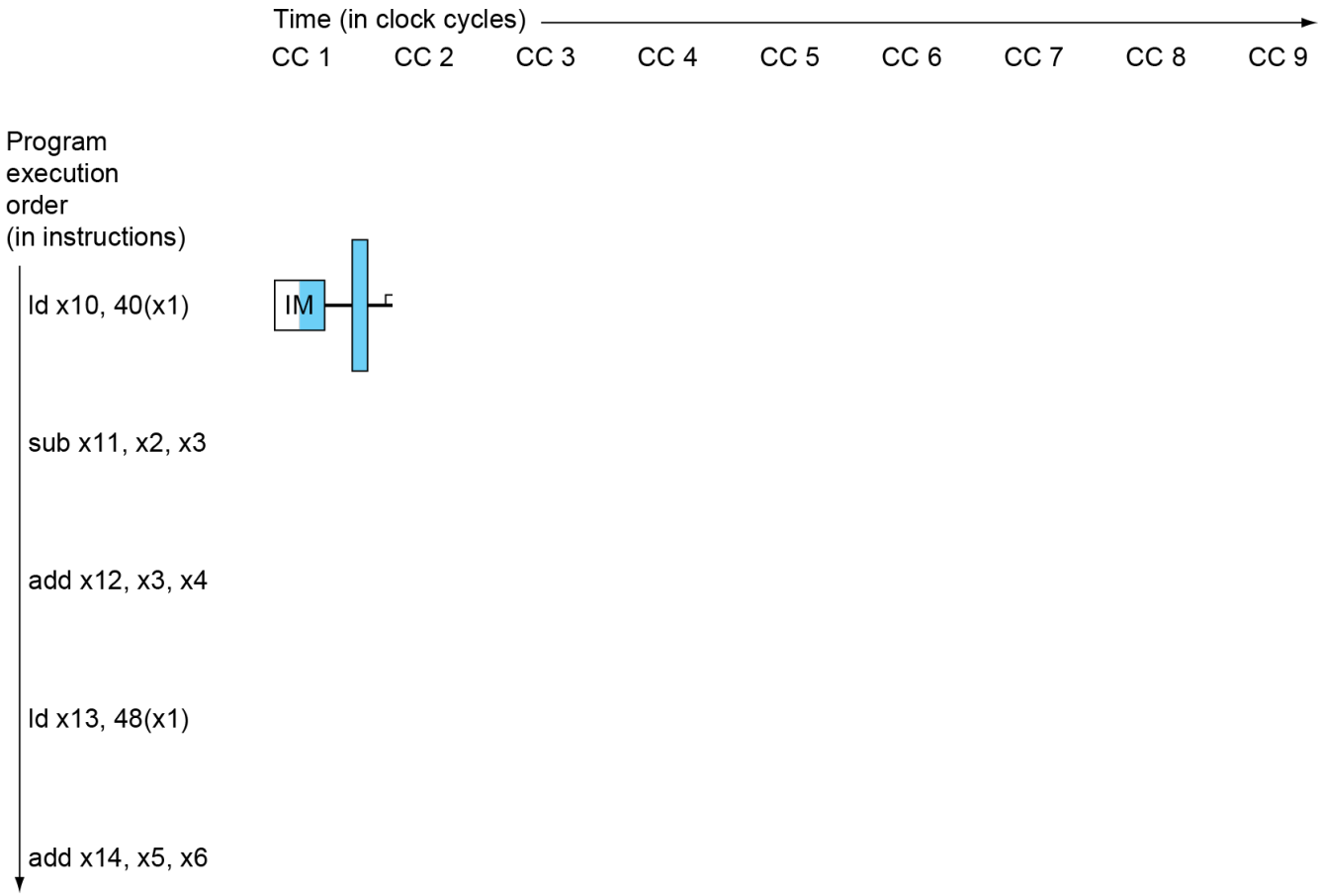
| Stage | Any instruction | | |
|---|---|---|---|
| IF | IF/ID.IR ← Mem[PC]<br>IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4}); | | |
| ID | ID/EX.A ← Regs[IF/ID.IR[rs1]]; ID/EX.B ← Regs[IF/ID.IR[rs2]];<br>ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;<br>ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]); | | |
| | **ALU instruction** | **Load instruction** | **Branch instruction** |
| EX | EX/MEM.IR ← ID/EX.IR;<br>EX/MEM.ALUOutput ←<br>ID/EX.A func ID/EX.B;<br>or<br>EX/MEM.ALUOutput ←<br>ID/EX.A *op* ID/EX.Imm; | EX/MEM.IR to ID/EX.IR<br>EX/MEM.ALUOutput ←<br>ID/EX.A+ID/EX.Imm;<br><br>EX/MEM.B ← ID/EX.B; | EX/MEM.ALUOutput ←<br>ID/EX.NPC +<br>(ID/EX.Imm << 2);<br><br>EX/MEM.cond ←<br>(ID/EX.A == ID/EX.B); |
| MEM | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.ALUOutput ←<br>EX/MEM.ALUOutput; | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.LMD ←<br>Mem[EX/MEM.ALUOutput];<br>or<br>Mem[EX/MEM.ALUOutput] ←<br>EX/MEM.B; | |
| WB | Regs[MEM/WB.IR[rd]] ←<br>MEM/WB.ALUOutput; | For load only:<br>Regs[MEM/WB.IR[rd]] ←<br>MEM/WB.LMD; | |

# Multi-Cycle Pipeline Diagram

- Form showing resource usage

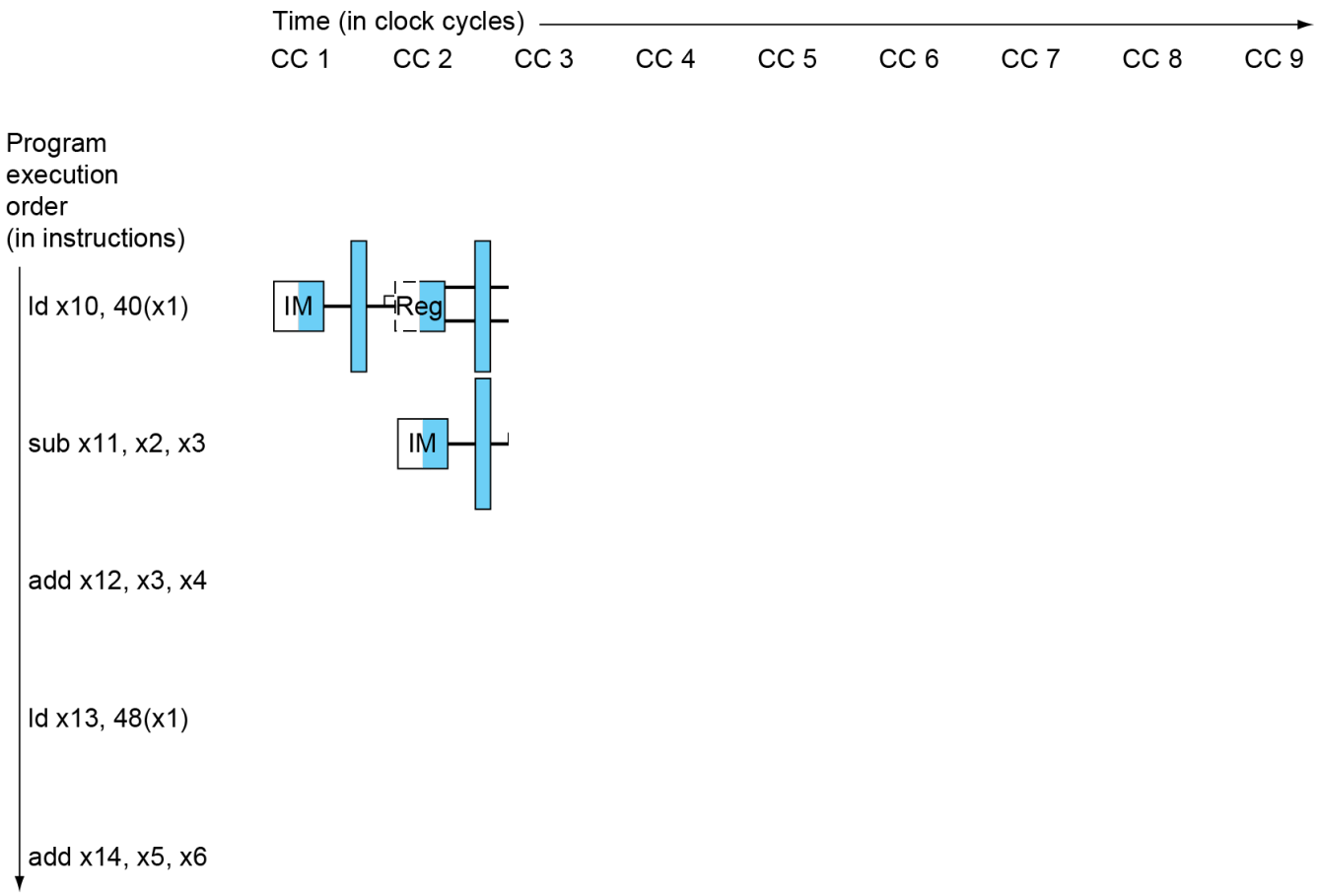
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



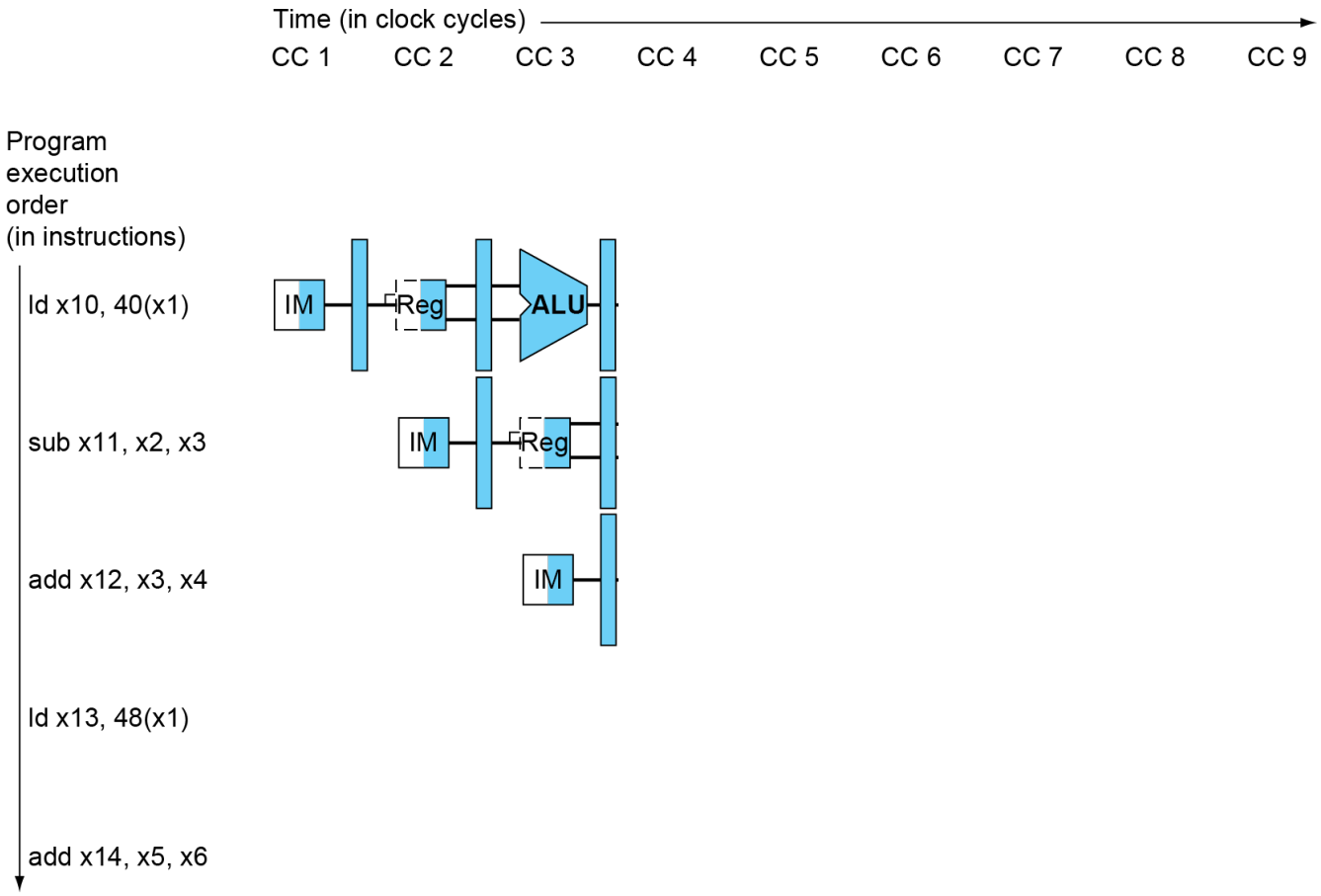
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



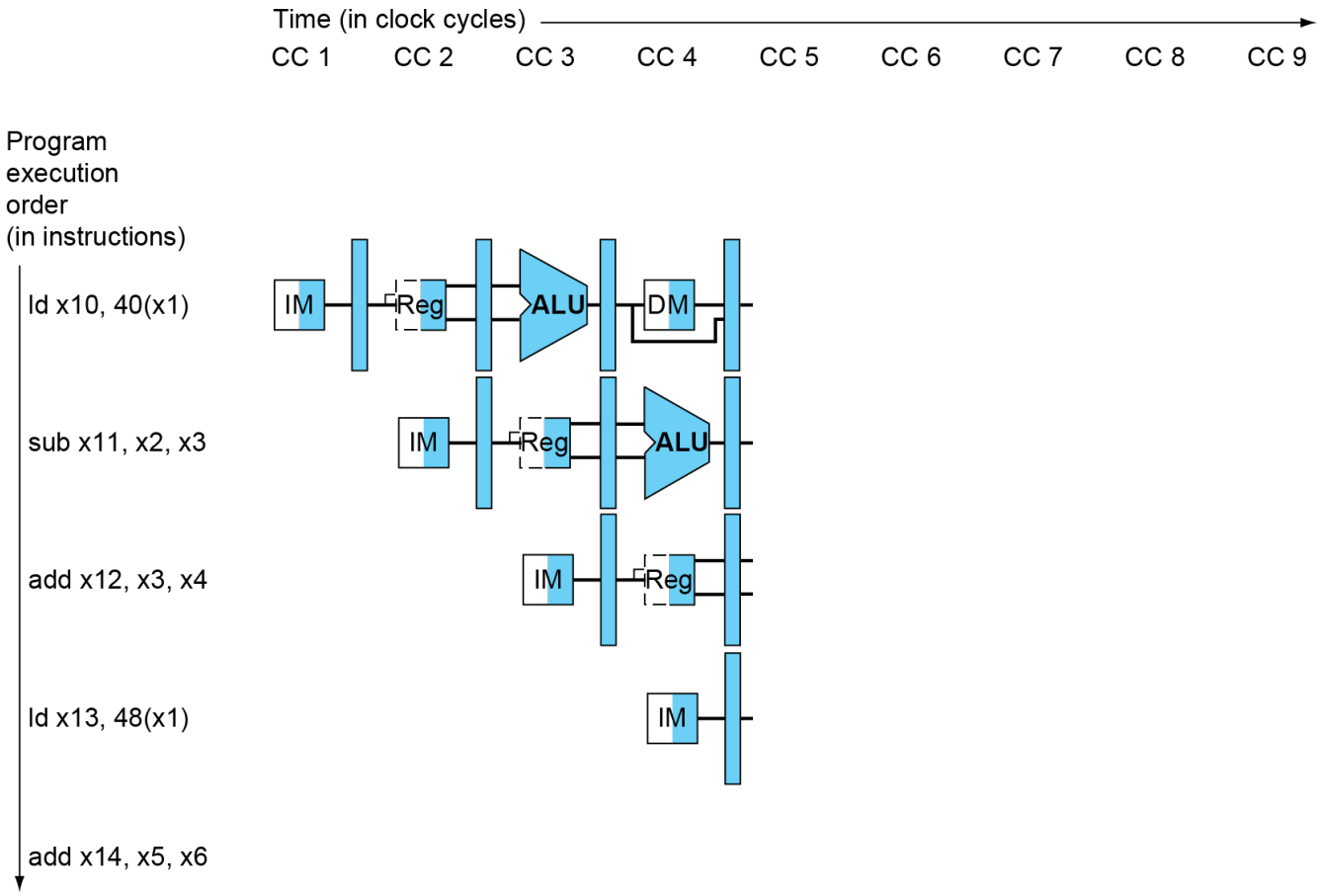
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



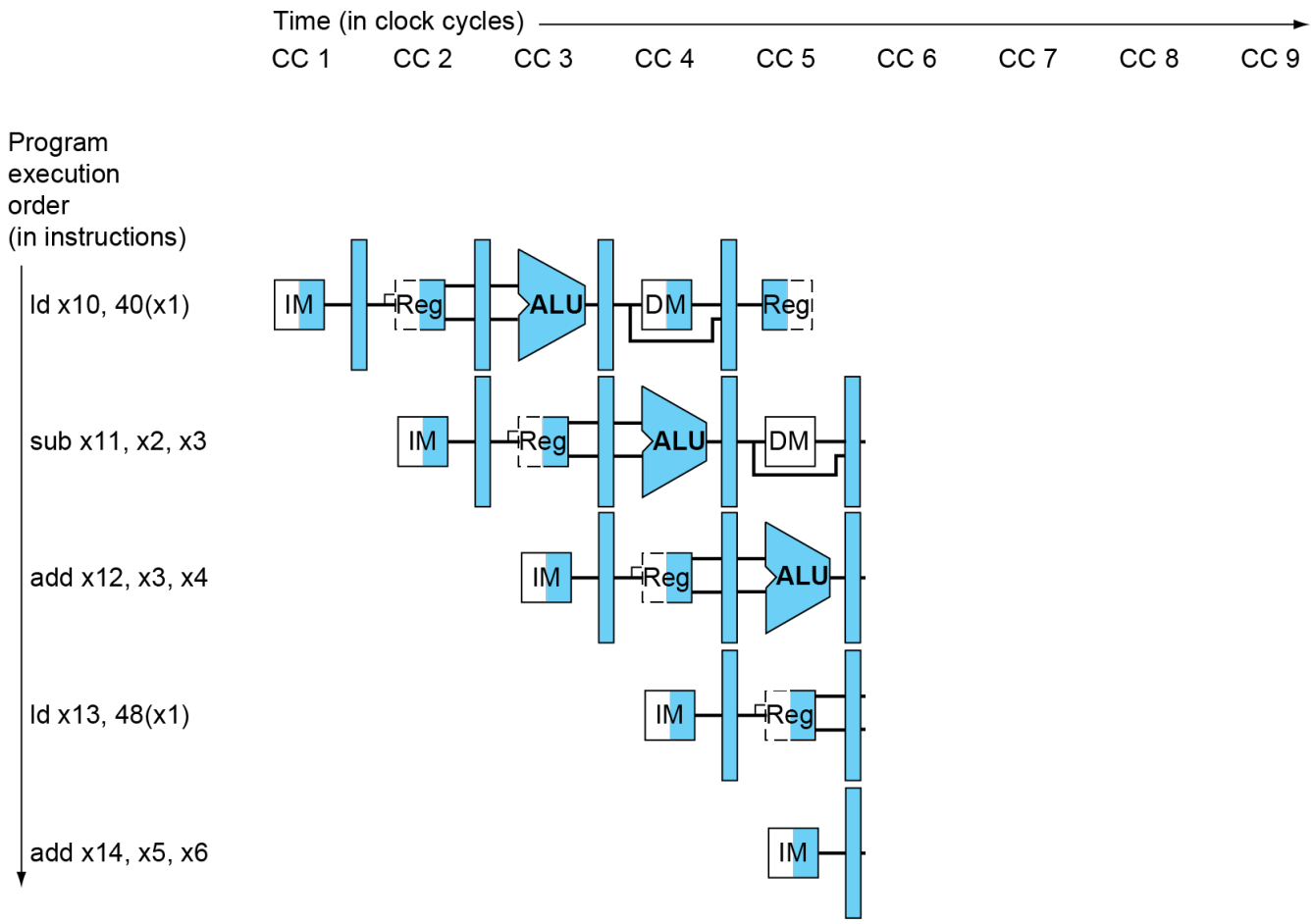
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



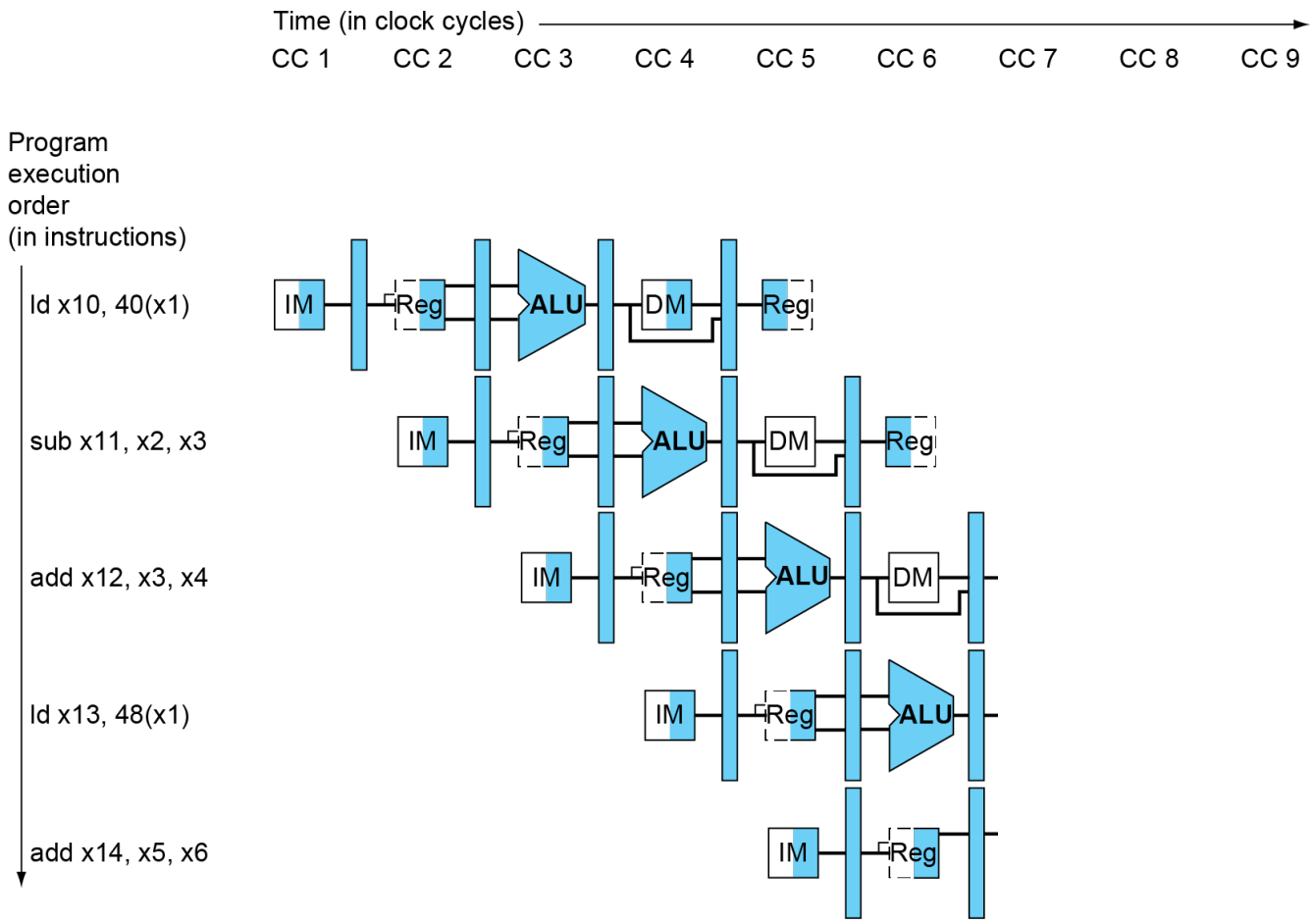
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



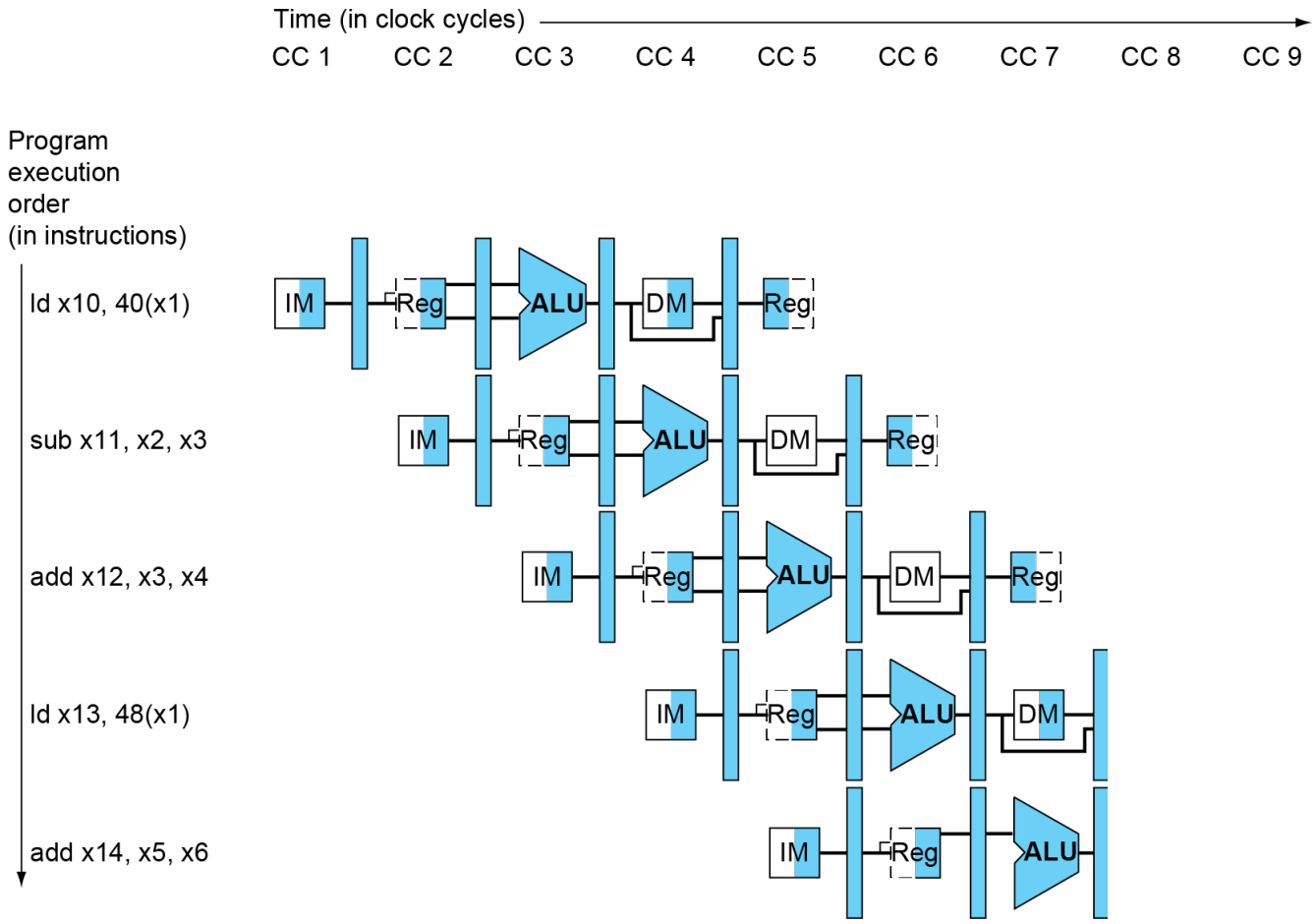
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



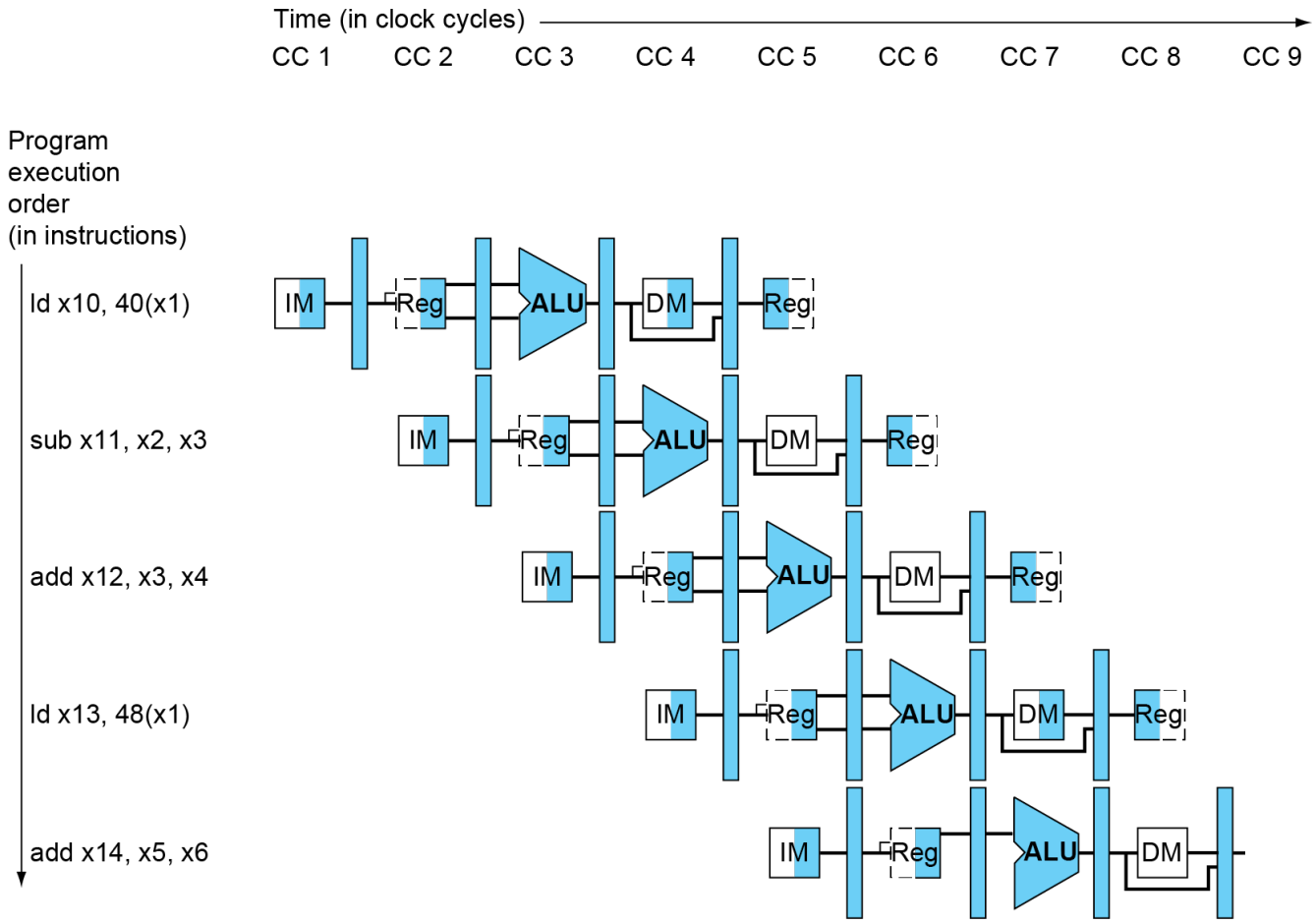
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



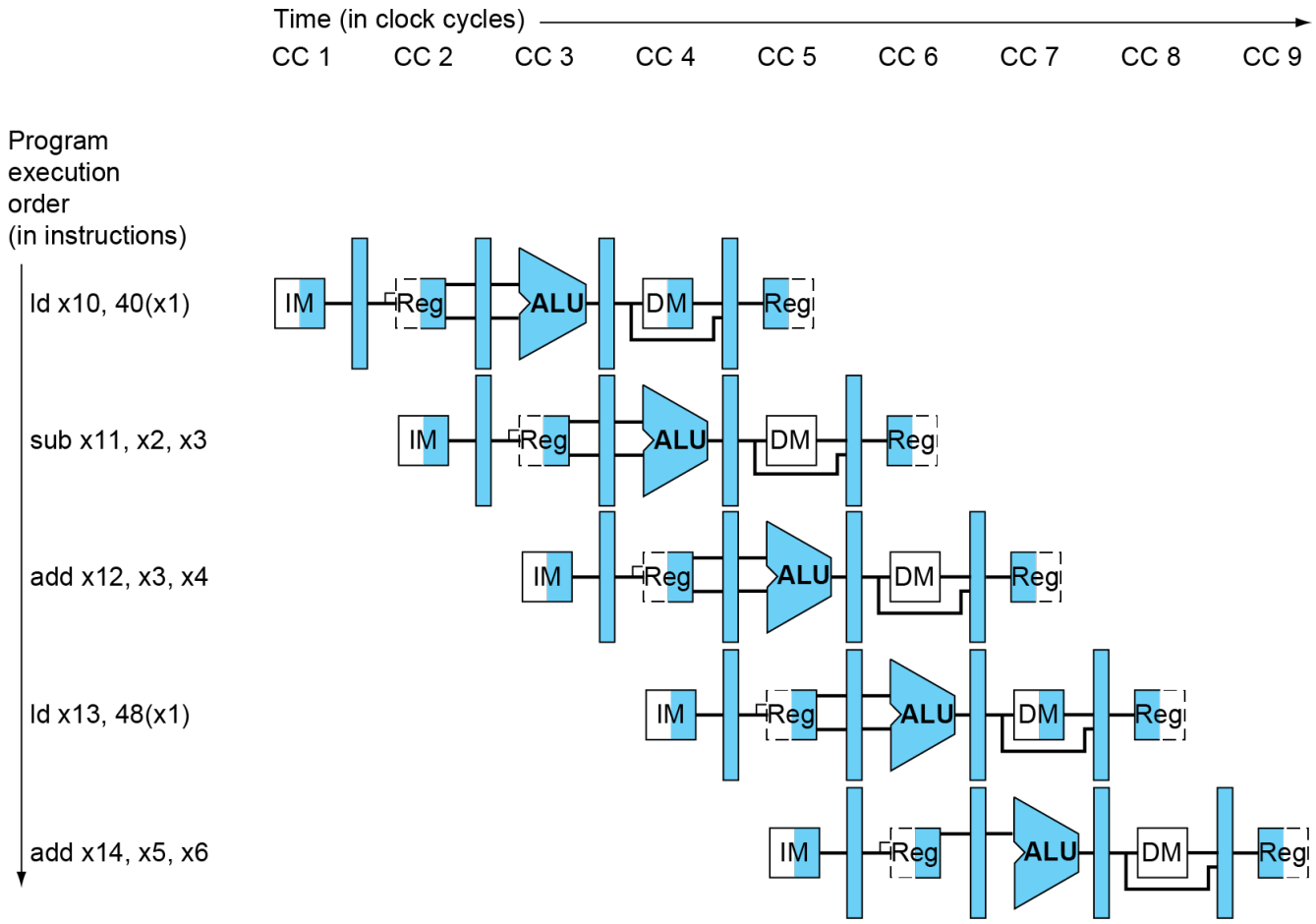
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



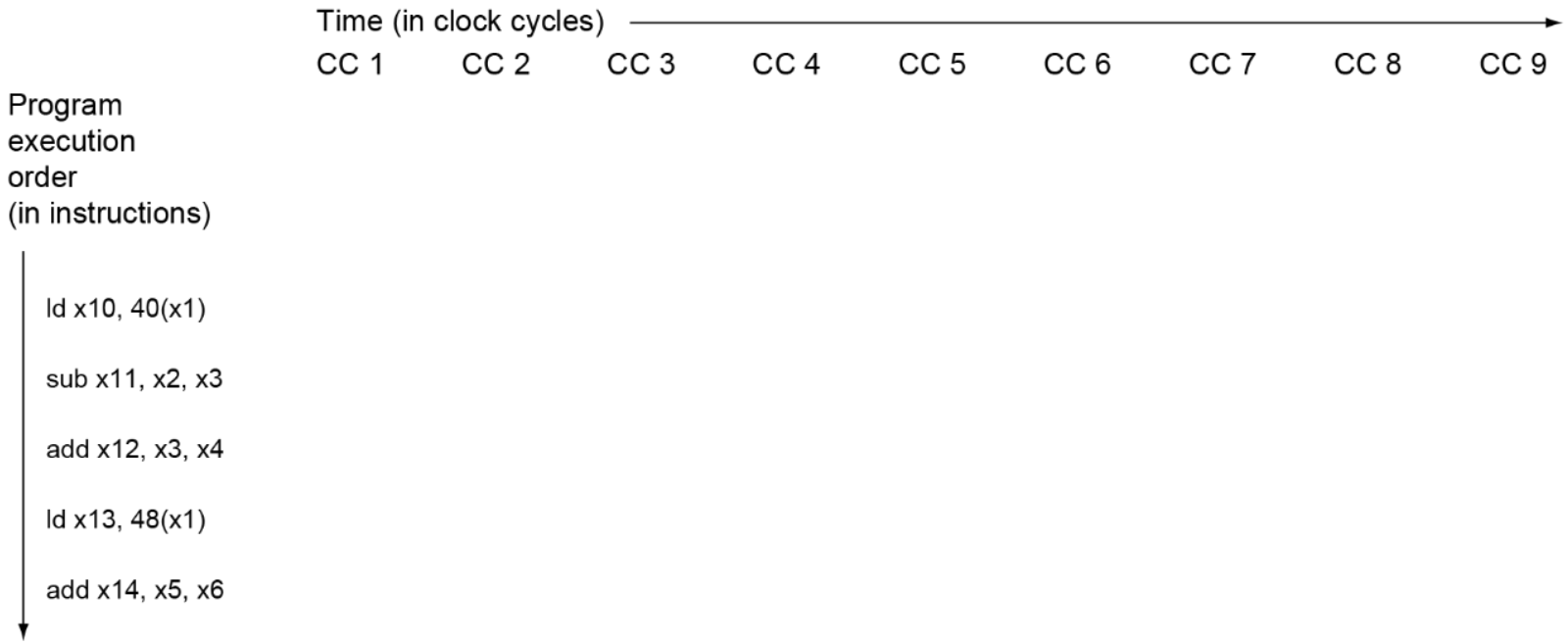
# Multi-Cycle Pipeline Diagram

- Form showing resource usage



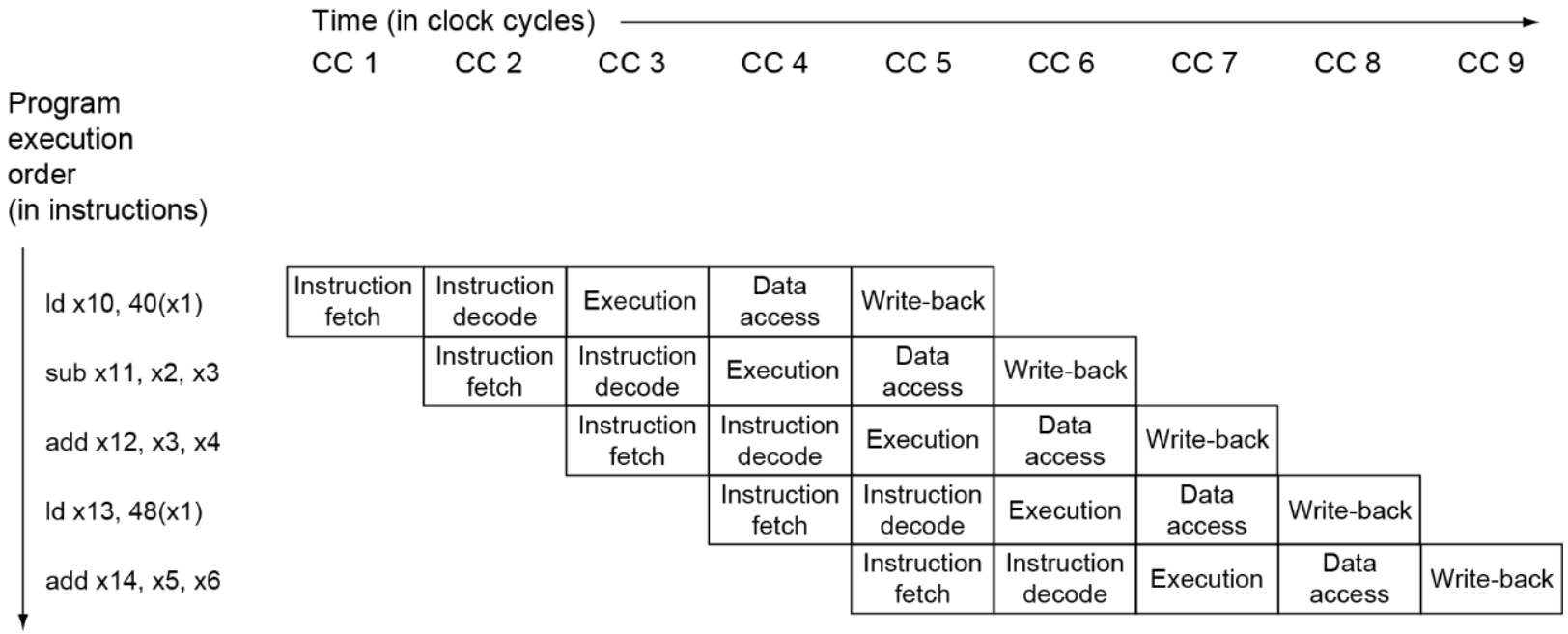
# Multi-Cycle Pipeline Diagram

- Traditional form



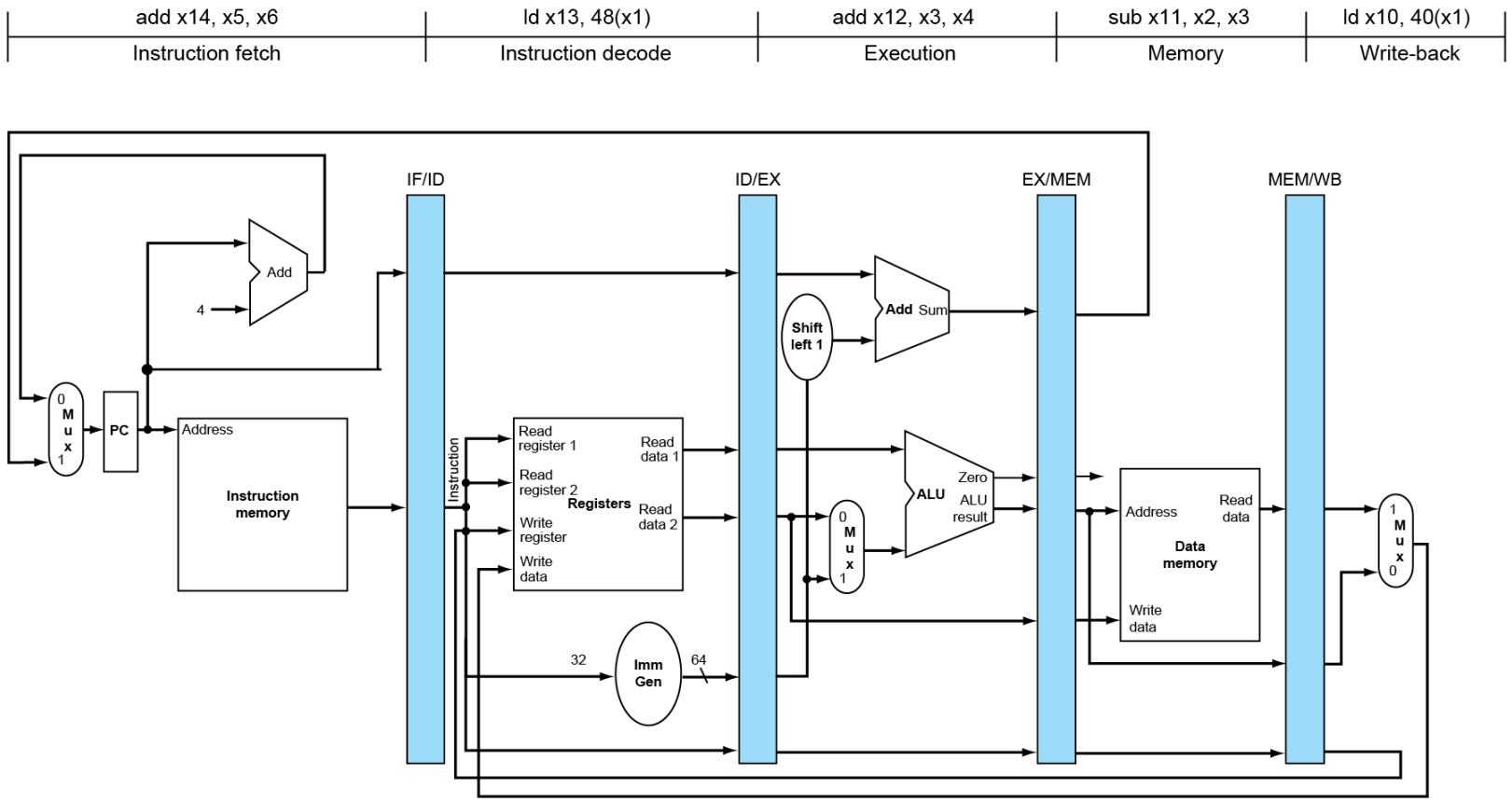
# Multi-Cycle Pipeline Diagram

- Traditional form



# Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle



# Hazards/Alee

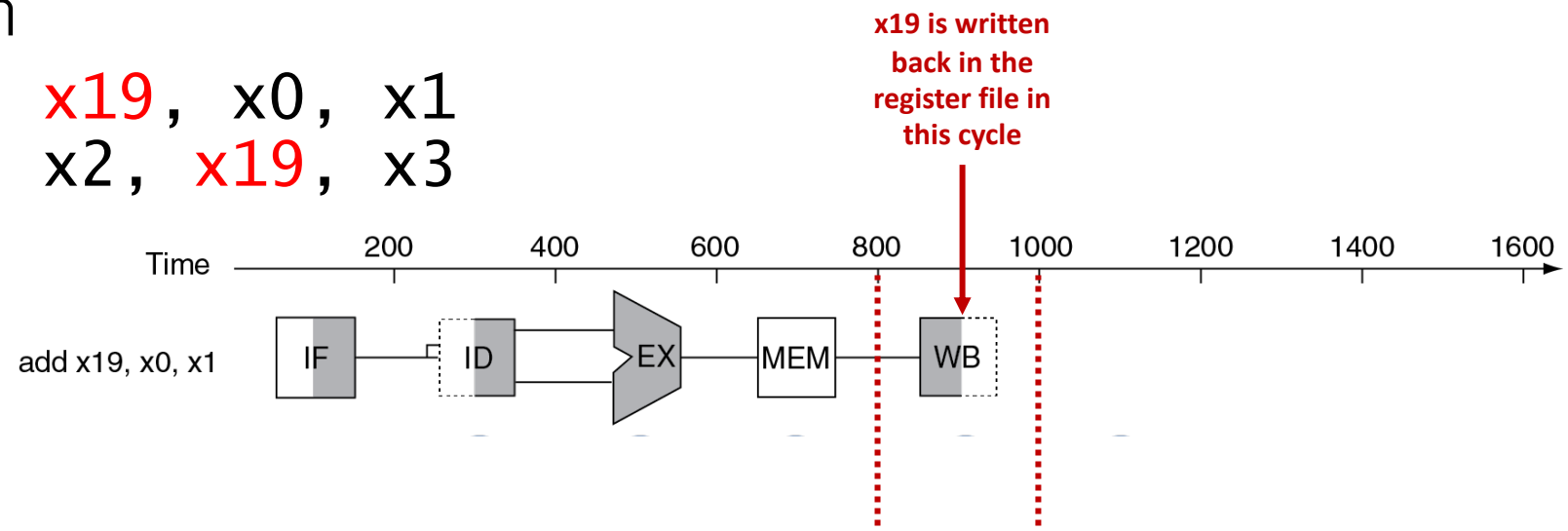
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

# Structural Hazards / Alee Strutturali

- Conflict for use of a resource
- In RISC-V 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 datapaths require separate instruction/data memories
      - Or separate instruction/data caches
  - Register file accessed in ID and WB.
    - Write the Register File on clock's negative edges (write on clock's first semi-period)
    - Read the Register File on clock's positive edges (read on clock's second semi-period)

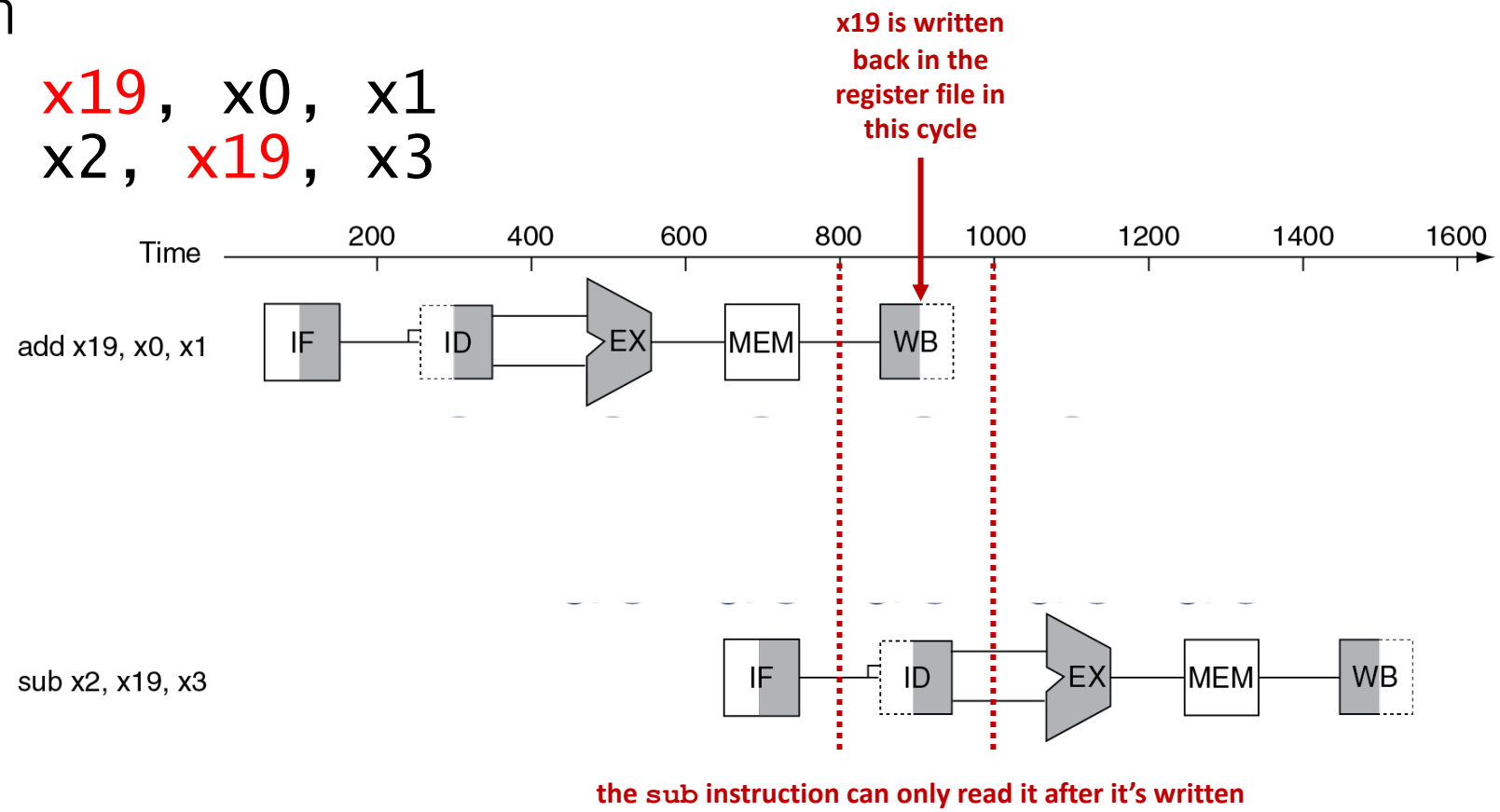
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`



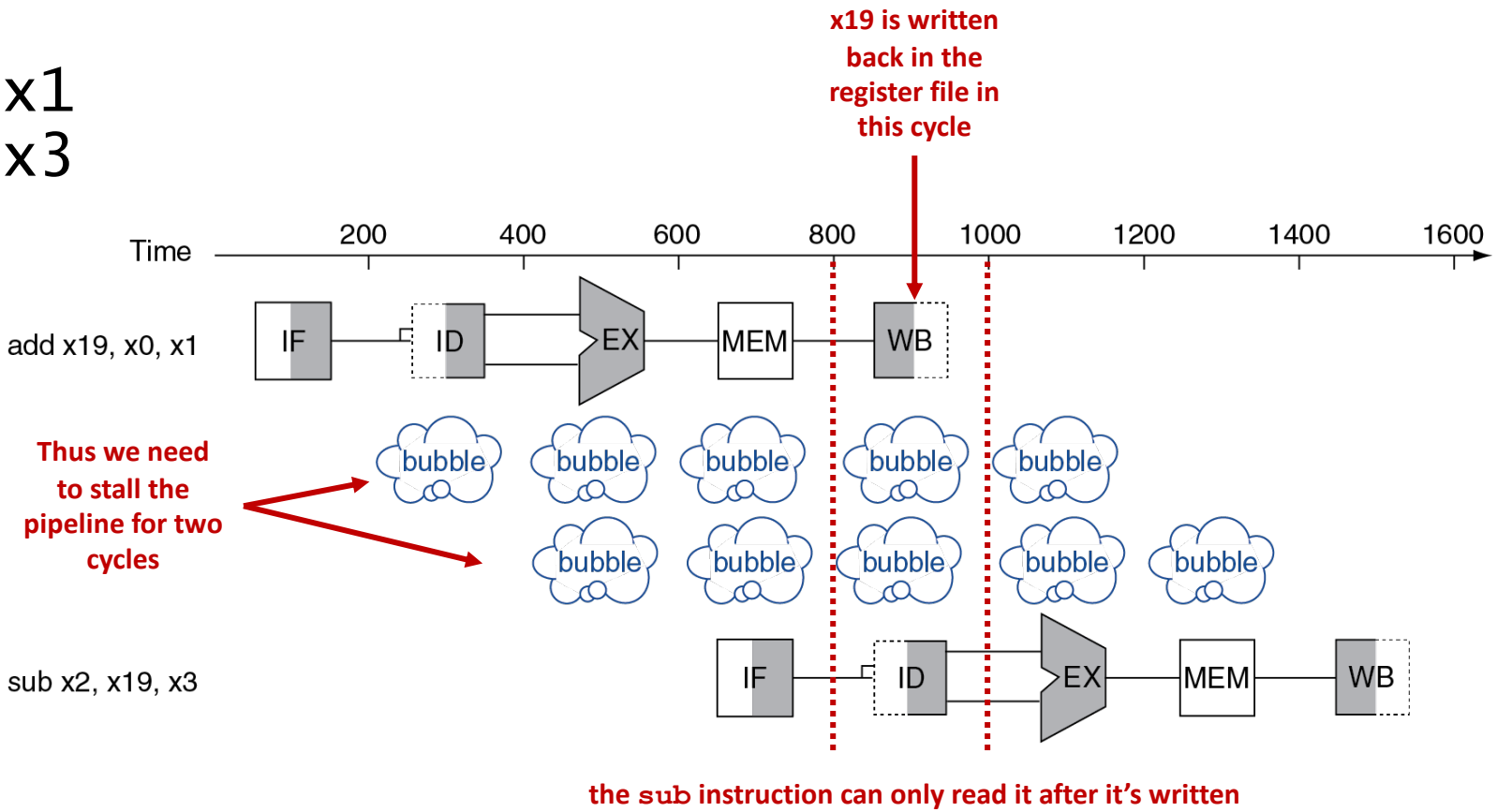
# Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`



# Data Hazards

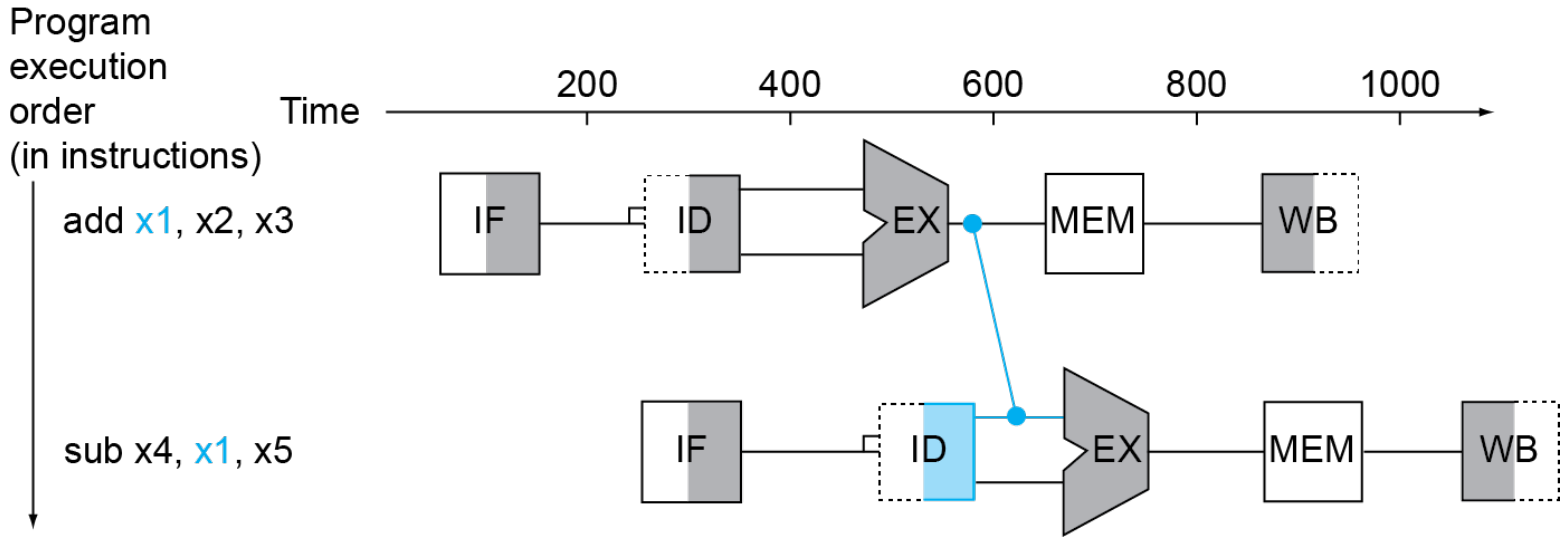
- An instruction depends on completion of data access by a previous instruction
  - `add  x19, x0, x1`
    `sub  x2, x19, x3`




*Pipeline interlock:*
Il componente *pipeline interlock* rileva le alee e stalla la pipeline finchè l'alea non è risolta.
Il CPI dell'istruzione cresce con la lunghezza degli stalli.

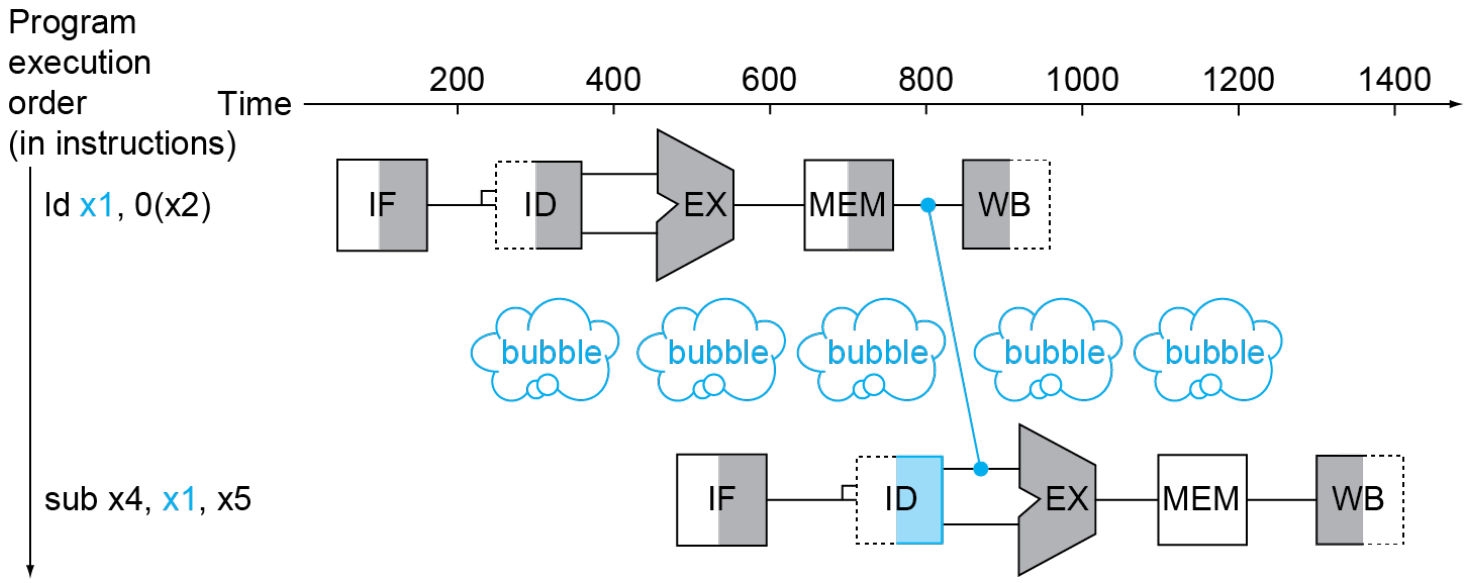
# Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



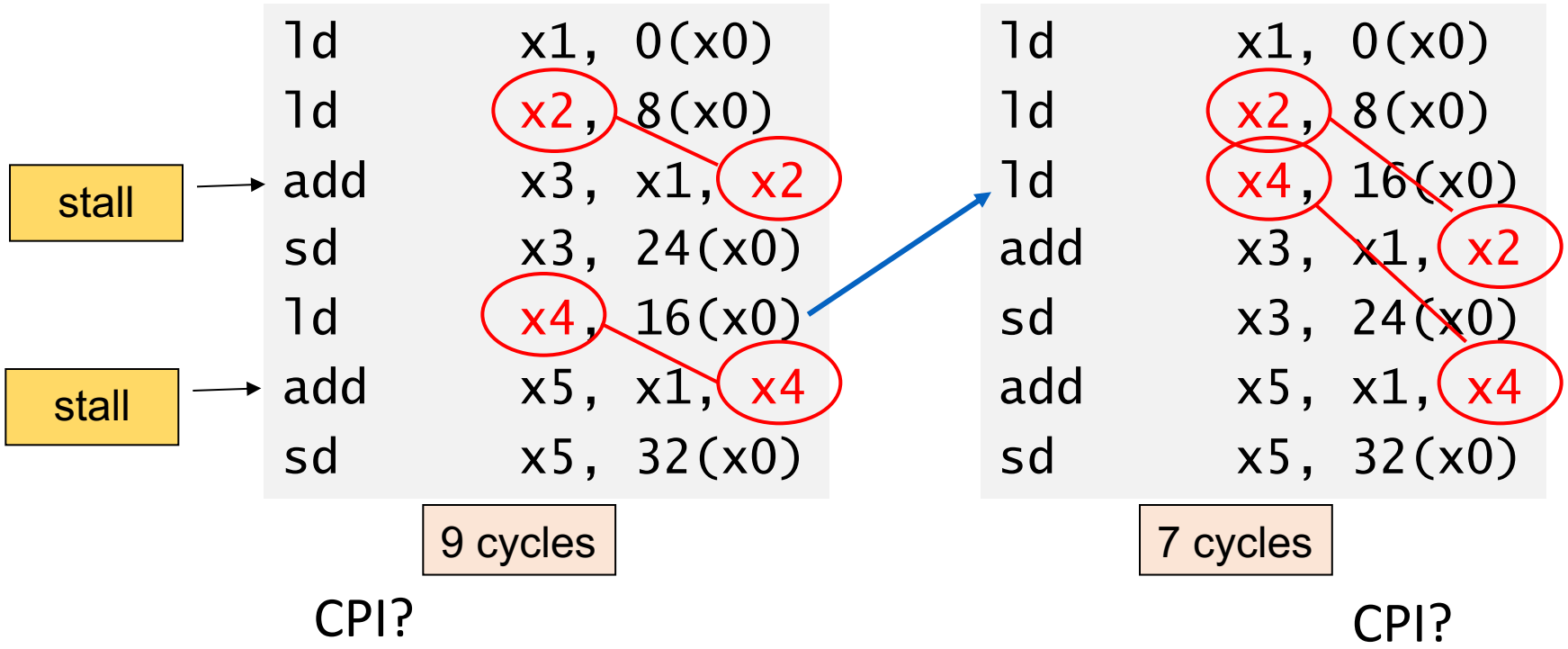
# Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



# Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `a = b + e; c = b + f;`

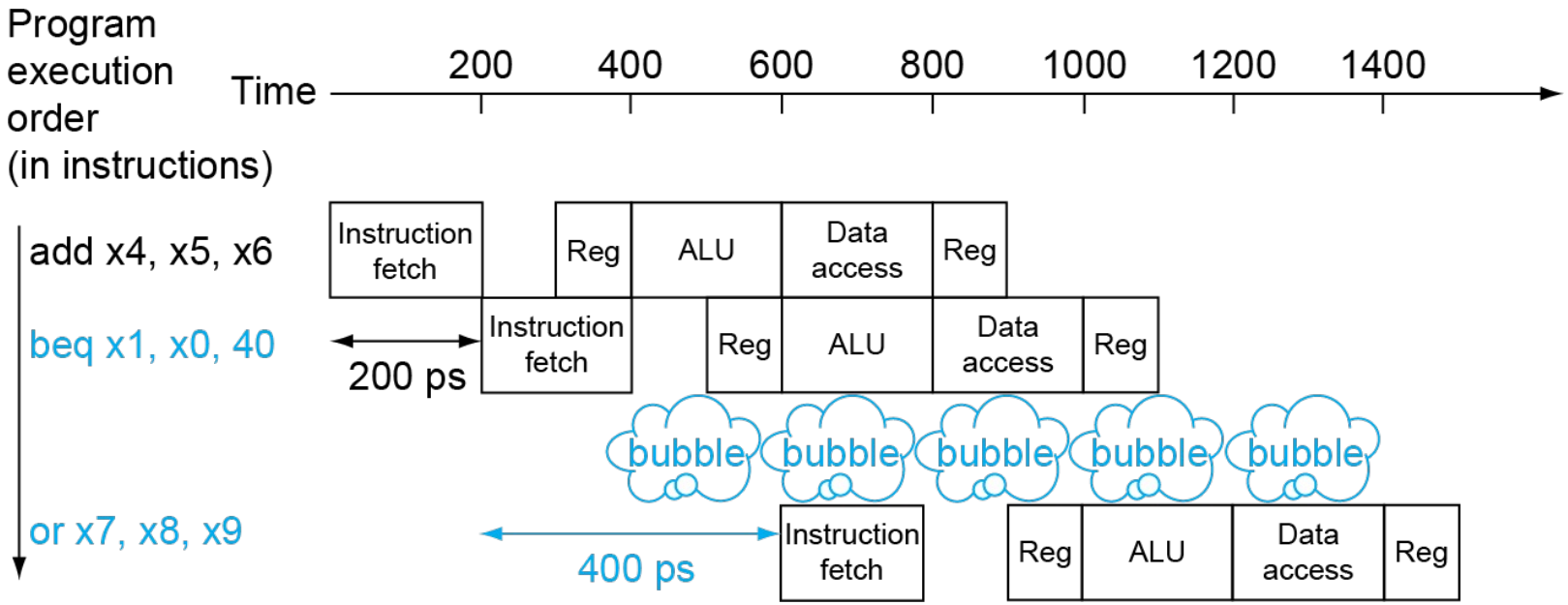


# Control Hazards

- 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 RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
- Countermeasure (assuming branch target address calculation in ID):
  - Static :
    1. Stall on every Branch
    2. Always predict taken or predict untaken
    3. Branch delay slot – not used anymore.
  - Dynamic branch prediction.

# 1.Static: Stall on Branch

- Wait until branch outcome determined before fetching next instruction



# 2. Static: Always predict “taken" or “untaken"

- If prediction wrong need to turn fetched instruction in a no-op to avoid changing the state of the processor.
- Compiler can write code to maximize branch perediction correctness.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Untaken branch instruction | IF | ID | EX | MEM | WB | | | | |
| Instruction $i+1$ | | IF | ID | EX | MEM | WB | | | |
| Instruction $i+2$ | | | IF | ID | EX | MEM | WB | | |
| Instruction $i+3$ | | | | IF | ID | EX | MEM | WB | |
| Instruction $i+4$ | | | | | IF | ID | EX | MEM | WB |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Taken branch instruction | IF | ID | EX | MEM | WB | | | | |
| Instruction $i+1$ | | IF | **idle** | **idle** | **idle** | **idle** | | | |
| Branch target | | | IF | ID | EX | MEM | WB | | |
| Branch target+1 | | | | IF | ID | EX | MEM | WB | |
| Branch target+2 | | | | | IF | ID | EX | MEM | WB |

*Processor State:*
Insieme di registri ed elementi di memoria che determina in modo univoco il comportamento della CPU (i.e. RegisterFile, Mem).

Ogni modifica allo stato del processore è irreversibile.

# Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# More-Realistic Branch Prediction

- 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

# Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

# Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per InClockstruction}$$
$$\text{CPU Time}_{NO\ STALLS} = \text{Instruction Count} \times \text{CPI}_{min} \times \text{Cycle Time}$$
$$\text{CPU Time}_{WITH\ STALLS} = \text{Instruction Count} \times (\text{CPI}_{min} + \#Stalls) \times \text{Clock Cycle Time}$$

- Given an ISA and a Program the number of Stalls (#Stalls) depends on the micro-architecture.
  - Branch resolved at MEM or ID?
  - Forwarding unit or interlock unit?

# Schema per ridurre il costo di branch