Ripasso concetti base: PIPELINE BASE

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2023/2024

Instruction Execution

- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ 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 $\leftarrow$ 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.
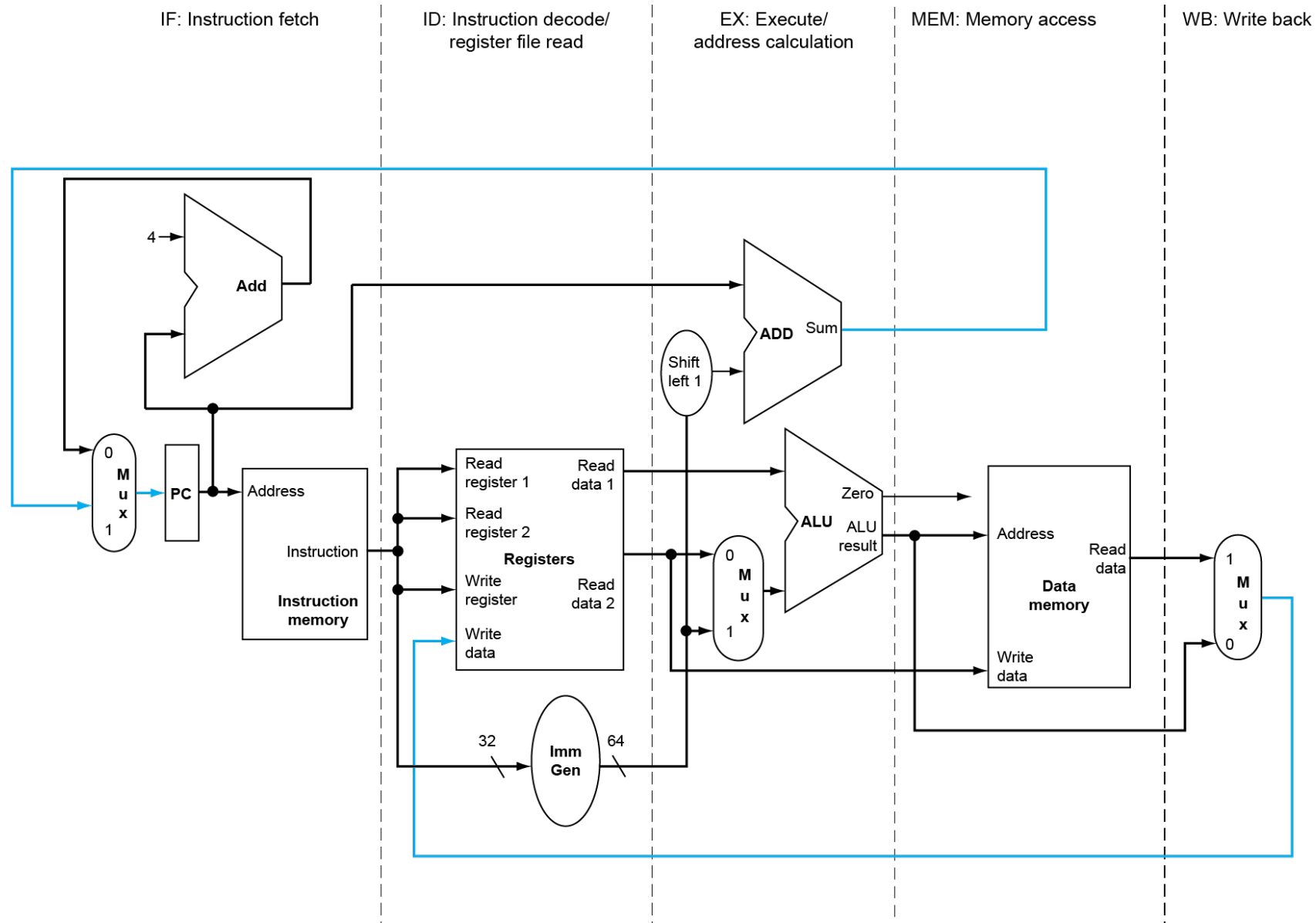
RISC Pipeline – Stadi

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
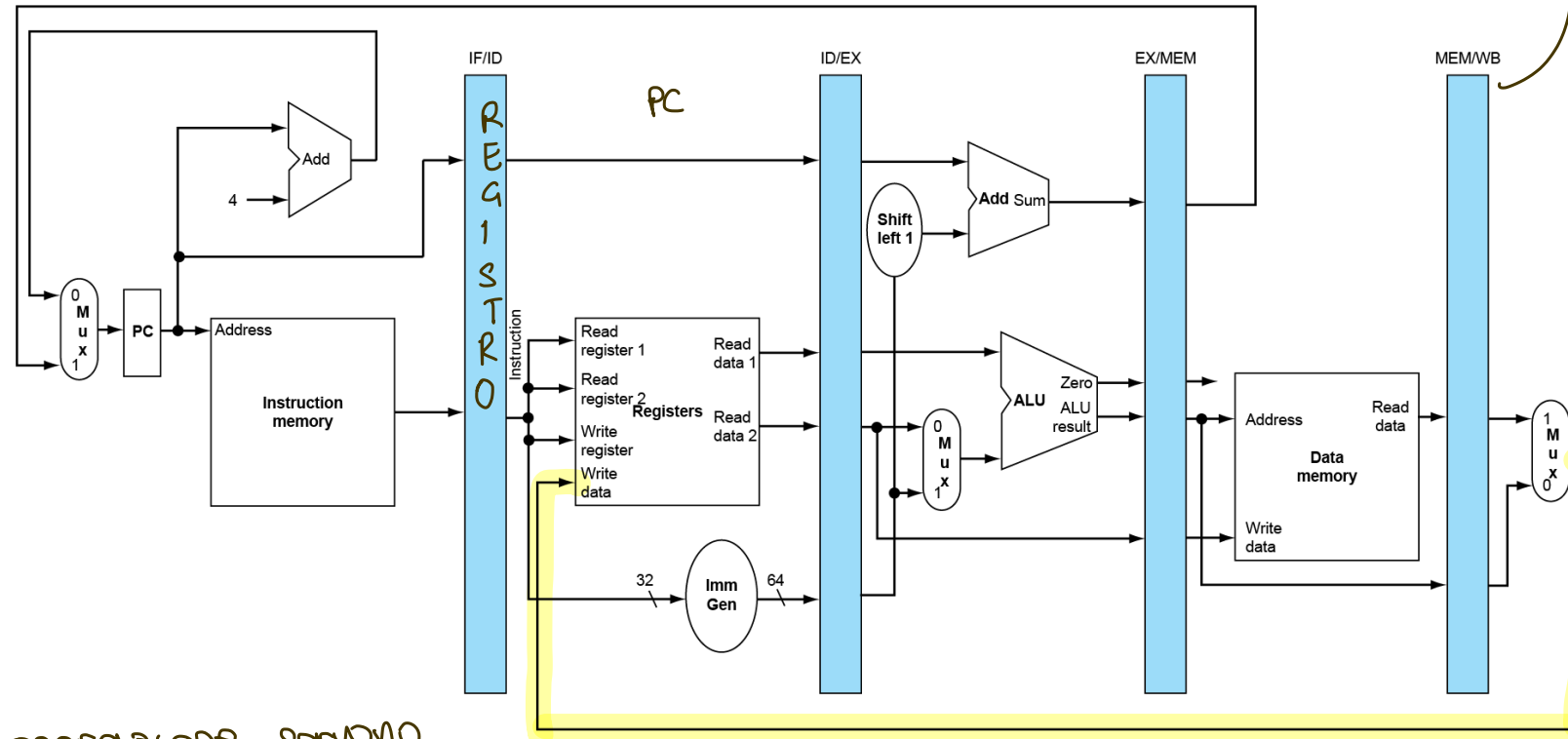
RISC-V Pipelined Datapath

Pipeline registers

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

Sono registri in cui si immagazzinano questi dati parziali.

Termino 1 istruzione x ciclo di ck

e ci sono 5 istruz.contemp. nella pipeline.

per completare un operazione servono 5 cicli di ck. la latenza peggiora un pó perchè ho inserito i registri.

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $Time\ between\ instructions_{pipelined} = \dfrac{Time\ between\ instructions_{nonpipelined}}{Number\ of\ stages}$
- 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
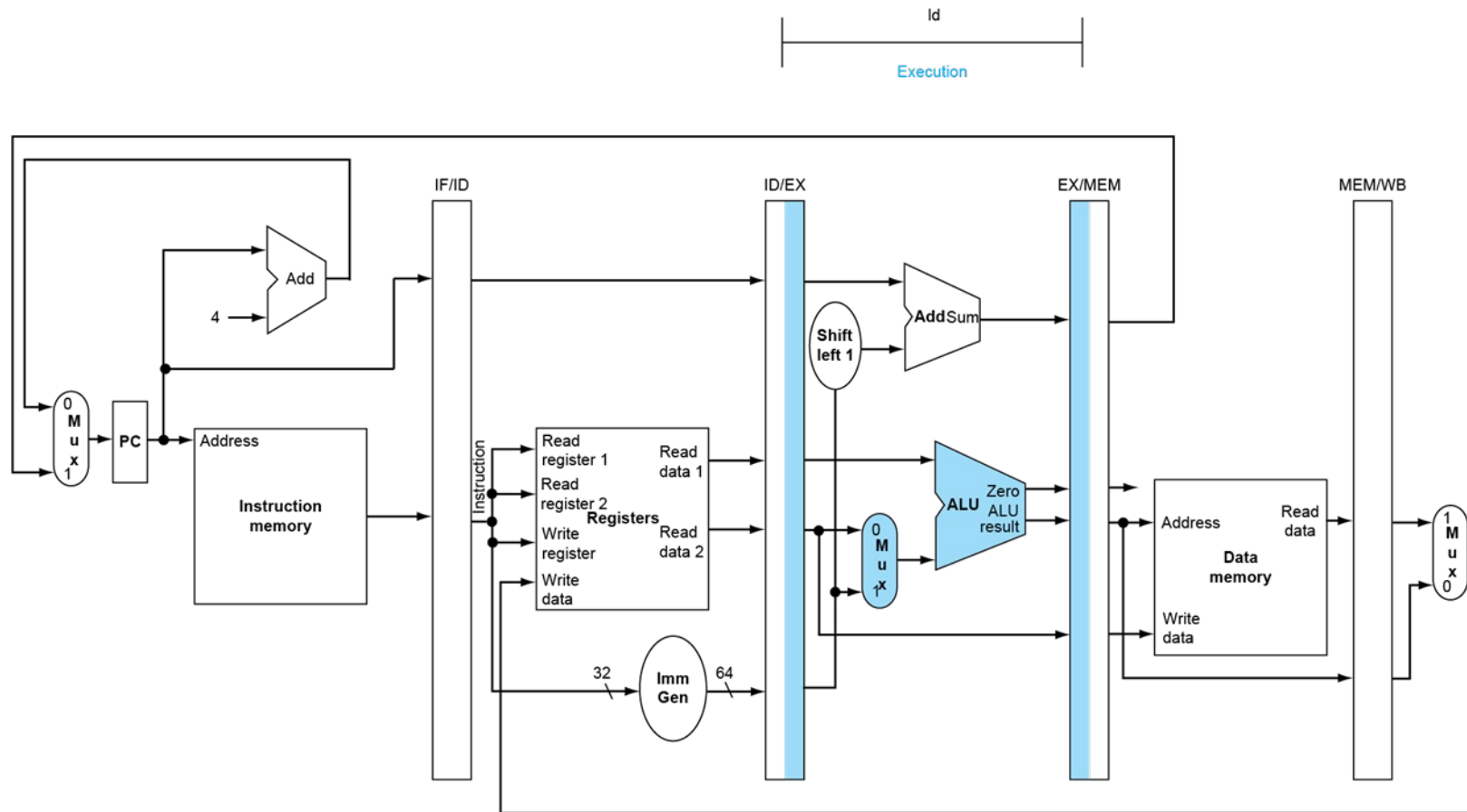   - Can calculate address in 3rd stage, access memory in 4th stage
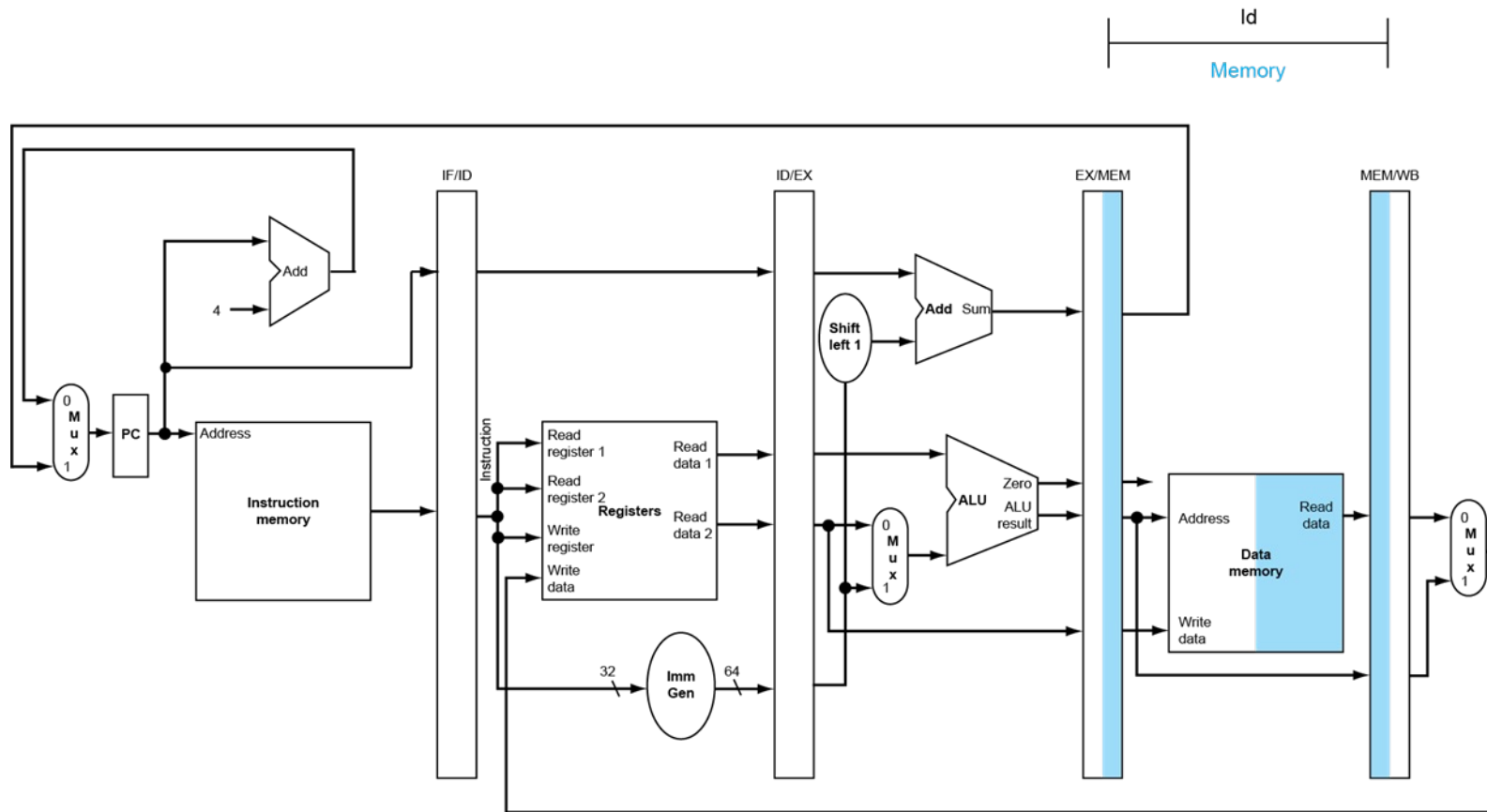
IF for Load, Store, ...

PROPAGO IL PC

Id
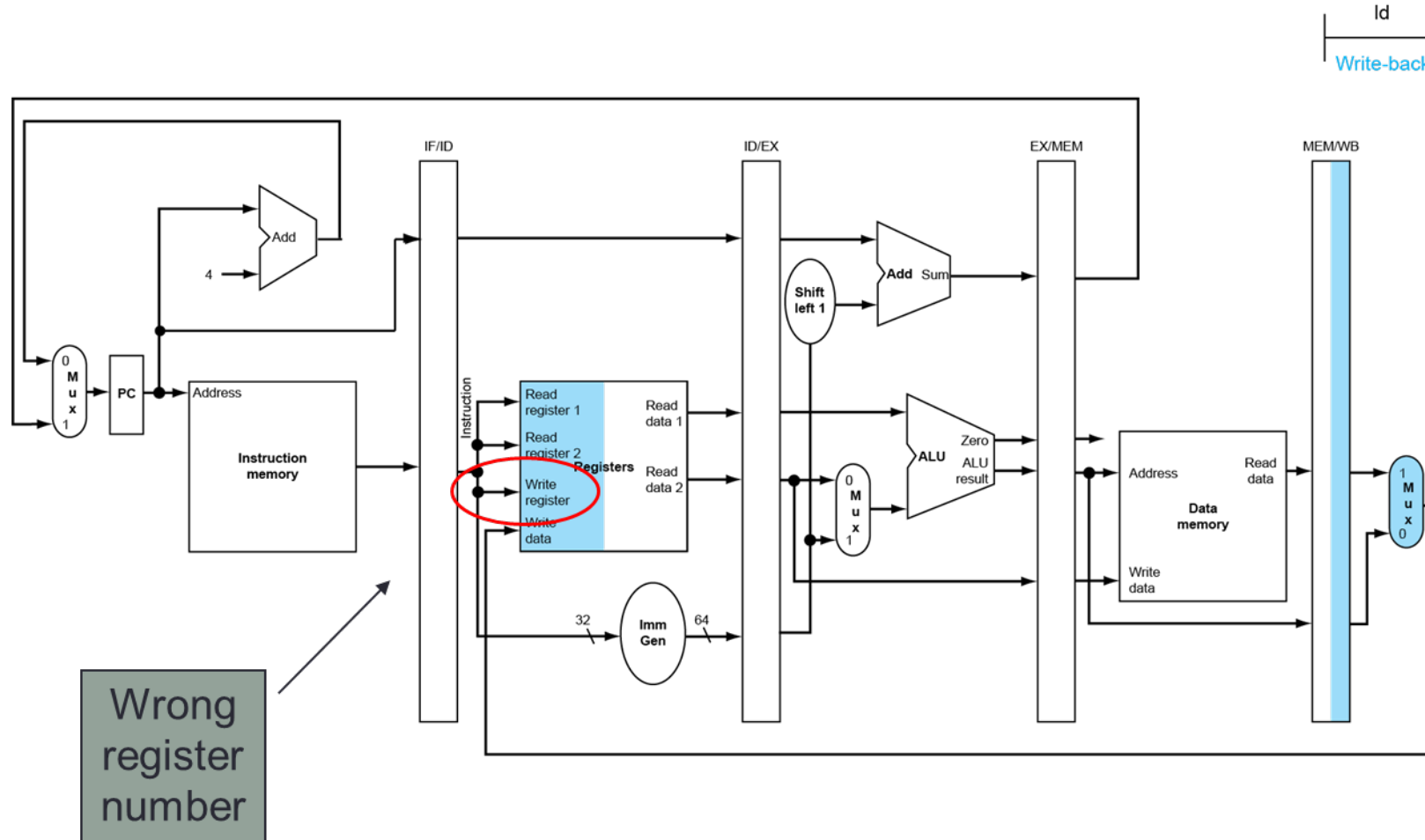Instruction fetch
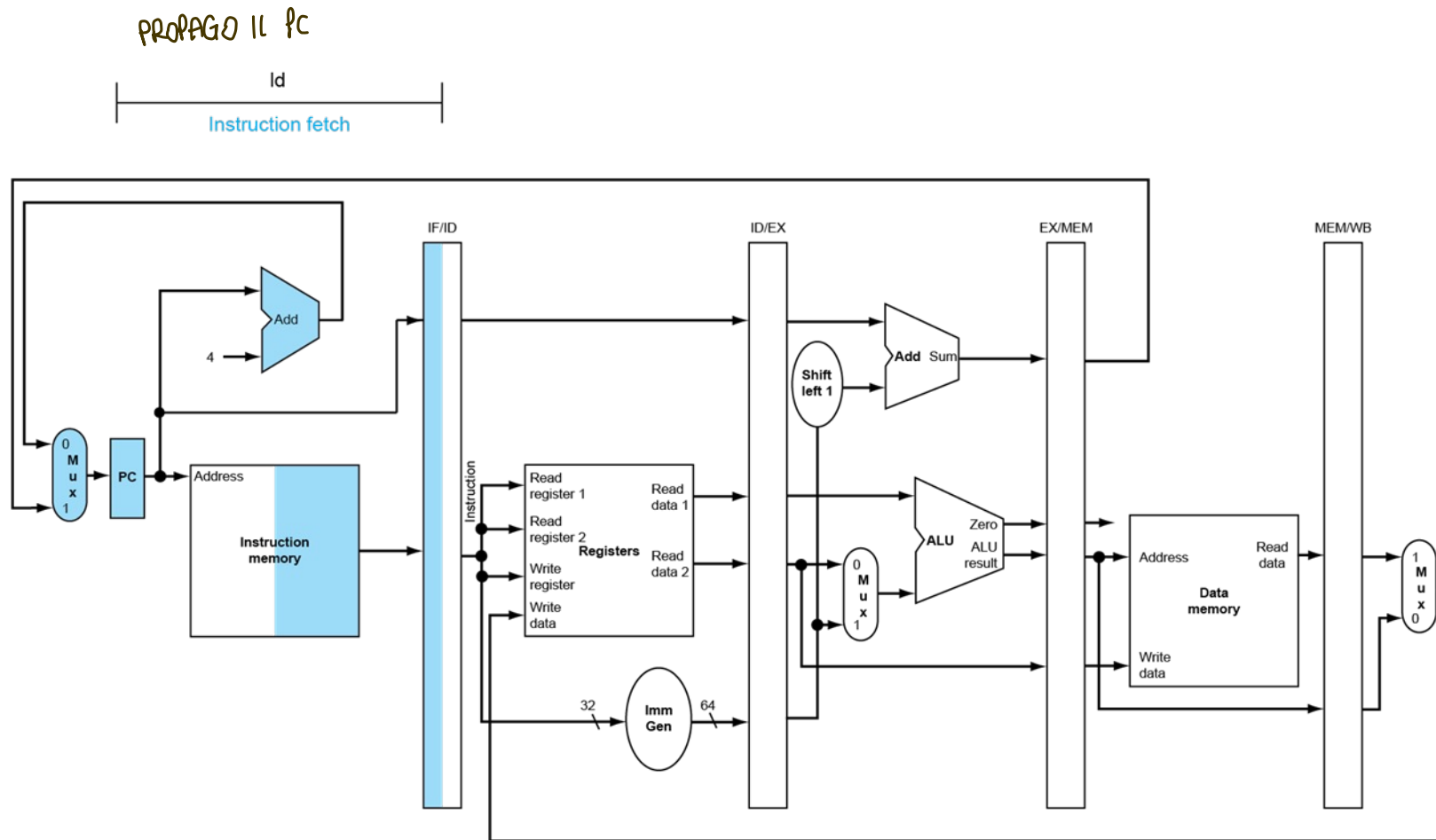
ID for Load, Store, …
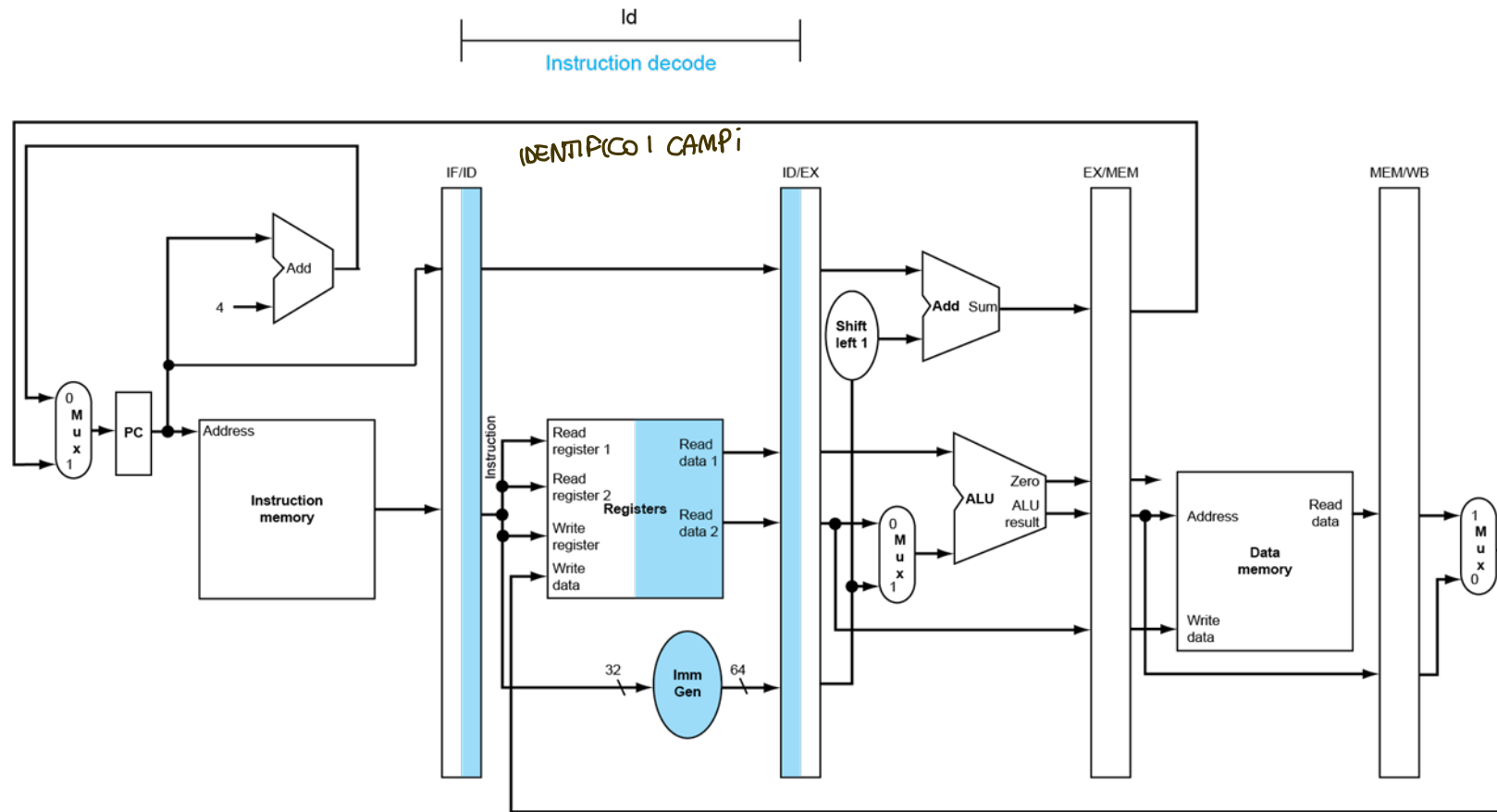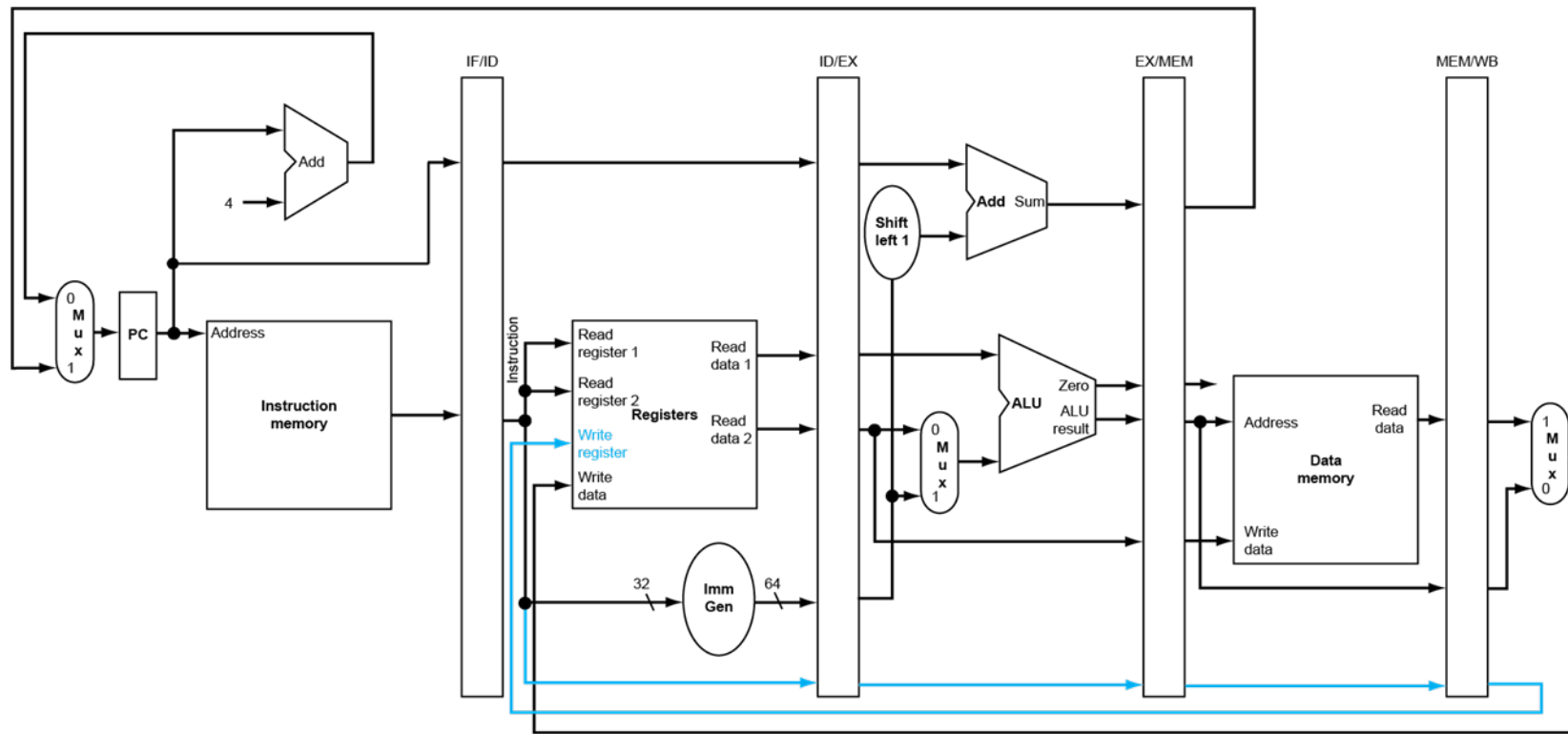
EX for Load

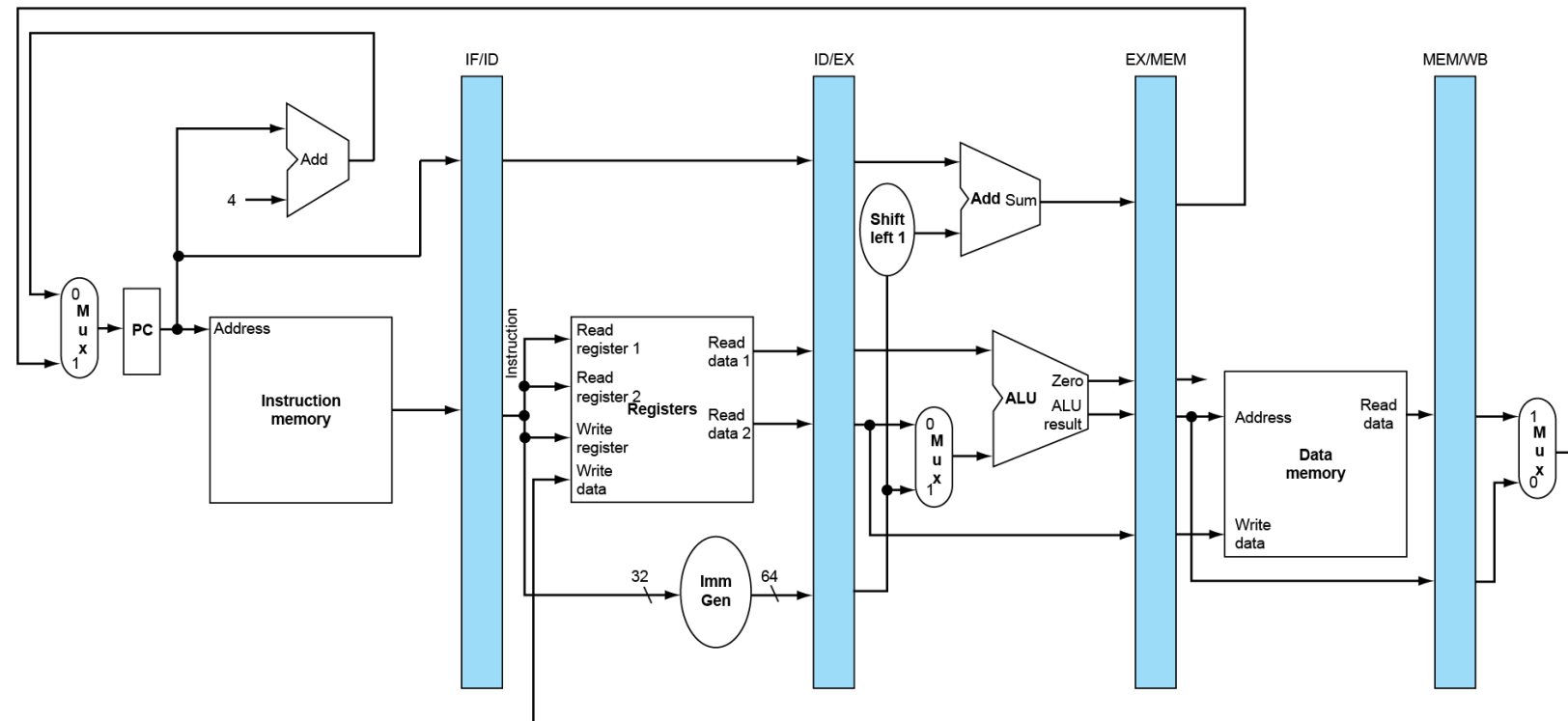# MEM for Load

WB for Load

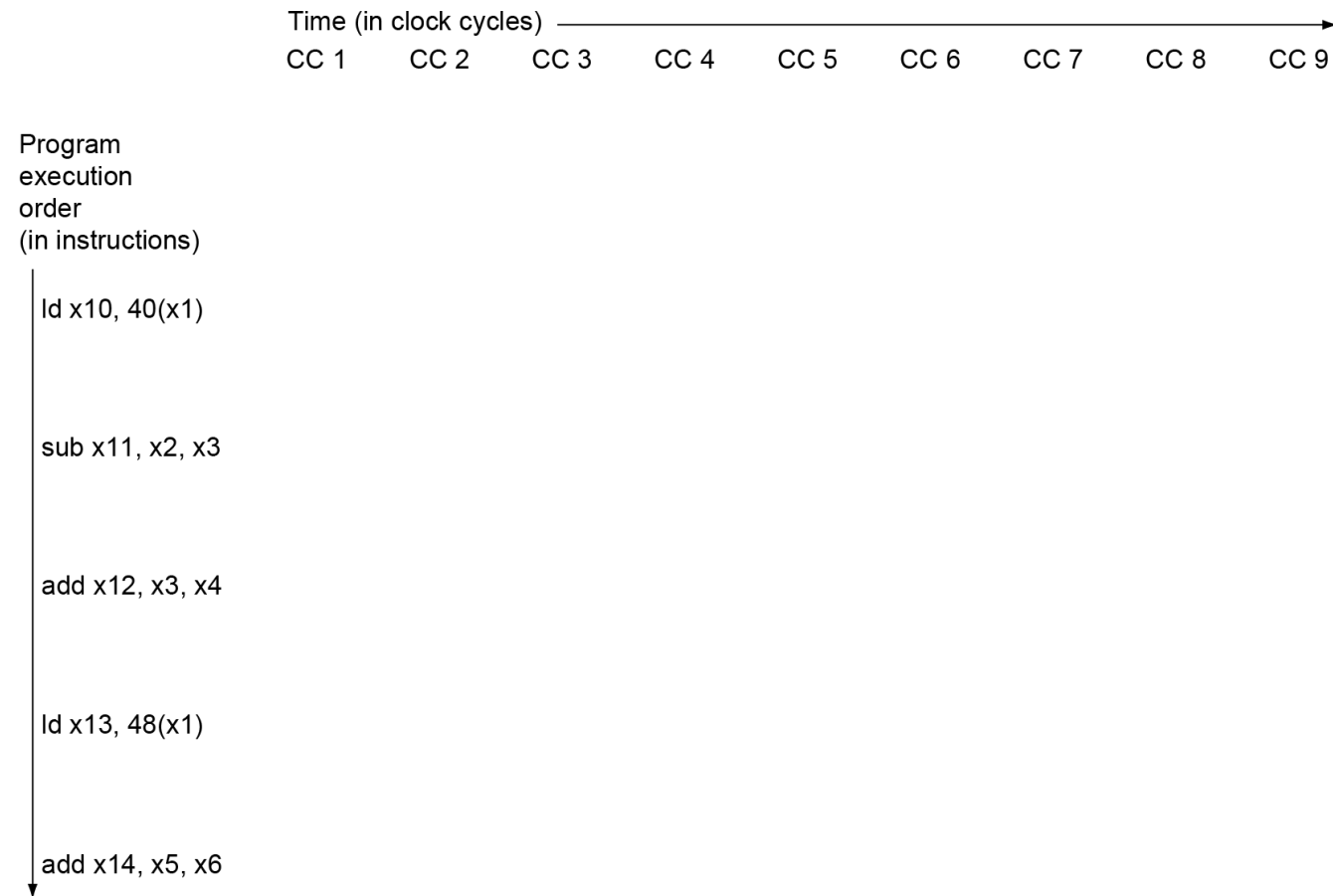# Corrected Datapath for Load

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]` `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]];`
	`ID/EX.NPC←IF/ID.NPC; ID/EX.IR←IF/ID.IR;`
	`ID/EX.Imm←sign-extend(IF/ID.IR[immediate field]);`

Stage	**ALU instruction**	**Load instruction**	**Branch instruction**

EX	`EX/MEM.IR←ID/EX.IR;`
	`EX/MEM.ALUOutput←`
	`ID/EX.A func ID/EX.B;`
	
	or
	
	`EX/MEM.ALUOutput←`
	`ID/EX.A op ID/EX.Imm;`
		`EX/MEM.IR to ID/EX.IR`
		`EX/MEM.ALUOutput←`
		`ID/EX.A+ID/EX.Imm;`
		
		`EX/MEM.B←ID/EX.B;`
			`EX/MEM.ALUOutput←`
			`ID/EX.NPC +`
			`(ID/EX.Imm<<2);`
			
			`EX/MEM.cond←`
			`(ID/EX.A == ID/EX.B);`

MEM	`MEM/WB.IR←EX/MEM.IR;`
	`MEM/WB.ALUOutput←`
	`EX/MEM.ALUOutput;`
		`MEM/WB.IR←EX/MEM.IR;`
		`MEM/WB.LMD←`
		`Mem[EX/MEM.ALUOutput];`
		or
		`Mem[EX/MEM.ALUOutput]←`
		`EX/MEM.B;`	

WB	`Regs[MEM/WB.IR[rd]]←`
	`MEM/WB.ALUOutput;`
		For load only:
		`Regs[MEM/WB.IR[rd]]←`
		`MEM/WB.LMD;`	

Multi-Cycle Pipeline Diagram

- Form showing resource usage

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

Program
execution
order
(in instructions)

ld x10, 40(x1)

sub x11, x2, x3

add x12, x3, x4

ld x13, 48(x1)

add x14, x5, x6

Multi-Cycle Pipeline Diagram

- Form showing resource usage

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

Program
execution
order
(in instructions)

ld x10, 40(x1)   IM

sub x11, x2, x3

add x12, x3, x4

ld x13, 48(x1)

add x14, x5, x6

Multi-Cycle Pipeline Diagram

- Form showing resource usage

Multi-Cycle Pipeline Diagram

- Form showing resource usage

Multi-Cycle Pipeline Diagram

- Form showing resource usage
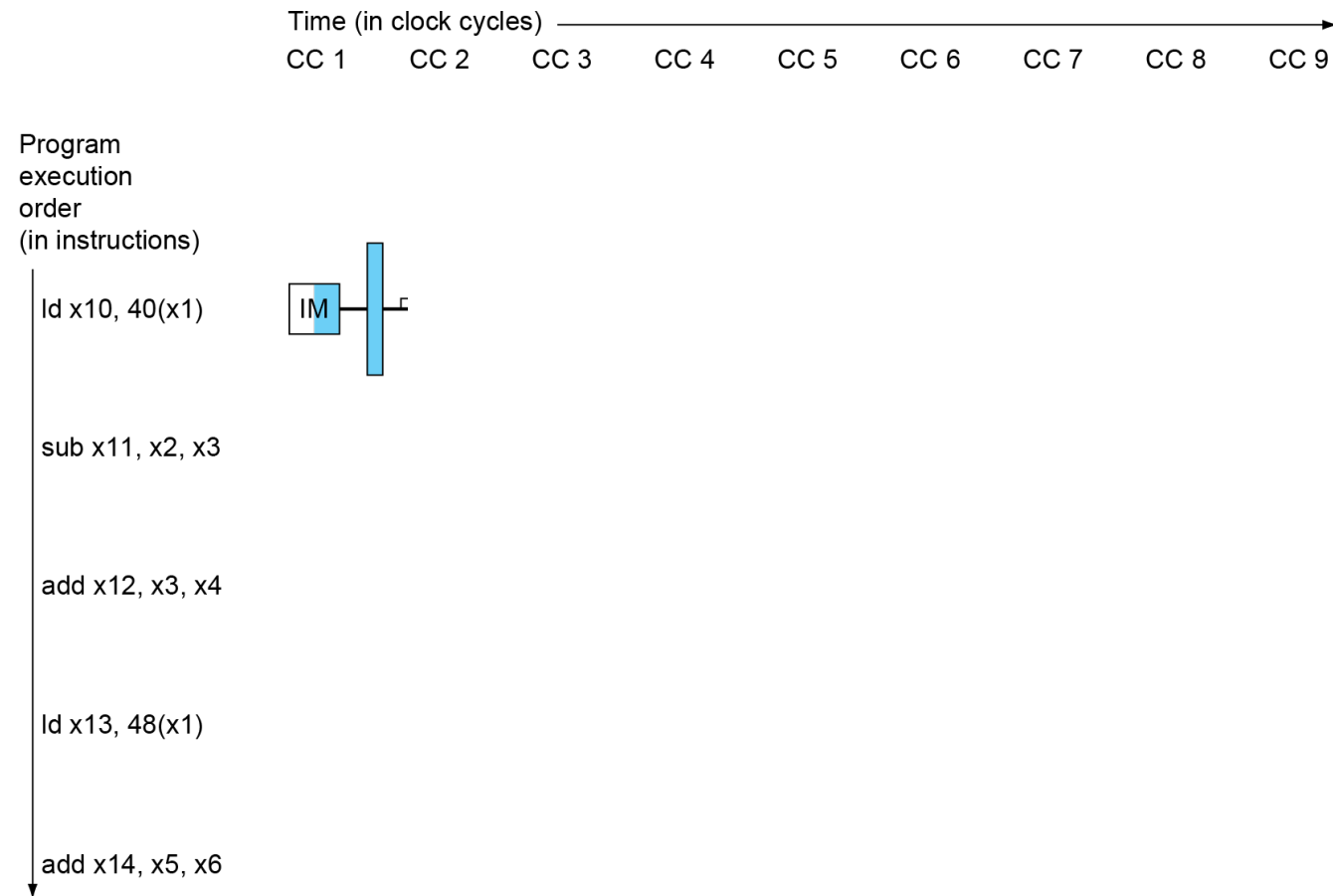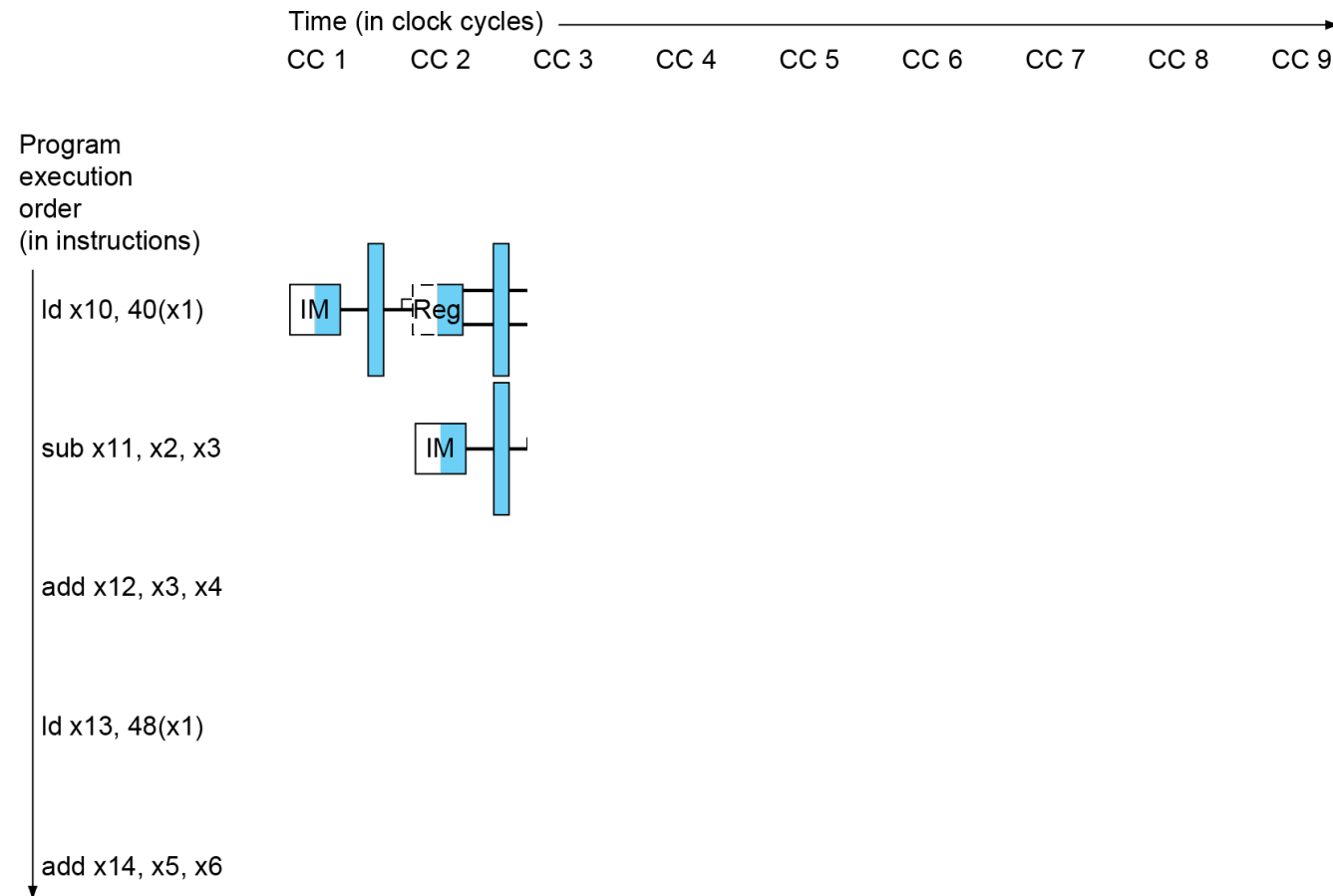
Multi-Cycle Pipeline Diagram

- Form showing resource usage

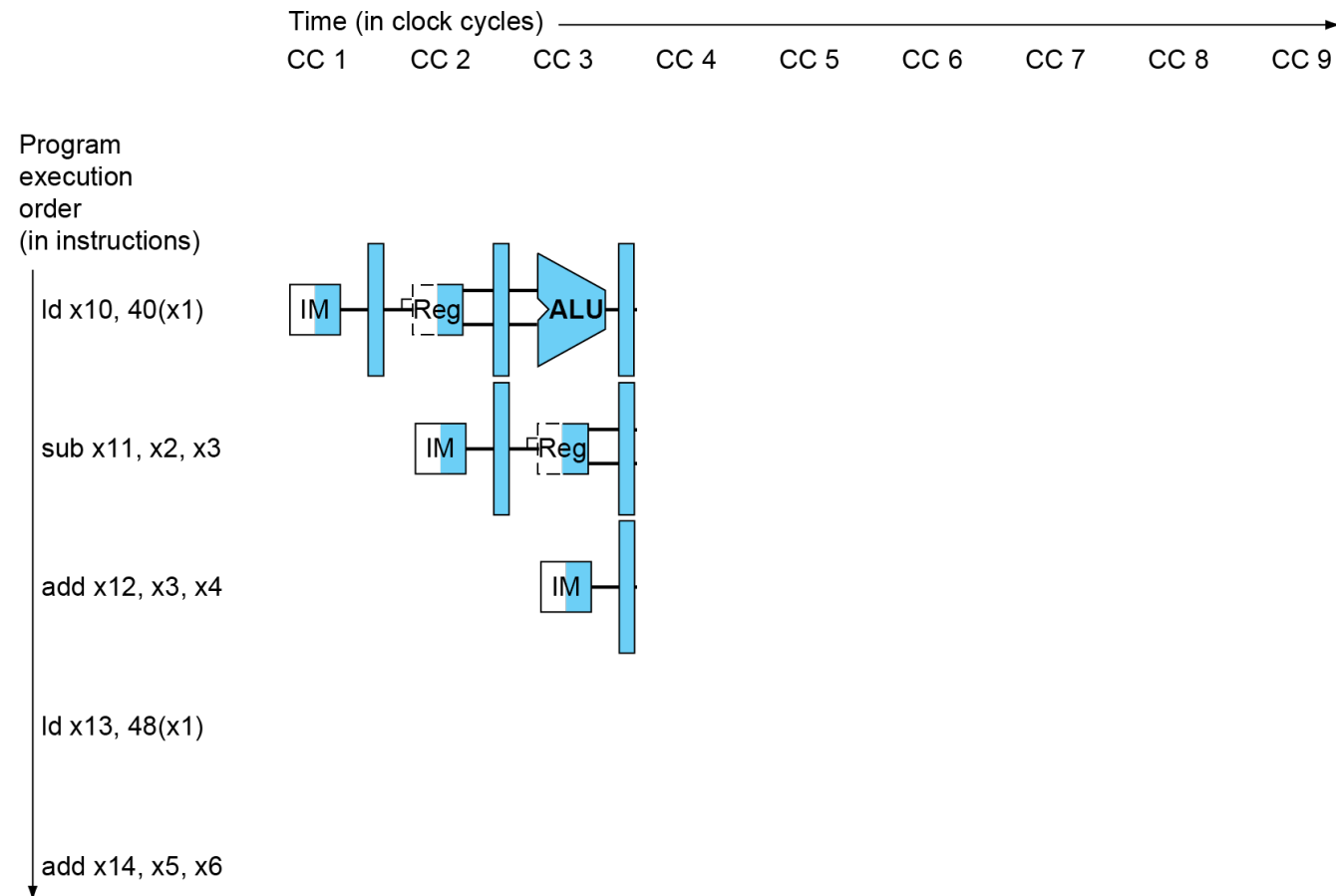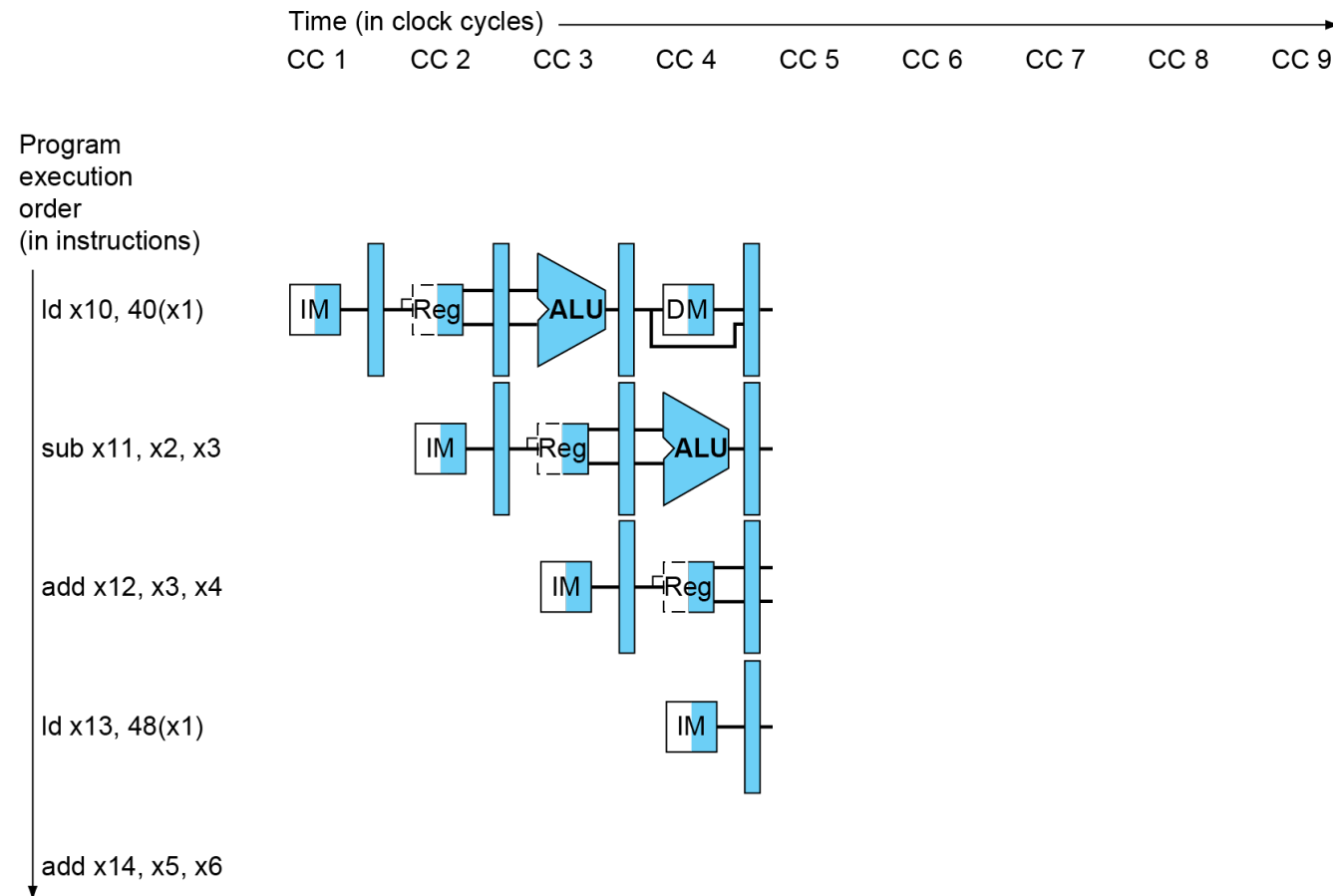# Multi-Cycle Pipeline Diagram

- 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

$$CPI = \frac{9}{5}$$

Multi-Cycle Pipeline Diagram

- Traditional form

Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9

Program
execution
order
(in instructions)

ld x10, 40(x1)

sub x11, x2, x3

add x12, x3, x4

ld x13, 48(x1)

add x14, x5, x6
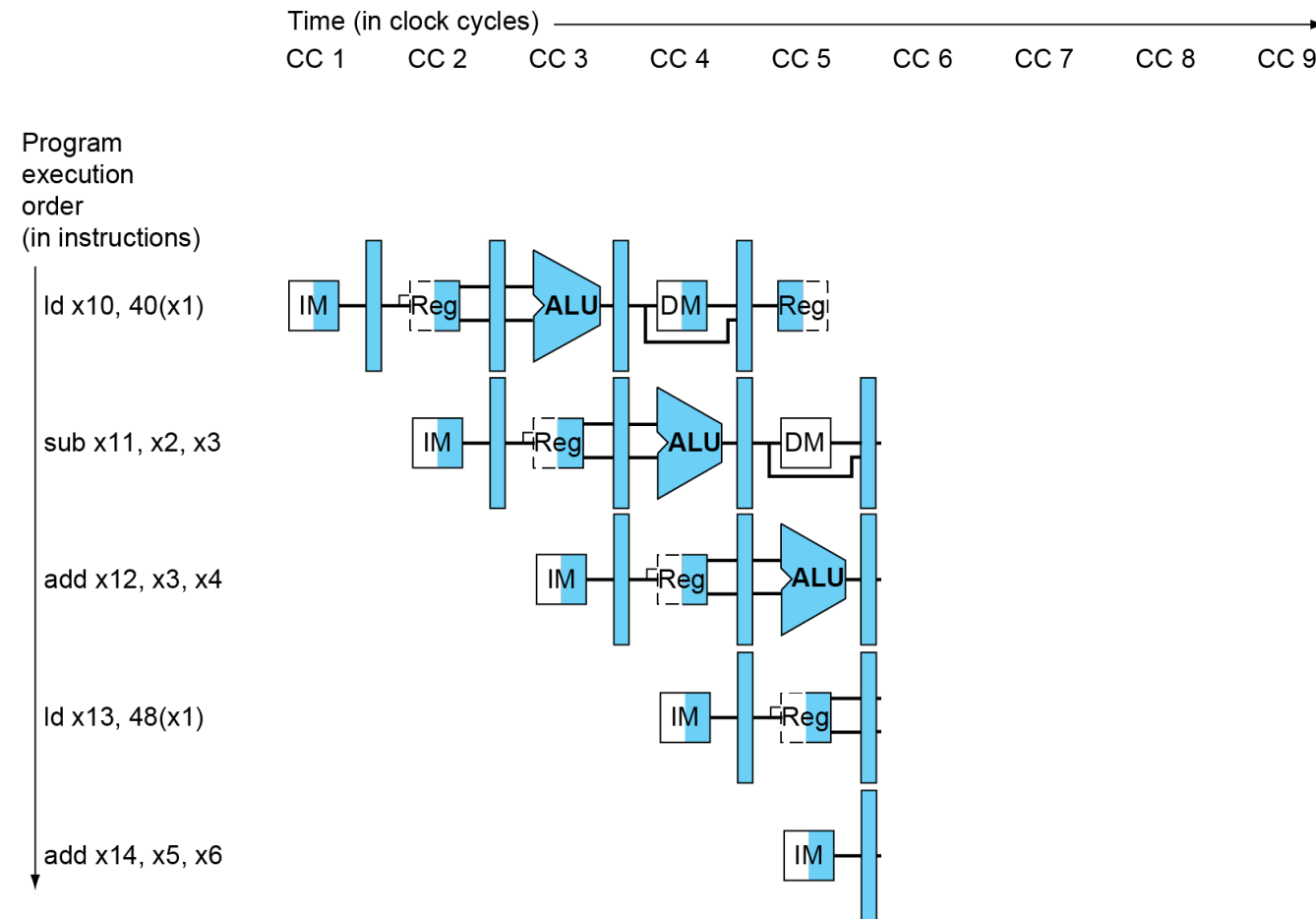
Multi-Cycle Pipeline Diagram

- Traditional form

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle
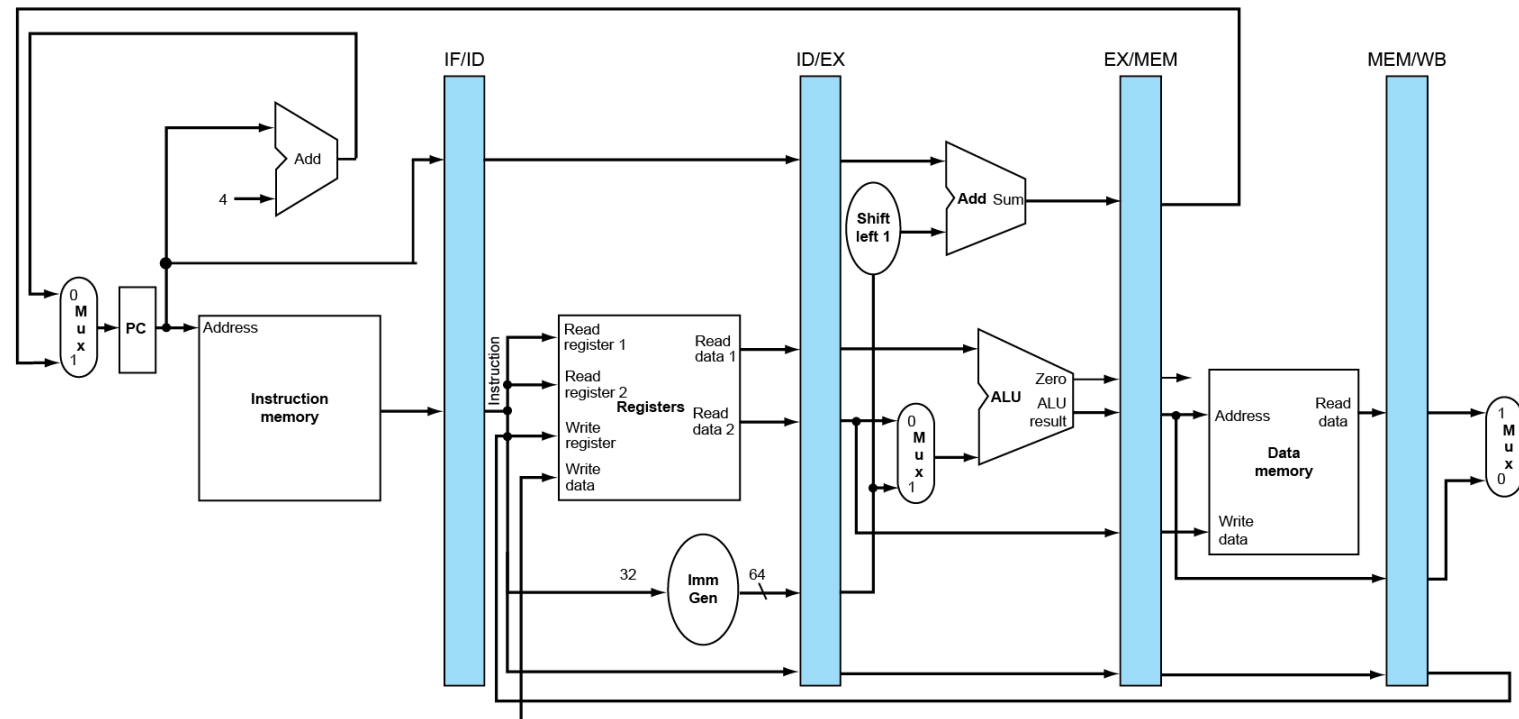
Hazards/Alee

- 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

x19 is written back in the register file in this cycle

| | 200 | 400 | 600 | 800 | 1000 | 1200 | 1400 | 1600 |

Time

add x19, x0, x1   IF   ID   EX   MEM   WB

Data Hazards

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

x19 is written back in the register file in this cycle

Time

add x19, x0, x1

IF ID EX MEM WB

sub x2, x19, x3

IF ID EX MEM WB
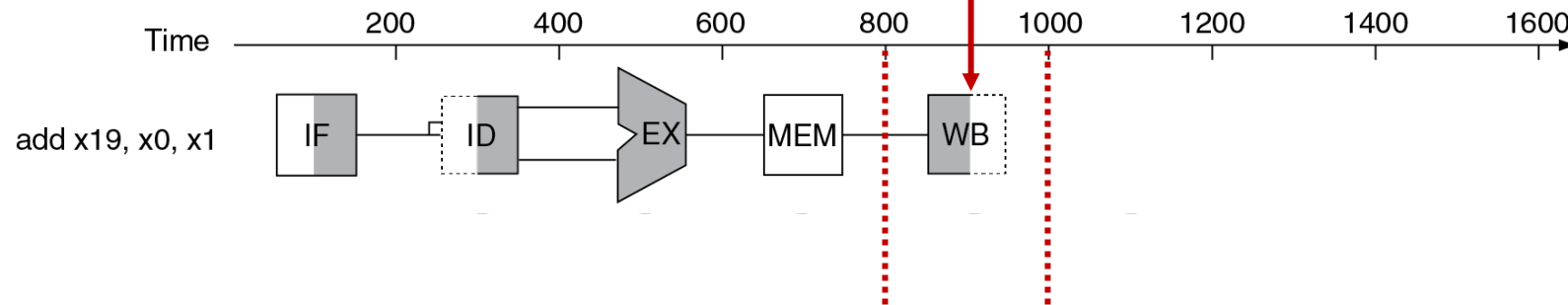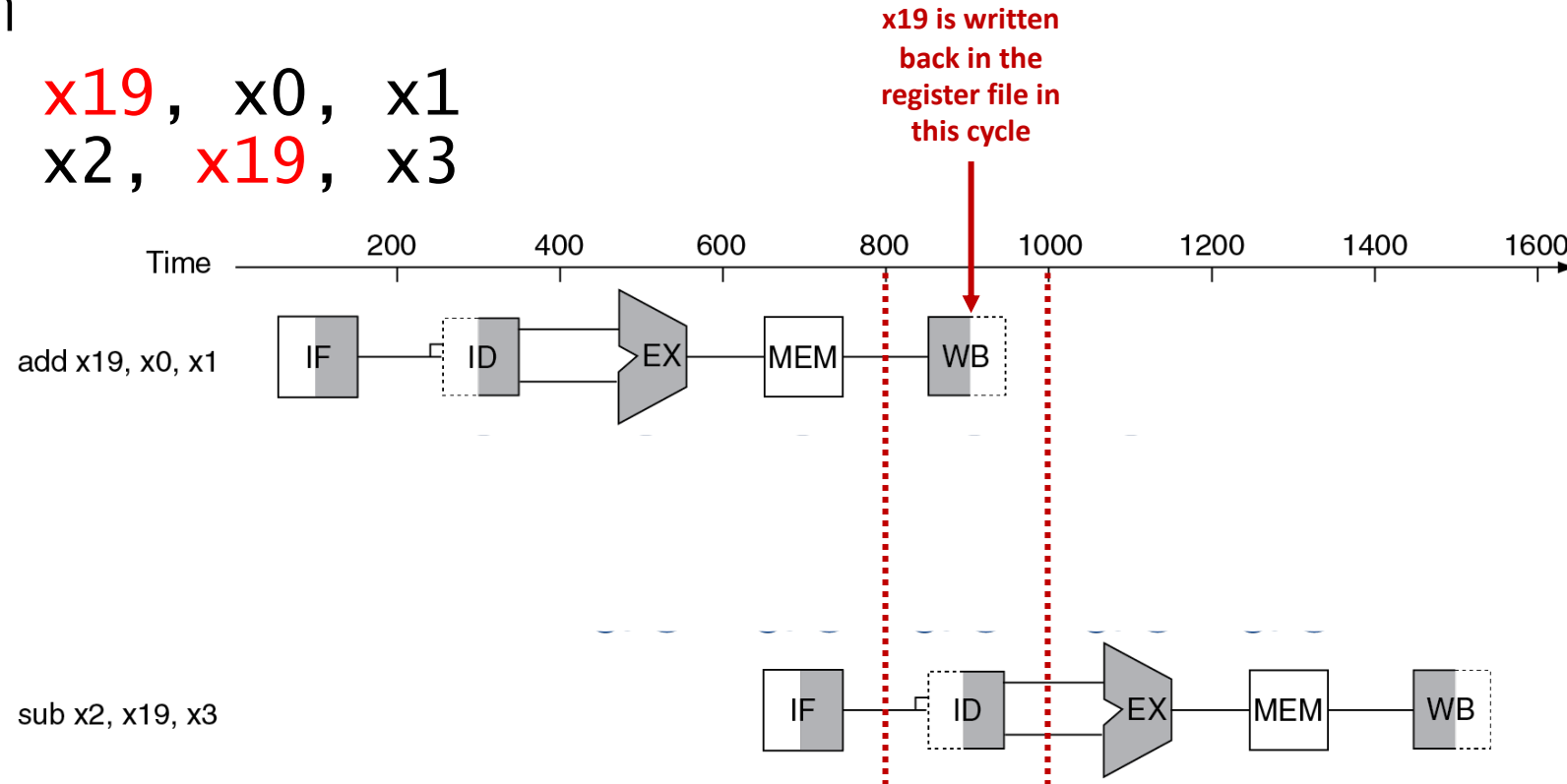
the `sub` instruction can only read it after it's written

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.

x19 is written back in the register file in this cycle

Time

| 200 | 400 | 600 | 800 | 1000 | 1200 | 1400 | 1600 |

add x19, x0, x1   IF — ID — EX — MEM — WB

Thus we need to stall the pipeline for two cycles

bubble bubble bubble bubble bubble
bubble bubble bubble bubble bubble

sub x2, x19, x3   IF — ID — EX — MEM — WB

the `sub` instruction can only read it after it's written
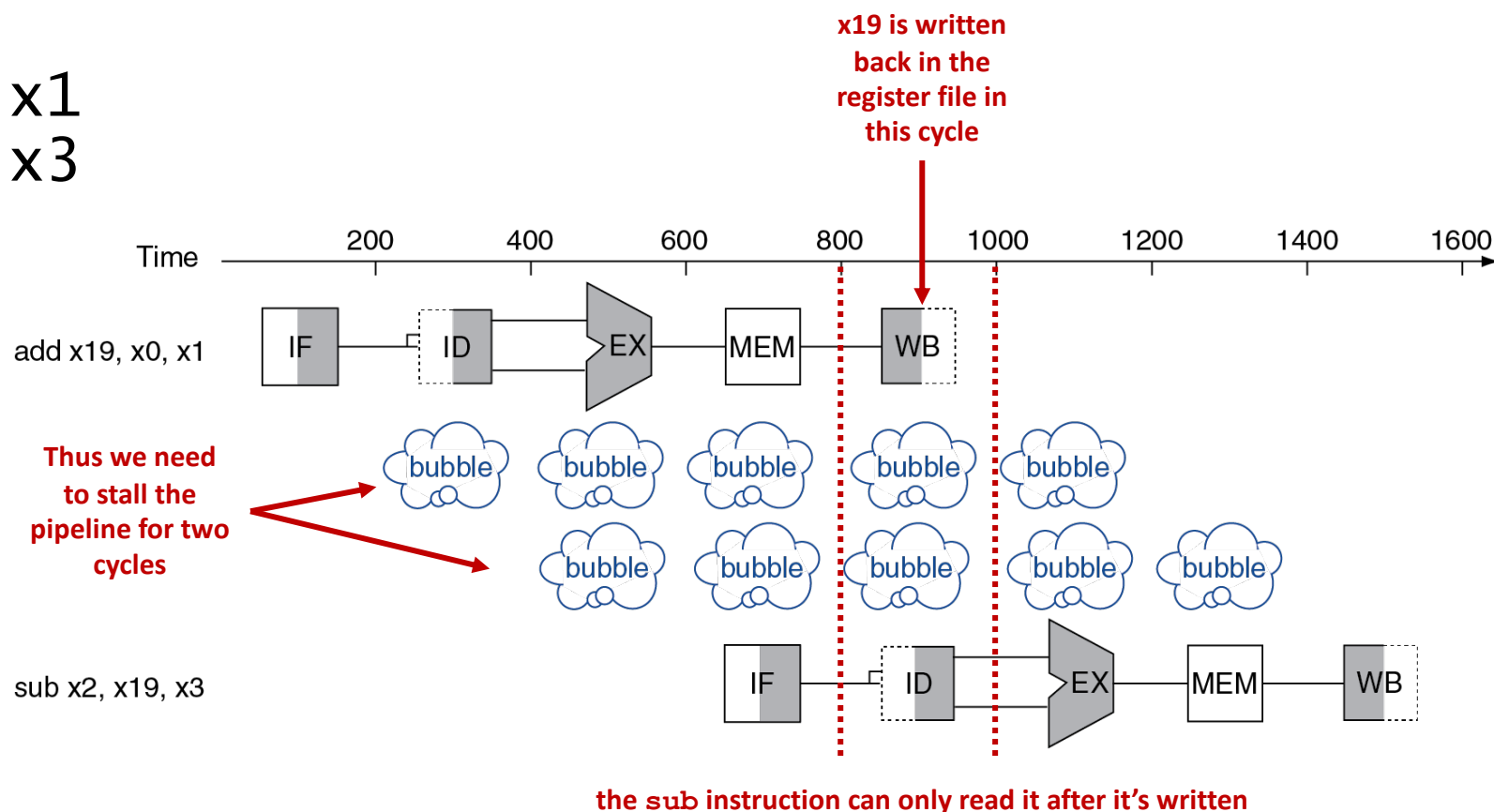
Forwarding (aka Bypassing)

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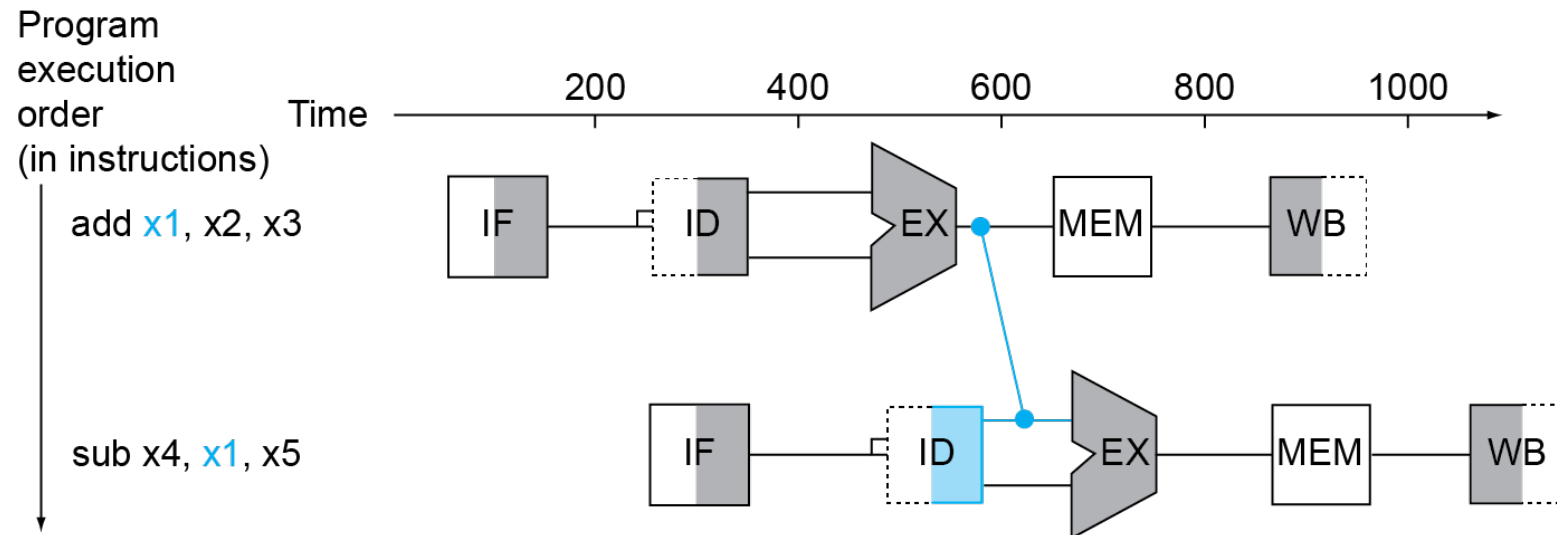
Load-Use Data Hazard

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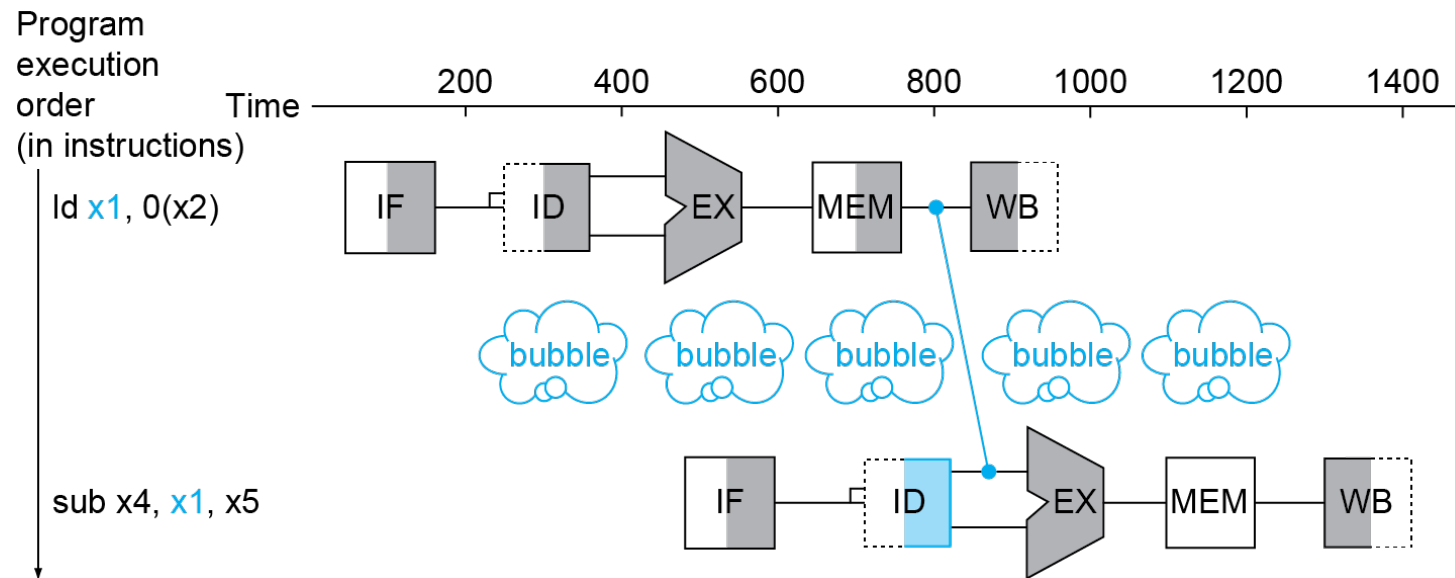
Code Scheduling to Avoid Stalls

b: 0 × 0
e: offset 8   (× 8)
f: offset 16 (× 16)

a: offset 24

c: 32 + (x0)

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

leggo b in x1 → e lo metto

leggo e ... in x2 →

stallo di 1 ciclo. → **stall**

ld conoscerà il valore numerico contenuto all'indirizzo 8(x0)

salvo in memoria il valore di A.

stall

```
ld      x1, 0(x0)
ld      x2, 8(x0)
add     x3, x1, x2
sd      x3, 24(x0)
ld      x4, 16(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

```
ld      x1, 0(x0)
ld      x2, 8(x0)
ld      x4, 16(x0)
add     x3, x1, x2
sd      x3, 24(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

9 cycles

CPI? 9/7 ≅ 1,2

7 cycles

CPI?
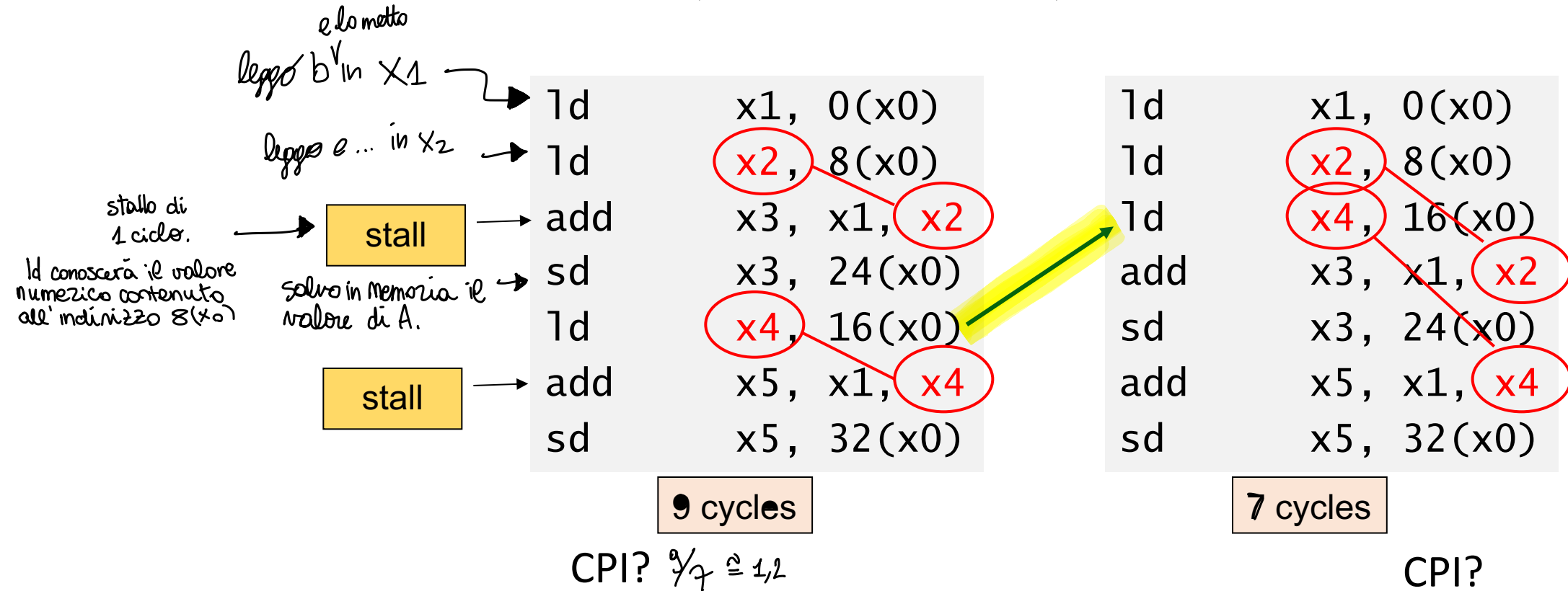
Control Hazards
quando c'è un istruz. di branch e devo aspettare l'esito.

- 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
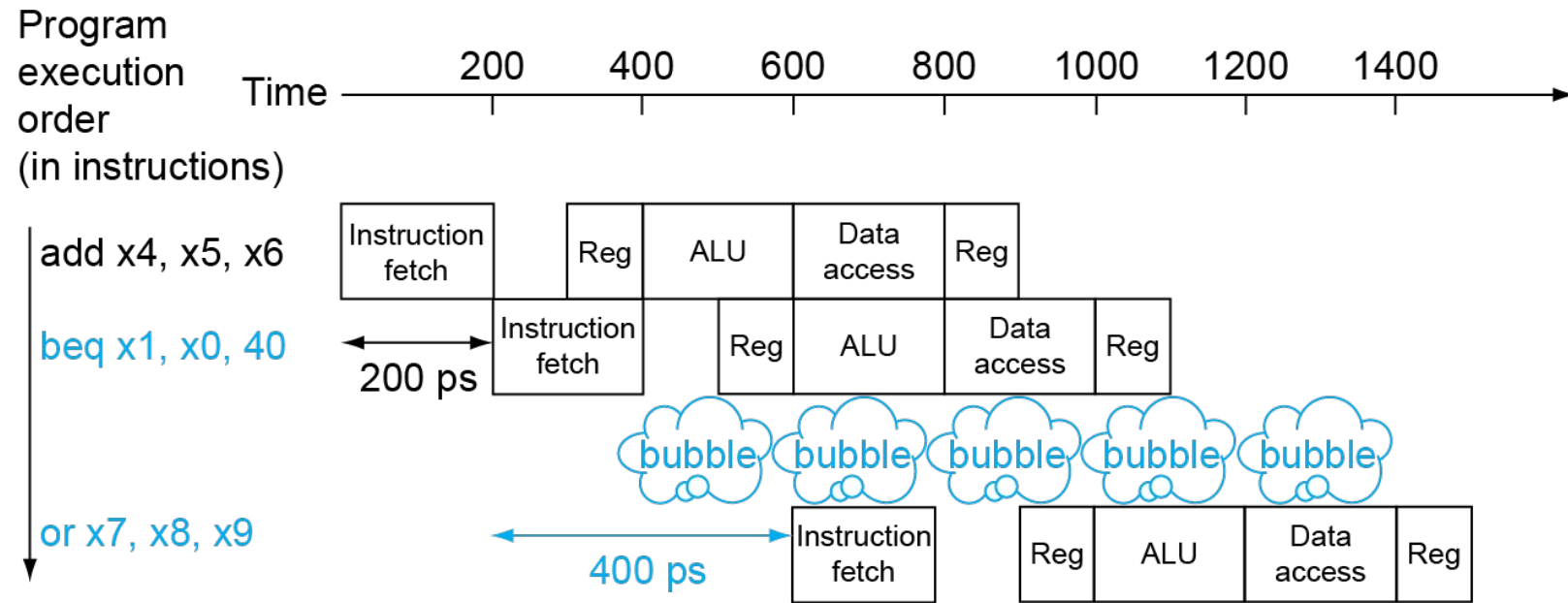
2. Static: Always predict "taken" or "untaken"

- 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