POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Il processore pipeline
Capitolo 4 P&H

4 . 11 . 2015

Pipelining

Tecnica per migliorare le prestazioni basata sulla sovrapposizione dell'esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale.

Idea base:
Il lavoro svolto in una *CPU* pipeline per eseguire un'istruzione è diviso in passi **(stadi di pipeline),** che richiedono una frazione del tempo necessario al completamento dell'intera istruzione.

Gli stadi sono sovrapposti per formare la pipeline: le istruzioni entrano da una estremità, vengono elaborate attraverso gli stadi, escono dall'altro estremo.

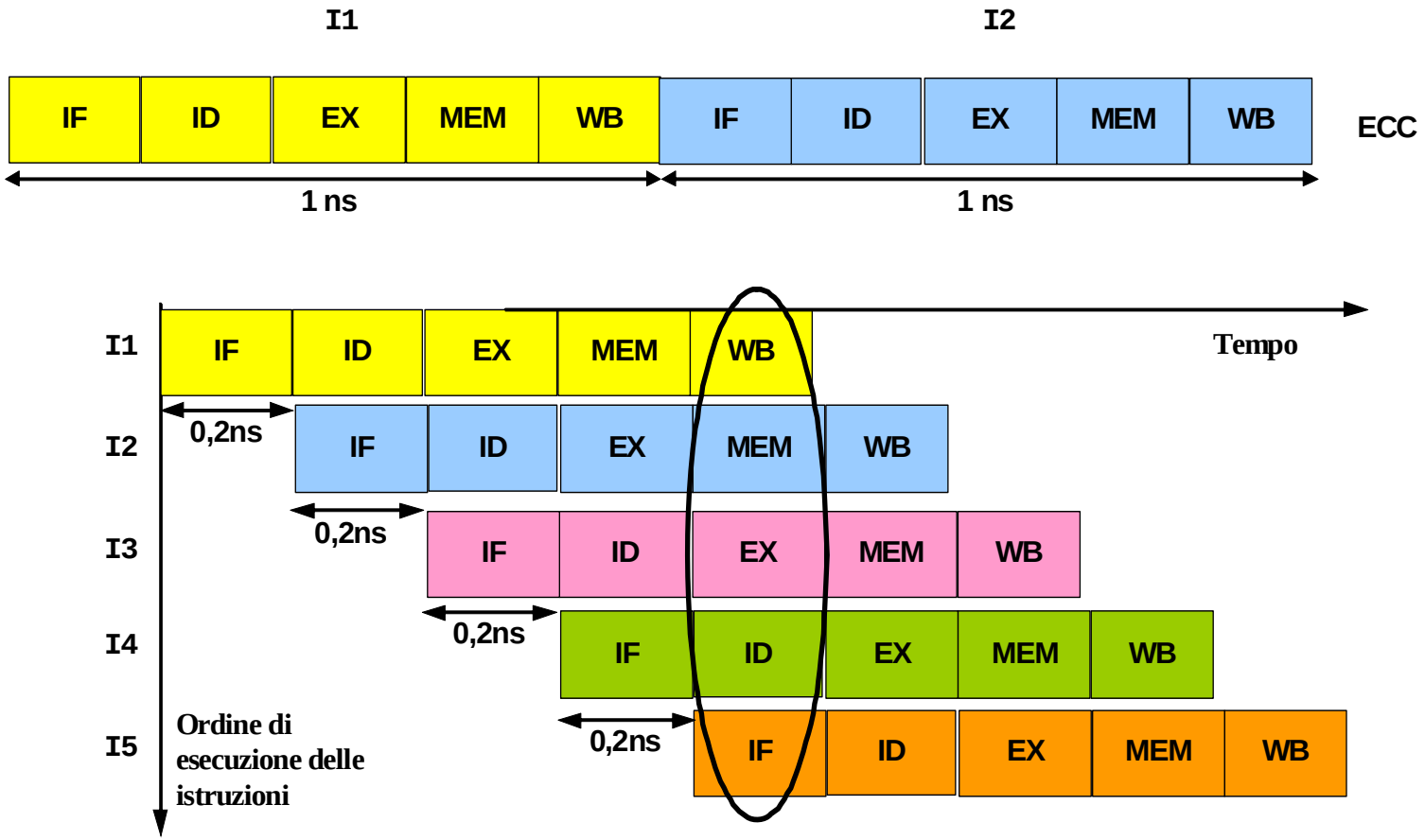
Pipelining (cont.)

Principale vantaggio: tecnica trasparente al programmatore.

Tecnica simile ad una catena di montaggio: una nuova automobile esce dalla catena nel tempo necessario per svolgere uno dei molti passi.

Una catena di montaggio non riduce il *tempo* di realizzazione di una singola automobile, ma aumenta il numero di automobili costruite simultaneamente e pertanto la *frequenza* con cui le automobili vengono iniziate e completate.

Esecuzione sequenziale vs. pipelining



Pipelining (cont.)

Il tempo necessario per far avanzare un'istruzione di uno stadio lungo la pipeline corrisponde idealmente ad un ciclo di clock.

Poiché gli stadi di pipeline sono collegati in successione, devono tutti operare in modo sincrono: la durata di un ciclo di clock è determinata dal tempo richiesto per lo stadio più lento della pipeline *(es. 200 ps).*

L'obiettivo dei progettisti è **bilanciare** la lunghezza di ciascuno stadio.

Se gli *stadi* sono *perfettamente bilanciati*, l'accelerazione *ideale* dovuta al pipelining è pari al numero di stadi di pipeline

- una *CPU* pipeline a *5* stadi è *5* volte più veloce della stessa *CPU* senza pipeline, in termini di completamento di una istruzione rispetto alla successiva

Tempo tra due istruzioni con pipeline =
Tempo tra due istruzioni senza pipeline / Numero degli stadi della pipeline

Pipelining (cont.)

In generale, gli stadi non sono perfettamente bilanciati e l'introduzione del pipelining comporta costi aggiuntivi

- L'intervallo di tempo fra il completamento di *2* istruzioni è superiore al valore minimo possibile
- L'incremento di velocità sarà minore del numero di stadi di pipeline introdotto (in genere una pipeline a *5* stadi non riesce a quintuplicare le prestazioni).

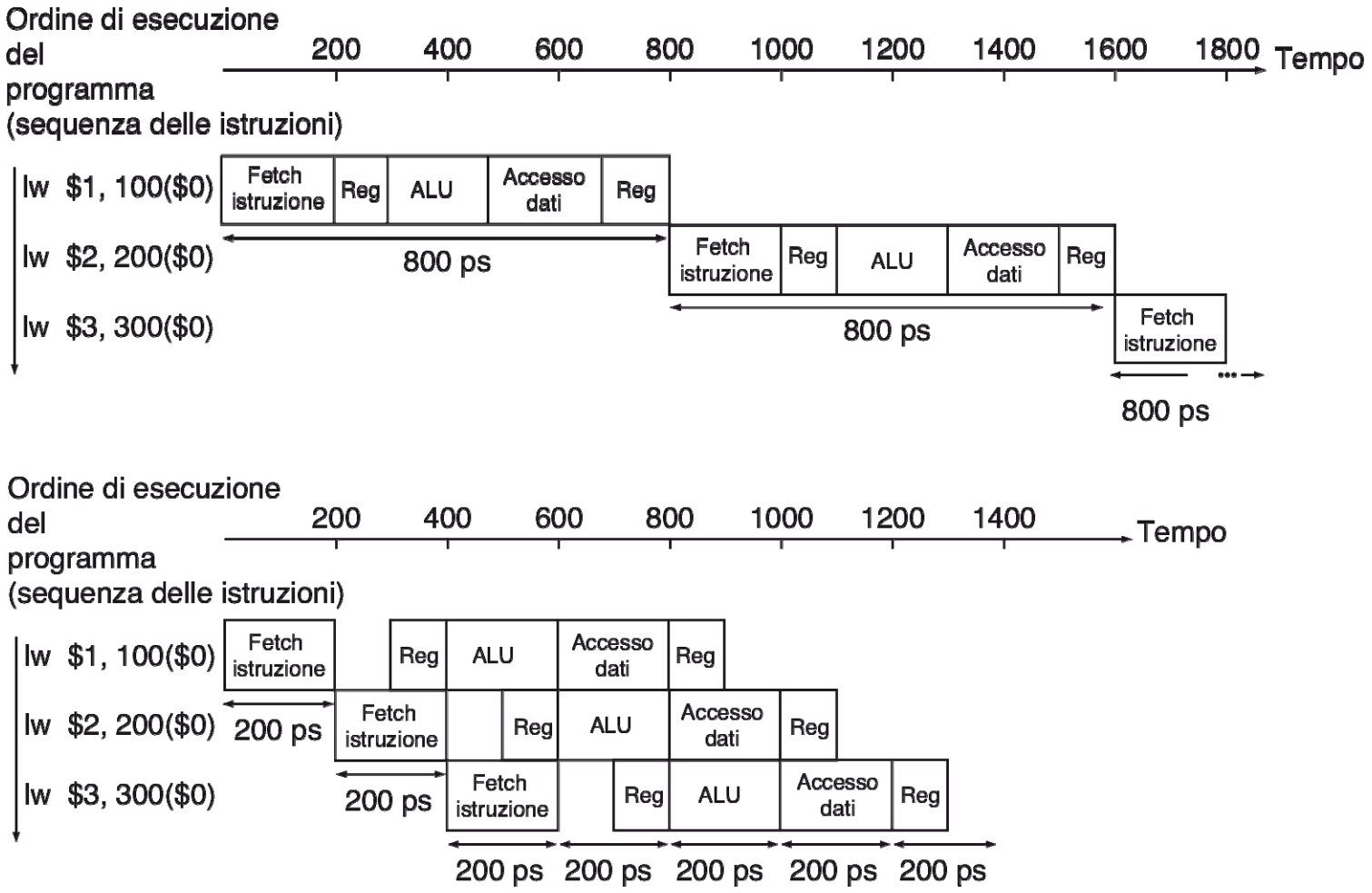
Miglioramento delle prestazioni

Architettura MIPS: Istruzioni suddivise al massimo in *5* passi ⇒ **Pipeline a 5 stadi**

Ogni stadio di pipeline ha una durata prefissata **(ciclo di pipeline),** che deve essere sufficientemente lunga da consentire l'esecuzione dell'operazione più lenta:

ad esempio si deve adottare un ciclo di almeno *200 ps*, sebbene alcuni stadi richiedano solo *100 ps*.

Confronto tra una esecuzione a singolo ciclo di clock e pipeline



Miglioramento delle prestazioni (cont.)

Il fattore di miglioramento relativo all'esecuzione di 3 istruzioni di *load* dovrebbe essere

$$T_{inizio}\ \text{istruzione\_4 senza pipeline} / T_{inizio}\ \text{istruzione\_4 con pipeline}$$

2400 (3x800)ps / 600 (3x200)ps = 4

Il fattore di miglioramento è 4 e non 5 perché alcuni passi dell'esecuzione richiedono in effetti un tempo inferiore al ciclo di clock della pipeline

Ma il *tempo di esecuzione totale* di *3* istruzioni di *load* comporta un miglioramento più modesto:

- *3* istruzioni di *load* senza pipeline: *3 x800 ps = 2400 ps*
- *3* istruzioni di *load* con pipeline: *2 x200 ps + 1000 ps = 1400 ps*
- Fattore di *miglioramento = 2400 / 1400 = 1,71*

Questa differenza è provocata dal *tempo necessario a riempire e svuotare la pipeline*: servono *4 stadi* per riempire la pipeline

Miglioramento delle prestazioni (cont.)

Al crescere del numero di istruzioni il rapporto tra i tempi totali di esecuzione dei programmi su macchine senza e con pipeline è vicino al limite ideale (stabilito dal rapporto dei tempi di completamento delle istruzioni):

- *1 000 000* istruzioni di *load* senza pipeline:
  $1\,000\,000 \times 0{,}8\ ns = 800\,000\ ns$
- *1 000 000* istruzioni di *load* con pipeline:
  $1\,000\,000 \times 0{,}2\ ns + 0{,}8\ ns = 200\,000{,}8\ ns$
- $\Rightarrow 800\,000\ ns / 200\,000{,}8\ ns \cong 3{,}999984$

Per 1000 istruzioni 800 ns/200,8ns = 3,984

Miglioramento delle prestazioni (cont.)

Caso ideale (Caso asintotico):

- La **latenza** (tempo di esecuzione totale) della singola istruzione di *load* è **peggiorata** perché è passata da *800 ps* a *1000 ps*

- Il **throughput** (numero di istruzioni completate nell'unità di tempo) è **migliorato** di *4* volte:
  (1 istruzione di load completata ogni *800 ps)* vs.
  (*1* istruzione di load completata ogni *200 ps).*

Miglioramento delle prestazioni (cont.)

Se avessimo considerato una *CPU1* a singolo ciclo da *1000 ps* composta da *5* passi ciascuno da *200 ps* e una *CPU2* pipeline ideale (caso asintotico) con *5* stadi da *200 ps* avremmo trovato:

- La **latenza** di ogni singola istruzione rimane **invariata** e pari a *1000 ps.*
- Il **throughput** è **migliorato** di *5* volte: (*1* istruzione di *load* completata ogni *1000 ps) vs. (1* istruzione completata ogni 200 ps).

Considerazioni sulla struttura delle istruzioni MIPS

- Le istruzioni MIPS hanno tutte la stessa lunghezza: semplificazione per le fasi di prelievo e decodifica delle istruzioni
- Le istruzioni MIPS hanno un numero molto ridotto di formati diversi e, non considerando il registro destinazione delle istruzioni aritmetico-logiche, gli altri *due registri* sono sempre specificati nella stessa posizione (*rs* e *rt*) dell'istruzione codificata
- Gli operandi residenti in memoria sono presenti solo nelle istruzioni di *load* e *store*
- L'allineamento degli operandi in memoria è obbligatorio: il trasferimento dati con la memoria avviene sempre con un solo accesso

Passi svolti durante l'esecuzione delle istruzioni

Istruzioni aritmetico-logiche: `op $x,$y,$z`

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente `$y` e `$z`	Op. ALU sui Dati Letti (`$y op $z`)	Scrittura nel Reg. Destinazione `$x`

Istruzioni di caricamento *(load)*: `lw $x,offset($y)`

Prelievo Istruz. & Increm. PC	Lettura Registro Base `$y`	Op. ALU (`$y+offset`)	Prelievo Dato `M($y+offset)`	Scrittura nel Reg. Destinazione `$x`

Istruzioni di memorizzazione *(store)*: `sw $x,offset($y)`

Prelievo Istruz. & Increm. PC	Lettura Registri Base `$y` & Sorg. `$x`	Op. ALU (`$y+offset`)	Scrittura Dato `M($y+offset)`

Istruzioni di salto condizionato: `beq $x,$y,offset`

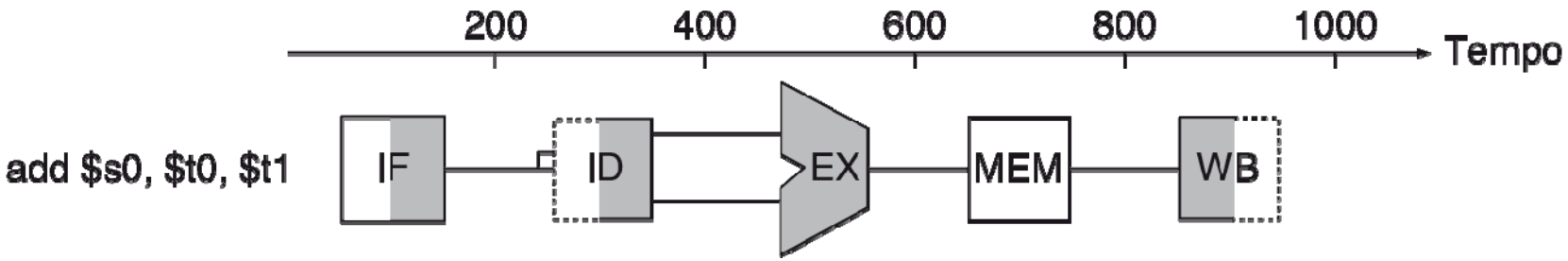
Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente `$x` e `$y`	Op. ALU (`$x-$y`) & (`PC+4+offset`)	Scrittura nel PC

Istruzioni di salto non condizionato: `j L1`

Prelievo Istruz. & Increm. PC		(`PC+4|L1|00`)	Scrittura nel PC

Passi svolti durante l'esecuzione delle istruzioni in modalità pipeline: formalizzazione

IF (Instruction Fetch)	**ID** (Instruction Decode)	**EX** (Execution)	**MEM** (Memory Access)	**WB** (Write Back)
Prelievo Istruzione	Decodifica Istruzione e lettura registri	Esecuzione	Accesso alla memoria dati	Riscrittura registri



Passi svolti durante l'esecuzione delle istruzioni in modalità pipeline

Prelievo Istruzione **IF** (Instruction Fetch)	Decodifica Istruzione **ID** (Instruction Decode)	Esecuzione **EX** (Execution)	Accesso alla Memoria **MEM** (Memory Access)	Riscrittura Registri **WB** (Write Back)

Istruzioni aritmetico-logiche: `op $x,$y,$z`

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente `$y` e `$z`	Op. ALU sui Dati Letti (`$y op $z`)		Scrittura nel Reg. Destinazione `$x`

Istruzioni di caricamento *(load)*: `lw $x,offset($y)`

Prelievo Istruz. & Increm. PC	Lettura Registro Base `$y`	Op. ALU (`$y+offset`)	Prelievo Dato `M($y+offset)`	Scrittura nel Reg. Destinazione `$x`

Istruzioni di memorizzazione *(store)*: `sw $x,offset($y)`

Prelievo Istruz. & Increm. PC	Lettura Registri Base `$y` & Sorg. `$x`	Op. ALU (`$y+offset`)	Scrittura Dato `M($y+offset)`	

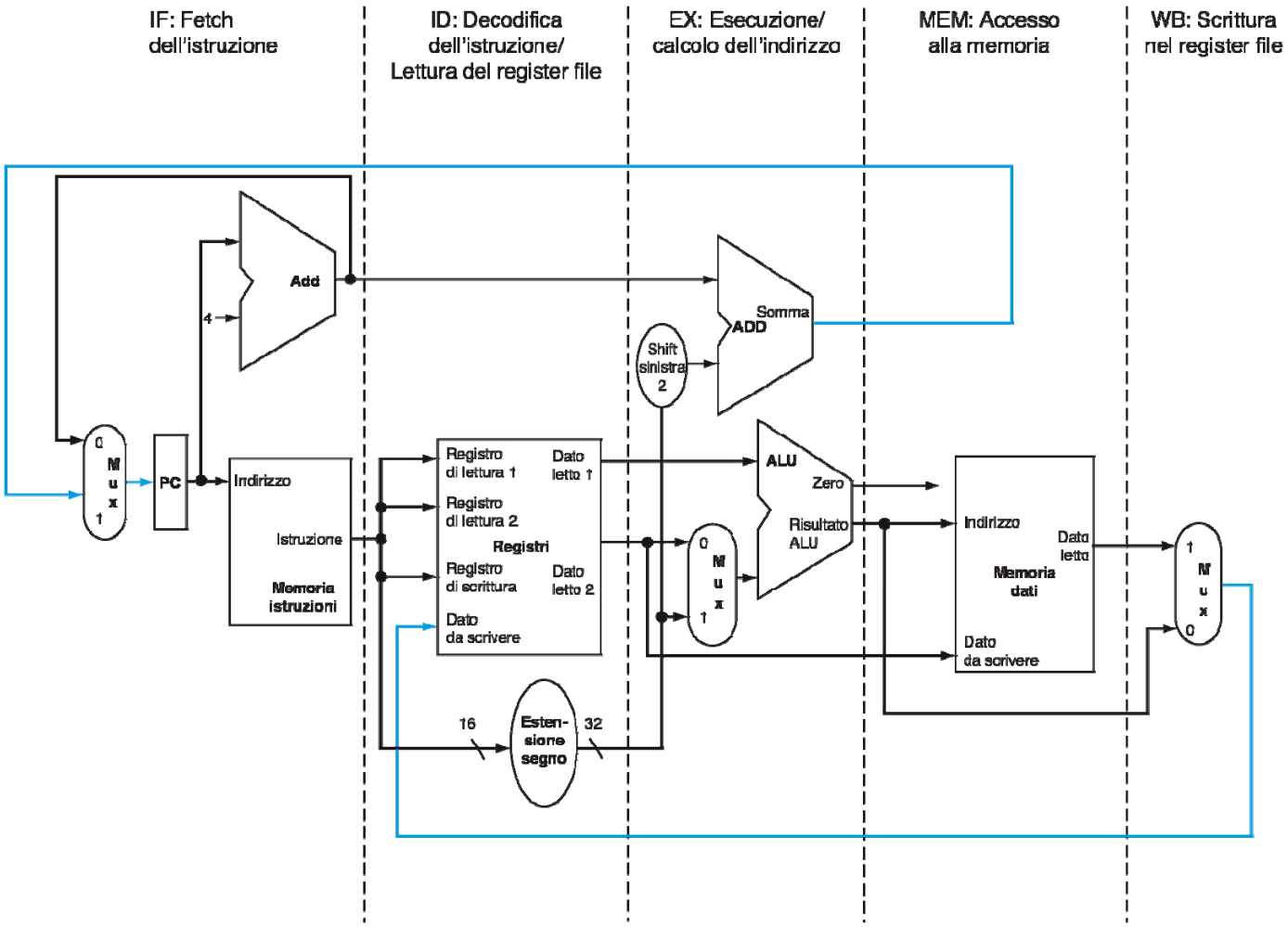
Istruzioni di salto condizionato: `beq $x,$y,offset`

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente `$x` e `$y`	Op. ALU (`$x-$y`) & (`PC+4+offset`)	Scrittura nel PC	

Istruzioni di salto non condizionato: `j L1`

Prelievo Istruz. & Increm. PC		(`PC+4|L1|00`)	Scrittura nel PC	

Fasi (passi) e stadi



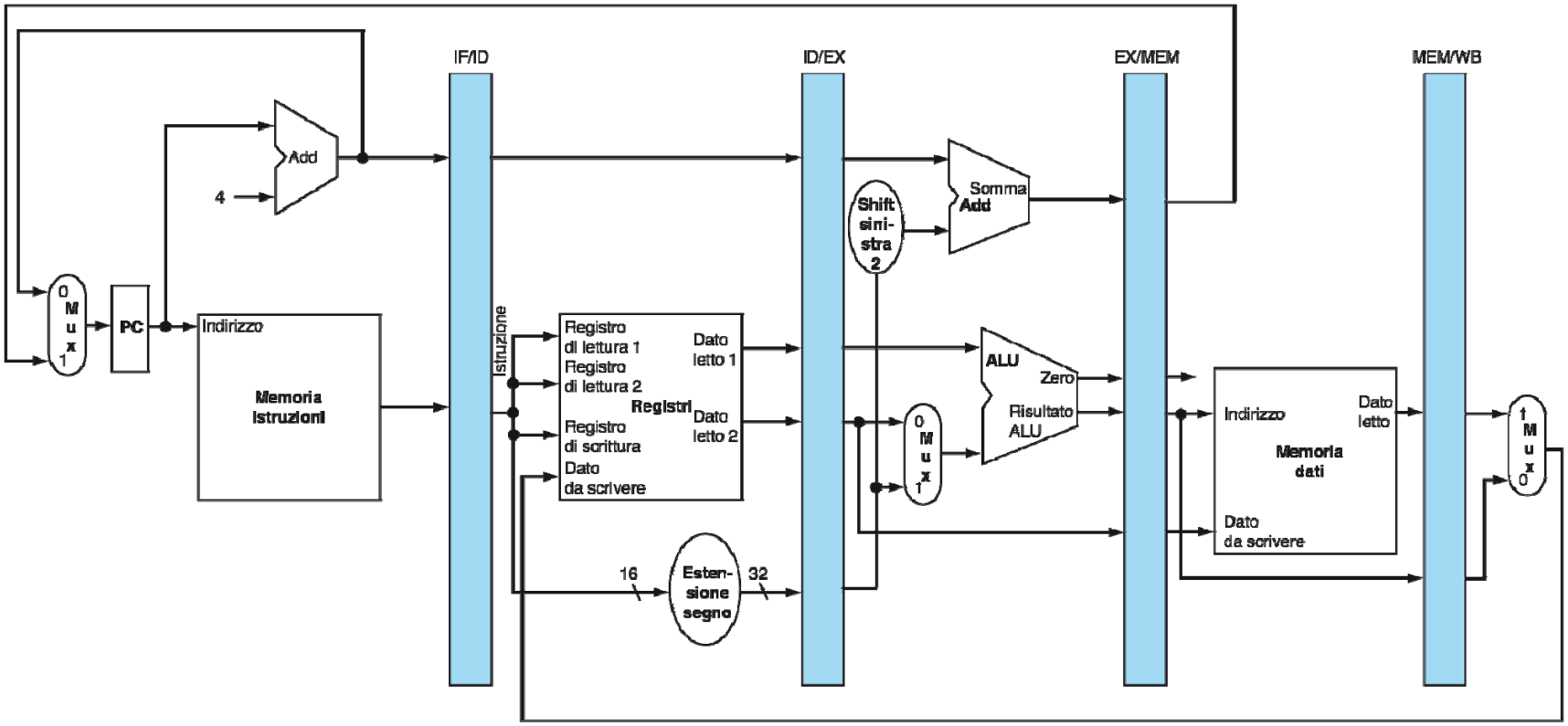
Struttura pipeline della CPU MIPS

La suddivisione dell'esecuzione di un'istruzione in *5* passi implica che in ogni ciclo di clock siano in esecuzione *5* istruzioni

- la struttura di una *CPU* pipeline a 5 stadi deve essere scomposta in ***5*** **parti** o ***stadi di esecuzione*****,** ciascuno corrispondente ad una delle fasi di pipeline

- devono essere introdotti ***registri di pipeline*** che separano i diversi stadi.

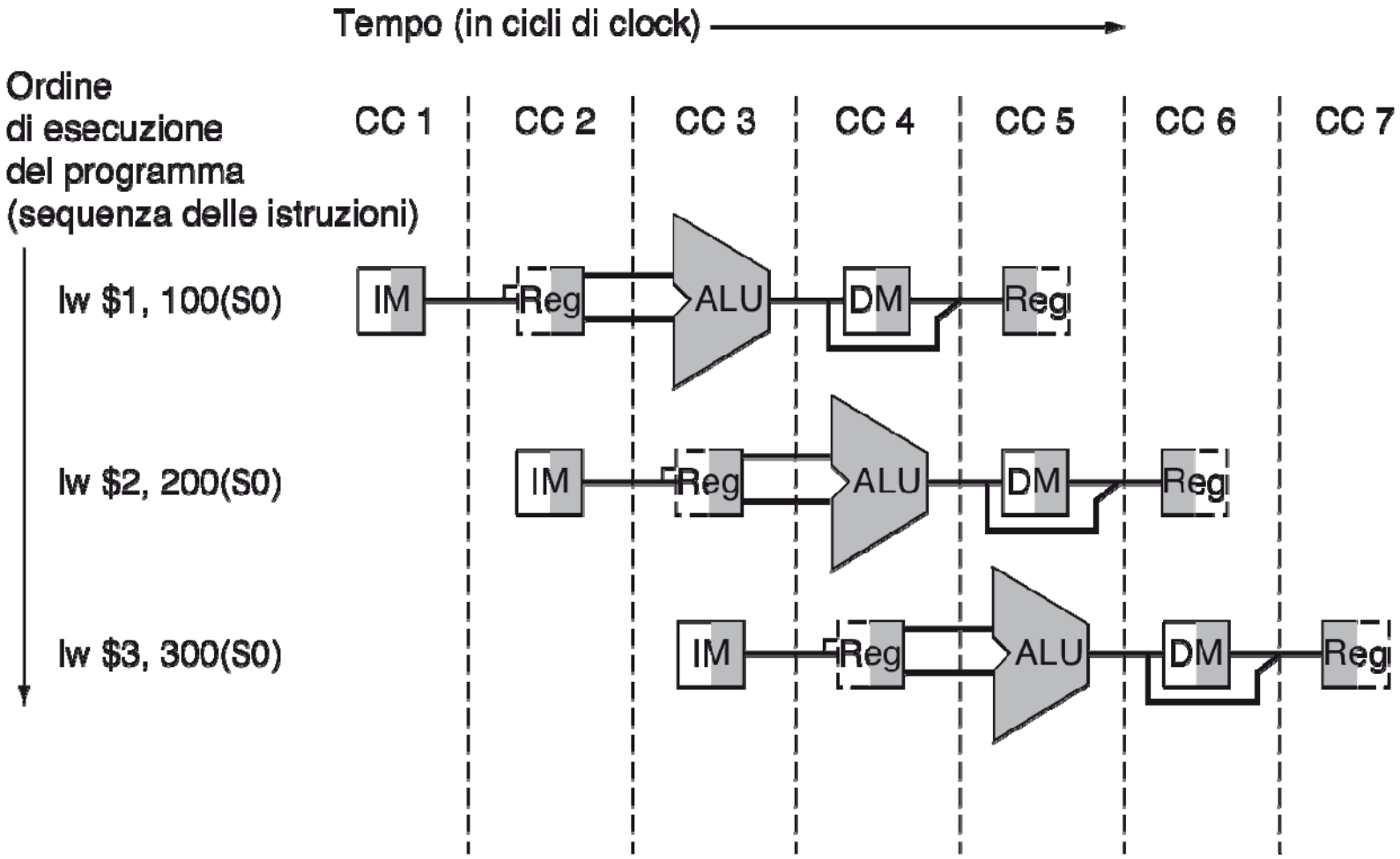
Struttura pipeline della CPU MIPS (cont.)



dimensione registri di pipeline: IF/ID (32), ID/EX(128), EX/MEM(97), MEM/WB(64)

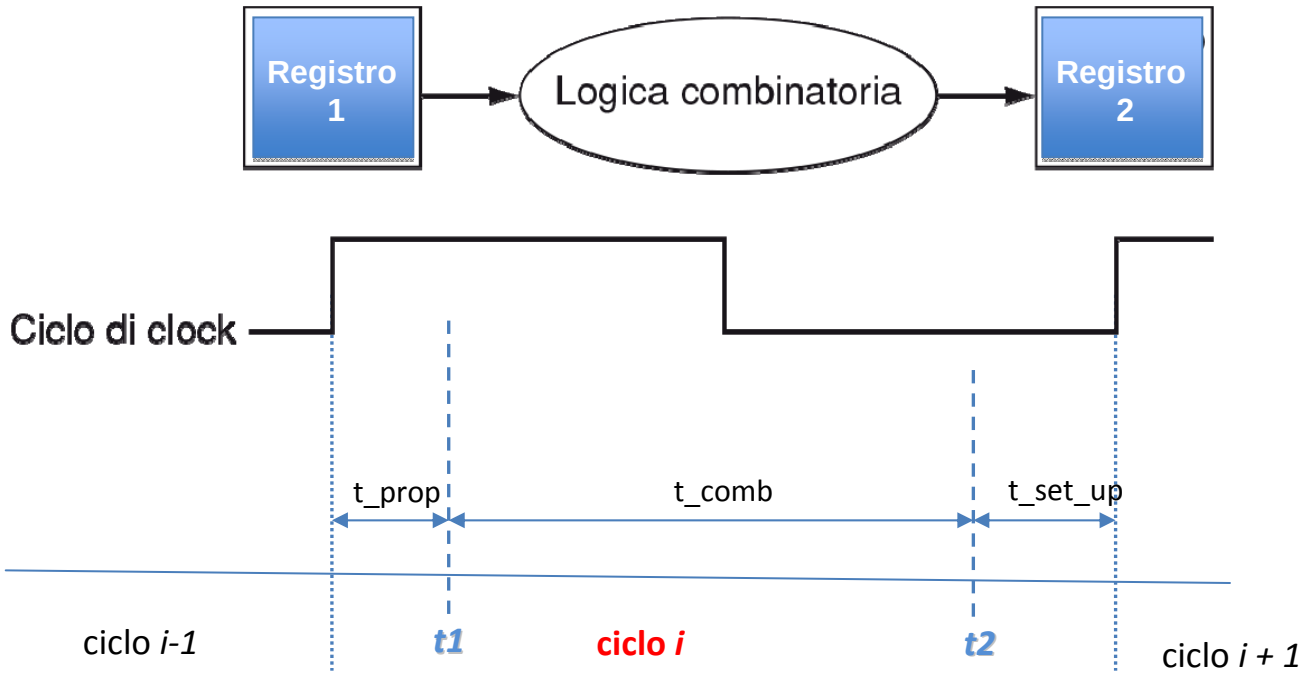
"ruolo" del registro PC: *registro di pipeline* dello stadio IF

Utilizzo delle risorse



Temporizzazione ***lettura e scrittura registri interstadio***

Registri interstadio e temporizzazione *edge-triggered*



Considerato un certo stadio

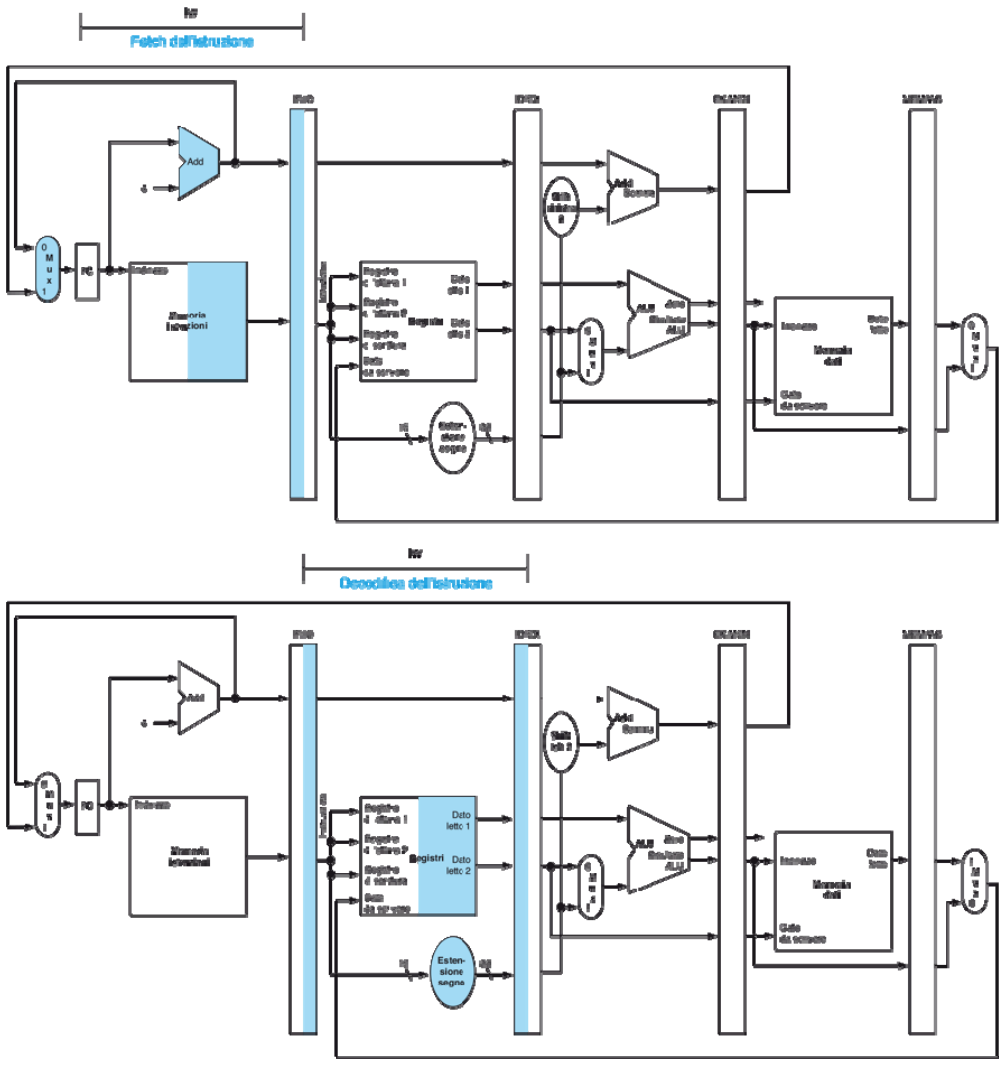
- Registro1 = registro interstadio di ingresso
- Logica combinatoria = elementi funzionali dello stadio
- Registro2 = registro interstadio di uscita

Sul fronte di salita che termina il ciclo *i-1*, i dati impostati in *i-1* vengono scritti in *Registro1.* Nel ciclo *i*:

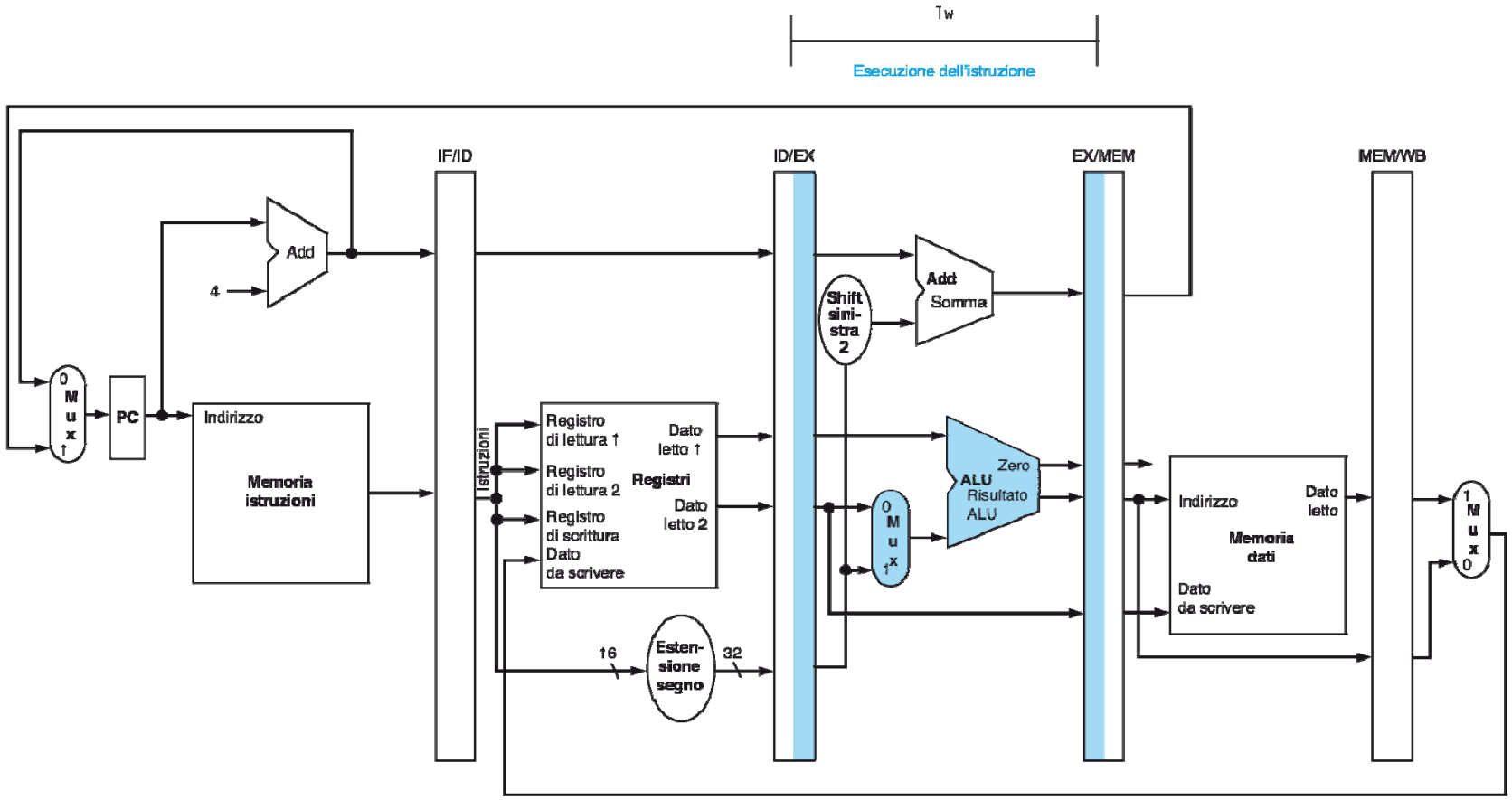
➢a partire da *t1* (quindi dopo *t_prop)* le uscite di *Registro1* sono da considerarsi stabili e vengono usate (lette) come ingressi per far lavorare lo stadio

➢a partire da *t2* e per *t_setup* le uscite degli elementi funzionali dello stadio sono da considerarsi stabili per poter essere scritte in *Registro2* sul fronte di salita di chiusura del ciclo i

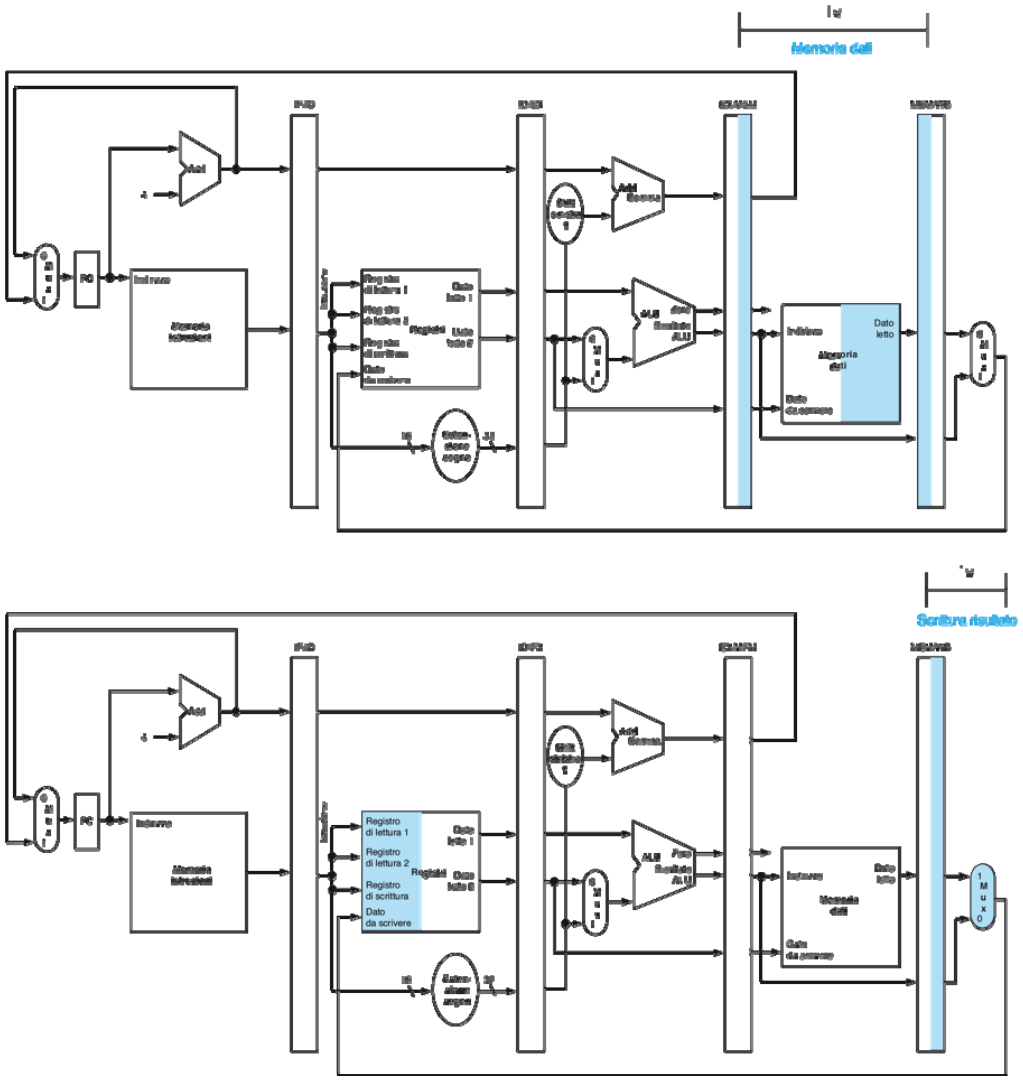
Fetch e decodifica di *load*



Esecuzione di *load*



Accesso a memoria e scrittura in registro di *load*



Struttura pipeline della CPU MIPS (cont.)

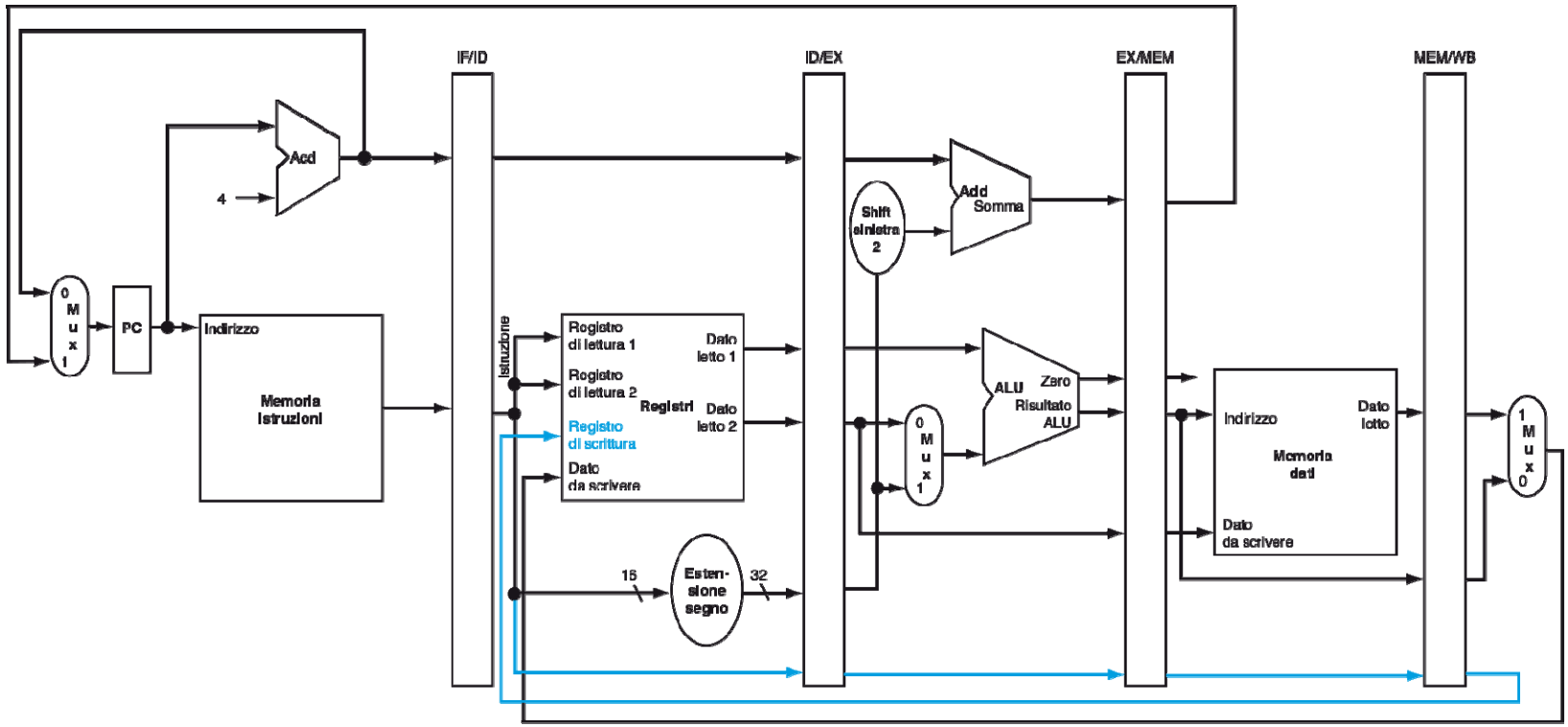
Le informazioni memorizzate nei registri interstadio sono relative ad ***istruzioni diverse***!

L'istruzione memorizzata nel registro di pipeline *IF/ID* fornisce il numero del registro di scrittura, mentre i dati scritti sono quelli relativi all'istruzione che si trova nel registro *MEM/WB*

- occorre modificare la *CPU* in modo da trasmettere attraverso i registri di pipeline (*ID/EX*, *EX/MEM* e *MEM/WB*) l'indirizzo del registro da scrivere durante lo stadio *WB*.

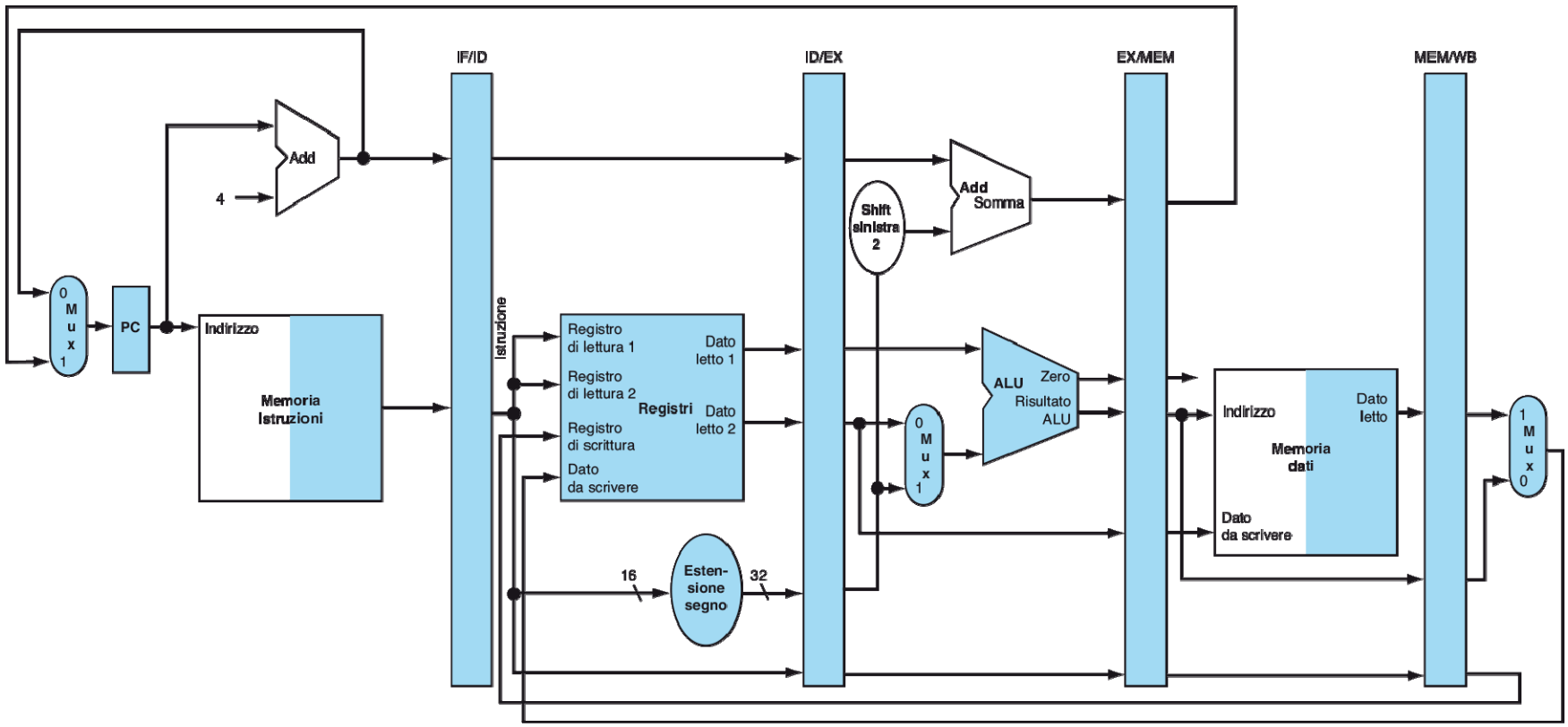
L'indirizzo del registro di scrittura, che viene fatto passare attraverso gli stadi intermedi, proviene dal registro di pipeline *MEM/WB* insieme ai dati.

Passaggio di informazioni tra stadi

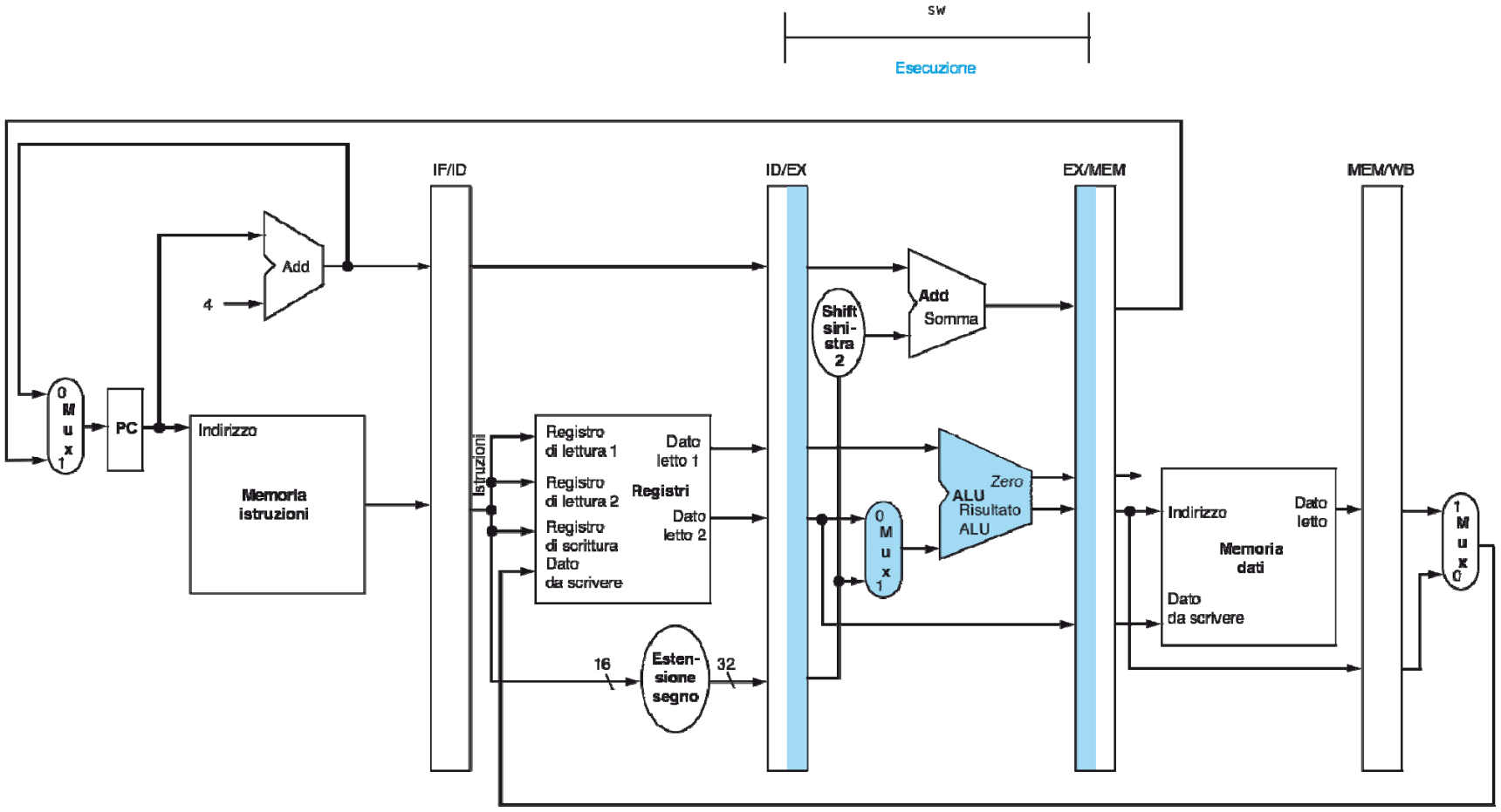


modifica registri di pipeline per propagazione indirizzo registro di scrittura

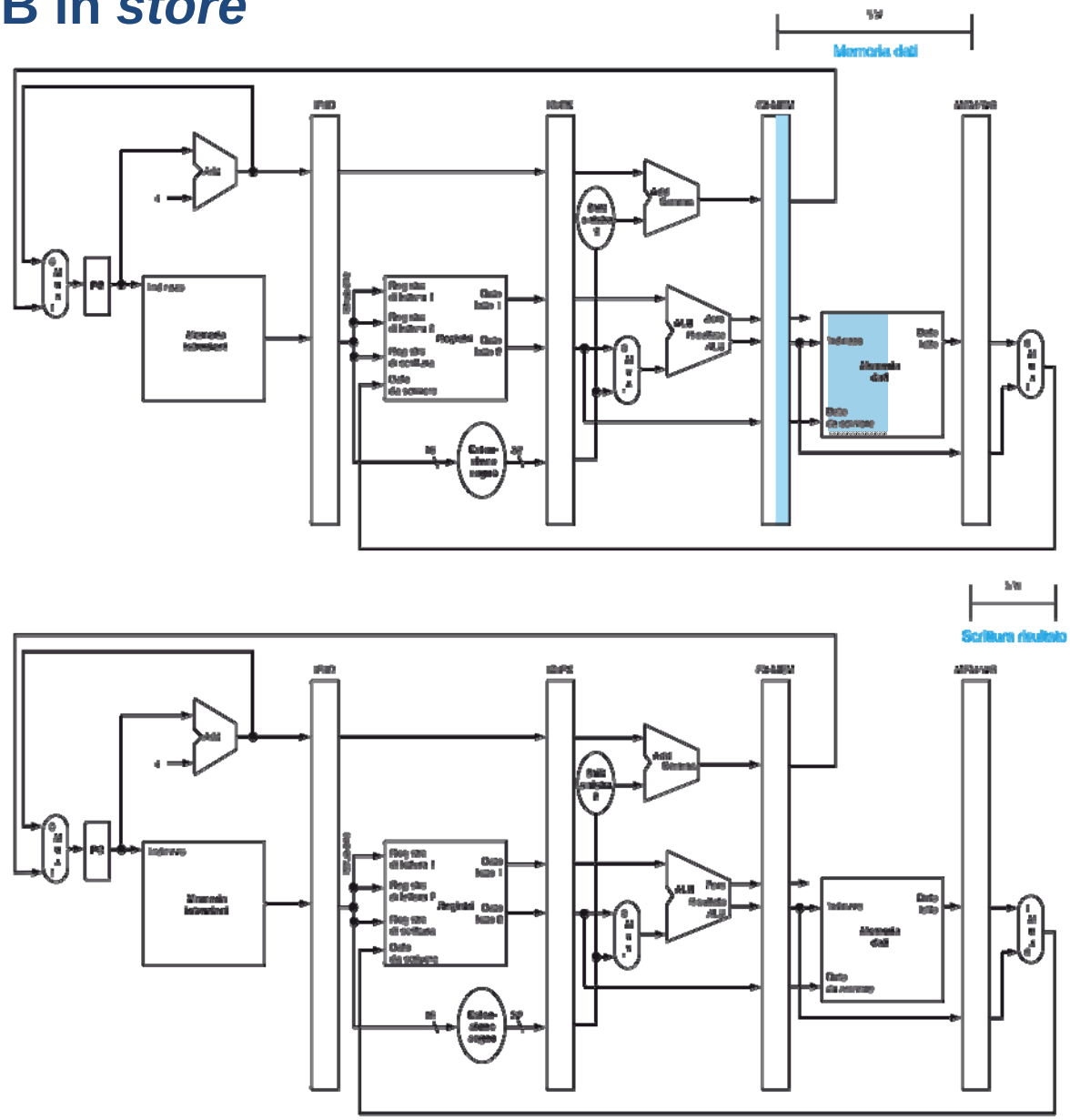
Risorse utilizzate per eseguire la *load*



Esecuzione di *store*



MEM e WB in *store*



Rappresentazione di 5 istruzioni in esecuzione (1)

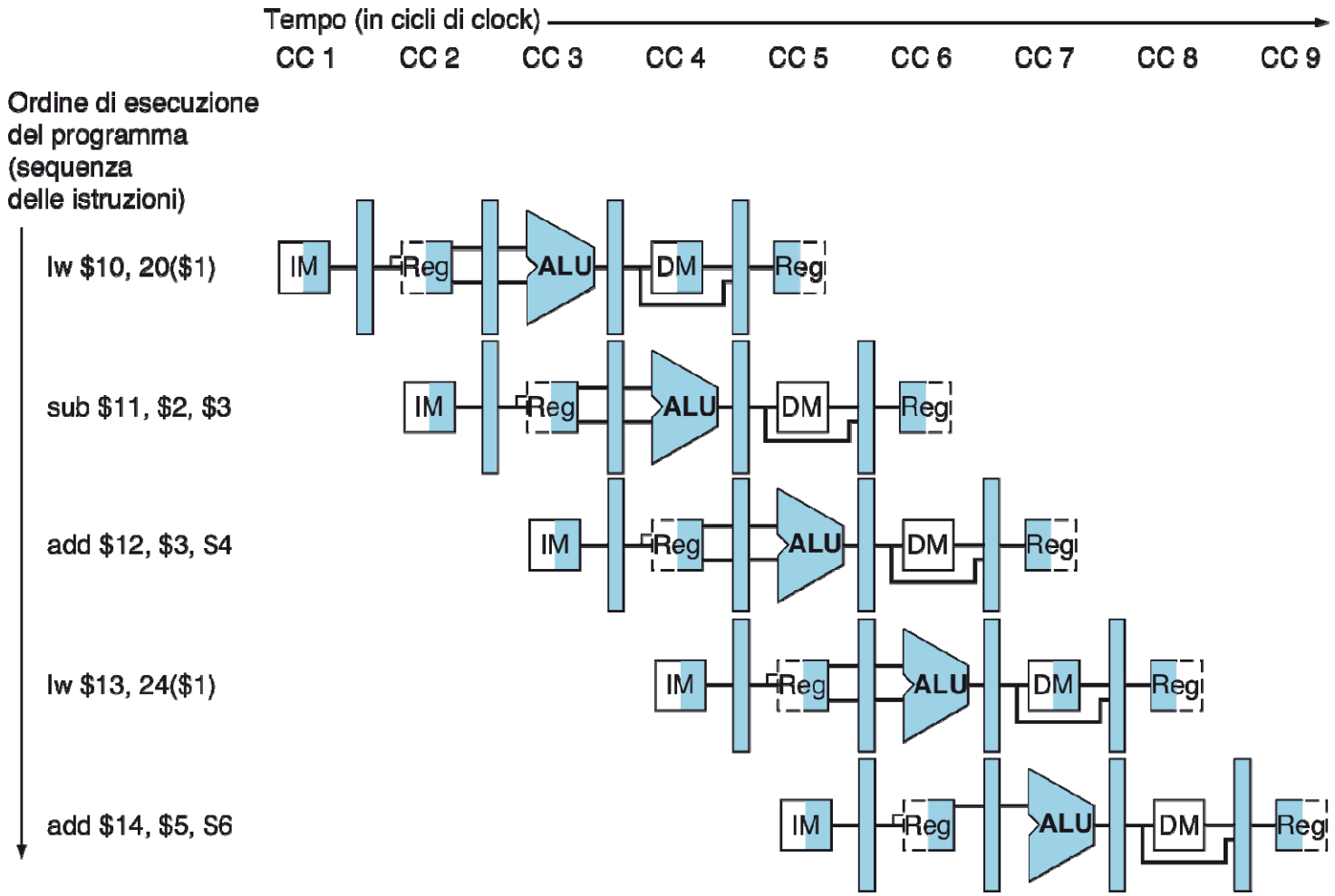


diagramma pipeline a più cicli con risorse fisiche in uso

Rappresentazione di 5 istruzioni in esecuzione (2)

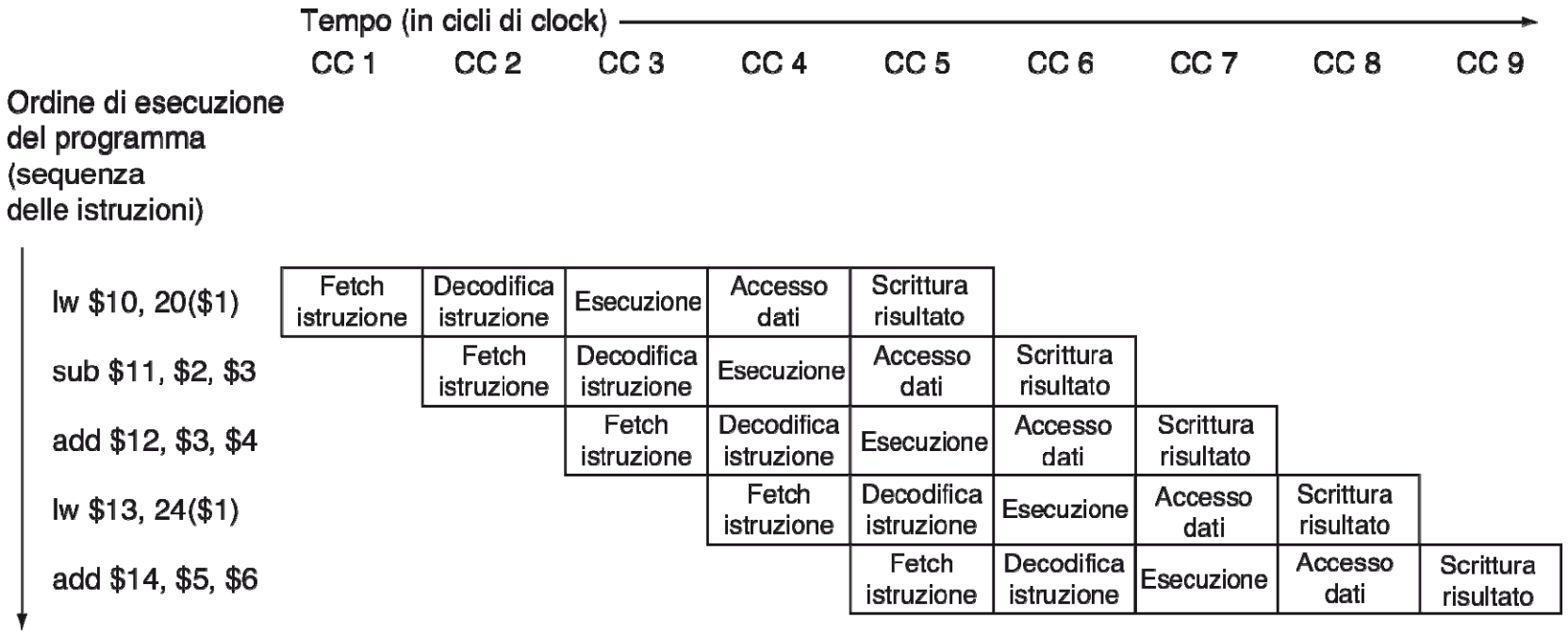
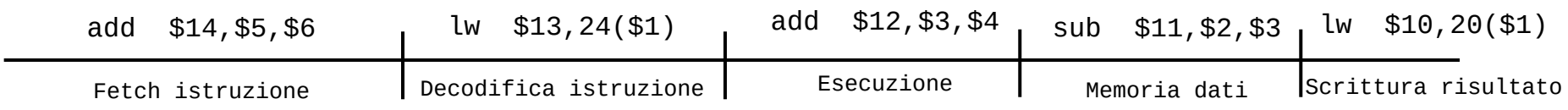
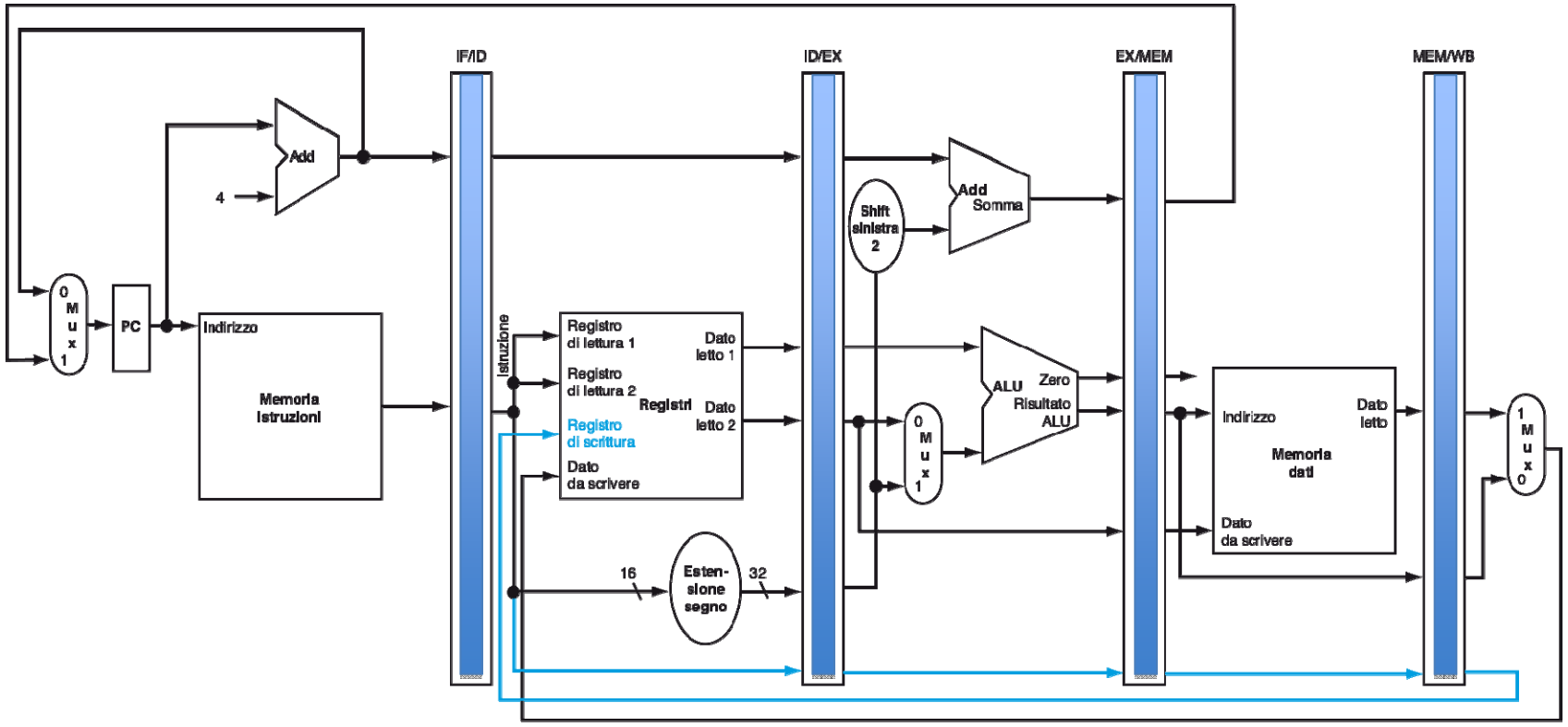


diagramma semplificato pipeline a più cicli

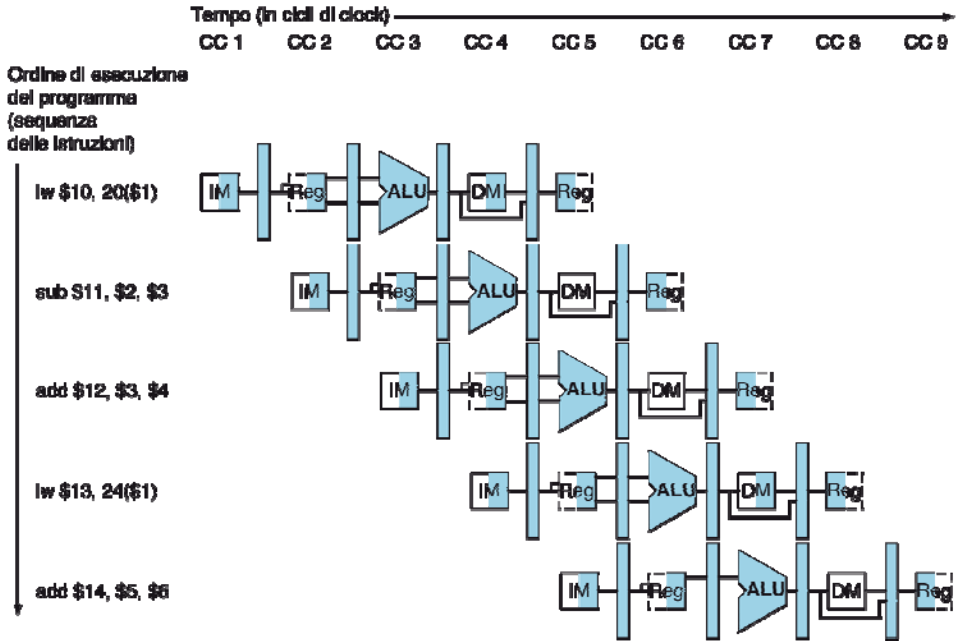
Rappresentazione di 5 istruzioni in esecuzione (3)





pipeline corrispondente ad un certo ciclo di clock con indicazione delle istruzioni nei vari stadi

Ancora sulla temporizzazione: il Register File e la sua temporizzazione in scrittura



In CC5 in ingresso al **Register File** si ha

•da **IF/ID** indirizzo registro da leggere dell'istruzione che ha iniziato l'esecuzione (istruzione lw $13, …)

•da **MEM/WB** indirizzo dato da scrivere, valore da scrivere, segnale **RegWrite** a 1 (istruzione lw $10, …)

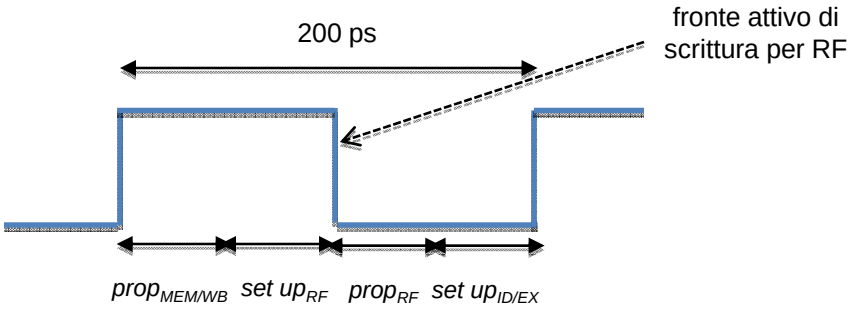
Al termine di CC5

•l'operazione di **scrittura deve essere terminata** (segnale RegWrite non più asserito per l'istruzione che la richiede): questo implica che il **fronte attivo *per la scrittura di RF*** deve essere quello di **discesa**

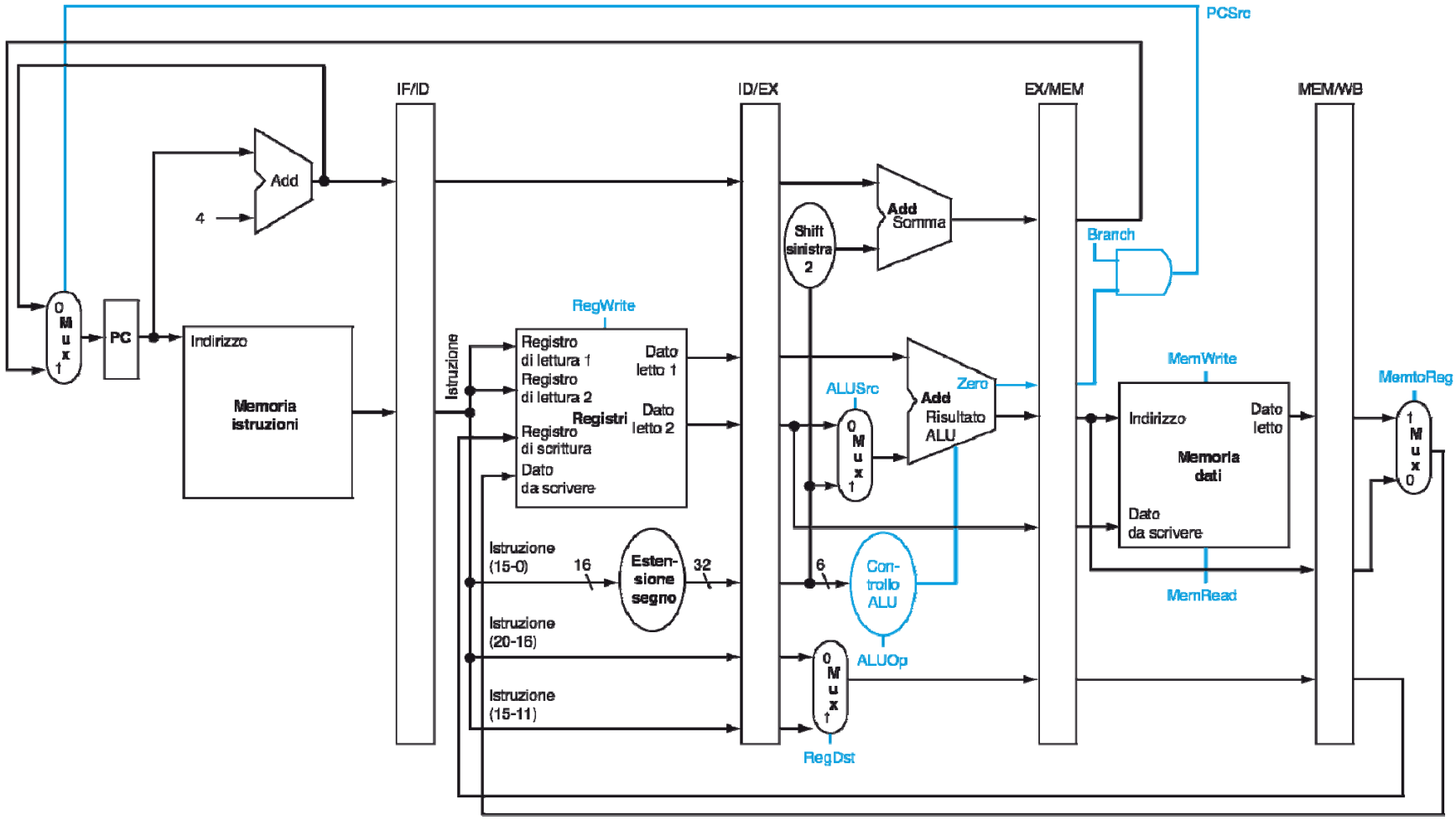
•in ID/EX ci deve essere il valore corretto del registro letto che serve all'istruzione che ha iniziato l'esecuzione

Se in uno stesso ciclo di clock si vuole che una lettura di un registro di RF restituisca il valore correntemente scritto **dello stesso registro** è necessario che

•$t\_prop_{MEM/WB} + t\_set\ up_{RF} <= 100$ ps *(scrittura RF)*
•*a 100 ps fronte di discesa e scrittura in RF*
•$t\_prop_{RF} + t\_set\ up_{ID/EX} <= 100$ ps *(lettura RF)*



Segnali dell'unità di controllo della pipeline



Segnali di controllo per la ALU (come CPU a singolo ciclo)

Codice operativo istruzione	ALUOp	Operazione associata all'istruzione	Codice funzione	Operazione della ALU	Input di controllo della ALU
LW	00	load word	XXXXXX	somma	0010
SW	00	store word	XXXXXX	somma	0010
Branch equal	01	branch equal	XXXXXX	sottrazione	0110
Tipo R	10	somma	100000	somma	0010
Tipo R	10	sottrazione	100010	sottrazione	0110
Tipo R	10	AND	100100	AND	0000
Tipo R	10	OR	100101	OR	0001
Tipo R	10	set less than	101010	set less than	0111

Si ricordi, a proposito delle ***4 linee di controllo dell'ALU***, l'osservazione già riportata per la CPU a singolo ciclo

Effetto segnali di controllo (come per la CPU a singolo ciclo)

Nome del segnale	Effetto quando non asserito (0)	Effetto quando asserito (1)
RegDst	Il numero del registro di scrittura proviene dal campo rt (bit 20-16)	Il numero del registro di scrittura proviene dal campo rd (bit 15-11)
RegWrite	Nulla	Il dato viene scritto nel registro (del register file) individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 16 bit meno significativi dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nulla	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea «dato letto»
MemWrite	Nulla	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea «dato scritto»
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

Segnali di controllo e registri di pipeline

Ai primi due stadi ***IF*** e ***ID*** non sono associati segnali di controllo espliciti

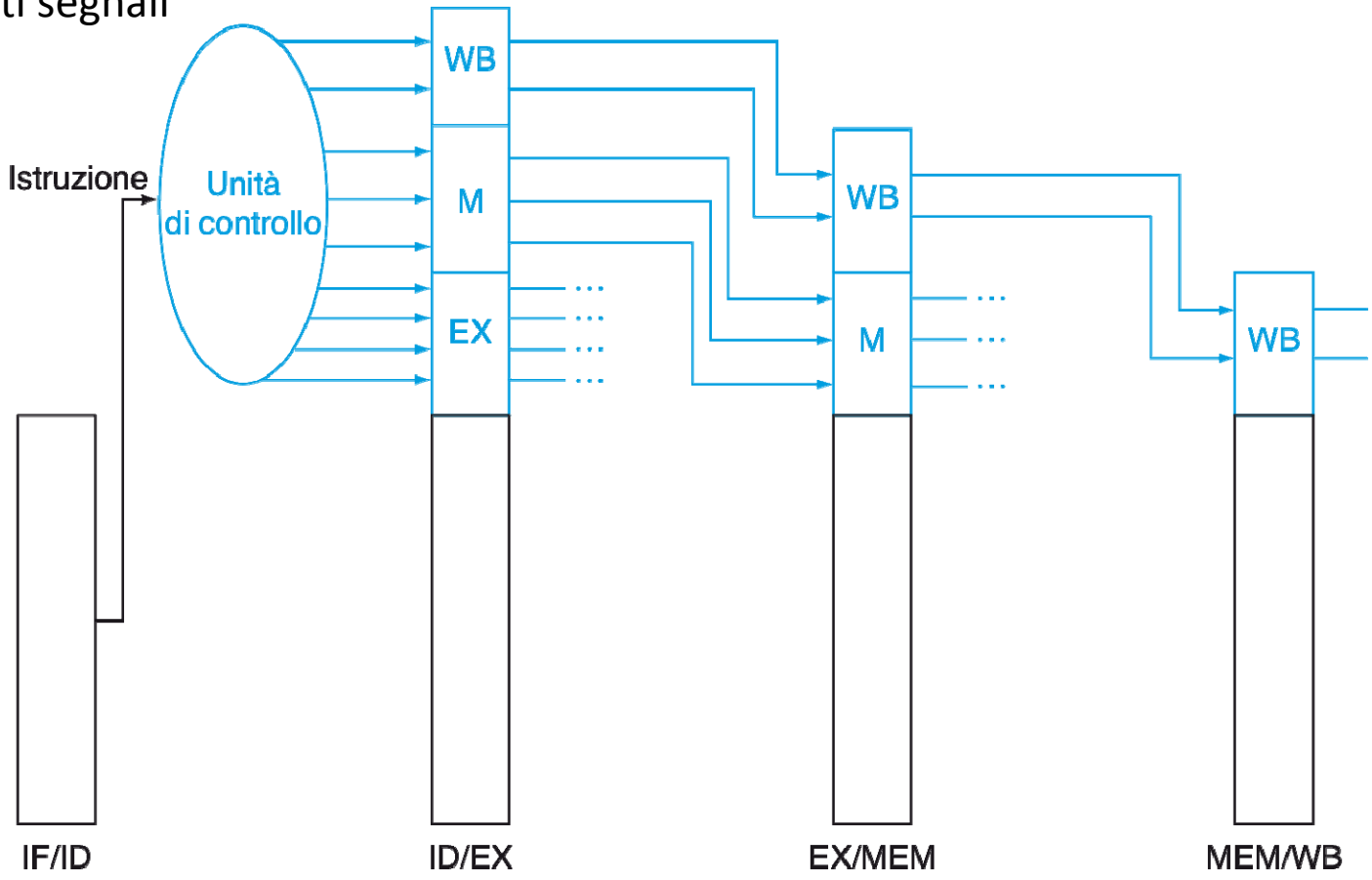
- l'unico considerato è il clock e il suo fronte attivo

Istruzione	Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo				Segnali di controllo dello stadio di accesso alla memoria dati			Segnali di controllo dello stadio di scrittura	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	RegWrite	Memto-Reg
Formato R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

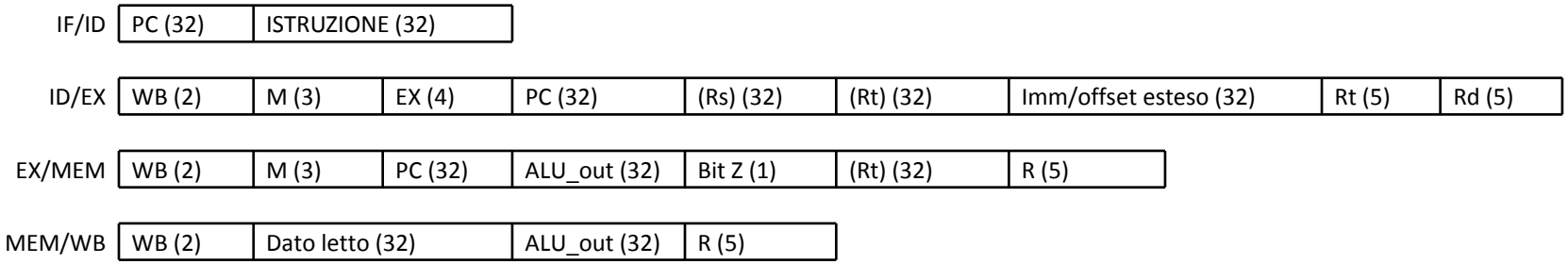
I valori dei segnali vengono tutti generati a partire dal contenuto del registro ***IF/ID*** che – dopo ogni fetch – contiene l'istruzione da eseguire e il valore del ***PC*** incrementato

Implementazione dell'unità di controllo

- E' necessario impostare correttamente il valore dei 9 segnali di controllo per ciascuno stadio della pipeline per ciascuna istruzione
- Modo più semplice: estendere i registri di pipeline per salvare e propagare questi segnali



Il dettaglio dei campi dei registri di pipeline



Legenda:
(32) ecc. = n° di bit del campo considerato
(Rs) o (Rt) = contenuto del registro Rs o Rt
Rt, Rd, R = numero di registro, se R può essere Rt o Rd

WB(2): RegWrite, MemtoReg; ***M(3)***: Branch, MemWrite, MemRead; ***EX(4)***: RegDest, ALUOp(2), ALUsrc

IF/ID.PC e ***ID/EX.PC*** = PC esecuzione in sequenza, ***EX/MEM.PC*** = PC branch taken

ID/EX.Rt = reg. destinazione se *load*, ***ID/EX.Rd*** = reg. destinazione se R

EX/MEM.ALU_out
rs op rt se R
rs – rt se beq, bne
rs + offset se lw/sw
rs op imm se arit/log con immediato

MEM/WB.ALU_out come EX/MEM.ALU_out ma significativi solo se
rs op rt se R
rs op imm se arit/log con immediato

EX/MEM.(Rt) significativo solo in caso di *store*

Unità di elaborazione completa con segnali di controllo