

Architettura degli Elaboratori

L'architettura della CPU – Gestione dei data hazard



SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Checco
checco@di.uniroma1.it

Special thanks and credits:

Andrea Sterbini, Iacopo Masi,
Claudio di Ciccio

[S&PdC]
4.4 - 4.7



Argomenti

Per la pipeline, lo schema adottato è quello di una **catena di montaggio**

- Obiettivo: fasi veloci (periodo di **clock** = durata della fase più lenta)
- **Ciascuna fase realizza** (localmente a un'istruzione) **un solo task**. Più task possono essere eseguiti in parallelo
- Ciascuna fase **riceve informazioni e segnali di controllo**
- Ciascuna fase **passa alla successiva le informazioni e segnali di controllo**
- I segnali necessari **devono restare stabili** durante tutta la fase

Si devono usare **elementi di stato** (es., latch) che cambiano solo al fronte di salita del clock

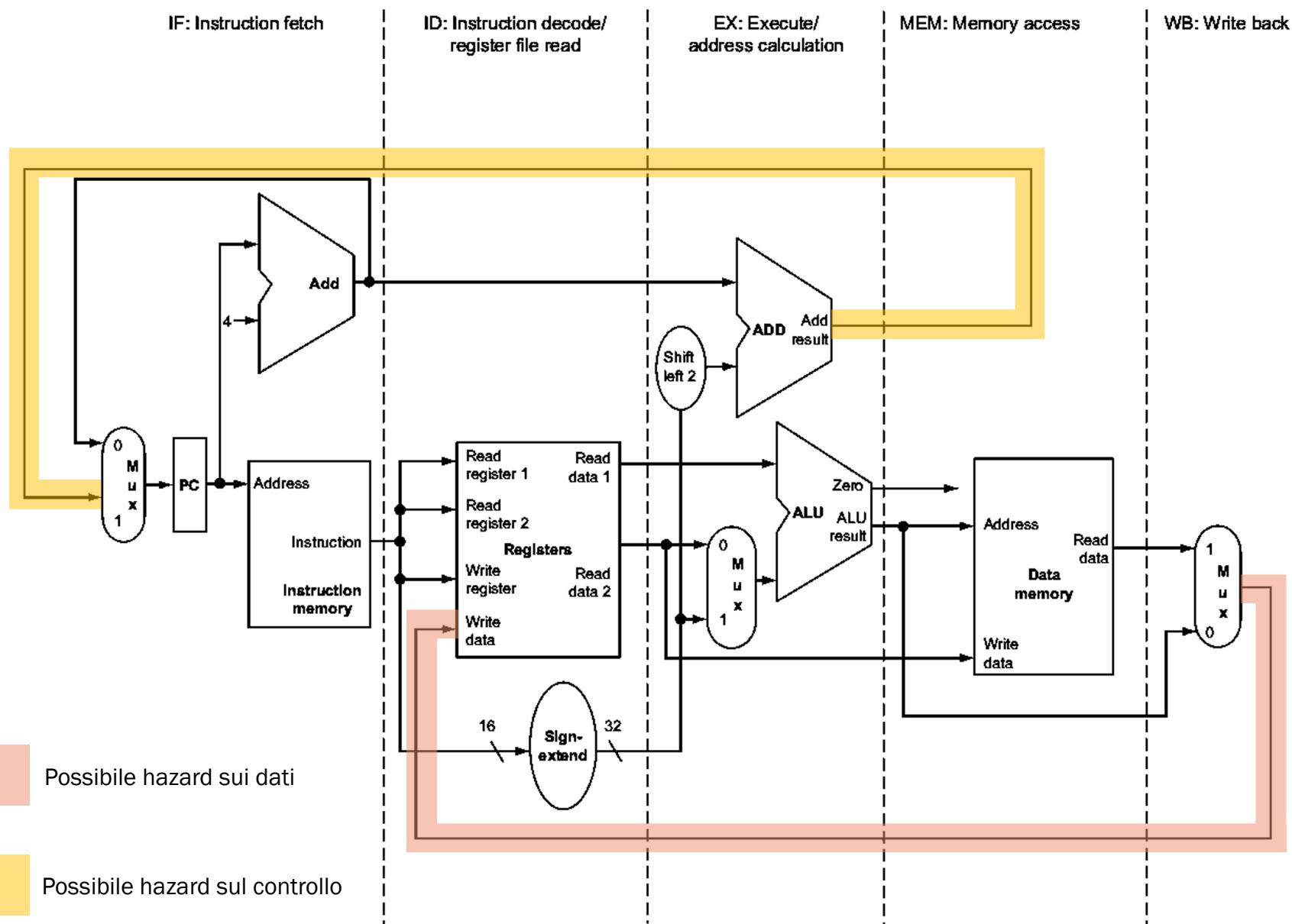
Soluzione: separare ciascuna fase dalla successiva con un **registro**

Architettura della CPU MIPS con pipeline



SAPIENZA
UNIVERSITÀ DI ROMA

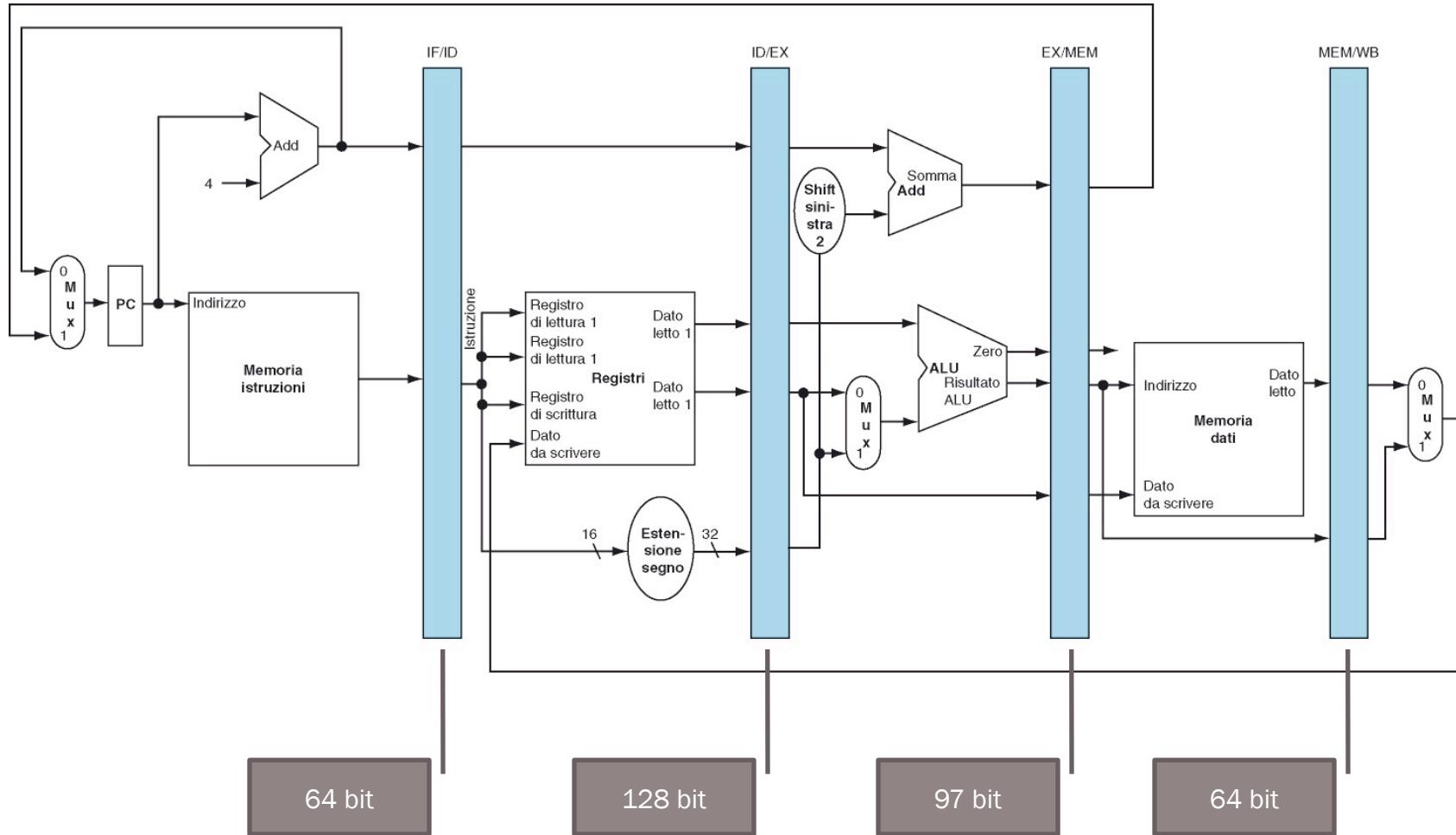
Unità funzionali delle 5 fasi



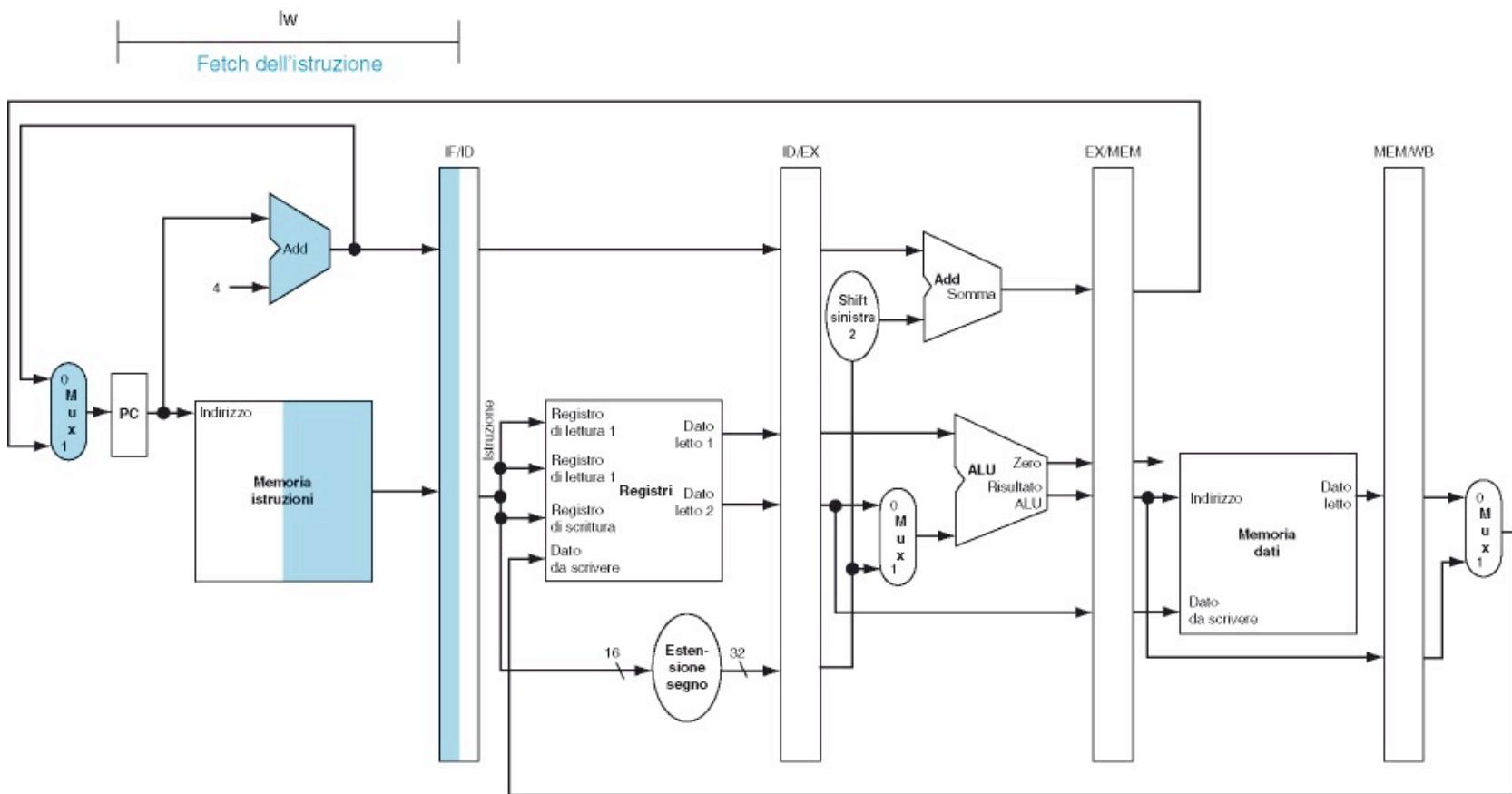
Possible hazard sui dati

Possible hazard sul controllo

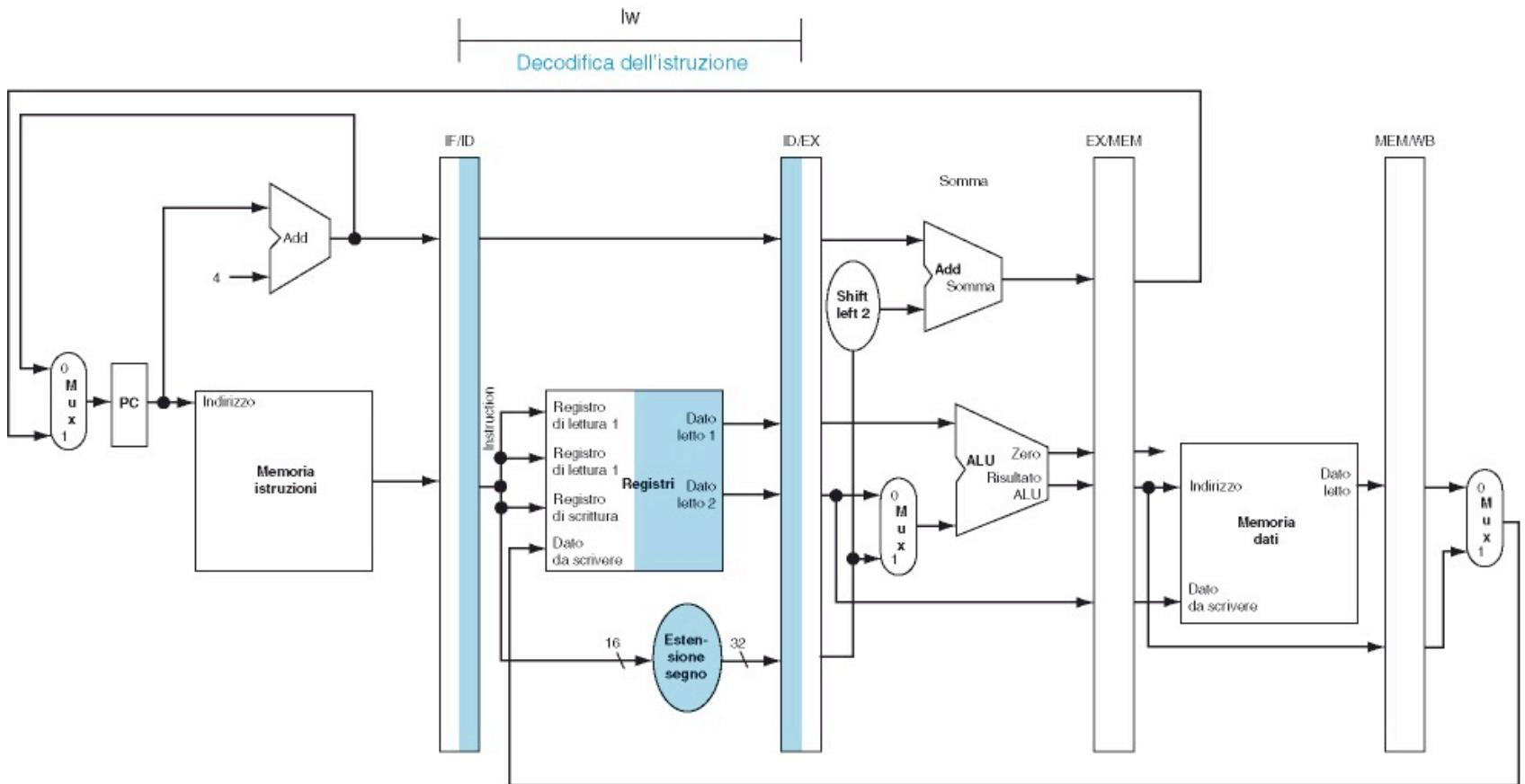
Inseriamo i registri tra le fasi



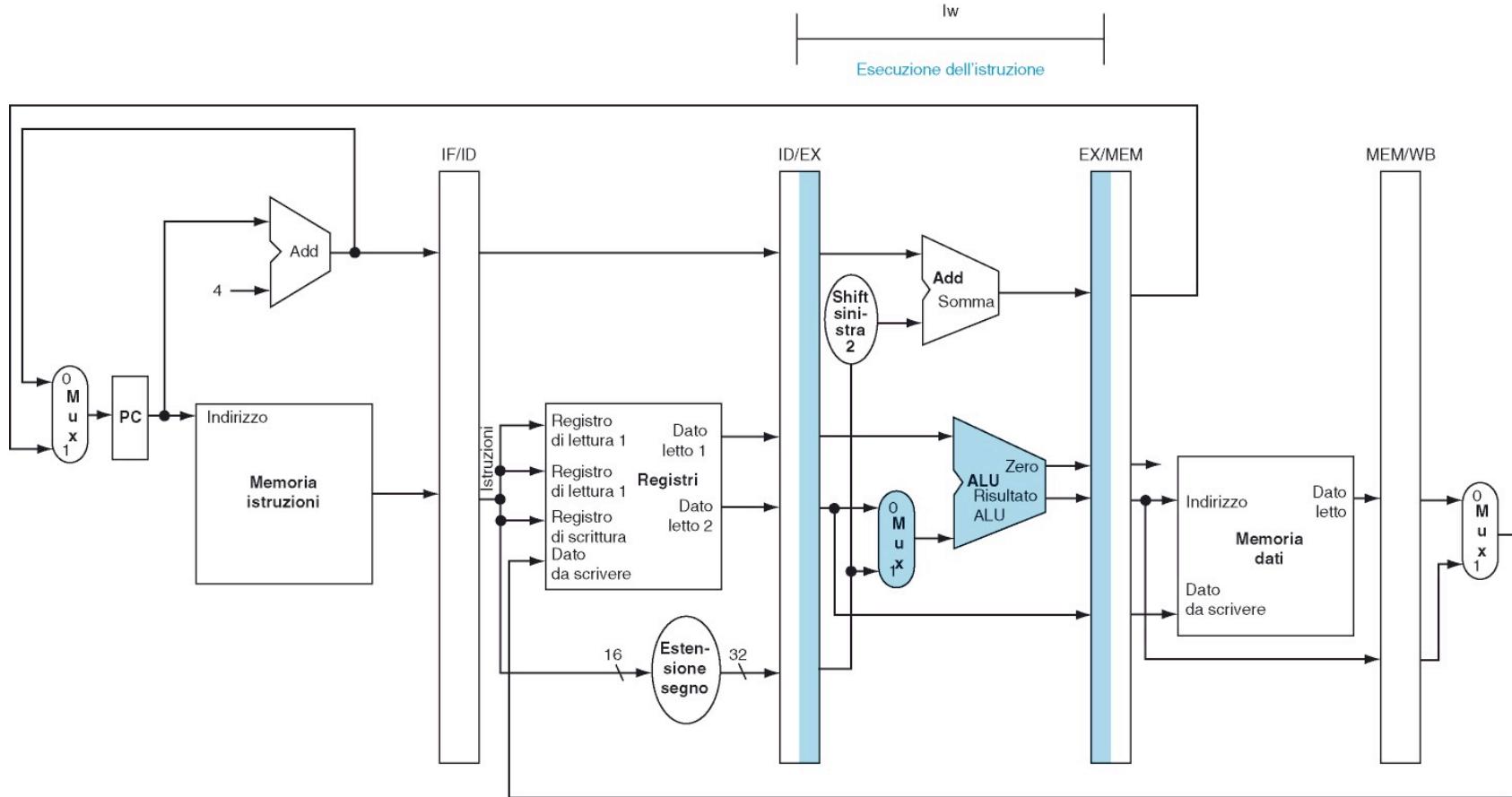
Instruction Fetch



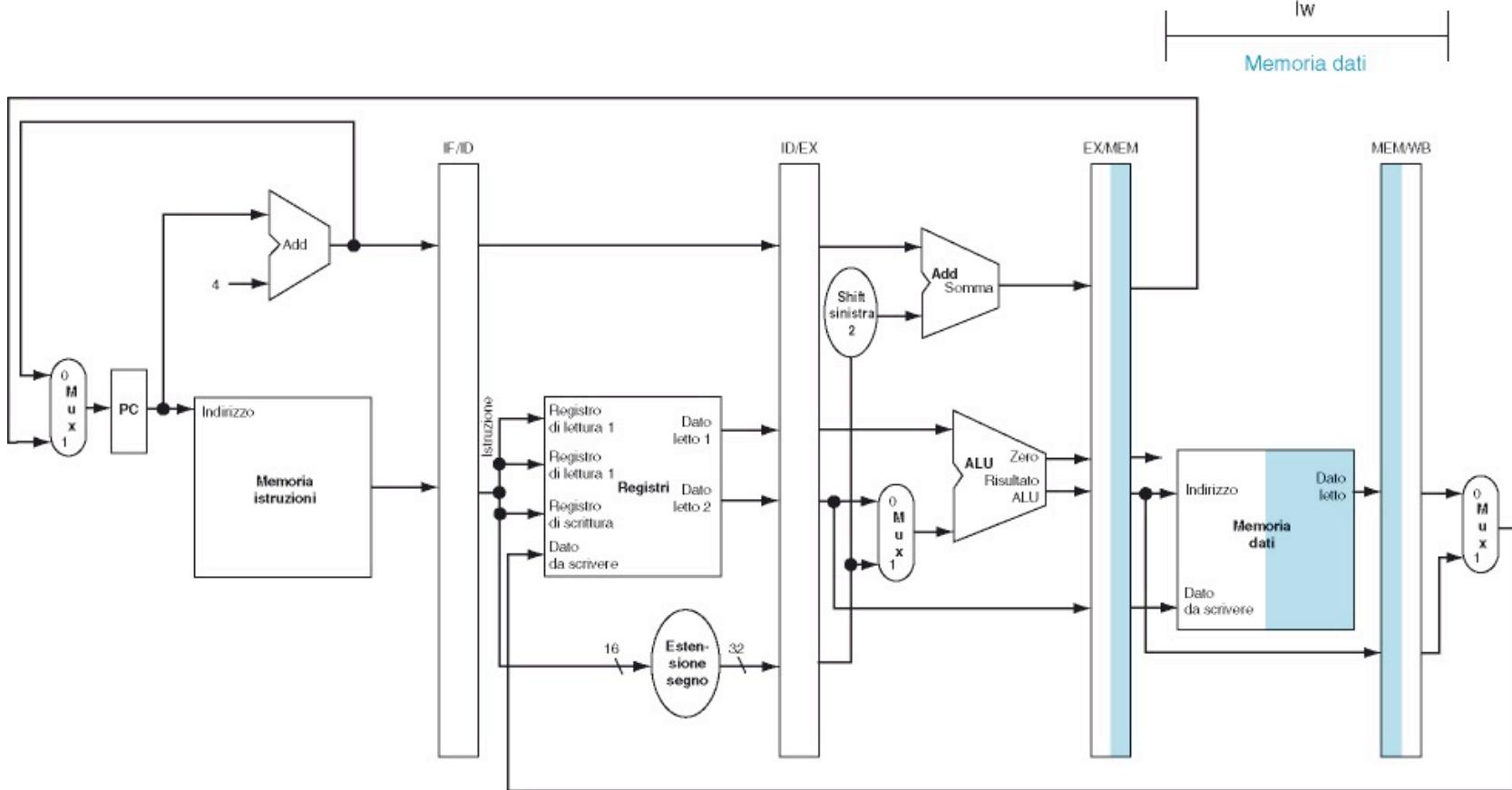
Instruction Decode



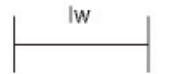
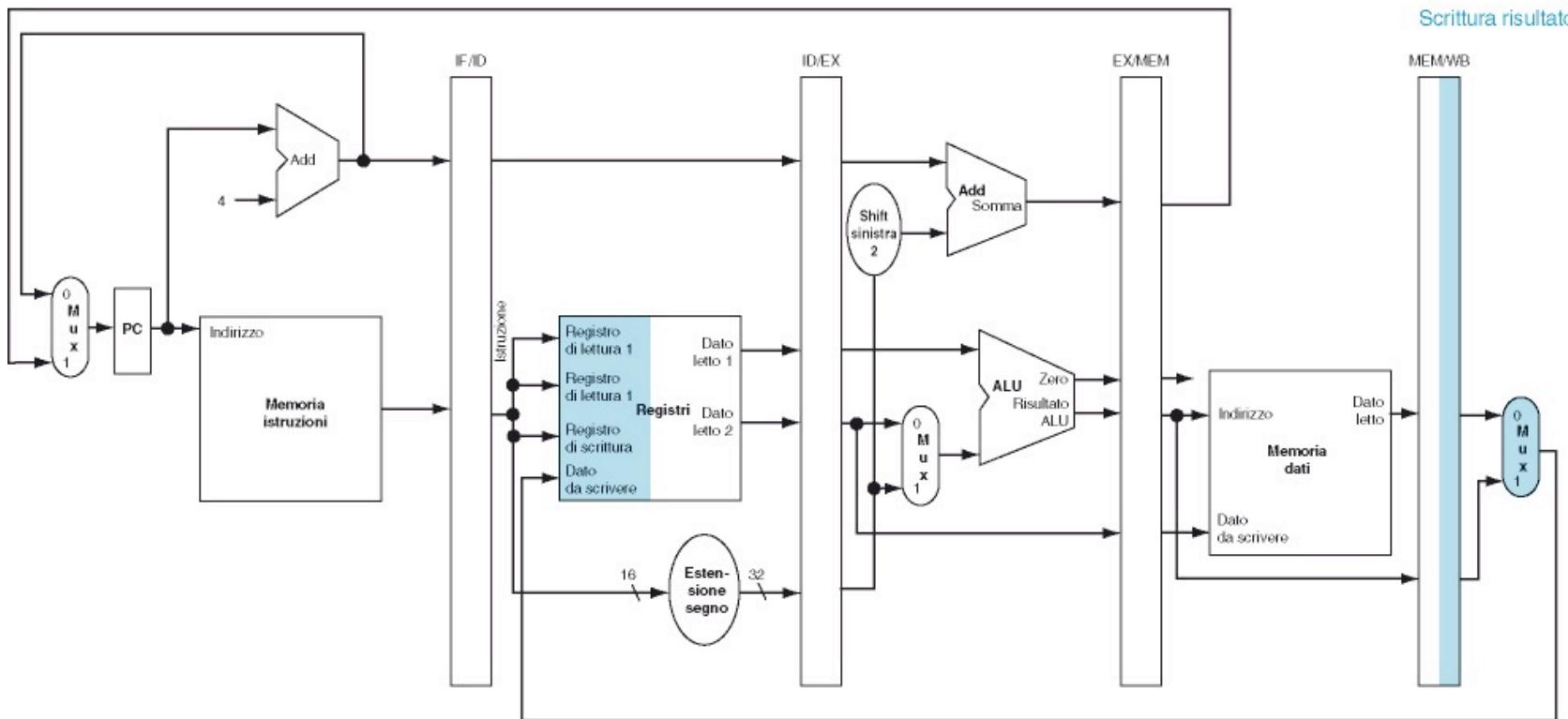
Execution



MEMory access



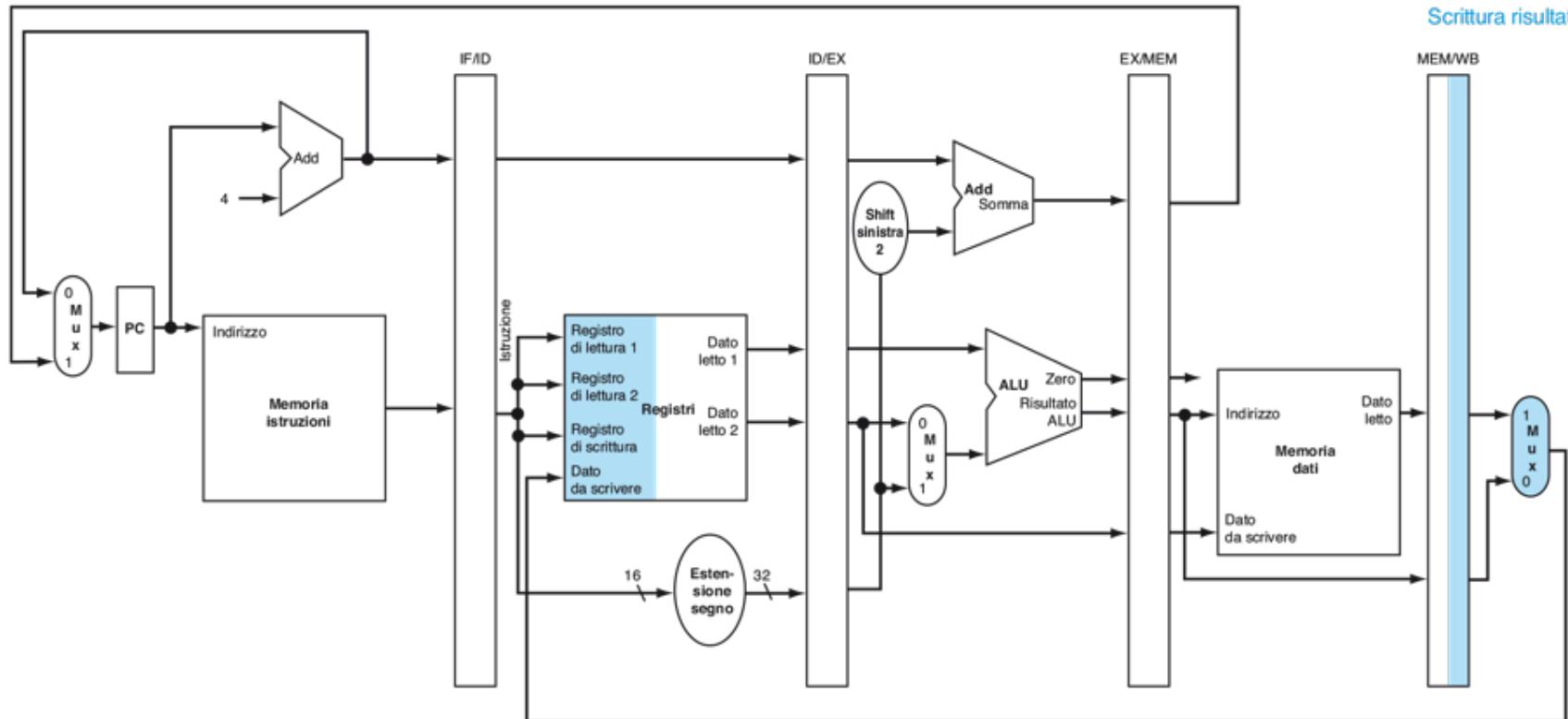
Write Back



Scrittura risultato

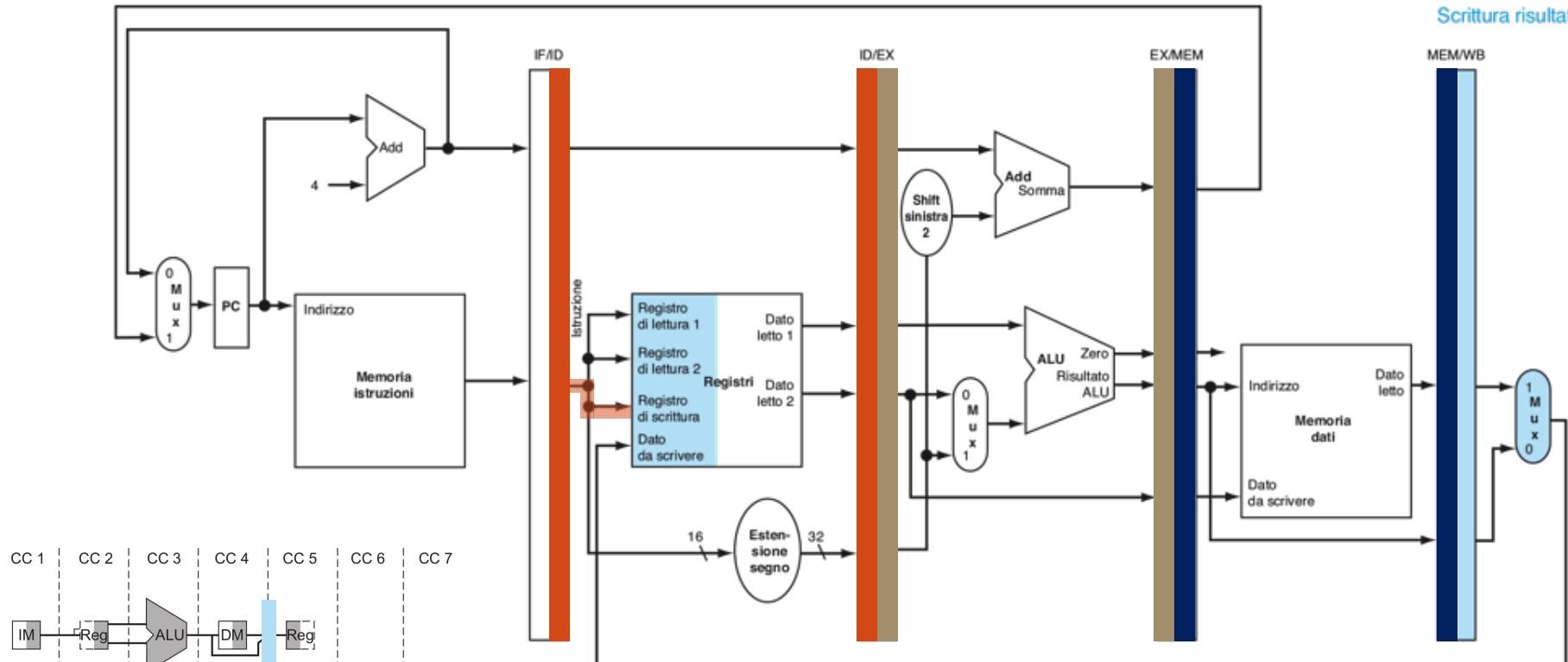
Write Back e altre istruzioni

- ① **lw** \$t4, (\$t0)
- ② **add** \$t3, \$t1, \$t2
- ③ **add** \$t5, \$t1, \$t4
- ④ **sw** \$t6, (\$t0)



Write Back e altre istruzioni

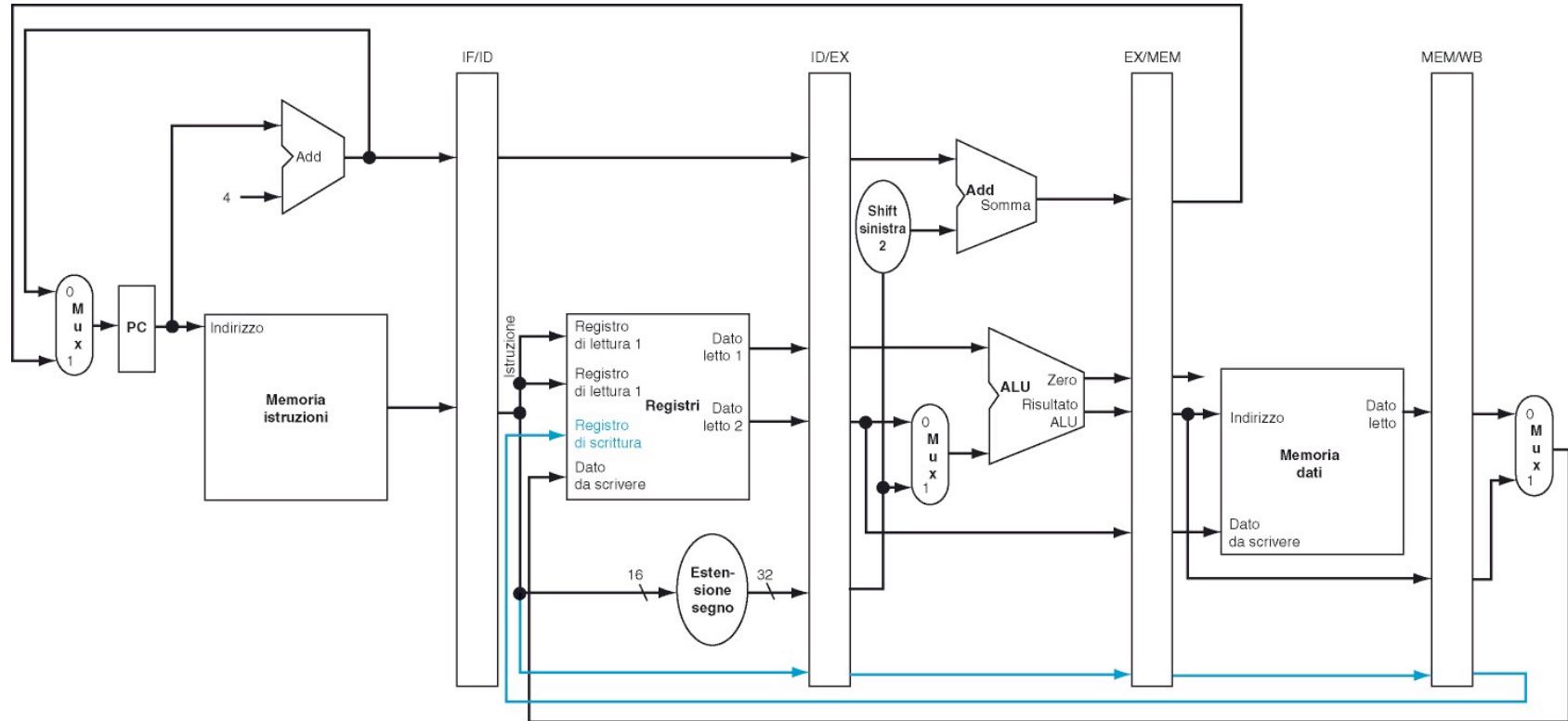
- ① **lw** \$t4, (\$t0)
- ② **add** \$t3, \$t1, \$t2
- ③ **add** \$t5, \$t1, \$t4
- ④ **sw** \$t6, (\$t0)



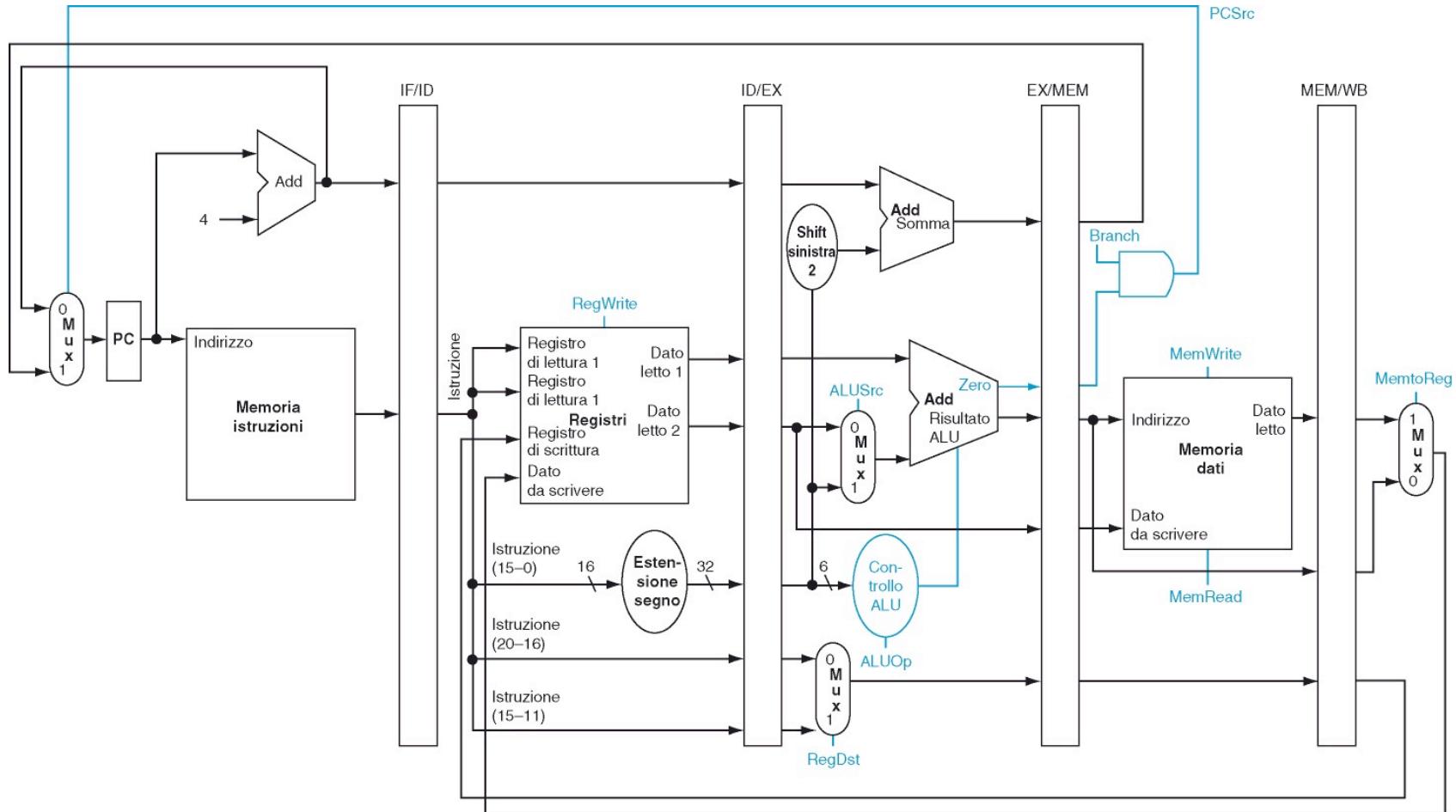
In questo modo, il write-back avviene sul registro sbagliato.
(l'indice del registro di destinazione proviene dalla
istruzione 3 passi dopo)

Write Back (corretto)

Tutte le informazioni ed i segnali di controllo
devono essere nel registro precedente della pipeline



Con la logica dei salti (beq)



I segnali di controllo della CU

Possiamo dividere i segnali di controllo delle fasi esecutive in 3 gruppi:

fase EXE

fase MEM

fase WB



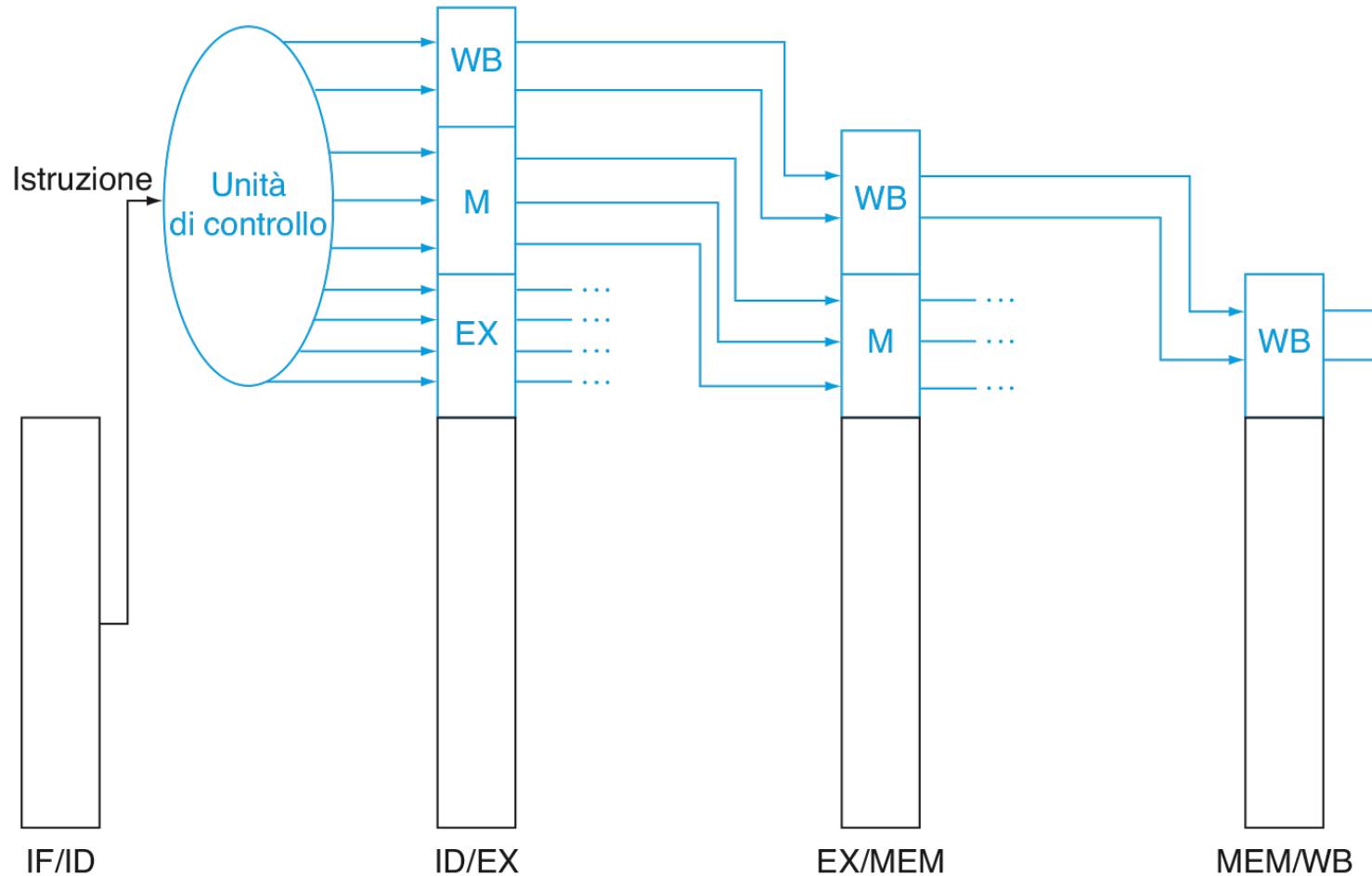
Istruzione	Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo				Segnali di controllo dello stadio di accesso alla memoria dati			Segnali di controllo dello stadio di Write-back	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch Mem	Read Mem	Write	RegWrite	Memto-Reg
Formato-R	1	1	0	0	0	0	0	1	0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

Questi segnali devono essere passati di fase in fase nei registri della pipeline

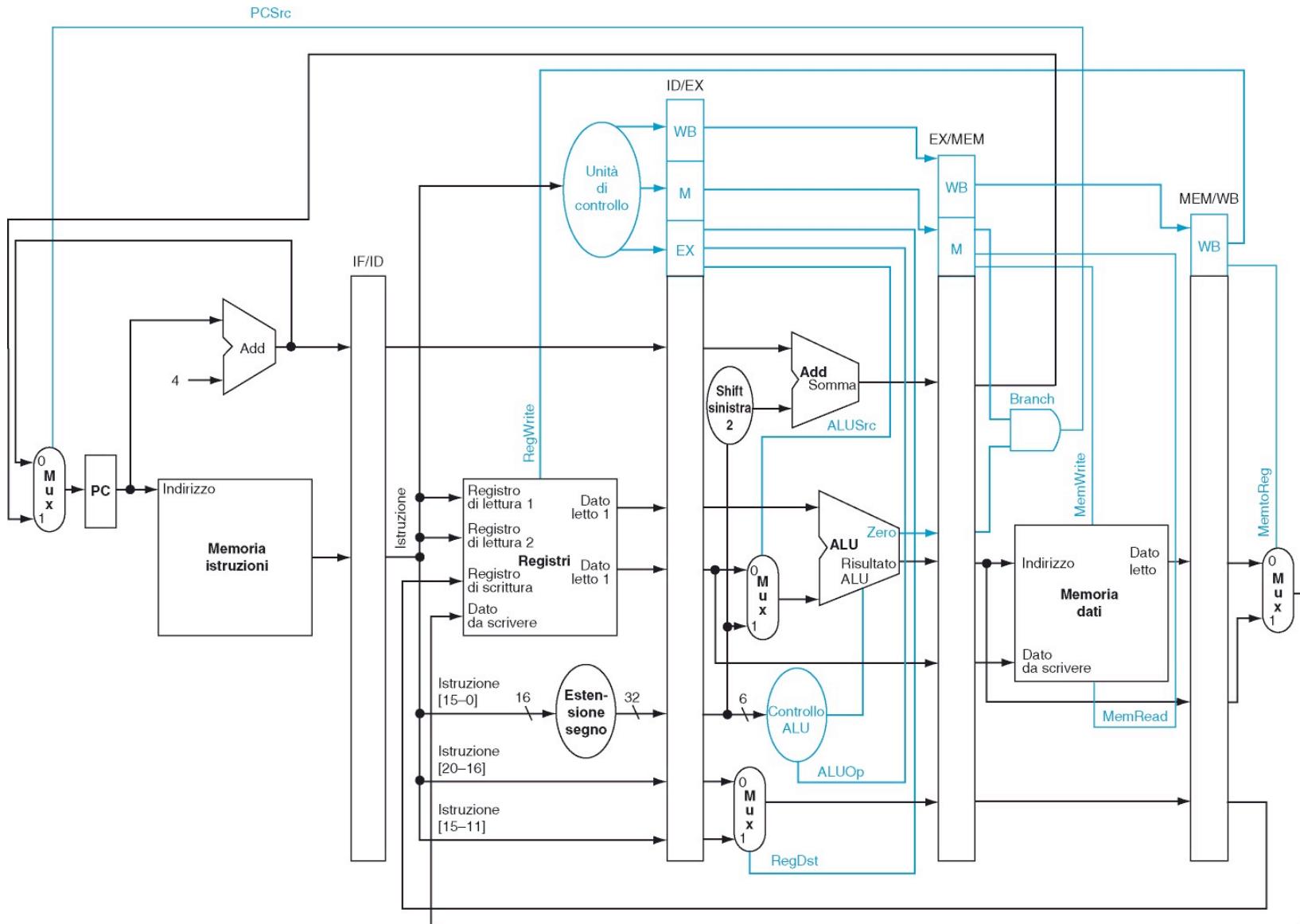
CU e registri della pipeline

Ad ogni fase salviamo i **dati** (letti dai registri o dalla parte immediata)

ed i **segnali di controllo** della stessa istruzione (generati dalla CU)



La CPU (quasi) completa



I segnali di controllo della CU

Possiamo dividere i segnali di controllo delle fasi esecutive in 3 gruppi:

fase EXE

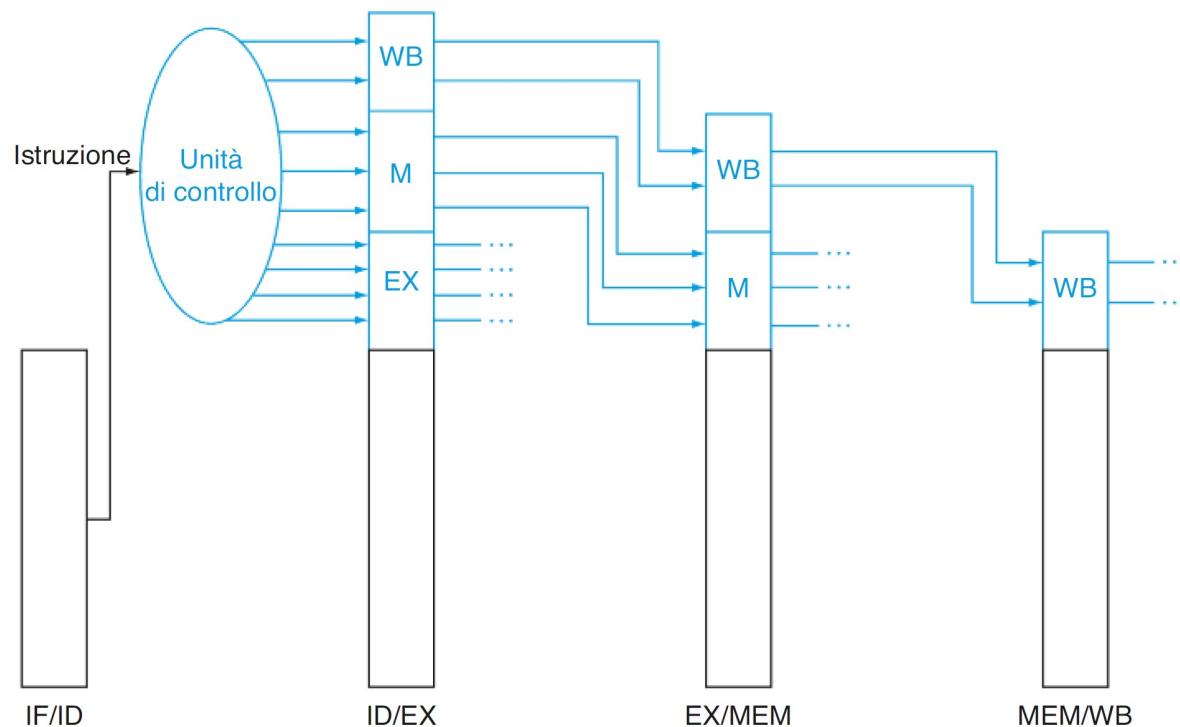
fase MEM

fase WB

Questi segnali devono essere passati di fase in fase nei registri della pipeline

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 Write-back	
	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

Propagare i segnali di controllo in pipeline



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 Write-back	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch Mem	Read Mem	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

Data hazard e forwarding



SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio per casa: controllo degli hazard

Calcolare il numero di cicli necessari ad eseguire le istruzioni seguenti:

- individuare i data e control hazard
- per determinare se con il forwarding possono essere risolti, tracciare il diagramma temporale della pipeline
- determinare quali non possono essere risolti e necessitano di stalli (e quanti stalli)
- tenere conto del tempo necessario a caricare la pipeline

Assumere

- che la beq salti alla fine della fase EXE,
- che il salto beq non sia ritardato,
- che j non introduca stalli.

Sommo un vettore di word

SommaVettore:

li \$t0, 0 # somma

li \$t1, 40 # fine

li \$t2, 0 # offset

Ciclo: beq \$t2, \$t1, **Fine**

lw \$t3, vettore(\$t2)

add \$t0, \$t0, \$t3

addi \$t2, \$t2, 4

j Ciclo

Fine: li \$v0, 10

syscall



Esercizio per casa: cosa sarebbe successo senza forwarding?

Esercizio per casa: controllo degli hazard (soluzione)

0 SommaVettore:

1	li \$t0, 0 # somma	# fine istruzione al 5° ciclo di clock	
2	li \$t1, 40 # fine	# IF ID EX MM WB 6°	
3	li \$t2, 0 # offset	# IF ID EX MM WB 7°	
4	ciclo: beq \$t2, \$t1, Fine	# -> -> IF ID EX MM WB 10° 18° 26° ... 90°	
5	lw \$t3, vettore(\$t2)	# IF ID EX MM WB 11° 19° 27°	
6	add \$t0, \$t0, \$t3	# -> -> IF ID EX MM WB 14° 22° ...	
7	addi \$t2, \$t2, 4	# IF ID EX MM WB 15° 23° ...	
8	j ciclo	# IF ID EX MM WB 16° 24° ...	
9	Fine: li \$v0, 10	# beq: -> IF ID EX MM WB 93°	
10	syscall	#	94°

Per cui:

- occorrono **2 stalli** tra **li** e **beq** (istruzioni i3 e i4, rispettivamente)
- il ciclo impiega 5 colpi di clock più **2 stalli** tra **lw** (i5) e **add** (i6) e **1 stall** tra **addi** (i7) e **beq** (i4)
- il control-hazard su **beq** (i4) alla fine del ciclo inserisce **2 stalli** (agisce a fine EXE)

Totale: 4[riempimento pipel.] + 3 + **2** + 10*(5+**3**) + 1(uscita ciclo) + 2 + **2** = **94** colpi di clock

Data hazard senza forwarding

- ① **sub** \$2,\$1,\$3 # Registro \$2 scritto da ①
- ② **and** \$12,\$2,\$5 # Primo operando dipende da ①
- ③ **or** \$13,\$6,\$2 # Secondo operando dipende da ①
- ④ **add** \$14,\$2,\$2 # Primo e secondo operando dipendono da ①
- ⑤ **sw** \$15,100(\$2) # Base address dipende da ①

Hazard:

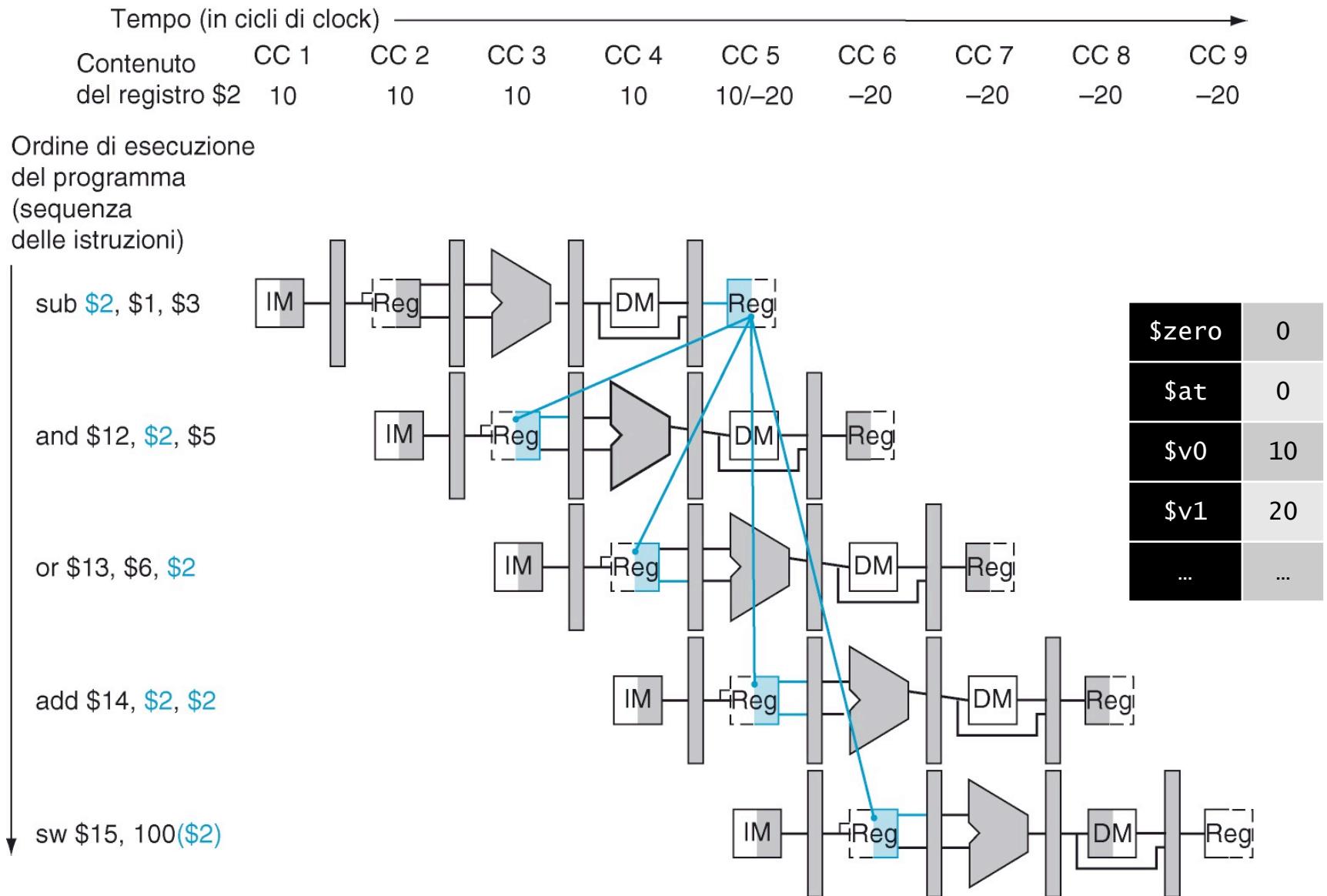
- ① **sub** \$2,\$1,\$3 # IF ID EX MM WB
- ② **and** \$12,\$2,\$5 # IF ID EX MM WB
- ③ **or** \$13,\$6,\$2 # IF ID EX MM WB
- ④ **add** \$14,\$2,\$2 # IF ID EX MM WB np: wb e ID
nello stesso ciclo
- ⑤ **sw** \$15,100(\$2) # IF ID EX MM WB

\$zero	0
\$at	0
\$v0	10
\$v1	20
...	...

Con due fasi di stallo in ②:

- ① **sub** \$2,\$1,\$3 # IF ID EX MM WB
- ② **and** \$12,\$2,\$5 # -> -> IF ID EX MM WB
- ③ **or** \$13,\$6,\$2 # IF ID EX MM WB
- ④ **add** \$14,\$2,\$2 # IF ID EX MM WB
- ⑤ **sw** \$15,100(\$2) # IF ID EX MM WB

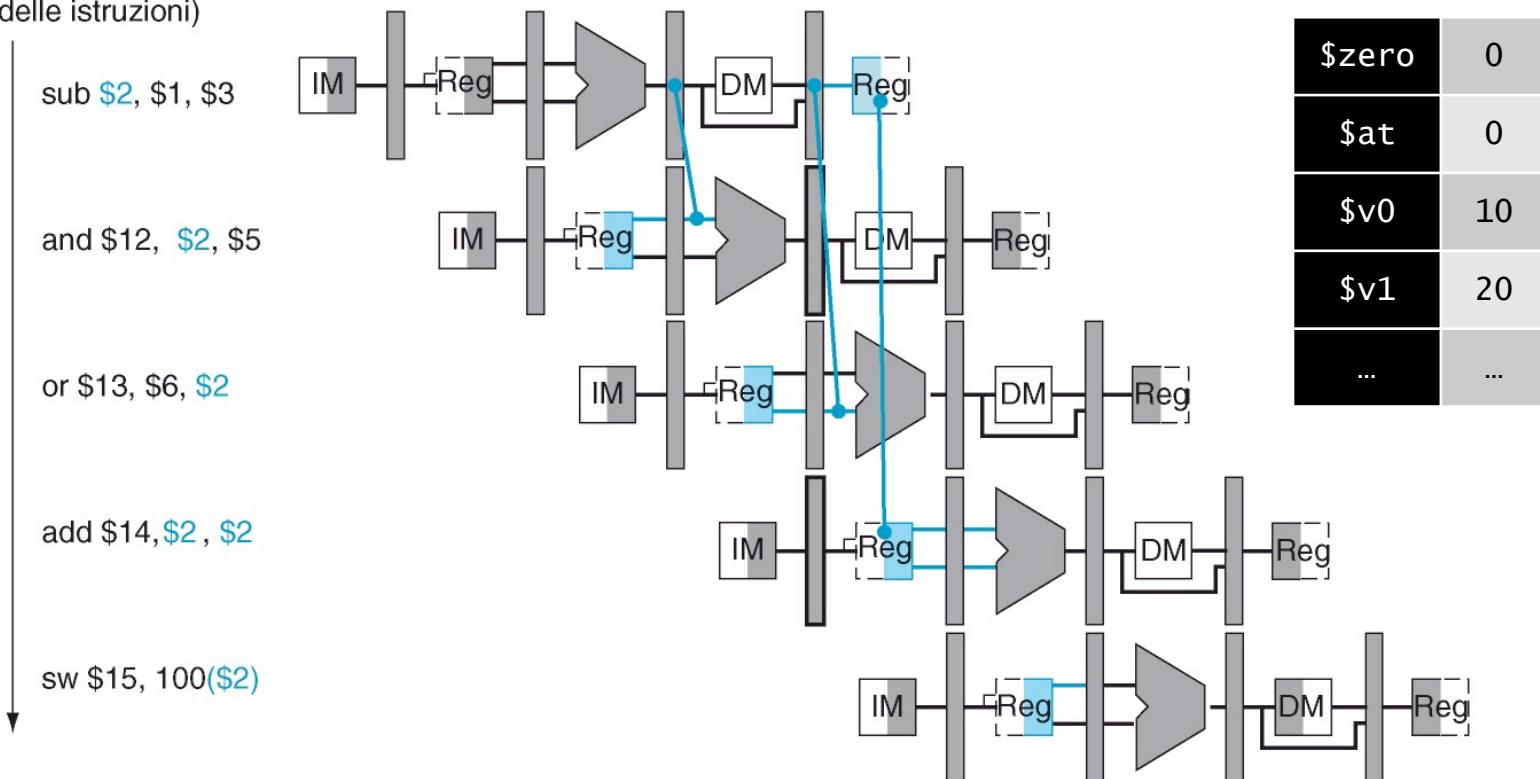
Data hazard senza forwarding



Data hazard con forwarding

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Contenuto del registro \$2:	10	10	10	10	10/-20	-20	-20	-20	-20
Contenuto del registro EX/MEM:	X	X	X	-20	X	X	X	X	X
Contenuto del registro MEM/WB:	X	X	X	X	-20	X	X	X	X

Ordine di esecuzione
del programma
(sequenza
delle istruzioni)



non basta trasferire i registri nella pipeline: ad esempio la ALU deve eseguire l'istruzione definita due fasi prima

Scoprire un data-hazard in EXE

Legenda: Segnali dei registri di pipeline: <reg.pipeline>.<segnalet>

In rosso i controlli aggiuntivi non menzionati sul libro di testo

1.a) ID/EX . rs == EX/MEM . rd	&& EX/MEM.RegWrite == 1	&& EX/MEM . rd != 0
	&& EX/MEM.MemRead == 0	
1.b) ID/EX . rt == EX/MEM . rd	&& EX/MEM.RegWrite == 1	&& EX/MEM . rd != 0
	&& EX/MEM.MemRead == 0	
2.a) ID/EX . rs == MEM/WB . rd	&& MEM/WB.RegWrite == 1	&& MEM/WB . rd != 0
	&& MEM/WB.MemRead == 0	
2.b) ID/EX . rt == MEM/WB . rd	&& MEM/WB.RegWrite == 1	&& MEM/WB . rd != 0
	&& MEM/WB.MemRead == 0	

- ① sub \$2,\$1,\$3 # IF | ID | EX | MM | WB
- ② and \$12,\$2,\$5 # IF | ID | EX | MM | WB (1.a)
- ③ or \$13,\$6,\$2 # IF | ID | EX | MM | WB (2.b)
- ④ add \$14,\$2,\$2 # IF | ID | EX | MM | WB
- ⑤ sw \$15,100(\$2) # IF | ID | EX | MM | WB

Se è presente un **data-hazard** su un argomento rispetto a **entrambe** le istruzioni precedenti?

Ha sempre la precedenza il valore prodotto per ultimo, ossia il contenuto di EX/MEM della pipeline (dell'istr. subito precedente a quella corrente in EXE)

Errore rilevamento hazard con EX/MEM.MemRead==1

	#	opcode	rs	rt	[Immediate]
lw \$8, 0x1800(\$10)	#	100011	01010	01000	0001	1000	0000 0000
	#	RegWrite==1, MemRead==1					
	#	opcode	rs	rt	rd	shamt	funct
add \$4, \$3, \$5	#	000000	00011	00101	00100	00000	100000
	#	RegWrite==1, MemRead==0					

① **lw** \$8, 0x1800(\$10) # IF | ID | EX | MM | WB

② **add** \$4, \$3, \$5 # IF | ID | EX | MM | WB

Non dovrebbe essere rilevato un hazard...

... eppure...

Errore rilevamento hazard con EX/MEM.MemRead==1

	#	opcode	rs	rt	[Immediate]
lw \$8, 0x1800(\$10)	#	100011	01010	01000	0001 1	000 0000 0000	
	#					Regwrite==1, MemRead==1	
	#	opcode	rs	rt	rd	shamt	funct
add \$4, \$3, \$5	#	000000	00011	00101	00100	00000	100000
	#					Regwrite==1, MemRead==0	

① lw \$8, 0x1800(\$10) # IF | ID | EX | MM | WB

② add \$4, \$3, \$5 # IF | ID | EX | MM | WB

1.a) ID/EX . rs == EX/MEM . rd && EX/MEM.RegWrite == 1 && EX/MEM . rd != 0

... eppure...

Scoprire un data-hazard in EXE

Legenda: Segnali dei registri di pipeline: <reg.pipeline>.<segnalet>

In rosso i controlli aggiuntivi non menzionati sul libro di testo

1.a) ID/EX . rs == EX/MEM . rd	&& EX/MEM.RegWrite == 1	&& EX/MEM . rd != 0
	&& EX/MEM.MemRead == 0	
1.b) ID/EX . rt == EX/MEM . rd	&& EX/MEM.RegWrite == 1	&& EX/MEM . rd != 0
	&& EX/MEM.MemRead == 0	
2.a) ID/EX . rs == MEM/WB . rd	&& MEM/WB.RegWrite == 1	&& MEM/WB . rd != 0
	&& MEM/WB.MemRead == 0	
2.b) ID/EX . rt == MEM/WB . rd	&& MEM/WB.RegWrite == 1	&& MEM/WB . rd != 0
	&& MEM/WB.MemRead == 0	

- ① sub \$2,\$1,\$3 # IF | ID | EX | MM | WB
- ② and \$2,\$2,\$5 # IF | ID | EX | MM | WB (1.a)
- ③ or \$13,\$6,\$2 # IF | ID | EX | MM | WB (1.b)
- ④ add \$14,\$2,\$2 # IF | ID | EX | MM | WB (2.a,b)
- ⑤ sw \$15,100(\$2) # IF | ID | EX | MM | WB

Se è presente un **data-hazard** su un argomento rispetto a **entrambe** le istruzioni precedenti?

Ha sempre la precedenza il valore prodotto per ultimo, ossia il contenuto di EX/MEM della pipeline (dell'istr. subito precedente a quella corrente in EXE)

Precedenza fra i data-hazard in EXE

Legenda: Segnali dei registri di pipeline: <reg.pipeline>.<segna>

- ① **add \$1,\$1,\$2** # IF | ID | EX | MM | WB
- ② **add \$1,\$1,\$3** # IF | ID | EX | MM | WB
- ③ **add \$1,\$1,\$4** # IF | ID | EX | MM | WB

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegistroRd ≠ 0)  
and not(EX/MEM.RegWrite and (EX/MEM.RegistroRd ≠ 0)  
       and (EX/MEM.RegistroRd ≠ ID/EX.RegistroRt)  
and (MEM/WB.RegistroRd = ID/EX.RegistroRt)) PropagaB = 01
```

Propaga MEM/WB solo se

- 1) necessario (**ID/EX . rt == MEM/WB . rd**) e
- 2) EX/MEM non è attivo (v. righe in blu)

Se è presente un **data-hazard** su un argomento rispetto a **entrambe** le istruzioni precedenti?

Ha sempre la precedenza il valore prodotto per ultimo, ossia il contenuto di EX/MEM della pipeline (dell'istr. subito precedente a quella corrente in EXE)

Perché controllare rd != 0

- Vedremo che ci sono speciali operazioni (nop) in cui si scrive (fittiziamente) su \$zero
- Questo è un caso speciale che non è un vero data hazard!

(nop =) sll \$zero, \$zero, 0

Realizzare il forwarding in EXE

Forwarding: sostituire il valore letto dal blocco registri con quello prodotto dall'istruzione precedente in fase EXE (o dalla 2° istruzione precedente in fase MEM)

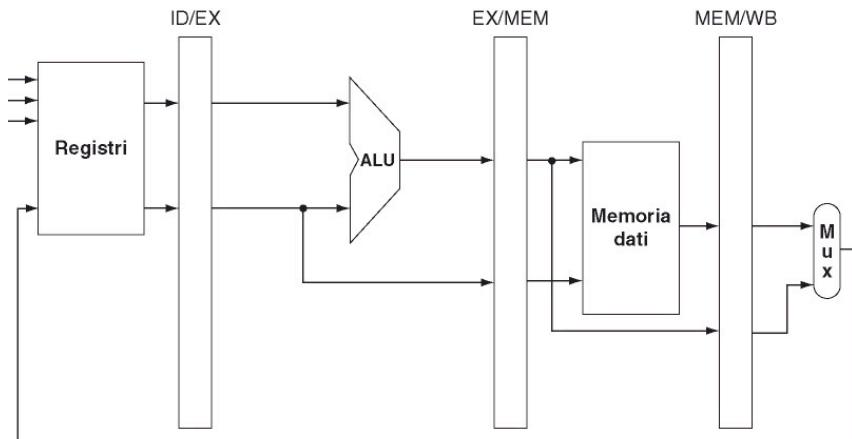
Modifiche al datapath: inserire un **MUX** prima della ALU per selezionare tra i 3 casi:

- **Non c'è forwarding:** il valore per la ALU viene dal registro **ID/EX** della pipeline
- **Forwarding dall'istr. prec.:** il valore per la ALU viene dal registro **EX/MEM** della p.
- **Forwarding dalla 2° istr. prec.:** il valore per la ALU viene dal registro **MEM/WB** della p.

Questo vale sia per il primo argomento che per il secondo argomento della ALU (2 MUX)

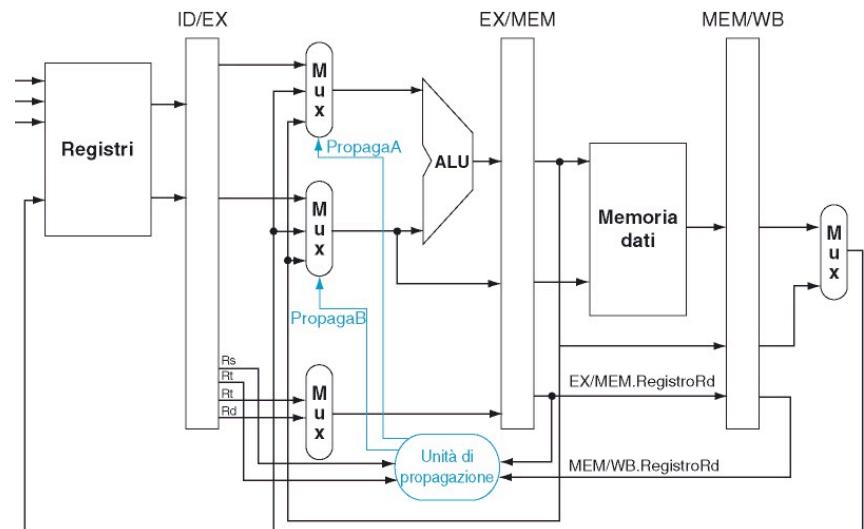
Unità di forwarding

Senza forwarding
(schema semplificato)



a. Nessuna propagazione

Con forwarding
(schema semplificato)



b. Con propagazione

Segnali per il forwarding

Controllo multiplexer	Sorgente	Spiegazione
PropagaA = 00	ID/EX	Il primo operando della ALU proviene dal register file.
PropagaA = 10	EX/MEM	Il primo operando della ALU viene propagato dal risultato della ALU nel ciclo di clock precedente.
PropagaA = 01	MEM/WB	Il primo operando della ALU viene propagato dalla memoria dati o da un precedente risultato della ALU.
PropagaB = 00	ID/EX	Il secondo operando della ALU proviene dal register file.
PropagaB = 10	EX/MEM	Il secondo operando della ALU viene propagato dal risultato della ALU nel ciclo di clock precedente.
PropagaB = 01	MEM/WB	Il secondo operando della ALU è propagato dalla memoria dati o da un precedente risultato della ALU.

Realizzare il forwarding in EXE

Forwarding: sostituire il valore letto dal blocco registri con quello prodotto dall'istruzione precedente in fase EXE (o dalla 2° istruzione precedente in fase MEM)

Modifiche al datapath: inserire un **MUX** prima della ALU per selezionare tra i 3 casi:

- **Non c'è forwarding:** il valore per la ALU viene dal registro **ID/EX** della pipeline
- **Forwarding dall'istr. prec.:** il valore per la ALU viene dal registro **EX/MEM** della p.
- **Forwarding dalla 2° istr. prec.:** il valore per la ALU viene dal registro **MEM/WB** della p.

Questo vale sia per il primo argomento che per il secondo argomento della ALU (2 MUX)

NOTA: è possibile realizzare il forwarding anche:

- **nella fase ID** (necessario SOLO se la **beq** viene anticipata in ID)
- **nella fase MEM** (necessario SOLO se **lw \$rt, ...** è subito seguita da **sw \$rt, ...**)
(se sw è preceduta da tipo R il forwarding avviene in fase EXE)

Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato **lw** e poi **sw** in questo esatto ordine?

Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato **lw** e poi **sw** in questo esatto ordine?

Quando vogliamo spostare un valore **dalla** memoria e posizionarlo in un **altro valore della memoria**. Una sorta di swap di valori in memoria.

Mem[offset+\$t3] ← Mem[offset+\$t1]

lw \$t0, offset(\$t1)

sw \$t0, offset(\$t3)

Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato **lw** e poi **sw** in questo esatto ordine?

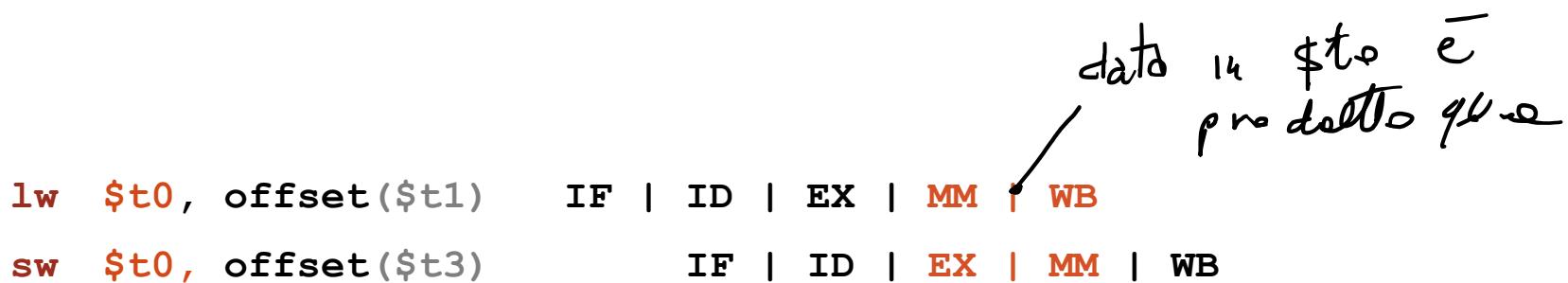
Quando vogliamo spostare un valore **dalla** memoria e posizionarlo in un **altro valore della memoria**. Una sorta di swap di valori in memoria.

```
lw $t0, offset($t1)      IF | ID | EX | MM | WB  
sw $t0, offset($t3)      IF | ID | EX | MM | WB
```

Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato **lw** e poi **sw** in questo esatto ordine?

Quando vogliamo spostare un valore **dalla** memoria e posizionarlo in un **altro valore della memoria**. Una sorta di swap di valori in memoria.



Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato **lw** e poi **sw** in questo esatto ordine?

Quando vogliamo spostare un valore **dalla** memoria e posizionarlo in un **altro valore della memoria**. Una sorta di swap di valori in memoria.

lw \$t0, offset(\$t1)
sw \$t0, offset(\$t3)

IF | ID | EX | MM | WB
IF | ID | EX | MM | WB

data 14 \$t0 è
prodotto quale
sw lo scrive
quala in
memoria

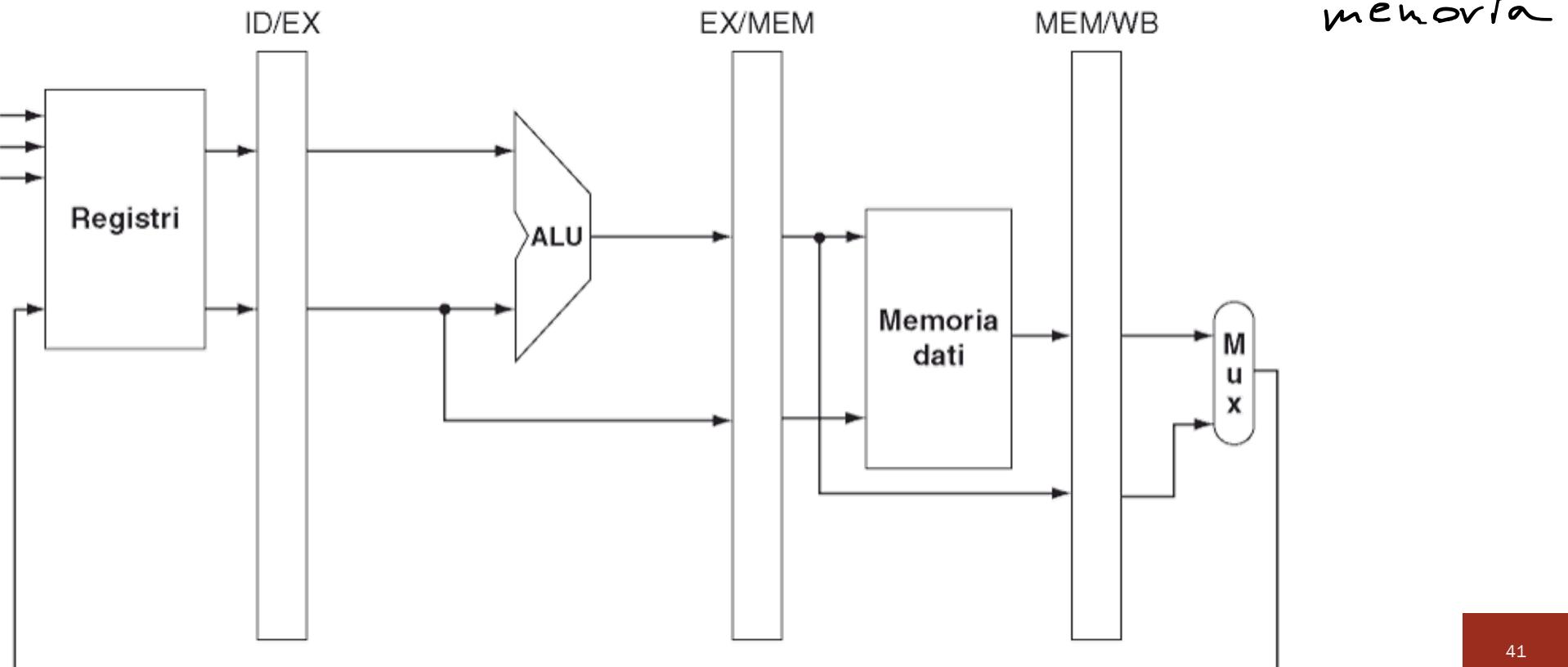
Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato **lw** e poi **sw** in questo esatto ordine? *data 14 \$t0 è
prodotto qd-e*

lw \$t0, offset(\$t1) IF | ID | EX | **MM** | WB

sw \$t0, offset(\$t3) IF | ID | EX | **MM** | WB

*sw lo scrive
qua in memoria*

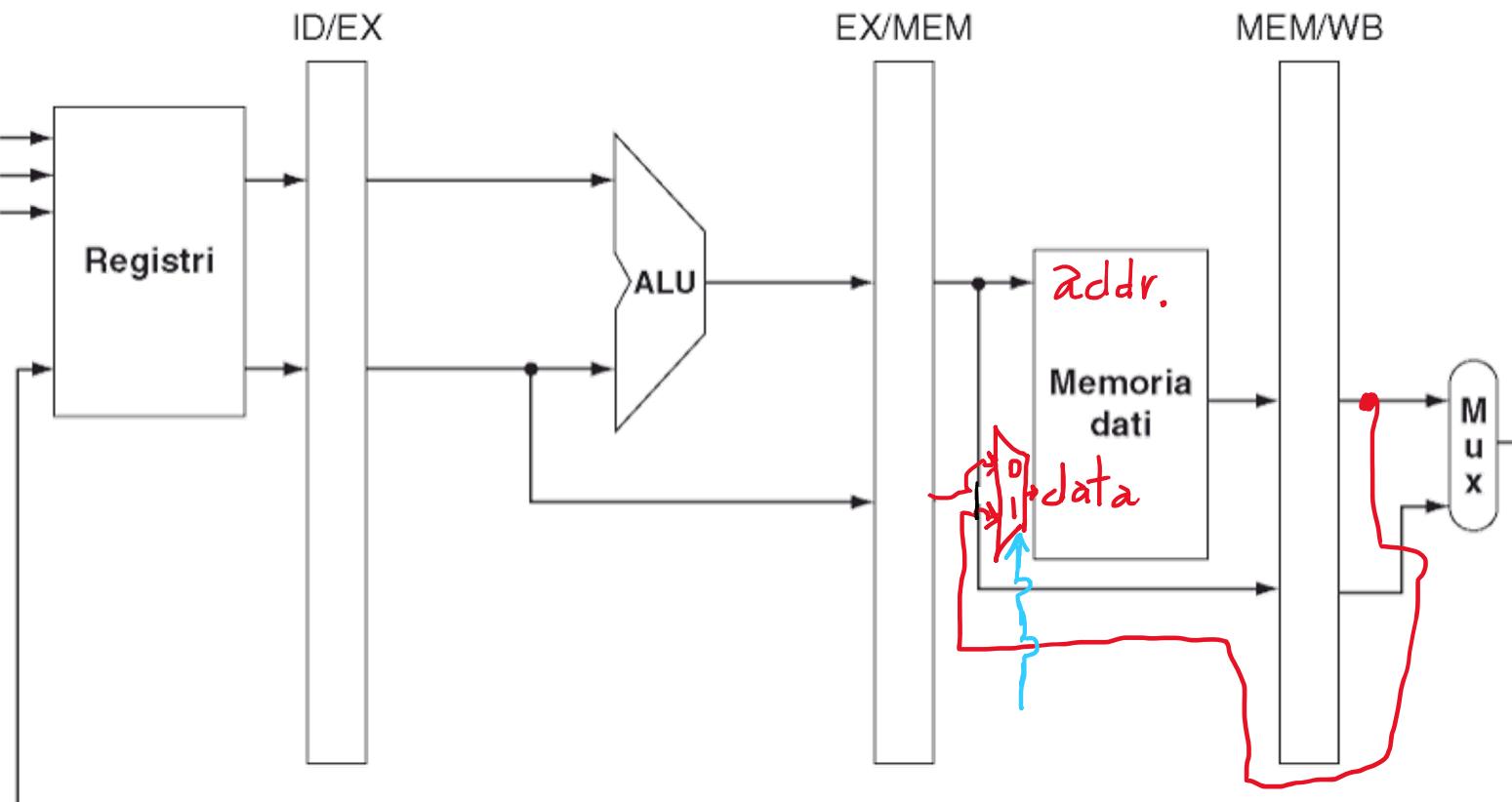


Realizzare il forwarding in MEM (lw/sw)

Quando è che viene usato lw e poi sw in questo esatto ordine? data 14 \$t0 è
prodotto quale

lw \$t0, offset(\$t1) IF | ID | EX | MM | WB
sw \$t0, offset(\$t3) IF | ID | EX | MM | WB

sw lo scrive
qua in memoria



Come rilevare il forwarding?

Un possibile modo di rilevazione:

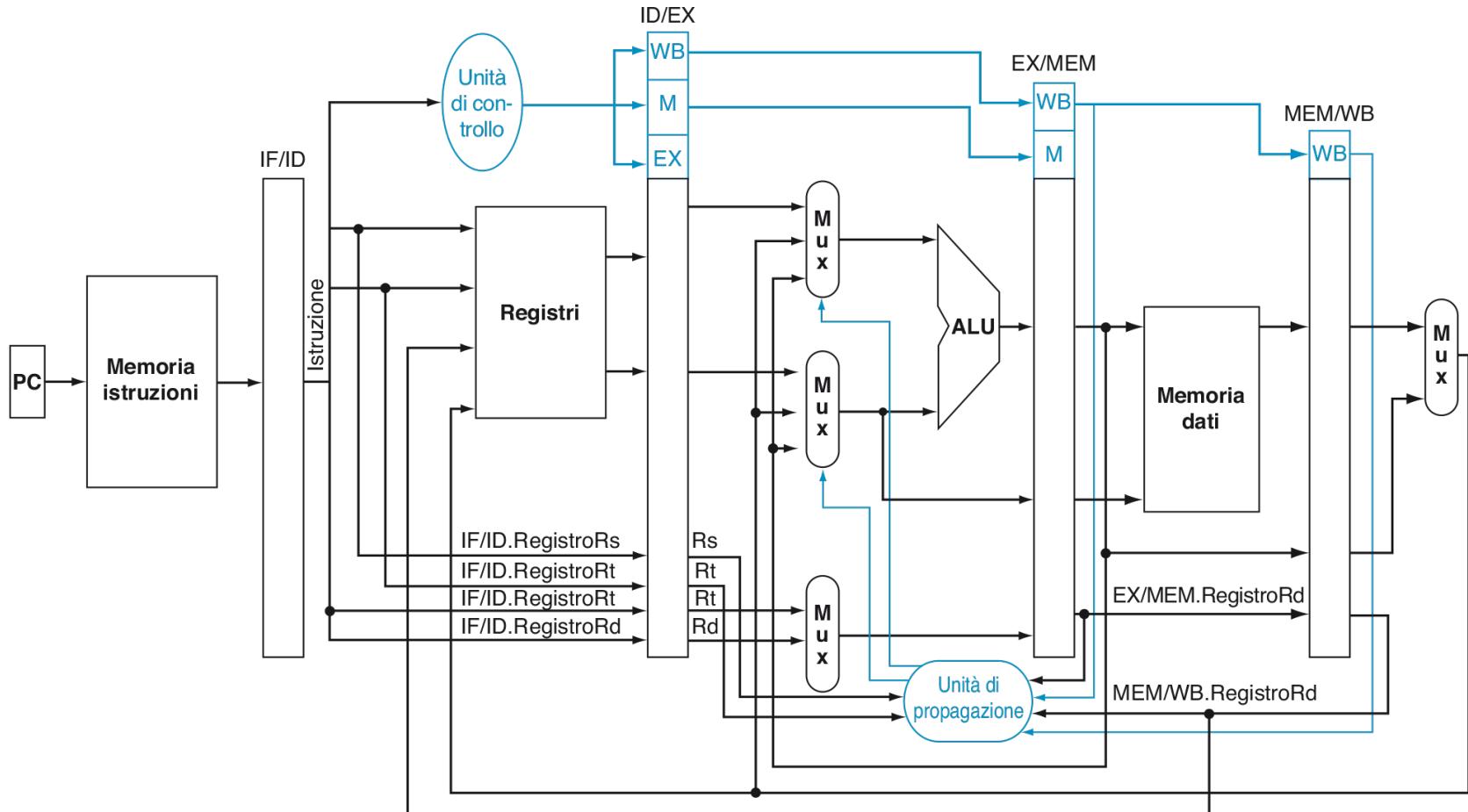
```
MEM/WB.MemToReg == 1      &&  MEM/WB.RegWrite == 1  && EX/MEM.MemWrite == 1  
&& MEM/WB . Rt == EX/MEM . Rt
```

Razionale:

1. La `lw` deve scrivere in WB nel register file dalla memoria
2. Nello stadio a monte (istruzione successiva del codice) occorre scrivere in memoria (`sw`)
3. I registri sono uguali ($\text{MEM/WB} . \text{Rt} == \text{EX/MEM} . \text{Rt}$)

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 Write-back	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch Mem	Read Mem	Mem-Write	RegWrite	Memto-Reg
Formato-R	1	1	0	0	0	0	0	1	0
<code>lw</code>	0	0	0	1	0	1	0	1	1
<code>sw</code>	X	0	0	1	0	0	1	0	X
<code>beq</code>	X	0	1	0	1	0	0	0	X

CPU con forwarding per data hazard (senza stallo)



Lo stallo dell'istruzione

Talvolta l'istruzione deve attendere che sia pronto il dato perché possa avvenire il forwarding

- ① **lw \$s1, vettore(\$s2)** # IF ID EX **MM** WB
- ② **addi \$s1, \$s1, 42** # \rightarrow IF ID EX MM WB
- ③ **addi \$s1, \$s1, 42** # (\rightarrow) IF ID **EX** MM WB

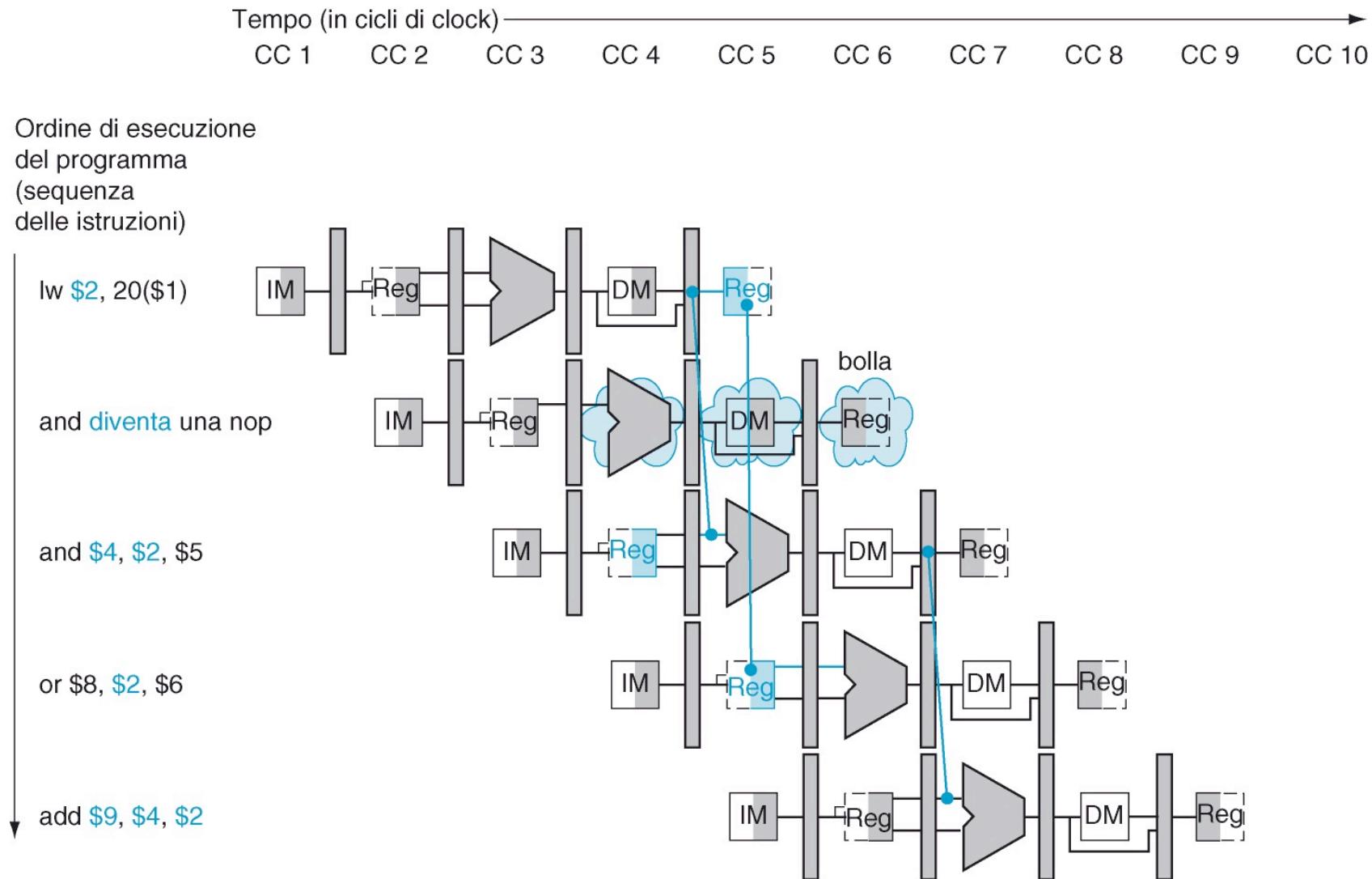
Per riconoscere di dover inserire uno stallo dobbiamo esaminare le due istruzioni:

- la seconda ne ha bisogno in EXE e la prima produrrà il dato solo nella fase MEM (**lw**)
 $ID/EX.MemRead == 1 \ \&\& (IF/ID . rs == ID/EX . rt \ | \ | IF/ID . rt == ID/EX . rt)$
- la seconda lo richiede in ID e la prima lo produrrà solo dopo EXE (solo **beq** anticipato)
 $ID/EX.RegWrite == 1 \ \&\& (IF/ID . rt == ID/EX . rd \ | \ | IF/ID . rs == ID/EX . rd) \ \&\& op == beq$

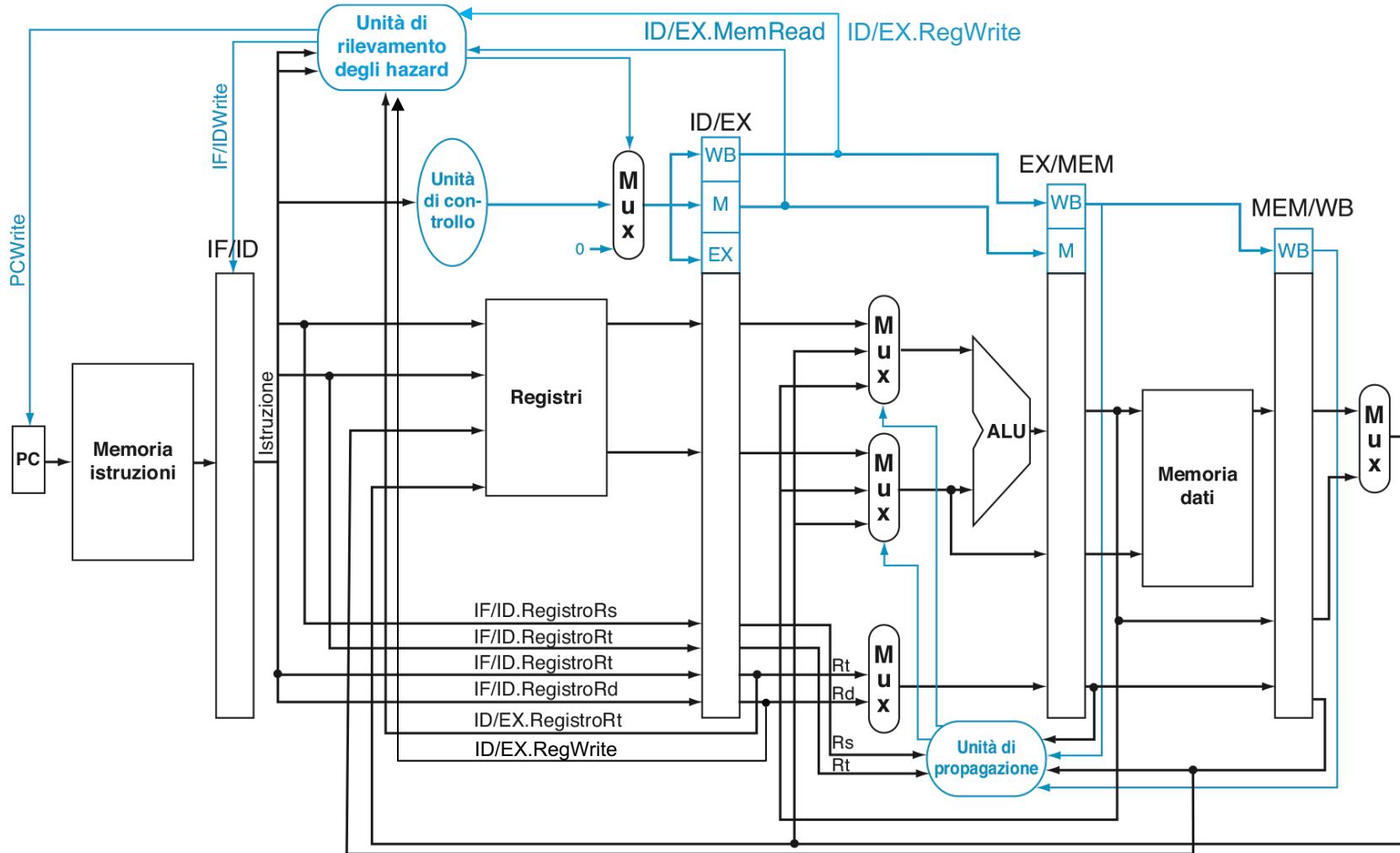
Per **fermare l'istruzione** con uno stallo, dobbiamo (nella fase ID):

- annullare l'istruzione che deve attendere (**bolla**)
 - occorre azzerare i segnali di controllo MemWrite e RegWrite nonché IF/ID.Istruzione
- **rileggere la stessa istruzione** di nuovo affinché possa essere rieseguita un CC dopo
 - occorre impedire che il PC si aggiorni

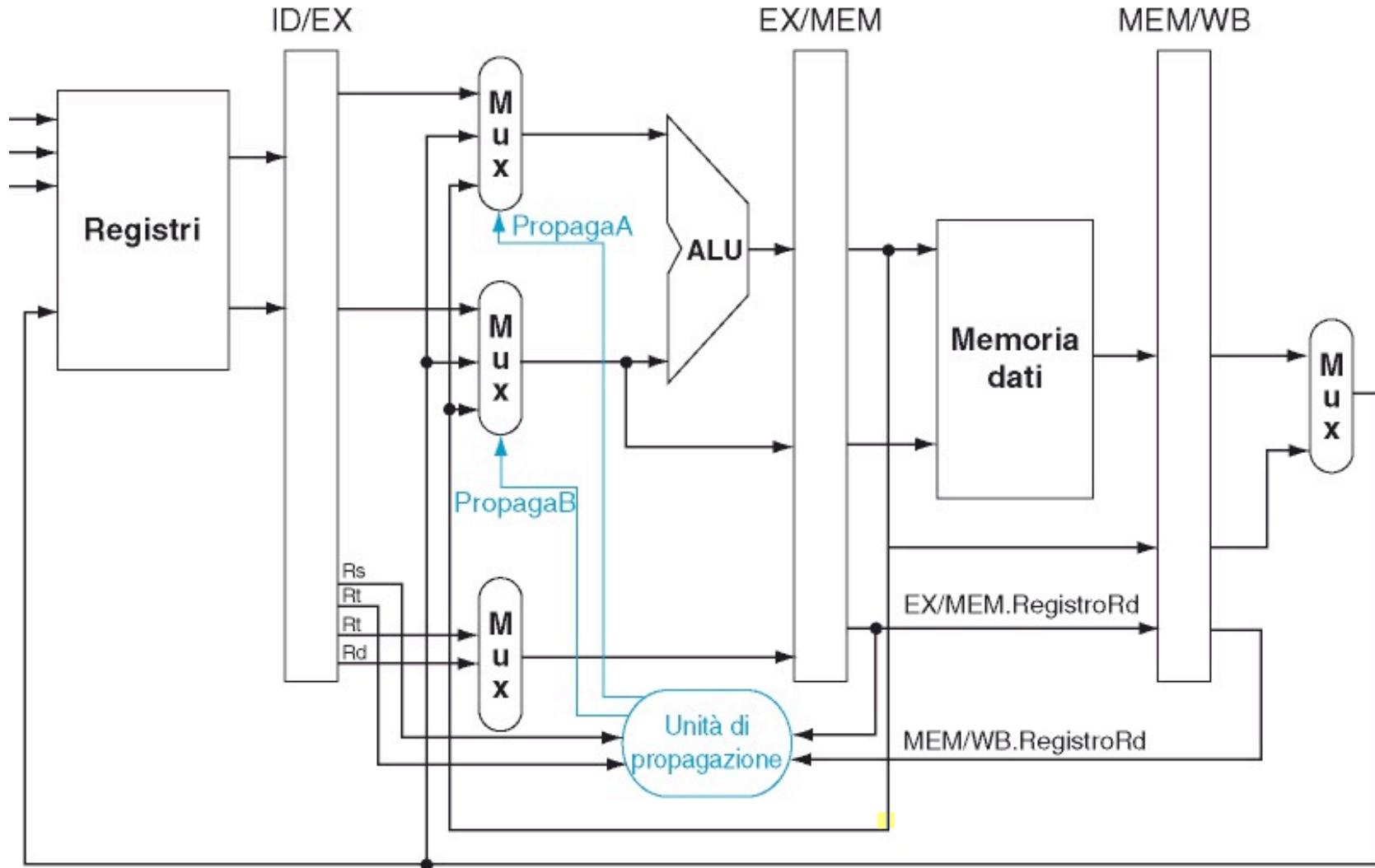
Stallo in azione



Architettura (quasi) completa



Con forwarding (schema semplificato)



b. Con propagazione

Esercizio per casa: controllo degli hazard

Calcolare il numero di cicli necessari ad eseguire le istruzioni seguenti:

- individuare i data e control hazard
- per determinare se con il forwarding possono essere risolti, tracciare il diagramma temporale della pipeline
- determinare quali non possono essere risolti e necessitano di stalli (e quanti stalli)
- tenere conto del tempo necessario a caricare la pipeline

Assumere

- che la beq salti alla fine della fase EXE,
- che il salto beq non sia ritardato,
- che j non introduca stalli.

Sommo un vettore di word

SommaVettore:

li \$t0, 0 # somma

li \$t1, 40 # fine

li \$t2, 0 # offset

Ciclo: beq \$t2, \$t1, **Fine**

lw \$t3, vettore(\$t2)

add \$t0, \$t0, \$t3

addi \$t2, \$t2, 4

j Ciclo

Fine: li \$v0, 10

syscall

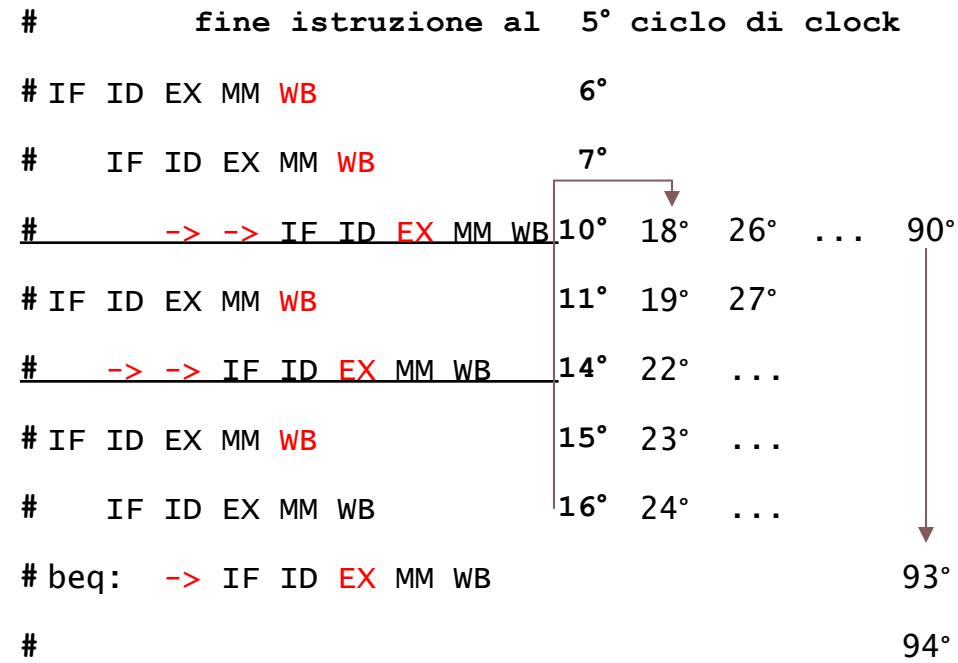


Esercizio per casa: cosa sarebbe successo senza forwarding?

Esercizio per casa: controllo degli hazard (soluzione)

0 SommaVettore:

```
1      li $t0, 0 # somma
2      li $t1, 40 # fine
3      li $t2, 0 # offset
4 ciclo: beq $t2, $t1, Fine
5      lw $t3, vettore($t2)
6      add $t0, $t0, $t3
7      addi $t2, $t2, 4
8      j ciclo
9 Fine: li $v0, 10
10    syscall
```



Per cui:

- occorrono **2 stalli** tra li e beq (istruzioni i3 e i4, rispettivamente)
- il ciclo impiega 5 colpi di clock più **2 stalli** tra lw (i5) e add (i6) e **1 stallo** tra addi (i7) e beq (i4)
- il control-hazard su beq (i4) alla fine del ciclo inserisce **2 stalli** (agisce a fine EXE)

Totale: $4[\text{riempimento pipel.}] + 3 + 2 + 10*(5+3) + 1(\text{uscita ciclo}) + 2 + 2 = 94 \text{ colpi di clock}$

- Con forwarding si possono togliere i data hazard sulla beq e ridurre a uno il data hazard della add

Totale: $4[\text{riempimento pipel.}] + 3 + 2 + 10*(5+1) + 1(\text{uscita ciclo}) + 2 + 2 = 72 \text{ colpi di clock}$

Esercizio per casa: da vj a vjal

Si vuole aggiungere alla CPU l'istruzione vectorised jump (**vj**), di tipo I e sintassi assembly

vj \$indice, vettore

che salta all'indirizzo contenuto nell'elemento \$indice-esimo del vettore di word.

Esempio: se in memoria si è definito staticamente il

vettore: .word 15, 24, 313, 42

e nel registro **\$t0** c'è il valore 3 allora

vj \$t0, vettore

salterà all'indirizzo **42** (che è l'elemento con indice 3 del vettore)

- a) si disegnino le modifiche necessarie a realizzare la funzione, aggiungendo tutti gli eventuali MUX, segnali di controllo, bus, ALU e sommatori (ecc.) che ritenete necessari.
- b) indicate i valori di tutti i segnali di controllo, in modo da eseguire l'istruzione **vj**.
- c) tenendo conto che il tempo di accesso ai registri (sia in lettura che scrittura) è di **25ns**, l'accesso alla memoria impiega **75ns**, la ALU e i sommatori impiegano **100ns** e ignorando gli altri ritardi, calcolate il tempo di esecuzione minimo della istruzione **vj** e indicate se è necessario aumentare il periodo di clock della CPU per poter svolgere questa nuova istruzione.

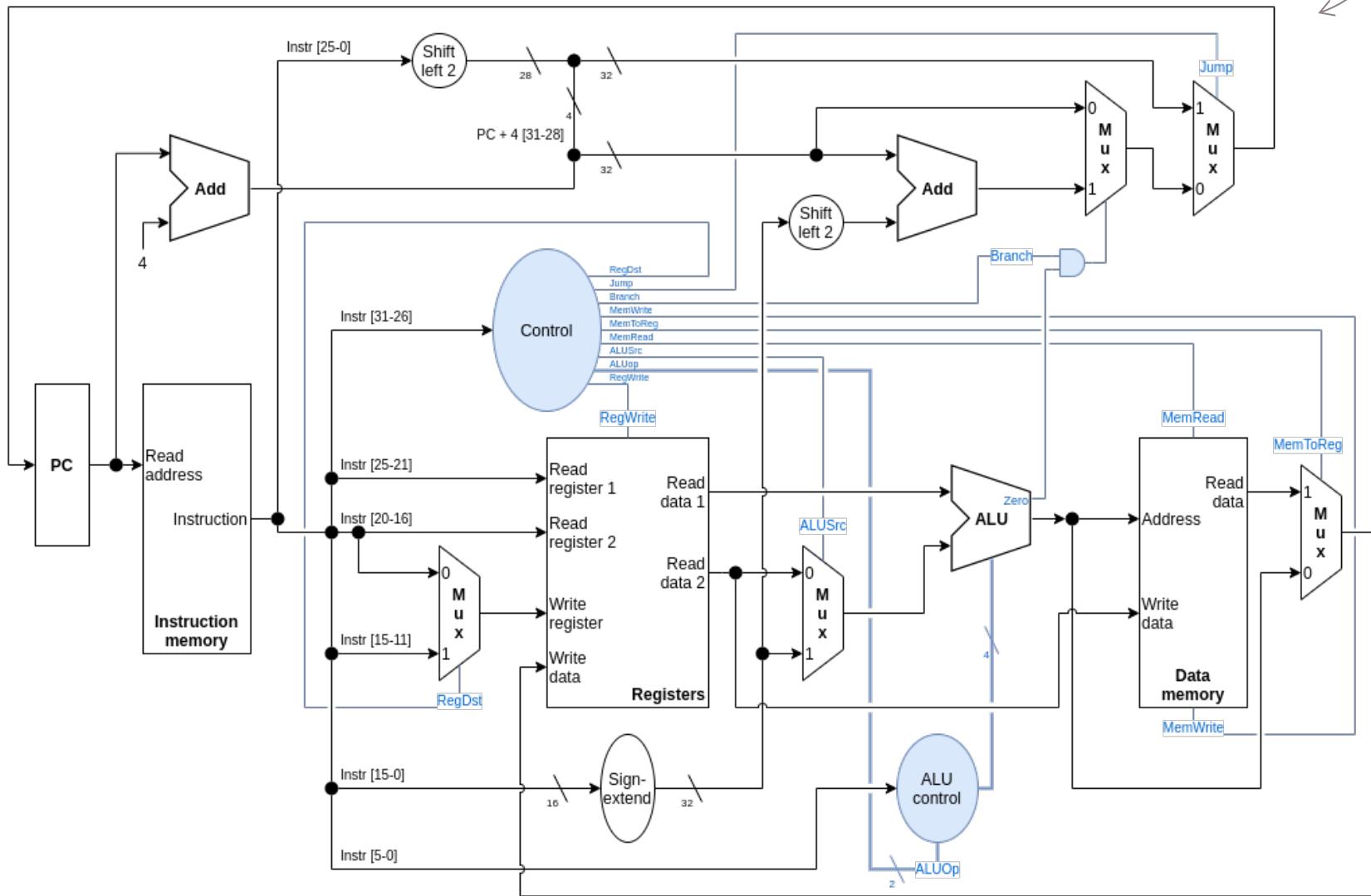


Esercizio per casa: modificare **vj** in **vjal** (salvando il **\$pc** corrente in **\$ra** prima del salto)

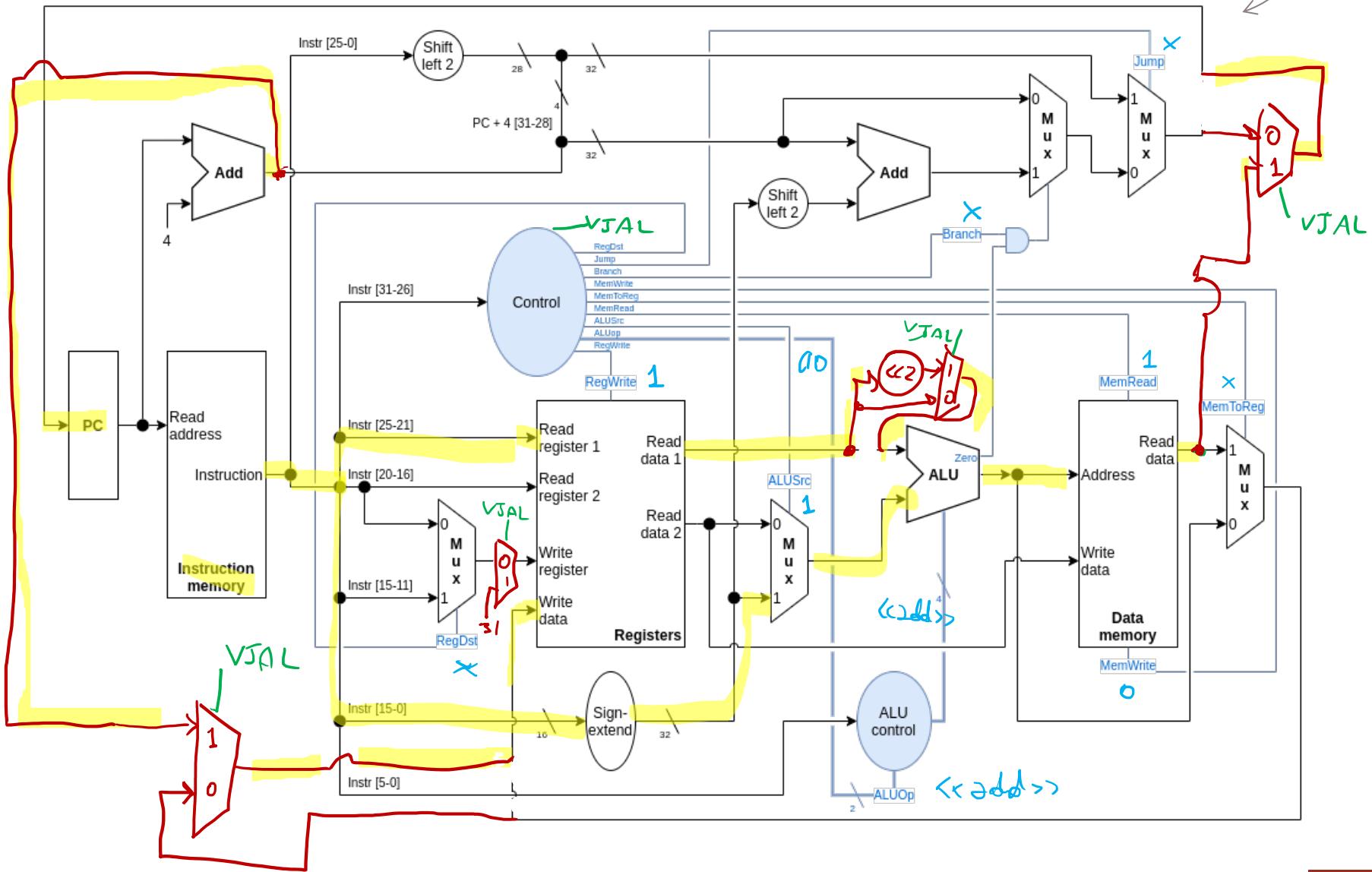
Qui:
scarabocchi



Esercizio per casa: vjal (soluzione /a)



Esercizio per casa: vjal (soluzione /a)



vjal rs, constant (soluzione /b, c)

(b) Segnali dalla Control Unit

VRJ	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
1	X	1	X	1	1	0	X	X	0	0

Accesso a memorie = 75ns, accesso ai registri = 25ns, ALU e sommatori = 100ns

(c) Tempo di esecuzione				
Fetch 75ns	Reg[rs] 25ns	add = rs<<2 + const. 100ns	jumpTo = Mem[add] 75ns	PC ← jumpTo 0ns
PC+4 100ns		Reg[31] = PC+4 25ns		
Totale: 275ns				

Non è necessario incrementare il periodo di clock

Istruzione	Instr. Fetch	Instr. Decode	Execution	MEM	Wr. Back	Totale
lw	75	25	100	75	25	300



Esercizio per casa (nuovo): vj

Ex1: vuole aggiungere alla CPU l'istruzione vectorized jump (vj), di tipo I e sintassi assembly

vj \$indice, vettore

che salta all'indirizzo contenuto nell'elemento \$indice-esimo del vettore di word.

Esempio: se in memoria si è definito staticamente il

vettore: .word 15, 24, 313, 42

e nel registro \$t0 c'è il valore 3 allora

vj \$t0, vettore

salterà all'indirizzo 42 (che è l'elemento con indice 3 del vettore)

- si disegnino le modifiche necessarie a realizzare la funzione, aggiungendo tutti gli eventuali MUX, segnali di controllo, bus, ALU e sommatori (ecc) che ritenete necessari.
- indicate i valori di tutti i segnali di controllo, in modo da eseguire l'istruzione vj.
- tenendo conto che il tempo di accesso ai registri (sia in lettura che scrittura) è di **25ps**, l'accesso alla memoria impiega **75ps**, la ALU e i sommatori impiegano **100ps** e ignorando gli altri ritardi, calcolate il tempo di esecuzione minimo della istruzione vj e indicate se è necessario aumentare il periodo di clock della CPU per poter svolgere questa nuova istruzione.

Ex2: Si svolga nuovamente l'ex se il vettore in memoria dati è ora

vettore: .half 15, 24, 313, 42

Capire se va bene la soluzione precedente oppure no.

L'Istruzione vj con half word

Ex2: Si svolga nuovamente l'ex se il vettore in memoria dati è ora
vettore: .half 15, 24, 313, 42

Capire se va bene la soluzione precedente oppure no.

Che cosa cambia?

Vengono lette **half word** dalla memoria ossia 2 byte, ossia 16 bit.

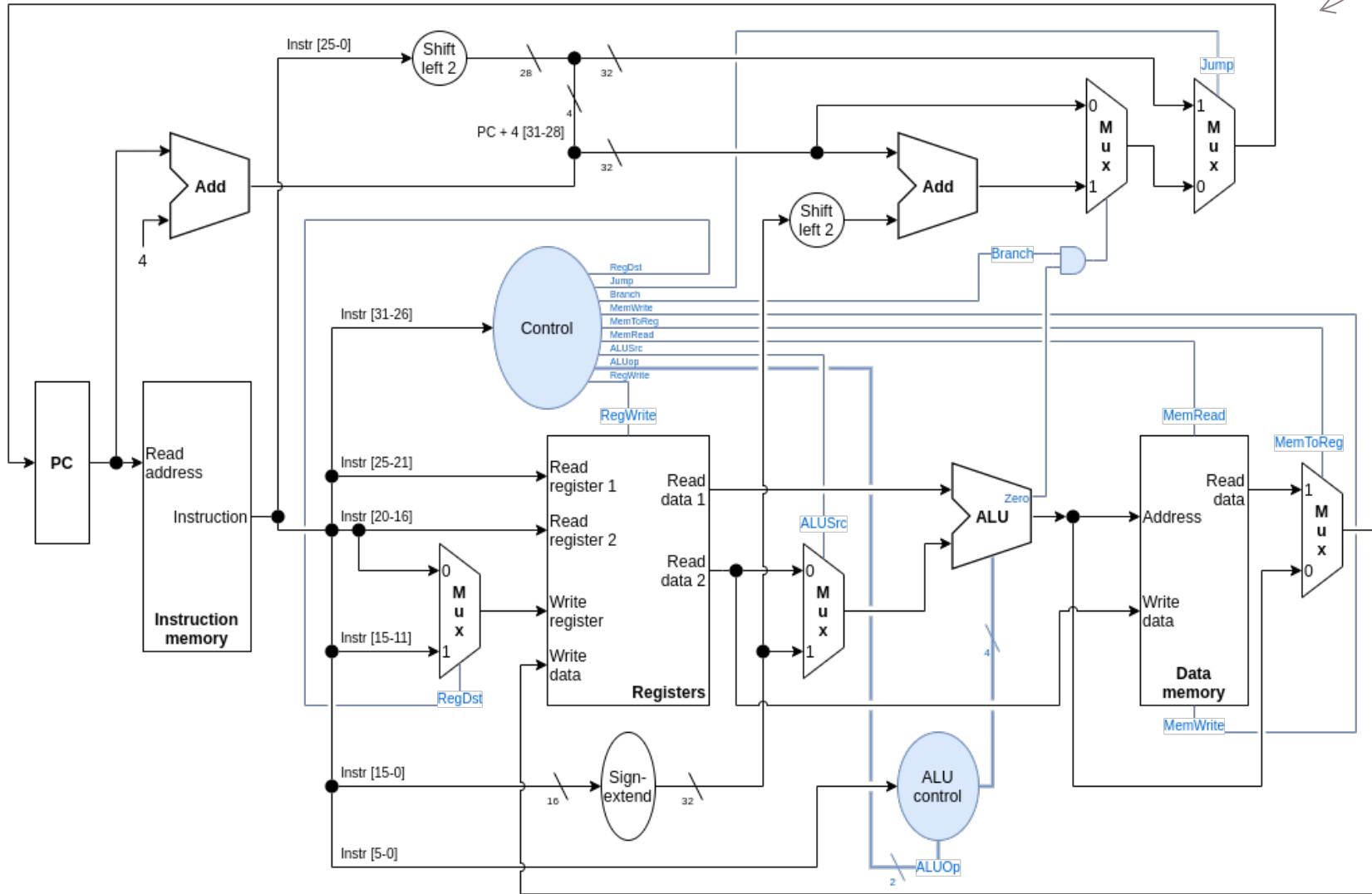
Possiamo usare la soluzione precedente come base ma è necessario:

1. Fare uno shift di una posizione ($\text{Reg}[rs] \ll 1$) in maniera tale da moltiplicare per 2 byte l'indice del vettore.
2. In uscita dalla memoria l'indirizzo è a 16 bit (o più precisamente solo i primi 16bit sono quello che vogliamo leggere), quindi non può essere usato direttamente per aggiornare il program counter (32 bit). Una possibilità è estendere il segno così da portare i 16 bit a 32 bit.

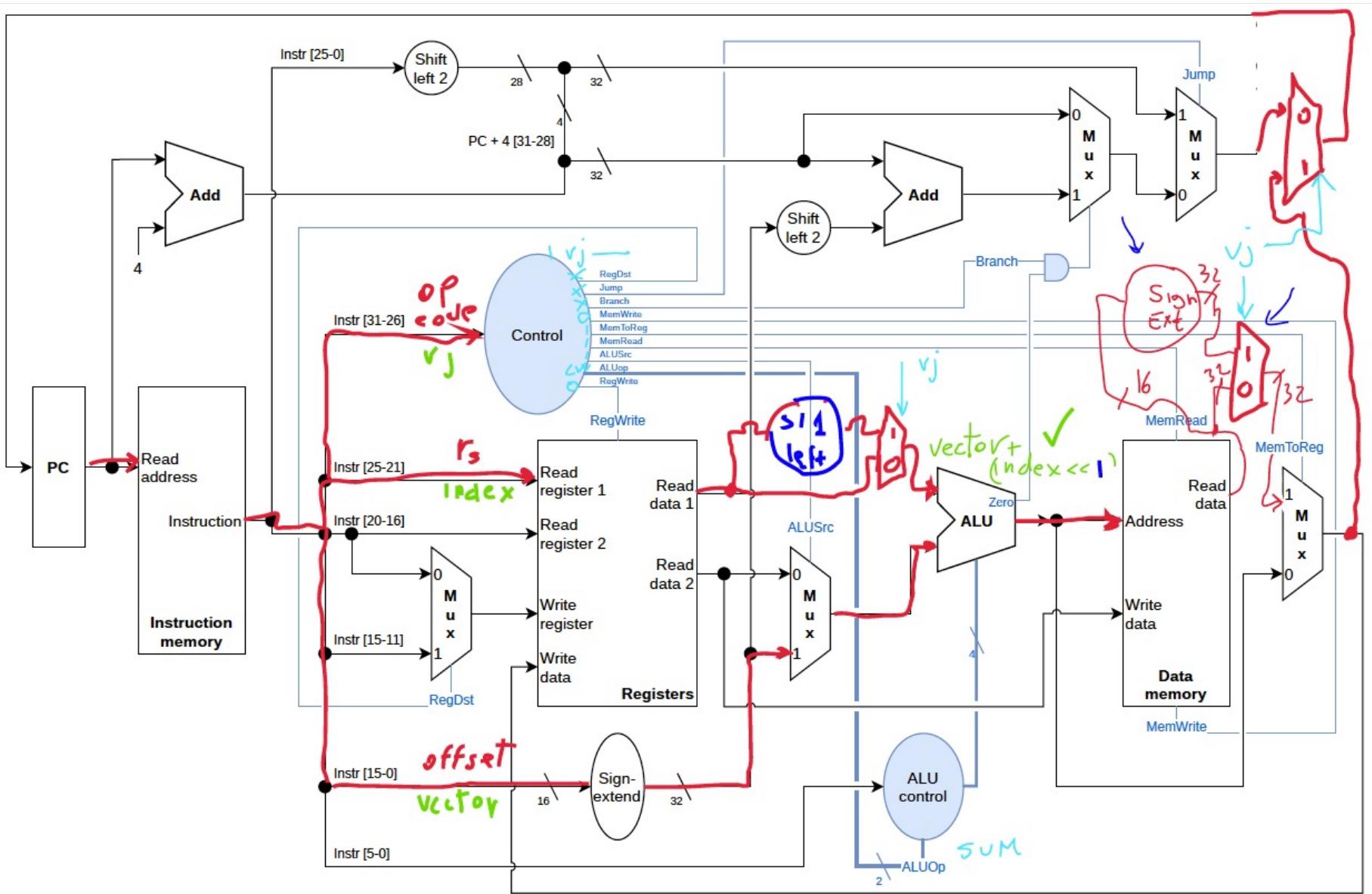
Qui:
scarabocchi



L'Istruzione vj con half word



L'Istruzione vj con half word



Esercizi per casa



SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio

Esercizio (esame 16-9-14)

1. Individuare i **data** e
control hazard

- assumendo la **decisione** dei branch in fase ID

2. Individuare i casi in cui il **forwarding** è applicabile

3. Individuare gli **stalli** necessari

4. Calcolare la **durata totale** in cicli di clock con **forwarding**

5. Calcolare la **durata** senza **forwarding**

6. Riordinare le istruzioni per ridurre gli **stalli** (con fwd.)

7. Calcolare la **durata** del programma **ottimizzato** (4.)

8. Cosa c'è in **pipeline** al 16° CC (v. programma originale)?

Nota: Assumiamo che `bge $t1, $zero, ciclo` sia un'istruzione del set standard. In realtà, è una pseudoistruzione tradotta in `slt $1, $9, $0` `beq $1, $0, <offset istr. ciclo>`

.data

vettore: .word 7, 14, ..., 48, 91 # 70 pari, 30 dispari

.text

main:

li \$t0, 0 # somma parziale

li \$t1, 396 # offset ultimo

ciclo:

lw \$t3, vettore(\$t1)

andi \$t2, \$t3, 1 # è dispari?

beqz \$t2, salta # se non lo è, lo ignoro

add \$t0, \$t0, \$t3 # altrimenti, lo sommo

salta:

addi \$t1, \$t1, -4 # prossimo elemento

bge \$t1, \$zero, ciclo # fine del ciclo? —

li \$v0, 1 # stampa...

move \$a0, \$t0 # ... la somma

syscall



Codice OPIS per valutare il corso

https://www.uniroma1.it/sites/default/files/field_file_allegati/vademecum_per_studenti_opis_2023_24_1.pdf

Codice OPIS per valutare il corso

DUZD3JFI