Esercizi sulla Pipeline con Data Forwarding

Architettura degli elaboratori

Laurea in Informatica

Docente: Federico Corò

Data Hazards

`istruzione i`
`istruzione i+1`

- Read after Write : “lettura dopo scrittura” (esempio di prima – *i+1* legge *prima* che *i* abbia scritto
- Write after Write : “scrittura dopo scrittura” – *i+1* scrive *prima* che *i* abbia scritto (problema con parallelismo)
- **Write after Read**: “scrittura dopo lettura”
  – *i+1* scrive *prima* che *i* abbia letto (caso raro in pipeline)

Soluzioni:

1. Introduzione di stalli
2. Data forwarding (propagazione in avanti del dato richiesto)
3. Riordino delle istruzioni

Pipeline RISC-V

Stadi della pipeline:

1. **IF** Instruction Fetch
2. **ID** Instruction Decode / register fetch
3. **EX** Execution / address calculation
   - tutte le istruzioni usano la ALU (tranne salto incondizionato):
     - operazioni logico-aritmentiche
     - load/store e jump per calcolare l'indirizzo
     - salti condizionati per calcolare la condizione
       - Un altro componente calcola la destinazione
4. **MEM** Memory access / branch completion
5. **WB** Write Back: scrittura del risultato nei registri

Pipeline RISC-V - Istruzioni

Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	Immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

Figura 2.19 Quattro dei formati delle istruzioni RISC-V. La Figura 4.16 rivelerà i due formati RISC-V mancanti: SB per i salti condizionati e UJ per i salti incondizionati. I formati SB e UJ prevedono gli stessi campi con la stessa ampiezza dei formati S e U ma che hanno alcuni bit dell'indirizzo spostati. Il razionale dietro questa codifica diventa chiaro dopo avere capito il funzionamento dell'hardware. Come vedremo nel Capitolo 4, i formati SB e UJ semplificano l'hardware ma producono un lavoro maggiore agli assemblatori (e agli autori!).

codop: codice operatore (indica come interpretare il resto dei bit)
rs1: specifica registro sorgente (primo)
rs2: specifica registro sorgente (secondo)
rd: specifica registro destinazione
immediato: dato immediato
funz3: bit aggiuntivi al codop di specifica per ALU (indica op. specifica)

Operandi RISC-V

Nome	Esempio	Commenti
32 registri	x0-x31	Accesso veloce ai dati. Nel RISC-V gli operandi devono essere contenuti nei registri per potere eseguire delle operazioni. Il registro x0 contiene sempre il valore 0
2^{30} parole di memoria	Memoria[0], Memoria[4], … Memoria[4 294 967 292]	Alla memoria si accede solamente attraverso istruzioni di trasferimento dati. Il RISC-V utilizza l'indirizzamento al byte, perciò due variabili successive) hanno indirizzi in memoria a distanza 4. La memoria consente di memorizzare strutture dati, vettori, o il contenuto dei registri

Linguaggio assembler RISC-V

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	`add x5, x6, x7`	`x5 = x6 + x7`	Operandi in tre registri
	Sottrazione	`sub x5, x6, x7`	`x5 = x6 – x7`	Operandi in tre registri
	Somma immediata	`addi x5, x6, 20`	`x5 = x6 + 20`	Utilizzata per sommare delle costanti
Trasferimento dati	Lettura parola	`lw x5, 40(x6)`	`x5 = Memoria[x6 + 40]`	Spostamento di una parola da memoria a registro
	Lettura parola, senza segno	`lwu x5, 40(x6)`	`x5 = Memoria[x6+40]`	Spostamento di una parola senza segno da memoria a registro
	Memorizzazione parola	`sw x5, 40(x6)`	`Memoria[x6+40] = x5`	Spostamento di una parola da registro a memoria
	Lettura mezza parola	`lh x5, 40(x6)`	`x5 = Memoria[x6+40]`	Spostamento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	`lhu x5, 40(x6)`	`x5 = Memoria[x6+40]`	Spostamento di una mezza parola senza segno da memoria a registro
	Memorizzazione mezza parola	`sh x5, 40(x6)`	`Memoria[x6+40]= x5`	Spostamento di una mezza parola da registro a memoria
	Lettura byte	`lb x5, 40(x6)`	`x5 = Memoria[x6+40]`	Spostamento di un byte da memoria a registro
	Lettura byte, senza segno	`lbu x5, 40(x6)`	`x5 = Memoria[x6+40]`	Spostamento di un byte senza segno da memoria a registro
	Memorizzazione byte	`sb x5, 40(x6)`	`Memoria[x6+40] = x5`	Spostamento di un byte da registro a memoria
	Lettura di una parola e blocco	`lr.d x5, (x6)`	`x5 = Memoria[x6]`	Caricamento di una parola come prima fase di un'operazione atomica sulla memoria
	Memorizzazione condizionata di una parola	`sc.d x7, x5, (x6)`	`Memoria[x6] = x5; x7 = 0/1`	Memorizzazione di una parola come seconda fase di un'operazione atomica sulla memoria
	Caricamento costante nella mezza parola superiore	`lui x5, 0x12345`	`x5 = 0x12345000`	Caricamento di una costante su 20 bit nei 12 bit più significativi di una parola
Logiche	And	`and x5, x6, x7`	`x5 = x6 & x7`	Operandi in tre registri; AND bit a bit
	Or inclusivo	`or x5, x6, x8`	`x5 = x6 | x8`	Operandi in tre registri; OR bit a bit
	Or esclusivo (Xor)	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Operandi in tre registri; XOR bit a bit
	And immediato	`andi x5, x6, 20`	`x5 = x6 & 20`	AND bit a bit tra un operando in registro e una costante

Figura 2.1 Il linguaggio assembler del RISC-V esaminato in questo capitolo. Si possono trovare queste informazioni anche nella prima colonna della scheda tecnica riassuntiva del RISC-V in fondo al libro. *(continua)*

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Logiche *(continua)*	Or esclusivo immediato	`ori x5, x6, 20`	`x5 = x6 | 20`	OR bit a bit tra un operando in registro e una costante
	Xor immediato	`xori x5, x6, 20`	`x5 = x6 ^ 20`	XOR bit a bit tra un operando in registro e una costante
Scorrimento (shift)	Scorrimento logico a sinistra	`sll x5, x6, x7`	`x5 = x6 << x7`	Scorrimento a sinistra mediante registro
	Scorrimento logico a destra	`srl x5, x6, x7`	`x5 = x6 >> x7`	Scorrimento a destra mediante registro
	Scorrimento aritmetico a destra	`sra x5, x6, x7`	`x5 = x6 >> x7`	Scorrimento a destra aritmetico mediante registro
	Scorrimento logico a sinistra immediato	`slli x5, x6, 3`	`x5 = x6 << 3`	Scorrimento a sinistra mediante costante
	Scorrimento logico a destra immediato	`srli x5, x6, 3`	`x5 = x6 >> 3`	Scorrimento a destra mediante costante
	Scorrimento aritmetico a destra immediato	`srai x5, x6, 3`	`x5 = x6 >> 3`	Scorrimento a destra aritmetico mediante costante
Salti condizionati	Salta se uguale	`beq x5, x6, 100`	`Se (x5 == x6) vai a PC+100`	Test di uguaglianza; salto relativo al PC
	Salta se non è uguale	`bne x5, x6, 100`	`Se (x5 != x6) vai a PC+100`	Test di disuguaglianza; salto relativo al PC
	Salta se minore di	`blt x5, x6, 100`	`Se (x5 < x6) vai a PC+100`	Comparazione di minoranza; salto relativo al PC
	Salta se maggiore o uguale di	`bge x5, x6, 100`	`Se (x5 >= x6) vai a PC+100`	Comparazione di maggioranza o uguaglianza; salto relativo al PC
	Salta se minore di, senza segno	`bltu x5, x6, 100`	`Se (x5 < x6) vai a PC+100`	Comparazione di minoranza senza segno; salto relativo al PC
	Salta se maggiore o uguale di, senza segno	`bgeu x5, x6, 100`	`Se (x5 >= x6) vai a PC+100`	Comparazione di maggioranza o uguaglianza senza segno; salto relativo al PC
Salti incondizionati	Salta e collega	`jal x1, 100`	`x1 = PC+4; vai a PC+100`	Chiamata a procedura con indirizzamento relativo al PC
	Salta e collega mediante registro	`jalr x1, 100(x5)`	`x1 = PC+4; vai a x5+100`	Ritorno da procedura; chiamata indiretta

Figura 2.1 Il linguaggio assembler del RISC-V esaminato in questo capitolo. *(continua)*

Pipeline RISC-V

- Le fasi IF e ID non dipendono dai segnali di controllo
- Nella Pipeline RISC-V è possibile individuare tutte le dipendenze dai dati nella fase ID
- Hazard Detection Unit
  - Inserisce stalli (bolle) o attiva meccanismi di data forwarding
- Data Forwarding: meccanismo che permette di trasferire dati tra buffer di registry di stadi diversi, per ridurre il numero di stalli da inserire

Data Forwarding

- Esempio: realizziamo un forwarding nella fase EX per una dipendenza di tipo RAW (Read After Write) con sorgente che proviene da una istruzione load

LW x1, 42 (x2)

..

ADD x3, x1, x8

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	EX	MEM	WB						
		IF	ID	~~ID~~	EX	MEM	WB				

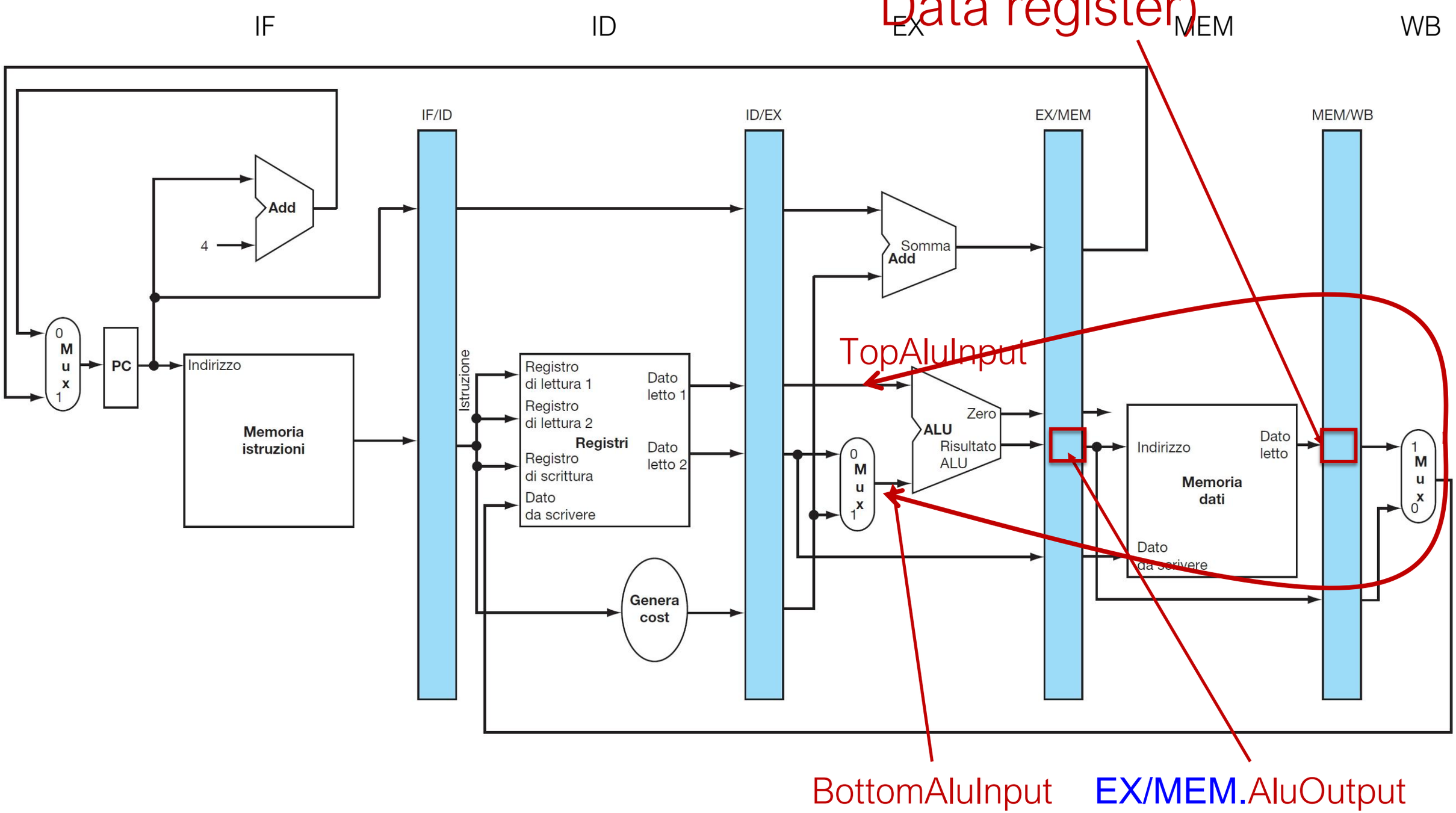
Qui viene scritto x1

Qui vuole leggere x1

Qui è noto il valore che prenderà x1

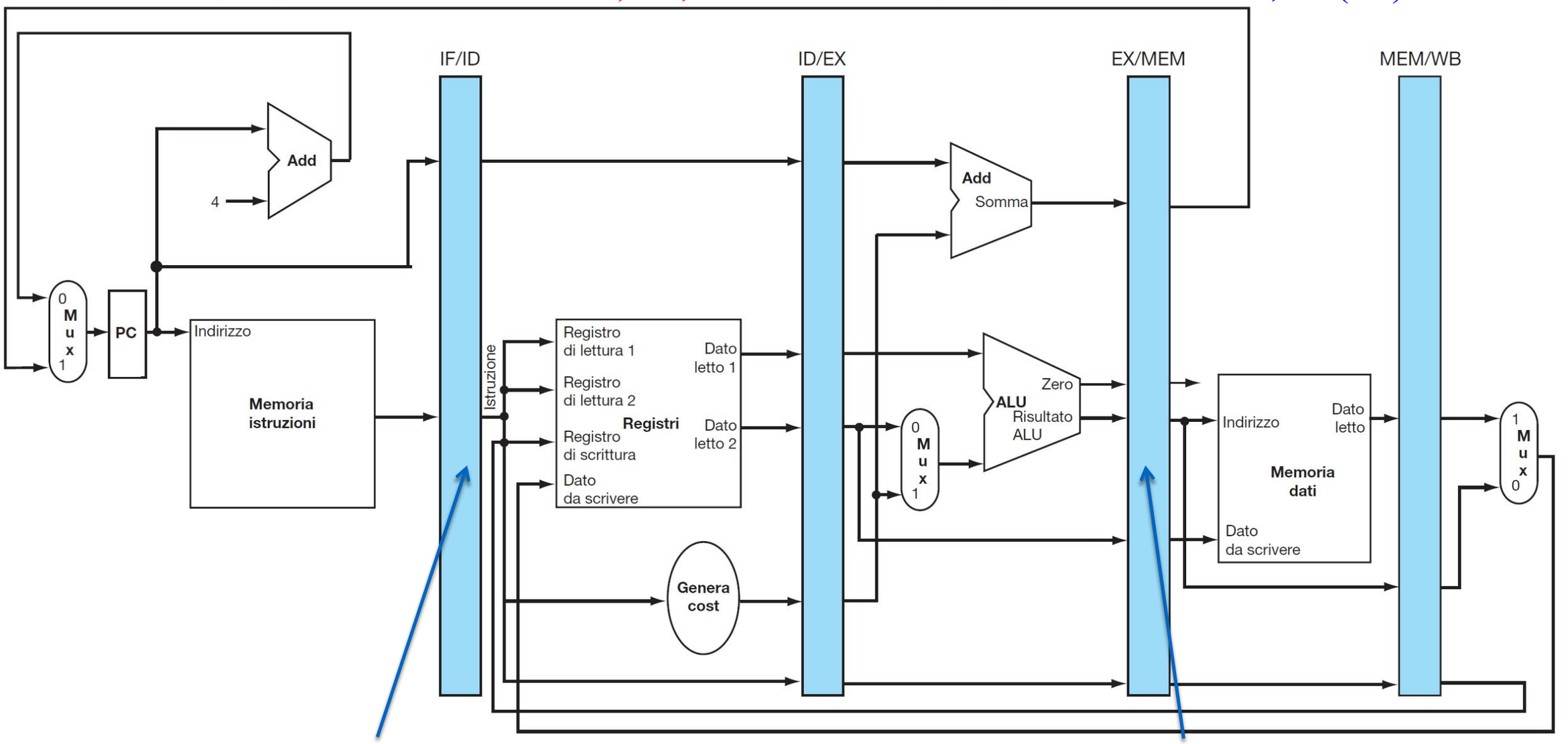
Pipeline RISC-V

MEM/WB.LMD (Load Memory Data register)



IF ID EX MEM WB

OR x9, x6, x7 ADD x3, **x1**, x8 ADD x5, x6, x7 LW **x1**, 42(x2)



ID/EX.IR[opcode] =Reg-Reg ALU

ID/EX.IR[rs1] = x1

MEM/WB.IR[opcode] = LW

MEM/WB.IR[rd] =x1

sono uguali!

C'è una RAW

risolvibile con forward

Hazard Detection Unit non fa nulla e SUB viene rilasciata

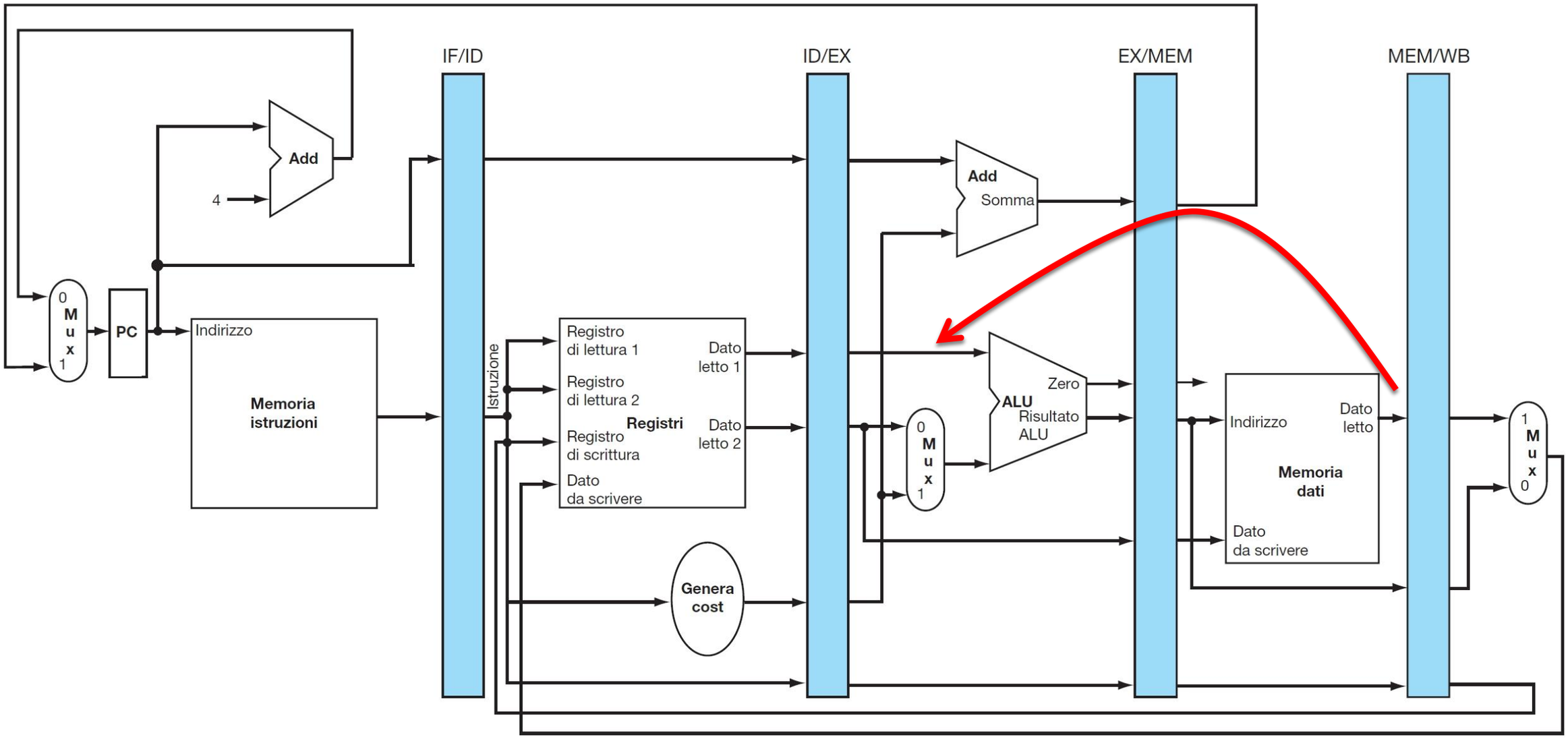
Read after

Write

ciclo successivo

IF ID EX MEM WB

OR x9, x6, x7 ADD x3, **x1**, x8 ADD x5, x6, x7 LW **x1**, 42(x2)

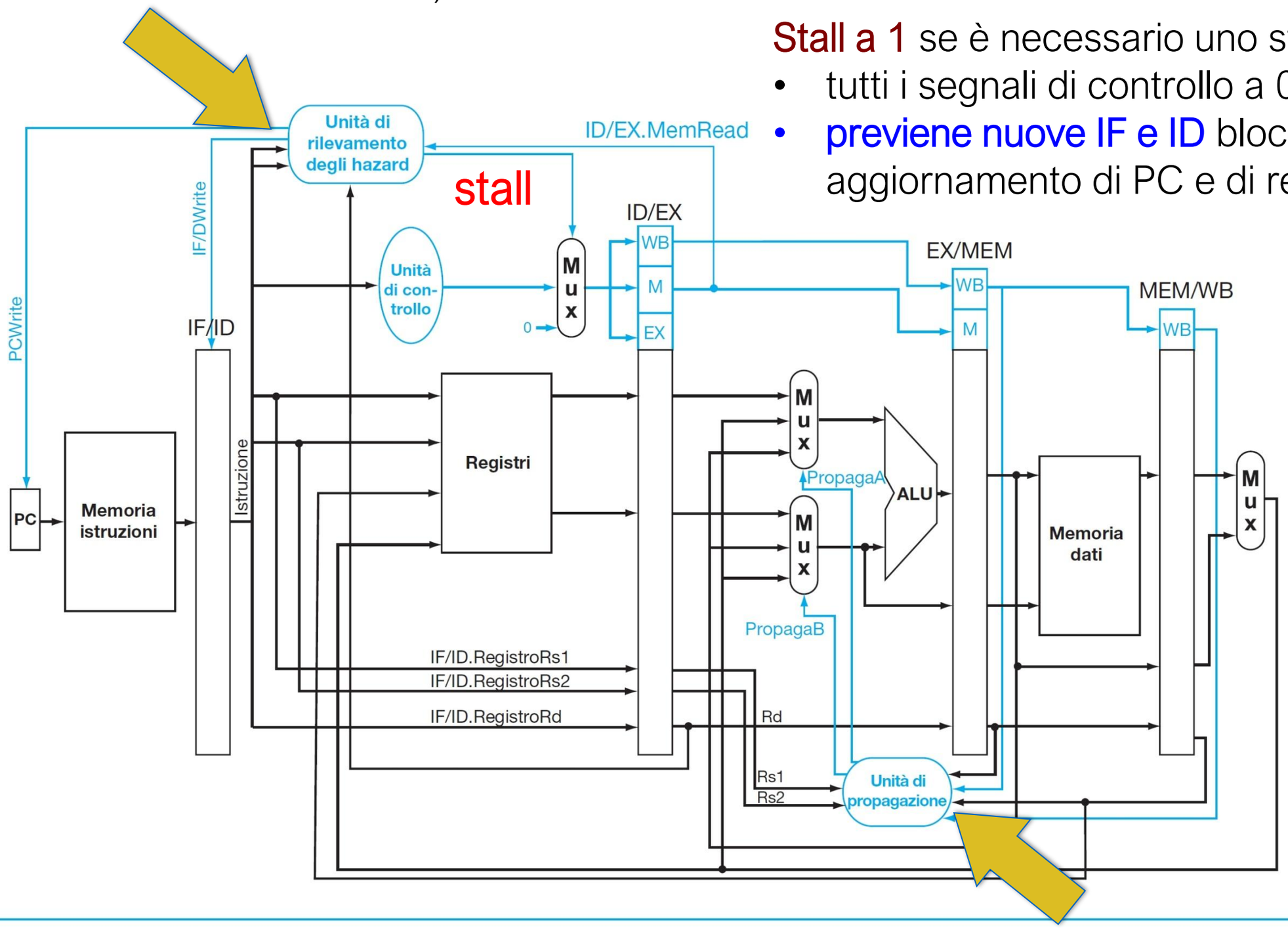


Forwarding Unit Effettua il forward da Load a Reg-Reg-ALU

- se MEM/WB.IR[rt] == ID/EX.IR[rs1] e MEM/WB.RegWrite allora **manda MEM/WB.LMD** a Top ALU Input

Pipeline RISC-V

Circuito di data forwarding implementato con multiplexer controllati dalla CU (in particolare dalla Hazard Detection Unit)



Stall a 1 se è necessario uno stallo:

- tutti i segnali di controllo a 0 (*bubble*)
- previene nuove IF e ID bloccando aggiornamento di PC e di registro IF/ID

Figura 4.62 Panoramica sul controllo della pipeline: nella figura compaiono i due multiplexer per la propagazione, l'unità di rilevamento degli hazard e l'unità di propagazione. Sebbene gli stadi ID ed EX siano stati semplificati (mancano la logica per l'estensione del segno del campo immediato e la logica dei salti), lo schema mostra gli elementi essenziali dell'hardware per la propagazione.

Pipeline RISC-V

Condizioni per riconoscere le dipendenze che richiedono STALLI

istr. precedente in fase EX

```
if  (ID/EX.MemRead and
     ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
       (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
       stall the pipeline
```

registro di **scrittura** istr. precedente

Istruzione corrente in fase ID

registro di **lettura** di Istruzione corrente

condizioni verificate da **Hazard Detection Unit** in fase **ID**

Pipeline RISC-V

Condizioni per riconoscere le dipendenze che richiedono data forwarding

istr. precedente in fase EX

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

istr. precedente in fase MEM

registro di **scrittura** istr. precedente

Istruzione corrente in fase EX

registro di **lettura** di Istruzione corrente

condizioni verificate da **ForwardingUnit** in fase **EX**

Pipeline RISC-V

Condizioni per riconoscere le dipendenze che richiedono forwarding da un hazard nello stadio EX

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

condizioni verificate da **ForwardingUnit** in fase **EX**

Pipeline RISC-V

Condizioni per riconoscere le dipendenze che richiedono forwarding da un hazard nello stadio MEM

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

condizioni verificate da **ForwardingUnit** in fase **EX**

Segnali di controllo per 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

Figura 4.57 Il valore assunto dai segnali di controllo dei multiplexer di propagazione della Figura 4.56. Il campo immediato dotato di segno che costituisce un ulteriore ingresso della ALU viene descritto nella sezione *Approfondimento* al termine di questo paragrafo.

Pipeline RISC-V

Condizioni per riconoscere le dipendenze da un hazard nello stadio MEM

- Una complicazione è il potenziale rischio di dati tra il risultato dell'istruzione nella fase WB, il risultato dell'istruzione nella fase MEM e l'operando sorgente dell'istruzione nella fase ALU. Ad esempio, quando si somma un vettore di numeri in un singolo registro, una sequenza di istruzioni leggerà e scriverà tutte nello stesso registro:

add x1, x1, x2

add x1, x1, x3

add x1, x1, x4

...

- In questo caso, il risultato dovrebbe essere inoltrato dalla fase MEM perché il risultato nella fase MEM è il risultato più recente. Pertanto, il controllo del pericolo MEM sarebbe (con le aggiunte evidenziate):

```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
         and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Possibili Data Forward

Provenienza:

- Da memoria dati: registro MEM/WB.LMD (Load Memory Data Register)
- Dall'output della ALU: registro EX/MEM.AluOutput

  oppure MEM/WB.AluOutput

Destinazione:

- Input della ALU:
  - TopAluInput o BottomAluInput (entrambe in EX/MEM)
  - ATTENZIONE: per store, TopAluInput è il registro per lo spiazzamento (rs1) mentre BottomAluInput è il registro sorgente del dato (rs2)

Esercizio 3: Pipeline RISC-V e data forwarding

Si consideri la pipeline RISC-V a 5 stadi vista a lezione, CON possibilità di data-forwarding e con possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock.
Per ognuna delle seguenti sequenze di istruzioni assembler:

1. indicare quali dipendenze dai dati sono presenti;
2. mostrare come evolve la pipeline durante l'esecuzione del codice, spiegando nel dettaglio i motivi di un eventuale stallo e/o forward
3. indicare se è possibile riordinare le istruzioni in modo da ridurre le dipendenze dai dati.

Sequenza 1	Sequenza 2	Sequenza 3
SUB x2, x7, x5	LW x3, 80 (x0)	SW x9, 0 (x1)
LW x1, 7 (x2)	ADD x2, x3, x1	LW x1, 7 (x9)
ADD x2, x1, x8	LW x1, 800(x2)	SUB x9, x1, x8
SW x3, 73 (x1)	ADDI x1, x1, -3	SW x3, 73 (x9)
ADDI x2, x3, -4	ADDI x2, x2, 4	ADDI x9, x3, -9
ADDI x7, x3, 8	SW x1, 108(x2)	SW x7, 78 (x9)
ADD x1,x7,x2	SUB x4,x3,x1	LW x9,A(x7)

Esercizio 3: Pipeline RISC-V e riordino

SUB x2, x7, x5

RAW LW x1, 7 (x2)

ADD x2, x1, x8

SW x3, 73 (x1)

ADDI x2, x3, -4

ADDI x7, x3, 8

ADD x1, x7, x2

1	2	3	4	5	6	7	8	9	10
IF	ID	EX	MEM	WB					
	IF	ID	EX	MEM	WB				

La Hazard Detection Unit non deve inserire stalli

Qui **l'Unità di Propagazio**ne si accorge che il registro di lettura della LW ID/EX.IR[rs1] è uguale al registro di scrittura della SUB EX/MEM.IR[rd] e EX/MEM.RegWrite => RAW x2

Risolvibile con data forwarding: EX/MEM.AluOutput inviato a ID/EX.TopAluInput

Esercizio 3: Pipeline MIPS e riordino

SUB x2, x7, x5
LW x1, 7 (x2)
WAW/ ADD x2, x1, x8
RAW
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2

1	2	3	4	5	6	7	8	9	10	11
IF	ID	EX	MEM	WB						
	IF	ID	EX	MEM	WB					
		IF	ID	~~ID~~	EX	MEM	WB			

RAW su x1: LW determina il valore da scrivere in x1 in fase MEM quindi serve
1 stallo (inserito da Hazard Detection Unit in fase ID)
Hazard Detection Unit si accorge che il registro di **lettura** IF/ID.IR[rs1]
è uguale al registro di **scrittura** ID/EX.IR[*rd*] e ID/EX.MemRead => **STALLO**

+ data forward: MEM/WB.LMD inviato a TopAluInput attivato da
Unità di propagazione in fase EX **da condizione MEM/WB**.IR[rd]
= ID/EX.IR[rs1] and MEM/WB.RegWrite

Esercizio 3: Pipeline MIPS e riordino

SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
RAW SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	EX	MEM	WB						
		IF	ID	~~ID~~	EX	MEM	WB				
			IF	~~IF~~	ID	EX	MEM	WB			

ID non può iniziare prima che l'istruzione precedente passi alla fase successiva

il registro di **lettura** IF/ID.IR[rs1]
è uguale al registro di **scrittura** *della LW*
MEM/WB.IR[*rd*] => RAW su x1

ma nella prima metà del ciclo 6 WB di LW scrive x1
e nella seconda metà ID legge x1, quindi **tutto ok**

Esercizio 3: Pipeline MIPS e riordino

SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2

WAW /RAR

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	EX	MEM	WB						
		IF	ID	~~ID~~	EX	MEM	WB				
			IF	~~IF~~	ID	EX	MEM	WB			
					IF	ID	EX	MEM	WB		

IF non può iniziare prima che l'istruzione precedente passi alla fase successiva

Tutto OK:
- x3 è usato in **lettura** sia da ADDI che da SW
- ADDI scrive su x2 in fase WB, senza conflitto con i precedenti

Esercizio 3: Pipeline MIPS e riordino

SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	EX	MEM	WB						
		IF	ID	~~ID~~	EX	MEM	WB				
			IF	~~IF~~	ID	EX	MEM	WB			
					IF	ID	EX	MEM	WB		
						IF	ID	EX	MEM	WB	

OK: x3 è già stato usato in lettura sia da ADDI che da SW

Esercizio 3: Pipeline MIPS e riordino

	1	2	3	4	5	6	7	8	9	10	11	12
SUB x2, x7, x5	IF	ID	EX	MEM	WB							
LW x1, 7 (x2)		IF	ID	EX	MEM	WB						
ADD x2, x1, x8			IF	ID	~~ID~~	EX	MEM	WB				
SW x3, 73 (x1)				IF	~~IF~~	ID	EX	MEM	WB			
ADDI x2, x3, -4						IF	ID	EX	MEM	WB		
ADDI x7, x3, 8							IF	ID	EX	MEM	WB	
RAW ADD x1, x7, x2								IF	ID	EX	MEM	WB

RAW: x7 è scritto dall'istruzione precedente e x2 da quella prima. Hazard Detection Unit però non si attiva (non servono stalli) perché…

Dipendenze risolvibili con data forward da Unità di Propagazione:

x7: EX/MEM.AluOutput inviato a ID/EX.TopAluInput attivata da condizione ID/EX.IR[rs1] = EX/MEM.IR[rd] and EX/MEM.RegWrite

x2: MEM/WB.AluOutput va a ID/EX.BottomAluInput attivata da condizione ID/EX.IR[rs2] = MEM/WB.IR[rd] and MEM/WB.RegWrite

Esercizio 3 seq 1: riordino?

- Idea: riordino le istruzioni per diminuire il tempo totale di esecuzione
    - Devo ridurre le dipendenze tra istruzioni che generano stalli

SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2

Esercizio 3 seq 1: riordino?

```
SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2
```

deve leggere il contenuto di x2, che è definito nell'istruzione precedente, quindi non si può anticipare

se si anticipa ADD prima di LW, si modifica il contenuto di x2, quindi LW carica indirizzo diverso e il programma cambia semantica

SW si può scambiare con ADD, ma non elimina la necessità dello stallo perché anche SW legge x1

deve venire dopo le due precedenti perchè legge x7 e x2

Esercizio 3: riordino?

SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1) ← posso posticiparla?
ADDI x2, x3, -4
ADDI x7, x3, 8 ← Non creava problemi, posso anticiparla?
ADD x1, x7, x2

Entrambe le possibilità sono accettabili, MA migliorano il tempo totale?

Esercizio 3 sequenza 1: riordino?

- Questa opzione toglie il doppio forward, ma non stalli (ne abbiamo inserito solo 1 per la ADD in linea 3)

```
SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2
```

```
SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
ADDI x2, x3, -4
ADDI x7, x3, 8
SW x3, 73 (x1)
ADD x1, x7, x2
```

Esercizio 3 sequenza 1: riordino?

- Questa opzione toglie lo stallo dalla ADD.
- Verificarlo (per casa)!

```
SUB x2, x7, x5
LW x1, 7 (x2)
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADDI x7, x3, 8
ADD x1, x7, x2
```

```
SUB x2, x7, x5
LW x1, 7 (x2)
ADDI x7, x3, 8
ADD x2, x1, x8
SW x3, 73 (x1)
ADDI x2, x3, -4
ADD x1, x7, x2
```

Esercizio 3: Sequenza 2

1. indicare quali dipendenze dai dati sono presenti;

2. mostrare come evolve la pipeline durante l'esecuzione del codice, spiegando nel dettaglio i motivi di un eventuale stallo e/o data forward

LW x3, 80 (x0)	R1 <- mem[80+[R0]]
ADD x2, x3, x1	R2 <- [R3]+[R1]
LW x1, 800(x2)	R1 <- mem[800+[R2]]
ADDI x1, x1, -3	R1 <- [R1]-3
ADDI x2, x2, 4	R2 <- [R2]+4
SW x1, 108(x2)	mem[108+[R2]] <- [R1]
SUB x4,x3, x1	R4 <- [R3]-[R1]

Soluzione esercizio 3 sequenza 2

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							

LW determina il valore da scrivere in x3 in fase MEM quindi serve **1 stallo**

Hazard Detection Unit si accorge che il registro di **lettura** IF/ID.IR[rs1] è uguale al registro di **scrittura** ID/EX.IR[rd] e ID/EX.MemRead=> STALLO

+ data forward determinato in fase EX da unità di Propagazione:
MEM/WB.LMD (Load Memory Data register) inviato a TopAluInput per la condizione MEM/WB.IR[rd] = ID/EX.IR[rs1] e MEM/WB.RegWrite

Soluzione esercizio 3 sequenza 2

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						

Hazard Detection Unit non rileva dipendenze per cui serve inserire uno stallo

Unità di Propagazione si accorge che il registro di **lettura** ID/EX.IR[rs1]
è uguale al registro di **scrittura** EX/MEM.IR[***rd***] e EX/MEM.RegWrite=> **RAW** su x2

risovibile con data forwarding: EX/MEM.AluOutput inviato a TopAluInput

Soluzione esercizio 3 sequenza 2

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	~~ID~~	EX	MEM	WB				

LW determina il valore da scrivere in x1 in fase MEM quindi serve **1 stallo + data forward**

Hazard Detection Unit si accorge che il registro di **lettura** IF/ID.IR[rs1] è uguale al registro di **scrittura** ID/EX.IR[rd] e ID/EX.MemRead=> STALLO

Unità di propagazione attiva data forward per la condizione **MEM/WB.REgWrite and MEM/WB.IR[rd]=ID/EX.IR[rs1]**: MEM/WB.LMD inviato a TopAluInput

Soluzione esercizio 3 sequenza 2

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	~~ID~~	EX	MEM	WB				
					IF	~~IF~~	ID	EX	MEM	WB			

Tutto OK

Soluzione esercizio 3 sequenza 2

Hazard Detection Unit non viene attivata

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW x3, 80 (x0)	IF	ID	EX	MEM	WB									
ADD x2, x3, x1		IF	ID	~~ID~~	EX	MEM	WB							
LW x1, 800(x2)			IF	~~IF~~	ID	EX	MEM	WB						
ADDI x1, x1, -3					IF	ID	~~ID~~	EX	MEM	WB				
ADDI x2, x2, 4						IF	~~IF~~	ID	EX	MEM	WB			
SW x1, 108(x2)								IF	ID	EX				
SUB x4,x3, x1														

Unità di propagazione si accorge di RAW su x2, (condizione EX/MEM.IR[rd]=ID/EX.IR[rs1] and EX/MEM.RegWrite) risolvibile con **forward** di EX/MEM.AluOutput verso TopAluInput

ma c'è un'altra RAW su x1 risolvibile **dall'unità di propagazione**:
SW in fase ID dovrebbe anche **leggere** x1=IF/ID.IR[rs2] e **propagarlo** nei registri di pipeline **fino alla fase MEM** ma il valore corretto di x1 sarà scritto in fase WB di ADDI
Per fare questo, su può utilizzare il circuito **di forward** di MEM/WB.AluOutput verso BottomAluInput (MEM/WB.IR[rd]=ID/EX.IR[rs2] and MEM/WB.RegWrite).

Notare che il dato in rs2 della SW in realtà NON passa per la ALU. Tuttavia, multiplexer gestiti da appositi segnali di controllo propagano il dato in avanti.

Soluzione esercizio 3 sequenza 2

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	~~ID~~	EX	MEM	WB				
					IF	~~IF~~	ID	EX	MEM	WB			
							IF	ID	EX	MEM	WB		

Soluzione esercizio 3 sequenza 2

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	~~ID~~	EX	MEM	WB				
					IF	~~IF~~	ID	EX	MEM	WB			
							IF	ID	EX	MEM	WB		
								IF	ID	EX	MEM	WB	

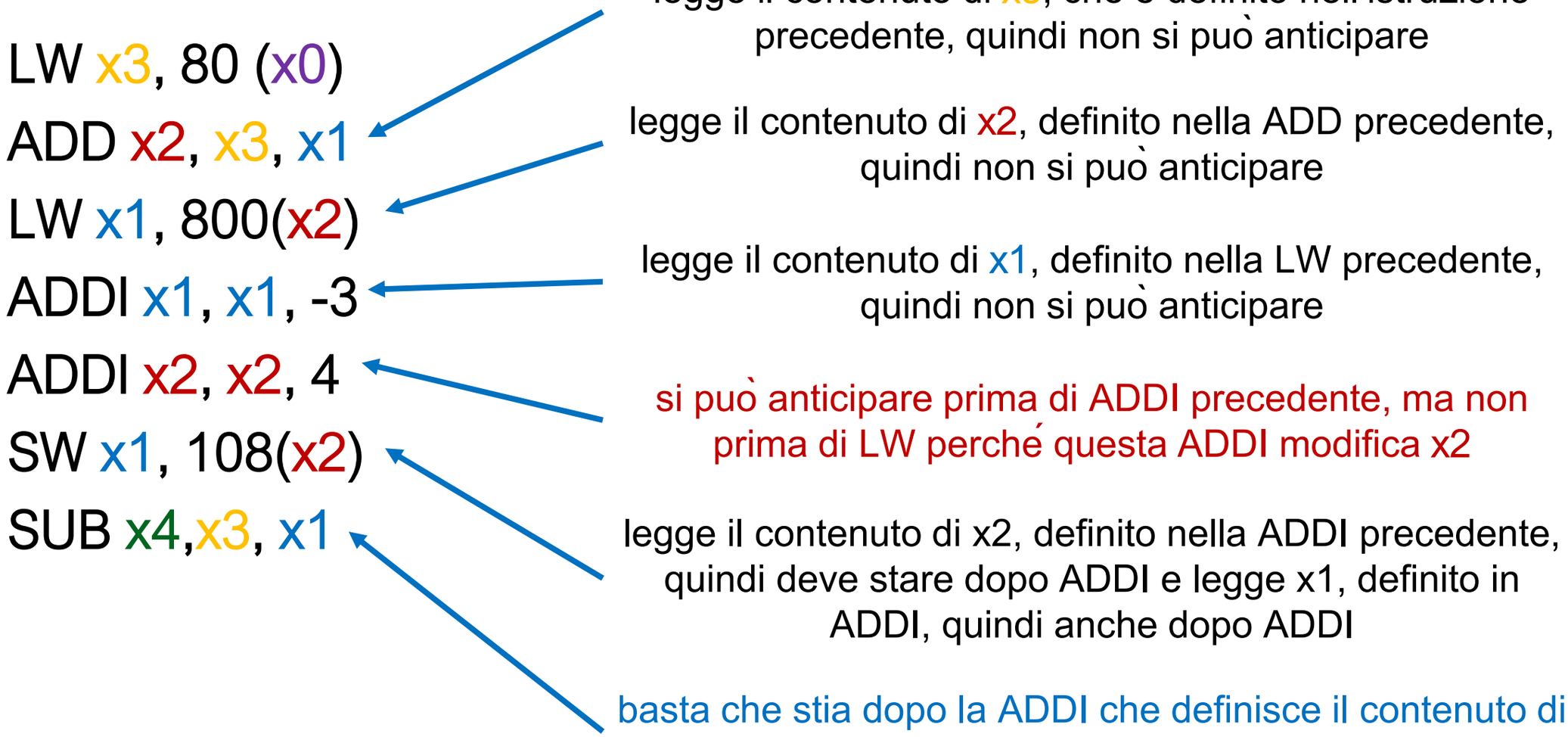
Tutto ok:
•nella prima metà del ciclo ADDI scrive x1
•nella seconda metà del ciclo SW può leggere x1

•Quindi non serve ne stallo ne DF
•ATTENZIONE: SW legge x1, non lo scrive, quindi non crea problemi!

Data Forward - considerazioni

- Riguardare (o eventualmente ricalcolare) la soluzione di Esercizio 3 sequenza 2 senza data forward
- Esercizio 3 sequenza 2 richiedeva 19 istanti temporali (e.g. cicli di clock) per completare la sua esecuzione senza data forward
- Con data forward solo 13!

Esercizio 3 sequenza 2: riordino?



Riordino esercizio 3 seq 2

```
LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x1, x1, -3
ADDI x2, x2, 4
SW x1, 108(x2)
SUB x4,x3, x1
```

```
LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x2, x2, 4
ADDI x1, x1, -3
SW x1, 108(x2)
SUB x4,x3, x1
```

Idea: se siamo fortunati rimuoviamo lo stallo per la dipendenza da x1 della prima ADDI rispetto alla seconda ADDI

Possibile riordino:

- La semantica del programma rimane la stessa

esercizio 3 sequenza 2 riordinata

LW x3, 80 (x0)
ADD x2, x3, x1
LW x1, 800(x2)
ADDI x2, x2, 4
ADDI x1, x1, -3
SW x1, 108(x2)
SUB x4,x3, x1

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	EX	MEM	WB					
					IF	ID	EX	MEM	WB				
						IF	ID	EX	MEM	WB			
							IF	ID	EX	MEM	WB		

- 12 cicli di clock invece che 13! Sfruttiamo maggiormente i circuiti di data forwarding

Esercizio 3 sequenza 3

SW x9, 0 (x1)

LW x1, 7(x9)

SUB x9, x1, x8

SW x3, 73 (x9)

ADDI x9, x3, -9

SW x7, 78 (x9)

LW x9, A (x7)

Esercizio 3 sequenza 3

SW x9, 0 (x1)
LW x1, 7, x9
SUB x9, x1, x8
SW x3, 73 (x9)
ADDI x9, x3, -9
SW x7, 78 (x9)
LW x9, A (x7)

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	EX	MEM	WB						
		IF	ID	~~ID~~	EX	MEM	WB				
			IF	~~IF~~	ID	EX	MEM	WB			
					IF	ID	EX	MEM	WB		
						IF	ID	EX	MEM	WB	
							IF	ID	EX	MEM	WB

Hazard Detection Unit (ID/EX. MemRead and ID/EX.IR[rd] =IF/ID.IR[rs1]) inserisce 1 stallo

Unità di propagazione verifica la condizione MEM/WB.RegWrite and MEM/WB.IR[rd] =ID/EX.IR[rs1]
E attiva Data Forward MEM/WB.LMD ->ID/EX.TopAluInput

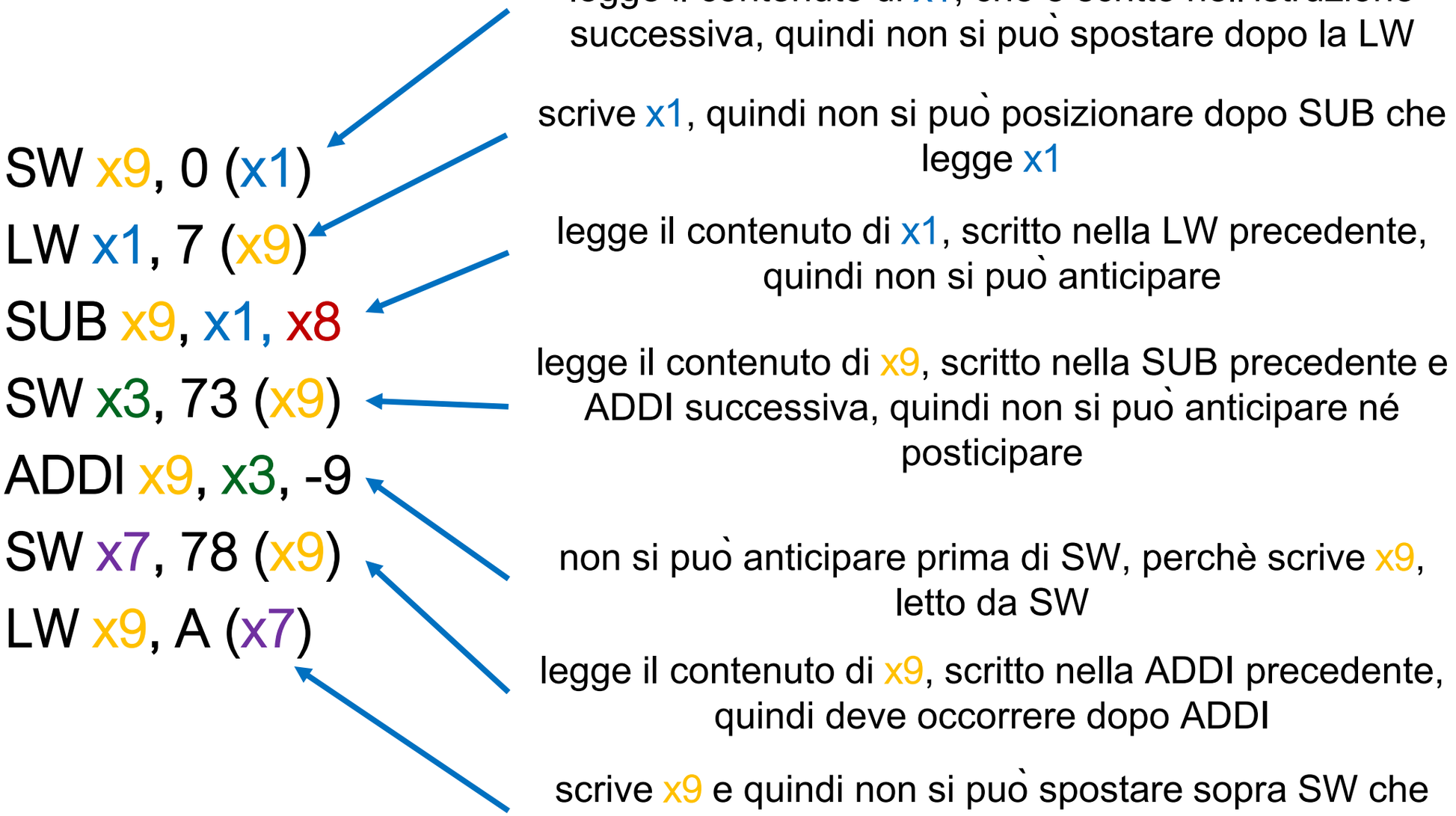
Esercizio 3 sequenza 3

SW x9, 0 (x1)
LW x1, 7, x9
SUB x9, x1, x8
SW x3, 73 (x9)
ADDI x9, x3, -9
SW x7, 78 (x9)
LW x9, A (x7)

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	EX	MEM	WB						
		IF	ID	~~ID~~	EX	MEM	WB				
			IF	~~IF~~	ID	EX	MEM	WB			
					IF	ID	EX	MEM	WB		
						IF	ID	EX	MEM	WB	
							IF	ID	EX	MEM	WB

Unità di propagazione verifica EX/MEM.RegWrite and EX/MEM.IR[rd] =ID/EX.IR[rs1] e attiva Data Forward EX/MEM.AluOutput -> TopAluInput

Esercizio 3 sequenza 3: riordino



Nessun riordino possibile!

Esercizio 4: sequenza 1 con branch

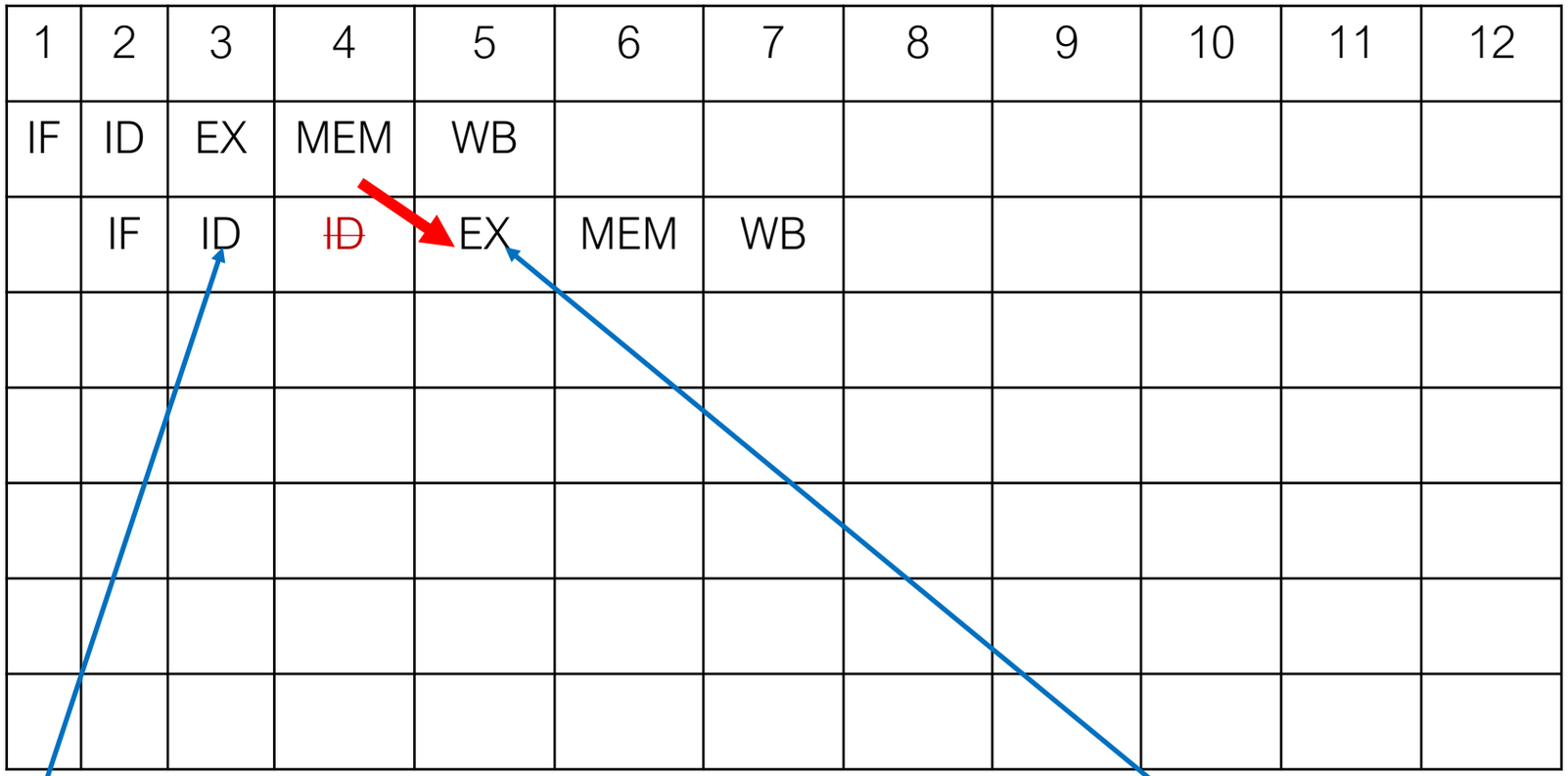
```
LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)
```

Per semplicità

- Le istruzioni sono di 32 bit. Quindi la BEQ a linea 4, se salta, va all'ultima SW
  - PC=PC+4+8
- non assumiamo alcuna tecnica di predizione dei salti
- in fase EX di istruzione beq calcola la condizione e il target, ma è in fase MEM che decide se saltare, cioè usa la condizione per decidere il valore di PC

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)



1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	~~ID~~	EX	MEM	WB					

Hazard Detection Unit verifica I/EX.MemRead and ID/EX.IR[rd] =IF/ID.IR[rs2]: infatti c'è RAW su x1
Inserisce quindi uno stallo

Unità di propagazione verifica MEM/WB.RegWrite and MEM/WB.IR[rd]=ID/EX.IR[rs2], quindi attiva **data forward**:
MEM/WB.**LMD ->** BottomALUInput

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	~~ID~~	EX	MEM	WB					
		IF	~~IF~~	ID	EX	MEM	WB				

RAW su x1 e x2

Hazard detection unit non inserisce stalli

dovrebbe leggere x2 in fase ID ma sfrutta DF EX/MEM.AluOut -> ID/EX.BottomAluInput per leggerlo in EX
Il DF viene attivato **dall'unità di propagazione** con la condizione EX/MEM.RegWrite and EX/MEM.IR[rd]=ID/EX.IR[rs2]

- lettura in seconda metà di ciclo così si risolve anche RAW x1

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)

Hazard detection unit non inserisce stalli

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	~~ID~~	EX	MEM	WB					
		IF	~~IF~~	ID	EX	MEM	WB				
				IF	ID	EX	MEM	WB			

beq legge sia x2 che x1, che erano scritte da istruzioni LW e ADD:
- LW al ciclo 6 è già completata

In ciclo 8 usa la condizione di salto.

Per il momento, supponiamo sia falsa

- Per x2 invece serve data forwarding **dall'unità di propagazione** da MEM/WB.AluOut a ID/EX.TopAluInput perchè MEM/WB.IR[rd]=ID/EX.IR[rs1]

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)

Hazard detection unit non inserisce stalli

1	2	3	4	5	6	7	8	9	10	11	12
IF	ID	EX	MEM	WB							
	IF	ID	~~ID~~	EX	MEM	WB					
		IF	~~IF~~	ID	EX	MEM	WB				
				IF	ID	EX	MEM	WB			
					IF	ID	EX	MEM	WB		

Unità di propagazione: nessuna condizione verificata

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	EX	MEM	WB					
					IF	ID	EX	MEM	WB				
						IF	ID	EX	MEM	WB			

Hazard detection unit
non inserisce stalli

Unità di propagazione verifica EX/MEM.RegWrite and EX/MEM.IR[rd] =ID/EX.IR[rs2] -> RAW x3
si risolve con forward:
EX/MEM.**ALUOutput ->** BottomALUInput

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	ID	EX	MEM	WB										
	IF	ID	~~ID~~	EX	MEM	WB								
		IF	~~IF~~	ID	EX	MEM	WB							
				IF	ID	EX	MEM	WB						
					IF	ID	EX	MEM	WB					
						IF	ID	EX	MEM	WB				
							IF	ID	EX	MEM	WB			

Unità di propagazione verifica DUE condizione
EX/MEM.IR[rd] =ID/EX.IR[rs1] = -> RAW x1
MEM/WB.IR[rd] =IF/ID.IR[rs2] -> RAW x3

Attiva quindi I due circuiti:
Data forward per x1: **EX/MEM.AluOutput** **->**TopAluInput
Data forward per x3: **MEM/WB.AluOutput** **->**BottomAluInput

Esercizio 4 sequenza 1

- Segnali di controllo

In fase MEM decide come aggiornare PC a seconda se salto preso o no

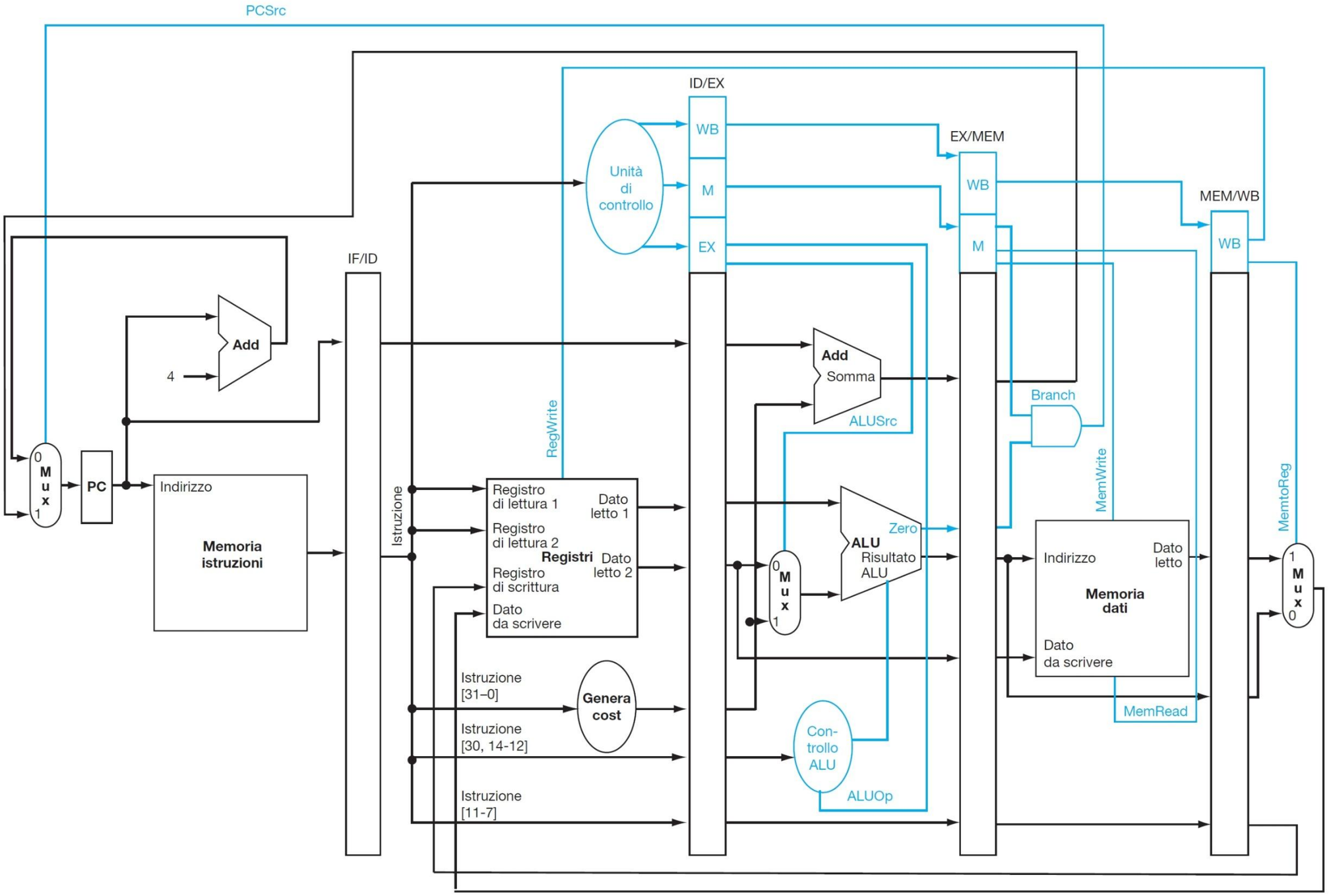


Figura 4.53 L'unità di elaborazione con pipeline della Figura 4.48 completa dei segnali di controllo uscenti dalla porzione dei registri di

Esercizio 4 sequenza 1

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
SW x3, 0 (x1)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	ID	EX	MEM	WB									
	IF	ID	~~ID~~	EX	MEM	WB							
		IF	~~IF~~	ID	EX	MEM	WB						
				IF	ID	EX	MEM	WB					
					IF	ID	EX						
						IF	ID						
							IF	ID	EX	MEM	WB		

Se il salto fosse stato preso?
Al tempo 8, avrebbe modificato il PC

Svuotamento Pipeline (vengono rimosse le due istruzioni successive)

Il PC non viene modificato, l'istruzione target va in IF (ci sarebbe andata lostesso)

Esercizio 4 sequenza 1 MODIFICATA

- E se l'istruzione target del salto non fosse già stata in pipeline?

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
…
SW x3, 0 (x1)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	ID	EX	MEM	WB										
	IF	ID	~~ID~~	EX	MEM	WB								
		IF	~~IF~~	ID	EX	MEM	WB							
				IF	ID	EX	MEM	WB						
					IF	ID	EX							
						IF	ID							

Se il salto fosse stato preso?
Al tempo 8, avrebbe modificato il PC, alterando l'istruzione di cui si fa il fetch

Esercizio 4 sequenza 1 MODIFICATA

- E se l'istruzione target del salto non fosse già stata in pipeline?

LW x1, 0 (x2)
ADD x2, x3, x1
SW x2, 21 (x1)
BEQ x2, x1, 8
ADD x3, x2, x2
ADD x1, x1, x3
…
SW x3, 0 (x1)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	ID	EX	MEM	WB										
	IF	ID	~~ID~~	EX	MEM	WB								
		IF	~~IF~~	ID	EX	MEM	WB							
				IF	ID	EX	MEM	WB						
					IF	ID	EX							
						IF	ID							
							IF	ID	EX	MEM	WB			

Se il salto fosse stato preso?
Al tempo 8, avrebbe modificato il PC

L'istruzione target NON era già in pipeline. La fase IF inizia DURANTE la fase MEM di BEQ perchè prima scrive il PC e poi lo legge

Esercizio 4: sequenza 2 con branch

```
        LW x2, 0 (x1)
Label1: BEQ x2, x0, Label2
        LW x3, 0 (x2)
        BEQ x3, x0, Label1
        ADD x1, x3, x1
Label2: SW x1, 0 (x2)
```

Per semplicità

- usiamo Label al posto del campo Imm (esteso a 32 bit) da usare come offset per calcolare l'indirizzo del salto:

  beq x1, x2, Imm if ([R1]-[R2]==0) then PC=PC+4+(Imm<<2)

- non assumiamo alcuna tecnica di predizione dei salti
- in fase EX di istruzione beq calcola la condizione e il target, ma è in fase MEM che decide se saltare, cioè usa la condizione per decidere il valore di PC
  - Quindi decide l'istruzione di cui fare IF nello stesso ciclo dells fase MEM della branch