POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Pipelining e Hazard
Capitolo 4 P&H

16. 11. 2015

Problema dei conflitti

Conflitti strutturali: tentativo di usare la stessa risorsa da parte di diverse istruzioni in modi diversi nello stesso intervallo di tempo:

- Esempio: se avessimo un'unica memoria per istruzioni e dati

Conflitti sui dati: tentativo di utilizzare un risultato prima che sia pronto

- Esempio: istruzione che dipende dal risultato di un'istruzione precedente che è ancora nella pipeline

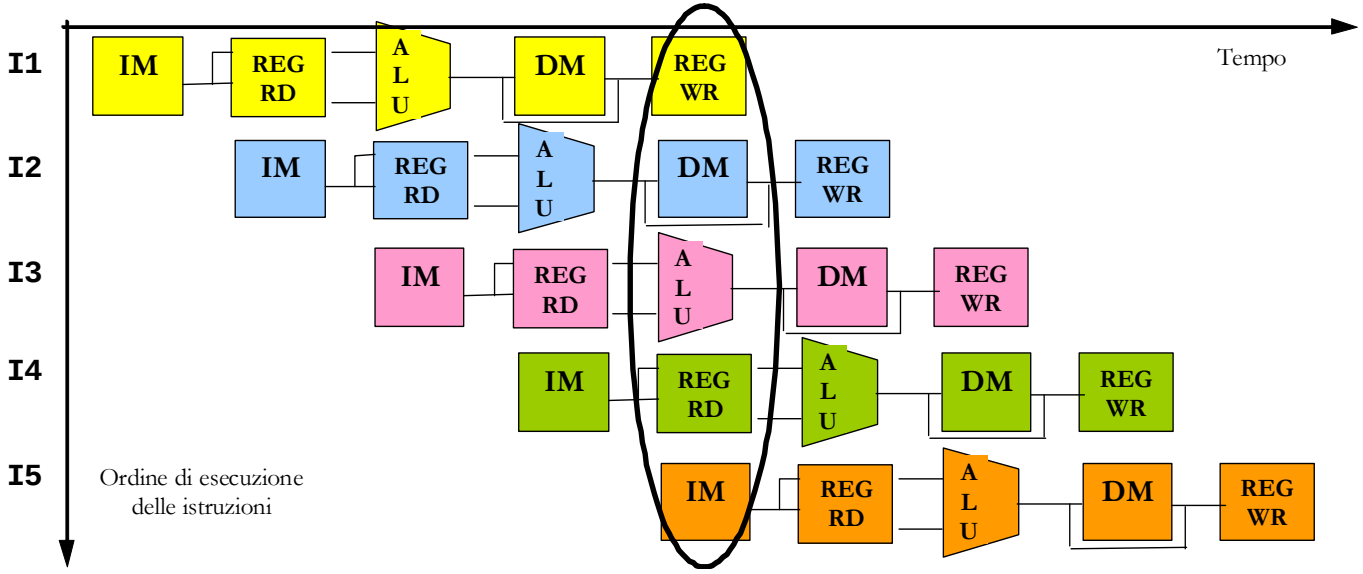
Conflitti sul controllo: tentativo di prendere una decisione sulla prossima istruzione da eseguire prima che la condizione sia valutata

- Esempio: istruzioni di salto condizionato

Problema dei conflitti strutturali

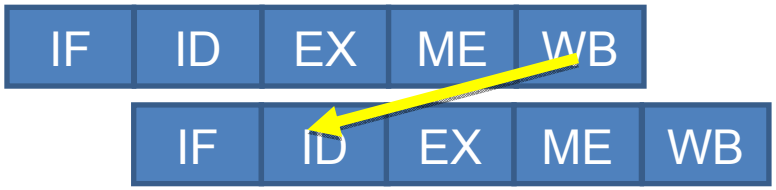
Nell'architettura MIPS **non** abbiamo conflitti strutturali:

- Memoria Istruzioni separata dalla Memoria Dati
- Banco di Registri usato nello stesso ciclo di clock in lettura e scrittura ma in modo coerente con la tecnica di temporizzazione

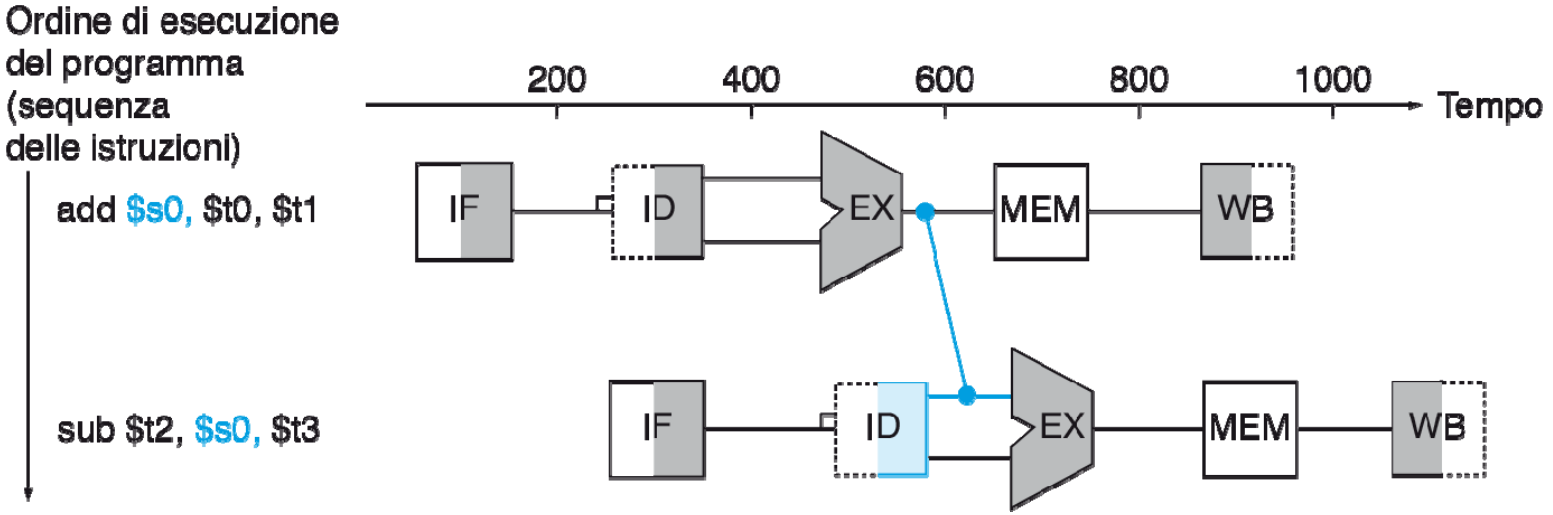


Conflitto di dati (*data hazard*): *R/R*

```
add  $s0, $t0, $t1
sub $t2, $s0, $t3
```

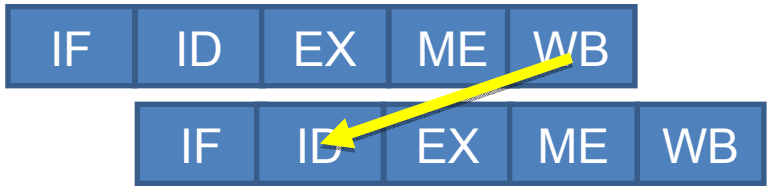


- il valore corretto è disponibile già all'uscita dello stadio EX di **add**
- circuito di *Propagazione/bypassing*: l'uscita dello stadio EX viene anche portata direttamente ai suoi ingressi (propagazione EX/EX)

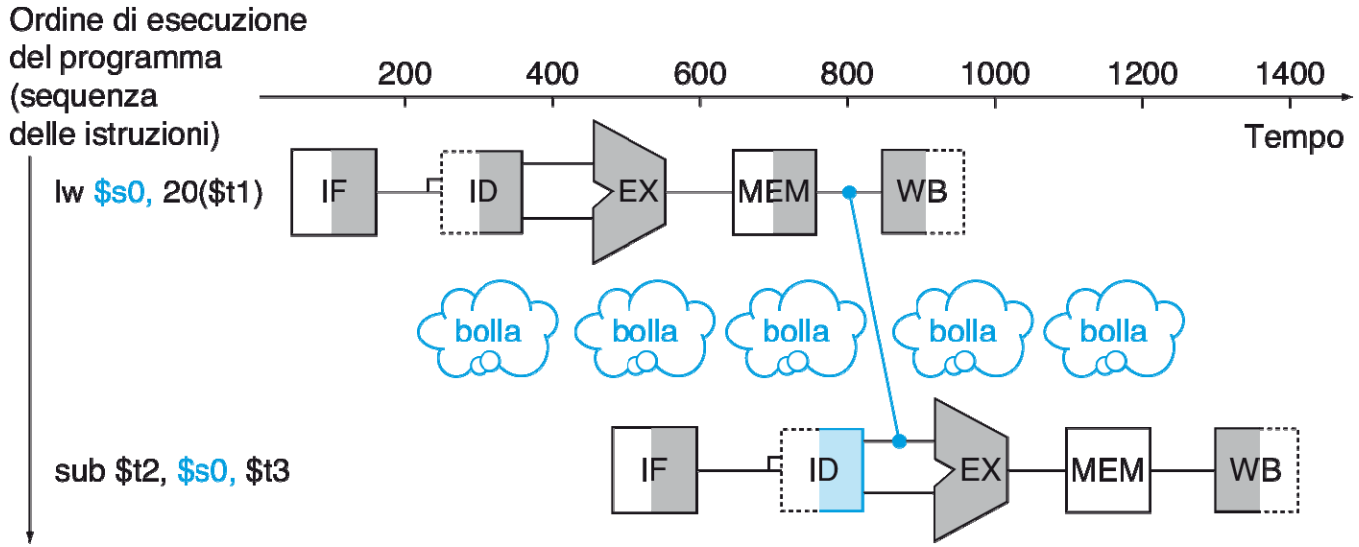


Conflitto di dati (*data hazard*): *Load/R*

```
lw  $s0, 20($t1)
sub $t2, $s0, $t3
```

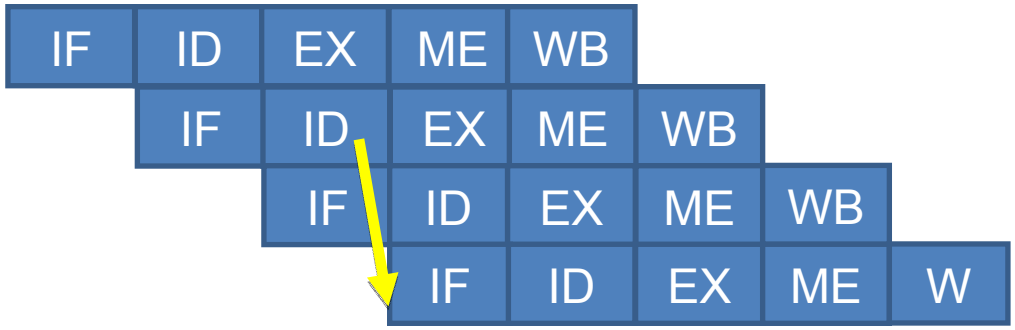


- il valore corretto è disponibile già all'uscita dello stadio MEM di **`lw`**
- circuito di *Propagazione/bypassing*: l'uscita dello stadio MEM viene anche portata direttamente agli ingressi dello stadio EX (MEM/EX), ma non basta .....
- si deve inserire un ciclo di ritardo nell'esecuzione di **sub**



Conflitto di controllo (*control hazard*): *beq*

```
     add  $4, $5, $6
Y    beq $1, $2, 40
Y+4  lw $3, 300($0)
Z    or $7, $8, $9
```



Pipeline ottimizzata

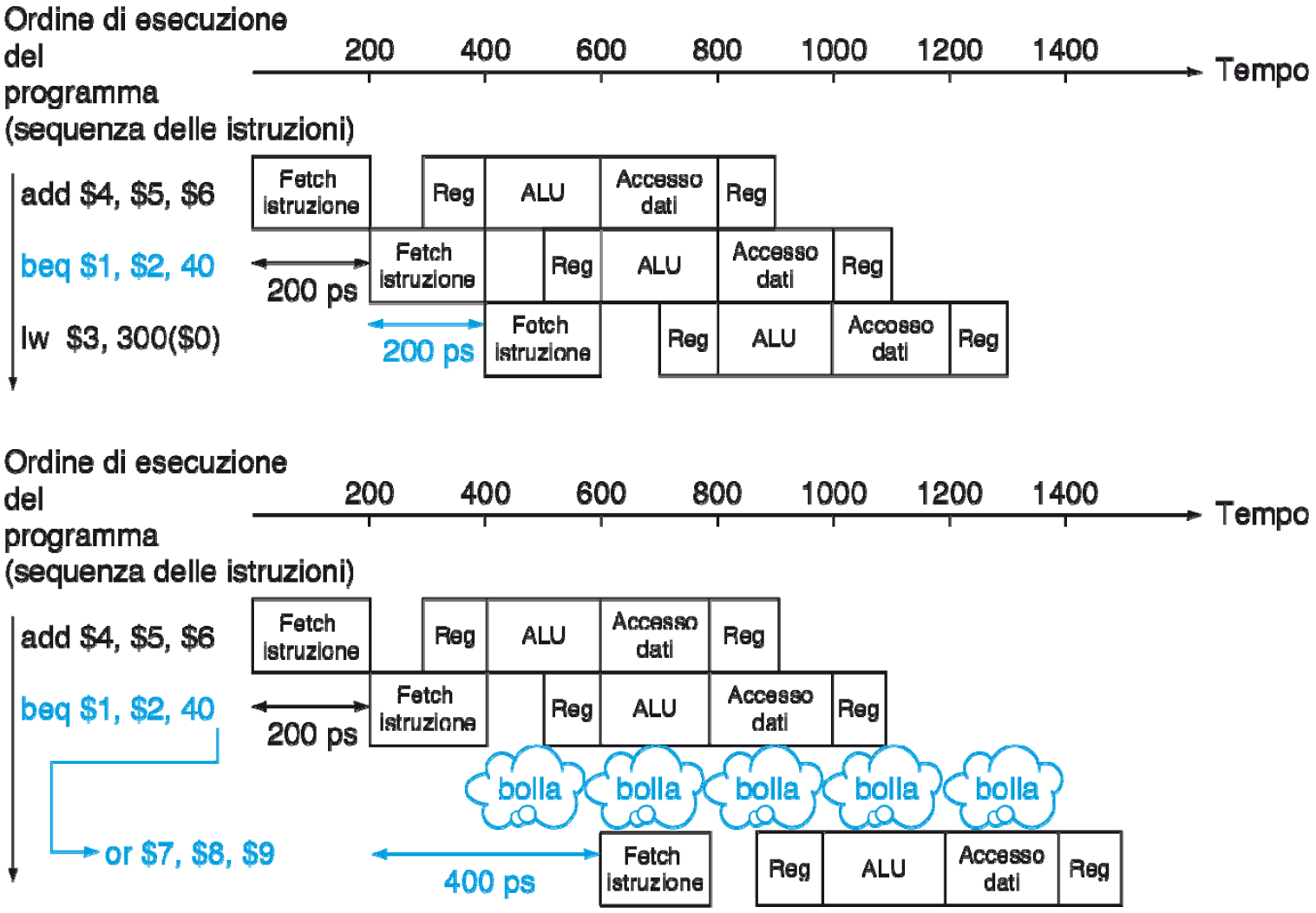
- esito confronto registri e calcolo indirizzo destinazione di salto disponibili alla fine della fase di ID

ma non basta ….

- se ***branch untaken*** la pipeline lavora "a pieno regime": esecuzione dell'istruzione in sequenza a Y+4
- se ***branch taken*** si deve inserire un ciclo di ritardo prima di eseguire l'istruzione destinazione di salto per avere il PC corretto (Z)

  se esito e indirizzo salto disponibili alla fine della fase di EX, allora due cicli di ritardo

Conflitto di controllo (*control hazard*): *beq (cont.)*



Problema del conflitto di dati

Abbiamo visto che se le istruzioni eseguite nella pipeline sono **dipendenti** tra loro possono nascere problemi dovuti a **conflitti di dati**

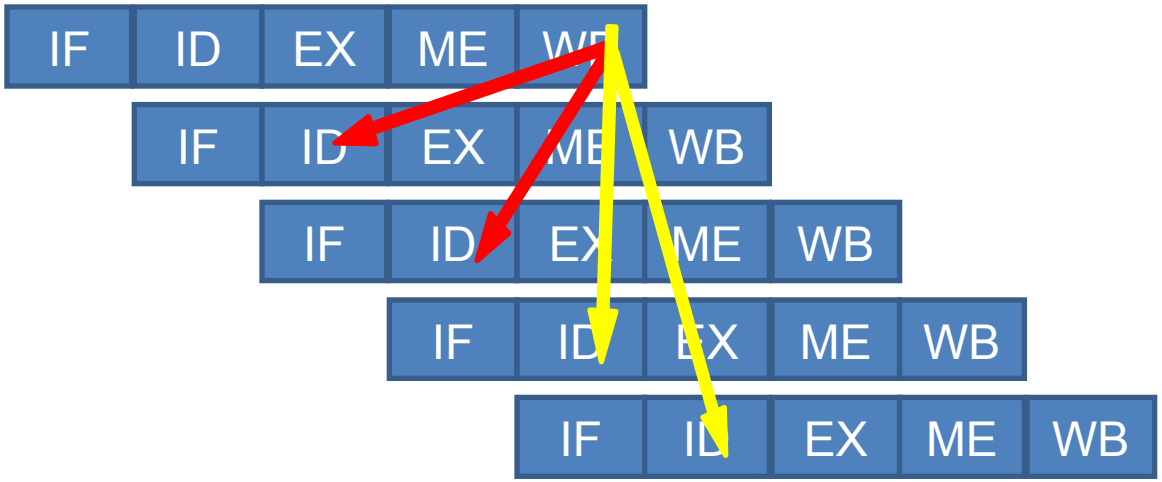
Esempio: le ultime quattro istruzioni dipendono dal risultato scritto nel registro **$2** dalla prima istruzione, ma solo le ultime due istruzioni accedono al valore corretto:

```
sub  $2, $1, $3   # Reg. $2 scritto dalla istruzione sub
and $12, $2, $5   # Il 1° operando($2) dipende dalla sub
or  $13, $6, $2   # Il 2° operando($2) dipende dalla sub
add $14, $2, $2   # 1° ($2) & 2° ($2) dipendono dalla sub
sw $15,100($2)    # Il reg. indice($2) dipende dalla sub
```

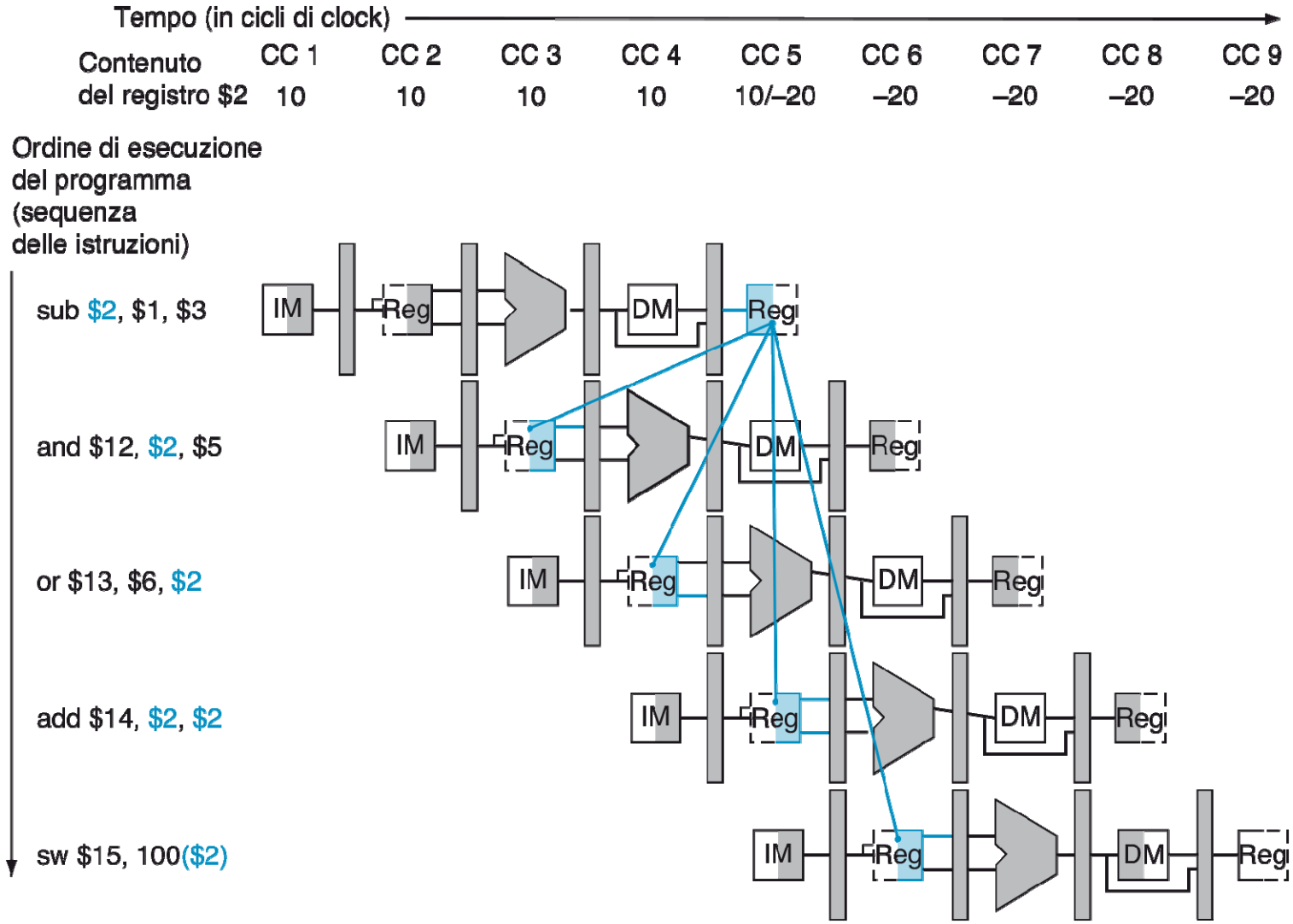
Dipendenze di dato e stadi

Le dipendenze di dato diventano ***conflitti*** se “vanno all’indietro nel tempo”

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```



Dipendenze di dato, risorse e registro coinvolto



Soluzioni al problema del conflitto di dati

Tecniche di compilazione del codice:

- Inserimento di istruzioni **nop** (*no operation).*
- *Scheduling* o **riordino delle istruzioni** in modo da impedire che istruzioni correlate siano troppo vicine.
  - Il compilatore cerca di inserire tra le istruzioni correlate (che presentano dei conflitti) delle istruzioni *indipendenti* dal risultato delle precedenti operazioni, facendo così scomparire tutti i conflitti.
  - Quando il compilatore non riesce a trovare istruzioni indipendenti deve inserire istruzioni **nop**.

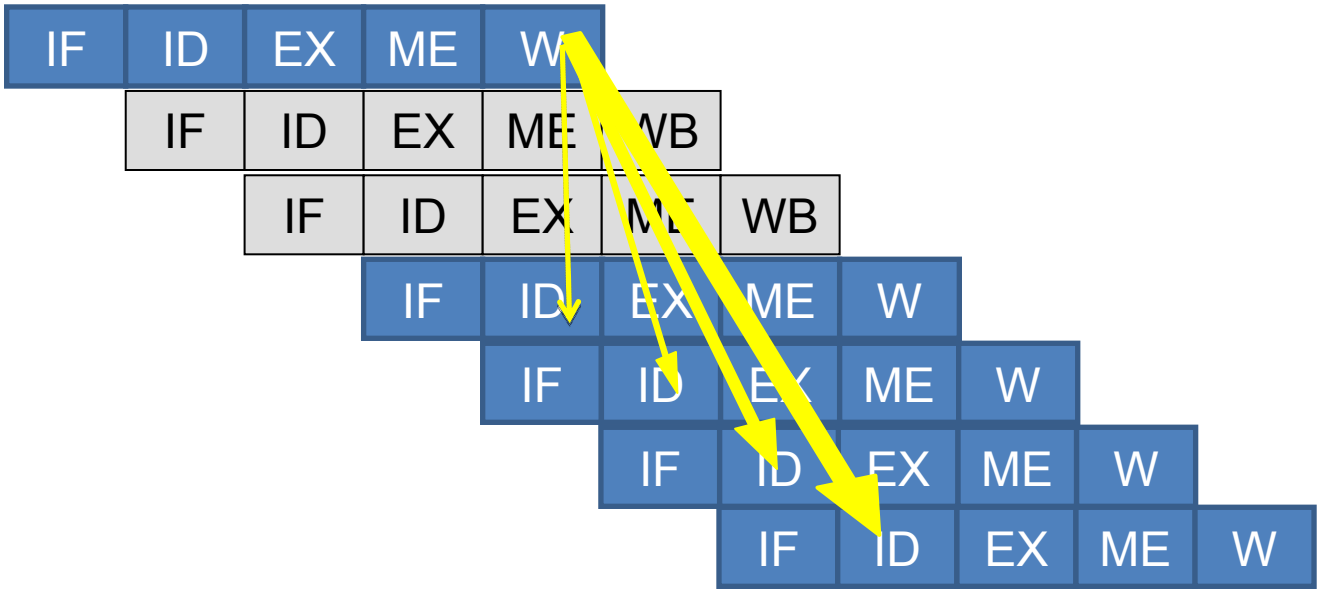
Tecniche di tipo hardware:

- Inserimento di *"bolle"* o **stalli** nella pipeline.
- **Propagazione** dei dati in avanti i dati *(forwarding o bypassing)*

Inserimento di nop: Esempio

Il compilatore deve inserire tra le istruzioni **sub** e **and** due istruzioni *nop*, facendo così scomparire tutti i conflitti.

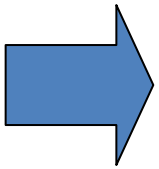
```
sub  $2, $1, $3
nop
nop
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```



Scheduling: Esempio

Esempio:

```
sub    $2, $1, $3
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15,100($2)
add    $4, $10, $11
and    $7, $8, $9
lw $16, 100($18)
lw $17, 200($19)
```



```
sub    $2, $1, $3
add    $4, $10, $11
and    $7, $8, $9
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15,100($2)
lw     $16, 100($18)
lw     $17, 200($19)
```

Inserimento di stalli: Esempio

Una possibile soluzione di tipo *hardware* consiste nell'inserire delle *"bolle"* nella pipeline, cioè *bloccare il flusso di ingresso di istruzioni* nella pipeline fino a quando il conflitto non è risolto. Lo stato in cui si trova la *CPU* quando le istruzioni in ingresso sono bloccate è indicato con il termine "*stallo*".

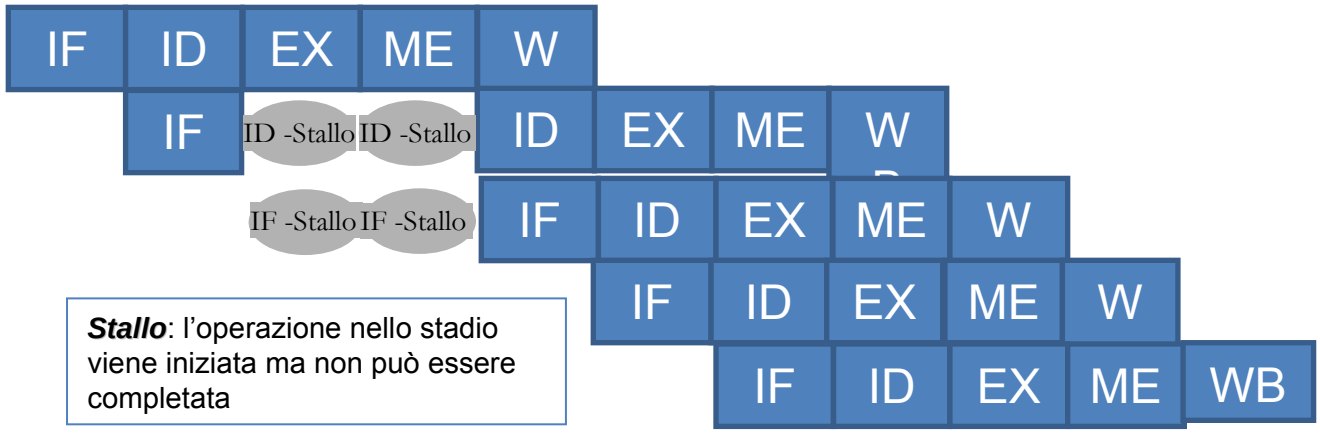
Nell'esempio significa inserire *2* bolle o stalli per fermare le istruzioni che seguono l'istruzione **sub** affinché possano essere letti i dati corretti.

Di conseguenza al «blocco»

- le istruzioni precedenti al blocco proseguono normalmente l'esecuzione nominale
- le istruzioni in cui è stato inserito lo stallo *in un particolare stadio* si comportano come «istruzioni in stallo»

Inserimento di stalli

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15,100($2)
```



Stallo: l'operazione nello stadio viene iniziata ma non può essere completata

- Lo stallo di una istruzione (in questo caso **and**) viene inserito nello **stadio ID** ***che rileva il conflitto*** (*vedremo come*) e che ***non può completare la scrittura del registro ID/EX*** con valori corretti associati all'istruzione (**and**) che ha generato lo stallo
- Lo ***stadio ID*** dovra' essere "rieseguito" per la **and** al termine del conflitto: ciò significa che i valori nel registro ***IF/ID*** (registro di ingresso allo stadio ID) devono essere "congelati" fino al termine del conflitto (cioè si deve bloccare la scrittura di IF/ID)
- Di conseguenza l'istruzione **or** di fatto **non può completare la fase *IF*** e quindi ***non può scrivere in IF/ID*** e per questo lo **stadio IF** viene messo in stallo contemporaneamente a **ID**. Il valore del **PC** (registro di ingresso a IF) di **or** rimane "congelato" fino al termine del conflitto (si deve bloccare la scrittura del PC)

Inserimento di stalli: comportamento degli stadi in stallo

Purché il contenuto dei registri *PC* e *IF/ID* vengano conservati, l'istruzione nello stadio *IF* continuerà ad essere letta utilizzando lo stesso *PC*, ed i registri nello stadio *ID* continueranno ad essere letti utilizzando la stessa istruzione nel registro di pipeline *IF/ID*.

L'istruzione **and** resta nel registro di pipeline *IF/ID* per *2* cicli di clock, mentre vengono generate *2* «bolle» separate nella pipeline.

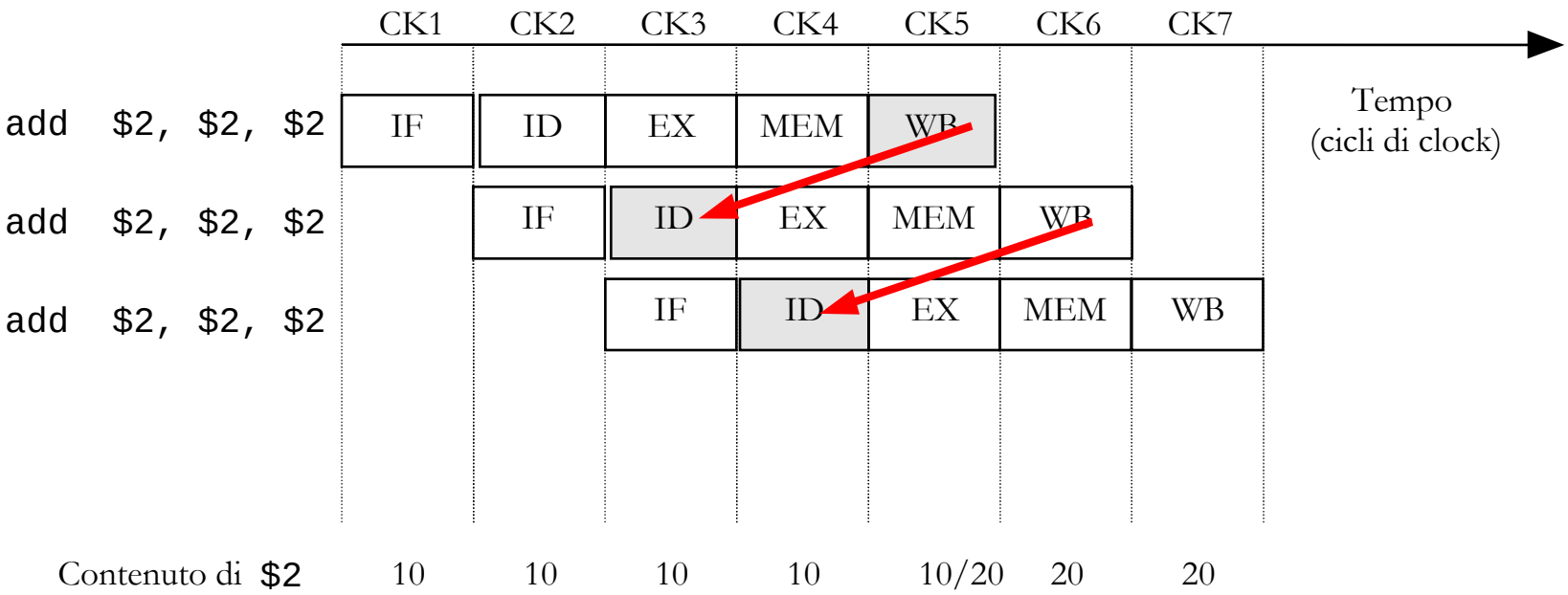
Che cosa vuol dire generare bolle? Significa definire anche il comportamento dei registri di interstadio ID/EX, EX/MEM e MEM/WB in conseguenza al blocco. Non banale ….. da capire!!

Azione possibile

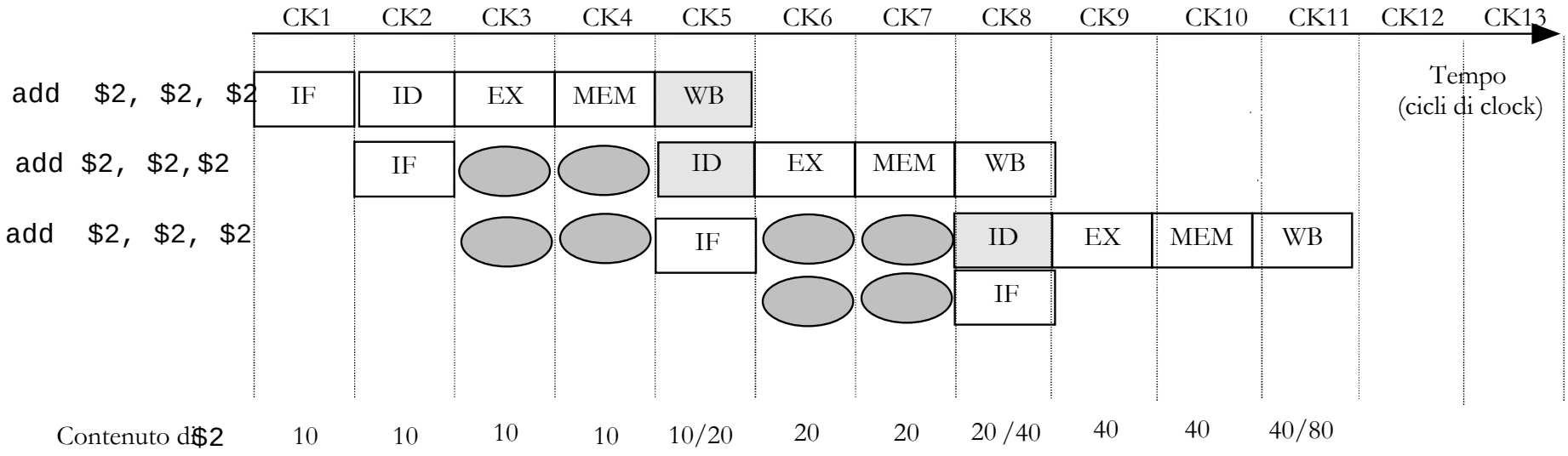
in **ciascun ciclo di stallo**, lo **stadio ID scriverà nella parte "segnali di controllo" del registro ID/EX solo valori a 0:** in questo modo nei cicli di clock seguenti gli stadi successivi a ID **di fatto** non lavorano, poichè i segnali di controllo a 0 si propagano finché non termina l'effetto dello stallo

Problema del conflitto di dati: un altro esempio

L1: add **$2**, $2, $2 # Reg. $2 scritto dall'istruz. L1
L2: add **$2, $2, $2** # 1° e 2°operando dipendono dalla L1
L3: add **$2, $2, $2** # 1° e 2°operando dipendono dalla L2

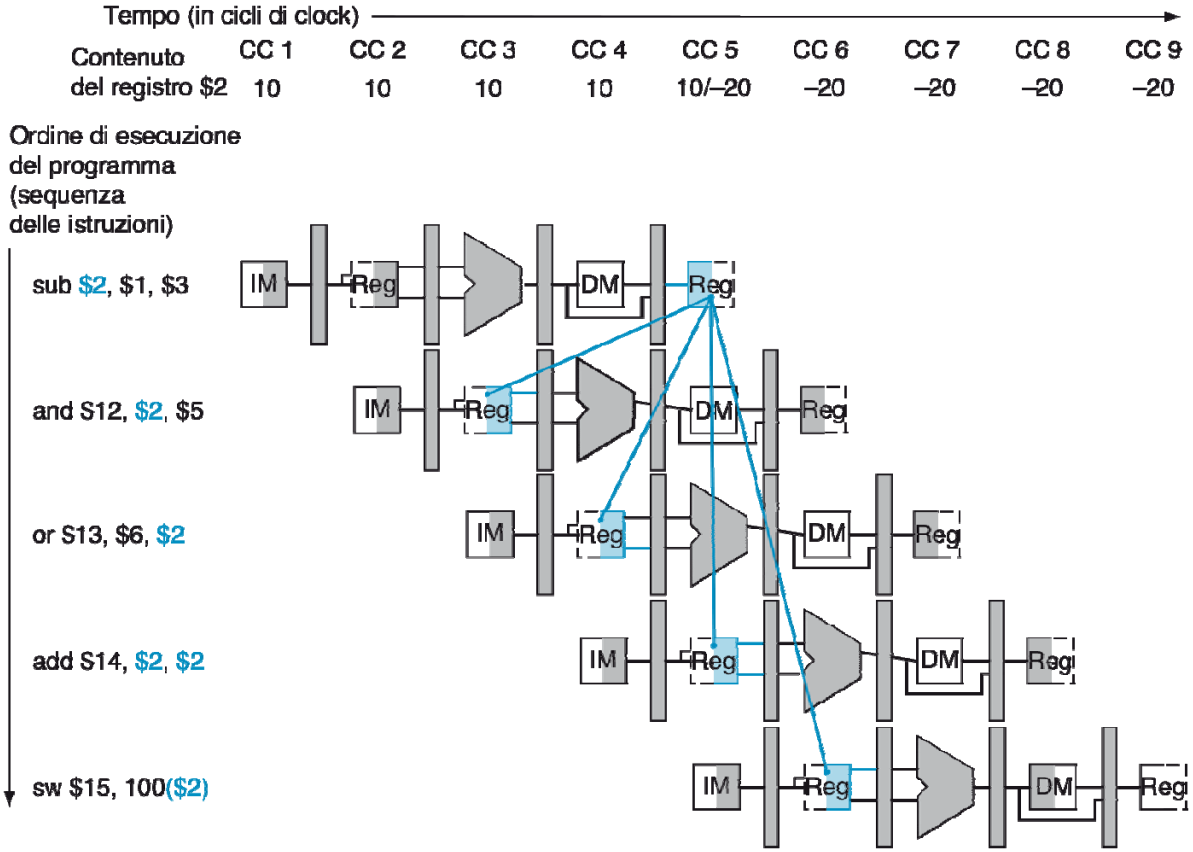


Problema del conflitto di dati: esecuzione corretta



Forwarding

Un'altra possibile soluzione *hardware* consiste nella **propagazione in avanti dei dati**



Il risultato dell'istruzione **sub** è disponibile alla fine del ciclo di clock 3 (stadio EX)

Questo succede per tutte le istruzioni di tipo R

and e **or** hanno bisogno del dato all'inizio dello stadio di EX, ossia nei cicli di clock 4 e 5 rispettivamente

si può eseguire il codice senza stalli propagando il dato non appena disponibile a qualsiasi unità che lo richieda prima che venga scritto nel registro destinazione

Forwarding

- Il valore richiesto dall'istruzione and, è già disponibile nel registro di pipeline *EX/MEM* dell'istruzione sub
- Analogamente al ciclo di clock successivo, l'ingresso dell'*ALU* per l'istruzione or si trova nel registro di pipeline *MEM/WB* dell'istruzione sub
- E' possibile fornire all'*ALU* gli ingressi richiesti dalle istruzioni and e or propagando in avanti i risultati che si trovano nei registri di pipeline, senza aspettare che siano scritti nel banco di registri.

Questa tecnica, che utilizza risultati temporanei invece di attendere che i registri siano scritti, si chiama **propagazione in avanti dei risultati (forwarding) o bypassing**

- Aggiungendo **multiplexer** agli ingressi dell'*ALU* in modo da prelevare gli ingressi da qualsiasi registro di pipeline e non solo dal registro *ID/EX*, la pipeline può procedere senza stalli anche in presenza di conflitti di dati.

Propagazione di dato

Quando un'istruzione nel suo stadio EX deve utilizzare un dato di un registro che non è ancora stato scritto da un'istruzione precedente nella sua fase di WB è necessario portare il dato all'ingresso corretto della ALU.

Coppie di condizioni che generano un conflitto di dato:

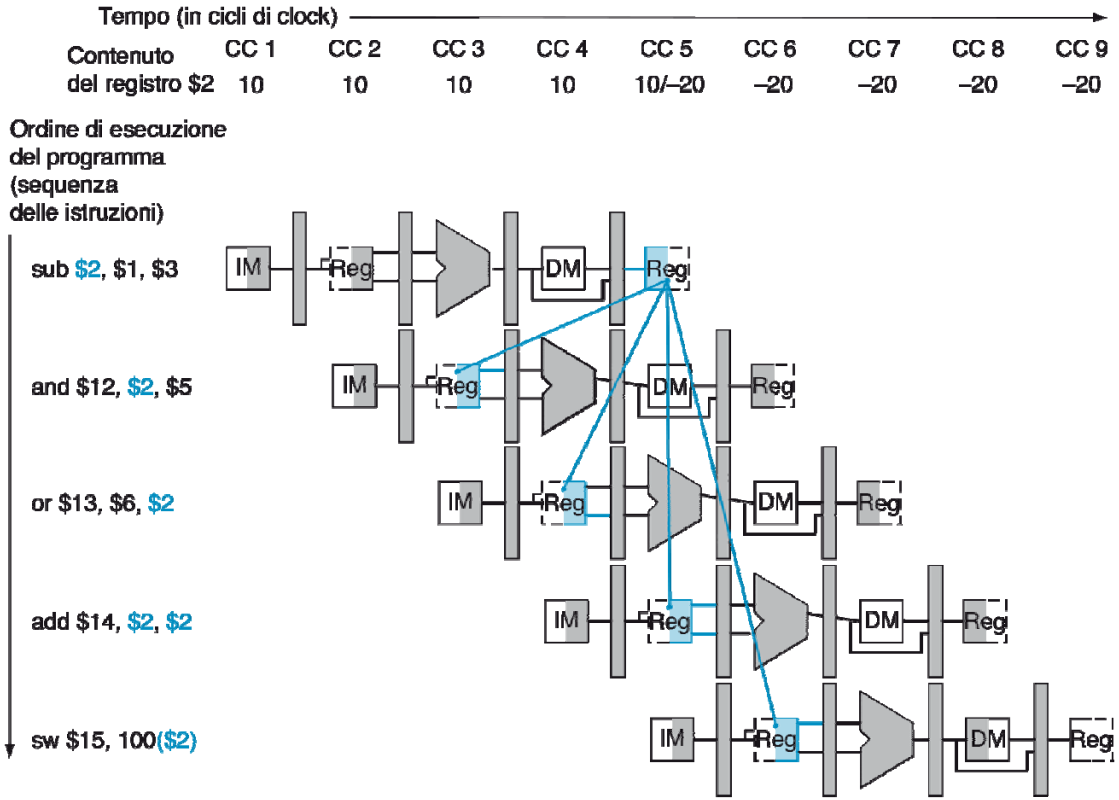
1a. EX/MEM.RegistroR = ID/EX.RegistroRs
1b. EX/MEM.RegistroR = ID/EX.RegistroRt

→ *hazard nello stadio EX*

2a. MEM/WB.RegistroR = ID/EX.RegistroRs
2b. MEM/WB.RegistroR = ID/EX.RegistroRt

→ *hazard nello stadio MEM*

Conflitti di dato: esempio



Conflitto sub – and: tipo 1a

EX/MEM.RegistroR = ID/EX.RegistroRs = $2 &&

EX/MEM.RegistroR ≠ 0

Conflitto sub – or: tipo 2b

MEM/WB.RegistroR = ID/EX.RegistroRt= $2 &&

MEM/WB.RegistroR ≠ 0

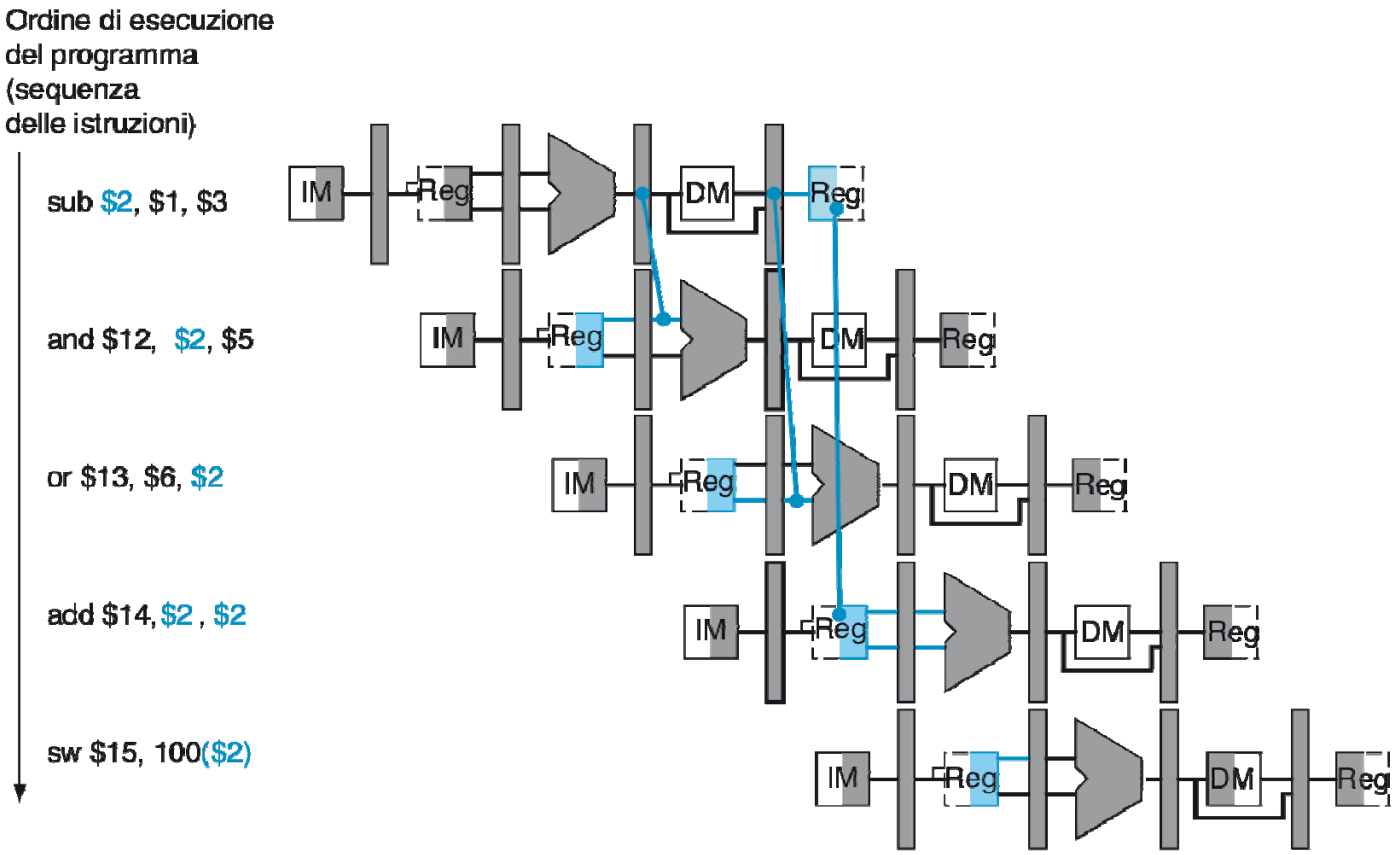
Implementazione

- prendere gli ingressi della ALU non solo dal registro ID/EX ma anche dagli altri registri di pipeline EX/MEM e MEM/WB per propagare i dati corretti
- aggiungere dei mux agli ingressi della ALU e i corrispondenti segnali di controllo
- la propagazione avviene solo se il **segnale RegWrite è asserito nello stadio che propaga** (da EX/MEM in avanti)

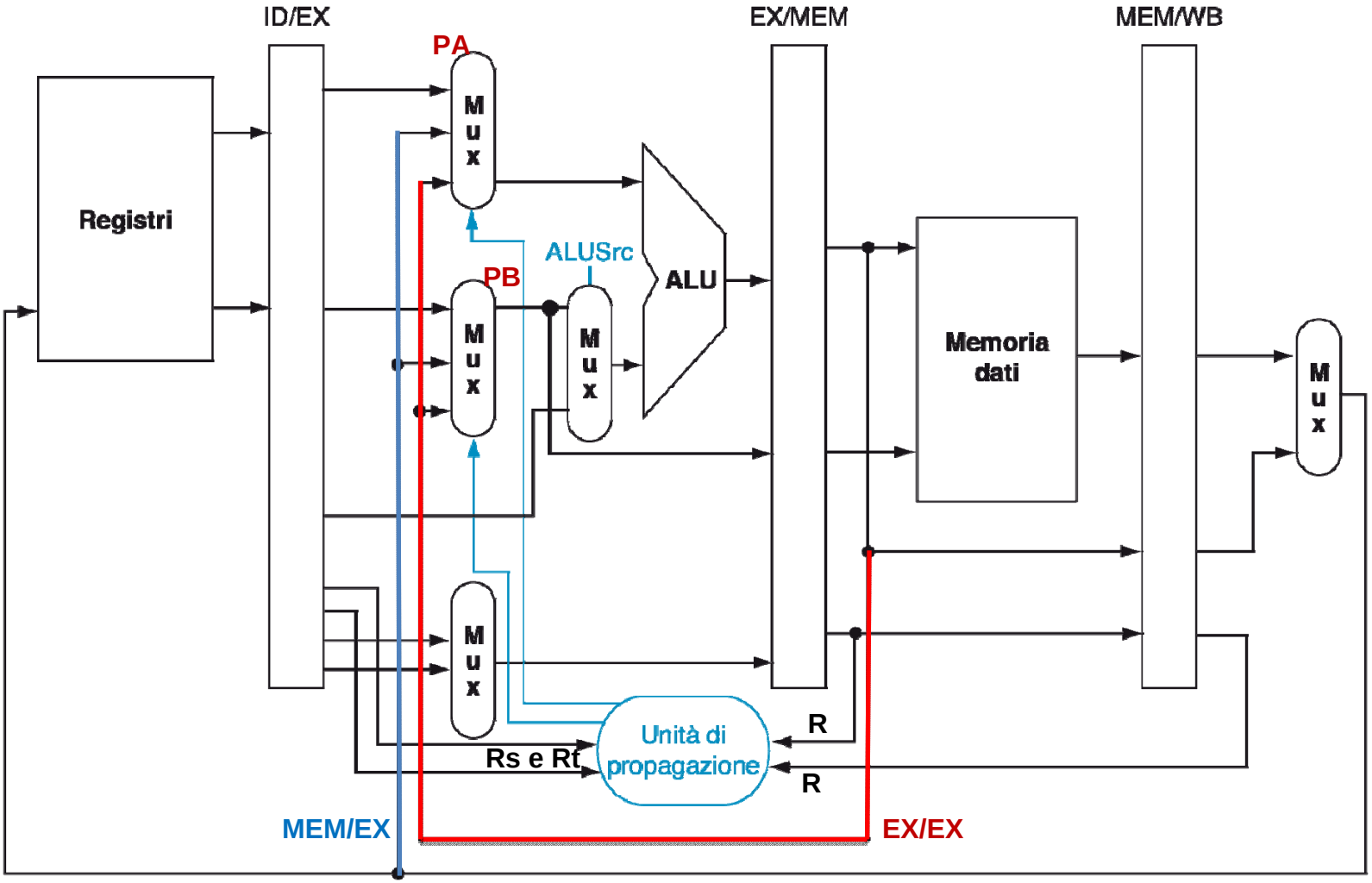
Forwarding: esempio

Tempo (in cicli di clock) →

	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



Pipeline con hardware di propagazione per istruzioni di tipo R



Condizioni per rilevare i conflitti e *propagare* dallo stadio EX e dallo stadio MEM

If (EX/MEM.RegWrite and EX/MEM.RegistroR ≠ 0) and (EX/MEM.RegistroR = ID/EX.RegistroRs)) Propaga = 10 — ***MUX PA***

If (EX/MEM.RegWrite and EX/MEM.RegistroR ≠ 0) and (EX/MEM.RegistroR = ID/EX.RegistroRt)) Propaga = 10 — ***MUX PB***

If (MEM/WB.RegWrite and MEM/WB.RegistroR ≠ 0) and (MEM/WB.RegistroR = ID/EX.RegistroRs)) Propaga = 01 — ***MUX PA***

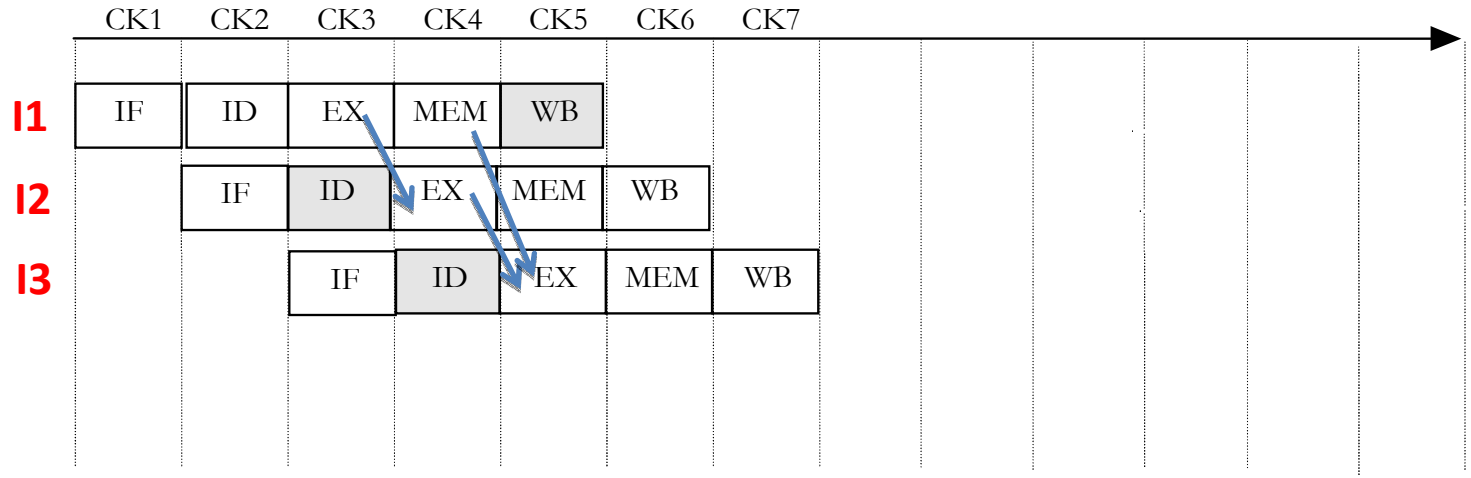
If (MEM/WB.RegWrite and MEM/WB.RegistroR ≠ 0) and (MEM/WB.RegistroR = ID/EX.RegistroRt)) Propaga = 01 — ***MUX PB***

Segnali di controllo del multiplexer di propagazione

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.

Ancora sulla propagazione: il problema del valore “più recente”

I1 add $1, $1, $2
I2 add $1, $1, $3
I3 add $1, $1, $4



a CK3 la condizione sotto è verificata e si propaga I1 per I2

If(EX/MEM.RegWrite and EX/MEM.RegistroR ≠ 0) and
(EX/MEM.RegistroR = ID/EX.RegistroRt)) Propaga = 10

I1 per I2

a CK4 sono verificate le due condizioni ma si deve propagare solo ***I2 per I3*** (valore più recente)

If (MEM/WB.RegWrite and MEM/WB.RegistroR ≠ 0) and
(MEM/WB.RegistroR = ID/EX.RegistroRt)) Propaga = 01

I1 per I3

If(EX/MEM.RegWrite and EX/MEM.RegistroR ≠ 0) and
(EX/MEM.RegistroR = ID/EX.RegistroRt)) Propaga = 10

I2 per I3

Ancora sulla propagazione: propagazione dallo stadio MEM

If (MEM/WB.RegWrite and (MEM/WB.RegistroR ≠ 0)

and not ((EX/MEM.RegWrite and (EX/MEM.RegistroR ≠ 0)) and
(EX/MEM.RegistroR = ID/EX.RegistroRs))

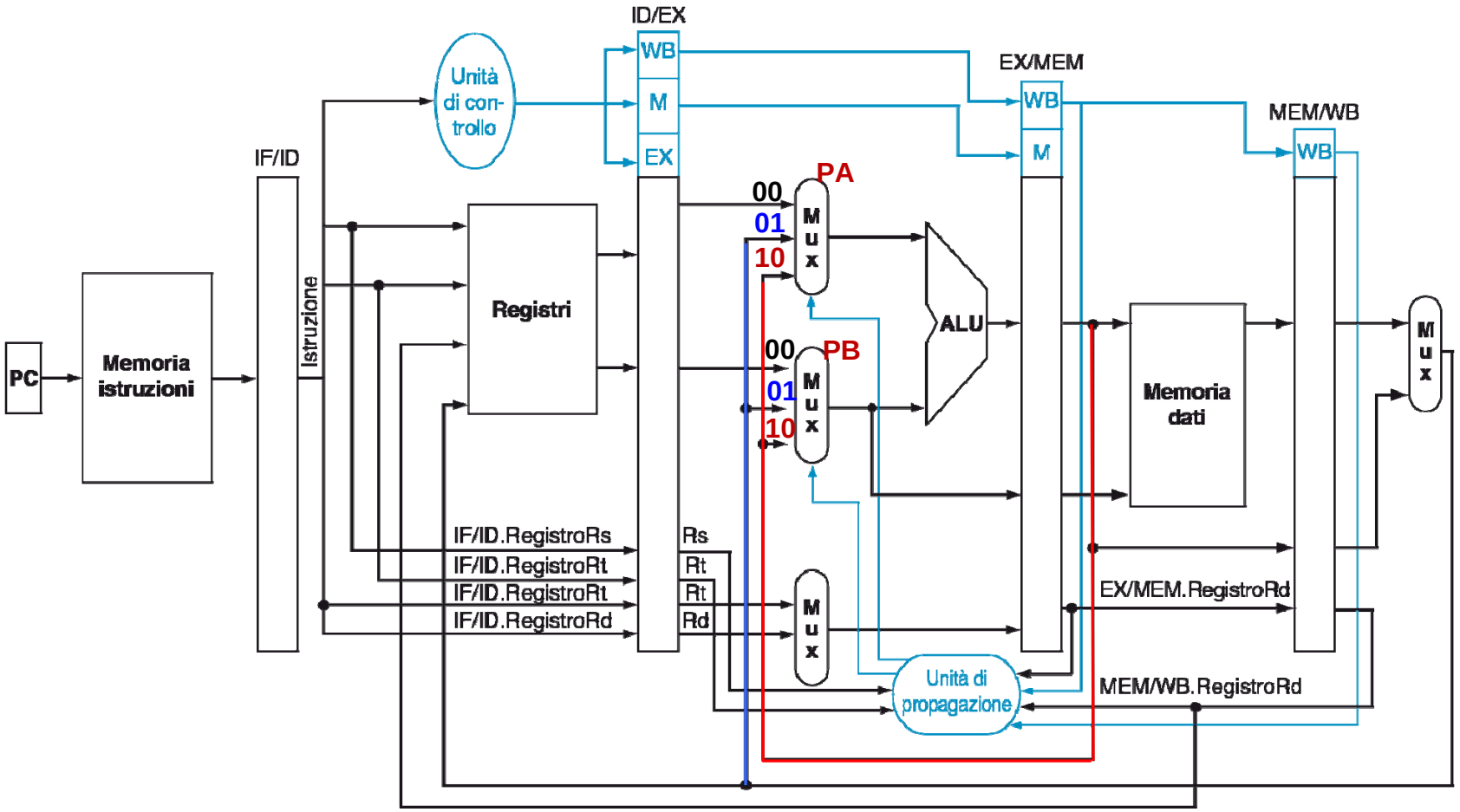
and (MEM/WB.RegistroR = ID/EX.RegistroRs)) Propaga = 01

If (MEM/WB.RegWrite and (MEM/WB.RegistroR ≠ 0)

and not ((EX/MEM.RegWrite and (EX/MEM.RegistroR ≠ 0)) and
(EX/MEM.RegistroR = ID/EX.RegistroRt))

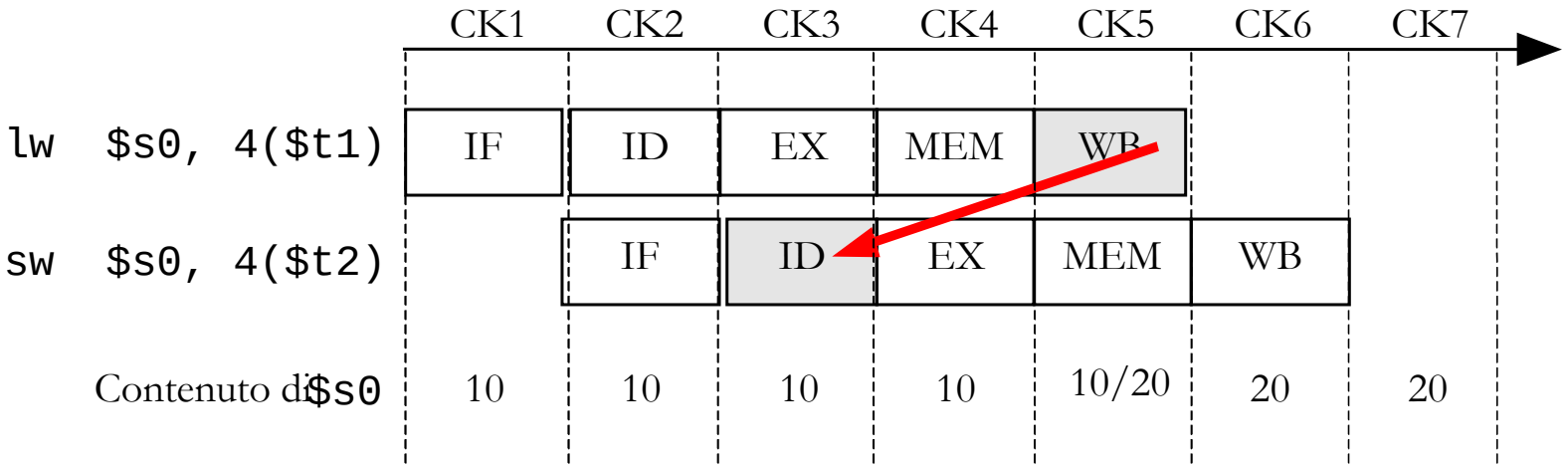
and (MEM/WB.RegistroR = ID/EX.RegistroRt)) Propaga = 01

Pipeline con unità di risoluzione dei conflitti di dato tramite propagazione



Problema del conflitto di dati: load/store

```
L1: lw $s0, 4($t1) # $s0 <- M [4 + $t1]
L2: sw $s0, 4($t2) # M[4 + $t2] <- $s0
```

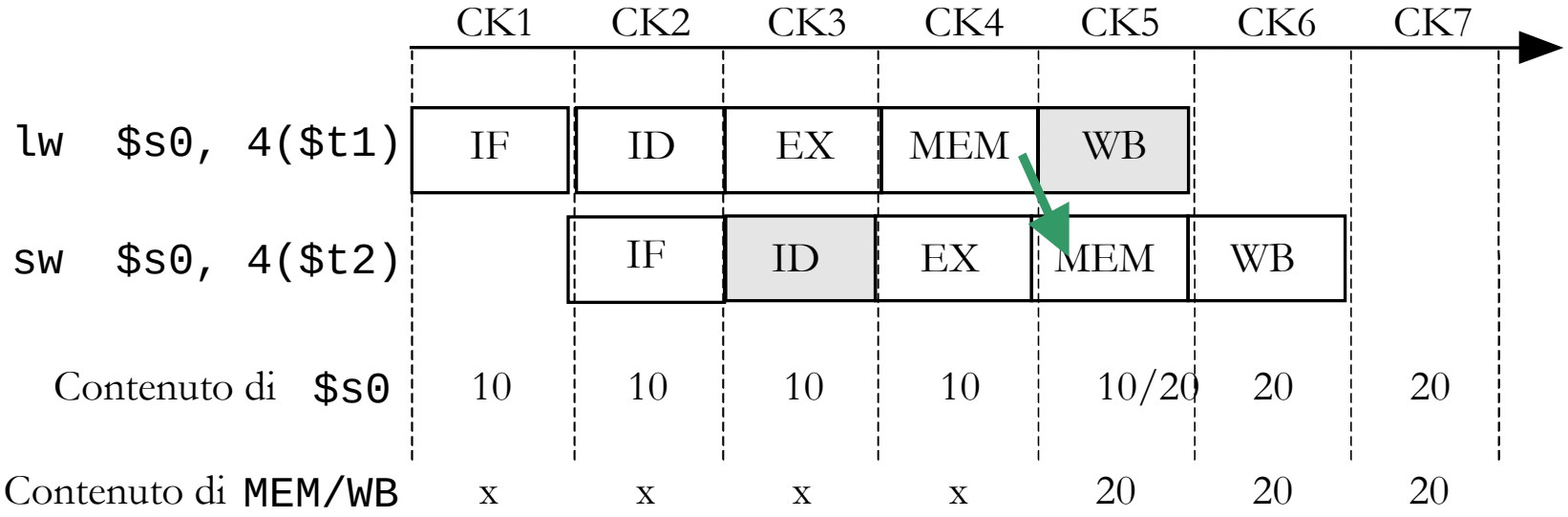


- Senza forwarding : Necessari 2 stalli

Problema del conflitto di dati: load/store

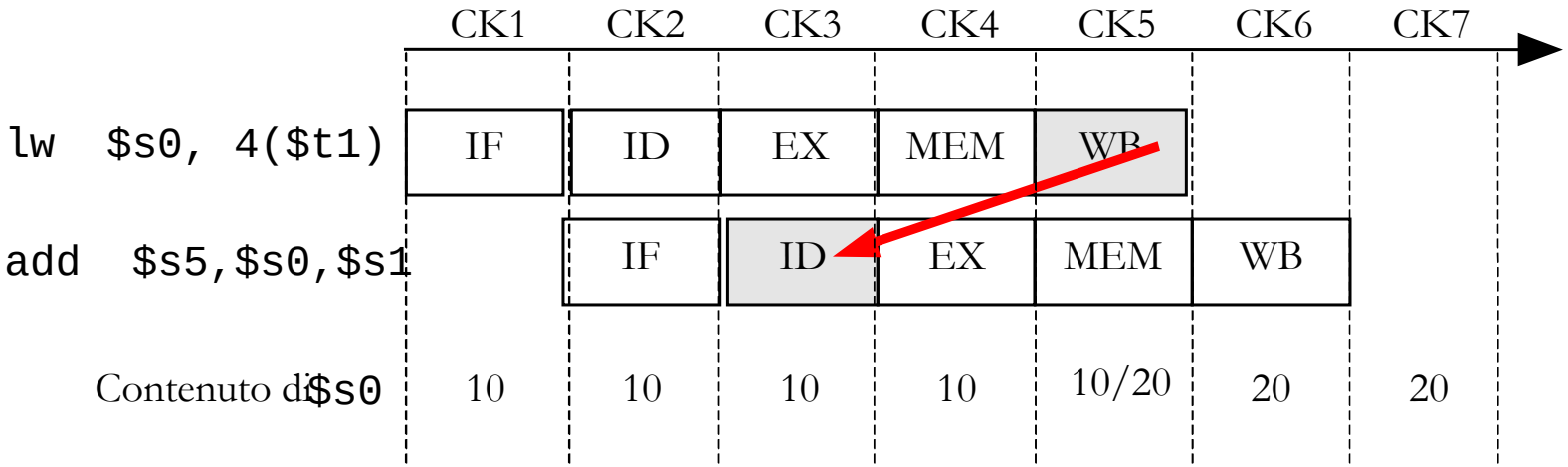
Con forwarding: Nessuno stallo, perché un'istruzione di load ha sempre il dato già «calcolato» nello stadio MEM

Devo usare un percorso di forwarding (**MEM/MEM**) per portare il risultato della *load* (*dato letto*) che è in MEM/WB all'ingresso *dato da scrivere* della memoria per effettuare la *store* e quindi devo aggiungere un **multiplexer** per selezionare il corretto *dato da scrivere*



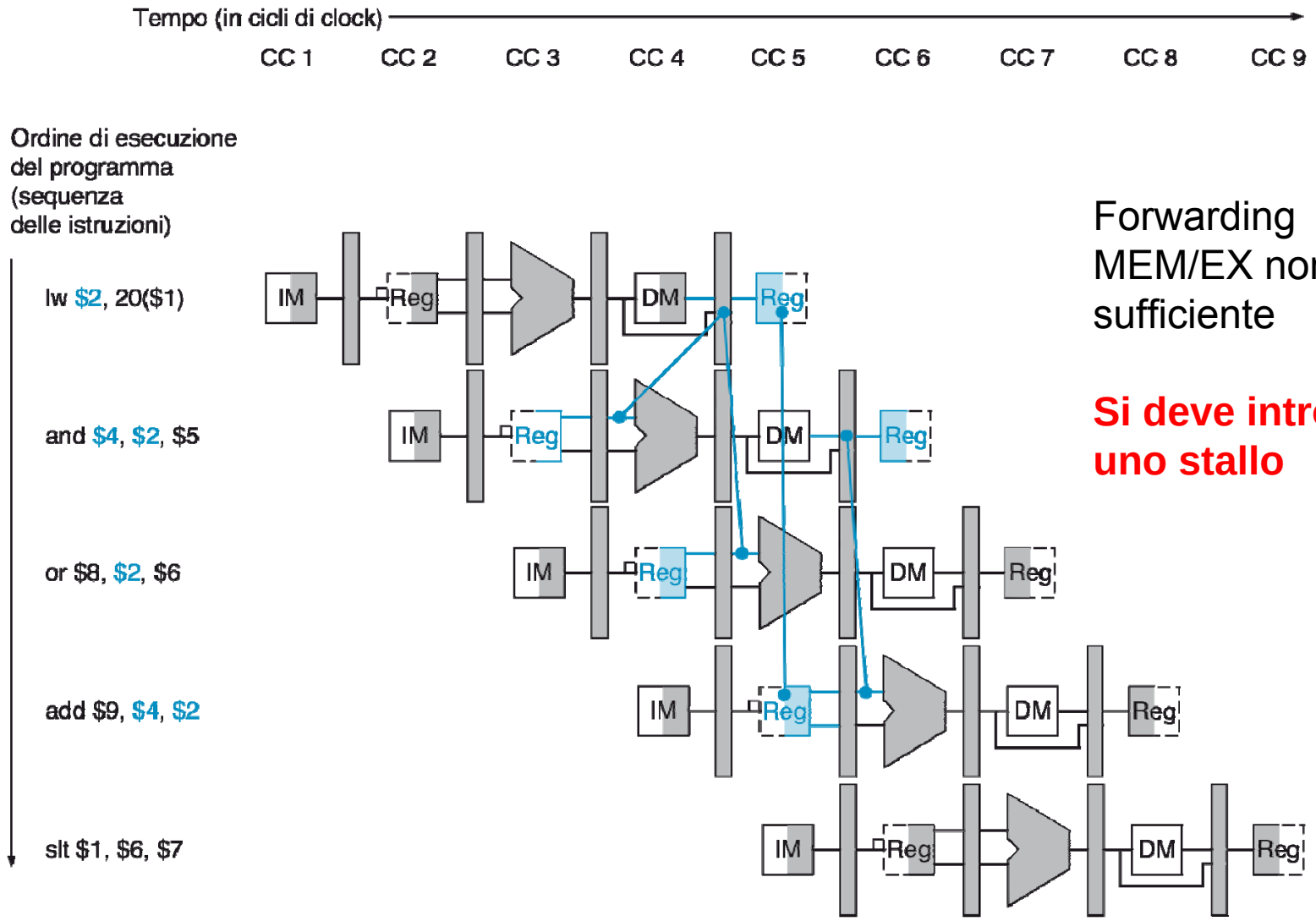
Hazard sui dati e stallo: esempio

```
L1: lw     $s0, 4($t1)   # $s0 <- M [4 + $t1]
L2: add    $s5, $s0, $s1 # Il 1° operando dipende dalla L1
```



- Senza forwarding : Necessari 2 stalli

Hazard sui dati e stallo: load/use



Forwarding MEM/EX non è sufficiente

Si deve introdurre uno stallo

Conflitto di dato *load/use*

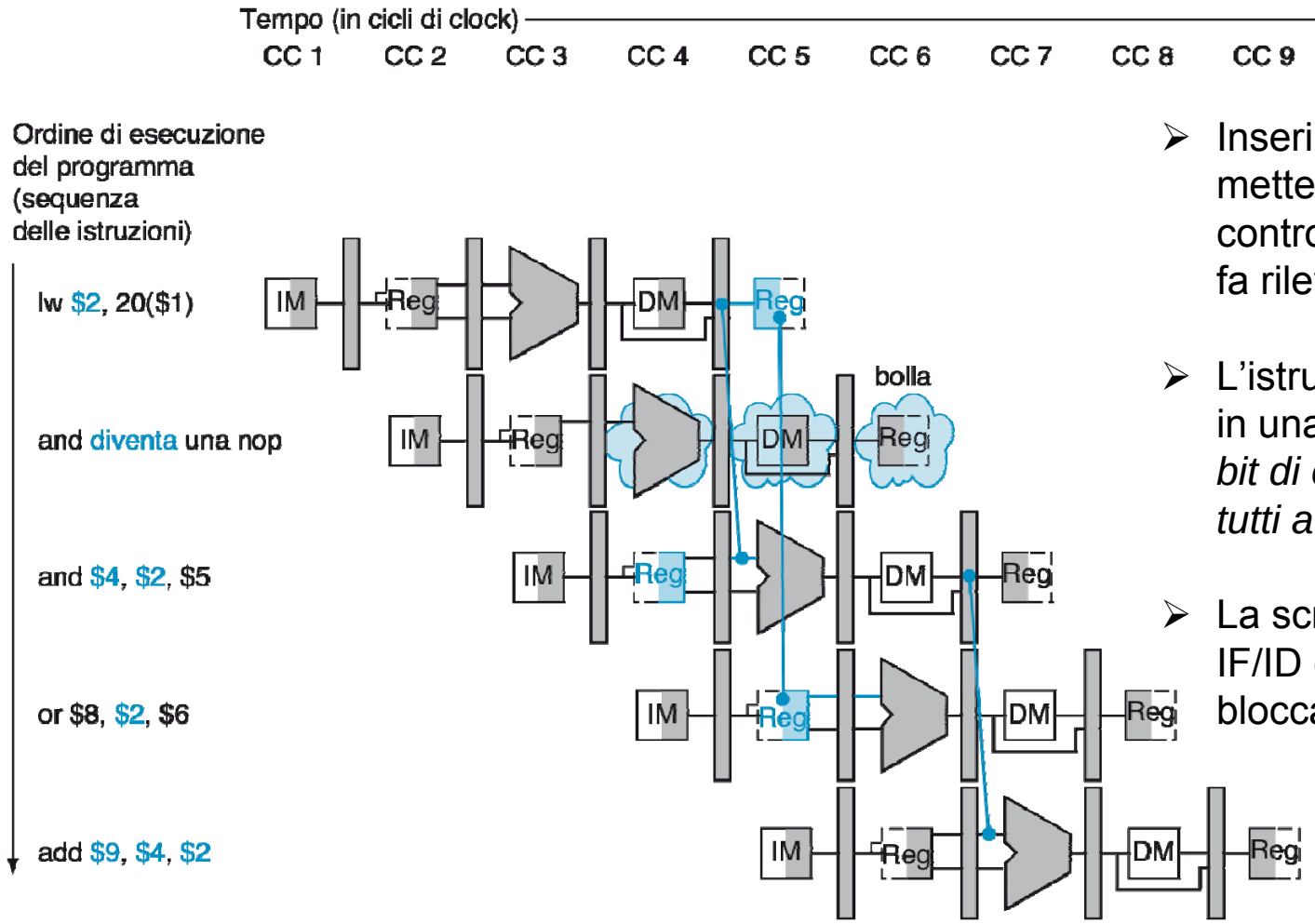
In aggiunta all'unità di propagazione è necessaria una unità di rilevamento dei conflitti che durante lo stadio di ID possa inserire uno stallo tra la lettura del dato e il suo utilizzo

Il conflitto viene rilevato utilizzando ID/EX di ***load*** e IF/ID di ***use***

If (ID/EX.MemRead and ((ID/EX.RegistroRt = IF/ID.RegistroRs)
or (ID/EX.RegistroRt = IF/ID.RegistroRt)))

- Metti in stallo la pipeline per un ciclo di clock
- Successivamente la dipendenza di dato è gestita dalla propagazione da MEM/WB a ID/EX (MEM/EX)

Modalità di inserimento degli stalli



- Inserire uno stallo significa mettere a 0 tutti i segnali di controllo dell'istruzione che fa rilevare il conflitto
- L'istruzione si «trasforma» in una *nop* (in cui almeno i *bit di controllo sono tutti tutti a 0)*
- La scrittura del registro IF/ID e del PC viene bloccata

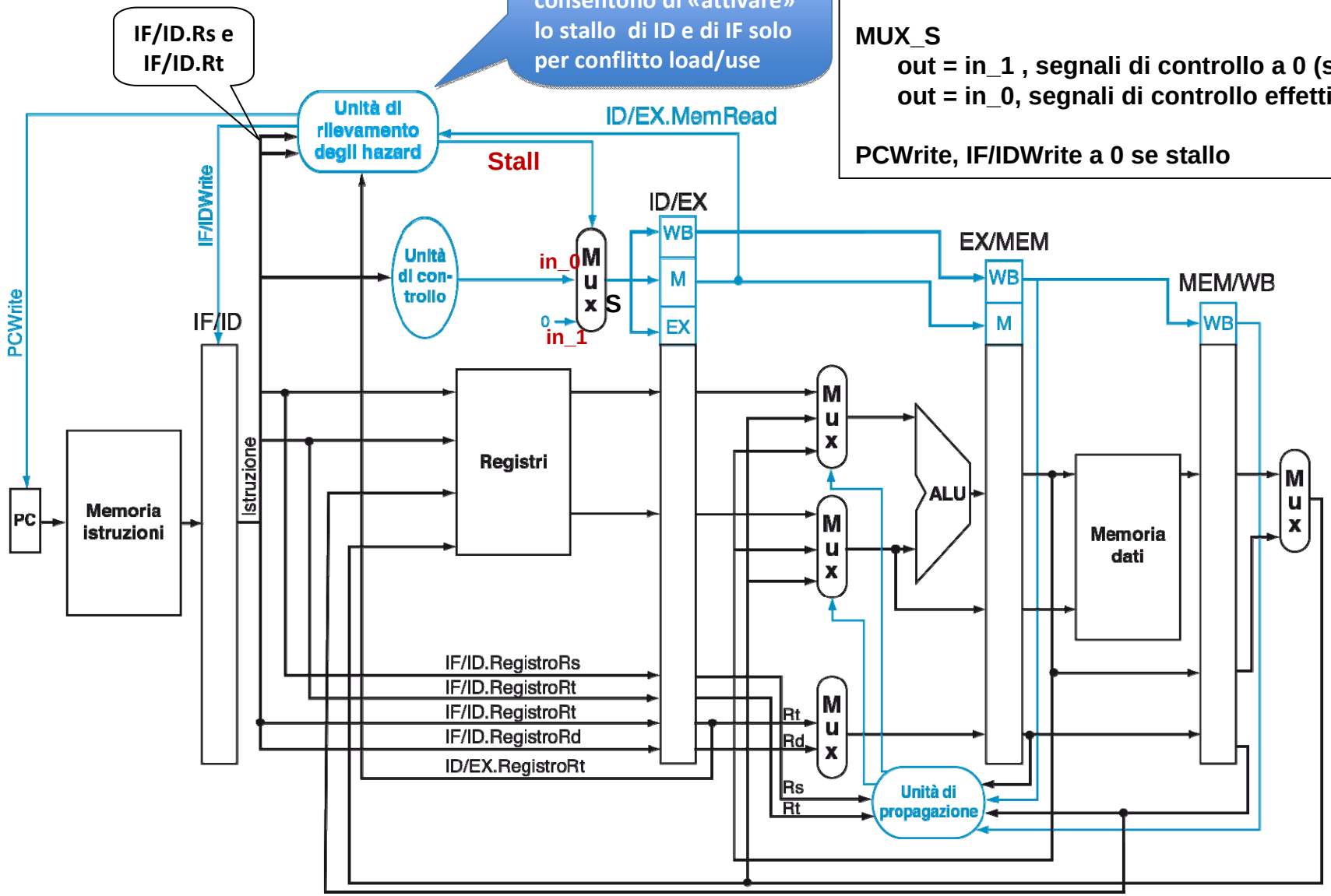
Controllo della pipeline

I segnali di ingresso all'unità di rilevamento consentono di «attivare» lo stallo di ID e di IF solo per conflitto load/use

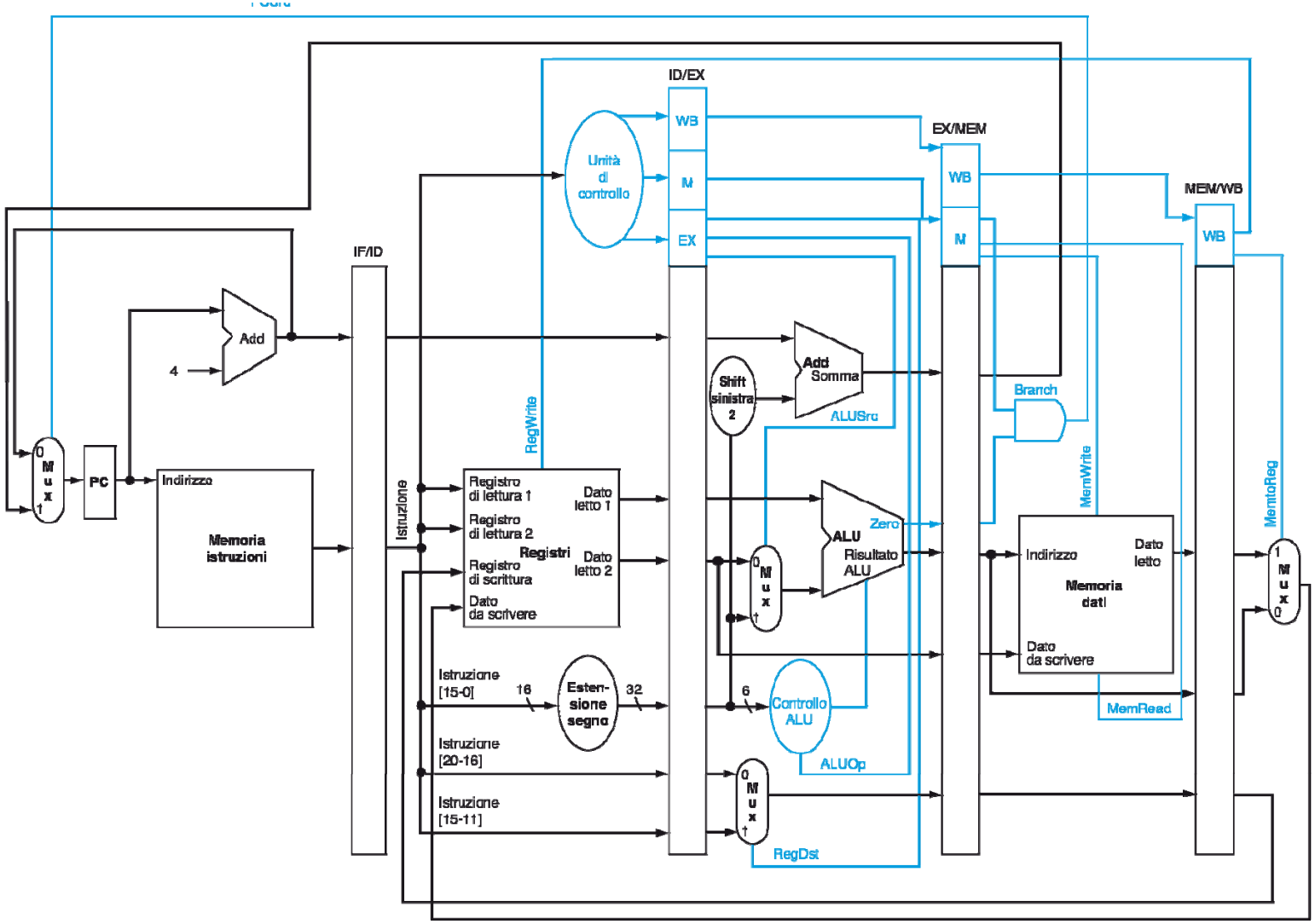
Stall a 1 se necessario stallo, 0 altrimenti

MUX_S
- **out = in_1 , segnali di controllo a 0 (stallo)**
- **out = in_0, segnali di controllo effettivi**

PCWrite, IF/IDWrite a 0 se stallo



Problema del conflitto di controllo



Problema del conflitto di controllo

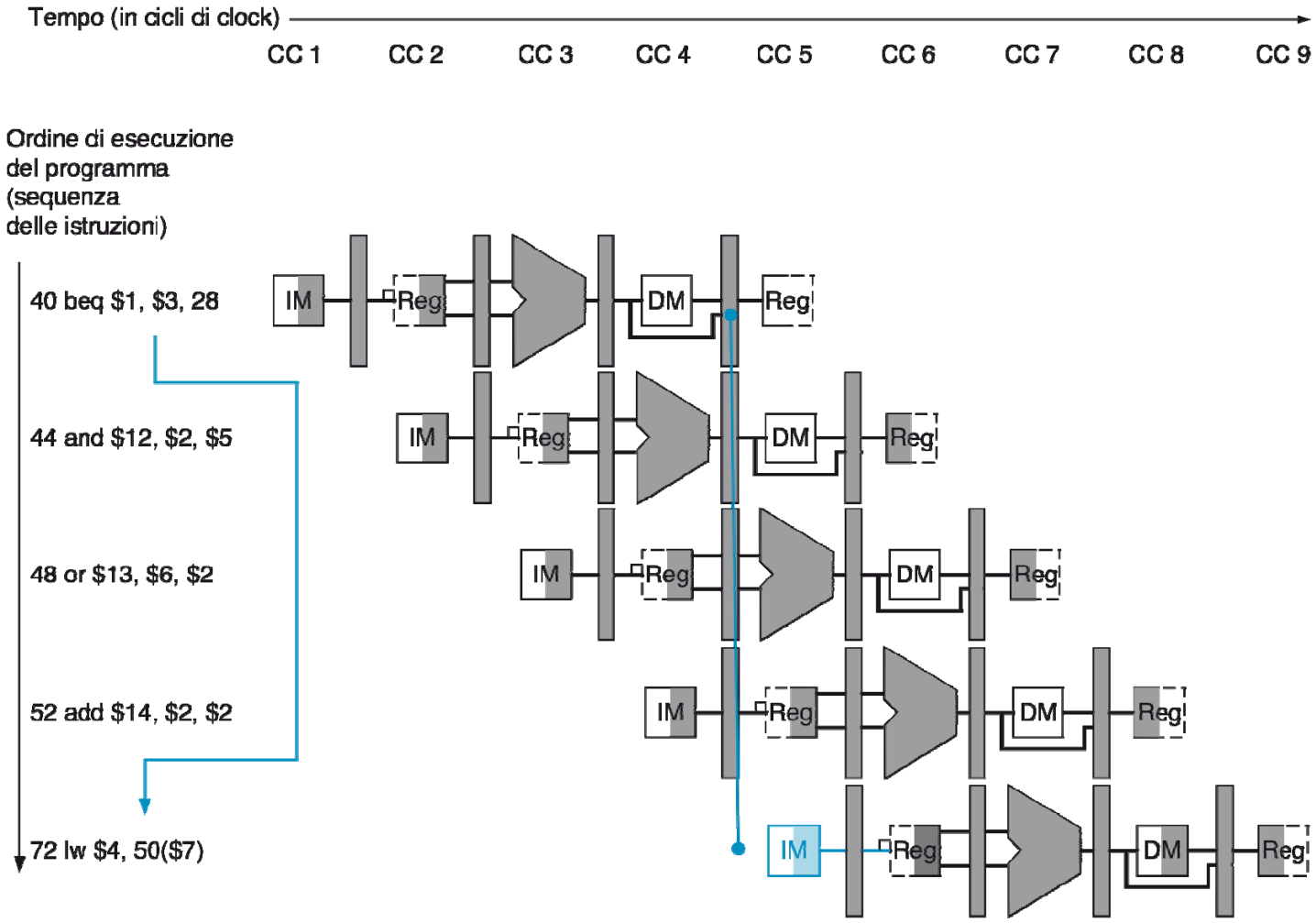
Per alimentare la pipeline si deve prelevare un'istruzione ad ogni ciclo di clock, però la decisione relativa al salto condizionato non viene presa fino allo stadio *MEM*.

La logica di controllo del salto viene determinata nello stadio *MEM* e - sul fronte di salita del ciclo corrispondente - il PC viene scritto con l'indirizzo destinazione di salto

Questo ritardo nel determinare l'istruzione corretta da prelevare viene chiamato **conflitto di controllo o conflitto di salto condizionato.**

Questi conflitti sono statisticamente meno frequenti dei conflitti sui dati.

Il problema dei salti condizionati



Soluzione al conflitto di controllo - 1

- Pipeline standard, non utilizzo di predizione: per garantire il corretto funzionamento di branch taken e untaken è possibile

  1. via software (compilatore) inserire 3 nop dopo ogni istruzione di salto condizionato
  2. via hardware bloccare la scrittura del PC e di IF/ID per 3 cicli (equivalenti a 3 stalli dello stadio IF) dopo una istruzione di salto condizionato

  le prestazioni diminuiscono ma siamo certi che il comportamento della pipeline è corretto per entrambi gli esiti della condizione

- Pipeline standard utilizzo della ***predizione*** (ha senso solo con soluzioni hardware)

  approccio più semplice: predire sempre che il salto venga **non eseguito** (untaken branch)

  - quando si verifica che il salto deve essere eseguito (stadio MEM) bisogna ***scartare*** le **tre** istruzioni già nella pipeline (devono diventare l'equivalente di 3 nop o di 3 stalli)

Soluzione al conflitto di controllo - 2

Ridurre i ritardi associati ai salti condizionati, per avere **una** sola istruzione da scartare (un solo stallo da inserire o una sola nop da inserire): ***pipeline ottimizzata***

- spostare la decisione del salto da MEM a uno stadio precedente
  - Richiede di anticipare il *calcolo dell'indirizzo* di destinazione del salto e la *valutazione del confronto* sulla base del quale si decide se effettuare o meno il salto

- Pipeline ***ottimizzata*** e utilizzo della ***predizione***

  predire sempre che il salto venga **non eseguito** (untaken branch)

  - Si genera stallo quando il salto deve essere eseguito e bisogna scartare **una** istruzioni già nella pipeline (che ha appena terminato lo stadio IF e scritto in IF/ID) ponendo almeno I suoi segnali di controllo a 0

Soluzione al conflitto di controllo - 3

Anticipazione del ***calcolo di destinazione*** del salto:

•PC e campo immediato sono disponibili nel registro IF/ID della pipeline una volta letta l'istruzione di salto

➔ sufficiente spostare il sommatore che calcola l'indirizzo del salto dallo stadio EX allo stadio ID

Anticipazione della ***decisione sul salto***:

•necessario confrontare il contenuto dei due registri letti nello stadio ID (confrontatore)

XOR bit a bit dei due valori: se tutti 0 i due valore sono uguali, quindi basta mettere un NOR sulle uscite e se uguale a 1 la condizione **beq** è verificata

Si noti che è necessario propagare l'esito del confronto allo stadio EX (scrittura dell'esito in ID/EX) per poter generare il valore corretto di ***PCsrc*** tramite l'uso del segnale di controllo ***Branch*** memorizzato anch'esso in ID/EX.

La ***logica di controllo del salto*** viene spostata di conseguenza dallo stadio MEM allo stadio EX

Possibili problemi

Necessità di propagazione

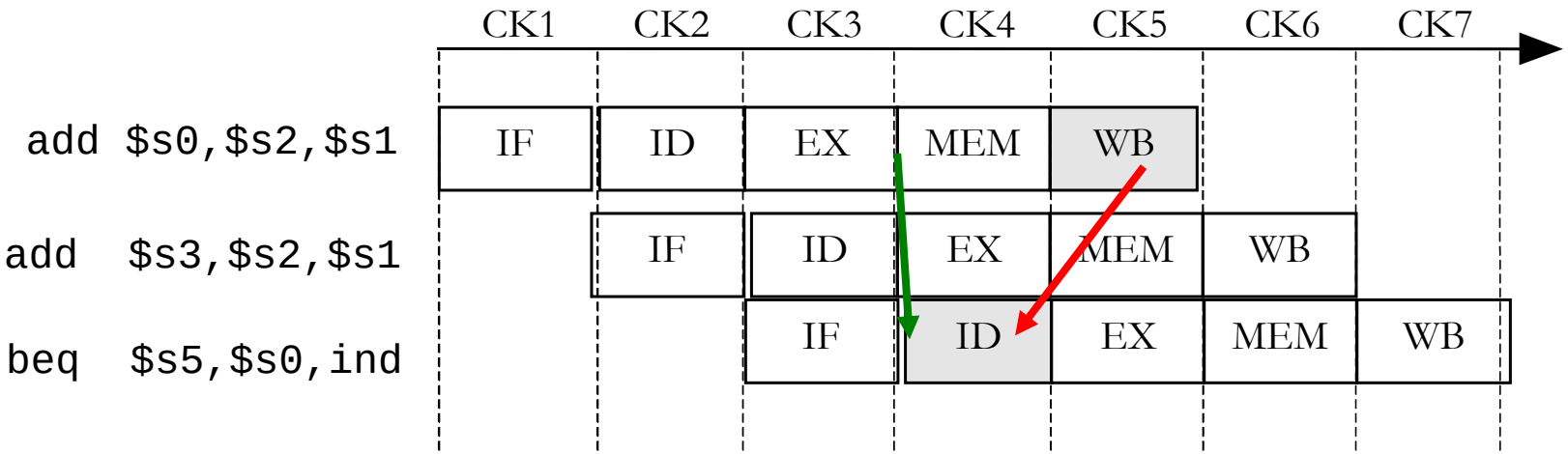
Conflitto sui dati

Anticipazione del confronto dei dati per il salto condizionato

- Per limitare il numero di stalli introdotti nello stadio ID dell'istruzione di salto è necessario prevedere ***percorsi di propagazione*** dei dati nel caso in cui gli operandi su cui eseguire il confronto si possano prelevare dai registri EX/MEM o MEM/WB (propagazione ***EX/ID*** e propagazione ***MEM/ID***). I percorsi e relativi multiplexer non sono mostrati nelle figure

- Se uno degli operandi con cui fare il confronto è calcolato dall'istruzione ***immediatamente precedente*** c'è conflitto sui dati ***non risolubile con percorsi di propagazione*** e quindi è necessario introdurre ***stalli*** per l'istruzione di salto:
  - 1 ciclo di stallo se l'operando e' calcolato nello stadio di EX (istruzione di tipo R)
  - 2 cicli di stallo se l'operando e' il risultato di una load e quindi disponibile al termine dello stadio MEM

Propagazione per anticipo del confronto: un esempio



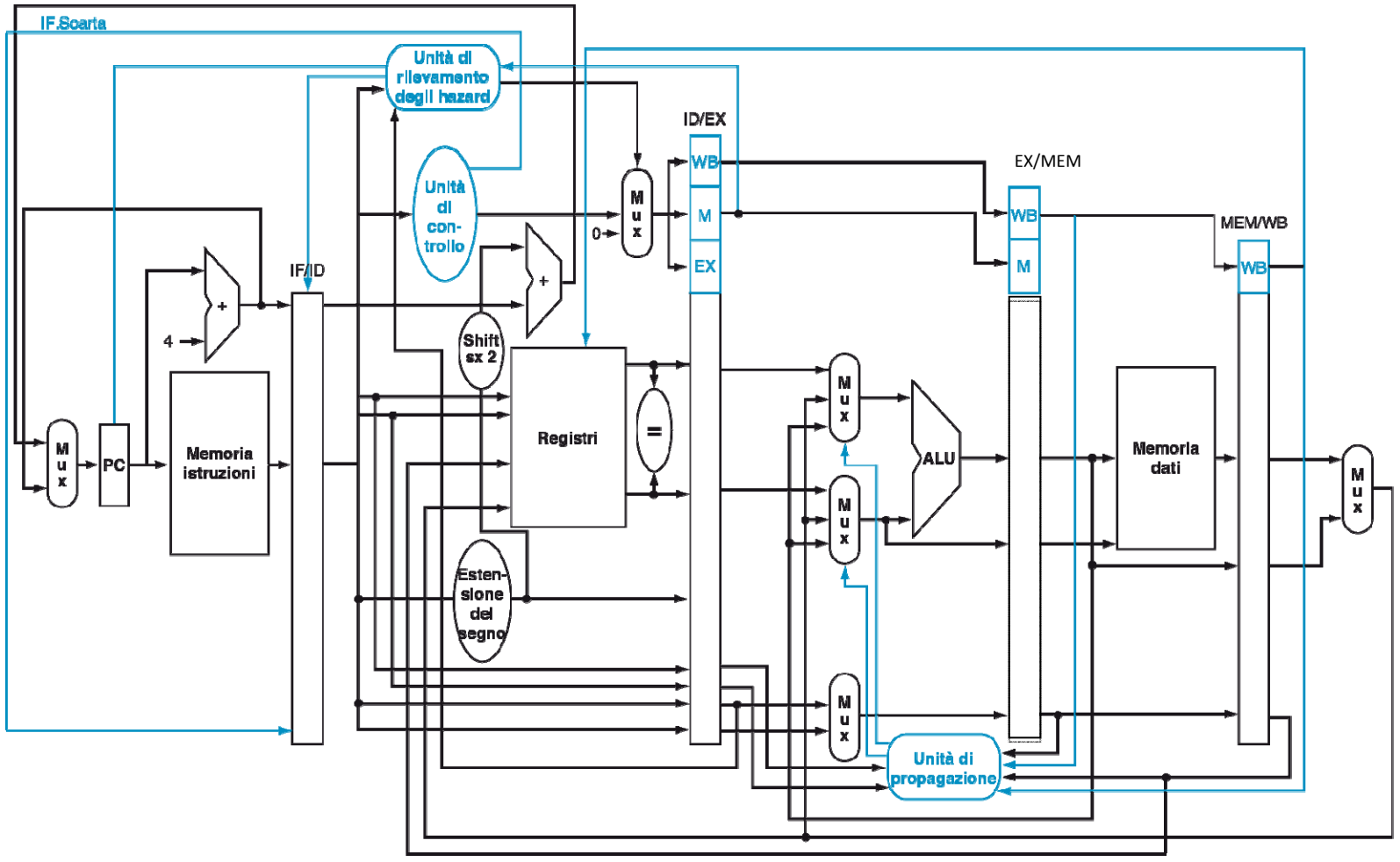
→ ***conflitto***: senza forwarding 1 stallo per beq (se confrontatore veloce)

→ ***propagazione*** **EX/ID** nessuno stallo

Spostamento esecuzione salto condizionato allo stadio ID

- Riduce la penalità di una istruzione di salto condizionato (***sulle successive***) a un solo ciclo di clock
  - Necessario scartare l'istruzione nella fase di fetch se il salto deve essere eseguito
  - Per eliminare l'istruzione in esecuzione nello stadio IF viene aggiunto un segnale di controllo ***IF.Scarta*** che azzera la parte del registro di pipeline IF/ID che contiene l'istruzione ➔ istruzione diventa una **nop**

Unità di elaborazione e di controllo della pipeline completa ……. qualche dettaglio da aggiungere (vedi slide successiva)



…. i dettagli da aggiungere

- Il confrontatore genera un'uscita Zero (= 1 se ingressi uguali, 0 altrimenti) che viene memorizzata in ID/EX (ID/EX.Zero)

- la logica di controllo del salto, presente nello stadio EX

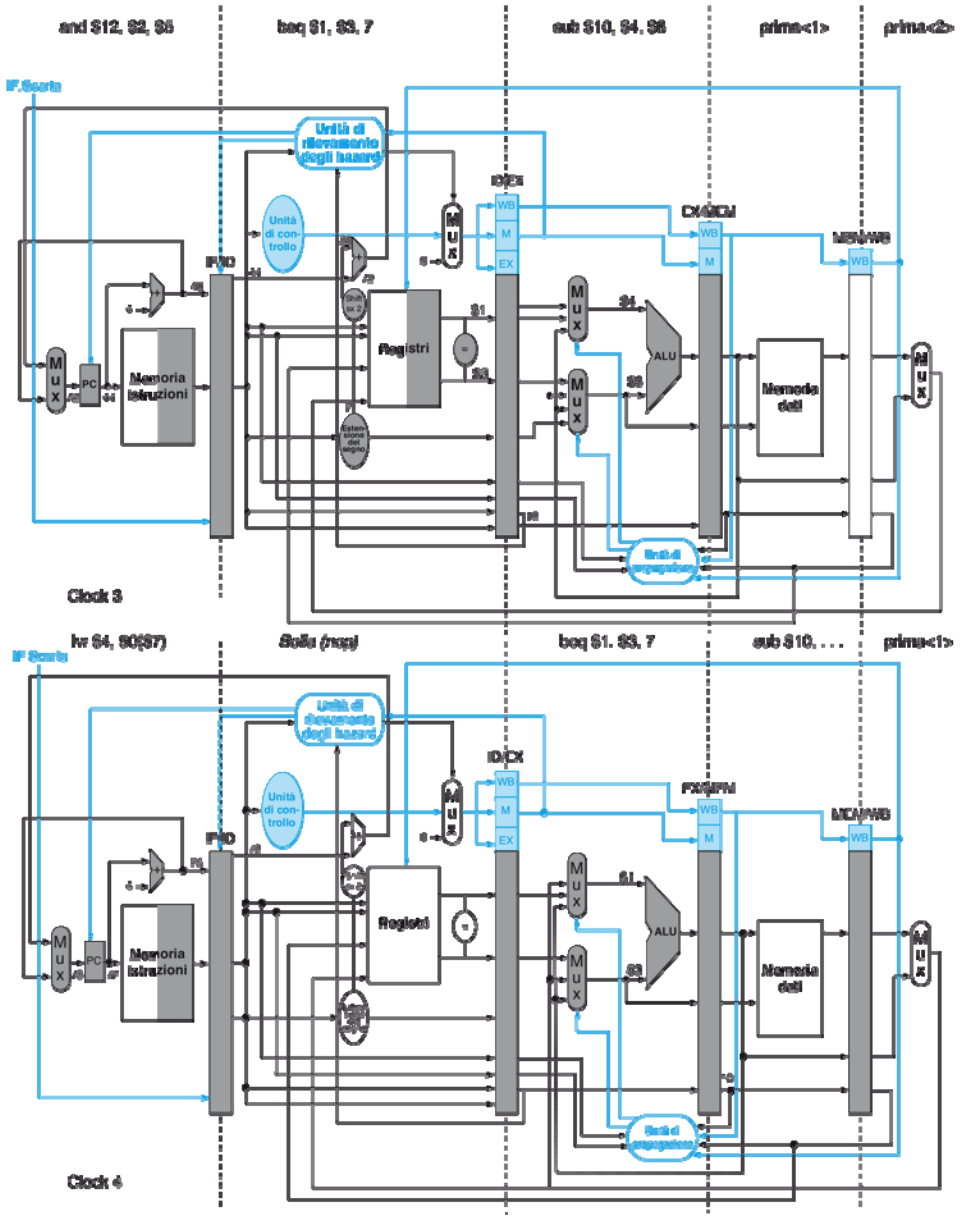
  PCsrc = ID/EX.Zero and ID/EX.M.Branch

  controlla il MUX in ingresso al PC ed è a 1 in caso di branch taken

- il segnale di controllo ***IF.Scarta*** che azzera la parte del registro di pipeline IF/ID che contiene l'istruzione è uguale a ***PCsrc***

Esempio

```
36 sub   $10, $4, $8
40 beq   $1, $3, 7      # salto relativo a PC:
                           #  40+4+7*4 = 72
44 and   $12, $2, $5
48 or    $13, $2, $6
52 add   $14, $4, $2
56 slt   $15, $6, $7
…
72 lw    $4, 50($7)
```


and $12, $2, $5
beq $1, $3, 7
sub $10, $4, $6
prima<1>
prima<2>
Unità di rilevamento degli hazard
Unità di controllo
Registri
Memoria istruzioni
Memoria dati
ALU
Clock 3
lw $4, 50($7)
Salta (nop)
beq $1, $3, 7
sub $10, . . .
prima<1>
Clock 4

Considerazioni sulle prestazioni nella pipeline

Definizioni:

- sia P una Prova (esecuzione di un programma ad alto livello trasformato in eseguibile con certi dati di ingresso)
- T(P) = tempo di CPU per la prova P
- Cicli(P) = numero di cicli di clock della CPU per la prova P
- CK = periodo di Clock
- FREQ = frequenza del Clock = 1/CK

La relazione fondamentale sulla Prova è

T(P) = Cicli (P) * CK = Cicli(P) /FREQ

Esempio - dato il tempo T(P) misurato e nota le frequenza di clock, determinare il numero di cicli di clock necessari all'esecuzione:

T(P) = 10s; FREQ = 2GHz

→ Cicli(P) = $20*10^9$

Prestazioni associate alle istruzioni

Definizioni:

- NI(P) = numero di istruzioni eseguite nell'esecuzione di P
- CPI(P) = cicli di clock per istruzione (valore medio calcolato su tutte le istruzioni del programma di P) = Cicli(P) / NI(P)

Tempo di esecuzione in funzione del numero di istruzioni

- T(P) = NI(P) * CPI(P) * CK

CALCOLO prestazioni nella pipeline

Stalli(P) = numero di cicli di stallo nella prova P

Cicli(P) = NI(P) + Stalli(P) + 4

≈ NI(P) + Stalli(P) (dato che per grandi numeri il 4 è trascurabile)

CPI(P) = Cicli(P) / NI(P) = (NI(P) + Stalli(P)) / NI(P)

T(P) = (NI(P) x CPI(P)) / f_{clock}