Calcolatori Elettronici B
a.a. 2007/2008

Tecniche Pipeline: Gestione delle criticità

Massimiliano Giacomin

Pipeline: i problemi

- Idealmente, il throughput è di una istruzione per ciclo di clock!
- Purtroppo, in realtà esistono diverse problematiche:
  - Criticità **strutturali**: HW non può eseguire una certa combinaz. di istruzioni [es.contesa della stessa risorsa da parte di più istruzioni]
  - Criticità **sui dati**: un'istruzione dipende dal risultato di un'istruzione precedente che si trova ancora nella pipeline. E' necessario attendere che il risultato sia pronto.
  - Criticità **sul controllo**: l'istruzione successiva ad un'istruzione di salto deve attendere l'esecuzione della precedente, per sapere se/dove saltare.

<u>Un chiarimento sui termini che useremo</u>

CRITICITA': un'istruzione non può essere eseguita nel ciclo di clock immediatamente seguente (pena comportamenti scorretti)

⇒ **STALLO**: sospensione di una unità della pipeline (e delle precedenti)

PIPELINE:

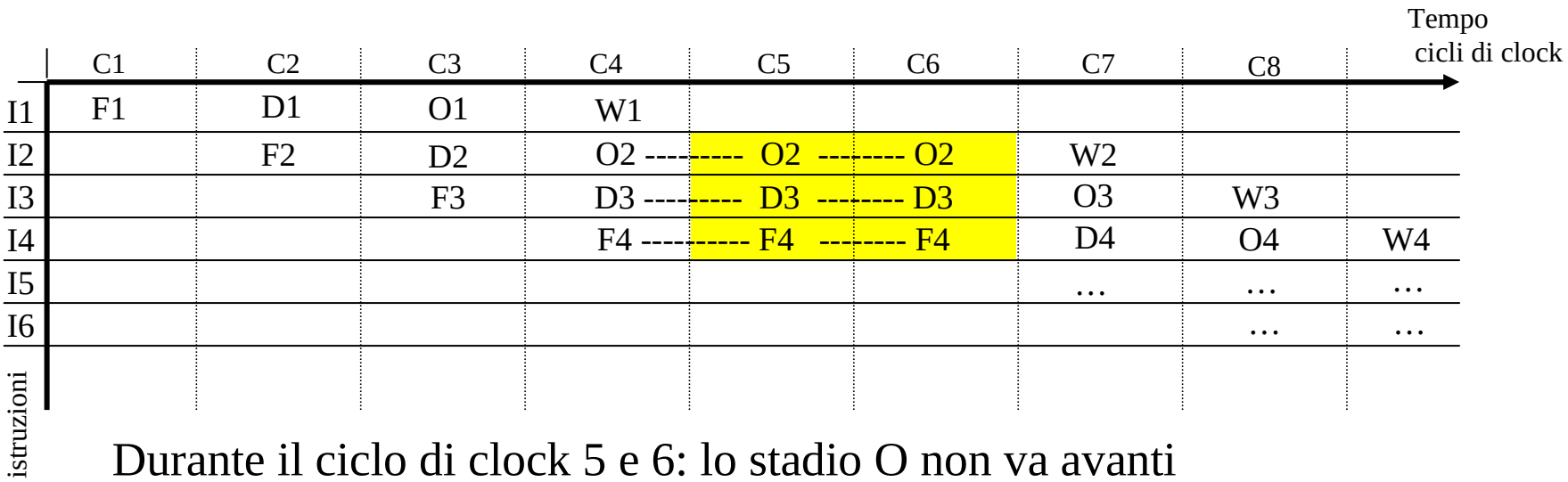
CRITICITA’ STRUTTURALI E STALLO

Un caso di criticità strutturale: stadio che richiede più cicli di clock

Esempio 1: Lo stadio di esecuzione O nell'istruzione I_2 richiede più di un ciclo di clock (per esempio, per effettuare una divisione).
NB: ciò perché si riusa la stessa unità di elaborazione, p.es. perché replicarla aumenterebbe costo e stadi (con gestione più difficile)

……
I1: Add R2,R3,R4
I2: Div R5,R7,R6
……..

⇒ **Stallo della pipeline** per far "aspettare" le istruzioni seguenti che richiedono lo stadio di esecuzione O



Durante il ciclo di clock 5 e 6: lo stadio O non va avanti

⟶ Lo stadio W non fa alcuna operazione (no dati su cui lavorare)

⟶ Lo stadio D rimane per I_3 e non accetta I_4 (O impegnato da I_2)

⟶ F rimane per I_4 e non accetta nuova istruzione

• La pipeline può essere pensata come un “registro a scorrimento”:

- al termine di ogni ciclo, ogni istruzione si “sposta in avanti” di uno stadio
  (le informazioni contenute nei registri interstadio si spostano in avanti di un registro)
- se uno stadio richiede più cicli di clock, necessariamente tutta la pipeline
  entra in stallo: l’istruzione nello stadio precedente (seguente nel programma!)
  non può entrare finché lo stadio non ha terminato il suo compito
  (e il registro “ricomincia a scorrere”)

⇒ Lo stadio bloccato e i precedenti non accettano nuove istruzioni

Cosa vuol dire che uno stadio “non accetta nuove istruzioni”?

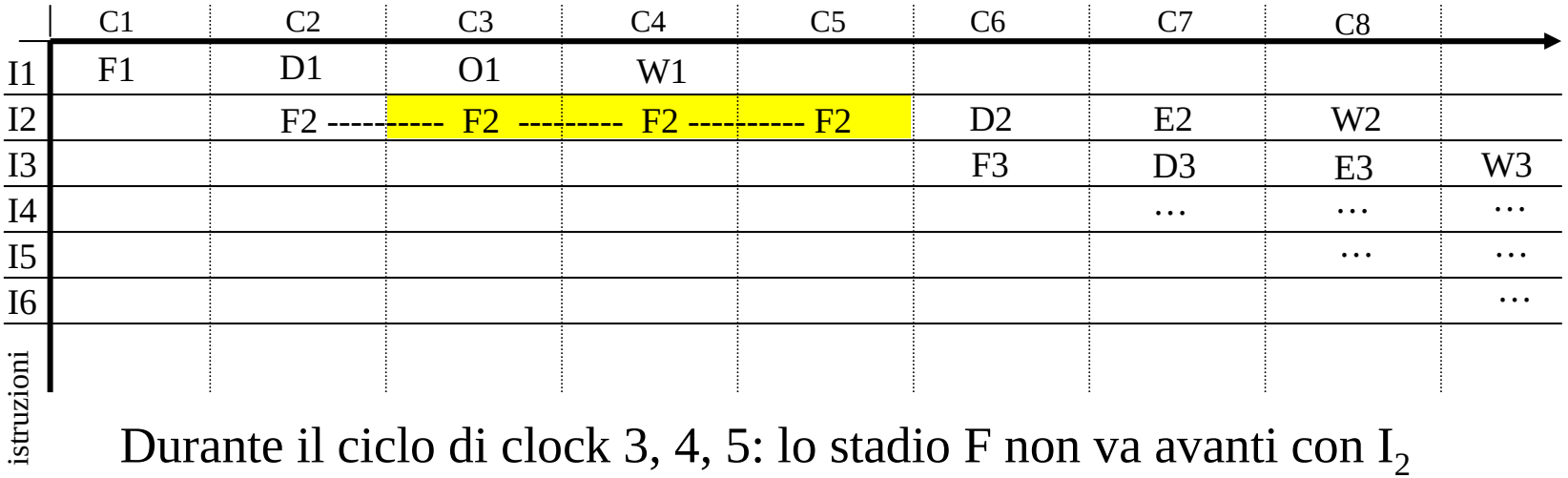
- Esso deve “mantenere” l’istruzione corrente.
- Il buffer interstadio che lo precede (in input) deve mantenere le stesse informazioni,
  cosicché nel ciclo successivo lo stadio continua con l’istruzione corrente

Nel caso precedente, gli stadi F – D – O non accettano nuove istruzioni per
2 cicli di clock: PC, B1 e B2 non sono sovrascritti, ma mantengono i valori precedenti.

NB: nel caso precedente, si dice che la pipeline è “in stallo” per 2 cicli di clock.

Esempio 2: Miss di cache

- La cache è necessaria per rendere efficace la tecnica con pipeline:
  se T_{clock} dovesse essere maggiore del tempo di accesso in memoria RAM,
  lo stadio di fetch sarebbe decisamente sbilanciato rispetto agli altri stadi
  (richiederebbe molto più tempo) ⇒ prestazioni scadenti
- Cache primaria nello stesso chip del processore
  ⇒ tempo di accesso pari al tempo per un'operazione interna al processore
- Tuttavia, sono possibili fallimenti di accesso alla cache istruzioni



Durante il ciclo di clock 3, 4, 5: lo stadio F non va avanti con I_2

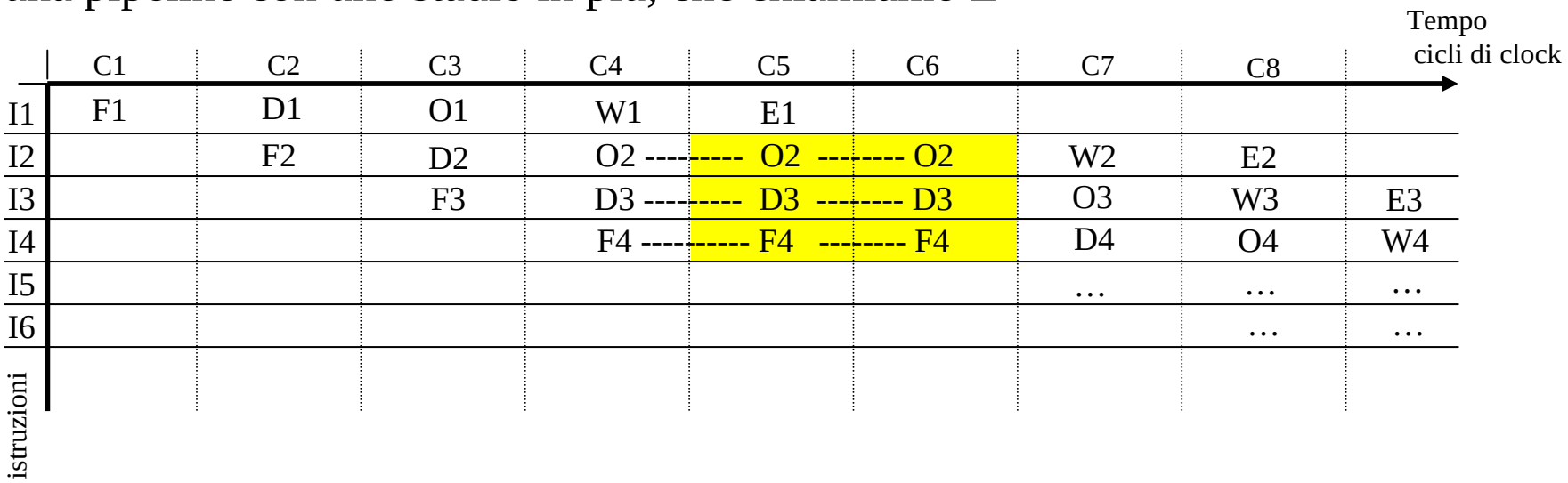
⟶ Lo stadio D non fa alcuna operazione (no dati su cui lavorare)

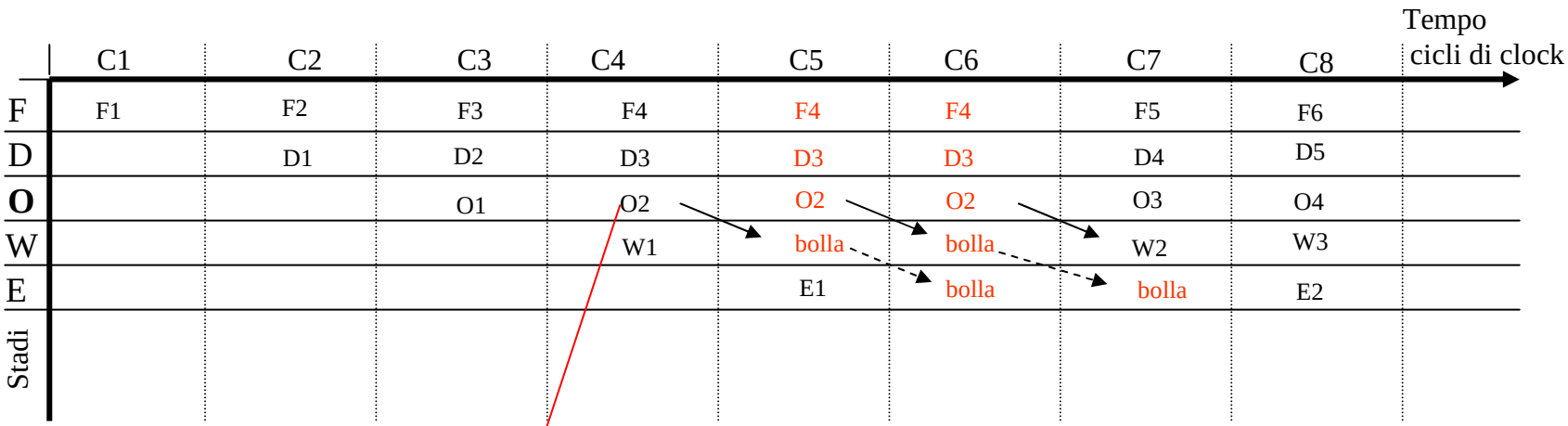
⟶ Lo stadio F non "accetta" nuove istruzioni

Stallo: concetti generali e realizzazione

1. Un certo stadio *S* necessita di attendere un certo numero *n* di cicli di clock addizionali.
   [es. precedenti: F per miss di cache, O per operazioni aritmetiche “lunghe”]
   ⇒ il buffer S-1 / S non viene scritto dallo stadio S-1 nei prossimi *n* fronti di clock [così per *n* cicli successivi i dati in ingresso rimangono immutati]
2. Gli stadi precedenti devono mantenere nei prossimi *n* cicli di clock le istruzioni correnti: l’operazione nello stadio S-1 non può procedere e così a catena per tutti gli stadi precedenti
   [cfr. pipeline come registro a scorrimento]
   ⇒ i buffer interstadio precedenti non vengono scritti e neppure PC per *n* fronti successivi di clock (dal ciclo corrente in poi, escluso l’ultimo)
3. Negli *n* cicli successivi lo stadio S+1 deve rimanere inattivo, perché non ha dati su cui lavorare: è necessario fare in modo che negli *n* cicli successivi esso non alteri in modo indefinito registri o memoria (e così a catena gli altri stadi)
   ⇒ per *n* fronti di clock successivi (dal ciclo corrente in poi, escluso l’ultimo), nel buffer S/S+1 vengono imposti segnali di controllo che corrispondono a non effettuare alcuna operazione:
   vengono in pratica create *n* **bolle** che a partire dallo stadio S+1 si spostano lungo la pipeline fino ad arrivare all’ultimo stadio

Esempio precedente: stallo di due cicli con S=O (istruzione I_2) in cui immaginiamo una pipeline con uno stadio in più, che chiamiamo E

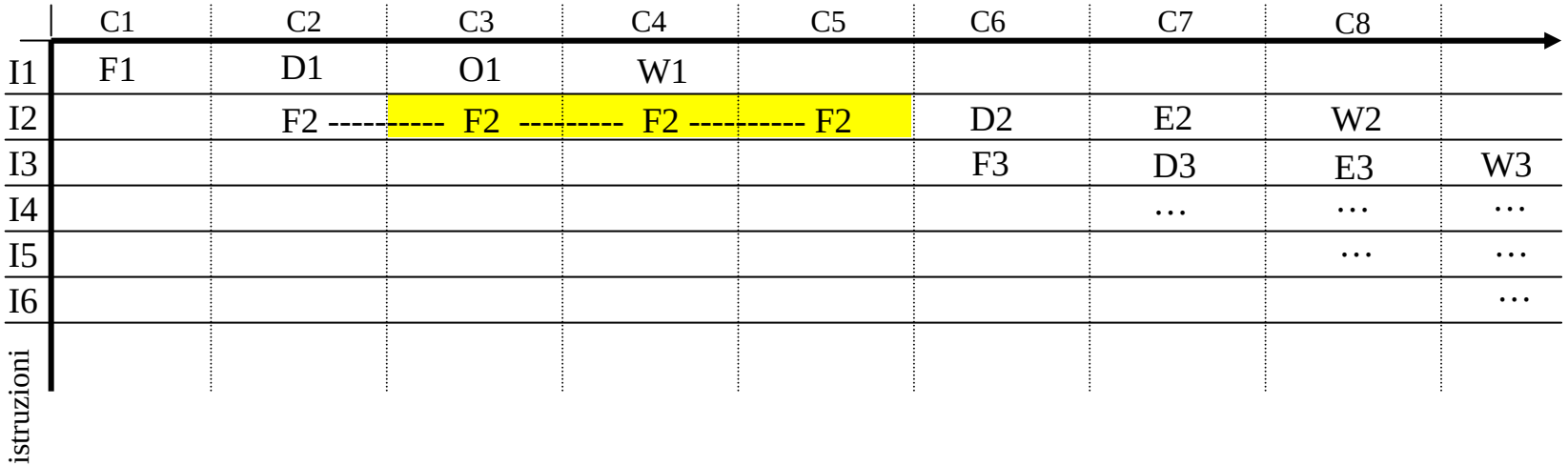


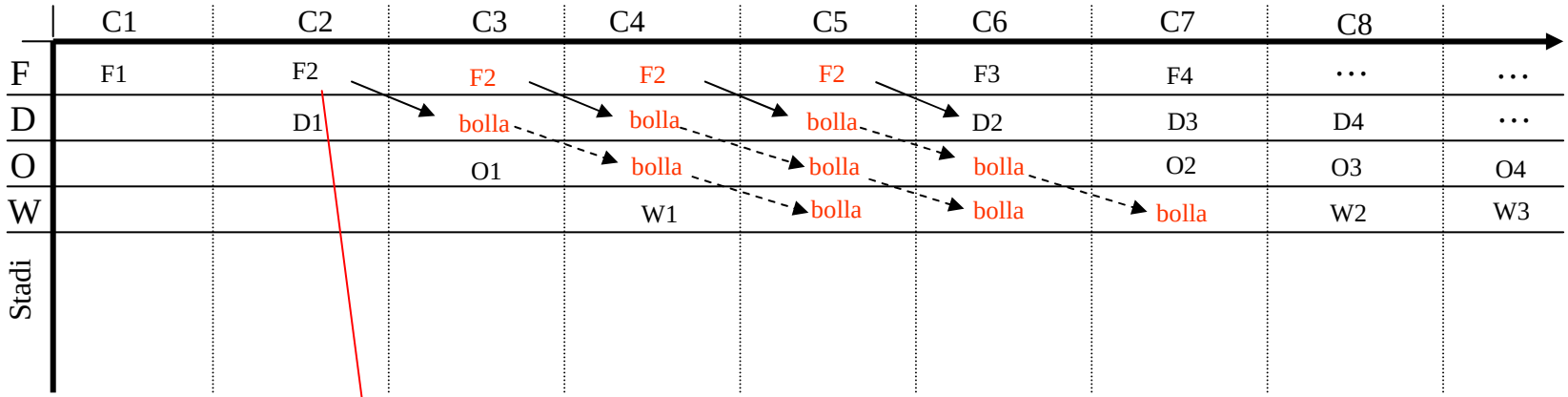


Riconoscimento situazione di stallo (2 cicli successivi)

NB: - lo stadio E in C5 prosegue normalmente
- propagazione della bolla nello stadio E

Esempio miss di cache: stallo di tre cicli con S = F (istruzione I_2)

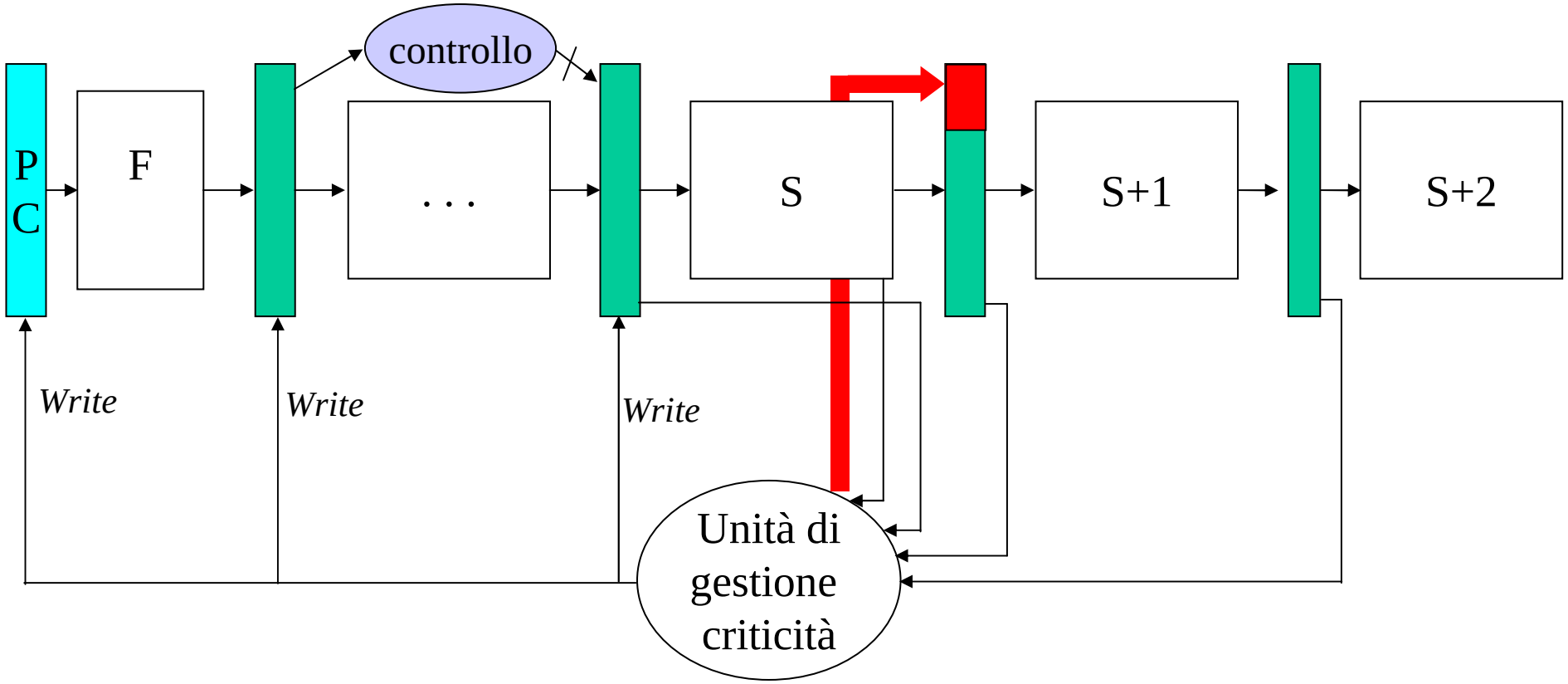




Riconoscimento situazione di stallo (3 cicli successivi)

NB: - lo stadio O in C3 prosegue normalmente
- propagazione delle bolle in stadi O e W

Stallo in un determinato stadio S: Realizzazione circuitale



- Sulla base dei valori nel buffer stadio S e successivi, si rileva necessità di stallo.
- Se la pipeline è posta in stallo:
  - i segnali *Write* impediscono la scrittura di PC e buffer precedenti:
    nei cicli di clock successivi di stallo nulla cambia per gli stadi F, … S
  - nel buffer S/S+1 vengono imposti i segnali di controllo per forzare operazione
    "innocua": formazione di una bolla che si propagherà negli stadi successivi

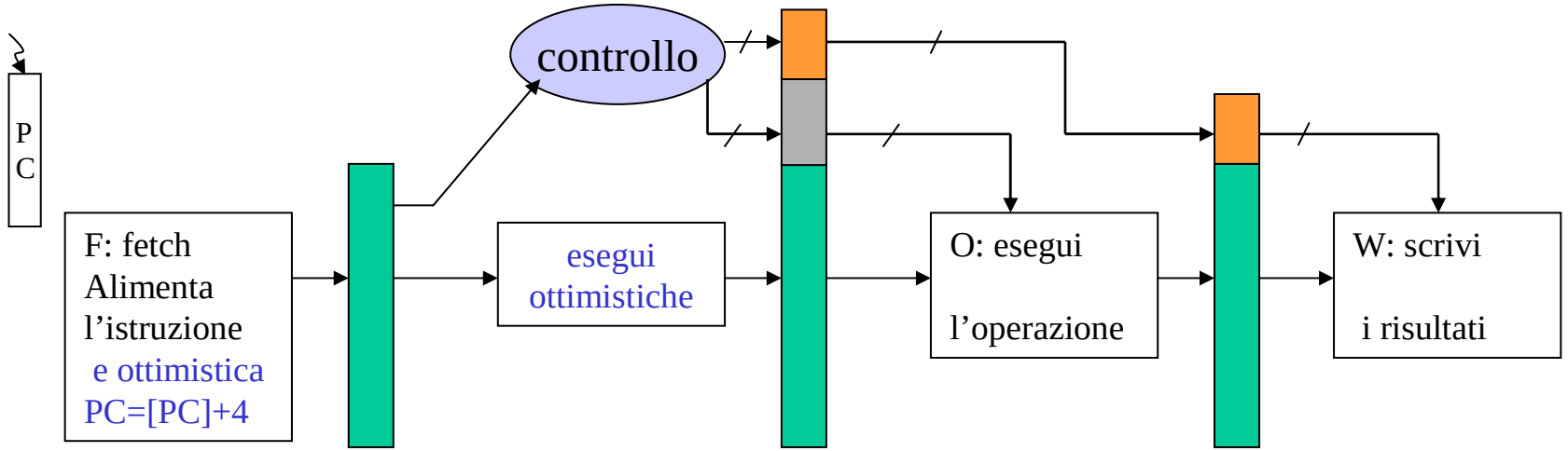
Esempio precedente (stallo per operazione complessa nello stadio O)

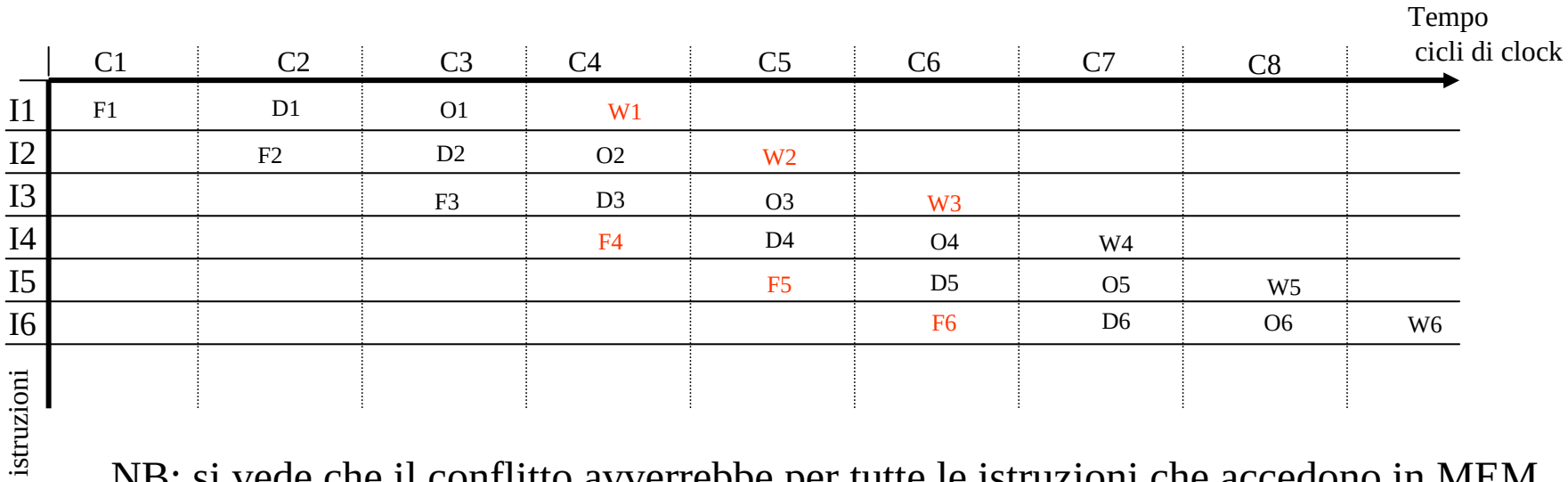
- L'unità di rilevazione criticità sulla base dei segnali provenienti dallo stadio O rileva la necessità di mettere in stallo la pipeline (S=O)
- Impedisce la scrittura di PC e dei due buffer F/D e D/O (attesa degli stadi di fetch, decode, execute)
- Forza i segnali di controllo del buffer O/W in modo da mettere una bolla nello stadio W (impedendo nei cicli di clock successivi la scrittura di registri e memoria)

Esempio miss di cache

- L'unità di rilevazione criticità sulla base dei segnali provenienti dalla cache rileva la necessità di mettere in stallo la pipeline (S=F)
- Impedisce la scrittura di PC mettendo in attesa lo stadio Fetch
- Forza i segnali di controllo del buffer F/D in modo da mettere una bolla nello stadio D (impedendo nei cicli di clock successivi la scrittura di registri e memoria)

Un altro caso di criticità strutturale: la memoria contesa in due stadi diversi

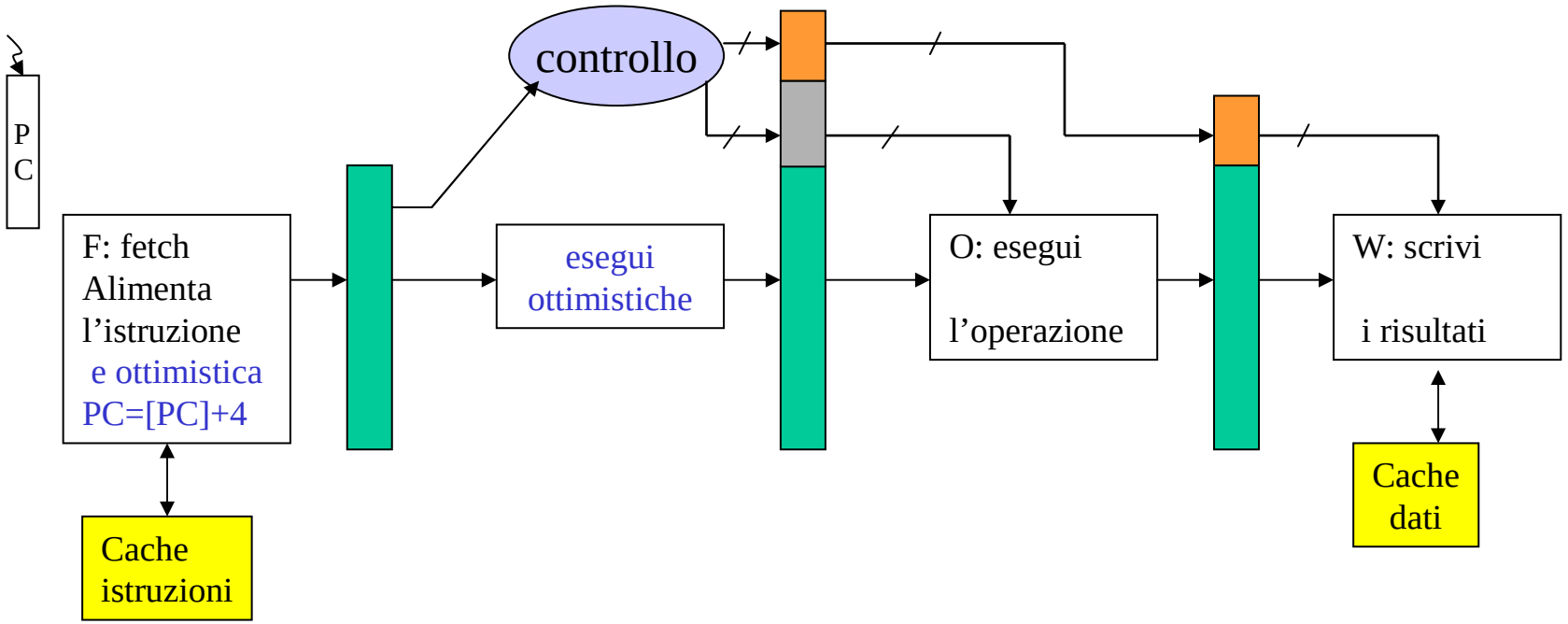




NB: si vede che il conflitto avverrebbe per tutte le istruzioni che accedono in MEM

Usare lo stallo è potenzialmente molto inefficiente: se tutte le istruzioni nella pipeline accedono alla memoria, dovrei attendere lo svuotamento della pipeline prima di caricare una nuova istruzione!

➡ RIMEDIO: DIVIDERE LA MEMORIA IN DUE!



NB: questa è la soluzione adottata nel MIPS didattico [cfr. Patterson & Hennessy]

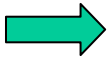
➡ In questo modo nel MIPS si evitano le criticità strutturali (a parte miss)

➡ Altro effetto: aumento capacità di trasferimento cache

ORGANIZZAZIONE HW FINO A QUESTO MOMENTO:

- **Unità di controllo** (gestisce il controllo della pipeline ma non le criticità).
- **Unità di rilevamento delle criticità** (gestisce le criticità, finora solo strutturali, mettendo in stallo la pipeline).

In linea teorica, tutte le criticità che vedremo potrebbero essere gestite in questo modo (soltanto mediante stallo), tuttavia questo approccio è di fatto assolutamente inefficiente.



Le criticità strutturali di solito sono evitate (per quanto possibile) a priori. Le altre criticità sono gestite con tecniche specifiche che vedremo nel seguito.

PIPELINE:

INFLUENZA DEGLI STALLI SULLE PRESTAZIONI

PIPELINE IN PRESENZA DI STALLI:

CPI = CPI ideale + Cicli di stallo per istruzione

$\Rightarrow$ $Throughput_{pipeline} = 1/CPI$ (espresso come istruzioni/ciclo)

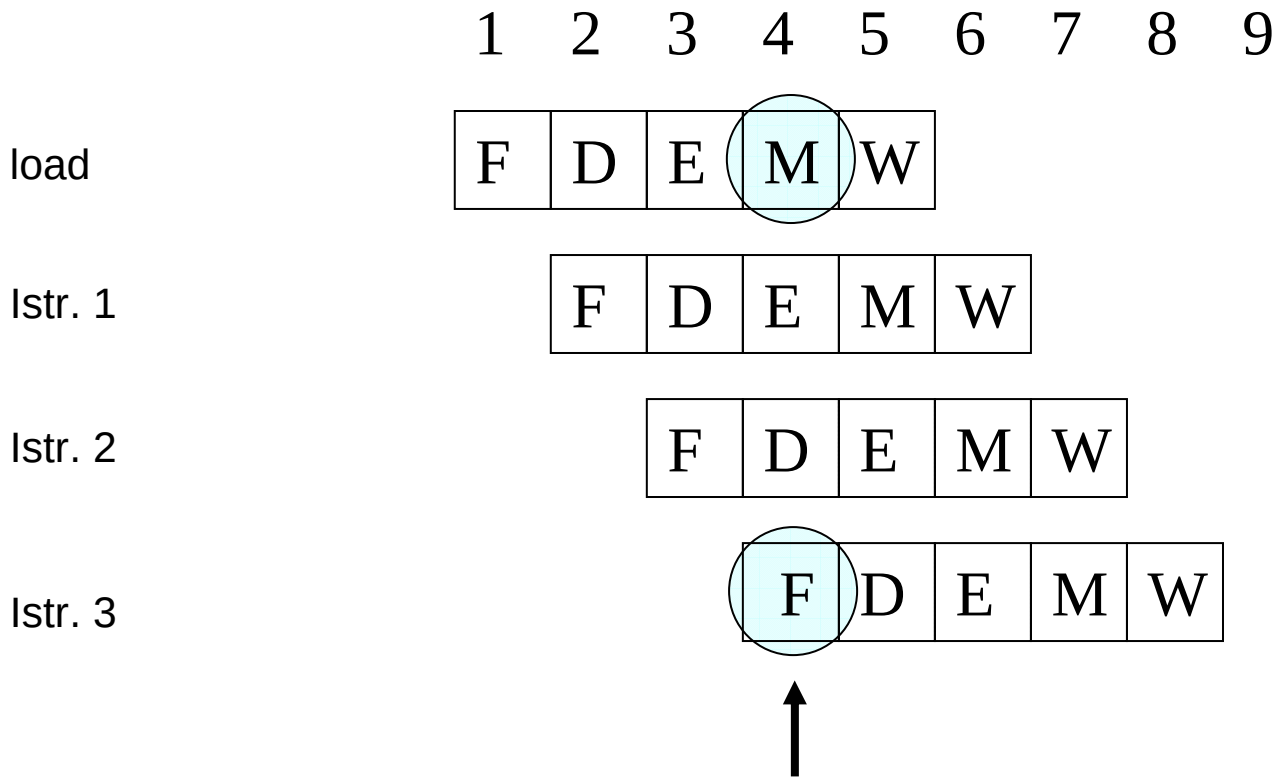
$1/(CPI*T_{clock})$ (espresso come istruzioni/sec)

Esempio: prestazioni in caso di criticità strutturale

- Si consideri un processore dotato di pipeline (5 stadi) che disponga di una sola memoria cache per i dati e per le istruzioni:

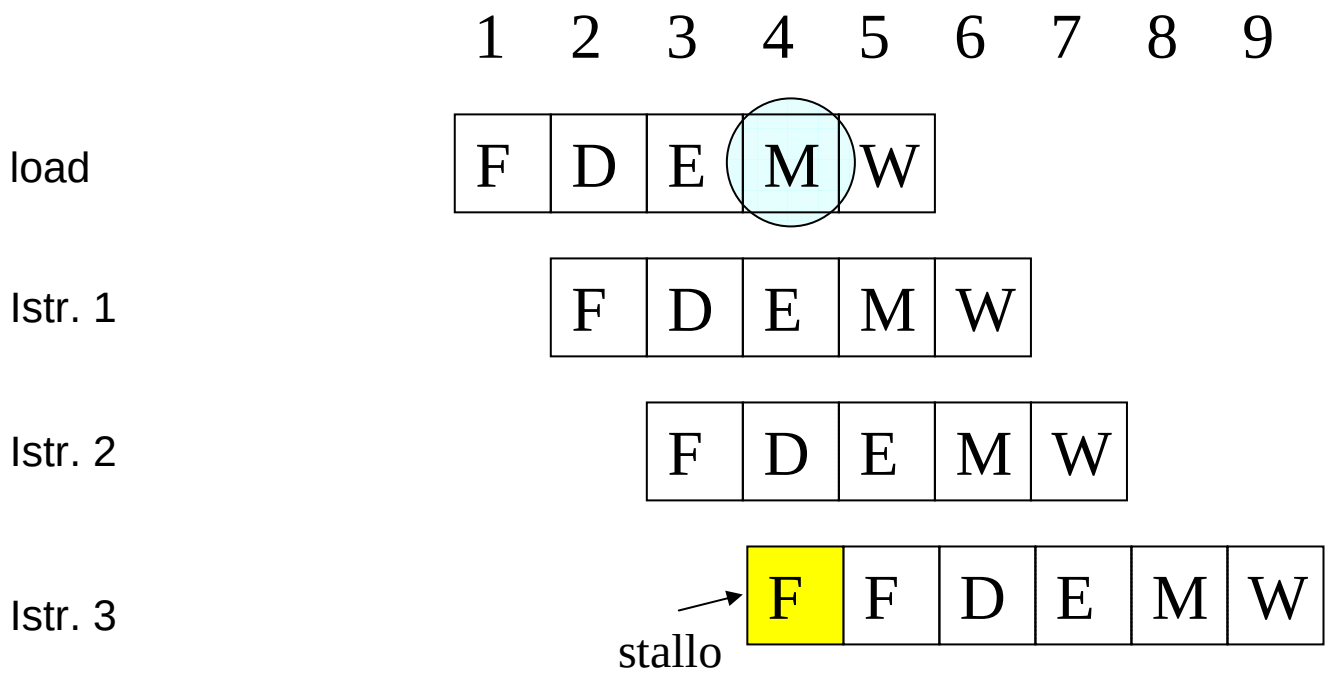
1) Si proponga una soluzione al problema basata sullo stallo della pipeline, descrivendo mediante diagramma temporale ciò che avviene in caso di conflitto sulla memoria.

2) Si supponga che il 40% delle istruzioni eseguite facciano riferimento alla memoria dati e che non esistano altri tipi di stalli: calcolare il CPI medio ed il throughput delle istruzioni.



- Ogni volta che un'istruzione richiede un accesso ai dati, c'è un conflitto con un'istruzione successiva che prevede, nello stadio di fetch, un accesso alla memoria per il prelevamento dell'istruzione.
- Per risolvere il conflitto, la pipeline deve essere messa in stallo per 1 ciclo di clock.

Una possibile soluzione: unità di criticità che pone in stallo la pipeline per un ciclo di clock



L'istruzione 3 viene completata alla fine del ciclo 9, nessuna istruzione viene completata nel ciclo 8

NB: notare che, ovviamente, è l'istruzione I_3 successiva alla load che "stalla"!!!

- Ricordando la formula per il CPI nel caso di stalli:

CPI = CPI ideale + Cicli di stallo per istruzione

Nel nostro caso: una sola memoria per dati e per istruzioni;

il 40% delle istruzioni eseguite fanno riferimento ai dati

e quindi "subiscono una penalità" ciascuna di un ciclo di clock

(anche se – come visto – in realtà la penalità è causata dallo stallo di una istruzione successiva)

$\Rightarrow$ Supponendo che non esistano altri tipi di stalli si ha

$$CPI = 1 + \underbrace{0{,}40 \times 1}_{\text{Cicli di stallo}} = 1{,}4$$

Throughput = 1/CPI = 1/1.4 = 0.71 istruzioni/ciclo

PIPELINE:

CRITICITA’ SUI DATI

Criticità sui dati

quando un dato sorgente è il risultato di un'operazione precedente

I_1: $A \leftarrow 5$

. . .

I_2: $A \leftarrow 3 + A$

I_3: $B \leftarrow 4 * A$

Con multiciclo: B=32

Con pipeline: B = 20 (I_3 accede al registro A non ancora modificato da I_2 – poiché la modifica avviene nell'ultimo stadio W)

Non accade con

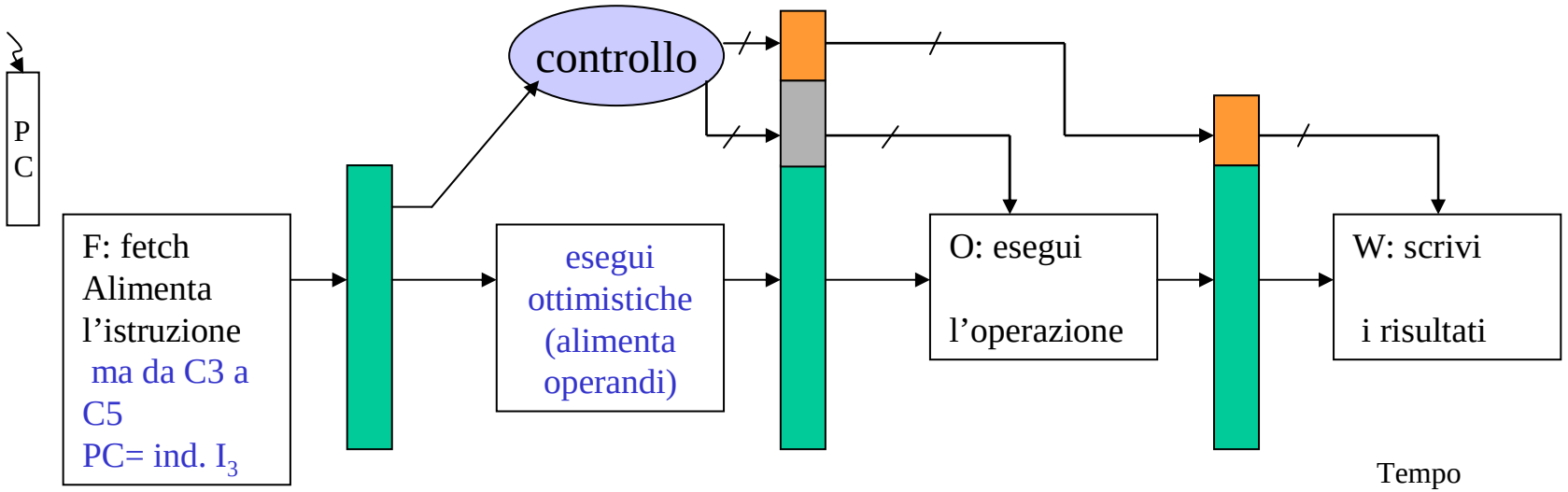
$A \leftarrow 5 + C$

$B \leftarrow 20 + C$

(istruzioni indipendenti)

Primo rimedio: gestire criticità soltanto con stallo pipeline

……
I1: Mul R4, R2,R3
I2: Add R5, R4,R6
……..



Tempo
cicli di clock

istruzioni	C1	C2	C3	C4	C5	C6	C7	C8	
I1	F1	D1	O1	W1					
I2		F2	D2	D2	D2	O2	W2		
I3			F3	F3	F3	D3	O3	W3	
I4						F4	D4	O4	W4
I5							F5	D5	O5
I6								F6	D6

Rilevazione criticità: D2 non può procedere a prelevare R4

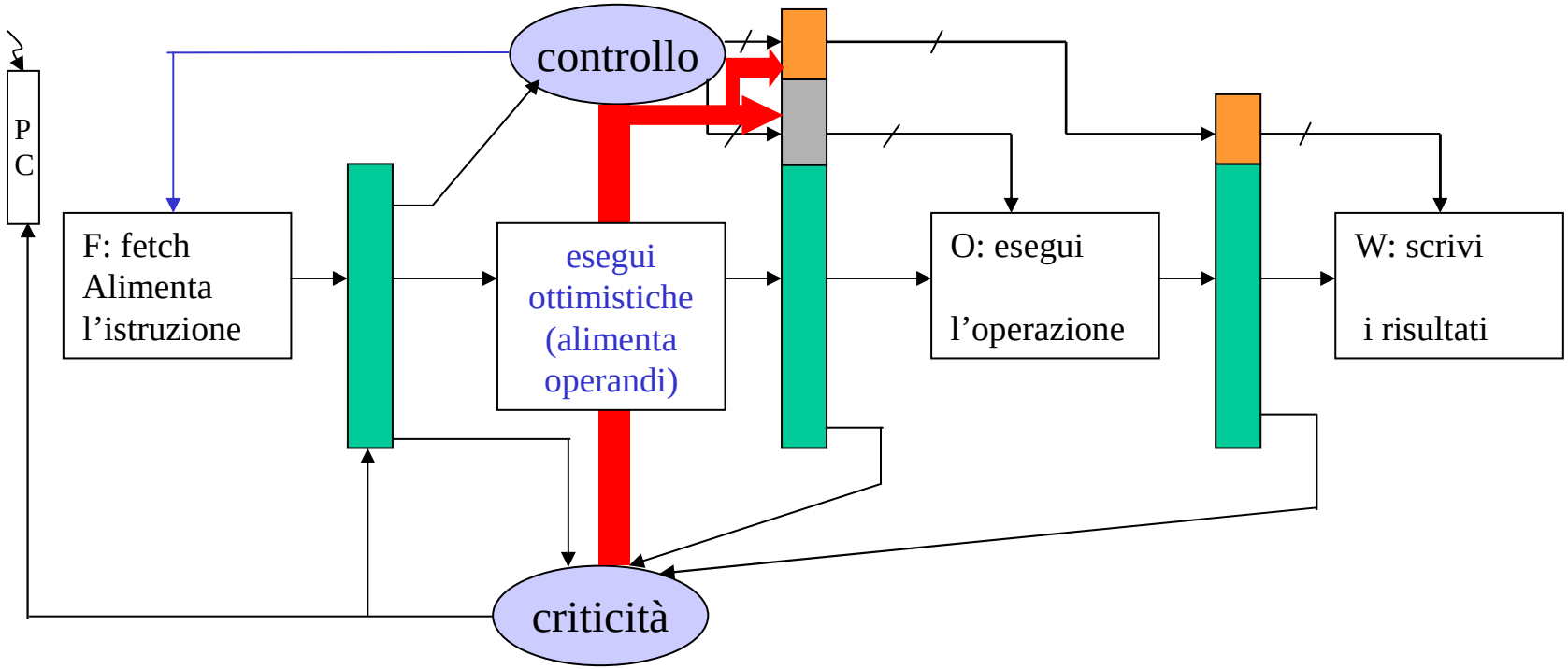
Dato aggiornato in R4 scritto da W1 alla fine di C4: si può procedere a prelevarlo

Commenti al lucido precedente:

- Il rilevamento della criticità si fa quando l'istruzione "dipendente" è nello stadio Decode: in questo stadio infatti avviene il prelievo degli operandi
- Sulla base dei campi dell'istruzione e dei registri interstadio successivi, l'unità di rilevazione criticità rileva che un operando sorgente coincide con la destinazione non ancora aggiornata di una istruzione precedente (presente in uno degli stadi successivi della pipeline!).
  Il prelievo dell'operando porterebbe quindi ad un valore errato.
- La pipeline viene allora messa in stallo nello stadio D: l'esecuzione dell'istruzione dipendente (e di conseguenza anche della successiva presente nello stadio Fetch) viene inibita in modo tale da procedere con la lettura dell'operando dopo che questo sia stato aggiornato [notare che le istruzioni precedenti poste negli stadi successivi - nel caso specifico I_1 - proseguono nella loro esecuzione]

NB: la gestione dello "stallo" corrisponde a quanto visto nel caso delle criticità strutturali

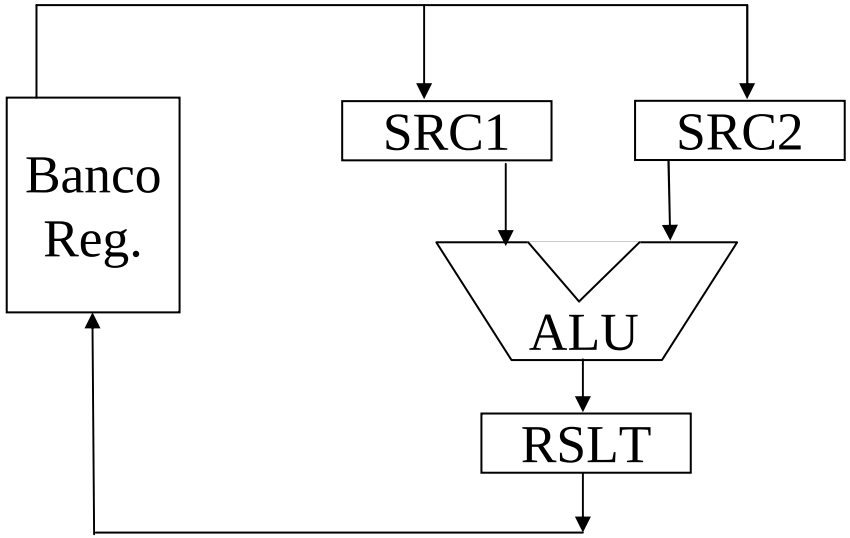
Realizzazione circuitale



NB: ovviamente, devono essere aggiunte le unità di gestione
delle criticità per gestire le criticità strutturali viste in precedenza.

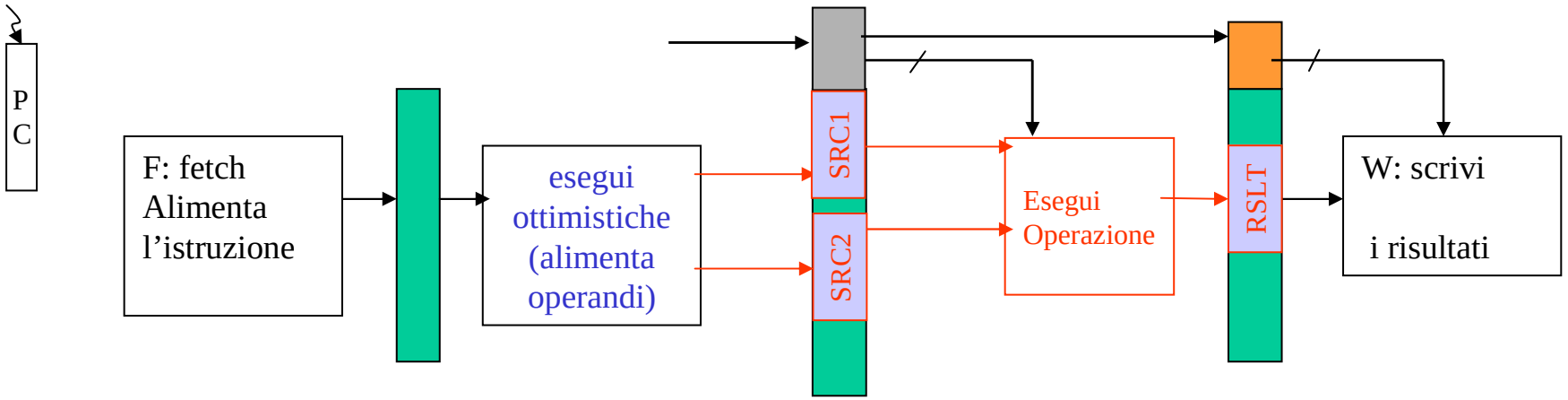
Secondo rimedio: propagazione dei dati

• NB: una possibile realizzazione della pipeline a 4 stadi potrebbe essere del tipo



• stadio D accede al banco registri in lettura
• stadio W accede al banco registri in scrittura

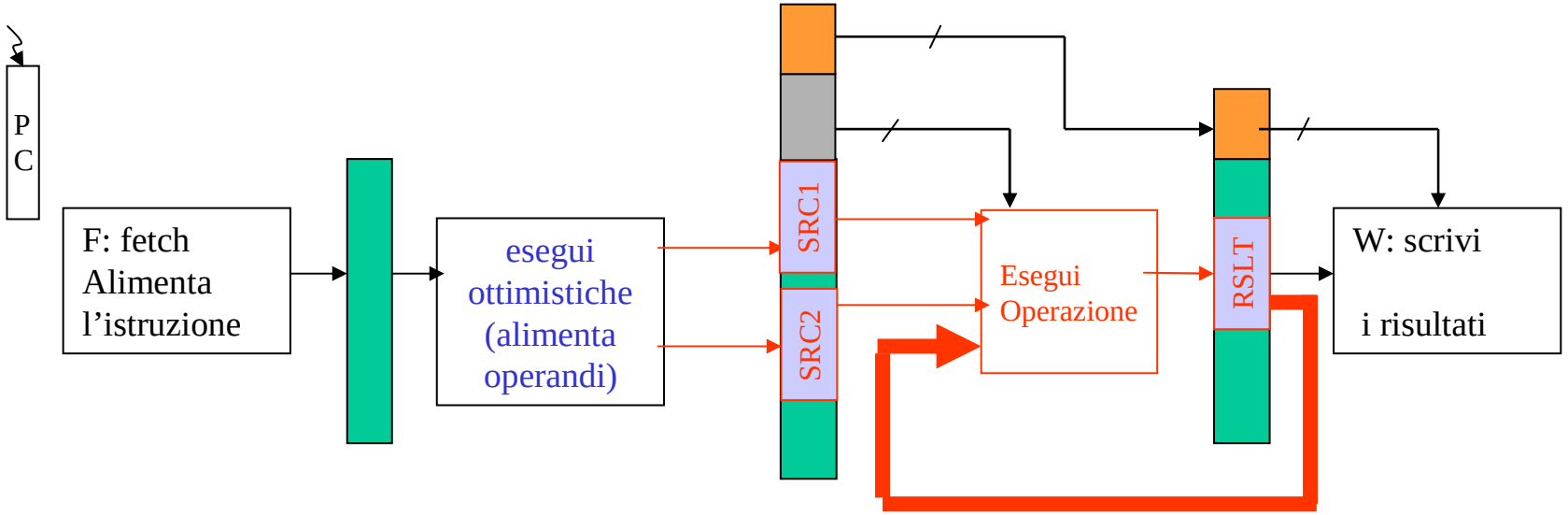
corrispondente a:



Rivediamo il caso precedente

……
I1: Mul R4, R2,R3
I2: Add R5, R4,R6
……..

IDEA: quando I2 arriva allo stadio O (uso di R4 per la ALU) il valore di R4 è già disponibile in RSLT



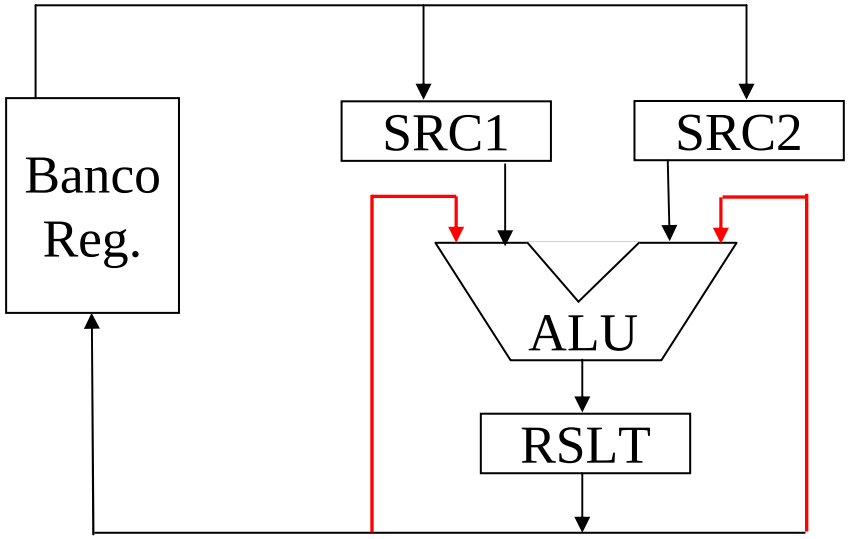
	C1	C2	C3	C4	C5	C6	C7	C8
I1	F1	D1	R2*R3	R4←				
I2		F2	D2	R4+R6	W2			
I3			F3	D3	O3	W3		
I4				F4	D4	O4	W4	
I5					F5	D5	O5	
I6						F6	D6	

Tempo in cicli di clock

istruzioni

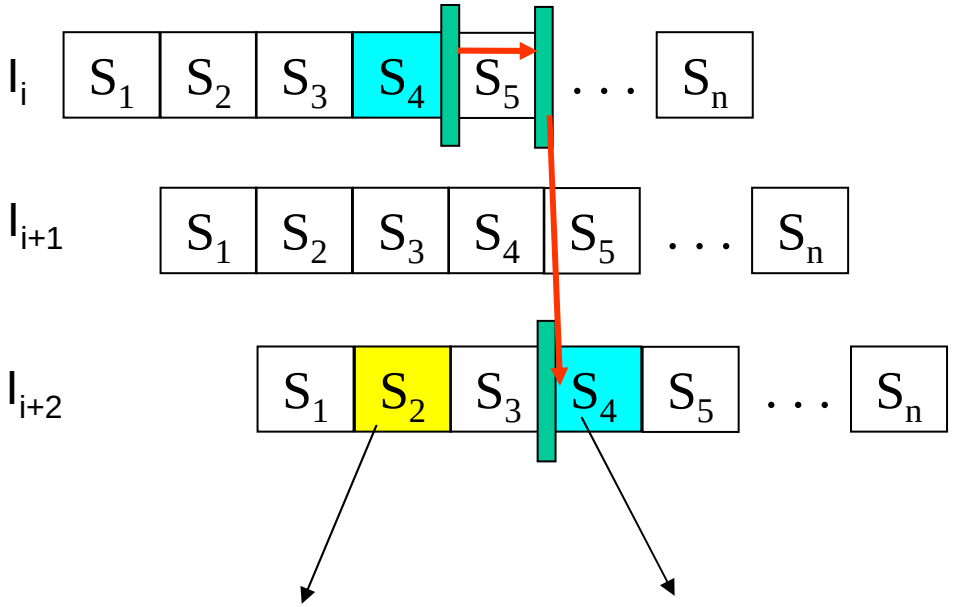
SI EVITA STALLO DI DUE CICLI DI CLOCK!

• Il percorso di anticipo dei dati viene realizzato aggiungendo i percorsi in rosso:



• Occorre una unità di propagazione che:
  - identifichi la situazione di potenziale criticità che può essere risolta con la propagazione
  - asserisca i segnali nei mux (omessi in figura) per determinare opportuno cammino dei dati; in particolare, la ALU deve prelevare l'operando "dipendente" non dal registro SRCi, ma da RSLT

In generale…



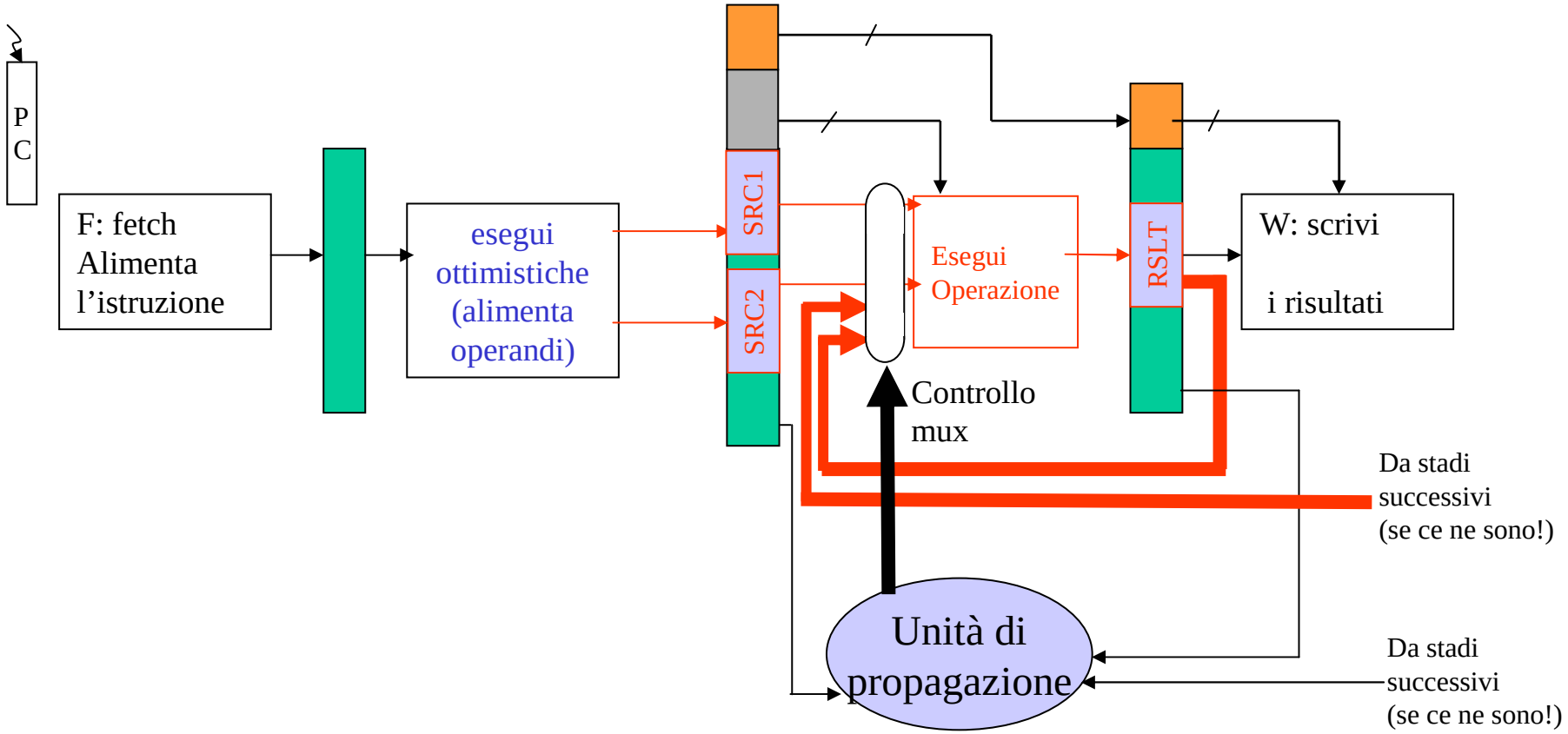
I_i produce un dato in S_4;
il dato è sorgente per I_{i+2}
che lo "usa" in S_4

Dato prelevato non ancora scritto da I_i

Dato prelevato non ancora scritto da I_i ma prodotto da I_i (in S_4) e disponibile in un registro interstadio successivo

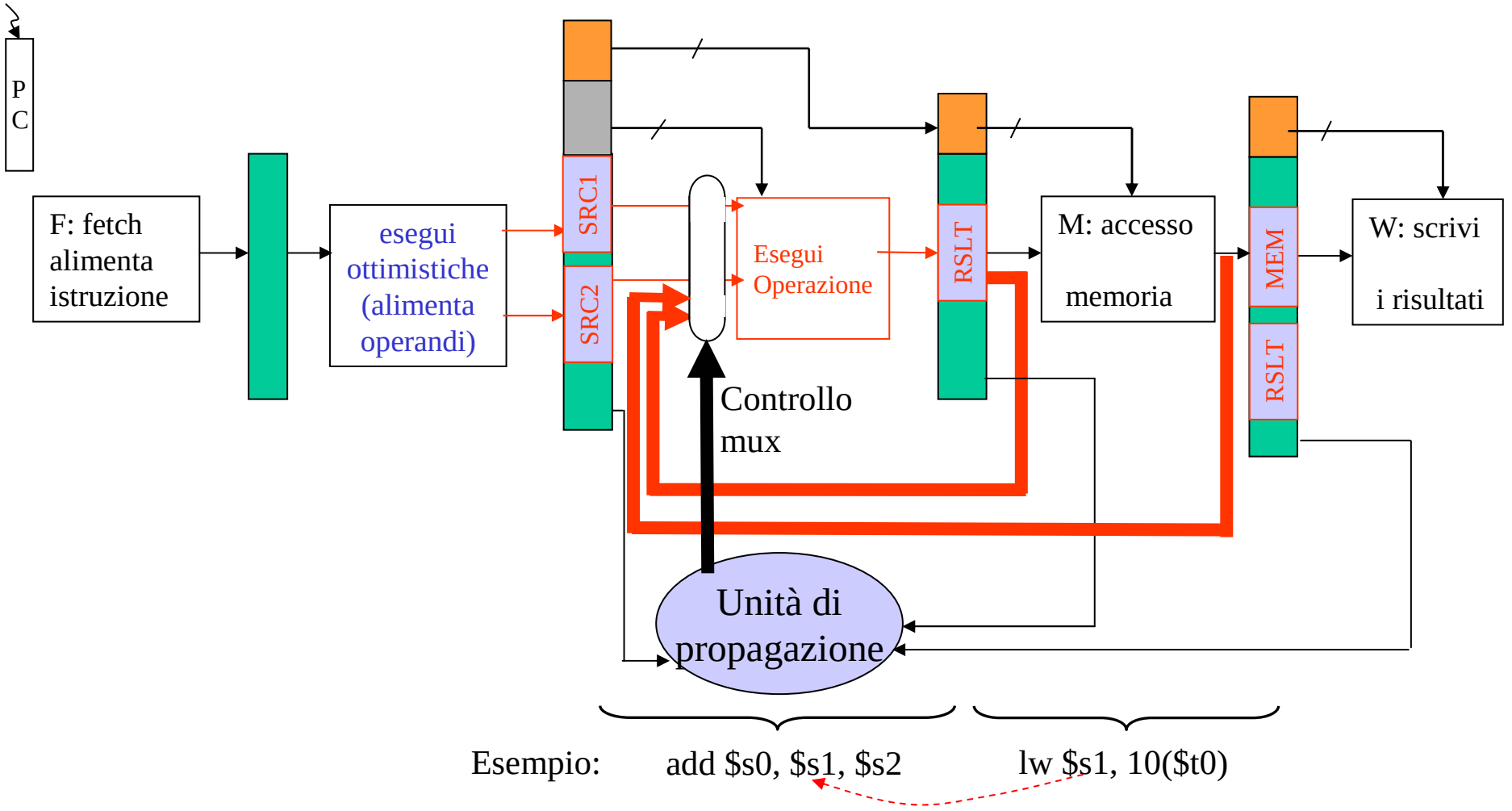
➡ Il dato necessario in S_4 (I_{i+2}) è prelevato dal registro interstadio successivo (in questo caso S_5/S_6) dove è disponibile

VEDIAMO COME REALIZZARE L'UNITA' DI PROPAGAZIONE…



- il valore è propagato nello stadio in cui viene effettivamente usato (O in questo caso), non in uno stadio precedente: "aspettando" è più probabile che un'istruzione precedente [quindi in uno stadio successivo!] abbia già prodotto il valore aggiornato;
- l'unità di propagazione, sulla base dell'istruzione potenzialmente dipendente e di quelle precedenti (in stadi successivi), determina se propagare operando sorgente
- i valori aggiornati vengono prelevati sempre da un registro di pipeline, non all'uscita di un elemento combinatorio di calcolo (ciò allungherebbe cammino critico e T_{clock})

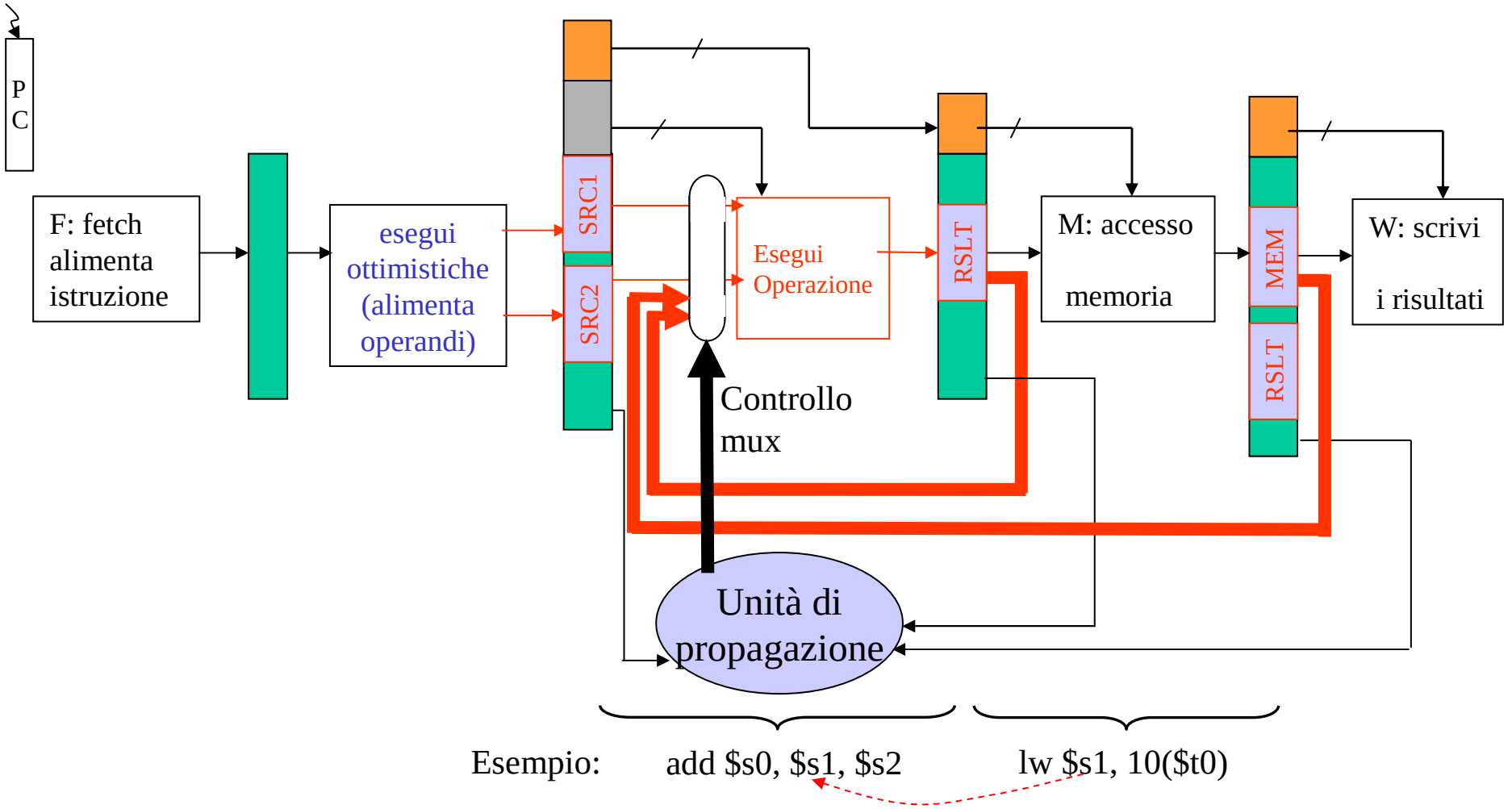
Esempio errato nel caso di pipeline a 5 stadi:



Se l'istruzione nello stadio E dipende da quella che si trova nello stadio MEM (per il dato prelevato dalla memoria) sembra di poter propagare da uscita memoria…

Ma avrei in serie memoria e ALU!!! Allungamento del periodo di clock!

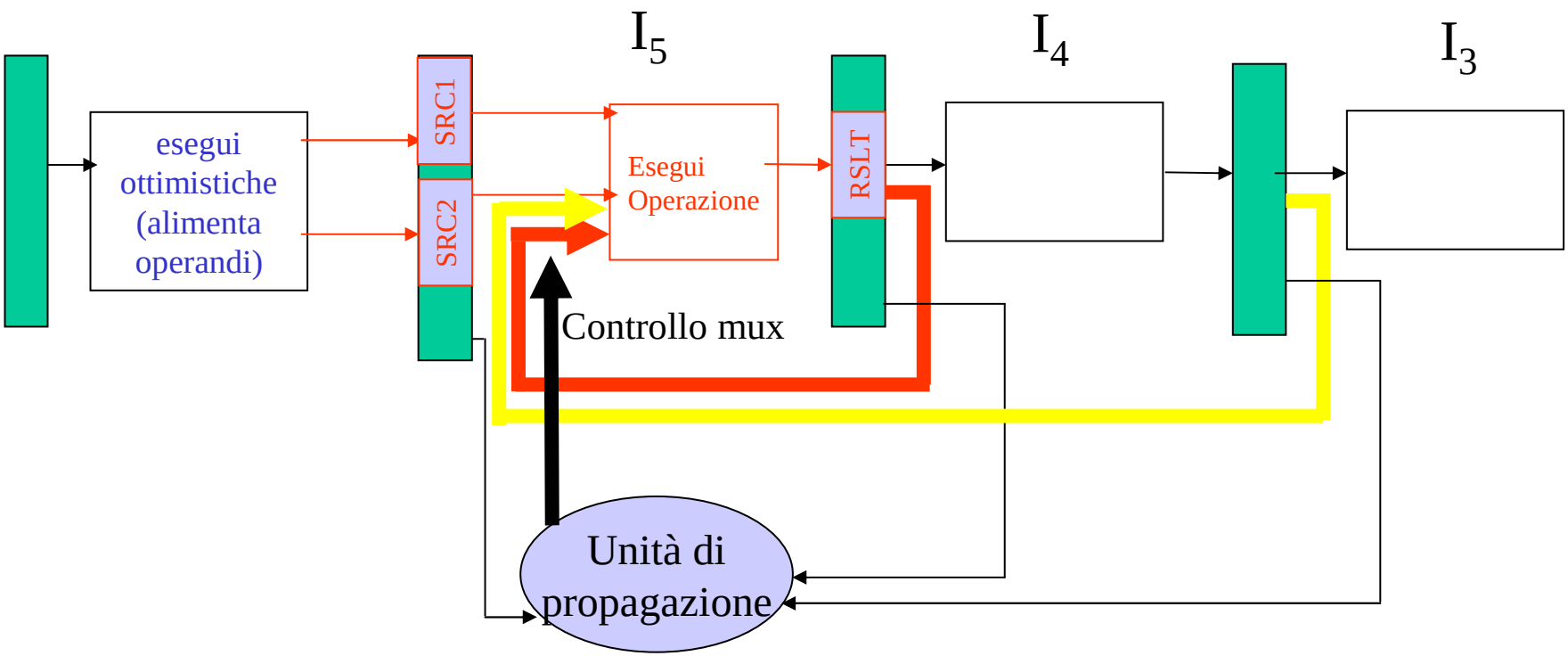
Esempio corretto nel caso di pipeline a 5 stadi:



Se l'istruzione nello stadio E dipende da quella che si trova nello stadio MEM (per il dato prelevato dalla memoria) è purtroppo necessario uno stallo!

(si parla di criticità "carica e usa")

E se la propagazione per lo stesso operando deve essere fatta da più stadi?



Il dato da propagare è quello dell'istruzione più recente, ovvero la "più vicina" tra quelle precedenti (I_4). Questa si trova nello stadio più a sinistra (il meno avanzato)

⇨ Vince il dato dello stadio meno avanzato! (il rosso nel caso in figura)

Es.

add R2, R3, R4 I_3
add R2, R5, R6 I_4
add R4, R2, R3 I_5

dipendenze

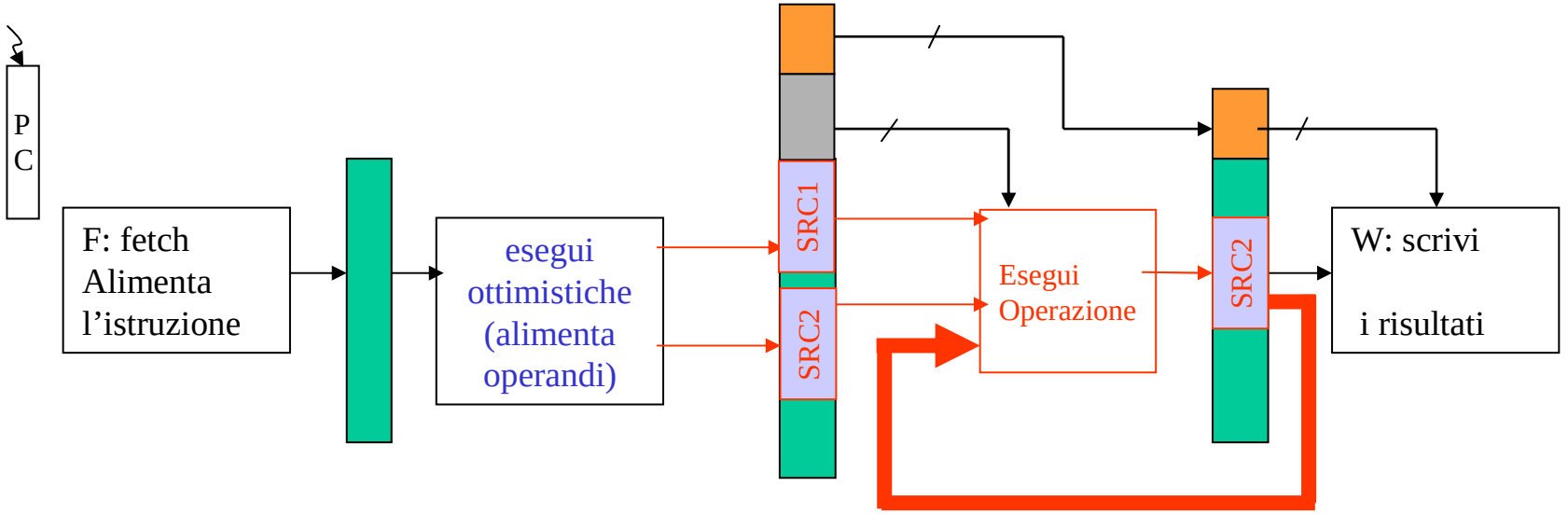
Vediamo un altro caso di propagazione

……

I1: Load R7, (R2)

I2: Add R6, R5, R7

……..



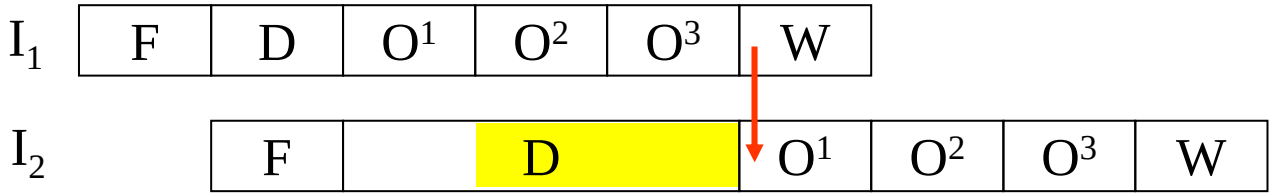
	C1	C2	C3	C4	C5	C6	C7	C8
I1	F1	D1	Read	R7←				
I2		F2	D2	O2 [+]	W2			
I3			F3	D3	O3	W3		
I4				F4	D4	O4	W4	
I5					F5	D5	O5	
I6						F6	D6	

Tempo in cicli di clock

istruzioni

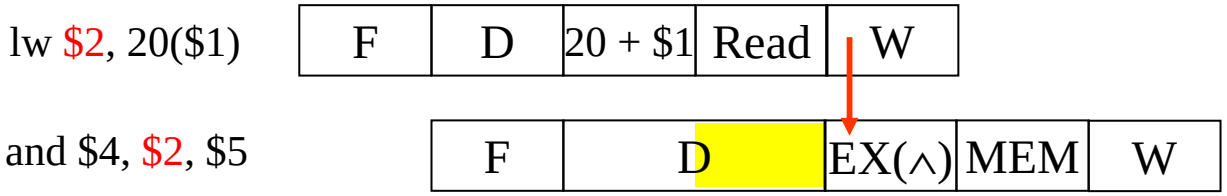
Purtroppo, la propagazione in generale non risolve tutte le criticità sui dati…

Esempio: pipeline più lunga (6 stadi)



Il dato è prodotto in O^3 da I_1 e utilizzato da I_2 in O^1: con propagazione si riduce stallo di due cicli di clock ma è inevitabile stallo di 2 cicli.

Esempio: pipeline a 5 stadi [MIPS] e uso dell'istruzione lw [MIPS]



Nello stadio Decode, si rileva che l'istruzione *and* non potrà gestire la dipendenza con la propagazione ⇒ stallo di un ciclo di clock, poi la propagazione gestirà la dipendenza

IL PUNTO DELLA SITUAZIONE

In base a quanto visto, sono presenti tre unità che controllano la pipeline:

- Unità di controllo: determina nello stadio D i segnali di controllo da assegnare al buffer interstadio D/D+1

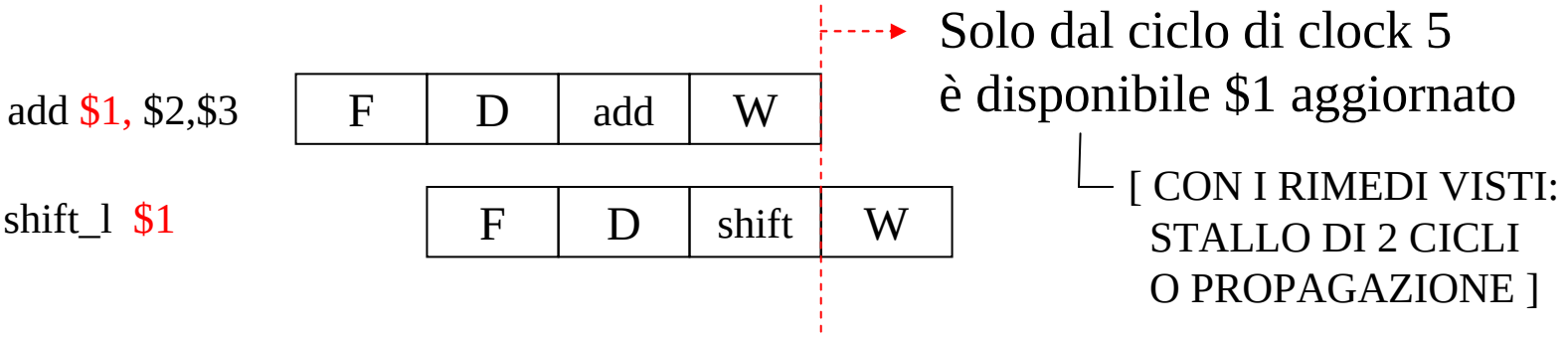
- Unità di propagazione: risolve alcune dipendenze anticipando gli operandi (posta nello stadio in cui si usano, non prima!)

- Unità di rilevamento delle criticità: gestisce le criticità strutturali e le criticità sui dati che non sono risolte dalla propagazione [è possibile identificarle già nello stadio Decode, poiché le istruzioni precedenti sono tutte negli stadi successivi]

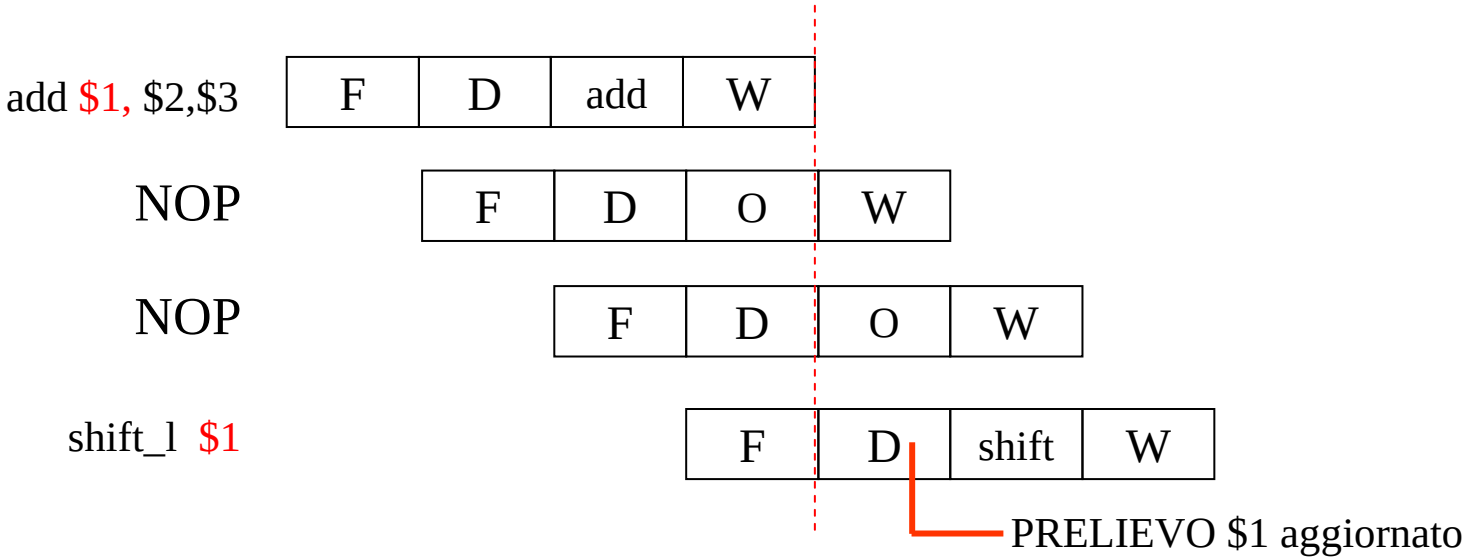
NB: se non si usasse la propagazione l'unità di rilevamento delle criticità sarebbe progettata in modo da riconoscere più criticità: si tornerebbe al caso "rimedio con solo stallo".

Terzo rimedio:
gestione delle dipendenze da parte del software

- Supponiamo che **non** siano previsti meccanismi per gestione propagazione & criticità e consideriamo ad esempio la pipeline a 4 stadi



Il compilatore potrebbe gestire la dipendenza inserendo due istruzioni indipendenti tra add e shift [eventualmente due istruzioni NOP]



Si possono avere diverse possibilità:

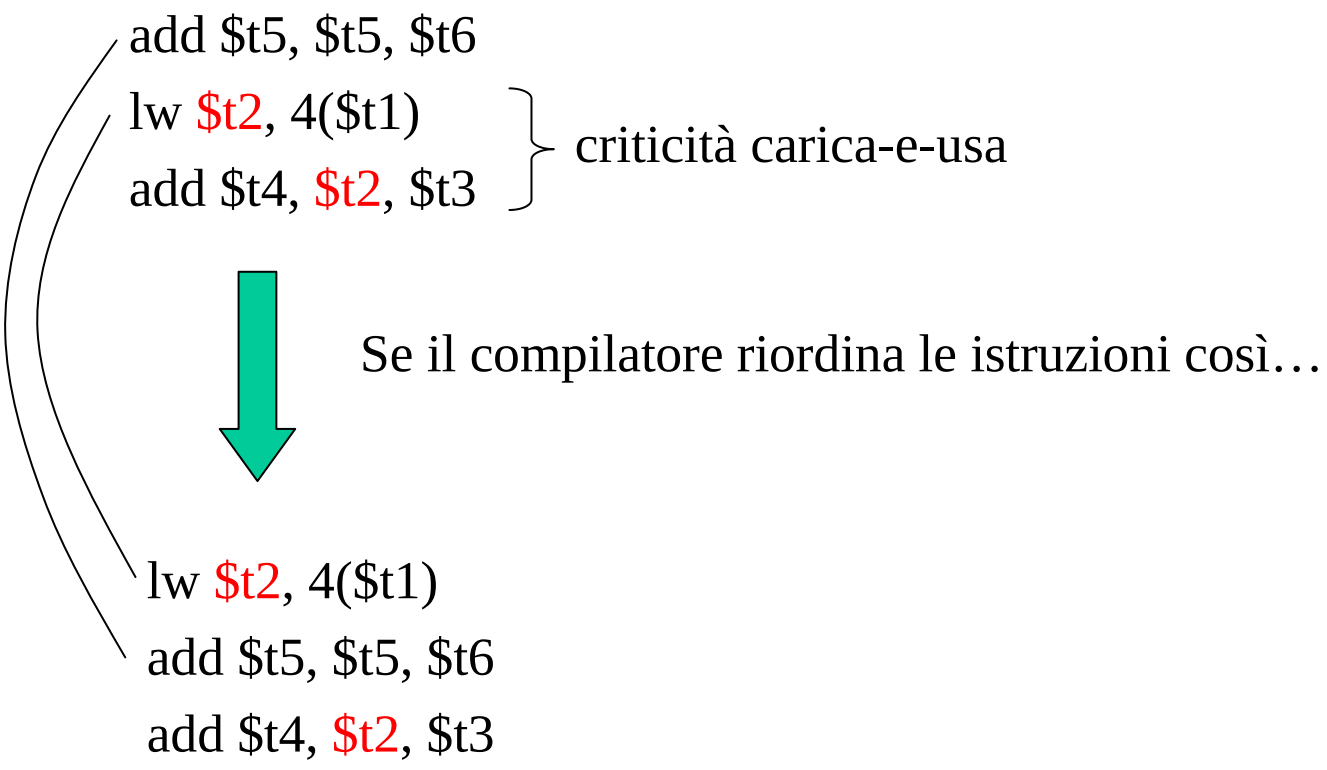
1. Criticità gestita completamente dal compilatore (no stallo pipeline)
2. Gestione criticità via stallo, senza propagazione
3. Gestione criticità con stallo e propagazione
   (che diminuisce il numero di cicli di stallo)

NB: le tre soluzioni sono alternative e ognuna di per sé
basterebbe a garantire il corretto funzionamento della pipeline.

NB: anche nei casi 2) e 3), il ruolo del compilatore è importante:
ordinando le istruzioni (per quanto possibile) in modo da “distanziare”
in modo opportuno le istruzioni dipendenti tra loro, vengono limitati
i cicli di clock in cui la pipeline rimane in stallo.

[vedi esempio successivo]

Esempio ipotetico programma MIPS
(presenza unità gest. criticità + propagazione)



Il risultato non cambia (la seconda add lavora su registri diversi!)
ma la distanza tra le due istruzioni dipendenti ora è aumentata:
la propagazione evita lo stallo!!!

QUADRO RIASSUNTIVO (FINO A QUESTO PUNTO)

- **Controllo pipeline:**
  - sul secondo stadio di “decodifica” [dopo lo stadio di fetch]
  - controllo combinatorio:

    IR $\rightarrow$ decode $\rightarrow$ Registro Interstadio

    (nello stadio di “decodifica” solo ottimistiche)

- **Problemi:**
  - Criticità strutturali (miss di cache + stadio richiede più cicli di clock)

    $\Rightarrow$ stallo “su” uno stadio S (es. stadio di fetch | esecuzione)

    $\Rightarrow$ unità di rilevazione delle criticità (forza lo stallo) sullo stadio S

    Riceve segnali da: stadio S (e registro interstadio)

    + stadi successivi (istruzioni precedenti!)
  - Criticità sui dati

    $\Rightarrow$ unità di propagazione dei dati: propagazione verso uno stadio S:

    propaga dati da uno stadio successivo

    (riceve info su stadio S e successivi!)

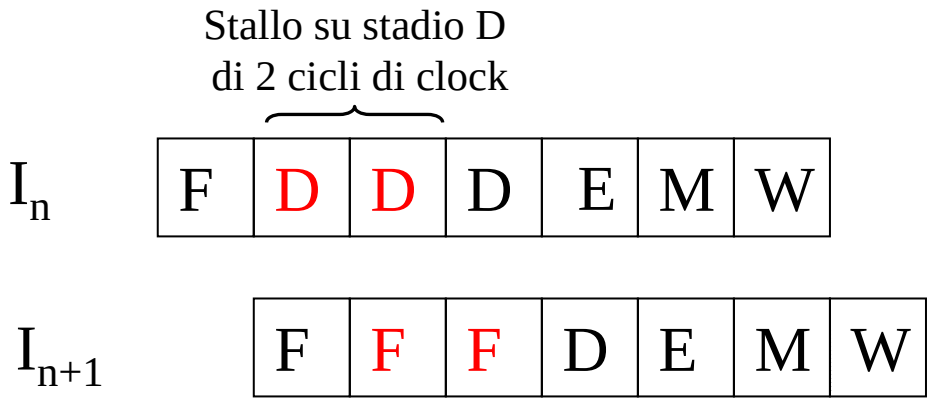
    NB: si propaga sempre verso lo stadio dove viene usato il dato:

    è così possibile “aspettare” più istruzioni che potrebbero produrlo

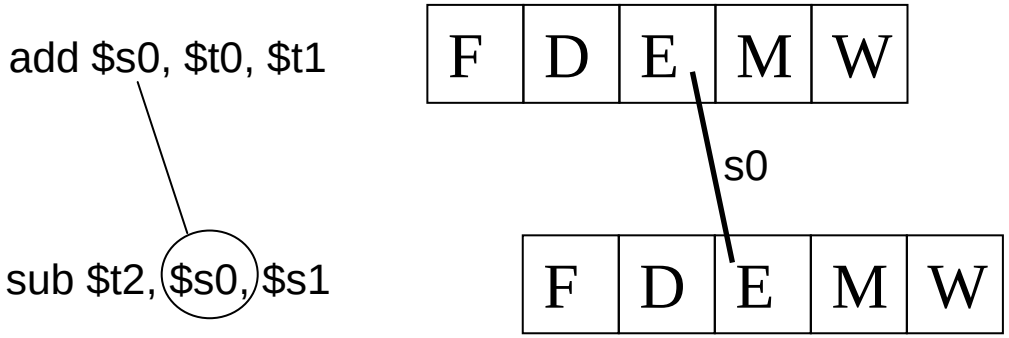
  $\Rightarrow$ UNITA’ DI PROPAGAZIONE “SGRAVA” UNITA’ RILEVAZ. CRITICITA’

Rappresentazione mediante diagramma temporale [a più cicli di clock]

Es. Pipeline a 5 stadi corrispondenti a quelli del MIPS



Es. Rappresentazione della propagazione

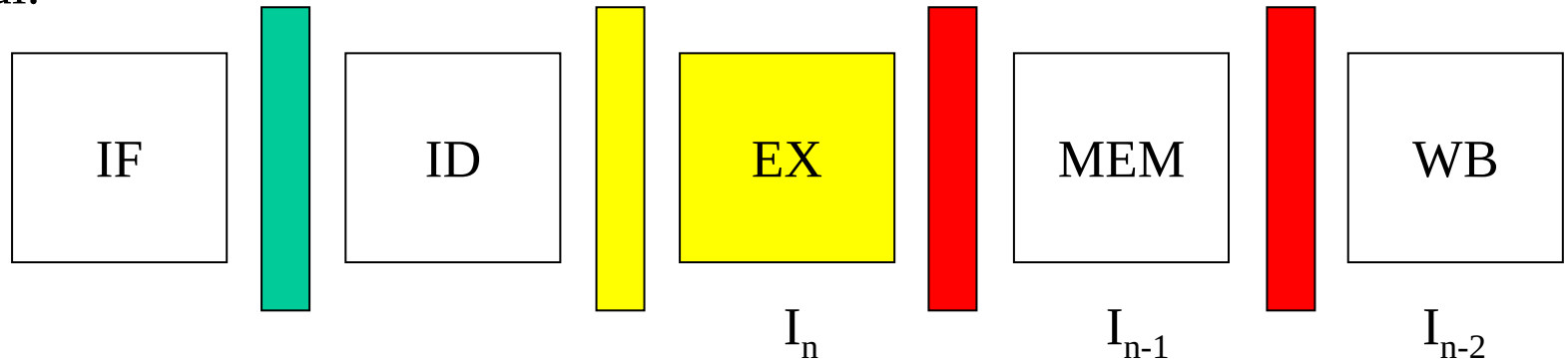


Propagazione
verso stadio E
(da registro E/M)

L'ESEMPIO DEL MIPS

[PAR. 6.4 e 6.5]

5 stadi:



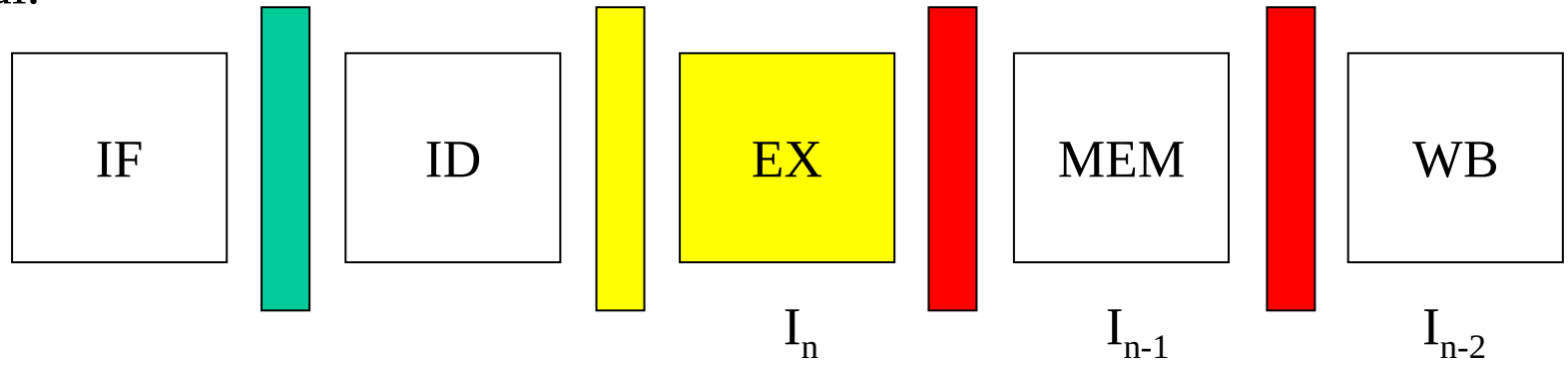
• Si utilizza una unità di propagazione verso lo stadio EX
(ALU per TIPO-R, calcolo indirizzo lw e sw, confronto operandi beq)
in grado di propagare i dati dai registri EX/MEM e MEM/WB.

L'unità di propagazione verso lo stadio EX riceve in ingresso:
- i valori dal registro ID/EX da cui può riconoscere quali operandi usa I_n
- i valori da EX/MEM da cui può riconoscere se I_{n-1} scrive in destinazione (*EX/MEM.RegWrite=1*) e qual è la destinazione di I_{n-1} (*EX/MEM.RegisterRd*)
- i valori da MEM/WB da cui può riconoscere se I_{n-2} scrive in destinazione (*MEM/WB.RegWrite=1*) e qual è la destinazione di I_{n-2} (*MEM/WB.RegisterRd*)

⇒ Se una delle due istruzioni precedenti I_{n-1} e I_{n-2} scrive in un registro destinazione pari ad un registro sorgente (di I_n) questo viene propagato dal registro interstadio

L'ESEMPIO DEL MIPS (continua) [PAR. 6.4 e 6.5]

5 stadi:

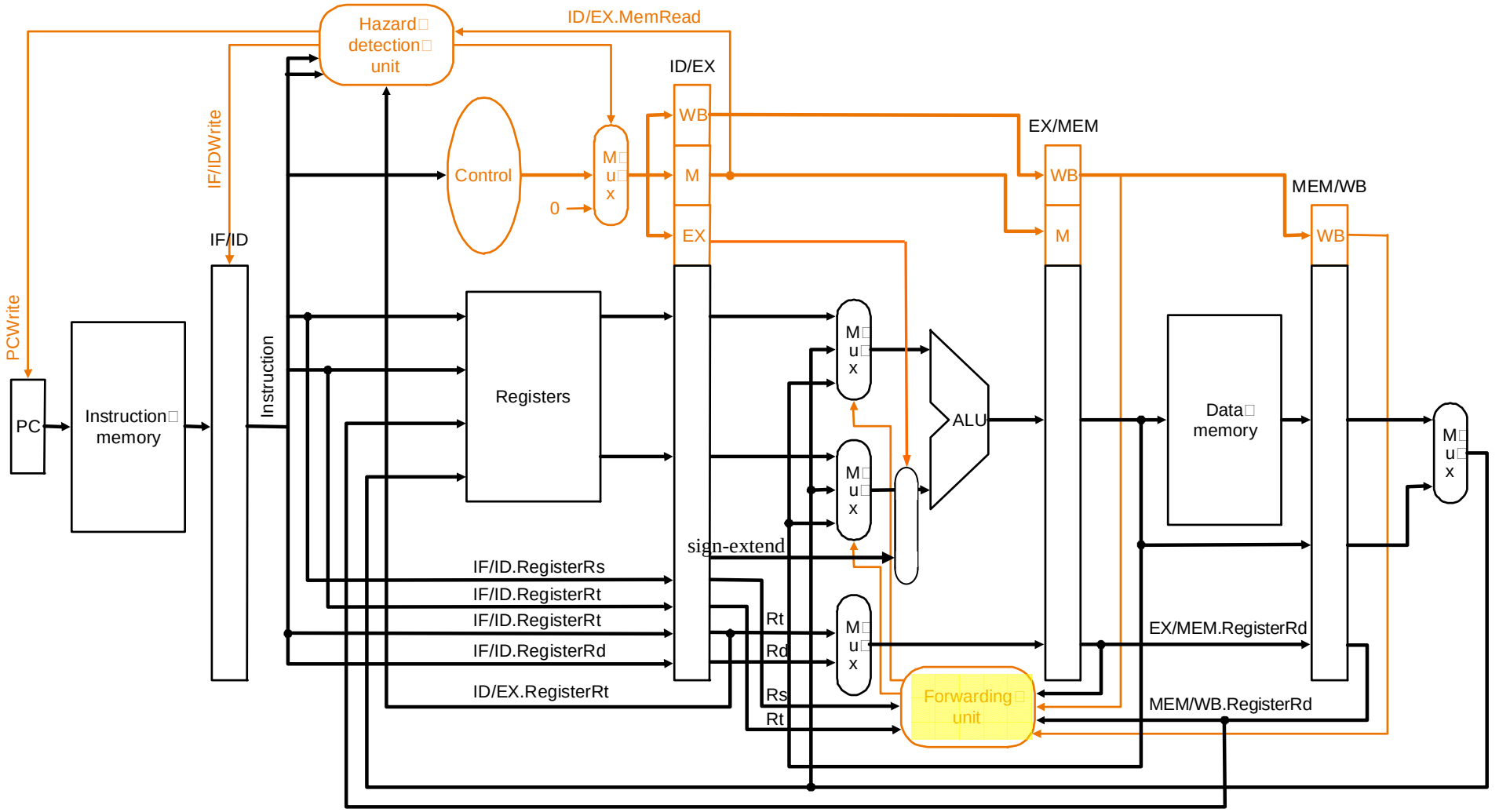


- come visto, il dato da I_{n-2} (ovvero da MEM/WB) va propagato solo se non deve essere propagato anche da I_{n-1} (EX/MEM): l'istruzione più recente vince!

- COMPLICAZIONE DOVUTA AL REGISTRO *$0*:
  nel MIPS il registro *$0* può essere utilizzato come destinazione, ma il suo valore non cambia e rimane 0 (il Register File restituisce sempre valore nullo!)

  ⇒ Se il registro interessato è *$0* (l'istruzione I_n usa *$0* come operando, una istruzione precedente scrive un valore nel registro *$0*) allora il valore letto dal Register File era corretto, mentre quello propagato sarebbe sbagliato (avrei il risultato di una istruzione precedente che in generale è non nullo)

L’unità di propagazione risultante è:



e la condizione da essa imposta:

Controllo del primo ingresso della ALU (“rs”):

```
if (EX/MEM.RegWrite and
     EX/MEM.RegisterRd < > 0 and
     EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 10
```

Propagazione da EX/MEM

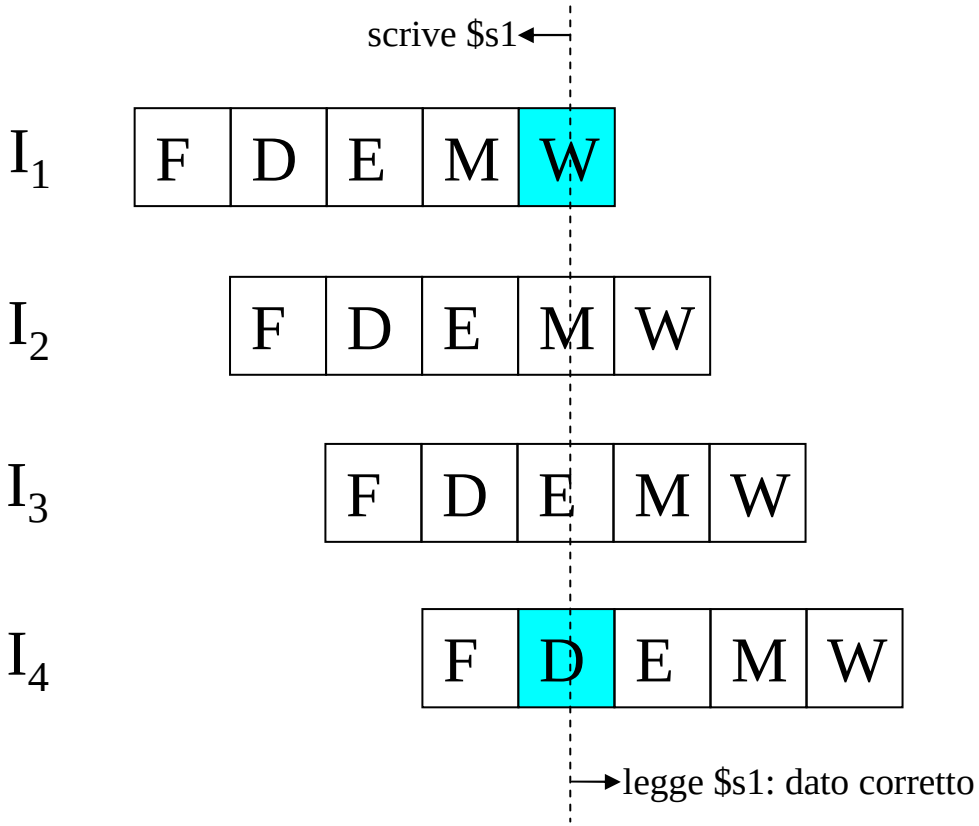
```
if (MEM/WB.RegWrite and
    MEM/WB.RegisterRd < > 0 and
    MEM/WB.RegisterRd = ID/EX.RegisterRs and
    not (EX/MEM.RegisterRd = ID/EX.RegisterRs
          and EX/MEM.RegWrite) )
then ForwardA = 01
```

Propagazione da MEM/WB

… e analogamente per il secondo ingresso (“rt”)…

NB: si è visto che l'unità di propagazione gestisce la dipendenza rispetto a due istruzioni precedenti (infatti propago da due registri interstadio: EX/MEM e MEM/WB). Come gestire la dipendenza dall'istruzione che precede di tre posizioni?

- Altro accorgimento usato nel MIPS:

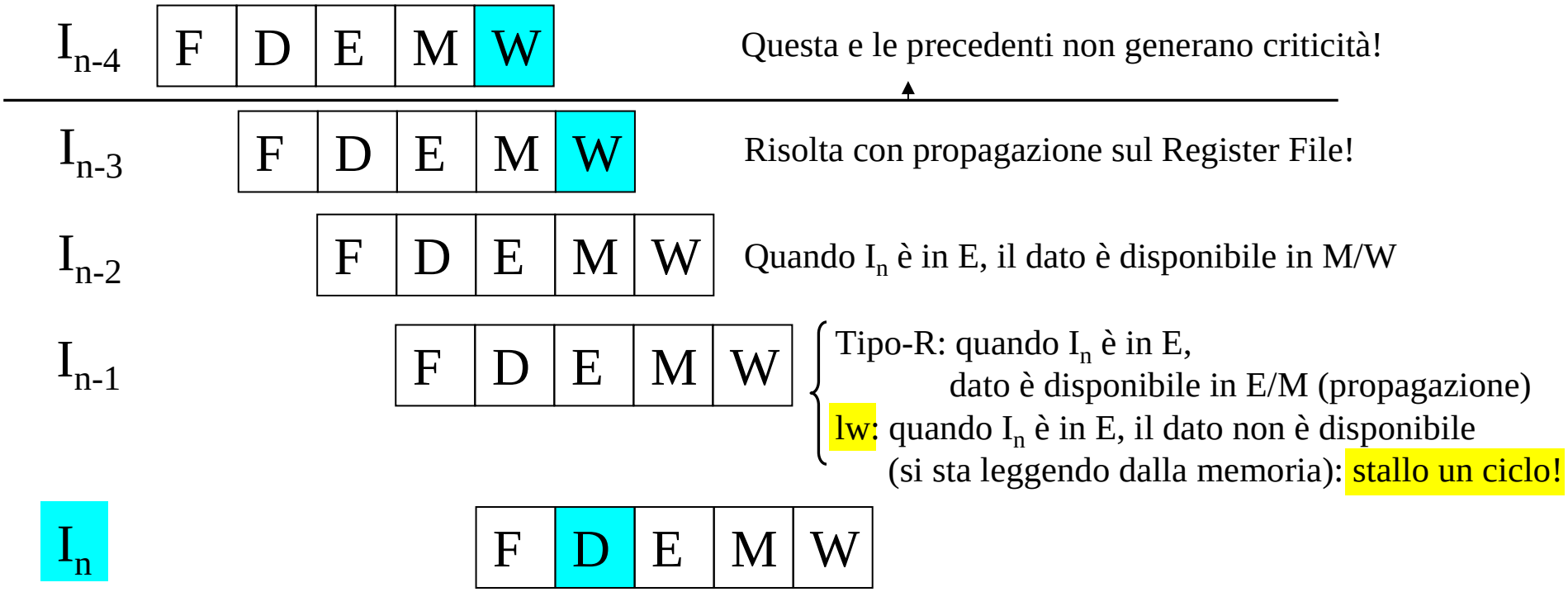


Il register file è progettato in modo che un dato venga scritto nella prima metà del ciclo di clock e venga letto nella seconda metà del ciclo di clock

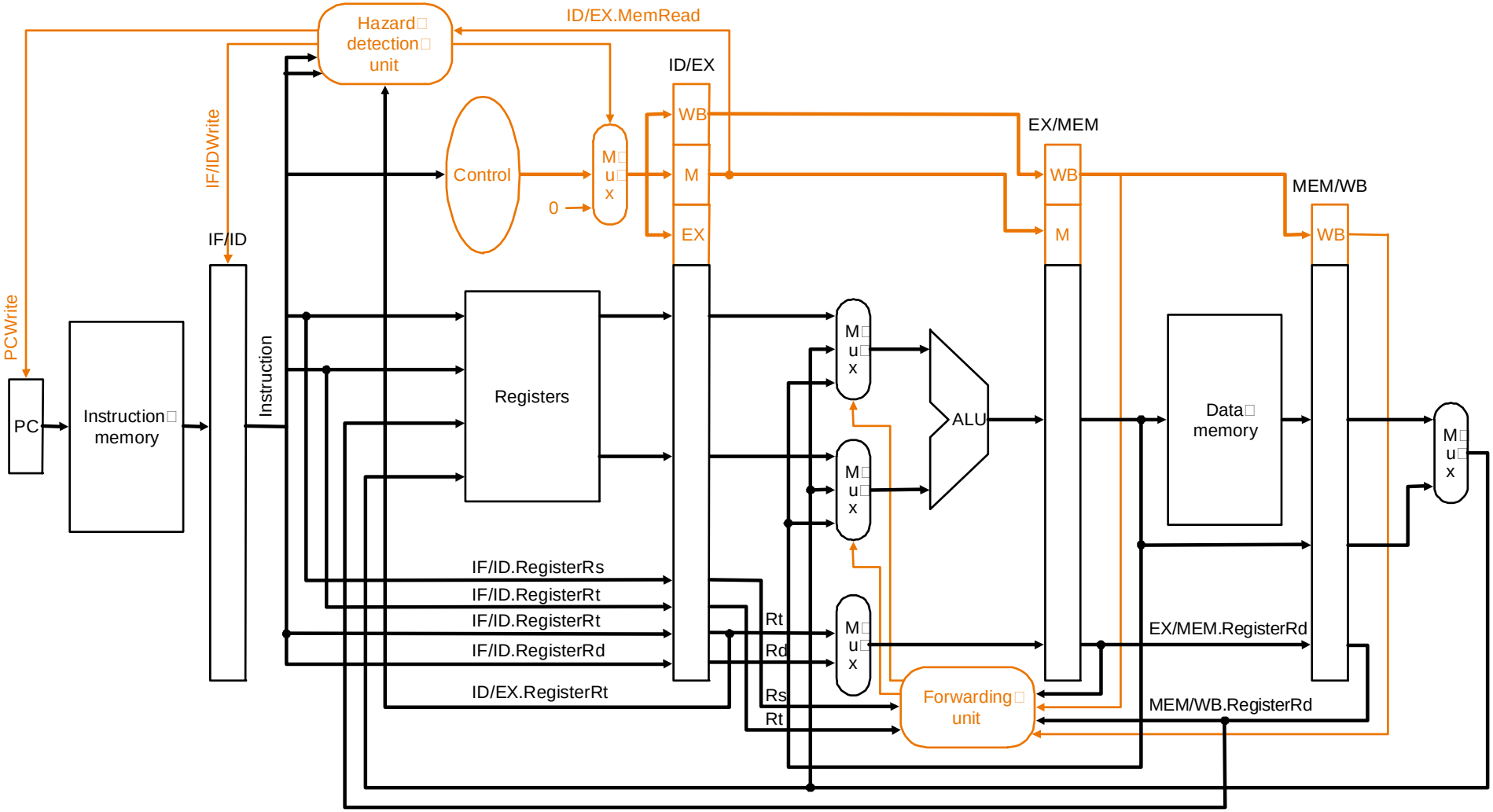
COME VISTO, L'UNITA' DI GESTIONE DELLE CRITICITA' DEVE RISOLVERE CON LO STALLO TUTTO CIO' CHE NON E' GESTITO DALLA PROPAGAZIONE

Vediamo cosa succede quando un'istruzione I_n si trova in D:

NB: - le uniche istruzioni che alterano la destinazione sono TIPO-R e lw
- le istruzioni TIPO-R producono il dato nello stadio E, lw nello stadio M



Unità di rilevazione criticità gestisce “carica-e-usa”



NB: Hazard Detection Unit riconosce se, quando l’istruzione che “usa” il dato è in D, nello stadio EX c’è istruzione lw (l’unica con MemRead = 1) e se questa ha come destinazione un operando sorgente (rs oppure rt): se sì, stallo di un ciclo di clock!

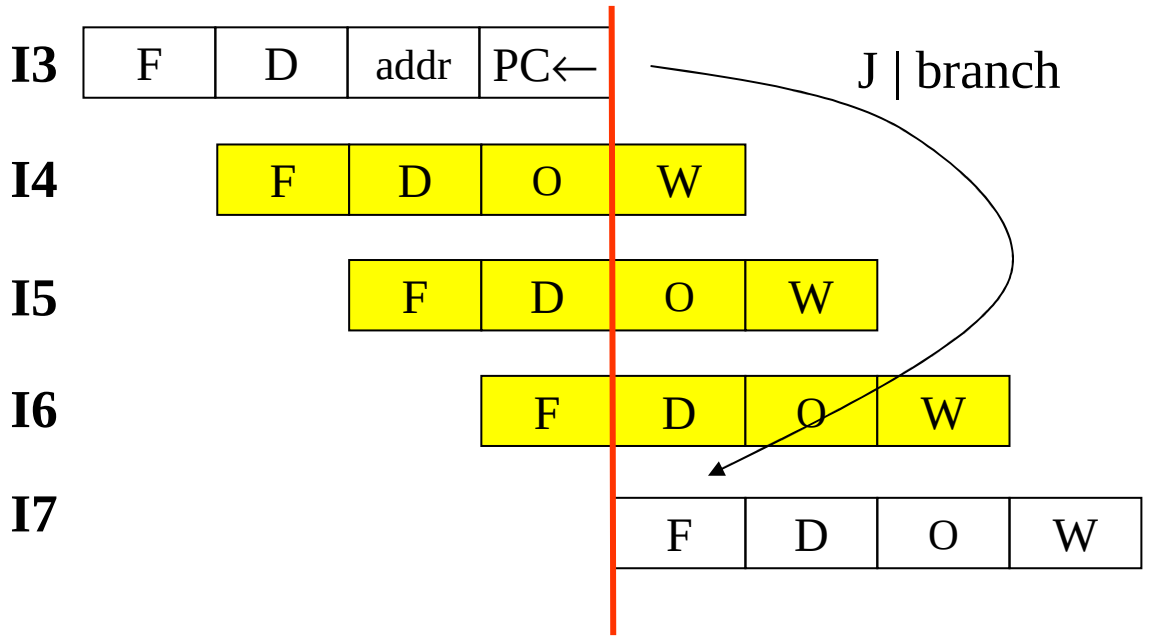
PIPELINE:

CRITICITA’ SUL CONTROLLO

Criticità sul controllo

• L'istruzione di salto viene eseguita in un certo stadio della pipeline (supponiamo il quarto):

……
I3: Jump | j. condiz.
I4: …
I5: …
I6: …
……..



⇨ Se il salto è effettuato, I4, I5, I6 vengono eseguite comunque, mentre l'istruzione a cui si salta viene caricata ed eseguita successivamente

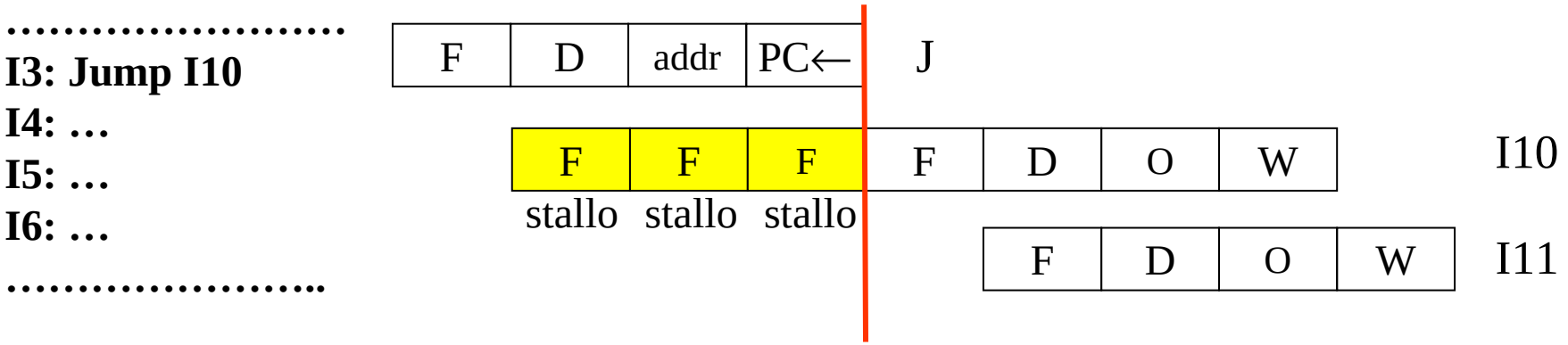
• Due tipi di salti:

- salto incondizionato (jump)
- salto condizionato o diramazione (branch)

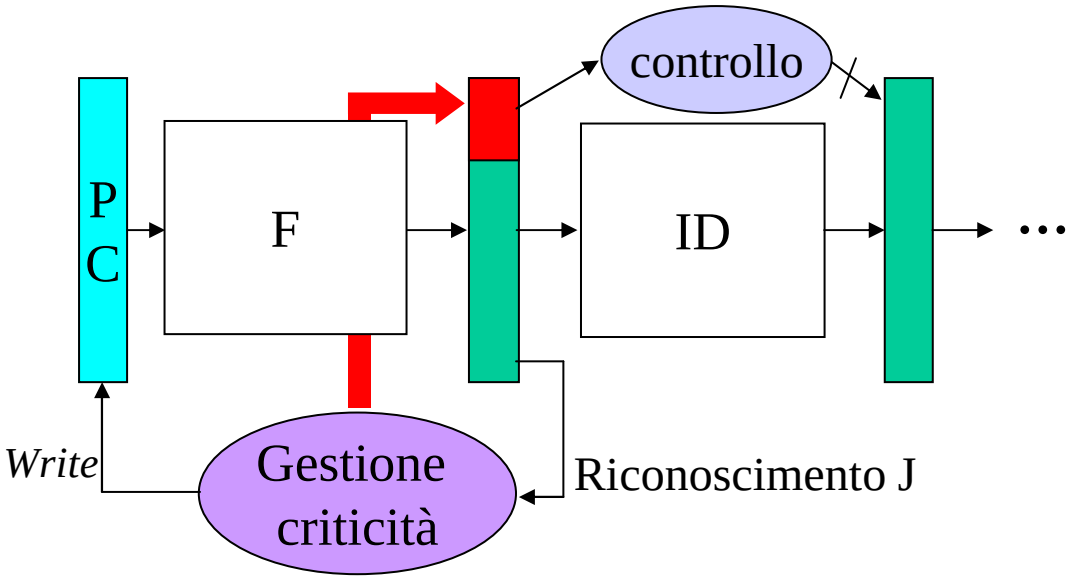
Primo rimedio: stallo della pipeline

Salto incondizionato

• Esempio precedente ⇒ bolla di 3 cicli di clock



• Realizzazione nel modo visto (con S = F e riconoscimento del salto posto in stadio ID)

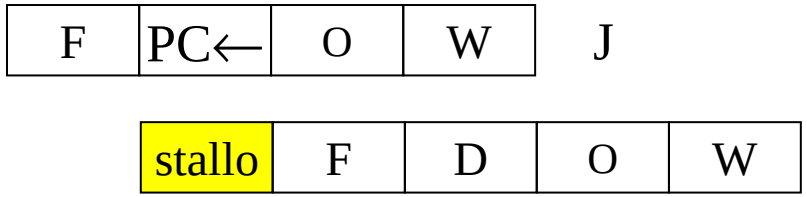


Durante il ciclo di clock:
- si riconosce J
- si evita scrittura PC per
  2 fronti successivi di clock
- si scrive la "bolla"
  [per n=3 fronti successivi]

NB: per limitare il numero di bolle [cicli di stallo della pipeline] è bene che l'esecuzione del salto [aggiornamento PC] sia fatto in uno stadio quanto più possibile anticipato.

➡ Meglio se il numero degli stadi della pipeline è limitato!

In ogni caso, anche anticipando l'esecuzione del salto nello stadio ID, rimane comunque uno stallo:



Salto condizionato

E' ovviamente possibile utilizzare la stessa tecnica; tuttavia, poiché il salto potrebbe non essere effettuato (e quindi eseguite le istruzioni seguenti) si utilizzano tecniche di previsione (vedi poi).

Secondo rimedio: il salto ritardato

Salto incondizionato

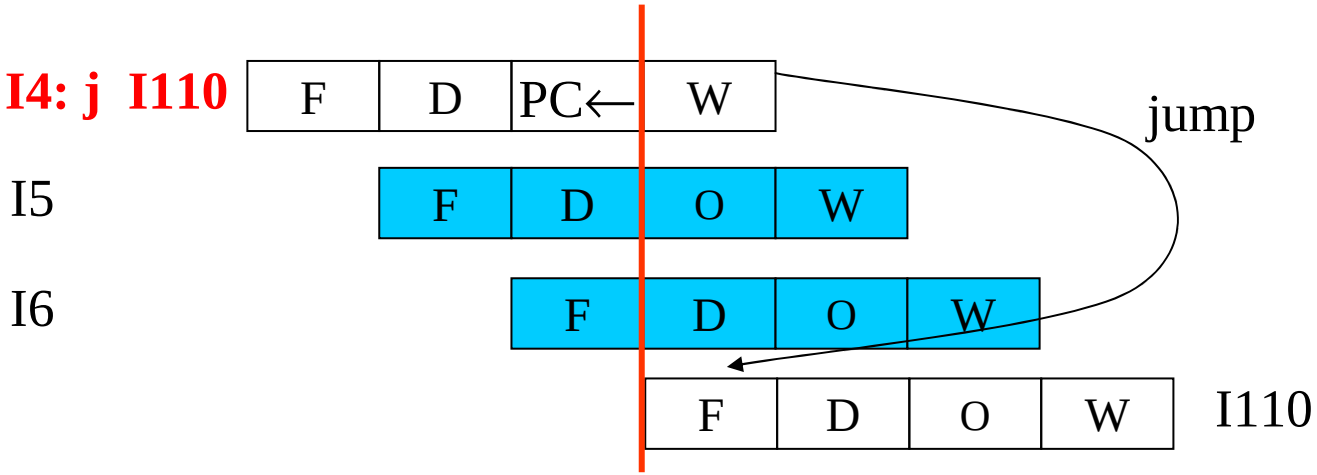
- Come nel caso delle dipendenze tra dati, la gestione può essere demandata interamente al software (compilatore): non presente l'unità di gestione criticità!

Esempio: supponiamo aggiornamento del PC nel terzo stadio (O)
[⇒ lo stallo creerebbe due bolle]

……
I3: lw R1, 100(R3)
I4: add R2, R2, R3
I5: sub R4, R5, R6
I6: j I110
……..

Riordino

……
I3: lw R1, 100(R3)
I4: j I110
I5: add R2, R2, R3
I6: sub R4, R5, R6
……..



- L’istruzione è caratterizzata da uno o più intervalli di ritardo del salto (“branch delay slot”) ovvero posti che il compilatore deve “riempire” con istruzioni che devono sempre essere eseguite.

- Il numero di “branch delay slot” corrisponde allo stadio in cui viene eseguito il salto (1 per il secondo stadio, 2 per il terzo, ecc.).

- Maggiore è il numero di slot, più difficile è il compito del compilatore: in mancanza di istruzioni, è sempre possibile usare l’istruzione NOP.

- Per i salti incondizionati è relativamente facile trovare istruzioni utili. Risulta più difficile per i salti condizionati…

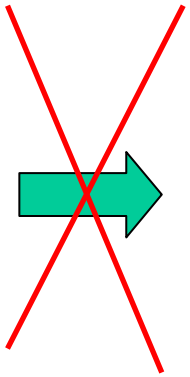
Salto condizionato: SUPPONIAMO CHE SI ABBIA UN SOLO SLOT

1. Il caso più semplice: istruzione presa dal codice precedente

```
……..                          ……..
Dest:  add R2, R1, R3   Riordino   Dest:  add R2, R1, R3
       sub R4, R2, R1   ⇒                 if (R1==0) j Dest
       if (R1==0) j Dest                  sub R4, R2, R1
……..                          ……..
```

Questa strategia non è applicabile se le istruzioni che precedono la diramazione modificano le condizioni su cui si basa il test di diramazione, ad esempio:

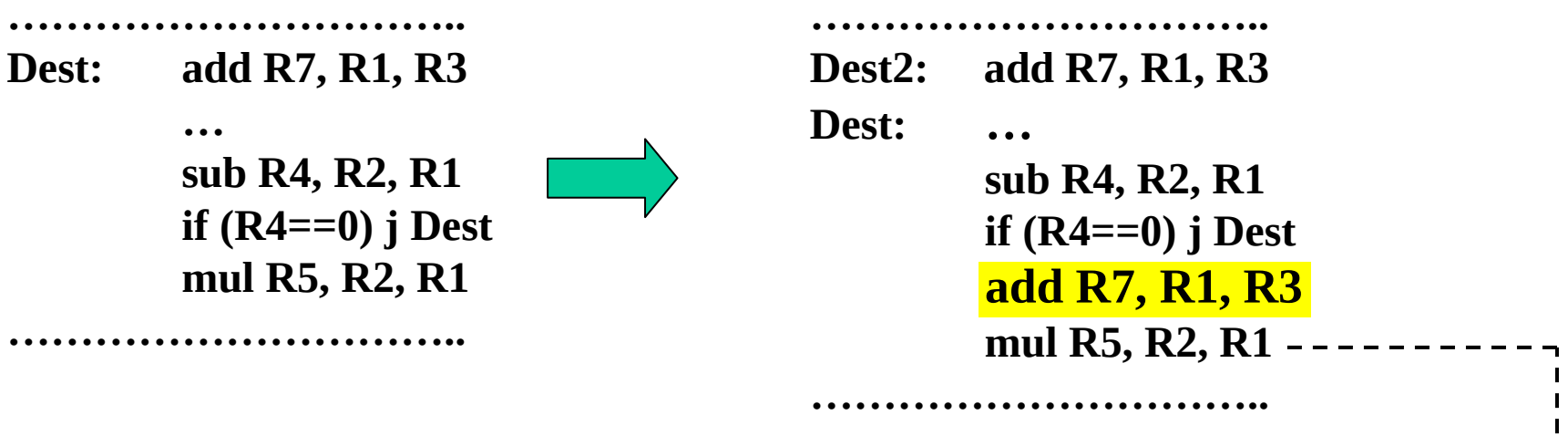
```
……..                          ……..
Dest:  add R7, R1, R3          Dest:  add R7, R1, R3
       …                               …
       sub R4, R2, R1                  if (R4==0) j Dest
       if (R4==0) j Dest               sub R4, R2, R1
       mul R5, R2, R1                  mul R5, R2, R1
……..                          ……..
```



[in tal caso la condizione di salto deve essere successiva a sub]

⇒ Si usa una delle due strategie seguenti…

2. Istruzione presa dal codice destinazione



- I salti esterni al ciclo destinati a Dest devono essere reindirizzati verso Dest2
- E' necessario assicurare che l'introduzione dell'istruzione add per riempire la bolla non alteri la logica del programma: ad esempio, nel caso

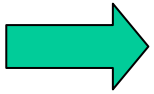
  add R7, R1, R3

  mul R5, R7, R1

  l'introduzione sarebbe errata [add altera l'operando usato da mul]
- Tipicamente, il registro scritto dall'istruzione aggiunta è "temporaneo", usato all'interno del ciclo, mentre non ha visibilità [valore indifferente] all'esterno

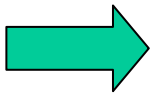
NB: questa tecnica ha senso se la probabilità che il branch venga effettuato è alta, ad esempio nel caso in cui il branch sia la condizione di uscita dal ciclo

```
……..
Dest:   add R7, R1, R3
        …
        sub R4, R2, R1
        if (R4==0) j Dest
        mul R5, R2, R1
……..
```



```
……..
Dest2:  add R7, R1, R3
Dest:   …
        sub R4, R2, R1
        if (R4==0) j Dest
        add R7, R1, R3
        mul R5, R2, R1
……..
```

[se il salto non viene effettuato, la bolla è occupata comunque da un'istruzione inutile, ancorché non dannosa]



Se la probabilità che il salto venga effettuato è bassa, si usa la tecnica seguente…

3. Istruzione “presa” dal codice successivo, ovvero: lasciare tutto com’è!

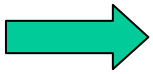
```
……..
        sub R4, R2, R1
        if (R4==0) j Dest
        mul R5, R2, R1
        . . . altre istruz.
Dest:
……..
```

- Se la diramazione non viene effettuata (come è probabile), l’istruzione mul è eseguita correttamente.
- Se la diramazione viene effettuata, l’istruzione mul è eseguita: il compilatore deve assicurare che essa non sia dannosa, come accade (nel caso sopra) se R5 è un registro temporaneo

COMMENTI SULLA TECNICA DEL SALTO RITARDATO

- Richiede collaborazione tra hardware (non dotato dell'unità di gestione criticità per porre in stallo la pipeline) e software (il compilatore deve riordinare le istruzioni tenendo presente i dettagli dell'hardware).

  $\Rightarrow$ Richiede collaborazione progettisti HW e SW
- E' utilizzabile per processori RISC di nuova generazione, mentre è impraticabile se si vuole mantenere la compatibilità con software pre-esistente non compilato per tener conto del salto ritardato (es. Intel: introdotta la pipeline dal 486)
- E' di fatto utilizzabile solo quando il numero di slot è basso (= 1)
- Con l'avvento delle pipeline superscalari e dinamiche (che vedremo) aumenta il numero di slot [in un ciclo cominciano l'esecuzione più istruzioni] ed il salto ritardato è difficilmente gestibile.

   Sono più efficaci le tecniche di previsione dinamica…

Terzo rimedio: tecniche di predizione delle diramazioni

• Idea di base:

- Con lo “stallo” ad ogni salto c’è penalità da pagare in termini di cicli di clock;
- Uso di meccanismo di “predizione” dei salti (salto eseguito o no)
- Se si “prevede” che il salto sia eseguito, l’unità di fetch carica le istruzioni a partire da quella di destinazione, altrimenti dalla posizione seguente;
- Le istruzioni si cominciano ad eseguire (esecuzione speculativa): se la previsione è corretta, non c’è costo; se la previsione è errata, le istruzioni in pipeline vengono scartate e si paga un costo di qualche ciclo di clock.

• Come effettuare la predizione?

TECNICHE DI PREDIZIONE STATICA: realizzata dal compilatore

- Modo più semplice: prevedere sempre che il salto non sia effettuato (se è effettuato le istruzioni già in pipeline vengono scartate)
- Modo più complesso: previsioni diverse a seconda delle istruzioni

TECNICHE PREDIZIONE DINAMICA: realizzata dal processore

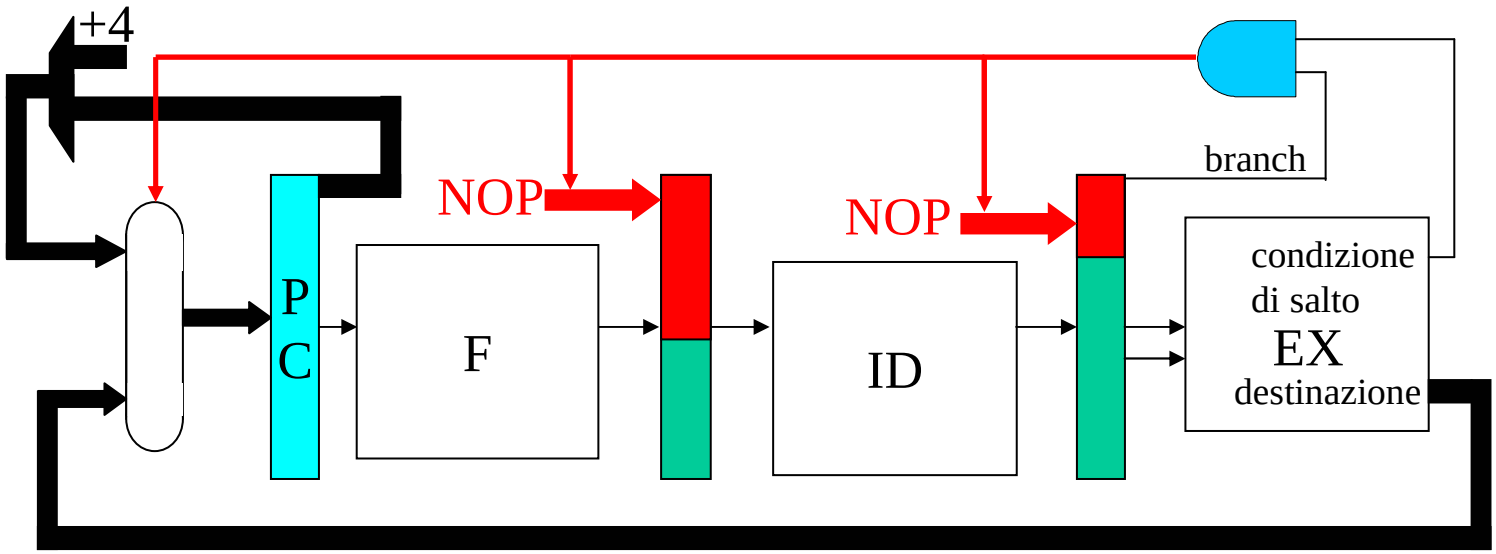
- La predizione è effettuata dal processore dinamicamente: la stessa istruzione in diverse istanze di esecuzione può avere predizioni diverse, sulla base della storia passata (salti in passato effettuati o no)

TECNICHE DI PREVISIONE STATICA

1) Tecnica più semplice: prevedere sempre che il salto non sia effettuato

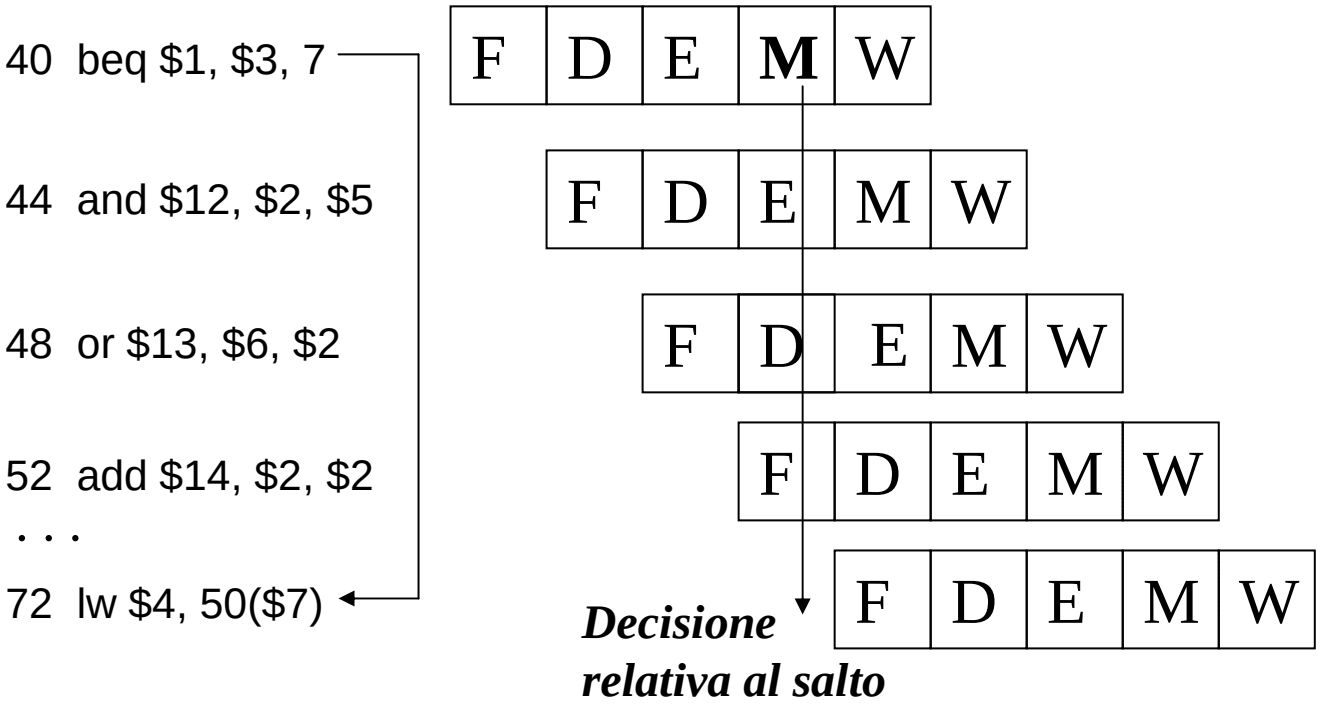
- Se appare un salto condizionato, si continuano a prelevare le istruzioni successive.
- Se la verifica della condizione di salto indica che il salto non deve essere effettuato, tutto procede normalmente.
- Se il salto deve essere effettuato, alla verifica della condizione di salto le istruzioni già presenti nella pipeline (negli stadi precedenti) devono essere scartate e si procede con il fetch a partire dall'indirizzo di destinazione.

Ipotizzando che la verifica + calcolo indirizzo avvenga nel terzo stadio:



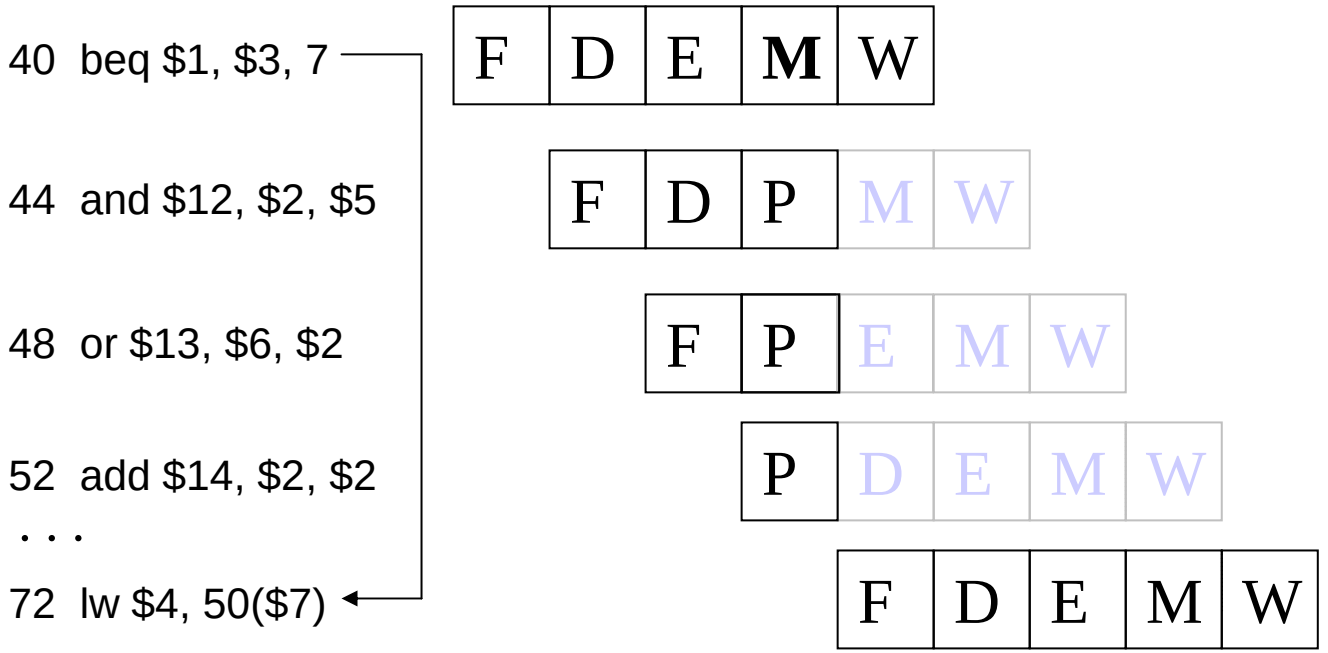
Se il salto è effettuato, le istruzioni caricate nei primi due stadi sono scartate (codici corrispondenti a NOP) e PC ← destinazione.

ES: supponendo che la decisione sul salto sia presa nel quarto stadio



Se il salto viene effettuato, le istruzioni 44, 48, 52 vanno scartate, si introduce un ritardo di 3 cicli di clock

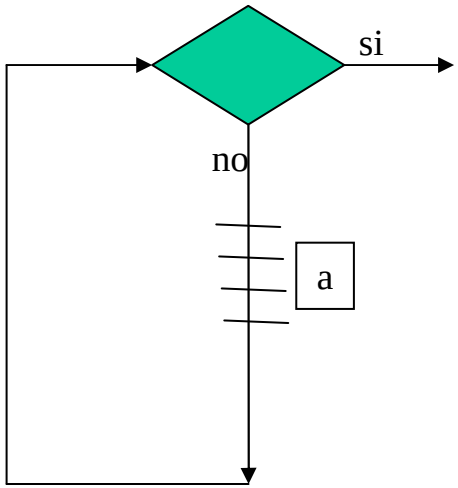
Questo si può indicare nel lucido con cicli di “purge” (P)



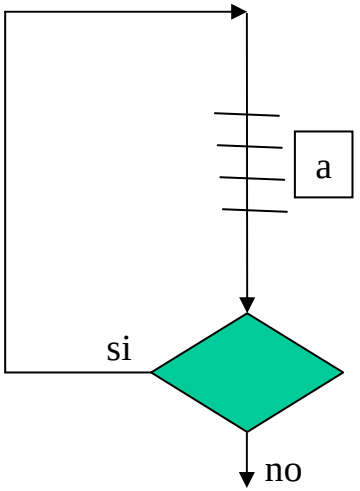
Esempio

Se il branch è la condizione di uscita di un loop posta all'inizio del loop stesso
e i dati fanno sì che il salto non sia effettuato n volte ed effettuato l'n+1
c'è un guadagno: per n volte non pago alcuna penalità!

Viceversa, se la condizione di uscita è alla fine del ciclo, è probabile che il salto sia effettuato per n volte e non la volta n+1: per n volte ho penalità!



VS



TECNICHE DI PREDIZIONE
PIU' COMPLESSE

2) Predizione demandata al compilatore

- la macchina ha diverse istruzioni di branch distinte dal codice operativo: un bit fornisce al controllo la predizione (di salto o di "non salto")
- il compilatore specifica opportunamente la sua predizione nella parola di codice operativo: deduce quale situazione sarà più frequente ed usa l'istruzione opportuna.

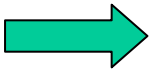
Esempio precedente

- se un salto è all'inizio del ciclo, è probabile non sia effettuato
  $\Rightarrow$ predizione di "non salto" quando il salto è in avanti
- se un salto è alla fine del ciclo, è probabile sia effettuato
  $\Rightarrow$ predizione di salto effettuato quando il salto è all'indietro

NB: ciò che si predice è se il salto è effettuato o no. Prima di saltare bisogna comunque aspettare il calcolo dell'indirizzo di destinazione! Se l'indirizzo di destinazione non è calcolato prima della condizione di salto, vantaggio nullo.

Problema:

- La predizione sarà sempre la stessa ogni volta che si incontra l'istruzione
- Se i dati danno torto al compilatore, alti costi della speculazione (necessario eliminare istruzioni dalla pipeline: costo di qualche ciclo di clock)



Con pipeline superscalari e dinamiche (alta penalità) si usano tecniche sofisticate di predizione dinamica

Esempio

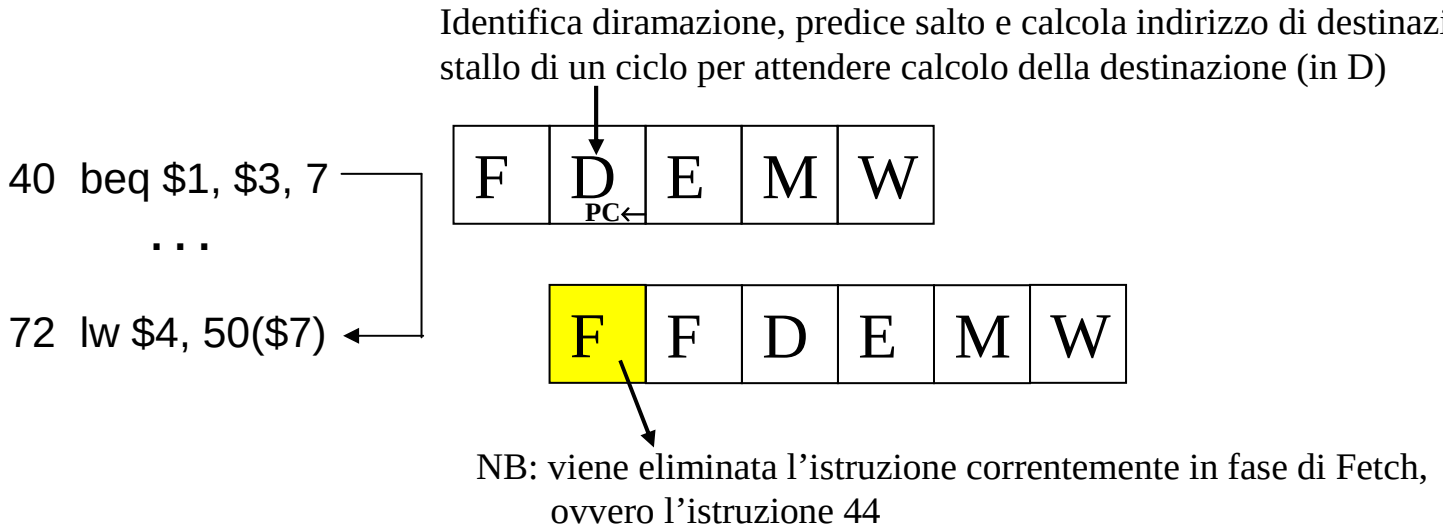
• 5 stadi; supponiamo destinazione calcolata in D, condizione salto in EX

Predizione di “non salto”: si svolge tutto come visto in precedenza

- predizione corretta: nessuna penalità
- predizione scorretta: purge di due istruzioni (penalità di 2 cicli): quando il salto è in EX si eliminano le istruzioni in F e D

Predizione di “salto”: è necessario aspettare (stallo) fino al calcolo destinazione

- caso predizione “di salto” corretta



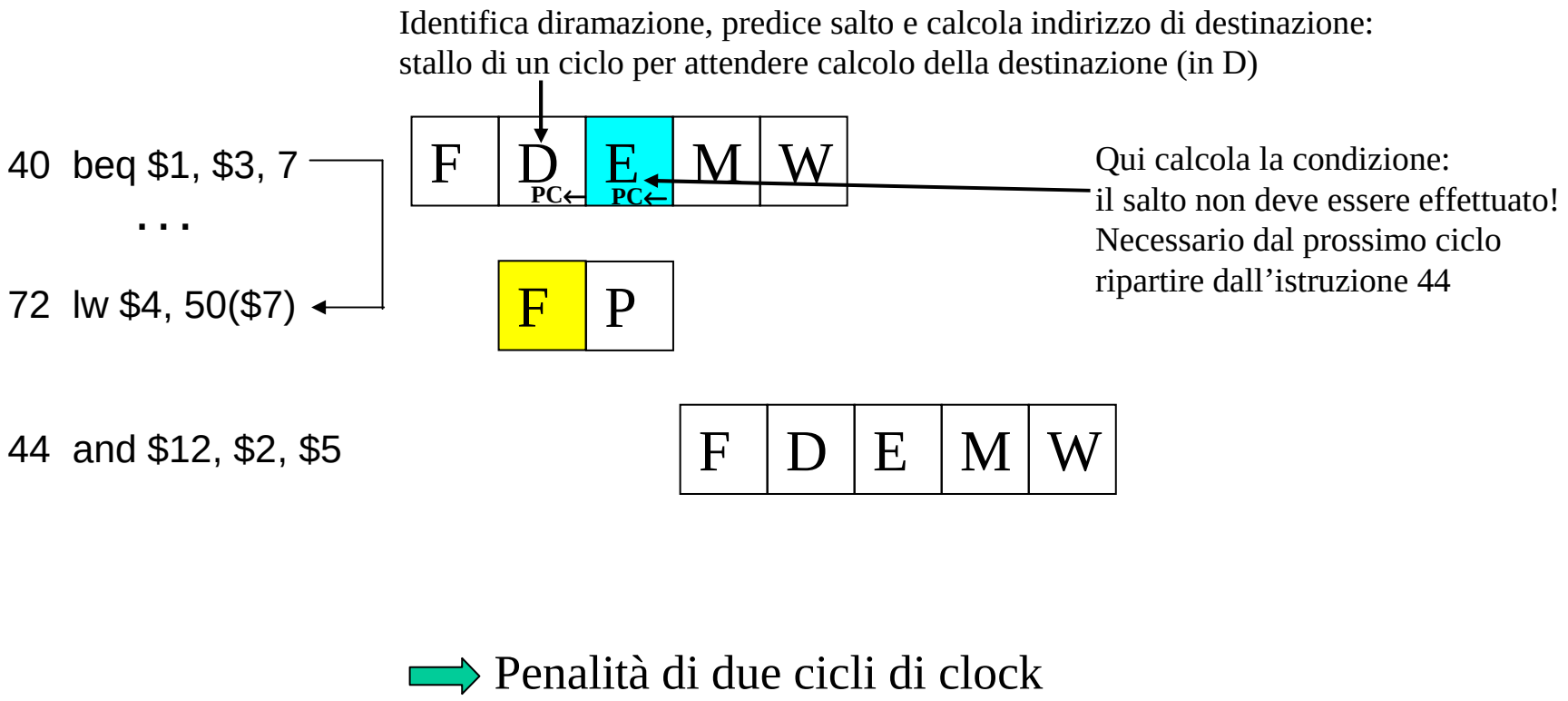
⇒ Penalità di un ciclo di clock

Esempio (continua)

- 5 stadi; supponiamo destinazione calcolata in D, condizione salto in EX

Predizione di “salto”

- caso predizione “di salto” errata



➡ Penalità di due cicli di clock

Inciso riguardante il caso MIPS

Qualunque rimedio si usi, la penalità dipende dallo stadio in cui l'istruzione di salto aggiorna PC.

Es: - stallo: cicli = numero dello stadio – 1
- salto ritardato: slot = numero dello stadio – 1
- previsione "salto non eseguito": eventuale penalità idem

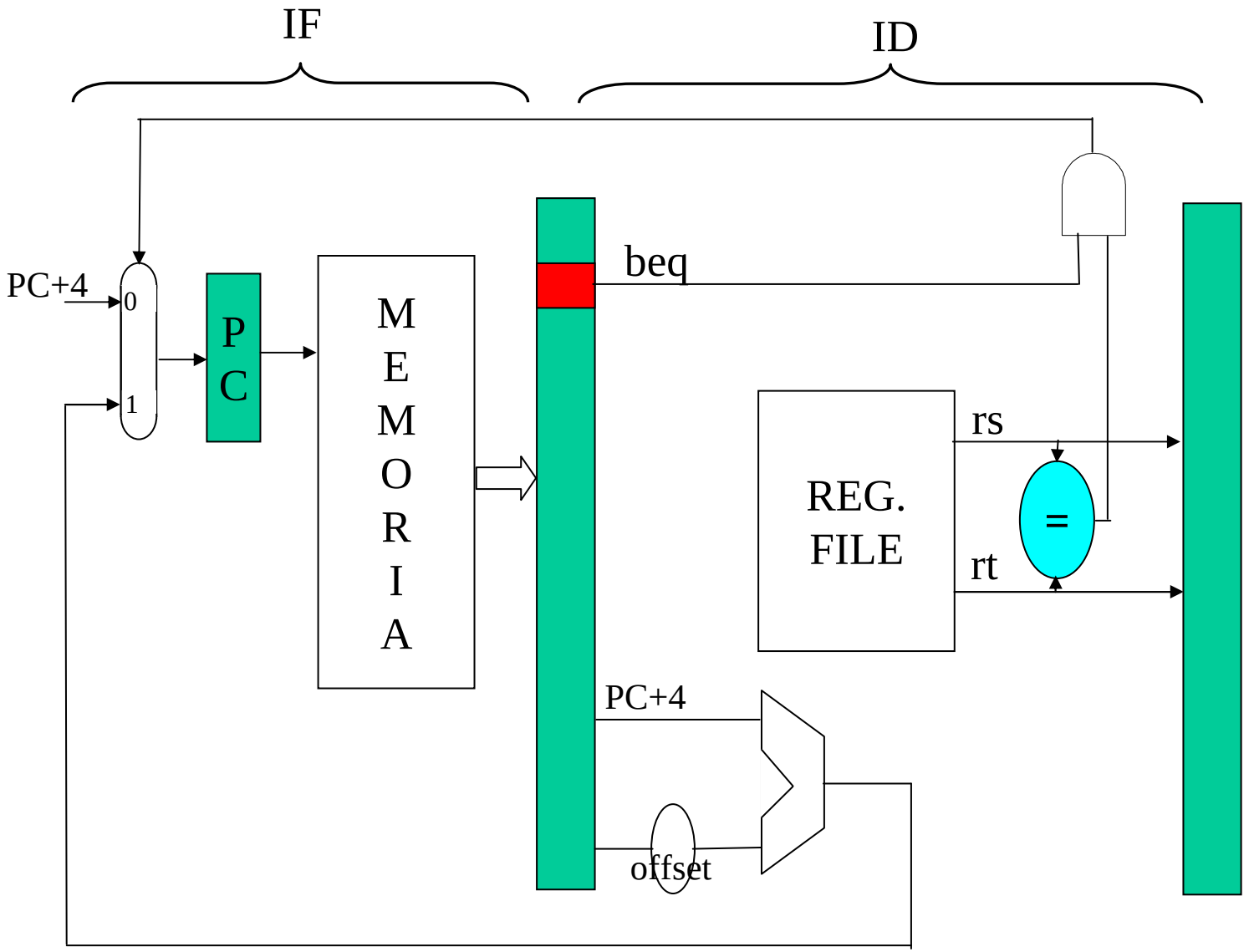
⇒ Conviene anticipare il più possibile l'esecuzione del salto

Idealmente, PC aggiornato nello stadio ID (prima non è possibile!)

Quali attività per i salti condizionati?

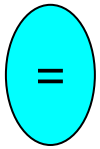
- riconoscimento istruzione di salto (deve essere molto semplice)
- calcolo dell'indirizzo del salto
  (nel caso del MIPS dipende da PC e offset, già disponibili in ID, quindi si possono calcolare facilmente "in parallelo" nello stadio ID)
- test della condizione di salto
  ATTIVITA' PIU' CRITICA: coinvolge uno o più registri letti da Reg. File
  ⇒ E' necessario fare il test a valle del register file: accettabile solo se la condizione di test è semplice.
  ES: beq: semplice test di eguaglianza bit-a-bit!

Nel caso MIPS: Posso modificare la pipeline anticipando l'esecuzione di beq in ID:



Un solo slot per il salto ritardato (o: una sola penalità…)

NB: per fare in modo che il test di uguaglianza sia veloce, non usiamo una ALU



Confronti bit-a-bit con XOR (1 solo se diversi)
+ porta OR che rileva se uno dei bit è diverso

NB2: se vi fossero istruzioni di salto condizionato più complesse, che richiedono ALU per il test di condizione, non sarebbe possibile l'anticipo in ID!

Inconveniente:

Gli operandi rs e rt usati in ID (per il confronto) possono dipendere da istruzioni precedenti!

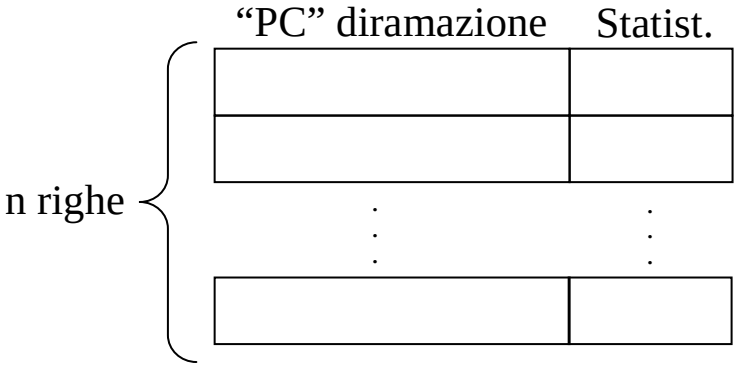
- Utilizzo di una unità di propagazione da registri interstadio successivi
  NB: ma non da ID/EX, visto che in quel registro non è stato prodotto alcun risultato!
- Poiché i dati "servono prima" (in ID) ci sono più situazioni che incorrono in stallo oltre alla criticità carica-e-usa. P.es:
  - dipendenza da istruzione TIPO-R precedente richiede un ciclo di stallo (es. add che modifica registro è in EX quando beq che la segue è in ID)
  - dipendenza da lw due istruzioni precedenti richiede un ciclo di stallo (lw si trova in MEM, occorre aspettare che abbia terminato MEM)

TECNICHE DI PREVISIONE DINAMICA

• L'hardware incorpora algoritmi per stabilire la probabilità che si prenda una via o l'altra in base alla *storia* recente (numero finito di passi).
Per la stessa diramazione, la predizione può cambiare nell'esecuzione del programma

Tabella di predizione delle diramazioni (Branch Prediction Buffer – BPB)

- Memoria che tiene traccia, per ogni diramazione, della storia recente di una diramazione.
- Per risparmiare spazio, gestita come una cache completamente associativa indicizzata dalla parte bassa dell'indirizzo di istruzione del salto.



- Quando si incontra una diramazione, si cerca il suo indirizzo nel BPB (in realtà solo bit meno significativi) e si aggiorna la statistica in base a esito della diramazione (salto o non salto)
- Se l'indirizzo non è presente l'inserimento e l'eliminazione di una diramazione è gestita secondo una certa politica

Es: con politica LRU: BPB tiene traccia delle ultime n distinte diramazioni incontrate.

Di solito si usano 5-6 bit per rappresentare l'indirizzo…

NB: è possibile che diramazioni distinte (i cui indirizzi hanno bit meno significativi uguali) accedano alla stessa riga del BPB

→ In lettura: utilizzo di una previsione di un'altra diramazione

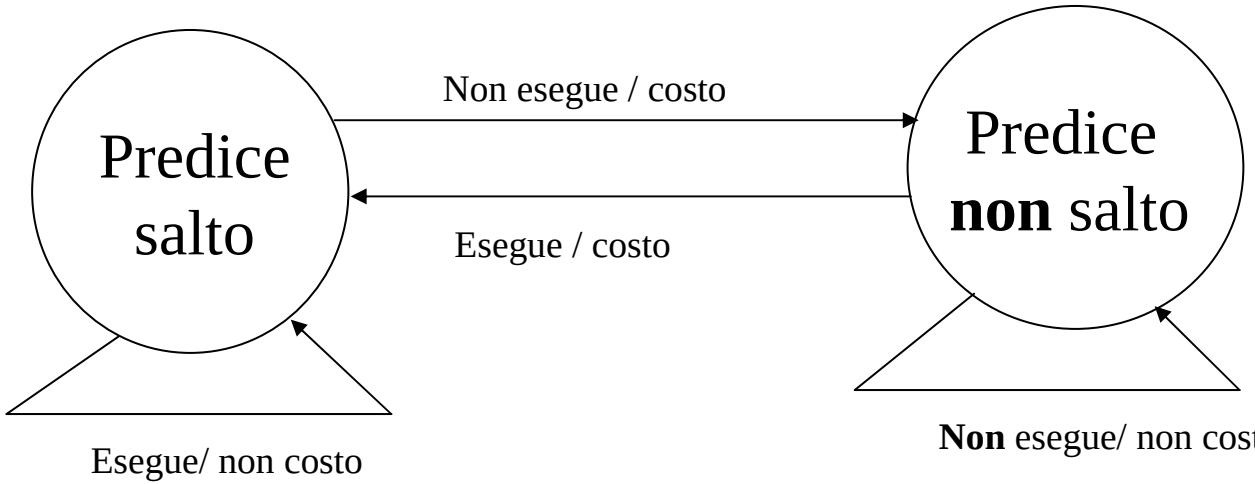
In scrittura: si "mescolano" le storie di diverse diramazioni

→ Diminuzione della accuratezza della previsione

Cosa è contenuto nel campo "statistica"?

Un certo numero di bit che indicano se le ultime volte che è stata incontrata la diramazione il salto è avvenuto oppure no…

La soluzione più semplice: memoria a un passo

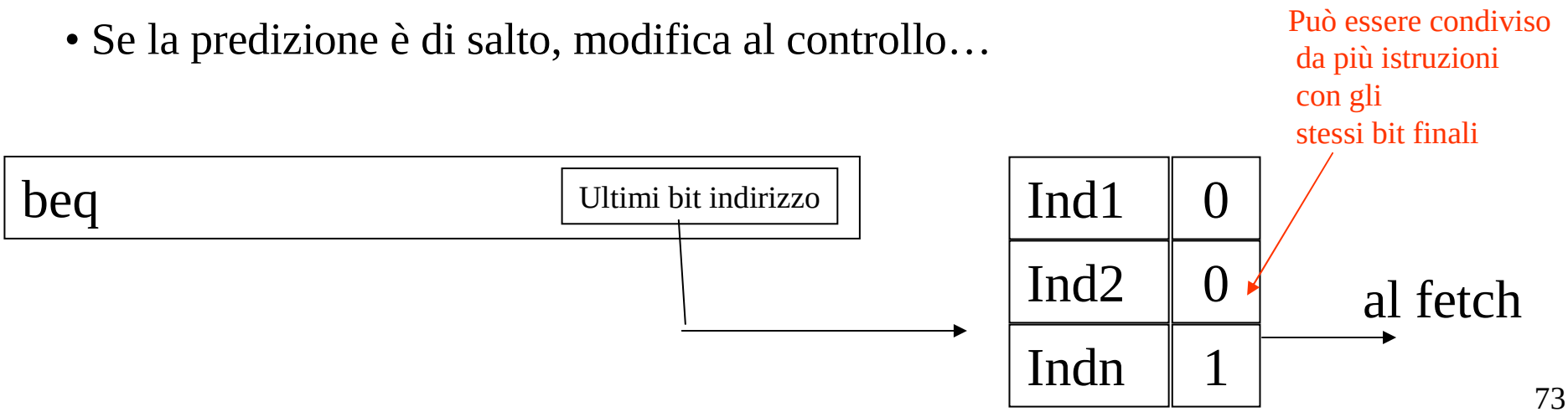


1 bit indica se ultima volta il salto è stato effettuato oppure no.

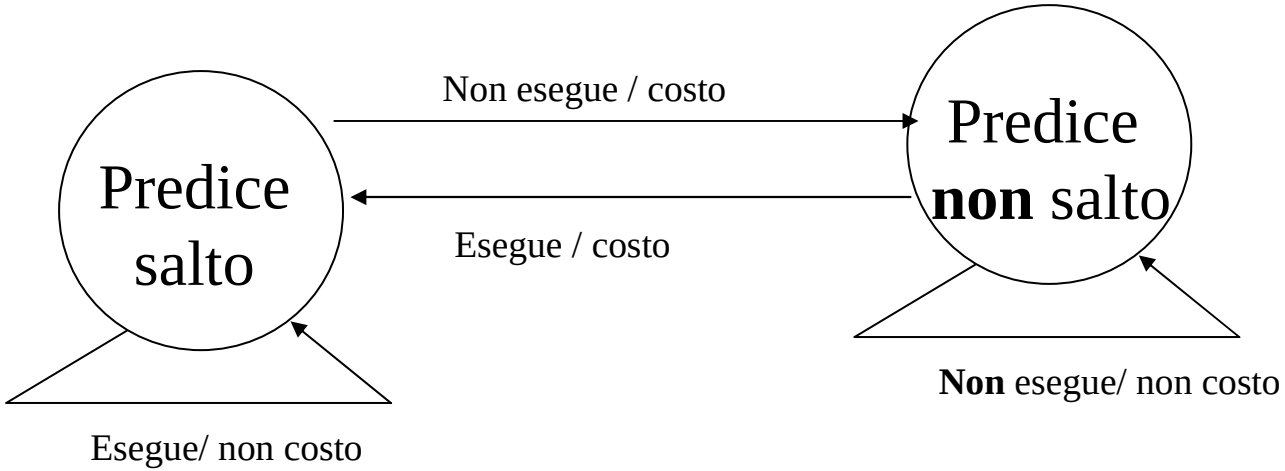
NB: “non costo” = predizione corretta, che comunque come visto può dar luogo a penalità

Modifica al controllo

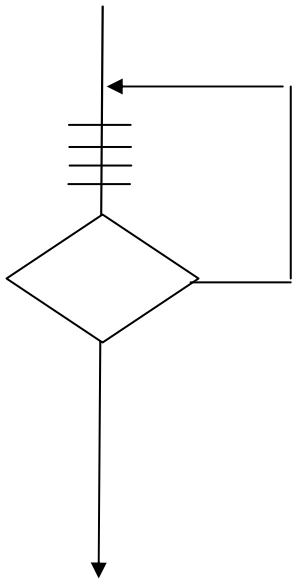
- Nella fase di fetch della diramazione, si accede a BPB; predizione valutata;
- Se la predizione è di salto, modifica al controllo…



Prestazioni di questa prima realizzazione



Nel caso di un ciclo, a regime risulterà scorretto due volte, al primo e ultimo controllo

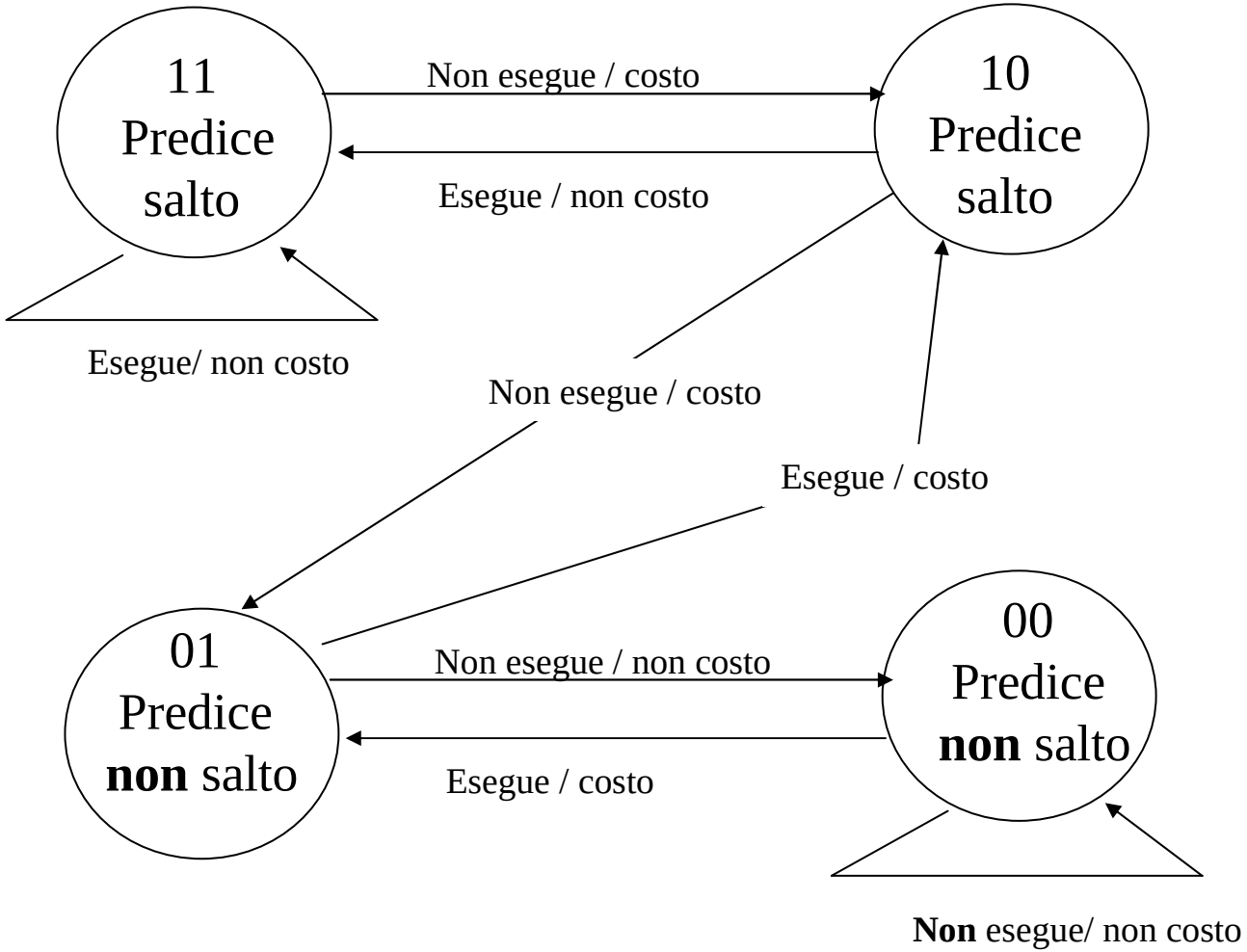


Es: 10 iterazioni del ciclo ogni volta che si incontra
(salto effettuato per nove volte, la decima esce):
- la prima volta predice non salto
(la volta precedente era uscito dal ciclo!)
$\Rightarrow$ errore
- 8 volte predizioni giuste
- ultima volta predice salto $\Rightarrow$ errore

Prediz. corretta nell'80% dei casi, quando il salto è eseguito nel 90% dei casi!

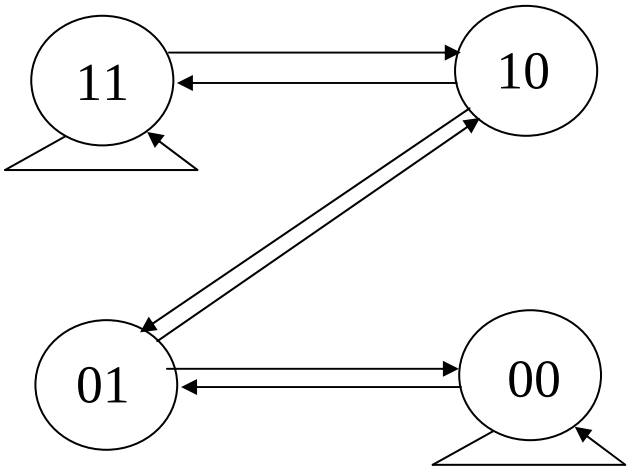
Un’altra soluzione: più memoria

- Due bit nella tabella per ricordare la storia passata: esiti degli ultimi due incontri
- Sono necessarie due predizioni scorrette per cambiare predizione

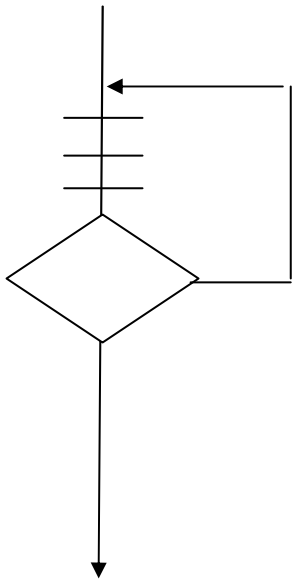


Predizione con <u>contatore a saturazione</u> a quattro stati:
incrementato quando salto effettuato, decrementato altrimenti.

Costi di questa realizzazione



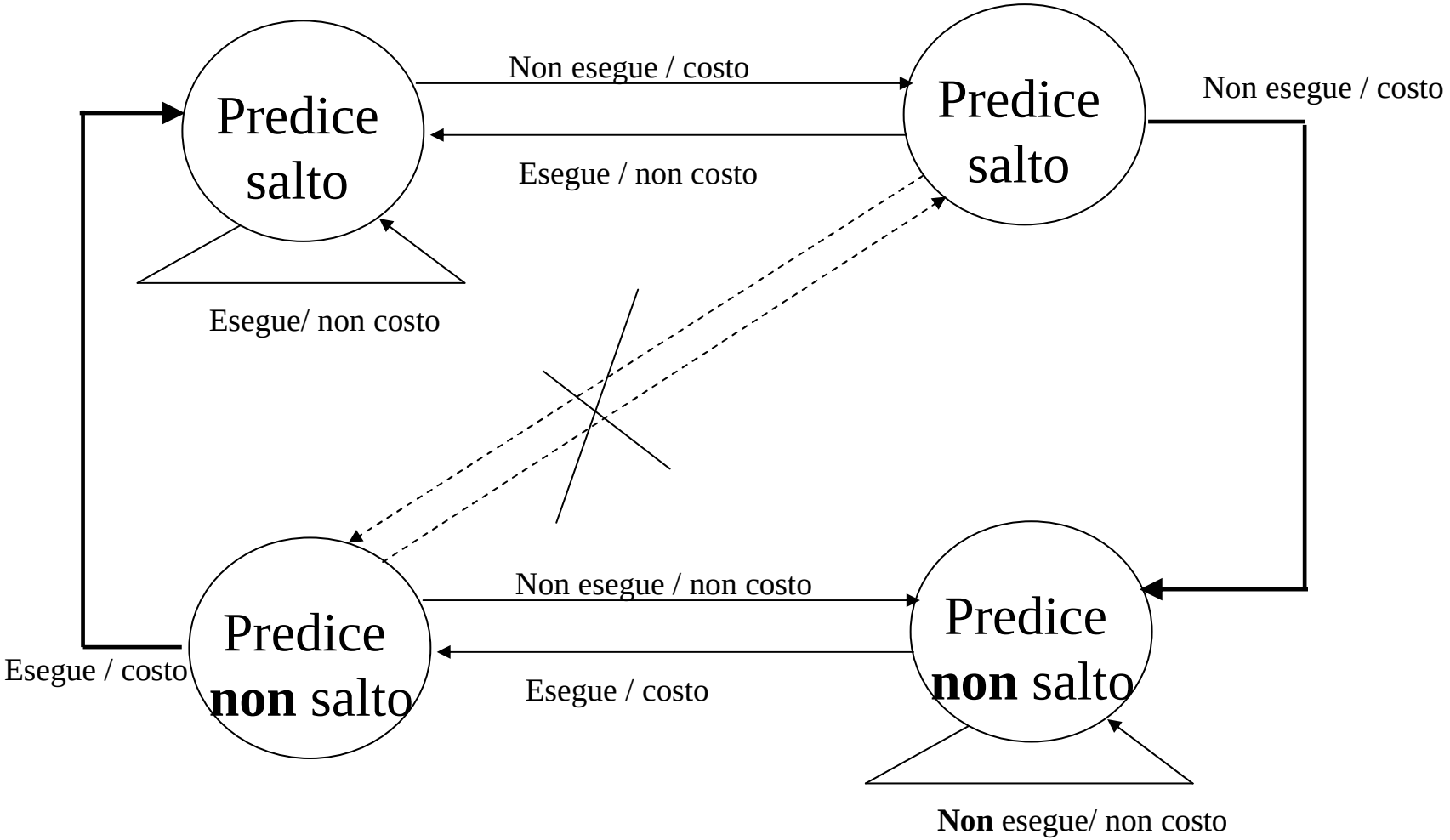
Cfr. ancora nel caso di un loop



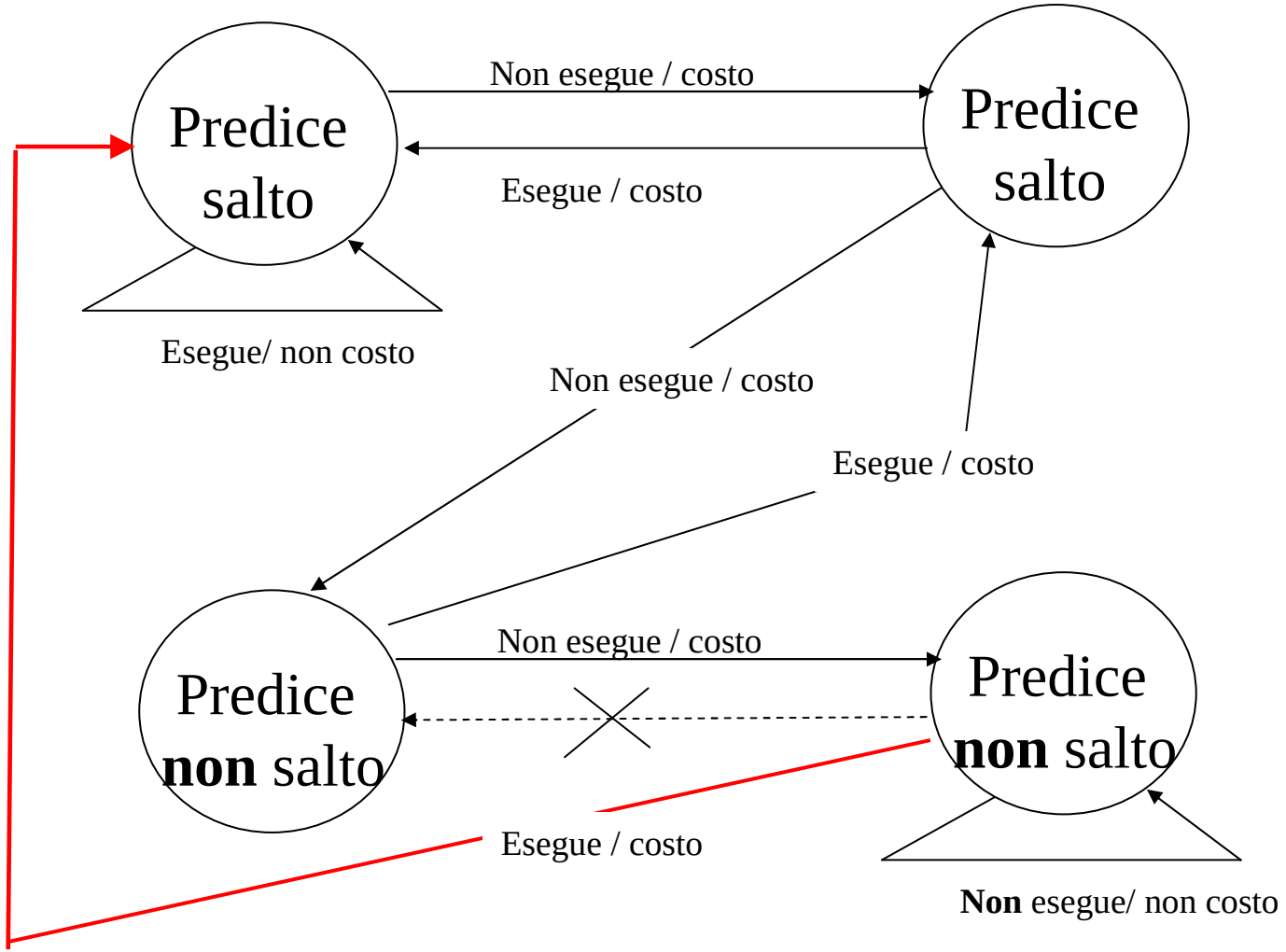
- Dopo qualche iterazione del ciclo, si stabilizza nello stato 11 (ultime 3 iterazioni: salto)
- All'uscita dal ciclo, stato 10 (=2), ma al successivo incontro si stabilizza di nuovo nello stato 11

→ A regime risulterà scorretto una volta (all'ultimo controllo): dimezza costi. Accuratezza 90%, pari a frequenza salto!

Sono possibili varianti…



Cfr. anche “pentium asimmetrico” in cui dallo stato di “non salto stabile”
si passa direttamente allo stato di “salto stabile”
(ottenuto dal primo schema introducendo un’asimmetria)



INDIRIZZO DI DESTINAZIONE NEL SALTO PREDETTO

Previsione di salto nella fase di fetch dell'istruzione di diramazione

$\Rightarrow$ se predetto, fetch istruzioni a partire da indirizzo destinazione

➡ Problema: l'indirizzo destinazione non è noto finché non si calcola!

➡ Il vantaggio è limitato se l'indirizzo di destinazione è calcolato in uno stadio avanzato! Occorre anticipare il più possibile il calcolo (vedi MIPS) e comunque il vantaggio si ha solo nel risparmio di tempo per calcolare la condizione di diramazione.

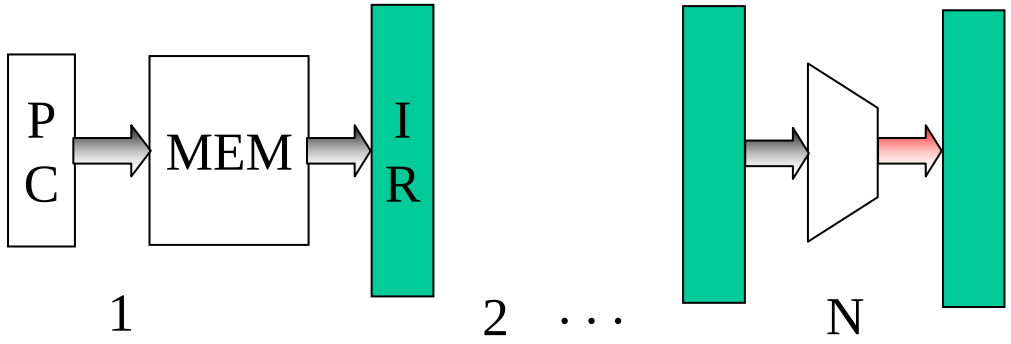
In ogni caso, vantaggio nullo se destinazione non è calcolata prima della condizione

Si può procedere come nel caso visto per la predizione statica:

- Predizione di non salto:
  - corretta: nessuna penalità
  - scorretta: salta quando ha calcolato la condizione di salto (e quindi anche la destinazione calcolata in uno stadio precedente) eliminando le istruzioni negli stadi precedenti (penalità)
- Predizione di salto: stallo finché non si è calcolata la destinazione
  - corretta: penalità associata all'attesa del calcolo della destinazione
  - scorretta: calcolata la condizione di salto, occorre eliminare le istruzioni negli stadi precedenti (penalità)

NB

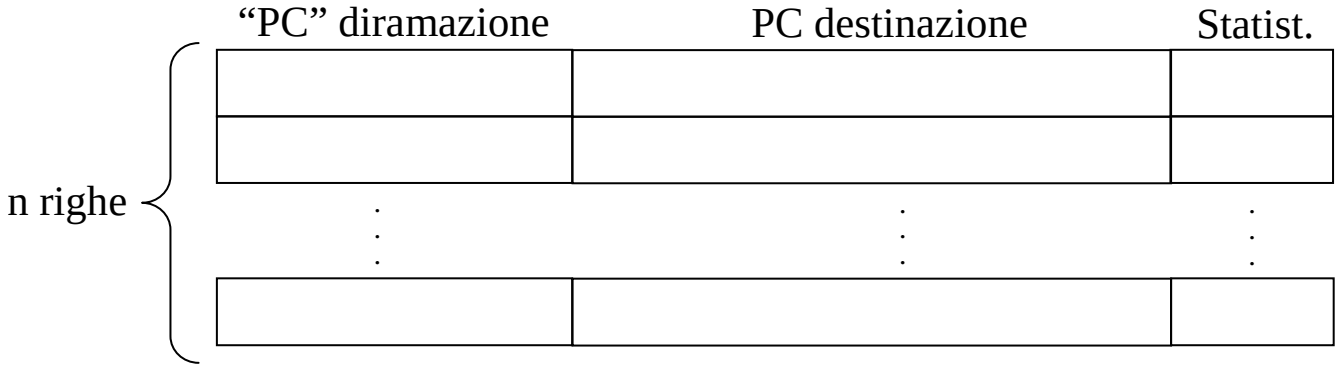
- Ricordare che non possiamo aumentare il cammino critico [no $\uparrow T_{clock}$], quindi se l'indirizzo è calcolato in uno stadio N [all'uscita dell'ALU] non può essere utilizzato per il prelievo in corso nello stadio di fetch! Quindi, calcolo in stadio nr. N $\Rightarrow$ N - 1 "bolle" (es: calcolo in stadio D $\Rightarrow$ 1 bolla)



Modificare indirizzo in fase di fetch richiede serie ALU-memoria: non praticabile

UNA SOLUZIONE: USARE UN CAMPO PER PREDIRE DESTINAZIONE

Tabella destinazione delle diramazioni (Branch Target Buffer – BTB)

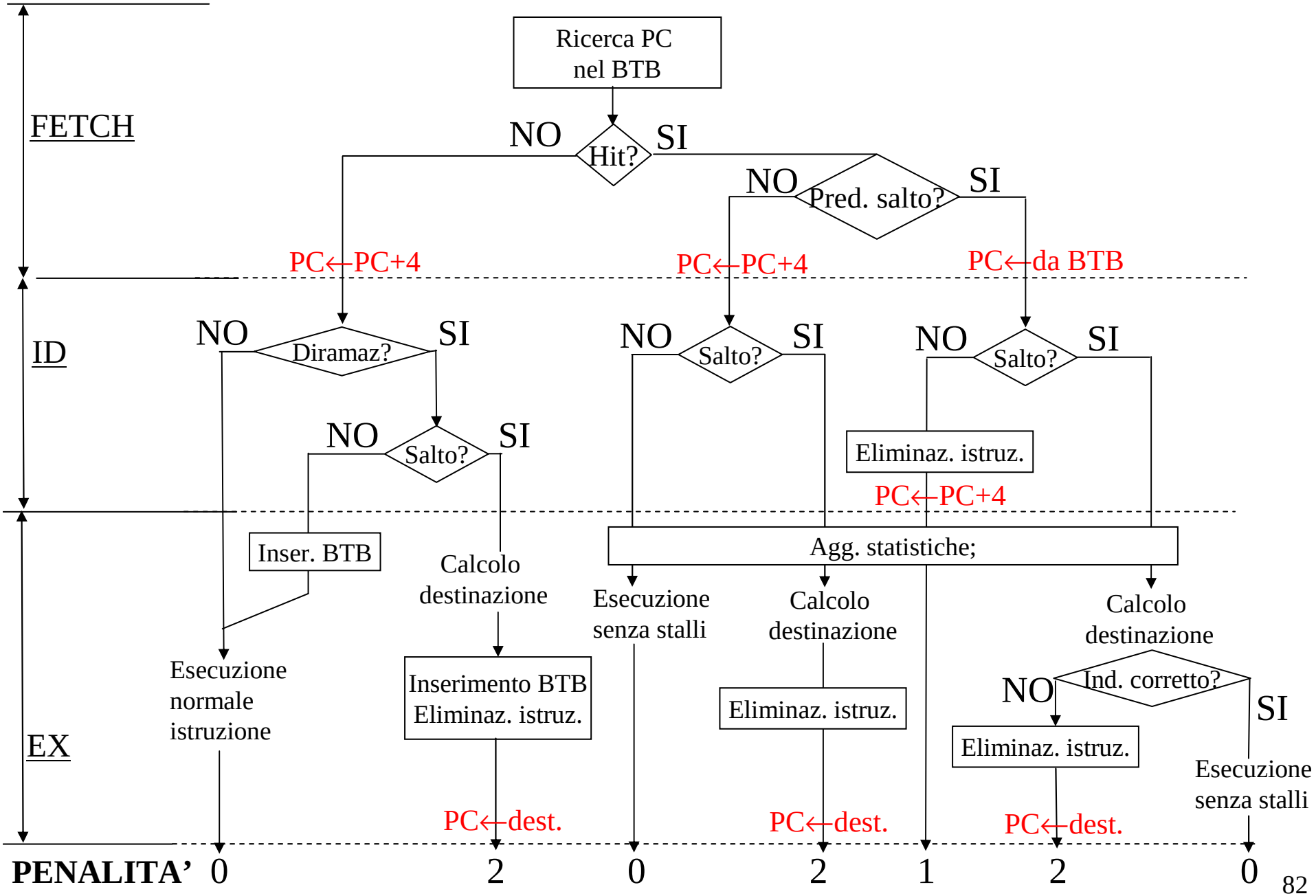


Idea di base: nella fase di fetch confronto PC con "PC" diramazione:

- Se non è presente: non dovrebbe essere diramazione e procedo
- Se è presente:
  predice non salto: fetch istruzione successiva
  predice salto: uso PC destinazione per il fetch dell'istruzione

NB: Può accadere che l'istruzione non sia in BTB ma comunque sia diramazione
⇒ quando ci si accorge, si può procedere come se salto non sia effettuato
(oppure utilizzare la tecnica di predizione statica)
Può accadere che l'indirizzo destinazione sia sbagliato
(ad es. perché si usano solo bit meno significativi di PC)

Più in dettaglio: supponiamo valutazione condizione diramazione in ID, indirizzo in EX



Note sulla precedente figura

- Le tre fasi F, ID, EX, sono riferite all'istruzione di (potenziale) diramazione.
- Si assume che la verifica della condizione di salto (se diramazione) avvenga nella fase ID, mentre il calcolo dell'indirizzo di destinazione (se diramazione) avvenga nella fase EX.
- Si assume che nei buffer interstadio vengano propagati i valori di:

  PC per l'accesso a BTB al fine di aggiornare statistiche

  PC+4 per tornare a istruzione successiva in caso di salto previsto ma non effettuato.
- Si assume invece che, anche se hit BTB, non venga propagato l'indirizzo destinaz. previsto, cosicchè nel caso di salto non previsto ma da effettuare (cosa conosciuta in ID) sia comunque necessario calcolare la destinazione in EX.
- Adottata la semplificazione che hit in BTB implichi che la istruzione sia una diramazione… altrimenti occorre complicare leggermente lo schema…

  [lasciato per esercizio ai più appassionati…]

NB: è chiaro che il problema dell'hit in BTB che poi non si rivela un branch può essere evitato se il tag del BTB include tutti i bit del program counter.

➡ Tutte le scelte sono sempre effettuate come compromesso ingegneristico: dimensioni BTB vs. accuratezza predizione

<u>Possibili strategie di ottimizzazione del BTB</u>

- Memorizzare in BTB solo diramazioni per le quali si prevede un salto effettivo
  [conviene risparmiare inserimento: miss in BTB comporta comunque predizione di "non salto"!]
- Politica per l'eliminazione di diramazioni da BTB:
  - meglio eliminare diramazioni che hanno bassa probabilità di essere incontrate [ok LRU in questo senso]
  - meglio eliminare le diramazioni che hanno bassa probabilità di portare a salti [per lo stesso motivo precedente]

$\Rightarrow$Algoritmo MPP (Minimum Performance Potential):
eleminata diramazione con minor
P(riferimento) * P(salto effettivo)
F(bit di LRU) F(bit di predizione)

TENDENZE ATTUALI NEI METODI PER RIDURRE COSTI DI BRANCH

- Anticipare la valutazione del salto ai primi stadi può ridurre il costo, ma non risolve completamente il problema.
- Con pipeline complesse, il salto ritardato è sempre più difficile da gestire per il compilatore [necessità di NOP che comportano un costo]

  $\Rightarrow$ tende ad essere abbandonato
- Usando tecniche di predizione, le penalità dovute ad errate predizioni sono tanto maggiori quanto maggiore è il numero di stadi ed il "grado di parallelismo" della pipeline

   Si introducono sistemi di predizione sempre più sofisticati

- Predittori a correlazione

  Combinano informazioni su comportamento globale (tutte le diramazioni) e comportamento locale (diramazione specifica considerata)

  Es. più predittori a due bit per ogni diramazione, tra i quali si sceglie sulla base del comportamento globale

- Predittori "tournament"

  Più predittori per ogni diramazione, si sceglie ogni volta quello che nella storia recente ha dato risultati migliori.

PIPELINE:

GESTIONE DELLE ECCEZIONI

Richiami

Tipologie di eccezioni

- Eccezioni interne
  - chiamata al sistema operativo da parte di un programma utente
  - uso di istruzioni indefinite, divisione per 0, overflow, ecc.
- Eccezioni esterne
  - Richiesta di un dispositivo di I/O
- Malfunzionamento dell'hardware [può essere interna o esterna]

Gestione delle eccezioni

Per ora, supponiamo basti soltanto fornire:

- Trasferimento del controllo ad una routine di servizio
- Indirizzo dell'istruzione "responsabile" dell'eccezione in EPC
  [Exception Program Counter] – in modo da poter poi riprendere il controllo
- Causa dell'eccezione:
  - modifica di un opportuno registro (routine unica per tutte le cause)
  Alternativa: interruzioni vettorizzate (una routine diversa per ogni causa)

In ogni caso, è importante notare che:

Il verificarsi di un'eccezione comporta l'intervento del sistema operativo, che deve riconoscere la causa dell'eccezione e prendere i relativi provvedimenti:

- Nel caso di istruzione indefinita, malfunzionamento HW, overflow aritmetico, normalmente il programma viene interrotto e viene segnalata l'eccezione
- Nel caso di richieste di I/O o chiamate al sistema operativo, viene salvato lo stato del programma corrente per "poi" riprenderne l'esecuzione (dopo aver svolto il compito richiesto)
- Si vedrà poi come le eccezioni ed il sistema operativo abbiano entrambi un ruolo nella gestione della memoria virtuale

Interrelazione HW – SW [Sistema Operativo]

Gestione delle eccezioni nel caso di controllo multiciclo:

- È sufficiente aggiungere al diagramma a stati finiti uno o più stati addizionali responsabili delle operazioni previste
- L'istruzione "responsabile" è quella in esecuzione al momento dell'occorrenza dell'eccezione, la causa dipende dall'evento occorso

- Per ogni causa uno stato specifico (cui si accede sulla base degli ingressi unità di controllo: opcode + segnali dal datapath + segnali esterni come ad esempio IRQ)
- A partire da questo stato:
  - scrittura del registro EPC con PC corrente [– "1 istruzione"]
  - eventuale scrittura del registro causa
  - aggiornamento PC con l'indirizzo della routine di servizio [unica o dipendente dalla causa]

Gestione delle eccezioni nel caso di controllo con pipeline:

PROBLEMI:

- Nel momento in cui avviene un'eccezione, ci saranno più istruzioni nella pipeline...
- Quale istruzione è responsabile dell'eccezione?
- Cosa fare delle istruzioni precedenti ancora presenti in pipeline?
- Cosa fare delle istruzioni successive già presenti nella pipeline?

La gestione può essere diversa a seconda delle tipologie di causa:

1. Eccezioni interne [chiamata S.O. – Overflow o istruzione indefinita…]
2. Interruzioni esterne da parte dei sistemi I/O
3. Malfunzionamento HW

Vediamo la gestione delle eccezioni interne:

• In questo caso l'istruzione responsabile dell'eccezione è chiaramente definita
  - l'eccezione avviene in uno stadio specifico S
    [p.es. overflow: stadio EX – istruzione indefinita: stadio D]
• E' necessario passare il controllo ad una routine di servizio, ma anche prevedere la gestione di più istruzioni nella pipeline:
  - l'istruzione responsabile dell'eccezione deve essere immediatamente interrotta ed eliminata [non deve modificare i registri]
  - le istruzioni che seguono l'istruzione responsabile [negli stadi precedenti S] devono essere eliminate: completare la loro esecuzione sarebbe dannoso dopo il verificarsi di un'eccezione
  - le istruzioni che precedono l'istruzione responsabile [negli stadi seguenti S] devono essere completate: la routine di servizio deve conoscere lo stato che "precede" l'esecuzione dell'istruzione responsabile dell'eccezione

[vedi esempio nel lucido seguente]

addi $1, $1, 100

→ Valore di $s1 usato da add [via propagazione!]

add $1, $2, $1

Overflow aritmetico: add interrotta nello stadio EX

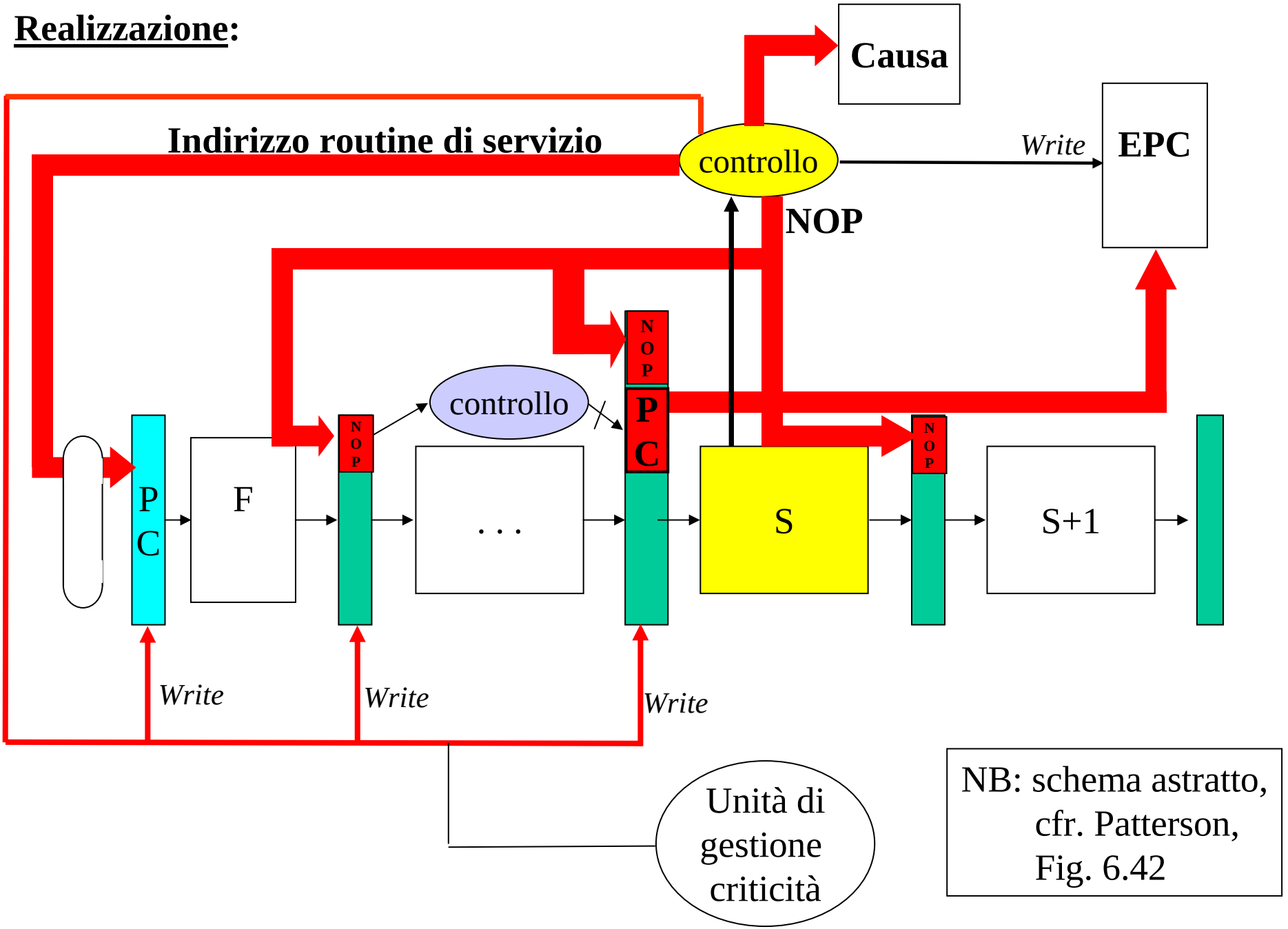
↓

Dipende anche dal valore di $1: la routine di servizio potrebbe dover conoscere il valore. Ma questo, quando add è nello stato EX, è stato propagato dalla addi ma non è ancora stato scritto nel register file dalla addi [che si trova in M].

⇒ E' necessario allora lasciare che le istruzioni precedenti vengano eseguite;
ciò non significa ritardare la chiamata alla routine di servizio:
le istruzioni della routine di servizio sono caricate in pipeline normalmente, subito "dopo" quelle eliminate!

QUINDI
- eliminare (segnali a reg. interstadio) istruzione responsabile e seguenti
- scrivere in opportuni registri la causa e l'indirizzo dell'istruzione interrotta
- modificare PC: caricamento prima istruzione routine di servizio

Realizzazione:



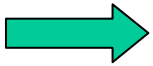
NB: schema astratto, cfr. Patterson, Fig. 6.42

NOTE AL LUCIDO PRECEDENTE:

- il concetto fondamentale è che le istruzioni fino a S sono scartate, quelle in stadi successivi continuano e i registri EPC, Causa, PC sono aggiornati opportunamente
- ovviamente i segnali Write ai registri interstadio sono in OR con tutti quelli che provengono da unità che necessitino di scrivere (o impedire la scrittura) nei registri interstadio (ovvero, l'unità di gestione delle criticità)
- l'introduzione di un valore "NOP" in un registro interstadio si intende ottenuta comandando un opportuno MUX [vedi p.es. i dettagli nella specifica realizzazione proposta dal Patterson]
  Analogamente, la scrittura dell'indirizzo della routine di servizio nel PC avviene comandando opportunamente il MUX uno dei cui ingressi è collegato a valore costante corrispondente all'indirizzo stesso
- nell'esempio si usa un registro di causa, ma è facile immaginare la modifica per interruzioni vettorizzate (p.es. basta usare un MUX opportuno verso PC, i diversi ingressi corrispondono ai diversi indirizzi delle routine di servizio)

PROBLEMA:

In uno stesso ciclo di clock possono accadere più eccezioni,
con cause diverse (da diversi stadi)



- Si assegna una priorità alle eccezioni: servita quella a priorità più alta (di solito si serve l'istruzione più vecchia, ovvero lo stadio più avanzato)
- Il registro Causa può mantenere tutte le eccezioni che si verificano in un ciclo di clock: quando è stata servita la prima, sono disponibili le informazioni sulle altre…

Per le interruzioni esterne:

- È possibile "attribuire" l'interruzione all'istruzione più conveniente, tipicamente completando tutte le istruzioni già entrate nella pipeline (altrimenti dovrei scartarle e avrei dei cicli di penalità inutili!)
- Per i malfunzionamenti hardware: trattati come le interruzioni esterne se di natura "esterna", altrimenti come le interruzioni interne!

Eccezioni precise e eccezioni imprecise

- La gestione delle eccezioni quando si hanno più istruzioni nella pipeline è in ogni caso complicata
- Con pipeline viste fino ad ora, è possibile attribuire a EPC l'indirizzo di una istruzione [causa dell'interruzione] e fare in modo che si abbia uno stato coerente [istruzioni seguenti scartate, precedenti completate]

  $\Rightarrow$ **eccezioni precise**
- Tuttavia, alcune soluzioni progettuali rilassano i requisiti ed ammettono che lo stato risultante dall'eccezione possa essere incoerente:

  $\Rightarrow$ si parla in tal caso di **eccezioni imprecise**

Esempi:

- un'istruzione successiva a quella che ha provocato l'eccezione arriva alla fase di completamento modificando i registri in modo incoerente; oppure
- EPC viene comunque scritto con il valore di PC, che corrisponde all'istruzione in fase di fetch a prescindere da quella che ha provocato l'eccezione.

Ha importanti riflessi sul sistema operativo!