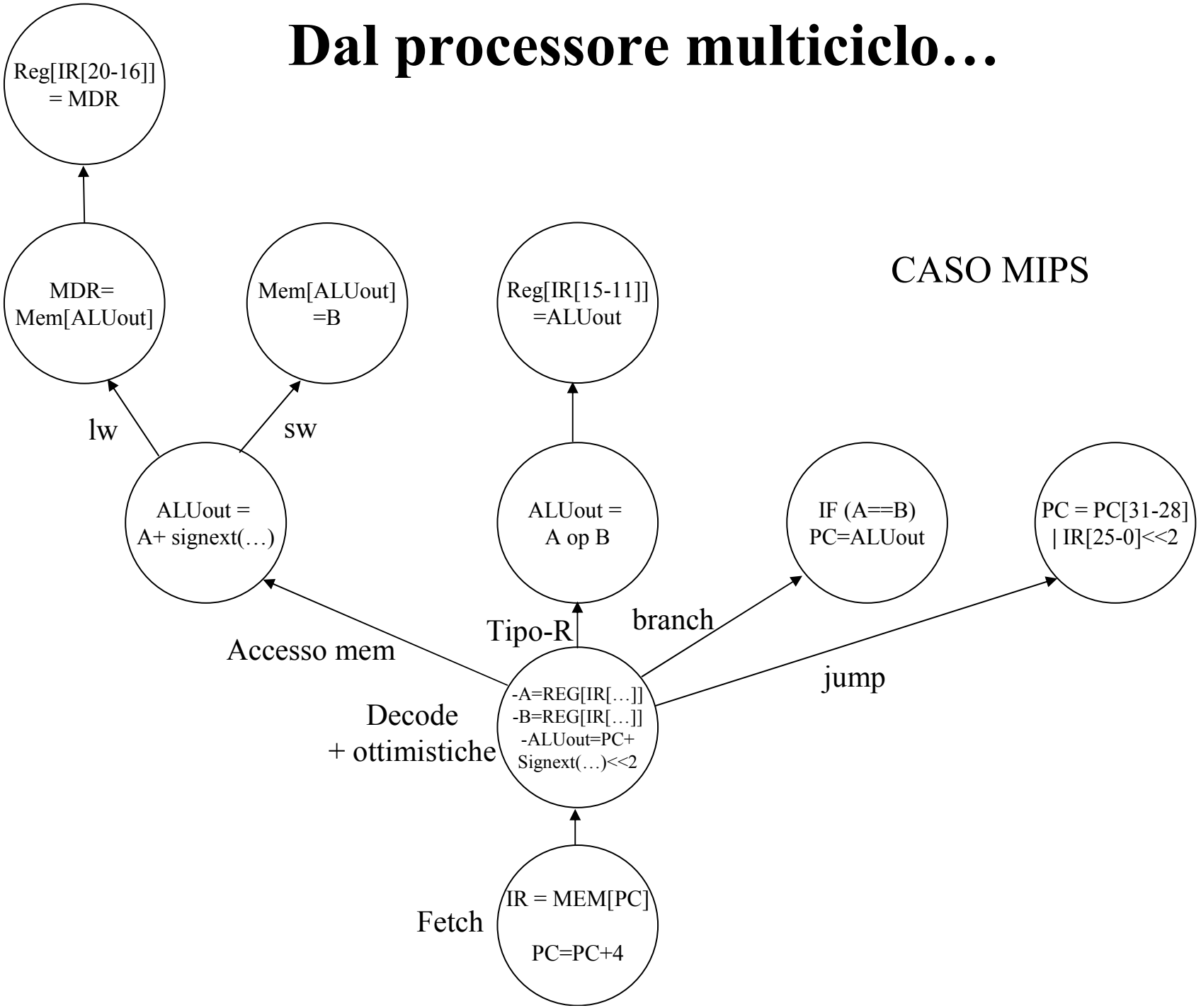
Calcolatori Elettronici B
a.a. 2004/2005

Tecniche Pipeline

Massimiliano Giacomin

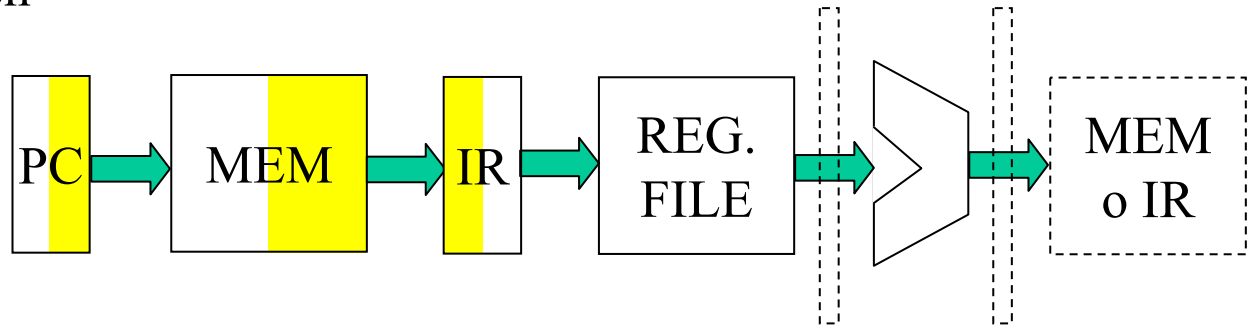
Dal processore multiciclo…



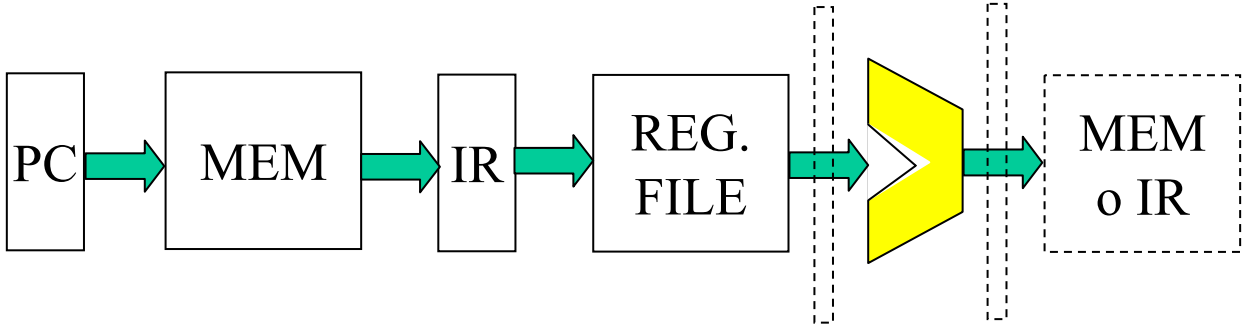
alla pipeline…

• Ogni stato richiede in genere solo una parte distinta dell'unità di elaborazione (il tratteggio indica eventuali registri temporanei)

P.es. Fetch



P.es. esecuzione operazione in ALU in istruzione di TIPO-R



• IDEA: come in una catena di montaggio, cominciare l'esecuzione dell'istruzione successiva non appena si libera lo stadio di fetch, procedendo per stadi successivi

Pipeline: concetti generali

Pipeline = "oleodotto" | "catena di montaggio"

- Tecnica di implementazione in cui più istruzioni sono sovrapposte durante l'esecuzione.
  Esempio "pratico": la costruzione di automobili in catena di montaggio.

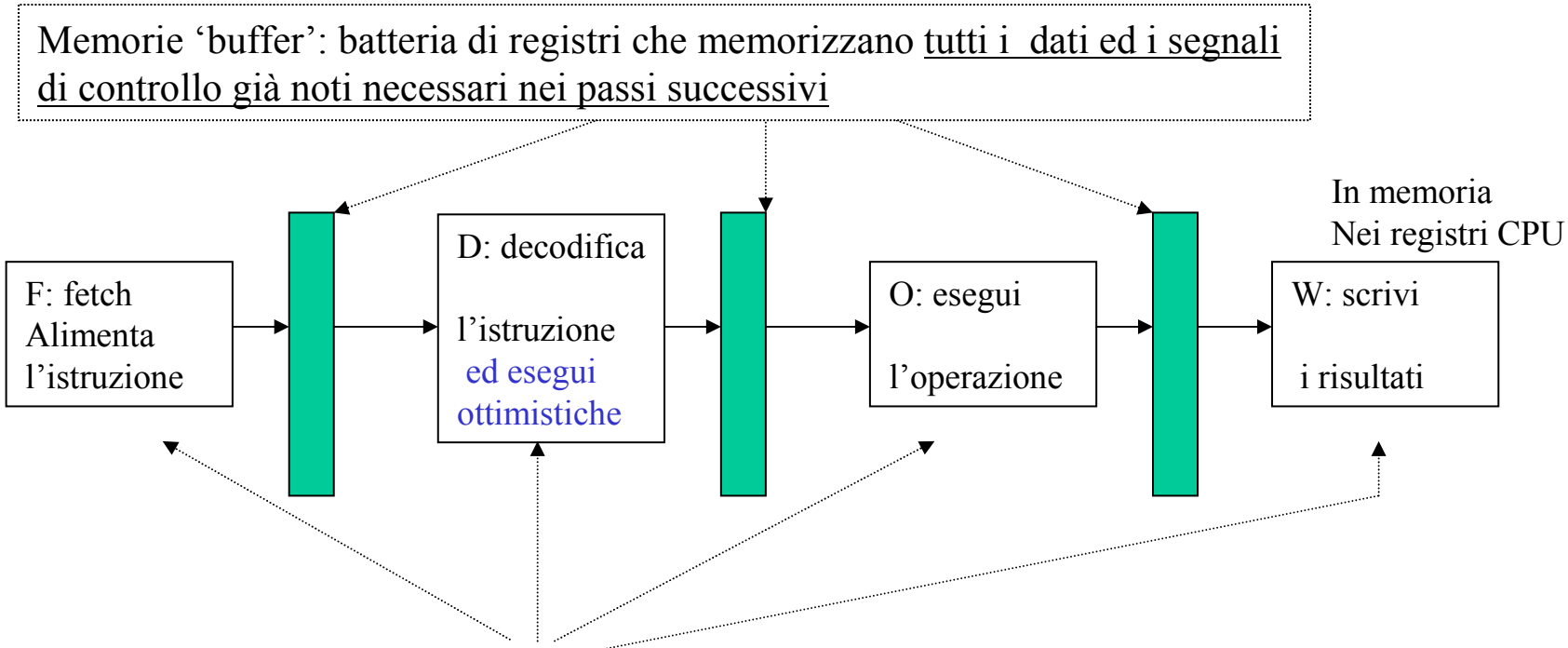
- La CPU è organizzata in un certo numero di "stadi" sequenziali
  - ogni istruzione richiede tutti gli stadi per la sua esecuzione
  - non appena uno stadio si libera, viene impegnato per l'istruzione seguente

  $\Rightarrow$ sfruttamento del parallelismo (ogni stadio è impegnato) nell'esecuzione di un flusso sequenziale di istruzioni.

- E' un esempio di Instruction Level Parallelism (IPL) poiché le istruzioni sono valutate in parallelo.

- Il tempo di esecuzione delle singole istruzioni non diminuisce, ma aumenta il "throughput" ["frequenza di operazione"]

Pipeline: l'organizzazione



L'esecuzione avviene in stadi (unità di esecuzione), che completano generalmente la propria attività in un ciclo di clock.

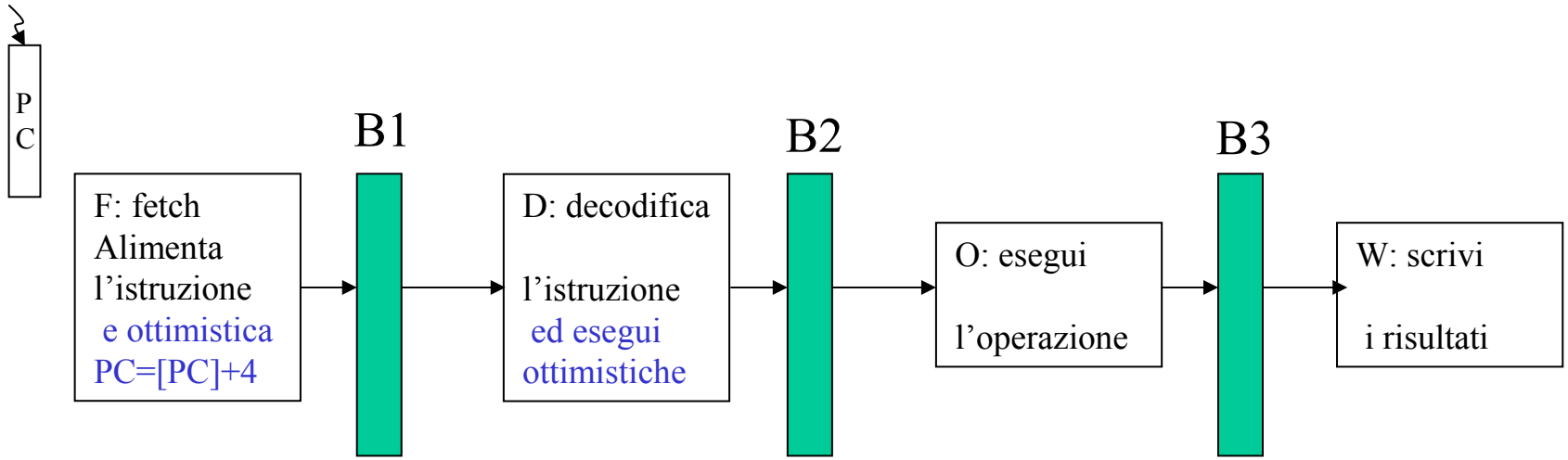
Ogni stadio è attivo in ogni ciclo di clock.

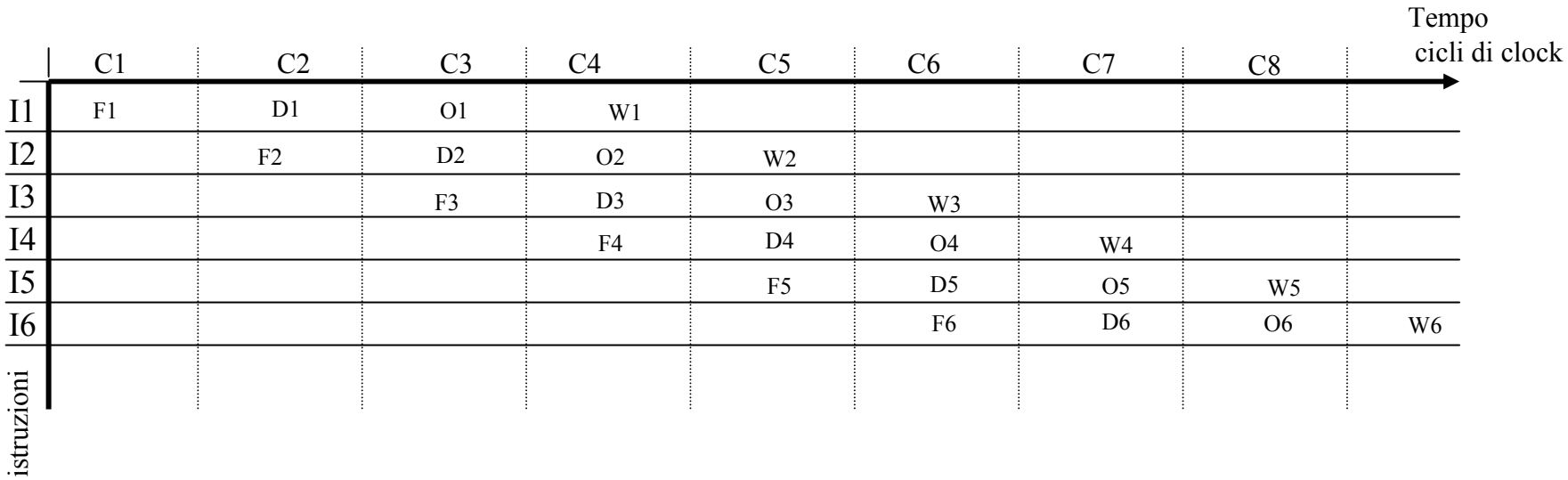
NB: notare che a differenza del MIPS stiamo considerando una suddivisione in 4 stadi.

NB: come operazioni ottimistiche consideriamo il prelievo degli operandi

[Potrebbero esserci anche altre operazioni, p.es. come nel caso multiciclo il calcolo indirizzo di salto ipotizzando una certa modalità di indirizzamento]

Esempio
di schema temporale dell'esecuzione





C1:

durante il ciclo: viene letta l'istruzione I_1 dalla memoria

alla fine del ciclo: il buffer B1 contiene I_1

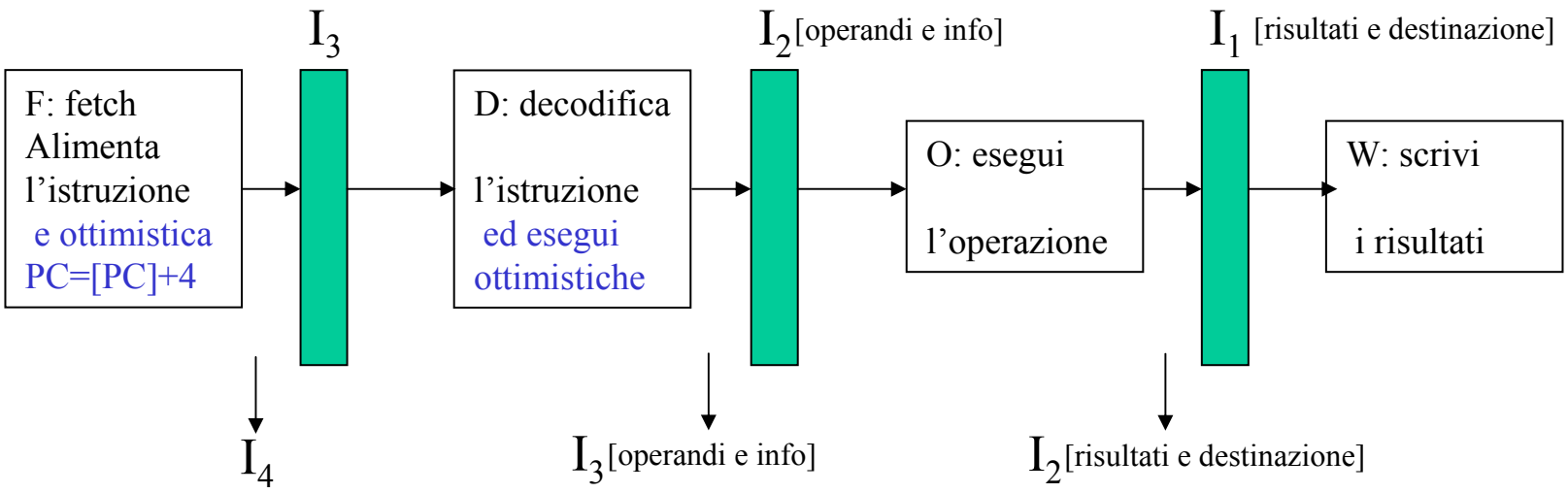
C2:

durante il ciclo: lettura I_2 dalla memoria + decodifica di I_1 (presente in B1)
lettura degli operandi di I_1

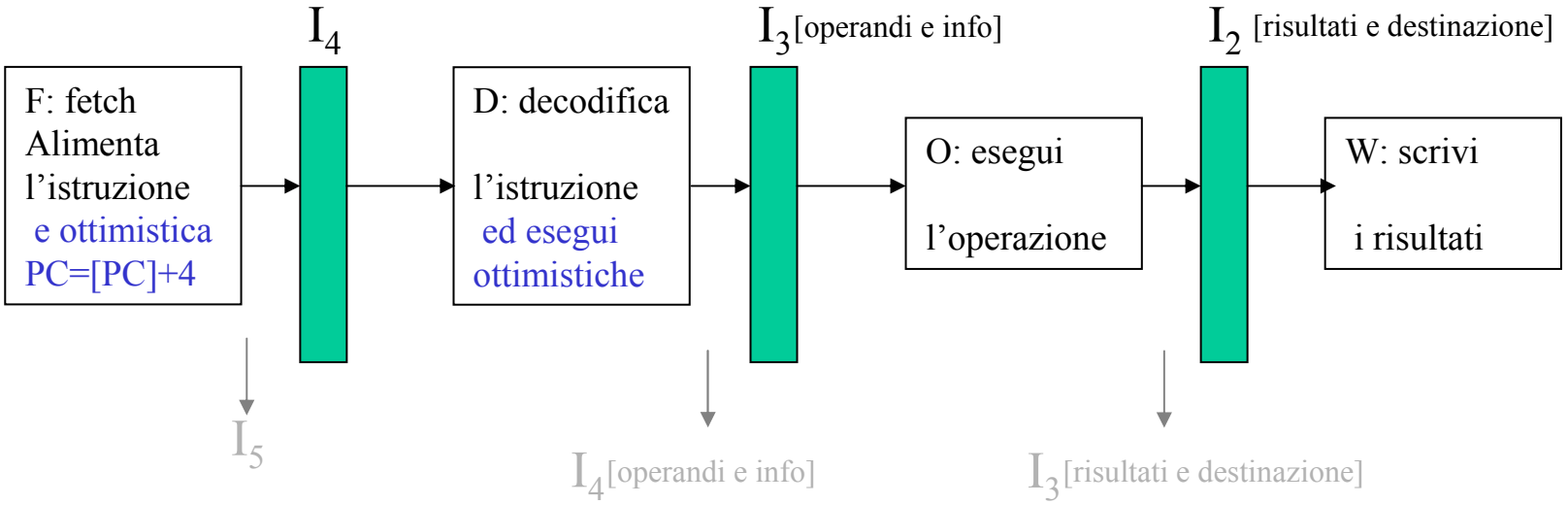
alla fine del ciclo: B1 contiene I_2, B2 contiene operandi sorgente I_1
e informazioni necessarie all'esecuzione
di I_1 negli stadi successivi

. . .

durante C4:



alla fine di C4:



Pipeline: prestazioni [ideali] e numero di stadi

- Abbiamo visto che “buone prestazioni” sono favorite (tra le altre cose) da:
  - velocità dei circuiti [alto grado di integrazione e densità circuitale]
  - diminuzione lunghezza del “cammino critico” per abbassare T_{clock}

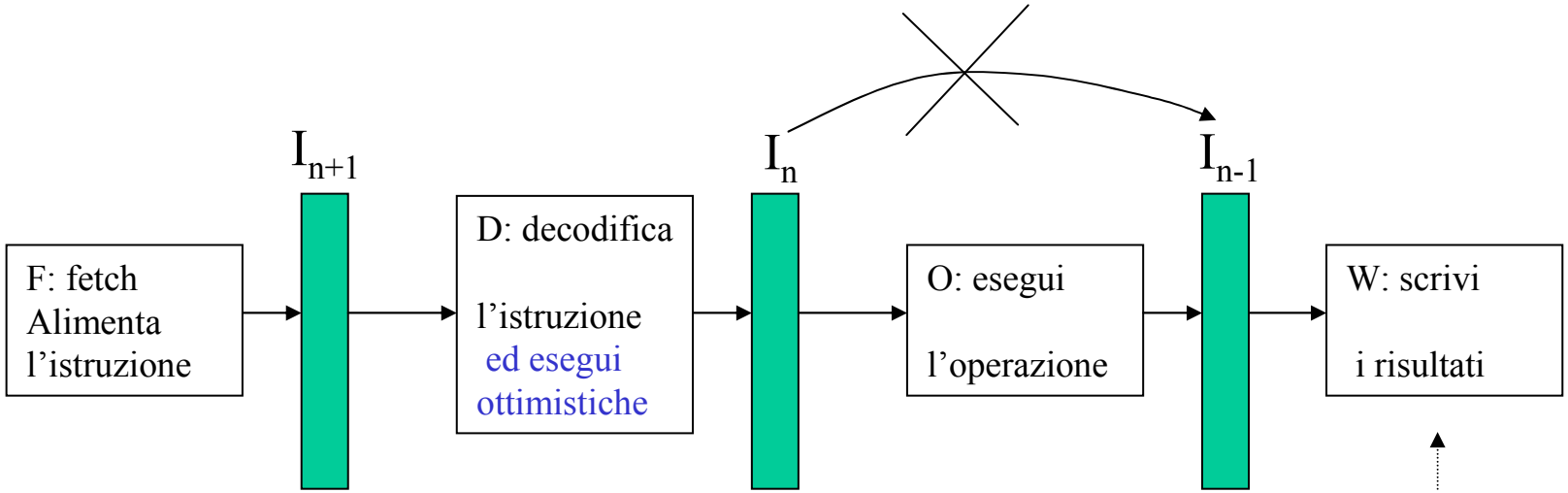
- Rispetto al processore a singolo ciclo, il multi-ciclo abbassa il tempo di esecuzione delle istruzioni più frequenti, che “non devono adeguarsi” alle istruzioni che hanno il “cammino critico” più lungo.
  Tuttavia, ogni istruzione deve aspettare la fine della precedente…

- La tecnica con pipeline aumenta il numero di istruzioni completate in un ciclo di clock (throughput), ovvero aumenta il CPI (clock per istruzione): idealmente, un’istruzione non deve mai aspettare la fine della precedente!

- Viceversa, il tempo di esecuzione della singola istruzione è in generale aumentato!
  In generale, servono più cicli di clock per una singola istruzione, pari al numero degli stadi, anche per le istruzioni che ne usino un sottoinsieme! [In questo senso, la tecnica con pipeline ha qualche somiglianza con singolo ciclo]
  Ciò però non ha alcuna importanza, visto che comunque parte una istruzione per ciclo di clock!

Infatti: supponiamo che una istruzione I_n non usi lo stadio di esecuzione

Alla fine di un certo ciclo di clock si ha:

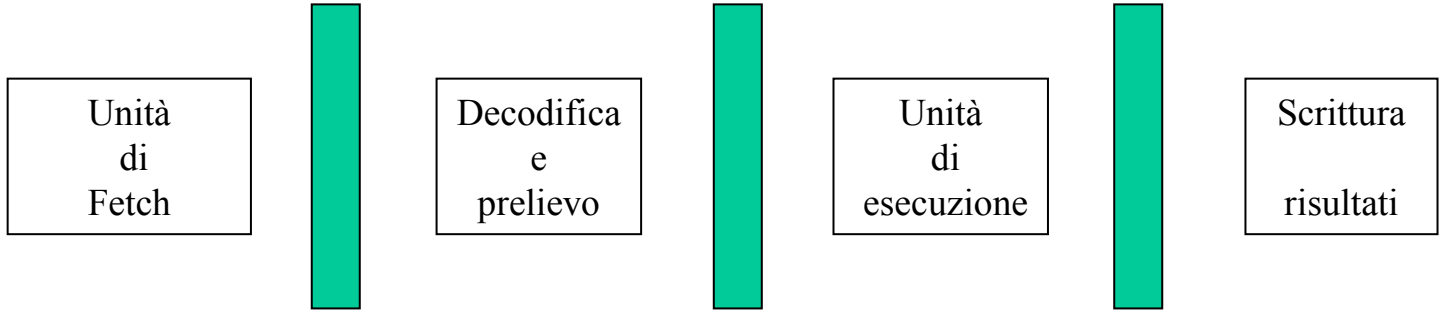


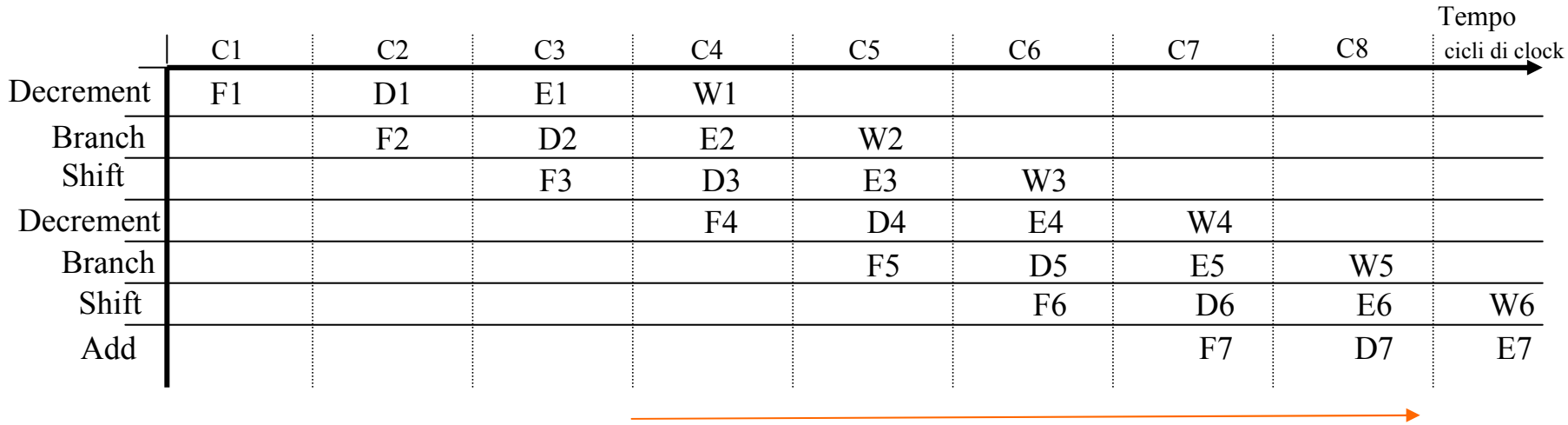
Non è possibile per I_n utilizzare nel ciclo successivo lo stadio W, perché già impegnato per l'esecuzione di I_{n-1}

D'altra parte, ciò non comporterebbe un vantaggio sostanziale: la decodifica di I_{n+1} (già caricata) comincia comunque, nessuna istruzione deve più aspettare la fine della precedente!

⇒ Ciò che conta è il throughput…

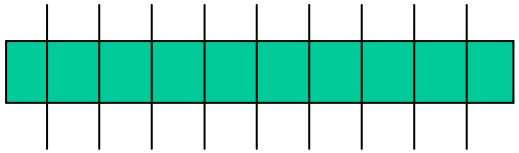
Throughput: Numero di istruzioni eseguite per ciclo di clock



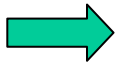


- Tra C4 e C8 conclude 5 istruzioni in 5 cicli: throughput di 1 istruzione per ciclo
- In realtà ciò vale a regime, ad esempio tra C1 e C5 ne conclude 2!

- Ovviamente, il numero di istruzioni eseguite al secondo aumenta diminuendo T_{clock}
- Idealmente, aumento throughput proporzionale al numero di stadi:



T_{clock} nel caso di singolo ciclo

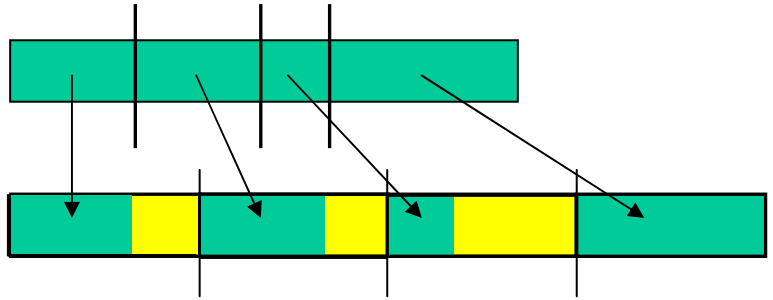


Mi basta aumentare il numero di stadi in cui suddivido l'esecuzione dell'istruzione per aumentare le prestazioni…

- Putroppo le cose non sono così semplici:
  - C'è un limite "fisico" al numero di stadi (le unità elementari di calcolo!)
  - Gli stadi devono essere **bilanciati**!!!

    T_{clock} deve essere maggiore dell'operazione più lunga tra i vari stadi! (stessa cosa accadeva nel caso del multiciclo)

Quindi in realtà:



  - aumento numero di stadi aumenta problemi nella gestione delle criticità (lo vedremo in seguito)

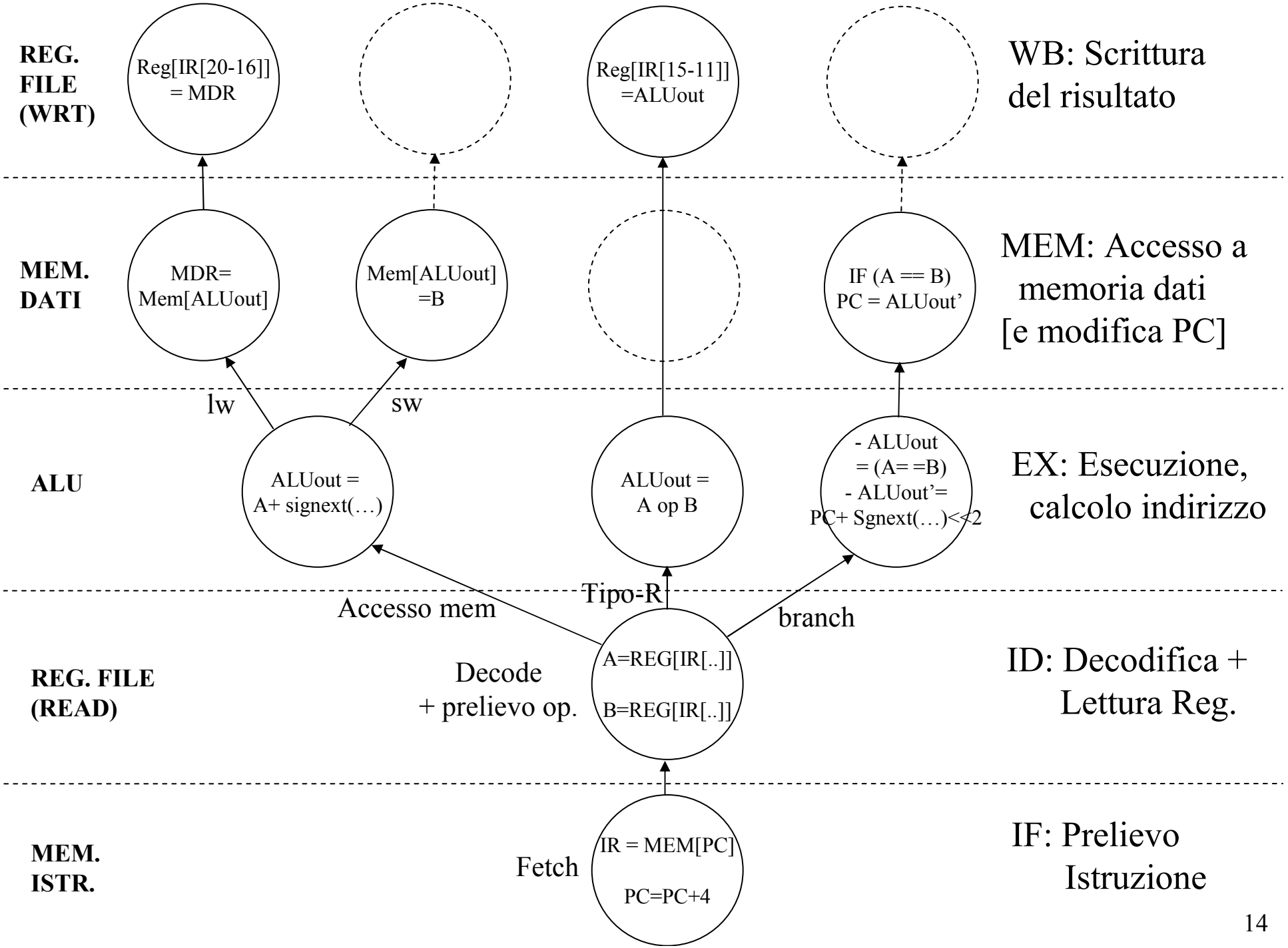
PROBLEMATICA:

in quanti e quali stadi suddividere l'esecuzione?

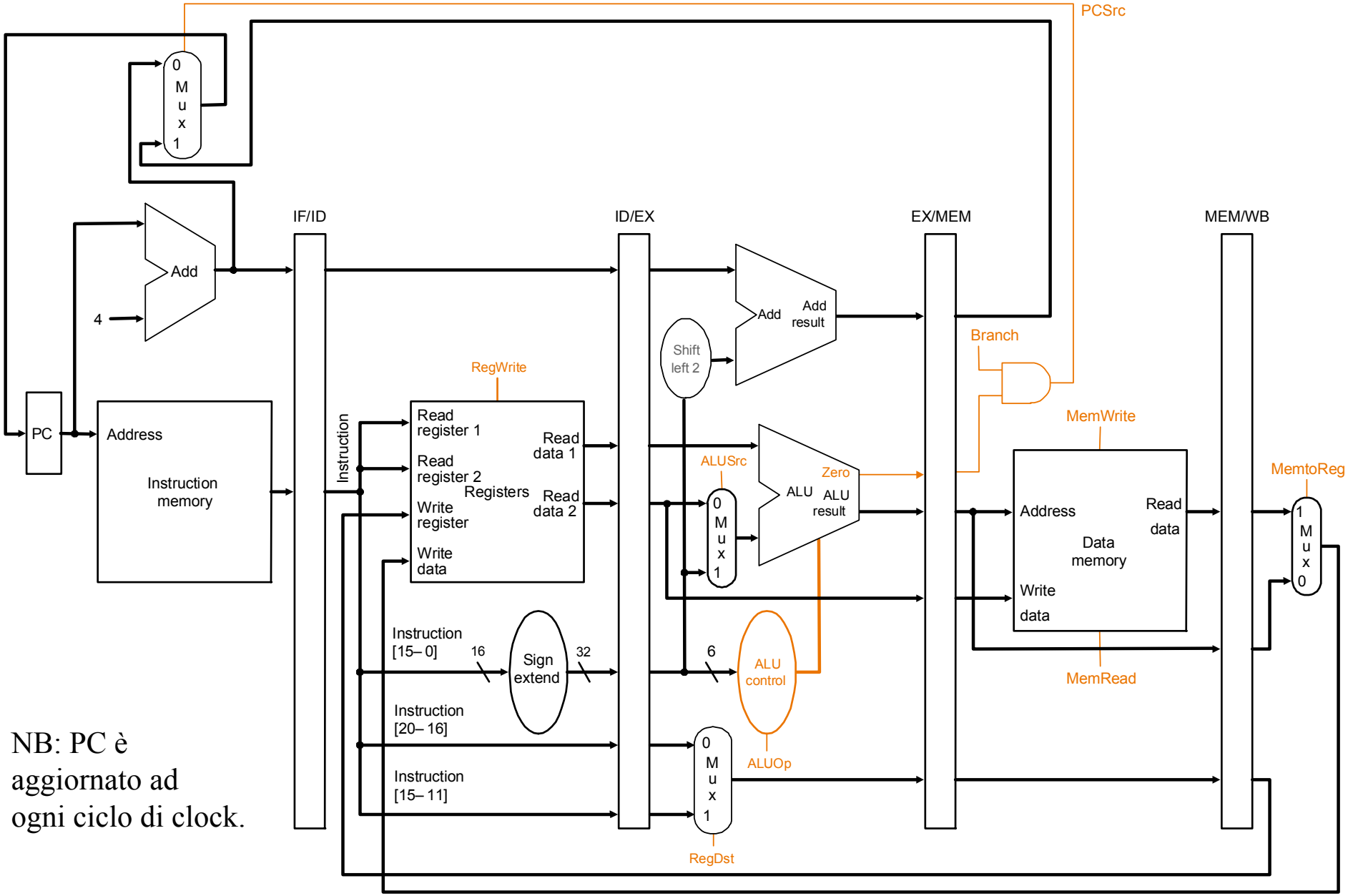
- Nei casi reali, il numero va da 2-3 a qualche decina…

- Vediamo il caso MIPS didattico presentato nel
  Patterson & Hennessy
  (in cui si considerano le istruzioni Tipo-R, lw, sw, beq)
  $\Rightarrow$ Suddivisione in stadi, ciascuno con distinte risorse HW.
  Per rendere bilanciati gli stadi, ci riferiamo alle risorse
  con operazioni considerate "più pesanti":
  Memoria, Register File, ALU

  NB: si considera memoria istruzioni distinta da memoria dati
  (vedremo poi perché)

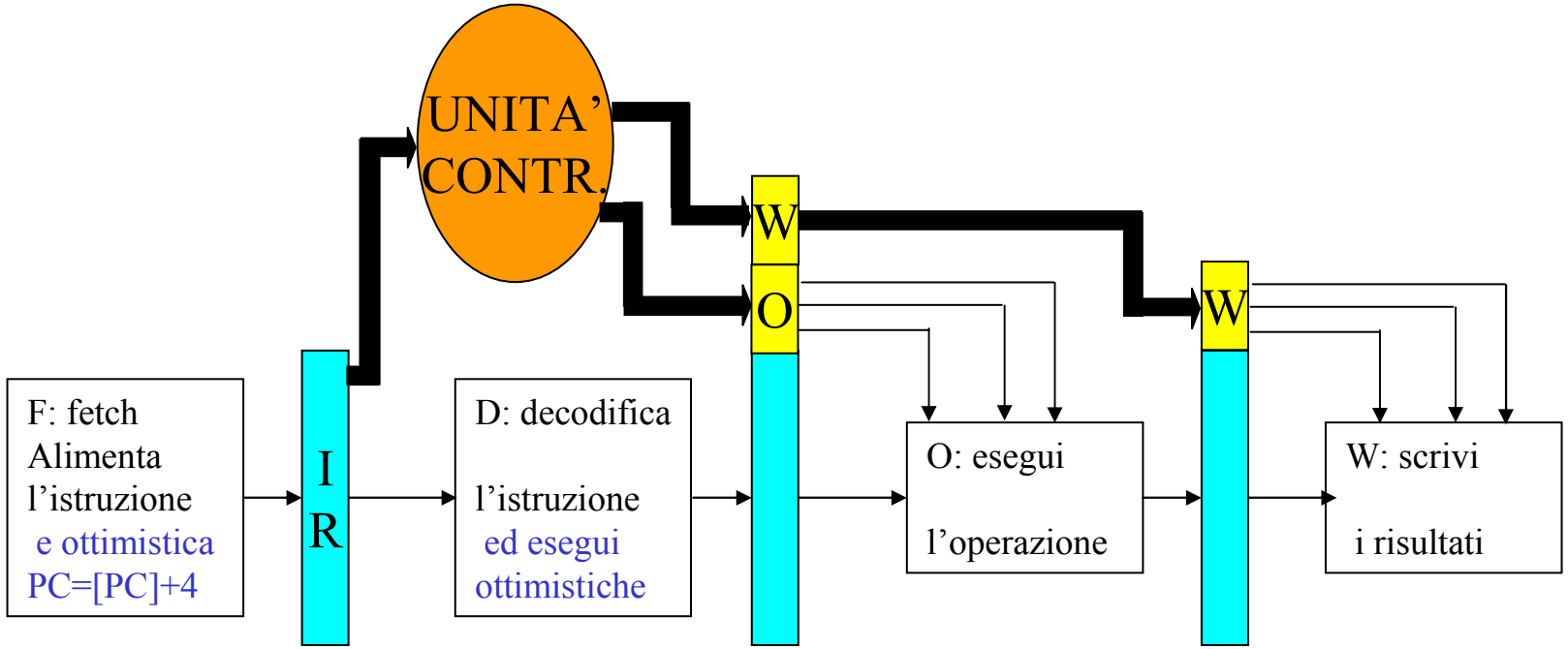

REG. FILE (WRT)
Reg[IR[20-16]] = MDR
Reg[IR[15-11]] =ALUout
WB: Scrittura del risultato
MEM. DATI
MDR= Mem[ALUout]
Mem[ALUout] =B
IF (A == B) PC = ALUout'
MEM: Accesso a memoria dati [e modifica PC]
lw
sw
ALU
ALUout = A+ signext(…)
ALUout = A op B
- ALUout = (A= =B) - ALUout'= PC+ Sgnext(…)<<2
EX: Esecuzione, calcolo indirizzo
Accesso mem
Tipo-R
branch
REG. FILE (READ)
Decode + prelievo op.
A=REG[IR[..]]
B=REG[IR[..]]
ID: Decodifica + Lettura Reg.
MEM. ISTR.
Fetch
IR = MEM[PC]
PC=PC+4
IF: Prelievo Istruzione

L’unità di elaborazione corrispondente è: [Fig. 6.25 Patterson & Hennessy]



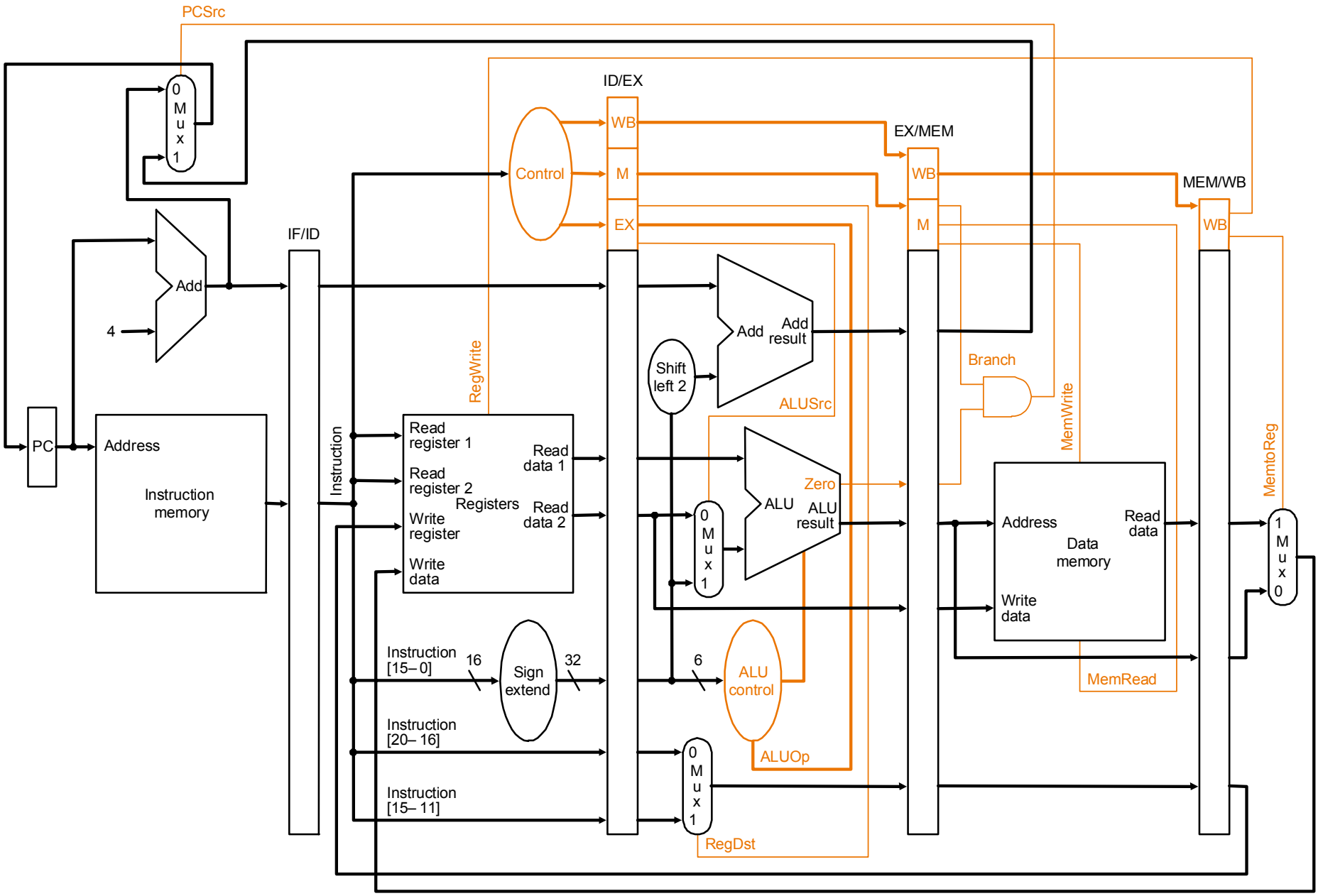
NB: PC è aggiornato ad ogni ciclo di clock.

Pipeline: il controllo



- L'unità di controllo è combinatoria: in fase di decodifica (D) calcola tutti i segnali di controllo che saranno necessari in tutti gli stadi.
- I segnali di controllo si spostano lungo la pipeline allo stesso modo delle altre informazioni (p.es. gli operandi)
- Questo è il modo più conveniente di realizzare un comportamento sequenziale: la decodifica è fatta nel primo stadio, che nel ciclo di clock successivo dovrà essere impegnato dall'istruzione successiva!

Nel caso del MIPS a 5 stadi [Fig. 6.30 Patterson & Hennessy]



Pipeline: i problemi

- Idealmente, il throughput è di una istruzione per ciclo di clock!
- Purtroppo, in realtà esistono problematiche:

  - Criticità **strutturali**: 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 usati</u>

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

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

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; replicarla aumenterebbe il costo e gli 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

istruzioni \ Tempo cicli di clock	C1	C2	C3	C4	C5	C6	C7	C8	
I1	F1	D1	O1	W1					
I2		F2	D2	O2	O2	O2	W2		
I3			F3	D3	D3	D3	O3	W3	
I4				F4	F4	F4	D4	O4	W4
I5							…	…	…
I6								…	…

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

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

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

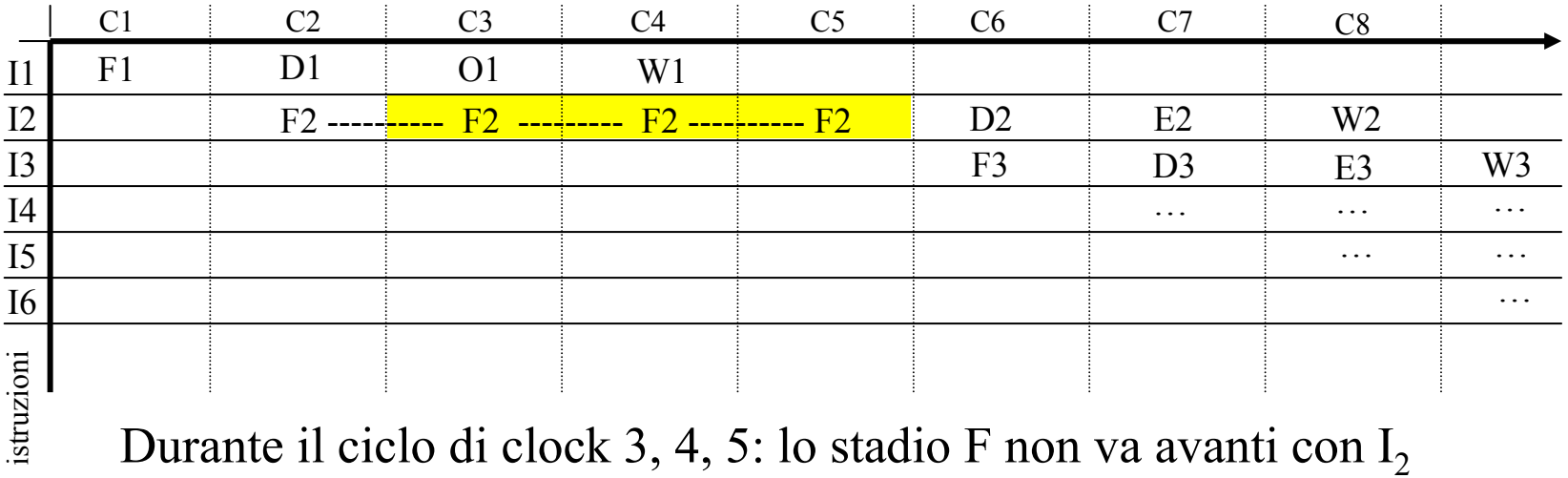
Nel caso precedente, gli stadi F – D – O non accettano nuove istruzioni per 2 cicli di clock: PC, B1 e B2 non sono sovrascritti, ma mantengono i valori precedenti.

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

• La pipeline può essere pensata come un “registro a scorrimento”:
- al termine di ogni ciclo, le informazioni contenute in tutti gli stadi si spostano in avanti di uno stadio
- se uno stadio richiede più cicli di clock, necessariamente tutta la pipeline entra in stallo: una nuova istruzione non può essere caricata finchè lo stadio non ha finito il suo compito (e il registro “ricomincia a scorrere”)

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

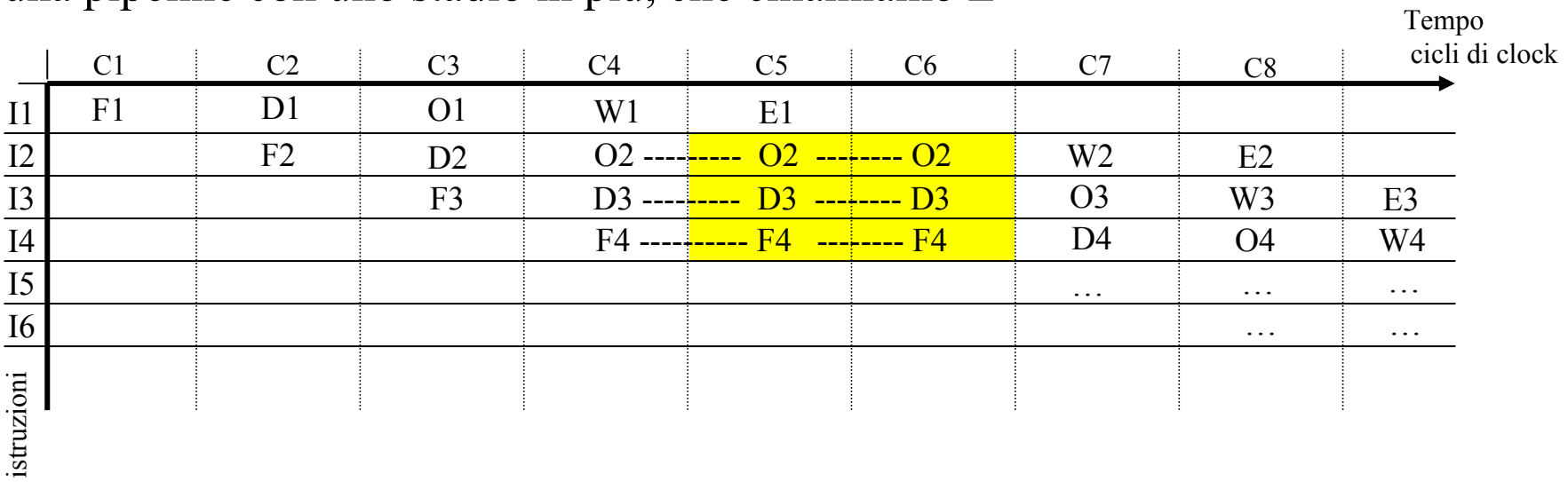
⟶ Lo stadio D non fa alcuna operazione (no dati su cui lavorare)

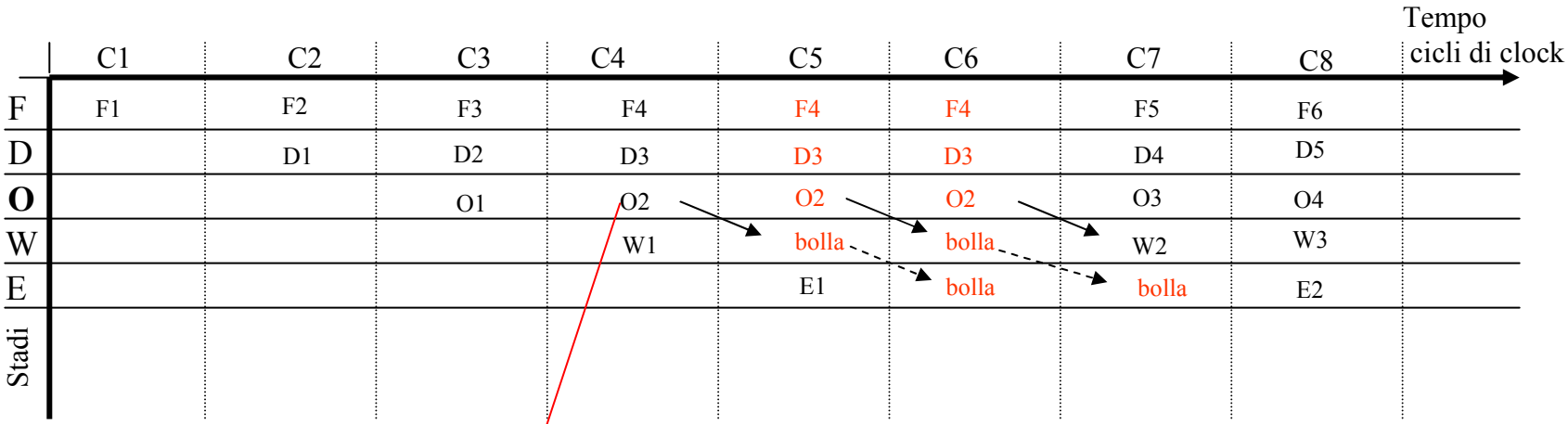
⟶ Lo stadio F non "accetta" nuove istruzioni

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"]
   $\Rightarrow$ il buffer S-1 / S non viene scritto dallo stadio S-1 negli n cicli successivi
2. Gli stadi precedenti devono mantenere negli n cicli di clock successivi 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]
   $\Rightarrow$ i buffer interstadio precedenti non vengono scritti e neppure PC negli n cicli di clock successivi
3. Nel ciclo di clock successivo 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)
   $\Rightarrow$ ad ogni ciclo degli n successivi, nel buffer S/S+1 vengono imposti valori 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

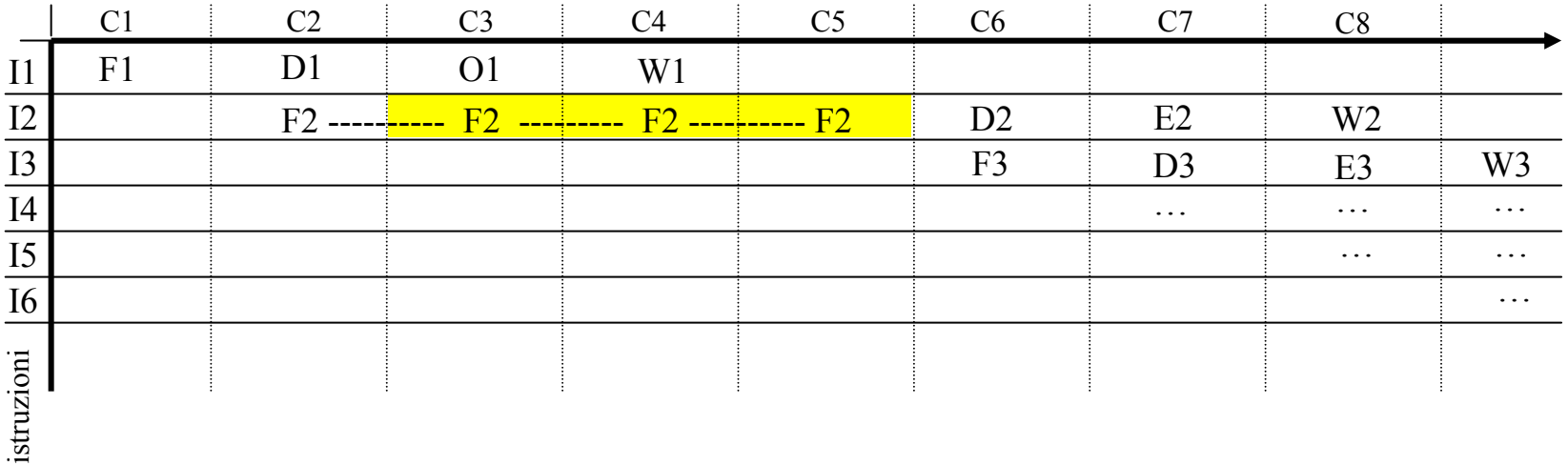


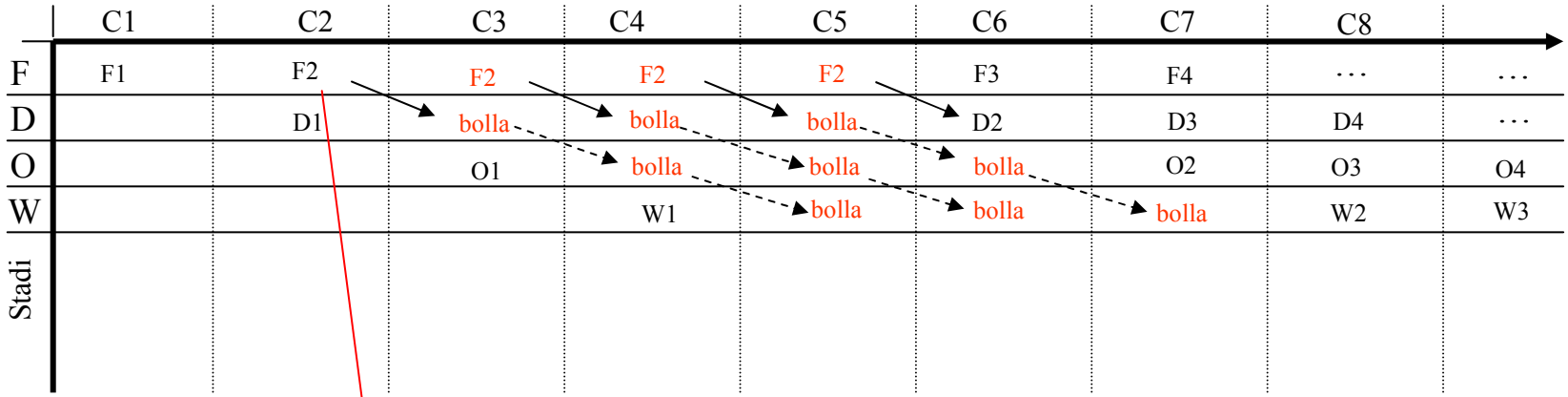


Riconoscimento situazione di stallo (2 cicli successivi)

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)

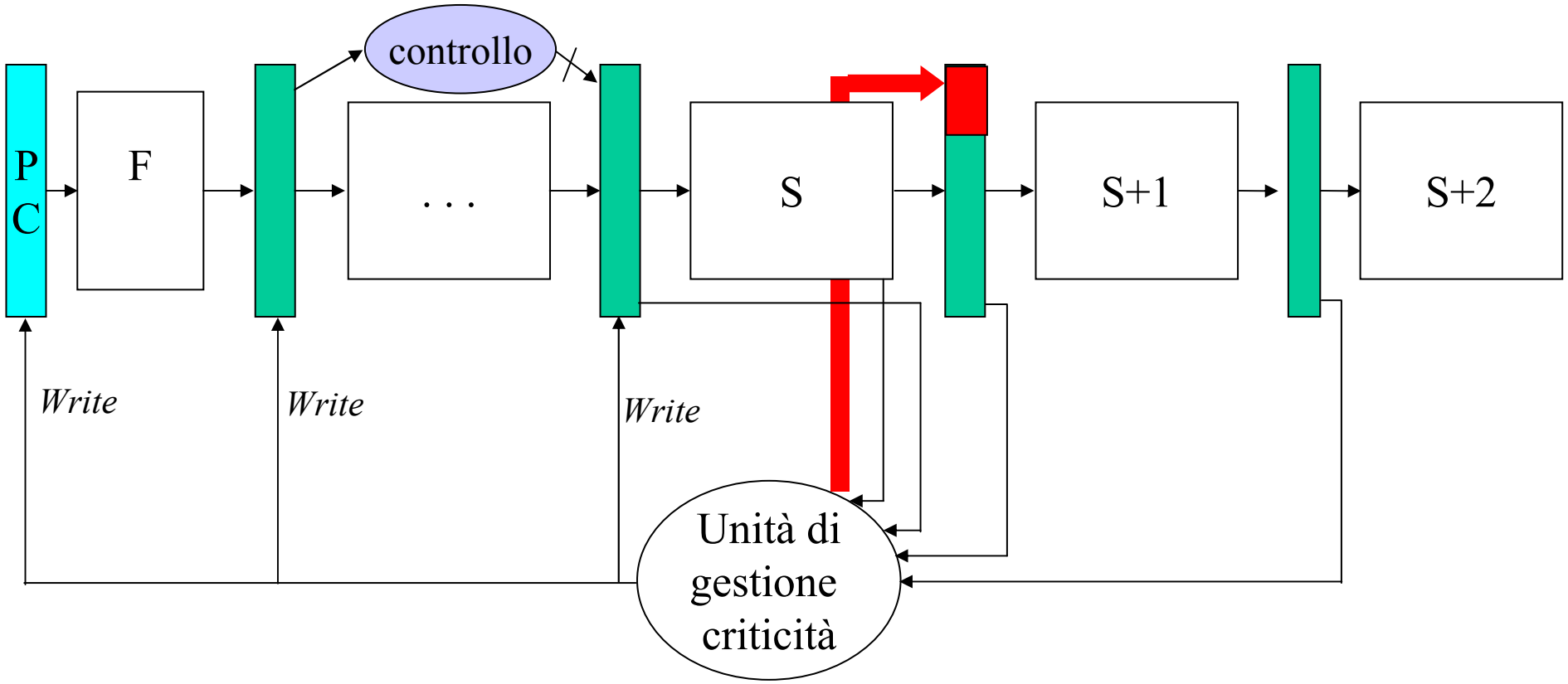




Riconoscimento situazione di stallo (3 cicli successivi)

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

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 forzati i segnali di controllo per forzare operazione
    "innocua": formazione di una bolla che si propagherà negli stadi successivi

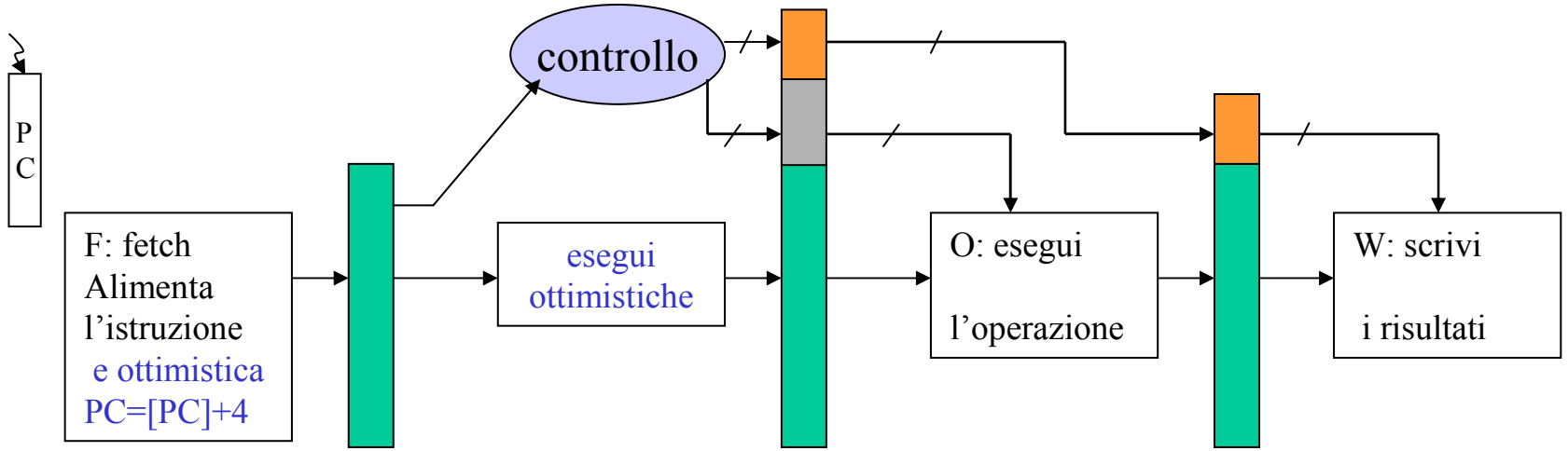
Esempio precedente (stallo per operazione complessa nello stadio O)

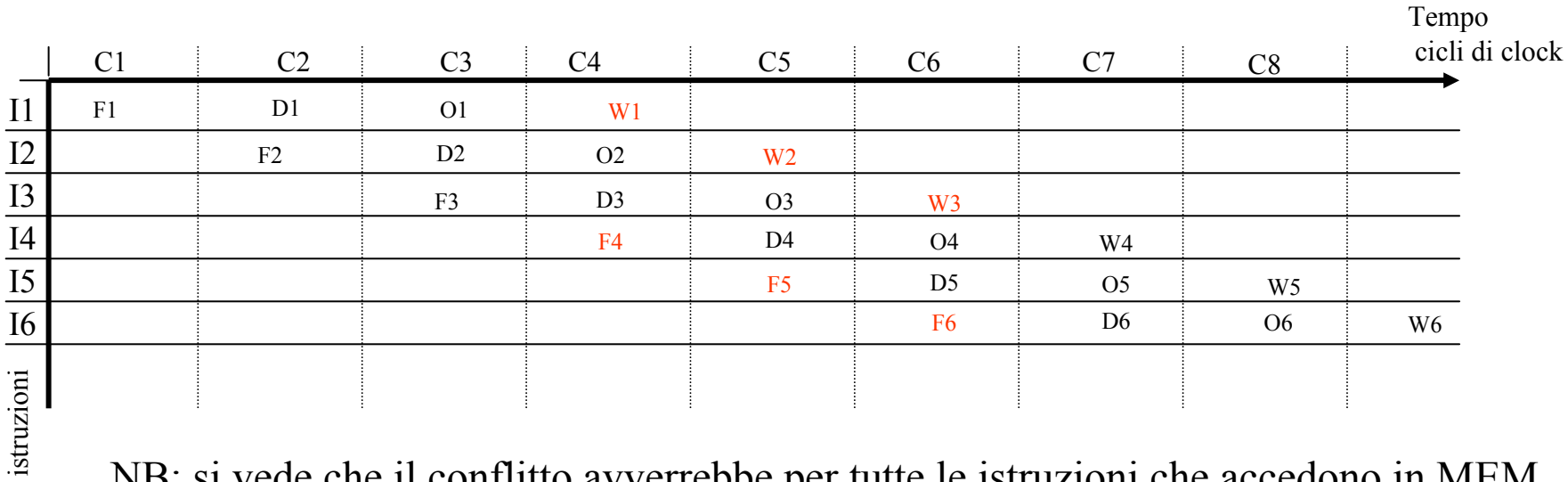
- L'unità di rilevazione criticità sulla base dei segnali provenienti dallo stadio O rileva la necessità di mettere in stallo la pipeline (S=O)
- Impedisce la scrittura di PC e dei due buffer F/D e D/O (attesa degli stadi di fetch, decode, execute)
- Forza i segnali di controllo del buffer O/W in modo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

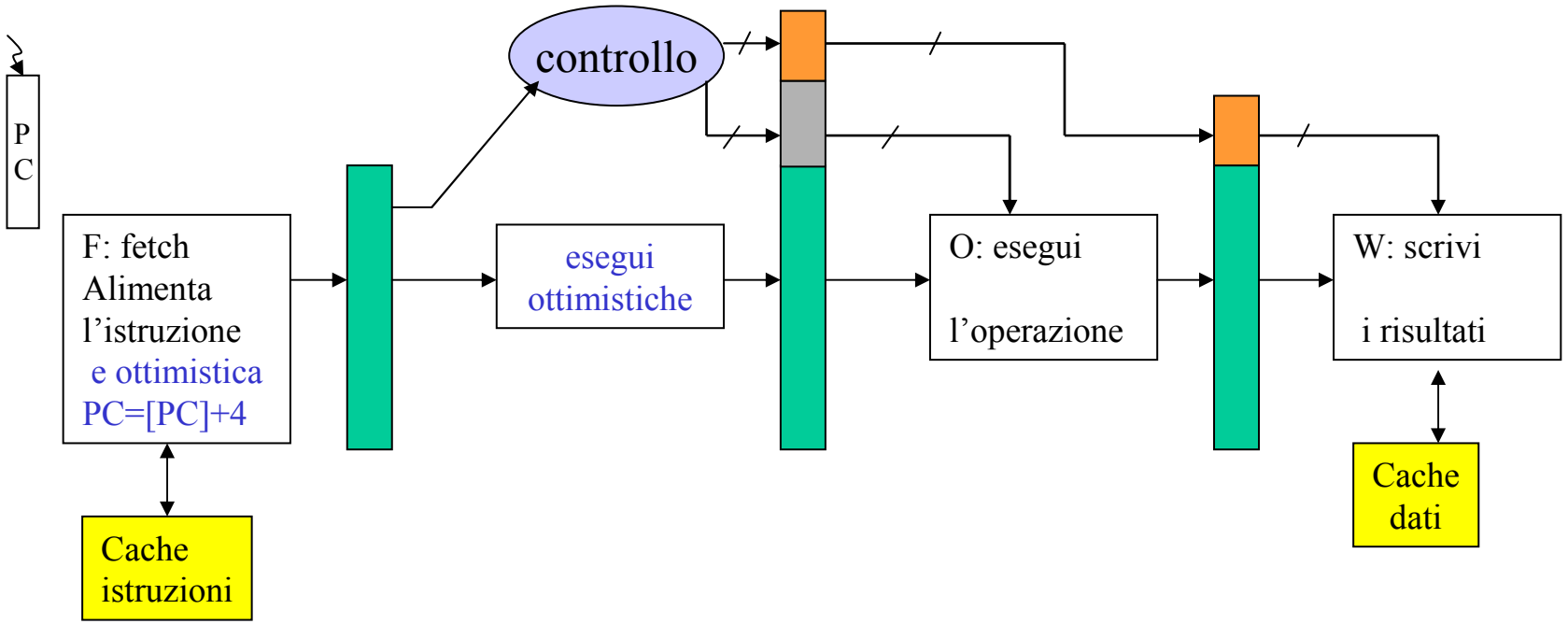




NB: si vede che il conflitto avverrebbe per tutte le istruzioni che accedono in MEM

Usare lo stallo è potenzialmente molto inefficiente: se molte istruzioni accedono in memoria, potrei dover 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 tutte le criticità strutturali

➡ 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 a priori (a parte il caso di operazioni lunghe). Le altre criticità sono gestite con tecniche specifiche che vedremo.

NB: è evidente che l'aumento del numero di stadi rende più probabile il verificarsi di criticità strutturali.

Criticità sui dati

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

$A \leftarrow 5$

. . .

$A \leftarrow 3 + A$

$B \leftarrow 4 \text{ X } A$

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

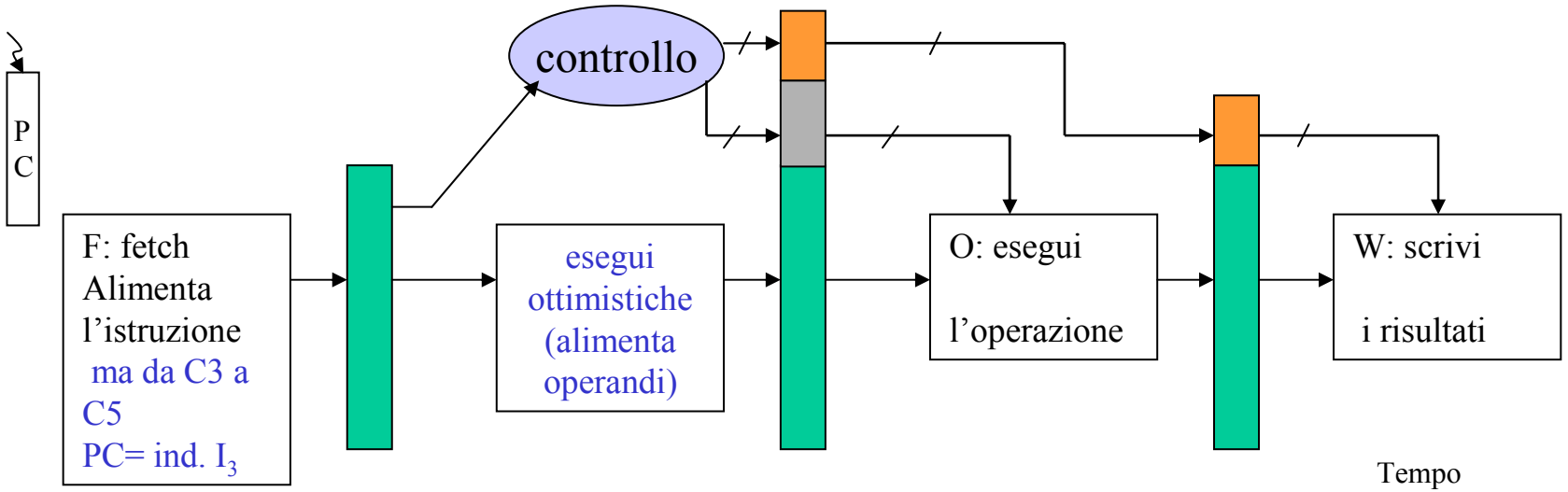
$A \leftarrow 5 + C$

$B \leftarrow 20 + C$

(istruzioni indipendenti)

Primo rimedio: gestire criticità soltanto con stallo pipeline

……
I1: Mul R2,R3,R4 [NB: la destinazione è il terzo registro in questo formato]
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

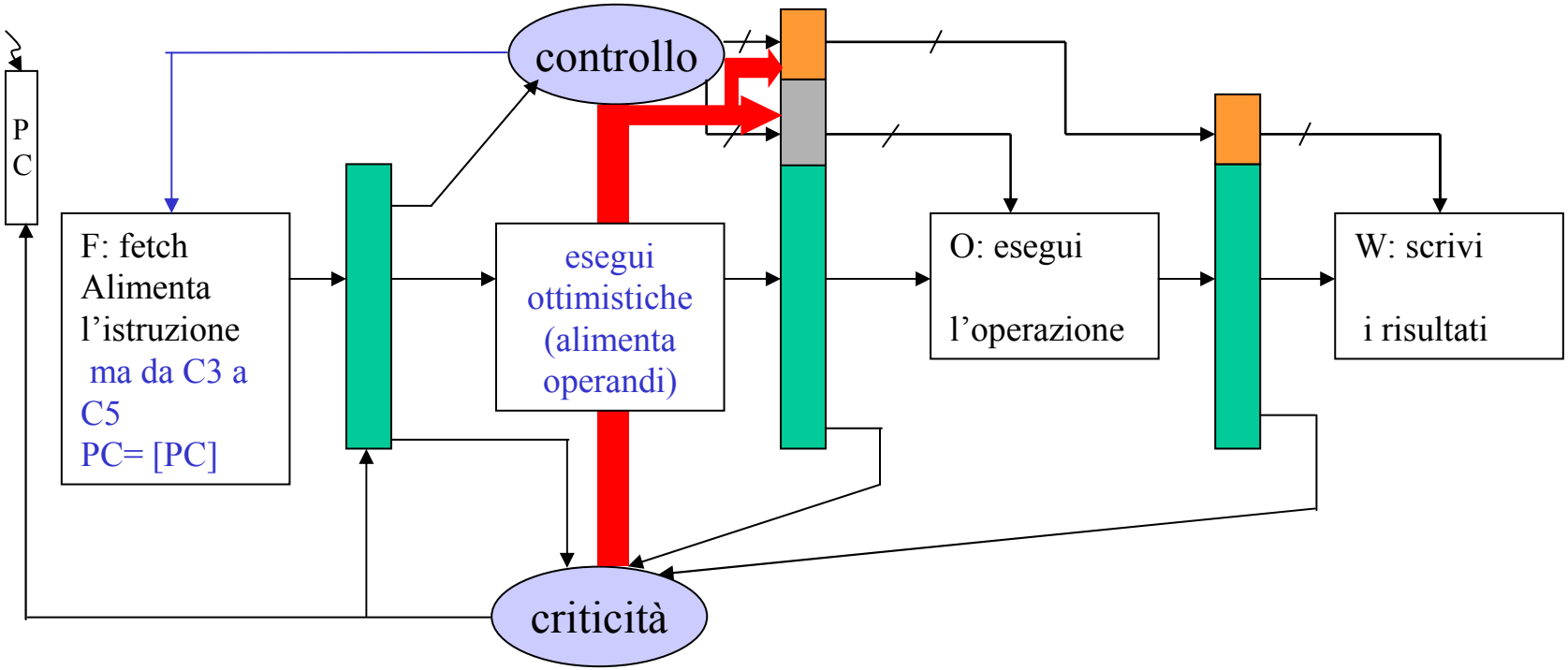
Rilevazione criticità: D2 non può procedere a prelevare R4

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 proseguono nella loro esecuzione]

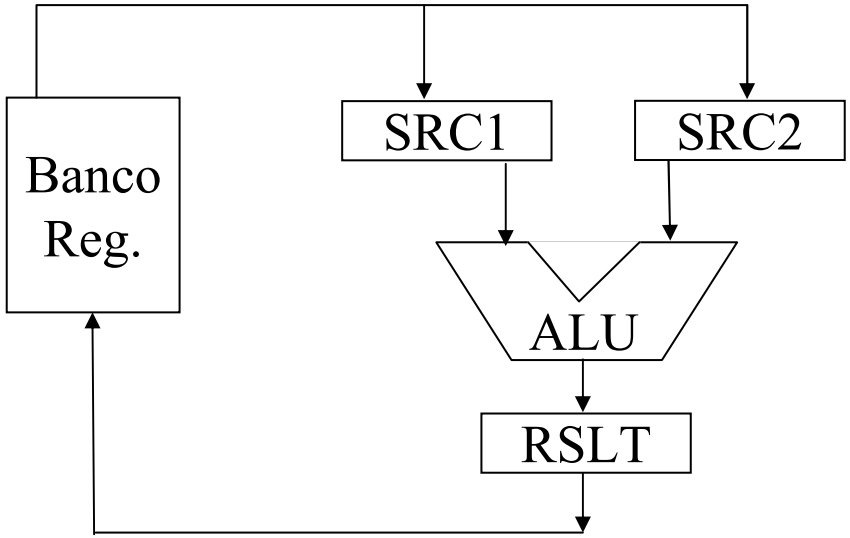
NB: la gestione dello "stallo" corrisponde a quanto visto nel caso delle criticità strutturali



NB: ovviamente, sono da aggiungere le unità di gestione
delle criticità per gestire le criticità strutturali viste in precedenza.

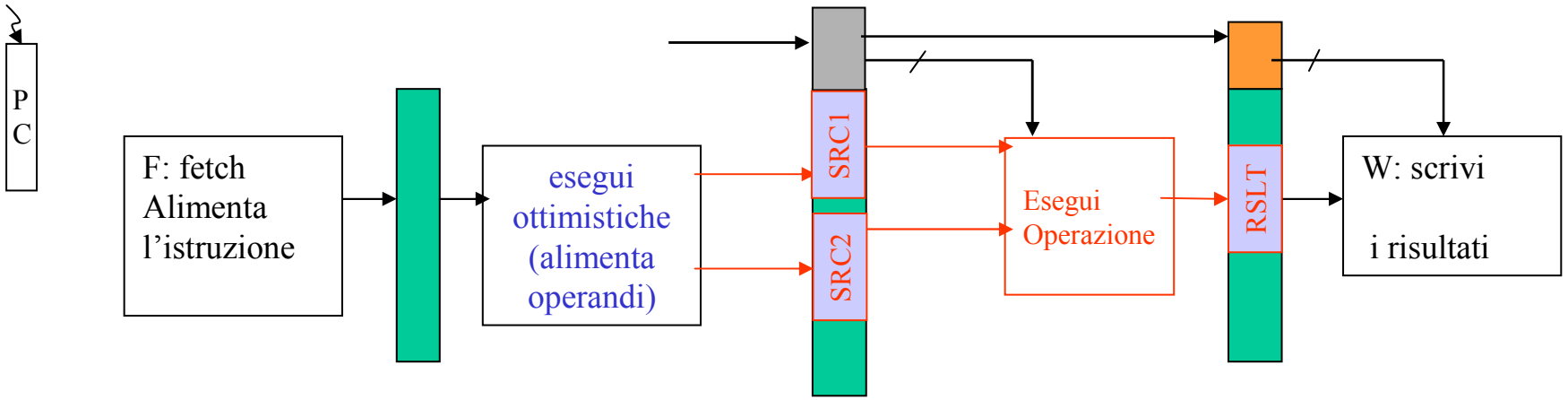
Secondo rimedio: propagazione dei dati

• 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

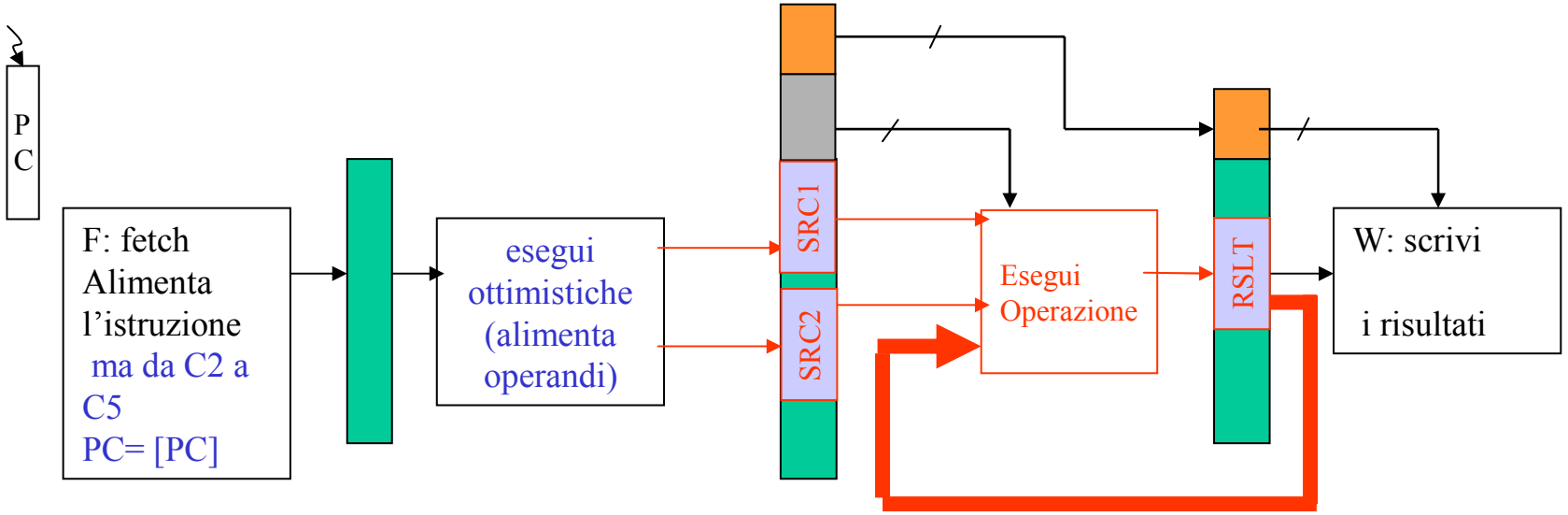
corrispondente a:



Rivediamo il caso precedente

……
I1: Mul R2,R3,R4
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



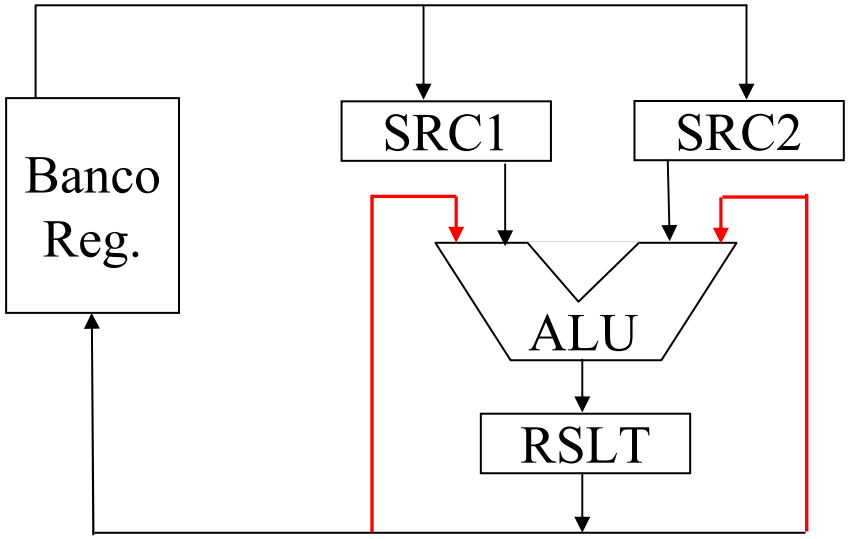
	C1	C2	C3	C4	C5	C6	C7	C8
I1	F1	D1	R2*R3	R4←				
I2		F2	D2	R5+R4	W2			
I3			F3	D3	O3	W3		
I4				F4	D4	O4	W4	
I5					F5	D5	O5	
I6						F6	D6	

Tempo in cicli di clock

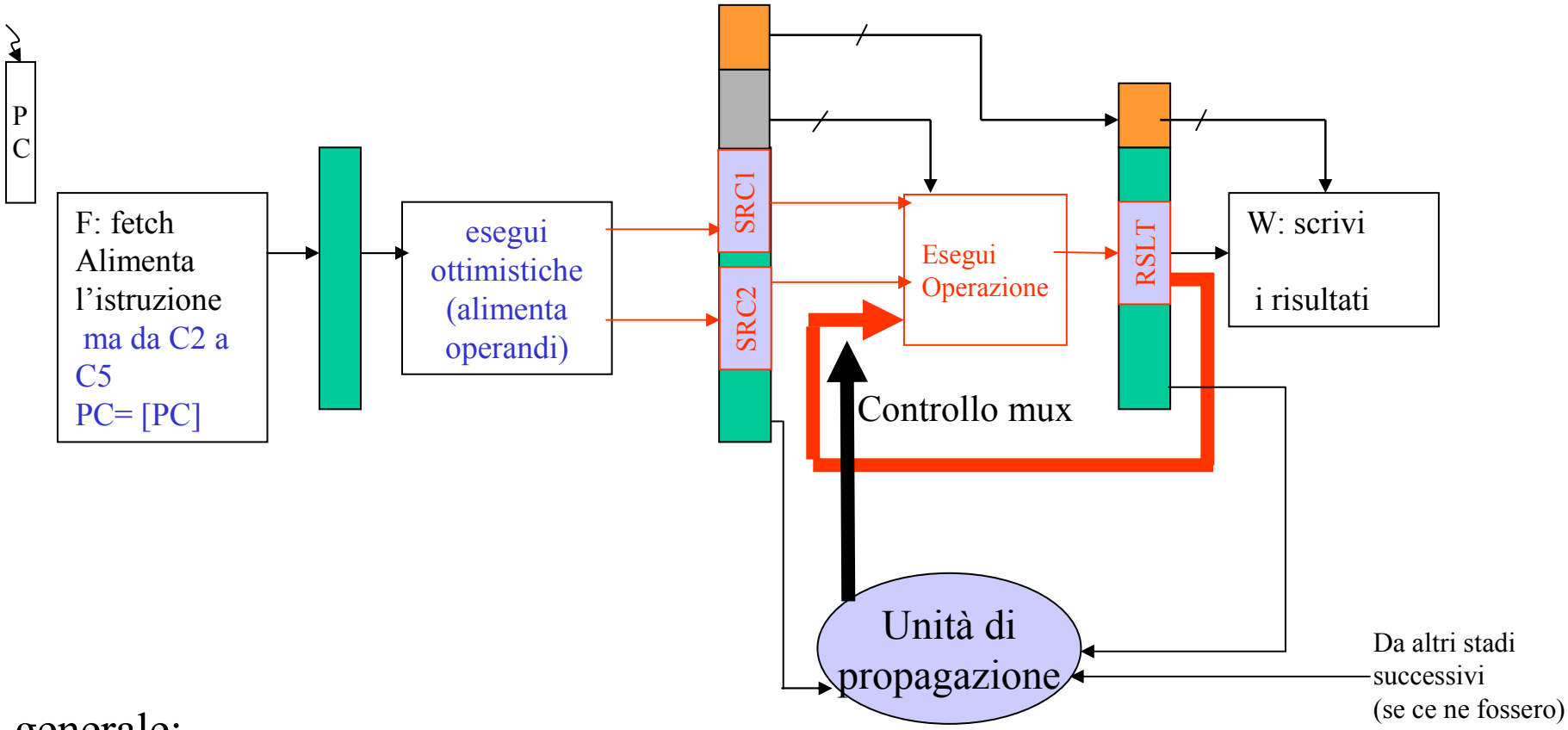
istruzioni

SI EVITA STALLO DI DUE CICLI DI CLOCK!

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



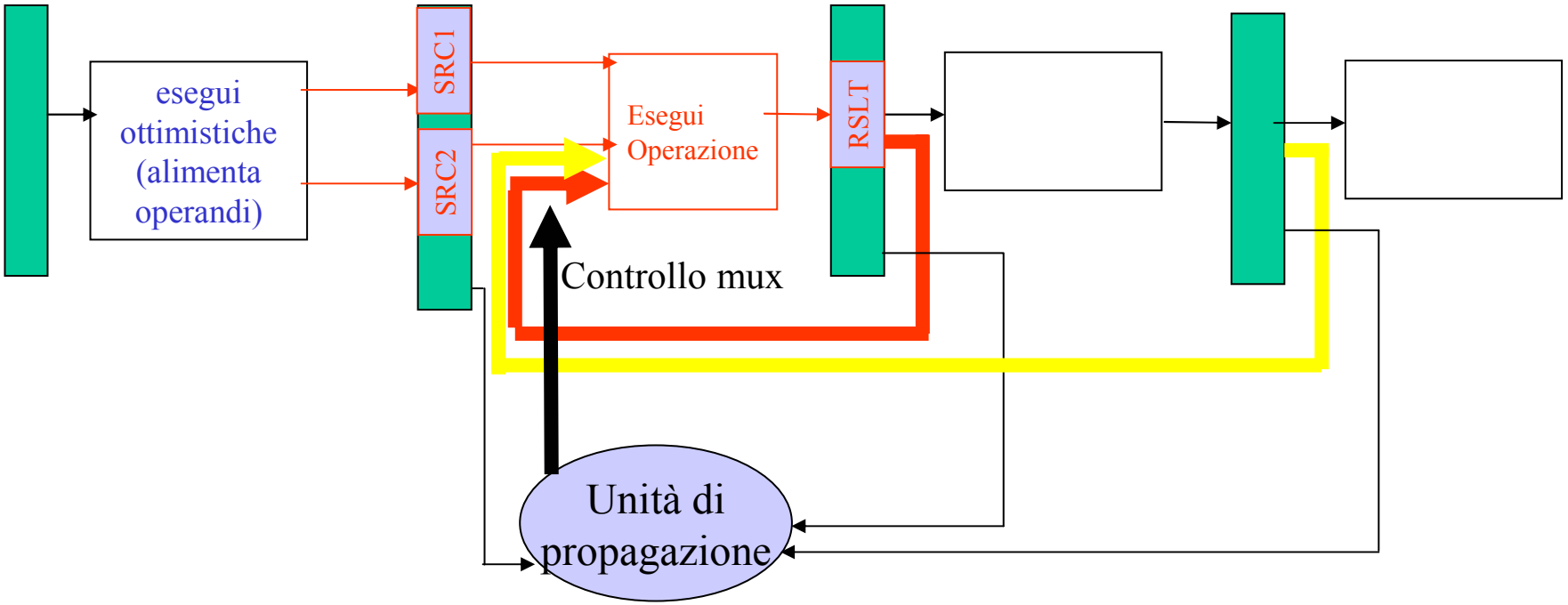
• Occorre una unità di propagazione che:
  - identifichi la situazione di potenziale criticità che può essere risolta con la propagazione
  - asserisca i segnali nei mux (omessi in figura) per determinare opportuno cammino dei dati



In generale:

- 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})

• Cosa succede 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. Questa si trova nello stadio più a sinistra (il meno avanzato)

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

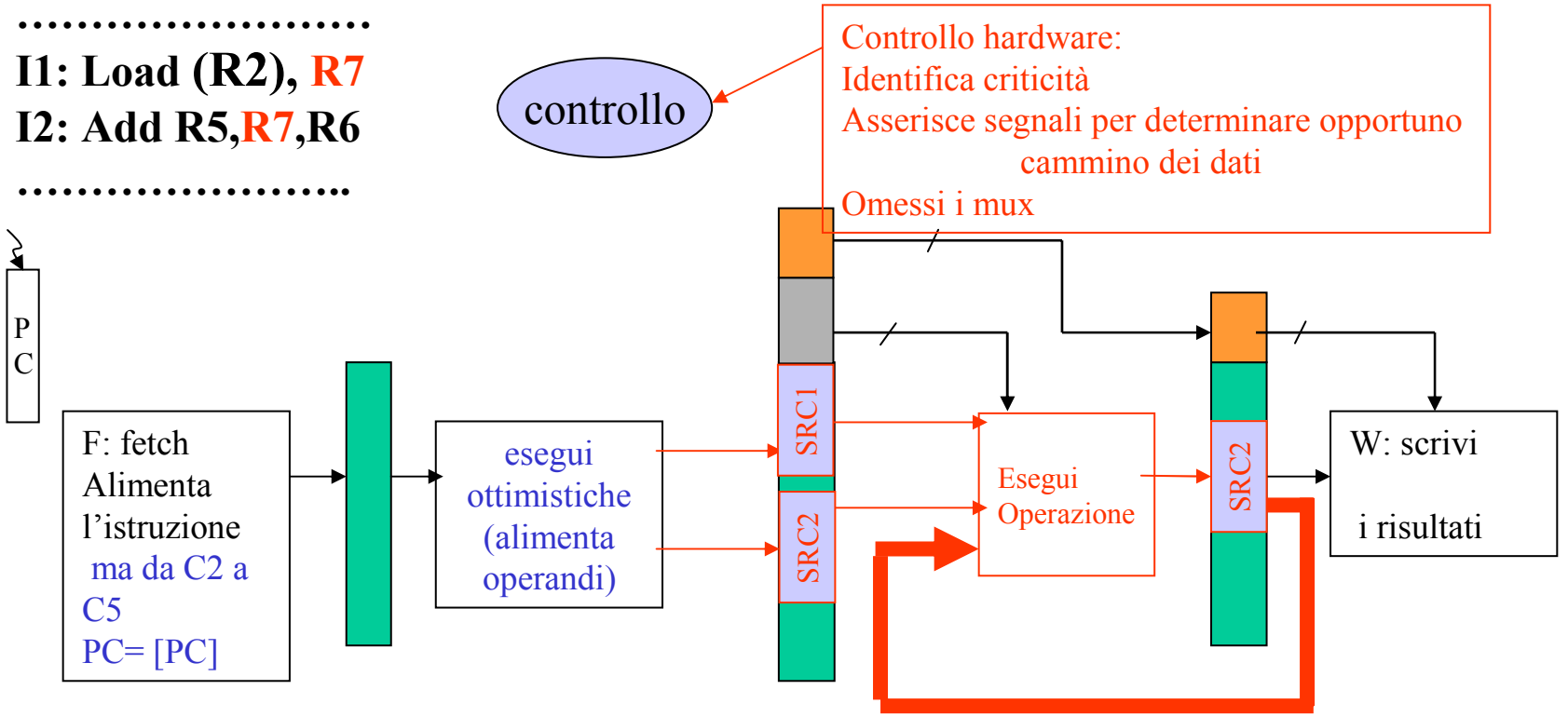
Altro caso di propagazione

……

I1: Load (R2), R7

I2: Add R5,R7,R6

……..



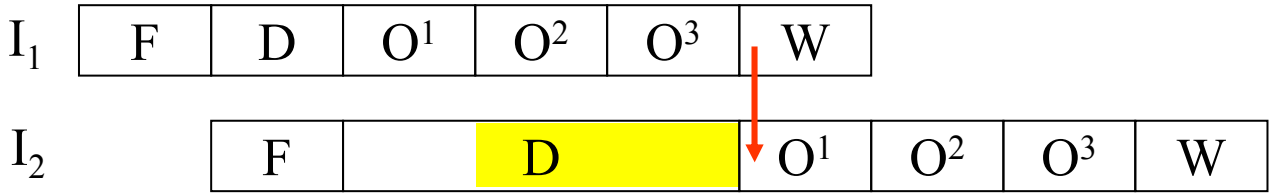
	C1	C2	C3	C4	C5	C6	C7	C8
I1	F1	D1	Read	R7←				
I2		F2	D2	O2 [+]	W2			
I3			F3	D3	O3	W3		
I4				F4	D4	O4	W4	
I5					F5	D5	O5	
I6						F6	D6	

Tempo in cicli di clock

istruzioni

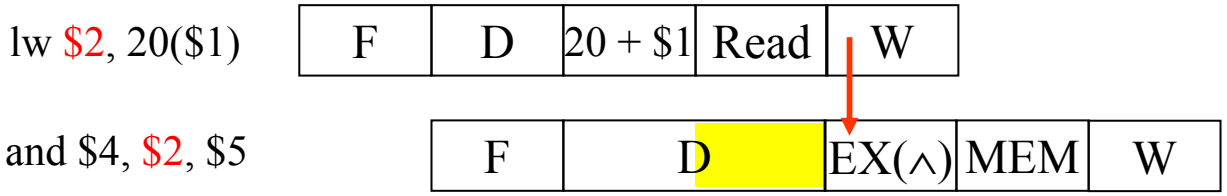
Purtroppo, la propagazione in generale non risolve tutte le criticità sui dati…

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 un ciclo 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

In ogni caso, si vede come l'aumento del numero di stadi:

- Rende più difficile risolvere le criticità con propagazione
- Tende a far aumentare il numero di cicli di clock di stallo pipeline

In base a quanto visto, sono presenti tre unità di controllo:

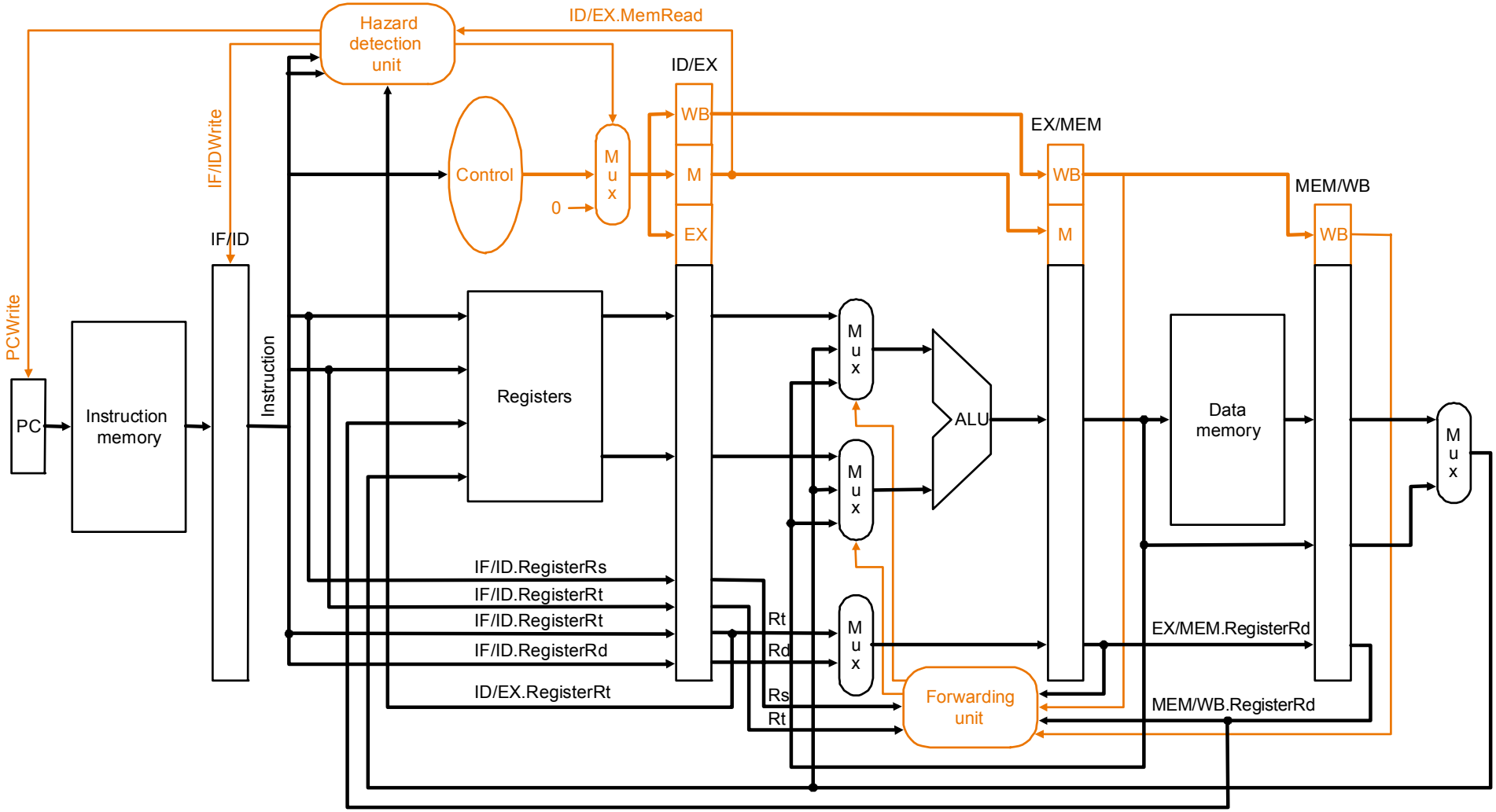
- Unità di controllo: determina nello stadio D i segnali di controllo da assegnare al buffer interstadio D/D+1

- 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 riconoscerebbe più criticità. Si torna al caso "rimedio con solo stallo".

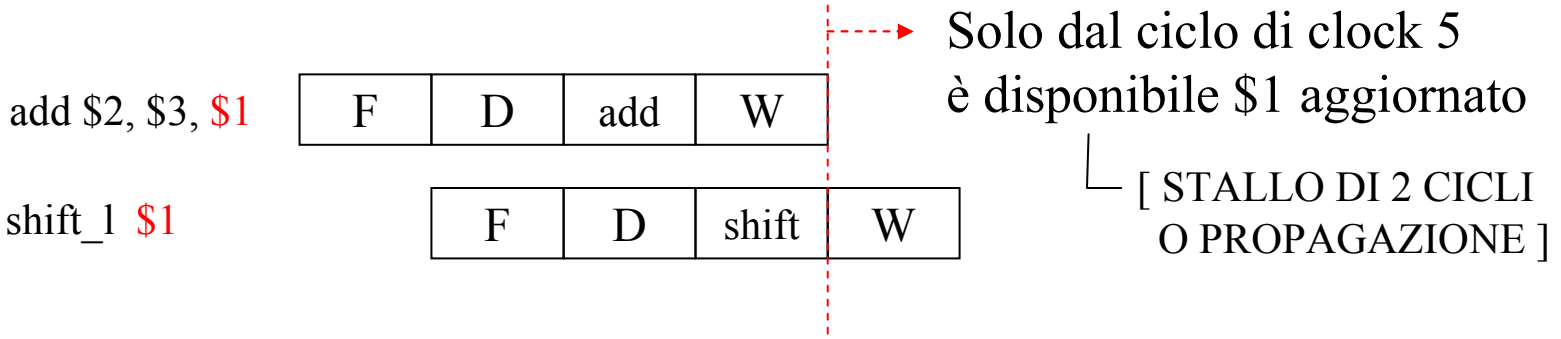
Esempio MIPS: pipeline a 5 stadi, unità di rilevazione criticità gestisce "carica-e-usa"



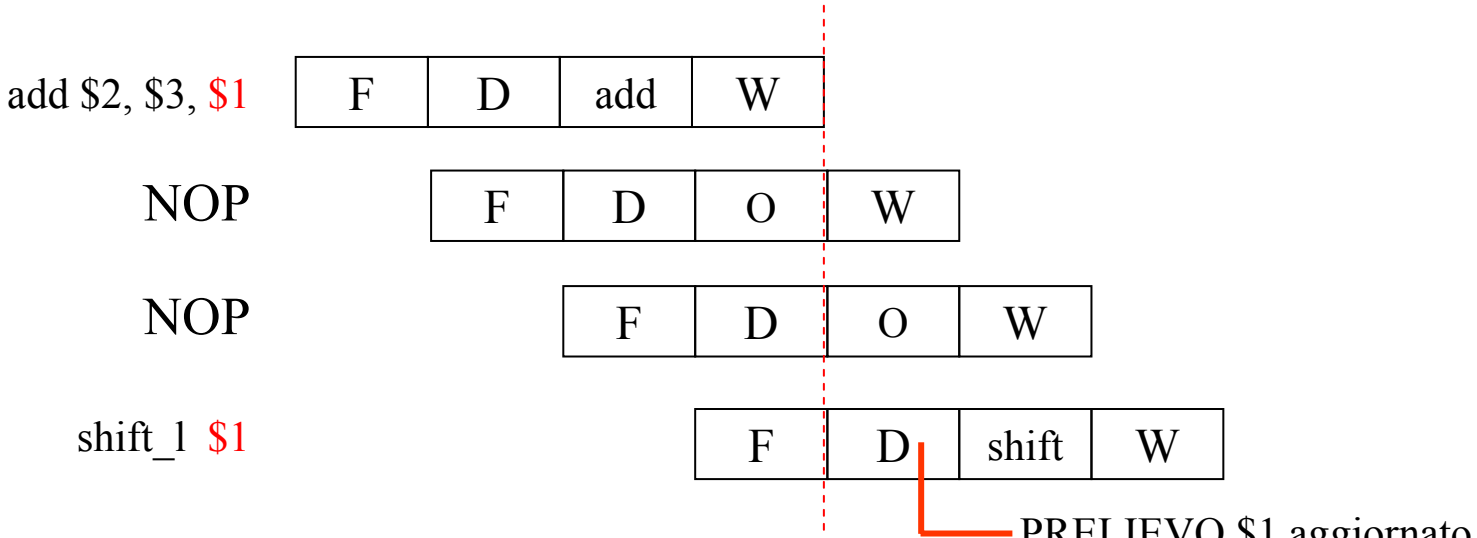
NB: Hazard Detection Unit riconosce se, quando l'istruzione che "usa" il dato è in D, nello stadio EX c'è istruzione lw (istruzione lw in MEM è gestita da propagazione, istruzione lw nello stadio WB è gestita dall'HW particolare del register file usato nel testo)

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:

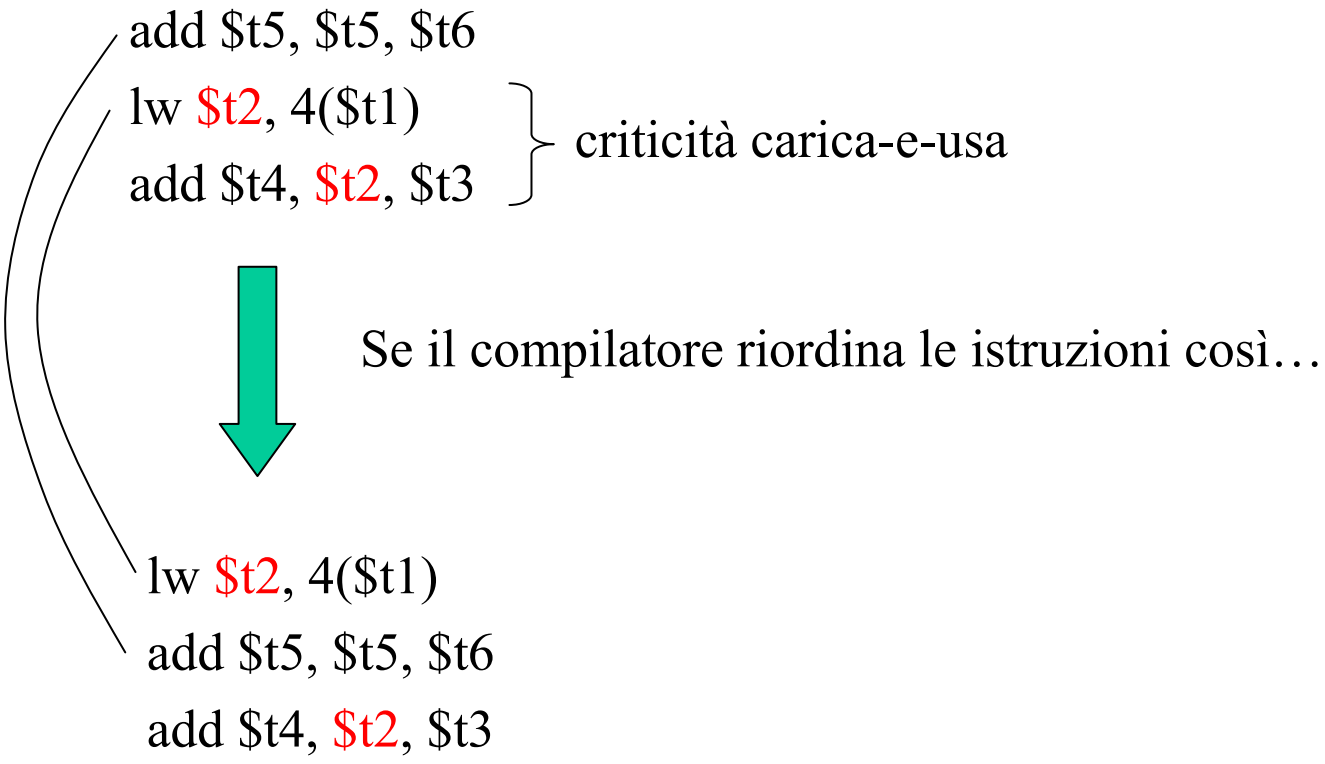
1. Criticità gestita completamente dal compilatore (no stallo pipeline)
2. Gestione criticità via stallo, senza propagazione
3. Gestione criticità con stallo e propagazione
   (che diminuisce il numero di cicli di stallo)

NB: le soluzioni 1) e 2) sono alternative e di per sé l'una o l'altra 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 eliminare le dipendenze [istruzioni indipendenti in sequenza] 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 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 → decode → Registro Interstadio

    (nello stadio di "decodifica" solo ottimistiche)

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

    ⇒ stallo "su" uno stadio S (es. stadio di fetch | esecuzione)

    ⇒ 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

    ⇒ unità di propagazione dei dati: propagazione verso uno stadio S:

    propaga dati da uno stadio successivo

    (riceve info su stadio S e successivi!)

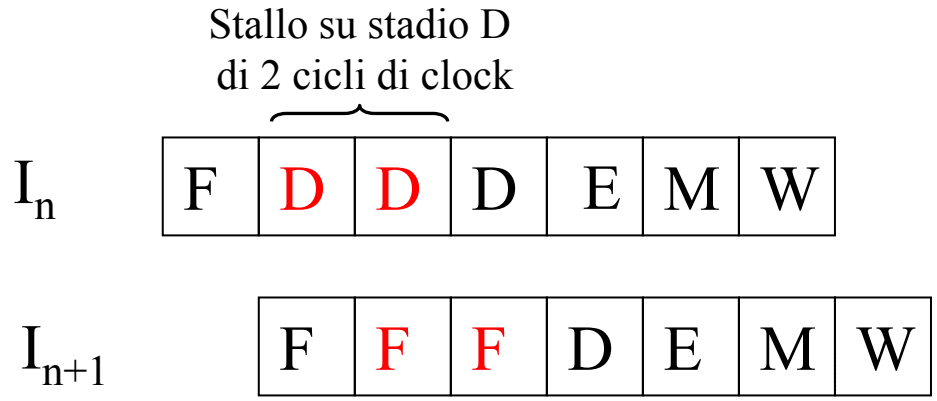
    NB: conviene propagare verso lo stadio dove viene usato il dato:

    è possibile "aspettare" più istruzioni che potrebbero produrlo

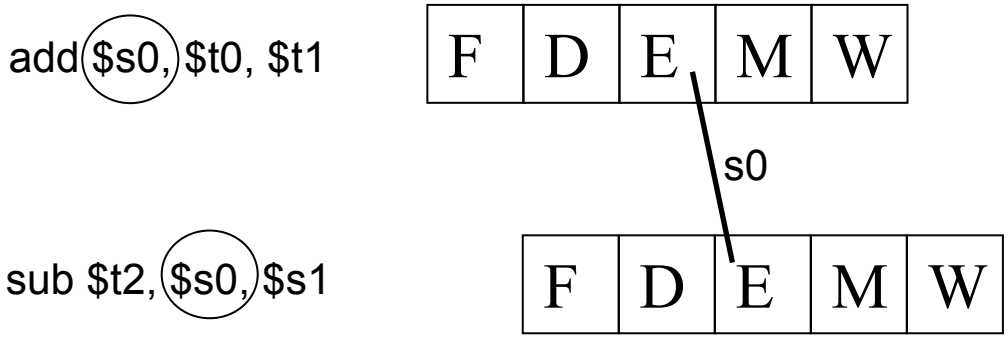
⇒ UNITA' DI PROPAGAZIONE "SGRAVA" UNITA' RILEVAZ. CRITICITA'

Rappresentazione mediante diagramma temporale [a più cicli di clock]

Es. Pipeline a 5 stadi corrispondenti a quelli del MIPS



Es. Rappresentazione della propagazione

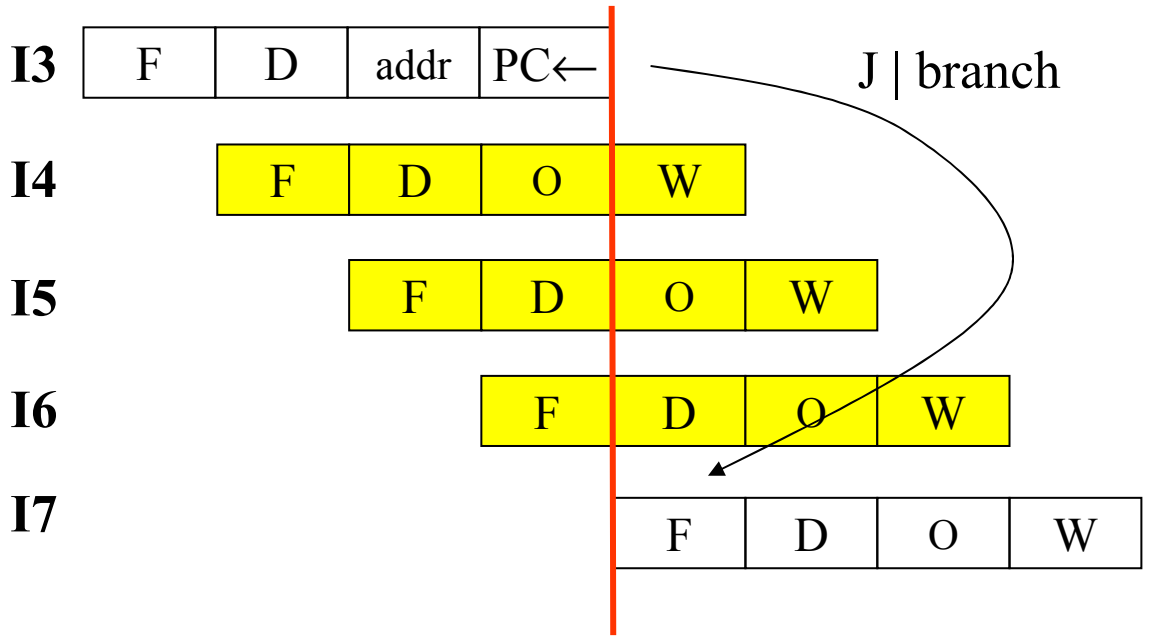


Propagazione verso stadio E (da stadio M)

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: …
……..



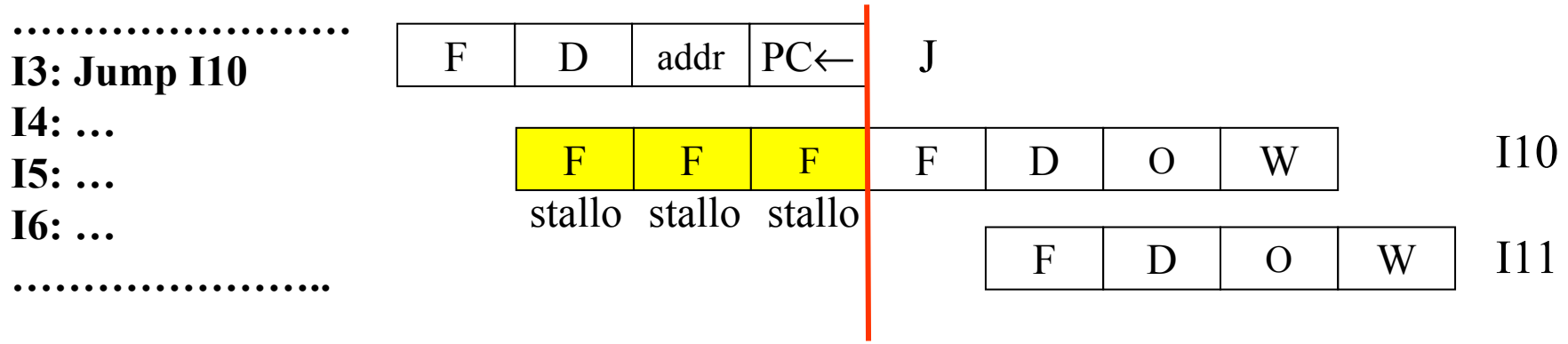
⇨ Se il salto è effettuato, I4, I5, I6 vengono eseguite comunque, mentre l'istruzione a cui si salta viene caricata ed eseguita successivamente

• Due tipi di salti:
- salto incondizionato (jump)
- salto condizionato o diramazione (branch)

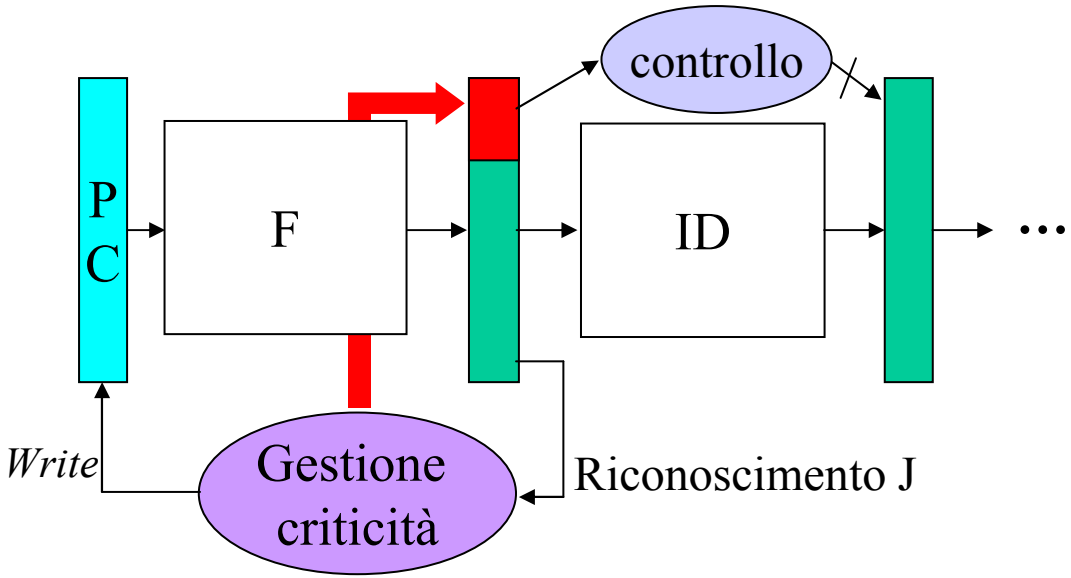
Primo rimedio: stallo della pipeline

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)

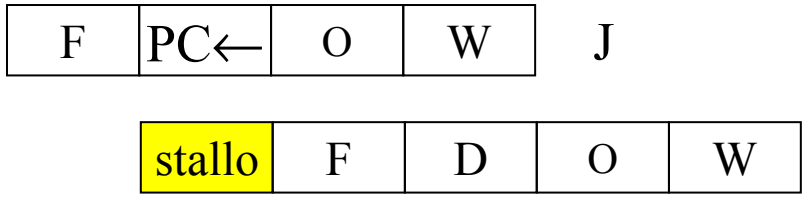


Durante il ciclo di clock:
- si riconosce J
- si evita scrittura PC al fronte del clock successivo [per n=3 fronti successivi]
- si scrive la "bolla" [per n=3 fronti successivi]

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

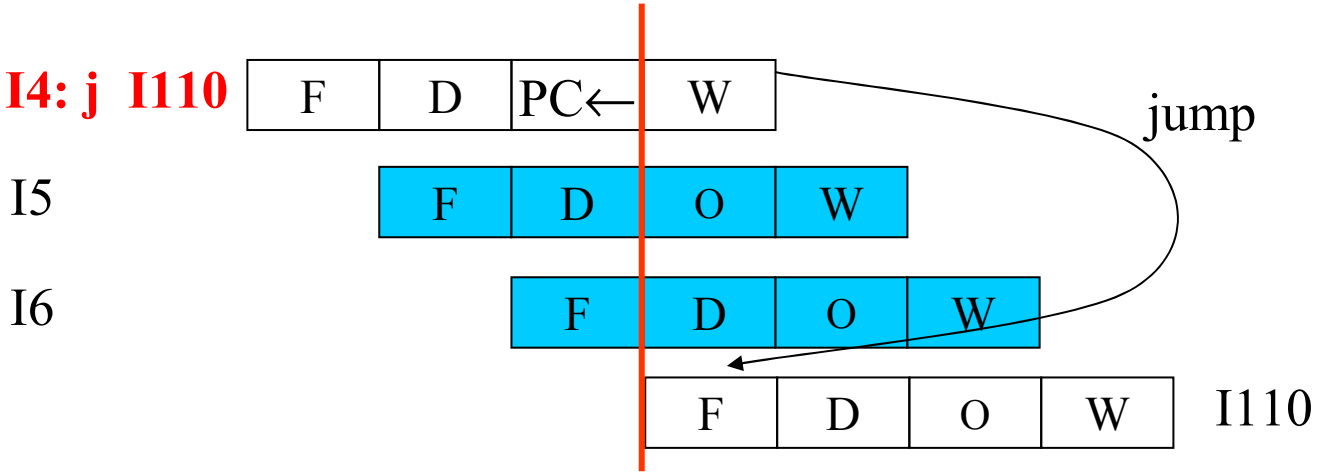
- Come nel caso delle dipendenze tra dati, la gestione è demandata interamente al software (compilatore): non presente l'unità di gestione criticità!

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…

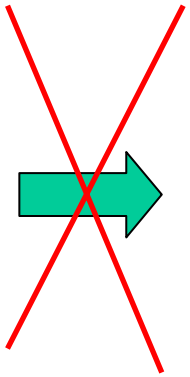
Salto condizionato: SUPPONIAMO CHE SI ABBIA UN SOLO SLOT

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
……..                          ……..
```

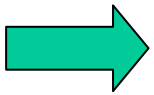


[in tal caso la condizione di salto deve essere successiva a sub]

⟹ Si usa una delle due strategie seguenti…

2. Istruzione presa dal codice destinazione

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



- 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

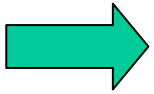
```
add R7, R1, R3
mul R5, R7, R1
```

l'introduzione sarebbe errata [add altera l'operando usato da mul]

- Tipicamente, il registro scritto dall'istruzione aggiunta è "temporaneo" all'interno del ciclo, mentre non ha visibilità [valore indifferente] all'esterno

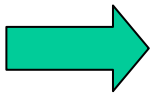
NB: questa tecnica ha senso se la probabilità che il branch venga effettuato è alta, come nel caso in cui il branch sia la condizione di uscita dal ciclo

```
……..
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

- In ogni caso, l'aumento del numero di stadi della pipeline tende ad aumentare il numero di slot, complicando il lavoro del compilatore.
- 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)
- Quando la previsione non può essere fatta in compilazione, sono più efficaci le tecniche di previsione dinamica (si vedranno in seguito).

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 quelle 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 che il salto sia sempre eseguito (se non è eseguito le istruzioni in più vengono scartate)

TECNICHE PREDIZIONE DINAMICA: realizzata dal processore

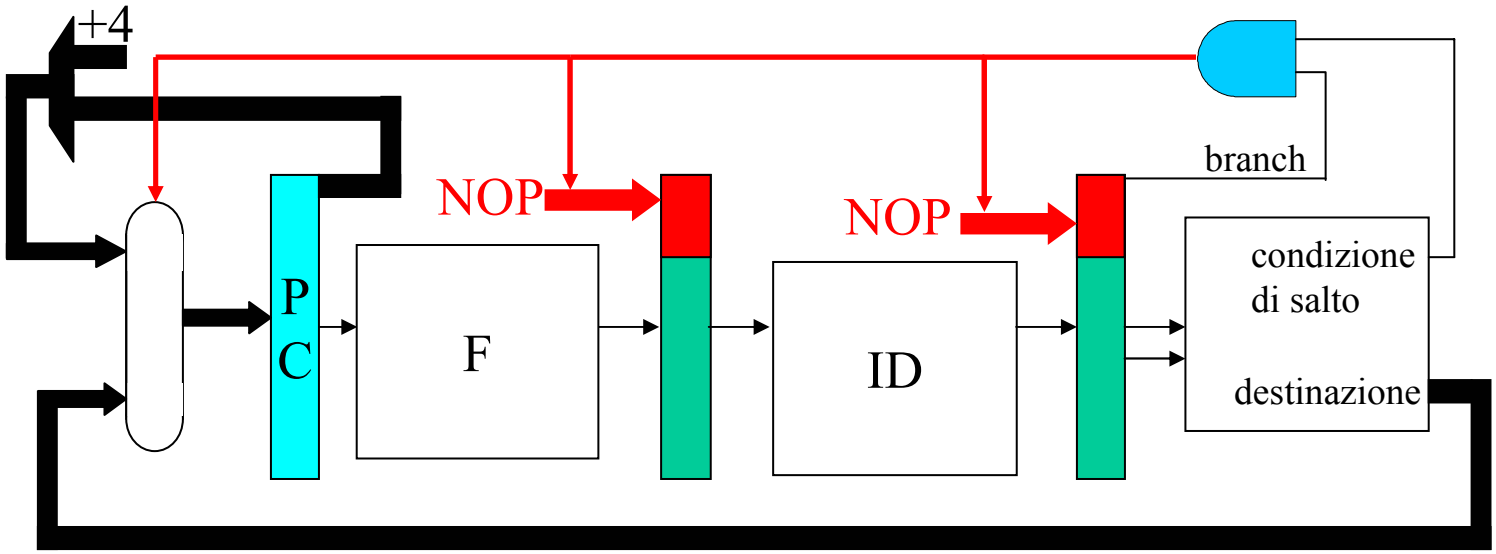
- La predizione è effettuata dal processore dinamicamente: la stessa istruzione in diverse istanze di esecuzione può avere predizioni diverse, sulla base della storia passata (predizioni passate esatte o errate)

TECNICHE DI PREVISIONE STATICA

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

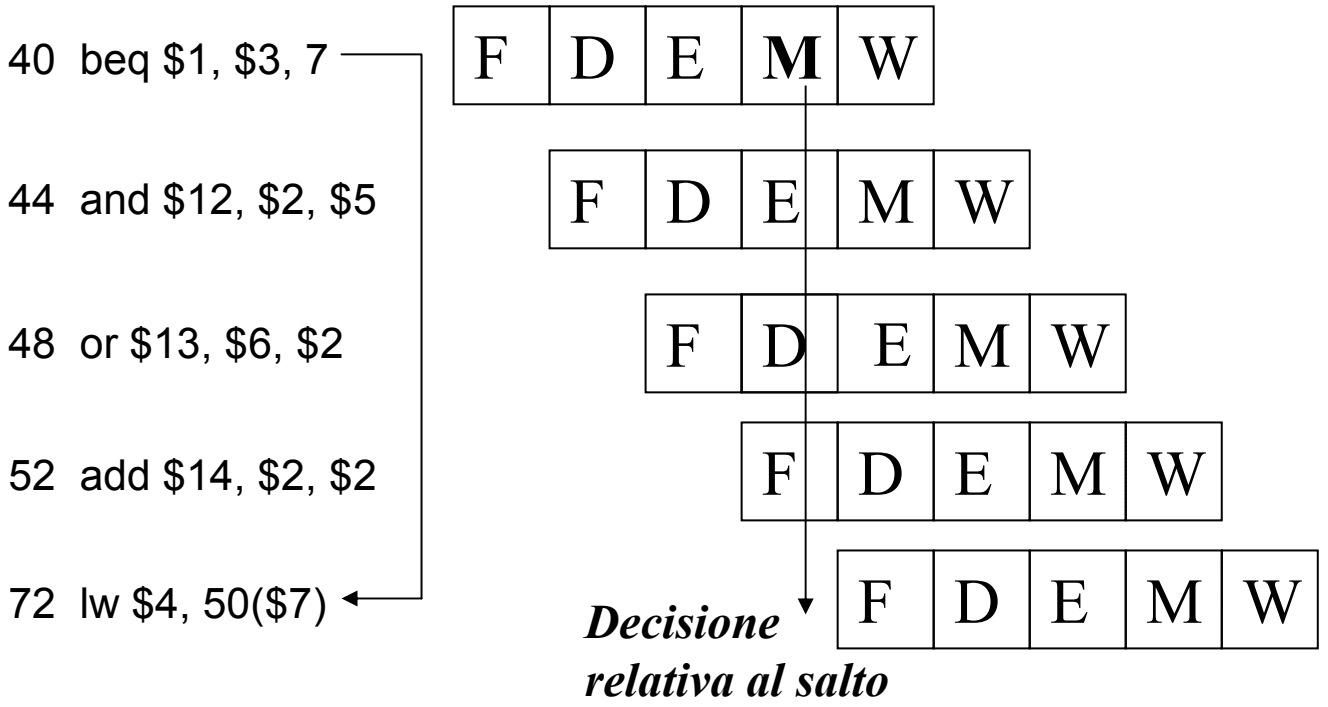
- Se appare un salto condizionato, si continuano a prelevare le istruzioni successive.
- Se la verifica della condizione di salto indica che il salto non deve essere effettuato, tutto procede normalmente.
- Se il salto deve essere effettuato, alla verifica della condizione di salto le istruzioni già 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

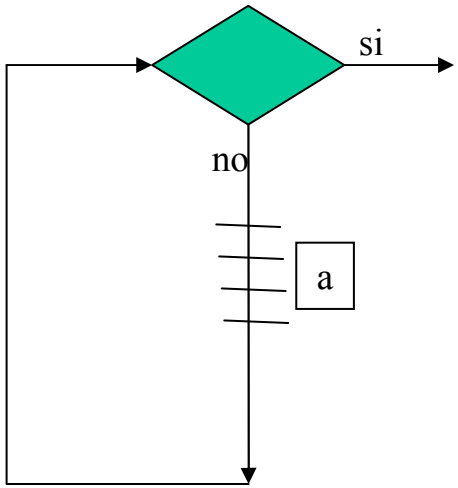


Se il salto viene effettuato, le istruzioni 44, 48, 52 vanno scartate, si introduce un ritardo di 3 cicli di clock

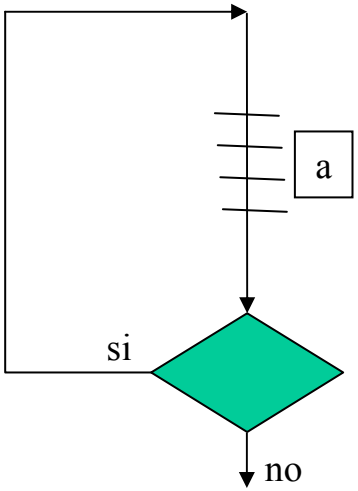
Esempio

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: in caso di previsione di salto, l'unità di fetch è costruita per alimentare le istruzioni come se saltasse: vedremo in seguito quali tecniche si usano.

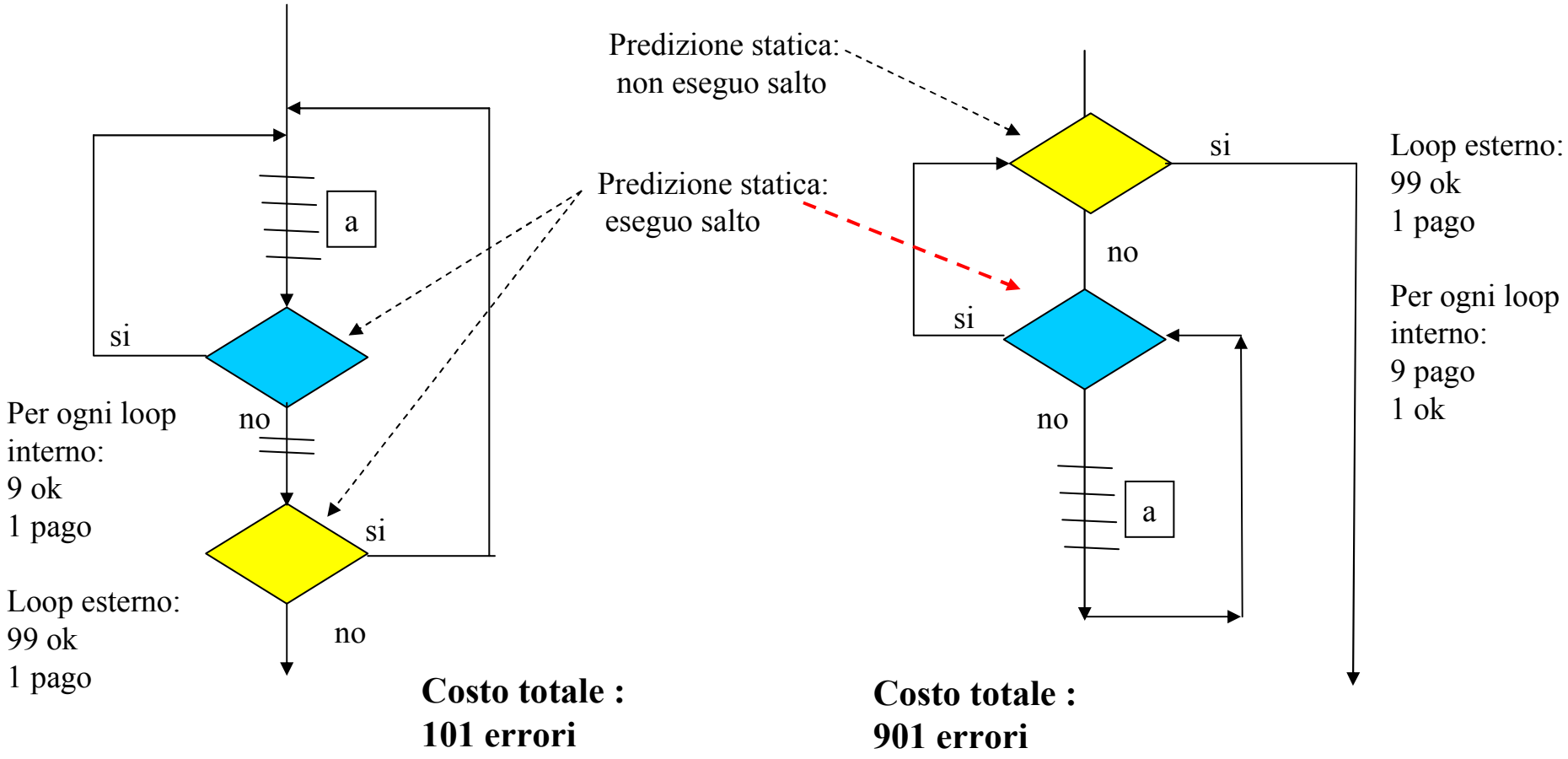
Problema:

- 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)

Previsione statica : i costi

[Hyp. di predizione secondo il salto a indirizzi minori o maggiori]

Dai dati: 10 giri interni (a) 100 esterni



Costo totale :
101 errori

Costo totale :
901 errori

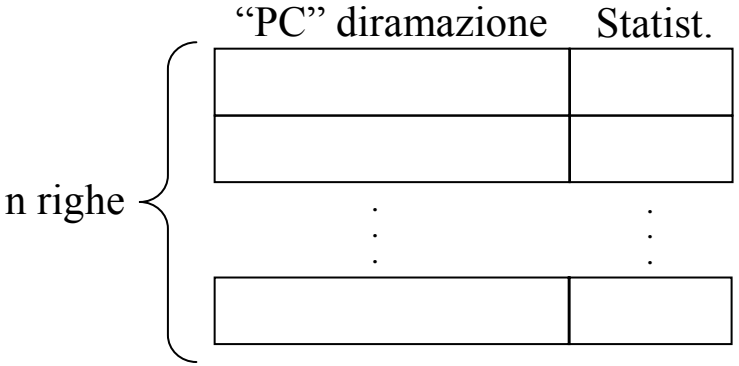
IN OGNI CASO, I DATI DI UTILIZZO DEL PROGRAMMA POSSONO ESSERE DIVERSI DA QUELLI PREVISTI IN FASE DI COMPILAZIONE…

TECNICHE DI PREVISIONE DINAMICA

• L'hardware incorpora algoritmi per stabilire la *verosimiglianza* che si prenda una via o l'altra in base alla *storia* recente (numero finito di passi).

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

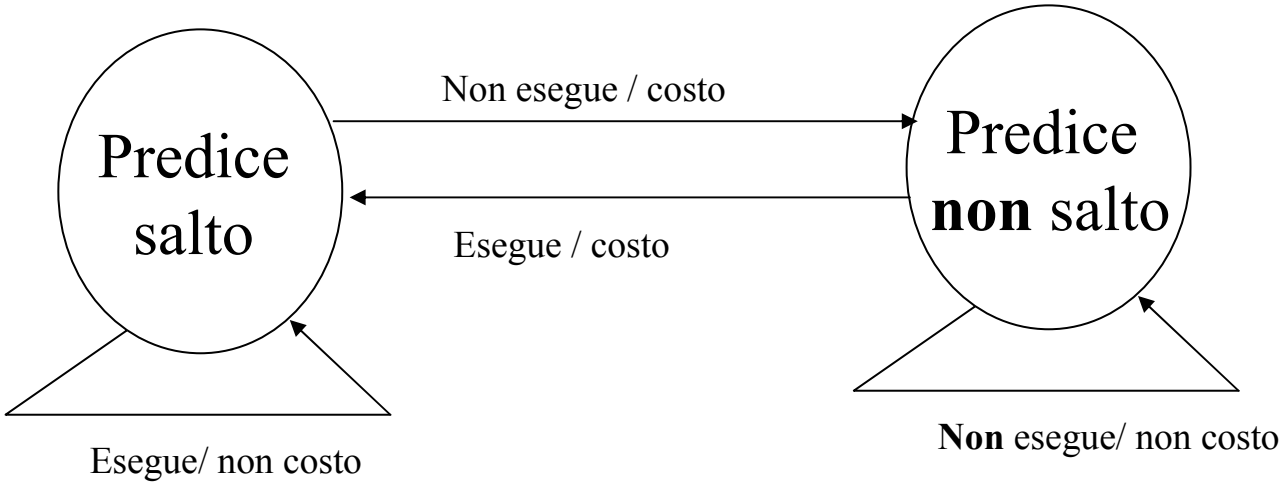
In scrittura: si "mescolano" le storie di diverse diramazioni

→ 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…

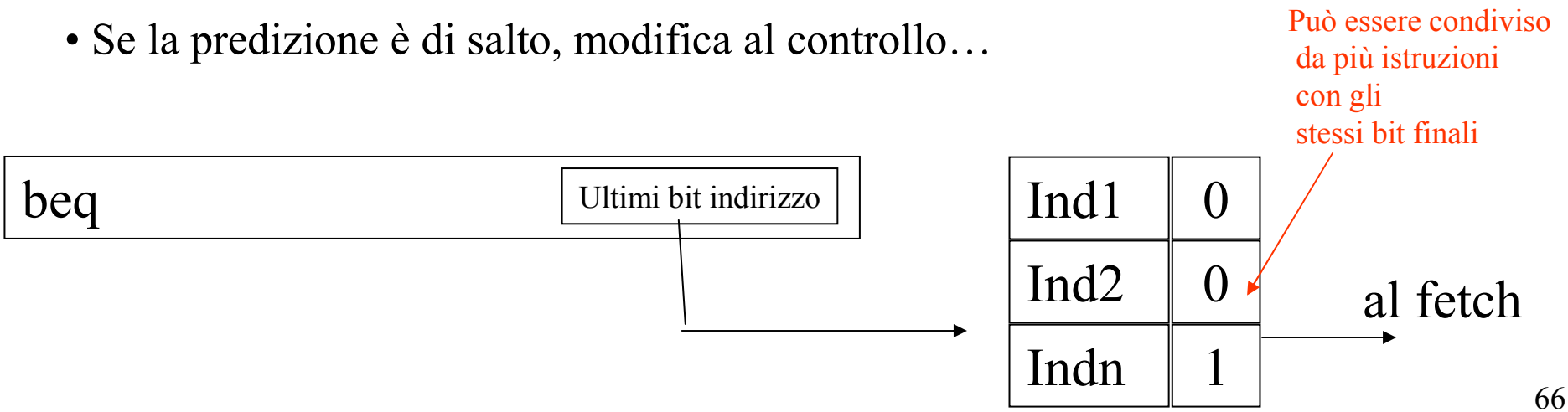
La soluzione più semplice: memoria a un passo



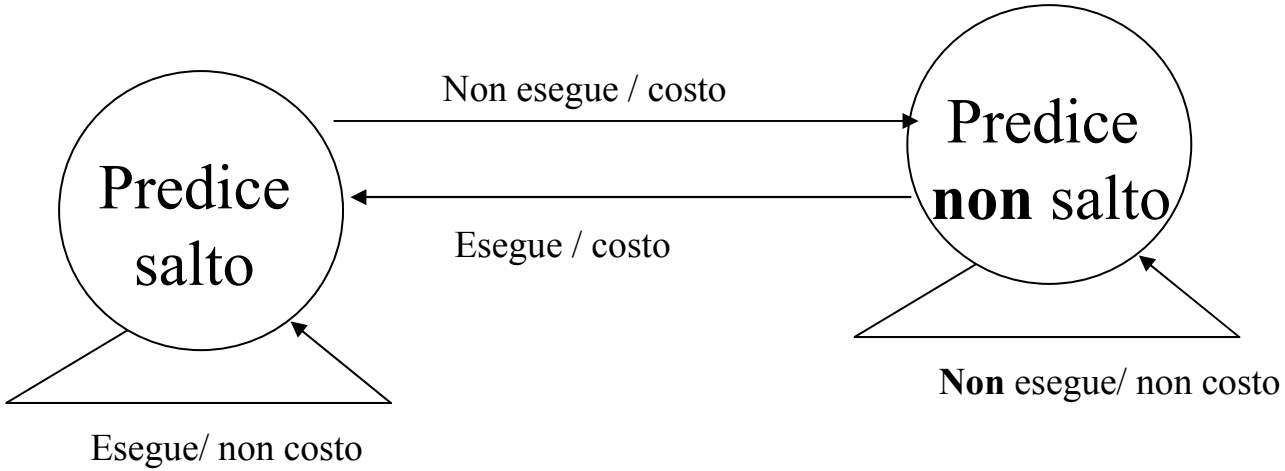
1 bit indica se ultima volta il salto è stato effettuato oppure no.

Modifica al controllo

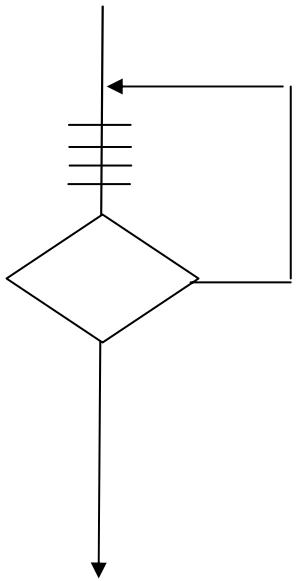
- Nella fase di fetch della diramazione, si accede a BPB; predizione valutata;
- Se la predizione è di salto, modifica al controllo…



Costi di questa prima realizzazione



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



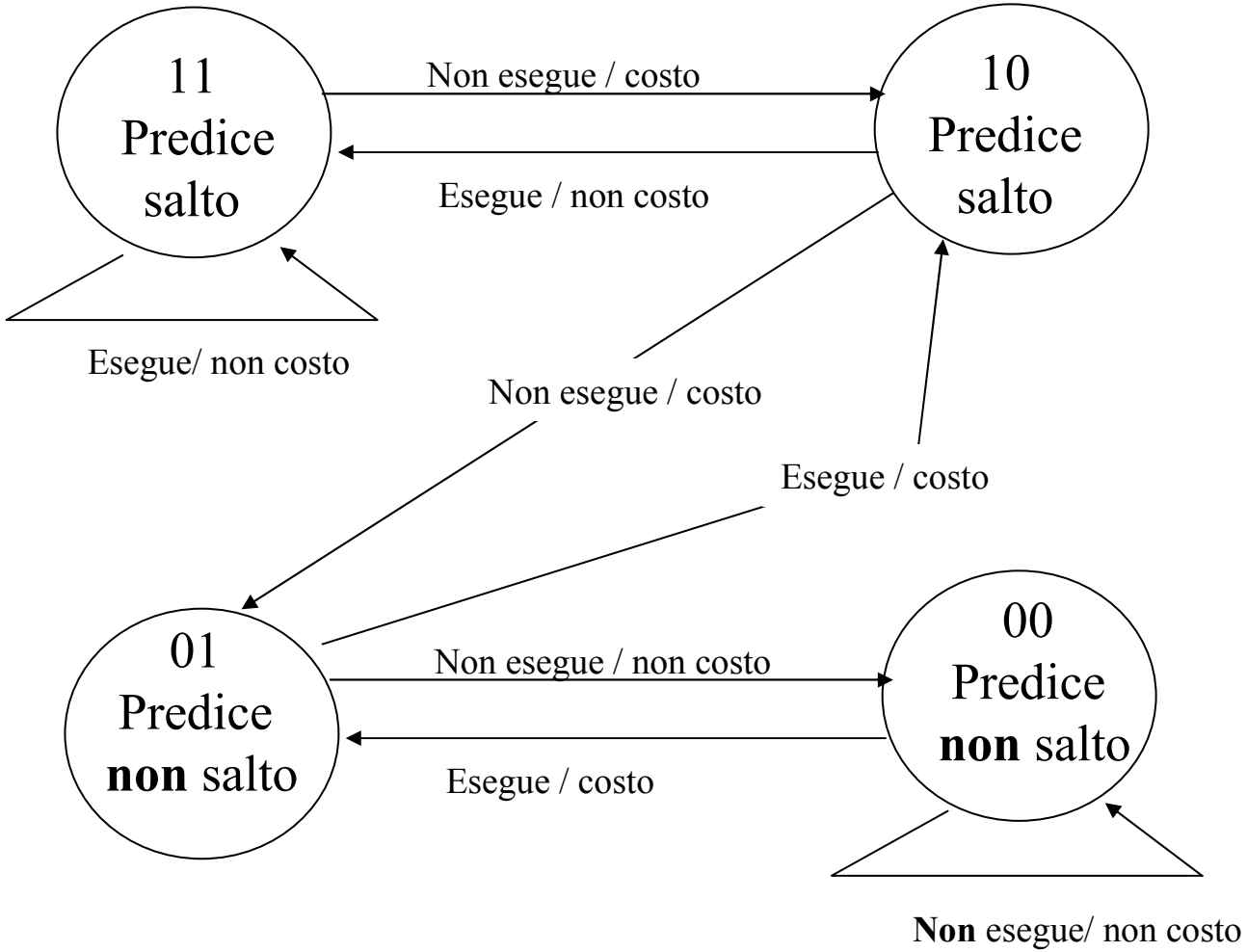
Es: 10 iterazioni del ciclo ogni volta che si incontra:

- la prima volta predice non salto
  (la volta precedente era uscito dal ciclo!)
  ⇒ errore
- 8 volte predizioni giuste
- ultima volta predice salto ⇒ errore

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

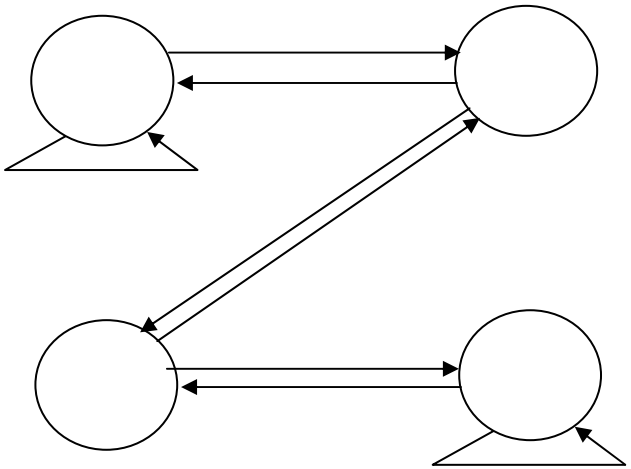
Un'altra soluzione: più memoria

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

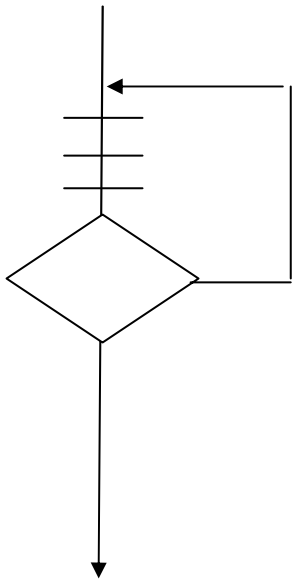


Predizione con contatore a saturazione a quattro stati:
incrementato quando salto effettuato, decrementato altrimenti.

Costi di questa realizzazione



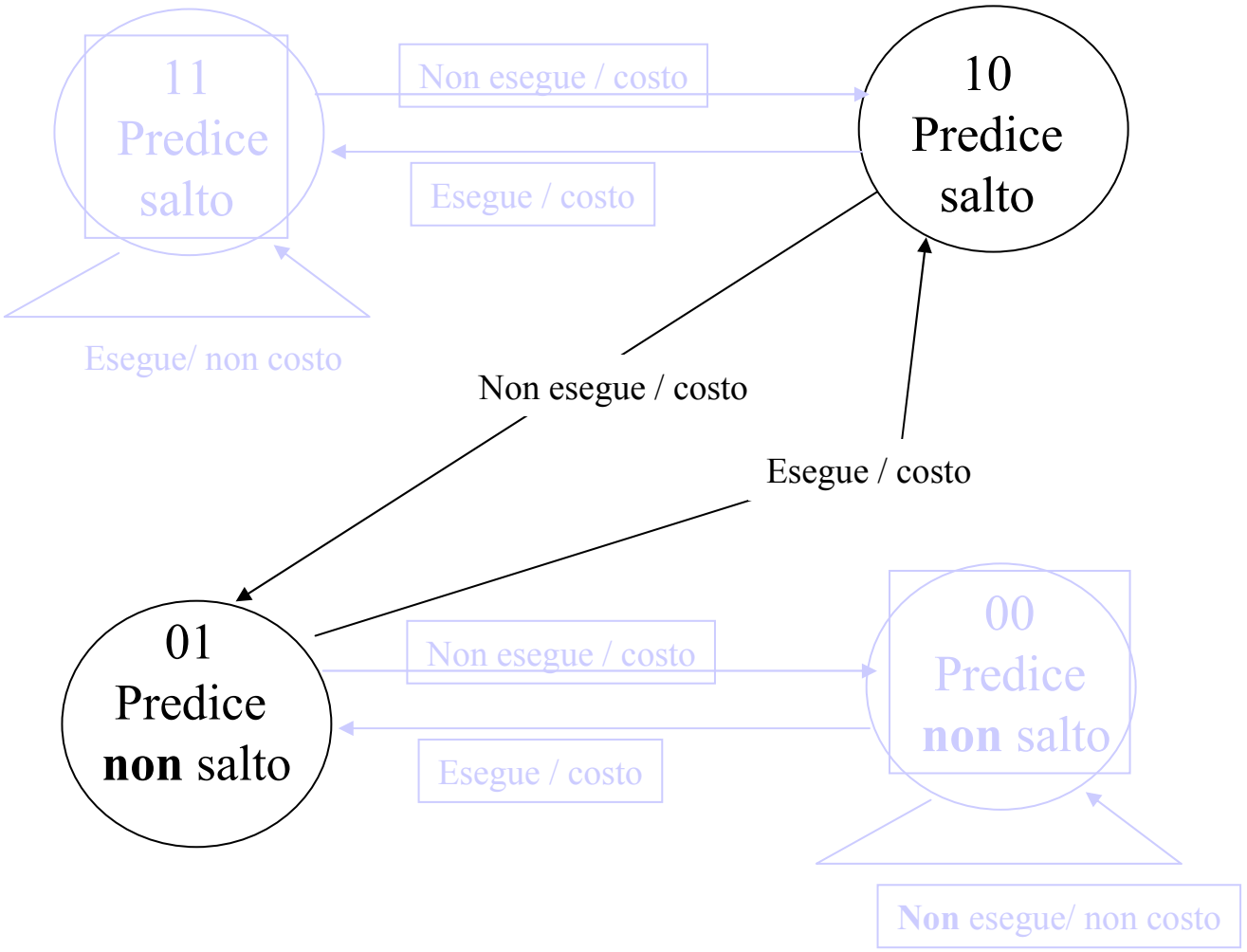
Cfr. ancora nel caso di un loop



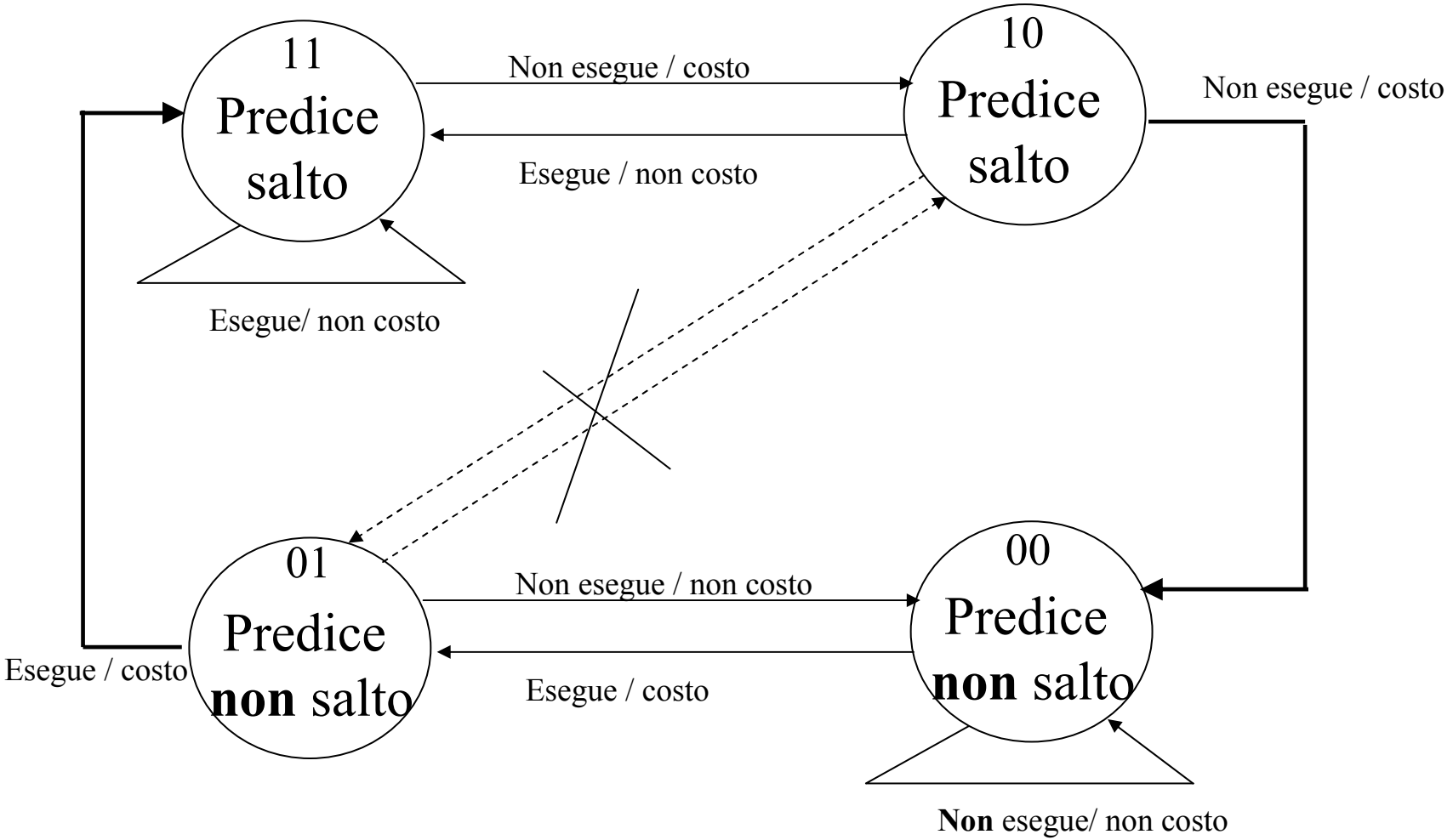
- Dopo qualche iterazione del ciclo, si stabilizza nello stato 11 (ultime 3 iterazioni: salto)
- All'uscita dal ciclo, stato 10 (=2), ma al successivo incontro si stabilizza di nuovo nello stato 11

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

NB: è ancora possibile un comportamento “oscillatorio” tra i due stati:

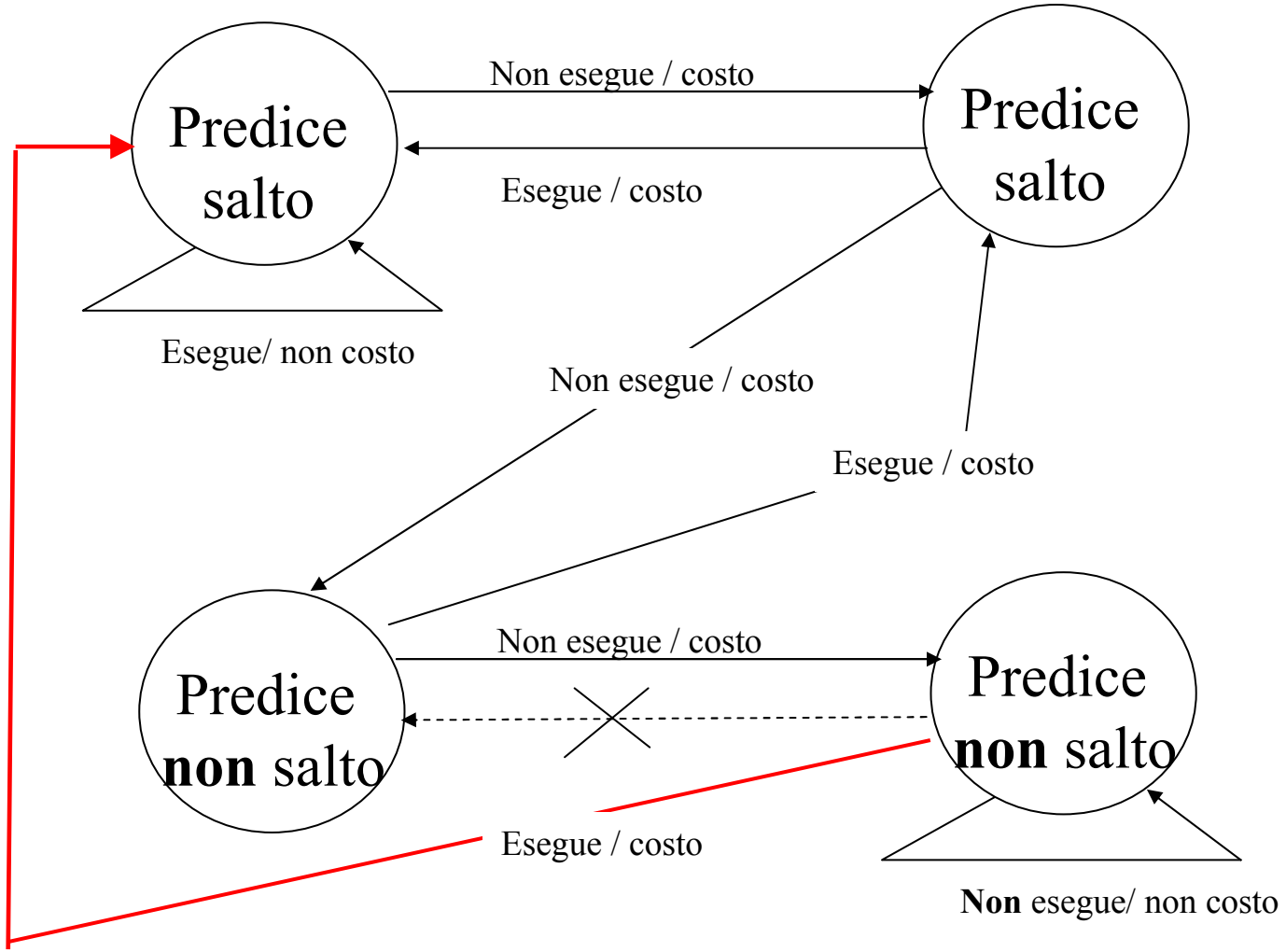


E’ possibile usare uno schema diverso



NB: la codifica degli stati corrisponde in questo caso agli ultimi due esiti (non così nello schema precedente!)

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)



NB: le scelte sono compiute in base a studi sperimentali.

Esempio: dati sperimentali [Hennessy & Patterson '93]

P(SS) = 0.97
P(NS) = 0.61
P(SN) = 0.54
P(NN) = 0.11

[probabilità che il salto sia efettuato]

NB: si cerca un compromesso tra accuratezza predizione e dimensioni richieste.

Esempio: dati sperimentali [Hennessy & Patterson '93]

P_{cp} = P (previsione corretta | riferimento a diramazione corretta) = 0.9
P_{cas} = P (previsione corretta | riferimento a diramazione errata) = 0.5
P = P (riferimento a diramazione corretta) = 0.9

⇒ Rappresentazione parziale dell'indirizzo:

Accuratezza = $P_{cp}*P+P_{cas}*(1-P)$ = 0.86

Rappresentazione totale dell'indirizzo (P=1):

Accuratezza = P_{cp} = 0.9

⇒ Miglioramento relativamente modesto vs. incremento memoria

INDIRIZZO DI DESTINAZIONE NEL SALTO PREDETTO

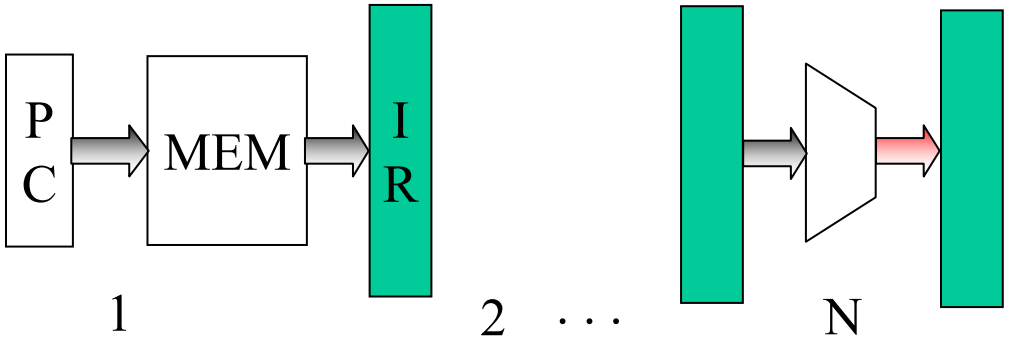
Previsione di salto nella fase di fetch dell'istruzione di diramazione

$\Rightarrow$ 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.

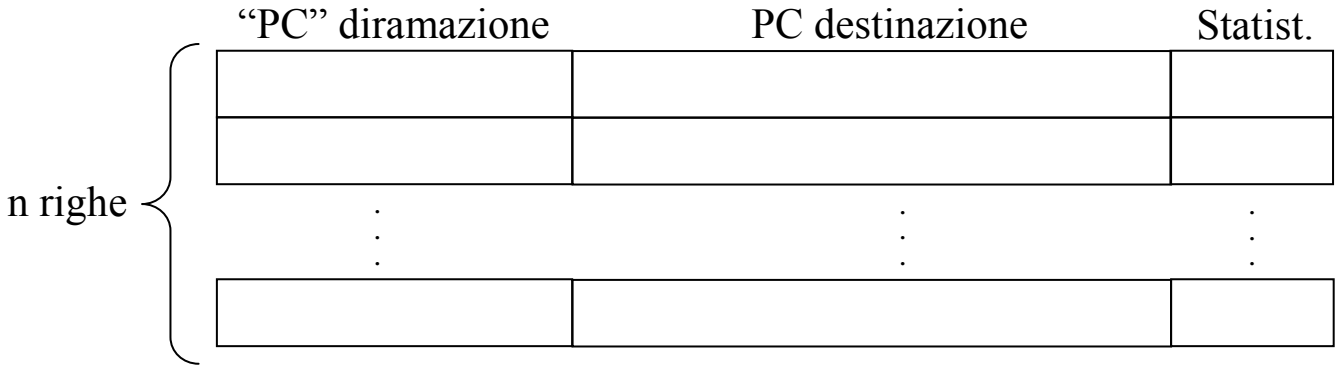
- Ricordare che non possiamo aumentare il cammino critico [no $\uparrow T_{clock}$], quindi se l'indirizzo è calcolato in uno stadio S [all'uscita dell'ALU] non può essere utilizzato per il prelievo in corso nello stadio di fetch! Quindi, calcolo in stadio nr. N $\Rightarrow$ N - 1 "bolle" (es: calcolo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



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

UNA SOLUZIONE: USARE UN CAMPO PER PREDIRE DESTINAZIONE

Tabella destinazione delle diramazioni (Branch Target Buffer – BTB)

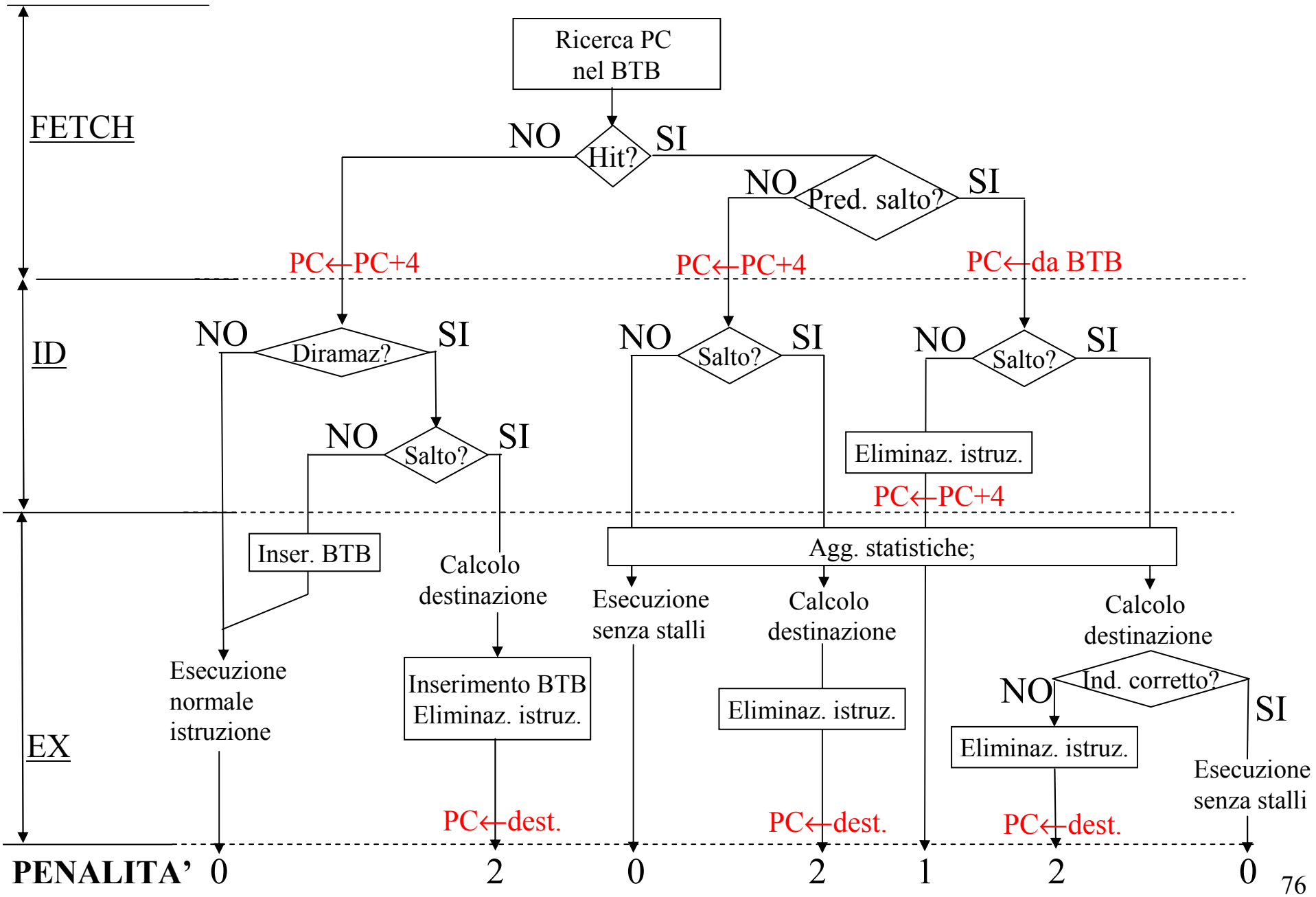


Idea di base: nella fase di fetch confronto PC con "PC" diramazione:

- 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 successiva

Può accadere che l'istruzione non sia in BTB ma comunque sia diramazione
Può accadere che l'indirizzo destinazione sia sbagliato
Ciò 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

Strategie di ottimizzazione del BTB

- Memorizzare in BTB solo diramazioni eseguite con successo (salto)
  [prevedendo che diramazione non eseguita non lo sia neppure la volta successiva, conviene risparmiarne l'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 essere eseguite con successo [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

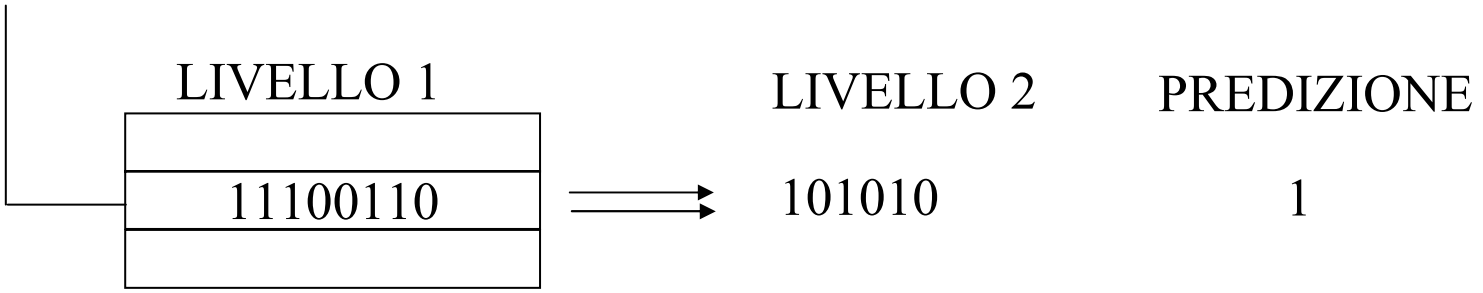
<u>Esempio</u>: Predittore adattivo a due livelli

- Primo livello: identifica informazioni storiche sulla diramazione
  [la situazione in cui ci si trova]
- Secondo livello: effettua la predizione vera e propria

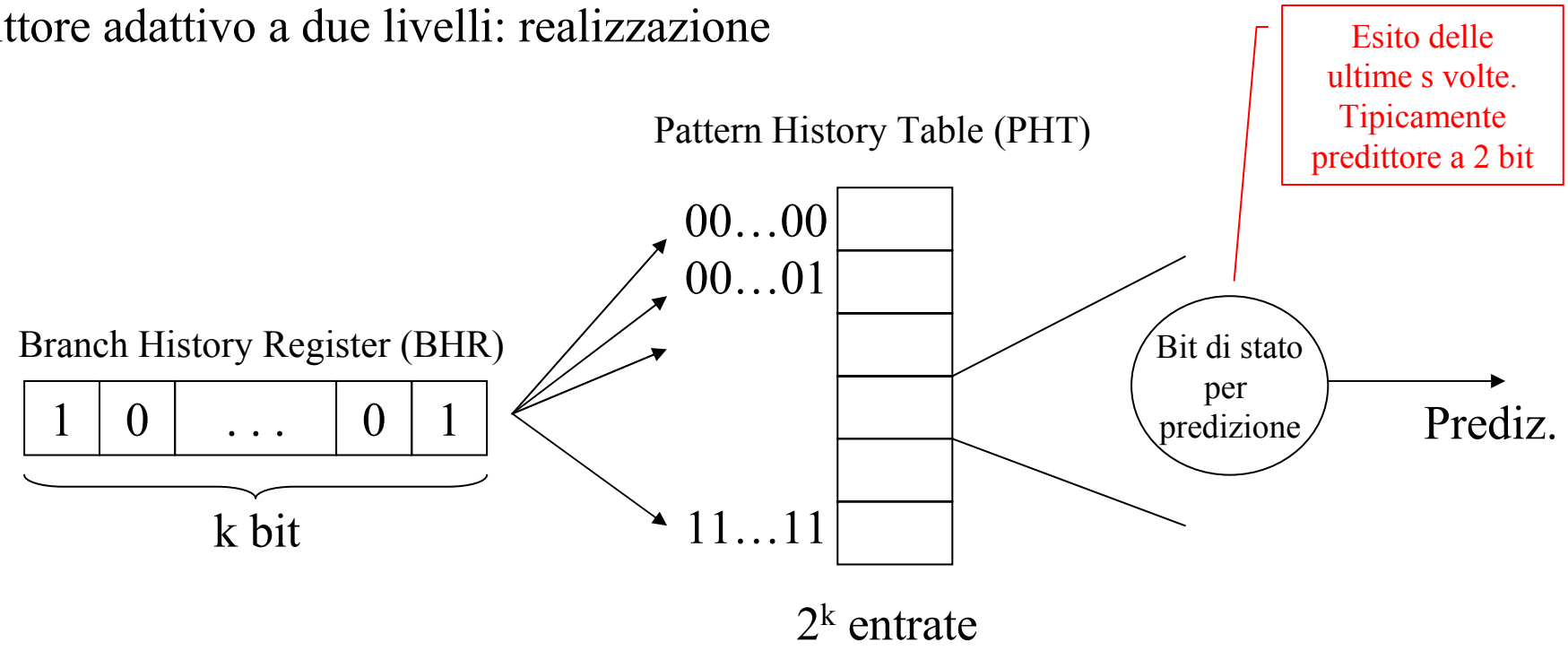
Tipicamente:

- il primo livello indica la successione degli esiti delle ultime *k* diramazioni
  Es. (con k=8) 11100110 [1=diramazione eseguita, 0 non eseguita]
- il secondo livello indica il comportamento che la diramazione ha avuto le ultime *s* volte che è stata preceduta dal corso storico indicato dal primo livello.

P.es. diramazione con storia 11100110 [k=8, s=6]



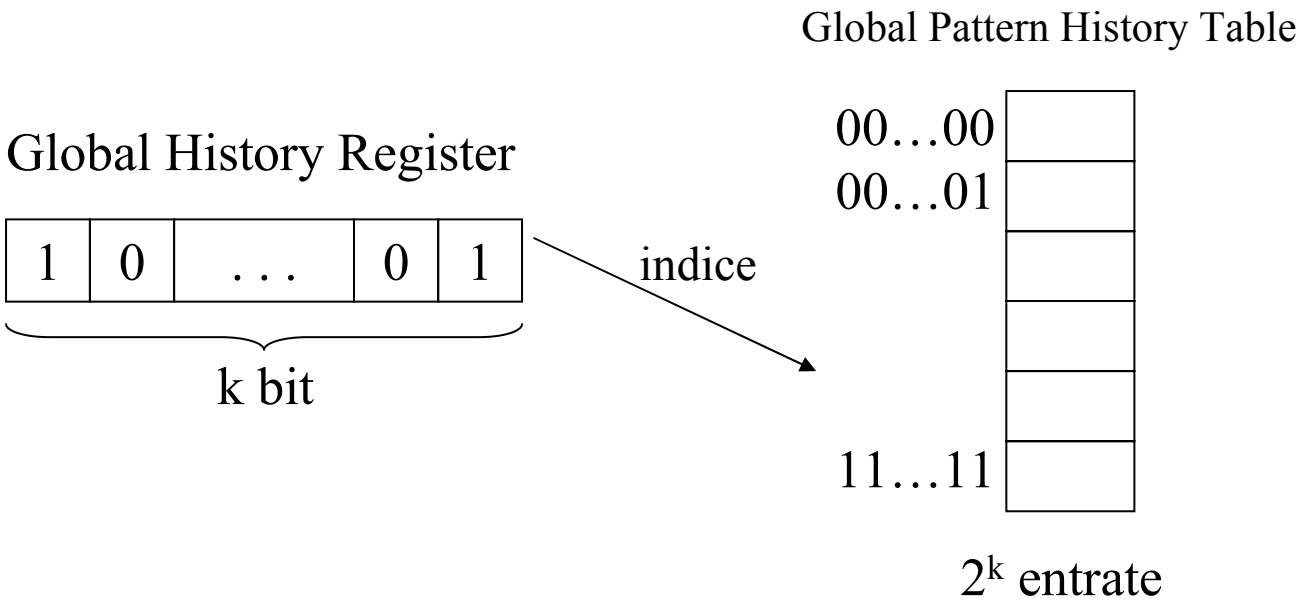
Predittore adattivo a due livelli: realizzazione



- BHR: registro a scorrimento (per l'aggiornamento trasla a sinistra)
- Al momento della predizione:
  - BHR, che tiene traccia di ultimi k esiti, utilizzato per indirizzare PHT
  - PHT, che tiene traccia della storia della diramazione dopo k esiti uguali, restituisce la predizione
  - dopo che la diramazione è stata risolta, bit per predizione aggiornati [nella stessa postazione] + BHR è aggiornato via scorrimento a sx e inserimento esito nella posizione meno significativa.

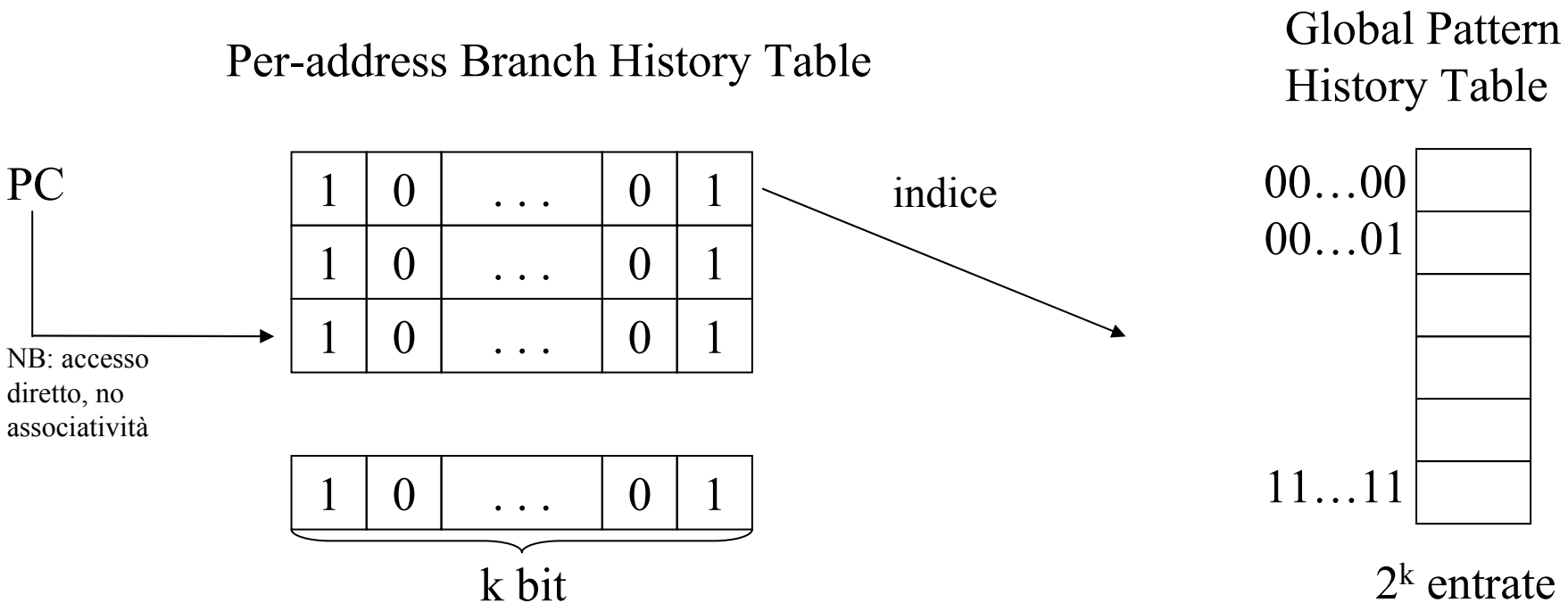
Le tre varianti possibili:

1) Global History Register e Global Pattern History Table



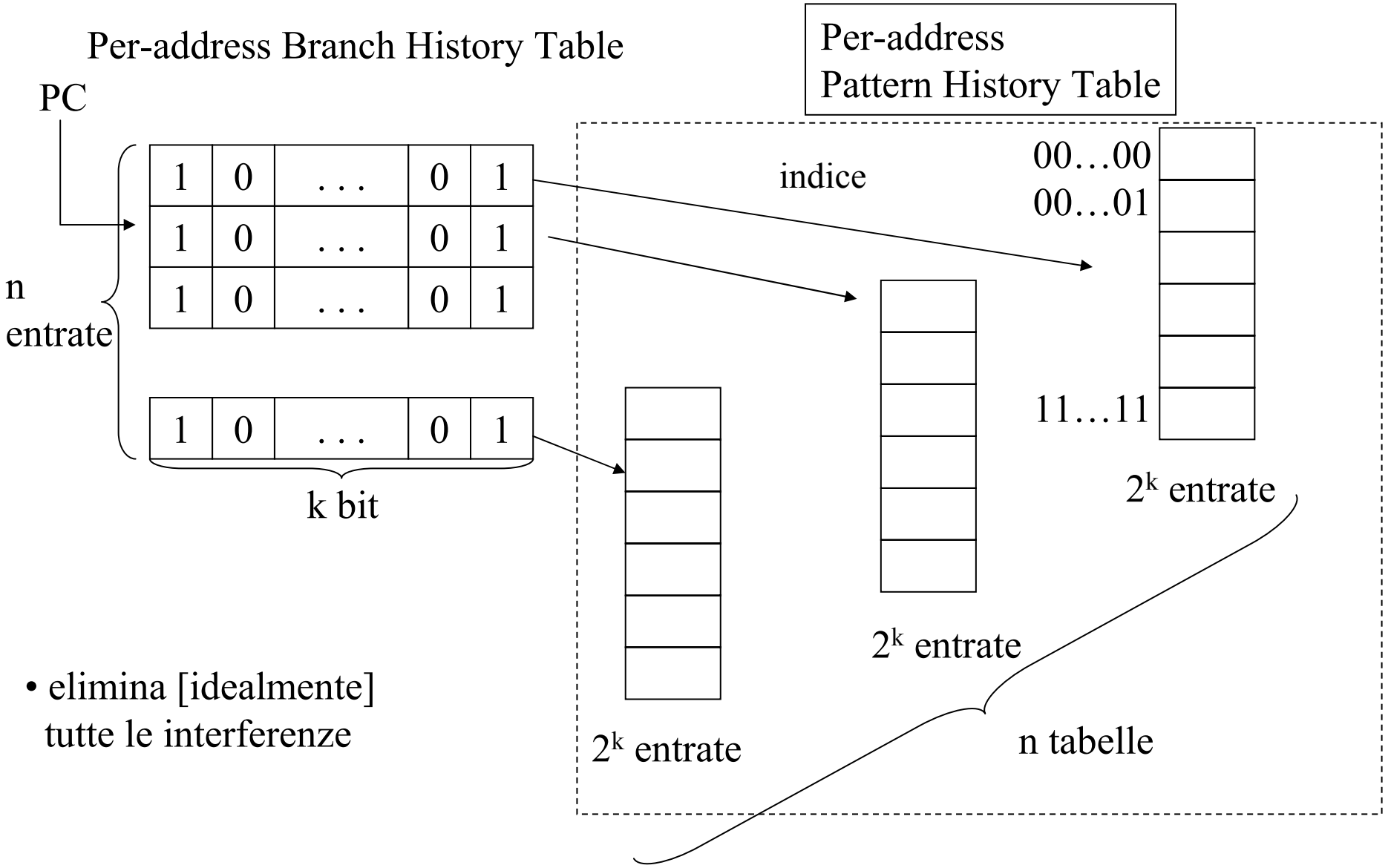
- Interferenze al primo livello: diramazioni diverse "mescolano" le storie nel GHR e quindi GHR non ha un valore molto attendibile
- Interferenze al secondo livello: diramazioni diverse che hanno la stessa storia modificano i valori della stessa entrata di GPHT…

2) Per-Address Branch History Table e Global Pattern History Table



- Elimina [idealmente] le interferenze al primo livello: ogni diramazione ha la sua entrata nel PBHT [no mescolamento delle storie]
- Permangono interferenze al secondo livello: diramazioni diverse con storia uguale accedono [e aggiornano] gli stessi bit di predizione [in GPHT]

3) Per-Address Branch History Table e Per-address Pattern History Table



• elimina [idealmente] tutte le interferenze

QUALE SCHEMA UTILIZZARE?

Come al solito, non esistono criteri assoluti, ma dati empirici e scelte di compromesso…

Es: dati sperimentali tratti da [Yeh & Patt '92]:

Predittore adattativo a 2 livelli: 97%
Contatore a saturazione (2 bit): 93%
Predittore "last time" (1 bit): 89%

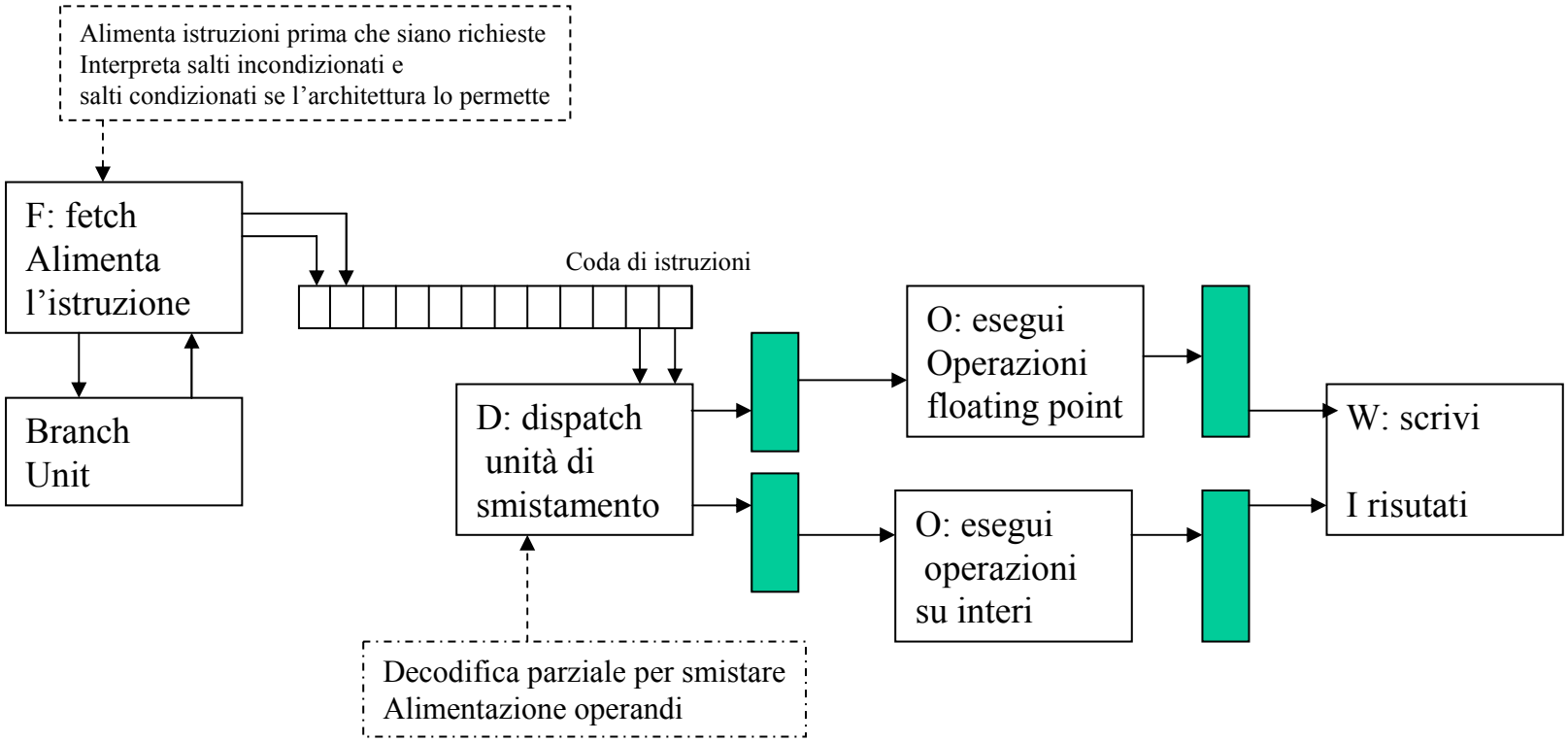
Per avere accuratezza del 97% (si usano solo alcuni bit meno signif. PC):

var.1: GHR a 18 bit $\Rightarrow$ GPHT con 2^{18} * 2 bit TOT. 2^{19} bit
var.2: PBHT a 512*12 bit $\Rightarrow$ GPHT con 2^{12} * 2 bit TOT. $< 2^{14}$
var.3: PBHT a 512*6 bit $\Rightarrow$ PPHT con 512 * 2^{6} * 2 bit TOT. $> 2^{16}$

Comunque tutto dipende dai benchmark di riferimento…

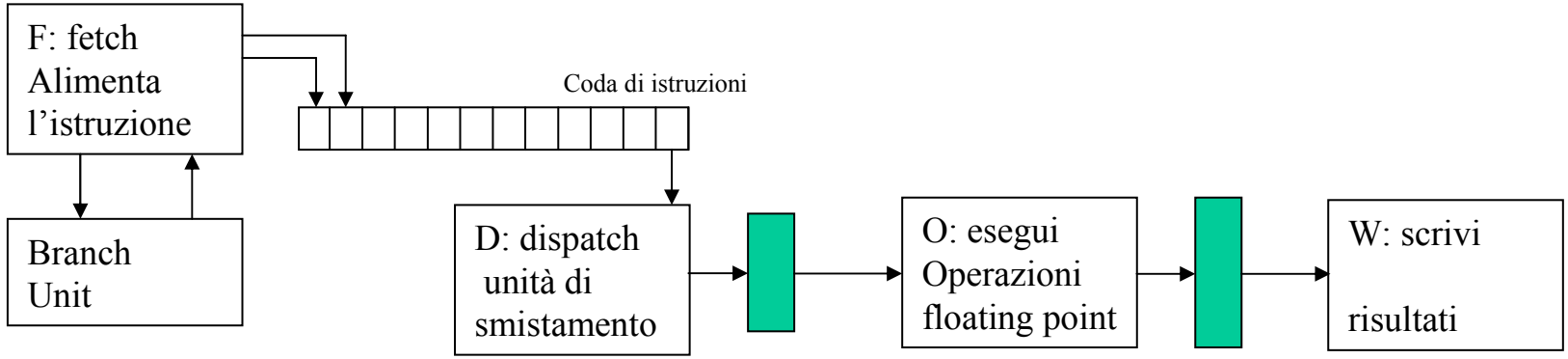
La coda delle istruzioni

Un modo per risolvere alcuni problemi di stallo è utilizzare una “coda delle istruzioni”



- In questo caso supponiamo più unità di esecuzione (unità di smistamento è in grado di instradare più istruzioni per ciclo di clock!)
- L’unità di fetch può prelevare dalla cache più istruzioni in un ciclo di clock; l’unità di smistamento preleva sempre le prime istruzioni nella coda e le invia alle unità di esecuzione appropriate

Consideriamo il caso di singola unità di esecuzione



- Per assicurare continuità del flusso di esecuzione istruzioni, unità di prelievo cerca di mantenere sempre piena la coda delle istruzioni: preleva più istruzioni in un ciclo di clock dalla cache e le pone in coda; in particolare,
  l'unità di prelievo può decodificare istruzioni di salto e inserire in coda le istruzioni da eseguire, nell'ordine corretto: vedremo che questo permette di risparmiare cicli.
- Unità di smistamento preleva istruzioni dalla testa della coda

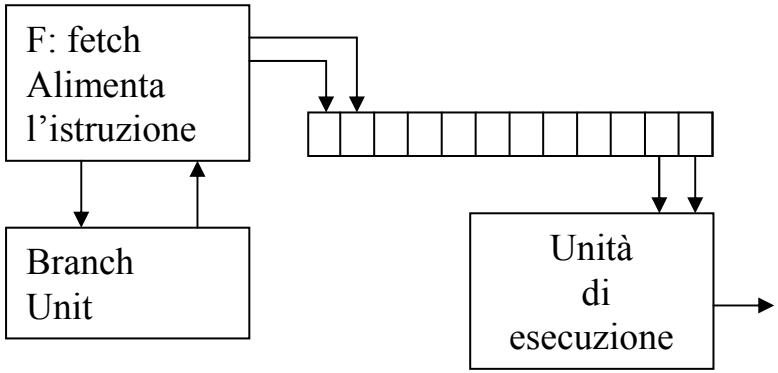
⇒ Miglioramento delle prestazioni nei seguenti casi:

- Fallimento di accesso alla cache [criticità strutturale]
- Salti incondizionati e condizionati (con condizione già valutata o prevista correttamente)

Primo caso: fallimento di accesso alla cache

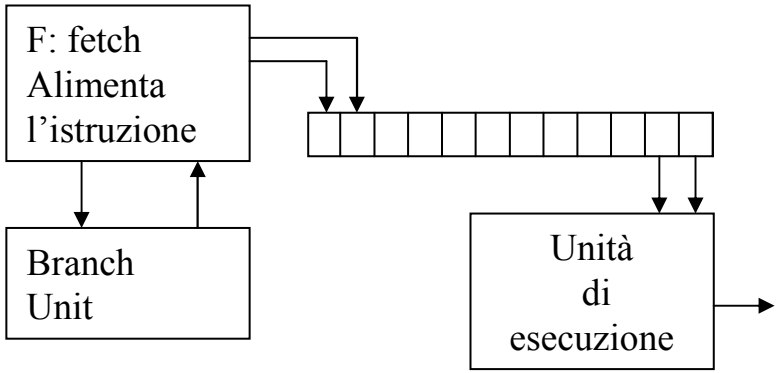
- Unità di prelievo preleva istruzioni indipendentemente da unità di smistamento: coda delle istruzioni mantenuta piena (per quanto possibile)
- Fallimento accesso cache ⇒
  - smistamento può continuare (l'unità di smistamento continua a prelevare istruzioni dalla coda finchè non è vuota)
  - intanto, il blocco della cache viene letto dalla memoria principale (o dalla cache secondaria)
- Se la frequenza di lettura dalla cache è sufficientemente elevata, la coda non si svuota completamente e non vi sono stalli dovuti a fallimento di accesso alla cache
  (altrimenti la coda si svuota e l'unità di smistamento è comunque in grado di mettere in stallo la pipeline – ovvero, di non inviare nuove istruzioni all'unità di esecuzione)

Secondo caso: salti (condizionati e incondizionati)



- L'unità di prelievo carica più istruzioni in un ciclo di clock C1 da una cache veloce
- Una volta caricate le istruzioni, l'unità di prelievo (nel ciclo di clock successivo C2) esamina ed "esegue" le istruzioni di salto presenti in coda. Supporremo sempre che il calcolo dell'indirizzo di destinazione (e la condizione di salto nel caso di un salto condizionato) si possano fare in un solo ciclo di clock (C2)
  - se si incontra un salto incondizionato, l'unità di prelievo calcola destinazione (C2) ed è in grado (nel ciclo successivo C3) di riempire la coda con nuove istruzioni a partire dalla destinazione.
  - se si incontra un salto condizionato, in C2 si calcola la destinazione; se è possibile valutare la condizione di salto in C2:
    - > se devo saltare: come nel salto incondizionato (carico destinazione in C3)
    - > se non devo saltare: si prosegue con istruzioni successive

Esempio: salti incondizionati



Ipotesi (che faremo quasi sempre):

- fetch [da cache veloce] **4** istruzioni alla volta
- nel ciclo di clock successivo a quello di fetch, l'unità di prelievo interpreta le istruzioni di salto incondizionato e calcola l'indirizzo di destinazione.
- se si incontra un salto incondizionato, l'unità di prelievo è in grado (nel ciclo successivo) di riempire la coda con nuove istruzioni a partire dalla destinazione.

Esempio (supponendo solo due fasi per le istruzioni – fetch + "esecuzione")

	C1	C2	C3	C4	C5	C6	C7	C8	Tempo cicli di clock
I1	F1	E1							
I2	F2		E2						
I3	F3	E3							
I4	F4								
I5			F_k	E_k					
I6			F_{k+1}		E_{k+1}				
I7			F_{k+2}			E_{k+2}			
I8			F_{k+3}				E_{k+3}		

I3 è un *salto incondizionato* "eseguito" dall'unità fetch.
Si noti che la penalità di salto è di –1 (!!!).

Salti condizionati

Per i salti condizionati, la decisione di saltare non può essere presa finchè la condizione di salto non è stata calcolata (dipende da istruzioni precedenti).

- Può essere che, quando l'unità di prelievo "incontra" l'istruzione di salto, la condizione di salto sia già stata valutata da un'altra unità: in tal caso l'unità di prelievo può lasciare la coda inalterata [se non si salta] oppure cominciare a prelevare (come nel caso del salto incondizionato) le istruzioni a partire dalla destinazione [se si salta]
- Però in generale la condizione di salto potrebbe essere ancora da valutare

⇒ Si usano le tecniche di predizione viste in precedenza (utilizzate però con la coda delle istruzioni)

Predizione dei salti condizionati con la coda delle istruzioni

Una volta prelevate le istruzioni, nel ciclo successivo al prelievo (C2) posso calcolare la destinazione dei salti; per i salti condizionati di cui non si è in grado di valutare la condizione di salto, uso la predizione:

- Predizione di salto effettuato ⇒ prelievo a partire da destinazione
- Predizione di salto non effettuato ⇒ "prelievo" da istruzione successiva
  [o mantengo istruzioni in coda]

Quando è possibile, l'unità di prelievo valuta la condizione di salto:

Predizione corretta ⇒ si ottengono gli stessi benefici del salto incondizionato

Predizione errata ⇒ scartare le istruzioni prima che modifichino registri

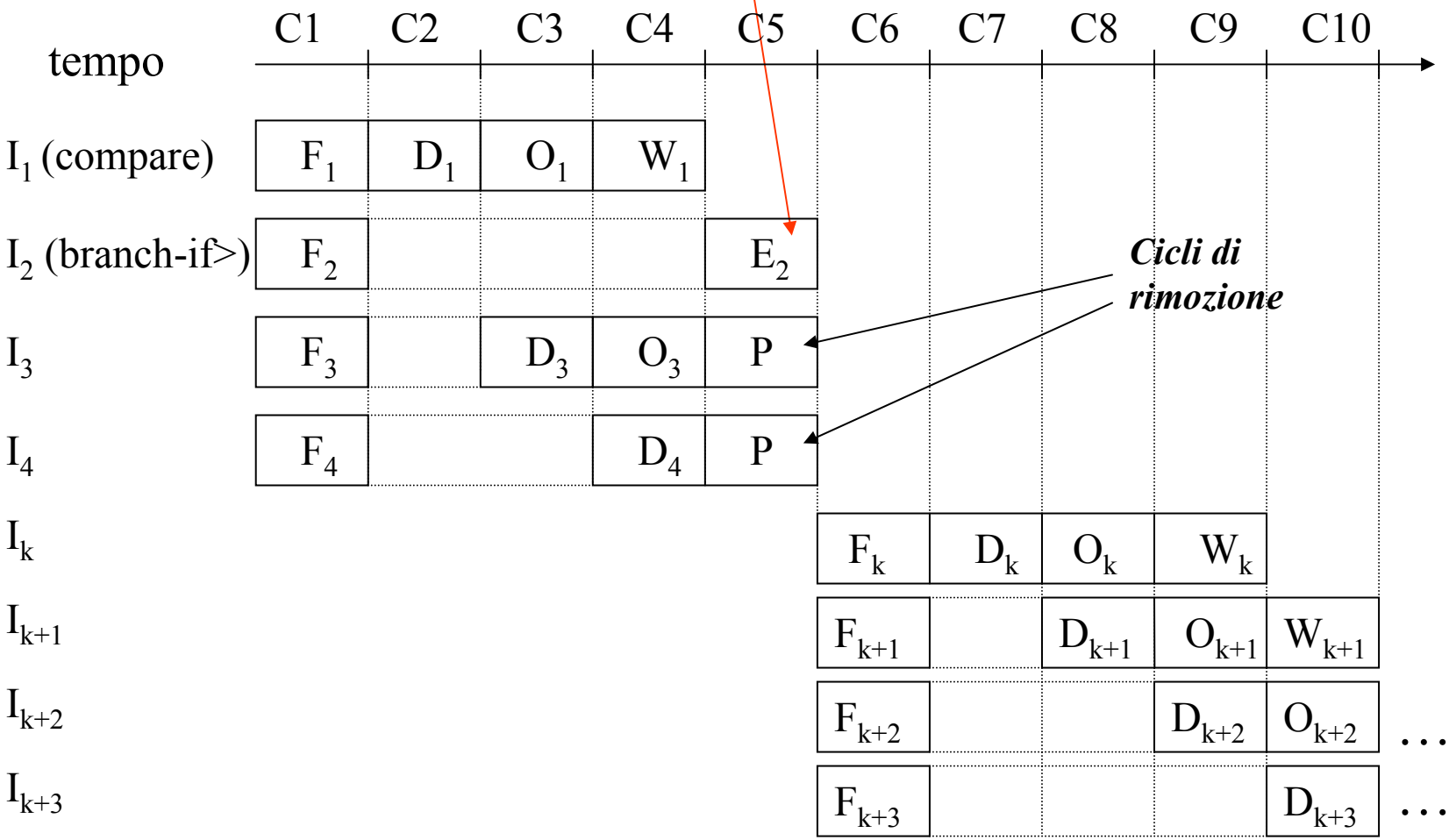
I passi di scrittura delle istruzioni eseguite su base speculativa (ovvero, le istruzioni "previste") non devono avvenire prima della valutazione della condizione di salto
[eventualmente, i passi di scrittura vengono ritardati]

Esempio di predizione errata (pipeline a 4 stadi, prelievo di 4 istruzioni per ciclo)

Supponiamo semplice previsione di salto non effettuato

Assunzione: calcolo condizione di branch e calcolo nuovo indirizzo in un ciclo. Però Branch Unit (parte dell'unità di prelievo) deve attendere il risultato del Compare
[NB: potrei anticipare E2 a C4 con la propagazione da I1]

Eseguite perché si predice salto non effettuato



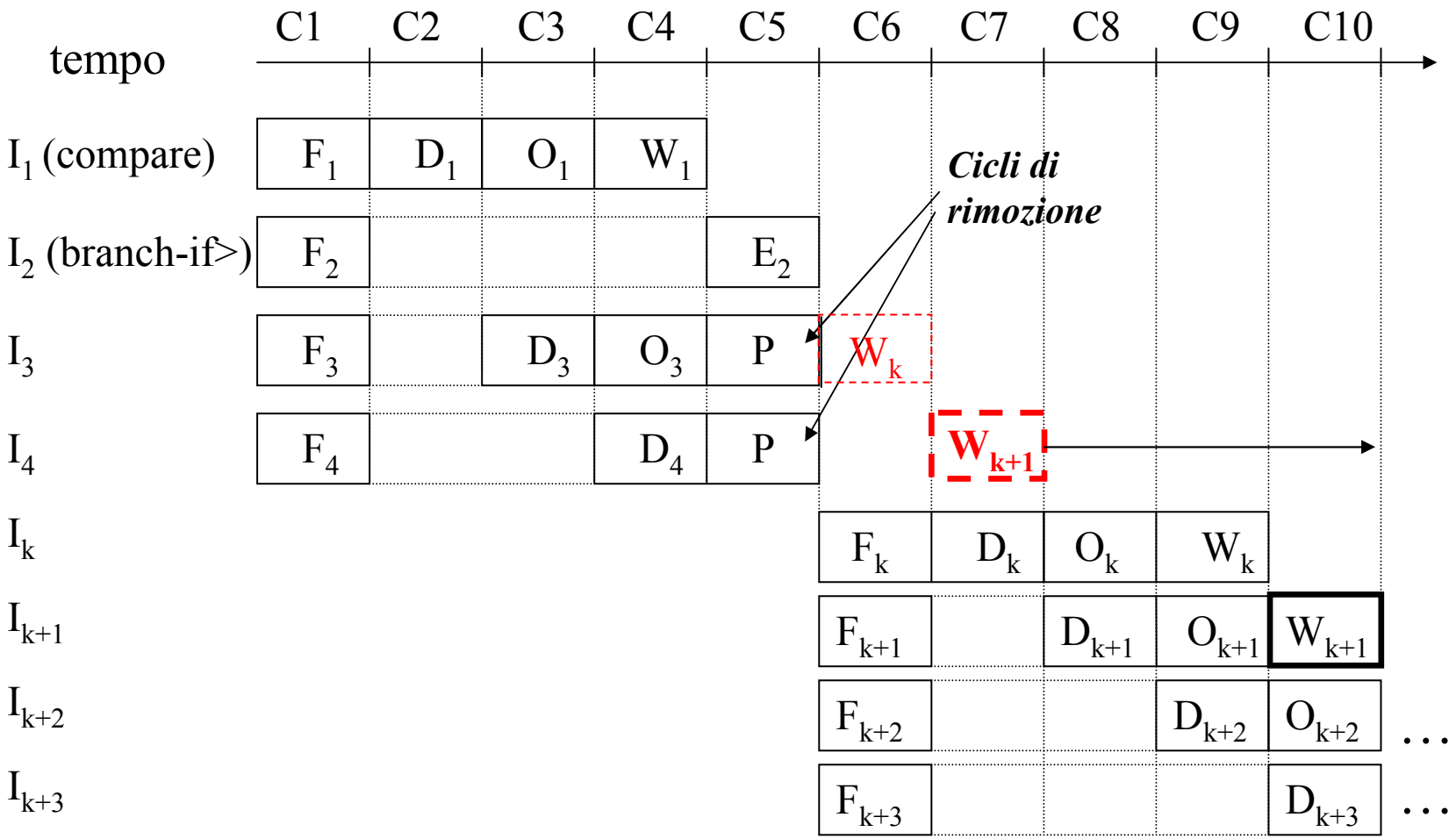
Chiarimenti sull'esempio (1)

- L'unità di prelievo predice che il salto non verrà effettuato, quindi le istruzioni I_3 e I_4 vengono fatte proseguire
- La decisione finale sul salto viene presa dall'unità di prelievo (stadio indicato con E_2) dopo che i flag dei codici di condizione sono stati aggiornati al termine dell'istruzione I_1 (e quindi al termine di W_1)
- Quindi, dopo il passo W_1, l'unità di prelievo si accorge che la predizione era errata e che le istruzioni I_3 e I_4 vanno rimosse (cicli 'P' in figura)
- Quattro nuove istruzioni, da I_k a I_{k+3} vengono prelevate a partire dall'indirizzo calcolato al passo E_2

Chiarimenti sull'esempio (2)

- Con pipeline a 4 stadi, in caso di predizione errata si ha una penalità di salto pari a 3 cicli
- Infatti, osservando la figura precedente si vede che la quarta istruzione (in questo caso I_{k+1}) terminerebbe al ciclo 7 invece che al ciclo 10 nel caso normale (senza salti e stalli di altro tipo)
- Tuttavia, se si attua anche l'anticipo degli operandi (propagazione dei dati) è possibile ridurre la penalità a 2 cicli… analizzare per esercizio come questo avviene
- Si noti infine che, se il risultato della condizione di salto fosse previsto correttamente, la penalità di salto sarebbe –1 (come già analizzato in precedenza nel caso di pipeline a 2 stadi)

Chiarimento: penalità è di tre cicli



Prestazioni per processore con pipeline

Si ricorda che, per un processore multiciclo, era:

$$T_{esecuzione} = \#istruzioni * CPI * T_{clock}$$

Con la pipeline, questi parametri hanno un significato diverso: viene completata (a parte il caso di stalli) una istruzione per ciclo di clock!!!

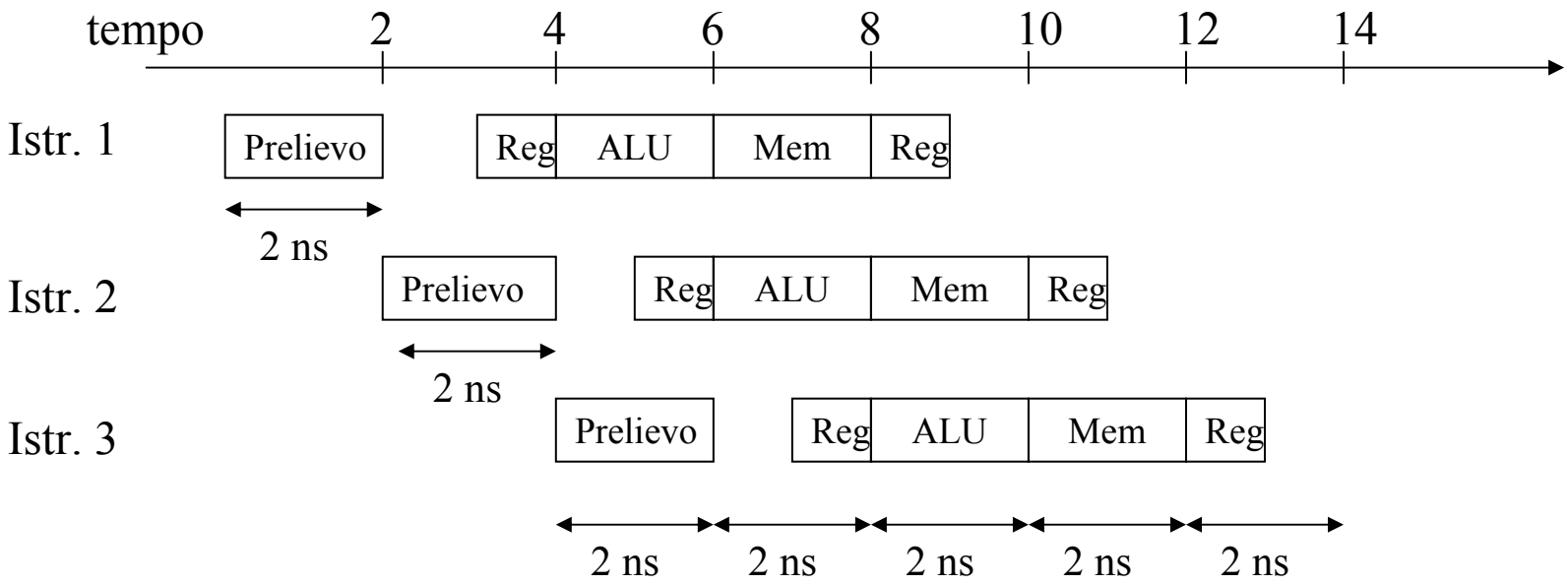
Ha senso considerare il
Tempo medio di esecuzione = tempo che intercorre tra il completamento di due istruzioni successive

Idealmente, se non ci sono stalli, 1 ciclo di clock tra un'istruzione e la successiva, ovvero 1 istruzione ogni T_{clock}

- Supponiamo di avere una pipeline a 5 stadi (prelievo, decodifica/lettura registri, esecuzione, accesso in memoria, scrittura registri)
- Supponiamo che la durata delle unità funzionali sia la seguente:
  - Prelievo, esecuzione con ALU, accesso in memoria = 2 ns
  - Decodifica/lettura registri, scrittura registri = 1 ns
- Ogni stadio di clock richiede un ciclo di clock: il periodo di clock deve essere abbastanza lungo da contenere l'operazione più lenta
- Quindi il ciclo di clock deve essere posto a 2 ns
- Il tempo per eseguire un'istruzione è sempre 10 ns (5 stadi $\times$ 2 ns)
- Ad ogni ciclo, cioè ogni 2 ns, viene completata un'istruzione:

  il *tempo medio di esecuzione* è il tempo che intercorre fra il completamento di due istruzioni successive (in questo caso 2 ns)

Esecuzione in pipeline di 3 istruzioni



- Il tempo totale di esecuzione di 3 istruzioni è 14 ns, a causa del tempo di startup della pipeline
- Tempo medio di esecuzione = 14/3 ≈ 4,67 ns ⇒ tempo non significativo, non siamo nella situazione a regime !!!

Tempo di esecuzione per processore con pipeline a regime

- Il tempo medio di esecuzione prima calcolato non è significativo
- Occorre considerare la situazione a regime
- Ad esempio, supponiamo di aggiungere 1000 istruzioni alle 3 di cui sopra
- Ogni istruzione in più fa aumentare il tempo di esecuzione di 2 ns
- Il tempo totale di esecuzione di 1003 istruzioni è quindi:

  $14 \text{ ns} + 1000 \times 2 \text{ ns} = 2014 \text{ ns}$
- Da cui, il tempo medio di esecuzione diventa = $2014/1003 \approx 2$ ns
- Naturalmente abbiamo considerato il caso ideale: senza alcuna criticità

Tempo medio di esecuzione, throughput e CPI

- Tempo medio di esecuzione pari a 2 ns significa che, a regime, viene completata un'istruzione ogni 2 ns (cioè ad ogni ciclo di clock)
- Cioè, grazie alla pipeline, aumenta il ***throughput delle istruzioni*** (ovvero il rapporto ***numero istruzioni/ciclo di clock***) e non il tempo di esecuzione di un'istruzione (che dipende anche dal numero di stadi della pipeline)

PIPELINE IDEALE:

Si considera CPI = 1, in quanto ad ogni ciclo di clock viene completata un' istruzione

PIPELINE IN PRESENZA DI STALLI:

CPI = CPI ideale + Cicli di stallo per istruzione

[VEDI ESERCIZI…]