

Calcolatori Elettronici B

a.a. 2005/2006

**Pipeline e prestazioni:
Esercizi**

Massimiliano Giacomini

Esercizio – confronto prestazioni pipeline vs. multiciclo

[già svolto nei lucidi]

- Si consideri la seguente combinazione di istruzioni eseguite
 - 22% load, 11% store, 49% formato-R, 16% salti condizionati, 2% salti incondizionati
- Si consideri la pipeline MIPS a 5 stadi con i seguenti tempi di esecuzione:
 - Prelievo, esecuzione con ALU, accesso in memoria = 4 ns
 - Decodifica/lettura registri, scrittura registri = 2 ns

Determinare il tempo medio di esecuzione per istruzione per un processore con controllo multiciclo e per un processore con pipeline, considerando per la pipeline il **caso ideale**: situazione *a regime* e *nessuna criticità*. Confrontare le prestazioni delle sue soluzioni.

Soluzione

$$T_{\text{clock}} = 4 \text{ ns}$$

(corrisponde al tempo di esecuzione dell'unità funzionale più lenta)

Tempo medio di CPU nel caso di controllo multi-ciclo

- Numero di cicli per ciascuna classe di istruzioni
 - Load = 5
 - Store = 4
 - Formato-R = 4
 - Salti cond. = 3
 - Salti incond. = 3
- $\text{CPI} = 0,22 \times 5 + 0,11 \times 4 + 0,49 \times 4 + 0,16 \times 3 + 0,02 \times 3 = 4,04$
- Tempo medio per istruzione = $\text{CPI} \times \text{Ciclo di clock}$
- Tempo medio per istruzione = $4,04 \times 4 \text{ ns} = 16,16 \text{ ns}$

Pipeline ideale e confronto prestazioni

- Per una pipeline ideale si considera $CPI = 1$
- Un ciclo di clock è pari a 4 ns
- Il tempo medio di esecuzione è pari a 1×4 ns
- Confrontiamo ora i tempi medi nei due schemi considerati

$$\text{Speedup} = \frac{\text{Tempo}_{\text{multiciclo}}}{\text{Tempo}_{\text{pipeline}}} = \frac{16,16}{4} = 4,04$$

Le prestazioni migliorano di 4,04 volte nel caso di controllo pipeline

Influenza delle criticità sulle prestazioni

- Tipi di criticità:
 - **criticità strutturale**: ad esempio, viene usata un'unica memoria per i dati e per le istruzioni; oppure caso di fallimento in cache
 - **criticità sui dati**: quando ci sono dipendenze di dati fra istruzioni consecutive
 - **criticità sul controllo**: dovuta ai salti
- Vediamo come questi tipi di criticità influenzano le prestazioni di un processore con pipeline
- Vedremo vari esercizi in cui le precedenti criticità (nell'ordine) influenzano le prestazioni

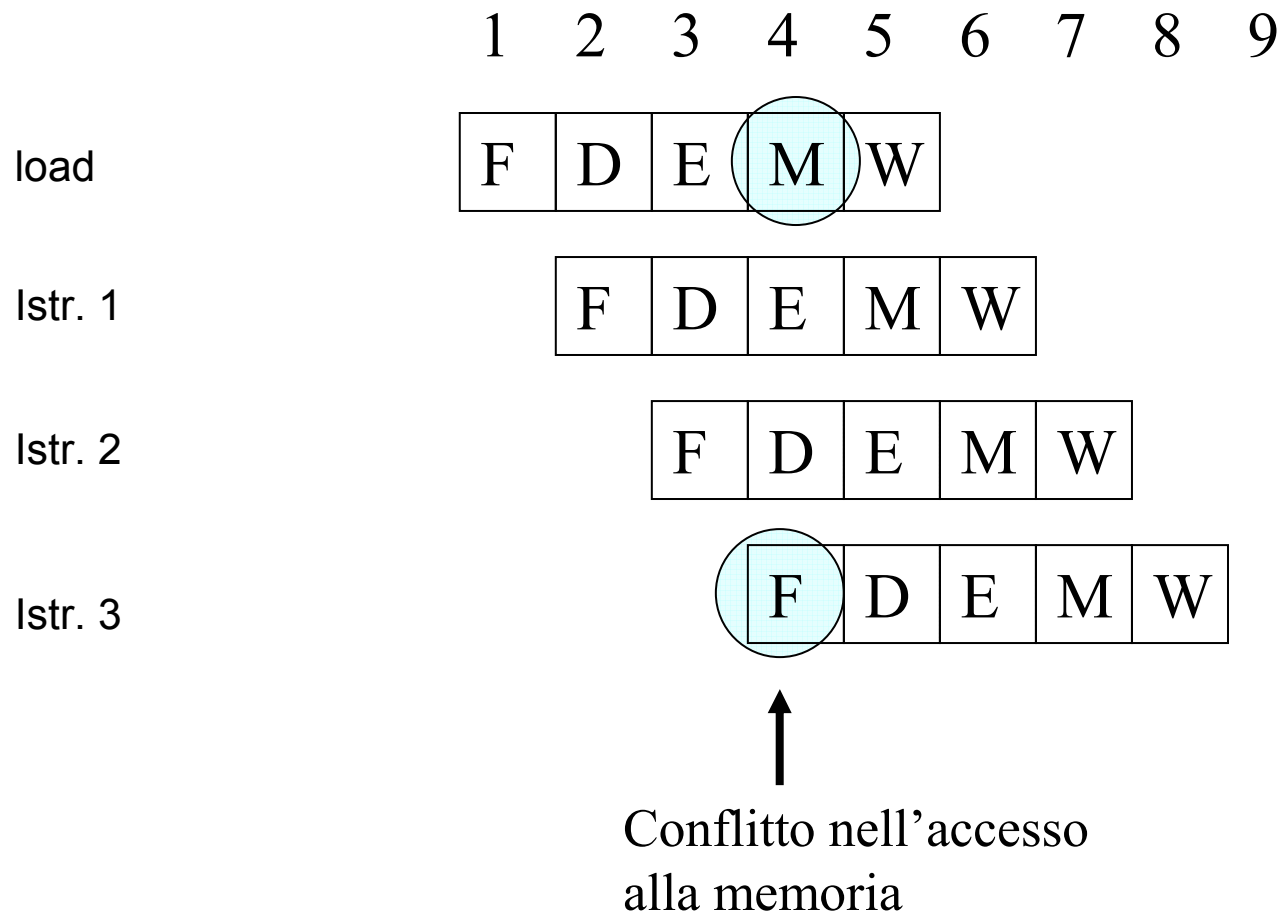
PRESTAZIONI CON PIPELINE:

CRITICITA' STRUTTURALI

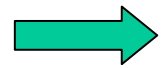
Esercizio – una sola memoria cache per dati e istruzioni

[già svolto nei lucidi; cfr. Es. 4, Tema d'esame 22 settembre '05]

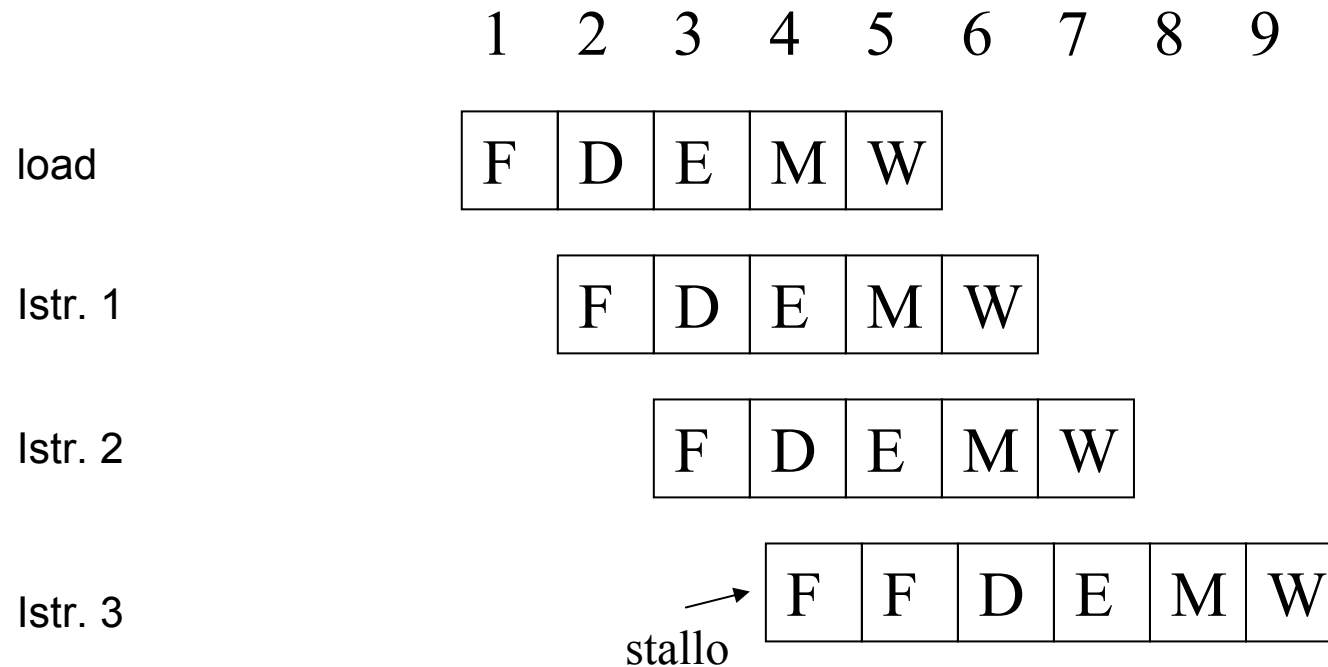
- Si consideri un processore dotato di pipeline (5 stadi) che disponga di una sola memoria cache per i dati e per le istruzioni:
 - 1) Si proponga una soluzione al problema basata sullo stallo della pipeline, descrivendo mediante diagramma temporale ciò che avviene in caso di conflitto sulla memoria.
 - 2) Si supponga che il 40% delle istruzioni eseguite facciano riferimento ai 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

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

$$\text{CPI} = \text{CPI ideale} + \text{Cicli di stallo per istruzione}$$

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

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

$$\text{Throughput} = 1/\text{CPI} = 1/1.4 = 0.71 \text{ istruzioni/ciclo}$$

Esercizio – miss di cache

Si consideri un sistema dotato solamente di **cache primaria** (senza cache secondaria) distinta per i dati e le istruzioni.

Per la cache si suppone:

- una penalità di fallimento di 10 cicli di clock
- una percentuale di successo del 95% per le istruzioni e del 90% per i dati

Si suppone inoltre che il 40% delle istruzioni faccia accesso a dati in memoria.

Si supponga che il carico sia tale che il CPI medio nel caso multiciclo risulti in assenza di miss di cache (come nel precedente esercizio) 4,04

Si calcolino il CPI effettivo ed il throughput per un processore multiciclo e per un processore con pipeline (in tal caso trascurando altre cause di stallo).
Si confrontino le prestazioni delle due soluzioni progettuali.

Consideriamo il processore con pipeline

- Se avviene un fallimento di accesso alla cache si ha un ritardo dell'istruzione in cui si è verificato il fallimento: si ha un incremento dei cicli di clock pari alla penalità di fallimento di accesso alla cache (10 cicli)
- Dobbiamo considerare l'incremento medio di CPI dovuto ai fallimenti di accesso alla cache, chiamiamo δ_{miss} questo incremento
- Supponendo 95% percentuale di successo per le istruzioni e 90% percentuale di successo per i dati
- Supponendo che il 40% delle istruzioni faccia accesso a dati in memoria:

$$\delta_{\text{miss}} = \underbrace{0,05 \times 10}_{\text{Cicli di stallo per le istruzioni}} + \underbrace{0,4 \times 0,1 \times 10}_{\text{Cicli di stallo per i dati}} = 0,9 \text{ cicli}$$

$\Rightarrow$ CPI effettivo (in assenza di altre cause di stallo):

$$\text{CPI} = 1 + \delta_{\text{miss}} = 1 + 0,9 = 1,9$$

$\Rightarrow \text{Throughput}_{\text{pipeline}} = 1/\text{CPI} = 1/1,9 = 0,53 \text{ istruzioni/ciclo}$

Consideriamo il processore multiciclo:

In assenza di miss di cache avrei $CPI = 4,04$

Ma per le istruzioni che danno miss (fetch è accesso ai dati) ho una penalità pari a quella del caso con pipeline:

$$\delta_{\text{miss}} = 0,05 \times 10 + 0,4 \times 0,1 \times 10 = 0,9 \text{ cicli}$$

$$\Rightarrow CPI_{\text{multiciclo}} = 4,04 + 0,9 = 4,94$$

$$\Rightarrow \text{Throughput}_{\text{multiciclo}} = 1/CPI = 1/4,94 = 0,20 \text{ istruzioni/ciclo}$$

Quindi l'incremento delle prestazioni nel caso con pipeline è

$$0,53/0,20 = 2,65 \quad [\text{Posso confrontare i throughput o, ugualmente, CPI – o, ugualmente, i tempi di esecuzione: il risultato non cambia!}]$$

Esercizio – due livelli di cache

[cfr. Es. 3, Tema d'esame 6 aprile 06]

Supponiamo di avere sia **cache primaria** che **cache secondaria**

- Per la cache primaria, come prima:
 - 95% percentuale di successo per istruzioni, 90% per i dati
 - Si supponga inoltre che per trasferire blocchi dalla cache secondaria alla cache primaria occorranzo 4 cicli.
 - Penalità di fallimento: 10 cicli.
- Per la cache secondaria:
 - 94% la percentuale di successo per le istruzioni, 92% per i dati.
- Come prima, supponiamo che il 40% delle istruzioni faccia accesso ai dati in memoria

Calcolare CPI e Throughput per un processore con pipeline (trascurando altri stalli)

Soluzione

$$\begin{aligned} \delta_{\text{miss}} &= \overbrace{0,05 \times (0,94 \times 4 + 0,06 \times 10)}^{\text{Cicli di stallo per le istruzioni}} + \overbrace{0,4 \times 0,1 \times (0,92 \times 4 + 0,08 \times 10)}^{\text{Cicli di stallo per i dati}} = \\ &= 0,4 \text{ cicli} \end{aligned}$$

Da cui, **CPI** = 1 + 0,4 = 1,4

Throughput = 1/1,4 = 0,71 istruzioni/ciclo

Nota all'esercizio precedente:

un fallimento alla cache secondaria (che segue un fallimento alla cache primaria) ha una penalità di fallimento pari a 10 cicli, ovvero quella della cache primaria: è il tempo necessario per portare il blocco mancante nella memoria cache primaria! [la cache secondaria viene caricata indipendentemente]

Per esercizio:

svolgere il calcolo per un processore multiciclo e confrontare le prestazioni in questo secondo caso (cache primaria + cache secondaria)

PRESTAZIONI CON PIPELINE:

**CRITICITA' SUI DATI E
PROPAGAZIONE**

Esercizio (di introduzione)

- Si supponga di avere il seguente codice assembler
 add \$s0, \$t0, \$t1
 sub \$t2, \$s0, \$t3

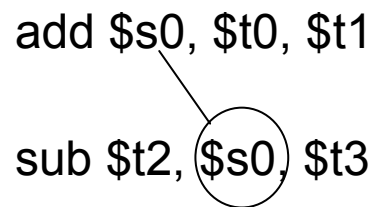
Si consideri la pipeline a 5 stadi del MIPS:

- si individuino le eventuali dipendenze tra i dati
- supponendo di non disporre dell'unità di propagazione, tracciare il diagramma temporale e individuare il numero di stalli necessari
- supponendo di disporre dell'unità di propagazione verso lo stadio E, tracciare il diagramma temporale

Dipendenze

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3



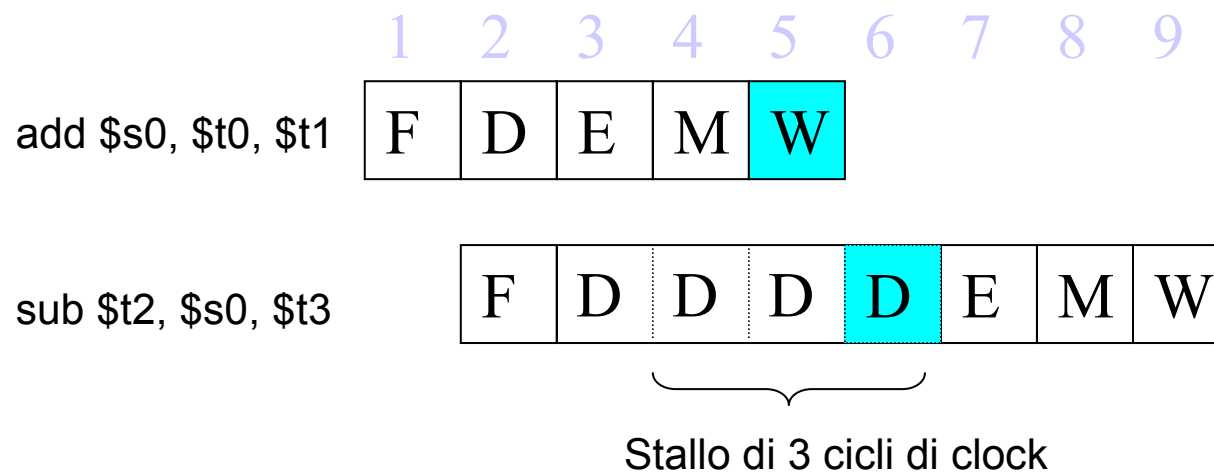
Abbiamo una sola dipendenza su s0

Senza propagazione:

Le istruzioni indipendenti procedono normalmente!

Per ogni dipendenza:

- individuare in quale stadio l'istruzione indipendente scrive il dato
- individuare in quale stadio l'istruzione dipendente legge il dato
- l'istruzione dipendente deve essere posta in stallo in modo che legga il dato dopo che è stato scritto



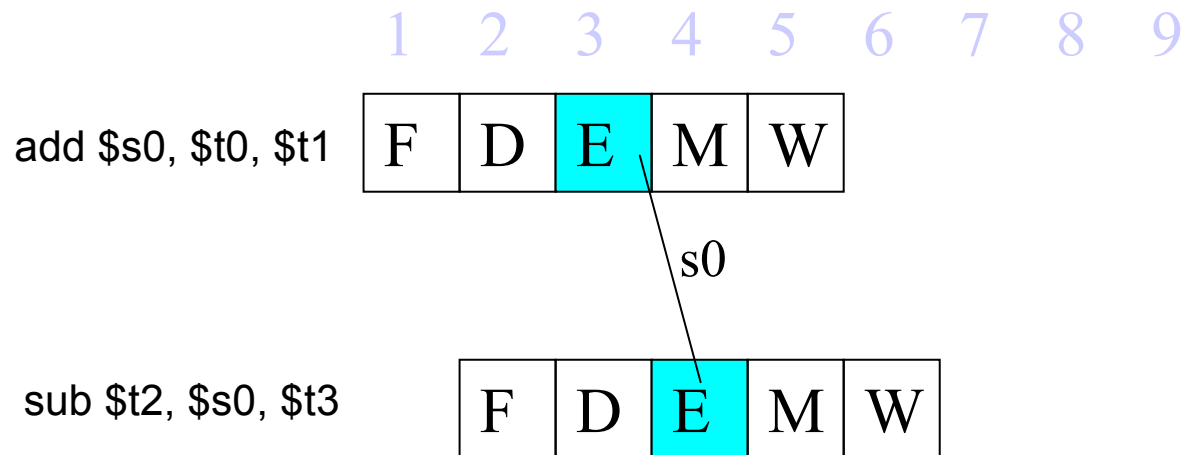
- L'operazione `sub`, per deve aspettare che sia disponibile il nuovo valore di `$s0`.
- Con una pipeline a 5 stadi (prelievo, decodifica, esecuzione, accesso in memoria, scrittura risultato) la `add` scrive il risultato nel quinto stadio e quindi si creano **tre bolle** nella pipeline (= stallo di 3 cicli di clock).

Con propagazione:

Le istruzioni indipendenti procedono normalmente!

Per ogni dipendenza:

- individuare in quale stadio l'istruzione indipendente produce il dato
- individuare in quale stadio l'istruzione dipendente può propagare il dato (di solito, dove lo utilizza)
- verificare se il dato è disponibile (prodotto in cicli di clock precedenti) quando l'istruzione dipendente arriva nello stadio in cui è disponibile una unità di propagazione:
nel primo caso viene propagato, nel secondo servono cicli di stallo e propagazione



Nessuno stallo in questo caso!

Esercizio – criticità carica-e-usa

- Si consideri il codice assembler:

lw \$s0, 10(\$t1)

sub \$t2, \$s0, \$t3

Supponendo di disporre dell'unità di propagazione verso lo stadio E, tracciare il diagramma temporale ed individuare il numero di stalli necessari.

- Supponendo che il 30% delle istruzioni eseguite siano delle load e che, nel 50% dei casi, a un'istruzione di load segua un'istruzione che dipende dal risultato della load:

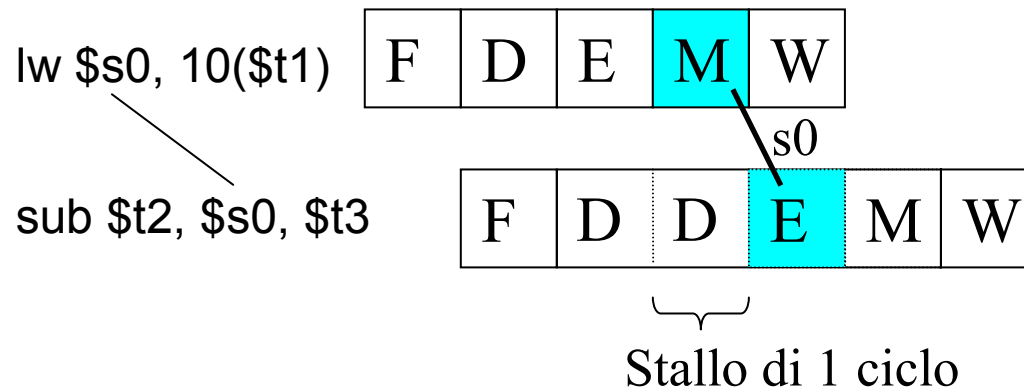
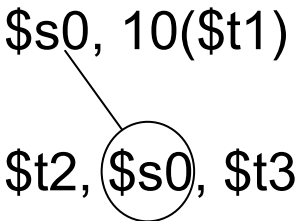
calcolare il CPI effettivo

[senza considerare altri tipi di criticità]

Diagramma temporale:

Dipendenze:

lw \$s0, 10(\$t1)
sub \$t2, \$s0, \$t3



⇒ Ogni volta che una load è seguita da un'istruzione che ne utilizza il dato, ho un ciclo di clock aggiuntivo

- Ricordiamo che:
 - 1) Nel caso ideale il CPI di un processore con pipeline è pari a 1
 - 2) In presenza di stalli:

$$\text{CPI} = \text{CPI ideale} + \text{Cicli di stallo per istruzione}$$

Quindi, supponendo che il 30% delle istruzioni eseguite siano delle load e che nel 50% dei casi, a un'istruzione di load segua un'istruzione che dipende dal risultato della load

Ogni stallo creato in questo modo è di 1 ciclo di clock (come abbiamo visto usando la tecnica di propagazione)

Supponiamo che non esistano altri tipi di stalli

Da cui

$$\text{CPI} = 1 + \underbrace{0,30 \times 0,50 \times 1}_{\text{Cicli di stallo}} = 1,15$$

Esercizio – criticità carica-e-usa con sw

- Si consideri il codice assembler:

```
lw      $s0, 10($t1)
sw      $s0, 20($t3)
```

Tracciare il diagramma temporale ed individuare il numero di stalli necessari, per ciascuno dei seguenti tre casi:

- non è presente alcuna unità di propagazione
- è presente un unità di propagazione verso lo stadio E
- è presente l'unità di propagazione verso lo stadio E e lo stadio M

Dipendenze:

lw \$s0, 10(\$t1)
sw (\$s0), 20(\$t3)

Diagramma senza propagazione:

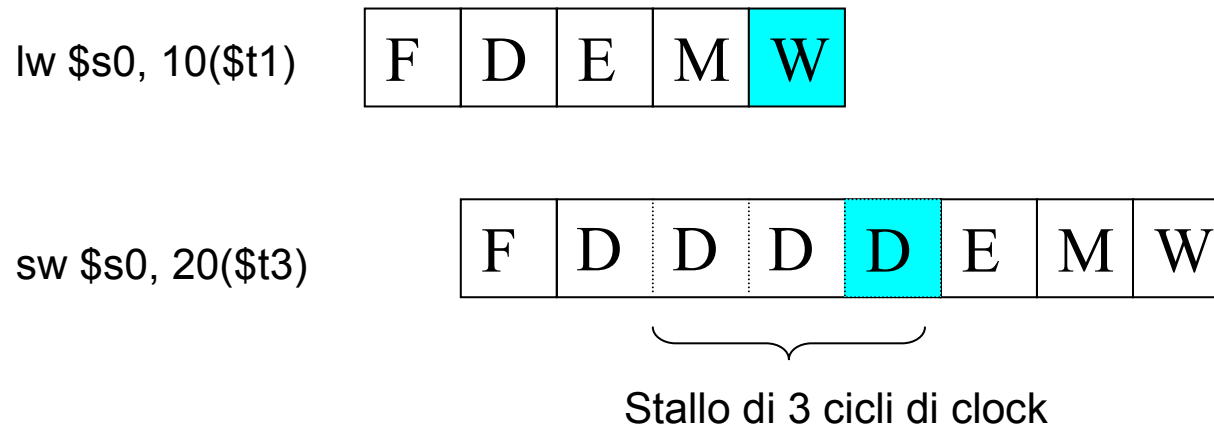
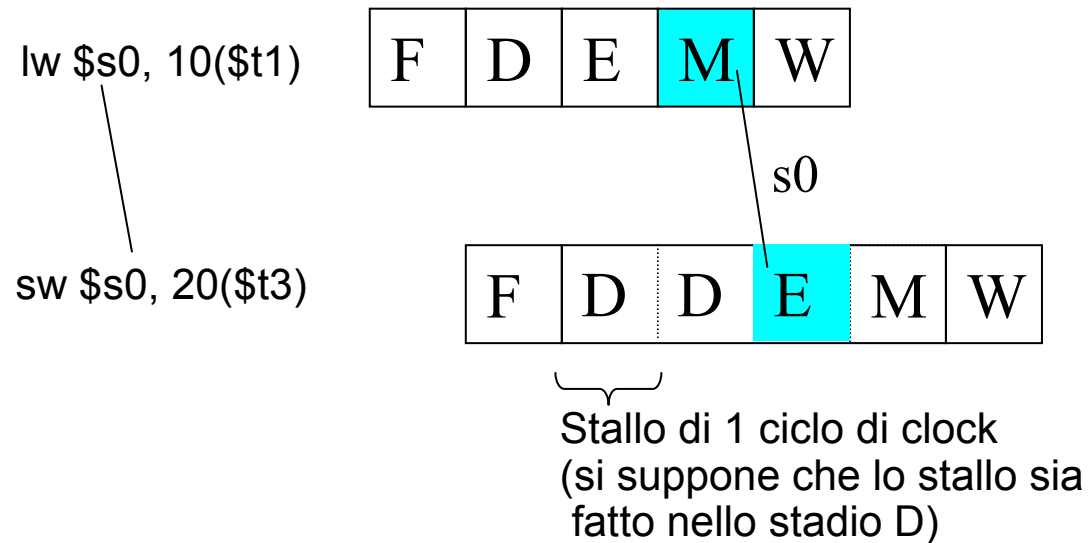


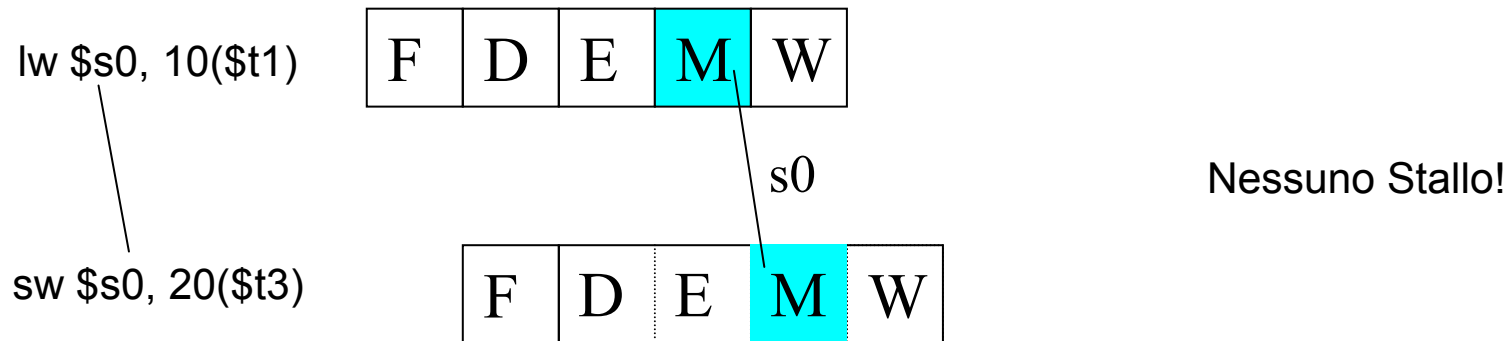
Diagramma con propagazione verso E:



NB: `s0` prodotto dalla `lw` nello stadio M [caricato dalla memoria] e propagato verso lo stadio E di `sw`: `sw` deve comunque “aspettare 1 ciclo”

NB2: Il dato propagato in ingresso a EX (istruzione `sw`) viene trasferito in EX/MEM (sarà usato da `sw` nello stadio M)

Propagazione verso E e verso M:



[Nel ciclo di clock 4, lw accede alla memoria per prelevare s0.
Il dato s0 è quindi disponibile (nel registro interstadio M/W)
nel ciclo di clock 5, quando può essere propagato verso M di sw
che lo pone in memoria nel suo stadio M.]

NB: l'implementazione del MIPS richiede un'estensione per supportare
la propagazione verso M [nella versione base supporta solo propagazione
verso E]

Esercizio: attenzione alla lieve differenza con il precedente...

- Si consideri il codice assembler:

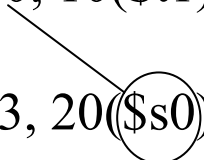
```
lw    $s0, 10($t1)
sw    $t3, 20($s0)
```

Tracciare il diagramma temporale ed individuare il numero di stalli necessari, supponendo di poter fare la propagazione verso qualunque stadio.

Soluzione

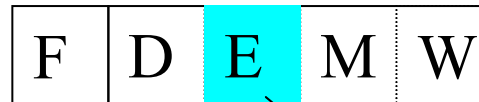
Innanzitutto, si individuano le dipendenze

```
lw    $s0, 10($t1)
sw    $t3, 20($s0)
```



Attenzione: occorre sempre analizzare in che stadio è utilizzato (serve) il dato!!!

sw \$t3, 20(\$s0)



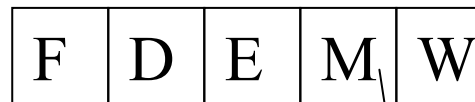
Calcola indirizzo: $\$s0 + 20$

Quindi in questo caso s0 va propagato necessariamente verso E: 1 stallo

lw \$s0, 10(\$t1)

sw \$t3, 20(\$s0)

1 2 3 4 5 6 7 8 9



s0



Stallo di 1 ciclo di clock
(si suppone che lo stallo sia
fatto nello stadio D)

Proposta di esercizi (consideriamo 3 istruzioni):

ES. 1

Si consideri il seguente codice assembler MIPS:

```
lw  $s0, 10($s1)
add $t0, $s0, $s1
add $t1, $t0, $s0
```

Individuare le dipendenze e, supponendo di disporre solo dell'unità di propagazione verso E, disegnare il diagramma temporale.

ES. 2

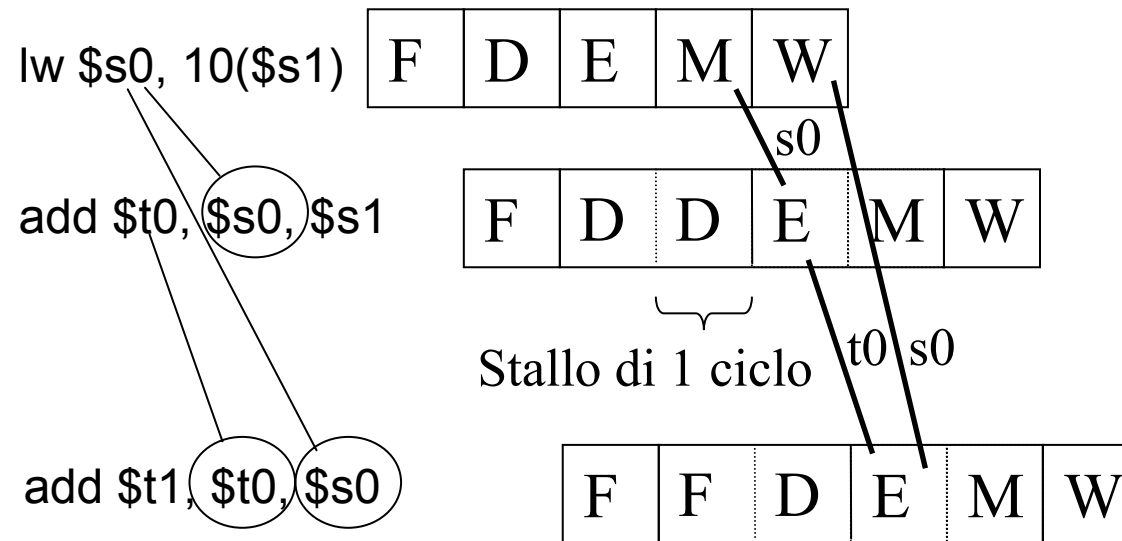
Si considerino i due seguenti codici assembler MIPS:

```
lw  $s0, 10($s1)
add $s0, $t1, $t2
sub $t3, $s0, $t4
```

```
lw  $s0, 10($s1)
add $s1, $t1, $t2
sub $t3, $s0, $t4
```

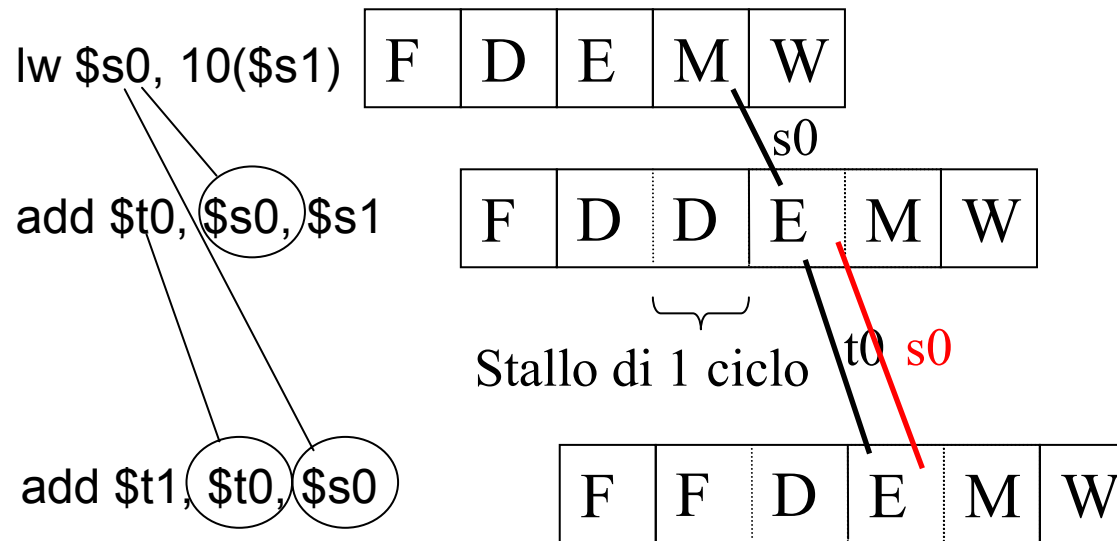
Per ciascuno di essi, individuare le dipendenze e, supponendo di poter propagare i dati verso qualunque stadio, disegnarne il diagramma temporale.

SOLUZIONE ESERCIZIO PROPOSTO N.1



➡ Stallo di un ciclo

ERRORE TIPICO

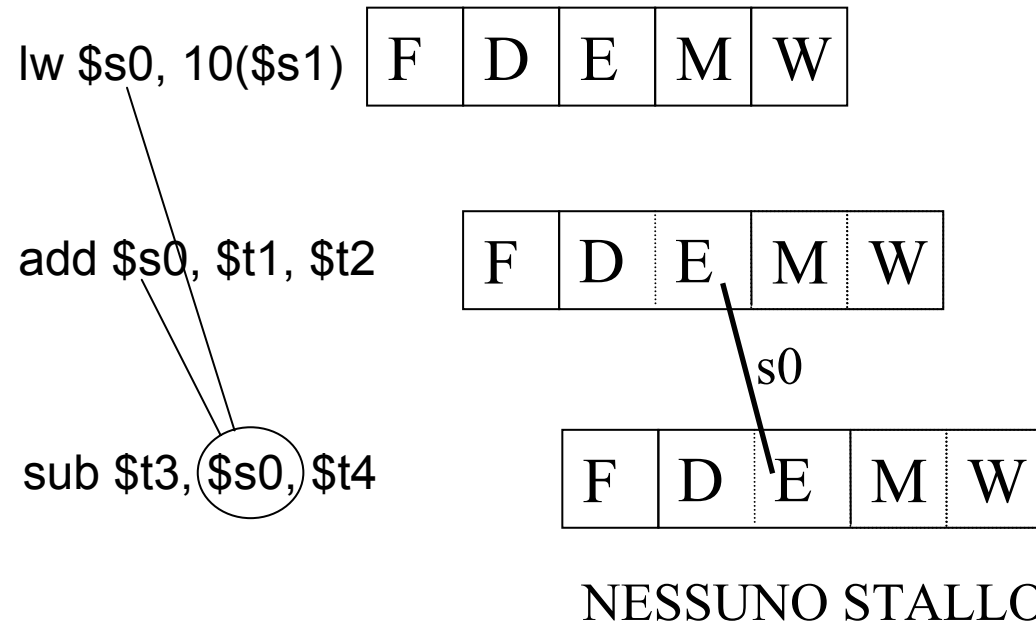


L'istruzione `add` non produce `s0`, anche se utilizza il valore `s0` corretto.
 La propagazione avviene sempre dall'istruzione da cui sussiste la dipendenza; per rendersene conto, si pensi al caso in cui la seconda istruzione sia `add $t0, $s6, $s1`:
 la propagazione deve avvenire secondo le stesse modalità!

SOLUZIONE ESERCIZIO PROPOSTO N.2

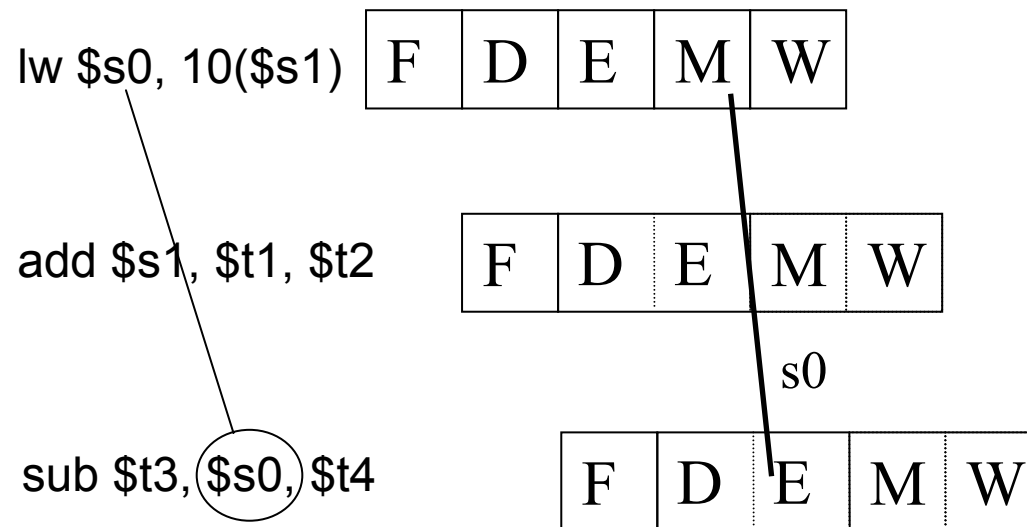
CODICE 1

IMPORTANTE:
INDIVIDUARE
TUTTE LE
DIPENDENZE,
NON SOLO QUELLE
DA PROPAGARE!



Importante: propago dall'istruzione add (la più recente!)

CODICE 2



NESSUNO STALLO

NB: in tal caso ovviamente propago dall'istruzione `lw`

Esercizio – Criticità sui dati e riordinamento delle operazioni

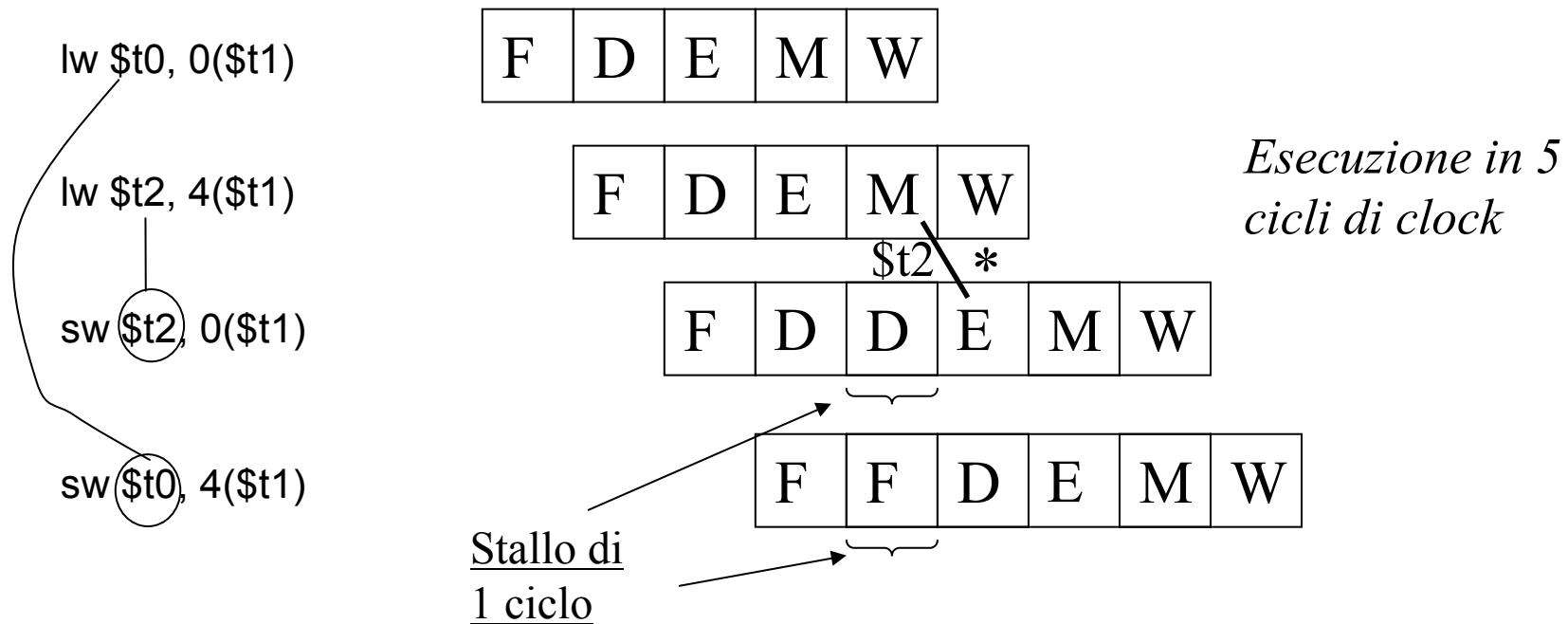
Codice di partenza (NB: è lo scambio $M[\$t1] \leftrightarrow M[\$t1+4]$)

lw	\$t0, 0(\$t1)	
lw	\$t2, 4(\$t1)	} Dipendenza dei dati
sw	\$t2, 0(\$t1)	
sw	\$t0, 4(\$t1)	

Trovare:

- diagramma temporale supponendo propagazione verso E
- supponendo di non disporre di unità di propagazione, riordinamento delle istruzioni per evitare il più possibile stalli della pipeline
- supponendo di disporre di unità di propagazione verso E, riordinamento delle istruzioni per evitare il più possibile stalli della pipeline

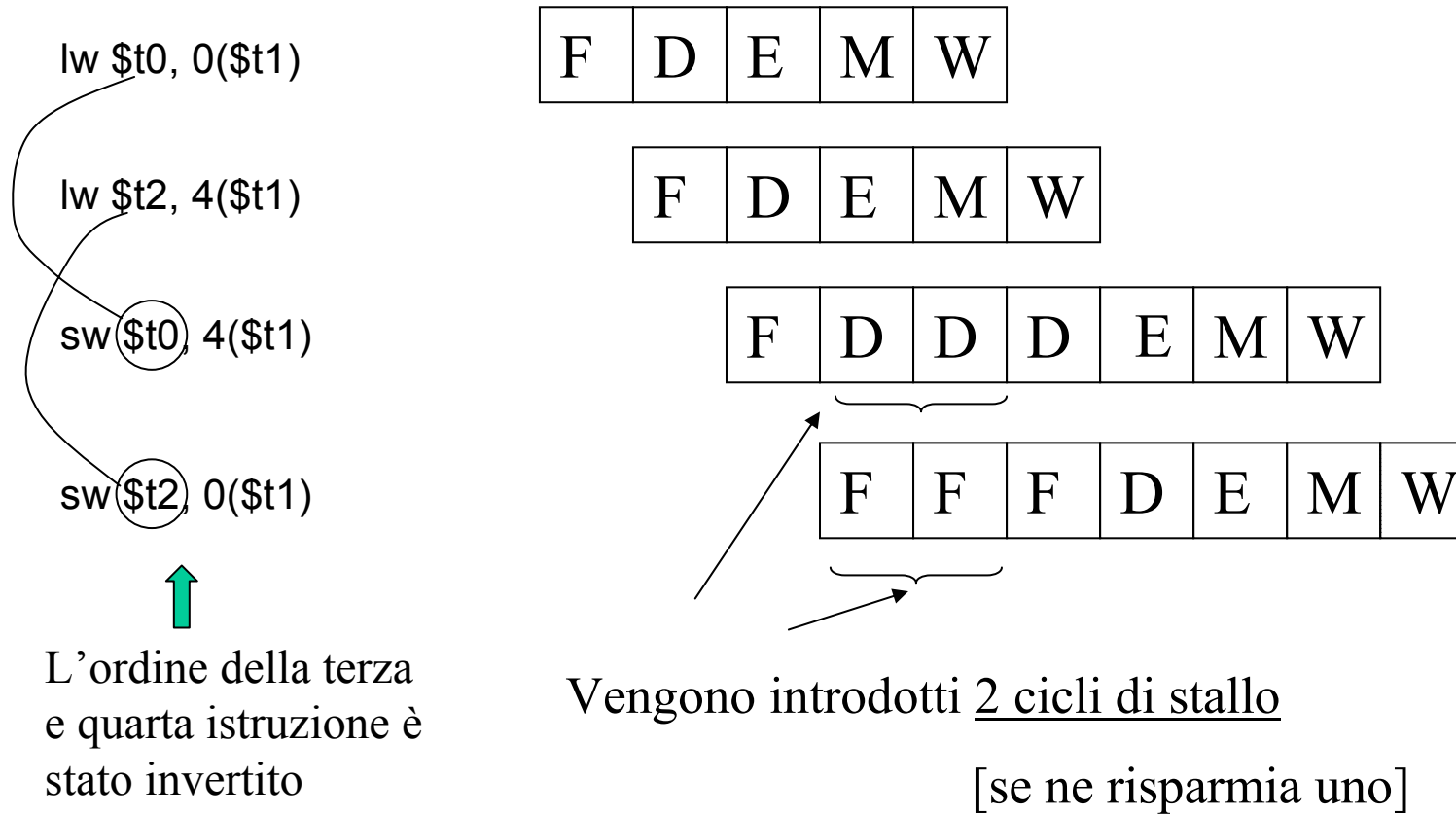
Propagazione verso E



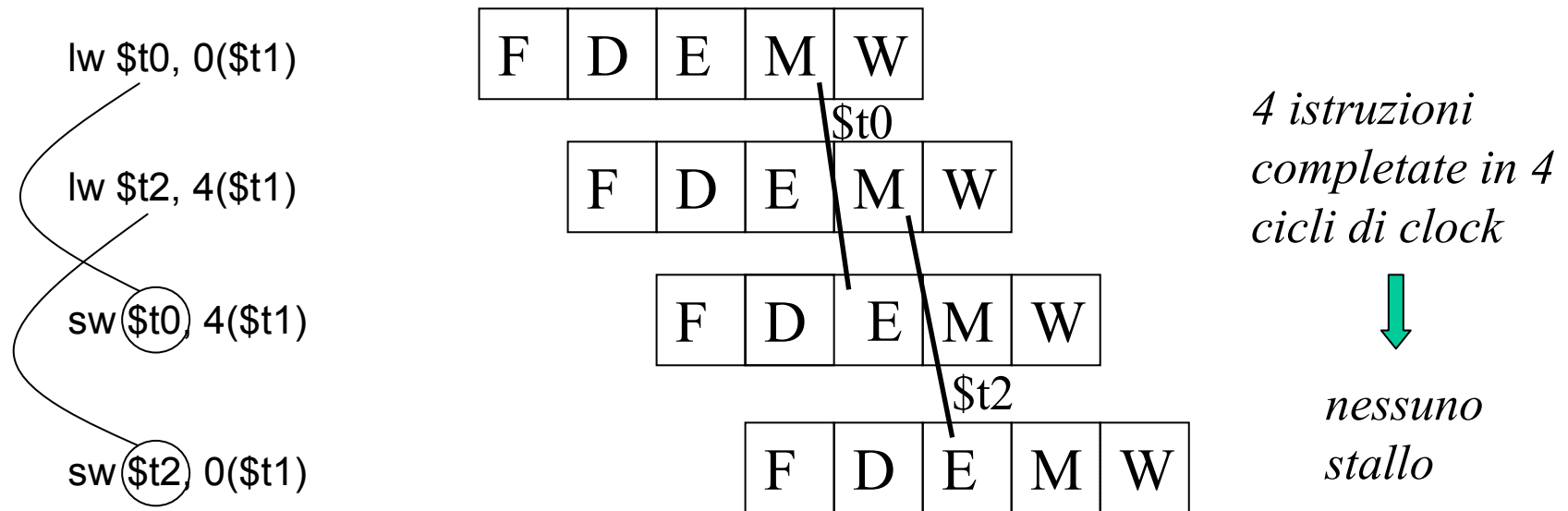
* Il dato letto viene propagato come ingresso di E che lo trasferisce in EX/MEM

NB: è facile constatare che, se non si disponesse dell'unità di propagazione, servirebbe uno stallo di 3 cicli di clock (anziché 1)

Uso del riordino (senza propagazione)



Uso del riordino e della propagazione (verso E)

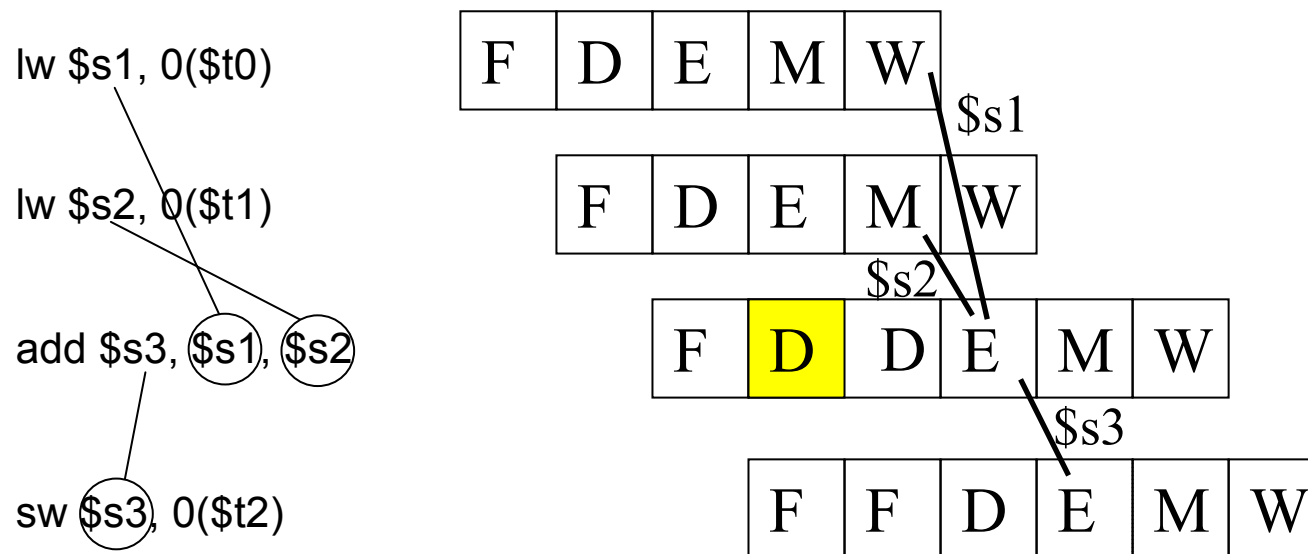


Esercizio/Esempio – Criticità sui dati e riordinamento delle operazioni

La tipica configurazione di codice che deriva da istruzioni del tipo

$$\mathbf{A = B + C}$$

genera **uno stallo** sul valore del secondo dato



Ruolo del compilatore:

- Il compilatore può **modificare la sequenza** di codice per eliminare il conflitto
- Il compilatore cerca di evitare di fare eseguire il **caricamento** e immediatamente dopo **usare il registro destinazione** del caricamento

VEDIAMO COME PARTENDO DA UN ESEMPIO...

- Codice di partenza

$a = b + c;$

$d = e - f;$

$\Rightarrow$ stallo su c nell'addizione

[se si carica prima b , altrimenti stallo su b]

$\Rightarrow$ stallo su f nella sottrazione

Supponiamo che

- b , c , e ed f siano nelle locazioni di memoria il cui indirizzo è rispettivamente in $\$t0$, $\$t1$, $\$t3$, $\$t4$
- a e d vadano memorizzati nelle locazioni il cui indirizzo è rispettivamente in $\$t2$ e $\$t5$

Esecuzione senza intervento del compilatore

lw	\$s1, 0(\$t0)	# carica b
lw	\$s2, 0(\$t1)	# carica c
add	\$s3, \$s1, \$s2	# somma con stallo
sw	\$s3, 0(\$t2)	# memorizza a
lw	\$s4, 0(\$t3)	# carica e
lw	\$s5, 0(\$t4)	# carica f
sub	\$s6, \$s4, \$s5	# sottrazione con stallo
sw	\$s6, 0(\$t5)	# memorizza d

...intervento del compilatore...

lw	\$s1, 0(\$t0)	# carica b
lw	\$s2, 0(\$t1)	# carica c
add	\$s3, \$s1, \$s2	# somma con stallo
sw	\$s3, 0(\$t2)	# memorizza a
lw	\$s4, 0(\$t3)	# carica e
lw	\$s5, 0(\$t4)	# carica f
sub	\$s6, \$s4, \$s5	# sottrazione con stallo
sw	\$s6, 0(\$t5)	# memorizza d



Esecuzione con intervento del compilatore

lw	\$s1, 0(\$t0)	# carica b
lw	\$s2 , 0(\$t1)	# carica c
lw	\$s4, 0(\$t3)	# carica e
add	\$s3, \$s1, \$s2	# (scambiate lw e add)
lw	\$s5 , 0(\$t4)	# carica f
sw	\$s3, 0(\$t2)	# scambiate lw e sw
sub	\$s6, \$s4, \$s5	# sottrazione
sw	\$s6, 0(\$t5)	# memorizza d



Con l'unità di propagazione verso E, non si ha più alcuno stallo!

Dal Tema d'esame 29 giu 2005 [ES. 1]

Si consideri il seguente frammento di codice MIPS:

lw \$t0, 20(\$s1)

sw \$t0, 22(\$s1)

add \$s1, \$s1, \$t0

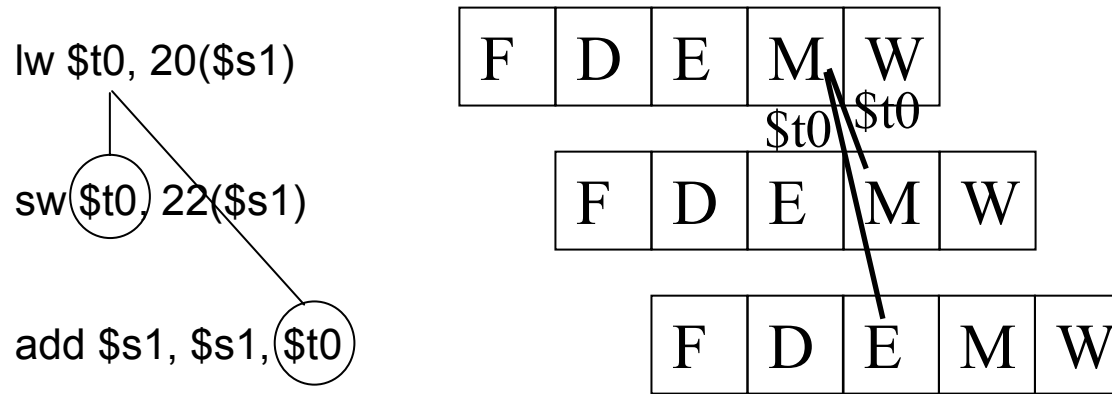
Si consideri l'implementazione con pipeline a 5 stadi

(F: Fetch, D: Decode, E: Execute, M: Mem, W: Write-Back). Si chiede di:

- a) individuare le dipendenze tra i dati
- b) tracciare il diagramma temporale delle istruzioni nell'ipotesi sia disponibile un'unità di propagazione verso lo stadio E ed un'unità di propagazione verso M
- c) tracciare il diagramma temporale supponendo sia disponibile soltanto l'unità di propagazione verso lo stadio M.

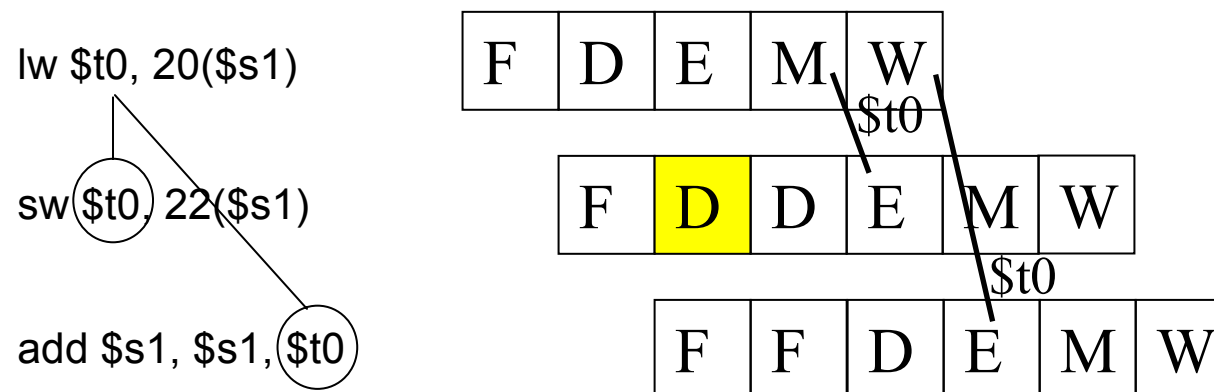
[6]

Caso “Propagazione verso E e verso M”



Nessuno stallo!

Caso “Propagazione solo verso E”



Uno stallo!

Dal Tema d'esame 13 lug 2005 [ES. 3]

Si consideri il seguente frammento di codice in un ipotetico linguaggio assembler:

```
lw $s0, ($s1)
```

```
add $t0, $s0, $s1
```

```
sub $t0, $t0, $s2
```

```
sub $s2, $t0, $s1
```

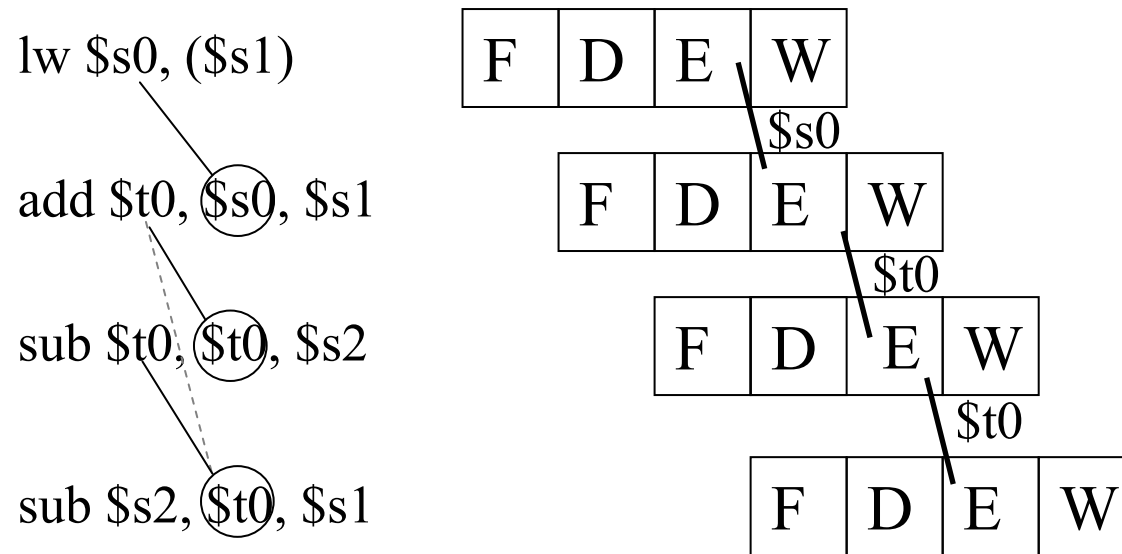
Si consideri l'implementazione con pipeline a 4 stadi
(F: Fetch, D: Decode, E: Execute, W: Write-Back). Si chiede di:

- a) individuare tutte le dipendenze tra i dati
- b) tracciare il diagramma temporale delle istruzioni nell'ipotesi sia disponibile un'unità di propagazione verso lo stadio E.

[4]

Soluzione

NB: si noti che la lettura dalla memoria avviene nello stadio E



NB2: si noti che l'ultima sub propaga t0 dalla sub precedente, non dalla add

Dal Tema d'esame 22 set 2005 [ES. 1]

Si consideri il seguente frammento di codice MIPS:

lw \$t0, 10(\$t0)

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

sw \$t0, 10(\$t0)

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

Si consideri l'implementazione con pipeline a 5 stadi

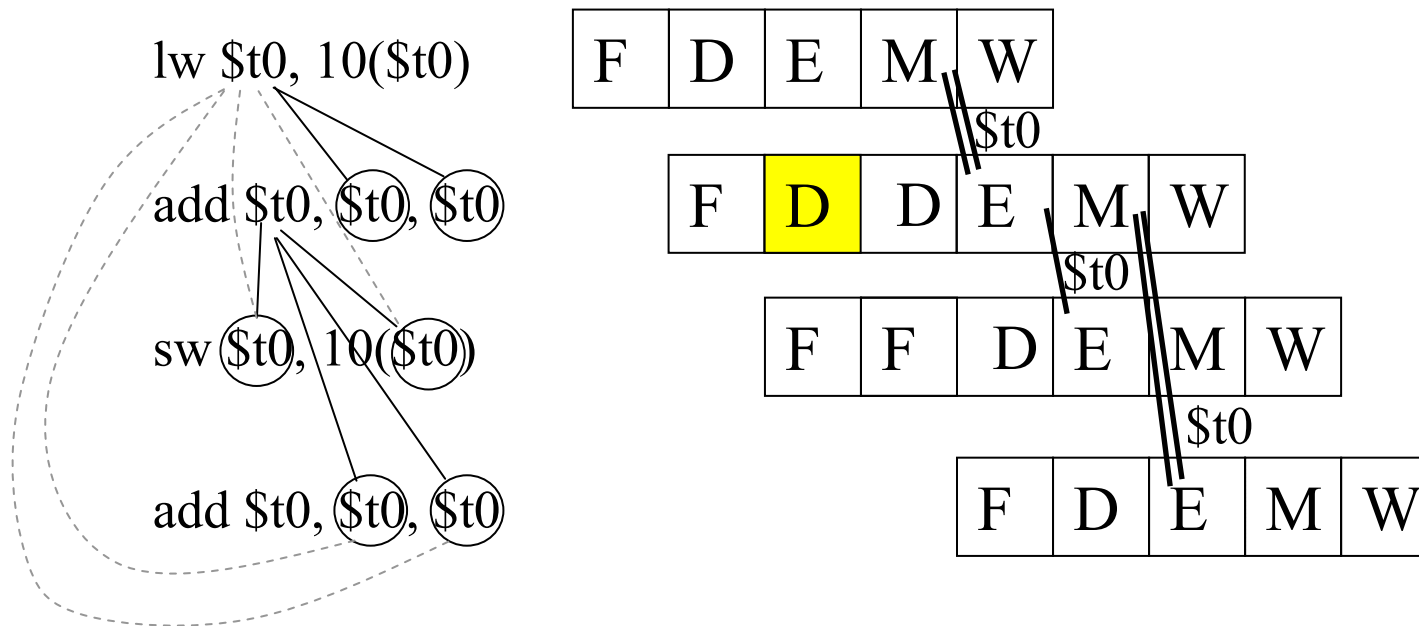
(F: Fetch, D: Decode, E: Execute, M: Mem, W: Write-Back). Si chiede di:

- a) individuare tutte le dipendenze tra i dati
- b) tracciare il diagramma temporale delle istruzioni nell'ipotesi sia disponibile un'unità di propagazione verso lo stadio E.

[4]

Soluzione

[le dipendenze a tratteggio sono quelle che non vengono propagate]



Uno stallo!

Dal Tema d'esame 13 dic 2005 [ES. 2]

Si consideri il seguente frammento di codice MIPS:

```
add    $s0, $t1, $t2  
  
sub    $s1, $s0, $t1  
  
add    $s1, $s1, $s0  
  
lw     $t0, 10($s1)  
  
add    $s1, $t0, $t0
```

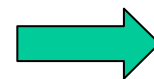
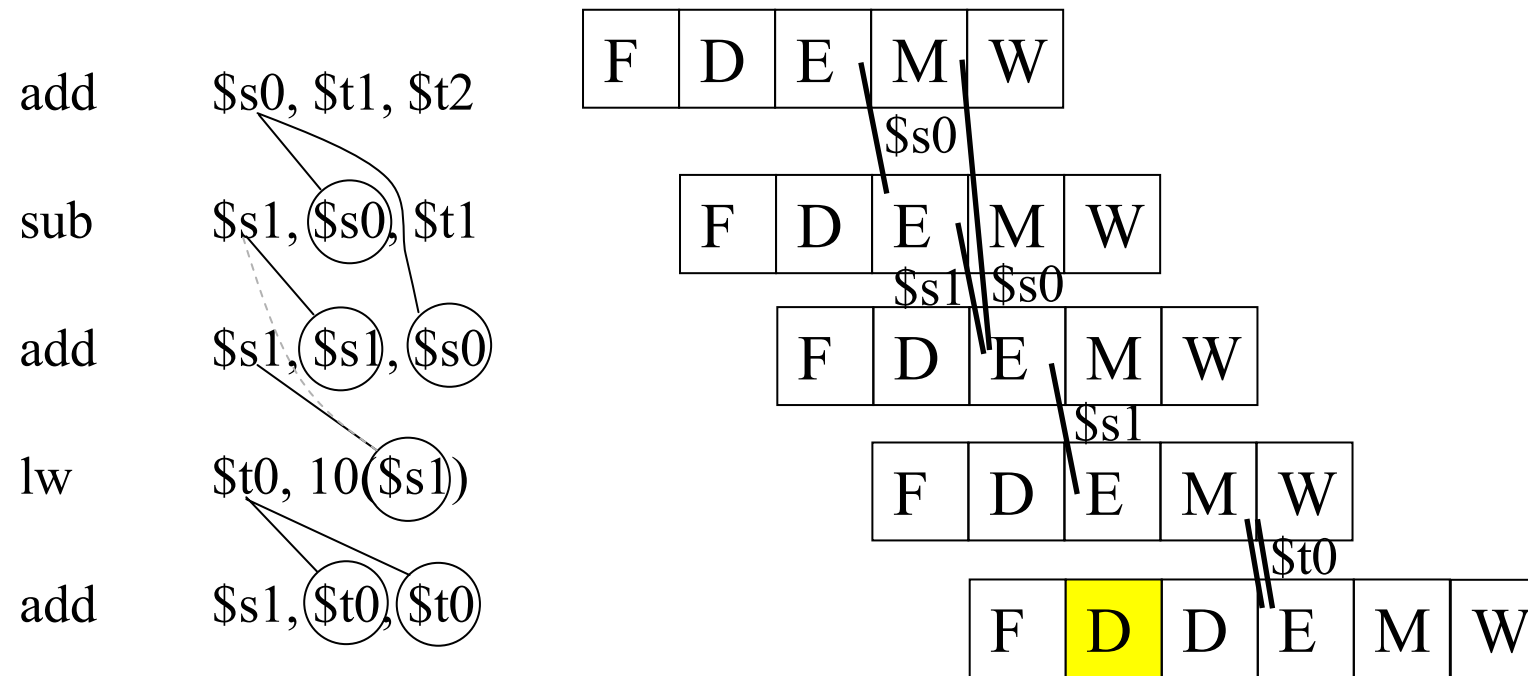
Si consideri l'implementazione con pipeline a 5 stadi

(F: Fetch, D: Decode, E: Execute, M: Mem, W: Write-Back). Si chiede di:

- individuare in modo preciso tutte le dipendenze tra i dati
- tracciare il diagramma temporale delle istruzioni nell'ipotesi sia disponibile un'unità di propagazione verso lo stadio E.
- risolvere il punto precedente supponendo sia disponibile un'unità di propagazione verso lo stadio E ed una verso lo stadio M.

[6]

Soluzione con propagazione verso E



Un ciclo di stallo

Soluzione con propagazione verso E+M

Si vede che non cambia nulla nel grafico! Infatti per l'ultima istruzione (l'unica che stalla) il dato è utilizzato (ovvero, “serve”) nello stadio E, quindi dovrebbe essere necessariamente propagato verso quello stadio anche se si disponesse dell'unità di propagazione verso M!

Dal Tema d'esame 13 dic 2005 [ES. 3]

Si consideri l'implementazione con pipeline a 5 stadi dell'esercizio precedente, per la quale si utilizza un hardware che richiede i seguenti tempi di esecuzione:

- prelievo istruzione e accesso alla memoria dati: 4 ns
- ogni altra operazione critica (ALU, decodifica, lettura e scrittura register file): 2 ns

Si assuma il seguente carico di lavoro:

lw:	20 %
sw:	20 %
formato-R:	40 %
beq:	15 %
j:	5 %

Si supponga che metà delle istruzioni di formato-R, sw e lw siano seguite da istruzioni che ne utilizzano il risultato; in particolare, il 25 % delle istruzioni che seguono lw utilizzano il risultato nello stadio E, il rimanente 25 % utilizzano il risultato in M.

Si calcoli il tempo medio di esecuzione per istruzione:

- nel caso ideale (trascurando tutte le criticità)

Inoltre, trascurando le criticità strutturali e le criticità sui salti, si calcoli il tempo medio di esecuzione nei due casi seguenti:

- disponendo di un'unità di propagazione solo verso lo stadio E
- disponendo di un'unità di propagazione verso lo stadio E ed una verso lo stadio M.

Soluzione

$$\begin{aligned} T_{\text{es.istruz.}} &= \text{CPI} * T_{\text{clock}} \\ &= (1 + \text{cicli_stallo}) * T_{\text{clock}} \end{aligned}$$

➡ $T_{\text{clock}} = 4 \text{ ns}$ (non ci sono due operazioni critiche in serie, la più gravosa è 4 ns)

➡ Cicli_stallo (dovuti solo a dipendenza dati):

- Caso ideale: 0

- Prop. Verso E: dovuti a criticità carica-e-usa, ovvero lw seguita immediatamente da istruzione che ne utilizza risultato (in tal caso un ciclo di stallo)

$$\begin{array}{ccc} 0.2 * 0.5 * 1 = 0.1 \\ \downarrow \quad \downarrow \\ 20\% \text{ di lw} \quad 50\% \text{ sono seguite da istruzioni che ne utilizzano il risultato (nello stadio E o M)} \end{array}$$

- Prop. Verso E+M: se lw è seguita da istruzione che ne utilizza il risultato nello stadio M, la criticità è risolta da propagazione verso M.

$$\begin{array}{ccc} 0.2 * 0.25 * 1 = 0.05 \\ \downarrow \quad \downarrow \\ 20\% \text{ di lw} \quad 25\% \text{ sono seguite da istruzioni che ne utilizzano il risultato nello stadio E} \end{array}$$

Svolgendo i conti risulta

$$T_{\text{es.istruz.}} = (1 + \text{cicli_stallo}) * T_{\text{clock}}$$

- Caso ideale (cicli_stallo = 0):

$$T_{\text{es.istruz.}} = 4 \text{ ns}$$

- Prop. E (cicli_stallo = 0.1):

$$T_{\text{es.istruz.}} = 1.1 * 4 \text{ ns} = 4.4 \text{ ns}$$

- Prop. E+M (cicli_stallo = 0.05):

$$T_{\text{es.istruz.}} = 1.05 * 4 \text{ ns} = 4.2 \text{ ns}$$

NB: notare che non abbiamo utilizzato tutti i dati del problema! Il carico di lavoro ci è servito solo per valutare quante lw ci sono (le uniche che generano una criticità!). Inoltre, le dipendenze interessano solo rispetto a lw.

PRESTAZIONI CON PIPELINE:
CRITICITA' SUL CONTROLLO

Esercizio – Criticità sui salti e stallo

Si consideri una pipeline a cinque stadi in cui l'esecuzione del salto incondizionato (jump) avviene nel secondo stadio, quella del salto condizionato (beq) nel terzo.

Si supponga che l'hardware risolva le criticità sui salti mediante lo stallo della pipeline.

Si supponga che il carico di lavoro preveda, tra le altre istruzioni:

2 % di istruzioni di salto incondizionato

15 % di istruzioni di salto condizionato

Si calcoli il CPI effettivo trascurando le criticità sui dati e strutturali.

Soluzione

Cicli di stallo per salto incondizionato: 1
(aggiornamento PC nel secondo stadio)

j

	PC	
	F	F

Cicli di stallo per salto condizionato: 2
(aggiornamento PC nel terzo stadio)

beq

		PC	
	F	F	F

➡ $\delta_{\text{miss}} = 0,02 \times 1 + 0,15 \times 2 = 0,32 \text{ cicli}$

$$\text{CPI} = 1 + \delta_{\text{miss}} = 1,32$$

Esercizio – Criticità sui salti, stallo e propagazione

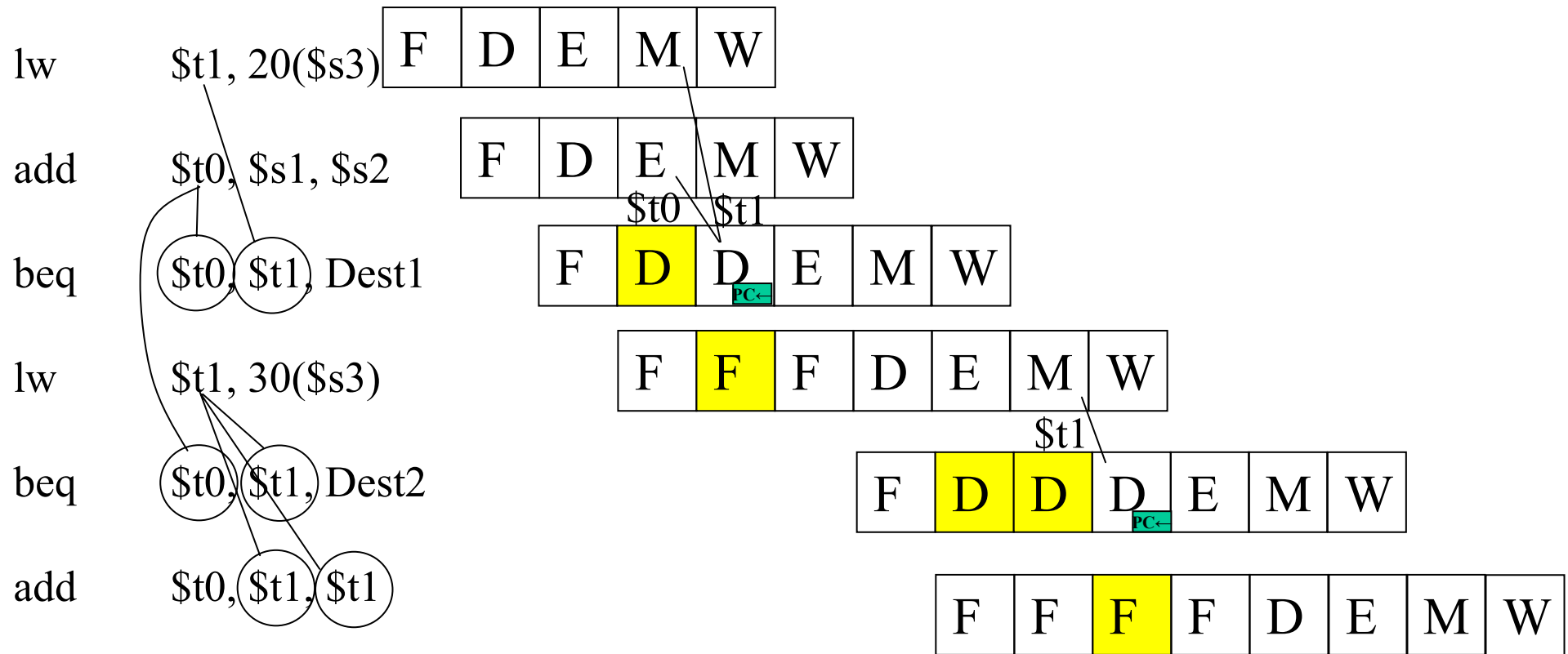
Si consideri il seguente frammento di codice MIPS:

```
lw    $t1, 20($s3)
add   $t0, $s1, $s2
beq   $t0, $t1, Dest1
lw    $t1, 30($s3)
beq   $t0, $t1, Dest2
add   $t0, $t1, $t1
```

Si consideri l'implementazione con pipeline a 5 stadi (F: Fetch, D: Decode, E: Execute, M: Mem, W: Write-Back) in cui le criticità sui salti condizionati sono risolti mediante stallo.

In particolare, sapendo che lo stallo in caso di criticità è di un ciclo di clock, si chiede di tracciare il diagramma temporale delle istruzioni nell'ipotesi che per entrambi i salti condizionati la condizione di salto non si verifichi.

NB: uno stallo nel caso della criticità: da ciò si deduce che il calcolo della condizione e l'aggiornamento del PC avvengono nello stadio D
 ⇒ è necessario propagare in ID se rs e rt dipendono da istruzioni precedenti



➡ In totale 5 cicli di stallo!

Brevi richiami sul salto ritardato

Istruzione di salto

prima istruzione successiva

seconda istruzione successiva

...

n-esima istruzione successiva

Destinazione del salto

**SLOT DI
RITARDO**

*Istruzioni eseguite
indipendentemente dal
successo del salto*

... di solito lo slot di ritardo è limitato a una sola istruzione

Comportamento del salto ritardato

ipotizzando per esempio un solo slot di ritardo:

Caso salto eseguito

Istruzione i di salto eseguito
Istruzione $i+1$ nello slot di ritardo
Destinazione del salto
Destinazione del salto + 1
Destinazione del salto + 2

Caso salto non eseguito

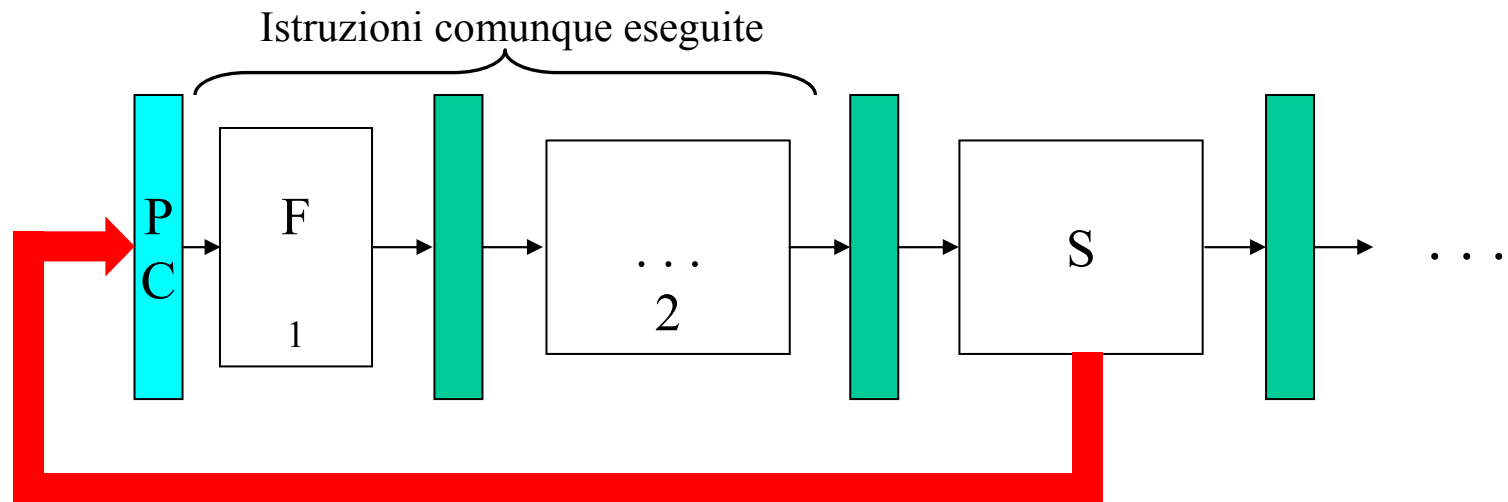
Istruzione i di salto non eseguito
Istruzione $i+1$ nello slot di ritardo
Istruzione $i+2$
Istruzione $i+3$
Istruzione $i+4$

Stadio di esecuzione e slot

- Il numero di slot di ritardo è legato direttamente alla posizione dello stadio in cui il salto viene eseguito (ovvero: PC aggiornato che determina il fetch istruzione destinazione nel ciclo successivo)
- In particolare:

stadio D (nr. 2)	⇒	1 slot di ritardo
stadio E (nr. 3)	⇒	2 slot di ritardo
ecc. ecc.		

Infatti



Esercizio – Riordino delle istruzioni

- Segmento di codice in assembler MIPS [realizza $\$s3 = 2^{\$s2}$]

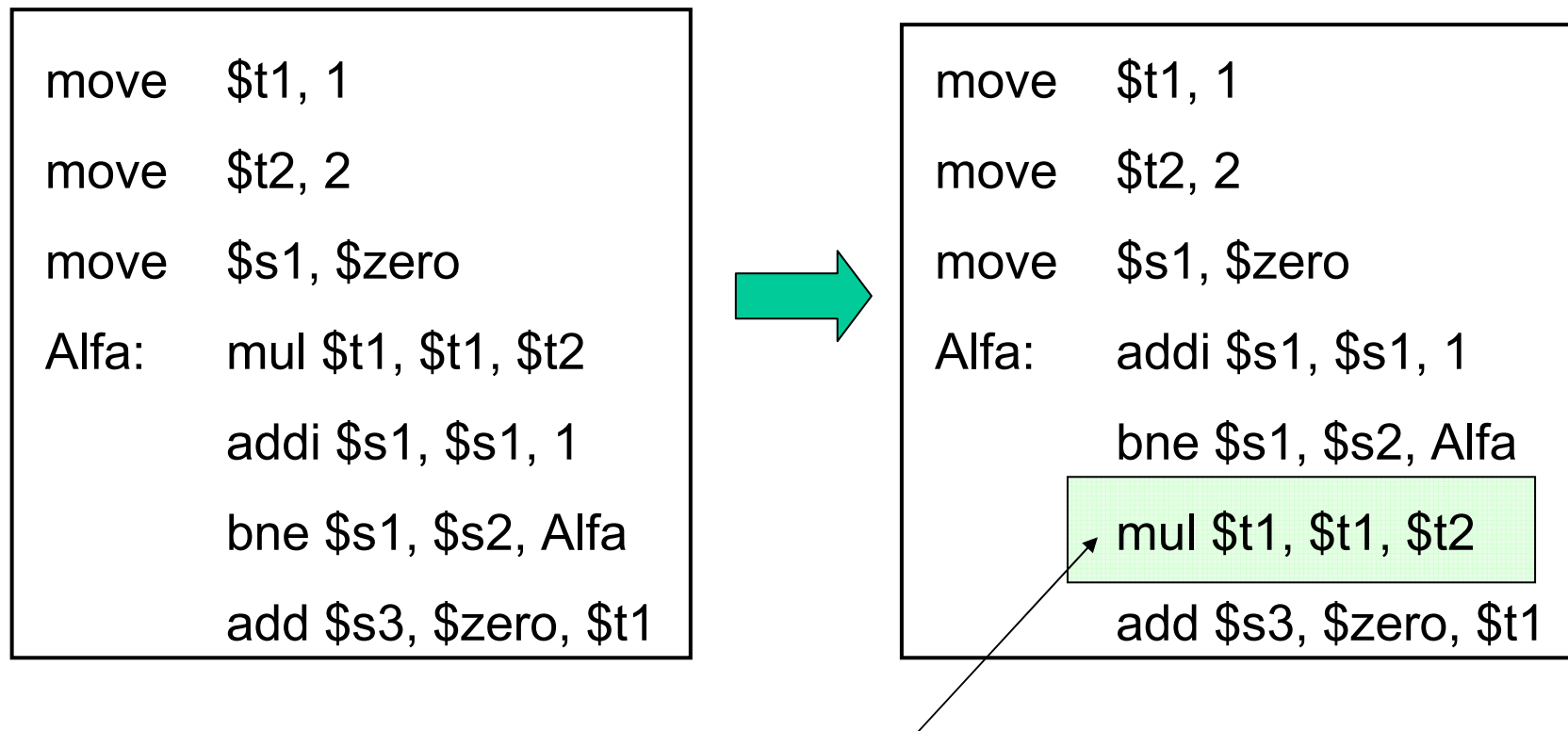
	move	\$t1, 1	// prodotto parz.
	move	\$t2, 2	// base=2
	move	\$s1, \$zero	// indice ciclo
Alfa:	mul	\$t1, \$t1, \$t2	
	addi	\$s1, \$s1, 1	
	bne	\$s1, \$s2, Alfa	
	add	\$s3, \$zero, \$t1	

Considerando una pipeline a due soli stadi (F + E) e utilizzando la tecnica del salto ritardato, riordinare il codice in modo da gestire la criticità sul salto bne e riportare i diagrammi temporali nel caso di “salto eseguito” e di “salto non eseguito”.

Pipeline a due stadi

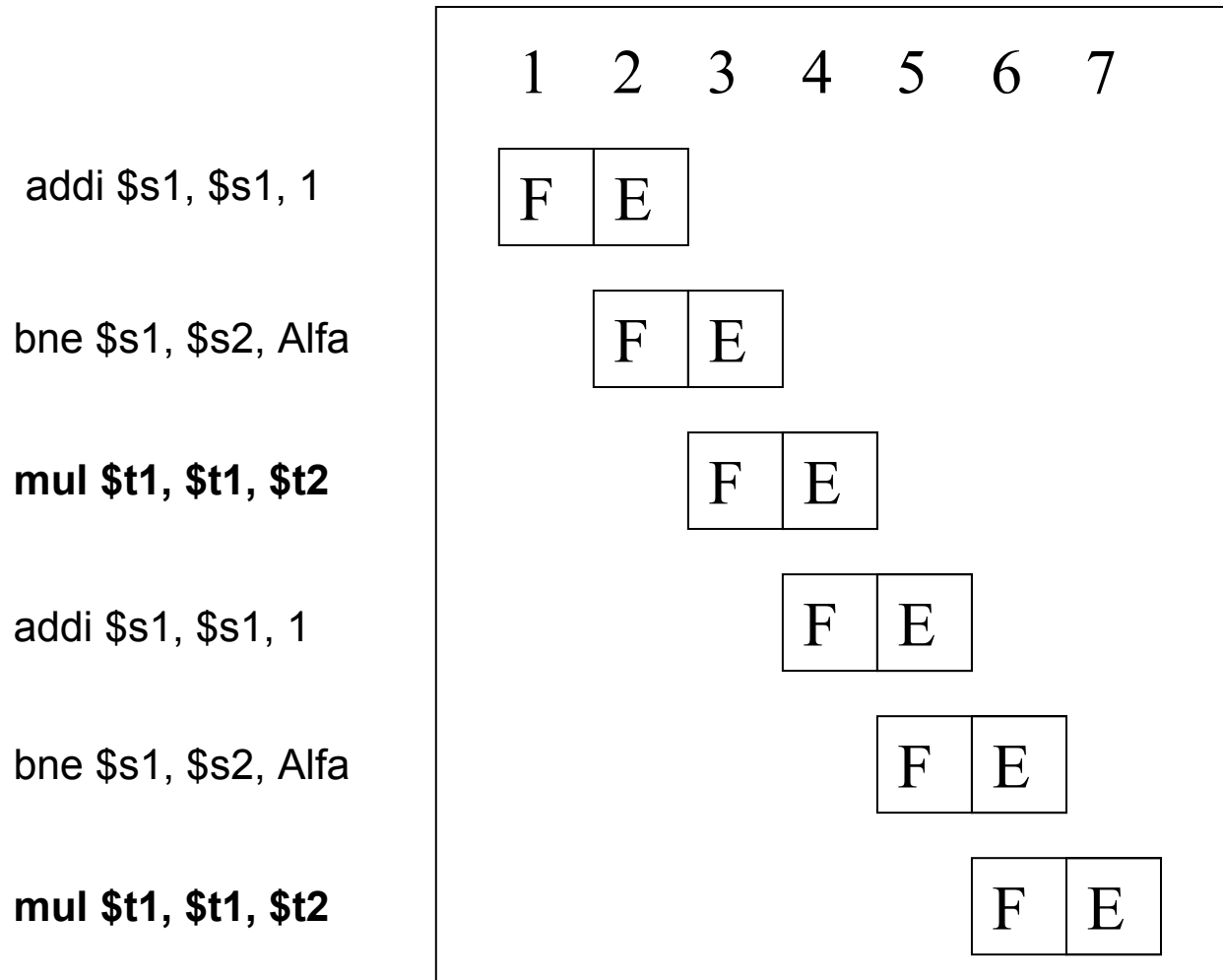
⇒ necessariamente l'aggiornamento del PC nello stadio E
e si ha un solo slot di ritardo!

Riordino delle istruzioni



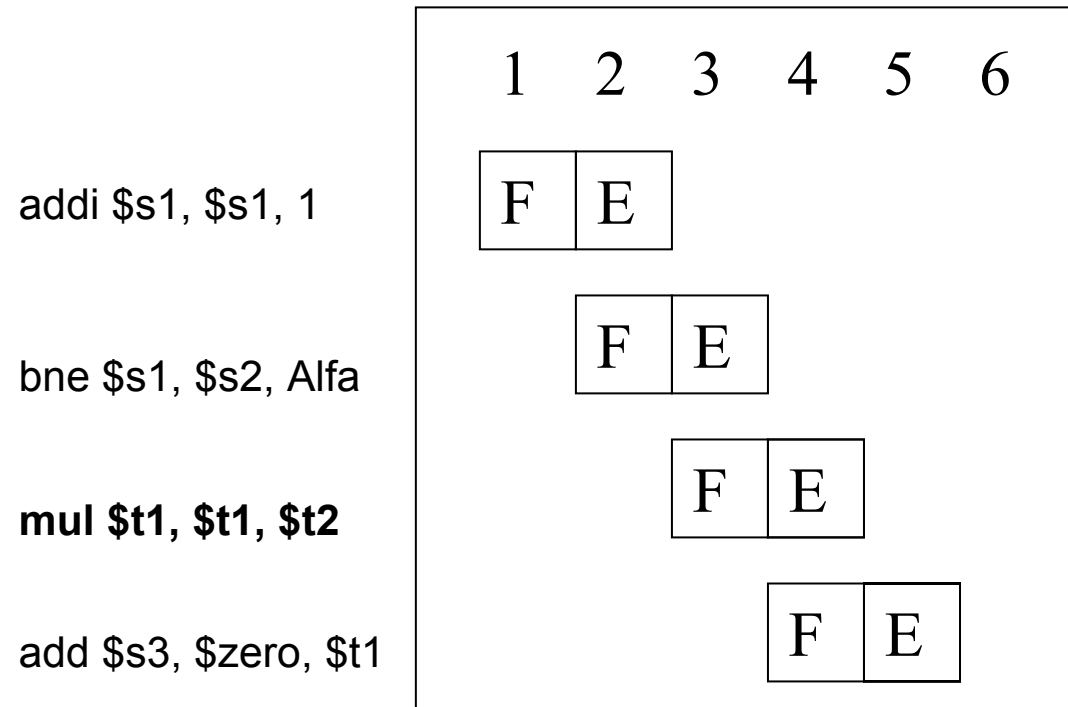
Istruzione inserita nello slot di ritardo

Diagramma temporale nel caso di salto eseguito

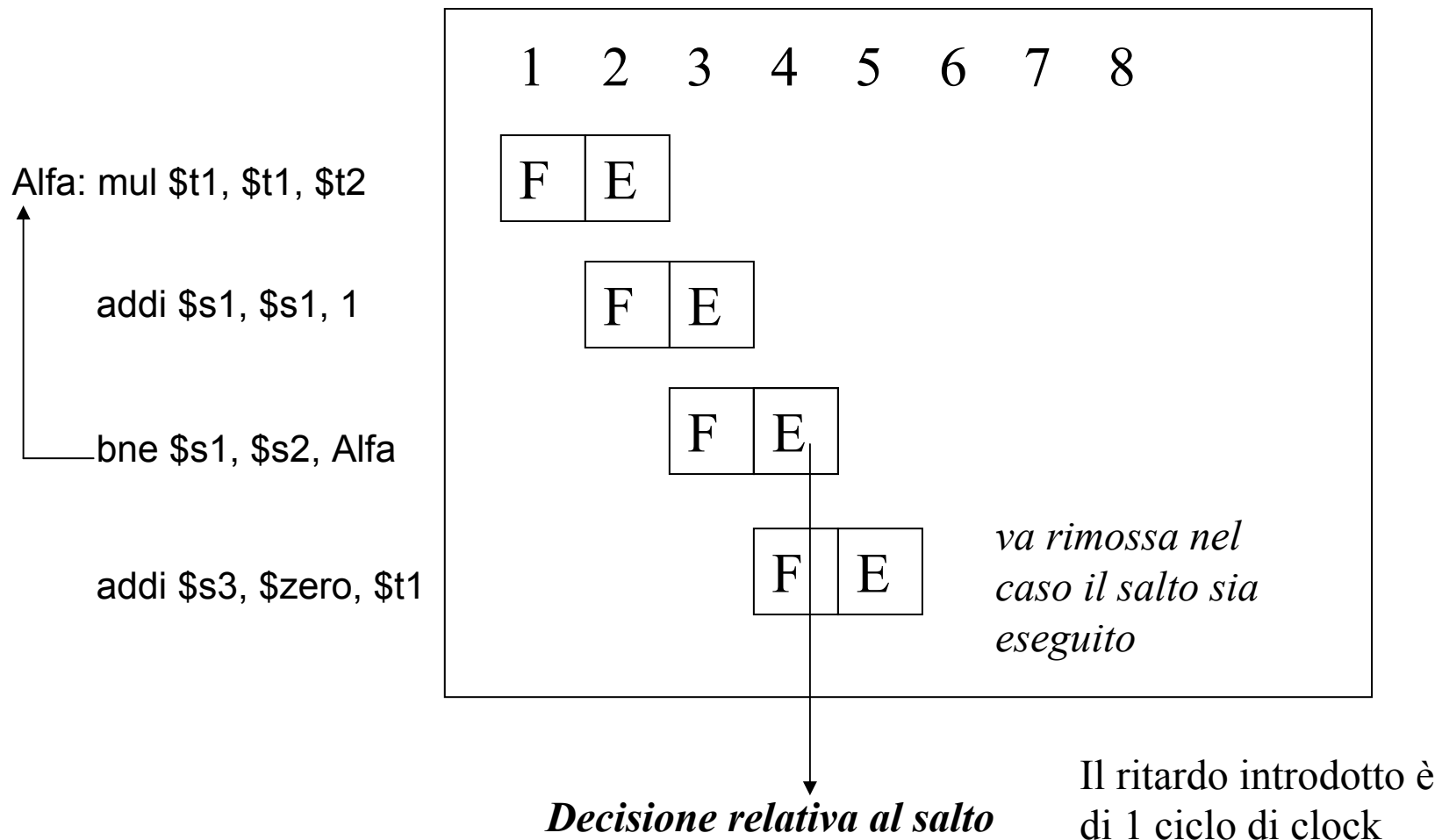


NB: non c'è propagazione di s1 da bne: necessariamente il prelievo degli operandi nelle istruzioni viene fatto in E, sicuramente dopo che l'istruzione precedente ha scritto la destinazione!

Diagramma temporale nel caso di salto non eseguito



NB: Diagramma temporale dell'esecuzione
senza salto ritardato (pipeline a 2 stadi)
-prediz. “salto non eseguito”



Esercizio – Riordino delle istruzioni bis

Consideriamo lo stesso codice ma ipotizzando:

- pipeline a 5 stadi (F-D-E-M-W)
- 1 slot di ritardo

Tracciare il diagramma temporale:

- predizione statica di salto non eseguito
- salto ritardato (caso di salto eseguito)
- salto ritardato (caso di salto non eseguito)

NB: dalle specifiche si capisce che l'esecuzione del salto avviene necessariamente nello stadio D [1 slot di ritardo]

Diagramma temporale dell'esecuzione senza salto ritardato (pipeline a 5 stadi)

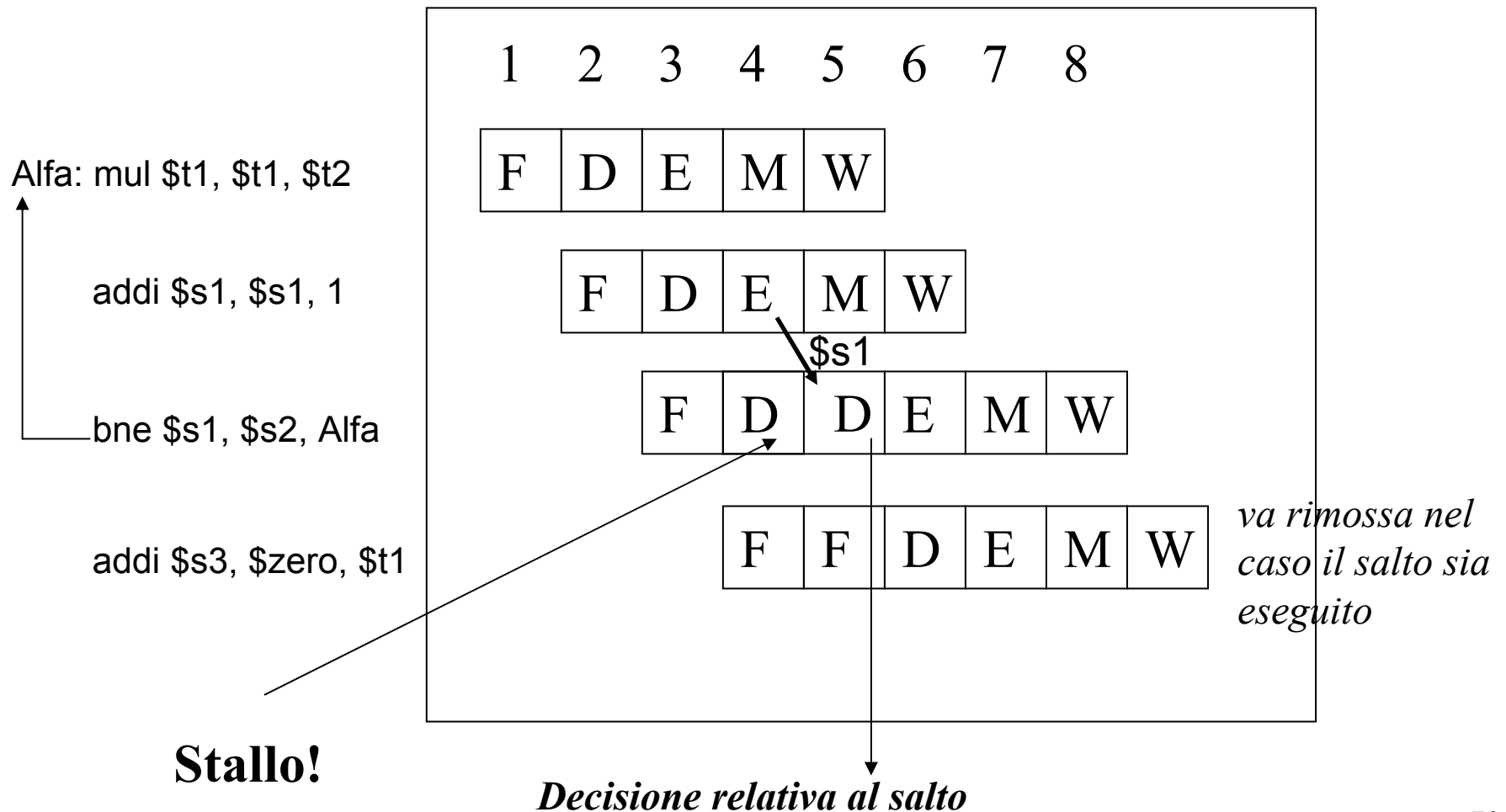


Diagramma temporale dell'esecuzione con salto ritardato: caso 1 (salto eseguito)

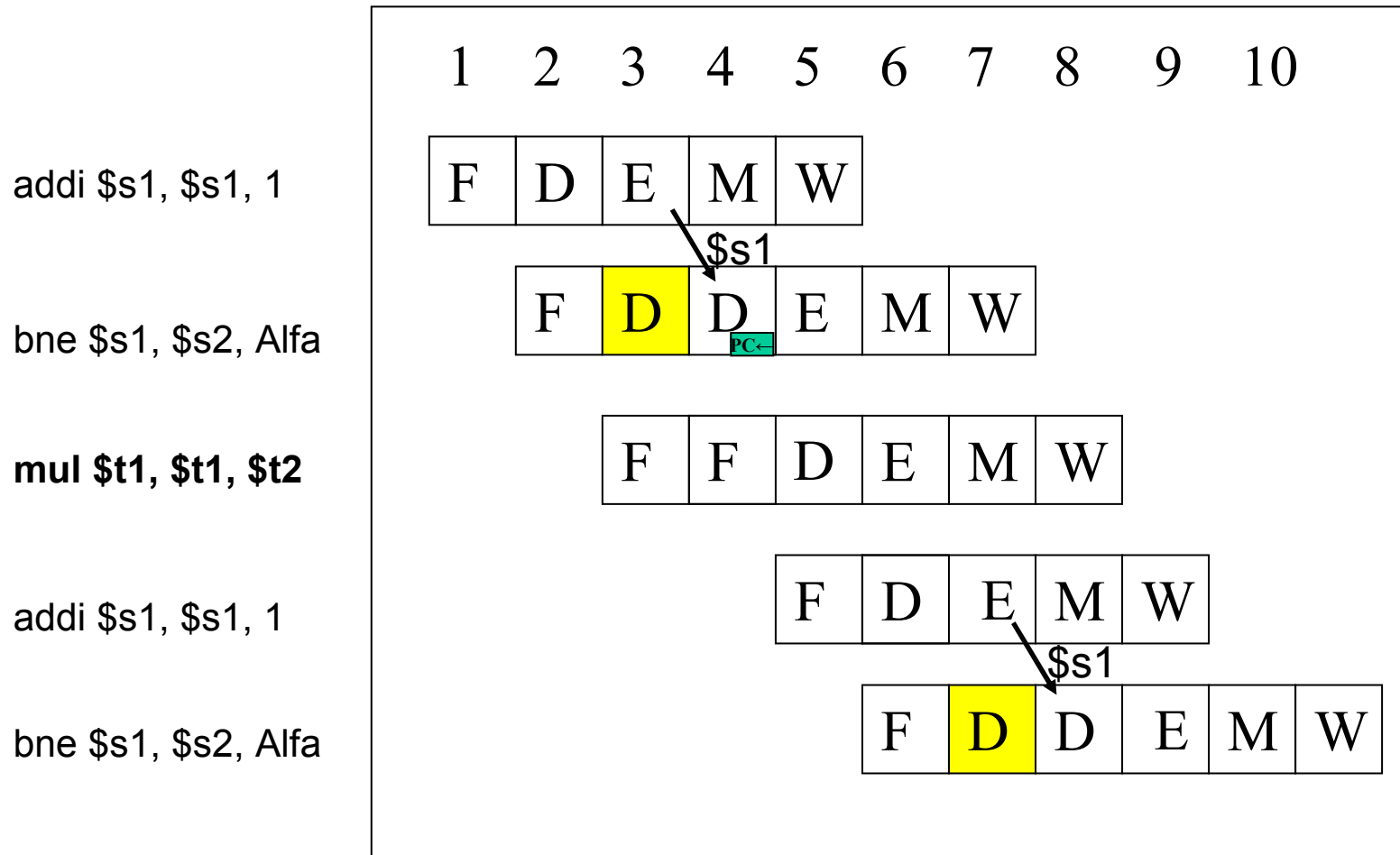
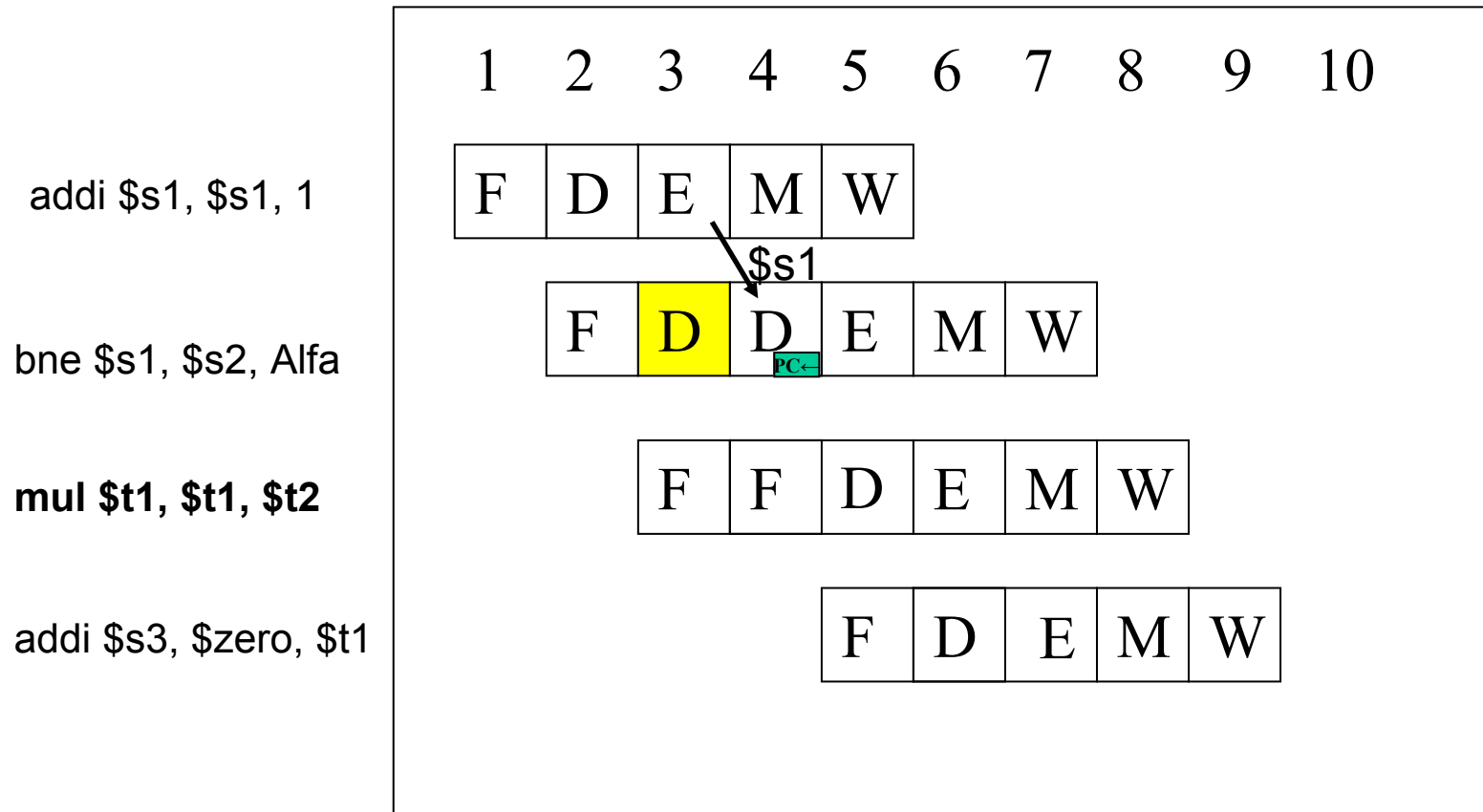


Diagramma temporale dell'esecuzione con salto ritardato: caso 2 (salto non eseguito)



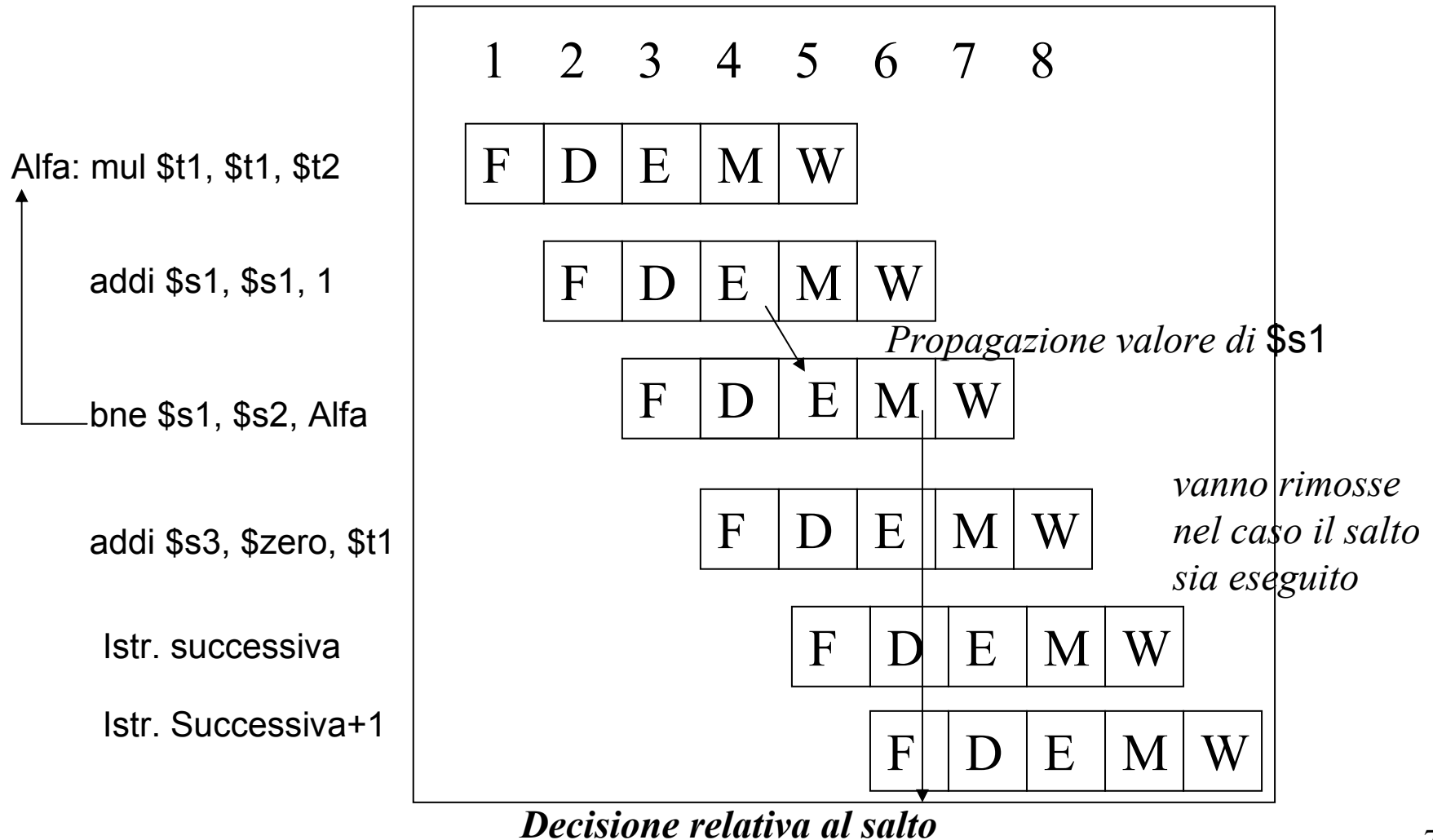
Esercizio – Riordino delle istruzioni tris

Supponiamo che (come nella prima implementazione del MIPS)
l'esecuzione del salto avvenga nello stadio M anziché nello stadio D
e la valutazione della condizione di salto nello stadio E
e ripetiamo l'esercizio

	move	\$t1, 1	// prodotto parz.
	move	\$t2, 2	// base=2
	move	\$s1, \$zero	// indice ciclo
Alfa:	mul	\$t1, \$t1, \$t2	
	addi	\$s1, \$s1, 1	
	bne	\$s1, \$s2, Alfa	
	add	\$s3, \$zero, \$t1	

Diagramma temporale dell'esecuzione senza salto ritardato (pipeline a 5 stadi)

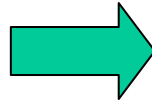
Modifica PC relativa al salto nello stadio M



Riordino delle istruzioni

Adesso gli slot da riempire sono tre! Ma abbiamo solo una istruzione disponibile!

```
move    $t1, 1
move    $t2, 2
move    $s1, $zero
Alfa:   mul $t1, $t1, $t2
        addi $s1, $s1, 1
        bne $s1, $s2, Alfa
        add $s3, $zero, $t1
```



```
move    $t1, 1
move    $t2, 2
move    $s1, $zero
Alfa:   addi $s1, $s1, 1
        bne $s1, $s2, Alfa
        mul $t1, $t1, $t2
        nop
        nop
        add $s3, $zero, $t1
```

Istruzioni inserite negli slot di ritardo

Diagramma temporale dell'esecuzione con salto ritardato: caso 1 (salto eseguito)

Modifica PC relativa al salto nello stadio M

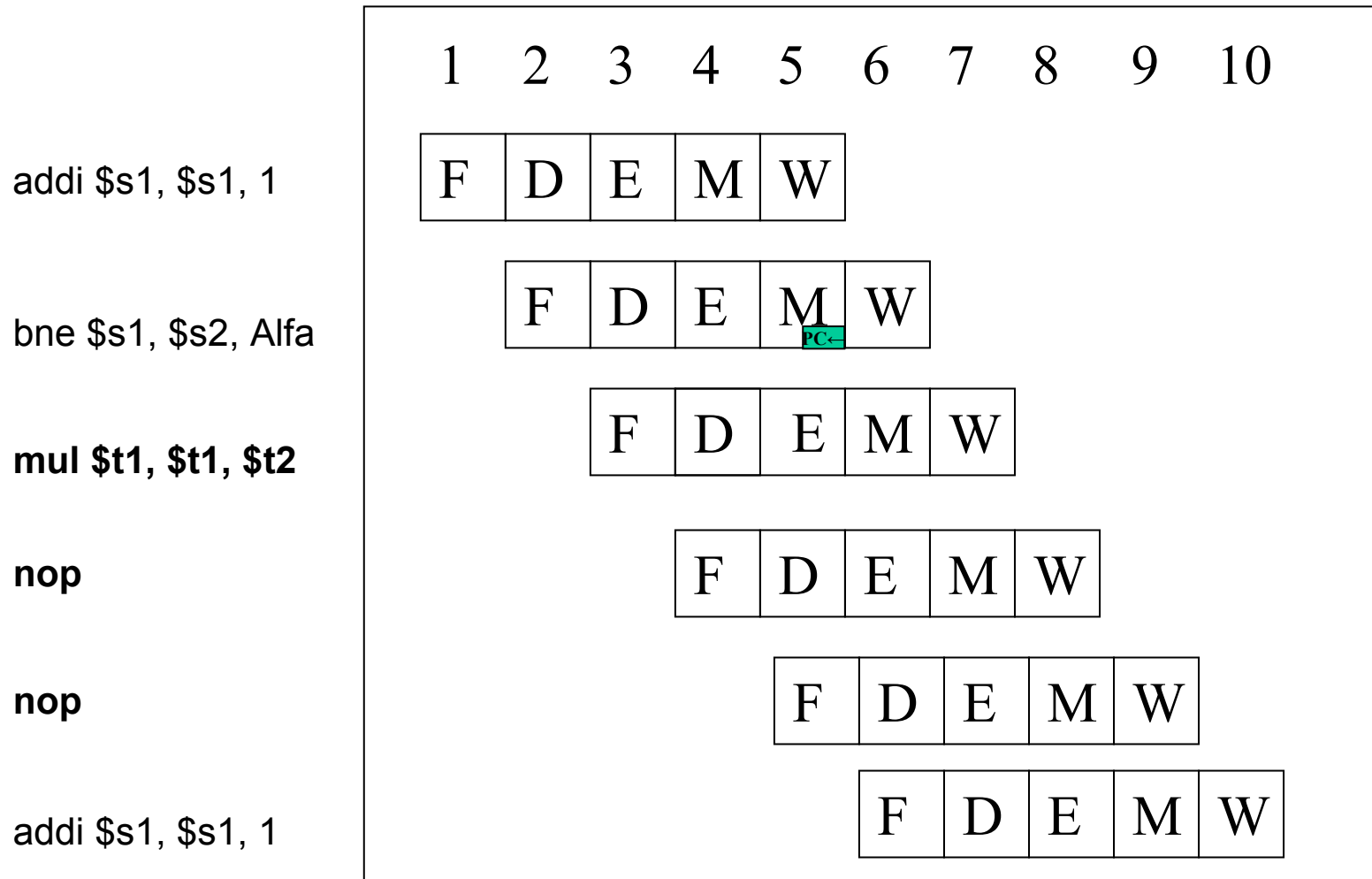
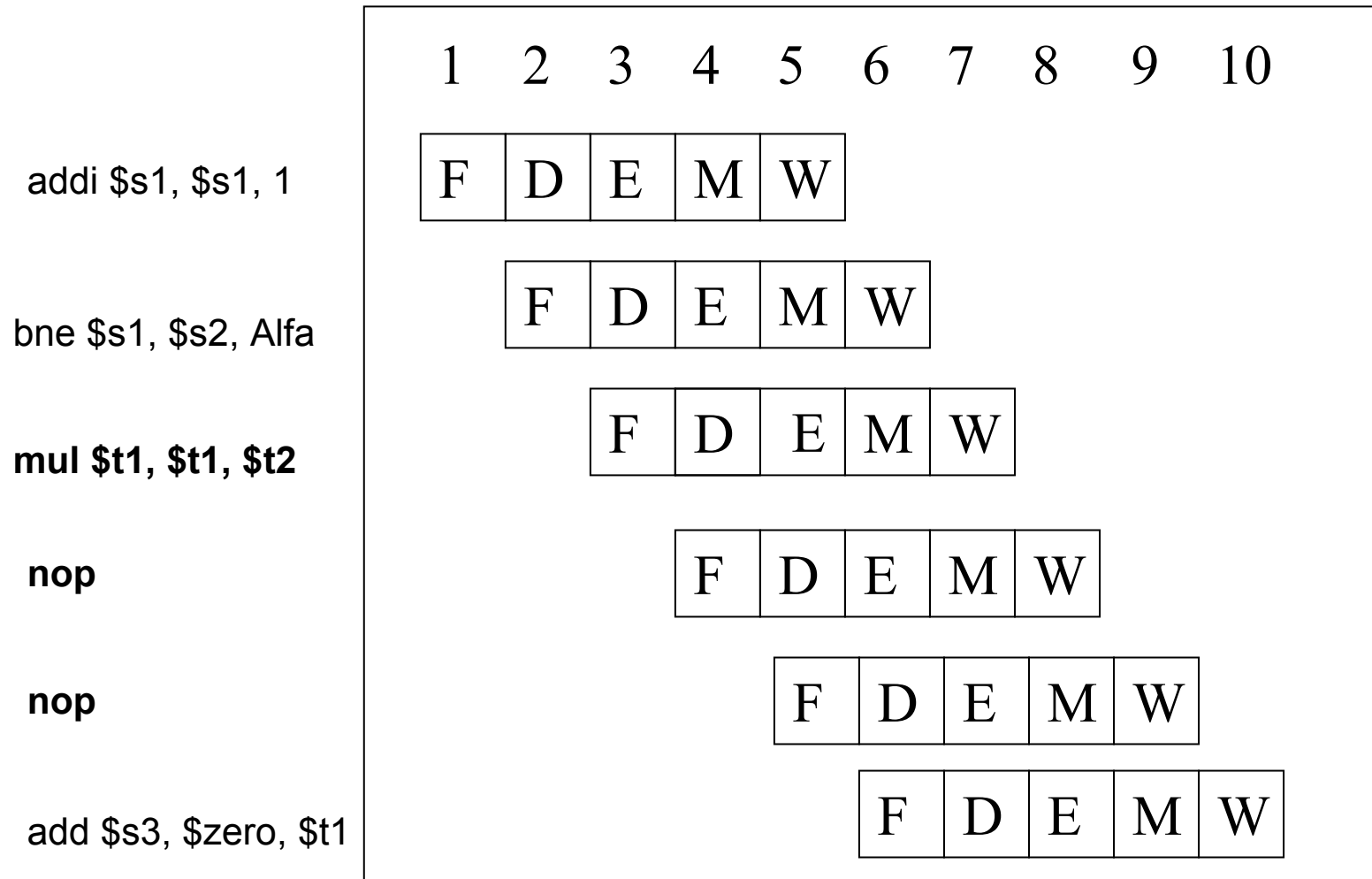


Diagramma temporale dell'esecuzione con salto ritardato: caso 2 (salto non eseguito)



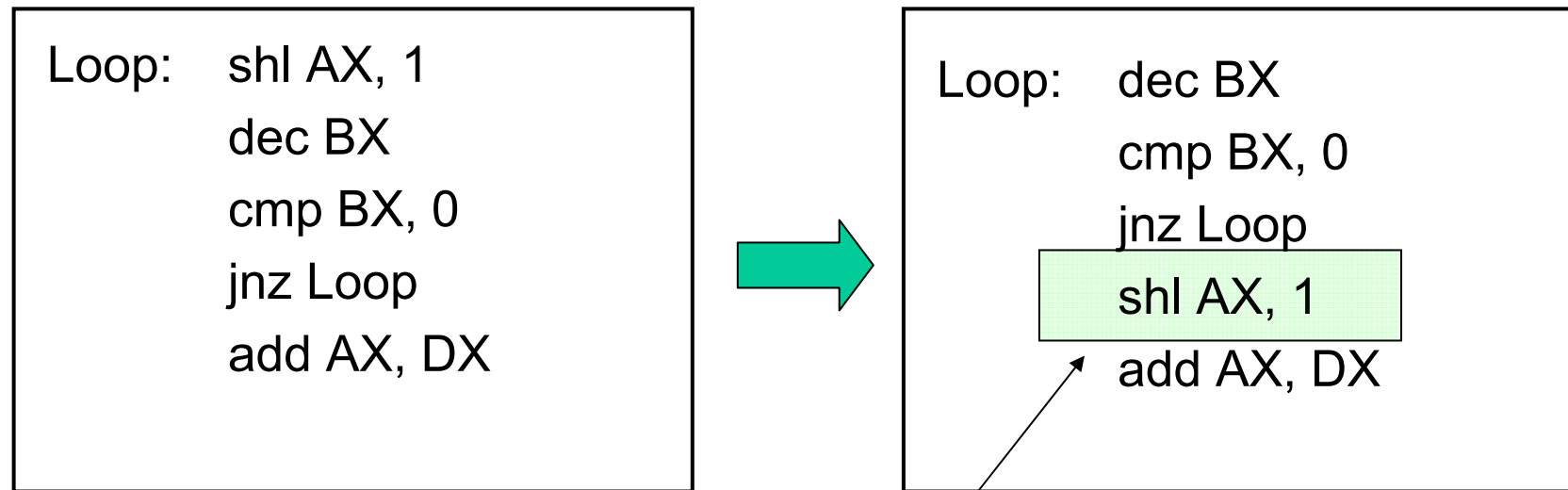
Altro Esempio

- Segmento di codice in assembler Intel

```
Loop: shl      AX, 1  
      dec      BX  
      cmp      BX, 0  
      jnz      Loop  
      add      AX, DX
```

NB: fa lo shift a sinistra di AX per un numero BX volte

Riordino delle istruzioni nel caso di salto ritardato



*Istruzione inserita nello slot di ritardo
(viene sempre eseguita)*

Esercizio: analizzare l'esecuzione del codice su una pipeline a 2 stadi

Esercizio – Riordino delle istruzioni 2

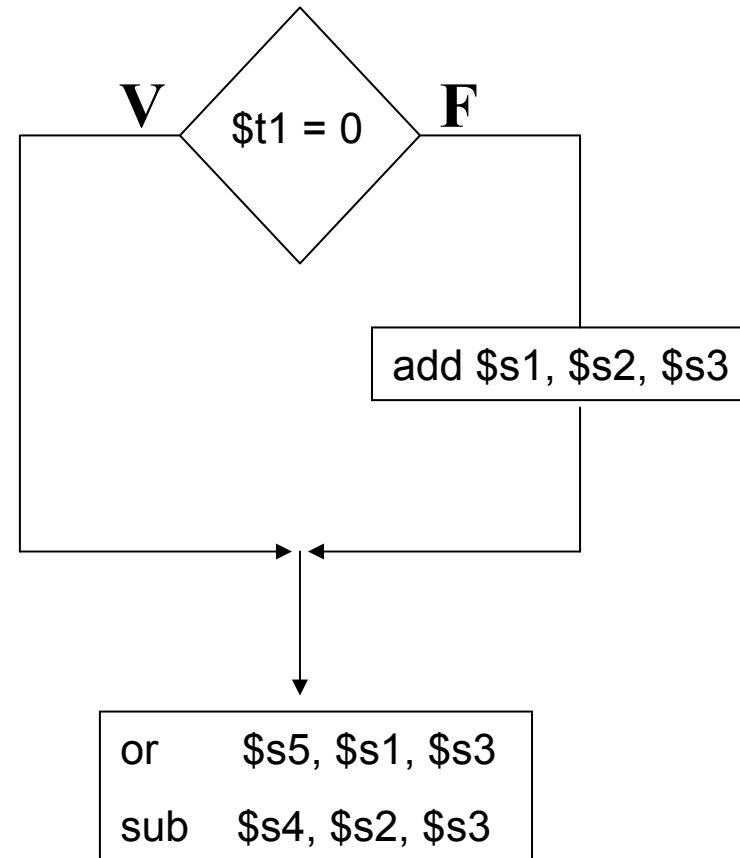
- Segmento di codice in assembler MIPS

```
        beq    $t1, $zero, Alfa
        add    $s1, $s2, $s3
Alfa:   or     $s5, $s1, $s3
        sub    $s4, $s2, $s3
```

Assumendo uno slot di ritardo,
riordinare il codice per gestire la criticità su beq

La logica del codice è:

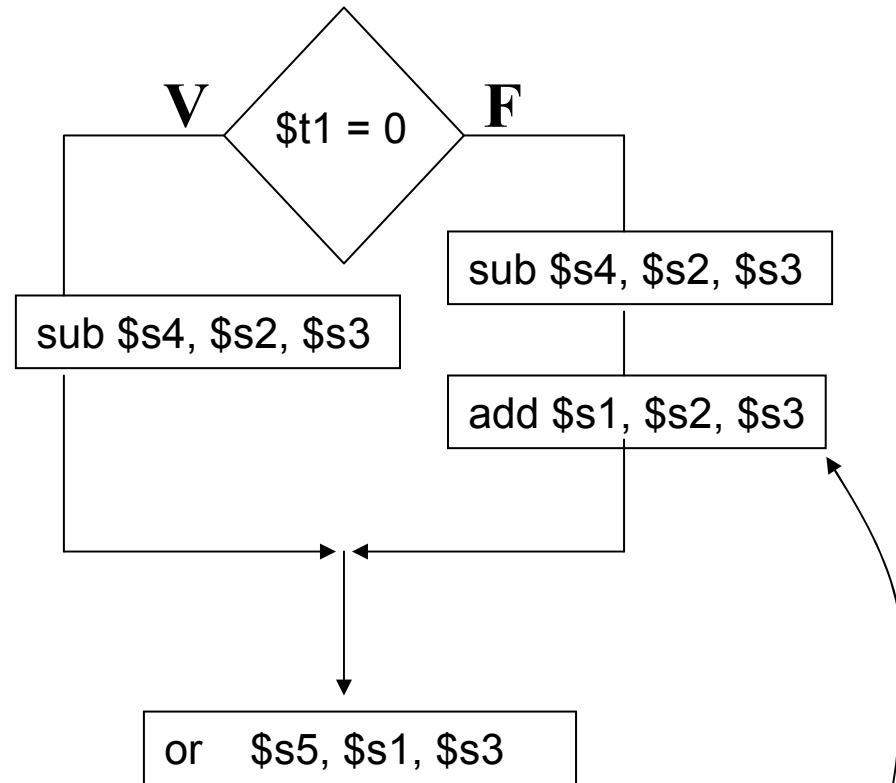
Alfa: beq \$t1, \$zero, Alfa
 add \$s1, \$s2, \$s3
 or \$s5, \$s1, \$s3
 sub \$s4, \$s2, \$s3



QUINDI:

- Segmento di codice in assembler MIPS

```
beq    $t1, $zero, Alfa
sub     $s4, $s2, $s3
add     $s1, $s2, $s3
Alfa:  or     $s5, $s1, $s3
```



NB: notare che la or non la potevamo anticipare, perché dipende dal risultato di

Esercizio P.C: analizzare l'esecuzione del codice su una pipeline a 2 stadi e a 5 stadi

Esercizio – Calcolo prestazioni 1

- Si ipotizzi che il 20% delle istruzioni eseguite da un calcolatore siano istruzioni di salto
- Il calcolatore adotta la tecnica del salto ritardato con 1 intervallo di ritardo
- Determinare il CPI nel caso in cui il compilatore sia in grado di utilizzare l'85% degli intervalli di ritardo (si assuma che non esistano altre cause di stallo oltre ai salti)

Soluzione:

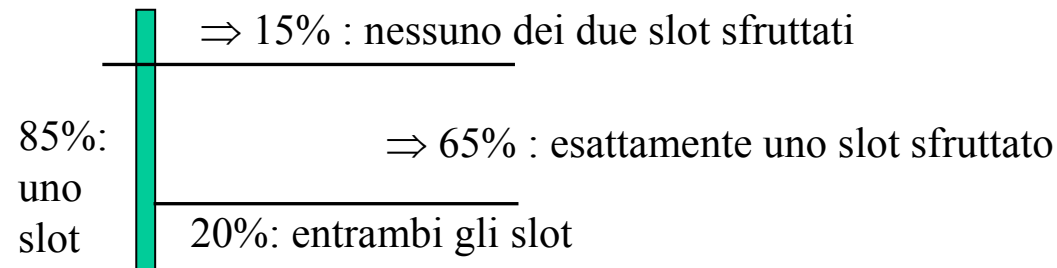
$$\text{CPI} = 1 + \underbrace{0,20 \times 0,15 \times 1}_{\text{Nel 15\% dei salti non viene utilizzato l'intervallo di ritardo, e quindi 1 ciclo viene perso}} = 1,3$$

Nel 15% dei salti non viene utilizzato l'intervallo di ritardo, e quindi 1 ciclo viene perso

Esercizio – Calcolo prestazioni 2

- Si ipotizzi che il 20% delle istruzioni eseguite da un calcolatore siano istruzioni di salto
- Il calcolatore adotta la tecnica del salto ritardato con 2 intervalli di ritardo
- Determinare il CPI supponendo che il compilatore sia in grado di riempire almeno il primo intervallo nell'85% delle volte, entrambi gli intervalli nel 20% dei casi (si assuma che non esistano altre cause di stallo oltre ai salti)

Soluzione:



$$\Rightarrow \text{CPI} = 1 + 0,20 \times \underbrace{(0,15 \times 2)}_{\substack{\text{nel 15\% dei casi} \\ \text{perde 2 cicli}}} + \underbrace{0,65 \times 1}_{\substack{\text{nel 65\% dei casi} \\ \text{perde 1 ciclo}}} = 1,19$$

Dal Tema d'esame 13 lug 2005 [ES. 4]

Si consideri il seguente frammento di codice MIPS:

```
lw $s0, 20($t1)
```

```
beq $s0, $s1, Ndb
```

```
add $t2, $t2, $t2
```

```
Ndb:  add $t2, $t2, $t3
```

```
sub $s2, $s2, $s3
```

Si consideri un'implementazione tramite pipeline a 6 stadi in cui la decisione e l'esecuzione del salto beq avviene nel secondo stadio della pipeline.

Per la gestione delle criticità sui salti, viene adottata la tecnica del salto ritardato.

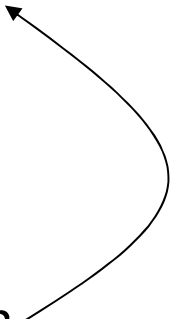
Quanti slot di ritardo sono presenti? Perché?

Si indichi, motivando brevemente la soluzione proposta, come potrebbe essere riordinato il codice per gestire la criticità sul salto beq.

[6]

Secondo stadio $\Rightarrow$ uno slot di ritardo

```
lw $s0, 20($t1)
beq $s0, $s1, Ndb
add $t2, $t2, $t2
Ndb: add $t2, $t2, $t3
sub $s2, $s2, $s3
```



Si vede che la

```
sub $s2, $s2, $s3
```

viene eseguita a prescindere dall'esito del salto; possiamo allora anticiparla nello slot, cosicché viene eseguita subito. Ciò è lecito perché non altera i sorgenti delle istruzioni add né queste ultime alterano i sorgenti della sub.

Quindi la soluzione è

```
lw $s0, 20($t1)
```

```
beq $s0, $s1, Ndb
```

```
sub $s2, $s2, $s3
```

```
add $t2, $t2, $t2
```

```
Ndb:  add $t2, $t2, $t3
```

Esercizio – predizione statica di “salto non eseguito”

Parte 1

Si consideri il codice assembler:

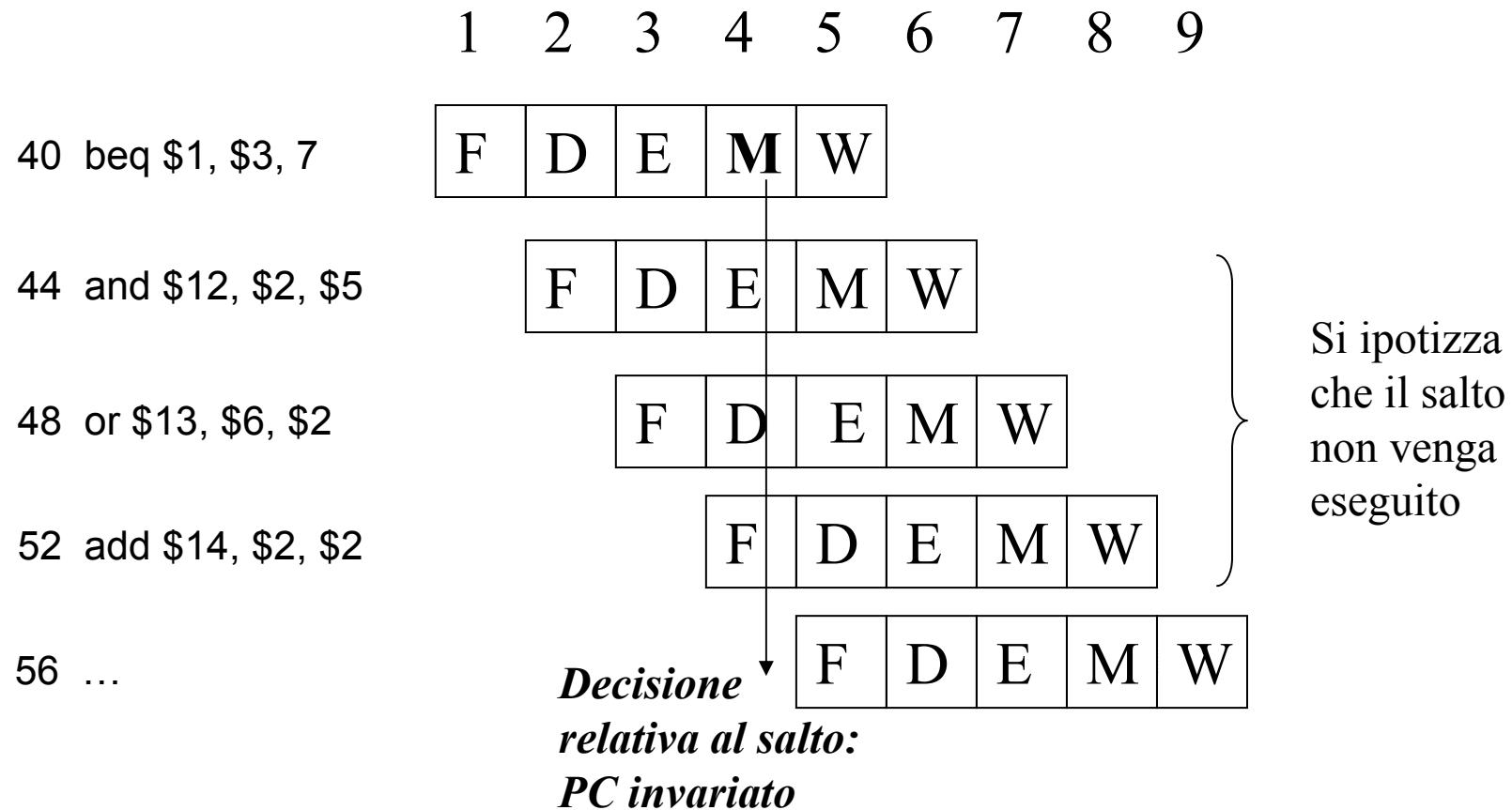
40	beq	\$1, \$3, 7	# salto a: $40 + 4 + 7 \cdot 4 = 72$
44	and	\$12, \$2, \$5	
48	or	\$13, \$2, \$6	
52	add	\$14, \$4, \$2	
...			
72	lw	\$4, 50(\$7)	

- Si consideri la tecnica di previsione statica “ipotesi di salto non eseguito”.
- Si consideri una pipeline a 5 stadi (F, D, E, M, W) in cui la decisione del salto (e l’aggiornamento del PC) si effettuano nello stadio M:

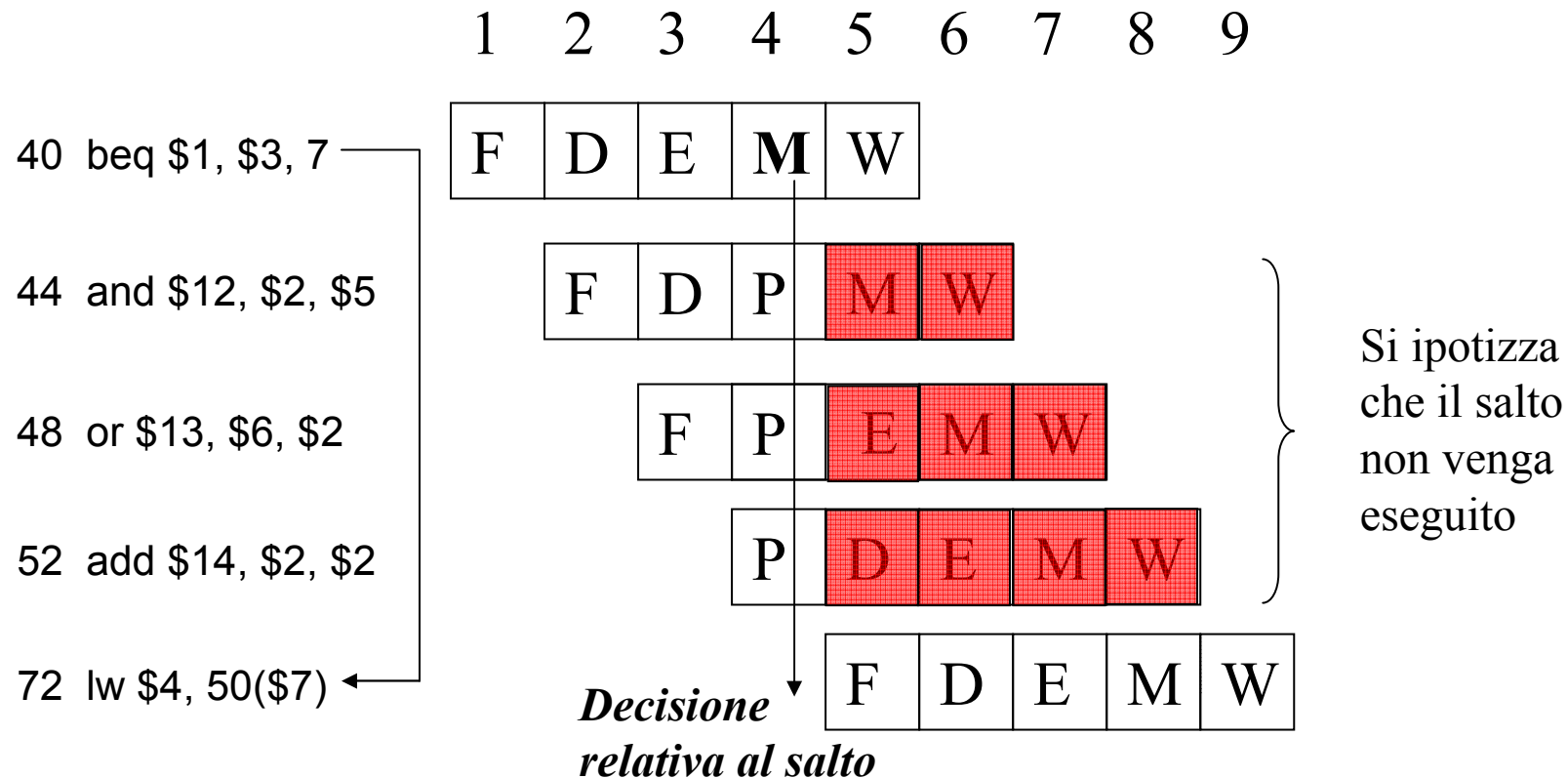
➡ Tracciare il diagramma temporale nell’ipotesi di previsione corretta e nell’ipotesi di previsione errata; indicare in questo caso il numero di cicli di ritardo.

➡ Ripetere esercizio considerando salto in stadio D (al posto di M)

Se il salto non viene effettuato [previsione corretta] tutto procede normalmente:



Se il salto viene effettuato [previsione errata] nel ciclo di clock 4 è aggiornato PC
[caricamento nuova istruzione di destinazione al ciclo di clock successivo]

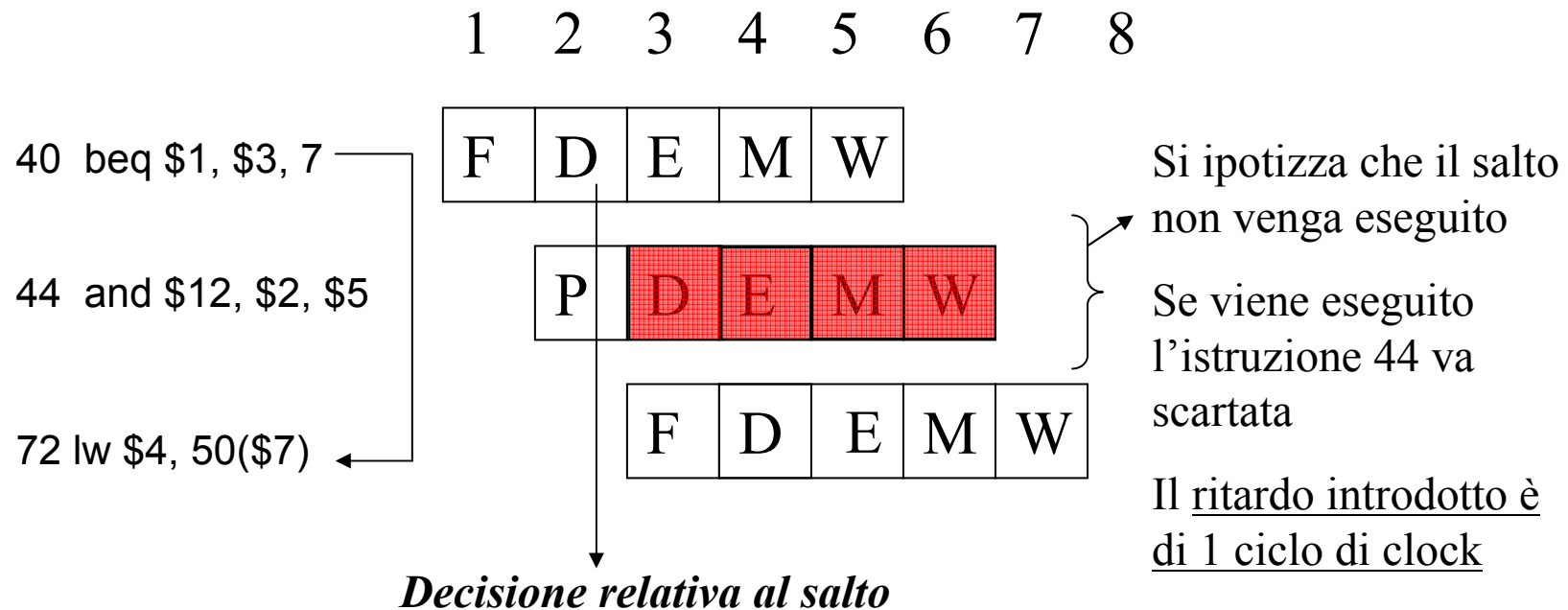


Se il salto viene effettuato [previsione errata], le istruzioni 44, 48, 52 vanno scartate, si introduce un ritardo di 3 cicli di clock.

In rosso sono indicate le bolle create nel ciclo di clock 4

- Spostando l'esecuzione dei salti dallo stadio M allo stadio D si riduce il ritardo a un solo ciclo:

nell'esempio se si verifica il salto, solamente l'istruzione 44 (di cui è stato fatto il fetch) va scartata



Parte 2

- Si consideri la seguente combinazione di istruzioni
 - 22% load, 11% store, 49% formato-R, 16% salti condizionati, 2% salti incondizionati
- Per esecuzione con pipeline [per branch, previsione di “salto non eseguito”]:
 - a) metà delle istruzioni di load seguita da un’istruzione che ne utilizza il risultato (dipendenza dati, senza riordino)
 - b) un quarto dei salti condizionati sono oggetto di previsione errata
 - c) ritardo causato dai salti incondizionati pari a 1 ciclo di clock (in questo caso si suppone che il salto venga effettuato nello stadio di decodifica e che si usi lo stallo)
- Tempo di ciclo: 2 ns (per l’unità funzionale più lenta)
 -  Confrontare le prestazioni relative ad uno schema multiciclo e ad uno schema con pipeline [considerando le due ipotesi di esecuzione dei salti, ovvero in M o D]

Al solito, per il multiciclo:

Numero di cicli per ciascuna classe di istruzioni (fig. 5.42)

Load = 5

Store = 4

Formato-R = 4

Salti cond. = 3

Salti incond. = 3

$$\text{CPI} = 0,22 \times 5 + 0,11 \times 4 + 0,49 \times 4 + 0,16 \times 3 + 0,02 \times 3 = 4,04$$

$$\text{Tempo medio per istruzione} = 4,04 \times 2 \text{ ns} = \mathbf{8,08 \text{ ns}}$$

Per la pipeline (esecuzione salto nello stadio M: ritardo 3 cicli di clock):

Calcoliamo prima il numero di cicli di ritardo per ciascuna classe di istruzioni (nel caso di pipeline a regime):

$$\text{Load} = 0,22 \times 0,50 \times 1 = 0,11 \quad (\text{vedi (a)})$$

$$\text{Salti cond.} = 0,16 \times 0,25 \times n = 0,04 \times n \quad (\text{vedi (b)})$$

$$\text{Salti incond.} = 0,02 \times 1 = 0,02 \quad (\text{vedi (c)})$$

dove $n = 3$ [esecuzione salti cond. in stadio M] o $n=1$ [in stadio D]

$$\delta_M = 0,11 + 0,12 + 0,02 = 0,25$$

$$\delta_D = 0,11 + 0,04 + 0,02 = 0,17$$

Quindi:

$$\text{CPI}_M = 1 + \delta_M = 1,25$$

$$\text{Tempo medio esecuz. } 1,25 \times 2\text{ns} = 2,5\text{ns}$$

$$\text{CPI}_D = 1 + \delta_D = 1,17$$

$$\text{Tempo medio esecuz. } 1,17 \times 2\text{ns} = 2,34\text{ns}$$

Confronto prestazioni pipeline vs. multiciclo: speedup

NB: facciamo riferimento alla soluzione migliore – esecuzione salto in stadio D
[per l'altro caso il calcoli sono analoghi]

L'incremento delle prestazioni (o speedup) del processore con pipeline rispetto al processore multi-ciclo è dato da:

$$\text{Speedup} = \frac{\text{Tempo}_{\text{multiciclo}}}{\text{Tempo}_{\text{pipeline}}} = \frac{8,08}{2,34} = 3,45$$

Il tempo con pipeline ideale sarebbe 2 ns e quindi lo speedup nel caso ideale sarebbe

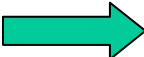
$$\text{Speedup} = \frac{\text{Tempo}_{\text{multiciclo}}}{\text{Tempo}_{\text{pipeline-ideale}}} = \frac{8,08}{2} = 4,04 (> 3,45)$$

Esercizio – predizione statica, stalli e dipendenze

Si consideri il codice assembler MIPS:

```
        add    $t1, $s1, $s2
        lw     $t0, 20($s4)
        beq    $t0, $t1, Dest
        sub    $s1, $s2, $s3
        ...
Dest:   add    $t1, $t0, $t0
```

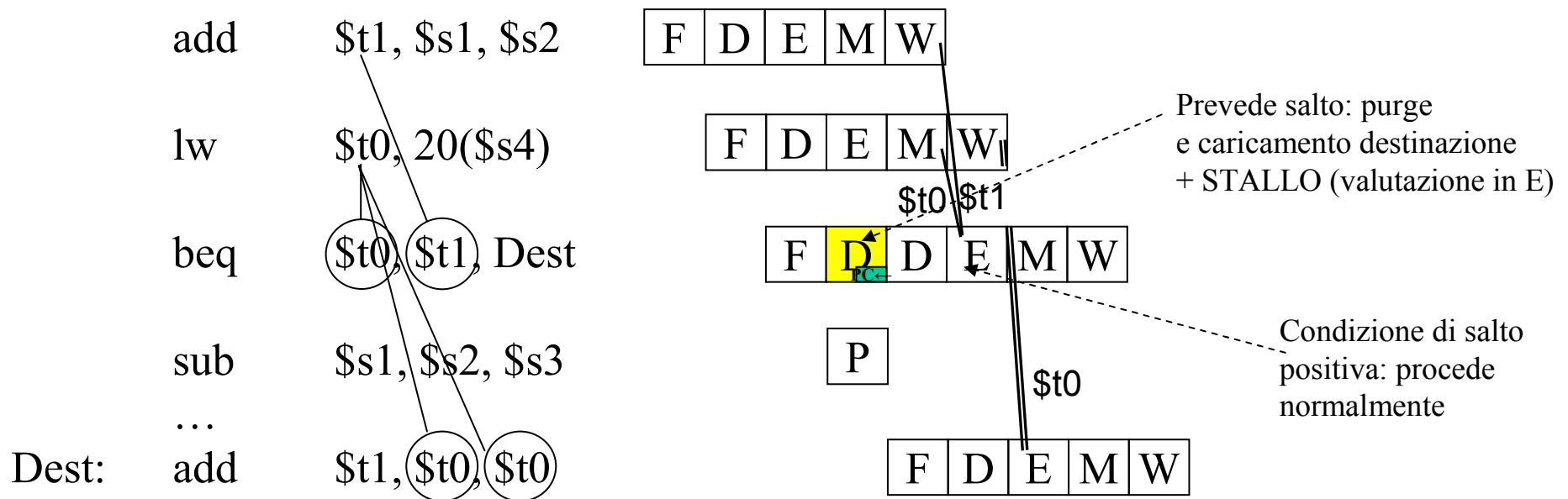
- Si consideri una pipeline a 5 stadi (F, D, E, M, W) in cui l'indirizzo di destinazione dei salti è calcolato nello stadio D, la valutazione della condizione di salto (e l'aggiornamento di PC) nello stadio E
- Per i salti condizionati, si utilizza la predizione statica demandata al compilatore; in questo caso, il compilatore ha previsto che l'istruzione beq salti effettivamente

 Tracciare il diagramma temporale nell'ipotesi di previsione corretta e nell'ipotesi di previsione errata; indicare in entrambi i casi il numero di cicli di ritardo.

Prima cosa: non dimentichiamoci delle dipendenze sui dati!

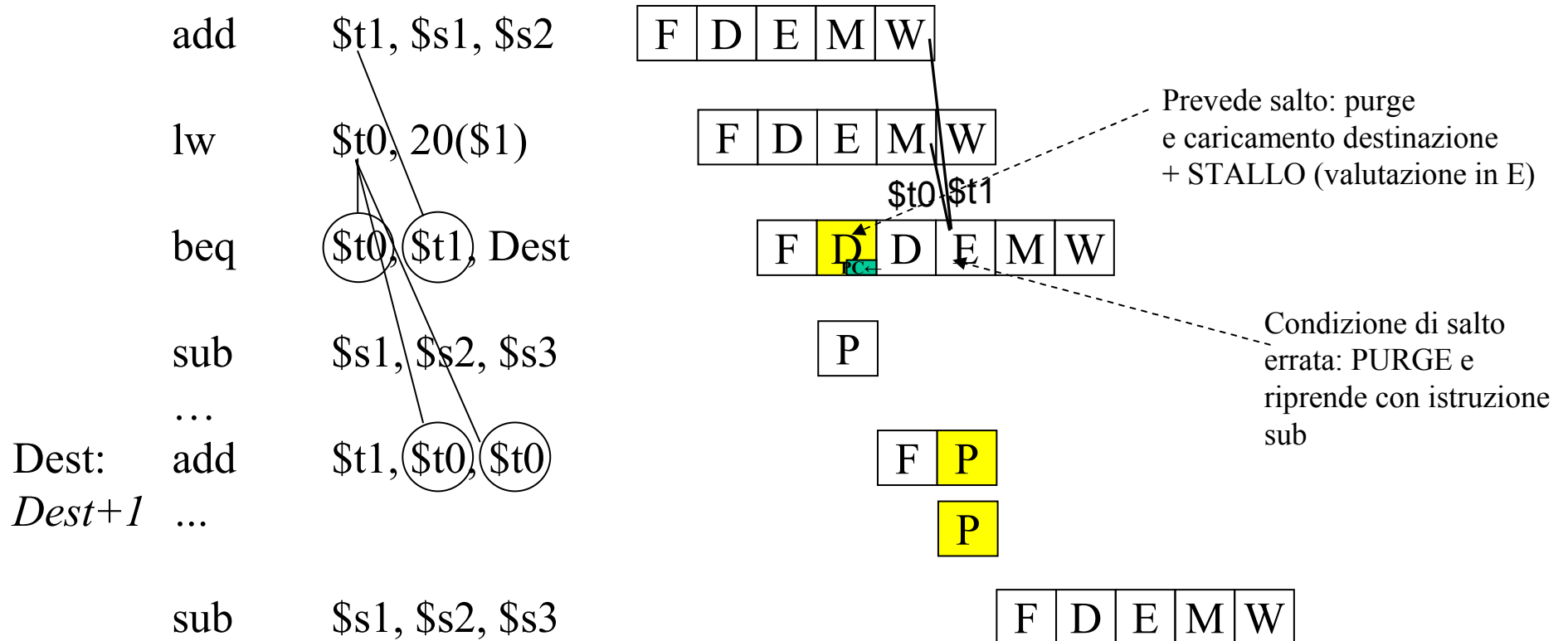
beq: calcola destinazione in D, valuta operandi in E

Caso “predizione di salto corretta”



In totale, ho penalità di un solo ciclo!

Caso “predizione di salto errata”



In totale, ho penalità di tre cicli
(1 per stallo dipendenza dati e 2 persi per la predizione errata)

Esercizio – Prestazioni e predizione dinamica

Si consideri una pipeline a 5 stadi in cui la valutazione della condizione di salto e l'aggiornamento del Program Counter avvengono nel terzo stadio (EX).

Si valutino le due seguenti soluzioni progettuali per la gestione dei salti condizionati:

- Predizione statica di “salto non effettuato”
- Predizione dinamica che utilizza un BTB in grado di fornire un indirizzo di destinazione sempre corretto ed un'accuratezza nella predizione della condizione di salto pari al 90%

In particolare, detta P la percentuale con cui i salti condizionati in media effettuano il salto, si valuti quale delle due soluzioni è preferibile in funzione di P .

Soluzione 1 “predizione salto non effettuato”

- Salto non effettuato (probabilità $1-P$):
Penalità 0
- Salto effettuato (probabilità P):
Penalità 2 [aggiornamento PC e “purge” al terzo stadio:
elimino due istruzioni]

$$\rightarrow \delta_{\text{miss}}^1 = P * 2$$

Soluzione 2 “predizione dinamica”

- Predizione corretta (probabilità 90%):
Penalità 0 [infatti si utilizza direttamente l’indirizzo
di destinazione corretto]
- Predizione errata (probabilità 10%):
Penalità 2 [aggiornamento PC e “purge” al terzo stadio:
elimino due istruzioni]

$$\rightarrow \delta_{\text{miss}}^2 = 0,1 * 2 = 0,2$$

Confronto

Preferibile la soluzione 1 se $P * 2 < 0,2$ ovvero salti eseguiti con frequenza ₁₀₀
inferiore al 10%