Microarchitetture ad Alte Prestazioni

Michele Favalli

Metriche per le prestazioni

- Latenza: intervallo di tempo che intercorre fra l'inizio e la fine nell'esecuzione di un certo task
  - latenza di un istruzione
  - latenza di un programma
- Throughput: numero di istruzioni elaborato nell'unità di tempo

Metriche per le prestazioni

- Latenza di un programma: tempo impiegato per eseguirlo
  - in generale si può calcolare come:

$$L = \sum_{i \in exec.instr.} CPI_i * T_{clock}$$

  - nella macchina a ciclo singolo si ha

$$L = \#exec.instr.* T_{clock}$$

- Throughput misurato eseguendo un programma:

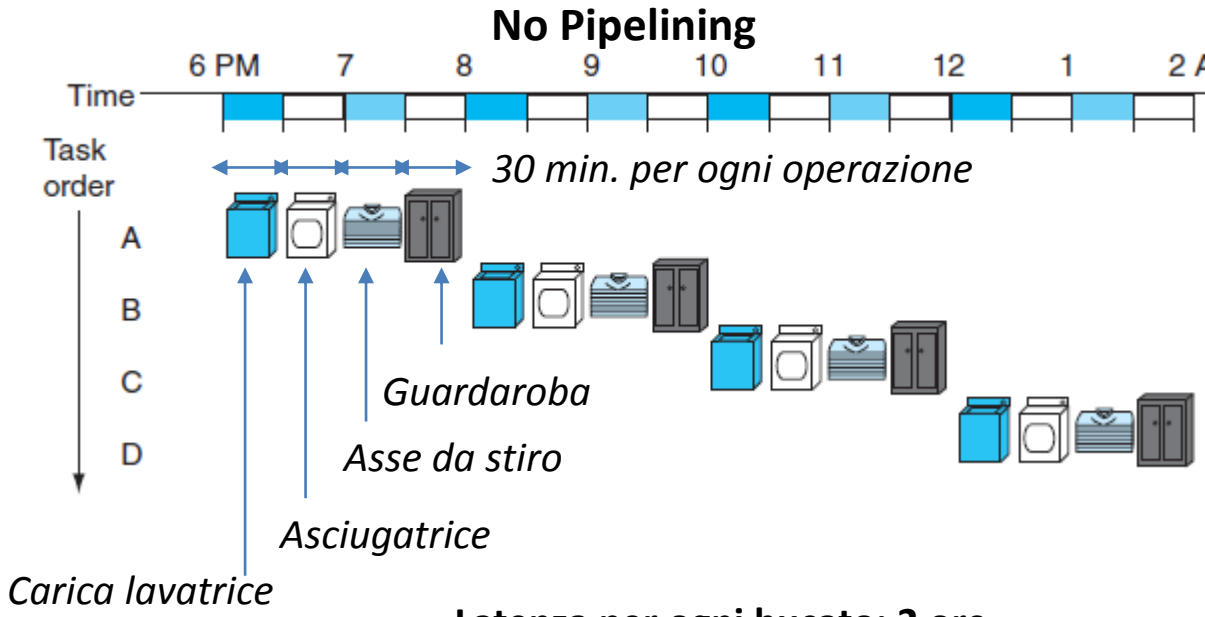
$$thr = \frac{\#exec.instr.}{L}$$

  - nella macchina a ciclo singolo si ha

$$thr = \frac{1}{T_{clock}} = f_{clock}$$

Pipelining

- **PIPELINING.** Tecnica implementativa in cui l'esecuzione di istruzioni multiple viene sovrapposta nel tempo, sul modello di una «catena di montaggio».
- SCOPO: **Migliorare il throughput, non la latenza delle singole operazioni!***



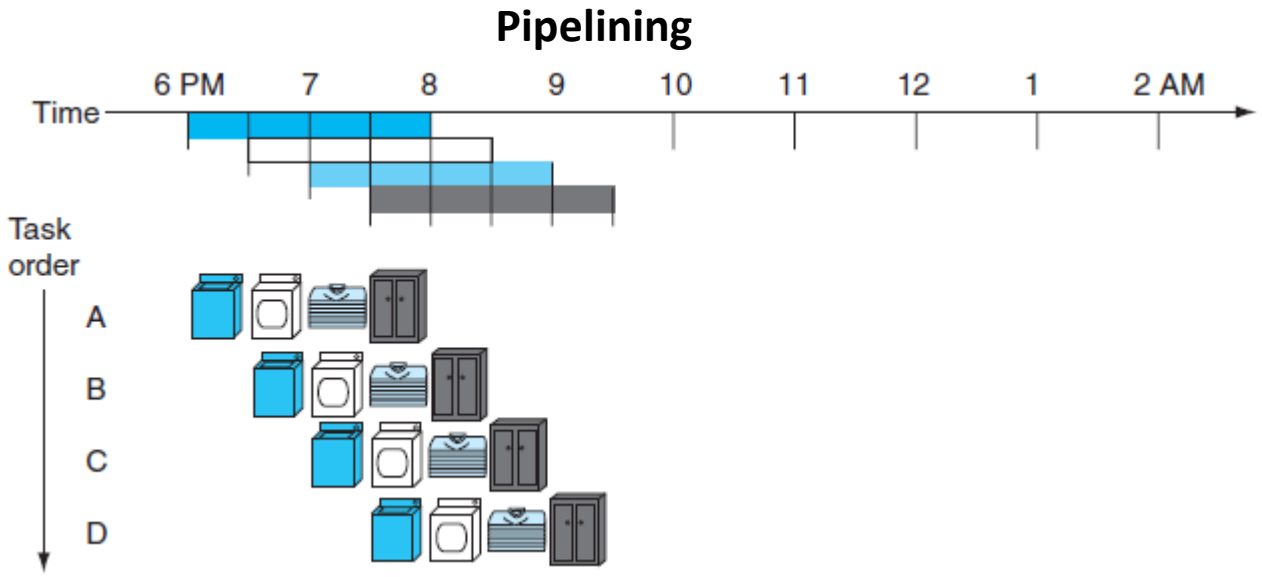
Latenza per ogni bucato: 2 ore

Throughput di bucati: 1 completato ogni 2 ore.

Pipelining

- **PIPELINING.** Tecnica implementativa in cui l'esecuzione di istruzioni multiple viene sovrapposta nel tempo, sul modello di una «catena di montaggio».

Una pipeline a 4 stadi è potenzialmente 4 volte più veloce rispetto alla versione base (ATTENZIONE, E' UNO SPEED-UP DI CASO MIGLIORE!)



Latenza per ogni bucato: 2 ore*

Throughput di bucati: 1 completato ogni 30 minuti.*

Una pipeline a N stadi è potenzialmente N volte più veloce rispetto alla versione base

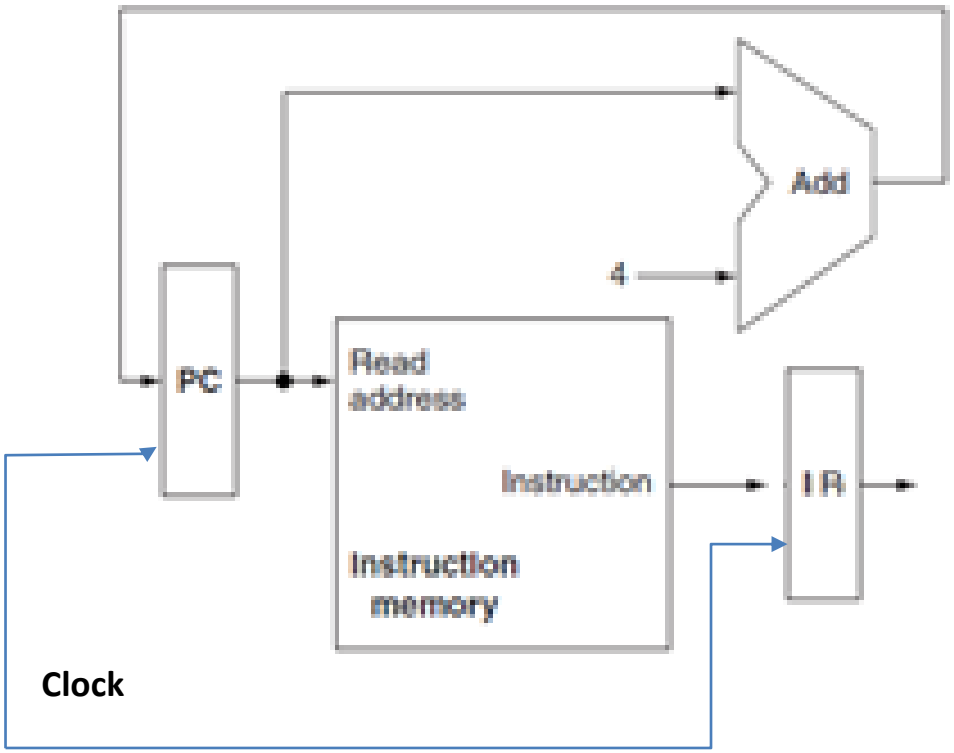
Assunzioni:

- La pipeline è bilanciata (ogni stadio impiega lo stesso tempo)!
- C’è lavoro sufficiente per tenere la pipeline sempre piena!
- Si trascurano il transitorio iniziale e quello finale

CPU pipelined

- **L'esecuzione di un istruzione deve essere spezzata in fasi che utilizzano risorse hardware indipendenti fra di loro e indipendenti dal tipo di istruzione**
- Se nel bucato avessi avuto una singola lava/asciuga, non avrei potuto fare una piepline a 4 stadi
- IN una CPU per rendere indipendenti le risorse (ALU ad esempio) fra loro devo introdurre dei registri

Stadio n. 1 della piepeline: Instruction Fetch

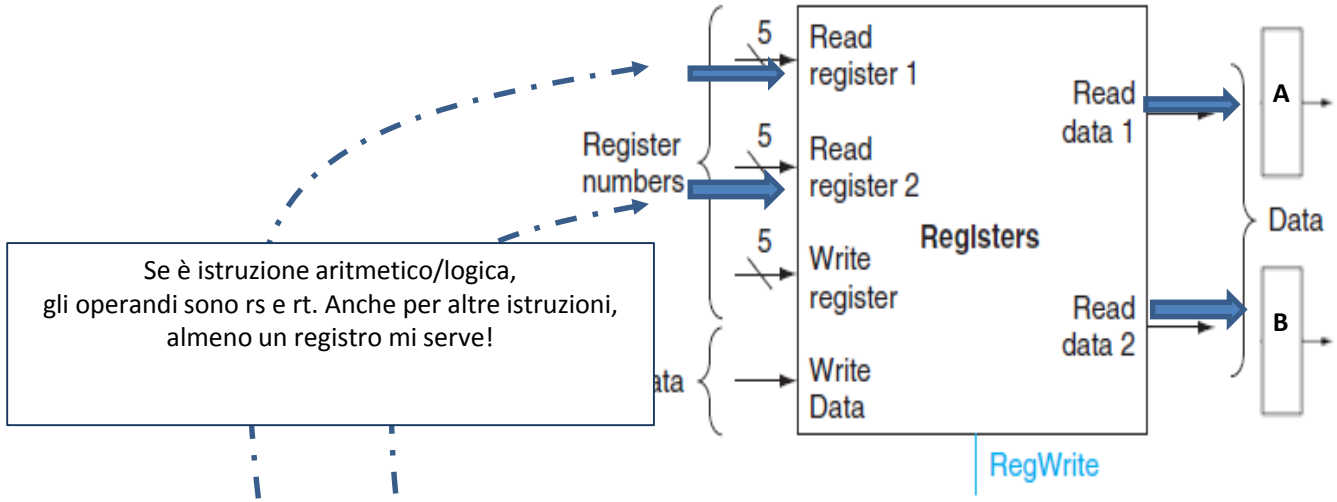


Qual'è il «critical path»?

Instruction Register = Memory [ProgramCounter]

PC <= PC + 4

Stadio n. 2 della pipeline: Register Fetch e Instruction Decode

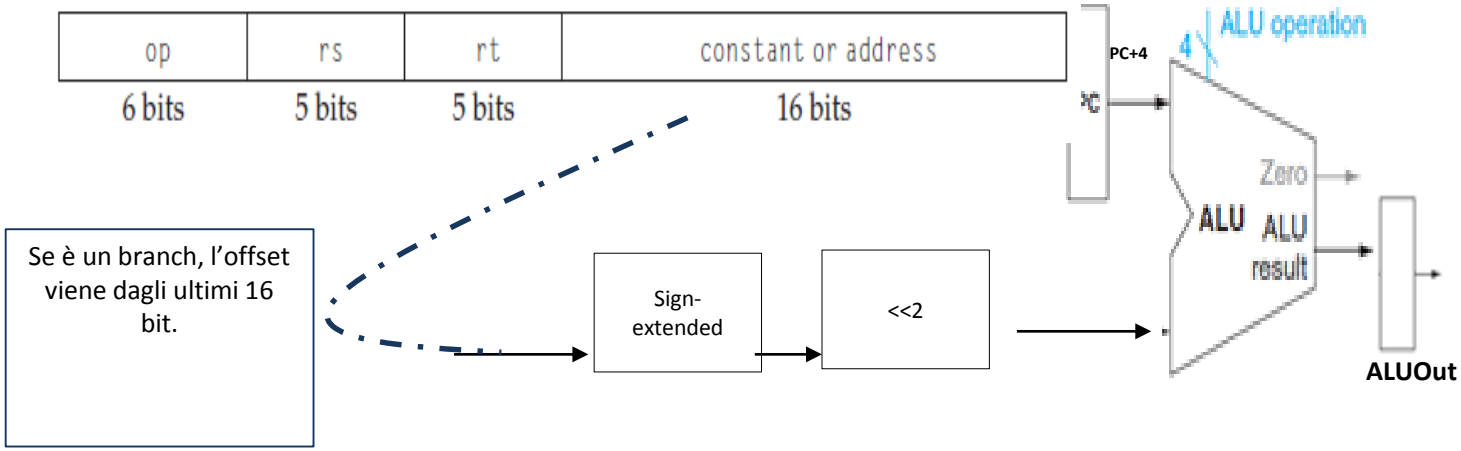


A <= Reg[registro operando rs]

B <= Reg[registro operando rt]

A e B sono da usare negli stadi successivi. E se non serviranno tutti perché non è una istruzione aritmetico-logica? Pazienza, non ho rovinato nulla!

Stadio n. 2 della pipeline: Register Fetch **e Instruction Decode**



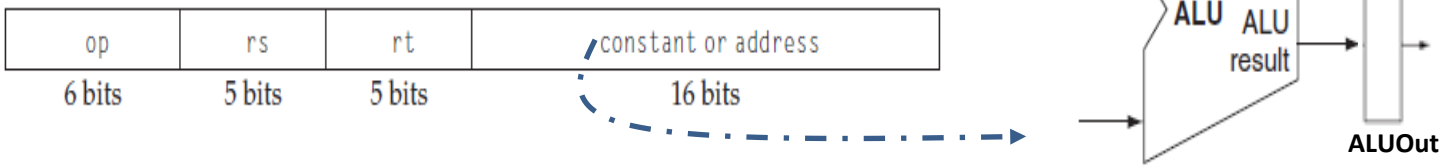
Branch target address <= PC + 4 + offset (sign-esteso e moltiplicato per 4)

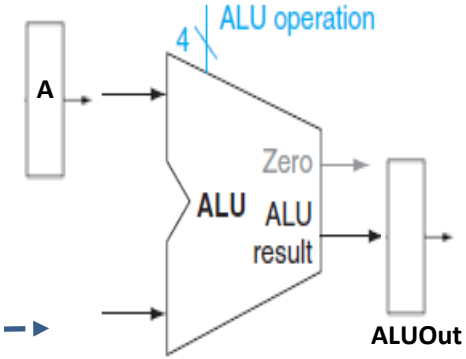
Note: PC+4 era stato calcolato nello stadio 1 e il BTA viene messo in ALUOut (nuovo registro)

Stadio n. 3 della pipeline: Execution

3a. Memory reference

Load/store



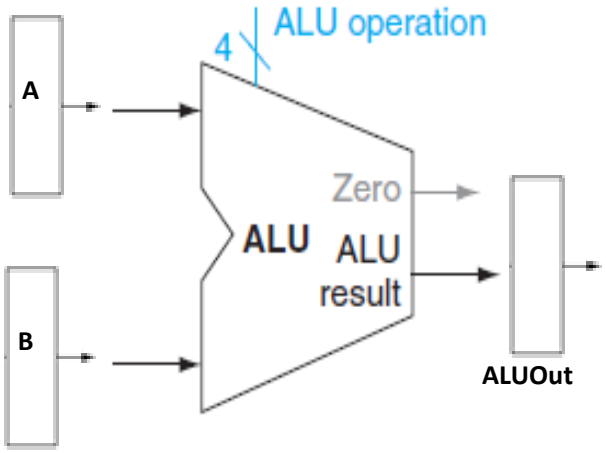


Calcolo indirizzo per l'accesso in memoria
ALUOut <= A + offset (sign-esteso)

Nota: A era stato scritto nello stadio 2, l'offset viene da IR e ALUOut viene sovrascritto rispetto al valore calcolato nello stadio 2

Stadio n. 3 della pipeline: Execution

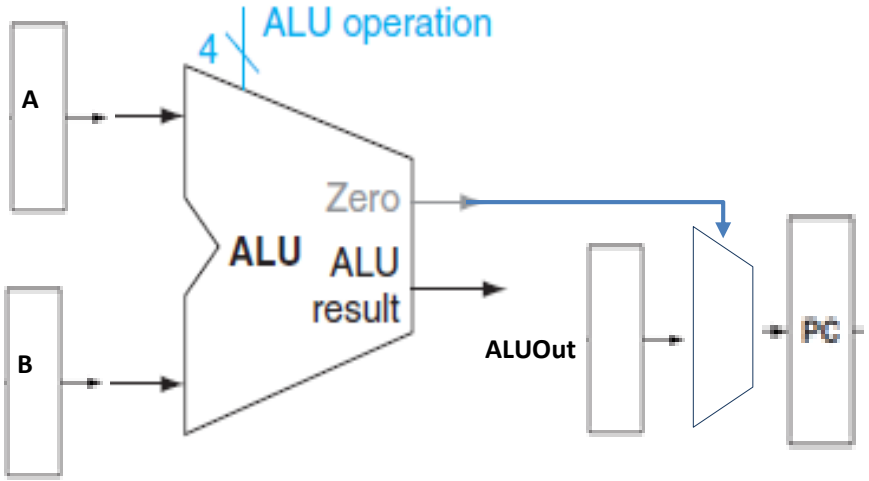
3b. Istruzione aritmetico-logica (tipo R)



ALUOut <= A op B
ALUOut viene sovrascritto rispetto al valore calcolato nello stadio 2

Stadio n. 3 della pipeline: Execution

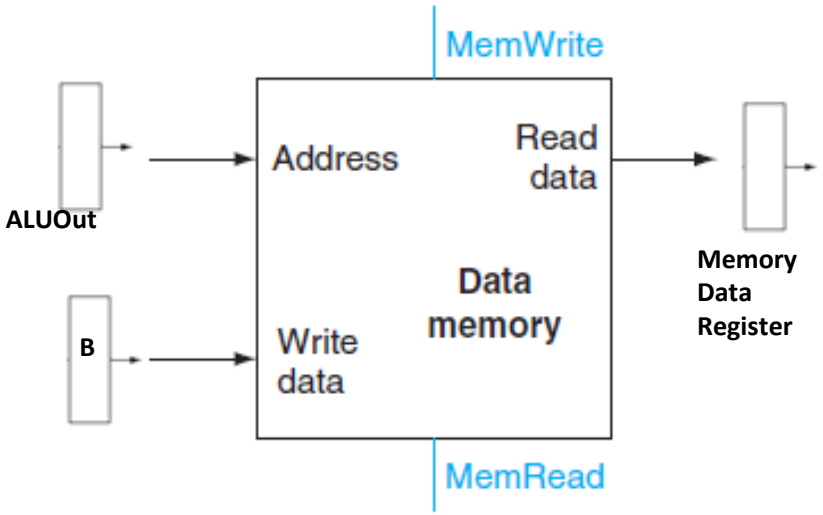
3c. Branch



If (A == B) PC <= ALUOut

ALUOut è stato calcolato al ciclo precedente

Stadio n. 4 della pipeline: Memory access



Load

MDR <= Memory [ALUOut]

Store

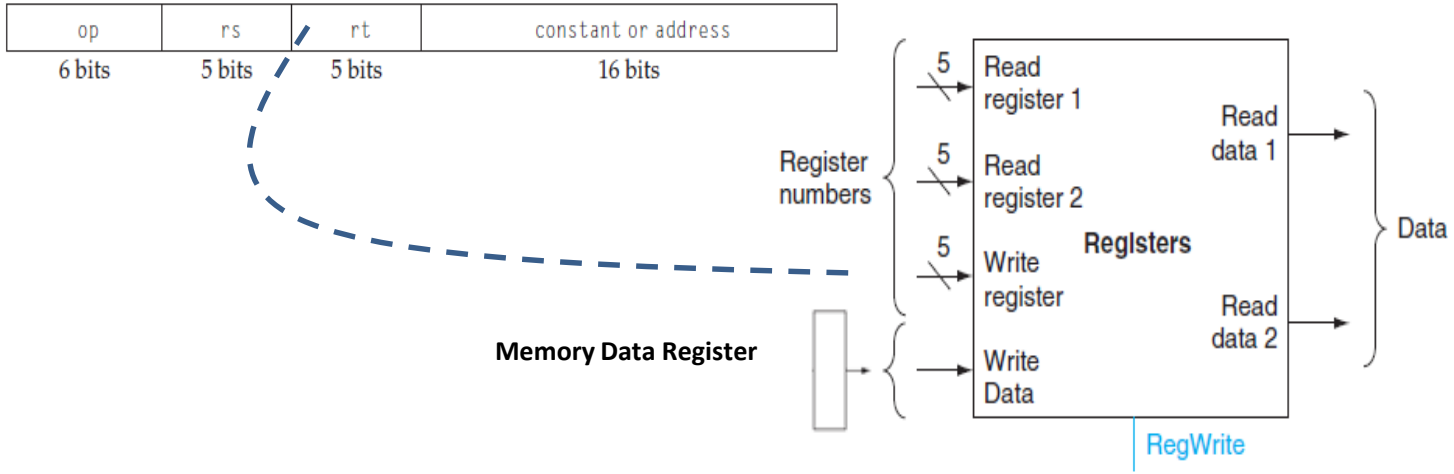
Memory [ALUOut] <= B

B disponibile dal decode stage e AluOut dal execute stage

STORE:

Stadio n. 5 della pipeline: Memory read completion (or Write-Back)

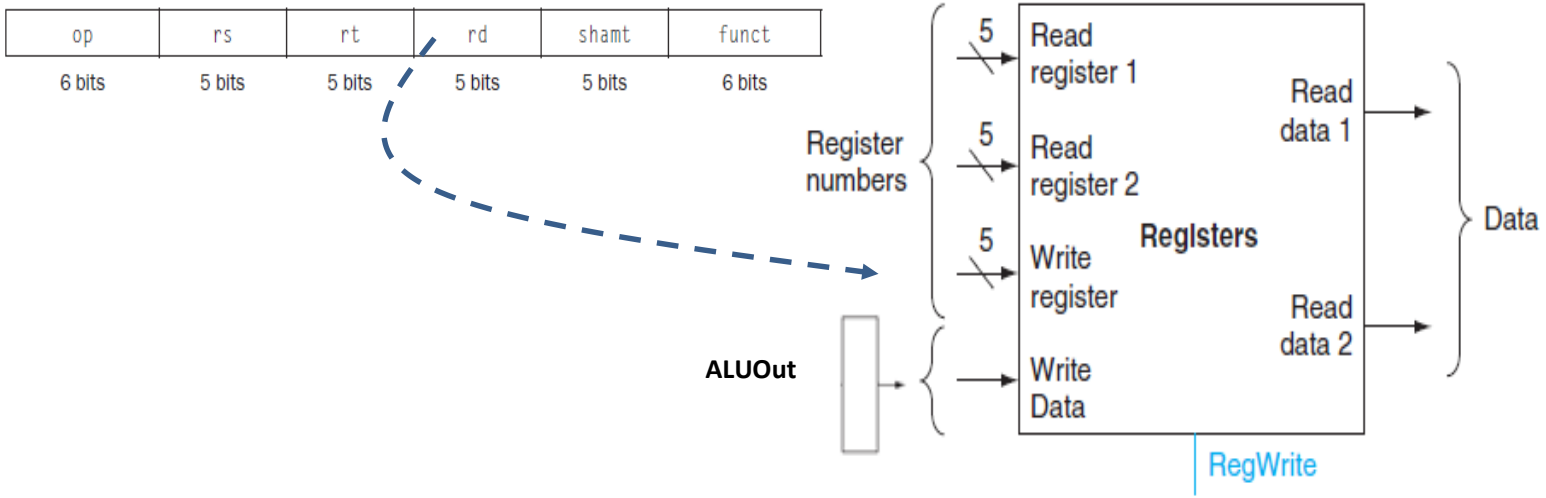
5a. Memory Reference



Registro destinazione[rt] <= MDR

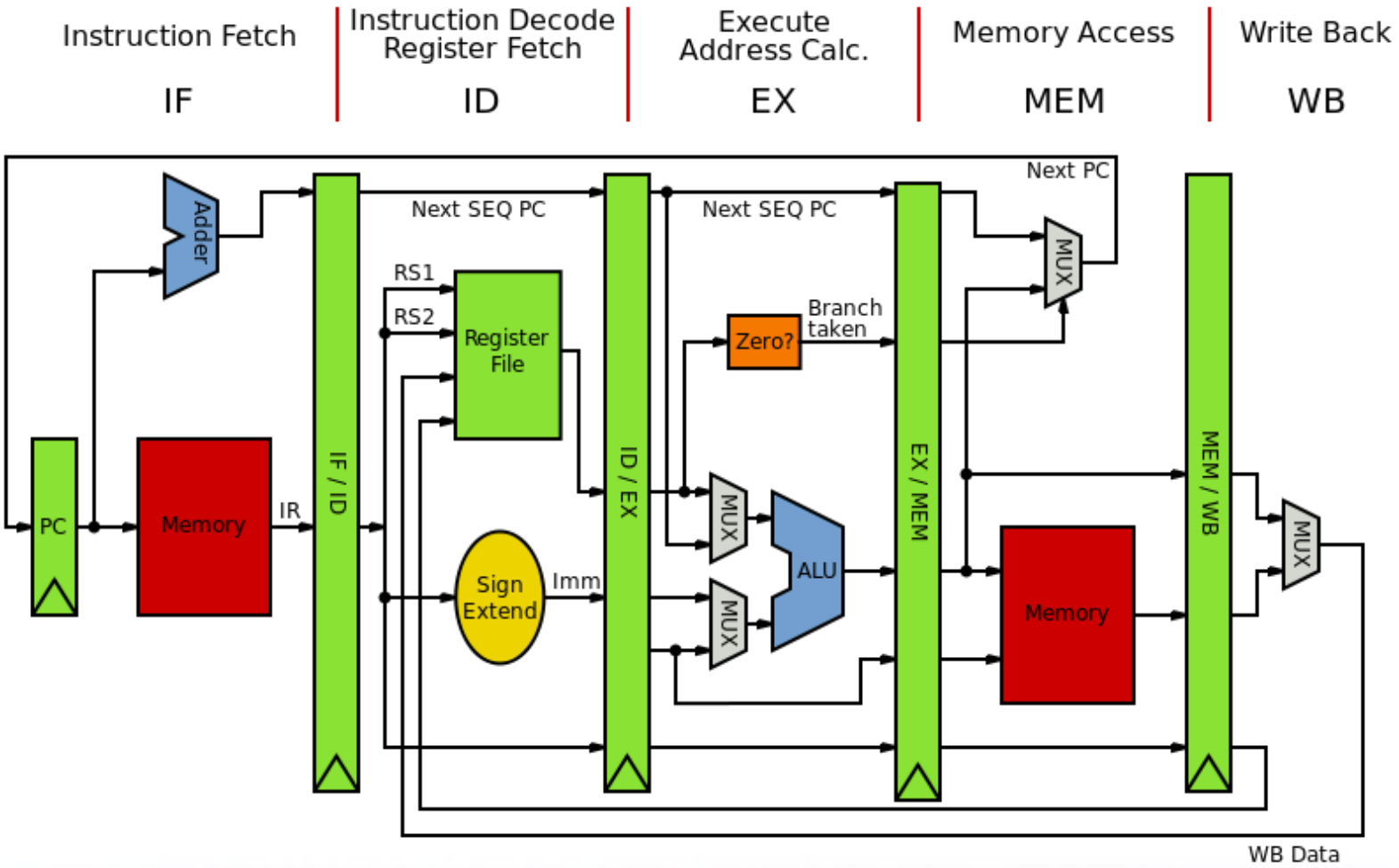
Stadio n. 5 della pipeline: Memory read completion (or Write-Back)

5b. Istruzione aritmetico-logica



Registro destinazione [rd] <=ALUOut

Schema complessivo



Sommario

Le istruzioni MIPS possono essere suddivise in 5 operazioni fondamentali (ottimizzate):

- **IF - Instruction Fetch**
- **ID - Instruction Decode and Register Reading**
- **EX - Execute or Address Calculation**
- **MEM - Memory access**
- **WB - Write Back in register file**

Vantaggi del pipelining

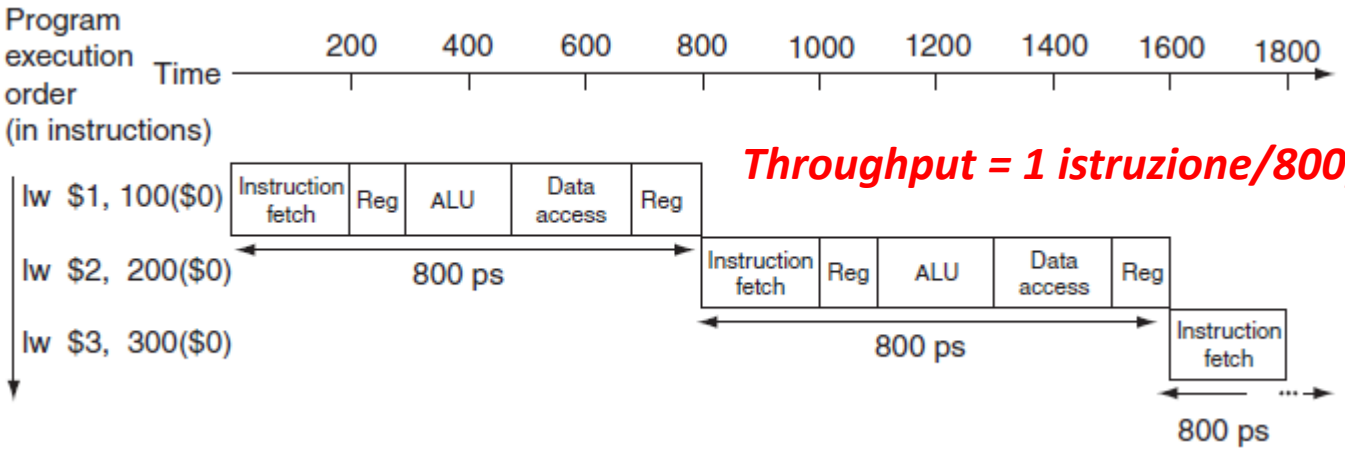
Restringiamo l'attenzione a sole 8 istruzioni per semplicità;
Assumiamo i seguenti tempi di esecuzione per le seguenti istruzioni:

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `and`, `or`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

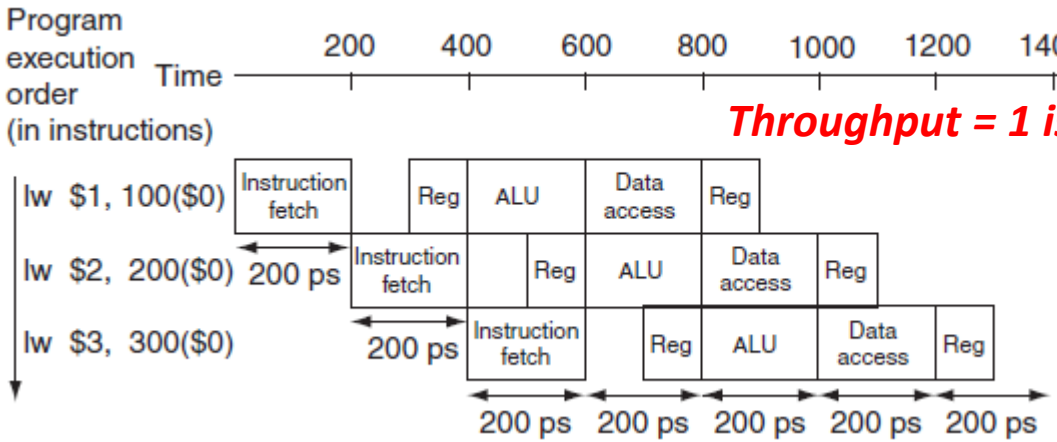
- In una implementazione a singolo ciclo, il periodo di clock sarebbe determinato **dall'istruzione più lenta**.
  - In questo caso, 800 ps
- Nel caso del pipelining, il periodo di clock sarebbe determinato **dall'operazione (stadio) più lenta**.
  - In questo caso, 200 ps

Single-Cycle vs. Pipelining

Single-Cycle:
Periodo di clock
a 800 ps



Pipelining:
Periodo di clock
a 200 ps



Nel pipelining, il «critical path» è di solito nell'esecuzione dell'ALU o nell'accesso in memoria.

Vantaggio per il throughput

- **Avevamo detto che……** Una pipeline ad N stadi aumenta il throughput potenzialmente di N volte …..

  …….quel «potenzialmente» è importante!

Slide precedente: **abbiamo introdotto 5 stadi di pipeline, dunque mi attendo un 5x di throughput**
Invece, lo speed-up per operazione è stato di 800ps/200ps=4! NON TORNA!
Spiegazione: la pipeline non era perfettamente bilanciata!

Slide precedente: **3 istruzioni eseguono in 2400ps (single-cycle) vs. 1400ps (pipelined)**
Lo speed-up in questo caso non è stato neppure di 4, ma di 1.7!
Spiegazione: 3 istruzioni sono poche, e si avverte ancora il ruolo del tempo di caricamento della pipeline. Se si considerano ad. es. 1M istruzioni, lo speed-up torna circa 4!

Metriche per le prestazioni nel caso pipelined

- La latenza delle singole istruzioni può aumentare se la pipeline non è bilanciata (ovvero divisa in stadi con uguale ritardo)
- La latenza nell'esecuzione di un programma migliora in maniera pari al numero di stadi della pipeline se questa è bilanciata, se il num. di istr. è molto grande
- Quello che migliora è la frequenza di clock

Metriche per le prestazioni nel caso pipelined

- Cosa succede alle formule che avevamo usato nel caso a ciclo singolo?
- Sotto le ipotesi restrittive che abbiamo fatto e altre che vedremo, rimane valido che la latenza è uguale a:

$$L = \#exec.instr.* T_{clock}$$

infatti viene terminata un istruzione a ogni ciclo di clock

- Quindi anche la formula del throughput misurato eseguendo un programma rimane valida:

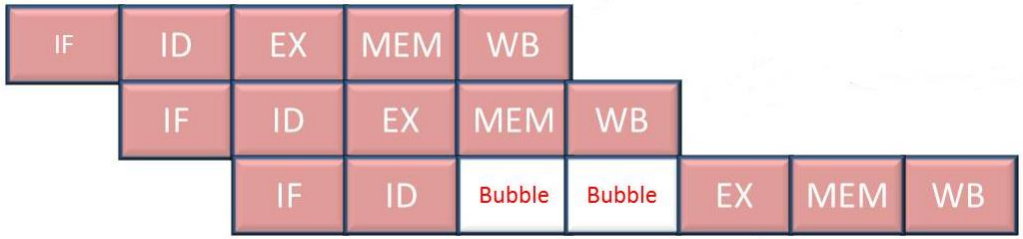
$$thr = \frac{\#exec.instr.}{L} = \frac{1}{T_{clock}} = f_{clock}$$

Problemi della pipeline

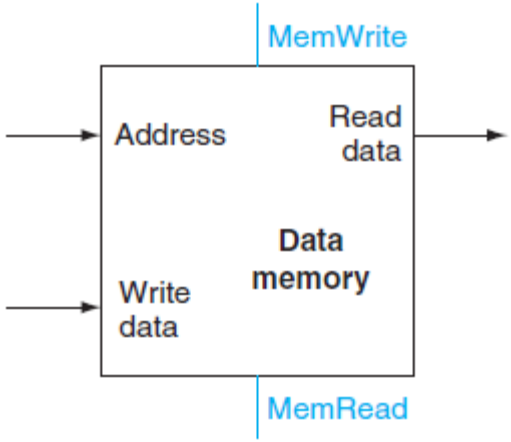
- Prima forse è meglio una pausa ….

Hazards

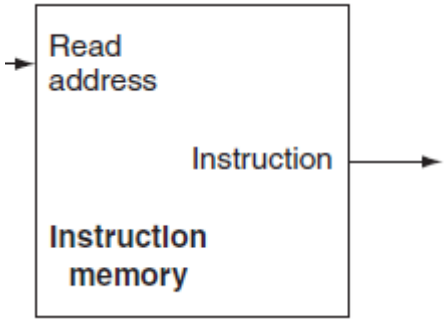
- **Ci sono situazioni col pipelining nelle quali la prossima istruzione non può essere eseguita nel ciclo di clock successivo (HAZARD)**
  - **Hazard Strutturali**
  - **Hazard Dati**
  - **Hazard di Controllo**
- **Penalità tipica introdotta dagli hazard: stalli della pipeline!**

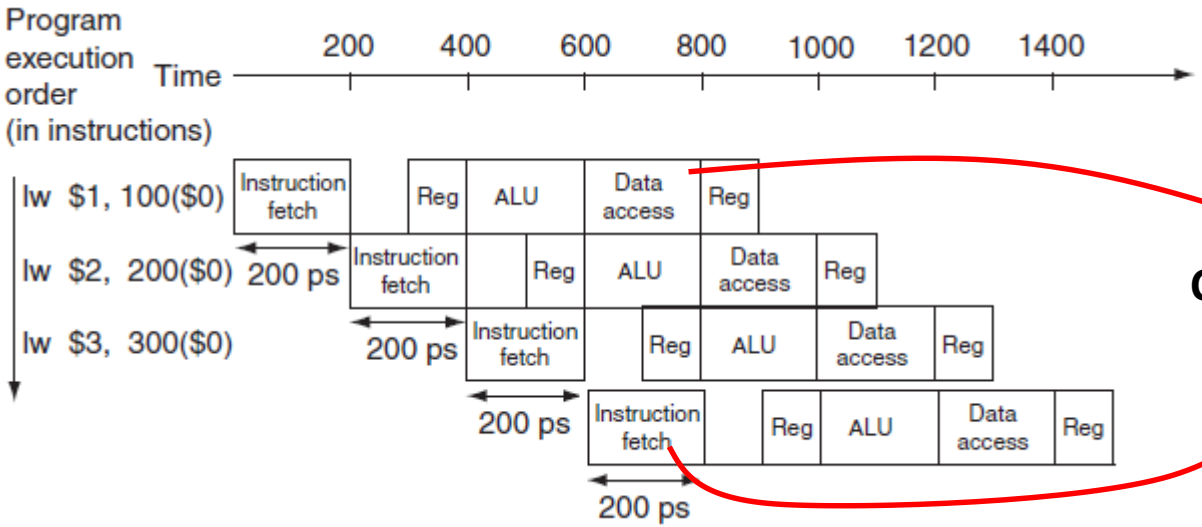


Hazard Strutturali (esempio)



Supponiamo che memoria dati e memoria istruzioni coincidano



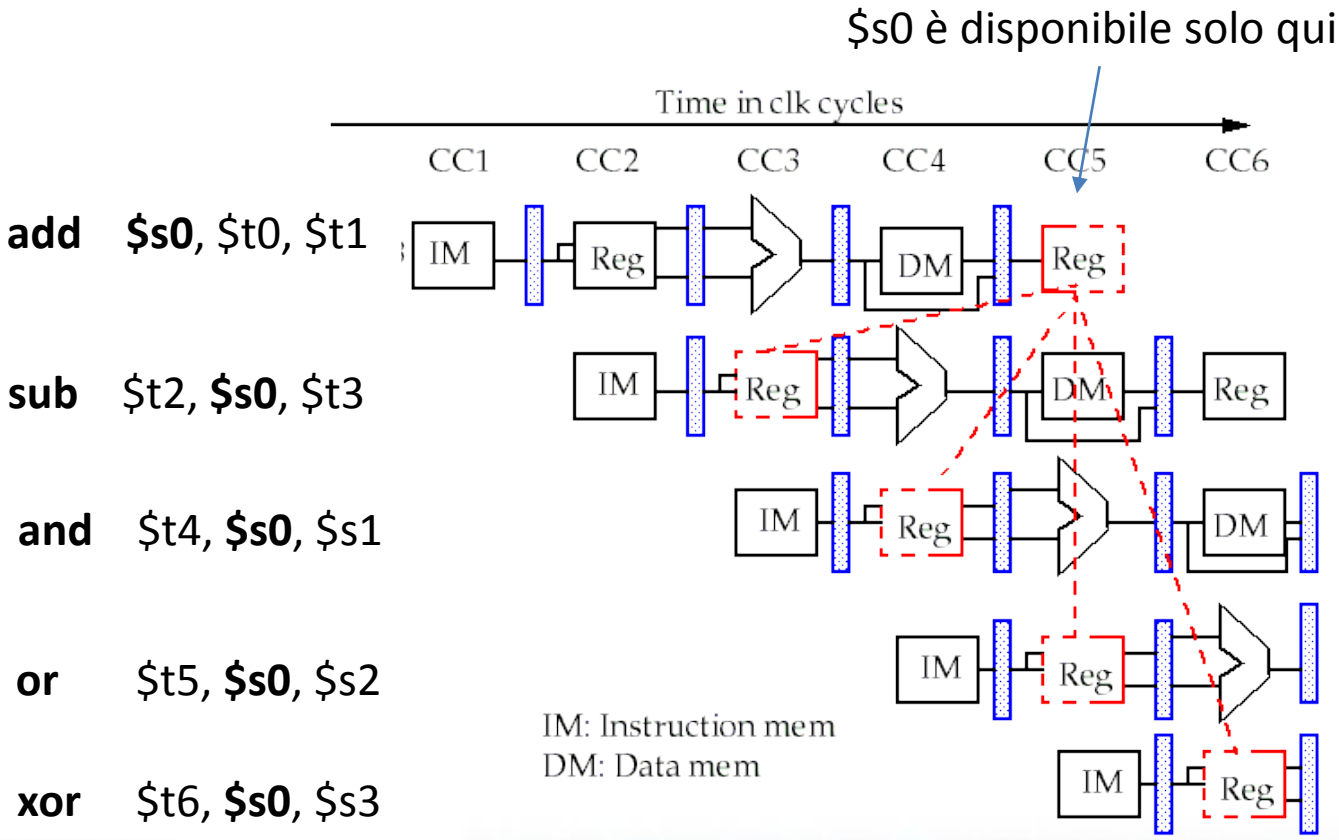


Conflitto sulla stessa memoria!

Ogni risorsa deve essere utilizzata in un solo stadio di pipeline!

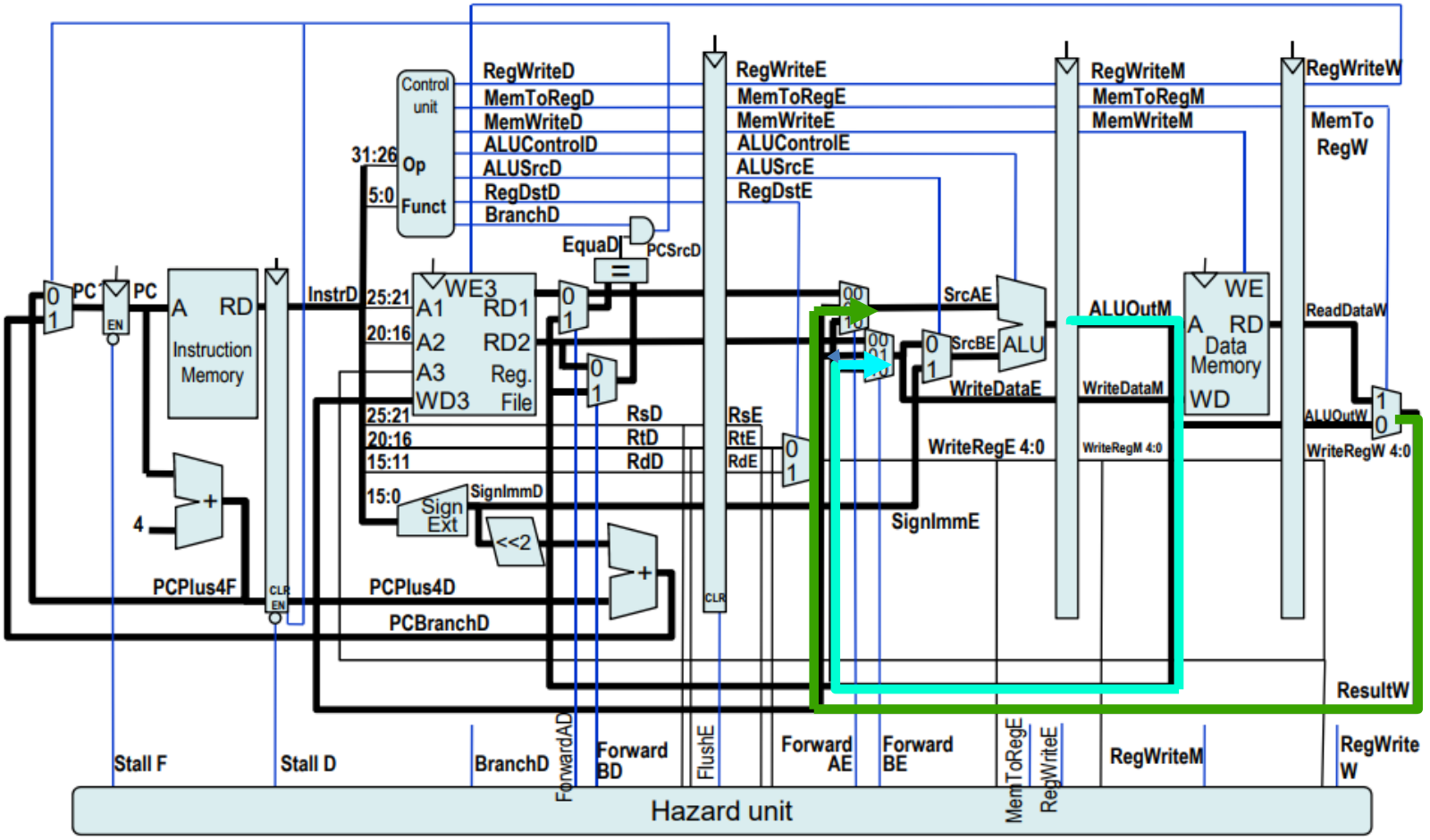
Hazard Dati – Il Problema

- La pipeline deve essere fermata perché uno stadio ha bisogno del completamento di un altro per poter eseguire correttamente



- Tutte le istruzioni dopo la ADD fanno uso del suo risultato
- Il write-back del registro $s0 avviene solo nell'ultimo stadio della pipeline
- Le operazioni SUB, AND accedono ad un valore errato del registro $s0
- Se il register file implementa «internal forwarding», l'istruzione OR legge il valore corretto.

Forwarding unit

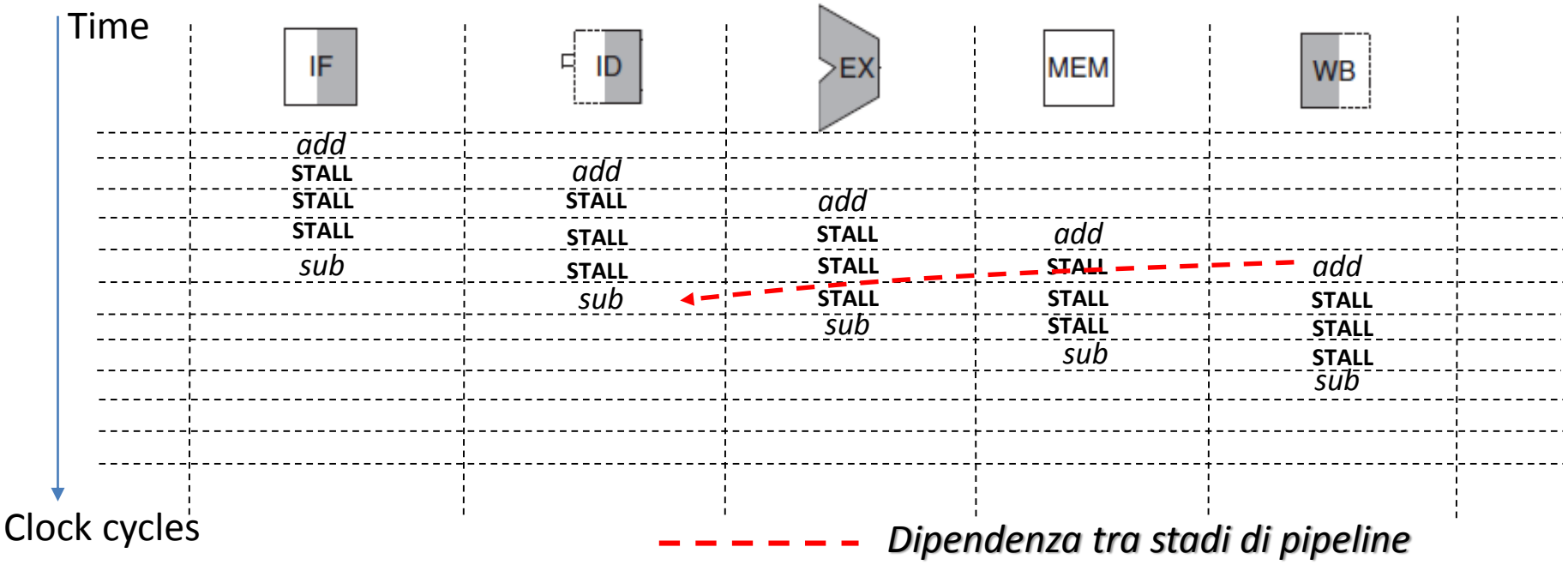


Hazard Dati - Implicazioni

- Gli hazard dati implicano «perdite di performance».

Implicazione del primo hazard:

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



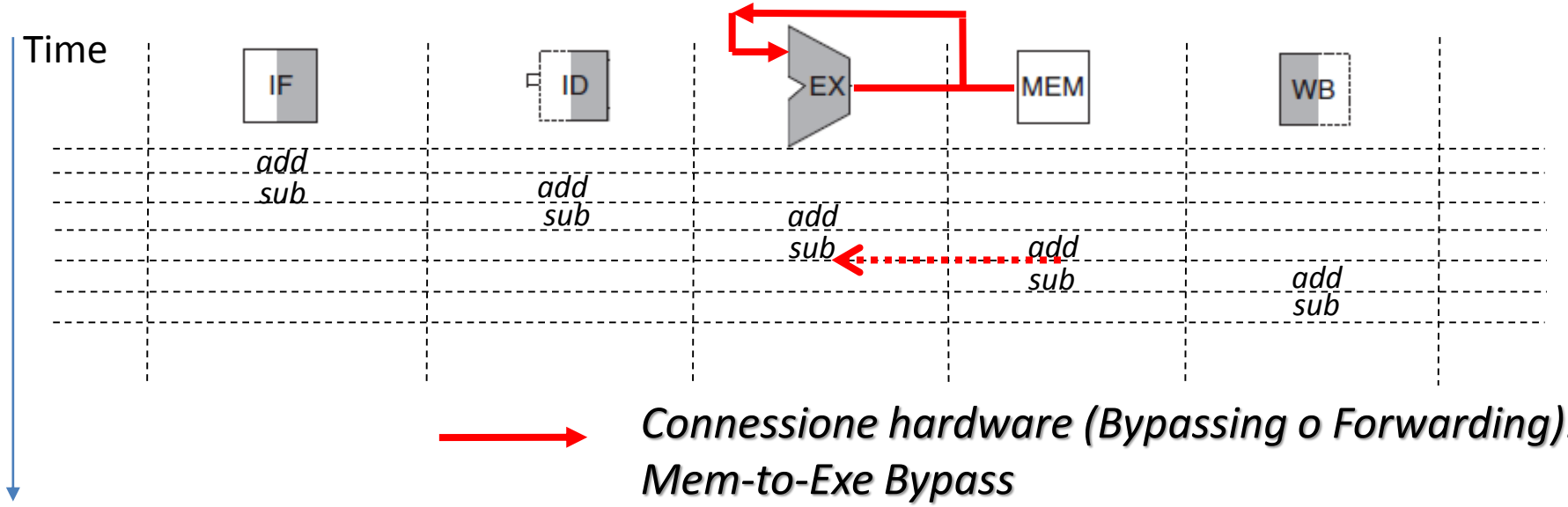
Abbiamo inserito tre «bolle» nella pipeline per evitare l'inconsistenza!

Hazard Dati – La soluzione

- Utilizziamo l'informazione appena è prodotta

Implicazione del primo hazard:

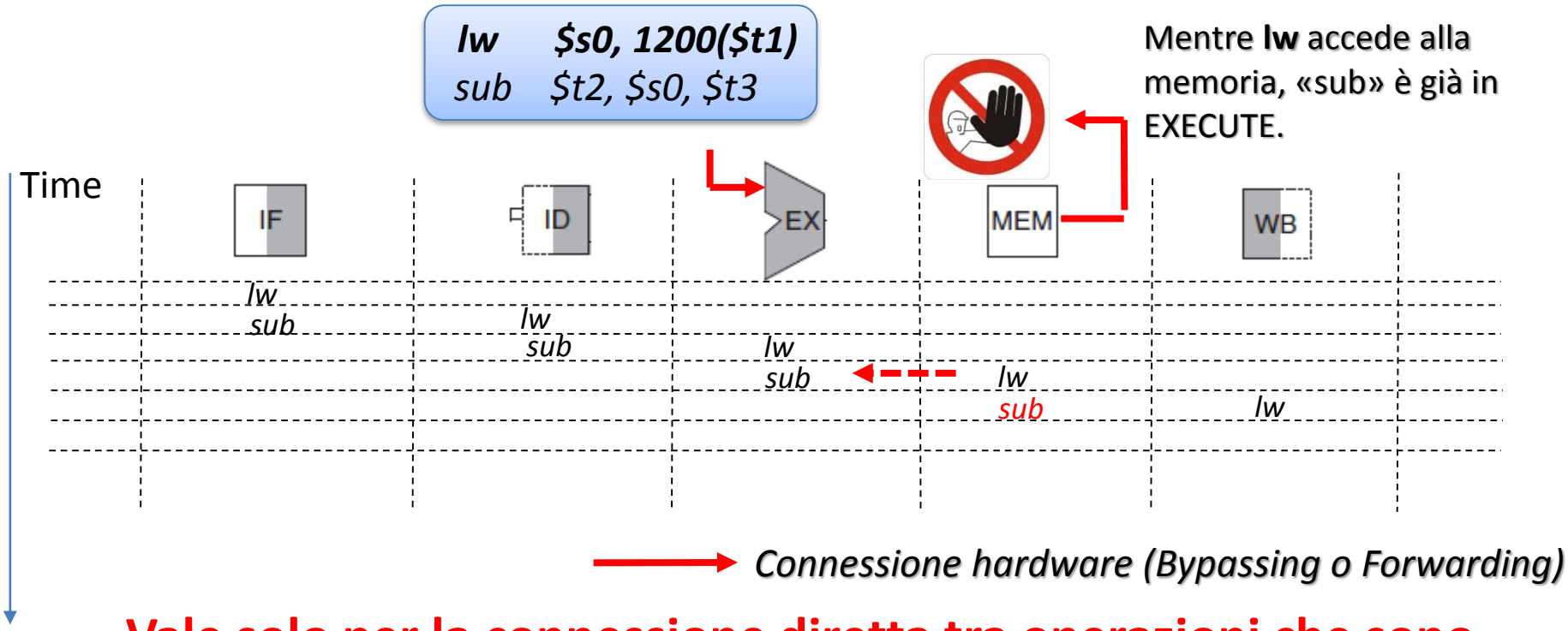
```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```



Non dobbiamo aspettare che il registro $s0 sia aggiornato, perché il suo valore futuro è disponibile direttamente all'ingresso dello stadio MEM.

Hazard Dati

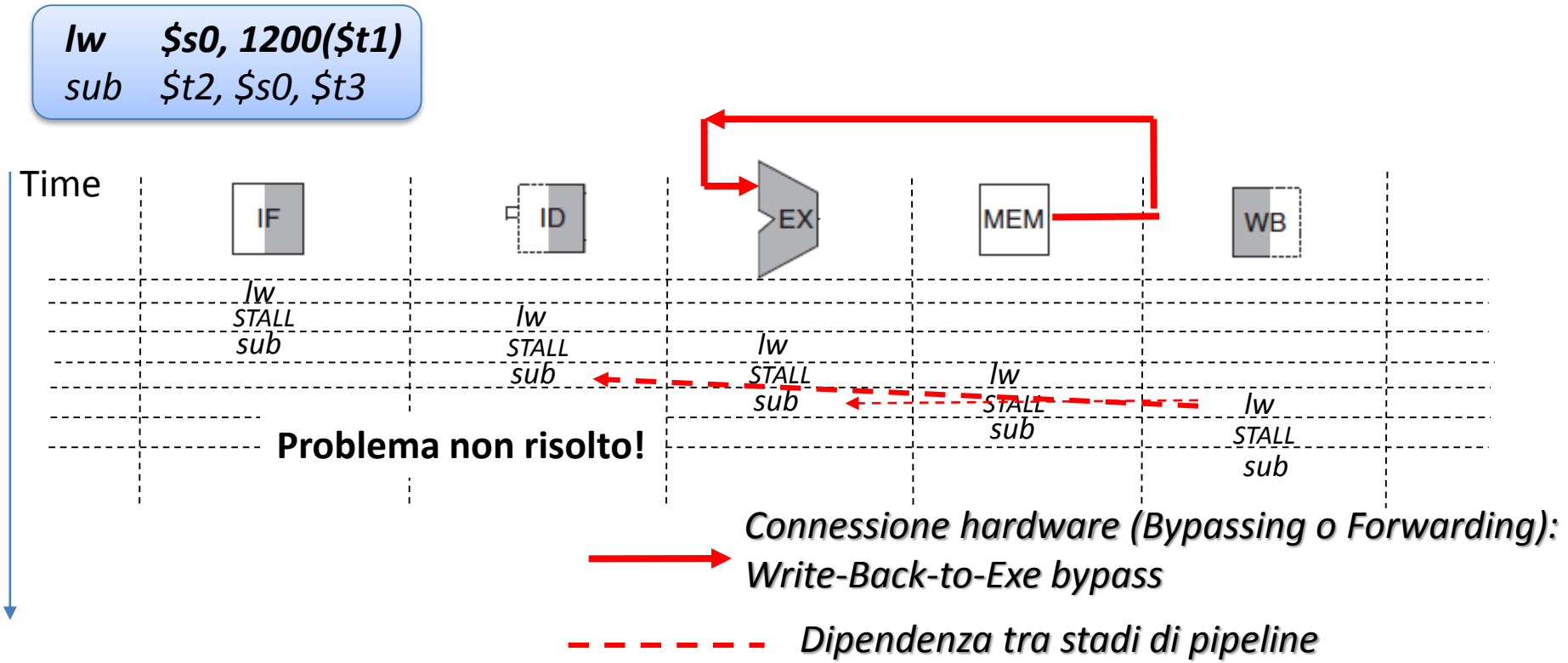
- Non tutti gli hazard dati possono essere risolti con bypassing, e alla fine stalli di pipeline sono inevitabili.



Vale solo per la connessione diretta tra operazioni che sono successive nel tempo.

Soluzione

- Non tutti gli hazard dati possono essere risolti con bypassing, e alla fine stalli di pipeline sono inevitabili.



Esempio

Linguaggio C

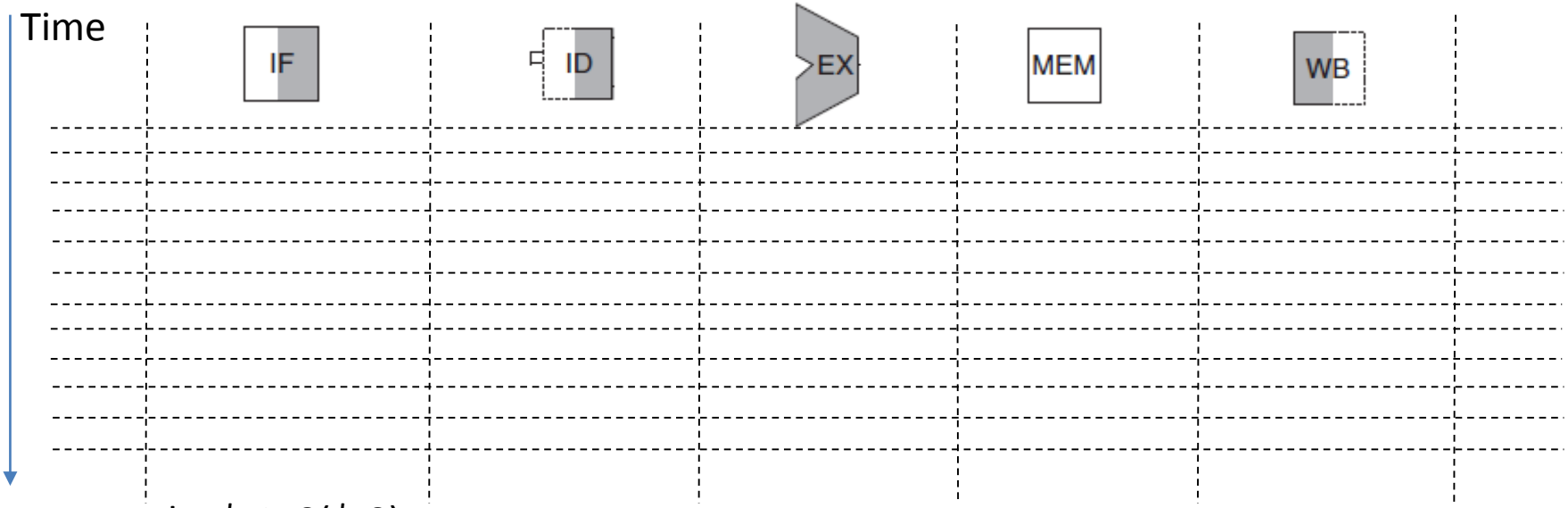
```
A = B + E;
C = B + F;
```

Linguaggio ASM

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($01)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

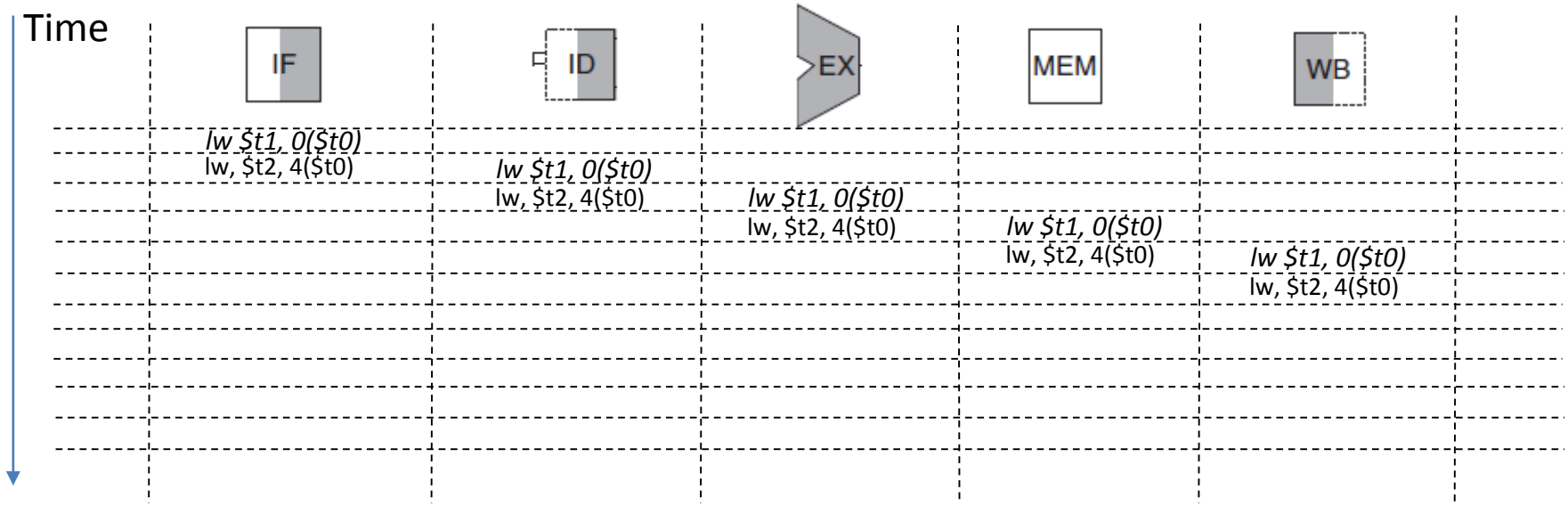
TROVARE e RISOLVERE GLI HAZARD NEL CODICE!

Esempio




```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

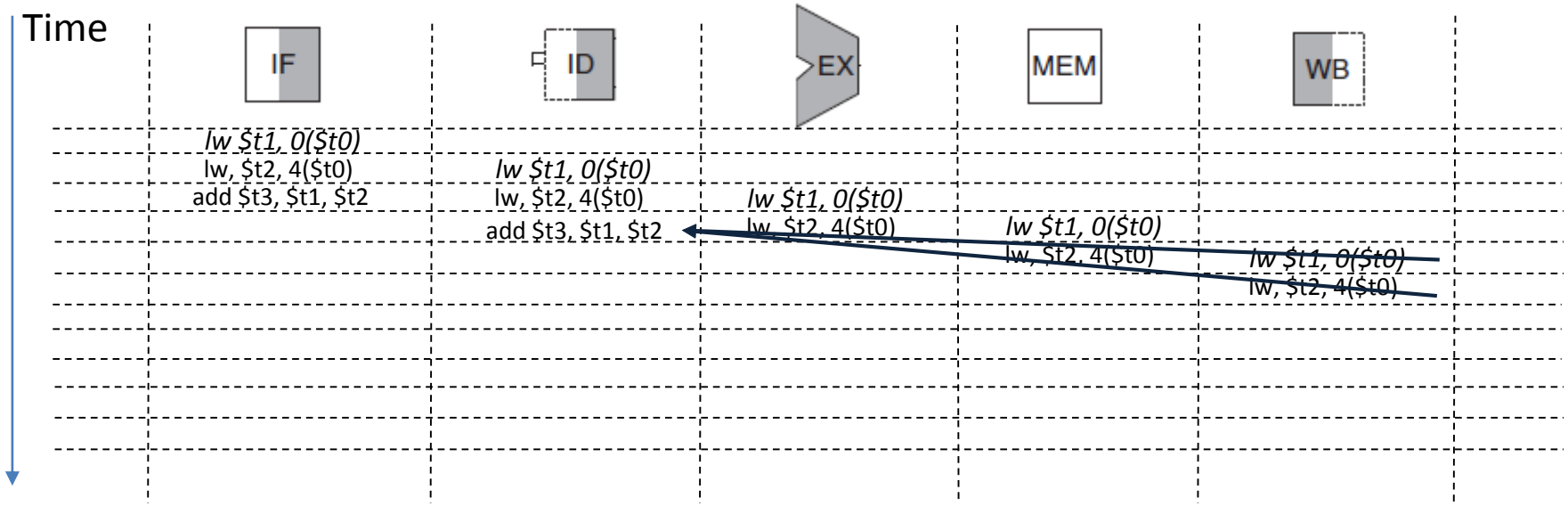
Esempio




```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

le prime due istruzioni non hanno dipendenze

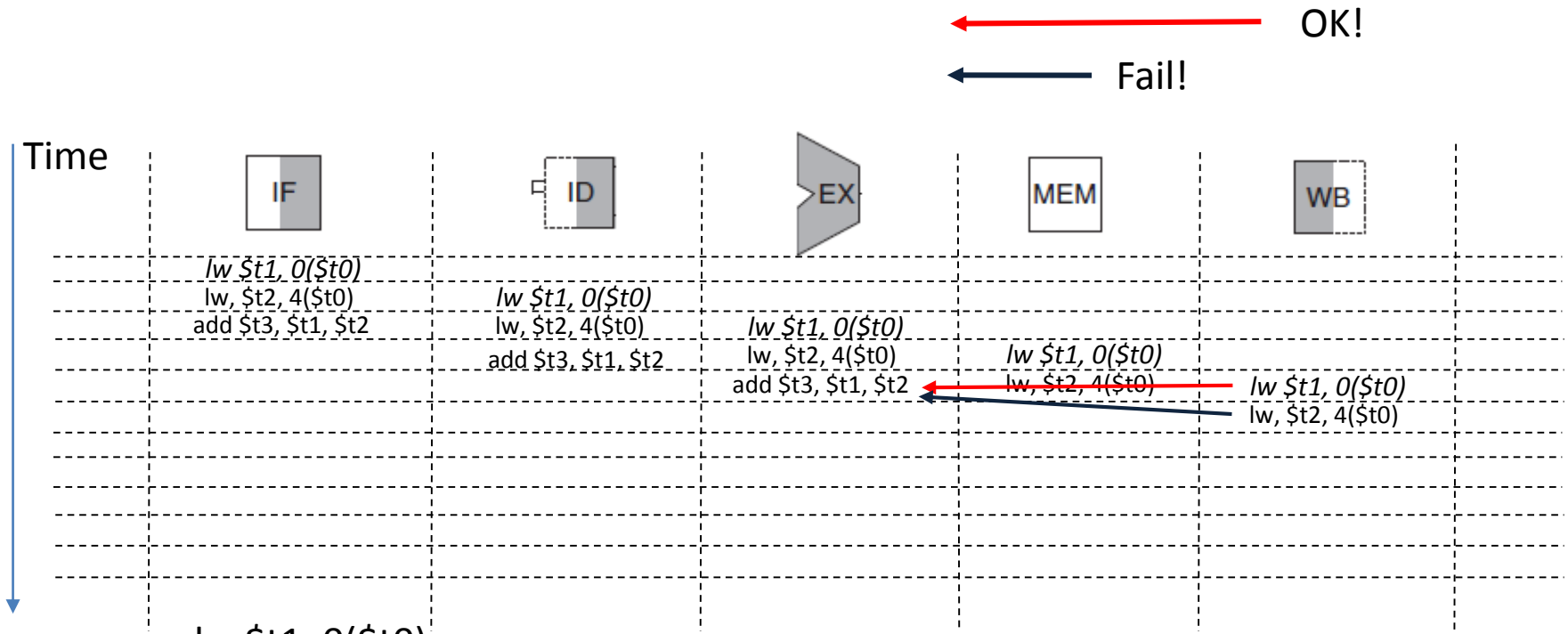
Esempio




```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Non si può fare. I registri sono aggiornati troppo tardi!
E con il bypassing?

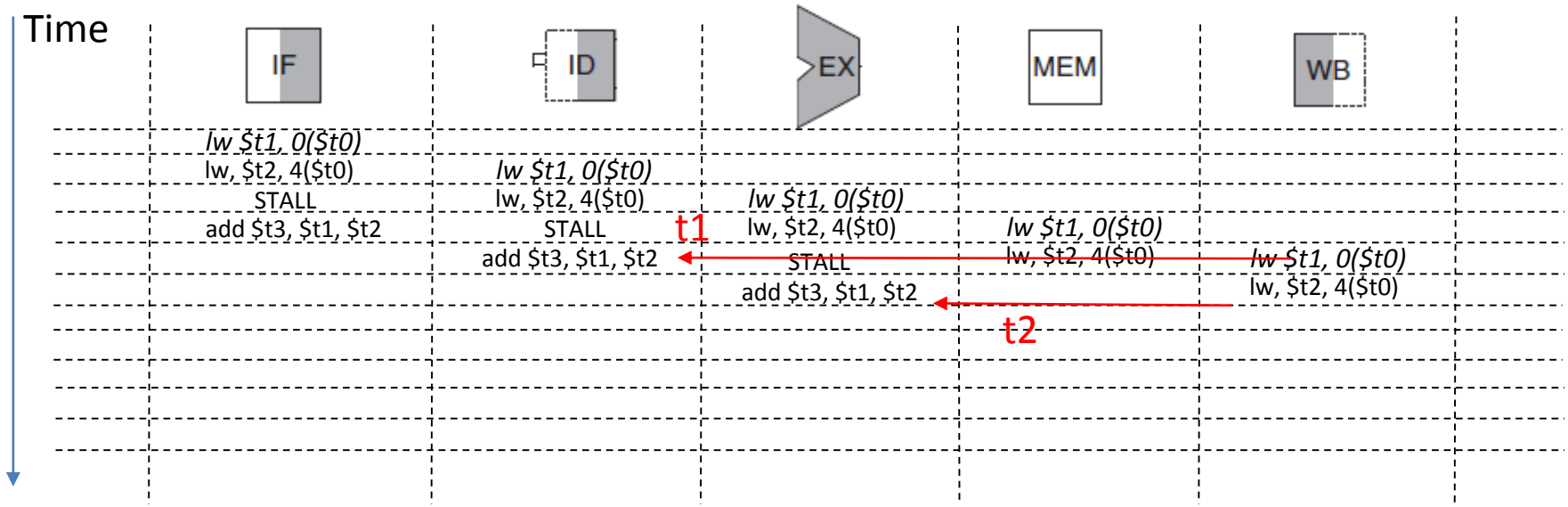
Esempio




```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Il problema si risolve solo parzialmente.
Devo introdurre per forza una bolla!!!!

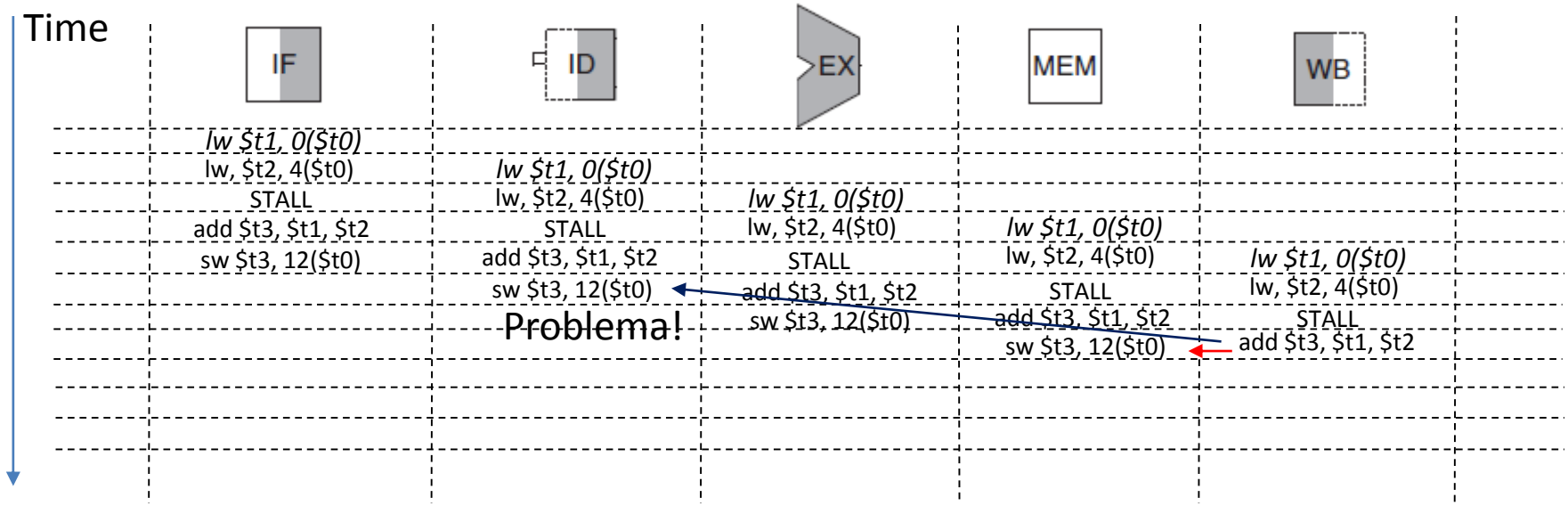
Esempio



lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)

Ora, con un doppio bypassing risolvo il problema:
Register File Internal forwarding e Write-back-to-Exe
Dentro al register file, mentre scrivo un registro lo posso anche leggere!

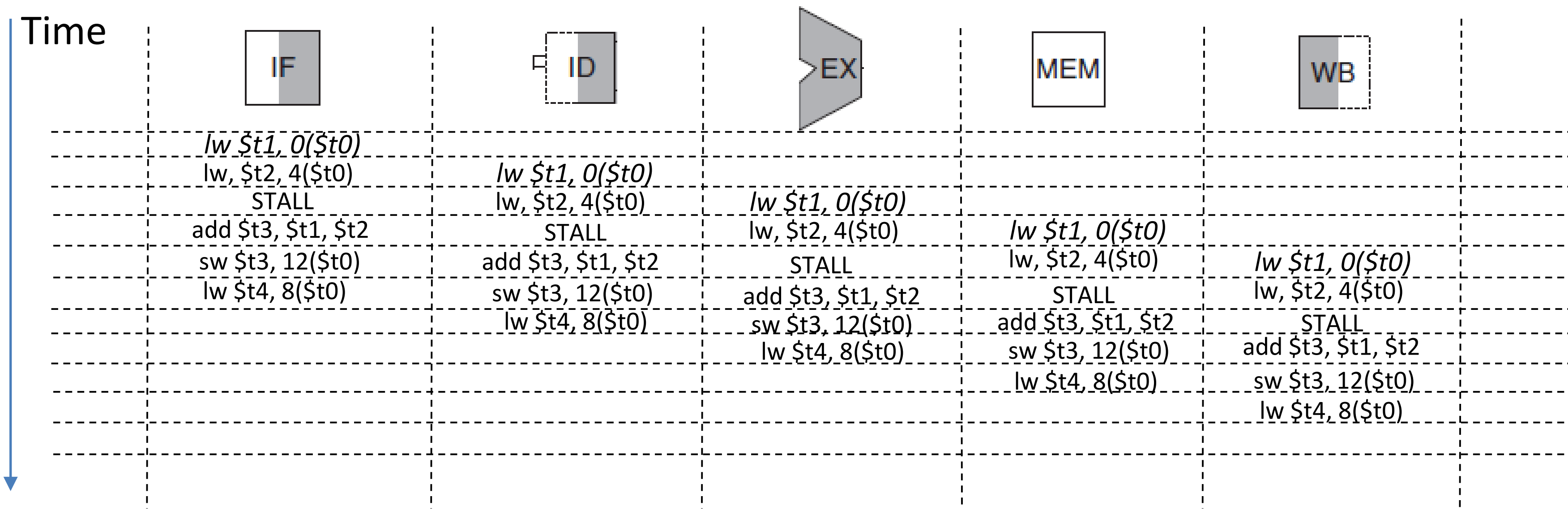
Esempio




```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

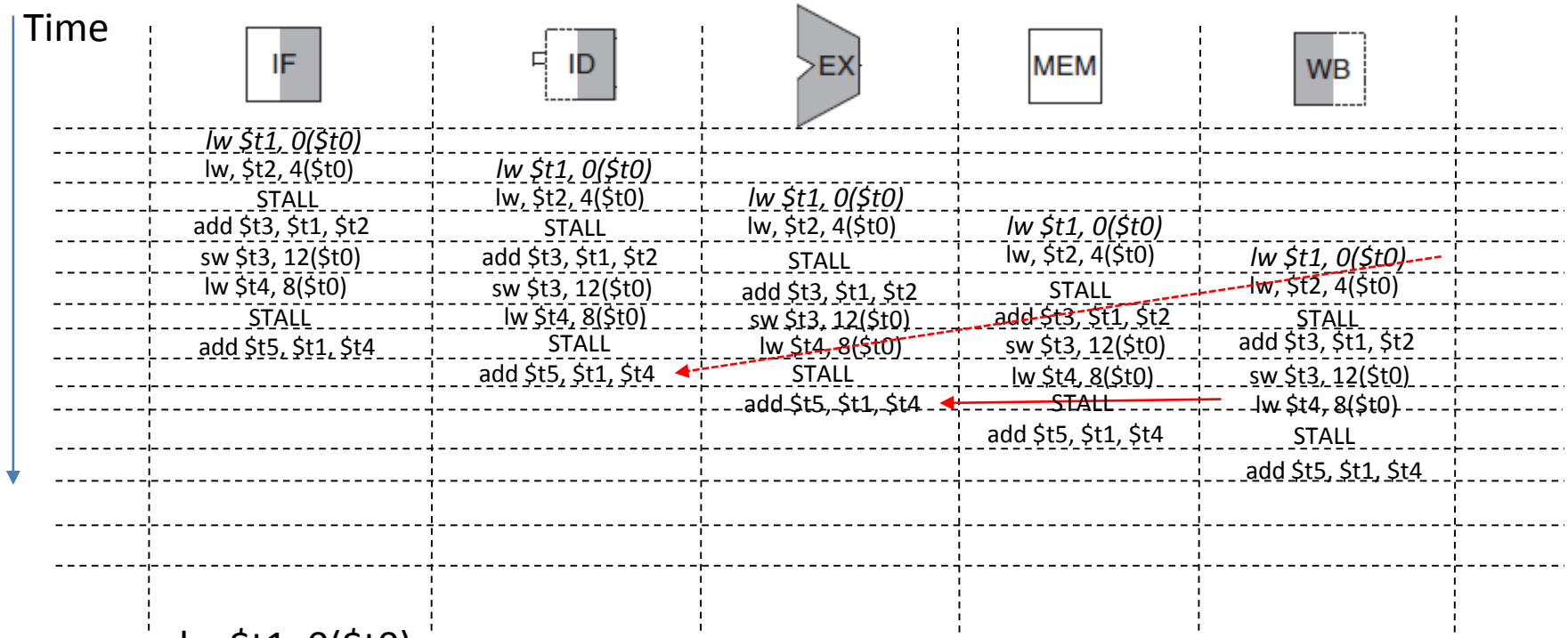
E' necessario un nuovo tipo di Forwarding da Write-Back a Memory

Esempio



lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)

Esempio



lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1, $t4
sw $t5, 16($t0)

$t1 è letto aggiornato, mentre per $t4 è necessario il bypass Write-Back-to-Exe

Esempio

Time

IF	ID	EX	MEM	WB
lw $t1, 0($t0)				
lw, $t2, 4($t0)	*lw $t1, 0($t0)*			
STALL	lw, $t2, 4($t0)	*lw $t1, 0($t0)*		
add $t3, $t1, $t2	STALL	lw, $t2, 4($t0)	*lw $t1, 0($t0)*	
sw $t3, 12($t0)	add $t3, $t1, $t2	STALL	lw, $t2, 4($t0)	*lw $t1, 0($t0)*
lw $t4, 8($t0)	sw $t3, 12($t0)	add $t3, $t1, $t2	STALL	lw, $t2, 4($t0)
STALL	lw $t4, 8($t0)	sw $t3, 12($t0)	add $t3, $t1, $t2	STALL
add $t5, $t1, $t4	STALL	lw $t4, 8($t0)	sw $t3, 12($t0)	add $t3, $t1, $t2
sw $t5, 16($t0)	add $t5, $t1, $t4	STALL	lw $t4, 8($t0)	sw $t3, 12($t0)
	sw $t5, 16($t0)	add $t5, $t1, $t4	STALL	lw $t4, 8($t0)
		sw $t5, 16($t0)	add $t5, $t1, $t4	STALL
			sw $t5, 16($t0)	add $t5, $t1, $t4
				sw $t5, 16($t0)

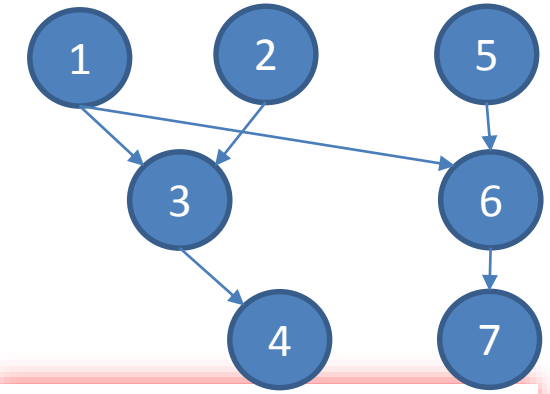
Tempo di completamento: 13 cicli!

Alternativa

Linguaggio C

```
A = B + E;
C = B + F;
```

L'ordine delle istruzioni nel codice assembler
definisce una relazione di ordine parziale



Se il compilatore si accorge degli hazard, mi puo'
cambiare l'ordine di esecuzione!

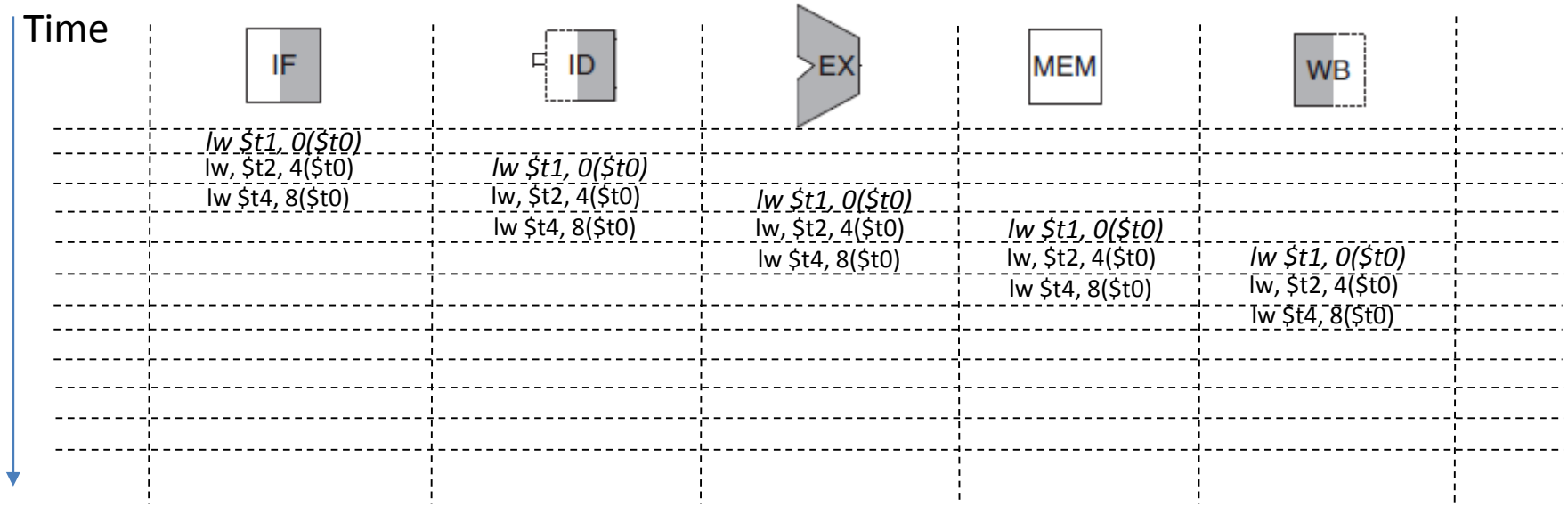
Linguaggio ASM

```
1  lw    $t1, 0($t0)
2  lw    $t2, 4($t0)
3  add   $t3, $t1,$t2
4  sw    $t3, 12($t0)
5  lw    $t4, 8($01)
6  add   $t5, $t1,$t4
7  sw    $t5, 16($t0)
```

Reordering

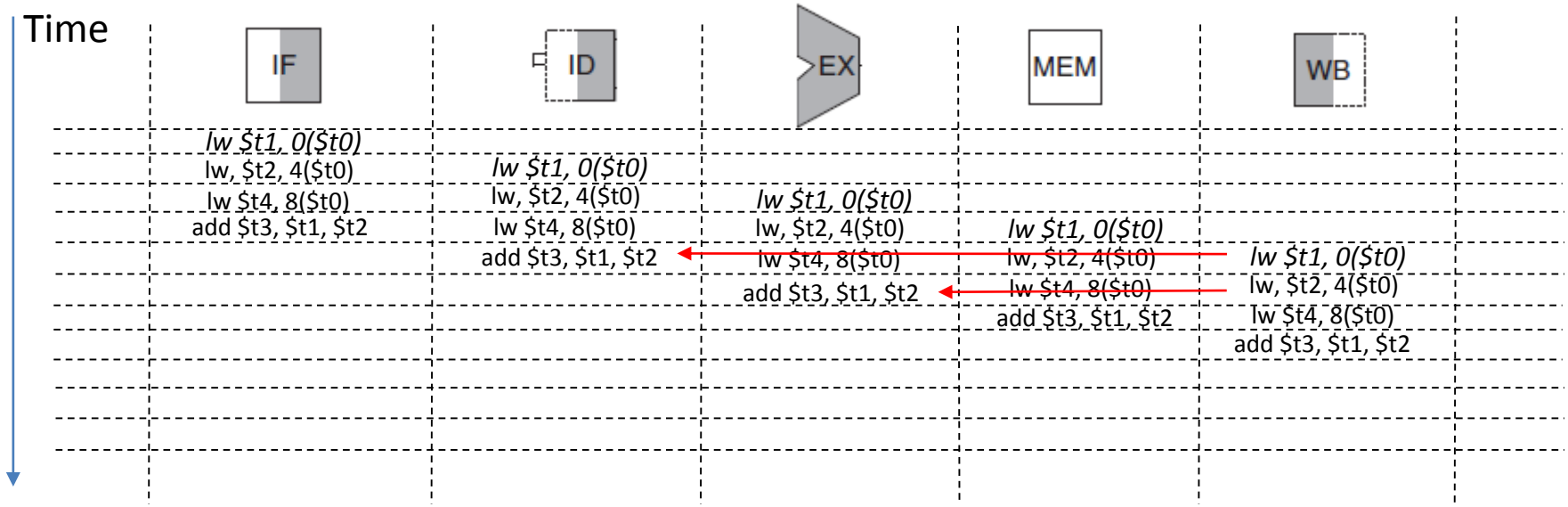
```
1  lw    $t1, 0($t0)
2  lw    $t2, 4($t1)
5  lw    $t4, 8($01)
3  add   $t3, $t1,$t2
4  sw    $t3, 12($t0)
6  add   $t5, $t1,$t4
7  sw    $t5, 16($t0)
```

Esempio con reordering

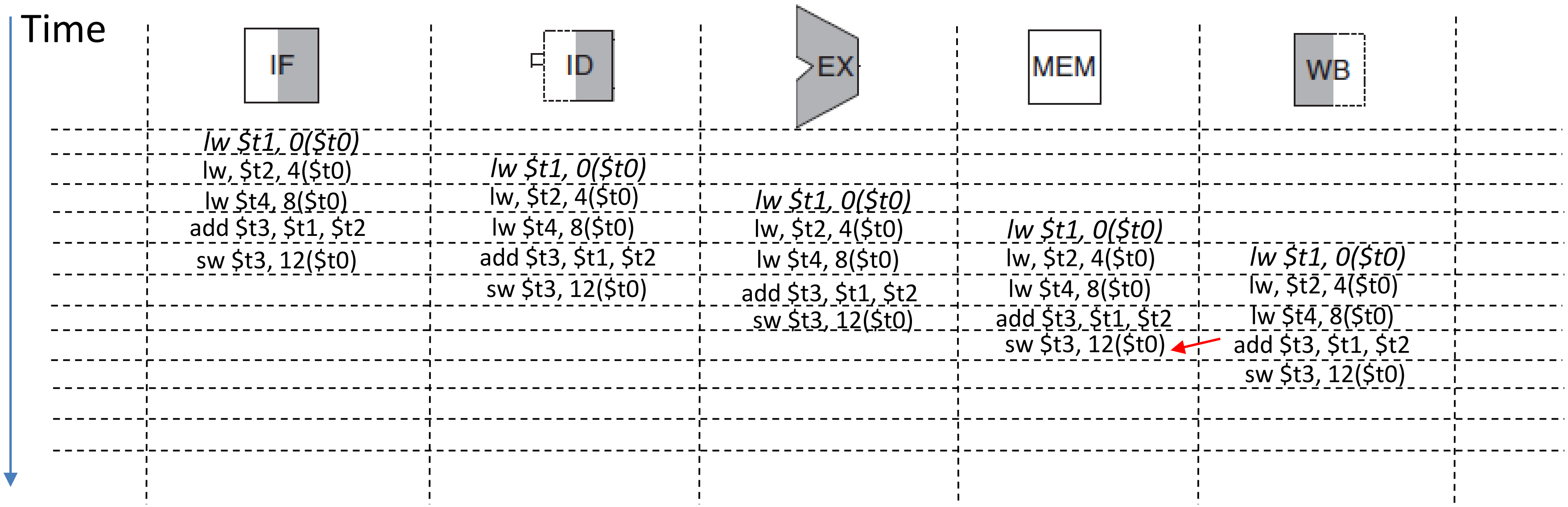



```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($01)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

Esempio con rordering

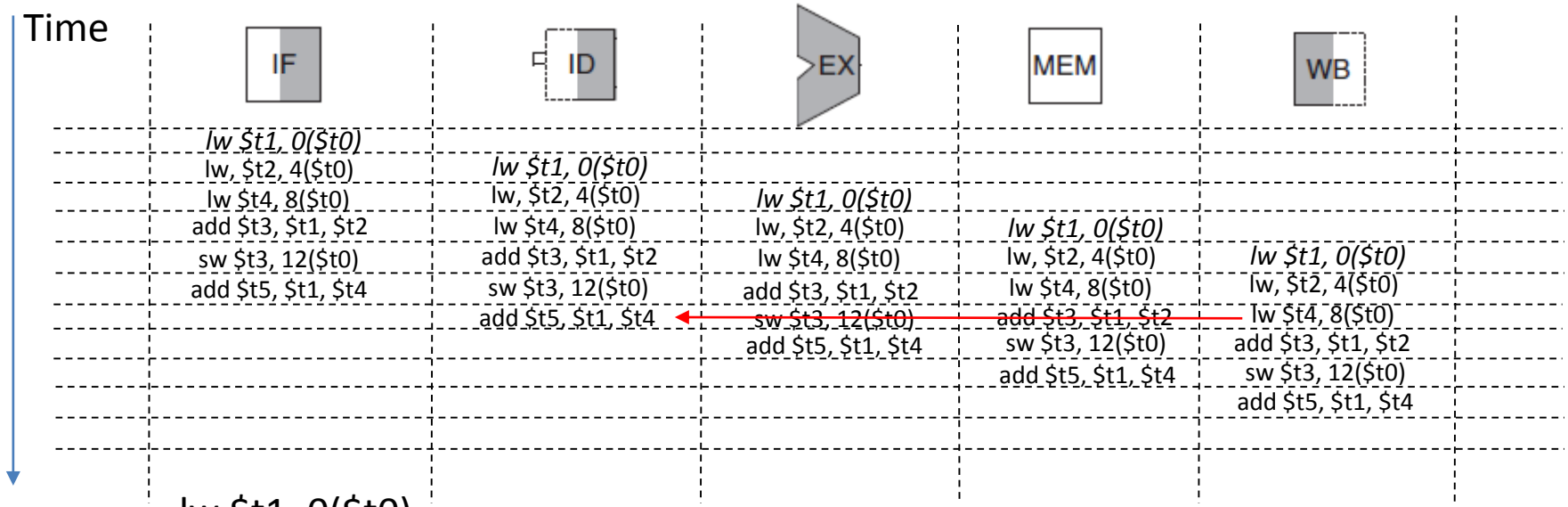


Esempio con reordering



lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($01)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)

Esempio con reordering



lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($01)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)

Esempio con reordering

Time

IF	ID	EX	MEM	WB
lw $t1, 0($t0)				
lw, $t2, 4($t0)	*lw $t1, 0($t0)*			
lw $t4, 8($t0)	lw, $t2, 4($t0)	*lw $t1, 0($t0)*		
add $t3, $t1, $t2	lw $t4, 8($t0)	lw, $t2, 4($t0)	*lw $t1, 0($t0)*	
sw $t3, 12($t0)	add $t3, $t1, $t2	lw $t4, 8($t0)	lw, $t2, 4($t0)	*lw $t1, 0($t0)*
add $t5, $t1, $t4	sw $t3, 12($t0)	add $t3, $t1, $t2	lw $t4, 8($t0)	lw, $t2, 4($t0)
sw $t5, 16($t0)	add $t5, $t1, $t4	sw $t3, 12($t0)	add $t3, $t1, $t2	lw $t4, 8($t0)
	sw $t5, 16($t0)	add $t5, $t1, $t4	sw $t3, 12($t0)	add $t3, $t1, $t2
		sw $t5, 16($t0)	add $t5, $t1, $t4	sw $t3, 12($t0)
			sw $t5, 16($t0)	add $t5, $t1, $t4
				sw $t5, 16($t0)

Tempo completamento: 11 cicli!

Vincoli di costo

- I processori, per motivi di costo, non implementano tutti i possibili bypass tra stadi di pipeline.
- Di conseguenza, il compilatore sfrutta i bypass esistenti, e laddove non disponibili:
  - o cambia l'ordine di esecuzione e risolve
  - o deve introdurre cicli di NOP
- Effetto degli stalli sulle prestazioni ?