

# Architettura degli Elaboratori

L'architettura della CPU – Introduzione alla pipeline



SAPIENZA  
UNIVERSITÀ DI ROMA

Alessandro Checco

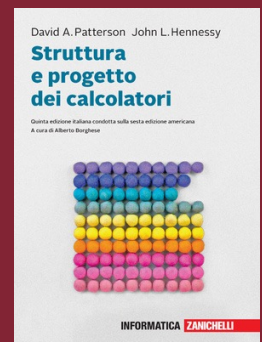
[checco@di.uniroma1.it](mailto:checco@di.uniroma1.it)

Special thanks and credits:

Andrea Sterbini, Iacopo Masi,

Claudio di Ciccio

[S&PdC] 4.5



# Argomenti

---

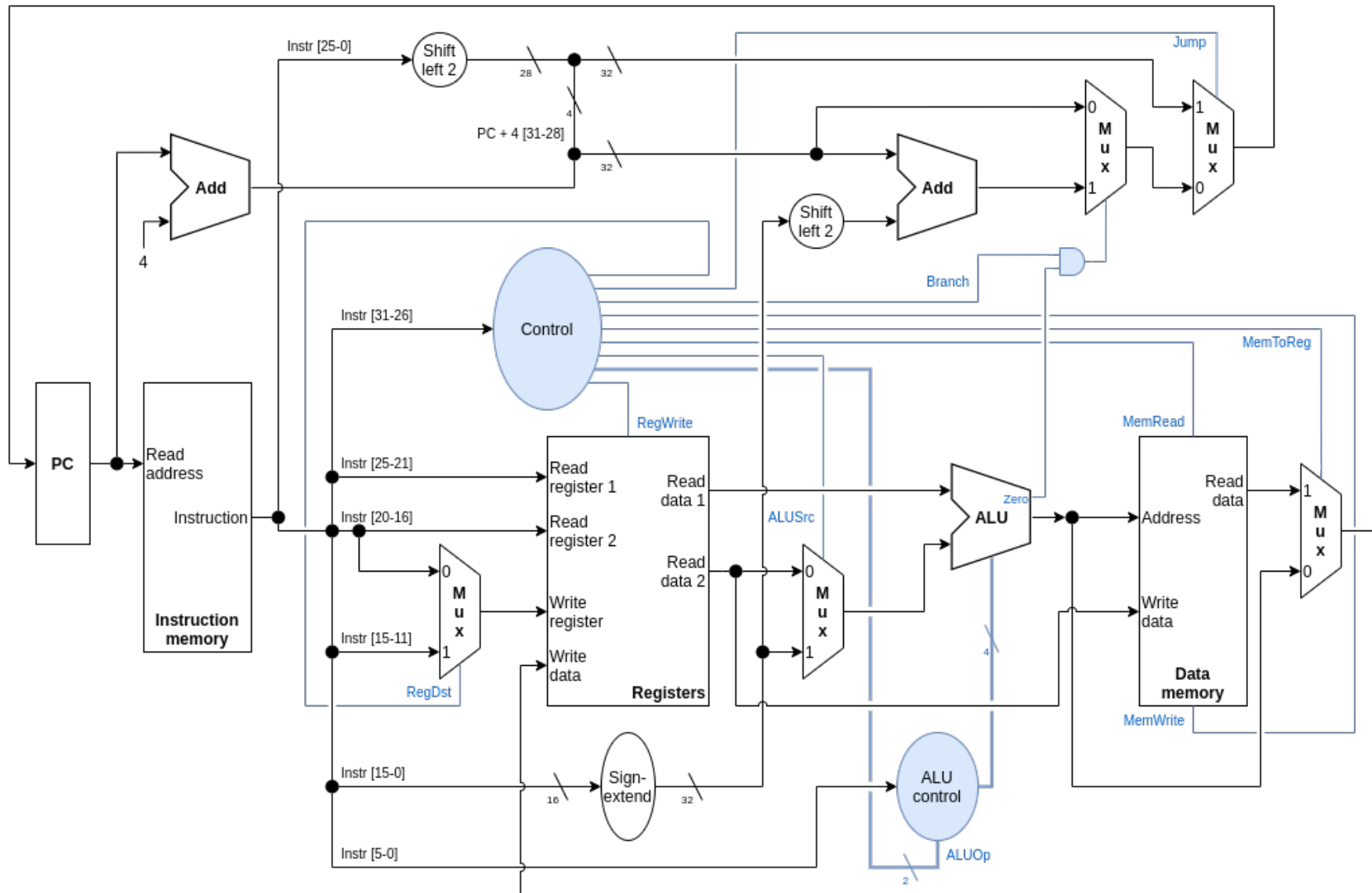
## Argomenti della lezione

- Le 5 fasi dell'istruzione
- Introduzione della pipeline
  - Solo una fase è attiva in ogni istante
  - Pipeline per processare più istruzioni contemporaneamente
  - Hazard sui dati e sul controllo


## Fasi dell'istruzione:

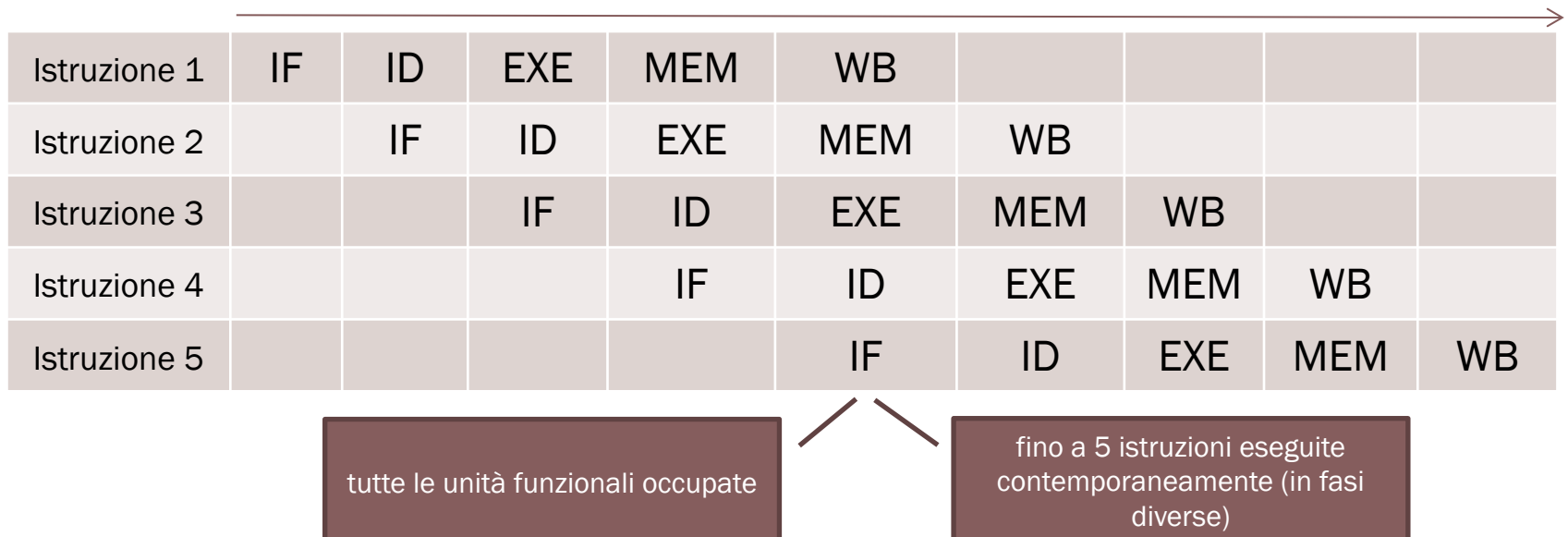
- **Fetch:** carica l'istruzione dalla memoria
- **Instruction Decode:** la CU decodifica l'istruzione e vengono letti gli argomenti dai registri
- **Execution:** l'ALU fa il calcolo necessario (tipo R o accesso alla memoria o branch)
- **Memory access:** viene letta/scritta la memoria (lw e sw)
- **Write Back:** il risultato dell'ALU o quello letto dalla memoria viene messo nel registro dest.

# CPU MIPS a un ciclo di clock



# La CPU è per l'80% inutilizzata!

- In ogni momento solo un'unità funzionale è attiva
  - Instruction Fetch (IF): **Memoria Istruzioni (e aggiornamento PC)**
  - Instruction Decode (ID): **Blocco registri (e CU)**
  - Execute (EXE): **ALU** 
  - Memory access (MEM): **Memoria dati**
  - Write Back (WB): **Banco registri**
- **Idea:** trasformare la CPU in una catena di montaggio
  - ogni unità funzionale elabora la fase che gli corrisponde e passa l'istruzione alla fase successiva



# Incremento della velocità

- Le 5 fasi devono sovrapporsi
  - individuare il periodo di clock uniforme per svolgere ognuna (durata della fase più lenta)

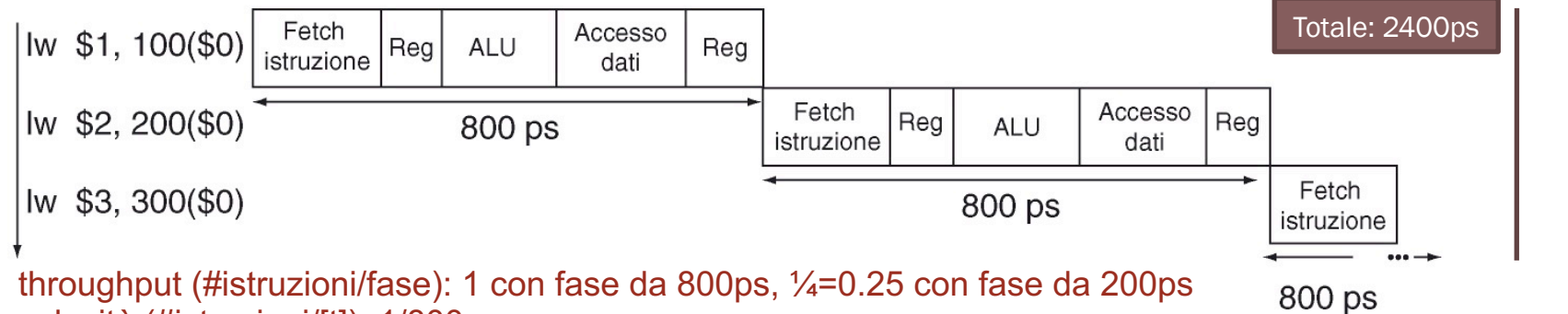
Classe dell'istruzione	Lettura dell'istruzione	Lettura dei registri	Operazione con la ALU	Accesso ai dati in memoria	Scrittura del register file	Tempo totale
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
Formato R (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Salto condizionato (beq)	200 ps	100 ps	200 ps			500 ps

- Possiamo ridurre il periodo di clock da **800ps** (durata massima di una istruzione) a **200ps** (durata massima di una fase)
  - periodo di clock ridotto ad un quarto → velocità quadruplicata
    - (in pratica, si cerca sempre di progettare la CPU in modo che le fasi abbiano tempi simili)
    - se tutte le fasi avessero tempi uguali si potrebbe aumentare la velocità di 5 volte
- Ad ogni colpo di clock una istruzione viene completata dalla pipeline
  - Possiamo arrivare a **quintuplicare (idealmente) il throughput (#istruzioni / fase)**
  - Il tempo per eseguire una singola istruzione è lo stesso  
(o addirittura maggiore se tempi non simili)

# Esempio di esecuzione

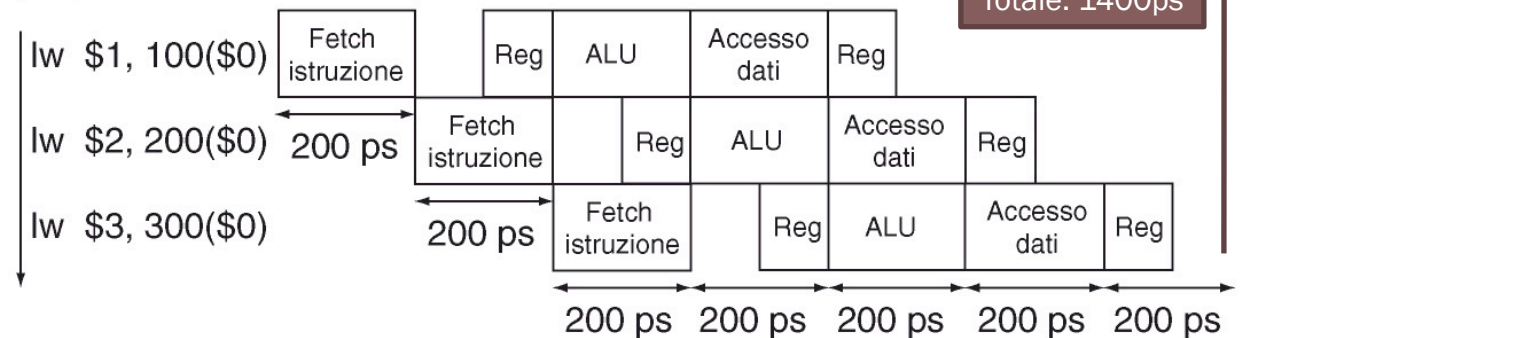
Ordine di esecuzione  
del  
programma

(sequenza delle istruzioni)

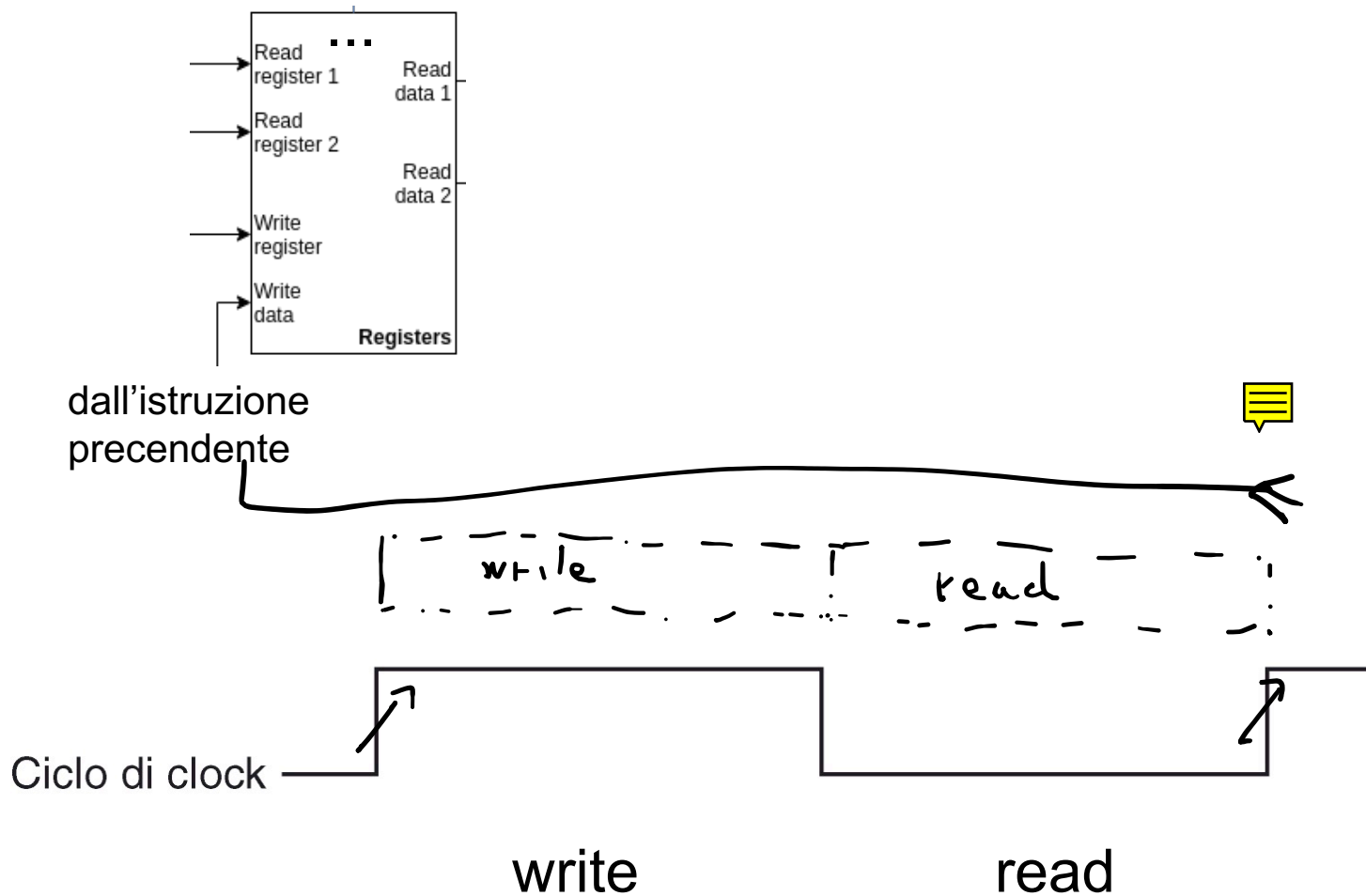


Ordine di esecuzione  
del  
programma

(sequenza delle istruzioni)



# Chiarimento su Reg e periodo di clock



# Ripasso: architetture CISC e RISC

MIPS è di tipo RISC

## Architettura CISC

(Complex Instruction Set Computer)

- Istruzioni di **dimensione variabile**

Per il fetch della successiva è **necessaria la decodifica**

- **Formato variabile**

Decodifica complessa



- Operandi in memoria

Molti accessi alla memoria per istruzione

- **Pochi registri interni**

Maggior numero di accessi in memoria

- **Modi di indirizzamento complessi**

Maggior numero di accessi in memoria

Durata variabile della istruzione

**Conflitti tra istruzioni più complicati**

- Istruz. Complesse: pipeline più complicata

## Architettura RISC

(Reduced Instruction Set Computer)

- Istruzioni di **dimensione fissa**

**Fetch della successiva senza decodifica della prec.**

- Istruzioni di **formato uniforme**

Per semplificare la fase di decodifica

- **Operazioni ALU solo tra registri**

**Senza accesso a memoria (ALU indipendente da REG)**

- **Molti registri interni**

Per i risultati parziali senza accessi alla memoria

- **Modi di indirizzamento semplici**

Con spiazzamento, 1 solo accesso a memoria

Durata fissa della istruzione

Conflitti semplici

- Istruz. semplici: pipeline più veloce



# Progettazione del set di istruzioni per pipeline

---

## Vantaggi del MIPS

1. Registri sorgente nella medesima posizione
  - Lettura di rs già durante la fase ID
2. Operandi in memoria solo per lw e sw (mai per op. logico-aritmetiche)
  - EXE per calcolare l'indirizzo, MEM per leggere/scrivere (ALU non più impegnata)
3. Un solo risultato, sempre all'ultimo stadio
  - Trasferimento del risultato sempre in coda
4. Allineamento degli operandi in memoria
  - Un singolo accesso in memoria necessario per dato (32 bit)
5. Medesima lunghezza di ogni istruzione
  - v. slide successiva

## Circa le dimensioni fisse e variabili

- La fase IF può sovrapporsi alla ID perché l'istruzione ha **dimensione uniforme**
  - Quindi possiamo calcolare il prossimo PC (+4) senza sapere di che tipo di istruzione si tratti
- Nelle architetture con **dimensione delle istruzioni variabile** è necessario aspettare anche la fase ID
  - Quindi la velocità tende ad essere più bassa

CISC									
IF	ID	altre	fasi	...					
		IF	ID	altre	fasi	...			
				IF	ID	altre	fasi	...	

RISC									
IF	ID	EX	MEM	WB					
	IF	ID	EX	MEM	WB				
		IF	ID	EX	MEM	WB			

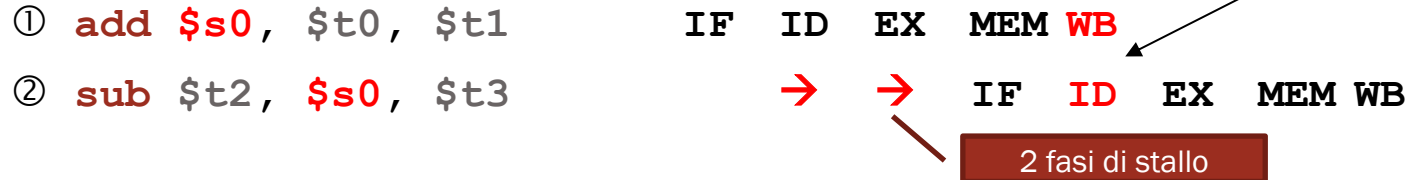
# Criticità (hazard)

## Criticità nella esecuzione (Hazard)

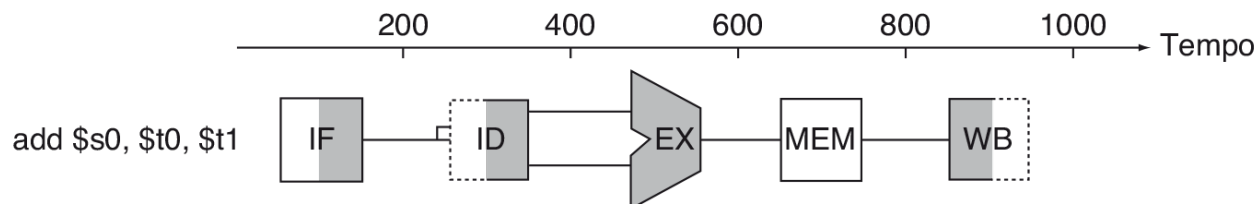
- **strutturali (structural hazard):** le risorse hardware non sono sufficienti
  - per es., memoria dati e memoria istruzioni in una sola unità (IF/MEM collision) *risolto in fase di design*
- **sui dati (data hazard):** il dato necessario non è ancora pronto
- **sul controllo (control hazard):** la presenza di un salto cambia il flusso di esecuzione delle istruzioni



Esempio (data hazard sul registro \$s0):



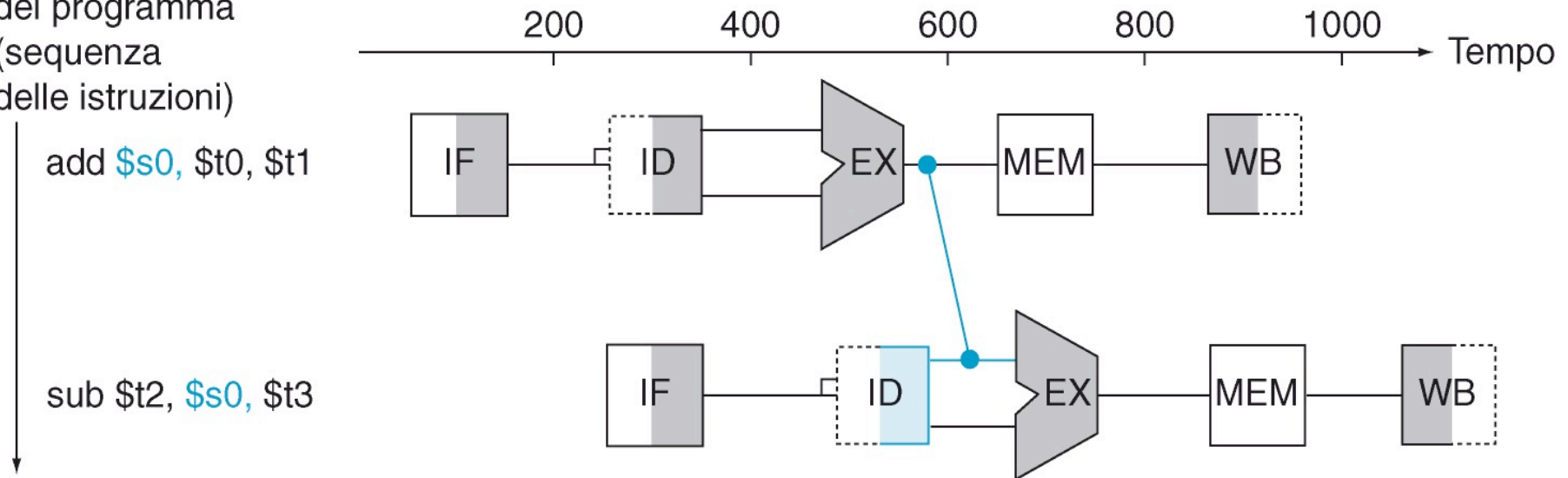
- La ② non può eseguire la lettura degli argomenti se la ① non esegue il WB
  - Le fasi WB della e ID della possono essere sovrapposte se la WB avviene nella prima metà e ID nella seconda metà del periodo di clock



# Propagazione (Bypassing/forwarding)

- In alcuni casi l'informazione necessaria è già presente nella pipeline prima del WB

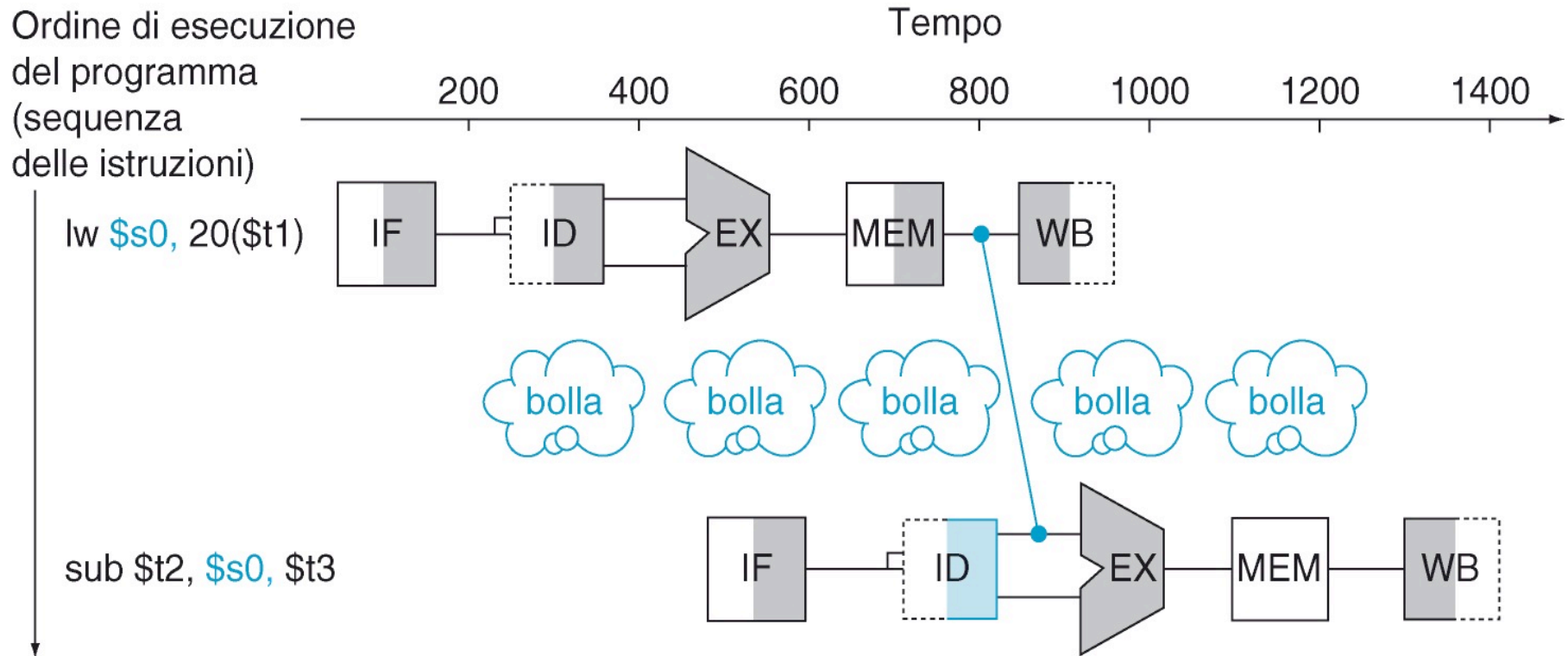
Ordine di esecuzione  
del programma  
(sequenza  
delle istruzioni)



- Possiamo **inserire nel datapath** delle «**scorciatoie**»
  - recapitano il dato all'unità funzionale che ne ha bisogno senza aspettare la fase di WB
- In questo esempio **NON** è necessario aspettare 2 colpi di clock come nel precedente
  - Possibile quando lo stadio che deve ricevere il dato è **successivo** a quello che lo produce nel **diagramma temporale** della pipeline

## Bolla (bubble)

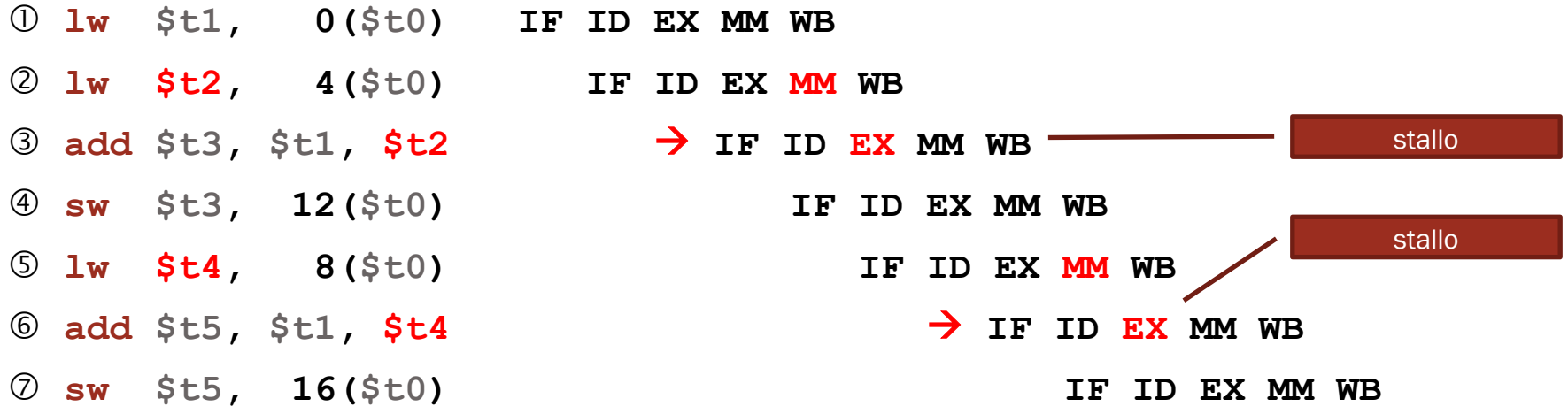
Se la fase che ha bisogno del dato si trova prima (nel tempo) di quella che lo produce sarà necessario inserire una attesa (stallo o bolla)



# Riordinamento delle istruzioni

Esempio:  $a = b + e$

$c = b + f$



# Riordinamento delle istruzioni (con bypass)

Esempio:  $a = b + e$   
 $c = b + f$

Possibile purché si mantenga la stessa semantica

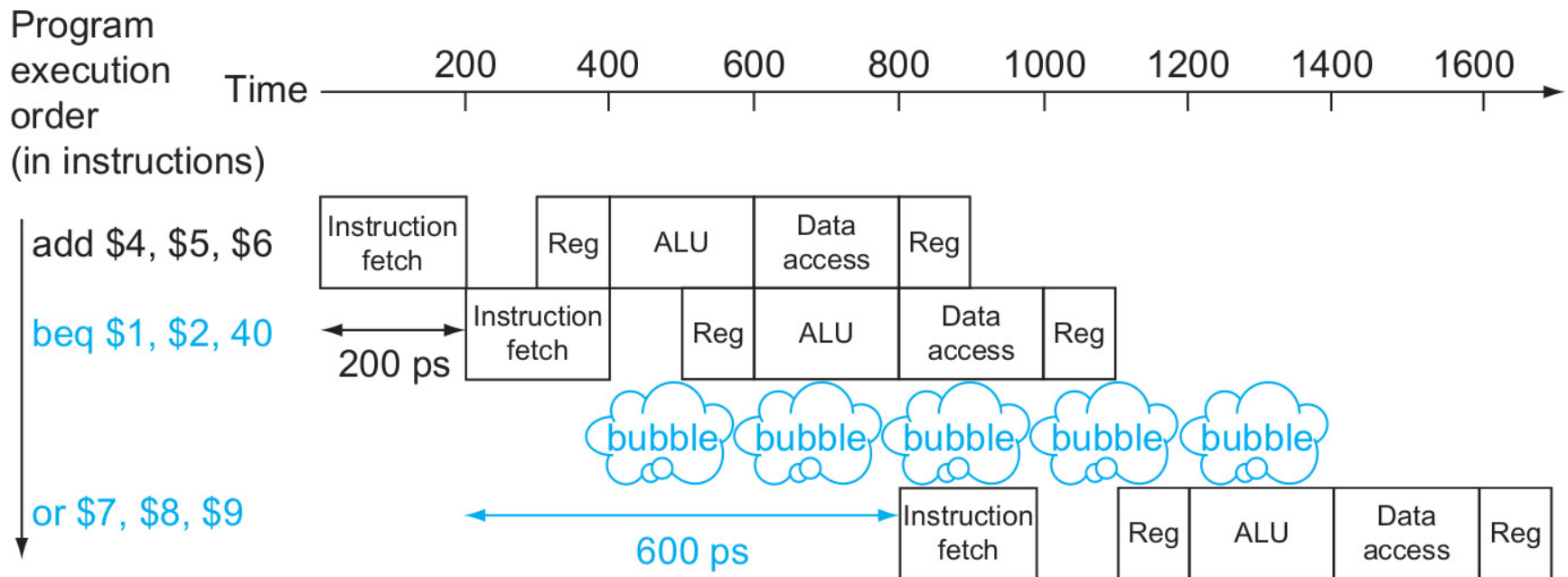


①	<b>lw</b>	\$t1,	0(\$t0)	IF	ID	EX	MM	WB	
②	<b>lw</b>	<b>\$t2</b> ,	4(\$t0)	IF	ID	EX	<b>MM</b>	WB	
③	<b>add</b>	\$t3,	\$t1, <b>\$t2</b>	→ IF	ID	<b>EX</b>	MM	WB	stallo
④	<b>sw</b>	\$t3,	12(\$t0)		IF	ID	EX	MM	WB
⑤	<b>lw</b>	<b>\$t4</b> ,	8(\$t0)		IF	ID	EX	<b>MM</b>	WB
⑥	<b>add</b>	\$t5,	\$t1, <b>\$t4</b>		→ IF	ID	<b>EX</b>	MM	WB
⑦	<b>sw</b>	\$t5,	16(\$t0)		IF	ID	EX	MM	WB

①	<b>lw</b>	\$t1,	0(\$t0)	IF	ID	EX	MM	WB
②	<b>lw</b>	\$t2,	4(\$t0)	IF	ID	EX	<b>MM</b>	WB
⑤	<b>lw</b>	\$t4,	8(\$t0)	IF	ID	EX	<b>MM</b>	WB
③	<b>add</b>	\$t3,	\$t1, \$t2	IF	ID	<b>EX</b>	MM	WB
④	<b>sw</b>	\$t3,	12(\$t0)	IF	ID	EX	MM	WB
⑥	<b>add</b>	\$t5,	\$t1, \$t4	IF	ID	<b>EX</b>	MM	WB
⑦	<b>sw</b>	\$t5,	16(\$t0)	IF	ID	EX	MM	WB

# Hazard sul controllo

- Esempio: salto condizionato (se `$1 == $2`, esegui una `or`; altrimenti, una `lw`)
  - L'istruzione seguente (già in pipeline) può dover essere scartata
    - per eseguire quella di destinazione del salto
  - L'istruzione seguente può essere caricata solo dopo che il salto è stato deciso
    - necessari 2 stalli se `beq` viene decisa in EXE o 1 se viene decisa in ID



in questo esempio salto deciso in EXE

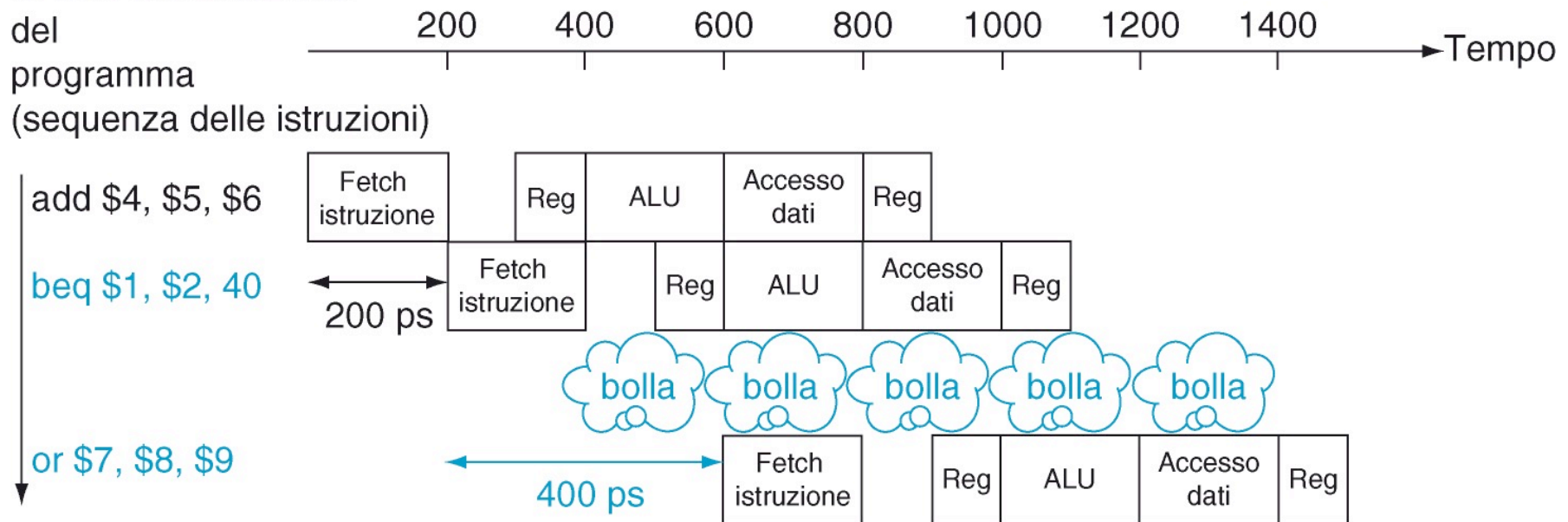


# Hazard sul controllo

- Esempio: salto condizionato
    - L'istruzione seguente (già in pipeline) può dover essere scartata
      - per eseguire quella di destinazione del salto
    - L'istruzione seguente può essere caricata solo dopo che il salto è stato deciso
      - necessari 2 stalli se beq viene decisa in EXE o 1 se viene decisa in ID
- (lo schema della slide assume che beq sia decisa nella fase ID usando HW aggiuntivo)



Ordine di esecuzione  
del  
programma  
(sequenza delle istruzioni)



in questo esempio salto deciso in ID

# Come mitigare i control hazard

- **Anticipare la decisione (soluz HW):**
  - se beq viene calcolata dalla ALU nella fase EXE ci saranno 2 istruzioni da «buttare»
  - se invece beq viene decisa nella fase ID è sufficiente «buttare» 1 istruzione
- **Ritardare il salto:**
  - se l'istruzione che segue la beq viene SEMPRE eseguita anche se il salto viene fatto, si elimina di fatto lo stallo eseguendo quell'istruz al posto della bolla (non sempre possibile)
- **Branch prediction:** (previsione del salto)
  - la CPU osserva i salti eseguiti e cerca di pre-caricare l'istruzione (seguente o di destinazione) eseguita più spesso
    - static predictors v. dynamic predictors

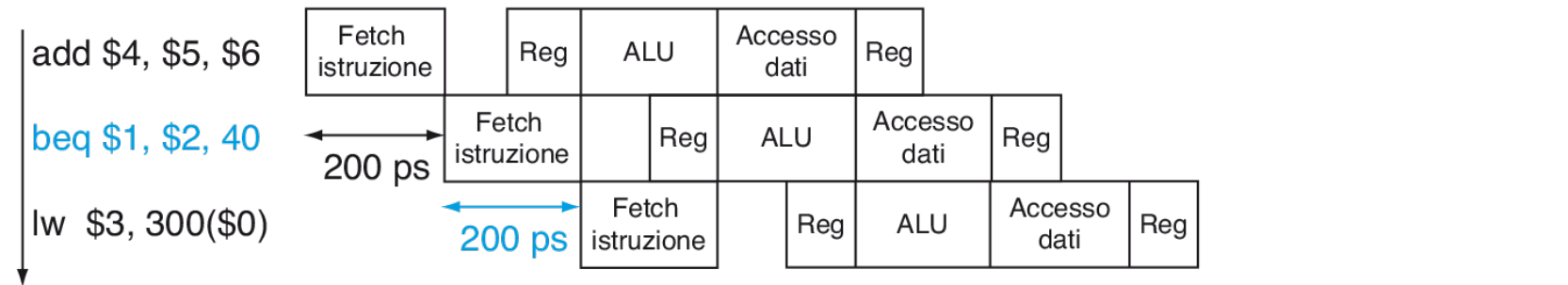
L'implementazione della beq che studiamo assume che il salto non venga eseguito → carica sempre l'istruzione seguente

```
1  .data
2  Array: .word 1,5,8,7,6
3  N: .word 5
4  # Save in $s0 the sum of the values in Array
5
6  .text
7  xor $t1,$t1,$t1 # int i = 0;
8  sub $s1,$s1,$s1 # int s = 0;
9  lw $s7,N # $s7 ← N # Load N onto a register
10 sll $s7,$s7,2 # Multiply $s7 by 4 (due to the word size)
11 While: # while (i < N) {
12     bge $t1,$s7,WhileEnd # IF i >= N THEN jump to WhileEnd
13     lw $t2,Array($t1) # Load Array[i];
14     add $s0,$s0,$t2 # s = s + Array[i];
15     addi $t1,$t1,4 # i+=1;
16     j While # }
17 WhileEnd:
```

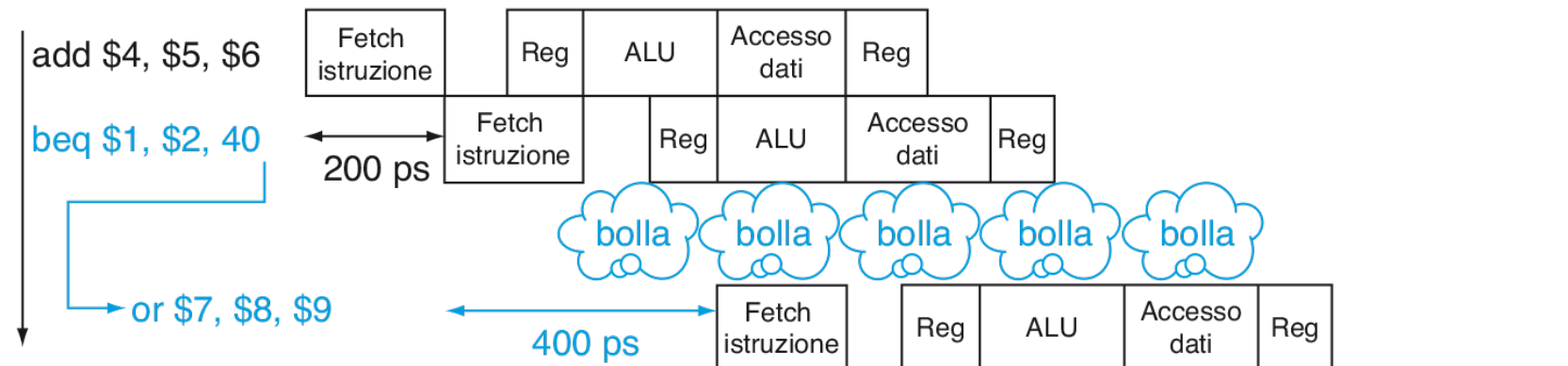


# Effetti della predizione

Ordine di esecuzione  
del  
programma  
(sequenza delle istruzioni)



Ordine di esecuzione  
del  
programma  
(sequenza delle istruzioni)



## Esercizio

Per ognuna delle sequenze di istruzioni indicate di seguito, stabilire quale delle tre condizioni seguenti è valida:

1. occorre mettere in stallo la pipeline
2. si possono evitare stalli utilizzando la propagazione (MEM -> EXE o EXE -> EXE)
3. si può eseguire il codice senza stalli né propagazione

Sequenza 1	Sequenza 2	Sequenza 3
<code>lw \$t0, 0(\$t0)</code> <code>add \$t1,\$t0,\$t0</code>	<code>add \$t1, \$t0, \$t0</code> <code>addi \$t2, \$t0, 5</code> <code>addi \$t4, \$t1, 5</code>	<code>addi \$t1, \$t0, 1</code> <code>addi \$t2, \$t0, 2</code> <code>addi \$t3, \$t0, 2</code> <code>addi \$t3, \$t0, 4</code> <code>addi \$t5, \$t0, 5</code>



## Esercizio per casa

Calcolare il numero di cicli necessari ad eseguire le istruzioni seguenti:

- individuare i **data e control hazard**
- per **determinare se con il forwarding possono essere risolti**, tracciare il diagramma temporale della pipeline
- determinare **quali non possono essere risolti e necessitano di stalli** (e quanti stalli)
- tenere conto del tempo necessario a **caricare la pipeline**

Assumere

- che la **beq** salti alla fine della fase EXE (slide 16),
- che il salto **beq** non sia ritardato,
- che **j** non introduca stalli.

*# Somma un vettore di word*  
*sommaVettore:*

```
        li $t0, 0 # somma
        li $t1, 40 # fine
        li $t2, 0 # offset
ciclo: beq $t2, $t1, fine
        lw $t3, vettore($t2)
        add $t0, $t0, $t3
        addi $t2, $t2, 4
        j ciclo
fine:  li $v0, 10
        syscall
```