



# Dispense complete sulla prima parte del corso di Architettura dei calcolatori e sistemi operativi a.a. 2016/2017

Architettura dei calcolatori e sistemi operativi (Politecnico di Milano)



---

# AXO - Architettura dei Calcolatori e Sistema Operativo

## organizzazione strutturata dei calcolatori



# I livelli

---

- I calcolatori sono progettati come una serie di **livelli** ognuno dei quali si **basa** sui **livelli precedenti**.
- Ogni **livello** rappresenta un'**astrazione diversa** con strutture dati e funzionalità differenti.
- L'insieme di tipi di dati, operazioni e caratteristiche d'ogni livello prende il nome d'**ARCHITETTURA (o STURUTTURA) (macchina di livello  $i$  -  $M_i$ )**.
- La descrizione dell'architettura d'un livello presenta l'insieme delle **caratteristiche visibili all'utente** di quel livello. Una macchina  $M_i$  ha associato un **linguaggio  $L_i$**  eseguibile su quella macchina.

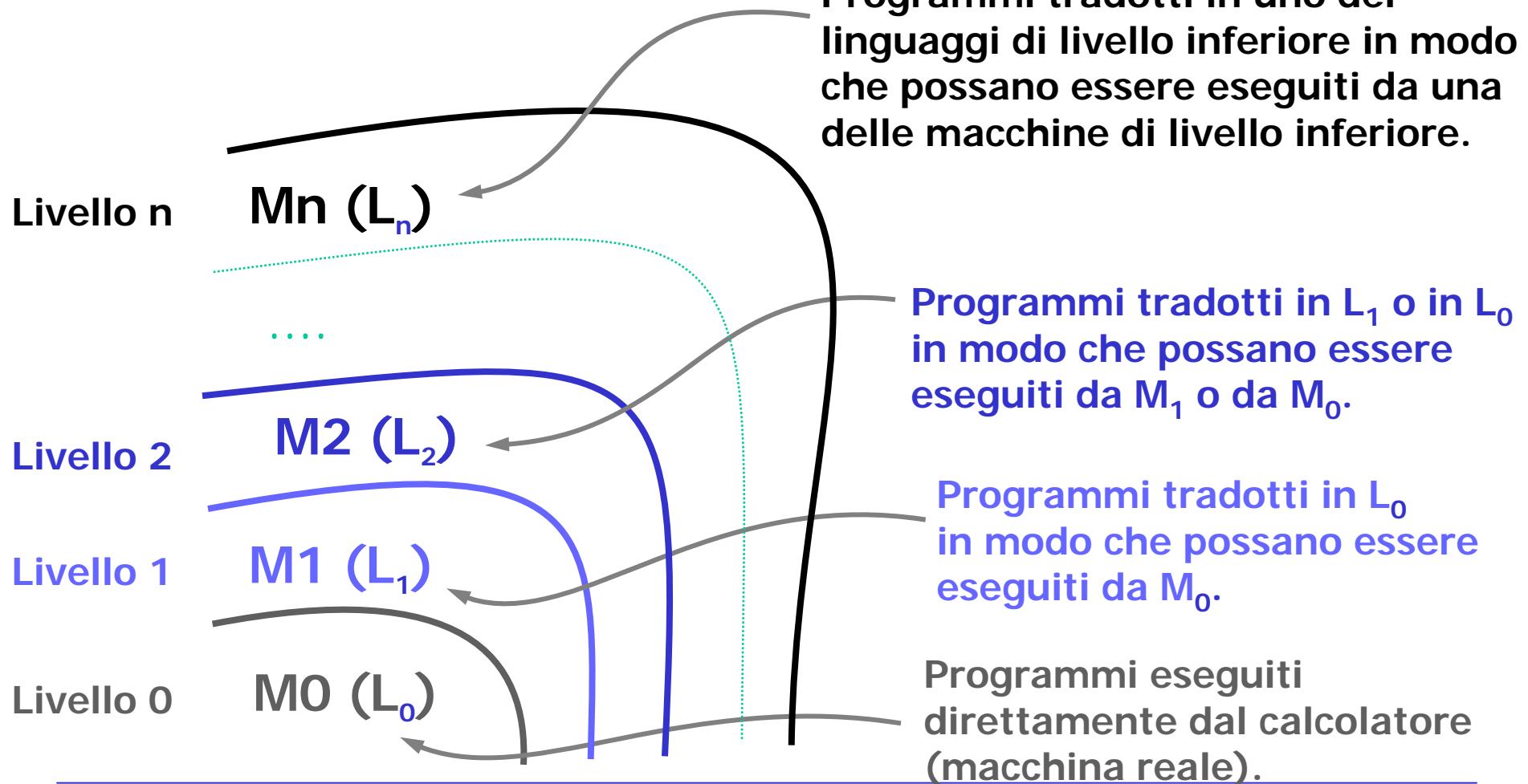


# Relazione tra una macchina e il relativo linguaggio macchina

- La macchina definisce il linguaggio:
  - una macchina (reale o virtuale) permette di definire il **linguaggio macchina** a essa associato, come l'insieme di tutte le istruzioni che la macchina stessa è in grado d'eseguire.
- Il linguaggio definisce la macchina:
  - un linguaggio permette di definire la macchina (reale o virtuale) a esso associata come l'esecutore capace di comprendere tutti i programmi scritti in quel linguaggio.



# Macchina a più livelli





# Come passare da $L_i$ a $L_{i-1}$

## □ Traduzione (o compilazione):

- un programma apposito (**compilatore**) traduce il programma  $P_{L_i}$ , scritto in linguaggio  $L_i$ , in un programma  $P_{L_{i-1}}$ , scritto in linguaggio  $L_{i-1}$ ;
- il nuovo programma  $P_{L_{i-1}}$  viene quindi eseguito.

## □ Interpretazione:

- un programma apposito (**interprete**) esamina il programma  $P_{L_i}$ , scritto in linguaggio  $L_i$ , e, istruzione per istruzione, lo traduce nel linguaggio  $L_{i-1}$  e lo esegue.



# Calcolatore a *n* livelli

- Per **scrivere i programmi per il livello *n*** non è necessario conoscere come venga effettuata la traduzione e quindi l'esecuzione.
- I programmi possono essere:
  - eseguiti direttamente dalla macchina reale;
  - tradotti direttamente nel linguaggio  $L_0$ ;
  - interpretati da un interprete che viene a sua volta interpretato da un altro interprete ...;
  - ...
- La conoscenza dei livelli intermedi è importante per chi voglia capire:
  - come **funzioni** un calcolatore;
  - come si **progettano** una macchina virtuale.



# I livelli nei calcolatori moderni

5. Livello applicativo
4. Livello del linguaggio macchina (o assemblatore)
3. Livello del sistema operativo
2. Livello dell'Instruction Set Architecture (ISA)
1. Livello della microarchitettura
0. Livello logico
  - 1. ... Livello dei dispositivi ...
  - 2. ... (fisica dello stato solido) ...



# Livello dei dispositivi

- ❑ Transistor che formano i circuiti elettronici di cui è composto un calcolatore.
- ❑ Raggiunge un livello di dettaglio che viene in genere trascurato nella progettazione dei calcolatori.
- ❑ Gli stessi transistor vengono descritti a livelli diversi:
  - switch, cioè interruttori che possono essere aperti o chiusi;
  - dispositivi caratterizzati da un comportamento approssimato da funzioni di 2º grado;
  - dispositivi caratterizzati da una serie d'equazioni empiriche.
- ❑ A un livello ancora più basso ci s'occupa della struttura dei transistor (**fisica dello stato solido**).



# Livello 0: la logica digitale

---

- La macchina è formata da **porte logiche**.
  - Ogni porta riceve in ingresso dei segnali binari (cioè segnali che possono essere 0 o 1) e calcola una funzione semplice (AND, OR, ...).
  - Alcune porte, collegate opportunamente, possono formare una **memoria da un bit** (bistabile).
  - Combinando N memorie da un bit si può formare un **registro** capace di memorizzare un numero binario (non più grande di  $2^N-1$ ).
  - Combinando le porte si realizzano i circuiti che formano i calcolatori.
-



# Livello 1: la microarchitettura





# Livello 1: la microarchitettura

- **Elaborazione - unità di calcolo (o data path)**
  - registri **d'uso generale** (general purpose) come memoria locale;
  - Unità Aritmetico-Logica (Arithmetic-Logic Unit, **ALU**) capace d'eseguire operazioni aritmetico-logiche semplici;
  - elementi di connessione tra registri e ALU.
- **Controllo - unità di controllo (o control path)**
  - registri dedicati al controllo (**PC**, **IR**, ...);
  - unità di controllo o **Control Unit**, che può essere:
    - microprogrammata
    - cablata



# Unità di Controllo: cablata o microprogrammata?

## □ Unità di Controllo Microprogrammata:

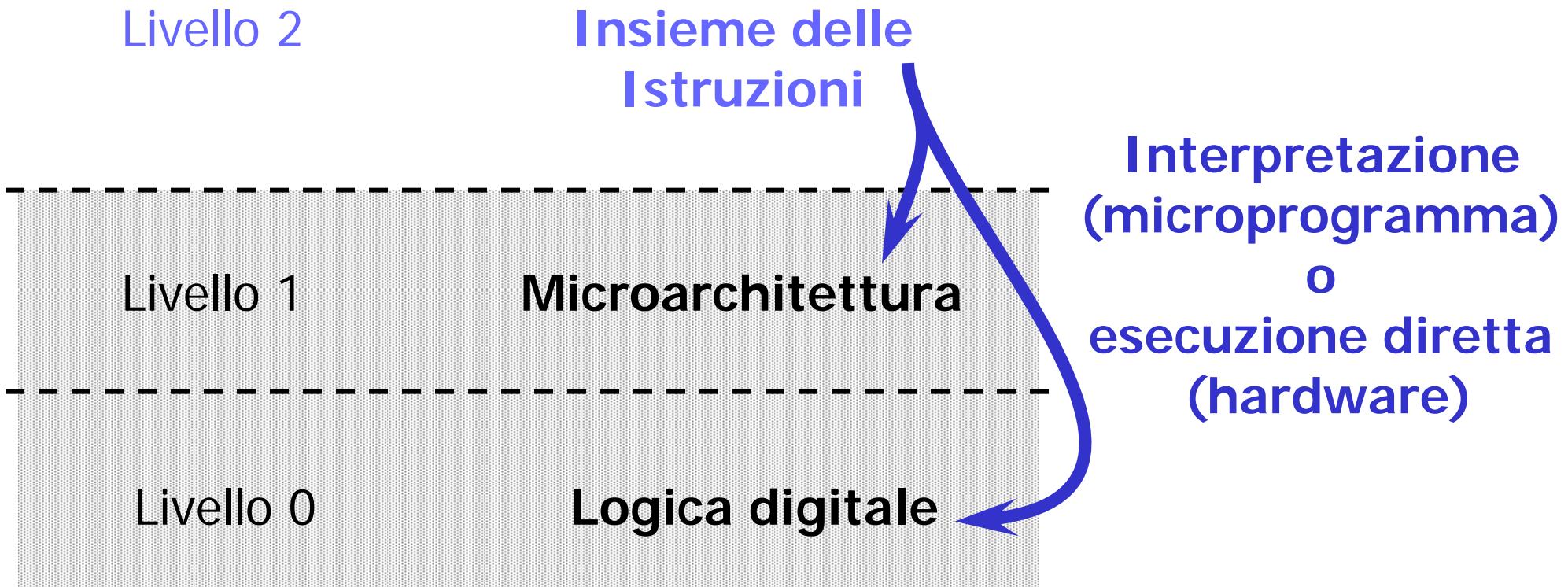
- il funzionamento dell'**unità di calcolo** viene controllato da un programma chiamato microprogramma;
- il microprogramma interpreta le istruzioni di livello 2, traducendole in operazioni eseguibili a livello 1.

## □ Unità di Controllo Cablata:

- il funzionamento dell'**unità di calcolo** viene controllato direttamente tramite dispositivi hardware;
- la sequenza di operazioni associate alle istruzioni di livello 2 non viene generata da un interprete, ma viene gestita direttamente via hardware.



# Livello 2: Instruction Set Architecture (ISA) (Struttura dell'Insieme delle Istruzioni)





## Livello 2: Instruction Set Architecture (ISA)

- ❑ Insieme delle istruzioni che possono essere comprese dalla  **$\mu$ -architettura** (la  $\mu$ -architettura agisce da interprete dell'Insieme delle Istruzioni).
- ❑ È il livello cui si fa riferimento quando si descrive il “**linguaggio macchina**” d'un calcolatore.



# Livello 3: il sistema operativo (S.O.)

Livello 3

**Sistema operativo**

Livello 2

**Insieme istruz.**

**Interpretazione  
parziale**

Livello 1

**Microarchitettura**

Livello 0

**Logica digitale**



# Livello 3: il sistema operativo

- Livello “ibrido”:
  - comprende molte istruzioni che si trovano già al livello 2;
  - comprende anche un insieme d’istruzioni aggiuntive;
  - ha una diversa organizzazione della memoria;
  - esegue più programmi contemporaneamente.
- Le nuove funzionalità sono eseguite da un interprete che viene definito “sistema operativo”.
- Le istruzioni identiche a quelle del livello 2 vengono eseguite direttamente dalla microarchitettura.

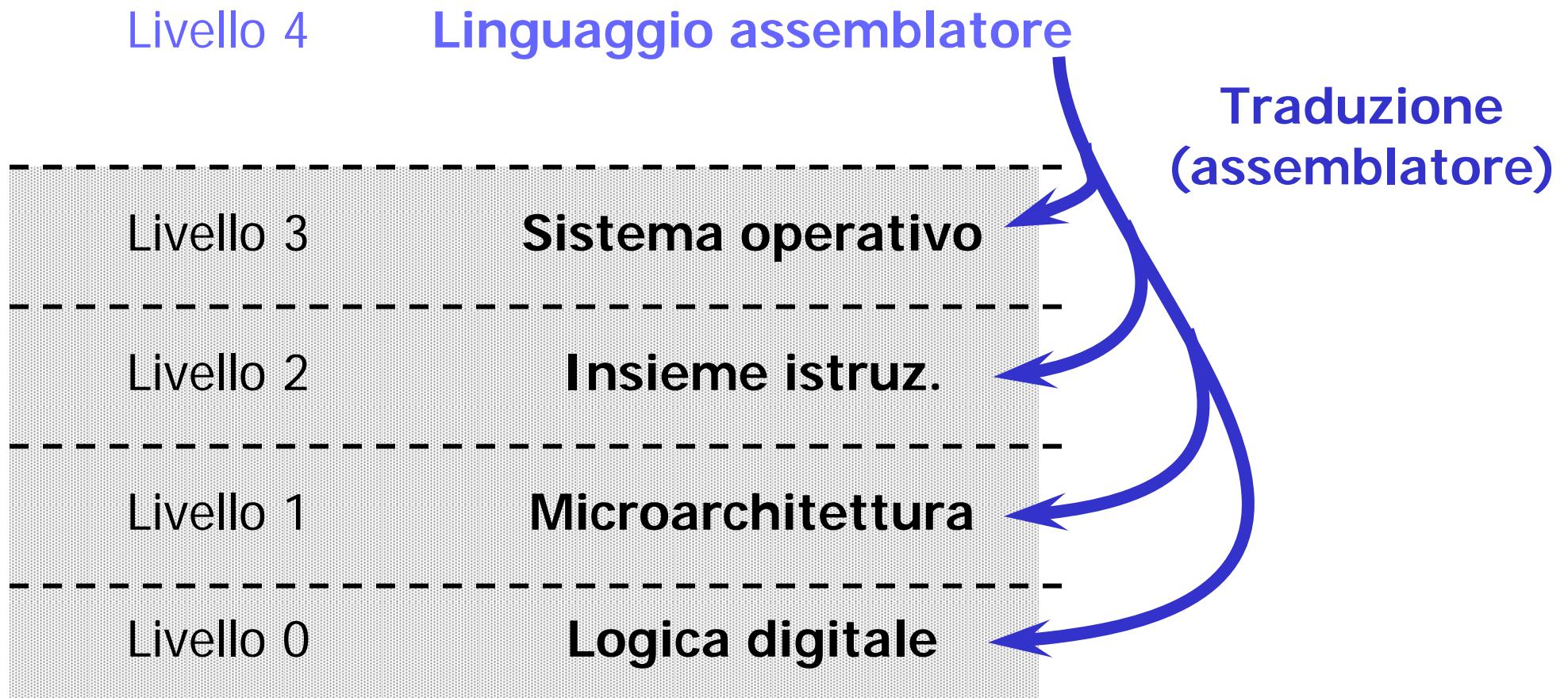


# Livelli bassi (1, 2, 3) vs livelli alti (4, 5)

- ❑ I livelli bassi supportano il funzionamento dei **compilatori** e degli **interpreti** utilizzati ai livelli alti:
  - livelli **bassi**               $\Leftrightarrow$  programmazione di **sistema**
  - livelli **alti**               $\Leftrightarrow$  programmazione d'**applicazioni**
- ❑ Per migliorare le prestazioni:
  - i **livelli 2 e 3** vengono sempre **interpretati**;
  - i **livelli 4 e 5** vengono (quasi) sempre **compilati**.
- ❑ Per questioni d'efficienza e di “usabilità”:
  - i linguaggi dei **livelli 2 e 3** sono **in codifica binaria**;
  - i linguaggi dei **livelli 4 e 5** sono **testuali (simbolici)**.



# Livello 4: il linguaggio assemblatore





# Livello 4: il linguaggio assemblatore

- Rappresentazione **simbolica** d'uno dei livelli sottostanti:
  - i linguaggi binari dei livelli "bassi" sono difficili da usare per un programmatore;
  - a **ogni** istruzione del linguaggio assemblatore corrisponde **un'**istruzione del linguaggio macchina.
- I programmi in linguaggio assemblatore vengono **tradotti** in un linguaggio di livello inferiore e poi eseguiti.  
Il programma che esegue la traduzione si chiama **assemblatore**.



# Livello 5: i linguaggi applicativi



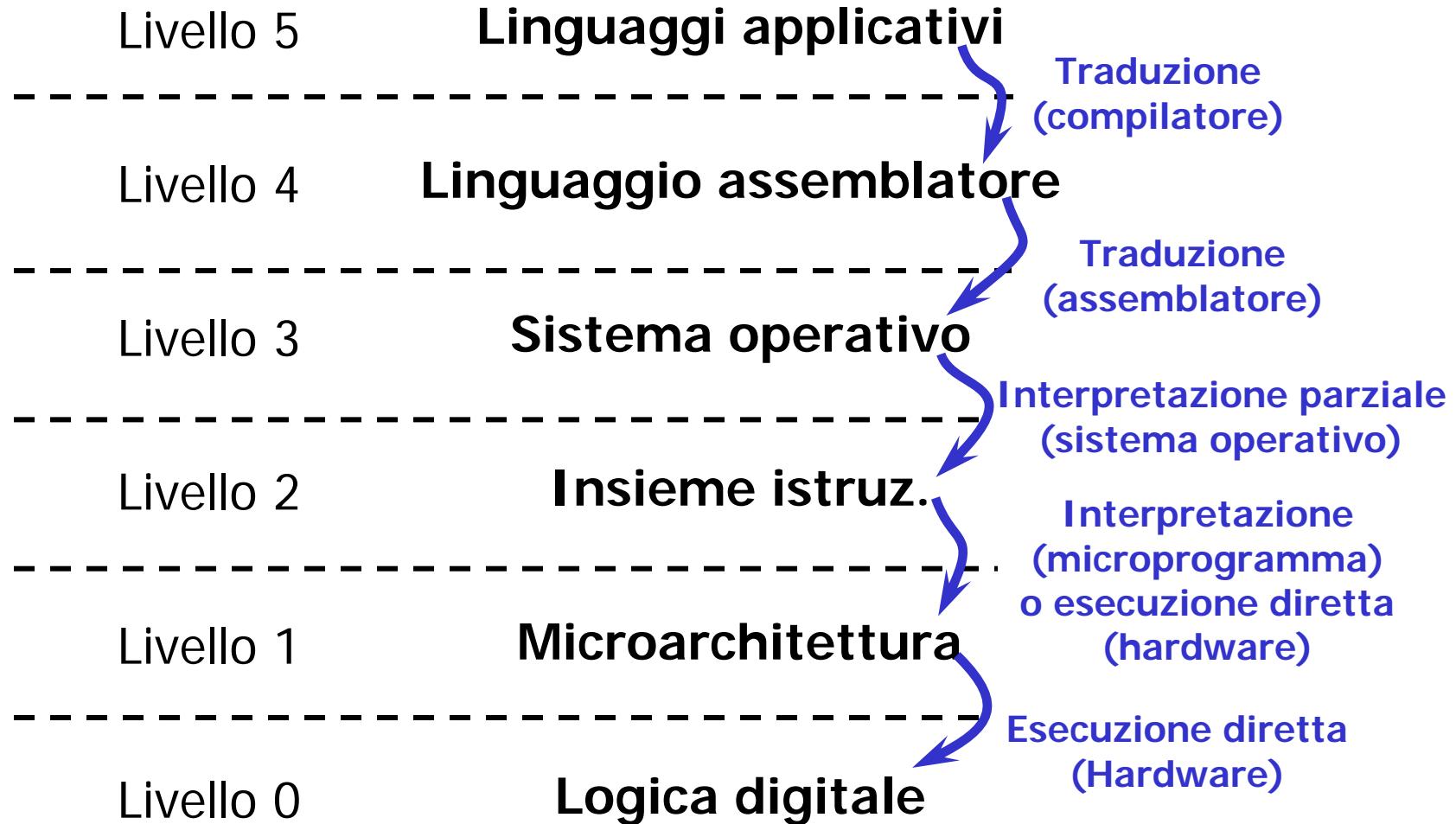


# Livello 5: i linguaggi applicativi

- Linguaggi d'alto livello come:
  - C, C++, Java, BASIC, LISP, ProLog, ...
- Sono utilizzati per la realizzazione di **programmi applicativi**.
- Il più delle volte la traduzione è affidata a un **compilatore**, mentre in alcuni casi s'usa un **interprete** (p. es. Java).



# I livelli: uno schema riassuntivo





**POLITECNICO**  
MILANO 1863

# Architettura dei calcolatori e sistemi operativi

## Architettura MIPS e set istruzioni Capitolo 2 P&H

# Instruction Set Architecture – ISA

Linguaggio assemblatore e linguaggio macchina

ISA processore MIPS

- Modello di memoria
- Registri
- Istruzioni macchina e tipi di formati
- Modalità di indirizzamento



POLITECNICO MILANO 1863

2

# Instruction Set Architecture - *ISA*

E' la descrizione del calcolatore riferita al suo linguaggio macchina, cioè all'insieme delle istruzioni (*instruction set*) che possono essere interpretate direttamente dall'architettura del processore

programma in linguaggio macchina = rappresentazione del programma per essere eseguito su un processore

Ogni architettura di processore ha il suo linguaggio macchina

- Architettura definita dall'insieme delle istruzioni  
**ISA (Instruction Set Architecture)**
- Due processori con lo stesso linguaggio macchina hanno la stessa architettura anche se le implementazioni hardware possono essere diverse



# Instruction Set Architecture – ISA (cont.)

E' necessario definire

- elementi a disposizione delle istruzioni macchina: modello della memoria e registri
- insieme delle istruzioni macchina
  - formato e dimensione
- riferimenti agli operandi
  - in memoria e/o nei registri del processore
  - tipi di indirizzamento
- tipi di dati e dimensioni
- modalità operative



# Linguaggio assemblatore e linguaggio macchina

Il linguaggio assemblatore è il **linguaggio simbolico** che consente di programmare un calcolatore utilizzando le istruzioni del linguaggio macchina

- le istruzioni del linguaggio assemblatore sono in corrispondenza (quasi) uno a uno con quelle linguaggio macchina
- la notazione simbolica consente di rappresentare istruzioni, registri, dati e riferimenti alla memoria

La questione delle *pseudo-istruzioni*

Useremo il linguaggio simbolico mettendo in evidenza le differenze tra linguaggio macchina e linguaggio assemblatore



# Linguaggio assemblatore

Per poter essere eseguito un programma scritto in assembler deve essere **tradotto in linguaggio macchina** in modo da tradurre i codici mnemonici delle istruzioni in codici operativi, sostituire tutti i riferimenti simbolici degli indirizzi con la loro forma binaria e riservare spazio di memoria per le variabili

- l'operazione di traduzione viene eseguita dall'**ASSEMBLATORE**
  - se nel codice sorgente sono presenti
    - riferimenti simbolici definiti in moduli (file) esterni a quello assemblato
    - riferimenti simbolici che dipendono dalla rilocazione del modulo
- è necessario anche il **LINKER**



# Linguaggio assembler: traduzione

**Programma  
in linguaggio  
assembler (MIPS)**

```
add $2, $4, $2  
add $3, $3,$2  
lw $15, 4($2)  
.....
```

**Assemblatore  
e Linker**

**Programma  
in linguaggio  
macchina**

```
00000011100010101010 ...  
00000011010100011101 ...  
010010000100000011 ...  
001000100010000 .....
```



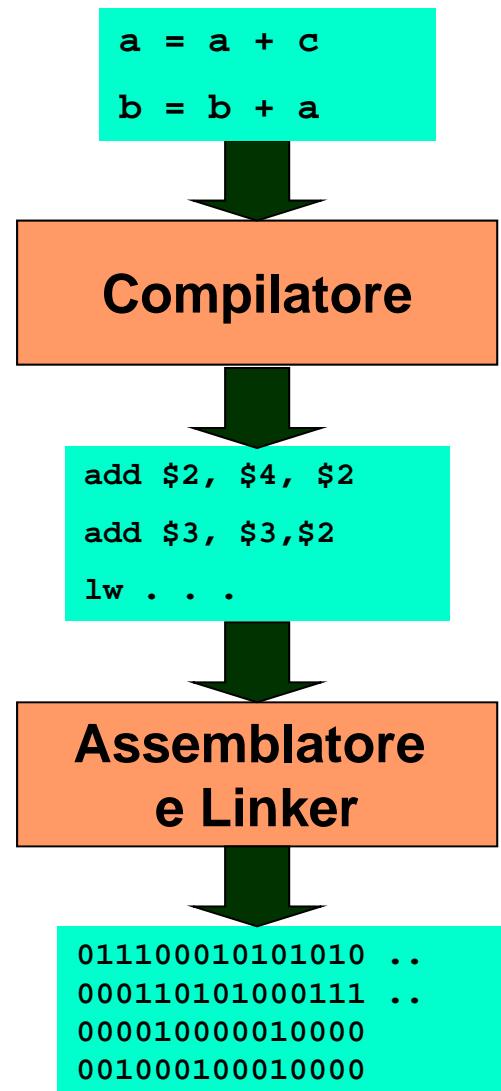
# Linguaggio C

**Programma in  
linguaggio ad alto  
livello (C)**

**Programma in  
linguaggio assembler  
(MIPS)**

assembler come linguaggio  
target della fase di compilazione

**Programma  
in linguaggio  
macchina**



# Istruzioni e variabili in linguaggio macchina

**istruzioni** - costituite da

- *codice operativo* (opcode) che identifica in modo univoco l'istruzione e definisce l'operazione associata
- *riferimento/i all'operando/i* su cui agisce l'istruzione. Il riferimento può essere
  - *implicito* nel codice operativo
  - direttamente il *valore numerico* dell'operando
  - un *registro* del processore
  - in modo diretto o indiretto una *locazione di memoria*

**variabili** - accessibili dal processore

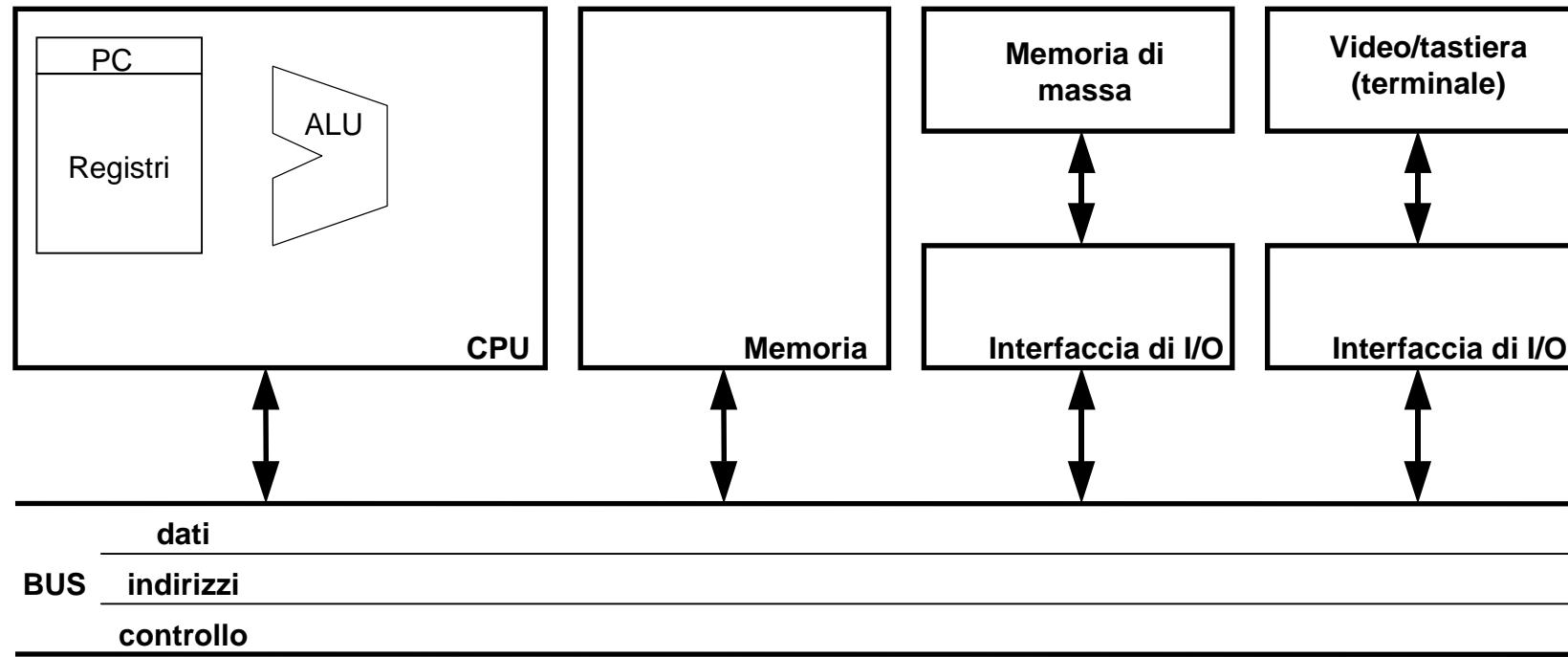
- *riferimento* rappresentato da un indirizzo di memoria (o di registro)
- *valore* contenuto nella parola di memoria associata all'indirizzo e rappresentato tramite codifica binaria opportuna



POLITECNICO MILANO 1863

9

# Architettura di riferimento dei calcolatori



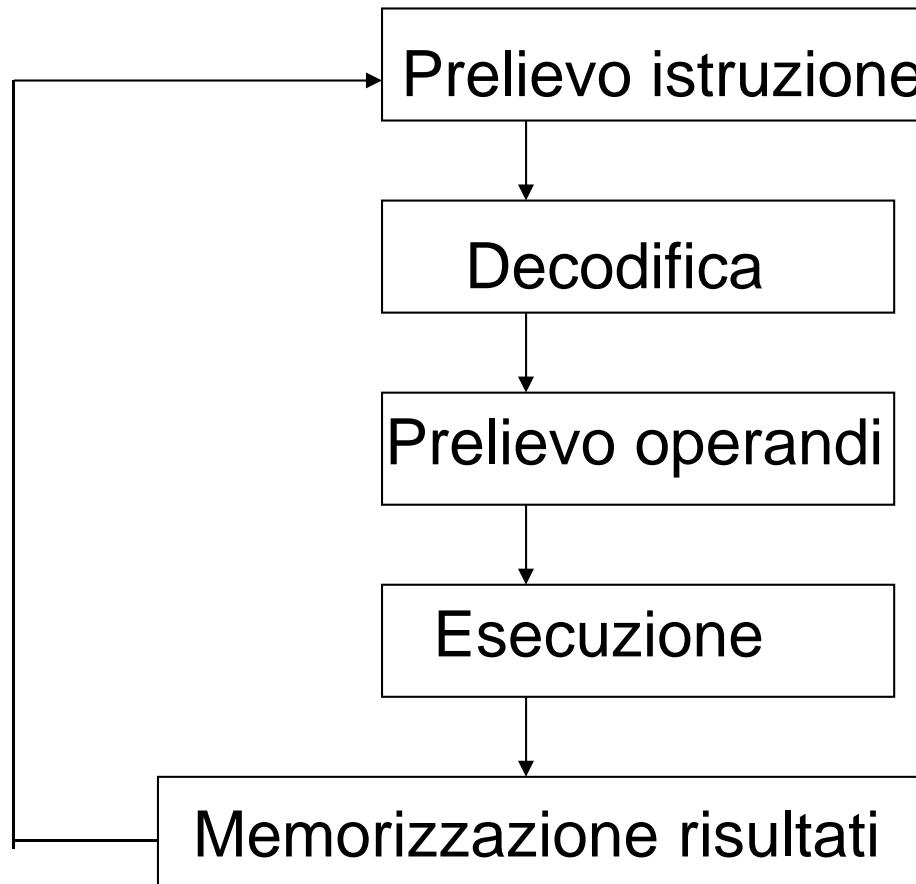
Architettura di Von Neumann



POLITECNICO MILANO 1863

10

# Esecuzione di una istruzione in linguaggio macchina



# Architettura del processore MIPS

Linguaggio assembler dell'architettura MIPS

Architettura MIPS appartiene alla famiglia delle architetture **RISC** (**Reduced Instruction Set Computer**) sviluppate dal 1980 in poi

- Esempi: Sun Sparc, HP PA-RISC, IBM Power PC, DEC Alpha, ARM

Principali obiettivi delle architetture RISC:

- Semplificare la progettazione dell'hardware e del compilatore
- Massimizzare le prestazioni
- Minimizzare i costi



# Processore di tipo RISC

CPU RISC (Reduced Instruction Set Computer)

ispirata al principio di eseguire soltanto istruzioni semplici: le operazioni complesse vengono scomposte in una serie di istruzioni più semplici da eseguire in un ciclo base ridotto, con l'obiettivo di migliorare le prestazioni ottenibili dalle *CPU CISC*

- *CPU* caratterizzata da istruzioni molto semplificate
- *CPU* relativamente semplice: si riducono i tempi di esecuzione delle singole istruzioni, che sono però meno potenti delle istruzioni *C/SC*
  - ➡ massimizzazione della frequenza di completamento dell'esecuzione delle istruzioni
- Dimensione **fissa** delle istruzioni: più semplice la gestione della fase di prelievo (fetch) e della decodifica delle istruzioni da eseguire
- Gli operandi dell'*ALU* possono provenire dai registri ma *non* dalla memoria. Per il trasferimento dei dati da memoria ai registri e viceversa si utilizzano delle apposite operazioni di caricamento (*load*) e di memorizzazione (*store*): **architetture load/store**



# Memoria – struttura e indirizzamento

## Architettura MIPS

- parole da 32 bit (4 byte), memoria indirizzabile a byte
- spazio di indirizzamento a byte 4 Gbyte (32 bit di indirizzo)
- spazio di indirizzamento a parole 1 Gparola (30 bit di indirizzo)
  - **vincolo di allineamento**: gli indirizzi di parola sono multipli di 4
- Enumerazione dei byte nella parola (indirizzamento per byte)
  - **big-endian**: l'indirizzo del byte più significativo specifica l'indirizzo di parola (byte numerati da sinistra a destra, ad es CPU Sparc, Motorola, **MIPS** ...), quindi il byte meno significativo si trova all'indirizzo più alto
  - **little-endian**: l'indirizzo del byte meno significativo specifica l'indirizzo di parola (byte numerati da destra a sinistra, ad es. Intel), quindi il byte meno significativo si trova all'indirizzo più basso
  - La modalità di enumerazione dei byte nella parola è significativa nel caso di accesso alla parola byte a byte



# Memoria: indirizzamento a byte e allineamento

	Indirizzo di byte			
Parola 0	0	1	2	3
Parola 4	4	5	6	7
Parola 8	8	9	10	11
...	MS byte			LS byte
Parola $2^k-4$	$2^k-4$	$2^k-3$	$2^k-2$	$2^k-1$

**big-endian** alla parola viene assegnato lo stesso indirizzo del suo byte più significativo

	Indirizzo di byte			
Parola 0	3	2	1	0
Parola 4	7	6	5	4
Parola 8	11	10	9	8
...	MS byte			LS byte
Parola $2^k-4$	$2^k-1$	$2^k-2$	$2^k-3$	$2^k-4$

**little-endian** alla parola viene assegnato lo stesso indirizzo del suo byte meno significativo



# Registri

Sono elementi di memoria interni alla CPU: a livello ISA interessano tutti e soli i **registri referenziabili** nelle istruzioni macchina

## Architettura MIPS

- 32 registri da 32 bit, accessibili anche a byte, organizzati in un banco di registri (*Register File*)
  - referenziabili con nome simbolico (preceduto da \$) nelle istruzioni assembler e associati univocamente a un «numero» tra 0 e 31
  - identificabili (indirizzabili) con 5 bit in linguaggio macchina
  - possono avere la loro specificità in termini di utilizzo, ma sono trattati in modo omogeneo
- 3 registri da 32 bit non referenziabili in ISA e di uso specifico



## Registri referenziabili



Nome	Numero	Utilizzo
\$0	0	Costante 0
\$at	1	Riservato all'assemblatore
\$v0-\$v1	2-3	Valori restituiti da una funzione e risultati calcolo espressioni
\$a0-\$a3	4-7	Argomenti
\$t0-\$t7	8-15	Variabili temporanee
\$s0-\$s7	16-23	Variabili da preservare
\$t8-\$t9	24-25	Altre variabili temporanee
\$k0-\$k1	26-27	Riservati al kernel del sistema operativo
\$gp	28	Global Pointer (puntatore area dati globali/statici)
\$sp	29	Stack Pointer (puntatore allo stack – prima piena)
\$fp	30	Frame Pointer (puntatore frame funzione)
\$ra	31	Return Address (utilizzato nelle chiamate a funzione)

## Non referenziabili

Registri non referenziabili	
pc	Program Counter
hi	Registro per moltiplicazioni e divisioni
lo	Registro per moltiplicazioni e divisioni



POLITECNICO MILANO 1863

17

# Insieme delle istruzioni

Classi di istruzioni tipiche in linguaggio macchina:

- aritmetico-logiche
- trasferimento da e in memoria (e tra registri)
- modifica del flusso di esecuzione:
  - salto condizionato, incondizionato
  - salto a sottoprogramma, ritorno da sottoprogramma
- trasferimento dati da e in periferica (istruzioni di I/O)
- istruzioni speciali (controllo)



# Formato delle istruzioni MIPS

Tutte le **istruzioni macchina** MIPS hanno la **stessa dimensione (32 bit)** e quindi occupano una sola parola di memoria

I 32 bit hanno un significato diverso a seconda del formato (o tipo) di istruzione

- il tipo di istruzione è riconosciuto in base al **codice operativo** dell'istruzione (**OPCODE**): **6 bit** più significativi dell'istruzione MIPS
  - OPCODE definisce non solo «che cosa fa» l'istruzione ma anche come «utilizzare» i bit rimanenti

Le istruzioni MIPS sono di **3** formati (tipi):

## **Tipo R (register)**

- Istruzioni aritmetico-logiche

## **Tipo I (immediate)**

- Istruzioni di accesso alla memoria o contenenti valori costanti

## **Tipo J (jump)**

- Istruzioni di salto (incondizionato)



# Modalità di indirizzamento in MIPS

Le modalità di indirizzamento previste in linguaggio macchina MIPS sono:

- Immediato
- A registro
- Con base e spiazzamento
- Relativo al Program Counter
- Pseudo-diretto (rispetto al Program Counter)

La **modalità di indirizzamento** è associata in modo univoco al **formato dell'istruzione** (cioè a come vengono interpretati i bit nell'istruzione in linguaggio macchina)

- **Tipo R (register):** a registro
- **Tipo I (immediate):** immediato, con base e spiazzamento, relativo al Program Counter
- **Tipo J (jump):** psuedo-diretto

Una singola istruzione «logica» può usare più di una modalità di indirizzamento, ad es. **add** e **addi** (che sono di due formati diversi, R e I rispettivamente e hanno anche due codici operativi diversi)

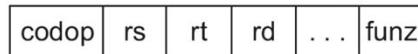


# Modalità di indirizzamento in MIPS (2)

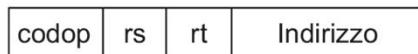
1. Indirizzamento immediato



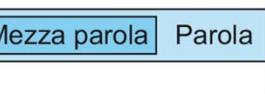
2. Indirizzamento tramite registro



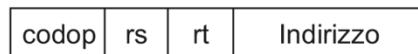
3. Indirizzamento tramite base



Memoria



4. Indirizzamento relativo al PC

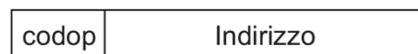


Memoria



Parola

5. Indirizzamento pseudodiretto



Memoria



Parola

Operando  
nell'istruzione

Operando  
in  
registro

Operando  
in  
memoria

Operando  
in  
memoria

Operando  
in  
memoria

→ Riferimento  
a dati

Riferimento  
a istruzioni



POLITECNICO MILANO 1863

21



**POLITECNICO**  
MILANO 1863

# **Architettura dei calcolatori e sistemi operativi**

**Set istruzioni e struttura del programma  
Direttive all'Assemblatore**

**Capitolo 2 P&H**

# Sommario

Istruzioni

Formati istruzioni

Struttura del programma e direttive all'assemblatore



POLITECNICO MILANO 1863

2

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# Sottoinsieme del linguaggio assembler MIPS

Operandi MIPS

Nome	Esempio	Commenti
32 registri	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Accesso veloce ai dati. Nel MIPS gli operandi devono essere contenuti nei registri per potere eseguire delle operazioni. Il registro \$zero contiene sempre il valore 0, e il registro \$at viene riservato all'assemblatore per la gestione di costanti molto lunghe.
$2^{30}$ parole di memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Alla memoria si accede solamente attraverso le istruzioni di trasferimento dati. Il MIPS utilizza l'indirizzamento al byte, perciò due parole consecutive hanno indirizzi in memoria a una distanza di 4. La memoria consente di memorizzare strutture dati, vettori, o il contenuto dei registri.

Linguaggio assembler MIPS

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Aritmetiche	Somma	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Operandi in tre registri
	Sottrazione	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Operandi in tre registri
	Somma immediata	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Utilizzata per sommare delle costanti
Trasferimento dati	Lettura parola	lw \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una parola da memoria a registro
	Memorizzazione parola	sw \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di una parola da registro a memoria
	Lettura mezza parola	lh \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una mezza parola da memoria a registro
	Lettura mezza parola, senza segno	lhu \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di una mezza parola da memoria a registro
	Memorizzazione mezza parola	sh \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di una mezza parola da registro a memoria
	Lettura byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di un byte da memoria a registro
	Lettura byte senza segno	lbu \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Trasferimento di un byte da memoria a registro
	Memorizzazione byte	sb \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1$	Trasferimento di un byte da registro a memoria
	Lettura di una parola e blocco	ll \$s1,20(\$s2)	$\$s1 = \text{Memoria}[\$s2+20]$	Caricamento di una parola come prima fase di un'operazione atomica
	Memorizzazione condizionata di una parola	sc \$s1,20(\$s2)	$\text{Memoria}[\$s2+20] = \$s1; \$s1 = 0 \text{ oppure } 1$	Memorizzazione di una parola come seconda fase di un'operazione atomica
Logiche	Caricamento costante nella mezza parola superiore	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Caricamento di una costante nei 16 bit più significativi
	And	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Operandi in tre registri; AND bit a bit
	Or	or \$s1,\$s2,\$s3	$\$s1 = \$s2   \$s3$	Operandi in tre registri; OR bit a bit
	Nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2   \$s3)$	Operandi in tre registri; NOR bit a bit

Tipo di istruzioni	Istruzioni	Esempio	Significato	Commenti
Logiche (segue)	And immediato	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	And bit a bit tra un operando in registro e una costante
	Or immediato	ori \$s1,\$s2,20	$\$s1 = \$s2   20$	OR bit a bit tra un operando in registro e una costante
	Scorrimento logico a sinistra	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Spostamento a sinistra del numero di bit specificato dalla costante
	Scorrimento logico a destra	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Spostamento a destra del numero di bit specificato dalla costante
Salvi condizionati	Salta se uguale	beq \$s1,\$s2,25	Se $(\$s1 == \$s2)$ vai a PC+4+100	Test di uguaglianza; salto relativo al PC
	Salta se non è uguale	bne \$s1,\$s2,25	Se $(\$s1 != \$s2)$ vai a PC+4+100	Test di disuguaglianza; salto relativo al PC
	Poni uguale a 1 se minore	slt \$s1,\$s2,\$s3	Se $(\$s2 < \$s3) \$s1 = 1;$ altrimenti $\$s1 = 0$	Comparazione di minoranza; utilizzata con bne e beq
	Poni uguale a uno se minore, numeri senza segno	sltu \$s1,\$s2,\$s3	Se $(\$s2 < \$s3) \$s1 = 1;$ altrimenti $\$s1 = 0$	Comparazione di minoranza su numeri senza segno
	Poni uguale a uno se minore, immediato	slti \$s1,\$s2,20	Se $(\$s2 < 20) \$s1 = 1;$ altrimenti $\$s1 = 0$	Comparazione di minoranza con una costante
	Poni uguale a uno se minore, immediato e senza segno	sltiu \$s1,\$s2,20	Se $(\$s2 < 20) \$s1 = 1;$ altrimenti $\$s1 = 0$	Comparazione di minoranza con una costante, con numeri senza segno
	Salto incondizionato	j 2500	Vai a 10000	Salto all'indirizzo della costante
	Salto indiretto	jr \$ra	Vai all'indirizzo contenuto in \$ra	Salto all'indirizzo contenuto nel registro, utilizzato per il ritorno da procedura e per i costrutti switch
	Salta e collega	jal 2500	$\$ra = PC+4;$ vai a 10000	Chiamata a procedura



POLITECNICO MILANO 1863

# Istruzioni aritmetico-logiche

In MIPS, un'istruzione aritmetico-logica ha *tre* operandi

- due **registri sorgente** contenenti i valori da elaborare
- un **registro destinazione** contenente il risultato

È quindi di tipo R e l'ordine degli operandi è **fisso**

- In assemblatore il formato istruzione è

**OPCODE    DEST,    SORG1,    SORG2**

where **DEST, SORG1, SORG2** sono registri referenziabili del MIPS



POLITECNICO MILANO 1863

4

# Istruzioni aritmetico-logiche (2)

Istruzioni di somma e sottrazione

```
add rd, rs, rt      # rd ← rs + rt
```

```
sub rd, rs, rt      # rd ← rs - rt
```

**addu** e **subu** lavorano in modo analogo su operandi senza segno (indirizzi) e non generano segnalazione di traboccamento (overflow)

Istruzioni logiche **and**, **or** e **nor** lavorano in modo analogo

```
or rd, rs, rt      # rd ← rs or rt (OR bit a bit)
```

Istruzioni di scorrimento logico

```
sll rd, rs, 10      # rd ← rs << 10
```

```
srl rd, rs, 10      # rd ← rs >> 10
```



## Varianti con operando immediato (costante)

**addi rd, rs, imm** #  $rd \leftarrow rs + imm$

**addiu rd, rs, imm** #  $rd \leftarrow rs + imm$  (no overflow)



# Qualche esempio

Codice C:

$$R = A + B$$

Codice MIPS:

```
add $s0, $s1, $s2
```

nella traduzione da C  
a linguaggio assemblatore  
le variabili sono state associate  
ai registri dal compilatore

Il fatto che ogni istruzione aritmetica abbia tre operandi sempre nella stessa posizione consente di semplificare lo HW, ma complica alcune cose...

Codice C:

$$A = B + C + D$$

$$E = F - A$$

Codice MIPS:

```
add $t0, $s1, $s2
```

```
add $s0, $t0, $s3
```

```
sub $s4, $s5, $s0
```

A	\$s0
B	\$s1
C	\$s2
D	\$s3
E	\$s4
F	\$s5



## Qualche esempio (2)

Espressioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici

Per esempio

*Codice C:*       $A = B + C + D + E$

*Codice MIPS:*    `add $t0, $s1, $s2`  
                      `add $t0, $t0, $s3`  
                      `add $s0, $t0, $s4`



# Istruzioni di trasferimento dati: *Load / Store*

MIPS fornisce due operazioni base per il trasferimento dei dati:

- **lw** (load word) per trasferire una parola di memoria in un registro
- **sw** (store word) per trasferire il contenuto di un registro in una parola di memoria

*lw* e *sw* hanno **due operandi**

- il registro destinazione (*lw*) o sorgente (*sw*) del trasferimento dei dati
- la parola di memoria coinvolta nel trasferimento (identificata dal suo indirizzo)



# Istruzioni di trasferimento dati: *Load / Store* (2)

In **linguaggio macchina** MIPS l'indirizzo della parola di memoria coinvolta nel trasferimento viene sempre specificato secondo la modalità

*offset(registro\_base)* dove

- offset è intendersi come spiazzamento su 16 bit e
- l'indirizzo della parola di memoria è dato dalla somma tra il valore immediato *offset* e il contenuto del *registro base*

In linguaggio macchina le istruzioni lw / sw hanno quindi **tre argomenti**

- **registro coinvolto nel trasferimento**
- **offset** valore immediato (su 16 bit) che rappresenta lo spiazzamento rispetto al registro base, e può valere anche 0
- **registro base**



# Istruzioni di trasferimento dati: *Load / Store* (3)

In **linguaggio assemblatore** MIPS l'indirizzo della parola di memoria coinvolta nel trasferimento può essere espresso in modo più *flessibile* come somma di

*identificatore + espressione + registro*

dove ciascuna parte può essere omessa e

- l'identificatore è costituito da un **simbolo rilocabile** che si riferisce all'area dati che contiene le *variabili globali* del programma; se è l'identificatore simbolico di una variabile scalare globale, la parte di indirizzo che deriva dall'identificatore è calcolata come **(\$gp) + offset** della variabile in area dati globale; il valore dell'offset è calcolato dal collegatore (linker)
- l'espressione può essere anche una costante con segno (**offset**)
- il registro è il **registro base**

```
lw $t0, ($a0)      # $t0 ← M[$a0 + 0]
lw $t0, 20($a0)    # $t0 ← M[$a0 + 20]
lw $t0, var1       # $t0 ← M[$gp + offset di var1 area dati globale]
```



## Qualche esempio

```
lw $s1, 100($s2)      # $s1 ← M[$s2 + 100]  
sw $s1, 100($s2)      # M[$s2 + 100] ← $s1
```

Codice C:      **A[12] = h + A[8];**

- variabile **h** associata al registro **\$s2**
- indirizzo del primo elemento dell'array **A** (*base address*) contenuto nel registro **\$s3**

Codice MIPS:

```
lw    $t0, 32($s3)      # $t0 ← M[$s3 + 32]  
add  $t0, $s2, $t0       # $t0 ← $s2 + $t0  
sw    $t0, 48($s3)      # M[$s3 + 48] ← $t0
```



## Ancora sugli array (vettori)

Sia A un array di 100 interi su 32 bit

Istruzione C:  $g = h + A[i]$

- le variabili **g**, **h**, **i** siano associate rispettivamente ai registri **\$s1**, **\$s2**, ed **\$s4**
- l'indirizzo del primo elemento dell'array (*base address*) sia contenuto nel registro **\$s3**
- l'elemento **i-esimo** di un array si troverà nella locazione **base\_address + 4 × i**
  - il fattore **4** dipende dall'indirizzamento al byte della memoria nel MIPS



## Ancora sugli array (cont.)

L'elemento  $i$ -esimo dell'array si trova nella locazione di memoria di indirizzo  
 $(\$s3 + 4 \times i)$

- *indirizzo di  $A[i]$  nel registro temporaneo  $\$t1$*

```
add $t1, $s4, $s4    # $t1 ← 2 × i
add $t1, $t1, $t1    # $t1 ← 4 × i
add $t1, $t1, $s3    # $t1 ← add. of A[i]
                      # that is ($s3 + 4 × i)
```

Oppure

```
sll $t1, $s4, 2      # $t1 ← 4 × i
add $t1, $t1, $s3
```

- *$A[i]$  nel registro temporaneo  $\$t0$*

```
lw $t0, 0($t1)       # $t0 ← A[i]
```

- *somma di  $h$  e  $A[i]$  e risultato in  $g$*

```
add $s1, $s2, $t0     # g = h + A[i]
```



# Costrutti di controllo e istruzioni di salto

Istruzioni di salto condizionato / incondizionato

- alterano l'ordine di esecuzione delle istruzioni
  - la prossima istruzione da eseguire non è necessariamente l'istruzione successiva all'istruzione corrente, ma quella individuata dalla **destinazione del salto**
- permettono di realizzare i costrutti di controllo condizionali e ciclici



# Istruzioni di salto

In linguaggio assembler si specifica l'indirizzo dell'istruzione destinazione di salto tramite un *nome simbolico* che è costituito da un'**etichetta (label)** associata appunto all'istruzione destinazione

In *linguaggio macchina* MIPS

- le istruzioni di salto *condizionato* sono di tipo **I**: la modalità di indirizzamento è quella *relativa al PC* ( $\text{new\_PC} = \text{PC} + \text{offset}$ ) con spiazzamento (offset) di 16 bit
- le istruzioni di salto *incondizionato* sono di tipo **J**: la modalità di indirizzamento è quella *pseudo-diretta*

Quindi partendo da **etichetta** l'operazione di traduzione in linguaggio macchina dell'indirizzo destinazione sarà diversa nei due casi



# Istruzioni di salto condizionato

Istruzioni di **salto condizionato (conditional branch)**: il salto viene eseguito solo se una certa condizione risulta soddisfatta

**beq** (*branch on equal*)

**bne** (*branch on not equal*)

```
beq r1, r2, label_1      # go to label_1 if (r1 == r2)
```

```
bne r1, r2, label_1      # go to label_1 if (r1 != r2)
```

È possibile utilizzarle insieme ad altre istruzioni per realizzare salti condizionati su esito di maggioranza o minoranza (vedi più avanti)



# Istruzioni di salto incondizionato

Istruzioni di **salto incondizionato ( unconditional jump)**:  
il salto viene sempre eseguito

- j        (jump)
- jr      (jump register – ritorno da sottoprogramma)
- jal     (jump and link – chiamata di sottoprogramma)

```
j L1          # go to L1
jr $ra         # go to address contained in $ra
jal L1         # go to L1. Save add. of next
                # instruction in reg. $ra
```



## Esempio if ... then ... else

Codice C:

```
if (i == j) f = g + h;  
else f = g - h;
```

Si suppone che le variabili **f**, **g**, **h**, **i** e **j** siano associate rispettivamente ai registri **\$s0**, **\$s1**, **\$s2**, **\$s3** e **\$s4**

Codice MIPS:

```
bne    $s3, $s4, Else      # go to ELSE if i≠j  
add    $s0, $s1, $s2       # f=g+h (skipped if i ≠ j)  
j      END_IF             # go to END_IF  
ELSE:  sub $s0, $s1, $s2   # f=g-h (skipped if i = j)  
END_IF: ...
```



# Condizioni di salto

registro **\$zero**

spesso la verifica di uguaglianza richiede il confronto con il valore 0 per rendere più veloce il confronto, in MIPS il registro **\$zero** contiene il valore 0 e non può mai essere utilizzato per contenere altri valori

Il processore tiene traccia di alcune informazioni sui risultati di operazioni per usarle nelle condizioni di successive istruzioni di salto condizionato

- queste informazioni sono memorizzate in bit o flag denominati *codici di condizione*
- il registro di stato o registro dei codici di condizione contiene i flag dei codici di condizione

I codici di condizione più usati sono:

N (negativo)	# posto a 1 se il risultato è negativo; # altrimenti posto a 0
Z (zero)	# posto a 1 se il risultato è zero; # altrimenti posto a 0
V (overflow)	# posto a 1 se si verifica un overflow aritmetico; # altrimenti posto a 0
C (riporto)	# posto a 1 se dall'operazione risulta un riporto; # altrimenti posto a 0



## Ancora sulle istruzioni di salto condizionato

Spesso è utile condizionare l'esecuzione di una istruzione al fatto che una variabile sia minore di una altra

```
slt $s1, $s2, $s3      # set on less than
```

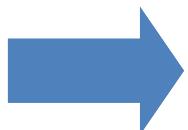
assegna il valore **1** a **\$s1** se **\$s2 < \$s3**; altrimenti assegna il valore **0**

Con **slt**, **beq** e **bne** si possono realizzare i test sui valori di due variabili (**=, !=, <, <=, >, >=**)



## Esempio

```
if (i < j)  k = i + j;  
else k = i - j;
```



```
#$s0 ed $s1 contendono i e j  
#$s2 contiene k
```

```
    slt $t0, $s0, $s1  
    beq $t0, $zero, ELSE  
    add $s2, $s0, $s1  
    j EXIT  
ELSE: sub $s2, $s0, $s1  
EXIT:
```



# Pseudo-istruzioni

Per semplificare la programmazione, MIPS fornisce un insieme di *pseudo-istruzioni*

- le pseudoistruzioni sono un modo compatto e intuitivo di specificare *un insieme di istruzioni*
- la traduzione della pseudo-istruzione nelle istruzioni equivalenti è attuata automaticamente dall'assemblatore

```
move $t0, $t1
    ▪ add $t0, $zero, $t1
```

```
mul $s0, $t1, $t2
    ▪ mult $t1, $t2
    ▪ mflo $s0
```



# ISTRUZIONI E PSEUDOISTRUZIONI

## ARITMETICA

add	\$s1, \$s1, \$s3	s1 := s2 + s3	addizione
addu	\$s1, \$s1, \$s3	s1 := s2 + s3	addizione naturale
addi	\$s1, \$s2, cost	s1 := s2 + cost	addizione di costante
addiu	\$s1, \$s2, cost	s1 := s2 + cost	addizione naturale di costante
sub	\$s1, \$s2, \$s3	s1 := s2 - s3	sottrazione
subu	\$s1, \$s2, \$s3	s1 := s2 - s3	sottrazione naturale

## ARITMETICA – pseudoistruzioni

subi	\$s1, \$s2, cost	s1 := s2 - cost	sottrazione di costante
subiu	\$s1, \$s2, cost	s1 := s2 - cost	sottrazione naturale di costante
neg	\$s1, \$s2	s1 := -s2	negazione aritmetica

## CONFRONTO

slt	\$s1, \$s2, \$s3	<b>if</b> s2 < s3 <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se minore stretto
sltu	\$s1, \$s2, \$s3	<b>if</b> s2 < s3 <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se minore str. nat.
slti	\$s1, \$s2, cost	<b>if</b> s2 < cost <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se minore str. cost.
sltiu	\$s1, \$s2, cost	<b>if</b> s2 < cost <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se min. str. cost. nat.



# ISTRUZIONI E PSEUDOISTRUZIONI

## LOGICA

or	\$s1, \$s2, \$s3	s1 := s2 <b>or</b> s3	somma logica bit a bit
and	\$s1, \$s2, \$s3	s1:= s2 <b>and</b> s3	prodotto logico bit a bit
ori	\$s1, \$s2, cost	s1:= s2 <b>or</b> cost	somma logica bit a bit costante
andi	\$s1, \$s2, cost	s1:= s2 <b>or</b> cost	prodotto logico bit a bit costante
nor	\$s1, \$s2, \$s3	s1:= s2 <b>nor</b> s3	somma logica negata bit a bit
sll	\$s1, \$s2, cost	s1:= s2 << cost	scorrimento a sinistra (left) del n° di bit specificato da cost
srl	\$s1, \$s2, cost	s1:= s2 >> cost	scorrimento a destra (right) del n° di bit specificato da cost

## LOGICA – pseudoistruzioni

not	\$s1, \$s2	s1 = <b>not</b> s2	(p) negazione logica
-----	------------	--------------------	----------------------



# ISTRUZIONI E PSEUDOISTRUZIONI

## SALTO INCONDIZIONATO E CON COLLEGAMENTO

j	indir	<b>PC</b> := cost (28 bit)	salto incondizionato assoluto
jr	\$r	<b>PC</b> := r (32 bit)	salto indiretto da registro
jal	indir	<b>PC</b> := cost (28 bit) e collega \$ra	salto assoluto e collegamento

## SALTO CONDIZIONATO

beq	\$s1, \$s2, spi	<b>if</b> s2 = s1 salta rel. a PC	salto cond. di uguaglianza
bne	\$s1, \$s2, spi	<b>if</b> s2 ≠ s1 salta rel. a PC	salto cond. di disuguaglianza

## SALTO CONDIZIONATO - pseudoistruzioni

blt	\$s1, \$s2, spi	<b>if</b> s2 < s1 salta rel. a PC	salta se minore stretto
bgt	\$s1, \$s2, spi	<b>if</b> s2 > s1 salta rel. a PC	salta se maggiore stretto
ble	\$s1, \$s2, spi	<b>if</b> s2 ≤ s1 salta rel. a PC	salta se minore o uguale
bge	\$s1, \$s2, spi	<b>if</b> s2 ≥ s1 salta rel. a PC	salta se maggiore o uguale



# ISTRUZIONI E PSEUDOISTRUZIONI

## TRASFERIMENTO MEMORIA

lw	\$s1, spi (\$s2)	s1 := mem (s2 + spi)	carica parola (a 32 bit)
sw	\$s1, spi (\$s2)	mem (s2 + spi) := s1	memorizza parola (a 32 bit)
lh, lhu	\$s1, spi (\$s2)	s1:= mem (s2 + spi)	carica mezza parola (a 16 bit)
sh	\$s1, spi (\$s2)	mem (s2 + spi) := s1	memor. mezza parola (a 32 bit)
lb, lbu	\$s1, spi (\$s2)	s1:= mem (s2 + spi)	carica byte (a 8 bit)
sb	\$s1, spi (\$s2)	mem (s2 + spi) := s1	memorizza byte (a 8 bit)

## TRASFERIMENTO in registro di COSTANTE

lui	\$s1, cost	s1 (16 bit più signif.) := cost	carica cost (in 16 bit più signifi)
-----	------------	---------------------------------	-------------------------------------

## TRASFERIMENTI tra REGISTRI e di COSTANTI/INDIRIZZI – pseudo istruzioni

move	\$d, \$s	d := s	copia registro
la	\$d, indir	d := indir (32 bit)	carica indirizzo a 32 bit
li	\$d, cost	d := cost (32 bit)	carica costante a 32 bit



# REGISTRI

## REGISTRI REFERENZIABILI

0	0	costante 0
1	at	uso riservato all'assembler-linker
2-3	v0 - v1	valore restituito da funzione
4-7	a0-a3	argomenti in ingresso a funzione
8-15	t0-t7	registri per valori temporanei
16-23	s0-s7	registri
24-25	t8-t9	registri per valori temporanei (in aggiunta a t0-t7), come i precedenti t
26-27	k0-k1	registri riservati per il nucleo del SO
28	gp	global pointer (puntatore all'area dati globale)
29	sp	stack pointer (puntatore alla pila)
30	fp	frame pointer (puntatore area di attivazione)
31	ra	registro return address

## REGISTRI NON REFERENZIABILI

	pc	Program Counter
	hi	Registro per risultato moltiplicazioni e divisioni
	lo	Registro per risultato moltiplicazioni e divisioni



# ***FORMATI ISTRUZIONE***



POLITECNICO MILANO 1863

29

# Formato istruzioni di tipo R - aritmetiche

Formato usato per istruzioni aritmetico-logiche

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Ai vari campi sono stati assegnati dei nomi mnemonici:

- **op**: (**opcode**) identifica il tipo di istruzione (**0**)
- **rs**: registro contenente il primo operando sorgente
- **rt**: registro contenente il secondo operando sorgente
- **rd**: registro destinazione contenente il risultato
- **shamt**: shift amount (scorrimento)
- **funct**: indica la variante specifica dell'operazione



# Istruzioni di tipo R: esempi

add \$s1, \$s2, \$s3						
Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
<b>add \$s1, \$s2, \$s3</b>	000000	10010	10011	10001	00000	100000

Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
<b>sub \$s1, \$s2, \$s3</b>	000000	10010	10011	10001	00000	100010



# Formato istruzioni di tipo I – lw/sw

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di istruzioni load/store, i campi hanno il seguente significato:

- **op** (**opcode**) identifica il tipo di istruzione; (**35 o 43**)
- **rs** indica il registro base;
- **rt** indica il registro destinazione dell'istruzione load o il registro sorgente dell'istruzione store;
- **indirizzo** riporta lo spiazzamento (offset)

Con questo formato, un'istruzione **lw (sw)** può indirizzare parole nell'intervallo  $-2^{15} \text{ } +2^{15}-1$  rispetto all'indirizzo base



# Istruzioni di tipo I: esempi lw e sw

**lw \$t0 , 32(\$s3)**

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
<b>lw \$t0 , 32 (\$s3)</b>	100011	10011	01000	0000	0000	0010	0000

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
<b>sw \$t0 , 32 (\$s3)</b>	101011	10011	01000	0000	0000	0010	0000



# Formato istruzioni di tipo I – con immediato

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di istruzioni **con immediati**, i campi hanno il seguente significato:

- **op (opcode)** identifica il tipo di istruzione;
- **rs** indica il registro sorgente;
- **rt** indica il registro destinazione;
- **indirizzo** contiene il valore dell'operando immediato

Con questo formato, un'istruzione con immediato può contenere costanti nell'intervallo  $-2^{15}$   $+2^{15}-1$



# Istruzioni di tipo I: esempi con operando immediato

addi \$s1, \$s1, 4

Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
<b>addi \$s1, \$s1, 4</b>	001000	10001	10001	0000	0000	0000	0100
Nome campo	op	rs	rt	indirizzo			
Dimensione	6-bit	5-bit	5-bit	16-bit			
<b>slti \$t0, \$s2, 8</b>	001010	10010	01000	0000	0000	0000	1000

# \$t0 = 1 if \$s2 < 8

slti \$t0, \$s2, 8



# Formato istruzioni di tipo I - branch

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di salti condizionati, i campi hanno il seguente significato:

- **op (opcode)** identifica il tipo di istruzione (**4 = beq**)
- **rs** indica il primo registro;
- **rt** indica il secondo registro;
- **indirizzo** riporta lo spiazzamento (offset)

Per l'offset si hanno a disposizione solo 16-bit del campo **indirizzo** ⇒ rappresentano un indirizzo di **parola** relativo al PC (**PC-relative word address**)



# Istruzioni di salto condizionato (tipo I)

I 16-bit del campo indirizzo esprimono l'**offset** rispetto al PC rappresentato in complemento a due per permettere salti in avanti e all'indietro

L'offset varia tra  $-2^{15}$  e  $+2^{15}-1$

Esempio: **bne \$s0, \$s1, L1**

L'assemblatore sostituisce l'etichetta **L1** con l'offset **di parola** relativo a PC: **(L1- PC)/4**

- PC contiene già l'indirizzo dell'istruzione successiva al salto
- La divisione per 4 serve per calcolare l'offset di parola

Il valore del campo **indirizzo** può essere negativo (salti all'indietro)



# Istruzioni di tipo I: esempio

Nome campo	op	rs	rt	indirizzo
Dimensione	6-bit	5-bit	5-bit	16-bit
<b>beq \$s1, \$s2, 100</b>	000100	10001	10010	0000 0000 0001 1001

Diagram illustrating the bit fields for the instruction **beq \$s1, \$s2, 100**:

- The **op** field (6-bit) is 000100.
- The **rs** field (5-bit) is 10001.
- The **rt** field (5-bit) is 10010.
- The **indirizzo** field (16-bit) is 0000 0000 0001 1001.

A bracket below the 16-bit address field indicates a value of 25.



# Formato istruzioni di tipo J

È il formato usato per le istruzioni di salto incondizionato (*jump*), per esempio `j L1`



In questo caso, i campi hanno il seguente significato:

- **op (opcode)** indica il tipo di operazione **(2)**
- **indirizzo** (composto da **26-bit**) riporta una parte (26 bit su 32) dell'indirizzo **assoluto** di destinazione del salto

I 26-bit del campo **indirizzo** rappresentano un indirizzo di parola **(word address)**



# Istruzioni di salto incondizionato (tipo J)

L'assemblatore sostituisce l'etichetta L1 con i 28 bit meno significativi traslati a destra di 2 (divisione per 4 per calcolare l'indirizzo di parola) per ottenere 26-bit

- in pratica elimina i due 0 finali
- si amplia lo spazio di salto:
  - si salta tra 0 e  $2^{28}$  byte ( $2^{26}$  word)

I 26-bit di indirizzo nelle jump rappresentano un indirizzo di parola (word address)  $\Rightarrow$  corrispondono ad un indirizzo di byte (byte address) composto da 28 bit.

Poiché il registro PC è composto da 32 bit  $\Rightarrow$  l'istruzione jump rimpiazza solo i 28 bit meno significativi del PC, lasciando inalterati i rimanenti 4 bit più significativi.

PC[31:28]

L1/4 (26 bit)

00

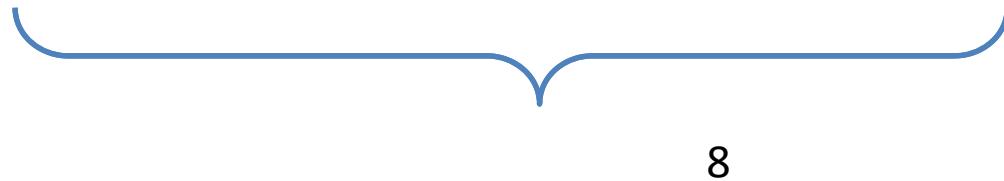


POLITECNICO MILANO 1863

40

# Istruzioni di tipo J: esempio

Nome campo	op	indirizzo				
Dimensione	6-bit	26-bit				
j 32	000010	00 0000	0000	0000 0000	0000	1000



A blue bracket is drawn under the 26-bit address field, which consists of the first four columns of binary digits. Below this bracket, the number '8' is centered, likely indicating the size or index of this field.



# Gestione costanti su 32 bit

Le istruzioni di **tipo I** consentono di rappresentare costanti esprimibili in 16 bit (valore massimo 65535 unsigned)

Se si hanno costanti da 32, l'assemblatore (o il compilatore) deve fare due passi per caricarla, suddividendo il valore della costante in due parti da 16 bit che vengono trattate separatamente come due *immediati* in due istruzioni successive:

- si utilizza l'istruzione **lui** (*load upper immediate*) per caricare il *primo immediato* (che rappresenta i 16 bit più significativi della costante) nei 16 bit più significativi di un registro; i rimanenti 16 bit meno significativi del registro sono posti a 0
- una successiva istruzione (ad esempio **ori** o anche **addi**) specifica tramite il *secondo immediato* i 16 bit meno significativi della costante



# Istruzioni di tipo I: istruzione lui

lui \$s0, 61						
Nome campo	op	rs	rt	indirizzo		
Dimensione	6-bit	5-bit	5-bit	16-bit		
<b>lui \$s0, 61</b>	001111	00000	10000	0000	0000	0011 1101



# Esempio

Per caricare in \$t0 il valore

0000 0000 1111 1111 0000 1001 0000 0000

lui \$t0, 255

ori \$t0, \$t0, 2034

255 (dec) = 0000 0000 1111 1111

2034 (dec) = 0000 1001 0000 0000

La versione in linguaggio macchina di lui \$t0, 255 # \$t0 è il registro 8:

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Il contenuto del registro \$t0 dopo avere eseguito lui \$t0, 255:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------



# Pseudoistruzione li

Consideriamo la costante a 32 bit:  $118345_{10}$  ( $0x1CE49$ )

0000 0000 0000 0001 1100 1110 0100 1001

16 bit più significativi  
corrispondenti al valore  $1_{10}$

16 bit meno significativi  
corrispondenti al valore  $52809_{10}$

La pseudoistruzione *load immediate* consente di caricarla nel registro

li \$t1, 118345                            # \$t1 ← 118345



# Pseudoistruzione *li*: esempio

L'assemblatore sostituisce la pseudoistruzione *li* con le seguenti istruzioni:

**lui \$at, 1** # 1 = 0000 0000 0000 0001

valore di **\$at**:

0000 0000 0000 0001 0000 0000 0000 0000

**ori \$t1, \$at, 52809** # \$t1 ← \$at or 52809

valore di **\$t1**:

0000 0000 0000 0001 1100 1110 0100 1001

**ori** esegue senza estensione di segno



# Pseudoistruzione la

Per caricare in un registro il valore dell'indirizzo di una variabile

**ADDR = *indirizzo simbolico*** (su 32 bit)

è disponibile la pseudoistruzione *load address* che lavora in modo simile alla *li* vista prima

**la \$t1, ADDR**       $\# \$t1 \leftarrow ADDR$

L'indirizzo su 32 può essere visto come la giustapposizione di due parti da 16 bit ciascuna

**ADDR<sub>High</sub> ADDR<sub>Low</sub>**

L'assemblatore e il collegatore espandono in modo opportuno (e più complicato ...) rispetto a *li*) la pseudo-istruzione sopra in un modo simile al seguente

```
lui reg, ADDR_HIGH      # i 16 bit più significativi dell'ind.
                             # sono caricati in reg e i rimanenti di reg posti a 0
```

```
ori reg, reg, ADDR_LOW    # i 16 bit meno significativi dell'ind.
                             # vengono giustapposti
```

**ori** esegue senza estensione di segno così come **addiu**



# ***Struttura del programma e direttive all'assemblatore***



POLITECNICO MILANO 1863

48

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# Come definiamo la struttura del programma

- Seguiamo uno schema di compilazione, ispirato a GCC, per tradurre da linguaggio sorgente C a linguaggio macchina MIPS
- Presuppone di conoscere MIPS: banco di registri, classi d'istruzioni, modi d'indirizzamento e organizzazione del sottoprogramma (chiamata rientro e passaggio parametri)
- Consiste in vari insiemi di convenzioni e regole per
  - segmentare il programma
  - dichiarare le variabili
  - usare (leggere e scrivere) le variabili
  - rendere le strutture di controllo
- Non attua necessariamente la traduzione più efficiente
- Sono possibili varie ottimizzazioni “ad hoc” del codice



# Come definiamo la struttura del programma

- dobbiamo definire un modello di architettura “run-time” per memoria e processore
- le convenzioni del modello run-time comprendono
  - collocazione e ingombro delle diverse classi di variabile
  - destinazione di uso dei registri
- il modello di architettura run-time consente interoperabilità tra porzioni di codice di provenienza differente, come per esempio codice utente e librerie standard precompilate
- esempio tipico in linguaggio C è la libreria standard di IO



# Struttura del programma e modello di memoria

- un programma in esecuzione (processo per il SO) ha tre segmenti essenziali
  - codice main e funzioni utente
  - dati variabili globali e dinamiche
  - pila aree di attivazione con indirizzi, parametri, registri salvati e variabili locali
- codice e dati sono segmenti dichiarati nel programma
- il segmento di pila viene creato al lancio del processo

Si possono avere anche

- due o più segmenti codice o dati
- segmenti di dati condivisi
- segmenti di libreria dinamica
- e altre peculiarità ...

questo modello di memoria è valido in generale



# Dichiarare i segmenti

gli indirizzi di impianto dei segmenti sono virtuali (non fisici)

```
// var. glob.          // segmento dati
...
// funzioni           .data
...
main (...) {         // indir. iniziale dati 0x 1000 0000
    // corpo          ...
    ...
}
                    // var globali e dinamiche
                    // segmento codice
                    .text
                    // indir. iniziale codice 0x 0040 0000
                    .globl main
main: ...           // codice programma
```

non occorre dichiarare esplicitamente il segmento di pila  
implicitamente esso inizia all'indirizzo 0x 7FFF FFFF  
e cresce verso gli indirizzi minori (ossia verso 0)



# Direttive dell'assemblatore

Sono direttive inserite nel *file sorgente* che vengono interpretate dall'assemblatore per generare il *file oggetto* e danno indicazioni su:

- variabili e strutture dati da allocare (eventualmente inizializzati) nel segmento dati del modulo (file) di programma (*data segment* – parte globale/statica) da assemblare
- istruzioni da allocare nel segmento codice (*text segment*) del modulo da assemblare
- indicazione di simboli globali definiti nel modulo, cioè referenziabili anche da altri moduli
- .....



# Convenzioni per le variabili

- in generale le variabili del programma sono collocate
  - globali in memoria a indirizzo fissato (assoluto)
  - locali, e parametri, nei registri del processore o nell'area di attivazione in pila (da precisare in seguito)
  - dinamiche in memoria (qui non sono considerate)
- le istruzioni aritmetico-logiche operano su registri
- dunque per operare sulla variabile (glob – loc – din)
  1. prima caricala in un registro libero (*load*)
  2. poi elaborala nel registro (confronto e aritmetica-logica)
  3. infine riscrivila in memoria (*store*)

Se variabile locale allocata in registro ⇒ salta (1) e (3)



## Come dichiarare le diverse classi di variabili

- in C la variabile è un oggetto formale e ha
  - nome per identificarla e farne uso
  - tipo per stabilirne gli usi ammissibili
- in MIPS la variabile è un elemento di memoria (byte parola o regione di mem) e ha una “collocazione” con
  - nome per identificarla
  - modo per indirizzarla
- in MIPS la variabile viene manipolata tramite indirizzo simbolico o nome di registro

occorre però distinguere tra diverse classi di variabile  
(var globale – parametro – var locale )



# Variabile globale nominale

- la variabile globale è collocata in memoria a indirizzo fisso stabilito dall'assemblatore e dal linker
- per comodità l'indirizzo simbolico della variabile globale coincide con il nome della variabile
- gli ordini di dichiarazione e disposizione in memoria delle variabili globali coincidono



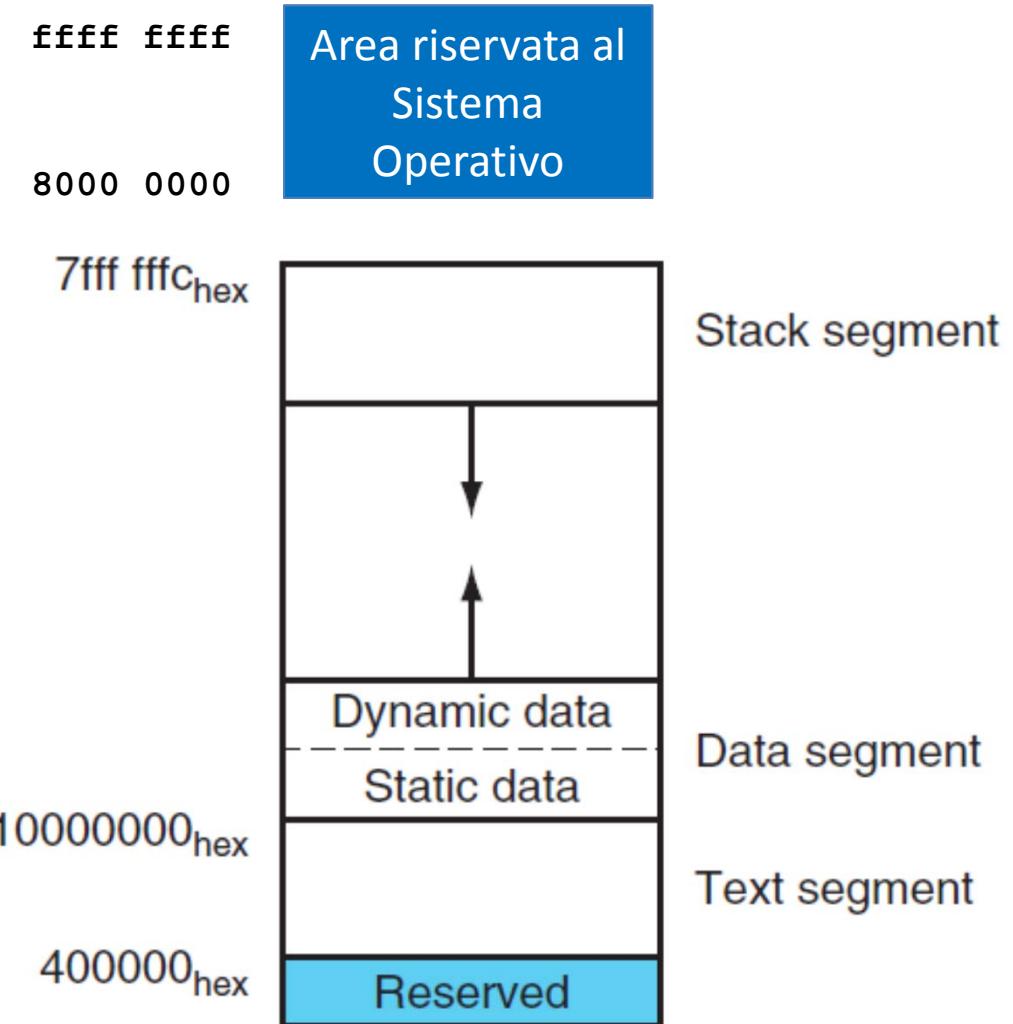
# Variabili globali e partizione della memoria

le variabili globali sono  
allocate a partire dall'indirizzo:  
**0x 1000 0000**

per puntare al segmento dati  
statici si può usare il registro  
*global pointer*: **\$gp**

il registro **\$gp** è inizializzato  
all'indirizzo:  
**0x 1000 8000**

con un *offset* di 16 bit rispetto  
al **\$gp** si indirizzano 64kbyte  
 $gp + 32\text{kbyte}$  e  $gp - 32\text{kbyte}$



# Esempio di dichiarazione di variabili globali

intero ordinario e corto a 32 e 16 bit rispettivamente

```
char c = `@`  
int a = 1  
short int B  
int vet [10]  
int * punt1  
char * punt2
```

	.data
C:	.byte 64 // @ valore iniziale
A:	.word 1 // 1 valore iniziale
B:	.half // non inizializzato
VET:	.space 40 // 10 elem × 4 byte)
PUNT1:	.word 0 // iniz. a NULL
PUNT2:	.word // non inizializzato

il valore iniziale è facoltativo

il puntatore è sempre una parola a 32 bit,  
indipendentemente dal tipo di oggetto puntato



# Direttive - 1

**.data <addr>**

Gli elementi successivi sono memorizzati nel segmento dati a partire dall'indirizzo **addr**

**.asciiz ''str''**

Memorizza la stringa **str** terminandola con il carattere **Null**

**.ascii ''str''**

ha lo stesso effetto, ma non aggiunge alla fine il carattere **Null**

**.byte b1,...,bn**

Memorizza gli **n** valori **b1**, .., **bn** in byte consecutivi di memoria

**.word w1, ...,wn**

Memorizza gli **n** valori su 32-bit **w1**, ..., **wn** in parole consecutive di memoria

**.half h1, ...,hn**

Memorizza gli **n** valori su 16-bit **h1**, ..., **hn** in halfword (mezze parole) consecutive di memoria

**.space n**

Allocata uno spazio pari ad **n** byte nel segmento dati



# Direttive - 2

## .text <addr>

Memorizza gli elementi successivi nel segmento testo dell'utente a partire dall'indirizzo

## .globl sym

Dichiara **sym** come etichetta globale (ad essa è possibile fare riferimento da altri file)

## .align n

Allinea il dato successivo a blocchi di **2<sup>n</sup>** byte: ad esempio

**.align 2 = .word** allinea alla parola (indirizzo multiplo di 4) il valore successivo

**.align 1 = .half** allinea alla mezza parola il valore successivo

**.align 0** elimina l'allineamento automatico delle direttive **.half**, **.word**, **.float**, e **.double** fino a quando compare la successiva direttiva **.data**

## .eqv

Sostituisce il secondo operando al primo. Il primo operando è un simbolo, mentre il secondo è un'espressione (direttiva #define del C)

Tutte le direttive che memorizzano valori o allocano spazio di memoria sono precedute da un'etichetta simbolica che rappresenta l'indirizzo iniziale



# Direttive: esempio

```
# Somma valori in un array
.data
array: .word    1,2,3,4,5,6,7,8,9,10  # dichiarazione array
.text
.globl  main

main:
        li $s0,10          # $s0 ← numero elementi
        la $s1,array        # $s1 ← registro base per array
        li $s2,0            # azzero $s2 contatore cicli
        li $t2,0            # azzero $t2 accumulatore
loop:   lw $t1,0($s1)      # accesso all'array
        add $t2,$t1,$t2    # calcolo risultato nell'acc. $t2
        addi $s1,$s1,4       # $s1 ← ind. del prossimo
                           # elemento dell'array
        addi $s2,$s2,1       # incremento $s2 contatore cicli
        bne $s2,$s0,loop     # test terminazione
```



# Parametri in ingresso a una funzione e valore restituito

i primi quattro parametri vanno passati nei registri *a0* (primo nella testata), *a1*, *a2* e *a3* (quarto)

- se sono di tipo scalare o puntatore (a 32 bit)
- il nome di vettore è considerato puntatore (al primo elem.)
- per passare una *struct* si veda la ABI del compilatore\*

gli eventuali parametri rimanenti vanno impilati a cura del chiamante, sempre come valori a 32 bit

- è raro che una funzione abbia più di quattro parametri

il valore in uscita (a 32 bit) va restituito nel registro *v0*

- se è di tipo scalare o puntatore
- il nome di vettore è considerato puntatore (al primo elem.)
- per restituire una *struct* si veda la ABI del compilatore\*

se il valore in uscita è di tipo double, si usa anche *v1*

\* la ABI di *gcc* impone di passare l'indirizzo dell'inizio della *struct*



## Variabile locale nominale

la variabile locale può essere gestita in vari modi, secondo il tipo di variabile e il grado di ottimizzazione del codice, e anche in dipendenza di come la variabile viene utilizzata

variabile di tipo scalare o puntatore – due modi

- in un registro del blocco s0 - s7, se per altro motivo non deve avere un indirizzo di memoria – vedi precisazioni sotto
- altrimenti nell'area di attivazione della funzione

variabile di tipo scalare, ma che viene anche acceduta tramite un puntatore – un solo modo

- nell'area di attivazione della funzione, perché deve avere un indirizzo

variabile di tipo *array* (o *struct*) – un solo modo

- sempre nell'area di attivazione della funzione



## Convenzioni

Come usare le diverse classi di variabili scalari

Come rendere le strutture di controllo

Come usare una variabile strutturata

Vedi ..... Come tradurre da C a MIPS

Attenzione alle **convenzioni ACSO**: per esempio

➤ Array (vettori)

- indirizzo base (= nome array) caricato in un registro
- indirizzo effettivo di un generico elemento calcolato tramite il registro

➤ Variabili locali

- vedi prima



POLITECNICO MILANO 1863

64

# Vettore – scansione sequenziale con indice

```
// variabili globali
int v [5] // vettore
int a      // indice
...
// testata del ciclo
for (a = 2; a <= 5; a++) {
    v[a] = v[a] + 6
} /* end for */
// seguito del ciclo
...
```

```
v:   .space 20           // 20 byte per v
A:   .word              // mem per a
      // assegna a = 2 già visto
FOR: li   $t0, 5          // inizializza $t0
      lw   $t1, A            // carica a
      bgt $t1, $t0, END     // se .. va' a END
      li   $t0, 6          // inizializza $t0
      la   $t1, V            // ind. iniz. di v
      lw   $t2, A            // carica a
      sll $t2, $t2, 2         // allinea indice
      addu $t1, $t1, $t2     // indir. di v[a]
      lw   $t3, 0($t1)       // carica v[a]
      add $t3, $t3, $t0      // v[a] + 6
      sw   $t3, 0($t1)       // memorizza v[a]
      // aggiorna a++ già visto
      j    FOR               // torna a FOR
END: ...                  // seguito ciclo
ottimizzazioni possibili trattando costanti e aritmetica
```



# ***Approfondimento – System Call***



POLITECNICO MILANO 1863

66

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# System Call

Sono disponibili delle **chiamate di sistema (system call)** predefinite che implementano particolari servizi (per esempio: stampa a video)

Ogni system call ha:

- un codice
- degli argomenti (opzionali)
- dei valori di ritorno (opzionali)



## System Call: qualche esempio

**print\_int**: stampa sulla console il numero intero che le viene passato come argomento;

**print\_string**: stampa sulla console la stringa che le è stata passata come argomento terminandola con il carattere **Null**;

**read\_int**: legge una linea in ingresso fino al carattere a capo incluso (i caratteri che seguono il numero sono ignorati);

**read\_string**: legge una stringa di caratteri di lunghezza **\$a1** da una linea in ingresso scrivendoli in un buffer (**\$a0**) e terminando la stringa con il carattere **Null** (se ci sono meno caratteri sulla linea corrente, li legge fino al carattere a capo incluso e termina la stringa con il carattere **Null**);

**sbrk** restituisce il puntatore (indirizzo) ad un blocco di memoria;

**exit** interrompe l'esecuzione di un programma;

E anche altre ...



# System Call

Nome	Codice	Argomenti	Risultato
print_int	1	\$a0	
print_float	2	\$f12	
print_double	3	\$f12	
print_string	4	\$a0	
read_int	5		\$v0
read_float	6		\$f0
read_double	7		\$f0
read_string	8	\$a0,\$a1	
sbrk	9	\$a0	\$v0
exit	10		



# System Call

Per richiedere un servizio a una chiamata di sistema (**syscall**) occorre:

- caricare il **codice** della **syscall** nel registro **\$v0**
- caricare gli **argomenti** nei registri **\$a0** - **\$a3** (oppure nei registri **\$f12** - **\$f15** nel caso di valori in virgola mobile)
- eseguire **syscall**
- l'eventuale **valore di ritorno** è caricato nel registro **\$v0** (**\$f0**)



# Esempio

```
#Programma che stampa: la risposta è 5
    .data
str:   .asciiz "la risposta è"
    .text

    li $v0, 4          # $v0 ← codice della print_string
    la $a0, str        # $a0 ← indirizzo della stringa
    syscall            # stampa della stringa

    li $v0, 1          # $v0 ← codice della print_integer
    li $a0, 5          # $a0 ← intero da stampare
    syscall            # stampa dell'intero

    li $v0, 10         # $v0 ← codice della exit
    syscall            # esce dal programma
```



# Esempio

```
#Programma che stampa "Dammi un intero: "
# e che legge un intero
.data
prompt:.asciiz "Dammi un intero: "
.text
.globl main
main:
    li $v0, 4          # $v0 ← codice della print_string
    la $a0, prompt    # $a0 ← indirizzo della stringa
    syscall           # stampa la stringa

    li $v0, 5          # $v0 ← codice della read_int
    syscall           # legge un intero e lo carica in $v0

    li $v0, 10         # $v0 ← codice della exit
    syscall           # esce dal programma
```





**POLITECNICO**  
MILANO 1863

# **Architettura dei calcolatori e sistemi operativi**

## **Sottoprogramma e MIPS**

### **Capitolo 2 P&H**

# Sommario

Modello di chiamata

Area di attivazione



POLITECNICO MILANO 1863

2

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# Sottoprogramma

Nei linguaggi di alto livello, per esempio in linguaggio C, la **chiamata a sottoprogramma** ha come effetto la creazione di un'**area (o record) di attivazione** sulla **pila (stack)**.

A ogni chiamata di sottoprogramma viene creata un'area di attivazione.

Quando il sottoprogramma termina l'area viene rilasciata dalla pila.

Con sottoprogrammi annidati le aree vengono tutte messe sulla pila e l'ultima messa (ossia quella in cima all'apila) corrisponde al sottoprogramma correntemente in esecuzione.

L'area di attivazione di *main* è la prima ad essere creata quando il programma viene lanciato e l'ultima a essere rilasciata (deallocata).



# Sottoprogramma (continua)

L'area di attivazione del sottoprogramma è associata in modo opportuno alle informazioni seguenti:

- i parametri formali (e i loro valori) passati al sottoprogramma
- l'indirizzo di ritorno al (sotto)programma chiamante
- informazioni per gestire lo spazio allocato per l'area di attivazione
- le variabili locali del sottoprogramma
- il valore restituito al (sotto)programma chiamante



# Sottoprogramma e ISA

A livello ISA la chiamata a sottoprogramma di alto livello deve essere espansa in più istruzioni macchina, eseguite in ambiti diversi:

- il **chiamante (caller)** gestisce la parte relativa al passaggio dei parametri
- il **chiamante** attiva il sottoprogramma tramite l'**istruzione di chiamata ISA**
- l'**esecuzione dell'istruzione di chiamata ISA** gestisce la parte relativa al salvataggio dell'indirizzo di ritorno (**PC**) e attiva il sottoprogramma
- il **chiamato (callee)** gestisce l'allocazione delle variabili locali e del valore restituito



# Modello di chiamata a sottoprogramma

La chiamata a sottoprogramma segue sempre l'espansione vista, però il ***modello di chiamata a sottoprogramma*** non è identico in tutti i processori.

A livello ISA:

- ❑ per vari aspetti, è necessario tenere conto anche dei registri del processore
- ❑ in generale è sempre possibile una certa flessibilità (il modello di chiamata non è totalmente vincolato)

Per esempio:

- Valori dei parametri attuali, valore restituito: registri o stack ?
- Indirizzo di ritorno: registro o stack ?
- Salvataggio dei registri usati nel modello di chiamata e nel sottoprogramma: chi li salva ?
- Come si definisce l'area di attivazione e come si gestiscono le variabili locali ?



# Modello di chiamata a sottoprogramma in MIPS

L'architettura ISA di MIPS:

- vincola in modo forte il ***salvataggio dell'indirizzo di ritorno*** (è fatto hardware) tramite l'istruzione di chiamata (e quella di ritorno) da sottoprogramma
- definisce delle convenzioni sempre adottate per il ***passaggio dei parametri*** e per il ***valore restituito***
- caratterizza i ***gruppi di registri*** in base al fatto che i valori corrispondenti siano o meno da considerare preservati dal chiamato
- se un registro è definito “preservato dal chiamato”, allora sarà compito del chiamato salvarne i valori per poi restituirlo integro al chiamante (***callee-saved registers***)



# Istruzione di chiamata e ritorno da sottoprogramma in MIPS

## ***Chiamata a sottoprogramma***

**jal label**      (jump and link)

# \$ra  $\leftarrow$  PC + 4

# PC  $\leftarrow$  i.e. *label*

L'etichetta *label* è il riferimento simbolico all'indirizzo della prima istruzione del sottoprogramma. Verrà tradotta in indirizzo effettivo (i.e.) dall'assemblatore e dal collegatore (linker).

## ***Ritorno da sottoprogramma***

**jr \$ra**      (jump register)

# PC  $\leftarrow$  \$ra



POLITECNICO MILANO 1863

8

# Convenzioni per il passaggio dei parametri e per il valore restituito

## Passaggio dei parametri

- i primi quattro parametri (**argument**), numerati da sx a dx nella testata, vengono passati nei registri **\$a0-\$a3**
  - se di tipo scalare o puntatore (a 32 bit)
  - il nome di vettore è considerato puntatore (al primo elemento)
- i parametri rimanenti, se presenti, sono passati sulla pila
  - se un sottoprogramma ha 6 parametri i valori di *arg5* e *arg6* sono passati sulla pila

## Valore restituito

- il valore restituito viene salvato nel registro **\$v0**
  - se di tipo scalare o puntatore (a 32 bit)
  - il nome di vettore è considerato puntatore (al primo elemento)
- se di tipo *double* (numero reale in vigola mobile) si usa anche **\$v1**



# Convenzioni per il salvataggio dei registri

- L'esecuzione di un sottoprogramma non deve interferire con l'ambiente del (sotto)programma chiamante.
- I registri usati dal chiamato devono essere ripristinabili al rientro.

## Convenzioni adottate da MIPS

- Per ottimizzare gli accessi alla memoria, il chiamante e il chiamato salvano (eventualmente) sulla pila soltanto un particolare gruppo di registri.
- Il chiamato può usare la pila per le strutture dati locali (p. es. array, strutture) e le variabili locali.

<i>Preservato dal callee (registri callee-saved)</i>	<i>Non preservato dal callee (registri caller-saved)</i>
registri saved: \$s0-\$7	registri temporanei: \$t0-\$t9
registro frame pointer: \$fp	registri argomento: \$a0-\$a3
registro return address: \$ra	registri di ritorno: \$v0-\$v1



# La gestione della pila in MIPS

La pila cresce da indirizzi di memoria alti verso indirizzi di memoria bassi e il registro *stack pointer* **\$sp** punta alla prima parola piena della pila.

L'inserimento di un dato nella pila (operazione di **push**) avviene decrementando il registro **\$sp** per allocare lo spazio, ed eseguendo l'istruzione **sw** per inserire il dato.

Esempio: salvare il registro **\$s0** sulla pila

```
addiu $sp, $sp, -4  
sw    $s0, 0($sp)
```

Il prelevamento di un dato dalla pila (operazione di **pop**) avviene eseguendo l'istruzione **lw** e incrementando il registro **\$sp** (per eliminare il dato), riducendo così la dimensione della pila.

Esempio: prelevare la cima della pila e salvarla nel registro **\$s0**

```
lw    $s0, 0($sp)  
addiu $sp, $sp, 4
```

Nota: **0 (\$sp)** si può abbreviare in **(\$sp)**, intendendo che lo spiazzamento sia 0.



# Passi del modello di chiamata in MIPS

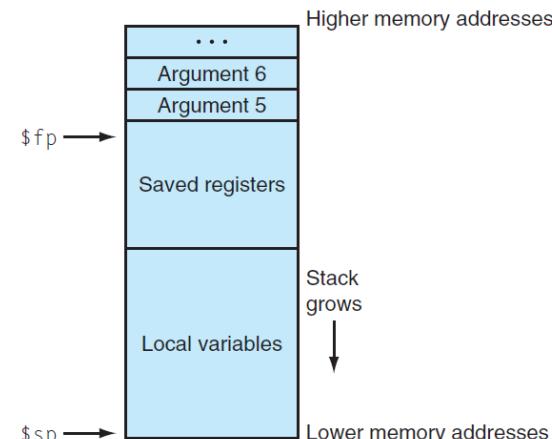
- La chiamata a funzione comporta queste fasi in successione:
  - prologo del chiamante
  - salto a chiamato
  - prologo del chiamato
  - corpo elaborativo del chiamato
  - epilogo del chiamato
  - rientro a chiamante
  - epilogo del chiamante
- Alcuni passaggi possono ridursi a poco, secondo le convenzioni o la situazione specifica.
- Relativamente alla pila:
  - il prologo del chiamante può comportare passaggio di parametri e salvataggio di registri
  - il prologo del chiamato può comportare il salvataggio di registri e l'allocazione di variabili locali
  - è necessario calcolare la **dimensione in byte** dell'area richiesta: quest'area è ***l'area (o record o frame) di attivazione*** della funzione



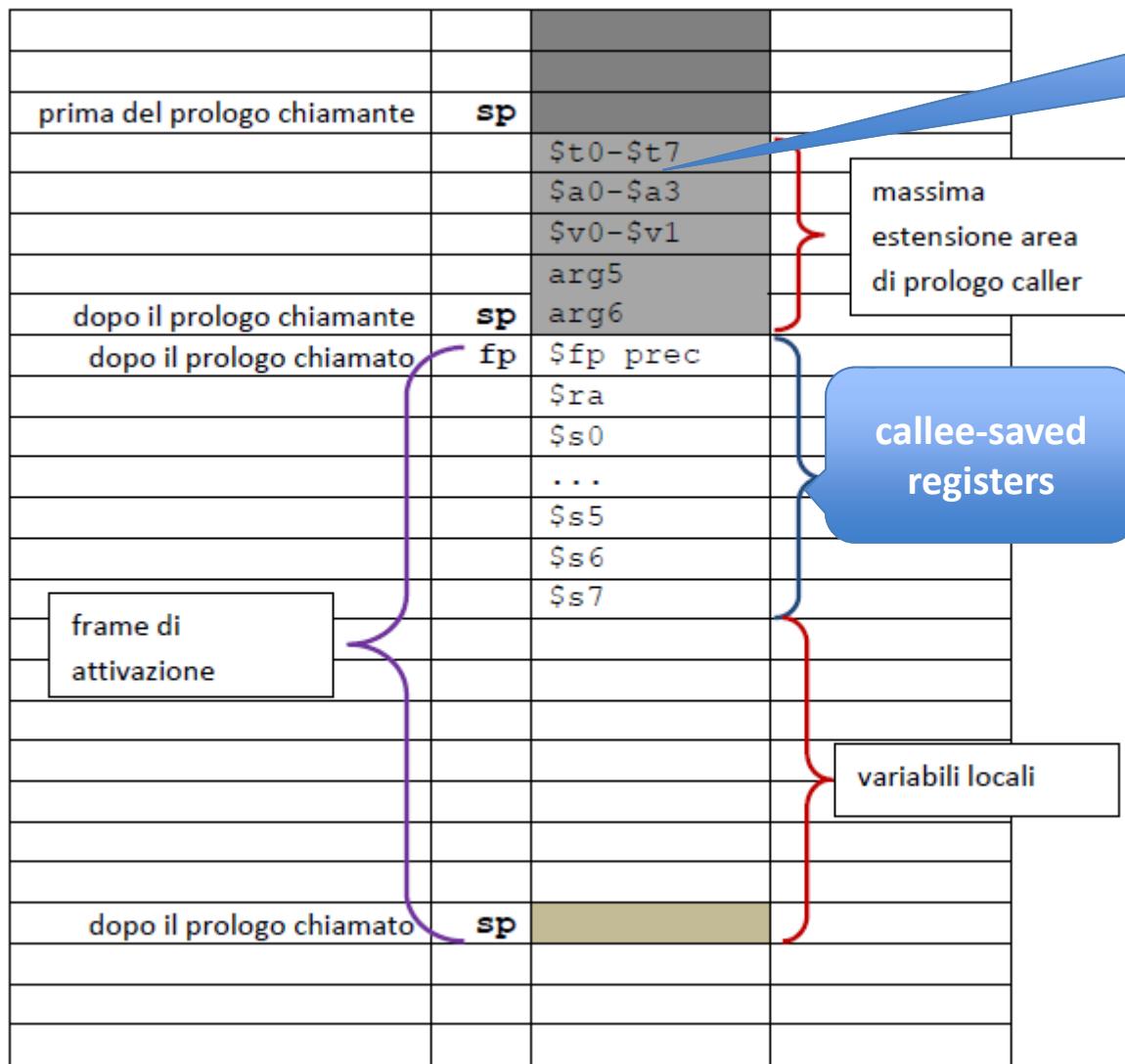
# Convenzioni (ulteriori) adottate in ACSO

- Il salvataggio dei registri nella pila viene fatto seguendo il numero d'ordine crescente all'interno di ciascun gruppo:  $t0, t1 \dots t9$  ecc.
  - per l'ordine di salvataggio dei «gruppi» si veda il dettaglio dell'area di attivazione
- Il salvataggio di un qualsiasi registro viene fatto solo se necessario. Per esempio:
  - ***fp*** è salvato solo se lo si usa per referenziare parole di memoria in pila
  - ***ra*** non viene salvato in procedure foglia (che non ne chiamano altre)
- Gli argomenti di un sottoprogramma vengono passati in ordine di elencazione:
  - primi quattro nei registri  $a0 - a3$
  - poi in pila (p. es. *arg5* e *arg6*)

**Nota bene:** alcune convenzioni ACSO possono differire da quelle del libro di testo.



# Area di attivazione



caller-saved  
registers

È mostrato:

- il caso più generale di salvataggio dei registri conforme alle convenzioni stabilite
- l'ordine di salvataggio dei «gruppi» di registri



# Convenzioni di chiamata *caller* (1)

## *prologo del chiamante*

- scrive nei registri *a0* - *a3* i paramenti in ingresso alla funzione
  - trascuriamo il caso di più di quattro parametri
- salva sulla pila i registri *t0* – *t9* che si vogliono riavere inalterati quando la funzione sarà rientrata (idem per i registri *v0* – *v1*)
- se occorre preservarli, salva sulla pila gli argomenti *a0* - *a3*
  - per ottimizzare un po', possiamo limitarci a salvare in pila, tra i parametri passati al chiamato, solo quelli che il chiamante non usa più a valle della chiamata, e che dunque esso non ha bisogno di riavere inalterati qualora il chiamato li avesse modificati (ma non è un errore salvarli sempre tutti)

## *salto a chiamato*

- **jal FUNZ**



# Convenzioni di chiamata *callee* (1)

## *prologo del chiamato*

- crea area di attivazione:  
**addiu \$sp, \$sp, -dim. area in byte**
- se il registro *\$fp* è in uso:
  - viene salvato sulla pila (salva *\$fp* precedente)
  - viene aggiornato in modo da puntare alla cima dell'area di attivazione, cioè alla parola di pila che contiene il valore di *\$fp* appena salvato
- se la funzione chiamata non è foglia:
  - il registro *\$ra* viene salvato sulla pila poiché sarà usato (e dunque sovrascritto) in chiamata annidata
- salva sulla pila i registri *s0 – s7* assegnati a variabili locali

## *corpo elaborativo del chiamato*



POLITECNICO MILANO 1863

16

# Convenzioni di chiamata *calle* (2)

## *epilogo del chiamato*

- scrive nel registro *v0* il valore di uscita
- ripristina dalla pila i registri *s0 – s7* assegnati a variabili locali
- se il registro *\$fp* è in uso, lo ripristina dalla pila
- se la funzione non è foglia, ripristina dalla pila il registro *\$ra*
- elimina area di attivazione:

**addiu \$sp, \$sp, dim. area in byte**

## *rientro a chiamante*

- **jr \$ra**



# Convenzioni di chiamata *caller* (2)

## *epilogo del chiamante*

- ripristina dalla pila i registri  $a0 – a3$  che erano stati preservati per il rientro dalla funzione chiamata
- ripristina dalla pila i registri  $t0 – t9$  che erano stati preservati per il rientro dalla funzione chiamata (idem per i reg.  $v0 – v1$ )
- trova nel registro  $v0$  il valore di uscita dalla funzione chiamata



## Esempio 1

varloc a allocata in registro s0 (senza fp)

```
int f (int n) {  
    int a  
    ...  
    a = a + n  
    // chiama funz.  
    ...  
    return a  
} /* f */
```

area di attivazione (dim. 8 byte)  
ra salvato 4(\$sp)  
sp-> s0 salvato 0(\$sp)

F:  
**addiu** \$sp, \$sp, -8 // crea area  
**sw** \$ra, 4(\$sp) // salva ra  
**sw** \$s0, 0(\$sp) // salva s0  
...  
**add** \$s0, \$s0, \$a0 // calcola a = a + n  
// chiama un'altra funzione  
...  
**move** \$v0, \$s0 // valore uscita  
**lw** \$s0, 0(\$sp) // ripristina s0  
**lw** \$ra, 4(\$sp) // ripristina ra  
**addiu** \$sp, \$sp, 8 // elimina area  
**jr** \$ra // rientra



## Esempio 2

varloc a allocata in pila (senza fp)

```
int f (int n) {  
    int a  
    ...  
    a = a + n  
    // chiama funz.  
    ...  
    return a  
} /* f */
```

area di attivazione (dim. 8 byte)

```
    ra salvato 4($sp)  
    sp-> varloc a 0($sp)
```

F: **addiu** \$sp, \$sp, -8 // crea area  
**.eqv** RA, 4 // spiazzamento in pila di ra  
**.eqv** A, 0 // spiazzamento in pila di a  
**sw** \$ra, RA(\$sp) // salva ra  
...  
**lw** \$t0, A(\$sp) // carica a  
**add** \$t0, \$t0, \$a0 // calcola a = a + n  
**sw** \$t0, A(\$sp) // memorizza a  
// chiama un'altra funzione  
...  
**lw** \$v0, A(\$sp) // valore uscita  
**lw** \$ra, RA(\$sp) // ripristina ra  
**addiu** \$sp, \$sp, 8 // elimina area  
**jr** \$ra // rientra

similmente se ci sono più variabili locali



POLITECNICO MILANO 1863

20

## Esempio 3

```
int f (int n) {
    int a
    ...
    a = a + n
    // chiama funz.
    ...
    return a
} /* f */
```

area di attivaz. (dim. 12 byte)  
con spiazzamenti riferiti a \$sp  
fp-> fp salvato 8(\$sp)  
ra salvato 4(\$sp)  
sp-> varloc a 0(\$sp)

area di attivaz. (dim. 12 byte)  
con spiazzamenti riferiti a \$fp  
fp-> fp salvato 0(\$fp)  
ra salvato -4(\$fp)  
sp-> varloc a -8(\$fp)

F: **addiu** \$sp, \$sp, -12 // crea area  
**sw** \$fp, 8(\$sp) // salva fp precedente  
**addiu** \$fp, \$sp, 8 // sposta fp corrente  
**sw** \$ra, -4(\$fp) // salva ra  
...  
**lw** \$t0, -8(\$fp) // carica a  
**add** \$t0, \$t0, \$a0 // calcola a = a + n  
**sw** \$t0, -8(\$fp) // memorizza a  
// chiama un'altra funzione  
...  
**lw** \$ra, -4(\$fp) // ripristina ra  
**lw** \$fp, 8(\$sp) // ripristina fp  
**addiu** \$sp, \$sp, 12 // elimina area  
 **ora \$sp potrebbe anche cambiare durante l'esecuzione  
**attenzione: spiazzamenti di \$ra e varloc a riferiti a \$fp**  
similmente con le combinazioni viste prima e/o con .eqv**





**POLITECNICO**  
MILANO 1863

# Architettura dei calcolatori e sistemi operativi

## Assemblatore e Collegatore (Linker)

Capitolo 2 P&H  
Appendice 2 P&H

# Sommario

Il processo di assemblaggio

Il collegatore (linker)



POLITECNICO MILANO 1863

2

# Assemblatore: traduzione in linguaggio macchina

L'**ASSEMBLATORE** traduce in linguaggio macchina un programma scritto in linguaggio assemblatore simbolico, e genera la **rappresentazione oggetto** del programma (che non è ancora il formato eseguibile).

L'assemblatore traduce le pseudo-istruzioni (e anche le macro) nella corrispondente sequenza di istruzioni native.

Come si è visto, le pseudo-istruzioni forniscono un insieme di istruzioni macchina simboliche più ricco di quello realizzato nativamente in HW, con il solo “costo” aggiuntivo dato dal registro **\$at** riservato all’uso da parte dell’assemblatore per espandere la pseudo-istruzione (o macro).



# Processo di assemblaggio

- È applicato modulo per modulo al programma, e per ogni **modulo (file) sorgente** costruisce il **modulo (file) oggetto** corrispondente.
- Esamina, riga per riga, il codice sorgente assemblatore di un modulo, e traduce le istruzioni simboliche nel corrispondente **formato del linguaggio macchina**.
  - i codici mnemonici sono tradotti nei corrispondenti codici binari
  - i riferimenti ai registri nei corrispondenti “numeri” di registro
  - i **riferimenti simbolici** del modulo (identificatori di variabili, etichette di salto, nomi di funzioni) sono tradotti – se possibile – negli indirizzi binari corrispondenti; per questa operazione l’assemblatore genera la **tabella dei simboli** del modulo
- Ogni segmento di ogni modulo è assemblato con indirizzi virtuali rilocabili di segmento (gli indirizzi di ogni segmento partono da 0).



# Processo di assemblaggio (continua)

Un programma eseguibile è generato a partire da più moduli (p. es. le librerie standard come la `stdio` del linguaggio C).

Non è sempre possibile tradurre tutti i riferimenti simbolici del modulo.

I riferimenti relativi a:

- identificatori definiti in altri moduli (**simboli esterni**)
- identificatori dipendenti dalla rilocazione del modulo (**simboli rilocabili**)

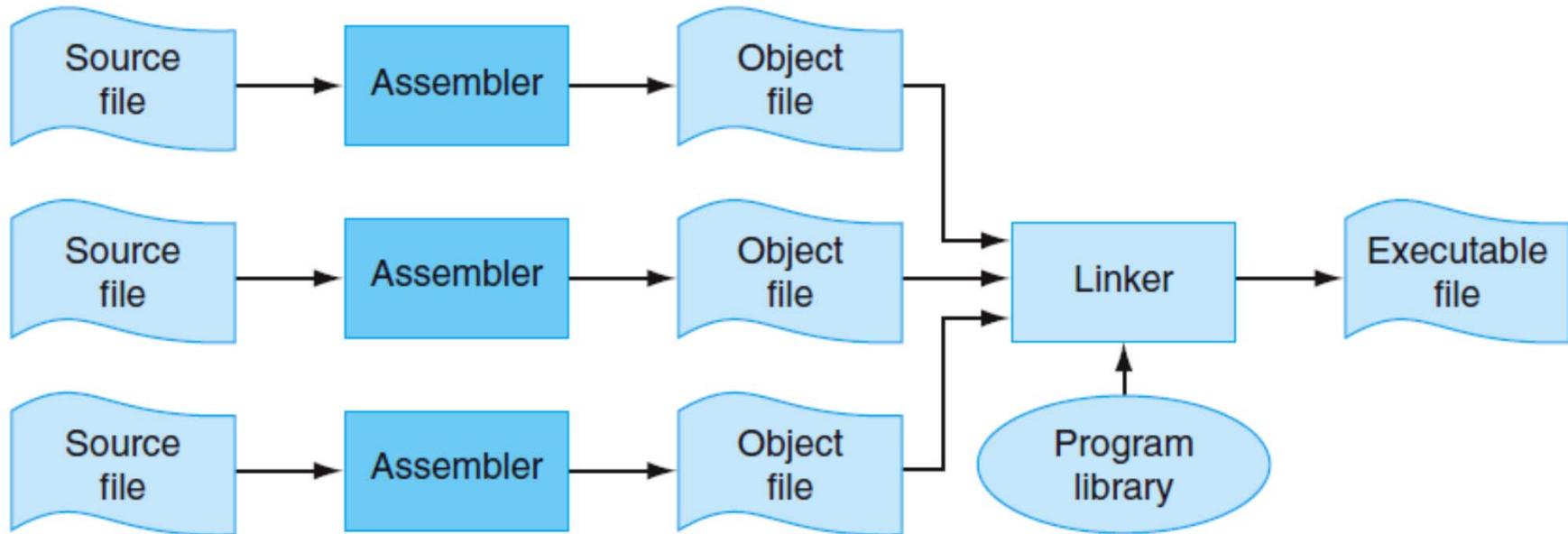
costituiscono i **riferimenti non risolti** del modulo e vengono gestiti dal **COLLEGATORE (LINKER)**

Quindi il file oggetto prodotto dall'assemblatore deve contenere anche:

- la tabella dei simboli
- informazioni sulla rilocazione



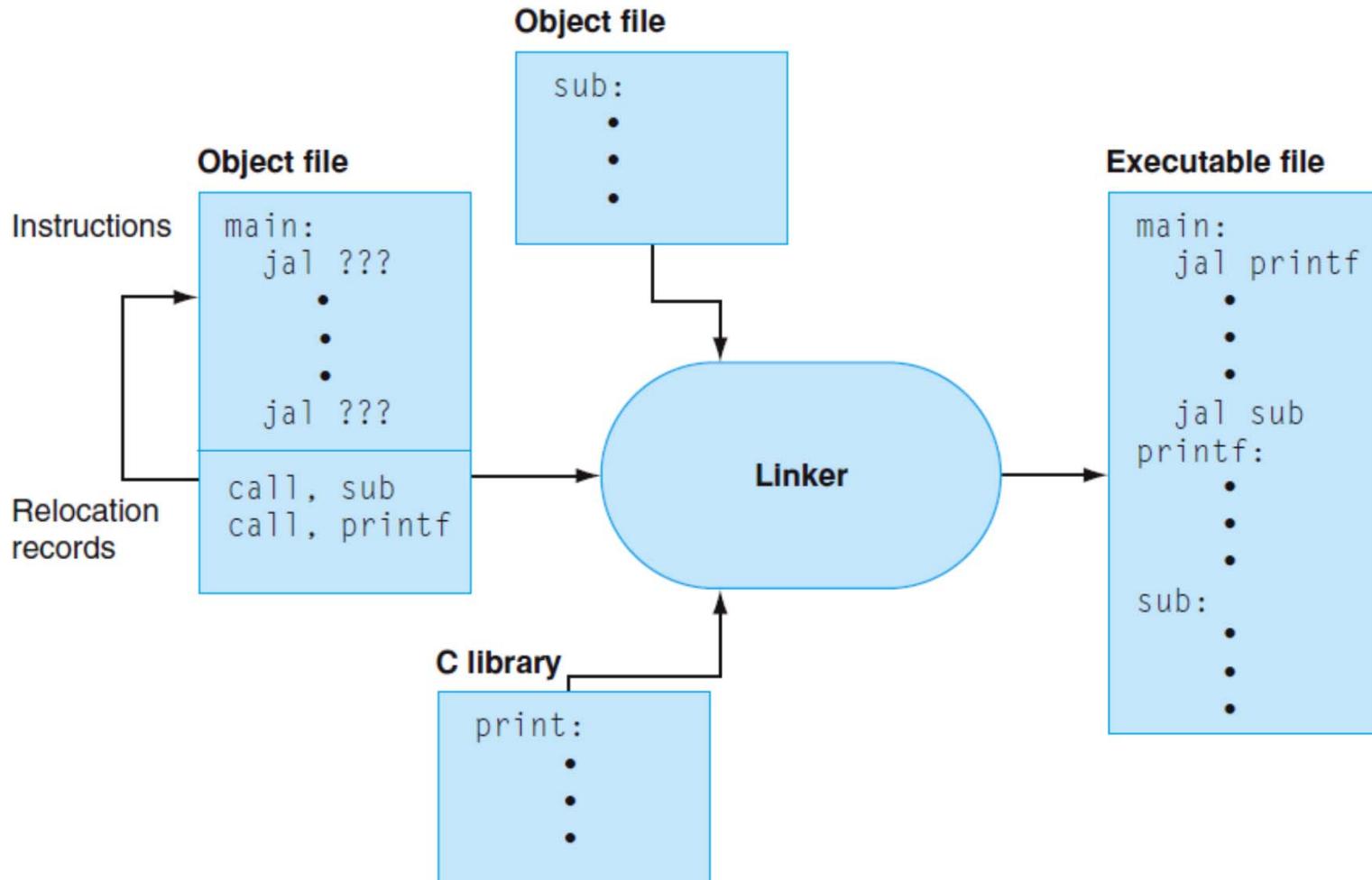
# Processo di assemblaggio (continua)



**Figura A.1.1. Il processo che produce un file eseguibile.** Un assemblatore traduce il file contenente codice assembler in un file oggetto, il quale viene collegato ad altri file e a funzioni di libreria per produrre un file eseguibile.



# Processo di assemblaggio (continua)



## Processo di assemblaggio – primo passo

Nel primo passo l'assemblatore non traduce nessuna istruzione, ma costruisce la **tabella dei simboli del modulo**.

I riferimenti simbolici inseriti nella tabella possono essere:

- simboli associati a direttive dell'assemblatore che definiscono costanti simboliche (.eqv) – nella tabella si crea la coppia  $\langle \text{simbolo}, \text{valore} \rangle$
- etichette che definiscono variabili del segmento dati – nella tabella si crea la coppia  $\langle \text{simbolo}, \text{indirizzo} \rangle$
- etichette che contrassegnano istruzioni destinazioni di salto – nella tabella si crea la coppia  $\langle \text{simbolo}, \text{indirizzo} \rangle$

I **valori degli indirizzi** inseriti sono quelli **rilocabili** rispetto al segmento considerato (T o D).



## Processo di assemblaggio – secondo passo

È la fase di traduzione vera e propria: usa la tabella dei simboli del modulo e genera – oltre alla traduzione – la **tabella di rilocazione** del modulo.

Un'istruzione **è tradotta in modo «incompleto»** (e quindi deve essere processata anche dal collegatore) se:

- il riferimento simbolico presente in essa è relativo a variabili del segmento **.data**
  - per variabili scalari la traduzione convenzionale è **0 (\$gp)**
  - per vettori (e altre) la traduzione è tramite la pseudoistruzione **1a**, ossia tramite **lui** e **ori (addiu)** con immediati convenzionali a **0**
- il riferimento è relativo a simboli non presenti nella tabella dei simboli del modulo: simbolo posto a **0** per convenzione
- è un'istruzione di salto di tipo J (non autorilocante): simbolo posto a **0** per convenzione

In corrispondenza di ogni traduzione “incompleta”, nella tabella di rilocazione viene inserito un elemento (una tripletta) nella forma seguente:

⟨ **indirizzo rilocabile istruzione, codice op. istruzione, simbolo da risolvere** ⟩



# ESEMPIO

## Sorgente procedura B:

```
.text
B:  bne $a0, $0, E
     sw  $a0, Y
E:  lui $t0, W
     ori $t0, $t0, W
.data
Y:  .word 0
```

## Oggetto procedura B:

(passo 1 assemblatore)

dimensione testo: ----

dimensione dati: ----

segmento di testo:

0	bne	\$a0, \$0, E
4	sw	\$a0, Y
8	lui	\$t0, W
C	ori	\$t0, \$t0, W

segmento dei dati:

0	0
---	---

tabella dei simboli:

B	0	T
E	8	T
Y	0	D

## Oggetto procedura B:

(passo 2 assemblatore)

dimensione testo: 0x10

dimensione dati: 0x04

segmento di testo:

0	bne	\$a0, \$0, 1
4	sw	\$a0, 0(\$gp)
8	lui	\$t0, 0
C	ori	\$t0,\$t0, 0

segmento dei dati:

0	0
---	---

tabella dei simboli:

B	0	T
E	8	T
Y	0	D

tabella di rilocazione:

4	sw	Y
8	lui	W
C	ori	W

Se l'istruzione è di salto in formato I,  
il simbolo viene risolto come

$$(\text{VS\_REL} - (\text{IADDR\_REL} + 4)) / 4$$

Per esempio in «**bne** \$a0, \$0, E» il simbolo E viene tradotto come  $(8 - (0 + 4)) / 4 = 1$



# Il file oggetto di un modulo

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

L'**intestazione** descrive le *dimensioni* del segmento testo e del segmento dati del modulo.

Il **segmento testo** contiene il codice in linguaggio macchina delle procedure (o funzioni) del file sorgente. Queste procedure potrebbero essere non eseguibili a causa di riferimenti non risolti.

Il **segmento dati statici** contiene una rappresentazione binaria dei dati definiti nel file sorgente. Anche i dati potrebbero essere incompleti a causa di riferimenti non risolti a etichette definite in altri file.

Le **informazioni di rilocazione** identificano le istruzioni e le parole di dati che dipendono da **indirizzi assoluti all'interno del file eseguibile** del programma completo. La posizione di queste istruzioni e dati deve essere modificata se parti del programma vengono spostate in memoria.

La **tabella dei simboli** associa un indirizzo alle etichette «esterne» contenute nel modulo sorgente e contiene l'elenco dei riferimenti non risolti del modulo.

Le **informazioni di debug**, che non consideriamo ulteriormente.



# Esempio di riferimento

## sorgente MAIN:

```
.text  
MAIN: lw $a0, X  
      beq $a0, $0, E  
      jal B  
.globl MAIN  
.data  
X: .word 128  
W: .word 0x12345678
```

## sorgente procedura B:

```
.text  
B: bne $a0, $0, E  
  sw $a0, Y  
E: lui $t0, W  
  ori $t0, $t0, W  
.data  
Y: .word 0
```

## oggetto MAIN:

dimensione testo: 0x0C  
dimensione dati: 0x08  
segmento testo:  
0 lw \$a0, 0(\$gp)  
4 beq \$a0, \$0, 0  
8 jal 0  
segmento dati:  
0 0x80  
4 0x12345678

## tabella dei simboli:

MAIN	0	T
X	0	D
W	4	D

## tabella di rilocazione:

0	lw	X
4	beq	E
8	jal	B

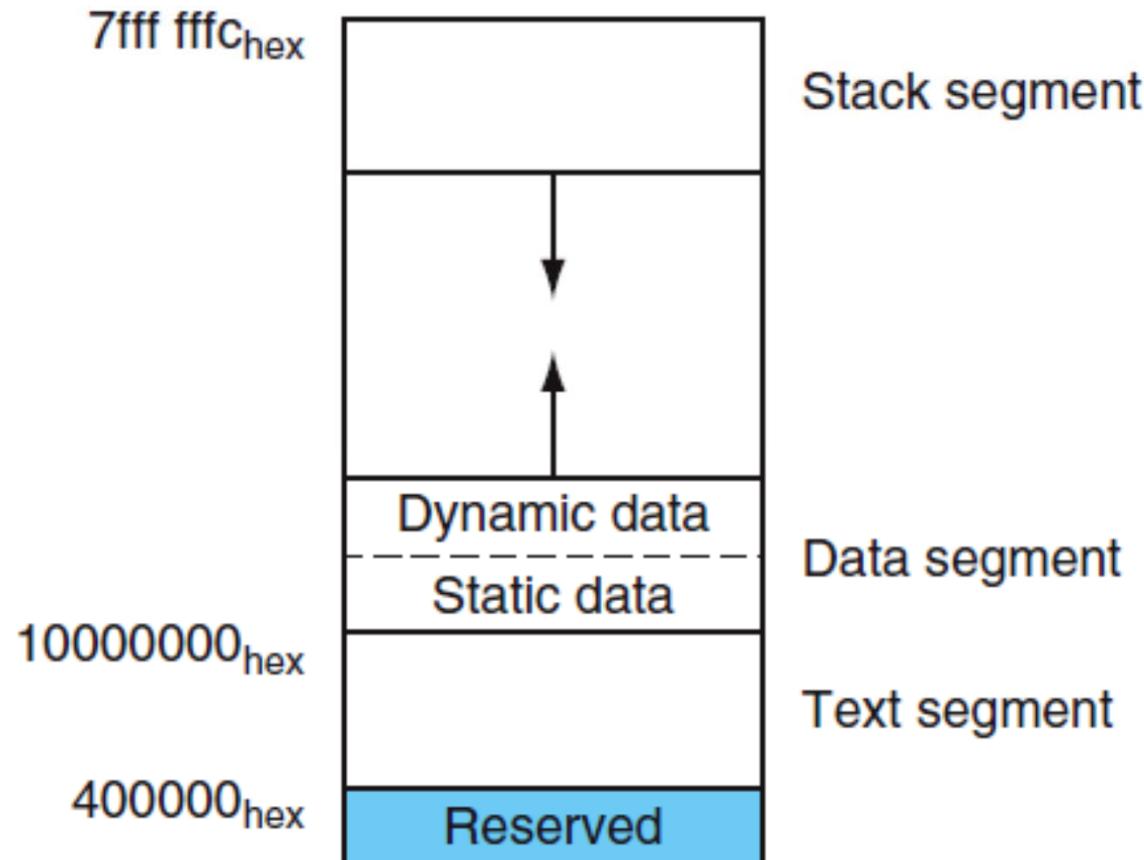
## oggetto procedura B:

dimensione testo: 0x10  
dimensione dati: 0x04  
segmento testo:  
0 bne \$a0, \$0, 1  
4 sw \$a0, 0(\$gp)  
8 lui \$t0, 0  
C ori \$t0, \$t0, 0  
segmento dati:  
0 0  
tabella dei simboli:  
B 0 T  
E 8 T  
Y 0 D  
tabella di rilocazione:  
4 sw Y  
8 lui W  
C ori W



# Organizzazione della memoria

spazio virtuale assoluto del programma



# Accesso all'area dati statici

- Il segmento dati inizia all'indirizzo 0x1000 0000:
  - dunque le istruzioni di accesso alla memoria non possono fare riferimento direttamente agli oggetti in esso contenuti, avendo campi di 16 bit
  - esempio: per caricare la parola all'indirizzo 0x1001 0020 nel registro \$v0 sono necessarie due istruzioni:

```
lui $s0, 0x1001          # 0x1001 significa 1001 in base 16
lw   $v0, 0x0020($s0)    # 0x1001 0000 + 0x0020 = 0x1001 0020
```
- Ottimizzazione: uso di un registro (\$gp) come **puntatore globale** (*global pointer*) al segmento dei dati statici. Ecco come si usa:
  - questo registro viene inizializzato con l'indirizzo 0x1000 8000
  - le istruzioni di *load* e *store* possono utilizzare il loro campo spiazzamento di 16 bit dotato di segno per accedere ai primi 64 kB del segmento dei dati statici
  - ora l'esempio precedente richiede una sola istruzione:

```
lw   $v0, 0x8020($gp)
```
- Il registro *global pointer* accelera l'indirizzamento delle locazioni di memoria comprese tra 0x1000 0000 e 0x1001 0000 rispetto a quello delle altre locazioni dell'area dati.
- Di solito il compilatore memorizza in questa area le **variabili globali di tipo scalare**, poiché esse sono memorizzate in locazioni fisse e sono più adatte a questa modalità di indirizzamento rispetto ad altri dati globali (come i vettori).



## Processo di collegamento

Il collegatore (linker) ha il compito di generare un **solo programma binario eseguibile** (in formato rilocabile) partendo dai diversi moduli.

→ un **solo spazio di indirizzamento** per tutto il programma  
(spazio di indirizzamento virtuale del programma)

Il collegatore (linker) opera in base a queste informazioni:

- la lunghezza del segmento testo e del segmento dati di ciascun modulo tradotto
- e gli indirizzi di impianto

Il collegatore (linker) calcola gli indirizzi dei riferimenti non risolti e completa la traduzione delle istruzioni presenti nelle tabelle di rilocazione.



## Operazioni svolte dal collegatore (linker)

- Determinare la posizione in memoria, cioè l'indirizzo iniziale o di base, delle sezioni codice e dati dei diversi moduli (vedi l'esempio precedente).
- Determinare il nuovo valore di tutti gli indirizzi simbolici che risultano modificati dallo spostamento della base, ossia creare una ***tabella globale dei simboli***.
- Correggere in tutti i moduli i riferimenti agli indirizzi simbolici che sono stati modificati, in base alle tabelle di rilocazione.



# Determinazione della posizione in memoria dei moduli

L'assemblatore alloca la sezione testo e la sezione dati a partire dall'indirizzo base 0.

Ma i moduli non possono essere tutti caricati nella stessa zona di memoria; essi vanno invece caricati sequenzialmente.

Inoltre essi devono rispettare la struttura generale della memoria.

Esempio di riferimento:

Testo del modulo B (base: 0x0040 000C)	Dati del modulo B (base: 0x1000 0008)
Testo modulo MAIN (base: 0x0040 0000)	Dati del modulo MAIN (base: 0x1000 0000)
RESERVED	



## Creazione della tabella globale dei simboli

È costituita dall'unione delle tabelle dei simboli di tutti i moduli da collegare, modificati (rilocati) in base all'indirizzo di base del modulo cui appartengono.

Esempio di riferimento:

Simbolo	Valore iniziale	Base di rilocazione	Valore finale
MAIN	0	0040 0000	0040 0000
X	0	1000 0000	1000 0000
W	4	1000 0000	1000 0004
B	0	0040 000C	0040 000C
E	8	0040 000C	0040 0014
Y	0	1000 0008	1000 0008



# Correzione dei riferimenti nei moduli

Siano:

- ISTR un'istruzione riferita dalla tabella di rilocazione di un modulo M, con simbolo S e indirizzo IND
- IADDR l'indirizzo di tale istruzione nell'eseguibile finale:

$$\text{IADDR} = \text{IND} + \text{BASE\_M}$$

dove BASE\_M è l'indirizzo di base del modulo M

- VS il valore di S nella tabella globale dei simboli
- GP il valore del registro *global pointer*

Regole da applicare in base al tipo di istruzione:

- ISTR è in formato J: inserire VS / 4 nell'istruzione
- ISTR è di salto in formato I: inserire  $(VS - (\text{IADDR} + 4)) / 4$
- ISTR è aritmetico-logica in formato I:
  - inserire i 16 bit meno significativi di VS (VS\_low)
  - ISTR è di tipo *load* o *store*: inserire VS – GP
- ISTR è l'istruzione **lui**:
  - inserire i 16 bit più significativi di VS (VS\_high)



POLITECNICO MILANO 1863

19

Indirizzo	Istruzione completa	Note
<b>Segmento codice (testo)</b>		
0040 0000	<b>lw</b> \$a0, <b>0x8000</b> (\$gp)	X – GP; 0x1000 0000 – 0x1000 8000
0040 0004	<b>beq</b> \$a0, \$0, <b>0x0003</b>	(E – (IADDR+4)) / 4; (0x0040 0014 – 0x0040 0008) / 4
0040 0008	<b>jal</b> <b>0x0010 0003</b>	B / 4; 0x0040 000C / 4
0040 000C	<b>bne</b> \$a0, \$0, 1	autorilocante; inizio modulo B
0040 0010	<b>sw</b> \$a0, <b>0x8008</b> (\$gp)	Y – GP; 0x1000 0008 – 0x1000 8000
0040 0014	<b>lui</b> \$t0, <b>0x1000</b>	W_high
0040 0018	<b>ori</b> \$t0,\$t0, <b>0x0004</b>	W_low
<b>Segmento dati</b>		
1000 0000	128	valore iniziale di X
1000 0004	0x1234 5678	valore iniziale di W
1000 0008	0	valore iniziale di Y



## Il file eseguibile del programma completo

intestazione	codice eseguibile	valori iniziali dei dati statici (variabili globali)	tabella globale dei simboli (facoltativa)
--------------	-------------------	--	---

Somiglia al file oggetto di un singolo modulo, ma ormai contiene un *programma completo* (unione di tutti i moduli) *eseguibile*, e informazioni su come caricarlo in memoria e gestirlo.

L'**intestazione** ha dimensioni fisse e contiene alcuni campi necessari per guidare il lancio del programma. Specifica le *dimensioni* del *codice eseguibile* e dei *valori iniziali dei dati*, e l'*indirizzo dell'istruzione iniziale* del programma, cioè l'istruzione con etichetta MAIN.

Il **codice eseguibile** è la *lista delle istruzioni* del programma. Tutte le istruzioni sono ormai in forma numerica completa, ossia tutti i loro campi sono numericamente risolti. Al lancio del programma, esse verranno caricate in memoria *in sequenza* nel segmento di testo.

I **valori iniziali dei dati statici (variabili globali)** sono la *lista dei valori iniziali da dare ai dati* (a quelli inizializzati – gli altri vengono azzerati per default). Al lancio del programma, i valori vengono caricati in memoria nel segmento dati, *agli indirizzi dei dati corrispondenti*.

La **tabella globale dei simboli** è includibile facoltativamente a solo *scopo di debug*. Se è presente, al lancio del programma viene caricata con il codice. Eseguendo il programma passo-passo per farne il debug, quando un'istruzione genera un *indirizzo numerico*, consultando la tabella il sistema di debug trova il *simbolo corrispondente* e lo visualizza. Serve per aiutare il programmatore a comprendere meglio il tipo di errore e a correggerlo.



## Caricamento ed esecuzione

Nei Sistemi Operativi di famiglia UNIX (Linux), il *nucleo (kernel)* del SO carica il programma nella memoria principale e ne lancia l'esecuzione. Operazioni:

1. Legge l'intestazione del file eseguibile per determinare le dimensioni dei segmenti *testo* e *dati*, e per conoscere *l'indirizzo dell'istruzione iniziale* del programma.
2. Crea un nuovo spazio di indirizzamento per il programma. Questo spazio è abbastanza grande per contenere i segmenti di *testo* (con eventuale tabella globale dei simboli) e dei *dati*, nonché un segmento per la *pila (stack)*.
3. Carica le istruzioni e i valori iniziali dei dati dal file eseguibile alla memoria, mettendo tutto quanto all'interno del nuovo spazio di indirizzamento.
4. Carica in pila gli eventuali argomenti passati al programma (`argc` e `argv` in ling. C).
5. Inizializza i registri dell'architettura. In generale, la maggior parte dei registri viene azzerata, tranne il registro *stack pointer*, cui viene assegnato l'indirizzo della prima cella libera della pila, e (per MIPS) il registro *global pointer*, come già visto.
6. Esegue un'apposita procedura di avvio, che fa parte del nucleo del SO. Tale procedura copia gli argomenti del programma dalla pila ai registri e poi chiama la procedura *main* del programma, saltando all'indirizzo dell'istruzione iniziale specificato nel file eseguibile (cioè salta all'istruzione con etichetta *MAIN*). Quando *main* termina, la procedura di avvio conclude il programma tramite la chiamata di sistema *exit*.





---

# AXO - Architettura dei Calcolatori e Sistemi Operativi

## reti combinatorie



# Sommario

---

- Il segnale binario
- Algebra di Boole e funzioni logiche
- Porte logiche
- Analisi e sintesi di circuiti combinatori



# 1- Segnali e informazioni

---

- Per elaborare informazioni, occorre rappresentarle (o codificarle)
- Per rappresentare (o codificare) le informazioni si usano segnali
- I segnali devono essere elaborati, nei modi opportuni, tramite dispositivi di elaborazione
- In un **sistema digitale** le informazioni vengono **rappresentate**, **elaborate** e **trasmesse** mediante grandezze fisiche (segnali) che si considerano assunere solo **valori discreti**. Ogni valore è associato a una cifra (**digit**) della rappresentazione.



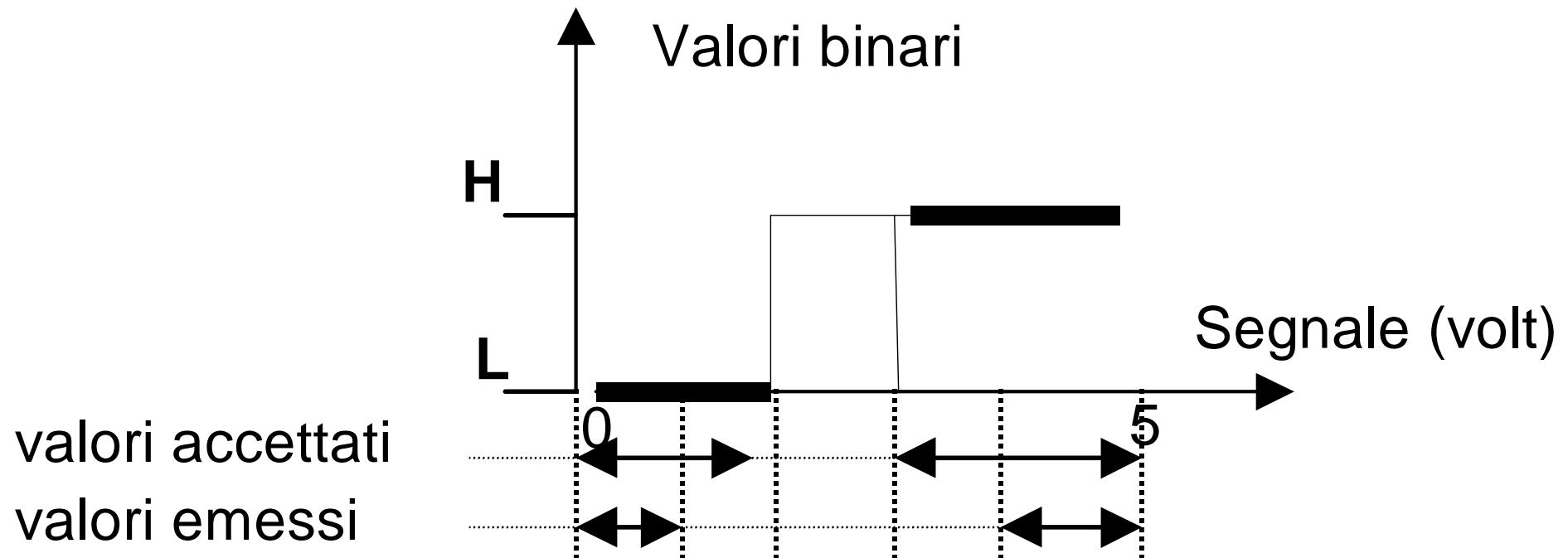
# Il segnale binario (i)

- **Segnale binario**: una grandezza che può assumere due valori distinti, convenzionalmente indicati con 0 e 1
  - $s \in \{0, 1\}$  (low, high - False, True)
- **Grandezze fisiche** utilizzate in un sistema digitale per la rappresentazione dell'informazione:
  - ⇒ segnali elettrici (tensione, corrente)
  - ⇒ grandezze di tipo magnetico (stato di magnetizzazione)
  - ⇒ segnali ottici



## Il segnale binario (ii)

La grandezza fisica che si utilizza (segnale elettrico di tensione) assume solo **due valori discreti** (binaria)





## Il segnale binario (iii)

- Elaborazione del segnale binario: si usano due classi di dispositivi di elaborazione
  - **reti combinatorie**: l'uscita all'istante  $t$  dipende dagli ingressi nello stesso istante
  - **reti sequenziali** (reti con memoria): l'uscita all'istante  $t$  dipende dagli ingressi nello stesso istante e dalla "storia passata" (= **stato della rete**)
- Sono tutti circuiti digitali (o numerici)



## 2 - Algebra di Commutazione

---

- Deriva dall'**algebra di Boole** e consente di descrivere matematicamente i circuiti digitali (o circuiti logici)
  - Definisce le **espressioni logiche** che descrivono il **comportamento** del circuito da realizzare nella forma  $U=f(I)$
  - A partire dalle equazioni logiche è possibile derivare la **realizzazione circuitale** (rete logica)
  - I componenti dell'algebra di Boole sono: le **variabili di commutazione**, gli **operatori fondamentali** e le **proprietà degli operatori** logici tramite le quali è possibile trasformare le espressioni logiche
-



# Variabili di commutazione e operatori

- Una **variabile di commutazione** (o variabile logica) corrisponde al singolo bit di informazione rappresentata e elaborata
- Gli **operatori fondamentali** sono

Negazione	$\neg A$ oppure $/A$	= 1 per $A=0$ = 0 per $A=1$
Somma logica	$A + B$	= 0 se e solo se $A=B=0$
Prodotto logico	$A \cdot B$	= 1 se e solo se $A=B=1$

Precedenza degli operatori: negazione, prodotto, somma



# Operatori logici

Somma

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

Prodotto

$$0 \ 0 = 0$$

$$0 \ 1 = 0$$

$$1 \ 0 = 0$$

$$1 \ 1 = 1$$

Inversione

$$\mathbf{!0 = 1}$$

$$\mathbf{!1 = 0}$$



# Proprietà degli operatori logici (1)

Legge	AND	OR
Identità	$1 \cdot A = A$	$0 + A = A$
Elemento nullo	$0 \cdot A = 0$	$1 + A = 1$
Idempotenza	$A \cdot A = A$	$A + A = A$
Inverso	$A \cdot !A = 0$	$A + !A = 1$



# Proprietà degli operatori logici (2)

Legge	AND	OR
Commutativa	$A \cdot B = B \cdot A$	$A + B = B + A$
Associativa	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C =$ $A + (B + C)$
Distributiva	AND rispetto a OR $A \cdot (B + C) =$ $A \cdot B + A \cdot C$	OR rispetto a AND $A + B \cdot C =$ $(A + B) \cdot (A + C)$
Assorbimento	$A \cdot (A + B) = A$	$A + A \cdot B = A$
De Morgan	$!(A \cdot B) = !A + !B$	$!(A + B) = !A \cdot !B$



## Esempio di trasformazione

□  $F(a,b,c) = !a!bc + !abc + !ab!c$

Espressione trasformata	proprietà utilizzata
$!a!bc + !abc + !ab!c$	idempotenza $x+x=x$
$!a!bc + !abc + !abc + !ab!c$	distributiva $xy + xz = x(y+z)$
$!ac(!b+b) + !ab(c+!c)$	inverso $x + !x = 1$
$!ac1 + !ab1$	identità $x1=x$
$!ac + !ab$	distributiva
$!a(c + b)$	



# Funzioni combinatorie

- Una **funzione combinatoria** (o funzione booleana, o funzione logica) corrisponde a un'**espressione booleana**, contenente una o più variabili booleane e gli operatori booleani AND, OR e NOT
  - Dando dei valori alle variabili booleane della funzione combinatoria, si calcola il corrispondente valore della funzione
- Esempio: funzione logica a 2 ingressi a e b e 2 uscite S e C
  - $S=1$  se e solo se solo uno degli ingressi vale 1
  - $C=1$  se e solo se entrambi gli ingressi valgono 1
- Espressioni booleane
  - $S=a!b + !ab$
  - $C=ab$



# Tabella delle verità

- Per specificare il comportamento di una funzione combinatoria è possibile specificare, per ogni possibile configurazione degli ingressi, il valore dell'uscita: **tabella delle verità**
- La tabella della verità di una funzione a  $n$  ingressi ha  $2^n$  righe, che corrispondono a tutte le possibili configurazioni di ingresso
- La tabella delle verità ha due “gruppi” di colonne:
  - **colonne degli ingressi**, le cui righe contengono tutte le combinazioni di valori delle variabili della funzione
  - **colonna dell'uscita**, che riporta i corrispondenti valori assunti dalla funzione



## Esempio

---

- $F(A, B, C) = AB + !C$  è una funzione combinatoria a 3 variabili A, B e C
    - $F(0, 0, 0) = 0 \ 0 + !0 = 0 + 1 = 1$
    - $F(0, 0, 1) = 0 \ 0 + !1 = 0 + 0 = 0$
    - $F(0, 1, 0) = 0 \ 1 + !0 = 0 + 1 = 1$
    - ... (e così via)
    - $F(1, 1, 1) = 1 \ 1 + !1 = 1 + 0 = 1$
-



# Esempio: Tabella delle verità

# riga	A	B	C	A B + /C	F
0	0	0	0	0 0 + /0	1
1	0	0	1	0 0 + /1	0
2	0	1	0	0 1 + /0	1
3	0	1	1	0 1 + /1	0
4	1	0	0	1 0 + /0	1
5	1	0	1	1 0 + /1	0
6	1	1	0	1 1 + /0	1
7	1	1	1	1 1 + /1	1

n = 3  
ingressi

$2^n = 2^3 = 8$   
righe

colonna  
uscita

(per comodità nella colonna centrale  
è riportato anche il calcolo)



## 3 - Porte logiche (circuiti combinatori elementari)

---

- I circuiti digitali sono formati da componenti digitali elementari, chiamati porte logiche
- Le **porte logiche** sono i circuiti minimi per l'elaborazione di segnali binari e **corrispondono agli operatori elementari** dell'algebra di commutazione
- Le porte logiche vengono classificate in base al **modo di funzionamento**: porta **NOT**, porta **AND**, porta **OR** (sono le porte logiche fondamentali e costituiscono un **insieme di operatori funzionalmente completo**)
  - Classificazione per numero di ingressi: porte a 1 ingresso, porte a 2 ingressi, porte 3 ingressi, e così via ...

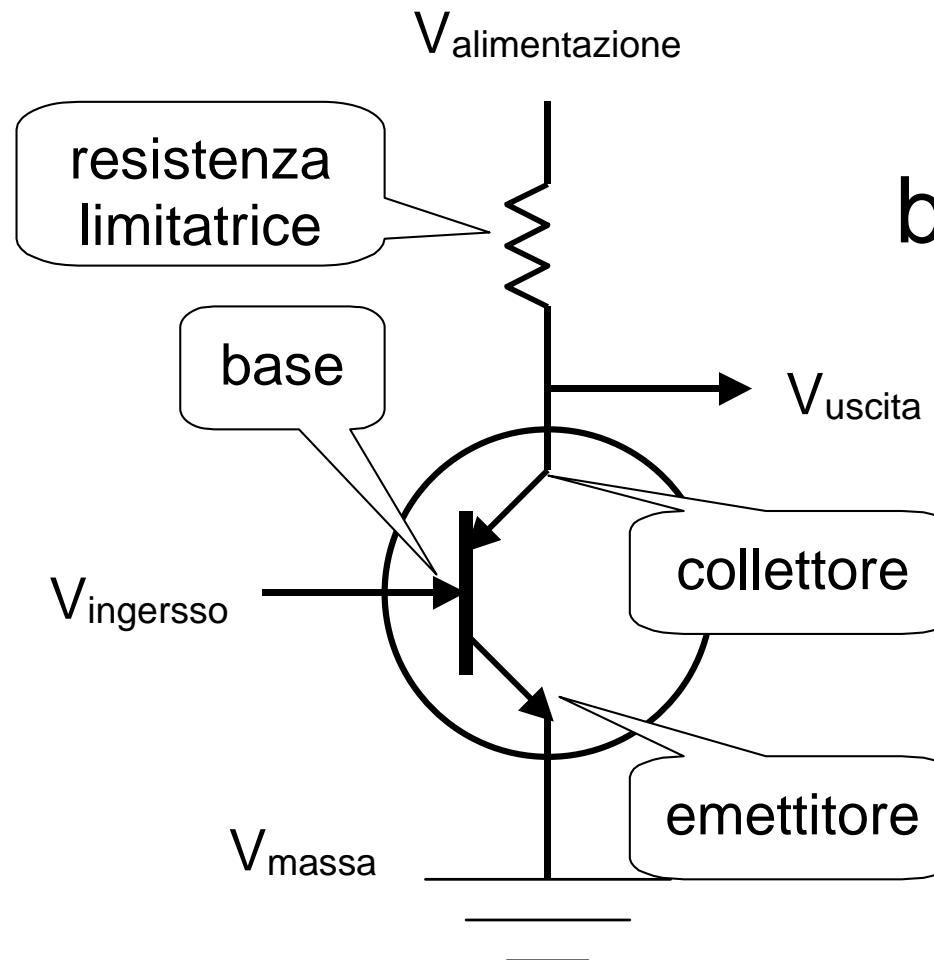


# Transistor

- L'elemento funzionale fondamentale per la costruzione di porte logiche è il transistor
- Il transistor è un dispositivo elettronico
- Il transistor opera su grandezze elettriche: tensione e corrente
- Il transistor funziona come un interruttore
- Ha due stati di funzionamento: interruttore aperto o interruttore chiuso



# Struttura del transistor



**transistor  
bipolare (BJT)**

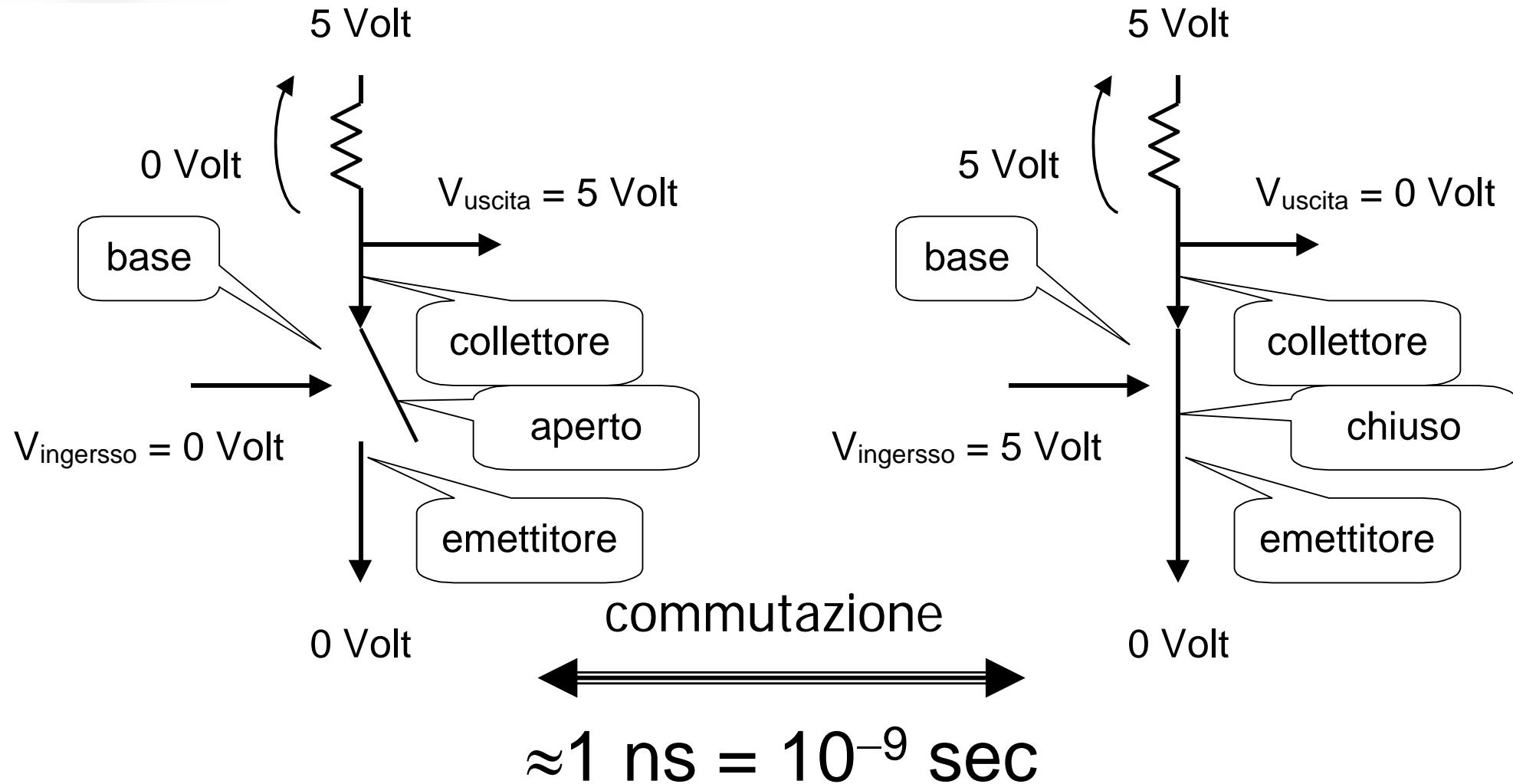


# Funzionamento del transistor

- Se la **tensione di base**  $V_{\text{ingresso}}$  è **inferiore** a una data **soglia** critica, il transistor si comporta come un **interruttore aperto**, cioè tra emettitore e collettore non passa corrente, e quindi la tensione di uscita diventa uguale a quella di alimentazione:  $V_{\text{uscita}} = V_{\text{alimentazione}} = 5 \text{ Volt}$  (in tecnologia TTL)
  
- Se la **tensione di base**  $V_{\text{ingresso}}$  è **superiore** a una data **soglia** critica, il transistor si comporta come un **interruttore chiuso**, cioè tra emettitore e collettore passa corrente, e quindi la tensione di uscita diventa uguale a quella di massa:  $V_{\text{uscita}} = V_{\text{massa}} = 0 \text{ Volt}$  (in tecnologia TTL)



# Funzionamento del transistor





# La porta NOT (invertitore)

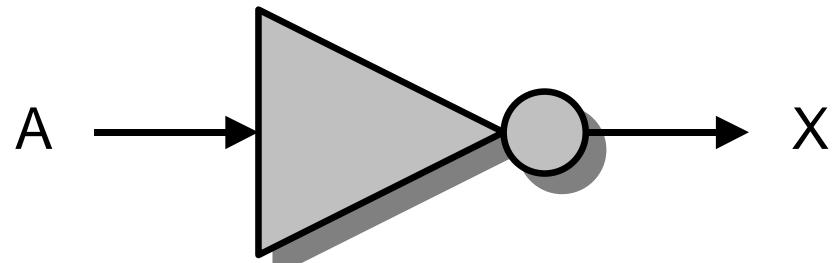
- Il singolo transistor della figura è una porta NOT
- Se l'ingresso vale 0 Volt, l'uscita vale 5 Volt
- Se l'ingresso vale 5 Volt, l'uscita vale 0 Volt
  
- La tabella rappresenta il funzionamento della porta NOT

$V_{\text{ingresso}}$	$V_{\text{uscita}}$
0 Volt	5 Volt
5 Volt	0 Volt



# Porta NOT (invertitore, negatore)

Simbolo funzionale



(a 1 ingresso)



simbolo semplificato

Tabella delle verità

A	X
0	1
1	0

L'uscita vale 1 se e solo se l'ingresso vale 0



# Porta AND

## Simbolo funzionale

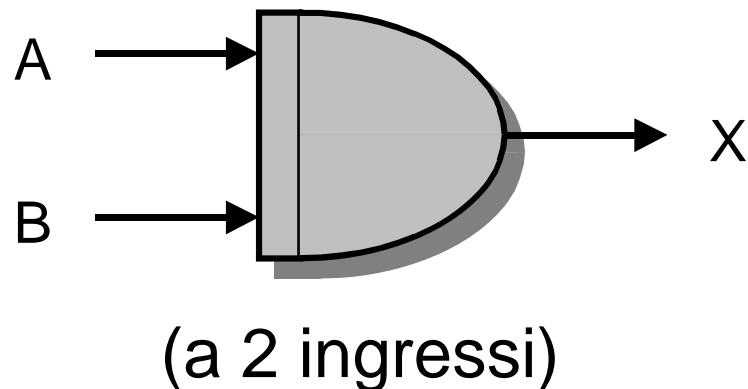


Tabella delle verità

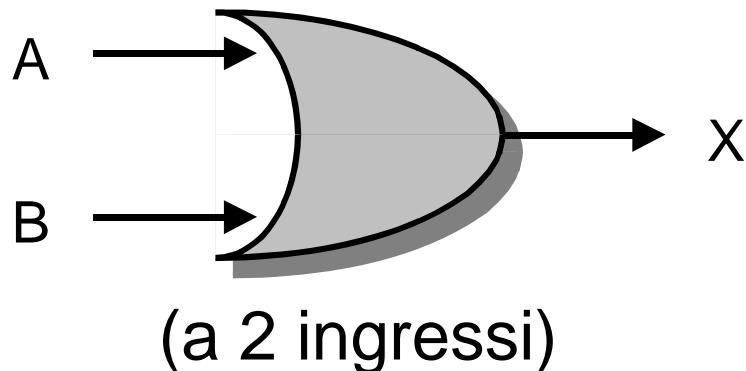
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

L'uscita vale 1 se e solo se entrambi gli ingressi valgono 1



# Porta OR

## Simbolo funzionale



## Tabella delle verità

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

L'uscita vale 1 se e solo se almeno un ingresso vale 1



# NAND (operatore funzionalmente completo)

Simbolo funzionale

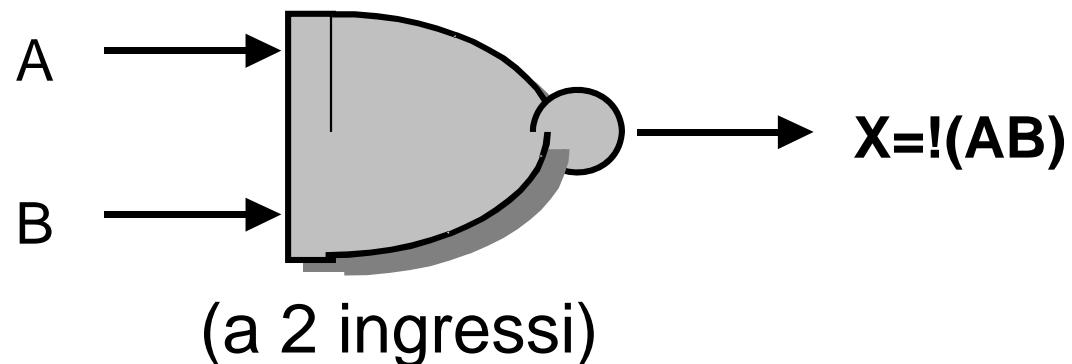


Tabella delle verità

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

L'uscita vale 0 se e solo se entrambi gli ingressi valgono 1



# NOR (operatore funzionalmente completo)

## Simbolo funzionale

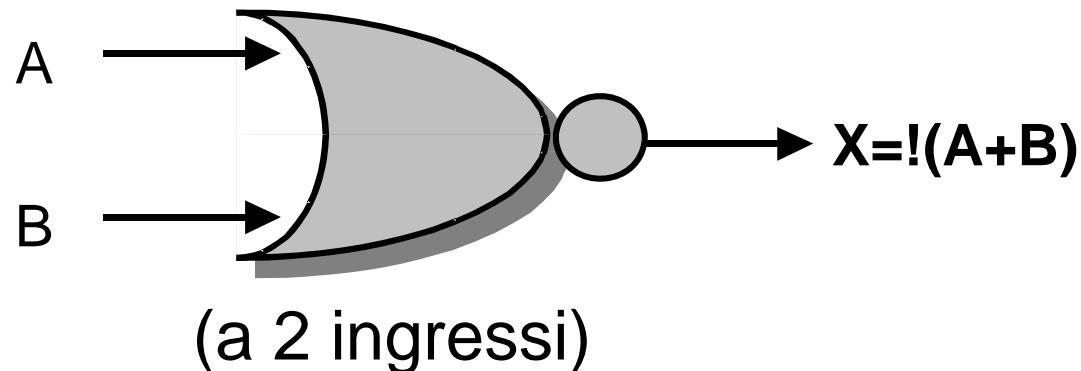


Tabella delle verità

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

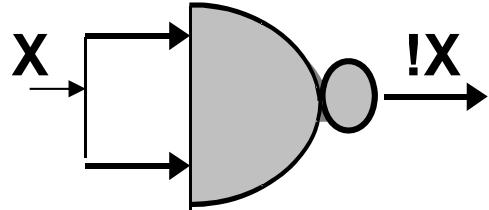
L'uscita vale 1 se e solo se entrambi gli ingressi valgono 0



# NAND - realizzazione di NOT, AND, OR

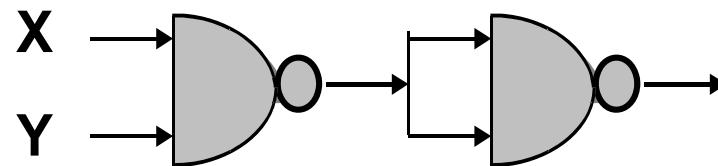
**NOT**

$$!X = !(XX)$$



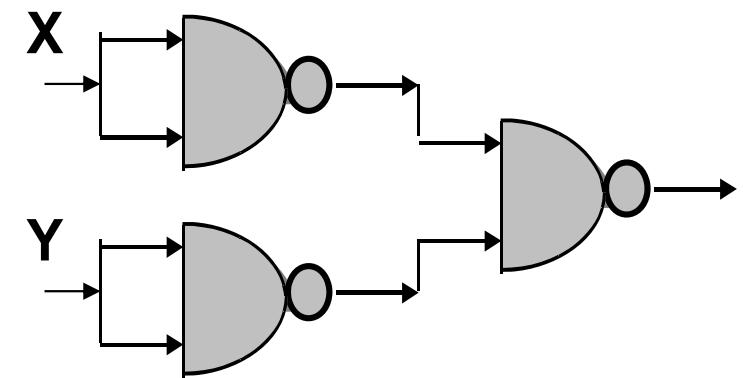
**AND**

$$XY = !! (XY)$$



**OR**

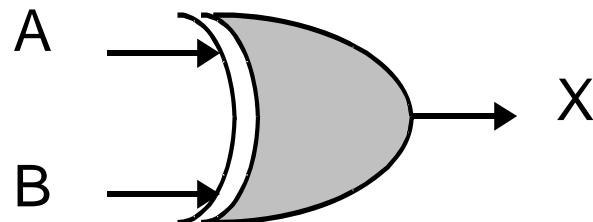
$$X+Y = !! (X+Y) = !(X!Y)$$





## Altri operatori: OR esclusivo (X-OR)

- X-OR a 2 ingressi: l'uscita vale 1 se e solo se una sola variabile vale 1 (diseguaglianza)
  - generalizzato a n variabili di ingresso: l'uscita vale 1 se e solo se il numero di 1 è dispari

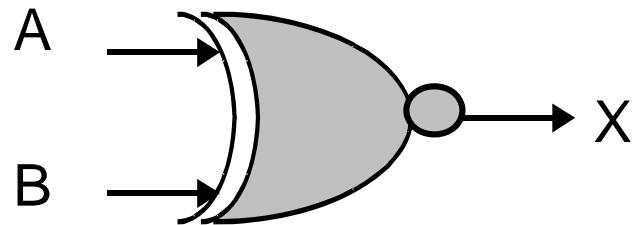


A	B	$X = !AB + A!B$
0	0	0
0	1	1
1	0	1
1	1	0



# Altri operatori: X-NOR esclusivo

- X-NOR a 2 ingressi: l'uscita vale 1 se e solo se entrambe le variabili valgono 0 o 1 (eguaglianza)
  - generalizzato a n variabili di ingresso: l'uscita vale 1 se e solo se il numero di 1 è pari



A	B	$X = AB + !A!B$
0	0	1
0	1	0
1	0	0
1	1	1



# Generalizzazioni

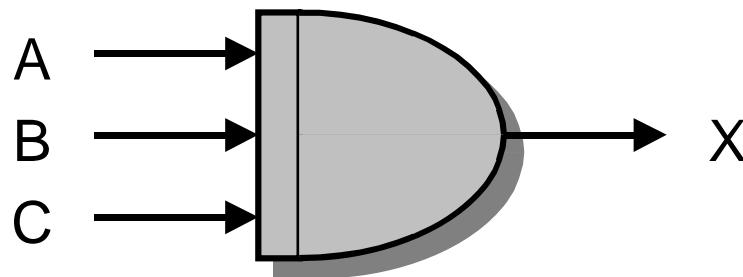
---

- Alcuni tipi di porte a 2 ingressi si possono generalizzare a 3, 4, ecc ingressi
- Le due porte a più ingressi maggiormente usate sono la porta AND e la porta OR
- Tipicamente si usano AND (o OR) a 2, 4 o 8 ingressi (raramente più di 8)



# Porta AND a 3 ingressi

## Simbolo funzionale



L'uscita vale 1 se e solo se tutti e 3 gli ingressi valgono 1

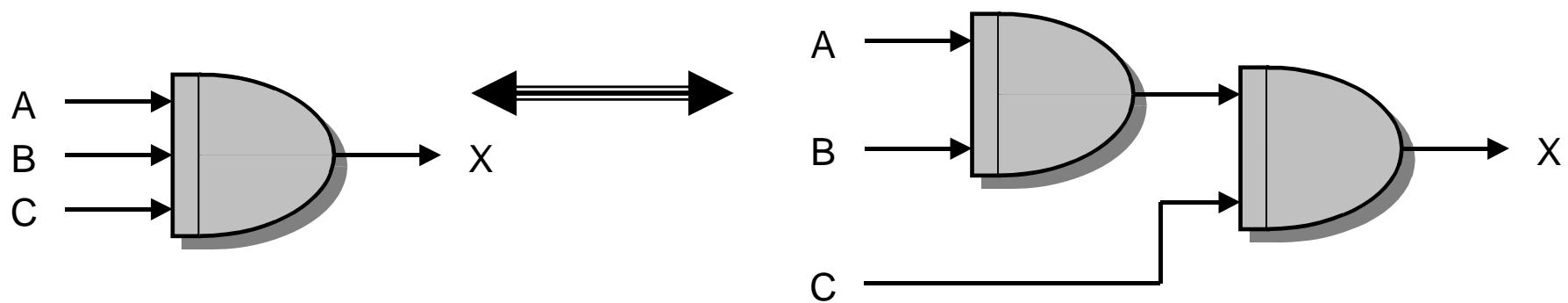
Tabella delle verità

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



# Realizzazione ad albero

- La porta AND a 3 ingressi si realizza spesso come albero di porte AND a 2 ingressi (ma non è l'unico modo)

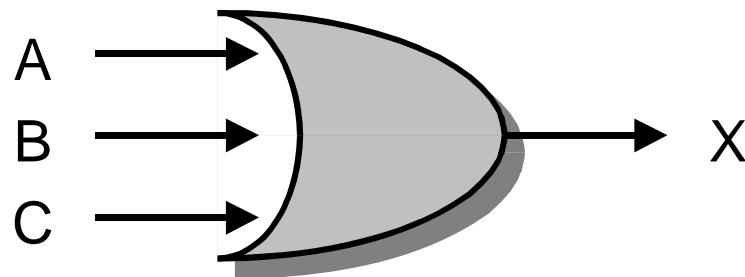


- Nota bene: non tutti i tipi di porte a più di 2 ingressi si possono realizzare come alberi di porte a 2 ingressi (funziona sempre con AND, OR, X-OR, X-NOR)



# Porta OR a 3 ingressi

## Simbolo funzionale



L'uscita vale 0 se e solo se tutti e 3 gli ingressi valgono 0

Tabella delle verità

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



# Costo e velocità di una porta logica

## □ Costo di realizzazione

- Il numero di transistor per realizzare una porta dipende dalla tecnologia, dalla funzione e dal numero di ingressi
- Porta NOT: 1 oppure 2 transistor, porte AND e OR: 3 oppure 4 transistor, altre porte:  $\geq 4$  transistor

## □ Velocità di commutazione

- La velocità di commutazione di una porta dipende dalla tecnologia, dalla funzione e dal numero di ingressi
- Le porte più veloci (oltre che più piccole) sono tipicamente le porte NAND e NOR a 2 ingressi: possono commutare in meno di 1 nanosecondo ( $10^{-9}$  sec, un miliardesimo di sec)

## □ Il costo delle porte logiche e la velocità di commutazione consentono di calcolare un'indicazione del **costo della rete logica** che realizza un'espressione booleana e del **ritardo di propagazione** associato alla rete stessa



---

## 4 - Analisi e sintesi di reti combinatorie



# Rete combinatoria (1)

---

- A ogni **funzione combinatoria**, data come **espressione booleana**, si può sempre associare un circuito digitale, formato da porte logiche, che viene chiamato **rete combinatoria**
  - Gli ingressi della rete combinatoria sono le variabili della funzione
  - L'uscita della rete combinatoria emette il valore assunto dalla funzione
-



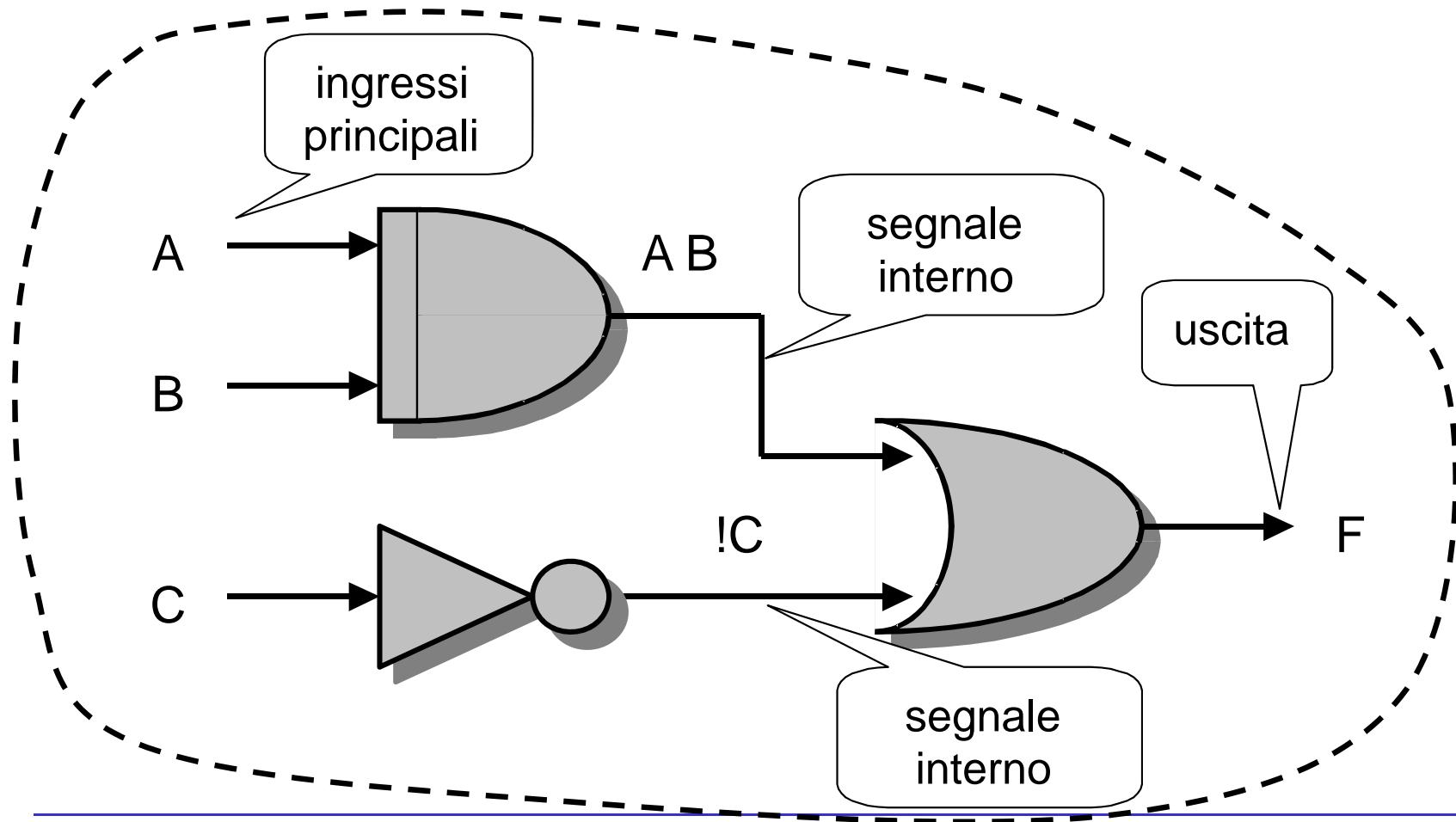
## Rete combinatoria (2)

- Una **rete combinatoria** è un circuito digitale:
  - dotato di  $n \geq 1$  ingressi principali e di un'uscita
  - formato da porte logiche AND, OR e NOT (eventualmente anche da altri tipi di porte)
  - e privo di retroazioni
- Costruzione della rete combinatoria a partire dalla funzione logica
  - variabili e variabili negate
  - ogni termine dell'espressione è sostituito dalla corrispondente rete di porte fondamentali
  - le uscite corrispondenti ad ogni termine si compongono come indicato dagli operatori ...



## Esempio

$$F(A, B, C) = A B + !C$$





# Simulazione circuitale

## (dalla rete alla tabella della verità)

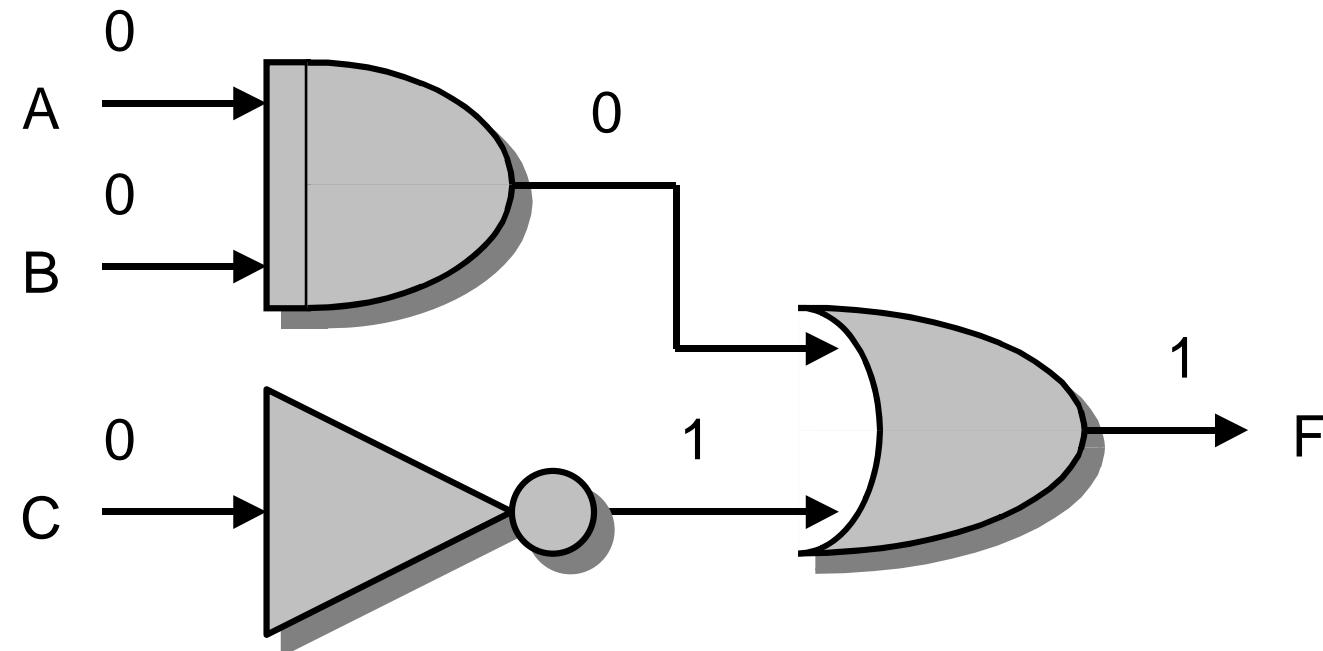
---

- La **tabella delle verità** di una rete combinatoria può anche essere ricavata per **simulazione** del funzionamento circuitale della **rete combinatoria** stessa
  
- Per simulare il funzionamento circuitale di una rete combinatoria, si applicano dei valori agli ingressi, e li si propaga lungo la rete fino all'uscita



# Simulazione circuitale

(corrisponde alla riga 0 della tabella)

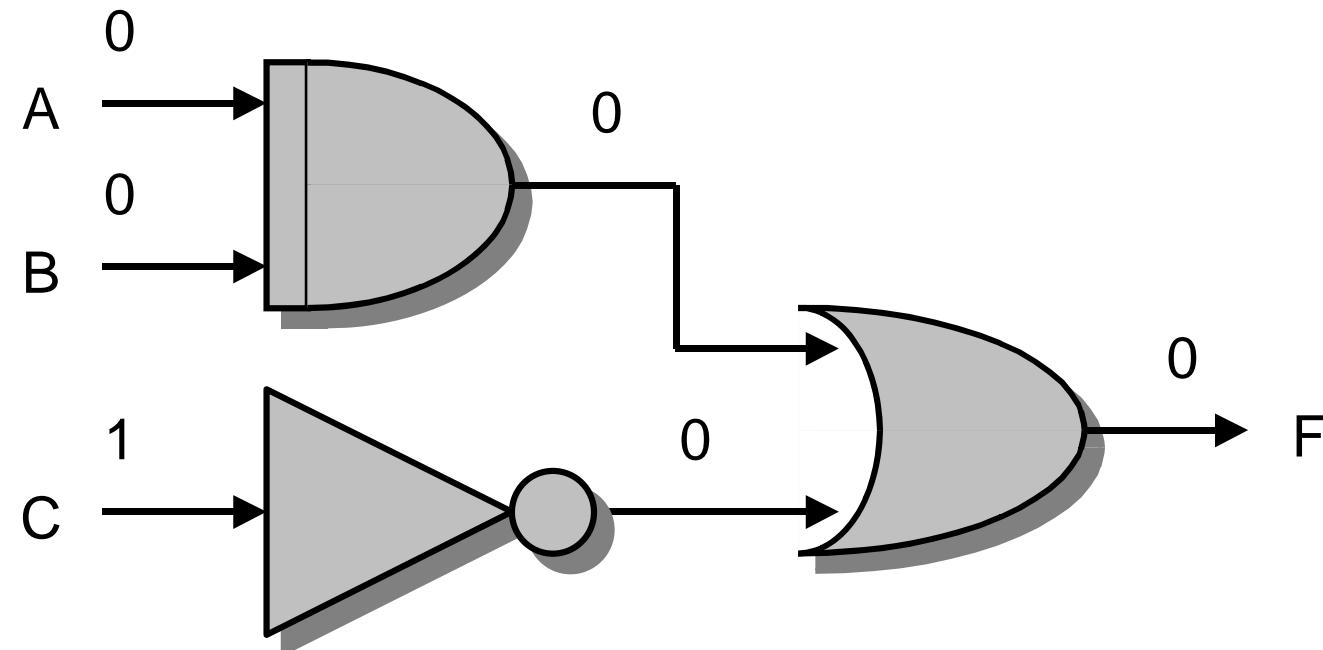


Risultato della simulazione:  $F(0, 0, 0) = 1$



# Simulazione circuitale

(corrisponde alla riga 1 della tabella)

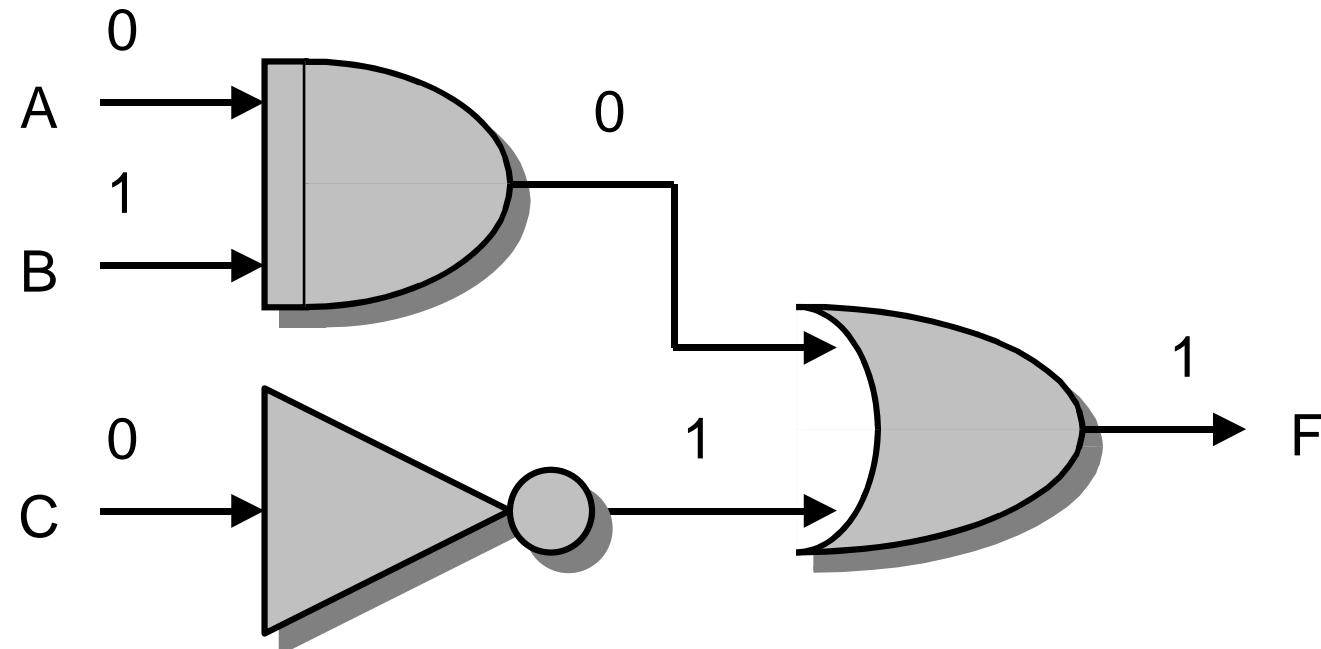


Risultato della simulazione:  $F(0, 0, 1) = 0$



# Simulazione circuitale

(corrisponde alla riga 2 della tabella)

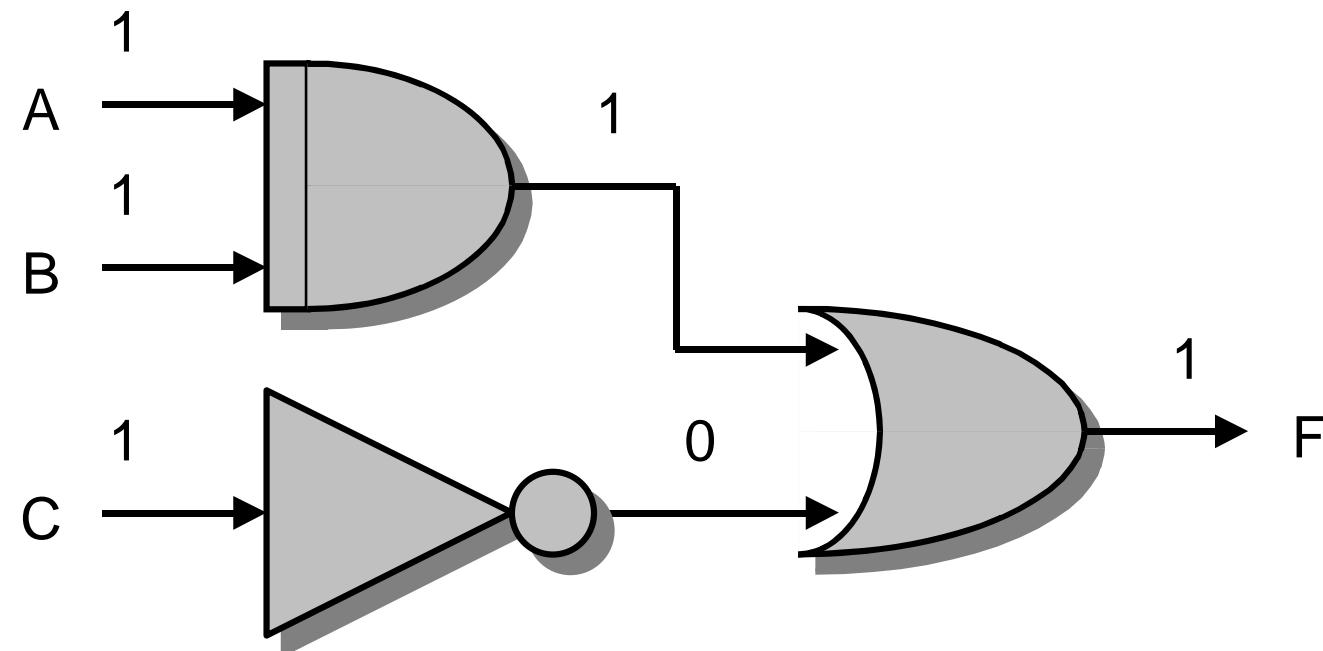


Risultato della simulazione:  $F(0, 1, 0) = 1$



# Simulazione circuitale

(corrisponde alla riga 7 della tabella)



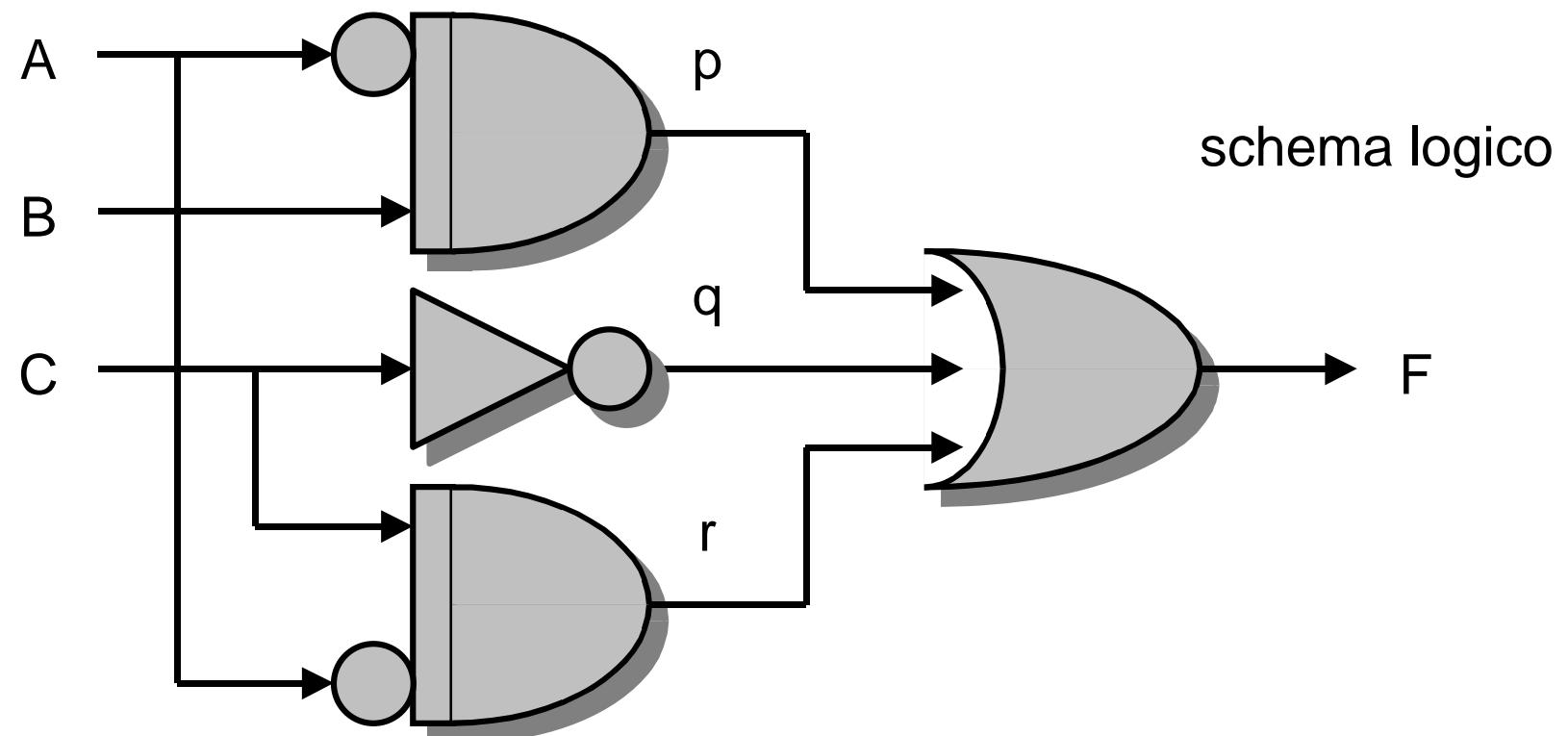
Risultato della simulazione:  $F(1, 1, 1) = 1$



## 4a - Analisi di reti combinatorie

- dalla rete all'espressione

(1) Si applicano nomi ai segnali interni: p, q e r





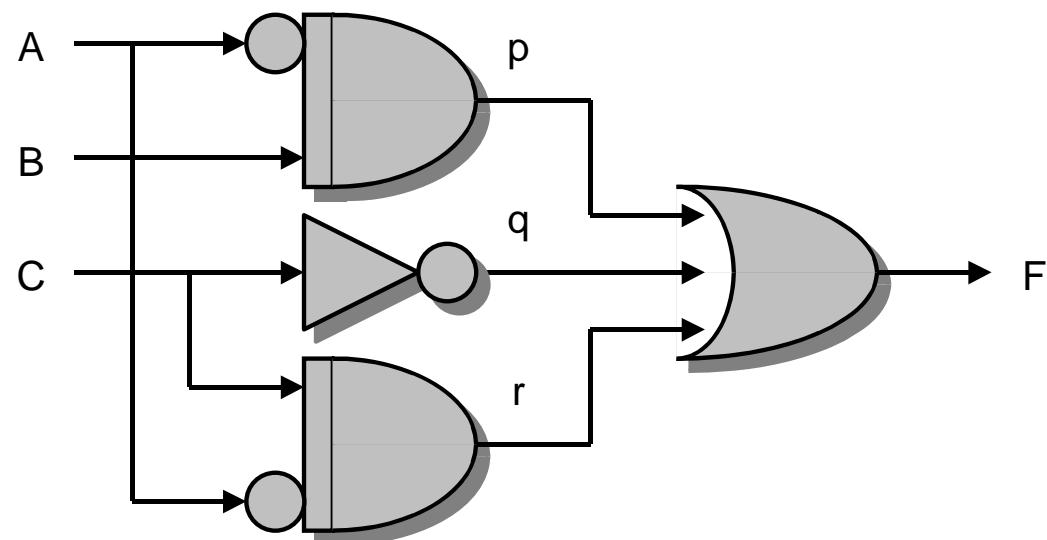
## 4a - Analisi di reti combinatorie

- dalla rete all'espressione

(2) Si ricavano le espressioni booleane corrispondenti ai segnali interni:

- $p = !A \cdot B$
- $q = !C$
- $r = !A \cdot C$

schema logico





## 4a - Analisi di reti combinatorie

### - dalla rete all'espressione

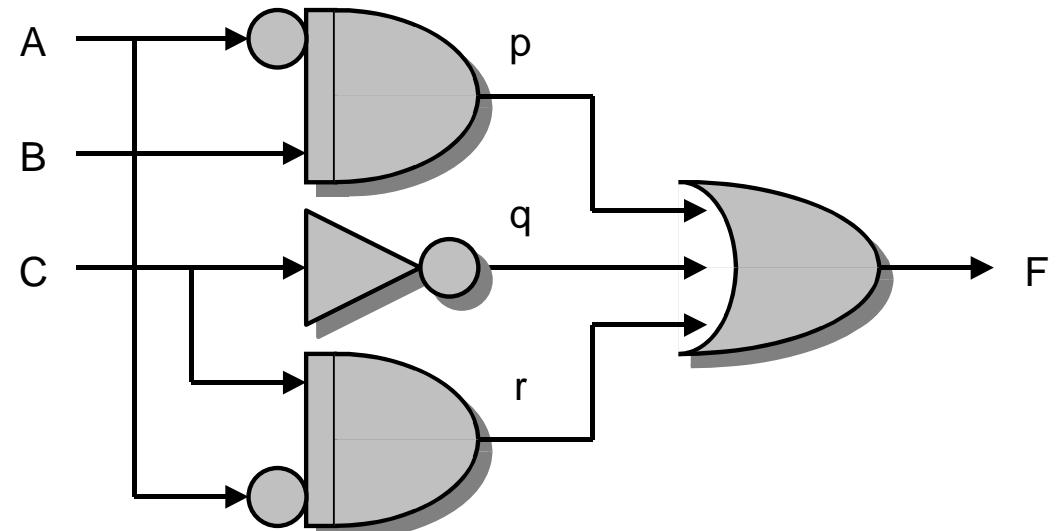
(3) Si ricava l'uscita come espressione booleana in funzione dei segnali interni:

- $F = p + q + r$

(4) Per sostituzione, si ricava l'uscita come espressione booleana in funzione degli ingressi principali:

- $F = p + q + r$
- $F(A, B, C) = !A B + !C + !A C$

L'espressione booleana così trovata ha una struttura conforme allo schema logico di partenza





## 4a - Analisi di reti combinatorie

- dall'espressione alla tabella delle verità

(per comodità è riportato  
anche il calcolo)

# riga	A	B	C	$/A \cdot B + /C + /A \cdot C$	F
0	0	0	0	$/0 \cdot 0 + /0 + /0 \cdot 0$	1
1	0	0	1	$/0 \cdot 0 + /1 + /0 \cdot 1$	1
2	0	1	0	$/0 \cdot 1 + /0 + /0 \cdot 0$	1
3	0	1	1	$/0 \cdot 1 + /1 + /0 \cdot 1$	1
4	1	0	0	$/1 \cdot 0 + /0 + /1 \cdot 0$	1
5	1	0	1	$/1 \cdot 0 + /1 + /1 \cdot 1$	0
6	1	1	0	$/1 \cdot 1 + /0 + /1 \cdot 0$	1
7	1	1	1	$/1 \cdot 1 + /1 + /1 \cdot 1$	0

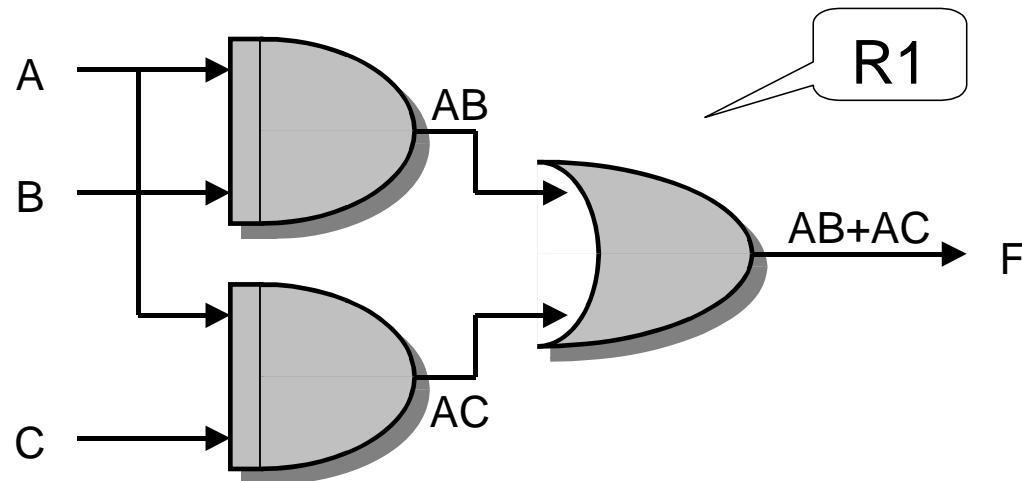


# Reti combinatorie equivalenti

- Una funzione combinatoria può ammettere più **reti combinatorie differenti** che la sintetizzano
- Reti combinatorie che realizzano la medesima funzione combinatoria si dicono **equivalenti**
- Esse hanno tutte la stessa funzione (tabella delle verità), ma possono avere struttura (e costo) differente



# Due reti equivalenti



$$F_1 = AB + AC$$

$$F_2 = A(B + C)$$

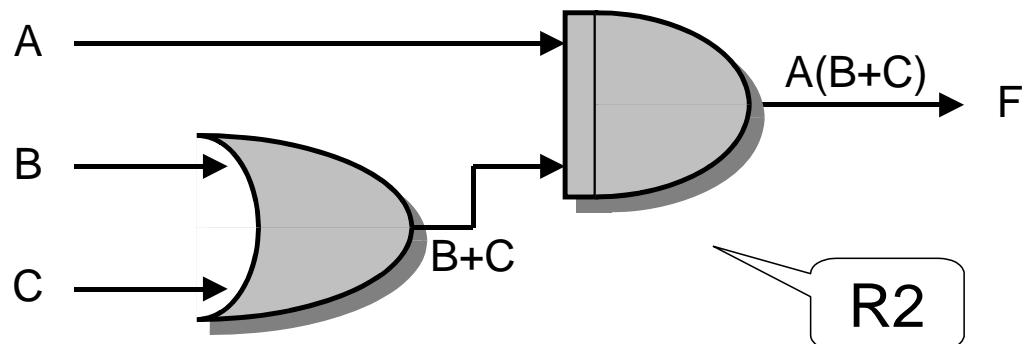
Trasformazione:

$$F_1 = AB + AC =$$

$$= A(B + C) =$$

$$= F_2$$

(prop. distributiva)



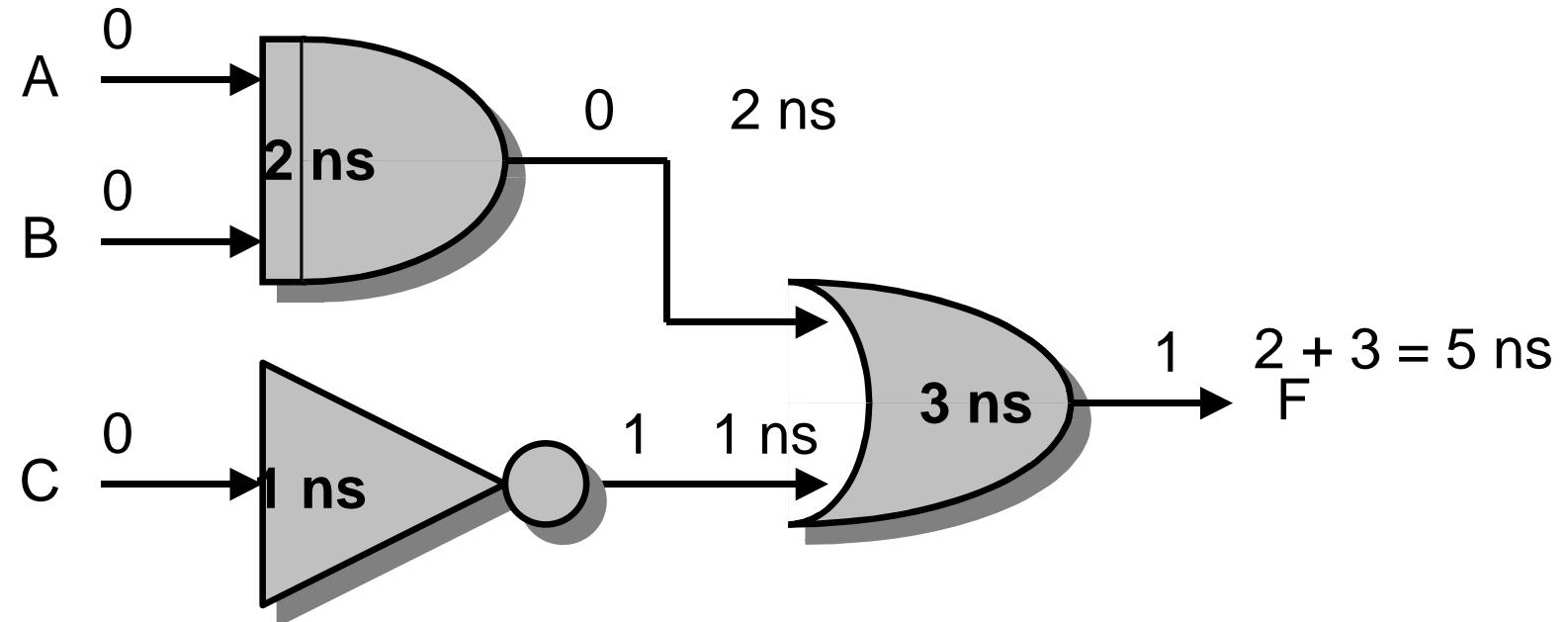


# Costo e velocità di reti combinatorie

- Il costo di una rete combinatoria si valuta in vari modi (criteri di costo):
  - Numero di porte, per tipo di porta e per quantità di ingressi della porta
  - Numero di porte universali (NAND o NOR)
  - Numero di transistor
  - Complessità delle interconnessioni
  - e altri ancora ...
- La velocità di una rete combinatoria è misurata dal tempo che una variazione di ingresso impiega per modificare l'uscita della rete (o ritardo di propagazione)
  - Per calcolare la velocità di una rete combinatoria, occorre conoscere i ritardi di propagazione delle porte logiche componenti la rete, e poi analizzare i percorsi ingressi-uscita



# Velocità: esempio

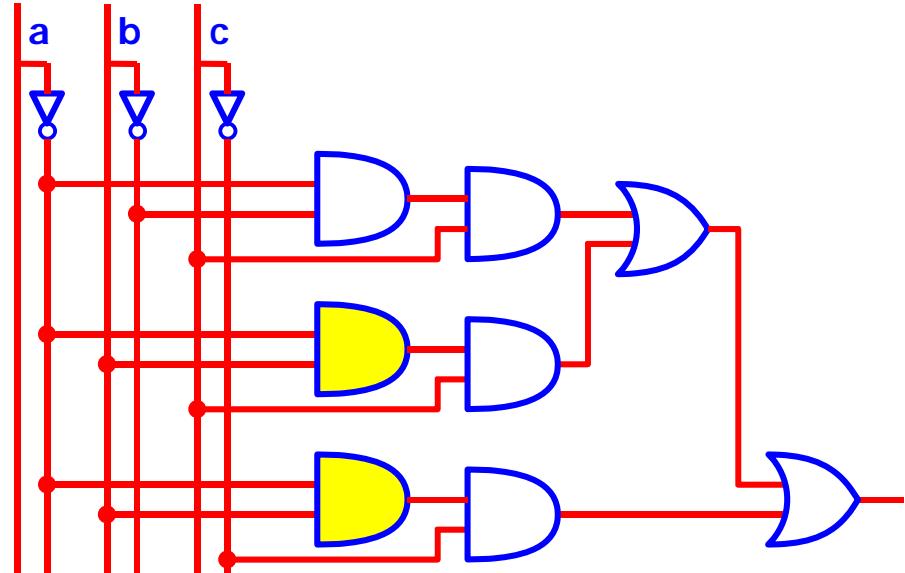
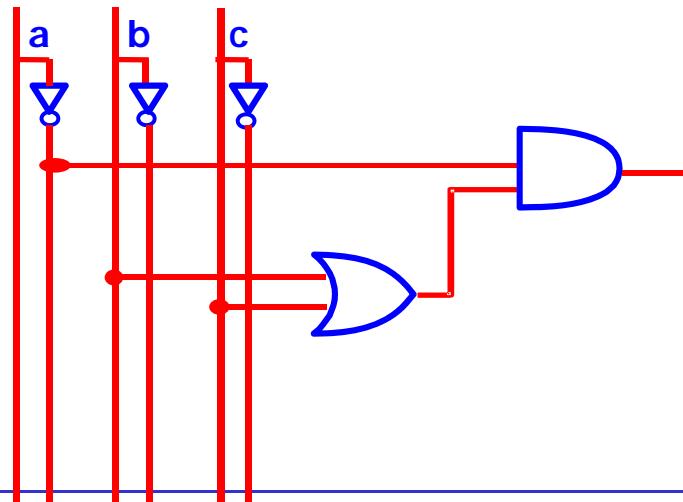
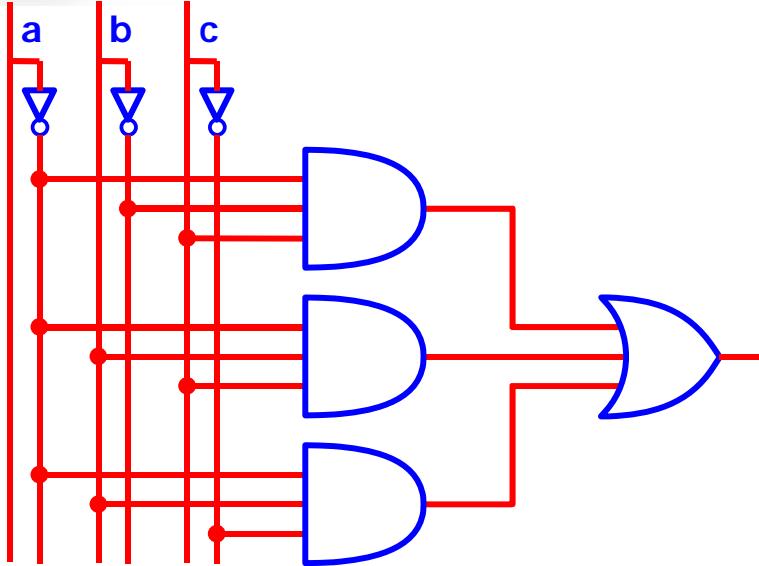


$$\text{Ritardo totale} = 5 \text{ ns} = 5 \cdot 10^{-9} \text{ sec}$$

$$\text{Freq. di commutazione} = 1 / 5 \text{ ns} = 200 \text{ MHz}$$



## Costo e ritardo di propagazione di reti equivalenti (porte a 2 ingressi)

$$F(a,b,c) = !a!bc + !abc + !ab!c = !a(c+b)$$


Porte a 2 ingressi

AND: Costo = 4, Rit. = 10

OR: Costo = 4, Rit. = 12

NOT: Costo = 1, Rit. = 2



## 4b - Sintesi di reti combinatorie

- La sintesi di una rete combinatoria espressa come tabella delle verità, consiste nel ricavare lo schema logico (il circuito digitale) che calcola la funzione combinatoria
- In generale, per una data **tabella delle verità** possono esistere **più reti combinatorie** (la soluzione al problema di sintesi non è dunque unica)
- Esistono svariate procedure di sintesi di reti combinatorie, che differiscono per:
  - Complessità della procedura di sintesi
  - Ottimalità della rete combinatoria risultante, per dimensioni e velocità



## 4b - Sintesi di reti combinatorie: forme canoniche

- Data una *funzione booleana*, la soluzione iniziale al problema di determinare una sua espressione consiste nel ricorso alle *forme canoniche*.
  
- Le forme canoniche sono, rispettivamente, la forma *somma di prodotti* (SoP) (*sintesi in 1<sup>a</sup> forma canonica*) e quella *prodotto di somme* (PoS) (*sintesi in 2<sup>a</sup> forma canonica*).
  
- Data una funzione booleana esistono *una ed una sola* forma canonica SoP ed una e una sola forma PoS che la rappresenta.



## Sintesi in 1<sup>a</sup> forma canonica (o sintesi come somma di prodotti)

---

Data una tabella delle verità, a  $n \geq 1$  ingressi, della funzione da sintetizzare, la **funzione F** che la realizza può essere specificata come

- la **somma logica (OR)** di tutti (e soli) i **termini prodotto (AND)** delle **variabili di ingresso corrispondenti agli 1** della funzione.
- Ogni termine prodotto (o **mintermine**) è costituito dal prodotto logico delle variabili di ingresso (letterale) prese in forma naturale se valgono 1, in forma complementata se valgono 0.



# 1<sup>a</sup> forma canonica

- Si consideri il seguente esempio:

a	b	f(a,b)
0	0	0
0	1	1
1	0	0
1	1	1

- È intuitivo osservare che la funzione possa essere ottenuta dal OR delle seguenti funzioni:

$$\begin{array}{|c|c|c|} \hline a & b & f(a,b) \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline a & b & f_1(a,b) \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline a & b & f_2(a,b) \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ \hline \end{array}$$



# 1<sup>a</sup> forma canonica

- Per cui, intuitivamente, si ottiene:

a	b	$f(a,b)$
0	0	0
0	1	1
1	0	0
1	1	1

=

a	b	$f_1(a,b)$
0	0	0
0	1	1
1	0	0
1	1	0

+

a	b	$f_2(a,b)$
0	0	0
0	1	0
1	0	0
1	1	1



$$f_1(a,b) = a'b \quad f_2(a,b) = ab$$

- Poiché, ad esempio, quando  $a=0$  e  $b=1$  il prodotto  $a'b$  assume valore 1 mentre vale 0 in tutti gli altri casi.



# 1<sup>a</sup> forma canonica

- Ne consegue:

a	b	$f(a,b)$
0	0	0
0	1	1
1	0	0
1	1	1

=

a	b	$f_1(a,b)$
0	0	0
0	1	1
1	0	0
1	1	0

+

a	b	$f_2(a,b)$
0	0	0
0	1	0
1	0	0
1	1	1

$$f(a,b) = a'b + ab$$

- Mettendo in OR i *mintermini* della funzione si ottiene l'*espressione booleana* della funzione stessa espressa come somma di prodotti.
  - nel *mintermine* (prodotto) una variabile compare nella forma naturale ( $x$ ) se nella corrispondente configurazione di ingresso ha valore 1, nella forma complementata ( $x'$ ) se ha valore 0



## Esempio: funzione maggioranza

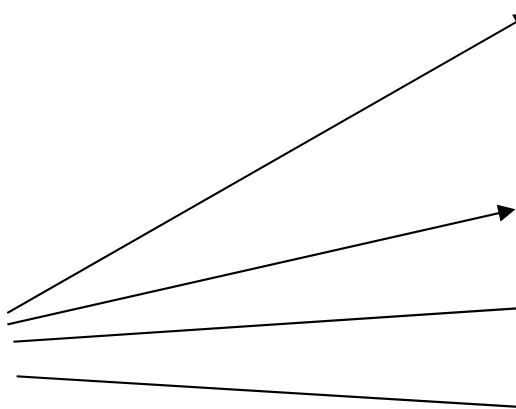
- Si chiede di sintetizzare (in 1<sup>a</sup> forma canonica) una funzione combinatoria dotata di 3 ingressi A, B e C, e di un'uscita F, funzionante come segue:
  - Se la maggioranza degli ingressi vale 0, l'uscita vale 0
  - Se la maggioranza degli ingressi vale 1, l'uscita vale 1



# Tabella delle verità

- L'uscita vale 1 se e solo se 2 o tutti e 3 gli ingressi valgono 1 (cioè se e solo se il valore 1 è in maggioranza)

mintermini



# riga	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



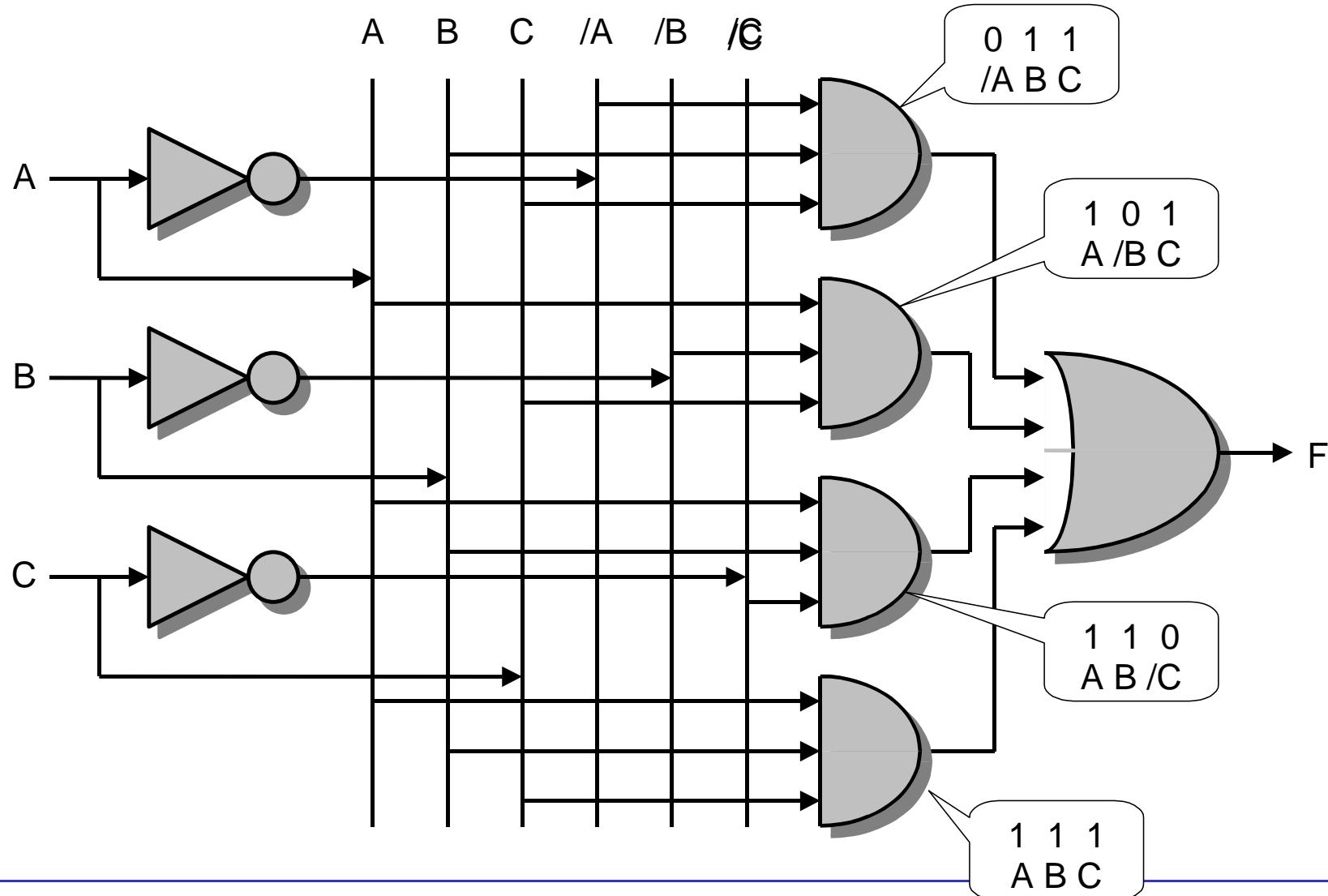
## Espressione booleana

---

- Dalla **tabella della verità** si ricava, tramite la sintesi in 1a forma canonica (somma di prodotti), l'**espressione booleana** che rappresenta la funzione
- $F(A, B, C) = \overline{A} \overline{B} C + A \overline{B} C + A \overline{B} \overline{C} + A B C$
- Dall'**espressione booleana** si ricava la **rete combinatoria** (il circuito) che è sempre **a 2 livelli**



# Rete combinatoria: schema logico





## Sintesi in 2<sup>a</sup> forma canonica (o sintesi come prodotto di somme)

---

Data una tabella delle verità, a  $n \geq 1$  ingressi, della funzione da sintetizzare, la **funzione F** che la realizza può essere specificata come

- la **prodotto logico (AND)** di tutti (e soli) i **termini somma (OR)** delle **variabili di ingresso corrispondenti agli 0** della funzione.
- Ogni **termine somma** (o **maxtermine**) è costituito dalla somma logica delle variabili di ingresso (letterale) prese in forma naturale se valgono 0, in forma complementata se valgono 1.



## 2<sup>a</sup> forma canonica

- Si consideri nuovamente lo stesso esempio:

a	b	f(a,b)
0	0	0
0	1	1
1	0	0
1	1	1

- È intuitivo osservare che la funzione possa essere ottenuta dall'AND delle seguenti funzioni:

a	b	f(a,b)
0	0	0
0	1	1
1	0	0
1	1	1

=

a	b	f <sub>1</sub> (a,b)
0	0	0
0	1	1
1	0	1
1	1	1

\*

a	b	f <sub>2</sub> (a,b)
0	0	1
0	1	1
1	0	0
1	1	1



## 2<sup>a</sup> forma canonica

- Per cui, intuitivamente, si ottiene:

a	b	$f(a,b)$
0	0	0
0	1	1
1	0	0
1	1	1

=

a	b	$f_1(a,b)$
0	0	0
0	1	1
1	0	1
1	1	1

\*

a	b	$f_2(a,b)$
0	0	1
0	1	1
1	0	0
1	1	1



$$f_1(a,b) = a+b$$

$$f_2(a,b) = a' + b$$

Infatti, ad es., quando  $a=0$  e  $b=0$  allora  $(a+b)$  assume valore 0 mentre vale 1 in tutti gli altri casi.



$$\textcolor{red}{f(a,b)} = \textcolor{red}{(a+b)} * \textcolor{red}{(a'+b)}$$

Mettendo in AND i **maxtermini** della funzione si ottiene l'*espressione booleana* della funzione stessa espressa come prodotto di somme.

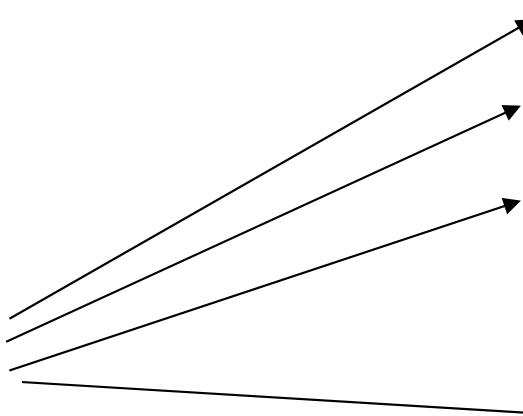
nel **maxtermine** (somma) una variabile compare nella forma naturale ( $x$ ) se nella corrispondente configurazione di ingresso ha valore 0, nella forma complementata ( $x'$ ) se ha valore 1



# Sintesi POS

□  $F = (A+B+C)(A+B+\neg C)(A+\neg B+C)(\neg A+B+C)$

maxtermini



# riga	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



---

# AXO - Architettura dei Calcolatori e Sistemi Operativi

## reti sequenziali



# Sommario

---

- Circuiti sequenziali e elementi di memoria
- Bistabile SR asincrono
- Temporizzazione e clock
- Bistabili D e SR sincroni
- Flip-flop



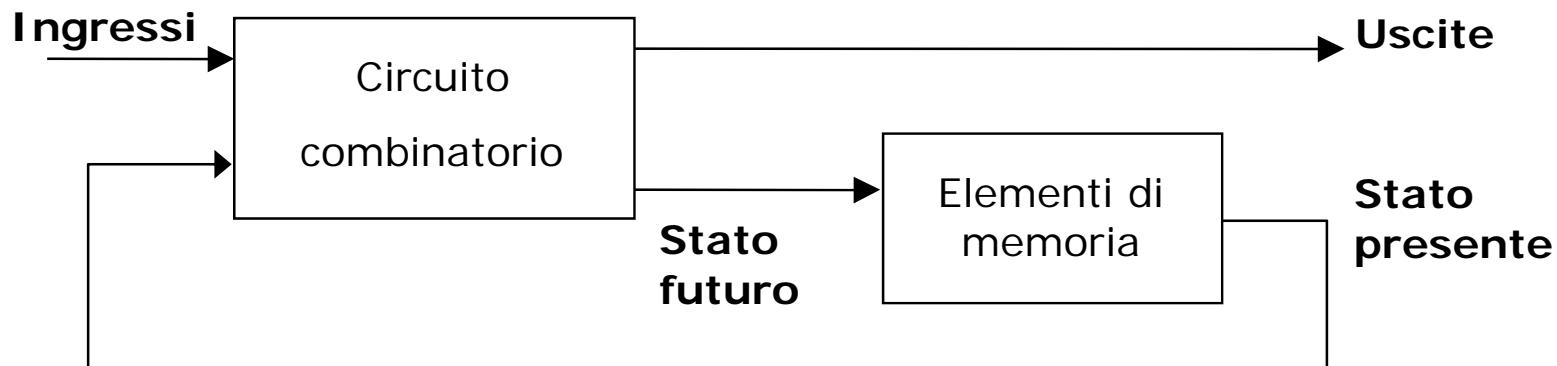
# Circuiti sequenziali

- Un circuito digitale è di tipo **sequenziale** se le sue uscite dipendono non solo dai valori correnti degli ingressi, ma anche da (alcuni di) quelli passati
  - Una stessa configurazione di ingresso applicata in due istanti di tempo successivi può produrre due valori di uscita differenti
- Un circuito digitale sequenziale (o rete sequenziale) è pertanto dotato, in ogni istante di tempo, di uno **stato** che, insieme ai valori degli **ingressi**, ne determina il **comportamento futuro**
  - Lo **stato** di un circuito sequenziale rappresenta una forma di **memoria** e contiene una sorta di descrizione della storia passata del circuito stesso
- L'elemento funzionale elementare per la realizzazione di circuiti sequenziali è il **bistabile (elemento di memoria)**, che è in grado di memorizzare un bit di informazione



# Circuiti sequenziali: struttura

- I circuiti sequenziali sono formati da:
  - **bistabili**, che hanno la funzione di memorizzare valori di singoli bit
  - **porte logiche**, organizzate in reti combinatorie, che hanno funzioni di elaborazione di informazioni
- Il circuito sequenziale ha, in ogni istante, uno **stato**: il **valore dei bit** memorizzati nei bistabili facenti parte del circuito





# Elementi di memoria

- Gli elementi di memoria fondamentali, o **bistabili**, sono caratterizzati da **due stati (0 e 1) stabili**
- Mantengono lo stato memorizzato finchè uno o più segnali di ingresso **forzano** il cambiamento di stato
- Vengono classificati in base a
  - numero di ingressi previsti per comandare il bistabile
  - modo in cui tali ingressi determinano il cambiamento di stato



# Bistabili: classificazione

- Esistono due famiglie di bistabili (circuiti digitali sequenziali):
  - **Asincroni**: sono privi di un segnale di sincronizzazione e modificano lo stato rispondendo direttamente a eventi sui segnali di ingresso
  - **Sincroni**: sono sensibili ad un segnale di controllo (o di sincronizzazione) e la transizione da uno stato all'altro può avvenire solo in corrispondenza di **eventi** del segnale di controllo.
    - Si può dire che il comportamento di un circuito sincrono viene osservato in **istanti discreti** di tempo
    - Il segnale di sincronizzazione tipicamente utilizzato è il clock
    - Ulteriore classificazione dei bistabili sincroni:
      - **bistabili sincroni controllati** (gated latch)
      - **flip flop**
        - » **flip flop master slave** (a livello o pulse triggered)
        - » **flip flop edge-triggered** (a fronte)



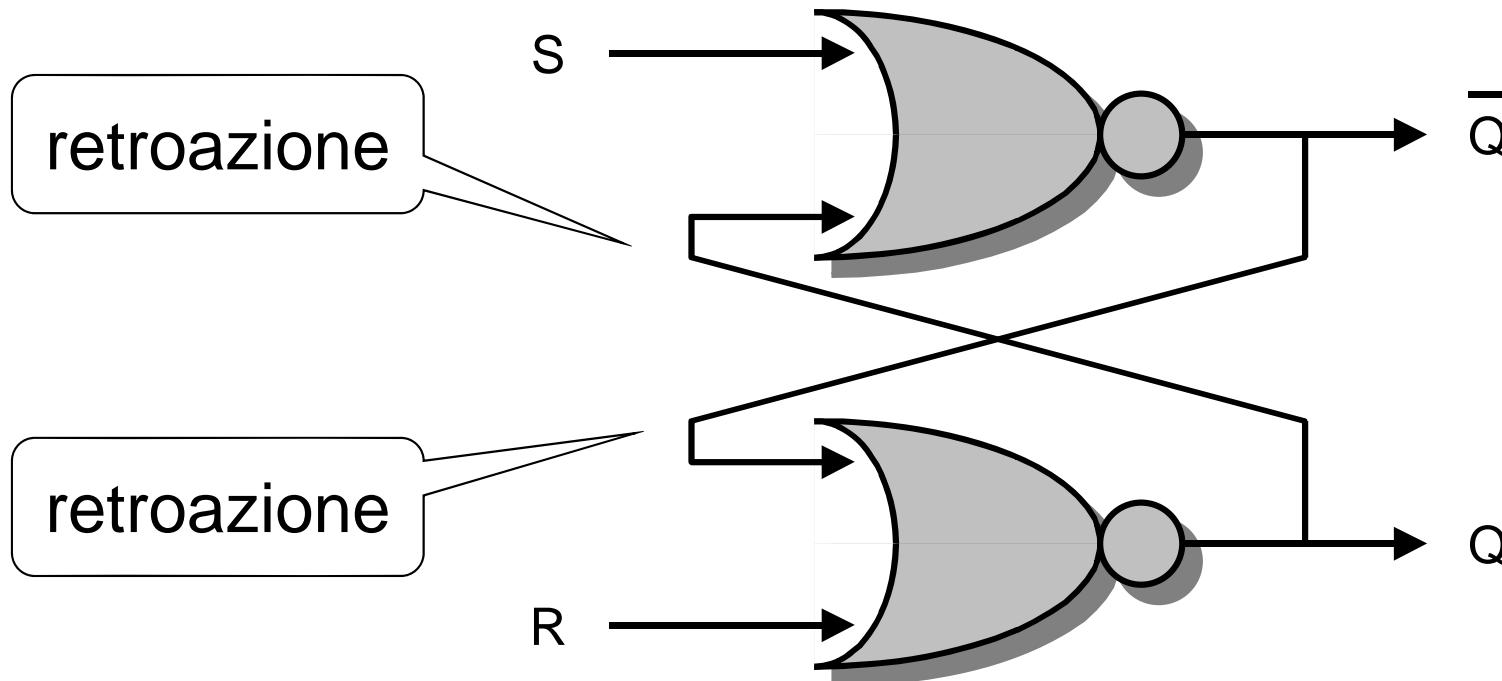
# Bistabile SR asincrono

- Il bistabile SR è dotato di 2 ingressi S (Set) e R (Reset) e di 2 uscite Q e  $\bar{Q}$ .
  - Se  $Q = 1$  ( $\bar{Q} = 0$ ) : stato di set
  - Se  $Q = 0$  ( $\bar{Q} = 1$ ) : stato di reset
- L'uscita **Q** rappresenta quindi lo **stato memorizzato**
  - se  $S = R = 0$ , le uscite Q e  $\bar{Q}$  possono valere 1 e 0, rispettivamente, ma ...
  - se  $S = R = 0$ , le uscite Q e  $\bar{Q}$  possono anche valere 0 e 1, rispettivamente
- Dunque, a parità di ingressi (cioè  $S = R = 0$ ) l'uscita Q ammette due possibili valori.



# Come memorizzare un bit

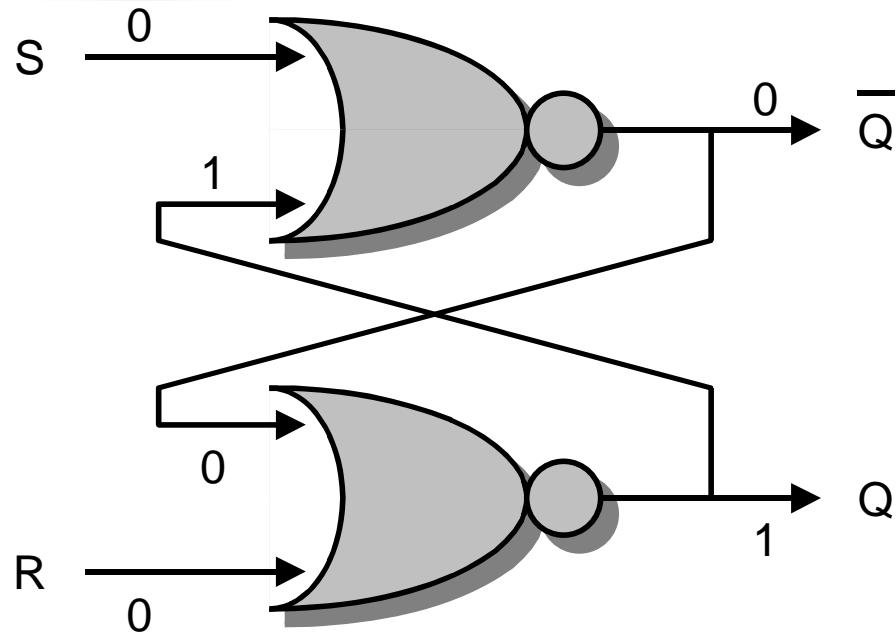
## Bistabile SR asincrono



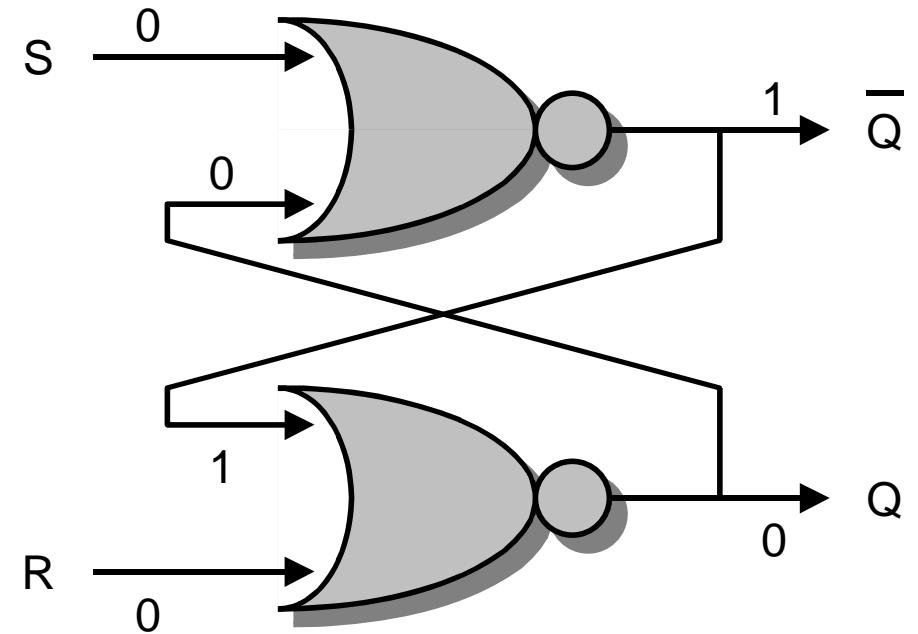
- Il circuito ha due ingressi, S e R, e due uscite: Q e  $\bar{Q}$  (la versione negata di Q)



# Come funziona il bistabile SR



$S = R = 0$  e  $Q = 1$   
memorizza il valore 1



$S = R = 0$  e  $Q = 0$   
memorizza il valore 0

Il circuito ha due stati di equilibrio (bistabile)



# Transizioni di stato

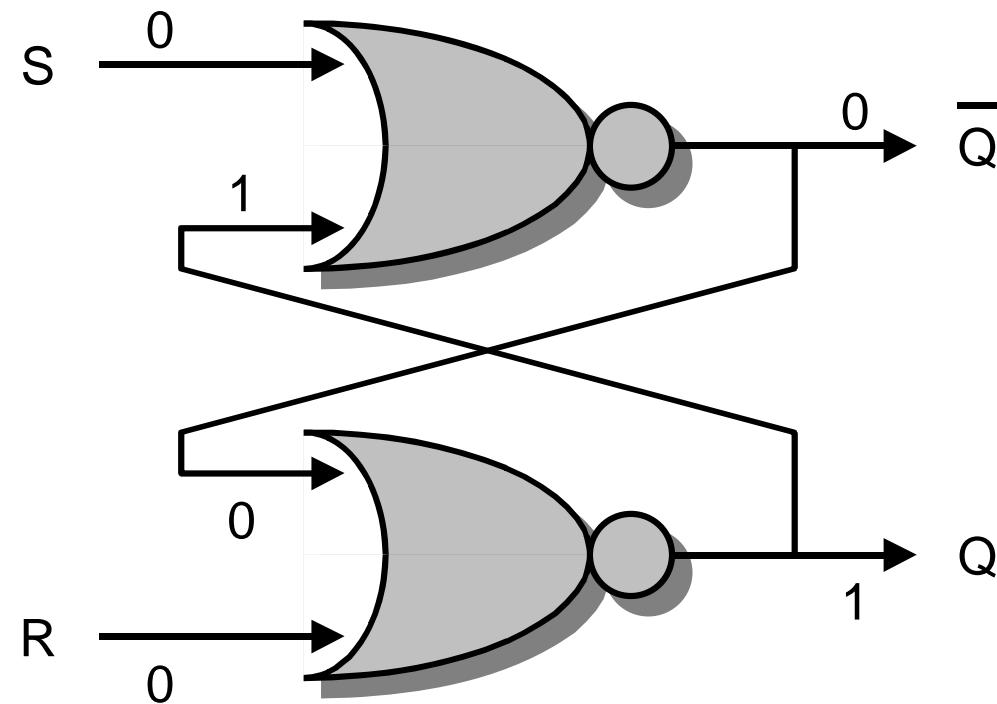
- Il bistabile SR è in grado di memorizzare due distinti valori logici:
  - se  $Q = 1$  il bistabile memorizza 1
  - se  $Q = 0$  il bistabile memorizza 0
- Stato a 0
  - se  $S=0$  e  $R=1$ , qualunque sia il valore dello stato presente,  $Q$  viene portata a 0, e  $\bar{Q}$  a 1. Il nuovo stato è 0
- Stato a 1
  - se  $S=1$  e  $R=0$ , qualunque sia il valore dello stato presente,  $Q$  viene portata a 1, e  $\bar{Q}$  a 0. Il nuovo stato è 1



# Transizione da 1 a 0 - Q=1 (1)

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



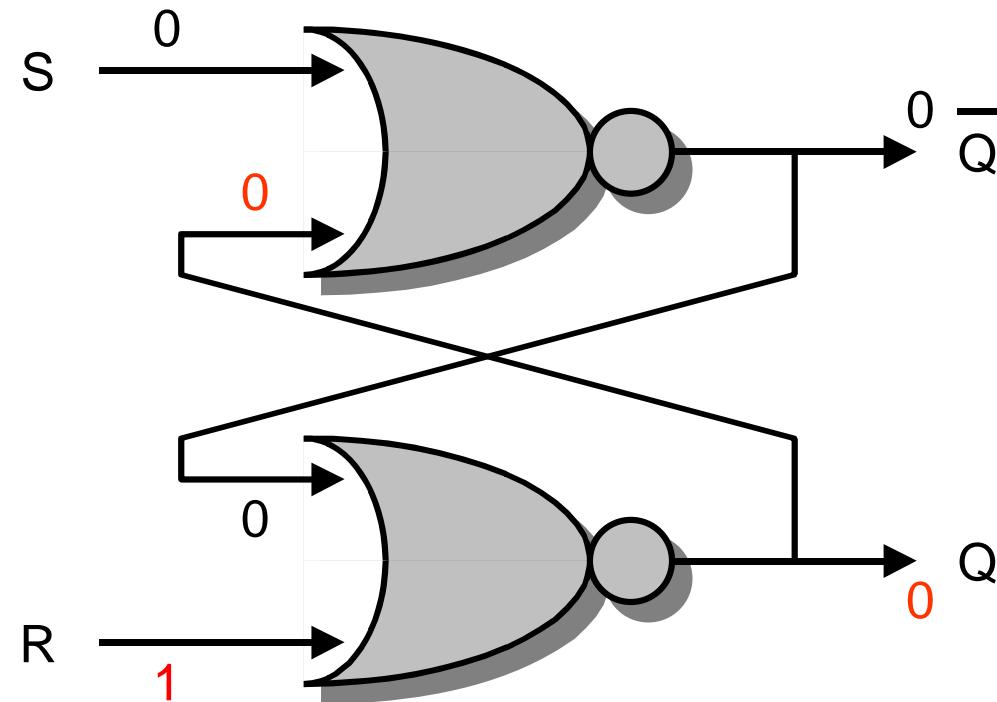
$$S = R = 0 \text{ e } Q = 1$$



# Transizione da 1 a 0 - Q=1 (2)

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

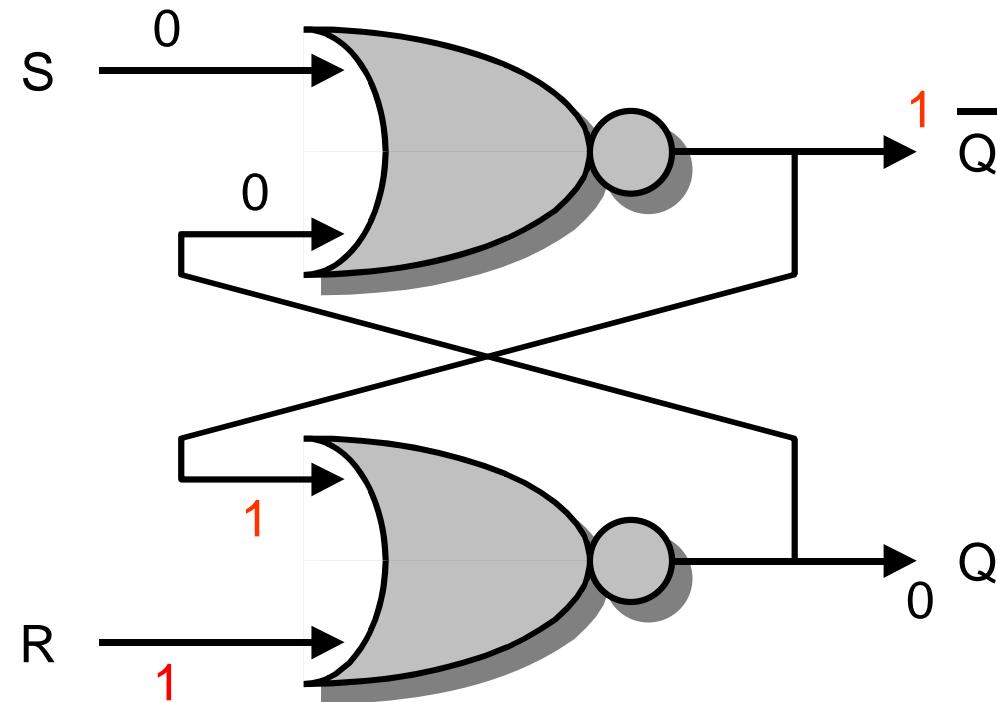




## Transizione da 1 a 0 - Q=1 (3)

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



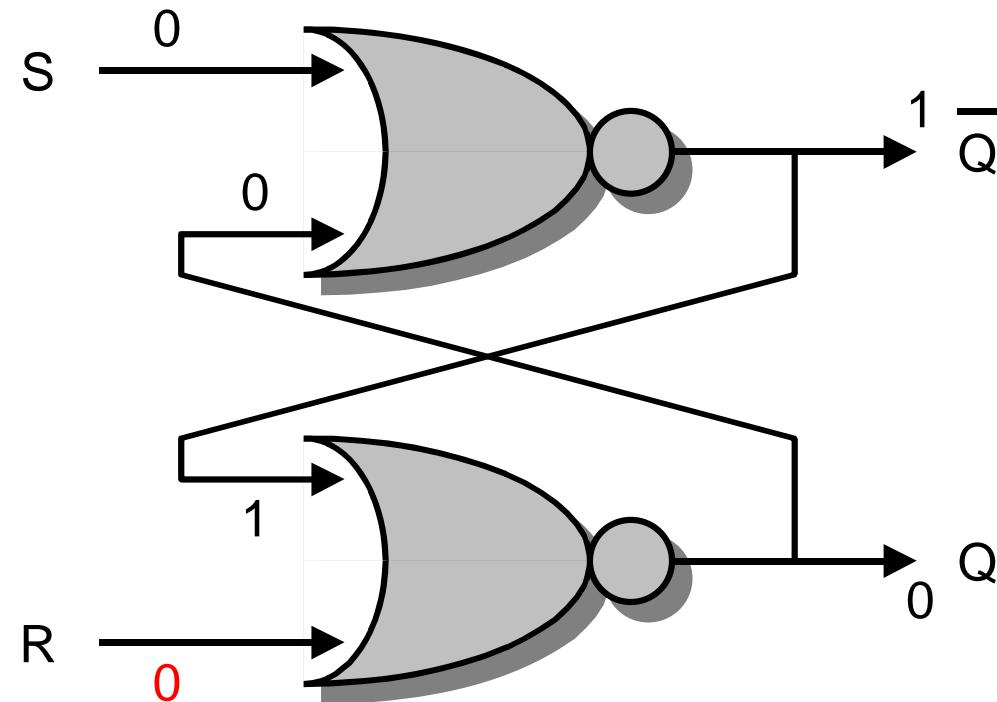
$S = 0$  e  $R = 1$ , allora  $Q$  diventa 0



# Transizione da 1 a 0 - Q=1 (4)

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



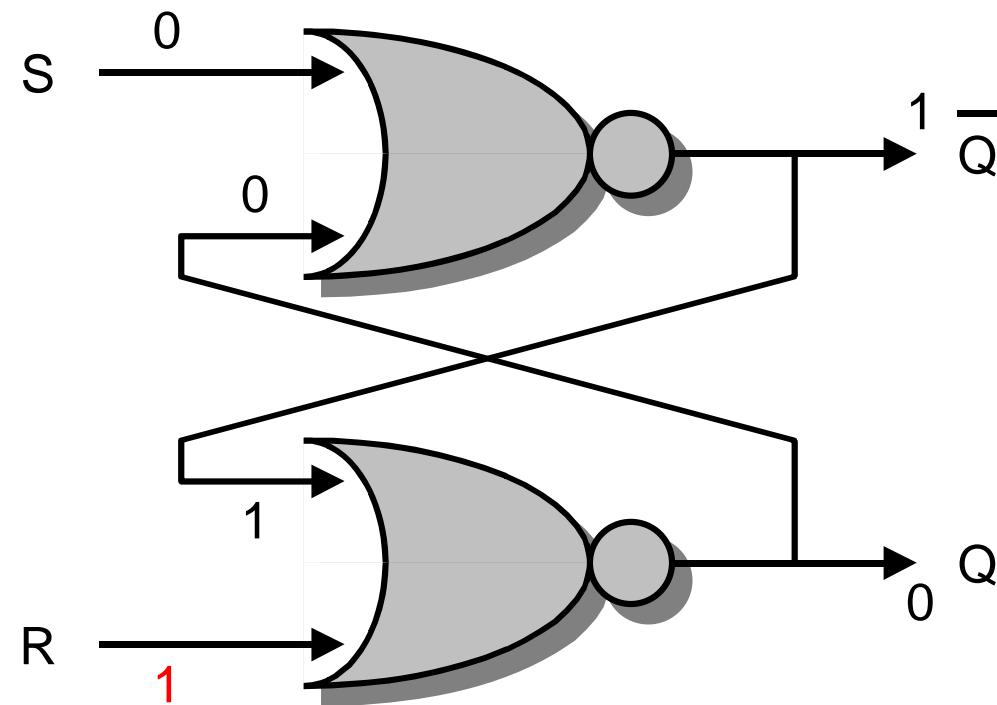
$S = R = 0$ , allora Q rimane 0



# Ingresso di Reset con Q=0

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



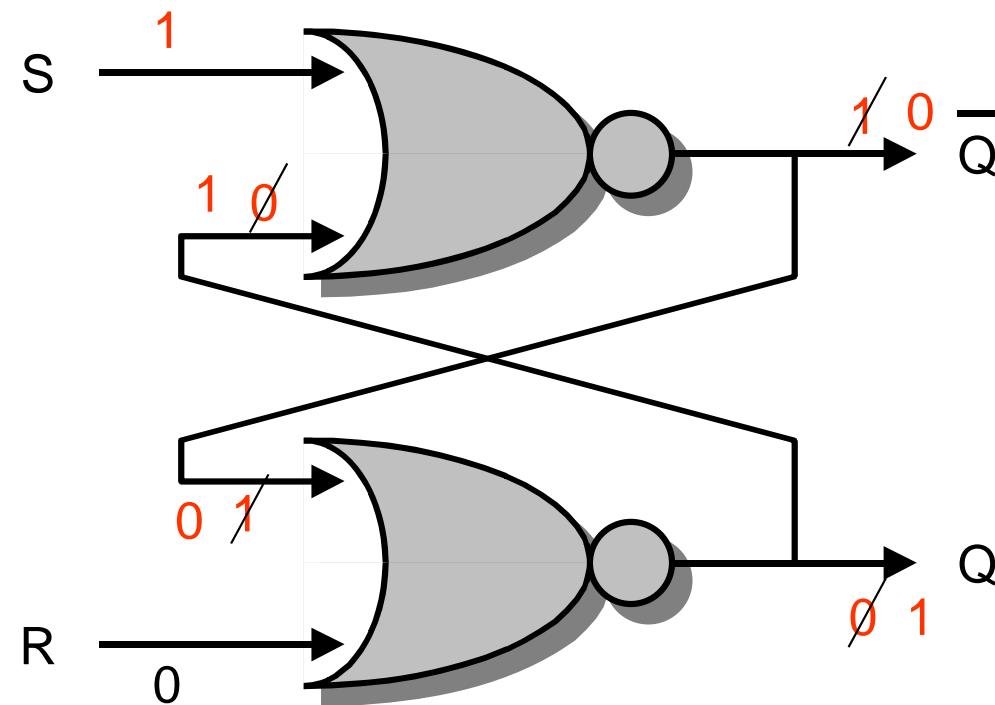
$S = 0$  e  $R = 1$ , Q rimane 0



# Transizione da 0 a 1 - Q=0

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



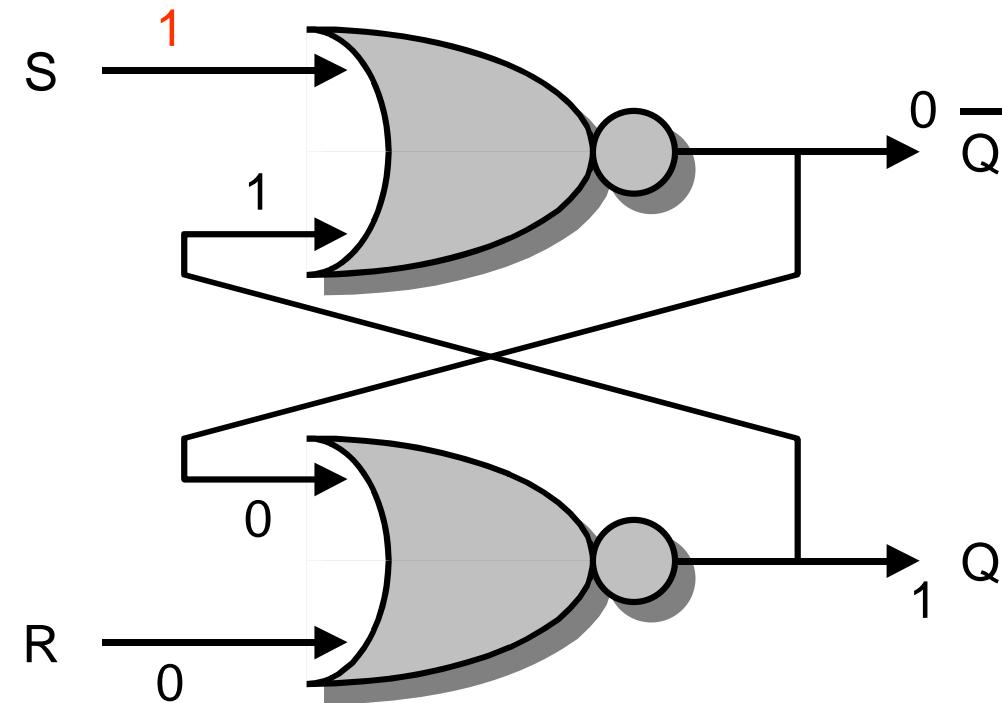
$S = 1$  e  $R = 0$ , allora  $Q$  diventa 1



# Ingresso di Set con $Q=1$

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



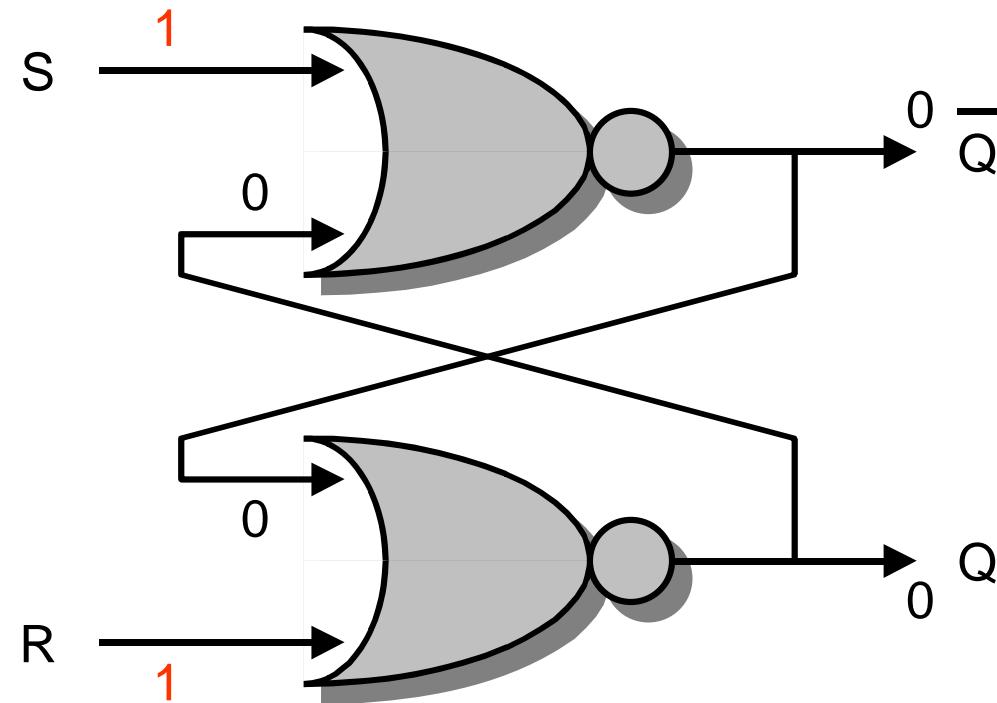
$S = 1$  e  $R = 0$ , allora  $Q$  rimane 1



# Anomalia di funzionamento S=R=1

NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0



$S = 1$  e  $R = 1$ , allora idealmente  $Q$  e  $\bar{Q}$  a 0



## Riassumendo

### □ Funzionamento del bistabile SR:

- se  $S = R = 0$ , l'uscita Q mantiene memorizzato il valore logico di un bit (0 oppure 1)
- se  $S = 1$  e  $R = 0$ , l'uscita Q assume il valore logico 1
- se  $S = 0$  e  $R = 1$ , l'uscita Q assume il valore logico 0
- è vietato applicare la configurazione di ingresso  $S = R = 1$  (in questa circostanza il comportamento del bistabile SR non è definito)

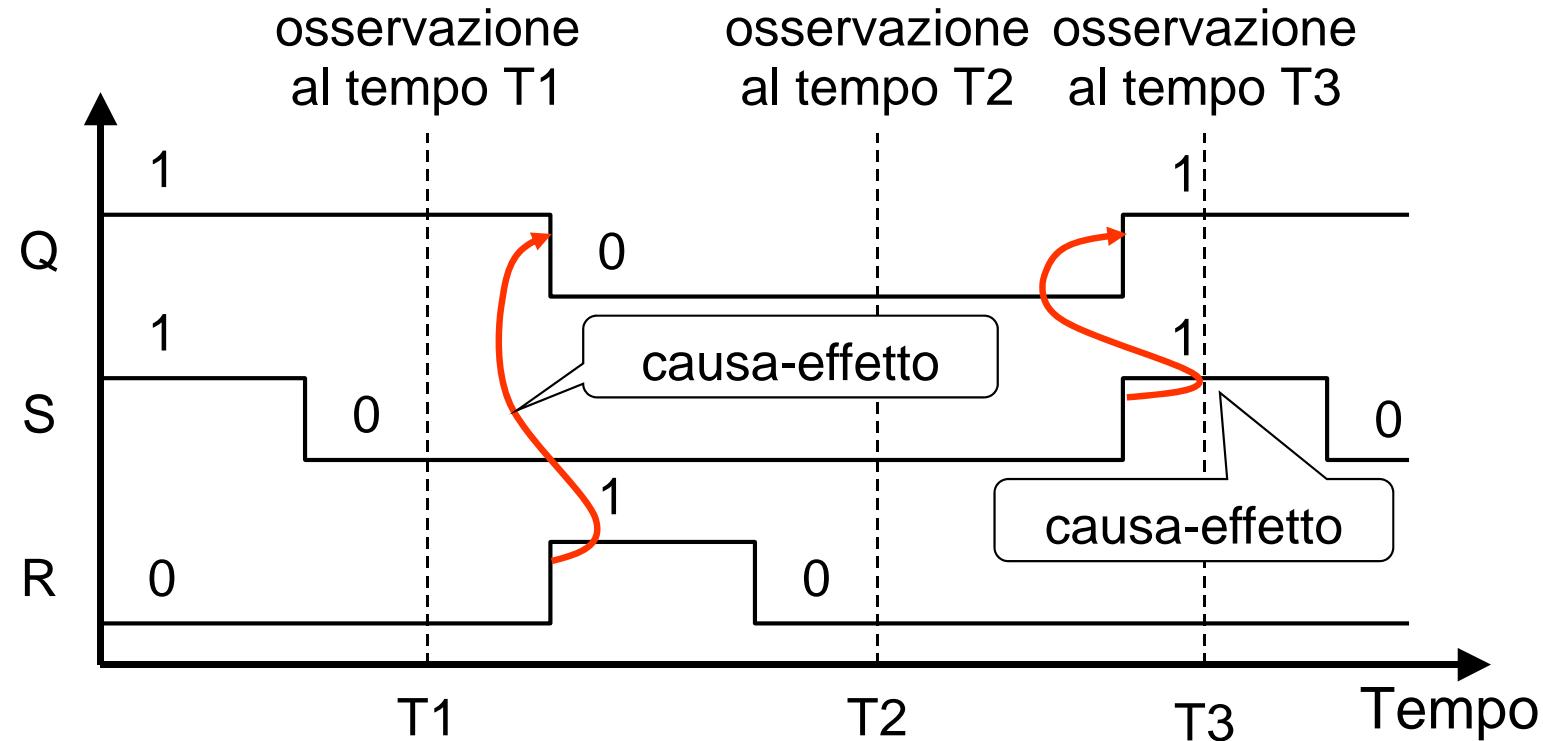


# Il diagramma temporale

- Un buon modo per visualizzare comportamenti di circuiti digitali che dipendono dal tempo e da eventi passati (circuiti sequenziali) è il **diagramma temporale**
- **Diagramma temporale**: sistema di assi cartesiani, con
  - in ascissa il tempo (in istanti discreti)
  - in ordinata i vari segnali i cui valori logici si succedono al trascorrere del tempo



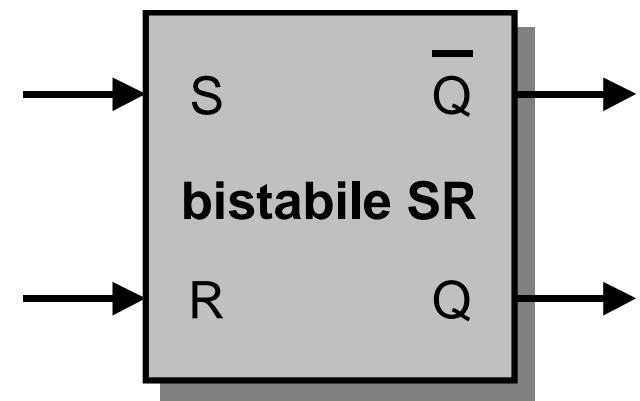
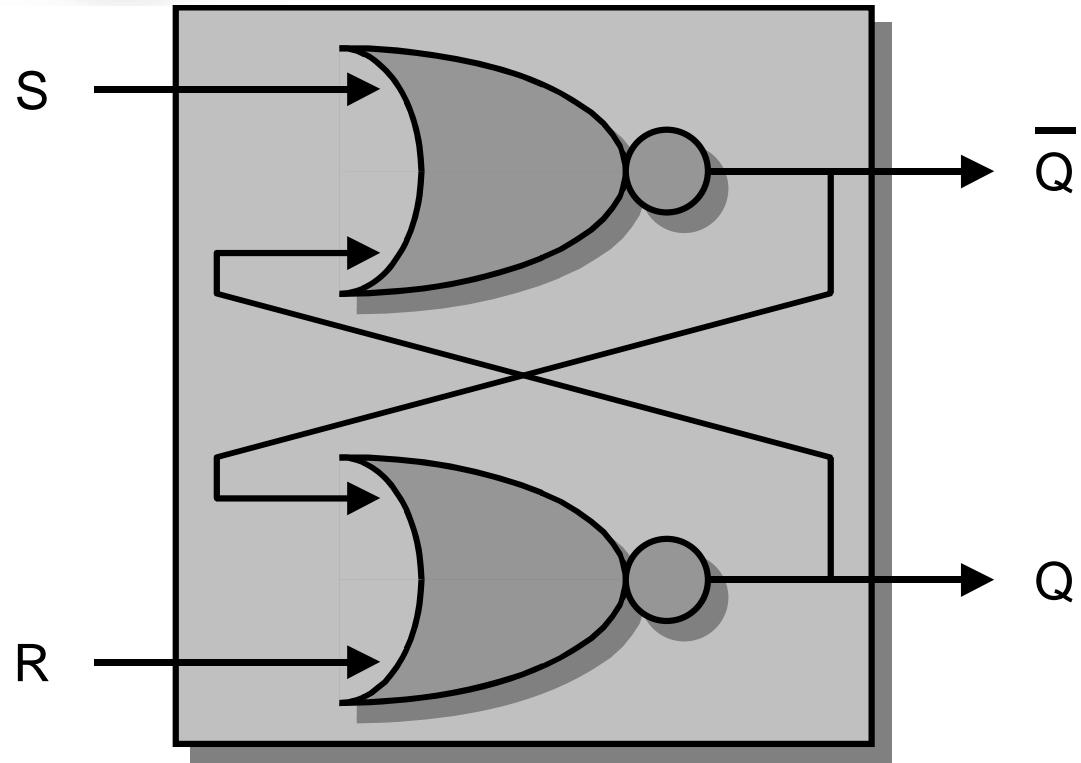
# Diagramma temporale del bistabile SR asincrono



Le frecce indicano un rapporto tra i fronti di tipo causa-effetto



# Rappresentazione



Il bistabile SR (set-reset) come blocco funzionale SEQUENZIALE

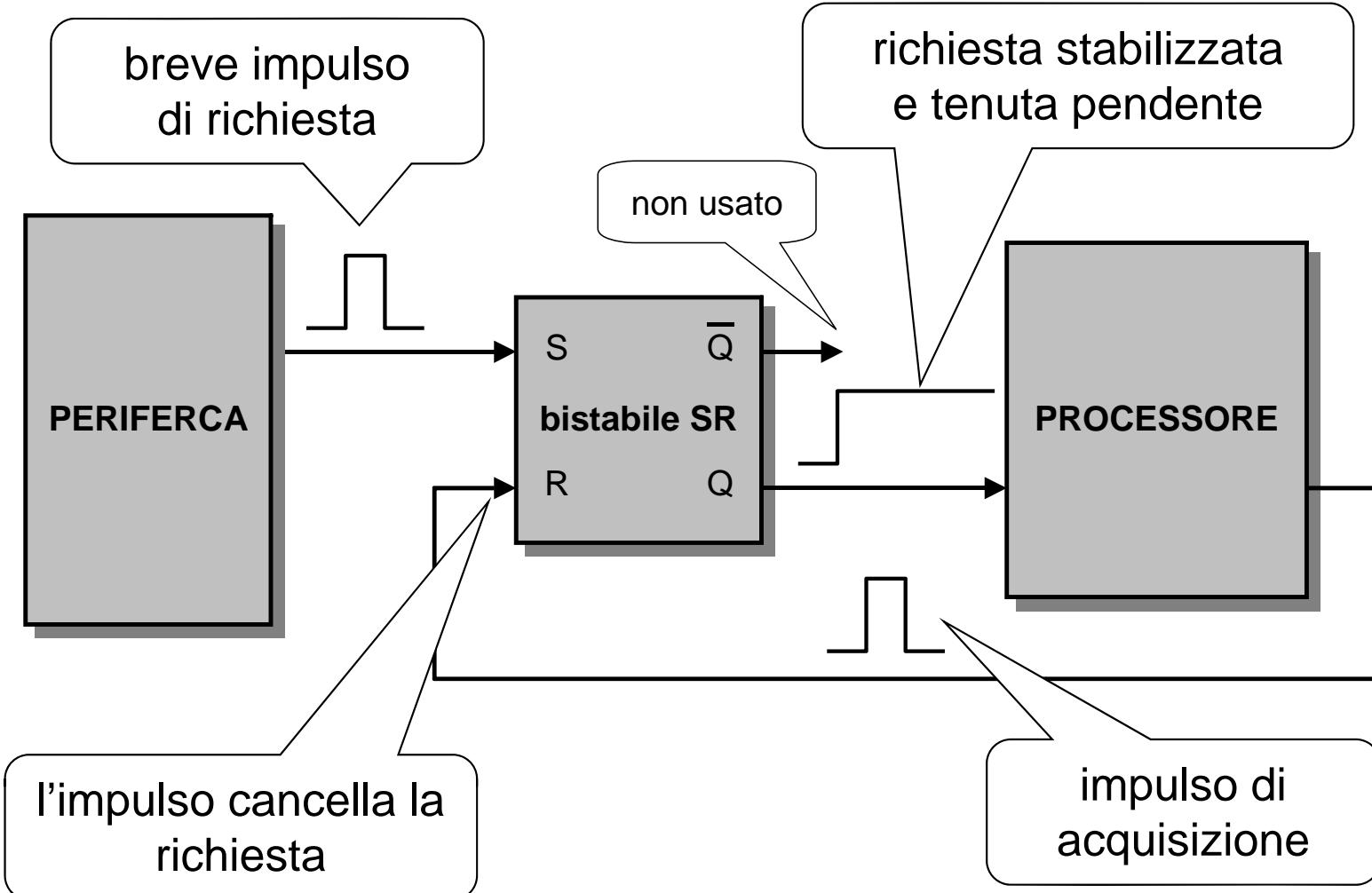


## Esempio di uso come adattatore

- Si supponga di avere una periferica che deve mandare un segnale di richiesta (p. es. di interruzione, interrupt) a un processore
- La periferica genera solo un breve impulso di richiesta e il processore potrebbe essere occupato e non in grado di rispondere subito alla richiesta, onorandola
- È dunque necessario interporre tra periferica e processore un circuito digitale adattatore (interfaccia), che:
  - riceva l'impulso di richiesta proveniente dalla periferica, lo memorizzi, stabilizzandolo, e lo mandi al processore
  - mantenga pendente la richiesta fintantoché il processore non sia disponibile a onorarla
  - cancelli la richiesta, quando il processore segnalasse di averla acquisita e di essere pronto a onorarla



# Schema logico



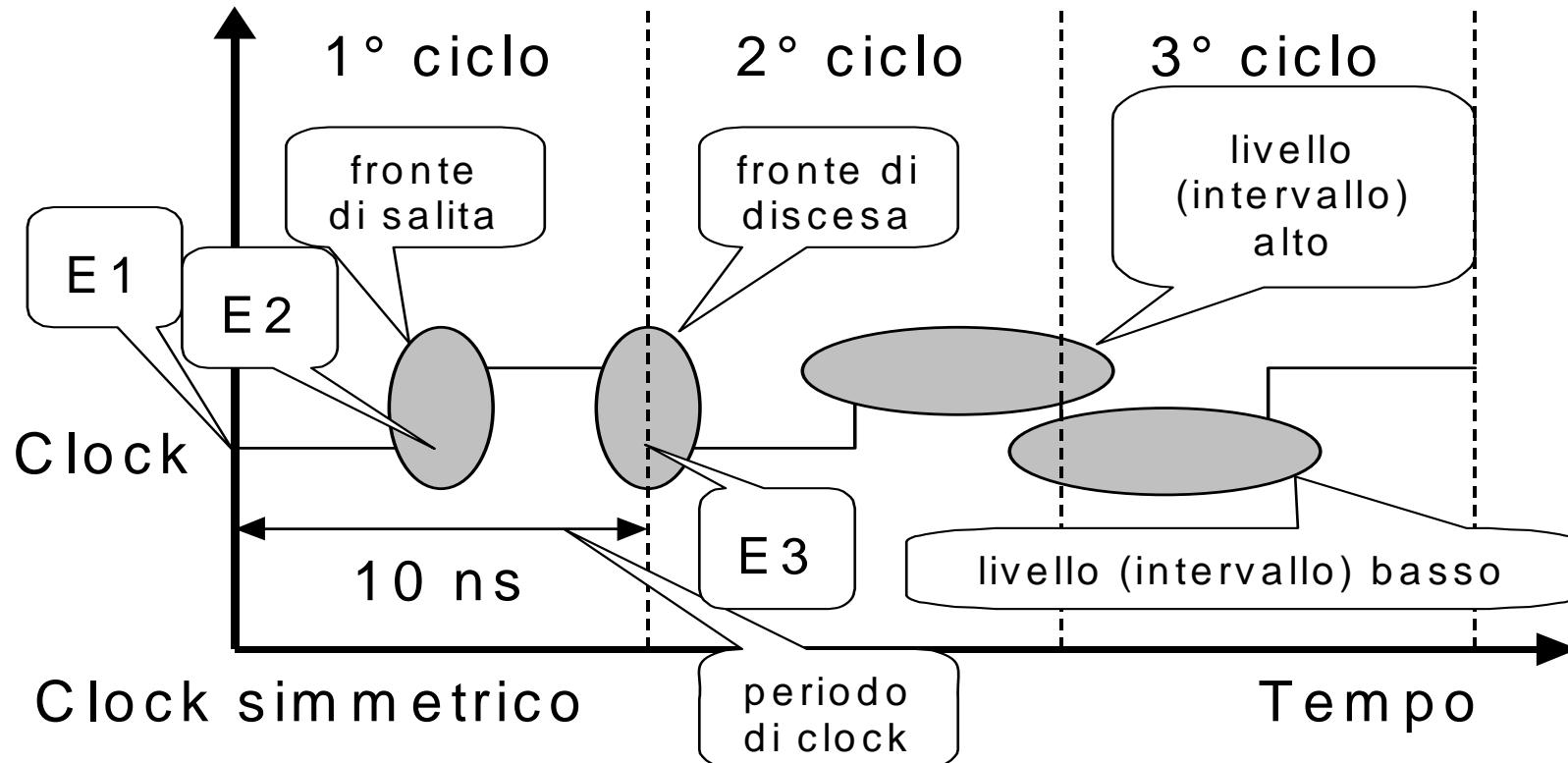


# Segnale di sincronizzazione

- In molte situazioni, è necessario che lo stato di un bistabile possa cambiare solo **in determinati istanti** di tempo o intervalli di tempo. Per ottenere questo occorre:
  - disporre di un **segnale di clock** (o di temporizzazione) che scandisca gli istanti o intervalli di tempo in cui le transizioni di stato possono avvenire
  - sincronizzare il bistabile con il clock
- Il **segnale di clock** è un segnale **binario**, con andamento **periodico nel tempo**
- Il segnale di clock è una **successione di impulsi**:
  - ogni impulso ha una larghezza costante e due impulsi consecutivi stanno a una distanza costante



# Segnale di clock



$$\text{Frequenza di clock} = 1 / \text{periodo di clock} = 1 / 10 \text{ ns} = 100 \text{ MHz}$$

Il ciclo di clock contiene 3 eventi (E1, E2, E3)



# Bistabili sincroni e temporizzazione

- I fattori che differenziano i bistabili riguardano due aspetti:
  - La relazione **ingresso-stato** (quando gli ingressi sono efficaci)
  - La relazione **stato-uscita** (quando vengono modificate le uscite)
- La relazione **ingresso-stato** (*tipo di temporizzazione*) definisce quando gli ingressi modificano lo stato interno del bistabile
  - 1. Temporizzazione basata sul **livello** del segnale di sincronizzazione
    - 1. Durante tutto l'intervallo di tempo in cui il segnale di controllo è attivo, qualsiasi variazione sui segnali di ingresso influenza il valore dello stato interno del bistabile. (bistabili con commutazione a livello)
  - Temporizzazione basata sul **fronte** del segnale di controllo
    - 1. Il valore dello stato interno del bistabile viene aggiornato solamente in corrispondenza di un fronte del segnale di controllo (bistabili con commutazione sul fronte - di salita oppure di discesa).



# Bistabili sincroni e temporizzazione

- La relazione *stato-uscita* definisce quando lo stato aggiorna le uscite.
  - Comutazione basata sul **livello** del segnale di controllo
    1. Durante tutto l'intervallo di tempo in cui il segnale di controllo è attivo un cambiamento dei segnali di ingresso modifica oltre allo stato interno anche le uscite.
    2. Bistabili con questa relazione stato-uscita sono denominati **LATCH**
      - Il segnale di controllo è solitamente chiamato *enable*.
      - Le uscite cambiano quando cambiano gli ingressi
  - 1. Comutazione basata sul **fronte** del segnale di controllo
    - Le uscite vengono aggiornate su di un fronte del segnale di sincronismo.
    - Bistabili con questa relazione stato-uscita sono denominati **FLIP-FLOP**
    - Le uscite cambiano in corrispondenza di un evento del clock



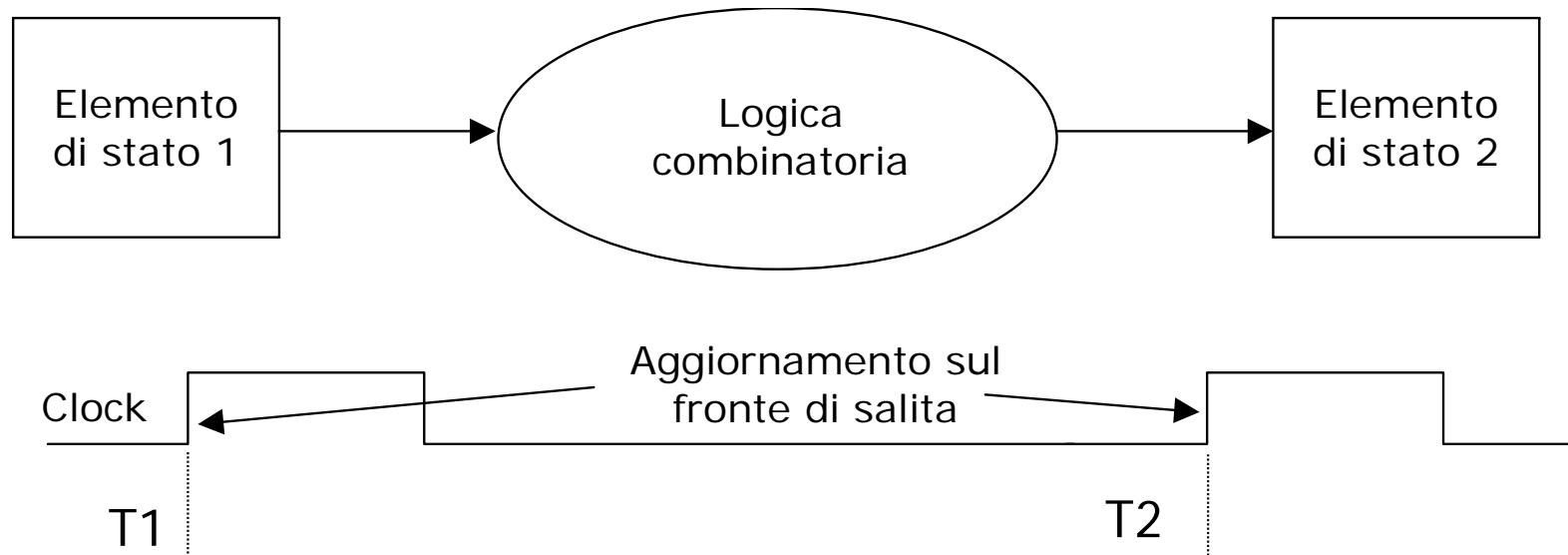
# Bistabili sincroni e temporizzazione

- Tabella riassuntiva

Relazione <i>Stato-Uscita</i>	
Relazione Ingresso-Stato	Livello
	Fronte
Fronte	Flip-Flop edge-triggered
Livello	Latch con Enable
Livello	Flip-Flop Master-Slave



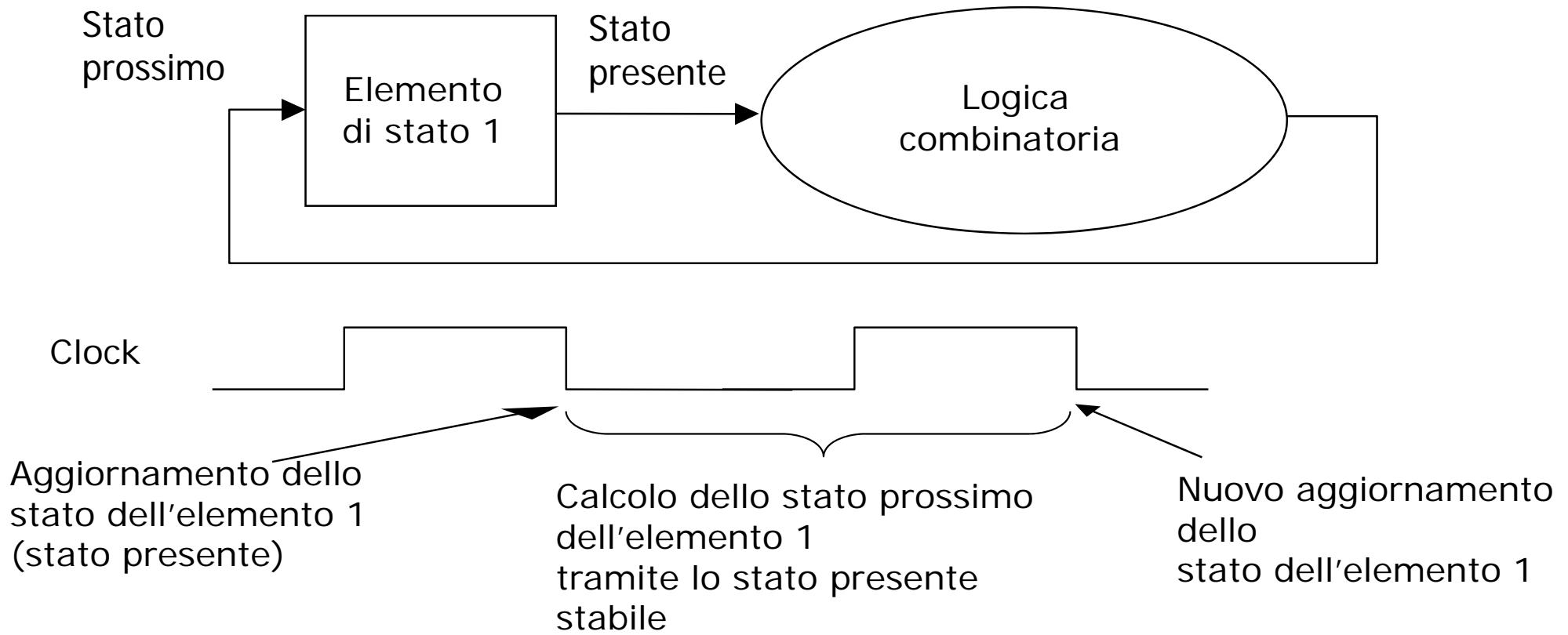
## Considerazioni sulla commutazione delle uscite sul fronte (1)



Il valore dello stato memorizzato nell'elemento 1 al tempo T1 viene utilizzato (con il valore degli ingressi primari) per determinare tramite la rete combinatoria il valore di stato che verrà memorizzato nell'elemento 2 al tempo T2



## Considerazioni sulla commutazione delle uscite sul fronte (2)



Lettura e scrittura dello stato in un ciclo di clock

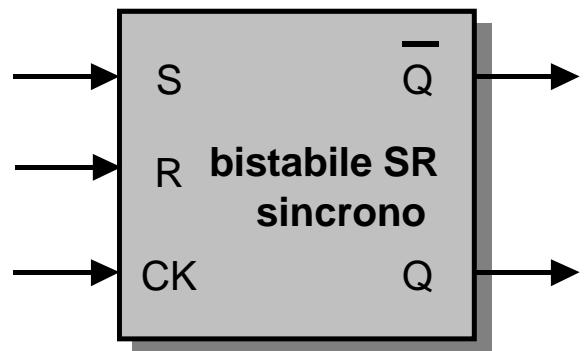
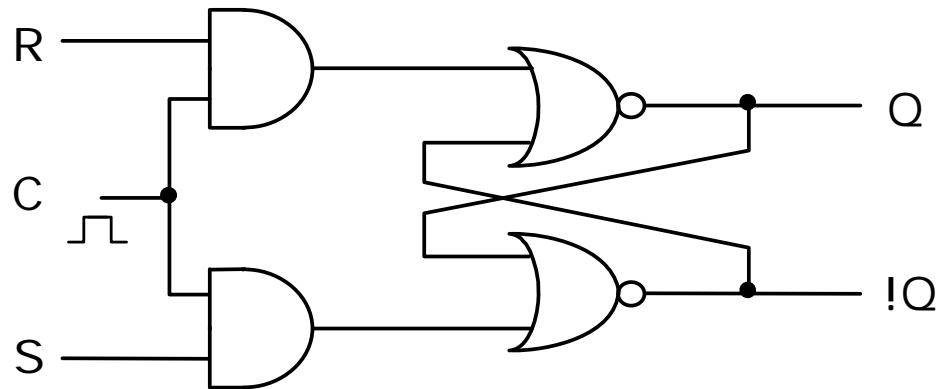


# Bistabile SR sincronizzato (SR-latch)

- Il bistabile SR sincronizzato ha
  - 2 ingressi S e R (che costituiscono i segnali di Set e Reset)
  - 1 ingresso di sincronizzazione (clock)
  - un'uscita Q, il cui valore rappresenta lo stato del bistabile, e un'uscita !Q
- Nel bistabile SR sincronizzato:
  - Se il clock vale 0, gli ingressi S e R non hanno alcun effetto (latch SR opaco) , e il bistabile mantiene memorizzato il suo stato corrente
  - Se il clock vale 1, **gli ingressi S e R sono efficaci** (latch SR trasparente) , e il comportamento è lo stesso descritto per SR asincrono



# Bistabile SR sincronizzato (SR-latch)



SR con controllo C

C	S	R	Q	!Q	
0	X	X	Q	!Q	Ingressi inibiti
1	0	0	Q	!Q	
1	0	1	0	1	Stato di Reset
1	1	0	1	0	Stato di Set
1	1	1	-	-	Stato indefinito

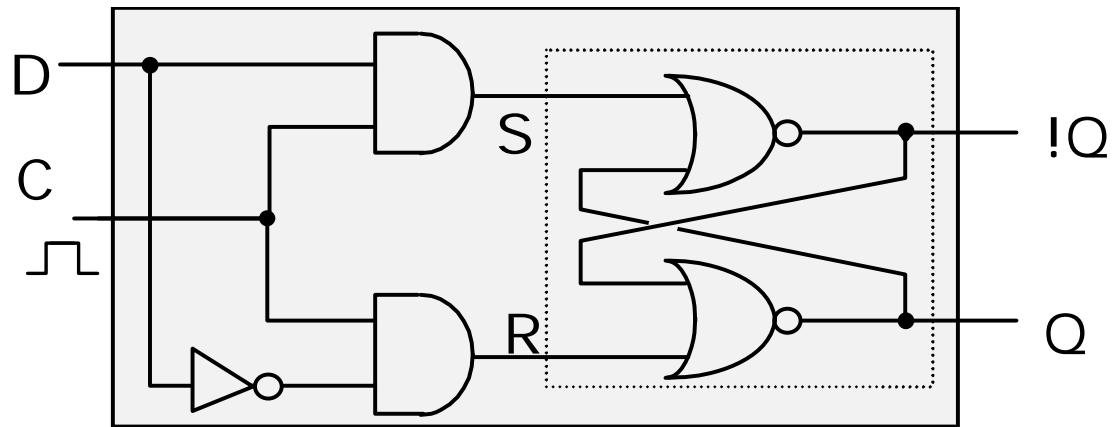


# Bistabile D sincronizzato (D-latch)

- Il bistabile D ha
  - 1 ingresso D (che rappresenta il dato che verrà memorizzato)
  - 1 ingresso di sincronizzazione (clock)
  - un'uscita Q, il cui valore rappresenta lo stato del bistabile, e un'uscita  $\bar{Q}$
- Nel bistabile D sincronizzato:
  - Se il clock vale 0, l'ingresso D non ha alcun effetto (latch D opaco), e il bistabile mantiene memorizzato il suo stato corrente
  - Se il clock vale 1, l'ingresso D è efficace (latch D trasparente), e il bistabile memorizza il valore logico (0 oppure 1) presente sull'ingresso D

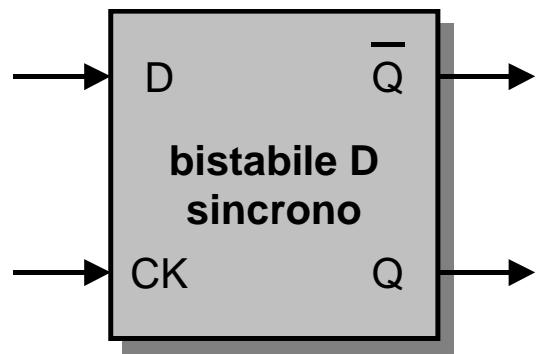


# Bistabile D sincronizzato (D-latch)



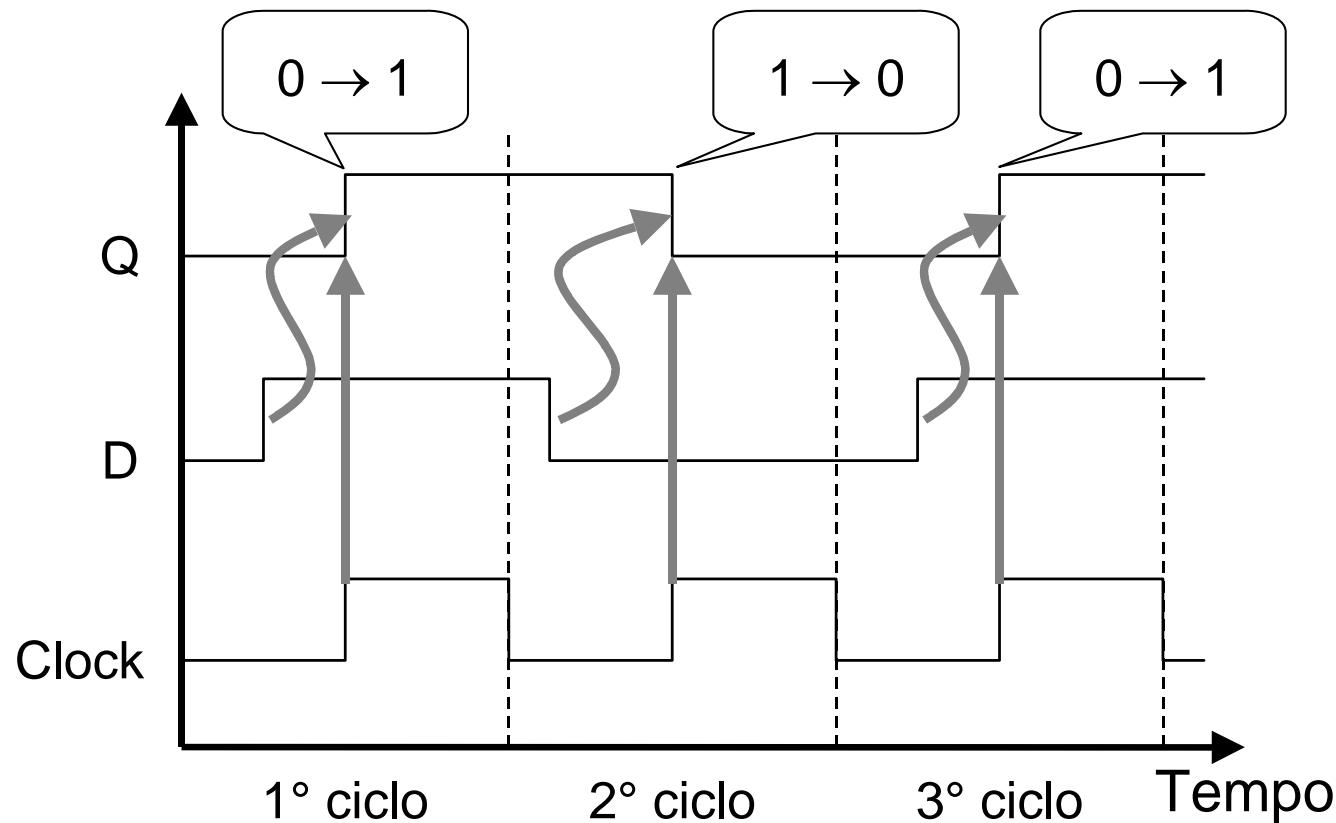
D con controllo C

C	D	Q	!Q
0	X	Q	!Q
1	0	0	1
1	1	1	0





# Diagramma temporale

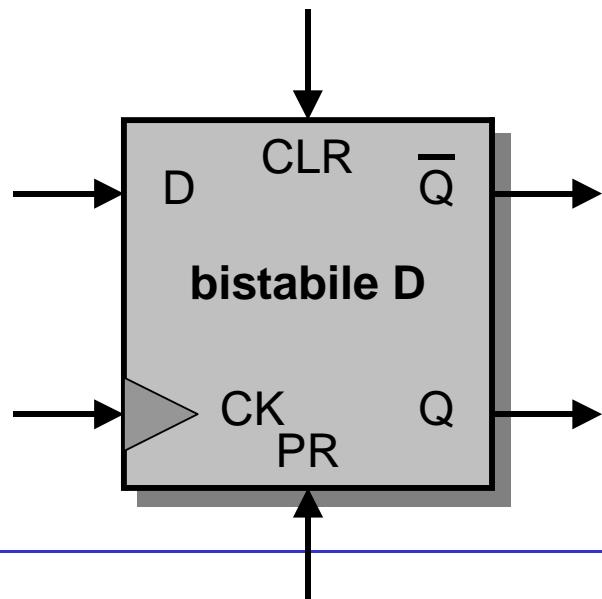


Le variazioni dell'uscita sono sincronizzate con il clock



# Comando di ripristino

- Tutti i tipi di bistabili dispongono di varianti dotate di un comando di ripristino CLR (**clear o reset**), che forza lo stato del bistabile a 0
- Il comando di ripristino è molto utile per (re)inizializzare lo stato dei bistabili
- Alcuni bistabili dispongono anche del comando di precarica PR (**preset**), che forza lo stato del bistabile a 1



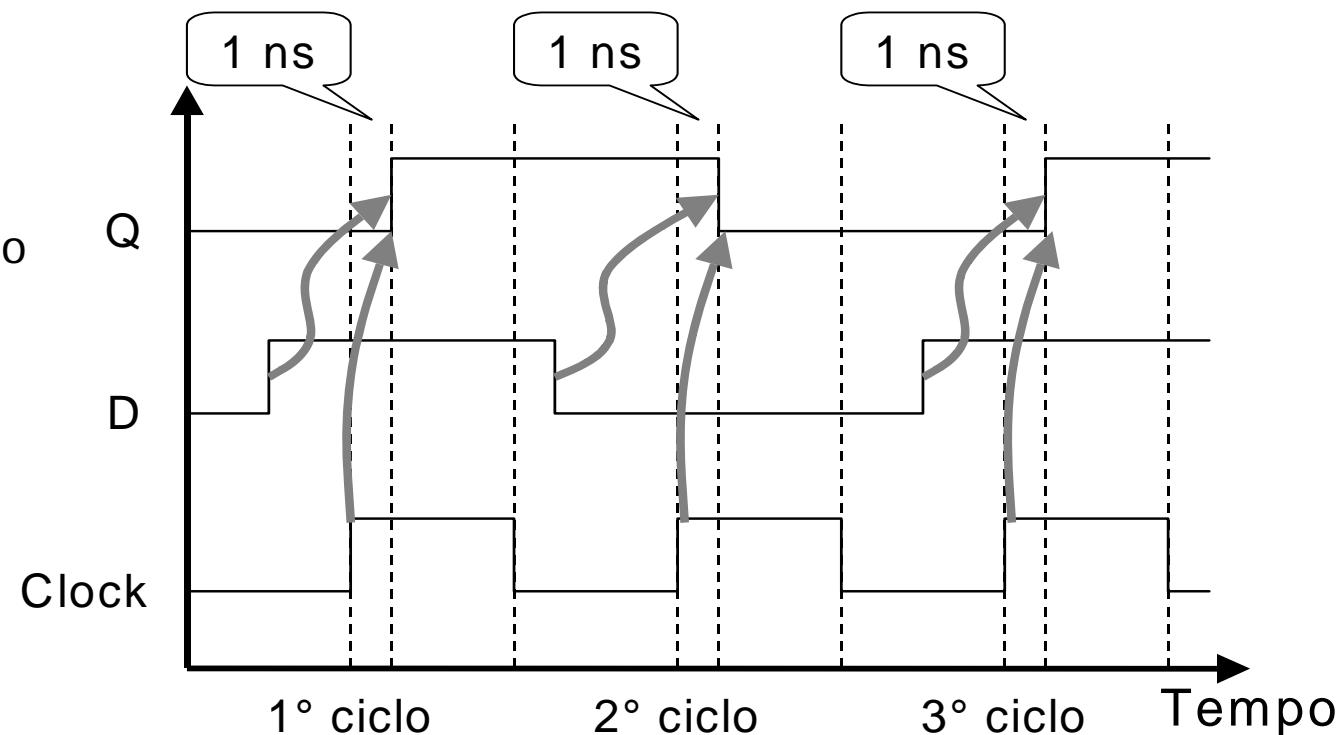
Bistabile di tipo D, dotato di comandi di ripristino e di precarica. Di norma questi comandi sono **asincroni**, cioè agiscono immediatamente, non appena vengono attivati, senza attendere il clock



# Ritardo di commutazione

- I bistabili (sincronizzati o no), come le porte logiche, presentano un **ritardo di commutazione dell'uscita**
  - La commutazione dell'uscita avviene con un certo ritardo rispetto alla variazione degli ingressi o rispetto al fronte di clock che hanno indotto la transizione di stato. Il ritardo di commutazione dipende dalla tecnologia

Il bistabile sincrono di tipo D ha un ritardo di commutazione di 1 ns dell'uscita rispetto al fronte del clock



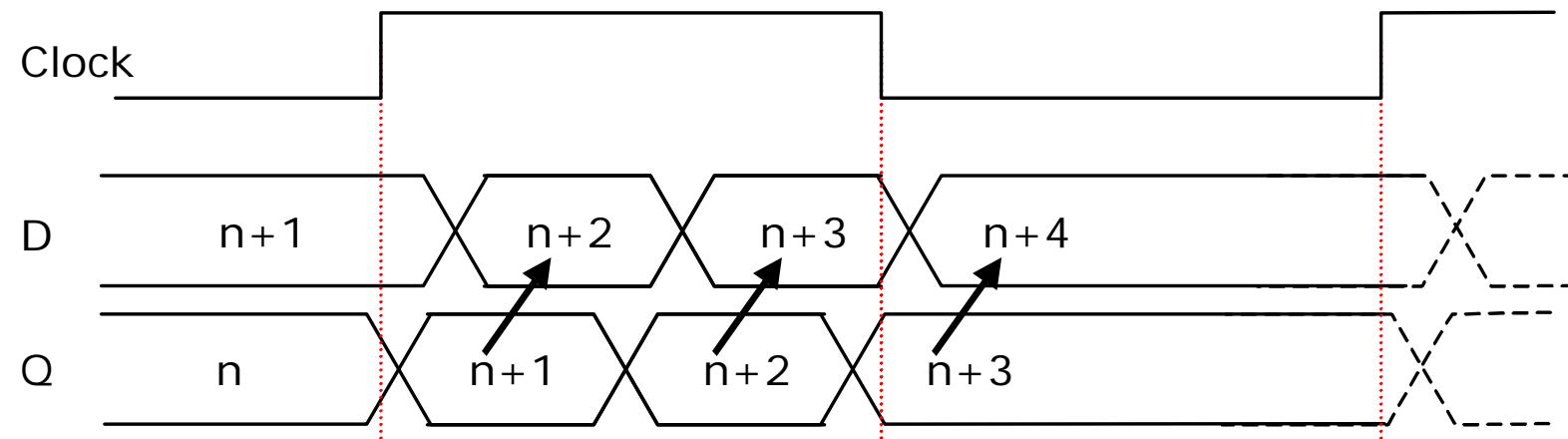
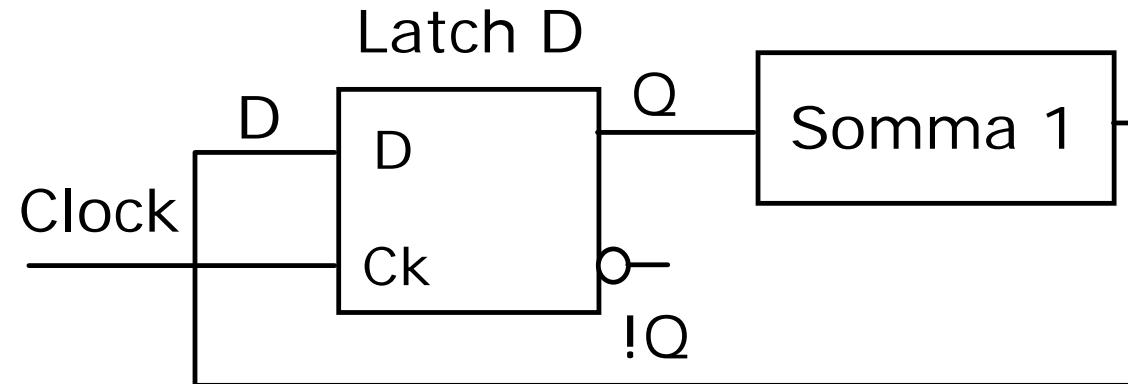


# Trasparenza

- I latch sincroni (SR o D) presentano, durante l'intervallo di tempo in cui il clock è attivo, il fenomeno di **trasparenza delle uscite** (fenomeno indesiderato).
  - In questo intervallo, se gli ingressi si modificano, le uscite seguono questa modifica
  - E' come se, nell'intervallo attivo del clock, i bistabili non esercitassero alcuna funzione effettiva di memorizzazione
- Per evitare il fenomeno di trasparenza si utilizzano i **flip-flop** (D o SR) che sono costituiti da due latch in cascata in modo che lo stato possa **modificare le uscite** solo in corrispondenza di un **evento (fronte) del segnale di controllo**.
- Nei **flip-flop**:
  - Relazione *stato-uscita* (aggiornamento della uscita):
    - sul fronte.
  - Relazione *ingresso-stato* (aggiornamento dello stato):
    - **a livello (Flip-Flop master-slave)**
    - a fronte (Flip-Flop *edge-triggered*).



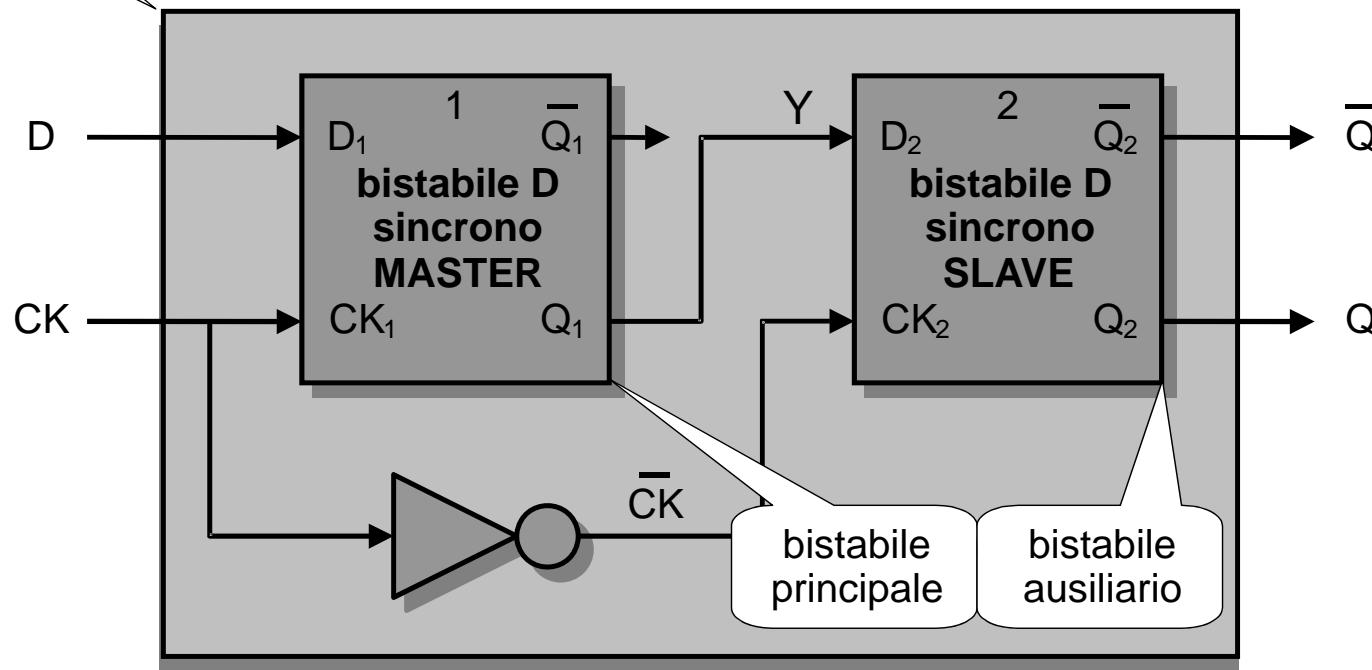
# Esempio di trasparenza





# Flip-flop D master-slave

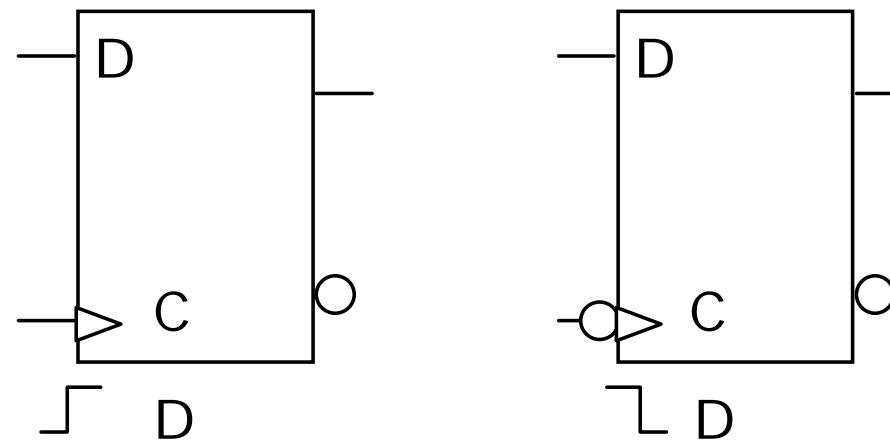
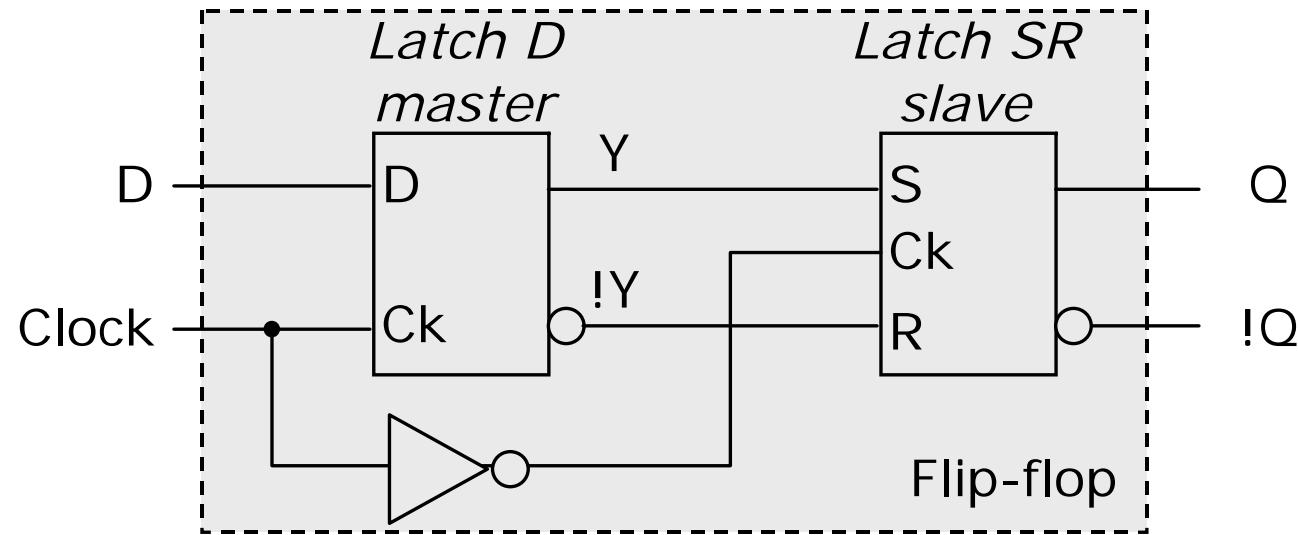
flip-flop D  
master-slave



Coppia di bistabili sincroni D trasparenti in cascata con clock invertiti;  
l'insieme dei due non presenta il fenomeno della trasparenza



# Flip-flop D master-slave



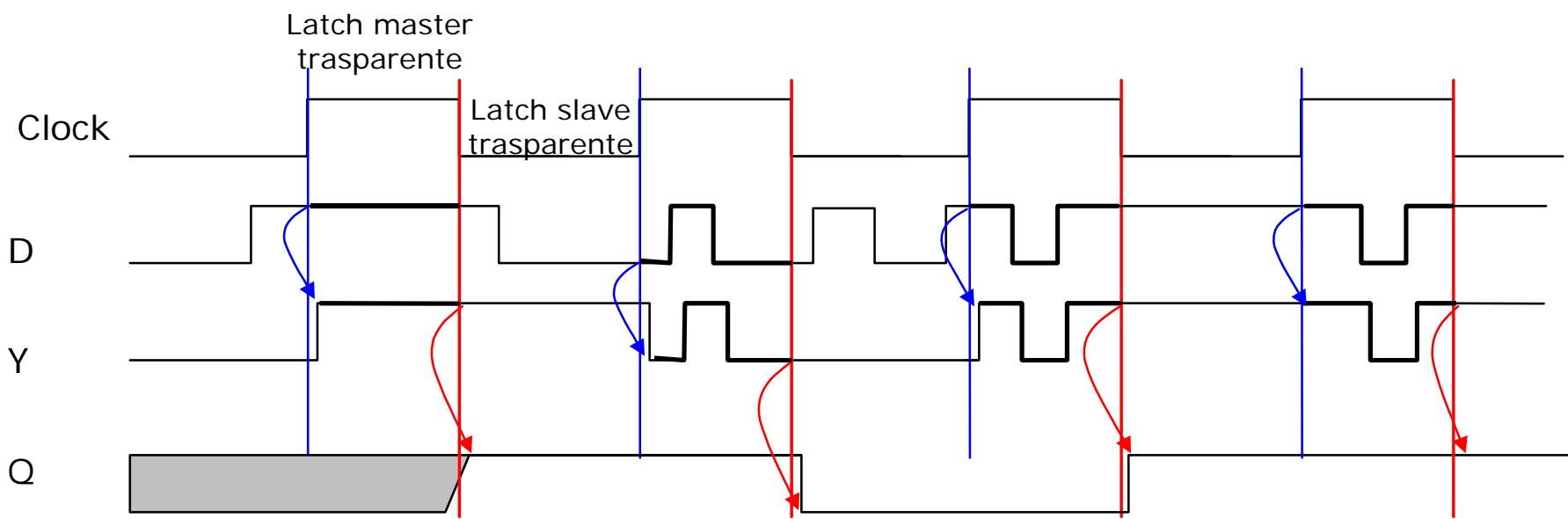


# Funzionamento

- Il bistabile principale campiona l'ingresso  $D = D_1$  durante l'intervallo alto del clock, lo emette sull'uscita  $Q_1$  e lo manda all'ingresso  $D_2$  del bistabile ausiliario
- Il bistabile ausiliario campiona l'ingresso  $D_2$  durante l'intervallo basso del clock e lo emette sull'uscita  $Q_2 = Q$
- L'uscita generale  $Q$  può variare solo nell'istante del **fronte di discesa del clock**
- **Trasparenza**
  - Nell'intervallo basso del clock, il bistabile SLAVE è in stato di trasparenza
  - Nell'intervallo alto del clock, il bistabile MASTER è in stato di trasparenza
  - Se l'ingresso D varia durante l'intervallo alto del clock, il bistabile MASTER si comporta in modo trasparente
  - Ma il bistabile SLAVE no, perché il suo clock si trova nell'intervallo basso

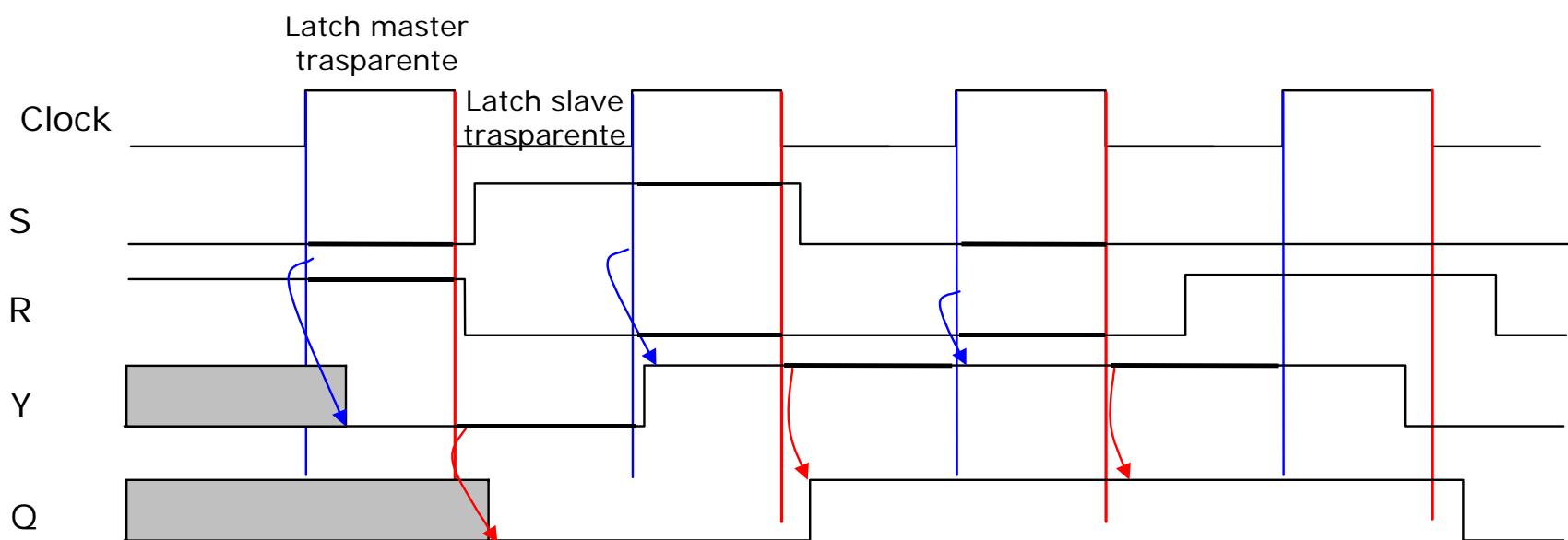
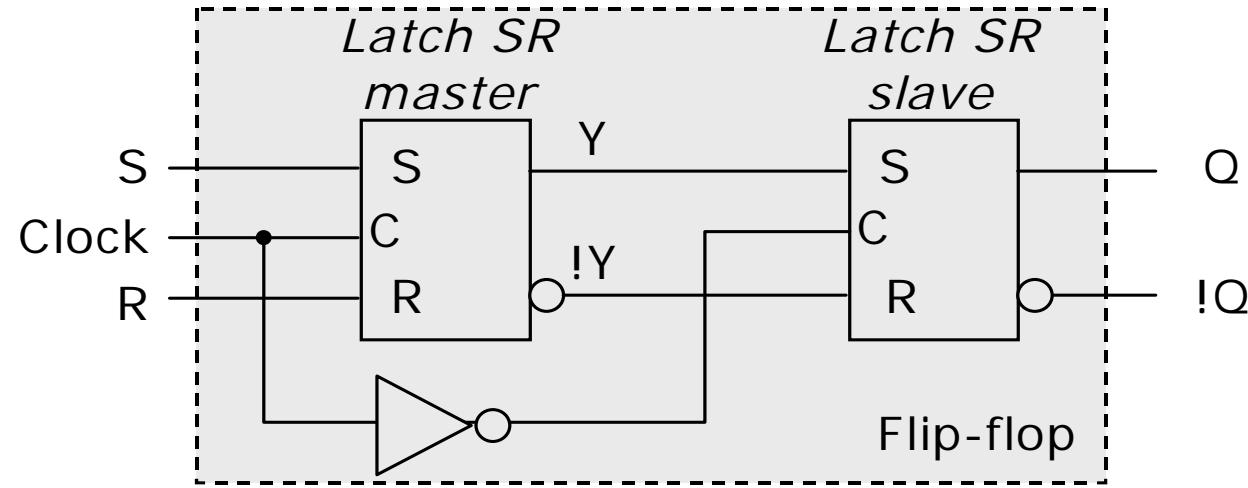


# Diagramma temporale



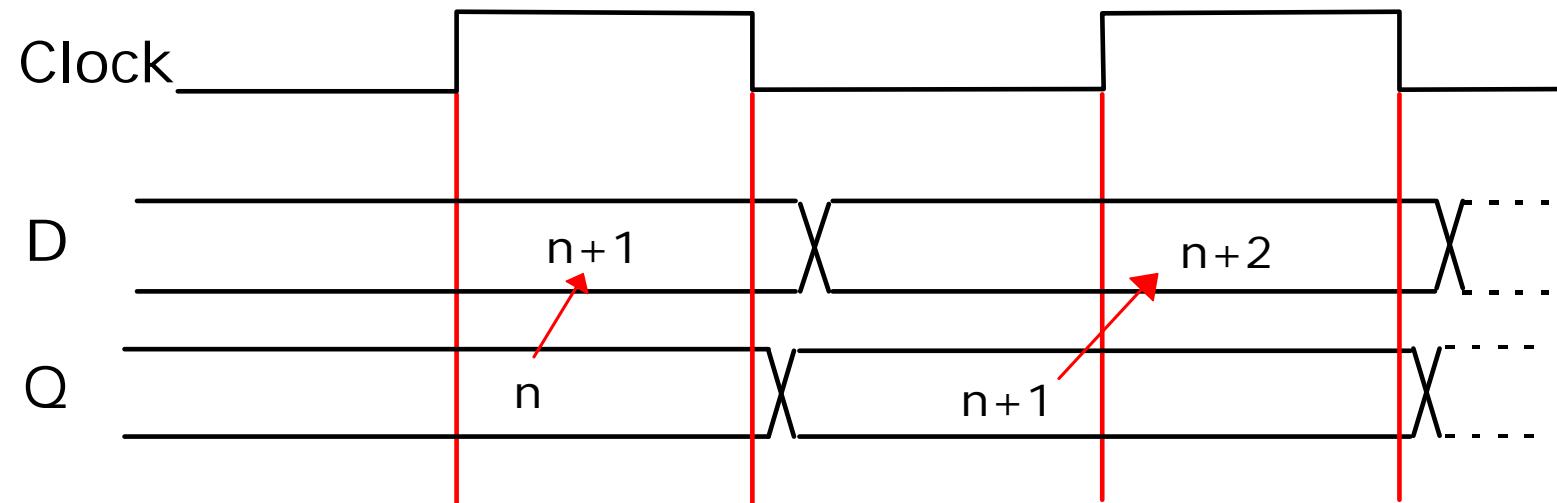
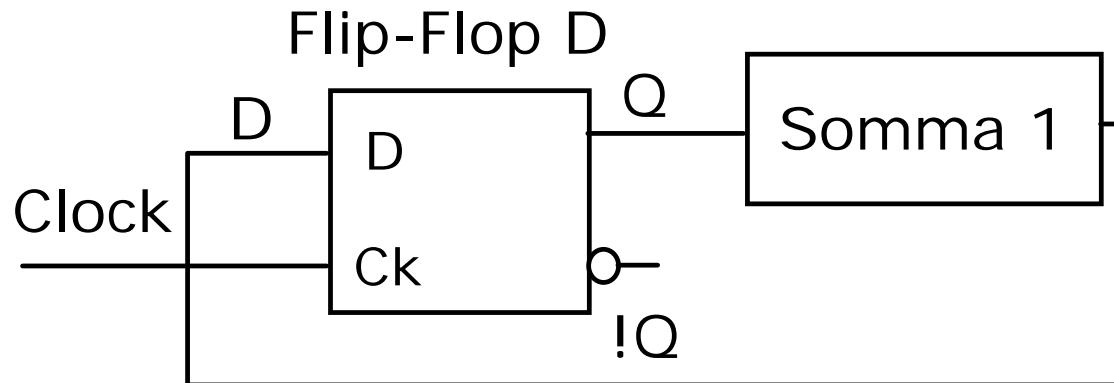


# Flip-flop SR master-slave





# Contatore a 1 bit





---

# Il Livello Logico-Digitale

## Blocchi funzionali combinatori

18 -10-2015

---



# Blocchi funzionali combinatori

- Esiste una ben nota e ormai stabilizzata **libreria di blocchi funzionali predefiniti** di tipo combinatorio che contiene i blocchi per tutte le funzioni combinatorie di base
  - La libreria contiene anche blocchi funzionali di tipo sequenziale
- I tipici blocchi funzionali combinatori sono:
  - Multiplexer
  - Demultiplexer
  - Decoder (decodificatore)
  - Confrontatore
  - Shifter combinatorio
  - Half adder e Full adder
  - Addizionatore a n bit
  - ALU or, and, not e somma



# Multiplexer

- Il blocco funzionale multiplexer ha:
  - $n \geq 1$  ingressi di selezione
  - $2^n \geq 2$  ingressi dati
  - un'uscita
- Gli ingressi dati sono numerati a partire da 0:  $k = 0, 1, 2, \dots, 2^n - 1$
- Se sugli ingressi di selezione è presente il numero binario  $k$ , il  $k^{\text{esimo}}$  ingresso dati viene inviato in uscita



# Multiplexer a 1 ingresso di selezione (1)

- 1 ingresso di selezione, 2 ingressi dati, un'uscita

I0	I1	Sel (Ctrl)	OUT
D1	D2	0	D1
D1	D2	1	D2

Tabella della verità

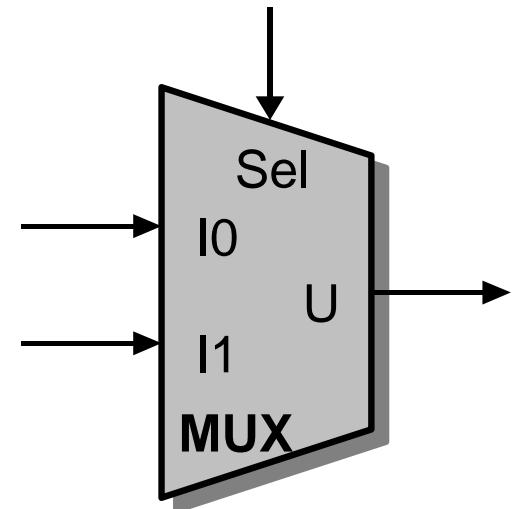
I0	I1	Sel (Ctrl)	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



## Multiplexer a 1 ingresso di selezione (2)

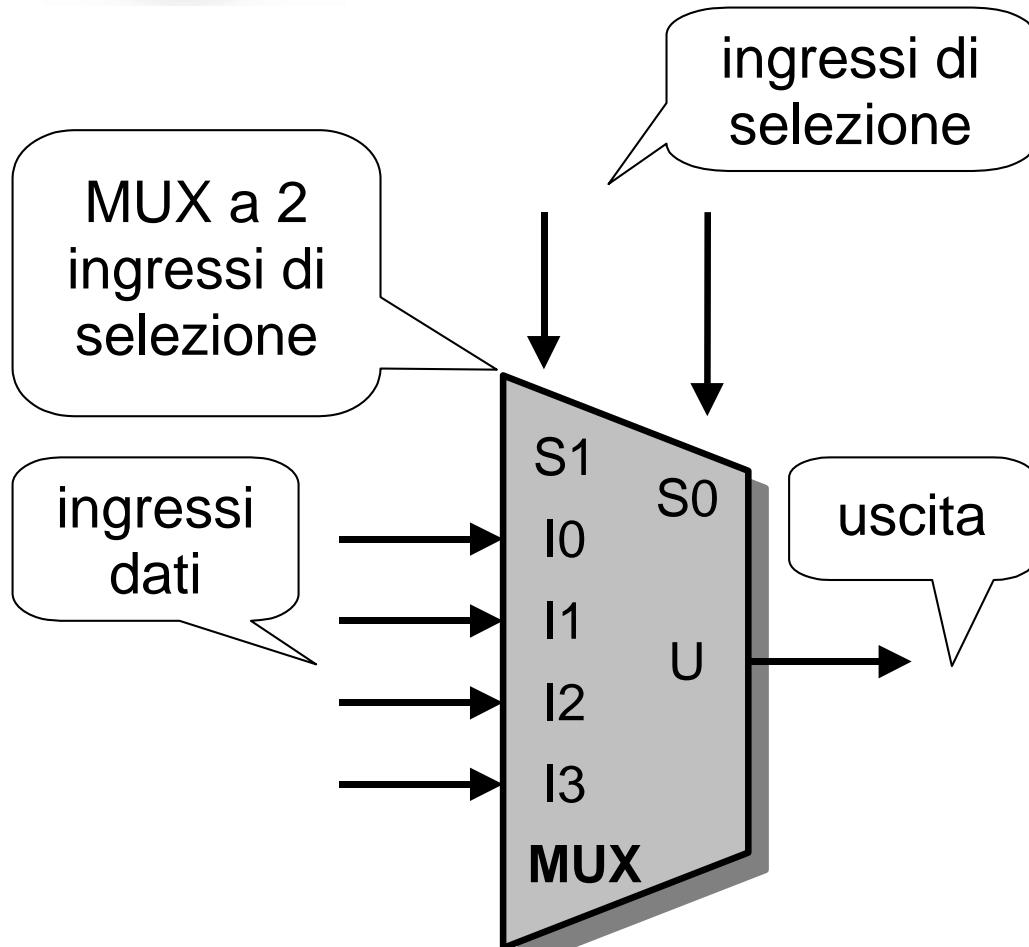
- 1 ingresso di selezione, 2 ingressi dati, un'uscita

$$\begin{aligned} \text{OUT} = & \text{ !Sel I0 !I1 + !Sel I0 I1} \\ & + \text{ Sel !I0 I1 + Sel I0 I1} \end{aligned}$$





# Multiplexer a 2 ingressi di selezione



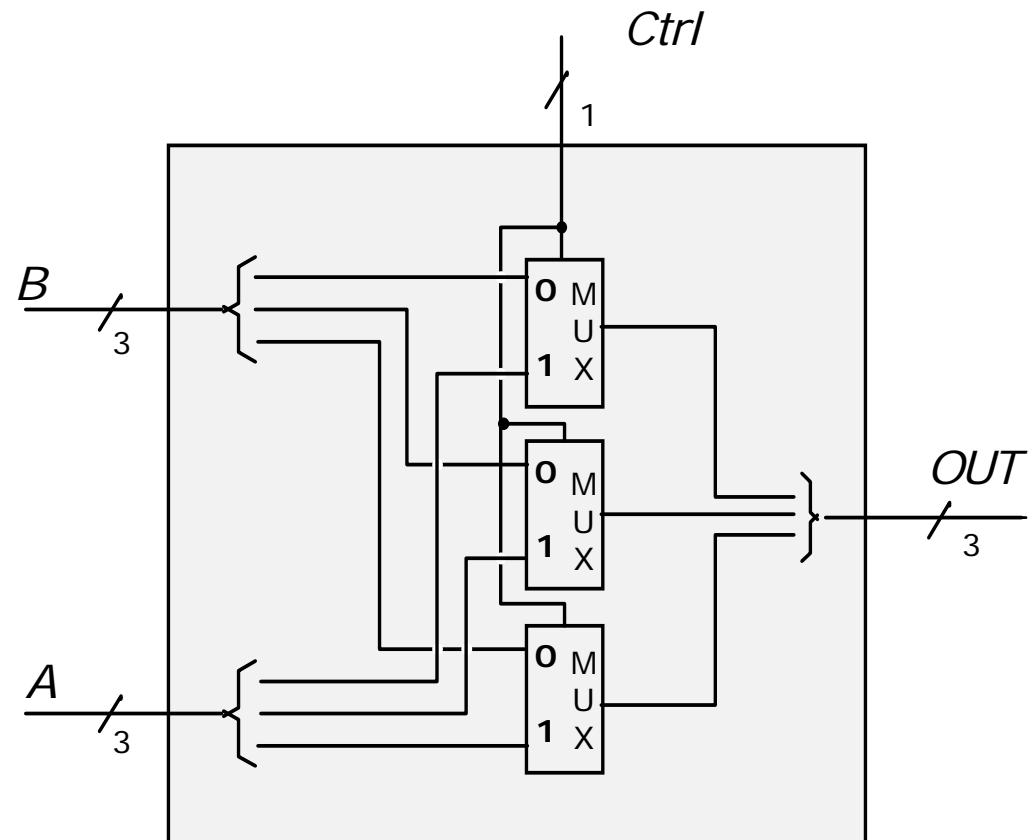
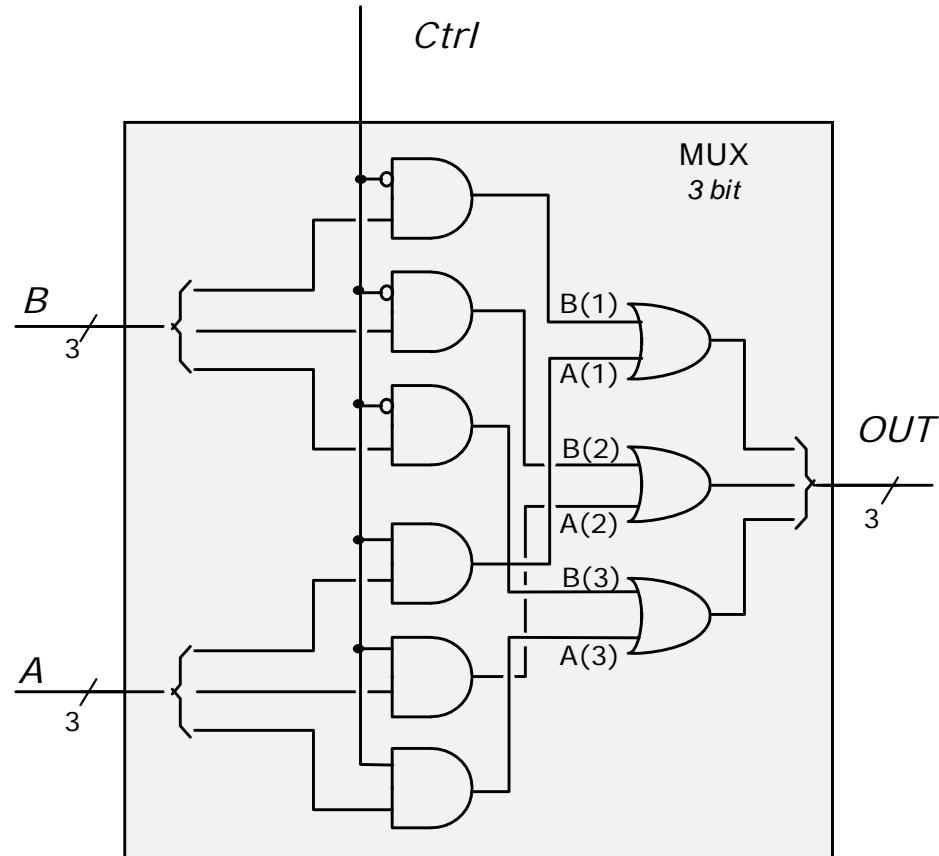
## Tabella delle verità

# riga	$S_1$	$S_0$	$I_0$	$I_1$	$I_2$	$I_3$	$U$
0	0	0	0	X	X	X	0
1	0	0	1	X	X	X	1
2	0	1	X	0	X	X	0
3	0	1	X	1	X	X	1
4	1	0	X	X	0	X	0
5	1	0	X	X	1	X	1
6	1	1	X	X	X	0	0
7	1	1	X	X	X	1	1



# Multiplexer a 2 ingressi dati da k bit

Esempio:  $k=3$





# Demultiplexer

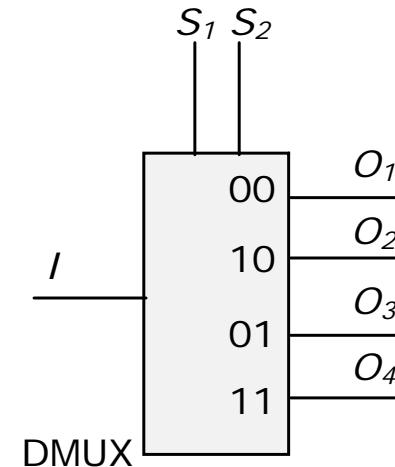
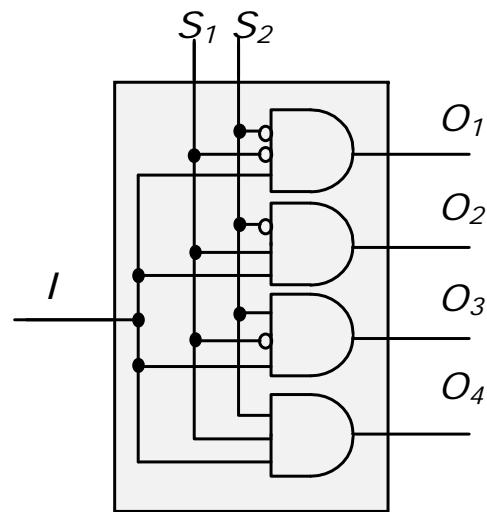
- Il blocco funzionale demultiplexer ha:
  - $n \geq 1$  ingressi di selezione
  - 1 ingresso dati
  - $2^n \geq 2$  uscite
- Le uscite sono numerate a partire da 0:  $k = 0, 1, 2, \dots, 2^n - 1$
- Se sugli ingressi di selezione è presente il numero binario  $k$ , l'ingresso dati viene inviato alla  $k^{\text{esima}}$  uscita, le rimanenti sono a 0



# Demultiplexer a 2 ingressi di selezione

Ingressi	Selezione		Uscite			
<i>I</i>	<i>S</i> <sub>1</sub>	<i>S</i> <sub>2</sub>	<i>O</i> <sub>1</sub>	<i>O</i> <sub>2</sub>	<i>O</i> <sub>3</sub>	<i>O</i> <sub>4</sub>
<i>D</i>	0	0	<i>D</i>	0	0	0
<i>D</i>	1	0	0	<i>D</i>	0	0
<i>D</i>	0	1	0	0	<i>D</i>	0
<i>D</i>	1	1	0	0	0	<i>D</i>

$$\begin{aligned}O_1 &= !S_1!S_2 I & O_3 &= !S_1S_2 I \\O_2 &= S_1!S_2 I & O_4 &= S_1S_2 I\end{aligned}$$





# Decodificatore (decoder)

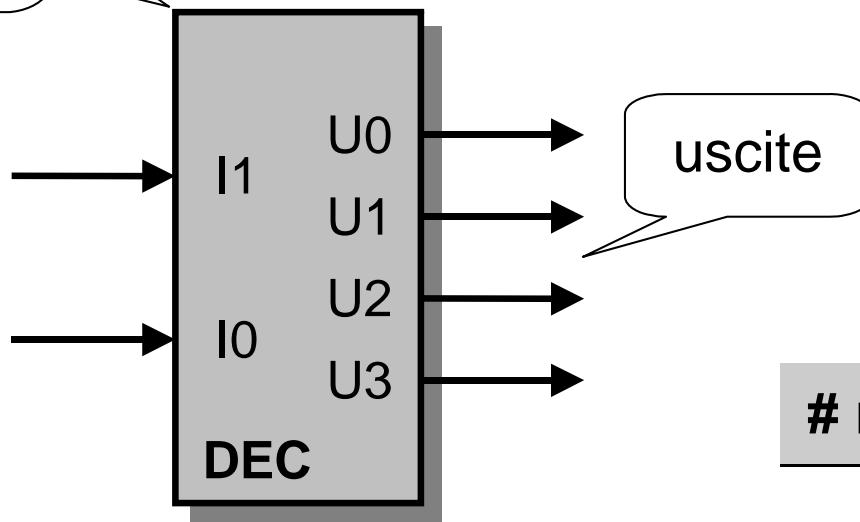
- Il blocco funzionale decodificatore ha:
  - $n \geq 1$  ingressi
  - $2^n \geq 2$  uscite
- Le uscite sono numerate a partire da 0:  $k = 0, 1, 2, \dots, 2^n - 1$
- Se sugli ingressi è presente il numero binario  $k$ , la  $k^{\text{esima}}$  uscita assume il valore 1 e le restanti uscite assumono il valore 0



# Decodificatore

DEC a 2 ingressi

ingressi



Decodificatore  
a 2 ingressi

Tabella delle verità

# riga	I1	I0	U0	U1	U2	U3
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1



# Confrontatore (comparator)

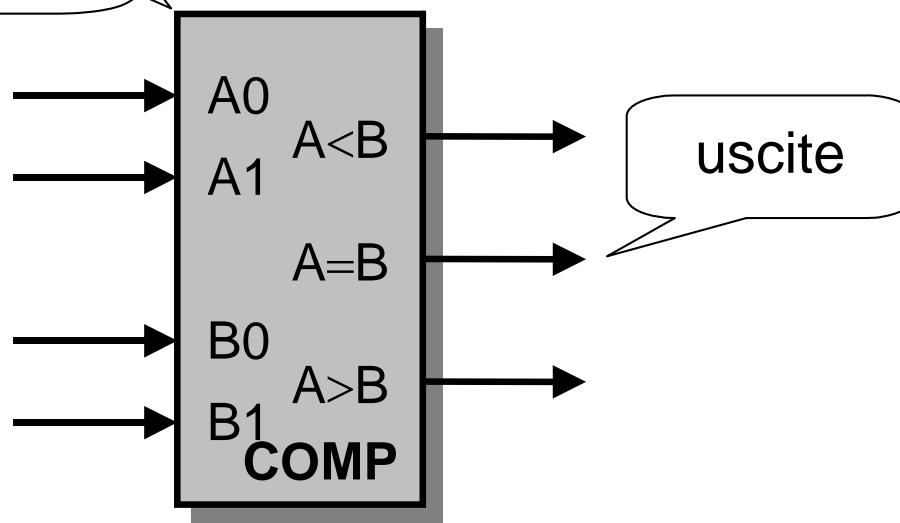
- Il blocco funzionale confrontatore ha:
  - due gruppi A e B di ingressi da  $n \geq 1$  bit ciascuno
  - tre uscite: minoranza  $A < B$ , uguaglianza  $A = B$  e maggioranza  $A > B$
- . Il blocco confronta i due numeri binari A e B da n bit presenti sui due gruppi di ingressi, e attiva (a 1) l'uscita corrispondente all'esito del confronto



# Confrontatore

COMP a 2 bit

ingressi



Confrontatore di numeri  
a 2 bit

Tabella delle verità

# riga	A1	A0	B1	B0	A<B	A=B	A>B
0	0	0	0	0	0	1	0
1	0	0	0	1	1	0	0
2	0	0	1	0	1	0	0
3	0	0	1	1	1	0	0
4	0	1	0	0	0	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	1	0	0
7	0	1	1	1	1	0	0
8	1	0	0	0	0	0	1
9	1	0	0	1	0	0	1
10	1	0	1	0	0	1	0
11	1	0	1	1	1	0	0
12	1	1	0	0	0	0	1
13	1	1	0	1	0	0	1
14	1	1	1	0	0	0	1
15	1	1	1	1	0	1	0

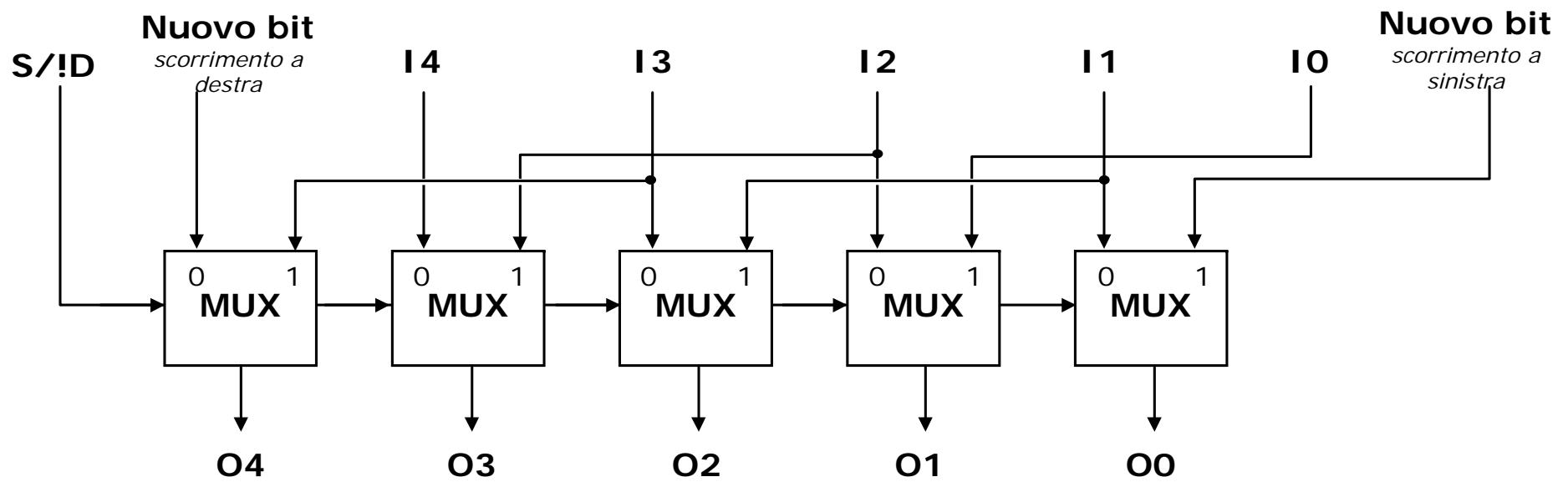


# Shifter combinatorio

- Il blocco funzionale shifter ha:
  - $n \geq 1$  ingressi
  - 1 ingresso per il bit aggiunto a dx (scorrimento a sinistra)
  - 1 ingresso per il bit aggiunto a sx (scorrimento a destra)
  - 1 ingresso di controllo che comanda lo scorrimento a destra o a sinistra
  - $n \geq 1$  uscite
- Uscite:
  - scorrimento a dx: bit aggiunto a sx + ingressi shiftati di una posizione a dx (viene "perso" il bit meno significativo degli ingressi)
  - scorrimento a sx: bit aggiunto a dx + ingressi shiftati di una posizione a sx (viene "perso" il bit più significativo degli ingressi)
- Si noti che se si considerano gli ingressi come un valore numerico espresso in binario naturale
  - lo scorrimento a dx (con bit aggiunto a sx = 0) equivale ad una **divisione per 2**
  - lo scorrimento a sx (con bit aggiunto a dx = 0) equivale ad una **moltiplicazione per 2**



# Shifter combinatorio 5 ingressi





# Blocchi aritmetici fondamentali

- Rappresentazione dei numeri in binario **naturale** intero su  $k \geq 1$  bit
  - Addizionatore ad 1 bit
    - half adder
    - full adder
  - Addizionatore a  $k$  bit in binario naturale intero
  - Sottrattore a 1 bit
  - Sottrattore a  $k$  bit in binario naturale intero



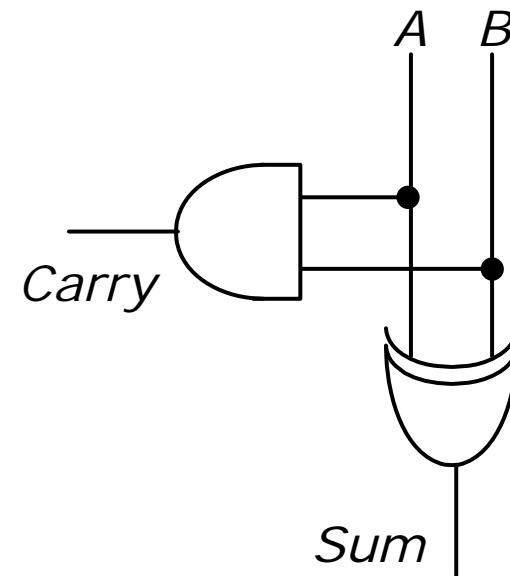
# Half adder (semisommatore)

## HALF-ADDER

<b>A</b>	<b>B</b>	<b>Carry</b>	<b>Sum</b>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\text{Sum} = A \oplus B$$

$$\text{Carry} = AB$$





# Full adder (sommatore completo)

FULL ADDER				
A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = A \text{ xor } B \text{ xor } C_{\text{in}}$$

$$C_{\text{out}} = AB + C(A \text{ xor } B)$$

$$\begin{aligned} S &= !A!BC + !AB!C + A!B!C + ABC = \\ &= C (!A!B + AB) + !C (!AB + A!B) \\ &= C !(A \text{ xor } B) + !C (A \text{ xor } B) \end{aligned}$$

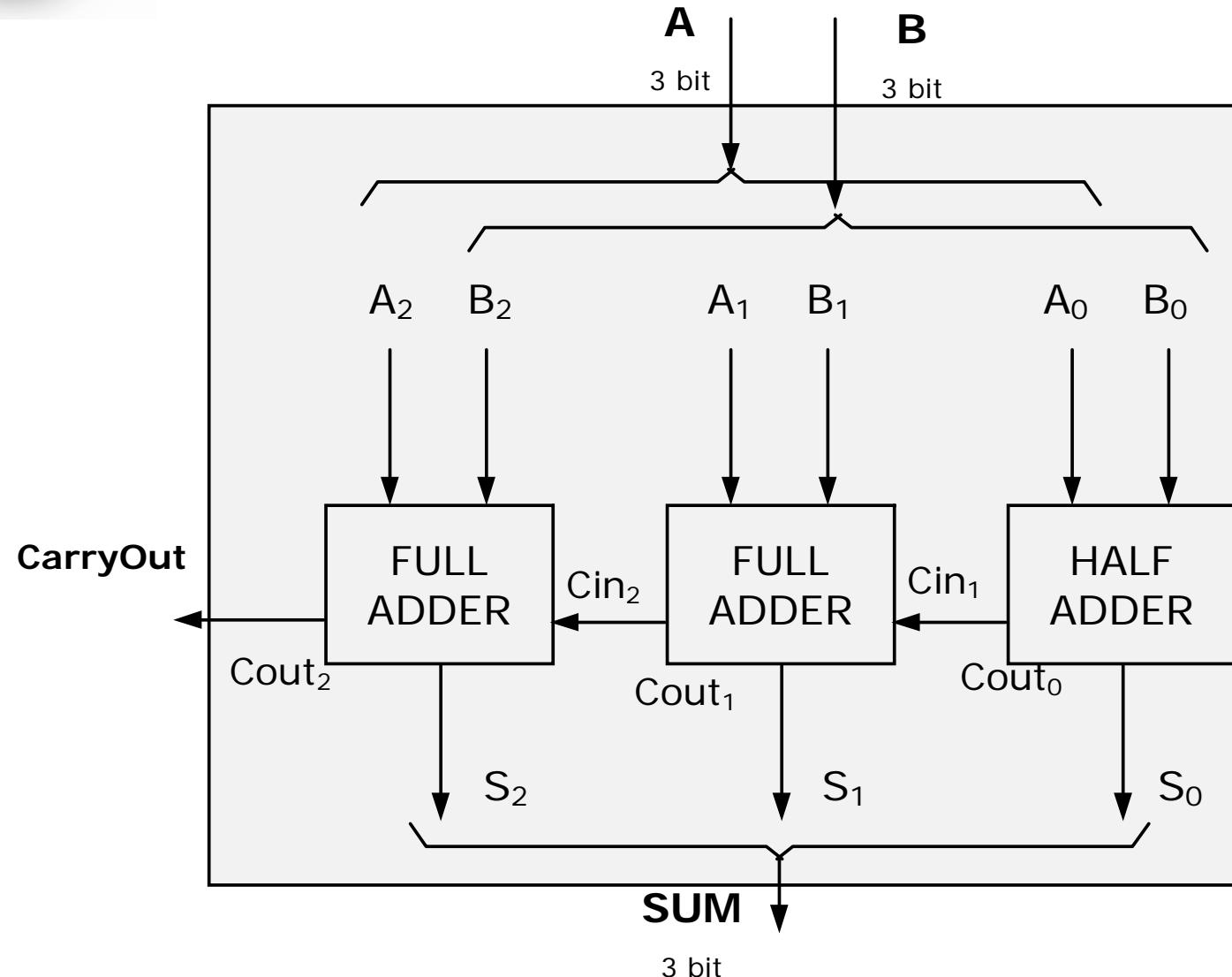
$$\begin{aligned} \text{ponendo } Z &= A \text{ xor } B \\ &= !Z C + Z !C = Z \text{ xor } C \\ &= A \text{ xor } B \text{ xor } C_{\text{in}} \end{aligned}$$

$$\begin{aligned} C_{\text{out}} &= !ABC + A!BC + AB!C + ABC \\ &= C (!AB + A!B) + AB(!C + C) \\ &= C (A \text{ xor } B) + AB \end{aligned}$$

..... *schema rete da ricavare .....*

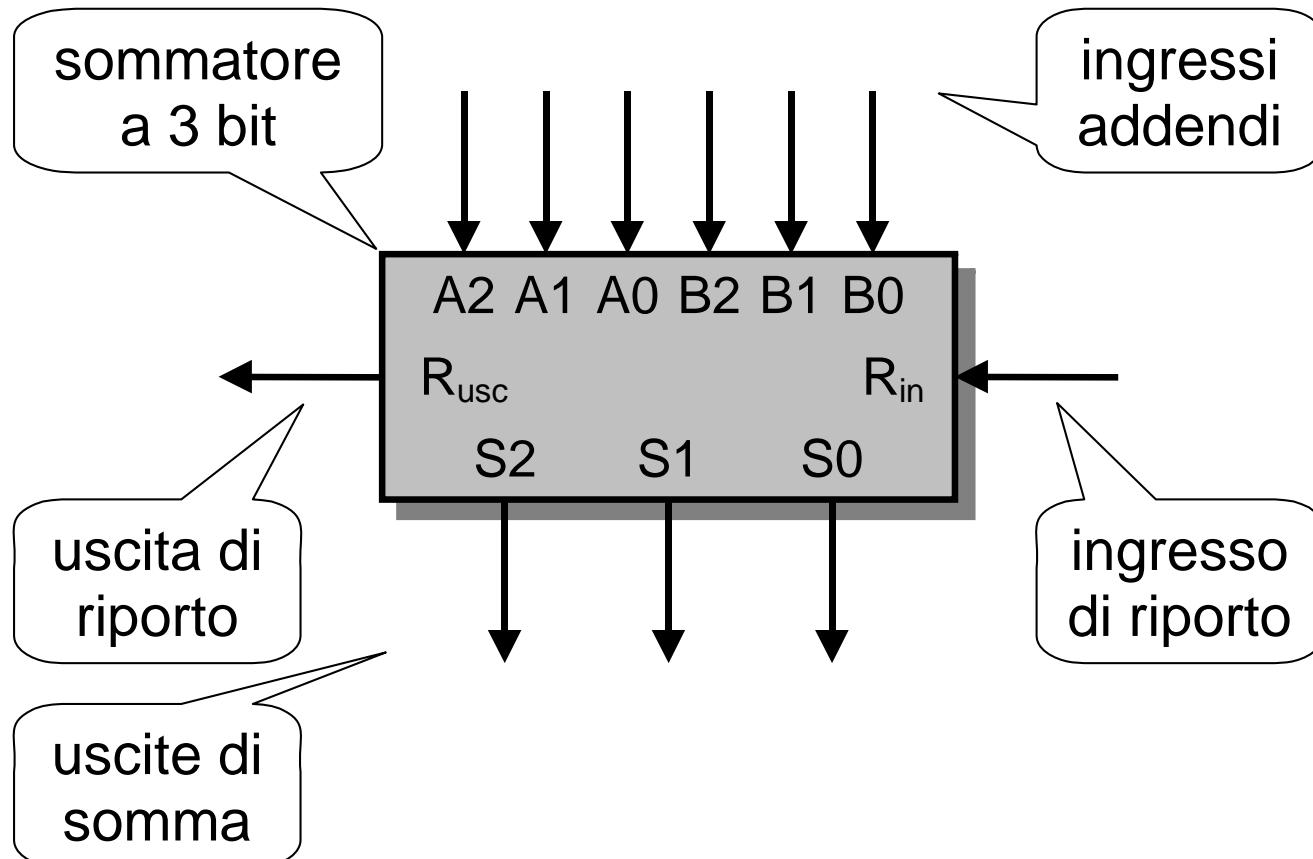


# Addizionatore a k bit in binario naturale intero





# Sommatore intero a n bit



Sommatore intero binario naturale a 3 bit



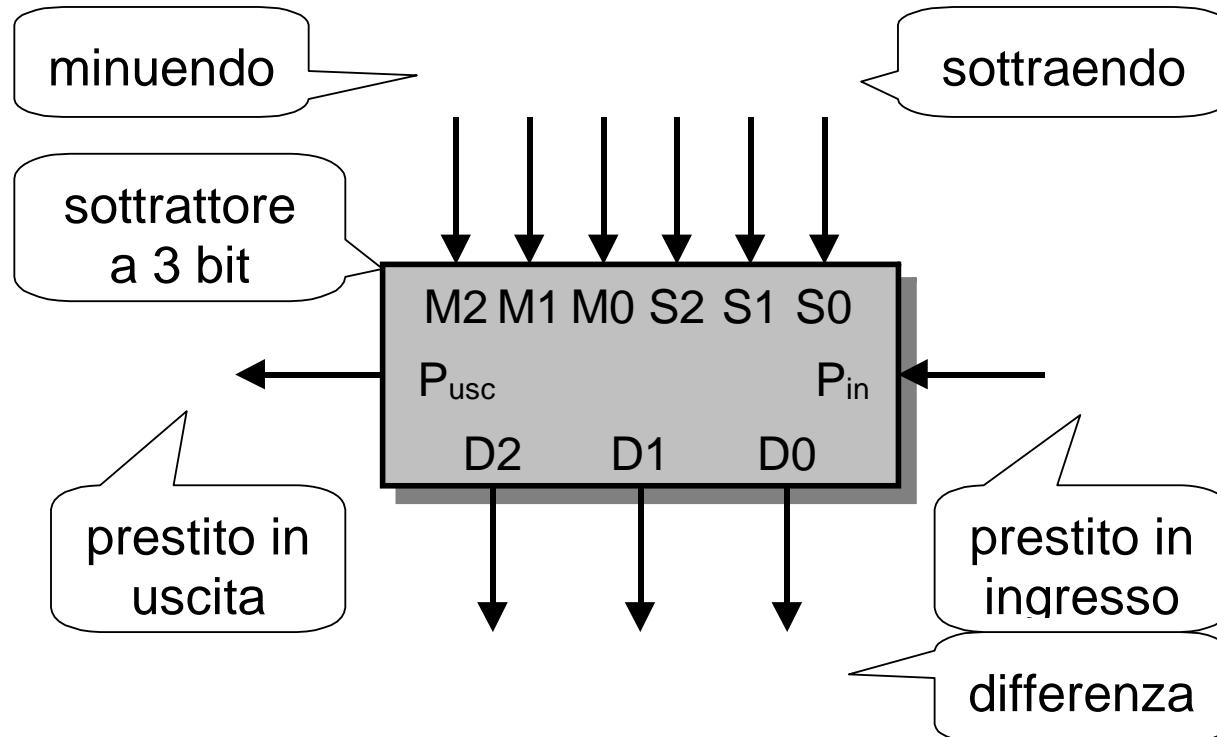
# Sottrattore a 1 bit e a k bit in binario naturale intero

---

- Ricavare le tabelle delle verità, l'espressione logica minima e la rete combinatoria che realizza
  - un **semi-sottrattore a 1 bit**
  - un **sottrattore completo a 1 bit**
  
- Disegnare la struttura modulare di un **sottrattore a k bit**



# Sottrattore intero a n bit



Sottrattore intero binario naturale a 3 bit



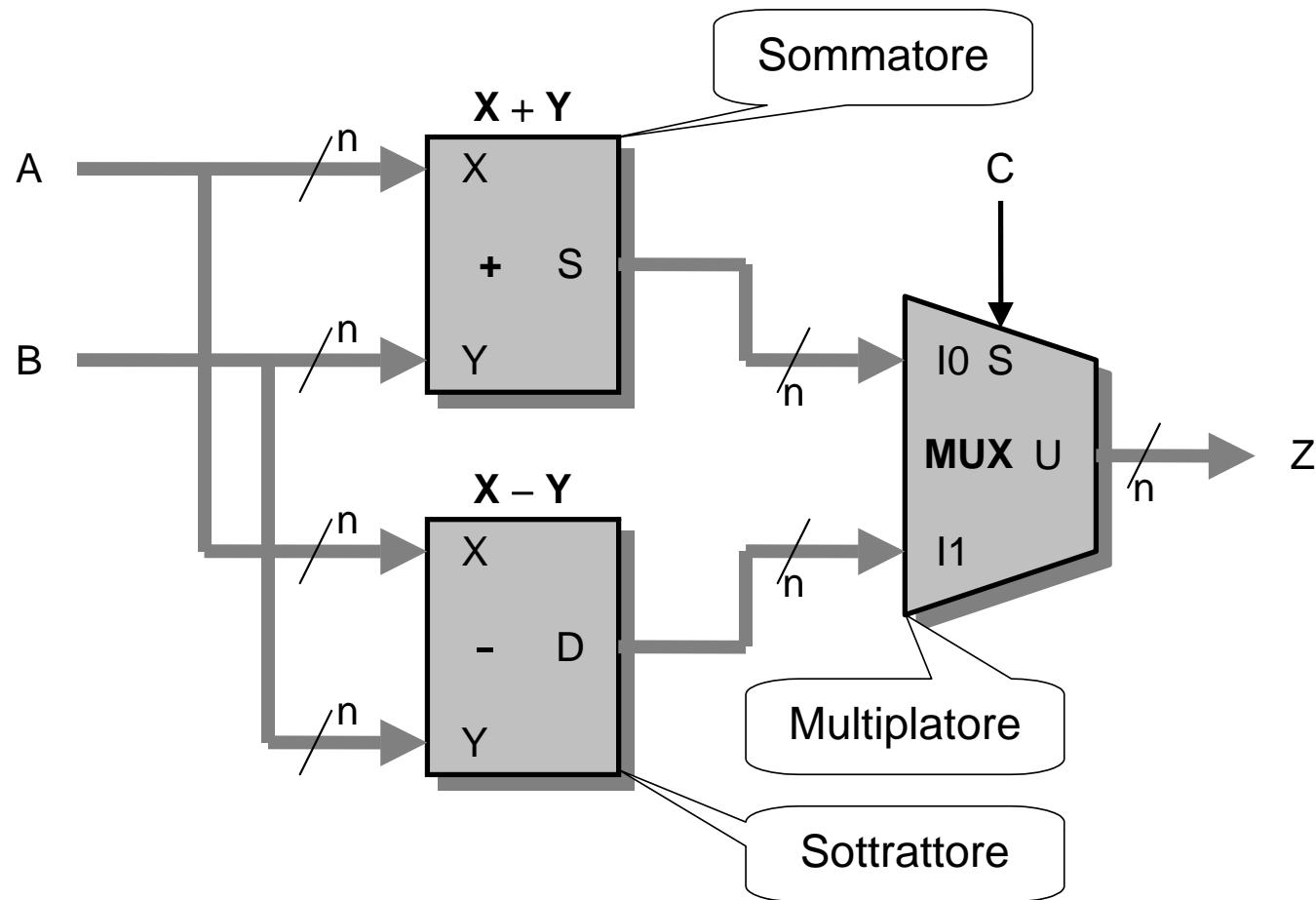
## Semplice esempio di progetto in stile funzionale

---

- Si chiede di progettare un circuito digitale combinatorio, che abbia:
  - in ingresso due numeri interi binari naturali (positivi)  $A$  e  $B$  da  $n \geq 1$  bit ciascuno
  - in ingresso un segnale di comando  $C$
  - in uscita un numero intero binario naturale  $Z$  da  $n \geq 1$  bit
- Su  $Z$  deve presentarsi la somma  $A + B$  se  $C = 0$ , la differenza  $A - B$  se  $C = 1$

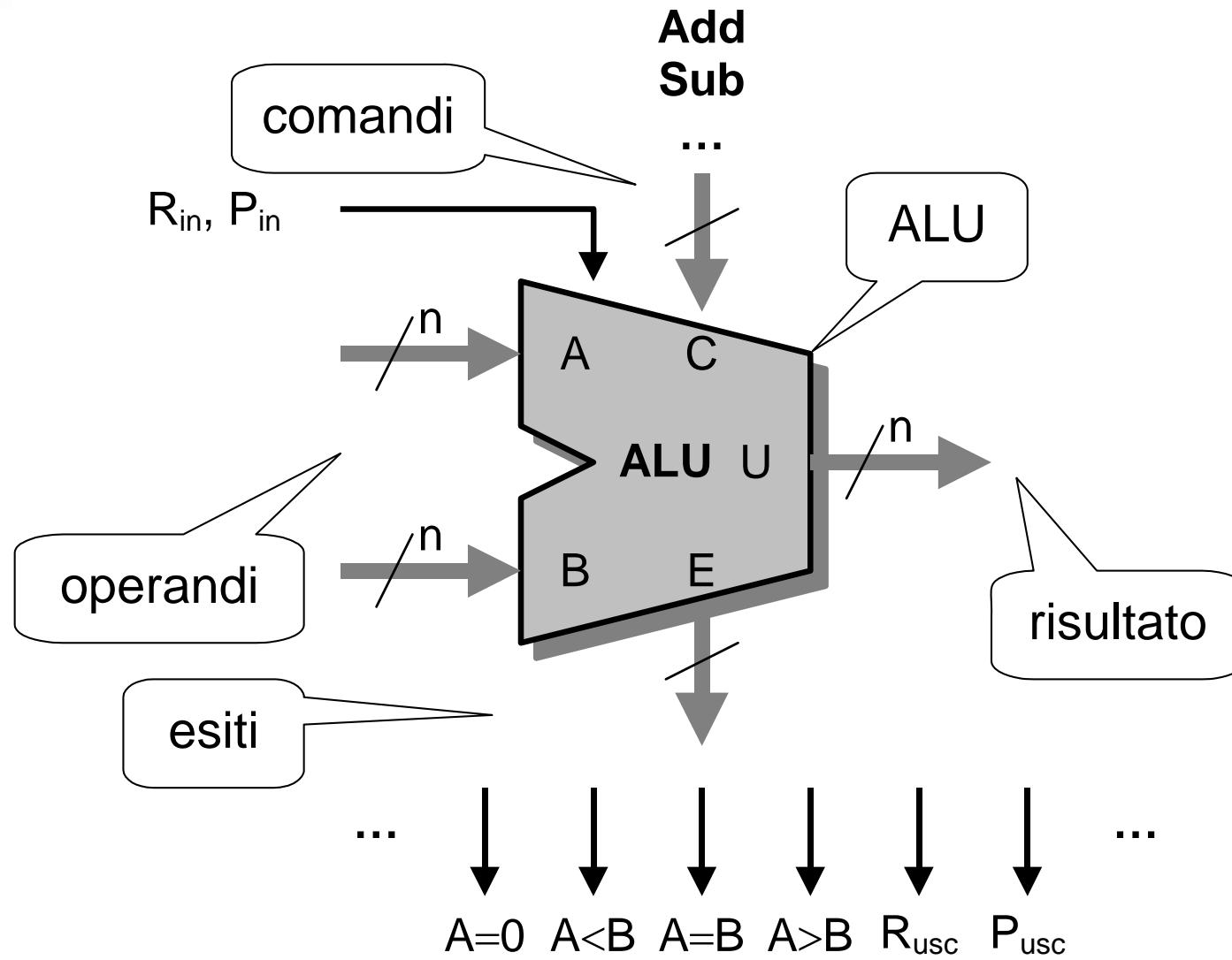


# Schema logico della soluzione





# Unità Aritmetico-Logica





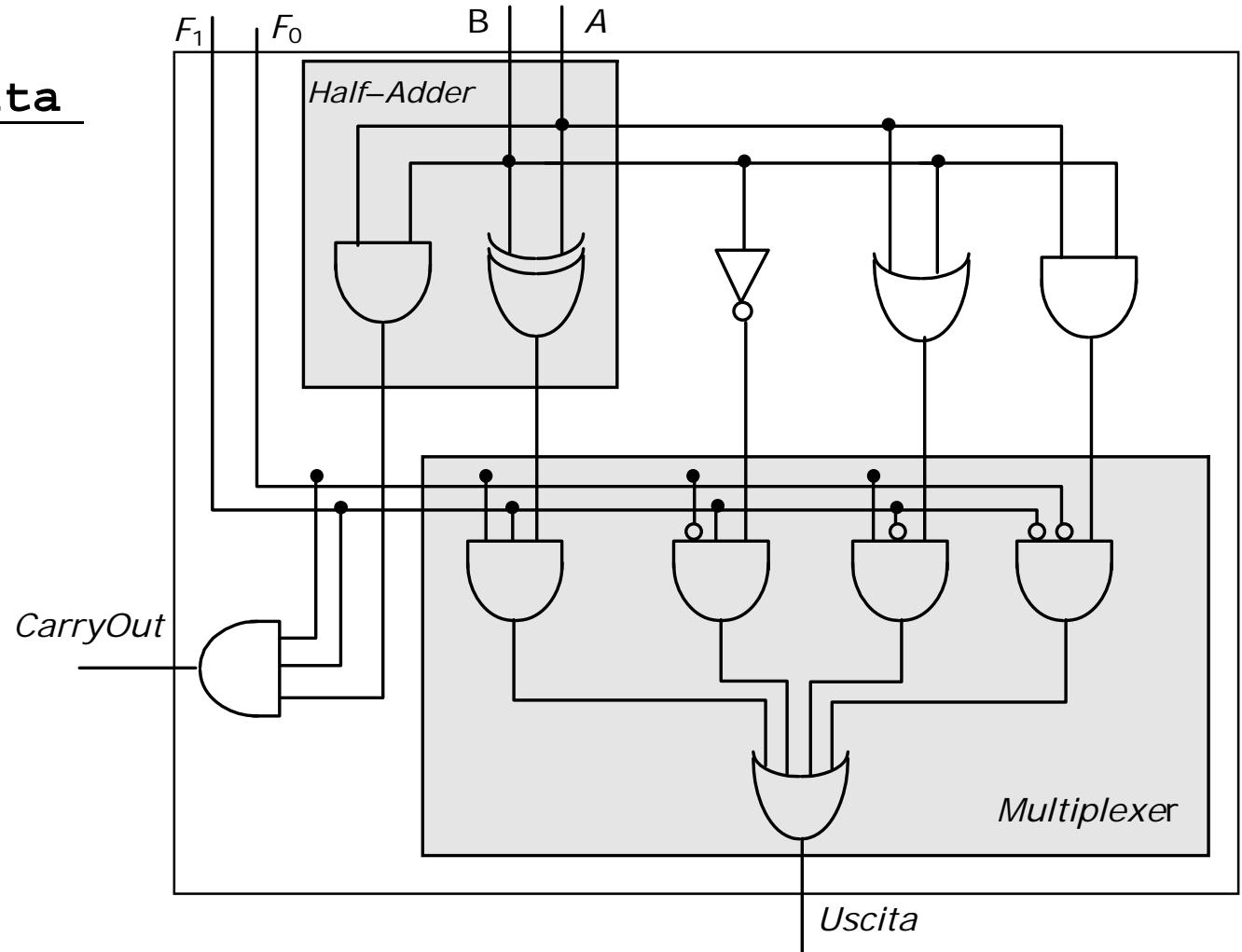
# Unità Aritmetico-Logica: esempio di comandi

# riga	Comando	Operazione	R	Esito
0	Add	somma A e B	$A + B + R_{in}$	riporto in uscita $R_{usc}$
1	Sub	sottrae B da A	$A - B - P_{in}$	prestito in uscita $P_{usc}$
2	Pass A	A passa in uscita	A	-
3	Pass B	B passa in uscita	B	-
4	Zero	annulla uscita	0	-
5	Shift Left A	A scorre a SX	2A	bit più significativo di A
6	Shift Right A	A scorre a DX	$A / 2$	bit meno significativo di A
7	Null	Confronta A con 0	-	$A = 0$
8	Compare	Confronta A con B	$A <,=,> B$	$A < B, A = B, A > B$
9	Multiply	prodotto di A e B	$A \times B$	riporto in uscita
10	Divide	divisione A / B	$A / B$	divisione per 0 ?
...	...	...	...	...



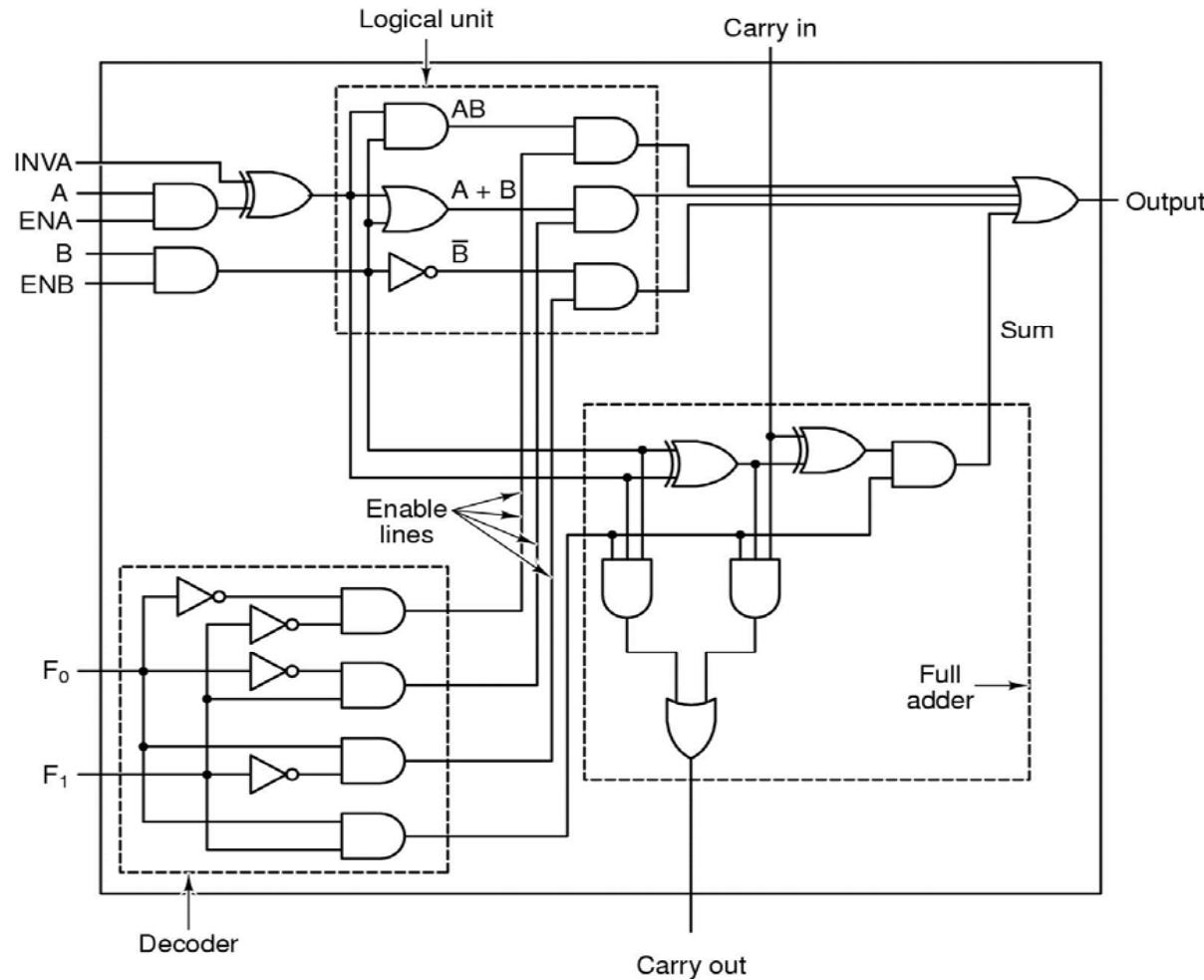
# ALU and, not, or e somma (1): realizzazione con MUX

$F_0 F_1$	Operazione svolta
0 0	$A \text{ and } B$
1 0	$A \text{ or } B$
0 1	$\neg B$
1 1	$A + B$





# ALU and, not, or e somma (2): realizzazione con DECODER



Schema logico di una ALU da 1 bit



# ALU e MIPS

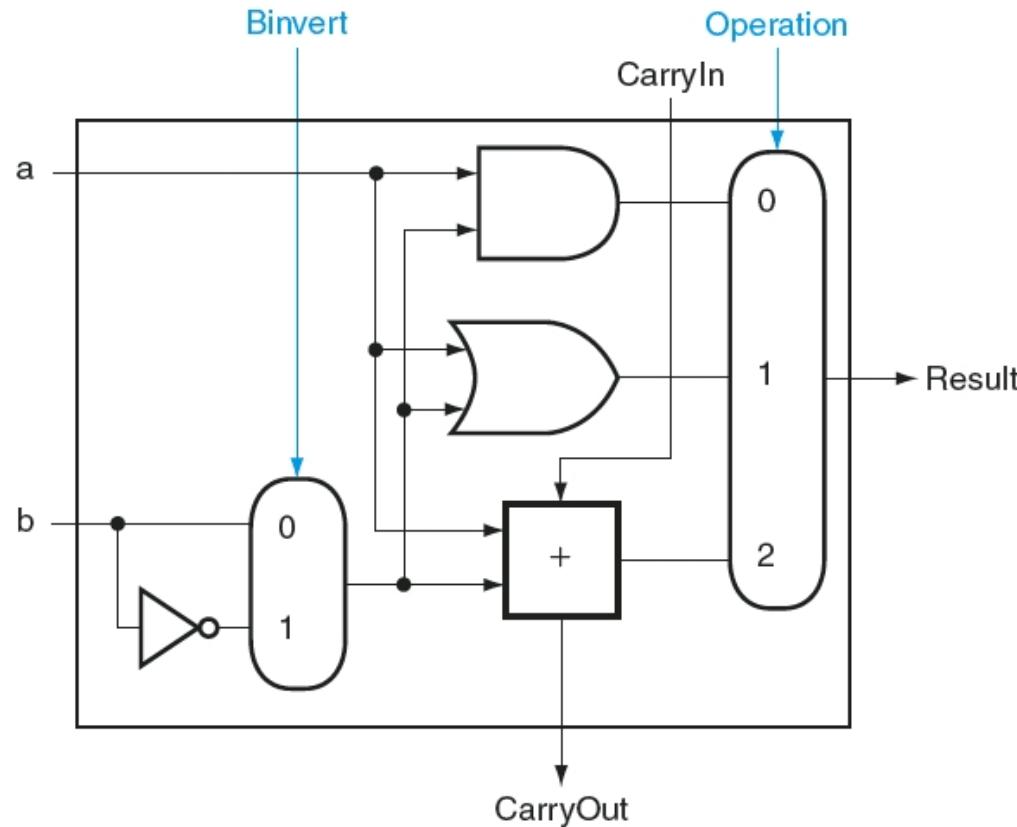
---

- Istruzioni aritmetico-logiche

- add
- sub
- or
- and
- nor



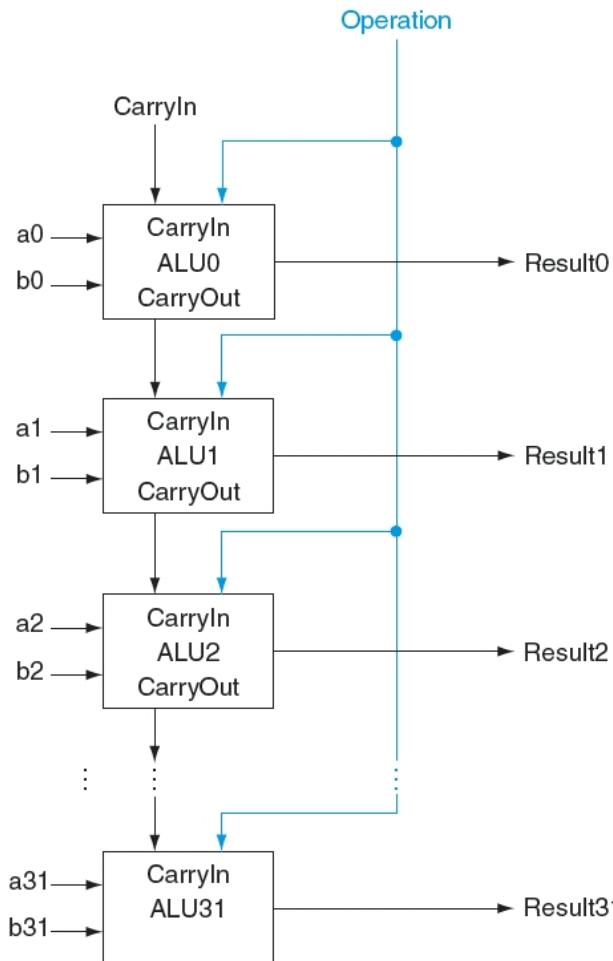
# Un altro schema ..... somma e sottrazione in complemento a 2



**FIGURE B.5.8 A 1-bit ALU that performs AND, OR, and addition on  $a$  and  $b$  or  $a$  and  $\bar{b}$ .** By selecting  $\bar{b}$  ( $\text{Binvert} = 1$ ) and setting  $\text{CarryIn}$  to 1 in the least significant bit of the ALU, we get two's complement subtraction of  $b$  from  $a$  instead of addition of  $b$  to  $a$ .



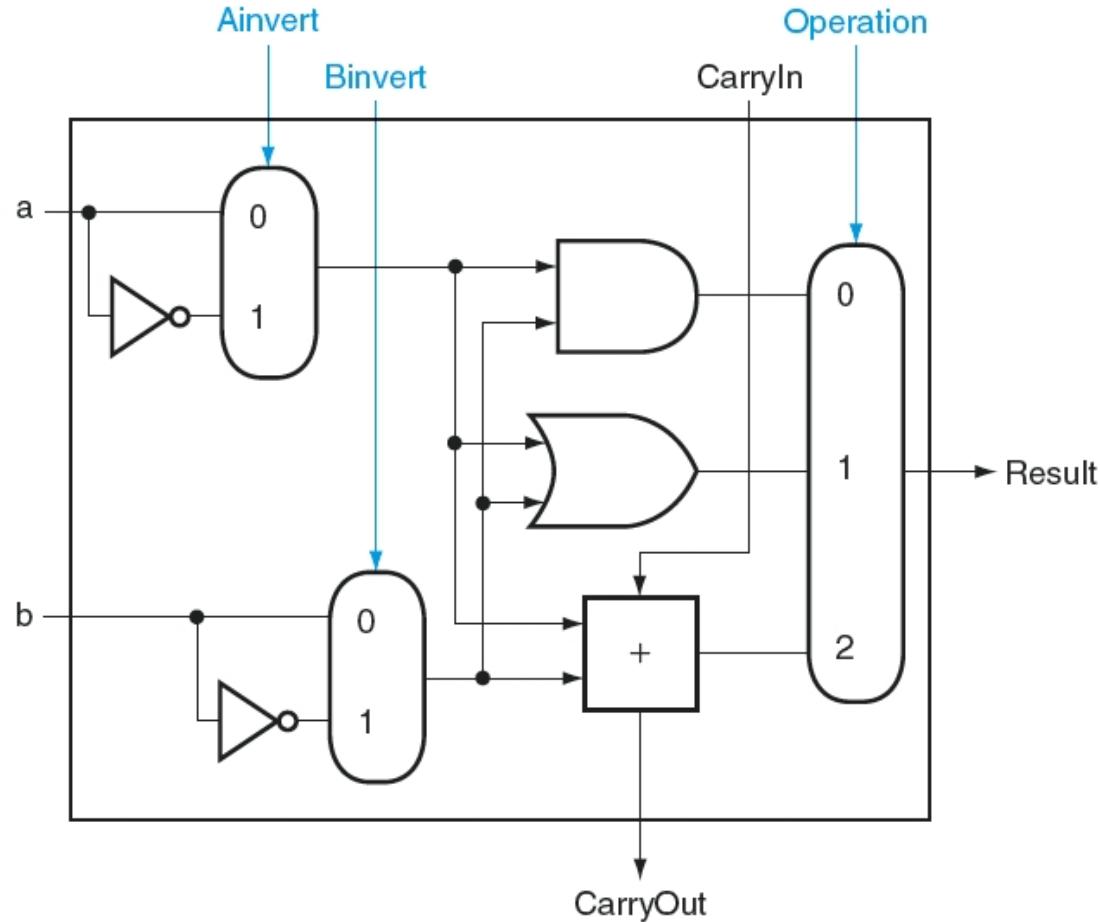
# ALU a 32 bit



**FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs.** CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.



# Si aggiunge il .... NOR



**FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on  $a$  and  $b$  or  $\bar{a}$  and  $\bar{b}$ .** By selecting  $\bar{a}$  ( $\text{Ainvert} = 1$ ) and  $\bar{b}$  ( $\text{Binvert} = 1$ ), we get a NOR  $b$  instead of a AND  $b$ .



# L'indispensabile «Overflow Detection»

... e anche la realizzazione di

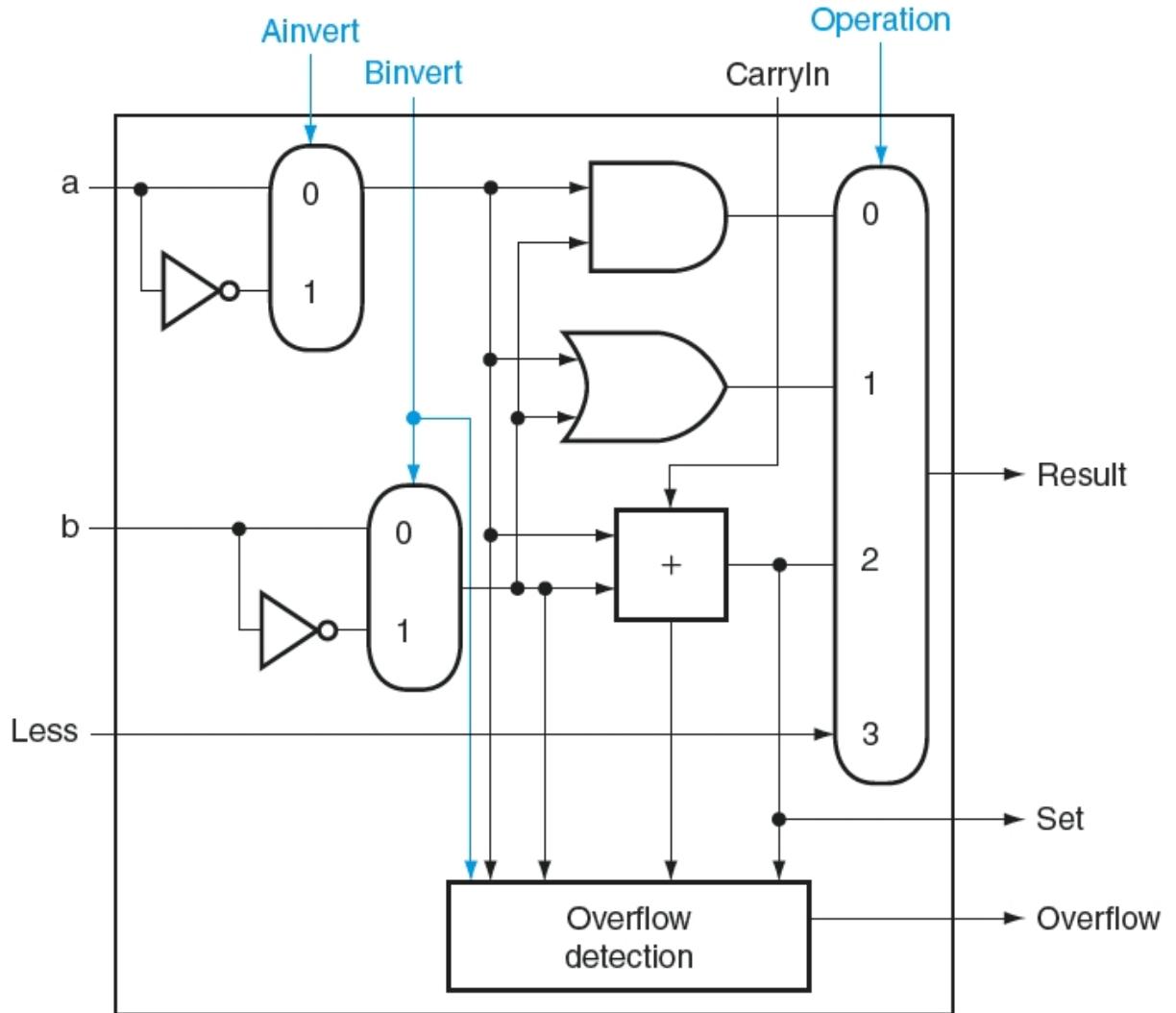
**slt \$s1, \$s2, \$s3**

se  $s2 < s3$  allora  $s1=1$  altrimenti  $s1=0$

$s2 < s3$  equivale a  $s2 - s3 < 0$

se  $s2 - s3 < 0$  allora il bit di segno del sommatore è 1 altrimenti è 0

L'ingresso **less** viene portato direttamente in uscita tramite il multiplexer e deve essere  
**0** per i 31 bit più significativi di  $\$s1$  e  
**1** oppure **0** per il bit meno significativo





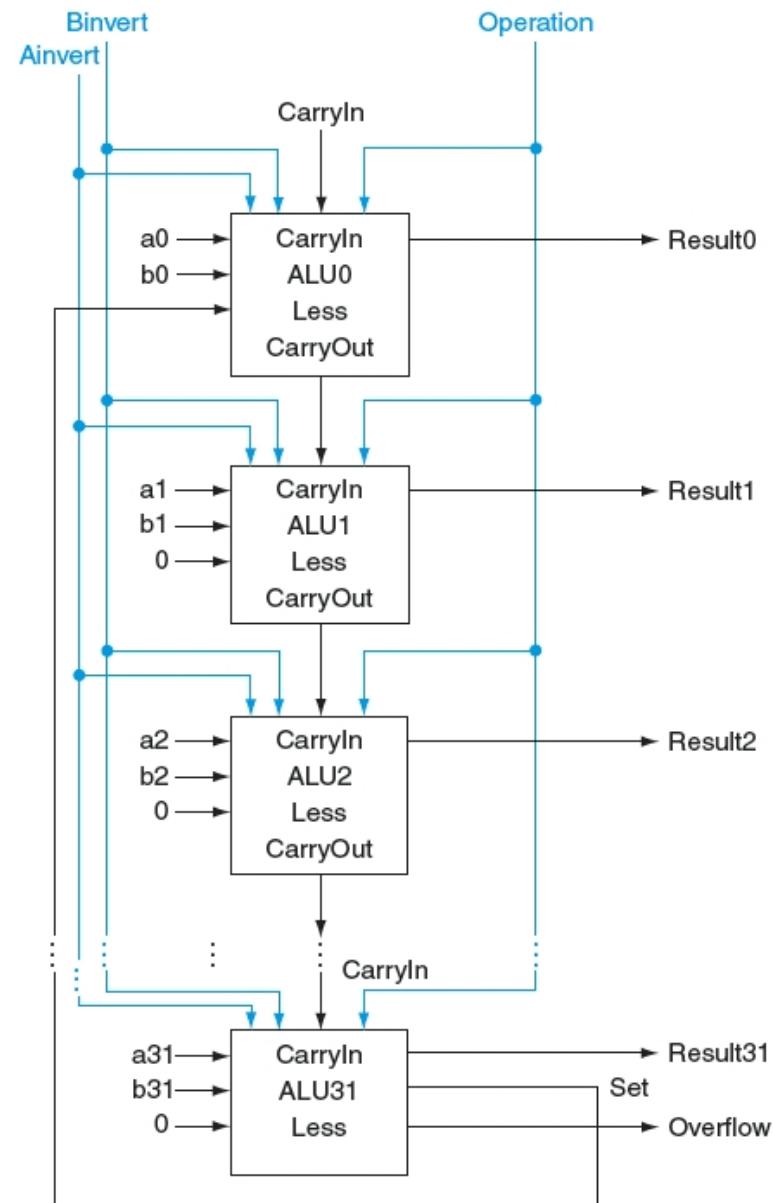
... a 32 bit

## Overflow

in uscita solo quello  
relativi al bit più  
significativo

## Slt

ecco i valori di tutti i bit





# Test zero e sottrazione

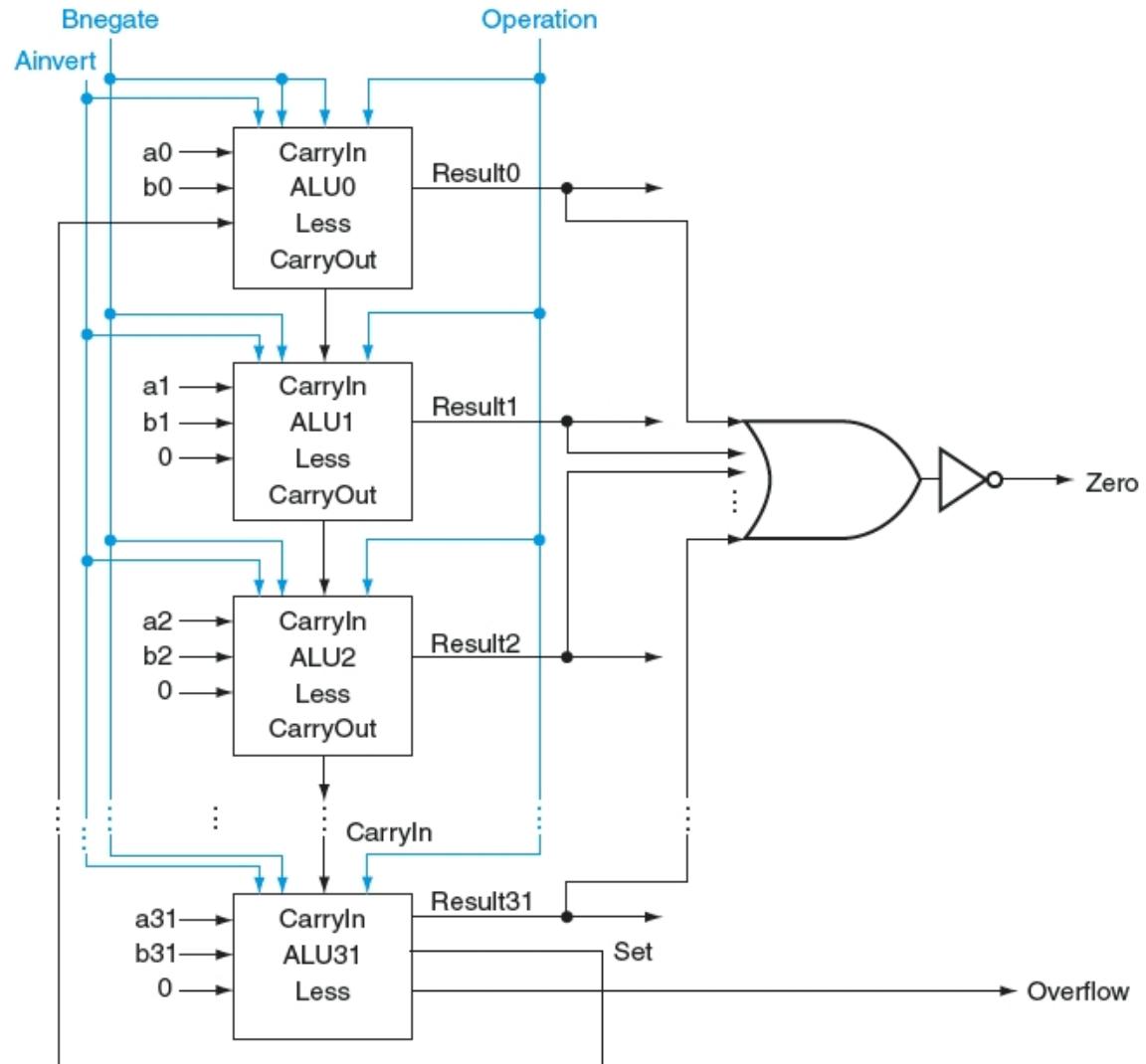
## bit **Zero**

- posto a 1 se il risultato dell'ALU vale zero
- altrimenti a 0

## **Bnegate**

Per la sottrazione in cpl2 è necessario avere i segnali **Binvert** e **Carryin** a 1 mentre per le altre operazioni considerate, eccetto il NOR, sono entrambi a 0

Vengono sostituiti dall'unico segnale **Bnegate**



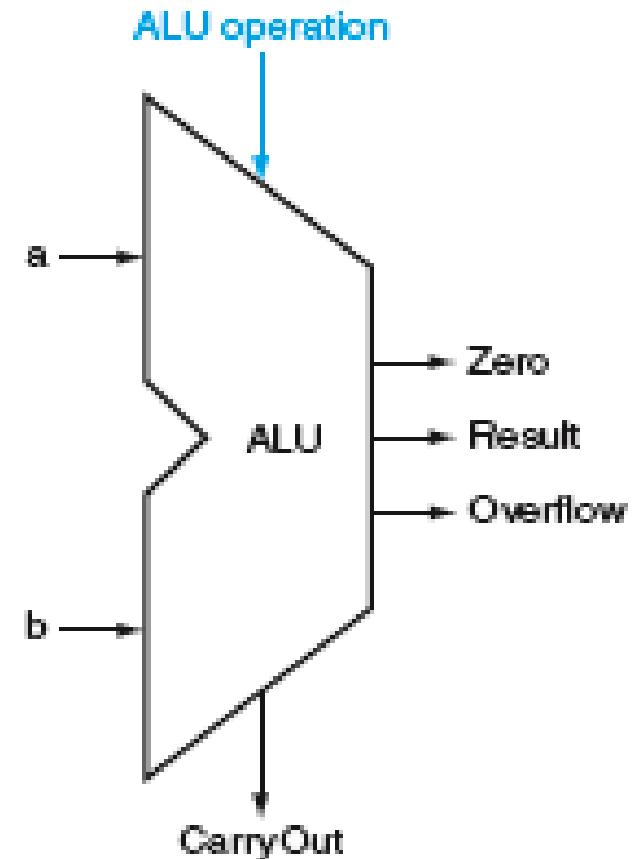


# ALU del MIPS e linee di controllo

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Le linee di controllo interne  
dell'ALU sono:

Ainvert, Bnegate, Operation(2)



... come blocco funzionale



# Numeri relativi e numeri reali

- I **numeri relativi** sono rappresentabili tramite sequenze di bit, proprio come i numeri interi naturali (sempre positivi)
  - La tecnica più usata per rappresentare i numeri interi relativi è il **complemento a due** (two's complement)
  - Le ALU sono normalmente in grado di operare sia con numeri interi naturali sia con numeri interi relativi rappresentati in complemento a due
- I **numeri reali** sono rappresentabili tramite sequenze di bit, proprio come i numeri interi
  - Esiste uno standard internazionale per la rappresentazione binaria di numeri reali: lo standard **IEEE 754** per la rappresentazione in **virgola mobile**
  - Esistono ALU in grado di effettuare i calcoli aritmetici con i numeri reali, oltre che con i numeri interi



---

# II Livello Logico-Digitale

## Blocchi funzionali sequenziali

22 -10 -2015



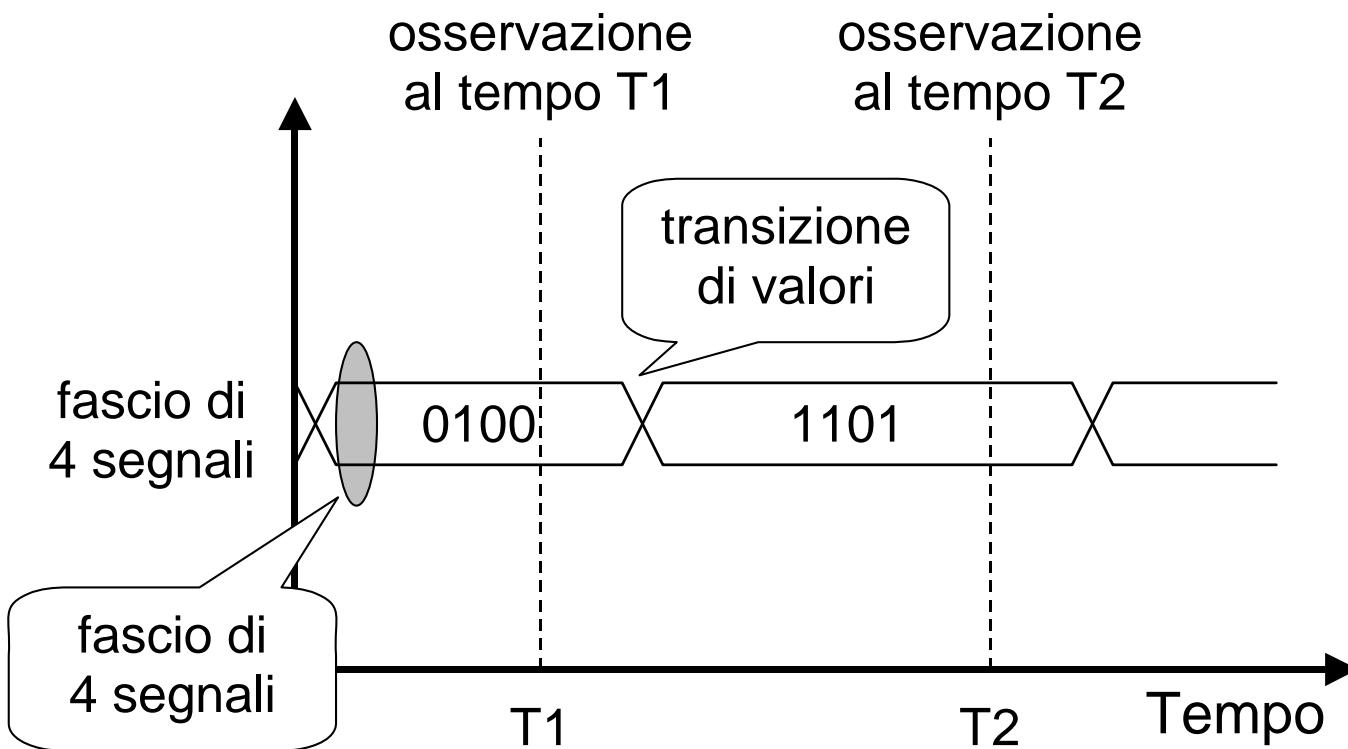
# Libreria di blocchi sequenziali

- Tipici principali componenti sequenziali di libreria:
  - Registro parallelo
  - Registro a scorrimento
  - Banco di registri
  - Memoria
- Ognuno di questi blocchi ammette numerose versioni e varianti



# Ancora diagramma temporale

Come rappresentare un **fascio di segnali**



- Al tempo T1 i 4 segnali valgono 0100, al tempo T2 i 4 segnali valgono 1101

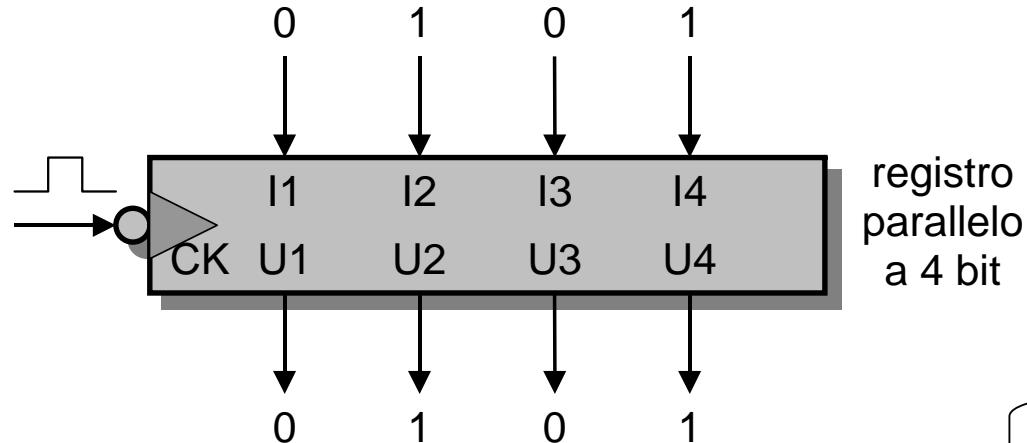


# Registro parallelo

- Il registro parallelo è un vettore di  $n \geq 1$  flip-flop di tipo D. Ha:
  - $n \geq 1$  ingressi  $I_1, \dots, I_n$
  - $n \geq 1$  uscite  $U_1, \dots, U_n$
  - e naturalmente l'ingresso di clock CK
- A ogni ciclo di clock, il registro legge e memorizza nel suo stato la parola di  $n$  bit presente in ingresso, e la presenta sulle  $n$  uscite nel ciclo successivo



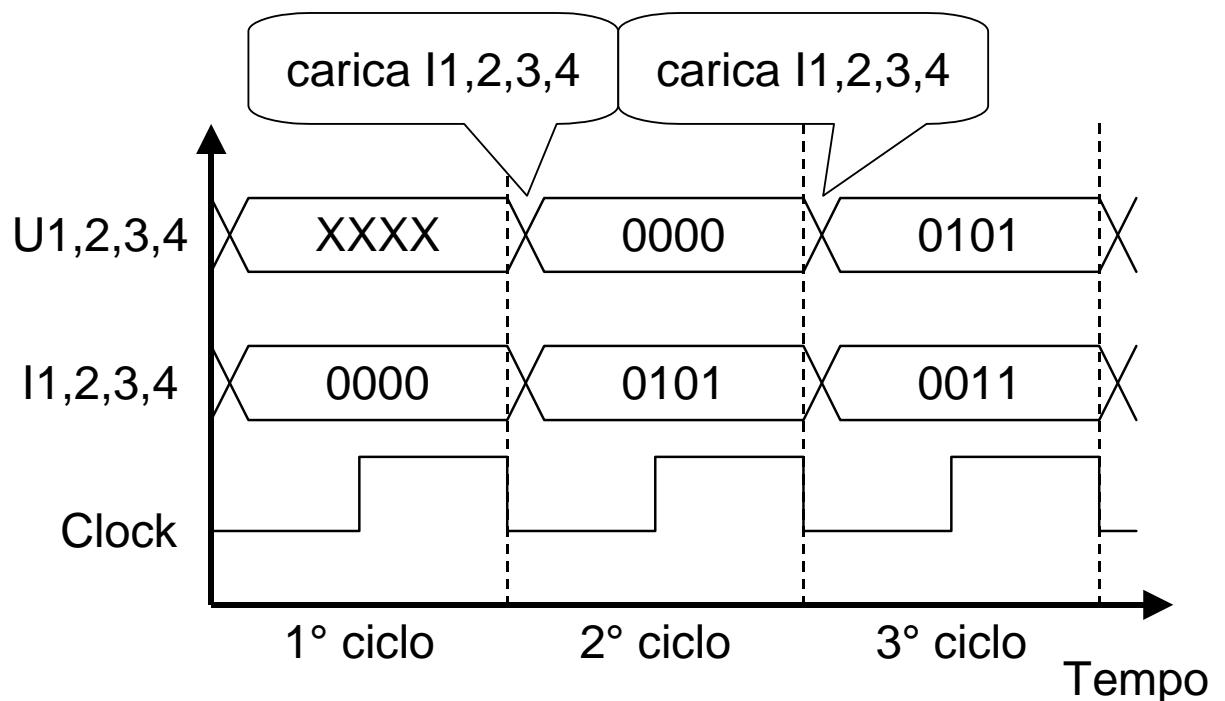
# Simbolo e funzionamento



registro  
parallelo  
a 4 bit

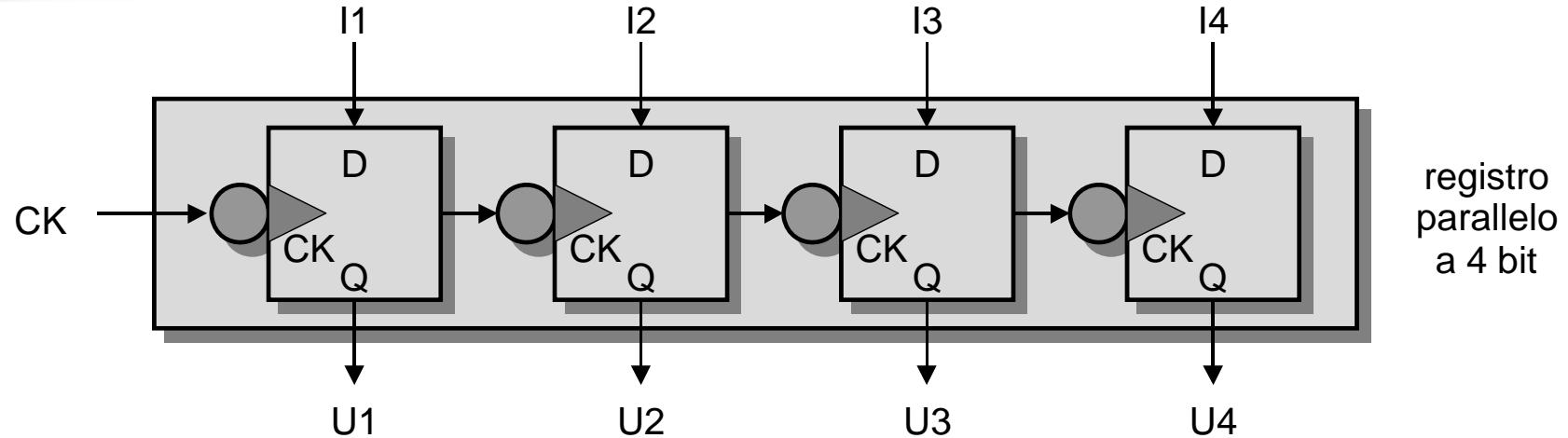
## Diagramma temporale

- carica 0000
- carica 0101
- ecc ...





# Progetto in stile funzionale



Registro parallelo progettato in stile funzionale, usando 4 **flip-flop** D sincroni sul fronte (di discesa)

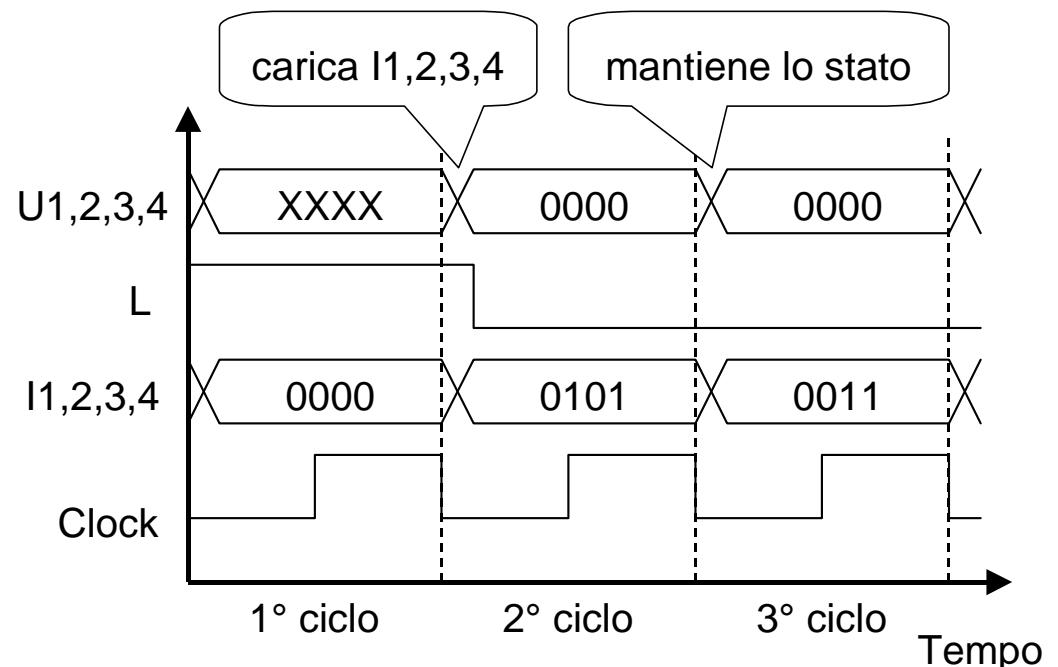
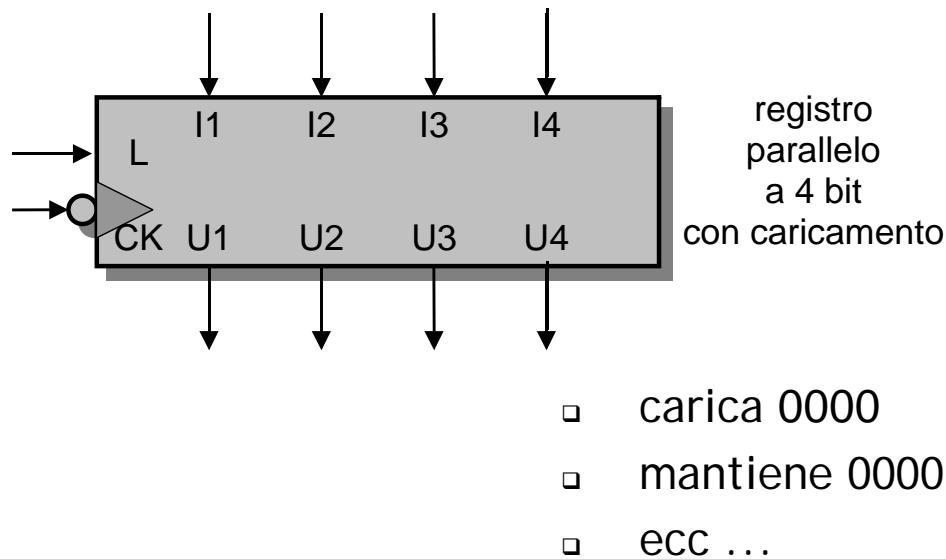
**Nota bene:**

se si usassero dei bistabili D trasparenti (sincronizzati sul livello), durante il livello alto del clock il registro sarebbe esso stesso del tutto trasparente, e dunque non si comporterebbe come un registro ...



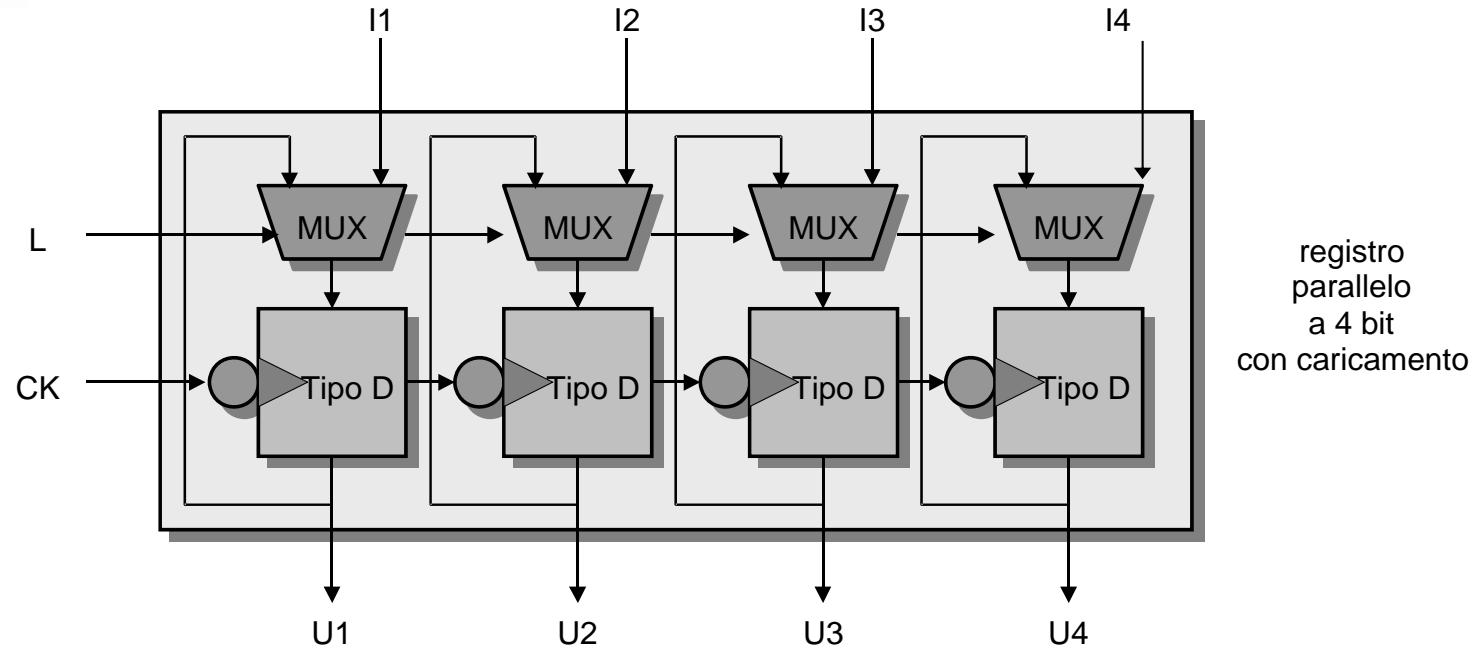
# Registro parallelo con comando di caricamento

- Funziona come il registro parallelo, ma ha in aggiunta un ingresso di comandi di caricamento (L, ingresso di Load):
  - Se il comando L è attivo (p. es.  $L = 1$ ), la parola in ingresso al registro viene memorizzata nel registro stesso e presentata in uscita nel ciclo successivo
  - Altrimenti (cioè  $L = 0$ ), il registro mantiene il suo stato corrente di memorizzazione





# Progetto in stile funzionale



Registro parallelo progettato in stile funzionale, usando 4 flip-flop D sincroni sul fronte (di discesa) e 4 multiplexer a un ingresso di selezione e due ingressi dati



# Varianti e integrazioni

- Registro parallelo con comando di ripristino, per azzerare il contenuto
- Registro parallelo con comando di ripristino e di precarica
- Registro parallelo universale, riunisce le funzioni di tutti i registri precedenti: comandi di caricamento, comando di ripristino e comando di precarica

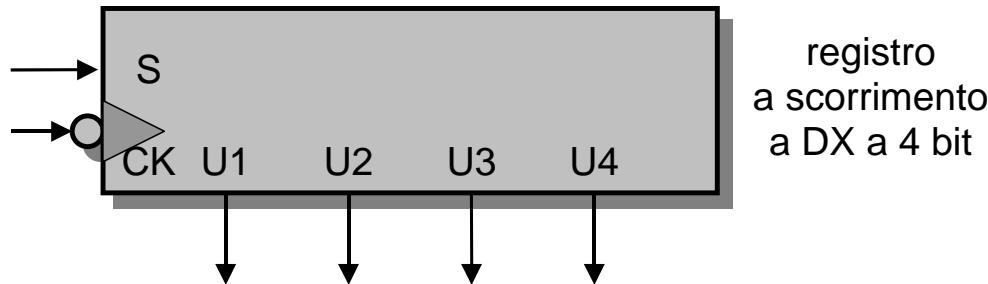


# Registro a scorrimento

- Il registro a scorrimento è una successione di  $n \geq 1$  flip-flop di tipo D collegati in cascata. Ha:
  - un ingresso seriale S
  - $n \geq 1$  uscite parallele  $U_1, \dots, U_n$
  - e naturalmente l'ingresso di clock
- A ogni ciclo di clock, fa scorrere di un bit verso DX la parola memorizzata, perdendo il bit più a DX e aggiungendo a sinistra il bit presente sull'ingresso seriale



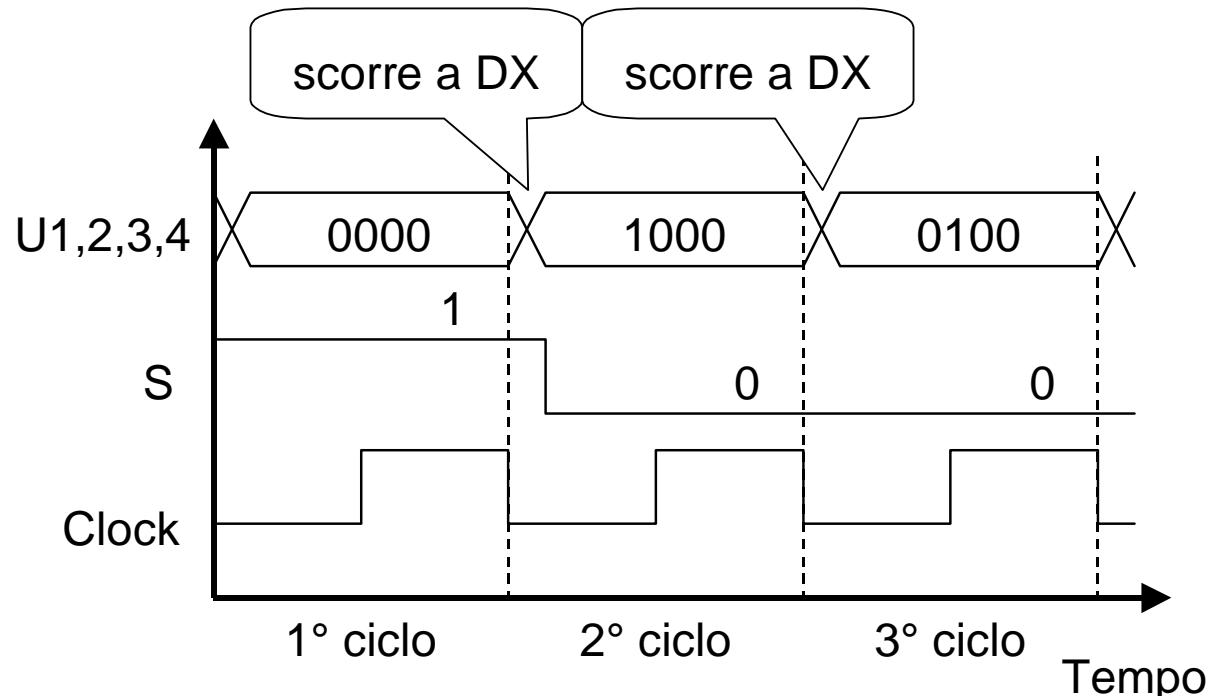
# Simbolo e funzionamento



registro  
a scorrimento  
a DX a 4 bit

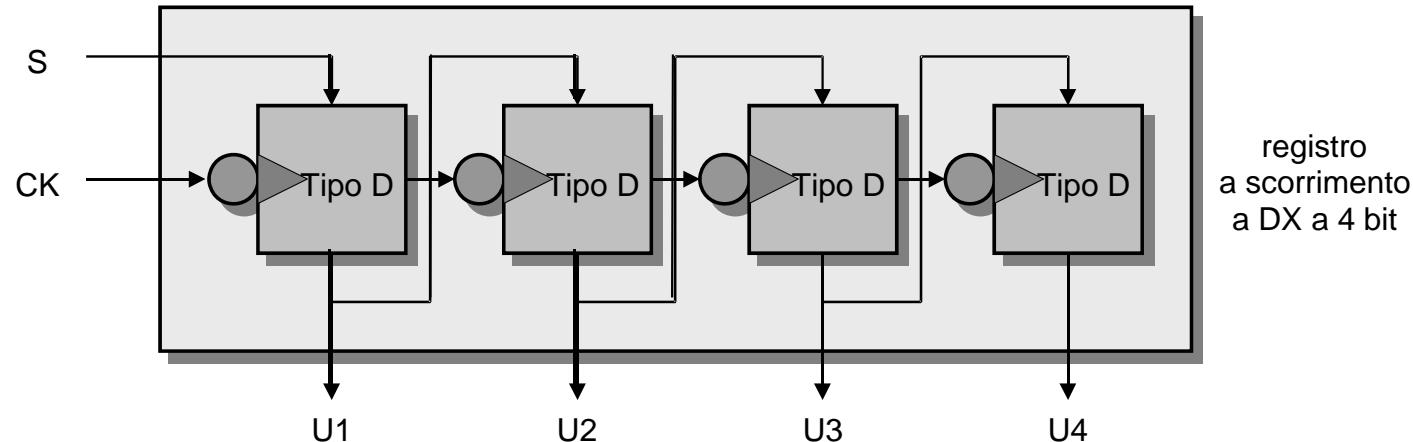
diagramma temporale

- scorre a DX
- scorre a DX
- ecc ...





# Progetto in stile funzionale



Registro a scorrimento a DX progettato in stile funzionale, usando 4 flip-flop D sincroni sul fronte (di discesa) collegati in cascata

**Nota bene:**

se si usassero dei bistabili D trasparenti (sincronizzati sul livello), durante il livello alto del clock un bit potrebbe propagarsi lungo l'intera catena di bistabili ... non sarebbe un comportamento accettabile!



# Varianti e integrazioni

- Registro a scorrimento a SX
- Registro a scorrimento universale: DX e SX (è dotato di un comando di scelta del verso di scorrimento)
- Registro a scorrimento (DX o SX) con funzione di caricamento parallelo
- Registro parallelo / a scorrimento universale: riunisce le funzioni dei registri parallelo e a scorrimento universali
  
- Registro IN seriale / OUT seriale
- Registro IN parallelo / OUT seriale
- Registro IN parallelo/seriale OUT parallelo/seriale



---

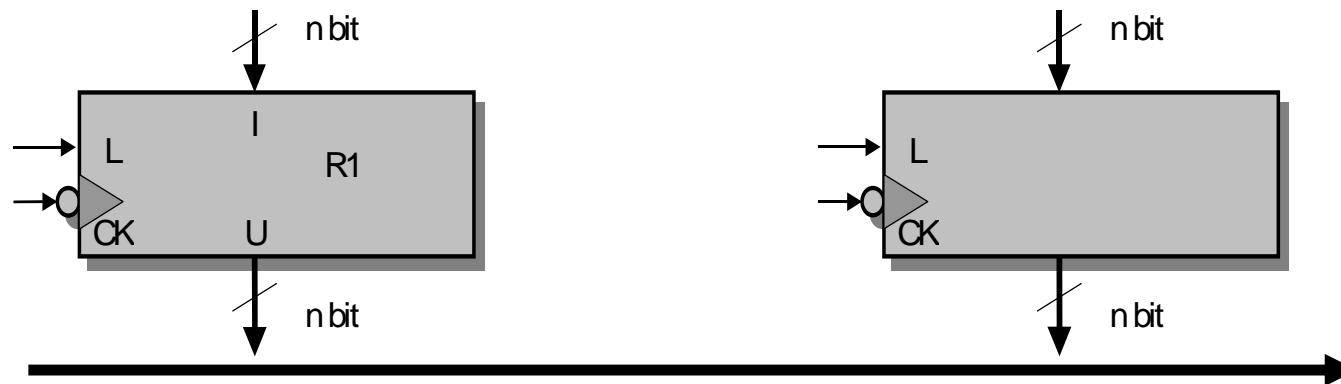
## Register File e memoria

- Componenti di memoria e uscite condivise
  - Register File (banco di registri): struttura e funzionamento
  - Memoria: struttura e funzionamento
  - Banco di memoria
  - Tecnologie di memoria
-



# Linee di uscita condivise

- L'organizzazione interna della memoria e la struttura dei banchi di memoria e dei banchi di registri prevedono generalmente che le **uscite di 2 o più componenti** siano **collegate alle stesse linee di uscita (bus)**
- Sono necessari opportuni “elementi funzionali” (**circuiti di pilotaggio delle uscite** del componente) che garantiscano la **NON** interferenza (dei segnali) tra i moduli che condividono le stesse linee di uscita





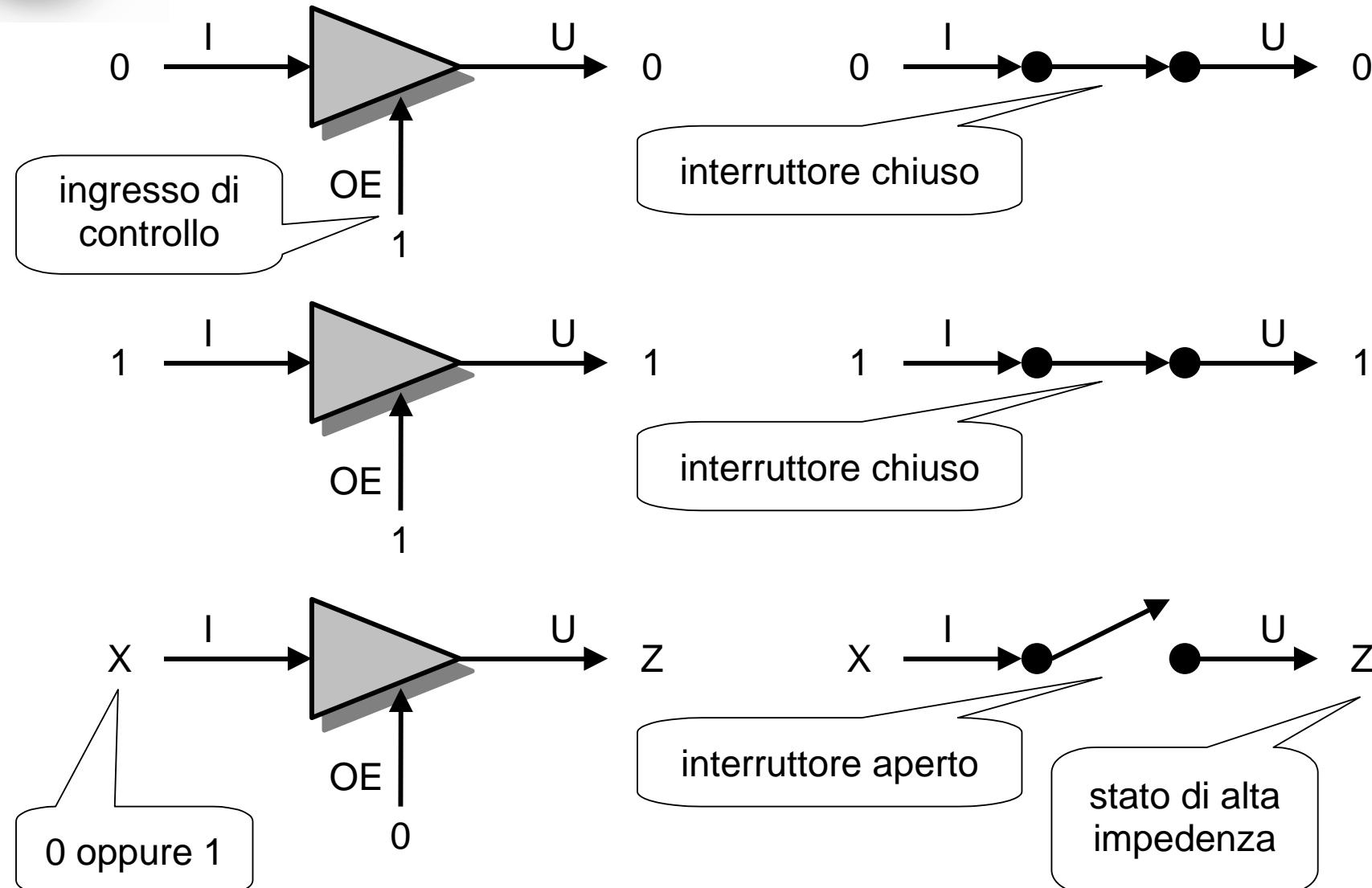
## Circuiti di pilotaggio delle uscite: Buffer tri-state

E' il circuito elementare modellabile come un contatto a tre posizioni:

- in stato di **bassa impedenza** consente di avere in uscita o il **livello alto** (1) o il **livello basso** (0)
- in stato di **alta impedenza** (Z) **isola elettricamente** l'uscita
- l'uscita tri-state viene gestita da un apposito ingresso di controllo (Output Enable) che, se non attivo, forza lo stato di alta impedenza

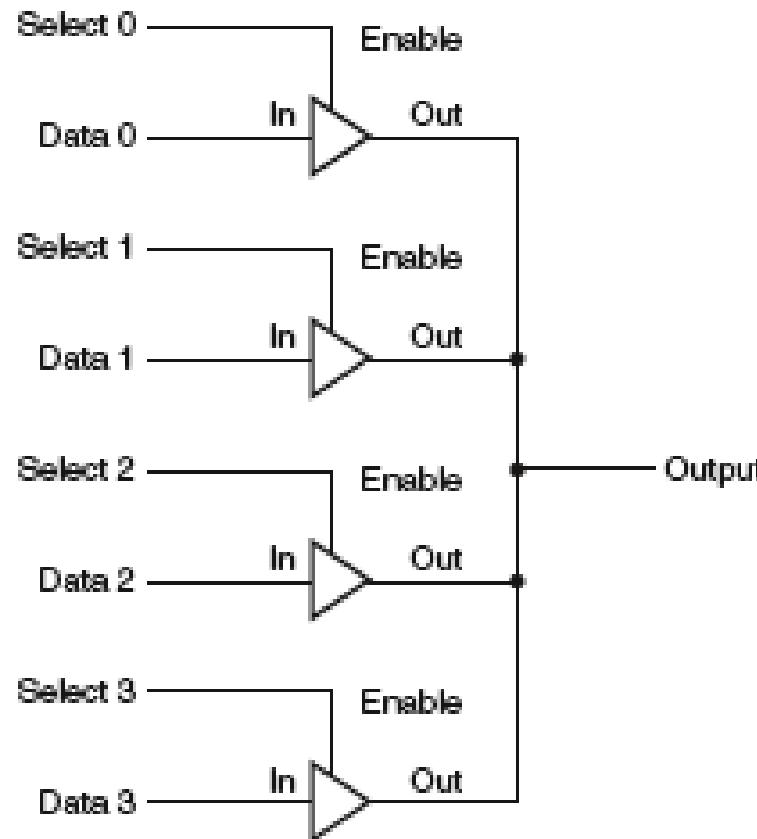


# Funzionamento





# Buffer tri-state: multiplexer sulle linee di uscita

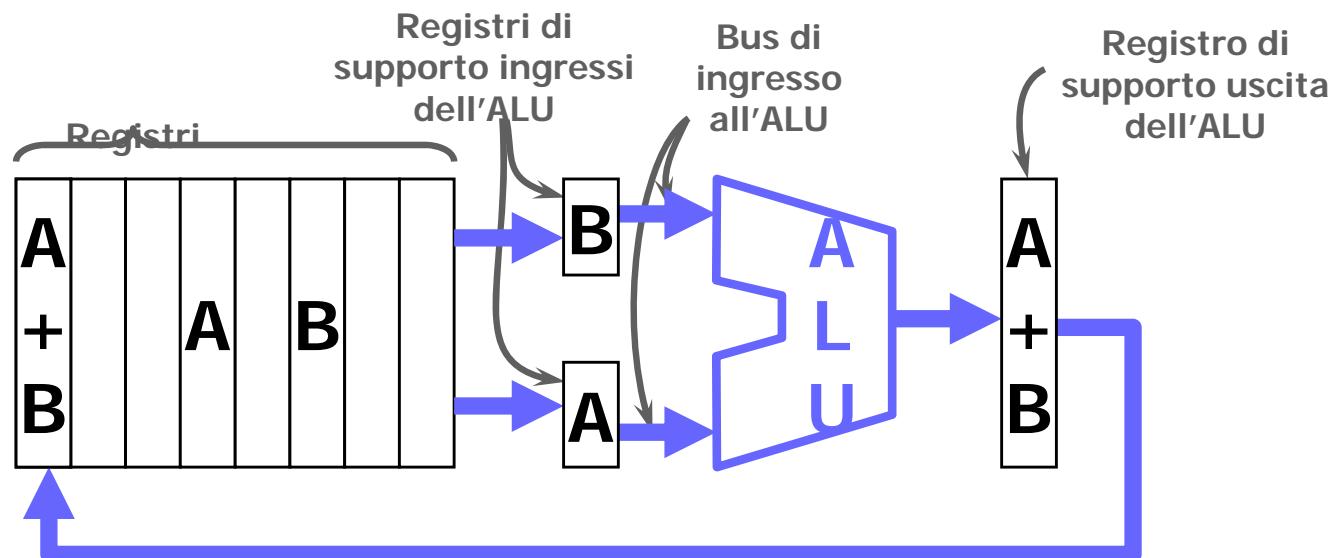


**FIGURE B.9.2** Four three-state buffers are used to form a multiplexor. Only one of the four select inputs can be asserted. A three-state buffer with a deasserted Output enable has a high-impedance output that allows a three-state buffer whose Output enable is asserted to drive the shared output line.



# Banco di registri (register file)

- Spesso occorre utilizzare un certo numero di registri paralleli con funzione di caricamento, tutti aventi le stesse dimensioni e le stesse funzioni (ad es. nel data-path della CPU)



- I registri vengono organizzati in una struttura a vettore, chiamata **banco di registri** o **register file (RF)**



# Banco di registri: caratteristiche

- ❑ Consideriamo come esempio di riferimento un banco di 32 registri da 32 bit ciascuno (Register File del MIPS)
- ❑ Ogni registro è identificato da un **indirizzo** (numero di registro) specificato su 5 bit
  - con  $n \geq 1$  registri occorrono  $\lceil \log_2 n \rceil$  bit
- ❑ Le operazioni eseguibili sul banco sono:
  - **lettura**: si presentano in uscita i 32 bit memorizzati nel registro indirizzato
  - **scrittura**: si memorizzano 32 bit acquisiti in ingresso nel registro indirizzato
- ❑ ***Porta di lettura/scrittura***: insieme di segnali che consentono la lettura e la scrittura dei registri
  - 1 porta di lettura e scrittura
  - 1 porta di lettura e 1 di scrittura
  - 2 porte di lettura e 1 di scrittura (vedi ALU)



# Register File del MIPS

- 2 porte di lettura indipendenti
- 1 porta di scrittura

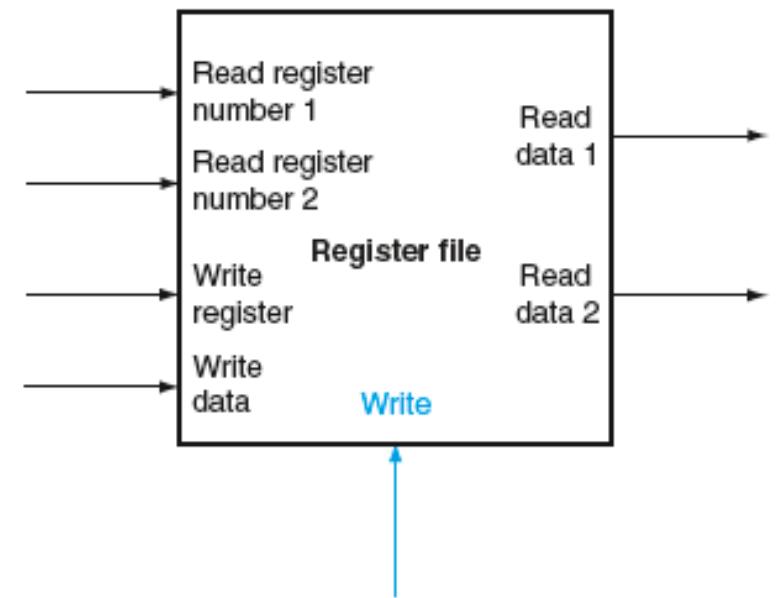
con le opportune temporizzazioni è possibile accedere in parallelo a 3 registri distinti

## *Accesso in lettura:*

- lo stato del registro non viene modificato
- poiché le porte di lettura e scrittura sono distinte e indipendenti non è necessario un comando di lettura esplicito: è sufficiente fornire l'indirizzo dei registri coinvolti e la lettura avviene ogni ciclo di clock
- la realizzazione del banco può richiedere buffer tri-state le uscite devono essere scollegate elettricamente

## *Accesso in scrittura:*

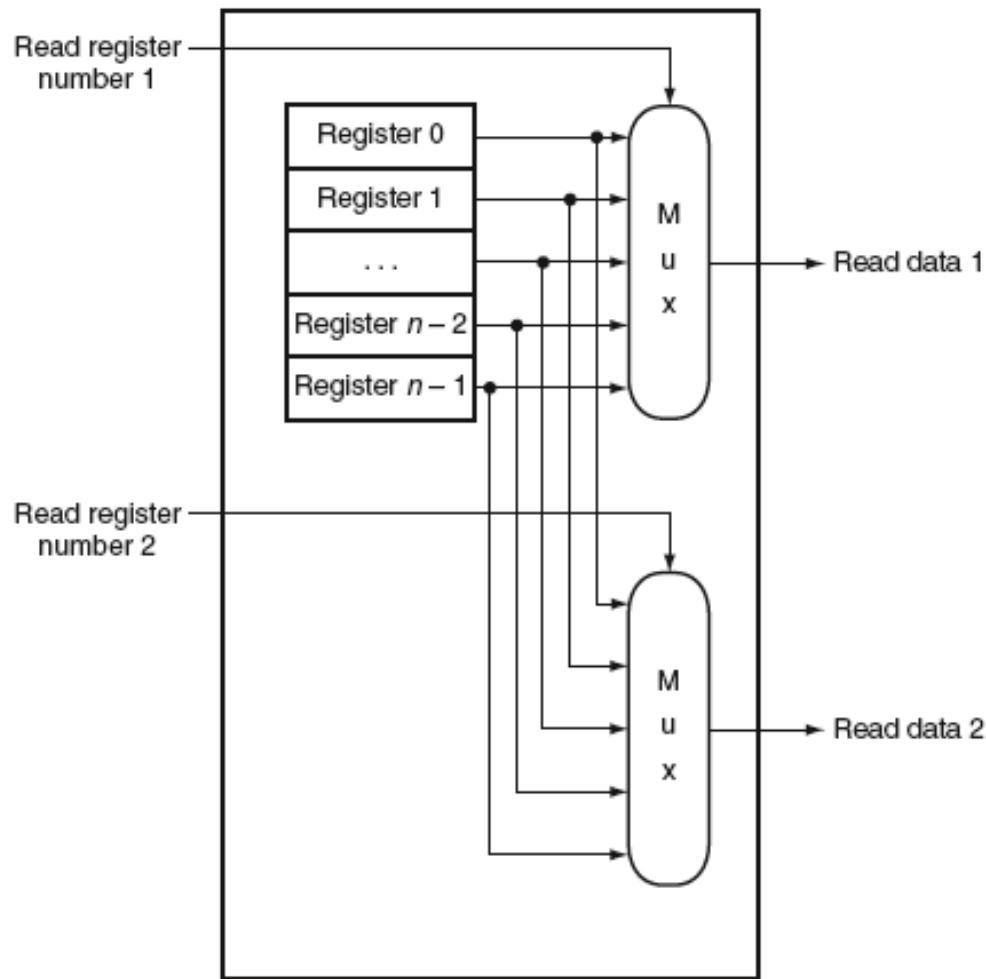
- indirizzo del registro
- dato da scrivere
- segnale di scrittura esplicito (**Write**)



Il segnale di clock è sottointeso



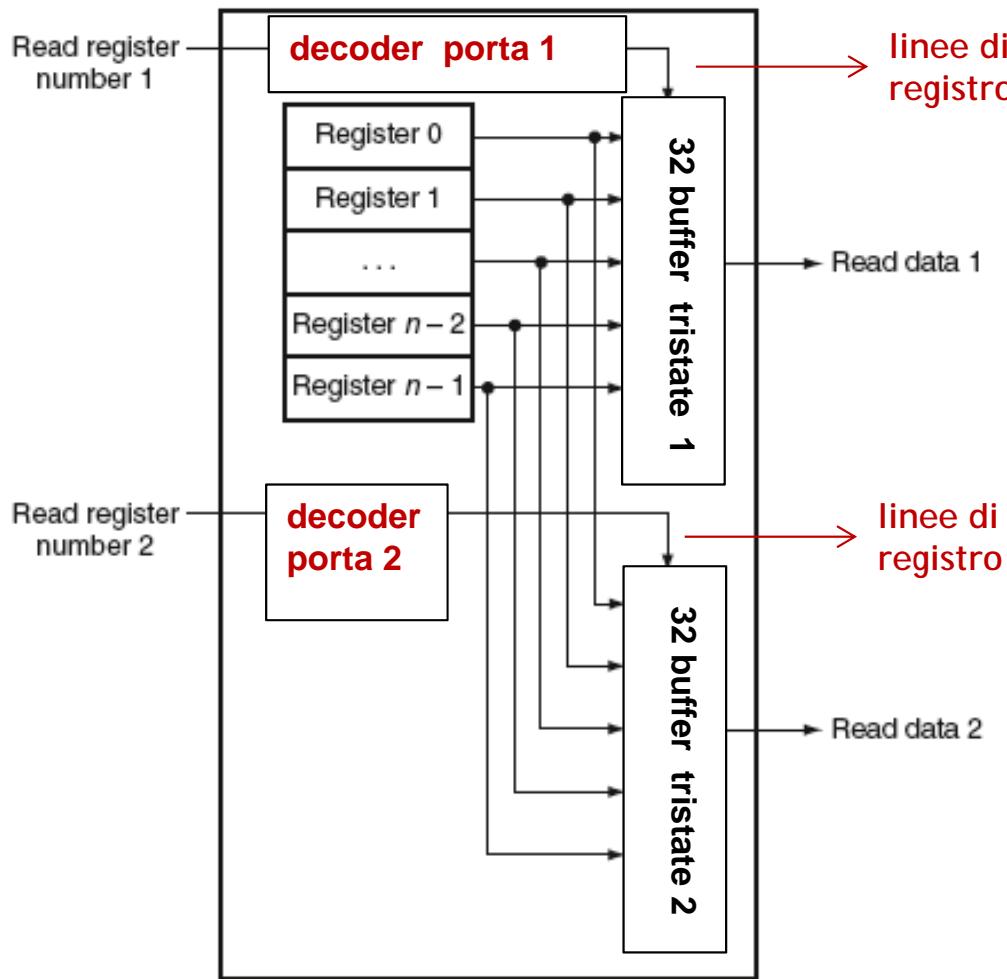
# Register File del MIPS: porte di lettura



- L'indirizzo del registro da leggere è usato come segnale di controllo del multiplexer
- I due multiplexer vengono controllati in modo indipendente



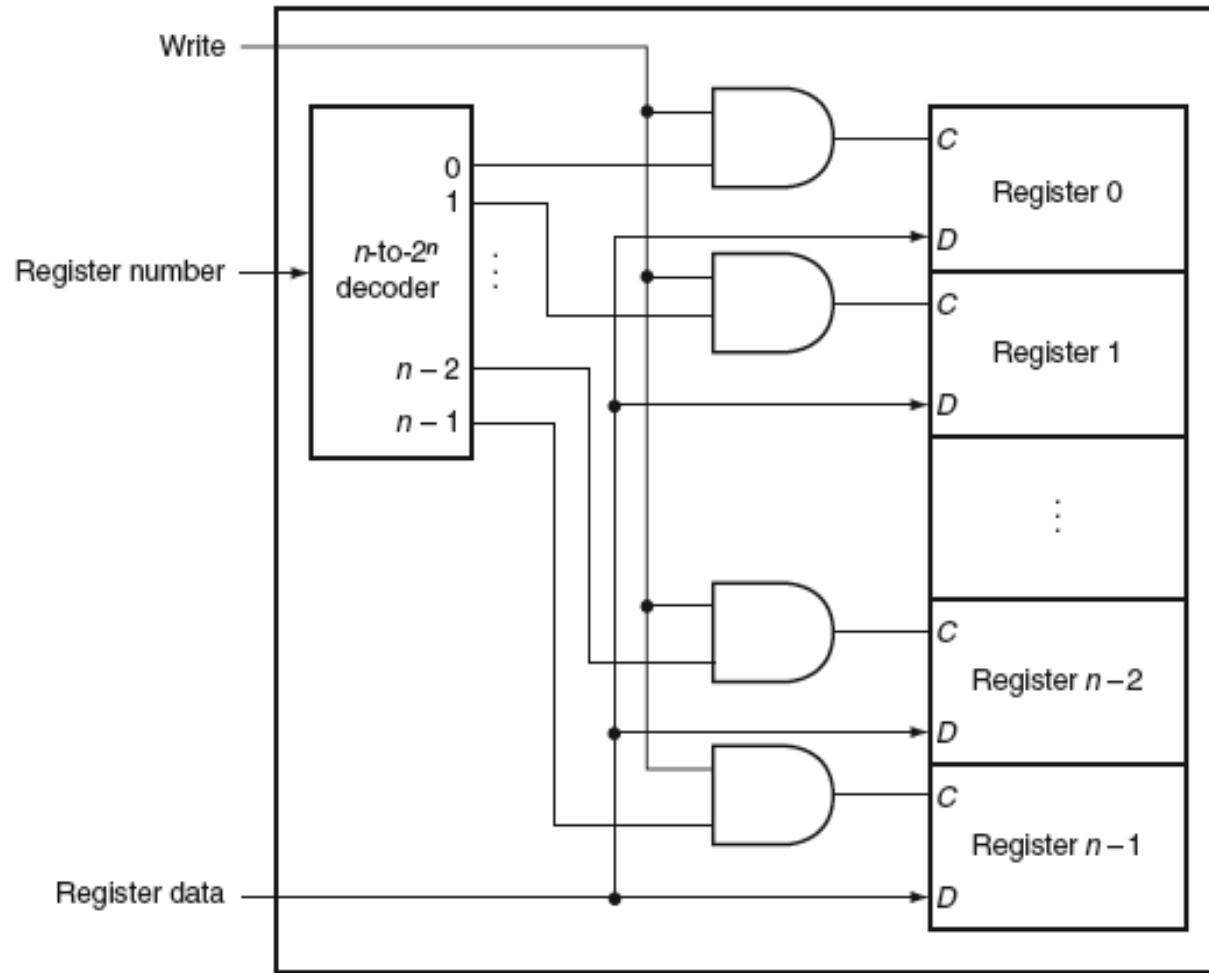
# Register File del MIPS: porte di lettura (un'altra versione)



- Tutte le uscite dei registri condividono le stesse linee di uscita
- L'indirizzo del registro da leggere è usato come segnale di ingresso al decodificatore (5:32)
- Le 32 linee di registro (uscite del decoder) sono attive in mutua esclusione e sono usate come linee di **Enable** per il buffer tristate della porta di lettura considerata
- I decodificatori sono controllati in modo indipendente



# Register File del MIPS: porta di scrittura

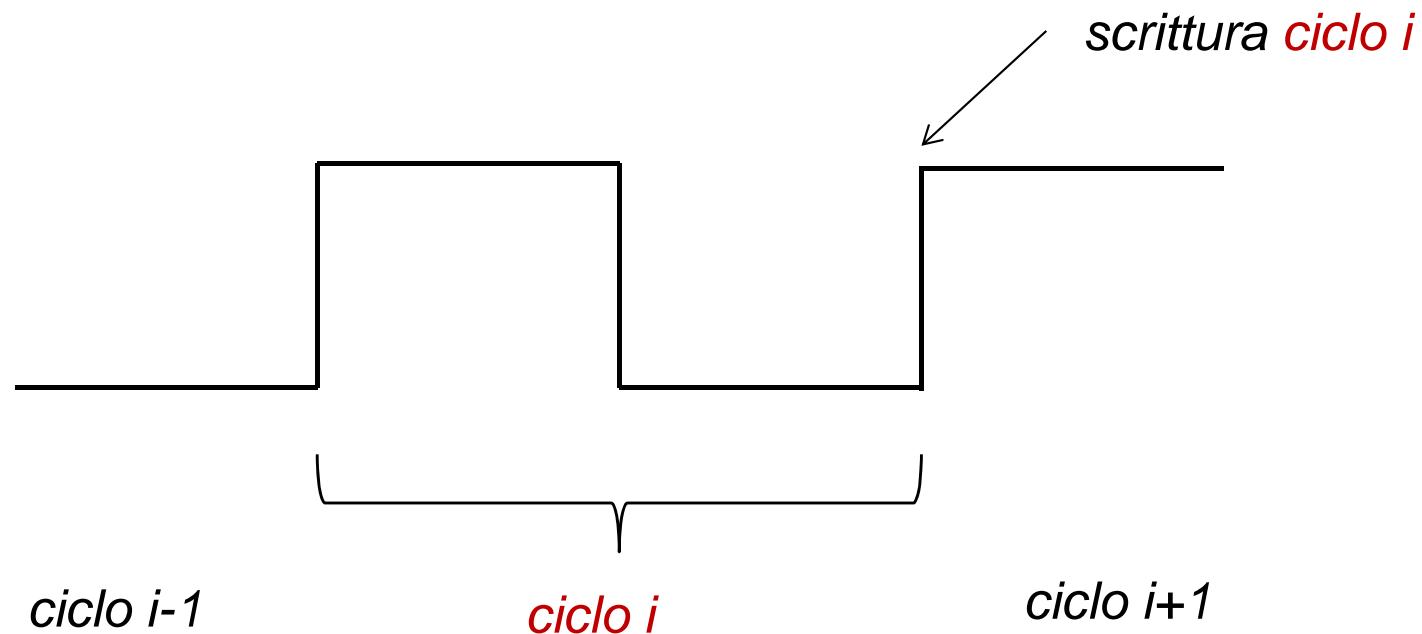


- Un solo registro deve modificare il proprio contenuto
- L'indirizzo del registro da scrivere è fornito come ingresso al decodificatore
- La **linea «di registro»** e il segnale di **write** abilitano un solo registro in scrittura
- Sul fronte attivo del clock (segnale non mostrato) il dato da scrivere **D** viene memorizzato nel registro e si presenta in uscita dopo un opportuno ritardo di propagazione



# Lettura e scrittura di un registro nello stesso ciclo di clock

La scrittura avviene sul fronte di salita del ciclo e quindi la lettura fornisce il valore scritto al ciclo precedente





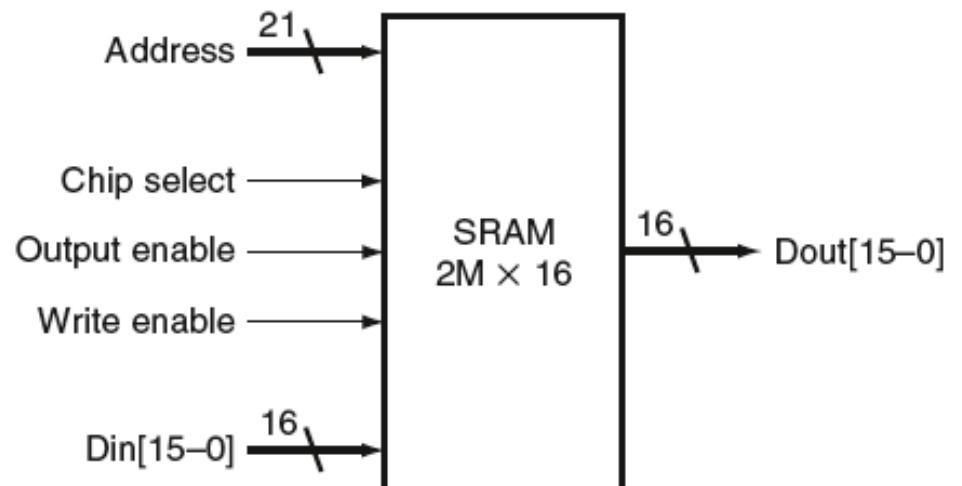
# Memoria

- La memoria è un **blocco funzionale** di tipo sequenziale complesso
  - Mantiene a tempo indefinito se alimentata (SRAM) le informazioni memorizzate e permette l'accesso in lettura o in scrittura
- Ha una **struttura a vettore** (almeno in termini di segnali di accesso esterni) i cui elementi sono le parole di memoria di un certa lunghezza
- Un **componente integrato** (chip) di memoria si caratterizza specificando:
  - la capacità, misurata in numero totale di bit memorizzabili: di solito si esprime come prodotto del numero di parole per il numero di bit per parola
  - le funzioni: lettura e scrittura, solo lettura
  - il numero di porte di accesso
  - e il tempo necessario per l'accesso



# Interfaccia di memoria

- Il contenuto della memoria viene letto o scritto una parola per volta, in un ciclo di clock (più cicli in memorie lente)
- Si accede a una parola di memoria tramite la **porta di accesso alla memoria**
- La porta di accesso alla memoria può funzionare in lettura e scrittura (è il caso più frequente), solo in lettura e teoricamente anche solo in scrittura (caso poco frequente)





# Segnali dell'interfaccia di memoria

- ❑ Gli **ingressi di indirizzo**, che codificano in binario l'indirizzo della parola su cui si deve operare
- ❑ Le **uscite/ingressi di dato**, che servono per leggere/scrivere una parola
- ❑ Per le linee di dato e indirizzo sono da rispettare i tempi di set\_up e hold, quindi questi segnali vengono forniti per primi in modo che siano stabili quando le linee di comando sono attivate a seconda dell'operazione
- ❑ il comando di scrittura, **Write enable**. Se Write enable è un impulso deve avere una durata minima che consenta di soddisfare quanto sopra
- ❑ il comando di abilitazione delle uscite dati, **OE** (output enable):  $OE = 1$  le uscite sono abilitate;  $OE = 0$  le uscite sono isolate
- ❑ il comando di abilitazione del componente, **CS** (chip select):  $CS = 1$  chip attivo, si può accedere al contenuto;  $CS = 0$  chip non attivato



# Cicli di lettura e scrittura

## Ciclo di lettura

- indirizzo della parola da leggere
- comando di lettura (WE a livello 0)
- non isolare le uscite dati (OE = 1)
- abilitare il componente (CS = 1)
- contenuto della parola disponibile sulle uscite. Ritardo di lettura: 8 - 20 ns

## Ciclo di scrittura

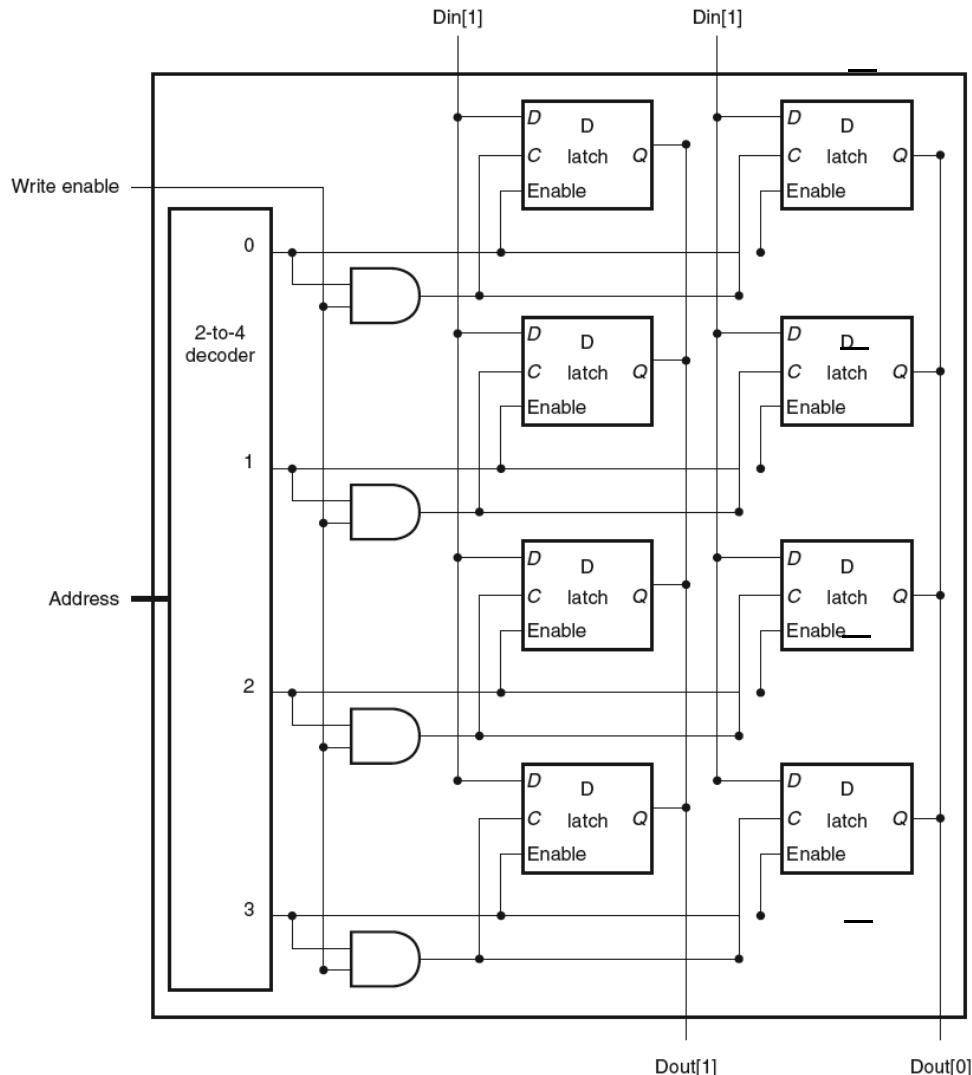
- indirizzo della parola da scrivere
- dato da scrivere in ingresso
- comando di scrittura (WE a livello 1)
- isolare le uscite dati (OE = 0)
- abilitare il componente (CS = 1). Ritardo di scrittura: 8 - 20 ns



# Struttura della memoria

## *organizzazione a matrice di bistabili*

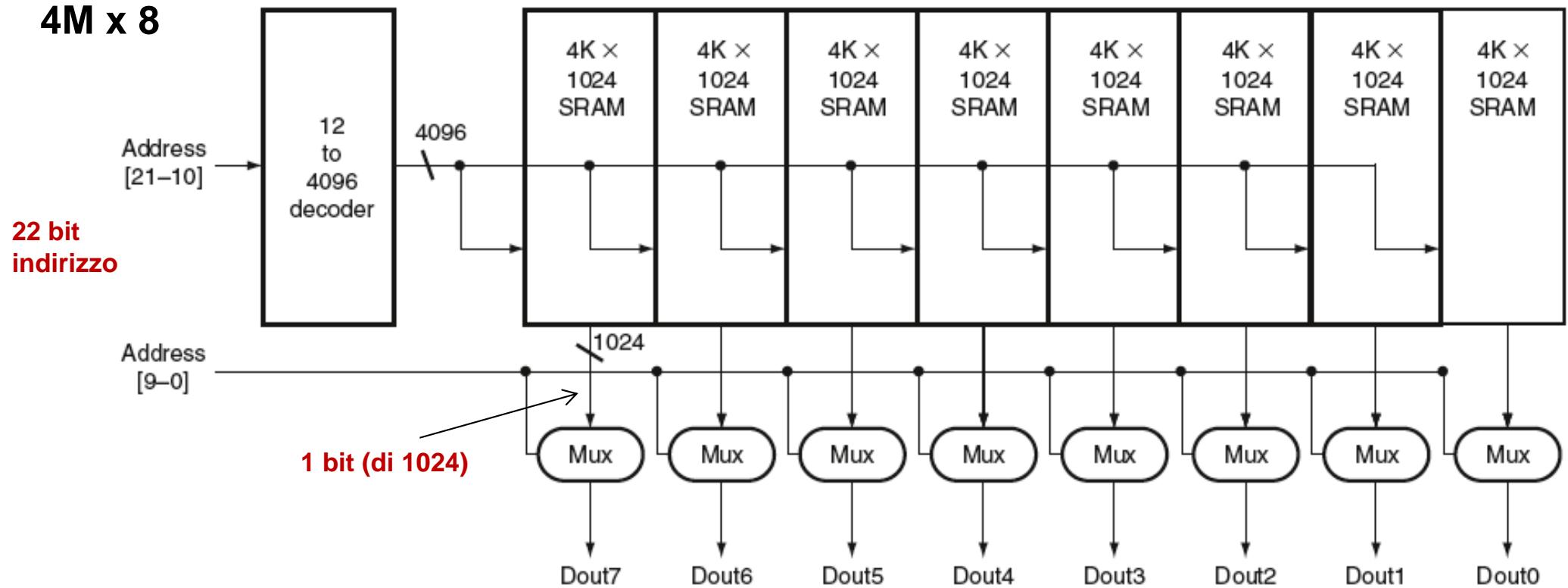
- Matrice di bistabili (righe=parole, colonne = bit della parola), completandola con reti combinatorie di controllo per gestire l'accesso alle parole
- Le uscite del decoder sono le linee di parola
- Ogni bistabile ha la sua uscita in tri-state perché condivisa con gli altri in posizione omologa
- Non adatta a dimensioni significative di memoria (decoder e linee di parola)





## Banco di memoria: organizzazione a matrice di componenti e decodifica dell'indirizzo a due livelli

**4M x 8**



**FIGURE B.9.4 Typical organization of a  $4M \times 8$  SRAM as an array of  $4K \times 1024$  arrays.** The first decoder generates the addresses for eight  $4K \times 1024$  arrays; then a set of multiplexors is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decode that would need either an enormous decoder or a gigantic multiplexor. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.



# DRAM: qualche considerazione

- RAM **dinamiche**: il singolo bit di informazione è «memorizzato» nella carica di un condensatore il cui accesso avviene tramite un transistor che può «leggere» o «scrivere» il suo valore (durata della carica alcuni millisecondi)
- Usano un singolo transistor per cella contro i 4-6 delle SRAM, quindi hanno un costo inferiore per bit e sono più dense. Tempi di accesso maggiori
- E' necessario il *refresh* delle parole di memoria (lettura e riscrittura del contenuto) per la degradazione della carica
- Sono organizzate sempre con decodifica dell'indirizzo a due livelli per diminuire i tempi di refresh
- Le operazioni di refresh consumano meno del 2% dei cicli di memoria

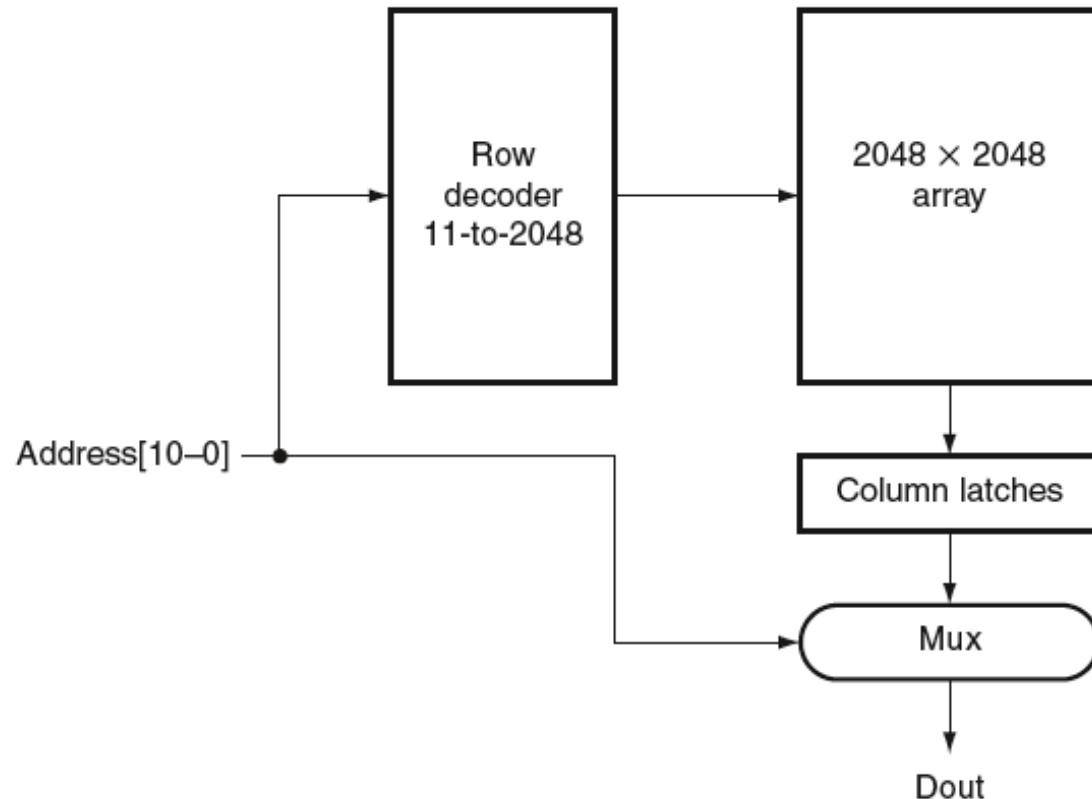


# Struttura di DRAM

Linee indirizzo multiplate:  
segnali RAS e CAS per  
utilizzare correttamente i  
bit indirizzo

Latch di colonna: contiene i  
contenuto di una riga letta

La singola operazione di  
refresh riscrive un'intera  
riga



**FIGURE B.9.6 A 4M × 1 DRAM is built with a 2048 × 2048 array.** The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.



# RAM Statica (SRAM)

- Memoria RAM (Random Access Memory) realizzata con bistabili
- Capacità medio-piccola
- Tempo di accesso molto breve
- Funziona in lettura e scrittura
- Volatile: senza alimentazione il contenuto della memoria svanisce
- Usi: svariati, in particolare come cache
  
- La memoria SRAM consuma parecchi transistor per bit memorizzato (circa 6 transistor per bit)



# RAM Dinamica (DRAM)

- La tecnologia **DRAM** usa circa 1 transistor per bit memorizzato
  - Sfrutta il fenomeno dell'accumulo temporaneo di carica sul transistor (capacità parassita)
  - Internamente contiene un circuito di rinfresco che rigenera le cariche
- Memoria RAM (matrice di transistor) ad altissima densità
- Capacità grande-grandissima
- Tempo di accesso medio
- Funziona in lettura e scrittura
- Volatile: senza alimentazione il contenuto della memoria svanisce
- Usi: numerossissimi, la memoria centrale dei calcolatori normalmente è DRAM



# ROM

- Memoria ROM (Read Only Memory), realizzata come matrice di transistor
- Capacità grande
- Tempo di accesso medio
- Funziona in sola lettura
- Persistente: il contenuto permane anche in assenza di alimentazione
- Usi: per memorizzare programmi permanenti, non modificabili; grandi volumi di produzione



# PROM, EPROM, EEPROM

- Capacità e tempo simili alla ROM
- Sola lettura e persistenti
- Sono programmabili sul campo, tramite un apposito programmatore:
  - PROM: programmabile una volta sola
  - EPROM: cancellabile con raggi UV
  - EEPROM: cancellabile elettricamente (si può anche scrivere un solo byte per volta)
- Usi: piccoli volumi di produzione, prototipi



# Memoria FLASH

- ❑ Capacità e tempo simili alla DRAM (o solo di poco inferiori)
- ❑ Funziona in lettura e scrittura (la scrittura però è a blocchi di byte)
- ❑ Persistente: il contenuto permane anche in assenza di alimentazione
- ❑ Usi: dati multimediali (p. es. immagini statiche, sequenze video), programmi fissi ma periodicamente aggiornabili



# Tabella riassuntiva

Tipo	Categoria	Modalità di cancellazione	Scrittura byte	Volatile	Usi specifici
SRAM	lett/scritt	elettrica	si	si	cache
DRAM	lett/scritt	elettrica	si	si	mem. centrale
ROM	sola lett	nessuna	no	no	grandi vol.
PROM	sola lett*	nessuna	no	no	piccoli vol.
EPROM	sola lett*	luce UV	no	no	prototipi
EEPROM	sola lett*	elettrica	si (lenta)	no	prototipi
FLASH	lett/scritt	elettrica	a blocchi	no	multimedia

\*Le memorie cancellabili vengono talvolta qualificate come “memorie prevalentemente a sola lettura” (read-mostly), invece che “a sola lettura” (read-only)



**POLITECNICO**  
MILANO 1863

# **Architettura dei calcolatori e sistemi operativi**

## **Il processore Capitolo 4 P&H**

**27 . 10. 2015**

# Sommario

Instruction Set di riferimento per il processore  
Esecuzione delle istruzioni  
Struttura del processore



POLITECNICO MILANO 1863

2

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# Caratteristiche principali dell'architettura MIPS

**Architettura RISC (Reduced Instruction Set Computer):**

esegue soltanto istruzioni semplici in un ciclo base ridotto e ha solo 3 formati istruzione diversi (R, I e J)

**Architettura LOAD/STORE:** gli operandi dell'ALU possono provenire soltanto dai registri di uso generale presenti nella CPU e **non** possono provenire direttamente dalla memoria.

Sono necessarie apposite istruzioni di:

- *caricamento (load)* dei dati da memoria ai registri;
- *memorizzazione (store)* dei dati dai registri alla memoria.

**Architettura pipeline:** tecnica per migliorare le prestazioni basata sulla sovrapposizione dell'esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale.



# Processore e set istruzioni di riferimento

Analizzeremo un'implementazione semplificata dal processore MIPS cui è associato il corrispondente set istruzioni

L'***insieme ridotto delle istruzioni*** (istruzioni di riferimento per la realizzazione del processore) fanno parte delle seguenti categorie

- Istruzioni aritmetico-logiche
- Istruzioni di trasferimento da/verso la memoria (*load/store*)
- Istruzioni di salto (condizionato e incondizionato)

Coprono i ***tre formati istruzioni*** (R, I e J) del processore MIPS



# Istruzioni di riferimento

Istruzioni **aritmetico-logiche** sia a soli registri che immediate, esempi:

```
add $s1, $s2, $s3      # $s1 ← $s2 + $s3  
addi $s1, $s1, 4        # $s1 ← $s1 + 4
```

Si potrebbe considerare anche la

```
slt $s1, $s2, $s3      # $s2 < $s3 $s1=1 altrimenti $s1=0
```

Istruzioni di **trasferimento da/verso la memoria** (load/store)

```
lw $s1, offset ($s2)    # $s1 ← M[$s2+offset]  
sw $s1, offset ($s2)    # M[$s2+offset] ← $s1
```

Istruzioni di **salto condizionato** (conditional branch): **beq** (*branch on equal*).

```
beq $s1, $s2, L1      # go to L1 if ($s1 == $s2)  
(bne $s1, $s2, L1      # go to L1 if ($s1 != $s2))
```

Istruzioni di **salto incondizionato** (unconditional jump): **j** (*jump*)

```
j L1                  # go to L1
```



# Formato istruzioni e dimensioni dei campi

I diversi formati (**R**, **I**, **J**) sono riconosciuti tramite il valore del primo campo **codice operativo (opcode)** di 6 bit che indica al processore come trattare i rimanenti bit dell'istruzione

Campo		rs	rt	rd	shamt	funz
Posizione dei bit	31-26	25-21	20-16	15-11	10-6	5-0

## Istruzioni di tipo R

Campo		rs	rt	indirizzo
Posizione dei bit	31-26	25-21	20-16	15-0

## Istruzioni di tipo I

Campo		indirizzo
Posizione dei bit	31-26	25-0

## Istruzioni di tipo J



# Formato istruzioni e significato dei campi (1)

## tipo R: aritmetico-logiche a 3 registri (codice operativo = 0)

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

**rs:** primo registro sorgente

**rt:** secondo registro sorgente

**rd:** registro destinazione

**shamt:** shift amount (scorrimento)

**funct:** indica l'operazione specifica

## tipo I: aritmetico-logiche con immediati

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

**rs:** registro sorgente

**rt:** registro destinazione

**indirizzo:** valore dell'operando **immediato**



# Formato istruzioni e significato dei campi (2)

## tipo I: load/store

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

**rs:** registro base

**rt:** registro destinazione se load, sorgente se store

**indirizzo:** **spiazzamento** da sommare al registro base per ottenere l'indirizzo effettivo di memoria

## I: salti condizionati beq/bne

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

**rs:** primo registro sorgente

**rt:** secondo registro sorgente

**indirizzo:** **spiazzamento di parola** da sommare a (PC + 4) per ottenere l'indirizzo effettivo della destinazione di salto se la condizione è verificata



# Formato istruzioni e significato dei campi (3)

## tipo J: salto incondizionato

op	indirizzo
6 bit	26 bit

**indirizzo** (composto da **26-bit**): è una parte (26 bit su 32) dell'indirizzo **assoluto** di destinazione del salto

I 26-bit del campo **indirizzo** rappresentano un indirizzo di parola (**word address**)

***Questa istruzione verrà considerata più avanti***



# Esecuzione delle istruzioni



POLITECNICO MILANO 1863

10

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# Passi svolti durante l'esecuzione delle istruzioni aritmetico-logiche *di tipo R*

Istruzioni aritmetico-logiche: **op \$x, \$y, \$z**

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$y e \$z	Op. ALU sui Dati Letti (\$y op \$z)	Scrittura nel Reg. Destinazione \$x
----------------------------------	--	--	--

Un'istruzione aritmetico-logica (**tipo R**), ad esempio **add \$x, \$y, \$z** viene eseguita in **4** passi:

- Prelievo istruzione dalla memoria istruzioni e incremento del *PC*.
- Lettura dei 2 registri sorgente (**\$y** e **\$z**) dal banco dei registri, utilizzando i bit [25-21] e [20 -16] per selezionare i registri
- Operazione dell'ALU sui dati letti dal banco dei registri, utilizzando il campo **function** per realizzare la funzione aritmetico-logica.
- Scrittura del risultato dell'ALU nel banco dei registri utilizzando i bit [15-11] dell'istruzione per selezionare il registro destinazione (**\$x**)



# Passi svolti durante l'esecuzione delle istruzioni di *load*

Istruzioni di *load*: **lw \$x,offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registro Base \$y	Op. ALU (\$y+offset)	Prelievo Dato M(\$y+offset)	Scrittura nel Reg. Destinazione \$x
----------------------------------	------------------------------	-------------------------	--------------------------------	--

Un'istruzione di load (tipo I), ad esempio **lw \$x,offset(\$y)** viene eseguita in 5 passi:

- Prelievo istruzione dalla memoria istruzioni e incremento del PC
- Lettura del registro base (\$y) dal banco dei registri, bit [25-21]
- Operazione dell'ALU per calcolare la somma del valore letto dal registro base e dei 16 bit meno significativi dell'istruzione estesi in segno (campo *offset*)
- Prelievo del dato nella memoria dati utilizzando come indirizzo di lettura il risultato dell'ALU
- Scrittura del dato proveniente dalla memoria nel banco dei registri; il registro destinazione (\$x) è indicato dai bit [20-16] dell'istruzione.



# Passi svolti durante l'esecuzione delle istruzioni di store

## Istruzioni di *store*: **sw \$x, offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registri Base \$y & Sorg. \$x	Op. ALU (\$y+offset)	Scrittura Dato M(\$y+offset)
----------------------------------	--	-------------------------	---------------------------------

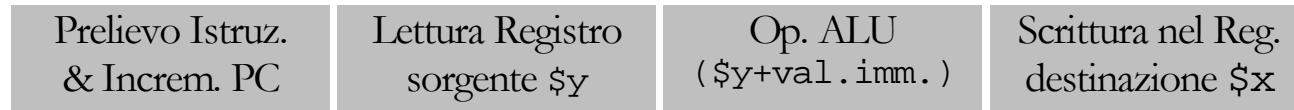
Un'istruzione di store (tipo I), ad esempio **sw \$x, offset(\$y)** viene eseguita in 4 passi:

- Prelievo istruzione dalla memoria istruzioni e incremento del PC
- Lettura del registro base (\$y), bit [25-21], e del registro sorgente (\$x) dal banco dei registri; il registro sorgente è indicato dai bit [20-16] dell'istruzione
- Operazione dell'ALU per calcolare la somma del valore letto dal registro base e dei 16 bit meno significativi dell'istruzione estesi in segno (**offset**)
- Scrittura del dato proveniente dal registro sorgente (\$x) nella memoria dati utilizzando come indirizzo di scrittura il risultato dell'ALU



# Passi svolti durante l'esecuzione dell' istruzione **addi**

Istruzione di *somma con immediato*: addi \$x,\$y, 22



Questa istruzione viene eseguita in 4 passi:

- Prelievo istruzione dalla memoria istruzioni e incremento del PC
- Lettura del registro sorgente (\$y) dal banco dei registri, bit [25-21]
- Operazione dell'ALU per calcolare la somma del valore letto dal registro sorgente e dei 16 bit meno significativi dell'istruzione estesi in segno (campo **immediato**)
- Scrittura del risultato dell'ALU nel banco dei registri; il registro destinazione (\$x) è indicato dai bit [20-16] dell'istruzione.



# Passi svolti durante l'esecuzione delle istruzioni di salto condizionato

## Istruzioni di salto condizionato: **beq \$x, \$y, offset**

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$x e \$y	Op. ALU (\$x-\$y) & (PC+4+offset)	Scrittura nel PC
----------------------------------	--	--------------------------------------	---------------------

Un'istruzione di salto condizionato (tipo I), ad esempio **beq \$x, \$y, offset** viene eseguita in 4 passi:

- Prelievo istruzione dalla memoria istruzioni e incremento del PC
- Lettura dei 2 registri sorgente (\$x e \$y) dal banco dei registri, bit [25-21] e [20 -16]
- Operazione dell'ALU per effettuare la sottrazione tra i valori letti dal banco dei registri. Il valore (PC+4) viene sommato ai 16 bit meno significativi dell'istruzione estesi in segno (offset); il risultato è l'indirizzo di destinazione del salto (Branch Target Address)
- L'uscita Zero dell'ALU viene utilizzata per decidere quale valore debba essere memorizzato nel PC: (PC+4) oppure (PC+4+offset)



# Passi svolti durante l'esecuzione delle istruzioni

Istruzioni aritmetico-logiche: **op \$x,\$y,\$z**

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$y e \$z	Op. ALU sui Dati Letti (\$y op \$z)	Scrittura nel Reg. Destinazione \$x
----------------------------------	--	--	--

Istruzioni di caricamento (*load*): **lw \$x,offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registro Base \$y	Op. ALU (\$y+offset)	Prelievo Dato M(\$y+offset)	Scrittura nel Reg. Destinazione \$x
----------------------------------	------------------------------	-------------------------	--------------------------------	--

Istruzioni di memorizzazione (*store*): **sw \$x,offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registri Base \$y & Sorg. \$x	Op. ALU (\$y+offset)	Scrittura Dato M(\$y+offset)
----------------------------------	--	-------------------------	---------------------------------

Istruzioni di salto condizionato: **beq \$x,\$y,offset**

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$x e \$y	Op. ALU (\$x-\$y) & (PC+4+offset)	Scrittura nel PC
----------------------------------	--	--------------------------------------	---------------------



POLITECNICO MILANO 1863

This document is available free of charge on

**StuDocu.com**

Distributing prohibited | Downloaded by Alexander Sarti (amk.sarti@gmail.com)

# Esecuzione delle istruzioni

Per ogni tipo di istruzione i primi 2 passi da eseguire sono identici:

- Inviare il contenuto del *Program Counter (PC)* ad una memoria che contiene il codice per prelevare l'istruzione (*Memoria Istruzioni*).
- Leggere uno o due registri dal banco dei registri utilizzando i campi dell'istruzione per selezionare i registri ai quali accedere

Dopo questi 2 passi, le azioni necessarie per concludere l'esecuzione dell'istruzione dipendono dal tipo di istruzione (codice operativo), sebbene tutte le istruzioni utilizzino l'*ALU* dopo la lettura dei registri:

- Le istruzioni aritmetico-logiche per l'esecuzione dell'operazione
- Le istruzioni di riferimento a memoria per il calcolo dell'indirizzo effettivo;
- I salti condizionati per valutare l'esito dei confronti.



# Esecuzione delle istruzioni (cont.)

Dopo aver utilizzato l'ALU, le azioni richieste per completare le varie istruzioni si differenziano ulteriormente:

- Le istruzioni aritmetico-logiche devono scrivere nel registro destinazione il risultato dell'ALU
- Le istruzioni di *load* richiedono l'accesso in lettura alla *Memoria Dati* ed eseguono il caricamento del dato letto nel registro destinazione
- Le istruzioni di *store* richiedono l'accesso in scrittura alla *Memoria Dati* ed eseguono la memorizzazione del dato proveniente dal registro sorgente
- Le istruzioni di salto condizionato, possono cambiare l'indirizzo dell'istruzione successiva in base al risultato del confronto.



# Unità funzionali richieste durante l'esecuzione in alcune istruzioni

Tipo di istruzione	Unità funzionali utilizzate
<b>Tipo R</b>	Memoria istruzioni Banco registri ALU Banco registri
<b>Caricamento parola</b>	Memoria istruzioni Banco registri ALU Memoria dati Banco registri
<b>Memorizzazione parola</b>	Memoria istruzioni Banco registri ALU Memoria dati
<b>Salto condizionato</b>	Memoria istruzioni Banco registri ALU
<b>Salto</b>	Memoria istruzioni



# Tempo di esecuzione delle istruzioni

Tipo di 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

- Nota: se il processore deve effettuare ogni istruzione in un solo ciclo di clock, la durata del ciclo non deve essere inferiore a *800 ps*, perché si deve tenere conto dell'istruzione più lenta



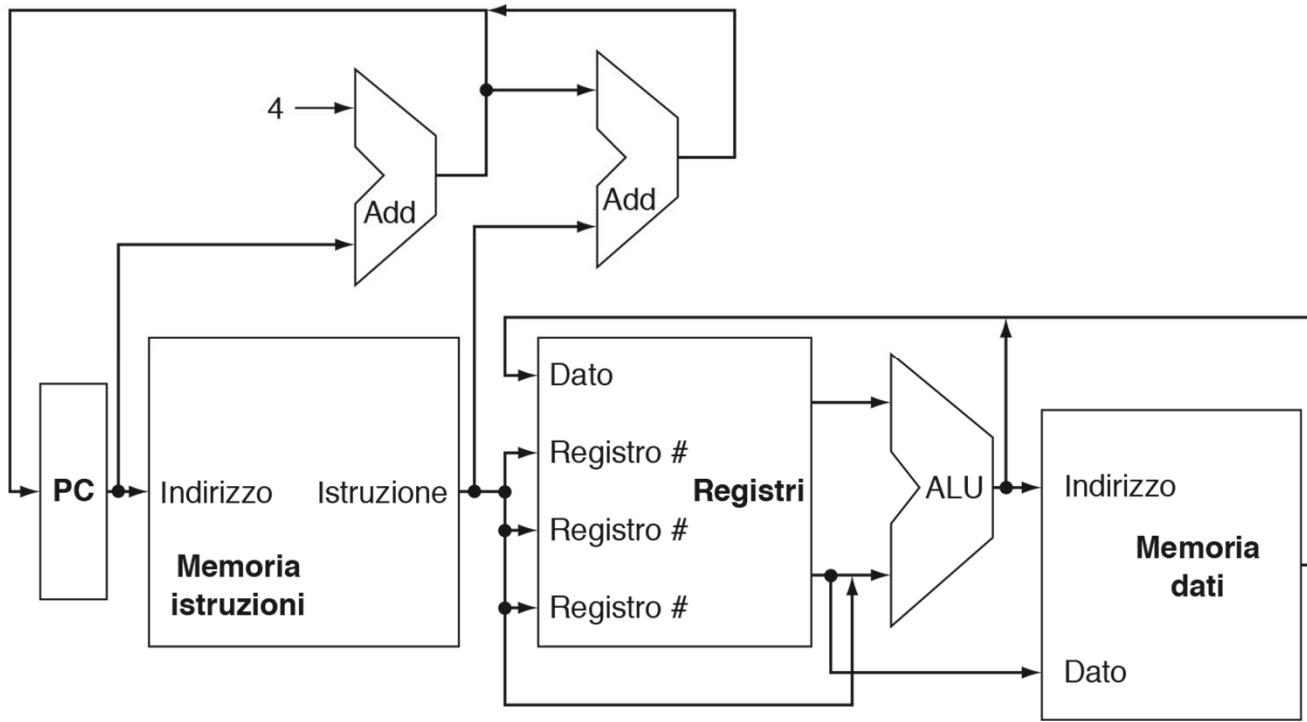
# Architettura del processore



POLITECNICO MILANO 1863

21

# Struttura base del processore MIPS



- *Memoria Istruzioni (MI)* (usata solo in lettura) separata dalla *Memoria Dati (MD)*
- I 32 registri del processore sono organizzati *Register File - RF* con *due* porte di lettura e *una* porta in scrittura.
  - I campi dell'istruzione indicano direttamente i registri che debbono essere utilizzati come operandi dell'istruzione e vengono perciò collegati agli ingressi del Register File



## Struttura base del processore MIPS (cont.)

Il *Register File* ha 4 ingressi e 2 uscite, che realizzano due porte di lettura e una porta di scrittura:

- 3 ingressi sono collegati ai campi dell'istruzione che specificano i registri sorgente o base e il registro destinazione (2 per porte in lettura e 1 per porta in scrittura)
- 1 ingresso è per i dati che possono essere scritti nel registro destinazione (per la porta di scrittura)
- 2 uscite del banco dei registri riportano il contenuto dei 2 registri letti (per le porte di lettura)

Gli operandi dell'**ALU** sono utilizzati per:

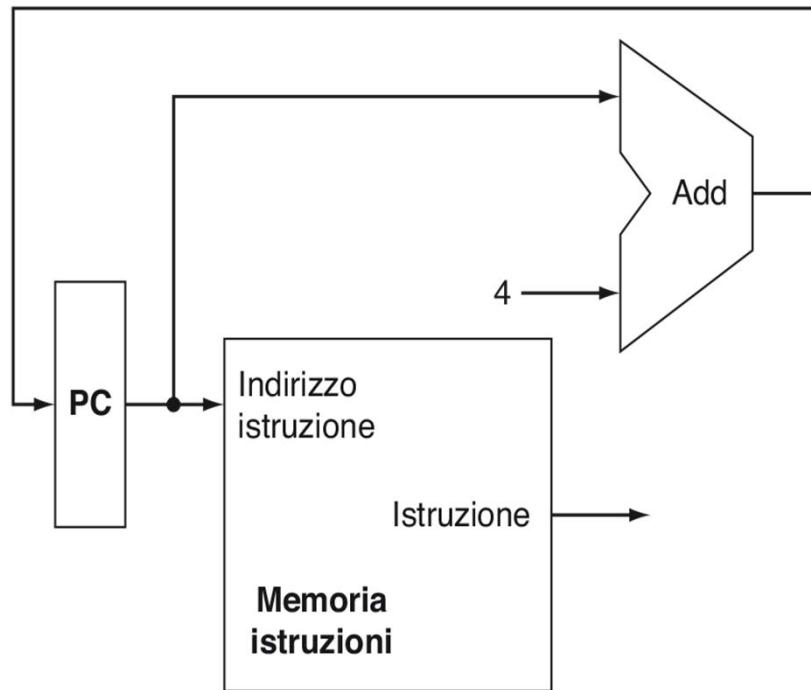
- Calcolare un risultato aritmetico (per un'istruzione aritmetico-logica);
- Calcolare un indirizzo di memoria (per load/store);
- Effettuare un confronto (per un salto condizionato).

Il risultato dell'ALU è utilizzato per:

- Scrittura in un registro destinazione (se l'istruzione è aritmetico-logica)
- Essere usato come indirizzo di lettura/scrittura della Memoria Dati (se l'istruzione è load/store)
- Calcolare l'indirizzo della prossima istruzione, secondo un'apposita logica di controllo (se l'istruzione è di salto condizionato).



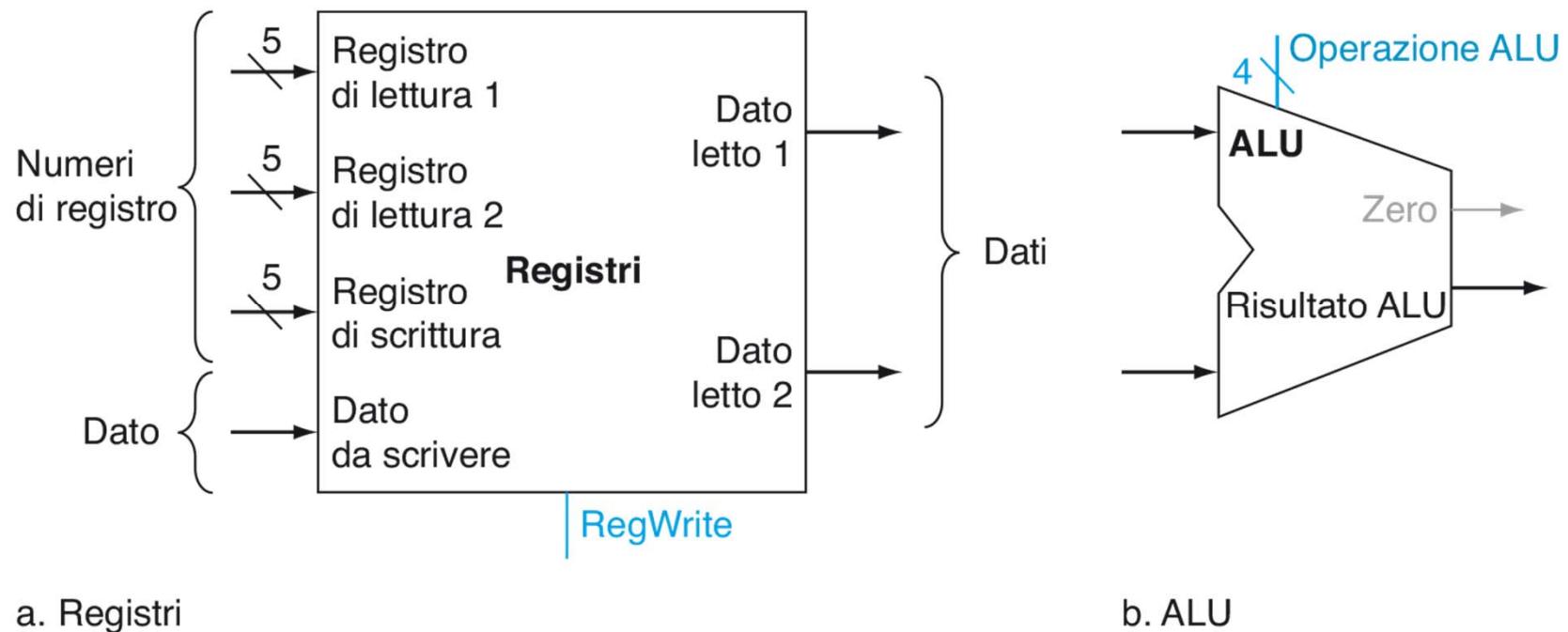
# Esecuzione della fase di caricamento (*fetch*) delle istruzioni



- *Memoria Istruzioni*
- *Program Counter (PC)* per memorizzare l'indirizzo dell'istruzione corrente;
- Un *Sommatore (Add\_seq)* per incrementare il *PC* in modo da poter indirizzare l'istruzione successiva ( $PC + 4$ ).



# Esecuzione delle istruzioni aritmetico-logiche



- *Register File*
- *ALU* a 32 bit che riceve 2 ingressi da 32 bit e che restituisce un risultato da 32 bit.



## Esecuzione delle istruzioni aritmetico-logiche (cont.)

Le istruzioni aritmetico-logiche hanno come operandi **3 registri**: per ogni istruzione si devono leggere due parole di dati dal banco di registri e se ne deve scrivere una

Per ogni parola letta dai registri sorgente sono necessari un ingresso (*5 bit*) al banco di registri, per specificare il numero del registro che si vuole leggere, ed un'uscita dal banco (*32 bit*) per il valore letto dal registro

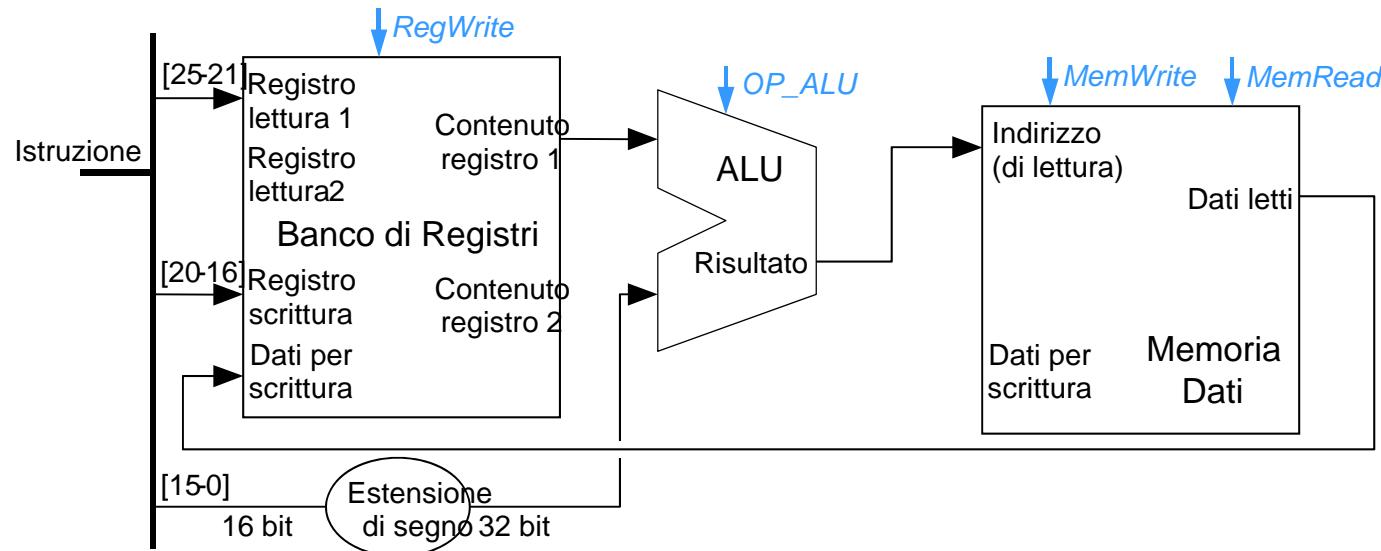
Per scrivere una parola nel registro destinazione sono necessari due ingressi: un ingresso (*5 bit*) per specificare il numero del registro in cui si vuole scrivere ed un ingresso (*32 bit*) per il dato da scrivere

→ Il banco di registri fornisce sempre in uscita il contenuto dei registri di lettura, mentre le scritture sono controllate da un apposito segnale di controllo della scrittura (*RegWrite*)

Il segnale di controllo *OP\_ALU* provvede a specificare all'ALU il tipo di operazione.



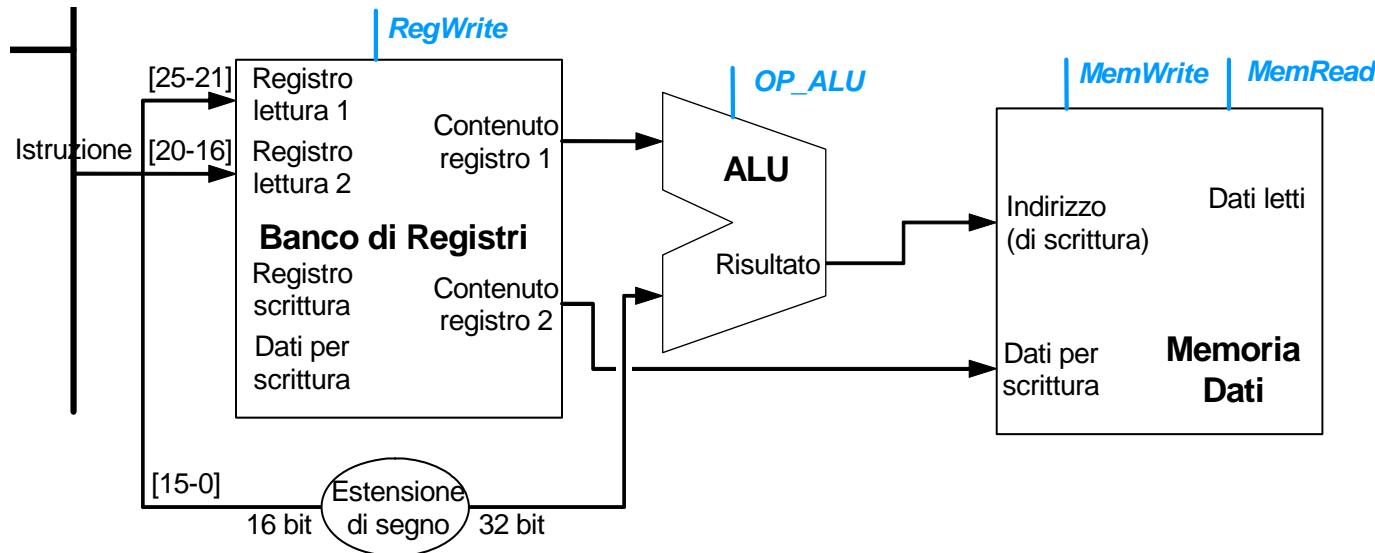
# Esecuzione delle istruzioni di *load*



- *Register File*
- ALU a 32 bit per calcolare l'indirizzo
- *Memoria Dati* da cui leggere;
- Unità per estendere con il segno corretto il valore dello spiazzamento (*offset*) dai 16 bit contenuti nell'istruzione ad un valore con segno a 32 bit.



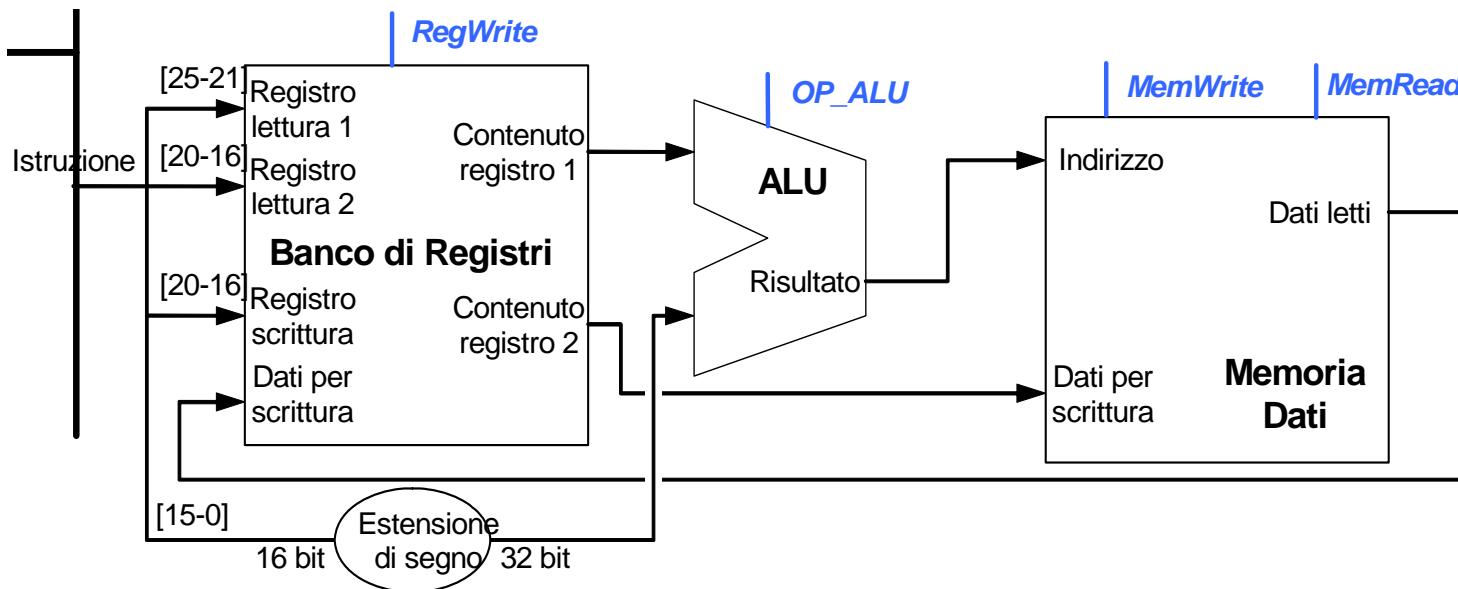
# Esecuzione delle istruzioni di store



- *Register File*
- *ALU* a 32 bit per calcolare l'indirizzo
- Una *Memoria Dati*
- Unità per estendere con il segno corretto il valore dello spiazzamento (*offset*) dai 16 bit contenuti nell'istruzione ad un valore con segno a 32 bit



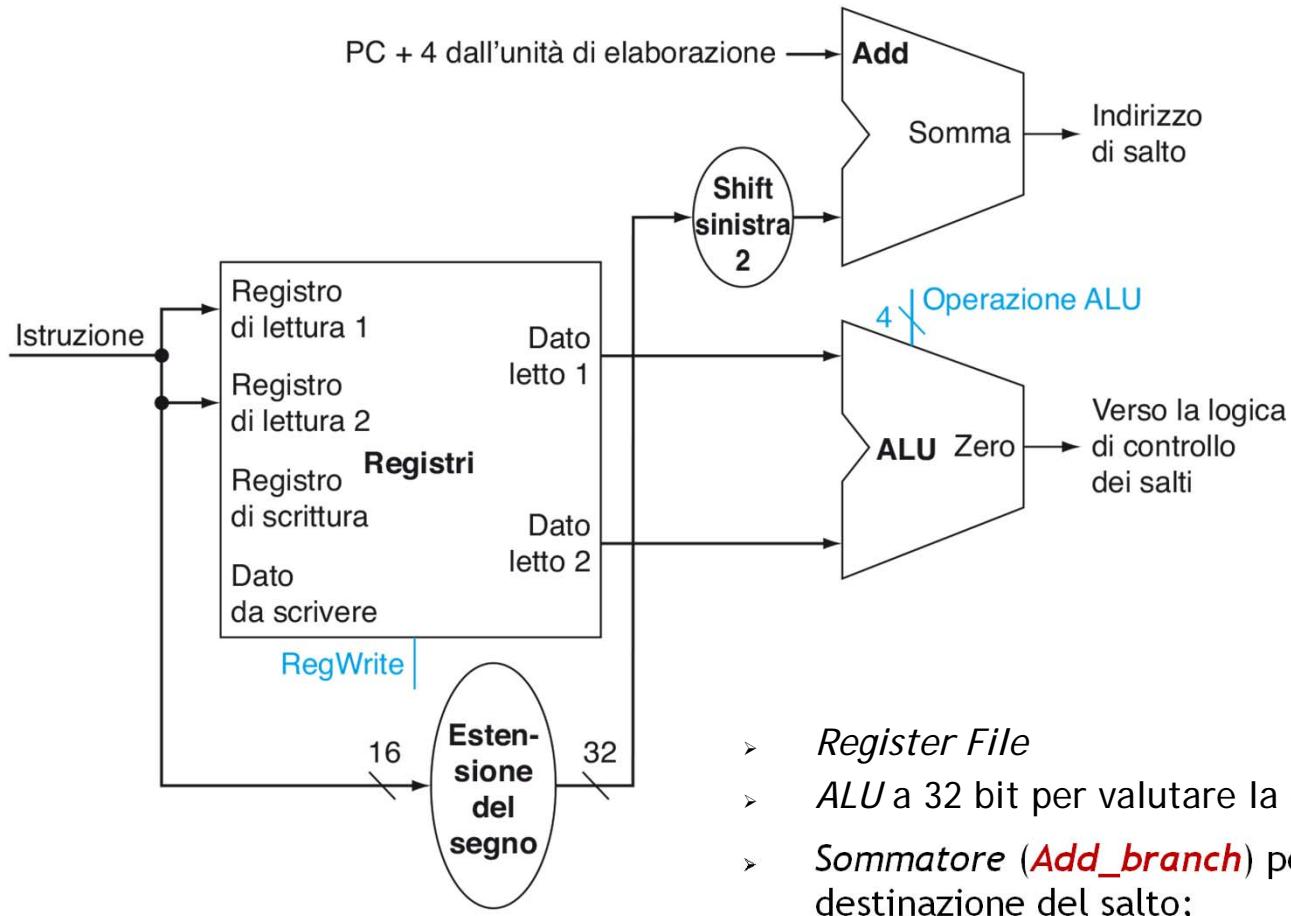
# Esecuzione delle istruzioni di *load/store*



- Il valore dello spiazzamento (*offset*), dopo l'estensione di segno, è utilizzato come secondo operando dell'*ALU*;
- Ogni operazione è costituita da lettura del RF, calcolo nell'*ALU* di un indirizzo di lettura/scrittura per accedere alla *Memoria Dati*, lettura/scrittura della *Memoria Dati* e, nel caso di *load*, scrittura del RF.
- Durante la *load*, il valore letto dalla *Memoria Dati* deve essere scritto nel registro destinazione del RF. Durante la *store*, il valore da scrivere nella *Memoria Dati* deve essere letto dal registro sorgente del RF.



# Esecuzione delle istruzioni di salto condizionato



- *Register File*
- *ALU* a 32 bit per valutare la condizione di salto
- *Sommatore (Add\_branch)* per calcolare l'indirizzo di destinazione del salto;
- Unità per l'*estensione del segno* e uno *shifter* a sinistra di 2 bit;
- Logica di controllo per decidere, in base al valore dell'uscita *Zero* dell'*ALU*, il nuovo valore del *PC*.



## Esecuzione delle istruzioni di salto condizionato (cont.)

La base per il calcolo dell'indirizzo di destinazione di un salto condizionato è l'indirizzo dell'istruzione che segue quella di salto ( $PC + 4$ ).

Poiché ogni istruzione occupa 4 byte, prima di effettuare la somma dello spiazzamento (*offset*) con il contenuto di ( $PC+4$ ), bisogna **moltiplicare per 4** il valore contenuto nell'istruzione  
⇒ **scorrimento a sinistra di 2 bit.**

L'operazione di salto condizionato è costituita da due operazioni:

- Calcolo dell'indirizzo di destinazione del salto attraverso un sommatore che effettua la somma tra il ( $PC+4$ ) e il valore contenuto nell'istruzione dopo avere esteso il segno e fatto scorrere a sinistra di 2 bit;
- Confronto nell'ALU del contenuto dei registri operandi letti dal *RF*.  
Se il segnale di uscita Zero dell'ALU è asserito ⇒ la condizione di salto è verificata e l'indirizzo di destinazione del salto diventa il nuovo *PC*.  
Se invece la condizione non è verificata ⇒ il *PC* incrementato sostituisce il *PC* attuale.



## Realizzazione del processore completo

Esaminati gli elementi richiesti da ogni tipo di operazione, è possibile combinarli in un'unica unità di elaborazione.

Si assume che tutte le istruzioni siano eseguite in un solo ciclo di clock

- Nessuna risorsa può essere utilizzata più di una volta per istruzione
- Qualsiasi risorsa di cui si ha bisogno più di una volta deve essere duplicata

Occorre quindi una *Memoria Istruzioni distinta dalla Memoria Dati*.



## Realizzazione del processore completo (cont.)

Alcune unità funzionali potrebbero essere **duplicate** nel momento in cui si combinano le varie unità di calcolo definite nella precedente sezione, mentre altre unità possono essere **condivise** da differenti flussi di istruzioni

Per condividere un elemento tra due diversi tipi di istruzione, si deve introdurre un **multiplexer** di dati per permettere connessioni multiple all'ingresso di un elemento e selezionare uno tra i vari ingressi in base alla configurazione delle linee di controllo.



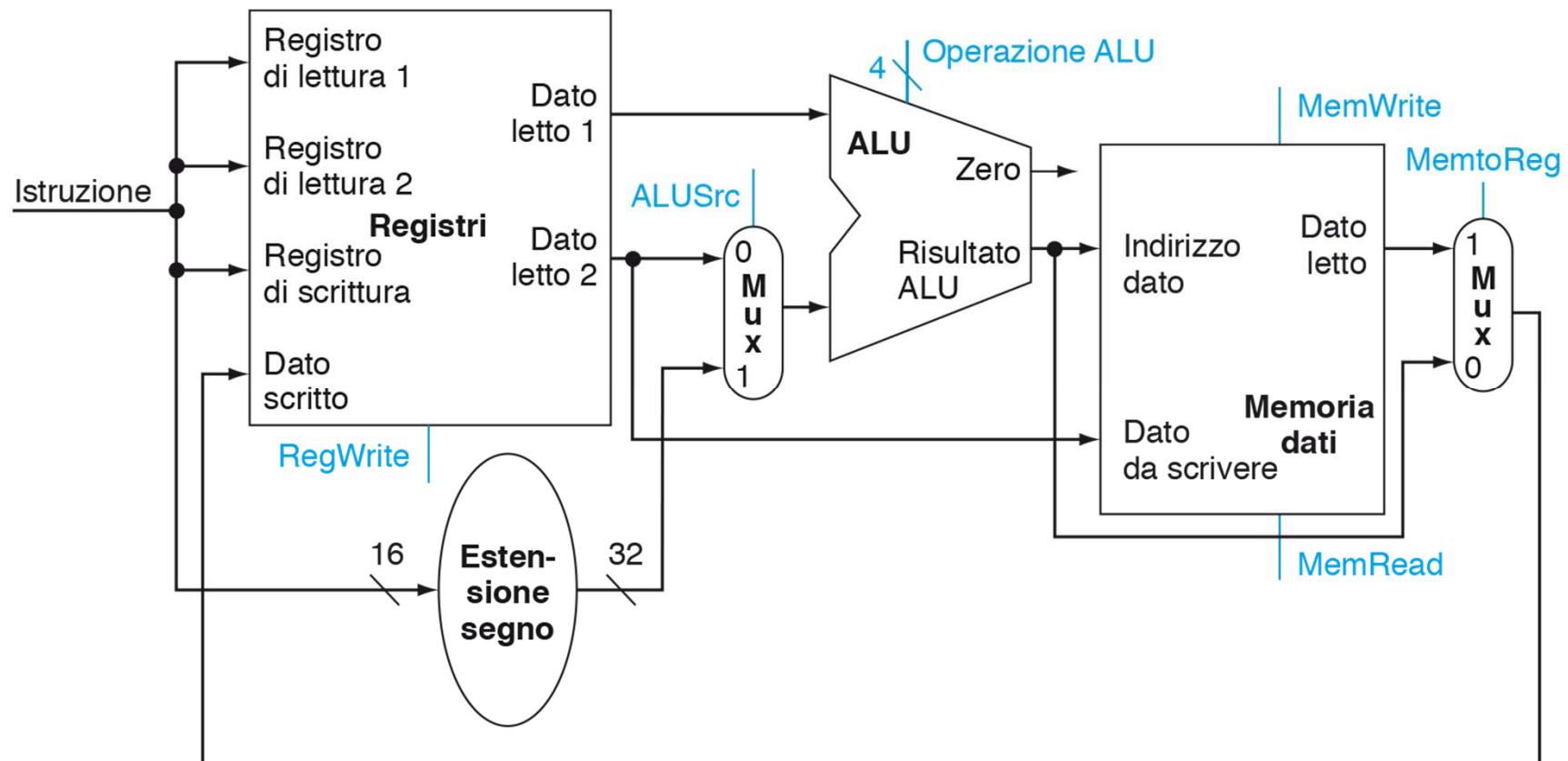
# Esecuzione delle istruzioni aritmetico-logiche e *load/store*

Da risolvere:

1. Il secondo ingresso dell'ALU è il contenuto di un registro (istruzione di tipo R) oppure la metà meno significativa dell'istruzione (istruzione di *load/store* o *aritmetiche immediate*)  
⇒ **MUX al secondo ingresso dell'ALU (*Mux A*)**
2. Il valore scritto nel registro destinazione proviene dal risultato dell'ALU (istruzione tipo R o aritmetica immediata) oppure dalla Memoria Dati (istruzione di *load*)  
⇒ **MUX all'ingresso dei Dati per Scrittura del RF (*Mux C*)**
3. Il numero del registro in cui si vuole scrivere il risultato è indicato da diversi campi (i bit [15-11] per le istruzioni di tipo R e bit [20-16] per istruzioni di *load/store* e *aritmetiche immediate*)  
⇒ **MUX all'ingresso del Registro Scrittura del RF (*Mux D*)**



# Esecuzione delle istruzioni aritmetico-logiche e *load/store* (cont.)

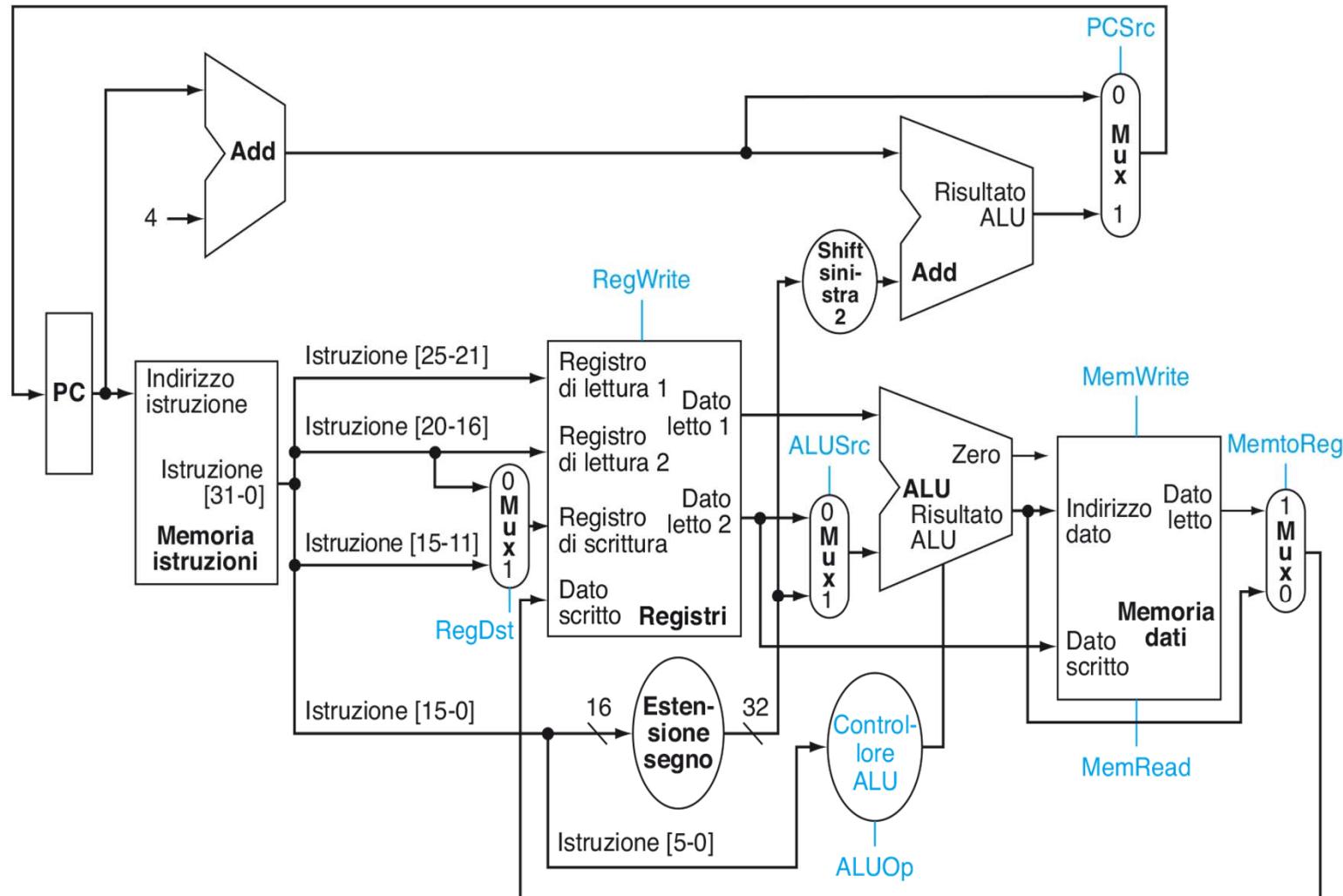


Non ancora risolto il «numero» del registro di scrittura

- istruzione aritmetica di tipo R, bit [15-11]
- istruzione di store, bit [20 -16]



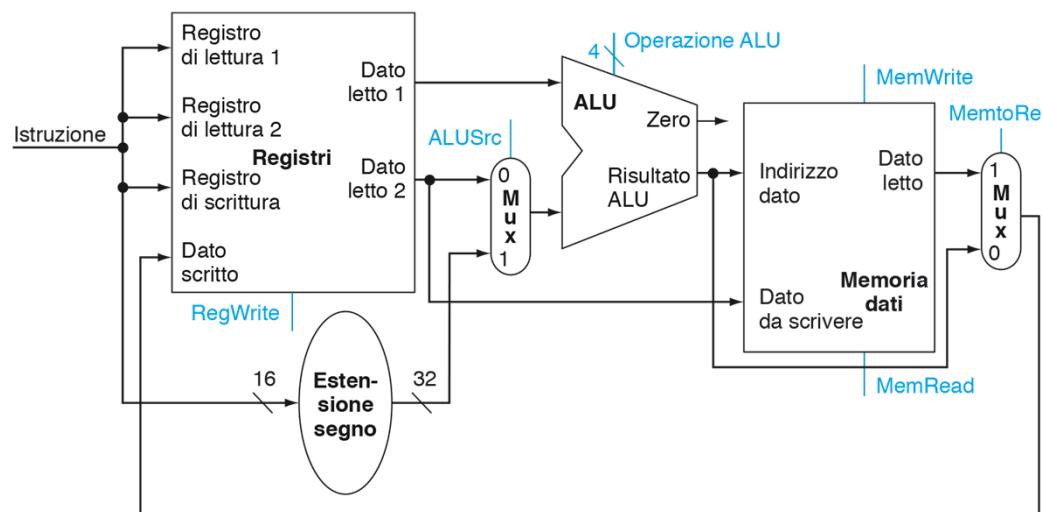
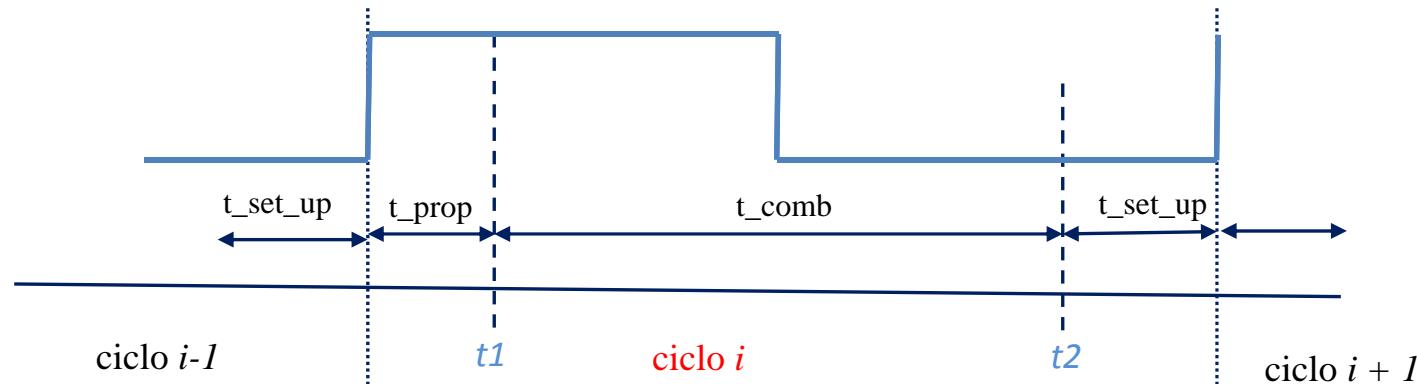
# Struttura della CPU



# Nota sulla temporizzazione

Nelle istruzioni aritmetiche (R e I) un registro sorgente può anche essere destinazione!!!!

$T_{clock} = T_{prop} + T_{set\_up} + T_{comb}$  dove  $T_{comb}$  è sostanzialmente il tempo di operazione dell'ALU

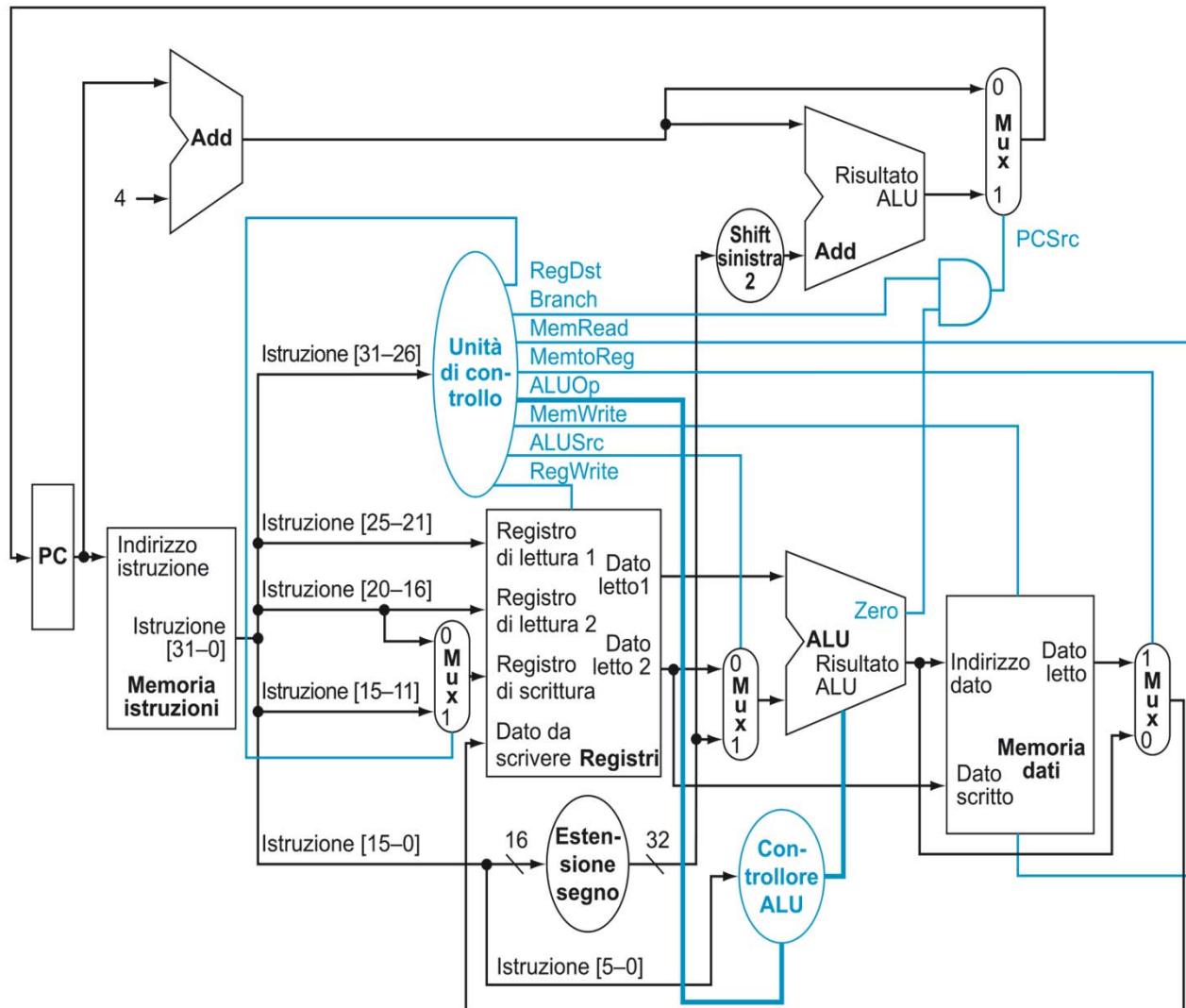


# I segnali di controllo

Nome del segnale	Effetto quando non asserito	Effetto quando asserito
RegDst	Il numero del registro di scrittura proviene dal campo rt (bit 20-16)	Il numero del registro di scrittura proviene dal campo rd (bit 15-11)
RegWrite	Nulla	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 16 bit meno significativi dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nulla	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea «dato letto»
MemWrite	Nulla	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea «dato scritto»
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati



# Struttura della CPU con unità di controllo



# Codice operativo e segnali di controllo

Istruzione	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Tipo R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



# ..... e per eseguire anche la bne?

La logica di controllo per l'istruzione di salto condizionato

$$\text{PCsrc} = \text{Zero and Branch}$$



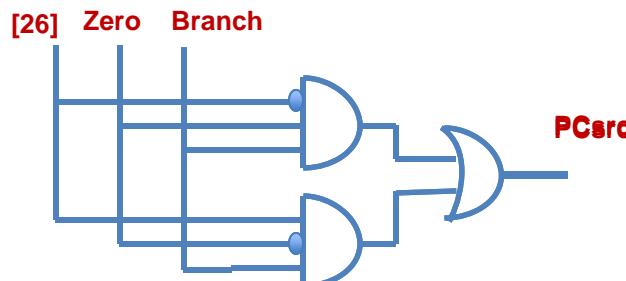
seleziona per la **beq** l'indirizzo della prossima istruzione da eseguire ( $\text{PCsrc}=1$  salta,  $\text{PCsrc}=0$ , in sequenza)

- per **beq** si salta se il bit Zero = 1 e cioè se i due registri selezionati hanno valori identici
- per **bne** si salta se il bit Zero = 0 e cioè se i due registri selezionati **non** hanno valori identici

Dobbiamo modificare la logica di controllo (in modo banale!!!) che genera il segnale **PCsrc** tenendo conto dei bit di codice operativo [31-26] per distinguere **beq** da **bne**

- beq – Op\_code = 4, in binario 000100
- bne – Op\_code = 5, in binario 000101
- quindi differiscono per il bit [26]

$$\text{PCsrc} = (\text{notOp\_code [26]} \text{ and Zero and Branch}) \text{ or } (\text{Op\_code [26]} \text{ and notZero and Branch})$$



# Implementazione delle istruzioni salto incondizionato (jump)

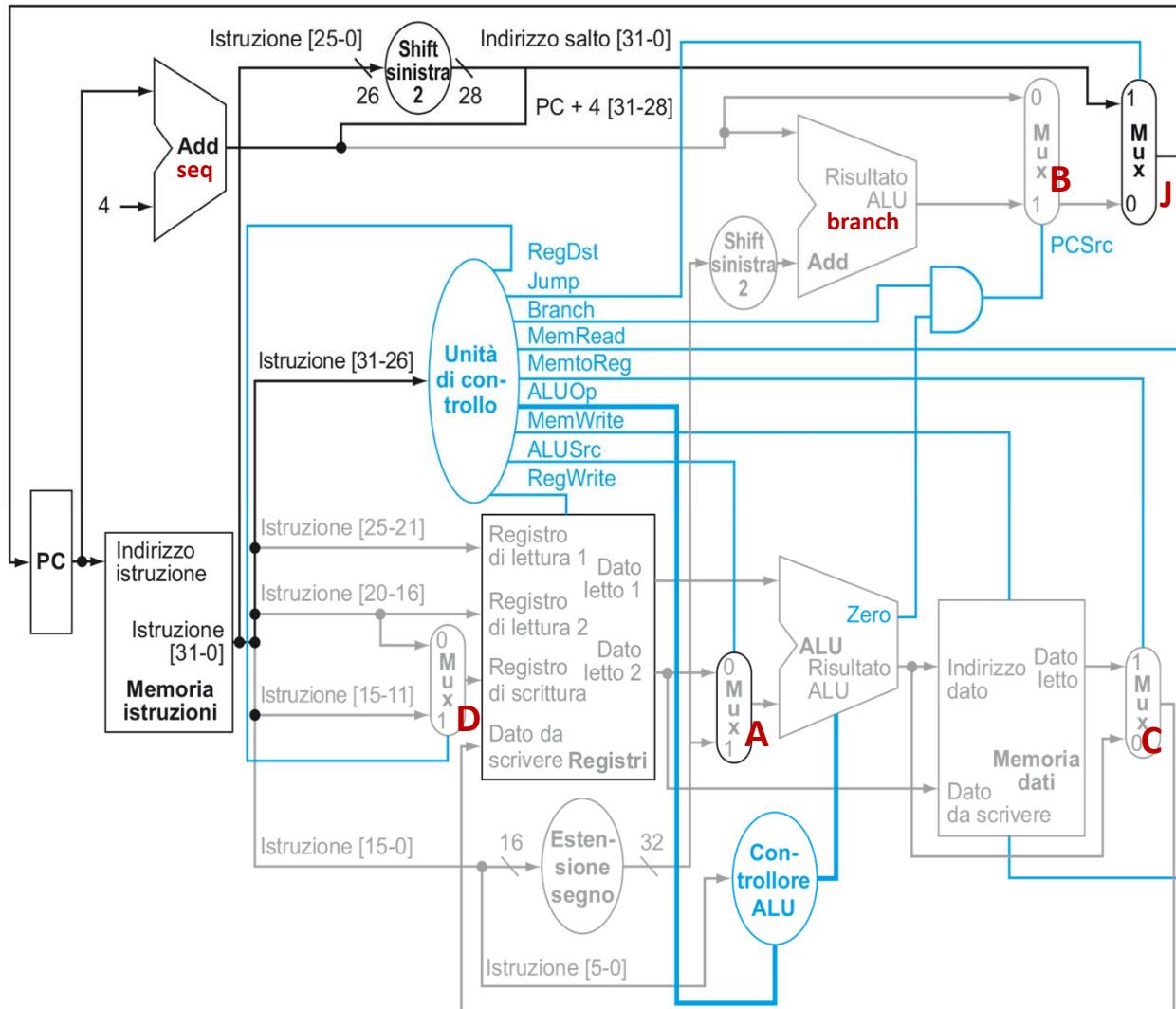
	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Commenti
Formato J	op [31-26]		campo indirizzo [25-0]				istruzioni di salto

Calcolo dell'indirizzo di destinazione del salto:

4 bit	26 bit	2 bit
PC+4 [31-28]	campo indirizzo [25-0]	00



# Estensione CPU e unità di controllo per istruzioni di jump



## Implementazione della CPU a singolo ciclo

Si avvia all'esecuzione un'istruzione per ogni ciclo di clock e ogni istruzione deve essere iniziata e completata in un solo ciclo di clock

Il ciclo di clock deve avere la stessa lunghezza per ogni istruzione ( $CPI = 1$ )

La lunghezza del ciclo di clock è determinata dal percorso più lungo (percorso critico o *critical path*): nell'esempio l'istruzione di load che utilizza 5 unità funzionali in serie e richiedere  $T = 800 \text{ ps}$  ( $f = 1250 \text{ MHz}$ )

L'implementazione su singolo ciclo non è molto veloce perché le altre istruzioni potrebbero essere implementate con un ciclo di clock più breve: nell'esempio le istruzioni tipo R richiedono  $T = 600 \text{ ps}$ , le istruzioni di store  $T = 700 \text{ ps}$ , istruzioni di salto condizionato  $T = 500 \text{ ps}$  e istruzioni di salto incondizionato può essere considerato pari a 200 ps





**POLITECNICO**  
MILANO 1863

# **Architettura dei calcolatori e sistemi operativi**

## **Il processore pipeline Capitolo 4 P&H**

**28 . 10 . 2015**

# Pipelining

Tecnica per migliorare le prestazioni basata sulla sovrapposizione dell'esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale.

## Idea base:

Il lavoro svolto in una *CPU pipeline* per eseguire un'istruzione è diviso in passi (**stadi di pipeline**), che richiedono una frazione del tempo necessario al completamento dell'intera istruzione.

Gli stadi sono sovrapposti per formare la pipeline: le istruzioni entrano da una estremità, vengono elaborate attraverso gli stadi, escono dall'altro estremo.



# Pipelining (cont.)

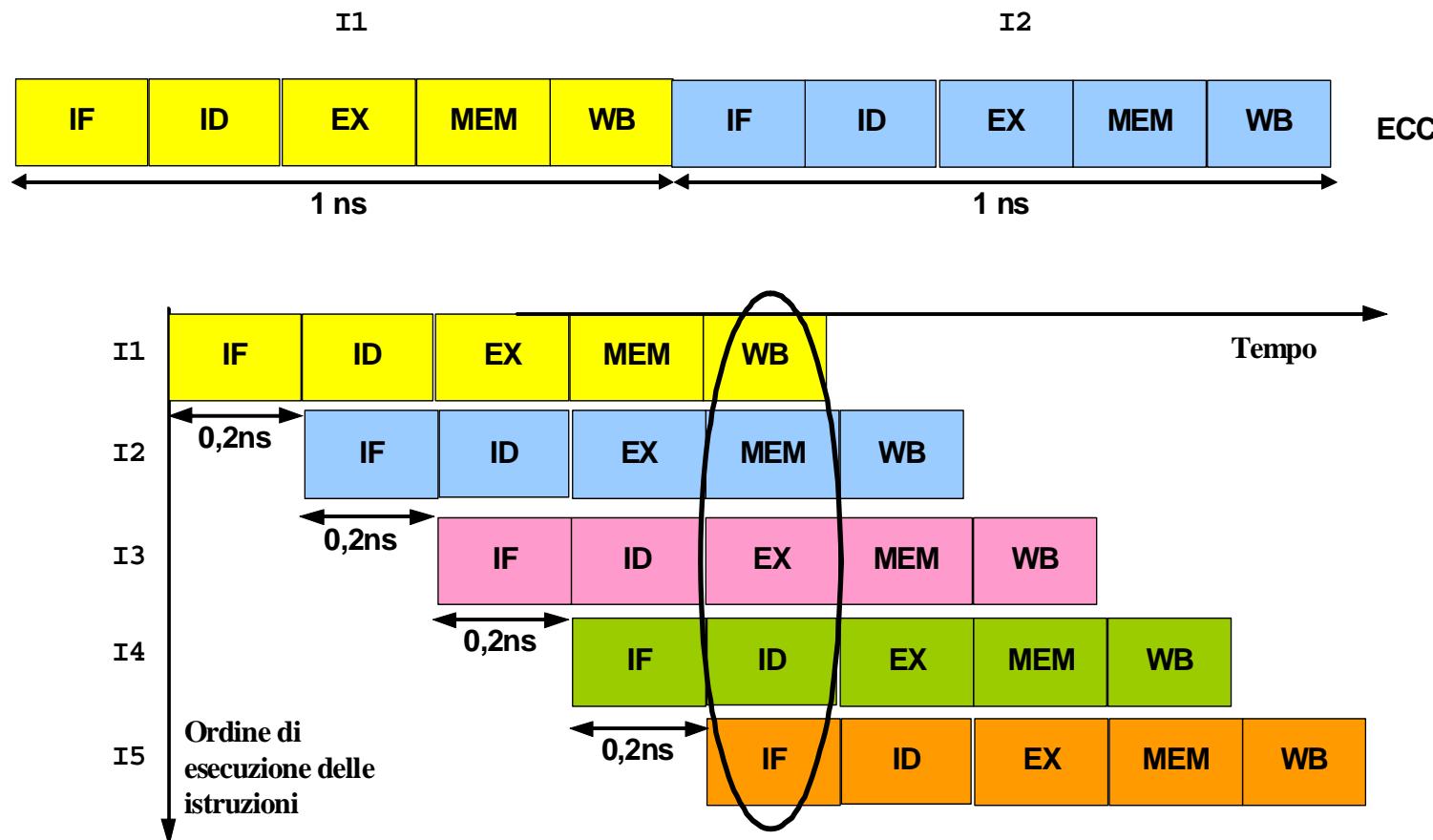
Principale vantaggio: tecnica trasparente al programmatore.

Tecnica simile ad una catena di montaggio: una nuova automobile esce dalla catena nel tempo necessario per svolgere uno dei molti passi.

Una catena di montaggio non riduce il *tempo* di realizzazione di una singola automobile, ma aumenta il numero di automobili costruite simultaneamente e pertanto la *frequenza* con cui le automobili vengono iniziate e completate.



# Esecuzione sequenziale vs. pipelining



# Pipelining (cont.)

Il tempo necessario per far avanzare un'istruzione di uno stadio lungo la pipeline corrisponde idealmente ad un ciclo di clock.

Poiché gli stadi di pipeline sono collegati in successione, devono tutti operare in modo sincrono: la durata di un ciclo di clock è determinata dal tempo richiesto per lo stadio più lento della pipeline (es. 200 ps).

L'obiettivo dei progettisti è **bilanciare** la lunghezza di ciascuno stadio.

Se gli stadi sono perfettamente bilanciati, l'accelerazione *ideale* dovuta al pipelining è pari al numero di stadi di pipeline: una *CPU* pipeline a 5 stadi è 5 volte più veloce della stessa *CPU* senza pipeline, in termini di completamento di una istruzione rispetto alla successiva



# Pipelining (cont.)

In generale, gli stadi non sono perfettamente bilanciati e l'introduzione del pipelining comporta costi aggiuntivi

- L'intervallo di tempo fra il completamento di 2 istruzioni è superiore al valore minimo possibile
- L'incremento di velocità sarà minore del numero di stadi di pipeline introdotto (in genere una pipeline a 5 stadi non riesce a quintuplicare le prestazioni).



# Miglioramento delle prestazioni

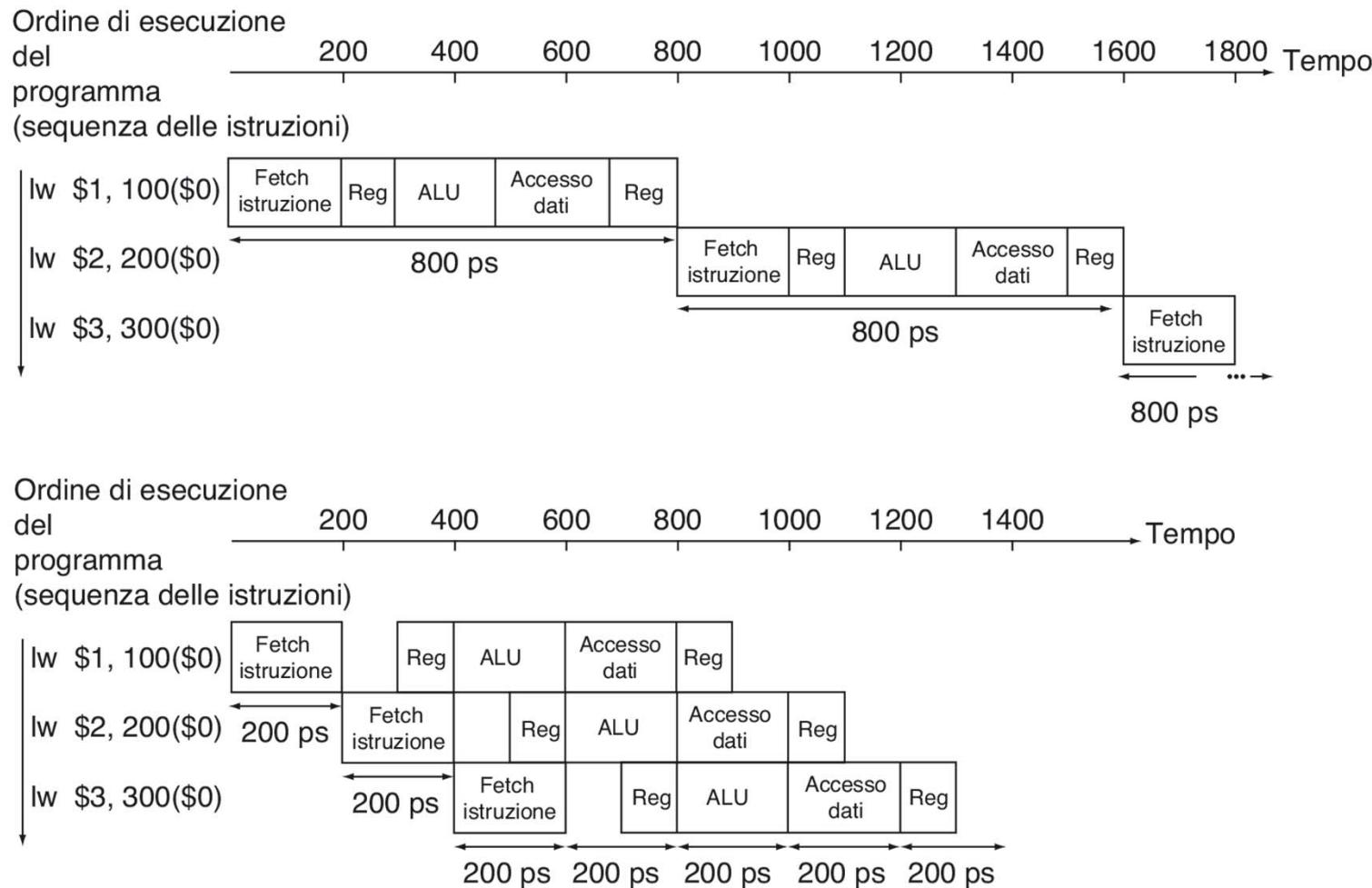
Architettura MIPS: Istruzioni suddivise al massimo in 5 passi ⇒  
**Pipeline a 5 stadi**

Ogni stadio di pipeline ha una durata prefissata (**ciclo di pipeline**), che deve essere sufficientemente lunga da consentire l'esecuzione dell'operazione più lenta:

ad esempio si deve adottare un ciclo di almeno  $200\text{ ps}$ , sebbene alcuni stadi richiedano solo  $100\text{ ps}$ .



# Confronto tra una esecuzione a singolo ciclo di clock e pipeline



# Miglioramento delle prestazioni (cont.)

Il fattore di miglioramento relativo all'esecuzione di 3 istruzioni di *load* dovrebbe essere

$$T_{\text{inizio}} \text{ istruzione\_4 senza pipeline} / T_{\text{inizio}} \text{ istruzione\_4 con pipeline}$$

$$2400 (3 \times 800) \text{ ps} / 600 (3 \times 200) \text{ ps} = 4$$

Il fattore di miglioramento è 4 e non 5 perché alcuni passi dell'esecuzione richiedono in effetti un tempo inferiore al ciclo di clock della pipeline

Ma il *tempo di esecuzione totale* di 3 istruzioni di *load* comporta un miglioramento più modesto:

- 3 istruzioni di *load* senza pipeline:  $3 \times 800 \text{ ps} = 2400 \text{ ps}$
- 3 istruzioni di *load* con pipeline:  $2 \times 200 \text{ ps} + 1000 \text{ ps} = 1400 \text{ ps}$
- Fattore di *miglioramento* =  $2400 / 1400 = 1,71$

Questa differenza è provocata dal *tempo necessario a riempire e svuotare la pipeline*: servono 4 stadi per riempire la pipeline



## Miglioramento delle prestazioni (cont.)

Al crescere del numero di istruzioni il rapporto tra i tempi totali di esecuzione dei programmi su macchine senza e con pipeline è vicino al limite ideale (stabilito dal rapporto dei tempi di completamento delle istruzioni):

- 1 000 000 istruzioni di *load* senza pipeline:  
 $1\ 000\ 000 \times 0,8\ ns = 800\ 000\ ns$
- 1 000 000 istruzioni di *load* con pipeline:  
 $1\ 000\ 000 \times 0,2\ ns + 0,8\ ns = 200\ 000,8\ ns$
- $\Rightarrow 800\ 000\ ns / 200\ 000,8\ ns \cong 3,999984$

Per 1000 istruzioni  $800\ ns / 200,8ns = 3,984$



# Miglioramento delle prestazioni (cont.)

Caso ideale (Caso asintotico):

- La **latenza** (tempo di esecuzione totale) della singola istruzione di *load* è **peggiorata** perché è passata da *800 ps* a *1000 ps*
- Il **throughput** (numero di istruzioni completeate nell'unità di tempo) è **migliorato** di 4 volte:  
(1 istruzione di load completata ogni *800 ps*) vs.  
(1 istruzione di load completata ogni *200 ps*).



# Miglioramento delle prestazioni (cont.)

Se avessimo considerato una *CPU1* a singolo ciclo da *1000 ps* composta da 5 passi ciascuno da *200 ps* e una *CPU2* pipeline ideale (caso asintotico) con 5 stadi da *200 ps* avremmo trovato:

- La **latenza** di ogni singola istruzione rimane **invariata** e pari a *1000 ps*.
- Il **throughput** è **migliorato** di 5 volte: (1 istruzione di *load* completata ogni *1000 ps*) vs. (1 istruzione completata ogni *200 ps*).



# Prestazioni della pipeline

Incremento della velocità prodotto dalla pipeline se gli stadi sono perfettamente bilanciati

In condizioni ideali

Tempo tra due istruzioni con pipeline =

Tempo tra due istruzioni senza pipeline

Numero degli stadi della pipeline



# Considerazioni sulla struttura delle istruzioni MIPS

- Le istruzioni MIPS hanno tutte la stessa lunghezza: semplificazione per le fasi di prelievo e decodifica delle istruzioni
- Le istruzioni MIPS hanno un numero molto ridotto di formati diversi e, non considerando il registro destinazione delle istruzioni aritmetico-logiche, gli altri *due registri* sono sempre specificati nella stessa posizione (*rs* e *rt*) dell'istruzione codificata
- Gli operandi residenti in memoria sono presenti solo nelle istruzioni di *load* e *store*
- L'allineamento degli operandi in memoria è obbligatorio: il trasferimento dati con la memoria avviene sempre con un solo accesso



# Passi svolti durante l'esecuzione delle istruzioni

Istruzioni aritmetico-logiche: **op \$x, \$y, \$z**

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$y e \$z	Op. ALU sui Dati Letti (\$y op \$z)	Scrittura nel Reg. Destinazione \$x
----------------------------------	--	--	--

Istruzioni di caricamento (*load*): **lw \$x, offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registro Base \$y	Op. ALU (\$y+offset)	Prelievo Dato M(\$y+offset)	Scrittura nel Reg. Destinazione \$x
----------------------------------	------------------------------	-------------------------	--------------------------------	--

Istruzioni di memorizzazione (*store*): **sw \$x, offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registri Base \$y & Sorg. \$x	Op. ALU (\$y+offset)	Scrittura Dato M(\$y+offset)
----------------------------------	--	-------------------------	---------------------------------

Istruzioni di salto condizionato: **beq \$x, \$y, offset**

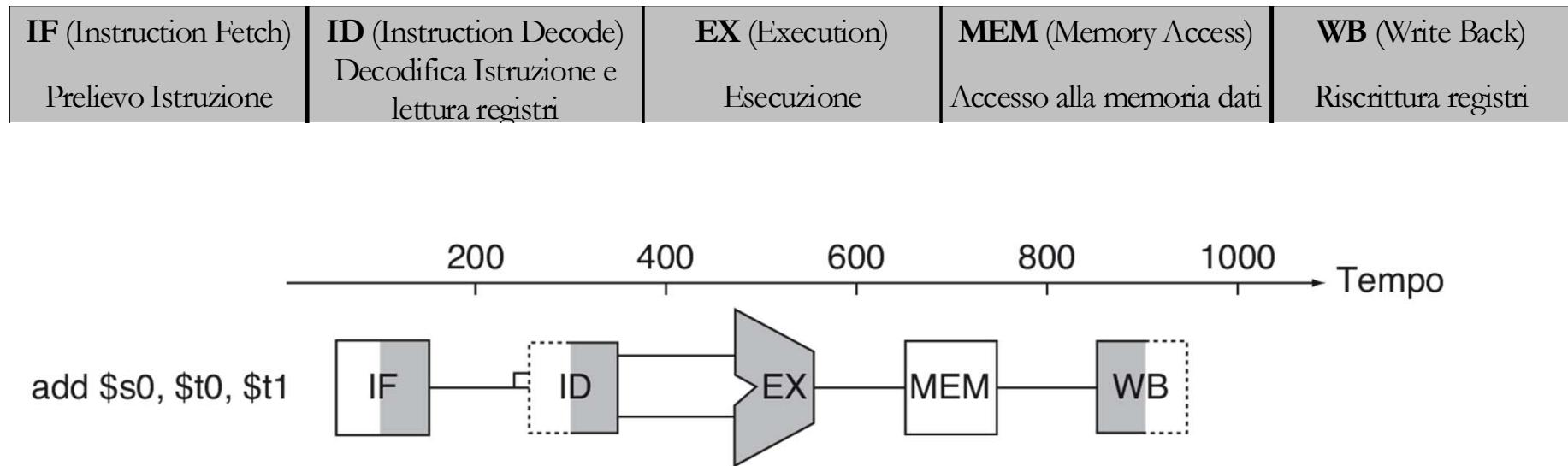
Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$x e \$y	Op. ALU (\$x-\$y) & (PC+4+offset)	Scrittura nel PC
----------------------------------	--	--------------------------------------	---------------------

Istruzioni di salto non condizionato: **j L1**

Prelievo Istruz. & Increm. PC		(PC+4   L1   00)	Scrittura nel PC
----------------------------------	--	------------------	---------------------



# Passi svolti durante l'esecuzione delle istruzioni in modalità pipeline: formalizzazione



# Passi svolti durante l'esecuzione delle istruzioni in modalità pipeline

Prelievo Istruzione <b>IF</b> (Instruction Fetch)	Decodifica Istruzione <b>ID</b> (Instruction Decode)	Esecuzione <b>EX</b> (Execution)	Accesso alla Memoria <b>MEM</b> (Memory Access)	Riscrittura Registri <b>WB</b> (Write Back)
--	---	-------------------------------------	--	--

Istruzioni aritmetico-logiche: **op \$x,\$y,\$z**

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$y e \$z	Op. ALU sui Dati Letti (\$y op \$z)		Scrittura nel Reg. Destinazione \$x
----------------------------------	--	--	--	--

Istruzioni di caricamento (*load*): **lw \$x,offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registro Base \$y	Op. ALU (\$y+offset)	Prelievo Dato M(\$y+offset)	Scrittura nel Reg. Destinazione \$x
----------------------------------	------------------------------	-------------------------	--------------------------------	--

Istruzioni di memorizzazione (*store*): **sw \$x,offset(\$y)**

Prelievo Istruz. & Increm. PC	Lettura Registri Base \$y & Sorg. \$x	Op. ALU (\$y+offset)	Scrittura Dato M(\$y+offset)	
----------------------------------	--	-------------------------	---------------------------------	--

Istruzioni di salto condizionato: **beq \$x,\$y,offset**

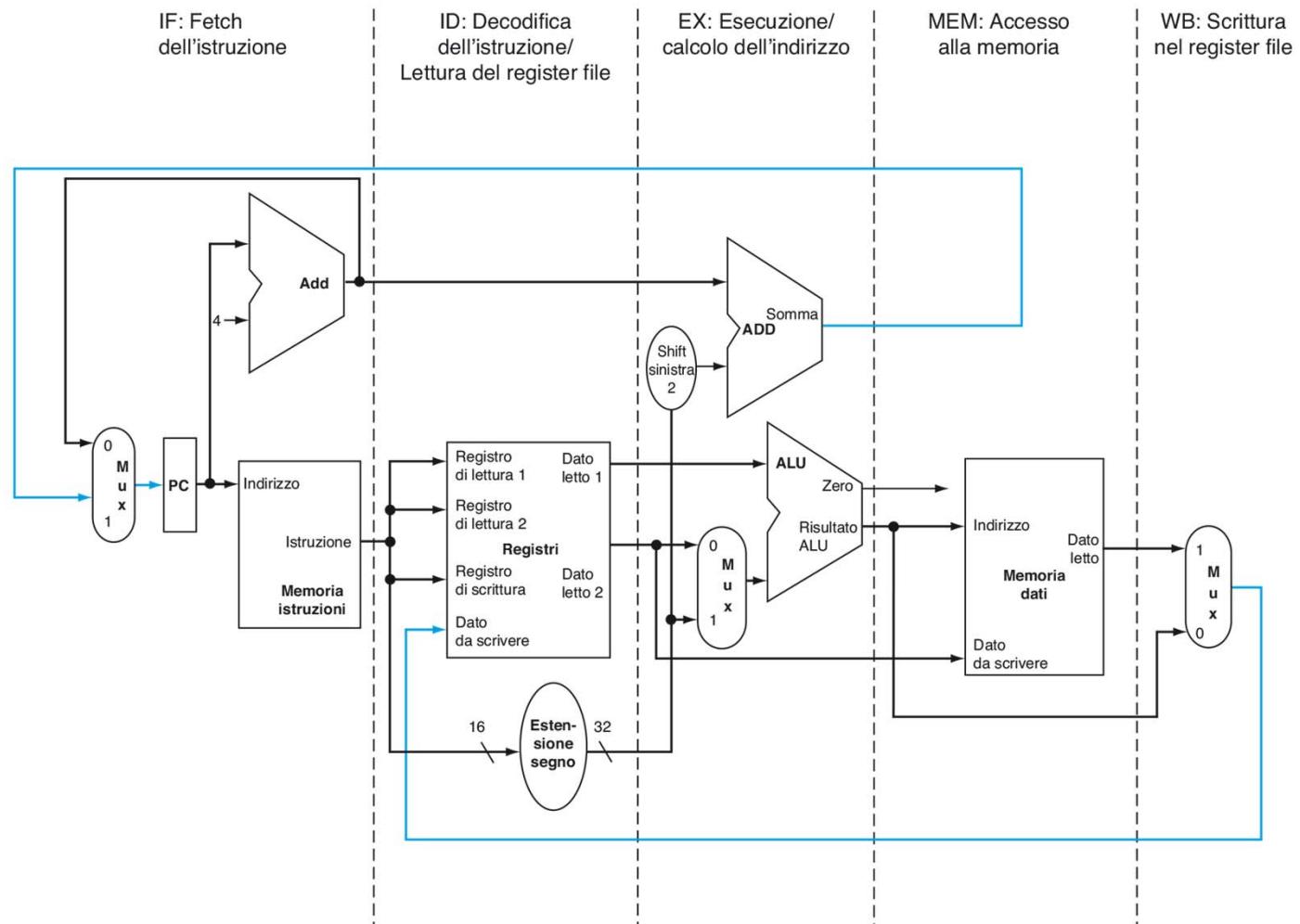
Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente \$x e \$y	Op. ALU (\$x-\$y) & (PC+4+offset)	Scrittura nel PC	
----------------------------------	--	--------------------------------------	---------------------	--

Istruzioni di salto non condizionato: **j L1**

Prelievo Istruz. & Increm. PC		(PC+4   L1   00)	Scrittura nel PC	
----------------------------------	--	------------------	---------------------	--



# Fasi (passi) e stadi



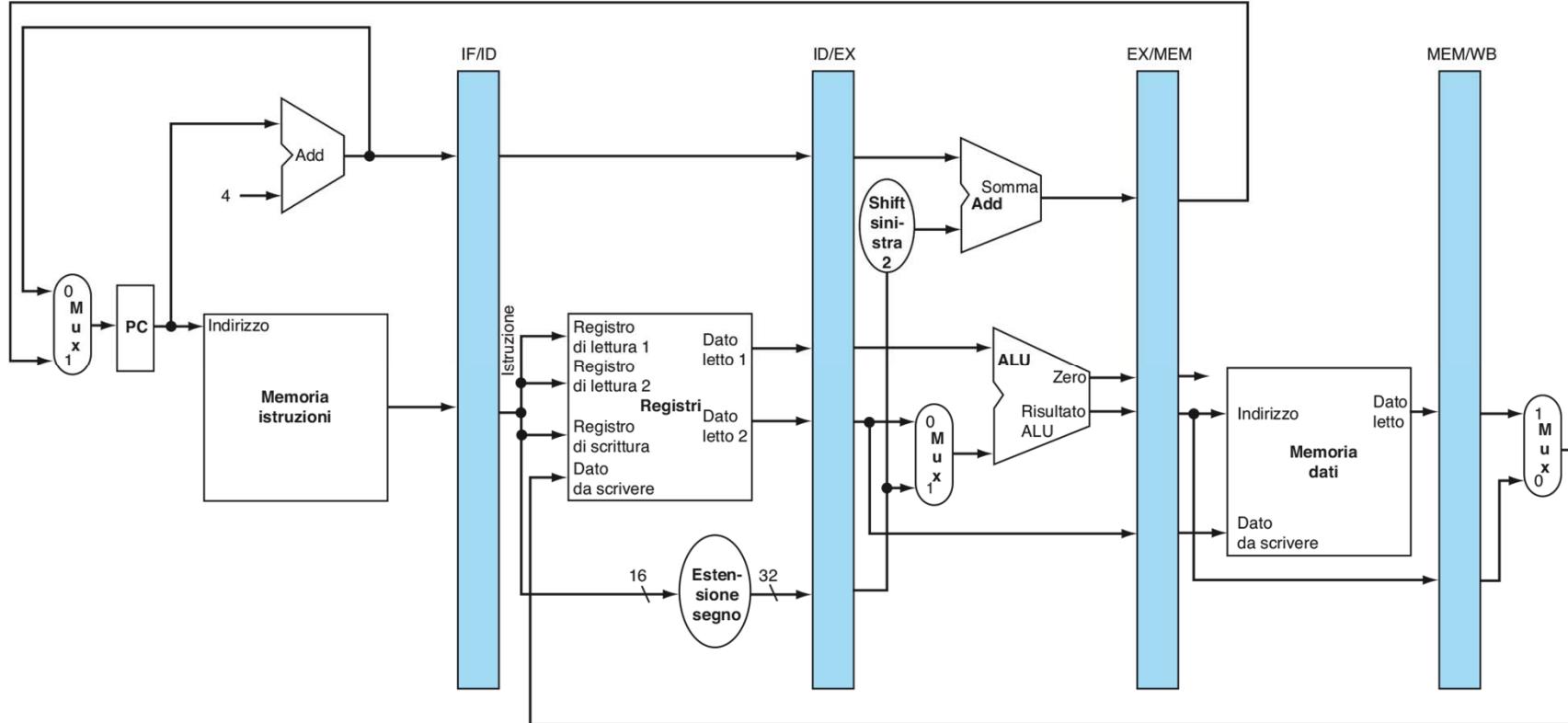
# Struttura pipeline della CPU MIPS

La suddivisione dell'esecuzione di un'istruzione in 5 passi implica che in ogni ciclo di clock siano in esecuzione 5 istruzioni

- la struttura di una *CPU* pipeline a 5 stadi deve essere scomposta in **5 parti** o ***stadi di esecuzione***, ciascuno corrispondente ad una delle fasi di pipeline
- devono essere introdotti ***registri di pipeline*** che separano i diversi stadi.



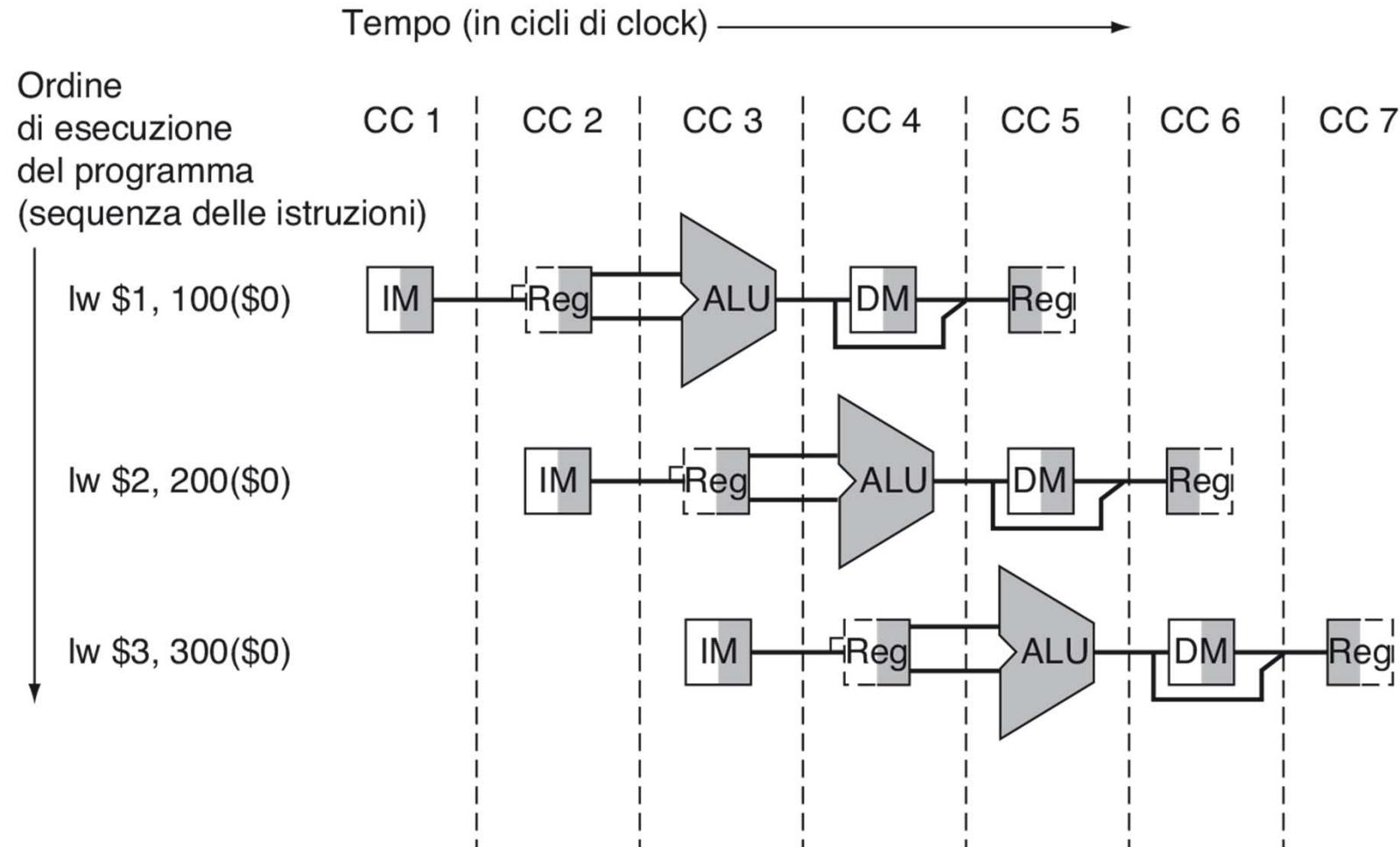
# Struttura pipeline della CPU MIPS (cont.)



dimensione registri di pipeline: IF/ID (32), ID/EX(128), EX/MEM(97), MEM/WB(64)  
“ruolo” del registro PC



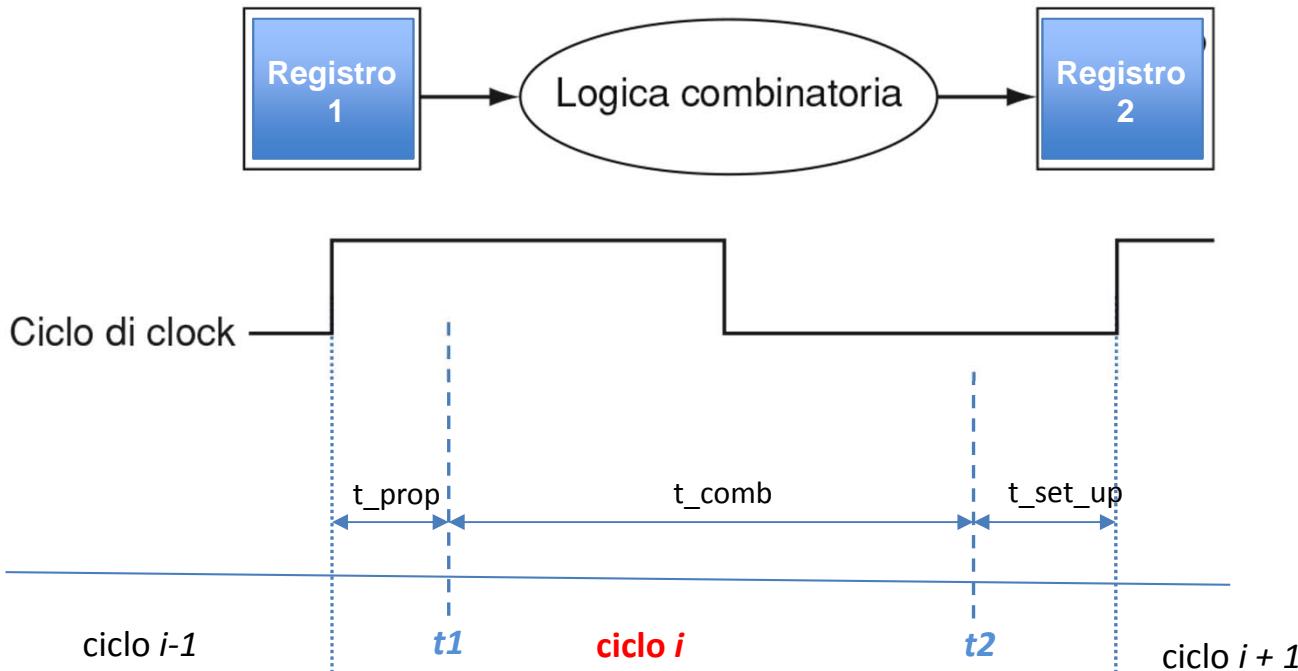
# Utilizzo delle risorse



Temporizzazione **lettura e scrittura registri interstadio**



# Registri interstadio e temporizzazione edge-triggered



Considerato un certo stadio

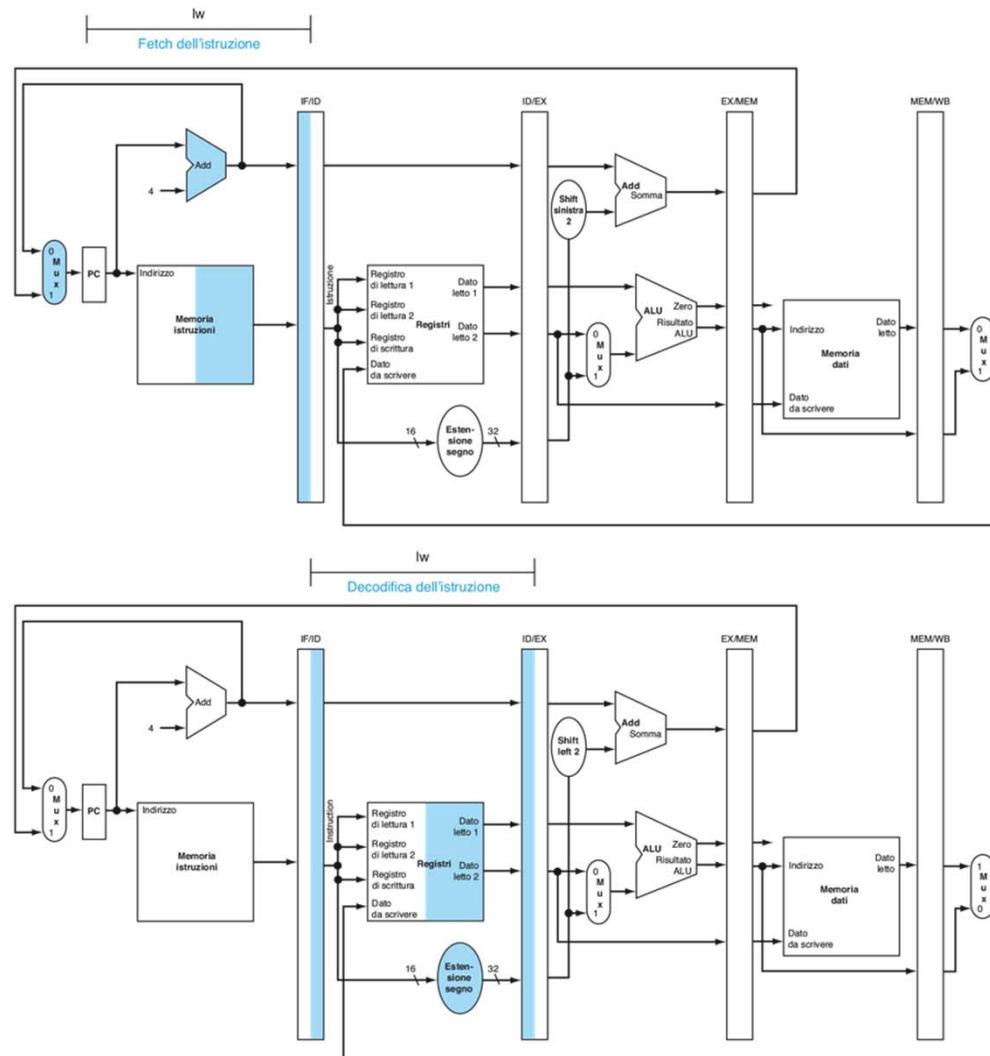
- Registro1 = registro interstadio di ingresso
- Logica combinatoria = elementi funzionali dello stadio
- Registro2 = registro interstadio di uscita

Sul fronte di salita che termina il ciclo  $i-1$ , i dati impostati in  $i-1$  vengono scritti in *Registro1*. Nel **ciclo  $i$** :

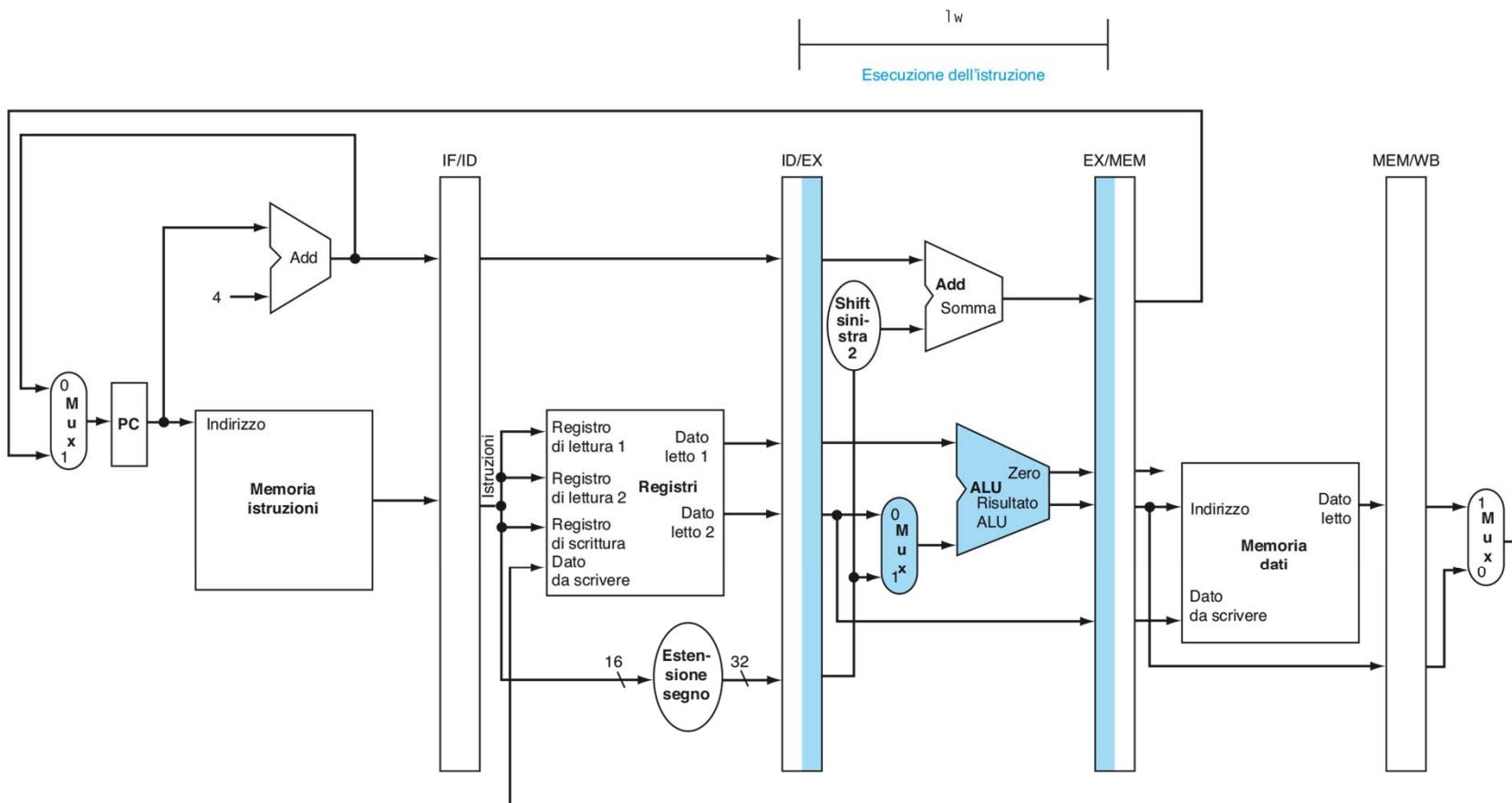
- a partire da  $t_1$  (quindi dopo  $t_{prop}$ ) le uscite di *Registro1* sono da considerarsi stabili e vengono usate (lette) come ingressi per far lavorare lo stadio
- a partire da  $t_2$  e per  $t_{setup}$  le uscite degli elementi funzionali dello stadio sono da considerarsi stabili per poter essere scritte in *Registro2* sul fronte di salita di chiusura del ciclo  $i$



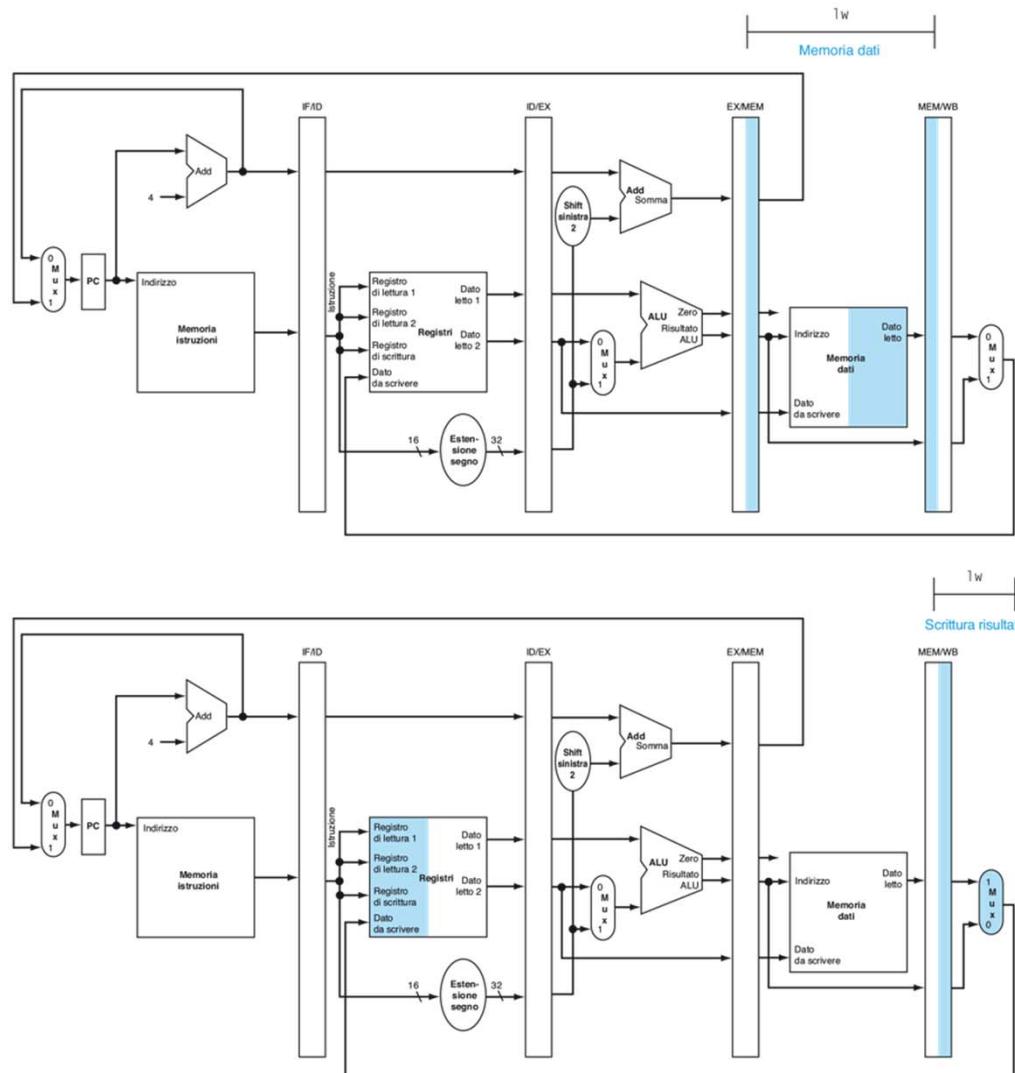
# Fetch e decodifica di *load*



# Esecuzione di *load*



# Accesso a memoria e scrittura in registro di load



# Struttura pipeline della CPU MIPS (cont.)

Le informazioni memorizzate nei registri interstadio sono relative ad ***istruzioni diverse!***

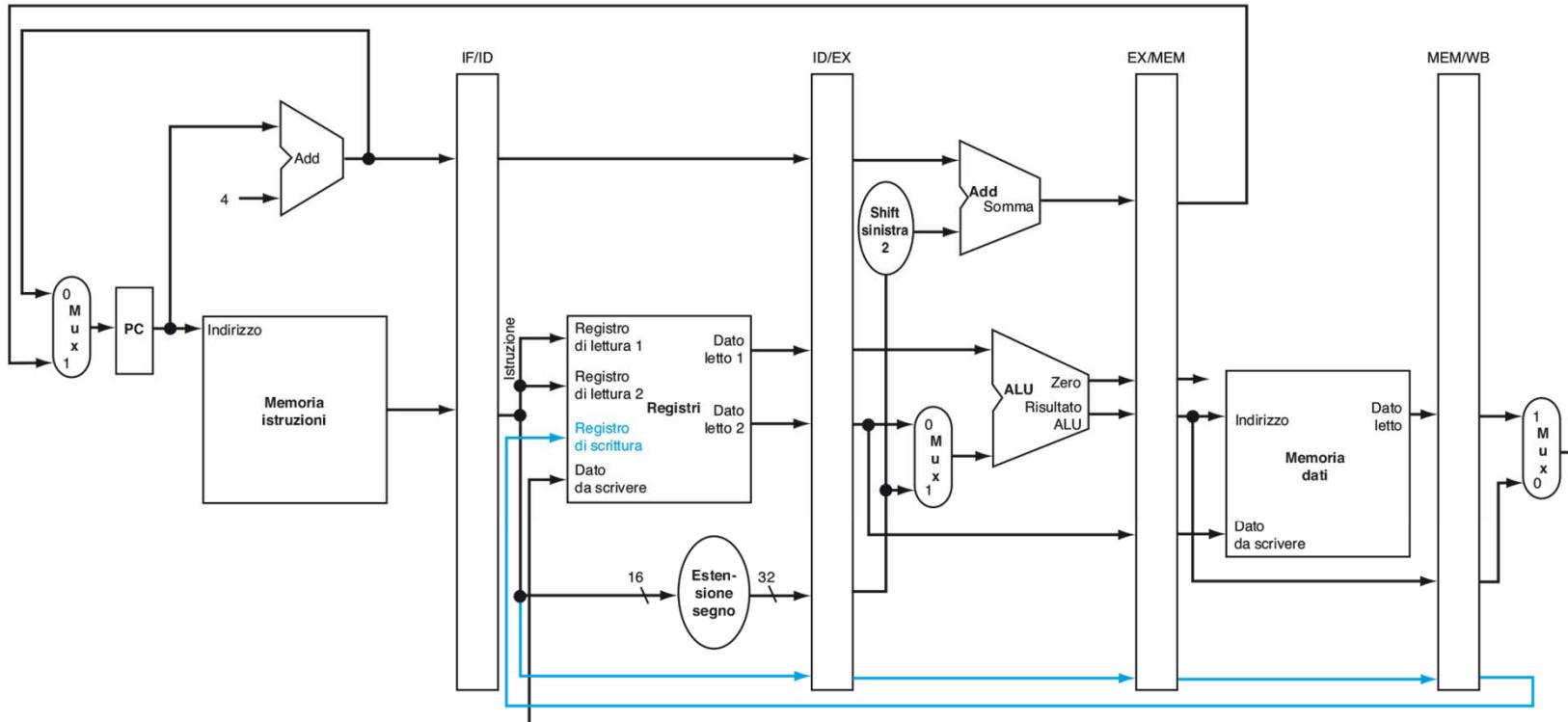
L'istruzione memorizzata nel registro di pipeline *IF/ID* fornisce il numero del registro di scrittura, mentre i dati scritti sono quelli relativi all'istruzione che si trova nel registro *MEM/WB*

- occorre modificare la *CPU* in modo da trasmettere attraverso i registri di pipeline (*ID/EX*, *EX/MEM* e *MEM/WB*) l'indirizzo del registro da scrivere durante lo stadio *WB*.

L'indirizzo del registro di scrittura, che viene fatto passare attraverso gli stadi intermedi, proviene dal registro di pipeline *MEM/WB* insieme ai dati.



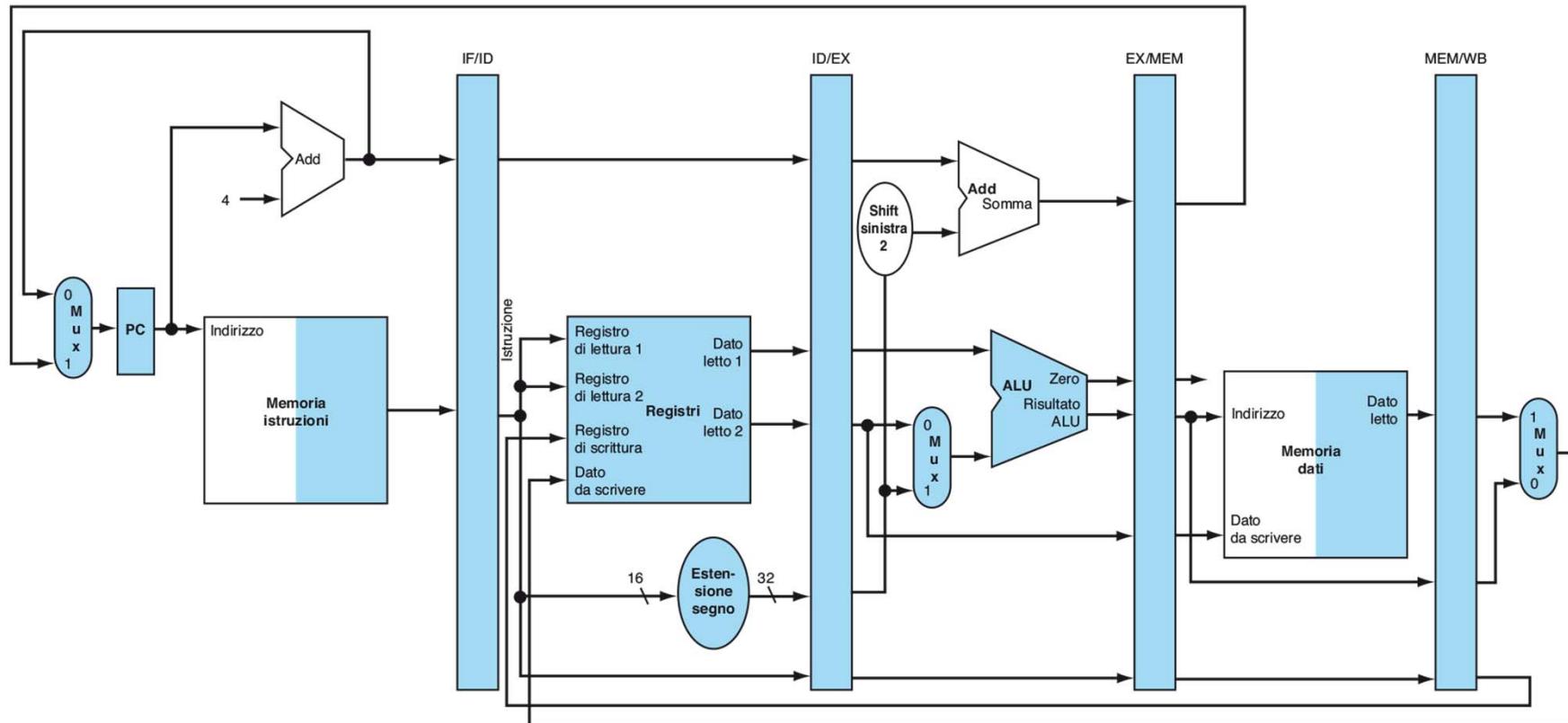
# Passaggio di informazioni tra stadi



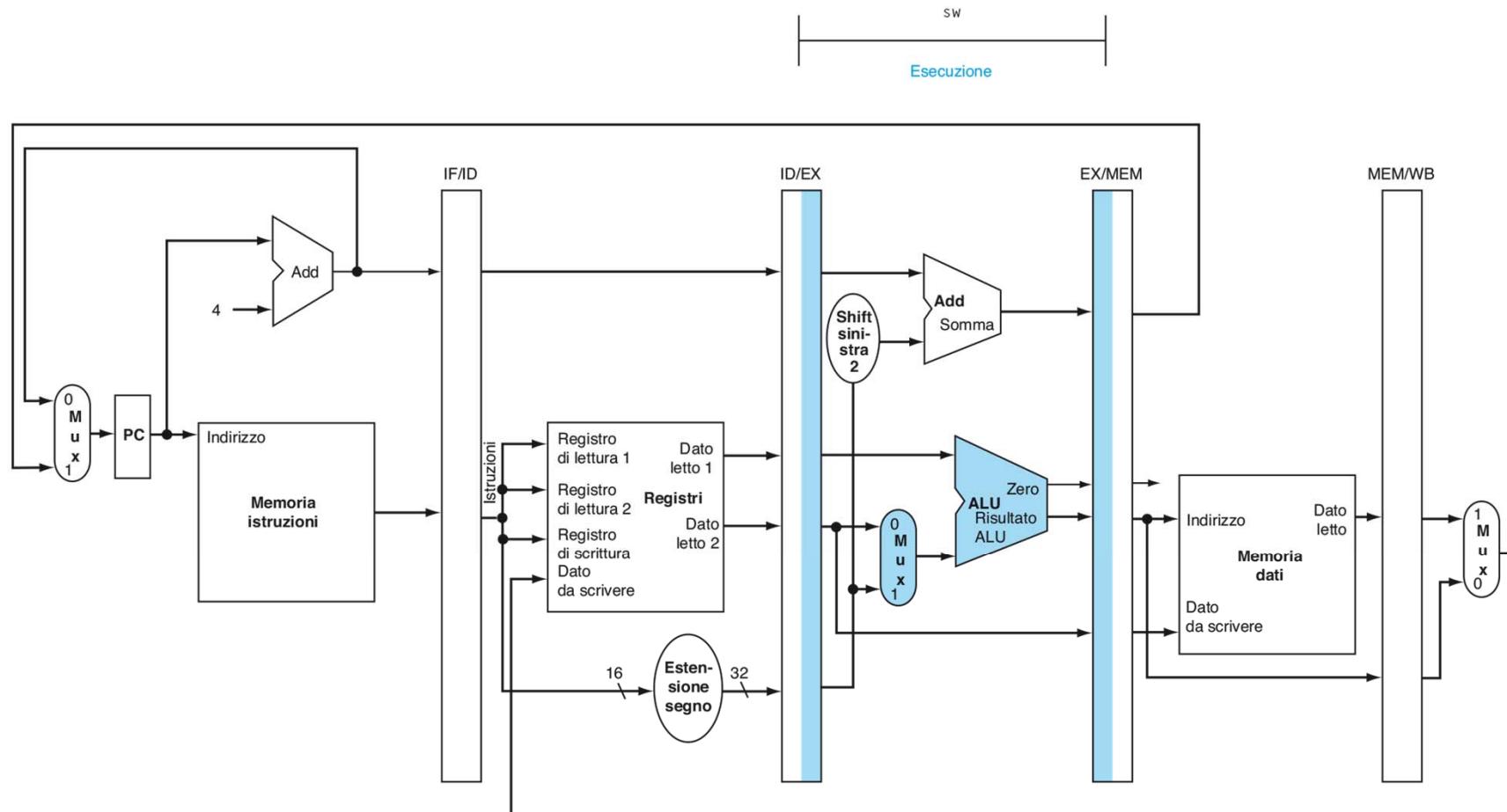
modifica registri di pipeline per propagazione indirizzo registro di scrittura



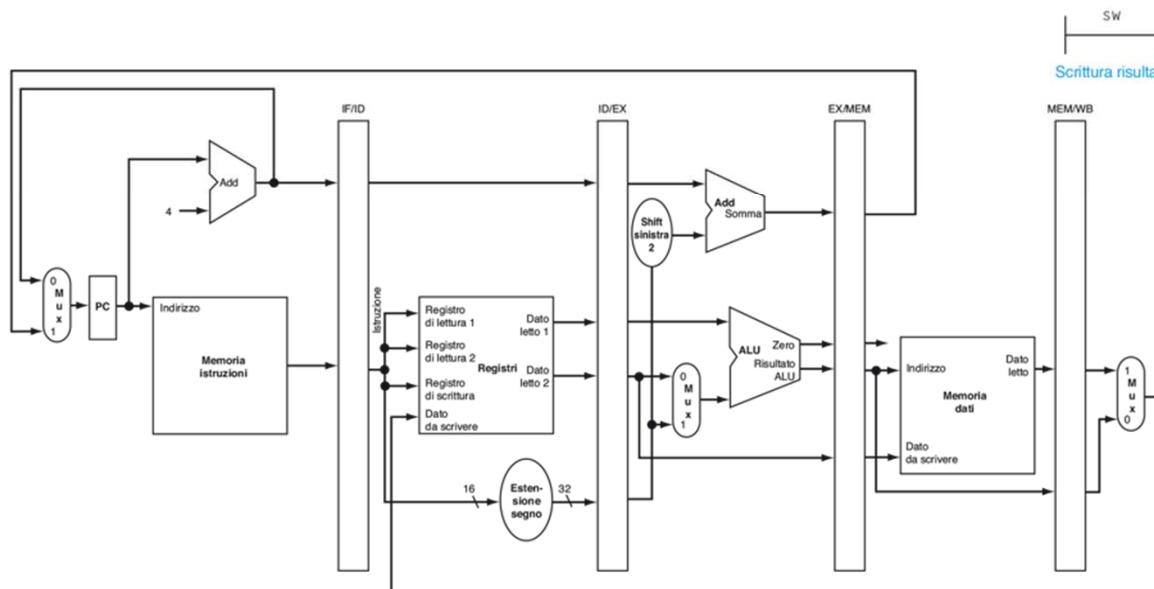
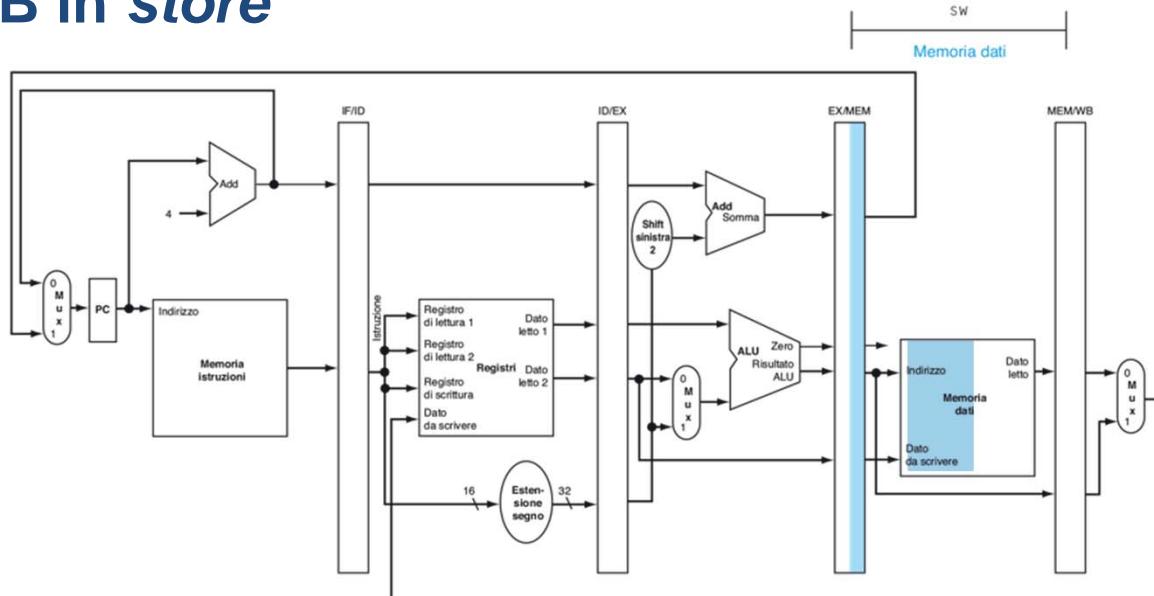
# Risorse utilizzate per eseguire la *load*



# Esecuzione di store



# MEM e WB in store



# Rappresentazione di 5 istruzioni in esecuzione (1)

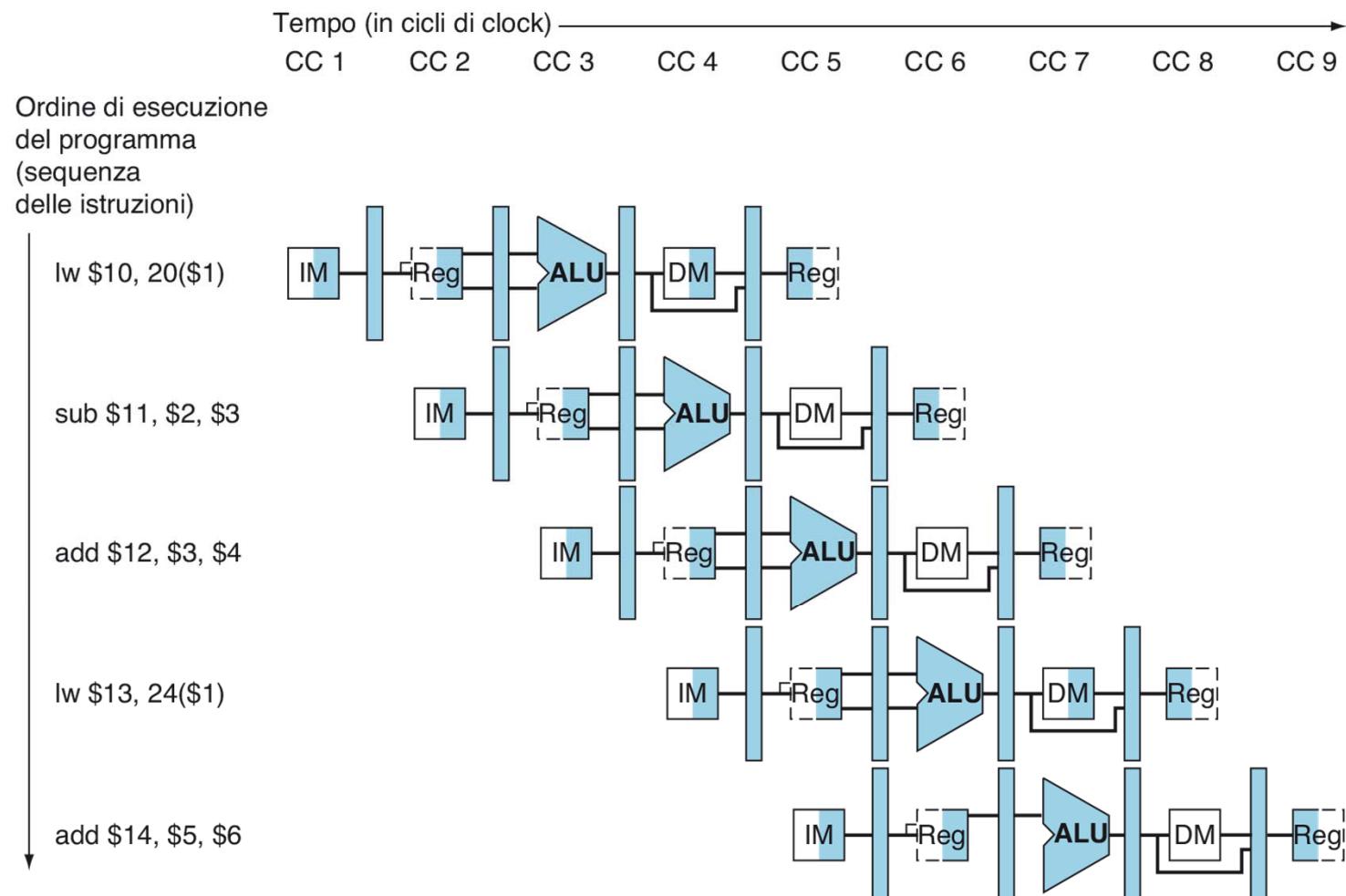
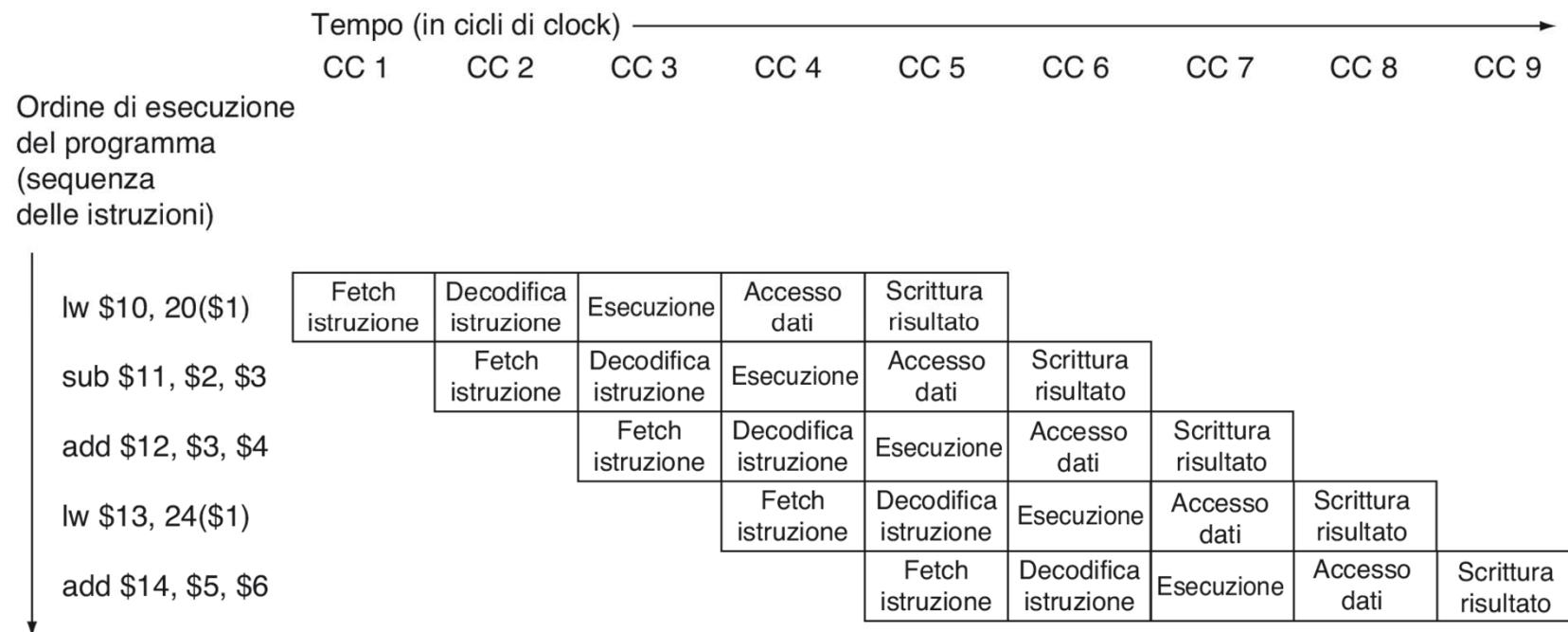


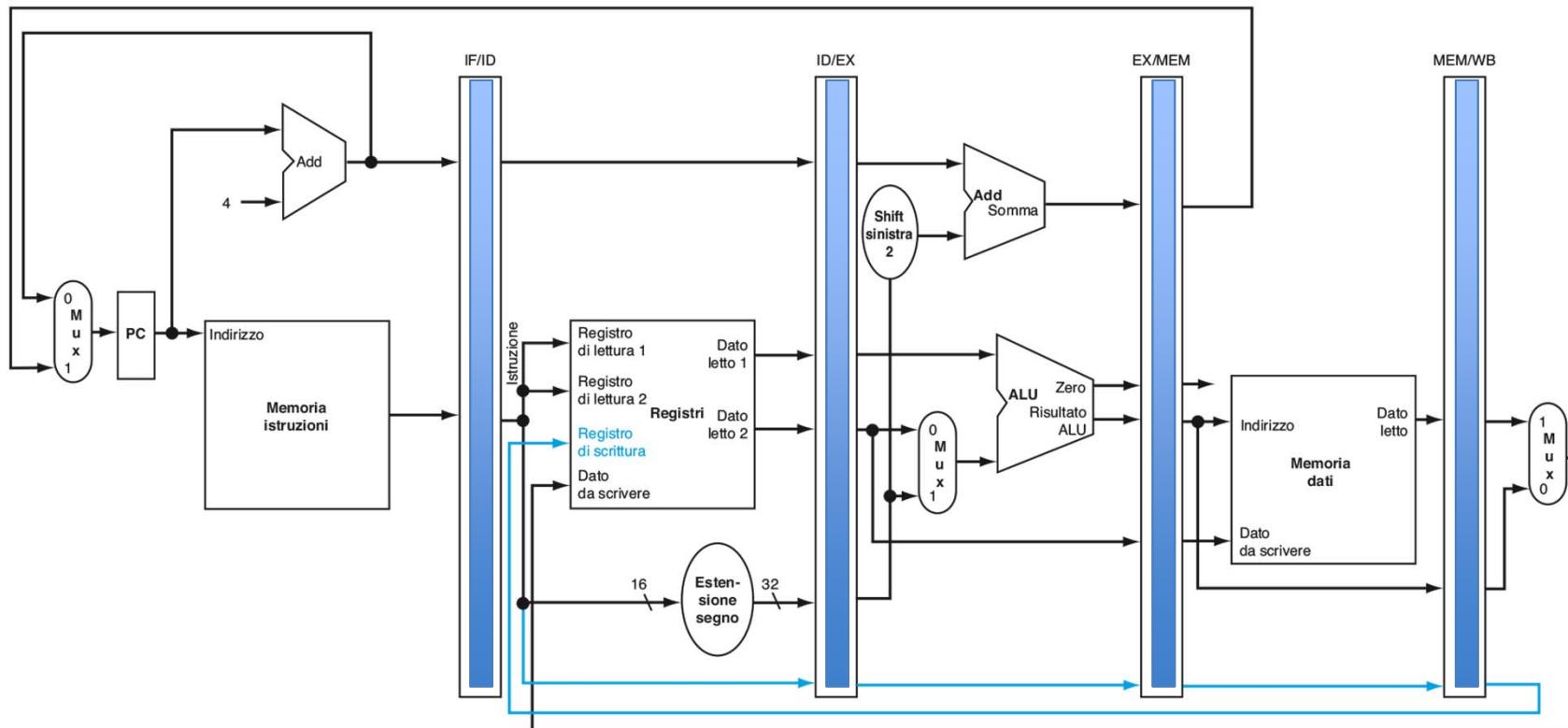
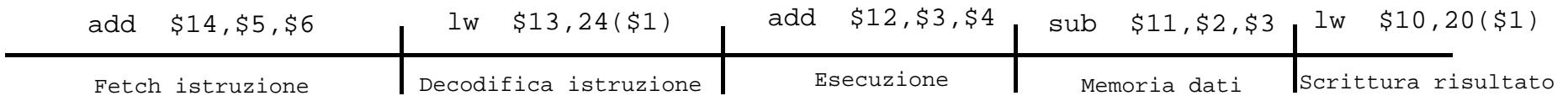
diagramma pipeline a più cicli con risorse fisiche in uso



## Rappresentazione di 5 istruzioni in esecuzione (2)



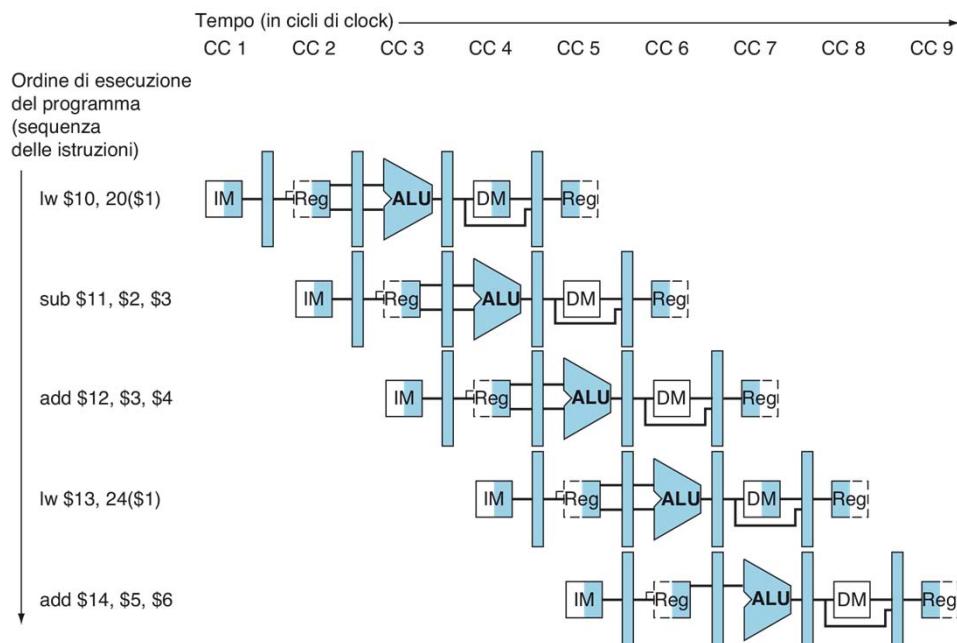
## Rappresentazione di 5 istruzioni in esecuzione (3)



pipeline corrispondente ad un certo ciclo di clock con indicazione delle istruzioni nei vari stadi



## Ancora sulla temporizzazione: il Register File e la sua temporizzazione in scrittura



Se in uno stesso ciclo di clock si vuole che una lettura di un registro di RF restituisca il valore correntemente scritto **dello stesso registro** è necessario che

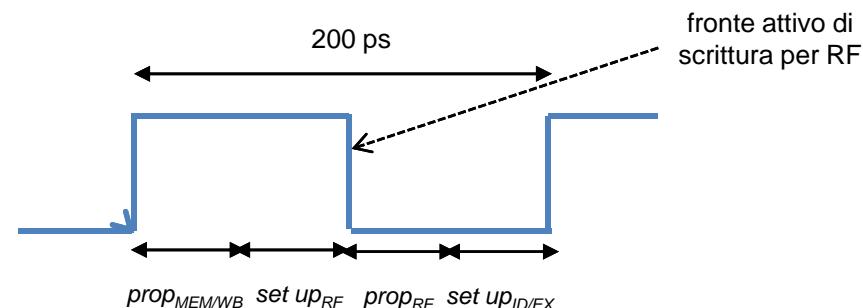
- $t_{prop, MEM/WB} + t_{set up, RF} \leq 100 \text{ ps}$  (scrittura RF)
- a 100 ps fronte di discesa e scrittura in RF
- $t_{prop, RF} + t_{set up, ID/EX} \leq 100 \text{ ps}$  (lettura RF)

In CC5 in ingresso al Register File si ha

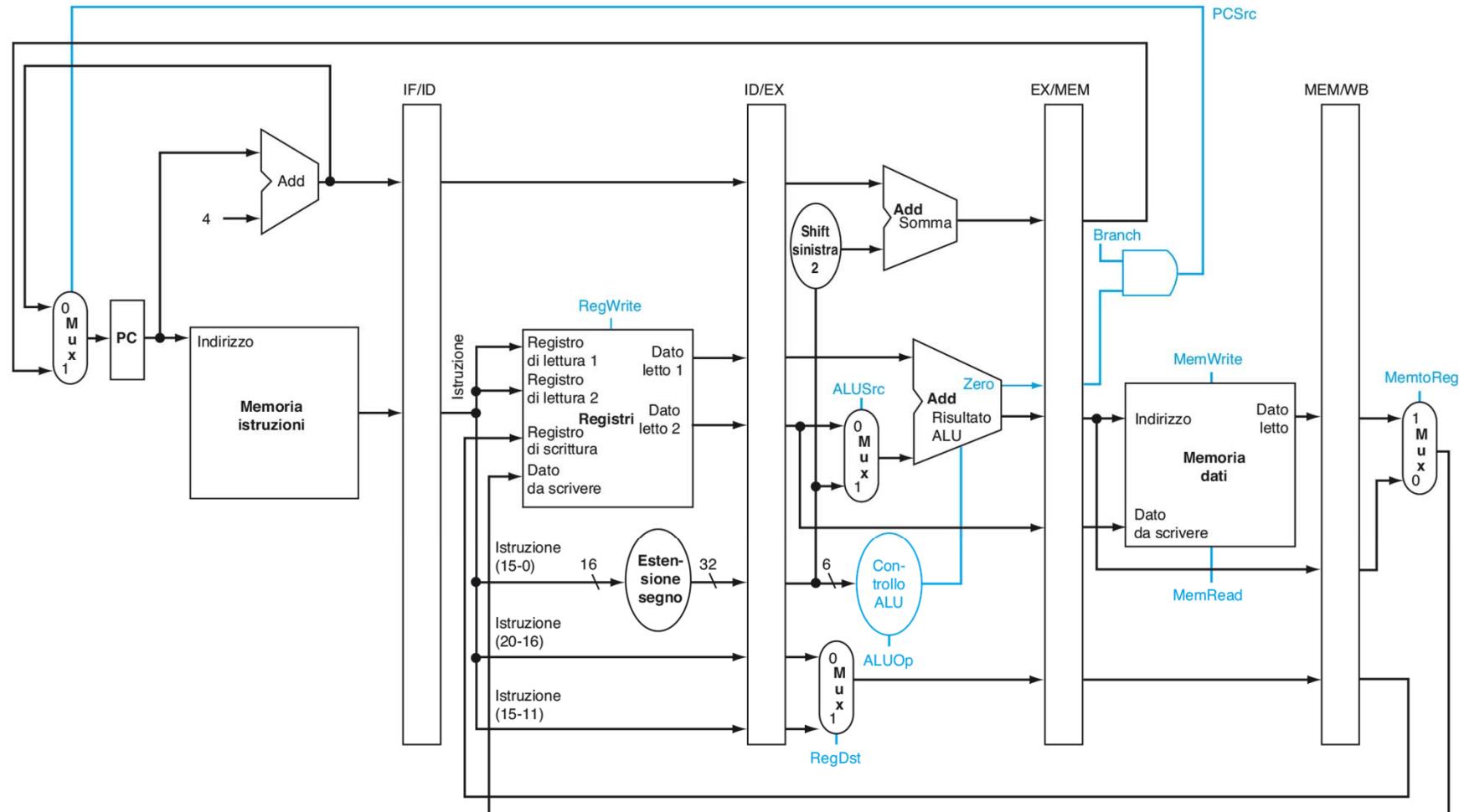
- da IF/ID indirizzo registro da leggere dell'istruzione che ha iniziato l'esecuzione (istruzione lw \$13, ...)
- da MEM/WB indirizzo dato da scrivere, valore da scrivere, segnale RegWrite a 1 (istruzione lw \$10, ...)

Al termine di CC5

- l'operazione di **scrittura deve essere terminata** (segnale RegWrite non più asserito per l'istruzione che la richiede): questo implica che il **fronte attivo** per la scrittura di RF deve essere quello di **discesa**
- in ID/EX ci deve essere il valore corretto del registro letto che serve all'istruzione che ha iniziato l'esecuzione



# Segnali dell'unità di controllo della pipeline



# Segnali di controllo per la ALU (come CPU a singolo ciclo)

Codice operativo istruzione	ALUOp	Operazione associata all'istruzione	Codice funzione	Operazione della ALU	Input di controllo della ALU
LW	00	load word	XXXXXX	somma	0010
SW	00	store word	XXXXXX	somma	0010
Branch equal	01	branch equal	XXXXXX	sottrazione	0110
Tipo R	10	somma	100000	somma	0010
Tipo R	10	sottrazione	100010	sottrazione	0110
Tipo R	10	AND	100100	AND	0000
Tipo R	10	OR	100101	OR	0001
Tipo R	10	set less than	101010	set less than	0111



# Effetto segnali di controllo (come per la CPU a singolo ciclo)

Nome del segnale	Effetto quando non asserito (0)	Effetto quando asserito (1)
RegDst	Il numero del registro di scrittura proviene dal campo rt (bit 20-16)	Il numero del registro di scrittura proviene dal campo rd (bit 15-11)
RegWrite	Nulla	Il dato viene scritto nel registro (del register file) individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 16 bit meno significativi dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nulla	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea «dato letto»
MemWrite	Nulla	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea «dato scritto»
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati



# Segnali di controllo e registri di pipeline

Ai primi due stadi IF e ID non sono associati segnali di controllo esplicativi

- l'unico considerato è il clock e il suo fronte attivo

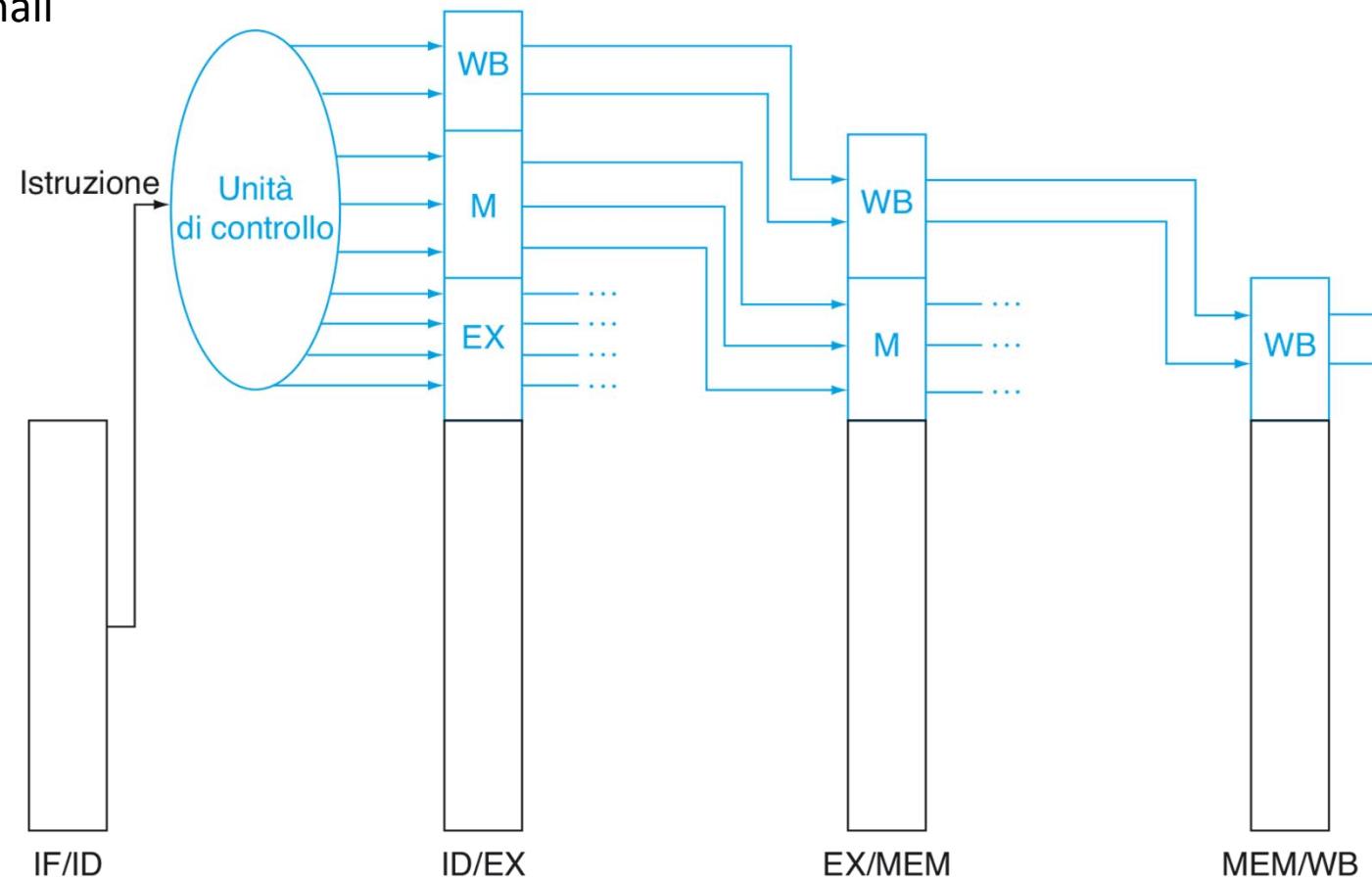
Istruzione	Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo				Segnali di controllo dello stadio di accesso alla memoria dati			Segnali di controllo dello stadio di scrittura	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	RegWrite	Memto-Reg
Formato R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

I valori dei segnali vengono tutti generati a partire dal contenuto del registro IF/ID che – dopo ogni fetch – contiene l'istruzione da eseguire e il valore del PC incrementato



# Implementazione dell'unità di controllo

- E' necessario impostare correttamente il valore dei 9 segnali di controllo per ciascuno stadio della pipeline per ciascuna istruzione
- Modo più semplice: estendere i registri di pipeline per salvare e propagare questi segnali



# Il dettaglio dei campi dei registri di pipeline

IF/ID	PC (32)	ISTRUZIONE (32)							
ID/EX	WB (2)	M (3)	EX (4)	PC (32)	(Rs) (32)	(Rt) (32)	Imm/offset esteso (32)	Rt (5)	Rd (5)
EX/MEM	WB (2)	M (3)	PC (32)	ALU_out (32)		Bit Z (1)	(Rt) (32)	R (5)	
MEM/WB	WB (2)	Dato letto (32)		ALU_out (32)		R (5)			

Legenda:

(32) ecc. = n° di bit del campo considerato

(Rs) o (Rt) = contenuto del registro Rs o Rt

Rt, Rd, R = numero di registro, se R può essere Rt o Rd

**WB(2)**: RegWrite, MemtoReg; **M(3)**: Branch, MemWrite, MemRead; **EX(4)**: RegDest, ALUOp(2), ALUsrc

**IF/ID.PC** e **ID/EX.PC** = PC esecuzione in sequenza, **EX/MEM.PC** = PC branch taken

**ID/EX.Rt** = reg. destinazione se *load*, **ID/EX.Rd** = reg. destinazione se R

## **EX/MEM.ALU\_out**

- rs op rt      se R
- rs – rt      se beq, bne
- rs + offset    se lw/sw
- rs op imm     se arit/log con immediato

## **MEM/WB.ALU\_out** come EX/MEM.ALU\_out ma significativi solo se

- rs op rt      se R
- rs op imm se arit/log con immediato

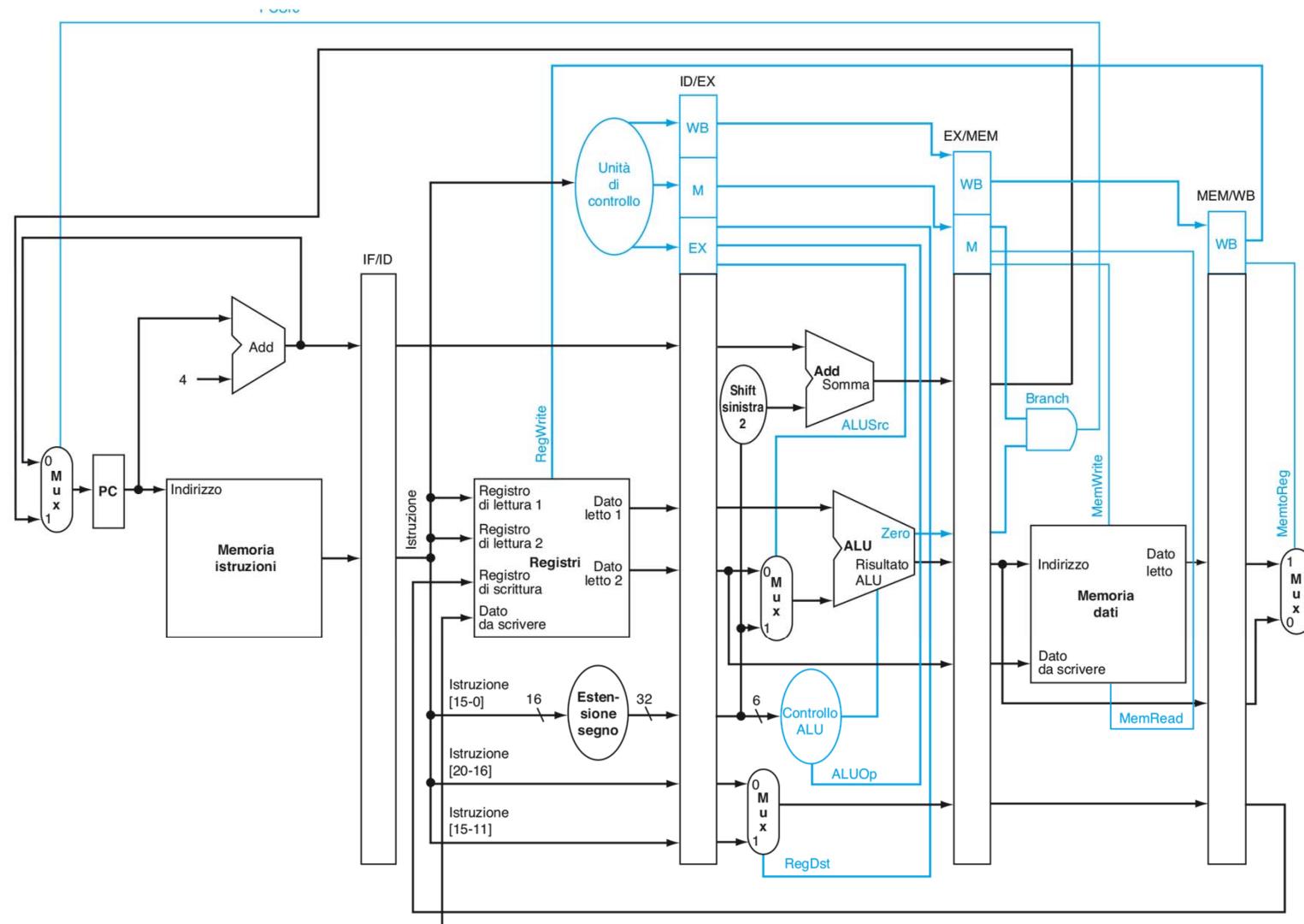
**EX/MEM.(Rt)** significativo solo in caso di *store*



POLITECNICO MILANO 1863

40

# Unità di elaborazione completa con segnali di controllo





**POLITECNICO**  
MILANO 1863

# Architettura dei calcolatori e sistemi operativi

## Pipelining e Hazard Capitolo 4 P&H

28. 10. 2015

# Problema dei conflitti

**Conflitti strutturali:** tentativo di usare la stessa risorsa da parte di diverse istruzioni in modi diversi nello stesso intervallo di tempo:

- Esempio: se avessimo un'unica memoria per istruzioni e dati

**Conflitti sui dati:** tentativo di utilizzare un risultato prima che sia pronto

- Esempio: istruzione che dipende dal risultato di un'istruzione precedente che è ancora nella pipeline

**Conflitti sul controllo:** tentativo di prendere una decisione sulla prossima istruzione da eseguire prima che la condizione sia valutata

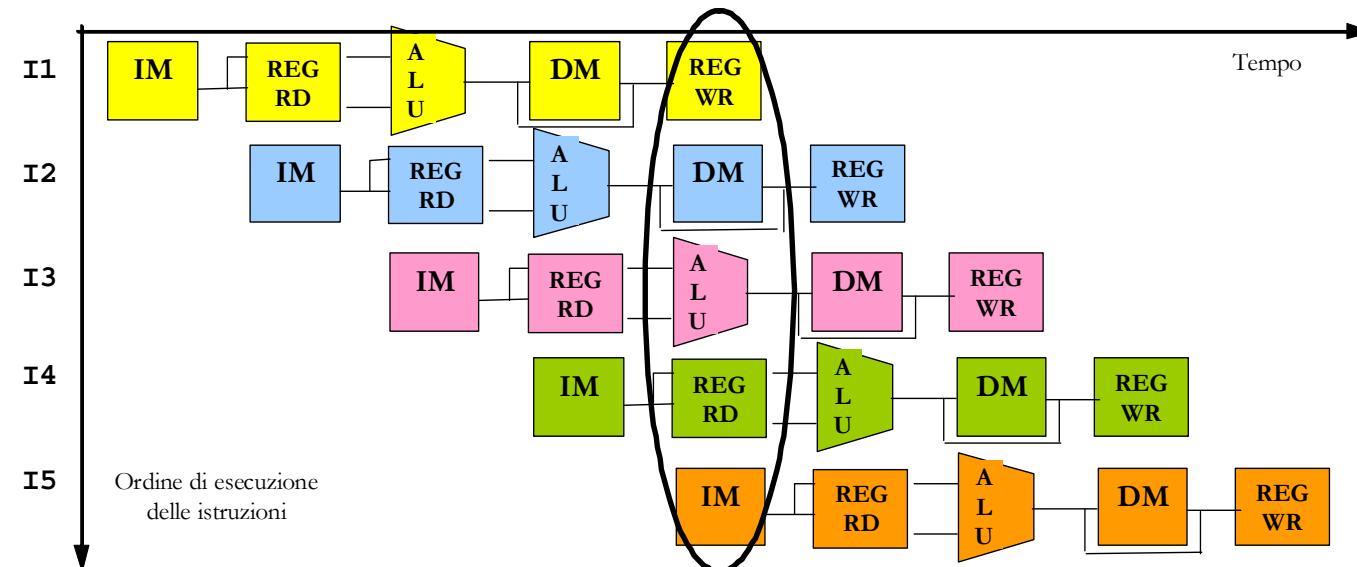
- Esempio: istruzioni di salto condizionato



# Problema dei conflitti strutturali

Nell'architettura MIPS **non** abbiamo conflitti strutturali:

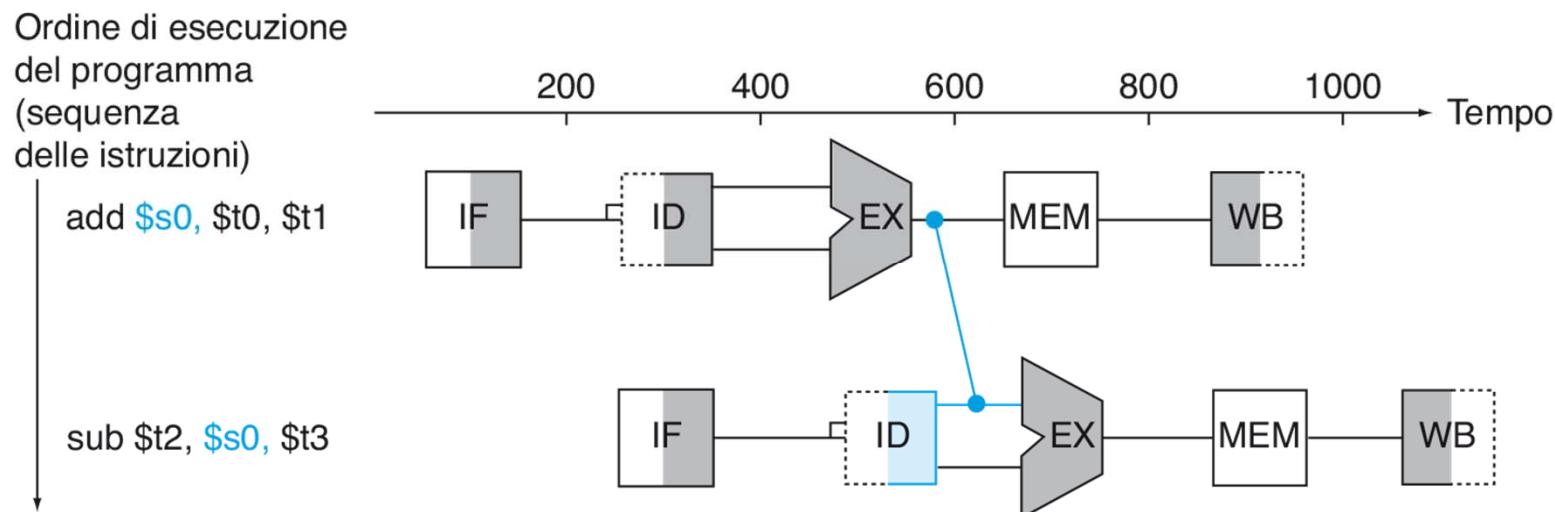
- Memoria Istruzioni separata dalla Memoria Dati
- Banco di Registri usato nello stesso ciclo di clock in lettura e scrittura ma in modo coerente con la tecnica di temporizzazione



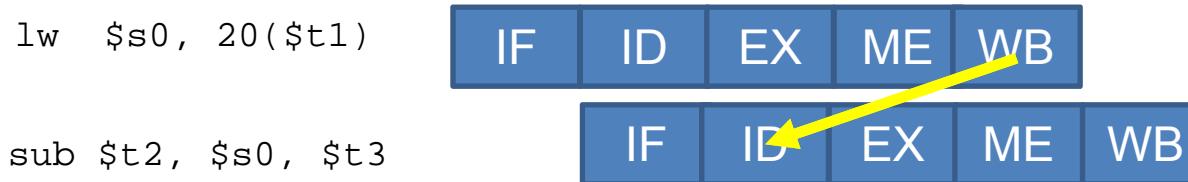
# Conflitto di dati (*data hazard*): R/R



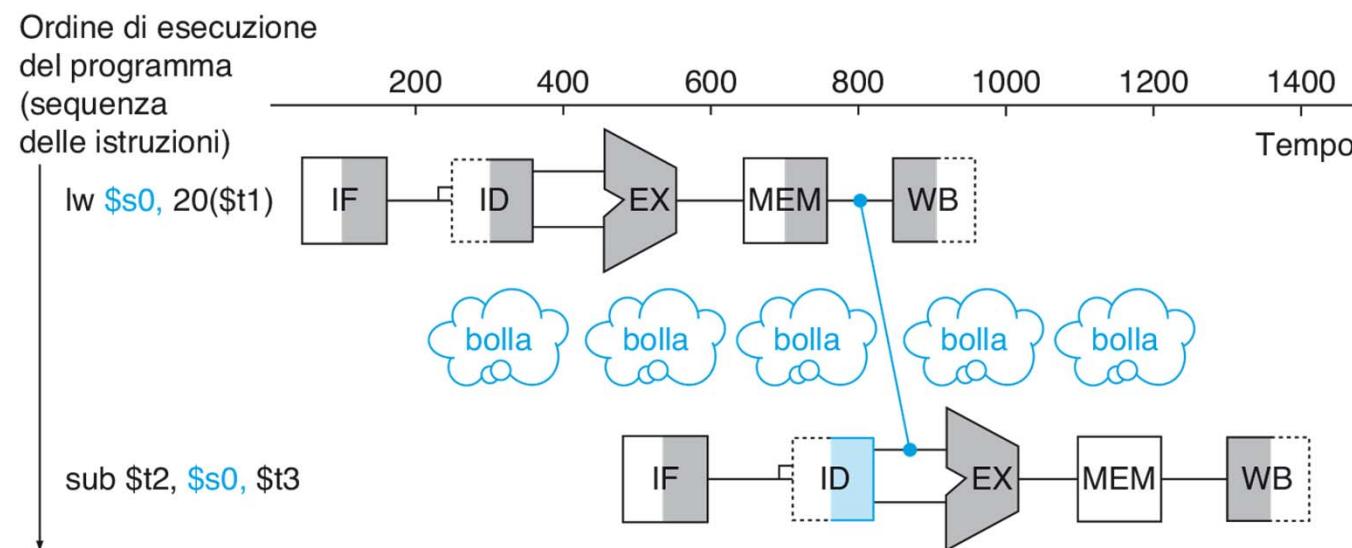
- il valore corretto è disponibile già all'uscita dello stadio EX di add
- circuito di *Propagazione/bypassing*: l'uscita dello stadio EX viene anche portata direttamente ai suoi ingressi (propagazione EX/EX)



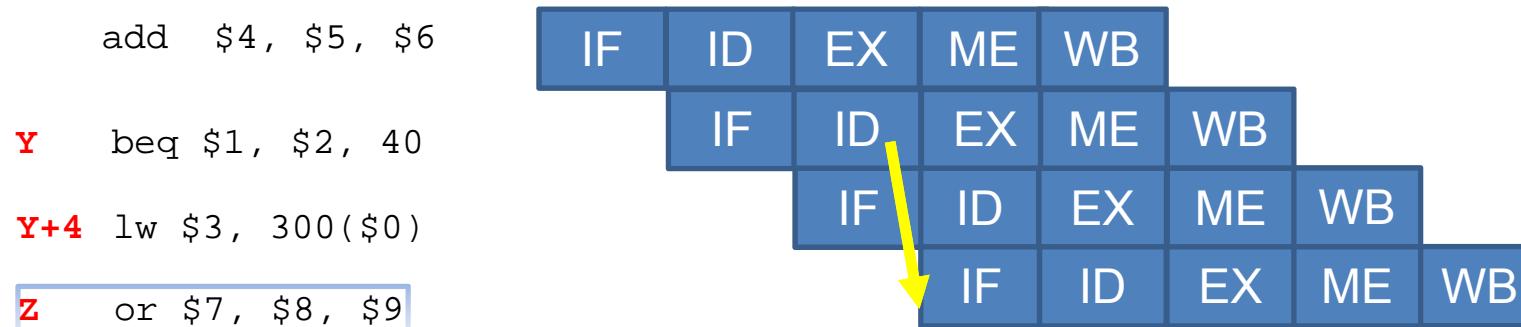
# Conflitto di dati (*data hazard*): Load/R



- il valore corretto è disponibile già all'uscita dello stadio **MEM** di `lw`
- circuito di *Propagazione/bypassing*: l'uscita dello stadio **MEM** viene anche portata direttamente agli ingressi dello stadio **EX** (MEM/EX), ma non basta .....
- si deve inserire un ciclo di ritardo nell'esecuzione di `sub`



# Conflitto di controllo (*control hazard*): *beq*



## Pipeline ottimizzata

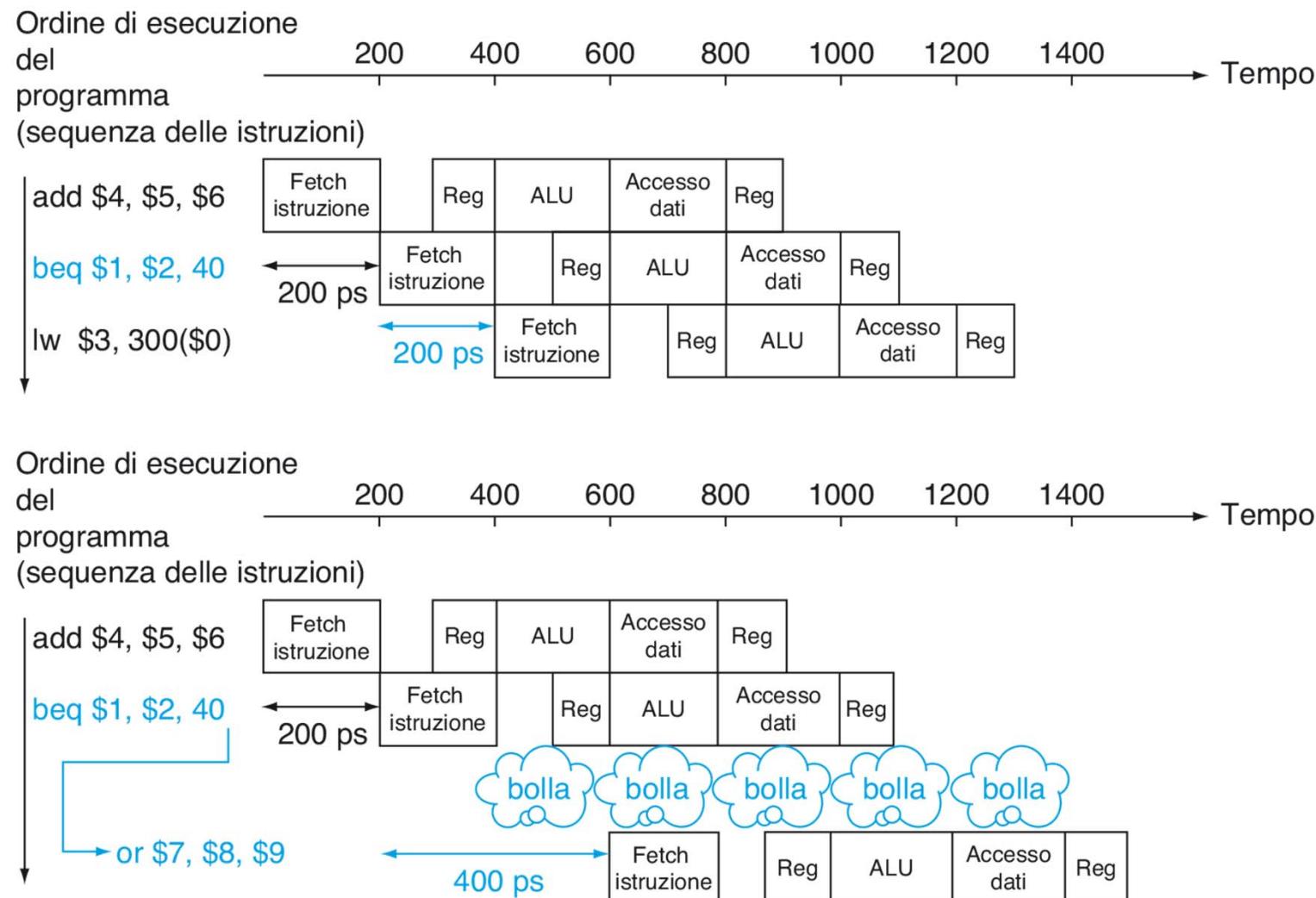
- esito confronto registri e calcolo indirizzo destinazione di salto disponibili alla fine della fase di ID

ma non basta ....

- se **branch untaken** la pipeline lavora “a pieno regime”: esecuzione dell’istruzione in sequenza a Y+4
- se **branch taken** si deve inserire un ciclo di ritardo prima di eseguire l’istruzione destinazione di salto per avere il PC corretto (Z)  
se esito e indirizzo salto disponibili alla fine della fase di EX, allora due cicli di ritardo



# Conflitto di controllo (*control hazard*): *beq* (cont.)



# Problema del conflitto di dati

Abbiamo visto che se le istruzioni eseguite nella pipeline sono **dipendenti** tra loro possono nascere problemi dovuti a **conflitti di dati**

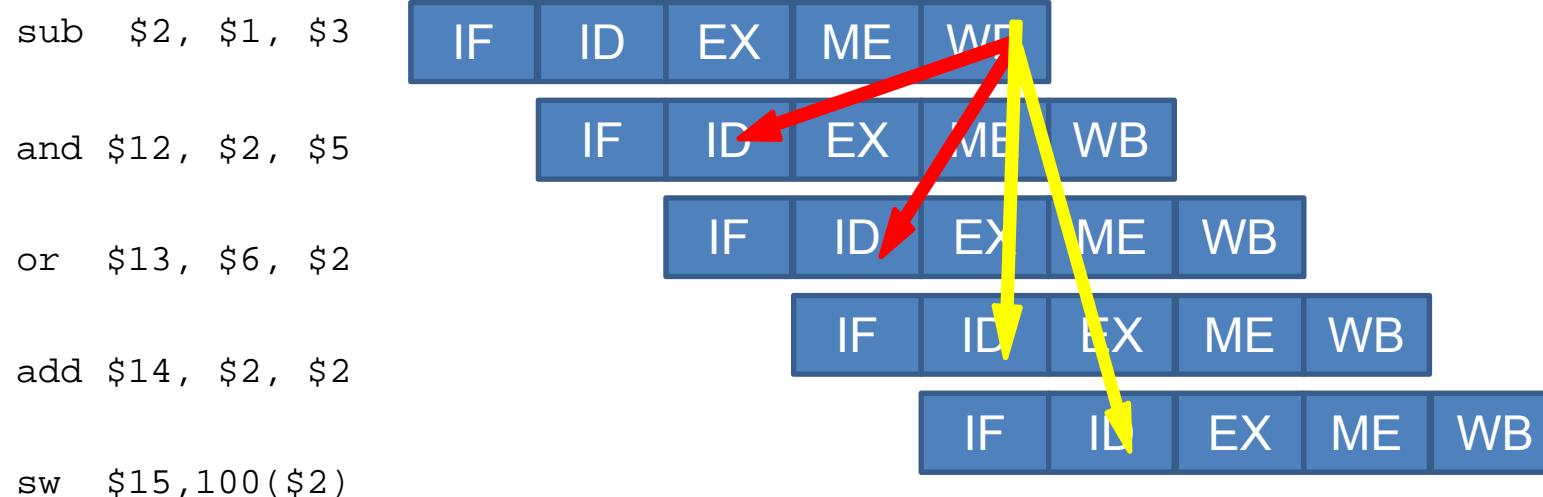
Esempio: le ultime quattro istruzioni dipendono dal risultato scritto nel registro **\$2** dalla prima istruzione, ma solo le ultime due istruzioni accedono al valore corretto:

```
sub $2, $1, $3    # Reg. $2 scritto dalla istruzione sub  
and $12, $2, $5  # Il 1° operando($2) dipende dalla sub  
or $13, $6, $2   # Il 2° operando($2) dipende dalla sub  
add $14, $2, $2   # 1° ($2) & 2° ($2) dipendono dalla sub  
sw $15,100($2)  # Il reg. indice($2) dipende dalla sub
```

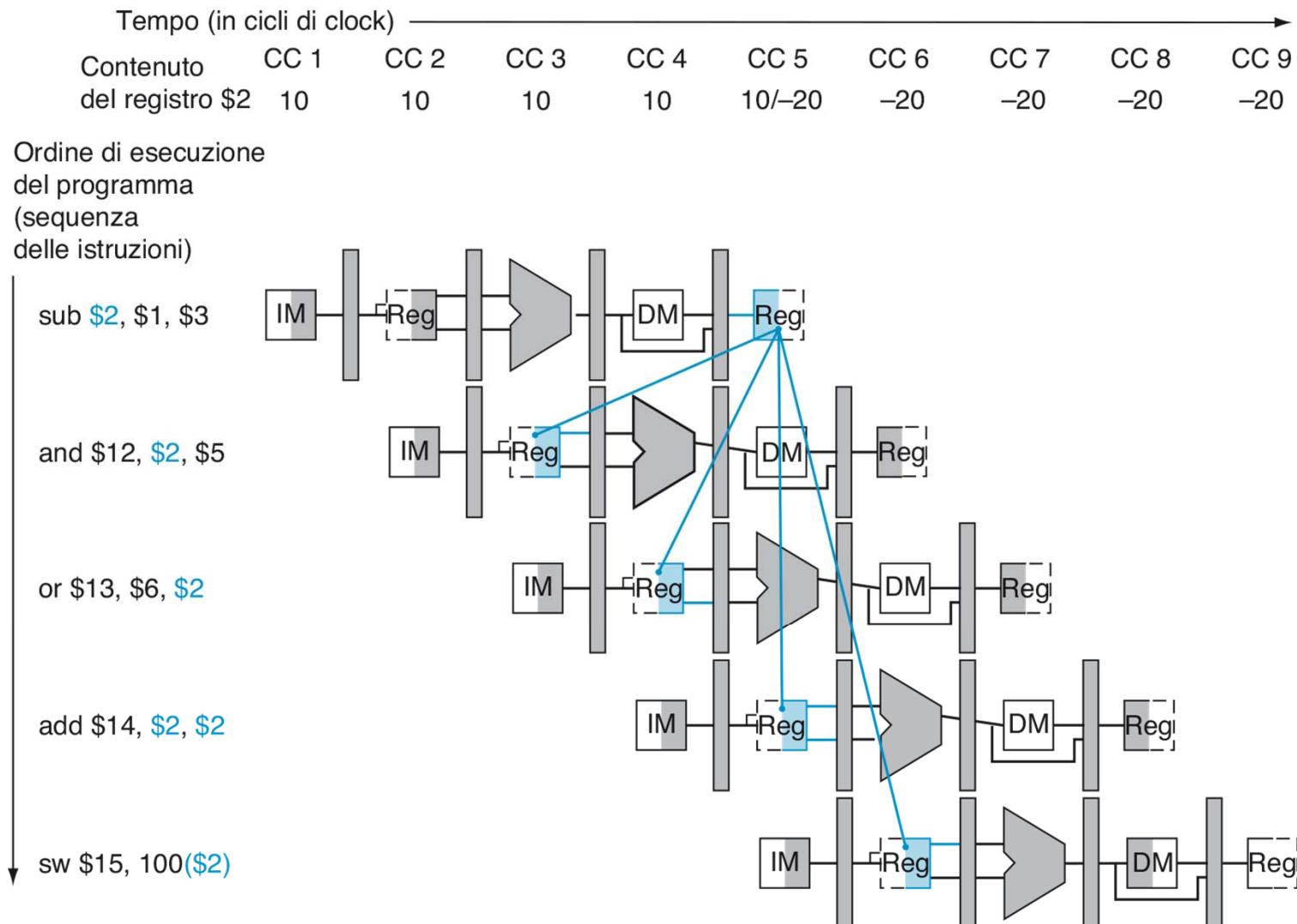


# Dipendenze di dato e stadi

Le dipendenze di dato diventano **conflitti** se “vanno all’indietro nel tempo”



# Dipendenze di dato, risorse e registro coinvolto



# Soluzioni al problema del conflitto di dati

## Tecniche di compilazione del codice:

- Inserimento di istruzioni `nop` (*no operation*).
- *Scheduling* o riordino delle istruzioni in modo da impedire che istruzioni correlate siano troppo vicine.
  - Il compilatore cerca di inserire tra le istruzioni correlate (che presentano dei conflitti) delle istruzioni *indipendenti* dal risultato delle precedenti operazioni, facendo così scomparire tutti i conflitti.
  - Quando il compilatore non riesce a trovare istruzioni indipendenti deve inserire istruzioni `nop`.

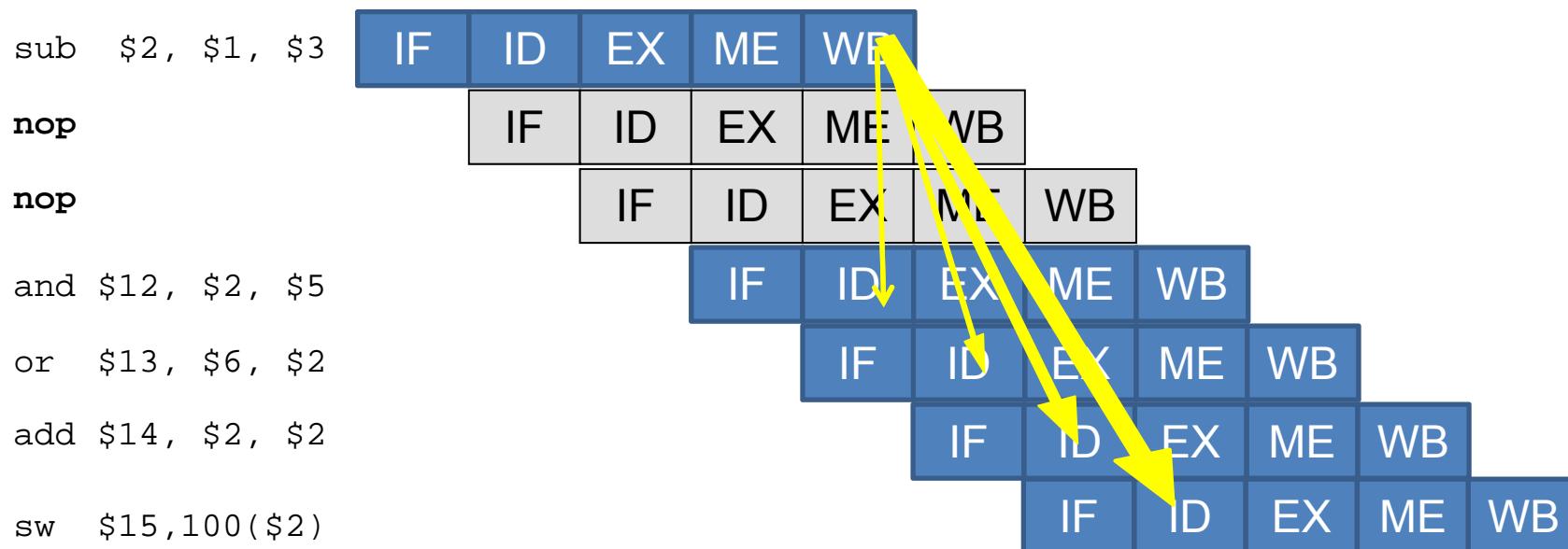
## Tecniche di tipo hardware:

- Inserimento di “bolle” o stalli nella pipeline.
- Propagazione dei dati in avanti i dati (*forwarding o bypassing*)



# Inserimento di *nop*: Esempio

Il compilatore deve inserire tra le istruzioni **sub** e **and** due istruzioni *nop*, facendo così scomparire tutti i conflitti.



# Scheduling: Esempio

Esempio:

sub      **\$2**, \$1, \$3

and      \$12, **\$2**, \$5

or      \$13, \$6, **\$2**

add      \$14, **\$2**, **\$2**

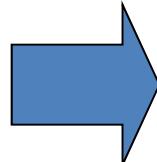
sw      \$15,100(**\$2**)

add      **\$4**, \$10, \$11

and      **\$7**, \$8, **\$9**

lw      \$16, 100(\$18)

lw      \$17, 200(\$19)



sub      **\$2**, \$1, \$3

add      **\$4**, \$10, \$11

and      **\$7**, \$8, **\$9**

and      \$12, **\$2**, \$5

or      \$13, \$6, **\$2**

add      \$14, **\$2**, **\$2**

sw      \$15,100(**\$2**)

lw      \$16, 100(\$18)

lw      \$17, 200(\$19)



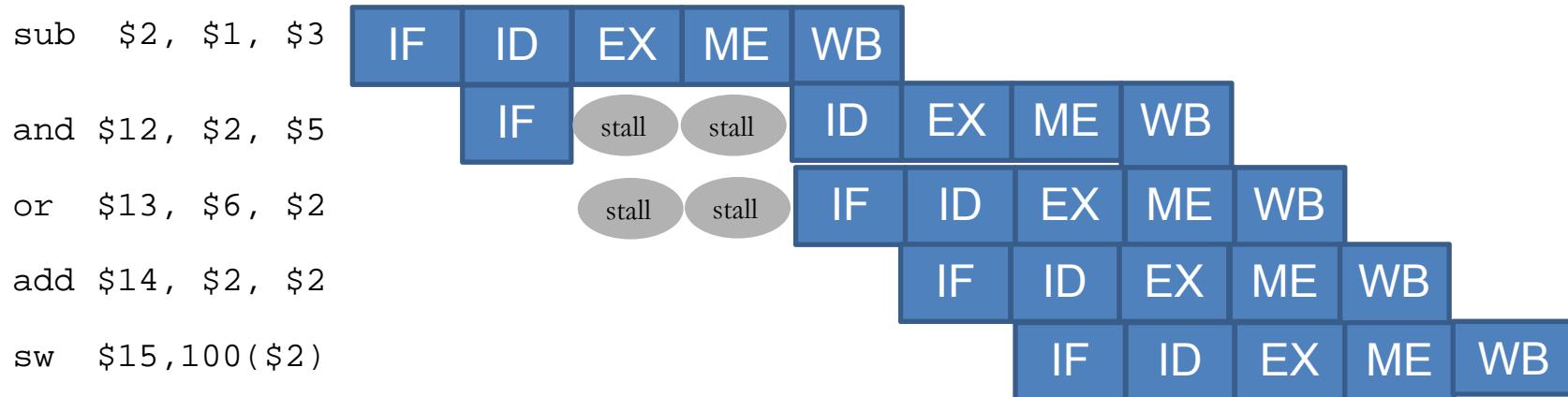
# Inserimento di bolle o stalli: Esempio

Una possibile soluzione di tipo *hardware* consiste nell'inserire delle “bolle” nella pipeline, cioè *bloccare il flusso di ingresso di istruzioni* nella pipeline fino a quando il conflitto non è risolto. Lo stato in cui si trova la *CPU* quando le istruzioni sono bloccate è indicato con il termine “stallo”.

Nell'esempio significa inserire 2 bolle o stalli per fermare le istruzioni che seguono l'istruzione **sub** affinché possano essere letti i dati corretti.



# Inserimento di bolle o stalli: Esempio



- Lo stallo di una istruzione (in questo caso **and**) viene “inserito graficamente” prima dello **stadio ID** che rileva il conflitto (vedremo come) e che **non può completare la scrittura del registro *ID/EX*** di uscita dello stadio
- Lo **stadio ID** dovrà essere “rieseguito” per la **and** al termine del conflitto: ciò significa che i valori nel registro ***IF/ID*** di ingresso allo stadio devono essere “congelati” fino al termine del conflitto
- Di conseguenza l’istruzione **or** di fatto non può completare la fase ***IF*** e quindi **non può scrivere in *IF/ID***. Il valore del **PC** di **or** rimane “congelato” fino al termine del conflitto



# Inserimento di bolle o stalli: Esempio

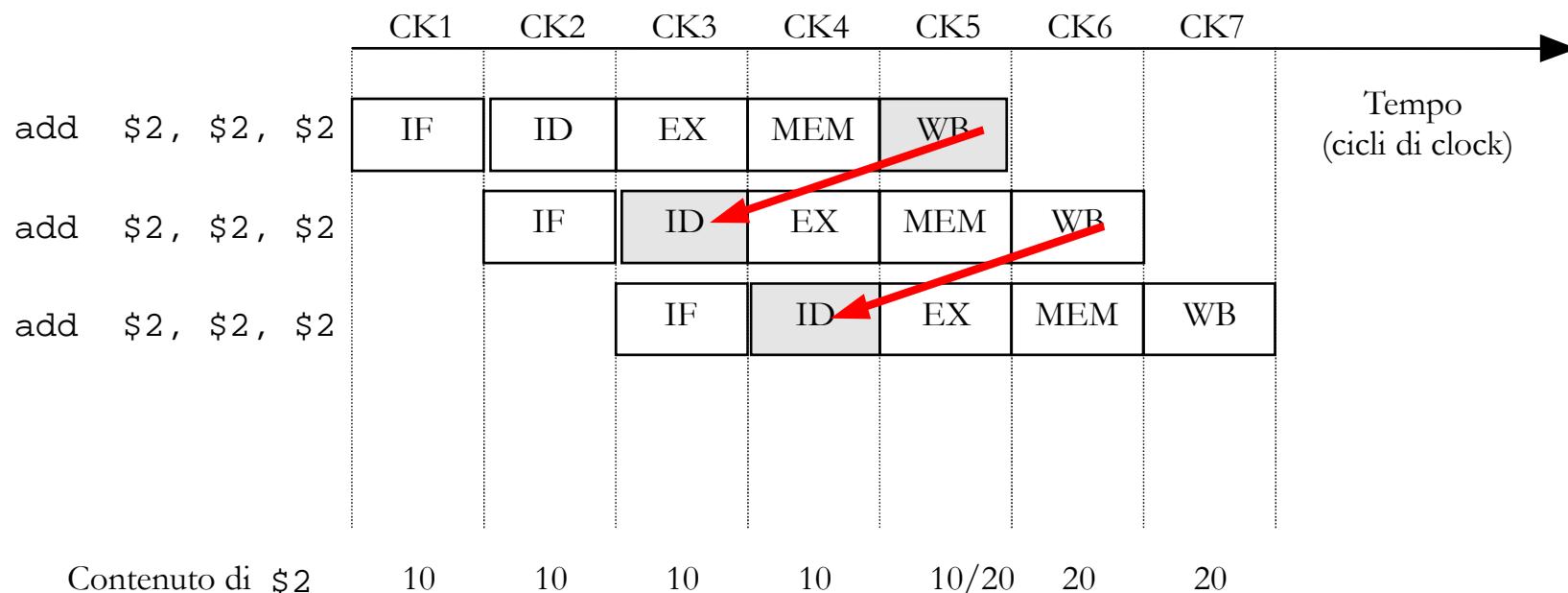
Purché il contenuto dei registri *PC* e *IF/ID* vengano conservati, l'istruzione nello stadio *IF* continuerà ad essere letta utilizzando lo stesso *PC*, ed i registri nello stadio *ID* continueranno ad essere letti utilizzando la stessa istruzione nel registro di pipeline *IF/ID*.

L'istruzione **and** resta nel registro di pipeline *IF/ID* per 2 cicli di clock, mentre vengono generate 2 bolle separate nella pipeline.

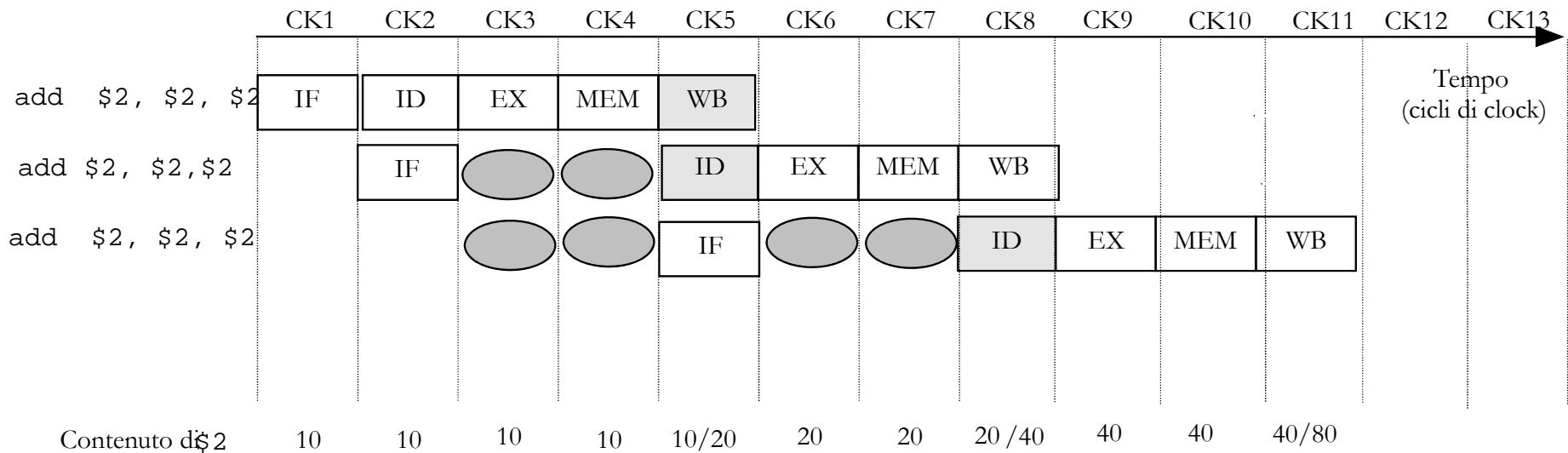


# Problema del conflitto di dati: un altro esempio

L1: add \$2, \$2, \$2 # Reg. \$2 scritto dall'istruz. L1  
L2: add \$2, \$2, \$2 # 1° e 2° operando dipendono dalla L1  
L3: add \$2, \$2, \$2 # 1° e 2° operando dipendono dalla L2

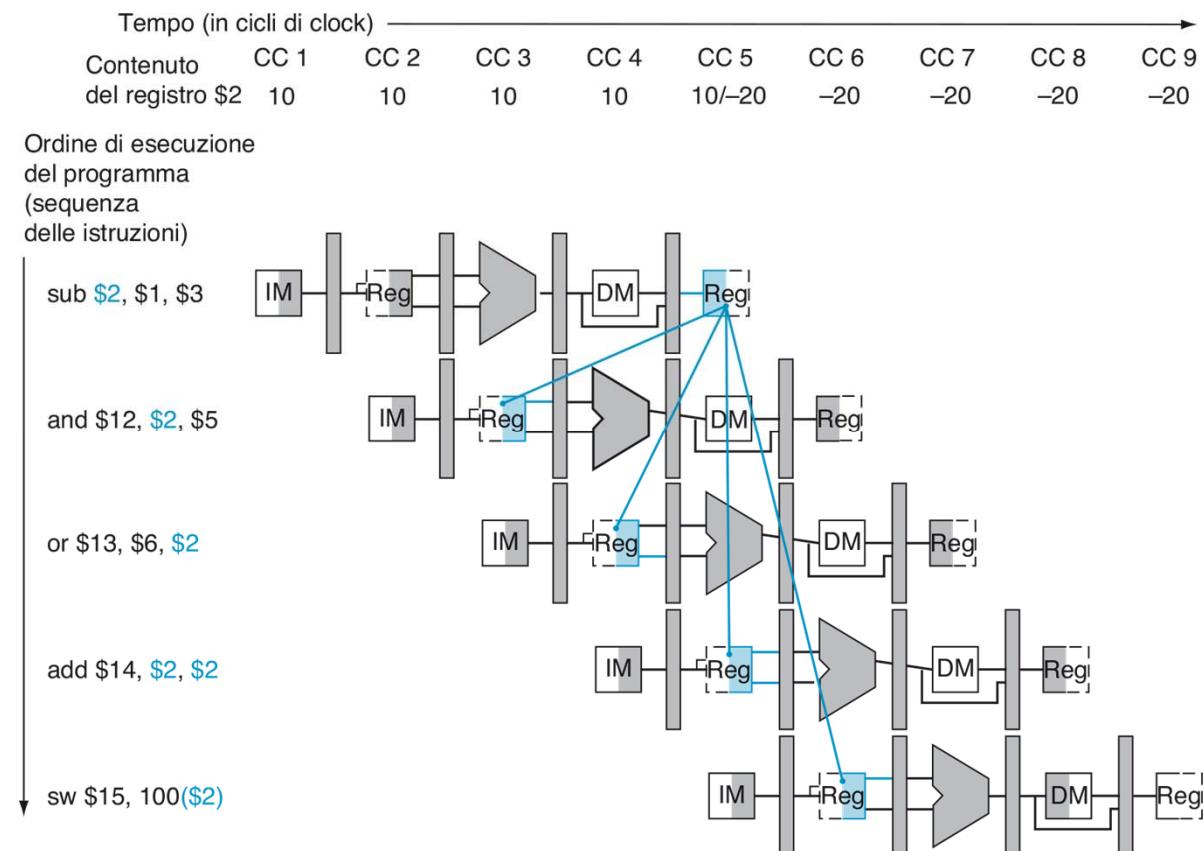


# Problema del conflitto di dati: esecuzione corretta



# Forwarding

Un'altra possibile soluzione *hardware* consiste nella **propagazione in avanti dei dati**



Il risultato dell'istruzione **sub** è disponibile alla fine del ciclo di clock 3 (EX)

**and e or** hanno bisogno del dato all'inizio dello stadio di EX, ossia nei cicli di clock 4 e 5 rispettivamente

→ si può eseguire il codice senza stalli semplicemente proagando il dato non appena disponibile a qualsiasi unità che lo richieda prima che venga scritto nel registro destinazione



# Forwarding

- Il valore richiesto dall'istruzione **and**, è già disponibile nel registro di pipeline *EX/MEM* dell'istruzione **sub**
- Analogamente al ciclo di clock successivo, l'ingresso dell'*ALU* per l'istruzione **or** si trova nel registro di pipeline *MEM/WB* dell'istruzione **sub**
- E' possibile fornire all'*ALU* gli ingressi richiesti dalle istruzioni **and** e **or** propagando in avanti i risultati che si trovano nei registri di pipeline, senza aspettare che siano scritti nel banco di registri.

Questa tecnica, che utilizza risultati temporanei invece di attendere che i registri siano scritti, si chiama **propagazione in avanti dei risultati (forwarding) o bypassing**

- Aggiungendo **multiplexer** agli ingressi dell'*ALU* in modo da prelevare gli ingressi da qualsiasi registro di pipeline e non solo dal registro *ID/EX*, la pipeline può procedere senza stalli anche in presenza di conflitti di dati.



# Propagazione di dato

Quando un'istruzione nel suo stadio EX deve utilizzare un dato di un registro che non è ancora stato scritto da un'istruzione precedente nella sua fase di WB è necessario portare il dato all'ingresso corretto della ALU.

***Copie di condizioni che generano un conflitto di dato:***

- 1a. EX/MEM.RegistroRd = ID/EX.RegistroRs
- 1b. EX/MEM.RegistroRd = ID/EX.RegistroRt



*hazard nello  
stadio EX*

- 2a. MEM/WB.RegistroRd = ID/EX.RegistroRs
- 2b. MEM/WB.RegistroRd = ID/EX.RegistroRt

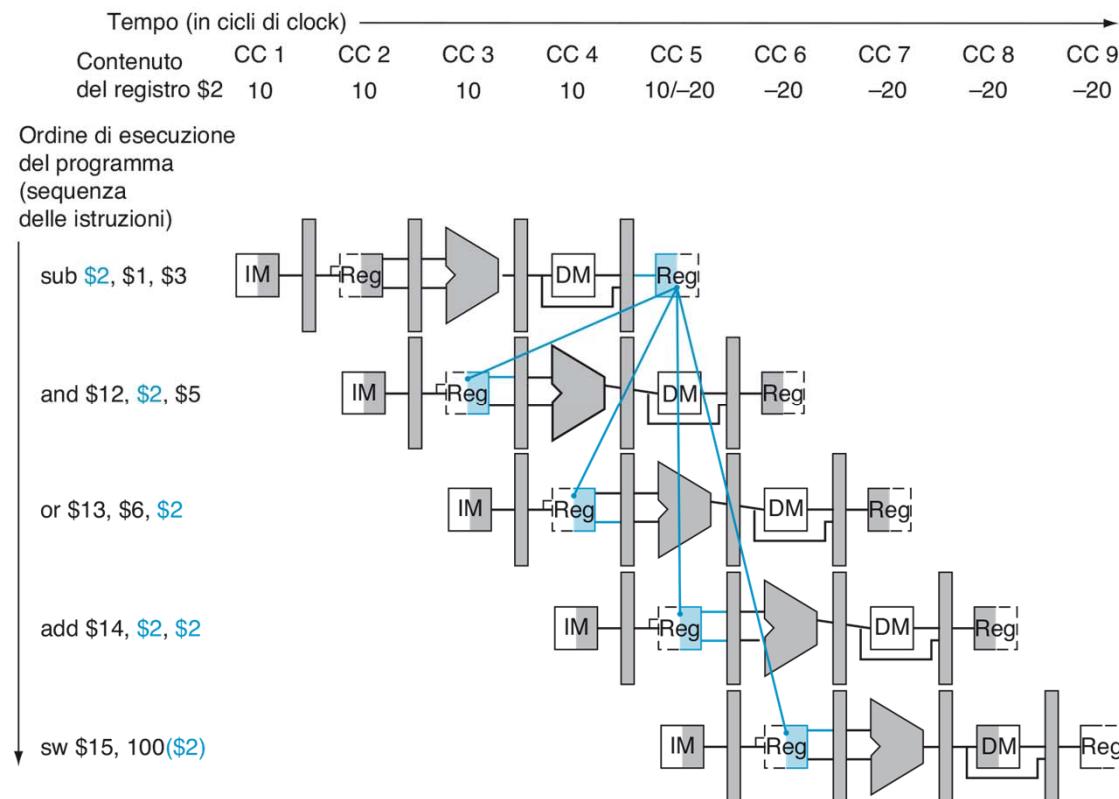


*hazard nello  
stadio MEM*



POLITECNICO MILANO 1863

# Conflitti di dato: esempio



## Implementazione

- prendere gli ingressi della ALU non solo dal registro ID/EX ma anche dagli altri registri di pipeline EX/MEM e MEM/WB per propagare i dati corretti
- aggiungere dei mux agli ingressi della ALU e i corrispondenti segnali di controllo
- la propagazione avviene solo se il **segnale RegWrite è asserito nello stadio del conflitto** (da EX/MEM in avanti)

## Conflitto sub – and: tipo 1a

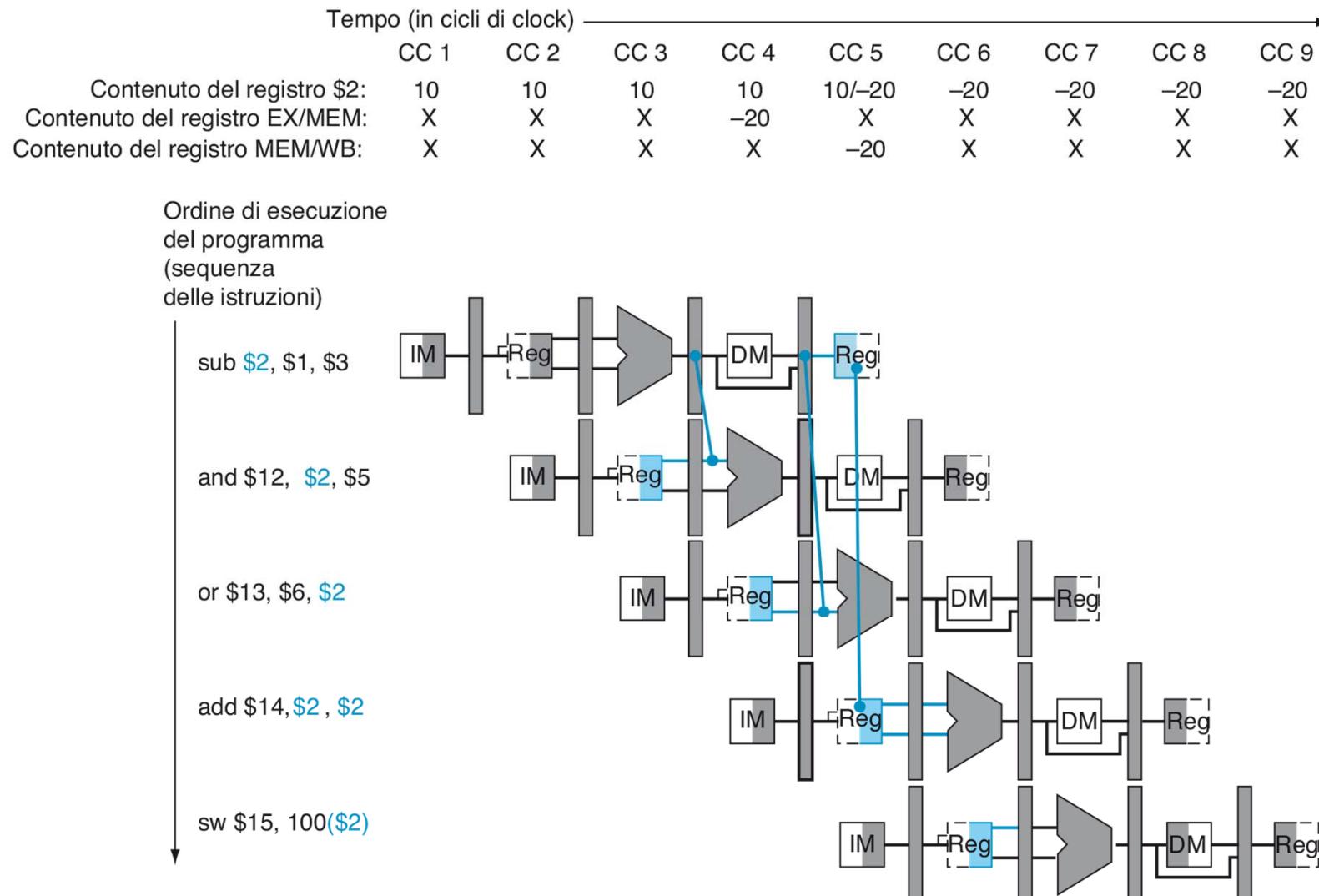
$\text{EX/EM.RegistroRd} = \text{ID/EX.RegistroRs} = \$2 \ \&\ \& \text{EX/EM.RegistroRd} \neq 0$

## Conflitto sub – or: tipo 2b

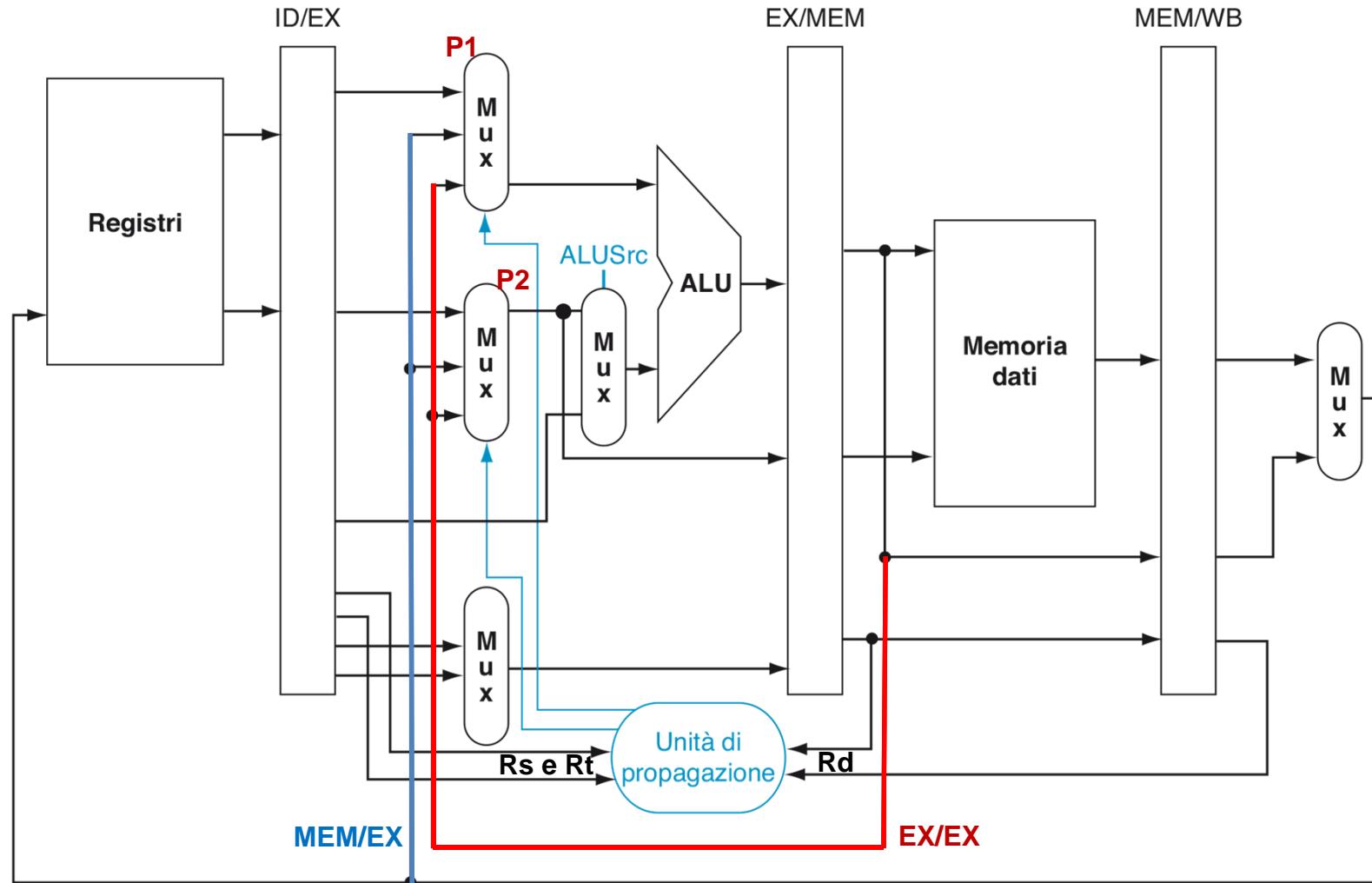
$\text{MEM/WB.RegistroRd} = \text{ID/EX.RegistroRt} = \$2 \ \&\ \& \text{MEM/WB.RegistroRd} \neq 0$



# Forwarding: esempio



# Pipeline con hardware di propagazione per istruzioni di tipo R



## Condizioni per rilevare i conflitti e *propagare* dallo stadio EX e dallo stadio MEM

If (EX/MEM.RegWrite and EX/MEM.RegistroRd ≠ 0) and  
(EX/MEM.RegistroRd = ID/EX.RegistroRs)) **Propaga** = 10

**MUX P1**

If (EX/MEM.RegWrite and EX/MEM.RegistroRd ≠ 0) and  
(EX/MEM.RegistroRd = ID/EX.RegistroRt)) **Propaga** = 10

**MUX P2**

If (MEM/WB.RegWrite and MEM/WB.RegistroRd ≠ 0) and  
(MEM/WB.RegistroRd = ID/EX.RegistroRs)) **Propaga** = 01

**MUX P1**

If (MEM/WB.RegWrite and MEM/WB.RegistroRd ≠ 0) and  
(MEM/WB.RegistroRd = ID/EX.RegistroRt)) **Propaga** = 01

**MUX P2**



# Segnali di controllo del multiplexer di propagazione

Controllo multiplexer	Sorgente	Spiegazione
PropagaA = 00	ID/EX	Il primo operando della ALU proviene dal register file.
PropagaA = 10	EX/MEM	Il primo operando della ALU viene propagato dal risultato della ALU nel ciclo di clock precedente.
PropagaA = 01	MEM/WB	Il primo operando della ALU viene propagato dalla memoria dati o da un precedente risultato della ALU.
PropagaB = 00	ID/EX	Il secondo operando della ALU proviene dal register file.
PropagaB = 10	EX/MEM	Il secondo operando della ALU viene propagato dal risultato della ALU nel ciclo di clock precedente.
PropagaB = 01	MEM/WB	Il secondo operando della ALU è propagato dalla memoria dati o da un precedente risultato della ALU.

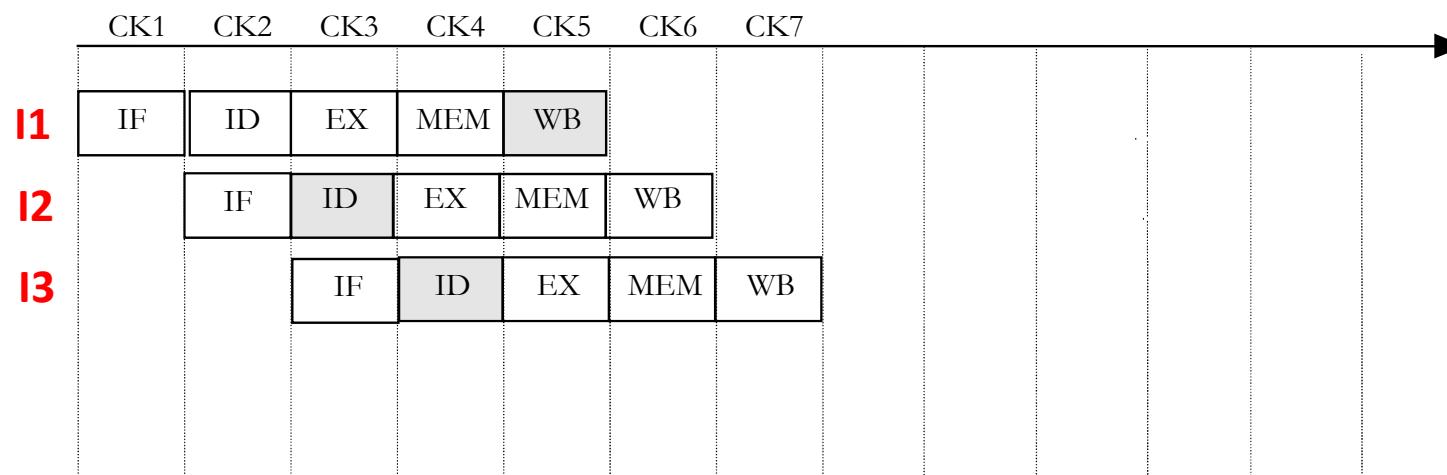


# Ancora sulla propagazione: il problema del valore “più recente”

I1 add \$1, \$1, \$2

I2 add \$1, \$1, \$3

I3 add \$1, \$1, \$4



a CK3 la condizione sotto è verificata e si propaga I1 per I2

If(EX/MEM.RegWrite and EX/MEM.RegistroRd ≠ 0) and  
(EX/MEM.RegistroRd = ID/EX.RegistroRt) Propaga = 10

*I1 per I2*

a CK4 sono verificate le due condizioni ma si deve propagare solo **I2 per I3** (valore più recente)

If (MEM/WB.RegWrite and MEM/WB.RegistroRd ≠ 0) and  
(MEM/WB.RegistroRd = ID/EX.RegistroRt) Propaga = 01

*I1 per I3*

If(EX/MEM.RegWrite and EX/MEM.RegistroRd ≠ 0) and  
(EX/MEM.RegistroRd = ID/EX.RegistroRt) Propaga = 10

*I2 per I3*



## Ancora sulla propagazione: propagazione dallo stadio MEM

If (MEM/WB.RegWrite and (MEM/WB.RegistroRd ≠ 0)

and not ((EX/MEM.RegWrite and (EX/MEM.RegistroRd ≠ 0)) and  
(EX/MEM.RegistroRd = ID/EX.RegistroRs))

and (MEM/WB.RegistroRd = ID/EX.RegistroRs)) Propaga = 01

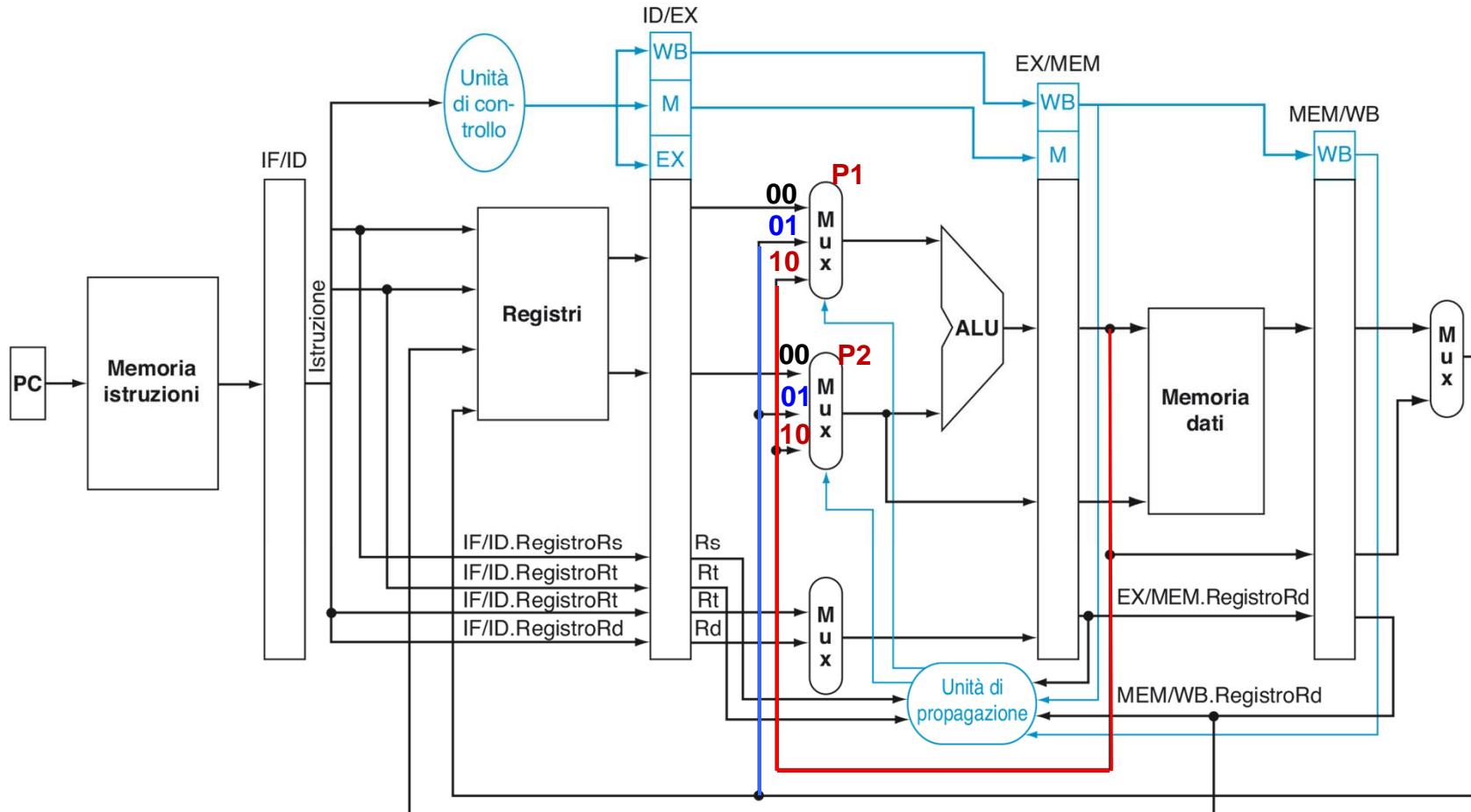
If (MEM/WB.RegWrite and (MEM/WB.RegistroRd ≠ 0)

and not ((EX/MEM.RegWrite and (EX/MEM.RegistroRd ≠ 0)) and  
(EX/MEM.RegistroRd = ID/EX.RegistroRt))

and (MEM/WB.RegistroRd = ID/EX.RegistroRt)) Propaga = 01

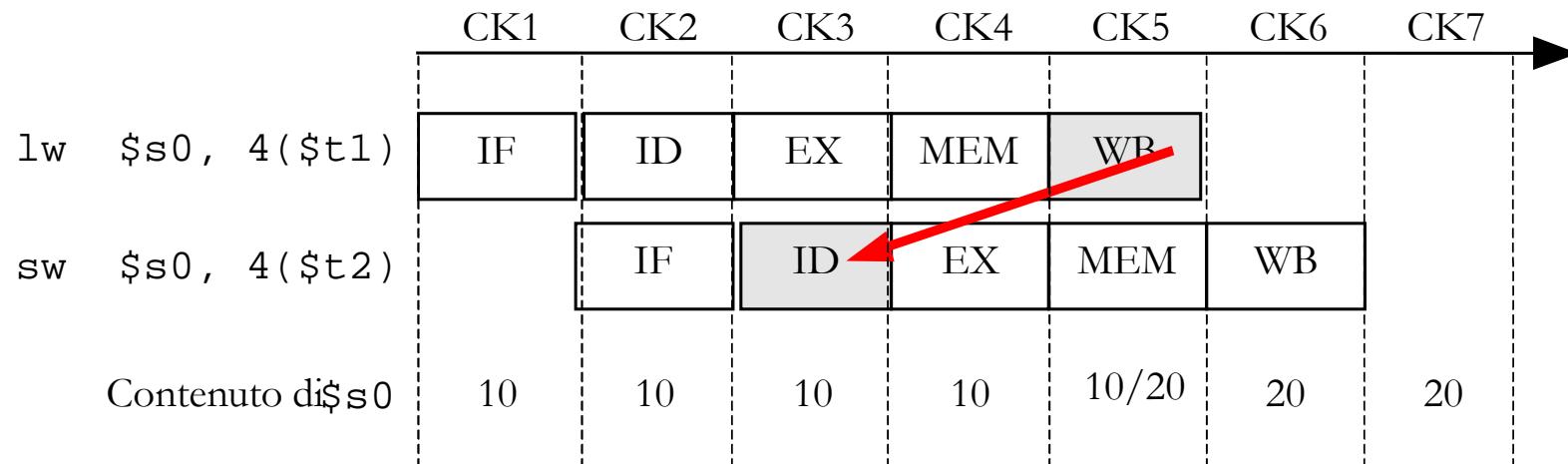


# Pipeline con unità di risoluzione dei conflitti di dato tramite propagazione



# Problema del conflitto di dati: load/store

```
L1: lw $s0, 4($t1) # $s0 <- M[4 + $t1]  
L2: sw $s0, 4($t2) # M[4 + $t2] <- $s0
```



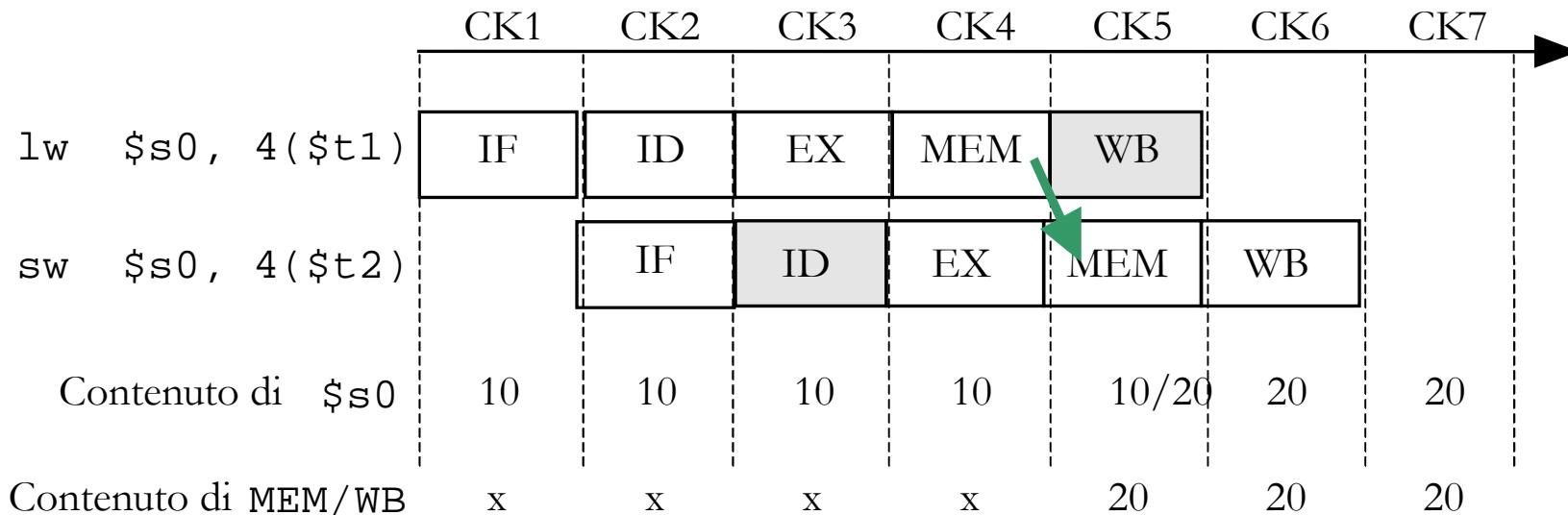
- Senza forwarding : Necessari 2 stalli



# Problema del conflitto di dati: load/store

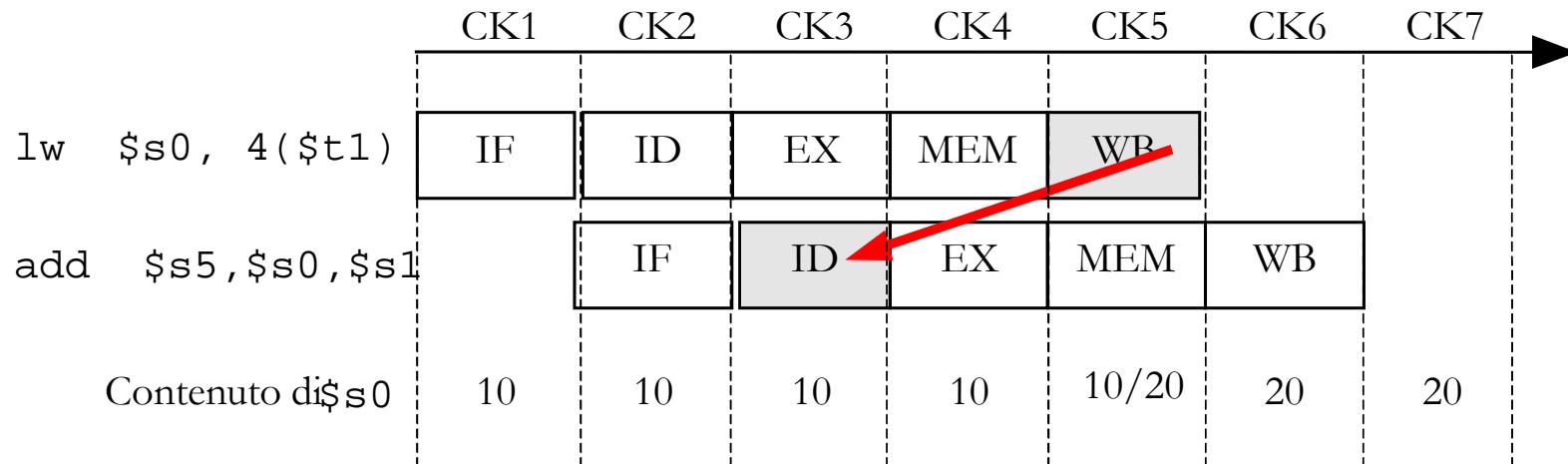
Con forwarding: Nessuno stallo

Devo usare un percorso di forwarding (**MEM/MEM**) per portare il risultato della *load* (*dato letto*) che è in MEM/WB all'ingresso *dato da scrivere* della memoria per effettuare la *store* e quindi devo aggiungere un multiplexer



# Hazard sui dati e stallo: esempio

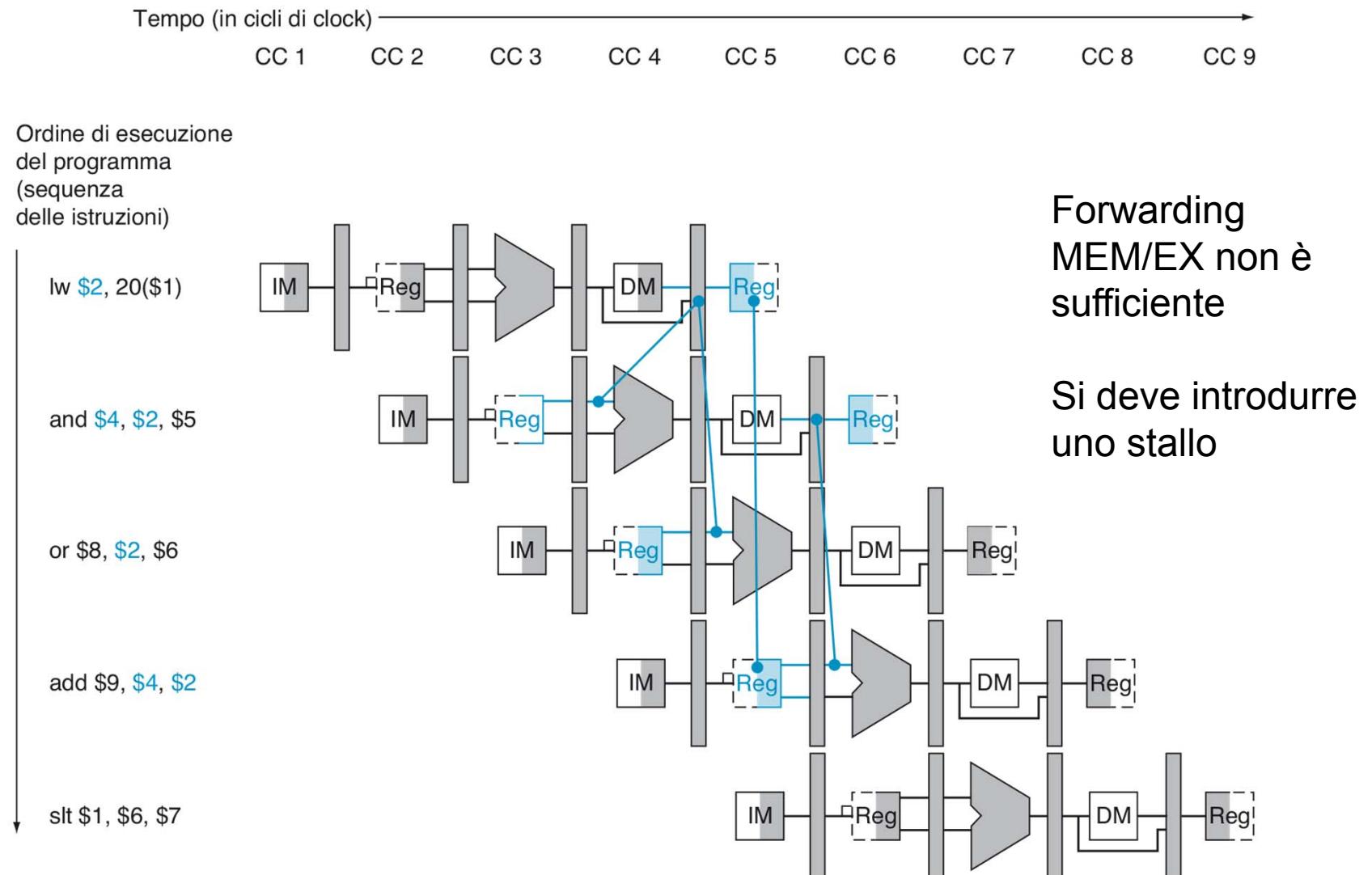
```
L1: lw      $s0, 4($t1)    # $s0 <- M [4 + $t1]  
L2: add    $s5, $s0, $s1 # Il 1° operando dipende dalla L1
```



- Senza forwarding : Necessari 2 stalli



# Hazard sui dati e **stallo**: load/use



# Conflitto di dato *load/use*

In aggiunta all'unità di propagazione è necessaria una unità di rilevamento dei conflitti che durante lo stadio di ID possa inserire uno stallo tra la lettura del dato e il suo utilizzo

Il conflitto viene rilevato utilizzando ID/EX di ***load*** e IF/ID di ***use***

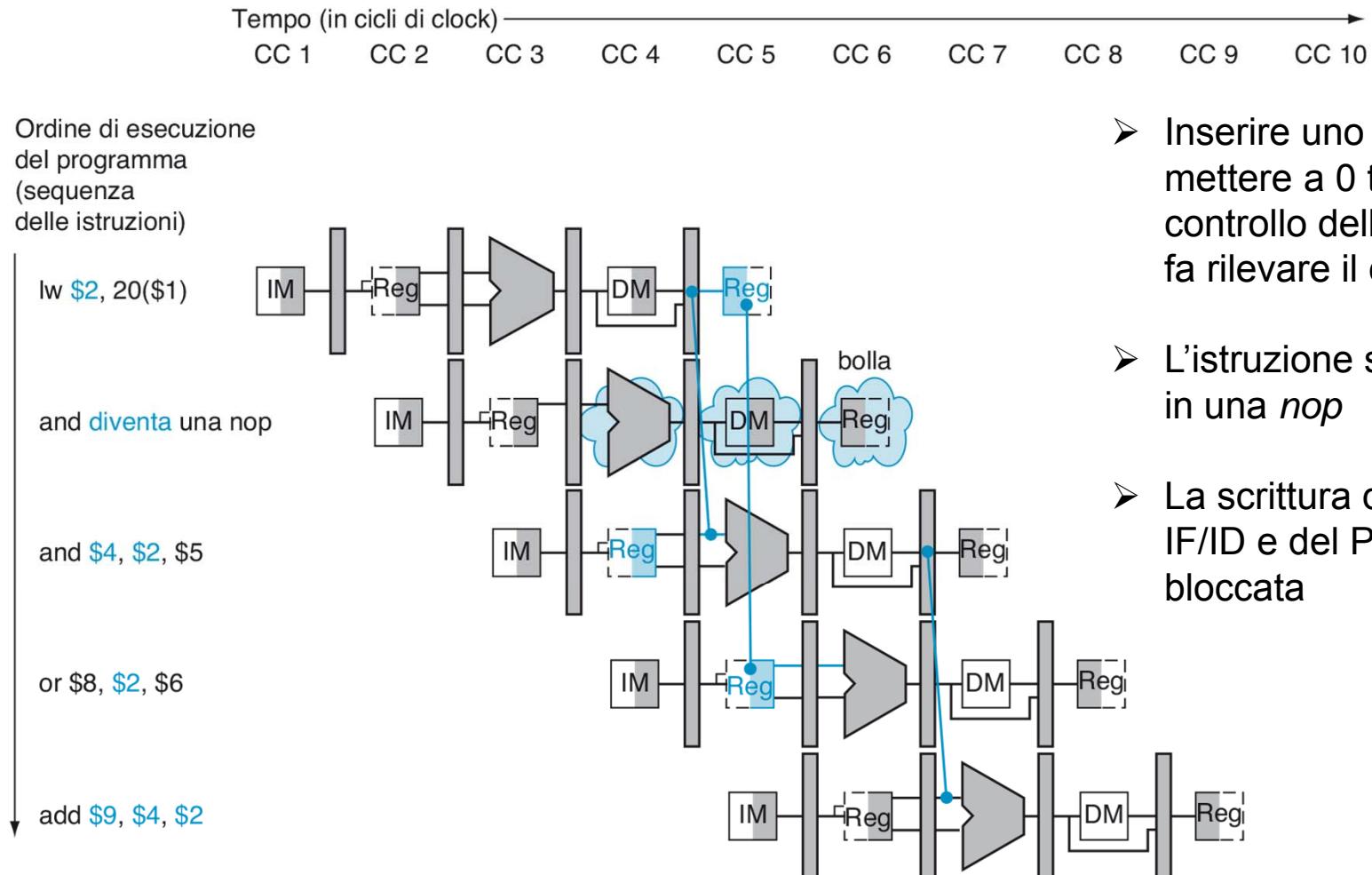


If (ID/EX.MemRead and ((ID/EX.RegistroRt = IF/ID.RegistroRs)  
or (ID/EX.RegistroRt = IF/ID.RegistroRt)))

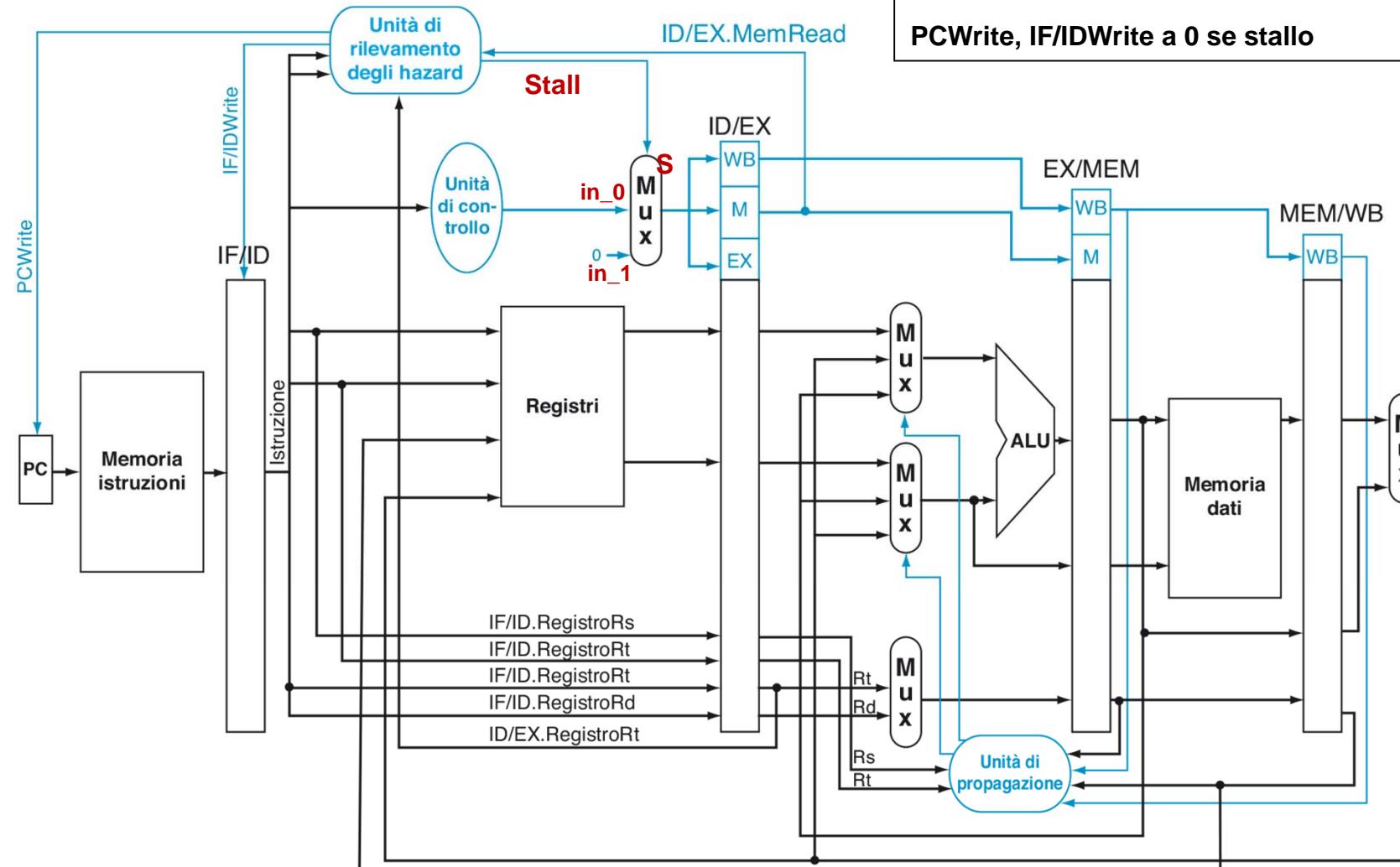
- Metti in **stallo** la pipeline per un ciclo di clock
- Successivamente la dipendenza di dato è gestita dalla **propagazione** da MEM/WB a ID/EX (**MEM/EX**)



# Modalità di inserimento degli stalli



# Controllo della pipeline



Stall a 1 se necessario stallo, 0 altrimenti

**MUX\_S**

**out = in\_1 , segnali di controllo a 0 (stallo)**  
**out = in\_0, segnali di controllo effettivi**

**PCWrite, IF/IDWrite a 0 se stallo**



# Problema del conflitto di controllo

Per alimentare la pipeline si deve prelevare un'istruzione ad ogni ciclo di clock, però la decisione relativa al salto condizionato non viene presa fino allo stadio *MEM*.

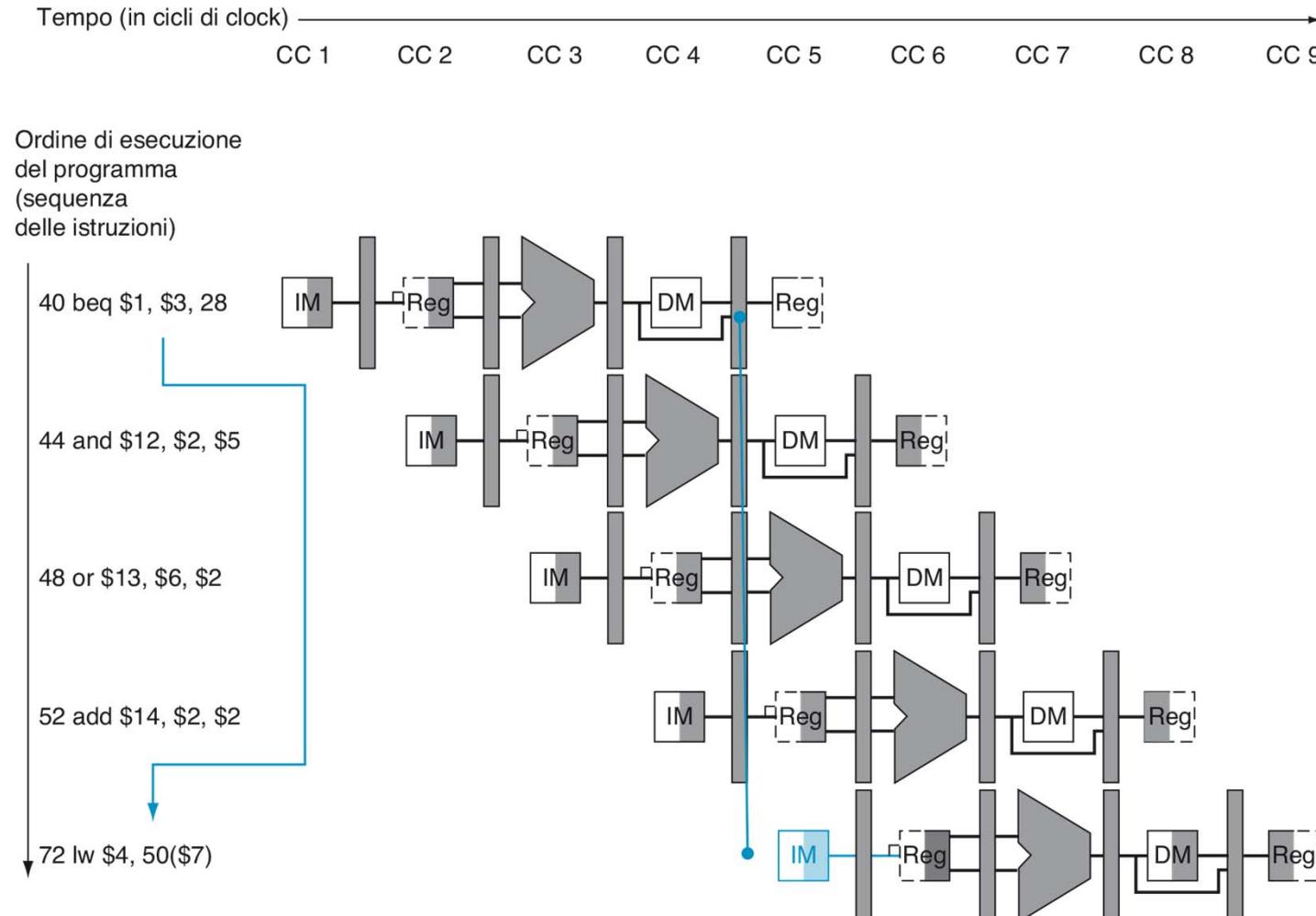
La logica di controllo del salto viene determinata nello stadio *MEM* e - sul fronte di salita del ciclo corrispondente - il PC viene scritto con l'indirizzo destinazione di salto

Questo ritardo nel determinare l'istruzione corretta da prelevare viene chiamato **conflitto di controllo o conflitto di salto condizionato**.

Questi conflitti sono statisticamente meno frequenti dei conflitti sui dati.



# Il problema dei salti condizionati



# Soluzione al conflitto di controllo - 1

Utilizzo della **predizione** nella gestione dei salti condizionati

- Approccio più semplice: predire sempre che il salto venga **non eseguito** (untaken branch)
  - Si verifica uno stallo quando il salto deve essere eseguito e bisogna scartare le istruzioni già nella pipeline (ponendo a 0 i segnali di controllo)

Ridurre i ritardi associati ai salti condizionati, per avere una sola istruzione da scartare: **pipeline ottimizzata**

- spostare la decisione del salto da MEM a uno stadio precedente
  - Richiede di anticipare il calcolo dell'indirizzo di destinazione del salto e la valutazione del confronto sulla base del quale si decide se effettuare o meno il salto



# Soluzione al conflitto di controllo - 2

Anticipazione del **calcolo di destinazione** del salto:

- PC e campo immediato sono disponibili nel registro IF/ID della pipeline una volta letta l'istruzione di salto

→ sufficiente spostare il sommatore che calcola l'indirizzo del salto dallo stadio EX allo stadio ID

Anticipazione della **decisione sul salto**:

- necessario confrontare il contenuto dei due registri letti nello stadio ID (**confrontatore**)

•XOR bit a bit dei due valori: se tutti 0 i due valore sono uguali, quindi basta mettere un NOR sulle uscite e se uguale a 1 la condizione **beq** è verificata

Si noti che è necessario propagare l'esito del confronto allo stadio EX (scrittura dell'esito in ID/EX) per poter generare il valore corretto di PCsrc tramite l'uso del segnale di controllo Branch memorizzato anch'esso in ID/EX. La **logica di controllo del salto** viene spostata di conseguenza dallo stadio MEM allo stadio EX

- Possibili problemi:

- Necessità di propagazione
- Conflitto sui dati



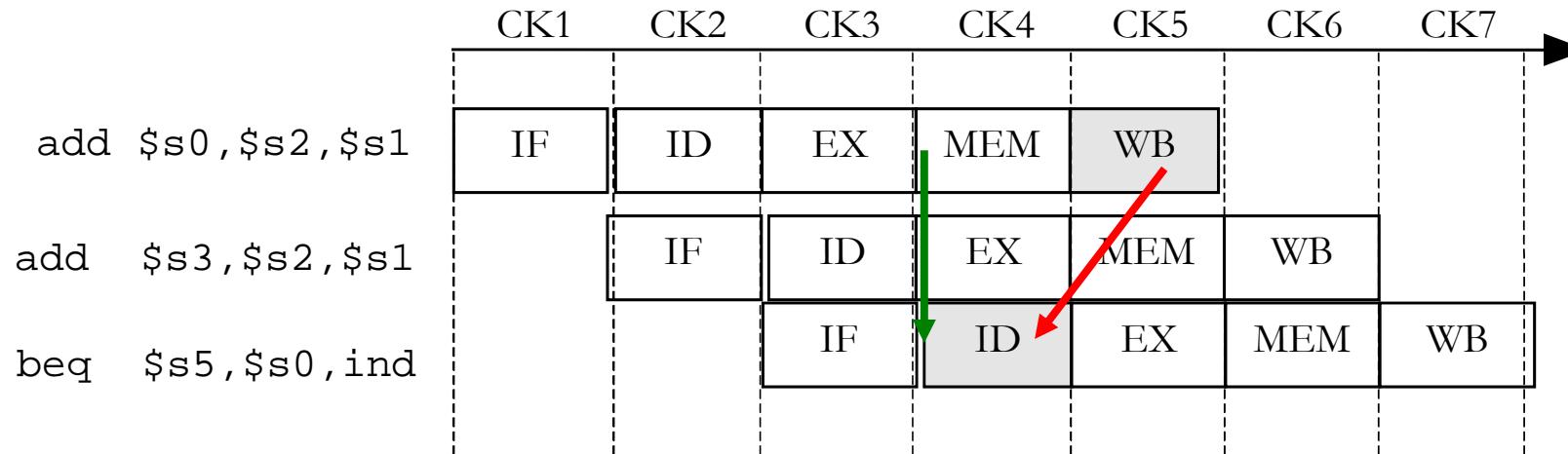
# Anticipazione del confronto dei dati per il salto condizionato

- Per limitare il numero di stalli introdotti nello stadio ID dell'istruzione di salto è necessario prevedere ***percorsi di propagazione*** dei dati nel caso in cui gli operandi su cui eseguire il confronto si possano prelevare dai registri EX/MEM o MEM/WB (propagazione ***EX/ID*** e propagazione ***MEM/ID***). I percorsi e relativi multiplexer non sono mostrati nelle figure
- Se uno degli operandi con cui fare il confronto è calcolato dall'istruzione ***immediatamente precedente*** c'è conflitto sui dati ***non risolvibile con percorsi di propagazione*** e quindi è necessario introdurre ***stalli*** per l'istruzione di salto:
  - 1 ciclo di stallo se l'operando e' calcolato nello stadio di EX
  - 2 cicli di stallo se l'operando e' il risultato di una load e quindi disponibile al termine dello stadio MEM



# Propagazione per anticipo del confronto: un esempio

```
add    $s0, $s2, $s1  
add    $s3, $s2, $s1  
beq    $s5, $s0, ind
```



→ **conflitto**: senza forwarding 1 stallo per beq (se confrontatore veloce)

→ **propagazione EX/ID** nessuno stallo

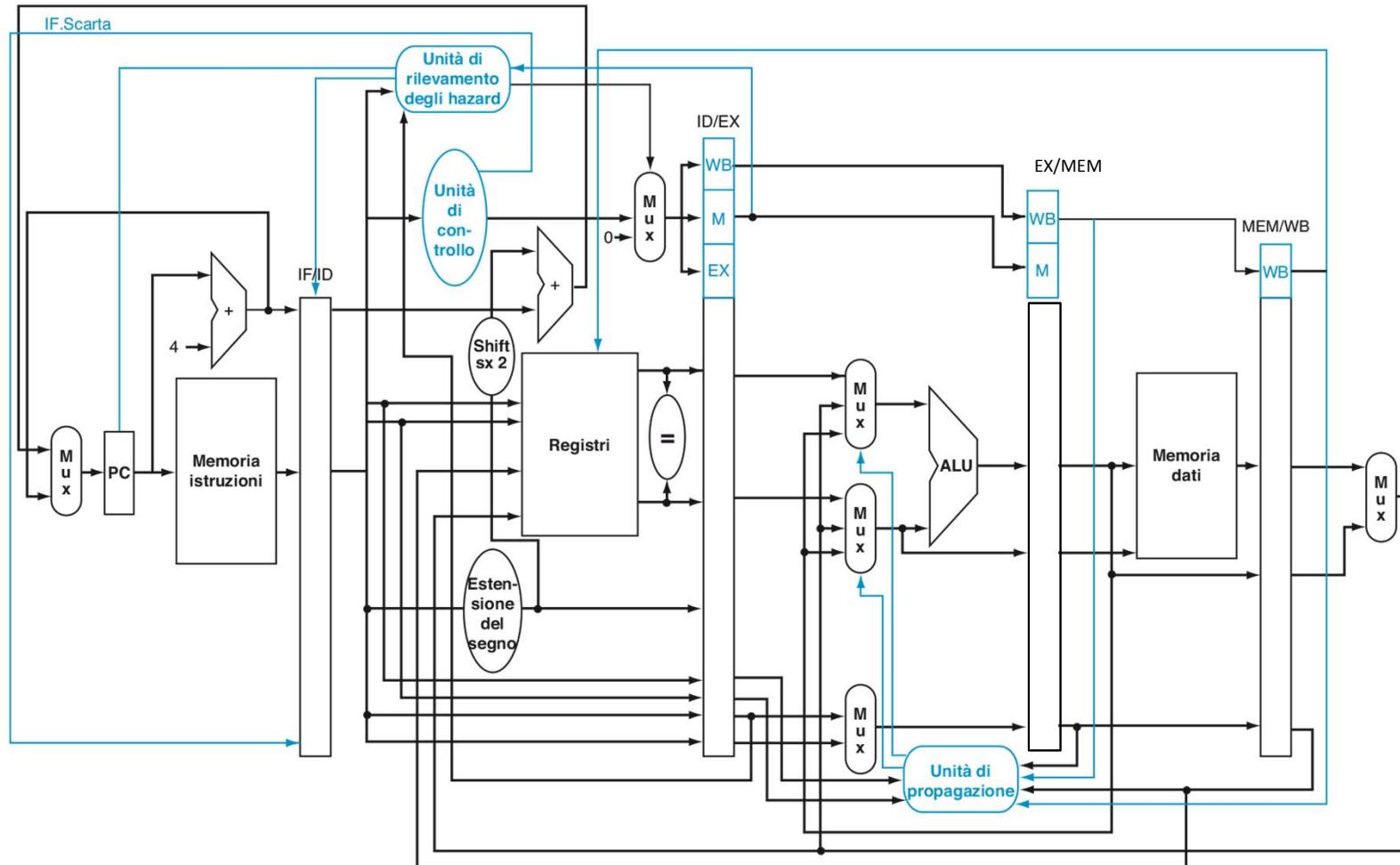


## Spostamento esecuzione salto condizionato allo stadio ID

- Riduce la penalità di una istruzione di salto condizionato (**sulle successive**) a un solo ciclo di clock
  - Necessario scartare l'istruzione nella fase di fetch se il salto deve essere eseguito
  - Per eliminare l'istruzione in esecuzione nello stadio IF viene aggiunto un segnale di controllo **IF.Scarta** che azzerà la parte del registro di pipeline IF/ID che contiene l'istruzione → istruzione diventa una **nop**



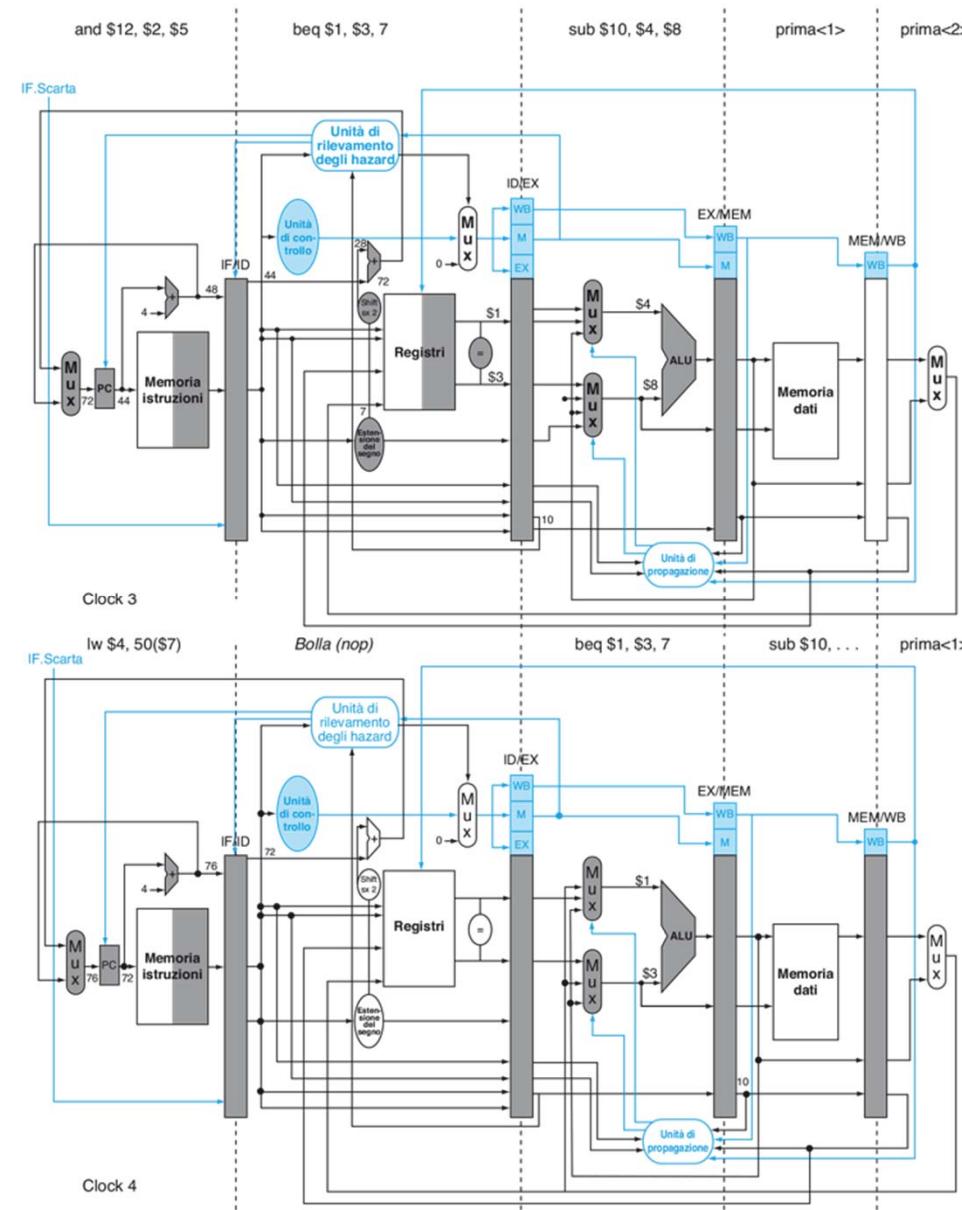
# Unità di elaborazione e di controllo della pipeline completa



## Esempio

```
36 sub    $10, $4, $8
37 beq    $1, $3, 7  # salto relativo a PC:
                  # 40+4+7*4 = 72
44 and    $12, $2, $5
48 or     $13, $2, $6
52 add    $14, $4, $2
56 slt    $15, $6, $7
...
72 lw     $4, 50($7)
```







---

# AXO - Architettura dei Calcolatori e Sistemi Operativi

## memoria cache



# Il problema della memoria

- Le prestazioni di un calcolatore possono essere migliorate anche attraverso il sistema di memoria, utilizzando il concetto di **gerarchia di memoria**
  - diversi livelli di memoria, con tecnologie diverse in modo da ottenere un buon compromesso costo/prestazioni. Lo abbiamo già visto in parte per la **memoria virtuale**
- **Architettura della memoria centrale**
  - . il tasso di crescita nella velocità dei processori non è stato seguito da quello delle memorie
  - **Architettura costituita dall'insieme di memoria centrale e memoria cache:** la memoria cache è molto veloce ma di dimensioni ridotte
  - Obiettivo:
    - . fornire agli utenti una memoria grande e veloce
    - . fornire al processore i dati alla velocità con cui è in grado di elaborarli



# Livelli della gerarchia di memoria

*Capacità  
Tempo di accesso*

**Registri CPU**

100s Bytes  
<10s ns

**Cache**

K Bytes  
10-100 ns

**Memoria centrale**

M Bytes  
200ns- 500ns

**Disco**

G Bytes, 10 ms  
(10,000,000 ns)

**Nastro**  
infinito  
sec-min  
 $10^{-8}$

*Predisposizione  
Unità di trasf.*

prog./compilatore  
1-8 bytes

Controllore cache  
8-128 bytes

OS  
512-4K bytes

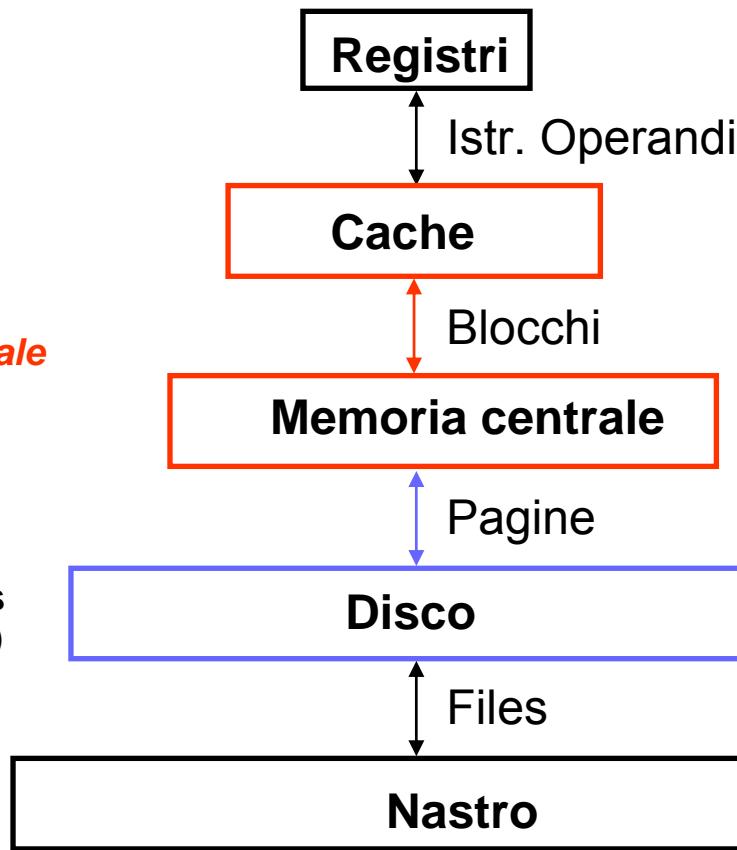
utente  
Mbytes

**Livello superiore**

+ veloce

+ grande

**Livello inferiore**





# Organizzazione di un sistema con memoria cache

- ❑ La memoria centrale e la memoria cache sono organizzate in **blocchi** di parole o di byte, di ugual dimensione
- ❑ La memoria cache contiene **copie di blocchi** della memoria centrale, oppure **blocchi liberi**
  - ad ogni blocco di cache è associato un bit - **valid** - che indica se il blocco è significativo o libero
- ❑ Il **sistema di gestione della cache** è in grado di
  - copiare (caricare) blocchi dalla memoria centrale alla memoria cache oppure di
  - ricopiare (scaricare) blocchi dalla memoria cache alla memoria centrale tramite un'apposita unità funzionale
- ❑ Il **processore accede** sempre e comunque prima alla memoria cache



# Funzionamento base

## Istruzioni

- ❑ Il processore deve leggere un'istruzione
- ❑ Se il blocco che contiene l'istruzione da prelevare si trova in cache, l'istruzione viene letta e eseguita
- ❑ Se l'istruzione non si trova in cache:
  - il processore sospende l'esecuzione
  - il blocco contenente l'istruzione viene caricato dalla memoria centrale in un blocco libero della memoria cache
  - il processore preleva l'istruzione dalla cache e prosegue l'esecuzione

## Dati

- ❑ Il processore deve leggere un dato dalla memoria cache: si procede come visto per la lettura di istruzioni
- ❑ Il processore deve scrivere un dato in memoria: si procede in modo simile ma esiste il problema della coerenza tra memoria cache e memoria centrale (scrittura differita o non differita)



## Il principio di località dei programmi

- ❑ L'uso della memoria cache per incrementare le prestazioni del sistema di memoria sfrutta il **principio di località dei programmi**
  - *località spaziale*: vengono trasferiti in cache più parole di quante non ne siano strettamente richieste (blocco o linea di cache)
  - *località temporale*: viene sfruttata nella scelta del blocco da sostituire nella gestione di un fallimento (es: sostituire il blocco a cui si è fatto accesso meno di recente)



# Memoria cache: definizioni

- **Hit (successo):** accesso (lett/scritt) a dati presenti in un blocco di cache
  - Hit Rate (tasso di successo): numero di accessi a memoria che trovano il dato in cache rispetto al numero totale di accessi
  - Hit Time (tempo di successo): tempo per accedere al dato in cache (= *tempo di accesso a cache + tempo per determinare successo / fallimento della richiesta*)
- **Miss (fallimento):** accesso fallito in cache, i dati devono essere recuperati nella memoria centrale
  - Miss Rate (tasso di fallimento) = 1 - (Hit Rate)
  - Miss Penalty (tempo di fallimento): tempo necessario a sostituire un blocco in cache + tempo di accesso al blocco



# Progetto di una memoria cache

## □ Metodo di **indirizzamento**

- come scegliere il blocco della cache in cui copiare un blocco di memoria centrale (*mapping*)

## □ Metodo di **identificazione**

- come localizzare un blocco di memoria centrale all'interno della cache

## □ Metodo di **scrittura** (coerenza)

- come comportarsi quando si deve scrivere una parola contenuta in un blocco della memoria cache
- 

## □ Metodo di **sostituzione**

- come si sostituiscono blocchi della cache per liberare spazio



# Organizzazione delle memorie cache

- Cache a indirizzamento diretto
- Cache completamente associative
- Cache set- associative



## Cache a indirizzamento diretto: mapping

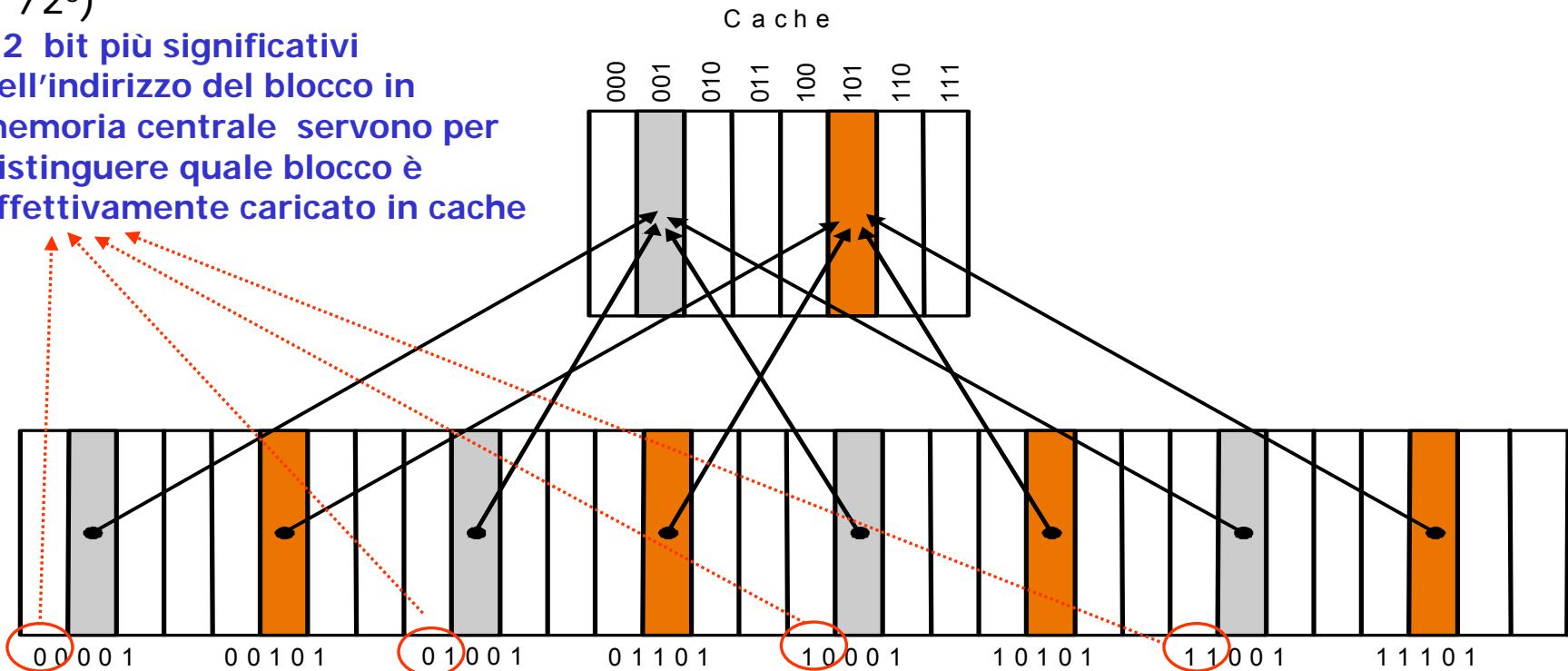
- Ogni blocco della memoria centrale è caricabile in **un solo** blocco della cache
- Più blocchi di memoria centrale possono essere caricati nello **stesso** blocco di memoria cache (*conflitti*)
- **Mapping:** il blocco di indice (indirizzo) ***j*** della memoria centrale è caricabile solo nel blocco di indice (indirizzo)  
*resto div intera di  $j/n$ ° blocchi della cache*



# Cache a indirizzamento diretto

- $2^3$  blocchi cache: 3 bit indirizzo (indice) del blocco
- $2^5$  blocchi memoria centrale: 5 bit indirizzo (indice) del blocco
- $2^2$  blocchi di memoria centrale si mappano in un identico blocco di cache ( $= 2^5 / 2^3$ )

I 2 bit più significativi  
dell'indirizzo del blocco in  
memoria centrale servono per  
distinguere quale blocco è  
effettivamente caricato in cache

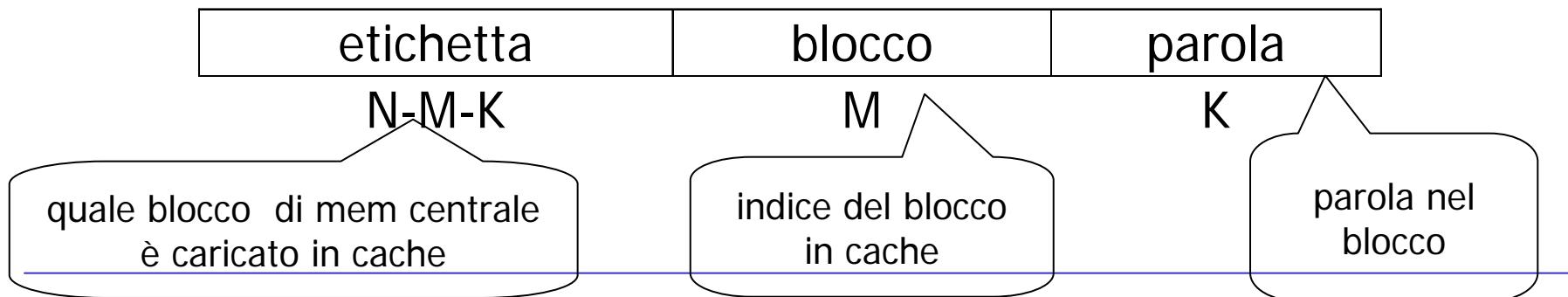




# Struttura degli indirizzi e etichette

- memoria centrale di  $2^N$  parole: N bit di indirizzo
- blocchi di  $2^K$  parole/byte: K bit di indirizzo per identificare una parola/byte nel blocco
- memoria cache di  $2^M$  blocchi: M bit di indirizzo per identificare un blocco nella cache
- memoria centrale di  $2^{(N - K)}$  blocchi ( $2^N / 2^K$ )
- $2^{L=N - M - K}$  blocchi di memoria centrale si mappano in un identico blocco di cache: si usano i bit corrispondenti dell'indirizzo per identificare il blocco effettivamente caricato in cache (**etichetta**)

N bit di indirizzo in memoria centrale





## Esempio

- Memoria centrale da 4Gbyte, cache a indirizzamento diretto da 1KByte, blocchi da 32Byte
- Accesso a memoria a byte:
  - blocchi da 32 byte: 5 bit di indirizzo per identificare il singolo byte
  - cache di 32 blocchi (1Kbyte/32byte): 5 bit di indirizzo per identificare il blocco
  - 22 bit di etichetta = 32 (4Gbyte) - 10



# Cache a indirizzamento diretto: identificazione

- ❑ Occorre memorizzare l'**etichetta** dell'indirizzo di memoria centrale
- ❑ Ogni posizione della cache include:
  - **Valid bit** che indica se questa posizione contiene o meno dati validi. All'accensione, tutte le posizioni della cache sono NON valide
  - **Campo etichetta** che contiene il valore che identifica univocamente l'indirizzo di memoria corrispondente ai dati memorizzati
  - **Campo dati** che contiene una copia dei dati
- ❑ Dato un **indirizzo di memoria centrale**, l'accesso a cache avviene nel modo seguente:
  - la parte di indirizzo che contiene l'indice del blocco in cache viene usata per selezionare il blocco
  - se il blocco è valido, la parte di indirizzo che contiene l'etichetta, viene confrontata con l'etichetta del blocco selezionato
  - se il confronto è positivo il dato è in cache e la parte di indirizzo che specifica il byte (o la parola) nel blocco viene utilizzata per accedere al byte (o parola) corretta

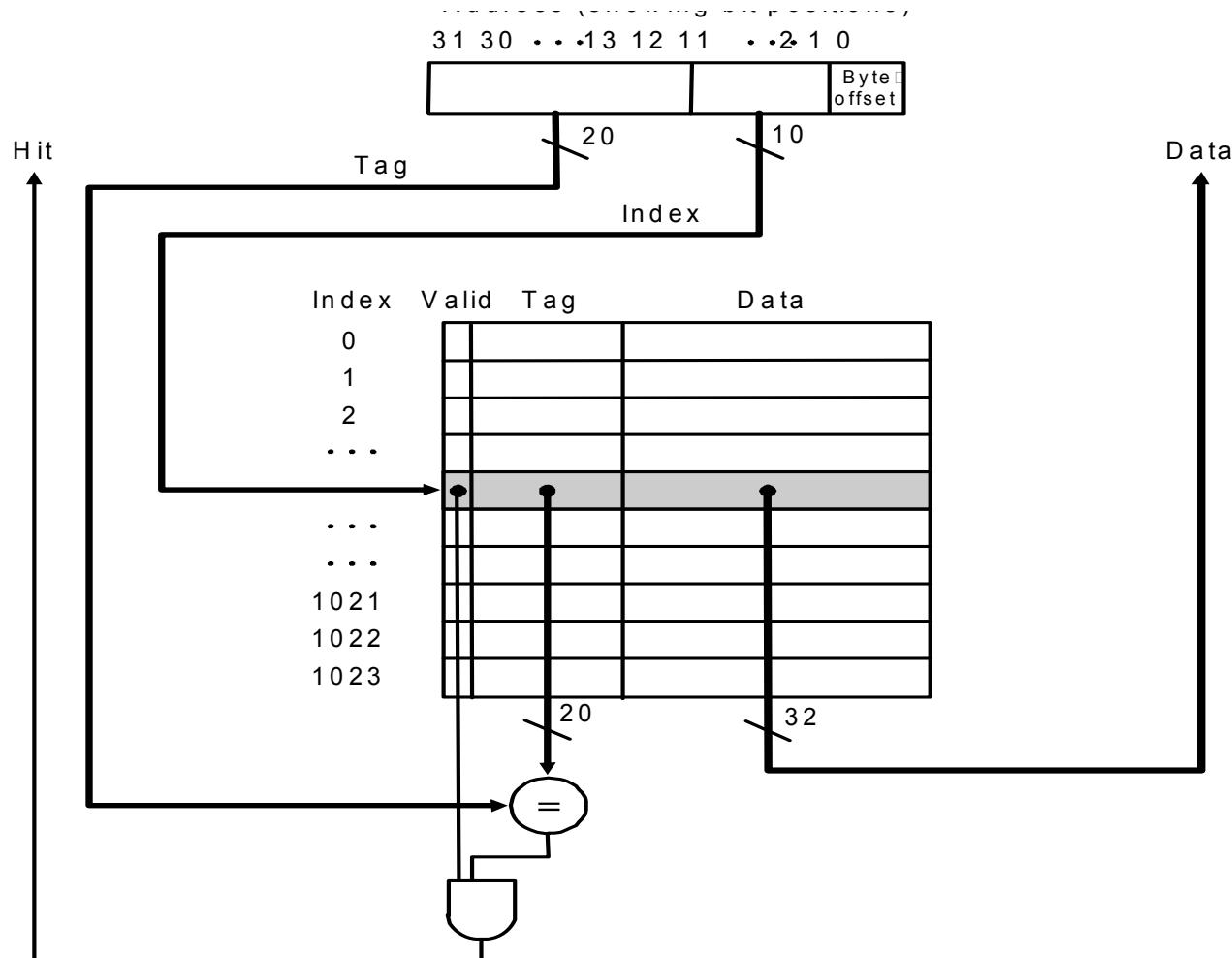


## Un altro esempio

- Indirizzi di memoria a 32 bit
- Cache a indirizzamento diretto:
  - blocco costituito da una parola di 4 byte
  - 1024 blocchi ( $2^{10}$ )
- Struttura dell'indirizzo di memoria:
- Bit 0 e 1 per individuare il singolo byte nella parola
- Bit 2-11 per individuare il blocco di cache
- Bit 12-31 come etichetta



# Cache a indirizzamento diretto





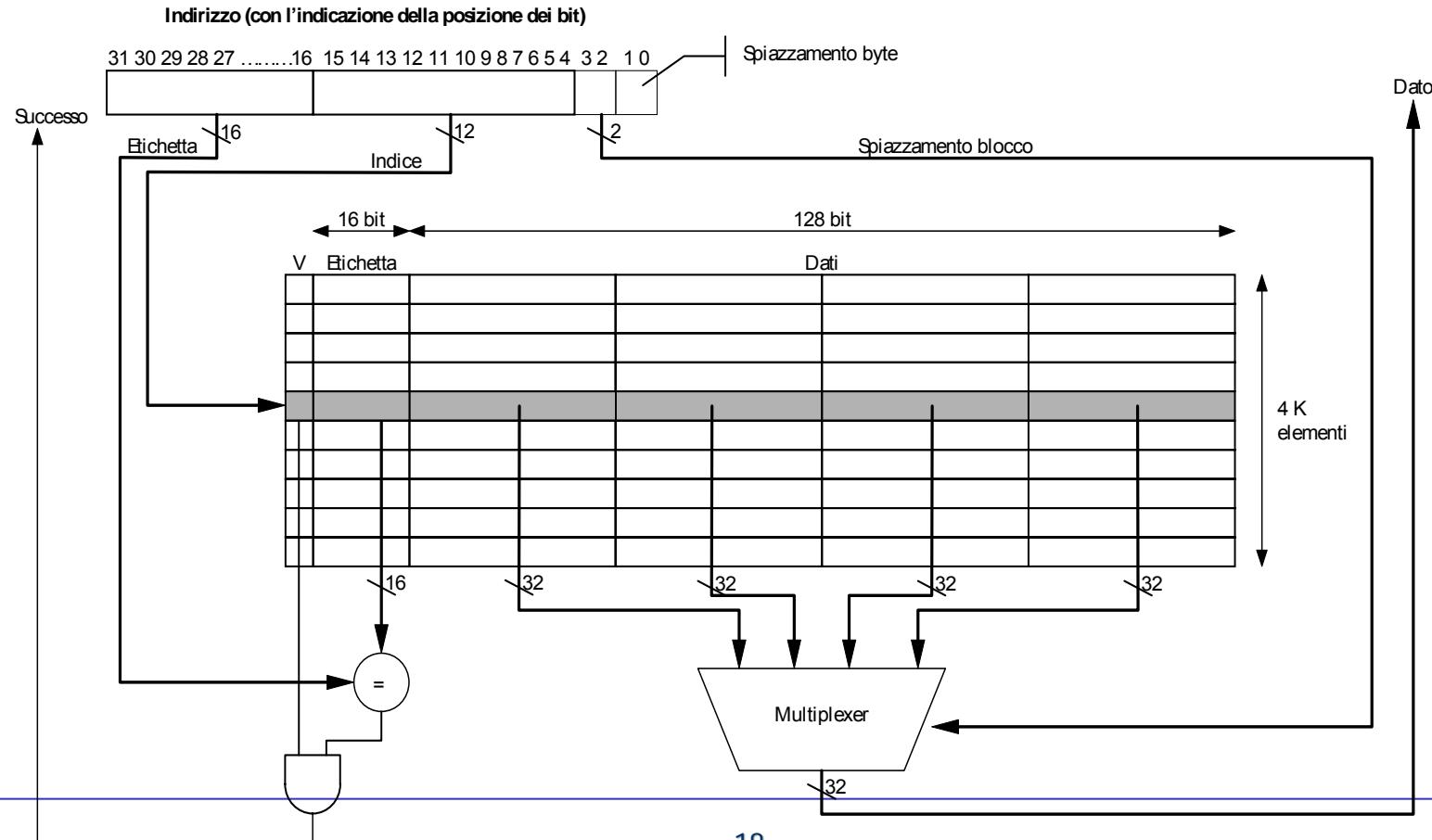
# Indirizzamento diretto: caso generale

- Indirizzo di memoria di N bit diviso in 4 campi
  - **B** bit meno significativi permettono di individuare il singolo byte della parola nel blocco di cache
    - Se la parola non è indirizzabile per byte B= 0
  - **K** bit per identificare la parola all'interno del blocco di cache
    - Se il blocco contiene una sola parola K=0
  - **M** bit per individuare la posizione del blocco in cache
  - **N-M-K-B** bit di etichetta per verificare che il blocco di cache contenga esattamente l'indirizzo cercato



# Cache a indirizzamento diretto

- Memoria centrale 4Gigabyte (32 bit indirizzo) - accesso a memoria a parole da 32 bit
- Blocchi da 4 parole (16 byte): 2 bit per parola nel blocco, 2 bit per byte nella parola
- Memoria cache da 16kparole: 12 bit per indirizzare un blocco ( $n^o$  blocchi =  $2^{14}/2^2$ )
- Etichetta da 16 bit





# Hit e Miss in lettura e scrittura

- Successo nella lettura
  - il dato è presente in cache
  
- Fallimento nella lettura
  - *stall* della CPU, richiesta del blocco contenente il dato cercato alla memoria, copia in cache, ripetizione dell'operazione di lettura in cache
  
- Successo nella scrittura:
  - Sostituzione del dato sia in cache sia in memoria (**write-through**)
  - Scrittura del dato solo nella cache (**write-back**): la copia in memoria avviene in un secondo momento
  
- Fallimento nella scrittura:
  - *stall* della CPU, richiesta del blocco contenente il dato cercato alla memoria, copia in cache, ripetizione dell'operazione di scrittura



## Indirizzamento diretto e sostituzione del blocco

- Come si sostituiscono blocchi della cache per liberare spazio:

è un **problema automaticamente risolto dalla modalità di mapping**,  
non è possibile alcuna scelta



# Miglioramento delle prestazioni

- Aumentando le dimensioni del blocco di cache, il numero di miss si riduce
- Migliorare sia larghezza di banda (velocità di esecuzione) sia latenza (tempo necessario per svolgere l'operazione): uso di cache multiple
- Introdurre una cache separata per istruzioni e dati (**split cache**)
  - Le operazioni di lettura/scrittura possono essere svolte in modo indipendente in ogni cache  $\Rightarrow$  raddoppia larghezza di banda della memoria
  - Processore necessita di due porte di collegamento alla memoria



# Cache associative: indirizzamento associativo

- ❑ Un blocco può essere memorizzato in un **qualsiasi** blocco della cache (pochi conflitti di allocazione)
- ❑ Non esiste una relazione tra indirizzo di memoria del blocco e posizione in cache. Non esiste quindi problema di mapping
- ❑ Struttura dell'indirizzo di memoria di N bit, con blocchi di cache di  $2^M$  byte:
  - M bit meno significativi dell'indirizzo individuano il byte nel blocco della cache
  - N-M bit più significativi: **etichetta** (lunga)



# Cache associative e identificazione

- ❑ Ricerca di un dato nella cache richiede il **confronto di tutte le etichette presenti in cache** con l'etichetta dell'indirizzo di memoria richiesto
- ❑ Per aumentare le prestazioni la **ricerca** avviene **in parallelo** tramite una **memoria associativa** (costo elevato)
- ❑ In caso di fallimento della ricerca è necessario copiare il dato dalla memoria centrale
- ❑ Se la cache è piena è necessario sostituire un blocco. Sostituzione del blocco
  - Scelta casuale
  - Scelta del dato meno utilizzato di recente (LRU)



## Cache set-associative $n$ vie

- I blocchi della cache sono divisi in **gruppi** (insiemi o set)
- La cache **si indirizza per gruppi**
- Ogni gruppo contiene almeno **2 ( $n$ )** blocchi di cache
- Ogni blocco della memoria centrale può essere **caricato in un solo gruppo** (prefissato), in uno **qualsiasi degli  $n$  blocchi**
- Una cache set-associativa in cui un blocco può andare in  $n$  posizioni viene definita set-associativa a  **$n$  vie**.



# Indirizzamento nelle cache set associative

- ❑ Ogni blocco della memoria centrale corrisponde ad un unico *gruppo* della cache ed il blocco può essere messo in uno *qualsiasi* degli elementi di questo gruppo
  - Combina la modalità a indirizzamento diretto per gli insiemi della cache, e la modalità completamente associativa per i blocchi all'interno dell'insieme
- ❑ Un indirizzo di memoria di N bit è suddiviso in 4 campi:
  - B bit meno significativi per individuare il byte all'interno della parola
  - K bit per individuare la parola all'interno del blocco
  - M bit per individuare il gruppo, indice del gruppo (*indirizzamento diretto per il gruppo*)
  - N-(M+K+B) come etichetta, quale blocco nel gruppo (*completamente associativa nel gruppo*)

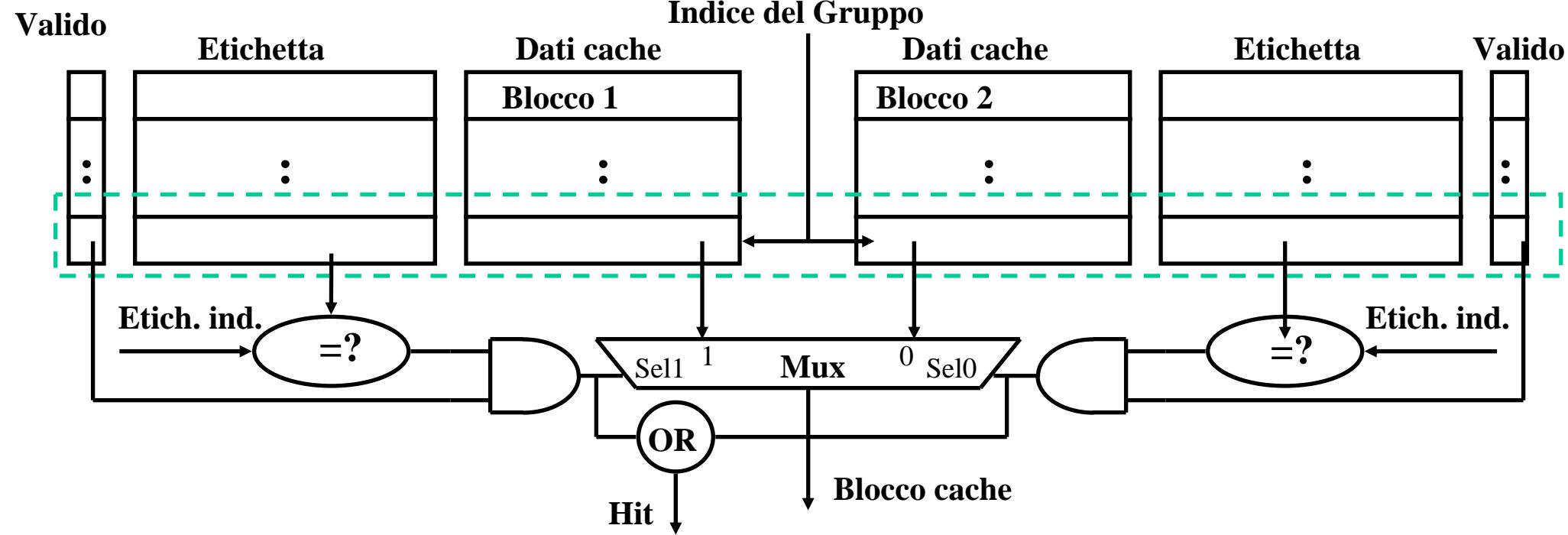


# Cache set associativa a 2 vie

- ❑ Cache a due vie: ogni gruppo è costituito da 2 blocchi
- ❑ La parte di indirizzo che individua il gruppo seleziona i due possibili blocchi della cache
- ❑ L'etichetta dell'indirizzo cercato viene confrontata in parallelo con le due etichette dei due blocchi del gruppo
- ❑ Il blocco viene selezionato in base al risultato dei due confronti



# Cache Set Associativa a due vie



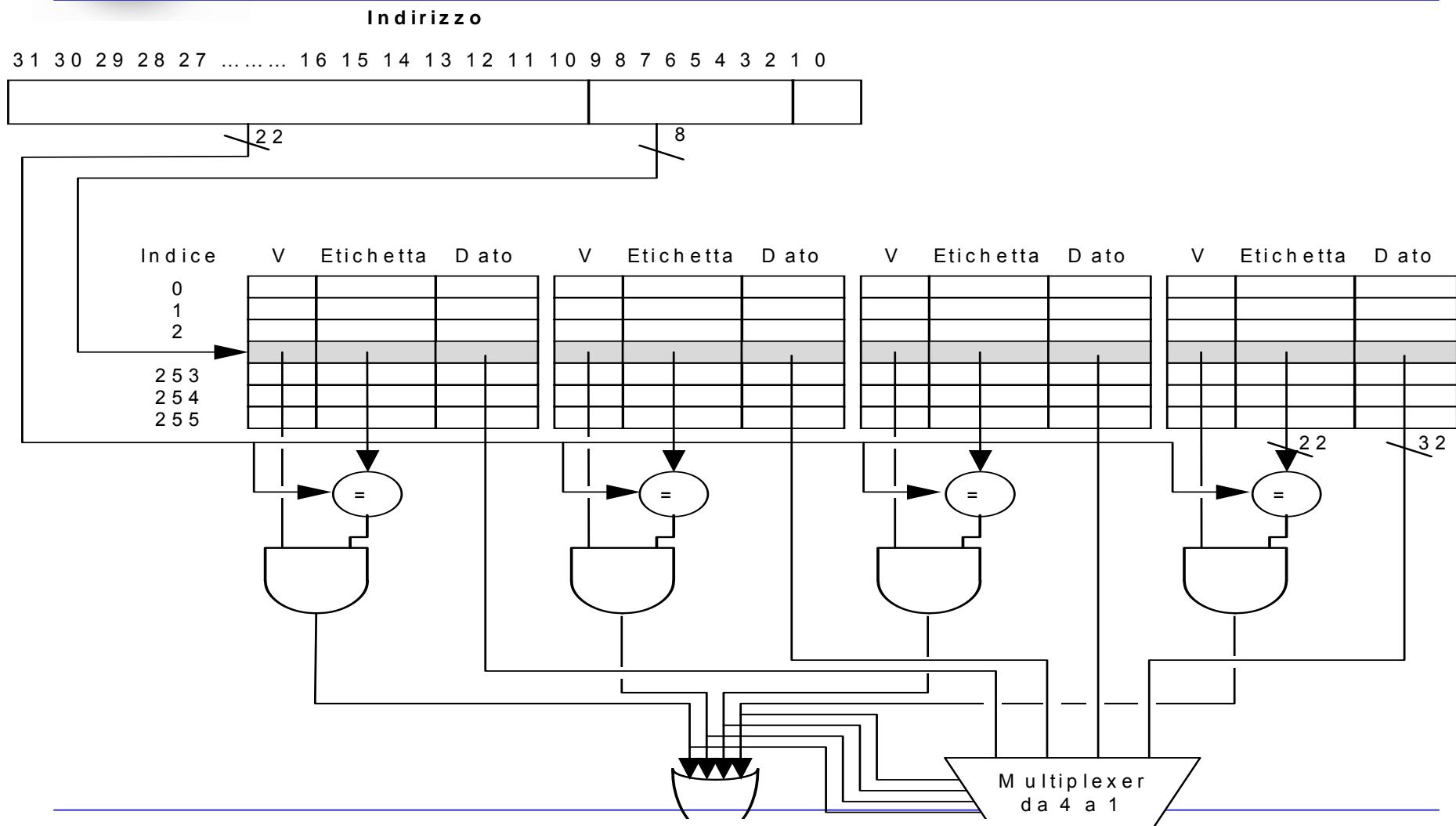


# Cache set associativa a 4 vie

- ❑ A 4 vie: ogni blocco di memoria centrale può essere caricato in 4 blocchi di cache, quindi la dimensione di ogni gruppo è di 4 blocchi
- ❑ Memoria centrale 4Gbyte: 32 bit
- ❑ Memoria cache 1K parole (1 parola=4byte), blocchi da 1 parola
- ❑ Organizzazione dell'indirizzo:
  - Bit 0 e 1 per indirizzare i byte
  - Numero blocchi nella cache = dimensioni della cache/dimensioni del blocco=  $2^{10}/1 = 2^{10}$
  - Numero di gruppi nella cache= numero di blocchi/ dimensioni del gruppo  $= 2^{10}/2^2 = 2^8$
  - Bit 2-10 indice del gruppo nella cache
  - Bit 31-11 etichetta



# Cache set associativa a 4 vie





# Confronto tra diverse organizzazioni di cache

- ❑ Cache set-associativa a N vie v. Cache a indirizzamento diretto:
  - N comparatori vs. 1
  - Un ritardo dovuto al MUX aggiuntivo per i dati
  - Dati sono disponibili solo **dopo** il segnale di Hit/Miss
- ❑ In una cache a indirizzamento diretto, il blocco di cache richiesto è disponibile **prima** del segnale di Hit/Miss:
  - Possibile ipotizzare un successo e quindi proseguire. Si recupera successivamente se si trattava in realtà di un fallimento.



## Conclusioni - 4 domande sulla gerarchia di memoria

---

- Q1: Dove portare un blocco nel livello di memoria superiore (cache)?  
*(Mapping o Posizionamento del blocco)*
- Q2: Come si identifica un blocco se si trova nel livello superiore (cache)?  
*(Identificazione del blocco)*
- Q3: Quale blocco deve essere sostituito nel caso di un fallimento?  
*(Sostituzione del blocco)*
- Q4: Cosa succede durante una scrittura?  
*(Strategia di scrittura)*



# Posizionamento (mapping) del blocco

- ❑ Indirizzamento diretto:
  - Posizione univoca: indirizzo di memoria modulo numero dei blocchi in cache
- ❑ Completamente associativa:
  - Posizione qualunque all'interno della cache
- ❑ Set associativa
  - Posizione univoca del gruppo
  - Posizione qualsiasi all'interno del gruppo prefissato



# Identificazione del blocco

- **Indirizzamento diretto:**
  - Identifica blocco
  - Verifica etichetta e verifica bit valido
- **Completamente associativo:**
  - Confronta etichetta in ogni blocco e verifica bit valido
- **Set-associativo**
  - Identifica gruppo
  - Confronta etichette del gruppo e verifica bit valido



# Sostituzione del blocco

- ❑ Cache a indirizzamento diretto: definito dall'indirizzo
- ❑ Cache set associative (o completamente associative):
  - Casuale
  - LRU (Least Recently Used)
    - Cache a 2 vie. Esiste un bit di uso per ogni blocco dell'insieme: quando un blocco viene utilizzato il suo bit viene messo a 0, l'altro a 1
    - Cache a 4 vie. Esiste un contatore di uso per ogni blocco dell'insieme: quando un blocco viene utilizzato il suo contatore viene messo a 0, gli altri incrementati di 1. Per limitare i ritardi nell'accesso viene usato un bit di uso per ogni blocco



# Strategie di scrittura di un blocco

- ❑ ***Write through***—L'informazione viene scritta sia nel blocco del livello superiore (cache) sia nel blocco di livello inferiore della memoria (memoria centrale)
  - Fallimenti in letture non si tramutano in scritture in memoria (penalità di fallimento più lunga)
- ❑ ***Write back***—L'informazione viene scritta solo nel blocco di livello superiore (cache). Il livello inferiore viene aggiornato solo quando avviene la sostituzione del blocco.
  - Per ogni blocco di cache è necessario mantenere l'informazione sulla scrittura
  - Ad ogni blocco è associato un bit MODIFICA che indica se il blocco in cache è stato modificato o meno e va quindi copiato in memoria in caso di sostituzione
  - Non si hanno aggiornamenti ripetuti delle stesse celle di memoria: l'aggiornamento avviene una volta sola



# PRESTAZIONI del processore MIPS pipeline senza e con memoria cache

# Prestazioni di processore GENERICO (P&H – pp 29-31)

Definizioni dei parametri di prestazione fondamentali del processore:

sia  $P$  una prova, ossia un programma da eseguire che in generale contiene cicli

$T(P)$  = tempo di CPU consumato per eseguire la prova  $P$

$cicli(P)$  = numero di cicli di clock di CPU consumati per eseguire la prova  $P$

$CK$  = periodo di clock del processore

$FREQ$  = frequenza di clock del processore =  $1 / CK$

La relazione fondamentale tra i parametri del processore è la seguente:

$$T(P) = cicli(P) \times CK = cicli(P) / FREQ$$

Esempio: dato il tempo misurato  $T(P)$  e nota la frequenza di clock  $FREQ$ , determina il numero di cicli di clock  $cicli(P)$  necessari all'esecuzione di  $P$

$$T(P) = 10 \text{ s} \quad FREQ = 2 \text{ GHz} = 2 \times 10^9 \text{ Hz}$$

$$\Rightarrow cicli(P) = 10 \times 2 \times 10^9 = 20 \times 10^9 = 20 \text{ miliardi di cicli}$$

# Parametri relativi alle istruzioni (P&H – pp 29-31)

Definizioni dei parametri relativi alle istruzioni macchina:

$NI(P)$  = numero di istruzioni eseguite durante la prova  $P$   
nota bene: contano anche le ripetizioni delle istruzioni nei cicli !

$CPI(P)$  = cicli di clock per istruzione  
cioè il valore medio su tutte le istruzioni eseguite durante la prova  $P$   
 $= cicli(P) / NI(P)$

o equivalentemente  $cicli(P) = NI(P) \times CPI(P)$

Tempo di esecuzione in funzione delle istruzioni macchina:

$$T(P) = NI(P) \times CPI(P) / FREQ$$

## Prestazioni di processore PIPELINE (P&H – pp 239)

In una pipeline senza stalli di alcun tipo, si ha sempre  $CPI (P) = 1$ .

Se la pipeline presenta stalli introdotti per risolvere conflitti, si ha:

$stalli (P)$  = numero di cicli di stallo nella prova  $P$

$cicli (P)$  =  $NI (P) + stalli (P) + 4 \approx NI (P) + stalli (P)$

dato che per grandi numeri il termine costante 4 è trascurabile.

Pertanto si ricava il parametro  $CPI$  in funzione di istruzioni e stalli:

$$CPI (P) = cicli (P) / NI (P) \approx ( NI (P) + stalli (P) ) / NI (P)$$

E come prima si ricava, con l'espressione di  $CPI$  stalli compresi:

$$T (P) = ( NI (P) \times CPI (P) ) / FREQ$$

## PIPELINE con mem cache (P&H – cap 5.4 fino a pp 358)

Se la memoria cache è ideale, ossia non presenta eventi di MISS, allora essa non influisce sulle prestazioni del processore (di qualunque tipo).

Se la memoria cache è reale, ossia ha MISS, allora la pipeline va stallata per un numero di cicli di clock pari al tempo per gestire il MISS.

Il numero di stalli di pipeline causati dalla memoria cache è definito così:

$stalliM(P)$  = numero di stalli causati dalla memoria cache durante la prova  $P$

Pertanto, con cache il tempo di esecuzione della prova  $P$  si esprime così:

$$T(P) = ( cicli(P) + stalliM(P) ) / FREQ$$

Nota:  $cicli(P)$  potrebbe già comprendere gli stalli di pipeline introdotti per risolvere i vari tipi di conflitto (dati e controllo), come visto prima.

## PIPELINE con mem cache (P&H – cap 5.4 fino a pp 358)

Definizioni dei parametri relativi alla memoria cache:

$h$  = *HIT rate* frequenza di eventi di HIT

se necessario, si divide il parametro  $h$  in  $hI$  e  $hD$  per distinguere tra Istruzioni e Dati

*HIT time* tempo di accesso a una parola in *memoria cache*

$m$  = *MISS rate* =  $1 - h$  frequenza di eventi di MISS

idem se necessario, si divide il parametro  $m$  in  $mI$  e  $mD$

$M$  = *MISS penalty* tempo di caricamento del blocco da *m. centrale* a *m. cache*

Generalmente la penalità di MISS  $M$  è espressa in cicli di clock  
(altrimenti la si esprime in  $s$  o in  $ns$ ).

# PIPELINE con mem cache (P&H – cap 5.4 fino a pp 358)

Gli stalli di memoria sono quantificabili nel modo seguente:

$stalliM(P)$  = numero di stalli causati dalla memoria cache  
= stalli in lettura + stalli in scrittura

Si suppone che le penalità di MISS in lettura e scrittura siano identiche (ossia si suppone si applicare la politica di *write through*), pertanto si ha:

$stalliM(P)$  = numero di accessi alla memoria ( $P$ )  $\times m \times M$

E la penalità di MISS  $M$  è calcolabile così (di solito in cicli di clock):

$M$  = dimensione del blocco (in parole)  $\times$  tempo di accesso a una parola in *memoria centrale*

# Esempio di PIPELINE con cache (P&H – cap 5.4)

Parametri di cache e processore noti:

$$mI = 2\% = 0,02 \quad mD = 4\% = 0,04 \quad M = 100 \text{ cicli}$$

$$CPI \text{ esclusi stalli memoria} = 2 \text{ cicli} \quad \text{percentuale di istruzioni } l/w \text{ e } sw = 36\% = 0,36$$

Determiniamo il *CPI* con cache reale (ossia con eventi di MISS):

$$\text{cicli per miss in cache Istruzioni} = NI \times 0,02 \times 100 = 2 \times NI$$

$$\text{cicli per miss in cache Dati} = NI \times 0,36 \times 0,04 \times 100 = 1,44 \times NI$$

$$\text{cicli totali} = 3,44 \times NI$$

$$CPI \text{ totale} = 2 + 3,44 = 5,44 \text{ cicli}$$

Confronto tra *CPI* con cache reale e ideale (ossia senza MISS):

$$CPI \text{ cache reale} / CPI \text{ cache ideale} = 5,44 / 2 = 2,72$$

$$\text{tempo speso in stalli di memoria} = 3,44 / 5,44 = 0,63 = 63\%$$

Se si accelera la pipeline riducendone il CPI (via forwarding elimina i conflitti di dato):

$$CPI \text{ esclusi stalli memoria scende a 1} = 4,44 / 1 = 4,44$$

$$\text{tempo speso in stalli di memoria} = 3,44 / 4,44 = 0,77 = 77\%$$

# Tempo medio di accesso alla memoria – AMAT

Definizione di *Average Memory Access Time*:

$$AMAT = HIT \text{ time} + MISS \text{ rate} \times MISS \text{ penalty}$$

Esempio:

$$CK = 1 \text{ ns}$$

$$HIT \text{ time} = \text{tempo di accesso a una parola in cache} = 1 \text{ ciclo}$$

$$m = 5 \% = 0,05$$

$$M = 20 \text{ cicli}$$

$$\Rightarrow AMAT = 1 \text{ ciclo} + 0,05 \times 20 \text{ cicli} = 2 \text{ cicli (ossia } 2 \text{ ns)}$$

Micro nota: alcuni autori definiscono  $AMAT = HIT \text{ rate} \times HIT \text{ time} + MISS \text{ rate} \times MISS \text{ penalty}$ ; questa formula (viene dalla proprietà additiva delle probabilità di eventi mutuamente esclusivi) implica la precedente se si definisce la *MISS penalty* in modo da comprendere lo *HIT time* dell'evento di HIT dovuto alla ripetizione dell'accesso alla parola immediatamente dopo che il blocco è stato caricato in cache, cioè se  $MISS \text{ rate} = \text{dimensione del blocco} \times \text{tempo di accesso a una parola in mem. centrale} + HIT \text{ time}$ ; dato che di solito lo *HIT time* è piuttosto piccolo, comprenderlo o no nella *MISS penalty* cambia pochissimo il valore di questa, pertanto le due formule di AMAT danno risultati quasi identici; qui si considera solo la formula data in alto nella slide, più breve da calcolare.



---

# AXO - Architettura dei Calcolatori e Sistemi Operativi

**come tradurre da C a MIPS  
un modello per generare codice macchina**

(tratto da Patterson & Hennessy – 4a ed. italiana)



---

# **catena di traduzione (breve riassunto)**



# struttura generale

- il processo di traduzione da linguaggio sorgente di alto livello a linguaggio macchina è diviso in quattro fasi
  - analisi sintattica e trattamento errori
  - generazione del codice macchina
  - ottimizzazione del codice macchina
  - assemblaggio e collegamento
  - ling. sorgente **alto livello**
  - ling. macchina **simbolico**
  - facoltativa
  - ling. macchina **numerico**
- di solito si vede il risultato: codice macchina eseguibile
- volendo si possono avere le rappresentazioni intermedie
  - albero sintattico del programma
  - codice macchina non ottimizzato
  - eventuali altre forme intermedie ...
- spesso si può scegliere il modello di processore per cui tradurre e così ottenere codice macchina “ad hoc”



# analisi sintattica e generazione di codice

## □ analisi sintattica

- verifica se il programma è sintatticamente corretto
- segnala e corregge errori

## □ generazione di codice

- il programma viene tradotto in linguaggio macchina
- il codice macchina prodotto è in forma simbolica
- non numerica ossia non eseguibile direttamente

ling. sorgente di alto livello

```
int a  
  
int b  
  
a = b + 1
```



```
A: .word // int a  
  
B: .word // int b  
  
lw      $t0, B  
addi   $t0, $t0, 1  
sw      $t0, A
```

ling. macchina simbolico



# assemblaggio e collegamento

- **assemblaggio**
  - risolve simboli assemblatore
    - indirizzo di memoria
    - etichetta d'istruzione
    - spiazzamento
  - l'istruzione simbolica viene tradotta in codifica numerica
- **collegamento**
  - unisce più moduli eseguibili
  - per esempio programma e librerie standard (IO ecc)

```
A: .word // int a  
B: .word // int b  
  
lw      $t0, B  
  
addi   $t0, $t0, 1  
  
sw      $t0, A
```

ling. macchina simbolico

codici operativi	
lw	1010...
addi	1100...
sw	1010...

tab. dei simboli

A	1010...
B	1101...

1010.....  
1100.....  
1010.....

ling. macchina numerico



---

# **struttura del programma di alto livello (segmentare il programma)**



# considerazioni generali

---

- schema di compilazione, ispirato a GCC, per tradurre da linguaggio sorgente C a linguaggio macchina MIPS
  - presuppone di conoscere MIPS: banco di registri, classi d'istruzioni, modi d'indirizzamento e organizzazione del sottoprogramma (chiamata rientro e passaggio parametri)
  - consiste in vari insiemi di convenzioni e regole per
    - segmentare il programma
    - dichiarare le variabili
    - usare (leggere e scrivere) le variabili
    - rendere le strutture di controllo
  - non attua necessariamente la traduzione più efficiente
  - sono possibili varie ottimizzazioni “ad hoc” del codice
-



# SO e modello di memoria del processo

- per il SO il processo tipico ha tre segmenti essenziali
  - codice main e funzioni utente
  - dati variabili globali e dinamiche
  - pila aree di attivazione con indirizzi, parametri, registri salvati, e variabili locali
- codice e dati sono segmenti dichiarati nel programma
- il segmento di pila viene creato al lancio del processo
- processi grandi o sofisticati hanno facoltativamente
  - due o più segmenti codice o dati
  - segmenti di dati condivisi
  - segmenti di libreria dinamica
  - e altre peculiarità ...
- questo modello di memoria è valido in generale



# dichiarare i segmenti tipici

gli indirizzi di impianto dei segmenti sono virtuali (non fisici)

```
// var. glob.           // segmento dati
...
// funzioni           .data
...
main (...) {          // indir. iniziale dati 0x 1000 0000
    // corpo           ...
    ...               // var globali e dinamiche
}
                    // segmento codice
                    .text
                    // indir. iniziale codice 0x 0040 0000
                    .globl main
main: ...           // codice programma
```

non occorre dichiarare esplicitamente il segmento di pila  
implicitamente esso inizia all'indirizzo 0x 7FFF FFFF  
e cresce verso gli indirizzi minori (ossia verso 0)



---

# **convenzioni per assegnare memoria e registri (allocare la memoria e come usare i registri)**



# considerazioni generali

---

- per tradurre da linguaggio sorgente, p. es. C, a linguaggio macchina, p. es. MIPS, bisogna definire un modello di architettura “run-time” per memoria e processore
  - le convenzioni del modello run-time comprendono
    - collocazione e ingombro delle diverse classi di variabile
    - destinazione di uso dei registri
  - il modello di architettura run-time consente interoperabilità tra porzioni di codice di provenienza differente, come per esempio codice utente e librerie standard precompilate
  - esempio tipico in linguaggio C è la libreria standard di IO
-



# convenzioni per variabili

- in generale le variabili del programma sono collocate così
  - globali in memoria a indirizzo fissato (assoluto)
  - locali nei registri del processore o nell'area di attivazione in pila (vedere le precisazioni che seguono)
  - dinamiche in memoria (qui non sono considerate)
- le istruzioni aritmetico-logiche operano su registri
- dunque per operare sulla variabile (glob – loc – din)
  1. prima caricala in un registro libero (*load*)
  2. poi elaborala nel registro (confronto e aritmetica-logica)
  3. infine riscrivila in memoria (*store*)
- variabile locale allocata in registro ⇒ salta (1) e (3)



# ingombro in memoria delle variabili

- l'unità di memoria minima indirizzabile è il singolo byte
- in C sotto Linux i diversi tipi di variabile ingombrano così

sizeof ( <b>char</b> )	= 1 byte	- byte (b)
sizeof ( <b>short int</b> )	= 2 byte	- mezza parola (h)
sizeof ( <b>int</b> )	= 4 byte	- parola (w)
sizeof ( <b>long int</b> )	= 8 byte	- parola lunga o doppia (d)
sizeof ( <b>array</b> )	= <b>somma</b> ingombri elementi	
sizeof ( <b>struct</b> )	= <b>somma</b> ingombri campi	

- in C sotto Windows l'intero ordinario ingombra due byte (e il resto scala di conseguenza)
- per il tipo reale (float) vedi lo standard IEEE 754 (qui non si considera il tipo reale)



# banco dei registri di MIPS

0	0	costante 0: inizializzazione di registri e variabili, e uso interno per macro e pseudosistruzioni
1	<i>at</i>	riservato all'assembler-linker: uso interno per macro e pseudoistruzioni
2-3	<i>v0 - v1</i>	valore (di tipo scalare intero o puntatore) in uscita da funzione ( <i>v1</i> si usa solo se occorre restituire una parola doppia)
4-7	<i>a0 - a3</i>	primi quattro argomenti (di tipo intero scalare o puntatore) in ingresso a funzione
8-15	<i>t0 - t7</i>	registri per valori temporanei: valori intermedi nel calcolo di un'espressione oppure locazioni temporanee per elaborare una variabile (globale o locale) allocata in memoria
16-23	<i>s0 - s7</i>	registri per variabili locali: usati per allocare variabili locali (di tipo scalare o puntatore), sotto opportune condizioni, invece di allocarle nell'area di attivazione della funzione
24-25	<i>t8 - t9</i>	registri per valori temporanei (in aggiunta a <i>t0 - t7</i> ) (generalmente non usati)
26-27	<i>k0 - k1</i>	registri riservati per il nucleo (kernel) del SO
28	<i>gp</i>	global pointer: puntatore al segmento dati statici (variabili globali)
29	<i>sp</i>	stack pointer: puntatore alla cima della pila (ultima cella occupata della pila)
30	<i>fp</i>	frame pointer: puntatore all'area di attivazione (uso facoltativo – vedere più avanti)
31	<i>ra</i>	return address: puntatore all'indirizzo di rientro a chiamante



## altri registri di MIPS (con funzioni speciali)

---

- PC – program counter (32 bit)
  - HI e LO – risultato moltiplicazione e divisione (32 bit)
    - usati implicitamente da istruzioni di moltiplicazione e divisione
  - nel coprocessore 0 (interruzioni trattate nella 2° parte)
    - EPC – indirizzo dell'istruzione interessata dall'interruzione
    - STATUS – registro di stato (vari bit con significati differenziati)
    - CAUSE – registro di causa (di interruzione)
  - nel coprocessore 1
    - registri di virgola mobile \$f0, ... (qui non usati)
-



# assemblatore e simulatore - MARS

The screenshot shows the MARS software interface with several windows open:

- Text Segment:** Displays assembly code for a program named "main". The code includes instructions like addi, lui, sw, and addi. The assembly source is:

```
addi $t0, $0, 1
lui $t1, 0x00001001
sw $t0, A
addi $t0, $t0, 1
sw $t0, B
sw $t0, 0x00000004($t1)
```
- Labels:** Shows the global label "main" at address 0x00400000 and the local labels A and B at addresses 0x10010000 and 0x10010004 respectively.
- Data Segment:** Displays memory starting at address 0x10010000 with values filled from 0x00000001 to 0x00000000 across multiple memory locations.
- Registers:** A table showing the state of the MIPS registers (\$zero through \$t0-\$t7, \$s0-\$s5, \$t0-\$t7, \$s0-\$s5, \$gp, \$sp, \$fp, \$ra, pc, hi, lo) with their corresponding numbers and values.
- Mars Messages:** A window showing the message: "-- program is finished running (dropped off bottom) --".



---

# **come dichiarare le diverse classi di variabile (globale – locale – parametro)**



# considerazioni generali

---

- in C la variabile è un oggetto formale e ha
    - nome per identificarla e farne uso
    - tipo per stabilirne gli usi ammissibili
  - in MIPS la variabile è un elemento di memoria (byte parola o regione di mem) e ha una “collocazione” con
    - nome per identificarla
    - modo per indirizzarla
  - in MIPS la variabile viene manipolata tramite indirizzo simbolico o nome di registro
  - occorre però distinguere tra diverse classi di variabile (var globale – parametro – var locale – eventualmente var dinamica)
-



# allineamento

- di default le variabili allocate in memoria sono allineate secondo il loro tipo, come segue

- byte (8 bit) a indirizzo qualunque: 0, 1, 2, 3, 4, ...
  - mezza parola (16 bit) a indirizzo pari: 0, 2, 4, 6, ...
  - parola (32 bit) a indirizzo multiplo di quattro: 0, 4, 8, 12, ...
  - parola lunga o doppia (64 bit) a indirizzo multiplo di otto: 0, 8, 16, 24, ...

- si può cambiare l'allineamento con una direttiva

```
.align 0      // nessun allineamento (come per byte)
.align 1      // allinea a indirizzo pari
.align 2      // allinea a indirizzo multiplo di 4
.align 3      // allinea a indirizzo multiplo di 8
```

- esempio

```
.align 1 // allinea a indirizzo pari
VAR: .word      // VAR (32 bit) a ind. pari (non mul. 4)
```



# variabile globale nominale

---

- la variabile globale è collocata in memoria a indirizzo fisso stabilito dall'assemblatore
- per comodità l'indirizzo simbolico della variabile coincide con il nome della variabile
  - solitamente l'indirizzo simbolico è scritto in carattere maiuscolo, ma solo per comodità di visualizzazione
- gli ordini di dichiarazione e disposizione in memoria delle variabili globali coincidono



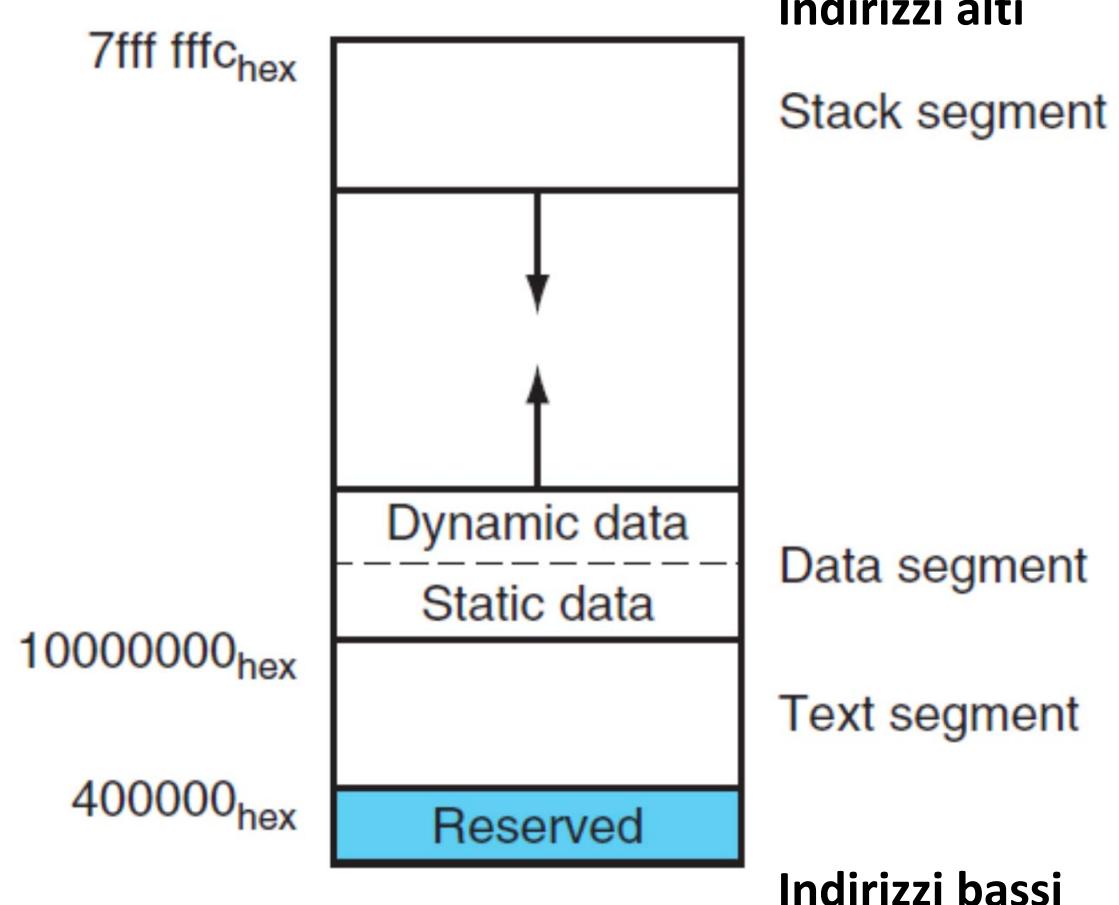
# segmento dati statici per variabili globali

le variabili globali sono allocate nel segmento dei dati statici a partire dall'indirizzo: **0x 1000 0000**

l'allocazione delle variabili globali procede in ordine di dichiarazione, riservando loro spazio dagli indirizzi bassi verso quelli alti

per puntare al segmento dati statici si usa il registro *global pointer: gp*

il registro *gp* è inizializzato all'indirizzo: **0x 1000 8000**





# variabile globale – scalare e vettore

tipo intero ordinario e corto a 32 e 16 bit rispettivamente

```
char c = `@`  
int a = 1  
short int b  
int vet [10]  
int * punt1  
char * punt2
```

```
.data // segmento dati  
C: .byte 64 // @ valore iniziale  
A: .word 1 // 1 valore iniziale  
B: .half // non inizializzato  
VET: .space 40 // 10 elem × 4 byte  
PUNT1: .word 0 // iniz. a NULL  
PUNT2: .word // non inizializzato
```

il valore iniziale è facoltativo

il puntatore è sempre una parola a 32 bit,  
indipendentemente dal tipo di oggetto puntato

NOTA: alcuni simulatori del processore MIPS, p. es. MARS, richiedono che le direttive **.word**, **.half** e **.byte** specifichino sempre un valore iniziale per la variabile, altrimenti non riservano spazio in memoria; in tale caso basta dare il valore iniziale 0.



# variabile globale – struct (per completezza)

tipo intero ordinario a 32 bit

```
struct {  
    int a  
    int b  
    char c  
} s  
  
c 8 bit (1 byte) 8  
b 32 bit (4 byte) 4  
s a 32 bit (4 byte) 0  
  
ind alti  
ind bassi  
s.b = 2
```

.data // segmento dati  
**s:** .space n // riserva n byte  
il compilatore riserva uno spazio di *n* byte (dagli indirizzi bassi verso quelli alti) sufficiente a contenere in sequenza i campi della *struct*  
ci sono varie convenzioni su come calcolare il numero *n* di byte, se i campi sono allineati o no, o se ogni campo occupa esattamente lo spazio sufficiente a contenerlo, o più spazio in parte libera («padding»)  
per queste convenzioni occorre vedere la ABI (Application Binary Interface) specifica usata dal compilatore

si accede a un campo interno della *struct* mediante lo spiazzamento del campo rispetto all'indirizzo iniziale della *struct* (vedi anche array)

```
li $t0, 2 // carica cost 2 in reg t0  
la $t1, s // carica ind di s in reg t1  
sw $t0, 4($t1) // aggiorna campo b di s
```



# parametro in ingresso alla funzione

- i primi quattro parametri vanno passati nei registri  $a0$  (primo nella testata),  $a1$ ,  $a2$  e  $a3$  (quarto)
  - se sono di tipo scalare o puntatore (a 32 bit)
  - il nome di vettore è considerato puntatore (al primo elem.)
  - per passare una *struct* si veda la ABI del compilatore\*
- gli eventuali parametri rimanenti vanno impilati a cura del chiamante, sempre come valori a 32 bit
  - è raro che una funzione abbia più di quattro parametri
- prima di chiamare la funzione, il chiamante salva sulla pila i registri  $t0 - t7$ ,  $a0 - a3$ ,  $v0 - v1$  (in questo ordine) che il chiamante vuole potere riavere inalterati al rientro

\* la ABI di gcc impone di passare l'indirizzo dell'inizio della *struct*



# valore in uscita dalla funzione

- il valore in uscita (a 32 bit) va restituito nel registro *v0*
  - se è di tipo scalare o puntatore
  - il nome di vettore è considerato puntatore (al primo elem.)
  - per restituire una *struct* si veda la ABI del compilatore\*
- se il valore in uscita è di tipo *double*, si usa anche *v1*

\* la ABI di *gcc* impone di passare l'indirizzo dell'inizio della *struct*



# variabile locale nominale

- la variabile locale può essere gestita in vari modi, secondo il tipo di variabile e il grado di ottimizzazione del codice, e anche in dipendenza di come la variabile viene utilizzata
- variabile di tipo scalare o puntatore – due modi
  - in un registro del blocco s0 - s7, se per altro motivo non deve avere un indirizzo di memoria – vedi precisazioni sotto
  - altrimenti nell'area di attivazione della funzione
- variabile di tipo scalare, ma che viene anche acceduta tramite un puntatore – un solo modo
  - nell'area di attivazione della funzione, perché deve avere un indirizzo !
- variabile di tipo *array* (o *struct*) – un solo modo
  - sempre nell'area di attivazione della funzione

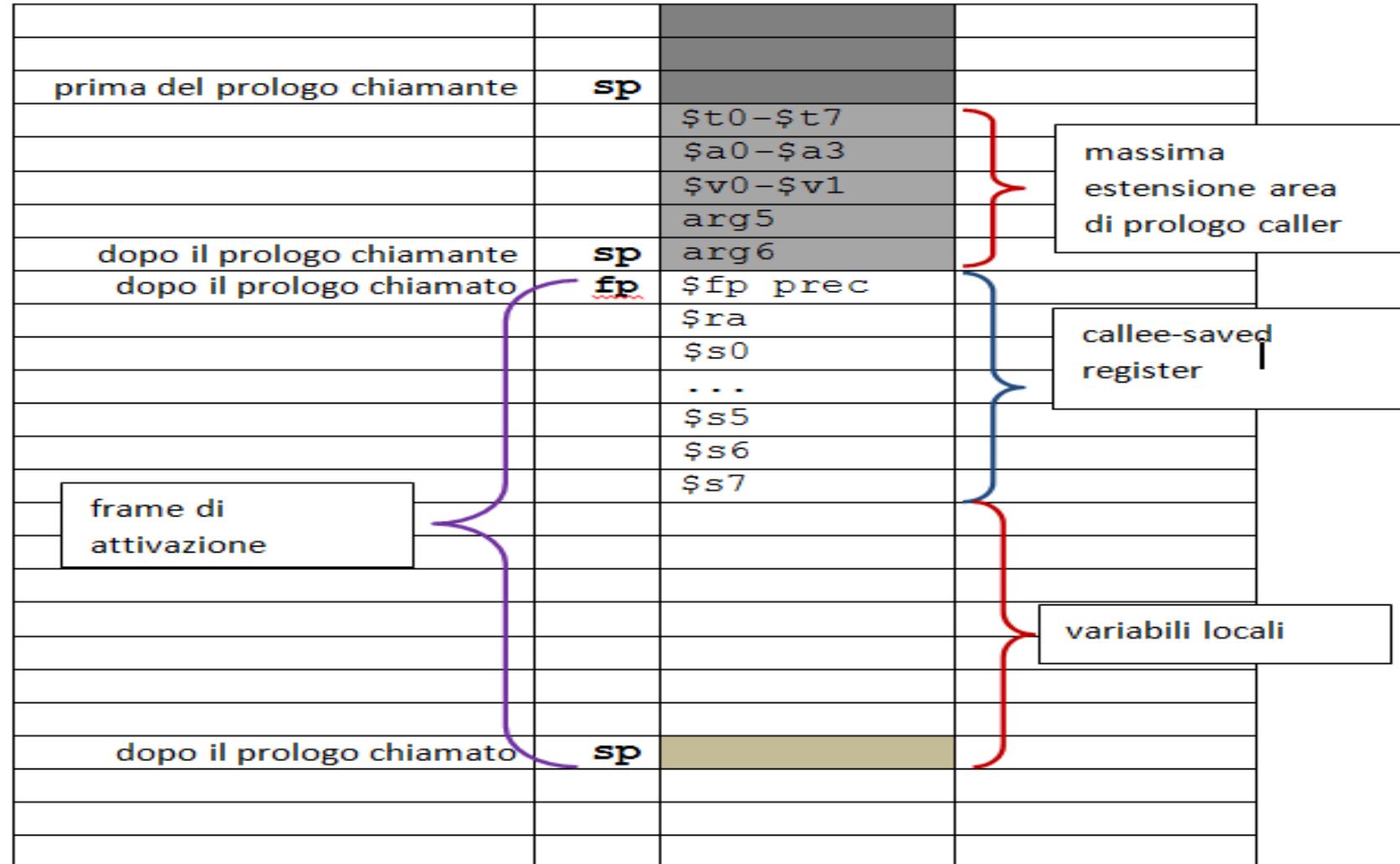


# area di attivazione - forma generale





# area di attivazione - in dettaglio





## precisazioni - salvare i registri

---

- i registri  $t0 - t7$ ,  $a0 - a0$  e  $v0 - v1$  vanno salvati (e poi ripristinati), da parte del **chiamante**, se **esso** deve o preferisce poterli riavere inalterati dopo la chiamata
  - salva solo quelli deve / preferisce riavere inalterati
  - se non occorre salvarne nessuno, essi non hanno spazio riservato in pila, e non vanno né salvati né ripristinati
- i registri  $s0 - s7$  vanno salvati (e poi ripristinati), da parte del **chiamato**, se **esso** li usa per allocare variabili locali
  - salva solo quelli che vengono assegnati a variabili locali
  - se non occorre salvarne nessuno, essi non hanno spazio riservato nell'area di attivazione, e non vanno né salvati né ripristinati



# precisazioni - salvare i registri

---

- il registro *ra* (return address) è facoltativo
- va usato se la funzione chiama un'altra funzione, o anche se chiama se stessa (funzione *ricorsiva*)
  - viene salvato da parte del **chiamato**
- non va usato se la funzione non chiama nessun'altra funzione e neppure se stessa (funzione *foglia*)
  - in questo caso il registro *ra* non ha spazio riservato nell'area di attivazione, e non va né salvato né ripristinato



## precisazioni - salvare i registri

- il registro *fp* (frame pointer) è facoltativo
- il registro *fp* va usato se serve un riferimento stabile all'area di attivazione della funzione in esecuzione
  - viene salvato da parte del **chiamato**
- se il registro *fp* non è usato, non ha spazio riservato nell'area di attivazione, e non va né salvato né ripristinato
  - in questo caso la funzione usa il registro *sp* per fare riferimento al contenuto dell'area di attivazione
  - di fatto il registro *fp* è usato raramente, ma ha una funzione specifica e conviene ricordarne il modo di uso
- gli spiazzamenti rispetto a *fp* degli elementi nell'area di attivazione differiscono dagli spiazzamenti rispetto a *sp* !
  - gli esempi illustrano la differenza



---

# come usare le diverse classi di variabile scalare (globale – locale – parametro)



## usare la variabile – regola base

- supponi che la variabile (loc o glob) sia allocata in memoria
- gran parte delle istruzioni lavora solo nei registri  
(in particolare tutte le istruzioni aritmetico-logiche)

### **REGOLA BASE PER TRADURRE**

- se hai uno statement C che usa e magari modifica la variabile  
(per esempio lo statement di l'assegnamento **a = a + 1** ), comportati così con la variabile (nell'esempio **a**)
  - caricala in un registro **del blocco t0 - t7** all'inizio dello statement
  - e, se è stata modificata, memorizzala alla fine dello statement
- se la variabile figura nello statement C successivo, non tentare di “tenerla” nel registro – memorizzala e ricaricala !



# usare la variabile – ottimizzazione

- supponi che la variabile sia locale, di tipo scalare o puntatore, e allocata in un registro del blocco di registri s0 - s7
- gran parte delle istruzioni lavora solo nei registri (in particolare tutte le istruzioni aritmetico-logiche)

## **REGOLA OTTIMIZZATA PER TRADURRE**

- se hai uno statement C che usa e magari modifica la variabile (per esempio l'assegnamento  $a = a + 1$ ), usala e modificala tenendola nel registro del blocco s0 - s7 dove è allocata
- ricorda però che il chiamato deve salvare il registro in pila (all'inizio) e ripristinare il registro da pila (alla fine)



# variabile globale – esempio

## int a 32 bit

<b>int a</b>	<b>A:</b> .word                        // spazio per int
...	li            \$t0, <b>1</b> // carica cost. in t0
<b>a = 1</b>	sw            \$t0, <b>A</b> // notazione usata qui
...	

i compilatori (p. es. gcc) espandono «**sw** \$t0, A» così:

sw            \$t0, A(\$gp)

per accedere alla variabile globale A, il collegatore (linker) calcolerà (con regole ad esso note) lo spiazzamento relativo al valore del registro global pointer gp

*qui noi scriviamo semplicemente «sw \$t0, A»*

*funziona in modo analogo con lw*



# variabile locale, parametro e valore di uscita – esempio

## varloc allocata in pila (senza fp)

```
int f (int n) {  
    int a  
    ...  
    a = a + n  
    // chiama funz.  
    ...  
    return a  
} /* f */
```

area di attivazione (8 byte)  
ra salvato 4(\$sp)  
sp-> varloc a 0(\$sp)

F: addiu \$sp, \$sp, -8 // crea area  
sw \$ra, 4(\$sp) // salva ra  
...  
lw \$t0, 0(\$sp) // carica a  
add \$t0, \$t0, \$a0 // calcola a = a + n  
sw \$t0, 0(\$sp) // memorizza a  
// chiama un'altra funzione  
...  
lw \$v0, 0(\$sp) // valore uscita  
lw \$ra, 4(\$sp) // ripristina ra  
addiu \$sp, \$sp, 8 // elimina area  
jr \$ra // rientra



# variabile locale, parametro e valore di uscita – esempio

varloc allocata in registro s0 (senza fp)

```
int f (int n) {  
    int a  
    ...  
    a = a + n  
    // chiama funz.  
    ...  
    return a  
} /* f */
```

area di attivazione (8 byte)  
ra salvato 4(\$sp)  
sp-> s0 salvato 0(\$sp)

<b>F:</b>	<b>addiu</b> \$sp, \$sp, -8 // crea area
	<b>sw</b> \$ra, 4(\$sp) // salva ra
	<b>sw</b> \$s0, 0(\$sp) // salva s0
	...
	<b>add</b> \$s0, \$s0, \$a0 // calcola a = a + n
	// chiama un'altra funzione
	...
	<b>move</b> \$v0, \$s0 // valore uscita
	<b>lw</b> \$s0, 0(\$sp) // ripristina s0
	<b>lw</b> \$ra, 4(\$sp) // ripristina ra
	<b>addiu</b> \$sp, \$sp, 8 // elimina area
	<b>jr</b> \$ra // rientra



# variabile locale, parametro e valore di uscita – esempio con direttiva .eqv per spiazzamenti

varloc allocata in pila (senza fp)

```
int f (int n) {  
    int a  
    ...  
    a = a + n  
    // chiama funz.  
    ...  
    return a  
} /* f */
```

area di attivazione (8 byte)  
ra salvato 4(\$sp)  
sp-> varloc a 0(\$sp)

F: .eqv RA, 4 // spiazzamento di ra in area attiv.  
.eqv A, 0 // spiazzamento di a in area attiv.  
**addiu** \$sp, \$sp, -8 // crea area  
**sw** \$ra, RA(\$sp) // salva ra  
...  
**lw** \$t0, A(\$sp) // carica a  
**add** \$t0, \$t0, \$a0 // calcola a = a + n  
**sw** \$t0, A(\$sp) // memorizza a  
// chiama un'altra funzione  
...  
**lw** \$v0, A(\$sp) // valore uscita  
**lw** \$ra, RA(\$sp) // ripristina ra  
**addiu** \$sp, \$sp, 8 // elimina area  
**jr** \$ra // rientra  
similmente se ci sono più variabili locali



# variabile locale, parametro e valore di uscita – esempio

## varloc allocata in pila e con *fp*

```
int f (int n) {
    int a
    ...
    a = a + n
    // chiama funz.
    ...
    return a
} /* f */
```

area di attivazione (12 byte)

	vs	sp	vs	fp
fp->	fp salvato	8(\$sp)	0(\$fp)	
	ra salvato	4(\$sp)	-4(\$fp)	
sp->	varloc a	0(\$sp)	-8(\$fp)	

```
F: addiu $sp, $sp, -12 // crea area
    sw    $fp, 8($sp)    // salva fp chiamante
    addiu $fp, $sp, 8    // sposta fp (punta a fp salvato)
    // da qui in avanti fp è il reg base per accedere all'area
    sw    $ra, -4($fp)    // salva ra
    ...
    lw    $t0, -8($fp)    // carica a
    add  $t0, $t0, $a0    // calcola a = a + n
    sw    $t0, -8($fp)    // memorizza a
    // chiama un'altra funzione
    ...
    lw    $v0, -8($fp)    // valore uscita
    lw    $ra, -4($fp)    // ripristina ra
    // fine dell'uso di fp come reg base per accedere all'area
    lw    $fp, 8($sp)    // ripristina fp chiamante a
    addiu $sp, $sp, 12    // elimina area
    jr    $ra                // rientra
```

ora il registro *sp* potrebbe anche cambiare durante l'esecuzione  
**attenzione: gli spiazzamenti di *ra* e varloc *a* sono riferiti al registro *fp*!**  
similmente con le combinazioni viste prima e/o con direttiva **.eqv**



# puntatore globale a variabile globale – esempio

## int da 32 bit

int * p	P:	.word	// spazio per p
int a	A:	.word	// spazio per a
...		...	
p = &a	la	\$t0, A	// carica ind. di a
	sw	\$t0, P	// p = &a
...		...	
*p = 1	li	\$t0, 1	// inizializza t0
...	lw	\$t1, P	// carica p
	sw	\$t0, (\$t1)	// *p = 1
in realtà il compilatore (p. es. gcc) espanderebbe così			
lui \$t0, A			
addiu \$t0, \$t0, A (oppure ori \$t0, \$t0, A)			
la pseudo-istruzione «la \$t0, A» con indirizzo A			



# puntatore globale a variabile locale – esempio

varloc allocata in pila (senza *fp*) – funz non ha param

```
int * p
void f ( ) {
    int a
    int b
    p = &a
    ...
    *p = 1
    // chiama funz.
    return
} /* f */
area di attiv. (12 byte)
    ra salvato 8($sp)
    varloc a  4($sp)
    sp-> varloc b  0($sp)
```

```
P: .word          // spazio per p
F: addiu $sp, $sp, -12 // crea area
    sw   $ra, 8($sp) // salva ra
    ...
    move $t0, $sp      // inizializza $t0 | a
    addiu $t0, $t0, 4   // ind. di a
    sw   $t0, P         // p = &a
    ...
    li    $t0, 1         // inizializza $t0
    lw    $t1, P         // inizializza $t1
    sw   $t0, ($t1)     // *p = 1
    ...
    lw    $ra, 8($sp)    // ripristina ra
    addiu $sp, $sp, 12   // elimina area
    jr   $ra             // rientra
```



---

# **come rendere le strutture di controllo**

## **(goto – if – while – do – for – break)**



## considerazioni generali

---

- le strutture di controllo comportano l'uso di salto
- servono etichette univoche per ogni struttura
  
- qui si danno schemi di salto corretti e uniformi
- ma non necessariamente i più efficienti possibili
  
- traduci in cascata le strutture di controllo annidate
- parti dalla struttura di controllo più interna e risali



# goto – salto incondizionato

in C la struttura «goto» modella il salto incondizionato

```
... // parte I  
goto MARCA  
... // parte II  
MARCA: ... // destinazione  
... // parte III
```

in C ben strutturato «goto» è usato raramente, ma esiste

```
... // parte I  
j MARCA // va' a MARCA  
... // parte II  
MARCA: ... // destinazione  
... // parte III
```

naturalmente si può anche saltare indietro  
nota: invece di «**j** MARCA» si potrebbe scrivere «**beq** \$0, \$0, MARCA» la cui condizione è sempre verificata (0 uguale a 0), ma così l'indirizzo è relativo a PC



# if – condizionale a una via

salto condizionato in avanti – **int** da 32 bit (come in Unix)

```
// var. globale  
int a  
...  
// condizione  
if (a == 5) {  
    // ramo then  
    ...  
} /* end if */  
... // seguito
```

```
A:      .word           // riserva mem per a  
          ...  
IF:      lw    $t0, A    // carica a  
        li    $t1, 5     // inizializza t1  
        // se a != 5 va' avanti a ENDIF  
        bne   $t0, t1, ENDIF  
THEN:    ...           // ramo then  
ENDIF:   ...           // seguito
```

va modificato come noto per “>”, “<”, “>=”, “<=” e “!=”  
similmente se **a** è varloc, param od oggetto puntato



# if – condizionale a due vie

## salto condizionato in avanti – **int** da 32 bit

```
// var. globale  
int a  
...  
// condizione  
if (a == 5) {  
    // ramo then  
    ...  
} else {  
    // ramo else  
    ...  
} /* end if */  
... // seguito
```

<b>A:</b>	<b>.word</b>	// riserva mem per a
	...	
<b>IF:</b>	<b>lw</b> \$t0, <b>A</b>	// carica a
	<b>li</b> \$t1, 5	// inizializza t1
	// se a != 5 va' avanti a ELSE	
	<b>bne</b> \$t0, t1, ELSE	
<b>THEN:</b>	...	// ramo then
	<b>j</b> ENDIF	// va' avanti a ENDIF
<b>ELSE:</b>	...	// ramo then
<b>ENDIF:</b>	...	// seguito

va modificato come noto per ">", "<", ">=", "<=" e "!="  
idem se **a** è varloc, param od oggetto puntato



# while – ciclo a condizione iniziale

salto condizionato in avanti – **int** da 32 bit

```
// var. globale  
int a  
...  
// condizione  
while (a == 5) {  
    // corpo  
    ...  
} /* end while */  
// seguito  
...
```

<b>A:</b>	<b>.word</b> // riserva mem per a ... WHILE: <b>lw</b> \$t0, <b>A</b> // carica a <b>li</b> \$t1, <b>5</b> // inizializza t1 // se a != 5 va' avanti a ENDWHILE <b>bne</b> \$t0, t1, ENDWHILE ...        // corpo <b>j</b> WHILE    // va' indietro a WHILE ENDWHILE: ...        // seguito
-----------	---

va modificato come noto per ">", "<", ">=", "<=" e "!="  
idem se **a** è varloc, param od oggetto puntato



# do – ciclo a condizione finale

## salto condizionato indietro – **int** da 32 bit

```
// var. globale  
int a  
...  
do {  
    // corpo  
    ...  
    // condizione  
} while (a == 5)  
    // seguito  
...
```

<b>A:</b>	<b>.word</b> // riserva mem per a ... <b>DO:</b> ... // corpo <b>lw</b> \$t0, <b>A</b> // carica a <b>li</b> \$t1, <b>5</b> // inizializza t1 // se a == 5 va' indietro a DO <b>beq</b> \$t0, t1, <b>DO</b> <b>ENDDO:</b> ... // seguito
-----------	---

va modificato come noto per ">", "<", ">=", "<=" e "!="  
idem se **a** è varloc, param od oggetto puntato



# for – ciclo a conteggio

## salto condizionato in avanti – **int** da 32 bit

```
// variabile globale  
int a  
...  
// testata del ciclo  
for (a = 2; a <= 5; a++) {  
    // corpo del ciclo  
    ...  
} /* end for */  
// seguito del ciclo  
...
```

la variabile di conteggio **a** viene aggiornata in fondo al corpo del ciclo (**«a++»** è post-incremento)

<b>A:</b>	<b>.word</b> // riserva mem per a
<b>a</b> = 2	<b>li</b> \$t0, 2 // inizializza t0
FOR:	<b>sw</b> \$t0, A // memorizza a
<b>a</b> <= 5	<b>lw</b> \$t0, A // carica a
	<b>li</b> \$t1, 5 // inizializza t1
	// se a > 5 va' avanti a ENDFOR
	<b>bgt</b> \$t0, \$t1, ENDFOR // <b>pseudo-istr.</b>
	... // corpo del ciclo
<b>a++</b>	<b>lw</b> \$t0, A // carica a
	<b>addi</b> \$t0, \$t0, 1 // incrementa
	<b>sw</b> \$t0, A // memorizza a
	<b>j</b> FOR // va' indietro a FOR
ENDFOR:	... // seguito del ciclo
	modifica per ">", "<", ">=", "==", "!=" , "a--", "++a", "--a"
	idem se <b>a</b> è varloc, param od oggetto puntato



# break – troncamento di ciclo

```
// var. globale  
int a  
...  
// condizione  
while (...) {  
    ... // corpo I  
    if (a == 5) {  
        break  
    } /* end if */  
    ... // corpo II  
} /* end while */  
... // seguito
```

```
A:      .word      // riserva mem per a  
          ...  
LOOP:    ...      // condizione ciclo  
          ...      // corpo del ciclo I  
        lw $t0, A // carica a  
        li $t1, 5 // inizializza t1  
        // se a == 5 va' avanti a ENDLOOP  
        beq $t0, t1, ENDLOOP  
          ...      // corpo del ciclo II  
        j  LOOP     // va' indietro a LOOP  
ENDLOOP: ...      // seguito del ciclo  
va modificato come noto per ">", "<", ">=", "<=" e "!="  
idem se a è varloc, param od oggetto puntato  
costrutto valido per ogni tipo di ciclo
```



---

# come usare una variabile strutturata (array)



# considerazioni generali

---

- la variabile strutturata è un aggregato di elementi
  - accesso a singolo elemento con indice costante
  - scansione con indice o puntatore scorrevole
  - sono possibili diverse varianti e ottimizzazioni
- 
- vettori e *struct* hanno peculiarità rispettive
  - nel seguito pochi casi semplici ed esplicativi
  - solo per la classe di variabile globale



# vettore – accesso costante

```
// variabili globali
int v [5] // vettore
...
// accesso costante
v[0] = 6
...
v[2] = v[1] + 7
```

attenzione estremi vettore  
con aritmetica puntatori  
 $*(v + 1) = 4$   
ecc

```
// riserva  $5 \times 4 = 20$  byte per v
v: .space 20
// v[0] = 6
li $t0, 6          // inizializza $t0
la $t1, v          // ind. iniz. di v
sw $t0, 0($t1)    // v[0] = 6
// v[2] = v[1] + 7
la $t0, v          // inizializza $t0
lw $t1, 4($t0)    // carica v[1]
li $t2, 7          // inizializza $t2
add $t1, $t1, $t2 // v[1] + 7
sw $t1, 8($t0)    // memorizza v[2]
il compilatore in realtà espanderebbe la pseudo-istruzione
«la $t0, V» con indirizzo V in questo modo:
lui $t0, v
ori $t0, $t0, v (o con addiu)
ossia carica in t1 con «lui» i 16 bit più significativi di V
e poi concatena in t1 con «ori» (o con «addiu») i 16 bit
meno significativi di V
ottimizzazioni possibili trattando costanti e aritmetica
```



# vettore – scansione sequenziale con indice

```
// variabili globali
int v [5] // vettore
int a      // indice
...
// testata del ciclo
for (a = 2; a <= 6; a++) {
    v[a] = v[a] + 7
} /* end for */
// seguito del ciclo
...
attenzione estremi vettore
con aritmetica puntatori
// prova a farlo !
```

```
V:   .space 20           // 20 byte per v
A:   .word               // mem. per a
      // esegui a = 2 come già visto
FOR: lw    $t0, A          // carica a
      li    $t1, 6           // inizializza $t1
      bgt $t0, $t1, END // se a>6 va' a END
      la    $t0, V           // ind. iniz. di v
      lw    $t1, A          // carica a
      sll  $t1, $t1, 2        // allinea indice
      addu $t0, $t0, $t1 // indir. di v[a]
      lw    $t2, ($t0)       // carica v[a]
      li    $t3, 7           // inizializza $t3
      add  $t2, $t2, $t3 // v[a] + 7
      sw    $t2, ($t0)       // memorizza v[a]
      // esegui a++ come già visto
      j     FOR              // torna a FOR
END: ...                  // seguito ciclo
ottimizzazioni possibili trattando costanti e aritmetica
```



# vettore – scansione sequenziale con puntatore

```
// variabili globali
int v [5] // vettore
int a      // contatore
int * p    // puntatore
...
// testata del ciclo
p = v
for (a = 2; a <= 6; a++) {
    *p = *p + 7
    p++
} /* end for */
// seguito del ciclo
...
attenzione estremi vettore
```

```
v: .space 20           // 20 byte per v
A: .word               // mem. per a
P: .word               // mem. per p
la   $t0, v            // ind. iniz. di v
sw   $t0, p              // p = v
                // esegui a = 2 come già visto
FOR: // verifica a <= 6 come già visto
lw   $t0, P              // carica p
lw   $t1, ($t0)          // carica *p
li   $t2, 7               // inizializza $t0
add  $t1, $t1, $t2        // *p + 7
sw   $t1, ($t0)          // memorizza *p
lw   $t0, P              // (ri)carica p
addiu $t0, $t0, 4          // p++
sw   $t0, P              // memorizza p
                // esegui a++ come già visto
j    FOR                  // torna a FOR
END: ...                 // seguito ciclo
ottimizzazioni possibili trattando costanti e aritmetica
```



# come allocare in memoria gli elementi del vettore - da ind. bassi a ind. alti

indirizzi alti



sp

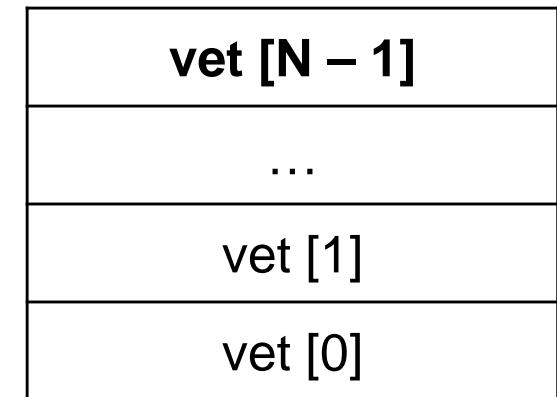
indirizzi bassi

vettore come  
variabile locale in pila

indirizzi alti

indirizzi bassi

segmento dati statici



vettore come  
variabile globale  
nel segmento  
dati statici



---

# **APPROFONDIMENTO**

## **come l'assemblatore tratta certe pseudo-istruzioni e il ruolo del collegatore (linker)**



## «*li* \$reg, costante»

---

La pseudo-istruzione «*li*» (*load immediate*) carica nel primo argomento *reg* il valore della costante numerica o simbolica specificata come secondo argomento. Per esempio:

.eqv VAL, 65538

**li** \$t0, VAL (oppure **li** \$t0, 65538)

poiché si ha *VAL* = 0x 0001 0002, ecco come è espansa:

**lui** \$t0, 0x0001

**addiu** \$t0, 0x0002 (oppure **ori** \$t0, 0x0002)

Il secondo argomento di «*li*» è una costante a 32 bit, che verrà caricata a 32 bit nel registro specificato come primo argomento. Si usa per inizializzare i registri di dato.

Va usata solo con costanti numeriche, oppure simboliche dichiarate con la direttiva **.eqv**! (affine a **#define** in ling. C).

---



## «**/w** \$reg, etichetta di indirizzo»

---

L'istruzione nativa «**/w**» ha il formato «**/w** \$t0, spi(\$t1)», dove il secondo argomento indirizza una cella di memoria.

Se però si scrive così, omettendo il registro base:

A: .word

lw \$t0, A

«**/w**» diventa una pseudo-istruzione ed è espansa così:

lw \$t0, A(\$gp)

dove A è lo spiazzamento rispetto al global pointer *gp*.

Similmente per *sw*!

Va usata solo per accedere a variabili globali scalari (non vettori o *struct*), dichiarate con direttive *.word* *.half* e *.byte*.

---



## «*/w \$reg, indirizzo simbolico» e linker*

---

Certi compilatori (p. es. gcc) sono esplicativi e scrivono così:

**lw** \$t0, %gp\_rel(A)(\$gp)

L'assemblatore codifica numericamente in modo parziale «*/w*», ma NON risolve l'etichetta di indirizzo A.

Lo farà invece il collegatore unendo tutti i moduli del programma e decidendo dove collocare i dati in memoria.

La notazione «%gp\_rel( )» è un comando passato al linker, di risolvere l'etichetta di indirizzo A come spiazzamento relativo al global pointer *gp*.

Il collegatore risolve il simbolo A come detto e completa la codifica numerica di «*/w*» con l'indirizzo numerico risolto.

---



## «*la* \$reg, indirizzo»

La pseudo-istruzione «*la*» (*load address*) carica nel primo argomento *reg* il valore dell'indirizzo numerico o simbolico specificato come secondo argomento. Per esempio:

A: .word

**la**       \$t0, A                                  (oppure **la** \$t0, 0x . . .)

P. es. se si ha A = 0x 0001 0002, ecco come è espansa:

**lui**       \$t0, 0x0001

**addiu** \$t0, 0x0002 (oppure **ori** \$t0, 0x0002)

Il secondo argomento di «*la*» è un indirizzo a 32 bit, che verrà caricato a 32 bit nel registro specificato come primo argomento. Si usa per inizializzare i registri puntatori.

Va usata solo con indirizzi numerici o con etichette di indirizzo (di solito di vettore), dichiarate con .space, . . . !



## «*la \$reg, indirizzo» e linker*

Certi compilatori (p. es. *gcc*) sono esplicativi ed espandono la pseudo-istruzione «*la \$t0, indirizzo» così:*

```
A:      .word
lui    $t0, %hi(A)
addiu $t0, %lo(A) (oppure ori $t0,%lo(A))
```

L'assemblatore codifica numericamente in modo parziale «*lui*» e «*addiu*», ma NON risolve l'etichetta di indirizzo A.

Lo farà invece il collegatore unendo tutti i moduli del programma e decidendo dove collocare i dati in memoria.

Le notazioni «%hi( )» e «%lo()» sono dunque comandi passati direttamente dall'assemblatore al collegatore (linker).

Il collegatore risolve il simbolo A come indirizzo virtuale assoluto e completa la codifica numerica di «*lui*» e «*addiu*» con le due parti dell'indirizzo numerico risolto.



## quali usi di queste pseudo-istruzioni

---

Per caricare (o memorizzare) una variabile globale scalare (non vettore o *struct*), come per esempio `char` oppure `int`:

**`lw`** (o **`sw`**) `$reg`, etichetta di indirizzo

Caso tipico: l'etichetta è il nome di una var. glob. scalare.

Per inizializzare un registro con una costante a 32 bit:

**`li`** `$reg`, costante numerica o simbolica

Per inizializzare un registro con un indirizzo a 32 bit:

**`la`** `$reg`, indir. numerico o etich. di indir.

Caso frequente: l'etichetta è il nome di un vettore, ossia è l'indirizzo ovvero il puntatore al primo elemento del vettore.

---



---

# **come rendere espressione e condizione (metodo sistematico)**



# calcolare l'espressione algebrica

- l'espressione algebrica modella un calcolo complesso
- va tradotta da linguaggio C a MIPS in modo sistematico
- calcola l'espressione algebrica nei registri di tipo  $t0 - t7$
- esamina l'espressione procedendo da sinistra verso destra, e scrivine il codice di calcolo così:
  - quando scandendo l'espressione trovi un operando (non costante), caricalo in un registro  $t0 - t7$  libero
  - non appena un operatore ha tutti i suoi operandi caricati nei reg, calcolalo, riusando il reg di numero minore per il risultato
- referenzia gli operandi (come hai visto prima) così:
  - secondo siano variabili globali, locali o parametri
  - oppure secondo siano variabili riferite per puntatore
- si suppone ci siano sempre sufficienti registri  $t0 - t7$  liberi



# esempio di espressione algebrica - I

$$a + b - c - a = ((a + b) - c) - a$$

le variabili *a*, *b* e *c* sono globali (in memoria)

il tipo int è da 32 bit

l'espressione è associata da sinistra verso destra

il calcolo è svolto nei registri temporanei

// dich.

**A:** .word

**B:** .word

**C:** .word

// si riusa il reg con numero minore

**lw** \$t0, **A** // 1: carica a

**lw** \$t1, **B** // 2: carica b

**add** \$t0, \$t0, \$t1 // 3: a+b

**lw** \$t1, **C** // 4: carica c

**sub** \$t0, \$t0, \$t1 // 5: (a+b)-c

**lw** \$t1, **A** // 6: (ri)carica a

**sub** \$t0, \$t0, \$t1 // 7: ((a+b)-c)-a

// il risultato è nel reg t0

ottimizza senza ricaricare **a** ...

modifica se var locale o parametro o puntatore



## esempio di espressione algebrica - II (espressione diversa dalla precedente)

$$a + (b - (c - a))$$

le variabili *a*, *b* e *c* sono globali (in memoria)

il tipo *int* è da 32 bit

l'espressione ha già un ordine di calcolo, non strettamente da sinistra verso destra, imposto dalle parentesi

il calcolo è svolto nei registri temporanei

le parentesi influenzano il numero di registri usati

// dich.

**A:** .word

**B:** .word

**C:** .word

// si riusa il reg con numero minore

lw \$t0, **A** // 1: carica a

lw \$t1, **B** // 2: carica b

lw \$t2, **C** // 3: carica c

lw \$t3, **A** // 4: (ri)carica a

sub \$t2, \$t2, \$t3 // 5: c-a

sub \$t1, \$t1, \$t2 // 6: b-(c-a)

add \$t0, \$t0, \$t1 // 7: a+(b-(c-a))

// il risultato è nel reg t0

ottimizza senza ricaricare la variabile **a** ...

modifica se var locale o parametro o puntatore



# esempio di espressione algebrica - III

## (espressione con moltiplicazione)

$$a + b \times c - a = (a + (b \times c)) - a$$

le variabili *a*, *b* e *c* sono globali (in memoria)

il tipo *int* è da 32 bit

l'espressione contiene una moltiplicazione, che per convenzione ha precedenza rispetto all'addizione iniziale

il resto dell'espressione è associato da sinistra verso destra

il calcolo è svolto nei registri temporanei

// dich.

**A:** .word

**B:** .word

**C:** .word

// si riusa il reg con numero minore

**lw** \$t0, **A** // 1: carica a

**lw** \$t1, **B** // 2: carica b

**lw** \$t2, **C** // 3: carica c

**mult** \$t1, \$t2 // 4.a: bxc

**mflo** \$t1 // 4.b: ris in t1

**add** \$t0, \$t0, \$t1 // 5: a+(bxc)

**lw** \$t1, **A** // 6: (ri)carica a

**sub** \$t0, \$t0, \$t1 // 7: (a+(bxc))-a

// il risultato è nel reg t0

ottimizza senza ricaricare la variabile **a** ...

modifica se var locale o parametro o puntatore

le istruzioni 4.a e 4.b si possono compattare con la pseudo-istruzione «**mul** \$t1, \$t1, \$t2»



# esempio di espressione algebrica - IV

## (espressione con funzione)

$$a + f(\dots) - b = (a + f(\dots)) - b$$

le variabili *a* e *b* sono globali  
(in memoria)

il tipo int è da 32 bit

l'espressione contiene una  
funzione, che va calcolata  
secondo la nota sequenza

il resto dell'espressione è  
associato da sinistra verso  
destra, come già visto

il calcolo è svolto nei registri  
temporanei

salva in pila i registri  
temporanei da ripristinare  
per poi continuare il calcolo

```
// dich.  
A: .word          // si riusa il reg con numero minore  
B: .word          // 1: carica a  
// testo           // 2.a: passa parametri ... a f()  
...               // 2.b: salva t0 in pila (push)  
// funzione        // 2.c: chiama f()  
F: ...            // 2.d: ripristina t0 da pila (pop)  
add $t0, $t0, $v0 // 3: a+f(...)  
lw $t1, B         // 4: carica b  
sub $t0, $t0, $t1 // 5: (a+f(...))-b  
// il risultato è nel reg t0  
modifica se var locale o parametro o puntatore
```



# verificare la condizione algebrica

---

- la condizione algebrica modella un confronto complesso
  - contiene una o due espressioni algebriche da confrontare
  - operatori di confronto numerico: “=”, “≠”, “<”, “>”, “≤” e “≥”
- 
- calcola le due espressioni separatamente
  - confrontane i risultati e salta condizionatamente
  - per calcolare tali espressioni, ricorri al metodo



# esempio di condizione algebrica

sufficienti registri  $t0-t7$  liberi – intero 32 bit

```
...
// cond. algebrica
if (espr1 == espr2) {
    // ramo then
    ...
} /* end if */
// seguito
...
// calcola espr1 - ris. in t1
// calcola espr2 - ris. in t2
// se espr1 != espr2 va' a END
bne $t1, $t2, END
THEN: ...
END: ...
// ramo then
// seguito del condiz.
```

va modificato come noto per “>”, “<”, “ $<=$ ”, “ $=$ ” e “ $!=$ ”,  
nonché per le altre strutture di controllo  
e vale anche per “ $\&\&$ ”, “| |” e “!” (AND OR e NOT)



# cenno a espressione e condizione logica

- spesso la struttura di controllo condizionale e il ciclo a condizione sono governati da un'espressione logica
- in teoria l'espressione logica con variabili booleane si calcola **analogamente** a quella algebrica (intera o reale)
- usa le istruzioni macchina logiche AND, OR e NOT
- però in compilazione l'espressione logica è resa **più efficientemente** come flusso di controllo del programma
- ossia al calcolo dell'espressione logica corrisponde una serie di istruzioni macchina di salto condizionato «*bxx*»
- lo schema di traduzione è **più complesso** e qui è solo esemplificato – spesso basta procedere intuitivamente



# esempio di condizione logica – I

**a** || **b** && !**c**

```
// cond. logica
if (a || b && !c) {
    ... // ramo then
} /* end if */
... // seguito
```

riscrivi il codice C come a destra, usando la tecnica di “shortcut” (scorciatoia) per ogni operatore verifica il caso che risolve subito la condizione (vera o falsa) e non esaminare il resto

```
// “shortcut” con catena di “if-goto”
// se a ≠ 0 la cond. è vera
if      (a != 0) goto THEN
// altrimenti se b = 0 la cond. è falsa
else if (b == 0) goto END
// altrimenti se c = 0 la cond. è vera
else if (c == 0) goto THEN
// altrimenti la cond. è senz'altro falsa
else           goto END
THEN: ...
END: ...
si generalizza facilmente al caso if-then-else
```



# esempio di condizione logica – II

**a** || **b** && !**c**

```
// condizione logica
// catena di "if-goto"
if      (a != 0) goto THEN
else if  (b == 0) goto END
else if  (c == 0) goto THEN
else          goto END
THEN: ... // ramo then
END: ... // seguito
```

```
A: .word
B: .word
C: .word
```

**a, b, c** var glob – **int** a 32 bit

```
lw    $t0, A    // carica a in t0
// se t0 != 0 va' a THEN
bne  $t0, $zero, THEN
lw    $t0, B    // carica b in t0
// se t0 == 0 va' a END
beq  $t0, $zero, END
lw    $t0, C    // carica c in t0
// se t0 == 0 va' a THEN
beq  $t0, $zero, THEN
b    END        // va' a END
THEN: ...           // ramo then
END: ...           // seguito
```



---

# **come chiamare la funzione e rientrare (area di attivazione)**



# considerazioni generali

---

- prima occorre calcolare la dimensione in byte dell'area di attivazione della funzione, secondo il contenuto che dovrà avere in ragione delle convenzioni adottate
  - la chiamata a funzione comporta
    - prologo del chiamante
    - salto a chiamato
    - prologo del chiamato
    - corpo elaborativo del chiamato
    - epilogo del chiamato
    - rientro a chiamante
    - epilogo del chiamante
  - alcuni passaggi possono ridursi a poco, secondo le convenzioni o la situazione specifica
-



# sequenza di chiamata / rientro a funzione

## □ prologo del chiamante

- salva sulla pila i registri  $a0-a3$  che devi o preferisci (tu chiamante) riavere inalterati dopo che la funzione sarà rientrata
- salva sulla pila i registri  $t0-t7$  che devi o preferisci (tu chiamante) riavere inalterati dopo che la funzione sarà rientrata
- salva sulla pila i registri  $v0-v1$  che devi o preferisci (tu chiamante) riavere inalterati dopo che la funzione sarà rientrata
- scrivi nei registri  $a0 - a3$  i paramenti in ingresso alla funzione
  - trascuriamo il caso di più di quattro parametri
- metti sulla pila gli eventuali parametri aggiuntivi ai primi quattro in ingresso alla funzione
  - caso raro qui trascurato

## □ salto a chiamato: **jal** funz



# sequenza di chiamata / rientro a funzione

## □ prologo del chiamato

- crea area di attivazione:  
**addiu \$sp, \$sp, -dim. area in byte**
- se il registro *fp* è in uso
  - salvalo nell'area di attivazione in pila
  - aggiornalo in modo da puntare al registro salvato
- se la funzione non è foglia, salva nell'area di attivazione in pila il registro *ra*
- salva nell'area di attivazione in pila i registri *s0 – s7* assegnati a variabili locali

## □ corpo elaborativo del chiamato



# sequenza di chiamata / rientro a funzione

## □ epilogo del chiamato

- scrivi nel registro *v0* il valore di uscita
- ripristina dalla pila i registri *s0 – s7* assegnati a variabili locali
- se la funzione non è foglia, ripristina dalla pila il registro *ra*
- se il registro *fp* è in uso
  - ripristinalo dalla pila
- elimina area di attivazione:  
**addiu \$sp, \$sp, dim. area in byte**

## □ rientro a chiamante: **jr \$ra**



# sequenza di chiamata / rientro a funzione

## □ epilogo del chiamante

- ripristina dalla pila i registri  $v0 - v1$ ,  $t0 - t7$  e  $a0 - a3$  che avevi dovuto o preferito salvare prima della chiamata
- trova nel registro  $v0$  il valore di uscita dalla funzione



## osservazioni e avvertimenti

---

- i compilatori più diffusi per MIPS possono adottare convenzioni un po' differenti, sia pure di poco
- per esempio GNU CC (GCC)
  - gestisce il registro *fp* in modo leggermente diverso, salvandolo in pila nella posizione qui descritta, ma facendolo puntare alla cima della pila come il registro *sp*
  - l'area di attivazione ha una dimensione minima di 24 byte, anche se parte di essa resta inutilizzata
  - ottimizza fortemente l'uso dei registri, senza rispettare rigorosamente le convenzioni di uso se ciò non causa errori
- in generale esistono più ABI (Application Binary Interface) per il processore MIPS, differenti e variamente utilizzate



---

# riassunto (pseudo)istruzioni e direttive importanti



# istruzioni e pseudo-istruzioni

## ARITMETICA

add	\$s1, \$s1, \$s3	$s1 := s2 + s3$	addizione
addu	\$s1, \$s1, \$s3	$s1 := s2 + s3$	addizione naturale
addi	\$s1, \$s2, cost	$s1 := s2 + \text{cost}$	addizione di costante
addiu	\$s1, \$s2, cost	$s1 := s2 + \text{cost}$	addizione naturale di costante
sub	\$s1, \$s2, \$s3	$s1 := s2 - s3$	sottrazione
subu	\$s1, \$s2, \$s3	$s1 := s2 - s3$	sottrazione naturale
mult	\$s1, \$s2	hi lo := $s1 \times s2$	moltiplicazione (risultato a 64 bit)

## ARITMETICA – pseudo-istruzioni

subi	\$s1, \$s2, cost	$s1 := s2 - \text{cost}$	sottrazione di costante
subiu	\$s1, \$s2, cost	$s1 := s2 - \text{cost}$	sottrazione naturale di costante
neg	\$s1, \$s2	$s1 := -s2$	negazione aritmetica (compl. a 2)



# istruzioni e pseudo-istruzioni

## CONFRONTO

slt	\$s1, \$s2, \$s3	<b>if</b> s2 < s3 <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se strettamente minore
sltu	\$s1, \$s2, \$s3	<b>if</b> s2 < s3 <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se strettamente minore naturale
slti	\$s1, \$s2, cost	<b>if</b> s2 < cost <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se strettamente minore di costante
sltiu	\$s1, \$s2, cost	<b>if</b> s2 < cost <b>then</b> s1 := 1 <b>else</b> s1 := 0	poni a 1 se strettamente minore di costante naturale



# istruzioni e pseudo-istruzioni

## LOGICA

or	\$s1, \$s2, \$s3	s1 := s2 <b>or</b> s3	somma logica bit a bit
and	\$s1, \$s2, \$s3	s1 := s2 <b>and</b> s3	prodotto logico bit a bit
ori	\$s1, \$s2, cost	s1 := s2 <b>or</b> cost	somma logica bit a bit con costante
andi	\$s1, \$s2, cost	s1 := s2 <b>and</b> cost	prodotto logico bit a bit con costante
nor	\$s1, \$s2, \$s3	s1 := s2 <b>nor</b> s3	somma logica negata bit a bit
sll	\$s1, \$s2, cost	s1:= s2 << cost	scorrimento a sinistra (left) del n° di bit specificato da cost
srl	\$s1, \$s2, cost	s1:= s2 >> cost	scorrimento a destra (right) del n° di bit specificato da cost

## LOGICA – pseudo-istruzioni

not	\$s1, \$s2	s1 = <b>not</b> s2	negazione logica
-----	------------	--------------------	------------------



# istruzioni e pseudo-istruzioni

## SALTO INCONDIZIONATO, DA REGISTRO E CON COLLEGAMENTO

j	indir	PC := indir (28 bit)	salto incondizionato assoluto
jr	\$r	PC := \$r (32 bit)	salto indiretto da registro
jal	indir	PC := indir (28 bit) e collega \$ra	salto assoluto e collegamento

## SALTO CONDIZIONATO

beq	\$s1, \$s2, spi	if s2 = s1 salta relativo a PC	salto cond. di uguaglianza
bne	\$s1, \$s2, spi	if s2 ≠ s1 salta relativo a PC	salto cond. di disuguaglianza

## SALTO CONDIZIONATO – pseudo-istruzioni

blt	\$s1, \$s2, spi	if s2 < s1 salta relativo a PC	salta se strettamente minore
bgt	\$s1, \$s2, spi	if s2 > s1 salta relativo a PC	salta se strettamente maggiore
ble	\$s1, \$s2, spi	if s2 ≤ s1 salta relativo a PC	salta se minore o uguale
bge	\$s1, \$s2, spi	if s2 ≥ s1 salta relativo a PC	salta se maggiore o uguale



# istruzioni e pseudo-istruzioni

## TRASFERIMENTO PROCESSORE-MEMORIA

lw	\$s1, spi (\$s2)	s1 := mem (s2 + spi)	carica parola (a 32 bit)
sw	\$s1, spi (\$s2)	mem (s2 + spi) := s1	memorizza parola (a 32 bit)
lh, lhu	\$s1, spi (\$s2)	s1 := mem (s2 + spi)	carica mezza parola (a 16 bit)
sh	\$s1, spi (\$s2)	mem (s2 + spi) := s1	memor. mezza parola (a 32 bit)
lb, lbu	\$s1, spi (\$s2)	s1 := mem (s2 + spi)	carica byte (a 8 bit)
sb	\$s1, spi (\$s2)	mem (s2 + spi) := s1	memorizza byte (a 8 bit)

## TRASFERIMENTO in registro di COSTANTE

lui	\$s1, cost	s1 (16 bit più signif.) := cost	carica cost (in 16 bit più signifi)
-----	------------	---------------------------------	-------------------------------------

## TRASFERIMENTO tra REGISTRI (non referenziabili)

mflo	\$s1	s1 := lo	copia registro lo
mfhi	\$s1	s1 := hi	copia registro hi

## TRASFERIMENTO tra REGISTRI e di COSTANTI / INDIRIZZI – pseudo-istruzioni

move	\$d, \$s	d := s	copia registro
la	\$d, indir	d := indir (32 bit)	carica indirizzo a 32 bit
li	\$d, cost	d := cost (32 bit)	carica costante a 32 bit



# registri

## REGISTRI REFERENZIABILI

0	0	costante 0 (denotabile anche come \$zero)
1	at	uso riservato all'assembler-linker (per espandere pseudo-istruzioni e macro)
2 - 3	v0 - v1	valore restituito da funzione (v0 dati di tipo scalare, v1 si aggiunge per float)
4 - 7	a0 - a3	argomenti in ingresso a funzione (max quattro)
8 - 15	t0 - t7	registri per valori temporanei (p.es. calcolo delle espressioni)
16 - 23	s0 - s7	registri (usualmente per variabili locali di tipo scalare di sottoprogramma)
24 - 25	t8 - t9	registri per valori temporanei (in aggiunta a t0 - t7), come i precedenti tx
26 - 27	k0 - k1	registri riservati per il nucleo del SO
28	gp	global pointer (puntatore all'area dati globale)
29	sp	stack pointer (puntatore alla pila)
30	fp	frame pointer (puntatore all'area di attivazione di sottoprogramma)
31	ra	return address (indirizzo di rientro da chiamata a sottoprogramma)

## REGISTRI NON REFERENZIABILI

	pc	program counter
	hi	registro per risultato di moltiplicazione e divisione (32 bit più significativi)
	lo	registro per risultato di moltiplicazione e divisione (32 bit meno significativi)



# direttive di assemblatore

## Direttive fondamentali

- .align n        allinea il dato dichiarato di seguito secondo l'argomento "n":  
                n=0 byte, n=1 mezza parola, n=2 parola e n=3 parola lunga
- .ascii s        riserva spazio per la stringa "s" nel segmento dati,  
                senza aggiungere il terminatore di stringa
- .asciiz s        riserva spazio per la stringa "s" nel segmento dati  
                e aggiungi il terminatore di stringa
- .byte n1, ...    riserva spazio nel segmento dati per i byte elencati  
                e inizializzali con i valori a 8 bit "n1", ...
- .data i        dichiara il segmento dati con indirizzo iniziale "i"  
                (default 0x 1000 0000)
- .eqv s, v        dichiara un simbolo: il simbolo "s" ha valore numerico "v",  
                senza allocare memoria (funziona come la #define in C ed è  
                usato soprattutto per definire lo spiazzamento in pila)
- .globl s1, ...    dichiara che i simboli "s1", ... elencati sono globali  
                e permetti di riferirsi ad essi in altri moduli o file  
                (indispensabile per dichiarare almeno "main")



# direttive di assemblatore

---

```
.half h1, ... riserva spazio nel segmento dati per le mezze parole elencate
e inizializzate con i valori a 16 bit "h1", ..., allineandole
a indirizzi pari (con .align si può cambiare allineamento)

.space n      riserva spazio nel segmento dati per "n" byte,
               senza inizializzarli

.text i        dichiara il segmento testo (codice) con indirizzo iniziale "i"
               (default 0x 0040 0000)

.word w1, ...  riserva spazio nel segmento dati per le parole elencate
e inizializzate con i valori a 32 bit "w1", ..., allineandole
a indirizzi multipli di quattro (.align cambia allineamento)
```

## Altre direttive

```
.kdata i      dichiara il segmento dati di kernel con indirizzo iniziale "i"

.ktext i      dichiara il segmento testo di kernel con indirizzo iniziale "i"

.float f1, ... riserva spazio nel seg. dati per i numeri reali elencati

.macro a1, ... dichiarazione di macro con argomenti "a1", ...

.end_macro    fine dichiarazione di macro
```

---



---

# riassunto su come accedere a variabili



# variabile globale (nel segmento dati)

C	assemblatore semplificato	espansione in assemblatore nativo (gcc)
int a  leggi a	A: .word  lw \$t0, A // var. a caricata in t0	A: .word  lw \$t0, A(\$gp) // var. a caricata in t0
int vet [5]  leggi vet [1]	VET: .space 20  la \$t0, VET  lw \$t1, 4(\$t0) // elem. vet[1] caricato in t1	VET: .space 20  lui \$t0, VET addiu \$t0, \$t0, VET lw \$t1, 4(\$t0) // elem. vet[1] caricato in t1
int i  int vet [5]  leggi vet [i]	// i caricato in t1  VET: .space 20  la \$t0, VET  sll \$t1, \$t1, 2 addu \$t0, \$t0, \$t1 lw \$t2, (\$t0) // elem. vet[i] caricato in t2	// i caricato in t1  VET: .space 20  lui \$t0, VET addiu \$t0, \$t0, VET sll \$t1, \$t1, 2 addu \$t0, \$t0, \$t1 lw \$t2, (\$t0) // elem. vet[i] caricato in t2



# variabile locale (nell'area di attivazione)

C	assemblatore semplificato	espansione in assemblatore nativo (gcc)
<pre>void f () {     int a      leggi a }</pre>	<pre>// spi. di a in area attiv. .eqv A, ...  lw \$t0, A(\$sp) // var. a caricata in t0</pre>	<pre>// spi. di a in area attiv. .eqv A, ... // spi. di a in area  lw \$t0, A(\$sp) // var. a caricata in t0</pre>
<pre>void f () {     int vet [5]      leggi vet [1] }</pre>	<pre>// spi. di vet in area attiv. .eqv VET, ...  move \$t0, \$sp addiu \$t0, VET lw \$t1, 4(\$t0) // elem. vet[1] caricato in t1</pre>	<pre>// spi. di vet in area attiv. .eqv VET, ...  addu \$t0, \$zero, \$sp addiu \$t0, VET lw \$t1, 4(\$t0) // elem. vet[1] caricato in t1</pre>
<pre>int i void f () {     int vet [5]      leggi vet [i] }</pre>	<pre>// i caricato in t1 // spi. di vet in area attiv. .eqv VET, ...  move \$t0, \$sp addiu \$t0, \$t0, VET sll \$t1, \$t1, 2 addu \$t0, \$t0, \$t1 lw \$t2, (\$t0) // elem. vet[i] caricato in t2</pre>	<pre>// i caricato in t1 // spi. di vet in area attiv. .eqv VET, ...  addu \$t0, \$zero, \$sp addiu \$t0, \$t0, VET sll \$t1, \$t1, 2 addu \$t0, \$t0, \$t1 lw \$t2, (\$t0) // elem. vet[i] caricato in t2</pre>



**POLITECNICO**  
MILANO 1863

# Architettura dei calcolatori e sistemi operativi

## Espressioni algebriche

### Capitolo 2 P&H

# Le espressioni algebriche

I compilatori traducono in linguaggio macchina la valutazione di un'espressione algebrica affrontando in modo ottimizzato i seguenti passi schematici:

- derivazione dell'albero sintattico dell'espressione
- ordinamento dei nodi dell'albero
- assegnamento dei registri

***noi adottiamo una «regola» semplice e intuitiva***

che non richiede la derivazione dell'albero sintattico  
e non ottimizza la traduzione in modo completo



# Valutazione di espressioni algebriche

espressione:  $a + b - 5 - (d - a) + c$

ma anche:  $a + b - (c + \text{funz}(b - c, d))$

Procedimento:

- completa l'espressione con le parentesi associando da sinistra
- ripeti fino ad avere esaminata tutta l'espressione
  - ✓ valuta il primo operatore valutabile (da sinistra)
  - ✓ un operatore è valutabile se le sue parti sinistra e destra sono:
    - ✓ una variabile
    - ✓ una costante
    - ✓ una sottoespressione già calcolata in un registro di tipo  $t$

**Nota:** le operazioni aritmetiche eseguono solo su registri

- ✓ se una variabile è in memoria deve essere prima caricata in un registro
- ✓ se è associata a un registro (argomenti, var locali ...) si usa il registro senza modificarlo



## Esempio 1: con variabili globali

espressione:  $a + b - 5 - (d - a) + c$

- Completa l'espressione con le parentesi associando da sinistra (potremmo anche omettere la parentesi più esterna)

$$((((a + b) - 5) - (d - a)) + c)$$

- Valuta l'espressione secondo le parentesi

```
lw    $t0, A      # carica a
lw    $t1, B      # carica b
add  $t0, $t0, $t1 # valuta ( )
subi $t0, $t0, 5  # valuta ( )
lw    $t1, D      # carica d
lw    $t2, A      # (ri)carica a
sub  $t1, $t1, $t2 # valuta ( )
sub  $t0, $t0, $t1 # valuta ( )
lw    $t1, C      # carica c
add  $t0, $t0, $t1 # valuta ( )
```

Regole implicite utilizzate:

- si riscrive il risultato nel registro con indice minore
- si riutilizzano i registri



## Esempio 2: con variabili globali e chiamata a sottoprogramma

espressione:  $a + b - (c + \text{funz}(b - c, d))$

- Completa l'espressione con le parentesi associando da sinistra

$$((a + b) - (c + \text{funz}(b - c, d)))$$

- Valuta l'espressione (rigorosamente da sinistra a destra)

```
lw      $t0, A          # carica a
lw      $t1, B          # carica b
add   $t0, $t0, $t1    # valuta ( )
lw      $t1, C          # carica c (figura a sinistra di funz (...))
# inizio sequenza di chiamata a sottoprogramma
lw      $t2, B          # (ri)carica b
lw      $t3, C          # (ri)carica c
sub   $a0, $t2, $t3    # calcola primo parametro di funz
lw      $a1, D          # carica secondo parametro di funz
addiu $sp, $sp, -4     # push di t0 (usato dopo chiamata a funz)
sw    $t0, ($sp)
addiu $sp, $sp, -4     # push di t1 (usato dopo chiamata a funz)
sw    $t1, ($sp)
jal   FUNZ            # chiama funz
lw      $t1, ($sp)      # pop di t1 (per continuare il calcolo)
addiu $sp, $sp, 4
lw      $t0, ($sp)      # pop di t0 (per continuare il calcolo)
addiu $sp, $sp, 4
# fine sequenza di chiamata a sottoprogramma
add   $t1, $t1, $v0    # valuta ( )
sub   $t0, $t0, $t1    # valuta ( )
```

