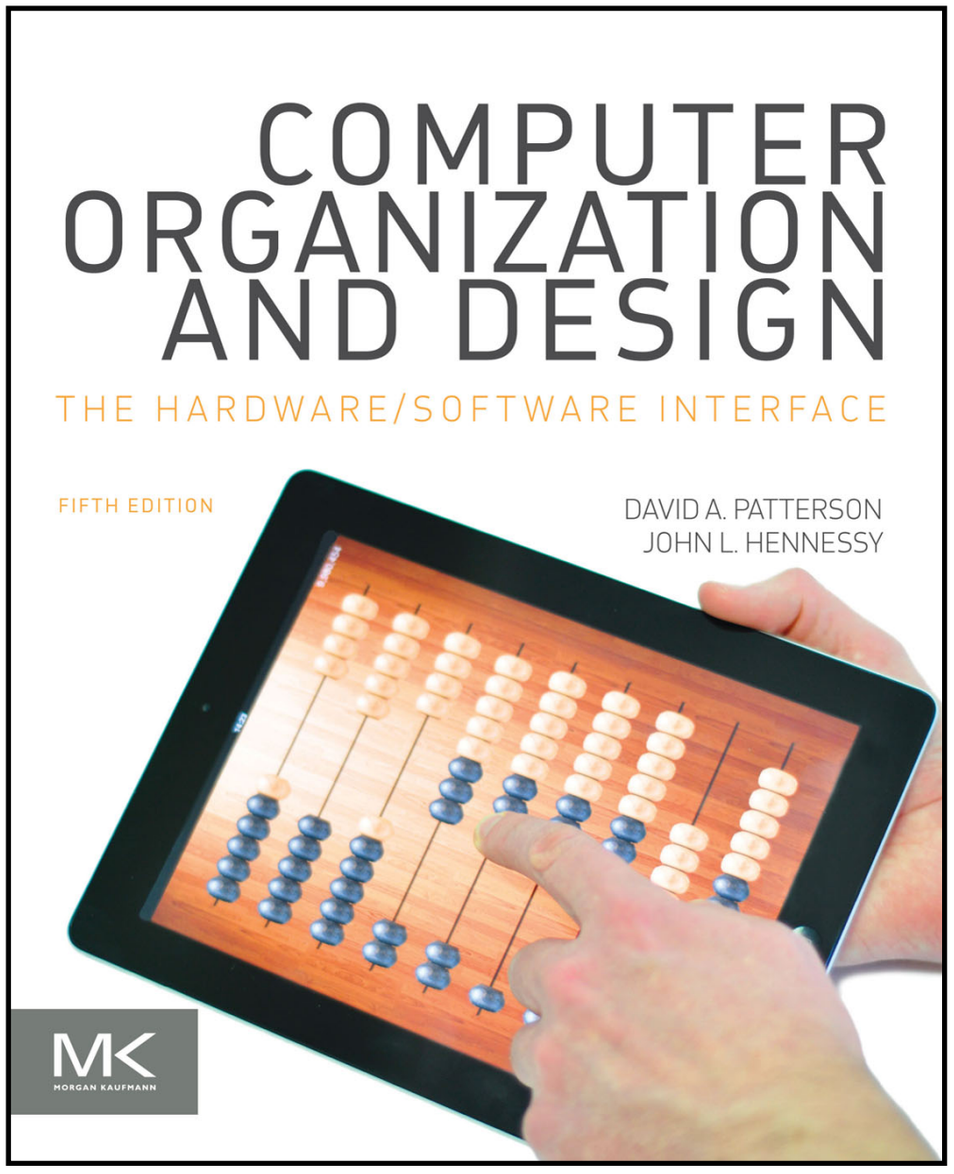
Sistemi di Elaborazione

Accademia Navale

nicola.tonellotto@unipi.it

Libro di testo

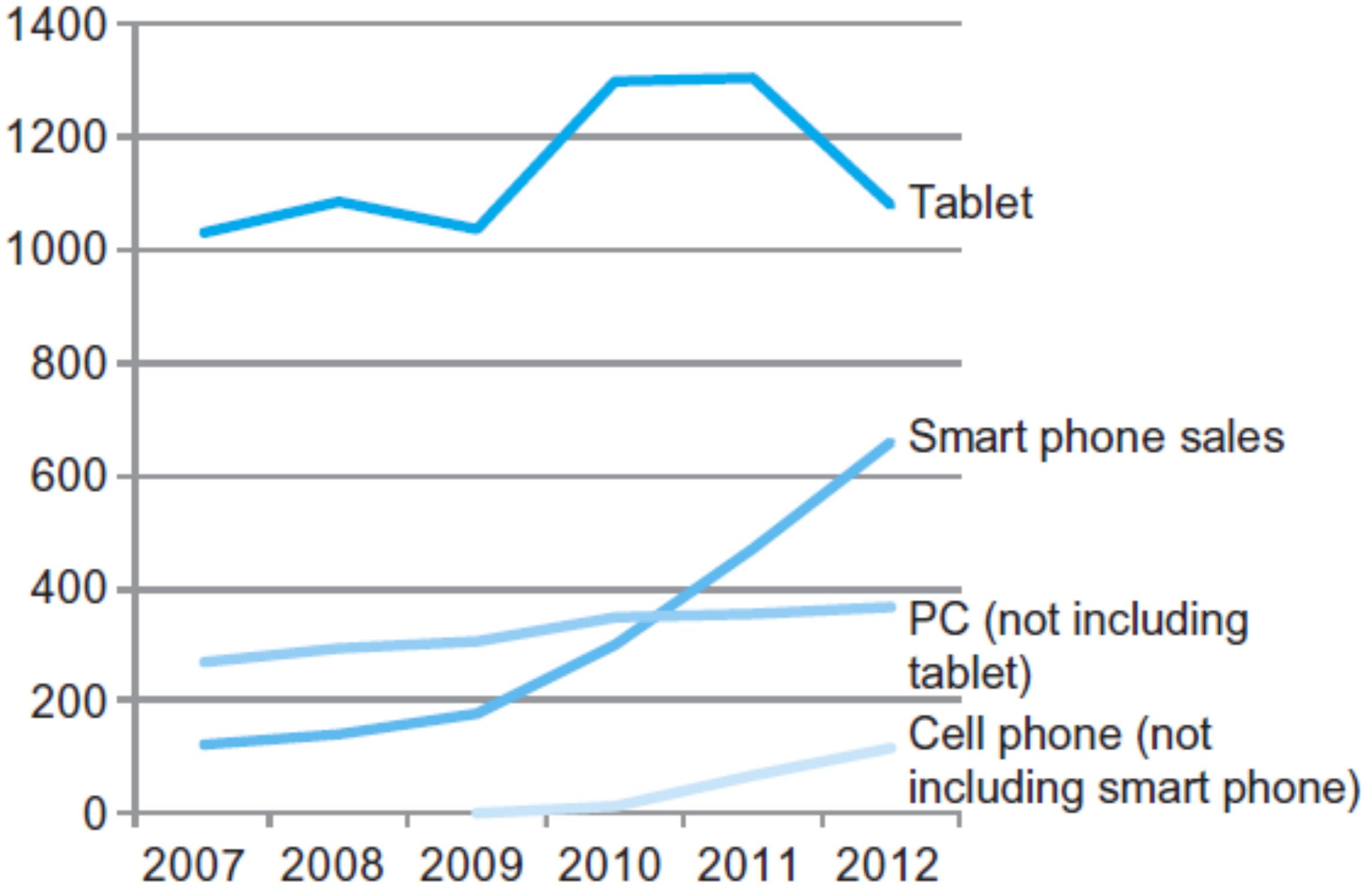


Capitolo 1

Classi di Computer

- **Personal Computer**
  - Scopi generali, molti software diversi
  - Soggetto a tradeoff costi/prestazioni
- **Server**
  - Basati su collegamenti di rete
  - Elevata capacità, prestazioni, affidabilità
  - Da piccoli server a interi capannoni
- **Supercomputer**
  - Complesse elaborazioni scientifiche/ingegneristiche
  - Massima capacità, ma rappresentano una piccola frazione del mercato dei computer
- **Computer Dedicati (*Embedded*)**
  - Nascosti come componenti di sistemi complessi
  - Rigorosi vincoli su potenza/prestazioni/costi

Era PostPC


1400
1200
1000
800
600
400
200
0
2007
2008
2009
2010
2011
2012
Tablet
Smart phone sales
PC (not including tablet)
Cell phone (not including smart phone)

Era PostPC

- **Dispositivi Mobile Personali**
  - Alimentati a batteria
  - Connessi a Internet
  - Poche centinaia di euro
  - Smart Phone, Tablet, Smart Watch
- **Cloud Computing**
  - Computer grandi quanto magazzini (*warehouse scale computers, WSC*)
  - Software come servizio (*Software as a Service, SaaS*)
  - Parte del software è in esecuzione su un dispositivo mobile personale e parte sul Cloud
  - Amazon, Google, Microsoft

Comprendere le prestazioni

- **Algoritmo**
  - Determina il numero di operazioni eseguite
- **Linguaggio di programmazione, compilatore, architettura**
  - Determina il numero di istruzioni macchina eseguite per operazione
- **Processore e memoria**
  - Determinano quanto velocemente le istruzioni sono eseguire
- **Sistema di I/O (incluso il SO)**
  - Determina quanto velocemente le operazioni di I/O sono eseguite

Legge di Moore

Il numero di transistor su un chip raddoppierà ogni anno

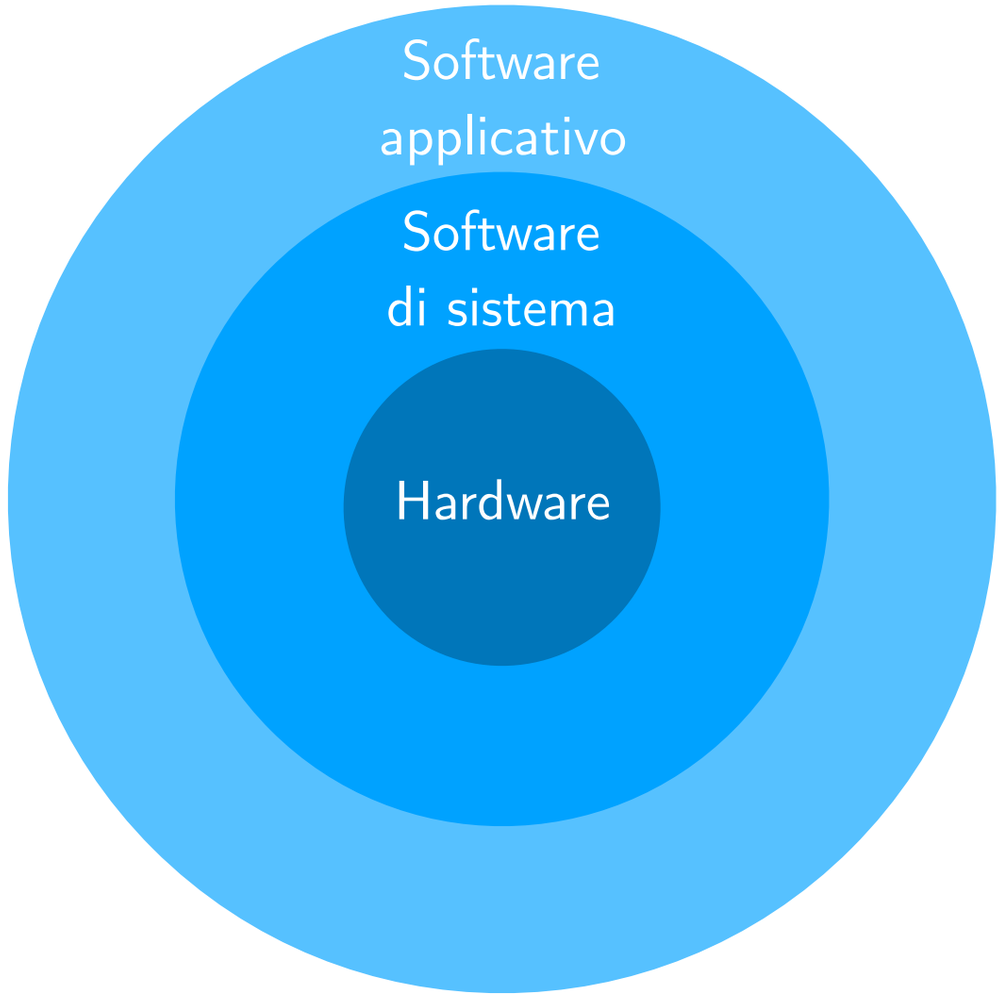
- Il numero di transistor su un chip raddoppia ogni 18 mesi
- Il costo di un chip è rimasto pressoché invariato
- Maggior densità di transistor su chip significa linee elettriche più corte, con maggiori prestazioni
- La minor dimensione aumenta la flessibilità
- Ridotti requisiti di potenza di alimentazione e di raffreddamento
- Un minor numero di interconnessioni aumenta l'affidabilità
- Gordon Moore è stato il co-fondatore della Intel

Otto Grandi Idee

- Progettare per la **Legge di Moore**
- Usare **astrazioni** per semplificare il progetto
- Rendere il **caso comune veloce**
- Prestazioni tramite **parallelismo**
- Prestazioni tramite **pipelining**
- Prestazioni tramite **predizione**
- **Gerarchia** di memoria
- **Affidabilità** tramite ridondanza

Dietro al vostro programma

- **Software applicativo**
  - Scritto in un linguaggio ad alto livello (HLL)
- **Software di sistema**
  - **Compilatore**: traduce il codice HLL in codice macchina
  - **Sistema Operativo**: codice di servizio
    - Gestisce input/output
    - Gestisce la memoria e l'archiviazione
    - Schedula i task e gestisce la condivisione delle risorse
- **Hardware**
  - Processore, memoria, controllori I/O

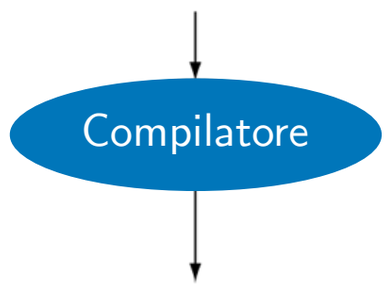


Livelli del codice di un programma

- **Linguaggio ad alto livello**
  - Livello di astrazione più vicino al dominio del problema
  - Fornisce produttività e portabilità
- **Linguaggio Assembly**
  - Rappresentazione testuale delle istruzioni
- **Rappresentazione Hardware**
  - Simboli binari (bit)
  - Istruzioni e dati codificati

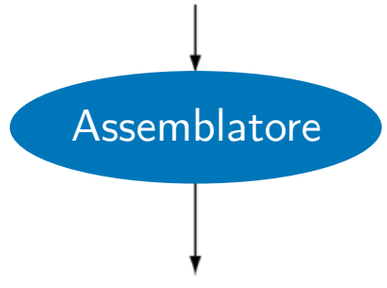
Programma in linguaggio ad alto livello (in C/C++)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



Programma in linguaggio Assembly (per MIPS)

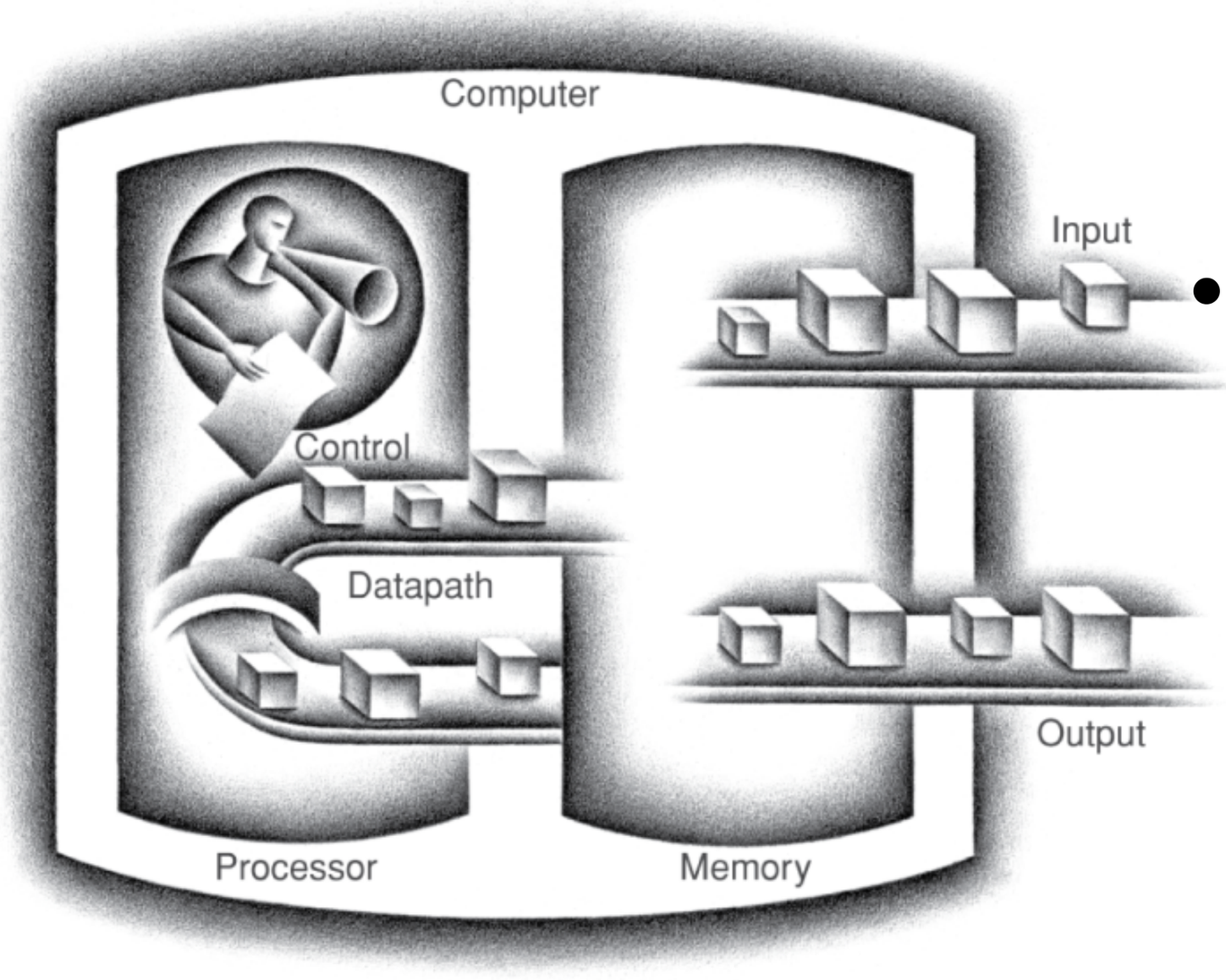
```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Programma in linguaggio macchina binario (per MIPS)

```
00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Componenti di un Computer

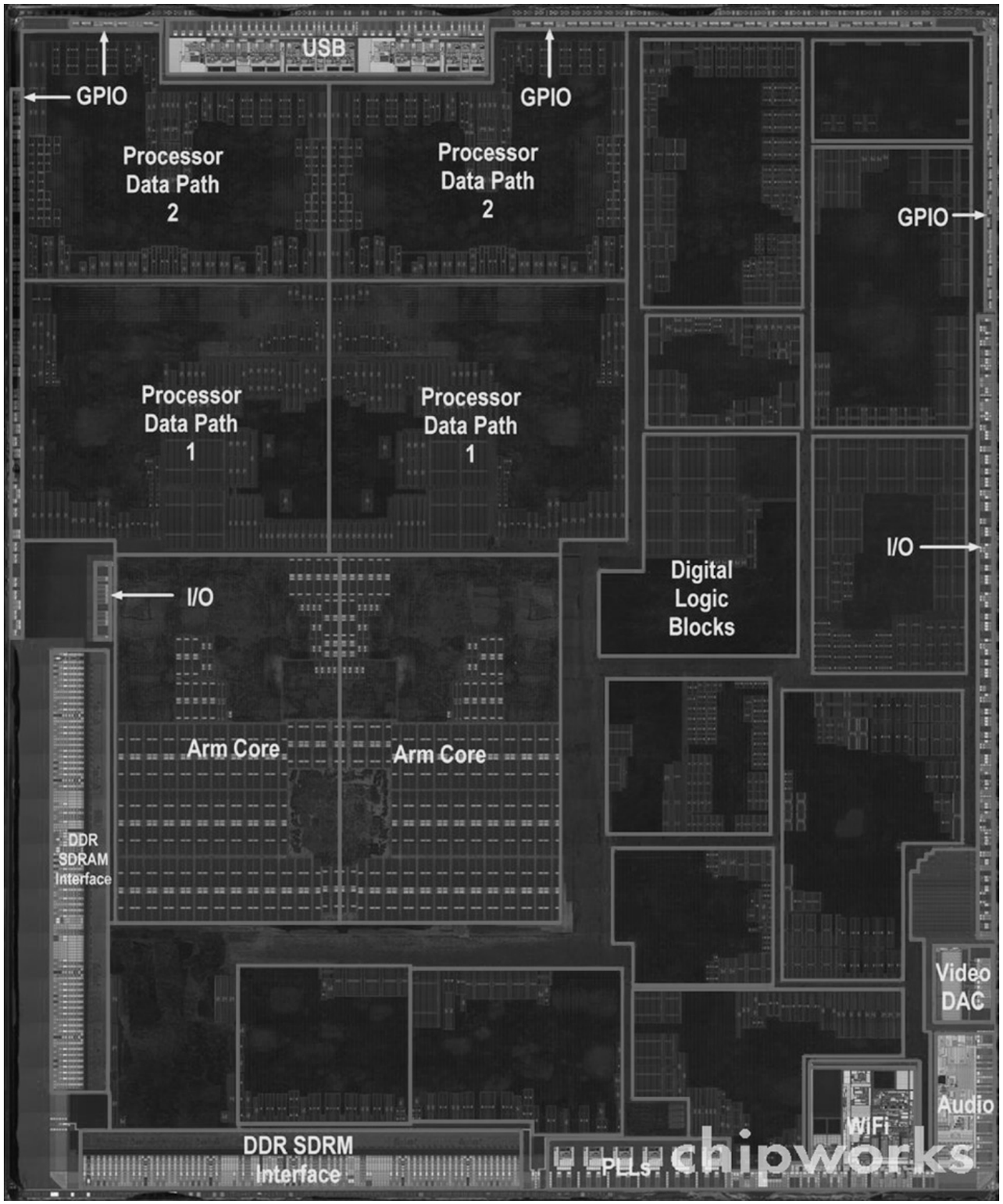


- Stessi componenti per tutti i tipi di computer
  - Desktop, server, embedded
- Input/output include
  - Dispositivi di interfaccia utente
    - Monitor, tastiera, mouse
- Dispositivi di archiviazione
  - Hard disk, CD/DVD, flash drive
- Adattatori di rete
  - Per comunicare con altri computer

Dentro la scatola

Dentro il processore (CPU)

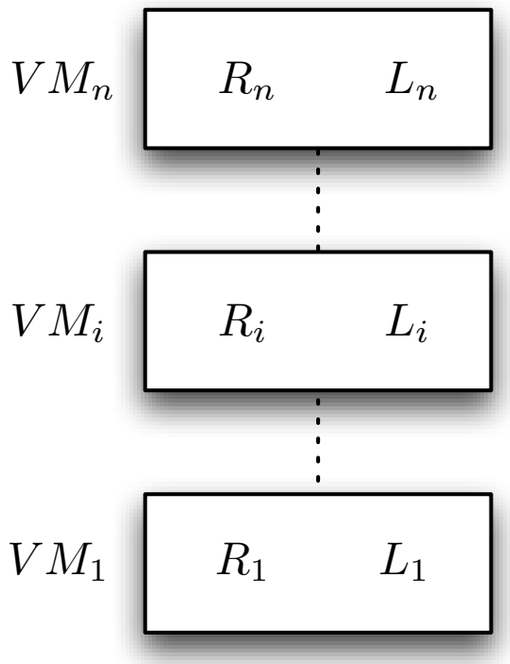
Apple A5

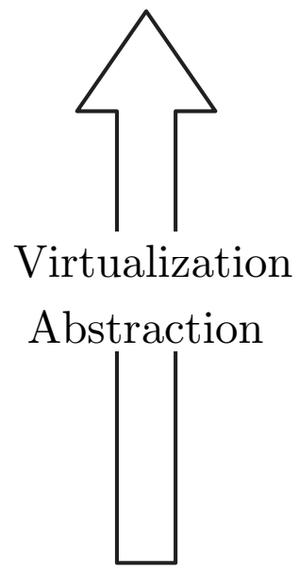


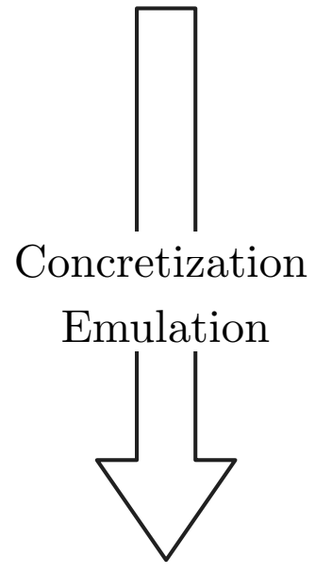
Livelli di astrazione

Astrazione:

- Utilizzata per gestire la complessità
- tipicamente distinta in livelli (*VMi*)
- ogni livello possiede il suo linguaggio L_i e le sue strutture dati R_i
- i livelli inferiori sono implementati in hardware
- i livelli superiori sono implementati in software







Livelli tipici:

5. Applicazioni
4. Assembler
3. Linguaggio Macchina (ISA)

Software

Hardware

2. Microarchitettura (firmware)
1. Logica digitale

Livello di logica digitale

- Linee e porte logiche (*gate*) realizzati tramite transistor
- Le porte logiche elaborano segnali binari (0/1)
- Le linee trasportano segnali binari (0/1)
- Tramite questi segnali è possibile rappresentare qualsiasi tipo di informazioni
- Interconnettendo porte logiche e linee è possibile realizzare funzioni complesse (es. moltiplicazioni di interi)
- Possibile realizzare elementi di memoria

Livello di microarchitettura

- Costruito sopra il livello della logica digitale, si occupa di interpretare ed eseguire le istruzioni del livello ISA

Livello del linguaggio macchina

- È il livello di macchina che appare al programmatore di sistema
- Comprende un insieme di *istruzioni* che di solito sono *diverse per ogni processore*
- La sintassi è adatta ad essere *interpretata facilmente* dal livello sottostante
- Le istruzioni sono *stringhe di bit* con *formato ben determinato* per permettere la facile individuazione di codici e operandi delle istruzioni
- I codici operativi (*opcode*) individuano l'operazione elementare che l'istruzione dovrà eseguire
- gli *operandi* si riferiscono alle locazioni che contengono i dati su cui eseguire l'operazione, e le locazioni dove memorizzare i risultati

Livello assembler

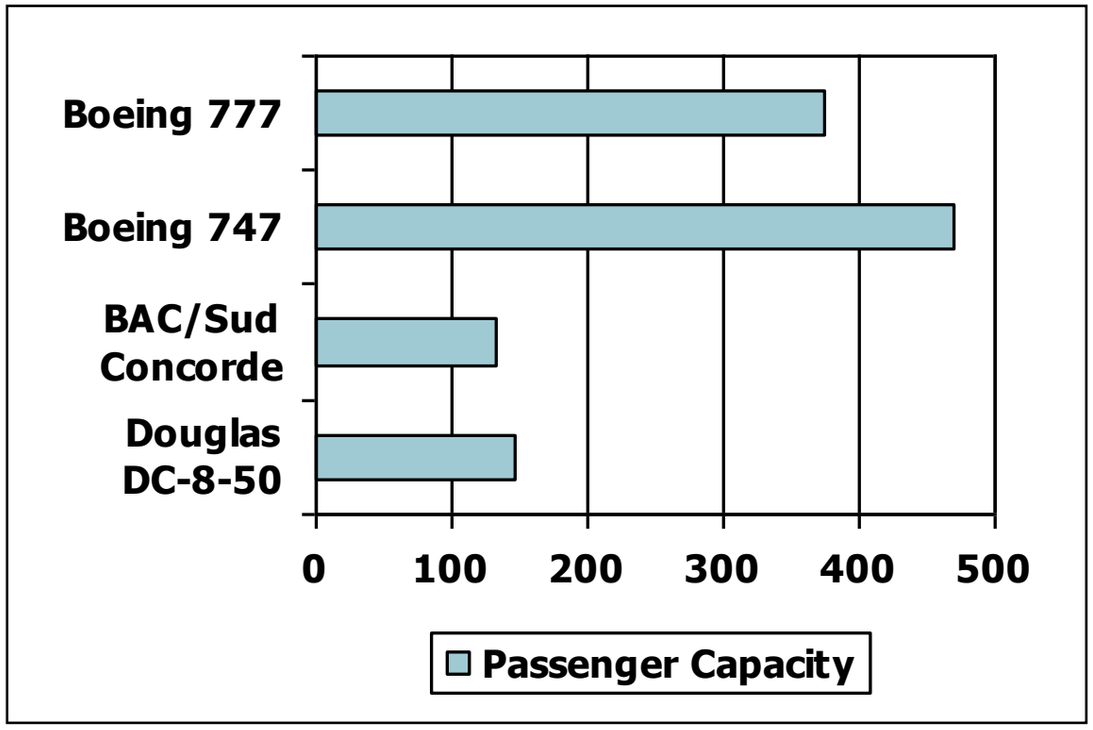
- Offre al programmatore di sistema una vista più "umana" del livello macchina:
  - istruzioni espresse con *stringhe di caratteri mnemoniche* invece di stringhe binarie
  - traduzione realizzata dall'*assemblatore* (uno speciale compilatore) rispetto al livello del linguaggio macchina
- L'*assemblatore* è stato uno dei primi software di sistema realizzato per facilitare la programmazione dei calcolatori

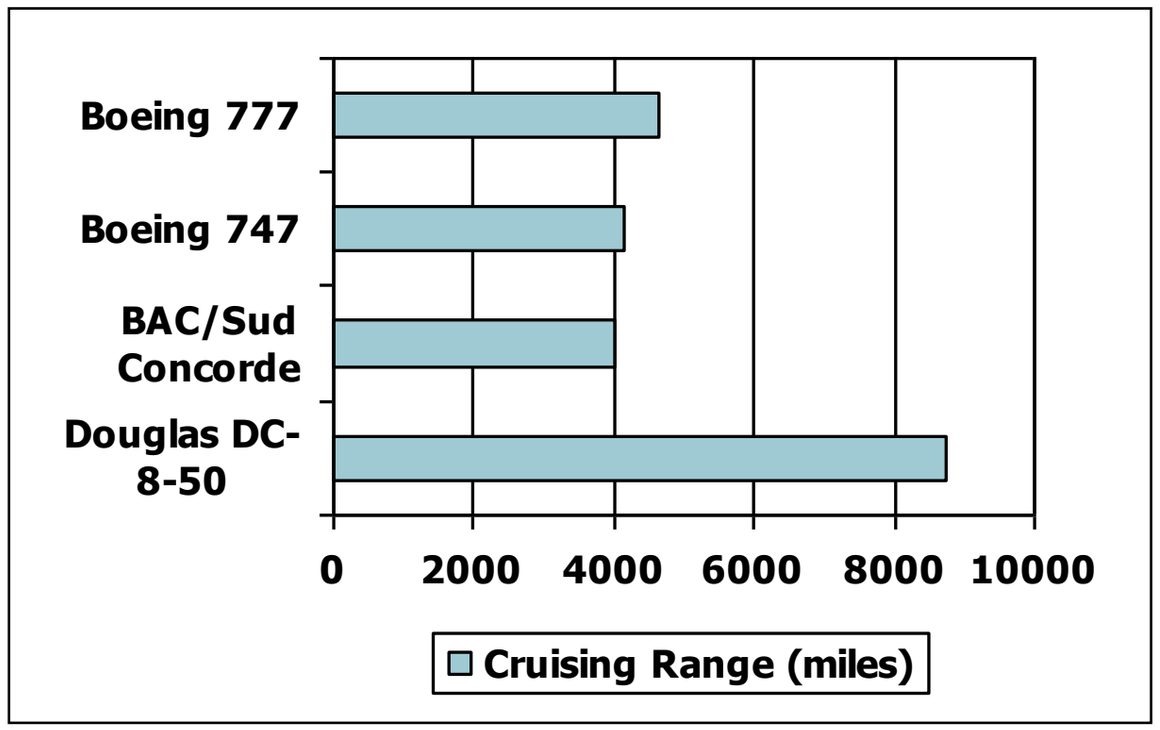
Livello delle applicazioni

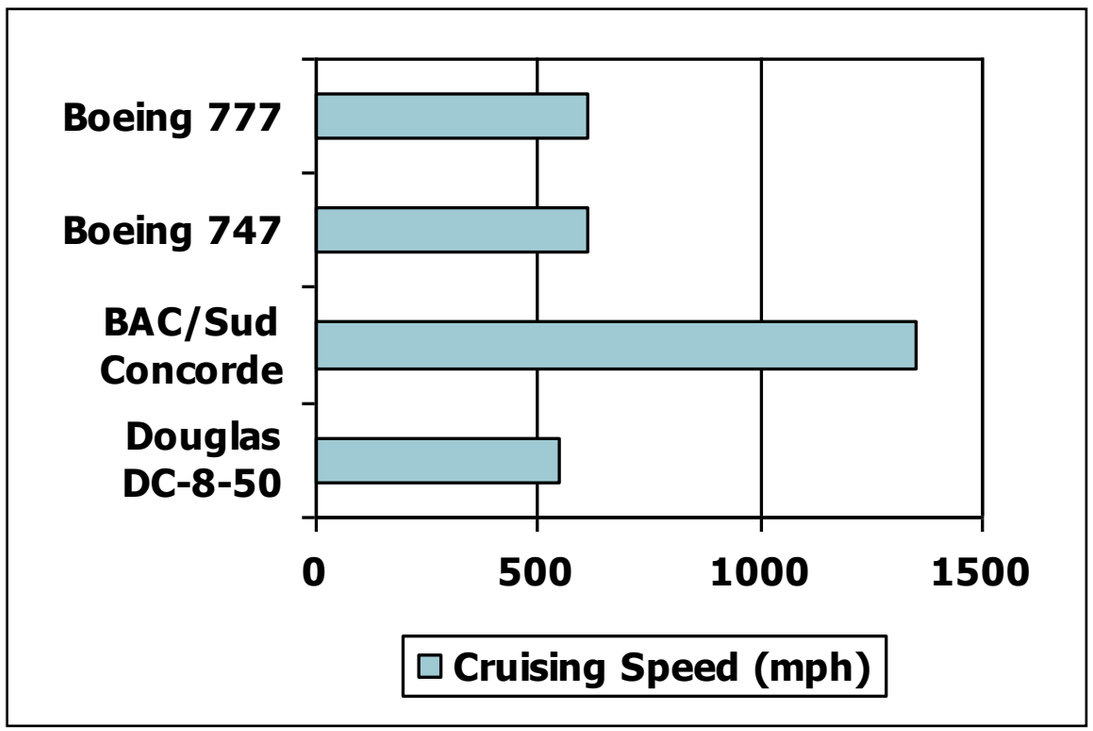
- Linguaggi ad alto livello: C, C++, Java, Python
- Permette al programmatore una maggiore astrazione rispetto ai livelli sottostanti della macchina
- Ha di solito bisogno di un traduttore (compilatore)
- E' il livello solitamente usato per produrre software
- Tale livello permette di realizzare la portabilità tra processori con diversa ISA
  - Basta che esista il compilatore/interprete implementato per la nuova ISA

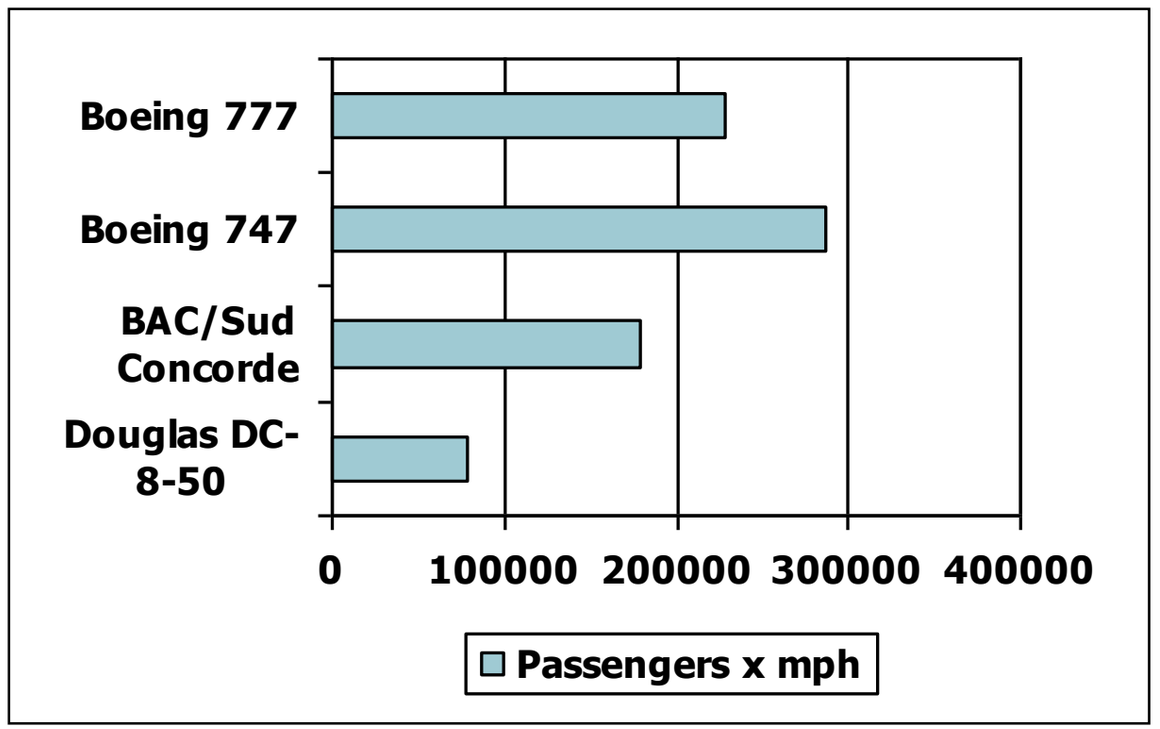
Definire le prestazioni

Quale aeroplano ha le migliori prestazioni?









Tempo di risposta e Throughput

- Tempo di risposta
  - Quanto si impiega per eseguire un task
- Throughput (banda)
  - Lavoro totale svolto per unità di tempo
    - Per esempio, task/transazioni/... all'ora
- Come cambiano il tempo di risposta e il throughput quando...
  - Si sostituisce il processore con una versione più veloce?
  - Si aggiungono più processori?
- Ci concentreremo sul tempo di risposta...

Performance relativa

- Si definisce Performance = 1/Tempo di esecuzione
- "X è *n* volte più veloce di Y"

$$\text{Performance}_X/\text{Performance}_Y = \text{Tempo di esecuzione}_Y/\text{Tempo di esecuzione}_X = n$$

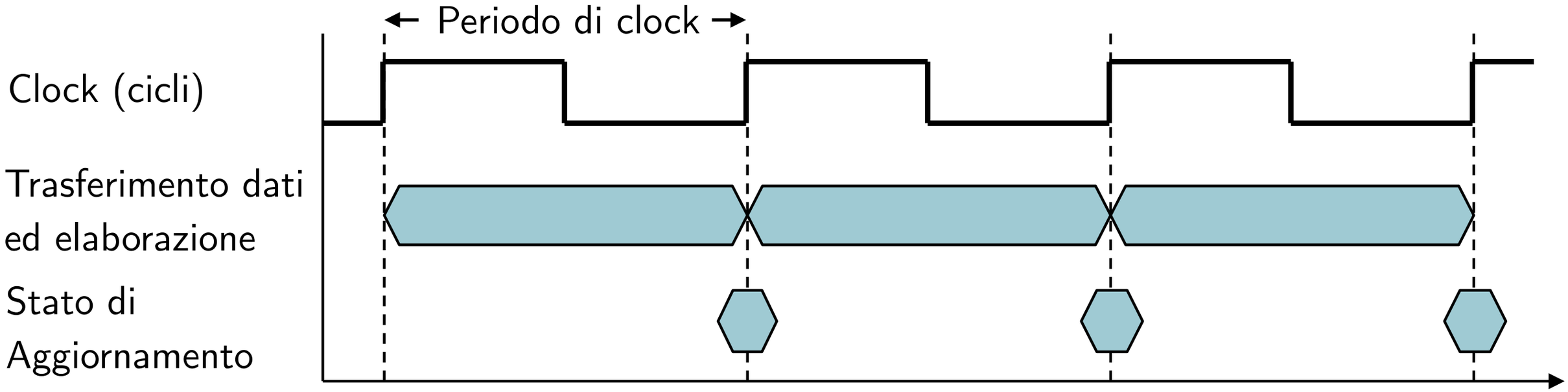
- Esempio: tempo impiegato a eseguire un programma
  - 10 secondi su A, 15 secondi su B
- Tempo di esecuzioneB / Tempo di esecuzioneA = 15/10 = 1.5
- Quindi A è 1.5 più veloce di B

Misurare il tempo di esecuzione

- Tempo trascorso
  - Tempo di risposta totale, tutto incluso
    - Elaborazione, I/O, overhead del SO, tempo di attesa
- Tempo di CPU
  - Tempo speso nell'esecuzione di un job
  - Comprende tempo di CPU utente e tempo di CPU di sistema
  - Programmi diversi subiscono diversamente le prestazioni di sistema e di CPU

Clock della CPU

- Le operazioni dell'hardware digitale sono governate da un clock a frequenza costante



- Periodo di clock: durata di un ciclo di clock
  - Per esempio, 250 ps = 0.25 ns = 250 x 10^{-12} s
- Frequenza di clock: cicli al secondo
  - Per esempio, 4.0 GHz = 4000 MHz = 4.0 x 10^{9} Hz

Tempo di CPU

$$\text{Tempo di CPU} = \text{Cicli di clock della CPU} \times \text{Periodo di clock} =$$

$$= \frac{\text{Cicli di clock della CPU}}{\text{Frequenza di clock}}$$

- Prestazioni migliorate tramite:
  - Riduzione del numero di cicli di clock
  - Aumento della frequenza di clock
  - Il progettista hardware deve spesso bilanciare frequenza di clock e numero di cicli di clock

Esempio Tempo di CPU

- Computer A: 2 GHz, 10 s tempo di CPU
- Progettare il computer B:
  - riducendo il tempo di CPU a 6 s
  - si può avere un clock più veloce, ma incorrendo in 1.2 x cicli di clock
- Quanto veloce deve essere il clock del computer B?

$$\text{Frequenza di clock}_B = \frac{\text{Cicli di clock}_B}{\text{Tempo di CPU}_B} = \frac{1.2 \text{ x Cicli di clock}_A}{6\text{s}}$$

$$\text{Cicli di clock}_A = \text{Tempo di CPU}_A \text{ x Frequenza di clock}_A$$
$$= 10 \text{ s x } 2 \text{ GHZ} = 20 \text{ x } 10^9$$

$$\text{Frequenza di clock}_B = \frac{1.2 \text{ x } 20 \text{ x } 10^9}{6 \text{ s}} = \frac{24 \text{ x } 10^9}{6\text{s}} = 4 \text{ GHz}$$

Instruction Count e CPI

$$\text{Cicli di clock} = \text{Instruction Count} \times \text{Cicli Per Istruzione (CPI)}$$

$$\text{Tempo di CPU} = \text{Instruction Count} \times \text{CPI} \times \text{Periodo di clock}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Frequenza di clock}}$$

- Instruction Count di un programma:
  - Determinato da programma, ISA e compilatore
- Cicli medi per istruzione
  - Determinati dall'hardware della CPU
  - Se istruzioni differenti hanno CPI differenti
    - Il CPI medio dipende dal mix delle istruzioni

Esempio CPI

- Computer A: periodo di clock = 250 ps, CPI = 2.0
- Computer B: periodo di clock = 500 ps, CPI = 1.2
- Stessa ISA
- Qual è il più veloce, e di quanto?

$$\text{Tempo di CPU}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Periodo di clock}_A$$
$$= I \times 2.0 \times 250 \text{ ps} = I \times 500 \text{ ps}$$
$$\text{Tempo di CPU}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Periodo di clock}_B$$
$$= I \times 1.2 \times 500 \text{ ps} = I \times 600 \text{ ps}$$
$$\frac{\text{Tempo di CPU}_B}{\text{Tempo di CPU}_A} = \frac{I \times 600 \text{ ps}}{I \times 500 \text{ ps}} = 1.2$$

CPI nei dettagli

- Se classi di istruzioni differenti impiegano un numero differente di cicli di clock

$$\text{Cicli di clock} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- CPI medio pesato

$$\text{CPI} = \frac{\text{Cicli di clock}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

Esempio CPI

- Sequenze di codice compilato alternative usano le istruzioni nelle classi A, B e C

Classe	A	B	C
CPI per classe	1	2	3
IC sequenza 1	2	1	2
IC sequenza 2	4	1	1

- Sequenza 1: IC = 5
  - Cicli di clock

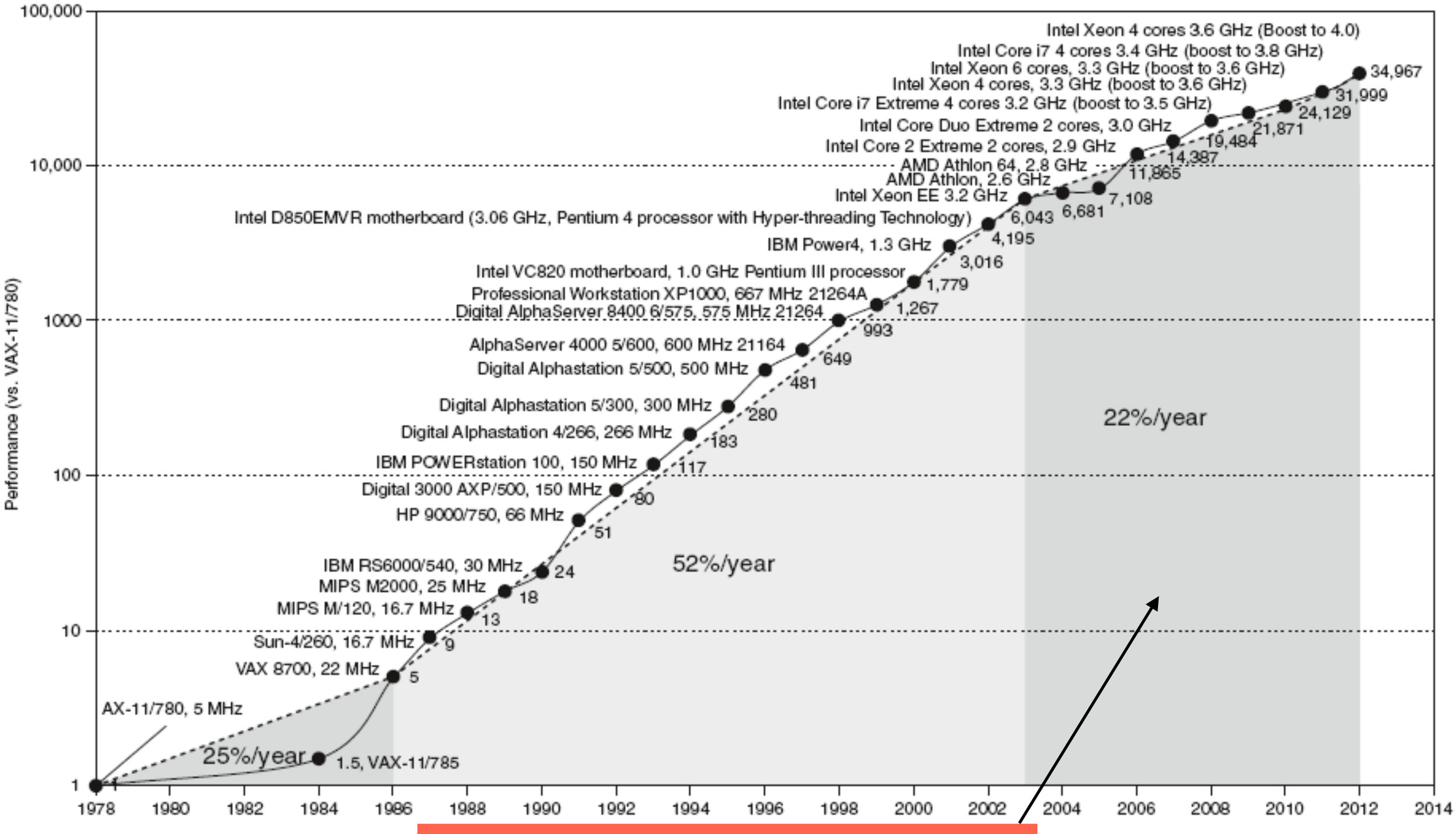
    = 2 x 1 + 1 x 2 + 2 x 3

    = 10
  - CPI medio = 10/5 = 2.0
- Sequenza 2: IC = 6
  - Cicli di clock

    = 4 x 1 + 1 x 2 + 1 x 3

    = 9
  - CPI medio = 9/6 = 1.5

Prestazioni uniprocessori



Vincolati da potenza, parallelismo a livello di istruzioni, latenza della memoria

Multiprocessori

- Microprocessori multicore
  - Più di un processore per chip
- Richiedono programmazione parallela esplicita
  - A differenza del parallelismo a livello di istruzioni
    - L'hardware esegue molteplici istruzioni alla volta
    - Nascosto al programmatore
- Difficile
  - programmare per le prestazioni
  - bilanciare il carico
  - ottimizzare le comunicazioni e la sincronizzazione

Sommario sulle prestazioni

$$\text{Tempo CPU} = \frac{\text{Istruzioni}}{\text{Programma}} \times \frac{\text{Cicli di clock}}{\text{Istruzione}} \times \frac{\text{Secondi}}{\text{Ciclo di clock}}$$

- Le prestazioni dipendono da:
  - Algoritmo: determina IC, possibilmente CPI
  - Linguaggio di programmazione: determina IC, CPI
  - Compilatore: determina IC, CPI
  - ISA: determina IC, CPI e periodo di clock

Capitolo 2

Instruction Set

- Il repertorio di istruzioni di un computer
- Computer diversi hanno instruction set diversi
  - Ma con molti aspetti in comune
- I primi computer avevano instruction set molto semplici
  - Per semplificare l'implementazione
- Anche molti computer moderni hanno instruction set semplici

Instruction Set del MIPS

- Usato come esempio per tutto il corso
- Progettato all'Università di Stanford e commercializzato da MIPS Technologies (www.mips.com)
- Grande fetta del mercato dei processori embedded
  - Applicazioni all'elettronica di consumo, archiviazione/rete, dispostivi, videocamere, stampanti, ...
- Rappresentativo di molte ISA moderne
  - Vedi *Scheda tecnica riassuntiva del MIPS* nel libro

Operazioni Aritmetiche

- Addizione e sottrazione, tre operandi
  - Due sorgenti e una destinazione

```
add a, b, c      # a diventa b + c
```

- Tutte le operazioni aritmetiche hanno questa forma
- **Principio di Progettazione 1**: *"La semplicità favorisce la regolarità."*
  - La regolarità rende l'implementazione più semplice
  - La semplicità permette maggiori prestazioni a costi inferiori

Esempio

- Codice C:

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

- Codice MIPS compilato:

```
add t0, g, h     # temp t0 = g + h
add t1, i, j     # temp t1 = i + j
sub f, t0, t1    # f = t0 - t1
```

Operandi su registro

- Le istruzioni aritmetiche usano operandi **registro**
- Il MIPS usa un **register file** di 32 registri a 32 bit
  - Usati per dati acceduti frequentemente
  - Numerati da 0 a 31
  - Un dato su 32 bit è detto "*word*"
- Nomi assembler:
  - `$t0`, `$t1`, ..., `$t9` per valori temporanei
  - `$s0`, `$s1`, ..., `$s7` per valori salvati
- **Principio di Progettazione 2**: *"Più piccolo è più veloce."*
  - Si pensi alla memoria: milioni di locazioni

Esempio

- Codice C:

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

  - `f`, ..., `j` in `$s0`, ..., `$s4`

- Codice MIPS compilato:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Operandi in memoria

- La memoria principale è usata per dati composti
  - Array, strutture, dati dinamici
- Per effettuare operazioni aritmetiche
  - Caricare i valori dalla memoria nei registri
  - Copiare il risultato dal registro alla memoria
- La memoria è indirizzata al byte
  - Ogni indirizzo identifica un byte su 8 bit
- Le parole sono allineate in memoria
  - Un indirizzo deve essere un multiplo di 4
- MIPS è *Big Endian*
  - Il byte più significativo è nell'indirizzo più basso di una parola
  - *Little Endian*: il byte meno significativo è nell'indirizzo più basso di una parola

Esempio 1

- Codice C:

```
g = h + A[8]
```

  - `g` in `$s1`, `h` in `$s2`, indirizzo base di `A` in `$s3`
- Codice MIPS compilato:
  - L'indice 8 richiede un offset di 32 byte
    - 4 byte per parola

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset — registro base

Esempio 2

- Codice C:

```
A[12] = h + A[8]
```

  - `h` in `$s2`, indirizzo base di `A` in `$s3`
- Codice MIPS compilato:
  - L'indice 8 richiede un offset di 32 byte

```
lw  $t0, 32($s3)   # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)   # store word
```

Registri vs. Memoria

- I registri sono acceduti più velocemente della memoria
- Operare su dati in memoria richiede delle *load* e delle *store*
  - Più istruzioni da eseguire
- Il compilatore deve usare i registri per le variabili il più possibile
  - Rivolgersi alla memoria solo per variabili usate meno frequentemente
  - L'ottimizzazione dei registri è importante!

Operandi immediati

- Dati costanti sono specificati nell'istruzione stessa

```
addi $s3, $s3, 4
```

- Nessuna istruzione di sottrazione immediata
  - Basta usare una costante negativa

```
addi $s2, $s2, -1
```

- **Principio di Progettazione 3**: *"Rendere il caso comune veloce"*
  - Piccole costanti sono comuni
  - Gli operandi immediati evita un'istruzione di *load*

La costante Zero

- Il registro MIPS 0 (`$zero`) è la constante 0
  - Non può essere sovrascritto
- Utile per molte operazioni comune
  - Per esempio, spostamenti tra i registri

```
add $t2, $t1, $zero
```

Interi binari senza segno

- Dato un numero su n bit

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

- Intervallo: da 0 a $2^n - 1$
- Esempio

  $\texttt{0000 0000 0000 0000 0000 0000 0000 1011}_2$

  $= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Usando 32 bit
  - da 0 a $4'294'967'295$

Interi con segno in complemento a 2

- Dato un numero su n bit

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

- Intervallo: da -2^{n-1} a $2^{n-1} - 1$
- Esempio

  $\texttt{1111 1111 1111 1111 1111 1111 1111 1100}_2$

  $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

  $= -2'147'483'648 + 2'147'483'644 = -4_{10}$

- Usando 32 bit
  - da $-2'147'483'648$ a $2'147'483'647$

Interi con segno in complemento a 2

- Il bit 31 è il bit del segno
  - 1 per numeri negativi
  - 0 per numeri non-negativi
- $-(-2^{n-1})$ non può essere rappresentato
- I numeri non-negativi hanno le stesse rappresentazioni in binario senza segno e in complemento a 2
- Alcuni numeri particolari:
  - 0: `0000 0000 ... 0000`
  - –1: `1111 1111 ... 1111`
  - Più negativo: `1000 0000 ... 0000`
  - Più positivo: `0111 1111 ... 1111`

Negazione con segno

- Complemento bit a bit a sommare 1
  - Complemento bit a bit significa $0 \rightarrow 1$, $1 \rightarrow 0$

$$x + \bar{x} = 111111....111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Esempio: negare +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$

    $= 1111\ 1111\ \ldots\ 1110_2$

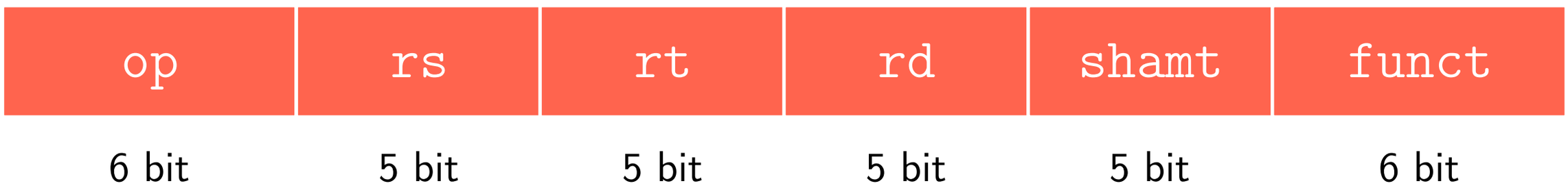
Estensione di segno

- Rappresentare un numero usando più bit
  - Mantenendo il valore numerico
  - Nell'instruction set del MIPS
    - `addi`: estendere il valore immediato
    - `lb, lh` : estendere il byte/halfword caricato
    - `beq, bne` : estendere lo spiazzamento
- Replicate il bit di segno a sinistra
  - per i valori senza segno: estendere con 0
- Esempi: da 8 bit a 16 bit
  - +2: 0000 0010 → 0000 0000 0000 0010
  - –2: 1111 1110 → 1111 1111 1111 1110

Rappresentare le istruzioni

- Le istruzioni sono codificate in binario
  - Sono chiamate codice macchina
- Istruzioni MIPS
  - Codificate come istruzioni word su 32 bit
  - Ridotto numero di formati per codificare il codice dell'operazione (opcode), numeri di registro, ...
  - Regolarità!
- Numeri di registro
  - `$t0` – `$t7` sono i registri 8 – 15
  - `$t8` – `$t9` sono i registri 24 – 25
  - `$s0` – `$s7` sono i registri 16 – 23

Istruzioni MIPS in formato R



- Campi dell'istruzione
- `op`: codice dell'operazione (opcode)
- `rs`: numero di registro della prima sorgente
- `rt`: numero di registro della seconda sorgente
- `rd`: numero di registro della destinazione
- `shamt`: shift amount (`00000` per ora)
- `funct`: codice della funzione (estende l'opcode)

Esempi formato R

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

`add $t0, $s1, $s2`

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Esadecimale

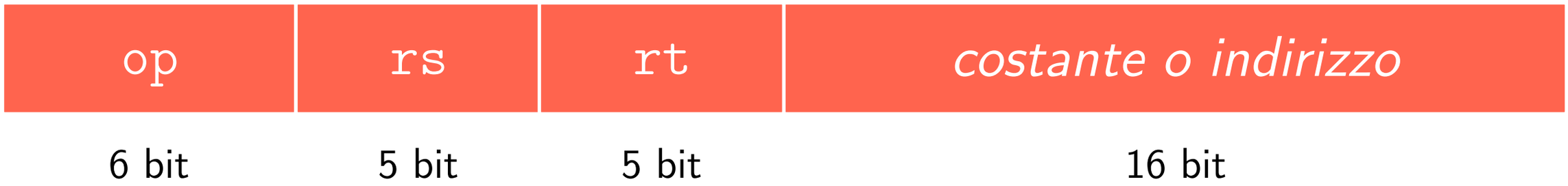
- Base 16
  - Rappresentazione compatta di stringhe di bit
  - 4 bit per cifra esadecimale

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Esempio: eca8 6420

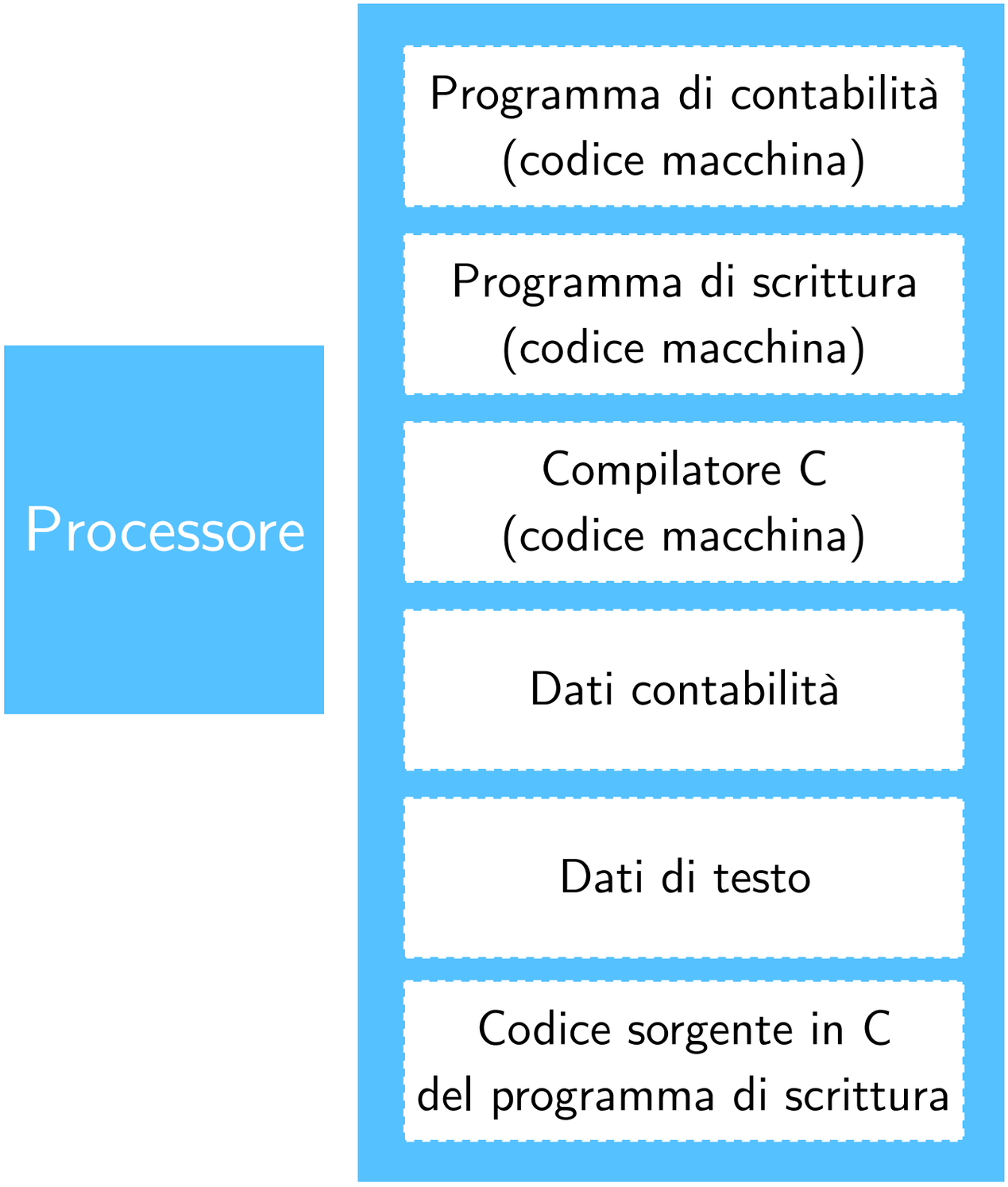
```
1110 1100 1010 1000 0110 0100 0010 0000
```

Istruzioni MIPS in formato I



- Istruzioni load/store e aritmetiche immediate
  - `rt`: numero di registro della sorgente o destinazione
  - *costante*: da -2^{15} a $2^{15}-1$
  - *indirizzo*: offset aggiunto all'indirizzo base in `rs`
- **Principio di Progettazione 4**: *"un buon progetto richiede dei buoni compromessi"*
  - Formati differenti complicano la decodifica, ma permettono istruzioni su 32 bit uniformi
  - Mantenere i formati il più simile possibile

Stored Program Computer



- Le istruzioni sono rappresentate in binario, proprio come i dati
- I dati e le istruzioni sono immagazzinati in memoria
- I programmi posso operare su programmi
  - Compilatori, linker, ...
- La compatibilità binaria permette a programmi compilati di funzionare su computer diversi
  - ISA standardizzate

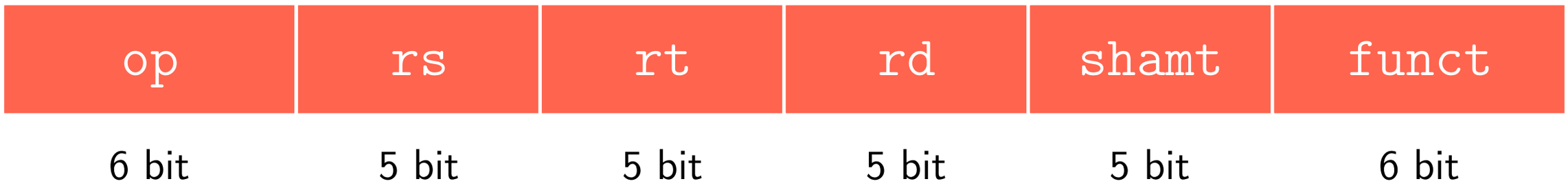
Operazioni logiche

- Istruzioni per la manipolazione bit a bit

Operazione	C	Java	MIPS
Shift a sinistra	<<	<<	`sll`
Shift a destra	>>	>>>	`srl`
AND bit a bit	&	&	`and, andi`
OR bit a bit	|	|	`or, ori`
NOT bit a bit	~	~	`nor`

- Utili per estrarre e inserire gruppi di bit in una word

Operazioni di shift



- `shamt` (shift amount): di quante posizioni scorrere
- Scorrimento logico a sinistra
  - scorrimento a sinistra riempiendo con bit a 0
  - `sll` (shift left logical) di *i* bit moltiplica per 2^i
- Scorrimento logico a destra
  - scorrimento a destra riempiendo con bit a 0
  - `srl` (shift right logical) di *i* bit divide per 2^i (solo unsigned)

Operazione AND

- Utile per mascherare bit in una word
  - Seleziona alcuni bit e pone gli altri a 0

```
and $t0, $t1, $t2
```

```
$t2  0000 0000 0000 0000 0000 0001 1101 1100 0000
$t1  0000 0000 0000 0000 0000 0011 1100 0000 0000
$t0  0000 0000 0000 0000 0000 0001 1100 0000 0000
```

Operazione OR

- Utile per includere bit in una word
  - Pone alcuni bit a 1 e lascia gli altri inalterati

```
or $t0, $t1, $t2
```

```
$t2  0000 0000 0000 0000 0000 0000 1101 1100 0000
$t1  0000 0000 0000 0000 0000 0011 1100 0000 0000
$t0  0000 0000 0000 0000 0000 0011 1101 1100 0000
```

Operazione NOT

- Utile per complementare bit a bit in una parola
  - Cambia 0 in 1 e 1 in 0
- Il MIPS ha l'istruzione NOR a 3 operandi
  - `a NOR b == NOT (a OR b)`

```
nor $t0, $t1, $zero
```

```
$t1  0000 0000 0000 0000 0011 1100 0000 0000
$t0  1111 1111 1111 1111 1100 0011 1111 1111
```

Operazioni condizionali

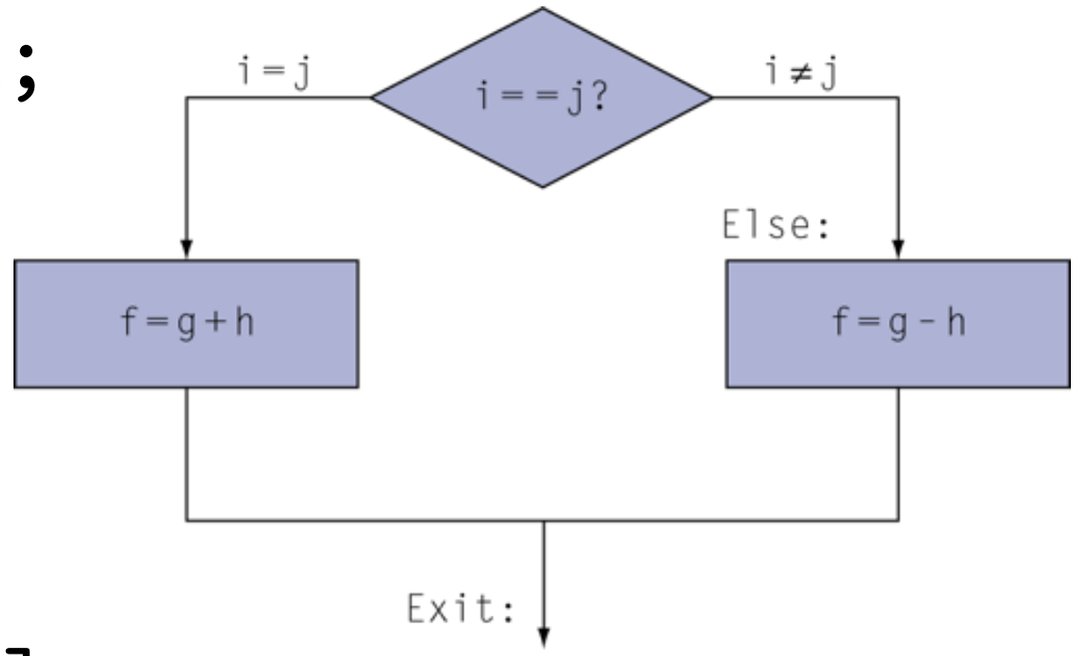
- Salto (branch) a un'istruzione etichettata se la condizione è vera
  - Altrimenti continuare in sequenza
- `beq rs, rt, L1`
  - se (`rs == rt`) salta all'istruzione con etichetta `L1`
- `bne rs, rt, L1`
  - se (`rs != rt`) salta all'istruzione con etichetta `L1`
- `j L1`
  - salta incondizionatamente all'istruzione con etichetta `L1`

Compilare comandi condizionali

- Codice C:

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

  - f, g, ... in $s0, $s1, ...



- Codice MIPS compilato:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j Exit
Else:   sub $s0, $s1, $s2
Exit:   ...
```

L'assemblatore calcola gli indirizzi

Compilare comandi di ciclo

- Codice C:

```
while (save[i] == k)
    i += 1;
```

  - `i` in `$s3`, `k` in `$s5`, indirizzo di `save` in `$s6`

- Codice MIPS compilato:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: ...
```

Altre operazioni condizionali

- Porre il risultato a 1 se la condizione è vera
  - Altrimenti porlo a 0
- `slt rd, rs, rt`
  - se (`rs` < `rt`) `rd` = 1; altrimenti `rd` = 0
- `slti rd, rs, constant`
  - se (`rs` < `constant`) `rt` = 1; altrimenti `rt` = 0
- Usato in combinazione con `bne` e `beq`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  # salta a L
```

Progettazione delle istruzioni di salto

- Perchè non `blt`, `bge`, etc.?
- L'hardware per $<$, $\geq$, ... è più lento di quello per $=$, $\neq$
  - Combinato con un salto richiede più lavoro per istruzione, e quindi un clock più lento
  - Tutte le istruzioni sono penalizzate!
- `beq` e `bne` sono i casi più comuni
- Questo è un buon compromesso di progetto

Invocazione di procedure

- Passi necessari:
  1. Disporre i parametri nei registri
  2. Trasferire il controllo alla procedura
  3. Acquisire spazio per la procedura
  4. Eseguire le operazioni della procedura
  5. Disporre il risultato nel registro per l'invocante
  6. Ritornare al punto di invocazione

Uso dei registri

- `$a0 - Sa3`: argomenti (registri 4 – 7)
- `$v0, $v1`: valori di ritorno (registri 2 e 3)
- `$t0 - $t9`: valori temporanei
  - Possono essere sovrascritti dalla procedura chiamata
- `$s0 - $s7`: valori salvati
  - Devono essere salvati/rispristinati dalla procedura chiamata
- `$gp`: *global pointer* per i dati statici (registro 28)
- `$sp`: *stack pointer* (registro 29)
- `$fp`: *frame pointer* (registro 30)
- `$ra`: indirizzo di ritorno (registro 31)

Istruzioni per invocare una procedura

- Invocazione di una procedura: *jump and link*

  `jal EtichettaProcedura`

  - L'indirizzo dell'istruzione successiva è posto in `$ra`
  - Salto all'indirizzo di destinazione
- Ritorno da una procedura: *jump register*

  `jr $ra`

  - Copia `$ra` nel program counter
  - Può anche essere usato per salti calcolati
    - per esempio, nei comandi case/switch

Esempio con procedura foglia

- Codice C:

```
int esempio_foglia(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Argomenti `g`, ..., `j` copiati in `$a0`, ... `$a3`
- Variable locale `f` in `$s0` (quindi dobbiamo salvare `$s0` sullo stack)
- Risultato copiato in `$v0`

Esempio con procedura foglia

- Codice MIPS:

```
esempio_foglia:
```

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

Salvare `$s0` sullo stack

```
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
```

Corpo della procedura

```
    add  $v0, $s0, $zero
```

Risultato

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

Ripristinare `$s0`

```
    jr $ra
```

Ritorno

Esempio con procedura non-foglia

- Procedure che invocano altre procedure
- Per invocazioni annidate, la procedura chiamante deve salvare sullo stack:
  - Il suo indirizzo di ritorno
  - Qualsiasi argomento e variabile temporanea necessaria al termine dell'invocazione
- Ripristinare tutto il necessario dallo stack al termine dell'invocazione

Esempio con procedura non-foglia

- Codice C:

```
int fattoriale(int n)
{
    if (n < 1)
        return 1;
    else
        return n * fattoriale(n - 1);
}
```

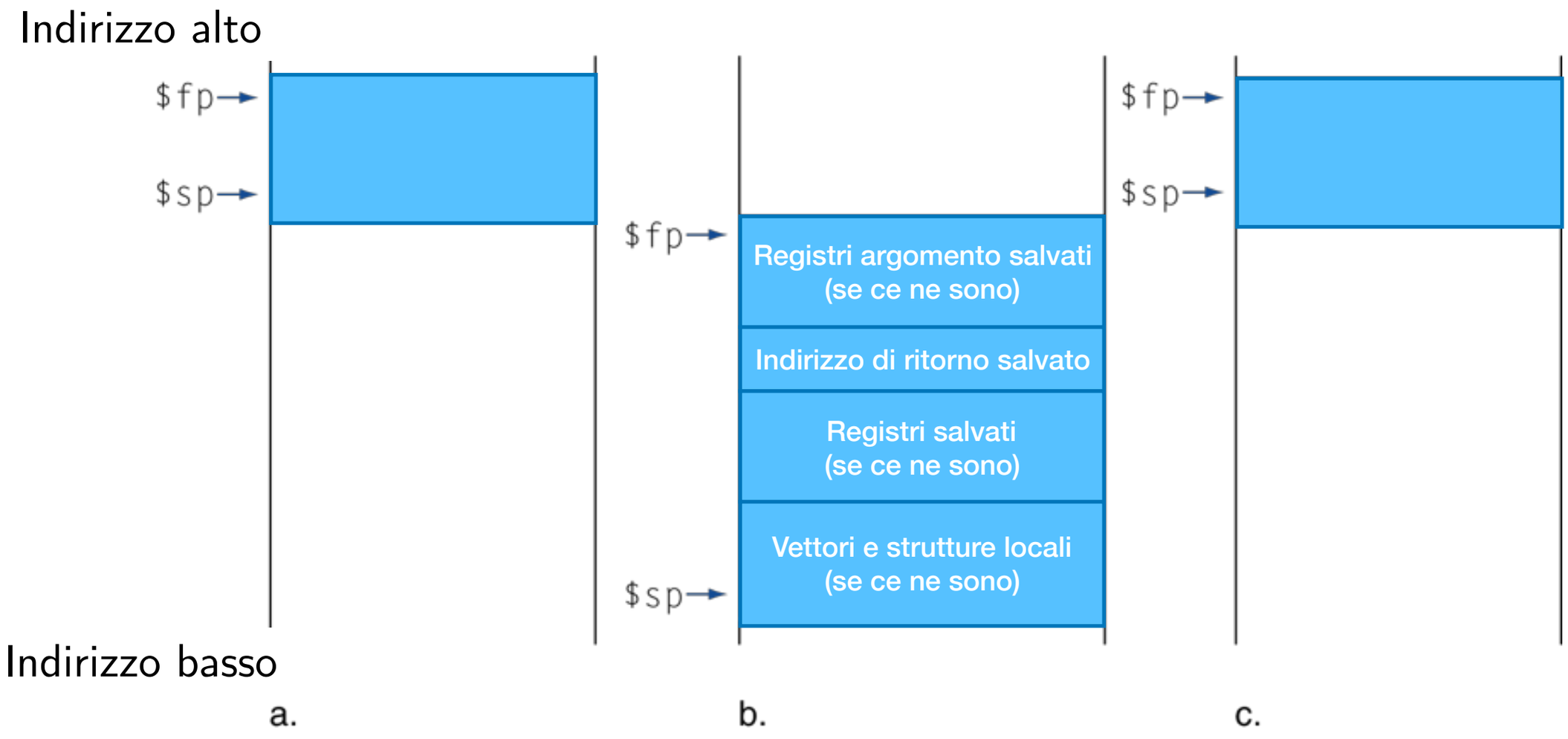
- Argomento `n` copiato in `$a0`
- Risultato copiato in `$v0`

Esempio con procedura non-foglia

- Codice MIPS:

```
fattoriale:
    addi $sp, $sp, -8       # prerarare lo stack per 2 elementi
    sw   $ra, 4($sp)        # salvare l'indirizzo di ritorno
    sw   $a0, 0($sp)        # salvare l'argomento
    slti $t0, $a0, 1        # test per n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # se è vero, il risultato è 1
    addi $sp, $sp, 8        #   estrarre 2 elementi dallo stack
    jr   $ra                #   e ritornare
L1: addi $a0, $a0, -1       # altrimenti decrementare n
    jal  fattoriale         #   e invocare ricorsivamente
    lw   $a0, 0($sp)        # ripristinare n originale
    lw   $ra, 4($sp)        #   e l'indirizzo di ritorno
    addi $sp, $sp, 8        # estrarre 2 elementi dallo stack
    mul  $v0, $a0, $v0      # moltiplicare per il risultato
    jr   $ra                # e ritornare
```

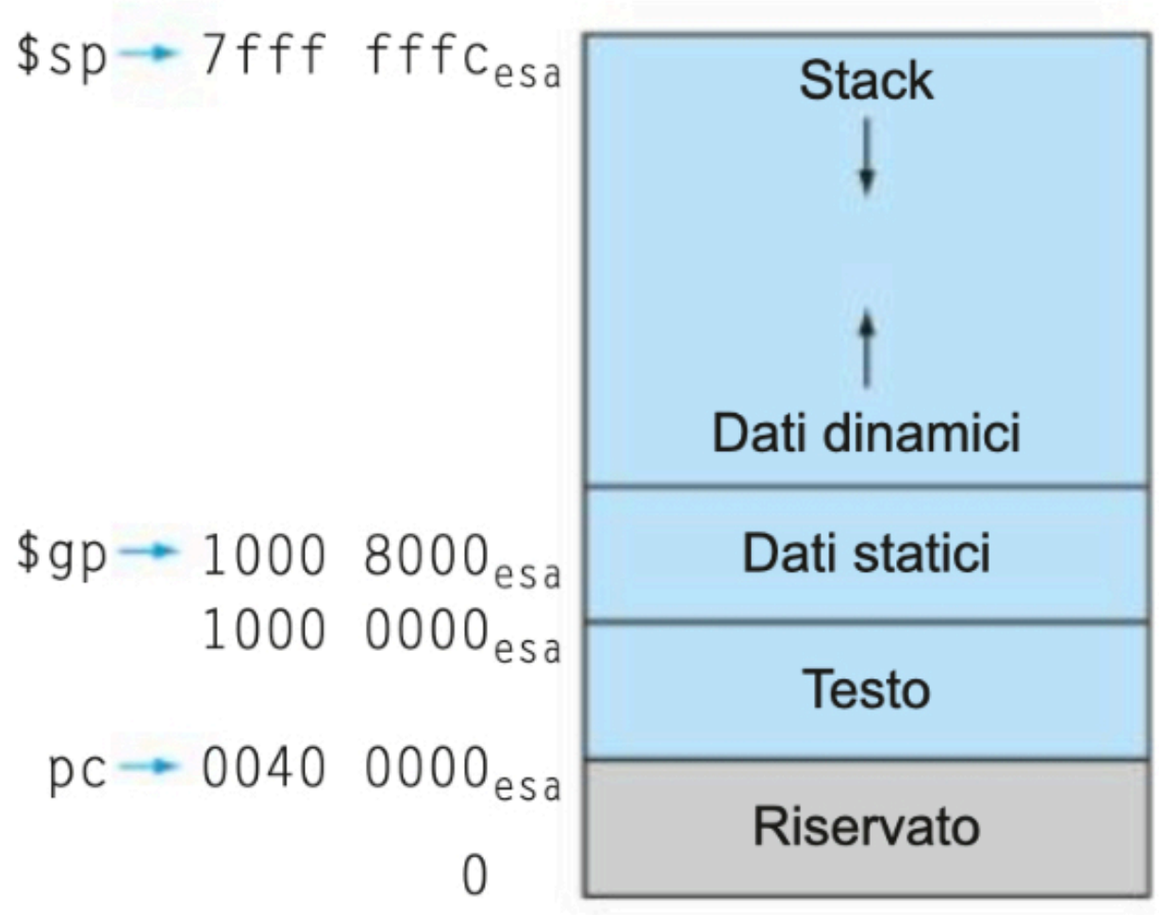
Dati locali sullo stack



- Dati locali allocati dal chiamato
  - per esempio, variabili C automatiche
- Frame della procedura (record di attivazione)
  - Usato da alcuni compilatori per gestire lo spazio dello stack

Layout della memoria

- Testo: codice del programma
- Dati statici: variabili globali
  - Per esempio, variabili statiche in C, array costanti e stringhe
  - $gp inizializzato all'indirizzo che permette ±offset in questo segmento
- Dati dinamici: heap
  - Per esempio, `malloc` in C, `new` in C++/Java
  - Stack: spazio automatico



Dati di tipo carattere

- Insiemi di caratteri codificati su byte
  - ASCII: 128 caratteri
    - 95 grafici, 33 di controllo
  - Latin-1: 256 caratteri
    - ASCII + 96 caratteri grafici aggiuntivi
- Unicode: insieme di caratteri su 32 bit
  - Usati in Java, C++ (wide characters), ...
  - La maggior parte degli alfabeti del mondo + simboli
  - UTF-8, UTF-16: codifiche a lunghezza variabile

Operazioni su byte/halfword

- Si potrebbero usare operazioni bit a bit
- MIPS load/store su byte/halfword
  - Elaborazione di stringhe è un caso comune

    `lb rt, offset(rs)` `lh rt, offset(rs)`
  - Estensione di segno su 32 bit in rt

    `lbu rt, offset(rs)` `lhu rt, offset(rs)`
  - Estensione di zero su 32 bit in rt

    `sb rt, offset(rs)` `sh rt, offset(rs)`
  - Semplicemente si memorizza il byte/halfword più a destra

Esempio di copia di stringa

- Codice C (banale):
  - Stringa termina con `null`

```
void copia_stringa(char x[], char[] y)
{
        int i;
        i = 0;
        while ((x[i] = y[i]) != '\0')
            i += 1;
}
```

- Indirizzi di `x`, `y` copiati in `$a0`, `$a1`
- `i` copiato in `$s0`

Esempio di copia di stringa

- Codice MIPS:

```
copia_stringa:
    addi $sp, $sp, -4       # prerarare lo stack per 1 elemento
    sw   $s0, 0($sp)        # salvare $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1      # indirizzo di y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # indirizzo di x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # uscire dal ciclo se y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # prossima iterazione del ciclo
L2: lw   $s0, 0($sp)        # ripristinare $s0 salvato
    addi $sp, $sp, 4        # estrarre 1 elemento dallo stack
    jr   $ra                # e ritornare
```

Costanti a 32 bit

- La maggior parte delle costanti sono piccole
  - Il campo `immediate` a 16 bit è sufficiente
- Per le rare costanti a 32 bit

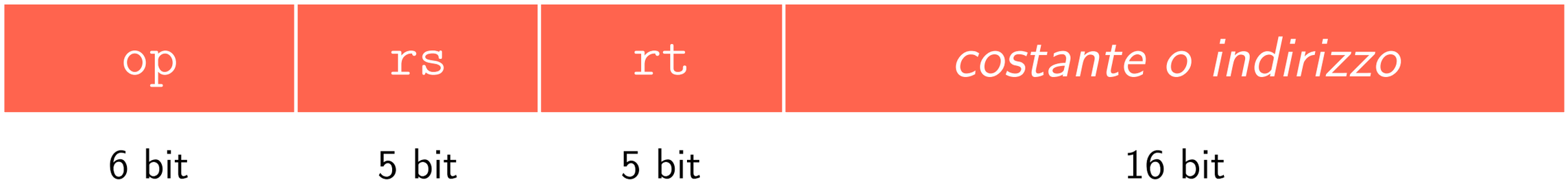
```
lui rt, costant
```

  - Copia la costante a 16 bit nei 16 bit a sinistra di `rt`
  - Resetta i 16 bit a destra di `rt` a 0

```
                    61 0000 0000 0000 0000 0000 0000 0111 1101
lui $s0, 61             0000 0000 0111 1101 0000 0000 0000 0000
                  2034 0000 0000 0000 0000 0000 1001 0000 0000
ori $s0, $s0, 2304  0000 0000 0111 1101 0000 1001 0000 0000
```

Indirizzamento nei salti

- Le istruzioni di salto specificano
  - il codice dell'operazione, due registri, l'indirizzo destinazione
- La maggior parte dei salti indirizzano istruzioni vicini
  - In avanti o all'indietro



- Indirizzamento relativo a PC
  - Indirizzo destinazione = PC + offset x 4
  - PC è già incrementato di 4 prima di questo punto (!!!)

Indirizzamento nei salti incondizionati

- Le destinazioni dei salti incondizionato (`j` e `jal`) potrebbero essere ovunque nel segmento text
  - Si codifica l'indirizzo completo nell'istruzione (formato J)

- Indirizzamento del salto (pseudo)-diretto:
  - Indirizzo destinazione = $PC_{31...28}$: (indirizzo x 4)

Esempio di indirizzamento

- Codice del ciclo da un esempio precedente
  - Si assuma che `Loop` sia alla locazione 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: ...
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Salti lontani

- Se la destinazione del salto è molto lontana per codificarla con un offset di 16 bit, l'assembler riscrive il codice
- Esempio:

```
    beq $s0, $s1, L1
```

↓

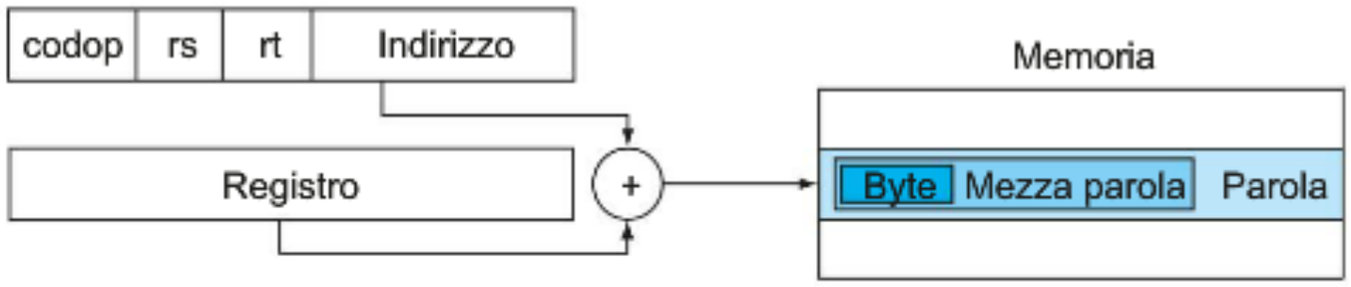
```
    bne $s0, $s1, L2
    j L1
L2:
```

Modalità di indirizzamento

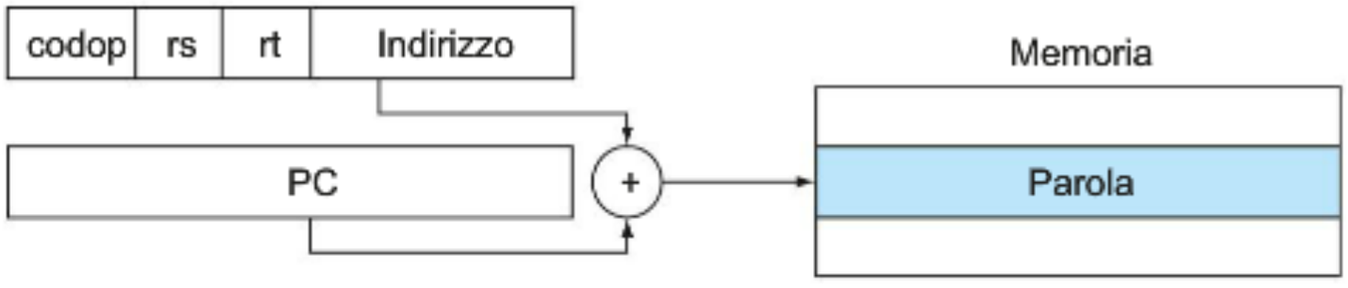
1. Indirizzamento immediato

2. Indirizzamento tramite registro

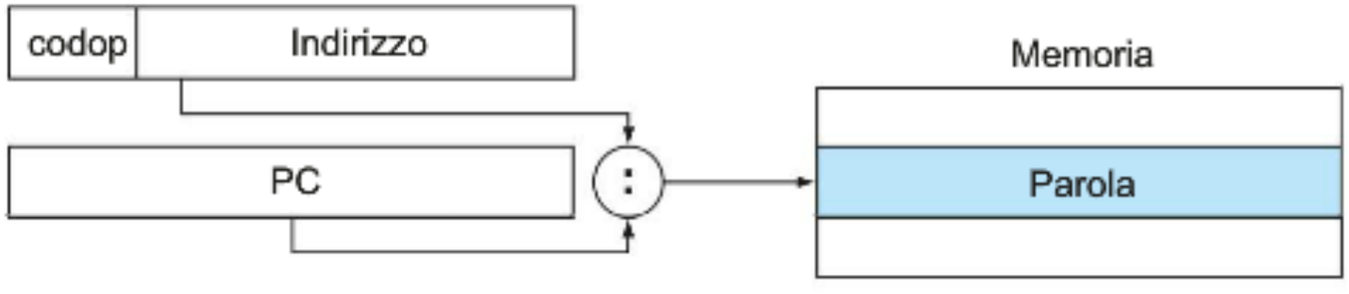
3. Indirizzamento tramite base



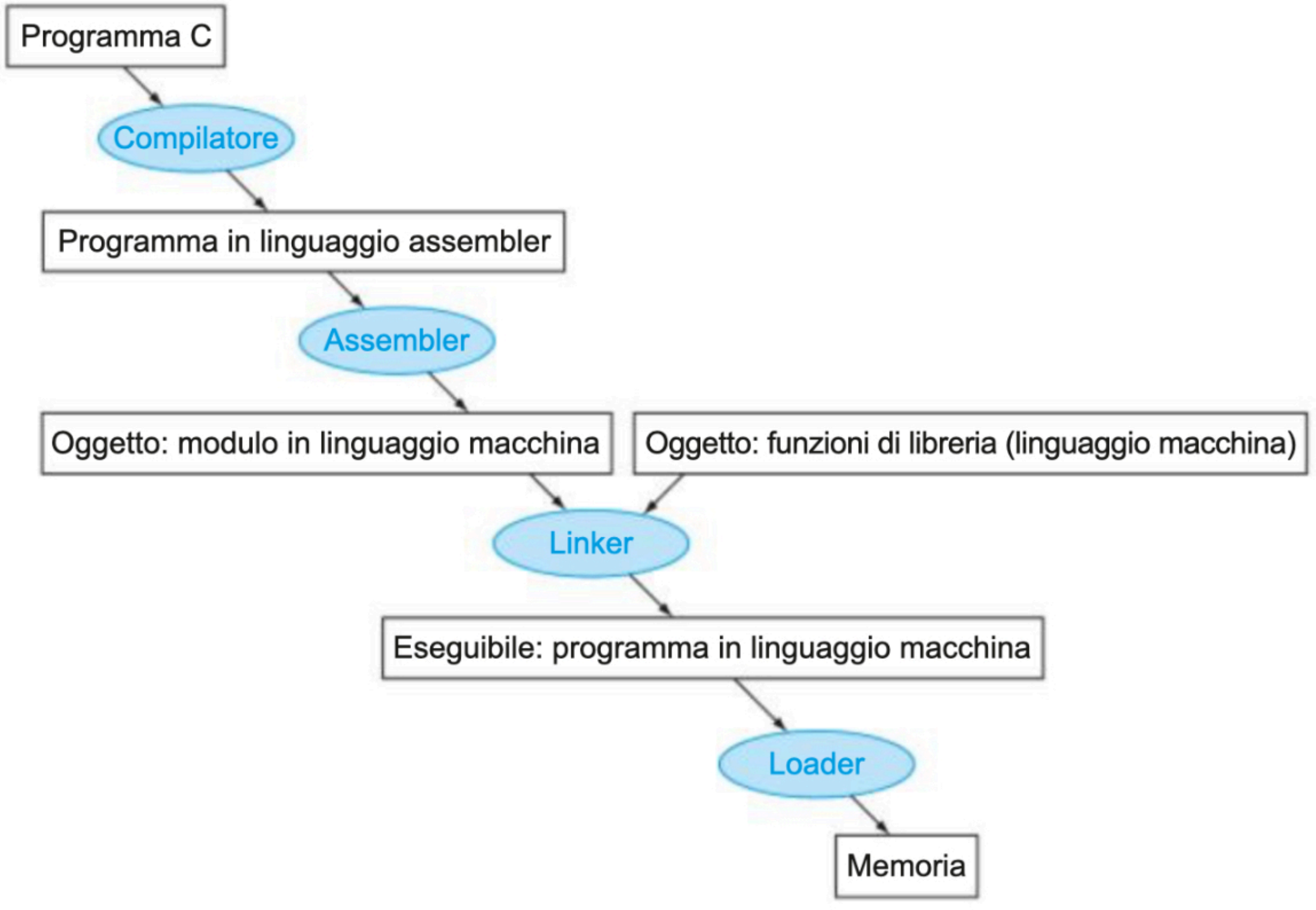
4. Indirizzamento relativo al PC



5. Indirizzamento pseudodiretto



Traduzione e avvio



Pseudo-istruzioni Assembler

- La maggior parte delle istruzioni assembler rappresentano uno a uno delle istruzioni macchina
- Pseduo-istruzioni: finzioni dell'immaginazione dell'assembler:

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

blt $t0, $t1, L   →  slt $at, $t0, $t1
                     bne $at, $zero, L
```

- `$at` (registro 1): assembler temporaneo

Produrre un modulo oggetto

- L'assembler (o il compilatore) traduce un programma in istruzioni macchina
- Fornisce informazioni per costruire un intero programma da diversi pezzi:
  - Intestazione: descrizione dei contenuti del modulo oggetto
  - Segmento di testo: istruzioni tradotte
  - Segmenti dei dati statici: dati allocati per la durata del programma
  - Informazioni di rilocazione: per i contenuti che dipendono dalla locazione assoluta del programma caricato in memoria
  - Tabella dei simboli: definizioni globali e riferimenti esterni
  - Informazioni di debug: per associare il programma al codice sorgente

Collegare un modulo oggetto

- Produce un'immagine eseguibile
  - Unisce i vari segmenti
  - Risolve le etichette (determina i loro indirizzi di memoria)
  - Corregge i riferimenti interni ed esterni
- Può lasciare dipendenze di locazione da sistemare tramite un loader rilocante
  - Ma con la memoria virtuale non ce n'è bisogno
  - Il programma può essere caricato in una locazione assoluta nello spazio della memoria virtuale

Caricare un programma

- Caricare da un file immagine sul disco in memoria
  1. Leggere l'intestazione per determinare le dimensioni del segmento testo e del segmento dati
  2. Creare lo spazio di indirizzamento per contenere i segmenti
  3. Copiare istruzioni e dati in memoria
  4. Inizializzare gli argomenti sullo stack passati al programma principale
  5. Inizializzare i registri (inclusi `$sp`, `$fp` e `$gp`)
  6. Saltare alla procedura di avviamento
     - Copia gli argomenti `$a0`, ... e invoca `main`
     - Quando `main` termina, invoca la procedura di sistema `exit`

Collegamento dinamico

- Collega/carica una procedura di libreria (.dll/.so) quando è invocata
  - Richiede che il codice della procedura sia rilocabile
  - Evita esplosione del codice a causa del collegamento statico di tutte le librerie indirizzate (transitivamente)
  - Sceglie automaticamente le nuove versioni delle librerie

La procedura scambia in C

- Illustriamo l'uso di istruzioni assembly per una funzione *bubblesort* in C
- Procedura scambia (foglia)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- `v` in `$a0`, `k` in `$a1`, `temp` in `$t0`

La procedura scambia

- Codice MIPS:

```
scambia: sll $t1, $a1, 2    # $t1 = k * 4
         add $t1, $a0, $t1  # $t1 = v + (k * 4)
                            #   (indirizzo di v[k])
         lw $t0, 0($t1)     # $t0 (temp) = v[k]
         lw $t2, 4($t1)     # $t2 = v[k + 1]
         sw $t2, 0($t1)     # v[k] = $t2 (v[k + 1])
         sw $t0, 4($t1)     # v[k + 1] = $t0 (temp)
         jr $ra             # ritorno alla procedura chiamante
```

La procedura ordina in C

- Non foglia (invoca `scambia`)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            scambia(v, j);
        }
    }
}
```

- `v` in `$a0`, `n` in `$a1`, `i` in `$s0`, `j` in `$s1`

Il corpo della procedura

```
          move $s2, $a0           # salva $a0 in $s2
          move $s3, $a1           # salva $a1 in $s3
          move $s0, $zero         # i = 0
for1tst:  slt  $t0, $s0, $s3      # $t0 = 0 se $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1  # vai a exit1 se $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, -1       # j = i - 1
for2tst:  slti $t0, $s1, 0        # $t0 = 1 se $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2  # vai a exit2 se $s1 < 0 (j < 0)
          sll  $t1, $s1, 2        # $t1 = j * 4
          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
          lw   $t3, 0($t2)        # $t3 = v[j]
          lw   $t4, 4($t2)        # $t4 = v[j + 1]
          slt  $t0, $t4, $t3      # $t0 = 0 se $t4 ≥ $t3
          beq  $t0, $zero, exit2  # vai a exit2 se $t4 ≥ $t3
          move $a0, $s2           # 1o parametro di scambia è v (vecchio $a0)
          move $a1, $s1           # 2o parametro di scambia è j
          jal  scambia            # invoca procedura scambia
          addi $s1, $s1, -1       # j -= 1
          j    for2tst            # salta a test del ciclo interno
exit2:    addi $s0, $s0, 1        # i += 1
          j    for1tst            # salta a test del ciclo esterno
```

L'intera procedura

```
ordina:  addi $sp,$sp, -20      # fare spazio per 5 registri
         sw $ra, 16($sp)        # salvare $ra sullo stack
         sw $s3,12($sp)         # salvare $s3 sullo stack
         sw $s2, 8($sp)         # salvare $s2 sullo stack
         sw $s1, 4($sp)         # salvare $s1 sullo stack
         sw $s0, 0($sp)         # salvare $s0 sullo stack
         …                      # corpo della procedura
         …
exit1:   lw $s0, 0($sp)         # copiare $s0 dallo stack
         lw $s1, 4($sp)         # copiare $s1 dallo stack
         lw $s2, 8($sp)         # copiare $s2 dallo stack
         lw $s3,12($sp)         # copiare $s3 dallo stack
         lw $ra,16($sp)         # copiare $ra dallo stack
         addi $sp,$sp, 20       # copiare lo stack pointer
         jr $ra                 # ritornare alla procedura chiamante
```

Intel x86 ISA

- Evoluzione con retro-compabilità
  - 8080 (1974): micropocessore a 8 bit
    - Accumulatore, più 3 coppie indice-registro
  - 8086 (1978): estensione a 16 bit del 8080
    - Insieme complesso di istruzioni (CISC)
  - 8087 (1980): coprocessore in virgola mobile
    - Aggiunge l'FPU e il registro stack
  - 80286 (1982): indirizzi a 24 bit, MMU
    - Memoria segmentata e protezione
  - 80386 (1985): estensione a 32 bit (ora IA-32)
    - Modi di indirizzamento e operazioni addizionali
    - Memoria paginata e segmentata

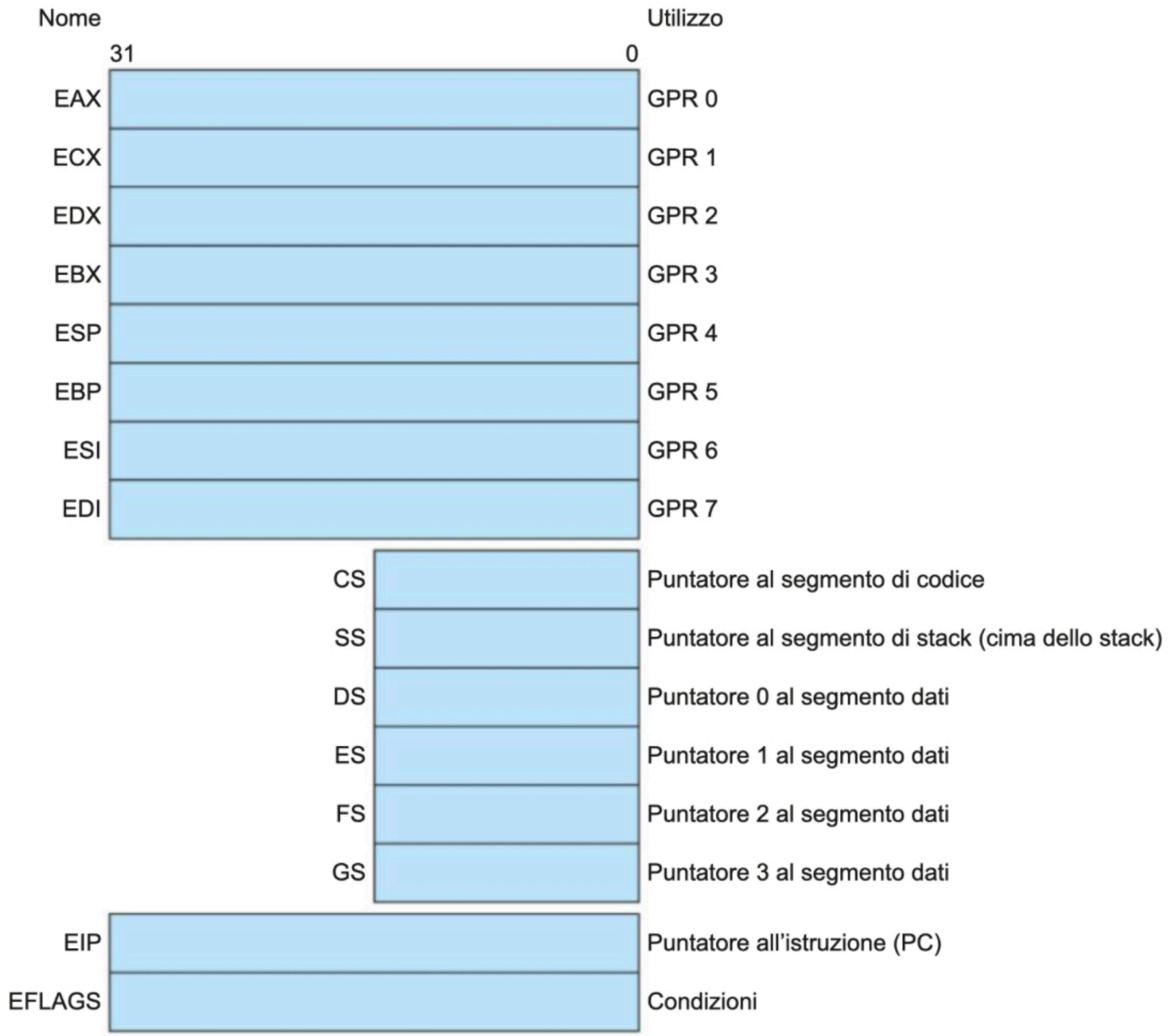
Intel x86 ISA

- Ulteriore evoluzione...
  - I486 (1989): pipelining, cache e FPU su chip
    - Avversari compatibili: AMD, Cyrix
  - Pentium (1993): superscalare, datapath a 64 bit
    - Le ultime versioni hanno aggiunto istruzioni MMX
    - L'infame bug FDIV
  - Pentium Pro (1995), Pentium II (1997)
    - Nuova microarchitettura
  - Pentium III (1999)
    - Aggiunte istruzioni SSE e registri associati
  - Pentium IV (2001)
    - Nuova microarchitettura
    - Aggiunte istruzioni SSE2

Intel x86 ISA

- E ancora...
  - AMD64 (2003): architettura estesa a 64 bit
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adottata da Intel (con modifiche)
    - Aggiunte istruzioni SSE3
  - Intel Core (2006)
    - Aggiunte istruzioni SSE4, support per macchine virtuali
  - AMD64 (2007): istruzioni SSE5
    - Intel ha deciso di fare altro...
  - Advanced Vector Extension (2008)
    - Registri SSE più lunghi, più istruzioni
- Se Intel non avesse esteso la retro-compabilità, i suoi avversari lo avrebbe fatto!
  - Eleganza tecnica $\neq$ successo di mercato

Registri x86 di base



Note conclusive

- Principi di progettazione
  1. La semplicità favorisce la regolarità
  2. Più piccolo è più veloce
  3. Rendere il caso comune veloce
  4. La buona progettazione richiede buoni compromessi
- Livelli di software/hardware
  - Compilatori, assembler, hardware
- MIPS: tipico esempio di ISA RISC
  - da confrontare con x86 (CISC)

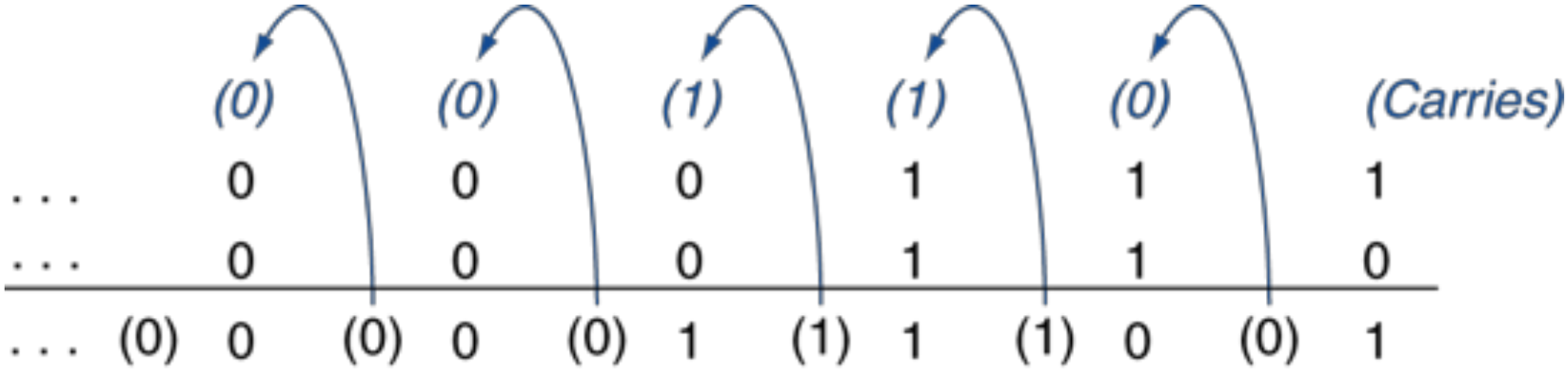
Capitolo 3

Aritmetica per Elaboratori

- Operazioni sugli interi
  - Addizione e sottrazione
  - Moltiplicazione e divisione
  - Gestione dell'overflow
- Numeri reali in virgola mobile
  - Rappresentazione e operazioni

Addizione Intera

- Esempio: 7 + 6



- Overflow se il risultato è fuori dall'intervallo
  - Sommando operandi positivo e negativo, no overflow
  - Sommando operandi positivi, overflow se il bit del segno è 1
  - Sommando operandi negativi, overflow se il bit del segno è 0

Sottrazione Intera

- Sommare la negazione del secondo operando
- Esempio: $7 - 6 = 7 + (-6)$

```
+7  0000 0000 ... 0000 0111
-6  1111 1111 ... 1111 1010
-------------------------------
+1  0000 0000 ... 0000 0001
```

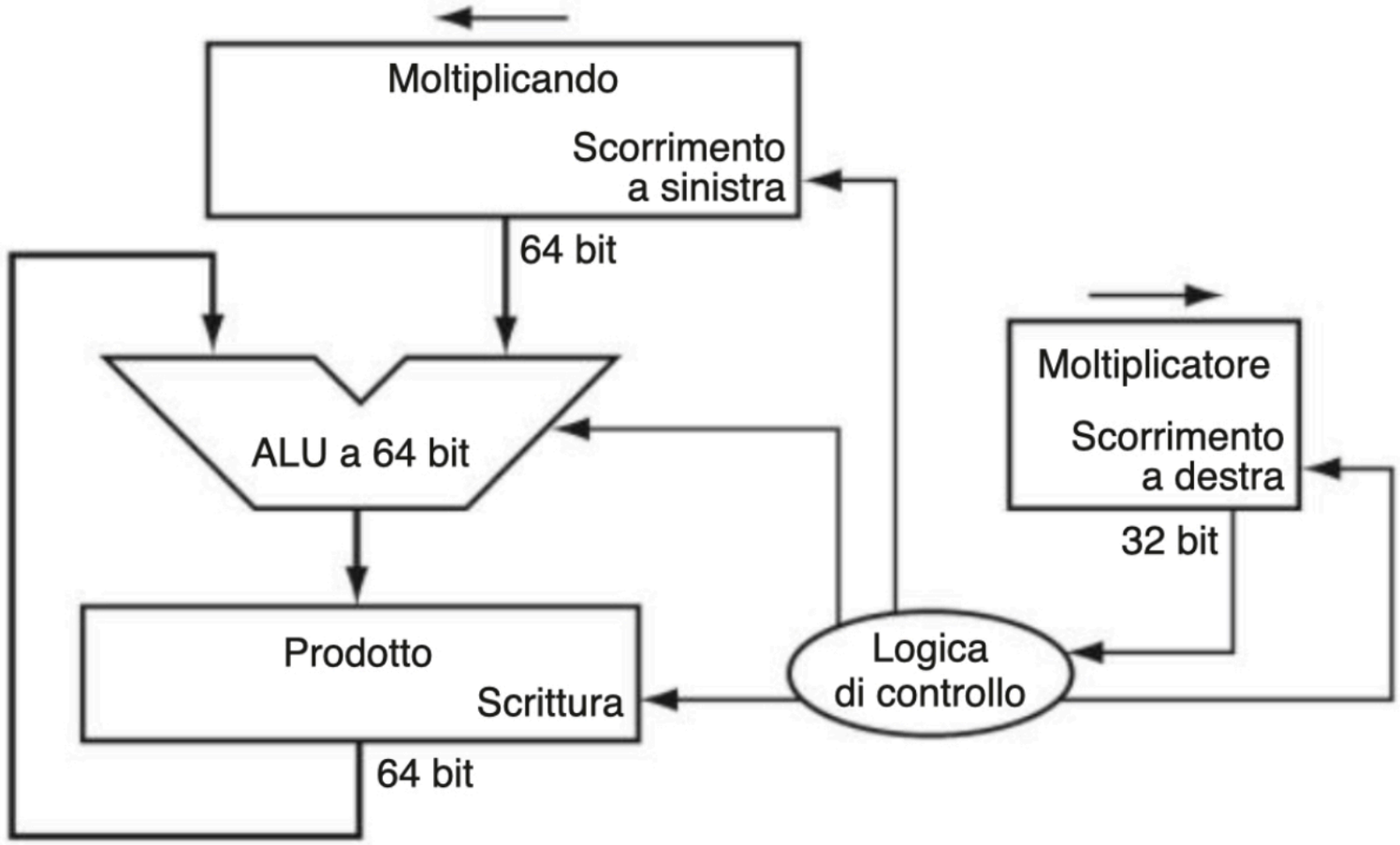
- Overflow se il risultato è fuori dall'intervallo
  - Sottraendo due operandi positivi o negativi, no overflow
  - Sottraendo un positivo da un negativo, overflow se il bit del segno è 0
  - Sottraendo un negativo da un positivo, overflow se il bit del segno è 1

Gestire l'overflow

- Alcuni linguaggi (per esempio, il C) ignorano l'overflow
  - Usano istruzioni MIPS `addu`, `addui`, `subu`
- Altri linguaggi (per esempio, Ada e Fortran) richiedono il sollevamento di un'eccezione
  - Usano istruzioni MIPS `add`, `addi`, `sub`
  - In caso di overflow, invocano il gestore dell'eccezione
    - Salvano PC nel registro EPC (e*xception program counter*)
    - Saltano all'indirizzo predefinito del gestore
    - L'istruzione `mfc0` (*move from coprocessor register*) può leggere il contenuto di EPC, per ritornare dopo le azioni correttive

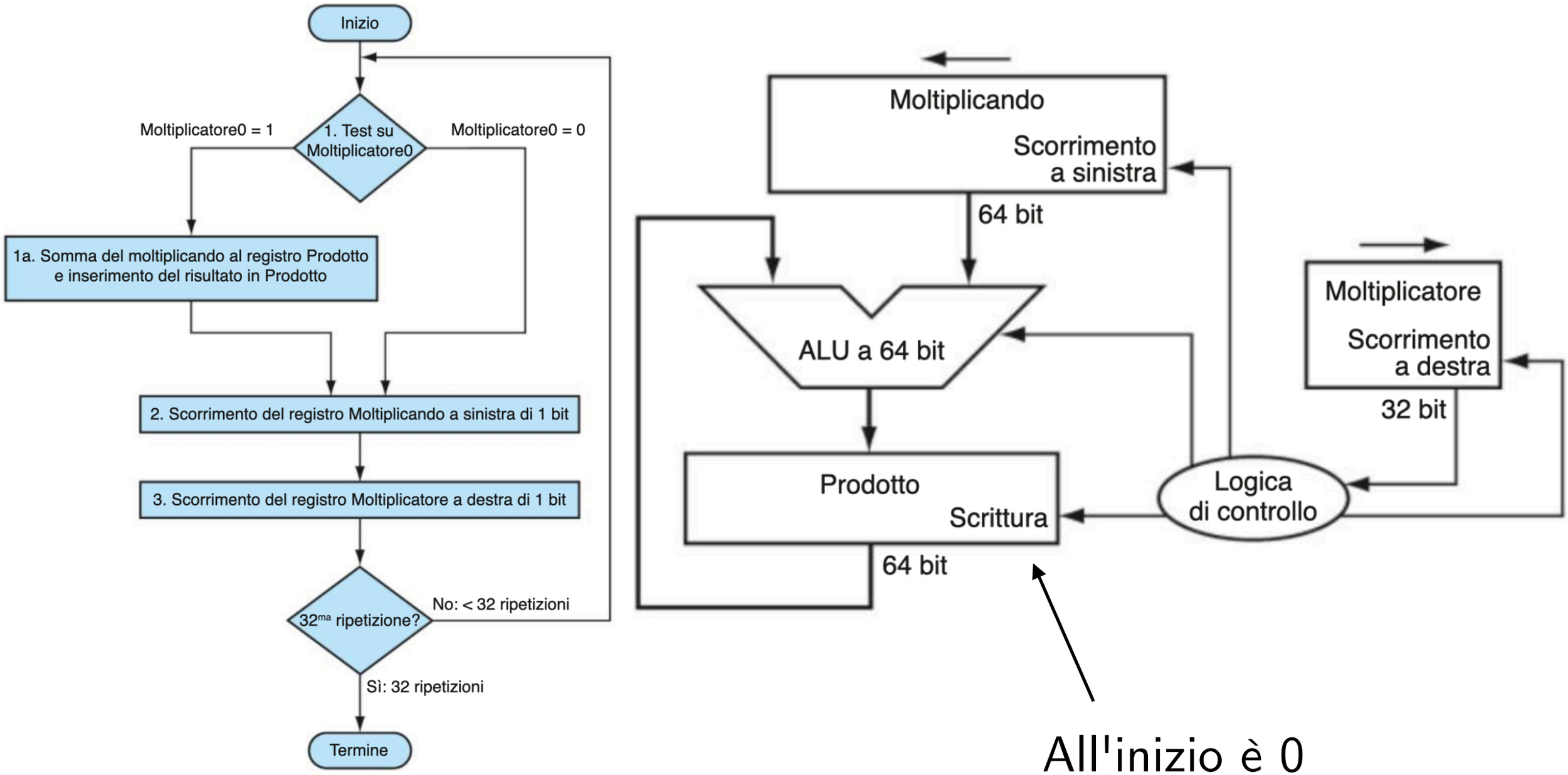
Moltiplicazione

Moltiplicando	×	1000_{dec}
Moltiplicatore		1001_{dec}
		1000
		0000
		0000
		1000
Prodotto		1001000_{dec}



La lunghezza del prodotto è la somma della lunghezza degli operandi

Hardware per la moltiplicazione



Virgola mobile

- Rappresentazione per numeri non interi
  - Inclusi numero molto piccoli e molto grandi
- Simile alla notazione scientifica
  - -2.34×10^{56} ⟵ normalizzato
  - $+0.002 \times 10^{-4}$ ⟵ non normalizzato
  - $+987.02 \times 10^{9}$ ⟵ non normalizzato
- In binario
  - $\pm 1.\text{xxxxxxx}_2 \times 2^{\text{yyyy}}$
- Tipi `float` e `double` in C

Standard virgola mobile

- Definito dallo standard IEEE Std 754-1985
- Sviluppato in risposta alla divergenza delle rappresentazioni
  - Problemi di portabilità nel codice scientifico
- Adesso adottato quasi universalmente
- Due rappresentazioni
  - Precisione singola (32-bit)
  - Precisione doppia (64-bit)

Formato IEEE

$$x = (-1)^s \times (1 + \text{mantissa}) \times 2^{(\text{esponente} - \text{polarizzazione})}$$

- s: bit del segno (0 non-negativo, 1 negativo)
- Significando normalizzato: $1.0 \leq |\text{significando}| < 2.0$
  - Il bit prima del punto decimale è sempre uguale a 1, quindi non è necessario rappresentarlo esplicitamente
  - Il significando è la mantissa con "1." ripristinato
- Esponente: notazione polarizzata
  - esponente = esponente reale + polarizzazione
  - Assicura che l'esponente è senza segno
  - Polarizzazione 127 (precisione singola), 1023 (precisione doppia)

Intervallo precisione singola

- Esponenti 00000000 e 11111111 riservati
- Valore più piccolo
  - Esponente: 00000001 → Esponente reale: $1 - 127 = -126$
  - Mantissa: 000...00 → Significando: 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Valore più grande
  - Esponente: 11111110 → Esponente reale: $254 - 127 = +127$
  - Mantissa: 111...11 → Significando ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Intervallo precisione doppia

- Esponenti 000000...00 e 111111...11 riservati
- Valore più piccolo
  - Esponente: 00000000001 → Esponente reale: $1 - 1023 = -1022$
  - Mantissa: 000...00 → Significando: 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Valore più grande
  - Esponente: 11111111110 → Esponente reale: $2046 - 1023 = +1023$
  - Mantissa: 111...11 → Significando ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Somma in virgola mobile

- Si consideri un esempio su 4 cifre decimali
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$

1. Allineare i punti decimali
   - Scorrere il numero con l'esponente più piccolo
   - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
2. Sommare i significandi
   - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
3. Normalizzare il risultato e controllare over/underflow
   - 1.0015×10^{2}
4. Arrotondare e ri-normalizzare se necessario
   - 1.002×10^{2}

Somma in virgola mobile

- Adesso Si consideri un esempio su 4 cifre binarie
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Allineare i punti binari
   - Scorrere il numero con l'esponente più piccolo
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
2. Sommare i significandi
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalizzare il risultato e controllare over/underflow
   - $1.000_2 \times 2^{-4}$, no over/underflow
4. Arrotondare e ri-normalizzare se necessario
   - $1.000_2 \times 2^{-4}$ (nessun cambiamento) $= 0.0625$

Istruzioni FP nel MIPS

- L'hardware FP è il coprocessore 1
  - Processore aggiuntivo che estende l'ISA
- Registri FP separati
  - Precisione singole a 32 bit: `$f0, $f1, … $f31`
  - Appaiati per precisione doppia: `$f0/$f1, $f2/$f3, …`
    - La release 2 dell'ISA del MIPS supporta 32 ×registri FP a 64 bit
- Le istruzioni FP operano sono su registri FP
  - I programmi generalmente non eseguono operazioni intere su dati FP o viceversa
  - Più registri con minimo impatto sulla dimensione del codice
- Istruzioni FP di lettura e scrittura
  - `lwc1, ldc1, swc1, sdc1`
    - Per esempio, `ldc1 $f8, 32($sp)`

Parallelismo sui dati

- La applicazioni grafiche e audio possono sfruttare l'esecuzione simultanea di operazioni su vettori
  - Esempio: sommatore a 128 bit:
    - Sedici somme a 8 bit
    - Otto somme a 16 bit
    - Quattro somme a 32 bit
- Detto anche parallelismo a livello di parola, parallelismo vettoriale, o Single Instruction, Multiple Data (SIMD)

Note conclusive

- I bit non hanno un significato intrinseco
  - L'interpretazione dipende dalle istruzioni applicate
- Rappresentazione digitale dei numeri
  - Intervallo e precisione finiti
  - Bisogna tenerne conto nei programmi
- Aritmetica supportata nell'ISA
  - Interi con e senza segno
  - Approssimazioni in virgola mobile dei reali
- Intervallo e precisione limitati
  - Le operazioni possono causare overflow e underflow
- ISA del MIPS
  - Istruzioni principali: 54 più frequentemente usate
  - Altre istruzioni: meno frequenti

Capitolo 4

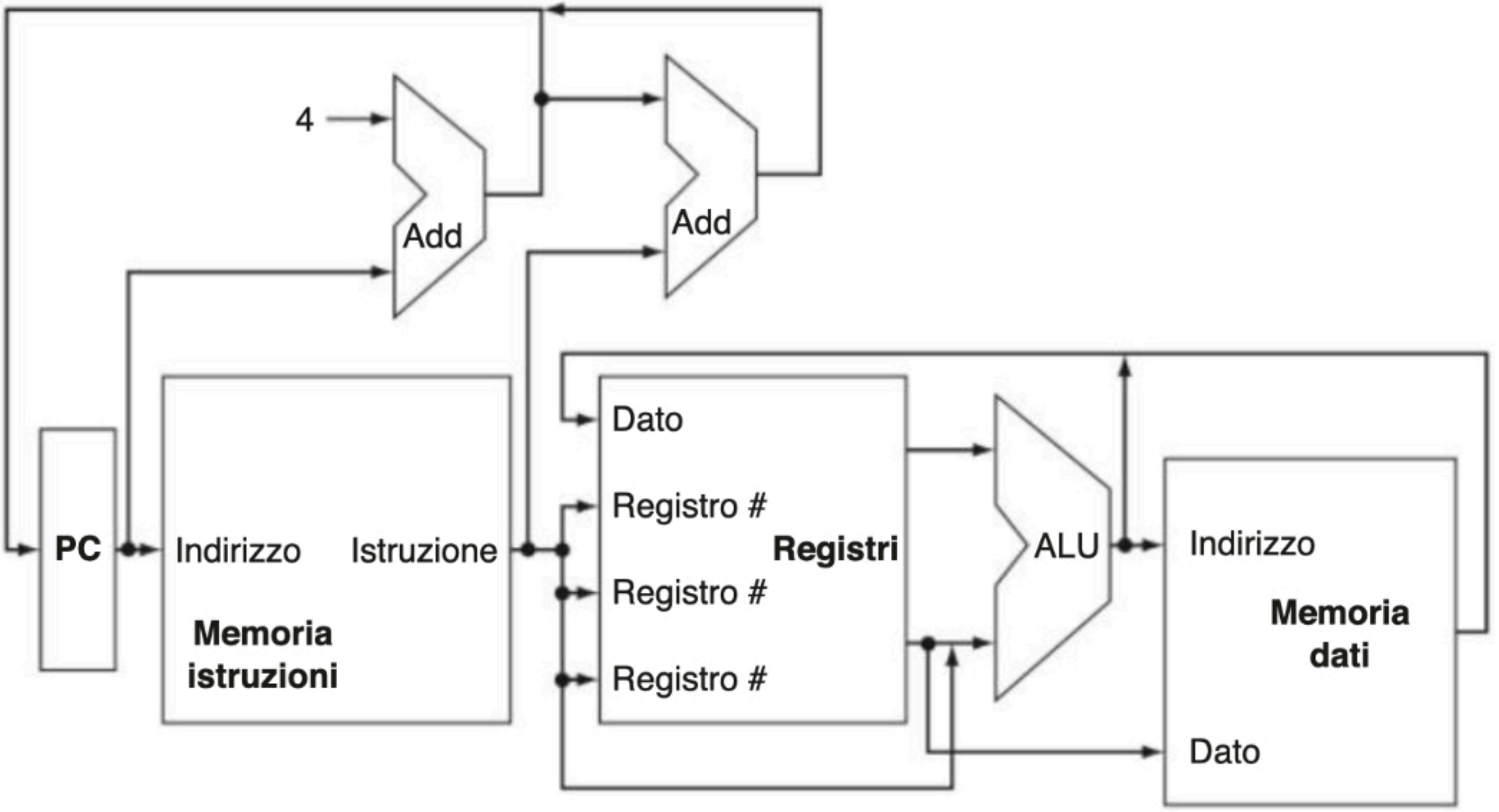
Introduzione

- Fattori delle prestazioni della CPU
  - Instruction count
    - Determinato dall'ISA e dal compilatore
  - CPI e periodo di clock
    - Determinato dall'hardware della CPU
- Studieremo due implementazioni del MIPS
  - Una versione semplificata
  - Una versione più realistica in pipeline
- Semplice sottoinsieme di istruzioni
  - Accesso alla memoria: `lw, sw`
  - Operazioni aritmetico/logiche: `add, sub, and, or, slt`
  - Controllo del flusso: `beq, j`

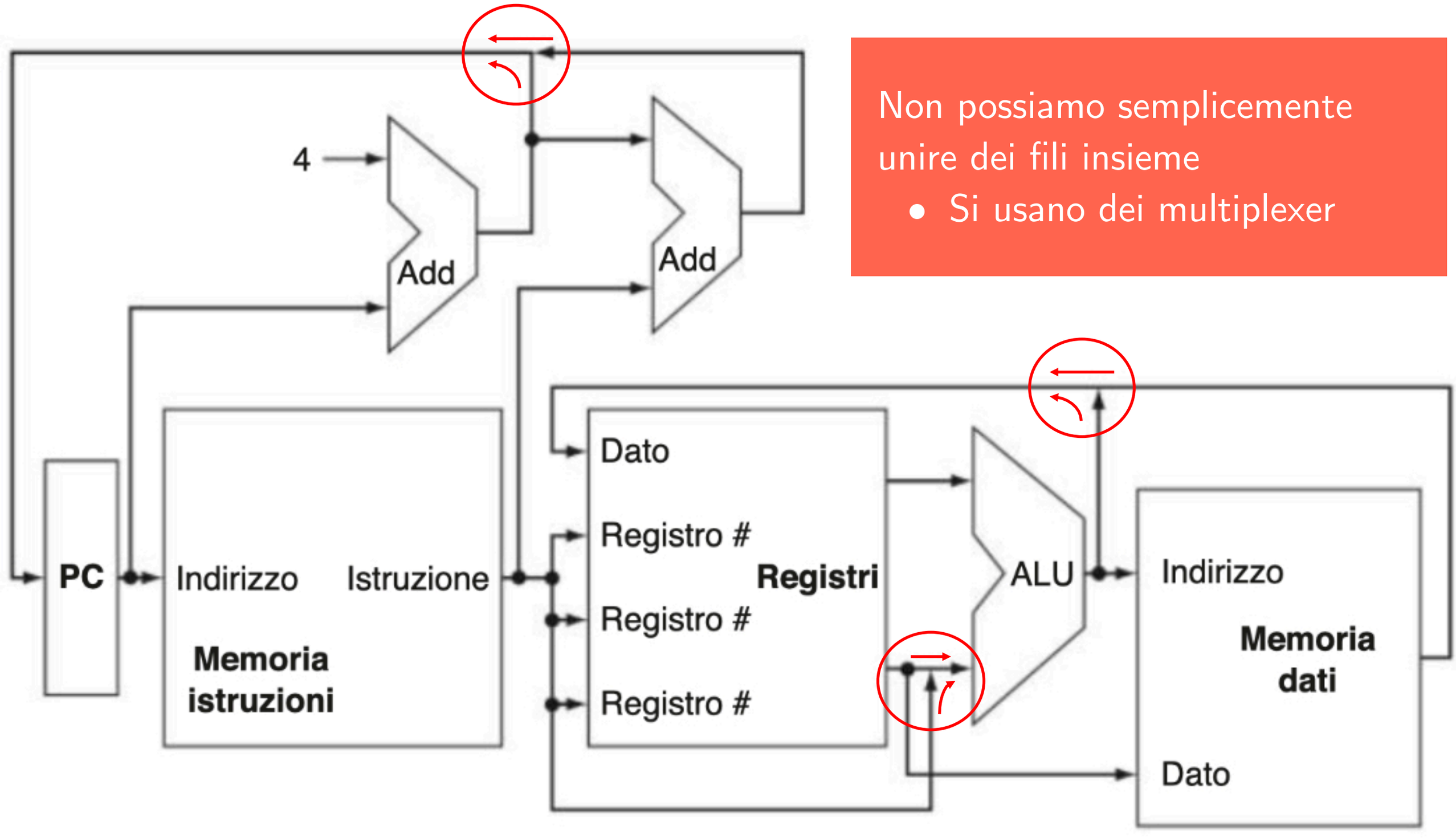
Esecuzione di un'istruzione

- PC $\rightarrow$ memoria istruzioni, preleva (fetch) l'istruzione
- Numeri di registro $\rightarrow$ accesso ai registri, lettura dei registri
- A seconda della classe di istruzioni
  - Usare la ALU per calcolare
    - Un risultato aritmetico
    - L'indirizzo di memoria di una load/store
    - Indirizzo di destinazione di un branch
  - Accedere alla memoria dati per load/store
  - PC $\leftarrow$ indirizzo di destinazione o PC + 4

Panoramica della CPU



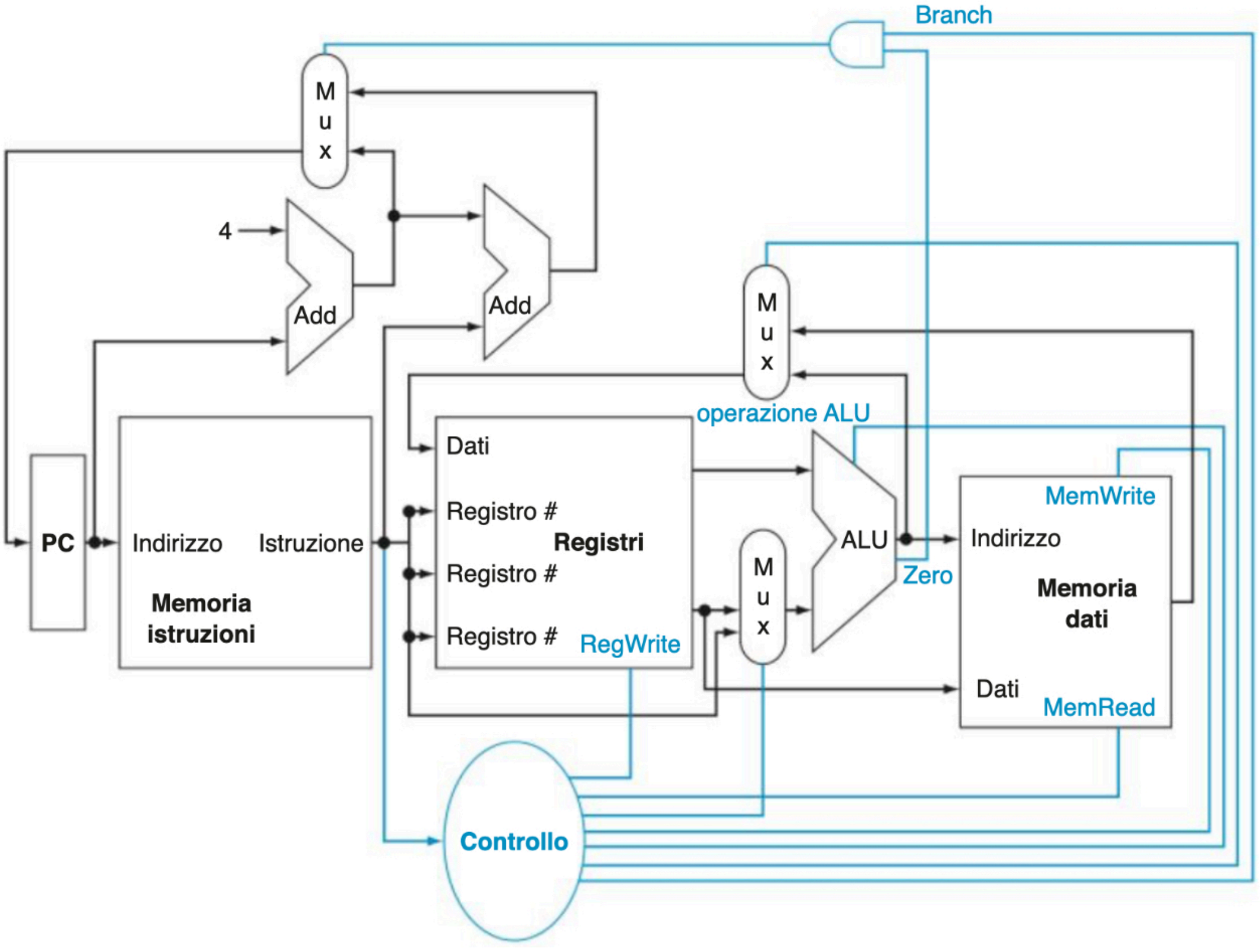
Multiplexer



Non possiamo semplicemente unire dei fili insieme

- Si usano dei multiplexer

Controllo

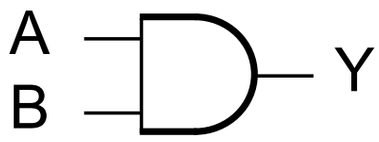

Branch
M
u
x
4
Add
Add
M
u
x
operazione ALU
PC
Indirizzo
Istruzione
Memoria
istruzioni
Dati
Registro #
Registri
Registro #
Registro #
RegWrite
M
u
x
ALU
Zero
MemWrite
Indirizzo
Memoria
dati
Dati
MemRead
Controllo

Basi di progettazione logica

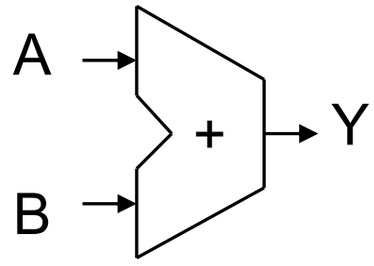
- L'informazione è codificata in binario
  - Tensione bassa = 0, Tensione alta = 1
  - Un filo per bit
  - Dati su più bit codificati su bus multi-filo
- Elemento combinatorio
  - Opera sui dati
  - L'uscita è una funzione dell'ingresso
- Elemento sequenziale (stato)
  - Memorizza l'informazione

Elementi combinatori

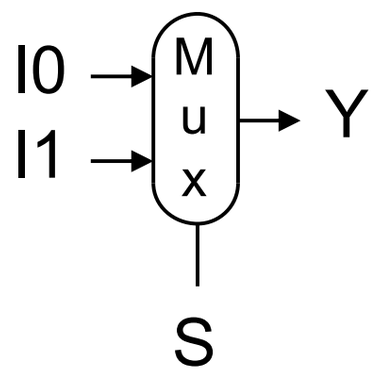
Porta AND
Y = A & B



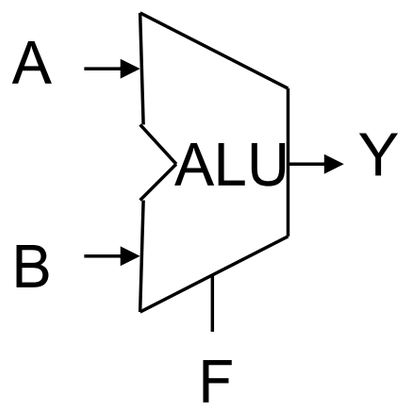
Sommatore
Y = A + B



Multiplexer
Y = S ? I1 : I0

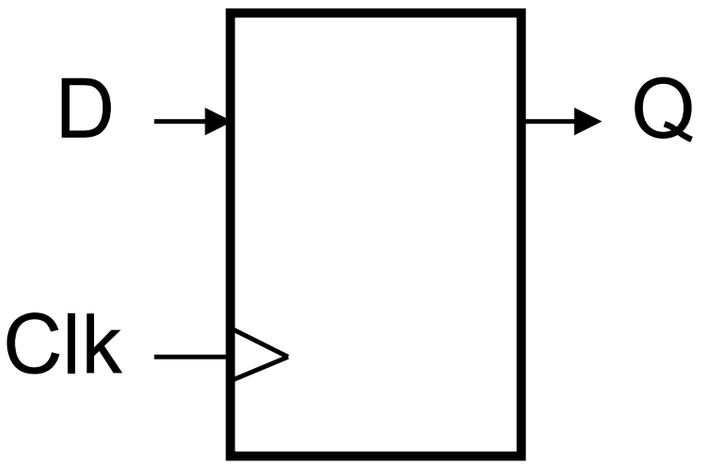


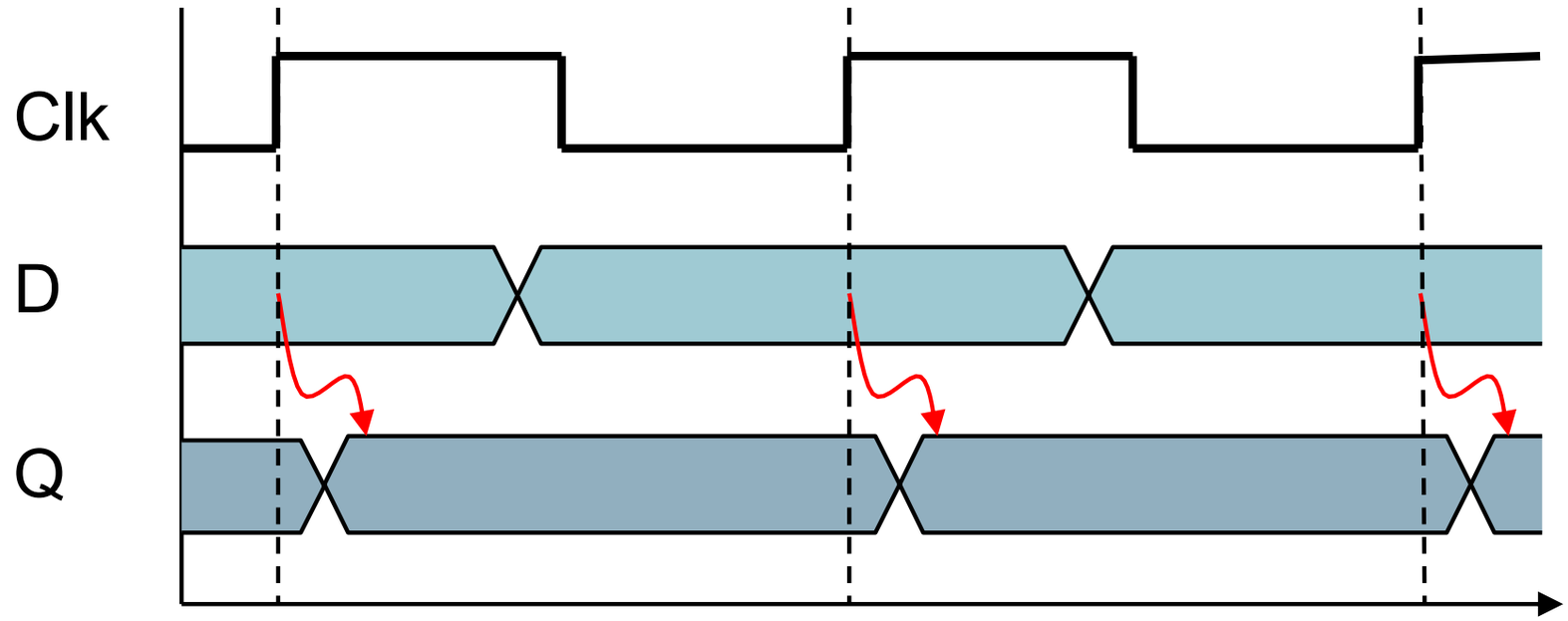
Unità Logico/Aritmentica
Y = F(A,B)



Elementi sequenziali

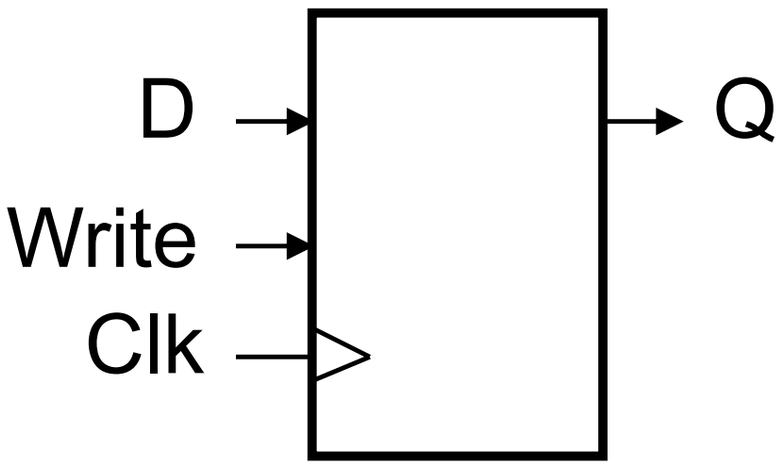
- Registro: memorizza dati in un circuito
  - Usa il segnale di clock per determinare quando aggiornare il valore memorizzato
  - *Edge-triggered*: aggiorna quando `Clk` cambia da 0 a 1

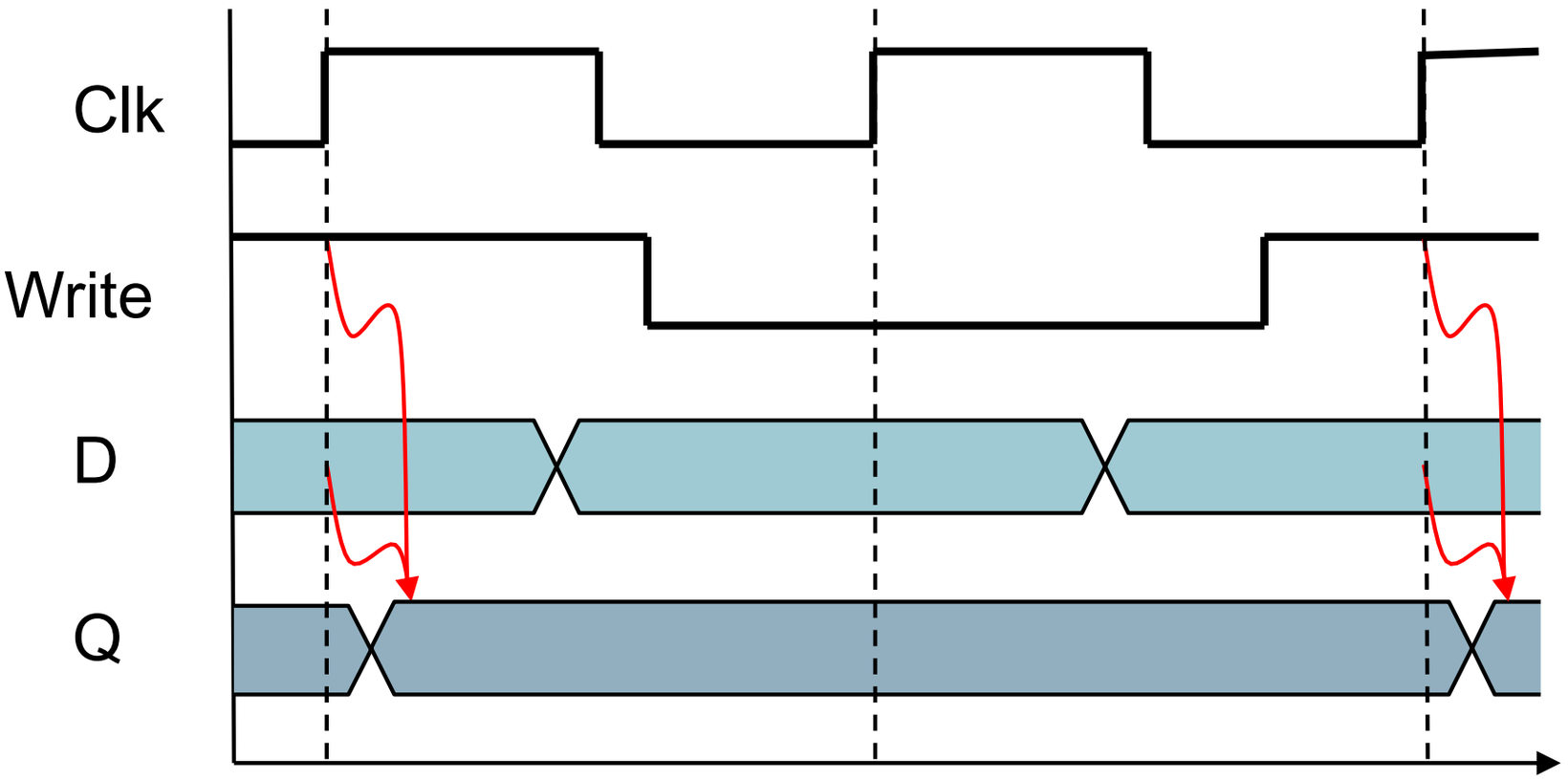




Elementi sequenziali

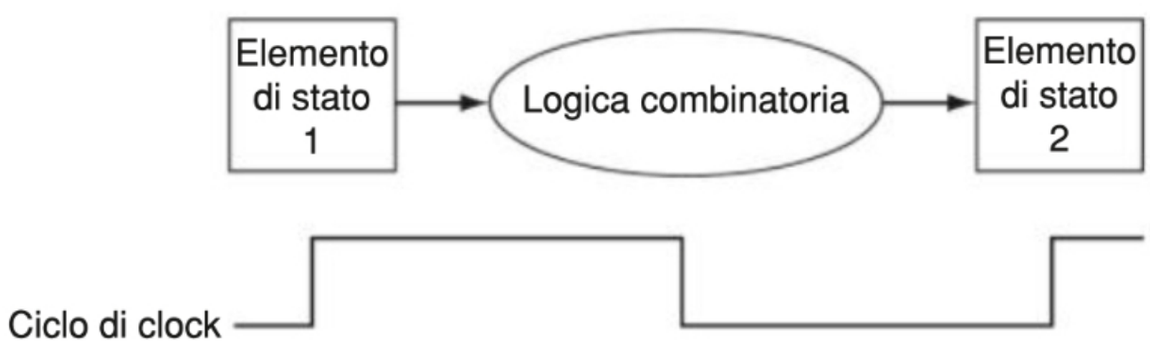
- Registro con controllo della scrittura
  - Aggiorna unicamente sul fronte del clock quando il l'ingresso di controllo `Write` vale `1`
  - Usato quando il valore memorizzato è richiesto dopo

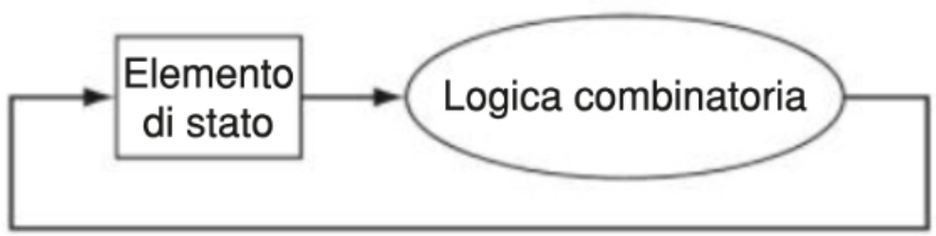




Metodologia di temporizzazione

- La logica combinatoria trasforma i dati durante i cicli di clock
  - Tra due fronti del clock
  - Ingresso da elementi di stato, uscita a elementi di stato
  - L'operazione più lunga determina il periodo di clock

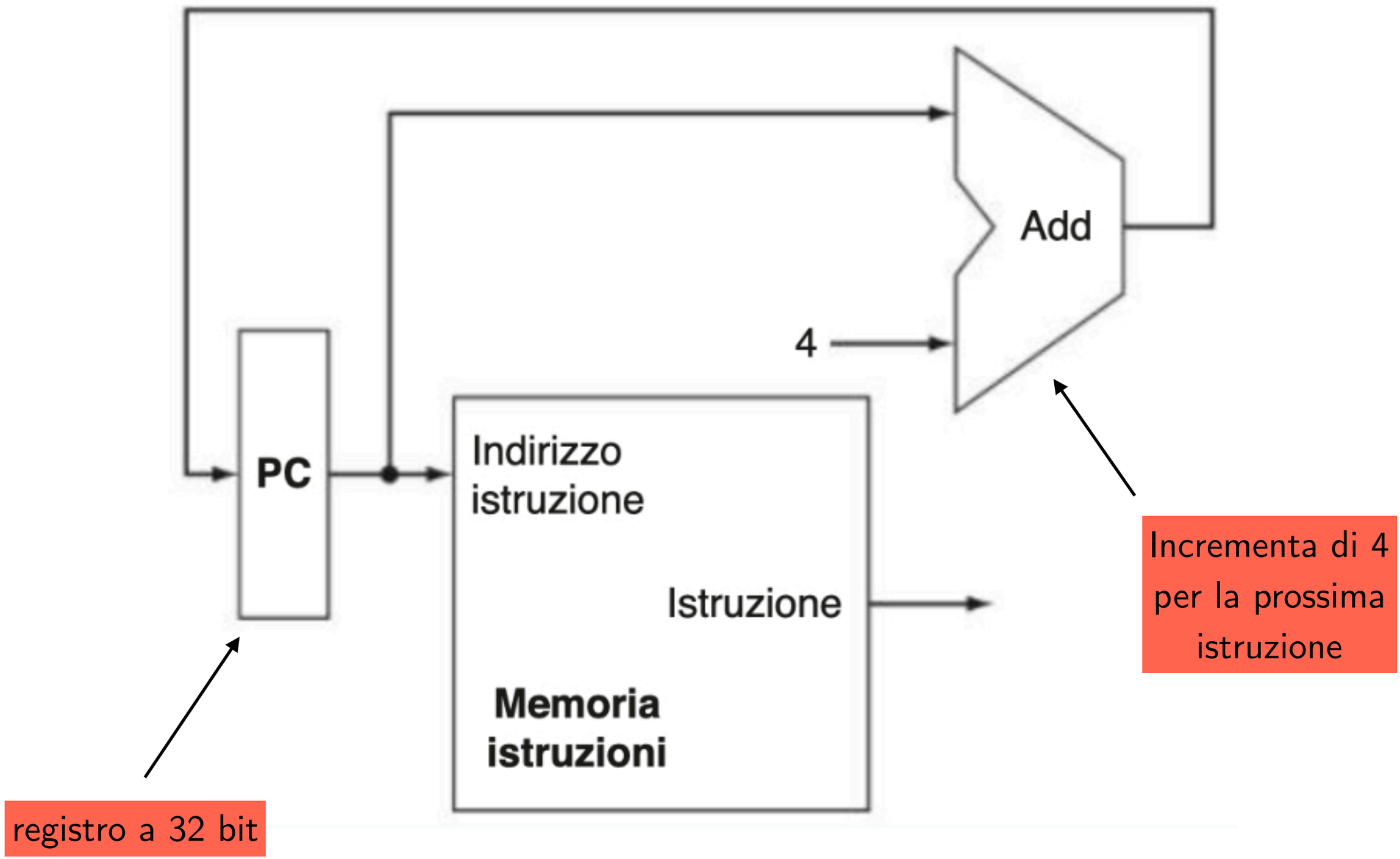




Realizzare una unità di elaborazione

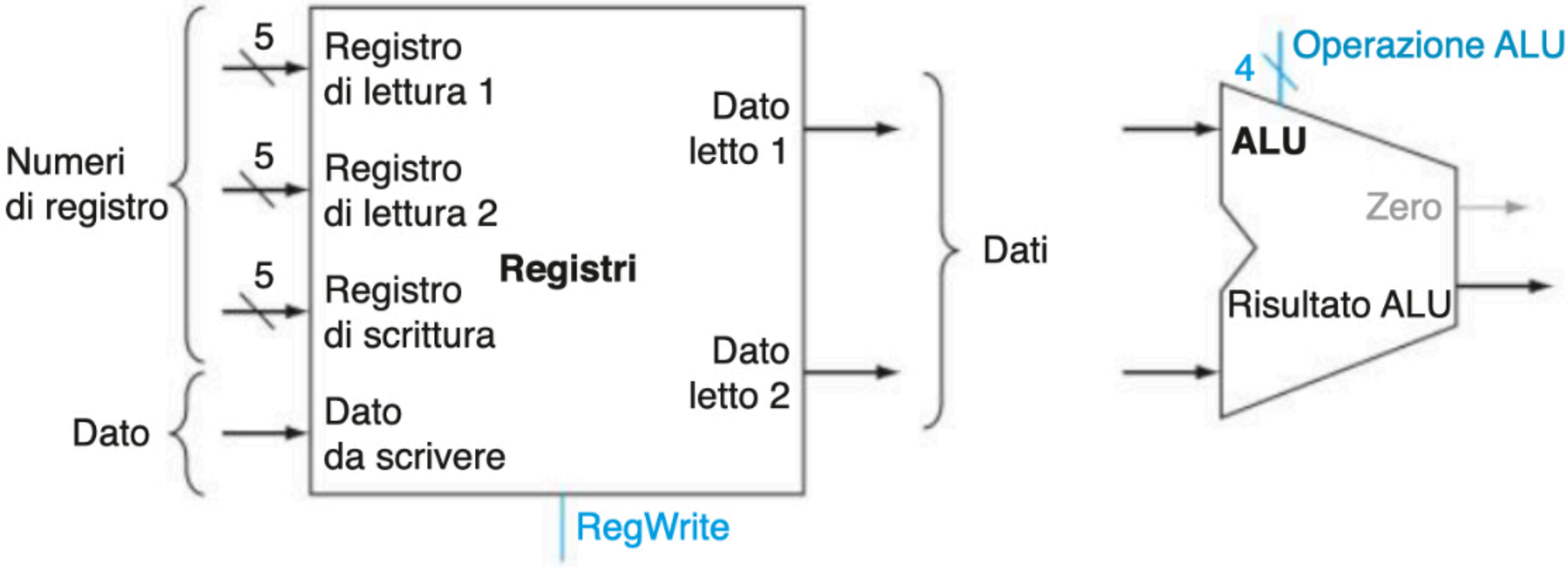
- Unità di elaborazione (datapath)
  - Elementi che elaborano i dati e gli indirizzi nella CPU
    - Registri, ALU, multiplexer, memorie, ...
- Costruiremo l'unità di elaborazione del MIPS incrementalmente
  - Rielaborando la panoramica di progetto

Fetch di un'istruzione



Istruzioni in formato R

- Leggere due operandi da registri
- Eseguire l'operazione aritmetico/logica
- Scrivere il risultato in un registro

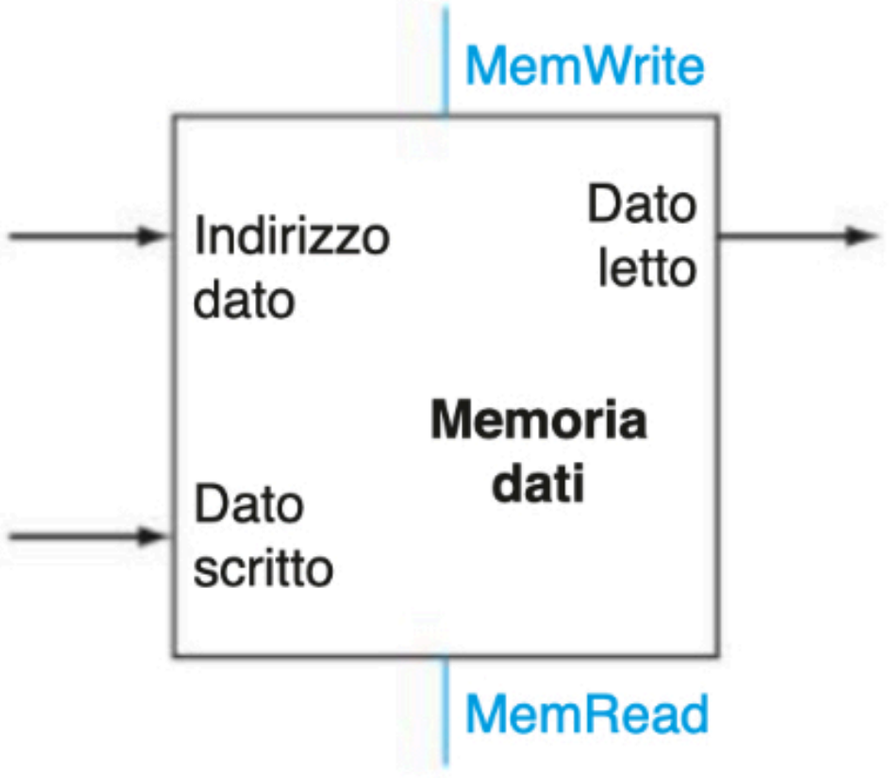


a. Registri

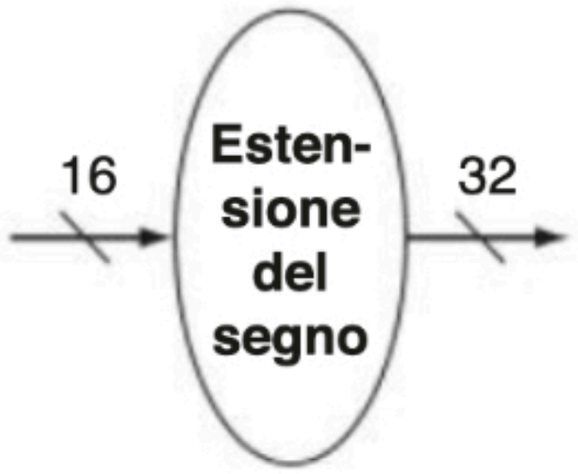
b. ALU

Istruzioni load/store

- Leggere operandi dai registri
- Calcolare l'indirizzo usando l'offset su 16 bit
  - Usando l'ALU, ma con offset esteso in segno
- Load: leggere dalla memoria e aggiornare registro
- Store: leggere un valore dal registro e copiarlo in memoria



a. Unità di memoria dati

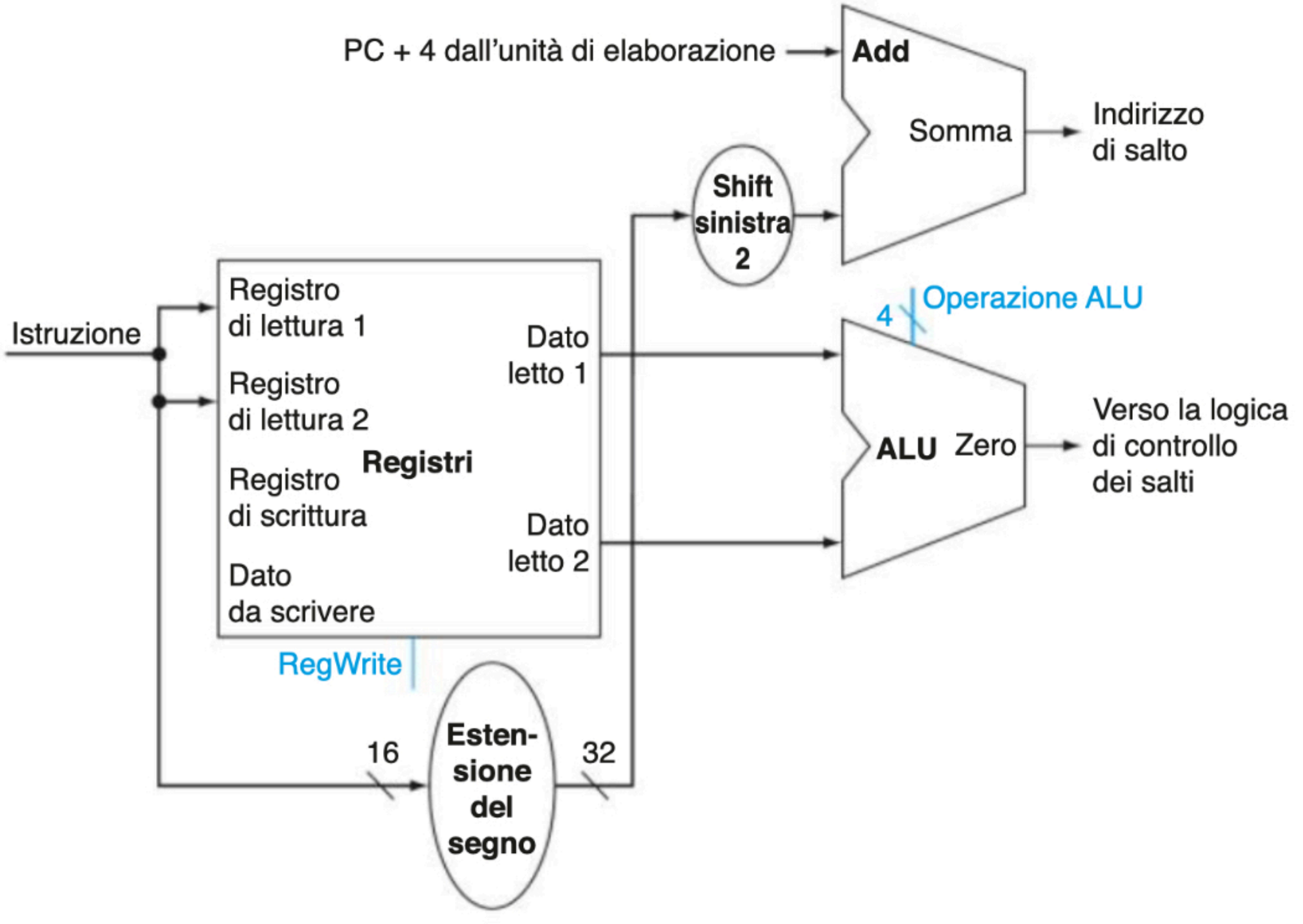


b. Unità di estensione del segno

Istruzioni di salto

- Leggere operandi dai registri
- Confrontare gli operandi
  - Usare l'ALU, sottrarre e controllare l'output `Zero`
- Calcolare l'indirizzo di destinazione
  - Spiazzamento esteso in segno
  - Scorrimento a sinistra di 2 bit (spiazzamento di parola)
  - Sommare a PC + 4
    - Già calcolato dal fetch dell'istruzione

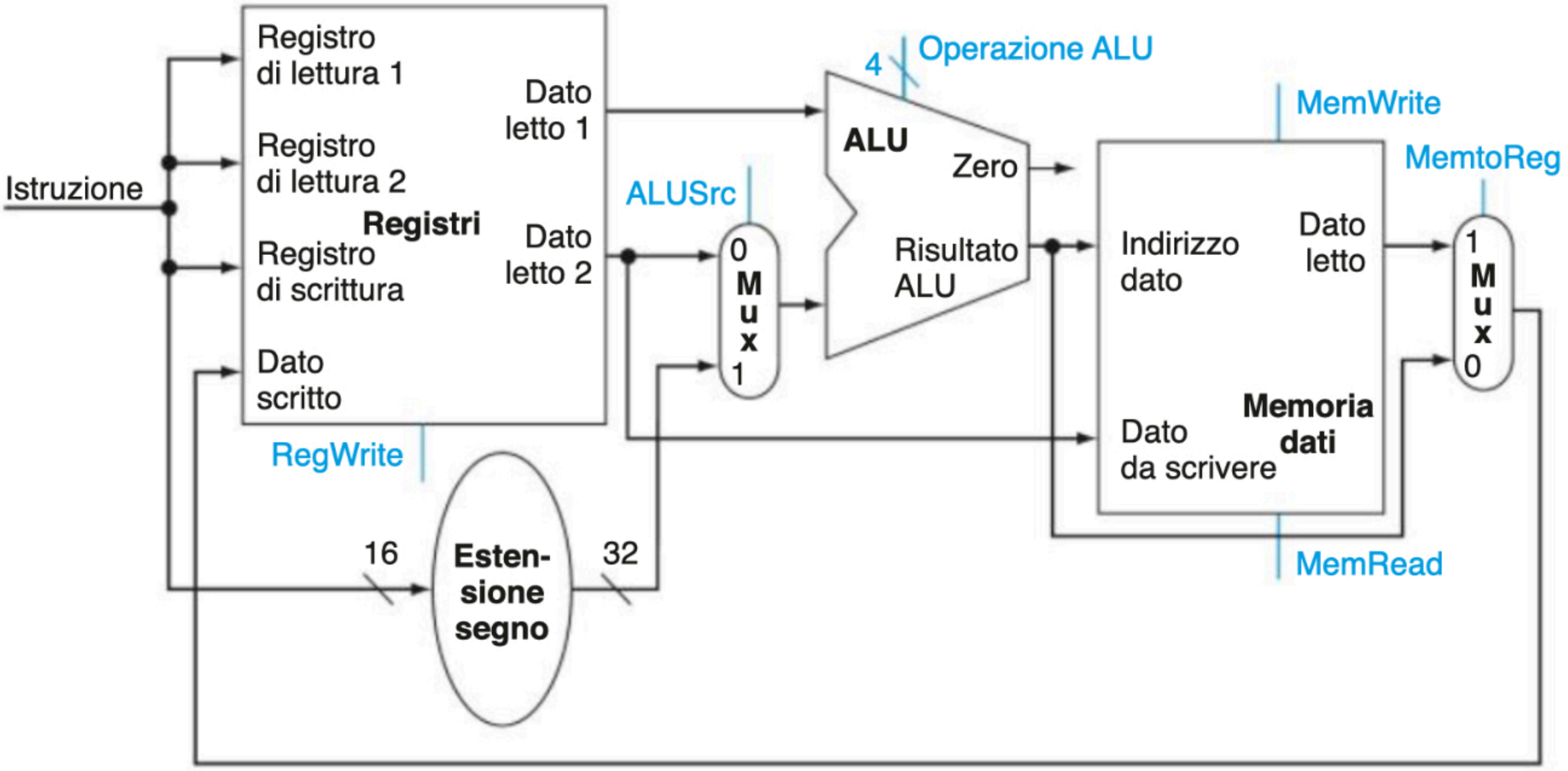
Istruzioni di salto



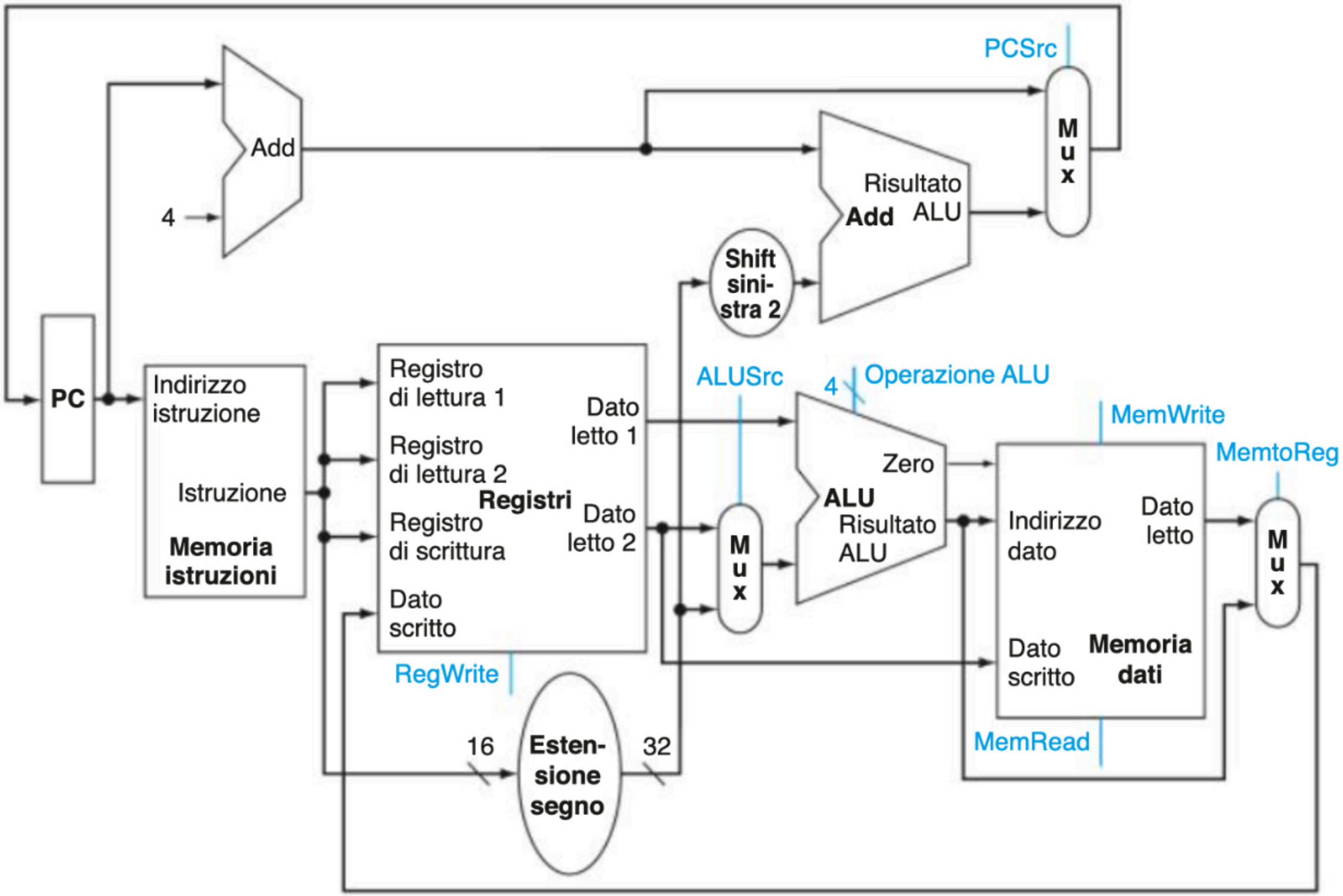
Comporre gli elementi

- L'unità di elaborazione più semplice cercherà di eseguire tutte le istruzioni in un singolo ciclo di clock
  - Nessuna risorsa dell'unità di elaborazione potrà essere utilizzata più di una volta per ogni istruzione
  - Quindi servirà quindi una memoria delle istruzioni separata da quella dei dati
- Usare multiplexer quando sorgenti alternative di dati sono usate per istruzioni differenti

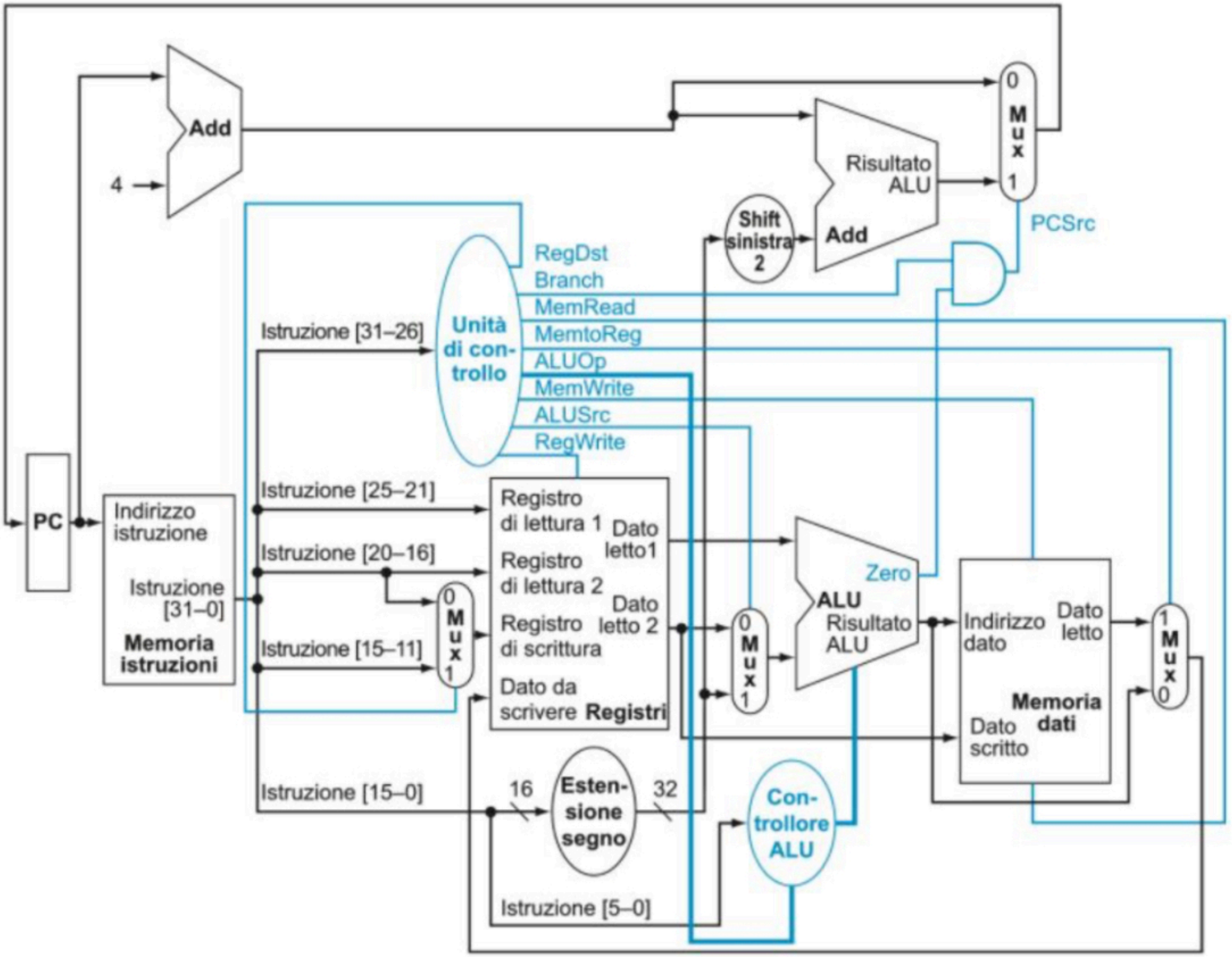
Datapath formato R/load/store



Datapath completo



Datapath con controllo



Controllo della ALU

- La ALU è usata per:
  - Load/Store: F = somma
  - Branch: F = sottrazione
  - Tipo R: F dipende dal campo `funct`

Controllo ALU	Funzione
0000	AND
0001	OR
0010	somma
0110	sottrazione
0111	set less than
1100	NOR

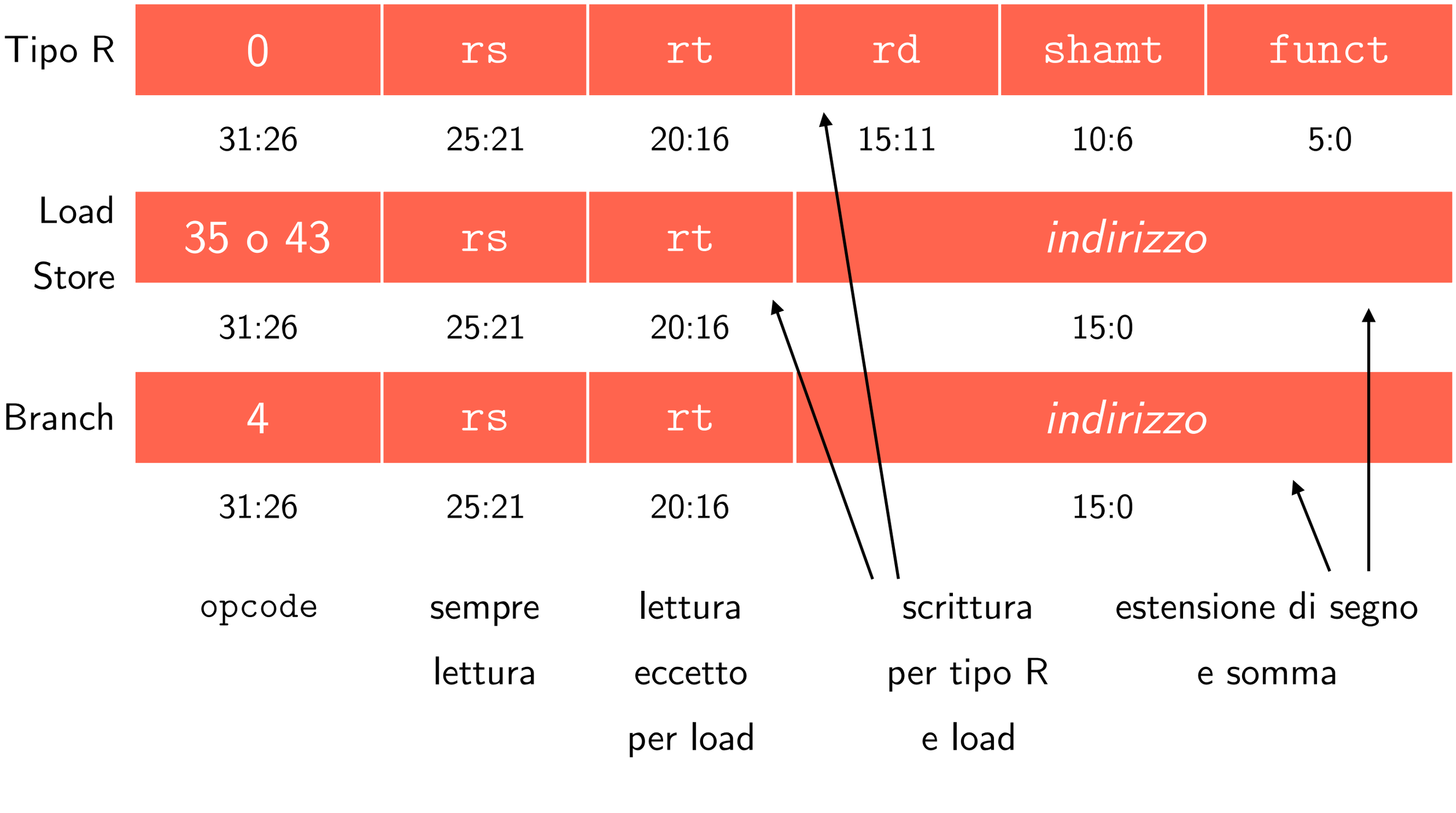
Controllo della ALU

- Si assuma un campo di controllo su 2 bit chiamato ALUOp derivato da opcode
  - Logica combinatoria per calcolare il controllo della ALU

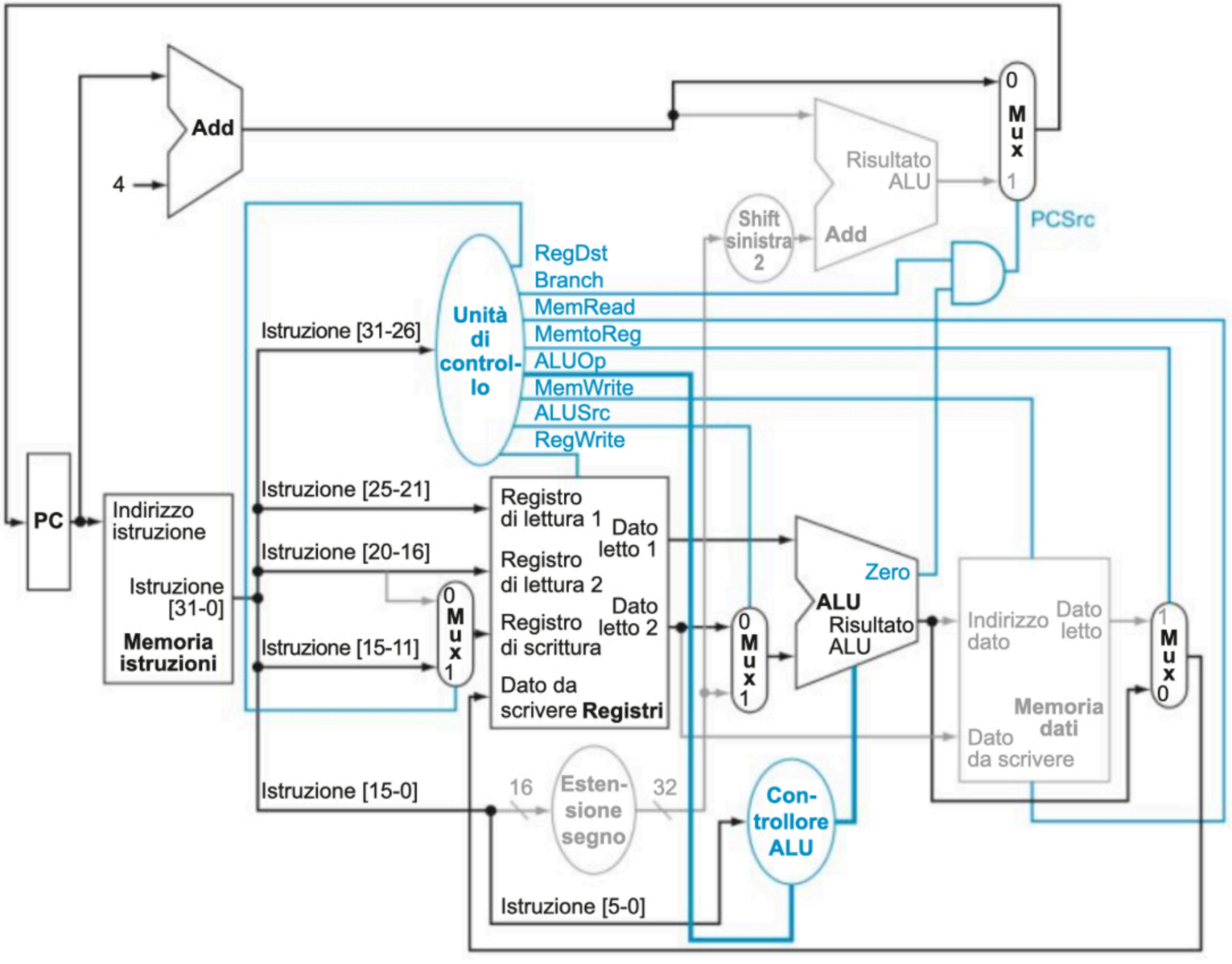
opcode	ALUOp	Operazione	funct	Funzione ALU	ALU control
lw	00	Load word	XXXXXX	somma	0010
sw	00	Store word	XXXXXX	somma	0010
beq	01	Branch equal	XXXXXX	sottrazione	0110
R-type 0111 1100	10	add	100000	somma	0010
		subtract	100010	sottrazione	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Unità di controllo principale

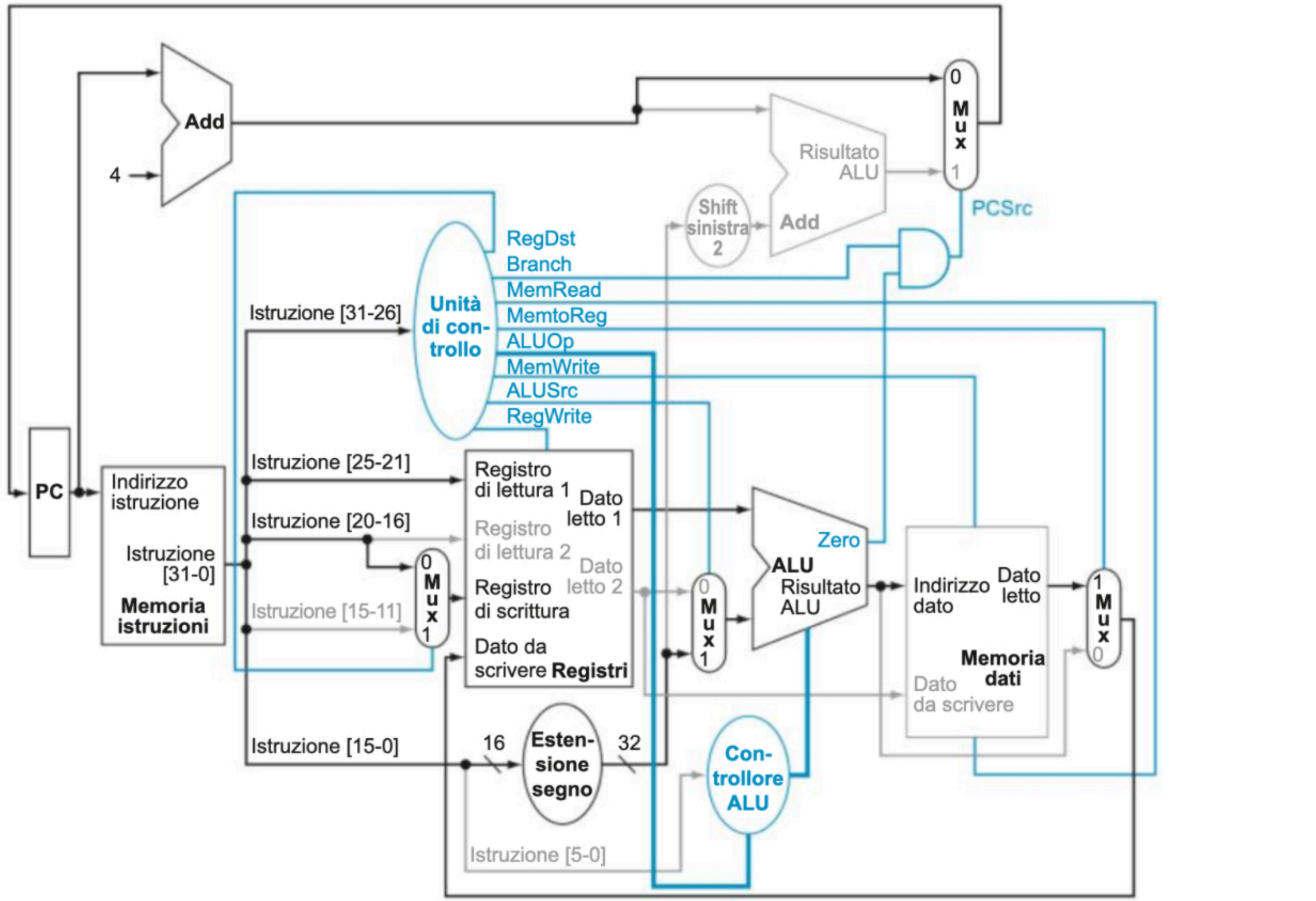
I segnali di controllo sono derivati dall'istruzione



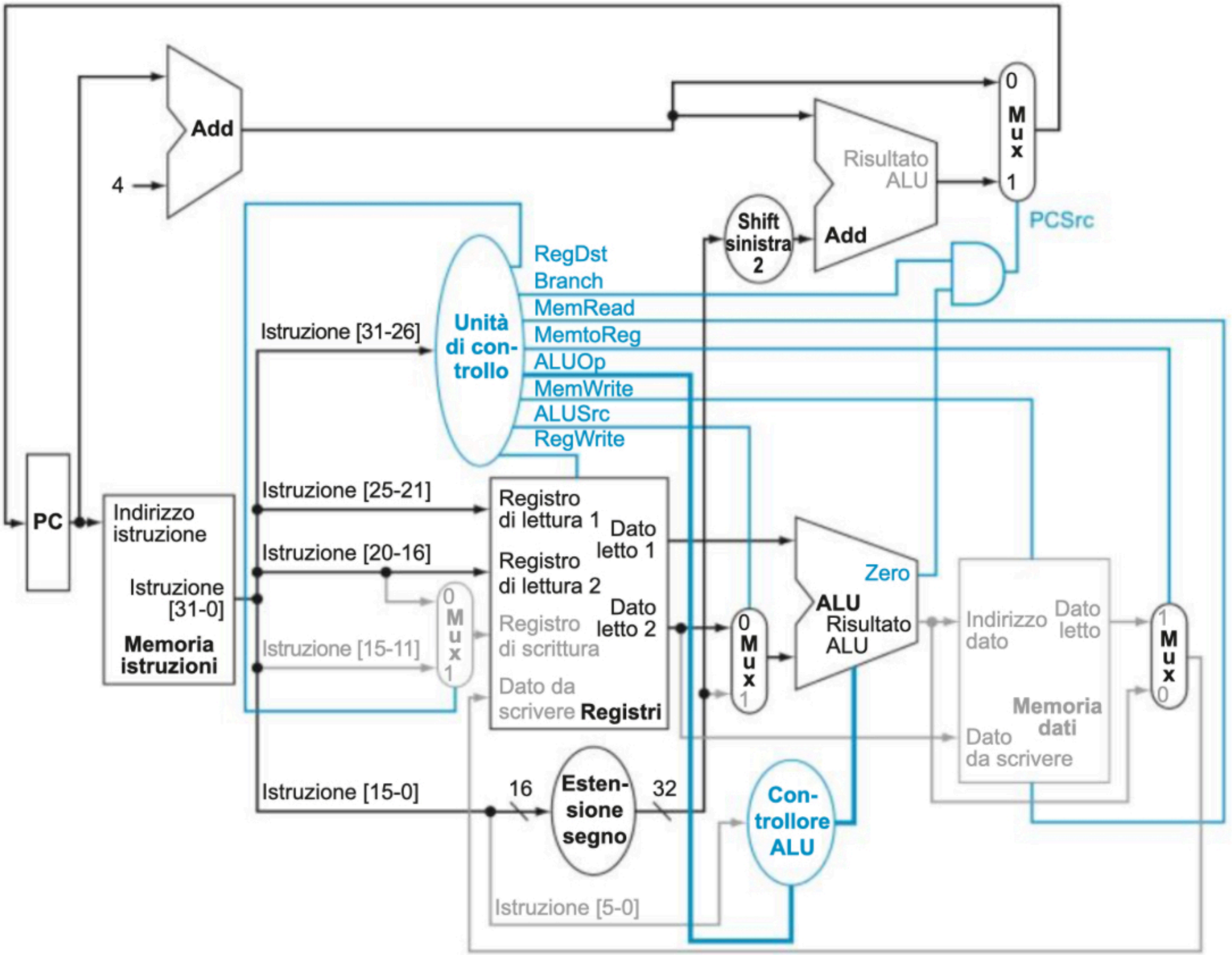
Istruzione di tipo R



Istruzione Load



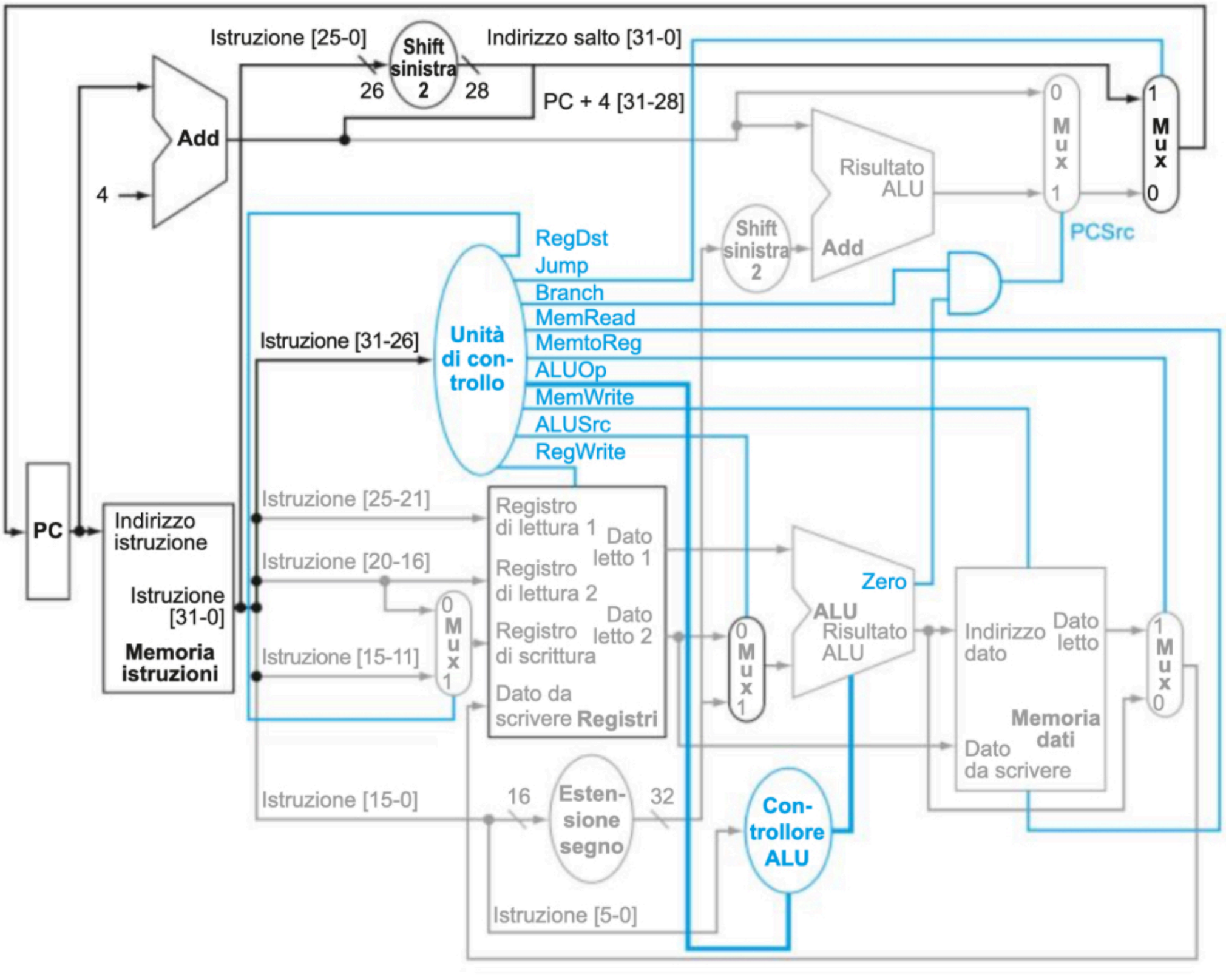
Istruzione Branch-on-Equal



Implementare i salti

- L'istruzione `jump` usa indirizzi di parola
- Aggiornare PC con la concatenazione di
  - I 4 bit più significativi del vecchio valore di PC
  - L'indirizzo di salto su 26 bit
  - `00`
- Necessità di un segnale di controllo extra decodificato da `opcode`

Datapath con aggiunta dei salti

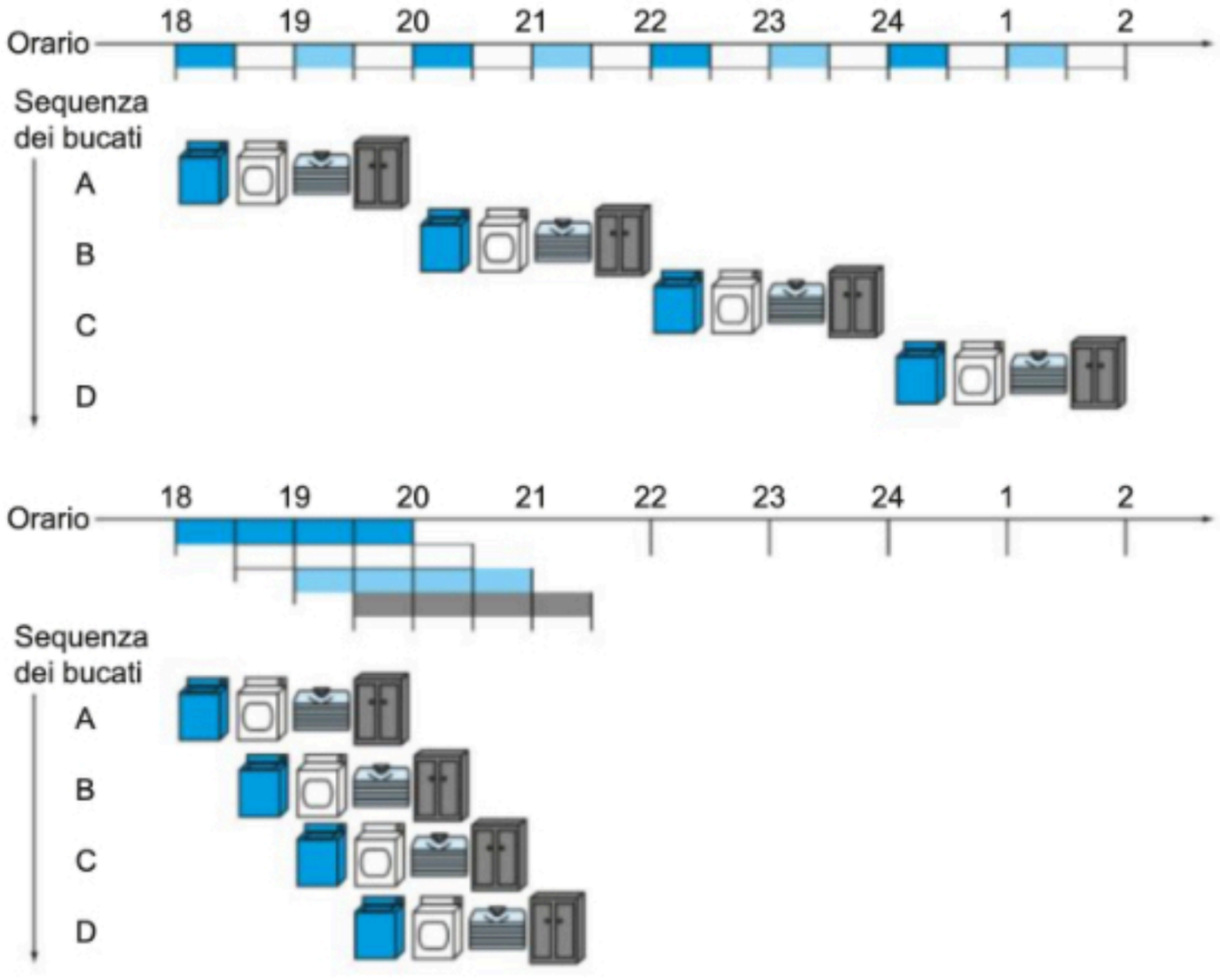


Problemi di prestazioni

- Il ritardo più lungo determina il periodo di clock
  - Percorso critico: istruzione Load
  - Memoria istruzioni $\rightarrow$ Registri $\rightarrow$ ALU $\rightarrow$ Memoria dati $\rightarrow$ Registri
- Non è ammissibile cambiare periodo di clock per istruzioni diverse
- Viola un principio di progetto
  - Rendere il caso comune veloce
- Miglioreremo le prestazioni tramite *pipelining*

Analogia della pipeline

- Lavanderia: sovrapporre esecuzioni
  - Il parallelismo migliora le prestazioni



- Quattro carichi:
  - Speedup
    $= 8 / 3.5 = 2.3$
- Non-stop:
  - Speedup
    $= 2n/(0.5n + 1.5) \cong$
    $4 =$ numero di stadi

Pipeline del MIPS

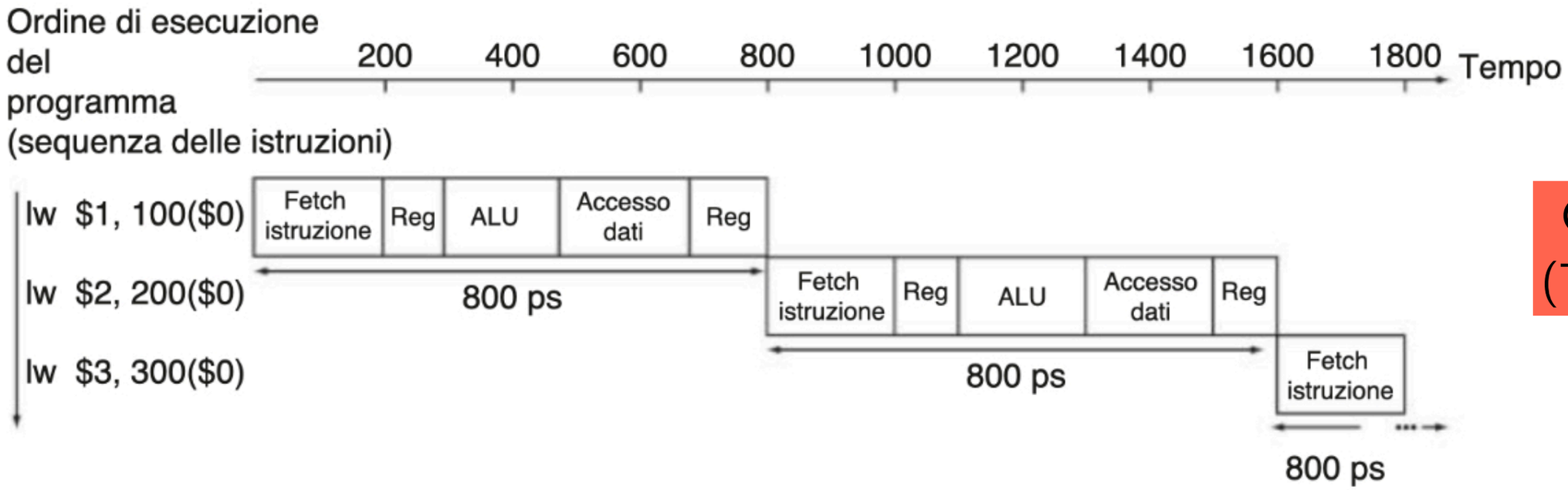
- Cinque stadi, un passo per stadio
  - IF: fetch dell'istruzione dalla memoria istruzioni
  - ID: decodifica dell'istruzione e lettura dei registri
  - EX: esecuzione dell'operazione o calcolo dell'indirizzo
  - MEM: accesso all'operando nella memoria dati
  - WB: scrittura del risultato in un registro

Prestazioni della pipeline

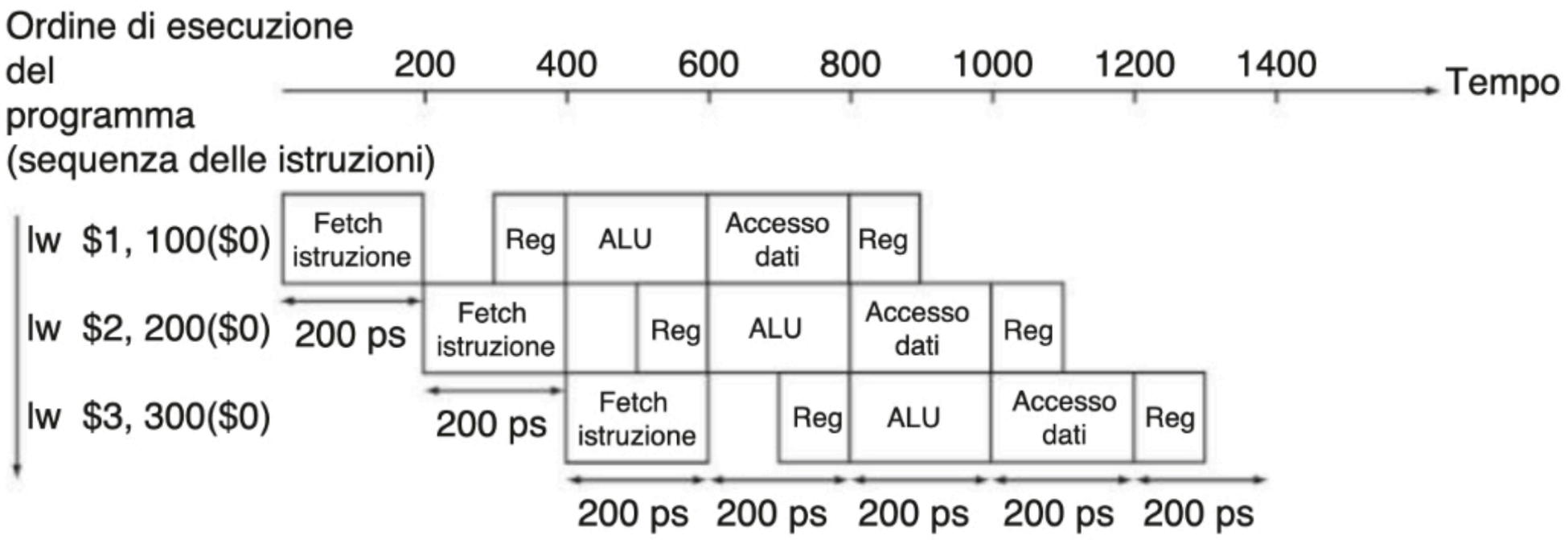
- Si assuma che il tempo di ogni stadio sia:
  - 100 ps per leggere o scrivere un registro
  - 200 ps per tutti gli altri stadi
- Si confrontino le unità di elaborazione in pipeline e a ciclo singolo

Tipo di istruzione	Fetch istruzione	Lettura registri	Operazione ALU	Accesso ai dati in memoria	Scrittura Registri	Tempo Totale
`lw`	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
`sw`	200 ps	100 ps	200 ps	200 ps		700 ps
Tipo R	200 ps	100 ps	200 ps		100 ps	600 ps
`beq`	200 ps	100 ps	200 ps			500 ps

Prestazioni della pipeline



Ciclo singolo
(T_C = 800 ps)



Pipeline
(T_C = 200 ps)

Speedup della pipeline

- Se tutti gli stadi sono bilanciati
  - cioè, tutti richiedono lo stesso tempo
  - Tempo tra le istruzioni$_{\text{CON PIPELINE}}$
    = Tempo tra le istruzioni$_{\text{SENZA PIPELINE}}$/Numero di stadi
- Se sbilanciato, lo speedup è inferiore
  - Latenza (tempo per ogni istruzione) non decresce

Pipeline e progetto dell'ISA

- L'ISA del MIPS è progettata per il pipelining
  - Tutte le istruzioni sono a 32 bit
    - Più facile prelevare e decodificare in un ciclo
    - c.f. x86: istruzioni da 1 a 17 bit
- Pochi e regolari formati di istruzioni
  - Si possono decodificare e leggere i registri in un passo
- Indirizzamento load/store
  - Si può calcolare l'indirizzo nel 3° stadio e accedere alla memoria nel 4° stadio
- Allineamento in memoria degli operandi
  - Un accesso alla memoria richiede un solo ciclo

Hazard (criticità)

- Situazioni che impediscono l'inizio della prossima istruzione nel prossimo ciclo
- Hazard strutturali
  - Una risorsa necessaria è occupata
- Hazard sui dati
  - Bisogna attendere che l'istruzioni completi la sua lettura/ scrittura del dato
- Hazard sul controllo
  - La decisione sull'azione di controllo dipende dall'istruzione precedente

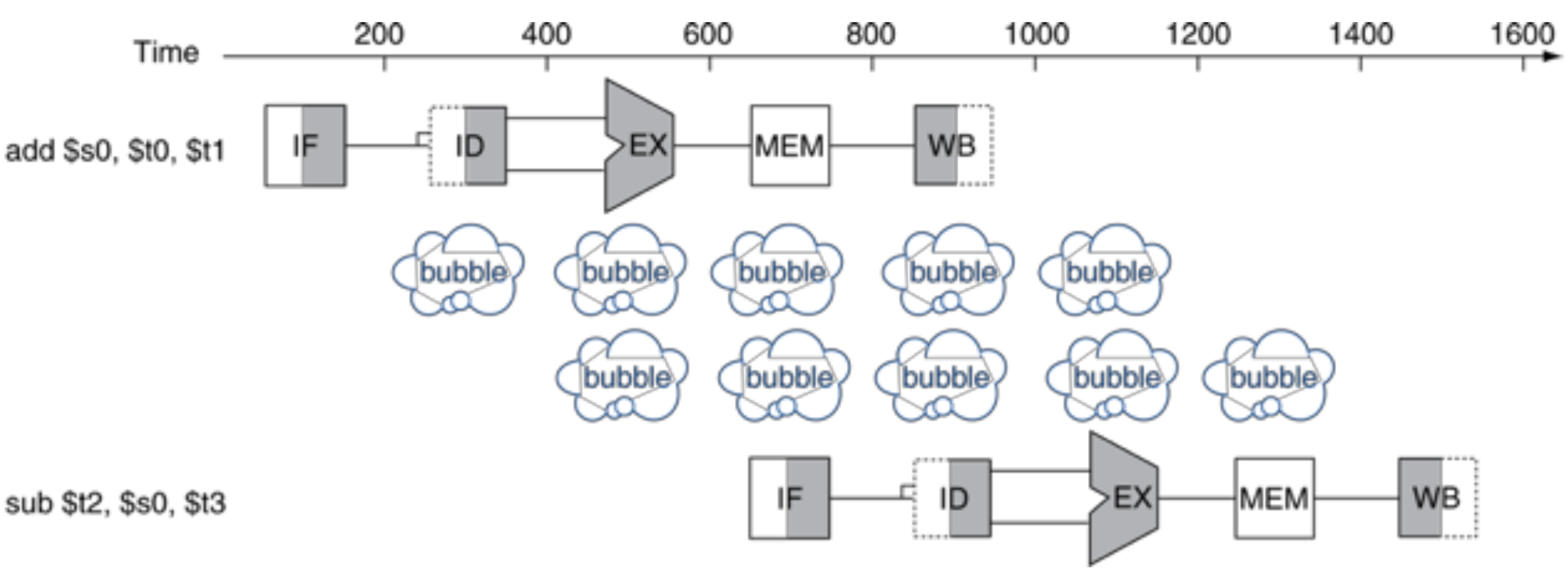
Hazard strutturali

- Conflitto per l'uso di una risorsa
- Nella pipeline del MIPS con una singola memoria
  - Load/store richiede l'accesso al dato
  - Il fetch dell'istruzione dovrebbe andare in *stallo* per quel ciclo
    - Causerebbe una *bolla* della pipeline
- Quindi, le unità di elaborazione in pipeline richiedono memorie istruzioni/dati separate
  - O cache istruzioni/dati separate

Hazard sui dati

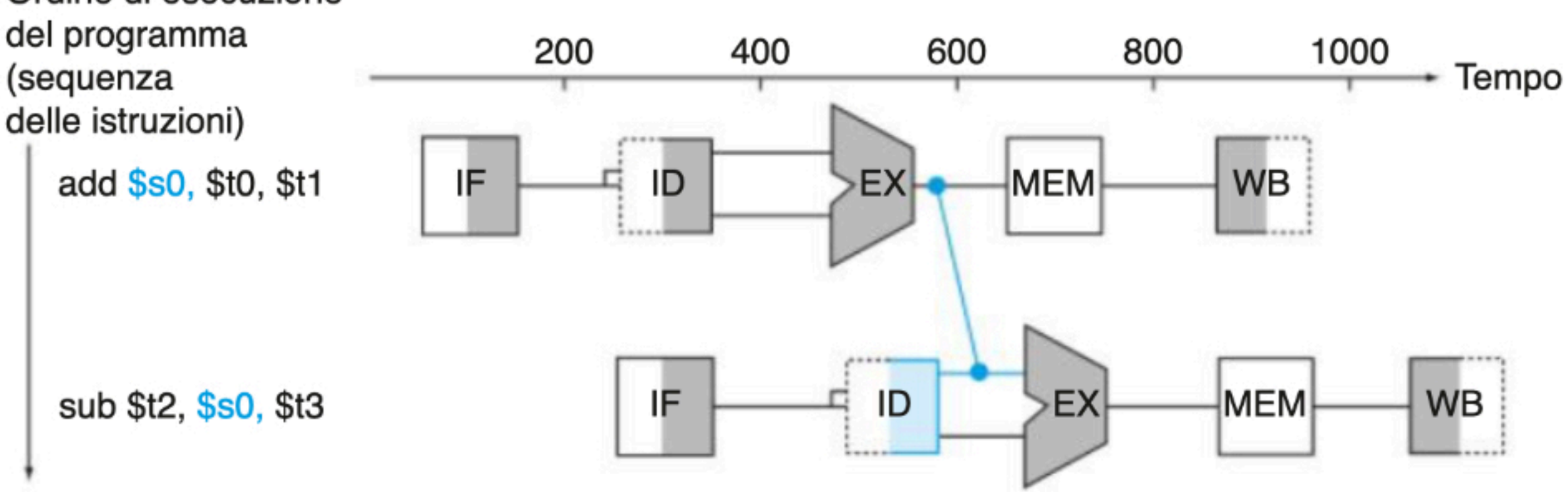
- Un'istruzione dipende dal completamento dell'accesso ai dati di un'istruzione precedente

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



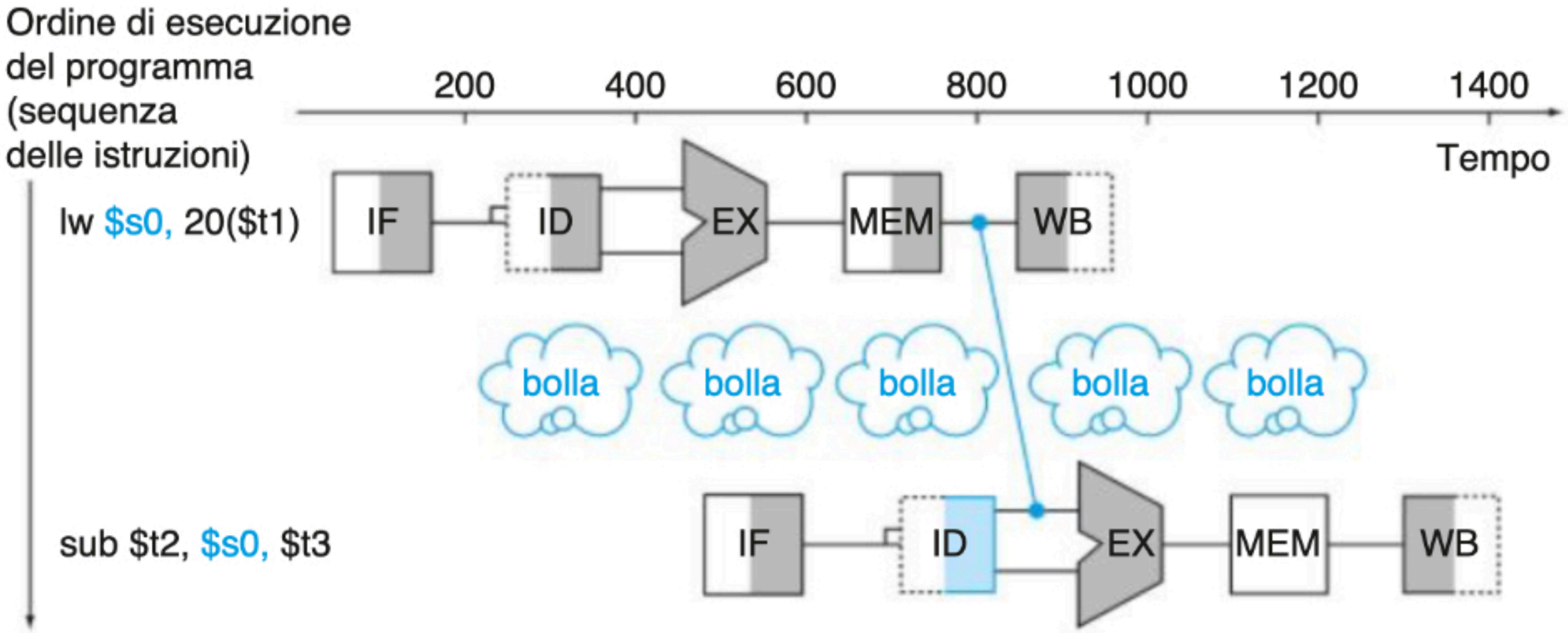
Propagazione (bypassing)

- Usare il risultato quando è stato calcolato
  - Non aspettare che sia memorizzato in un registro
  - Richiede connessioni extra nell'unità di elaborazione



Hazard sui dati di una load

- Non si possono evitare sempre gli stalli tramite propagazione
  - Se il valore non è ancora calcolato quando necessario
  - Non si può propagare indietro nel tempo!

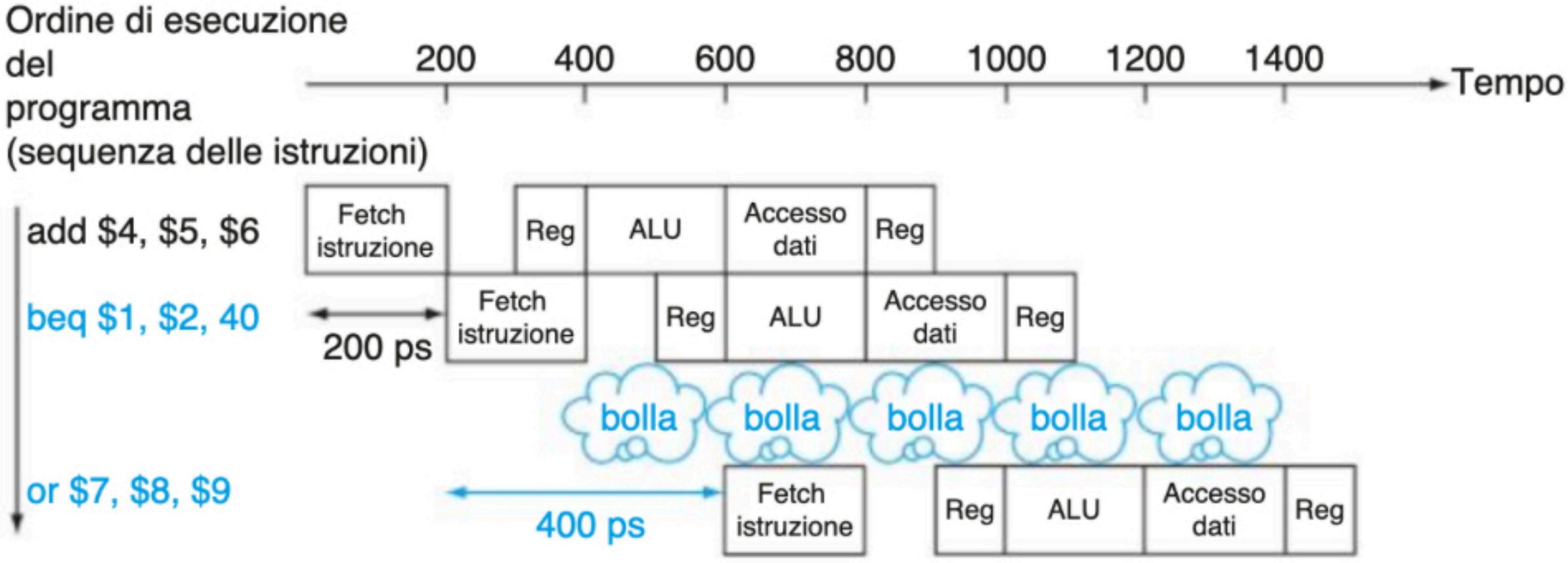


Hazard sul controllo

- Un salto determina il flusso del controllo
  - Il prelievo della prossima istruzione dipende dal risultato del salto
  - La pipeline non può prelevare sempre l'istruzione corretta
    - Se sta ancora elaborando lo stadio ID del salto
- Nella pipeline del MIPS
  - E' necessario confrontare i registri e calcolare la destinazione prima nella pipeline
  - Aggiungere hardware per farlo nello stadio ID

Stallo dovuto ai salti

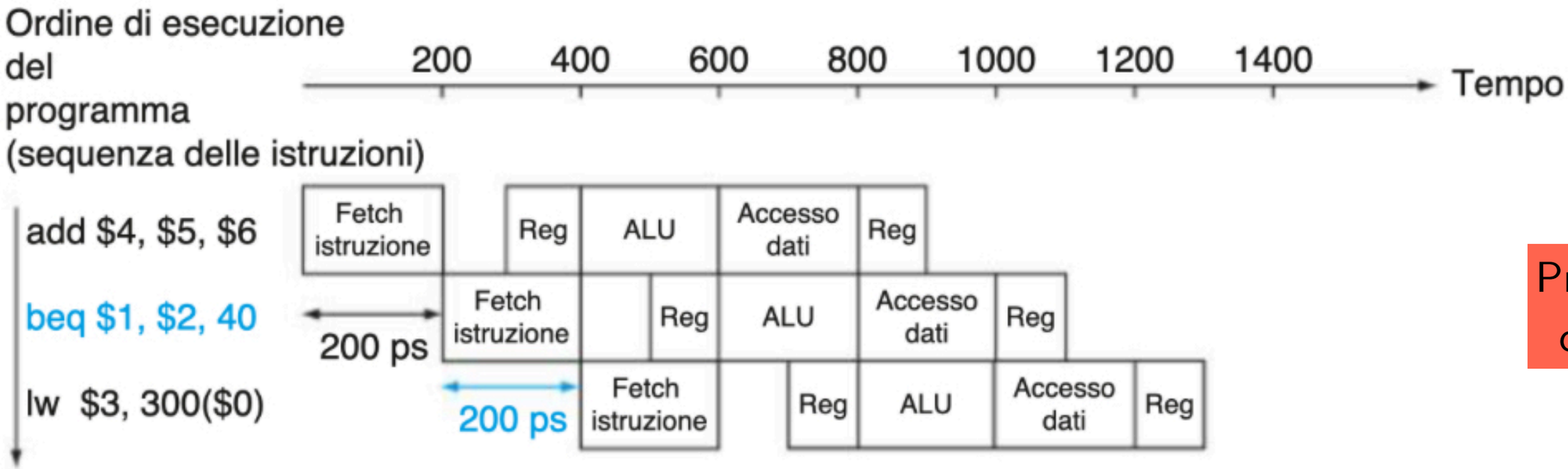
- Bisogna attendere finché il risultato del salto non è stato calcolato prima di prelevare la prossima istruzione



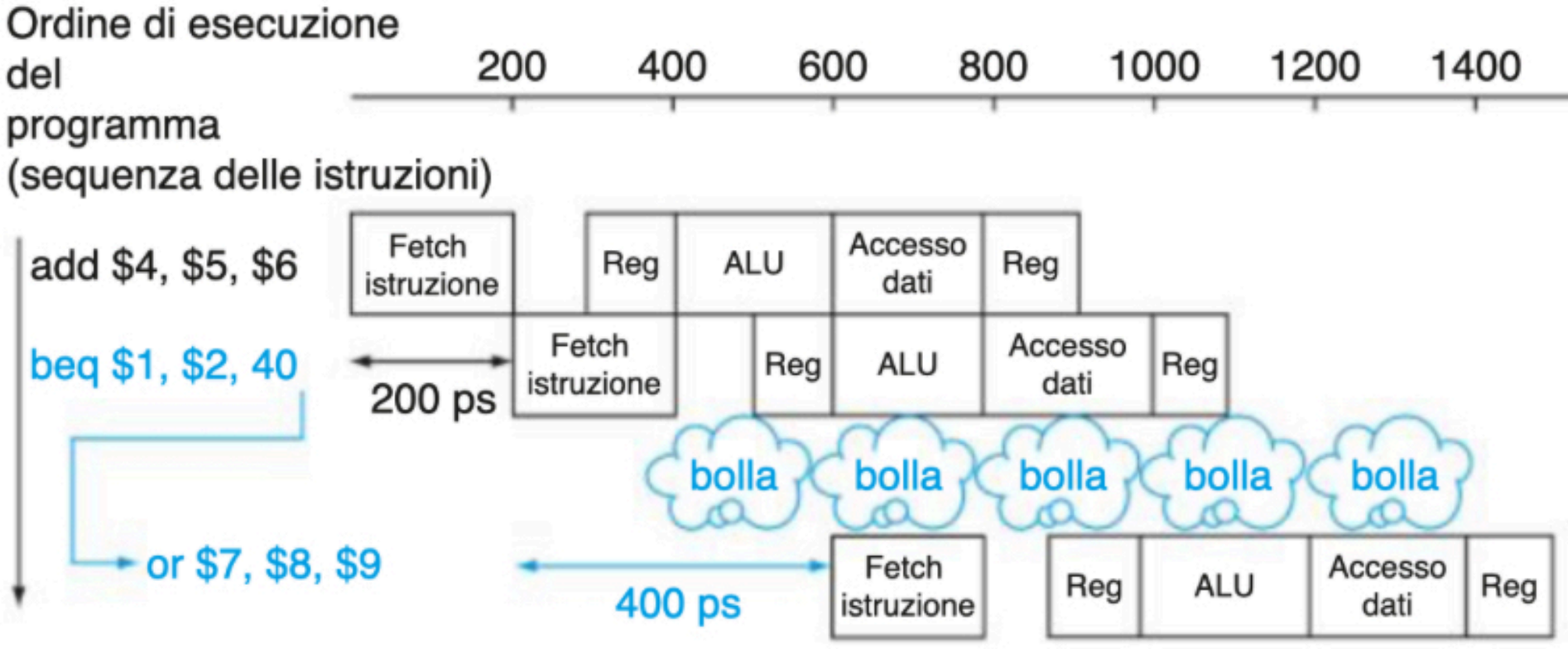
Predizione dei salti

- Le pipeline più lunghe non possono determinare per tempo il risultato di un salto
  - La penalità di stallo diventa inaccettabile
- Predizione del risultato di un salto
  - Si entra in stallo solo se la predizione è sbagliata
- Nella pipeline del MIPS
  - E' possibile predire i salti non effettuati
  - Prelievo dell'istruzione dopo il salto, senza ritardo

MIPS con predizione



Predizione corretta



Predizione Sbagliata

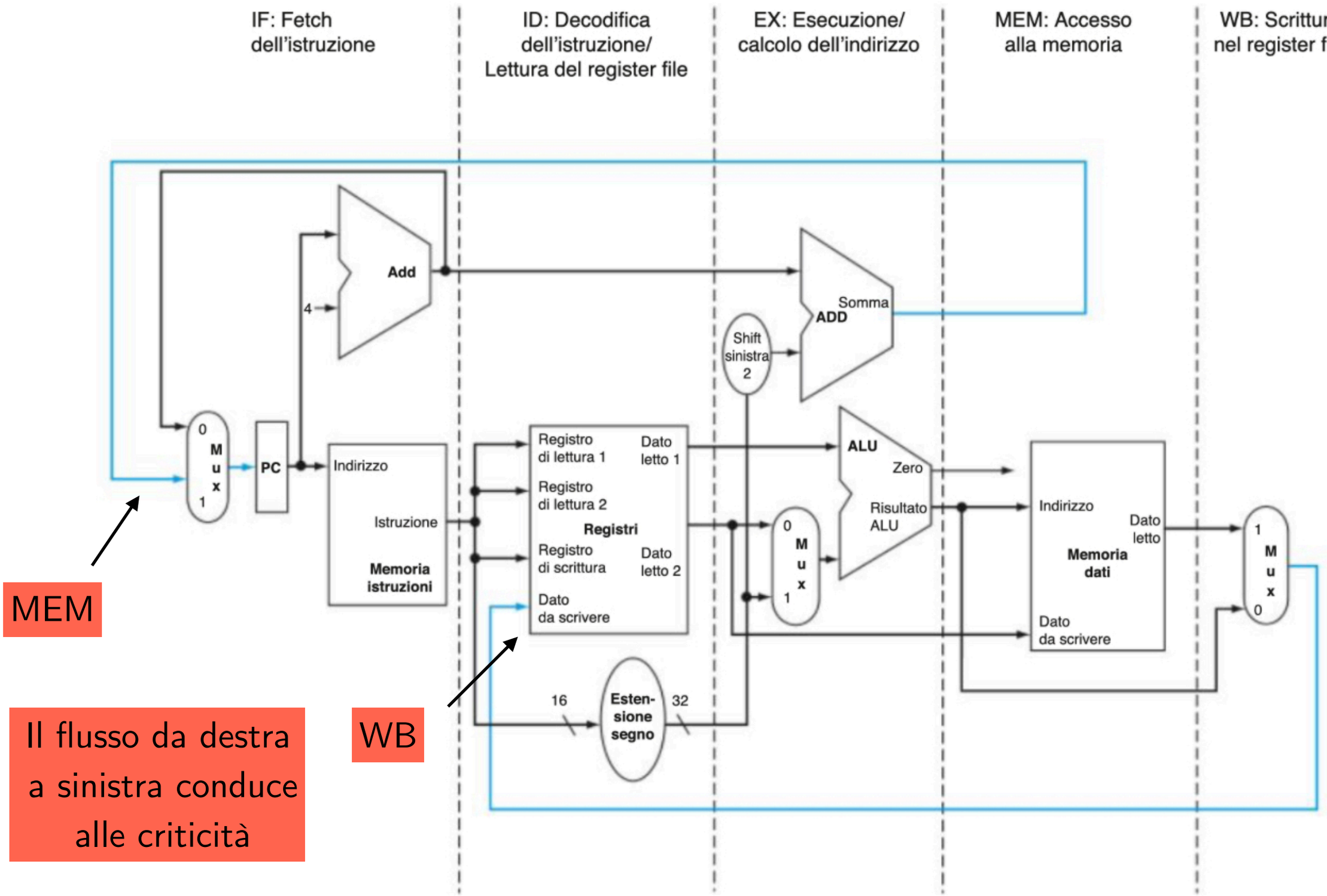
Predizione dei salti più realistica

- Predittore statico
  - Si basa sul comportamento tipico dei salti
  - Esempio: salti nei cicli e nei comandi condizionali
    - Si predice che i salti all'indietro sono scelti
    - Si predice che i salti in avanti non sono scelti
- Predittore dinamico
  - L'hardware misura il comportamento attuale del salto
    - Per esempio, registrando la storia recente di ogni salto
  - Assume che il comportamento futuro seguirà il trend
    - Quando si sbaglia, si entra in stallo con nuovo prelievo e si aggiorna la storia

Riepilogo sulla pipeline

- Il pipelining migliora le prestazioni incrementando il throughput delle istruzioni
  - Esegue molteplici istruzioni in parallelo
  - Ogni istruzione ha la stessa latenza
- Soggetta a hazard
  - Strutturali, sui dati, sul controllo
- Il progetto dell'insieme di istruzioni ha un impatto sulla complessità dell'implementazione della pipeline

Datapath con pipeline del MIPS



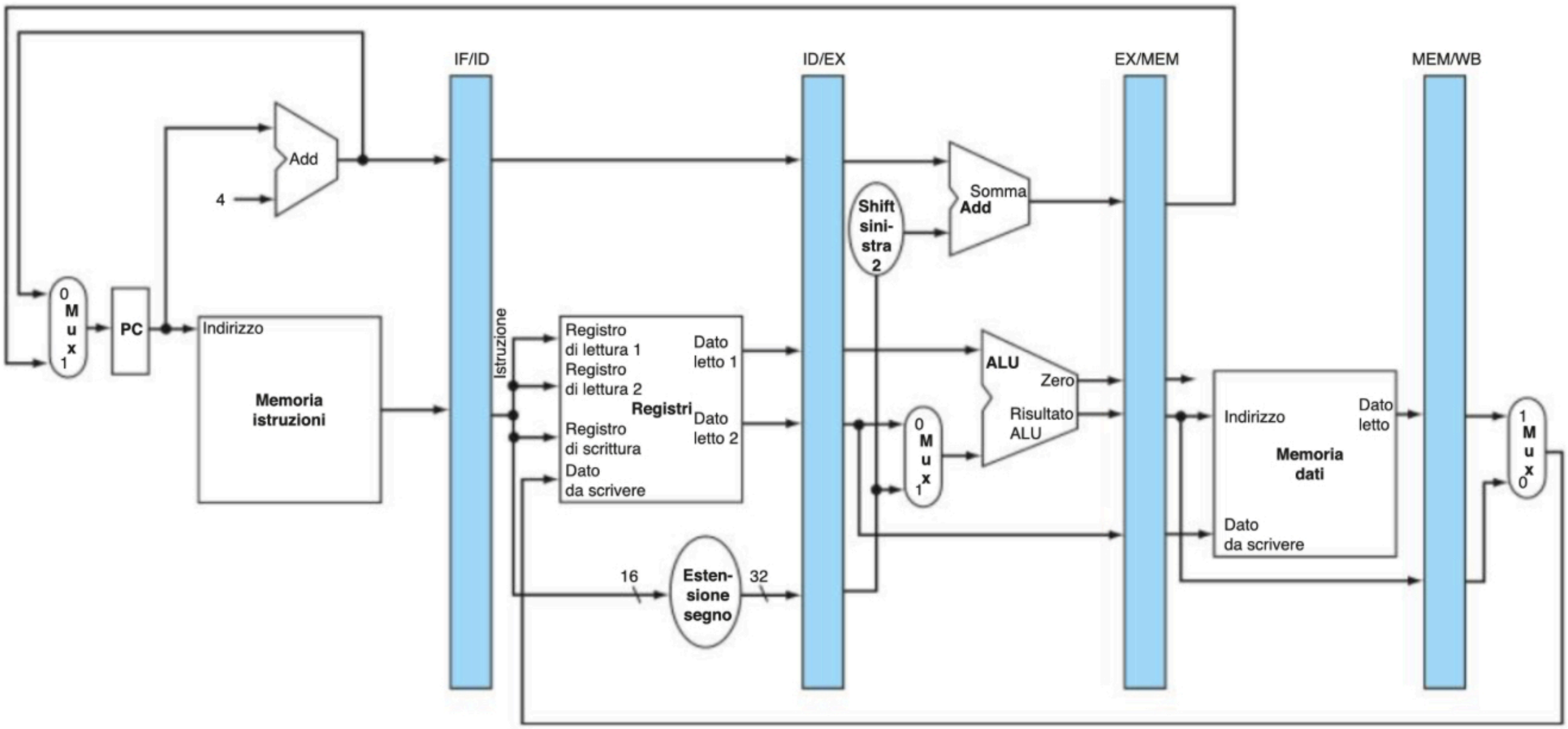
MEM

WB

Il flusso da destra a sinistra conduce alle criticità

Registri Pipeline

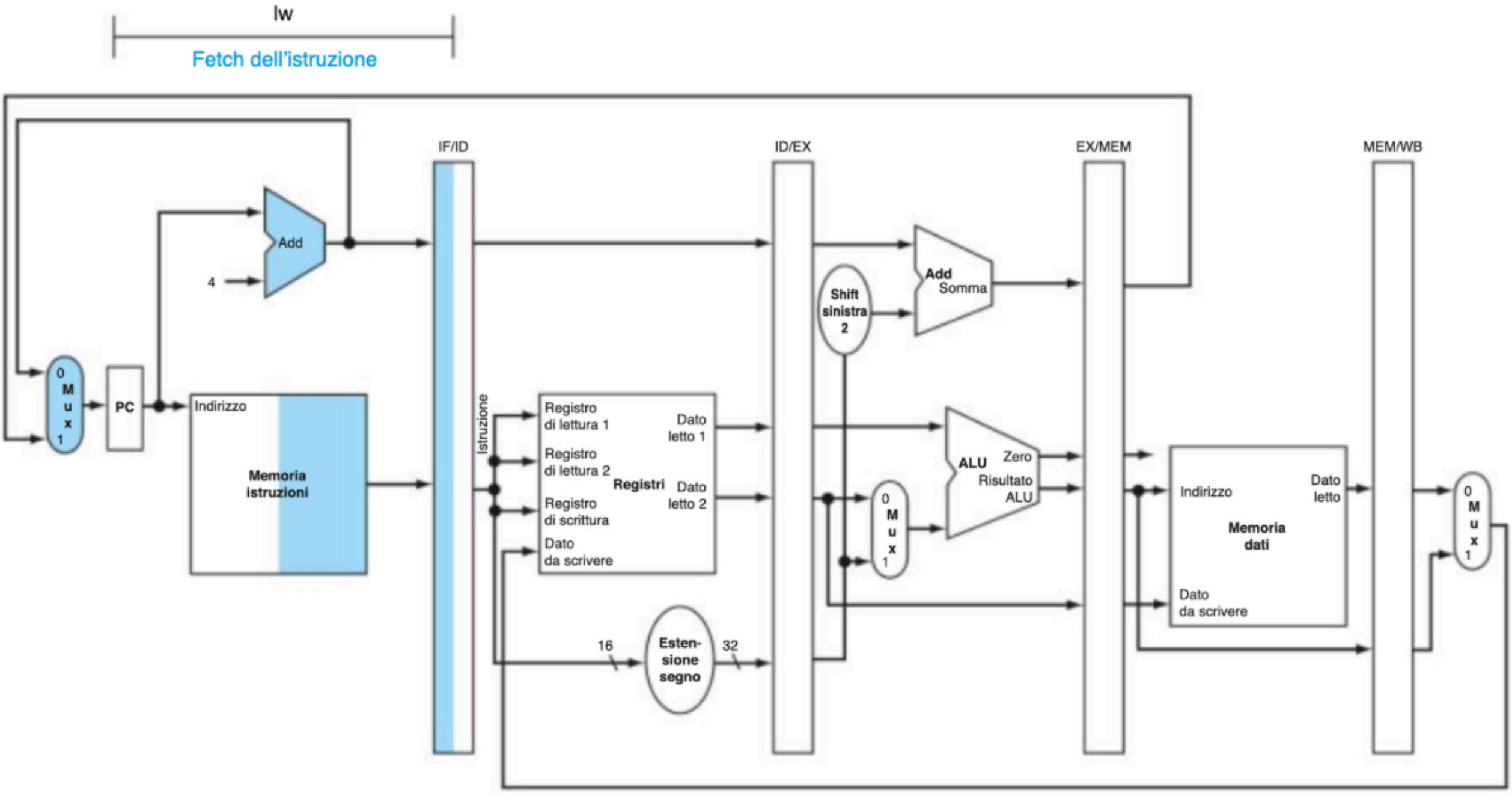
- Sono necessari dei registri tra i vari stadi
  - Per immagazzinare le informazioni prodotte nel ciclo precedente



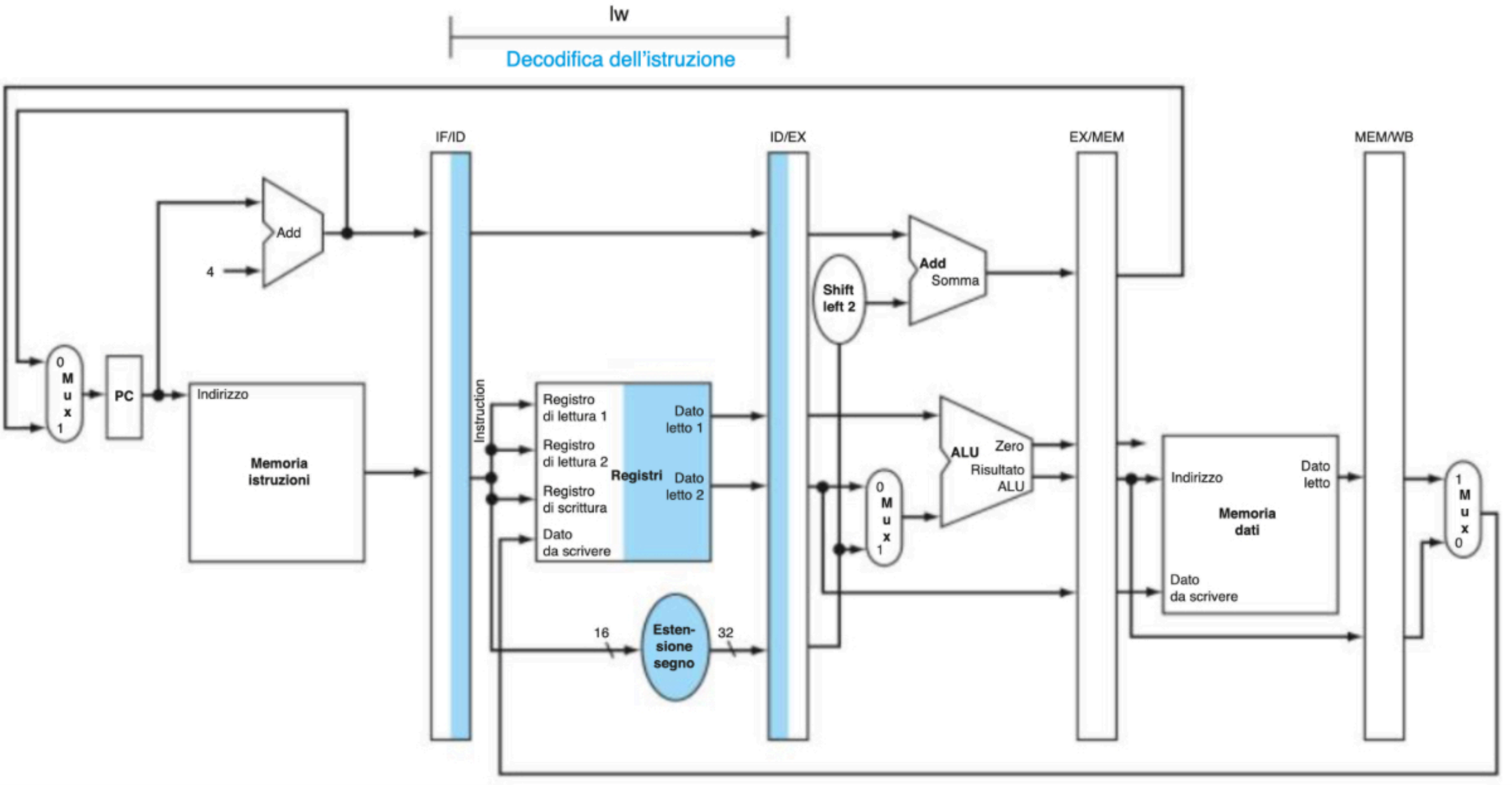
Funzionamento della Pipeline

- Flusso di istruzioni ciclo per ciclo attraverso l'unità di elaborazione in pipeline
  - Diagramma della pipeline "a singolo ciclo di clock"
    - Mostra l'uso della pipeline in un singolo ciclo
    - Evidenzia le risorse utilizzate
  - Diagramma della pipeline "a multiplo ciclo di clock"
    - Grafico dell'operazione nel tempo
- Studieremo i diagramma "a singolo ciclo di clock" per la load e la store

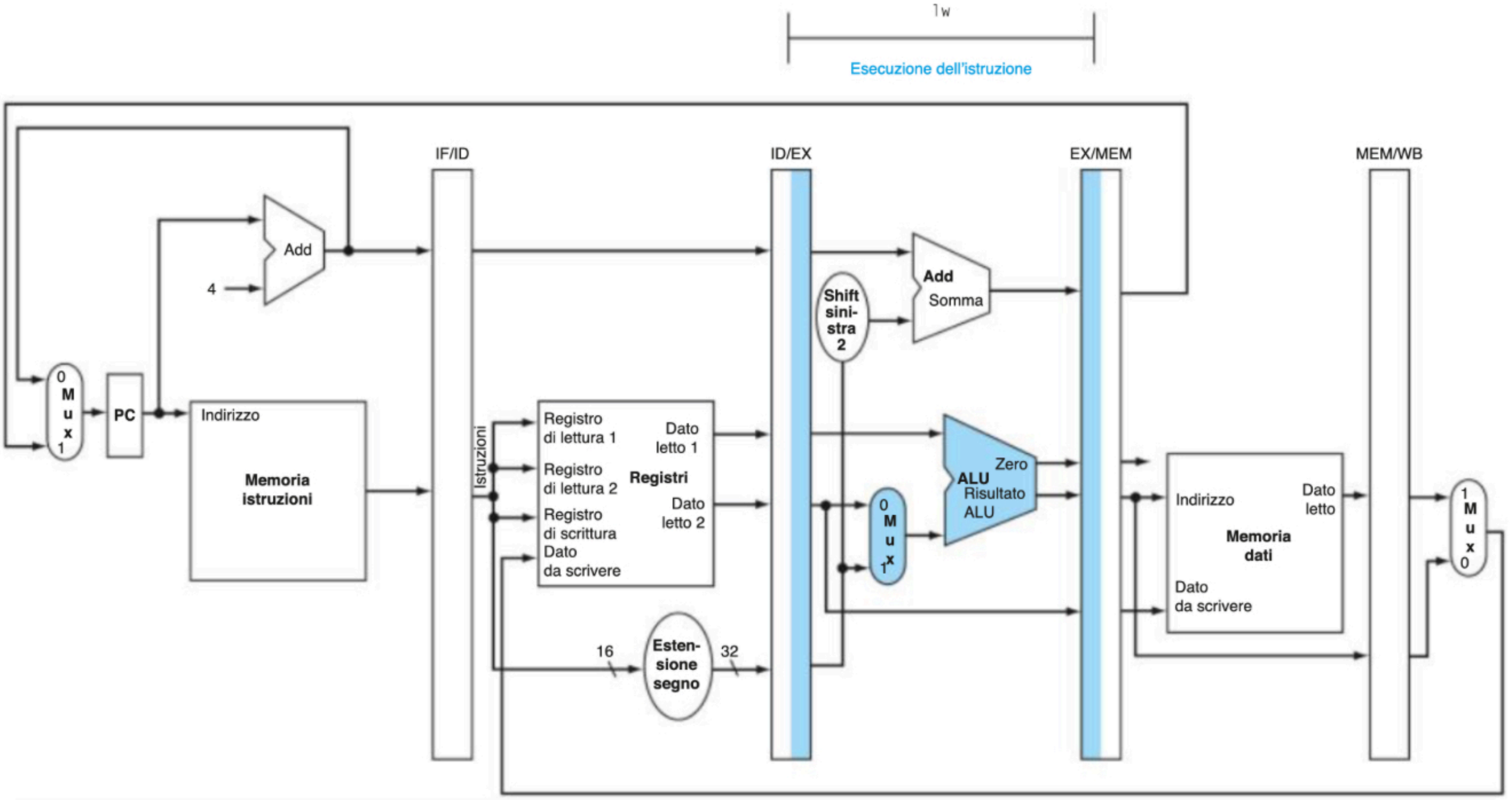
IF per Load, Store, ...



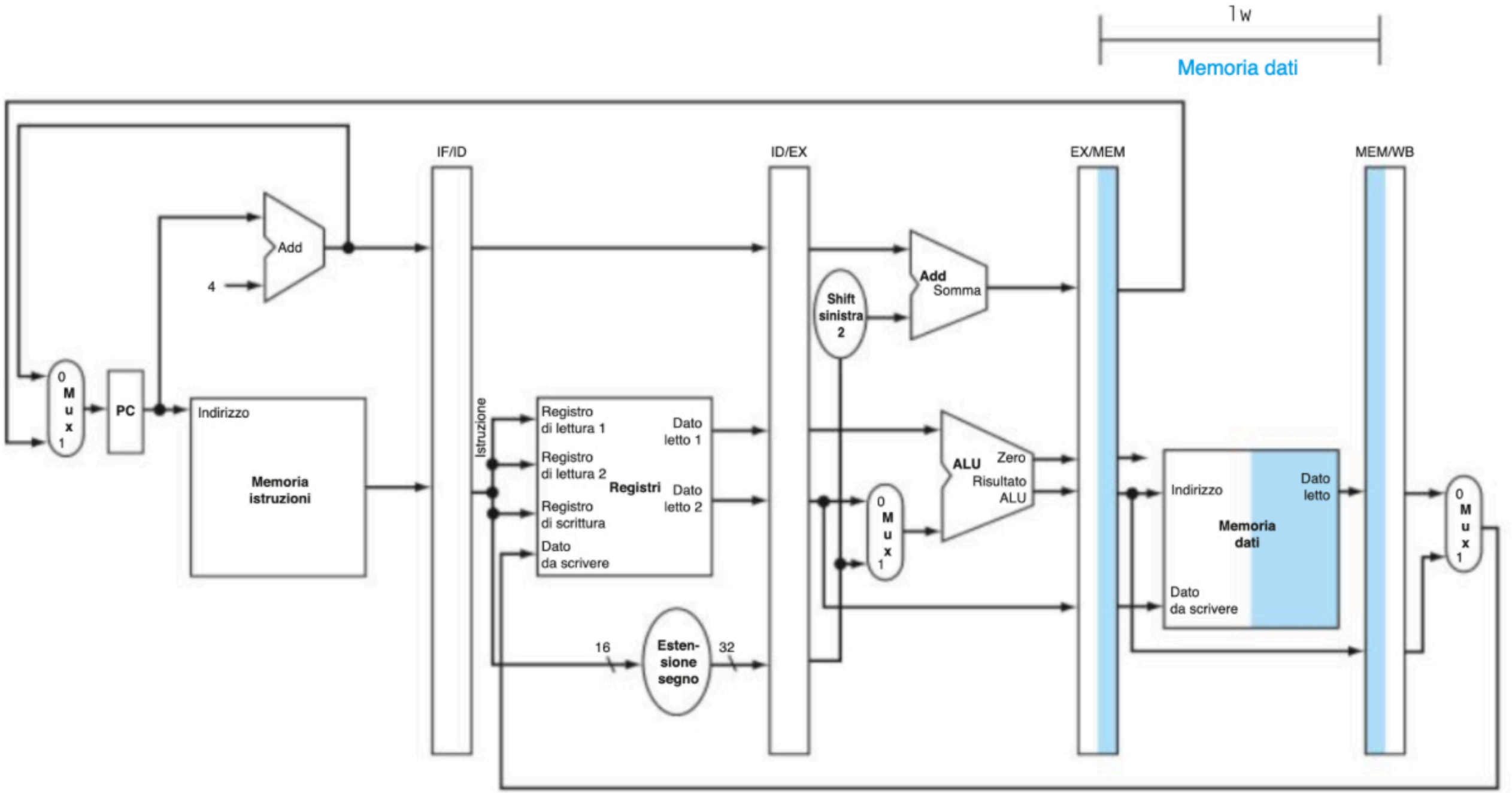
ID per Load, Store, ...



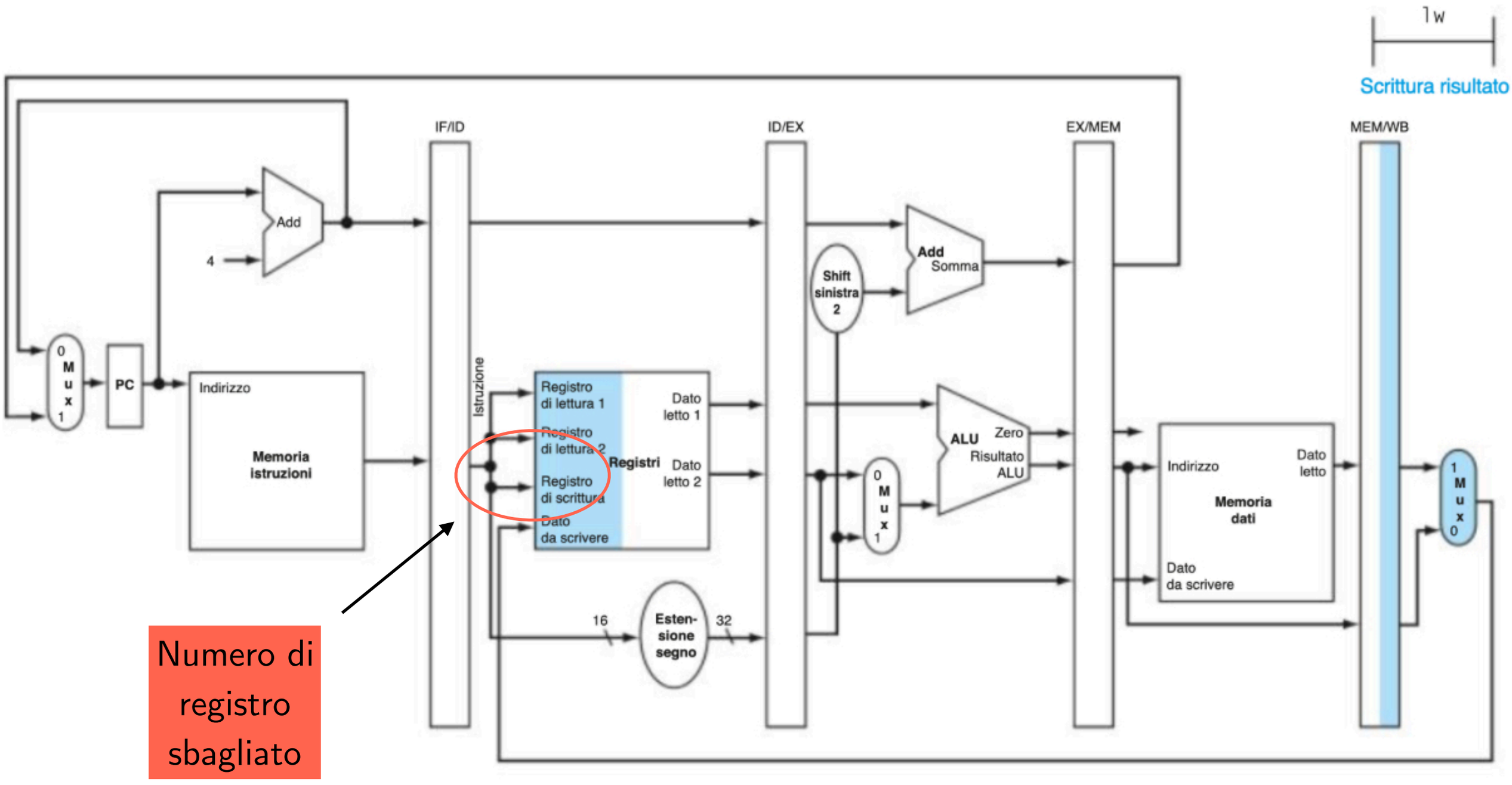
EX per Load


lw
Esecuzione dell'istruzione
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Indirizzo
Memoria istruzioni
Istruzioni
Registro di lettura 1
Registro di lettura 2
Registro di scrittura
Dato da scrivere
Registri
Dato letto 1
Dato letto 2
16
Estensione segno
32
Shift sinistra 2
Add
Somma
0
M
u
x
1
ALU
Zero
Risultato ALU
Indirizzo
Memoria dati
Dato letto
Dato da scrivere
1
M
u
x
0

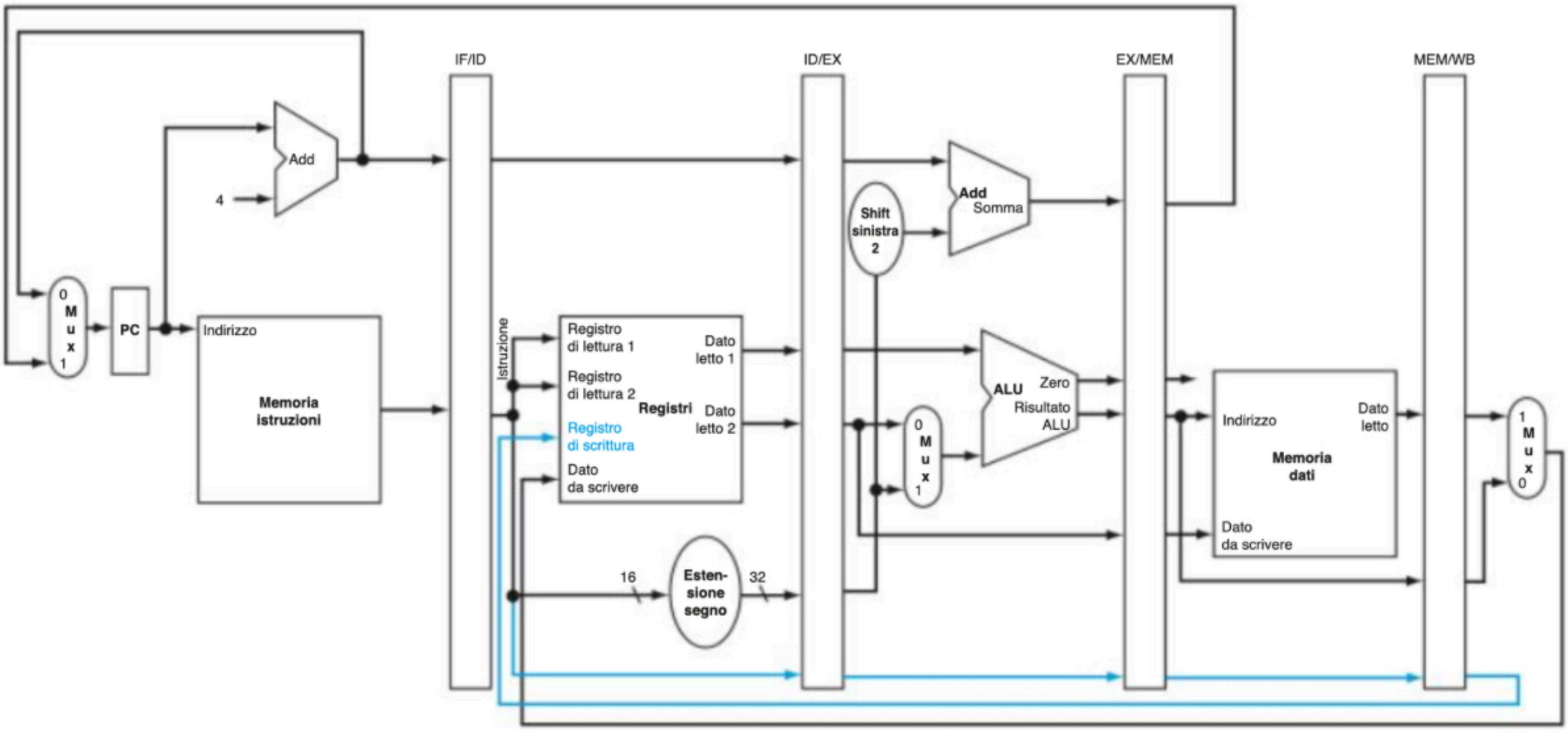
MEM per Load


lw
Memoria dati
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Indirizzo
Memoria
istruzioni
Istruzione
Registro
di lettura 1
Registro
di lettura 2
Registro
di scrittura
Dato
da scrivere
Registri
Dato
letto 1
Dato
letto 2
16
Esten-
sione
segno
32
Shift
sinistra
2
Add
Somma
ALU
Zero
Risultato
ALU
Indirizzo
Dato
letto
Memoria
dati
Dato
da scrivere

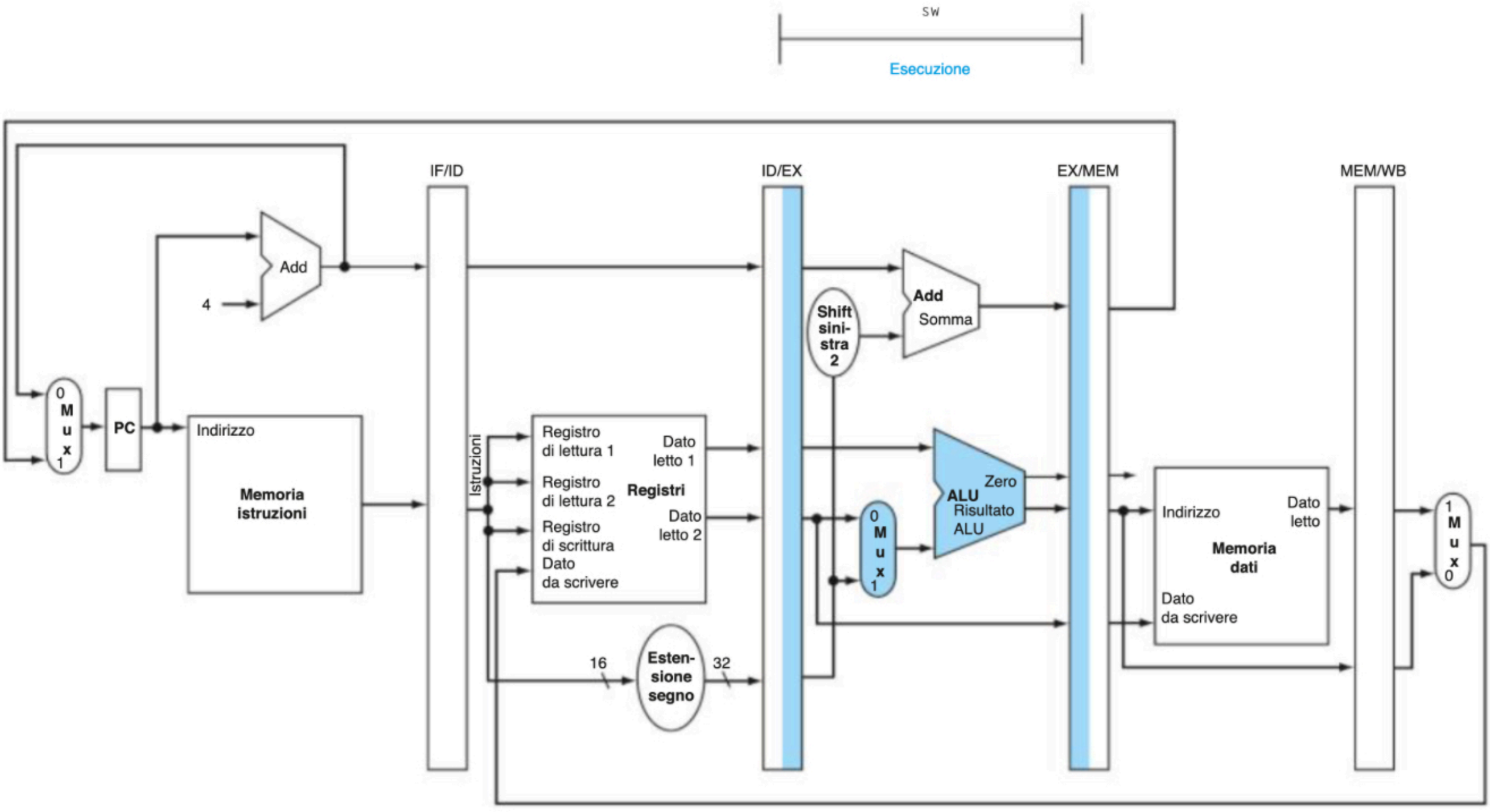
WB per Load


lw
Scrittura risultato
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Indirizzo
Memoria
istruzioni
Istruzione
Registro
di lettura 1
Registro
di lettura 2
Registro
di scrittura
Dato
da scrivere
Registri
Dato
letto 1
Dato
letto 2
16
Esten-
sione
segno
32
Shift
sinistra
2
Add
Somma
ALU
Zero
Risultato
ALU
Indirizzo
Dato
letto
Memoria
dati
Dato
da scrivere
1
M
u
x
0
Numero di registro sbagliato

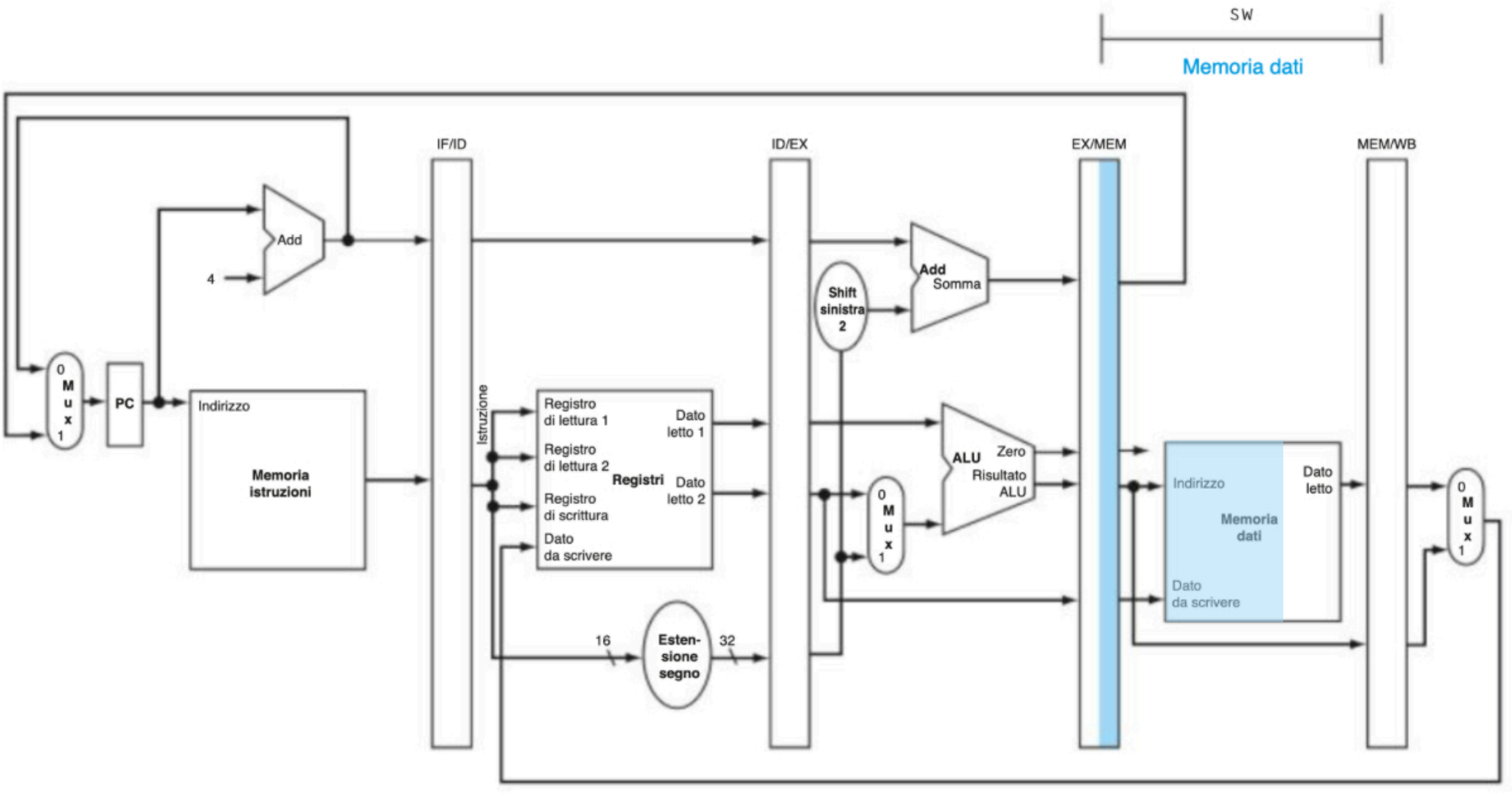
Datapath corretto per Load



EX per Store


SW
Esecuzione
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Indirizzo
Memoria
istruzioni
Istruzioni
Registro
di lettura 1
Registro
di lettura 2
Registro
di scrittura
Dato
da scrivere
Dato
letto 1
Registri
Dato
letto 2
16
Esten-
sione
segno
32
Shift
sini-
stra
2
Add
Somma
0
M
u
x
1
Zero
ALU
Risultato
ALU
Indirizzo
Dato
letto
Memoria
dati
Dato
da scrivere
1
M
u
x
0

MEM per Store



WB per Store

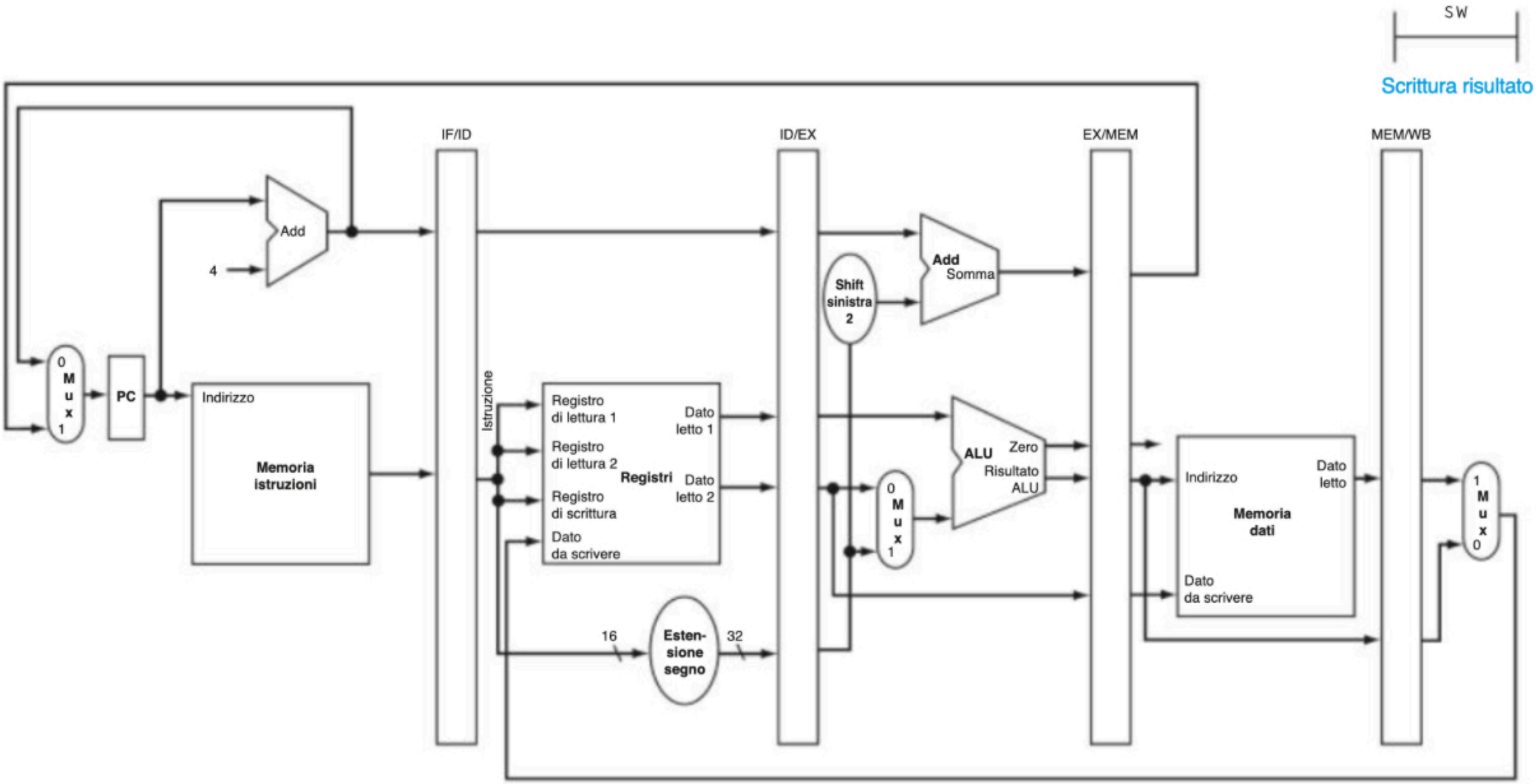

SW
Scrittura risultato
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Indirizzo
Memoria
istruzioni
Istruzione
Registro
di lettura 1
Registro
di lettura 2
Registri
Registro
di scrittura
Dato
da scrivere
Dato
letto 1
Dato
letto 2
16
Esten-
sione
segno
32
Shift
sinistra
2
Add
Somma
ALU
Zero
Risultato
ALU
Indirizzo
Memoria
dati
Dato
letto
Dato
da scrivere
1
M
u
x
0

Diagramma pipeline multi-ciclo

- Forma che mostra l'uso delle risorse

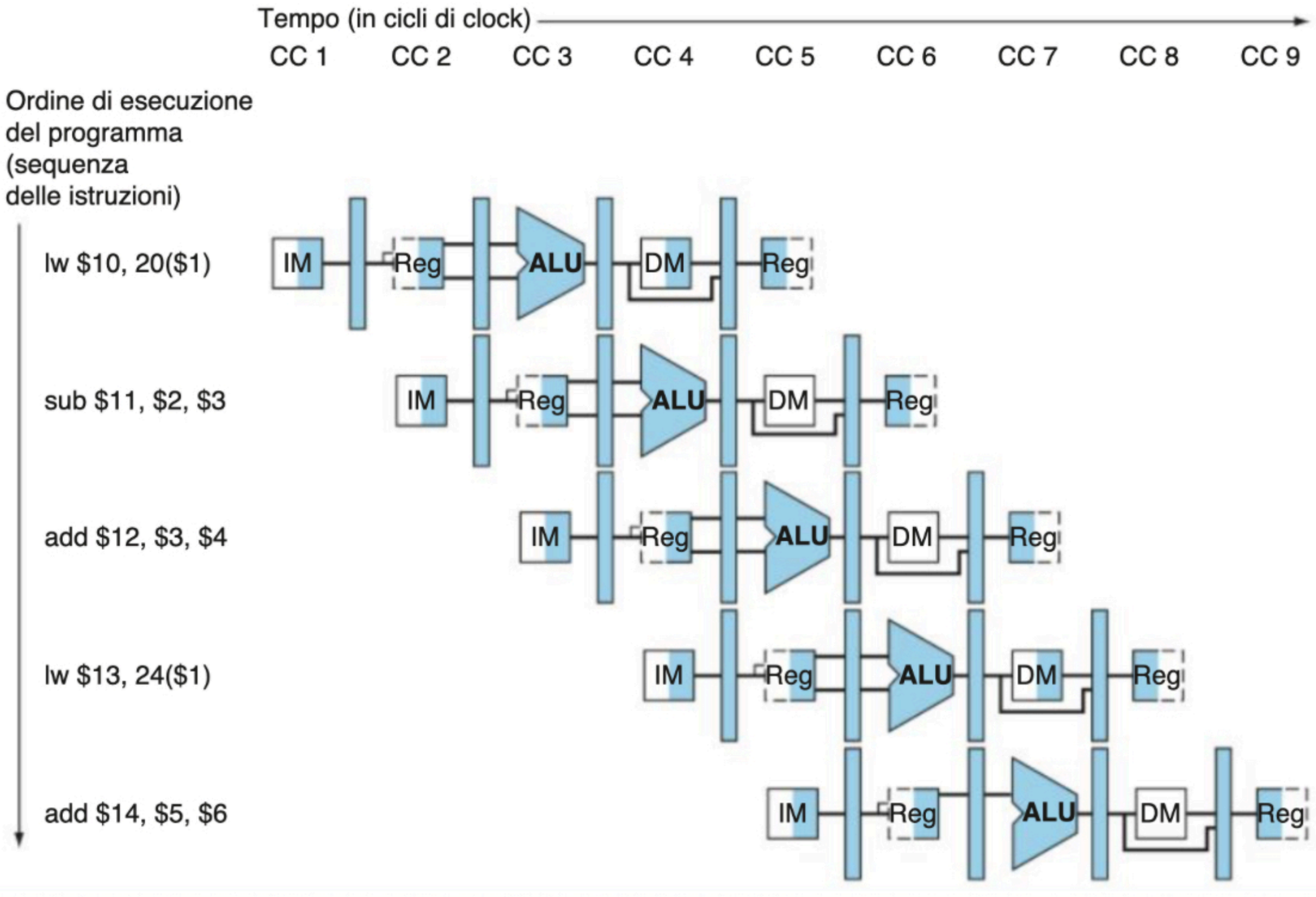


Diagramma pipeline multi-ciclo

- Forma tradizionale

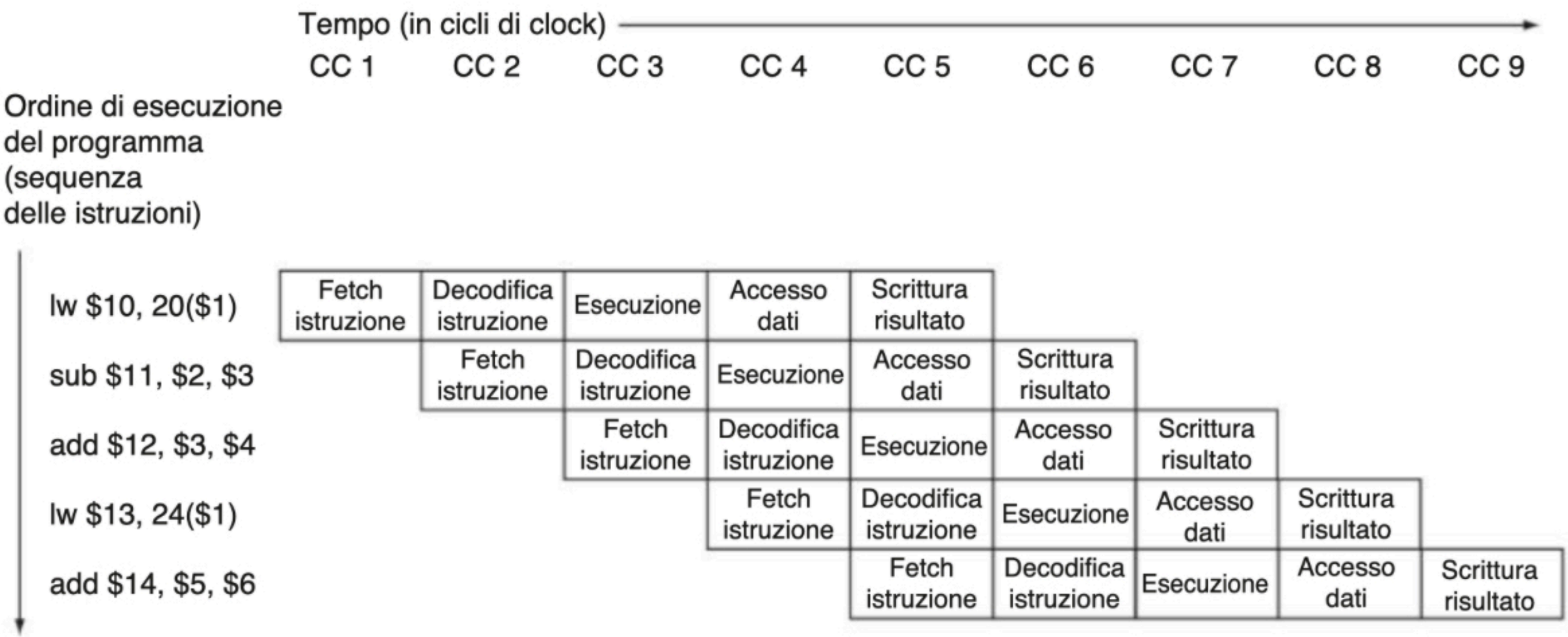
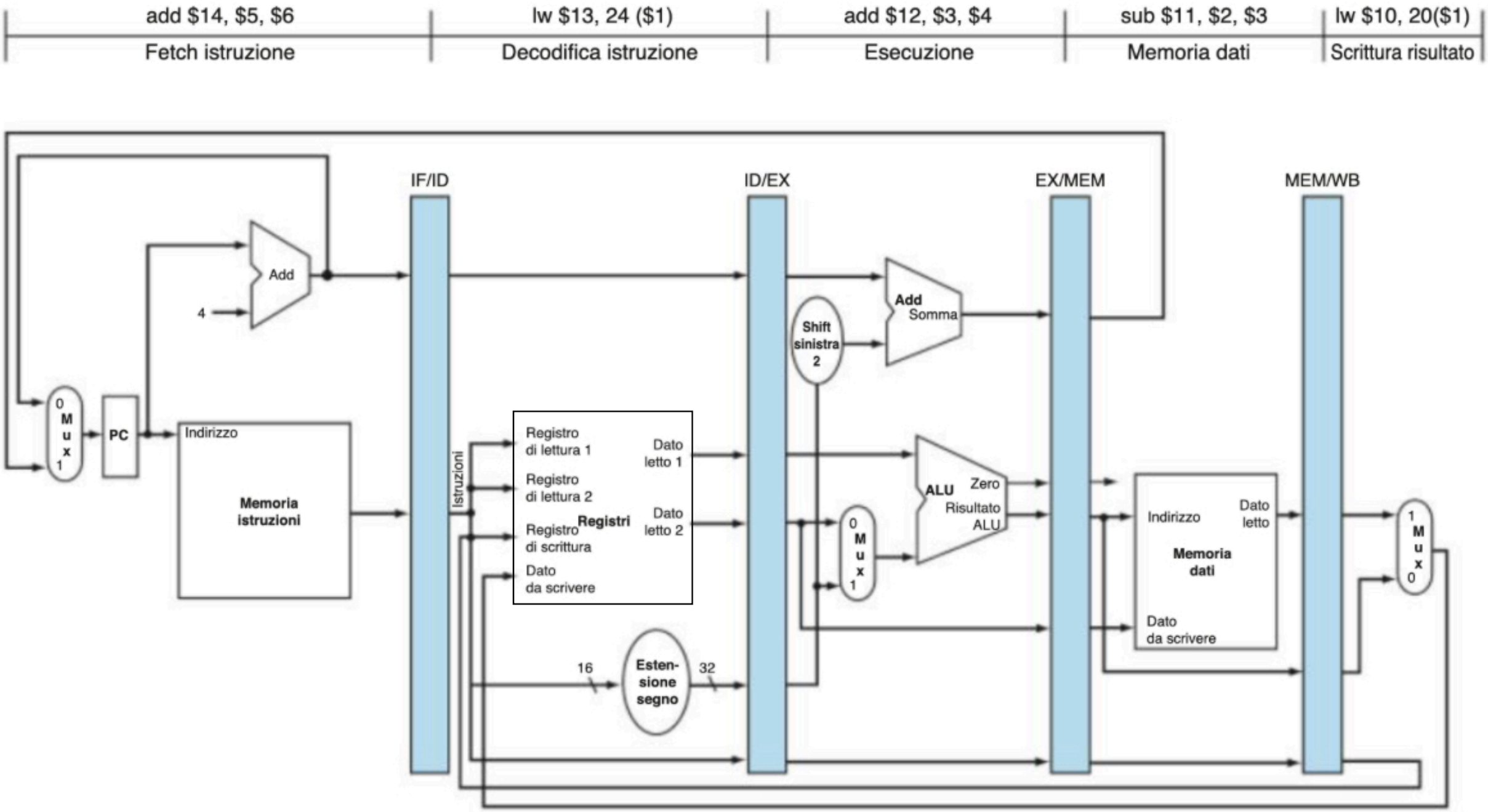
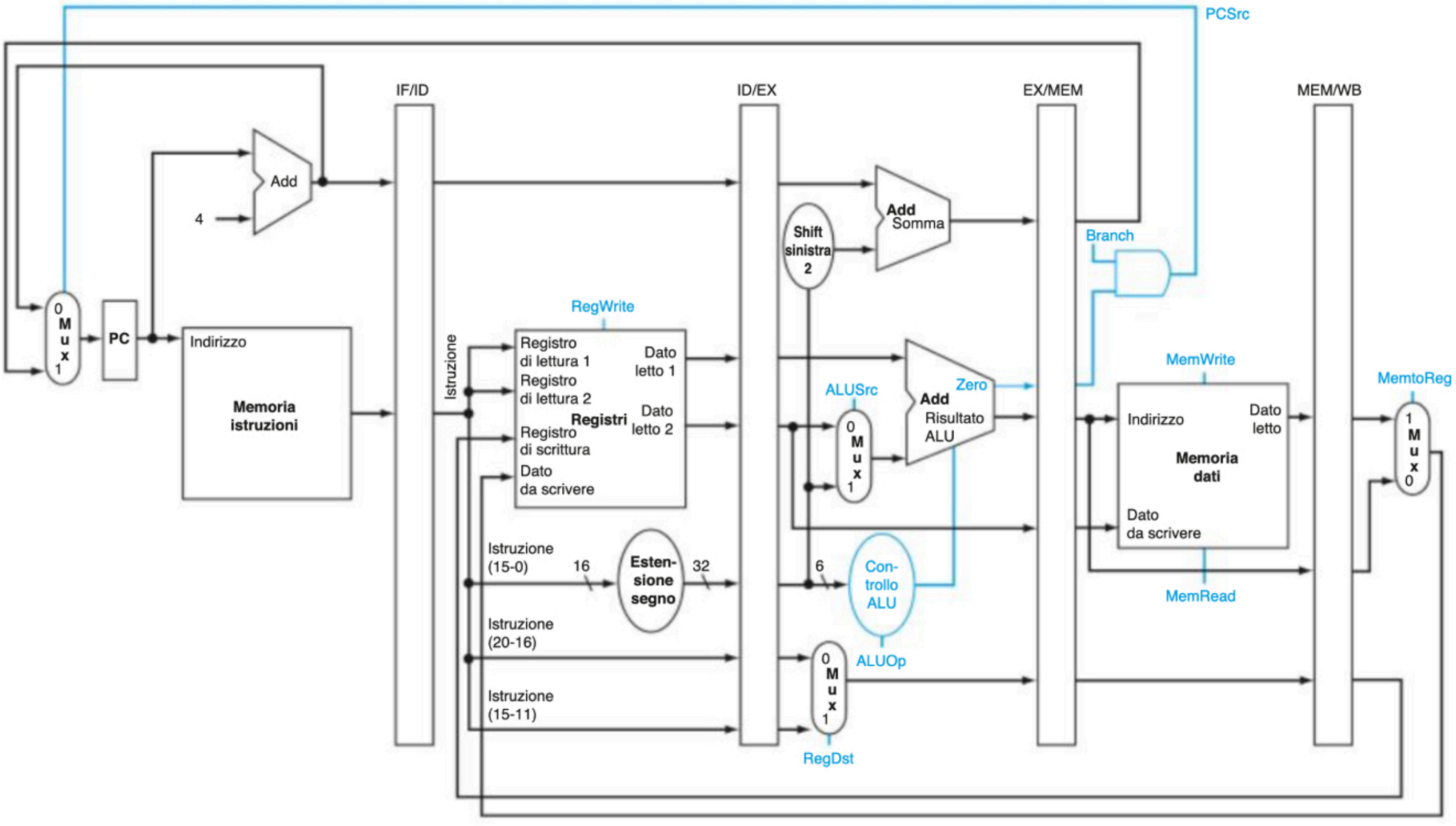


Diagramma pipeline a ciclo singolo

- Stato della pipeline in un dato ciclo

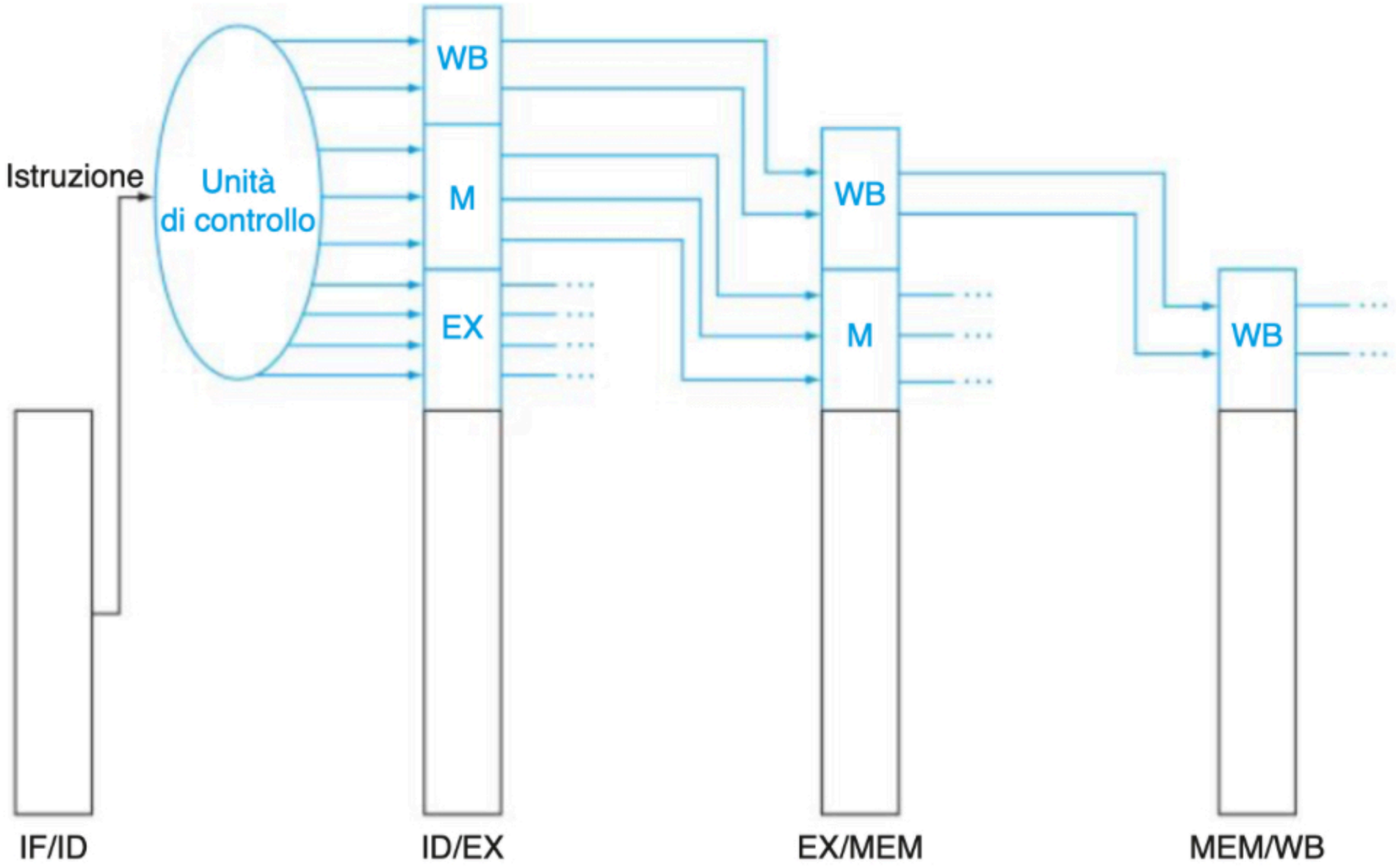


Controllo della pipeline (semplificato)

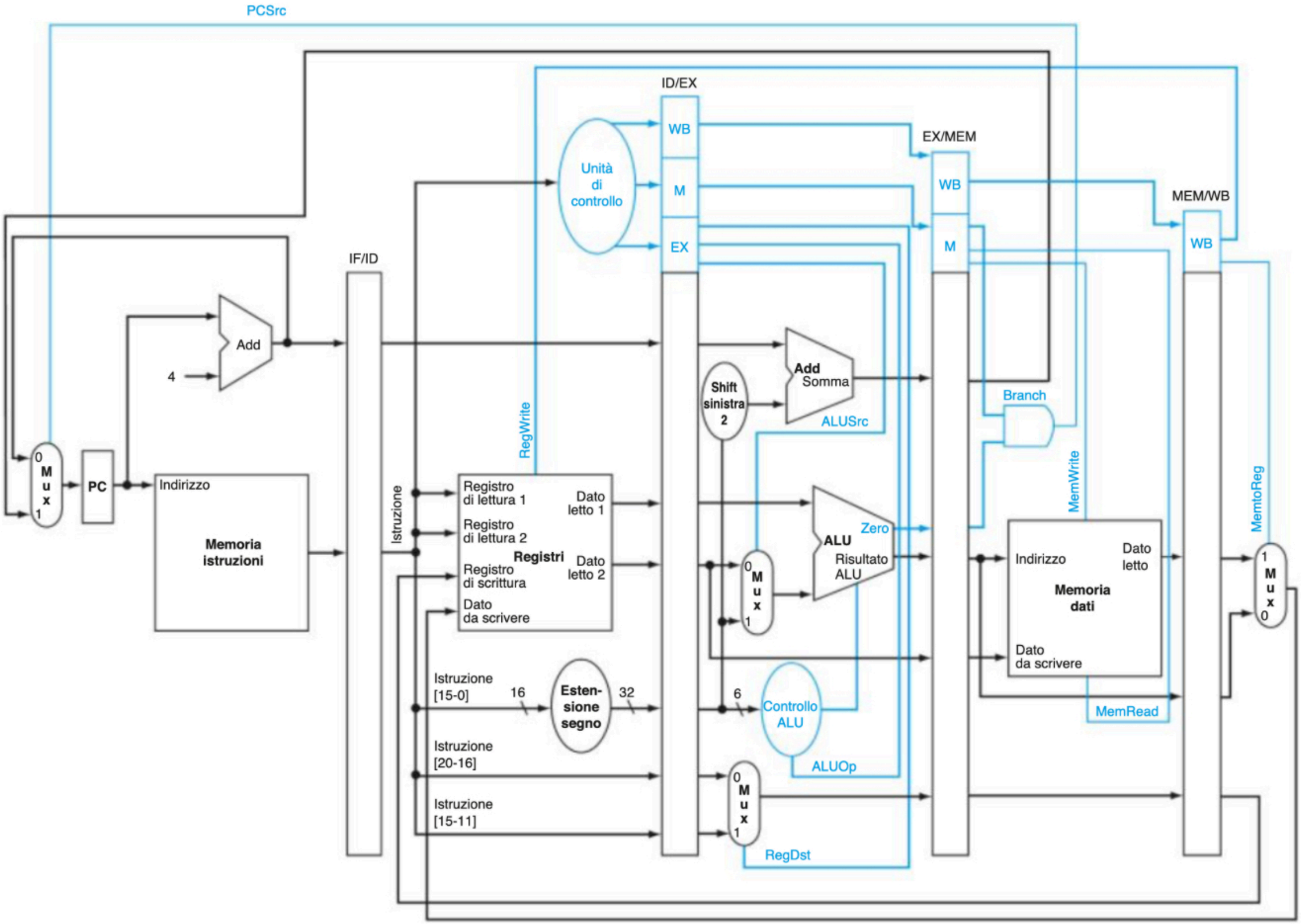


Controllo della pipeline

- I segnali di controllo sono derivati dall'istruzione
  - Come nell'implementazione a ciclo singolo



Controllo della pipeline

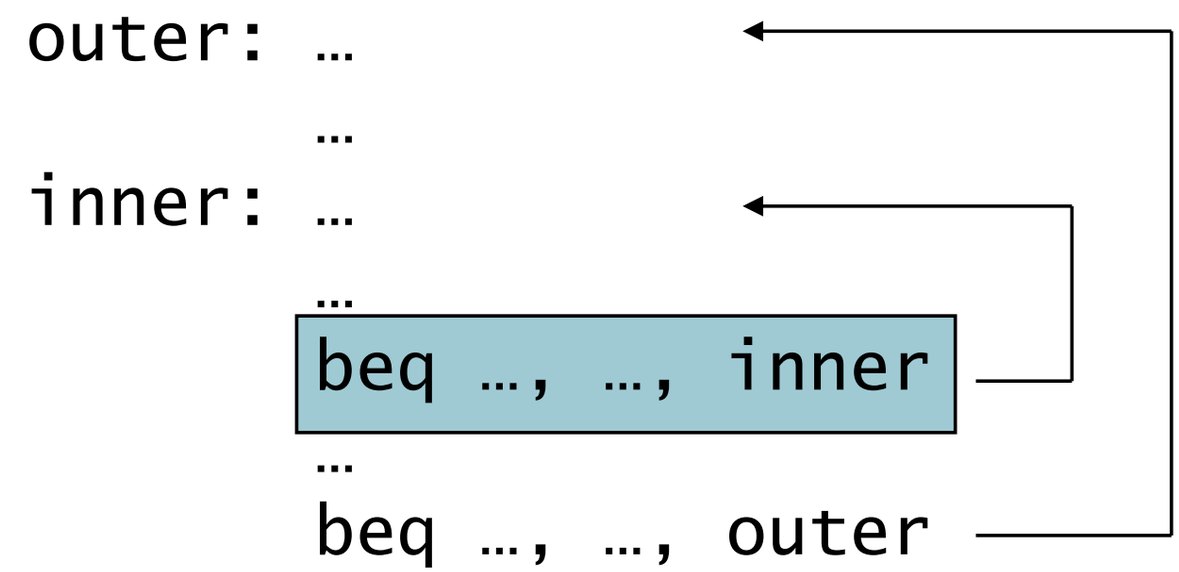


Predittore dinamico dei salti

- Nelle pipeline lunghe e superscalari, la penalità di salto è molto significativa
- Si usa la predizione dinamica
  - Buffer di predizione dei salti (aka tabella di storia dei salti)
  - Indicizzato dagli indirizzi delle istruzioni di salto più recenti
  - Memorizza i risultati (preso/non preso)
  - Per eseguire un salto
    - Controllare la tabella, aspettandosi lo stesso risultato
    - Iniziare il fetch con l'istruzione corrispondente al risultato predetto
    - Se la predizione risulta sbagliata, svuotare la pipeline e invertire la predizione

Predittore a 1 bit: limitazioni

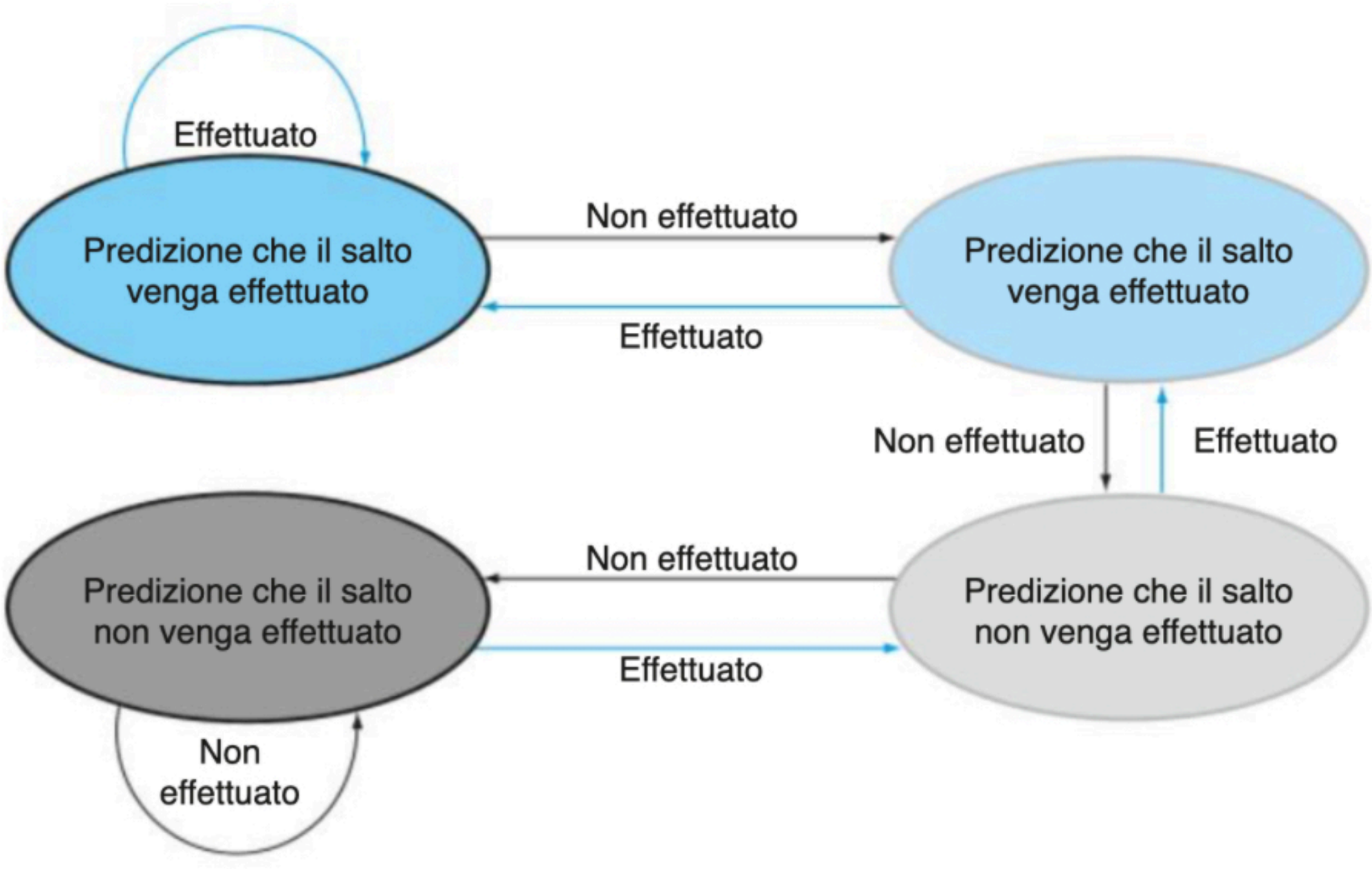
- I salti nei cicli interni sono predetti male due volte



- La predizione sbagliata "preso" avviene all'ultima iterazione del ciclo interno
- La predizione sbagliata "non preso" avviene alla prima iterazione del ciclo interno nella successiva iterazione del ciclo esterno

Predittore a 2 bit

- Si cambia la predizione solo dopo due predizioni sbagliate consecutive



Calcolare la destinazione di salto

- Anche col predittore, è sempre necessario calcolare l'indirizzo di destinazione
  - Penalità di 1 ciclo per un salto "preso"
- Buffer di predizione dei salti
  - Cache degli indirizzi di destinazione
  - Indicizzato da PC quando l'istruzione è prelevata
    - In caso di hit e di predizione di salto "preso", si può prelevare subito l'istruzione di destinazione

Eccezioni e interruzioni

- Eventi "inattesi" richiedono un cambiamento nel flusso di controllo
  - ISA diverse usano termini diversi
- Eccezione
  - Sollevata all'interno della CPU
    - Per esempio, opcode non definito, overflow, chiamata di sistema
- Interruzione
  - Proviene da un controllore I/O esterno
- Gestirle senza sacrificare le prestazioni è difficile

Gestione delle eccezioni

- Nel MIPS le eccezioni sono gestite da coprocessore di controllo di sistema (CP0)
- Salvare PC dell'istruzione che ha sollevato l'eccezione (o interrotta)
  - Nel MIPS: program counter delle eccezioni (EPC)
- Salvare la causa del problema
  - Nel MIPS: registro `causa`
  - Assumeremo sia di 1 bit
    - 0 per opcode non definito, 1 per overflow
- Salto all'handler all'indirizzo 8000 0180

Un meccanismo alternativo

- Interrupt vettorizzati
  - L'indirizzo dell'handler è determinato dalla causa
- Esempio:
  - opcode non definito: `C000 0000`
  - overflow: `C000 0020`
  - ...: `C000 0040`
- Le istruzioni possono:
  - Gestire l'interruzione
  - Saltare all'handler vero e proprio

Azioni dell'handler

- Leggere la causa e trasferire il controllo all'handler competente
- Determinare l'azione necessaria
- Se recuperabile
  - Eseguire le azioni correttive
  - Usare EPC per ritornare al programma
- Altrimenti
  - Terminare il programma
  - Riportare l'errore usano EPC, `causa`, ...

Eccezioni nella pipeline

- Un'altra forma di hazard sul controllo
- Si consideri un overflow sulla somma nello stadio EX

`add $1, $2, $1`

  - Evitare che `$1` sia "sporcato"
  - Completare le istruzioni precedenti
  - Scartare la `add` e le istruzioni successive
  - Impostare i valori dei registri EPC e causa
  - Trasferire il controllo all'handler
- Simile alla predizione sbagliata dei salti
  - Usa molto dello stesso hardware

Proprietà di un'eccezione

- Eccezioni recuperabili
  - La pipeline può scartare l'istruzione
  - L'handler va in esecuzione, quindi ritorna all'istruzioni
    - Ri-prelevata ed eseguita da capo
- PC è salvato nel registro EPC
  - Identifica l'istruzione che ha causato l'eccezione
  - In realtà, si memorizza PC + 4
    - L'handler deve aggiustare il valore

Esempio di eccezione

- Eccezione sulla add in

```
40  sub  $11, $2, $4
44  and  $12, $2, $5
48  or   $13, $2, $6
4C  add  $1,  $2, $1
50  slt  $15, $6, $7
54  lw   $16, 50($7)
...
```

- Gestore

```
80000180  sw   $25, 1000($0)
80000184  sw   $26, 1004($0)
...
```

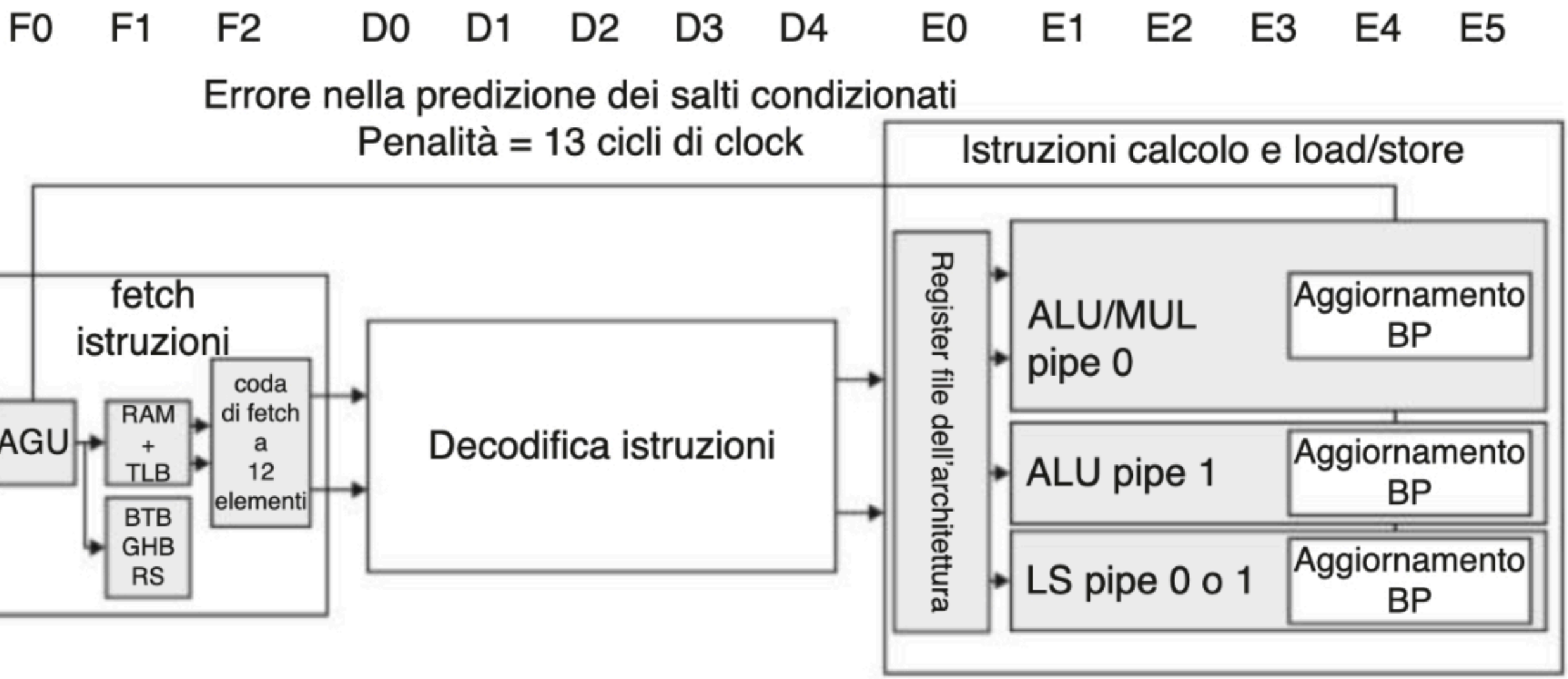
Eccezioni multiple

- Il pipelining sovrappone molteplici istruzioni
  - Si possono verificare molteplici eccezioni in una volta
- Approccio semplice: gestire l'eccezione della prima istruzione
  - Scartare le istruzioni successive
  - Eccezioni "precise"
- Nelle pipeline complesse
  - Molteplici istruzioni per ciclo
  - Completamento fuori ordine
  - Mantenere eccezioni precise è difficile!

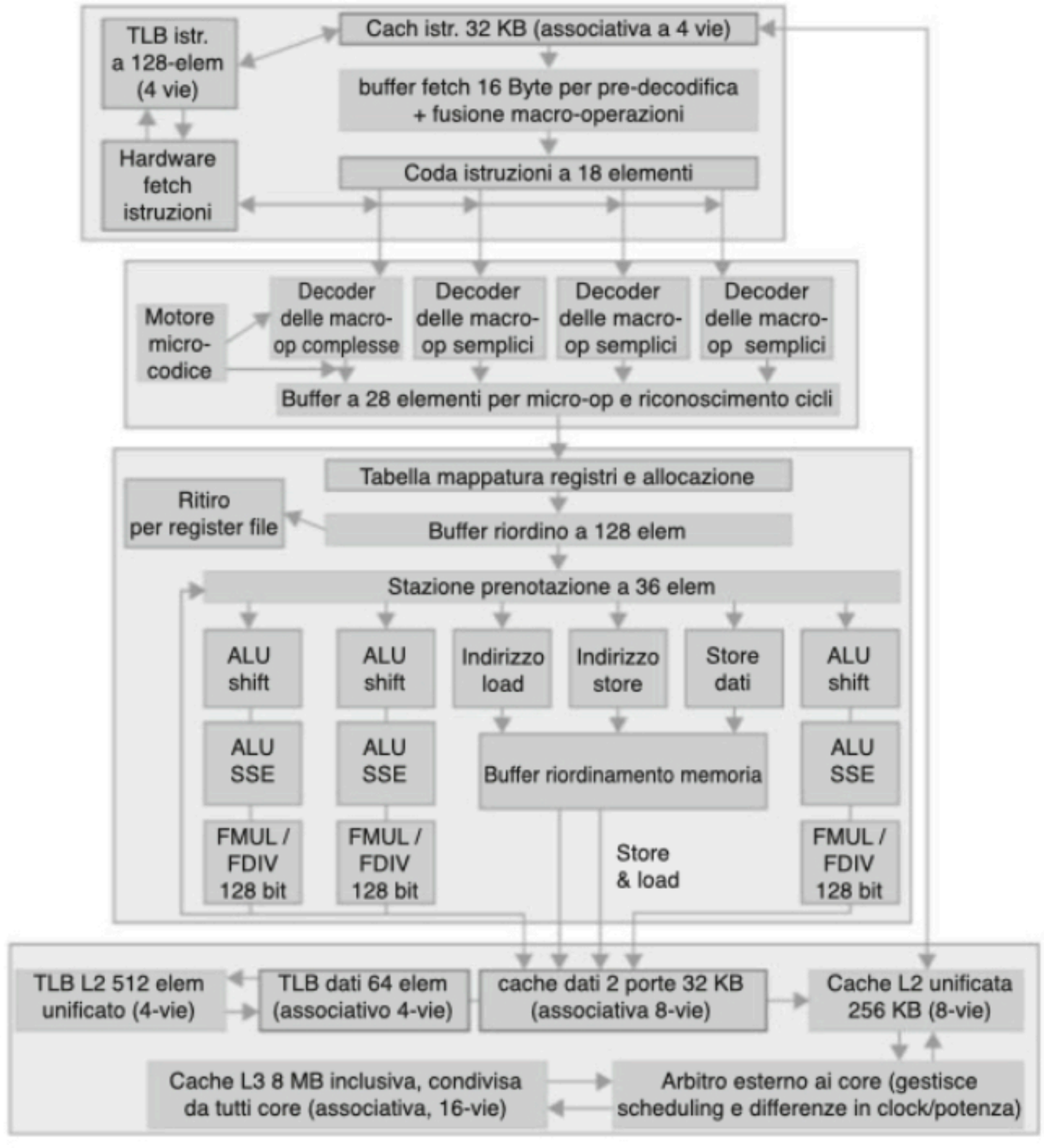
Cortex A8 e Intel i7

Processore	ARM A8	Intel Core i7 920
Mercato	Dispositivi Mobili Personali	Server, Cloud
Potenza termica di progetto	2 watt	130 watt
Frequenza di clock	1 GHz	2,66 GHz
Core/chip	1	4
Virgola mobile?	No	Sì
Parallelizzazione dell'esecuzione?	Dinamica	Dinamica
Istruzioni/ciclo di clock (picco)	2	4
Stadi della pipeline	14	14
Scheduling della pipeline	Statico in ordine	Dinamico fuori ordine, con speculazione
Predizione dei salti	a 2 livelli	a 2 livelli
Cache 1° livello/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
Cache 2° livello/core	128-1024 KiB	256 KiB
Cache 3° livello (codivisa)	–	2-8 MiB

Arm Cortex A8 Pipeline



Core i7 Pipeline



Errori

- Il pipelining è semplice (!)
  - L'idea di base è semplice
  - Il diavolo sta nei dettagli
    - Per esempio, identificare gli hazard sui dati
- Il pipelining è indipendente dalla tecnologia
  - Allora perché non abbiamo sempre usato il pipelining?
  - Più transistor permettono tecniche più avanzate
  - La progettazione di ISA con pipeline deve tenere conto dei trend tecnologici

Trabocchetti

- Una progettazione scadente dell'ISA può rendere il pipelining più difficile
  - Per esempio, insiemi di istruzioni complesse (VAX, IA-32)
    - Overhead significativo per far funzionare il pipelining
    - Approccio microistruzioni IA-32
  - Per esempio, modi di indirizzamento complessi
    - Indirezione della memoria, effetti collaterali sull'aggiornamento dei registri
  - Per esempio, salti ritardati
    - Le pipeline avanzate hanno lunghi slot di ritardo

Note conclusive

- L'ISA influenza il progetto dell'unità di elaborazione e del controllo
- L'unità di elaborazione e il controllo influenzano il progetto dell'ISA
- Il pipelining migliora il throughput delle istruzioni usando il parallelismo
- Più istruzioni completate al secondo
- La latenza di ogni istruzione non è ridotta
- Hazard: strutturali, sui dati, sul controllo

Capitolo 5

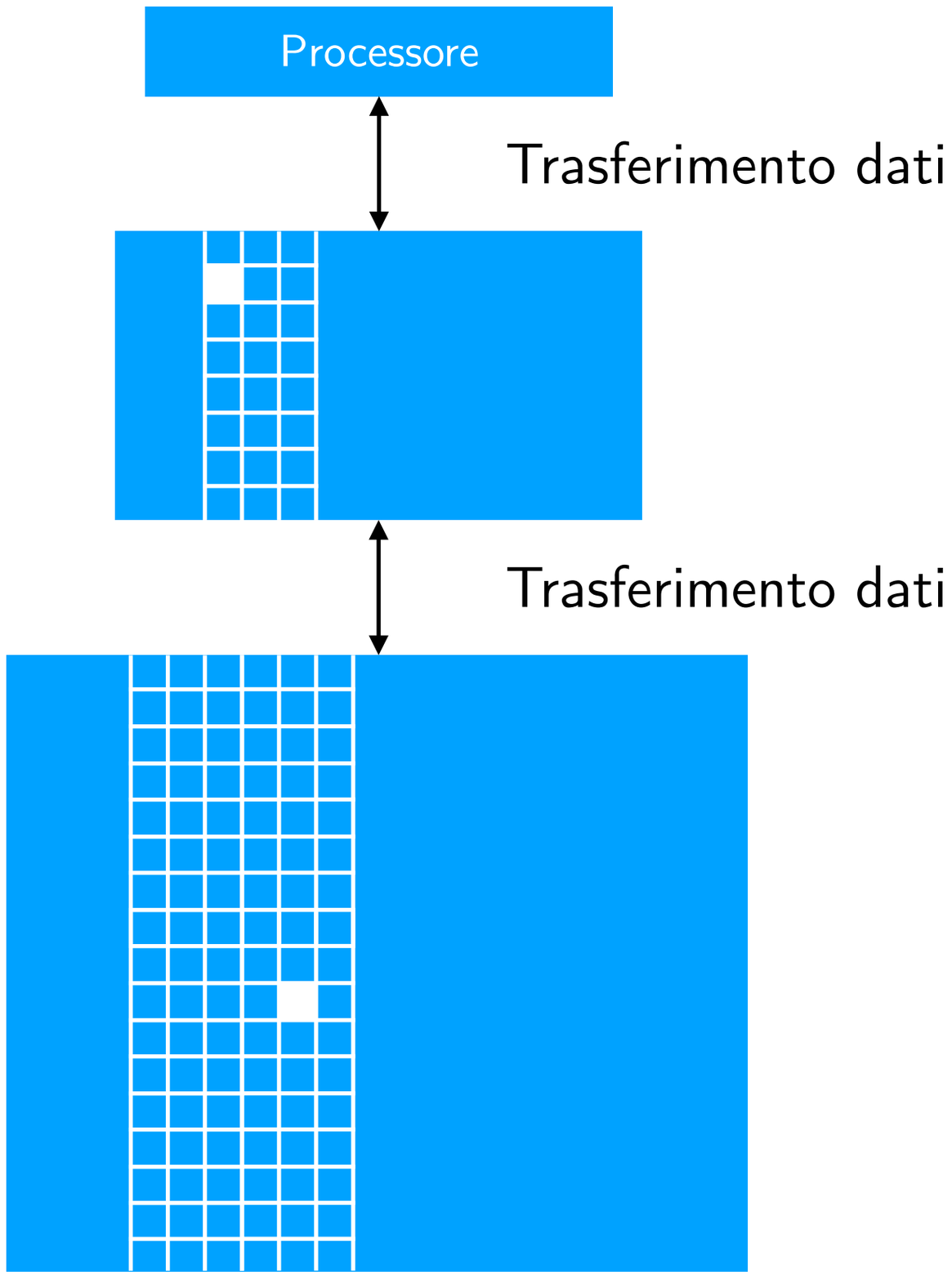
Principio di località

- I programmi accedono a una piccola parte del loro spazio di indirizzamento in ogni istante
- **Località temporale**
  - Oggetti acceduti recentemente saranno con alta probabilità acceduti a breve
  - Per esempio, istruzioni in un ciclo
- **Località spaziale**
  - Oggetti vicini a quelli acceduti recentemente saranno con alta probabilità acceduti a breve
  - Per esempio, accesso sequenziale delle istruzioni, dati negli array

Sfruttare la località

- Gerarchia di memoria
- Memorizzare tutto su disco
- Copiare gli oggetti acceduti di recente (e vicini) dal disco a una memoria DRAM più piccola
  - Memoria principale
- Copiare gli oggetti acceduti ancor più di recente (e ancor più vicini) dalla memoria DRAM a una memoria SRAM più piccola
  - Memoria cache collegata direttamente alla CPU

Livelli di gerarchia di memoria



- **Blocco** (o **linea**): unità di copia
  - Può contenere word multiple
- Se il dato richiesto è presente nel livello superiore
  - **Hit**: accesso soddisfatto dal livello superiore
    - **Hit ratio** = # hit / # accessi
- Se il dato richiesto è assente
  - **Miss**: blocco copiato dal livello inferiore
    - Tempo richiesto: penalità di miss
    - **Miss ratio** = # miss / # accessi = 1 - hit ratio
  - Quindi il dato richiesto è fornito dal livello superiore

Tecnologie della memoria

- RAM Statica (SRAM)
  - 0.5 ns – 2.5 ns, 500$ – 1000$ per GB
- RAM Dinamica (DRAM)
  - 50 ns – 70 ns, 10$ – 20$ per GB
- Memoria flash
  - 5 µs – 50 µs, 0.75$ – 1$ per GB
- Disco magnetico
  - 5 ms – 20 ms, 0.05$ – 0.10 $ per GB
- Memoria ideale
  - Tempo di accesso di una SRAM
  - Capacità e costo/GB di un disco magnetico

Tecnologia SRAM

- Circuiti integrati organizzati come vettori di memoria
  - 6 – 8 transistor per bit
- Stesso tempo di accesso per tutti i dati
  - Ma i tempi di accesso in lettura e scrittura possono essere diversi
- Tempo di accesso molto vicino al ciclo di clock

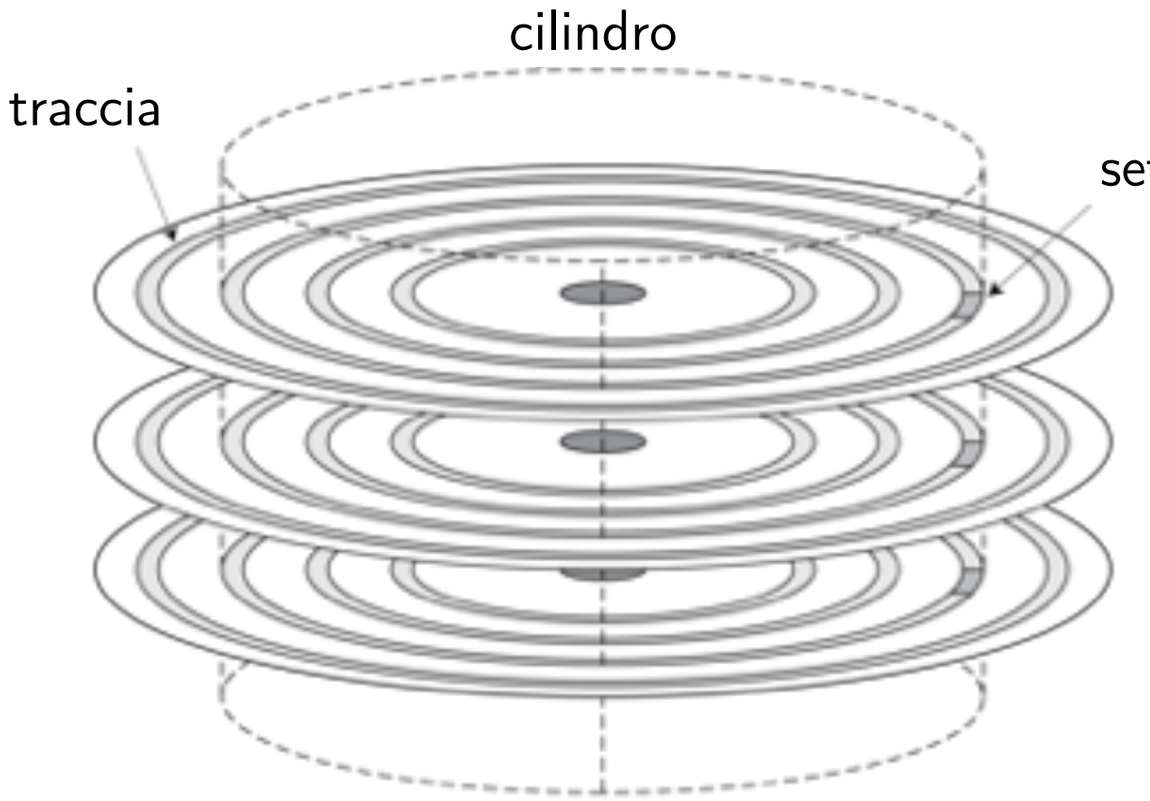
Tecnologia DRAM

- I dati sono memorizzati come carica di un condensatore
  - Un transistor usato per accedere alla carica
  - Deve essere periodicamente aggiorntato
    - Leggere i contenuti e riscriverli
    - Eseguito "a righe"
- I bit di una DRAM sono organizzati come array rettangolari
  - La DRAM accede a una riga intera
  - Modalità burst: fornisce parole successive da una riga con latenza ridotta
- DDR (double data rate) RAM
  - Trasferimento sui fronti in salita e in discesa del clock

Tecnologia flash

- Memoria a sola lettura, cancellabile elettronicamente e programmabile (EEPROM)
  - NOR flash: cella di bit come una porta NOR
    - Accesso in lettura/scrittura casuale
    - Usata per le memoria delle istruzioni nei sistemi embedded
  - NAND flash: cella di bit come una porta NAND
    - Più densa (# bit/area) ma accesso un blocco alla volta
    - Meno costosa per GB
    - Usata per le chiavi USB
- I bit si deteriorano dopo un certo numero (1000+) di scritture
  - Inadatte per rimpiazzare direttamente il disco o la RAM
  - Livellamento dell'usura (*wear leveling*)
    - Rimappare i blocchi di memoria che sono stati scritti più spesso sui blocchi che sono stati scritti meno di frequente
    - Perdita di prestazioni nominali

Tecnologia a disco

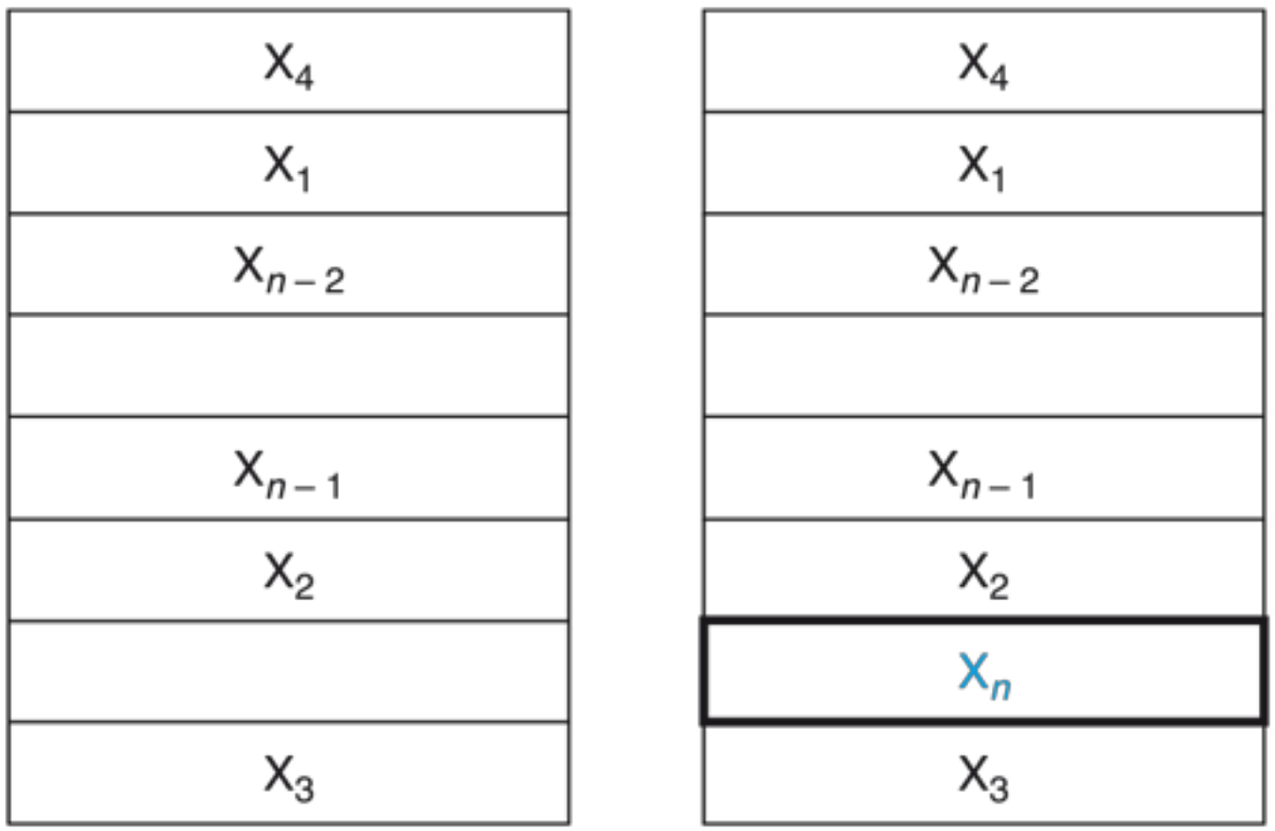


Tecnologia a disco

- Ogni settore memorizza, tra le altre cose
  - ID del settore
  - Dati (512 – 4096 byte)
  - Codice di correzione di errore (ECC)
- L'accesso a un settore richiede
  - Ritardo di attesa se altri accessi sono in attesa
  - Ricerca (*seek*): muovere le testine
  - Latenza rotazionale
  - Trasferimento dati
  - Overhead del controllore

Memoria Cache

- Livello della gerarchia di memoria più vicino alla CPU
- Dati gli accessi X_1, ..., Xn-1, Xn

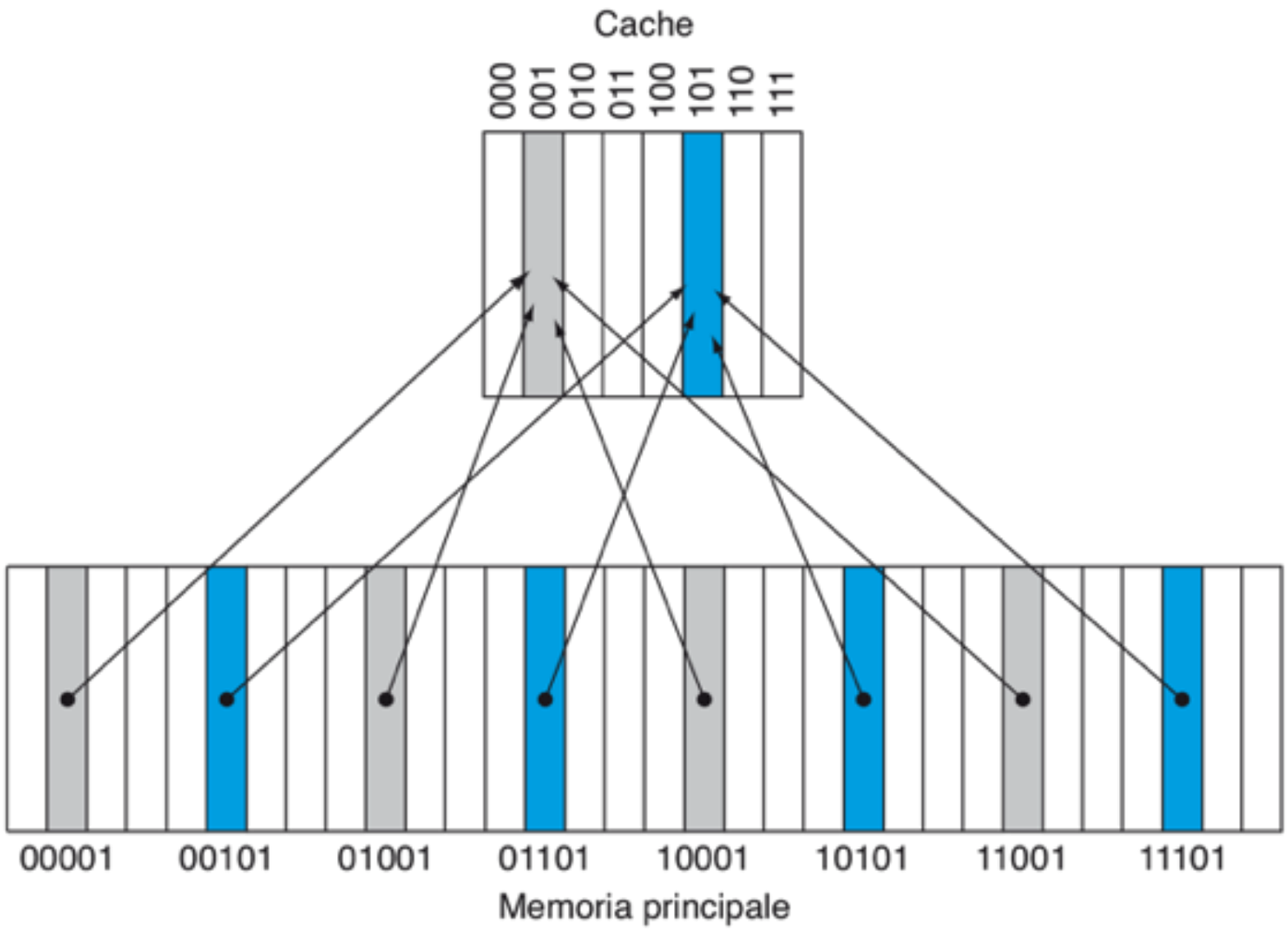


a. Prima di richiedere il dato X_n b. Dopo avere richiesto il dato X_n

- Come sappiamo se il dato è presente in cache?
- Dove andiamo a cercare il dato?

Cache a mappature diretta

- Locazione determinata dall'indirizzo
- **Mappatura diretta**: un'unica possibile scelta
  - (indirizzo del blocco) `modulo` (num. blocchi in cache)



- Num. blocchi è una potenza di 2
- Si usano i bit di indirizzo meno significativi

Tag e bit di validità

- Come sappiamo quale particolare blocco è memorizzato in una data locazione di cache?
  - Si memorizza l'indirizzo del blocco insieme ai dati
  - In realtà, ci servono solo i bit più significativi
  - Costituiscono il **campo tag**
- Cosa succede se non ci sono dati in una locazione di cache?
  - **Bit di validità**: 1 = dato presente, 0 = dato assente
  - Inizialmente posto a 0

Esempio

- 8 blocchi, 1 word per blocco, mappatura diretta
- Stato iniziale

Indice	V	Tag	Dati
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Esempio

Indirizzo di memoria decimale	Indirizzo di memoria binario	Hit/Miss	Blocco di cache
22	10 110	Miss	110

Indice	V	Tag	Dati
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	S	10	Mem[10110]
111	N		

Esempio

Indirizzo di memoria decimale	Indirizzo di memoria binario	Hit/Miss	Blocco di cache
26	11 010	Miss	010

Indice	V	Tag	Dati
000	N		
001	N		
010	S	11	Mem[11010]
011	N		
100	N		
101	N		
110	S	10	Mem[10110]
111	N		

Esempio

Indirizzo di memoria decimale	Indirizzo di memoria binario	Hit/Miss	Blocco di cache
22	10 110	Hit	110
26	11 010	Hit	010

Indice	V	Tag	Dati
000	N		
001	N		
010	S	11	Mem[11010]
011	N		
100	N		
101	N		
110	S	10	Mem[10110]
111	N		

Esempio

Indirizzo di memoria decimale	Indirizzo di memoria binario	Hit/Miss	Blocco di cache
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

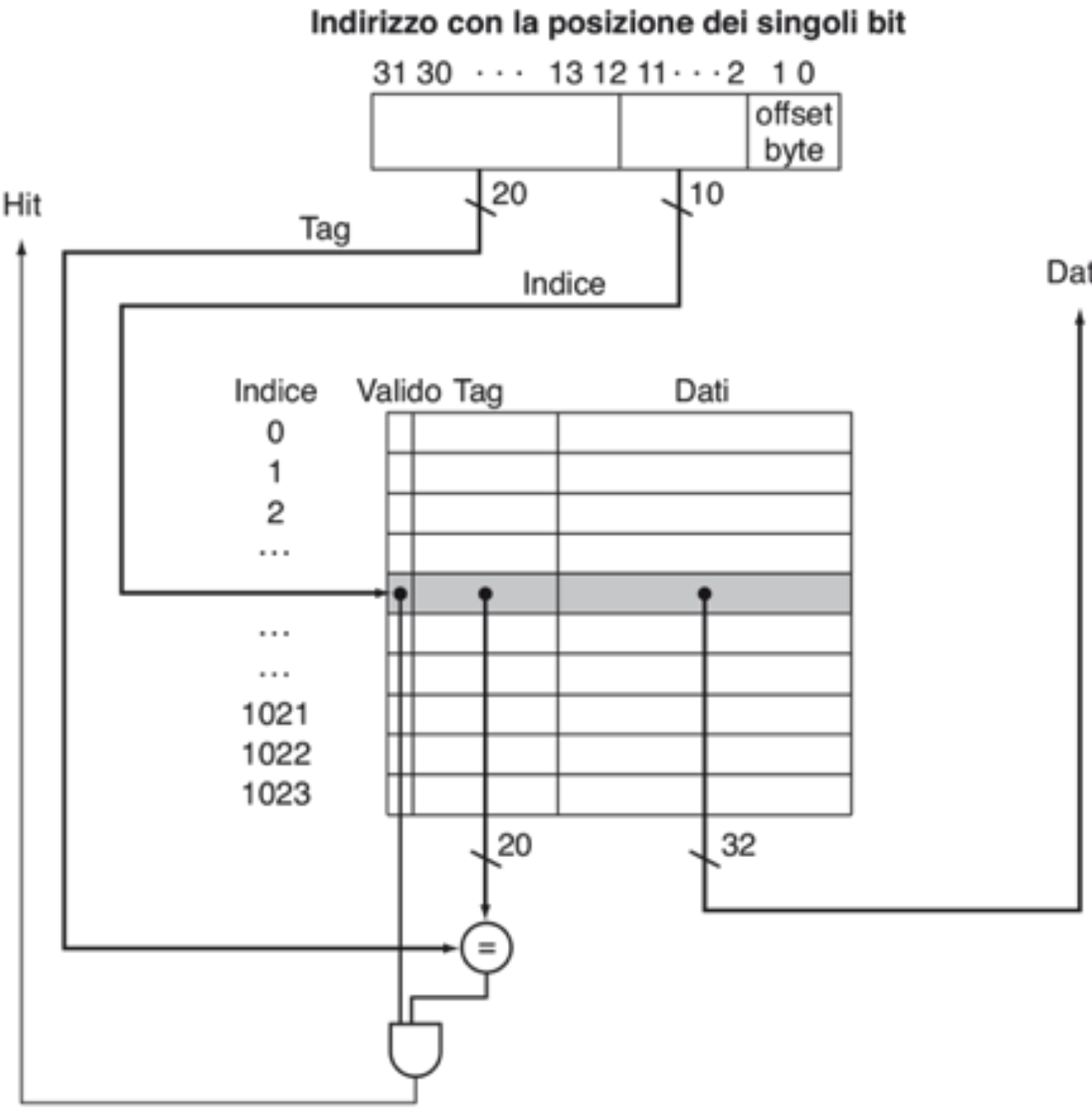
Indice	V	Tag	Dati
000	S	10	Mem[10000]
001	N		
010	S	11	Mem[11010]
011	S	00	Mem[00011]
100	N		
101	N		
110	S	10	Mem[10110]
111	N		

Esempio

Indirizzo di memoria decimale	Indirizzo di memoria binario	Hit/Miss	Blocco di cache
18	10 010	Miss	010

Indice	V	Tag	Dati
000	S	10	Mem[10000]
001	N		
010	S	10	Mem[10010]
011	S	00	Mem[00011]
100	N		
101	N		
110	S	10	Mem[10110]
111	N		

Suddivisione dell'indirizzo



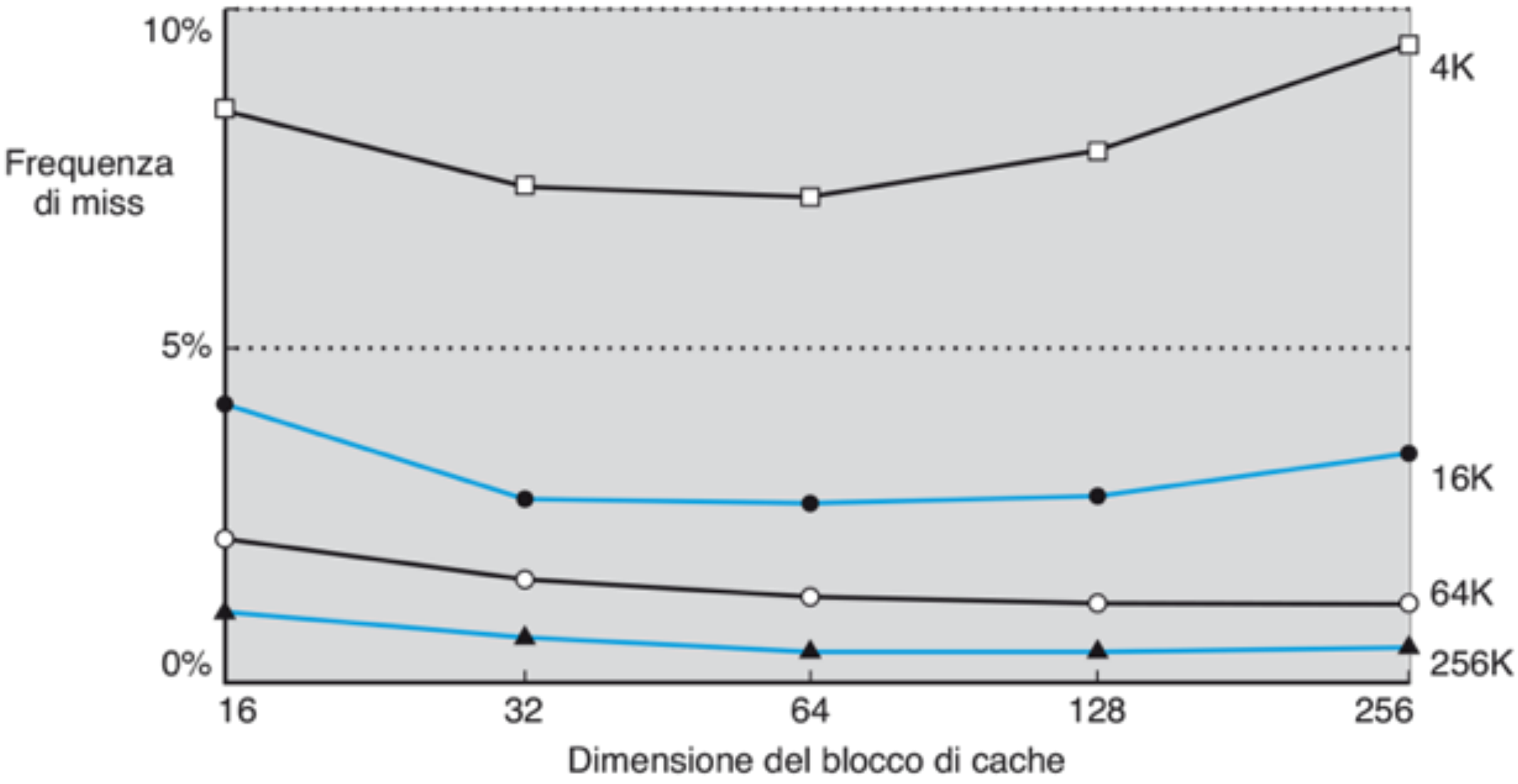
Blocchi composti da più parole

- 64 blocchi, 16 byte per blocco
- A quale numero di blocco corrisponde l'indirizzo 1200?
- Indirizzo di blocco = $\lfloor 1200 \ / \ 16 \rfloor = 75$
- Numero di blocco = 75 `modulo` 64 = 11

Dimensione del blocco

- Blocchi più grandi dovrebbero ridurre il tasso di miss
  - A causa della spazialità locale
- Ma in una cache a dimensione fissa
  - Blocchi più grandi ⇨ Meno blocchi
    - Maggiore competizione ⇨ Aumento del tasso di miss
  - Blocchi più grandi ⇨ Maggior "sporcizia"
- Maggiore penalità di miss
  - Può vanificare i benefici di un tasso di miss ridotto
  - Tecniche avanzate per aiutare in questi casi

Prestazioni



Miss della cache

- In caso di hit, la CPU procede normalmente
- In caso di miss
  - La pipeline della CPU entra in stallo
  - L'unità di controllo preleva il blocco dal livello successivo della gerarchia di memoria
- Miss della cache per un'istruzione
  - Si ripete dalla fase di fetch l'istruzione
- Miss della cache per un dato
  - Si completa l'accesso al dato

Write-Through

- Nel caso di hit in scrittura, si potrebbe semplicemente aggiornare il blocco di cache
  - Ma in tal caso cache e memoria sarebbero inconsistenti
- **Write through**: aggiornare anche la memoria
- Ma rende le scritture più lente
  - Ad esempio, se il CPI di base è 1, il 10% delle istruzioni sono di scrittura e la scrittura richiede 100 cicli di clock
    - CPI effettivo = 1 + 0.1 x 100 = 11
- Soluzione: **buffer di scrittura**
  - Contiene i dati in attesa di essere scritti in memoria
  - La CPU continua immediatamente
    - Entra in stallo sulla scrittura solo se il buffer di scrittura è pieno

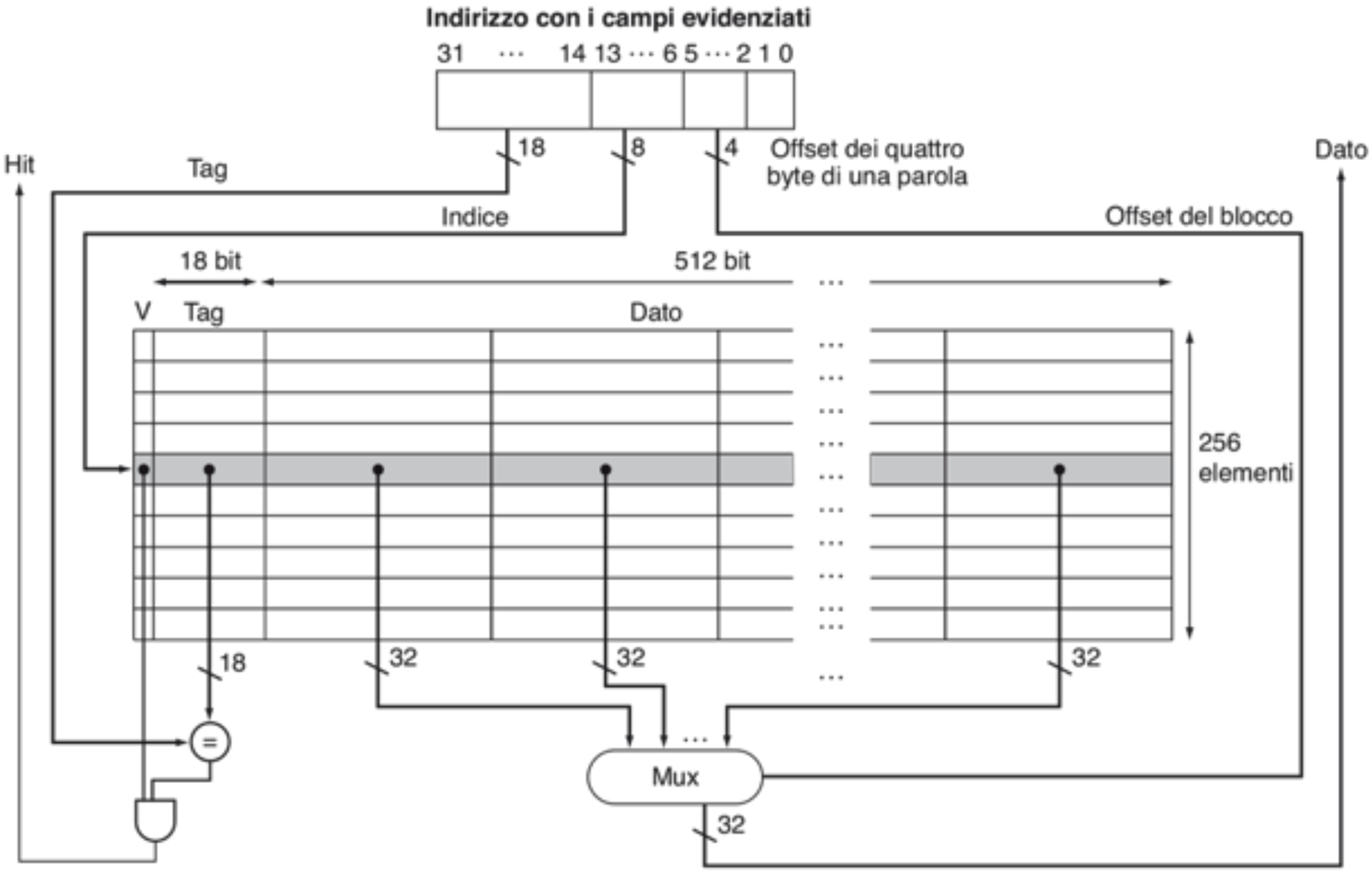
Write-Back

- Alternativa: nel caso di hit in scrittura si aggiorna solo il blocco di cache
  - Bisogna tenere traccia di quali blocchi sono "sporchi" (*dirty*)
- Quando un blocco "sporco" è rimpiazzato
  - Si ri-scrive sulla memoria
  - Si può usare un buffer di scrittura per permettere di rimpiazzare il blocco che deve essere letto per primo

Esempio: Intrinsity FastMATH

- Processore MIPS dedicato
  - Pipeline a 12 stadi
  - Accesso a dati e istruzioni in ciascun ciclo
- Split cache: cache separate per istruzioni e dati
  - Ciascuna di 16 KB: 256 blocchi per 16 parole/blocco
  - Cache dati: write-through o write-back
- Tassi di miss:
  - Cache istruzioni: 0.4 %
  - Cache dati: 11.4 %
  - Media pesata: 3.2 %

Esempio: Intrinsity FastMATH



Misurare le prestazioni della cache

- Componenti del tempo di CPU
  - Cicli di esecuzione del programma
    - Incluso il tempo di hit in cache
  - Cicli di stallo della memoria
    - Principalmente dovuti alle miss in cache
- Con assunzioni semplificative

Cicli di stallo della memoria
= Num. accessi in memoria x tasso di miss x penalità di miss
= Num. istruzioni x num miss/istruzione x penalità di miss

Esempio

- Dati
  - Tasso di miss della I-cache = 2 %
  - Tasso di miss della D-cache = 4 %
  - Penalità di miss = 100 cicli
  - CPI base (cache ideale) = 2
    - Load e store sono il 36% delle istruzioni
- Cicli di miss per istruzione
  - I-cache: 0.02 x 100 = 2
  - D-cache: 0.36 x 0.04 x 100 = 1.44
- CPI reale = 2 + 2 + 1.44 = 5.44
  - La CPU ideale è 5.44/2 = 2.72 volte più veloce

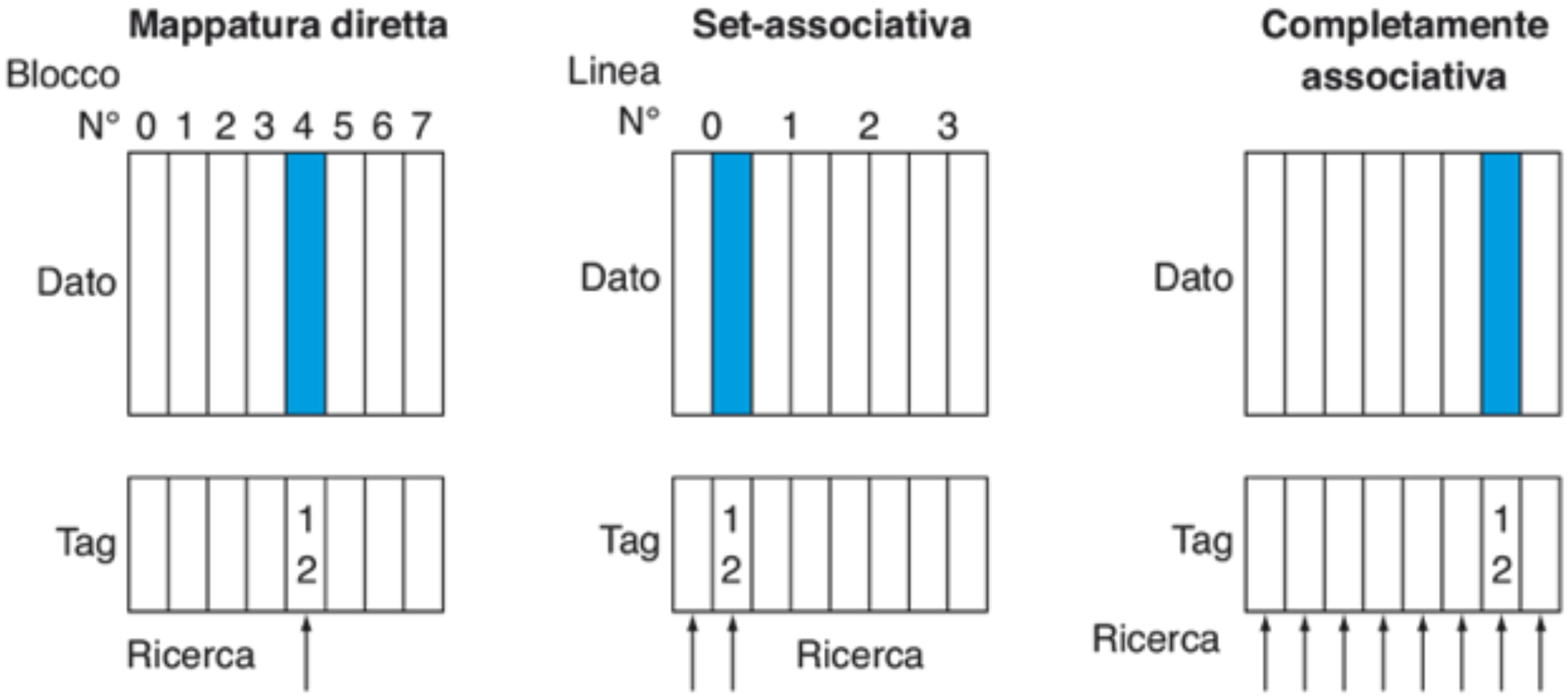
Sommario sulle prestazioni

- Quando le prestazioni della CPU aumentano
  - La penalità di miss diventa più significativa
- Diminuendo il CPI base
  - Una maggior parte del tempo è speso per gli stalli di memoria
- Aumentando la frequenza di clock
  - Gli stalli di memoria costano più cicli di clock
- Non si può trascurare il comportamento della cache quando si valutano le prestazioni di un sistema

Cache associative

- **Completamente associative**
  - Ogni blocco di memoria può andare in qualsiasi blocco di cache
  - Richiede di ricercare in parallelo tutti i blocchi di cache
  - Un comparatore per blocco di cache (costoso)
- **Set-associative**
  - Ogni set contiene *n* blocchi di cache
  - Il numero di blocco determina il set
    - (Numero di blocco) `modulo` (numero di set nella cache)
  - Richiede di ricercare in parallelo tutti i blocchi di un set
  - Solo n comparatori (meno costoso)

Esempio di cache associativa



Varianti di associatività

Cache set-associativa a una via
(a mappatura diretta)

Blocco	Tag	Dato
0		
1		
2		
3		
4		
5		
6		
7		

Cache set-associativa a due vie

Linea	Tag	Dato	Tag	Dato
0				
1				
2				
3				

Cache set-associativa a quattro vie

Linea	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato
0								
1								

Cache set-associativa a otto vie (completamente associativa)

Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato

Esempio

- Confrontiamo una cache a 4 blocchi
  - A mappatura diretta, set-associativa a 2 vie, completamente associativa
- Sequenza di accesso ai blocchi: 0, 8, 0, 6, 8

- A mappatura diretta

Indirizzo di blocco	Indice di cache	Hit/miss	Contenuto della cache dopo l'accesso			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Esempio

- Set-associativa a 2 vie

Indirizzo di blocco	Indice di cache	Hit/miss	Contenuto della cache dopo l'accesso			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[0]	Mem[6]		

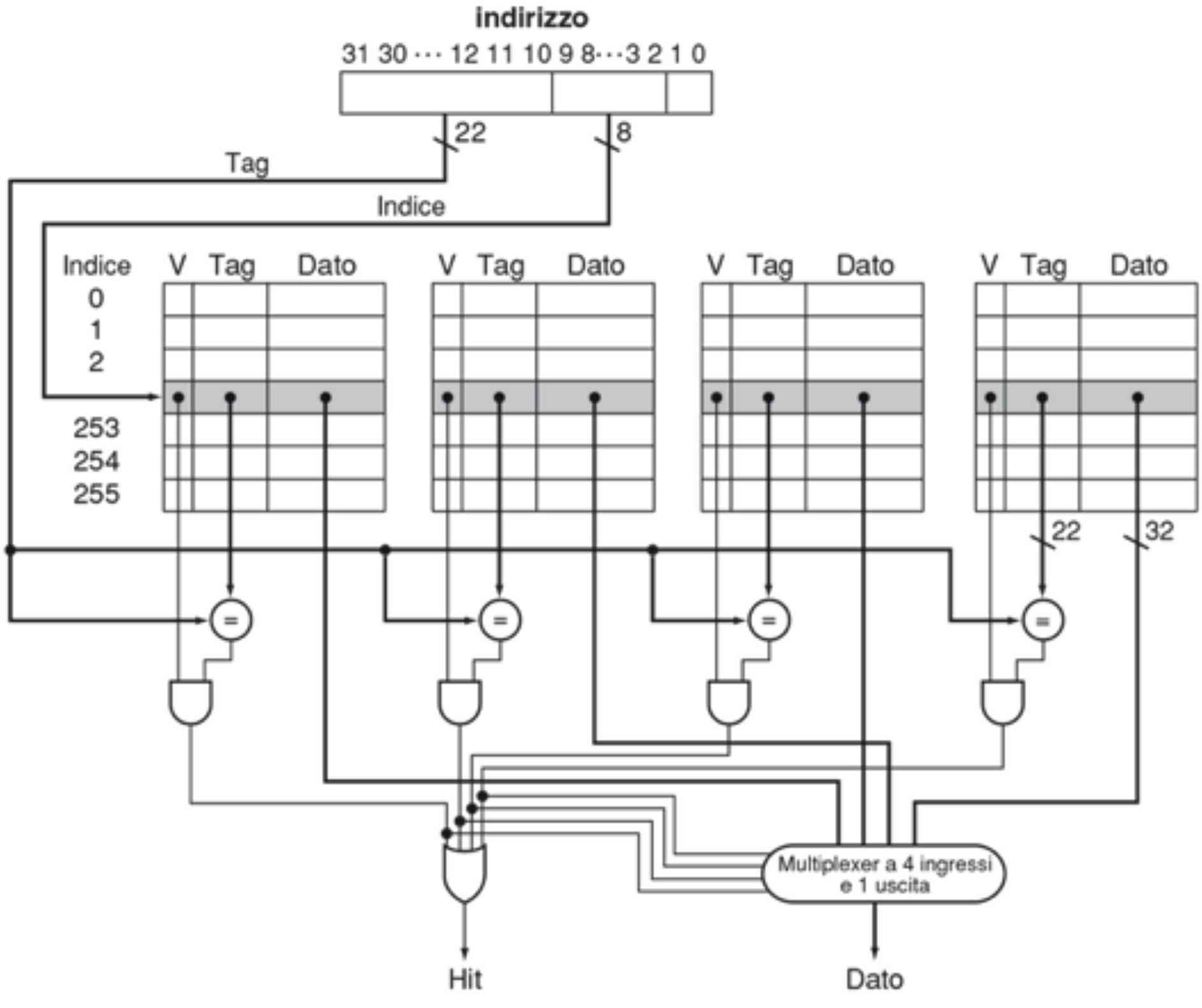
- Completamente associativa

Indirizzo di blocco		Hit/miss	Contenuto della cache dopo l'accesso			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

Quanta associatività

- Aumentare l'associatività diminuisce il tasso di miss
  - Ma con benefici sempre minori
- Simulazione di un sistema con 64 KB D-cache, blocchi di 16 parole:
  - 1 via: 10.3%
  - 2 vie: 8.6%
  - 4 vie: 8.3%
  - 8 vie: 8.1%

Organizzazione di una cache set-associativa



Politica di rimpiazzamento

- A mappature diretta: nessuna scelta
- Set-associativa
  - Preferire blocchi non validi, se possibile
  - Altrimenti, scegliere tra i rimanenti blocchi nell'insieme
- **Utilizzo meno recente** (LRU, *least recently used*)
  - Scegliere il blocco rimasto inutilizzato più a lungo
    - Semplice per 2 vie, gestibile per 4 vie, troppo complesso per più vie
- **Casuale** (*random*)
  - Approssimativamente esibisce le stesse prestazioni di LRU per elevata associatività

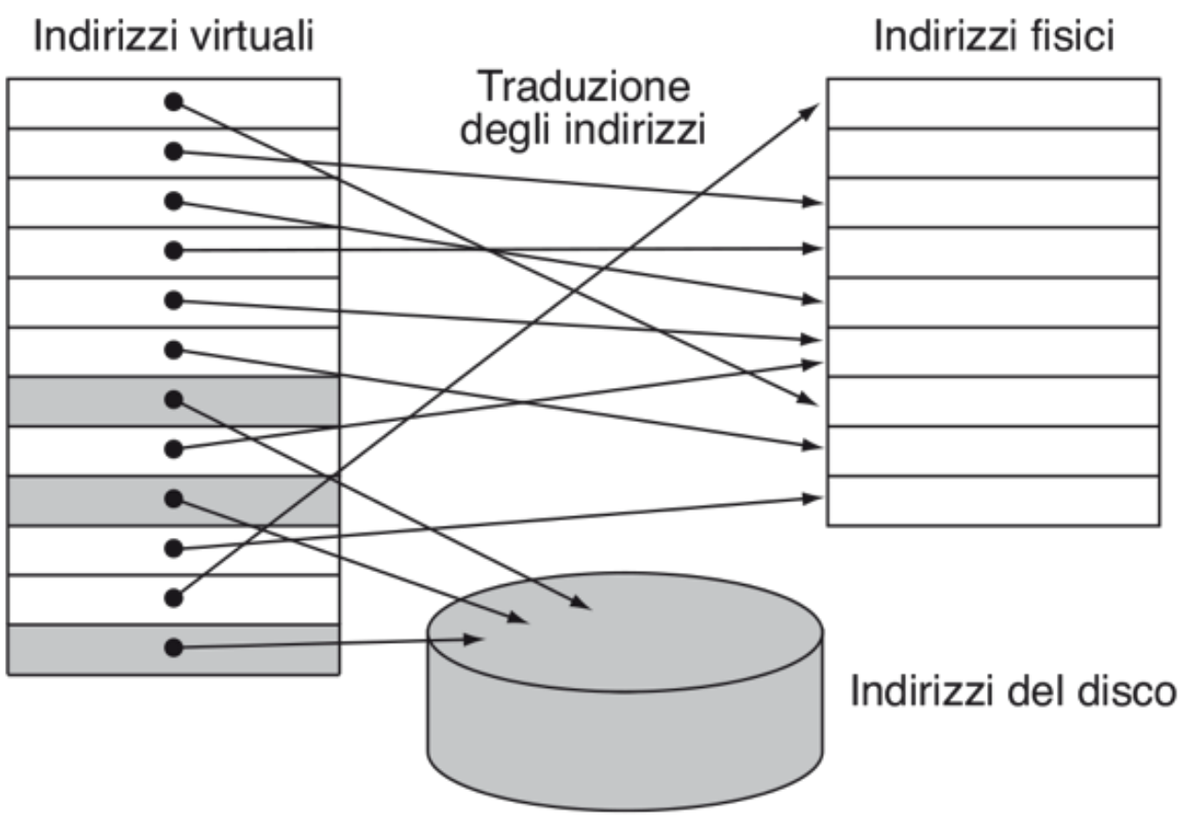
Cache multilivello

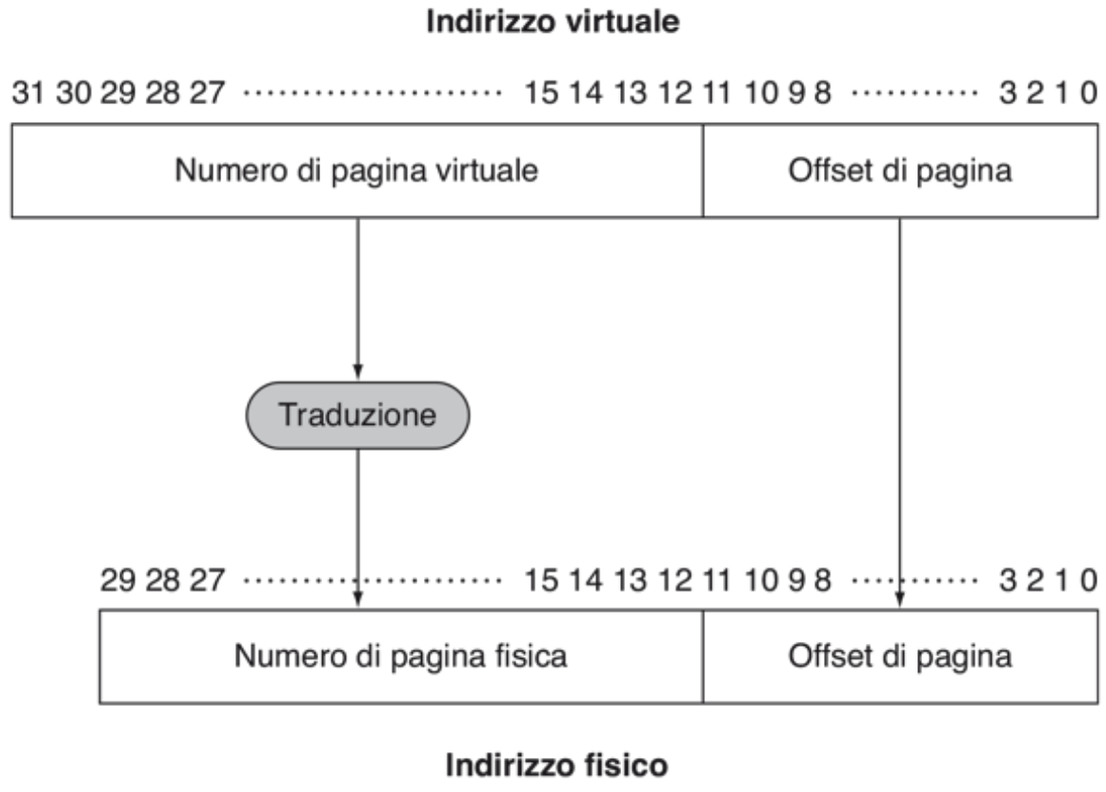
- Cache principale (di primo livello, L1) unita alla CPU
  - Piccola ma veloce
- Cache di secondo livello (L2) serve le miss della cache principale
  - Più grande, più lenta, ma più veloce della memoria principale
- La memoria principale serve le miss della cache L2
- Alcuni sistemi ad elevate prestazioni includono una cache L3

Memoria Virtuale

- Uso della memoria principale come "cache" per i dispositivi di memorizzazione secondari (dischi)
  - Gestita congiuntamente da hardware della CPU e dal sistema operativo
- I programmi condividono la memoria principale
  - Ciascuno ottiene uno **spazio di indirizzamento virtuale** contenente il codice e i dati usati frequentemente
  - Protetto dagli altri programmi
- La CPU e il sistema operativo traducono gli indirizzi virtuali in indirizzi fisici
  - Un blocco di memoria virtuale si chiama **pagina**
  - Una miss di memoria virtuale si chiama **fault di pagina**

Traduzione degli indirizzi





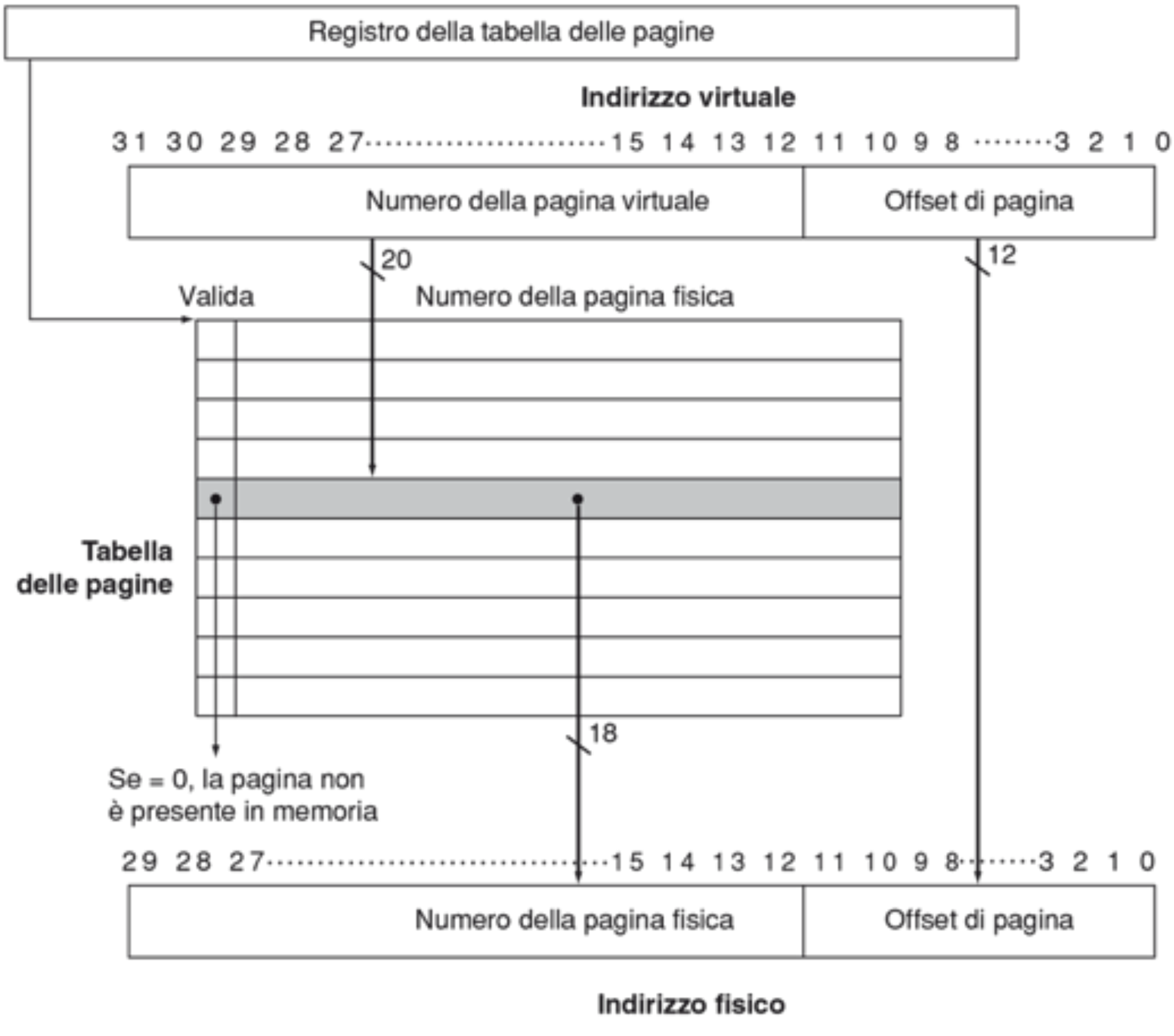
Penalità di fault di pagina

- In caso di fault di pagina, la pagina deve essere prelevata dal disco
  - Richiede milioni di cicli di clock
  - Gestito dal codice del sistema operativo
- Bisogna cercare di minimizzare il tasso di fault di pagine
  - Gestione completamente associativa
  - Algoritmi di rimpiazzamento intelligenti

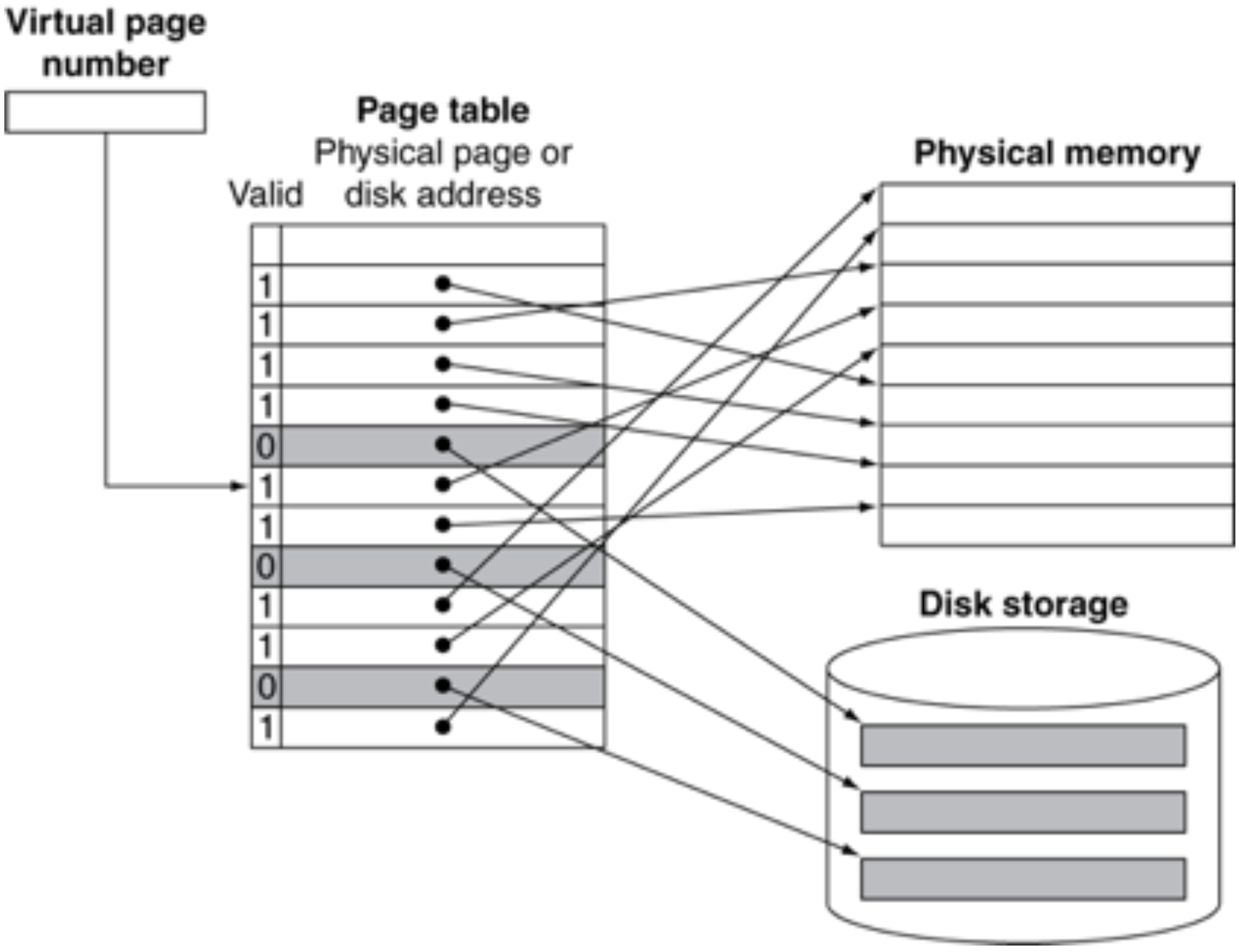
Tabella delle pagine

- Contiene le informazioni di piazzamento delle pagine
  - Array di elementi indicizzato dal numero di pagina virtuale
  - Il **registro della tabella delle pagine** nella CPU punta alla tabella delle pagine corrente nella memoria principale
- Se una pagina è presente in memoria
  - L'elemento della tabella delle pagine memorizza il numero di pagina fisica
  - Più altri bit di status (references, dirty, ecc)
- Se una pagina non è presente
  - L'elemento della tabella delle pagine può riferire la locazione nello **spazio di swap** del disco

Traduzione con la tabella delle pagine



Mappare pagine su memoria



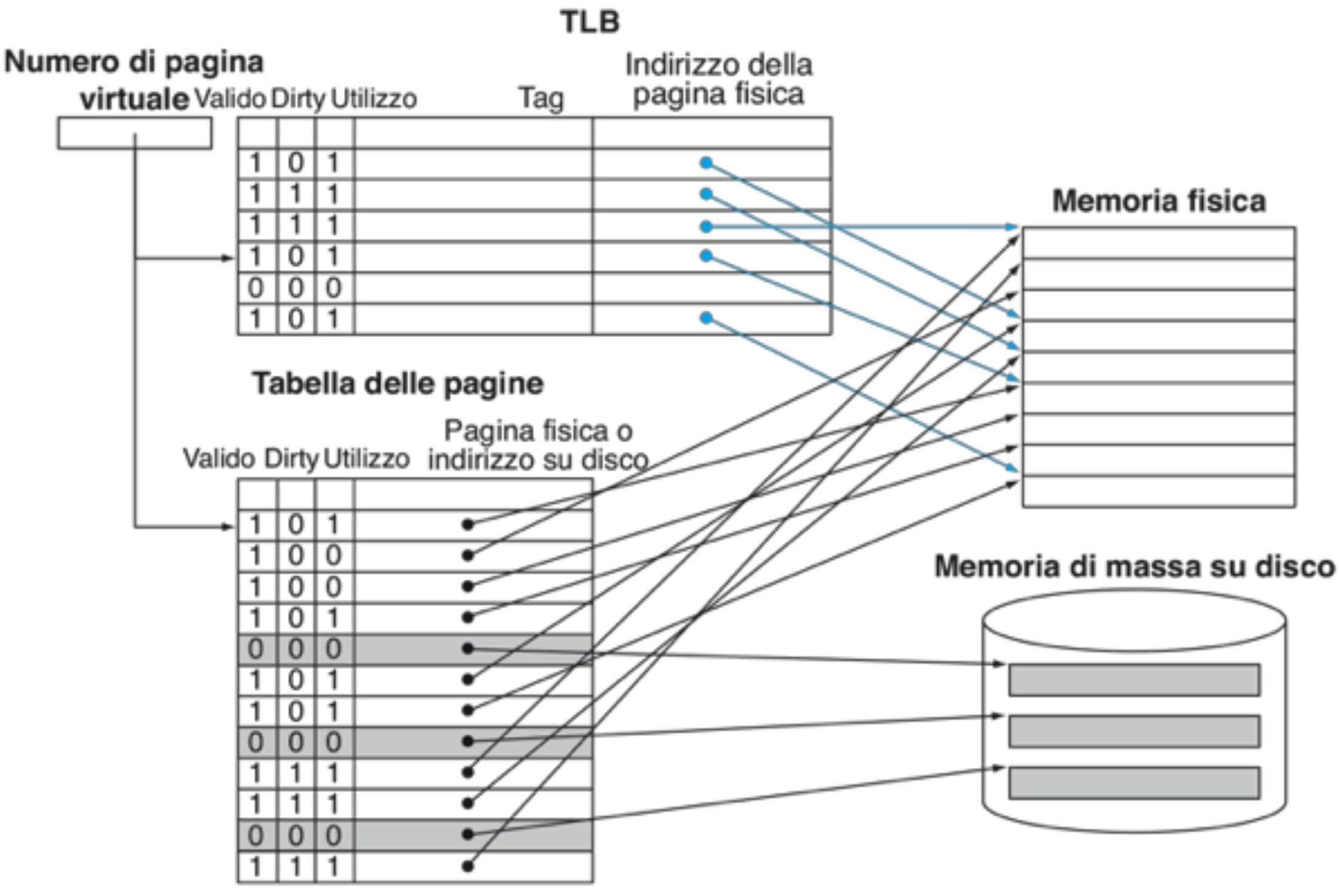
Rimpiazzamento e scritture

- Per ridurre il tasso di fault di pagina, si preferisce usare il rimpiazzamento LRU
  - **Bit di indirizzamento** (detto anche **bit di uso**) nell'elemento della tabella delle pagine settato a 1 se una pagine è acceduta
  - Periodicamente resettato a 0 dal sistema operativa
    - Una pagina con bit di indirizzamento a 0 non è stata acceduta di recente
- Le scritture su disco richiedono milioni di cicli
  - Scrivere a blocchi, non singole locazioni
  - Write-through non è pratico
  - Usare write-back
  - Il **bit sporco** nell'elemento della tabella delle pagine settato quando una pagine è scritta

Traduzione veloce tramite TLB

- La traduzione dell'indirizzo richiede accessi alla memoria extra
  - Uno per accedere all'elemento della tabella delle pagine
  - Uno per accedere all'elemento di memoria
- Ma l'accesso alla tabella delle pagine ha una buona località
- Quindi usiamo una cache veloce per gli elementi della tabella delle pagine all'interno della CPU
- Chiamato **translation lookaside buffer** (TLB)
- Tipicamente: 16 – 512 elementi, 0.5 – 1 ciclo per hit, 10 – 100 cicli per miss, 0.01% – 1% tasso di miss
- Le miss possono essere gestite via hardware o via software

Traduzione veloce tramite TLB



Miss nel TLB

- Se una pagina è in memoria
  - Caricare l'elemento della tabella delle pagine dalla memoria e riprovare
  - Può essere gestito via hardware
    - Può diventare complicato per strutture della tabella delle pagine più complesse
  - O via software
    - Sollevare un'eccezione speciale, con un handler ottimizzato
- Se la pagina non è in memoria (fault di pagina)
  - Il sistema operativo gestisce il caricamento della pagina e l'aggiornamento della tabella delle pagine
  - Quindi ri-esegue l'istruzione che ha causato il fault

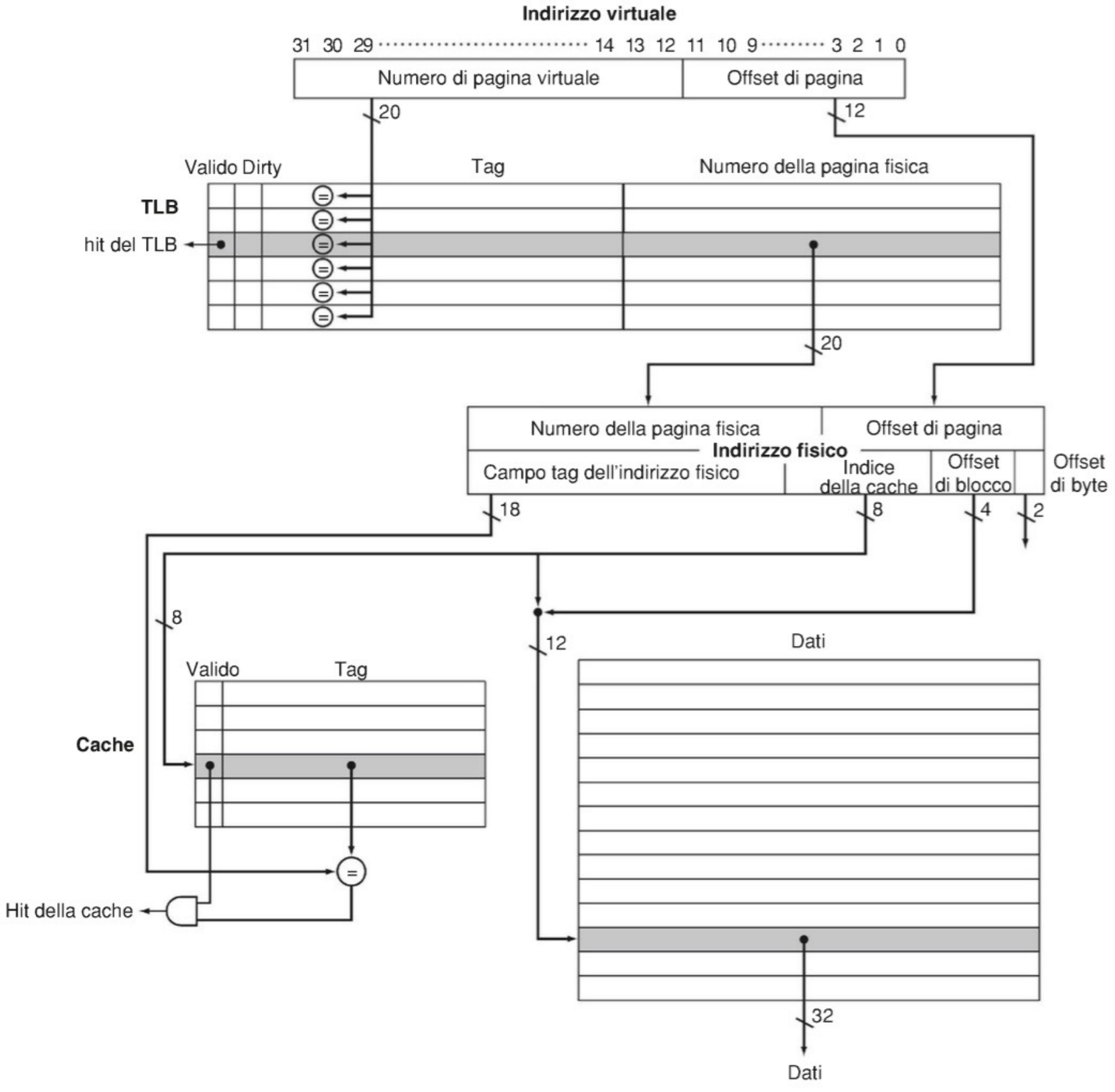
Handler delle Miss nel TLB

- Una miss nel TLB indica
  - Pagina presente, ma l'elemento della tabella delle pagine non è presente nel TLB
  - Pagina non presente
- Deve riconoscere una miss nel TLB prima che il registro di destinazione sia sovrascritto
  - Sollevando un'eccezione
- L'handler copia l'elemento della tabella delle pagine dalla memoria al TLB
  - Quindi ri-esegue l'istruzione
  - Se la pagina non è presente, si verificherà un fault di pagina

Handler dei fault di pagine

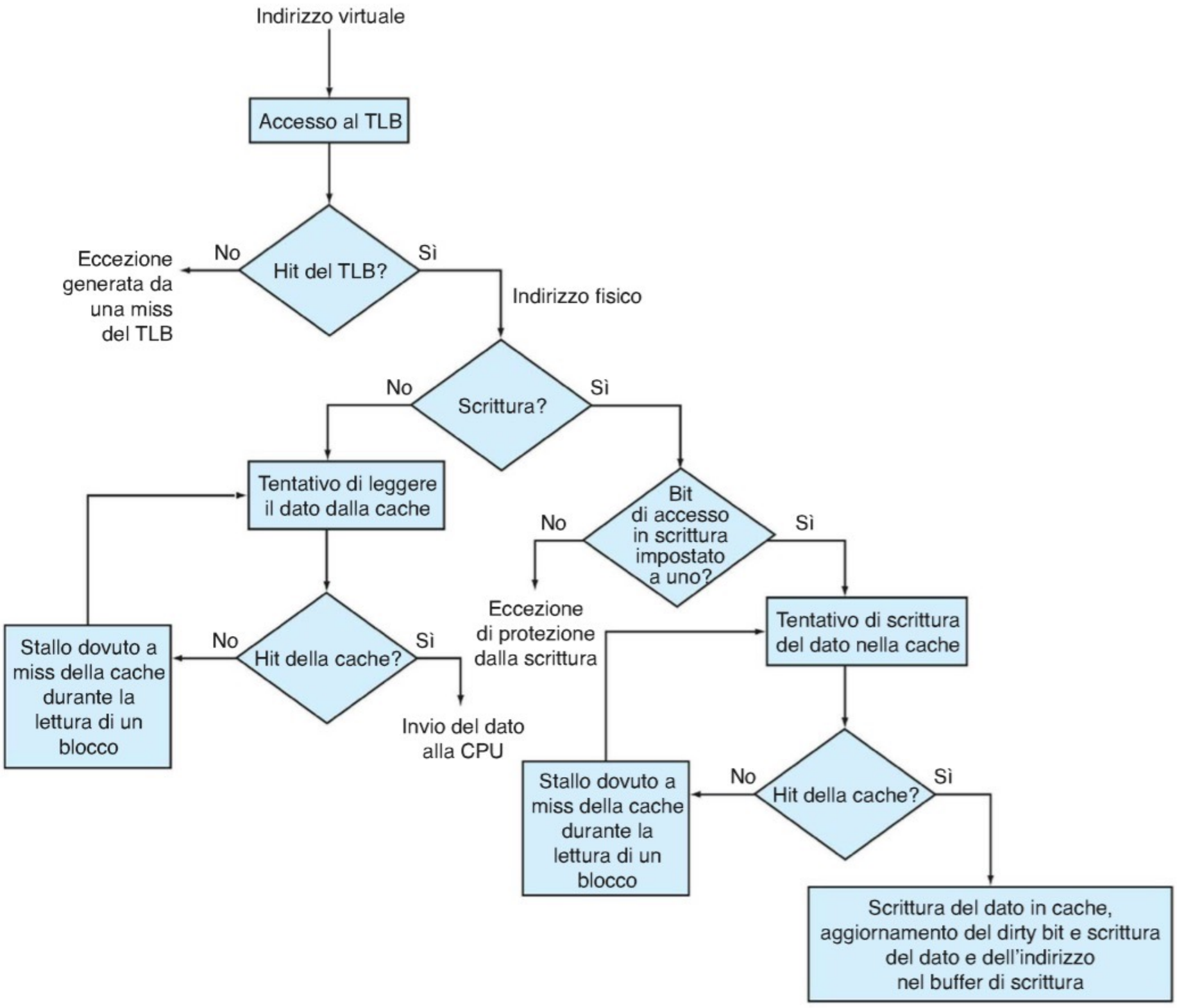
- Usare l'indirizzo virtuale cha ha causato il fault per localizzare l'elemento della tabella delle pagine
- Localizzare la pagina sul disco
- Scegliere la pagina da rimpiazzare
  - Se sporca, prima scriverla sul disco
- Caricare la pagina in memoria e aggiornare la tabella delle pagine
- Rimandare il processo in esecuzione
  - Ripartendo dall'istruzione che ha causato il fault

Interazione tra cache e TLB



- Se il tag della cache usa l'indirizzo fisico
  - Bisogna tradurre prima della ricerca nella cache
- Alternativamente, usare nel tag l'indirizzo virtuale
  - Complesso a causa dell'aliasing
    - Indirizzi virtuali diversi per indirizzi fisici condivisi

Gestione di lettura/scrittura write-through nel TLB e cache



Possibili miss con TLB, memoria virtuale e cache

TLB	Tabella delle pagine	Cache	Possibile? Se sì, in quale situazione?
Hit	Hit	Miss	Possibile, sebbene la tabella delle pagine non venga mai realmente controllata se si verifica una hit del TLB.
Miss	Hit	Hit	Miss del TLB, ma l'elemento si trova nella tabella delle pagine; al secondo tentativo, il dato viene trovato nella cache.
Miss	Hit	Miss	Miss del TLB, ma l'elemento si trova nella tabella delle pagine; al secondo tentativo, si verifica però una miss della cache.
Miss	Miss	Miss	Miss del TLB, seguita da un page fault; al secondo tentativo, l'accesso al dato deve provocare una miss della cache.
Hit	Miss	Miss	Impossibile: il TLB non può fornire la traduzione di una pagina che non è presente in memoria.
Hit	Miss	Hit	Impossibile: il TLB non può fornire la traduzione di una pagina che non è presente in memoria.
Miss	Miss	Hit	Impossibile: i dati non possono trovarsi nella cache se la pagina non è presente in memoria.

Protezione della memoria

- Task diversi possono condividere parte dei loro spazi di indirizzamento virtuali
  - Ma bisogna proteggerli da accessi accidentali
  - Richiede l'assistenza del sistema operativo
- Supporto hardware per la protezione del sistema operativo
- **Modalità supervisore** (o **modalità kernel**) privilegiata
- Istruzioni privilegiate
- Tabelle delle pagine e altre informazioni di stato accessibili sono in modalità supervisore
- Eccezione di **chiamata di sistema** (*syscall* nel MIPS)

Tipologia di miss

- Nelle varie gerarchie di memoria, le miss si possono verificare per cause diverse
  - **Modello delle tre C** per classificare le miss
  - Ci riferiremo al livello cache, anche se il modello si applica anche agli altri livelli della gerarchia di memoria
- Tipi di miss
- **Miss Certi**
  - miss di partenza a freddo, che si verifica quando il blocco deve essere portato nella cache per la prima volta
- **Miss per Capacità**
  - la cache non è in grado di contenere tutti i blocchi necessari all'esecuzione del programma
- **Miss per Conflitti**
  - due blocchi sono in conflitto per una certa posizione
  - può verificarsi anche se la cache NON è piena
  - questo tipo di miss non si verifica se abbiamo una cache completamente associativa

Cache di ARM e Intel

Caratteristica	Cortex-A8 ARM	Core i7 Intel
Organizzazione cache L1	Cache separate per dati e istruzioni	Cache separate per dati e istruzioni
Dimensione cache L1	32KiB per istruzioni/dati	32KiB per istruzioni/dati per ogni core
Grado associatività cache L1	Set-associativa a 4 vie (I), a 4 vie (D)	Set-associativa a 4 vie (I), a 8 vie (D)
Modalità sostituzione L1	Random	LRU approssimato
Dimensione blocco L1	64 byte	64 byte
Politica scrittura L1	Write-back, Write-allocate (?)	Write-back, No-write-allocate
Tempo hit L1 (utilizzo load)	1 ciclo di clock	4 cicli di clock (pipeline)
Organizzazione cache L2	Un'unica cache per istruzioni e dati	Un'unica cache per istruzioni e dati
Dimensione cache L2	Da 128 KiB a 1 MiB	256 KiB (0,25 MiB)
Grado associatività cache L2	Set-associativa a 8 vie	Set-associativa a 8 vie
Modalità sostituzione L2	Random (?)	LRU approssimato
Dimensione blocco L2	64 byte	64 byte
Politica scrittura L2	Write-back, Write-allocate (?)	Write-back, Write-allocate
Tempo hit L2	11 cicli di clock	10 cicli di clock
Organizzazione cache L3	-	Un'unica cache per istruzioni e dati
Dimensione cache L3	-	8 MiB, condivisa dai core
Grado associatività cache L3	-	Set-associativa a 16 vie
Modalità sostituzione L3	-	LRU approssimato
Dimensione blocco L3	-	64 byte
Politica scrittura L3	-	Write-back, Write-allocate
Tempo hit L3	-	35 cicli di clock

TLB di ARM e Intel

Caratteristica	Cortex-A8 ARM	Core i7 Intel
Indirizzo virtuale	32 bit	48 bit
Indirizzo fisico	32 bit	44 bit
Dimensione pagina	Variabile: 4, 16, 64 KiB, 1, 16 MiB	Variabile: 4 KiB, 2/4 MiB
Organizzazione TLB	1 TLB per le istruzioni e 1 TLB per i dati	1 TLB per le istruzioni e 1 TLB per i dati per core
	Entrambi i TLB sono completamente associativi, con 32 elementi, sostituzione di tipo *round robin* (circolare)	Entrambi i TLB di L1 sono set-associativi a 4 vie, sostituzione LRU
	Le miss dei TLB sono gestite in hardware	Il TLB Istruzioni di L1 ha 128 elementi per pagine piccole, 7 per ogni thread per pagine grandi.
		Il TLB Dati ha 64 elementi per pagine piccole, 32 elementi per pagine grandi
		Il TLB di L2 è set associativo a 4 vie, con sostituzione LRU
		Il TLB L2 ha 512 elementi
		Le miss dei TLB sono gestite in hardware