

## 1. Introduzione all'Architettura del Calcolatore

### Il Calcolatore

Il calcolatore:

- esegue *istruzioni semplici*
- ↑ in modo *sequenziale* → *programma*
- produce un output complesso
- utilizza *energia elettrica* → composto da *circuiti elettronici*

### CPU 9. Architettura di una CPU

E' la **CPU** (Central Processing Unit) in grado di:

- eseguire un insieme di "*istruzione elementari*"

### Memoria 11. Architettura della Memoria

La memoria è un circuito elettronico in grado di *preservare l'informazione*.

Quest'ultima può essere costituita da:

- **Istruzioni**, eseguite dalla CPU
  - **Dati**, utilizzati dalle istruzioni eseguite
- Può essere:
- **Volatile**: perdita dell'informazione (spento il calcolatore)
    - **RAM**: Random Access Memory
      - sia *lettura* che *scrittura* di informazioni
      - contiene i programmi *attualmente in esecuzione*
  - **Non-volatile**: permane l'informazione (spento il calcolatore)
    - **ROM**: Read Only Memory
      - solo *lettura*
      - pre-programmata
      - contiene il *boot-code*

**RAM e ROM fanno parte della Memoria Centrale**

Oltre RAM e ROM un calcolatore può avere una **memoria di massa** (HD - SSD):

- non-volatile
- sia lettura che scrittura
- catalogati come *dispositivi di I/O*

### Interfacce I/O 13. Il sistema di Input e Output

Le interfacce di Input/Output sono quei circuiti elettronici che permettono alla CPU di interagire con l'utente

### Bus di Sistema 12 Bus di Sistema

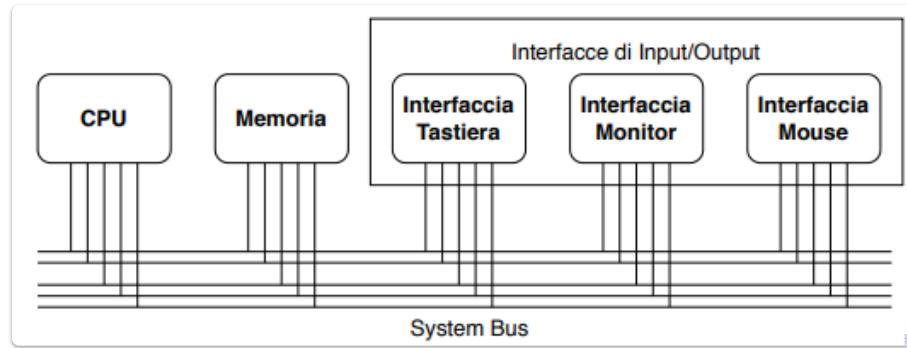
E' l'insieme di collegamenti elettronici che interconnette CPU, memoria, interfacce di I/O.

Ogni collegamento ha il suo ruolo e "trasporta" una tipologia di informazione ben precisa:

- **Address**

- Data
- Control

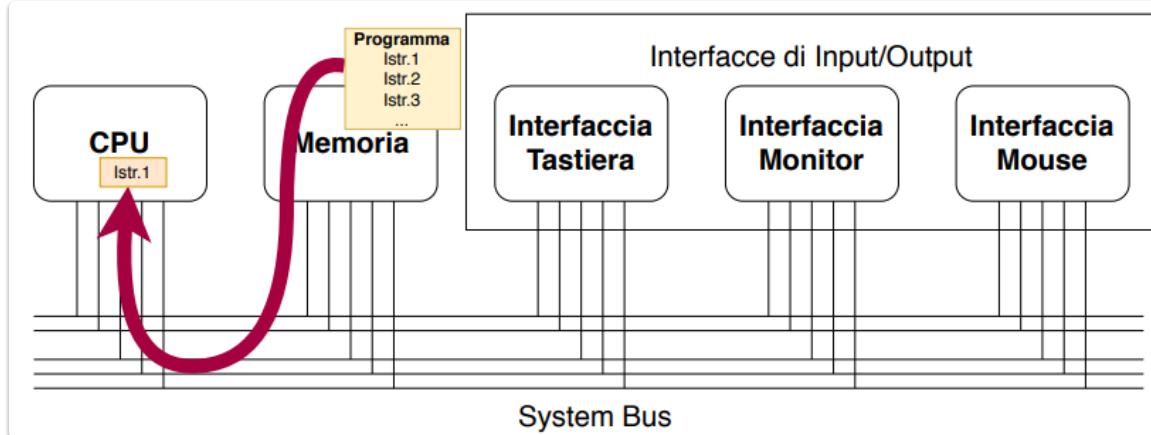
## Architettura Base di un Calcolatore



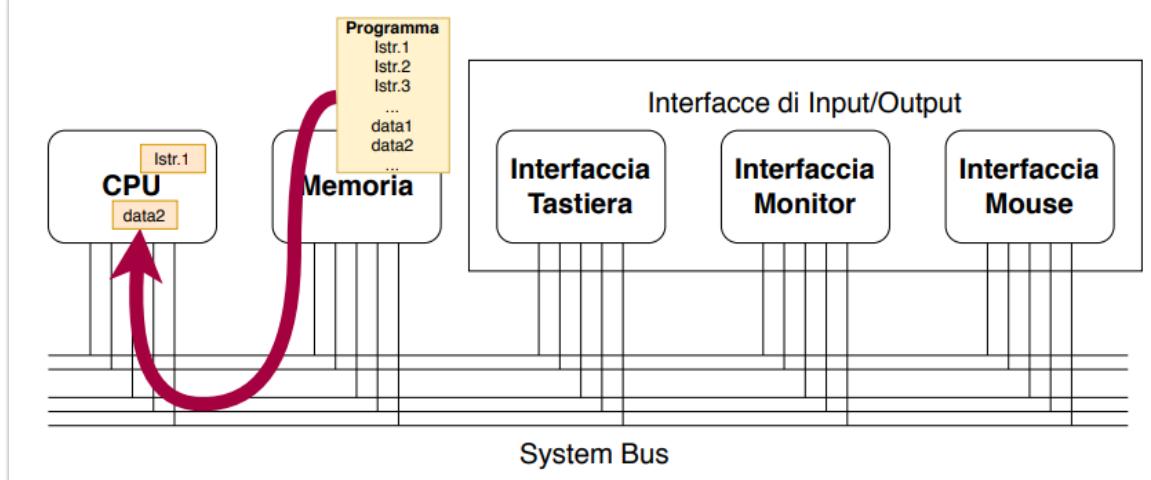
## Esecuzione di un programma

Lavoro della CPU:

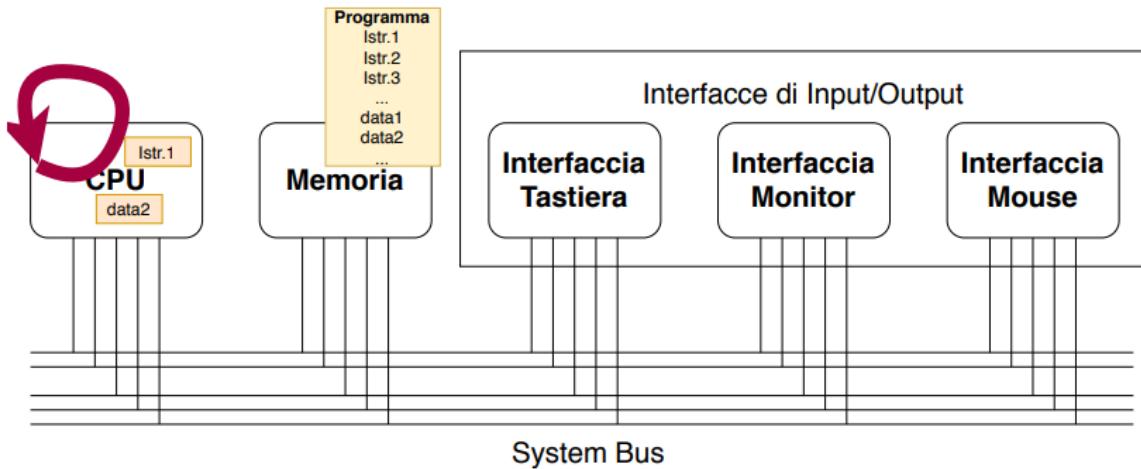
- **Fase 1: Fetch** dell'istruzione:



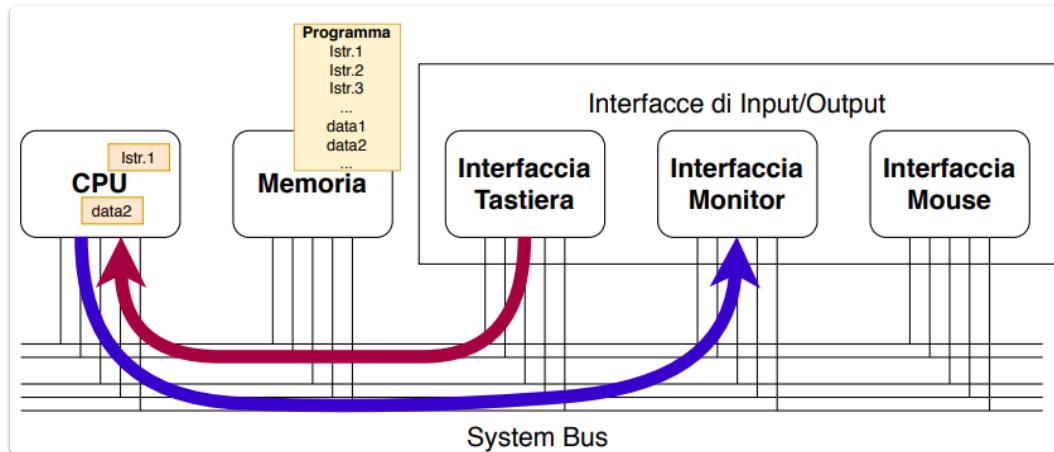
- "lettura" dell'istruzione da eseguire dalla memoria
  - memoria → istruzione → CPU
- **Fase 2: Interpretazione** dell'istruzione:



- qualora l'istruzione necessiti di **dati aggiuntivi** vengono letti (**Fetch**)
- **Fase 3: Execute** dell'istruzione:

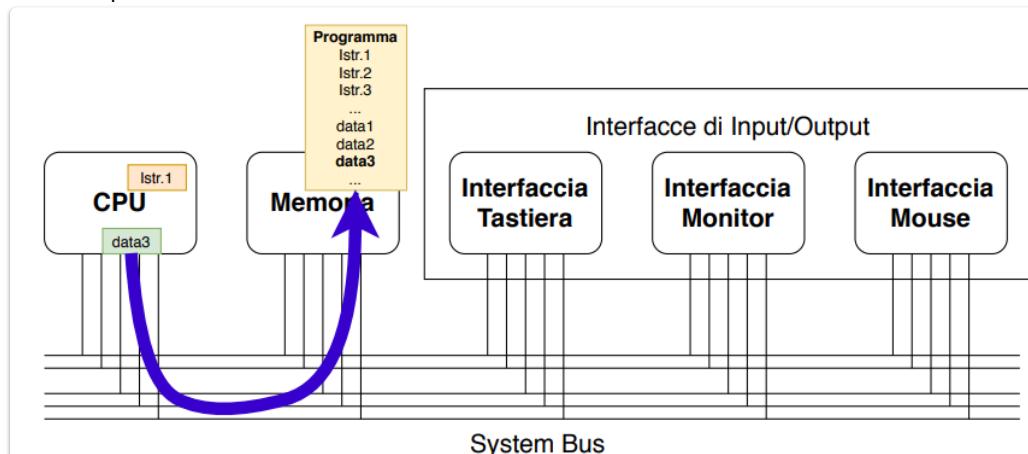


- l'istruzione viene eseguita all'interno dalla CPU
- **Exceptions:**
  - *se* l'istruzione richiede un'interfaccia di I/O:



- il dato viene "letto" (*input*) da una periferica
- il dato viene "scritto" (*output*) su una periferica

- *se* l'istruzione produce un nuovo dato:



- viene "*scritto*" in memoria

## 2. Algebra Booleana

### Definizione e Operazioni

L'**Algebra di Boole** (o **Booleana**) è un *sistema formale* (algebra) utilizzato per la *logica proposizionale* (utilizzato nei calcolatori perché rappresenta ON e OFF).

E' formato da:

- $S: \{0, 1\} \rightarrow$  rispettivamente OFF e ON
  - Operazioni:
    - "binarie":
      - somma (logica)
      - prodotto (logico)
    - "unarie"
      - negazione (logica)
- | Il comportamento delle operazioni viene definito da **tabelle di verità**

## Somma Logica

Viene anche chiamata OR perché  $y = 1 \Leftrightarrow x_1 = 1$  oppure  $x_2 = 1$ :

$x_1$	$x_2$	$y = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

- è indicata con i simboli  $+$ , OR,  $\vee$

## Proprietà

1. **Commutativa**:  $x_1 + x_2 = x_2 + x_1$
2. **Associativa**:  $x_1 + x_2 + x_3 = (x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$
3. **Idempotenza**:  $x + x = x$
4. **Elemento neutro**:  $x + 0 = x$
5. **Massimo**:  $x + 1 = 1$
6. **Complemento**:  $x + \bar{x} = 1$

## Prodotto Logico

Viene anche chiamata AND perché  $y = 1 \Leftrightarrow x_1 = 1$  e  $x_2 = 1$ :

$x_1$	$x_2$	$y = x_1 \cdot x_2 = x_1 x_2$
0	0	0
0	1	0
1	0	0
1	1	1

- è indicata con i simboli  $\cdot$ , AND,  $\wedge$

## Proprietà

1. **Commutativa:**  $x_1 * x_2 = x_2 * x_1$
2. **Associativa:**  $x_1 * x_2 * x_3 = (x_1 * x_2) * x_3 = x_1 * (x_2 * x_3)$
3. **Idempotenza:**  $x * x = x$
4. **Elemento neutro:**  $x * 1 = x$
5. **Minimo:**  $x * 0 = 0$
6. **Complemento:**  $x * \bar{x} = 0$

## Negazione

$x$	$y = \bar{x}$
0	1
1	0

- è indicata con i simboli  $\bar{x}$ , NOT,  $\neg$

## Proprietà

1. Doppia negazione:  $\bar{\bar{x}} = x$

## Funzioni Logiche

Una *funzione logica o booleana*  $f(x_1, x_2, \dots, x_n)$  è un'espressione più o meno complessa che utilizza gli *operatori booleani* sulle variabili  $x_1, x_2, \dots, x_n$ :

$$f(x_1, x_2, x_3) = x_1 + \bar{x}_2 x_3$$

$x_1$	$x_2$	$x_3$	$x_2 x_3$	$\bar{x}_2 x_3$	$x_1$	$x_1 + \bar{x}_2 x_3$
0	0	0	0	1	0	1
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	0	1	1	1
1	1	1	1	0	1	1

## Teorema di De Morgan

- $\overline{x_1 + x_2} = \bar{x}_1 * \bar{x}_2$
- $\overline{x_1 * x_2} = \bar{x}_1 + \bar{x}_2$

## Sintesi di Funzioni logiche

Da una tabella di verità è possibile derivarne una *funzione logica* attraverso:

- Somme di prodotti:

$x_1$	$x_2$	$y = f(x_1, x_2)$	Prodotto
0	0	1	$\overline{x_1} \overline{x_2}$
0	1	0	
1	0	0	
1	1	1	$x_1 x_2$

$$f(x_1, x_2) = \overline{x_1} \overline{x_2} + x_1 x_2$$

- si individuano **tutti i casi** in cui  $y = 1$
- per ogni **caso** si costruisce un prodotto delle  **$n$**  variabili (**mintermine**) in cui ogni  $x_i$  è presa così com'è se  $x_i = 1$ , negata se  $x_i = 0$
- si **sommano** i prodotti ottenuti

- Prodotti di somme:

$x_1$	$x_2$	$y = f(x_1, x_2)$	Somme
0	0	1	
0	1	0	$x_1 + \overline{x_2}$
1	0	0	$\overline{x_1} + x_2$
1	1	1	

$$f(x_1, x_2) = (\overline{x_1} + x_2)(x_1 + \overline{x_2})$$

- si individuano **tutti i casi** in cui  $y = 0$
- per ogni **caso** si costruisce un prodotto delle  **$n$**  variabili (**mintermine**) in cui ogni  $x_i$  è presa così com'è se  $x_i = 0$ , negata se  $x_i = 1$
- si **moltiplicano** le somme ottenute

La **Sintesi di una Funzione Logica** non garantisce che sia il numero minimo di operazioni richieste per ottenere tale risultato, si effettua perciò un processo di **minimizzazione**

Minimizzazione **tramite Mappe di Karnaugh**

$x_1$	$x_2$	$x_3$	$y = f(x_1, x_2, x_3)$	Prodotti
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{x}_1 \ x_2 \ x_3$
1	0	0	0	
1	0	1	0	
1	1	0	1	$x_1 \ x_2 \ \overline{x}_3$
1	1	1	1	$x_1 \ x_2 \ x_3$

$$\begin{aligned}
 f(x_1, x_2, x_3) &= \overline{x}_1 \ x_2 \ x_3 + x_1 \ x_2 \ \overline{x}_3 + x_1 \ x_2 \ x_3 = \\
 &= \overline{x}_1 \ x_2 \ x_3 + x_1 \ x_2 (\overline{x}_3 + x_3) = \\
 &= \overline{x}_1 \ x_2 \ x_3 + x_1 \ x_2 \ 1 = \\
 &= \overline{x}_1 \ x_2 \ x_3 + x_1 \ x_2
 \end{aligned}$$

Le **Mappe di Karnaugh** sono uno strumento grafico che permette la sintesi di una funzione logica in *forma minima*.

Raggruppando quadrati che rappresentano la stessa uscita, è possibile identificare quali variabili non contribuiscono e pertanto eliminarle nella forma analitica.

Sono formate da un **reticolo** in cui:

- le righe e le colonne = valori *variabili di input*
- la sequenza delle variabili deve essere tale che *due righe (colonne) differiscono nel valore di una sola variabile*
- il contenuto dei quadrati = valore *variabile di output*

Esempio:

$x_1$	$x_2$	$x_3$	$y = f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

		$x_1 x_2$	
		00	01
		11	10
$x_3$	0	0	0
	1	0	1

		$x_1 x_2$	
		00	01
		11	10
$x_3$	0	0	0
	1	0	1

● Box **rosso**:  $x_1$  varia, quindi **non contribuisce**,  $x_2 = 1, x_3 = 1 \Rightarrow x_2 x_3$

● Box **verde**:  $x_3$  varia, quindi **non contribuisce**,  $x_1 = 1, x_2 = 1 \Rightarrow x_1 x_2$

$$f(x_1, x_2, x_3) = x_1 x_2 + x_2 x_3$$

Casi speciali:

$x_1$	$x_2$	$x_3$	$y = f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

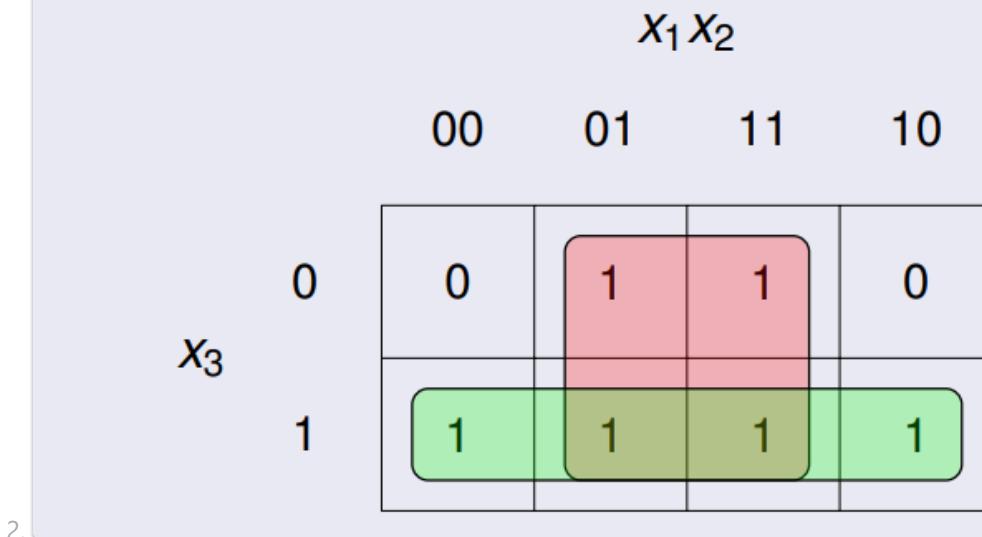
$x_1 x_2$

00      01      11      10

0	0	1	0
1	1	0	1

1.

$x_1$	$x_2$	$x_3$	$y = f(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Per la **realizzazione** delle operazioni booleane si fa l'utilizzo di "[5. Porte Logiche e Circuiti Combinatori](#)"

Come rappresentare il vasto range di numeri con soli due simboli? Esistono [3. I Sistemi di Numerazione](#):

- Grazie ad un **sistema di numerazione binaria** che permette, tramite differenti combinazioni di 0 e 1, di rappresentare **numeri** (interi, reali, ecc...)

### 3. I Sistemi di Numerazione

**Costituito** da:

- 10 simboli: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- ogni simbolo rappresenta una **quantità** ben precisa
- ↑ dipende anche dalla **posizione** del simbolo

### Definizione e Proprietà

Un **sistema di numerazione** è definito da:

- **base B**, intera
- insieme di B simboli:  $S = \{s_0, \dots, s_{B-1}\} \rightarrow 0, 1, 2, \dots, B-1$
- un numero a **n cifre**  $p_{n-1}p_{n-2} \dots p_1p_0$ :
  - (i) posizione della cifra = **peso**
  - cifra più a **destra** → cifra **meno** significativa

- cifra più a **sinistra** → cifra **più** significativa

-  $p_i \in S$

-

$$\sum_{i=0}^{n-1} p_i \cdot B^i$$

- il numero è espresso come **somma di potenze della base**

Considerando l'[2. Algebra Booleana](#):

- $(1100)_2 = 0 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 = 4 + 8 = (12)_{10}$

## Proprietà

### 1. Prodotto per potenze della base

- sia dato  $(n)_B$  e un esponente  $k \geq 0$ :
  - $(n)_B * B^k = (n)_B 0_1 \dots 0_k$

$$(37)_{10} \cdot (100)_{10} = (3700)_{10}$$

$$(3B)_{16} \cdot (16)_{10} = (3B0)_{16}$$

$$(1101)_2 \cdot (8)_{10} = (1101)_2 \cdot (2^3)_{10} = (1101000)_2$$

•

### 2. Divisione intera per potenze della base

- sia dato  $(n)_B$  e un esponente  $k \geq 0$ :
  - $(n)_B / B^k = (n)_B - p_1 \dots p_k$

$$(37)_{10} / (10)_{10} = (3)_{10}$$

$$(3B)_{16} / (16)_{10} = (3)_{16}$$

$$(10101)_2 / (8)_{10} = (10101)_2 / (2^3)_{10} = (10)_2$$

•

### 3. Intervallo rappresentabile con $k$ cifre

- sia data una base  $B$  e  $k \in N$ :
  - con  $k$  cifre è possibile rappresentare  $B^k$  interi  $\rightarrow [0, B^k - 1]$

## Alcuni Sistemi di Numerazione

### Sistema a Esadecimale

Costituito da:

- **16 simboli**
- $S = \{0, \dots, 9, A, B, C, D, E, F\}$

*Base 16 → Base 10 = OK*

*Base 16 → Base 2 = NOPE*

### Sistema Binario

Sia data una base  $B = 2^k, k \in N$ :

- $[0, 2^k - 1] \equiv [0, B - 1]$

Esempio:

Base 8 = $2^3$	Base 2
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Base 16 = $2^4$	Base 2
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

## Aritmetica Binaria

### Somma

0	+	0	=	0
0	+	1	=	1
1	+	0	=	1
1	+	1	=	0 con riporto di 1

Esempio:

Riporti	1	1				
Primo addendo	1	0	0	1	+	
Secondo addendo	1	0	0	1	1	=
Risultato	1	1	1	0	0	

$$(1001)_2 + (10011)_2 = (11100)_2$$

$$9 + 19 = 38$$

## Conversioni

### Conversione da Base 10 a Base B

Si utilizza la tecnica della *divisioni successive* (Divisioni con Resto):

1. sia  $(N)_{10}$  il numero da convertire
  2. si calcola la *divisione intera*  $N = N/B$  e si mette da parte il **resto R**
  3. se  $N > 0$  si ritorna al passo 2.
  4. se  $N = 0$  si riportano i vari resti da destra → sinistra: **rappresentano** il numero convertito in base B
4. Rappresentazione dell'Informazione

## Definizioni e Terminologie

Una cifra binaria è detta **BIT** (BInary digiT). Sulla base di ciò si può dire che su  $k$  *collegamenti elettrici* si possono rappresentare  $2^k$  bit → se  $k = 1 \rightarrow 1$  bit

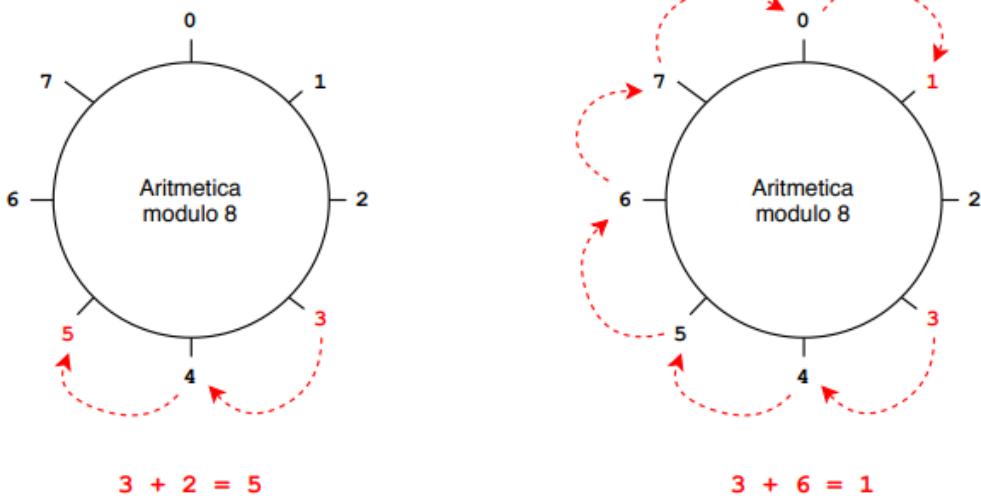
1. Una combinazione di 8 bit è detta **BYTE**. Il suo intervallo è  $[0, 2^8 - 1]$ 
  - se  $k = 8 \rightarrow 8$  bit - 1 byte
2. Una combinazione di 16 bit è detta **WORD**. Il suo intervallo è  $[0, 2^{16} - 1]$ 
  - se  $k = 16 \rightarrow 16$  bit - 2 byte - 1 word
3. Una combinazione di 32 bit è detta **DOUBLE WORD**. Il suo intervallo è  $[0, 2^{32} - 1]$ 
  - se  $k = 32 \rightarrow 32$  bit - 4 byte - 2 word

## Aritmetica Modulare

E' l'aritmetica che viene utilizzata quando l'insieme di riferimento è **finito**. Essa fa uso dell'operazione di **modulo** = *resto della divisione intera*:

- $a \bmod n =$  resto della divisione tra  $a$  e  $n$
- Introduce il concetto di **congruenza modulo n**:
- $a \equiv b \pmod n \Leftrightarrow (a - b)$  multiplo di  $n$

**Esempio:**



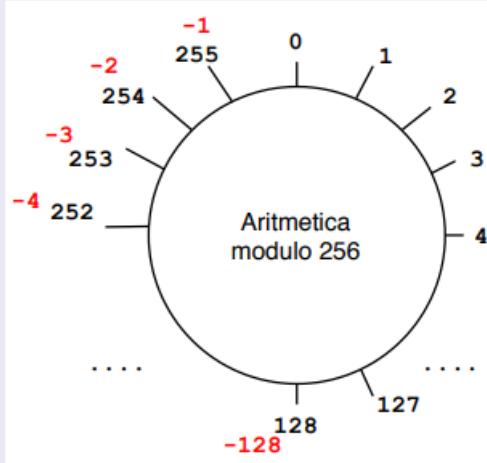
## Complemento a 2

Il complemento a 2 permette di rappresentare un sottoinsieme dell'insieme  $\mathbb{Z}$   
(*numeri interi relativi* → positivi e negativi)

Con  $k$  bit si possono rappresentare  $2^k$  in  $\mathbb{Z}$ :

- $[-2^{k-1}, 2^{k-1} - 1] \subset \mathbb{Z}$
- Il **Most Significant Bit** (più a *sinistra*) permette di distinguere i *positivi* dai *negativi*:
  - MSB = 0 → *positivo*
  - MSB = 1 → *negativo*

Esempio:



Binario	Decimale	Numero rappresentato
1000 0000	128	-128
...	...	...
1111 1101	253	-3
1111 1110	254	-2
1111 1111	255	-1
0000 0000	0	0
0000 0001	1	+1
0000 0010	2	+2
...	...	...
0111 1111	127	+127

### Problemi

Dati  $k$  bit:

1. numeri positivi:  $[1, 2^{k-1} - 1]$
2. numeri negativi:  $[2^{k-1}, -1]$
3. in caso di **overflow** il risultato sarà corretto *ma negativo*

### Operazioni

sia  $x \in \mathbb{N}$  rappresentato in binario in *complemento a 2* a  $k$  bit

1. *negare* tutti i  $k$  bit
2. *sommare* (aritmeticamente) 1
3. *troncare* i bit in eccesso
4. risultato =  $-x$

**Esempio:**

Numero	Binario	
5	0000 0101	inversione
	1111 1010 + 1 =	somma di 1
-5	1111 1011	

- *Verifica:*

5 +	1 1111 111	+
-5 =	0000 0101	=
0	1111 1011	=
0	0000 0000	=

## Rappresentazione dei Reali

Si ha la possibilità di rappresentare un sottoinsieme di  $\mathbb{R}$  in cui ogni numero rappresentato verrà *approssimato* al valore più vicino rappresentabile.

Le tecniche di rappresentazione sono:

- virgola **fissa**:

- *Fixed Point*

### Fixed Point, $k = 8, r = 4$

$$00110110 = 0011.0110 = 2^1 + 2^0 + 2^{-2} + 2^{-3} \\ = 3 + 0.25 + 0.125 \\ = 3.375$$

- fissato un numero di  $k$  bit, si *stabilisce* una posizione prefissata della virgola
- si sceglie un numero  $r$  di bit da usare per *rappresentare la parte frazionaria* di un numero
- questa parte sarà rappresentata da  $k - r$  bit
- le cifre dopo la virgola avranno *peso* ( $\text{left} \rightarrow \text{right}$ ):  $2^{-1}, 2^{-2}, \dots, 2^{-r}$
- Conversione *Reale*  $\rightarrow$  *Binario*
  - la *parte intera* si converte usando le divisioni successive
  - la *parte frazionaria* si segue l'*algoritmo*:
    - si *moltiplica* il numero per 2
    - si *estrae* la parte intera del risultato (cifra: 0, 1)  $\rightarrow$  e. g. 0.5
    - si *estrae* decimale del risultato e se ! = 0 si ritorna al passo (1.)  $\rightarrow$  e. g. 0.5
    - le parti intere estratte si ricopiano da sinistra verso destra
- *Esempio:*

## Fixed Point, $k = 8, r = 4$

3.25	3	1	0.25	0.5	0	
	1	1	0.5	1.0	1	(0011.0100) <sub>2</sub>
	0		0			

3.6	3	1	0.6	1.2	1	
	1	1	0.2	0.4	0	(0011.1001) <sub>2</sub>
	0		0.4	0.8	0	
	0.8		1.6	1		
	0.6		...	...		

$$\begin{aligned}
 00111001 &= 0011.1001 = 2^1 + 2^0 + 2^{-1} + 2^{-4} \\
 &= 3 + 0.5 + 0.0625 \\
 &= 3.5625
 \end{aligned}$$

- virgola mobile

- Floating Point



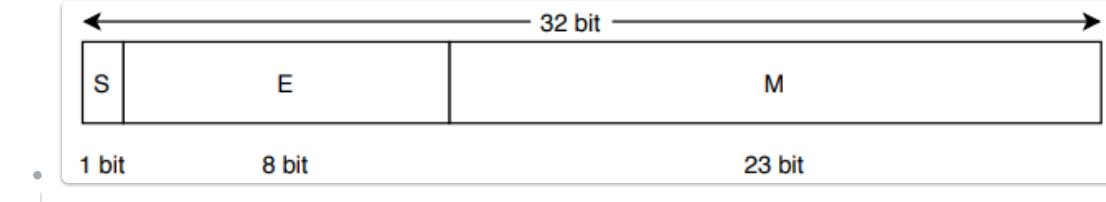
- fissato un numero di  $k$  bit si stabiliscono:

- $m$  bit di mantissa

- $k - m$  bit di esponente

$$\text{valore} = M \cdot 2^E$$

- IEEE 754  $(-1)^S \cdot 1.M \cdot 2^{E-127}$



- 1 bit

- 8 bit

- 23 bit

- $+/- = 1$  bit

- esponente = 8 bit (con codifica a eccesso 127)

- mantissa = 23 bit (con parte intera pari a 1)

- Esempio:

$$(BF\ A0\ 00\ 00)_{16}$$

1011 1111 1010 0000 0000 0000 0000 0000

1	01111111	01000000000000000000000000000000
---	----------	----------------------------------

$$(-1)^1 \times (1.01)_2 \times 2^{127-127}$$

$$-(1.01)_2 = -1.25$$

$(C0\ 53\ 33\ 33)_{16}$

1100 0000 0101 0011 0011 0011 0011 0011

1	10000000	1010011001100110011001100110011
---	----------	---------------------------------

$(-1)^1 \times (1.10100110011001100110011)_2 \times 2^{128-127}$

$-(1.10100110011001100110011)_2 \times 2$

$-(11.0100110011001100110011)_2 \cong -3.3$

5.1.0 M, E, D

[5. Porte Logiche e Circuiti Combinatori](#)

## 5.1 Multiplexer

Un **multiplexer**  $K - a - 1$  è una rete logica che possiede:

- $K = 2^n$  input dati  $\rightarrow x_0, x_1, \dots, x_{K-1}$
  - $n$  input di controllo
  - una singola uscita
- Permette di "connettere" uno solo degli ingressi  $x_i$  all'uscita  $y$  sulla base degli  $n$  bit di controllo:
- $S$  è il valore decimale rappresentato dagli  $n$  bit
  - $x_S$  verrà copiato sull'uscita
  - tutti gli altri ingressi saranno ignorati

### Tabella di verità

Multiplexer 4 –  $a – 1$ :

Tabella:

$x_0$	$x_1$	$x_2$	$x_3$	$c_1$	$c_0$	$y$
0	X	X	X	0	0	0
1	X	X	X	0	0	1
X	0	X	X	0	1	0
X	1	X	X	0	1	1
X	X	0	X	1	0	0
X	X	1	X	1	0	1
X	X	X	0	1	1	0
X	X	X	1	1	1	1

Il valore  $X$  rappresenta il **don't care** significa che l'uscita  $y$  è indipendente dal valore della variabile  $x_K = X$  che può assumere qualunque valore

La **funzione logica** che ne rappresenta il comportamento è:  $y = x_0\bar{c}_1\bar{c}_0 + x_1\bar{c}_1c_0 + x_2c_1\bar{c}_0 + x_3c_1c_0$

## 5.2 L'Encoder

Un **encoder**  $K - to - n$  è una rete logica che possiede:

- $K = 2^n$  input dati  $\rightarrow x_0, x_1, \dots, x_{K-1}$

- $n$  output  $y_{n-1}, \dots, y_0$   
Permette di generare, sulle uscite  $y_n$ , il *pattern binario* che rappresenta il *numero dell'ingresso*  $x_K = 1$ :
- nel caso in cui molteplici  $x_K = 1$ :
  - si può dare priorità all'ingresso di ordine più basso
  - si può includere un ulteriore uscita che segnala un'eventuale condizione non valida

### Tabella di verità

Encoder 8-to-3:

Tabella:

$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$y_2$	$y_1$	$y_0$
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

### 5.3 II Decoder

Un **decoder**  $n$ -to- $K$  è una rete logica che possiede:

- $n$  input  $x_{n-1}, \dots, x_0$
- $K = 2^n$  output  $\rightarrow y_0, y_1, \dots, y_{K-1}$   
Permette di svolgere il *comportamento inverso* rispetto all'Encoder. Attiva l'uscita  $y$  sulla base del pattern binario degli ingressi al valore  $x_i$

### Tabella di verità

Decoder 3-to-8:

Tabella:

$x_2$	$x_1$	$x_0$	$y_0$	$y_1$	$y_2$	$y_3$	$y_4$	$y_5$	$y_6$	$y_7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

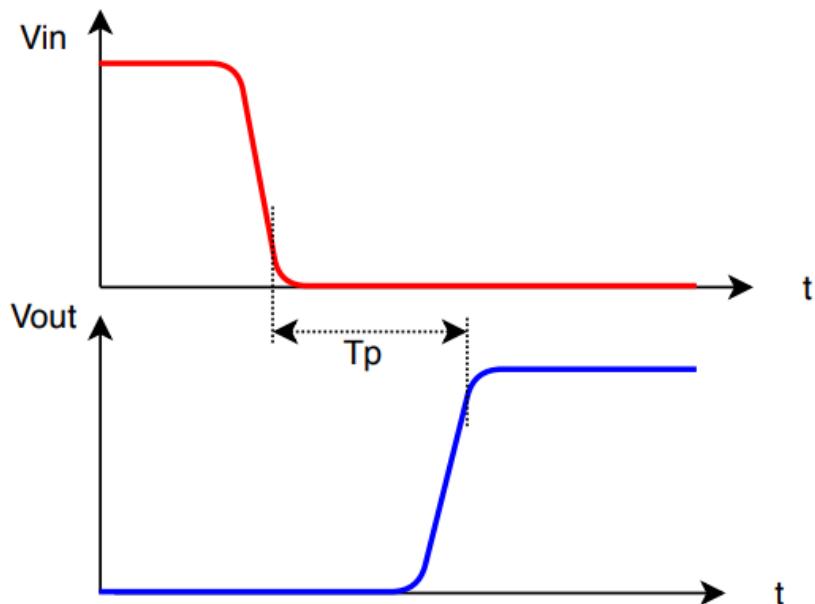
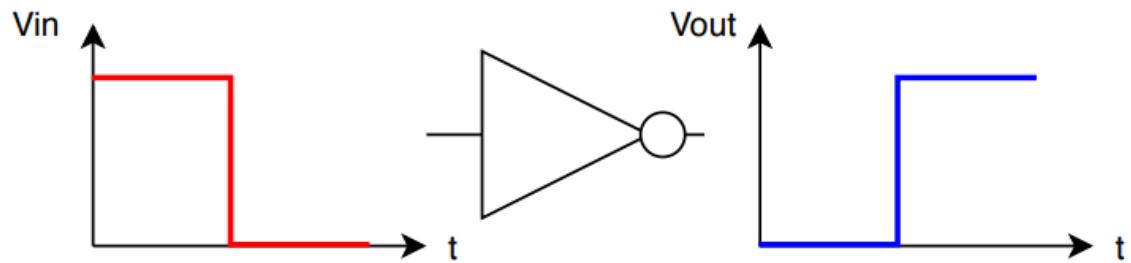
## 6. Aspetti Elettronici delle Porte Logiche

- Definizione: [5. Porte Logiche e Circuiti Combinatori](#)

## Ritardo di propagazione

La **commutazione** del transistor MOS a seguito di un comando elettrico *non avviene instantaneamente* ma richiede un **tempo (o ritardo) di propagazione** nell'ordine  $10^{-9}s$ :

- I **fronti** sono variazioni 1-to-0 e 0-to-1 non perfettamente verticali
- il segnale si propaga in uscita dopo il **tempo di propagazione**  $T_p$

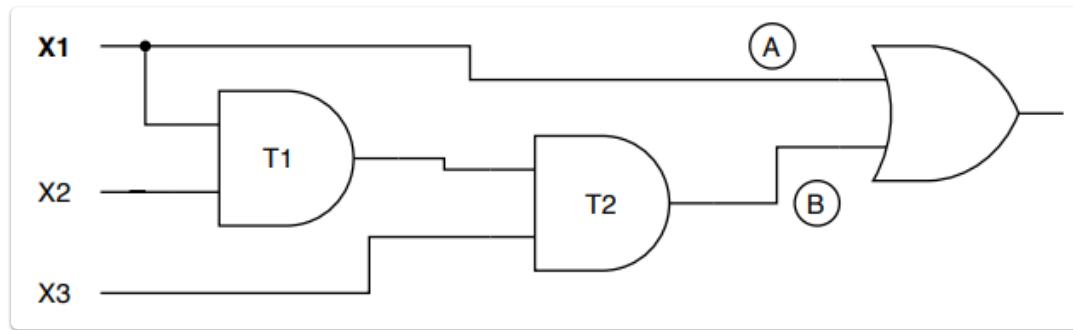


## Problemi

E' di buona norma cercare di bilanciare le reti in modo che i rami dei circuiti all'ingresso di una porta logica finale non abbiano tempo di propagazione diversi

Nel circuito d'esempio X1 arriva direttamente al collegamento A mentre dopo  $T_1 + T_2$  al collegamento B

Circuito:



## Transistor MOS

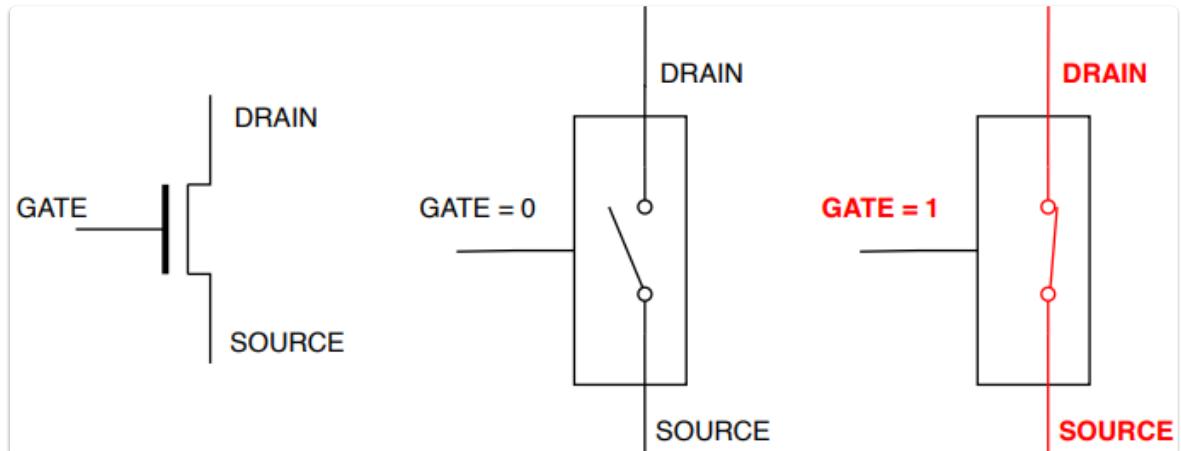
E' un componente elettronico dotato di tre collegamenti:

- Gate
- Source
- Drain

e agisce da interruttore elettronico collegando Source e Drain sulla base del valore logico applicato al Gate

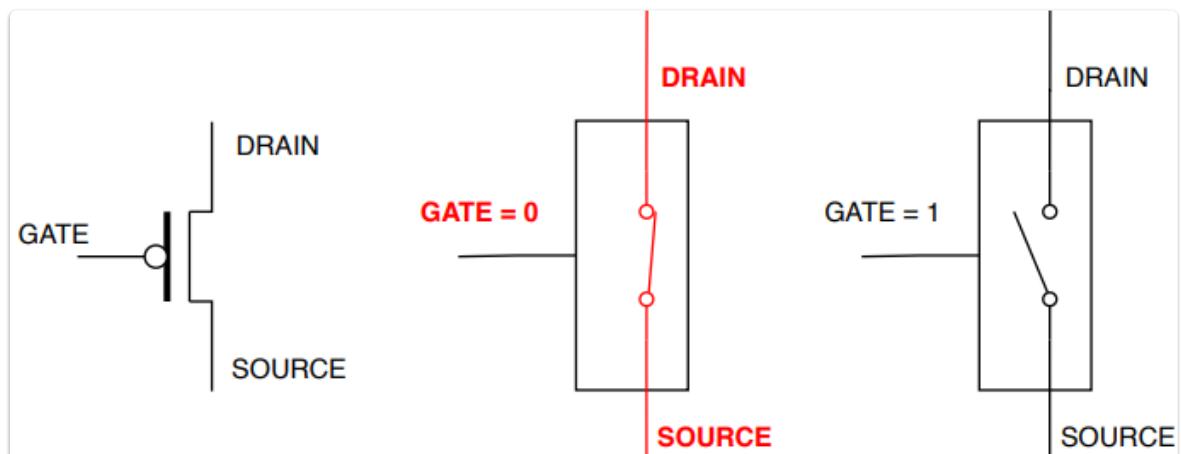
Esistono due tipi di MOS:

- NMOS



- il collegamento si *apre* se il Gate = 0
- il collegamento si *chiude* se il Gate = 1

- PMOS

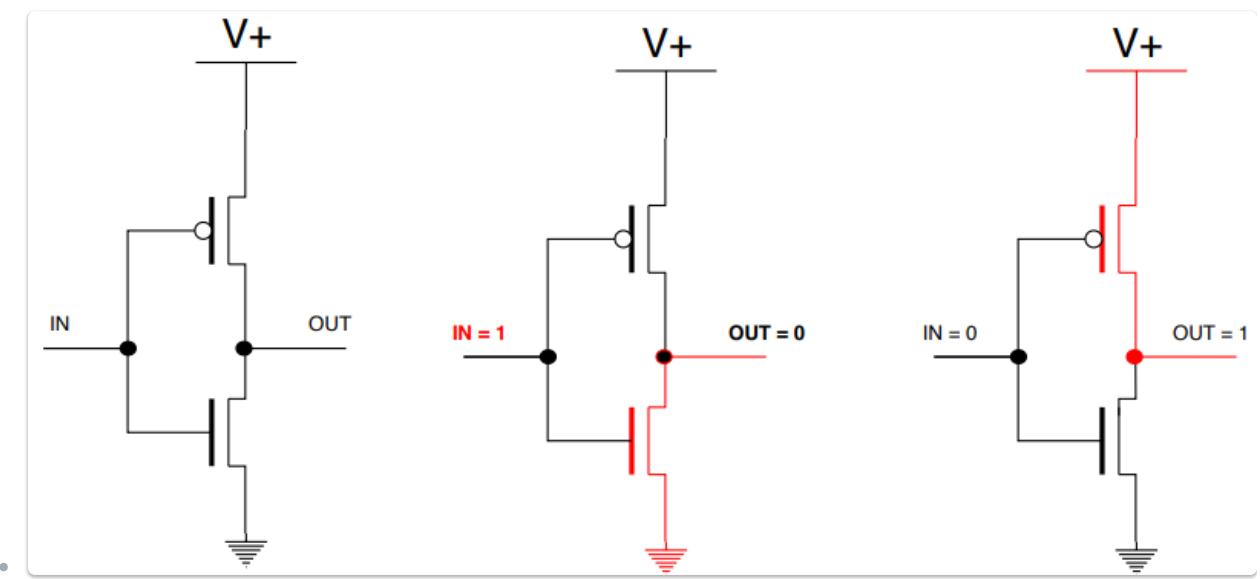


- il collegamento si *chiude* se il Gate = 0
- il collegamento si *apre* se il Gate = 1

## Inverter a MOS

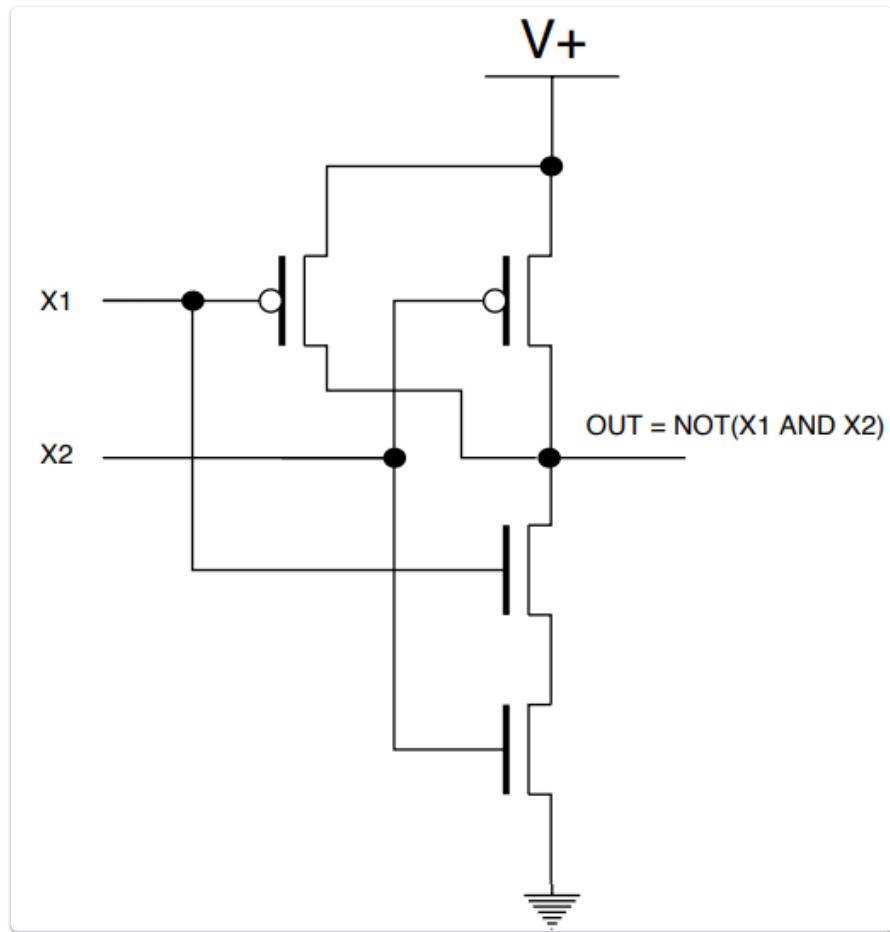
Fa l'utilizzo di *Transistor PMOS* e *Transistor NMOS*

Circuito:



## Porta NAND a MOS

Circuito:



## 7. Circuiti Sequenziali

### Definizione e Circuito di Base

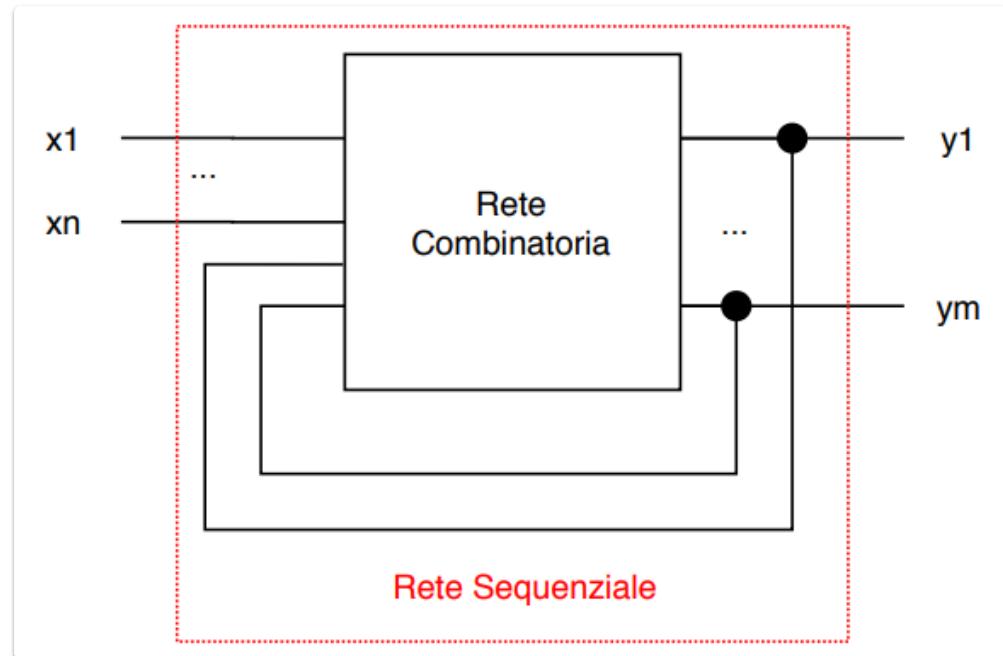
Sono **circuiti**, reti di **porte logiche**, in cui le uscite \*dipendono anche dal **tempo**\*.

Le uscite all'istante di tempo  $t^*$  dipendono:

- dal valore degli ingressi all'istante  $t^*$

- dal valore delle uscite stesse agli istanti precedenti,  $t < t^*$   
dunque dagli **ingressi e dalle uscite stesse**.

Ciò viene ottenuto aggiungendo ad una rete combinatoria un meccanismo a **feedback**:



La **tabella di verità** per un circuito sequenziale come quello d'esempio include:

- i valori degli **ingressi**
- i valori delle **uscite all'istante precedente**
- i nuovi valori delle **uscite all'istante attuale**

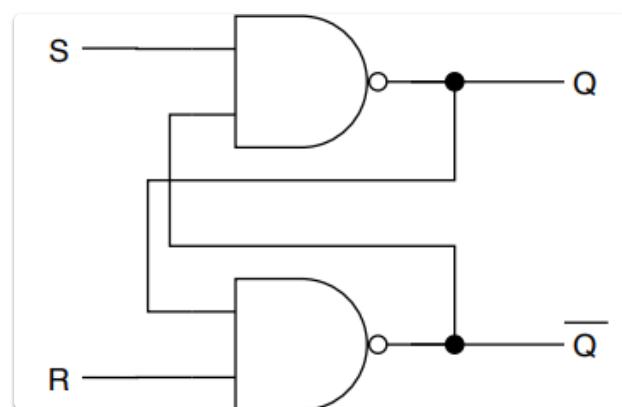
**Tabella:**

$x_1$	...	$x_n$	$y_1(t-1)$	...	$y_m(t-1)$		$y_1(t)$	...	$y_m(t)$
0	...	0	0	...	0		0	...	1
0	...	0	0	...	1		0	...	1
1	...	0	0	...	1		1	...	0
1	...	0	1	...	0		1	...	0

## Flip-Flop Set-Reset

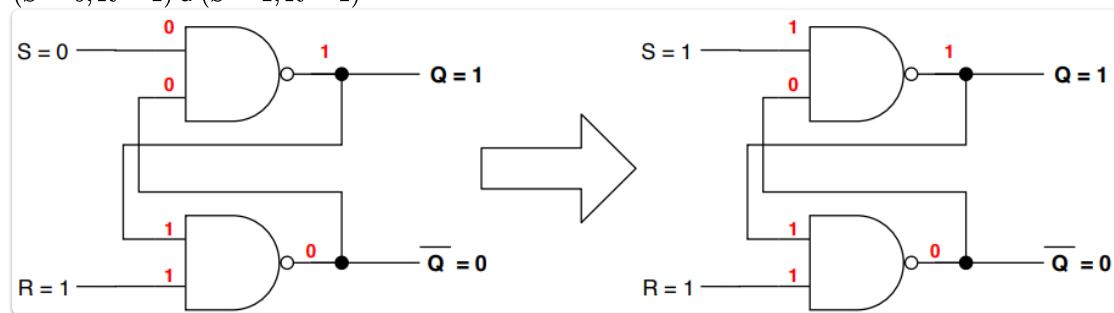
Il nome indica che si tratta di un circuito in grado di commutare tra due stati: **flip<=>flop**.  
**Set-Reset** indicano due operazioni rispettivamente di set e reset

**Circuito:**

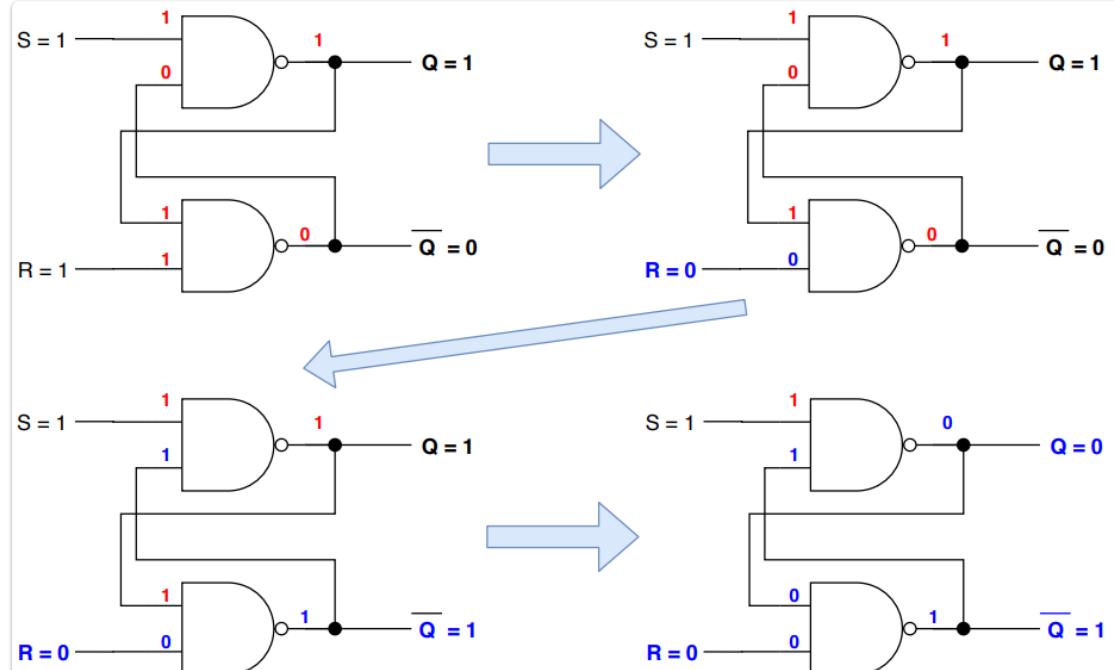


- Considerando il passaggio degli ingressi:

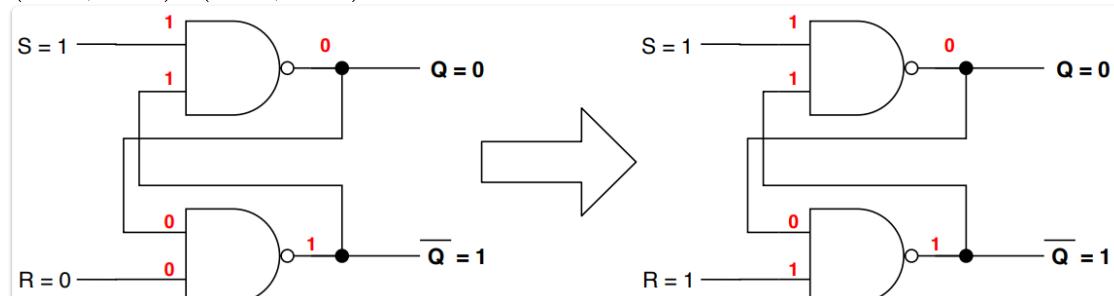
- $(S = 0, R = 1) \rightarrow (S = 1, R = 1)$



- $(S = 1, R = 1) \rightarrow (S = 1, R = 0)$



- $(S = 1, R = 0) \rightarrow (S = 1, R = 1)$



- Tabella di verità:

$S$	$R$	$Q(t)$	$\bar{Q}(t)$	
0	1	1	0	Stato di "Set"
1	0	0	1	Stato di "Reset"
1	1	$Q(t - 1)$	$\bar{Q}(t - 1)$	Stato mantenuto
0	0	1	1	instabile!

### Flip-Flop Set-Reset "Gated"

Si modifica il FF-SR aggiungendo altre due porte NAND che vincolano gli ingressi ed un ulteriore ingresso chiamato **clock**, che quando esso vale:

- $0 \rightarrow S^* = 1$  e  $R^* = 1$  nessun effetto sullo stato flip-flop qualunque variazione sugli ingressi S e R
- $1 \rightarrow S^* = \bar{S}$  e  $R^* = \bar{R}$  potrà cambiare stato del flip-flop

Circuito:

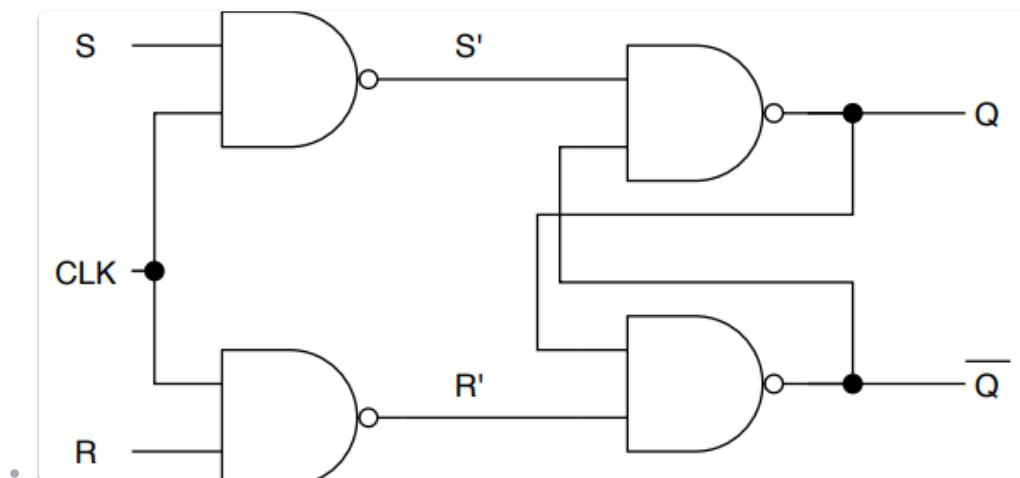


Tabella di verità

$S$	$R$	$Clk$	$S'$	$R'$	$Q(t)$	$\bar{Q}(t)$	
X	X	0	1	1	$Q(t-1)$	$\bar{Q}(t-1)$	Stato mantenuto
1	0	1	0	1	1	0	Stato di "Set"
0	1	1	1	0	0	1	Stato di "Reset"
0	0	1	1	1	$Q(t-1)$	$\bar{Q}(t-1)$	Stato mantenuto
1	1	1	0	0	1	1	instabile!

### D-Type FF-SR

Attraverso una modifica al FF-SR, è possibile ottenere un circuito sequenziale in grado di *memorizzare il valore d'uscita all'istante*  $Q(t - 1)$

Circuito:

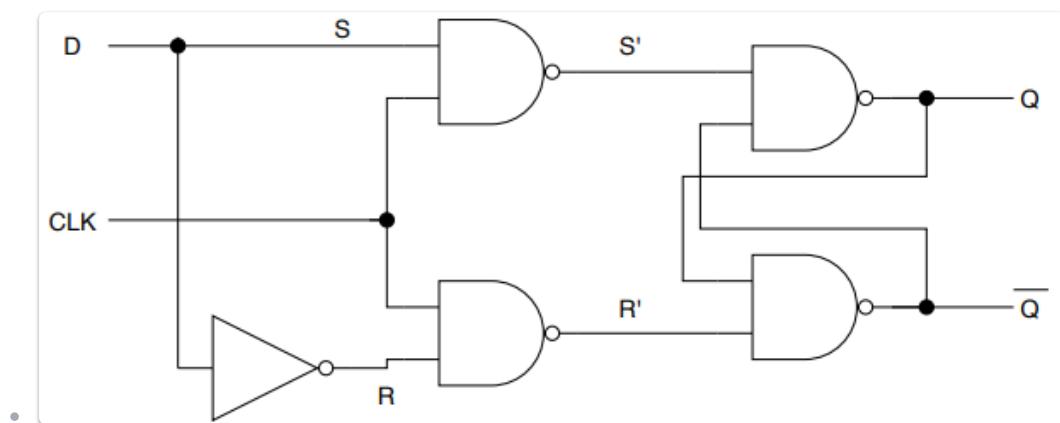


Tabella di verità:

$D$	$Clk$	$S$	$R$	$S'$	$R'$	$Q(t)$	$\bar{Q}(t)$	
X	0	$D$	$\bar{D}$	1	1	$Q(t-1)$	$\bar{Q}(t-1)$	Stato mantenuto
0	1	0	1	1	0	0	1	Stato di "Reset"
1	1	1	0	0	1	1	0	Stato di "Set"

Si può notare:

- che quando il **clock** = 1  $Q(t) = D$
- quando il **clock** = 0,  $Q(t) = Q(t - 1)$  e il circuito opera da *memoria ad un bit*

## 8 Bit Register

Collegando in parallelo 8 D-Type FF-SR si ottiene un *elemento di memoria a 8 bit* (registro a 8 bit)

Circuito:

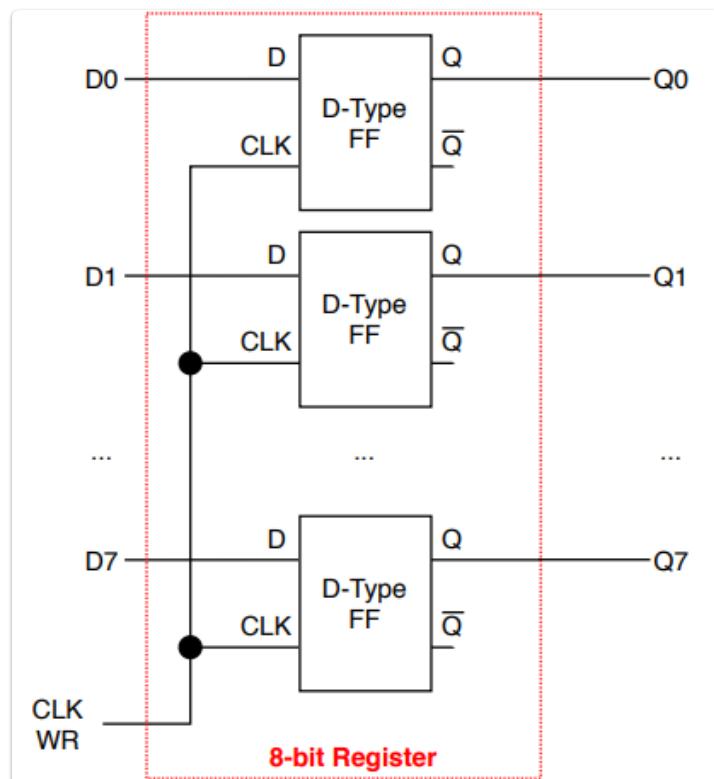


Tabella di verità

$D_n$	$WR$	$Q_n(t)$
X	0	$Q_n(t - 1)$
X	1	$D_n$

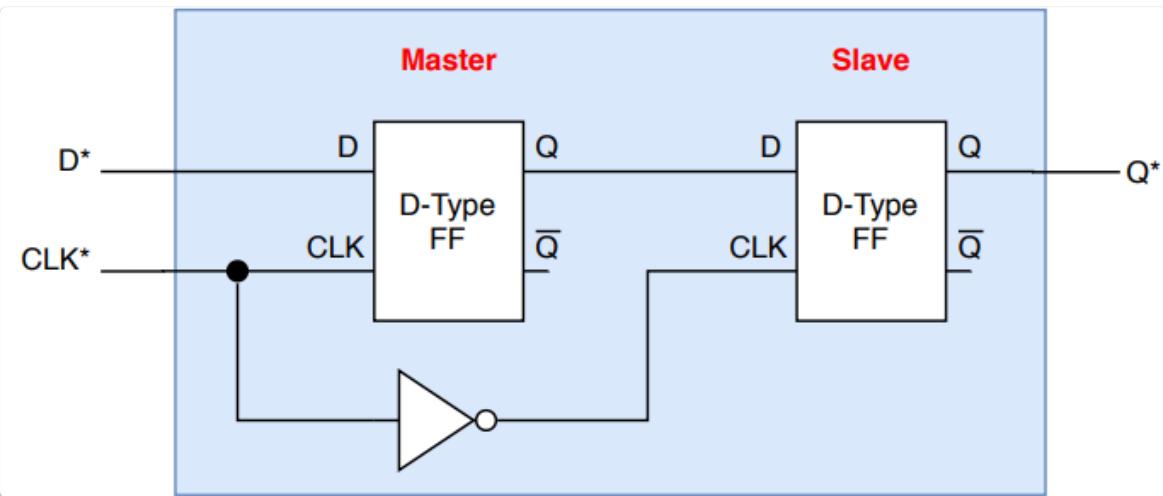
## Edge Triggering e D-Type Master-Slave

Per un D-Type-FF, il clock *memorizza* il valore  $D$  quando  $\text{clock} = 1$ . Ma se mentre il clock = 1 e il valore di  $D$  cambia, *l'uscita cambierà* di conseguenza.

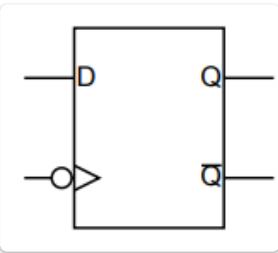
Tuttavia, se *colleghiamo in serie* due D-Type-FF otteniamo che:

- $CLK^* = 1 \rightarrow$  il segnale su  $D^*$  si propaga sull'uscita del **Master** FF
- $CLK^* = 1 \rightarrow$  il segnale su  $D^*$  si propaga sull'uscita del **Master** FF fino all'uscita dello **Slave** FF e quindi su  $Q^*$

il circuito complessivo memorizza  $D^* \rightarrow Q^*$  quando  $CLK^* = 1 \rightarrow 0$ , su un fronte di *discesa del segnale*:

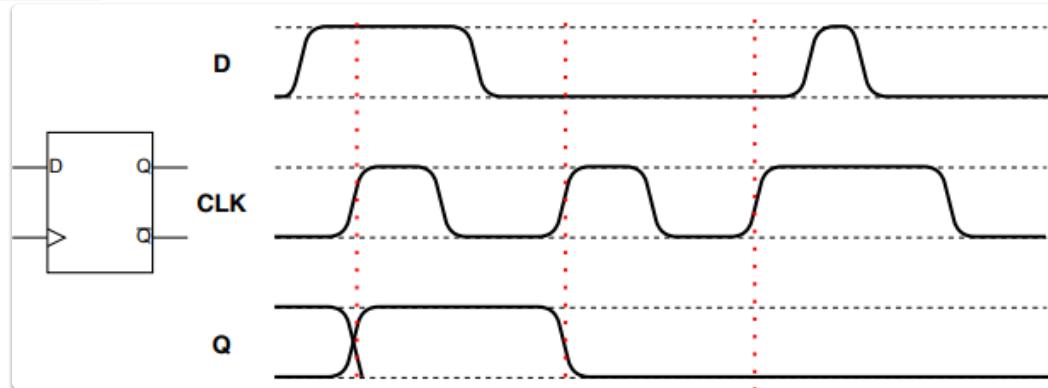


Questo circuito, che *opera su un fronte del segnale*, può essere rappresentato con il *simbolo*:

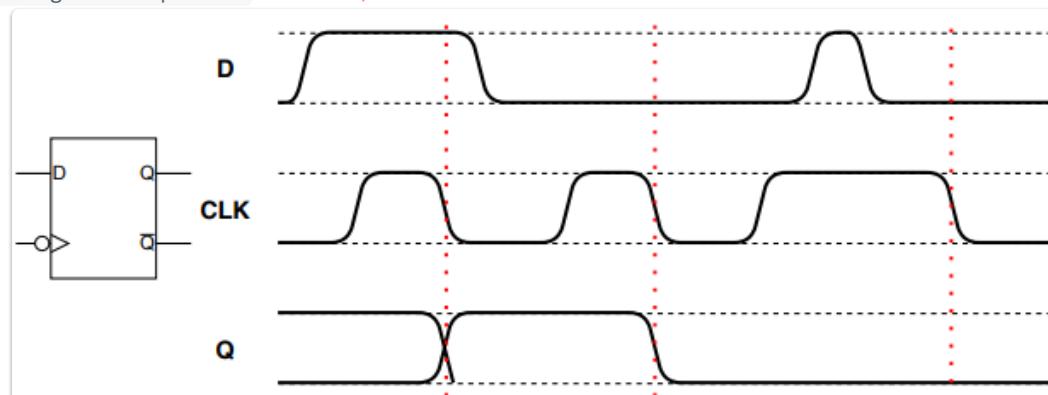


che indica un FF di tipo *edge-triggered* in cui:

- il "triangolo" indica il *fronte di salita*  $CLK = 0 \rightarrow 1$



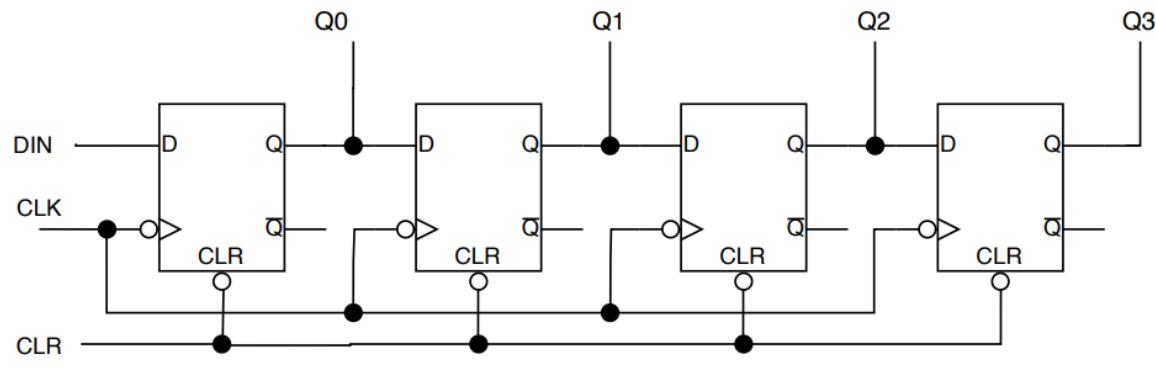
- il "triangolo con pallino" indica il *fronte di discesa*  $CLK = 1 \rightarrow 0$



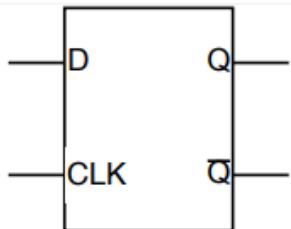
## Shift Register

Attraverso dei D-Type-FF *collegati in serie* (*serial-to-parallel*) è possibile implementare uno *shift register* ad  $n$  bit

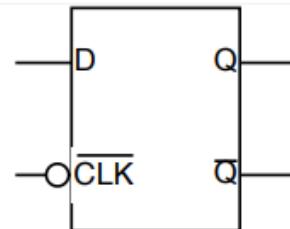
**Circuito:**



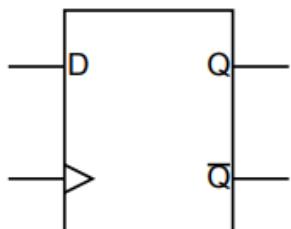
### Simboli dei vari D-Type-FF



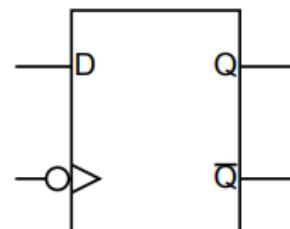
Level-Trigger, on CLK = 1



Level-Trigger, on CLK = 0



Edge-Trigger, on rising edge

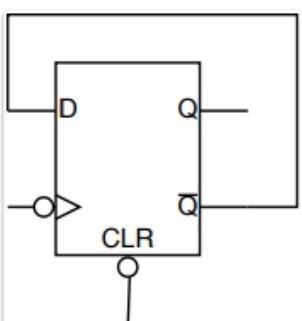


Edge-Trigger, on falling edge

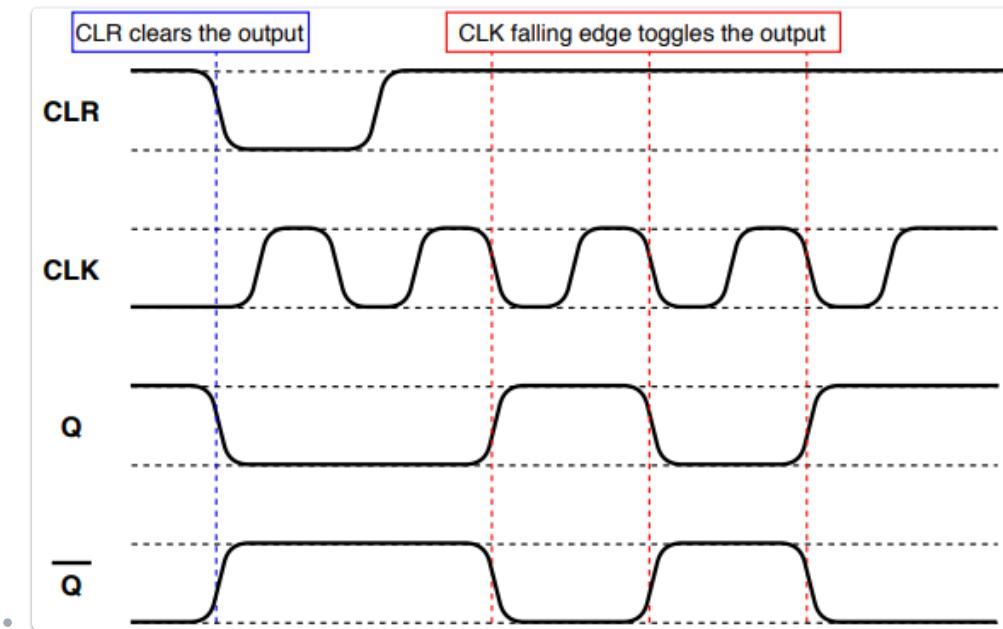
### T-Type-Flip-Flop

E' un D-Type-FF con la modifica del collegamento dell'ingresso  $D$  all'uscita  $\bar{Q}$  in cui ad ogni *colpo di clock* le uscite *cambiano di stato* ( $\bar{Q}$ ) viene copiata su  $Q$

Circuito:

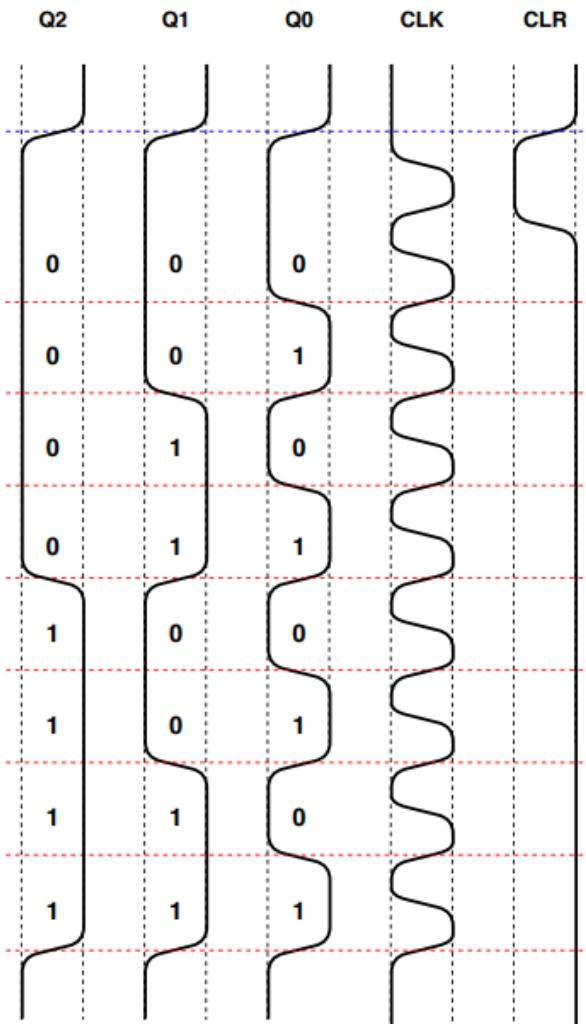
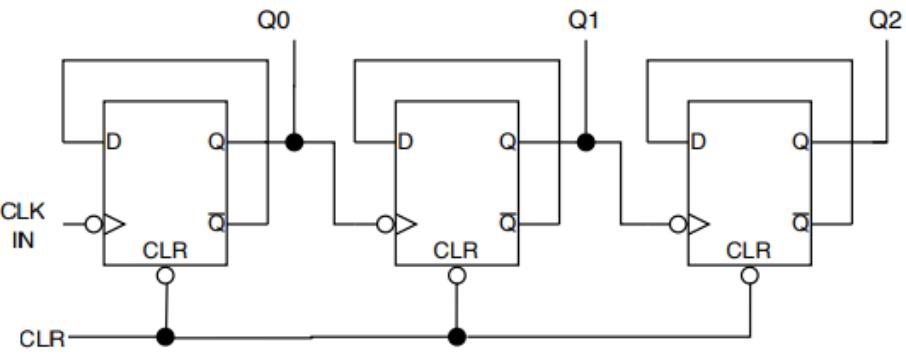


Comportamento:



### Counter Circuit

Attraverso dei T-Type-FF *collegati in serie* si possono costruire circuiti contatori ad  $n$  bit:



## 8. Three-State Logic

[6. Aspetti Elettronici delle Porte Logiche](#) in questa lezione, studiando i circuiti con Transistor MOS, si è potuto vedere come in essi lo *stadio di uscita* risulta sempre:

- verso il **positivo ( $V_+$ )**
- verso **ground**

Concettualmente esiste uno stato per la terza uscita che non è collegata né al positivo né al ground: l'uscita **floating**.

Ciò si può ottenere disabilitando sia il **PMOS** che il **NMOS** e aggiungendo un ulteriore MOS in serie allo stato di uscita pilotato da un segnale di **enable**:

Circuito:

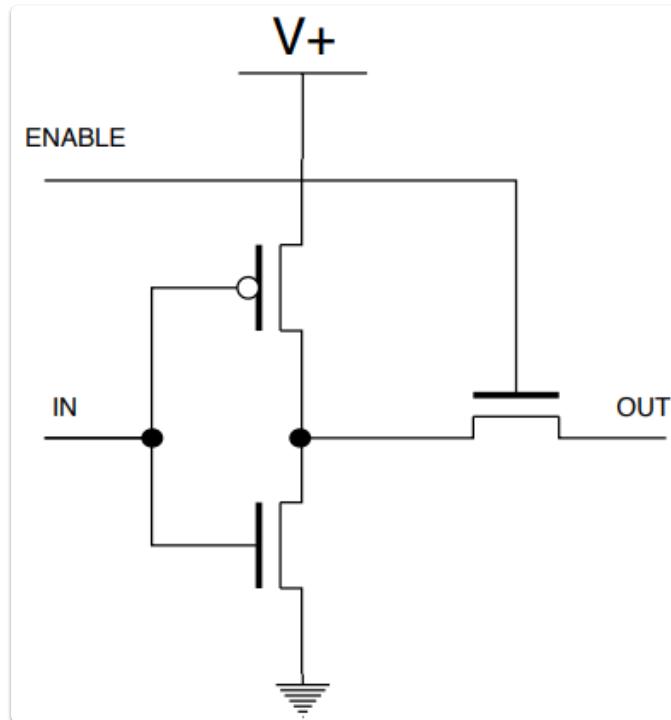


Tabella di verità:

IN	ENABLE	OUT
X	0	HiZ
1	1	0
0	1	1

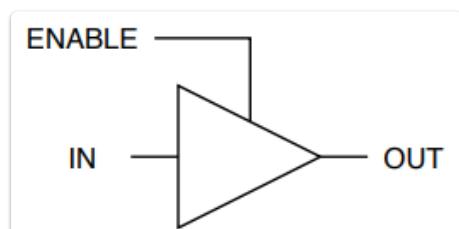
L'uscita può assumere tre stati elettrici:

1. Low, 0
2. High, 1
3. Floating o "High Impedance", HiZ

### Buffer "Three-State"

Realizzato tramite due Inverter MOS in serie, non applica alcuna trasformazione all'ingresso ma permette di introdurre un *enable* in un collegamento

Circuito:



### Utilizzi

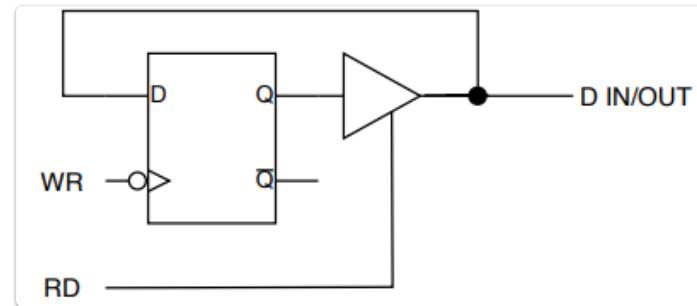
Diventa fondamentale nel momento in cui si desidera **interrompere un collegamento**. Le tipiche applicazioni:

- realizzazione di collegamenti **bidirezionali**
- connessione (tra loro) di **diverse uscite**, con selezione mutua

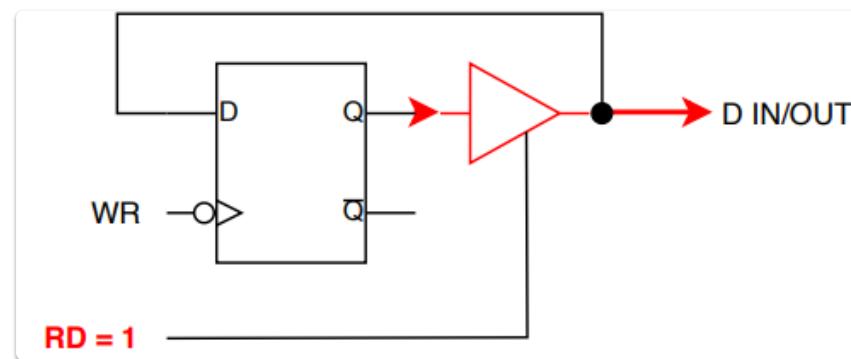
## Linee Bidirezionali

Il circuito in figura rappresenta una memoria a 1 bit in cui la linea dati è *bidirezionale* con un buffer in uscita che permette lo "switch" tra IN e OUT

Circuito:

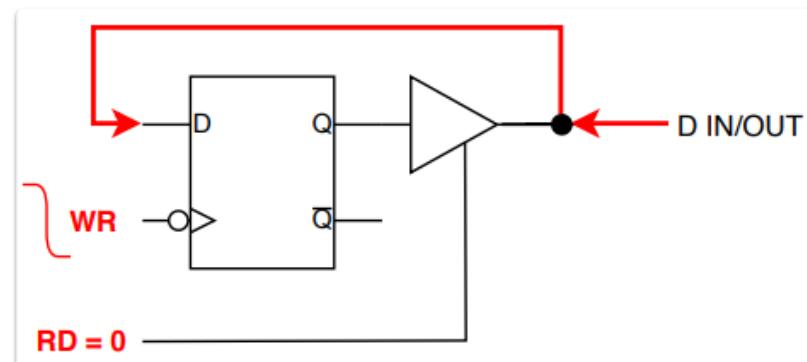


Lettura RD



1.  $RD = 1$ , il buffer permette il passaggio sulla linea *D IN/OUT* del dato memorizzato
2. linea *D IN/OUT* si comporta come **uscita**
3. *WR* è ininfluente

Scrittura WR

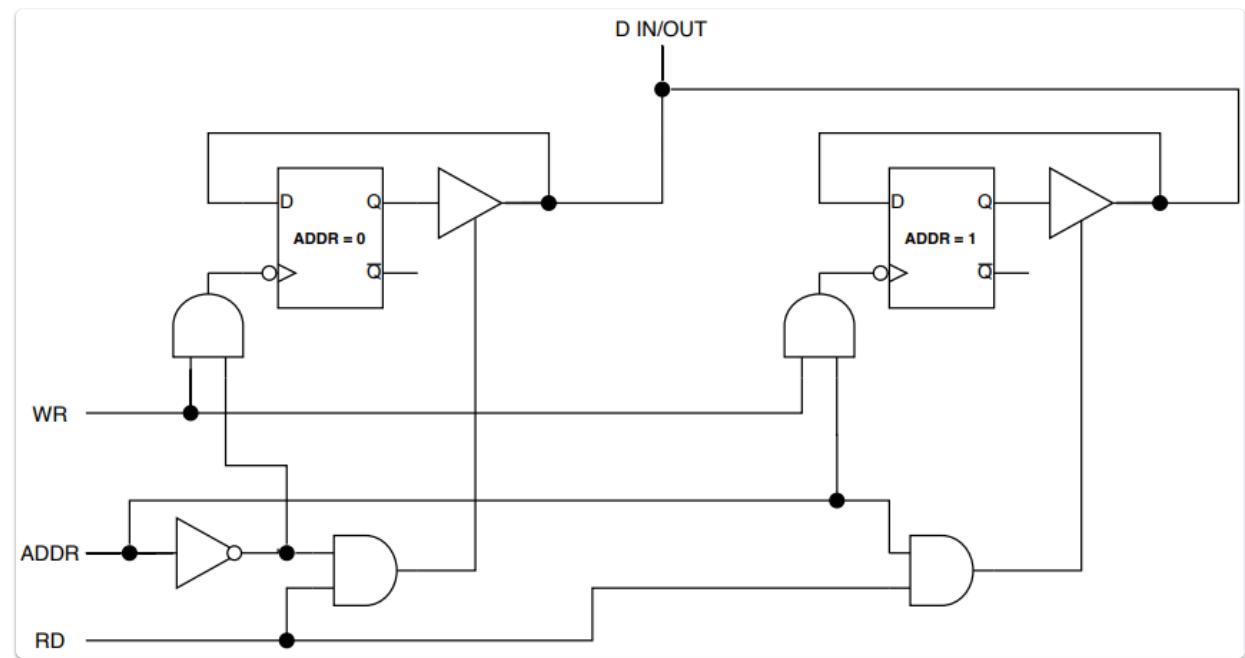


1.  $RD = 0$ , il buffer permette di isolare l'uscita del D-Type-FF
2. linea *D IN/OUT* si comporta come **ingresso** fornendo il dato da memorizzare
3. *WR* è influente sulla memorizzazione del dato presente in *D IN/OUT*

## Connessione di più uscite

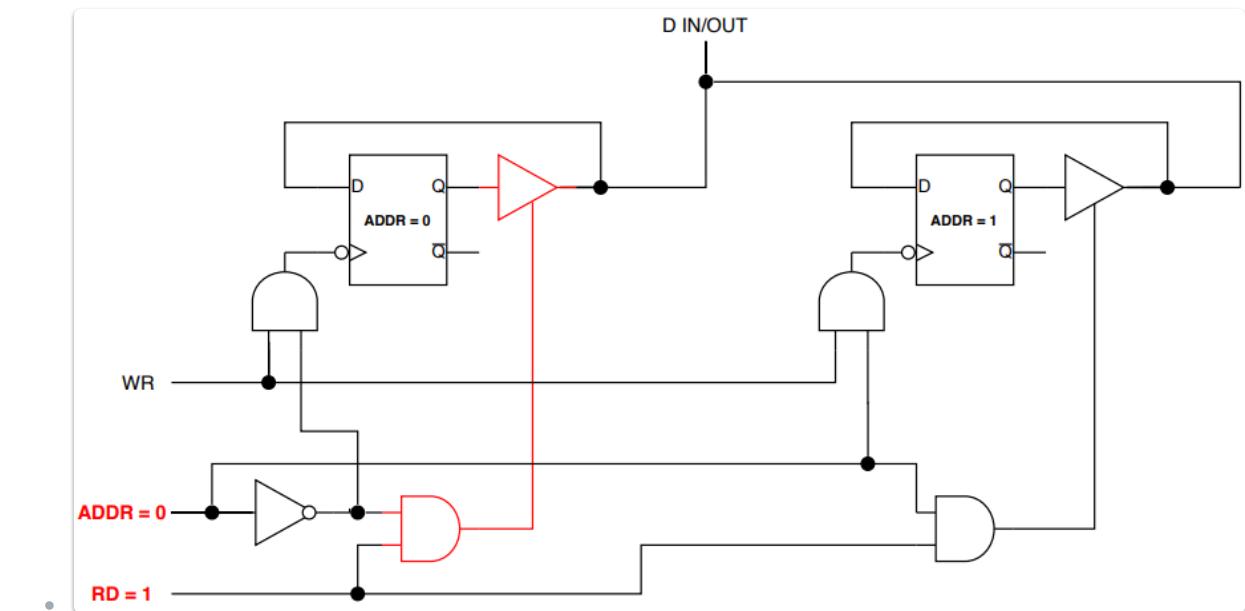
Qui le porte AND agiscono da **Gate** permettendo di attivare la *lettura* o la *scrittura* della locazione di memoria selezionata da **ADDR**

Circuito:

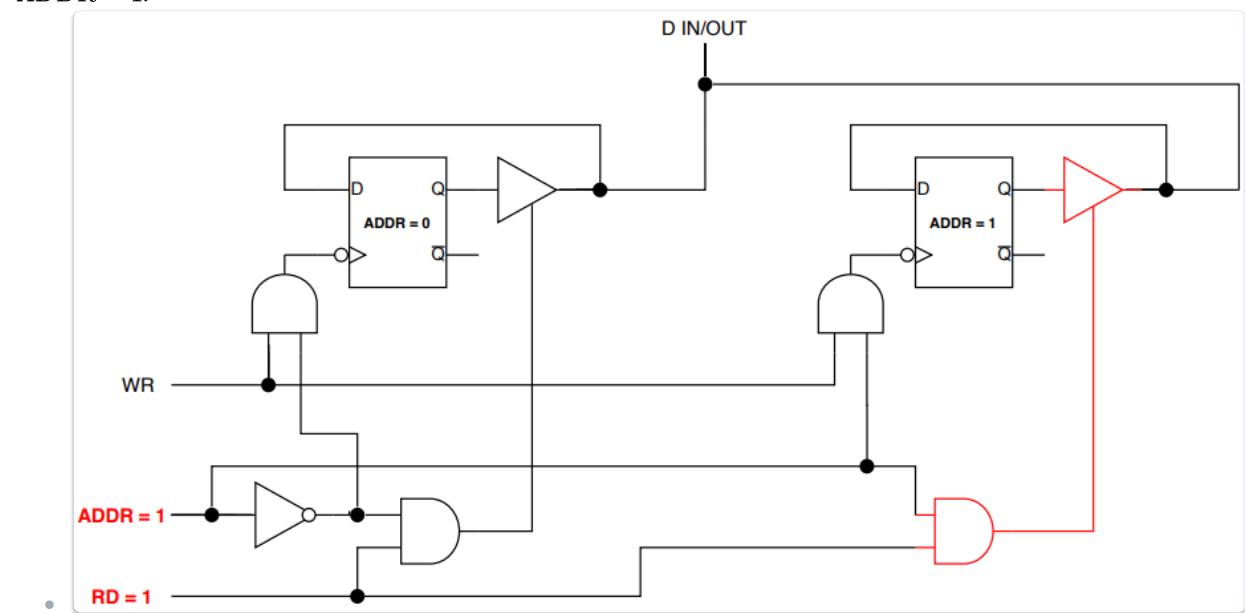


Lettura RD Locazione 0 e 1

- $ADDR = 0$ :

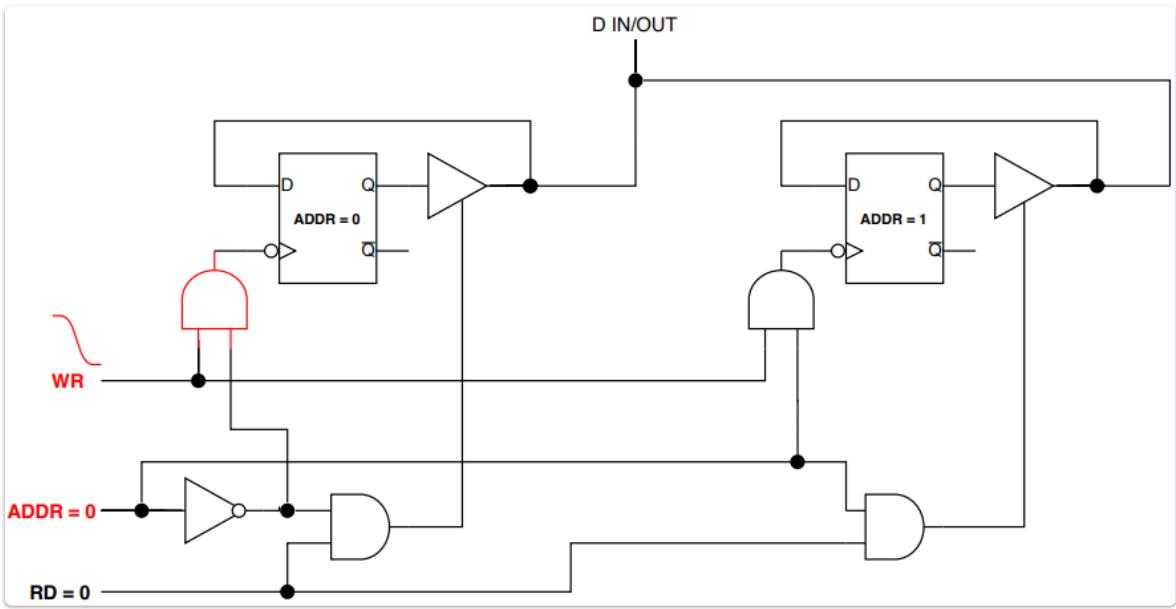


- $ADDR = 1$ :

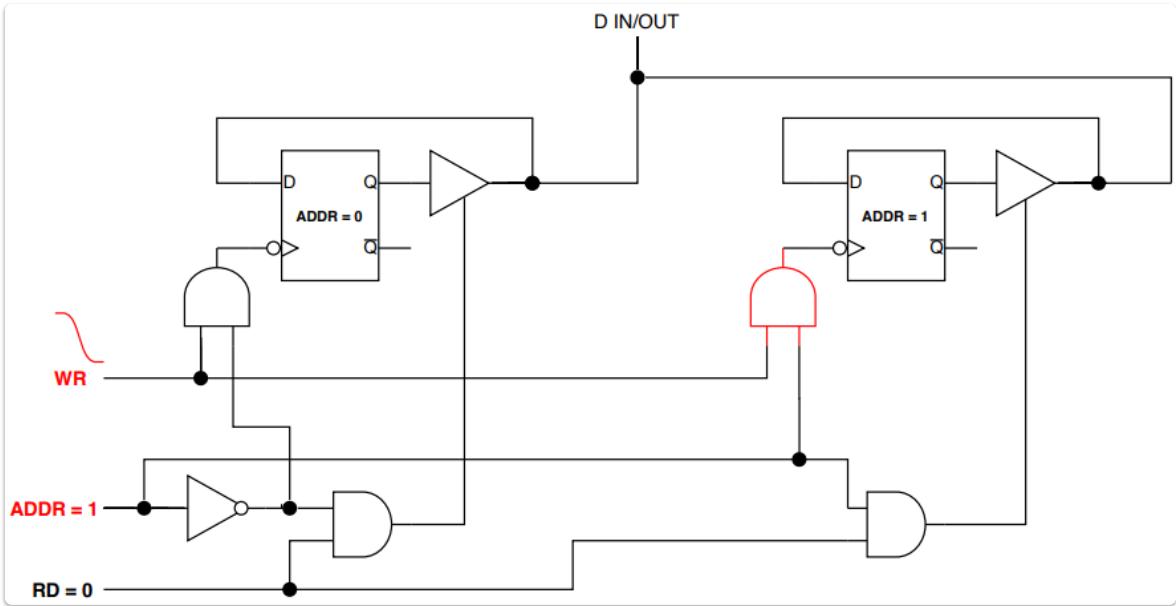


## Scrittura WR Locazione 0 e 1

- $ADDR = 0$ :



- $ADDR = 1$ :



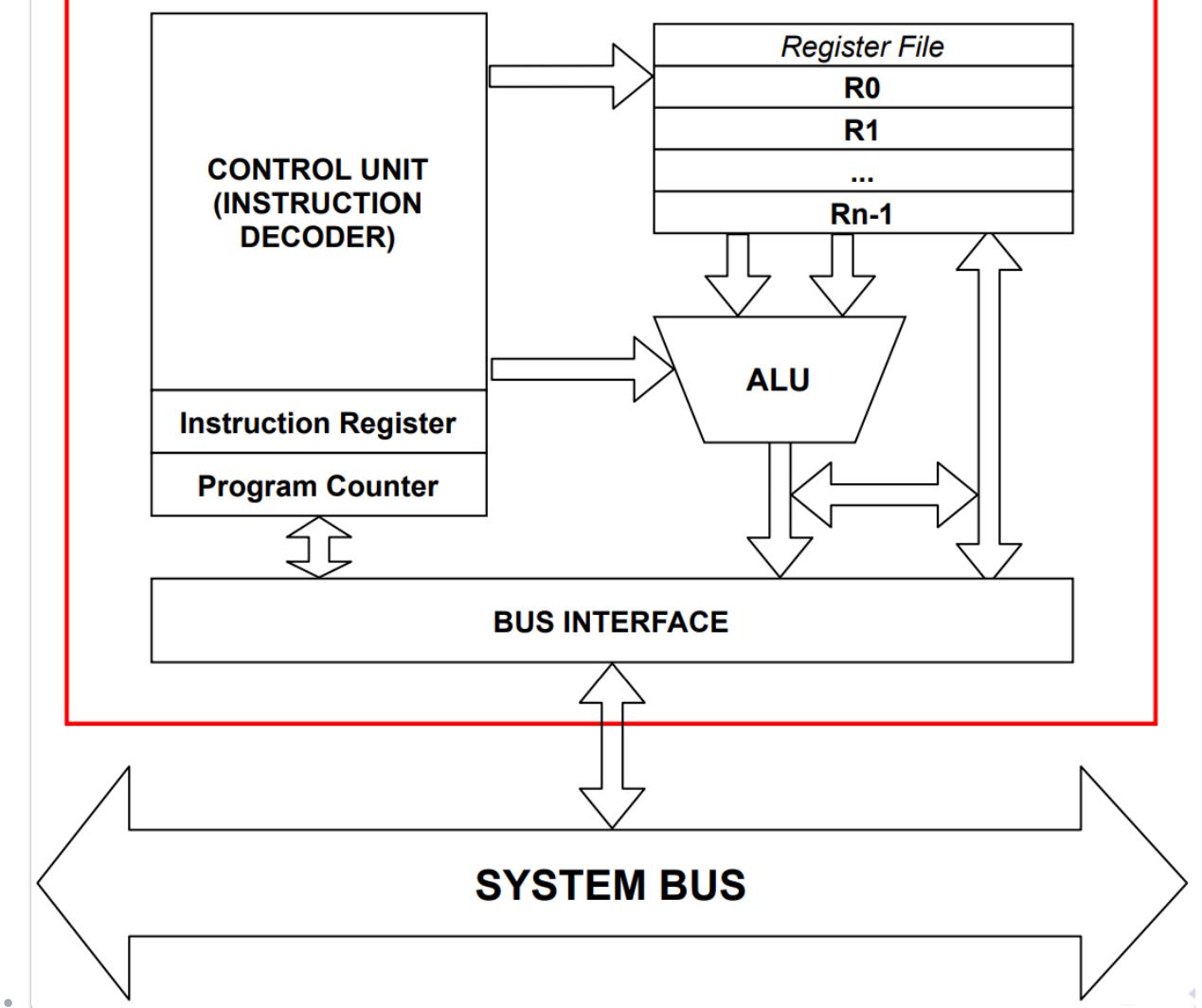
## 9. Architettura di una CPU

**Architettura** specifica di una CPU:

- [9.1 Inside the CPU](#)

**Architettura** generale di una CPU:

## CENTRAL PROCESSING UNIT



### Register File

E' un insieme di *registri a n bit* usati per dati temporanei.

Inoltre costituiscono gli *operandi* e il *risultato delle operazioni* aritmetiche/logiche.

La *dimensione* (*n* bit) di tali registri è un parametro importante di una CPU e ne indica la *capacità elaborativa*:

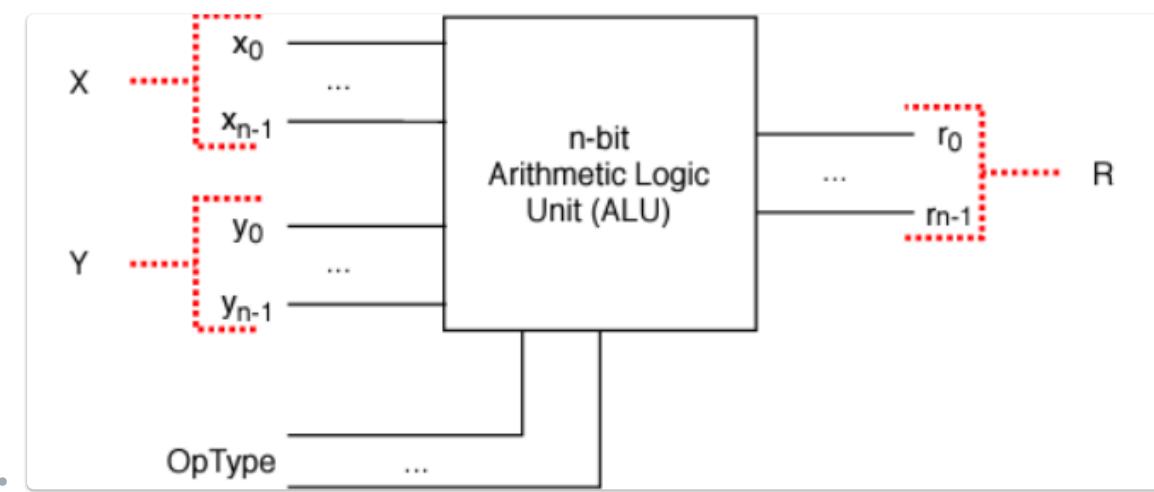
- 8
- 16
- 32
- 64

### Arithmetic Logic Unit (ALU)

Contiene i *circuiti* che permetto la *realizzazione e l'implementazione* delle operazioni aritmetiche/logiche.

Almeno uno degli operandi è un registro e il risultato dell'operazione viene generalmente memorizzato su un altro registro.

*Circuito:*



## Instruction Register (IR)

E' un registro che contiene l'*indirizzo dell'istruzione che la CPU sta eseguendo* il quell'istante di tempo  $t^*$ . Essa è un *insieme di bit*, ciascuno avente un significato specifico (*decodifica*):

- selezione dell'operazione dell'ALU
- selezione dei registri da usare
- ...

## Instruction Decoder (Control Unit)

E' il circuito che *preleva i bit dell'istruzione dall'IR* e "attiva" opportunamente il Register File e l'ALU

## Program Counter

E' un registro critico per il flusso di esecuzione di un programma poiché contiene l'indirizzo di memoria dell'*istruzione successiva* da eseguire.

## Bus Interface

E' il circuito che *permette l'interazione* (tramite il **bus di sistema**) tra i componenti della CPU ed i componenti esterni di un calcolatore (Memoria Centrale - Periferiche di I/O)

### 9.1 Inside the CPU

#### 9. Architettura di una CPU

La classificazione delle architetture di una CPU si divide in:

## RISC (processore ARM, Nintendo DS, iPhone)

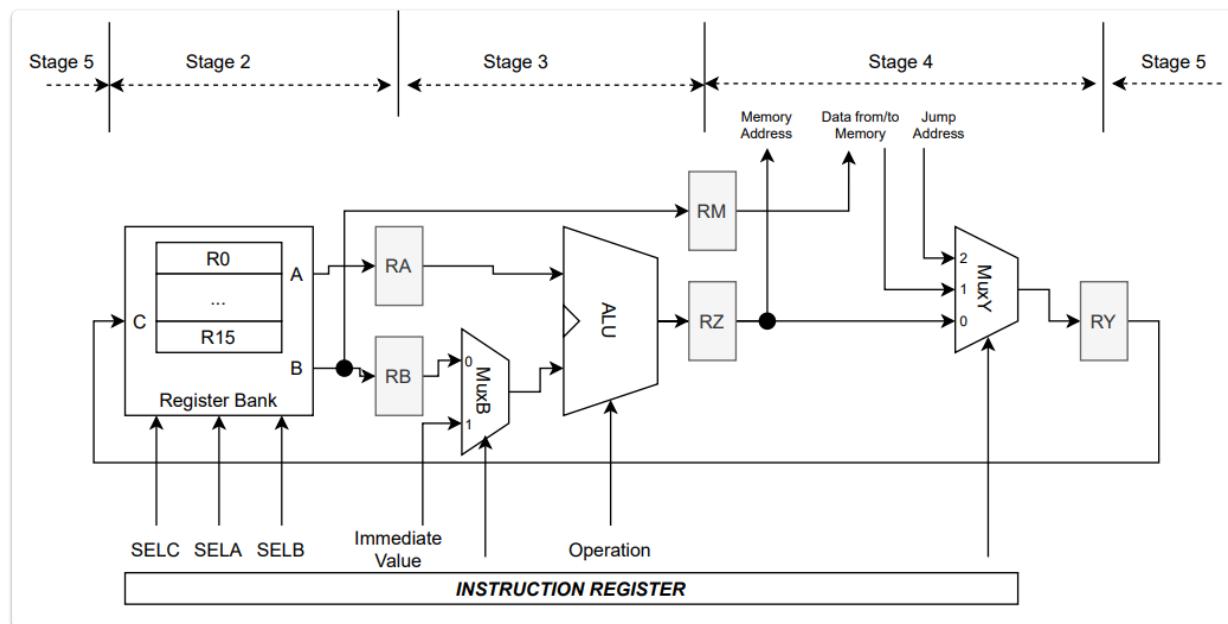
- **RISC** (*Reduced Instruction Set Computer*)
  - set di istruzioni *semplificato*
  - istruzioni aritmetiche/logiche *solo tra registri*
  - insieme *ridotto* di modalità di indirizzamento
  - ogni opcode (istruzione) *occupa una word*
  - esecuzione "a *stadi*" con possibilità di [10. Pipelining](#)

**Esempi** di istruzione RISC:

```
LDR Rx, [PC, #20]
ADD Rx, R7
LDR Rx, [Rx]
```

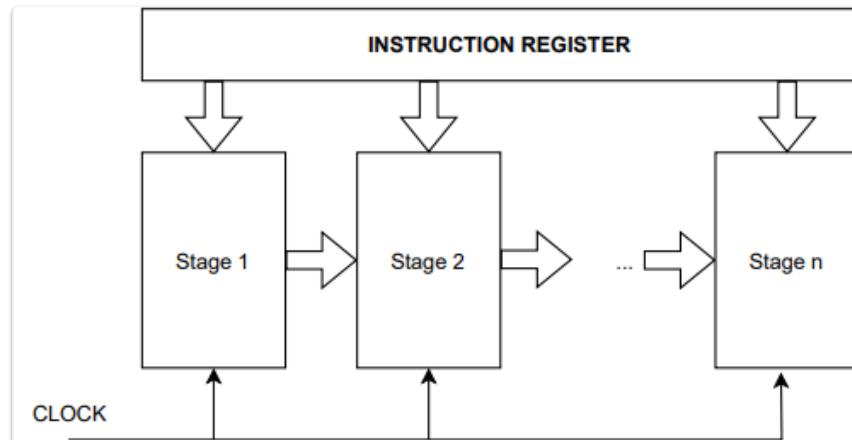
ADD R6, Rx  
B #n+24

## Sistema base di un processore RISC



## Stage-based Architecture

Architettura:

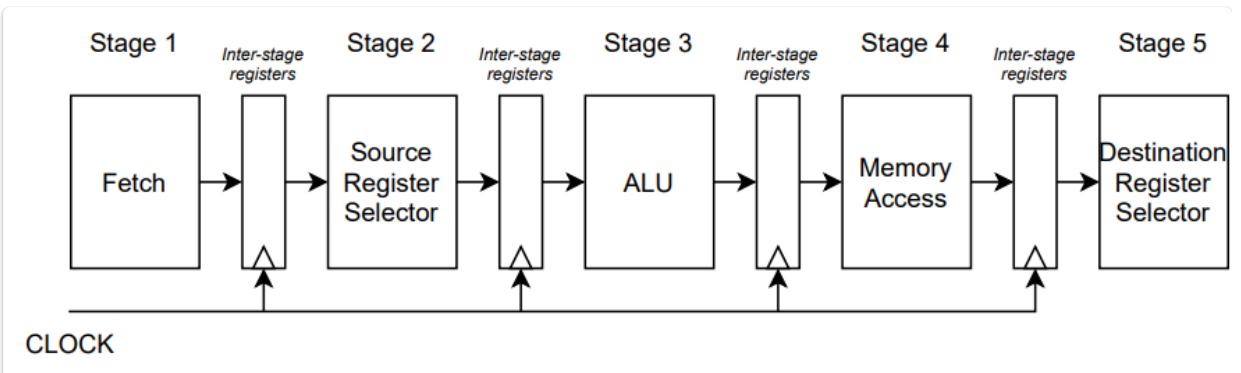


*Caratteristiche:*

- l'unità di elaborazione è composta da diversi **stadi (stage)** posti in sequenza
- ogni stage è **specializzato in un compito preciso** (operazioni aritmetiche/logiche, trasferimenti di memoria, ...)
- l'operazione specifica eseguita dal singolo stage è **selezionata da opportuni bit** dell'istruzione
- i dati trattati nell'istruzione "scorrono" attraverso gli stage
- l'insieme di stage viene denominato **Data Path**

## 5-Stage Architecture

Architettura:



- 1. *fetch* dell'istruzione
  2. *decode* o selezione dei registri "sorgente" per l'istruzione
  3. *compute* dell'eventuale operazione aritmetico/logica
  4. LOAD O STORE
  5. *write* sul registro di destinazione

**Inter-stage register** sono dei registri (interni) dove vengono *memorizzati i dati parziali* elaborati da ogni stadio

## CISC (Intel x86, AMD)

- CISC (Complex Instruction Set Computer)
  - set di istruzioni *ricco*
  - istruzioni aritmetiche/logiche anche *tra registri e memoria*
  - *svariate* modalità di indirizzamento
  - ogni opcode (istruzione) può occupare *più di una word*
  - programmi più "*compatti*" ma *meno performanti*

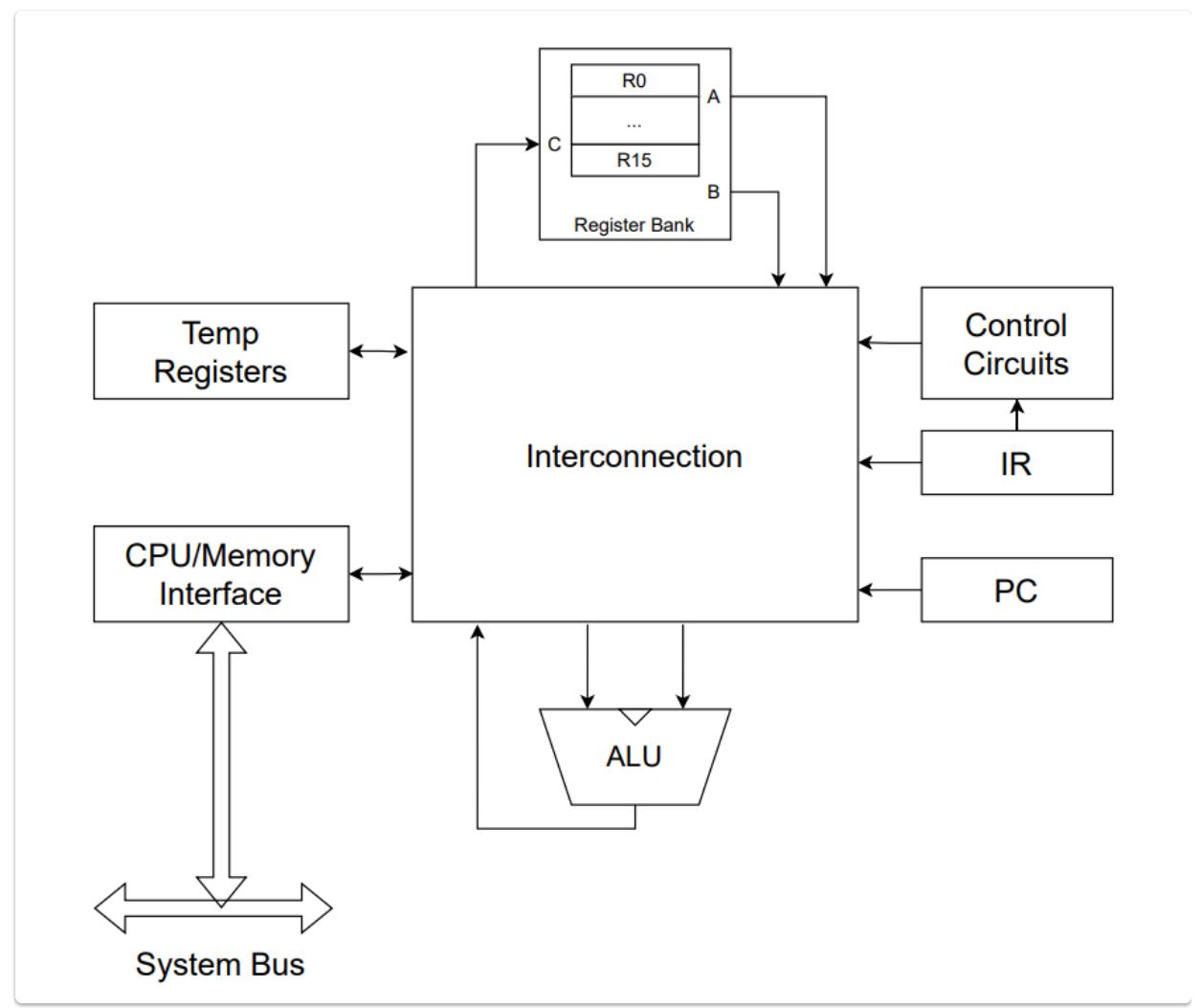
Esempi di istruzione CISC:

```

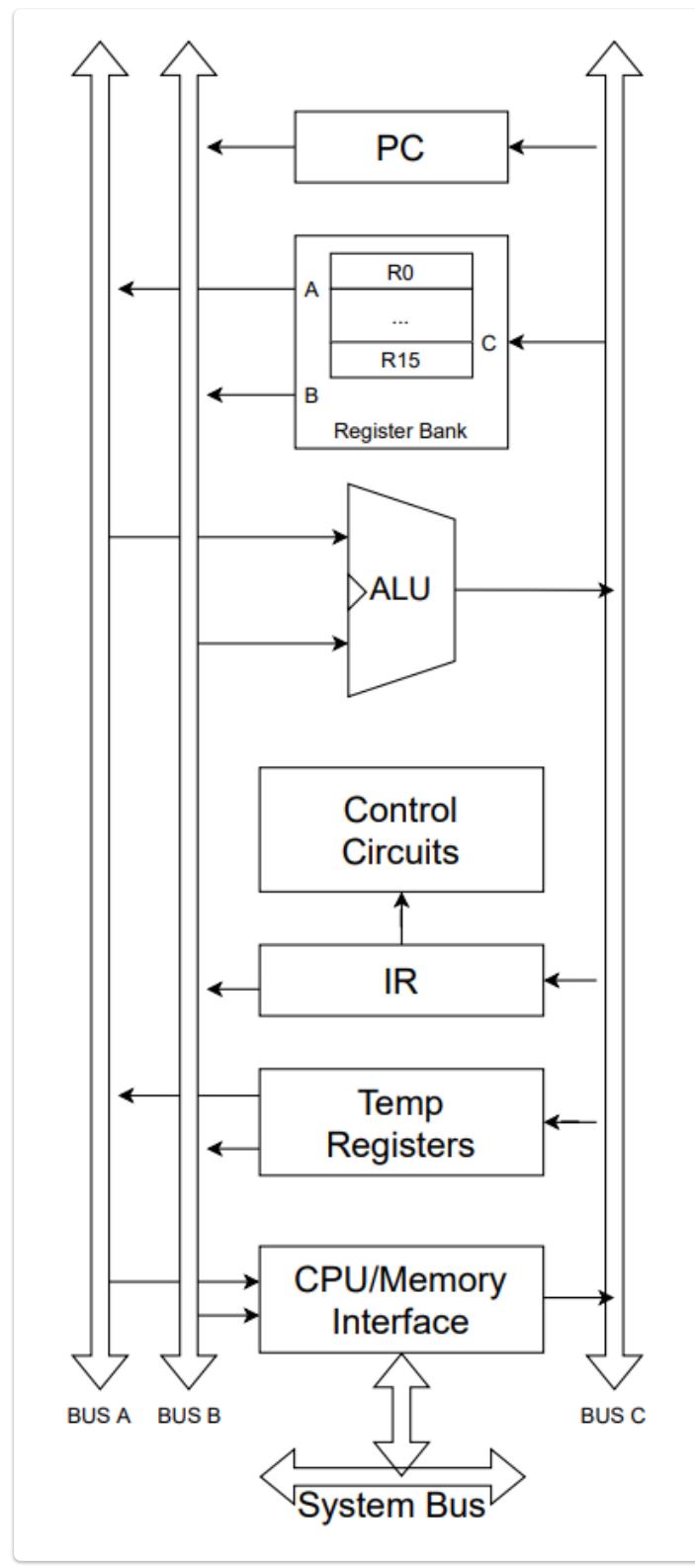
ADD r6, [r7]           ;1 word
ADD r6, [r7, #offset32] ;2 word

```

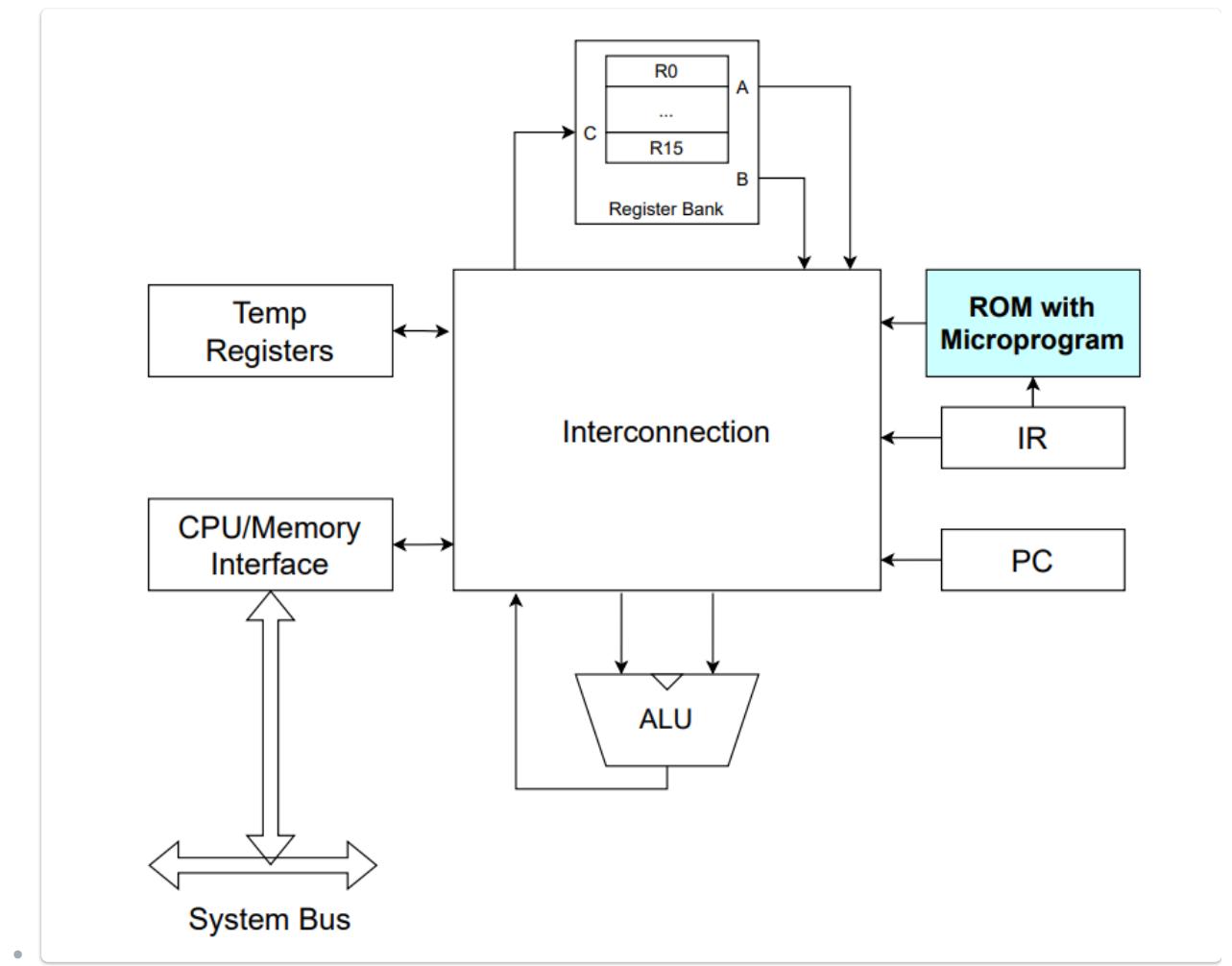
## Sistema base di un processore CISC



### 1. Interconnessione basata su Bus Interni



## 2. Interconnessione basata su Bus Microprogramma



## CPU e Microprogrammazione

Microprogramma per l'istruzione `ADD R6, [R7, #offset]`:

1.  $RM \leftarrow [PC]$ ,  $PC \leftarrow [PC] + 4$
2.  $IR \leftarrow [[RM]]$
3. decodifica dell'istruzione
4.  $RM \leftarrow [PC]$ ,  $PC \leftarrow [PC] + 4$
5.  $RTemp1 \leftarrow [[RM]]$
6.  $RTemp1 \leftarrow [RTemp1] + [R7]$
7.  $RM \leftarrow [RTemp1]$
8.  $RTemp1 \leftarrow [[RM]]$
9.  $RTemp1 \leftarrow [RTemp1] + [R6]$
10.  $R6 \leftarrow [RTemp1]$

## Fasi della CPU 1. Introduzione all'Architettura del Calcolatore

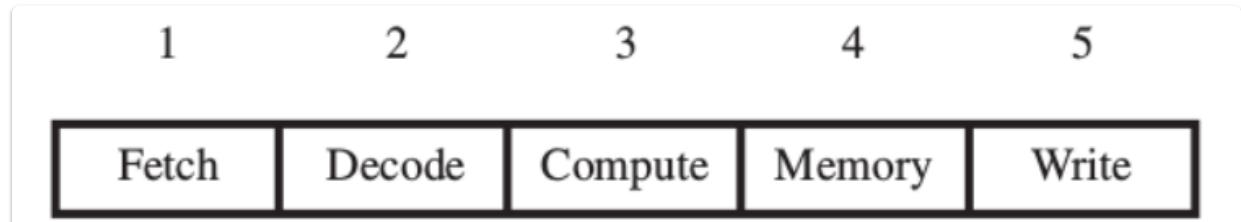
- **Fetch** dell'istruzione: trasferimento su IR della word contenuta nella locazione puntata dal PC =  $IR \leftarrow [[PC]]$
- Fase dell'**interpretazione**: incremento del PC in modo da puntare alla word successiva =  $PC \leftarrow [PC] + 4$
- **Execute** dell'istruzione: decodifica ed esecuzione dell'istruzione presente su IR

## 10. Pipelining

Il **Pipelining** è una tecnica che migliora l'efficienza nell'esecuzione delle istruzioni, aumentando il numero totale di istruzioni eseguite in un periodo di tempo determinato dal *ciclo di clock*

## Data Path e Stages

I 5 stage nell'esecuzione di un'istruzione possono essere rappresentati:



- Implementando la tecnica del Pipelining ciascuna istruzione richiede (almeno) un solo ciclo di clock.

*Funzionamento:*

- mentre lo stage  $k$  sta eseguendo l'istruzione  $I_j$ , lo stage precedente  $k - 1$  è libero
- lo stage  $k - 1$  può pertanto eseguire l'istruzione  $I_{j+1}$

## Data Dependency Hazard

E' sempre possibile il Pipelining? No.

Questo perché esistono casi in cui le istruzioni  $I_j$  e  $I_{j+1}$  hanno una **data dependency** tra loro.

Cosa fare:

- occorre **bloccare** (mettere in **stallo**) la pipeline
- $I_{j+1}$  deve attendere il completamento fino allo stage 5 dell'istruzione  $I_j$
- la condizione che provoca uno stallo della pipeline viene detta **hazard**

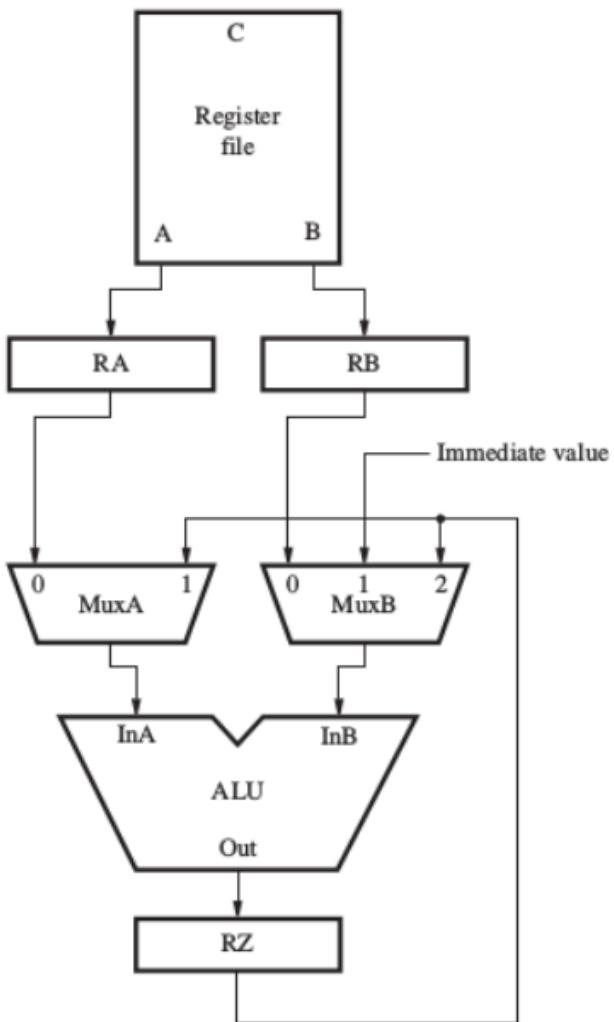
**Esempio:**

- ADD R2, R3, #100 e SUB R9, R2, #30 :



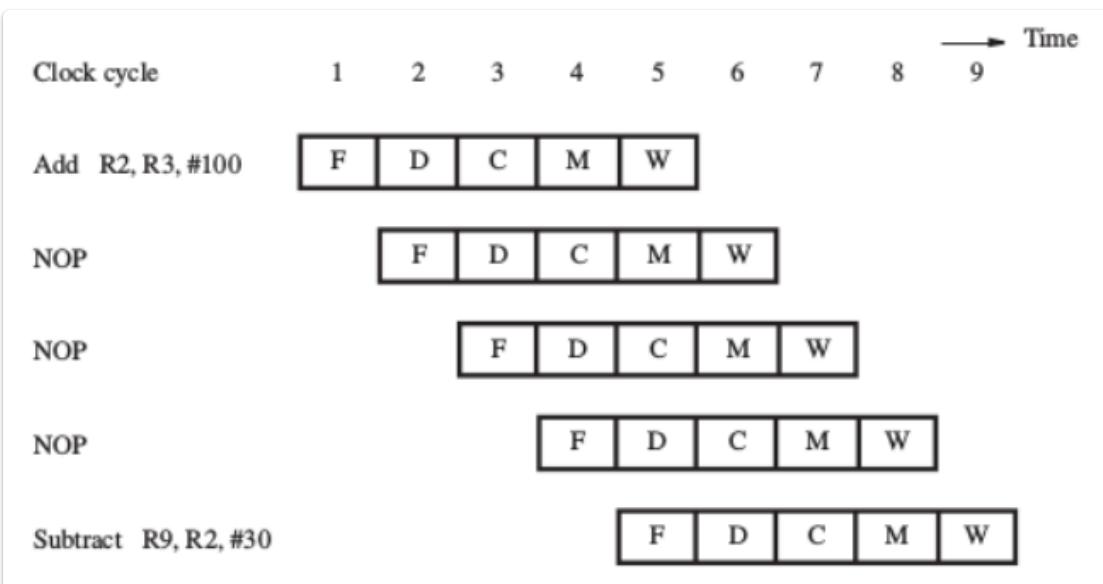
## Soluzione: Operand Forwarding

Il Data Path viene modificato in modo da far sì che l'*output dello stage 3* possa essere (anche) inviato all'input dell'ALU:



### Soluzione: NOP Insertion

Le NOP permettono alla `ADD` di completare il percorso nel Data Path:

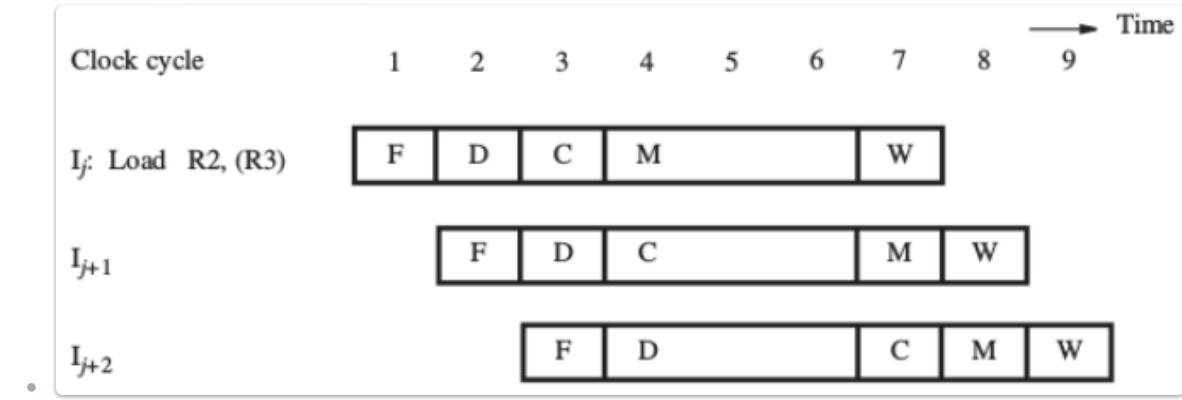


### Memory Delay Hazard

L'accesso alla memoria richiede (in genere) vari cicli di clock. Se la dipendenza include una `LOAD` o `STORE`, occorre *bloccare la pipeline* per tutta la durata dell'accesso alla memoria

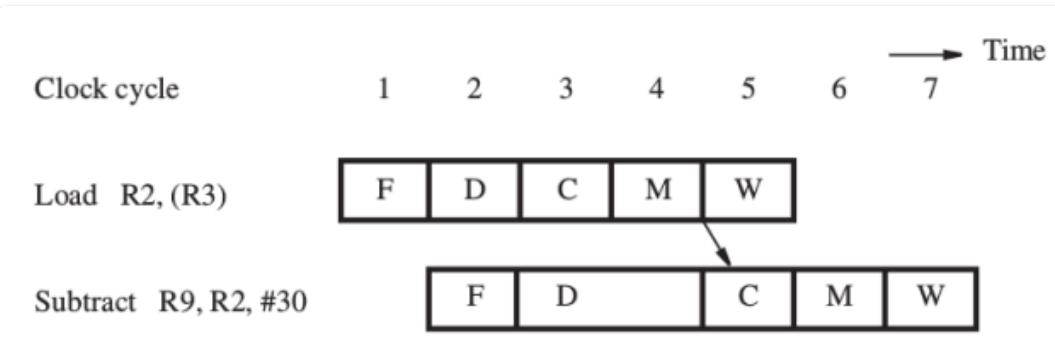
**Esempio:**

- LDR R2, [R3] e SUB R9, R2 #30:



### Soluzione: Operand Forwarding

L'uscita dello stage 4 viene inviata come input aggiuntivo dello stage 3 della prossima istruzione:

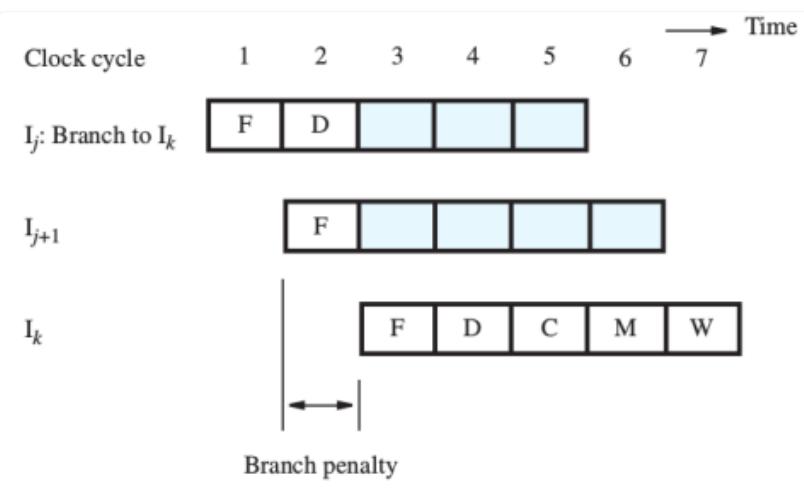


### Branch Hazard

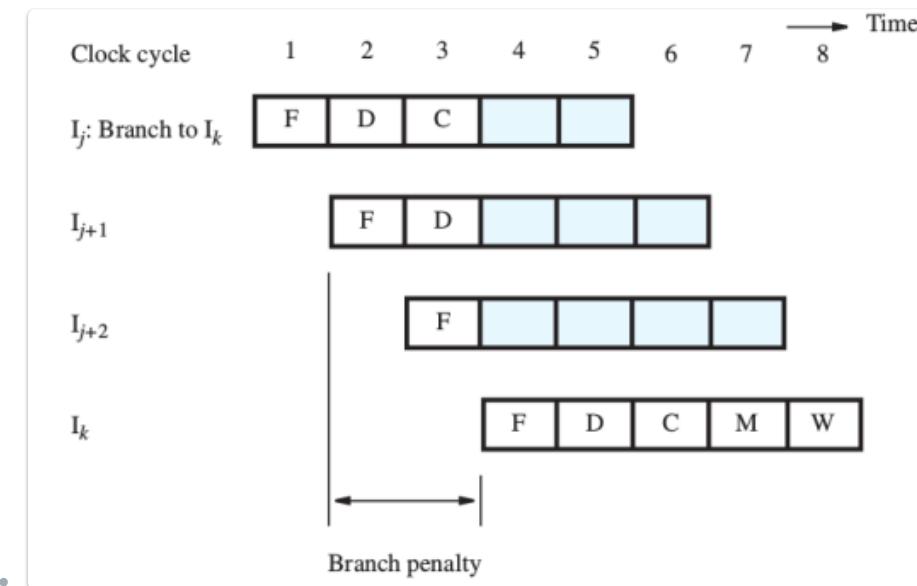
L'istruzione di **branch** implica una "penalità" di efficienza.

#### Branch Incondizionati

Il branch con destinazione **assoluta** comporta un ritardo di esecuzione già a partire dalla *fase di fetch*:



Il branch con destinazione **relativa** comporta un ritardo di esecuzione a partire dalla *fase di compute*:



## Branch Condizionati

Il branch condizionato implica la *valutazione di una condizione*.

In funzione della condizione occorre eseguire il fetch o dell'**istruzione successiva** o dell'**istruzione all'indirizzo target**: prevale  $I_{j+1}$  o  $I_k$ ?

### Static Branch Prediction

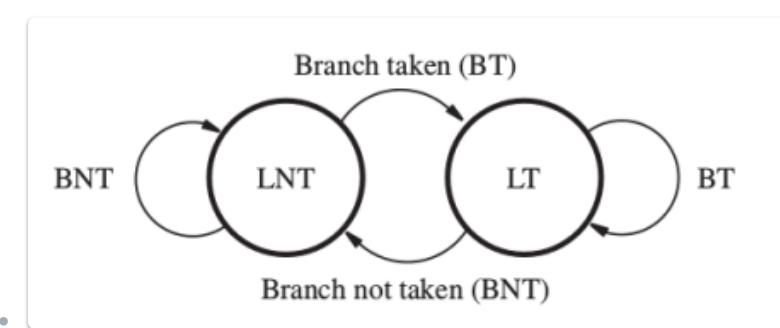
Si decide **a priori** (design-time) se effettuare il fetch di  $I_{j+1}$  o  $I_t$  e ogni volta che la **predizione è errata** si paga la **branch penalty**

### Dynamic Branch Prediction

Si utilizza una macchina a stati finiti con **due** stati:

- **LT** = Branch *Likely to be Taken* ( $I_t$ )
  - **LNT** = Branch *Likely Not to be Taken* ( $I_{j+1}$ )
- Si decide lo stato iniziale e ogni volta che la **predizione è errata** si paga la **branch penalty**

Grafo:

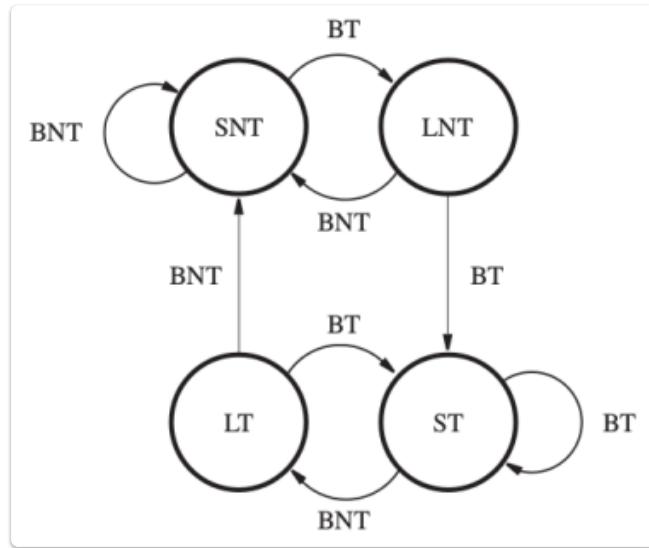


### More Accurate Dynamic Branch Prediction

Si utilizza una macchina a stati finiti con **quattro** stati:

- **ST** = Strongly likely to be Taken (salto molto probabilmente effettuato  $I_t$ )
- **LT** = Branch *Likely to be Taken* (salto probabilmente effettuato  $I_t$ )
- **LNT** = Branch *Likely Not to be Taken* ( $I_{j+1}$ )
- **SNT** = Strongly likely Not to be Taken (salto molto probabilmente non effettuato  $I_{j+1}$ )

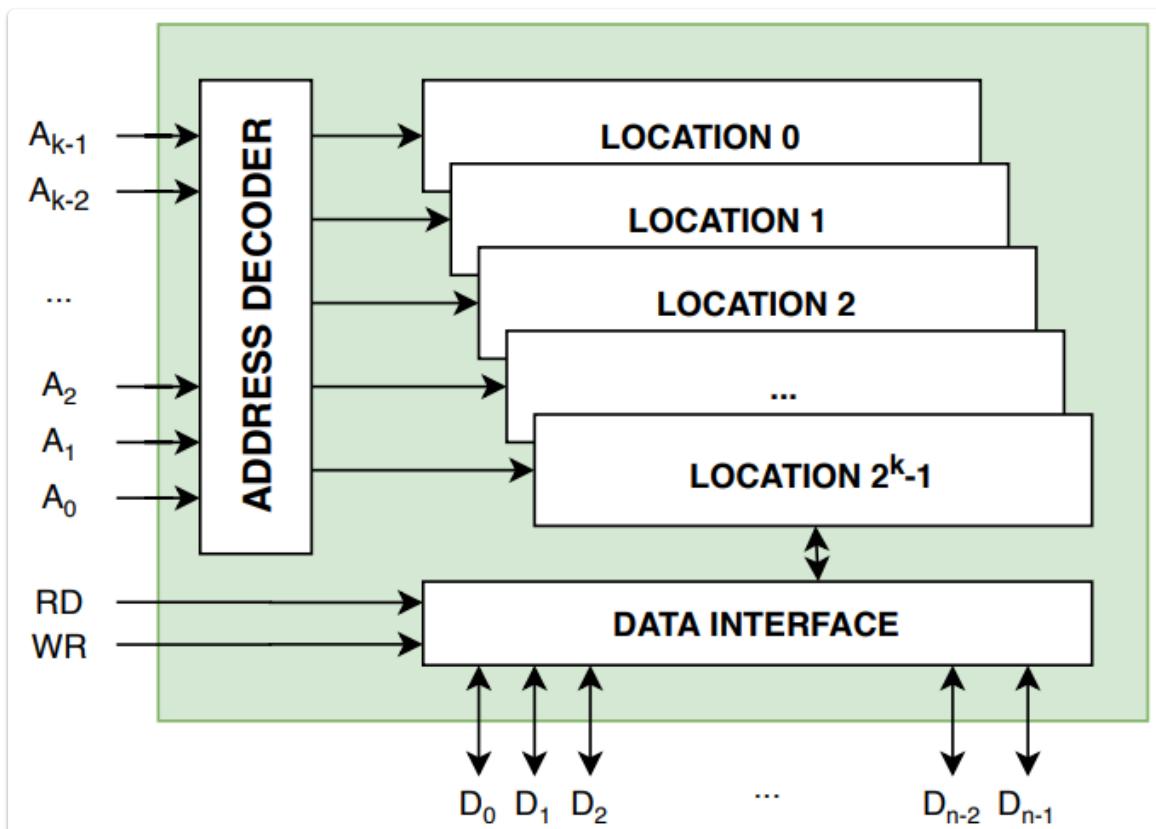
Grafo:



## 11. Architettura della Memoria

- [11.1 La Memoria](#)

Architettura:



Un dispositivo di memoria, concettualmente, è costituito da un *insieme di registri ad n bit*, ognuno dei quali rappresenta una **locazione** (o **cella**) di memoria:

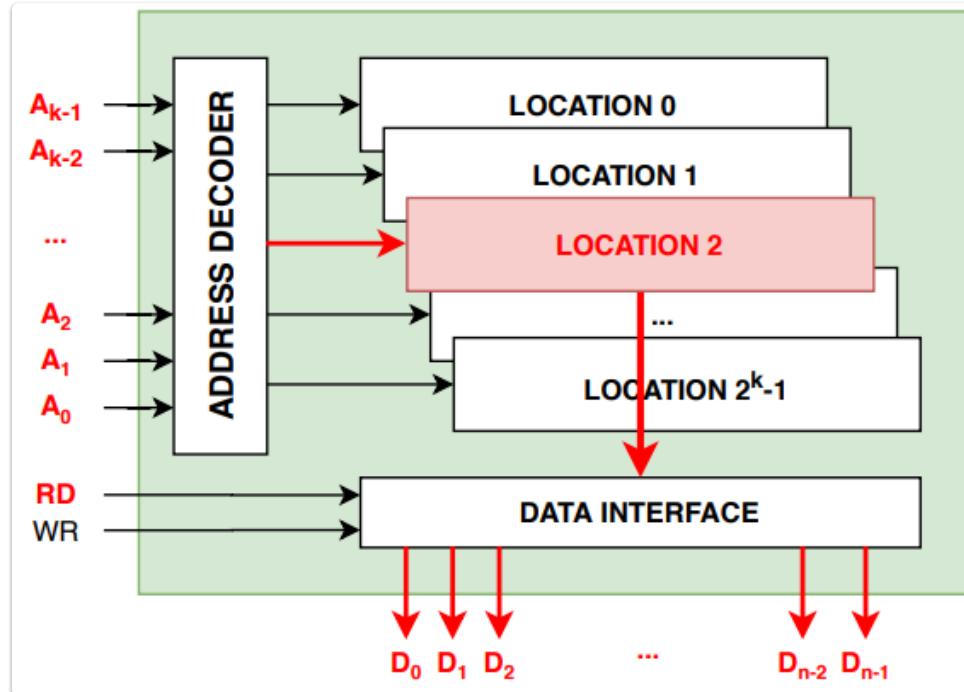
- la selezione di quest'ultima avviene attraverso un insieme di  $k$  linee le cui combinazioni binarie permettono di *indirizzare  $2^k$  celle* ( $[0, 2^{k-1}]$ )

## Composizione

- $A_0, \dots, A_{k-1}$  sono le **address lines** (indirizzi) ciascuna delle quali indica la cella alla quale si vuole accedere (*sequenzialmente* e una cella per volta)
- $RD$  e  $WR$  sono le **control lines** (lettura e scrittura)
- $D_0, \dots, D_{n-1}$  sono le **data lines** e sono *bidirezionali* (input e output)

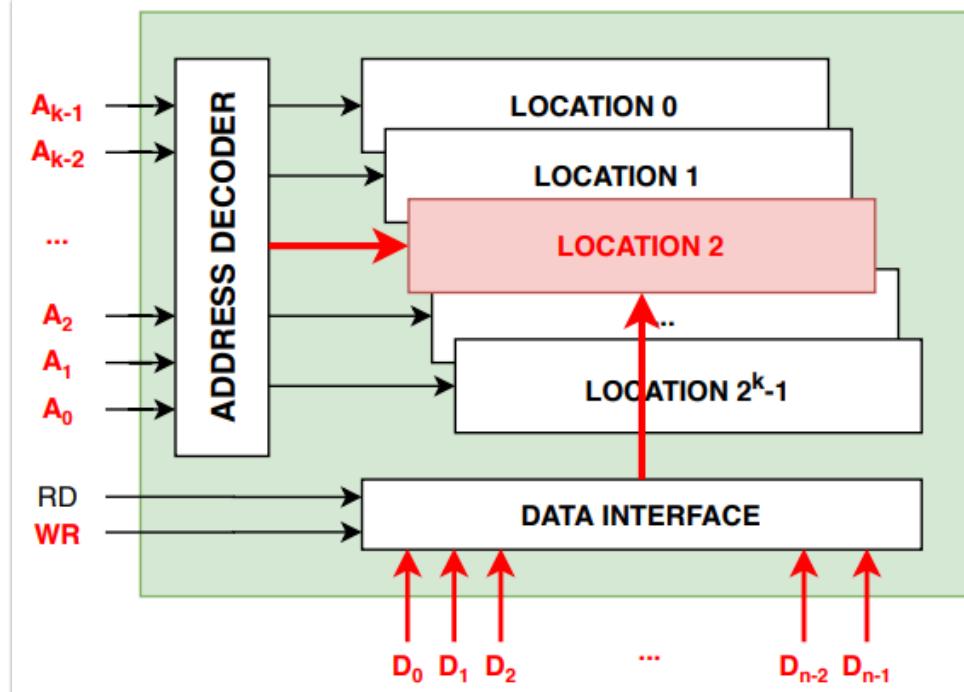
## Funzionamento (RD e WR)

- RD:



- $A_i$  specificano la cella di memoria da leggere
- **Address Decoder** "seleziona" il circuito relativo alla cella di memoria
- **RD** viene attivato
- **Data Interface** "emette" sulle linee dati il contenuto della cella indirizzata

- WR



- $A_i$  specificano la cella di memoria da leggere
- **Address Decoder** "seleziona" il circuito relativo alla cella di memoria
- **WR** viene attivato
- **Data Interface** "copia" sulle linee dati il contenuto della cella indirizzata

## Organizzazione

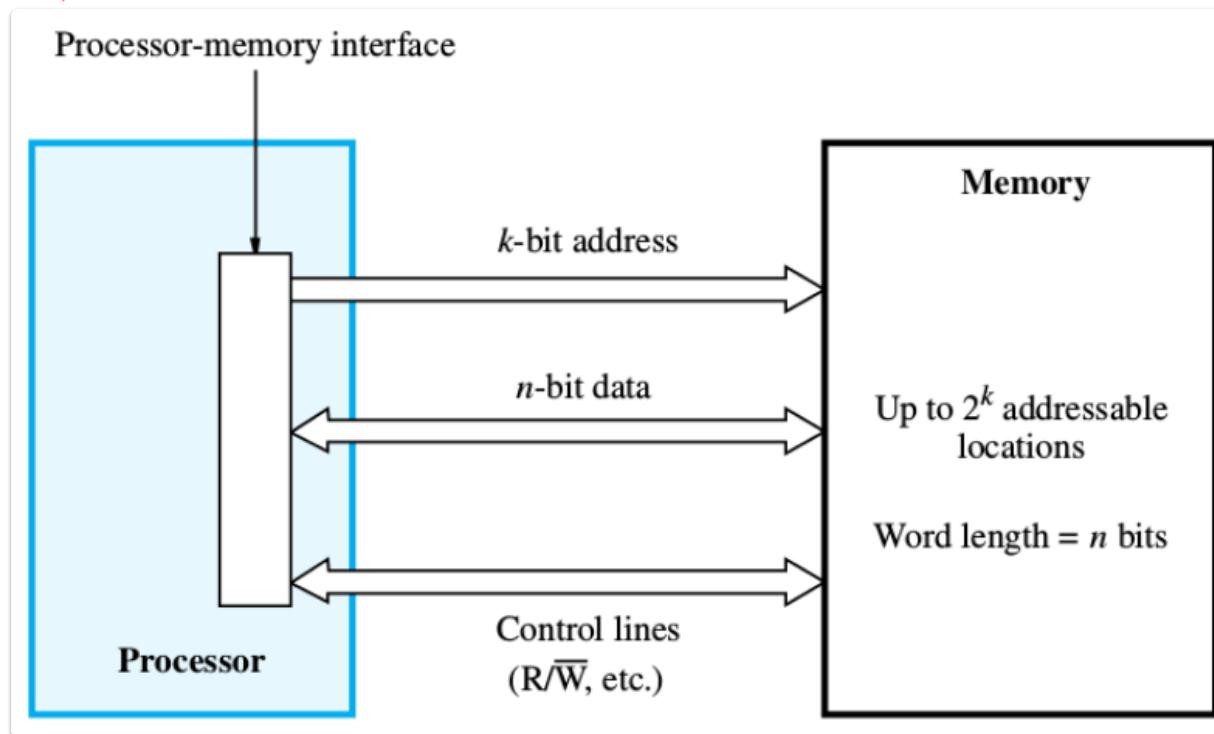
La memoria è *organizzata in BYTE* dunque ogni cella di memoria consente di memorizzare fino ad un massimo di 8 bit.

Se l'address bus ha  $k$  linee di indirizzi, la memoria avrà  $2^k$  byte.

## 11.1 La Memoria

- [11. Architettura della Memoria](#)

Interfaccia CPU → Memoria:



### Control Lines

- $\overline{RD} = 0$  indica un'operazione di **read**
- $\overline{WR} = 0$  indica un'operazione di **write**

Tabella di verità:

$\overline{RD}$	$\overline{WR}$	Azione
1	1	Nessuna azione
0	1	Read
1	0	Write
0	0	<b>non valido</b>

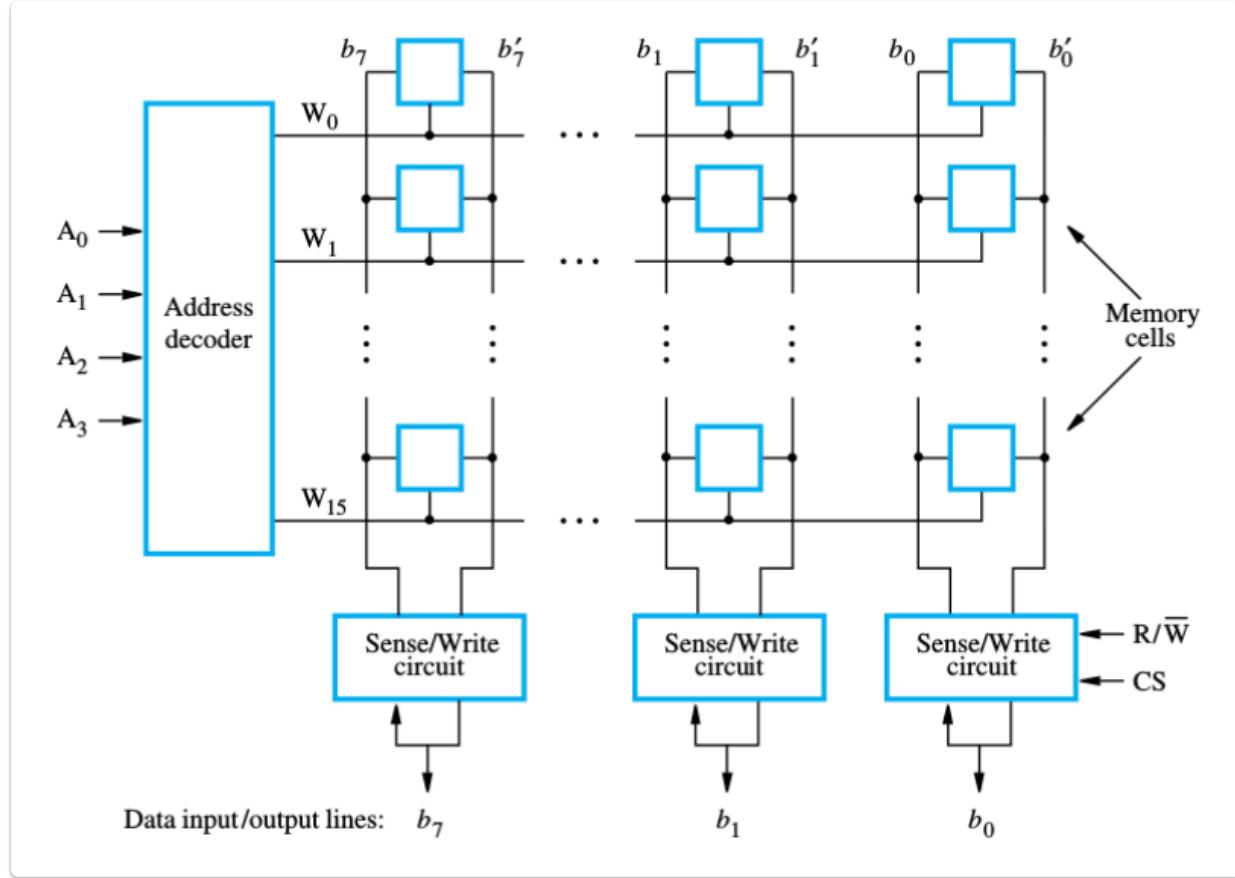
- $\overline{CS}$ , **Chip Select**, = 0 indica che il dispositivo è selezionato per un'operazione
  - è un segnale, generato dal microprocessore, che si usa per selezionare o abilitare uno specifico chip di memoria
- $R/\overline{W} = 1/0$  indica un'operazione di **read** e un'operazione di **write**

Tabella di verità:

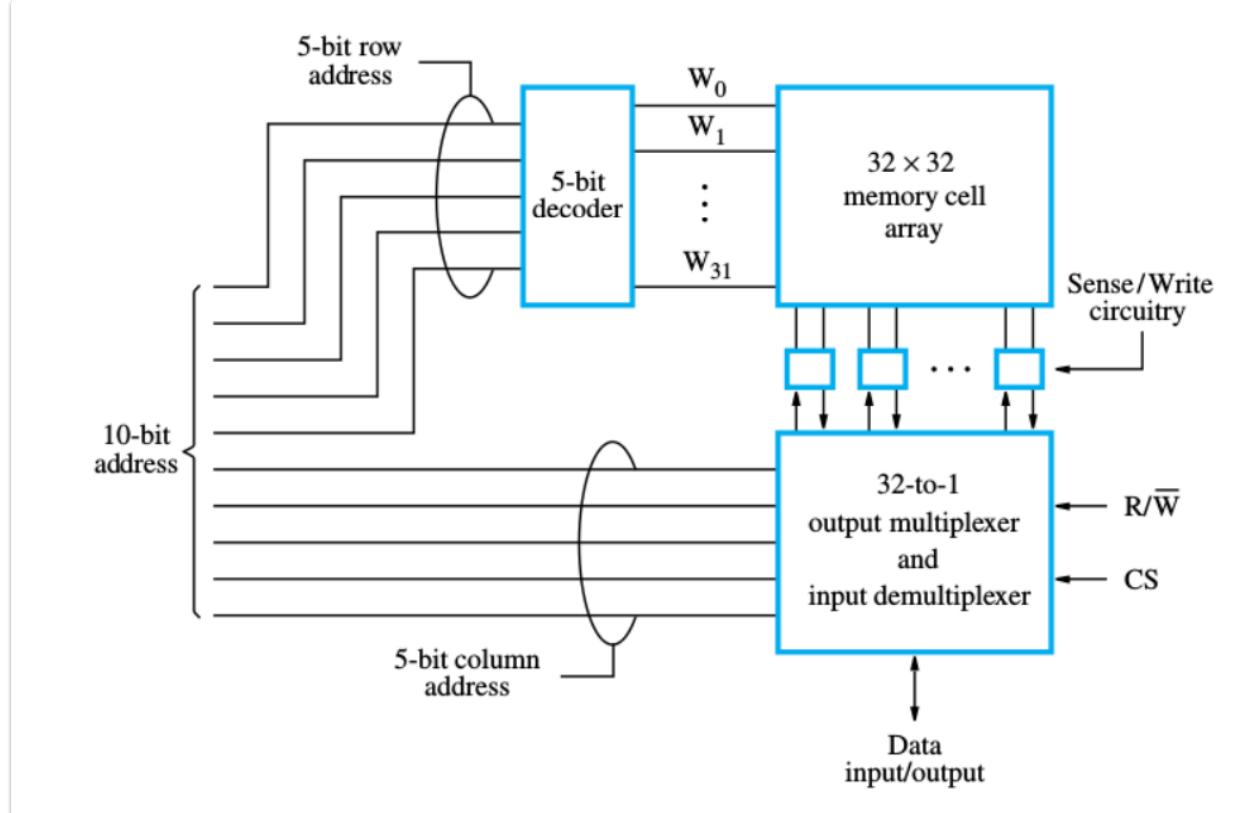
$\overline{CS}$	$R/\overline{W}$	Azione
1	X	Nessuna azione
0	1	Read
0	0	Write

### Struttura base di un chip di memoria

*Architettura* di base:



*Architettura* RAM a Matrice:



## Tipologie di Memorie

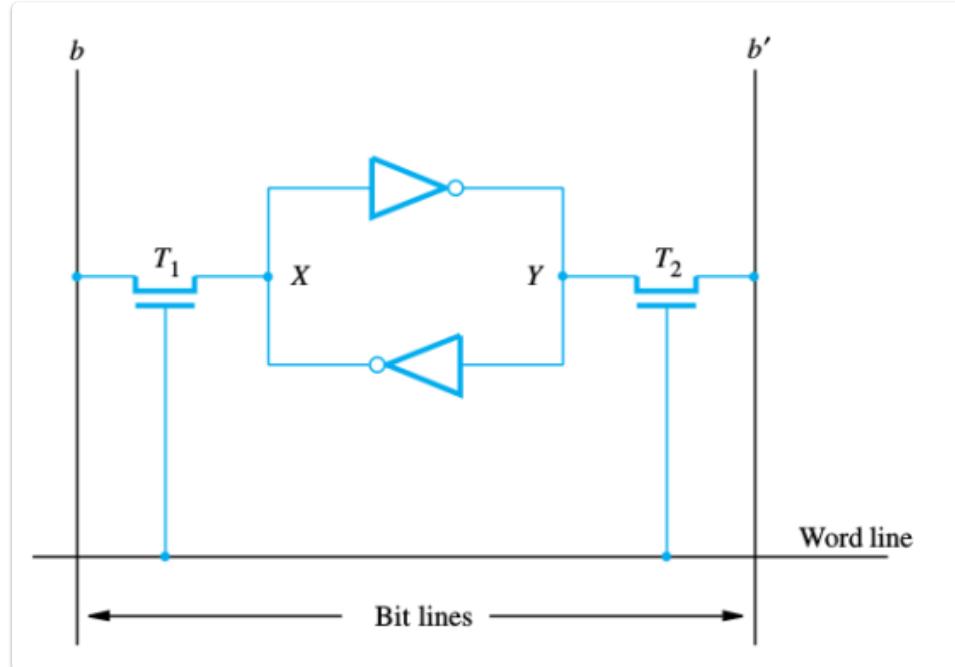
Cos'è una memoria RAM? [1. Introduzione all'Architettura del Calcolatore](#)

Esistono due tipologie di memoria RAM:

### 1. SRAM

- **SRAM** (Static RAM)
  - la cella è realizzata tramite un *flip-flop* o componente analogico
  - capacità (su silicio) *limitata*
  - consumo *elevato*

*Circuito:*

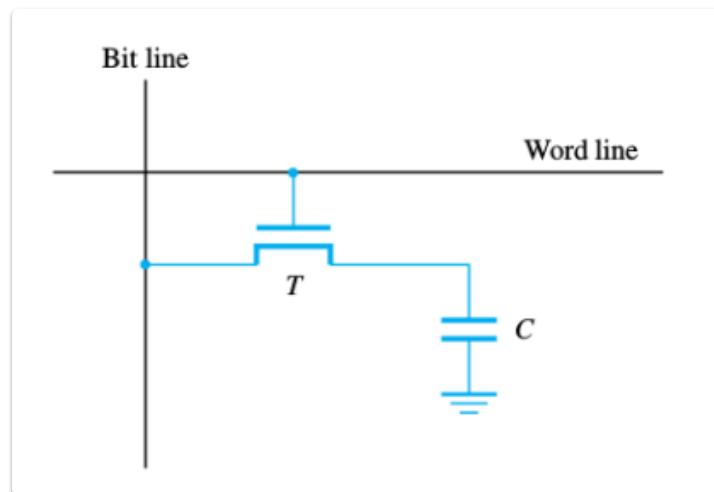


- **WRITE**:  $b$  è un *input*; attivando la Word Line il dato rimane intrappolato nel loop dei due inverter
- **READ**:  $b'$  è un *output*; attivando la Word Line il loop emette il suo valore

## 2. DRAM

- **DRAM** (Dynamic RAM)
  - la cella è realizzata tramite un *condensatore*
  - capacità (su silicio) *maggiori*
  - consumo *molto basso*

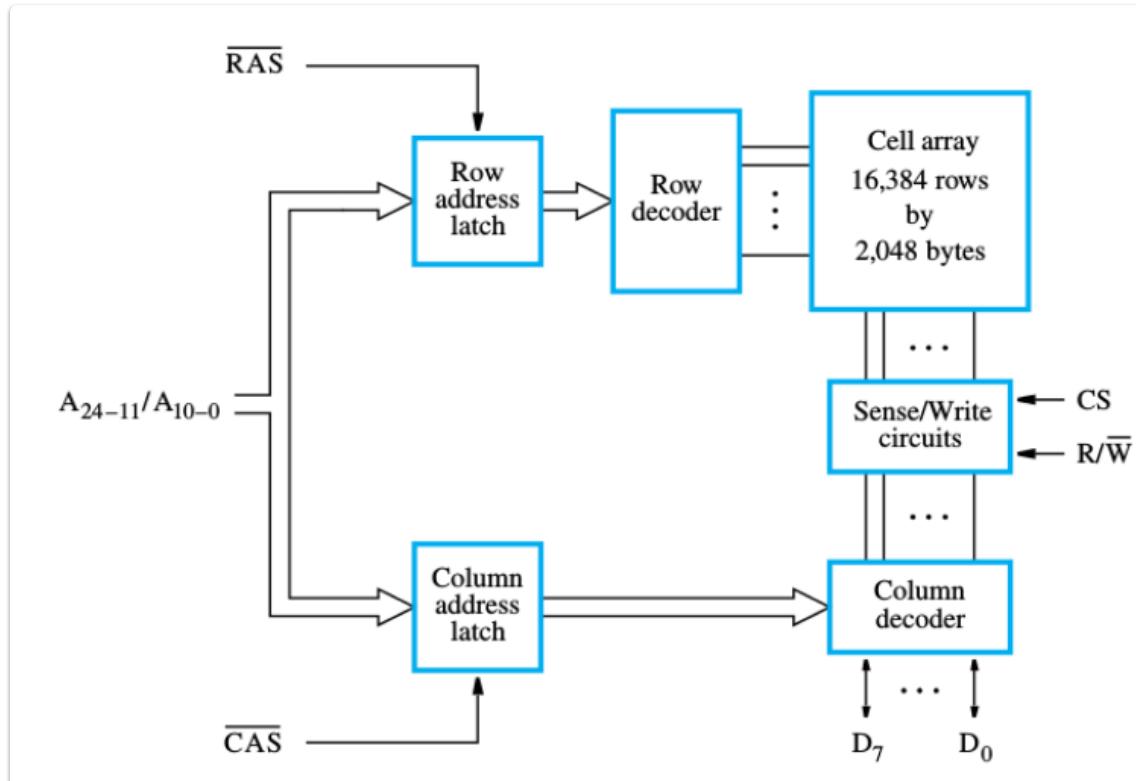
*Circuito:*



- **WRITE**: la Bit Line è un *input*; attivando la Word Line il condensatore si carica se c'è 1
- **READ**: la Bit Line è un *output*; attivando la Word Line il condensatore emette la sua carica sulla Bit Line

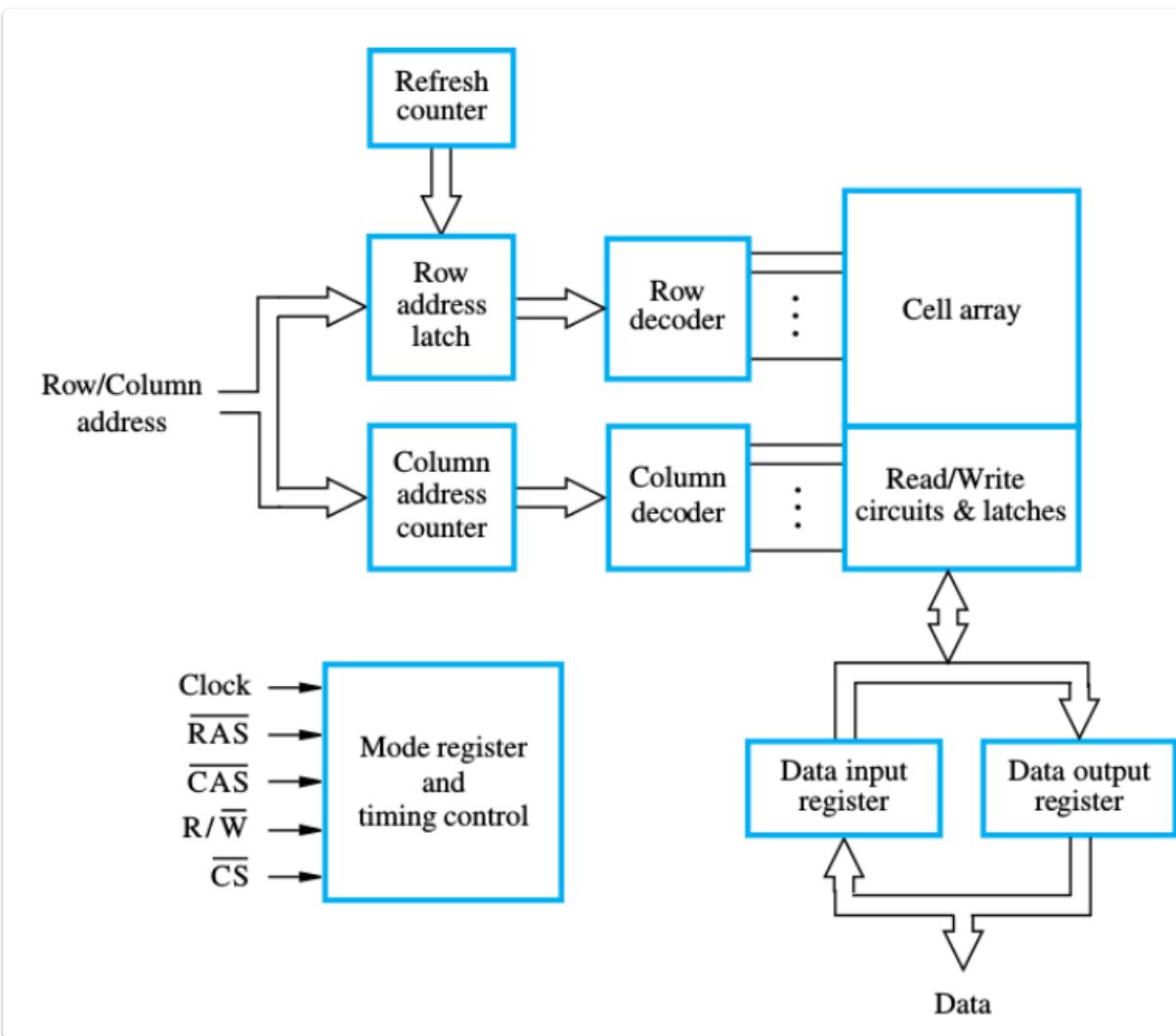
**CHIP**

- Architettura:



## 2.1 SDRAM

**CHIP SDRAM (Synchronous Dynamic RAM)**



- permette delle *lettura/scrittura multiple* "a burst"

## Memory Controller

E' un circuito che *controlla il flusso dei dati* tra CPU e chip di memoria.

*Coordina* le operazioni di lettura e scrittura e assicura una *comunicazione efficiente* tra CPU e *moduli di memoria* (componenti collegati alla motherboard che memorizzano temporaneamente dati e codice)

## ROM e PROM

Cos'è una memoria ROM? [1. Introduzione all'Architettura del Calcolatore](#)

### ROM

In una memoria ROM la cella di memoria contiene un "ponticello" (P) che viene inserito o rimosso durante il processo di fabbricazione

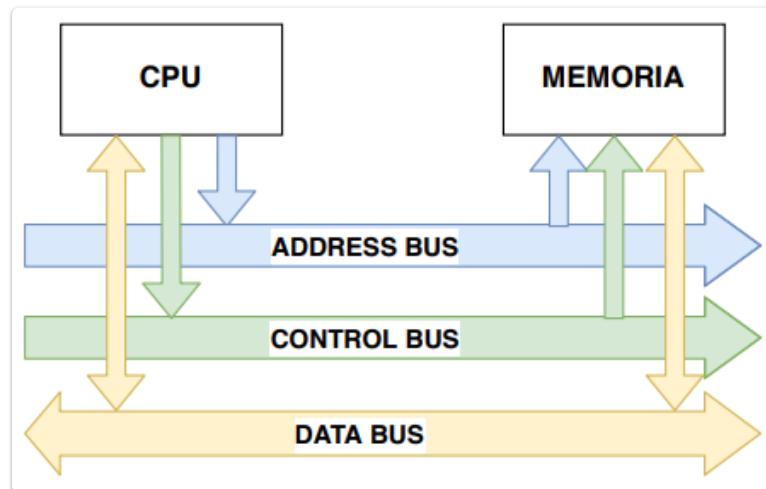
### PROM

La PROM è una ROM programmabile "once-time" il cui ponticello è un "fusibile" che può essere interrotto inviando una tensione superiore ad una certa soglia

## 12 Bus di Sistema

- [12.1 Bus Sincrono e Asincrono](#)

Architettura:



La connessione tra CPU e Memoria avviene tramite il Bus di Sistema

## Address Bus

Contiene i collegamenti che giungono alle *address lines* della memoria: *trasporta l'indirizzo* della cella da leggere o da scrivere

## Control Bus

Contiene i collegamenti che giungono alle *control lines* della memoria e permette alla CPU di *specificare l'operazione* (RD o WR).

Contiene anche molte altre linee legate alle periferiche di I/O

## Data Bus

Contiene i collegamenti delle *data lines* e consentono alla CPU di *scambiare con la memoria* il dato da leggere o da scrivere

## 12.1 Bus Sincrono e Asincrono

- [12 Bus di Sistema](#)

### Bus di Sistema, Input/Output e Sincronizzazione

Le operazioni tra CPU e periferica comportano uno *scambio di segnali elettrici* per cui è importante sia l'**aspetto logico** (livello di tensione) che l'**aspetto temporale**

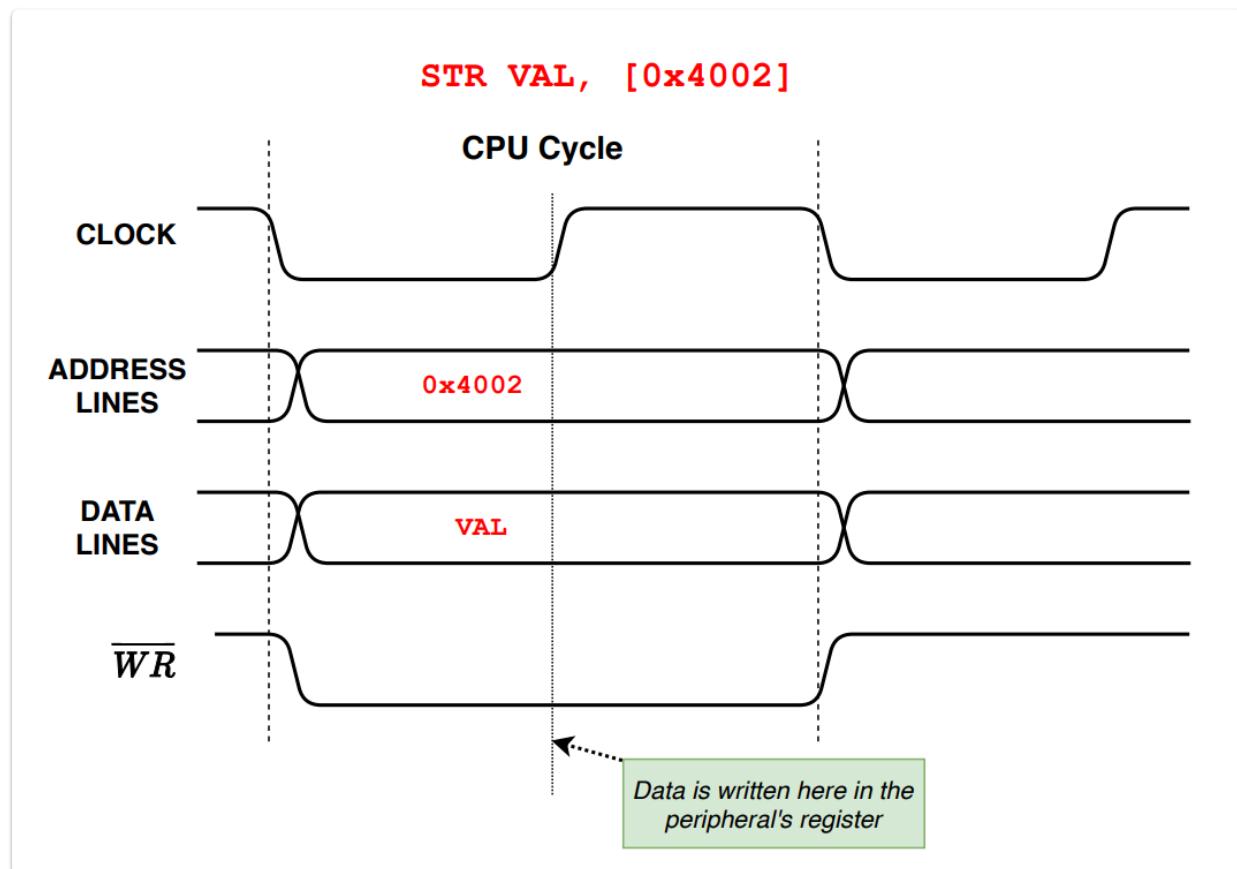
Tali interazioni sono regolate da un protocollo, che specifica l'andamento temporale dei segnali, di due tipi:

#### 1. Sincrono

E' presente un segnale di *clock* e gli eventi avvengono solo durante i fronti del *clock di sistema*

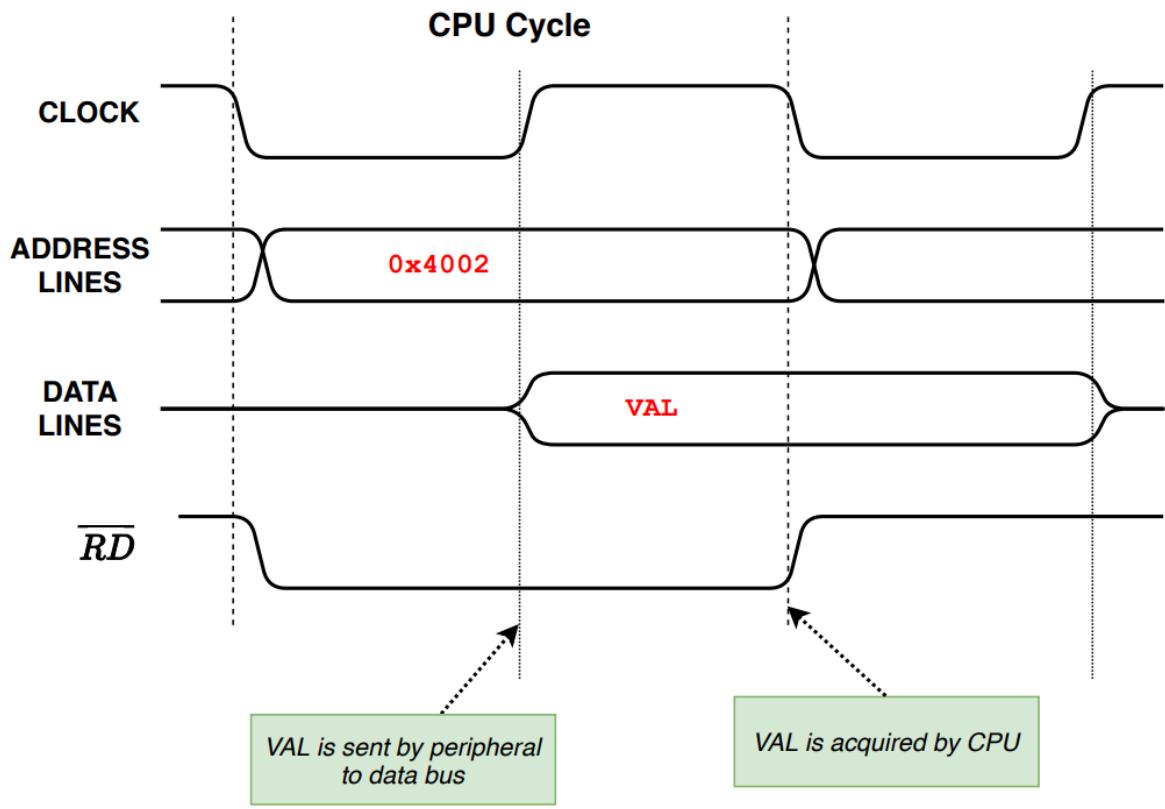
In fase di progettazione viene *assegnato* uno dei due fronti alla CPU e l'altro alla memoria/interfacce di I/O così da poter garantire la sincronizzazione stretta nell'interazione tra CPU e sistema periferico

#### Operazione di $\overline{WR}$



#### Operazione di $\overline{RD}$

## LDR REG, [0x4002]



## 2. Asincrono

Le operazioni avvengono solo all'occorrenza dei *fronti di segnali speciali* di **handshake**

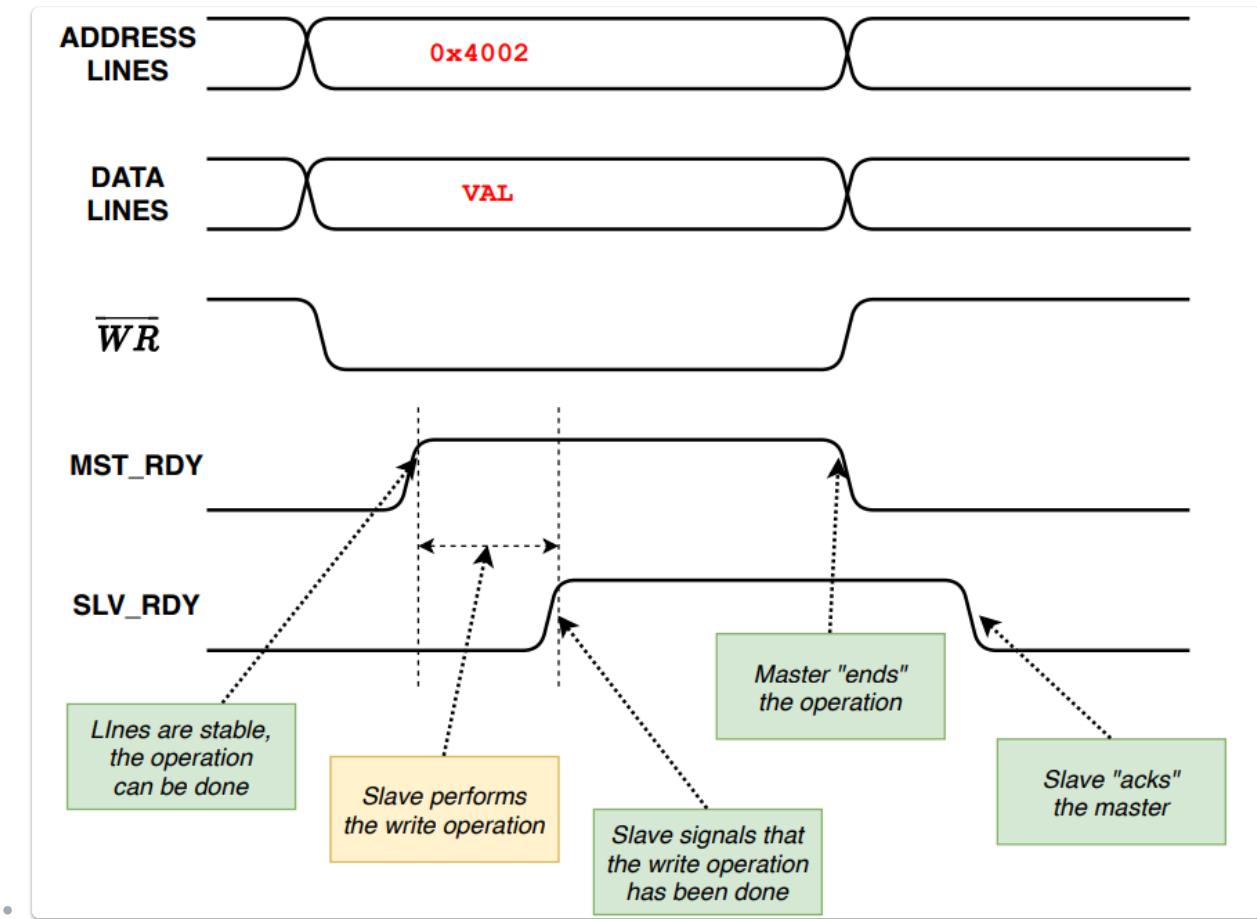
Il clock di sistema è presente ma *non regola le operazioni* poiché sono regolati dall':

- handshake, avviene attraverso segnali sul Bus di Controllo che regolano la segnalazione delle attività tra CPU, **master**, e periferica, **slave**

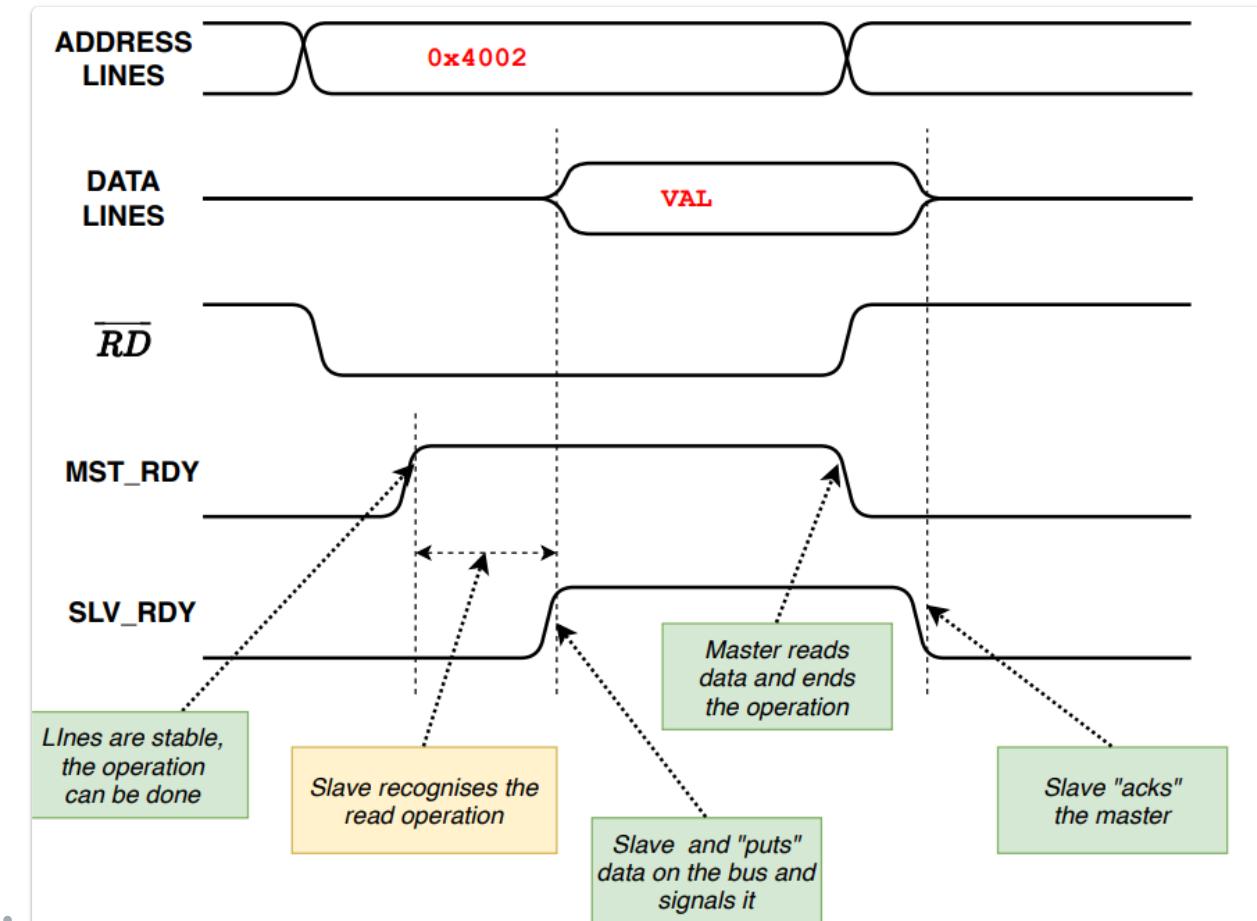
I segnali di controllo sono tipicamente due:

1. **Master-Ready** (CPU → periferica) in cui la CPU segnala che le informazioni su Bus dati/indirizzi sono valide
2. **Slave-Ready** (periferica → CPU) in cui la periferica segnala che l'operazione richiesta è stata effettuata

## Operazione di $\overline{WR}$



### Operazione di RD

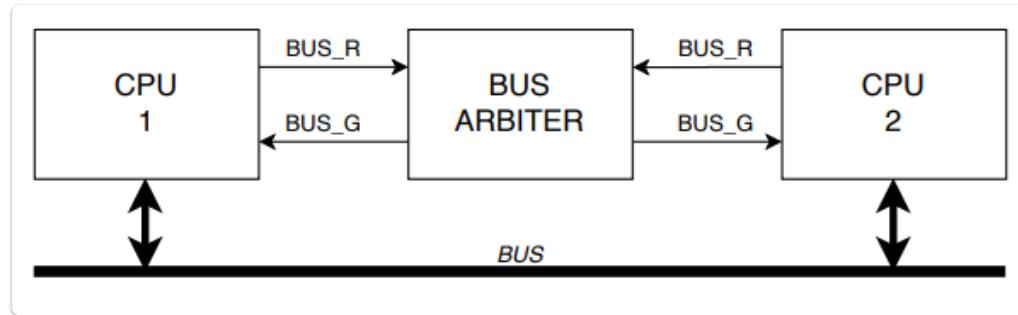


### Multi-Master e Arbitraggio

Nei computer multi-CPU *il bus è unico e condiviso* da tutte le CPU ma può essere acceduto una CPU per volta: il circuito **Bus Arbiter** gestisce questi accessi in modo mutuamente esclusivo attraverso:

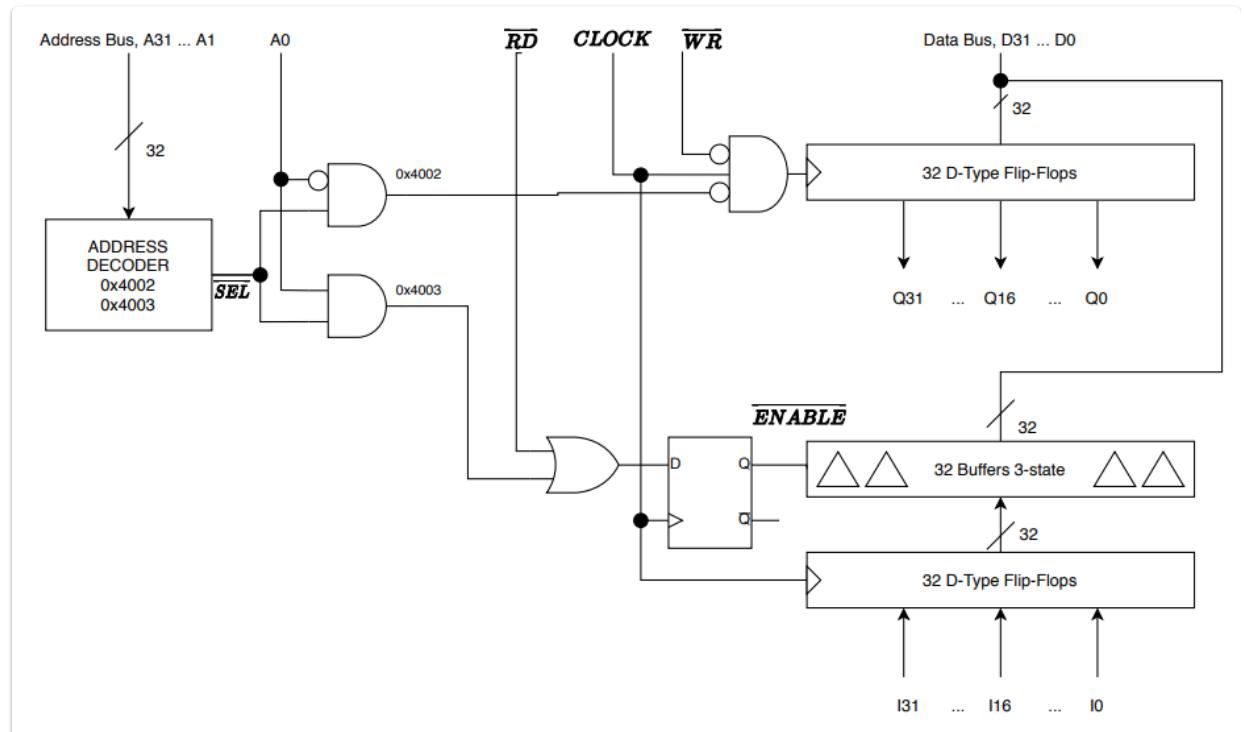
- *BUS\_R*, Bus Request ( $\rightarrow$  CPU)
- *BUS\_G*, Bus Grant ( $\rightarrow$  Bus Arbiter)

**Struttura:**



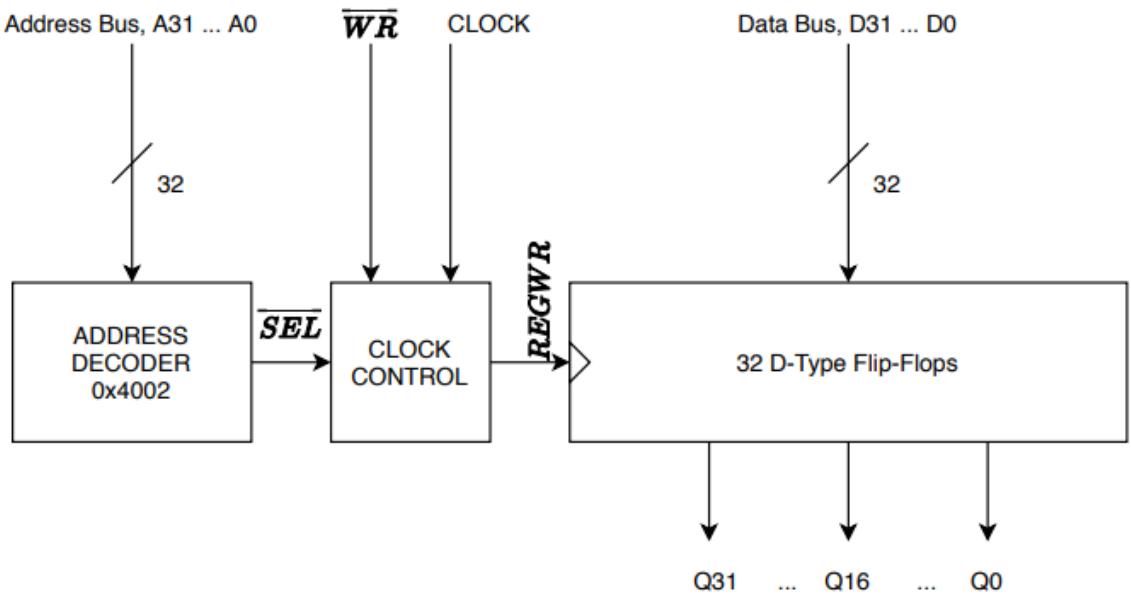
## Porta Digitale di Input/Output

**Circuito:**

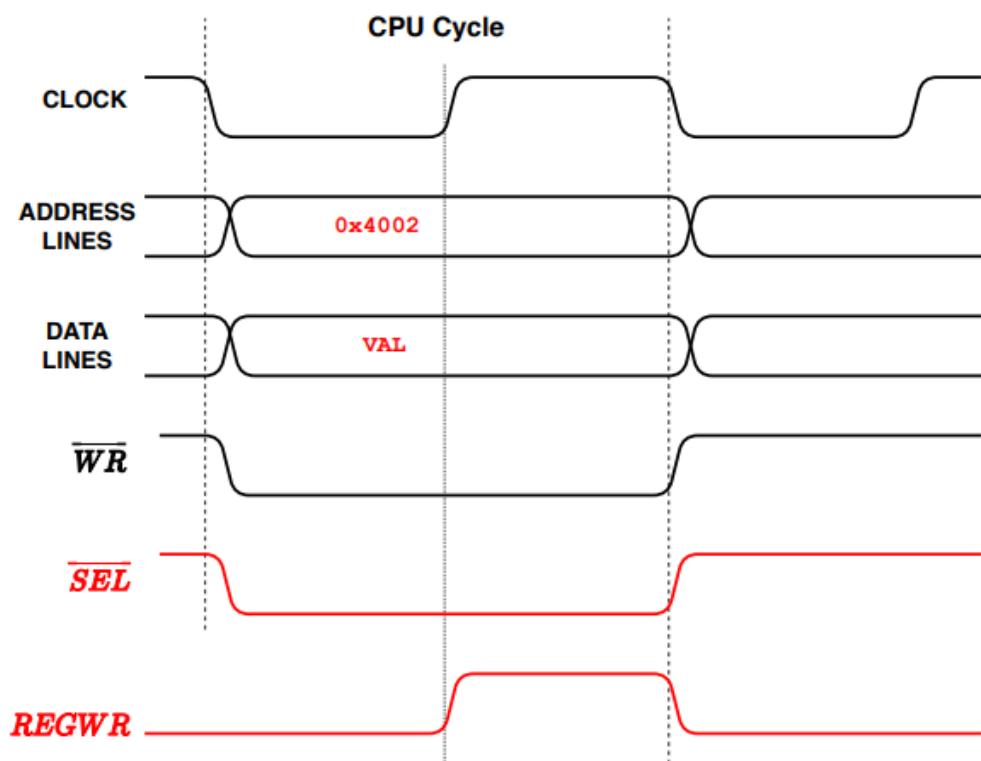


## Porta Digitale di Output

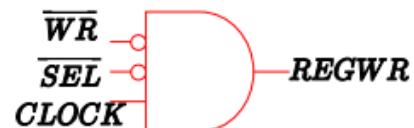
**Circuito:**



Fronti:

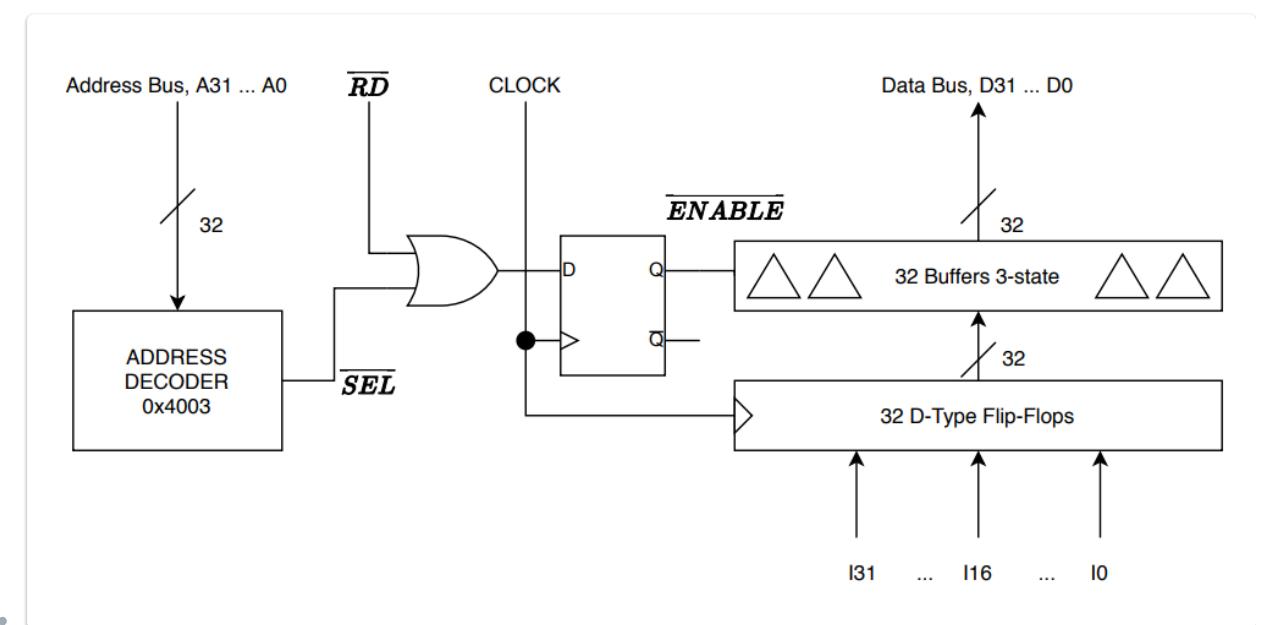


$$REGWR = CLOCK \overline{WR} \overline{SEL}$$

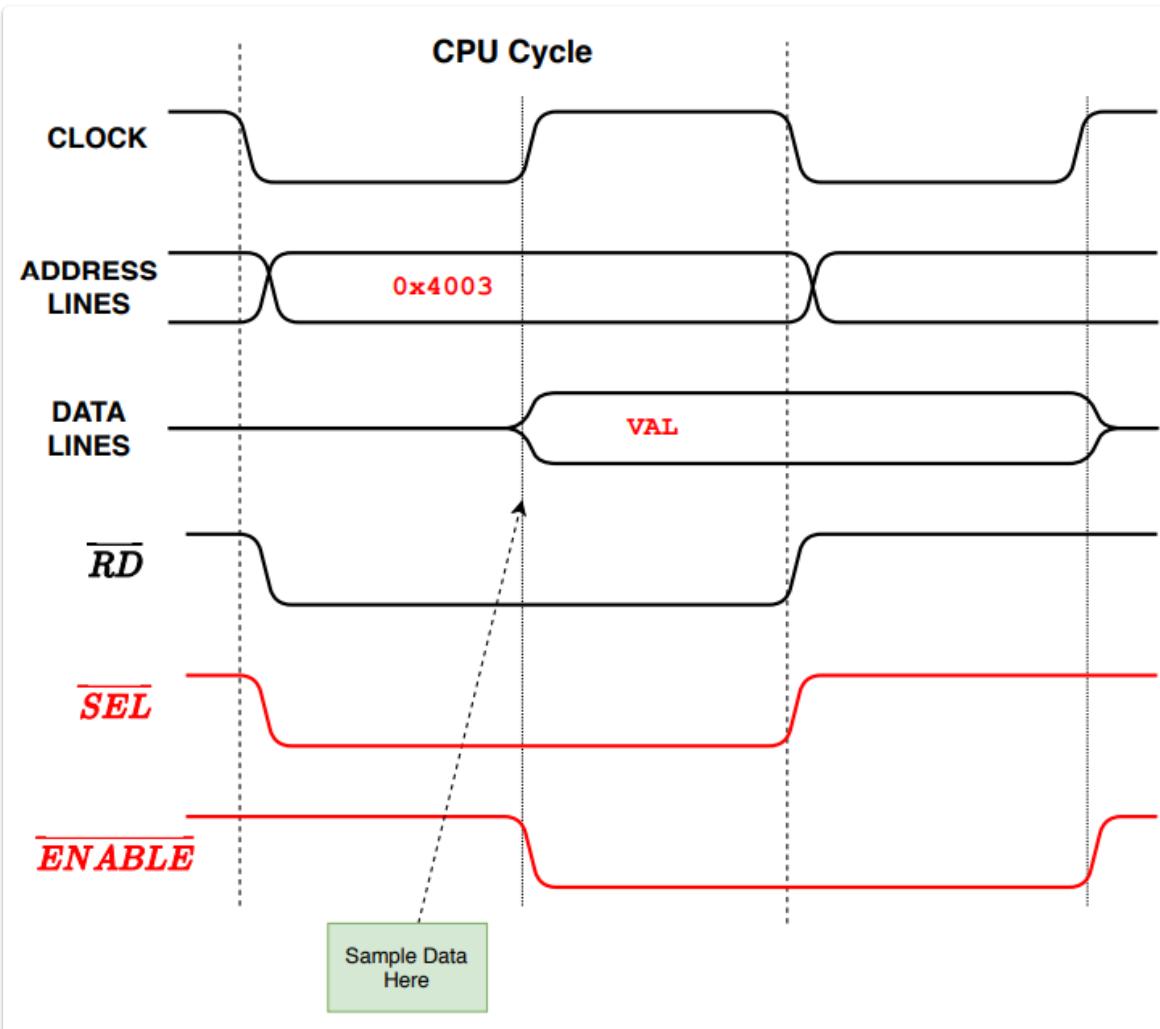


## Porta Digitale di Input

Circuito:



Fronti:



## Porta Seriale UART

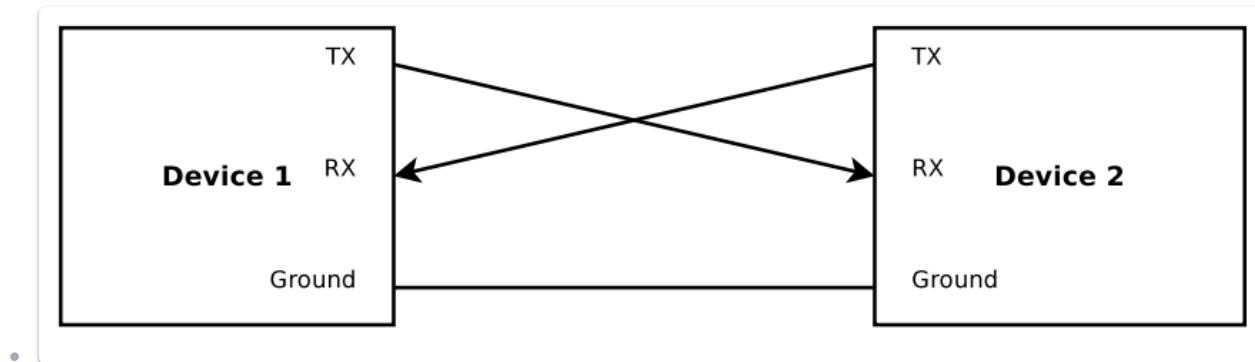
UART (Universal Asynchronous Receiver/Transmitter) è una *porta seriale* (o *COM port*), ossia una periferica per la *comunicazione punto-punto* tra due dispositivi a processore in cui la comunicazione avviene *in seriale*.

Parametri di trasmissione:

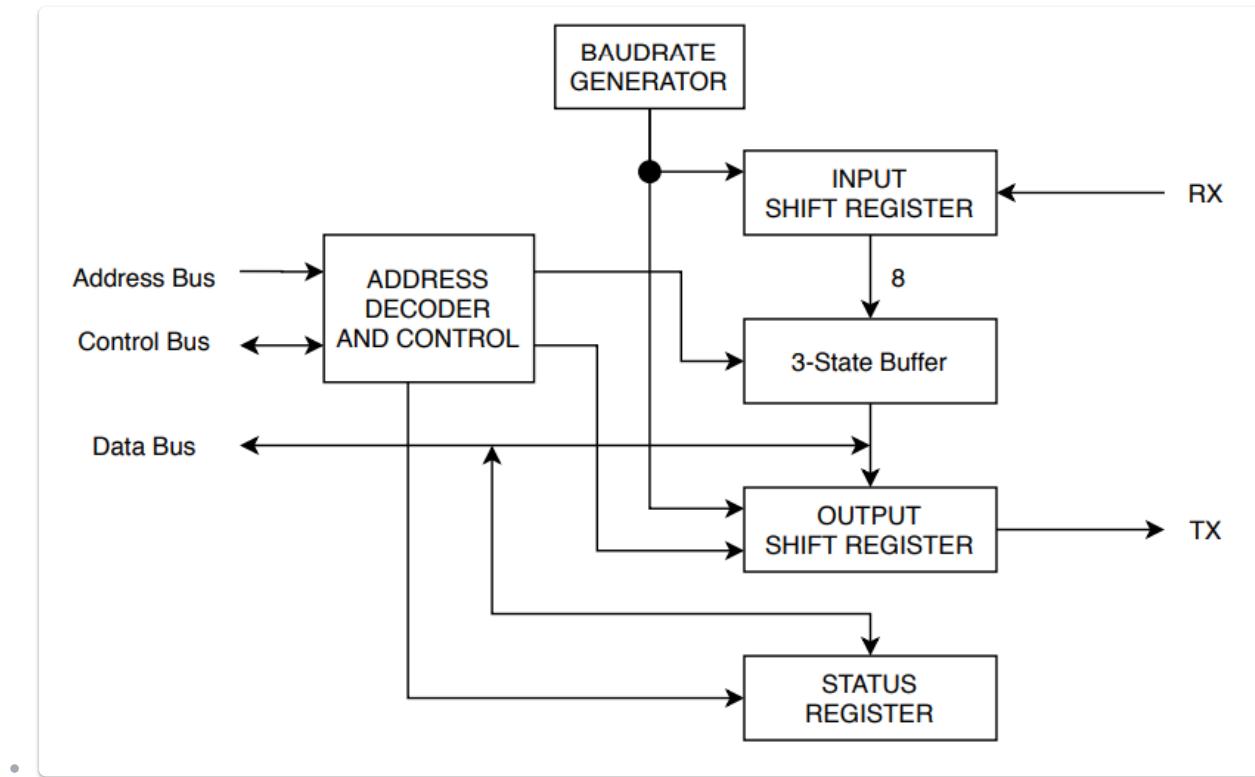
- *velocità di comunicazione*, in bps (bit per second) o baud

- *numero di bit per carattere*, normalmente 8
- *presenza/assenza del bit di parità*, usato per il controllo d'errore
- *numero di bit di stop*, normalmente 1

Interfaccia:



Schema concettuale:



### 13. Il sistema di Input e Output

Una caratteristica fondamentale di un computer è quella di poter interagire con il mondo esterno: questo avviene grazie alle **interfacce di Input/Output**, ossia dispositivi hardware che mettono a disposizione diverse funzionalità e collegamenti con il mondo esterno.

Esse interagiscono con la CPU attraverso i [12 Bus di Sistema](#).

Come fa la CPU a "vedere" un'interfaccia di I/O?

- attraverso l'uso di istruzioni che fanno riferimento ad *indirizzi specifici*

### I/O Mappato in Memoria

I dispositivi sono trattati come fossero delle *locazioni di memoria*

Si stabiliscono diverse funzionalità specifiche:

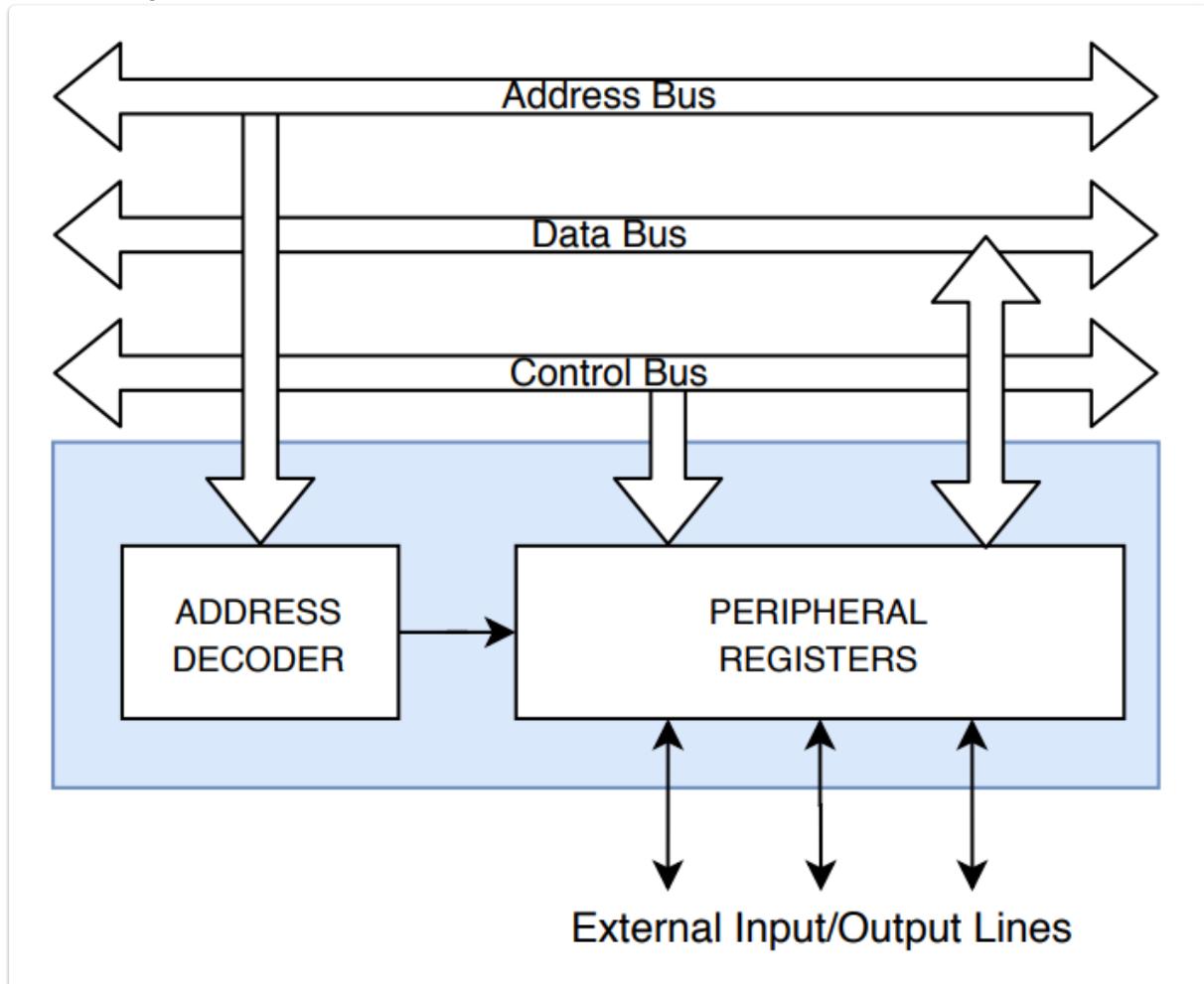
- una **parte della memoria** (stabilità a design-time) viene **riservata** per l'Input/Output
- un **set di indirizzi specifico** dell'area di memoria è assegnato all'Input/Output
- ogni interfaccia di I/O viene **allocata** (staticamente o dinamicamente) su uno o più indirizzi dell'**area riservata**
- l'accesso avviene attraverso operazioni di **LOAD/STORE** che utilizzano l'indirizzo assegnato all'interfaccia stessa

## Control Bus

Espone due linee che indicano l'operazione che si sta effettuando:

- $\overline{RD} = 0$  operazione di **READ/LOAD**
- $\overline{WR} = 0$  operazione di **WRITE/STORE**

**Architettura** generale:



## I/O su Spazio Separato

I dispositivi sono considerati come **entità separate**

Si stabiliscono diverse funzionalità specifiche:

- un **set di indirizzi separato** da quello della memoria centrale è assegnato all'Input/Output
- ogni interfaccia di I/O viene **allocata** (staticamente o dinamicamente) su uno o più indirizzi di **tale area**
- l'accesso avviene attraverso delle operazioni speciali di **IN/OUT** (differenti dalle **LOAD/STORE**) che utilizzano l'indirizzo assegnato all'interfaccia stessa

- [9.1 Inside the CPU](#)

La CPU esegue un programma fatto da istruzioni poste sequenzialmente in memoria. Durante l'esecuzione può accadere un evento che richiede "attenzione":

- viene inizialmente gestito dell'hardware che *attiva* uno o più bit nei registri della relativa periferica di I/O
- il software *interroga* (*poll*) le locazioni di memoria di quei registri per controllare se e quale evento si sia verificato

Il software normalmente fa altro e non può continuamente effettuare tale verifica. motivo per cui è stato introdotto il meccanismo di *interrupt*:

- quando avviene un evento l'hardware *attiva* uno o più bit nei registri della relativa periferica di I/O e *invia un segnale di interrupt alla CPU* attraverso una linea specifica del Control Bus
- la CPU *interrompe* l'esecuzione del programma corrente ed effettua un *JUMP* ad un indirizzo prestabilito dove deve trovarsi la routine che gestirà la richiesta di *interrupt* (*ISR - Interrupt Service Routine*)
- la ISR termina con l'istruzione *Return-from-Interrupt* e la CPU riprende l'esecuzione del programma precedentemente interrotto

## Identificazione

Ogni CPU è dotata di un *ingresso interrupt* ( $\overline{INT}$  o  $\overline{IRQ}$ ) che viene posto allo *stato logico 0* (fronte di discesa) quando una periferica intende *generare un interrupt*

L'interrupt viene *identificato prima* della fase di fetch in cui coinvolge l'**Interrupt Controller**:

- se l'ingresso dell'Interrupt è a 0 la CPU attiva la procedura di *gestione dell'interrupt*

## Gestione dell'Interrupt

### - senza Interrupt Controller

Come si comporta la CPU?

- salva l'attuale PC nello stack
- carica il PC con un indirizzo predefinito (**Interrupt Vector**)
- continua con il suo normale funzionamento

All'Interrupt Vector *serves* la ISR la quale:

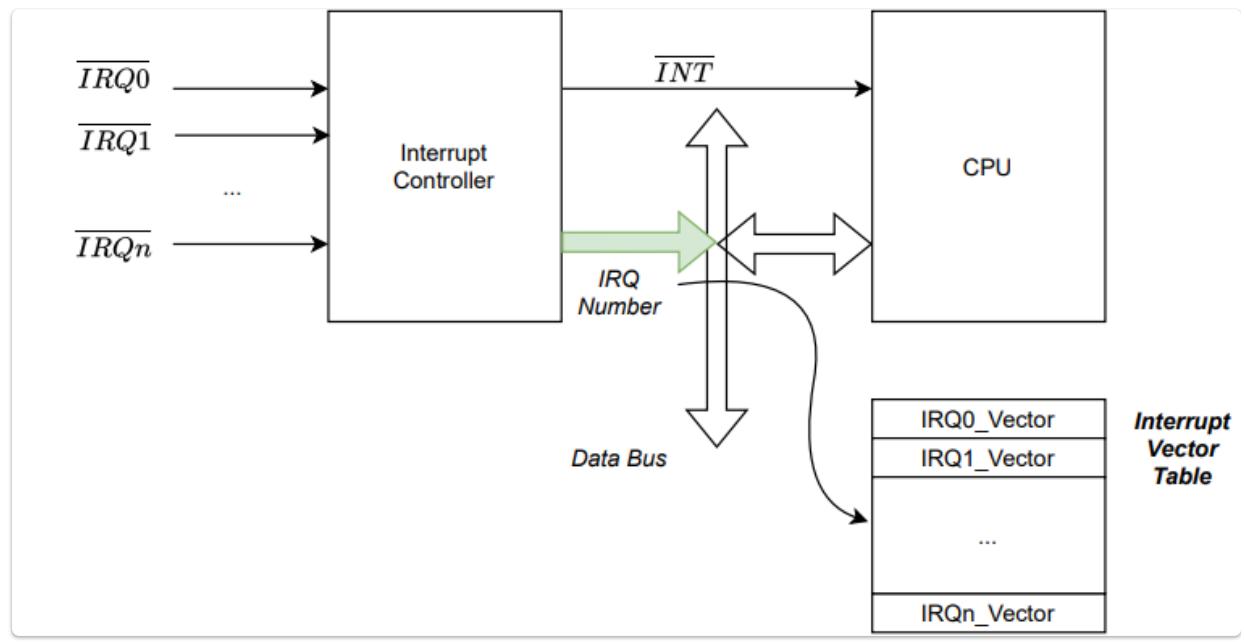
1. *salva* sullo stack i registri che utilizzerà (oppure tutti)
2. *esegue* le operazioni relative alla gestione dell'evento
3. *segnala* alla CPU che l'interrupt è stato servito
4. *recupera* i registri dallo stack
5. *esegue* una "Return-from-Interrupt"

### - con Interrupt Controller

Le interrupt possono essere generate da varie periferiche ma la CPU ha (in genere) un solo ingresso di interrupt Occorre pertanto un meccanismo di *multiplexing* che:

1. permetta di *collegare* diverse sorgenti di interrupt alla *stessa linea* di  $\overline{INT}$
2. consenta alla CPU di *capire* quale periferica ha *richiesto attenzione*

Questo lavoro è svolto dall'**Interrupt Controller**:



- quando un interrupt viene generato su una linea  $\overline{IRQ}$ , l'**IC** memorizza (in suo registro interno) il numero di  $\overline{IRQ}$  e gira la richiesta all' $\overline{INT}$  della CPU
- l'**IC** pone sul Data Bus il numero dell' $\overline{IRQ}$  che era stato attivato
- la CPU acquisisce l' $\overline{IRQ}$  number e lo usa come offset dell'Interrupt Vector Table (tabella di puntatori)
- l'elemento indicizzato viene usato come Interrupt Vector e viene eseguito il salto all'indirizzo specificato

## Exception

Sono degli eventi analoghi all'interrupt ma generati **internamente** dalla CPU

Avvengono in seguito ad **eventi inaspettati (eccezionali)** che possono accadere durante l'esecuzione di un programma:

- la CPU interrompe il programma
  - effettua un salto alla IRS associata all'evento
- Le entries della IVT contengono anche i vettori delle **eccezioni**

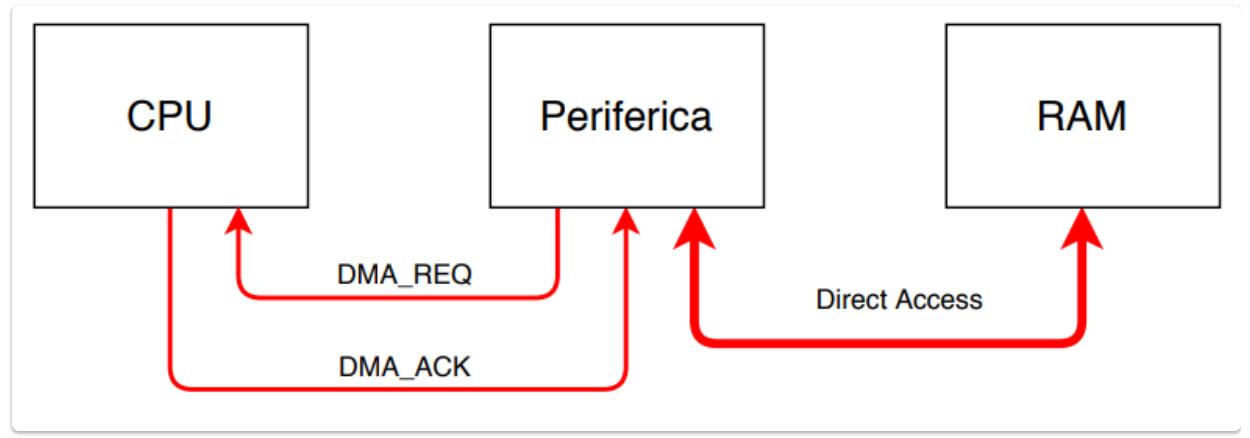
## Direct Memory Access (DMA)

Alcune periferiche possono avere la necessità di trasferire una grande mole di dati (ad esempio un **intero blocco di disco**)

Il trasferimento word-by-word potrebbe risultare lento e allora si adotta una tecnica detta **Direct Memory Access (DMA)**:

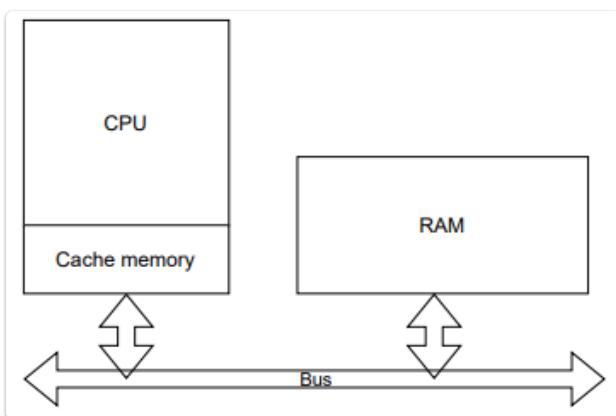
- la periferica ha la possibilità di accedere direttamente alla memoria di sistema senza prendendone il controllo
- dalla RAM possono essere prelevati uno o più blocchi di dati, senza coinvolgere la CPU

**Architettura** generale:



## 15. La Cache Memory

**Architettura:**



## Definizione e Funzionamento

La **Cache** (memoria *tampone*) è un dispositivo di memoria:

- *più veloce* della memoria centrale
- *meno capacità* di memoria
- *velocizza* operazioni di `LOAD` e `STORE` per i dati frequentemente utilizzati

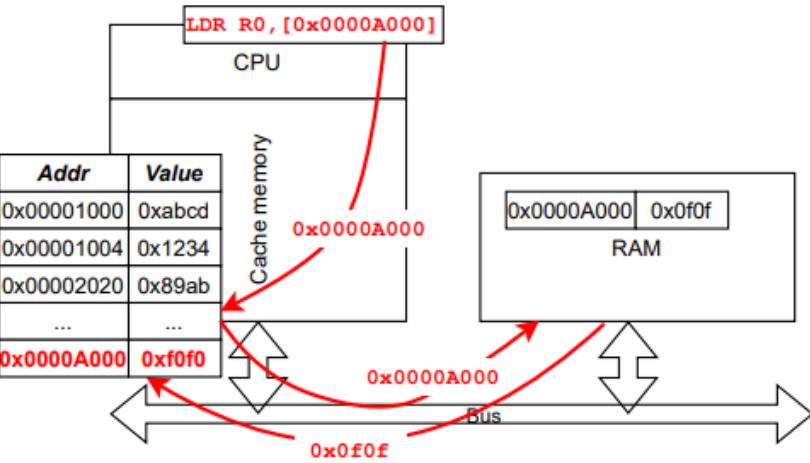
Come *funziona*?

- memorizza **copie** (*Address, Value*):
  - Address è l'indirizzo in memoria centrale
  - Value è la relativa word memorizzata

## Stati della Cache

### Cache Miss

**Schema:**

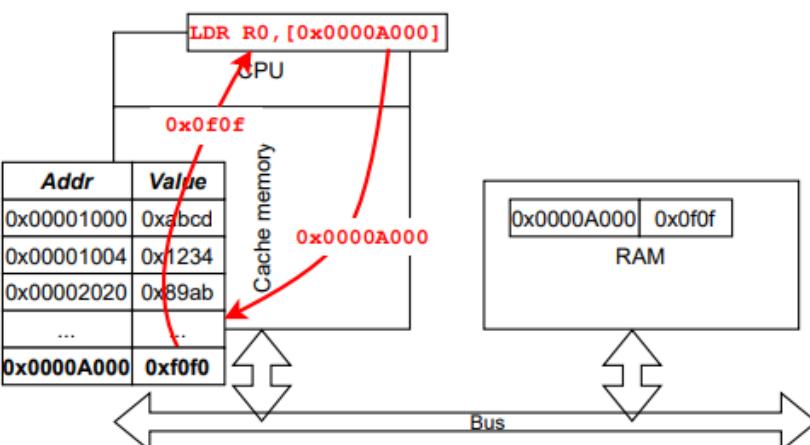


Si verifica nel momento in cui:

- una `LDR` provoca la ricerca dell'indirizzo nella cache
- l'indirizzo *non viene reperito* e si verifica un evento di **cache miss**
- il **cache controller** *recupera* il dato dalla memoria centrale e lo *inserisce* nella cache

## Cache Hit

Schema:



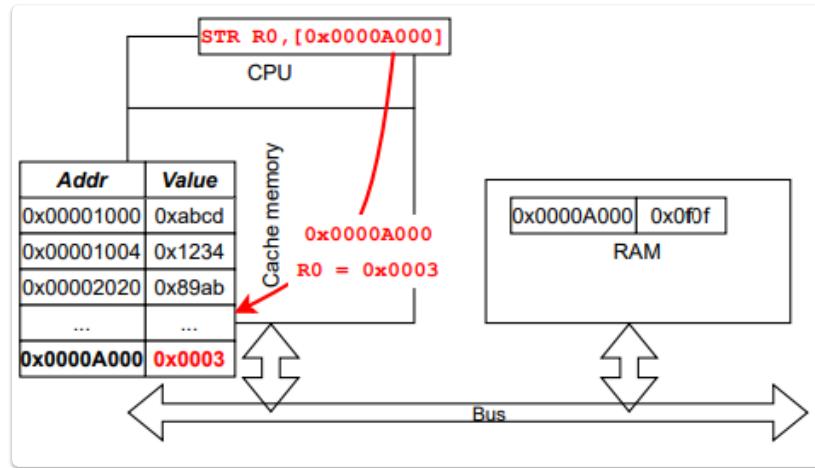
Si verifica nel momento in cui:

- una `LDR` provoca la ricerca dell'indirizzo nella cache
- l'indirizzo *viene reperito* e si verifica un evento di **cache hit**
- il **cache controller** *recupera* il dato dalla cache e lo *restituisce* senza coinvolgere memoria centrale e bus

## Operazioni

### Cache Write

Schema:

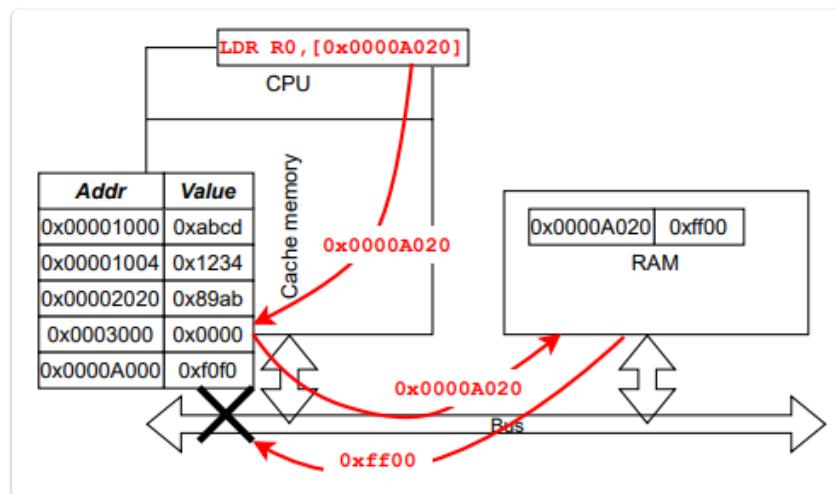


**Funzionamento:**

- una `STORE` provoca l'aggiornamento del valore all'indirizzo nella cache
- il valore in memoria centrale sarà *differente* ma poiché le `LOAD` fanno riferimento alla cache si otterrà sempre il dato aggiornato
- la locazione di memoria viene macchiata *dirty*

## Cache Full e Write-Back

La memoria cache è una memoria molto piccola, per questo può accadere che a seguito di una `LDR` con *cache miss* non ci sia più *spazio in memoria*:



Occorre:

- scegliere una *entry* della cache da liberare
- se l'*entry* è marchiata *dirty* occorre *aggiornare* il valore nella RAM (*write back*)

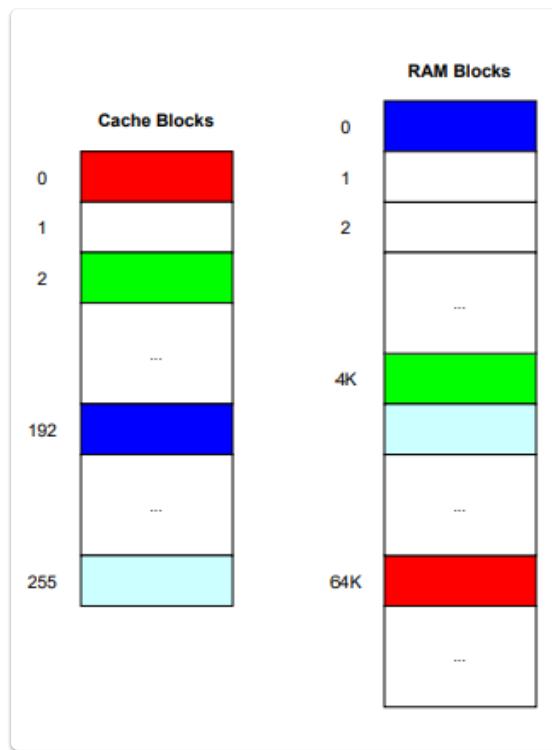
## Principio di Località

Può essere, data una locazione di memoria  $X$  acceduta al tempo  $t$ :

1. **spaziale**:
  - la probabilità che vengano accedute *locazioni contigue* a  $X$  nell'immediato futuro è molto elevata
2. **temporale**:
  - la probabilità che *essa* venga acceduta nuovamente nell'immediato futuro è molto elevata

## Organizzazione della Cache

**Organizzazione:**



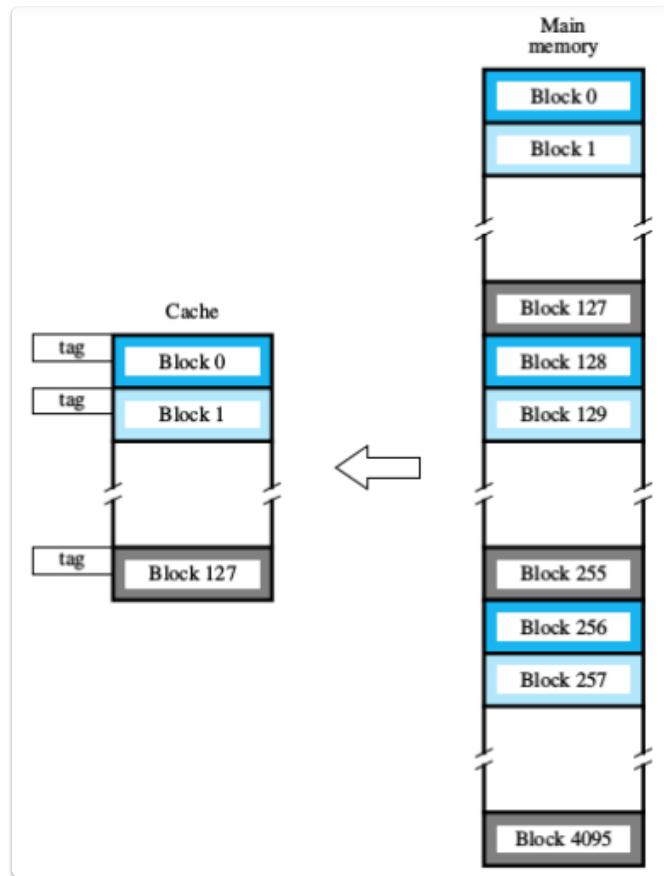
In virtù del principio di località *spaziale*:

- la cache è organizzata *a blocchi* o *pagine* di dimensioni costanti e potenze del 2 (generalmente 64 Byte)
- l'accesso tramite `LDR` o `STR` implica il coinvolgimento dell'*intero blocco* che contiene  $X$
- la cache include una **funzione di mapping** che lega il numero di un blocco della cache al numero del blocco nella RAM (e viceversa)

## Algoritmi di Mapping

### 1. Direct Mapping

**Schema:**



Un blocco  $j$  delle memoria centrale è mappato sul blocco  $j \bmod 128$  della cache:

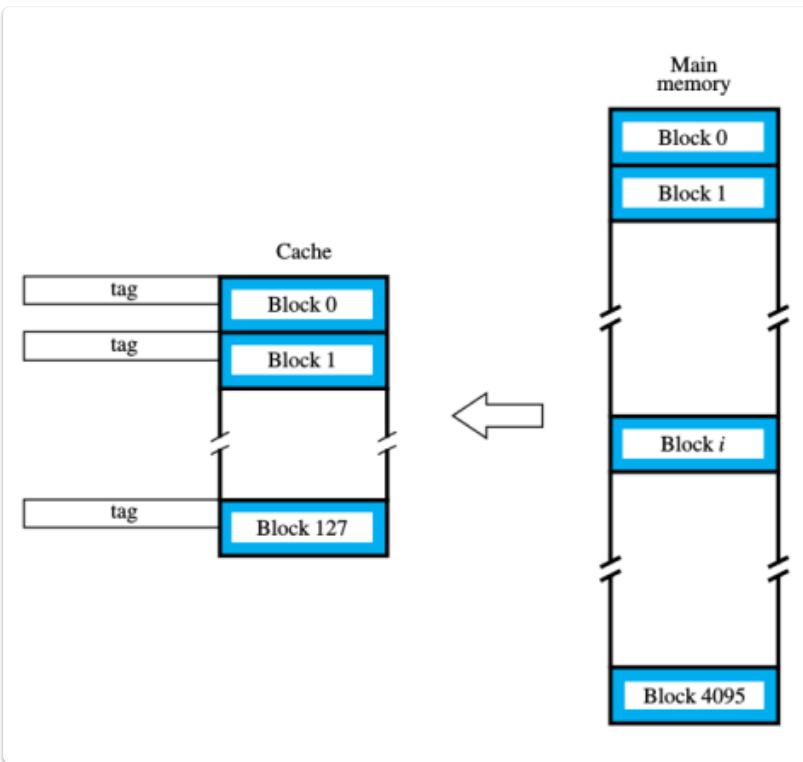
- ogni blocco contiene un **tag** utilizzato per ricostruire l'indirizzo del blocco in RAM

**Esempio** con indirizzo a 16 bit:

Tag	Block	Offset
3	7	6

## 2. Associative Mapping

**Schema:**



Un blocco  $j$  della memoria centrale è mappato su un blocco qualunque  $k$  della cache:

- ogni blocco contiene un **tag** utilizzato per ricostruire l'indirizzo del blocco in RAM

**Esempio** con un indirizzo a 16 bit:

Tag	Offset
10	6

### 3. LRU: Least Recently Used

**Funzionamento:**

- ogni blocco  $k$  della cache possiede un contatore  $C_k$  che rappresenta da quanto tempo il blocco non è stato riferito
- quando si accede al blocco  $k$  (**cache hit**), tutti i contatori degli altri blocchi che sono inferiori a  $C_k$  sono incrementati di 1:
  - $\forall j \neq k : C_j < C_k , C_j \leftarrow C_j + 1$
- quando accade un **cache miss** e *non ci sono più blocchi liberi*, si sceglie il blocco  $k$  con il  $C_k$  massimo, se *dirty* si effettua un *write-back*, e si ospita il nuovo blocco in arrivo dalla memoria nel blocco  $k$
- $C_k$  viene inizializzato a 0

## Gerarchia delle Memorie

