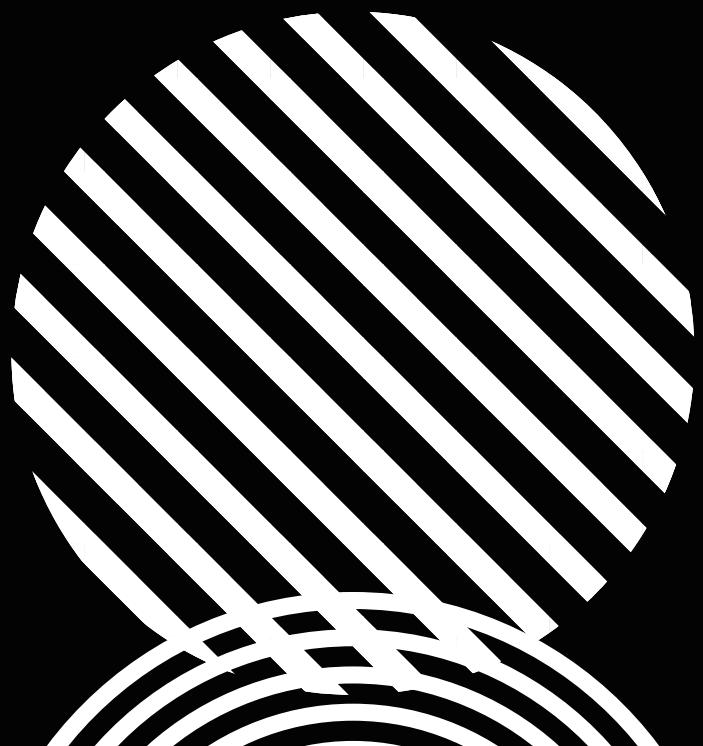


A.D.E



# SISTEMI NUMERICI

27/02/2023

Per rappresentare delle informazioni, il calcolatore usa il SISTEMA BINARIO, 0 e 1. Si potrebbe fare un'analogia con l'elettronica.

\* Per decifrare quindi le sequenze di 0 e 1 il calcolatore usa dei sistemi detti di "CODIFICA"

$$\begin{aligned} 8 \text{ bit} &= 2.56 \text{ inf. } (2^8) \\ k \text{ bit} &= 2^k \text{ combinazioni} \end{aligned}$$

**CODIFICA:** regole utilizzate nella rappresentazione di binario

POSSANO USARE PIÙ BIT PER RAPPRESENTARE DIVERSI TIPI DI INFORMAZIONI

ESEMPIO: byte (8 bit), nybble (4 bit), word (32 bit), half word (16 bit), doubleword (64 bit)

COME POSSO QUINDI RAPPRESENTARE UN'ENTITÀ?



**Rappresentazione:** un modo per descriverla, ha sue regole (un sistema)

Sistemi:

\* **decimale:** classico, è posizionale (ogni posizione di cifre è riferente, DECINE, CENTINAIA, MILLEIAIA)

\* **romano:** non posizionale, ogni lettera è un numero (I, X, V, L, C, M, ...)

\* **numeri posizionali:**

$$N = \sum_{i=-m}^{n-1} d_i \cdot r^i$$

\*  $d$  è la singola cifra

\*  $r$  è la radice base del sistema

$$\text{Esempio: } 123,45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

\*  $n$  è il numero di cifre della parte intera ( $n$ )

\*  $m$  è il numero di cifre della parte frazionaria ( $, m$ )

ESEMPIO

$$123,45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

\* **SISTEMA BINARIO**

base  $r = 2$   
cifre  $d = 0, 1$

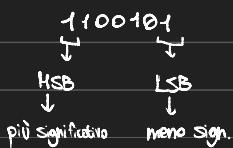
$$N = d_{n-1} \cdot 2^{n-1} + \dots + d_0 \cdot 2^0 + d_{-1} \cdot 2^{-1} + \dots + d_{-m} \cdot 2^{-m}$$

ESEMPIO (da binario a decimale)

$$101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{10}$$

→ byte = 8 bit

→ Word: 32 bit



\* **SISTEMA ESADECIMALE** →  $N_{16}$  oppure  $N_H$

• base  $r = 16$

• cifre  $d = 0, 1, 2, 3, \dots, 9, A, B, C, D, E, F$

$$N = d_{n-1} \cdot 16^{n-1} + \dots + d_0 \cdot 16^0 + d_{-1} \cdot 16^{-1} + d_{-m} \cdot 16^{-m}$$

ESEMPIO

$$A1_{16} = A \cdot 16^1 + 1 \cdot 16^0 = 161_{10}$$

La rappresentazione è **compatto**: uso meno cifre in genere per rappresentare un numero  
Usi: IP, ASCII, CSS

## CONVERSIONE TRA SISTEMI NUMERICI

\* da base  $r$  a base 10

$$N_r = d_{r-1} \cdot r^{r-1} + \dots + d_0 \cdot r^0 = M_{10}$$

esempio:

$$1010_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 2 = 10_{10}$$

$$\text{Kilo} = 1024 \quad (2^{10})$$

$$\text{Mega} = 1048576 \quad (2^{20})$$

$$\text{Giga} = 1073741824 \quad (2^{30})$$

\* da base 10 a base  $r$ 

→ divisione in colonna: dividere il numero per la base  $r$  fino a 0,  
poi scrivo il resto dal basso verso l'alto

esempio:  $120_{10} \rightarrow ???_8$

$$\begin{array}{r|rr}
 120 & 8 & 0 \\
 15 & 8 & 7 \\
 1 & 8 & 1 \\
 \hline
 & 0 &
 \end{array}
 \quad \begin{matrix} \uparrow \\ = 170_8 \end{matrix}$$

\* da base  $p$  a base  $q$  (qualsiasi)① converto da  $p$  a 10② converto da 10 a  $q$ 

PER LA CONVERSIONE DEVO FARE ATTENZIONE AL NUMERO DI CIFRE DISPONIBILI

→ in caso, potrebbe verificarsi un overflow nella conversione poiché le cifre non bastano

OPERAZIONI ARITMETICHE: sistema binario

$$\cdot 0 + 0 = 0$$

$$\cdot 0 + 1 = 1$$

$$\cdot 1 + 1 = 0 \quad \text{con resto } 1 \quad \left. \begin{array}{l} \text{con resto 1} \\ \text{riporto} \end{array} \right\}$$

esempio

$$\begin{array}{r}
 (19) \quad 0 \ 1 \ 0 \ 0 \ 1 \ 1 + \\
 (17) \quad 0 \ 1 \ 0 \ 0 \ 0 \ 1 \\
 \hline
 (36) \quad 1 \ 0 \ 0 \ 1 \ 0 \ 0
 \end{array}$$

SOMMA

$$\cdot 0 - 0 = 0$$

$$\cdot 1 - 0 = 1$$

$$\cdot 1 - 1 = 0$$

$$\cdot 0 - 1 = 1 \quad (\text{con prestito da bit immediatamente superiore})$$

SOTTRAZIONE

$$\begin{array}{r}
 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ - \\
 0 \ 1 \ 1 \ 1 \ 1 \ 0 \\
 \hline
 0 \ 1 \ 1 \ 1 \ 1 \ 1
 \end{array}$$

→  $0 - 1$  fa 1, ma il bit superiore diventerà 0

ATTENZIONE: se il riporto e il prestito non sono possibili si ha un OVERFLOW: non posso procedere

↳ la rappresentazione è troppo piccola, c'è bisogno di più bit

## MODULO e SEGNO



ha di disposizione 1 byte (8 bit)

\* i primi bit sono valutati assoluto

\* il bit più a sinistra (msb) è per il segno

### IL PROBLEMA DELLO 0

Ci sono due modi diversi di rappresentarlo

$$\begin{array}{l} 0000_2 = +0_{10} \\ 1000_2 = -0_{10} \end{array}$$

sono entrambe due rappresentazioni dello stesso numero, poiché "non ha segno" lo 0

$f = -$   
 $0 = +$   
SENO VAL ASSOLUTO

Quanti numeri posso rappresentare?

$$[-(2^n - 1), +(2^n - 1)]_{10}$$

**CRITICITÀ:** lo 0 ha due rapp., 1 bit è impiegato nel segno

## COMPLEMENTO A 1 (CA1)

→ il complemento è l'operazione che associa ad un bit (o ogni sequenza)

il suo opposto, cioè il valore ottenuto sostituendo tutti gli 1 con 0 e 0 con 1

$$1001 \xrightarrow{\text{compl}} 0110$$

### COME SI FA

Se il numero da codificare è:

(+) lo converti in binario nel classico metodo

$$3_{10} = 0011_2$$

(-) converti in binario il suo modulo e

quindi esegui l'operazione di complemento (metto i bit)

$$-3_{10} = 0011_2 = 1100_2$$

SUSSISTE IL PROBLEMA DELLO 0 ANCHE QUI, VERRÀ RISOLTO CON IL CA2

## COMPLEMENTO A 2 (CA2)

→ si usa solo per i numeri negativi

→ basato sul CA1

### COME SI FA

Se il numero da codificare è:

(+) invertito

(-) si fa il complemento a 1 sul valore da codificare, poi si somma +1 al risultato

### 3 METODI DI CALCOLO

1) Definizione

$$CA2(x) = 2^n - x$$

2) CA1+1

$$CA2(x) = CA1(x) + 1$$

3) Regola pratica

• parto da destra, trasocio tutti gli 0 fino al primo 1 e si trasociano anche quelli

• si fa il complemento a 1 di tutti gli altri bit

-16

esempio:  $-7_{10}$  in CA2 con 4 bit

### METODI

$$\textcircled{1} \quad CA2(-7) = 2^4 - 7 = 01000 - \underline{111} \\ \underline{1001}_{2^{n-2}}$$

$$\textcircled{2} \quad CA2(x) = CA1(x) + 1$$

$$CA1(-7) = 0111_2 = 1000_2$$

$$CA2 = 1000 + 1 = 1001_2$$

$$\textcircled{3} \quad \text{PRATICA}$$

$$7_{10} = 0111_2 \xrightarrow{\text{step 1}} ??1_2 \xrightarrow{\text{step 2}} 1001_2$$

# DA CA2 A DECIMALE

+ classica conversione del binario puro

- CA2 a questo valore, converto del binario puro } parte da destra, trascrivo tutti gli 0 fino al primo 1 e poi  
 (HSB=1) e aggiungo il segno - complemento tutto al resto → convertito in decimale

## OPERAZIONI MODULO E SEGNO

• Confronto i bit di segno

A) UUALI; bit di segno in testa, sommo i bit (controllo se overflow)

B) DIVERSI; \*confronto i due valori assoluti:

- \* il più grande darà il suo segno al risultato
- \* eseguo la differenza bit a bit

## SOMMA

## OVERFLOW OPERAZIONI

Si può avere solo quando:

- sommo due operandi con segno concorde
- sottraggo due operandi con segno discorde

Controllo la compatibilità del segno con operandi

$$+A + B = -C \quad \text{oppure} \quad -A + (-B) = -C$$

overflow

## SOTTRAZIONE

• Confronto i bit di segno

A) UUALI, prendo il bit di segno del val. assoluto più grande,  
 il risultato è il modulo della differenza dei moduli.

$$\begin{array}{c} \times |A| - |B| \\ \text{segno} \end{array} \quad \begin{array}{l} A = \text{più grande} \\ B = \text{più piccolo} \end{array}$$

B) DIVERSI, il bit di segno sarà quello del minuendo  
 e il risultato sarà il modulo della somma dei moduli

$$\begin{array}{c} \times |A| + |B| \\ \text{segno} \end{array}$$

## ESTENSIONE DEL SEGNO

Da CA2 di n bit a m bit dove m > n

⊕ aggiungo zeri:

$$+18 = \underline{\hspace{2cm}} 00010010$$

$$+18 = \underline{\hspace{2cm}} 00000000 \quad \underline{\hspace{2cm}} 0010010$$

⊖ aggiungo 1

$$-18 = \underline{\hspace{2cm}} 10110$$

$$-18 = 1111 \quad \underline{\hspace{2cm}} 10110$$

## OPERAZIONI IN CA2

## SUMMA

① eseguo la somma normalmente

② il riporto (carry) oltre il bit di segno viene scartato

→ vale sia per la somma sia per la sottrazione

③ Se gli operandi sono di segno concorde bisogna verificare se ci sia l'overflow (il risultato ha segno discorde rispetto agli addendi)

## ESEMPIO

$$\begin{array}{r} (+3) \quad 0000 \quad 0011 \\ +(-8) \quad 1111 \quad 1000 \\ \hline 1111 \quad 1011 \end{array} \quad \left| \begin{array}{l} (-2) \quad 1111 \quad 1110 \\ + (-5) \quad 1111 \quad 1011 \\ \hline (-7) \quad 1111 \quad 1001 \end{array} \right. \quad \downarrow \quad \text{scarto il carry}$$

## SOTTRAZIONE

Si ragiona come se fosse una somma tra segni discordi

$$A + (-B) = A - B$$

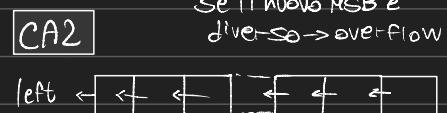
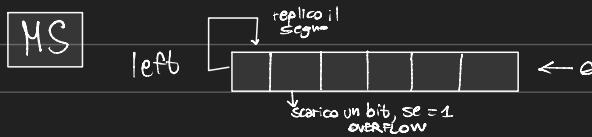


in questo caso, non c'è  
overflow

## SHIFT

→ sposta verso destra o sinistra la posizione delle cifre di un numero in base qualsiasi, inserisce 0 nelle posizioni rimaste libere

- left equivale a moltiplicare il numero per la base
- right equivale a dividere il numero per la base



## ECESSO 2<sup>n-1</sup>

$$X + 2^{n-1}$$

- $n = \text{bit per rappresentare l'eccesso}$

**REGOLA PRATICA:** bisogna complementare il bit più significativo (MSB) del CA2  $\rightarrow$  ottengo così l'eccesso

ESEMPIO: Eccesso 128 ( $2^7$ )

$$X + 128$$

$$-3_{10} \rightarrow -3 + 128 = 125 \rightarrow 01111101_2$$

Usando il CA2 (e quindi eccesso 128)

$$\rightarrow \underbrace{00000011_2}_3 \rightarrow 11111100_{CA1} \rightarrow 11111101_{CA2} \xrightarrow{\text{complemento MSB}} \underbrace{01111101}_{\substack{128 \\ \text{eccesso}}}$$

**RAPPRESENTAZIONE DEI REALI** → criticità: underflow, la parte frazionaria ha troppi pochi bit, i valori più piccoli perdono di precisione

I reali possono essere rappresentati come

\* **virgola fissa (fixed point)**: conosco la posizione della virgola,

è implicita e uguale per tutti i dati di quel tipo (numero)  $\rightarrow$  ovviamente si può scegliere quanti bit dedicare a decimali int.

↓  
• rappresenta solo i decimali positivi

• **Unsigned fixed point**: dati N bit a disposizione

$$i+d=N \quad \left\{ \begin{array}{l} A) 1 < N \text{ bit per rappresentare la parte intera} \\ B) D = N-1 \text{ bit per la parte decimale} \end{array} \right.$$

$$\left[ 0, 2^i - 1 \right] \quad \left[ 0, 2^D - 1 \right] \quad \left\{ \begin{array}{l} \text{Intera} \\ \text{Decim.} \end{array} \right.$$

• **SIGNED fixed point**: dati N bit

- 1 bit per il segno

    almeno 1 bit libero per il segno

-  $1 < (N-1)$  bit per intera

    bit eclettico: nullo se segno

-  $D = N - (i+1)$  bit per la parte decimale

$$\left[ 0, 2^D - 1 \right]$$

### CONVERSAZIONE VIRGOLA<sub>10</sub> IN VIRGOLA<sub>2</sub>

- LA PARTE INTERA LA CONVERTO IN BINARIO PURO

- LA PARTE DECIMALE viene scomposta, moltiplicata per 2 finché non trova 1, poi scrivo la sequenza in ordine ↓

$$\begin{aligned} 0,25 \times 2 &= 0,5 \rightarrow 0 \\ 0,5 \times 2 &= 1 \rightarrow 1 \end{aligned} \quad \left. \begin{array}{l} 0,5 \\ 1 \end{array} \right\} 1,0$$

### SVANTAGGI

\* rigidità della posizione della virgola (bit fissi)

\* impatta la precisione della parte intera (potre: non aver tutti i bit necessari)

## \* Virgola mobile:

$$N = (-1)^s \cdot M \cdot B^{\pm e}$$

↑ segno  
↑ mantissa  
↑ base  
↑ esponente

(es. notazione scientifica)

$$\downarrow$$

mantissa con segno  $\pm$   
 $\downarrow$   
 $M \cdot 10^{-2}$     { 10 = base  
                   { -2 = esponente

due rappresentazioni della virgola mobile:

- 1) Non normalizzato: più cifre intere per la mantissa (3639...5678,3)
- 2) Normalizzato: una sola cifra intera per la mantissa (1,xxx)

notazione scientifica e base 2

$$(-1)^s \cdot (1 + 0.m) \cdot 2^e$$

$\left\{ \begin{array}{l} s = \text{segno} \\ e = \text{esponente} \\ m = \text{mantissa} \end{array} \right.$

## ERROI RAPPRESENTAZIONI REALI

→ possono essere erretti di approssimazione

- Singola precisione a 32 bit



Se il numero è NORMALIZZATO,  
il primo bit della parte intera della mantissa  
in virgola mobile è detto "bit nascosto"

- doppia precisione a 64 bit



le cifre significative hanno un numero limitato a seconda della  
grandezza del numero

→ ERRORE ASSOLUTO:  $e_a = n - n'$       ~ dipende dalle cifre significative  
e l'ordine di grandezza del numero

→ ERRORE RELATIVO:  $e_r = e_a/n = (n - n')/n$

più il risultato reale è  
"diverso" più l'errore è grande  
(margine di errore)

## STANDARD IEEE 785 (approfondimento)

È uno standard comitato nel 1989 per la rappresentazione dei numeri  
in virgola mobile. È in uso ancora oggi. Non è proprietario (non dipende dall'architettura dell'elaboratore)

RAPPRESENTA L'ESPOLENTE IN ECESSO 127 con intervallo  $[-127, 128]$

## ALTRI TIPI DI INFORMAZIONI

I calcolatori possono rappresentare più informazioni

### \* CARATTERI

- ASCII standard: 1 carattere con 7 bit, 128 simboli

- LETTERE MAIUSCOLE [65-90]
- LETTERE MINUSCOLE [97-122]
- CIFRE [48-57]
- CARATTERI DI CONTROLLO [0-31]

- ASCII estesa: 1 carattere con 8 bit, 256 simboli

- UNICODE: 1 carattere è rappresentato tra 8 e 32 bit (quantità variabile), dipende dalla versione di codifica (il più comune è 8)  
 ↓  
 usato per l'inclusione di più lingue internazionali

\* più di 4 miliardi di caratteri

UTF-8: 8 bit, è used in JAVA per le stringhe

## CIRCUITI Logici

→ realizzati come circuiti integrati su chip di silicio  
↓

- Ci sono porte (gate) e fili depositati su chip di silicio, insieme, in un package e collegati all'esterno con un certo insieme di pin (pinout)

diversi gradi di integrazione

- SSI (Small Scale Integrated): 1-10 porte
- MSI (Medium // //): 10-100 porte
- LSI (Large // //): 100-100.000 porte
- VLSI (Very Large): > 100.000 porte

\* SSI: poche porte, collegate ai pin esterni.  
\* MSI: alcuni componenti base ancora usate oggi.  
\* VLSI: possono contenere una CPU

## INGRESSI e USCITE

↳ assumono 2 valori: → SEGNALE ALTO (1 per convenzione)  
→ SEGNALE BASSO (0 per convenzione)

## VALORI BINARI

- FALSO: segnali con voltaggio  $\leq 1$
  - VERO: segnali con voltaggio  $> 1$
- 

## TIPI DI CIRCUITI

• Un circuito (o rete) **combinatorio** è quel circuito in cui lo stato delle uscite dipende solo dalla funzione logica applicata alle sue entrate;

• Un circuito **sequenziale** è quel circuito cui lo stato delle uscite non dipende solo dalla funzione logica applicata agli ingressi, ma anche sulla base di valori collocati in memoria

## PORTE LOGICHE

→ Componenti elettronici che permettono di svolgere le operazioni logiche primitive oltre che a quelle direttamente derivabili

• Una porta logica è un circuito elettronico che dati dei segnali 0 e 1 input produce un segnale in output risultato di operazione booleana sugli ingressi

• Le porte logiche hanno n input e generalmente 1 output  
→ ad ogni combinazione di valori input corrisponde una sola combinazione output

• Porte logiche **FONDAMENTALI**: AND, OR, NOT

• Porte logiche **DERIVATE**: NAND, NOR, XOR

## FONDAMENTALI

### AND

→ prodotto logico

due entrate A e B



A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

### OR

somma logica



A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

### NOT



A	$\bar{A}$
0	1
1	0

La porta logica NOT è l'unica ad accettare un solo input, altre accettano anche N input

Nella pratica, con porte a 2 ingressi a disposizione soltanto, posso costruire circuiti a cascata in modo da ottenere comunque i miei N ingressi.

Ad esempio:

AND a 3 ingressi



**DERIVATE** → sono derivate delle fondamentali, realizzate combinando alcune porte logiche

• **NAND**



AND+NOT

negazione dell'AND

A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

• **NOR**



"NOT-OR"

A	B	out	A	B	NOR
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	1	1	0

• **XOR**

Restituisce 0 se i due input  
sono uguali, altrimenti 1



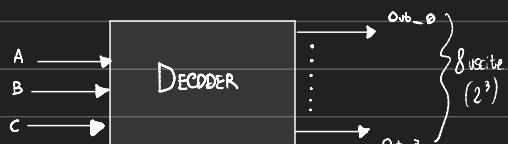
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

**DECODER**

Dispositivo da m ingressi e 2<sup>n</sup> uscite, serve per associare una codifica m base 10  
all'input richiesto (in binario)

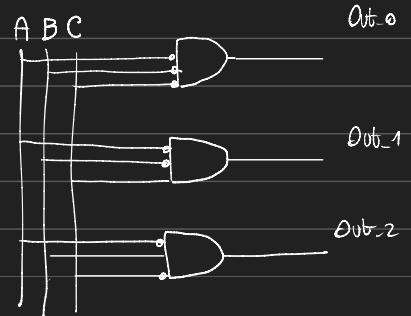
↳ \* gli input formano un numero senza segno

valore binario corrispondente



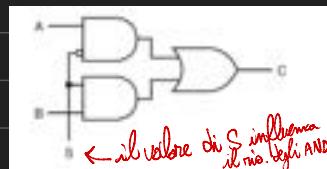
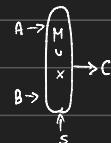
A	B	C	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

assume 1 lungo la diagonale



P = selettore

### MULTIPLEXOR (o selettore)



\* 2<sup>n</sup> entrate principali

\* n entrate di controllo (selettore)

\* 1 uscita

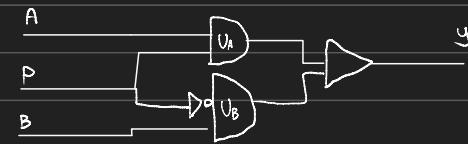
→ Il valore del selettore determina quale input diviene output

\* se ha in input n segnali, avrà bisogno di log<sub>2</sub>n selettori, e considererà quindi in:

a) un decoder che genera n segnali

b) array di porte logiche AND

c) una sola porta logica OR



quando P = 0

$$\begin{array}{l} P=0 \text{ e } A=0 \Rightarrow B \\ P=1 \text{ e } A=1 \Rightarrow 1 \end{array} \quad \text{OR} = 1$$

### LOGICHE A DUE LIVELLI

→ Combinazioni di porte primitive per creare di più complesse

DUE TIPI:

- SOMMA DI PRODOTTI: somma logica (OR) di prodotti (AND)
- PRODOTTO DI SOMME: prodotto (AND) di somme (OR)

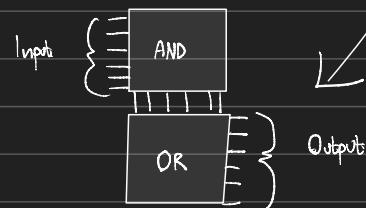
Notezionali: A vuol dire 0, A invece 1

La somma di prodotti è anche riconosciuta come PLA (Programmable Logic Array)

costituita da:

- Un insieme di input
- i corrispondenti input complementari (inverter) per gestire più uscite
- Logica a due passaggi:

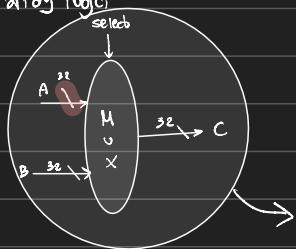
- 1) Porte AND (prodotti)
- 2) Porte OR (somme)



### ARRAY DI ELEMENTI LOGICI

• Le operazioni sono per la maggiore eseguite su 32 bit,

c'è bisogno degli array logici



bus: collezione di linee in più convergenti in un solo segnale

è un array di multiplexor a 1-bit (32 multiplexor)

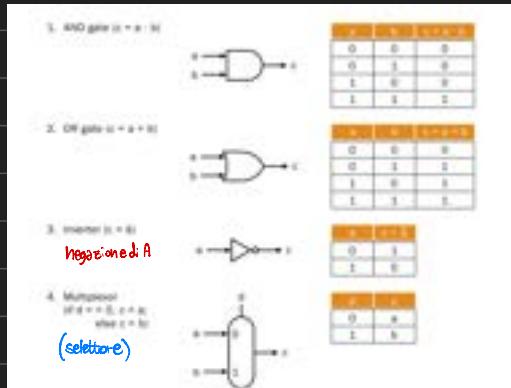
### ALU

Aritmetico Logic Unit:

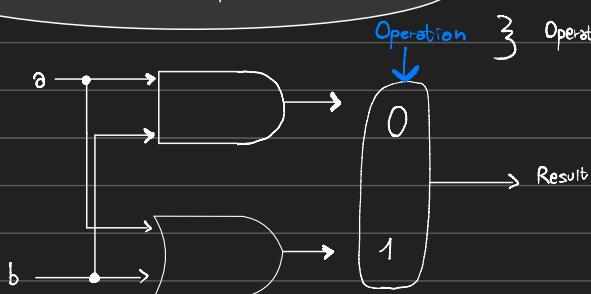
\* parte del processore che esegue le operazioni aritmetico-logiche

\* insieme di circuiti combinatori, combina operazioni aritmetiche (somma e sottrazione) e op. logiche (AND e OR)

Da cosa è formato l'ALU?



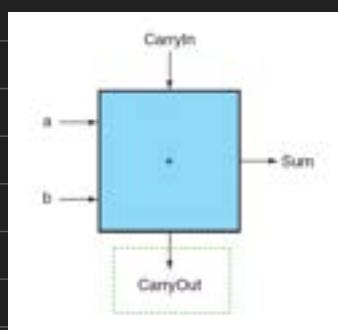
ALU su 1 bit che implementa AND e OR



Operation → Operation è un valore fornito dalla CPU per selezionare l'operazione (AND oppure OR)

## OPERAZIONI ARITMETICHE

\* ADDER (bit a bit)



→ L'ADDER fa la somma su ogni cifra, per 32 bit servono 32 ADDER

→ Il CARRY conserva il riporto della SOMMA PRECEDENTE e lo RIPORTA in quella nuova

→ In SUM viene messo il risultato della somma

CarryIn = il carry precedente in ingresso  
CarryOut = il nuovo carry risultato dell'operazione

• ADDIZIONE - TABELLA DI VERITÀ CON CARRYIN e CARRYOUT

a	b	CarryIn	CarryOut	Somma
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

faccio ogni possibile combinazione  
di carryIn 0 e 1

per costruire l'equazione, guardo quando questo è = 1

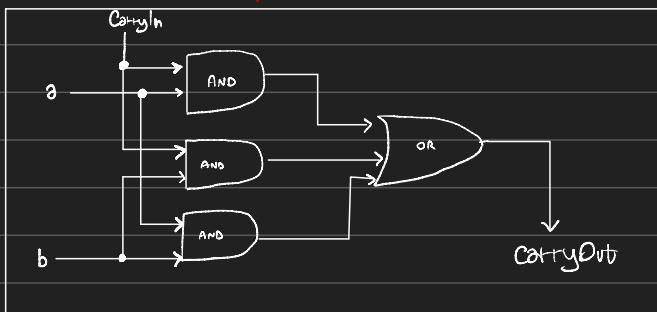
$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

posso rimuoverlo perché tutti i  $a$  e  $b$  devono essere per forza 1 per fare tot=1

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

- ADDIZIONE ~ HARDWARE

TRAMITE EQUAZIONE, COSTRUISSO IL CIRCUITO



il circuito è analogo all'equazione

$$\text{carryOut} = (b \cdot \text{carryIn}) + (a \cdot \text{carryIn}) + (a \cdot b)$$

infatti AND=moltiplicazione e OR=addizione

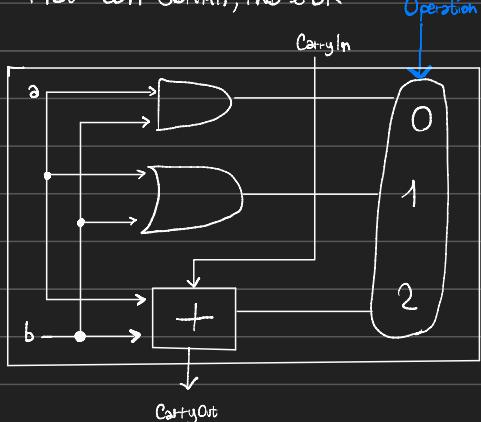
Adesso, ragiono su quando la somma = 1

- A)  $a=0, b=0, \text{carryIn}=1$
- B)  $a=0, b=1, \text{carryIn}=0$
- C)  $a=1, b=0, \text{carryIn}=0$
- D)  $a=1, b=1, \text{carryIn}=1$

l'equazione della somma quindi sarà  $\Sigma = 0$

$$\text{Sum} = (\bar{a} \cdot \bar{b} \cdot \text{carryIn}) + (\bar{a} \cdot b \cdot \overline{\text{carryIn}}) + (a \cdot \bar{b} \cdot \overline{\text{carryIn}}) + (a \cdot b \cdot \text{carryIn})$$

- ALU con SOMMA, AND e OR



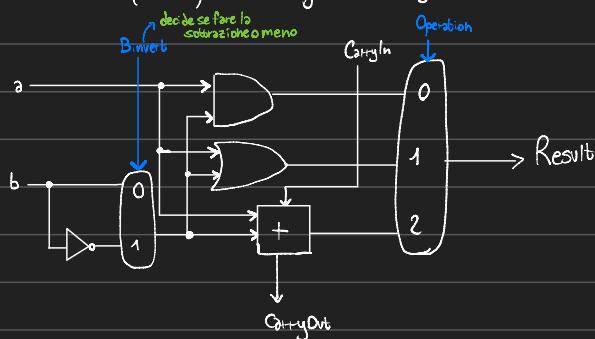
In pratica

\* IN BASE AL VALORE DI Operation nel MULTIPLEXOR,

l'ALU esegue

- A) 0 = AND (moltiplicazione)
- B) 1 = OR (somma)
- C) 2 = ADDER (somma con riporto)

- SOTTRAZIONE (in CA2)  $\rightarrow$  si nega b e carryIn=1  $\rightarrow a - b = a + (\bar{b} + 1)$



1° (ESSE)  $\neg b / (p \text{ AND } q) = \neg b \text{ AND } \neg p \text{ and } \neg q$

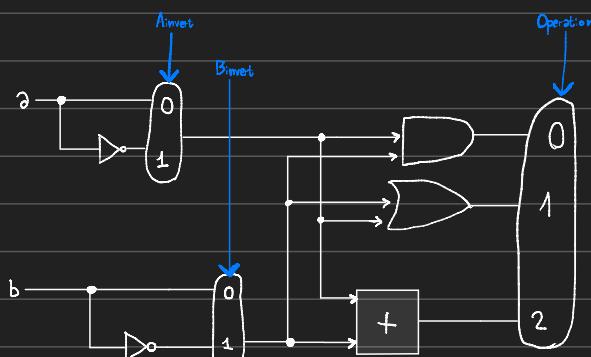
2° (ESSE)  $\neg b / (p \text{ OR } q) = \neg b \text{ OR } \neg p \text{ and } \neg q$

- ALU con NOR  $\rightarrow$  NOT ( $a+b$ )

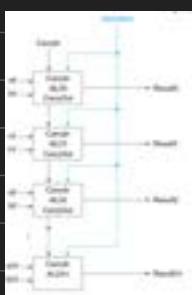
Usiamo De Morgan

$$\text{NOT } (a \text{ OR } b) = \text{NOT } a \text{ AND } \text{NOT } b \Rightarrow (\overline{a+b}) = \overline{a} \cdot \overline{b}$$

$$\text{NOT } (a \text{ AND } b) = \text{NOT } a \text{ OR } \text{NOT } b \Rightarrow (\overline{a \cdot b}) = \overline{a} + \overline{b}$$



- ALU 1-bit adiacenti (32 ALU IN SERIE)



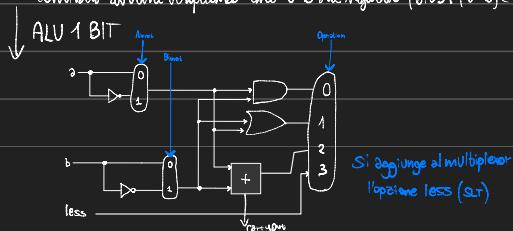
L'organizzazione in catena di carryIn e out si chiama Ripple Carry  
(2 bit, a e operation + bit carryIn) [INPUT]

### ALU ~ Operazioni di confronto

- SLT, set-on-less-than  $\rightarrow$  confronta due segnali  $a < b$  e genera 1 come risultato se  $a < b$

Il bit 0 dà il risultato (0 oppure 1), gli altri (bit 1-31) saranno ignorati.

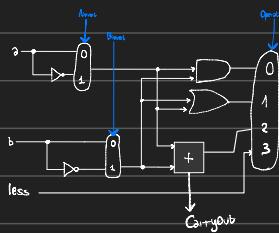
$\rightarrow$  Il controllo avrà invece verificato che  $a - b$  sia negativo ( $b[31] (a-b) = 1$ )



Si aggiunge al multiplexer l'operazione less (SLT)

## ALU A 32 BIT - SLT

→ ALU 0 - ALU 30



Della ALU 1 a 30 il risultato è 0, nell'ALU 0 il  
Risultato dipende dalla ALU 31 (dove c'è il segno, MSB)  
→ Se nell'ALU 31 carryIn=carryOut, less=0 (o non è minore di b)  
Bisogna prevedere l'over-flow tramite OVERFLOW DETECTION (posto in basso nell'ALU)

**BEQ: Branch on Equal**

Output: 1 se  $a=b$ , altrimenti 0

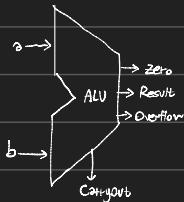
Come funziona: Se  $a-b$  è uguale a 0, l'equazione sostituisce 1 all'output

Come si realizza: OR di tutti i bit e nego il risultato



Collega in AND a tutte le  
ALU da 1 bit

## RAPPRESENTAZIONE ALU



**CIRCUITI SEQUENZIALI:** sono in grado di calcolare funzioni che dipendono anche da uno STATO → lo stato è memorizzato in memoria  
↳ i combinatori dipendono solo dall'input

### Tipi

- asincroni, non usano il clock (es: SR-latch)
- sincroni, usano il clock (es: flip-flop)

### COMPOSIZIONE

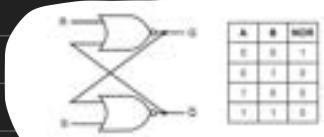
- elementi di memoria (di vario tipo): memorizzano informazioni
- reti combinatorie: elaborano informazioni

I dati memorizzati determinano lo STATO

Vengono memorizzati in piccole memorie da 1 BIT → il LATCH, un circuito di porte logiche che ha funzione  
di elemento di memoria (nello stabilisce lo stato)

## ESISTONO VARI TIPI DI LATCH

S-R Latch, SET e RESET



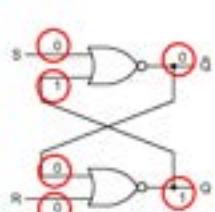
l'OR mette in confronto S/R con OLDQ

Due porte NOR concatenate:

due ingressi SET (S) e RESET (R),

le uscite sono Q e Q-bar (complementari)

RIPOSO (0,0)



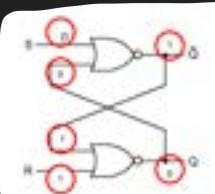
Input S R	Stato Interno Old Q	Output Q Q-bar
0 0	0	0 1
0 0	1	1 0
0 1	1/0	0 1
1 0	1/0	1 0
1 1	1/0	1 1

→ si mantiene il valore memorizzato in precedenza

→ è lo stato di RIPOSO

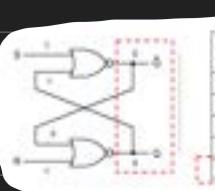
Q può assumere due valori, 0 e 1: bisogna prevedere entrambi i casi come old Q

RESET (0,1)



Input S R	Stato Interno Old Q	Output Q Q-bar
0 0	0	0 1
0 0	1	1 0
0 1	1/0	0 1
1 0	1/0	1 0
1 1	1/0	1 1

SET = 1, RESET = 1

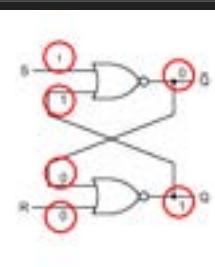


Input S R	Stato Interno Old Q	Output Q Q-bar
0 0	0	0 1
0 0	1	1 0
0 1	1/0	0 1
1 0	1/0	1 0
1 1	1/0	1 1

Viola la proprietà di Q (Q e Q-bar devono essere opposti)

↓  
CONFIG. INSTABILE, NON VALIDA (non viene usata)

SET



Input S R	Stato Interno Old Q	Output Q Q-bar
0 0	0	0 1
0 0	1	1 0
0 1	1/0	0 1
1 0	1/0	1 0
1 1	1/0	1 1

I segnali devono essere stabili e valere (0,1) o (1,0) per

memorizzare un valore corretto

→ Set e Reset sono calcolati di norma da un circuito combinatorio



La stabilità dell'output viene acquisita in un certo intervallo di tempo che dipende dal numero di porte e il ritardo (delay); inoltre bisogna evitare che vengano memorizzati valori instabili.

Usiamo il **CLOCK** come adunzione: rende il ciclo **SINCRONO**

Un segnale di gradi, il periodo è sufficientemente grande per assicurare la stabilità degli output: il clock abilita la scrittura nei latch periodica (determina ritmo, caboli e memorizzazione)



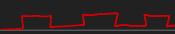
freq = Hz

$$f = 1/T$$

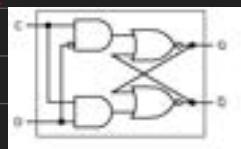
→ deve essere stabile (e quindi abbastanza ampio) un po' prima dell'apertura del latch (setup time) e chiusura (hold time, dura per un po' del nuovo stato)

### Latch con clock (D-Latch)

→ è sincronizzato con il clock, il cambio di stato è dettato temporaneamente



→ D=1 è il **SETTING**, corrisponde a quello che era S=1 e R=0 nel SR-latch



S=1 e R=1 non sarà mai possibile

→ D=0 è il **RESETTING**, corrisponde a S=0 e R=1

DE-ASSERTED

S=0 e R=0: il clock è DE-ASSERTED (mantenuto il valore precedente in memoria, memorizzazione sospesa)

deasserted

ASSERTED consente memorizzazione (in funzione del valore D, il segnale di clock è alto)

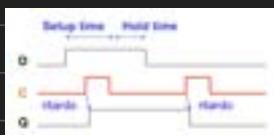
asserted

### SEGNALE D

\* DEVE essere già STABILE quando C entra in ASSERTED

\* RIMANERE STABILE per tutta la durata di C=1 (Setup time, segnale alto)

\* RIMANERE STABILE per un altro po' di tempo onto evitare malfunzionamenti (Hold time)



→ Stabilità per tutto setup e hold

→ nei circuiti reali il delay, anche se minimo, risulta ed è da prevedere

può causare glitch temporanei che si stabiliscono col tempo (per questo non devono memorizzare valori intermedi)

### COMPORTAMENTO DEL D-LATCH

- 1) Durante l'intervallo alto del clock il valore di D viene memorizzato nel latch
- 2) D si propaga immediatamente verso Q (non tutto il circuito)
- 3) Le parallelle variazioni D si diffondono e a più variazione termina l'intervallo alto del clock
- 4) Con il clock che torna a 0, Q si stabilizza

→ nell'intervallo basso del clock il latch non memorizza

} TRASPARENZA DEL LATCH

## METODI DI CONTROLLO

→ Level triggered: durata sul livello alto-basso del clock (quello visto finora) [alto → basso → alto → ecc]

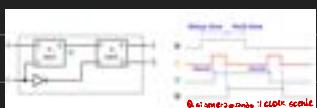
→ Edge triggered: avviene sul fronte di salita/cese del clock (FLIP FLOP)

\* la memorizzazione è istantanea

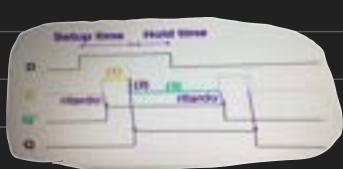
\* il segnale "spaccio" non arriva in tempo (tempo immidiativo)

D Flip-flop: si può usare come input e output durante l'ultimo ciclo di clock  
 ↓  
 sono 2 D-latch in serie, il primo è stato HISTER e il secondo slave

3 D dev'essere aperto per tutto il setup time + hold time di C

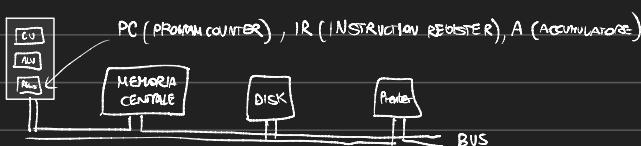


Step 1: il primo latch è aperto, Q' fluisce  
 ma il secondo latch è chiuso



Step 2: il clock scende, il secondo latch viene aperto  
 per memorizzare il valore di Q'

Step 3: il secondo latch è aperto, memorizza Q' e fa fluire il nuovo Q nello slave (minimizza il circuito col nuovo valore Q)  
 [il primo latch nel mentre è chiuso]

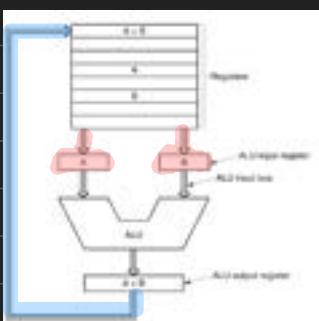


Datapath → parte operativa della CPU

Composto da:

\* ALU

\* REGISTRI (OPERANDI ISTRUZIONI LOGICHE MEM)



→ il clock non entra direttamente nei vari flip-flop, viene messo in AND con un segnale di controllo "WRITE"

→ "WRITE" determina se, nel falling edge (discese) del clock, il valore deve essere memorizzato nel registro  
 1 = OK  
 0 = STOP

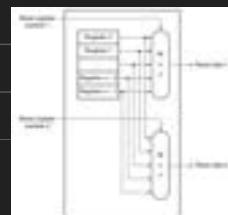
REGISTRI → costituiti da n flip-flop, sono organizzati  
 ↓  
 tramite Register File  
 può contenere valori di qualsiasi tipo rispettando i bit a disposizione



struttura ad Array che consente la lettura di 2 registri e la scrittura di 4 registri  
 \* input: dato che verrà memorizzato (in clock alto) e clock [write data: dati che deve arrivare; write register: indirizzo registro a cui arrivare]  
 \* output: valore attuale dei due registri (32 bit)

LETTURA REGISTER FILE

Per leggere un Register File si usano due Multiplexer (32 ingressi e un segnale di controllo a 5 bit)

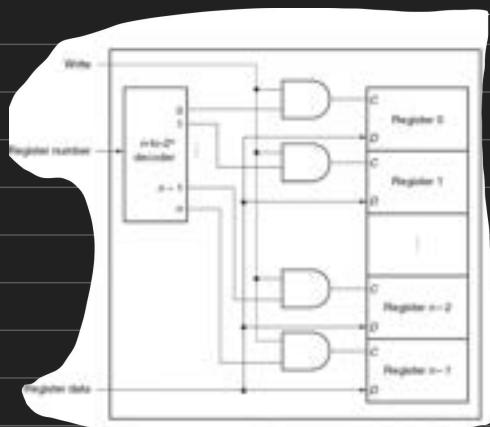


## SCRITTURA REGISTER FILE

- 3 SEGNALI

- 1) Registro su cui scrivere (register number)
- 2) Valore da scrivere (register data)
- 3) Segnale di controllo (write)

- IL REGISTER NUMBER viene decodificato da un DECODER
- il segnale WRITE (già in AND col clock) è in AND con l'output del decoder
- Se Write non è stato affermato nessun valore sarà scritto nel registro



## MEMORIA

Si distinguono per diversi criteri

- 1) Dimensione: quantità di dati memorizzabili
- 2) Velocità: intervallo di tempo tra la richiesta del dato e il momento in cui è disponibile
- 3) Consumo: potenza richiesta
- 4) Costo: costo per bit (REGISTER e CACHE i più costosi, SEGUONO MEMORIA CENTRALE e HDD)

Oltre una memoria con <sup>massimi</sup> i migliori criteri possibili è difficile, si cerca un compromesso

## RAM (MEMORIA PRINCIPALE)

Memoria volatile, più vicina dei registri e si usa per memorizzare dati strutturati o codice di programma (è connessa alla CM)

→ insieme di celle da 1 byte col proprio indirizzo

→ Operazioni: lettura e scrittura

→ indirizzamento: l'elaboratore seleziona una cella di memoria da usare tramite diversi criteri

→ registro indirizzo:  $n$  bit =  $2^n$  celle di memoria



Lettura (Read): dal registro indirizzi

viene localizzata la cella di memoria corrispondente, il contenuto va nel registro dati

## VARIAZIONI

• S-RAM (static RAM) è più veloce

→ realizzata tramite latch  $\longrightarrow$  Matrice  $H \times W$

→ tempo d'accesso tra 0,5 - 2,5 ns



→ sincronizzata con il clock

Scrittura (Write): il contenuto del Registro Dati è scritto nella cella di memoria individuata dal registro indirizzi

• D-RAM (Dynamic RAM): più capienti ma più lente

→ memoria tramite CONDENSATORI

→ tempi tra 50 - 70 ns

## MACCHINE A STATI FINITI

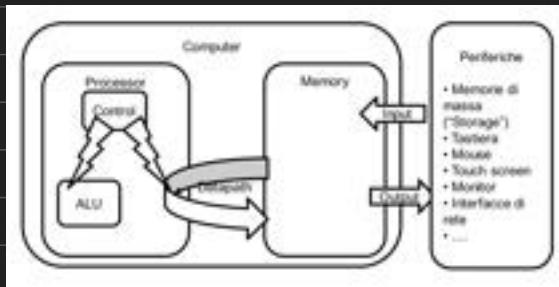
- FSM: descrivono i circuiti sequenziali
- Composte da un set di stati e due funzioni:
  - Next State Function: determina lo stato successivo partendo dello stato corrente e gli input
  - Output function: determina l'insieme di risultati partendo dello stato

→ sono sincronizzate dal clock

## ESISTONO DUE MACCHINE A STATI FINITI

- ① Moore: usa stato corrente, impiegato come controller per l'output
- ② Meadley: usa stato corrente e input

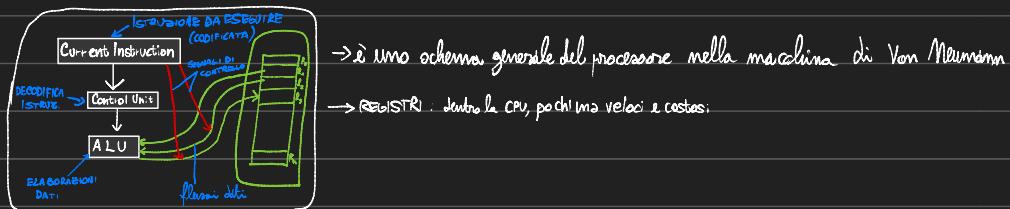
## Schema generale di un computer



La memoria centrale chiama a se input e output, comunica con le periferiche tramite bus e con il processore tramite DATA PATH

→ MEMORIA: grande, lenta (una al massimo cicli di clock) e mediamente costosa  
↳ insieme di celle, o "locazioni di memoria"

## PROCESSORE - SCHEMA



→ Lo CPU può essere realizzata in diversi modi:

### RISC (Reduced Instruction Set Computing)

→ poche semplici istruzioni

→ esecuzioni molto veloci

→ struttura circolarmente semplice

→ accorrono più istruzioni per fare cose semplici

### CISC (Complex Instruction Set Computing)

[es: Pentium, Intel x86...]

→ istr. complesse

→ Struttura e funzionalità complessa

→ Esecuzione più lenta per singola istruzione

## MIPS

→ è un tipo di architettura RISC intuitiva

→ 32 registri da 32 bit

→ Istruzioni da 32 bit

→ I dati sono manipolati solo su registri

→ I trasferimenti dati avvengono tra memoria e registri

→ Il flusso di controllo (esecuzione callta) è lineare, ma può essere alterato tramite salto (jump)

### ISTRUZIONI IN 32 BIT: come esprimere?

→ esistono diversi "formati" di strutture a 32 bit, che identificano tipi di istruzioni diverse

\* R-type → istruzioni che operano direttamente sui registri

\* I-type → istruzioni che sfruttano valori immediati o su un solo registro

\* J-type → istruzioni di salto

(R-type) → lavora sui tre registri: due input e un output

32 bit:

Op. operation code (tipo di operazione)	rs: primo registro sorgente (argomento)	rt: secondo registro sorgente (destinazione)	rd: terzo registro destinazione	shamt (shift)	funct variazione della funzione
000 000	01000	01001	01010	00000	10000

6 bit 5 bit 5 bits 5 bits 5 bits 6 bits

3 traduzione: Somma i contenuti del registro 8 e del registro 9 e metti il risultato nel registro 10

questo codice è detto "sorgente", che poi viene tradotto in ESEMPLIFICO e carica per l'esecuzione

Semplificato in assembly: add \$10, \$8, \$9

(I-type) → 1 sorgente, 1 valore imm. (costante) & 1 destinazione

Op. operation code	rs: primo registro (argomento)	rt: secondo registro (destinazione)	Valore imm. edato
001000	01000	01010	000000000000100

6 bits 5 bits 5 bits 16 bits

3 Sommo il valore 4 al registro 8 e salvo in 10

↓

add: \$10, \$8, 4

load → serve a leggere dati dalla memoria

Op. operation code	rs: registro base	rt: registro destinazione	offset (immediato)
100011	01000	01010	000000000000100

6 bits 5 bits 5 bits 16 bits

3 Carica nel registro 10 il contenuto della parola a

32 bit, che è all'indirizzo di memoria ottenuto dalla somma del contenuto del registro 8 e l'offset immediato

↓

Assembly: lw \$10, 4(\$8) (1 word)

Copia 32 bit a partire dall'offset in input (base+offset)

store

→ scrive dati sulla memoria

Op. operation code	rs: registro base	rt: registro destinazione	offset (immediato)
100011	01000	01010	000000000000100

6 bits 5 bits 5 bits 16 bits

3 Memorizza il contenuto del registro 10 (32 bit) all'indirizzo di memoria

ottenuto dall'offset (+) + del contenuto del registro 8

↓

Assembly: sw \$10, 4(\$8)

J-Type

→ J JMP, salta all'istruzione di indirizzo a 26 bit

OP	Jump Word
000010 6 bits	100000000000000000000011010

3 Salta all'istruzione di indirizzo  $0010e8_{16}$

verrà caricata l'istruzione in Program Counter, che cambierà la prossima istruzione da eseguire

I-type branch

→ salta al branch address se il contenuto di rs è diverso dal registro rt

op operazione	rs: primo registro	rt: secondo registro	RA: branch address
000101	10000	10001	??????

ASSEMBLY: bne \$t6,\$t7, Exit

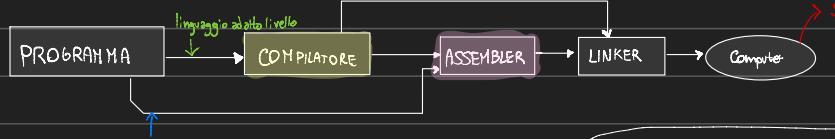
R-type: shift left

→ shift del contenuto e poi inserimento nella destinazione sinistro

Op: operazione code	rt: registro sorgente	rd: registro destinazione	Shift: quantità shift	func: variante dell'operazione
000 000	00000	1 0000	01010	00100 000 000

ASSEMBLY: sll \$t0, \$t6, 4 (shift left logical)

CONVERSIONE DA ALTO LIVELLO



→ LOADER: legge l'indirizzazione dell'eseguibile per determinare la lunghezza del segmento di testo (istruzione) e di quelli dati (variabili). Crea quindi lo spazio di indirizzamento sufficiente a contenere testo e dati.

→ OLTRECIÒ:

- \* copia istru. e dati del .exe alla memoria
- \* si occupa del salto all'inizio di struttura per l'esecuzione del programma (chiama il main())

\* una volta terminata la procedura, chiama la funzione exit (sys)

→ LINKER: riceve gli OBJECT (avendo la conversione in ASSEMBLY) e si occupa di creare il .exe (eseguibile)

COME LO FA?

\* inserisce in memoria il codice e i moduli

\* determina indirizzi dati ed etichette

\* risolve i riferimenti interni ed esterni e risolve i riferimenti in espresso

\* genera il file eseguibile

→ il COMPILATORE può essere di due tipi:

\* produce output ASSEMBLY, che va portato nell'ASSEMBLER

per la conversione in BINARIO

\* produce direttamente BINARIO, non serve l'ASSEMBLER

→ ASSEMBLER: converte il file sorgente ASSEMBLY in Linguaggio Macchina (.obj)

OLTRECIÒ:

- gestisce le etichette (indirizzi)

→ gestisce pseudo istruzioni

→ gestisce numeri in base diverse (binario, dec., esadec..)

**ASSEMBLATO** → processo sequenziale che esamina, riga per riga, il codice sorgente in Assembly, traducendo ciascuna riga in codice macchina

\* si applica modulo per moduli ("pezzi" del programma)

↓  
~ costruisce la tabella di simboli per ogni modulo

~ traduce ogni codice simbolico delle righe in binario

~ traduce le informazioni simboliche (var., reg., etichette di salto e parametri) in indirizzi numerici

\* L'ASSEMBLATORE LEGGE IL PROGRAMMA 2 VOLTE per intuire da soli tutte le etichette di salto nel codice

↓  
una lettura completa si chiama "passo"  
ASSEMBLER è un TRADUTTORE A DUE PASSI

\* l'insieme di default di un modulo dissemblato è:

**DEBUGGER** → strumento per aiutare il programmatore a trovare bug

↓  
\* Esegue il codice in modo controllato passo-passo (fornisce continuamente gli stabili di variabili e programma)

**DIRETTIVE DI ASSEMBLER** → istruzioni date all'assemblatore in memoria di associazioni e designazioni

↓  
opzione  
Lo standard è \$0000000

\* .data <addr> → inserire nel segmento dati (opzionale / inizializza)

\* .word b1, ..., bn → carica i valori in word (blocki di 4 byte in RAM (0, 4, 8, ...))

\* .byte b1, ..., bn → carica i valori in byte (blocki di 1 byte in RAM)

\* .text <addr> → inserire nel segmento text (programma)

\* .ascii (carica stringa ascii, ogni carattere è un byte)

\* .asciiz (aggiunge alla fine il carattere di fine stringa)

\* .Space (riserva una porzione di memoria (in byte))

\* .globl st (rende visibile all'esterno una porzione di memoria)

\* .extern str16 (rende visibile all'esterno, definisce una dimensione di 16 per la porzione di memoria)

**PSEUDO ISTRUZIONE**

↓  
istruzione assembly che non ha un corrispondente diretto in istruzioni macchina

↓  
è una sequenza di istruzioni native riassunta in una che l'assemblatore può capire

↓  
ma il registro \$1 (\$st)

→ registrare \$10 contiene codice syscall (va caricato) sono di terminali possono essere:

→ exit  
→ read\_int  
→ print\_int  
→ read\_str  
→ print\_str

**SYSCALL** → chiamata di funzione

f chiama g, viene eseguito e poi va finire da f

chiamata (call) → esegue procedura → entra in return → sono di nuovo al punto di sospensione (nella linea di ritorno)

**UN'ISTRUZIONE ha lunghezza 1 WORD (4 byte)**

Per sapere dove tornare, MIPS offre l'istruzione jump and link

↓  
Salta alla procedura e crea un collegamento per dove deve tornare per continuare l'esecuzione del chiamante

↓  
il return address, \$10, \$31 (qui è salvato il collegamento)

**Istruzione JR** → jump register

↓  
Salta all'indirizzo contenuto in un indirizzo

istruzione: jr \$ra è usata per tornare al chiamante dopo un jal

**Convenzioni assembly: nomi e usi dei registri**

→ per restituire i dati al chiamante  
→ passaggio di parametri alla chiamata funzione

→ I REGISTRI \$t sono temporanei e a libera disposizione della

procedura, il chiamante deve assicurarsi di salvare i loro valori prima della chiamata alla procedura

→ I REGISTRI \$s, invece, restano immutabili e pertanto se la procedura vuole farne

uso dovrà salvare il contenuto e ripristinarlo al ritorno

↓ Dove? nello stack

**OBJECT File** → output di un assembler

COMPOSTO DA:

~ object file header: dimensione e posizione delle altre parti del file

~ text segment: istruzioni macchina che compongono il programma

~ data segment: rappresentazione binaria dei dati nel file

~ relocation information: istruzioni che dipendono da riferimenti assoluti

~ symbol table: associa gli indirizzi delle etichette/label globali ed elenca le unresolved references

~ debugging info: descrizione simbolica

## PROCEDURE INNESTATE

→ FOGLIA: non chiama altre procedure

→ NON FOGLIA: lo chiama (è parte di una catena di chiamate)

↳ return address

Bisogna salvare il contenuto del registro \$ra e \$s

## STACK

Una area di memoria in cui i dati vengono inseriti da "pila" (↓) e verranno poi rimossi nell'ordine opposto (↑)

INIZIO FISSO, fine variabile

Due registri "puntatori":

→ \$sp, stack pointer avverte l'inizio (dal basso ↑) del percorso di memoria della procedura (punta alla prima parola del frame)

→ \$fp, frame pointer, fine percorso memoria della procedura (punta all'ultima parola del frame)

QUINDI, LA PROCEDURA:

\* ALLOCA SPAZIO NELLO STACK

↳ decrementa \$sp (lo fa scorrere ↓) per creare spazio di memoria (↓)

• salva \$ra

• salva anche altri elementi registri da \$sp come base

\* E UNA VOLTA FINITO...

• ripristina i registri

• incrementa \$sp fino al punto iniziale ("disallocazione")

• JR \$ra (ritorno)