

# APPUNTI DI ARCHITETTURE DEI CALCOLATORI beta

## PRIMO SEMESTRE - a.a. 2018-19

DOCENTE: BRUNO CICIANI - [ciciani@dis.uniroma1.it](mailto:ciciani@dis.uniroma1.it)

*Appunti (incompleti) di Christian*

---

<b>I codici</b>	<b>4</b>
Distanza di Hamming	5
Codice di parità	6
Codici Hamming	7
Iperecube	9
<b>Algebra di Boole</b>	<b>10</b>
Algebra di commutazione	11
Porta NOT	11
Porta AND	11
Porta OR	12
Funzioni di commutazione	12
Teorema di Shannon	12
Forma SP	14
Forma PS	14
PORTA NAND e PORTA NOR	15
Porta XOR	15
Inverter Three-state	16
<b>Reti combinatorie - Sintesi</b>	<b>17</b>
Mappe di Karnaugh (MK)	18
Funzioni parzialmente specificate	23
Prestazioni e costi	24
<b>Moduli di base</b>	<b>26</b>
Addizionatore (Half Adder, Full Adder)	26
Codificatore	30
Decodificatore	30
Decodificatore con enable	32
Realizzazione di funzioni con decoder	33
ROM	34
Temporizzazioni	35

---

---

Multiplexer	36
ALU	37
<b>Reti iterative</b>	<b>40</b>
Full Adder sequenziale	40
Rete combinatoria con circuito sequenziale	40
Contatore Up & Down	43
Sintesi circuito sequenziale - Mealy	44
Macchina sequenziale con ROM	47
Sintesi macchina sequenziale - Moore	47
Diagramma di flusso	50
<b>Flip-Flop e registri</b>	<b>51</b>
Flip-flop S-R	52
Segnale di sincronizzazione	54
Flip-flop S-R di tipo Latch	54
Flip-flop D	55
Registri	56
Registro parallelo-parallelo	56
Shifter Register	56
Registro circolare	57
<b>Sintesi reti LCC</b>	<b>58</b>
Classificazione variabili di ingresso	58
Reti LCC	58
Il modello strutturale di Mealy e Moore	59
Esempio contatore Up-Down	60
Esercizio	62
<b>Rappresentazione numeri interi e reali</b>	<b>66</b>
Conversione numero da binario a decimale	66
Conversione da decimale a binario	66
Complemento a 2	68
Rappresentazione numeri decimali (standard IEE 754)	69
<b>Flags: alcune considerazioni</b>	<b>72</b>
<b>Il processore z64</b>	<b>75</b>
Il modello di Von Neumann	75
Architettura Harvard	75
Architettura IBM	76
Generazione di un programma	77

---

---

Il processore z64	77
z64: l'unità di processamento (PU)	79
<b>Sistemi di reti LLC</b>	<b>80</b>
Macchina di Mealy e di Moore: un rapido ripasso	80
Catena sequenziale libera di macchine di Mealy	81
Catena sequenziale libera di macchine di Moore	83
Catena sequenziale di combinazioni di macchine di Moore e Mealy	85
Alcune considerazioni	87
Catena chiusa con macchine di Mealy	87
Catena chiusa con macchine di Mealy	87
Ciclo con combinazioni di Mealy e Moore	88
Stabilizzare ciclo macchine di Mealy	89
Pipeline	89
Architetture Pipeline parallele	90
Reti di sistemi per interconnessione qualunque	91
<b>Sistemi Digitali Connessi</b>	<b>93</b>
<b>Il sottosistema di calcolo (SCA)</b>	<b>95</b>
Interconnessione registri-circuiti di calcolo	95
Interconnessione registri-circuiti di calcolo tramite bus	97
Organizzazione vettoriale dei registri	98
Soluzioni intermedie	98
<b>Il sottosistema di controllo (SCO)</b>	<b>100</b>
Modello di Mealy per il SCO	100
Struttura del SCO con selezione	103
Circuito di mascheramento non codificato	103
Modello di Moore per il SCO	104
Esempio	106
Controllo per strutture Pipeline	109
Esempio di Pipeline	112
Sistemi con molti microprogrammi	115
<b>Comunicazione fra due sistemi digitali</b>	<b>118</b>
Sincronizzazione tra due unità in comunicazione mediante flip-flop di semaforo	121
<b>Il processore z64</b>	<b>123</b>
La macchina di Von Neumann	123
Suddivisione SCA-SCO	123
Tipologie di memoria: accenni	123

---

---

z64: struttura generale	124
z64: Sottosistema di Calcolo (SCA)	125
Interconnessione diretta tra registri	126
Interconnessione tramite multiplexer	126
Interconnessione tramite bus	127
Trasferimento dati tra registri e circuiti di calcolo (con multiplexer)	127
Trasferimento dati tra registri e circuiti di calcolo (con singolo BUS)	128
Esempio 1	128
Esempio 2	128
NOTE	129

## I codici

Un **codice** ( $C$ ) è un insieme di **simboli** e di regole fini alla generazione di parole che rappresentano gli elementi di un insieme di **entità** ( $C'$ ).

Con i codici è possibile procedere con la:

- **codifica** (data una parola di  $C$  viene associato un elemento di  $C'$ )
- **decodifica** (dato un elemento di  $C'$  si fa corrispondere una parola di  $C$ )

Un codice è detto **non ambiguo** quando tra ogni parola di  $C$  ed ogni elemento di  $C'$  c'è una corrispondenza univoca.

Un modo per rendere un linguaggio non ambiguo può essere quello di usare parole di lunghezza costante.

Se si indica con

**b** (detta **base**) il numero di simboli usati per identificare una parola

**n** la lunghezza costante della parola

**m** il numero minimo di caratteri per avere un'informazione non ambigua

---

**N** la cardinalità di C'

Allora  $b^m \geq N$

Se  $n = m$  allora il codice si dice irridondante, se  $n > m$  si dice ridondante, se  $n < m$  si dice ambiguo.

### Distanza di Hamming

La **distanza di Hamming**  $d(x,y)$  fra due parole è il **numero di posizioni** (bit) per cui esse differiscono. Tale distanza può essere utile per definire quando un codice è ambiguo, ridondante o irridondante.

Ex.  $d(10010, 01001) = 4$

La distanza minima di un codice è data dalla minima fra le distanze minime

$$d_{\min} = \min(d(x,y))$$

Se  $h = 1$  (e  $n = m$ ) si ha un codice irridondante

se  $h > 1$  (e  $n > m$ ) si ha un codice ridondante

Se  $h = 0$  si ha un codice ambiguo.

## Esempi di calcolo distanza di Hamming

Parole di C	Prima codifica	Seconda codifica	Terza codifica	Quarta codifica	Quinta codifica
<b>alfa</b>	000	0000	00	0000	110000
<b>beta</b>	001	0001	01	0011	100011
<b>gamma</b>	010	0010	11	0101	001101
<b>delta</b>	011	0011	10	0110	010110
<b>mu</b>	100	0100	00	1001	011011

$h = 1$	$h = 1$	$h = 0$	$h = 2$	$h = 3$
Irr.	Rid.	Amb.	Rid. <i>Rivela</i>	Rid. <i>Rivela</i>
			<i>errori</i>	<i>e corregge</i>
				<i>errori</i>

Esistono diversi tipi di codice. I codici carattere vengono usati per rappresentare in binario i simboli non numerici. Il codice più diffuso è il codice **ASCII**.

La rilevazione di errori di trasmissioni viene solitamente eseguita introducendo ridondanza nelle informazioni trasmesse. Su una trasmissione di un codice  $(n, k)$  ci saranno  $n$  bit trasmessi per  $k$  bit di informazioni, con  $n > k$ .

Il **peso di un errore** rappresenta il numero di bit che sono stati modificati durante la trasmissione.

*Un codice a distanza minima  $d$  è capace di rilevare errori di peso  $\leq d - 1$*

### Codice di parità

Il codice di parità è un codice in cui si inserisce un bit che vale 0 quando il numero di 1 è pari e che vale 1 quando è dispari.

Ex. **01010001 1**, poiché il numero di 1 dispari

---

Ex. 01111000 **0**, poiché il numero di 1 è pari

Il codice di parità può essere ottenuto dalle seguenti espressioni

$$b_1 + b_2 + b_3 + \dots + b_n + p = 0 \quad \text{parità oppure}$$

$$b_1 + b_2 + b_3 + \dots + b_n + p = 1 \quad \text{disparità}$$

Dove

- **n** è il numero di bit usati per rappresentare in binario gli oggetti (informazione),
- **+** e' l'operatore di somma modulo 2
- **p** il bit di "parità/disparità" da aggiungere a quelli di informazione per costruire parole del codice

Bit di informazione	Parità	Disparità
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

Il codice ottenuto è un codice di **distanza minima pari a 2**, cioè in grado di rilevare errori di pesi 1 (single error). Introducendo il bit di parità (o di disparità) è infatti facile vedere come la distanza minima, fra tutti gli elementi del codice, aumenti di 1 e diventi proprio pari a 2.

## Codici Hamming

Quello della generazione di codici Hamming è un modo per costruire codici a **distanza minima 3**.

Per ogni  $i$  è possibile costruire un codice a  $2^i - 1$  bit con i bit di parità e  $2^i - 1 - i$  bit di informazione. I bit in posizione corrispondente ad una potenza di 2 sono bit di parità, i rimanenti sono bit di informazione.

Ogni bit di parità controlla la correttezza dei bit di informazione la cui posizione, espressa in binario, ha un 1 nella potenza di 2 corrispondente al bit di parità.

---

Per capire di cosa si tratta immaginiamo di avere una informazione a 4 bit. Si avrà quindi un codice con 4 bit di informazione. Poiché i bit d'informazione erano pari a  $2^i - 1 - i = 4$ , si potrà dire che i, ossia il numero di bit di parità, sarà pari a 3.

Poiché i bit di parità hanno come posto la posizione corrispondente ad una potenza di 2, si avrà un bit di parità nelle posizioni 1, 2, 4.

**p<sub>1</sub> p<sub>2</sub> I<sub>3</sub> p<sub>4</sub> I<sub>5</sub> I<sub>6</sub> I<sub>7</sub>**

p<sub>1</sub> controlla la parità di I<sub>3</sub> I<sub>5</sub> I<sub>7</sub>

p<sub>2</sub> controlla la parità di I<sub>3</sub> I<sub>6</sub> I<sub>7</sub>

p<sub>4</sub> controlla la parità di I<sub>5</sub> I<sub>6</sub> I<sub>7</sub>

Si procederà quindi con il calcolo dei bit di parità scegliendo quello che verifica le seguenti equazioni

$$p_1 \oplus I_3 \oplus I_5 \oplus I_7 = 0$$

$$p_2 \oplus I_3 \oplus I_6 \oplus I_7 = 0$$

$$p_4 \oplus I_5 \oplus I_6 \oplus I_7 = 0$$

### Esempio

Se si vuole trasmettere l'informazione 1100, si avrà p<sub>1</sub> p<sub>2</sub> 1 p<sub>4</sub> 1 0 0

e, dai calcoli, p<sub>1</sub> = 0, p<sub>2</sub> = 1, p<sub>4</sub> = 1. Quindi l'informazione da inviare sarà **0111100**.

Qualora il codice ricevuto sia **1111100**, basterà verificare se le condizioni sono verificate per identificare l'eventuale presenza di errori. Infatti, in questo caso:

$$p_1 \oplus I_3 \oplus I_5 \oplus I_7 = 1$$

$$p_2 \oplus I_3 \oplus I_6 \oplus I_7 = 0$$

$$p_4 \oplus I_5 \oplus I_6 \oplus I_7 = 0$$

Che letto dall'ultimo al primo ottengo **001**, ossia avrò un errore in posizione **1**.

Nel caso in cui il codice ricevuto sia 0111110

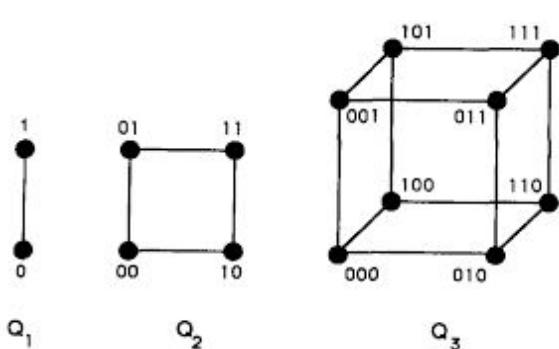
$$p_1 \oplus l_3 \oplus l_5 \oplus l_7 = 0$$

$$p_2 \oplus l_3 \oplus l_6 \oplus l_7 = 1$$

$$p_4 \oplus l_5 \oplus l_6 \oplus l_7 = 1$$

Avrà un errore in posizione **110**, ossia in posizione **6**.

## Ipercubo



L'iper cubo è una particolare struttura che permette di determinare la distanza di Hamming minima in un codice.

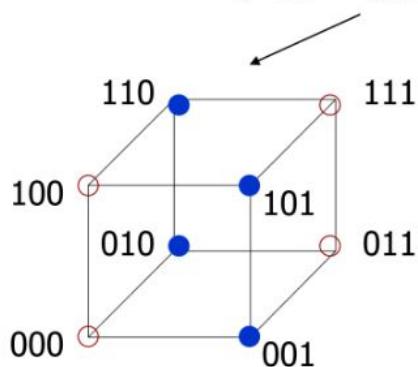
Qui affianco sono mostrati gli ipercubi di dimensione 1, 2 e 3.

Immaginiamo ora di avere il seguente codice

Sono colorati in blu i vertici corrispondenti ai termini inclusi nel codice.

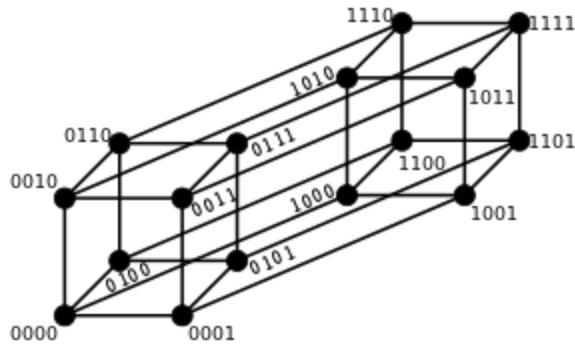
A =>	000
B =>	100
C =>	011
D =>	111

La distanza di Hamming potrà essere calcolata come il minimo fra i percorsi (e quindi di spigoli) necessari per passare da un vertice colorato ad un altro colorato.



In questo la  $d_{min} = 1$ , poiché la distanza di Hamming fra 001 e 101 è pari ad uno (basta infatti attraversare un solo spigolo per passare da 001 a 101).

Ovviamente ci possono essere ipercubi di dimensione superiore, quello affianco è un ipercubo di dimensione 4.



## Algebra di Boole

L'algebra di Boole ruota attorno ad alcuni postulati

Un insieme  $I$  e due operatori binari  $+$ ,  $\cdot$  formano un'algebra di Boole se soddisfano i seguenti assiomi ( $x, y, z$  sono elementi di  $I$ ):

- $\forall x, y \in I \quad x+y \in I; \quad x \cdot y \in I$  (chiusura delle operazioni)
- $\exists 0 \in I \mid \forall x \in I, \quad x+0=x$  (elemento neutro per  $+$ )
- $\exists 1 \in I \mid \forall x \in I, \quad x \cdot 1=x$  (elemento neutro per  $\cdot$ )
- $\forall x, y \in I \quad x+y=y+x; \quad x \cdot y = y \cdot x$  (proprietà commutativa)
- $\forall x, y, z \in I$   
 $x+(y+z)=(y+x)+z; \quad x \cdot (y \cdot z) = (y \cdot x) \cdot z$  (proprietà associativa)
- $\forall x, y, z \in I$   
 $x \cdot (y + z) = (x \cdot y) + (x \cdot z); \quad x+(y \cdot z) = (x+y) \cdot (x+z)$  (proprietà distributiva)
- $\forall x \in I \quad \exists \neg x \in I \mid x + \neg x = 1; \quad x \cdot \neg x = 0$  (esistenza dell'inverso)

Ciò che realmente importa, però, è la conoscenza di alcune importanti proprietà

**Idempotenza:**  $x + x = x, \quad x \cdot x = x$

Assorbimento (poco usato):  $x + x \cdot y = x, \quad x \cdot (x + y) = x$

$$x + (\neg x) \cdot y = x + y, \quad x \cdot ((\neg x) + y) = xy$$

---

**De Morgan:**  $\neg(x + y) = (\neg x)(\neg y)$

$$\neg(xy) = (\neg x) + (\neg y)$$

**Involuzione:**  $\neg(\neg x) = x$

## Algebra di commutazione

L'algebra di commutazione corrisponde ad un'applicazione dell'algebra di Boole ad un insieme con due soli valori: 0 e 1.

Gli operatori maggiormente diffusi sono tre: somma logica (OR), prodotto logico (AND) e negazione (NOT). NOT  $x_1$  (o  $\neg x_1$ ) può essere indicato con  $\overline{x_1}$ .

Ad essi corrispondono tre porte logiche.

### Porta NOT

Simbolo funzionale

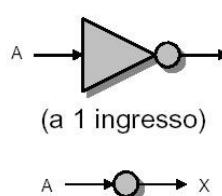


Tabella delle verità

A	X
0	1
1	0

### Porta AND

Simbolo funzionale

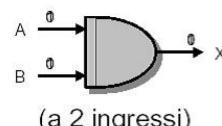


Tabella delle verità

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

### Porta OR

Simbolo funzionale

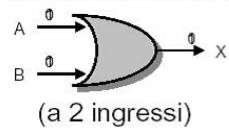


Tabella delle verità

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

## Funzioni di commutazione

Dati valori di ingresso nel dominio {0,1} una funzione di commutazione produce una variabile in uscita sempre nel dominio {0,1}.

Una funzione di commutazione può essere rappresentata mediante la sua tabella di verità.

La tabella qui a lato rappresenta la funzione che date 3 variabili restituisce 1 o 0, a seconda che il numero di 1 nelle tre variabili sia pari o dispari.

$x_3$	$x_2$	$x_1$	$y$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

## Teorema di Shannon

Il teorema di Shannon consente di trasformare le tabelle di verità in espressioni.

$$f(x_1, \dots, x_{i-1}, X_i, x_{i+1}, \dots, x_n) =$$

$$x_i f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n) + \neg x_i f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$$

$$1 \leq i \leq n$$

$$f(x_3, x_2, x_1) = x_3 f(1, x_2, x_1) + \bar{x}_3 f(0, x_2, x_1) =$$

$$= x_3 [x_2 f(1, 1, x_1) + \bar{x}_2 f(1, 0, x_1)] + \bar{x}_3 [x_2 f(1, 1, x_1) + \bar{x}_2 f(1, 0, x_1)] =$$

$$= x_3 x_2 f(1, 1, x_1) + x_3 \bar{x}_2 f(1, 0, x_1) + \bar{x}_3 x_2 f(0, 1, x_1) + \bar{x}_3 \bar{x}_2 f(0, 0, x_1) =$$

$$= x_3 x_2 x_1 f(1, 1, 1) + x_3 x_2 \bar{x}_1 f(1, 1, 0) + x_3 \bar{x}_2 x_1 f(1, 0, 1) + x_3 \bar{x}_2 \bar{x}_1 f(1, 0, 0) + \dots + \bar{x}_3 \bar{x}_2 \bar{x}_1 f(0, 0, 0)$$

In questo modo è possibile dimostrare che con il teorema di Shannon può essere descritta una tabella di verità in espressioni. Infatti  $f(x, x, x)$  rappresenta l'output data una certa combinazione di input. Poiché l'output possibile è rappresentato da 0 ed 1, i termini che rimarranno saranno solo quelli corrispondenti ad un output 1, ossia proprio i termini che descrivono la veridicità della tabella di verità.

Ad esempio, nella tabella seguente

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

0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

$f(0,0,0)$  varrà 1, quindi nell'espressione comparirà il termine  $\bar{x}_3 \bar{x}_2 \bar{x}_1$ , mentre poiché  $f(1,1,1)=0$  non comparirà il termine  $x_3 x_2 x_1$ .

Data questa tabella di verità  $f(x_1, x_2, x_3) = \bar{x}_3 \bar{x}_2 \bar{x}_1 + \bar{x}_3 x_2 \bar{x}_1 + x_3 \bar{x}_2 \bar{x}_1 + x_3 x_2 \bar{x}_1$

L'equazione prodotta è detta in **forma canonica Somma di Prodotti (SP)**.

## Forma SP

Un'espressione in forma SP contiene al più  $2^n$  termini, ove n è il numero delle variabili di input. Ogni termine è detto **mintermine** ed è il prodotto di n variabili dirette o negate.

In  $f(x_1, x_2, x_3) = \bar{x}_3 \bar{x}_2 \bar{x}_1 + \bar{x}_3 x_2 \bar{x}_1 + x_3 \bar{x}_2 \bar{x}_1 + x_3 x_2 \bar{x}_1$  si hanno 4 mintermini.

in una scrittura più formale:

$$\cdot f(x_1, \dots, x_n) = \sum_{k=0}^{2^{n-1}} m_k f(k) \Rightarrow f(x_1, \dots, x_n) = \sum_{k|f(k)=1} m_k \quad \textcircled{1}$$

dove:

$$\cdot m_k = \prod_{i=1}^n x_i^{\alpha_i} \quad (\alpha_i = 0 \text{ o } 1) \quad \text{mintermine} \quad \textcircled{2}$$

$\cdot f(k)$  il valore  $f(\alpha_1, \dots, \alpha_n)$ , con  $\alpha_1, \dots, \alpha_n$

$$\text{tali che } \sum_{k=0}^{2^{n-1}} \alpha_i 2^{i-1} = k$$

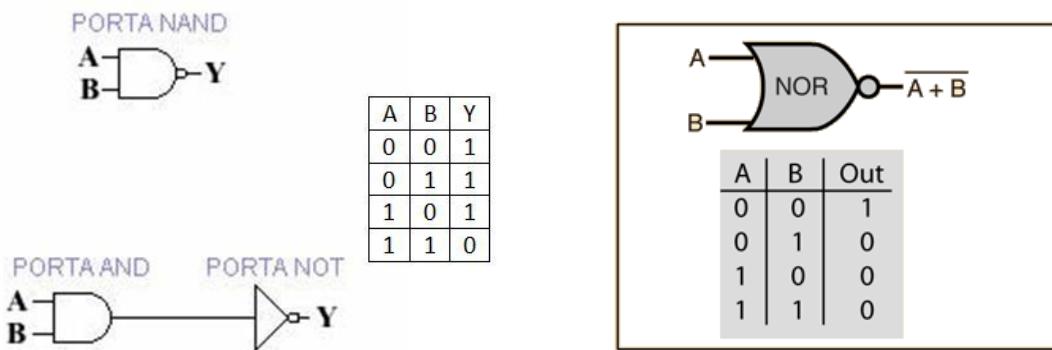
Ove (1) rappresenta la forma SP costituita da  $m_k$  mintermini e (2) rappresenta la forma di ogni mintermine.

## Forma PS

Esiste anche una forma canonica prodotto di somme (PS) che sfrutta logiche complementari. Non essendo nel programma qui viene saltata. È possibile trovare dei riferimenti alle diapositive 30-32 nel pacco slide 3 intitolato "AC-3-algebra di boole.ppt"

## PORTE NAND e PORTE NOR

NAND e NOR vengono chiamati **operatori universali** perché costituiscono un insieme funzionalmente completo di operatori logici. Infatti, data una funzione di commutazione, per poter lavorare abbiamo bisogno dell'OR, dell'AND e della negazione, oltre a 0 e 1. L'operatore NAND, così come il NOR, riesce a generare la somma logica, il prodotto logico, la negazione, lo 0 e l'1. In questo modo, in fase di sintesi dei circuiti, al posto di realizzare col silicio porte AND, OR o NOT posso pensare di realizzare un'unica componente che specializzo nel tempo. È dunque possibile realizzare componenti tutti uguali e la specializzazione è data dalle variazioni nelle connessioni.



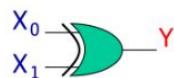
## Porta XOR

- or esclusivo, detto anche "somma modulo 2" o "anticoincidenza", indicato col simbolo  $\oplus$

$$x \oplus y = x\bar{y} + \bar{x}y = (x+y)(\bar{x}+\bar{y})$$

- $x \oplus y = y \oplus x$  (proprietà commutativa)
- $(x \oplus y) \oplus z = x \oplus (y \oplus z)$  (associativa)
- $x \oplus 1 = \neg x$
- $x \oplus 0 = x$
- $x \oplus x = 0$
- $x \oplus \neg x = 1$

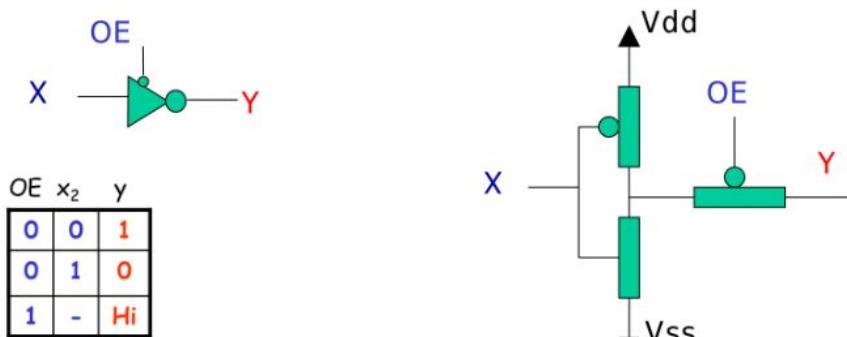
X <sub>1</sub>	X <sub>2</sub>	Y
0	0	0
0	1	1
1	0	1
1	1	0



Lo XOR è molto utilizzato per azzerare velocemente un registro oppure per vedere se il numero di bit di una configurazione sia pari o dispari. Non è un operatore universale.

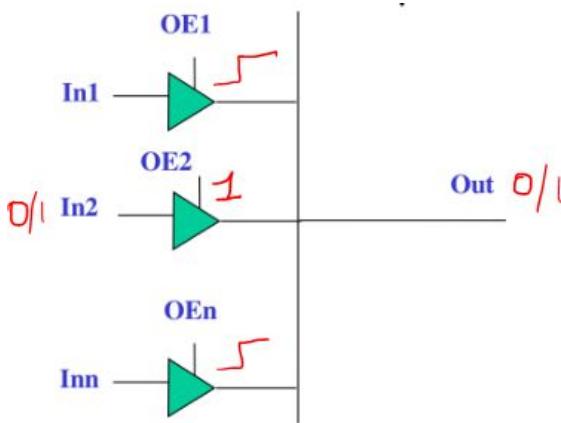
### Inverter Three-state

L'inverter Three-State **non** è una porta logica. La sua uscita può assumere 3 valori: 0, 1 ed alta impedenza elettrica, utile per sconnettere l'uscita dagli altri circuiti ad essa collegati.



OE è un segnale di abilitazione, che mi dice se lavora o non lavora. Presenta quindi in uscita un valore (0 o 1) quando lavora, presenta alta impedenza quando non lavora.

Questa implementazione è alla base del funzionamento di un bus. In un bus convergono diversi input di trasmissione, ma solo uno può essere trasmesso (per non provocare cortocircuiti), per questo si fa uso di un Inverter Three-state. A coordinare la serie di Inverter è un sistema di controllo.



Il secondo input è l'unico che potrà trasmettere, poiché tutti gli inverter sono in modalità alta impedenza.

## Reti combinatorie - Sintesi

Come già visto, una funzione può essere vista come sommatoria di prodotti di variabili, ossia come una sommatoria di mintermini.

In realtà una funzione può essere rappresentata con termini formati da un minor numero di variabili, detti **implicanti**. Un implicante è così chiamato perché quando esso è verificato lo è anche la funzione.

Un **implicante minimo** è un implicante per il quale non è possibile eliminare un letterale ed ottenere così un implicante "ridotto". Non possono banalmente essere fatte semplificazioni.

Un'**espressione minima** è un'espressione nella quale non possono essere eliminati né un letterale né un termine senza alterare la funzione rappresentata dall'espressione stessa.

$$\text{Sia } f(x_1, x_2, x_3) = \overline{x_3} \overline{x_2} \overline{x_1} + \overline{x_3} x_2 \overline{x_1} + x_3 \overline{x_2} \overline{x_1} + x_3 x_2 \overline{x_1}$$

Prendendo a due a due i termini possiamo vedere che sono possibili delle semplificazioni. Ricordando infatti che la somma fra una variabile e la sua negata fa 1:

$$\overline{x_3} \overline{x_2} \overline{x_1} \text{ e } \overline{x_3} x_2 \overline{x_1} \text{ diventano } \overline{x_3} \overline{x_1} (\overline{x_2} + x_2) = \overline{x_3} \overline{x_1} (1) = \overline{x_3} \overline{x_1}$$

$$x_3 \overline{x_2} \overline{x_1} \text{ e } x_3 x_2 \overline{x_1} \text{ diventano } x_3 \overline{x_1} (\overline{x_2} + x_2) = x_3 \overline{x_1} (1) = x_3 \overline{x_1}$$

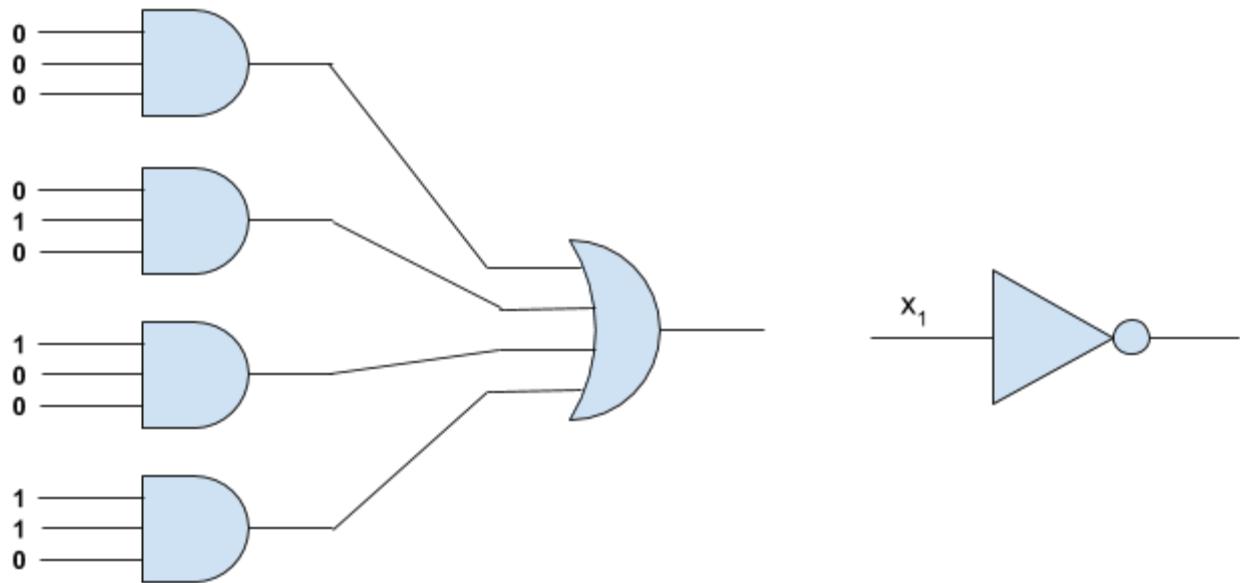
$$\text{Ma considerando proprio } \overline{x_3} \overline{x_1} \text{ e } x_3 \overline{x_1}, \text{ raccogliendo } \overline{x_1} \text{ otteniamo } \overline{x_1} (\overline{x_3} + x_3) = \overline{x_1}$$

Quindi siamo giunti ad un implicante minimo che non può essere ulteriormente semplificato. In questo caso si ha un solo termine, ma potevano venire tranquillamente anche più termini.

$$\text{Scrivere } f(x_1, x_2, x_3) = \overline{x_3} \cdot \overline{x_2} \cdot \overline{x_1} + \overline{x_3} \cdot x_2 \cdot \overline{x_1} + x_3 \cdot \overline{x_2} \cdot \overline{x_1} + x_3 \cdot x_2 \cdot \overline{x_1}$$

$$\circ f(x_1, x_2, x_3) = \overline{x_1}$$

è quindi completamente equivalente. Il vantaggio nell'aver ridotto al minimo l'espressione è però evidente. Nella forma iniziale servirebbe un circuito da due livelli (con 4 porte AND ed un OR), nella forma ridotta basta un solo NOT e quindi un solo livello.



Nell'attività di progetto è necessario tenere conto sia delle prestazioni che del costo. Le prestazioni sono soprattutto legati al numero dei livelli presenti, ciascuno ritarda il passaggio del segnale di un tempo pari circa a  $\Delta$ , quindi nel primo caso si ha un tempo di circa  $2\Delta$ , nel secondo di circa  $\Delta$ .

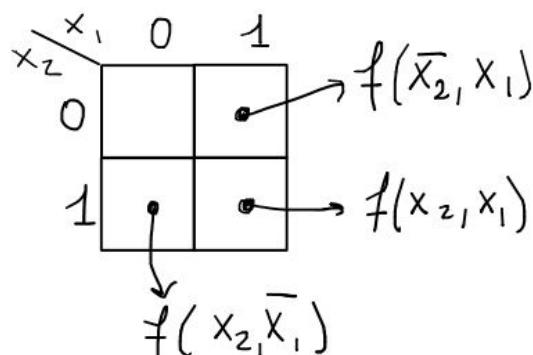
Il costo inoltre si ricava dalla somma del numero di letterali e degli implicanti. In questo caso il costo è  $3+3+3+3+4 = 16$

## Mappe di Karnaugh (MK)

Abbiamo visto come sia possibile passare da una tabella di verità di una funzione ad un'espressione e come questa possa essere ridotta ad implicant minimi.

Bisogna però ricordare che il numero di termini nell'espressione della funzione di commutazione cresce esponenzialmente al crescere delle variabili di input. Risulterebbe quindi scomodo rilevare le semplificazioni possibili manualmente. Si adottano per questo motivo le Mappe di Karnaugh, delle particolari tabelle in cui è possibile costruire "cubi" che identificano le possibili semplificazioni.

Su una mappa di Karnaugh vengono inseriti gli output ottenuti a partire dalle diverse combinazioni di input.



Se ad esempio al blocco in corrispondenza di 1-1 ci dovesse essere un 1, ciò comporterebbe che la funzione di commutazione conterebbe il termine  $x_2 x_1$

Immaginiamo la seguente tabella di verità e rappresentiamo la corrispondente mappa di Karnaugh.

$x_2$	$x_1$	$f$
0	0	1
0	1	1
1	0	0
1	1	0

In una mappa K un implicante primo corrisponde ad un raggruppamento di  $2^1$  celle adiacenti (dette cubi), sia orizzontalmente che verticalmente, non incluso in altri raggruppamenti.

In parole povere dovremo trovare raggruppamenti di 1 adiacenti a gruppi di 2,4,8,16... elementi.

$x_2$	0	1
0	1	1
1	0	0

Abbiamo riconosciuto una cubo di 2 elementi, quindi potremo scrivere che  $f(x_2, x_1) = \bar{x}_2$

Il motivo è che in ciascuna delle due celle  $x_1$  assumerà valori diversi, mentre  $x_2$  rimarrà costante a 0.

Se andassimo ad effettuare l'operazione di semplificazione manualmente otterremmo sempre

$$f(x_2, x_1) = \bar{x}_2 \bar{x}_1 + \bar{x}_2 x_1 = \bar{x}_2 \underbrace{(\bar{x}_1 + x_1)}_1 = \bar{x}_2$$

In questo secondo caso, sempre a due variabili non sono possibili raggruppamenti e quindi non si può procedere con semplificazioni.

$x_2$	$x_1$	$f$
0	0	1
0	1	0
1	0	0
1	1	1

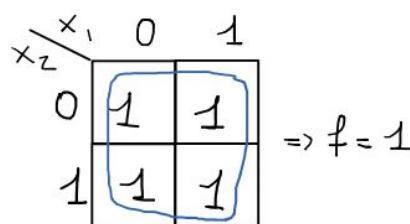
  

$x_2$	0	1
0	1	0
1	0	1

$$f(x_2, x_1) = \bar{x}_2 \bar{x}_1 + x_2 x_1$$

In questo terzo caso non è facile intuire che, essendo in tutti i casi sempre 1, la funzione varrà sempre 1. Inoltre poiché nei blocchi della mappa c'è un cambiamento di valore sia per la variabile  $x_2$  che per la  $x_1$  allora saprò che nessuna delle due comparirà nella funzione.

$x_2$	$x_1$	$f$
0	0	1
0	1	1
1	0	1
1	1	1

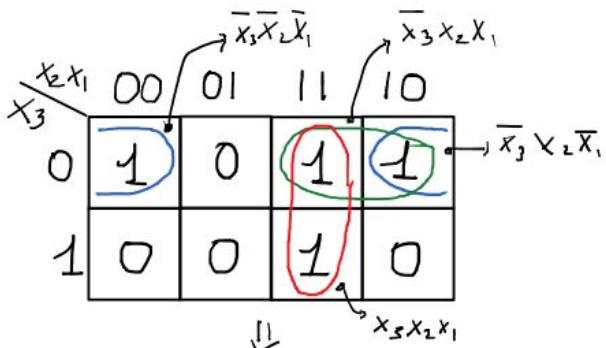


$$\Rightarrow f = 1$$

$$f(x_2, x_1) = \underbrace{\overline{x}_2 \overline{x}_1}_{\overline{x}_2 (\overline{x}_1 + x_1)} + \underbrace{\overline{x}_2 x_1}_{x_2 (\overline{x}_1 + x_1)} + \underbrace{x_2 \overline{x}_1}_{\overline{x}_2 + x_2} + x_2 x_1$$

Aumentiamo il numero di **variabili a 3**.

$x_3$	$x_2$	$x_1$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



$$f(x_3, x_2, x_1) = \overline{x}_3 \overline{x}_1 + x_2 x_1 + \overline{x}_3 x_2$$

$$f(x_3, x_2, x_1) = \underbrace{\overline{x}_2 \overline{x}_1 \overline{x}_3}_{\overline{x}_3 (\overline{x}_2 + x_2)} + \underbrace{\overline{x}_3 x_2 \overline{x}_1}_{x_2 (\overline{x}_3 + x_3)} + \underbrace{x_3 x_2 x_1}_{\overline{x}_1 (x_1 + \overline{x}_1)} + \underbrace{\overline{x}_3 x_2 x_1}_{\overline{x}_3 x_2 (\overline{x}_1 + x_1)} + \underbrace{\overline{x}_3 x_2 \overline{x}_1}_{\overline{x}_1 x_2 (\overline{x}_1 + x_1)}$$

Da notare che nella forma "estesa" i termini associati alle celle 011 e 010 vengono presi due volte perché rientrano ciascuno in due cubi. Inoltre uno dei cubi è formato dagli 1 delle celle 000 e 010, che seppur distaccate nella mappa concettualmente risultano essere adiacenti.

Procediamo con una mappa con **4 variabili**. Per motivi di spazio non rappresentiamo la tabella di verità.

$x_2 \backslash x_1$	00	01	11	10
$x_4 \backslash x_3$	00	1	0	1
01	1	0	0	0
11	1	1	0	1
10	1	0	0	1

$\Rightarrow f(x_4, x_3, x_2, x_1) = \bar{x}_2 \bar{x}_1 + x_4 x_3 \bar{x}_2 + x_4 \bar{x}_1 + \bar{x}_4 \bar{x}_3 x_2 x_1$

Perchè il raggruppamento verde produce un implicante minimo  $x_4 \bar{x}_1$  ?

A rimanere, come detto, sono i valori delle variabili che fra le varie celle non subiscono variazioni. Vediamolo nel dettaglio

$x_2 \backslash x_1$	00	01	11	10
$x_4 \backslash x_3$	1	1	0	1
11	1	0	0	1
10	1	0	0	1

$\Rightarrow x_4 \bar{x}_1$

I termini in viola non cambiano e sono quelli che compaiono poi nell'implicante. Da notare che  $x_4$  non è negato perché esso assume valore 1, mentre  $x_1$  è negato perché il suo valore è 0.

Ultimo caso, un pochino più difficile è quello con **5 variabili**. Prima di iniziare bisogna tenere in considerazione alcune cose:

- Si utilizzano due mappe, una con  $x_5$  fissato a 0, l'altra fissata ad 1
- Le due mappe sono raffigurate in maniera simmetrica, ma è come se fossero sovrapposte. Questo implica che ogni elemento di una mappa può, in linea teorica, essere accoppiato con il suo corrispondente nella seconda mappa.

		$x_5 = 0$				$x_5 = 1$						
		$x_4 x_3$	$x_2 x_1$	00	01	11	10	1	10	11	01	00
$x_4 x_3$	$x_2 x_1$	00	01	11	10	1	10	11	01	00	1	1
00	00	1	1	1				1	1	1	1	1
01	01			1					1	1		
11	11			1					1	1		
10	10	1		1					1	1		

$$\Rightarrow f(x_5, x_4, x_3, x_2, x_1) = \bar{x}_4 \bar{x}_3 x_1 + x_5 \bar{x}_2 + \bar{x}_5 x_2 x_1 + \bar{x}_3 \bar{x}_2 \bar{x}_1$$

Nelle diapositive 14-16-18 pacco slide 4a-reti combinatorie.

### Funzioni parzialmente specificate

Ci possono essere funzioni in cui non sono possibili alcune configurazioni delle variabili di ingresso o non interessa il valore di uscita per alcune configurazioni di ingresso.

Prendiamo ad esempio come caso una funzione che, per i numeri da 0 a 9 - espressi in BCD (Binary Code Decimal) con l'ausilio di 4 variabili -, sia verificata per i numeri pari. Poiché con 4 variabili si possono ottenere 16 combinazioni (e a noi interessano solo le prime 10), ci saranno 6 combinazioni che non ci interesserà studiare ed il cui risultato non è né 0 né 1, ma è identificato con d.c.c. (don't care condition).

$x_4$	$x_3$	$x_2$	$x_1$	$f_{\text{FARI}}$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	0	1	0
4	0	1	0	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	0	0
10	0	1	0	d.c.c.
11	0	1	1	d.c.c.
12	1	0	0	d.c.c.
13	1	0	1	d.c.c.
14	1	1	0	d.c.c.
15	1	1	1	d.c.c.

$x_4$	$x_3$	$x_2$	$x_1$	
00	00	00	10	
01	01	00	10	
11	11	—	—	
10	10	—	—	

In corrispondenza delle d.c.c. piazziamo dei trattini. Gli spazi lasciati vuoti dalle d.c.c. rappresentano per noi dei "jolly", possiamo cioè piazzare un uno al posto del trattino, lì dove ci può aiutare a fare semplificazioni, senza alterare il risultato della funzione.

$x_4$	$x_3$	$x_2$	$x_1$	
00	1	0	0	1
01	1	0	0	1
11	1	—	—	1
10	1	—	—	1

$\Rightarrow f(x_4, x_3, x_2, x_1) = \overline{x_1}$

Immaginiamo ora di voler considerare una funzione che, per i numeri compresi fra 0 e 9, sempre espressi in BCD con l'ausilio delle 4 variabili, produca in output 1 per i valori compresi fra 1 e 5. Ragioniamo allo stesso modo.

$x_4$	$x_3$	$x_2$	$x_1$	$f_{1 \leq x \leq 5}$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	0	1	1
4	0	1	0	1
5	0	1	1	1
6	0	1	0	0
7	0	1	1	0
8	1	0	0	0
9	1	0	1	0
10	0	1	0	d.c.c.
11	0	1	1	d.c.c.
12	1	0	0	d.c.c.
13	1	0	1	d.c.c.
14	1	1	0	d.c.c.
15	1	1	1	d.c.c.

$x_4$	$x_3$	$x_2$	$x_1$	
00	00	01	11	10
01	01	11	00	
11	—	—	—	
10	00	—	—	

$x_4$	$x_3$	$x_2$	$x_1$	
00	0	1	1	1
01	1	1	0	0
11	1	1	—	—
10	0	0	1	1

$$\Rightarrow f(x_4, x_3, x_2, x_1) = x_3 \bar{x}_2 + \bar{x}_3 x_2 + \bar{x}_4 \bar{x}_2 x_1$$

## Prestazioni e costi

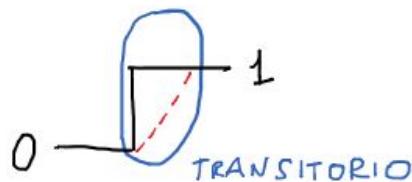
Come già detto, tenere in considerazione costi e prestazioni è fondamentale.

Nei circuiti combinatori il costo è proporzionale al numero di porte e, a parità di porte, dipenderà dal numero di livelli. A livello teorico è poi conveniente avere porte con meno ingressi possibili (seppur oggi, grazie agli enormi miglioramenti nelle fasi produttive, questo discorso può influire marginalmente).

Le prestazioni, invece, dipenderanno dal numero di livelli.

È interessante anche parlare delle **alee**. Un'alea è un fenomeno aleatorio spesso legato alle variazioni di commutazione delle variabili in ingresso. Ipotizziamo di avere un sensore di umidità che sia in grado di rappresentare il valore con un solo bit.

Il passaggio dal valore 0 all'1 non è immediato ma, essendo rappresentati da segnale elettrico, sarà presente una regione transitoria in cui non so esprimere qual è il valore dell'ingresso e quindi anche quello dell'uscita.



Per ovviare a questo problema ci sono specifici processi di temporizzazione delle variabili in ingresso, così che non vengano valutati i transitori.

La presenza di questo e di altri fenomeni (come i ritardi trasmissivi) determinano le prestazioni dei circuiti.

Sulla stazione spaziale internazionale la presenza di fenomeni aleatori è ancora più accentuata dal vento solare, è per questo che i circuiti degli apparati elettronici utilizzati implementano soluzioni tali da stabilizzare al massimo gli ingressi e le uscite.

# Moduli di base

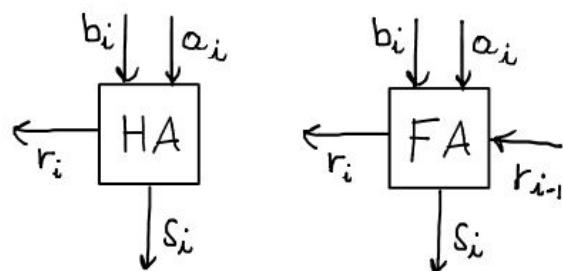
## Addizionatore (Half Adder, Full Adder)

Esistono addizionatori ad uno o più bit. Un addizionatore svolge, come dice il nome, l'operazione dell'addizione.

$$\begin{array}{r} \textcolor{red}{1} \\ 0 \ 1 \ 0 \ 1 \ + \\ 0 \ 1 \ 1 \ 0 = \\ \hline 1 \ 0 \ 1 \ 1 \end{array}$$

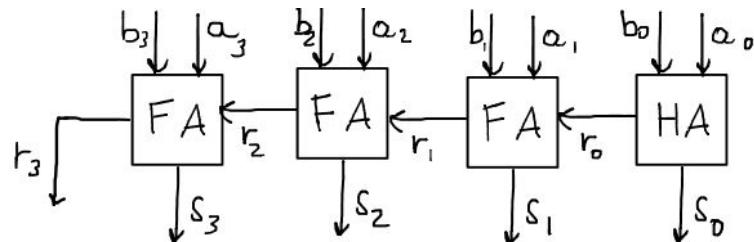
Per svolgere l'operazione somma bisognerà realizzare un circuito che tenga conto non solo della somma delle due variabili di corrispondente posizione, ma anche del possibile riporto.

Tale circuito può essere realizzato facendo uso di due componenti "elementari", l'half adder (HA) ed il full adder (FA).



L'HA prende due variabili in input e produce la somma ed il riporto, l'FA prende in input tre variabili (le due variabili da sommare ed un riporto).

Sfruttando quindi tali componenti potremmo realizzare il seguente circuito che, prese 4 variabili del primo addendo e 4 del secondo produca 4 valori di output ed un eventuale riporto.



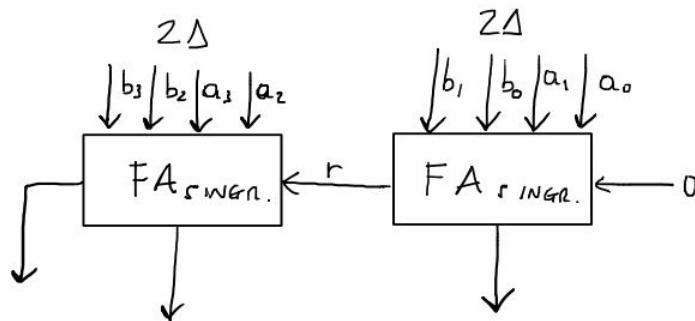
Nell'ipotesi in cui volessi utilizzare tutti Full Adder, sarà sufficiente mettere al full adder più a destra un ingresso costante pari a 0 (lui non potrà mai ricevere un riporto diverso da 0).

Il vantaggio di questa soluzione è che è **scalabile**, ma lo svantaggio è nel tempo impiegato.

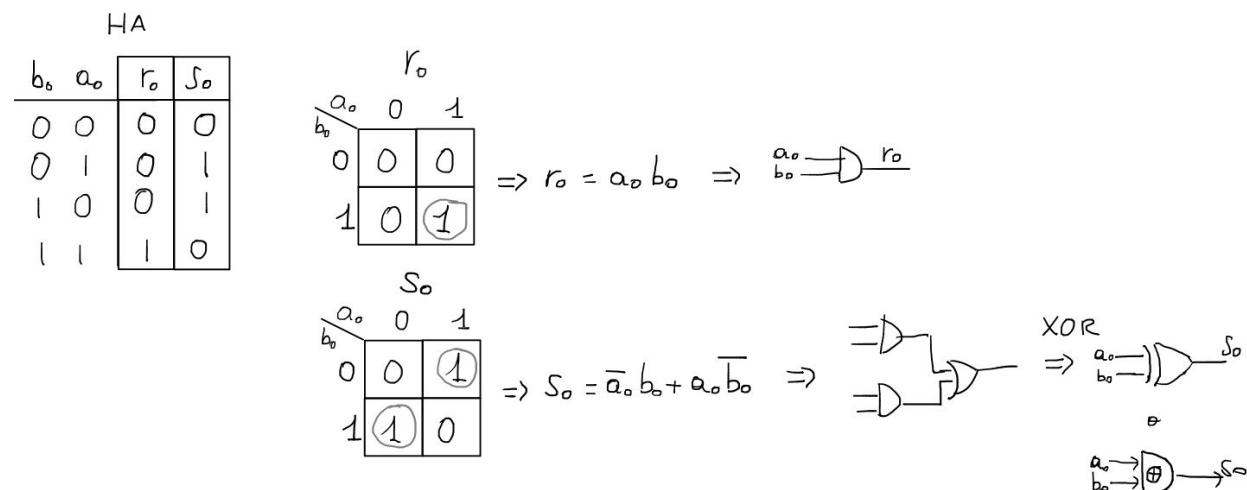
Se infatti chiamo  $\Delta$  il tempo massimo di commutazione della porta logica, ogni singolo elemento avrà due porte logiche e quindi impiegherà un tempo pari a  $2\Delta$ .

Svolgendo semplici somme è possibile vedere che  $S_3$  si stabilizza dopo  $8\Delta$ , perché il componente più a sinistra deve attendere che lavorino, in sequenza, tutti gli altri componenti. La soluzione **Bit Slice** ha quindi il vantaggio della scalabilità ma il tempo impiegato non è certo il minimo possibile. I sistemi di elaborazione a Bit Slice sono molto utili per il signal processing, perché in questo contesto serve definire la risoluzione con cui si vuole lavorare.

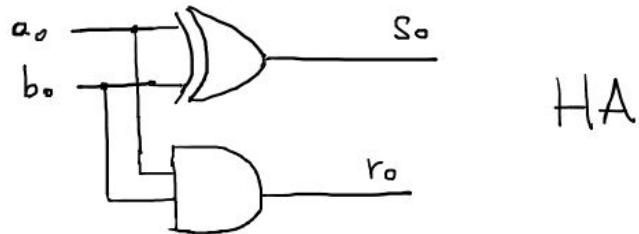
Per ridurre il tempo impiegato possiamo fare la sintesi, riducendo il numero di componenti e, ad esempio, introducendo Full Adder a 5 ingressi.



Procediamo con la sintesi dell'Half Adder.



Quindi un HA può essere rappresentato nel seguente modo:



Procediamo anche con la sintesi del Full Adder.

FA				
$b_1$	$a_1$	$r_o$	$r_1$	$s_1$
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

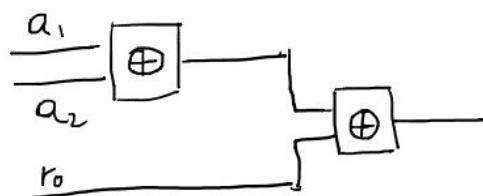
$b_1, a_1$	00	01	11	10
$r_o$	0	0	1	0
0	0	0	1	0
1	0	1	1	1

$$r_1 = b_1 a_1 + a_1 r_o + b_1 r_o$$
  

$b_1, a_1$	00	01	11	10
$r_o$	0	0	1	1
0	0	1	0	1
1	1	0	1	0

$$s_1 = \overline{b}_1 \overline{a}_1 r_o + \overline{b}_1 a_1 \overline{r}_o + b_1 a_1 r_o + b_1 \overline{a}_1 \overline{r}_o$$

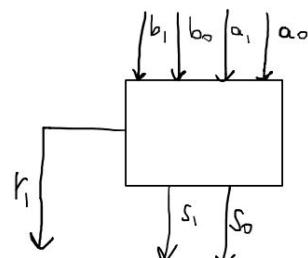
Guardando la tabella di verità si può vedere che  $r_1$  vale 1 quando il numero di 1 in  $b_1, a_1, r_o$  è pari.  $r_1$  può essere vista come una funzione di parità, mentre  $s_1$ , per simmetriche considerazioni, come una funzione di disparità. Entrambe queste funzioni potrebbero essere implementati con degli XOR. Quindi potremmo realizzare un FA con degli XOR in cascata.



È importante ricordare che questa soluzione è più che altro logica, in quanto non sostituisce quella prodotta con il processo di sintesi (in cui si fa uso di AND e di OR).

### Esercizio sintesi: Addizionatore a 5 bit

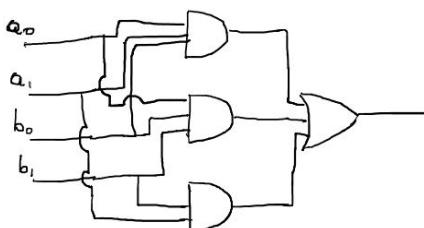
$b_1 \ b_0 \ a_1 \ a_0$	$r_1$	$s_1$	$s_0$
0 0 0 0	0	0	0
0 0 0 1	0	0	1
0 0 1 0	0	1	0
0 0 1 1	0	1	1
0 1 0 0	0	0	1
0 1 0 1	0	1	0
0 1 1 0	0	1	1
0 1 1 1	1	0	0
1 0 0 0	0	1	0
1 0 0 1	0	1	1
1 0 1 0	1	0	0
1 0 1 1	1	0	1
1 1 0 0	0	1	1
1 1 0 1	1	0	0
1 1 1 0	1	0	1
1 1 1 1	1	1	0



$r_1$	$a_1 \ a_0$	00	01	11	10
	$b_1 \ b_0$	00	0	0	0
00	00	0	0	1	0
01	01	0	0	1	0
11	10	0	1	1	1
10	11	0	0	1	1

$$r_1 = b_0 a_1 a_0 + b_1 b_0 a_0 + b_1 a_1$$

$r_1$



$s_1$	$a_1 \ a_0$	00	01	11	10
	$b_1 \ b_0$	00	0	0	0
00	00	0	0	0	0
01	01	0	1	0	1
11	10	1	0	1	0
10	11	1	1	0	0

$$s_1 = b_1 \bar{a}_1 \bar{a}_0 + b_1 \bar{b}_0 \bar{a}_1 + \bar{b}_1 a_1 \bar{a}_0 + \bar{b}_1 \bar{b}_0 a_1 + \bar{b}_1 b_0 \bar{a}_1 a_0 + b_1 b_0 a_1 a_0$$

# Moduli di base

Finora è stata affrontato il processo della sintesi e si è visto come sfruttarla per realizzare i circuiti di componenti come l'Half Adder ed il Full Adder. Vediamo ora alcuni moduli di base indispensabili.

## Codificatore

Un codificatore realizza la funzione di codifica binaria, ossia associare ad ogni elemento di un insieme  $\Gamma$  composto da  $m$  simboli, una sequenza distinta di  $n$  bit.

Un classico codificatore è quello che da cifre decimali fa arrivare a cifre in BCD (Binary Code Decimal). Voglio ad esempio fare in modo che, quando ad esempio viene premuto il pulsante di un tastierino numerico, il numero cliccato (da 0 a 9) sia rappresentato con 4 bit.

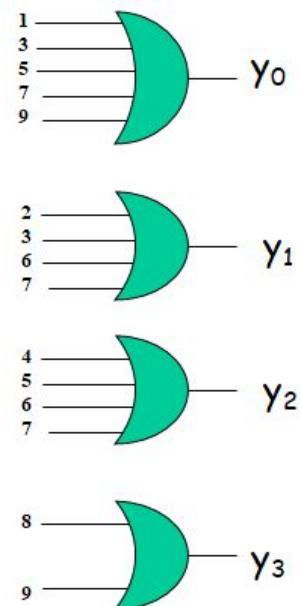
Possiamo rappresentare la tabella di verità connessa a questo caso.

	$y_3y_2y_1y_0$
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

È facilmente percepibile che il bit  $y_0$  vale 1 per tutti gli elementi dispari, ossia quando viene pigiato i tasti 1,3,5,7,9.

Il secondo bit  $y_1$  vale 1 quando vengono pigiati i tasti 2,3,6,7.

Mi basterà quindi usare 4 OR per ogni bit da produrre, ciascuno attivato quando uno dei tasti della specifica combinazione viene pigiato.



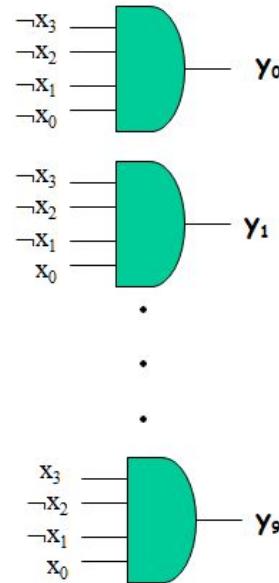
## Decodificatore

Il decodificatore realizza la funzione inversa del codificatore, a partire da una parola di un codice in binario genera un'uscita che identifica uno dei simboli dell'insieme  $\Gamma$ .

Un Decoder BCD-Cifre Decimale prende in input il numero espresso in Binary Code Decimal e produce, per le cifre comprese fra 0 e 9, 10 valori in output.

Una prima realizzazione che può esser fatta di questo decoder consiste nel far associare, ad ogni combinazione di input l'output che deve valere 1.

$X_3 X_2 X_1 X_0$	$Y_9 Y_8 Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0$
0000	0 0 0 0 0 0 0 0 0 1
0001	0 0 0 0 0 0 0 0 1 0
0010	0 0 0 0 0 0 0 1 0 0
0011	0 0 0 0 0 0 1 0 0 0
0100	0 0 0 0 0 1 0 0 0 0
0101	0 0 0 0 1 0 0 0 0 0
0110	0 0 0 1 0 0 0 0 0 0
0111	0 0 1 0 0 0 0 0 0 0
1000	0 1 0 0 0 0 0 0 0 0
1001	1 0 0 0 0 0 0 0 0 0

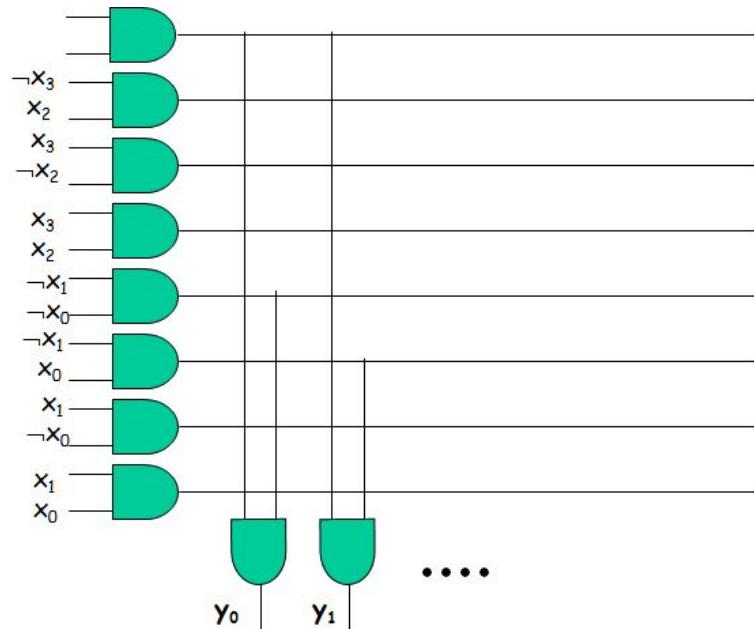


È possibile però produrre una soluzione **scalabile**. Il concetto di scalabilità è spesso associato a quello di linearità. In fase di progettazione si deve fare in modo di mantenere i costi realizzativi e progettuali il più possibile lineari. Il prezzo delle componenti aumenta in maniera non lineare, infatti porte con input 2 variabili costano 2, porte con 4 variabili in input costano 5, con 8 variabili 20, con 16 variabili 100 e così via.

Utilizzando esclusivamente componenti a due variabili di input si può riuscire a mantenere i costi il più contenuto possibile.

In questa seconda realizzazione vado ad utilizzare i componenti più volte. Ad esempio, se vado a vedere  $y_0$  in ingresso ha  $\neg x_0 \neg x_1 \neg x_2 \neg x_3$ , ma  $\neg x_2 \neg x_3$  vengono usati anche da  $y_1, y_2, y_3$ . Quindi utilizzo due livelli di AND tali da poter riutilizzare più volte la stessa combinazione di variabili di ingresso.

$x_3x_2x_1x_0$	$y_9y_8y_7y_6y_5y_4y_3y_2y_1y_0$
0000	0 0 0 0 0 0 0 0 0 1
0001	0 0 0 0 0 0 0 0 1 0
0010	0 0 0 0 0 0 0 1 0 0
0011	0 0 0 0 0 0 1 0 0 0
0100	0 0 0 0 0 1 0 0 0 0
0101	0 0 0 0 1 0 0 0 0 0
0110	0 0 0 1 0 0 0 0 0 0
0111	0 0 1 0 0 0 0 0 0 0
1000	0 1 0 0 0 0 0 0 0 0
1001	1 0 0 0 0 0 0 0 0 0



Il vantaggio di queste soluzioni, aldilà dei costi di realizzazione, è che, rispetto a componenti con tanti ingressi, ha consumi inferiori e genera meno calore. Fattori non trascurabili nella fase di progettazione, soprattutto quando non ci si può permettere particolari sistemi di raffreddamento.

I componenti, ora allineati in una colonna ed una riga, vengono solitamente distribuiti lungo tutta la matrice per migliorare la dissipazione del calore.

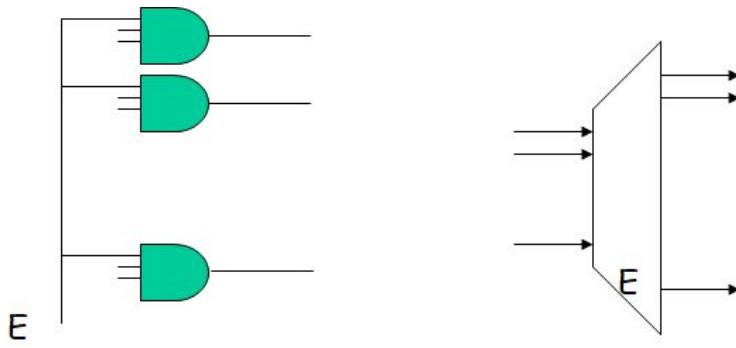
### Decodificatore con enable

In alcuni contesti si andranno ad usare dei decodificatori con **ingresso di abilitazione**, nel senso che do l'autorizzazione a lavorare. Il componente, se non lavora, resta in alta impedenza.

(Giusto per fare un esempio facilmente comprensibile: le prese della corrente, fintanto che non ci si mette la spina, stanno in alta impedenza).

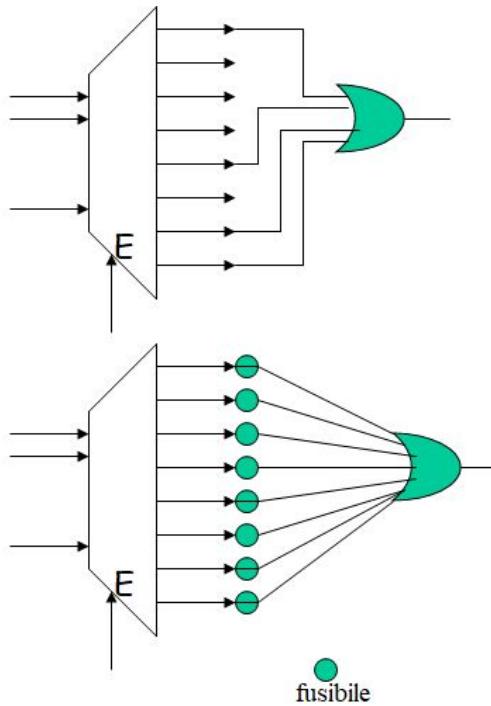
Per la legge di Ohm,  $V = R \cdot i$ . Se ho  $R$  che tende ad infinito,  $i$  tende a 0. Parlare di alta impedenza vuol dire che gli elettroni che passano sono pochi.

Se l'enable è pari a 0, il circuito non lavora e tutte le uscite sono in alta impedenza, mentre se è pari ad 1, quello che c'è in ingresso del decoder viene decodificato.



## Realizzazione di funzioni con decoder

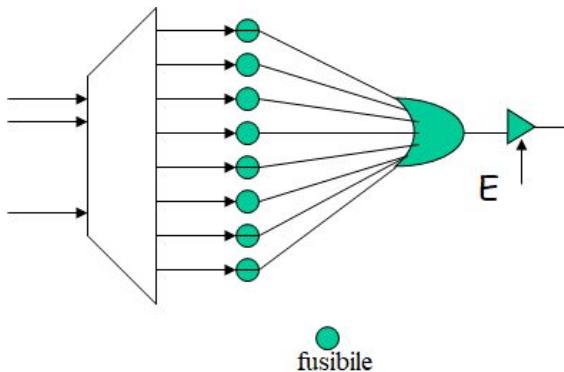
$x_2x_1x_0$	f
000	1
001	0
010	0
011	0
100	1
101	0
110	1
111	1



Ora, supponiamo di voler realizzare una funzione. Ho a disposizione un decodificatore a 3 ingressi e a 8 uscite. Ognuna delle 8 combinazioni dei tre ingressi corrisponderà ad un'uscita del decoder; tuttavia solo quelle che mi devono restituire un 1 sono quelle effettivamente connesse all'output. Questa implementazione necessita che siano effettuate connessioni "personalizzate"

con il saldatore. È evidente come questa soluzione non sia la più comoda.

La seconda soluzione fa uso di fusibili. Posso infatti realizzare circuiti standard ove però procedo a bruciare i fusibili che sono associati a quelle combinazioni di input che non devono risultare nell'output. Il tempo necessario per bruciare fusibili è sicuramente inferiore del dover effettuare delle saldature precise. Inoltre, i fusibili che spesso vengono utilizzati sono **rigenerabili**, cioè dei fusibili che trattati chimicamente possono essere rigenerati.



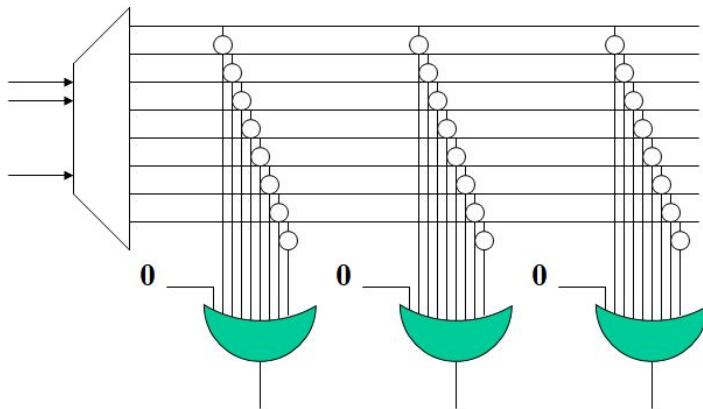
Qui affianco si ha una soluzione paragonabile alla precedente ove, oltre ai fusibili è presente un Enabler Three-state.

Sarà l'enable a determinare se tutto ciò che è stato calcolato a monte passa o non passa.

## ROM

Una **ROM (Read Only Memory)** è un **circuito combinatorio** perché dato un ingresso l'uscita è sempre la stessa.

Lo schema logico di una ROM si basa su un decodificatore, alle cui uscite sono "agganciate" più porte OR. Una ROM ad 8 bit presenterà 8 porte OR.



- **fusibile**

Se non brucio nessun fusibile, qualunque ingresso nel decoder produrrà un 1 in uscita, se li brucio tutti otterrò sempre 0.

Le ROM possono essere :

- già "pronte all'uso" con fusibili bruciati dalla casa produttrice (ad esempio per uno specifico BIOS).

- 
- programmabili (con i fusibili ancora non bruciati) - in questo caso si parla di PROM (Programmable Read Only Memory)
  - EPROM (Erasable Programmable Read Only Memory), nel caso in cui i fusibili possano essere rigenerati (per un certo numero finito di volte). Le memorie allo stato solido sono delle EEPROM e possono essere riprogrammate un numero di volte di ordine di grandezza  $10^5$ .

Poiché un accesso in memoria viene eseguito con un ordine di grandezza di  $10^{-9}$ , con un normale programma si arriverà a compiere  $10^9$  accessi in memoria. Supponiamo di realizzare un ciclo eseguito per 3 milioni di volte in cui il programma modifica sempre lo stesso indirizzo, il mio dispositivo risulta inutilizzabile dopo meno di un millisecondo.

Il trucco che viene adottato è di non utilizzare la stessa riga più di n volte, ma di bilanciare l'utilizzazione di tutte le righe.

Il problema di questa implementazione è che il numero di input di ciascuna porta OR aumenta esponenzialmente all'aumentare del numero di ingressi del decoder. Con 10 ingressi del decoder, la porta OR dovrebbe avere  $2^{10}$  ingressi. Ci si rende facilmente conto di come ciò sia nella pratica non implementabile.

INSERISCI SCHEMA ONENOTE SU SURFACE

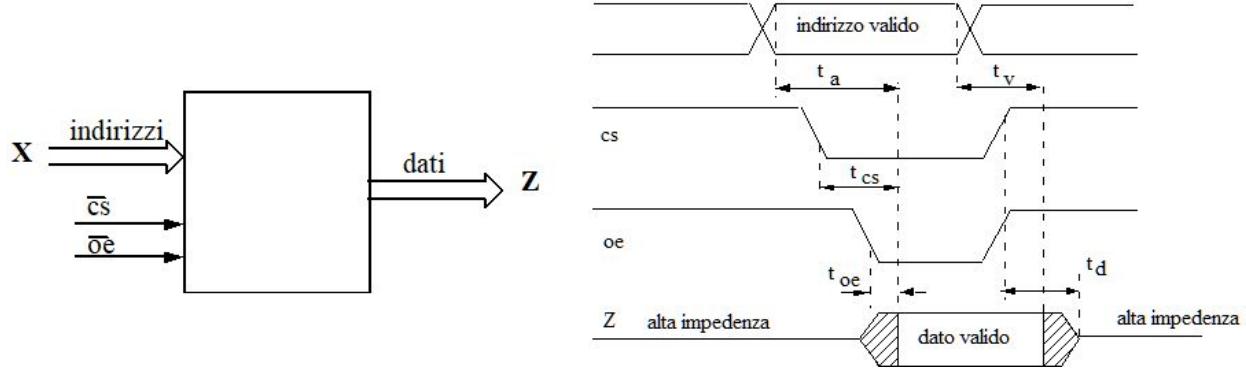
## Temporizzazioni

Dal momento in cui cambio l'ingresso a quello in cui l'uscita è utilizzabile passa del tempo. Questo dipende sia dai ritardi di propagazione, sia da quelli "computazionali".

Tempo di accesso basico è il tempo di propagazione dopo il quale l'uscita potrebbe essere valida.

Chip Select - Output enable

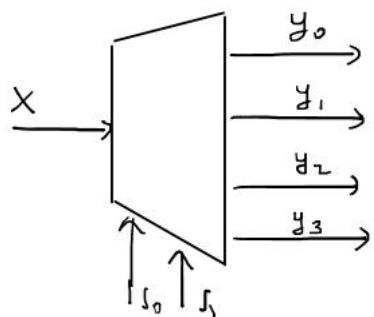
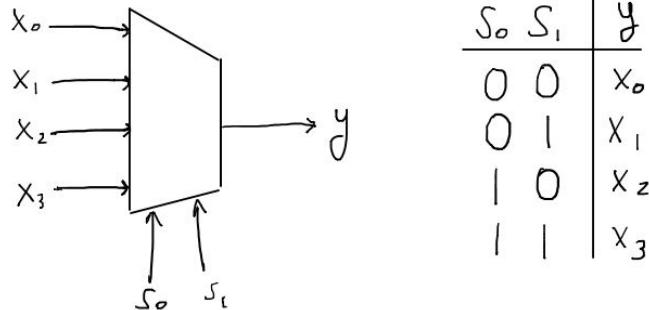
//Minuto 49 circa DISEGNO CORPO



## Multiplexer

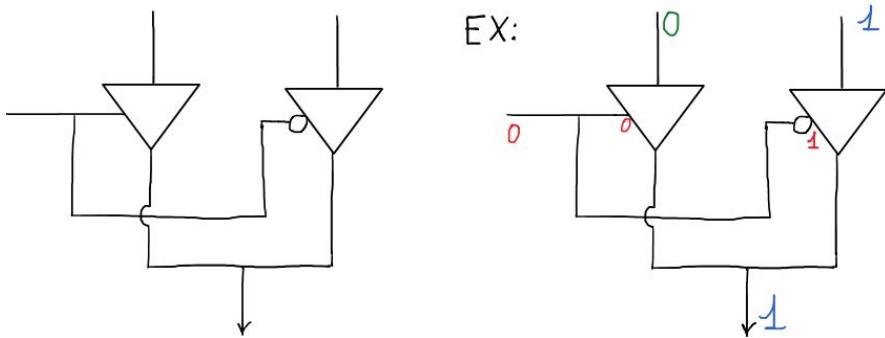
I multiplexer sono dei commutatori che fanno uso di  $m=2^n$  ingressi dati con  $n$  ingressi di controllo. Con i segnali di controllo decido come mettere in comunicazione un ingresso con la relativa uscita.

Se ipotizziamo di avere 4 ingressi dati e 2 ingressi di controllo l'uscita del multiplexer dipenderà dalla combinazione degli ingressi di controllo.



Il Demultiplexer svolgerà una funzione opposta, accetterà cioè un ingresso e "deciderà" quale delle uscite deve assumere il valore dell'ingresso.

Un Multiplexer può essere realizzato in un modo molto semplice ed immediato con l'ausilio di due **Buffer Three-State**.



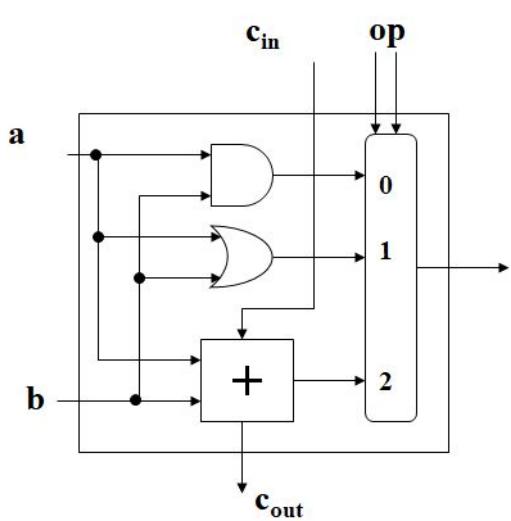
Si utilizza un segnale di abilitazione che giunge negato ad uno dei due buffer. Così facendo solo un buffer alla volta riceverà un segnale di abilitazione pari ad 1. Il buffer che riceverà segnale di abilitazione pari a 0 si metterà in uno stato di alta impedenza e solo quello che è stato abilitato rilascerà in output il valore che gli giunge in input.

Nell'esempio di sopra il segnale di abilitazione è 0, quindi solo il buffer three-state di destra sarà abilitato a trasmettere in uscita il suo ingresso (1).

Un multiplexer con questa implementazione costituisce la versione più elementare di **BUS**, poiché così facendo solo uno dei due segnali di input potrà viaggiare sul mezzo condiviso.

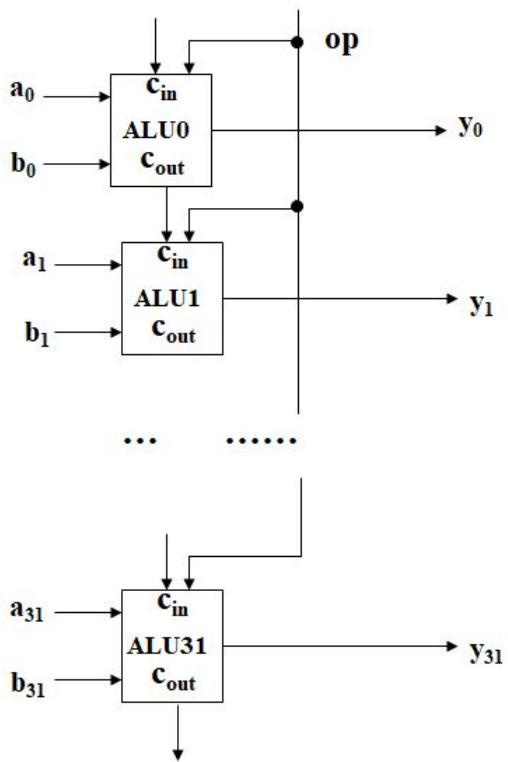
Il complemento ad una base è molto utile per le sottrazioni. Il complemento di un numero [1:10:00].

## ALU



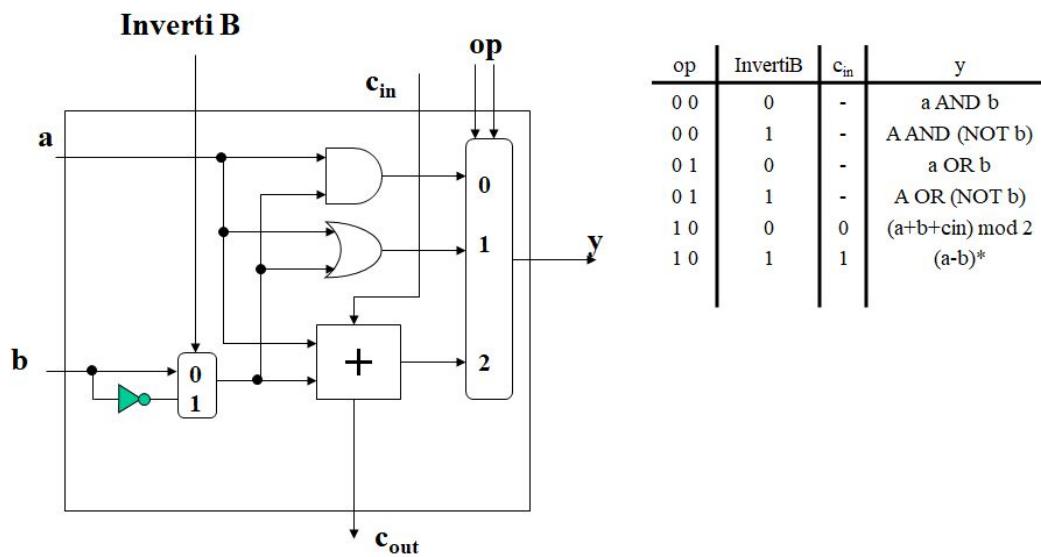
Questa mostrata è una ALU (Arithmetic logic unit) capace di poter fare semplici operazioni quali l'AND, l'OR e la somma.

La cosa interessante è che quando io inserisco dei valori d'ingresso a e b, tutti e tre i circuiti presenti lavorano, ma prendo una sola uscita utilizzando il multiplexer (posto sulla destra).



Posso andare a complicare la situazione andando ad aumentare il numero di ingressi e a replicare la cella dell'ALU ad un bit. Con 32 celle ALU si otterrà, come è logico che sia, una ALU a 32 bit.

Questa soluzione è scalabile, ma presenta un problema: l'ultimo bit, che è significativo se devo fare la somma o la sottrazione, viene ottenuto dopo l'elaborazione di tutte le celle ALU precedenti, comportando tempi potenzialmente alti di esecuzione. Se mi accontento di questi tempi, questa soluzione resta comunque efficace.



In questo altro caso si ha una ALU in grado di poter svolgere anche la sottrazione, in particolare Inverti B varrà 1 se bisogna procedere con la sottrazione, 0 nel caso della somma.

---

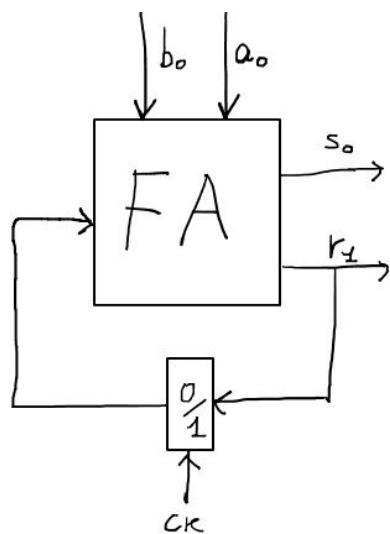
La condizione  $a=b$  è utile per eseguire istruzioni in modo condizionato (eseguire cioè dei jump).

# Reti iterative

## Full Adder sequenziale

Finora si è visto come strutturare moduli di base per realizzare circuiti combinatori, ove si mettevano in sequenza tali moduli per eseguire le operazioni richieste. Potremmo però pensare di realizzare **circuiti sequenziali**, che computano determinati ingressi e che riutilizzano, iterativamente, le uscite come nuovi ingressi.

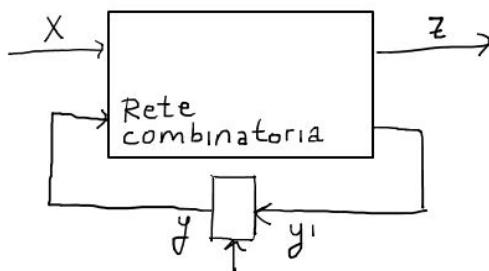
Un caso immediato è quello del Full Adder sequenziale. Potremmo infatti realizzare un solo modulo che, con l'aiuto di un elemento di memorizzazione del riporto, mi permetta di svolgere l'intera operazione.



Prima di effettuare una nuova operazione si attende che il sistema si stabilizzi, viene quindi "catturata un'istantanea" dell'uscita che, attraverso dei **flip flop**, viene memorizzata e riusata come input del riporto alla successiva computazione.

## Rete combinatoria con circuito sequenziale

Procediamo con la progettazione di una rete combinatorio con un'uscita, degli ingressi e degli stati.



---

Per fare tutto questo partiamo, come prima cosa, dalla definizione di macchina sequenziale.

Una macchina sequenziale è una quintupla  $M_s = \{ I, O, S, \delta, \omega \}$ , ove:

- $I$  è l'alfabeto di Ingresso  $I=\{i_1, \dots, i_m\}$
- $O$  è l'alfabeto d'Uscita  $O=\{o_1, \dots, o_q\}$
- $S$  è l'insieme degli Stati  $S=\{s_1, \dots, s_n\}$
- $\delta$  è la funzione dello stato successivo  $\delta: S \times I \rightarrow S$
- $\omega$  è la funzione di uscita
  - $\omega: S \times I \rightarrow O$  (Mealy)
  - $\omega: S \rightarrow O$  (Moore)

Per rappresentare le funzioni  $\delta$  ed  $\omega$  si possono usare:

- diagramma degli stati
- Tabella degli stati/uscite (di transizione)
- Algorithm State Machine (ASM)
- Matrice di connessione

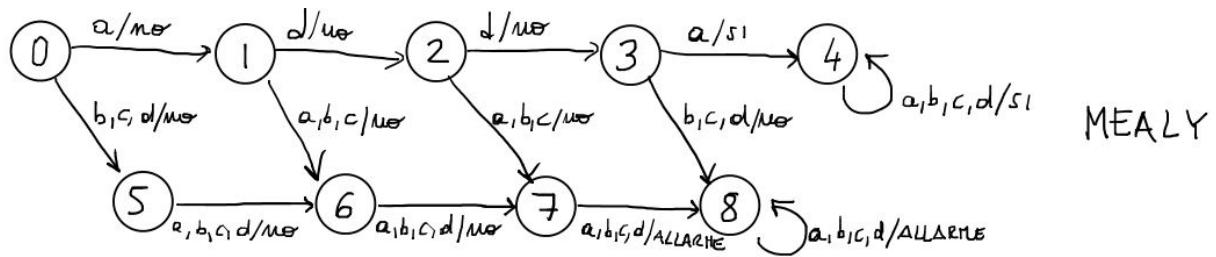
### Esempio

Cerchiamo di rappresentare la macchina sequenziale caratterizzata dagli insiemi  $I = \{a, b, c, d\}$  e  $O = \{\text{si, no, allarme}\}$

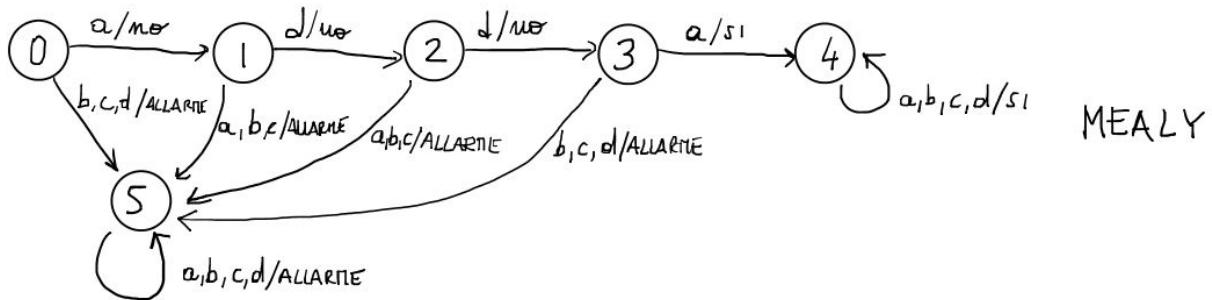
Tale macchina deve verificare se che i primi 4 caratteri di una stringa compongano la sequenza "adda". Se tale condizione è verificata verrà prodotto come output "si", anche quando dovessero essere premuti altri caratteri; in caso contrario, verrebbe prodotto in output "allarme".

Si noti inoltre che in questa formulazione dell'esercizio è richiesto che l'allarme scatti dopo aver inserito almeno 4 caratteri e non al primo carattere sbagliato.

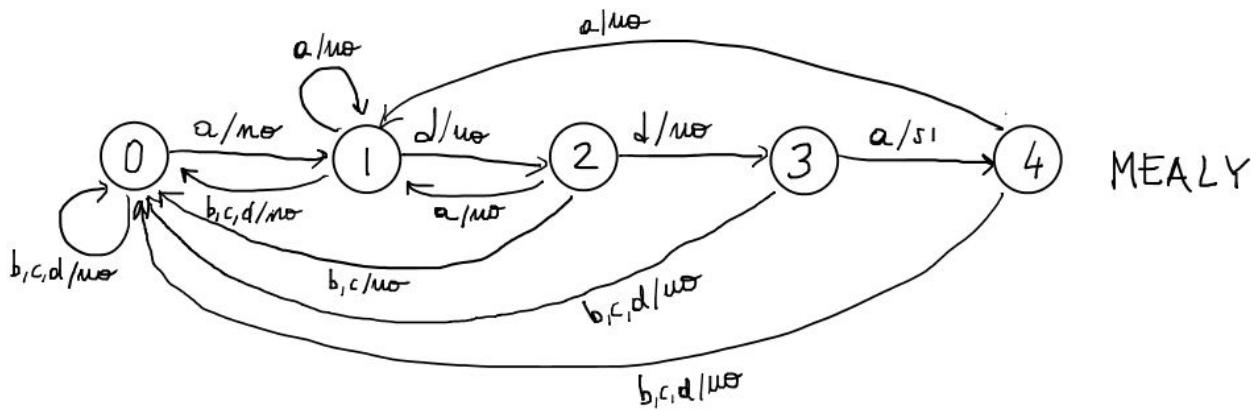
In questo caso partiamo con la rappresentazione di Mealy tramite diagramma degli stati.



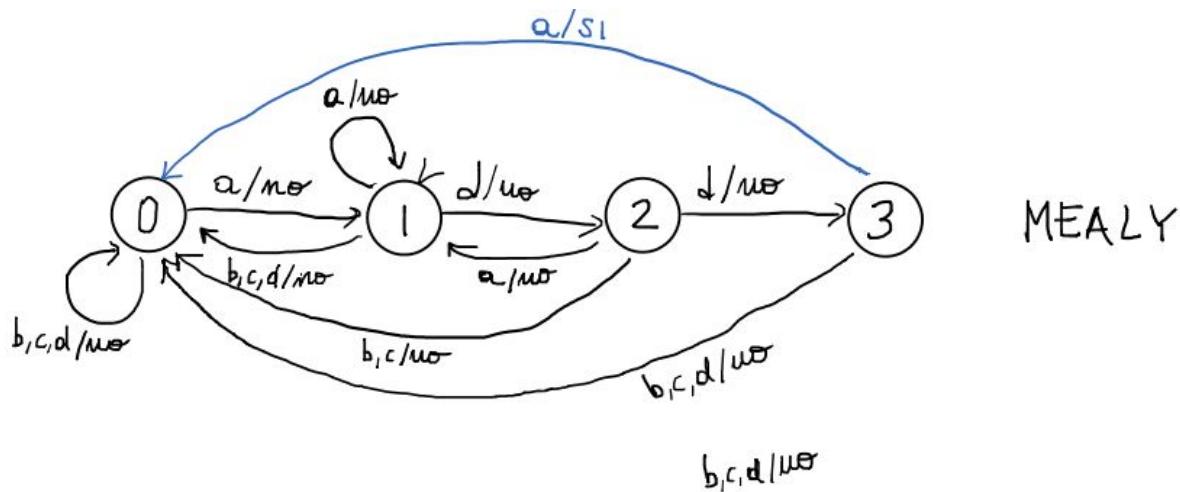
Qualora volessi far suonare l'allarme al primo carattere sbagliato, il diagramma diverebbe



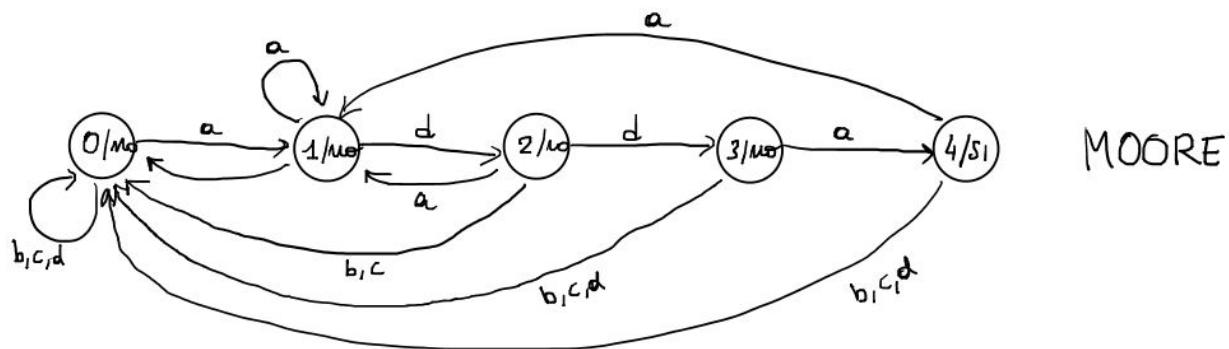
Immaginiamo di voler accettare sequenze di "adda", come "addaaddaadda", il diagramma sarebbe



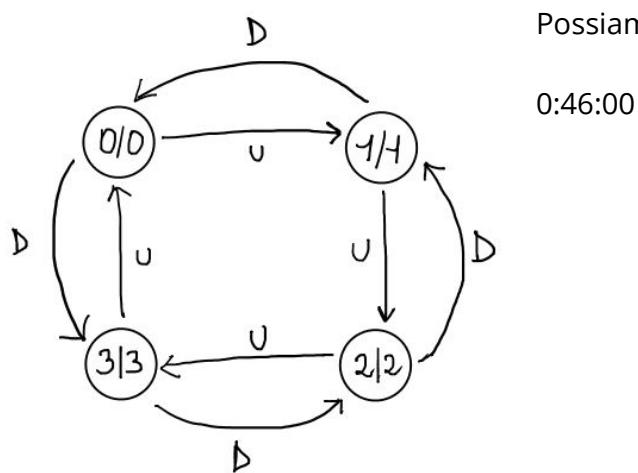
Potremmo in realtà realizzare un diagramma equivalente che però presenti solo 4 stati, anziché 5 come in questo caso. Tale diagramma è detto **diagramma minimo**. Infatti lo stato 4 potrebbe essere bypassato senza apportare modifiche al funzionamento del diagramma stesso.



Finora si è adattato sempre Mealy, qui di seguito viene mostrato lo stesso diagramma adottando Moore



### Contatore Up & Down



Possiamo realizzare anche un circuito Up & Down.

0:46:00

## Sintesi circuito sequenziale - Mealy

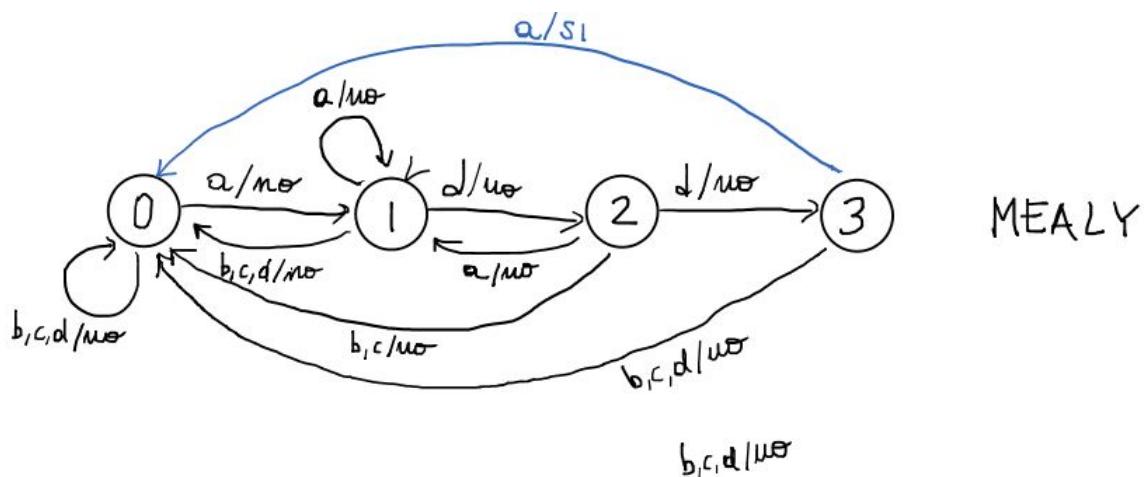
Rappresentata la macchina sequenziale con il diagramma degli stati è arrivato il momento di effettuare la sintesi. Si deve innanzitutto partire dalla tabella di verità.

Sfruttiamo l'esempio di prima.

$$I = \{a, b, c, d\}$$

$$O = \{\text{si}, \text{no}\}$$

$$S = \{0, 1, 2, 3\}$$



Per rappresentare gli elementi di I serviranno come minimo 2 variabili, nel caso di O 1 variabile ed infine per S almeno 2 variabili. Nel seguente schema vediamo una rappresentazione di I con 2 variabili affiancata da una possibile rappresentazione con 4.

$I$	$x_1 \ x_0$	$x_3 \ x_2 \ x_1 \ x_0$	$O$	$z$	$S$	$y_1 \ y_0$
a	0 0	0 0 0 1	si	1	0	0 0
b	0 1	0 0 1 0	no	0	1	0 1
c	1 0	0 1 0 0			2	1 0
d	1 1	1 0 0 0			3	1 1

Da qui possiamo procedere con la rappresentazione tabellare. Ad esempio dallo stato 0, con la lettera a, si giungerà allo stato 1 (prima cella) e si produrrà in uscita "no". Dallo stato 2, data la lettera d, si giungerà allo stato 3 e si produrrà in uscita "no".

$x_1 x_0$	00	01	10	11	
$y_1 y_0$	I	a	b	c	d
00 0	1/no	0/no	0/no	0/no	
01 1	1/no	0/no	0/no	2/no	
10 2	1/no	0/no	0/no	3/no	
11 3	0/si	0/no	0/no	0/no	

eve       $s_1 = 1$   
 $m_0 = 0$

$S_{\text{partenza}}$	I	$S_{\text{arrivo}}$	$\Sigma$
$y_1 y_0$	$x_1 x_0$	$y'_1 y'_0$	
0 0	0 0	0 1	0
0 0	0 1	0 0	0
0 0	1 0	0 0	0
0 0	1 1	0 0	0
0 1	0 0	0 1	0
0 1	0 1	0 0	0
0 1	1 0	0 0	0
0 1	1 1	1 0	0
1 0	0 0	0 1	0
1 0	0 1	0 0	0
1 0	1 0	0 0	0
1 0	1 1	1 1	0
1 1	0 0	0 0	1
1 1	0 1	0 0	0
1 1	1 0	0 0	0
1 1	1 1	0 0	0

E attraverso essa posso realizzare la tabella di verità. Infatti posso combinare in input lo stato di partenza e la lettera ottenuta e considerare come output lo stato in cui arrivo e l'uscita prodotta (si, no - 1, 0)

A partire dalla tabella di verità è possibile realizzare il classico circuito attraverso il processo di sintesi.

(Ricordiamo che la rappresentazione tabellare precedente non è una mappa di Karnaugh)

$y_1'$	$x_1 x_0$	00	01	11	10
$y_0'$	$y_1 y_0$	00	0	0	0
		01	0	0	1
		11	0	0	0
		10	0	0	1

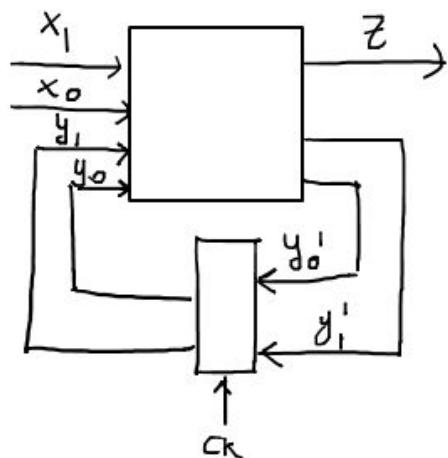
$$y_1' = \bar{y}_1 y_0 x_1 x_0 + y_1 \bar{y}_0 x_1 x_0 \Rightarrow$$

$y_0'$	$x_1 x_0$	00	01	11	10
$y_1'$	$y_1 y_0$	1	0	0	0
		1	0	0	0
		0	0	0	0
		1	0	1	0

$$y_0' = \bar{y}_1 \bar{x}_1 \bar{x}_0 + \bar{y}_0 \bar{x}_1 \bar{x}_0 + y_1 \bar{y}_0 x_1 x_0 \Rightarrow$$

$z$	$x_1 x_0$	00	01	11	10
$y_1 y_0$	0	0	0	0	0
	0	0	0	0	0
	1	0	0	0	0
	0	0	0	0	0

$$z = y_1 y_0 \bar{x}_1 \bar{x}_0 \Rightarrow$$

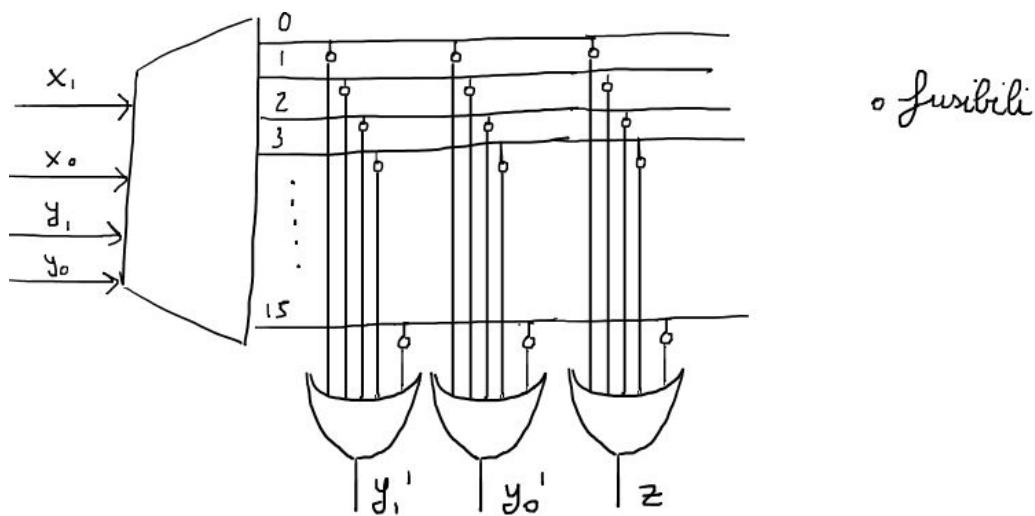


La combinazione di questi componenti consente di ottenere quindi una macchina sequenziale (in questo caso di Mealy).

Nulla è stato detto su come strutturare il componente che, sincronizzato con il clock, dovrà "salvare" il segnale di output di stato. Vedremo che si usano i cosiddetti flip flop D.

## Macchina sequenziale con ROM

Del caso precedente si è vista la sintesi di logica cablata. Proviamo a svolgere la stessa cosa tramite una ROM a 4 ingressi e 3 uscite (quindi si userà un decodificatore a 4 ingressi e 16 uscite). Aldilà del fatto che in commercio ROM a 4 ingressi e 3 uscite non esistono (è possibile al massimo trovare ROM da 4 ingressi e 4 uscite o da 4 ingressi e 8 uscite), devo però stabilire con quale logica devono essere bruciati (o non bruciati) i fusibili.



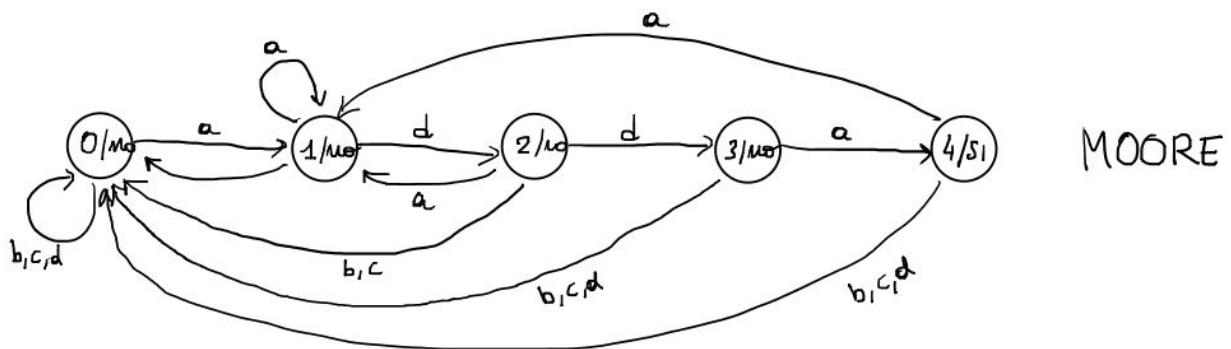
In realtà la risposta è più semplice di quanto si possa pensare. Per fare la sintesi con la ROM non serve realizzare la mappa di Karnaugh, ma una volta realizzata la tabella di verità basta vedere le colonne relative alle uscite. Lì dove ho uno 0 brucio, dove c'è un 1 non brucio (dove ho un 1 infatti voglio che il segnale passi e risulti in uscita).

Se avessi voluto usare una ROM per realizzare la stessa macchina sequenziale ma con lo schema di Moore, avrei dovuto utilizzare un decodificatore a 32 uscite (dato che gli ingressi in questo caso sono 5).

## Sintesi macchina sequenziale - Moore

Proviamo ad effettuare la sintesi della macchina di Moore già incontrata precedentemente. Si noti come in questo caso dovrà essere utilizzata anche una variabile  $y_2$ , essendo il numero di stati richiesti maggiore.

Di seguito il diagramma degli stati che interessa in questo caso.



MOORE

I	$x_1 \ x_0$	O	$z$	S	$y_2 \ y_1 \ y_0$
a	0 0	si	1	0	0 0 0
b	0 1	mo	0	1	0 0 1
c	1 0			2	0 1 0
d	1 1			3	0 1 1
				4	1 0 0

$y_2 \ y_1 \ y_0 \ x_1 \ x_0$	$y_2' \ y_1' \ y_0'$	$z$				
0 0 0 0 0	0 0 1	0	1 0 0 0 0	0 0 1	1	
0 0 0 0 1	0 0 0	0	1 0 0 0 1	0 0 0	1	
0 0 0 1 0	0 0 0	0	1 0 0 1 0	0 0 0	1	
0 0 0 1 1	0 0 0	0	1 0 0 1 1	0 0 0	1	
0 0 1 0 0	0 0 1	0	1 0 1 0 0	d.c.c.	d.c.c.	
0 0 1 0 1	0 0 0	0	1 0 1 0 1	d.c.c.	d.c.c.	
0 0 1 1 0	0 0 0	0	1 0 1 1 0	d.c.c.	d.c.c.	
0 0 1 1 1	0 1 0	0	1 0 1 1 1	d.c.c.	d.c.c.	
0 1 0 0 0	0 0 1	0	1 1 0 0 0	d.c.c.	d.c.c.	
0 1 0 0 1	0 0 0	0	1 1 0 0 1	d.c.c.	d.c.c.	
0 1 0 1 0	0 0 0	0	1 1 0 1 0	d.c.c.	d.c.c.	
0 1 0 1 1	0 1 1	0	1 1 0 1 1	d.c.c.	d.c.c.	
0 1 1 0 0	1 0 0	0	1 1 1 0 0	d.c.c.	d.c.c.	
0 1 1 0 1	0 0 0	0	1 1 1 0 1	d.c.c.	d.c.c.	
0 1 1 1 0	0 0 0	0	1 1 1 1 0	d.c.c.	d.c.c.	
0 1 1 1 1	0 0 0	0	1 1 1 1 1	d.c.c.	d.c.c.	

Procedo ora con le mappe di Karnaugh.

		$y_2 = 0$				$y_2 = 1$			
		00	01	11	10	10	11	01	00
$y_1$		00	0	0	0	0	0	0	0
01		0	0	1	0	-	1	-	-
11		0	0	0	0	-	-	-	-
10		0	0	1	0	-	1	-	-

$$y_1' = \bar{y}_1 y_0 x_1 x_0 + y_1 \bar{y}_0 x_1 x_0$$

		$y_2 = 0$				$y_2 = 1$			
		00	01	11	10	10	11	01	00
$y_0$		00	1	0	0	0	0	0	0
01		1	0	0	0	-	-	-	1
11		0	0	0	0	-	-	-	-
10		1	0	1	0	-	1	-	1

$$y_0' = \bar{y}_1 \bar{x}_1 x_0 + \bar{y}_0 \bar{x}_1 x_0 + y_1 \bar{y}_0 x_1 x_0$$

		$y_2 = 0$				$y_2 = 1$			
		00	01	11	10	10	11	01	00
$y_0$		00	1	0	0	0	0	0	0
01		1	0	0	0	-	-	-	1
11		0	0	0	0	-	-	-	-
10		1	0	1	0	-	1	-	1

$$y_0' = \bar{y}_1 \bar{x}_1 x_0 + \bar{y}_0 \bar{x}_1 x_0 + y_1 \bar{y}_0 x_1 x_0$$

		$y_2 = 0$				$y_2 = 1$			
		00	01	11	10	10	11	01	00
$z$		00	0	0	0	0	1	1	1
01		0	0	0	0	-	1	-	1
11		0	0	0	0	-	1	-	1
10		0	0	0	0	-	1	-	1

$$z = y_2$$

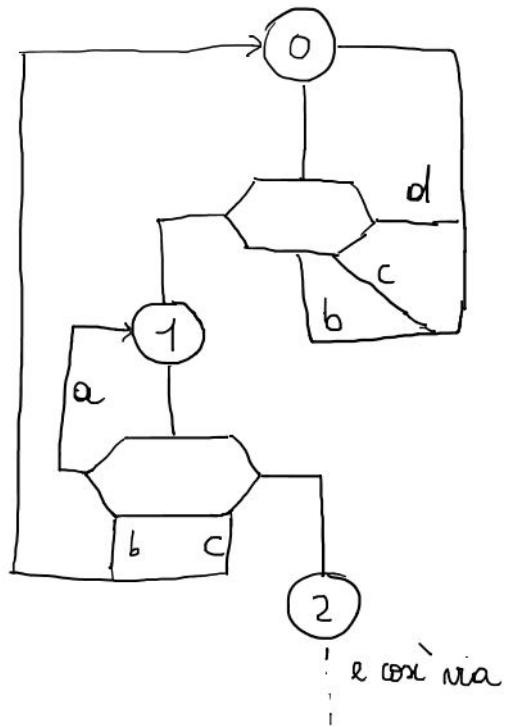
---

## Diagramma di flusso

Molte volte si è parlato di diagramma degli stati, ma un altro diagramma molto utile è quello di flusso.

Il diagramma di flusso è un modo molto pratico per rappresentare lo sviluppo di una macchina sequenziale ed il passaggio da uno stato ad un altro.

Inoltre, il diagramma di flusso viene usato anche per la microprogrammazione.



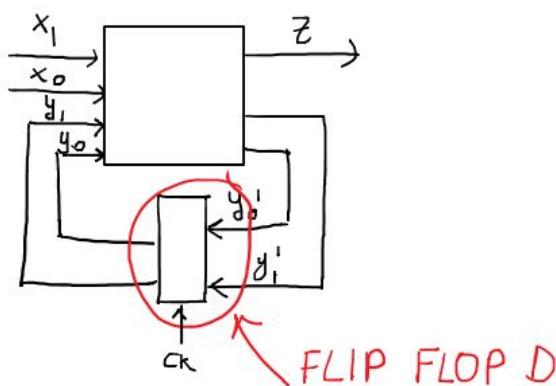
## Flip-Flop e registri

Come già visto nelle macchine sequenziali si può avere la necessità di memorizzare lo stato risultante di un'operazione per utilizzarlo come successivo stato corrente.

Questa operazione di archiviazione elementare può essere svolta con l'ausilio di flip-flop. I flip-flop prendono questo nome perché in passato erano realizzati con due transistor a valvole che facevano *flip* e *flop* quando venivano accesi e spenti.

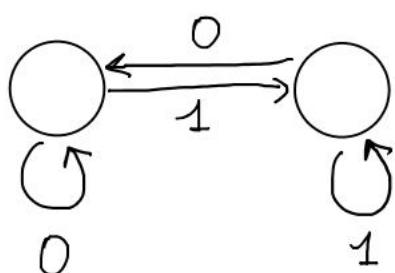
Il flip-flop, anche detto **bistabile**, può essere:

- **asincrono**, quando non è presente un segnale di sincronizzazione. Il bistabile rimane "stabile" in uno stato fino al variare delle variabili d'ingresso. È quindi il cambiamento dell'ingresso a comportare un cambiamento di stato.
- **clocked** (o **sincrono**), quando si fa uso di un clock per scandire i momenti in cui il flip-flop deve commutare in un nuovo stato (o permanere in quello corrente).

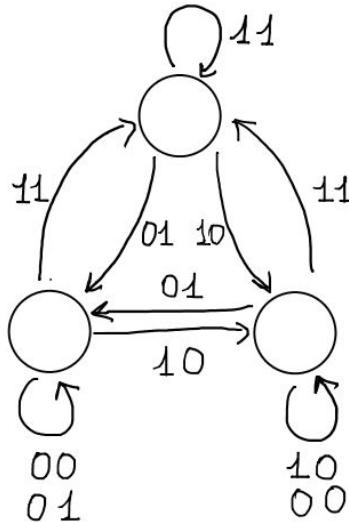


Nella macchina sequenziale prima affrontata si fanno proprio uso di flip-flop (in particolare di tipo D) per memorizzare temporaneamente lo stato prodotto in output dalla rete combinatoria ed utilizzarlo successivamente in input. A temporizzare questo meccanismo è un ciclo di clock.

I bistabili **asincroni** più semplici, con un solo bit di ingresso e un bit di stato possono essere rappresentati come nel seguente diagramma.

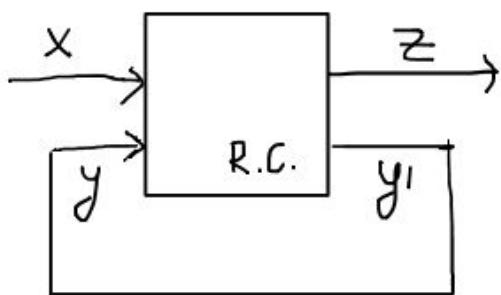


Quando si è nel primo stato e si ha un ingresso pari ad 1 si transita nel secondo stato, altrimenti se l'ingresso è 0 si rimane in quello. Uno stato è detto **transiente** quando esiste un ingresso che a partire da quello stato porta ad un altro stato.



Nel caso di due variabili la rappresentazione diventa un pochino più complicata. Introduciamo infatti un terzo stato cui si giunge con ingressi 11. A partire dallo stato in basso a sinistra posso rimanere fermo in quello stato per ingressi 00 e 01, mentre transito nello stato di destra quando l'ingresso cambia di un bit (da 00) diventando 10. Cambiando di 1 bit (da 10) si ottiene invece l'ingresso 11 e si passa allo stato in alto. Un discorso analogo varrà per gli altri stati. Tale diagramma corrisponde al **flip-flop JK**.

Questo bistabile ed, in maniera ancora più evidente, il flip-flop S-R vengono utilizzati in macchine asincrone. Tali macchine, a differenza di quelle già incontrate non presentano il segnale di sincronizzazione e sono perciò rappresentate con uno schema leggermente diverso.



Il vantaggio di questo circuito è che le variabili possono essere modificate in più modi, sia a livelli che impulsivamente, purché quando se ne varia una le altre rimangano fisse.

Il limite sta proprio in quest'ultimo fatto. Fintanto che siamo noi a variare i valori delle variabili di ingresso il comportamento corretto del circuito

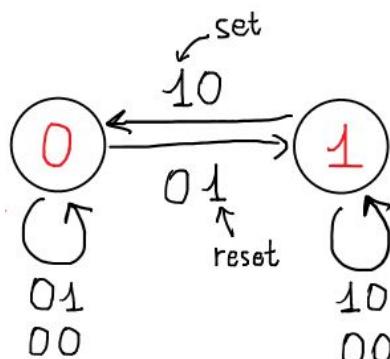
sarà garantito, ma quando tali ingressi non siano, per motivi fisici, controllabili tale garanzia non potrà sussistere.

Per quanto il nostro obiettivo sia quello di evitare la contemporaneità, non si ha mai pieno controllo sulla Natura. Non è quindi difficile risultare in fenomeni talvolta inspiegabili.

Ai fini pratici progettare un circuito di questo tipo corrisponde a porre le basi della progettazione di un circuito sincrono.

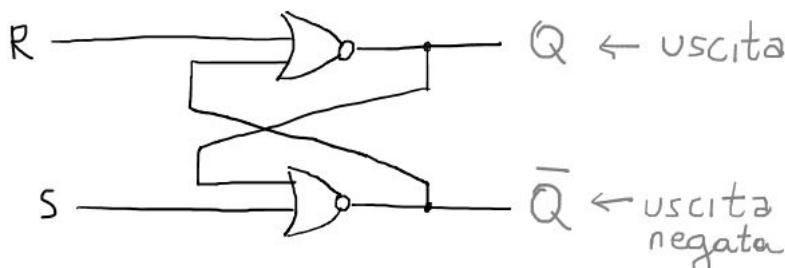
### Flip-flop S-R

Una delle realizzazioni circuitalmente meno onerose di un bistabile è quella di tipo S-R (Set-Reset) costituito da due porte NOR retroazionate. Il flip-flop S-R è concettualmente ancora più semplice di quello JK prima visto.



Il diagramma degli stati di questo flip-flop è infatti banale. Si hanno due stati e due variabili di ingresso (rispettivamente set e reset). Al variare di una variabile di stato si può soggiornare nello stato corrente oppure passare all'altro stato.

La condizione, già prima affrontata è che si varino solo uno dei bit alla volta. Questo flip-flop può essere infatti visto come un interruttore della luce: se si "attiva" il bit del set (si preme sul pulsante on) la luce si accende; rimane accesa fintanto che ad attivarsi è il bit di set o fintanto che non si attivi alcuni bit. Non appena si attiva il bit del reset (si preme sul pulsante off) la luce si spegne. E rimarrà spenta fino a che non si riattivi il bit del set. L'attivazione di entrambi i bit (ossia un segnale di ingresso 11) porterebbe il sistema in una condizione di instabilità: sarebbe infatti come premere sugli interruttori ON ed OFF contemporaneamente.



Dal punto implementativo posso utilizzare due porte NOR retroazionate. La cosa interessante è che, semplicemente giocando con le variabili d'ingresso, è possibile vedere come l'uscita non cambia quando entrambe le variabili sono poste a 0, mentre cambia quando si attiva il set od il reset. Inoltre è possibile vedere come l'uscita del secondo NOR possa essere vista come la negazione dell'uscita. In questo modo si ottiene anche il risultato negato "gratuitamente", senza l'utilizzo di una porta NOT applicata appunto all'uscita del flip-flop.

## Segnale di sincronizzazione

Le macchine asincrone non sono oggetto di studio nel corso. I flip-flop JK e quelli S-R servono infatti come base di partenza. Di nostro interesse sono invece i circuiti con un temporizzatore.

Un segnale di sincronizzazione è una variabile binaria che viene usata per abilitare la commutazione di un flip-flop (sincronizzato). Nel caso ideale la variabile oscilla da 0 ed 1 in maniera netta.



In realtà i livelli 0 ed 1 possono subire disturbi transitori, per questo si sfruttano principalmente i fronti di salita (**positive edge-triggered**) e quelli di discesa (**negative edge-triggered**), molto più immediati da percepire. Quando si usano i livelli i flip-flop sono anche chiamati **Latch**.

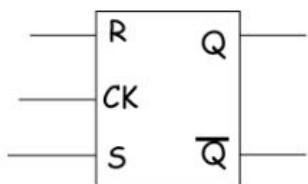
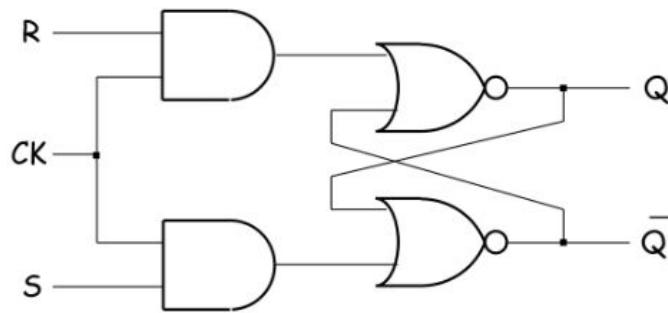
Per evitare i transitori (ed i problemi annessi) spesso si utilizzano soluzioni **Master Slave**, ove si campiona sui fronti di salita e si commuta sui fronti di discesa. In questo soluzioni a favore dell'affidabilità si aumentano però i costi perché servono due flip-flop.

## Flip-flop S-R di tipo Latch

Nel nostro caso verranno usati i Latch di serie D, che presentano in uscita ciò che viene campionato in ingresso durante il livello. Questo flip-flop si basa in parte sul flip-flop S-R.

Gli ingressi S ed R del bistabile invece di essere diretti vengono “mascherati” dal segnale di abilitazione (clock) tramite delle porte AND. Se il segnale di abilitazione vale 0 gli ingressi al bistabile saranno entrambi 0 e quindi il bistabile rimarrà nello stato corrente.

Se il clock è pari ad 1 ed il bit R di reset vale 1, il flip flop verrà resettato; se è spento rimane spento, se è acceso viene spento. In maniera simmetrica questo avviene col bit del Set.



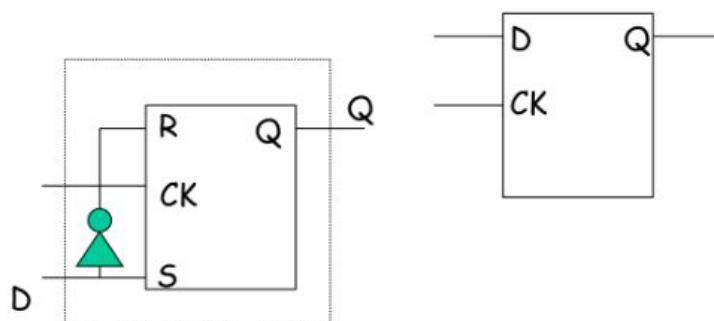
$CK$	$S$	$R$	$Q'$
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	?

Le uscite prodotte, anche in questo caso sono due, dato che si può sfruttare la logica del circuito per produrre anche la negazione della variabile d'uscita, senza ulteriori costi.

## Flip-flop D

Il flip-flop D è realizzato in un modo simile a quello S-R, ma con una porta NOT che permette di poter utilizzare un solo ingresso più uno di abilitazione.

Così facendo quando il segnale di input vale 0, il valore di Set è 0, quello di Reset è 1 (e viceversa).



Questo flip-flop produce in uscita ciò che è presente in ingresso quando il CK = 1, altrimenti presenta l'ultimo valore di D quando il CK commuta da 1 a 0.

## Registri

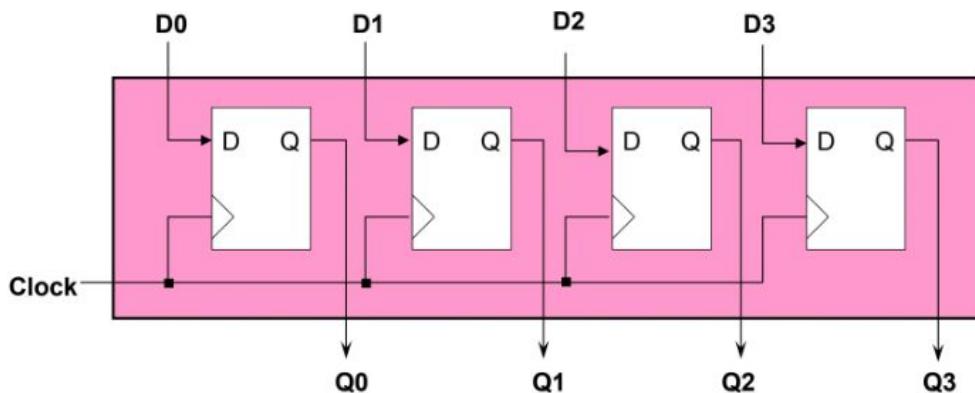
I registri sono elementi di memoria in grado di memorizzare un insieme di  $n$  bit. Essi sono costituiti da un'insieme di  $n$  flip-flop S-R di tipo Latch o di tipo D. I tipi di registri sono svariati e si differenziano sia per le modalità di scrittura/scrittura dei dati (**parallelo o seriale**), sia per le operazioni sui dati (**scorrimento a destra, scorrimento a sinistra, scorrimento circolare**).

### Registro parallelo-parallelo

Un registro parallelo-parallelo è caratterizzato da una serie in parallelo di flip-flop di tipo D.

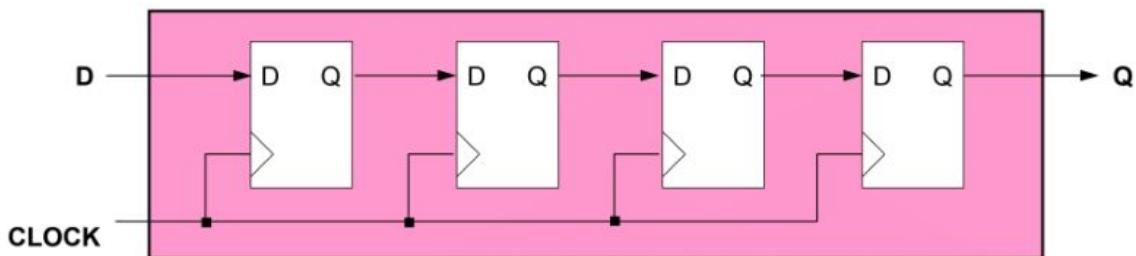
La seguente figura mostra un registro a 4 bit. Ci sono 4 bit di ingresso e 4 d'uscita. Fino a che non viene dato il campionamento, in uscita si ha la vecchia scrittura, non appena viene dato il segnale di abilitazione si modifica il contenuto del flip-flop.

Quindi, se al primo flip-flop ho in ingresso 0, in uscita, dopo un clock, si avrà 0.



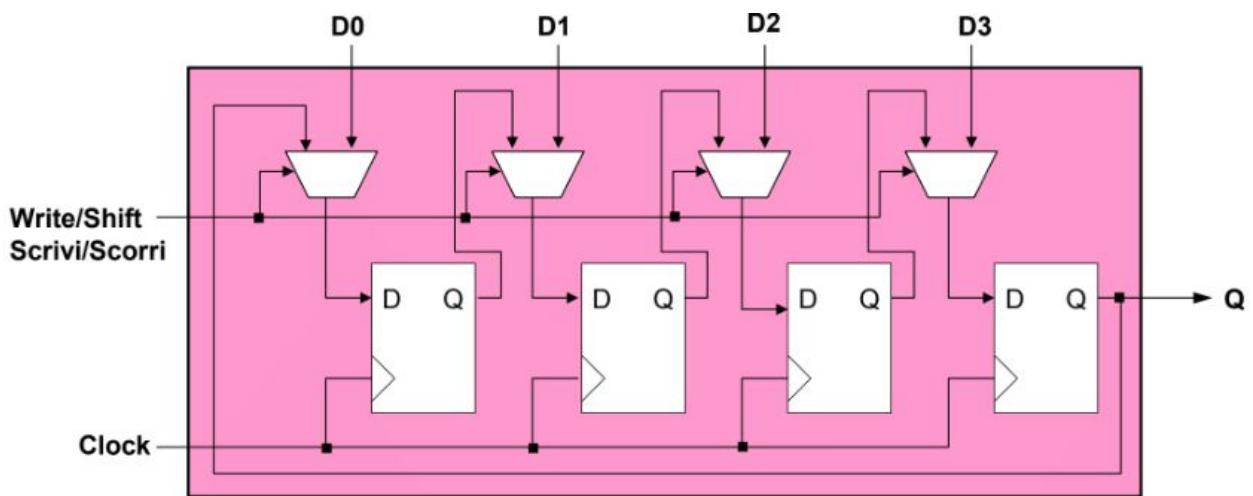
### Shifter Register

Lo shifter register ha un solo ingresso e  $1$  o  $n$  uscite. Il suo scopo è quello di produrre in output l'ingresso dopo  $n$  colpi di clock. Uno shifter register risulta molto comodo nelle operazioni matematiche, come per le moltiplicazioni.



## Registro circolare

Un registro circolare è basato su uno shifter register ove l'uscita di ogni flip-flop D viene posta anche come potenziale ingresso del successivo flip-flop. In particolare si utilizzano dei commutatori (quindi dei multiplexer) per selezionare come input del flip-flop l'uscita passata dal precedente flip-flop o l'ingresso passato dall'esterno. Tali multiplexer commutano dunque, in base al segnale di abilitazione ricevuto, il registro circolare nella modalità Scrivi o in quella Scorri.



Questa soluzione presenta, come in tutti i casi, punti a favore e a sfavore.

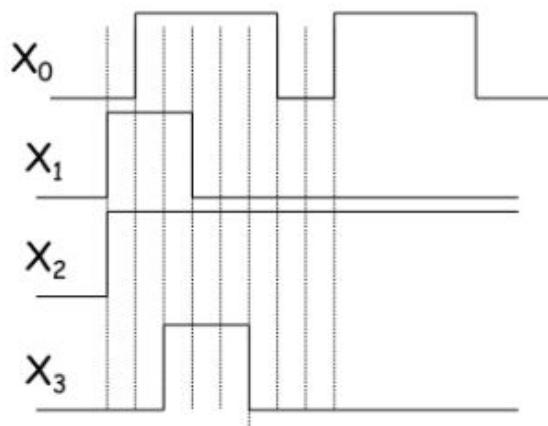
Per fare uno shift di una posizione avrò bisogno di un colpo di un solo colpo di clock, ma se il registro è di 64 bit e mi serve uno shift di 63 posizioni me ne serviranno molti di più. Quindi il numero di colpi di clock cresce al crescere del numero di posizioni da shiftare.

## Sintesi reti LCC

Si è già visto come fare la sintesi di una macchina sequenziale. Tuttavia per realizzarne una non è solo necessario codificare gli insiemi I,S,O con variabili di commutazione e realizzare le funzioni  $\delta$  ed  $\omega$  con reti combinatorie, ma bisogna anche considerare il comportamento temporale delle variabili ingresso-uscita.

In particolare si vedranno solo reti di tipo LCC (Level Level Clocked).

### Classificazione variabili di ingresso



Un segnale può essere **a livelli** oppure **impulsivo**, a seconda del suo comportamento nei vari istanti di campionamento.

Nella figura affianco abbiamo rappresentati diversi segnali.

Analizziamo  $X_0$  e  $X_1$ . Si può notare che in alcuni istanti quando  $X_0$  vale 0,  $X_1$  vale 1 (e viceversa).

Si può quindi dire che  $X_0$  e  $X_1$  sono a livelli.

$X_2$ , quando commuta, rimane sempre costante ad 1.  $X_3$  invece varrà 1 solo in una "porzione". Si dirà quindi che  $X_2$  è a livelli e che  $X_3$  è impulsiva rispetto al livello (o impulsiva rispetto a  $X_2$ ).

La  $X_3$  rispetto alla  $X_1$  ha un comportamento a livelli, mentre rispetto a  $X_0$  è impulsiva.

### Reti LCC

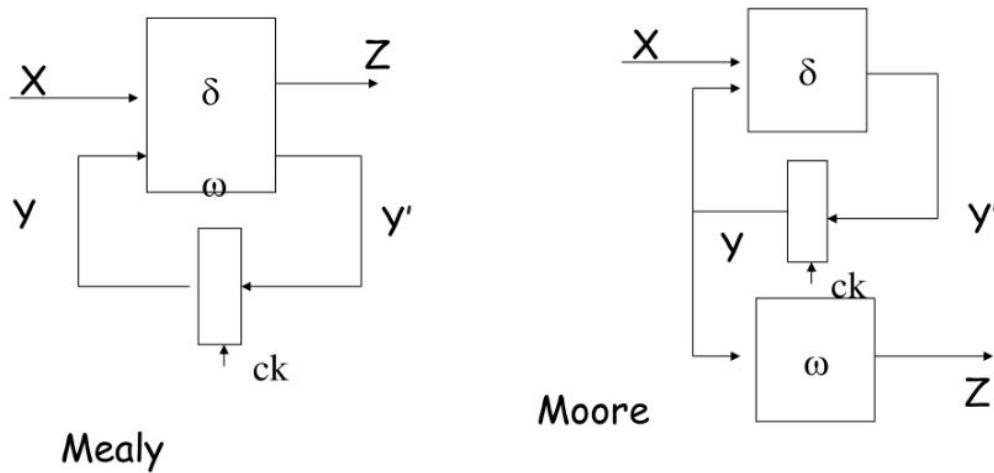
La rete sequenziale lavora con le seguenti ipotesi:

- Variabili d'ingresso di tipo a livello (ossia i valori in ingresso rimangono fissi per un periodo  $T$  sufficientemente lungo per far assumere all'uscita il nuovo valore di regime, ossia  $T > d$ )
- Variabili di uscita a livello

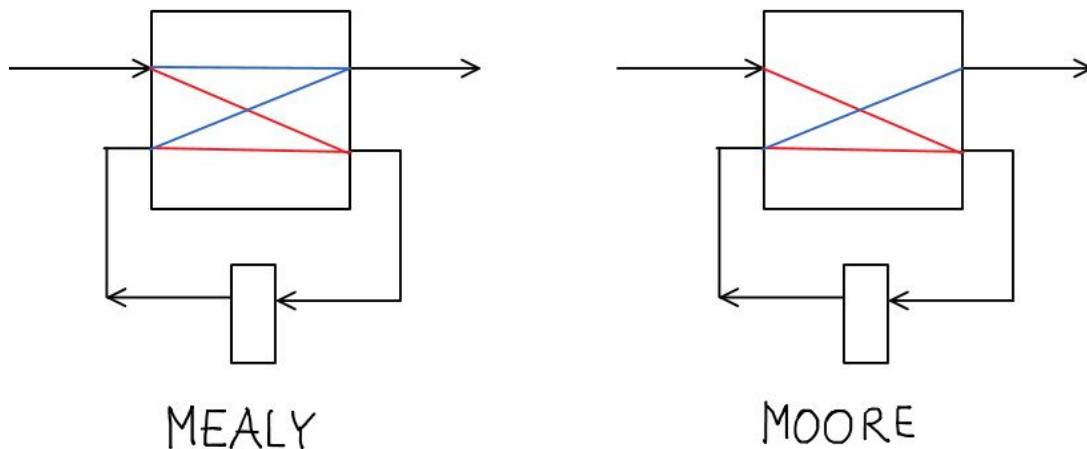
- Segnale di abilitazione “positive or negative edge triggered” (già visti in precedenza) o a livelli. In quest’ultimo caso la variabile di commutazione deve essere pari ad 1 per un periodo di tempo sufficiente per far commutare i flip-flop, ma inferiore al minimo tempo di commutazione dei circuiti combinatori che calcolano lo stato successivo, altrimenti si potrebbero avere più commutazioni.

## Il modello strutturale di Mealy e Moore

Di Mealy e Moore si è già ampiamente discusso, ma qui viene proposto il modello strutturale. Una macchina Mealy e quella Moore possono essere così rappresentati.



In molte situazioni si adotta tuttavia una rappresentazione comune delle due macchine e si pone il rilievo sulle relazioni ingresso-stato ed ingresso-stato-uscita.



Nella macchina di Mealy lo stato dipende sia dall’ingresso che dallo stato corrente.

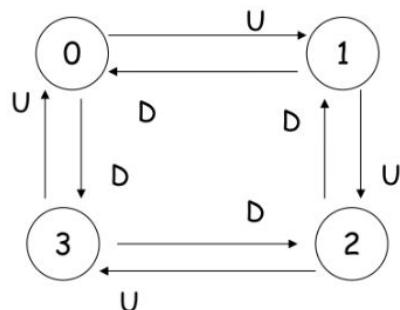
L'uscita dipende sia dallo stato che dall'ingresso.

Nella macchina di Moore, invece, l'uscita dipende solo dallo stato.

Supponiamo che cambi l'ingresso, indipendentemente dal clock, nel caso di Mealy cambia lo stato (dopo che essere stato campionato) ed anche l'uscita, nel caso di Moore a cambiare è solo lo stato. Quindi nel caso di Moore dato l'ingresso prima che cambi l'uscita bisognerà attendere che esso sia campionato (e quindi faccia cambiare lo stato) e che si stabilizzi il circuito.

### Esempio contatore Up-Down

Si era già valutata la possibilità di realizzare un contatore Up-Down sfruttando la logica di una macchina di Mealy o di Moore. Per esercizio procediamo con il processo di sintesi.



uscita = stato

$$\begin{aligned} I &= \{U, D\} \\ O &= \{0, 1, 2, 3\} \\ S &= \{0, 1, 2, 3\} \end{aligned}$$

ingresso	stato		uscita
	U	D	
0	1	3	0
1	2	0	1
2	3	1	2
3	0	2	3

I	x
U	0
D	1

S	y <sub>2</sub> y <sub>1</sub>
0	00
1	01
2	10
3	11

O	z <sub>2</sub> z <sub>1</sub>
0	00
1	01
2	10
3	11

Gli ingressi, gli stati e gli output dovranno esser tutti codificati in binario. L'ultima mappa non è ancora la mappa di Karnaugh, che può comunque essere facilmente calcolata.

ingresso	stato		uscita
	U	D	
0	1	3	0
1	2	0	1
2	3	1	2
3	0	2	3

x	0	1	z <sub>2</sub> z <sub>1</sub>
y <sub>2</sub> y <sub>1</sub>	00	01	00
01	10	00	01
10	11	01	10
11	00	10	11

$y_2$	$x$	0	1
$y_1$	00	0	1
	01	1	0
	11	1	0
	10	0	1

$$\Rightarrow y_2 = y_1 \bar{x} + \bar{y}_1 x$$

$y_1$	$x$	0	1
$y_2$	00	1	1
	01	0	0
	11	1	1
	10	0	0

$$\Rightarrow y_1 = \bar{y}_2 \bar{y}_1 + y_2 y_1$$

$z_2$	$x$	0	1
$y_2 y_1$	00	0	0
	01	0	0
	11	1	1
	10	1	1

$$\Rightarrow z_2 = y_2$$

$z_1$	$x$	0	1
$y_2 y_1$	00	0	0
	01	1	1
	11	0	0
	10	1	1

$$\Rightarrow z_1 = \bar{y}_2 y_1 + y_2 \bar{y}_1$$

In realtà  $z_2$  e  $z_1$ , essendo "indipendenti" dall'ingresso sarebbero potute essere stati calcolati senza mappa di Karnaugh, apportando le semplici semplificazioni a mano. Qui per completezza sono state rappresentate.

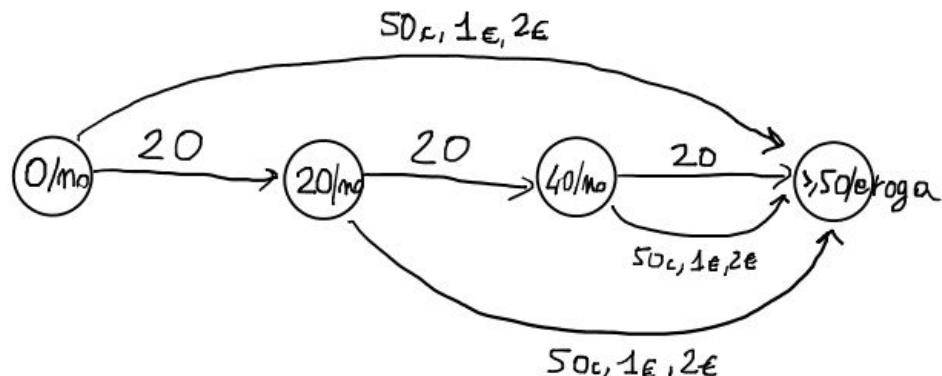
## Esercizio

Vogliamo implementare una macchina sequenziale del distributore del caffè. Possono essere inseriti nella macchina 4 tipi di moneta, cioè gli ingressi, e la macchina si potrà trovare in due stati.

$$I = \{20\text{c}, 50\text{c}, 1\text{€}, 2\text{€}\}$$

$$S = \{\text{eroga}, \text{non eroga}\}$$

Un caffè costa 0,50€, pertanto solo inserendo un importo uguale o superiore verrà erogato. La macchina, per semplicità non dà resto.



Associamo i singoli elementi di stato ed ingresso ad una combinazioni di variabili binarie.

	$X_1$	$X_0$		$y_1$	$y_0$		$z$	
$20\text{c}$	1	1		0	0		si*	1
$50\text{c}$	1	0		20	1	1	mo*	0
$1\text{€}$	0	1		40	1	0	*si = eroga	
$2\text{€}$	0	0		>50	0	1	*mo = non eroga	

Si procede con la realizzazione della tabella di transizione che mostra gli stati raggiunti dati gli stati correnti e gli ingressi possibili. La tabella di transizione, per quanto sia utile per la scrittura delle mappe di Karnaugh, non è una mappa di Karnaugh!

$y_1' y_0'$	$x_1 x_0$	11	10	01	00
$y_1 y_0$		11	01	01	01
00	11	01	01	01	01
11	10	01	01	01	01
10	01	01	01	01	01
01	-	-	-	-	-

$y_1' y_0'$	$x_1 x_0$	11	10	01	00
$y_1 y_0$		0	0	0	0
00	0	0	0	0	0
11	0	0	0	0	0
10	0	0	0	0	0
01	1	1	1	1	1

A questo punto si potrebbe procedere con la tabella di verità e da lì con le vere mappe di Karnaugh.

$y_1$	$y_0$	$x_1$	$x_0$	$y_1' y_0'$	$z$
0	0	0	0	0	0
0	0	0	1	0	0
:	:	:	:	:	:

Questa volta però vogliamo far uso della tabella di transizione prima realizzata. Quelli nelle celle sono gli stati raggiunti dati gli ingressi e gli stati correnti. Tali stati raggiunti sono rappresentati da due bit, il primo sarà  $y_1'$  ed il secondo  $y_0'$ . Quindi analizzando i due bit singolarmente possiamo vedere che il bit della variabile  $y_1'$  è pari ad 1 solo nelle celle 00-11 e 11-11, altrove è sempre zero. Si potrà fare un'analogia considerazione anche per il secondo bit. Queste considerazioni sono le stesse che si sarebbero potute fare con una tabella di verità, quindi possiamo procedere a rappresentare le mappe di Karnaugh per i singoli bit dello stato raggiunto.

$y_1'$

$x_1 x_0$	00	01	11	10
$\bar{y}_1 y_0$	0	0	1	0
00	0	0	1	—
01	—	—	1	—
11	0	0	1	0
10	0	0	0	0

$\Rightarrow y_1' = \bar{y}_1 x_1 x_0 + y_0 x_1 x_0$

$y_0'$

$x_1 x_0$	00	01	11	10
$\bar{y}_1 y_0$	1	1	1	1
00	1	1	—	1
01	1	1	—	1
11	1	—	0	1
10	1	—	—	1

$\Rightarrow y_0' = \bar{x}_1 + \bar{y}_0 + \bar{x}_0$

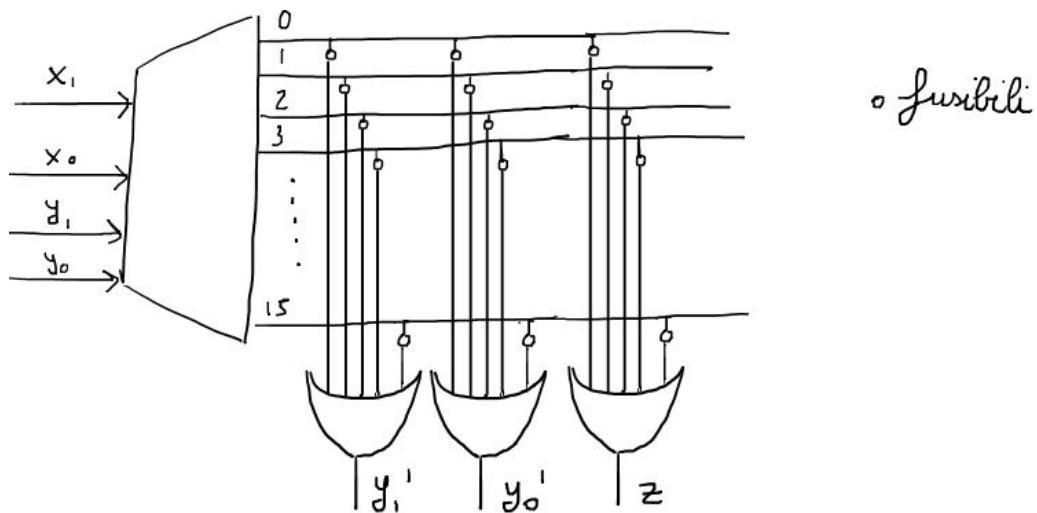
$z$

$x_1 x_0$	00	01	11	10
$\bar{y}_1 y_0$	0	0	0	0
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	0	0	0	0

$\Rightarrow z = \bar{y}_1 y_0$

Qui affianco è rappresentata, per completezza, anche la mappa di Karnaugh di z. Sarebbe stato tuttavia possibile calcolarlo direttamente a mente.

È interessante considerare il fatto che si sarebbe potuto realizzare la stessa macchina sequenziale anche con l'ausilio di una ROM.



Per definire quali fusibili bruciare e quali no, avremmo dovuto realizzare la tabella di verità

$S_{\text{partenza}}$	$I$	$S_{\text{arrivo}}$	$Z$			
$y_1, y_0$	$x_1, x_0$	$y_1', y_0'$	$z$			
0 0 0 0		0 1	0	1	0 0 0	0 1 0
0 0 0 1		0 1	0	1	0 0 1	0 1 0
0 0 1 0		0 1	0	1	0 1 0	0 1 0
0 0 1 1		1 1	0	1	0 1 1	0 1 0
0 1 0 0		d.c.c.	0	1	1 0 0	0 1 1
0 1 0 1		d.c.c.	0	1	1 0 1	0 1 1
0 1 1 0		d.c.c.	0	1	1 1 0	0 1 1
0 1 1 1		d.c.c.	0	1	1 1 1	1 0 1

In corrispondenza di ogni uscita si sarebbero dovute bruciare i fusibili corrispondenti ad una combinazione che risulta in 0, non bruciare quelli in 1.

I fusibili corrispondenti al don't care condition (d.c.c.) potranno essere bruciati.

# Rappresentazione numeri interi e reali

Per comprendere appieno ciò che verrà trattato più avanti sono qui proposti alcuni concetti preliminari sulla rappresentazione e conversione di numeri interi e reali.

## Conversione numero da binario a decimale

La conversione da binario a decimale è banale. Si noti solo che nel caso in cui compaia la virgola, si andranno a convertire separatamente le cifre prima e dopo la virgola.

$$\begin{aligned} & \left( \underbrace{101011}_{1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0}, \underbrace{1011}_{1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4}} \right)_2 = \\ & 32 + 0 + 8 + 0 + 2 + 1 + 0,5 + 0 + 0,125 + 0,0625 \\ & = \left( \underbrace{43}_{10}, \underbrace{6875}_{10} \right)_{10} \end{aligned}$$

## Conversione da decimale a binario

Si divide sempre per 2 e si segna il resto.

$$(57)_{10} = (111001)_2$$

$$\begin{array}{r|rr} 57 & 2 & 1 \\ 28 & 2 & 0 \\ 14 & 2 & 0 \\ 7 & 2 & 1 \\ 3 & 2 & 1 \\ 1 & 2 & 1 \\ \hline 0 & & \uparrow \end{array}$$

Nel caso di un numero frazionario anziché dividere per 2 si moltiplica per due e si segna la parte intera.

$$(0,6875)_{10} = (0,1011)_2$$

$0,6875$	$\times 2$	$1,375$	$1$	$\downarrow$
$0,375$		$0,75$	$0$	
$0,75$		$1,5$	$1$	
$0,5$		$1$	$1$	
$0$				

Non tutti i numeri decimali possono essere convertiti in binario. Può succedere infatti che un numero decimale risulti in un numero periodico in binario. Bisognerà dunque tener conto del livello di errore introdotto nel momento in cui si lavora con questi numeri.

Nel seguente caso si arriva in un punto in cui ricompare 0,2. Quindi tutto ciò che seguirà sarà una ripetizione di quanto precedente scritto.

$$(0,1)_{10} = (0,0\overline{0011})_2$$

$0,1$	$0,2$	$0$	$\downarrow$
$\rightarrow 0,2$	$0,4$	$0$	
$0,4$	$0,8$	$0$	
$0,8$	$1,6$	$1$	
$0,6$	$1,2$	$1$	
$\rightarrow 0,2$	$0,4$		

Un elaboratore che lavora in binario non è quindi necessariamente più preciso di un calcolatore umano ma, tenuto in considerazione l'errore, è infinitamente più veloce.

Per la rappresentazione di numeri negativi si può intuitivamente anteporre un bit di segno (che vale 1 nel caso in cui il numero sia negativo e 0 in cui sia positivo). È evidente che quello del bit di segno sia un'informazione aggiuntiva. Per questo motivo si può adottare un'altra rappresentazione dei numeri, quella del complemento a 2.

## Complemento a 2

Dato un numero decimale il suo corrispettivo complemento a due è calcolato nel seguente modo

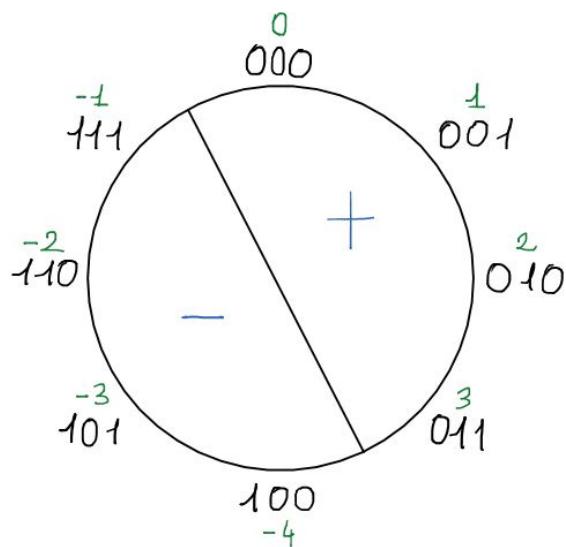
$$N \rightarrow 2^n - N$$

ove n è il numero di bit con cui si vuole rappresentare il numero. È evidente che il numero di bit utilizzati condizionerà anche l'insieme di numeri rappresentabili

$$N = 10100 \quad n=8 \quad \begin{array}{r} 10000000 - \\ 00010100 = \\ \hline 11101100 \end{array} \Rightarrow 11101100$$

Tutti i valori che iniziano con 1 saranno negativi, altrimenti positivi. Un numero in complemento a 2 può essere ottenuto invertendo tutti i bit a partire dal primo bit 1 meno significativo (che non va invertito) andando verso sinistra.

Possiamo fare alcune considerazioni sul complemento a 2. Per farlo scegliamo n=3.



Nello schema affianco i valori sono rappresentati attorno ad una circonferenza, come le ore di un orologio.

- tutti i numeri che iniziano con 1 sono negativi, quelli che iniziano con 0 sono positivi (come tra l'altro ci saremmo aspettati)
- Una stringa di bit con tutti 1 vale -1, indipendentemente da n

$$010 \rightarrow +2 \\ 110 \rightarrow -2$$

$$100 \rightarrow -4 \\ 100 \rightarrow -4$$

- Con 3 bit non è possibile ottenere il numero +4. Infatti, mentre complementando +2 si ottiene -2 (o viceversa), complementando -4 si ottiene di nuovo -4. Questo introduce un ulteriore fattore di errore da tenere in considerazione.

### Rappresentazione numeri decimali (standard IEE 754)

Un altro importante tema è quello della rappresentazione di un numero decimale su un calcolatore moderno. Imponendo limiti troppo stretti sulla parte decimale o su quella intera (in termini di bit dedicati) si finirebbe col limitare i campi applicativi.

Questo problema si risolve con la virgola mobile ed in particolare con lo standard IEE 754.

Un numero in virgola mobile presenta

- 1 bit di segno  $s$  (0 = positivo, 1 = negativo)
- 8 bit di esponente  $e$ 
  - si hanno dunque  $2^8 = 256$  possibili valori, di cui 244 utilizzabili (0 e 255 assumono particolari significati, mostrati più avanti), in un intervallo  $[-126, 127]$ , con bias 127.
- 23 bit di mantissa  $m$

Posto  $E = e - 127$ , un numero  $N$  può essere visto come  $N = (-1)^s \cdot 2^E \cdot 1.m$

In particolare

$e$	$m$	$N$
$[1, 254]$	qualsiasi	$(-1)^s \cdot 2^E \cdot 1.m$
0	$\neq 0$	$(-1)^s \cdot 2^E \cdot 0.m$ Numeri denormalizzati
0	0	$(-1)^s \cdot 0$
255	0	$(-1)^s \cdot \infty \leftarrow \pm \infty$
255	$\neq 0$	NaN $\leftarrow$ Not a number

Utilizziamo questa notazione perché la corrispondente implementazione hardware, la FPU (floating-point unit), è molto veloce nelle operazioni.

### Esercizio

Convertiamo in virgola mobile il numero  $(-5,828125)_{10}$

$$N = (-5,828125)_{10}$$

$S = 1$  ← bit di segno

$\begin{array}{c cc} 5 & 2 & 1 \\ 2 & 2 & 0 \\ -1 & 2 & 1 \uparrow \\ 0 & & \end{array}$	$\begin{array}{c ccc} 0,828125 & 1,65625 & 1 & \downarrow \\ 0,65625 & 1,3125 & 1 & \\ 0,3125 & 0,625 & 0 & \\ 0,625 & 1,25 & 1 & \\ 0,25 & 0,5 & 0 & \\ 0,5 & 1 & 1 & \\ 0 & & & \end{array}$
$(5)_{10} = (101)_2$	$(0,828125)_{10} = (0,110101)_2$

Dai  $101,110101$  che non è però nella forma  $1, m$

$$\text{Dunque } 101,110101 = \underbrace{1,01110101}_{m} \cdot 2^{2 \rightarrow E}$$

$$e = E + 127 = 2 + 127 = 129 \leftarrow \begin{array}{l} \text{normalizzazione} \\ \text{in bias.} \end{array}$$

$$e = 10000001$$

$\begin{array}{c cc} 129 & 2 & 1 \\ 64 & 2 & 0 \\ 32 & 2 & 0 \\ 16 & 2 & 0 \\ 8 & 2 & 0 \\ 4 & 2 & 0 \\ 2 & 2 & 0 \\ 1 & 2 & 1 \uparrow \end{array}$
--

---

Dunque

$$N = \underbrace{1}_{S} \underbrace{10000001}_{E} \underbrace{01110101}_{m} \underbrace{000...0}_{\substack{\text{zeri} \\ \text{aggiuntivi}}}$$

Notiamo come la mantissa prima ottenuta non occupi tutti i 23 bit a sua disposizione, si andranno quindi a porre degli zeri aggiuntivi.

### Esercizio

Convertiamo ora un numero nel formato IEE 754 in decimale

$$S = 1$$

$$E = 10000001 \Rightarrow (129)_{10} \Rightarrow E = 129 - 127 = 2$$

$$m = 0100\dots00 \Rightarrow 1,01 \Rightarrow 1 \cdot 2^0 + 1 \cdot 2^{-2} = 1 + 2^{-2} = 1,25$$

$$\Rightarrow N = (-1)^S \cdot 2^E \cdot m, m = -1 \cdot 2^2 \cdot 1,25 = \underline{-5}$$

All'elaboratore non importa se deve elaborare un int, un unsigned int od un float; non importa dunque il tipo della parola. Ciò che conta è cosa l'elaboratore deve fare con quella sequenza di bit, a prescindere dalla rappresentazione adottata e dal tipo di dato.

## Flags: alcune considerazioni

I flags sono registri, presenti in tutti i processori, che contengono dei bit di stato che tengono traccia di alcuni tipi di operazioni.

L'operazione **cmp** (compare) disponibile in Assembly prende compta il primo termine con il secondo. A quel punto con una successiva operazione si potrà verificare l'entità della comparazione. Ad esempio comparando due valori si può, con il comando **jbe** verificare che il primo sia minore o uguale del secondo. La CPU tiene traccia dell'operazione di compare in alcuni appositi flag e a quel punto potrà verificare se una data condizione sia verificata o meno.

Come già detto, al processore non interessa conoscere la tipologia di dato, almeno in fase computativa. Tuttavia alcuni flag vengono settati a determinati valori se, ad esempio, si fa uso di un int o di un unsigned int.

Il **flag N** (Negative) contiene un bit informativo per sapere se il risultato di una data operazione è negativo.

La ALU tratterà gli operandi come se fossero segnati. Quindi il bit N ricorderà che se viene fatta l'operazione con una logica non segnata allora il risultato è o meno negativo.

Il **flag C** (carry) consente di rappresentare il bit di riporto della sigla più significativa. Così facendo si può capire se il numero del risultato non sia rappresentabile con il numero di bit a disposizione.

Per mostrare l'importanza dei flag facciamo delle considerazioni a titolo esclusivamente esemplificativo (la seguente istruzione nella realtà non verrebbe compilata).

`cmp -11, 10`

$$\begin{array}{r} 10 \\ - 11 \\ \hline ? \end{array} \quad \rightarrow \text{In binario, con 4 bit}$$

$$\begin{array}{r} 1010 \\ (1)0101 \\ \hline (1)1111 \\ \downarrow N=1 \\ \downarrow C=1 \end{array}$$

$N=1$  e  $C=1 \Rightarrow 11 > 10$

Da notare che il numero -11 non è rappresentabile con soli 4 bit, viene quindi settato il carry ad 1.

In presenza di N=1 e C=1 la ALU può determinare che il primo operando è maggiore del secondo.

Proviamo con un altro compare.

cmp 10, 11

$$\begin{array}{r}
 11 + \\
 -10 = \\
 \hline ?
 \end{array}
 \rightarrow \text{In binario, con 4 bit}$$

1011+
(1)0110
<hr/>
1(0)0001
\Leftrightarrow
N=0
C=0
Z=0

$$N=0, C=0 \text{ e } Z=0 \Rightarrow 10 < 11$$

In questo caso è stato introdotto anche il **flag Z** (Zero) che vale 1 quando il risultato è pari a 0, 1 altrimenti. In presenza di N=0, C=0 e Z=0 la ALU può determinare che il primo operando è minore del secondo. Da notare come in questo l'operazione produce in totale 6 bit di risultato, di cui 1 va perso (ed è per questo sbarrato).

cmp 10, 10

$$\begin{array}{r}
 11 + \\
 -10 = \\
 \hline ?
 \end{array}
 \rightarrow \text{In binario, con 4 bit}$$

1010+
(1)0110
<hr/>
1(0)0000
\Leftrightarrow
N=0
C=0
Z=1

$$N=0, C=0 \text{ e } Z=1 \Rightarrow 10 = 10$$

Anche in questo caso -10 non può essere rappresentato con soli 4 bit e così, nell'operazione, ha luogo l'overflow. A tenere traccia della presenza o meno dell'overflow è il **flag OF**. Il flag OF può esser visto come il risultato dell'XOR applicato agli ultimi due bit di riporto dell'operazione.

Se OF = 0 e N = 1, allora non c'è overflow.

cmp -5, -6

$$\begin{array}{r} -6 \\ +5 \\ \hline ? \end{array} \rightarrow \text{In binario, con 4 bit}$$

OF = 0, N = 1  $\Rightarrow$  NO OVERFLOW

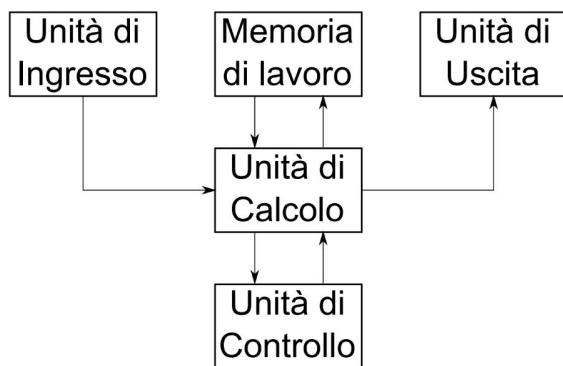
N = 1, C = 0 e Z = 0  $\Rightarrow$  -5 > -6

OF = 0  
0 0 0  
1 0 1 0  
- 0 1 0 1  
—————  
(0) 1 1 1 1  
N = 1      Z = 0  
C = 0

# Il processore z64

## Il modello di Von Neumann

Quello di Von Neumann è uno dei primi modelli di architettura di un elaboratore.



Questo elaboratore prevede una memoria di lavoro che gestisce i dati in input o i dati del programma; di un'unità di calcolo che svolge le operazioni; di un'unità di controllo che interpreta le istruzioni e definisce come devono essere eseguite.

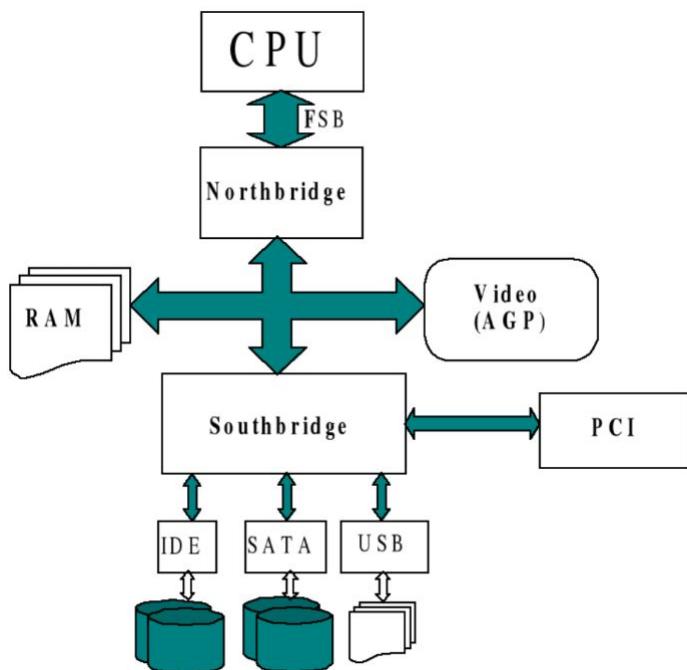
## Architettura Harvard

Un altro modello di architettura è quello di Harvard, utilizzato ancora oggi per l'implementazione della cache.



## Architettura IBM

I moderni calcolatori sfruttano una combinazione delle due architetture. L'architettura oggi considerata come standard è quella di IBM, la quale risolve una problematica comune a tutte le architetture.

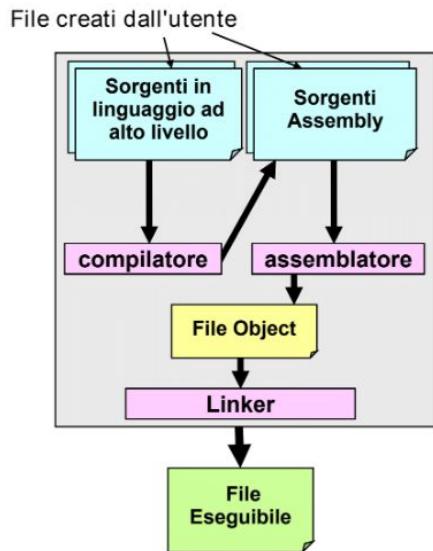


Un elaboratore è infatti costituito da dispositivi di velocità variabile. Se tutti i dispositivi fossero connessi in maniera diretta fra loro, quelli più veloci dovrebbero eseguire operazioni ad un ritmo più basso di quello teoricamente sostenibile. Immaginiamo di avere tante macchine in fila su una strada e che sia vietato il sorpasso. Le macchine in coda, siano esse Ferrari o Fiat Panda, dovranno attendere che a procedere sia la prima macchina della fila.

In un'architettura IBM il processore dialoga con un **northbridge**, un adattatore che gli consente di interfacciarsi con componenti più lente. In questo modo il processore continua ad elaborare alla sua velocità senza dover stare al passo con il dispositivo più lento.

Ogni qual volta che un dispositivo più lento fa qualcosa, i vari bridge comunicano con il processore avvisandolo che il dato *X*, richiesto ad esempio 300 colpi di clock prima, è ormai pronto.

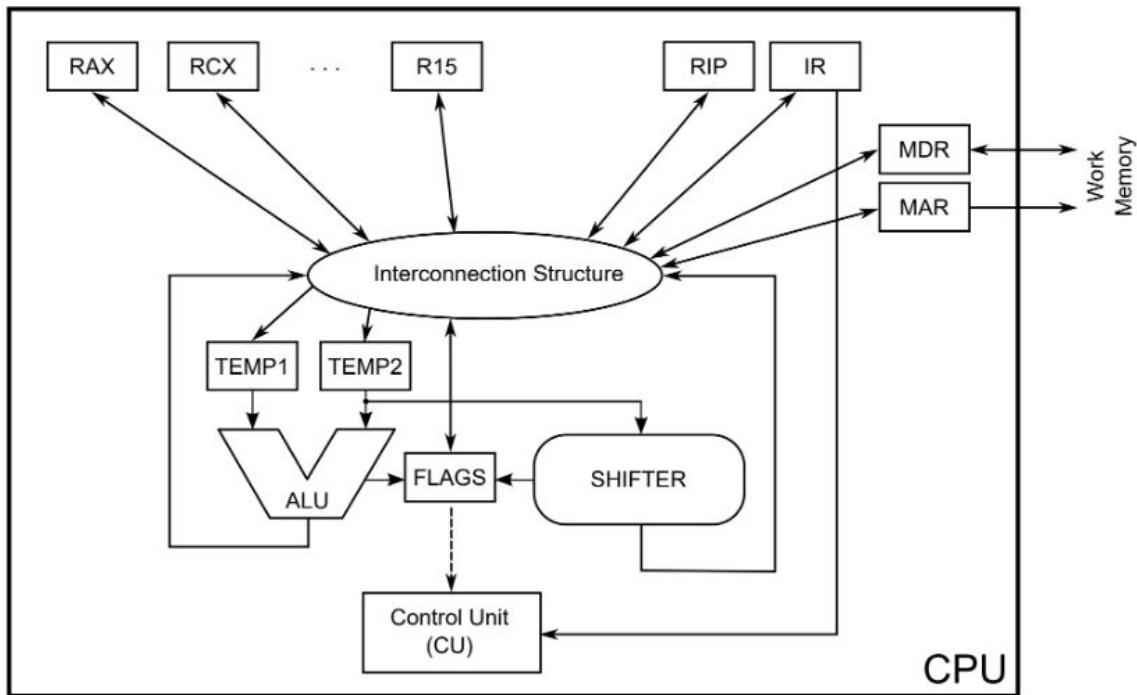
## Generazione di un programma



Durante la generazione di un programma l'utente può creare dei file in linguaggio ad alto livello o in Assembly. Nel primo caso spetterà al compilatore tradurre le righe di codice in sorgenti Assembly. A questo punto un assemblatore si occuperà di tradurre i file Assembly in file oggetto e da qui, attraverso dei linker, si produrrà il codice eseguibile che può essere computato dall'elaboratore.

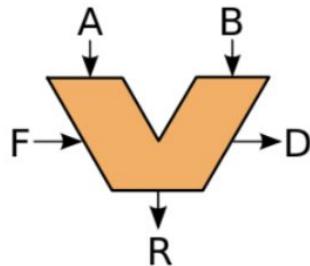
## Il processore z64

L'architettura di un processore z64 sarà quella adottata in questo corso. Tale architettura presenta una serie di moduli base e una serie di possibili moduli che possono essere connessi ad esso per le varie funzioni.



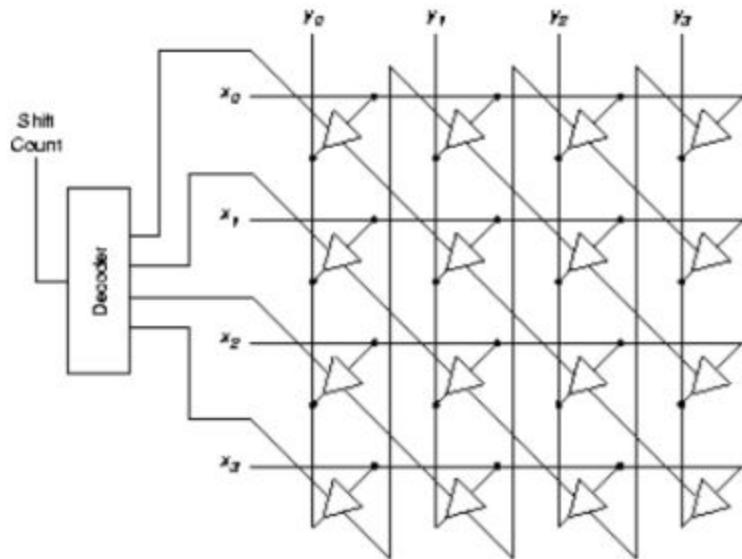
In particolare in questa rappresentazione vediamo l'**ALU**, la arithmetic logic unit che svolge le operazioni di calcolo, la **CU**, la control unit, che coordina le varie operazioni che i singoli componenti devono svolgere. Riconosciamo poi i registri dei flags, uno shifter (che vedremo più avanti) e i registri visibili e non visibili.

Per memorizzare dati in un processore si può adottare una soluzione molto semplice ma efficace, quella dei flip-flop, temporizzati con un clock. Sfruttando più flip flop si possono realizzare i **registri**, fondamentali per il lavoro di un processore.



A, B: operandi  
 F: segnali dalla CU  
 D: segnali di stato  
 R: risultato

L'**ALU** presenta tre entrate e due uscite. Un'entrata A e B da cui si prendono gli operandi ed F, ossia i segnali di abilitazione della Control Unit (OP CODE). Un'uscita D che consente di settare i parametri del processore in appositi registri ed un'uscita R che consente di restituire il risultato. Un processore che lavora a 64 bit avrà bisogno, per ogni operando di 64 fili e per il risultato di altri 64 fili. La struttura del processore è quindi piuttosto articolata e le schede su cui essi sono posti sono per niente banali.



Uno **shifter** è un circuito digitale che può traslare i bit di una parola di un numero specificato di posizioni in un solo colpo di clock.

Uno shifter fa uso di buffer three-state, abilitati da un apposito segnale di abilitazione.

L'implementazione qui raffigurata è diversa dallo shifter register visto in precedenza.

Lo shifter **aritmetico** tiene conto della rappresentazione in binario e quindi pone un 1 se i numeri sono negativi. Lo shifter logico esegue l'operazione senza tener conto di ciò.

---

## **z64: l'unità di processamento (PU)**

CONTINUARE

## Sistemi di reti LLC

Abbiamo fino ad ora visto come progettare reti LLC e macchine in grado di sfruttarle. La domanda a cui vogliamo ora trovare una risposta è la seguente: dati vari sistemi come faccio a farli lavorare insieme?

Una prima risposta è che tutti i sistemi connessi fra loro devono lavorare alla stessa frequenza di clock.

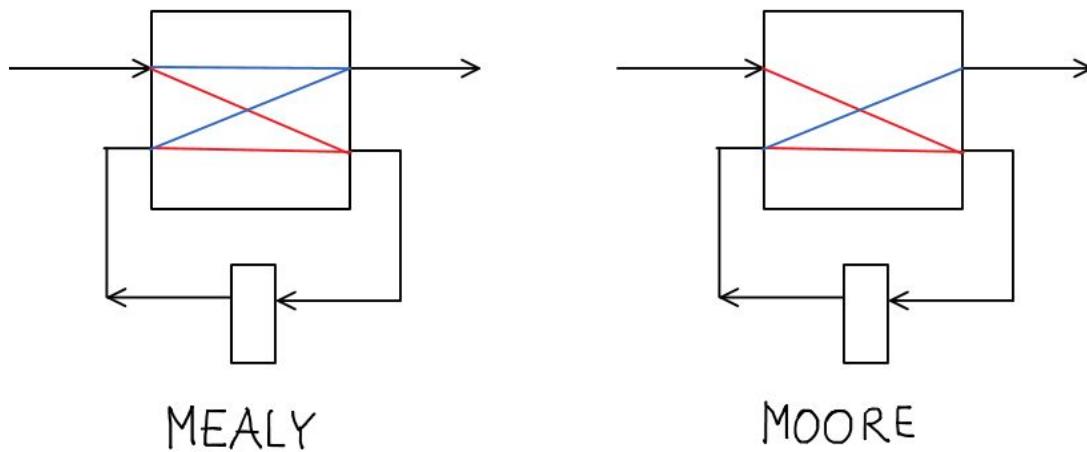
Qui di seguito verranno mostrati:

- catene aperte (o libere) con macchine di Moore e di Mealy
- catene chiuse con sole macchine di Moore, sole macchine di Mealy e con entrambe
- Pipeline
- reti per interconnessione qualunque

Tutte le considerazioni che verranno fatte dovranno tener conto della struttura delle macchine implementate e dell'importanza dei **periodi di funzionamento del clock**.

### Macchina di Mealy e di Moore: un rapido ripasso

Ricordiamo, brevemente, gli schemi delle macchine di Mealy e di Moore.

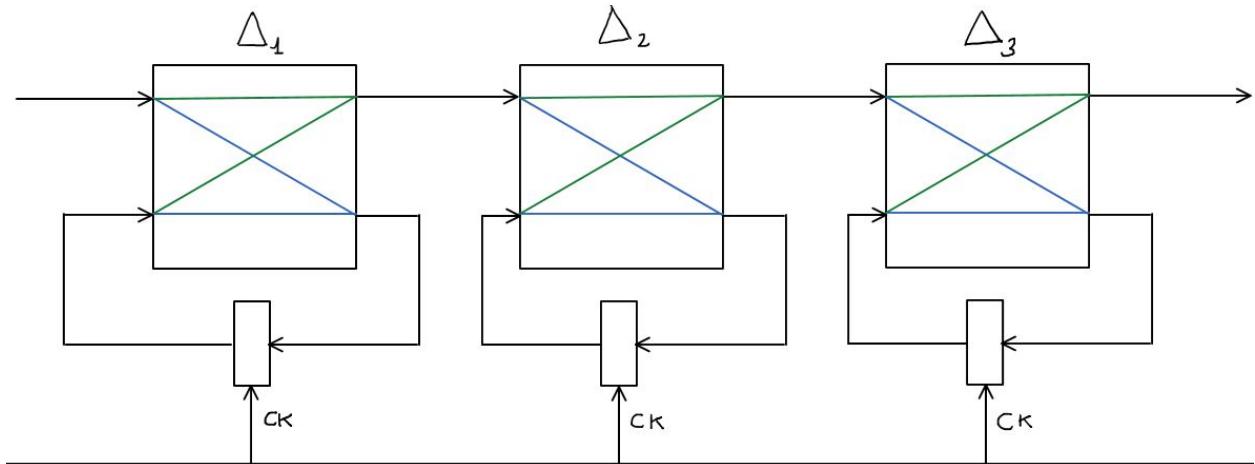


Questa rappresentazione mostra in maniera molto lampante la relazione tra ingresso-stato, ingresso-uscita e stato-uscita.

---

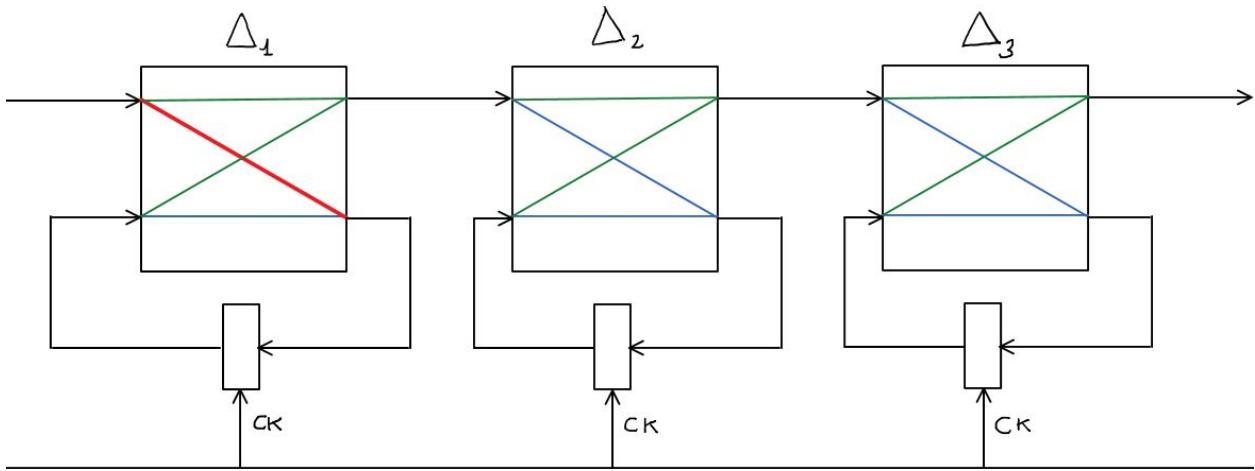
## Catena sequenziale libera di macchine di Mealy

Supponiamo di avere una semplice sequenza di macchine di Mealy.

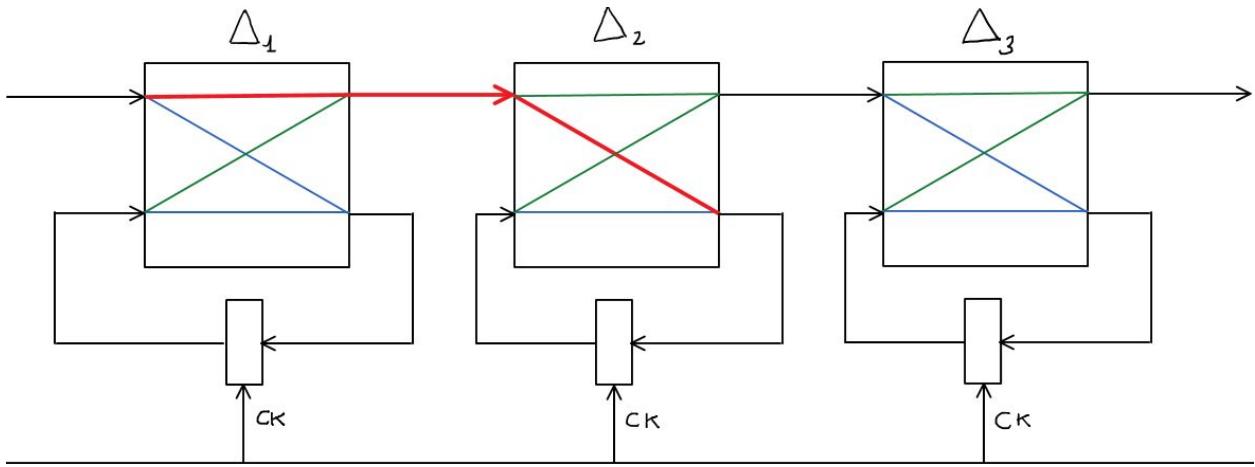


Usiamo lo stesso clock per tutte e tre le macchine e supponiamo i tempi di elaborazione di ogni rete siano  $\Delta_1$ ,  $\Delta_2$  e  $\Delta_3$ .

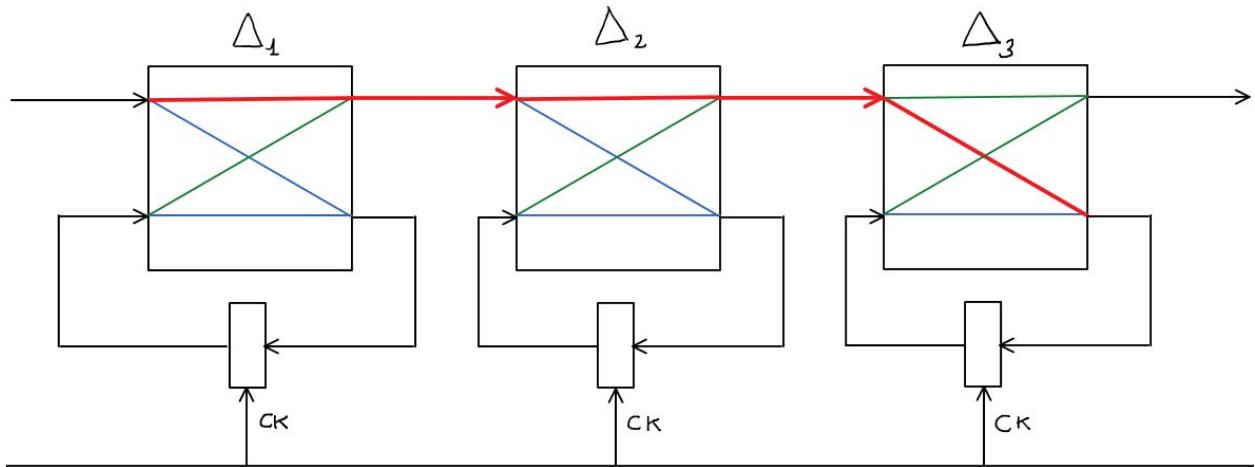
Vogliamo fare la fotografia dei vari ingressi. Per la prima macchina basterà che l'ingresso attraversi la rete e si dovrà quindi attendere un tempo  $\Delta_1$  prima di vedere degli effetti sullo stato.



Se ora consideriamo anche il secondo sistema possiamo vedere che lo stato verrà aggiornato (e quindi "fotografato") non appena l'informazione sarà passata per la prima macchina e per la seconda. Quindi bisognerà attendere un tempo  $\Delta_1 + \Delta_2$ .



Procedendo con la terza macchina si potrà “fotografare” l’ingresso dopo un tempo  $\Delta_1 + \Delta_2 + \Delta_3$ .



Notiamo quindi che ogni macchina dovrà attendere un tempo pari alla somma di tutti i tempi di commutazione delle macchine precedenti ed del suo prima di poter “fotografare” l’ingresso.

Il clock dovrà quindi avere un periodo dato dalla somma di  $\Delta_1 + \Delta_2 + \Delta_3$ .

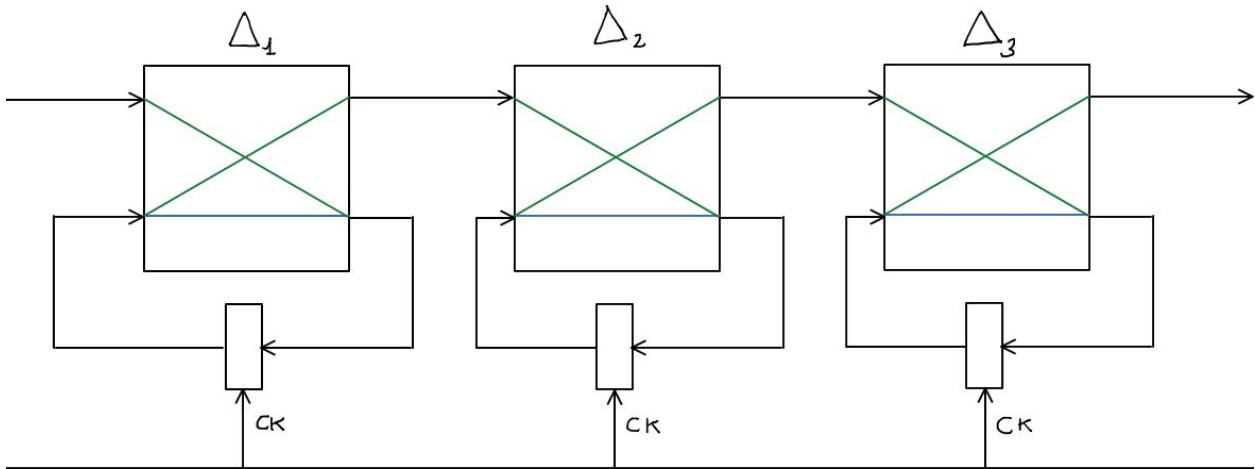
Se supponiamo che i tempi di commutazione siano  $\Delta_1 = 1\text{nsec}$ ,  $\Delta_2 = 10\text{nsec}$  e  $\Delta_3 = 100\text{nsec}$ , è evidente che il primo circuito potrebbe lavorare velocemente ma, essendo incluso in un circuito con componenti più lente, sono costretto a rallentarlo.

Il periodo del clock deve essere mutuato dalla componente più lenta (e quindi limitante).

---

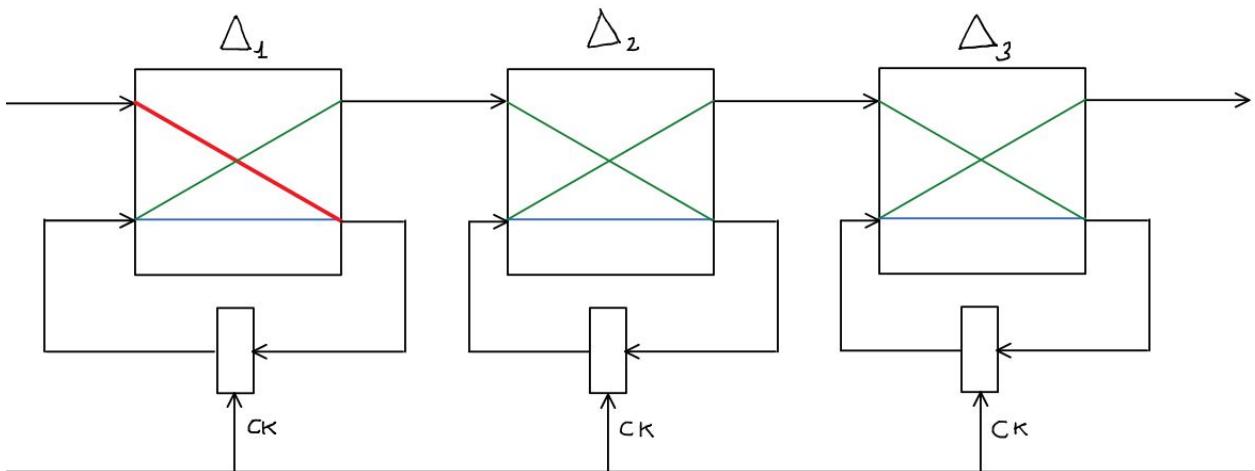
## Catena sequenziale libera di macchine di Moore

Ipotizziamo ora di avere macchine di Moore.

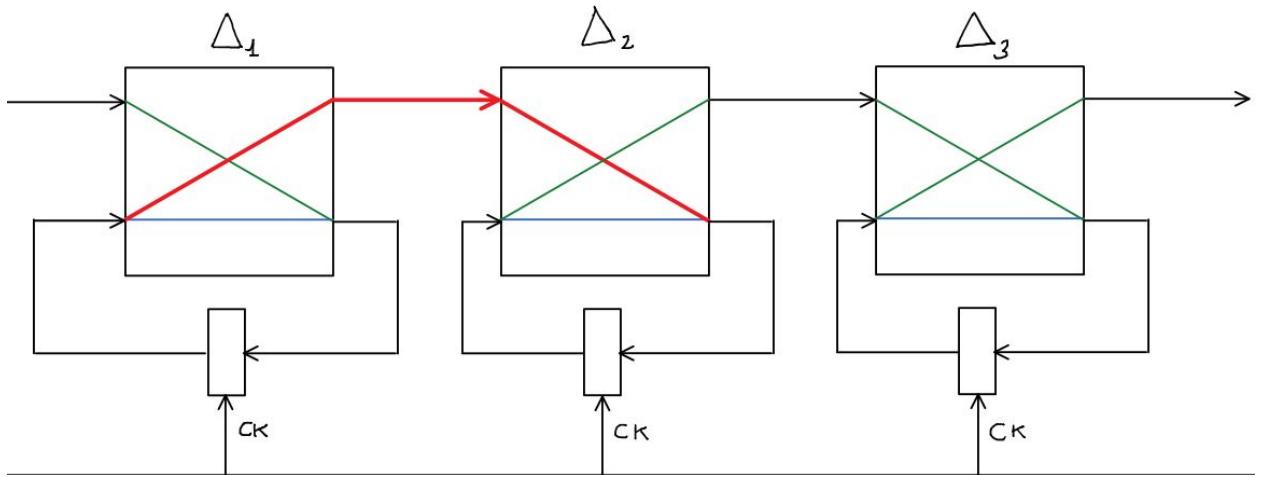


Usiamo lo stesso clock per tutte e tre le macchine e supponiamo i tempi di elaborazione di ogni rete siano  $\Delta_1$ ,  $\Delta_2$  e  $\Delta_3$ .

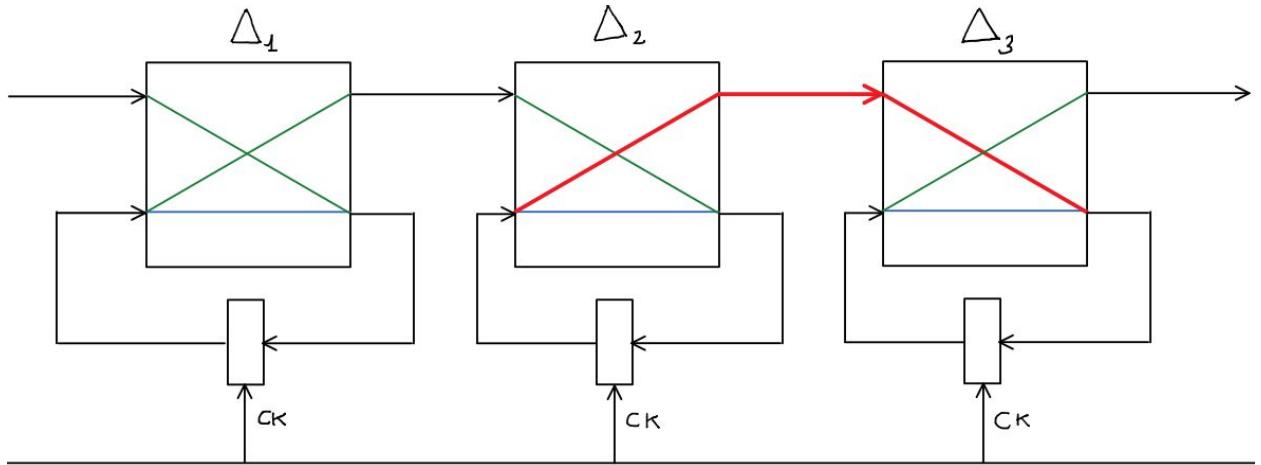
Vogliamo, come prima, fare la fotografia dei vari ingressi. Per la prima macchina basterà che l'ingresso attraversi la rete 1 e si dovrà quindi attendere un tempo  $\Delta_1$ .



Le cose cambiano leggermente quando si dovrà fare la stessa considerazione per la seconda macchina. Possiamo facilmente vedere che per "fotografare" l'ingresso questo deve essere prodotto dalla precedente macchina e deve attraversare la seconda macchina. Bisognerà dunque attendere un tempo  $\Delta_1 + \Delta_2$ .



Andando ancora avanti possiamo vedere che il tempo che dovremo attendere per la terza macchina è  $\Delta_2 + \Delta_3$ .



Nell'ipotesi di avere un quarto circuito, sempre con una macchina di Moore, avrei potuto scansionare l'ingresso dopo un tempo  $\Delta_3 + \Delta_4$ .

Supponendo che tutti i tempi di commutazione siano equivalenti, ossia che  $\Delta_i = \Delta$ , si potrà dire che il clock ha periodo  $\overline{\square \square} = 2\Delta$  nel caso di Moore, mentre nel caso di Mealy

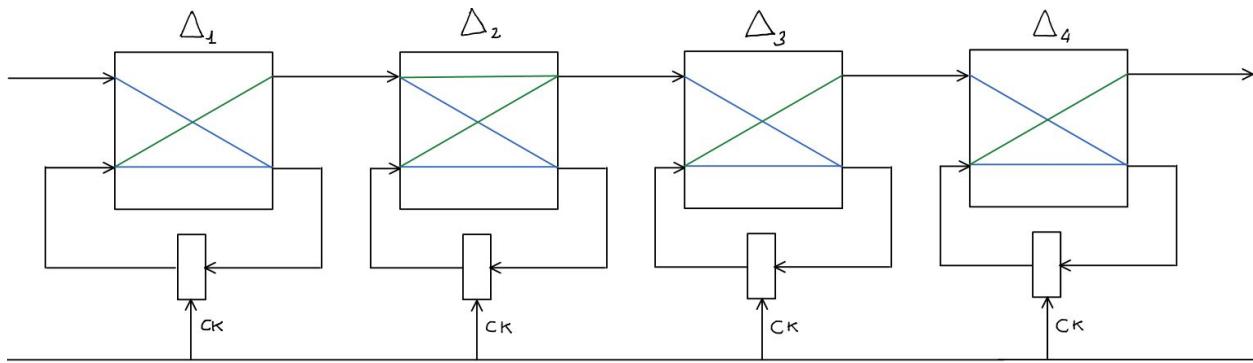
$$\overline{\square \square} = N\Delta.$$

Da queste considerazioni si potrebbe concludere che, essendo il periodo di clock inferiore nel caso di Moore, questa è la configurazione migliore. Si deve tuttavia tenere ben in mente

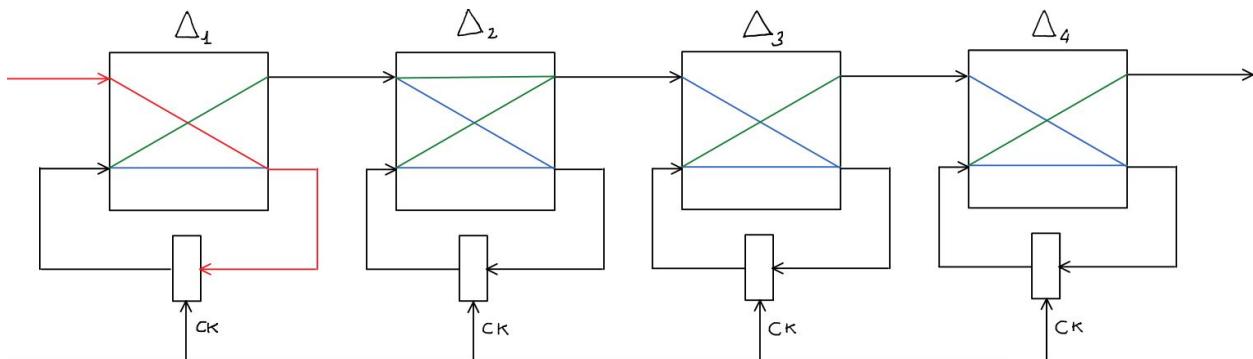
che il tempo di trasmissione dell'ingresso nell'uscita è pari a  $2\Delta$  nel caso di Moore (e quindi superiore rispetto a Mealy).

### Catena sequenziale di combinazioni di macchine di Moore e Mealy

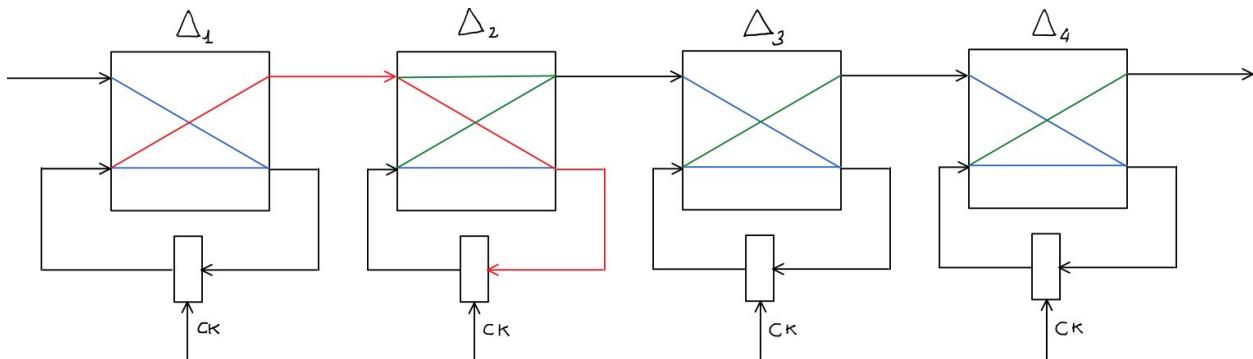
Procediamo ora con alcuni esempi in cui Mealy e Moore risultano essere combinati. Nella seguente configurazione abbiamo quattro macchine, di cui una di Mealy e le altre di Moore.



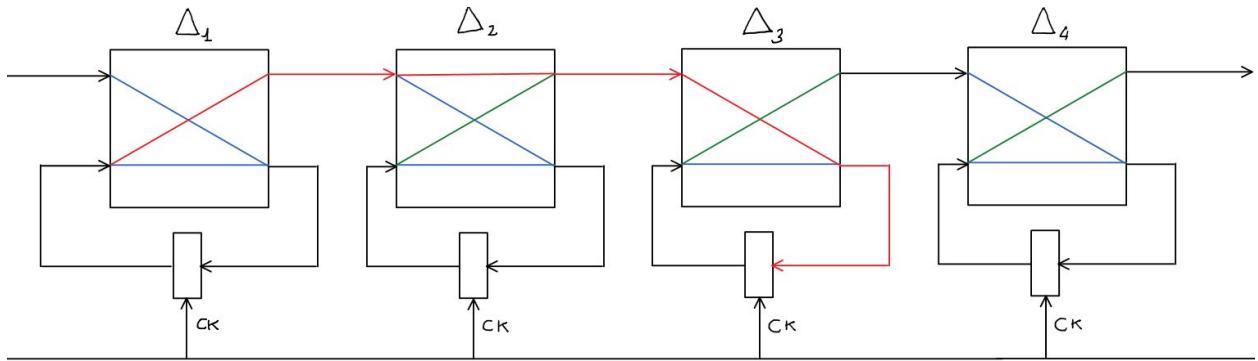
Partiamo con la prima macchina. L'ingresso entra e, dopo il tempo di commutazione  $\Delta_1$ , può essere fotografato.



L'ingresso della seconda macchina, prima di giungere ad essa, dovrà transitare per la macchina 1. Poi, una volta entrato potrà essere fotografato dopo il suo tempo di commutazione. Quindi bisognerà attendere un tempo  $\Delta_1 + \Delta_2$ .

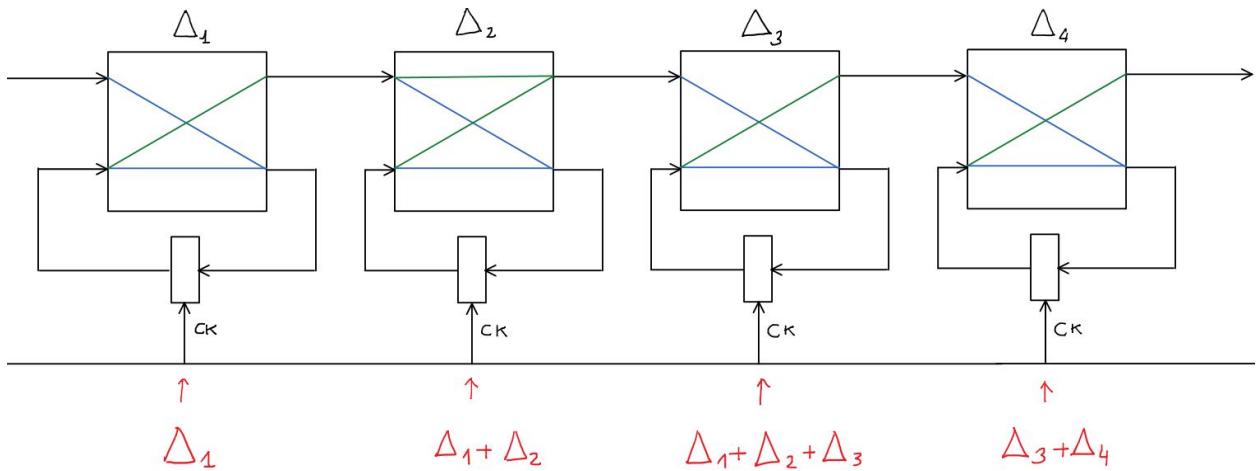


Per la macchina 3 l'ingresso dovrà prima attraversare la macchina 1 e la 2 e poi la 3.

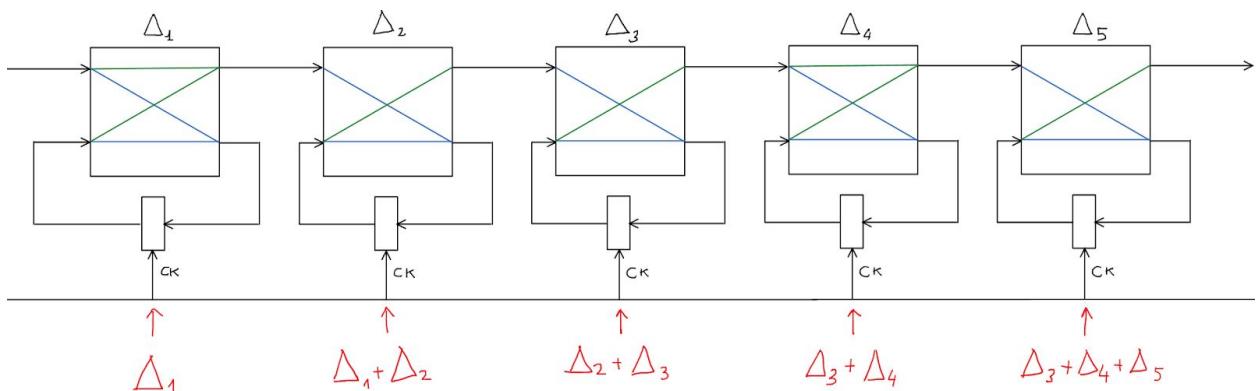


Quindi bisognerà attendere un tempo  $\Delta_1 + \Delta_2 + \Delta_3$ . Per la quarta macchina potranno essere effettuate delle considerazioni simili alla macchina 2.

Riassumendo, si riportano tutti i tempi sotto ciascuna macchina



Qui di seguito un ulteriore esempio. Sono riportati i tempi sotto ciascuna macchina (senza passaggi specifici).



---

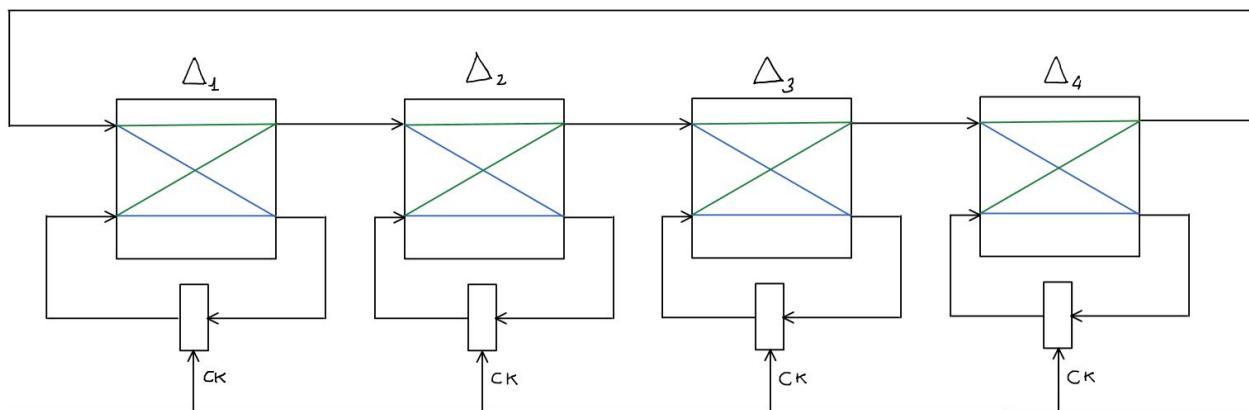
## Alcune considerazioni

Analizzando gli schemi precedenti è interessante vedere come la catena cominci o da un ingresso o da un registro di un circuito che implementa la macchina di Moore.

Se ho  $N$  macchine sequenziali risulta, alla luce di questa assunzione, più facile dimensionare; perché dovrò partire dalle macchine di Moore, lasciando le Mealy che sicuramente saranno intermedie. Detto in parole povere, questo vuol dire che per determinare il periodo di clock basterà prendere in considerazione la sequenza di macchine di Mealy più lunga fra tutte quelle presenti ed

## Catena chiusa con macchine di Mealy

Procediamo con un ciclo che sfrutta macchine di Mealy

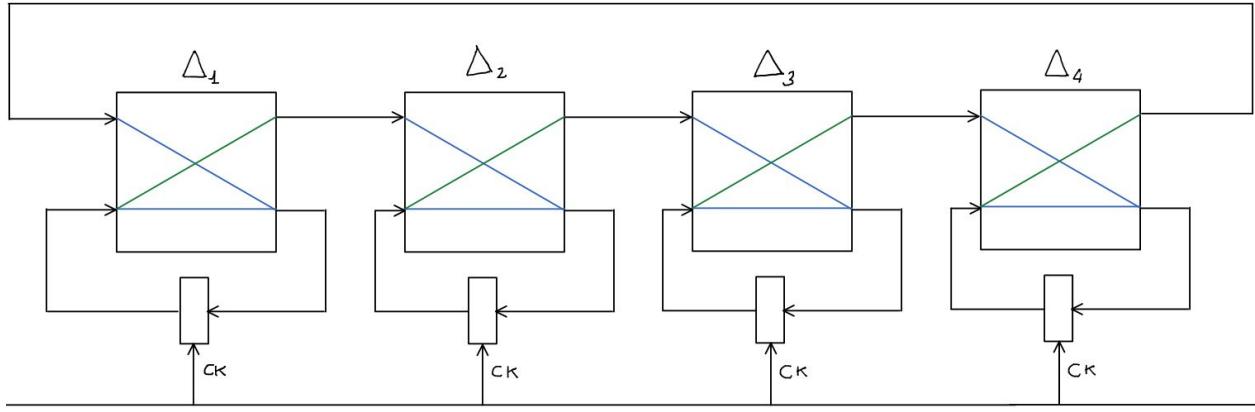


Questa catena chiusa non può essere realizzata perché, col procedere dei cicli, risulterebbe instabile. Mano a mano infatti crescerebbero i tempi di attesa prima di poter "fotografare" l'ingresso, rendendo insostenibile la situazione.

## Catena chiusa con macchine di Moore

La situazione è diversa nel caso di cicli realizzati con la macchina di Moore.

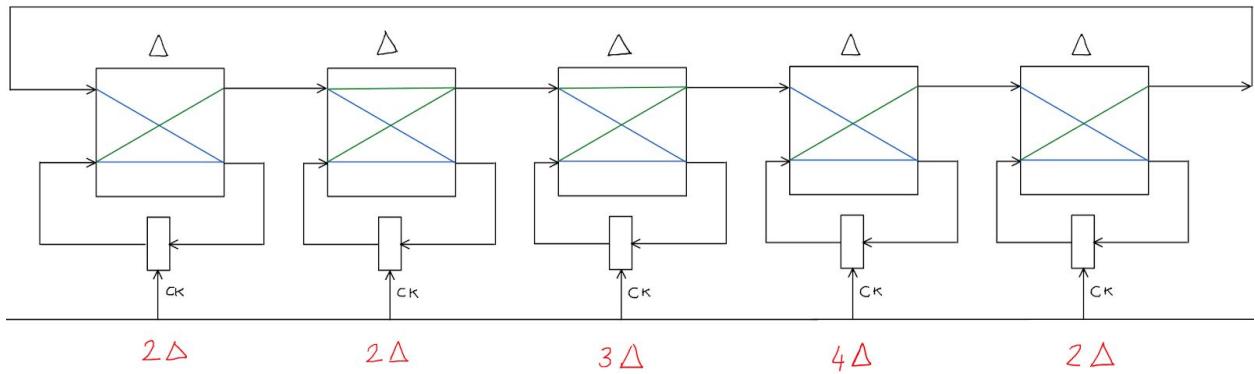
In questo caso si avrà come periodo massimo di clock  $2\Delta$  (supposti uguali tutti i tempi di commutazione), essendo infatti ogni ingresso generato derivato dal flip-flop D della macchina precedente. Esso quindi dovrà attraversare solo due macchine di Moore.



L'avere una situazione completamente antitetica a quella prima avuta con sole macchine di Mealy suggerisce che per risolvere il problema dell'instabilità prima descritta basterà far sì che in un ciclo ci sia almeno una macchina di Moore.

### Ciclo con combinazioni di Mealy e Moore

Ipotizziamo di avere una combinazione di macchine di Mealy e di Moore e che tutte abbiano stesso tempo di commutazione  $\Delta$ .



Sotto ogni macchina è presente il tempo che deve essere atteso prima di poter "fotografare" l'ingresso. Dovrà essere preso il tempo tempo maggiore, e quindi  $4\Delta$ .

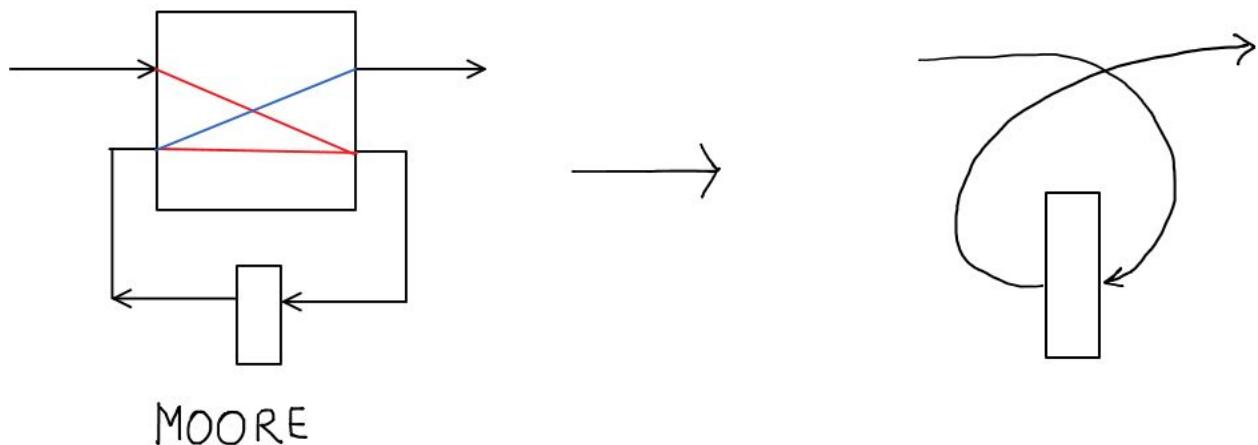
---

Per determinare il periodo possiamo dire che basterà contare quante macchine di Mealy sono incluse fra due macchine di Moore. È proprio il susseguirsi di macchine di Mealy che fa aumentare i tempi richiesti.

### Stabilizzare ciclo macchine di Mealy

Come detto, in presenza di cicli con sole macchine di Mealy si ha una situazione di instabilità, risolvibile inserendo almeno una macchina di Moore.

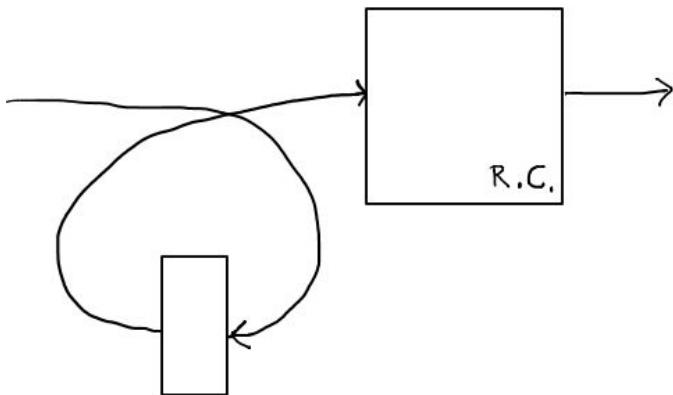
In realtà basterebbe un **registro** (realizzato con flip-flop D) per realizzare la più semplice macchina di Moore esprimibile.



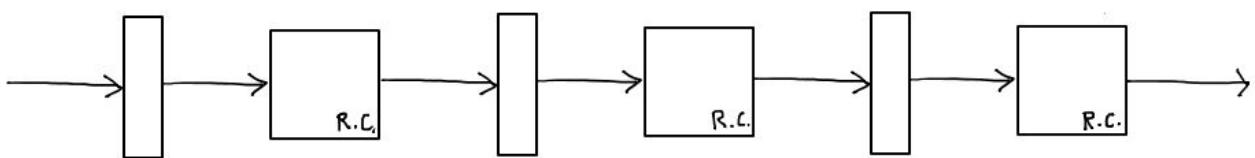
Posso infatti vedere la rete come un semplice filo che va e viene dal registro. Mettendo un registro che interrompe la catena si rende l'intero sistema stabile.

### Pipeline

Riprendiamo una catena di sole macchine di Moore. Immaginiamo di non avere porte logiche cui viene sottoposto l'ingresso e che questo vada diretto nel registro. Mi ritroverei di fronte ad uno schema del genere



Se ora si andassero a mettere in sequenza elementi di questo tipo si otterrebbe la seguente struttura



Questa è la struttura **pipeline**.

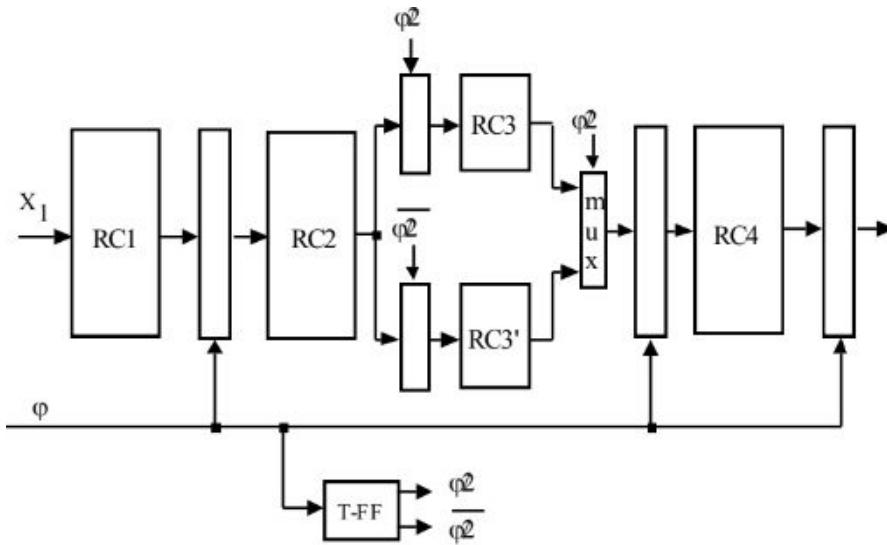
I processori RISC (Reduced Instruction Set Computer) sfruttano la struttura Pipeline. Essa non migliora il tempo di esecuzione di ogni singola istruzione, ma migliora il tempo di esecuzione dell'intero processo. Aumenta quindi il **throughput**, ossia il numero di operazioni eseguibili per unità di tempo. Ciò è ottenuto grazie alla riduzione del periodo di clock.

### Architetture Pipeline parallele

Si ipotizzi che la prima rete combinatoria del disegno precedente impieghi 1ms, la seconda 2ms e la terza 1ms. Il periodo di funzionamento di questa pipeline è 2ms, dunque l'intero sistema è limitato dalla componente più lente.

Possiamo pensare di mettere in parallelo la seconda rete con un'altra, in modo da suddividere il “lavoro” e quindi ridurre il periodo di funzionamento ad 1ms.

Questa soluzione è impiegata per le architetture Pipeline parallele.



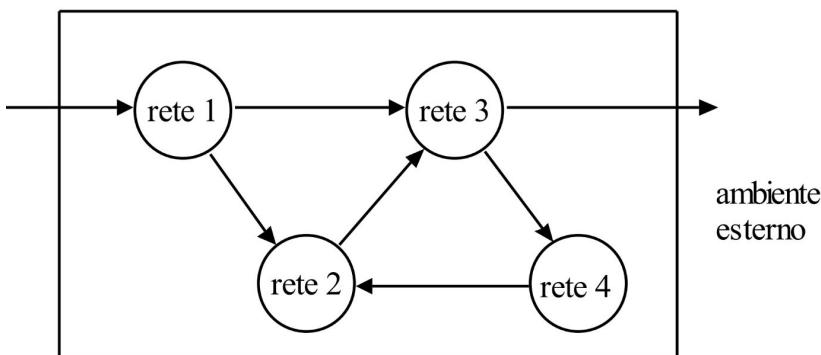
Per permettere questa soluzione si fa uso di due registri ed un multiplexer.

### Reti di sistemi per interconnessione

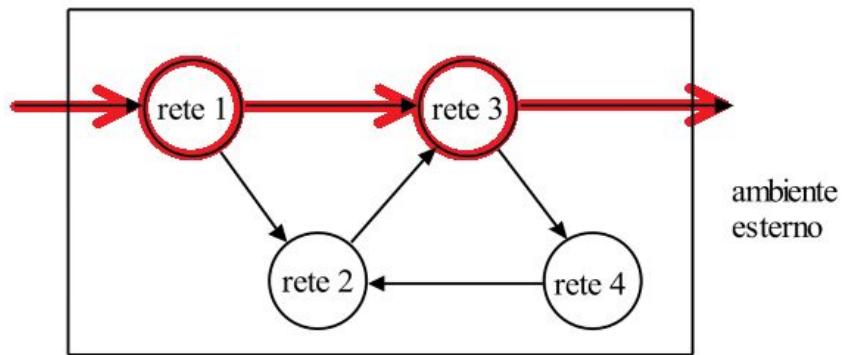
#### qualsiasi

In precedenza si è parlato di reti sequenziali a catena libera e di reti sequenziali a catena chiusa. Ora possono essere analizzate le reti di sistemi per interconnessione qualsiasi, di cui quello mostrato di seguito ne è un esempio.

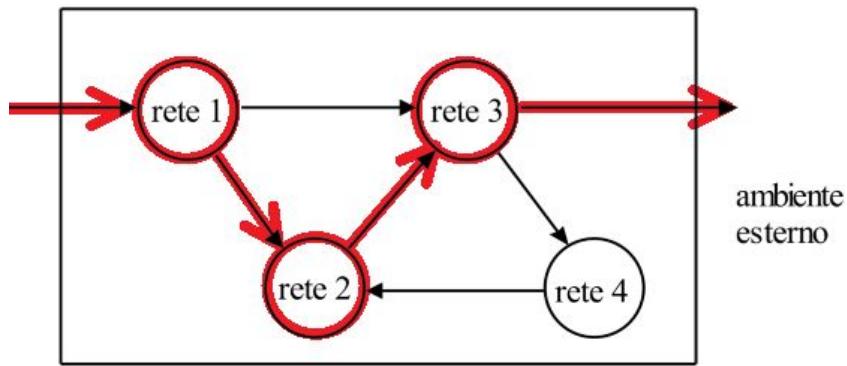
La domanda cui si troverà risposta è come fare a dimensionare i periodi di funzionamento del clock in queste situazioni.



Prima di analizzare questo insieme di reti nel complesso procediamo con l'analizzarlo in porzioni. Prendendo in considerazione solo le reti "in alto", ossia rete 1 e rete 3, riconosciamo facilmente una rete sequenziale libera (aperta). In questo caso, essendo due le reti, a prescindere dal tipo di macchine utilizzate per implementare le due reti, il periodo minimo di clock sarà  $2\Delta$ .

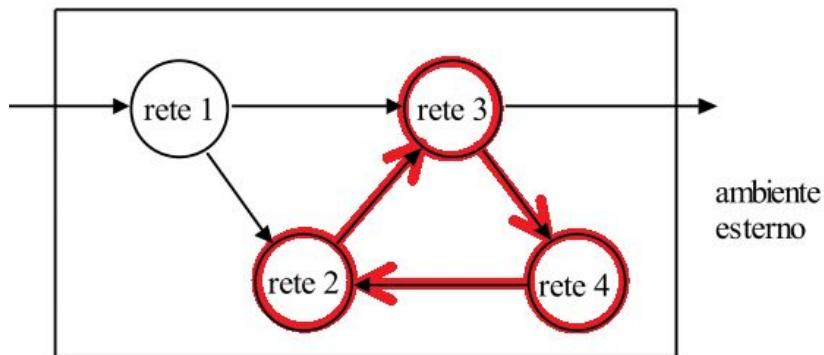


Passiamo ora con l'analizzare una seconda possibile rete sequenziale libera, quella formata a partire dalle reti 1,2,3.



Il periodo minimo di funzionamento del clock sarà  $2\Delta$  e, nel peggior dei casi,  $3\Delta$  (ossia nel caso in cui siano tutte macchine di Mealy).

Possiamo poi individuare anche un ciclo.



---

La presenza di un ciclo mi implicherà che le reti 2, 3, 4 non possano essere tutte macchine di Mealy (perché si renderebbe l'intero sistema instabile).

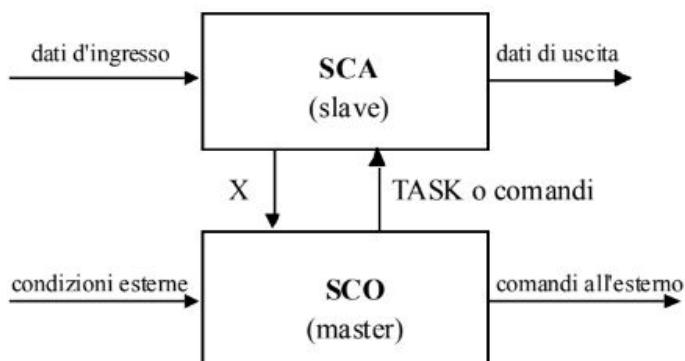
Da ciascuna di queste 3 possibili catene potrà ottenere dei tempi di funzionamento. Per essere sicuro che il sistema funzioni correttamente basterà mettersi su quello maggiore.

Quindi, ricapitolando, in questi casi si dovranno analizzare tutte le catene presenti - siano esse libere o chiuse- e, per ciascuna, si dovranno determinare i periodi di clock. Il periodo di funzionamento del clock sarà determinato dal maggiore dei periodi delle singole catene.

La procedura può essere iterata su "agglomerati" di reti delle più disparate dimensioni.

## Sistemi Digitali Connessi

Quando si progetta un programma si lavora con una logica atta a modificare, spostare e manipolare i dati. La logica di manipolazione (ossia ciò che decide cosa fare) è data dal **sottosistema di controllo** SCO. I dati vengono invece manipolati dal sottosistema di calcolo SCA.



**Sistema digitale complesso suddiviso in SCO-SCA**

Qualunque sistema digitale sarà quindi organizzato in un SCA e un SCO.

Il **SCO** può essere visto come un automa a stati finiti e quindi come un circuito sequenziale che può essere implementato in logica cablata (rete combinatoria) o tramite una ROM (per la microprogrammazione).

---

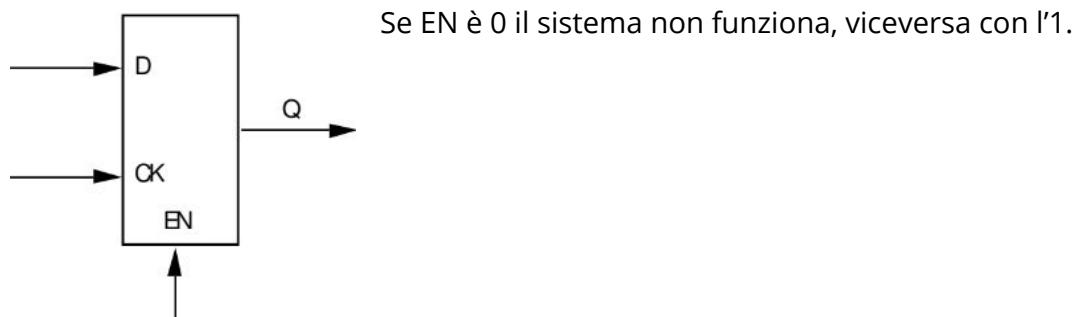
Il **SCA** sarà realizzato da addizionatori, shifter, bus, registri, eventualmente memorie e quindi da tutte quelle entità che vengono governate dal SCO.

Sia SCO che SCA possono ricevere dati dall'ambiente ed inviarli verso l'ambiente.

## Il sottosistema di calcolo (SCA)

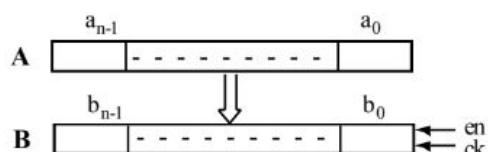
Prima di parlare del sottosistema di calcolo e delle sue varie implementazioni è utile ricordare alcuni concetti di base.

I registri servono per archiviare informazioni, sono stati da noi implementati con flip-flop di tipo D, capaci di presentare in uscita ciò che c'è in entrata al colpo di clock. Può essere anche presente un segnale abilitativo EN, che appunto abilita il registro.



Il trasferimento dati da un registro e l'altro avviene accostando gli N flip-flop che costituiscono un registro agli N flip-flop che costituiscono il secondo. Così facendo l'uscita di un registro costituisce l'ingresso dell'altro.

Questa operazione non apporta modifiche al contenuto del primo registro.

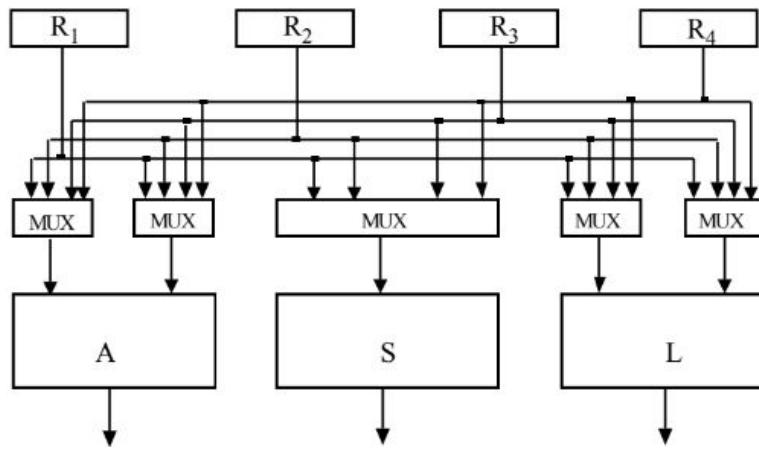


Trasferimento dati tra due registri

### Interconnessione registri-circuiti di calcolo

Giunti a questo punto si vorrà poter mettere in comunicazione i singoli registri con i circuiti di calcolo, come addizionatori e shifter.

L'implementazione più semplice possibile fa uso di multiplexer che selezionano il registro da cui prendere l'informazione.

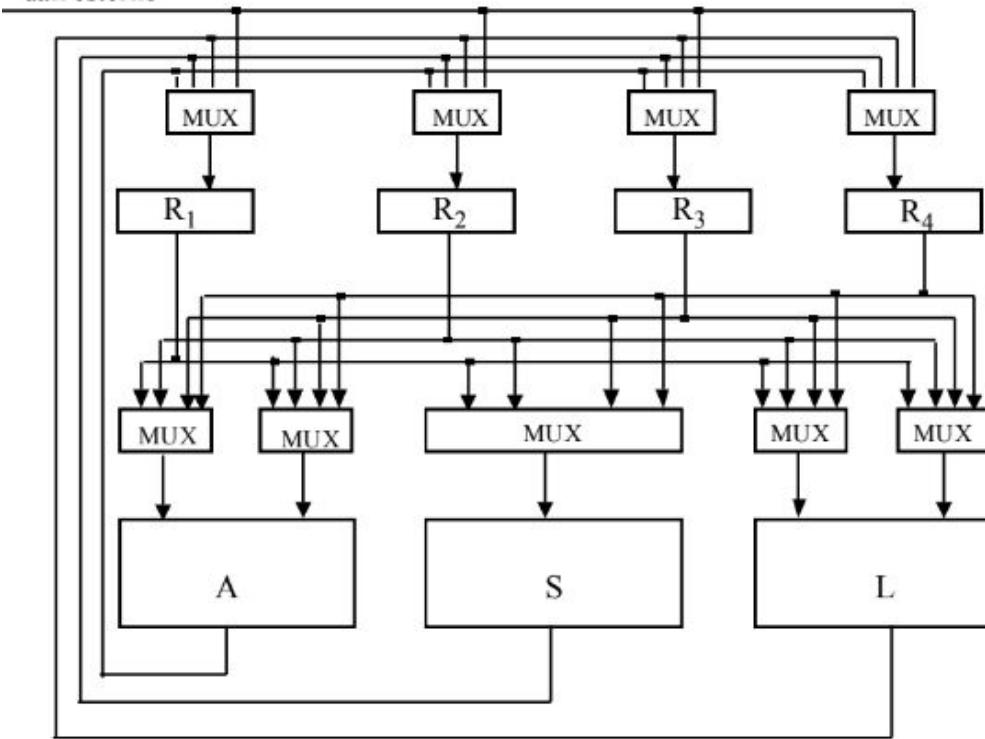


Nell'immagine qui sopra si hanno quattro registri che devono essere messi in comunicazione con tre circuiti combinatori. Ciascuno avrà un numero variabile di operandi (due per A ed L, uno per S).

Un certo numero di fili (64 se si lavora con 64 bit) partirà da ogni registro e giungerà da ciascun multiplexer. Il multiplexer interessato all'operazione da eseguire seleziona poi l'informazione da far passare.

L'uscita prodotta dai circuiti combinatori non dovrà essere persa, ma dovrà tornare nei registri. La soluzione più immediata è la seguente.

dall'esterno



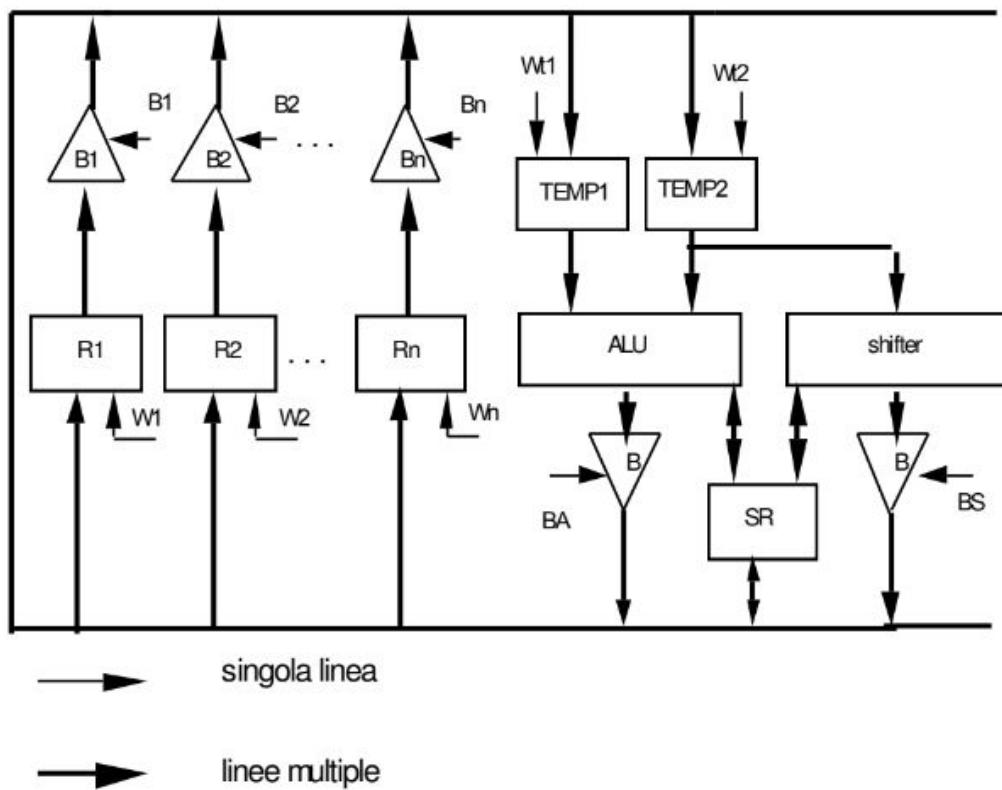
Il **vantaggio** di questa soluzione è che più di un'operazione può essere eseguita in **parallelo**, purché non ci sia conflitto sui dati.

Lo **svantaggio** risiede nella quantità di collegamenti richiesti. Ipotizzando che si lavori a 64 bit, serviranno 64 fili per ogni registro, ma anche 64 fili per ogni circuito combinatorio. Ciò aumenta in maniera sostanziale la **complessità** realizzativa e, dunque, anche i **costi**.

Quindi questa soluzione, che è molto veloce, costa un patrimonio.

### Interconnessione registri-circuiti di calcolo tramite bus

Una **soluzione alternativa** è quella che utilizza il **BUS**.



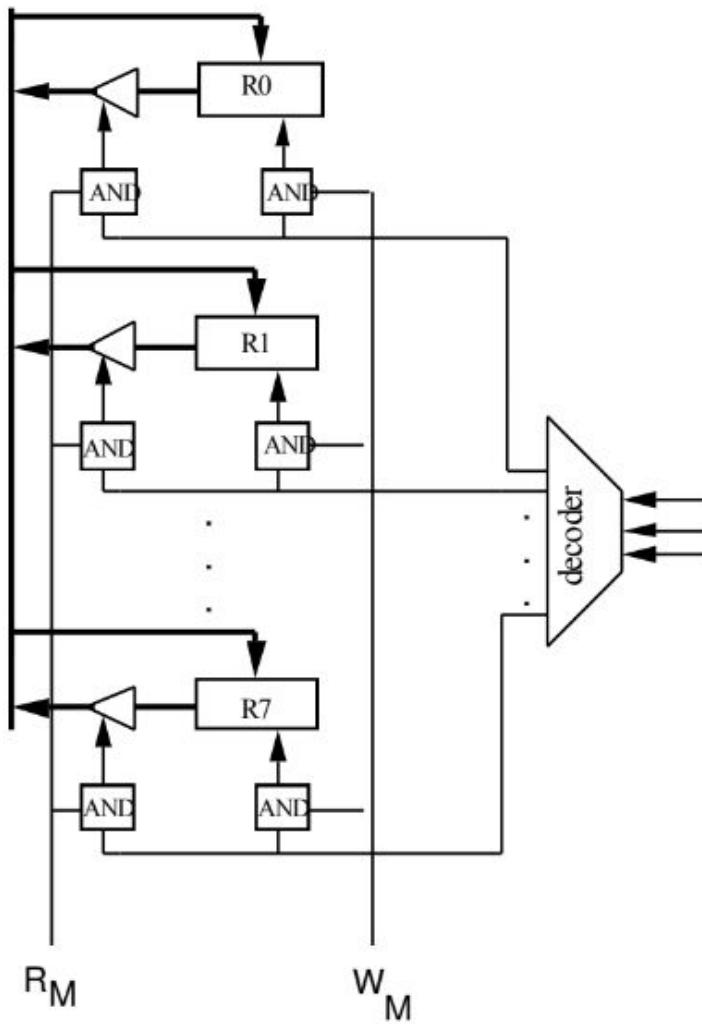
Se si lavora su 64 bit sarà sufficiente avere 64 fili per il bus, usato per trasmettere l'informazione. Naturalmente questo impone limiti più stringenti perché sul bus potrà viaggiare un'informazione alla volta. Per gestire i turni di comunicazioni si utilizzeranno dei buffer three-state.

Nella rappresentazione precedente è presente l'SR, lo **status register**, ossia l'insieme di registri che rappresentano lo stato del sistema. Tra i vari flag ricordiamo il flag Zero, il flag Overflow, il flag Carry.

Supponiamo di dover sommare il contenuto di R1 e di R2 e di mettere la somma in Rn. Il sottosistema di controllo dovrà abilitare B1 e contemporaneamente Wt1, poi B2 e Wt2. La ALU farà la somma e, dopo un po' di tempo il SCO dovrà attivare BA e Wn.

Il vantaggio di questa soluzione risiede nella semplicità realizzativa, lo svantaggio invece sta nei numerosi fili che comunicano con il SCO. Come detto infatti Wt1...Wtn, W1...Wn, B1...Bn rappresentano segnali di abilitazioni di lettura/scrittura e sono quindi implementati come fili che collegano le singole componenti al Sottosistema di Controllo.

### Organizzazione vettoriale dei registri



Per ridurre il numero di fili in comunicazione con il SCO si può adottare una soluzione di questo tipo.

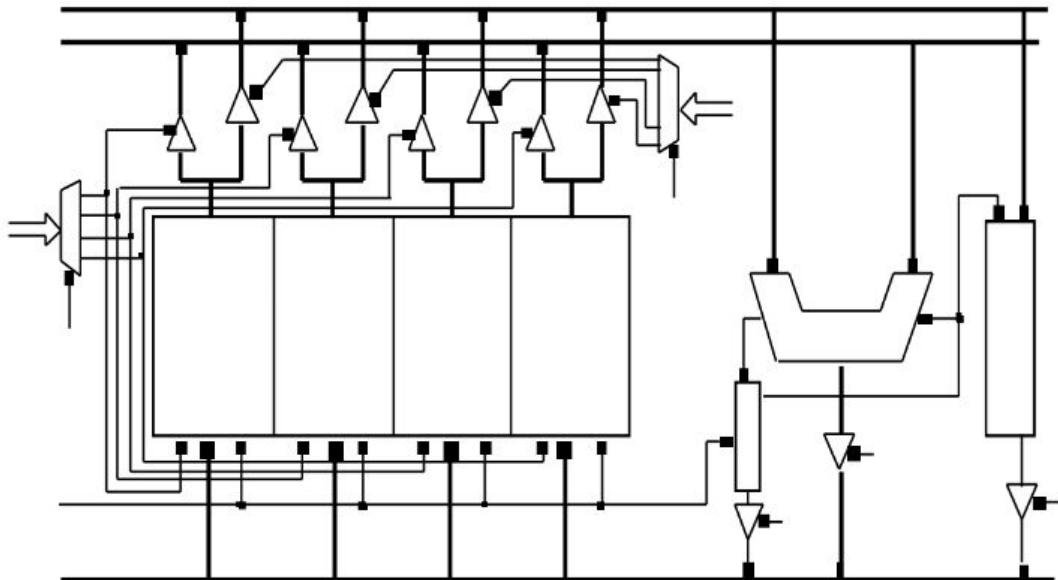
Si utilizza un decoder per abilitare un particolare registro e due fili, uno per la lettura e l'altro per la scrittura.

Se si vorrà ad esempio scrivere sul registro R1 basterà abilitare il registro con il decoder ed abilitare il segnale di scrittura. In corrispondenza del circuito R1 la porta AND farà sì che solo lì risulti un 1 per la scrittura.

---

## Soluzioni intermedie

Per velocizzare la trasmissione dei dati si potrebbe optare per un maggior numero di Bus, uno per ogni operando e uno per l'uscita dei circuiti combinatori.



La soluzione qui sopra sfrutta tre buffer, si avranno quindi  $64 * 3$  fili. Il vantaggio è nelle prestazioni maggiori, ma, di nuovo, si ha un aumento del cablaggio.

---

## Il sottosistema di controllo (SCO)

Il SCO è una macchina sequenziale che può essere implementata

- come **logica cablata**, comportando però qualche problema di progettazione ogni qual volta si deve modificare qualcosa
- con una **ROM** (o PROM), consentendo l'uso della microprogrammazione. La flessibilità garantita da una ROM sussiste a discapito dei tempi impiegati.

Il SCO è una macchina a stati finiti in cui ci sono un ingresso, un'uscita e delle variabili di stato, rappresentabile tramite una macchina di Moore o di Mealy. Solitamente noi utilizziamo la Moore perché l'uscita è stabile per tutto il tempo fintanto che non cambia lo stato, consentendo, in fase di progettazione, di diminuire i transitori. Vedremo tutte e due le soluzioni.

---

### Alcune considerazioni aggiuntive

Si può adottare anche una **EPROM**, ossia una ROM che, come già affrontato in precedenza, può essere riprogrammata più volte. Questa possibilità può essere sfruttata dai malintenzionati per far riprogrammare la ROM e quindi far "comportare" l'elaboratore in modo indesiderato.

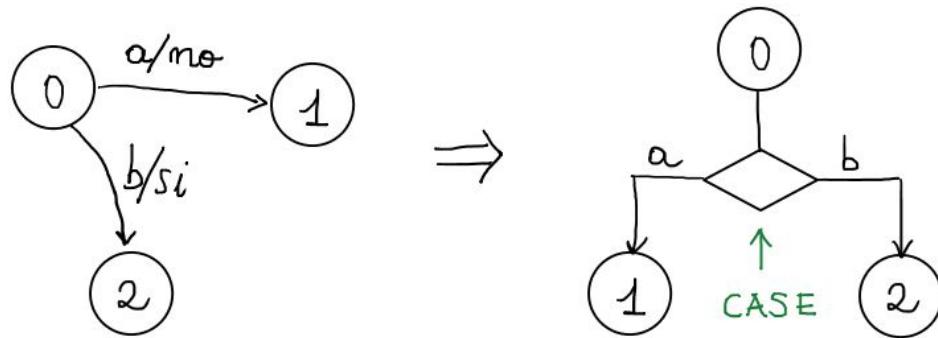
Altro importante elemento è rappresentato dalla **programmazione speculativa**. Oggigiorno i processori sfruttano infatti particolari tecniche che consentono di computare in maniera preventiva e speculativa le informazioni, anche quando queste non siano strettamente necessarie. Immaginiamo di avere una struttura if-else. Nell'attesa di capire se la condizione dell'if sia rispettata (magari perché dipende dall'input di un utente) il processore elabora le informazioni di entrambi i casi in maniera preventiva.

---

### Modello di Mealy per il SCO

Iniziamo ad analizzare il caso di una macchina di Mealy.

Innanzitutto, un diagramma degli stati può essere riscritto con una serie di **case**.



In funzione dell'ingresso ( $a$  o  $b$ ) si giunge ad un particolare stato (1 o 2) e si produce una **task**, ossia un'uscita (*si* o *no*).

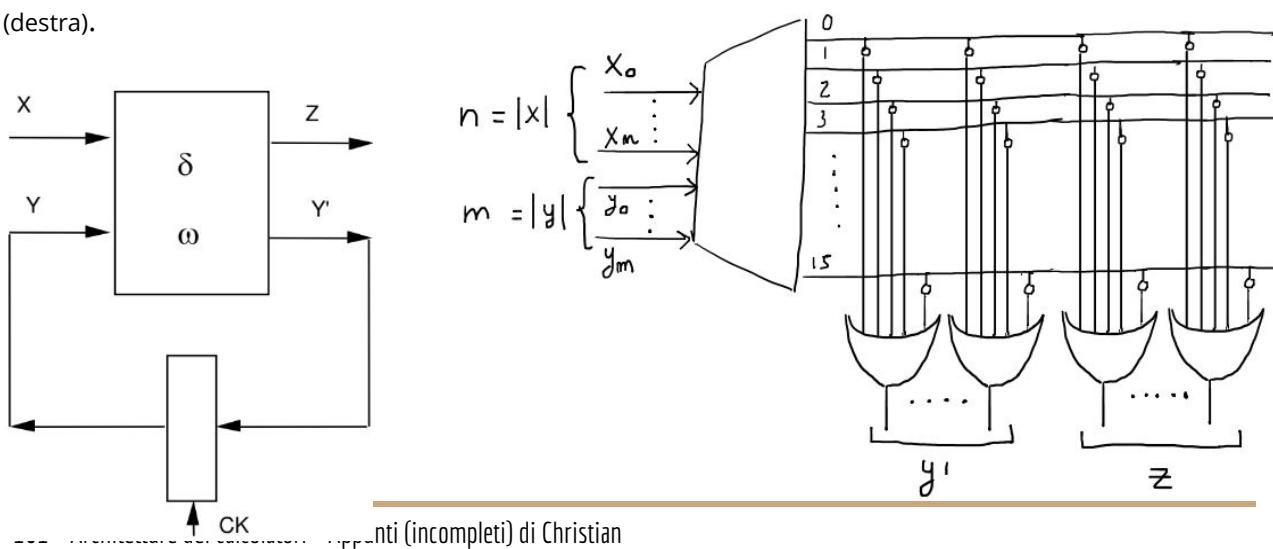
Da un punto di vista sintattico tutto ciò potrà essere scritto come microistruzioni. Una microistruzione ha un formato del tipo

$$\mu_i : C_1(T_1, \mu_{i1}), C_2(T_2, \mu_{i2}), \dots, C_m(T_m, \mu_{im})$$

dove:

- $C_1, C_2, \dots, C_m$  ( $m \leq 2k$ ,  $k = n$ ) sono le condizioni derivanti dalle variabili di decisione  $x_1, x_2, \dots, x_n$ ;
- $T_1, T_2, \dots, T_m$  sono le corrispondenti azioni da effettuare (le Task, ossia le uscite);
- $\mu_{i1}, \mu_{i2}, \dots, \mu_{im}$  le microistruzioni successive a  $\mu_i$ .

Il modello strutturale standard della macchina di Mealy è il seguente (sinistra). Si hanno ingresso, uscita (le task) e lo stato. Come detto, può essere sfruttata anche una ROM (destra).

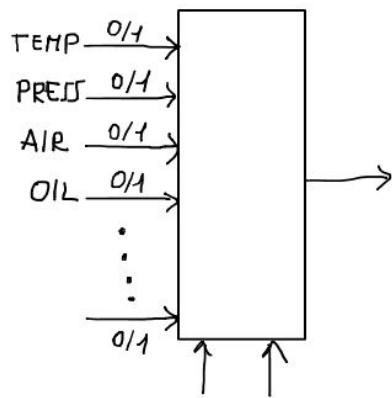


... (incompleti) di Christian

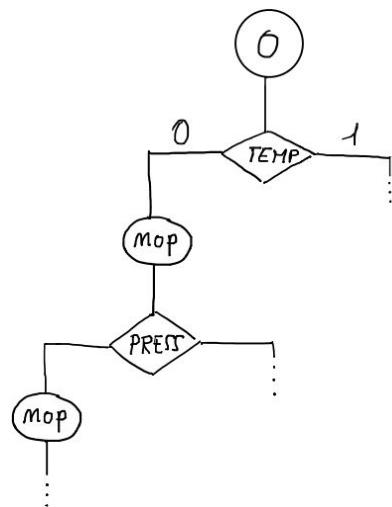
La ROM sarà data un certo numero di  $n$  variabili di condizione ( $2^n$  in uscita dal decoder) e da  $m$  variabili di stato ( $2^m$  in uscita dal decoder). L'uscita della ROM è data dallo stato successivo ( $y^1$ ) e dall'uscita Z (la Task).

In ingresso avrò quindi  $n+m$  righe, mentre in uscita ben  $2^{n+m}$  righe. È evidente che questo valore esponenziale può essere potenzialmente molto alto.

Supponiamo di avere un'auto con  $n=100$  ingressi (ossia sensori vari), avrei bisogno di almeno  $2^{100}$  fili per implementare la ROM, ciò significa che non basterebbero miliardi di fili. Improponibile sul lato pratico.



Un escamotage per risolvere questo problema è analizzare gli input dei sensori sequenzialmente a gruppi. Nell'auto infatti è difficile che tutti i sensori siano attivi contemporaneamente e che debbano essere controllati nello stesso momento. Dato uno stato si possono quindi selezionare solo le variabili, e quindi le condizioni d'ingresso, che interessano per poter andare avanti.



Controllare i sensori uno ad uno (e quindi selezionando gli ingressi da controllare volta per volta) consente di poter realizzare ROM più piccole e più “espressive”, seppur meno efficienti.

Nel caso qui affianco si parte dallo stato 0, si controlla il bit della temperatura (supponiamo che lo 0 corrisponda a “tutto ok”, 1 a “temperatura anomala”). Verificato che la temperatura è corretta, verrà controllata la pressione e così via.

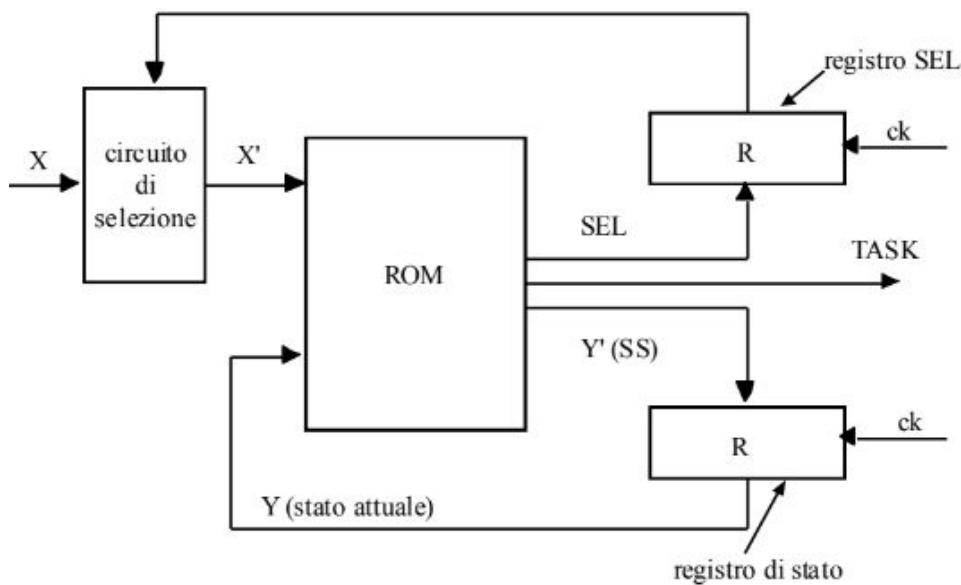
Se abbiamo un clock da 1 Gigahertz, il tempo necessario per passare dallo stato 0 allo stato *nop* successivo è di 1 nsec. È evidente che per raggiungere il quarto *nop* serviranno 4 nsec e così via.

Fintanto che l'utilizzatore del sistema sia un essere umano (come nel caso della macchina) un ritardo di 4 nanosecondi per raggiungere il quarto *nop* è assolutamente trascurabile (i tempi di reazione in auto sono dell'ordine dei millisecondi). Se l'utilizzatore del sistema è un altro sistema che lavora molto velocemente ritardi di questo tipo potrebbero essere eccessivi.

In questo secondo scenario posso prendere ad esempio case a due variabili, così da ridurre il numero di livelli (e quindi il numero di stati e di tempi). Così facendo i tempi si dimezzano. Di nuovo, così facendo la ROM diviene più complessa.

### Struttura del SCO con selezione

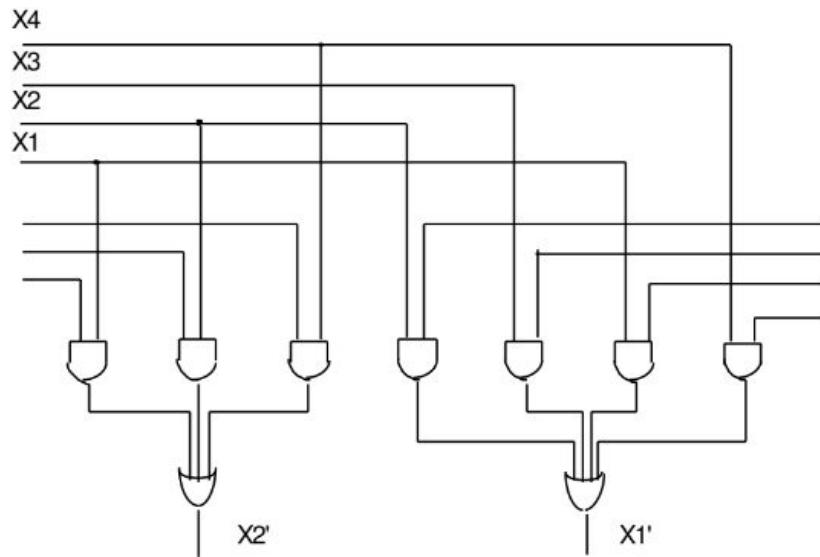
Scelta l'implementazione con una ROM possiamo quindi voler selezionare le variabili di condizione. Questo può essere ottenuto attraverso un circuito di selezione, che può essere implementato ad esempio con un multiplexer.



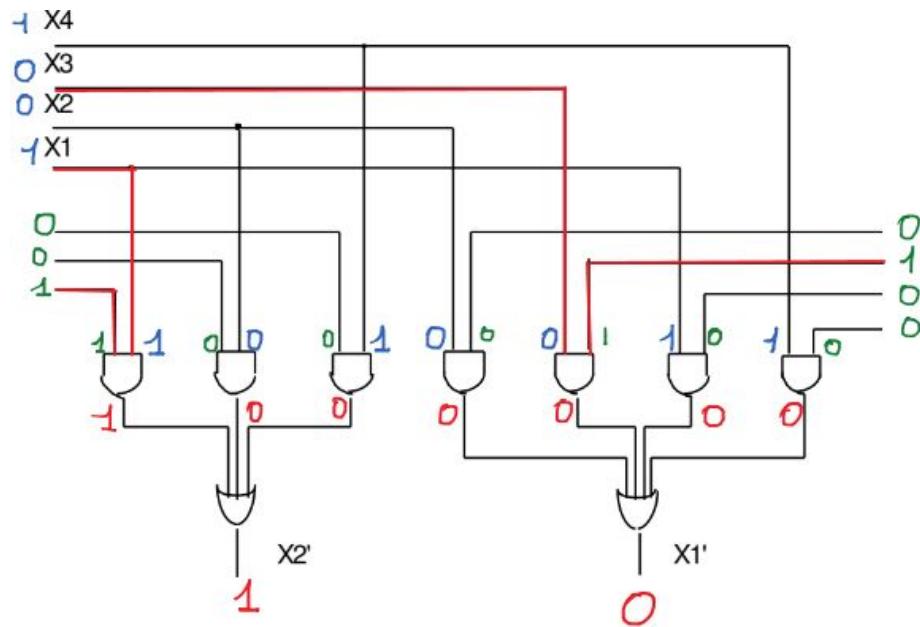
### Circuito di mascheramento non codificato

Nel circuito di mascheramento seguente, date 4 variabili se ne prendono solo due.  $X2'$  può assumere il valore di  $X1$ ,  $X2$ ,  $X3$ , mentre  $X1'$  può assumere il valore di una qualunque delle variabili.

Questo tipo di logica è speculare a quello delle maschere in Assembly.

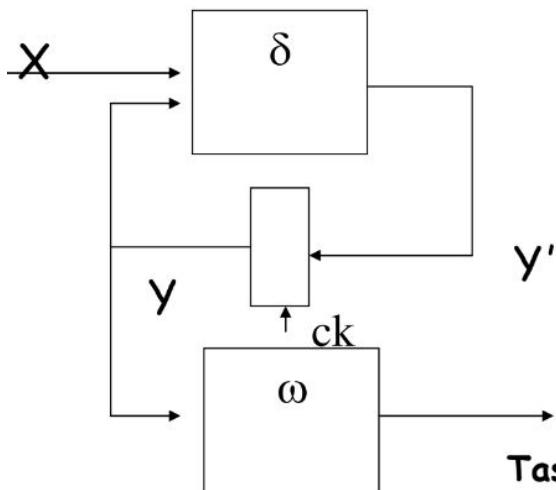


Viene di seguito rappresentato un esempio. Con questa configurazione  $X2'$  e  $X1'$  assumeranno rispettivamente il valore di  $X1$  e  $X3$ , ossia 1 e 0.



### Modello di Moore per il SCO

Nel modello di Moore l'uscita (Task) è relativa allo stato, ossia dato uno stato si ha l'uscita. In funzione delle variabili di condizione si può poi scegliere uno stato successivo.



L'implementazione di questo tipo di macchina può esser fatta con due ROM distinte, una che implementa  $\delta$  e l'altra che implementa  $\omega$ .

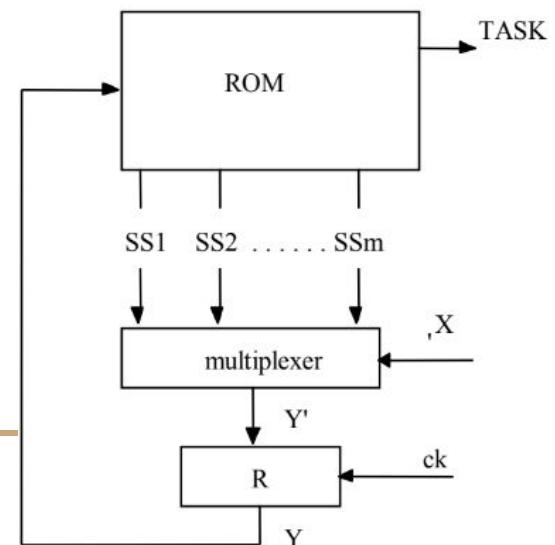
Volendo anche solo prendere in considerazione la prima ROM si può vedere, analogamente a quanto prima mostrato, che il numero di righe sarebbe pari a  $2^{n+m}$ . Sarebbe richiesto, anche in questo caso, un eccessivo numero di fili.

Un'altra soluzione è quella di adottare una sola ROM (come è stato fatto sempre anche durante il corso). In questo caso anziché utilizzare la solita tabella che affianca tutte le possibili combinazioni di stato ed ingresso allo stato successivo e all'uscita, potremmo utilizzare la **rappresentazione matriciale**.

Stato attuale $Y_1\ Y_0$	$X_2$ $X_1$ $X_0$ <b>000</b>	$X_2$ $X_1$ $X_0$ <b>001</b>	$X_2$ $X_1$ $X_0$ <b>010</b>	$X_2$ $X_1$ $X_0$ <b>011</b>	$X_2$ $X_1$ $X_0$ <b>100</b>	$X_2$ $X_1$ $X_0$ <b>101</b>	$X_2$ $X_1$ $X_0$ <b>110</b>	$X_2$ $X_1$ $X_0$ <b>111</b>	Uscita $Z_1\ Z_0$
<b>0 0</b>									
<b>0 1</b>									
<b>1 0</b>									
<b>1 1</b>									

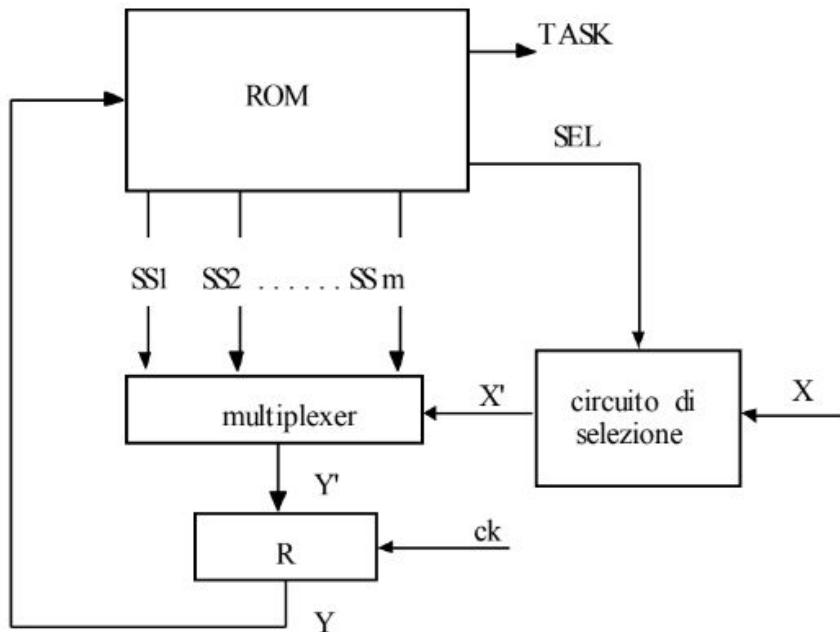
Questa rappresentazione mostra in forma condensata lo stato successivo o l'uscita dato lo stato attuale (prima colonna) e l'ingresso (combinazioni prima riga).

Anche questa seconda soluzione (vedi figura affianco), che fa uso di una sola ROM, non è però facilmente realizzabile. Anche se presente una sola ROM, questa necessiterebbe ugualmente di tante righe e quindi di un gran numero di fili ( $2^m + k$ , con  $k=n$ ).



Si usa un multiplexer per selezionare lo stato successivo, a seconda dell'ingresso.

La soluzione a questo problema ricalca quella adottata per la macchina di Mealy. Si sfrutta un circuito di selezione per prelevare le variabili da selezionare.



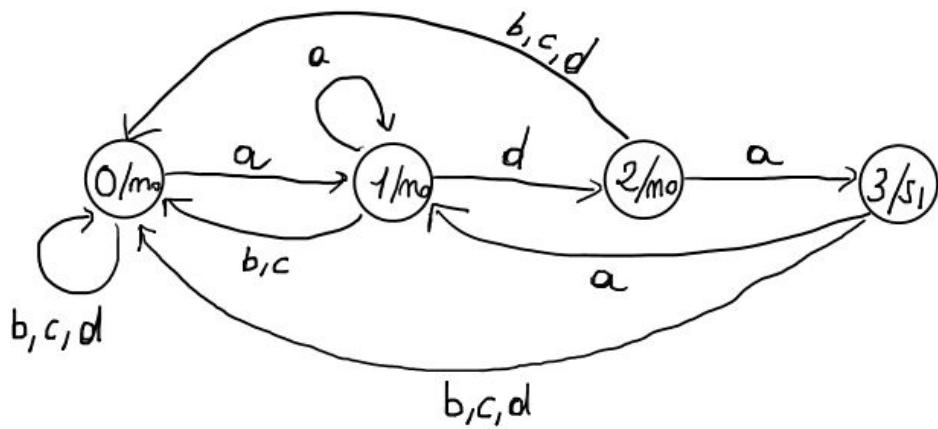
Le microistruzioni, nel caso di Moore, possono essere scritte nella forma:

$$\mu_i : T_1; C_1(\mu_{i1}), C_2(\mu_{i2}), \dots, C_m(\mu_{im})$$

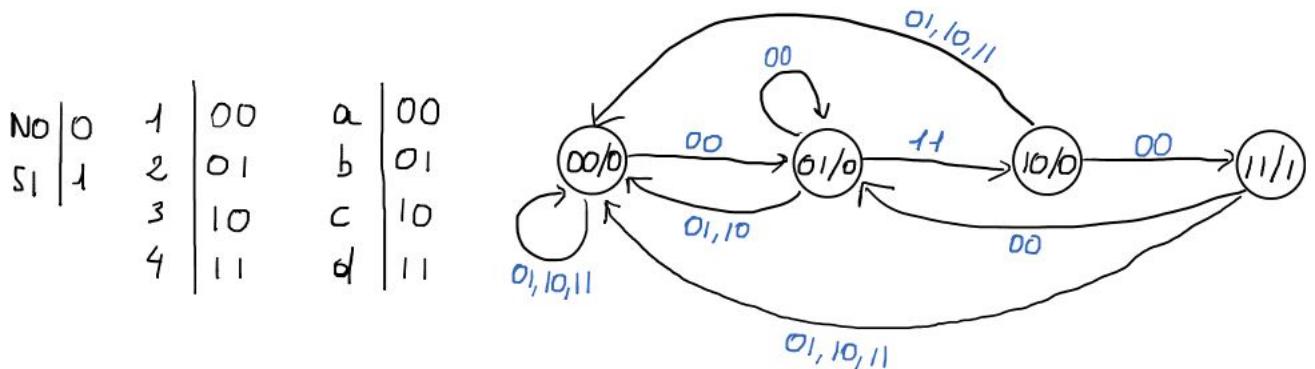
### Esempio

Vogliamo mostrare come, a partire da un diagramma degli stati sia possibile realizzare un microprogramma. Per far ciò prendiamo in analisi il diagramma corrispondente all'automa a stati finiti che riconosce le sequenze di "ada". Verrà quindi riconosciuta una qualsiasi ripetizione della parola "ada", come "adaadaada".

Prima di tutto procediamo con la realizzazione del diagramma degli stati.



Codificando in binario possiamo riscrivere il diagramma nel seguente modo.



A questo punto possiamo procedere con lo scrivere il microprogramma.

00 : 0; 00(01); 01(00); 10(00); 11(00)  
 $\uparrow$        $\uparrow$        $\uparrow$        $\uparrow$        $\uparrow$   
 micro operaz. task condizione stato succ. condizione stato succ. condizione stato succ. condizione stato succ.

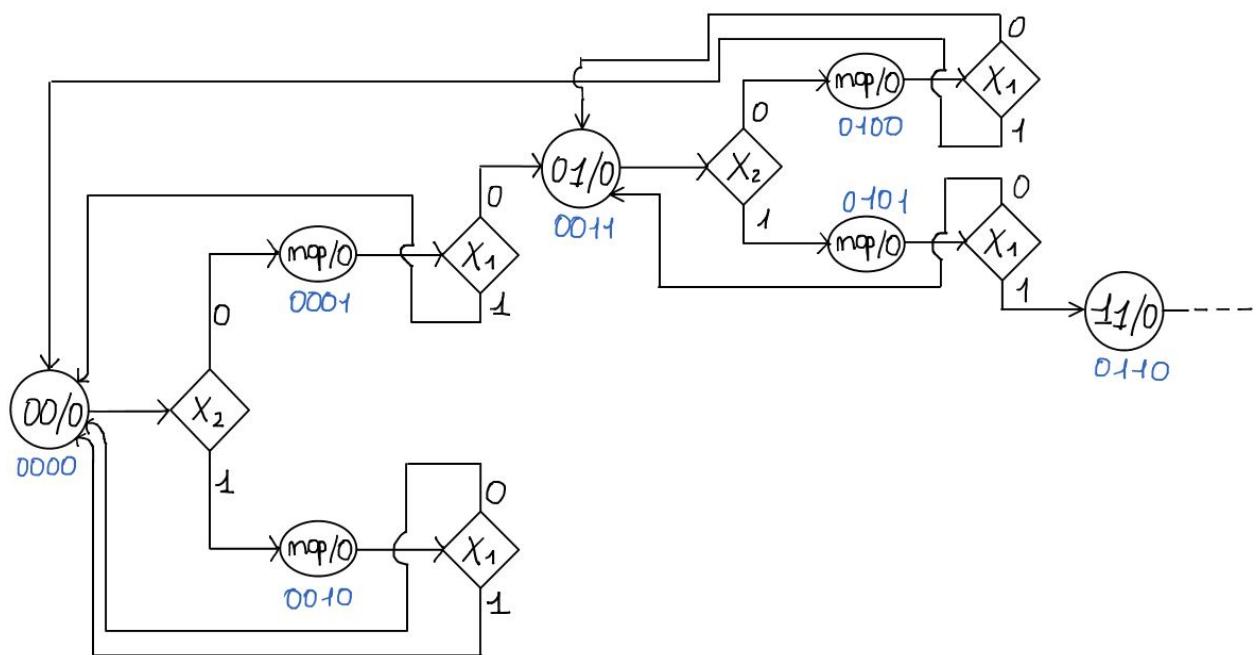
01 : 0; 00(01); 01(00); 10(00); 11(10)

10 : 0; 00(11); 01(00); 10(00); 11(00)

11: 1; 00(01); 01(00); 10(00); 11(00)

Pensiamo alla struttura del SCO nel caso del modello di Moore con circuito di selezione. Per quanto mostrato prima possiamo implementare una singola ROM assieme ad un circuito di selezione che va a definire, volta per volta, quale delle due variabili in ingresso deve essere

verificata. Di seguito è mostrato un diagramma degli stati in cui, appunto si valuta il valore della singola variabile. Partendo dal primo stato e supposto che debba essere valutato  $x_2$  si avrà una biforcazione. Se  $x_2$  è uguale a 0 si arriverà ad uno stato *nop* (no operation) e, a quel punto, si controllerà il valore di  $x_1$ . Se  $x_1$  è pari ad 1 si tornerà nello stato 00, altrimenti si procederà con lo stato 01. In termini pratici non si fa nulla di più di quanto rappresentato nel diagramma precedente, ma questa volta lo si fa valutando una variabile dopo l'altra, alternativamente.



L'introduzione degli stati *nop* ci impone di aumentare il numero di bit necessari per rappresentare l'intero insieme di stati. Supposto che per rappresentare lo stato si usino 4 variabili (scritte in azzurro) e supposto che la variabile di selezione SEL sia pari a 0 nel caso in cui si debba valutare  $x_2$  e pari ad 1 qualora si valuti  $x_1$ , si potrà scrivere il diagramma in **microlinguaggio di tipo 1**.

Le istruzioni avranno il seguente formato:

stato\_partenza : task ; sel , valore\_sel\_0 (stato\_succ) , valore\_sel\_1 (stato\_succ)

---

TASK	SEL	
↓	↓	
0000 : 0 ; 0 , 0 (0001), 1(0010)		
0001 : 0 ; 1 , 0 (0011), 1(0000)		
0010 : 0 ; 1 , 0 (0000), 1(0000)		
0011 : 0 ; 0, 0 (0100), 1 (0101)		
0100 : 0 ; 1 , 0 (0011), 1(0000)		
0101 : 0 ; 1 , 0 (0011), 1 (0110)		
⋮		

Analizziamo la prima riga. Partendo dallo stato 0000 (ove si ha task 0), si valuta la variabile di selezione 0, ossia  $x_2$ . Se tale variabile è pari a 0 (dallo stato 0000) si passa allo stato 0001; se la variabile SEL è pari ad 1 si passa allo stato 0010.

Seconda riga: si parte dallo stato 0001, bisognerà quindi valutare la variabile  $x_1$  (SEL sarà dunque pari ad 1). Se  $x_1$  vale 0 si passa allo stato 0011, se vale 1 allo stato 0000.

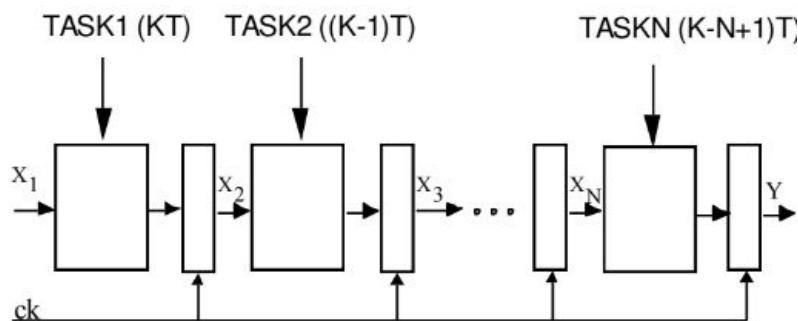
Il ragionamento si itera per tutti gli altri stati. Si noti che ciò che è stato scritto in microlinguaggio di tipo 1 segue esattamente il diagramma realizzato sopra nel caso di case ad una variabile.

### Controllo per strutture Pipeline

La struttura generale di una Pipeline è stata già precedentemente affrontata. Una Pipeline può essere a task fissa o variabile nel tempo. La difficoltà aggiunta nel caso in cui la task sia variabile nel tempo è che si deve avere un modo di seguire le operazioni che vengono svolte in una struttura pipelining mano a mano durante i vari colpi di clock.

Per chiarire questo fatto, immaginiamo di essere in una catena di montaggio automobilistica. Il cliente può scegliere il colore della propria vettura (fra rosso, bianco, celeste), il tessuto degli interni (Velluto, Cotone, Visone) ed il materiale dei cerchi in lega (acciaio, ferro, oro).

È evidente che un cliente potrà procedere con una qualsiasi delle combinazioni possibili e che, in fase di realizzazione, si dovrà fare in modo che le task siano aggiornate in base alla specifica auto da realizzare.



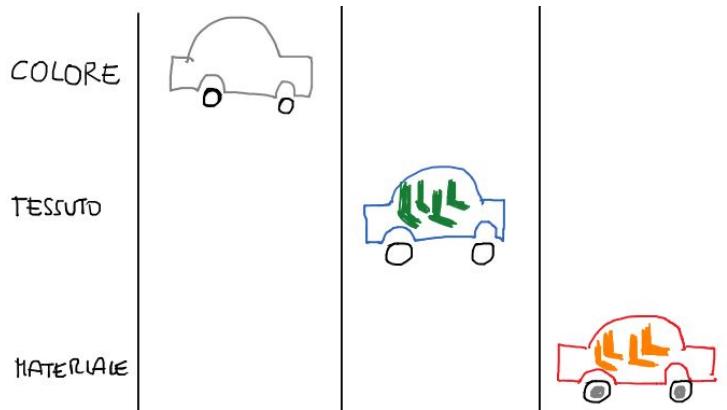
Immaginiamo di voler realizzare un'auto rossa, con interni in velluto arancione e con cerchi in lega d'acciaio, una seconda auto celeste con tessuti in velluto e cerchi in oro.

COLORE			
TESSUTO			
MATERIALE			

Se inizialmente il colore è rosso, con la fabbricazione della seconda macchina sarà celeste.

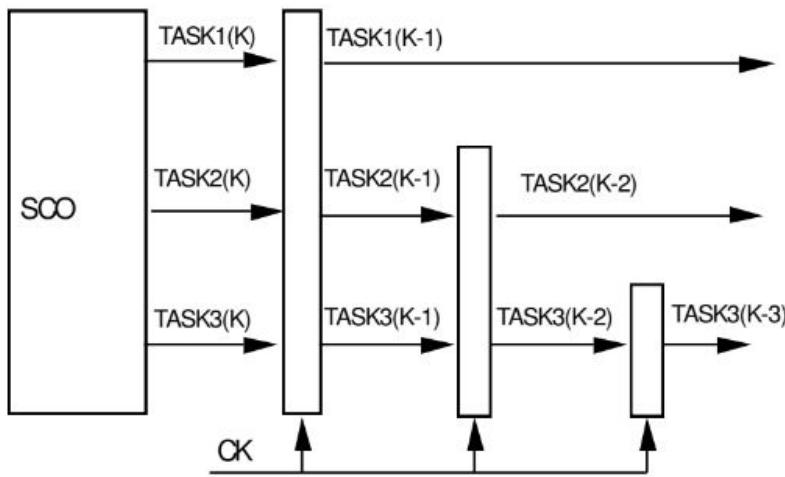
COLORE			
TESSUTO			
MATERIALE			

Al terzo colpo di clock il tessuto da applicare sarà il velluto verde e non più il cotone arancione.



Quindi, in funzione dell'auto su cui dobbiamo lavorare ad ogni "step" dovremo scegliere cosa fare.

Nel caso di una struttura Pipeline il "cosa fare" dovrà essere determinato dal sottosistema di controllo. Per impartire "ordini" corretti bisognerà fare in modo che le Task successive alla prima siano mantenute nei successivi colpi di clock per permettere ai singoli moduli della Pipeline di sapere esattamente cosa fare.



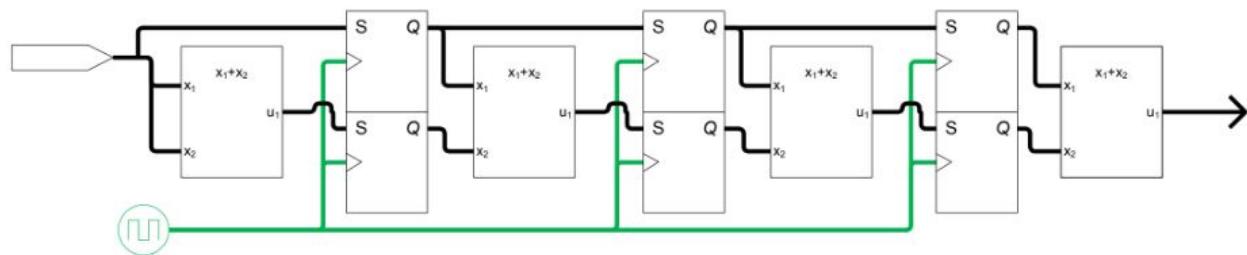
Si utilizza dunque una struttura di questo tipo. Le task vengono inizialmente inserite in un registro. La prima task sarà quindi impartita alla prima rete combinatoria, mentre le successive saranno nuovamente salvate in un altro registro. Al successivo colpo di clock la

seconda istruzione sarà impartita e le successive nuovamente salvate in un altro registro e così via.

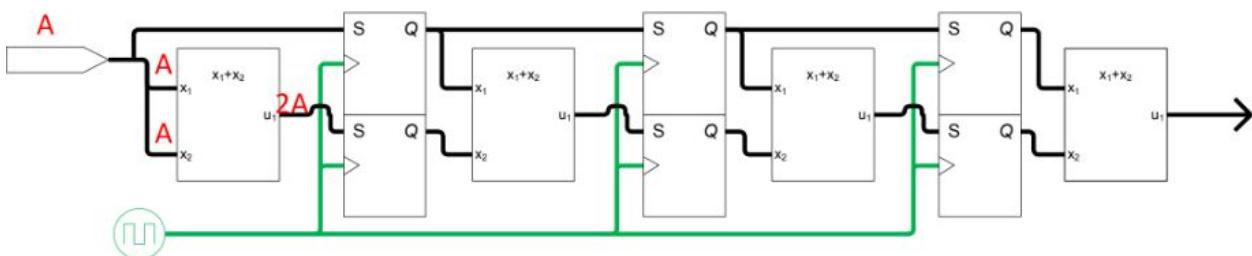
In questo modo le istruzioni specifiche da eseguire dopo  $n$  colpi di clock vengono sfalsate nel tempo e rese disponibili nel momento in cui servono.

## Esempio di Pipeline

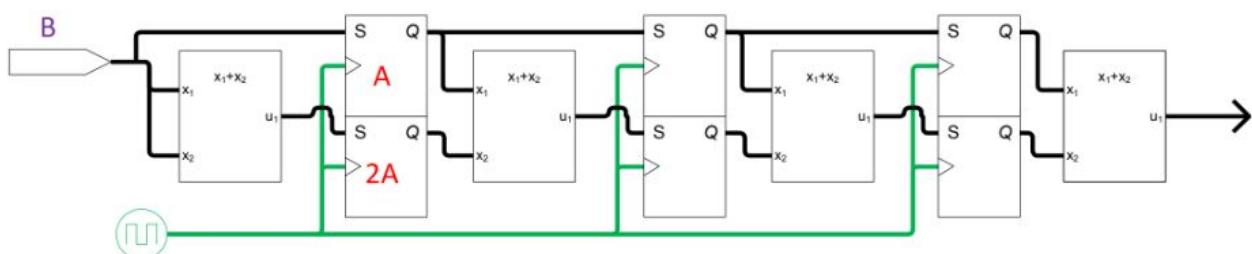
Immaginiamo di voler implementare con una pipeline un moltiplicatore  $5 \times n$  attraverso cinque addizionatori che sommano lo stesso numero per cinque volte consecutive.



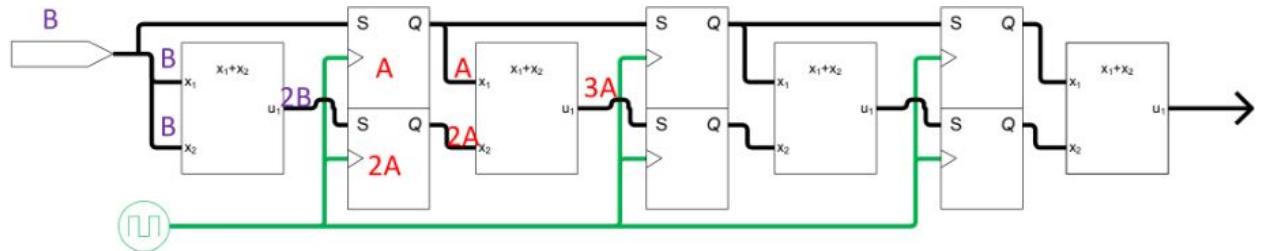
In una serie di registri si conservano e si rilasciano, al successivo colpo di clock, sia il valore di partenza (che andrà ri-sommato) sia il risultato della precedente addizione.



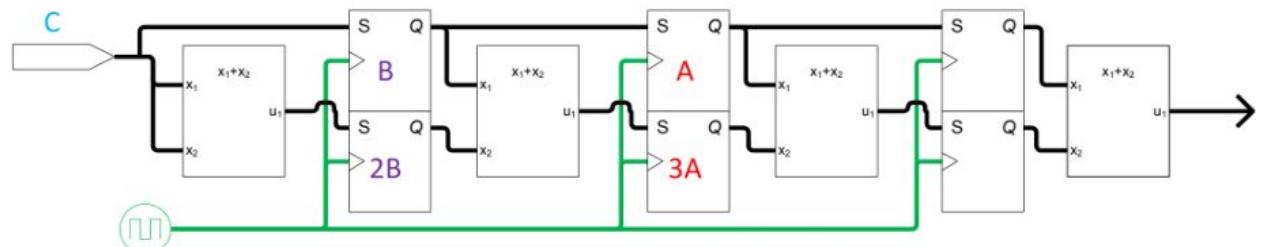
Inizialmente si presenta in input il dato da moltiplicare, che verrà dato come valore per entrambi gli operandi dell'addizionatore. Inoltre, come mostrato nella seguente immagine, A viene anche salvato nel primo registro, insieme al risultato dell'addizionatore (2A).



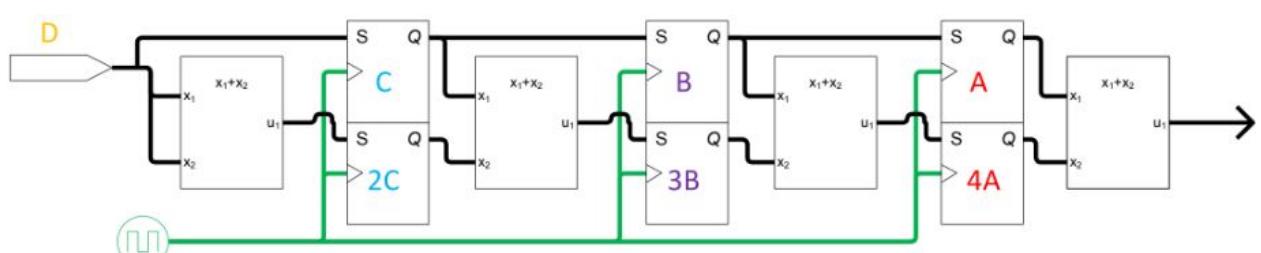
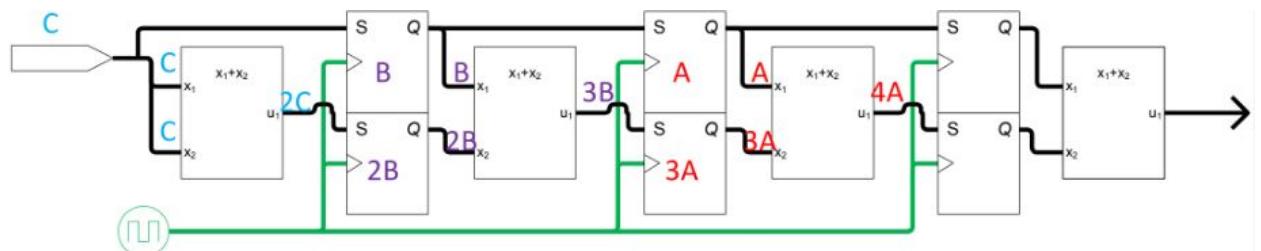
Potrà quindi essere posto in input un altro dato da moltiplicare, mentre A e 2A saranno presi come operandi del secondo addizionatore.

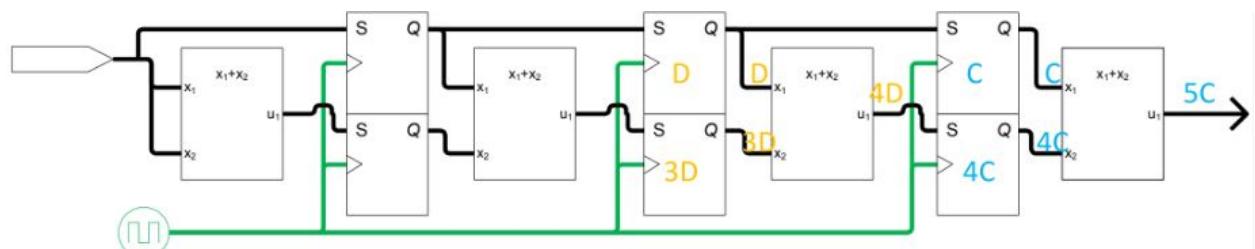
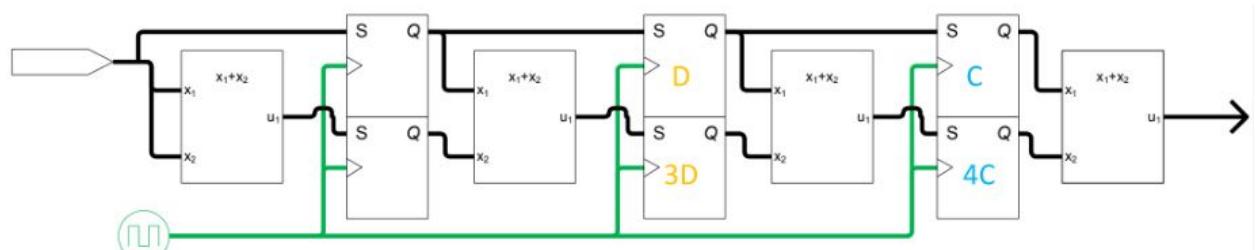
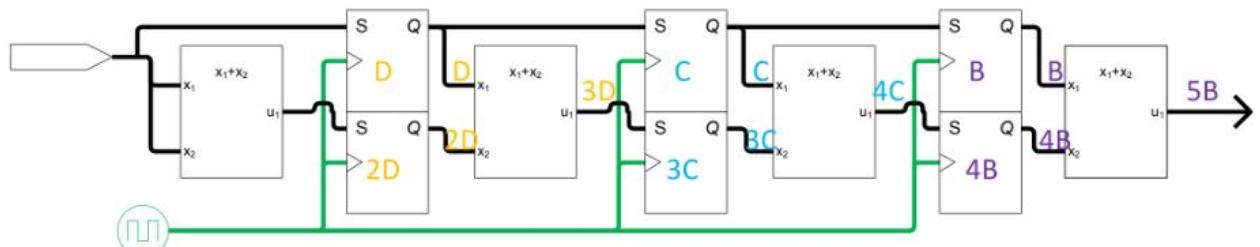
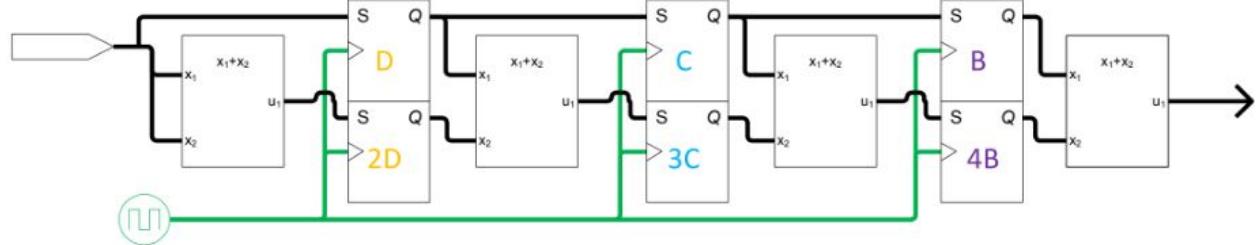
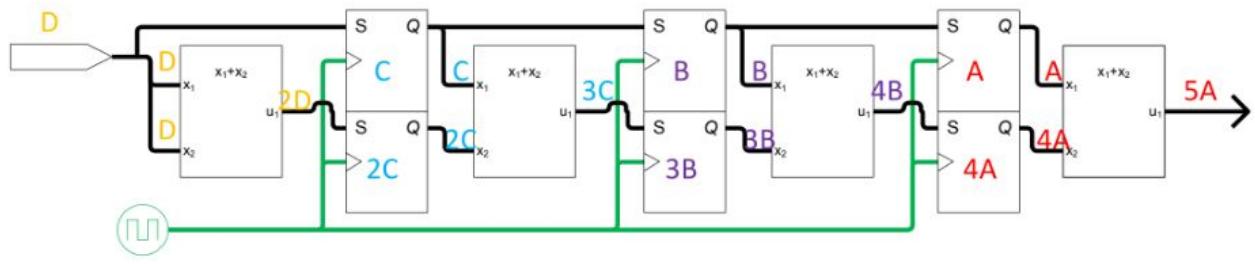


A questo punto nel primo registro si conserveranno A e 2B, mentre il secondo conterrà A e 3A. Potrà a questo punto essere caricato anche un terzo elemento C da moltiplicare.



Le operazioni si ripeteranno analogamente a quanto detto ad ogni colpo di clock.





Da notare come il dato prodotto è sempre 5 volte quello iniziale. Proseguendo verrà prodotto anche 5D (gli step sono omessi per motivi di spazio - file: AC-11b-esempio-pipeline)

---

Grazie al Pipelining, come già detto, migliora il throughput (numero di operazioni eseguibili per unità di tempo).

## Sistemi con molti micropogrammi

Precedentemente nell'esempio era stato realizzato un sistema in grado di riconoscere la sequenza "ada", cioè si è andato a realizzare macchine sequenziali specializzate in un'operazione.

Potrei però realizzare un sistema in grado di riconoscere diverse sequenze, come "ada", "gamma", "delta", "giovanni" e capace di implementare di una macchina sequenziale piuttosto che un'altra.

Anche nel caso del sottosistema di controllo dei processori odierni ogni volta si esegue un'istruzione si fa questa attività. La prima operazione, uguale per tutte le istruzioni, che viene eseguita dal sottosistema di controllo è la fase di **fetch**. Viene caricata dalla memoria l'istruzione da eseguire e viene caricata nell'Instruction Register. Poi, in funzione dell'OPCODE, si fa una cosa piuttosta che un'altra.

Supponiamo che si debbano riconoscere le seguenti sequenze ciascuna caratterizzata da macchine sequenziale di n stati:

- *iniziale* → 5 stati (\*)
- ada → 4 stati
- gamma → 6 stati
- delta → 6 stati
- giovanni → 8 stati

(\*) In più c'è anche da scrivere la macchina sequenziale che mi dice quale sequenza devo riconoscere. Questa macchina la chiamiamo "iniziale" e supponiamo sia costituita da 5 stati.

Il sottosistema di controllo viene organizzato in **pagine**. Le pagine sono numerabili, sono modulari ed hanno la stessa dimensione (size). Ogni pagina conterrà una macchina sequenziale e questo farà sì che le pagine dovranno avere size almeno uguale alla size necessaria dalla macchina più grande.

---

Ogni pagina sarà poi provvista di righe, utili per trovare l'informazione necessaria nella pagina. Il numero di righe dipenderà ovviamente dalla size delle pagine e quindi dalla dimensione delle macchine che devono essere implementate.

000	000 111	5/8
001	000 111	4/8
010	000 111	6/8
011	000 111	6/8
100	000 111	8/8

Nel caso d'esempio la macchina più "grande" ha 8 stati e quindi si avranno pagine da 8 righe ciascuna.

Non tutte le pagine saranno occupate pienamente, non essendo tutte le macchine della stessa grandezza.

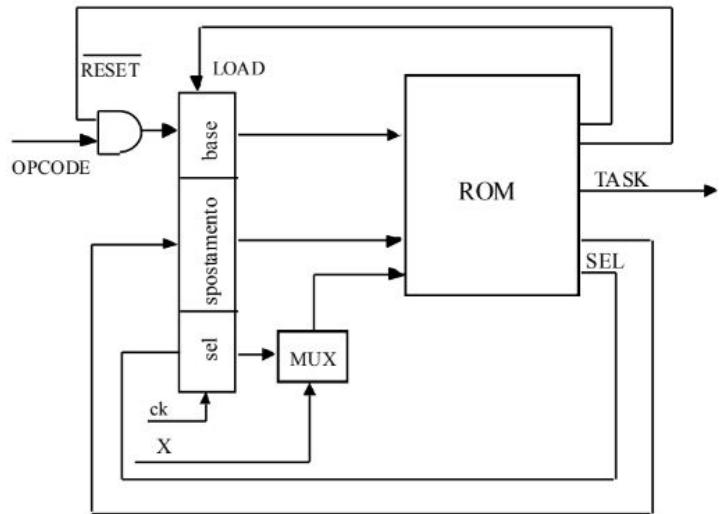
La macchina "iniziale" occuperà infatti 5 righe su 8, "ada" occuperà 4 righe su 8 e così via.

Quindi, in questo caso, avremmo avuto bisogno di  $2^3$  righe per ciascuna pagina. La ROM che dovrà essere utilizzata dovrà avere almeno 5 pagine, quindi utilizzeremo una ROM di  $2^3$  pagine ( minimo multiplo di 2 che include il 5). In totale avremo  $2^6$  righe.

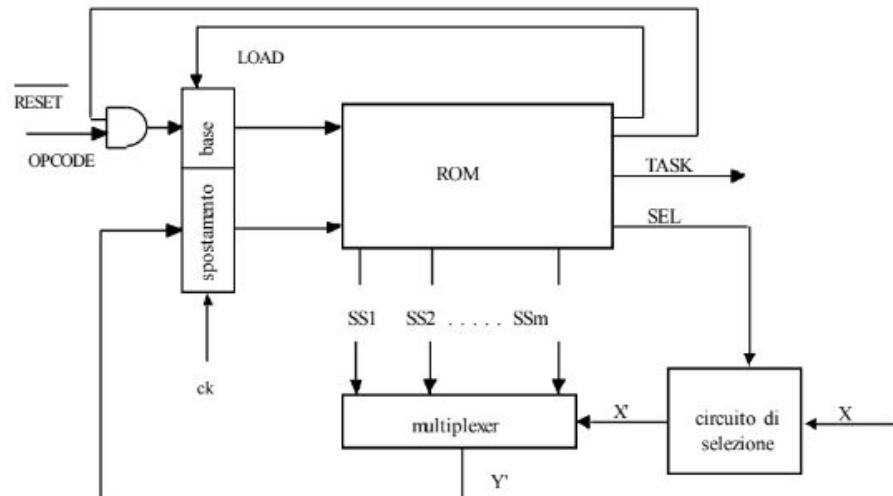
Quindi, non solo non verranno occupate tutte le righe di ogni pagina, ma ci saranno anche delle pagine vuote. Si utilizzeranno infatti solo 5 pagine (numerate da 000 a 100) sulle 8 totali disponibili (rimarranno dunque vuole le pagine numerate da 101 a 111).

L'ultima microistruzione presente in ciascuna pagina indica di dover tornare a far fetch.

Alla luce di questi fatti alla ROM andrà .



Schema di SCO con molti microprogrammi (modello di Mealy)



Schema di SCO con molti microprogrammi (modello di Moore)

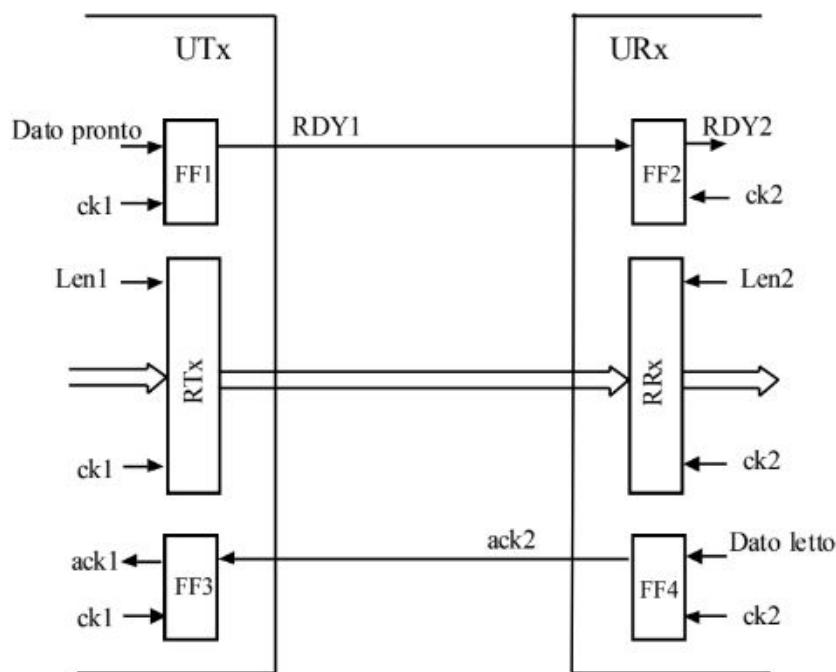
## Comunicazione fra due sistemi digitali

Fino a questo punto si è parlato unicamente di sistemi LLC (Level Level Clocked) che interagiscono fra loro, ove il clock viene dimensionato tenendo conto dei tempi dei singoli sistemi.

La connessione secondo queste modalità di un sistema molto veloce con uno più lento fa però sì che quello veloce sia costretto a lavorare a frequenze ridotte per stare allo stesso passo del sistema più lento. È evidente che si introducono delle limitazioni molto importanti, soprattutto quando i due sistemi hanno frequenze di ordini di grandezza diversi.

Immaginiamo quindi di avere due sistemi digitali complessi indipendenti, che lavorano ciascuno con il proprio clock. Così facendo nessuno è limitato dall'altro, ma sarà necessario stabilire un protocollo per consentir loro di comunicare (si parla anche di **protocollo di Handshaking** o protocollo di comportamento).

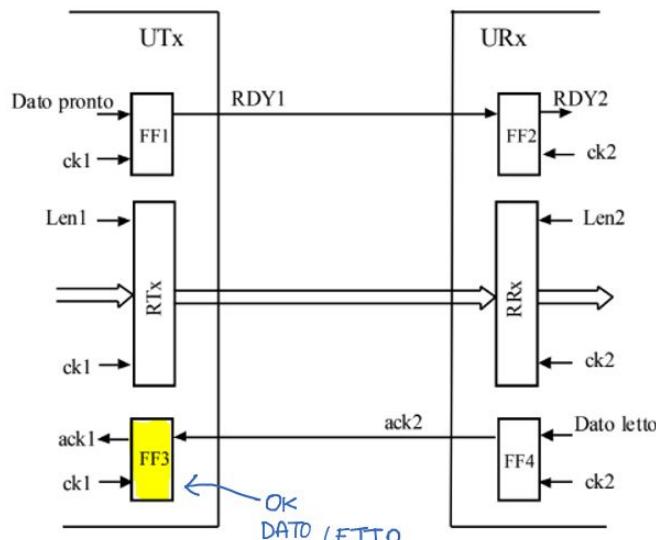
Quella che segue è l'interfaccia tipica di due sistemi che si devono scambiare informazioni.



Si ha un registro in cui si deve registrare l'informazione da parte della sorgente, un registro in cui il destinatario si va a leggere l'informazione ed una serie di flip-flop.

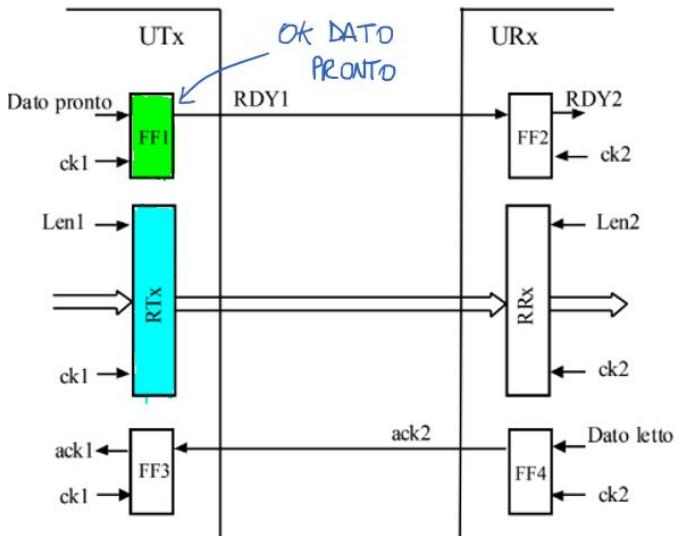
Questo circuito non solo è adatto per far comunicare sistemi digitali a frequenze differenti, ma anche a tensioni di alimentazione differenti (cioè con voltaggi diversi fra i due sistemi). I registri presentano degli adattatori di impedenza, per permettere di interfacciarsi appunto con informazione fornita con voltaggi distinti.

Il funzionamento di questo sistema di connessione è piuttosto semplice e si basa sul concetto di semaforo.

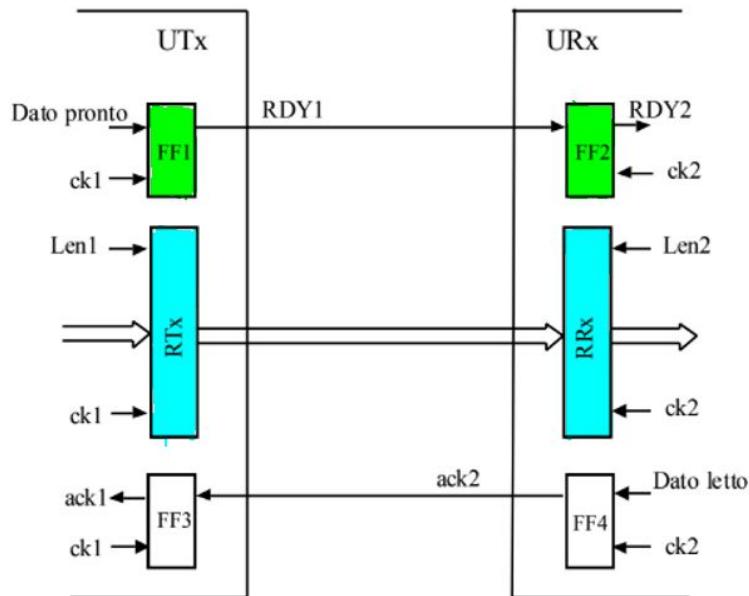


Innanzitutto il produttore del dato, prima di poter scrivere sul suo registro, verifica se l'altro sistema ha prelevato l'informazione precedentemente inserita nel registro (per evitare di sovrascriverla). Viene quindi verificato il contenuto del flip-flop di dato letto.

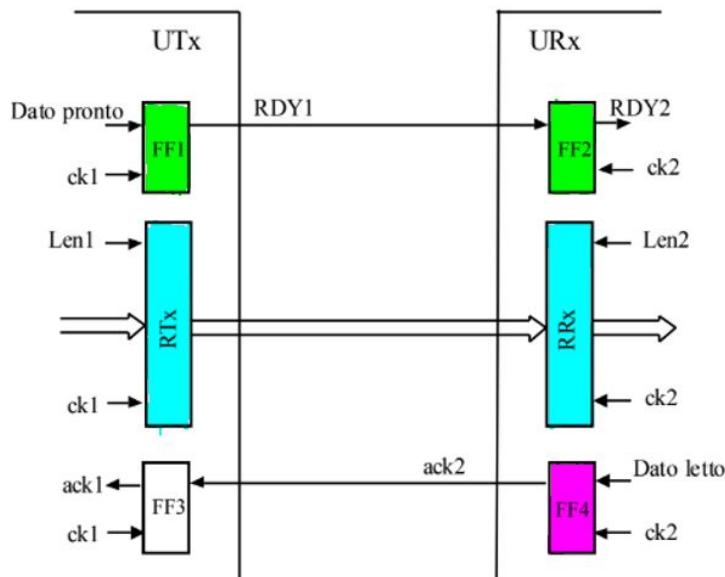
Se questo è vero (e cioè se è stata prelevata - e quindi se il flip-flop di dato letto è settato), il produttore scrive la nuova informazione nel suo registro e da il "segnale" di dato pronto.



A questo punto il ricevente, con la sua velocità, va a campionare l'uscita del registro di dato pronto e, essendosi accorto che un nuovo dato può essere letto, lo va a leggere.



Infine, commuta il suo flip-flop di dato letto, così che il produttore, dopo averlo campionato, possa procedere con l'invio di un nuovo dato.



Il comportamento del produttore e del consumatore

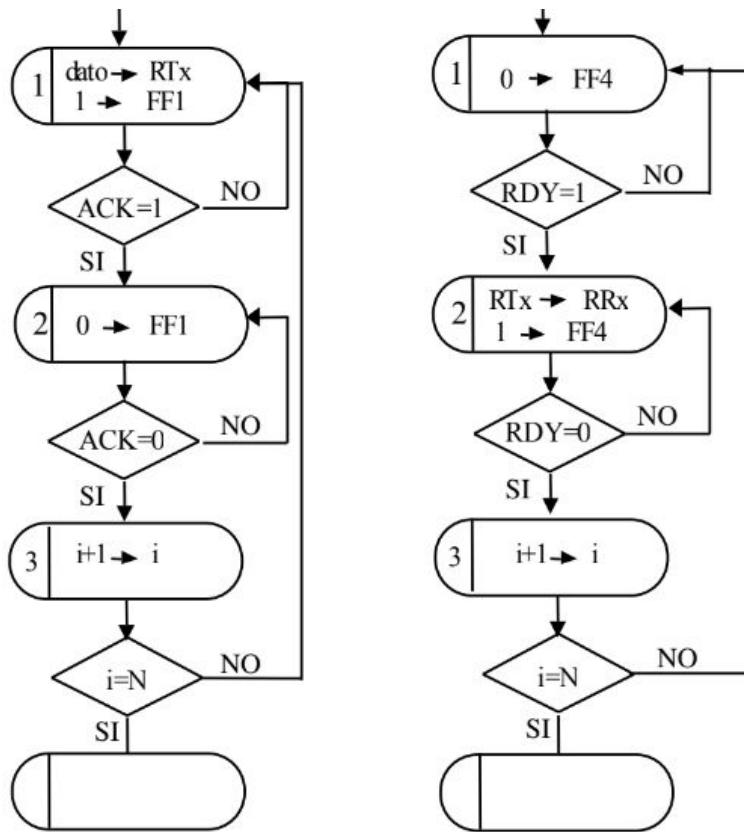
- UTx**
- 1: dato  $\rightarrow$  RTx, 1  $\rightarrow$  FF1;
  - 2: if ack1=0, then vai a 2;
  - 3: 0  $\rightarrow$  FF1;
  - 4: if ack1=1, then vai a 4;
  - 5: i+1  $\rightarrow$  i;
  - 6: if i  $\neq$  N, vai a 1

**URx**

- 1: 0  $\rightarrow$  FF4;
- 2: if RDY2=0, then vai a 2;
- 3: RTx  $\rightarrow$  RRx, 1  $\rightarrow$  FF4;
- 4: if RDY2=1, vai a 4 ;
- 5: i+1  $\rightarrow$  i;
- 6: if i  $\neq$  N, vai a 1

viene specificato con un protocollo, che può essere così espresso (cioè come una sequenza di "passi" che i due sistemi devono svolgere) o che può esser visto come macchina sequenziale.

Possiamo anche esprimere delle sequenze di microistruzioni eseguite dai due sistemi durante il protocollo di comunicazione, mostrate qui di seguito con un diagramma degli stati.

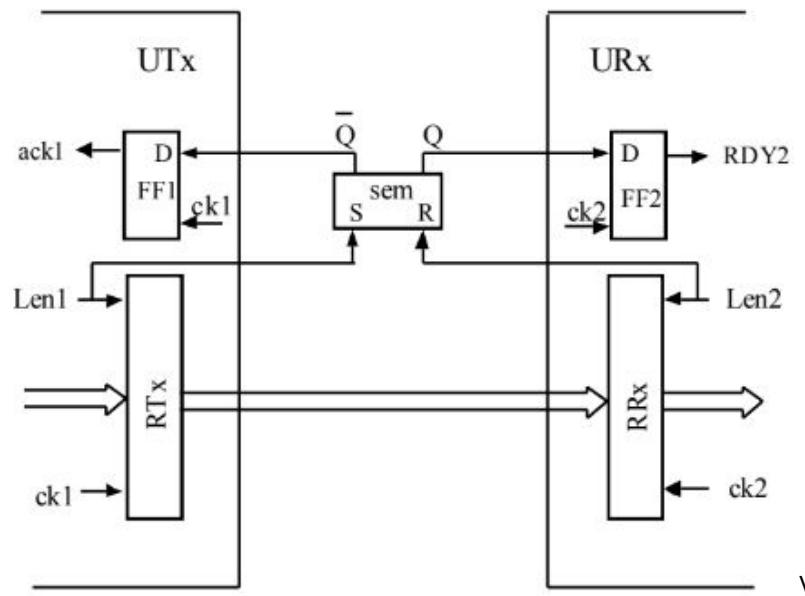


### Sincronizzazione tra due unità in comunicazione mediante flip-flop di semaforo

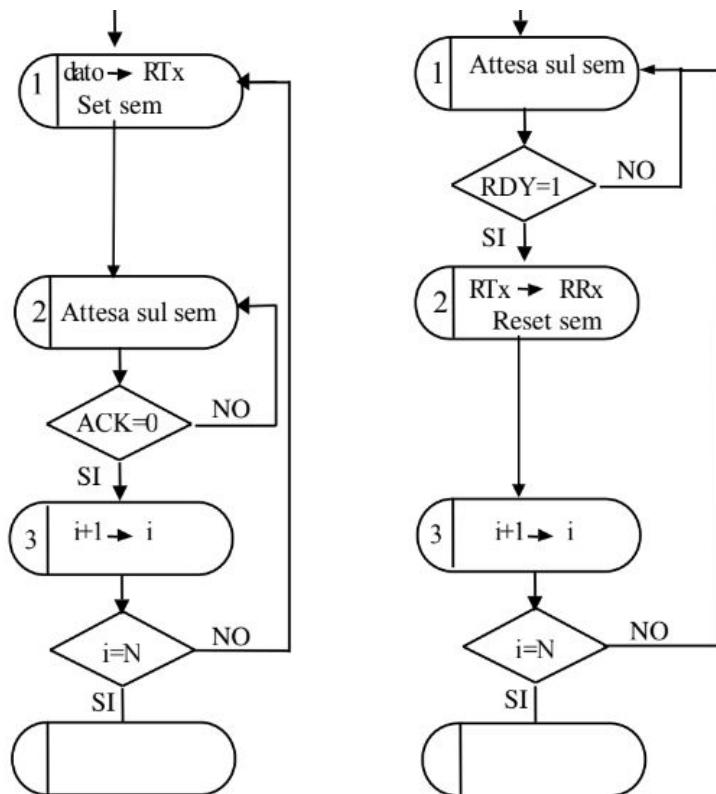
La struttura prima rappresentata può essere ulteriormente semplificata sfruttando un flip-flop di semaforo.

Questo flip-flop ha due segnali di Set e di Reset in ingresso. Quando il produttore scrive il dato il flip-flop di semaforo viene settato (ad 1). Q negato vale 0 e quindi il produttore recepisce che non può caricare altri dati, mentre il ricevente *riceve* Q, ossia 1 e quindi può

procedere col prelevare il dato. Una volta prelevato farà il reset del semaforo, dando la possibilità al produttore di scrivere un nuovo dato nel registro.



Ciò può essere scritto anche nel seguente modo.



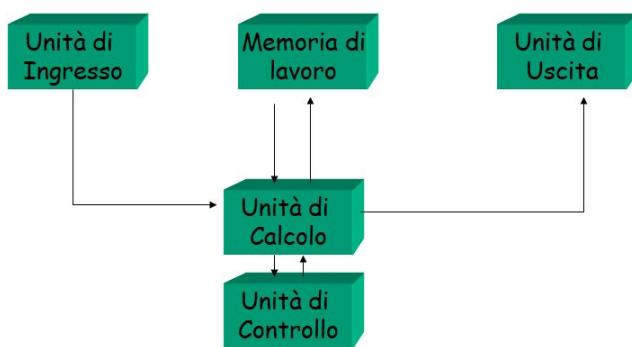
# Il processore z64

NOTA: Molte delle informazioni qui presenti sono riprese dal modulo del professor Pellegrini.

## La macchina di Von Neumann

Il primo processore con sistema programmabile è stato realizzato, a livello industriale, nell'800 in Inghilterra per comandare i telai nell'industria tessile. In questi contesti si usavano delle schede perforate.

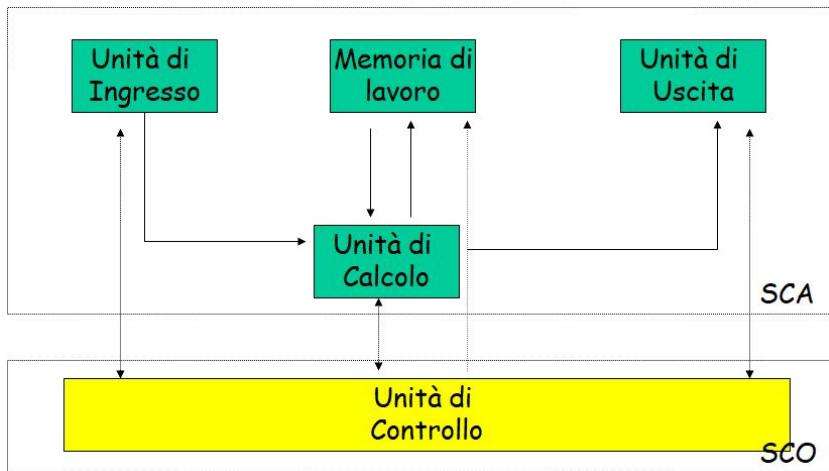
Il processore che andremo ad implementare si basa sulla macchina di Von Neumann, che fu sviluppata negli anni 40 per scopi militari (ed in particolare per calcolare la traiettoria dei proiettili nemici).



L'unità di Calcolo lavora con la Memoria di lavoro in cui sono presenti sia i dati che le istruzioni. Questo aspetto segna un grosso passo in avanti. In precedenza infatti si usano, come detto, delle schede perforate che erano a tutti gli effetti "esterne" rispetto alla memoria del sistema. Von Neumann invece pensò di prendere i programmi come input e di metterli in memoria, così che potessero essere riutilizzati dal sistema. Questo perché la memoria di lavoro può essere mandata alla velocità più prossima a quella del processore.

## Suddivisione SCA-SCO

Nella nostra accezione di sistema digitale complesso c'è poi una perfetta suddivisione fra sottosistema di calcolo e sottosistema di controllo.

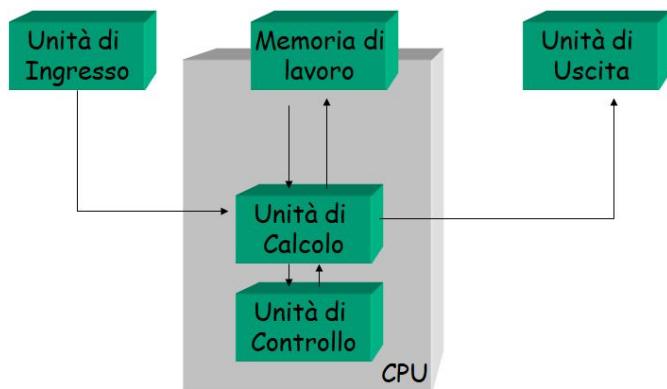


### Tipologie di memoria: accenni

Le tipologie di memoria sono sostanzialmente due: la memoria statica e la memoria dinamica.

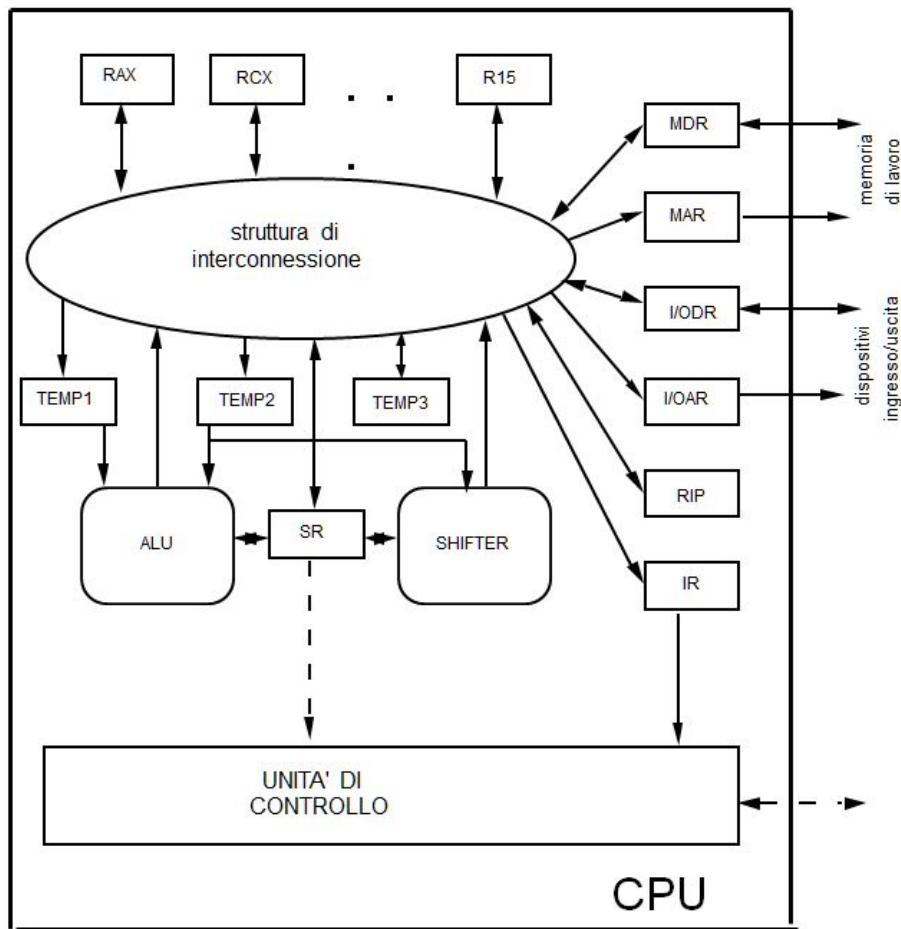
La memoria dinamica è quella utilizzata come memoria nel sistema, per i suoi costi ridotti. Mentre la memoria statica, assai più costosa, è al massimo utilizzata per i registri e per alcune tipologie di Cache.

Dato che l'unità di calcolo è molto veloce si va a mettere un po' di memoria il più possibilmente veloce e vicina all'unità di calcolo e di controllo ed una restante parte che interagisce con la restante parte del sistema.



## z64: struttura generale

In un processore a 64bit di tipo Intel, come lo z64, si hanno 32 registri, di cui solo alcuni visibili all'utente (nel modulo del professor Pellegrini si forniscono maggiori dettagli). Si hanno anche registri temporanei, dei registri di "comodo", come lo status register, il program counter e l'instruction register, e registri di interfaccia col mondo esterno.



## z64: Sottosistema di Calcolo (SCA)

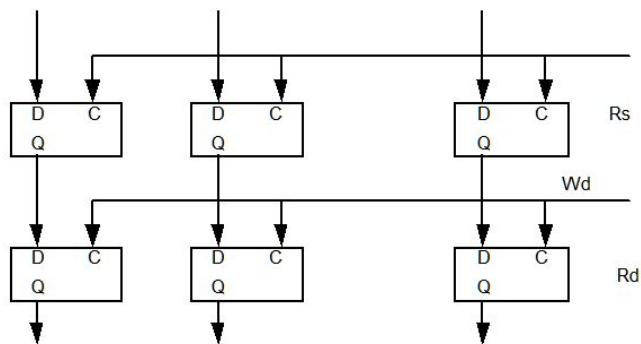
Per il SCA verranno usati:

- Registri (basati su Flip-Flop D con segnale di Enable)
  - speciali
  - generali
- Dispositivi di calcolo

- Shifter
- ALU (somma e sottrazione)
- Multiplexer
- Decodificatori
- Strutture di interconnessione (BUS)

Le architetture RISC supportano pochi accessi in memoria, mentre quelle CISC supportano accessi multipli in memoria. [Da riscrivere]

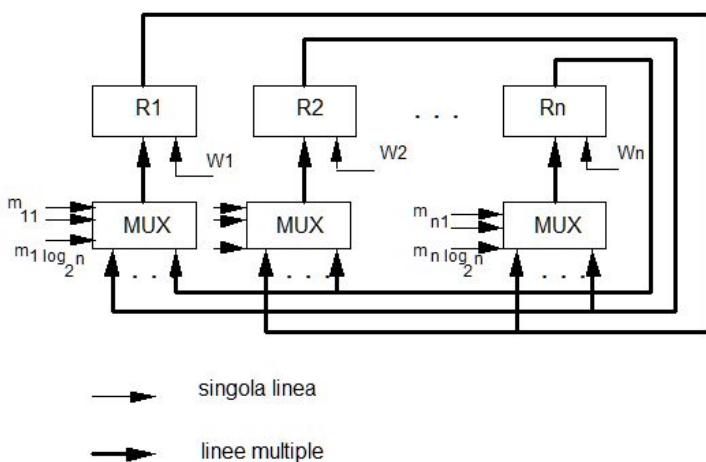
### Interconnessione diretta tra registri



Vantaggi: semplice da implementare

Svantaggi: difficoltà di interconnettere più registri

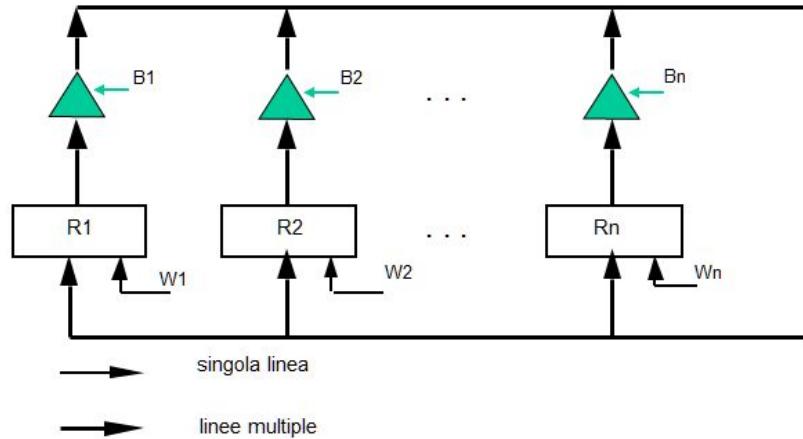
### Interconnessione tramite multiplexer



Vantaggi: semplice da implementare; possibilità di trasferire più dati contemporaneamente

Svantaggi: costo eccessivo dei multiplexer e delle linee di interconnessione

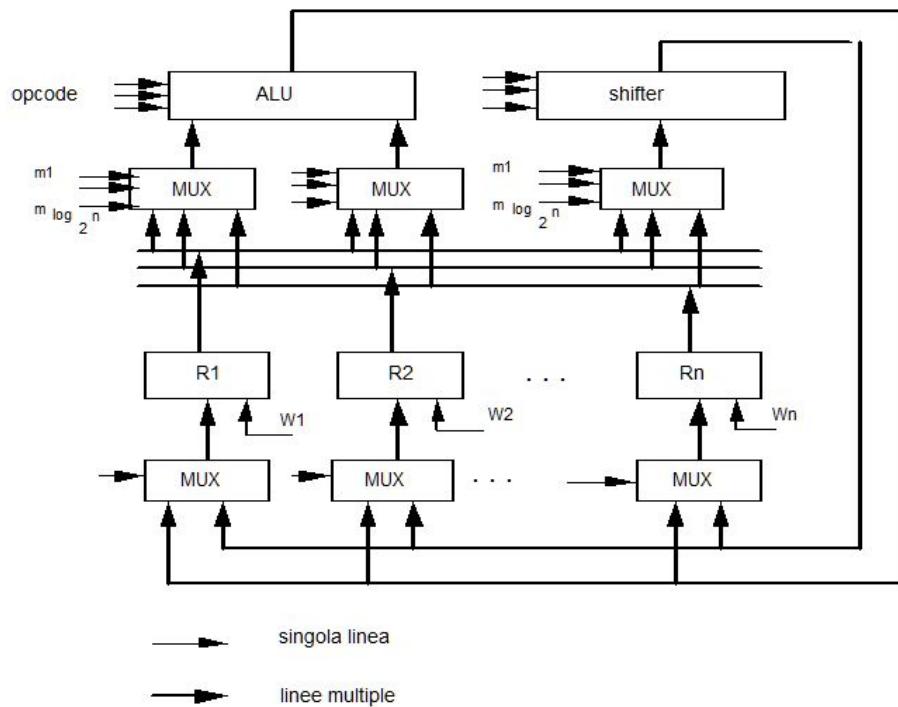
### Interconnessione tramite bus



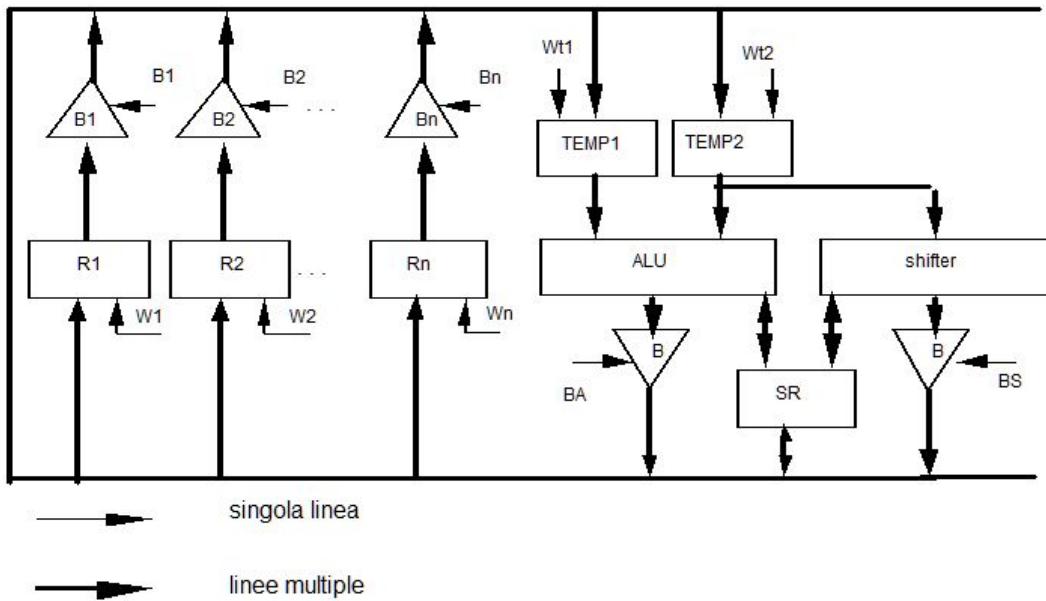
Vantaggi: semplice da implementare; costo contenuto

Svantaggi: è possibile un solo trasferimento alla volta; eccessivo numero di segnali di controllo

### Trasferimento dati tra registri e circuiti di calcolo (con multiplexer)



## Trasferimento dati tra registri e circuiti di calcolo (con singolo BUS)



Questa soluzione, già affrontata prima, non è particolarmente costosa ed ottima in molti contesti seppur, essendoci solo un bus, una sola informazione alla volta può essere trasmessa.

### Esempio 1

Scriviamo una macchina a stati finiti che permette il trasferimento della somma di  $R_1$  e  $R_2$  in  $R_n$ .

$$R_1 + R_2 \rightarrow R_n$$



Inizialmente verranno abilitati  $B_1$  e  $W_{t1}$ , così facendo il dato contenuto in  $R_1$  verrà scritto nel registro  $Temp1$ . Stabiliamo una nomenclatura per cui ad essere scritti sono solo le variabili che sono significative ad ogni stato e che cioè devono essere settate ad 1 ( $B_1$  e  $W_{t1}$  per il primo stato). Dopo un colpo di clock si passerà allo stato successivo e verranno abilitati anche  $B_2$  e  $W_{t2}$ . Poi, dopo un ulteriore colpo di clock si passerà allo stato 2 in cui ad

essere abilitata è la ALU per fare la somma (ipotizziamo di utilizzare il codice operativo ALUsomma). La ALU lavorerà per un certo numero di colpi di clock, noti a priori a seconda di come è implementato l'addizionatore. Dopodiché si passa allo stato 3 in cui si settano BA e Wn.

## Esempio 2

Ipotizziamo ora di fare  $R2 + R2 \rightarrow R1$

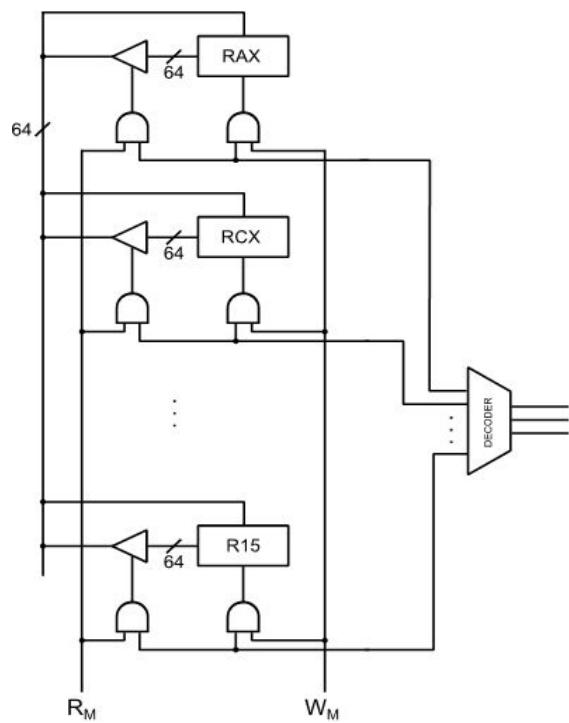
$$R2 + R2 \rightarrow R1$$



Vediamo come la struttura sia praticamente equivalente.

Questa soluzione dimostra come sia possibile declinare questo diagramma per rispondere alla specifica operazione da eseguire. Questa implementazione è però difficile da realizzare perché presupporrebbe di avere tante variabili (e quindi tanti fili).

L'implementazione più conveniente è quella di un banco di registro con due soli fili per la scrittura e la lettura e di un decoder che abilita un particolare registro a scrivere o leggere.



(Scrivi macchina a stati finiti generale)

---

## NOTE

Questi appunti sono ancora incompleti e potrebbero contenere numerosi errori. ~~Cercherò di integrare ciò che manca e di correggere gli errori dopo la sessione estiva (supponendo di riuscire a superarla indenne ;)~~ Purtroppo la magistrale mi sta risucchiando ogni possibile minuto di tempo libero, non credo che riuscirò a terminare il lavoro. Qualora qualche giovane studente volesse contribuire all'opera, può contattarmi su Telegram (@chrismar97).

Futuri aggiornamenti saranno pubblicati nell'apposita repository GitHub

<https://github.com/chrismarinoni/architetture-dei-calcolatori-ciciani-sapienza>



Questi appunti sono distribuiti con Licenza [Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale.](#)